University of Bergen
Department of informatics

An Algorithm for k-insertion into a Binary Heap

Author: Marie Natland Berg

Supervisors: Martin Vatshelle and Pål Grønås Drange

May 31, 2023

Abstract

In this thesis, we present an algorithm for k-insertion into a binary heap inspired by the article [15] running in worst-case time $\mathcal{O}(k + \log(k) \cdot \log(\frac{n+k}{k}))$, improving the standard k-insertion algorithm for binary heaps in terms of worst-case running time. The algorithm combines two routines called Heapify and Walk down. We assess the performance of the algorithm in theory and practice and compare it to some well-known k-insertion methods. We do this by implementing the algorithms and measuring and comparing their running times on different datasets. Through practical tests, we conclude that the algorithm performs better than the standard k-insertion algorithm on worst-case input while being slightly slower on randomized input. We, therefore, conclude that it can be better in real-world applications.

Acknowledgements

First and foremost, I would like to thank my supervisors, Martin Vatshelle and Pål Grønås Drange, for their guidance and many valuable discussions during the master's program. I would also like to especially thank my friends and family for their support throughout this past year.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In 1964 J. W. William introduced "Algorithm 232" [17], a data structure that later became known as the binary heap. The data structure was initially inspired by a paper published by Floyd in 1962 on sorting using a tournament tree [6]. The data structure has since become one of the most fundamental data structures in computer science.

Over the last 60 years, a tremendous amount of research has been done within the design and analysis of the data structure, originating back to the initial work of Williams and Floyd. Porter and Simon [14] proved that insertion into a binary heap admits a constant average-case running time. Sack and Strothotte [15] have published a paper showing how to merge two binary heaps of size n and k in time $\mathcal{O}(k \cdot \log(n) \cdot \log(k))$ where $k \leq n$. Fredman and Tarjan [8] invented a new form of heap called the Fibonacci heap where insertion can be done in $\mathcal{O}(1)$ time and removal in $\mathcal{O}(\log(n))$ time. Fredrickson [7] proved that selecting the k smallest elements in a binary heap can be done in time $\mathcal{O}(k)$. In 2010, Kamp published the paper "You're doing it wrong" [11] claiming to improve the traditional implementation of the binary heap with the B-heap, a binary heap implemented to keep subtrees in a single page on the computer. This is only a fraction of the extensive research done on the data structure. Although many results have been made to improve on the theoretical runtime of the binary heap, in practice the very simple implementation of a binary heap is difficult to beat [2]. The main reason for this is the constant average-case runtime of the insertion method when considering random input [14].

In this thesis, we study k-insertion into a binary heap; given a binary heap of size n, we want to simultaneously insert a collection of k elements into the heap. SEQUENTIAL INSERTION, with running time $\mathcal{O}(k \cdot \log(n))$, is the most common k-insertion algorithm for the binary heap. It is widely adopted in various programming languages and libraries. For instance, in Java, the `PriorityQueue` class, which utilizes a binary heap, employs SEQUENTIAL INSERTION as the default insertion method when adding a collection of elements to a priority queue. BULK INSERTION WITH HEAPIFY, with running time $\mathcal{O}(n + k)$, is also a popular choice of the k-insertion method. Since the algorithm's worst-case running time is linear, even when k is approaching n in size, the algorithm is especially favorable for large datasets.

Sack and Strothotte [15] have described an algorithm for merging priority queues organized as binary heaps. A byproduct of the paper is the existence of an efficient k-insertion algorithm, improving both SEQUENTIAL INSERTION and BULK INSERTION WITH HEAPIFY in terms of worst-case running time. The paper also proposes such an algorithm. However, due to the inherent intricacies of the algorithm, we develop a new k-insertion algorithm inspired by the one presented in [15], running in time $\mathcal{O}(k + \log(k) \cdot \log(\frac{n+k}{k}))$.

Furthermore, we investigate and compare the behavior of the algorithm's running time in theory and practice to the running times of SEQUENTIAL INSERTION and BULK INSERTION WITH HEAPIFY by implementing the algorithms and performing benchmark testing on different datasets.

Chapter 2

Preliminaries

In this chapter we give a gentle introduction to most of the terminology used later in this thesis. The primary purpose of this chapter is to collect terms and results in one place for easy reference later. The chapter is divided into three main parts, graph theory, data structures, and binary heaps. In Section 2.1 we will briefly introduce some of the most fundamental results and definitions in graph theory. This will be relevant when discussing binary heaps in Section 2.3. In Section 2.2 we will give a brief introduction to data structures and also quickly go through some data structures that will be relevant later. In this section we will not deal with the binary heap data structure, but this will instead be gone through in greater detail in Section 2.3. Finally, in Section 2.3 we will, by using the terminology and results provided in Section 2.1, describe how one can implement a binary heap data structure and also analyze the time complexity of the implementation.

2.1 Graph theory

In this section we will give a brief introduction to some fundamental definitions and results in graph theory. We will first define general graphs and terms related to this and then move over to proving basic properties of trees and rooted trees. Lastly, in the section we deal with binary trees and we will present the results needed to discuss the binary heap data structure in Section 2.3.

2.1.1 Graphs

A *graph* $G = (V, E)$ consists of a set V of *nodes* and a collection E of pairs of nodes, called *edges*. $V(G)$ and $E(G)$ denote the set of nodes and edges of a graph G, respectively. A node is sometimes also called a vertex. An edge $\{u, v\}$ that has u and v as *endpoints* is denoted uv. If $e = uv \in E(G)$ for some graph G, u and v are said to be *neighbors*. Moreover, e is *incident* with u and v. An edge uu that joins a node to itself is called a *loop*. If two or more edges join the same two vertices, the edges are called *multiple edges*. A graph is said to be *simple* if it has no loops or multiple edges. Unless otherwise stated, we will in this thesis only consider simple graphs. The *neighborhood* of a vertex u in a graph G is the set of vertices it is adjacent to and is denoted $N_G(u)$. More formally $N_G(u) = \{v \in V(G) : \{u, v\} \in E(G)\}$. The degree of a vertex u in a simple graph G is the size of its neighborhood and is denoted by $d_G(u)$. A graph H is said to be a *subgraph* of a graph G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Moreover, H is said to be a *spanning subgraph* of G if $V(H) = V(G)$ [3].

A *walk* in a graph G is a sequence of nodes $v_0, v_1, v_2, ..., v_n$ such that $v_i v_{i+1} \in E(G)$. The first node v_0 of a walk is called the *initial node*, and the last node v_n is called the *final node*. The number of edges in a walk is called the *length* of the walk. A walk where all the edges are distinct is called a *trail*. A trail in which all nodes are distinct, except possibly $v_n = v_0$, is called a *path*. A path with $v_n = v_0$ is called a *cycle*. A graph without cycles is called an *acyclic graph*. A graph is *connected* if, for any two vertices u and v in the graph, there is a path from u to v. If a graph is not connected, it comprises several connected components. A *connected component* in a graph G is a maximal connected subgraph of G [1].

2.1.2 Trees

A *forest* is a simple acyclic graph. A connected forest is called a *tree.* A node u of a tree T where $d_T(u) = 1$ is called a leaf. The nodes of a tree that are not leaves are called *internal* nodes. [3]

Theorem 2.1.1 ([1]). *Let $n \in \mathbb{Z}$ and let T be a graph with n vertices. Then the following statements are equivalent:*

(i) T is a tree;

(ii) T has $n - 1$ edges and no cycles;

(iii) T has $n - 1$ edges and is connected.

A *rooted tree* is a connected tree where one node, called the *root*, is distinguished from the leaf nodes and internal nodes. If the rooted tree only consists of one node, the root is also a leaf node. Let T be a rooted tree and let u and v be two nodes of T. If r is the root of T, T is said to be rooted at r. If v is the last node visited on the path from the root to a u, v is called the *parent* of u. Conversely, u is said to be a child of v. Consequently, the leaves do not have any children and the root is the only node that does not have a parent. Given a path in T from the root to u, the predecessor vertices on that path are known as the *ancestors* of u. If u is an ancestor of v, v is called an *descendant* of u. The *least common ancestor (lca)* of u and v is the first common node on the respective paths from u and v to the root and is denoted $lca(u, v)$. The subtree of T rooted u is denoted T_u and consists of only descendants of u. The *height of a node* in a rooted tree is the length of the longest path from the node to a leaf node, where one only traverses the descendants of the initial node. The *height of a rooted tree* is equal to the height of its root. The *level of a node* in a rooted is the length of the path from the node to the root of the tree [9].

If every internal vertex of a rooted tree has at most m children, the tree is called an *m-ary tree.* A *binary tree* is a rooted tree where each internal node has at most two children. The children of a node in a binary tree are often referred to as the *left and right children of the node.* A binary tree is said to be *full* if every node has either two or no children. If each level in a binary tree is filled, except for possibly the last, and all nodes are as far left as possible, the tree is called a *complete binary tree.* A *perfect binary tree* is a binary tree where each internal node has exactly two children, and all leaf nodes are on the same level [9].

Theorem 2.1.2 ([10]). *The maximum number of nodes in a complete binary tree of height h is $2^{h+1} - 1$.*

Theorem 2.1.3 ([16]). *The height of a complete binary tree of size n is $\lfloor log_2(n) \rfloor$.*

2.2 Data structures

In this section we will give a short introduction to some of the most fundamental theories regarding data structures. In Section 2.3.1 we define terms related to general data structures, while in Sections 2.2.2 - 2.2.4 we give examples of three relevant data structures: namely, the array, the linear list, and the priority queue data structure. The data structures will be discussed briefly on a theoretical level, and their implementation will not be discussed in great detail. Even though a binary heap is a data structure, it will be skipped in this section and will be addressed in Section 2.3.

2.2.1 Definitions

A *data structure* is a way of storing and organizing data so that it can be used efficiently. More formally, a data structure is a collection of data values along with a group of operations permitted on them and a set of axioms describing the semantics of the operations. This definition is often referred to as an *abstract data type* because the axioms do not imply a specific form of representation. The definition only states what the data structure should do and not necessarily how it does it. An *implementation of a data structure* is a specification of how each data element is represented and requires that every function be implemented according to the specified axioms [10].

2.2.2 Arrays

Intuitively, an *array* is a set of pairs of indices and values. For each defined index, there is an associated value. The main properties of an array are storing and retrieving values, and some of the essential functions of an array are:

- Retrieve: either return the value associated with an input index or an error if no such element exists.
- Add: enters a given key-value pair into the array.

Arrays are most often implemented using contiguous memory, but not necessarily [10].

2.2.3 Linear lists

One of the most commonly found data structures is the linear list. A linear list is an abstract data structure that is either empty or can be written as

$$l = (a_0, a_1, ..., a_{n-1})$$

Where the a_i's are data elements of some data type, and n is referred to as the *length* of the list. There are a variety of operations that can be performed on this type of list, some of them being:

- Size: returns the length of the list.
- Retrieve: returns an element at the specified position.
- Add: stores a new value at a given position.

- INSERT: inserts a new value at a specified position. Given i as the input index, this operation causes the elements at index $i, i+1, ..., n-1$ to be renumbered as $i+1, i+2, ..., n$.

- DELETE: deletes an element at a given position. Given i as the input index, this operation causes the elements numbered $i+1, i+2, ..., n-1$ to be renumbered as $i, i+1, ..., n-2$.

- READ: read the list from left to right (or right to left).

Note that in the operations that take a position i as input, we assume that $0 \leq i \leq n-1$, where n is the size of the list. If i is outside this bound, we expect the functions to return an error. Also, note that the DELETE and INSERT operations update the list size accordingly [10].

A common way of representing a linear list is by using an array. Each array element a_i is associated with the array index i. In this thesis, we consider zero-indexed linear lists. For a list of size n, the first element is located at index 0, while the last element is positioned at index $n-1$.

2.2.4 Priority Queues

A *queue* is a type of data structure where one element is called the front, and deletion always occurs here. Given a queue

$$q = (a_0, a_1, ..., a_{n-1})$$

We say that a_0 is at the *front* of the queue. The two primary operations performed on queues are deletion and insertion.

- ADD: adds an element at the rear end.

- REMOVE MIN: removes and returns the front element. This function will throw an error if the queue is empty.

How a new element is inserted into the queue depends on the implementation. For example, a *linked list* is a particular type of queue where insertion occurs at the queue's end. Thus, in a linked list, the element first inserted is the one first removed.

A *priority queue* is a type of queue where each data element has an associated priority. The data elements can, therefore, be viewed as tuples containing the element and its priority. The priority queue is also given a set of rules describing how to compare the priority of the data elements. The element with the highest priority is placed at the front of the queue, while the element with the lowest priority is placed at the end. A data element is inserted into the priority queue by comparing the element's priority with the priority of the elements already in the priority queue. Therefore, elements with higher priority are removed before elements with lower priority. Priority queues are often implemented using a data structure called a binary heap, but they can also be implemented by a sorted linear list [16]. In this thesis, we will consider priority queues implemented using a binary heap data structure.

2.3 Binary heaps

In this section we provide a thorough introduction to the binary heap data structure. First, in Section 2.3.1 we present some basic definitions and properties of the binary heap data structure. Then, in Section 2.3.2 and Section 2.3.3 we describe how a binary heap can be represented using an array. Lastly, in Section 2.3.4 we define and implements some fundamental operations performed on heaps. We also prove the correctness of the implementation.

The binary heap data structure is built upon the tree data structure. In computer science, a tree is an abstract data structure where the elements are connected hierarchically. In Section 2.1, we dealt with trees from a graph theoretical standpoint. This theory can be applied to the tree data structure. Therefore, we will not go into more depth on the tree data structure but instead use the definitions and properties already defined in Section 2.1

2.3.1 Definitions and properties

A *heap* is a tree-based data structure where the tree takes the form of a complete tree, satisfying the *heap property* that for every node except the root, the value stored at a node is larger than the value stored at its parent. Immediately from this, we get that the smallest element in the heap is held at the root, and the largest is stored at one of the leaves. However, there is no ordering between siblings or cousins in a heap. The heap relationship is only between nodes and their parents, grandparents, etc. [16].

Some of the most common operations performed on heaps are:

- HEAPIFY: transforms a linear list into a heap.
- ADD: inserts a new item into a heap.
- FIND MIN: returns, but does not remove, the item of minimum value.
- REMOVE MIN: returns and removes the item with the minimum key.

A *binary heap* is a heap data structure where the tree is a complete binary tree [16]. From Theorem 2.1.3, we then get the property that the height of a binary heap is $\lfloor \log_2(n) \rfloor$.

2.3.2 Array based binary heap implementation

There are several ways of implementing a binary heap, but throughout this thesis, we will mainly focus on binary heaps implemented using an array. This section will first examine how one can represent a complete binary tree using an array. Then, in

Section 2.3.3, we will show how one can ensure that the heap property is satisfied through a process called Heapify.

Let T be a complete binary tree with n elements. The array representation of T can be achieved by traversing T level by level, starting at the root and storing the nodes consecutively in an array, starting at index 0. The parent-child relationship between the nodes can be found using arithmetic on the array indices.

Theorem 2.3.1. *Let T be a perfect binary tree where each internal node has exactly two children. Let A be the array representation of T constructed as described above, and suppose that i is a valid index of A storing a node u of T. Then*

a) *if u is an internal node, the right child of u is located at index $2i + 2$.*

b) *if u is an internal node, the left child of u is located at index $2i + 1$.*

c) *if u is not equal to the root of T, the parent of u is located at index $\left\lfloor \frac{i-1}{2} \right\rfloor$.*

Proof.

a) Let l be the level of u in T, and let v be the last node on level l. Since T is a perfect binary tree we have by Theorem 2.1.2, that the number of nodes contained in the layers up to and including l is $2^{l+1} - 1$. The index of v is $2^{l+1} - 2$. Let j be the number of nodes between u and v in T. We then get that

$$i = 2^{l+1} - 2 - j.$$

Note that there are exactly $2j$ nodes between the right child of u and the right child of v. Furthermore, the right child of v is, by Theorem 2.1.2, located at index $2^{l+2} - 2$ in A. We then get that the index of the right child of u is

$$2^{l+2} - 2 - 2j = 2(2^{l+1} - 2 - j) + 2 = 2i + 2$$

b) This follows immediately from **a)** and the observation that the left child of any node is always one place before its right child.

c) let v be the parent of u and let j be the index of v in A. By **a)** and **b)** we have that

$$j = \begin{cases} \frac{i-1}{2}, & \text{if u is a left child} \\ \frac{i-2}{2}, & \text{if u is a right child.} \end{cases} \tag{2.1}$$

Now let $j = \left\lfloor \frac{i-1}{2} \right\rfloor$. If u is a left child, the expression directly satisfies 2.1. Moreover, if u is the right child of v we have that

$$j = \left\lfloor \frac{i-1}{2} \right\rfloor = \left\lfloor \frac{i-2}{2} + \frac{1}{2} \right\rfloor = \frac{i-2}{2}$$

□

Note that Theorem 2.3.1 assumes that T is a perfect binary tree. It is not necessarily true for a complete binary tree that each internal node has exactly two children, and all leaf nodes are on the same level. If a tree consists of an even number of nodes, one can easily see that the last node in the tree doesn't have a sibling. However, one can still use the results from Theorem 2.3.1 on binary heaps by checking that the resulting index is valid, i.e., between 0 and $n - 1$, where n is the size of the binary heap.

2.3.3 Heapify

HEAPIFY is the process of creating a binary heap data structure from a complete binary tree, introduced by Floyd in 1964 [5]. HEAPIFY works bottom-up, ensuring that each node in the input tree becomes the root of a binary heap. This is commonly achieved using a helper function referred to as SINK or SIFT DOWN in literature. For example, given a node u in a binary heap H, SINK attempts to restore the heap property of H_u. It assumes that both the left and right subtrees of u satisfy the heap property, with the only possible violation being u being greater than its immediate children. To fix this violation, u is successively exchanged with its smallest child until the heap property is satisfied in H_u.

Algorithm 1: SINK

```
Data: Array based binary heap $H$ and index $i \in \{0, 1, ..., |H| - 1\}$ where the
      left and right subtree of the node at index $i$ satisfy the heap property.
Result: Restores the heap property of $H_i$
while $i$ is a non-leaf do
    $j \leftarrow$ index of smallest child of $i$;
    if $H[j] < H[i]$ then
        swap H[i] and H[j];
        $i \leftarrow j$;
    else
        break
    end
end
```

In the worst case, a node has to sink down to become a leaf. Hence, from Theorem 2.1.3, Algorithm 1 achieves a worst-case running time of $\mathcal{O}(\log(n))$, where n is the size of the input heap. Generally, when calling Algorithm 1 on a node at height h, we achieve a running time of $\mathcal{O}(h)$ [16].

Algorithm 2: HEAPIFY

```
Data: Array $H$
Result: Converts $H$ into a binary heap
$end \leftarrow$ index of parent of last element of $H$;
for $i = end$ downto 0 do
    SINK$(H, i)$;
end
```

Theorem 2.3.2 ([16])**.** *Algorithm 2 correctly converts an array of size n into a binary heap in time $\mathcal{O}(n)$.*

2.3.4 Heap operations

In this section we discuss some of the most basic operations performed on binary heaps. We will mainly focus on removal and insertion into a binary heap.

2.3.4.1 Removal

As mentioned, the smallest element in a binary heap is located at the root, i.e., at index 0. The Find min operation, defined in Section 2.3.1, can therefore be implemented in constant time by simply returning the element at index 0. The procedure of restoring the heap property after the root is removed consists of two parts. First, the last element in the binary heap is placed at the empty root location, and then the heap property is restored. Since the root is the only node in the binary heap violating the heap property, it can be fixed by making a call to the function Sink, defined in Section 2.3.3 [12]. The algorithm Remove min can therefore be implemented as follows:

Algorithm 3: Remove min

```
Data: Array based binary heap $H$
Result: Returns and removes the smallest element of $H$
$smallest \leftarrow$ root of $H$;
$h[0] \leftarrow$ last element of $H$;
Sink(H, 0);
return smallest
```

The number of operations required to remove the smallest element from a binary heap while maintaining the heap property depends on the number of levels the new root needs to be swapped. Thus, from Theorem 2.1.3, we conclude that the worst-case running time of Algorithm 3 is $\mathcal{O}(\log(n))$, where n is the size of H.

Theorem 2.3.3 ([16])**.** *Algorithm 3 correctly removes the smallest element of the input heap in time $\mathcal{O}(\log(n))$ while maintaing the heap property.*

2.3.4.2 Insertion

As with deletion, inserting an element into a binary heap consists of two parts. First, the new element is added at the first available leaf position. This is to preserve the complete binary tree property. Then, the heap property is restored. Restoring the heap property is done bottom-up by successively comparing and swapping the new node with its parent. This operation is often referred to as Swim or Sift up in the literature and can be implemented in the following way:

Algorithm 4: Swim

```
Data: Array based binary heap $H$ and index $i \in \{0, 1, ..., |H| - 1\}$
Result: Moves the node at index $i$ up the tree until the heap property is
        satisfied.
while $i > 0$ do
    $j \leftarrow$ parent of $i$;
    if H[i] < H[j] then
        swap $H[i]$ and $H[j]$;
        $i \leftarrow j$;
    else
        break
    end
end
```

The running time of Algorithm 4 depends on the number of levels the elements at position i need to be swapped. From Theorem 2.1.3, we have that the height of a complete binary tree is logarithmic in the heap size. Thus, Algorithm 4 has a worst-case running time of $\mathcal{O}(\log n)$, where n is the size of the heap.

Algorithm 5: ADD

```
Data: Array based binary heap $H$ and element $e$
Result: Adds $e$ to $H$ while satisfying the heap invariant
Add $e$ to the end of $H$;
SWIM(H, |H| - 1);
```

Algorithm 5 admits a worst-case running time of $\mathcal{O}(\log n)$ [12]. However, if the input data has a specific pattern or is already partially sorted, it may result in fewer swaps during the SWIM operation. In 1967 the paper [14] was published, proving that the average number of levels a random element moves up when inserted into a random binary heap is bounded by a constant, giving a constant average-case running time for ADD.

Theorem 2.3.4 ([16]). *Algorithm 5 correctly inserts an element into a binary heap of size n while maintaining the heap property in time $\mathcal{O}(\log(n))$.*

Chapter 3

Naive approaches

In this chapter, we discuss two simple and commonly used approaches to k-insertion into a binary heap of size n: BULK INSERTION WITH HEAPIFY and SEQUENTIAL INSERTION. BULK INSERTION WITH HEAPIFY involves directly inserting the k elements into the binary heap in a bulk manner and then reconstructing the heap property using the HEAPIFY algorithm discussed in Section 2.3.3. SEQUENTIAL INSERTION, on the other hand, involves sequentially inserting each of the k elements one by one using the insertion method described in Section 2.3.4.2. We first discuss BULK INSERTION WITH HEAPIFY in Section 3.1, and then move on to SEQUENTIAL INSERTION in Section 3.2. Both algorithms consider heaps implemented by an array. We explore the steps involved in each approach, including the time complexity. Throughout this section, n will denote the size of the initial heap, and k will indicate the number of elements to be added to the heap.

3.1 Bulk insertion with heapify

The main idea of bulk insertion is to directly add the k elements into the binary heap in a batch or bulk manner. The process involves inserting the elements at the end of the binary heap, effectively creating a "partial" heap, and restoring the heap property. The most common way to restore the heap property is by using the

HEAPIFY algorithm described in Section 2.3.3.

Algorithm 6: BULK INSERTION WITH HEAPIFY

```
**Data:** Array based binary heap $H$ and collection $c$ of elements
**Result:** Adds the elements of $c$ to $H$ while keeping the heap invariant intact
Append the elements of $c$ to the end of $H$;
HEAPIFY(H);
```

From Theorem 2.3.2, we can conclude that the running time of Algorithm 6 is $\mathcal{O}(n+k)$. However, if for example the elements are already partially ordered, such as in a sorted or partially sorted array, the HEAPIFY operation may require fewer comparisons and swaps, resulting in better performance.

3.2 Sequential insertion

SEQUENTIAL INSERTION involves inserting the k elements into the binary heap one by one. The elements are added using the insertion algorithm presented in Section 2.3.4. Each element is placed in the first available leaf position and then "bubbled up" the binary heap using the SWIM operation.

Algorithm 7: SEQUENTIAL INSERTION

```
**Data:** Array based binary heap $H$ and collection $c$ of elements
**Result:** Adds the elements of $c$ to $H$ while keeping the heap invariant intact
for $e$ in $c$ do
    ADD(H, e)
end
```

From Theorem 2.3.4, we obtain that the running time of Algorithm 7 is $\mathcal{O}(k \cdot \log(n+k))$ in the worst case; it performs k insertions at a cost of $\mathcal{O}(\log(n + k))$ each. As mentioned, Algorithm 5 admits a constant average-case running time. Consequently, Algorithm 7 has an average-case running time of $\mathcal{O}(k)$.

3.3 Sequential vs. Bulk Insertion with heapify

In general, Sequential insertion is often preferred over Bulk insertion with heapify for k-insertion into a binary heap. This is due to the better time complexity per element and the simplicity of the algorithm. The time complexity of Sequential insertion, specifically its average-case running time, is favorable compared to Bulk insertion with heapify and ensures efficient performance, especially for small datasets. The Sequential insertion algorithm offers a simple and intuitive solution to the k-insertion problem. Each element is added individually, and the heap is adjusted to maintain the heap property. This straightforward process makes it easy to understand and implement, reducing the chances of errors or complexities in the code. Consequently, Sequential insertion has gained popularity and is widely adopted in various programming languages and libraries. For example, in Java, the `PriorityQueue` class, which uses a binary heap, employs Sequential insertion as the default insertion method when adding a collection of elements to the binary heap.

However, Bulk insertion with heapify can be more efficient in certain cases and is typically used when k is large compared to the initial heap size. This is because Bulk insertion with heapify reduces the number of operations required compared to individual element insertion. When inserting elements individually into a binary heap, each element must be inserted and adjusted to maintain the heap property. Bulk insertion with heapify, on the other hand, inserts multiple elements into the binary heap at once and then rearranges them to satisfy the heap property in a single operation. By using Bulk insertion with heapify for k-insertion, we eliminate the need for k individual insertion operations and their associated adjustments. This can lead to significant time savings, especially when k is large.

Bulk insertion with heapify is more commonly used for heap construction compared to Sequential insertion. When initializing a binary heap with Bulk insertion with heapify, the initial heap is empty, and the algorithm is analogous to Algorithm 2. Thus it admits an $\mathcal{O}(n)$ running time. Applying Heapify to the entire dataset eliminates the per-element adjustment step needed by the Sequential insertion algorithm and builds the binary heap in linear time. This can result in significant time savings, especially for large datasets.

Chapter 4

Bulk insertion with walkdown

The paper [15] presents an algorithm for merging priority queues organized as binary heaps of size n and k. The algorithm runs in $\mathcal{O}(k + \log(k) \cdot \log(n))$ time when considering binary heaps implemented by an array. As a by-product, an efficient algorithm for k-insertion into a binary heap is obtained. First, construct a binary heap from the k elements using the Heapify method presented in Section 2.3.3, and then merge the two heaps using the algorithm presented in [15].

Theorem 4.0.1 ([15])**.** *A collection of k elements can be inserted* [*] *into a binary heap of size n in time $\mathcal{O}(k + \log(n) \cdot \log(k))$.*

This approach improves the running time of both Sequential insertion and Bulk insertion with walk down presented in Chapter 3, the most commonly used k-insertion algorithms for binary heaps. However, the algorithm for merging two binary heaps presented in [15] is complex and consists of numerous amount of special cases. In addition, the algorithm runs out-of-place and requires moving subheaps of the original heap into separate storage before performing operations on it and then moving it back. This moving process is also not described properly in the paper. This

[*]Here the length of the walk down path is approximated by $\mathcal{O}(\log(n))$. A better approximation is $\mathcal{O}(\log(n) - \log(k))$ which reduces the total runtime to $\mathcal{O}(k + \log(k) \cdot \log(\frac{n}{k}))$

makes the algorithm hard to implement correctly with the desired running time. We will therefore not consider this algorithm directly and we will instead present a new k-insertion algorithm, inspired by some of the results presented in [15], that retains the running time $\mathcal{O}(k + \log(k) \cdot \log(\frac{n+k}{k}))$. In this chapter, we present this algorithm. The algorithm is referred to as Bulk insertion with walk down.

We will define *slots* as leaves added to a binary heap when a collection of elements is inserted into the heap. We say that a node u *covers* a group of slots if all slots are descendants of u. Throughout this chapter, n will denote the size of the initial binary heap, and k indicates the number of elements to be added to the binary heap.

4.1 Subroutines

Bulk insertion with walk down is a bulk insertion method that combines ideas presented in the paper [15] with the Heapify algorithm discussed in Section 2.3.3. The algorithm first adds the k elements from the collection to the end of the binary heap in a bulk manner and then restores the heap property using three subroutines: Find two nodes, Heapify subheap, and Walk down. These three procedures will be explained in Sections 4.1.1 - 4.1.3.

4.1.1 Find two nodes

The algorithm Find two nodes takes as input a binary heap of size n and a number k and finds two nodes p_l and p_r of the input heap of height $\mathcal{O}(\log(k))$ that together cover the k first slots of the heap. The existence of these nodes is established in Theorem 4.1.1.

Theorem 4.1.1 ([15]). *Let H be a binary heap of size n. There exist two nodes p_1 and p_r of H such that:*

(i) The first k slots of H are descendants of either p_l or p_r, i.e they are covered by p_l and p_r, and

(ii) Both p_l and p_r have height $\lfloor \log_2(k) \rfloor + 1$ in H.

The algorithm *find_2_nodes* in [15] establishes the existence of p_l and p_r as described above. However, this algorithm runs in time $\mathcal{O}(\log(n))$. This is due to the fact that the algorithm uses a top-down approach, starting from the least common ancestor p of all k slots and moving down in a specific manner until it reaches p_l and p_r. The length of the path from p to p_l and p_r is at most $\mathcal{O}(\log(n))$, and thus the algorithm runs in $\mathcal{O}(\log(n))$ steps [15]. A bottom-up approach is one way to avoid the need for the least common ancestor and reduces the running time to $\mathcal{O}(\log(k))$.

Algorithm 8: FIND TWO NODES

```
Data: Array based binary heap $H$ and number $k$
Result: Two indices $p_l$ and $p_r$ that together covers the first $k$ slots of $H$
if $k \geq \frac{n}{2}$ then
    return (0, 0)
end
$p_l \leftarrow$ first slot of $H$;
$p_r \leftarrow k^{th}$ slot of $H$;
for $i$ = 0 to $\lfloor \log_2(k) \rfloor + 1$ do
    $p_l \leftarrow$ parent of $p_l$;
    $p_r \leftarrow$ parent of $p_r$;
    if $p_l == p_r$ then
        break;
    end
end
return ($p_l$, $p_r$)
```

When moving p_l and p_r up the tree in the for loop, it might be the case that we reach the least common ancestor of p_l and p_r, i.e., $p_l = p_r$. In that case, the algorithm returns a tuple where both elements contain the least common ancestor of all k slots. The correctness and time complexity of Algorithm 8 are established in Theorem 4.1.2.

Theorem 4.1.2. *Algorithm 8 finds two nodes p_l and p_r as described in Theorem 4.1.1 in time $\mathcal{O}(\log(k))$.*

Proof. Let $h = \lfloor \log_2(k) \rfloor$, and let p be the least common ancestor of all k slots. Since p covers all slots, so do the left child l and the right child r of p together. Additionally, the rightmost descendant $p_l child_r$ of l and the leftmost descendant $p_r child_l$ of r are both slots and they are also adjacent. Because the slots are in consecutive locations, there are at most $k-1$ slots to the left of $p_l child_r$. Moreover, these slots are covered by the h^{th} ancestor of $p_l child_r$. This node, which has height $h+1$, will be referred to as p_l. Since p_l covers all slots to the left of $p_l child_r$, it must be the case that p_l is a common ancestor of $p_l child_r$ and the leftmost slot. Let p_r be the h^{th} ancestor of $p_l child_r$. Then a similar argument holds for p_r, showing that p_r covers the nodes not covered by p_l and that this node is a common ancestor of $p_r child_r$ and the rightmost slot. Thus, it is true that the k slots are covered by the h^{th} ancestors of the leftmost and rightmost slots.

The running time of the algorithm follows immediately from the fact that the for loop of the algorithm runs at most $\mathcal{O}(\log(k))$ times, and the only work being done at each iteration is moving p_l and p_r to their parent, which can be done in constant time.

□

4.1.2 Heapify subheap

One of the subroutines of the reconstruction steps of Heapify with walk down involves performing the routine Heapify on the subheaps rooted at the nodes returned by the algorithm Find two nodes. One way to do this is to feed the subheaps rooted at p_l and p_r directly into Algorithm 2. However, this approach would require moving the subheaps to a separate array, performing the procedure, and then moving them back into the original heap. To avoid the need for extra storage, we have therefore chosen to develop an algorithm similar to Algorithm 2 that takes as input a binary heap and a node of the heap and in-place performs the procedure HEAPIFY on the subheap rooted at the input index.

Algorithm 9: HEAPIFY SUBHEAP

```
Data: Array based binary heap $H$ and index $i \in \{0, 1, ..., |H| - 1\}$
Result: Restores the heap property of $H_i$
$right \leftarrow$ Rightmost leaf of $H_i$;
$left \leftarrow$ leftmost node in $H_i$ on the same layer as $right$;
while $right \neq i$ do
    for $j = left$ to $right$ do
        SINK(H, j);
    end
    $left \leftarrow$ parent of $left$;
    $right \leftarrow$ parent of $right$;
end
```

Theorem 4.1.3. *Algorithm 9 correctly ensures that H_i satisfies the heap property in time $\mathcal{O}(k)$ where k is the size of H_i*

Proof. Both the correctness and the time complexity of the algorithm follow immediately from Theorem 2.3.2. □

4.1.3 Walk down

Consider the task of inserting a small number of elements into a large binary heap using the algorithm SEQUENTIAL INSERTION. In the worst case, each new element needs to be swapped from a leaf position to the root of the binary heap. In some cases, the paths taken by the consecutive SWIM operations have some edges in common.

The algorithm WALK DOWN ensures that this path called the *walk done path*, will only be traversed once. The procedure is inspired by the algorithm *walk_down* in [15], with slight modifications, providing a solution for situations where the heap property is broken between a node and its parent, and one wants to restore the heap property in the entire heap, not just parts of it.

WALK DOWN takes as input a binary heap H and a valid index i of H, and proceeds as follows: start with the node at index i and compare it to the root of H. If the node is smaller than the root of H, exchange the two and perform the SINK procedure with H and i as input. Repeat this step for successive nodes on the path from the root of H and the node at index i, i.e., the nodes on the walk down path. The correctness of the algorithm is established in Theorem 4.1.4.

Algorithm 10: WALK DOWN

```
Data: Array based binary heap $H$ and index $i \in \{0, 1, ..., |H| - 1\}$ where
      the node at index $i$ might be smaller than its parent
Result: Restores the heap property of $H$
$j \leftarrow 0$;
while $i \neq j$ do
    if H[i] < H[j] then
        swap $H[i]$ and $H[j]$;
        SINK$(H, i)$;
    end
    $j \leftarrow$ next node on the path from 0 to $i$ to;
end
```

Theorem 4.1.4. *Let H be a binary heap and u a node of H at index i. Then after a call to Algorithm 10 with H and i as input:*

(a) *The heap property is satisfied in u;*

(b) *the heap property of H is maintained.*

Proof. By induction, we will show that after the h^{th} element on the walk down path has been considered, the heap property is maintained in the k first nodes on the walk down path and that the heap property is satisfied in H_u.

For $h = 1$, we consider the root, and the results follow immediately. Let $h > 1$. By induction, the elements 1 to $h-1$ on the path are less than or equal to the h^{th} element and less than or equal to all elements in H_u. Also, the heap property is satisfied in H_u. In processing the h^{th} element two cases arises:

- $H[h] > u$ and the procedure exchanges the h^{th} and u, thereby replacing u with a smaller element, which by induction is larger than the $k-1$ preceding elements. This maintains the heap structure of H. Using the procedure SINK, the heap property of H_u is subsequently restored.
- $H[h] \leq u$ and no exchange is made; thus, the results follow immediately.

Subsequently, the proof is completed by letting h equal to i. □

One thing worth mentioning about Algorithm 10 is the walk down path. Since each internal node in a binary heap has two children, it is not possible to efficiently find the path from the root to i directly. Instead, the walk down path can easily be located by moving from i to the root, storing the traversed nodes in a list, and then reversing the list. The walk down path can be obtained either directly through Algorithm 10 or passed as an input to the algorithm. Regardless of the approach, the path needs to be stored in a separate storage location with a size proportional to the length of the path.

Theorem 4.1.5. *The running time of Algorithm 10 is $\mathcal{O}(h \cdot (\log(n) - h))$ where n is the size of the input heap and h the height of the subheap rooted at the input index.*

Proof. Observe that the running time of Algorithm 10 is the length of the walk down path times the height of H_i where H is the input heap and i is the input index.

From Theorem 2.1.3 the height of H is $\mathcal{O}(\log(n))$. Thus, the length of the walk down path is $\mathcal{O}(\log(n) - h)$. From the observation above the running time of Algorithm 10 is therefore $\mathcal{O}(h \cdot (\log(n) - h))$ □

4.2 The algorithm

As mentioned before, Bulk insertion with walk down is a bulk insertion method where the reconstruction step combines the three procedures Find two nodes, Heapify subheap, and Walk down presented in Section 4.1. The algorithm first adds the k elements from the collection at the end of the binary heap in a bulk manner. Then, it finds two the nodes p_l and p_r covering the k slots using the algorithm Find two nodes. Next, it performs the procedure Heapify subheap twice with p_l and p_r as inputs, obtaining a partial binary heap where the only places where the heap property might be broken are between p_l and p_r and their parents. To fix this, Bulk insertion with walk down then performs Walk down two times with p_l and p_r as input.

Algorithm 11: Bulk insertion with walkdown

```
Data: Array based binary heap $H$ and collection $c$ of elements
Result: Adds the elements of $c$ to $H$ while keeping the heap invariant intact
Append the elements of $c$ to the end of $H$;
$k \leftarrow$ size of $c$;
$(p_l, p_r) \leftarrow$ find two nodes(H, k);
Heapify($H$, $p_l$);
Walk down($H$, $p_l$);
Heapify($H$, $p_r$);
Walk down($H$, $p_r$);
```

Theorem 4.2.1. *Algorithm 11 correctly inserts a collection of k elements into a binary heap of size n in time $\mathcal{O}(k + \log(k) \cdot \log(\frac{n+k}{k}))$ while maintaining the heap* *property.*

Proof. The correctness of the algorithm follows immediately from the Theorems 4.1.2, 4.1.3, and 4.1.4.

By Theorem 4.1.2, both p_l and p_r can be found in time $\mathcal{O}(\log(k))$ and have height $\log_2(k)$. Combining this and Theorem 4.1.5, the Algorithm 8 obtains a running time of

$$\mathcal{O}(\log(k) \cdot (\log(n+k) - \log(k))) = \mathcal{O}(\log(k) \cdot \log(\frac{n+k}{k}))$$

.

Note that the sizes of the subheaps rooted at p_l and p_r are $\mathcal{O}(k)$. Combining this and Theorem 4.1.3, Algorithm 9 runs in time $\mathcal{O}(k)$. Thus the total running time of Algorithm 11 is

$$\mathcal{O}(k + \log(k) + \log(k) \cdot \log(\frac{n+k}{k})) = \mathcal{O}(k + \log(k) \cdot \log(\frac{n+k}{k}))$$

□

4.3 Optimization

Certain optimizations have been incorporated in implementing the Bulk insertion with walk down algorithm. These optimizations have been intentionally omitted from the algorithm description. By isolating them from the main algorithm description, we aim to provide a clear and concise understanding of the fundamental workings of the algorithm. This subsection is therefore dedicated to presenting these optimizations in detail. It should be noted that while these optimizations do not alter the worst-case running time of the algorithm, they significantly impact its practical performance.

The first optimization relates to the behavior of Bulk insertion with walk down when it calls Heapify subheap and Walk down on both p_l and p_r. In specific scenarios, the paths followed by the Walk down procedure for p_l and p_r may overlap or even be identical. In some cases, it, therefore, becomes more efficient to disregard p_l and p_r and instead select a single node higher in the tree that has both p_l and p_r as descendants. This eliminates calling Heapify subheap and Walk down two times and instead performs the procedures once. However, this increases the number of nodes in the sub-heap considered in Heapify subheap and Walk down. One situation where we know it is advantageous to do this is when p_l and p_r are siblings, meaning they share the same parent. By choosing the parent, we consider one extra node but avoid calling Heapify subheap and Walk down two times and instead perform both procedures once, thus reducing redundant work.

The following optimization also relates p_l and p_r and the fact that in some cases, p_l and p_r cover more slots than necessary, and it is safe to consider one of their descendants instead. Since the running time of Heapify subheap is equal to the number of nodes in the subheaps under p_l and p_r while the running time of Walk down is equal to the height of the same subheaps times $\log_2(n)$ it is beneficial to reduce the height of the subheaps as much as possible. One way of reducing the heights of p_l and p_r as much as possible while still making sure that they together cover all slots is to realize that when moving p_l and p_r up the heap, and p_l is a right child the parent of p_l might cover nodes to the left of the first slot. So moving to its parent increase the height of the subheap considered by Heapify subheap without covering more slot. Similarly, when moving up the heap and p_r is a left child, its parent might cover leaf positions to the right of the k^{th} slot. However, when implementing this optimization, it is essential to realize that even though in one of the iterations of the for loop of Find two nodes, it might be the case that p_l is the right child one of its ancestors still might cover some of the k slots, and we can not terminate the loop instantly. This also goes for p_r. This must be taken into consideration when implementing this optimization.

The final optimization is regarding the algorithm Walk down. This algorithm involves traversing the path from the root of the binary heap to a given index, denoted as i, and comparing each element with the node at index i. A swap is performed if the node on the path is larger than the node at index i. As we move to the next node on the path, we are essentially moving to a child node, which, according to the heap property, is always larger than its parent. To minimize unnecessary swaps, we can take advantage of this and ensure that the largest available value is always placed at index i. To accomplish this, we store the values of the nodes on the walk down path in a double-ended list. By the heap property, the smallest element is kept at the front of the list, while the largest element is at the end. We then compare the front of the list with the element at index i. To simplify the explanation, let us denote the current node at index i as u and the current position on the walk down path as j. If the element at the top of the list is smaller than u, we know that this element is the smallest available element. Consequently, we remove the top element from the list and assign it to index j. On the other hand, if the element at the top of the list is larger than u, we know that u is the smallest available value and should be placed at index j. However, instead of moving the element at index j down the tree to index i, we want to place the largest available value at index i. Accordingly, we remove the last element from the double-ended list and assign it to index i before calling the Sink operation. This process is repeated for the next node on the walk down path. By storing all the values of the nodes on the walk down path in the double-ended list and always comparing u with the top of the list, we ensure that no values are skipped. Furthermore, since we always place the largest available value at index i and the smallest at index j, we ensure that the heap property is maintained and avoid unnecessary swaps in subsequent steps, thus reducing extra work.

Chapter 5

Experimental results

In this chapter we investigate and compare the behavior of the empirically computed running times of the three k-insertion algorithms discussed in this thesis: Sequential insertion, Bulk insertion with heapify and Bulk insertion with walk down. We investigate the behavior of the algorithms both on random input and worst-case input. First, in Section 5.1 we will go through how the experiments are conducted and the experimental setup. Then, in Section 5.2 and Section 5.3, we present and discuss the results of running each algorithm on the different datasets. Throughout this chapter, n will denote the size of the original binary heap while k is the size of the collection added to the binary heap.

5.1 Setup

To assess and compare the performance of the algorithms, we implement them in Java and carry out timed experiments on different data sets. We initially create an empty binary heap for each insertion method. Following this, we generate two distinct data sets; one of size n and one of size k. The data set of size n is referred to as the *initial data*, while the data set of size k is referred to as the *input data*. The standard insertion method is employed to add the initial data to the binary heaps, ensuring that the ordering of the elements in the three heaps remains the same before applying the k-insertion algorithms. Afterward, the time each of the three k-insertion algorithms takes to add the input data to their respective binary heap is recorded.

Given that the worst-case running times of Bulk insertion with heapify is $\mathcal{O}(k+n)$, while the running times of Sequential insertion and Bulk insertion with walk down are $\mathcal{O}(k \cdot \log(n+k))$ and $\mathcal{O}(k + \log(k) \cdot \log(\frac{n+k}{k}))$ respectively, it is expected, as discussed in Section 3.3, that the performance of these algorithms will be significantly influenced by the size of k compared to n. To analyze this relationship, we, therefore, fix the value of n and examine how the algorithms perform on different values of k. Specifically, we set $n = 2^{24}$ to create a perfect binary heap, and vary k as 1%, 10%, 20%, 30%, and 40% of the value of n.

To measure the execution time of each algorithm, we use the `nanoTime()` function in Java's `Java.lang.System` library. This method retrieves the current value of the running Java Virtual Machine's (JVM) time source in nanoseconds. The method provides nanosecond precision but not necessarily nanosecond accuracy. The value returned represents the number of nanoseconds elapsed since a fixed but arbitrary point in time and becomes meaningful when calculating the difference between two such values [13]. Consequently, we invoke the method before and after the insertion step and examine the resulting difference. Measuring the running times of the algorithms like this might cause some problems. When running a Java program, numerous operations are performed in the background. The frequency of these operations is often sporadic and results in a temporary halt in the execution of the running program. So when timing an algorithm like this, we are also timing the background operations. One of these operations is the garbage collector. In Java, garbage collection is the process of identifying and reclaiming unused objects, referred to as garbage, in order to free up memory and resources for other objects. During the garbage collection, the running program may need to pause the program temporarily. The duration of these pauses can vary, leading to inaccuracies in the recorded running times [4]. Hence, it is important to consider this when analyzing the results. Consequently, we disregard certain instances of the longest running times in certain cases.

To illustrate the running times of the algorithms, we utilize a line diagram. Prior to creating the plot, we arrange the running time values of each algorithm in increasing order. This sorting process makes it easier to understand and interpret the plots. By organizing the running times in ascending order, we generate a line plot that effectively demonstrates the distribution of performance values. This visualization can help identify patterns or trends. Additionally, sorting the running times enables us to detect any outliers or extreme values that might distort the overall distribution. In conclusion, sorting the running times before plotting contributes to a more comprehensive understanding of the performance of each algorithm.

When conducting experiments like this, it is worth mentioning that the results depend on the implementation of the algorithms. As Java's `PriorityQueue` class is implemented with a binary heap and uses SEQUENTIAL INSERTION as its default k-insertion method, we, therefore, use this directly to obtain the result for SEQUENTIAL INSERTION. Also, we use Java's implementation of the HEAPIFY method to get the result for BULK INSERTION WITH HEAPIFY. As for BULK INSERTION WITH WALK DOWN, we implement this algorithm ourselves using as much as possible of the code from Javas's standard library. We also implement the optimizations discussed in Section 4.3. The fact that we are using Java's built-in methods for some of the algorithms and that these are probably highly optimized might impact the results.

5.2 Random input

In this section, we examine the running times of the three algorithms on random data. The purpose of this is to compare their behavior when presented with randomized input. To create the random scenarios, we generate both random initial and input data without imposing any constraints on the elements of the sets.

5.2.1 Expectations

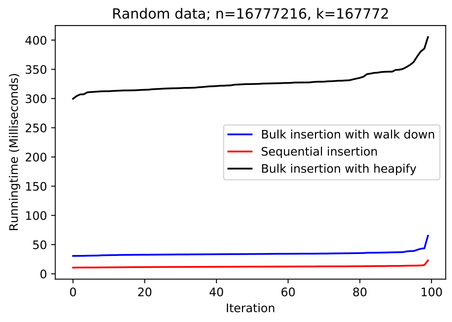
Since Sequential insertion has an average-case running time of $\mathcal{O}(k)$ and this is optimal for k-insertion, we don not anticipate that any of the algorithms will surpass Sequential insertion in terms of performance on random data, regardless of the relative sizes of k and n.

When k is small, we expect Bulk insertion with heapify to have a longer running time than the other algorithms. However, as k increases, we anticipate that the running time of Bulk insertion with heapify will become more comparable to the others. This is because Bulk insertion with heapify always considers the entire initial heap. In contrast, the other two algorithms only consider a portion of the initial heap determined by the size of k. Consequently, as k increases while n remains constant, Bulk insertion with walk down and Sequential insertion will consider a more significant part of the initial heap.

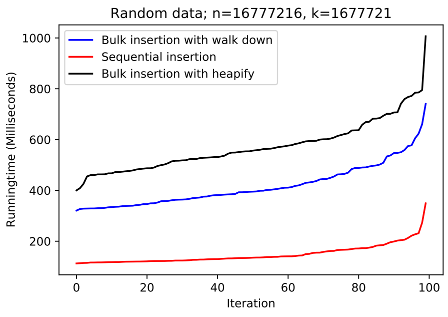
Although we anticipate that the running time of Bulk insertion with heapify will approach the others as k increases, we do not expect it to be faster than Bulk insertion with walk down. This is because, as k increases, the majority of the work performed by Bulk insertion with walk down is the Heapify subheap operation. As a result, the two algorithms become more similar in their performance. However, Bulk insertion with walk down operates on a significantly smaller portion of the binary heap compared to Bulk insertion with heapify. Hence, we anticipate it to be faster.

5.2.2 Results

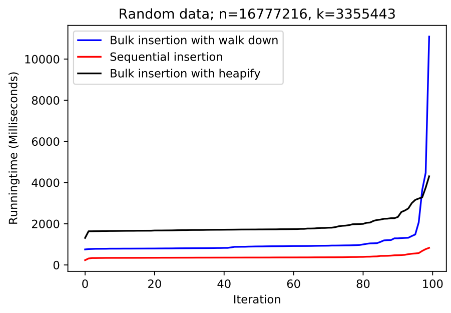
Figure 5.1 depicts the running times of the three k-insertion algorithms on random data, with k being relatively small compared to n. As anticipated, Bulk insertion with heapify consistently exhibits the longest running times in such cases. Furthermore, Sequential insertion and Bulk insertion with walk down display similar behavior in this scenario. However, as expected, Sequential insertion is marginally faster overall.



(a) k is 1% of n



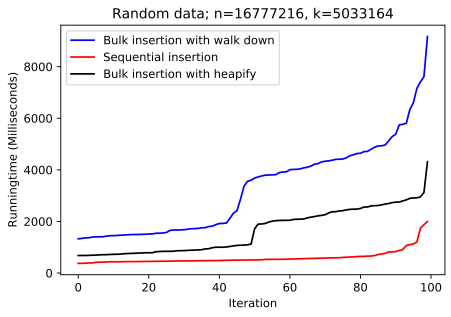
(b) k is 10% of n



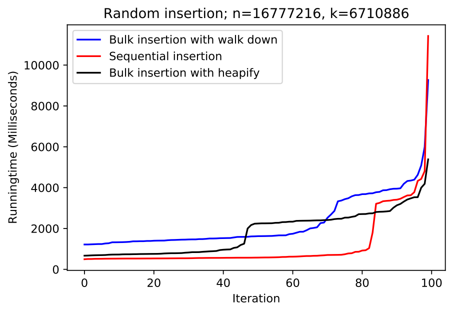
(c) k is 20% of n

Figure 5.1: The running times of Sequential insertion, Bulk insertion with heapify and Bulk insertion with walk down on 100 different randomized data sets with different values of k and $n = 2^{24}$.

Figure 5.2 illustrates the running times of the three algorithms on random data for larger values of k compared to the ones in Figure 5.1. Once again, we observe that Sequential insertion achieves the lowest running time and performs the best. As anticipated, Bulk insertion with heapify exhibits noticeable improvements as k increases. Specifically, when k transitions from 20% to 30% of n, Bulk insertion with heapify begins to outperform Bulk insertion with walk down. However, as k reaches 40% of n, the running times of all three algorithms become more comparable, making it less evident which bulk insertion method performs the best in this scenario.



(a) k is 30% of n



(b) k is 40% of n

Figure 5.2: The running times of Sequential insertion, Bulk insertion with heapify and Bulk insertion with walk down on 100 different randomized data sets with different values of k and $n = 2^{24}$.

The surprising result is that Bulk insertion with heapify outperforms Bulk insertion with walk down in some instances, contrary to expectations. When k constitutes 40% of n, there is some variability in which bulk insertion method performs better. This might have to do with the fact that we are considering random data. However, when k is 30% of n, Bulk insertion with heapify consistently performs faster than Bulk insertion with walk down across all datasets. As explained in Section 5.2.1, Bulk insertion with walk down operates on a significantly smaller portion of the binary heap compared to Bulk insertion with heapify, thus we initially expect it to be faster. Considering that Bulk insertion with heapify is mostly quicker than Bulk insertion with walk down when k is 30% of the value of n, this behavior might have something do to with how Bulk insertion with walk down works on this particular value of k

One noteworthy finding is the sudden spike in running time for each algorithm. This is particularly evident in Figure 5.1 **(c)**. The plot illustrates that the running time of Bulk insertion with walk down remains relatively stable for most datasets, except the maximum running time, which is notably larger. Considering that we are dealing with random data, some variation in the running times among different datasets is expected, but the magnitude of this peak is remarkable. Additionally, none of the other algorithms display similar behavior. Therefore, it is highly probable that this is attributed to the timing and the Java garbage collector mentioned in Section 5.1 rather than the actual algorithms themselves.

5.3 Worst case input

In this test case we aim to evaluate the practical performance of algorithms in their worst-case scenario. The datasets have input elements that are smaller than the initial elements, requiring all input elements to be placed at the top of the binary heap. Furthermore, the input data is sorted in decreasing order. As a result, the Sequential insertion algorithm must place each added element at the top of the binary heap, showcasing its worst-case running time in practice. In the case of the Bulk insertion with walk down algorithm, when confronted with this scenario, it first moves the input elements to the top of the subheaps rooted at p_l and p_r. Additionally, every element on the paths during the walk down process must be swapped with the roots of the subheaps under p_l and p_r and moved down the heap. Hence, the running time of Bulk insertion with walk down on this dataset is equal to its worst-case running time. Similarly, Bulk insertion with heapify also attains its worst-case running time in this particular scenario as each element needs to be moved to the top of the heap. Consequently, these types of datasets provoke the worst-case running time for each of the tree k-insertion algorithms.

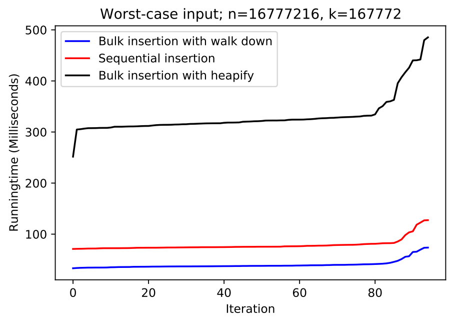
5.3.1 Expectations

Considering that the worst case running time of Bulk insertion with walk down is $\mathcal{O}(k + \log(k) \cdot \log(\frac{n+k}{k}))$ while it is $\mathcal{O}(k + n)$ and $\mathcal{O}(k \cdot \log(n + k))$ for Bulk insertion with heapify and Sequential insertion, we expect Bulk insertion with walk down to perform the best.

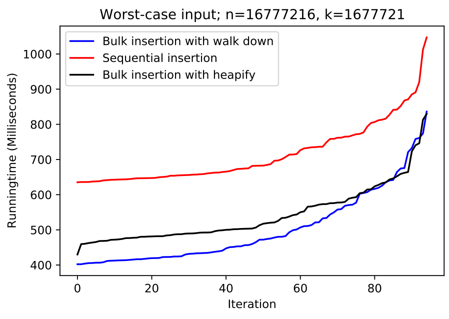
When it comes to Bulk insertion with heapify and Sequential insertion, we expect Sequential insertion to have a lower running time for small values of k compared to Bulk insertion with heapify. As k increases in size, we expect Bulk insertion with heapify to become faster. This was discussed in Section 5.2.1 for the random case and is due to the fact that Bulk insertion with heapify always considers the entire binary heap, while Sequential insertion considers a portion of the binary heap determined by the size of k.

5.3.2 Results

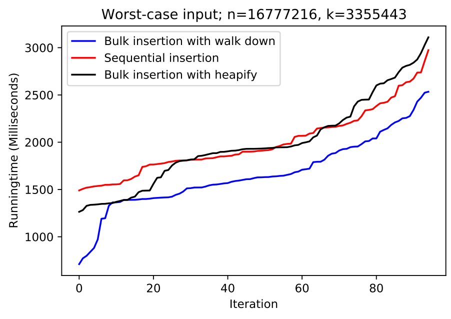
When conducting these experiments, we ran the algorithms on 100 datasets. However, we have opted to exclude the five largest running times from all plots. The reason for this decision is that the algorithms exhibit significantly longer running times on a few datasets compared to the rest. Including these extremely high running times in the plots would render them unreadable, and no useful information could be derived from them. This behavior was also observed in Section 5.2, where it was concluded that the likely cause of this is the implementation of the timing and the background operations performed when running the program. Since the highest running times are likely caused by Java's program execution rather than the actual algorithm performance, we chose to disregard them to enhance readability.



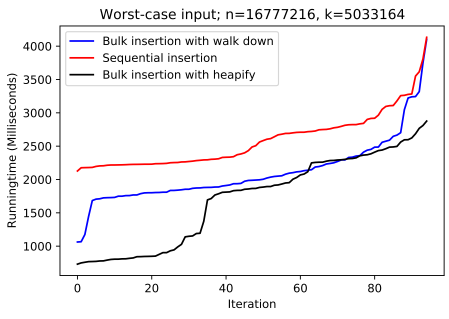
(a) k is 1% of n



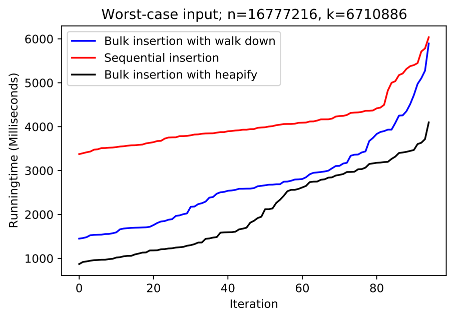
(b) k is 10% of n



(c) k is 20% of n



(d) k is 30% of n



(e) k is 40% of n

Figure 5.3: The running times of Sequential insertion, Bulk insertion with heapify and Bulk insertion with walk down on 95 different data sets where the elements of the input data are smaller than the initial data. The value of k is different in each plot while the value of n is consistently 2^{24}.

Figure 5.3 illustrates the running times of the three algorithms on various datasets with smaller values of input data compared to the initial data. Consistent with expectations, when k is 1% of n and relatively small, BULK INSERTION WITH WALK DOWN demonstrates the lowest overall running time, while BULK INSERTION WITH HEAPIFY exhibits the poorest performance. As the size of k increases, SEQUENTIAL INSERTION becomes less efficient and is surpassed by the BULK INSERTION WITH HEAPIFY algorithm. As expected, BULK INSERTION WITH HEAPIFY demonstrates significantly improved efficiency as k increases in size. Surprisingly, BULK INSERTION WITH HEAPIFY begins to outperform BULK INSERTION WITH WALK DOWN when k exceeds 20% of n. This behavior was also observed in the random case, specifically when k constituted 30% of the value of n, but it is even more pronounced in these plots.

One hypothesis for why BULK INSERTION WITH HEAPIFY is faster than BULK INSERTION WITH WALK DOWN in some cases even though BULK INSERTION WITH WALK DOWN has a lower wost-case running time is the extra work done by the algorithm. BULK INSERTION WITH WALK DOWN first needs to find the two nodes p_l and p_r, which in practice involves traversing up and down the binary heap twice. Also, as mentioned before, when k is large, the majority of the work done by the algorithm is the HEAPIFY SUBHEAP procedure. Since HEAPIFY SUBHEAP is implemented in place, it needs to loop through the nodes, level by level, keeping a pointer on the leftmost and rightmost node on the current level. BULK INSERTION WITH HEAPIFY can simply loop through the nodes in the array representation from right to left. This might make BULK INSERTION WITH WALK DOWN less efficient on large datasets and might overshadow the time gained by only considering a fraction of the binary heap. However, the difference in number of nodes considered by the two algorithms is significantly large. For example, if the node returned by FIND TWO NODES is the child of the root, HEAPIFY SUBHEAP operates on half as many nodes as BULK INSERTION WITH HEAPIFY. The difference becomes rapidly larger when the p_l and p_r are further down the tree. Considering that n is 2^{24} in the experiments, there is a noticeable difference in the number of nodes considered by the two algorithms. Thus, it is not certain whether this hypothesis is true or not.

Another hypothesis pertains to the implementation of the algorithms. In our experiments, we utilize code from Java's standard library directly to implement Bulk insertion with heapify, meaning we do not implement this algorithm ourselves. This algorithm has likely been extensively optimized. Conversely, we implement the Bulk insertion with walk down algorithm by combining relevant code from Java's standard library with our own code. Both the Find two nodes and Walk down algorithms are implemented by us. It is possible that our implementations of these algorithms are not be the most optimal. This could therefore make Bulk insertion with heapify faster in practice than Bulk insertion with walk down, despite the theoretical expectation being the opposite. However, we cannot ascertain the accuracy of this hypothesis.

Chapter 6

Conclusion

From these results, it can be concluded that Sequential insertion behaves according to its theoretical running time. The algorithm has an average-case running time of $\mathcal{O}(k)$. Thus, we expect it to be superior to both Bulk insertion with walk down and Bulk insertion with heapify on random data; this is also the case in practice. The algorithm consistently exhibits the fastest running time on random data with Bulk insertion with walk down being just slightly slower in some cases. We have not conducted a formal analysis on the average-case running time on the two bulk insertion methods; however, these results suggest that the average-case running time of Bulk insertion with walk down may be within a constant factor of the one of Sequential insertion. This indicates an $\mathcal{O}(k)$ average-case running time of Bulk insertion with walk down. Thus, there might be a possibility of making Bulk insertion with walk down faster than Sequential insertion by further improving the algorithm.

The Sequential insertion algorithm has a worst-case running time of $\mathcal{O}(k \cdot \log(n + k))$; thus, we do not expect it to perform the best on worst-case data. This is also the case in practice. Its performance is satisfactory only when the value of k is considerably small compared to n, and then its running time rapidly increases as k increases and gets outperformed by the other algorithms. Thus, it can be concluded that Sequential insertion is a good choice of k-insertion algorithm when presented with random data. However, when faced with worst-case input, this is not the case.

Regarding the two bulk insertion methods, we expect Bulk insertion with walk down to be noticeably faster than Bulk insertion with heapify on worst-case input on all values of k considered in the experiments. However, this is not the case in practice. On worst-case input, Bulk insertion with walk down demonstrates superior performance among all the algorithms when k is small and below 20% of the value of n. However, as k exceeds this threshold, Bulk insertion with heapify outperforms the other methods. We also see tendencies to this in the random scenario; however, it is not as prominent here. The results may suggest that there are some situations where Bulk insertion with walk down doesn't perform as well as expected. The exact reason for this behavior is uncertain and could stem from various factors, none of which can be definitively identified in this thesis. In conclusion, Bulk insertion with walk down is the better choice of k-insertion method when faced with worst-case data and the size of the input data is small compared to the initial heap and as k increases in size Bulk insertion with heapify becomes a better choice. Nonetheless, these results suggest the possibility of optimizing Bulk insertion with walk down by directly utilizing Bulk insertion with heapify when k is large.

From these results, it can therefore be concluded that there is yet no known overall best k-insertion algorithm for binary heaps that performs the best on all types of data. On random data, Sequential insertion consistently exhibits an ideal running time compared to the other algorithms. However, Bulk insertion with walk down is not much slower in this scenario. According to the results, combining Bulk insertion with walk down with Bulk insertion with heapify for larger values of k can create a k-insertion algorithm that is superior to Sequential insertion on worst-case input for an extensive range of values of k, and also not much slower on random data. Thus, if one aims to improve the worst-case running time of k-insertion into a binary heap, one should opt for a combination of Bulk insertion with walk

DOWN and BULK INSERTION WITH HEAPIFY instead of SEQUENTIAL INSERTION.

6.1 Summary

In this thesis, we have considered k-insertion into a binary heap. We have developed the k-insertion algorithm BULK INSERTION WITH WALK inspired by the article [15] running in worst case time $\mathcal{O}(k + \log(k) \cdot \log(\frac{n+k}{k}))$. The algorithm is a bulk insertion algorithm where the reconstruction step consists of three subroutines; FIND TWO NODES, HEAPIFY SUBHEAP, and WALK DOWN. Therefore, it only needs to consider a small portion of the original binary heap and ensure that the paths taken by sequentially inserting the elements are only traversed once.

We have compared the algorithm to two of the most well-known and widespread k-insertion algorithms, namely SEQUENTIAL INSERTION and BULK INSERTION WITH HEAPIFY. We did this by implementing all algorithms in Java and then conducting timed tests on different types of datasets. We examined the behavior of the algorithms on two types of datasets, random data, and worst-case data. Throughout the experiments, we kept the value of n constant while varying the size of k between 1% and 40% of the value of n.

From the experimental results, we concluded that SEQUENTIAL INSERTION performed the best out of all the algorithms on random data, both in theory and practice. However, BULK INSERTION WITH WALK DOWN was only marginally slower, indicating an $\mathcal{O}(k)$ average-case runtime. The performance of SEQUENTIAL INSERTION degraded significantly on worst-case data as k increased. BULK INSERTION WITH WALK DOWN performed overall better than BULK INSERTION WITH HEAPIFY on random data, but there were situations where BULK INSERTION WITH HEAPIFY was faster. BULK INSERTION WITH WALK DOWN was faster than BULK INSERTION WITH HEAPIFY on worst-case input when the size of the input data was small, but as k increased, Bulk insertion with heapify started outperforming Bulk insertion with walk down. We are unsure why the algorithms exhibited this behavior, despite their theoretical worst-running time indicating the opposite. We presented some hypotheses for this results, none of which we could conclude definitely to be true. In the future, more research can therefore be done on this area. Nonetheless, from these results, we concluded that combining Bulk insertion with walk down and Bulk insertion with heapify for larger values of k creates a k-insertion algorithm that is superior to Sequential insertion in real-life applications.

6.2 Future work

This chapter briefly discusses some open questions regarding the k-insertion algorithm Bulk insertion with walk down presented in this thesis that we could not further investigate due to either timing restrictions or them falling outside the scope of this thesis.

For future work along the lines of k-insertion into a binary heap, the main focus should be on analyzing the threshold value for when it is more efficient to directly call the Bulk insertion with heapify algorithm instead of Bulk insertion with walk down. The results provided in Chapter 5 indicate that this could improve the practical performance of the algorithm significantly, both on random input and worst-case input. We conducted a rough analysis on this and loosely concluded that if k is greater than 50% of the value of n, it would be more efficient to use Bulk insertion with heapify directly. However, the results provided in Section 5.3.2 indicate that this threshold could be lowered. Therefore, a more thorough analysis can be conducted.

Also, for future work, an assessment of the practical performance of the three k-insertion algorithms, especially Bulk insertion with walk down, on different types of datasets than the one presented in this thesis can be performed. The worst-case data presented in this thesis is specifically constructed to display the worst-case behavior of the algorithms, while the random testcase demonstrates their behavior on random input. Although these two datasets give meaningful information on the behavior of the algorithms, they don't necessarily reflect all the different scenarios one might encounter in the real world. Examining Bulk insertion with walk down on other datasets would therefore give a more comprehensive picture of the algorithm's behavior in real-life scenarios.

Also, as it was demonstrated in Chapter 5, Sequential insertion is more efficient than Bulk insertion with walk down on random input. In contrast, the situation is opposite on worst-case input. An interesting analysis is therefore to first let the input data consist of random data and then let a larger and larger portion consist of small values that need to be placed at the top of the binary heap and compare the running times of the two algorithms and look at when they start surpassing each other.

Since Bulk insertion with walk down utilizes the methods Walk down and Heapify subheap the algorithm imposes an entirely different structure on the resulting binary heap than Sequential insertion and Bulk insertion with heapify. An interesting analysis is to study how this structure affect the running times of the other heap operations. More specifically, compare how consecutive Remove min operations behave after utilizing the three different k-insertion algorithms.

After spending some time pursuing the possibility of implementing a variation of the Bulk insertion with walk down algorithm for other kinds of heap implementations than the one used in this thesis, it was concluded that this lies outside the scope of this thesis. Introductory in Section 1, both the Fibonacci heap and the B-heap were mentioned. For future work, one could therefore explore how the Bulk insertion with walk down algorithm translates to these kinds of heap implementations.

Bibliography

[1] Ian Anderson. *A first course in discrete mathematics.* eng. London, 2001.

[2] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. “Strict Fibonacci Heaps”. In: *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing.* STOC ’12. New York, New York, USA: Association for Computing Machinery, 2012, pp. 1177–1184. ISBN: 9781450312455. DOI: `10.1145/2213977.2214082`. URL: `https://doi.org/10.1145/2213977.2214082`.

[3] Reinhard Diestel. *Graph Theory.* eng. Berlin, Heidelberg, 2017.

[4] Christine H. Flood. *Understanding Garbage Collectors.* Accessed on May 09, 2023. URL: `https://blogs.oracle.com/javamagazine/post/understanding-garbage-collectors`.

[5] Robert Floyd. “Algorithm 245: Treesort3”. In: *Communications of the ACM* 7.12 (1964), p. 701.

[6] Robert W Floyd. “Algorithm 113: Treesort”. In: *Communications of the ACM* 5.8 (1962), p. 434.

[7] G.N. Frederickson. “An Optimal Algorithm for Selection in a Min-Heap”. eng. In: *Information and computation* 104.2 (1993), pp. 197–214. ISSN: 0890-5401.

[8] Michael L. Fredman and Robert Endre Tarjan. “Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms”. In: *J. ACM* 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411. DOI: `10.1145/28869.28874`. URL: `https://doi.org/10.1145/28869.28874`.

[9] Rowan Garnier. *Discrete mathematics : proofs, structures and applications*. eng. Boca Raton, FL, 2009.

[10] Ellis Horowitz. *Fundamentals of data structures*. eng. London, 1977.

[11] Poul-Henning Kamp. “You’re Doing It Wrong”. In: *Commun. ACM* 53.7 (July 2010), pp. 55–59. ISSN: 0001-0782. DOI: `10.1145/1785414.1785434`. URL: `https://doi.org/10.1145/1785414.1785434`.

[12] Jon Kleinberg. *Algorithm design*. eng. Harlow, Essex, 2014.

[13] Oracle. *Class system*. Accessed on May 09, 2023. URL: `https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--`.

[14] Thomas Porter and Istvan Simon. “Random insertion into a priority queue structure”. In: *IEEE Transactions on Software Engineering* SE-1.3 (1975), pp. 292–298. DOI: `10.1109/TSE.1975.6312854`.

[15] Jörg -R. Sack and Thomas Strothotte. “An algorithm for merging meaps”. In: *Acta Informatica* 22.2 (1985), pp. 171–186. URL: `https://doi.org/10.1007/BF00264229`.

[16] Robert Sedgewick. *Algorithms*. eng. Upper Saddle River, N.J, 2011.

[17] John William Joseph Williams. “Heap Sort”. In: *Communications of the ACM* 7.6 (1964), pp. 347–349.