FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Experimental evaluation of worst-case optimal heaps
Student:	Adam Volek
Supervisor:	RNDr. Tomáš Valla, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2020/21

Instructions

Heap is a data structure supporting fast insert and extraction of minimum element, together with several other fast operations.
The theoretical lower bound is O(log n) time for extraction and O(1) for the remaining operations in worst-case.
Recently, several data structures meeting this lower bound have been published. We aim to contribute to this topic by experimental evaluation of selected recently published heaps (e.g.
Brodal, G. S. L.; Lagogiannis, G.; Tarjan, R. E. (2012). Strict Fibonacci heaps. Proceedings of the 44th symposium on Theory of Computing - STOC '12. p. 1177
Brodal, Gerth S. (1996), "Worst-Case Efficient Priority Queues", Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 52–58).
The goals are:
- to survey the existing results in the field
- to efficiently implement selected structures in the C language
- to perform experimental evaluation and comparison of these structures on several data sets

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague March 6, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Experimental evaluation of worst-case optimal heaps

Adam Volek

Department of Theoretical Computer Science
Supervisor: RNDr. Tomáš Valla, Ph.D.

May 16, 2019

Acknowledgements

I want to thank my supervisor, RNDr. Tomáš Valla, Ph.D., for his guidance and patience during the work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 16, 2019 _____

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Volek, Adam. *Experimental evaluation of worst-case optimal heaps*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

V rámci této bakalářské práce je vytvořena implementace Brodalovy haldy, prioritní fronty, která je i v nejhorším případě asymptoticky optimální. Spolu s ní je vytvořen jednoduchý test měřící efektivitu haldy, ten je následně použit na zjištění efektivity poskytnuté implementace oproti existující implementaci Fibonacciho haldy.

Klíčová slova prioritní fronta, halda, asymptotická složitost, benchmarking

Abstract

This thesis provides an implementation of Brodal heap, a worst case efficient priority queue, along with a simple benchmark to assess its efficiency against another priority queue. The implementation is tested against an existing implementation of Fibonacci heap.

Keywords priority queue, heap, asymptotic complexity, benchmarking

Contents

List of Figures

List of Tables

Introduction

A priority queue is a data structure supporting fast insertion and fast extraction of the minimum element, along with some other operations, its implementation being a classical problem in computer science [1]. Whether it is possible to implement a priority queue with worst-case optimal asymptotic complexity was a long-standing open question, first answered by Brodal in [2]. Different implementations meeting the same asymptotic bounds were found later [3, 4].

While these implementations answered an important open question, their practical applicability is unclear as they are often very complex and their asymptotic complexity might not provide an advantage on datasets of practical sizes.

This thesis aims to provide an implementation of worst-case optimal priority queue described in [2] and perform an experimental evaluation of its performance, comparing it to an existing implementation of the Fibonacci heap, a priority queue matching Brodal heap's complexity in an amortised sense.

The theoretical section of this thesis provides a detailed description of the queue selected for this work. The practical section describes the methodology used for the evaluation of the selected queue, details about its implementation, and the results of the evaluation.

Chapter 1

Theoretical overview

A priority queue is a data structure representing a collection of (key, value) pairs. The structure supports three operations: Insert, which adds a new pair into an existing queue; FindMin, which returns a pointer to a pair with the minimum key, and DeleteMin, which modifies the queue so that it will not contain the pair with a minimum key anymore. Additionally, some priority queues support two other operations: DecreaseKey, which replaces the key of a given element with a smaller one, and Meld, which takes two queues and constructs one containing all the elements contained in the two original queues.

Binary heap [5] was the first found implementation of a priority queue that supported sublinear asymptotic complexity for Insert, DeleteMin and FindMin operations: $\mathcal{O}(\log n)$ for the first two and $\mathcal{O}(1)$ for FindMin. Since its discovery, heap structures have been extensively studied, resulting in many other implementations [1]. Fibonacci heap, described in [7], was the first to achieve constant amortised asymptotic complexity for all FindMin, Insert, Meld and DecreaseKey operations, and logarithmic for DeleteMin operation. These complexities are optimal for a comparison based priority queue, in the sense that complexity of DeleteMin operation cannot be decreased without increasing complexities of other operations (which itself follows from lower bound of $\Omega(n \log n)$[6, pp. 191–193] for comparison based sorting algorithm).

Whether a priority queue achieving the same complexities in the worst-case exists was an open question first answered by Brodal in [2]. The resulting structure is described as "quite complicated", with Brodal noting there is a need to "simplify the construction to make it applicable in practice."

This section provides a detailed description of Brodal heap, elaborating on its internal structure, its invariants, and mechanisms that are in place to make sure those invariants would not get violated. It also provides a brief description of the Fibonacci heap, along with a link for further reading, as the Fibonacci heap has been chosen for practical performance comparison with the Brodal heap.

1.1 Fibonacci heap

Fibonacci heap was the first priority queue to achieve optimal asymptotic complexity in the amortised sense. The primary motivation behind its creation was to decrease the asymptotic complexity of other algorithms that rely on a priority queue internally, most notably the Dijkstra's shortest path algorithm, which runs in $\mathcal{O}(n \log n + m)$ time when using Fibonacci heap [7].

Internally, the Fibonacci heap is somewhat similar to a Binomial heap, but its structure is somewhat more flexible. Each element is stored in a single node and apart from it, every node stores four pointers: a pointer to its parent, to some of its child, and its left and right sibling. Additionally, for each node x, we maintain the number of its children $r(x)$ and a flag signifying whether it is marked. Nodes together form heap ordered trees. Sons of a node are stored in a circular doubly linked list. The whole heap is then a collection of such trees, stored itself in a circular doubly linked list (using the roots' otherwise unused sibling pointers). As the trees are heap ordered, we know the minimum of the heap is stored in one of the roots. We maintain a pointer to that node to support FindMin in $\mathcal{O}(1)$ time. We maintain an invariant that no root is marked. Other than that, there are no explicit requirements on the heap structure.

With the pointer to the smallest element, operation FindMin is trivial. Meld simply concatenates the lists of trees and re-establishes the minimum pointer. With Meld implemented, Insert can then be performed as making a Fibonacci heap containing a single element and using Meld to merge it with the original queue.

DeleteMin operation is a bit more complicated. So far, we did not enforce any structure on the heap. DeleteMin is allowed to take amortised $\mathcal{O}(\log n)$ time, so we use that time to clean the heap up somewhat. Internally, we make use of a link operation, which takes two trees of the same rank n and makes the one with larger element a son of the other, producing a tree of rank $n + 1$. When deleting the smallest element from the heap, we start by deleting the node it is in (we know its location as we maintain a pointer to it) and concatenate the list of its sons with the list of trees in the heap. Then we perform the following operation until there are no two trees of the same rank in the heap: we find two roots of the same rank in the heap, link them, and we add the resulting tree to the list of heap's trees. After this, we search all the roots, unmarking those that were marked, finding the node with the smallest element and point the minimum pointer to it.

Finally, DecreaseKey operation works as follows: we decrease the element of the given node x and check whether its element is smaller than that of its parent. If not, or if x is root, we redirect heap's minimum pointer if necessary and finish. Otherwise, we decrease the rank of $p(x)$, cut x from its parent and add it to the list of trees. We check whether the former $p(x)$ is marked and if not and it is not root, we mark it and finish. Otherwise, we also cut $p(x)$ from

its parent and perform the marking and cutting recursively on it, potentially cutting nodes all the way to root if we do not run into a previously unmarked parent along the way.

Fibonacci heap is asymptotically optimal in amortised sense, meaning its amortised complexities are $\mathcal{O}(1)$ for all operations except DeleteMin, which has amortised complexity of $\mathcal{O}(\log n)$ [7]. The amortised analysis of the structure is beyond the scope of this thesis but is provided in [7].

1.2 Brodal heap

Brodal heap is the first implementation of a priority queue to achieve worst-case optimal asymptotic complexity. This section explains how the heap works based on its description in [2], along with an explanation of some edge cases not covered by the original paper. First, two auxiliary data structures are presented: extendible arrays and guides. The heap is described from section 1.2.3 onwards.

1.2.1 Extendible array

The extendible array is an array that can be extended by one element in worst-case $\mathcal{O}(1)$ time. Extendible arrays "can be obtained from ordinary arrays by array doubling and incremental copying" [2]. If we still have extra space in the underlying array, extension operation simply updates the length of the array that is being used up. If we do not have enough space, we allocate a new array twice the size of the original, one and with each consecutive extension, we copy a constant portion of the old array to the new one. By the time we run out of space in the new array, we would have incrementally copied the whole old array into the new one and deleted the old one, returning the extendible array to its original state of having a single array.

1.2.2 Guide

Suppose we have a sequence of integers $(a_n, a_{n-1}, \ldots, a_1)$ (we write guide related sequences right to left for convenience) for which we want to maintain a property that all elements in the sequence are smaller or equal than a given integer parameter t. On this sequence, we are forced to increase or decrease elements at a given index, and for each increase, we are allowed to perform an $\mathcal{O}(1)$ number of Reduce operations, which takes an index n and performs one of the following transformations on the sequence:

- decrease nth element by one,
- decrease nth element by at least two and increase the $(n+1)$st element by at most one.

Guide's task is to tell us after each increase at which indices to perform the Reduce operation so that all elements of the sequence are smaller or equal than t.

The following notation is used to illustrate transformations on the guided sequence:

$$0, 1, 2, 1, 1, 0^{\uparrow}$$
$$0, 1, 2^{\downarrow}, 1, 1, 1$$
$$0, 2, 0, 1, 1, 1$$

The sequence is written right to left, $a_n^{\uparrow}$ represents a forced increase on an index n and $a_n^{\downarrow}$ represents a Reduce operation on the index n.

When constructing the guide, we can w.l.o.g. assume $t = 2$. If we are required to construct a guide for a different t', we can present to the guide an alternative sequence $(a'_n, a'_{n-1}, \ldots, a'_1)$ where

$$a'_n = \begin{cases} 3, & \text{if } a_n = t' + 1 \\ 2, & \text{if } a_n = t' \\ 1, & \text{if } a_n = t' - 1 \\ 0, & \text{otherwise} \end{cases}$$

and assume $t = 2$.

Internally, the guide splits the sequence into smaller sequences $(a_x, \ldots, a_y)$ of the length of at least two. We call these subsequences blocks. The elements of the original sequence can belong to at most one block. The guide tries to maintain the following two invariants:

G1 for each block $(a_n, \ldots, a_m)$ the first element $a_m = 0$,

G2 if $a_n = 2$ then a block $(a_n, \ldots, a_m)$ exists.

Intuitively, the two invariants say the maximum values of elements in a block can be $(2, 1, \ldots, 1, 0)$, where the number of ones can be zero.

The motivation for these invariants is avoiding the worst-case scenario for a guide—a sequence with more than $\mathcal{O}(1)$ consecutive "2" elements. If we were forced to increase the first of those elements, we might be forced to perform more than $\mathcal{O}(1)$ Reduce operations to make sure no element in the sequence is greater than two as shown in this example:

$$0, 2, 2, \ldots, 2, 2, 2^{\uparrow}$$
$$0, 2, 2, \ldots, 2, 2, 3^{\downarrow}$$
$$0, 2, 2, \ldots, 2, 3^{\downarrow}, 1$$
$$0, 2, 2, \ldots, 3^{\downarrow}, 1, 1$$
$$\ldots$$
$$0, 2, 3^{\downarrow}, \ldots, 1, 1, 1$$
$$0, 3^{\downarrow}, 1, \ldots, 1, 1, 1$$
$$1, 1, 1, \ldots, 1, 1, 1$$

$$0, 1, 1, \underline{2, 1, 1, 0}, \underline{2, 0}, \underline{1, 0}, 0, \underline{2, 0, 1, 0}$$

Figure 1.1: An example of a possible valid partitioning of a guide

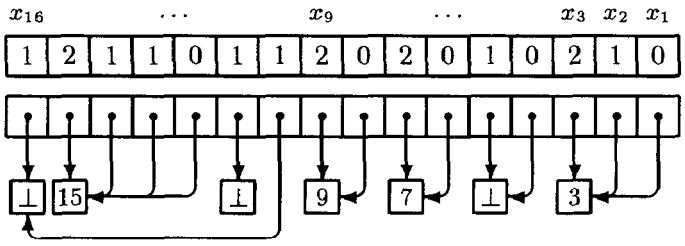

Figure 1.2: Ilustration of guide's internals [2, p. 4]

Invariants G1 and G2 guarantee such a scenario cannot arise, as each "2" must be at the tail of some block and each block must start with "0", i.e. each "2" is separated from each other at least by a "0".

The notation used to depict the state of a guided sequence can be extended by underlining to show its partitioning in blocks as shown in the figure 1.1. Blocks are usually in the form of $\underline{2, 1, \ldots, 1, 0}$, but a forced decrease of one of its elements can lead to blocks similar to those in the figure 1.1. This case does not violate guide invariants.

Internally, blocks are represented by a structure which contains the index of block's last element (usually a "2"). The guide maintains an array of pointers to the block structures the same length as the sequence, each element in the sequence thus having its pointer. Pointers of all elements in a block point to the same block structure. If an element is not in a block, its pointer points to a block structure which contains a special index $\perp$ instead of an index of some existing element in the sequence. This structure is illustrated in figure 1.2. Representing guide's blocks this way gives us two essential properties—in worst-case $\mathcal{O}(1)$ time, we can:

- find out whether an element is in a block given its index, and if yes, what is the index of the block's tail,
- destroy a block by rewriting its tail index to $\perp$.

1.2.2.1 Group destruction

Destroying a block can leave the guide in an invalid state—if the tail of the block were "2", the guide would not satisfy invariant G2. If that is the case,

however, we can re-establish the invariant by performing a single Reduce operation and re-establishing blocks, which can be done in $\mathcal{O}(1)$ time. The following table explains the resolution of all the cases that can arise during block destruction:

$$\begin{array}{c|c|c|c|c} \underline{2^{\downarrow},1,1,0} & 1,\underline{2^{\downarrow},1,1,0} & 1,\underline{2^{\downarrow},1,1,0} & \underline{\ldots,0},\underline{2^{\downarrow},1,1,0} & \underline{\ldots,0},\underline{2^{\downarrow},1,1,0} \\ 1,1,1,0 & 1,0,1,1,0 & \underline{2,0},1,1,0 & \underline{\ldots,0},0,1,1,0 & \underline{\ldots,1,0},1,1,0 \end{array}$$

Note that, given guide's internal representation of blocks, the last two cases are valid even if the trailing 0 is not a part of a block, meaning it is a part of a block with tail at index $\perp$ in the internal representation, as adding the newly created zero to such block results in it not being in any block. Using this procedure, we can destroy a block and re-establish guide's invariants in worst-case $\mathcal{O}(1)$ time using a single Reduce operation.

1.2.2.2 Handling forced increments

The main task of the guide structure is to decide where to perform Reduce operations to maintain all its invariants when it is notified about a forced increase. Upon notification, one of the three cases can occur. This section analyses each of them and explains the actions needed to make the guide valid again.

If an element has been increased and its value is now 1, we simply check whether said element is in a block and destroy it as described in section 1.2.2.1. After this, no element is greater than 2 and if the invariant G1 became violated by the increase, it has been re-established by the block destruction operation. This takes in worst-case $\mathcal{O}(1)$ time and requires a single Reduce operation.

Otherwise, if an element has been increased and its value is now 2, we first perform block destruction on its block if it was in one. This could decrease the element to 0 if it was a tail of its block in which case the invariants became re-established and we can stop. If that is not the case, the following table demonstrates how to re-establish guide's invariants in that instance:

$$\begin{array}{c|c|c|c|c} 2^{\downarrow} & 1,\underline{2^{\downarrow}} & 1,\underline{2^{\downarrow}} & \underline{\ldots,0},\underline{2^{\downarrow}} & \underline{\ldots,0},\underline{2^{\downarrow}} \\ 1 & 1,0 & \underline{2,0} & \underline{\ldots,0},0 & \underline{\ldots,1,0} \end{array}$$

The same remark about the possible trailing zero's block from section 1.2.2.1 holds here, i.e. the case would still yield a valid result even if the element was not in a block before the Reduce operation. As the group destruction operation takes at worst $\mathcal{O}(1)$ time and uses a single Reduce operation, we know in this case we need at worst $\mathcal{O}(1)$ time and two Reduce operations to re-establish guide's invariants.

The last case that can occur is when an element has been increased, and its value is now 3. Such an element must have had value 2 before the increment, and because of invariant G2, we know it is a tail of a block. First, we perform a Reduce on the element. If the operation decreased the element to 2, we

could end as the previously violated invariant G2 has become re-established. Otherwise, we proceed depending on the value of the next element, as shown here:

$$\begin{array}{c|c|c|c|c} 0, \underline{3^{\downarrow}, \ldots} & \underline{\ldots, 0}, \underline{3^{\downarrow}, \ldots} & \underline{\ldots, 0}, \underline{3^{\downarrow}, \ldots} & \ldots, 1, \underline{3^{\downarrow}, \ldots} & \ldots, 1, \underline{3^{\downarrow}, \ldots} \\ x^1, 1, \ldots & \underline{\ldots, 0}, x^1, \ldots & \underline{\ldots, 1^{\uparrow}}, x^1, \ldots & \ldots, 1, x^1, \ldots & \ldots, 2^{\uparrow}, x^1, \ldots \end{array}$$

In the first, second and fourth case, the invariant G2 has become re-established, and we did not violate the invariant, G1 which means we can finish. That is not true for the third and the fifth case, where the invariant G1 and G2 respectively became violated. We can think of the situation as another forced increase though, and invoke the current procedure recursively for these cases. They will be handled by the first and second case of the procedure so no recursive cycle can occur, and since we know those cases take at worst $\mathcal{O}(1)$ time and use at most two Reduce operations, resolution of this case will take at worst $\mathcal{O}(1)$ time and will use at most three Reduce operations.

1.2.2.3 Implementation remarks

The structure used for representing blocks can be extended by adding a reference counter to it. When we allocate a new one for a new block, we set the counter to zero, increasing it every time we add an element to the block it represents and decreasing it every time we remove one. Once we remove the last element from a block, we can free up the space the structure was taking. This way we, know there will be at most $\mathcal{O}(n)$ of them at any given time during the guide's operation.

Note that throughout the case analysis, the guide is shown to perform Reduce operations only on elements of the sequence whose value is 2 or 3. This property will be important later on when constructing the heap.

By making guide's internal extendible, we can support extending guide's domain in worst-case $\mathcal{O}(1)$ time. If the last element in the newly extended guided sequence is 0 or 1, we just allocate a new block structure with tail set to $\perp$, i.e. we make sure the new element is not in a block. We require this always to be the case when extending the guide.

Similarly, we can also shrink a domain of the guide. If the last element is the tail of a group, we destroy that group and remove the last element from it. After that, if the underlying array supports it, we can shrink it to free up resources.

[1]Can be either 0 or 1 depending on the outcome of the Reduce operation but since the element is not in a block, neither G1 nor G2 can become violated by the outcome, and we do not have to care about it.

1.2.3 Heap structure

The heap is represented by two rooted trees T_1 and T_2. Each element in the heap is stored in one node belonging one of the two trees. Nodes are assigned positive integer ranks. In this thesis, the same notation as in [2] is used:

x	either node x or the element stored in the node, depending on the context
t_i	root of the tree T_i
$p(x)$	parent of node x
$r(x)$	rank of x
$n_i(x)$	number of sons of rank i that x has

The heap structure is governed by the node's ranks. The way ranks are assigned, and the way this assignment forces a structure on the two trees is formalised by the first five heap invariants:

S1	x is a leaf $\implies r(x) = 0$
S2	$r(x) < r(p(x))$
S3	$r(x) > 0 \implies n_{r(x)-1} \geq 2$
S4	$n_i(x) \in \{0, 2, 3, \ldots, 7\}$
S5	$T_2 = \emptyset \vee r(t_1) \leq r(t_2)$

Invariants S1 and S2 say leaves are assigned a rank of zero and they increase towards the root. Invariant S3 says that except for leaves, each node x of a rank $r(x)$ has at least two sons of a rank $r(x) - 1$. Invariant S4 limits the number of sons of a given rank of x to 7. The reason this invariant does not allow a node to have a single son of a given rank is that it allows us to cut the sons of highest rank from a node and assign a new rank to the node so that S1–S4 are still satisfied. Invariant S5 then simply states that either the rank of t_2 is greater or equal to the rank of t_1, or T_2 is an empty tree. The motivation for these invariants is to make a size of a subtree rooted at a node linked to its rank and, as a side effect, to its height. This is formalised by lemmata 1.1 and 1.2.

Lemma 1.1. *The size of the subtree rooted in a node x is at least $2^{r(x)+1} - 1$.*

Proof. Let $l(n)$ be the lower bound on the size of the subtree rooted at a node x of a the rank n. From S3 it follows the smallest number of sons a non-leaf node x can have is two, each of the rank $r(x) - 1$. We can, therefore, recursively define $l(n)$ as follows:

$$l(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + 2l(n-1), & \text{otherwise} \end{cases}$$

For any integer $n \geq 0$, let $S(n)$ denote the statement

$$S(n) : l(n) = 2^{n+1} - 1$$

We can prove $S(n)$ for any positive integer n by mathematical induction:
Base step($n = 0$): $S(0)$ says $l(0) = 2^1 - 1$ which is true by the definition of $l(n)$.
Inductive step $S(k) \implies S(k+1)$: Fix some $k \geq 0$ and assume that $S(k)$ holds:

$$\begin{aligned} l(k) &= 2^{k+1} - 1 && \text{(By } S(k)\text{, the ind. hyp.)} \\ 1 + 2l(k) &= 2^{k+2} - 1 \\ l(k+1) &= 2^{k+2} - 1 && \text{(By definition of } l(n)\text{)} \end{aligned}$$

Which proves the induction hypothesis. This proves that $l(n) = 2^{n+1} - 1$, and since we defined $l(n)$ to be the lower bound on the size of a subtree rooted at a node of rank n, this also proves the lemma. □

Lemma 1.2. *The size of the subtree rooted in a node x is at most $2^{3r(n)}$.*

Proof. Let $u(n)$ be the upper bound on the size of the subtree rooted at the node x of rank n. From S4 it follows a non-leaf node x can have at most seven sons of each rank smaller than $n = r(x)$. We can, therefore, recursively define $u(n)$ as follows:

$$u(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{i=0}^{n-1} 7u(i), & \text{otherwise} \end{cases}$$

For any integer $n \geq 0$, let $S(n)$ denote the statement

$$S(n) : u(n) = 2^{3n}$$

We can prove $S(n)$ for any positive integer n by mathematical induction:
Base step($n = 0$): $S(0)$ says $u(0) = 2^0$, which is true by the definition of $u(n)$.
Inductive step $S(k) \implies S(k+1)$: Fix some $k \geq 0$ and assume that $S(k)$ holds:

$$\begin{aligned} u(k) &= 2^{3k} && \text{(By } S(k)\text{, the ind. hyp.)} \\ u(k) + 7u(k) &= 2^3 \cdot 2^{3k} \\ 1 + \sum_{i=0}^{k-1} (7u(i)) + 7u(k) &= 2^{3(k+1)} && \text{(By definition of } u(n)\text{)} \\ 1 + \sum_{i=0}^{k} (7u(i)) &= 2^{3(k+1)} \\ u(k+1) &= 2^{3(k+1)} && \text{(By definition of } u(n)\text{)} \end{aligned}$$

Which proves the induction hypothesis. This proves that $u(n) = 2^{3n}$, and since we defined $u(n)$ to be the upper bound on the size of a subtree rooted at a node of rank n, this also proves the lemma. □

Corollary 1.2.1. *Proofs of lemmata 1.1 and 1.2 showed that the size of a subtree rooted at a node x is exponential in $r(x)$. Inversely, this shows that the rank of a node x is logarithmic in the size of the subtree rooted at x. From S1–S3 it follows that the height of a subtree rooted at a node x is $r(x)$, therefore the height of such subtree is also logarithmic in its size.*

1.2.4 Heap order

Comparison based heap structures usually satisfy a heap order invariant, requiring element in each node except for root(s) to be greater or equal to the node's parent's element. That is not the case for this structure, as violations of heap order are allowed in specific conditions. We call a node x violating iff $x < p(x)$, and we call a node good iff it is not violating. To track the violating nodes, we extend the tree node structure by sequences W and V that store references to nodes y that are good in regards to x (i.e. $x \leq y$) but might be violating nodes.

For the sake of clarity, we extend the notation introduced in section 1.2.3 as follows:

$V(x)$	V sequence of node x
$W(x)$	W sequence of node x
$w_i(x)$	number of nodes of rank i in $W(x)$

To formalise the role of W and V sequences and to ensure we use them effectively to keep track of heap order violation, we introduce invariants O1–O5:

O1	$t_1 = min(T_1 \cup T_2)$
O2	$y \in V(x) \cup W(x) \implies y \geq x$
O3	$y < p(y) \implies (\exists x \neq y)(y \in V(x) \cup W(x))$
O4	$w_i(x) \leq 6$
O5	if $V(x) = (y_{\lvert V(x)\rvert}, \ldots, y_2, y_1)$, then $r(y_i) \geq \lfloor (i-1)/\alpha \rfloor$ for $i = 1, 2, \ldots, \lvert V(x)\rvert$ where α is a constant

Invariant O1 requires the minimum element in the heap to be stored at T_1. Without this property, we might not be able to implement FindMin operation in worst-case $\mathcal{O}(1)$ time as a minimum element could be either t_1, t_2, or it could be any violating node in the heap. Invariant O2 says that a node has to be good in respect to the node whose violation sequence it belongs in. Invariant O3 requires all violating nodes to belong to some violation sequence, and finally, invariants O4 and O5 bound the lengths of violation sequences (proven by lemmata 1.3 and 1.4). The reason invariants O4 and O5 are stated so differently is because the sequences W and V play a very different role in the construction of the heap. Later we show we will add newly created violations only to $W(t_1)$ in cases of violations of a rank smaller or equal to $r(t_1)$ and $V(t_1)$ in cases of violations of larger rank.

Lemma 1.3. *The length of any W sequence is at most logarithmic in the number of elements contained in the heap.*

Proof. Let x denote the node of maximum rank in the heap, i.e. for all nodes $y \neq x$ in the heap we know $r(y) \leq r(x)$. Because of invariant S2, we know this is one of t_1 and t_2. Invariant O4 states there cannot be more than six nodes of a given rank in a violation sequence. If there is no node of rank greater than $r(x)$, the sequence cannot contain more than $6r(x)$ nodes. Corollary 1.2.1 shows that the rank of a node is logarithmic in the size of the subtree rooted at it. We, therefore, know the length of any W sequence is at most logarithmic in the size of the subtree rooted at x, and since this subtree cannot be bigger than the entire heap, we also know the length of any W sequence can be at worst logarithmic in the size of the entire heap. □

Lemma 1.4. *The length of any V sequence is at most logarithmic in the number of elements contained in the heap.*

Proof. Invariant O5 states that for each rank r, any V sequence can contain at most α nodes of rank equal or larger to r. We can make a similar argument as in the proof of lemma 1.3: given the node of maximum rank x, there cannot be more than $\alpha r(x)$ elements in any V sequence, and since α is constant and $r(x)$ is at most logarithmic in the size of the heap, we know the length of any V sequence is at most logarithmic in the size of the heap. □

The role of α is to represent a maximum number of new large violations that can be created during a single heap operation. The only important thing about it is that it is a constant as will be shown later, its exact value is needed neither for the analysis nor for the construction of the structure.

1.2.5 Tree roots

For nodes t_1 and t_2, we strengthen the invariants as follows:

$$\begin{array}{ll} \text{R1} & n_i(t_j) \in \{2,3,\ldots,7\} \text{ for } i = 0,1,\ldots,r(t_j)-1 \\ \text{R2} & |V(t_1)| \leq \alpha r(t_1) \\ \text{R3} & y \in W(t_1) \implies r(y) < r(t_1) \end{array}$$

Invariant R1 is a stronger version of invariant S4, requiring roots t_j to have at least two sons of each rank smaller than $r(t_j)$. Invariant R2 strengthens invariant O5 for the node t_1, requiring the length of $V(t_1)$ to be at most $\alpha r(t_1)$ instead of $\alpha r(x)$ where x is the node of maximum rank. Finally, invariant R3 requires all nodes in $W(t_1)$ to have rank smaller than that of t_1.

1.2.6 Implementation details

Each node in the tree can has arbitrarily many sons. The list of sons is implemented by a doubly linked list, which requires each node to store three pointers: one to the beginning of the list of its sons and two to the previous and the next son in the list of sons a node belongs to. Additionally, nodes store a pointer to their parent. W and V sequences are implemented in the same way: node stores two pointers that represent the beginning of its W and V sequence, and two other pointers to the previous and the next node in the violation sequence it belongs to. More pointers are not needed as a node cannot belong to more than one violation sequence. The leftmost node in a given violation sequence $W(x)$ or $V(x)$ has a pointer to the previous node in the sequence pointed to x.

The only violation sequences we add new violations to are $W(t_1)$ and $V(t_1)$. We require that the nodes of the same rank in $W(t_1)$ are adjacent. To support adding a new node to the sequence in $\mathcal{O}(1)$ time, we maintain an extendible array of pointers to nodes of a given rank in $W(t_1)$. We may w.l.o.g. maintain the property that the pointer will point to the rightmost node of a given rank. This can simplify the implementation of some heap operations. We also store a single guide that helps us maintaining invariant O4 for $W(t_1)$, as described in section 1.2.7.4.

We require the sons of all nodes to be stored in decreasing order by their rank, i.e. the leftmost sons of the node are the sons of the highest rank. To support adding and cutting sons of roots, we maintain two extendible arrays that point to a son of a given rank of t_1 and t_2 respectively. As with the $W(t_1)$ array, we may w.l.o.g maintain the property that the pointer will point to the rightmost node of a given rank. This also simplifies the implementation of some heap operations. We also store four additional guides, to maintain the upper and lower bound of invariant R1 for both roots as described in section 1.2.7.2.

1.2.7 Low-level operations

This section describes low-level operations that are used to maintain heap's invariants during higher level operations. As Brodal said in [2], "the maintenance of R1 and O4 turns out to be nontrivial but they can all be maintained by applying the same idea"—the idea behind the guide structure. Applying the idea to help to maintain the two invariants is described in section 1.2.7.2 and 1.2.7.4. All operations in this section run in worst-case $\mathcal{O}(1)$ time.

1.2.7.1 Tree linking and delinking

Let x, y and z be three nodes of the same rank n. Assume w.l.o.g that $x \leq y$ and $x \leq z$ and make y and z the leftmost sons of x. By increasing $r(x)$ by one,

structural invariants will have become re-established for x. This operation is called tree linking, and it results in a single tree of rank $n + 1$.

Delinking can be thought of as an inverse operation to linking, although it is slightly more complicated. Let x be a node of non-zero rank. If it has more than three sons of rank $r(x) - 1$, we can cut its two leftmost sons and return those to delink it. The rank of x will not change in that case. If it has exactly two or three sons of rank $r(x) - 1$, we cut those and assign the node plus one of the rank of its leftmost son. This operation takes one node of rank n and produces two or three nodes of rank $n - 1$ and at most one node of rank smaller or equal to n.

1.2.7.2 Maintaining invariant R1

During other heap operations, we might want to get a son of a given rank from under one of the root, or to push an existing son there. For each root, we have two guides that help us with maintaining invariant R1 in such cases, one for guiding the upper bound, and the other for lower bound. Note that is the reason for such arbitrarily looking bounds in invariant S4 and O1. Since we will only link sons of rank r if $n_r(t_i) \geq 7$ and delink son of rank $r + 1$ if $n_r(t_i) \leq 2$, and since those operations produce or require at most three sons, $\{2, 3, \ldots, 7\}$ is the smallest interval we could have chosen so that the guides guiding lower and upper bound of $n_i(t_j)$ would not interfere with each other. Also note that guiding lower bound on a sequence is the same as guiding upper bound on an inverted sequence, no edits to the guide are necessary.

If we want to add a subtree below the root, we notify the guide taking care of the upper bound that there has been a forced increase in its sequence. Guide then tells us for what ranks to perform Reduce operation, which, in this case, maps to the link operation from section 1.2.7.1. No new violations are created during this operation.

If we want to take a subtree from a root, we notify the guide taking care of the lower bound. Guide then tells us where to perform delink operations to re-establish invariant R1. Each delinking can create up to three new violations if we delink sons of t_2, which we handle as described in section 1.2.7.4. The additional node left after delinking we can just add under the root as described before.

Finally, we have to consider increasing and decreasing of root's rank. If we were to add subtrees to t_i indefinitely, eventually the guide would want us to link three nodes of rank $r(t_i) - 1$, and in such a case we might have no way of re-establishing the invariant R1 in constant time. The same situation can happen with delinking, where guide might want us to delink a son of rank $r(t_i)$ when t_i can have no such son. To solve this situation, we change the scheme to let guides control only sequence $(n_1(t_i), n_2(t_i), \ldots, n_{r(t_i)-3}(t_i))$, and we will maintain invariant R1 for $n_{r(t_i)-2}(t_i)$ and $n_{r(t_i)-1}(t_i)$ separately.

Let's first consider the situation of adding a subtree to the unguided section. If we add a subtree there and the root still satisfies invariant R1, we can quit. If it doesn't, we can w.l.o.g. assume that the addition happened at $n_{r(t_i)-1}(t_i)$ (if it did not, we just link at rank $r(t_i) - 2$ and consider the situation as an addition on $n_{r(t_i)-1}(t_i)$). Since we are in a situation where we might increase the rank of t_i and hence the domain of its guides, we need to make sure both guides can be extended, i.e. we need to make sure $2 < n_{r(t_i)-2}(t_i) < 7$. If that is not the case, we can perform a single link or delink operation to change that. Delink might additionally cause $n_{r(t_i)-1}(t_i) \leq 7$ in which case R1 became re-established, and we might end. If that is not the case, we will increase the rank of t_j, extend guides and arrays related to them and finally re-establish the R1 by two link operations.

The situation is a little more complicated when we want to take the subtree from the unguided section. If we take a subtree and the root still satisfies invariant R1, we can quit. If it does not, and we took a tree of rank $r(t_i) - 2$, we can perform a single delink operation and regard the situation as if we took a root of rank $r(t_i) - 1$. From here, we either want to make sure $n_{r(t_i)-1}(t_i) = 0$ so we can decrease the rank of the root and end, or make sure we have re-established R1 without decreasing the root's rank. At this point, we know the following holds:

$$n_{r(t_i)-1}(t_i) = 1$$

$$2 \leq n_{r(t_i)-2}(t_i) \leq 7$$

If $n_{r(t_i)-2}(t_i) \geq 5$ we can solve the situation with a single link and end. Otherwise we perform another delink on rank $r(t_i) - 1$. If after this $n_{r(t_i)-1}(t_i) = 0$ we can decrease the rank of t_1 and end. Otherwise, we know the following holds:

$$n_{r(t_i)-1}(t_i) = 1$$

$$4 \leq n_{r(t_i)-2}(t_i) \leq 7$$

Notice how the lower bound of $n_{r(t_i)-2}(t_i)$ got increased by two by this sequence of operations. We can perform it again, which will either solve the problem or increase the lower bound by two again, leaving us in this state:

$$n_{r(t_i)-1}(t_i) = 1$$

$$6 \leq n_{r(t_i)-2}(t_i) \leq 7$$

If that happens to be the case, we can now solve the situation with a single link operation, after which $n_{r(t_i)-1}(t_i) = 2$ and $3 \leq n_{r(t_i)-2}(t_i) \leq 4$ which means R1 got re-established and we can end.

1.2.7.3 Reducing the number of violations

This operation takes two potential violations (nodes in some violating set) of a same rank $n < r(t_1)$ which are not roots or sons of roots and either makes one of them good, or makes both of them good and produces at most one violation of rank $n + 1$, thereby decreasing the total number of potential violations in a heap by one in constant time.

We start by checking that both of the nodes (let's call them x and y) are still violations. If not, we remove the one that is not from its violation sequence and end. Otherwise, we continue as follows. Because of O4, we know both x and y has at least one sibling of the same rank. If the two nodes are not siblings already, we can w.l.o.g. assume $p(x) \leq p(y)$ and swap the subtrees rooted at x and the sibling of y. This cannot create a new violation, does not violate any invariant and makes x and y siblings.

Since t_1 is the smallest element in a heap, we can make any node good by adding it under t_1 as described in section 1.2.7.2. This operation tries to do precisely that, making sure no invariant gets violated along the way.

If x and y have another sibling of the same rank, we can simply cut x off, add it under t_1, extract it from its violation set and end. This does not violate O4 since there will still be two sons of the same rank. If x and y are the only siblings of the same rank and they are not the sons of the highest rank of their parent, we can cut off both of them, add them under t_1 and end. This, again, does not violate O4. The only case remaining now is when x and y are the sons of the highest rank of $p(x)$, in which case cutting them off affects the rank of $p(x)$ because of O3. To make sure this does not affect the node $p(p(x))$, we cut all of the nodes x, y and $p(x)$ from their parent and replace $p(x)$ by a node of the same rank that we can obtain from t_1 as described in section 1.2.7.2. If the replacement for $p(x)$ is now a violating node we simply add it to $W(t_1)$, make all x, y and $p(x)$ good nodes of t_1 and end.

1.2.7.4 Maintaining invariants O4 and R2

When we create a new violation x, we add it to $W(t_1)$, if $r(x) \leq r(t_1)$ or $V(t_1)$ otherwise. This might cause a violation of invariant O4 or R2.

To prevent violating R2, with each heap operation, we move a constant number of sons of t_2 to T_1, increasing its rank. This allows us to create α new big violations with each operation without violating R2. We can only do this when T_2 is not empty, but in cases where it is, t_1 is guaranteed to be the node of maximum rank, no new big violations cannot be created, and R2 cannot become violated in the first place.

When adding a new violation of rank n to $W(t_1)$, we notify its guide that $w_n(t_1)$ has been forced to increase. The guide then tells us at which ranks to perform the violation reducing transformations to maintain O4. We actually only do the transformation in cases where there are at least two violations of

that rank that are not sons of t_2. If there are more than four nodes of the same rank in $W(t_1)$ that are sons of t_2, we just cut one, link it under t_1 and remove it from the violation set. This will not affect guides guiding R1 on t_2.

1.2.8 High-level operations

With the lower level operations built, we can finally proceed to high-level operations. Brodal heap supports the following six operations: MakeQueue, FindMin, Insert, Meld, DecreaseKey and DeleteMin. First five are supported in worst-case $\mathcal{O}(1)$, while the last one takes worst-case $\mathcal{O}(\log n)$.

MakeQueue is trivial, returning a structure where both T_1 and T_2 are empty. FindMin is simple because of invariant O1; we just return the element stored in t_1 assuming the heap is not empty. Furthermore, Insert is defined in terms of Meld. We define the heap with a single element as a heap with one node as a root of T_1 storing that element. Insert then simply creates such a queue containing the element to be inserted and then uses Meld to insert the element in the original heap. Other three operations are explained in separate subsections:

1.2.8.1 Meld

Meld handles a maximum of four trees, two from each heap. First, the tree with the smallest root is made the new T_1. If this tree also happens to be the tree of maximum rank, we add all other trees under t_1 as described in section 1.2.7.2. If this is not the case, we make the tree of maximum rank T_2 and link the other trees under t_2. Possibly we have to delink them if some other trees have the same rank, but it is guaranteed that no more than three delinks are necessary for each tree to lower its rank by one.

The last case we have to consider is when all trees are of rank 0, where we could not use the procedure above as we cannot delink a tree of rank 0. If Meld handles two trees of rank 0, it simply makes the one with smaller root T_1 and the other T_2. If it handles more of them, it makes the one with smallest root T_1, increases its rank to 1 and adds the other two or three nodes as its sons. We could not have done this in case of two rank zero trees because we would violate invariant S4.

Finally, we drop the arrays and guides related to trees that are not roots anymore after the Meld operation.

1.2.8.2 DecreaseKey

With the violation handling mechanisms from section 1.2.7.4 implemented, the main invariant we have to worry about is O1 because we cannot allow any element to be smaller than t_1. If this is the case after a DecreaseKey operation on a node x, we swap the element stored in x and t_1. After that, if the node is a violation, we handle the situation as described in section 1.2.7.4.

1.2.8.3 DeleteMin

This is the only operation that is allowed to take worst-case $\mathcal{O}(\log n)$ time. First we make T_2 empty by incrementally increasing the rank of t_1. The new minimum (we name the node m) is found by searching the sons of t_1 (among which is now the original t_2) and its violation sequences. From the proofs of lemmata 1.1, 1.2, 1.3 and 1.4, we know this is at the most $\mathcal{O}(\log n)$ nodes to search. If m is a violation, we swap it with a son of t_1 of the same rank. This might create a violation which for now we just call v we remember to solve it later. Finally, we remove m from its violation set as it is not a violating node anymore.

The node m is now a son of t_1. We cut it from its parent and use the procedure described in section 1.2.7.2 to re-establish invariant R1. We then cut all sons of m (there is at most $\mathcal{O}(\log n)$ of them) from their parent and add them below t_1, again using the procedure from 1.2.7.2 to maintain R1 for t_1.

We now take a sequences $W(t_1)$, $V(t_1)$, $W(m)$ and $V(m)$, concatenate them, adding the node v to one of them if v is a violation, and use a linear time sorting algorithm such as pigeonhole sort [8] to sort them by their rank. We know this sequence can be at most $\mathcal{O}(\log n)$ long and the highest ranking node in the heap has a rank of at most $\mathcal{O}(\log n)$, we know this will take at worst $\mathcal{O}(\log n)$ time. We now make this sequence into the new $W(t_1)$, emptying $V(t_1)$, $W(m)$ and $V(m)$, and use the transformation described in section 1.2.7.3 at most $\mathcal{O}(\log n)$ times to make sure $w_i(t_1) \leq 2$ for all $i \in 0, \ldots, r(t_1) - 1$. Note that we do not have to be strict about the two here, if the implementation of the transformation from section 1.2.7.3 requires there to be at least three nodes to function, we can increase the limit. As long as we will not increase it above five, we do not violate O4, and we will not make the guide guiding $W(t_1)$ invalid. This re-establishes invariant O4 for t_1 if it has been invalidated, and does not affect the validity of guide guiding $w_i(t_1)$. It also re-establishes invariant O2 and O3 for v if it became a violation after the swap with m at the beginning.

At this point, all heap invariants have been re-established. Node m is neither in T_1 nor in T_2, it has no sons, and its violation sequences are empty. It has no additional information useful for the rest of the heap and can safely be deleted, together with the former smallest element it now holds. The root of T_1 now holds the next smallest element, and all heap invariants are satisfied.

This procedure assumes that we can always cut a son of t_1 and re-establish invariant R1 for it using the procedure described in section 1.2.7.2. There is one important edge case where this is not true—when the heap contains exactly three elements (we call them a, b and c and assume w.l.o.g. that $a \leq b \leq c$). In this case, $r(t_1) = 1$, t_1 has exactly two sons, and there is no way of cutting one of them without violating R1. This can be solved easily though, as we can just delete a and make b and c rank zero roots of T_1 and T_2 respectively.

This resolves the edge case as it results in a valid heap, and does not affect the asymptotic complexity of the DeleteMin operation.

Chapter 2

Implementation and performance evaluation

2.1 Brodal heap

As a part of this thesis, an implementation of the structure described in section 1.2 in the C language is provided. The implementation has the following API:

```
typedef int (*bh_compare_pt)(void *, void *);

void bh_initialize(bh_heap_t *heap, bh_compare_pt compare);
void *bh_find_min(bh_heap_t *heap);
bh_tree_node_t *bh_insert(bh_heap_t *heap, void *elem);
int bh_meld(bh_heap_t *heap, bh_heap_t *other);
bh_tree_node_t *bh_decrease_key(bh_heap_t *heap,
    bh_tree_node_t *node, void *elem);

bh_tree_node_t *bh_delete_min(bh_heap_t *heap);
void bh_delete(bh_heap_t *heap);
```

The first six functions implement respective high-level heap operations from section 1.2.8, while `bh_delete` is a helper function that frees up all resources allocated by heap during initialisation and operation. The heap is represented by `bh_heap_t` and `bh_tree_node_t` represents a single node in the heap. The implementation works for any comparable object as it takes `void *` as the elements it stores and the only assumption it makes about the elements is that they are comparable using the compare function provided to `bh_initialize`. The return value of `bh_decrease_key` represents the node in which the new `elem` is stored. It can be different from `node` as a consequence of the way the operation works (more details on this are in section 1.2.8.2).

The return value of `bh_delete_min` represents the address of a node that got freed up during the deletion and is no longer a part of the heap.

2.2 Performance evaluation

To assess a priority queue performance, I built a test suite that performs operations on the structure in a pseudorandomized fashion, measuring the duration of each operation and logging the result. Its core function is `make_heap`, which takes a number of operations n, initialises an empty heap and an array storing references to nodes for DecreaseKey, and performs one of the following n times:

- Generate a pseudorandom number and insert it to the heap. Take the node reference returned by the operation, store it to the helper array and mark it valid.
- Choose a pseudorandom reference from the helper array and check its validity. If it is not valid, choose another pseudorandom reference. Repeat this until a valid reference is chosen. If a valid reference has not been chosen after a constant number of attempts, do nothing. Otherwise, generate a pseudorandom number and, if it is smaller than the element of the chosen node, perform the DecreaseKey operation on the chosen node.
- If the heap is not empty, perform the DeleteMin operation on it. Find the returned reference in the helper array and mark it invalid there.

Which operation is chosen depends on the output of a pseudorandom number generator. In my benchmarks, the first operation had 60% chance of happening, the second 30% and the last 10%. This ensures the heap will always grow in size over time.

When the benchmark starts, first, a heap is created using the `make_heap` function with parameter 10000. This will be the main heap for the duration of the benchmark. Then the following operations are performed sequentially in a loop:

- The DeleteMin operation is performed on the main heap.
- A new small heap is created using the `make_heap` function with parameter 1000 and is merged into the main heap using the Meld operation.
- The DeleteMin operation is performed on the main heap again.

The loop is repeated 100000 times to get as diverse measurements as possible. The reason the DeleteMin operation is performed in the main loop as well as

Table 2.1: Summary of the benchmark results

	Brodal heap		Fibonacci heap	
	mean	sd	mean	sd
Insert	5.9×10^{-7} s	3.49×10^{-7}	4.93×10^{-7} s	3.03×10^{-7}
Meld	7.81×10^{-7} s	4.37×10^{-7}	4.43×10^{-7} s	2.57×10^{-7}
DecreaseKey	1.06×10^{-6} s	5.16×10^{-7}	4.39×10^{-7} s	2.4×10^{-7}
DeleteMin	6.49×10^{-6} s	2.66×10^{-6}	3.93×10^{-6} s	1.83×10^{-6}

in `make_heap` function is because this is the only operation that is asymptotically slower than $\mathcal{O}(\log n)$ and it is important to test its running time on bigger heaps as well. Finally, the DeleteMin operation is called on the main heap until it is empty. This tests how the heaps behave when consecutive DeleteMin operations are performed on the heap, which is the case in some algorithms using heaps such as heapsort [5]. The runtime of the consecutive deletions at the end is logged separately from the runtimes of other DeleteMin operations.

2.3 Evaluation results

I choose the Fibonacci heap [7] for a comparison with the Brodal heap implementation I provided. The reason for this is that Fibonacci heap supports the same set of operations as Brodal heap and their amortised asymptotic complexity matches the worst-case asymptotic complexity of the Brodal heap. The main difference between the two is the Brodal heap being vastly more complex, with many complicated internal bookkeeping mechanisms allowing for its asymptotic optimality, which according to [2] makes the structure unlikely to be applicable in practice. I choose the implementation of a Fibonacci heap by [9].

I ran the benchmark described in section 2.2 on a PC with Intel Core i7-3820QM CPU. The summary of its results is provided in table 2.1. We see that on average, Brodal heap is slower during every operation: by 20% in case of Insert, 65% in case of DeleteMin, 76% in case of Meld, and 141% in case of DecreaseKey operation. Since the DeleteMin operation does not run in a constant time given heap size, it is also interesting to see the relation between its runtime and heap size. Figures 2.1 to 2.4 help to illustrate that. Figures 2.1 and 2.2 were created with data only from DeleteMin operations that happen in the main loop, whereas figures 2.3 and 2.4 show data from the last part of the benchmark where the heap gets deleted by consecutive DeleteMin operations, showing how the heap would behave if it were used to implement the heap sort algorithm.

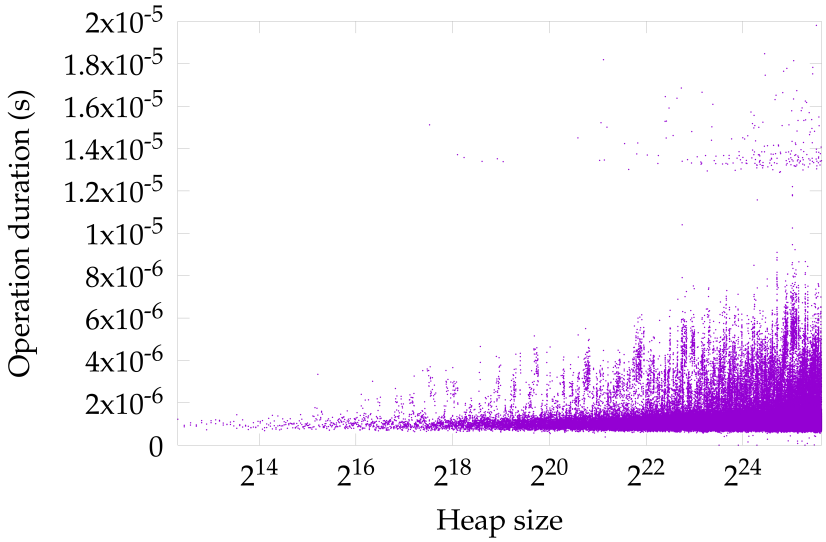

Figure 2.1: DeleteMin runtimes on Fibonacci heap given its size

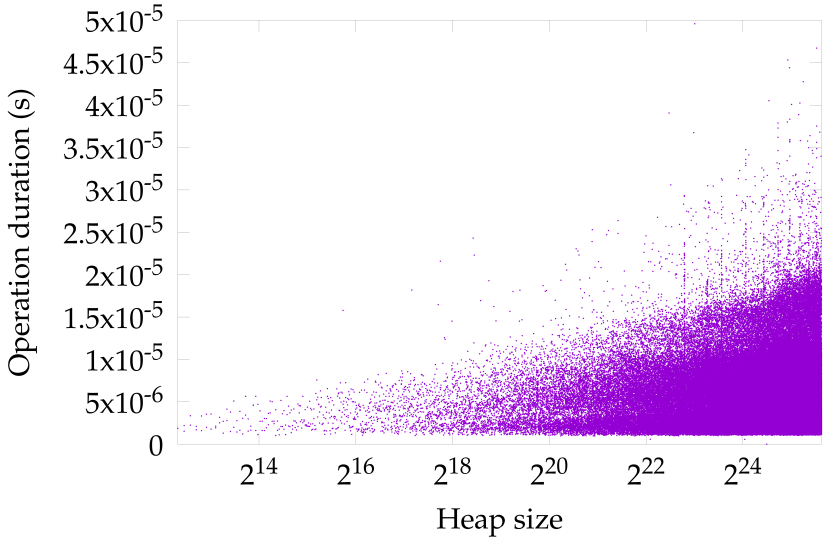

Figure 2.2: DeleteMin runtimes on Brodal heap given its size

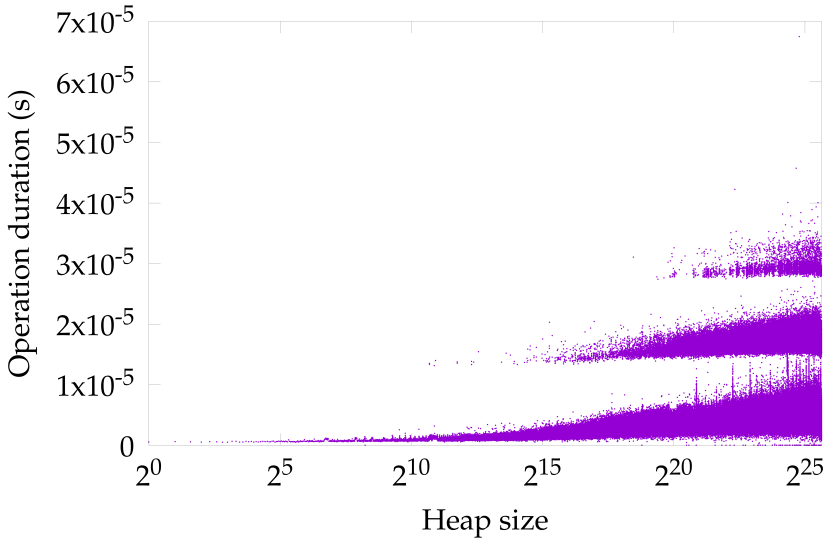

Figure 2.3: Consecutive DeleteMin runtimes on Fibonacci heap given its size

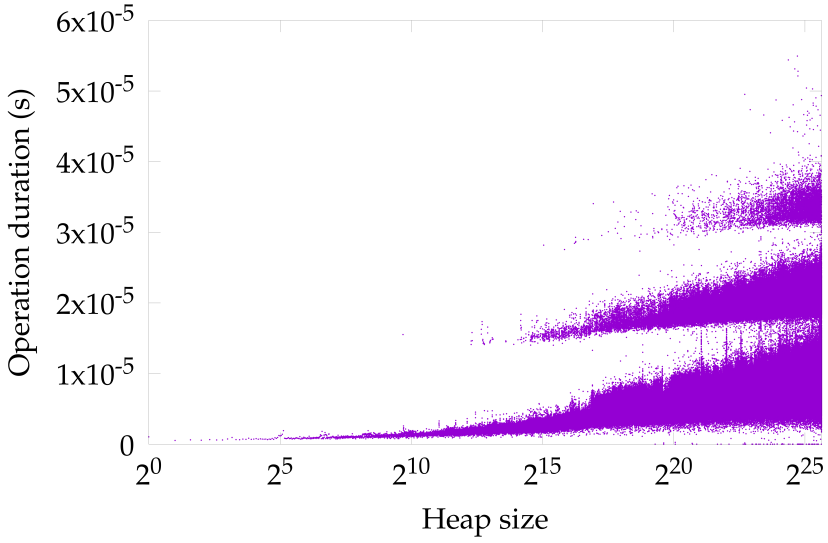

Figure 2.4: Consecutive DeleteMin runtimes on Brodal heap given its size

Conclusion

In my thesis, I provided an implementation of Brodal heap, the first priority queue achieving constant worst-case asymptotic complexity for all FindMin, Insert, Meld and DecreaseKey operations, and worst case logarithmic asymptotic complexity for DeleteMin operation. I explained in detail how the structure works and how it achieves its optimal complexity, following through on some aspects of the structure that were left unexplained in [2], such as the details of guide operation, proofs of lemmata 1.1, 1.2, 1.3 and 1.4, or edge cases concerning operations on the unguided portion of root's sons in section 1.2.7.2.

I designed a simple benchmark and used it to compare my implementation of Brodal heap to an existing implementation of the Fibonacci heap by [9]. Although I did find that the Brodal heap was the less efficient than the Fibonacci heap, the relative disparity between the two was smaller than initially expected, especially given the disparity in the structure complexity, with the Brodal heap implementation spanning over two thousand lines of C. Performance of the Insert operation on the Brodal heap was the most surprising as it was only 20% slower than its counterpart from Fibonacci heap.

An interesting problem for future work is to assess the performance of newer worst-case efficient priority queues. Since Brodal's paper in 1996, two new priority queues matching Brodal heap's asymptotic complexities were discovered [3, 4]. With each being more simple than the original Brodal heap, it would be interesting to see how would they fare against it in a practical benchmark.

Bibliography

[1] Brodal, G.S., 2013. A survey on priority queues. In *Space-Efficient Data Structures, Streams, and Algorithms* (pp. 150–163). Springer, Berlin, Heidelberg.

[2] Brodal, G.S., 1996, January. Worst-Case Efficient Priority Queues. In *SODA* (Vol. 96, pp. 52–58).

[3] Brodal, G.S., Lagogiannis, G. and Tarjan, R.E., 2012, May. Strict fibonacci heaps. In *STOC* (pp. 1177–1184).

[4] Elmasry, A. and Katajainen, J., 2012, July. Worst-case optimal priority queues via extended regular counters. In *International Computer Science Symposium in Russia* (pp. 125–137). Springer, Berlin, Heidelberg.

[5] Williams, J.W.J., 1964. Algorithm 232: heapsort. *Commun. ACM*, 7, pp. 347–348.

[6] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2009. *Introduction to algorithms*. pp. 191–193. MIT press.

[7] Fredman, M.L. and Tarjan, R.E., 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* (*JACM*), 34(3), pp. 596–615.

[8] Black, P.E., pigeonhole sort, in *Dictionary of Algorithms and Data Structures* [online], ed. 11 February 2019. (accessed 14 May 2019) Available from: https://www.nist.gov/dads/HTML/pigeonholeSort.html

[9] Message, R., fibonacci, ed. 10 July 2018. (accessed 14 May 2019) [online], Available from: https://github.com/robinmessage/fibonacci

Appendix A

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `heap` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
- `tools` ..... various tools described in readme
- `data.tar.xz` ..... original datasets used for this thesis