On Quantum Data Structures

MSc Thesis *(Afstudeerscriptie)*

written by

Daan Schoneveld
(born August 6th, 2001 in Hoorn, Netherlands)

under the supervision of **prof. dr. Stacey Jeffery** and **dr. Subhasree Patro**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
July 4th, 2024	dr. Malvin Gattinger (chair)
	prof. dr. Stacey Jeffery
	prof. dr. Ronald de Wolf
	dr. Subhasree Patro
	dr. Ronald de Haan
	dr. Māris Ozols

Institute for Logic, Language and Computation

Abstract

Data structures serve as fundamental building blocks in classical computing, allowing for efficient ways of organising, storing and manipulating data. To develop certain time-efficient quantum algorithms, classical data structures must be translated to the quantum context. However, several challenges emerge in this translation, such as uniqueness of representation, uniqueness of memory and the worst-case limitation problem. In this thesis we explore the field of quantum data structures, and their limitations, in the context of element distinctness. Our results are presented in a (what we call) Quantum Word Random Access Machine (QWRAM) model, extending that of [BLPS22], by quantifying both classical *and* quantum elementary word operations as a variable γ.

The main theorem is a black-box quantum algorithm for element distinctness with time complexity $O(N^{2/3}(\gamma + T_L + T_{ID} + T_C) + T_{\text{init}})$, for *any* quantum data structure that can lookup elements in time T_L, insert and delete elements in time T_{ID}, check for collisions in time T_C and initialise the data structure in time T_{init}. This theorem eliminates the need for an explicit diffusion operator, previously thought required from our quantum data structure on top of the other operations. As a result, designing quantum data structures becomes considerably simpler. For this proof we introduce a new formal definition of quantum data structure and a new kind of graph, called the *permuted Johnson graph*.

Using our theorem, we can slightly simplify two existing quantum data structures and reanalyse their time complexity in the QWRAM model: Ambainis' quantum skip list [Amb03] and the quantum radix tree [BJLM13, Jef14, BLPS22]. With the former we obtain a time complexity of $O(N^{2/3}\gamma \log^3 N)$ and with the latter a time complexity of $O(N^{2/3}\gamma \log M)$.

Next, we introduce a quantum analogue of a hash table, which achieves an optimal $O(N^{2/3}\gamma)$ time complexity for element distinctness. The algorithm uses a more general version of our main theorem that replaces both the lookup cost, and the insert and delete cost, with an average cost. The proof of this theorem exploits a very recent result that allows us to take an average-case complexity over subroutines [BJY23]. The space complexity of this quantum hash table, however, is $\tilde{O}(N^{4/3})$. We therefore also introduce a space-efficient version of the quantum hash table achieving a time complexity of $O(N^{2/3}\gamma \log N)$, yet costing us a logarithmic factor in the process.

Finally, we suggest a method for constructing a quantum binary search tree, which is deemed impossible to implement in the quantum setting [BJLM13]. For this, we use an extension of the technique introduced in [BJLM13], by maintaining a superposition over all possible tree structures.

Based on our findings, we conclude that the limitations of using classical data structures in the quantum setting are much less restrictive than previously thought.

Contents

1 Introduction

Undoubtedly, quantum computing stands as one of the most promising recent areas of research. Through the exploitation of quantum phenomena such as interference, entanglement, and quantum parallelism, quantum computers can tackle once-thought-intractable computational problems exceeding classical approaches. Consider for example Shor's algorithm [Sho97] that, once large quantum computers are realised, would practically break all current cryptographic systems.

Generally, the efficiency of algorithms is measured in terms of time complexity, i.e. the number of elementary operations the algorithm performs. Obtaining time complexity bounds on quantum algorithms, however, often requires a lot of work. Hence a common starting point is to consider the query complexity first. This complexity represents how often the input of a problem is accessed or queried. A lower bounds on the query complexity serves as lower bound on the time complexity and is significantly easier to analyse.[1] Nevertheless, to accurately understand the practical power of quantum computers, quantum time complexity bounds are imperative.

In classical computing data structures serve as fundamental building blocks, allowing for efficient ways of organising, storing and manipulating data. By selecting the appropriate data structure, we can solve computational problems while minimising the computational and space costs required. Numerous data structures exist, from simple sorted arrays to complicated hash tables, each having different qualities. In our desire to achieve quantum time complexity bounds, the need for data structures in the quantum setting become apparent.

One of the first to notice this need was Ambainis in his algorithm for element distinctness [Amb03]. To prove a good time complexity bound, he required a data structure able to store a set of elements in some efficient manner.[2] Naturally, Ambainis wanted to use one of the several data structures known in the classical setting. Translating classical data structures directly to the quantum setting, however, led to numerous complications.

The primary concern is the so-called uniqueness problem. Often in data structures, the representation of a set depends on the order in which the elements are inserted or removed. Consider, for example, storing the set $\{1, 2, 3\}$ in an array. The arrays $[1, 2, 3]$, $[3, 2, 1]$ and $[1, 3, 2]$ all represent the same set, yet they are technically three different objects. For the quantum setting this means that there are several different quantum states representing the same set. Consequently, phenomena like interference cannot occur, since the different states do not cancel out anymore. This problem spoils the analysis in many of the quantum algorithms and likely the actual performance.

Other challenges include: running good average-case data structure operations on a superposition of inputs and making these operations reversible. Over the years, various papers have addressed these initial issues, allowing for more freedom in choosing the desired data structure. This raises the question: to what extent are we limited in constructing quantum data structures?

[1] Moreover, while being practically less interesting, query complexity upper bounds often act as a first step in proving meaningful time complexity upper bounds. As we will also see in this thesis.

[2] Efficient in this context means at most polylogarithmic time.

More specifically, our main research question is:

> *"What properties are required of quantum data structures, and how can we translate classical data structures for use in the quantum setting?"*

We answer these questions in the context of the problem of element distinctness, since it is a simple problem and, as noted before, one of the first to require a quantum data structure. We do so by synthesising the relevant literature and introducing new techniques to further combat the above concerns.

1.1 Previous Work

Informally, the problem of element distinctness can be stated as follows: given some list of N integers, each having a value of at most M, do there exist two elements with the same value?

Before proving a time complexity bound, Ambainis first showed a query complexity bound of $O(N^{2/3})$ for element distinctness [Amb03]. This bound is optimal, as it matches the earlier proven lower bound by Shi [Shi02]. Ambainis achieved this bound by translating a random walk search algorithm to the quantum setting. Inspired by his work, several more general quantum walk frameworks were introduced, including the Szegedy quantum walk [Sze04], MNRS quantum walk [MNRS11] and electric network quantum walk [Bel13] frameworks. We use this last framework to prove our results.

Ambainis introduced an ad-hoc data structure to adhere to the aforementioned concerns, combining a hash table with a skip list. Using this data structure, he proved a $O(N^{2/3}\log^4(N+M))$ time complexity bound for element distinctness. His paper actually addressed the more general problem of k-element distinctness, where one needs to decide if there are k integers with the same value.[3] His time complexity bound for this more general problem is $O(N^{k/(k+1)}\log^4(N+M))$.

A much simpler data structure, which is a quantum version of a radix tree, was later introduced by Bernstein, Jeffery, Lange and Meurer [BJLM13]. They used this quantum radix tree for the subset-sum problem, which requires a quantum walk similar to that for element distinctness. Despite the radix tree having a unique representation, obtaining a space-efficient unique memory representation is not obvious. By introducing a new technique of maintaining a superposition over all available memory locations, they were able to solve this issue.

In her PhD thesis [Jef14], Jeffery provided a more thorough explanation as to how quantum radix trees can be used for k-element distinctness. She showed a time complexity bound of $O(N^{k/(k+1)}(\log N + \log M))$, which gives a slightly better $O(N^{2/3}(\log N + \log M))$ time bound for element distinctness. The paper by Buhrman, Loff, Patro and Speelman [BLPS22] worked out the quantum radix tree even further by addressing some of the issues initially overlooked by Jeffery. These improvements were, however, not made in the context of element distinctness.

1.2 Our Contributions

Our results are presented in a (what we call), Quantum Word Random Access Machine (QWRAM) model of computation. This model extends the Quantum Random Access Machine (QRAM) model introduced in [BLPS22], that quantifies basic *classical* word operations as a variable γ. Depending on the specific architecture of the quantum computer, we either have $\gamma = O(1)$ or $\gamma = O(\log N)$. In the new model, we argue that we can also quantify elementary *quantum* word operations as γ.[4] We believe it is more likely that such operations can be implemented in constant time on a quantum computer than basic classical operations [KvdW24]. We use γ_N and

[3] For $k = 2$ we obtain the 'normal' element distinctness problem.

[4] See Section 2.4.4 for our definition of an elementary quantum word operation.

γ_M to distinguish between complexities of operations on $\log N$ and $\log M$ (qu)bits respectively.

Our main theorem is a black-box quantum algorithm for element distinctness with time complexity $O(N^{2/3}(\gamma_M + T_L + T_{ID} + T_C) + T_{\text{init}})$, for *any* quantum data structure that is able to lookup elements in time T_L, insert and delete elements in time T_{ID}, check for collisions in time T_C and initialise the data structure in time T_{init}. For this, we give a new formal definition of what is required of a quantum data structure for element distinctness. Moreover, we eliminate the need for the diffusion operator, making the required data structure operations all of a classical kind. The novel techniques used to prove this theorem are expected to be beneficial for quantum algorithms other than just element distinctness. These techniques include the use of the electric network framework [Bel13], creating dead-end edges [Bel13], and the introduction of a new type of graph called a permuted Johnson graph. In particular, they illustrate the intuitiveness of the electric network framework in analysing quantum walks on more complicated graph structures.

As a result of our theorem, we can slightly simplify two existing quantum data structures and reanalyse their time complexity in the QWRAM model. With Ambainis' quantum skip list [Amb03] we obtain a time complexity of $O(N^{2/3}\gamma_M \log^3 N)$. With the quantum radix tree, we adapt the improvements by [BLPS22] to the context of element distinctness, to obtain a time complexity of $O(N^{2/3}\gamma_M \log M)$. Both data structures are presented more formally and explained on an intuitive level.

Next, we introduce a quantum analogue of a hash table, which achieves an optimal $O(N^{2/3}\gamma_M)$ time complexity for element distinctness. The algorithm uses a more general version of our main theorem that replaces both the lookup cost, and the insert and delete cost, with an average cost. The proof of this theorem exploits a very recent result that allows us to take an average-case complexity over subroutines [BJY23]. The space complexity of this quantum hash table, however, is $\tilde{O}(N^{4/3})$. We therefore also introduce a space-efficient version of the quantum hash table achieving a time complexity of $O(N^{2/3}\gamma_M \log N)$, yet costing us a logarithmic factor in the process.

Finally, we suggest a method for constructing a quantum binary search tree, which is deemed impossible to implement in the quantum setting [BJLM13]. For this, we use an extension of the technique introduced in [BJLM13], by maintaining a superposition over all possible tree structures. However, to construct a quantum version of a binary search that we are able to use in an algorithm for element distinctness, several difficulties emerge. We therefore leave the concrete construction as a suggestion for future work.

See Table 1.1 for an overview of all time and space complexity bounds found for element distinctness.

Data structure	**Time complexity**	**Space complexity**
Quantum skip list (Corollary 5.8)	$O(N^{2/3}\gamma_M \log^3 N)$	$O(N^{2/3} \log^2 N \log M)$
Quantum radix tree (Corollary 5.11)	$O(N^{2/3}\gamma_M \log M)$	$O(N^{2/3} \log M)$
Quantum hash table 1 (Corollary 5.14)	$\boldsymbol{O(N^{2/3}\gamma_M)}$	$\boldsymbol{O(N^{4/3}\log M)}$
Quantum hash table 2 (Corollary 5.17)	$\boldsymbol{O(N^{2/3}\gamma_M \log N)}$	$\boldsymbol{O(N^{2/3}\log M)}$

Table 1.1: An overview of the time and space complexity bounds found for element distinctness using the different quantum data structures introduced. Here N is the input size, M the maximum value of any entry the input can have and γ_N,γ_M are the costs of classical and quantum elementary operations on $\log N$ or $\log M$ (qu)bits respectively. In bold are the new data structures introduced in this thesis.

1.3 Related Contributions

Quantum data structures are also used outside the context of element distinctness. For example, a combination of the quantum skip list and quantum radix tree was used by Aaronson, Chia, Lin, Wang and Zhang [ACL+19] in proving time complexity bounds for the closest pair and related problems. We believe our solutions can be applied to this context, potentially simplifying their presentation.

As mentioned before, the improvement by [BLPS22] to the quantum radix tree were given outside the context of element distinctness. Instead, they showed how to 'compress' sparse space-inefficient quantum data structures to space-efficient data structures, while keeping the overall time complexity roughly the same. As a result, this theorem simplified many of the proofs using data structures, including that of element distinctness. However, the proof is non-constructive, meaning we still need to construct a concrete space-efficient data structure for implementation purposes. Their presentation can only be used for proof purposes.

Finally, the Master's thesis by Gilyén [PG14] showed that with the use of a reversible sorting network [AKS83], a simple sorted array would suffice as an efficient data structure for element distinctness, when evaluated in a parallel model of computation. For this thesis, we do not consider this parallel model of computation as it is not commonly used throughout the literature.[5] His thesis does give a very detailed proof of how this time complexity is achieved, which inspired the proof of our main theorem. Belovs' paper on the electric network framework [Bel13] also gives a detailed proof, for 3-element distinctness. He, however, does not provide a concrete data structure. Yet, the techniques used are crucial for our main result.

1.4 Structure of the Thesis

The first two chapters lay the foundational groundwork for the thesis. In Chapter 2, we cover the basics of computational complexity and quantum computing, and introduce the QWRAM model of computation. We next present common quantum subroutines such as Grover's algorithm and amplitude amplification, discuss the translation of classical subroutines to the quantum setting, and demonstrate how to make any quantum subroutine controlled. Finally, we present a very recent result allowing us to take an average-case complexity over subroutines [BJY23]. In Chapter 3 we explore the three primary quantum walk frameworks: the Szegedy quantum walk, MNRS quantum walk, and electric network quantum walk frameworks. We use the notation as used in [JZ23], which differs slightly from the previous presentations.

In Chapter 4, we formally introduce the problem of element distinctness and present a known optimal query complexity algorithm using the MNRS quantum walk framework. Next, we discuss the need for quantum data structures for proving a matching time complexity. The remainder of the chapter is dedicated to defining a quantum data structure for element distinctness and proving our main theorem, for which we introduce the new permuted Johnson graph. At the end of the chapter we give intuition for the four main techniques used in the proof.

We apply our theorem to concrete quantum data structures in Chapter 5. After providing a thorough overview of classical data structures, we explain the challenges of translating classical data structures directly to the quantum setting. We finally demonstrate how to implement the quantum skip list, quantum radix tree and two versions of a quantum hash table. Additionally we suggest a construction for a quantum version of a binary search tree. The last section of the chapter is dedicated to reviewing the solutions to the earlier stated challenges.

We conclude the thesis in Chapter 6 with a discussion of the results and suggestions for future research directions.

[5] This might seem contradictory, as we ourselves introduce a new model of computation. However, switching from our model to the more common QRAM model is trivial.

2 | Preliminaries

In this chapter we provide the necessary background information needed to understand the results of this thesis. We start with an overview of what is covered in each section.

2.1 We describe some basic notations and conventions.

2.2 We discuss the basic notions of a computational problem, what it means for an algorithm to solve such a problem and metrics to measure its efficiency.

2.3 We briefly introduce the field of quantum computing.

2.4 We describe the models of computation used for evaluating quantum algorithms and introduce our own QWRAM model.

2.5 We discuss common quantum subroutines, the translation of classical subroutines to the quantum setting, a procedure to make any subroutine controlled and a method that allows us to take an average-case complexity over subroutines.

2.1 Basic Notation

We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$ and $[0, n]$ to denote the set $\{0, 1, \ldots, n\}$. When not specified otherwise, a number denoted by one of the letters j, k, ℓ, m, n is assumed to represent an integer. In most cases i is used as an integer, but only if it cannot be mistaken for a complex number. Capital letters like M, N usually represent powers of 2, where the exponent is denoted with the respective lower case letters m, n. With $\log(\cdot)$ we denote the base-2 logarithm $\log_2(\cdot)$ and we use $\oplus$ for addition modulo 2. If $z := a + bi$ is a complex number, we use $\overline{z} := a - bi$ to denote its complex conjugate. For $x \in \{0,1\}^n$ an n-bit string or an n-dimensional vector we use x_i to denote the value of the i^{th} character or coordinate respectively. For an n-dimensional vector v we define its norm as $||v|| = \sqrt{|v_1|^2 + |v_2|^2 + \cdots + |v_n|^2}$. Finally, for tuples $t, t' \in [n]^k$ we let d_H be the Hamming distance defined as $d_H(t, t') := |\{i \mid t_i \neq t'_i\}|$, denoting the number of positions where t and t' differ.

2.2 Computational Complexity Theory

We can use algorithms, i.e. a finite sequence of fixed instructions, to solve computational problems. Mathematically, we define such a problem as a family of functions $F_n : \{0,1\}^n \to \{0,1\}^m$, where $n, m \in \mathbb{N}$ are the input and output sizes respectively. We can often restrict ourselves to the setting where $m = 2$, i.e. inputs are either accepted or rejected, also called yes- or no-instances.

Definition 2.1 (Decision problem)**.** Let $n \in \mathbb{N}$. A *decision problem* is a family of functions $F_n : \{0,1\}^n \to \{0,1\}$, one for each input size n.

Whenever possible we leave the input size n implicit and simply write F. When needed to compute a more general problem $F : \{0,1\}^n \to \{0,1\}^m$ we can simply consider a collection of m

decision problems.

In many situations we do not require algorithms to solve problems exactly. Instead we can allow for some small error to occur. There are three error models we consider. The first one, confusingly named *zero-error*, ensures that every given output is correct. It is only with some small error the algorithm outputs $\perp$, indicating that it does not know the answer. Secondly we have *one-sided error*, meaning the algorithm always correctly rejects an input but it can accept inputs, with some small error, that should be rejected. Lastly we have the bounded-error (or two-sided error) setting, where the error can occur on both the yes- and no-instances of the problem. In quantum computing, most algorithms are of this last type.

Definition 2.2 (Algorithm with bounded error ε)**.** Let F be a decision problem. We say that an algorithm $\mathcal{A}$ solves F with *bounded error* ε if for all $x \in \{0,1\}^n$ we have $\Pr[\mathcal{A}(x) \neq F(x)] \leq \varepsilon$.

If $\mathcal{A}$ computes F with bounded error $1/3$, we simply say it computes F with *bounded error*.[1]

The efficiency of an algorithm is typically measured by the number of 'basic operations' it performs as a function of its input length. This metric is known as its time complexity. Other measures of efficiency include space complexity, which denotes the number of bits of memory needed, and query complexity, which represents the number of calls made to the input of a problem. A computational problem can also have a time, space and query complexity. This refers to the best-known algorithm solving that problem having said complexity.

The following notations will be useful in expressing these complexities.

Definition 2.3 (Asymptotic notation)**.** Let f and g be non-decreasing functions from $\mathbb{N}$ to $\mathbb{R}$, we have

- $O(f(n))$ denoting the set of functions $g(n)$ for which there exists a $c \in \mathbb{R}_{\geq 0}$ and $N \in \mathbb{N}$ such that $0 \leq g(n) \leq c \cdot f(n)$ for all $n \geq N$.
- $o(f(n))$ denoting the set of functions $g(n)$ for which for all $c \in \mathbb{R}_{\geq 0}$ there exists a $N \in \mathbb{N}$ such that $0 \leq g(n) < c \cdot f(n)$ for all $n \geq N$.
- $\Omega(f(n))$ denoting the set of functions $g(n)$ for which there exists a $c \in \mathbb{R}_{\geq 0}$ and $N \in \mathbb{N}$ such that $g(n) \geq c \cdot f(n)$ for all $n \geq N$.
- $\omega(f(n))$ denoting the set of functions $g(n)$ for which for all $c \in \mathbb{R}_{\geq 0}$ there exists a $N \in \mathbb{N}$ such that $g(n) > c \cdot f(n)$ for all $n \geq N$.
- $\Theta(f(n))$ denoting the set of functions $g(n)$ that are both in $O(f(n))$ and $\Omega(f(n))$.

Generally, $O(\cdot)$ and $o(\cdot)$ are used to indicate upper bounds, $\Omega(\cdot)$ and $\omega(\cdot)$ for lower bounds and $\Theta(\cdot)$ for tight bounds. It is conventional to write $f(n) = O(g(n))$ instead $f(n) \in O(g(n))$ for any of the asymptotic notations. We also use polylog(n) to denote $\log^{O(1)}(n)$ and sometimes $\tilde{O}(\cdot)$ to hide polylogarithmic factors.

2.3 Quantum Computing

A brief introduction to quantum computing as given here cannot fully cover this interesting area of research. We strongly advise the reader to refer to either the book by Nielsen and Chuang [NC00] or the lecture notes by de Wolf [dW23], on which this section is based, for needed clarifications.

[1] The convention is to use an error of $1/3$, but any constant fraction less than $1/2$ would suffice.

2.3.1 Quantum states

In quantum mechanics, a *quantum state* can be in multiple classical states, i.e. *branches*, at the same time, known as a *superposition*. With a classical state we mean the state that the system is found in after we observe it. While this might seem true for any state of a system, for quantum states this property is not presumed. Upon observing a quantum system, more commonly referred to as *measuring*, the underlying state of the system is influenced by this measurement. It *collapses* to one of its classical states.

Mathematically, we denote a quantum state as a unit vector living in a finite-dimensional complex vector space, in the literature referred to as a *Hilbert space*.[2] We use $\mathcal{H}(N)$ to denote an N-dimensional Hilbert space. Whenever possible we leave the dimension N implicit and simply write $\mathcal{H}$. Vectors living in $\mathcal{H}$ are written as 'kets' $|\psi\rangle$ and vectors living in the dual space $\mathcal{H}^*$ as 'bras' $\langle\psi|$. Thus $\langle\psi| = |\psi\rangle^\dagger$, where $\dagger$ acts as the conjugate transpose, i.e. $|\psi\rangle^\dagger = \overline{|\psi\rangle}^T$. This notation is called the *Dirac notation* and is standardly used in quantum mechanics.[3]

In classical computing, systems are constructed by concatenating smaller subsystems. Usually 2-level systems are called *bits*. Similarly, for quantum computing we consider quantum states as consisting of n 2-level systems called quantum bits or *qubits* for short.

Definition 2.4 (Qubit)**.** A *qubit* is a vector $|\varphi\rangle \in \mathcal{H}(2)$ defined as

$$|\varphi\rangle := \alpha_0 |0\rangle + \alpha_1 |1\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix},$$

where $\alpha_0, \alpha_1 \in \mathbb{C}$ and $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

Note that in the above definition, we have explicitly chosen some basis in which the vector is represented, namely $|0\rangle := \begin{bmatrix} 1 & 0 \end{bmatrix}^T$ and $|1\rangle := \begin{bmatrix} 0 & 1 \end{bmatrix}^T$. This is called the *standard* or *computational basis*. Whenever we measure a qubit it will collapse to one of the basis states, for example $|0\rangle$, with probability of the associated amplitude squared, i.e. $|\alpha_0|^2$. This concept is known as *Born's rule* and is one of the fundamental axioms of quantum mechanics.

We can create multi-qubit states by considering their *tensor product*, denoted as $|\varphi\rangle \otimes |\psi\rangle$ (or alternatively as $|\varphi\rangle |\psi\rangle, |\varphi, \psi\rangle$ or $|\varphi\psi\rangle$). We use $|\varphi\rangle^{\otimes n}$ to denote the n-fold tensor product $|\varphi\rangle \otimes |\varphi\rangle \otimes \cdots \otimes |\varphi\rangle$ which can be simplified to $|\overline{\varphi}\rangle$ whenever the underlying dimension is clear from context. An n-qubit state $|\varphi\rangle \in \mathcal{H}$ can be represented in the computational basis as

$$|\varphi\rangle := \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + \alpha_{N-1} |N-1\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{N-1} \end{bmatrix},$$

where $\alpha_i \in \mathbb{C}$ for all $i \in [0, N-1]$ and $\sum_{i=0}^{N-1} |\alpha_i|^2 = 1$. Note that the labels of the basis vectors are written in decimals instead of bit strings for sake of readability.

2.3.2 Operations on quantum states

In the previous section we found qubits to be complex unit vectors. Consequently, the natural way to operate on qubits is by using matrices. In particular we need a matrix M that is *norm-preserving*, meaning $\|M |\varphi\rangle\| = \||\varphi\rangle\|$, to ensure outcomes remain unit vectors and thus valid quantum states. It is an easy exercise to see that *unitary matrices*, i.e. matrices U satisfying

[2] Formally a Hilbert space is a complete vector space, potentially infinite-dimensional, equipped with an inner product. So it is technically a subclass of Hilbert spaces.

[3] It, for example, allows for convenient representation of the inner product as $\langle\psi| \cdot |\varphi\rangle = \langle\psi|\varphi\rangle$ and the outer product as $|\psi\rangle \langle\varphi|$.

the property $UU^\dagger = I$, are precisely those matrices. Here we use $\dagger$ once more to denote the conjugate transpose, only now for a matrix. We let $\mathcal{U}(N)$ denote the set of unitary operators from $\mathcal{H}(N)$ to itself. Similar to quantum states, we use $U^{\otimes n}$ to denote the n-fold tensor product of unitary matrices $U \otimes U \otimes \cdots \otimes U$. Note that since unitaries always have an inverse, all quantum computations are inherently reversible.

We now list some of the common unitary operations used throughout the literature. Perhaps the most important unitary is the Hadamard operation

$$\mathsf{H} := \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

It can be used to create uniform superpositions. For example, by applying $H^{\otimes n}$ to $\left|\overline{0}\right\rangle$, we can create the uniform superposition over all n-bit strings

$$\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle .$$

Other useful operations are the four Pauli matrices

$$\mathsf{X} := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \mathsf{Z} := \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad \mathsf{Y} := \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad \mathsf{I} := \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

For the computational basis states $|0\rangle$ and $|1\rangle$, the X acts as a bit-flip operation, Z introduces a -1 phase to $|1\rangle$, $\mathsf{Y} = i\mathsf{XZ}$ acts as a combination of the two (while adding a global i phase) and I is the identity, leaving the state unaltered. The final important 1-qubit unitaries are the phase gates

$$\mathsf{R}_\varphi := \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix} \qquad \mathsf{S} := \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \qquad \mathsf{T} := \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix},$$

rotating the phase of $|1\rangle$ by an angle $\varphi \in [0, 2\pi)$.[4]

Common 2- and 3-qubit unitaries are the so-called *controlled operations*. Here a 1-qubit unitary U is applied if and only if the control qubit is set to $|1\rangle$. For example, the CNOT gate flips the second qubit if the first qubit is set to $|1\rangle$. It is defined as

$$\mathsf{CNOT} := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \mathsf{I} & 0 \\ 0 & \mathsf{X} \end{bmatrix}.$$

In general, any m-controlled n-qubit unitary U is defined as

$$U_m^c := \begin{bmatrix} \mathsf{I}^{\otimes m} & 0 \\ 0 & U \end{bmatrix}.$$

We call these n qubits the *target qubits* and the m qubits the *controlling qubits*.

In the previous section we said that another way to operate on quantum states is by performing a measurement. Note that this operation is not reversible at all, since we collapse to a particular basis state of a superposition, losing all information of the other branches. We assume all measurements to be done in the computational basis. If the change of basis is not straightforward, there is probably some computational power hidden in that measurement.

[4] Note that $\mathsf{Z} = \mathsf{R}_\pi$, $\mathsf{S} = \mathsf{R}_{\pi/2}$ and $\mathsf{T} = \mathsf{R}_{\pi/4}$.

2.4 Model of Computation

The complexity of an algorithm is measured in a *model of computation*. Such a model allows for studying the performance of an algorithm independently of the technological implementation. In the setting of quantum computing, this is of particular importance, since currently no large-scale quantum computers exist.[5] We introduce a new model of computation by extending step by step from the commonly used *quantum circuit model*.[6]

2.4.1 Quantum circuit model

The classical circuit model consists of a labelled directed acyclic graph, called a *circuit*, whose vertices represent either *gates*, input nodes or output nodes. A *universal gate set* is a set of gates to which any operation can be reduced to. In the classical setting the NAND gate, or alternatively the AND and NOT gate, form such a universal gate set.

Translating the classical circuit model to the quantum setting then simply consists of replacing the AND and NOT gates with quantum analogues, that is, some unitary operator. For example, we could use the Toffoli gate that acts as a double controlled-not operation

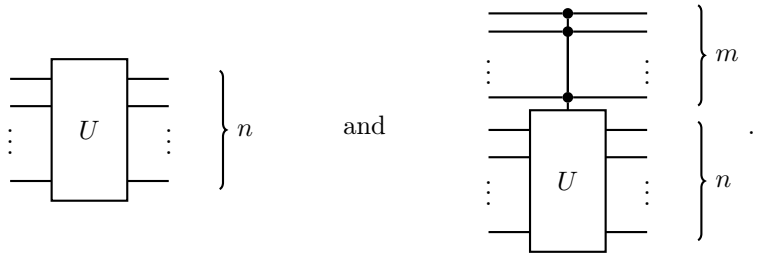
$$\mathsf{CCNOT} : |a\rangle\,|b\rangle\,|c\rangle \mapsto |a\rangle\,|b\rangle\,|c \oplus ab\rangle\,.$$

It is not hard to see that by choosing $c = 0$ we get the result of $\mathsf{AND}(a, b)$ in the last register and by choosing $a = 1$ and $b = 1$ we get $\mathsf{NOT}(c)$. The Toffoli gate alone, however, does not capture the full computational power of quantum computers. In fact, since there are uncountably many unitary operations, we technically would need an uncountable set of quantum gates for this.[7]

From a theoretic standpoint, this is no big issue; we can simply choose all 1-qubit gates together with the CNOT gate to be our universal gate set [NC00, Section 4.5.2]. However, implementation-wise it is an issue, as it is unrealistic to implement gates to infinite precision. Fortunately, due to the Solovay-Kiteav theorem, a finite gate set like $\{\mathsf{CNOT}, \mathsf{T}, \mathsf{H}\}$ can approximate any unitary operation with only polylogarithmic overhead [NC00, Appendix 3].

For our purposes, we consider all 1-qubit gates together with the CNOT also as our elementary operations, i.e. having time complexity $O(1)$. In particular, this includes computational basis measurements, which can be seen as a special type of 1-qubit gate. The time complexity of an algorithm now corresponds to the size of the circuit that implements it, i.e. the number of elementary gates required. The space complexity is represented by the height of the circuit, i.e. the number of different qubits that are acted on.

A n-qubit unitary and an m-controlled unitary on n-qubits are denoted respectively in a quantum circuit as

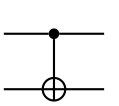
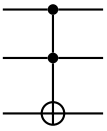


For controlled not operations, for example CNOT and CCNOT, we use $\oplus$ instead of the X-gate. Thus we respectively have

[5] And as mentioned before, it is unsure if such large quantum systems are even feasible.

[6] There are many different models, like for example the quantum Turing machine model [BV97]. By the quantum strong Church-Turing thesis all these models can be efficiently simulated by each other [KLM07].

[7] We use quantum gate and unitary operation synonymously.

 and  .

2.4.2 Quantum query model

One can extend the quantum circuit model to also allow access to an *oracle*.

Definition 2.5 (Standard oracle)**.** Let $f : \{0,1\}^n \to \{0,1\}$ be a function. The *standard oracle* $\mathcal{O}_f$ for f is defined as

$$\mathcal{O}_f : |i\rangle\,|b\rangle \mapsto |x\rangle\,|b \oplus f(i)\rangle\,,$$

for $i \in \{0,1\}^n$ and $b \in \{0,1\}$.

We often use an oracle to represent the input to a decision problem F. To prevent confusion between the input of the problem and the problem itself we represent its input as an N-bit string $x \in \{0,1\}^N$ where $x_i := f(i)$ for each $i \in \{0,1\}^n$. We denote an oracle for x as $\mathcal{O}_x$ and a call to $\mathcal{O}_x$ is called a *query*. The number of queries an algorithm makes is called the *query complexity* of the algorithm. Abstracting the input as a sort of black-box allows us to ignore the details as to how to input is presented to us. Consequently, it is unclear what the cost of this operation should be. It is, however, standard to assume a cost of $O(1)$.

In the end, the *quantum query model* is the quantum circuit model together with oracle access to the input of the considered computational problem.

2.4.3 Quantum RAM model

In classical computing, Random Access Memory (RAM) reads and writes are usually counted as elementary operations. For example, adding two integers stored in memory is thought of as a basic operation, while it would probably take at least $\Omega(\log N)$ to get these integers in the working memory.[8] For the quantum setting we would like to use similar types of gates. If we let $x \in \{0,1\}^N$ be an N-bit string, $i \in [N]$ an index and $b \in \{0,1\}$ a bit, we define a quantum RAM read as

$$\mathsf{QREAD}_\mathsf{N}\,|i\rangle\,|b\rangle\,|x\rangle = |i\rangle\,|b \oplus x_i\rangle\,|x\rangle$$

and write

$$\mathsf{QWRITE}_\mathsf{N}\,|i\rangle\,|b\rangle\,|x\rangle = |i\rangle\,|b\rangle\,|x_1,\ldots,x_{i-1},b \oplus x_i,x_{i+1},\ldots,x_n\rangle\,.$$

It is not hard to see that from the above gates we can construct the following important gate.[9]

Definition 2.6 (QRAG)**.** Let $x \in \{0,1\}^N$ be an N-bit string, $i \in [N]$ an index and $b \in \{0,1\}$ a bit. We define a quantum random access gate (QRAG) as

$$\mathsf{QRAG}_\mathsf{N}\,|i\rangle\,|b\rangle\,|x\rangle = |i\rangle\,|x_i\rangle\,|x_1,\ldots,x_{i-1},b,x_{i+1},\ldots,x_n\rangle\,.$$

Whenever possible we leave the gate size N implicit and simply write QRAG. As in the classical setting, we count $\mathsf{QRAG}s$ as elementary operations.

We can now state the *Quantum RAM* (QRAM) model as presented in [BLPS22].[10] Our final gate set is the set $\mathcal{G} := \mathcal{U}(2) \cup \{\mathsf{CNOT}\} \cup \{\mathsf{QRAG}_\mathsf{N} \mid N \text{ a power of } 2\}$. A quantum algorithm

[8] If for example the memory was stored in a tree-like structure.

[9] Consider reading, writing and then reading again.

[10] A similar model of computation is also referred to throughout the literature as QRAQM [Kup11] or QAQM [NPS20].

can now be written as a sequence of unitaries $U_0\mathcal{O}_x U_1 \mathcal{O}_x \cdots U_T$, where $U_i \in \mathcal{G}$ and $\mathcal{O}_x$ are queries, applied in sequence to some initial state $|\psi_0\rangle$. The output of the algorithm consists of a measurement of the final state $|\psi_T\rangle$. It is a simple exercise to show that every in-between measurement can be deferred to end of the computation by using one extra auxiliary bit for each measurement. Hence we make the assumption that we can make a measurement at any step in the computation, which simplifies the presentation.

2.4.4 Quantum word RAM model

Another convention in classical computing is to count simple operations on *words*, i.e. on a logarithmic number of bits, as elementary operations. Examples of these operations include addition, comparison, modulus and more. Such a model is known as the *word RAM model*. It is additionally standard to assume that the word size w matches that of the input size N, so $w = \Theta(\log N)$, which is known as a *transdichotomous model*. Practically all classical algorithms are evaluated in this model.

For quantum computing one should be hesitant to make these same assumptions, as the precise implementation of quantum computers is unclear. In [BLPS22] simple word operations are not counted as $O(1)$, but instead quantified using the variable γ. Depending on the specific architecture of the quantum computer, γ can be considered as either $O(1)$ or $O(\log N)$. Another paper by Belovs, Jeffery and Yolcu [BJY23] makes a similar quantification.

We extend this model even further by quantifying elementary *quantum* operations on words also as γ. With an elementary quantum word operation we mean applying the same 1-qubit gate from our gate set $\mathcal{G}$ to a logarithmic number of qubits.[11] For example, applying $\mathsf{H}^{\otimes \log N}$ on $\left|\overline{0}\right\rangle$ would normally cost $O(\log N)$ gates, but in our new model we count this as γ. Note that we also count oracle queries as elementary operations. We call our extended model the *quantum word RAM (QWRAM) model*, which we will use to evaluate all our results.

There are two arguments for why we believe this extension to be meaningful. The first is an argument for word operations in general. The speed of operations on multiple bits, depends heavily on the specific architecture used. Looping for a logarithmic number of steps in an algorithm, however, cannot be sped-up in the same way. Hence it is natural to assign a different cost to these operations. Secondly, simple classical operations require often a lot more gates in the quantum setting. For example, the addition of two $\log(N)$-bit numbers already requires $O(\log N)$ Toffoli gates [KvdW24, Section 7.6], while this would only count as γ in the new model. Hence we feel it is even more natural to quantify basic quantum word operations with γ, as we are asking less from our model.

As we will see in Section 4.1, the input of some problems consists of N integers of size at most M. In that case it is less clear what the word size should be: $\log N$, $\log M$ or perhaps $\log(N + M)$? Thus, we then make a distinction between γ_N and γ_M, quantifying the cost of basic operations on $\log N$ or $\log M$ (qu)bits respectively.

2.5 Quantum Subroutines

Using known quantum algorithms as subroutines is crucial for developing new algorithms. In this section we explain how to simulate any classical subroutine on a quantum computer and how to make any quantum subroutine controlled. But first, we dive into two essential quantum algorithms: Grover's algorithm and amplitude amplification.

[11] Since we know from Section 2.4.1 that it is unrealistic from an implementation stand-point to use all 1-qubit gate, we could also consider only counting the standard $\mathsf{H}, \mathsf{X}, \mathsf{Z}, \mathsf{Y}, \mathsf{I}, \mathsf{S}$ and T gates as elementary gates on which we can perform word operations. For our presentation this does not matter.

2.5.1 Grover's algorithm

Consider the following search problem.

Definition 2.7 (Unstructured search problem)**.** Given a string $x \in \{0,1\}^N$, find an index $i \in [n]$ such that $x_i = 1$. If no such i exists, output 0.

Denote the number of solutions in x by t, i.e. t is the Hamming weight of x defined as $|x|_H := |\{i \mid i \in [n] \text{ and } x_i = 1\}|$..

A classical algorithm would need at most $O(N)$ query and also time to solve this problem. When considering probabilistic classical algorithms, this can be improved to $O(N/t)$ expected cost (or if t is known worst-case one-sided error). In the quantum setting, we can do quadratically better.

Theorem 2.8 (Grover's algorithm [Gro96])**.** There exists a bounded-error quantum algorithm solving the unstructured search problem using at most $O(\sqrt{N/t})$ queries and $O(\gamma_N \sqrt{N/t})$ time assuming that t is known.

It is good to note that the γ_N in the above theorem comes from the fact that we view $\mathsf{H}^{\otimes \log N}$ as a basic quantum word operation (see Section 2.4.4).

2.5.2 Amplitude amplification

We can generalise Grover's algorithm to the following setting: suppose that we have a quantum circuit C that, when applied to the state $\left|\overline{0}\right\rangle$, gives

$$|\varphi\rangle := C\left|\overline{0}\right\rangle = \sqrt{p}\,|\varphi_G\rangle + \sqrt{1-p}\,|\varphi_B\rangle\,.$$

Here $|\varphi_G\rangle$ and $|\varphi_B\rangle$ are two orthogonal states, where measuring $|\varphi\rangle$ gives the 'good state' $|\varphi_G\rangle$ with probability p and the 'bad state' $|\varphi_B\rangle$ with probability $1-p$. Our goal is to only obtain the good state $|\varphi_G\rangle$ with some high probability. The assumption is that there exists a measurement (or a reflection) that can distinguish between $|\varphi_G\rangle$ and $|\varphi_B\rangle$.[12]

The naive solutions is to measure and prepare this state $O(1/p)$ times, which obtains the good state with probability close to 1. The quantum algorithm of amplitude amplification speeds this up quadratically.

Theorem 2.9 (Amplitude amplification [BHMT02])**.** Assume we can distinguish between the good and bad part of a state using a measurement or reflection. There exists a bounded-error quantum algorithm for amplifying the probability p of a good part of a state, created by some quantum circuit C, to close to 1, using $O(1/\sqrt{p})$ applications of C.

2.5.3 Classical subroutines

Fundamental to quantum algorithms is the ability to use classical subroutines. Recall that it is enough to only consider decision problems, since we can construct any computational problem with a collection of decision problems. As we have seen, quantum algorithms are naturally reversible, which is not necessarily the case for decision problems. Fortunately, we can make any irreversible decision problem $F : \{0,1\}^n \to \{0,1\}$ reversible by constructing the decision problem $F_r : \{0,1\}^{n+1} \to \{0,1\}^{n+1}$ defined as

$$F_r(x,b) := x \parallel b \oplus f(x),$$

[12] For example the good state could be $|\varphi_G\rangle = |\psi_G\rangle\,|0\rangle$ and the bad state $|\varphi_B\rangle = |\psi_B\rangle\,|1\rangle$, for some quantum states $|\psi_G\rangle$ and $|\psi_B\rangle$.

where $\|$ denotes the concatenation of strings. In fact, note that in the quantum setting this is simply the standard oracle of F (see Definition 2.5). So it suffices to construct a quantum circuit that implements $\mathcal{O}_F$, given that we have some classical circuit solving F.

We saw in Section 2.4.1 that the gate set $\{\mathsf{AND}, \mathsf{NOT}\}$ is universal for classical computing, so we can assume that our classical circuit consists only of gates in this set. Moreover, in that section we saw how to replace these gates with the Toffoli gate, denoted as CCNOT, assuming we can fix input bits and add auxiliary bits. So in fact we can easily turn the given classical circuit into a quantum circuit by just replacing the AND and NOT gates with the appropriate version of the CCNOT gate. There is, however, a lot of 'garbage' left on the auxiliary bits, resulting in the operation not being unitary. To clean this up we copy out the result of solving F to a new auxiliary qubit using a CNOT and run the circuit in reverse.[13] Note that this works since the Toffoli gate is self-inverse. While the new circuit solves F in time only twice as long, we do use a lot of auxiliary bits, namely linear in the size of the circuit.

Fortunately, we are able to decrease the auxiliary bits by dividing the circuit into smaller parts and uncompute as soon as possible, allowing us to recycle auxiliary bits. For a complete proof we refer to the lecture notes by Preskill [Pre98, Section 6.1].

Theorem 2.10 (Classical subroutines [Pre98, Pat23])**.** Let F be a decision problem. Suppose there exists a classical circuit C that solves F in time $T(n)$. Then there exists a quantum algorithm solving F in time $(T(n))^{1+o(1)}$ using at most $\tilde{O}(\log T(n))$ additional auxiliary qubits.

The impact of this lemma is tremendous: it means that we can implement any classical algorithm directly on a quantum computer, while keeping (roughly) the same time complexity. As a result, many of the algorithms in this thesis can be significantly simplified and explained purely on a classical level.

2.5.4 Controlled subroutines

Another common feature of quantum algorithms is that of applying a certain subroutine only when some control bit is activated. The following lemma shows how to make any unitary controlled.

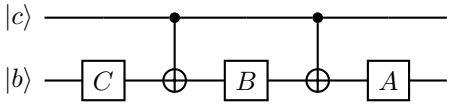
Lemma 2.11 (Controlled subroutines [Pat23])**.** Let U be a unitary which can be implemented by a circuit using T gates from $\mathcal{G}$. We can implement the controlled version of U defined as

$$U^c : |c\rangle |x\rangle = \begin{cases} |c\rangle (U |x\rangle) & \text{if } c = 1 \\ |c\rangle |x\rangle & \text{if } c = 0 \end{cases},$$

using $O(T)$ gates from $\mathcal{G}$.

Proof. Let $G \in \mathcal{G}$ be a gate used to implement U. We consider three cases:

- First suppose that $G \in \mathcal{U}(2)$. We know that there exist 1-qubit unitaries A, B and C, such that $ABC = \mathsf{I}$ and $AXBXC = U$ [NC00, Corollary 4.2]. It is easy to see that the circuit

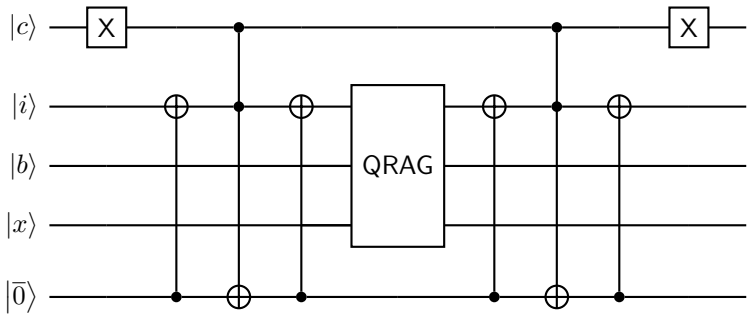


  implements G in a controlled way.

[13] We should be careful with the word 'copy' as this is not allowed in quantum due to the no-cloning theorem [WZ82]. The fact that it is allowed here is because the solution of F is either $|0\rangle$ or $|1\rangle$, and thus a classical state.

- Next suppose that $G = \mathsf{CNOT}$. We can simply transform G to a Toffoli gate to implement it in a controlled way. We can make the Toffoli gate using 16 gates from $\mathcal{G}$ [NC00, Figure 4.9].[14]

- Lastly suppose that $G = \mathsf{QRAG}_N$ for N some arbitrary power of 2. Since the QRAG_N acts as an identity when the index is 0, we can simply replace the index with an all-zero quantum state of $\log N$ qubits, see the following circuit:



  The two CNOT gates together with the Toffoli gate acts as a controlled swap on $|i\rangle$ and $|\overline{0}\rangle$.

□

2.5.5 Averaging over subroutines

Something that we take for granted in classical computing is averaging out the complexity over different calls to some subroutine. Suppose that there exists some subroutine U that for inputs $z \in \mathcal{Z}$ runs in time $T(z)$. The cost of running the subroutine on $z \in \mathcal{Z}$, chosen with probability p_z, classically is quite obviously $\sum_{z\in\mathcal{Z}} p_z T(z)$. If, however, we run this subroutine on the superposition of inputs $\sum_{z\in\mathcal{Z}} |z\rangle$ we seem to incur a cost of $\max_{z\in\mathcal{Z}} T_z$. Naively, we are required to 'wait' for the branch of the superposition with the longest run-time to finish before we can move on to the rest of the algorithm.

A very recent paper by Belovs, Jeffery and Yolcu [BJY23] solves this problem, by arguing that we can in fact take some kind of average over the different branches of the superposition.

Theorem 2.12 ([BJY23, Theorem 3.10])**.** Let $U = \sum_{z\in\mathcal{Z}} |z\rangle\langle z| \otimes U(z)$ be subroutine that, controlled on the input z, executes $U(z)$ in time $T(z)$. Suppose a quantum algorithm $\mathcal{A}$ uses T basic operations and Q queries to U, where $\{|\psi_1\rangle, |\psi_2\rangle, \ldots, |\psi_Q\rangle\}$ are the states right before the queries. Then there exists another quantum algorithm $\mathcal{A}'$ that can simulate $\mathcal{A}$, i.e. it has the same input and output behaviour, with bounded error in time complexity

$$O\left(T + \sum_{q\in[Q]} \sum_{z\in\mathcal{Z}} ||(|z\rangle\langle z| \otimes \mathsf{I})|\psi_q\rangle||^2 \cdot T(z)\right).$$

The above theorem is an adaptation of [BJY23, Theorem 3.10] to match with our presentation. By noting that $L^{(z)}(A, O \oplus B(O), \xi) \equiv \sum_{q\in[Q]} |(|z\rangle\langle z| \otimes \mathsf{I})|\psi_q\rangle||^2$, we can go from their notation to ours.

[14] More specifically, two H gates, seven T gates, one S gate and six CNOT gates.

3 | Quantum Walk Frameworks

Quantum walks frameworks act as blueprints for creating quantum algorithms based on the well-studied concept of random walks. In most cases, allowing for a generic speed-up over their classical counterparts. In this chapter, we explore the three main quantum walk frameworks. Our presentation is based on the notation as seen in [JZ23] and closely follows [Jef24]. We start with an overview of what is covered in each section.

3.1 We explain classical random walks and how they can transformed to the quantum setting.

3.2 We discuss the graph search problem and how it can be solved using the Szegedy [Sze04] and MNRS [MNRS11] quantum walk frameworks, both of which are direct translations of their classical counterpart.

3.3 We state Belovs electric network quantum walk framework [Bel13].

3.1 Random and Quantum Walks

Before we can study random and quantum walks, we first need to introduce the mathematical object that they are related to, namely the *Markov chain*.

3.1.1 Markov chains

Informally, a Markov chain is a random process that moves among the different elements of some state space. Crucially, we move in such a way that the probability of moving to a state x only depends on the state you are currently in.

Definition 3.1 (Markov chain)**.** A *Markov chain* is a sequence of random variables $(X_t)_{t=0}^{\infty}$ on some state space $\mathcal{X}$ such that for all $t \geq 0$ and $x_0, \ldots, x_t \in \mathcal{X}$ it satisfies the *Markov property*:

$$\Pr[X_t = x_t \mid X_{t-1} = x_{t-1}, \ldots, X_0 = x_0] = \Pr[X_t = x_t \mid X_{t-1} = x_{t-1}].$$

We restrict ourselves to finite-state time-invariant Markov chains.[1] A process that gives rise to such a chain is called a *Markov process*, which we can conveniently represent using a *transition matrix* $P \in \mathbb{R}^{\mathcal{X} \times \mathcal{X}}$ because of these restrictions. Each entry $P_{x,y}$ of the matrix gives the probability of moving from state x to state y. Thus P is a stochastic matrix, i.e. a matrix where all entries are non-negative and the sum of elements in a row add up to 1. We use $P_{x,\cdot}$ to denote the row of P associated with state x, which is interpreted as the probability distribution over all states that x can 'reach'.

To obtain a Markov chain from our Markov process we also need some initial distribution $\rho_0 \in \mathbb{R}^{\mathcal{X}}$ of X_0 to start in. From here, we can calculate the distribution of the next random variable X_1

[1] A Markov chain $(X_t)_{t=0}^{\infty}$ is called *time-invariant* if for all $t \geq 0$ and $x_0, x_1 \in \mathcal{X}$ we have $\Pr[X_t = x_1 \mid X_{t-1} = x_0] = \Pr[X_1 = x_1 \mid X_0 = x_0]$. Thus the probability of moving from a particular state to another state is independent of the time step the process is currently in

by computing $\rho_0 P$.[2] More generally, the distribution $\rho_t \in \mathbb{R}^{\mathcal{X}}$ of X_t can now be computed by $\rho_0 P^t$. An important distribution is the so-called *stationary distribution*.

Definition 3.2 (Stationary distribution)**.** A probability distribution $\pi \in \mathbb{R}^{\mathcal{X}}$ is called a *stationary distribution* of Markov process P if it is a left 1-eigenvector of P, i.e. $\pi P = \pi$.

If the Markov process is able to return to any state x in any number of steps, a property known as *irreducibility*[3], then it has a unique stationary distribution. Moreover, if additionally for all states the greatest common divisor of all path lengths from that state to itself is 1, a property known as *aperiodicity*, then any starting distribution also converges to that unique stationary distribution. A Markov process is *ergodic* if it is both irreducible and aperiodic. We formalise the above in the following theorem.

Theorem 3.3 ([LPW08, Theorem 4.9])**.** Let P be an ergodic Markov process. We have that:

(i) P has a unique stationary distribution $\pi \in \mathbb{R}^{\mathcal{X}}$.

(ii) For every initial distribution $\rho \in \mathbb{R}^{\mathcal{X}}$ we have

$$\lim_{t\to\infty} \sum_{x\in\mathcal{X}} ||\rho P^t - \pi||_{tv} = 0.$$

Here $||\cdot||_{tv}$ is the *total variation distance*, for distributions $\rho, \sigma \in \mathbb{R}^{\mathcal{X}}$ defined as

$$||\rho - \sigma||_{tv} := \frac{1}{2} \sum_{x\in\mathcal{X}} |\rho(x) - \sigma(x)|.$$

3.1.2 Random walks

When given some graph $G = (V, E)$ we can construct a Markov process over the vertices V whose transition matrix P is defined by the underlying structure of the graph. Such a Markov process is called a *random walk on a graph*, or simply a *random walk*. We consider walking only on weighted undirected graphs, which precisely correspond to *time-reversible* Markov processes.[4]

Definition 3.4 (Random Walk)**.** Let $G = (V, E)$ be a weighted undirected graph with weight function $w : E \to \mathbb{R}_{\geq 0}$. A *random walk* over the vertices V in G is described by the Markov process $P \in \mathbb{R}^{V\times V}$, that for $u, v \in V$ is defined as

$$P_{u,v} = \frac{w(\{u, v\})}{w(u)},$$

where $w(u) = \sum_{v\in V} w(\{u, v\})$ is the sum of the weights of all edges containing u.

For sake of readability, from now on we use $w(u, v)$ instead of $w(\{u, v\})$, yet emphasising that still $w(u, v) = w(v, u)$. It is also useful to extend the domain of the weight function to all pairs of vertices, where $w(u, v) = 0$ if $\{u, v\} \notin E$. If no specific weight function is defined for a graph we assume all edges have a weight of one. Lastly, we denote the neighbourhood of a vertex $v \in V$ as $\mathcal{N}(v) := \{u \in V \mid \{u, v\} \in E\}$.

We can easily show the existence of a stationary distribution for a random walk.

[2] Notice how we need to use right multiplication due to the construction of P, meaning that ρ_0 and ρ_1 are row vectors. This is a convention in the literature that we stick to for coherency. Although we could have just as well constructed P to work with the, perhaps more common, left multiplication.

[3] Formally, a Markov process P is *irreducible* if for any two states $x, y \in \mathcal{X}$ there exists an integer t (which is allowed to depend on x and y) such that $(P^t)_{x,y} > 0$

[4] A Markov process P is called *time-reversible* if for all $x, y \in \mathcal{X}$ and π a stationary distribution it satisfies the *detailed balance* property: $\pi(x)P_{x,y} = \pi(y)P_{y,x}$

Theorem 3.5. Let P be a random walk on the graph $G = (V, E)$. The probability distribution $\pi \in \mathbb{R}^V$ defined as

$$\pi(u) = \frac{w(u)}{2W(G)},$$

for $u \in V$ where $W(G) := \sum_{e \in E} w(e)$, is a stationary distribution for P.

Proof. First note that π is indeed a distribution since

$$\sum_{u \in V} \pi(u) = \sum_{u \in V} \frac{w(u)}{2W(G)} = \frac{\sum_{u,v \in V} w(u,v)}{2W(G)} = 1.$$

Let $u \in V$ be an arbitrary vertex, we have

$$(\pi P)(u) = \sum_{v \in V} \pi(v) P_{u,v} = \frac{\sum_{v \in V} w(u,v)}{2W(G)} = \pi(u),$$

from which it follows that $\pi P = \pi$ is a left 1-eigenvector of P and thus a stationary distribution by Definition 3.2. □

Fundamental for random walks is the ability to, as suggested by its name, simulate *walking* on a graph. When present in some vertex u we can 'randomly' walk to any of its neighbours by sampling a vertex from the distribution $P_{u,\cdot}$. Our goal is to use this walking simulation as the basis for search algorithms (see Section 3.2). To simplify analysis of these algorithms we want to use the result of Theorem 3.3. Fortunately, ergodic Markov processes correspond exactly to *connected* and *non-bipartite* graphs[5]. Recall that a bipartite graph means that we can partition the vertices into two sets $V_1 \cup V_2 = V$ such that $\{v_1, v_2\}$ can be an edge only if $v_1 \in V_1$ and $v_2 \in V_2$. We only consider walking on these types of graphs.

3.1.3 Quantum walks

To use the walking behaviour imposed by the random walk in the quantum setting, we require the simulation to be unitary. Currently, the states are vertices and after each step of the walk we forget the vertex that we came from, making the step irreversible and thus non-unitary. As discussed in Section 2.5.3, one can make any operation reversible by simply remembering the original input. Thus we can transform our states to now consisting of two vertices, the one you are currently in, and the one that you came from. We can view this as walking over the edges of the graph, instead of the vertices.

Given a random walk P, we can dissect taking a step from a vertex u to one of its neighbours into two operations:

1. *Sample* some $i \in [d(u)]$ according to $P_{u,\cdot}$, where $d(u)$ is the *degree* of u.
2. *Transition* from u to its i^{th} neighbour.

As to how these operations are computed depends on the model in which the underlying graph of the random walk is given in. For this thesis, we assume the graph to represented in the *edge list model*; here the graph is described by a list of neighbours for each vertex. Formally, we model this as having query access to two functions. For each $u \in V$ a function $f_u : [d(u)] \to V$ whose image is $\mathcal{N}(u)$ and a query to $d(\cdot)$, to get the degree of a vertex. First we compute the degree of u using $d(u)$, after which we can easily sample an $i \in [d(u)]$. Next we can compute $f_u(i)$ to obtain the i^{th} neighbour $v \in V$ of u. Replacing u with v results in the desired operation.

The above model assumes that the edges are labelled by integers 1 up to $d(u)$. One can generalise this by saying that for each vertex u we can have a label set $L(u)$ that is in one-to-one correspondence with $[d(u)]$. So we now have query access to a function $f_u : L(u) \to V$, again with image

[5] Connectedness implies an irreducibility and non-bipartiteness implies aperiodicity.

$\mathcal{N}(u)$. An example for a situation where a label set differing from $[d(u)]$ is more convenient, is for *Johnson graphs* (see Definition 4.2). Here vertices are represented by sets $S \subseteq \binom{[n]}{k}$ for some integer $k \in \mathbb{N}$. We can move between different vertices by removing some $i \in S$ and inserting some $j \in [n] \setminus S$. Thus a natural label set of this graph is $(i, j) \in S \times ([n] \setminus S) =: L(S)$. We consider this general label set from now on.

So what are the quantum versions of the sampling and transition operations? The only way for quantum algorithms to use randomness is by collapsing a superposition. For example, to sample $i \in [d(u)]$ we can create a superposition over all $|i\rangle$ and apply a measurement. However, for quantum walks to improve over random walks we need to ability for quantum effects to occur. Therefore we keep the superposition intact and let interference do the work.

Thus, the quantum version of the sampling map means generating the superposition

$$|u, 0\rangle \mapsto \sum_{i \in L(u)} \sqrt{P_{u,f_u(i)}}\, |u, i\rangle \,.$$

For the transition operation we need to generate the map

$$|u, \ell\rangle \mapsto |v, \ell'\rangle \,,$$

where $v = f_u(\ell)$ and $\ell' \in L(v)$ is such that $\ell_v(\ell') = u$. When $L(u) = \mathcal{N}(u)$ for all $u \in V$ the above map simply is the vertex swap $|u, v\rangle \mapsto |v, u\rangle$. Although this might seems convenient to use, in the case of Johnson graphs this vertex swap is incredibly inefficient, requiring the copying of an r-sized set.

3.2 From Classical to Quantum Walk Algorithms

The beauty of random walks, and thus quantum walks, is that they can be used to build search algorithms. In particular they solve the following computational problem.

Definition 3.6 (Graph search problem)**.** Given a weighted undirected graph $G = (V, E)$ and a set of 'marked' vertices $\mathcal{M} \subseteq V$, determine whether $\mathcal{M} \neq \emptyset$ (decision version) or find some $v \in \mathcal{M}$ if there exists one (search version).

What this marked set looks like depends on the underlying problem we want to solve. In a sense, algorithms solving the above problem act as *frameworks* in which other computational problems can be encoded.

In this section we introduce two quantum algorithms that solve the above problem, both being a direct quadratic improvement of its corresponding classical algorithm. For our purposes, we restrict our attention to only the decision version of the graph search problem.[6]

[6] One might think that there is no difference between the decision and search version since deciding if a marked vertex exists requires finding a vertex that is marked. Although this is classically the case, quantumly we are able to prove the existence of a marked vertex without actually finding one. For example for the Szegedy framework we introduce next it took several years to prove a quantum algorithm finding a marked vertex [AGJK19], while the decision problem was already solved.

3.2.1 Szegedy's quantum walk framework

A natural classical algorithm for solving the graph search problem is as follows:

Algorithm 1 (Random walk search 1)**.** Let P be a random walk on $G = (V, E)$ and $\mathcal{M} \subseteq V$,

1. Sample $u \in V$ from the stationary distribution $\pi \in \mathbb{R}^V$.
2. Repeat T times:
   (a) Check if $u \in \mathcal{M}$, if so, output u.
   (b) Sample $v \in V$ form $\mathcal{N}(u)$ according to $P_{u,\cdot}$ and set u equal to v. This can be seen as taking a step of the walk.
3. Output 'no marked vertices'.

Here T is some fixed function over the input size that prevents the algorithm from running indefinitely. The question now remains: What is the smallest that T can be, while having high probability of outputting the correct answer? The following property quantifies this question.

Definition 3.7 (Hitting-Time)**.** Let P be a random walk on $G = (V, E)$ and $\mathcal{M} \subseteq V$. The hitting-time, denoted as $HT(P, \mathcal{M})$, is the expected number of steps that a walker needs to take to go from the stationary distribution to some marked vertex $u \in \mathcal{M}$.

It is a simple consequence of Markov's inequality[7] that the probability a walker reaches a marked vertex in the first $O(HT(P, \mathcal{M}))$ steps is at least $\Omega(1)$.

We can bound T using the hitting-time, resulting in the time complexity for Algorithm 1 being

$$O(\mathsf{S} + HT(P, \mathcal{M})(\mathsf{U} + \mathsf{C})).$$

Here S is the cost of sampling a vertex from the stationary distribution (*setup cost*), U the cost of sampling a neighbour v of any vertex u and transitioning to v (*update cost*) and C the cost of checking if a vertex u is marked (*check cost*). Generally, we find that the exact value of the hitting-time is unknown as it depends implicitly on the input of the problem. So we often replace the hitting-time with some known upper-bound, which we denote as $\mathcal{HT}$.

In turns out that when given a classical algorithm of the form of Algorithm 1, we immediately obtain a quantum algorithm that performs quadratically better.

Theorem 3.8 (Szegedy's framework [Sze04, Jef24])**.** Let $G = (V, E)$ be some graph and $\mathcal{M} \subseteq V$ a set of marked vertices. Let P be a random walk on G and $\pi \in \mathbb{R}^V$ its stationary distribution. Suppose that we can implement the following subroutines:

Setup In cost S generate the state

$$|\pi\rangle := \sum_{u \in V} \sqrt{\pi(u)}\, |u\rangle\,,$$

Update In cost U we can

1. For any $u \in V$ construct the *sampling map*

$$|u, 0\rangle \mapsto \sum_{\ell \in L(u)} \sqrt{P_{u, f_u(\ell)}}\, |u, \ell\rangle\,,$$

[7] Markov's inequality says that if X is some nonnegative random variable with expectation μ, then $\Pr[X \geq c\mu] \leq 1/c$, where $c > 0$ is some constant.

2. For $u \in V$, $\ell \in L(u)$ implement the *transition map*

$$|u, \ell\rangle \mapsto |v, \ell'\rangle\,,$$

where $v := f_u(\ell)$ and ℓ' is such that $f_v(\ell') = u$

Check In cost C, for any $u \in V$ implement the map

$$|u\rangle \mapsto \begin{cases} -|u\rangle & \text{if } u \in \mathcal{M} \\ |u\rangle & \text{otherwise} \end{cases}$$

Then there exists a quantum algorithm that decides if $\mathcal{M} \neq \emptyset$ with bounded error in complexity

$$O\left(\mathsf{S} + \sqrt{\mathcal{HT}}\,(\mathsf{U} + \mathsf{C})\right),$$

here $\mathcal{HT}$ is an upper bound on the hitting-time $HT(P, \mathcal{M})$ whenever $\mathcal{M} \neq \emptyset$.

3.2.2 MNRS quantum walk framework

In Algorithm 1 we check if a vertex is marked at every step of the walk. It could be that for some application this checking step is very expensive. In that case we want to limit these steps. Consider the following classical algorithm for the graph search problem.

Algorithm 2 (Random walk search 2)**.** Let P be a random walk on $G = (V, E)$ and $\mathcal{M} \subseteq V$,

1. Sample $u \in V$ from the stationary distribution $\pi \in \mathbb{R}^V$.
2. Repeat T_1 times:
   (a) Check if $u \in \mathcal{M}$, if so, output u.
   (b) Repeat T_2 times:
      i. Sample $v \in V$ form $\mathcal{N}(u)$ according to $P_{u,\cdot}$ and set u equal to v. This can be seen as taking a step of the walk.
3. Output 'no marked vertices'.

Let us discuss which values of T_1 and T_2 we should take, starting with the latter. We know from Theorem 3.3 that every initial distribution of the random walk eventually convergences to its unique stationary distribution. We additionally know that the stationary distribution for random walks is very similar to the uniform distribution (see Theorem 3.5).[8] Sampling from that distribution gives us the 'best' probability of finding a marked vertex. The following quantity represents the rate of convergence.

Definition 3.9 (Mixing-time)**.** Let P be a random walk with stationary distribution π. The *mixing-time* τ_{mix} of P is defined as the smallest $t \geq 0$ such that for any initial distribution $\rho \in \mathbb{R}^V$ we have

$$||\rho P^t - \pi||_{tv} \leq \frac{1}{3}.$$

As with the hitting-time, the mixing-time is somewhat difficult to calculate exactly. Hence we often bound the mixing-time using another property.

[8] In fact, when the graph is regular, it is exactly the uniform distribution.

Definition 3.10 (Spectral Gap)**.** Let P be a random walk with eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_N$. The *spectral gap* δ of P is defined as $\delta := \lambda_1 - \lambda_{\max}$, where $\lambda_{\max} := \max_{i \geq 2} |\lambda_i|$.

Theorem 3.11 ([LPW08, Theorem 12.4])**.** Let P be a random walk with stationary distribution π, mixing-time τ_{mix} and spectral gap δ. We have

$$\left(\frac{1}{\delta} - 1\right) \ln\left(\frac{3}{2}\right) \leq \tau_{\text{mix}} \leq \frac{1}{\delta} \ln\left(\frac{3}{\pi_{\min}}\right),$$

where $\pi_{\min} = \min_{u \in V} \pi(u)$.

Recall that P is a stochastic matrix, so all eigenvalues are in $[-1, 1]$. If we would use the above bound, we could have that situation that $\delta = 0$ when $\lambda_{\max} = 1$. However, since we restrict ourselves to only connected graphs we have $\lambda_2 < 1$. Furthermore, since we also require the graph to be non-bipartite, we additionally have $\lambda_N > -1$. Therefore the bound in Theorem 3.11 is always well-defined.

Thus taking $T_2 \geq 1/\delta$ steps ensures that we end-up close to the stationary distribution. The probability of sampling a marked vertex from the stationary distribution is $\varepsilon = \sum_{u \in \mathcal{M}} \pi(u)$. So by taking $T_1 \geq 1/\varepsilon$, Algorithm 2 solves the graph search problem with bounded error in complexity

$$O\left(\mathsf{S} + \frac{1}{\varepsilon}\left(\frac{1}{\delta}\mathsf{U} + \mathsf{C}\right)\right).$$

Here, S, U and C are once more the setup, update and check costs respectively.

Similar to the Szegedy quantum walk, we are able to instantly get a quadratic quantum speed-up when given a classical algorithm of the form of Algorithm 2.

Theorem 3.12 (MNRS framework [MNRS11, Jef24])**.** Let $G = (V, E)$ some graph and $\mathcal{M} \subseteq V$ a set of marked vertices. Let P be a random walk on G and $\pi \in \mathbb{R}^V$ its stationary distribution. Suppose that we can implement the following subroutines:

Setup In cost S generate the state

$$|\pi\rangle := \sum_{u \in V} \sqrt{\pi(u)}\, |u\rangle,$$

Update In cost U we can

1. For any $u \in V$ construct the *sampling map*

$$|u, 0\rangle \mapsto \sum_{\ell \in L(u)} \sqrt{P_{u, f_u(\ell)}}\, |u, \ell\rangle,$$

2. For $u \in V$, $\ell \in L(u)$ implement the *transition map*

$$|u, \ell\rangle \mapsto |v, \ell'\rangle,$$

where $v := f_u(\ell)$ and ℓ' is such that $f_v(\ell') = u$

Check In cost C, for any $u \in V$ implement the map

$$|u\rangle \mapsto \begin{cases} -|u\rangle & \text{if } u \in \mathcal{M} \\ |u\rangle & \text{otherwise} \end{cases}$$

Then there exists a quantum algorithm that decides if $\mathcal{M} \neq \emptyset$ with bounded error in complexity

$$O\left(\mathsf{S} + \frac{1}{\sqrt{\varepsilon}}\left(\frac{1}{\sqrt{\delta}}\mathsf{U} + \mathsf{C}\right)\right),$$

here δ is a lower bound on the spectral gap and ε a lower bound on the fraction of marked vertices.

We can relate the complexity of the above theorem to that of Theorem 3.8 by noting that

$$\frac{1}{\varepsilon} \leq HT(P, \mathcal{M}) \leq \frac{1}{\varepsilon\delta}.$$

So in deciding which theorem to use, we need to consider where the hitting-time falls in this range and the relative costs of the update and check step. Often walks will be over graphs whose spectral gap is known beforehand, making the MNRS framework easier to apply directly.[9]

3.3 Electrical Network Quantum Walk Framework

We can generalise the quantum walks of the previous section by letting our initial distribution be any distribution ρ, instead of only the stationary distribution. To achieve this generalisation we need to observe that a weighted graph can be viewed as a network of resistors, where each edge of weight w can be seen as a $1/w$ ohm transistor. The properties of these *electrical networks* can be directed related to that of a random walk.

3.3.1 Electrical networks

In an electrical network, current *flows* from one node to the other depending on their *potential difference*. To model this as an undirected weighted graph $G = (V, E)$ we fix some arbitrary direction over the vertices denoted by $\vec{E}$. So for all $\{u, v\} \in E$, we either have $(u, v) \in \vec{E}$ or $(v, u) \in \vec{E}$, but not both. The inherent structure of the graph remains undirected however.

Definition 3.13 (Flow)**.** Let $G = (V, E)$ be a graph. A *flow* on G is a function $\theta : \vec{E} \to \mathbb{R}$ that is extended to have a domain over edges in both directions by $\theta(v, u) = -\theta(u, v)$ for every $(u, v) \in \vec{E}$. For all $u \in V$ we define the flow coming out of u as $\theta(u) := \sum_{v \in \mathcal{N}(u)} \theta(u, v)$. For $u \in V$,

- if $\theta(u) = 0$ we say the flow is *conserved* in u.
- if $\theta(u) > 0$ we say u is a *source*.
- if $\theta(u) < 0$ we say u is a *sink* or a *target*.

Definition 3.14 (st-flow)**.** If a flow θ has a unique source s and target t we call it an st-flow. If $\theta(s) = 1$, it is a *unit* st-flow.

Definition 3.15 (Energy)**.** The *energy* of a flow θ is defined as

$$\mathcal{E}(\theta) = \sum_{(u,v) \in \vec{E}} \frac{\theta(u, v)^2}{w(u, v)}$$

In an electrical network, the flow corresponds to the unit st-flow that has minimal energy. This property is captured by the *effective resistance* between s and t, and is defined as $R_{s,t} := \min\{\mathcal{E}(G) \mid \theta \text{ a unit } st\text{-flow}\}$. The energy is minimised by spreading the flow across the graph as much as possible.

[9] Additionally, with the MNRS framework one was also always able to find a marked vertex, unlike with the Szegedy framework initially.

3.3.2 Belovs' quantum walk framework

We can relate the properties of an electrical network to that of a random walk by allowing a flow to start in some distribution ρ of the vertices (the initial distribution) and end in a set of vertices (the set of marked vertices).

Definition 3.16 ($\rho\mathcal{M}$-flow)**.** Let θ be a flow on G, ρ a distribution on V and $\mathcal{M}$ a set of marked vertices. If $\theta(u) = \rho(u)$ for all $u \in V \setminus \mathcal{M}$ we call θ a $\rho\mathcal{M}$-flow.

Definition 3.17 (Effective Resistance)**.** Let θ be a flow on G, ρ a distribution on V and $\mathcal{M}$ a set of marked vertices. The effective resistance between ρ and $\mathcal{M}$ is defined as the minimum energy of any ρM-flow

$$R_{\rho,\mathcal{M}}(G) := \{\mathcal{E}(\theta) \mid \theta \text{ a } \rho\mathcal{M}\text{-flow}\}$$

In a very non-trivial way, Belovs was able to express the hitting-time of this more general setting using the properties of electric networks.

Theorem 3.18 (Electric network quantum walk framework [Bel13, Jef24])**.** Let $G = (V, E)$ some graph and $\mathcal{M} \subseteq V$ a set of marked vertices. Let P be a random walk on G and $\rho \in \mathbb{R}^V$ some intitial distribution. Suppose that we implement the following subroutines:

Setup In cost S generate the state

$$|\rho\rangle := \sum_{u \in V} \sqrt{\rho(u)}\,|u\rangle\,,$$

Update In cost U we can

1. For any $u \in V$ construct the *sampling map*

$$|u, 0\rangle \mapsto \sum_{\ell \in L(u)} \sqrt{P_{u,f_u(\ell)}}\,|u, \ell\rangle\,,$$

2. For $u \in V$, $\ell \in L(u)$ implement the *transition map*

$$|u, \ell\rangle \mapsto |v, \ell'\rangle\,,$$

where $v := f_u(\ell)$ and ℓ' is such that $f_v(\ell') = u$

Check In cost C, for any $u \in V$ implement the map

$$|u\rangle \mapsto \begin{cases} -|u\rangle & \text{if } u \in \mathcal{M} \\ |u\rangle & \text{otherwise} \end{cases}$$

Then there exists a quantum algorithm that decides if $\mathcal{M} = \emptyset$ with bounded error in complexity

$$O(\mathsf{S} + \sqrt{\mathcal{R}\mathcal{W}}(\mathsf{U} + \mathsf{C})),$$

where $\mathcal{W}$ is an upper bound on the total weight of the graph and $\mathcal{R}$ an upper bound on the effective resistance $R_{\rho,\mathcal{M}}(G)$

Comparing the above theorem with Szegedy's framework (Theorem 3.8) we have that the setup step now depends on some initial distribution ρ instead of the stationary distribution π. Further, the upper bound on the hitting-time $\mathcal{HT}$ is now replaced with $\mathcal{RW}$. Important to note is that we do not automatically get a quadratic speed-up over the classical counterpart, since $\mathcal{RW}$ is

not the complexity of a classical walk algorithm. It turns out, that in the case where $\rho = \pi$, we have that $HT(\mathcal{M}, P) = 2W(G)R_{\pi,\mathcal{M}}(G)$ [CL23]. Thus giving us a concrete way of computing the hitting-time needed for Szegedy's framework.

The above theorem is of particular importance in this thesis. Properties of an electric network are often much more intuitive to analyse compared to the spectrum of a graph (as needed for the MNRS framework). Hence making it more versatile for analysing quantum walks on more complicated graph structures, which we will need for proving our main theorem (Theorem 4.8 and Theorem 4.9).

4 Data Structure Dependent Time Complexity of Element Distinctness

The quantum walk frameworks introduced in the previous chapter can be utilised to construct an optimal query algorithm for the problem of element distinctness. However, proving a similar time complexity bound demands the use of an efficient quantum data structure. In this chapter, we demonstrate precisely what is required from such a data structure by proving a data structure dependent time complexity bound for element distinctness. We start with an overview of what is covered in each section.

4.1 We state the problem of element distinctness, show an optimal query complexity quantum algorithm and discuss the need for quantum data structures.

4.2 We prove our main theorem, which gives a data structure dependent time complexity of element distinctness by using a formal definition of a quantum data structure.

4.3 We provide an overview of the techniques used in the proof of the main theorem and explain them at a more intuitive level.

4.1 Element Distinctness

Consider the following decision problem: given some list of integers, do there exist two elements with the same value. This type of problem is known as *element distinctness*.

Definition 4.1 (Element distinctness problem)**.** Given as input a list of $N = 2^n$ positive integers $x_1, \ldots, x_N \in [M]$ of size at most $M = 2^m$, do there exist distinct $i, j \in [N]$ such that $x_i = x_j$.

We say that an $i \in [N]$ is an *index* for the *value* $x_i \in [M]$. When there are two distinct $i, j \in [N]$ such that $x_i = x_j$ we call it a *collision*. We assume that $M > N$, since otherwise by the pigeonhole principe, there trivially exists a collision.

Recall from Section 2.4.2 that we are able to access the input to the problem via the oracle $\mathcal{O}_x$. The cost of this operation is seen as $O(1)$, but since we are querying integers instead of bits, we would technically need $O(\log M)$ steps to write the result of the query down. In our QWRAM model (Section 2.4.4), however, we view querying to an oracle as an elementary quantum operation, so in fact we count the actual cost as γ_M. Important to note is that querying an integer still counts as one query, it is only the time complexity of the query that we count as γ_M.

4.1.1 Query complexity

Let us now first focus our attention on the query complexity of element distinctness. It is not hard to see that in the classical setting one needs exactly N queries: querying only $N-1$ indices

could lead to the one colliding index not being queried. Quantumly, one might try to apply Grover's algorithm (Section 2.5.1), since we can view the element-distinctness problem as an unstructured search problem over the pairs $(i, j) \in [N]^2$. By Theorem 2.8, this gives a query complexity of $O(\sqrt{N^2}) = O(N)$, merely matching the classical bound.

We can make smarter use of Grover's algorithm to get something better than classical, as shown by Buhrman, Durr, Heiligman, Hoyer, Magniez, Santha and de Wolf [BDH+05]. First query $r < N$ random indices from $[N]$ and let $\mathcal{I}$ denote this set. We are going to check if any of the indices $i \in \mathcal{I}$ are part of a collision. We do so by running a Grover search over the other $j \in [N] \setminus \mathcal{I}$ indices to see if $x_j \in \{x_i \mid i \in \mathcal{I}\}$. This takes a total of $r + O(\sqrt{N-r}) = O(r + \sqrt{N})$ queries by Theorem 2.8. The probability that this search returns a collision, if there is one, is at least r/N. By repeating the above procedure $O(\sqrt{N/r})$ we can boost the succes probability to be at least $\Omega(1)$ using amplitude amplification (Section 2.5.2). This gives a total query complexity of $O(\sqrt{N/r} \cdot (r + \sqrt{N})) = O(\sqrt{Nr} + N/\sqrt{r})$ which is minimised by choosing $r = \sqrt{N}$ resulting in a final query complexity of $O(N^{3/4})$.

At the same time, the best query lower bound known for element distinctness is $\Omega(N^{2/3})$, as proven by Shi [Shi02], leaving us with a gap between the known lower and upper bounds. Fortunately, it turns out that we can indeed also solve this problem in at most $O(N^{2/3})$ queries by using one of the quantum walk frameworks described in the previous chapter. The first to prove such a bound was Ambainis [Amb03], using his own version of a quantum walk.[1] His algorithm, however, is quite complex and requires many pages of analysis. We therefore give a more traditional proof using the MNRS quantum walk framework (Section 3.2.2), which is significantly simpler.

For this, we consider walking over a special type of graph.[2]

Definition 4.2 (Johnson graph)**.** Let $n, k \in \mathbb{N}$ for $k < n$. The *Johnson graph* $J(n, k) = (V, E)$ is a graph with vertex set

$$V := \binom{[n]}{k} = \{S \subseteq [n] \mid |S| = k\}$$

and edge set

$$E := \{\{S, S'\} \mid S, S' \in V \text{ and } |S \cap S'| = k - 1\}.$$

Intuitively, two vertices are connected if they differ in exactly one element. Thus we can move from one vertex to the other by first deleting one element and then inserting some new element. See Figure 4.1 for an example of a Johnson graph.

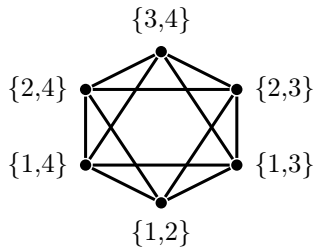


Figure 4.1: A representation of the Johnson graph $J(4, 2)$.

In the following lemma we describe a few useful properties of Johnson graphs needed for the analysis of the quantum walk in the MNRS framework.

[1] In fact, the quantum walk frameworks described in Chapter 3 are generalisations of the methods used in Ambainis his paper.

[2] Actually, Ambainis' original quantum walk was over a bipartite variant of the Johnson graph.

Lemma 4.3. Let $J(n,k) = (V,E)$ be a Johnson graph, we have that

(i) $J(n,k)$ has $\binom{n}{k}$ vertices.

(ii) $J(n,k)$ is $k(n-k)$-regular.

(iii) $J(n,k)$ has spectral gap $\delta = \frac{n}{k(n-k)}$.

Proof. The proofs of (i) and (ii) are simple combinatorial exercises left for the reader. For (iii) we use the known fact that the eigenvalues of the normalised adjacency matrix[3] of $J(n,k)$ are given by $\lambda_i = \frac{(k-i)(n-k-i)-i}{k(n-k)}$ for $i \in [k]$ [BH12, Section 12.3.2]. It follows by the definition of the spectral gap (Definition 3.10) that

$$\delta = \lambda_1 - \lambda_{\max} = 1 - \frac{(k-1)(n-k-1)-1}{k(n-k)} = \frac{n}{k(n-k)}.$$

□

Theorem 4.4 ([Amb03])**.** There exists a bounded-error quantum algorithm solving the element distinctness problem in $O(N^{2/3})$ queries.

Proof. Consider an instance $x_1, x_2, \ldots, x_N \in [M]$ of element distinctness. We are going to construct a random walk P (Definition 3.4) over the Johnson graph $J(N,r) = (V,E)$ where r is some integer that we choose later. By the definition of a Johnson graph, we have that each vertex is represented by some $S \subseteq [N]$ of size $|S| = r$. We are going to change this slightly and say each vertex $S \in V$ is represented by the set $X_S := \{(i, x_i) \mid i \in S\}$ instead (so each index in the vertex now also holds its corresponding value). The way the vertices are connected in the graph is identical as before. We define the set of marked vertices as

$$\mathcal{M} := \{X_S \in V \mid \text{there exist distinct } i,j \in S \text{ such that } x_i = x_j\}.$$

We know from Lemma 4.3(ii) that $J(N,r)$ is $r(N-r)$-regular, so its stationary distribution π is unique and equal to the uniform distribution by Theorem 3.5. The associated random walk for $X_S, X_{S'} \in V$ is by definition

$$P_{X_S,X_{S'}} = \begin{cases} \frac{1}{r(N-r)} & \text{if } \{X_S, X_{S'}\} \in E \\ 0 & \text{otherwise} \end{cases}.$$

For each $X_S \in V$ we define the label set $L(X_S) := S \times ([N] \setminus S)$. Intuitively, a label $(i,j) \in L(X_S)$ means that we are deleting (i, x_i) from X_S and inserting (j, x_j) with $j \notin S$. We need to implement the following subroutines in order to use the MNRS framework.

Setup Generating the state

$$|\pi\rangle := \frac{1}{\sqrt{\binom{N}{r}}} \sum_{X_S \in V} |X_S\rangle,$$

takes r queries in superposition, since each X_S is of size r. We fix the elements of $|X_S\rangle$ to appear in sorted order over the indices.[4]

[3] An adjacency matrix is a $|V| \times |V|$ matrix used to represent a graph, where an entry $a_{i,j} = 1$ if there is an edge between vertex i and j and 0 otherwise.

[4] Note that we could have chosen any encoding, since it does not have an effect on the query complexity.

Update For $X_S \in V$, constructing the sampling map

$$|X_S\rangle\,|\overline{0}\rangle \mapsto \frac{1}{\sqrt{r(N-r)}} \sum_{(i,j)\in L(X_S)} |X_S\rangle\,|i,j\rangle\,,$$

takes zero queries, since the labels do not depend on the values. Implementing the transition map

$$|X_S\rangle\,|i,j\rangle \mapsto \left|X_{(S\setminus\{i\})\cup\{j\}}\right\rangle|j,i\rangle$$

takes two queries, since we need to query to value of the new index j and query the value for i once more to delete it.

Check For any $X_S \in V$, checking if S contains any collision requires zero queries since we already have access to all associated values of S by construction of X_S.

Analysis By Lemma 4.3(iii) we have that $\delta = \frac{N}{r(N-r)} \geq \frac{1}{r}$. We can bound the fraction of marked vertices ε by assuming that there is exactly one collision. More collisions would only increase the number of marked vertices, making the problem easier. We have

$$\epsilon = \frac{|\mathcal{M}|}{|V|} \geq \frac{\binom{N-2}{r-2}}{\binom{N}{r}} = \frac{r(r-1)}{N(N-1)} \geq \frac{r^2}{N^2},$$

where the first inequality follows from Lemma 4.3(i). Thus by Theorem 3.12 there exists a bounded-error quantum algorithm for element distinctness with query complexity

$$O\left(S + \frac{1}{\sqrt{\varepsilon}}\left(\frac{1}{\sqrt{\delta}}U + C\right)\right) = O\left(r + \frac{N}{\sqrt{r}}\right).$$

This cost is minimised by choosing $r = N^{2/3}$, giving the desired bound of $O(N^{2/3})$. □

4.1.2 The need for quantum data structures

We have seen how to construct an optimal query algorithm, but what about the time complexity of element distinctness? Let us return to the classical setting first, where we discussed the need for N queries. Once we have obtained all values of these queries we can store them in a sorted array, which takes $O(N \log N)$ time. To find a collision, we can simply traverse through the array in $O(N)$ time and see if there are two elements next to each other with the same value. This algorithm has a time complexity of $O(N \log N)$, which is only a logarithmic factor away from the optimal classical query complexity. We say the time complexity *matches* the query complexity if these are within a polylogarithmic factor of each other.

Considering the time complexity of Grover's algorithm, we find the initial two quantum algorithms introduced in the previous section to take $O(N \log(N))$ and $O(N^{3/4} \log(N))$ time respectively. Both match their query complexity, but are, as we have seen, not optimal. Instead we need to consider the time complexity of the algorithm of Theorem 4.4, i.e. the time it takes to implement each of the subroutines. For now let us only focus on the update step.

In the update step we need to delete an element from $|X_S\rangle$ and insert one. Since the algorithm requires the elements to appear in sorted order this would take $O(r)$ time in the worst case, as we would need to move at most r elements.[5] Let us reanalyse the run-time of the quantum walk algorithm, under the (unrealistic) assumption that the time complexity of the setup and checking steps are identical to their respective query complexity. We have

$$O\left(S + \frac{1}{\sqrt{\varepsilon}}\left(\frac{1}{\sqrt{\delta}}U + C\right)\right) = O\left(r + N\sqrt{r}\right),$$

[5] We would have the same issue for any chosen ordering of the elements.

which is minimised by taking r to be some constant, resulting in a time complexity of $O(N)$. This cost is once more equal to the optimal classical bound. Yet if there was some efficient way, say in $O(\text{polylog}(N))$ time, for inserting and deleting elements from the set X_S the time complexity would become $O\left(N^{2/3}\text{polylog}(N)\right)$, by choosing $r = N^{2/3}$. Of course the setup and checking steps are more likely to be some logarithmic factor away from their query complexity, but then the argument still stands.

The only way to obtain these polylogarithmic complexities is by considering some efficient *data structure* storing X_S. Recall that a data structure is essentially just a way of organising and storing data, while allowing for efficient access and manipulation of the data. Technically we even have represented $|X_S\rangle$ in a data structure, namely a sorted array, albeit an inefficient one. We emphasise that the above is exactly the reason for needing data structures in the quantum setting.

The question now remains: what does a quantum version of a data structure look like? And what is exactly required of such a quantum data structure, besides efficient insertion and deletion? We use the rest of the chapter to answer these non-trivial questions.

4.2 Data Structure Dependent Time Complexity

In general, a quantum data structure is a mapping from some object we would like to store to some Hilbert space. On this space we are allowed to perform certain unitary maps, i.e. the data structure operations.[6] In this section we are going to prove our main theorem, which gives a time complexity bound for element distinctness assuming some specific quantum data structure. We previously found that we needed to store a set X_S of indices and values from which we could efficiently insert and delete elements. Let us now delve deeper into all the exact operations required from our quantum data structure.

4.2.1 Initial requirements

Recall the proof of Theorem 4.4. If we examine each of the subroutines we find that we need the following data structure operations:

- Determine if a particular (i, x_i) is in our data structure. This operation is needed in the setup step, since we first need to create a superposition over sets, i.e. distinct elements, before we can insert each element into the data structure.
- Insert an element (i, x_i) into the data structure. This operation is needed r times for the setup step and also once in the update step.
- Delete an element (i, x_i) from the data structure, needed only once in the update step.
- Construct a superposition over all (i, x_i) such that $i \in S$, known as the *diffusion operator*. This operation is needed in the sampling map of the update step, since we have to create a superposition over the labels $(i, j) \in S \times ([N] \setminus S)$.
- Determine if there is a collision, i.e. if there are $(i, x_i), (j, x_j)$ such that $i \neq j$ and $x_i = x_j$. Obviously this is required for the last checking subroutine.

In Chapter 5 we find a natural way to construct a quantum data structure by modifying known classical data structures. The first three operations are common for most classical data structures, so they probably require the least effort to implement in the quantum setting. The last operation is specific to the problem of element distinctness and we find this to be quite straightforward to implement: we simply augment the data structure with a counter that represents the number of collisions and we update this counter after each insertion or deletion (see Section 5.3 and Section 5.4 for concrete examples). The diffusion operation, on the other hand, is a quantum-specific operation and demands the most attention. It turns out that we can eliminate the need for the diffusion operation, by using, as far as we know, a new technique. Moreover, this technique allows for further simplification of the implementation of the setup subroutine.

[6] Recall that all operations on quantum states must be unitary (see Section 2.3.2)

4.2.2 Permuted Johnson graph

We introduce a new type of graph, which 'lies in between' the Johnson graph and another graph called a *Hamming graph*.

Definition 4.5 (Hamming graph)**.** Let $n, k \in \mathbb{N}$ with $k < n$. The *Hamming graph* $H(n,k) = (V,E)$ is a graph with vertex set

$$V := [n]^k = \{t := (t_1, t_2, \ldots, t_k) \mid t_i \in [n], \forall i \in [k]\}$$

and edge set

$$E := \{\{t, t'\} \mid t, t' \in V \text{ and } d_H(t,t') = 1\},$$

where d_H is the Hamming distance.

While Johnson graphs connect sets that differ in exactly one value, Hamming graphs connect tuples that differ in exactly one coordinate, see Figure 4.2 for an example. The representation in this figure is not standard, but it facilitates later comparison. Like the Johnson graph, the Hamming graph is frequently used in quantum walks. For example, it can be used as an alternative to show an optimal query complexity bound for element distinctness [Chi13, Lecture 13].

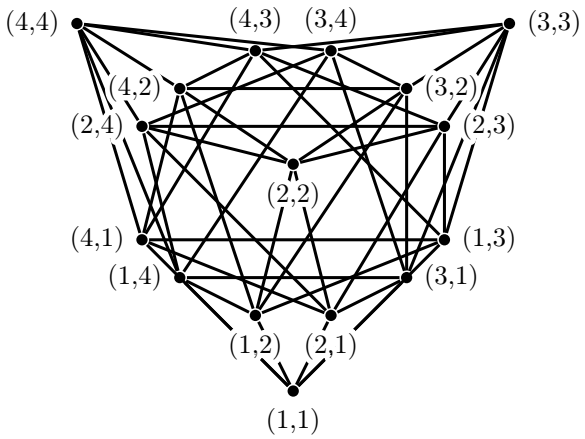


Figure 4.2: A representation of the Hamming graph $H(4,2)$.

Instead of allowing elements to be repeated within a tuple, we restrict our attention to those tuples that are permutations of distinct elements. We call this the *permuted Johnson graph*.

Definition 4.6 (Permuted Johnson graph)**.** Let $n, k \in \mathbb{N}$ with $k < n$. The *permuted Johnson graph* $PJ(n,k) = (V,E)$ is a graph with vertex set

$$V := \left\{\sigma(S) := (i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(k)}) \mid \forall \sigma \in \mathcal{S}_k \text{ and } S := \{i_1 < i_2 < \cdots < i_k\} \in \binom{[n]}{k}\right\}$$

and edge set

$$E := \{\{\sigma(S), \sigma'(S')\} \mid \sigma(S), \sigma'(S') \in V \text{ and } d_H(\sigma(S), \sigma'(S') = 1\},$$

where $\mathcal{S}_k$ represents the set of all permutations of $[k]$ and d_H the Hamming distance.

Similar to the Hamming graph we connect two vertices if they differ in exactly one coordinate and similar to the Johnson graph we have that elements are not duplicated. Thus we can either view the graph as a Johnson graph where each vertex is replaced with its $k!$ possible permutations or as a Hamming graph where we prevent vertices from having duplicate elements. We prefer the first interpretation (hence the name), as we find the structure of the graph to resemble the Johnson graph the most.[7] See Figure 4.3 for an example of a permuted Johnson graph. We compare each of the three graph structures in Figure 4.4.

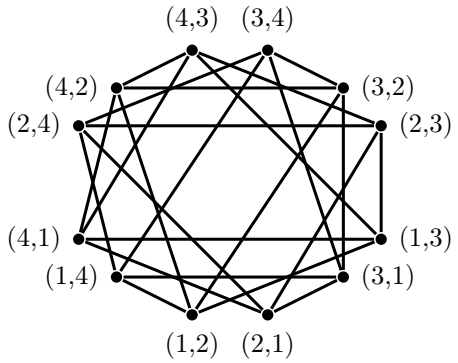
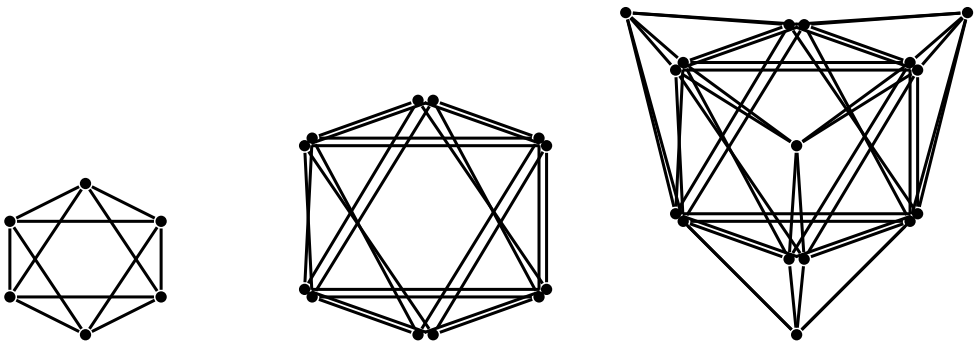


Figure 4.3: A representations of the permuted Johnson graph $PJ(4, 2)$.



(a) $J(4, 2)$ (b) $PJ(4, 2)$ (c) $H(4, 2)$

Figure 4.4: Comparison of the structures of a Johnson graph (a), a permuted Johnson graph (b) and a Hamming graph (c). Vertex labels are deliberately left out to improve legibility.

4.2.3 The main theorem

We show in the proof of our main theorem, how we can use the permuted Johnson graph to get rid of the diffusion operator and simplify the setup subroutine. As stated before, this theorem can be used in a black-box way to prove a data structure dependent time complexity for element distinctness. By simply plugging in a quantum data structure we obtain a quantum algorithm for element distinctness. For this we first need formally define what a quantum data structure for element distinctness precisely is.

[7] Alternatively we could have called the graph the *non-repeating Hamming graph*.

Definition 4.7 (Quantum data structure for element distinctness)**.** Let $x_1, x_2, \ldots, x_N \in [M]$ be an instance of element distinctness and let $s \in [N]$. An s-sized *quantum data structure for element distinctness* is a family of mappings $\mathsf{D}_s : \{X_S \mid S \in \binom{[N]}{\leq s}\} \to \mathcal{H}_s$ defined as $X_S \mapsto |\mathsf{D}_s(X_S)\rangle$ on which we can perform the following unitary operations:

Initialise A map U_{init} on $\mathcal{H}_s$ defined as $\left|\overline{0}\right\rangle \mapsto |\mathsf{D}_s(X_\emptyset)\rangle$ in time at most T_{init}.

Lookup A map U_L on $\text{span}\{|i\rangle \mid i \in [N]\} \otimes \mathcal{H}_s \otimes \text{span}\{|0\rangle, |1\rangle\}$ defined as

$$|i\rangle |\mathsf{D}_s(X_S)\rangle |0\rangle \mapsto |i\rangle |\mathsf{D}_s(X_S)\rangle |(i \in S)\rangle ,$$

where $(i \in S_k) = 1$ if $i \in S_k$ and 0 otherwise, in time at most T_L. We do not care how U_L acts on other inputs, as long as it is controlled on $|i\rangle$.

Insert & Delete A map U_I on $\text{span}\{|i, x_i\rangle \mid (i, x_i) \in [N] \times [M]\} \otimes \mathcal{H}_s$ for $S \in \binom{[N]}{<s}$ and $i \notin S$ defined as

$$|i, x_i\rangle |\mathsf{D}_s(X_S)\rangle \mapsto |i, x_i\rangle \left|\mathsf{D}_s(X_{S\cup\{i\}})\right\rangle$$

and a map $U_D := U_I^\dagger$ that for $i \in S$ acts like

$$|i, x_i\rangle |\mathsf{D}_s(X_S)\rangle \mapsto |i, x_i\rangle \left|\mathsf{D}_s(X_{S\setminus\{i\}})\right\rangle ,$$

in time at most T_{ID}. Again we do not care how U_{ID} acts on other inputs, as long as it is controlled on $|i, x_i\rangle$.

Check A map U_C on $\mathcal{H}_k \otimes \text{span}\{|0\rangle, |1\rangle\}$ defined as

$$|\mathsf{D}_s(X_S)\rangle |0\rangle = \begin{cases} |\mathsf{D}_s(X_S)\rangle |1\rangle & \exists \text{ distinct } i, j \in S_k \text{ such that } x_i = x_j \\ |\mathsf{D}_s(X_S)\rangle |0\rangle & \text{otherwise} \end{cases} ,$$

in time at most T_C.

Since we are only considering the element distinctness problem in this thesis, we simply say a *quantum data structure*, when referring to a quantum data structure for element distinctness. Whenever possible, we leave the maximum set size s implicit and simply write D.

We are now finally able to state our main theorem, which we break down into two versions for presentation purposes. The first shows a data structure dependent quantum algorithm for element distinctness, where the worst-case complexity is considered for each of the subroutines. The second version shows how we can improve upon this worst-case complexity, by considering an average-case lookup cost and average-case insert and delete cost.

Theorem 4.8 (Main theorem (Version 1))**.** Let D be an $(N^{2/3}+1)$-sized quantum data structure. There exists a bounded-error quantum algorithm for element distinctness with time complexity at most

$$O(N^{2/3}(\gamma_M + T_L + T_{ID} + T_C) + T_{\text{init}}).$$

Proof. Consider an instance $x_1, x_2, \ldots, x_N \in [M]$ of element distinctness. Similar to the proof of Theorem 4.4 we are going to construct a random walk P, but this time over the newly introduced permuted Johnson graph $PJ(N, r) = (V, E)$, where again r is some integer we choose later. Similar to before, we change the original definition of a permuted Johnson graph (Definition 4.6) slightly by saying each vertex is represented by the permutation $\sigma(X_S)$, where σ is the permutation corresponding to the original vertex $\sigma(S)$.

The natural label set for $\sigma(X_S) \in V$ is $L(\sigma(X_S)) = [r] \times [N] \setminus S$, since we are not allowed to insert an element already present in the tuple. Creating a superposition over elements not in S

would, however, still require the diffusion operator. A much nicer label is $L(\sigma(X_S)) = [r] \times [N]$, since we can easily create superpositions over the elements $[r]$ and $[N]$. To achieve this label set we are going to extend the edge set E by allowing for r^2 so-called *dead-end edges* for every vertex. This is an adaptation of the trick used in [Bel13, Lemma 6]. Let

$$V_{de} := \bigcup_{\sigma(X_S) \in V} \{Z_{\sigma(X_S),\ell,j} \mid \ell \in [r], j \in [S]\}$$

be the set of dead-end vertices and

$$E_{de} := \{\{\sigma(X_S), Z_{\sigma(X_S),\ell,j}\} \mid Z_{\sigma(X_S),\ell,j} \in V_{de}\}$$

the set of dead-end edges. Now let $\overline{G} = (\overline{V}, \overline{E})$ be the new graph with vertex set $\overline{V} := V \cup V_{de}$ and edge set $\overline{E} := E \cup E_{de}$. Notice how because of the extra vertices, we indeed have the label set $L(\sigma(X_S)) = [r] \times [N]$. The dead-end edges, however, also need a label set. This is simply the set $L(Z_{\sigma_{X_S},\ell,j}) = \{0\}$, as they all have only one neighbour.

Once again we can assume there is a unique collision, if one exists, as more collisions only make the problem easier. Let $\{a, b\}$ for $a, b \in [N]$ be this unique collision. For convenience we partition V into $V_\emptyset, V_a, V_b$ and $V_{a,b}$ where

$$\begin{aligned} V_\emptyset &:= \{\sigma(X_S) \in V \mid a, b \notin S\} \\ V_a &:= \{\sigma(X_S) \in V \mid a \in S, b \notin S\} \\ V_b &:= \{\sigma(X_S) \in V \mid a \notin S, b \in S\} \\ V_{a,b} &:= \{\sigma(X_S) \in V \mid a, b \in S\} \end{aligned}$$

So we have that $\overline{V} = V_\emptyset \cup V_a \cup V_b \cup V_{a,b} \cup V_{de}$. We switch between the different representations of $\overline{V}$ whenever convenient. It is also helpful to take make a similar partition for the set $\mathscr{S} := \{S \mid S \in \binom{[n]}{r}\}$, thus $\mathscr{S} = S_\emptyset \cup S_a \cup S_b \cup S_{a,b}$. We choose $\mathcal{M} := V_{a,b}$ to be our set of marked vertices. By construction each vertex in V has degree rN and each vertex in V_{de} has degree 1. Thus by Definition 3.4 the associated random walk P for $u, v \in \overline{V}$ is simply

$$P_{u,v} := \begin{cases} 1/(rN) & \text{if } u \in V \text{ and } \{u, v\} \in \overline{E} \\ 1 & \text{if } u \in V_{de} \text{ and } \{u, v\} \in E_{de} \\ 0 & \text{otherwise} \end{cases}.$$

If we want to use the MNRS framework as before (Section 3.2.2) we would need to evaluate the spectrum of $\overline{G}$, which is not obvious at all. Hence we choose to evaluate the quantum walk using the electric network framework (see Section 3.3). In this framework one needs to give an upper bound on the effective resistance and the total weight of the graph, which both turn out to be quite intuitive to compute for our new graph. For this framework we need to choose some initial distribution $\rho \in \mathbb{R}^{\overline{V}}$ of P, instead of its stationary distribution. A natural choice is the uniform distribution over all $\sigma(X_S) \in V_\emptyset$.

A vertex $\sigma(X_S) \in V$ is represented quantumly as

$$|\sigma(X_S)\rangle = |\mathsf{D}(X_S)\rangle \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle |0\rangle_{de} \tag{4.1}$$

and a vertex $Z_{\sigma(X_S),\ell,j} \in V_{de}$ as

$$\left|Z_{\sigma(X_S),\ell,j}\right\rangle = |\mathsf{D}(X_S)\rangle \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle |\ell\rangle |j\rangle |1\rangle_{de}. \tag{4.2}$$

Here D is the assumed quantum data structure and the last qubit indicates if we are dealing with a dead-end vertex or not. For simplicity we often do not write this indication qubit. Lastly, it is important to recall that $i_1 < i_2 < \cdots < i_r$.

We show how to implement each of the subroutines.

Setup We need to construct the map

$$|0\rangle \mapsto |\rho\rangle := \frac{1}{\sqrt{|V_\emptyset|}} \sum_{\sigma(X_S)\in V_\emptyset} |\sigma(X_S)\rangle = \frac{1}{\sqrt{\binom{N-2}{r} r!}} \sum_{S\in S_\emptyset} |\mathsf{D}(X_S)\rangle \sum_{\sigma\in\mathcal{S}_r} \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle .$$

First create a uniform superposition over all indices $i_1 \in [N]$. We can do so by applying $\log N = n$ Hadamards H to $\left|\overline{0}\right\rangle$, giving

$$\mathsf{H}^{\otimes n} \left|\overline{0}\right\rangle = \frac{1}{\sqrt{N}} \sum_{i_1\in[N]} |i_1\rangle .$$

We can use one query to $\mathcal{O}_x$ in superposition to obtain all the values corresponding to the indices

$$\frac{1}{\sqrt{N}} \sum_{i_1\in[N]} \mathcal{O}_x |i_1\rangle \left|\overline{0}\right\rangle = \frac{1}{\sqrt{N}} \sum_{i_1\in[N]} |i_1, x_{i_1}\rangle .$$

Next we can use one insertion U_I in superposition to store the indices and values in our data structure

$$\frac{1}{\sqrt{N}} \sum_{i_1\in[N]} U_I |i_1, x_{i_1}\rangle |\mathsf{D}(X_\emptyset)\rangle = \frac{1}{\sqrt{N}} \sum_{i_1\in[N]} |i_1, x_{i_1}\rangle \left|\mathsf{D}(X_{\{i_1\}})\right\rangle .$$

Note that we first have made use of U_{init} to get $|\mathsf{D}(X_\emptyset)\rangle$ from $\left|\overline{0}\right\rangle$. Inserting the first index does not cause any issues since the data structure was still empty. For every subsequent index, we only want to insert those indices that are not already present in the data structure. This is exactly the initial reason for needing the lookup operation.

We create a uniform superposition over all elements $i_2 \in [N]$ and use a lookup to get a bit indicating if the index i_2 is already in the data structure

$$\frac{1}{N} \sum_{i_1,i_2\in[n]} |i_1, x_{i_1}\rangle U_L |i_2\rangle \left|\mathsf{D}(X_{\{i_1\}})\right\rangle |0\rangle = \frac{1}{N} \sum_{i_1,i_2\in[n]} |i_1, x_{i_1}\rangle |i_2\rangle \left|\mathsf{D}(X_{\{i_1\}})\right\rangle |(i_2 \in \{i_1\})\rangle .$$

We now would like to amplify the indices $|i_2\rangle$ that are not already in the data structure. Normally, one would use amplitude amplification (Theorem 2.9), however, for this case, it is overkill. The fraction of indices not already in our data structure is $(N-1)/N = \Theta(1)$, so there is no need to speed this up quadratically, introducing unnecessary complexity. Instead we simply measure the indicating bit. If the outcome is 0, we remain with a superposition over all indices i_2 not already in our data structure, so then we are done. If, however, the outcome is 1 we discard the state, which now contains precisely those indices that are already in our data structure, and try again until we measure a 0. We only need to repeat this $O(N/(N-1)) = O(1)$ times with very high probability.

We now have the state

$$\frac{1}{\sqrt{\binom{N}{2} 2!}} \sum_{i_1\in[N], i_2\in[N]\setminus\{i_1\}} |i_1\rangle |x_{i_1}\rangle |i_2\rangle \left|\mathsf{D}(X_{\{i_1\}})\right\rangle .$$

We can query the values of these non-duplicate indices and insert them into our data structure to obtain

$$\frac{1}{\sqrt{\binom{N}{2} 2!}} \sum_{i_1\in[N], i_2\in[N]\setminus\{i_1\}} |i_1, x_{i_1}\rangle |i_2, x_{i_2}\rangle \left|\mathsf{D}(X_{\{i_1,i_2\}})\right\rangle .$$

Repeating this procedure for the other $r-2$ indices $i_3, i_4, \cdots, i_r$ results in the state

$$\frac{1}{\sqrt{\binom{N}{r} r!}} \sum_{i_1\in[N], i_2\in[N]\setminus\{i_1\}, \ldots, i_r\in[N]\setminus\{i_1,i_2,\ldots,i_{r-1}\}} |i_1, x_{i_1}\rangle |i_2, x_{i_2}\rangle \cdots |i_r, x_{i_r}\rangle \left|\mathsf{D}(X_{\{i_1,i_2,\ldots,i_r\}})\right\rangle .$$

Note that we are still able to 'amplify' the state in the procedure in constant time since the probability of measuring 0 is still $O((N-(r-1))/N) = \Theta(1)$. Unquerying each value gives the state

$$\frac{1}{\sqrt{\binom{N}{r}r!}} \sum_{i_1\in[N], i_2\in[N]\setminus\{i_1\},\ldots,i_r\in[N]\setminus\{i_1,i_2,\ldots,i_{r-1}\}} |i_1\rangle\,|i_2\rangle\cdots|i_r\rangle\,\left|\mathsf{D}(X_{\{i_1,i_2,\ldots,i_r\}})\right\rangle.$$

Which alternatively can be written as

$$\frac{1}{\sqrt{\binom{N}{r}r!}} \sum_{S\in\mathscr{S}} |\mathsf{D}(X_S)\rangle \sum_{\sigma\in\mathcal{S}_r} \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle.$$

We would now like to obtain only the sets that do not contain a or b, thus the $S \in S_\emptyset$. Fix some $\mathsf{D}(X_S)$ and let $f : [N] \to \{0,1\}$ be the function defined as

$$f(i) = \begin{cases} 1 & \exists j \in S \text{ such that } (j, x_i) \in X_S \\ 0 & \text{otherwise} \end{cases}.$$

If we find an i with $f(i) = 1$, then $S \notin S_\emptyset$ and if no such i exists we have $S \in S_\emptyset$. We can compute $f(i)$ for some arbitrary $i \in [N]$ as follows

$$\begin{aligned}
|\mathsf{D}(X_S)\rangle\,|i\rangle\left|\overline{0}\right\rangle|0\rangle\,|0\rangle &\mapsto |\mathsf{D}(X_S)\rangle\,|i, x_i\rangle\,|0\rangle\,|0\rangle && O_x \otimes I \otimes I \\
&\mapsto |\mathsf{D}(X_S)\rangle\,|i, x_i\rangle\,|(i \in S)\rangle\,|0\rangle && U_L \otimes I \\
&\mapsto \begin{cases} \left|\mathsf{D}(X_{S\cup\{i\}})\right\rangle|i, x_i\rangle\,|0\rangle\,|0\rangle & i \notin S \\ |\mathsf{D}(X_S)\rangle\,|i, x_i\rangle\,|1\rangle\,|0\rangle & i \in S \end{cases} && U_I^c \otimes I \\
&\mapsto \begin{cases} \left|\mathsf{D}(X_{S\cup\{i\}})\right\rangle|i, x_i\rangle\,|0\rangle\,|f(i)\rangle & i \notin S \\ |\mathsf{D}(X_S)\rangle\,|i, x_i\rangle\,|1\rangle\,|f(i)\rangle & i \in S \end{cases} && U_C \\
&\mapsto |\mathsf{D}(X_S)\rangle\,|i, x_i\rangle\,|(i \in S)\rangle\,|f(i)\rangle && U_D^c \otimes I \\
&\mapsto |\mathsf{D}(X_S)\rangle\,|i, x_i\rangle\,|0\rangle\,|f(i)\rangle && U_L \otimes I \\
&\mapsto |\mathsf{D}(X_S)\rangle\,|i\rangle\left|\overline{0}\right\rangle|0\rangle\,|f(i)\rangle && O_x \otimes I \otimes I.
\end{aligned}$$

Here U_I^c and U_D^c are controlled versions of the insert and delete operations respectively.[8] The way that we are able to compute $f(i)$ is by inserting (i, x_i) into the data structure. If there indeed exists a $j \in S$ such that $(j, x_i) \in X_S$, then adding (i, x_i) causes X_S the contain a collision. We can check for collisions with U_C, an output the result in an auxiliary bit. This bit then exactly corresponds to the value of $f(i)$.

In total, computing $f(i)$ takes times at most $O(\gamma_M + T_L + T_{ID} + T_C)$. Using Grover's search (Theorem 2.8) we can find if $f(i) = 1$ for some $i \in [N]$ in time $O(\gamma_N\sqrt{N}(\gamma_M + T_L + T_{ID} + T_C))$. Using this procedure we can obtain the state

$$\frac{1}{\sqrt{\binom{N}{r}r!}} \sum_{S\in\mathscr{S}} |\mathsf{D}(X_S)\rangle \sum_{\sigma\in\mathcal{S}_r} \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle |(S \notin S_\emptyset)\rangle.$$

Now simply measure $|(S \notin S_\emptyset)\rangle$. If the output is 1 then $S \notin S_\emptyset$, which means that $S \in S_a \cup S_b \cup S_{a,b}$. This is only possible if this set is not empty, meaning that there exists a collision. So we solved the element distinctness problem. If, on the other hand, the output is 0 we obtain the state

$$\frac{1}{\sqrt{\binom{N-2}{r}r!}} \sum_{S\in S_\emptyset} |\mathsf{D}(X_S)\rangle \sum_{\sigma\in\mathcal{S}_r} \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle = |\rho\rangle,$$

which is exactly what we wanted. In total, creating this states takes time at most

$$\mathsf{S} = O\Big(r(\gamma_M + \gamma_N + T_L + T_{ID}) + \gamma_N\sqrt{N}(\gamma_M + T_L + T_{ID} + T_C) + T_{\text{init}}\Big)$$

[8] Recall from Section 2.5.4 that we can indeed make every unitary controlled

Update First consider the vertices $Z_{\sigma(X_S),\ell,j} \in V_{de}$. The sampling map is simply the identity since $Z_{\sigma(X_S),\ell,j}$ only has one label, namely 0. From (4.2) it follows that for the transition map we can simply disregard $|0\rangle$ to obtain the desired map

$$\left|Z_{\sigma(X_S),\ell,j}\right\rangle |0\rangle \mapsto |\mathsf{D}(X_S)\rangle \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle |\ell\rangle |j\rangle .$$

Now consider the vertices $\sigma(X_S) \in V$ and recall (4.1). We need to create the following sampling map

$$|\mathsf{D}(X_S)\rangle \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle \mapsto \frac{1}{\sqrt{Nr}} \sum_{(\ell,j)\in[r]\times[N]} |\mathsf{D}(X_S)\rangle \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle |\ell, j\rangle .$$

Constructing this map is quite straightforward. Simply make a superposition over all $\ell \in [r]$ and all $j \in [N]$, using $O(\log r + \log N) = O(\log N)$ Hadamards, which costs γ_N. We again emphasise that this extra work is precisely to make this map very simple.

We can now construct the transition map as follows: first do a lookup to see if the element is already in the data structure, we get

$$|\mathsf{D}(X_S)\rangle \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle |\ell, j\rangle |(j \in S)\rangle$$

If the last qubit is $|0\rangle$, change the $|0\rangle_{de}$ to $|1\rangle_{de}$. Else insert j into the data structure by first making a query and then applying U_I and then unquerying, we get

$$\left|\mathsf{D}(X_{S\cup\{j\}})\right\rangle \left|i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}\right\rangle |\ell, j\rangle |(j \in S)\rangle .$$

Next switch j with i_ℓ by applying a QRAG (Definition 2.6), query the value for i_ℓ, delete i_ℓ from the data structure using U_D and then unquery again. We get

$$\left|\mathsf{D}(X_{(S\cup\{j\})\setminus\{i_\ell\}})\right\rangle \left|i_{\sigma(1)}, \ldots, i_{\ell-1}, j, i_{\ell+1}, \ldots, i_{\sigma(r)}\right\rangle |\ell, i_\ell\rangle .$$

Notice how this is exactly the state we want to obtain from the transition map. In total, the update cost is at most $\mathsf{U} = O(\gamma_M + \gamma_N + T_L + T_{ID})$.

Check First note that the dead-end vertices are never marked vertices and we can distinguish them from 'normal' vertices with the indicator qubit. For $\sigma(X_S) \in V$ we need to implement the map

$$|\sigma(X_S)\rangle \mapsto \begin{cases} -|\sigma(X_S)\rangle & \text{if } \sigma(X_S) \in \mathcal{M} \\ |\sigma(X_S)\rangle & \text{otherwise} \end{cases} .$$

This map can be simply obtained by applying a Z gate to the last qubit of the result of the checking map U_C on $|\sigma(X_S)\rangle |0\rangle$. We need to clear up to working qubits so we apply U_C once more. Clearly this takes time at most $\mathsf{C} = T_C$.

Analysis The total number of edges, and thus the total weight, of $\overline{G}$ is upper bounded by

$$W(\overline{G}) = |E| + |E_{de}| = \frac{\binom{N}{r} r(N-r)}{2} r! + \binom{N}{r} r! r^2 = O\left(\binom{N}{r} r(N-r) r!\right) =: \mathcal{W}.$$

We define a possible flow (Definition 3.13) for an edge $e \in \overline{E}$ as

$$\theta(e) := \begin{cases} 1/\left(\binom{N-2}{r} r r!\right) & e = (v_0, v_1) \in V_\emptyset \times V_a \\ 1/\left(\binom{N-2}{r-1} (r-1) r!\right) & e = (v_1, v_2) \in V_a \times V_{a,b} \\ 0 & \text{otherwise} \end{cases} ,$$

that 'flows' from the set of unmarked vertices to the set of marked vertices and spreads out the flow across its edges as much as possible. Let us check if this is a correct $\rho\mathcal{M}$-flow (Definition 3.16). For $u \in V_\emptyset$ we have

$$\theta(u) = \sum_{\substack{v\in\mathcal{N}(u):\\ v\in V_a}} \frac{1}{\binom{N-2}{r} r r!} + \sum_{\substack{v\in\mathcal{N}(u):\\ v\notin V_a}} 0 = \frac{1}{\binom{N-2}{r}} = \rho(u).$$

and for $u \in V_a$ we have

$$\begin{aligned}
\theta(u) &= \sum_{\substack{v\in\mathcal{N}(u):\\ v\in V_\emptyset}} -\frac{1}{\binom{N-2}{r} rr!} + \sum_{\substack{v\in\mathcal{N}(u):\\ v\in V_{a,b}}} \frac{1}{\binom{N-2}{r-1}(r-1)r!} + \sum_{\substack{v\in\mathcal{N}(u):\\ v\notin V_\emptyset\cup V_{a,b}}} 0 \\
&= -(N-1-r)r!\frac{1}{\binom{N-2}{r} rr!} + (r-1)r!\frac{1}{\binom{N-2}{r-1}(r-1)r!} \\
&= -\frac{N-1-r}{\binom{N-2}{r} r} + \frac{N-1-r}{\binom{N-2}{r} r} \\
&= \rho(u).
\end{aligned}$$

For $u \in V_b \cup V_{de}$ we immediately have $\theta(u) = 0 = \rho(u)$, since there are no non-zero flow edges coming in or out of them. It follows that we can indeed use θ as an upper bound on the effective resistance (Definition 3.17), so we get

$$R_{\rho,\mathcal{M}} \leq \sum_{(u,v)\in\vec{E}} \frac{\theta(u,v)^2}{w_{u,v}} = \frac{1}{\binom{N-2}{r} rr!} + \frac{1}{\binom{N-2}{r-1}(r-1)r!} = O\left(\frac{1}{\binom{N-2}{r-1}(r-1)r!}\right) =: \mathcal{R}.$$

It follows that

$$\mathcal{R}\mathcal{W} = O\left(\frac{N!(r-1)!(N-2-(r-1))!r(N-r)}{r!(N-r)!(N-2)!(r-1)}\right) = O\left(\frac{N(N-1)r(N-r)}{r(N-r)(r-1)}\right) = O\left(\frac{N^2}{r}\right).$$

Thus by Theorem 3.18 there exists a bounded-error quantum algorithm for element distinctness with time complexity

$$\begin{aligned}
O\Big(&r(\gamma_M + T_L + T_{ID}) + \sqrt{N}\gamma_N(\gamma_M + T_L + T_{ID} + T_C) \\
&+ \frac{N}{\sqrt{r}}(\gamma_M + T_L + T_{ID} + T_C) + T_{\text{init}}\Big),
\end{aligned}$$

where we have used that $\gamma_N = O(\gamma_M)$. By choosing $r = N^{2/3}$ we get

$$O(N^{2/3}(\gamma_M + T_L + T_{ID} + T_C) + T_{\text{init}}).$$

□

If all data structure operations are at most $O(\text{polylog}(N))$, we have that Theorem 4.8 gives a matching time complexity for element distinctness. We are, however, able to improve upon this complexity.

As we will find in Chapter 5, a quantum data structure can use the initialisation map to create an initial superposition over some random information. For example to implement a quantum hash table (see Section 5.4.1), we use U_{init} to create a superposition over all hash functions living in some family of hash functions. We can now control our operations based on these hash functions h, and the input i. The run-time T of a data structure operation then depends on the value of i and h, which we denote as $T(i,h)$. In Version 1 of the main theorem the cost of each operation is counted as $\max_{i,h} T(i,h)$, since naively one would need to 'wait' for the largest branch of the superposition to finish, before we can move on to the rest of the algorithm (see Section 2.5.5).

In Version 2 of the main theorem we show that we can incur an average-case cost over the different $T(i,h)$, thus generalising the statement of Version 1. As the cost of a check is always $O(\gamma_N)$ (see Section 5.3 and Section 5.4) and the initialise cost is additive, there is no need for us to consider the average-case complexity of these operations.

Theorem 4.9 (Main theorem (Version 2))**.** Let D be an $(N^{2/3}+1)$-sized quantum data structure and R a set of randomness. Suppose that U_{init} acts as

$$\left|\overline{0}\right\rangle \mapsto \sum_{\rho\in R} \alpha_\rho \left|\rho\right\rangle \left|\mathsf{D}^\rho(X_\emptyset)\right\rangle,$$

for some $\left|\mathsf{D}^\rho(X_\emptyset)\right\rangle$ and amplitudes α_ρ, and each of the data structure operations are controlled on ρ, then there exists a bounded-error quantum algorithm for element distinctness with time complexity at most

$$O(N^{2/3}(\gamma_M + T_L^{\text{avg}} + T_{ID}^{\text{avg}} + T_C) + T_{\text{init}}),$$

where

$$T_{ID}^{\text{avg}} := \sum_{\substack{i\in[N]\\ \rho\in R}} \frac{1}{N\cdot|\alpha_\rho|^2}\cdot T_{ID}(i,\rho) + \max_{S\in\binom{[N]}{N^{2/3}+1}} \left\{ \sum_{\substack{i\in S\\ \rho\in R}} \frac{1}{(N^{2/3}+1)\cdot|\alpha_\rho|^2}\cdot T_{ID}(i,\rho)\right\}$$

and

$$T_L^{\text{avg}} := \sum_{\substack{i\in[N]\\ \rho\in R}} \frac{1}{N\cdot|\alpha_\rho|^2}\cdot T_L(i,\rho),$$

for $T_L(i,\rho)$ and $T_{ID}(i,\rho)$ the lookup and insertion (and deletion) cost dependent on the element $i\in[N]$ you lookup or insert (or delete) respectively and some randomness $\rho\in R$.

Proof. Consider the algorithm $\mathcal{A}$ of Theorem 4.8. We are going to make use of Theorem 2.12 to get an average cost over both T_{ID} and T_L. Let us start with the former.

We have that $\mathcal{A}$ makes queries to the subroutines U_I and U_D. Since both these operations are controlled on $|i\rangle$ by Definition 4.7 and on $|\rho\rangle$ by assumption we have that

$$U_I = \sum_{\substack{i\in[N]\\ \rho\in R}} |i,\rho\rangle\langle i,\rho| \otimes U_I(i,\rho)$$

and

$$U_D = \sum_{\substack{i\in[N]\\ \rho\in R}} |i,\rho\rangle\langle i,\rho| \otimes U_D(i,\rho),$$

where $U_I(i,\rho) : \left|\mathsf{D}^\rho(X_S)\right\rangle \mapsto \left|\mathsf{D}^\rho(X_{S\cup\{i\}})\right\rangle$ and $U_D(i,\rho) : \left|\mathsf{D}^\rho(X_S)\right\rangle \mapsto \left|\mathsf{D}^\rho(X_{S\setminus\{i\}})\right\rangle$. Let $T_{ID}(i,\rho)$ be the cost of executing $U_I(i,\rho)$ and $U_D(i,\rho)$ controlled on $|i,\rho\rangle$. We consider the two different cases.

First let $|\psi_{I,q}\rangle$ be the state right before making the q^{th} query to U_I for some arbitrary q. We have that

$$|\psi_{I,q}\rangle = \frac{1}{\sqrt{N\cdot|\alpha_\rho|^2}} \sum_{S\in\binom{[N]}{s_q}} \sum_{i\in[N]} |i\rangle \sum_{\rho\in R} |\rho\rangle\otimes\alpha_S \left|\mathsf{D}^\rho(X_S)\right\rangle,$$

where $s_q\in[N^{2/3}]$ is the size of X_S in the q^{th} query and α_S is the amplitude for each S.

We now have

$$\begin{aligned}
\sum_{\substack{i\in[N]\\ \rho\in R}} ||(|i,\rho\rangle\langle i,\rho|\otimes \mathsf{I})\,|\psi_{I,q}\rangle\,||^2\cdot T_{ID}(i,\rho) &= \sum_{\substack{i\in[N]\\ \rho\in R}} ||\frac{1}{\sqrt{N\cdot|\alpha_\rho|^2}}\sum_{S\in\binom{[N]}{s_q}} |i,\rho\rangle\,\alpha_S\,|\mathsf{D}^\rho(X_S)\rangle\,||^2\cdot T_{ID}(i,\rho)\\
&= \sum_{\substack{i\in[N]\\ \rho\in R}}\sum_{S\in\binom{[N]}{s_t}} ||\frac{1}{\sqrt{N\cdot|\alpha_\rho|^2}}\,|i,\rho\rangle\,\alpha_S\,|\mathsf{D}^\rho(X_S)\rangle\,||^2\cdot T_{ID}(i,\rho)\\
&= \sum_{\substack{i\in[N]\\ \rho\in R}}\frac{1}{N\cdot|\alpha_\rho|^2}\sum_{S\in\binom{[N]}{s_t}} |\alpha_S|^2\cdot||\,|i,\rho\rangle\,|\mathsf{D}^\rho(X_S)\rangle\,||^2\cdot T_{ID}(i,\rho)\\
&= \sum_{\substack{i\in[N]\\ \rho\in R}}\frac{1}{N\cdot|\alpha_\rho|^2}\cdot T_{ID}(i,\rho),
\end{aligned}$$

where the second equality follows from that fact that $|\mathsf{D}^\rho(X_S)\rangle$ are pairwise orthogonal for all S.

Next let $|\psi_{D,q}\rangle$ be the state right before making the q^{th} query to U_D for some arbitrary q. We have that

$$|\psi_{D,q}\rangle = \frac{1}{\sqrt{s\cdot|\alpha_\rho|^2}}\sum_{S\in\binom{[N]}{s}}\sum_{i\in S}|i\rangle\sum_{\rho\in R}|\rho\rangle\otimes\alpha_S\,|\mathsf{D}^\rho(X_S)\rangle\,,$$

where $s = N^{2/3}+1$ and α_S is the amplitude for each S.[9] We now have

$$\begin{aligned}
\sum_{\substack{i\in[N]\\ \rho\in R}} ||(|i,\rho\rangle\langle i,\rho|\otimes \mathsf{I})\,|\psi_{D,q}\rangle\,||^2\cdot T_{ID}(i,\rho) &= \sum_{\substack{i\in[N]\\ \rho\in R}} ||\frac{1}{\sqrt{s\cdot|\alpha_\rho|^2}}\sum_{S\in\binom{[N]}{s}:i\in S} |i,\rho\rangle\,\alpha_S\,|\mathsf{D}^\rho(X_S)\rangle\,||^2\cdot T_{ID}(i,\rho)\\
&= \sum_{\substack{i\in[N]\\ \rho\in R}}\sum_{S\in\binom{[N]}{s}:i\in S} ||\frac{1}{\sqrt{s\cdot|\alpha_\rho|^2}}\,|i,\rho\rangle\,\alpha_S\,|\mathsf{D}^\rho(X_S)\rangle\,||^2\cdot T_{ID}(i,\rho)\\
&= \sum_{\substack{S\in\binom{[N]}{s}\\ \rho\in R}}\frac{1}{s\cdot|\alpha_\rho|^2}\sum_{i\in S}|\alpha_S|^2\cdot||\,|i,\rho\rangle\,|\mathsf{D}^\rho(X_S)\rangle\,||^2\cdot T_D(i,\rho)\\
&= \sum_{S\in\binom{[N]}{s}}|\alpha_S|^2\sum_{\substack{i\in S\\ \rho\in R}}\frac{1}{s\cdot|\alpha_\rho|^2}T_D(i,\rho)\\
&\le \max_{S\in\binom{[N]}{s}}\left\{\sum_{\substack{i\in S\\ \rho\in R}}\frac{1}{\sqrt{s\cdot|\alpha_\rho|^2}}\cdot T_D(i,\rho)\right\}.
\end{aligned}$$

We now have that T_{ID}^{avg} is an upper bound on

$$\sum_{\substack{i\in[N]\\ \rho\in R}} ||(|i,\rho\rangle\langle i,\rho|\otimes \mathsf{I})\,|\psi_{D,q}\rangle\,||^2\cdot T_{ID}(i,\rho).$$

We also have that $\mathcal{A}$ makes queries to the subroutine

$$U_L = \sum_{\substack{i\in[N]\\ \rho\in R}}|i,\rho\rangle\langle\rho,i|\otimes U_L(i,\rho),$$

where $U_L(i,\rho):|\mathsf{D}_s(X_S)\rangle\,|0\rangle\mapsto|\mathsf{D}_s(X_S)\rangle\,|(i\in S)\rangle$ in cost $T_L(i,\rho)$. Let $|\psi_{L,q}\rangle$ be the state right before making the q^{th} query to U_L for arbitrary q. Since we are always doing a lookup before we

[9] Note that we only delete from sets of size $N^{2/3}+1$.

are inserting we have that $|\psi_{L,q}\rangle = |\psi_{I,q}\rangle$.[10] Thus by similar arguments as before we have that

$$\sum_{\substack{i\in[N]\\ \rho\in R}} ||(|i,\rho\rangle\langle i,\rho| \otimes \mathsf{I})\,|\psi_{L,q}\rangle\,||^2 \cdot T_{ID}(i,\rho) = \sum_{\substack{i\in[N]\\ \rho\in R}} \frac{1}{N\cdot|\alpha_\rho|^2}\cdot T_L(i,\rho) = T_L^{\text{avg}}.$$

It now follows directly from Theorem 2.12 together with Theorem 4.8 that there exists a bounded-error quantum algorithm for element distinctness with time complexity at most

$$O(N^{2/3}(\gamma_M + T_L^{\text{avg}} + T_{ID}^{\text{avg}} + T_C) + T_{\text{init}}).$$

□

4.3 Overview of the Techniques

We conclude this chapter by discussing the four primary techniques used to prove both versions of the main theorem. Starting with the three main techniques needed for the proof of Version 1.

4.3.1 Electric network framework

The general approach of the proof was to simplify the required subroutines as much as possible by expanding the structure of the (Johnson) graph we initially walked over in Theorem 4.4. Of course, this expansion must be done in such a way as to not affect the desired complexity. The electric network framework (Section 3.3) allowed us to precisely understand how adding vertices and edges to our graph influenced the overall analysis.

Usually in the evaluation of quantum walks we are dependent on knowing the spectrum, i.e. the set of eigenvalues of the adjacency matrix, of the graph we are walking over (see Section 3.2.2). Since mostly Johnson and Hamming graphs are used, and both spectra are known, this is often not such a problem. However, as soon as one starts to make modification to graph, like adding extra vertices and edges, it is not obvious at all as to how to spectrum is effected.

Using the electric network properties, on the other hand, is much more intuitive. The total weight of the graph is generally easy to compute, in particular since for our case this corresponded exactly to the number of edges in the graph. The effective resistance, however, might seem less straightforward. Yet, if you keep in mind that the energy of a flow is minimised by spreading the flow across the graph as much as possible, the choice of flow becomes much clearer. We want to move from the unmarked vertices $V_\emptyset$ to the marked vertices $V_{a,b}$, so it makes sense to 'flow' in that direction.[11] Then to spread the flow out as much as possible we simply consider a uniform distribution over these out-going flow edges.

4.3.2 Dead-end edges

The main difficulty of the update subroutine, and the only reason for needing the diffusion operator, is the sampling map. For this map we need to create a superposition over the labels $(i,j) \in S \times ([N] \setminus S)$. By using Belovs dead-end edge trick [Bel13] we can change the label set to $S \times [N]$. The reason why we can simply add these edges is that it only affects the total weight of the graph in a negligible way and not the effective resistance.[12] A normal Johnson graph contains $\binom{N}{r}r(N-r)$ edges. This means that for every vertex, of which there are $\binom{N}{r}$ many, we can add at most $O(r(N-r))$ dead-end edges without affecting the total weight. Since a permuted Johnson graph simplify has $r!$ times more edges and vertices, this trick also worked for that graph.

[10] Technically the lookup also changes one qubit, which remains the same for the insertion (and deletion).

[11] We decided to first 'flow' to V_a and then to $V_{a,b}$, but we could have just as well first 'flowed' to V_b, as long as we go in the direction of the marked vertices.

[12] Since we are obviously not defining a flow on these edges.

As an alternative to the dead-end edge trick, one could also use amplitude amplification (Section 2.5.2) on the superposition $\sum_{j\in[N]} |j\rangle$ to get close to the desired superposition of $\sum_{j\in[N]\setminus S} |j\rangle$. The downside of this method is that it incurs unnecessary logarithmic factors. In a sense, this dead-end edges trick integrates this amplification into the quantum walk itself, omitting the logarithmic factors. Again, the ease with which we can reanalyse our quantum walk over an extended graph structure shows the intuitiveness of the electric network framework.

4.3.3 Permuted Johnson graph

Using only the dead-end edge trick does not get rid of the diffusion operator. With the label set $S \times [N]$, we still need to create a superposition over the elements in S. Hence we considered walking on the permuted Johnson graph. The benefit of this graph is that we 'walk' from one vertex to another by deleting a value at a specific coordinate, instead of for a specific value as with the normal Johnson graph. This means that the label set can be further reduced down to $[r] \times [N]$, where $\ell \in [r]$ specifies the location of the element you want to remove.

Not only can the permuted Johnson graph eliminate the diffusion operator, it also significantly simplifies the setup step. To implement the setup subroutine in the proof of Theorem 4.4 we needed to create a superposition over all possible r sized subsets $S \subseteq [N]$. The method described in the proof of the main theorem is basically the most natural way to approach this. Unfortunately we get that the data structure storing the set is entangled with all the possible permutations of the elements. To directly use the Johnson graph, we would need to unentangle these states. This is the biggest problem in creating the setup state. Getting rid of this permutation is possible, using for example a method as shown by Gilyén [PG14], but takes unnecessary time.

The beauty of the permuted Johnson graph is that we can view its state as an combination of an r-sized set and a permutation over these elements. We now have that the obtained state of the setup step is exactly the superposition over the vertices of this graph. So instead of eliminating these permutations, we simply embed these states into our graph. Since the whole graph is increased by the same $r!$ factor, the impact on the total weight and effective resistance cancel out. Thus the overall complexity is not influenced.

4.3.4 Averaging over subroutines

The main technique needed to prove Version 2 of the main theorem is to exploit the very recent result by Belovs et al. [Bel13] (see Section 2.5.5). In a sense, all the hard work has been done in the proof of Version 1 and we only need to make use of Theorem 2.12 to obtain the desired claim. Important to note is that we are allowed to take the average over both the different inputs, as-well as the different randomness. We only average over the randomness (hash functions) to construct our quantum hash table (Section 5.4.1), but we believe that for other settings averaging over the inputs might also come to use.

5 Quantum Data Structures for Element Distinctness

From the previous chapter we know that to obtain a matching[1] time complexity for element distinctness we need a quantum data structure (Definition 4.7). More specifically, by the main theorem (Theorem 4.8 and Theorem 4.9), this data structure needs to store a set X_S of size at most $N^{2/3} + 1$ in such a way that insertions, deletions, lookups and checks take at most $O(\text{polylog}(N))$ time. In this chapter we are going to explain how to achieve these desired properties using concrete instances of the previously left abstract object of a quantum data structure. We start with an overview of what is covered in each section.

5.1 We show how to store a set of integers using known classical data structures like binary search trees, skip lists, radix trees and hash tables.

5.2 We discuss the issues one can run into when trying to use classical data structures directly in the quantum setting.

5.3 We present, and slightly simplify, the two known quantum data structures for element distinctness, namely the quantum skip list and quantum radix tree.

5.4 We introduce two versions of a quantum hash table and suggest a construction for a quantum version of a binary search tree.

5.5 We reflect on the different solutions, given in the previous two sections, for the issues that arise in translating classical data structures to the quantum setting.

5.1 Classical Data Structures

For decades, data structures have been a fundamental part of classical computing.[2] Numerous structures are created for all kinds of situations, so they are a logical starting point in our search for a quantum data structure. We know by Theorem 2.10 that quantum algorithms are at least as good as classical algorithms. So, naively, one might think that we can simply use a classical data structure with the required operations and complexities, translate it to the quantum setting and obtain the desired quantum data structure. Unfortunately, this turns out to be significantly more complex (see Section 5.2).

For now we simplify the requirements and only look at classical data structures storing a set of integers $X \subseteq [M]$ of size at most $s \in [N]$. The variables here match those of Definition 4.7 to make the upcoming translation to the quantum setting more intuitive. We further only require the ability to lookup, insert and delete elements from X.[3] For both insertion and deletion we assume that only new elements are inserted and existing elements are deleted, as these condition

[1] Recall that we say a time complexity is matching if it within a polylogarithmic factor of the optimal query complexity.

[2] The concept of the first data structure already dates back to the 1940s. See the book by Knuth [Knu68, Section 2.6] for a complete overview of the history of data structures.

[3] This is known as a *dynamic search data structure*. As we will see in Section 5.3 and Section 5.4, checking for collisions only consists of checking a counter, which we can update during the insertion and deletion operations.

are required from the quantum version of these operations. The simpler setting helps to clarify the essence of each of the data structures, providing a starting point for understanding the more complicated quantum data structures that follow.

Perhaps the two simplest methods of storing a set of integers are a *linked list* and a *sorted array*. While in a linked list we have $O(1)$ insertion cost, since we are appending each new element to the head of the list, both lookup and delete require us to traverse the whole list in the worst case, inquiring a very bad $O(s)$ cost. A sorted array tries to fix the latter problem, allowing for the lookup of elements in cost $O(\log s)$, using binary search. Unfortunately, for both insertion and deletion we would need to move s elements in the worst case, again resulting in a bad cost of $O(s)$. As discussed before, these data structures are undesirable, since we would like all operations to be at most polylogarithmic in their input size.

In this section we go over several classical data structures that do achieve these desired costs. See Table 5.1 for an overview of the data structures and their respective time and space complexities.

Data Structure		**Lookup**	**Insertion**	**Deletion**	**Space complexity**
Linked list	Worst case	$O(s)$	$O(1)$	$O(s)$	$O(s \log M)$
Sorted array	Worst case	$O(\log s)$	$O(s)$	$O(s)$	$O(s \log M)$
Binary search tree	Worst case	$O(s)$	$O(s)$	$O(s)$	$O(s \log M)$
	Average case	$O(\log s)$	$O(\log s)$	$O(\log s)$	$O(s \log M)$
Radix tree	Worst case	$O(\log M)$	$O(\log M)$	$O(\log M)$	$O(s \log M)$
Skip list	Worst case	$O(s)$	$O(s)$	$O(s)$	$O(s^2 \log M)$
	Expected case	$O(\log s)$	$O(\log s)$	$O(\log s)$	$O(s \log M)$
Indexed array	Worst case	$O(1)$	$O(1)$	$O(1)$	$O(M \log M)$
Hash table	Worst case	$O(s)$	$O(1)$	$O(s)$	$O(s^2 \log M)$
	Expected case	$O(1)$	$O(1)$	$O(1)$	$O(s \log M)$

Table 5.1: An overview of the time and space complexity upper bounds of different classical data structures storing a set of integers $X \subseteq [M]$ of size at most $s \in [N]$. The red, yellow and green colours indicate the very bad, acceptable and very good complexities respectively. We use average-case and expected-case complexity to differentiate between averaging over all possible configurations of the data structure and averaging over the random choices made in the data structure, respectively.

5.1.1 Binary search tree

A *binary search tree* (BST) [Win60], as the name suggests, is a rooted tree-like structure where each *node* is allowed to have at most two (hence binary) *children*. The 'search' part relates to how these nodes are configured in the tree. At every node, its corresponding value is smaller than all of the values in its right subtree and larger than all of the values in its left subtree. If we want to search for a particular node, we can *traverse* the tree. We start at the root and compare the values along the way, moving to the left or right subtree accordingly. The number of traversed nodes heavily depends on the insertion order of the elements of the underlying set. For example when elements are inserted in a sorted order, we obtain a linked list like structure, while more random insertions result in more of a tree-like structure, see Figure 5.1. Clearly the worst-case complexity of a traversal is $O(s)$. Yet, if we average out over all the $s!$ possible permutations of inserting elements from X we find the average height to be $O(\log s)$ [CLRS09, Theorem 12.4].

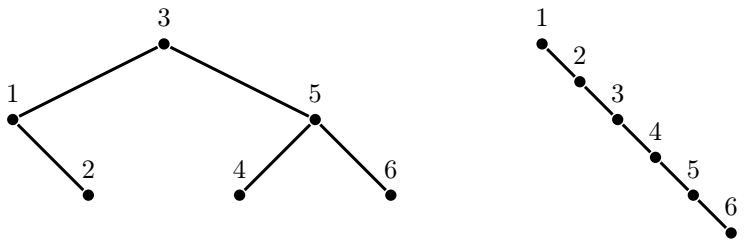


Figure 5.1: Two possible representations of a BST storing the set $\{1, 2, 3, 4, 5, 6\}$.

A BST is *balanced* if the height of the tree is minimised. For a BST storing X of size s, the minimum height is $\lfloor \log(s) \rfloor + 1$. When $s = 2^h - 1$, for some $h \in \mathbb{N}$ we say that the BST is *perfectly balanced* or *fully balanced* with height h. In that case each node, except for the leaves, has exactly two children. One way to reduce the complexity of the traversal, is to keep the tree as balanced as possible. We will not be considering these *self-balancing binary search trees*, but they could serve as an interesting future work direction.

Let us now discuss the time complexities of each of the operations and the space complexity of storing the set X in a BST. For this, let $x \in [M]$ be some arbitrary integer.

Lookup Start at the root and compare its value r to x. If $r = x$ we can stop and output true. If on the other hand $r > x$ or $r < x$ we move to the left or right child of the root respectively. We continue this process for each of the subsequent child nodes viewed as roots for their relative subtree. If we reach a leaf whose value is not equal to x we know that the value x is not present in the tree, so we output false. As this operation is basically a traversal of the tree, we have that the worst-case cost is $O(s)$ and the average-case cost is $O(\log s)$.

Insertion We traverse the BST until we reach a leaf node. Depending on whether x is larger or smaller than the value of the leaf, we create a new right or left child having the value x, respectively. Thus new elements are always inserted at the leaves. The dominating cost is again the traversal, which we know takes $O(s)$ in the worst case and $O(\log s)$ in the average case.

Deletion We look up element x in the BST using the procedure above, leaving us with three cases:

(1) If x is a leaf, we can simply delete x directly by removing the corresponding node together with the edge going into it.

(2) If x has one child with value y, we exchange x and y and delete the node with value x using method (1), since it is now a leaf.

(3) If x has two children, we need to find the in-order successor of x. Since we know that x has a right child, the in-order successor y of x is simply the left most node in the right subtree of x. So switch the values x and y and once more delete the node with value x using (1).

The trick of the deletion operation is to move the value x to a leaf node without disrupting the structure of the tree. The complexity of deleting an element is completely determined by the lookup cost. In the first two cases we simply need to find the element x and then perform some constant number of operations. For the last case we also need to find the in-order successor, which can basically be seen as a lookup in the corresponding subtree. Thus the cost of deletion is once more $O(s)$ in the worst case and $O(\log s)$ in the average case.

Space complexity We need s memory blocks in the worst case to store each of the nodes. Each block contains a tuple (z, p_l, p_r), where $z \in [M]$ is the value of the node and $p_l, p_r \in [0, \log s]$

the pointers to the left and right children respectively. In total this requires at most $O(s \log M)$ space.

5.1.2 Radix tree

Instead of comparing integers directly, we can also consider comparing their underlying bit strings. Normally when translating from binary to decimals, the all-zero string[4] represents the integer 0. However, since we are dealing with the set $[M]$, we would like start counting from 1. Hence from now on, whenever we talk about the bit string representation of some integer $x \in [M]$ we mean the string corresponding to $x - 1$.

A common method for storing strings is using *tries*[5], which are a type of search tree. Unlike a BST, elements of a binary trie are stored only in the leaves and the internal nodes do not have a specific value. The structure of the trie is determined by the labels given to the edge. More specifically we have that the edges are labelled by some string such that the concatenation of the labels along a path from the root to a leaf yields the value stored in the leaf. Thus the children of each parent share a common prefix with the label of the edge coming into the parent. We can optimise the number of internal nodes needed by labelling each edge with the largest common prefix of their children. Such a space-optimised binary trie is known as a *binary radix tree* or simply a *radix tree* [Mor68].[6]

Let us look more precisely at how to store the set X in a radix tree. We first find the longest common prefix $p \in \{0,1\}^{\leq \log M}$ of X and label the edge going into the root with p. In case of no common prefixes we have that p equals the empty string ϵ. Let $X(p)$ be the set of strings in X having p as a prefix, there are two cases:

- If $|X(p)| = 1$ there is only one element to store, so the root node is a leaf that we label with the only $i \in X(p)$.
- If $|X(p)| > 1$, the root has two children, one labelled by the longest common prefix $p_0 \in \{0,1\}^{\leq \log M}$ of $X(p \parallel 0)$ and the other by the longest common prefix $p_1 \in \{0,1\}^{\leq \log M}$ of $X(p \parallel 1)$.

We now view the left and right children as root nodes storing the sets $X(p_0)$ and $X(p_1)$ respectively and repeat the above two cases for all subsequent sets. See Figure 5.2 for an example of a radix tree.

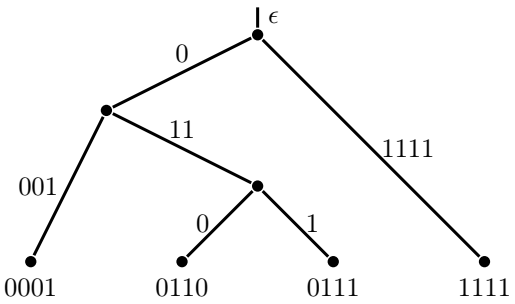


Figure 5.2: A radix tree storing the set $\{2, 7, 8, 16\} \equiv \{0001, 0110, 0111, 1111\}$.

[4] We often drop the word *bit* in bitstring whenever possible.

[5] This is not a spelling mistake: *trie* revers to the efficient re*trie*val of keys sharing a common prefix. Other names include a *digital tree* and *prefix tree*, of which the latter should not be confused with the *prefix-sum tree* used in Section 5.3.2.

[6] This was first known as a Patricia tree.

Let us again discuss the time complexities of each of the operations and the space complexity of storing the set X in a radix tree. This time let $x \in [M]$ be some arbitrary string.[7]

Lookup Traverse the tree starting from the root by taking each edge such that the concatenation of the edges along the corresponding path form a prefix of x. Stop once such an edge does not exist anymore. If we stop in a leaf node we know that $x \in X$ so we output true, else we have that $x \notin X$ and output false. Traversing the tree takes $O(\log M)$ time, since that is the number of bits needed to store an element in X and thus the maximum number of labels that can be compared.

Insertion Try to lookup x in the radix tree with the procedure above. Recall that for the insertion operation we assume that $x \notin X$. Hence we know that we will stop at a non-leaf node, meaning that one the labels of its outgoing edges must share some non-empty prefix with the remainder of x. Here with remainder we mean the substring of x without the prefix of the already traversed path. Let e be this edge. We split e up into two new edges e_1, e_2 with a new node n in the middle. The label of e_1 consists of the longest common prefix of the remainder of x with the label of the old edge e. The label of e_2 is the remainder of the label of e. We now create a new leaf node n_i that we connect to n with an edge whose label is the remainder of x, but without the label of e_1 as a prefix. Creating these new nodes and edges can be done in $O(1)$ time, but since we need to traverse the tree first, insertion takes time at most $O(\log M)$.

Deletion The deletion of x from the radix tree is basically the inverse of the above insertion procedure. Since for the deletion procedure we assume that $x \in X$ we are guaranteed to end up in a leaf. We now remove the leaf and the corresponding edge going into it. Finally, we combine the edge of the parent node and its other child node into one edge which is the concatenation of the labels of the two edges. Again because the traversal is the most expensive step, deletion also takes time $O(\log M)$.

Space complexity In the worst case there are $2s-1$ total nodes: s leaf nodes and $s-1$ internal nodes. For simplicity, let us assume that there are $2s$ memory blocks. Each block stores a node represented by the tuple (z, p_l, p_r). Here $z \in \{0,1\}^{\leq \log M}$ is the label from the edge coming into the node and $p_l, p_r \in [0, 2s]$ are pointers to the memory blocks of the left child and right child respectively. In total this requires $O(s \log M)$ space.

5.1.3 Skip list

A *skip list* [Pug90] is a more recent data structure that combines the benefits of both linked lists and sorted arrays in an alternative way. Different from tree-like structures, a skip list consists of multiple levels of linked lists. The lowest level is a sorted linked list that contains all elements, while each subsequent level contains a decreasingly smaller subsets of these elements. Levels are connected by pointers between equal elements and the first element of each level is connected to some starting point. We have that each higher level acts a *shortcut* or *express lane* for the levels below, which allows for efficient lookups by *skipping* large parts of the lowest linked list.[8] As to how efficient this lookup is, that depends on the distribution of the elements across the different levels.

In a *perfect skip list*, each i^{th} level l_i contains shortcuts for every i^{th} element in the list below, see Figure 5.3 for an example. If we take our set X of size s, there would be $O(\log s)$ levels in total. Looking up an element now takes $O(\log s)$, since we take at most one step at each level.[9] Unfortunately, inserting or deleting an element still requires us to update in the worst case $O(s)$ elements, since we need to maintain the perfect structure for all the elements.

[7] We change between the bit string and decimal representations of an integer depending on the context.
[8] Hence the name skip list.
[9] If we take two steps at a level, then we could have actually done it as one step one level higher.

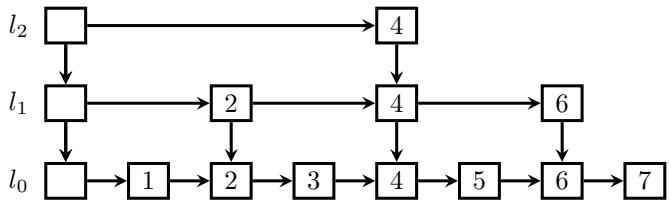


Figure 5.3: A perfect skip list storing the set $\{1, 2, 3, 4, 5, 6, 7\}$.

To address this problem, one can use a probabilistic approach to determine the level at which each element is stored. We say an element at level l has a $1/2$ probability of also being included in level $l + 1$.[10] Thus an element has $1/2^i$ probability of being included in the i^{th} level. Note that there are not necessarily $O(\log s)$ levels, since technically an element can span s levels.

Once more let $x \in [M]$ be some arbitrary integer. We describe the complexities of each of the operations on the probabilistic skip list and the space complexity of storing X. We want to highlight the difference between the expected run-time as given here and the average-case run-time as described for the BST (Section 5.1.1). For the skip list we average over the randomness of the data structure itself, while for the BST we average over all possible configurations of the data structure. Both are considered as average-case complexity throughout the literature, however we want to emphasise the difference between the two. We use *expected-case* complexity for the former method and *average-case* complexity for the latter.

Lookup Traverse the highest level $l_{\max}$ until we find the last element $y_{l_{\max}} \leq x$. If they are equal, then we output true, else move one level down to $l_{\max-1}$ and traverse the level from $y_{l_{\max}}$ until you find the last element $y_{l_{\max-1}} \leq x$. Again check if the elements are equal and else move down accordingly. Continue this process until you have traversed the last level l_0. The expected run-time of this operation depends on the expected number of horizontal steps in each level and the number of levels.

In the worst case we have that every node span every level, so the lookup cost is the same as in a linked list, namely $O(s)$. For the expected cost, we can assume that there are a maximum of $O(\log s)$ levels.[11] The number of vertical steps in then obviously $O(\log s)$. The expected number of horizontal steps in a level is 1, since we 'move' to a higher level with probability $1/2$. Thus the expected lookup cost is $O(\log s)$.

Insert Try to lookup x in the skip list. Since we know that $x \notin X$, so we end the traversal in level 0 at the closest predecessor of x. Now simply insert a new node there and include it in each subsequent level with probability $1/2$. In the worst case this node spans every level, so we need to update at most s pointers of predecessor nodes of x at each level. The expected number of levels that x will be included in, on the other hand, is $O(\log s)$. Since we found the predecessor of x in each level already with the lookup, updating these pointers can be done in $O(1)$. Giving the total insertion time of $O(\log s)$ in the expected case and $O(s)$ in the worst case.

Delete For deletion we once more lookup x, which by assumption we will find in some level l_j. Now change the pointer of the predecessor of x to the pointer going out of x and move down to level l_{j-1} (if possible). Continue until you reach level l_0. Again this takes expected time $O(\log s)$ and $O(s)$ in the worst case.

Space complexity In the worst case each node span every level, so we need s^2 memory blocks. Each block stores a node represented by the tuple (z, p_n, p_l), where $z \in \{0, 1\}^M$ is the value of

[10] Any probability p would suffice, but $1/2$ and $1/4$ are most commonly used.
[11] The expected number of elements in the levels higher than $\log s$ is only 2.

the node and $p_n, p_l \in [0, s^2]$ are pointers to the memory blocks of the next element and the lower level respectively. Thus the space complexity is $O(s^2 \log M)$ in the worst case. In contrast, the expected number of nodes summed over all levels is

$$s \cdot \frac{1}{2^0} + s \cdot \frac{1}{2^1} + s \cdot \frac{1}{2^2} \cdots = s \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2s,$$

as it is a simple geometric series. Meaning that the expected space complexity is $O(s \log M)$.

5.1.4 Hash table

The even simpler, and naive, way to store X that we have not covered yet is by using an array indexed by the unique keys in $[M]$. This way each possible integer in X has some fixed position in the array and we can access that position using a random access gate on the integer value. As a result we have a constant $O(1)$ insert, delete and lookup cost using this method. However, it is far from optimal because it requires $O(M \log M)$ space. *Hash tables* provide a solution while keeping the complexities the same in the expected time case. In a sense they act as balance between the time and space complexities.

Instead of using the key values directly as indices, we use a *hash function* $h : [M] \to [B]$ to map the *universe* $U = [M]$ to some *table* of B *buckets* with $B < M$. The resulting hash table only requires $O(B \log M)$ space. The downside is that, due to the pigeonhole principe, some elements in X_S are mapped to the same bucket since $B < M$; this is called a *collision*.[12] There are a multitude of ways to deal with collisions, but a common method is *separate chaining*, where colliding elements are stored in a linked list. The maximum size of the linked list that we allow is called the *height* of the bucket.

The beauty of hashing is that operations like insert, delete and lookup on some $x \in [M]$ become incredibly easy. Simply compute $h(x)$ and execute the required operation on the linked list given by the $h(x)^{\text{th}}$ bucket. Clearly the complexity of these operations depends on the size of the linked list. This size is directly determined by the number of collisions in a bucket, so a good hash function keeps this as low as possible. We additionally would like the number of buckets to be roughly $O(s)$, as to not take up too much extra space compared to the elements we want to store.

Unfortunately, if $M \geq (s-1)s + 1$, then there always exists a set X whose s elements all map to the same bucket, again due to the pigeonhole principle. As a result, the time complexity of delete and lookup becomes $O(s)$ in the worst case, which is far from desirable. Thus if we fix some hash function h, an adversary would be able to choose the pre-image of a bucket, causing all elements to land in the same bucket, making the hash table essentially useless. To combat this problem one can use some randomness in picking h, with the goal of obtaining good complexities in expectation.

We are going to pick a hash function h uniformly at random from a *family of hash functions* $\mathscr{H}$. We require this family to adhere to the following conditions:

1. A small number of collisions in expectation for every X.
2. Efficient sampling from $\mathscr{H}$.
3. Every $h \in \mathscr{H}$ should be efficiently computable and have low space complexity.

We can satisfy the first condition with the following property, first introduced by Carter and Wegman [CW79].

[12] This should not be confused with the collision as defined for element distinctness (see Section 4.1).

Definition 5.1 (Universality [CW79])**.** Let $\mathscr{H} = \{h : U \to [B]\}$ be a family of hash functions. We say $\mathscr{H}$ is *universal* if for all $x, y \in U$ such that $x \neq y$ we have

$$\Pr[h(x) = h(y)] \leq \frac{1}{B},$$

where the probability is taken over the uniform choice of $h \xleftarrow{\$} \mathscr{H}$.

Theorem 5.2 ([CW79, Proposition 2])**.** Let $\mathscr{H}$ be a universal hash function and $C_h(x) := \{y \in X \setminus \{x\} \mid h(x) = h(y)\}$ be the number of collisions for an element $x \in X$ and $h \in \mathscr{H}$. We have that

$$\mathbb{E}[|C_h(x)|] \leq \frac{s-1}{B},$$

where the probability is taken over the uniform choice of $h \xleftarrow{\$} \mathscr{H}$.

Proof. By the universality of $\mathscr{H}$ and the linearity of the expectation we have

$$\mathbb{E}[|C_h(x)|] = \sum_{y \in X : x \neq y} \Pr[h(x) = h(y)] \leq \sum_{y \in X : x \neq y} \frac{1}{B} = \frac{s-1}{B}$$

□

If we indeed take $B = O(s)$, then the expected number of collisions in each bucket becomes $O(1)$. The questions now remains: does there exist a universal family of hash functions? Fortunately, the answer is yes.

Theorem 5.3 ([CW79, Proposition 7])**.** Let $p \geq M$ be a prime number, $\mathbb{F}_p$ be a prime field and B be the bucket size. The family of hash functions over the universe $U = [M]$ defined as

$$\mathscr{H} := \{h(x) = ((ax + b) \mod p) \mod B \mid a, b \in \mathbb{F}_p, a \neq 0\},$$

is universal.

Proof. Pick $h \xleftarrow{\$} \mathscr{H}$ uniformly at random and let $x, y \in U$ such that $x \neq y$. We have that $h(x) = h(y)$ if and only if

$$ax + b \equiv ay + b + i \cdot B \mod p,$$

for some $i \in [0, \lfloor (p-1)/B \rfloor]$. These values of i ensure that $i \cdot B < p$, so that term is unaffected by the modulo p. Since $p > B$, larger integer values of i could result in the term $i \cdot B$ not being cancelled after the modulo B, resulting in $h(x) \neq h(y)$. We are allowed to rewrite the above into

$$a \equiv i \cdot B(x-y)^{-1} \mod p,$$

since $x \neq y$, thus $x - y \neq 0$ and $p \geq M$ meaning that $x - y$ has a inverse modulo p. There are $p - 1$ possible values for a since $a \neq 0$ and $\lfloor (p-1)/B \rfloor$ possible non-zero values for the right term. It follows that

$$\Pr[h(x) = h(y)] = \frac{\lfloor (p-1)/B \rfloor}{p-1} \leq \frac{1}{B},$$

thus $\mathscr{H}$ is universal. □

The above hash functions also have the added benefit that they are very simple to compute. They only need some basic arithmetic word operations like addition, multiplication and modulus. To sample a hash function uniformly at random we simply need to generate a uniform at random $a, b \in \mathbb{F}_p$ with $a \neq 0$. Thus the above family also satisfies condition 2 and 3.

Now for the last time let $x \in [M]$ and let us discuss the time complexities of each of the operations and the space complexity of storing X is in a hash table with $B = O(s)$.[13]

Lookup Compute $h(x)$ and lookup x in the linked list. In the worst case the height of the bucket (and thus the size of the linked list) is s, so the complexity is $O(s)$. However, due to Theorem 5.2 the expected-case complexity is $O(1)$.

Insert Compute $h(x)$ and insert x in the linked list. Since linked list insertions already take $O(1)$ steps, we have both a worst-case and expected-case cost of $O(1)$.

Delete Once more compute $h(x)$ and delete x from the linked list. By a same reasoning as for the lookup we have a worst-case complexity of $O(s)$ and an expected-case complexity of $O(1)$.

Space complexity There are $B = O(s)$ buckets, each needing to hold s memory blocks in the worst case. Each node correspond to a tuple (z, p_n), where $z \in [M]$ is the value to store and $p_n \in [0, s^2]$ a pointer to the next element in the linked list. Thus the worst case space complexity is $O(s^2 \log M)$. However, since by Theorem 5.2 the height of each bucket is constant, the expected space complexity is $O(s \log M)$.

Finally we have that in some situations we require an even stronger property from our family of hash functions.

Definition 5.4 (d-wise independence [WC81])**.** Let $\mathscr{H} = \{h : U \to [B]\}$ be a family of hash functions. We say $\mathcal{H}$ is *d-wise independent*, also known as d-universal, if for all pairwise distinct $x_1, x_2, \ldots, x_d \in U$ and $i_1, i_2, \ldots, i_d \in [B]$ we have

$$\Pr[h(x_1) = i_1, h(x_2) = i_2, \ldots, h(x_d) = i_d] = \prod_{j \in [d]} \Pr[h(x_j) = i_j] = \frac{1}{B^d},$$

where the probability is taken over the uniform choice of $h \xleftarrow{\$} \mathcal{H}$.

This property basically says that $h(x_1), h(x_2), \ldots, h(x_d)$ are independent random variables. It is not hard to see that the family of hash functions $\mathscr{H}$ given in Theorem 5.3 is in fact 2-wise independent, but not 3-wise independent.[14] One can easily extend $\mathscr{H}$ to become d-wise independent.

Theorem 5.5 ([WC81])**.** Let $d \in \mathbb{N}$, $p \geq M$ be a prime number, $\mathbb{F}_p$ be a prime field and B be the bucket size. The family of hash functions over the universe $U = [M]$ defined as

$$\mathscr{H} := \{h(x) = ((a_0 + a_1 x + a_2 x^2 + \cdots a_{d-1} x^{d-1}) \mod p) \mod B \mid a_i \in \mathbb{F}_p, a_i \neq 0\},$$

is d-wise independent.

5.2 From Classical to Quantum Data Structures

As hinted in the previous section, we cannot directly translate classical data structures to the quantum setting. As discussed in [Amb03, BJLM13, PG14], there are several properties quantum data structures should adhere to, differing from classical data structures:

- Data structure operations need to be reversible.

[13] Once more we emphasise the difference between the expected-case complexity considered here, and the average-case complexity used in evaluating the BST (Section 5.1.1).

[14] If we are given $i_1 = h(x_1)$ and $i_2 = h(x_2)$ we have that $a = (i_1 - i_2)(x_1 - x_2)^{-1}$ and $b = i_1 - x(i_1 - i_j)(x - y)^{-1}$. Thus $h(x_3) = (i - j)(x_3 - x_1)(x_1 - x_2)^{-1} + x_1$. This is not possible with only $h(x_1)$.

- The data structure needs to be unique in its representation.
- An operation on our data structure must run in some fixed time t. Operations that take average time t cannot be used directly.

The first requirement is a fundamental characteristic of quantum computing in general. For insertion, deletion and lookup we can simply remember the input in an extra qubit, as shown in Definition 4.7, to make the operations reversible. The other two requirement are a bit more subtle.

5.2.1 Uniqueness

We require quantum data structures to be unique in their representation to let interference happen. With uniqueness we mean that the overall structure is not influenced by the order of insertions and deletions. Take for example a BST (Section 5.1.1) storing the set $A := \{1, 2, 3\}$. We find that there are three possible tree structures, depending on the order in which the elements are inserted (see Figure 5.4). Let $|T_1(A)\rangle, |T_2(A)\rangle$ and $|T_3(A)\rangle$ be the quantum states storing each of the three BST representations. Although these are three different quantum states, all of them represent the same set A. During quantum algorithms, identical states are often cancelled. This cancelation is called (destructive) *interference* and is crucial for obtaining quantum speed-ups, for example in quantum walks. Due to these different representations, however, we could be in the situation where we have $|T_1(A)\rangle - |T_2(A)\rangle$ as part of our state. These states technically represent the same set, so you want them to interfere. However, they are fundamentally different objects so the desired interference does not occur.

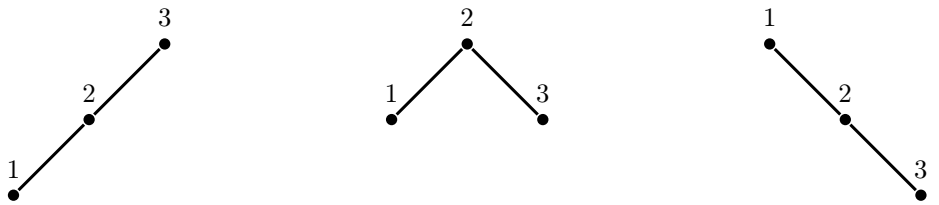


Figure 5.4: All possible binary search trees storing the set $\{1, 2, 3\}$.

The most natural solution is to require the classical data structures to have a unique representation already. A sorted array is a simple example of such a unique data structure. Note, however, that this solution is not enough. Despite having a unique representation, it could still be the case that there are several different ways of storing the data structure in memory. Take for example a sorted linked list storing the set $\{1, 2\}$. Clearly the data structure representation is unique, however, depending on the insertion order, the memory either looks like $[1, 2]$ or $[2, 1]$. Again this leads to different quantum states representing the same object.

Thus we would need a classical data structure with a unique representation and additionally some unique way of storing it in memory.[15] We call this the *uniqueness of representation problem* and the *uniqueness of memory problem* respectively. Note that the uniqueness constraint is already baked into the definition of a quantum data structure (Definition 4.7), since the mapping is well-defined.

5.2.2 Worst-case limitation

The standard method of translating classical algorithms to the quantum setting is by first expressing the algorithm as a classical circuit (Section 2.5.3). The size of this circuit reflects the

[15] Note that we could easily make any memory representation unique by using an indexed array as least as large as the universe (see Section 5.1.4). This way, every element you would want to ever store has a unique location in the array. Obviously this solution is undesirable, as it takes up way too much space.

worst-case complexity of the corresponding algorithm. Many data structure operations, however, provide only good average-case or expected-case performance (see Table 5.1). They either take an average over all possible permutations of an input (BSTs) or introduce randomness in their implementation (skip lists and hash tables), respectively. Classically, for the expected-case, we can get a bounded-error worst-case complexity by using Markov's inequality; run the algorithm for a constant times the expected run-time and only with small probability will the algorithm fail. For the quantum setting this is surprisingly less obvious.

To introduce randomness in a quantum algorithm, we need to make a superposition. For example, for the hash table, we would need to create a uniform superposition over all hash functions in some family of hash functions. In evaluating classical subroutines on the superposition of hash functions, we are required to 'wait' for the hash function of the superposition with the longest run-time to finish, if the circuit is fixed beforehand. Hence it seems that we are limited to only consider the worst-case time complexity of the operations we use on our quantum data structure. We call this the *worst-case limitation problem.*

A recent paper by Jeffery [Jef23], however, shows in a very non-trivial way that we can take an average over all branches of the superposition instead of having to wait for the longest branch to finish. This technique is generalised by Belovs et al. [BJY23], whose statement was already presented in Section 2.5.5. We showed in Version 2 of our main theorem (Theorem 4.9) how to use this technique to obtain an average over the lookup and insertion (and deletion) cost. We will see in Section 5.4 that this theorem is crucial for solving the worst-case limitation problem for hash tables.

5.3 Known Quantum Data Structures

We can now finally explore how to concretely store X_S for arbitrary $S \in \binom{[N]}{\leq s}$ in a quantum data structure and use it to obtain a matching time complexity for element distinctness. In this section we explain the two known quantum data structures used for element distinctness: a quantum skip list [Amb03] and a quantum radix tree [BJLM13, Jef14, BLPS22]. Our presentation differs from that of the original papers by using a more formal approach, needed to match with our definition of a quantum data structure (Definition 4.7). Additionally, both data structures are simplified slightly since we do not need the diffusion operator anymore due to Theorem 4.8.

5.3.1 Quantum skip list

In his paper on element distinctness, Ambainis did not only show an optimal query complexity, but he also came up with the first quantum data structure needed to obtain a matching time complexity [Amb03]. To satisfy all requirements of Section 5.2 he used a somewhat ad-hoc data structure, namely that of a hash table (Section 5.1.4) combined with a skip list (Section 5.1.3). We refer to this as the *quantum skip list* data structure, as the hash table is primarly used as a memory representation.

Skip list The main data structure allowing for efficient insertions, deletions and checks is a skip list. This should be surprising, since we know from Table 5.1 that skip lists only have good expected-case performance. Moreover, the probabilistic nature of skip lists mean that they are far from unique in their representation.[16] Using non-trivial analysis and tricks, Ambainis is able to overcome both these concerns.

His skip list consists of $\log(N)+1$ levels l_j for $j \in [0, \log(N)]$. Each entry, of each level, of the skip list stores some $(i, x_i) \in X_S$ and a pointer to the index of next element $j \in S$. There is no need for the pointer to store an actual address of a memory location, as this is taken care of by the hash table we describe later. We also do not need a pointer to some lower level for a similar reason. Crucially, the skip list is sorted first in the order of increasing x_i (and in increasing

[16] A perfect skip list is unique in its representation, however it has bad time complexities.

i if there are values with the same index). This way of sorting allows for easier collision detection.

There is a simple trick to make the representation unique: for each $i \in [N]$ we just assign some random level l_i with probability $1/2^i$ beforehand. Storing the levels for each of the i would, however, take at least $\Omega(N)$ space and choosing the levels would take at least $\Omega(N)$ time, both are undesirable. To fix this issue, Ambainis decided to define the levels by using $\log(N)$ hash functions[17] $h_1, h_2, \ldots, h_{\log N} : [N] \to \{0,1\}$ picked uniformly at random from a d-wise independent family of hash functions $\mathscr{H}$ (Definition 5.4), for $d = 4\log(N) + 1$. We say that an $i \in S$ belongs to a level $l_j < l_{\log N}$ if $h_1(i) = \cdots = h_j(i) = 1$, but $h_{j+1}(i) = 0$ and to level $l_{\log N}$ if $h_1(i) = \cdots = h_{\log N}(i) = 1$. If we consider the family of hash functions of Theorem 5.5 the time complexity of computing a single hash function is $O(d\gamma_N)$.[18] With Horner's method [Hor19] we only need d multiplications and d additions to compute a polynomial of degree d, which are both word operations on $\log N$ bits (see Section 2.4.4). Thus in total determining all levels takes time $O(\gamma_N \log^2 N)$. The space taken up by the hash function is $O(d \log N \log M) = O(\log^2 N \log M)$. If for any operation on the skip list we need to access more than $O(\log^2 N)$ pointers, the data structure will throw an error and abort.

Hash table Recall that the uniqueness problem is two-fold, both the data structure representation and its memory representation need to be unique. To deal with the uniqueness of memory problem, Ambainis decided to use a hash table.[19] Instead of storing each node in the skip list separately, we can combine nodes spanning several levels by keeping track of a list of $\log N + 1$ pointers, one for each level.

Now let $h : [N] \to [s]$ be the hash function defined as $h(i) = \lceil i \cdot s/N \rceil$ mapping the elements $(i, x_i) \in X_S$ to s buckets depending on the index i.[20] We fix each bucket to have a height of $\log N$ and the hash table will throw an error and abort whenever the bucket overflows. Since multiple items can be mapped to the same bucket, this ensures that the worst-case lookup time is at most $O(\log N)$. Within each bucket, the (i, x_i) are placed in a sorted array in increasing order of i instead of using a linked list. The sorting guarantees that the representation of the hash table is unique, something that is not true for linked lists. As explained, each entry of the bucket also contains memory for at most $\log(N) + 1$ pointers to other entries.[21] Recall that these pointers are given by the index $j \in S$ of the next element. We can find the corresponding $(j, x_j) \in X_S$ by simply computing $h(j)$ and traversing the $h(j)^{\text{th}}$ bucket.

We can now define a mapping from X_S to some Hilbert space. Let $\mathscr{N}_{\mathsf{SL},s}$ be the set of valid nodes of the hash table, holding the skip list storing X_S of size at most s, defined as

$$\mathscr{N}_{\mathsf{SL},s} := [N] \times [M] \times [0, s \log N]^{\log(N)+1}.$$

Each node corresponds to a tuple $(i, x_i, p_1, p_2, \ldots, p_{\log(N)+1})$, where (i, x_i) is the element to store and $p_1, p_2, \ldots, p_{\log(N)+1}$ are pointers to the next index in the skip list for each level. Define the

[17] Note that we are only using a property of hash functions here, this is not another hash table. We also only need $\log N$ functions, since level l_0 contains all elements by definition.

[18] Originally, Ambainis counted the cost of computing a single hash function as $O(d \log^2 N)$ [Amb03, Theorem 1], but based on the citation given it is unclear which d-wise independent family of hash functions he sampled from.

[19] As explained in Section 5.1.4, a hash table is essentially a space optimised version of the large array memory representation suggested in the footnote of Subsection 5.2.1

[20] Actually, Ambainis used the hash function $h : [N] \to [s]$ defined as $h(i) = \lfloor i \cdot s/N \rfloor + 1$. However, this function is not defined on s, but on $s + 1$ buckets. Perhaps he used the sets $[N]$ and $[s]$ to start from 0 and then the function would be well-defined, although this seems unlikely, as the notation of $[\cdot]$ is standardly defined to start from 1. This issue is of course non-significant, but it is good to highlight, as another paper uses a similar data structure with the same hash function [ACL+19].

[21] In the original paper, Ambainis also needed to store $\lfloor \log(s) \rfloor$ counters in each bucket for the implementation of the diffusion operation, which are not needed anymore. The space improvement of $O(s \log s)$ is negligible asymptotically, but it does tremendously simplify the presentation of the data structure. Apparently the irrelevancy of these counters was also noticed by Aaronson et al. [ACL+19] stating that it is "(...) easily handled in the case of a quantum walk on a Johnson graph". Confusingly in their proof, however, the diffusion operation is still needed.

Hilbert space

$$\mathcal{H}_{\mathsf{SL},s} := (\mathrm{span}\{|\nu\rangle \mid \nu \in \mathcal{N}_{\mathsf{SL},s} \cup \{\overline{0}\}\})^{\otimes s \log N},$$

and the vector $\left|\mathsf{SL}(X_S)_{h_1,\ldots,h_{\log N}}\right\rangle \in \mathcal{H}_{\mathsf{SL},s}$ where $\mathsf{SL}(X_S)_{h_1,\ldots,h_{\log N}}$ is the encoding of the skip list storing X_S in the hash table as described above, for some fixed hash functions $h_1, \ldots, h_{\log N} : [N] \to \{0,1\}$.

Quantum skip list The mapping as described before would not suffice as a quantum skip list, since quantumly we cannot choose the hash functions $h_1, \ldots, h_{\log N}$ uniformly at random. For this we would need to maintain a uniform superposition over all possible choices. Additionally, to ensure an efficient checking operation we keep track of a counter that counts the number of collisions in the current X_S being stored. We can now define the quantum version of a skip list for element distinctness.

Definition 5.6 (Quantum skip list)**.** Let $s \in [N]$ and let $\mathscr{H}$ be the family of $(4\log(N)+1)$-wise independent hash functions (Definition 5.4) with $U = [N]$ to $B = \{0,1\}$. The s-sized *quantum skip list* QSL is a family of mappings from $\{X_S \mid S \in \binom{[N]}{\leq s}\}$ to the Hilbert space

$$\mathcal{H}_s := \mathcal{H}_{\mathsf{SL},s} \otimes \mathrm{span}\{|h_1, \ldots, h_{\log N}\rangle \mid h_1, \ldots, h_{\log N} \in \mathscr{H}\} \otimes \mathrm{span}\{|c\rangle \mid c \in [0,s]\}$$

defined as $X_S \mapsto |\mathsf{QSL}(X_S)\rangle$, where

$$|\mathsf{QSL}(X_S)\rangle := \frac{1}{\sqrt{\binom{\log N}{|\mathscr{H}|}}} \sum_{h_1,\ldots,h_{\log N} \in \mathscr{H}} \left|\mathsf{SL}(X_S)_{h_1,\ldots,h_{\log N}}\right\rangle |h_1, \ldots, h_{\log N}\rangle |c\rangle .$$

Theorem 5.7. For all $s \in [N]$ and $\mathscr{H}$ some universal family of $(4\log(N)+1)$-wise independent hash functions, the s-sized quantum skip list QSL is a quantum data structure (Definition 4.7). Furthermore we have $T_{\text{init}} = O(\text{polylog}(N)), T_L = O(\gamma_N \log N)$, $T_{ID} = O(\gamma_M \log^3 N)$ and $T_C = O(\gamma_N)$.

Proof. Consider $\mathscr{H}$ to be a family of $(4\log(N)+1)$-wise independent hash functions as defined in Theorem 5.5, with $U = [N]$ and $B = \{0,1\}$. To prove that QSL is a quantum data structure we only need to check if it can perform each of the required operations. By linearity it is enough to show the claimed complexities for some arbitrary branch $\left|\mathsf{SL}(X_S)_{h_1,\ldots,h_{\log N}}\right\rangle |h_1, \ldots, h_{\log N}\rangle |c\rangle$ of the superposition.

Initialise By construction of QSL we have that

$$|\mathsf{QSL}(X_\emptyset)\rangle = \frac{1}{\sqrt{\binom{\log N}{|\mathscr{H}|}}} \sum_{h_1,\ldots,h_{\log N} \in \mathscr{H}} \left|\overline{0}\right\rangle |h_1, \ldots, h_{\log N}\rangle \left|\overline{0}\right\rangle .$$

Creating the superposition over the hash functions takes time at most $T_{\text{init}} = O(\text{polylog}(N))$.[22]

Lookup We compute $h(i)$ in γ_N and traverse the $h(i)^{\text{th}}$ bucket comparing each i to the first $\log N$ bits of each entry. If we find the entry (i, x_i) we output 1 and 0 otherwise. The height of the bucket is at most $\log N$ and the comparison takes γ_N time, thus $T_L = O(\gamma_N \log N)$.

Insert & Delete Before we can insert (i, x_i) we first need to determine the levels it will span. For this we need to compute $h_1(i), \ldots, h_{\log N}(i)$ which we already saw takes $O(\gamma_N \log^2 N)$.

We can insert (i, x_i) into the skip list using the normal insertion procedure (see Section 5.1.3).[23] The time of the insertion depends on the number of pointers it accesses in each level, which

[22] We do not care about the exact encoding of these hash function since the cost of T_{init} is an additive cost for the algorithm we are going to use this data structure in (see Theorem 4.8).

[23] This can be seen as first trying to insert x_i and then i.

we have fixed to be $O(\log^2 N)$. Finding the next memory location based on the pointer takes $O(\gamma_N \log N)$, as this is basically a lookup. The total time of a traversal is then $O(\gamma_N \log^3 N)$. We can update the pointers of the predecessor nodes of (i, x_i) (that were already in the data structure) by computing $h(i)$, which takes γ_N, and traversing the $h(i)^{\text{th}}$ bucket. There at most $O(\log N)$ predecessor node pointers to change, so this takes $O(\gamma_N \log^2 N)$.

Now to actually put the new node in memory we need to insert it into the hash table. First we need to lookup where to insert (i, x_i) in the hash table, which takes $O(\gamma_N \log N)$. Next we need to write down the new node, taking $O(\gamma_N + \gamma_M + \gamma_N \log N)$, and potentially move up to $\log N$ elements to keep the bucket in sorted order. In total this takes $O(\log N(\gamma_M + \gamma_N \log N)) = O(\gamma_M \log^2 N)$.

Recall that deletion is the inverse of insertion, so we have that the total cost is

$$T_{ID} = O(\gamma_N \log^2 N + \gamma_N \log^3 N + \gamma_N \log^2 N + \gamma_M \log^2 N) = O(\gamma_M \log^3 N),$$

where we have used that $\gamma_N = O(\gamma_M)$.

Check We simply need to check if $c > 1$, so $T_C = O(\gamma_N)$.[24] □

Corollary 5.8. There exists a bounded-error quantum algorithm for element distinctness with time complexity at most $O(N^{2/3}\gamma_M \log^3 N)$ and space complexity $O(N^{2/3} \log^2 N \log M)$.

Sketch of Proof. Let QSL be the $(N^{2/3} + 1)$-sized quantum skip list. We have that there are two parts where QSL can fail: either the bucket of the hash table overflows or the skip list needs to access too many pointers. Ambainis showed that both the height of a bucket exceeding $\log N$ and the skip list needing to access more than $\log^2 N$ many pointers, happens only with low probability [Amb03, Lemma 6]. In particular, let $|\psi\rangle$ by the final state of the algorithm of Theorem 4.8 combined with the imperfect quantum skip list QSL that aborts if either of these two failures happen. Next let $|\psi'\rangle$ be the same algorithm, however now with the quantum skip list QSL′ that never aborts. It follows by [Amb03, Lemma 5 & 6] that $|| \, |\psi\rangle - |\psi'\rangle \, || \leq O(1/\sqrt{N})$. Thus the probability of any of the two failures occurring is negligible, meaning that the imperfectness of the quantum skip list data structure has no significant effect. The claim follows now directly from Theorem 4.8 combined with Theorem 5.7. □

5.3.2 Quantum radix tree

Although the quantum skip list of the previous section achieves a matching time complexity bound, its presentation and analysis require a lot of work. It turns out that there is a much easier data structure achieving an even better bound.[25] This data structure is a quantum version of a radix tree. As we saw in Section 5.1.2, a radix tree already achieves logarithmic cost for lookup, insertion and deletion. Moreover, the data structure representation of a radix tree is also already unique.[26] The only issue lies in its memory representation, since there is no obvious way to make it space-efficient *and* unique.

With the quantum skip list data structure, the uniqueness of memory problem was solved by using a hash table. We could use this same method also for radix trees, however, there is a more elegant and efficient trick instead. This trick was first introduced by Bernstein et al. [BJLM13], worked out formally by Jeffery [Jef14] and further improved by Buhrman et al. [BLPS22]. These last improvement were not given specifically for element distinctness yet, so we will do so here.

[24] Notice how this operation is now extremely simple, since most of the hard work has been done during the insertion and deletion operations.

[25] Unless $M = 2^{\log^4 N}$.

[26] The space optimisation ensures that the representation of the trie is unique. Note that this is exactly the reason for considering radix trees over tries, as the space optimisation itself is only minor.

Recall from Section 5.1.2 that we need at most $2s$ blocks of memory to store a radix tree. To decide the *memory layout*, i.e. in which blocks each of the nodes get stored, we can use some injective function τ giving an initial allocation of nodes to memory blocks. Once we insert and delete elements from the radix tree however, we will be moving to different allocations. By creating a uniform superposition over all possible memory allocations, i.e. all injective functions τ, we are allowed to map from one allocation to another. As long as we keep the superposition uniform, this memory representation will be unique. For this purpose we need another data structure.

Prefix-sum tree We can use a *prefix-sum tree*, also known as a *Fenwick tree* or *binary indexed tree*, to keep track of which blocks of memory are empty and which are used up by a node.[27] A prefix-sum tree is a fully-balanced binary tree where the value of the leaves are bits. Each internal node only holds a counter indicating the number of 1-valued leaves descending from it. Assume that s is a power of 2. For a subset $F \subseteq [2s]$ we label the i^{th} leaf of the tree with 1 if and only if $i \in F$. See Figure 5.5 for an example.

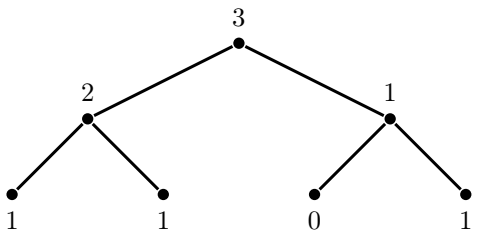


Figure 5.5: A prefix-sum tree storing the set $F = \{1, 2, 4\}$.

Define the Hilbert space

$$\mathcal{H}_{\mathsf{PST},s} := (\text{span}\{|c\rangle \mid c \in [0, s]\})^{\otimes 2s-1} \otimes (\text{span}\{|b\rangle \mid b \in \{0, 1\}\})^{\otimes 2s}$$

and the vector $|\mathsf{PST}(F)\rangle \in \mathcal{H}_{\mathsf{PST},s}$ where $\mathsf{PST}(F)$ is the encoding of the prefix-sum tree storing some set $F \subseteq [2s]$. We fix the blocks to appear in order of a breadth-first traversal of the prefix-sum tree.[28] Notice how there is no need to store the pointers of the nodes since every node has a fixed position and thus a fixed child node it points to.[29] Since we are only updating the values of the nodes, and not deleting or inserting any, the memory representation stays unique.

To use the prefix-sum tree as a memory allocator we want to implement the map

$$U_{\text{alloc}} : |\mathsf{PST}(F)\rangle \left|\overline{0}\right\rangle \left|\overline{0}\right\rangle \mapsto \frac{1}{\sqrt{|F|}} \sum_{\ell \in F} |\mathsf{PST}(F \setminus \{\ell\})\rangle |\ell\rangle \left|\overline{0}\right\rangle$$

and also its inverse U_{free}. As explained in [BLPS22, Page 7] this map can be implemented in $O(\gamma_N \log s)$ time.[30]

Quantum radix tree In Section 5.1.2 we saw how store a set of integers using a radix tree. We are going to modify this data structure slightly to allow us to store the pairs $(i, x_i) \in X_S$. To do so we extend each leaf with a list of size at most two in which we can store the indices with the same x_i in sorted order.[31] Since we also want an efficient way to check for collisions,

[27] Initially, prefix-sum trees were used to solve the dynamic prefix-sum problem (as suggested by the name).

[28] Any fixed traversal would be fine.

[29] For example, given that we are in some node at address n at height h, we can move to its left and right child node by going to the address 2^h and $2^h + 1$ respectively.

[30] The actual cost is $O(\gamma_N \log \frac{s}{\varepsilon})$ where ε is the probability of error. The error is introduced because of the need for the map $U_{\text{superimpose}} : |k\rangle \left|\overline{0}\right\rangle \mapsto \frac{1}{\sqrt{k}} \sum_{j=1}^{k} |k\rangle |j\rangle$, for $k \leq s$ some positive integer, which is impossible to implement efficiently without error from our basic gate set $\mathcal{G}$ (see Section 2.4.3). The level of this error is a similar to what one would get from translating $\mathcal{G}$ to a finite gate set (see Section 2.4.1), which is too low level to accurately keep track of here, hence the reason for omitting it.

[31] Once more this sorting makes sure that the representation is unique.

we additionally augment each internal node with a counter which counts the number of collision in its subtree. See Figure 5.6 for an example of such an augmented radix tree.

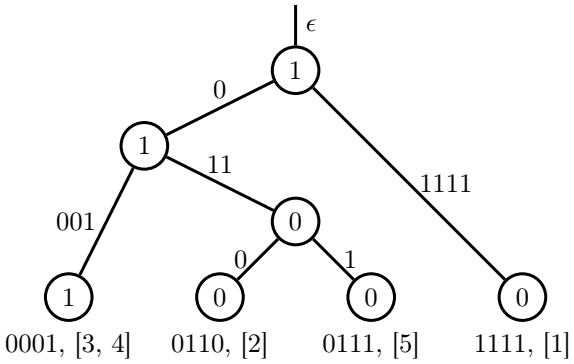


Figure 5.6: Augmented radix tree storing the set $\{(1, 1111), (2, 0110), (3, 0001), (4, 0001), (5, 0111)\}$.

Let $\mathscr{N}_{\mathsf{RT},s}$ be the set of valid nodes for the augmented radix tree storing the set X_S of size at most s, defined as

$$\mathscr{N}_{\mathsf{RT},s} := \{0,1\}^{\leq \log M} \times \{0,1\}^{\log M} \times [N] \times [0,N] \times [0,2s]^3 \times [0,s].$$

Each node is of the form $(\ell, z, i_1, i_2, p_l, p_r, p_p, c)$, where ℓ represents the label of the edge going into the node, z represent the value of the node, i_1 and i_2 the indices with the same value z, p_l, p_r and p_p the pointers to the left child, right child and parent respectively and c the number of collisions in the subtree of the node.[32] We have $z = i_1 = i_2 = 0$ for the internal nodes and $p_l = p_r = 0$ for the leaf nodes.[33] Define the Hilbert space

$$\mathcal{H}_{\mathsf{RT},s} := (\mathrm{span}\{|\nu\rangle \mid \nu \in \mathscr{N}_{\mathsf{RT},s} \cup \{\overline{0}\}\})^{\otimes 2s},$$

and a vector $|\mathsf{RT}_\tau(X_S)\rangle \in \mathcal{H}_{\mathsf{RT},s}$, where $\mathsf{RT}_\tau(X_S)$ is the encoding of the augmented radix tree storing X_S as described above, for some fixed injective function $\tau : \mathscr{N}_{\mathsf{RT}}(X_S) \mapsto [2s]$ with $\tau(\mathsf{root}) = 1$. Here $\mathscr{N}_{\mathsf{RT}}(X_S)$ denotes the exact set of nodes in the augmented radix tree storing X_S and $\mathsf{root} \in \mathscr{N}_{\mathsf{RT}}(X_S)$ is the root node that we fix to the first memory location.[34]

Definition 5.9 (Quantum radix tree)**.** Let $s \in [N]$. The s-sized quantum radix tree QRT is a family of mappings from $\{X_S \mid S \in \binom{[N]}{\leq s}\}$ to the Hilbert space

$$\mathcal{H}_s := \mathcal{H}_{\mathsf{RT},s} \otimes \mathcal{H}_{\mathsf{PST},s}$$

defined as $X_S \mapsto |\mathsf{QRT}(X_S)\rangle$ where

$$|\mathsf{QRT}(X_S)\rangle := \frac{1}{\sqrt{|\mathscr{T}|}} \sum_{\tau \in \mathscr{T}} |\mathsf{RT}_\tau(X_S)\rangle\, |\mathsf{PST}(F_\tau)\rangle\,.$$

Here $\mathscr{T}$ is the set of all injective functions $\tau : \mathscr{N}_{\mathsf{RT}}(X_S) \to [2s]$ with $\tau(\mathsf{root}) = 1$, $|\mathscr{T}| = (2s-1)!/(2s-(2|X_S|-1))!$ and $F_\tau = [2s] \setminus \tau(\mathscr{N}_{\mathsf{RT}}(X_S))$ is the complement of the image of τ.

[32] For leaves this is the number of collisions inside that leaf.

[33] Also note that $i_2 = 0$ unless there exists a collision.

[34] Any location is fine, as long as we know where to start our traversal from. Also note that we are able to fix the memory location of the root node since it cannot be deleted.

Theorem 5.10. For all $s \in [N]$ the s-sized quantum radix tree QRT is a quantum data structure (Definition 4.7). Furthermore we have $T_{\text{init}} = O(1)$, $T_L = O(\gamma_M \log M)$, $T_{ID} = O(\gamma_M \log M)$ and $T_C = O(\gamma_N)$.

Proof. Again by Definition 4.7 we only need to check if $\mathsf{RT}_Q(X_S)$ can perform each of the required operations and analyse their complexity.

Initialise By construction of QRT we have that $|\mathsf{QRT}(X_\emptyset)\rangle = |\overline{0}\rangle$, thus $T_{\text{init}} = O(1)$.

Lookup Simply lookup x_i in the radix tree using the normal lookup procedure (see Section 5.1.2). If we reach a leaf node with value x_i, check if the index i matches any of the two possible indices already stored in the corresponding leaf. If yes, output 1, otherwise 0. The complexity of the lookup is the same as in a 'normal' radix tree, namely $O(\gamma_M \log M)$.[35]

Insert & Delete We insert x_i based on the normal radix tree insertion procedure (see Section 5.1.2). This time, however, we could end up in a leaf node already labelled with $z = x_i$. In that case, simply insert i into the correct sorted position in $O(1)$ since the list is only of size 2. Next, traverse back up the tree, incrementing the counter of each node along the way.[36] It could be that the list already contained two elements, in that case we simply do not insert (i, x_i) into the radix tree.

If we do not end up in a leaf node, we follow the normal procedure with two changes. If we create a new internal node, we set the counter to the counter of its child node. If we create a new leaf node instead, we set the counter to 0.

We need to allocate memory for the new node $|n_i\rangle$ that we want to insert and the extra created node $|n\rangle$ in that insertion. We use U_{alloc} twice to obtain a superposition over the memory locations that can hold the new nodes. Next we use a QRAG (Definition 2.6) twice to swap to the contents of the nodes into memory. Thus

$$\begin{aligned}
&\frac{1}{\sqrt{|\mathscr{T}|}} \sum_{\tau \in \mathscr{T}} |\mathsf{RT}_\tau(X_S)\rangle\, |\mathsf{PST}(F_\tau)\rangle\, |\overline{0}\rangle\, |n_i\rangle\, |\overline{0}\rangle\, |n\rangle \\
\overset{U_{\text{alloc}}}{\mapsto} &\frac{1}{\sqrt{|\mathscr{T}| \cdot |F|^2}} \sum_{\tau \in \mathscr{T}} |\mathsf{RT}_\tau(X_S)\rangle \sum_{\ell,\ell' \in F} |\mathsf{PST}(F_\tau \setminus \{\ell, \ell'\})\rangle\, |\ell\rangle\, |n_i\rangle\, |\ell'\rangle\, |n\rangle \\
\overset{\mathsf{QRAG}}{\mapsto} &\frac{1}{\sqrt{|\mathscr{T}| \cdot |F|^2}} \sum_{\tau \in \mathscr{T}} |\mathsf{RT}_\tau(X_{S \cup \{i\}})\rangle \sum_{\ell,\ell' \in F} |\mathsf{PST}(F_\tau \setminus \{\ell, \ell'\})\rangle\, |\ell\rangle\, |\overline{0}\rangle\, |\ell'\rangle\, |\overline{0}\rangle\,.
\end{aligned}$$

The problem is now that the locations of $|\ell\rangle$ and $|\ell'\rangle$ are 'entangled' with the work qubits. We need to set them back to $|\overline{0}\rangle$ for this to be a valid operation. Fortunatly, a copy of these locations appears as the child pointers of the parents of the nodes that we have just created. We can use these pointers to zero out the locations.

The cost of allocating new memory is $O(\gamma_N \log s)$ and the cost of actually inserting is $O(\gamma_M \log M)$. Thus the total cost is $T_{ID} = O(\gamma_M \log M)$, since deletion is simply the inverse of insertion.

Check We simply check if the counter of the root node is greater than 1, so $T_C = O(\gamma_N)$. □

Corollary 5.11. There exists a bounded-error quantum algorithm for element distinctness with time complexity at most $O(N^{2/3}\gamma_M \log M)$ and space complexity $O(N^{2/3} \log M)$.

Proof. Let QRT be the $(N^{2/3}+1)$-sized quantum radix tree. The claim follows now directly from Theorem 5.10 combined with Theorem 4.8. □

[35] Note that we have an extra γ_M because comparing labels of size at most $\log M$ is a word operation. In the standard classical model of computation this is seen as $O(1)$, hence why it is missing in Table 5.1.

[36] Note that this is exactly the reason for having a pointer to the parent.

5.4 New Quantum Data Structures

In the previous section we saw two examples of quantum data structures adhering to the requirements posed in Section 5.2. The quantum radix tree uses a smart trick to solve the uniqueness of memory problem, but both solve the uniqueness of representation problem by choosing data structures already having a unique representation.[37] It was due to this problem that Bernstein et al. said: "*(...) problem is much more serious: it rules out balanced trees, red-black trees, most types of hash tables, etc.*" [BJLM13, page 7].

In this section we show how to construct two quantum versions of a hash table, a space-inefficient one achieving an optimal time complexity bound for element distinctness, and a space-efficient one that costs us an extra logarithmic factor.[38] Additionally we give a suggestions as to how a quantum version of a BST can be created, although we do not give a full construction.

5.4.1 Quantum hash table

Note that the hash table as introduced in Section 5.1.4 does not 'directly' violate any of the issues raised. In fact it is probably the most natural option among the covered data structures, as it solves both uniqueness problems immediately. For this, we only need to keep the elements in each bucket in a sorted array instead of in a linked list.

Unfortunately, this data structure suffers from the worst-case limitation problem. The $O(1)$ expected-case complexity of the hash table is achieved by choosing a hash function uniformly at random from a family of universal hash function (see Section 5.1.4). The quantum equivalent of this sampling is creating a uniform superposition over all possible hash functions, similar to what Ambainis did. We can circumvent this problem by making use of Corollary 2.12. We consider hash function mapping elements from $[M]$ to $[s]$ buckets.

Of course we also need an efficient way to check for collisions.[39] Since we are hashing the values of X_S, we are guaranteed that colliding values map to the same bucket. However, if multiple elements are stored in a bucket, this does not have to mean that there exists a collision. Therefore we need to augment the hash table with a counter c_b for each bucket $b \in [s]$ that counts the number of distinct collisions in that bucket. To prevent needing to check each of these counters, we create a general counter c that is the sum of all bucket counters, i.e. $c = \sum_{b\in[s]} c_b$. Each bucket has entries storing (i, x_i) first in order of increasing x_i and then in i.

Now let $\mathscr{N}_{\mathsf{HT},s} = [N] \times [M]$ be the set of valid entries in a hash table storing the set X_S of size at most s. Each node is of the form (i, x_i) and we do not need any pointers between the elements, since we allocate a fixed s-sized array for each bucket. Define the Hilbert space

$$\mathcal{H}_{\mathsf{HT},s} := (\text{span}\{|\nu\rangle \mid \nu \in \mathscr{N}_{\mathsf{HT},s} \cup \{\overline{0}\}\})^{\otimes s^2},$$

and a vector $|\mathsf{HT}_h(X_S)\rangle \in \mathcal{H}_{\mathsf{HT},s}$, where $\mathsf{HT}_h(X_S)$ is the encoding of the above described hash table storing X_S, for some fixed hash function $h : [M] \to [s]$.

We can now formally define our quantum hash table.

[37] Or in the case of the quantum skip list, a superposition over unique representations. Although this was only needed to ensure a good complexity of the data structure operations.

[38] Note that Ambainis only used a hash table to represent the unique memory of the skip list. He did not use it directly as the data structure.

[39] This is the collision as defined for element distinctness, not for hash tables.

Definition 5.12 (Quantum hash table)**.** Let $s \in [N]$ and let $\mathscr{H}$ be a family of universal hash functions (Definition 5.1) for $U = [M]$ and $B = s$. The s-sized *quantum hash table* QHT is a family of mappings from $\{X_S \mid S \in \binom{[N]}{\leq s}\}$ to the Hilbert space

$$\mathcal{H}_s := \mathcal{H}_{\mathsf{HT},s} \otimes \operatorname{span}\{|h\rangle \mid h \in \mathscr{H}\} \otimes \operatorname{span}\{|c\rangle \mid c \in [0,s]\}^{\otimes(s+1)}$$

defined as $X_S \mapsto |\mathsf{QHT}(X_S)\rangle$, where

$$|\mathsf{QHT}(X_S)\rangle := \frac{1}{\sqrt{|\mathscr{H}|}} \sum_{h \in \mathscr{H}} |\mathsf{HT}_h(X_S)\rangle\, |h\rangle\, |c_1, c_2, \ldots, c_s\rangle\, |c\rangle\,.$$

Theorem 5.13. For all $s \in [N]$ and $\mathscr{H}$ some universal family of hash functions, the s-sized quantum hash table QHT is a quantum data structure (Definition 4.7). Furthermore, we have $T_{\text{init}} = O(\text{polylog}(N))$, $T_L(i,h) = O(\gamma_M + \gamma_M|C_h(i)|)$, $T_{ID}(i,h) = O(\gamma_M + \gamma_M|C_h(i)|)$ and $T_C = O(\gamma_N)$, where $i \in [N]$, $h \in \mathscr{H}$ and $C_h(i)$ the number of collisions for an element i with hash function h.[40]

Proof. Consider the universal family of hash functions as defined in Theorem 5.3, for $U = [M]$ and $B = s$. To prove that QHT is a quantum data structure we only need to check if it can perform each of the required operations. By linearity it is enough to show the claimed complexities for some arbitrary branch $|\mathsf{HT}_h(X_S)\rangle\, |h\rangle\, |c_1, c_2, \ldots, c_s\rangle\, |c\rangle$ of the superposition.

Initialise By construction of QHT we have that

$$|\mathsf{QHT}(X_\emptyset)\rangle = \frac{1}{\sqrt{|\mathscr{H}|}} \sum_{h \in \mathscr{H}} \left|\overline{0}\right\rangle |h\rangle \left|\overline{0}\right\rangle \left|\overline{0}\right\rangle.$$

Creating the superposition over the hash functions takes time at most $T_{\text{init}} = O(\text{polylog}(N))$.[41]

Lookup Compute $h(x_i)$, in γ_M, and traverse the $h(x_i)^{\text{th}}$ bucket. If (i, x_i) is in the bucket output 1, else output 0. We have $T_L(i,h) = \gamma_M + \gamma_M|C_h(i)|$, since a comparison takes γ_M and there are $|C_h(i)|$ elements in the $h(x_i)^{\text{th}}$ bucket.

Insert & Delete Compute $h(x_i)$ and traverse the $h(x_i)^{\text{th}}$ bucket. If x_i is already present exactly once set $c_{h(x_i)} = c_{h(x_i)} + 1$ and $c = c + 1$. To insert (i, x_i) we need to change at most $|C_h(i)|$ other elements in the worst case to keep the bucket in sorted order. Both the traversal as well as keeping the elements in sorted order takes $\gamma_M|C_h(i)|$. Since deletion is the inverse of insertion we have that $T_{ID}(i,h) = \gamma_M + \gamma_M|C_h(i)|$.

Check Simply check if the global counter $c \geq 1$, so $T_C = O(\gamma_N)$. □

Corollary 5.14. There exists a bounded-error quantum algorithm for element distinctness with time complexity at most $O(N^{2/3}\gamma_M)$ and space complexity $O(N^{4/3} \log M)$.

[40] Recall the notation from Theorem 4.9

[41] Similar to the quantum skip list (see Theorem 5.7) we do not care about the exact encodings of these hash functions.

Proof. Let QHT be the $(N^{2/3}+1)$-sized quantum hash table. It follows from Theorem 5.13 that

$$\begin{aligned}
T_{ID}^{\text{avg}} &= 2\gamma_M + \gamma_M \frac{1}{N} \sum_{\substack{i\in[N]\\ h\in\mathscr{H}}} \frac{1}{|\mathscr{H}|}|C_h(i)| + \gamma_M \max_{S\in\binom{[N]}{N^{2/3}+1}} \left\{ \frac{1}{(N^{2/3}+1)} \sum_{\substack{i\in S\\ h\in\mathscr{H}}} \frac{1}{|\mathscr{H}|}|C_h(i)| \right\} \\
&= 2\gamma_M + \gamma_M \frac{1}{N} \sum_{i\in[N]} \mathbb{E}[|C_h(i)|] + \gamma_M \max_{S\in\binom{[N]}{N^{2/3}+1}} \left\{ \frac{1}{(N^{2/3}+1)} \sum_{i\in S} \mathbb{E}[|C_h(i)|] \right\} \\
&\leq 2\gamma_M + \gamma_M \frac{s-1}{s} + \gamma_M \frac{s-1}{s} \\
&= O(\gamma_M),
\end{aligned}$$

where the inequality follows from Theorem 5.2. We can also find $T_L^{\text{avg}} = O(\gamma_M)$ by a similar reasoning. Thus, by Theorem 4.9 there exists a bounded-error quantum algorithm for element distinctness with time complexity at most

$$O(N^{2/3}(\gamma_M + T_L^{\text{avg}} + T_{ID}^{\text{avg}} + T_C) + T_{\text{init}}) = O(N^{2/3}\gamma_M).$$

□

In the construction of the above quantum hash table we have been quite conservative by choosing the bucket size to be s, incurring a space complexity of $\tilde{O}(s^2)$. To be more space efficient, we can consider each bucket to contain a pointer to some sorted linked list of size at most s, making the space complexity $\tilde{O}(s)$. As we have seen for the quantum radix tree, making this memory unique requires us to use a prefix-sum tree (see Section 5.3.2).

Now let $\mathscr{N}_{\mathsf{HT}^{\mathsf{SE}},s} = [N] \times [M] \times [0,s]$ be the set of valid entries in this space-efficient version of a hash table storing the set X_S of size at most s. Each node is of the form (i, x_i, p_n), where $(i, x_i) \in X_S$ is the element you want to store and p_n a pointer to next element in the bucket. Define the Hilbert space

$$\mathcal{H}_{\mathsf{HT}^{\mathsf{SE}},s} := (\text{span}\{|p\rangle \mid p \in [0,s]\})^{\otimes s} \otimes (\text{span}\{|\nu\rangle \mid \nu \in \mathscr{N}_{\mathsf{HT}^{\mathsf{SE}},s} \cup \{\overline{0}\}\})^{\otimes s},$$

and a vector $\left|\mathsf{HT}^{\mathsf{SE}}{}_{h,\tau}(X_S)\right\rangle \in \mathcal{H}_{\mathsf{HT}^{\mathsf{SE}},s}$, where $\mathsf{HT}^{\mathsf{SE}}{}_{h,\tau}(X_S)$ is the encoding of above described space-efficient hash table storing X_S, for some fixed hash function $h : [M] \to [s]$ and injective function $\tau : \mathscr{N}_{\mathsf{HT}_h}(X_S) \mapsto [s]$. Here $\mathscr{N}_{\mathsf{HT}_h}(X_S)$ denotes the exact set of nodes in the space-efficient hash table storing X_S.

We can now formally define our space-efficient quantum hash table.

Definition 5.15 (Space-efficient quantum hash table)**.** Let $s \in [N]$ and let $\mathscr{H}$ be a family of universal hash functions (Definition 5.1) for $U = [M]$ and $B = s$. The s-sized *space-efficient quantum hash table* $\mathsf{QHT}^{\mathsf{SE}}$ is a family of mappings from $\{X_S \mid S \in \binom{[N]}{\leq s}\}$ to the Hilbert space

$$\mathcal{H}_s := \mathcal{H}_{\mathsf{HT}^{\mathsf{SE}},s} \otimes \text{span}\{|h\rangle \mid h \in \mathscr{H}\} \otimes \text{span}\{|c\rangle \mid c \in [0,s]\}^{\otimes(s+1)} \otimes \mathcal{H}_{\mathsf{PST},s}$$

defined as $X_S \mapsto \left|\mathsf{QHT}^{\mathsf{SE}}(X_S)\right\rangle$, where

$$\left|\mathsf{QHT}^{\mathsf{SE}}(X_S)\right\rangle := \frac{1}{\sqrt{|\mathscr{H}| \cdot |\mathscr{T}|}} \sum_{h\in\mathscr{H}} \sum_{\tau\in\mathscr{T}} \left|\mathsf{HT}^{\mathsf{SE}}{}_{h,\tau}(X_S)\right\rangle |h\rangle\, |c_1, c_2, \ldots, c_s\rangle\, |c\rangle\, |\mathsf{PST}(F_\tau)\rangle\,.$$

Here $\mathscr{T}$ is the set of all injective functions $\tau : \mathscr{N}_{\mathsf{HT}_h}(X_S) \to [s]$ with $\tau(\mathsf{root}) = 1$, $|\mathscr{T}| = s!/(s - |X_S|)!$ and $F_\tau = [s] \setminus \tau(\mathscr{N}_{\mathsf{HT}_h}(X_S))$ is the complement of the image of τ

Theorem 5.16. For all $s \in [N]$ and $\mathscr{H}$ some universal family of hash functions, the s-sized space-efficient quantum hash table $\mathsf{QHT}^{\mathsf{SE}}$ is a quantum data structure (Definition 4.7). Furthermore, we have $T_{\text{init}} = O(\text{polylog}(N))$, $T_L(i,h) = O(\gamma_M + \gamma_M|C_h(i)|)$, $T_{ID}(i,h) = O(\gamma_M + \gamma_M|C_h(i)| + \gamma_N \log s)$ and $T_C = O(\gamma_N)$, where $i \in [N]$, $h \in \mathscr{H}$ and $C_h(i)$ the number of collisions for an element i with hash function h.

Proof. Consider the universal family of hash functions as defined in Theorem 5.3, for $U = [M]$ and $B = s$. Note that the only difference between $\mathsf{QHT}^{\mathsf{SE}}$ and the space-inefficient quantum hash table of Definition 5.12 is in needing time to allocate memory for the insertion (and free memory for the deletion). As seen in the proof of Theorem 5.10 allocating (and also freeing) a memory block with the prefix-sum tree incurs an extra cost of $O(\gamma_N \log s)$, on top of the normal insertion cost. It then follows from Theorem 5.13 that $\mathsf{QHT}^{\mathsf{SE}}$ is a quantum data structure having the claimed complexities. □

Corollary 5.17. There exists a bounded-error quantum algorithm for element distinctness with time complexity at most $O(N^{2/3}\gamma_M \log N)$ and space complexity $O(N^{2/3} \log M)$.

Proof. Let $\mathsf{QHT}^{\mathsf{SE}}$ be the $(N^{2/3}+1)$-sized space-efficient quantum hash table. From Corollary 5.14 and Theorem 5.16 it follows that $T_L^{\text{avg}} = O(\gamma_M)$. Moreover, by a similar reasoning we find that $T_{ID}^{\text{avg}} = O(\gamma_M + \gamma_N \log(N^{2/3}+1)) = O(\gamma_M \log N)$. Thus, by Theorem 4.9 there exists a bounded-error quantum algorithm for element distinctness with time complexity at most

$$O(N^{2/3}(\gamma_M + T_L^{\text{avg}} + T_{ID}^{\text{avg}} + T_C) + T_{\text{init}}) = O(N^{2/3}\gamma_M \log N).$$

□

5.4.2 Quantum binary search tree

The quantum hash table of the previous section is practically the best quantum data structure for element distinctness that you could have, since the γ_M is inevitable due to needing to query integers in $[M]$. It is, however, instructive to show that it is possible to use BSTs as quantum data structures, since these might be desirable in other settings. For this we introduce a new solutions for dealing with the uniqueness of representation problem.[42]

This technique is an extension of the superposition over memory layout trick discussed in Section 5.3.2. Let $\mathcal{T}(X_S)$ be the set of possible BST structures for storing the set X_S. We now let the uniform superposition over these tree structures be the unique representation of our quantum data structure. The main difficulty of this is to keep the superposition uniform as we replace elements in the set. While we were not able to overcome this difficulty, this section can be seen as an interesting discussion for future work.

To allows us to store the elements $(i, x_i) \in X_S$ we are going to augment the BST in a similar way like the radix tree (Section 5.3.2). For each node we add a list of size at most two in which we can store the indices with the same x_i in sorted order. We also add a counter to each node, counting the number of collisions in its subtree, including the node itself. See Figure 5.7 for an example of such an augmented BST.

Similar to a radix tree we have that the memory representation of a BST is not unique. Hence we also use a prefix-sum tree here. Let $\mathscr{N}_{\mathsf{BST},s}$ be the set of valid nodes for the augmented BST storing the set X_S of size at most s, defined as

$$\mathscr{N}_{\mathsf{BST},s} := [M] \times [N] \times [0,N] \times [0,s]^3 \times [0,s].$$

[42] If we consider fully balanced BSTs we do have that the representation is unique. Unfortunately in constructing the setup state it is impossible to remain in a fully balanced tree since we would not have that correct amount of elements at every step.

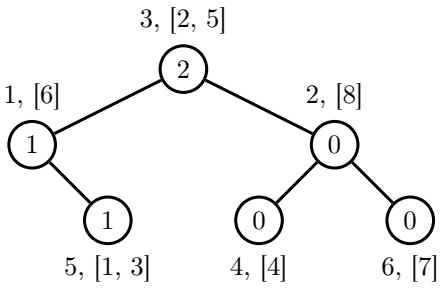


Figure 5.7: An example of a representation of an augmented BST storing the set $\{(1,5),(2,3),(3,5),(4,4),(5,3),(6,1),(7,6),(8,2)\}$.

Each node is of the form $(z, i_1, i_2, p_l, p_r, p_p, c)$, where z represent the value of the node, i_1 and i_2 the indices with the same value z, p_l, p_r and p_p the pointers to the left child, right child and parent respectively and c the number of collisions in the subtree of the node. We have $p_l = p_r = 0$ for the leaf nodes. Define the Hilbert space

$$\mathcal{H}_{\mathsf{BST},s} := (\mathrm{span}\{|\nu\rangle \mid \nu \in \mathscr{N}_{\mathsf{BST},s} \cup \{\overline{0}\}\})^{\otimes s}$$

and a vector $|\mathsf{BST}_{\tau,T}(X_S)\rangle \in \mathcal{H}_{\mathsf{BST},s}$, where $\mathsf{BST}_{\tau,T}(X_S)$ is the encoding of the augmented BST storing X_S as described above, for some fixed tree structure $T \in \mathcal{T}(X_S)$ and injective function $\tau : \mathscr{N}_{\mathsf{BST}}(X_S) \mapsto [s]$ with $\tau(\mathsf{root}) = 1$. Here $\mathscr{N}_{\mathsf{BST}}(X_S)$ denotes the exact set of nodes in the augmented BST storing X_S and $\mathsf{root} \in \mathscr{N}_{\mathsf{BST}}(X_S)$ the root node that we fix to the first memory location.

We can now formally define a quantum version of the BST.

Definition 5.18 (Quantum BST)**.** Let $s \in [N]$. The s-sized *quantum binary search tree* QBST is a family of mappings from $\{X_S \mid S \in \binom{[N]}{\leq s}\}$ to the Hilbert space

$$\mathcal{H}_s := \mathcal{H}_{\mathsf{BST},s} \otimes \mathcal{H}_{\mathsf{PST},s}$$

defined as $X_S \mapsto |\mathsf{QBST}(X_S)\rangle$ where

$$|\mathsf{QBST}(X_S)\rangle := \frac{1}{\sqrt{|\mathcal{T}(X_S)| \cdot |\mathscr{T}|}} \sum_{T \in \mathcal{T}(X_S)} \sum_{\tau \in \mathscr{T}} |\mathsf{BST}_{\tau,T}(X_S)\rangle\, |\mathsf{PST}(F_\tau)\rangle\,.$$

Here $\mathscr{T}$ is the set of all injective functions $\tau : \mathscr{N}_{\mathsf{BST}}(X_S) \to [s]$ with $\tau(\mathsf{root}) = 1$, $|\mathcal{T}(X_S)| = C_{|X_S|} = \frac{1}{|X_S|+1}\binom{2|X_S|}{|X_S|}$ is the $|X_S|^{\text{th}}$ Catalan number, $|\mathscr{T}| = (s-1)!/(s-|X_S|)!$ and $F_\tau = [s] \setminus \tau(\mathscr{N}_{\mathsf{BST}}(X_S))$ is the complement of the image of τ.

Let us discuss each of the operations needed for the quantum BST for it to be a quantum data structure (Defintion 4.7). Going from easiest, to hardest to implement.

Initialise By construction of the QBST we have that $|\mathsf{QBST}(X_S)\rangle = |\overline{0}\rangle$, thus $T_{\text{init}} = O(1)$.

Check We simply check if the counter of the root node is greater than 1, so $T_C = O(\gamma_N)$.

Lookup We can traverse the tree like in the normal lookup procedure (see Section 5.1.1). The worst-case cost of this traversal will be $O(s)$, which is undesirable. As to how to obtain the average cost of $\log s$ is unclear. We would like to use Version 2 of our main theorem (Theorem 4.9), however that does not allow us to average over the different tree structures directly.

Insert & Delete By the definition of a quantum data structure, deletion is the inverse of insertion. If we, however, look at the normal way to insert and delete from a BST, we find that they are not each other's inverses at all. We only insert elements in a leaf, while we can delete an element from anywhere. One could consider inserting a node at any part of the tree, yet it is not clear how to achieve this with good average-case complexity. In particular while needing to maintain a uniform superposition over the possible tree structures of varying sizes.

Since at this point it is unclear how to solve these issues, we suggest the concrete implementation of a quantum BST as an interesting future research direction.

5.5 Overview of the Solutions

Let us reflect on the solutions we have seen in the last two sections for the problems that arise in translating classical data structures to the quantum setting (as discussed in Section 5.2).

5.5.1 Uniqueness of memory problem

Let us start with the uniqueness of memory problem. The first, and probably the most natural, solution is to use a hash table as the memory representation, which we saw with the quantum skip list (see Section 5.3.1). The downside of this method, however, is that it takes unnecessary extra time and space, since you also need to traverse the hash table when modifying the data structure. In our quantum hash table (Section 5.4.1) this is of course not the case, since the data structure itself is already a hash table.

The faster solution is to use the superposition over memory representation trick, as seen with the quantum radix tree (Section 5.3.2) and quantum binary search tree (Section 5.4.2). By exploiting the prefix-sum tree, one can allocate memory in only $O(\gamma_N \log s)$ steps. Although not proven here, one can use this trick for any pointer based data structure. Thus, Ambainis could have alternatively used the prefix-sum tree trick to make the memory representation of his data structure unique, presumable also saving a logarithmic factor in the process.

5.5.2 Uniqueness of representation problem

We showed two ways of solving the uniqueness of representation problem. The first is to simply use a classical data structures that has such a unique representation, which we saw with the quantum radix tree (Section 5.3.2) and quantum hash table (Section 5.4.1). Also the quantum skip list (Section 5.3.1) uses this solution, albeit with some smarter tricks. The nodes in a skip list are fixed in a unique sorted order, yet the levels that each node spans depends on random coin flips. Clearly the representation is not unique then, however by choosing the levels beforehand this problem is easily solved.[43]

The second solution uses a newly introduced technique of taking a superposition over all data structure representations, as used for the quantum binary search tree (Section 5.4.2). This technique can be seen as a direct extension of the technique used to solve the memory representation problem. In fact, with this technique we can make any classical data structure unique in its representation. It shows the beauty of solving a problem inherent to quantum with the strengths of quantum. Nevertheless, maintaining this superposition is much more complex than for the memory representation. So despite providing a method to 'solve' the uniqueness of representation problem completely, it unclear if these techniques can actually be used concretely. This is an interesting direction for future work.

[43]The uniqueness of representation problem of the data structure is easily solved, as we have seen, ensuring that the skip list does not take up too much space and run for too much time is far from obvious.

5.5.3 Worst-case limitation problem

Also for the worst-case limitation problem we have seen two solutions. The first cannot really be considered a 'solution', but it is to compare a data structure that aborts after a certain number of steps with a perfect data structure. As seen in the proof sketch of Corollary 5.8, as long as the underlying probability distributions of the final states of the algorithm are close to each other, we can get a better than worst-case analysis.

The second solution, however, is much more elegant and uses a very recent result. By using Version 2 of our main theorem we are able to average over some initial randomness, which in the case of our quantum hash table were the hash functions. What that theorem additionally shows is that we are able to average over the different inputs as-well. Although not needed for the quantum data structures presented here, we believe this should be helpful in realising the quantum version of the BST in the future.

6 | Conclusion

After the two preliminary chapters, we proved our main theorem in Chapter 4 in two parts. In Version 1 we showed a data structure dependent time complexity bound for element distinctness and generalised this in Version 2 by giving an average-case complexity bound. Moreover, these theorems eliminate the need for an explicit diffusion operation. We believe the combination of existing and novel techniques will be beneficial for quantum algorithms other than element distinctness (see Section 4.3 for an overview of the techniques).

Using Version 1 of the main theorem we reanalysed both the quantum skip list (Section 5.3.1) and the quantum radix tree (Section 5.3.2) in the QWRAM model. With the former we obtain a time complexity of $O(N^{2/3}\gamma_M \log^3 N)$, which is slightly lower than the original $O(N^{2/3}\log^4(N+M))$ time bound [Amb03], due to the counting word operations as γ. For the quantum radix tree we show a potentially higher bound of $O(N^{2/3}\gamma_M \log M)$ compared to the original $O(N^{2/3}(\log N + \log M))$ [Jef14], since there the comparison of the labels was counted as $O(1)$.

Next we showed, using Version 2 of the main theorem, how to construct a quantum version of a hash table (Section 5.4.1). Not only is that construction significantly simpler than the data structure used before, it also achieved an optimal time complexity bound of $O(N^{2/3}\gamma_M)$. We say optimal, as the γ_M factor is inevitable, since we are required to write down the queries for integers of size at most M. Of course there are some other word operations hidden in that factor, but we do not view these as more costly. The downside is that the hash table hash large space complexity. Hence we also gave a space-efficient alternative achieving a $O(N^{2/3}\gamma_M \log N)$ time complexity, yet at cost of an extra logarithmic factors. Nevertheless, even this version is better than both the quantum skip list and quantum radix tree. In particular if M is very large, for example $M = 2^{\text{polylog}(N)}$, our quantum hash table is much more efficient than the quantum radix tree. See Table 6.1 for a detailed breakdown of the exact complexities for each of the quantum data structures.

Finally, we gave a construction for a potential version of a quantum BST (Section 5.4.2) by using a superposition over all the possible tree structures. For now, we were not able to overcome the difficulties posed in that section and leave the concrete construction as a suggestion for a future work direction.

Let us now reflect back on our initial research question:

> *"What properties are required of quantum data structures, and how can we translate classical data structures for use in the quantum setting?"*

With our formal definition of a quantum data structure (Definition 4.7) we give concrete requirements of a quantum data structure. The uniqueness constraint (Section 5.2.1) is already baked into the definition, since the mapping is well-defined, but it still needs to be tackled. The worst-case limitation problem (Section 5.2.2), on the other hand, is not a real constraint as shown by Version 2 of the main theorem. Besides being sufficient, we also believe these requirements to be necessary. Thus the concrete requirements of a quantum data structure (for element distinctness) are as follows: a (unique) mapping from a set to some Hilbert space on which we are able to perform a lookup, insert, delete and check operation.

As for the second part of our research questions we show numerous existing techniques and also new ones for making this translation possible (see Section 5.5 for an overview). The elimination of an explicit diffusion operator makes the remaining operations only of a classical nature. Although the checking for collisions operator is more specific to the problem of element distinctness, its implementation is (almost) always only checking if a counter is of a particular size. Consequently, translating classical data structure for use in the quantum setting requires much less modification.

Based on our findings, we conclude that the limitations of using classical data structures in the quantum setting are much less restrictive than previously thought.

Data Structure	**Lookup**	**Insertion & Deletion**	**Check**	**Space complexity**
QSL	$O(\gamma_N \log N)$	$O(\gamma_M \log^3 N)$	$O(\gamma_N)$	$O(s \log^2 N \log M)$
QRT	$O(\gamma_M \log M)$	$O(\gamma_M \log M)$	$O(\gamma_N)$	$O(s \log M)$
QHT	$\boldsymbol{O(\gamma_M)^*}$	$\boldsymbol{O(\gamma_M)^*}$	$\boldsymbol{O(\gamma_N)}$	$\boldsymbol{O(s^2 \log M)}$
QHT^{SE}	$\boldsymbol{O(\gamma_M)^*}$	$\boldsymbol{O(\gamma_M \log N)^*}$	$\boldsymbol{O(\gamma_N)}$	$\boldsymbol{O(s \log M)}$

Table 6.1: An overview of the time and space complexity of the operations required from quantum data structures for element distinctness storing a set X_S of size at most s. The yellow and green colours indicate the acceptable and very good complexities respectively. In bold are the new data structures introduced in this thesis. Cells labelled with * indicate average-case complexities.

6.1 Future Work

We conclude the thesis by giving suggestions for future work directions.

As mentioned in Section 5.4.2, constructing a concrete quantum version of a BST poses several challenges. As such a data structure might be desirable in other settings, it is interesting to overcome these challenges. Using a similar trick as for the quantum BST, one can try to construct quantum versions of self-balancing BSTs like a red-black tree or AVL tree. These trees remain in a (close to perfectly) balanced structure, so it is perhaps easier to achieve the logarithmic cost desired from a traversal. However, rebalancing these trees at certain time step is a complicated operation and it is unclear how to 'spread-out' this cost over the different steps.

Another direction is to see if the main theorem can be proven directly using the Hamming graph (Definition 4.5), instead of the permuted Johnson graph. Technically the Hamming graph is even more natural for getting rid of the diffusion operator, since the label set is already $[s] \times [N]$. Thus we do not even need the dead-end edges trick. Not only that, but the setup step also becomes significantly simpler. Simply create a superposition over all indices $i_1, i_2, \ldots, i_r \in [N]$ in $O(s\gamma_N)$ steps and the desired setup state is created. The problem is, however, that then we would need to store tuples in our data structure instead of sets. Furthermore, these tuples can have repeated elements, which makes the analysis far more complex. These limitation do not mean that it is impossible to use the Hamming graph, so we leave this as future work.

Finally, the techniques used to prove our main theorem could be applied to computational problems other than element distinctness. In particular for problems already using a quantum walk over a Johnson graph. For example, we believe the translation of the main theorem, and the quantum data structures, to work for the k-element distinctness problem to be quite straightforward. Determining if other Johnson-graph type problems, like the subset-sum problem [BJLM13] and the closest pair problem [ACL$^+$19], can be improved requires more work however.

Bibliography

[ACL+19] Scott Aaronson, Nai-Hui Chia, Han-Hsuan Lin, Chunhao Wang, and Ruizhe Zhang. On the quantum complexity of closest pair and related problems. 2019. doi:10.48550/ARXIV.1911.01973.

[AGJK19] Andris Ambainis, András Gilyén, Stacey Jeffery, and Martins Kokainis. Quadratic speedup for finding marked vertices by quantum walks, 2019. doi:10.48550/ARXIV.1903.07493.

[AKS83] Miklós Ajtai, Janos Komlos, and Endre Szemerédi. An o(n log n) sorting network. volume 3, pages 1–9, 01 1983. doi:10.1145/800061.808726.

[Amb03] Andris Ambainis. Quantum walk algorithm for element distinctness. 2003. doi:10.48550/ARXIV.QUANT-PH/0311001.

[BDH+05] Harry Buhrman, Christoph Dürr, Mark Heiligman, Peter Høyer, Frédéric Magniez, Miklos Santha, and Ronald de Wolf. Quantum algorithms for element distinctness. *SIAM Journal on Computing*, 34(6):1324–1330, January 2005. doi:10.1137/s0097539702402780.

[Bel13] Aleksandrs Belovs. Quantum walks and electric networks, 2013. doi:10.48550/ARXIV.1302.3143.

[BH12] Andries E. Brouwer and Willem H. Haemers. *Spectra of Graphs*. Springer New York, 2012. doi:10.1007/978-1-4614-1939-6.

[BHMT02] Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation, 2002. doi:10.1090/conm/305/05215.

[BJLM13] Daniel J. Bernstein, Stacey Jeffery, Tanja Lange, and Alexander Meurer. Quantum algorithms for the subset-sum problem. Cryptology ePrint Archive, Paper 2013/199, 2013.

[BJY23] Aleksandrs Belovs, Stacey Jeffery, and Duyal Yolcu. Taming quantum time complexity, 2023. doi:10.48550/ARXIV.2311.15873.

[BLPS22] Harry Buhrman, Bruno Loff, Subhasree Patro, and Florian Speelman. Memory compression with quantum random-access gates. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.TQC.2022.10.

[BV97] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997. doi:10.1137/S0097539796300921.

[Chi13] Andrew Childs. Quantum algorithms: Lecture notes, 2013.

[CL23] Narknyul Choi and nbsp;Min-Ho Lee. Hitting time in random walks and effective resistance in electric networks. *New Physics: Sae Mulli*, 73(3):291–295, March 2023. doi:10.3938/npsm.73.291.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[CW79] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. doi:10.1016/0022-0000(79)90044-8.

[dW23] Ronald de Wolf. Quantum computing: Lecture notes, 2023. doi:10.48550/ARXIV.1907.09415.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996. doi:10.48550/ARXIV.QUANT-PH/9605043.

[Hor19] William George Horner. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, 109:308–335, 1819. doi:10.1098/rstl.1819.0023.

[Jef14] Stacey Jeffery. *Frameworks for Quantum Algorithms*. Phd thesis, University of Waterloo, 2014.

[Jef23] Stacey Jeffery. Quantum subroutine composition, 2023. doi:10.48550/ARXIV.2209.14146.

[Jef24] Stacey Jeffery. Advanced quantum algorithms: Lecture notes, 2024.

[JZ23] Stacey Jeffery and Sebastian Zur. Multidimensional quantum walks, with application to k-distinctness, 2023. doi:10.48550/ARXIV.2208.13492.

[KLM07] Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing*. Oxford University Press, 2007.

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, 1968.

[Kup11] Greg Kuperberg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem, 2011. doi:10.48550/ARXIV.1112.3333.

[KvdW24] Aleks Kissinger and John van de Wetering. Picturing quantum software: An introduction to the zx-calculus and quantum compilation (lecture notes), 2024.

[LPW08] D.A. Levin, Y. Peres, and E.L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Soc., 2008.

[MNRS11] Frédéric Magniez, Ashwin Nayak, Jérémie Roland, and Miklos Santha. Search via quantum walk. *SIAM Journal on Computing*, 40(1):142–164, January 2011. doi:10.1137/090745854.

[Mor68] Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, oct 1968. doi:10.1145/321479.321481.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[NPS20] María Naya-Plasencia and André Schrottenloher. Optimal merging in quantum k-xor and k-sum algorithms. In *Advances in Cryptology – EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II*, page 311–340, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-45724-2_11.

[Pat23] Subhasree Patro. *Quantum fine-grained complexity*. Phd thesis, Institute for Logic, Language and Computation, 2023.

[PG14] András Pál Gilyén. Quantum walk based search methods and algorithmic applications. Master's thesis, Eötvös Loránd University, 2014.

[Pre98] John Preskill. Lecture notes for physics quantum information and computation, 1998.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, jun 1990. doi:10.1145/78973.78977.

[Shi02] Yaoyun Shi. Quantum lower bounds for the collision and the element distinctness problems. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, SFCS-02. IEEE Comput. Soc, 2002. URL: http://dx.doi.org/10.1109/SFCS.2002.1181975, doi:10.1109/sfcs.2002.1181975.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997. doi:10.1137/s0097539795293172.

[Sze04] M. Szegedy. Quantum speed-up of markov chain based algorithms. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 32–41, 2004. doi:10.1109/FOCS.2004.53.

[WC81] Mark N. Wegman and J.Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981. doi:10.1016/0022-0000(81)90033-7.

[Win60] P. F. Windley. Trees, Forests and Rearranging. *The Computer Journal*, 3(2):84–88, 01 1960. doi:10.1093/comjnl/3.2.84.

[WZ82] William K Wootters and Wojciech H Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, 1982. doi:10.1038/299802a0.