Selectable Heaps and Their Application to Lazy Search Trees

by

Lingyi Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis is based on the materials of a paper I co-authored with Bryce Sandlund.

[38] Bryce Sandlund and Lingyi Zhang. Selectable heaps and optimal lazy search trees, 2020

Abstract

We show the $O(\log n)$ time extract minimum function of efficient priority queues can be generalized to the extraction of the k smallest elements in $O(k \log(n/k))$ time. We first show the heap-ordered tree selection of Kaplan et al. can be applied on the heap-ordered trees of the classic Fibonacci heap to support the extraction in $O(k \log(n/k))$ amortized time. We then show selection is possible in a priority queue with optimal worst-case guarantees by applying heap-ordered tree selection on Brodal queues, supporting the operation in $O(k \log(n/k))$ worst-case time. Via a reduction from the multiple selection problem, $\Omega(k \log(n/k))$ time is necessary.

We then apply the result to the lazy search trees of Sandlund & Wild, creating a new interval data structure based on selectable heaps. This gives optimal $O(B + n)$ lazy search tree performance, lowering insertion complexity into a gap Δ_i to $O(\log(n/|\Delta_i|))$ time. An $O(1)$-time merge operation is also made possible under certain conditions. If Brodal queues are used, all runtimes of the lazy search tree can be made worst-case. The presented data structure uses soft heaps of Chazelle, biased search trees, and efficient priority queues in a non-trivial way, approaching the theoretically-best data structure for ordered data.

Acknowledgements

I would like to thank everyone who made this thesis possible.

Table of Contents

List of Tables

Chapter 1

Introduction

In this thesis, we deal with extending the operations of a priority queue and its efficient implementation as a "heap". While originally defined by Williams [43] as an implicit representation of a tree, the term heap has come to be used for any tree based data structure for the abstract data type priority queue. That is, one supporting the operations insert and extract minimum. The key property is that the trees need to satisfy heap ordering, i.e., a tree is heap ordered if the value associated with each node is at most that associated with its parent.

The operations of a priority queue have also been extended for various applications. Efficient creation (better than inserting elements one at a time) of a heap is an obvious extension, as is the merging of two heaps or decreasing the value of an element.

In this thesis we extend the operations even further, adding the deletion of several elements, the extraction of the k smallest and, perhaps most notably, selecting the k smallest values (in no particular order). For this reason, we call our implementation a selectable heap.

The operations supported by our extended priority queues are:

- `MakePQ()` := Create a new, empty priority queue.
- `Merge(`q_1`,` q_2`)` := Return a new priority queue containing all elements of priority queue q_1 and q_2, destroying q_1 and q_2.
- `Insert(`e`)` := Add element e to the priority queue.
- `DecreaseKey(`e`,` v`)` := Decrease the key of e to v.
- `Delete(`e`)` := Delete element e from the priority queue.
- `FindMin()` := Return the minimum element of the priority queue.
- `ExtractMin()` := Return and remove the minimum element of the priority queue.
- `SelectK(`k`)` := Return the k smallest elements of the priority queue, in no particular order.

- `ExtractK(`k`)` := Return and remove the k smallest elements of the priority queue, in no particular order.
- `Delete(`$e_1, \ldots, e_k$`)` := Remove elements $e_1, \ldots, e_k$ from the priority queue.

We extended the original priority queue data type over previous work to support `SelectK(`k`)`, `ExtractK(`k`)`, and `Delete(`$e_1, \ldots, e_k$`)` operations.

The term "heap", as noted above was originally defined by Williams [43] as what we would now call an implicit data structure [35], i.e. the elements are stored in the first n locations of an array and the only structural information is that which can be inferred from their relative values. The heap was a representation of a perfectly balanced and left adjusted binary tree in which parent values were at most that of their children. With the root (smallest value) in position $A[1]$, the children of $A[i]$ are in locations $2i$ and $2i+1$. While the original purpose of this structure was to enable the first $O(n \log n)$ in place sorting algorithm, it was obvious that it could be used as a priority queue (operations insert and extract minimum). Soon after, Floyd [18] showed a heap could be implemented in about $2n$ comparisons and later Gonnet and Munro [22] reduced this to $1.625n$.

Over the past 60 years, the term heap has come to be applied to any tree ordered (parent less than child) implementation of a priority queue; and, as noted, the notion of priority queue has been extended to include more operations as required for particular applications. Before getting into the details of the complexity for selectable heap, we start with the literature of priority queues.

The binary heaps were the first minimum priority queue data structure that were invented in 1964 by Williams for the heapsort algorithm [43]. Binary heaps support $O(\log n)$ time extract minimum and insert operations. Due to their simplicity and storage of elements in an array, binary heaps or their generalization to d-ary heaps [27, 40] continue to be one of the most practical priority queues. Later the priority queues were extended since some applications require additional operations. The binomial heap was invented in 1978 [41], providing a simple priority queue that supports insertion in $O(1)$ amortized time and the merge of two heaps also in $O(1)$ amortized time [33].

A breakthrough in efficient priority queue literature, Fibonacci heaps, came in 1984 via generalizing binomial heaps to support an efficient decrease-key operation [20]. Fibonacci heaps are created first to efficiently support Dijkstra's single source shortest paths algorithm which makes essential use of priority queues, and in particular the primitive of lowering the priority of an existing element in the priority queue. Fibonacci heaps can perform insert, merge, and decrease-key all in $O(1)$ amortized time. Fibonacci heaps provide improved runtimes in shortest path and minimum spanning tree algorithms, among other applications [20]. A number of priority queues with time bounds matching or close to Fibonacci heaps have since been developed, including pairing heaps [21, 15], strict Fibonacci heaps [5], rank-pairing heaps [23], Brodal queues [3], and many others [7, 16, 24, 14, 25, 2, 30, 31, 37]. The pairing heaps [21, 15] simplifies the `ExtractMin` operations, but they pay a higher cost for `DecreaseKey` operations. This was later improved in Rank-pairing heaps [23] which combine the asymptotic efficiency of Fibonacci heaps with much of the simplicity of pairing heaps. Brodal gave two priority queues

matching Fibonacci heap bounds but in the worst-case. The Brodal queues [3] were first presented in 1996, and the improved version strict Fibonacci heaps [5] were given in 2012. Although many results claim to be a simpler or more practical alternative to Fibonacci heaps [7, 24, 23, 16, 30, 31, 37], Fibonacci heaps continue to be one of the most-taught, most-performant, and simplest-to-code priority queues with optimal theoretical efficiency [24, 34]. More on the history and breadth of research on priority queues is given in the recent survey by Brodal [4].

1.1 Overview

In this thesis we show the extract minimum function of priority queues, particularly, Fibonacci heaps [20] and Brodal queues [6], can be generalized to the extraction of the k smallest elements. Via a reduction from the multiple selection problem, we show a comparison-based priority queue supporting insertion in $o(\log n)$ time must take $\Omega(k \log(n/k))$ time for the extraction of the k smallest elements. This lower bound is not surprising since we need to pay for sorting the remaining elements after the k smallest extraction, but we save the cost for sorting the k elements removed from the set.

The Fibonacci heap is data structure for a broad set of priority queue operations that has a better amortized runtime than many other priority queue implementations. It is a collection of heap-ordered trees such that the size of the (sub)tree rooted at any node x of degree d in the heap must have size at least F_{d+2}, where F_k is the kth Fibonacci number. In the original Fibonacci heap paper [20], it was proven that Fibonacci heaps support `MakePQ()`, `Merge()`, `Insert()`, `DecreaseKey()`, and `FindMin()` in $O(1)$ amortized time and `ExtractMin()` and `Delete()` in $O(\log n)$ amortized time. We show `FindMin()` and `ExtractMin()` operations can be expanded to `SelectK(`k`)` and `ExtractK(`k`)`, respectively. We apply the heap-ordered tree selection algorithm of Kaplan, Kozma, Zamir, and Zwick [29] to the heap-ordered trees of Fibonacci heaps, supporting `SelectK(`k`)` and `ExtractK(`k`)` in $O(k \log(n/k))$ amortized time.

We then show Brodal queues [3] also be made to support efficient selection. The Brodal queue is a priority queue implementation that take advantage of lazy merge and the power of constant time array index access. Brodal queues match all time bounds of Fibonacci heaps for `MakePQ()`, `Merge()`, `Insert()`, `DecreaseKey()`, `FindMin()`, `ExtractMin()`, and `Delete()` operations but in the worst-case rather than just amortized. We show heap-ordered selection [29] can be made to work on a Brodal queue, giving `SelectK(`k`)` and `ExtractK(`k`)` in $O(k \log(n/k))$ worst-case time. This provides a priority queue with optimal worst-case time operations for all standard operations while also supporting optimal worst-case time `ExtractMin()`. Finally, we show both Fibonacci heaps and Brodal queues support `Delete(`$e_1, \ldots, e_k$`)` in $O(k \log(n/k))$ time amortized and worst-case respectively.

The exact conditions necessary for efficient selection in a heap are rather delicate. In order to have the selection algorithm [29] fast enough for k selection, we want to limit the number of nodes with large degrees. For the worst case, a forest of size n with k roots and all non-root nodes being the direct children of the root nodes will require $O(n)$ time for

Operation	FindMin	ExtractMin	Insert	DecreaseKey	Merge
Binary	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Binomial	$O(1)$	$O(\log n)$	$O(1)*$	$O(\log n)$	$O(1)*$
Fibonacci	$O(1)$	$O(\log n)*$	$O(1)$	$O(1)*$	$O(1)$
Pairing	$O(1)$	$O(\log n)*$	$O(1)$	$O(\log n)*$	$O(1)$
Brodal	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
Rank-pairing	$O(1)$	$O(\log n)*$	$O(1)$	$O(1)*$	$O(1)$
Strict Fibonacci	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$

We use $*$ when the runtime is amortized; otherwise, the runtime we list in the table is all in worst-case time.

Table 1.1: The runtime for the priority queue implementations

selection. One of the necessary conditions will be that the degree of each node must be bounded by the logarithm of subtree size, but this alone is not sufficient. For example, a chain of nodes each with $O(\log n)$ child leaves would require $\Omega(k \log n)$ time to select the k smallest. Fibonacci heaps do not permit such structures because child subtrees must be of exponentially-increasing size, the key property we use in our proof. Brodal queues follow even more rigorous structure constraints, and the technical difficulty of our proof for efficient selection is getting around the potentially $O(n)$ nodes throughout the heap that do not satisfy heap order.

Various existing priority queues seem roughly evenly-divided in their ability to also support efficient selection. Specifically, it appears relaxed heaps [14], two-tier pruned binomial queues [17], hollow heaps [24], and thin and fat heaps [30] can also support selection in $O(k \log(n/k))$ time, while quake heaps [7], strict Fibonacci heaps [5], pairing heaps [21], rank-pairing heaps [23], and violation heaps [16] do not.

It is not surprising that various seemingly-unrelated ordered data structures see simultaneous use in our construction. As optimal solutions in their own domains, an optimal solution in the general domain will likely require similar ideas. On the other hand, with regards to selection in priority queues, it may seem surprising that existing efficient priority queues can support the selection of the k smallest elements in optimal time without much modification, despite being designed only for the extraction of the minimum element. In any case, the fact these classic heap data structures support efficient selection suggests they make few comparisons beyond the minimal required of the underlying partial order and keep this information efficiently accessible.

1.2 Application

One of the applications of selectable heap is that it can be used an interval data structure for the lazy search tree [37]. A lazy search tree is a comparison-based data structure that supports the operations of a binary search tree; this operation set is referred to as a *sorted dictionary* in [37]. (A formal description of allowed operations is given in Section 3.) Instead of sorting elements on insertion, as does a binary search tree, lazy search trees store

bags of unsorted elements in a partition into *gaps* based on key order. Specifically, a set of gaps $\Delta_1, \ldots, \Delta_m$ are maintained such that for any $x \in \Delta_i$ and $y \in \Delta_{i+1}$, $x \leq y$. Inserted elements are placed into a gap respecting the key-order partition, and each query falls into a gap and *splits* the gap into two new gaps at a position associated with the query operation. Lazy search trees are able to provide superior runtimes to binary search trees on operation sequences with small number of queries or with non-uniformly distributed queries. For example, n insertions and q queries can be served in $O(n \log q + q \log n)$ time, q queries for k consecutive keys with n interspersed uniformly-distributed insertions can be served in $O(n \log q + qk \log n + n \log\log n)$ time, or the data structure can be used directly as a priority queue with $O(\log\log n)$ time insert and decrease-key operations. More generally, if we take $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$, lazy search trees serve an operation sequence of n insertions and q distinct queries in $O(B + \min(n \log\log n, n \log q))$ time, where $\Omega(B + n)$ is a lower bound. Precise performance statements and further applications are stated in [37].

The interval data structure given in the original lazy search tree paper can be made into a selectable heap with time complexities matching those stated herein for Fibonacci heaps and Brodal queues, except that `Insert()` and `DecreaseKey()` take $O(\log\log n)$ time and `SelectK(`k`)` can be supported in $O(k)$ amortized time. (Although operations `FindMin()` and `SelectK(`k`)` are not explicitly addressed in [37], in Section 1.2, "Selectable Priority Queue", it is stated how they can be performed.) By building an interval data structure off Fibonacci heaps or Brodal queues, we show the following:

1. Lazy search trees can achieve optimal $O(B + n)$ time performance over a sequence of n insertions and q distinct queries, lowering insertion complexity into gap Δ_i from $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|, \log q))$ to $O(\log(n/|\Delta_i|))$. This lowers the time complexity for n uniformly-distributed insertions and q interspersed queries for k consecutive keys to $O(n \log q + qk \log n)$ and improves insertion and decrease-key complexity as a priority queue to $O(1)$ time. (This answers open problem 2 from [37].)

2. Lazy search trees can be made to support $O(1)$ time merge when used as a priority queue, among other situations. (This answers open problem 4 from [37].)

3. Queries in a lazy search tree can be made worst-case in the general case of two-sided gaps. (This addresses open problem 5 from [37]; a fully-general worst-case time solution does not appear possible while keeping change-key in the exact model given in [37]. We do offer a worst-case time solution with fully-general gap merge and change-key supported as deletion and re-insertion in $O(\log n)$ time.)

The proposed data structure makes fundamental use of soft heaps [8, 32, 29], biased search trees, and efficient priority queues. A soft heap is a variant on the simple heap data structure that allows constant amortized time deletion. The main difference between soft heaps and regular heaps is that soft heaps are allowed to increase the keys of some of the items in the heap by an arbitrary amount. Such items are said to become corrupt. In some sense, this research approaches a unification of different ordered data structures into a single theory of a best data structure for ordered data. Optimal priority queues (a single gap Δ_1), online dynamic multiple selection [1] (gaps are separated by queried

ranks), and binary search trees (every element is in its own gap) are special cases of our solution. This paper closes the book theoretically in the gap-based model proposed in [37]; any further work in this research direction requires generalization of the gap model, stated in [37] as open problem 1: Extend the model and provide a data structure so that the order of operations performed is significant.

1.3 Organization

We organize the remainder of this thesis as follows. In the next chapter, in Section 2.1, we show a priority queue supporting insertion in $o(\log n)$ time must take $\Omega(k \log(n/k))$ time to extract the k smallest elements. In Section 2.2, we describe the general framework of heap-ordered tree selection from [29]. In Section 2.3, we show how to support `SelectK(`k`)`, `ExtractK(`k`)`, and `Delete(`$e_1, \ldots, e_k$`)` in a Fibonacci heap in $O(k \log(n/k))$ amortized time. In Section 2.4, we show how to support `SelectK(`k`)`, `ExtractK(`k`)`, and `Delete(`$e_1, \ldots, e_k$`)` in a Brodal queue in $O(k \log(n/k))$ worst-case time. In Chapter 3, we apply selectable heaps to lazy search trees, achieving optimal performance, supporting merge as a priority queue, and giving worst-case time operations for the general case of two-sided gaps. We give concluding remarks in Chapter 4.

This thesis is written assuming the reader is familiar with the previous work on priority queues, especially in Fibonacci heaps [20] and Brodal queues [3], though a brief summary of this background is given in Section 2.3 and 2.4. In Section 2.2, we will give a high level idea about the algorithm we are using. However, the reader can get a better understanding of the algorithm we are using and other selection algorithms from the original paper [29].

Chapter 2

Selectable Heaps

Before giving the full details of the implementation of the selectable heap and analysis of its complexity, we will start with the lower bound on selection in a priority queue in Section 2.1. Then we will introduce the major tool we use in Section 2.2. We will give the details of implementation based on Fibonacci heap and Brodal queue respectively in Sections 2.3 and 2.4 respectively.

2.1 Lower bound

In this section, we give a simple lower bound for extraction of the k smallest elements in a priority queue.

Theorem 1. *A priority queue supporting insertion and extraction of the k smallest elements must have $\Omega(\log(n/k))$ time insertion per element or $\Omega(k\log(n/k))$ time extraction of the k smallest elements.*

Proof. We can reduce a multiple selection instance [28] to the selectable priority queue. We insert all n elements and repeatedly extract the k smallest. The lower bound for this multiple selection instance is $n\log(n/k) + o(n\log(n/k)) + O(n)$ comparisons [28]. Since selectable priority queue requires both insertion operations and extraction operations, this implies either insertion must take $\Omega(\log(n/k))$ time or extraction of the k smallest elements must take $\Omega(k\log(n/k))$ time. □

Corollary 2. *A priority queue supporting `Insert()` and `ExtractK(k)` must spend either $\Omega(\log n)$ time on `Insert()` per element or $\Omega(k\log(n/k))$ time on `ExtractK(k)`.*

Proof. As we consider priority queues with extraction operations with k specified as a parameter, Theorem 1 implies such a priority queue must either have $\Omega(\log n)$ time insertion per element or $\Omega(k\log(n/k))$ time extraction of the k smallest elements. □

Corollary 2 implies the priority queues we consider in this work must take $\Omega(k\log(n/k))$ time on `ExtractK(`k`)` operations.

2.2 Heap-ordered tree selection

We use the heap-ordered tree selection algorithm `Soft-Select-Heapify(`r`)` from Kaplan, Kozma, Zamir, and Zwick [29], where r is the root of the tree. Algorithm `Soft-Select-Heapify(`r`)` works on arbitrary heap-ordered trees in which nodes may have different degrees. We will use `Soft-Select-Heapify(`r`)` as a black box. Given a heap-ordered tree with root r, `Soft-Select-Heapify(`r`)` will return a set of k smallest items from the tree. This is in contrast to a different algorithm by Frederickson [19] which can be used to achieve the same result but is quite complicated.

We need the following definition from [29].

Definition 3 ([29])**.** *For any subtree, S rooted at the root of T, let $\Delta(S)$ denote the sum of the degrees of the nodes of S in the tree T. Then, define*

$$D(T,k) := \max(\Delta(S)) \text{ for all subtrees } S \text{ of size } k \text{ rooted at the root of } T.$$

Kaplan et al. [29] give the following examples. A large enough d-ary tree has $D(T,k) = dk$, as each node has degree d. As another example, consider a tree T where each node at level i has degree $i+2$; then, $D(T,k) = \sum_{i=0}^{k-1} i+2 = k(k+3)/2$, where the subtrees achieving this maximum are paths starting from the root. Their heap-ordered tree selection theorem is the following.

Theorem 4 ([29])**.** *Let T be a heap-ordered tree with root r. Algorithm `Soft-Select-Heapify(`r`)` selects the set of k smallest items in T in $O(D(T,3k))$ time.*

We give the high level idea of the algorithm `Soft-Select-Heapify(`r`)` here. The algorithm uses the soft heap for selection. A soft heap is a variant on the simple heap data structure that allows "corruption". A node is corrupted if the key is increased. The number of corrupted nodes in the tree can be controlled by a selected parameter $0 < \epsilon \leq 1/2$. More precisely, the guarantee offered by soft heaps is the following: for a fixed value ϵ, at any point in time there will be at most ϵn corrupted keys in the heap, where n is the number of elements inserted across the lifetime of the heap (which can be more than the number of current elements). Soft heaps can achieve constant amortized time `Delete`, `Insert`, `Merge`, `ExtractMin`, and `FindMin` operations. Deterministic linear time selection algorithm is one of the applications of soft heaps.

Algorithm `Soft-Select-Heapify(`r`)` starts by initializing an empty soft heap Q and set S. The algorithm first adds the root r of the input heap into the soft heap Q. The algorithm then performs $k-1$ iterations of selection. In each iteration, the algorithm first performs $(e,C) :=$`ExtractMin(`Q`)`, where e is the node with minimum key in Q and C is the set of nodes that are corrupted by the operation. If e is not corrupted, it is added to C. Then the algorithm inserts all the children of nodes in C into both Q and S. After $k-1$ iterations, the algorithm performs a k selection on set S to find the k smallest items required.

With a carefully chosen parameter $\epsilon = 1/6$, we can limit the total number of corruptions of the above algorithm to $2k$. Thus, there will be at most $3k$ nodes whose children are inserted into the soft heap Q. We have the runtime of the algorithm to be $O(D(T,3k))$.

2.3 The Fibonacci heap selection

In this section we describe our algorithm and analysis for the Fibonacci heap [20] selection. As we mentioned before, Fibonacci heaps support `MakePQ()`, `Merge()`, `Insert()`, `DecreaseKey()`, and `FindMin()` in $O(1)$ amortized time and `ExtractMin()` and `Delete()` in $O(\log n)$ amortized time. We will show in this section how we apply the heap-ordered tree selection algorithm of Kaplan, Kozma, Zamir, and Zwick [29] to expand `FindMin()` and `ExtractMin()` operations to `SelectK(`k`)` and `ExtractK(`k`)` and prove that `SelectK(`k`)` and `ExtractK(`k`)` operations can be done in $O(k \log(n/k))$ amortized time.

Fibonacci heaps store a forest of heap-ordered trees and a pointer to the minimum root. `FindMin()` returns the minimum root that the pointer points to. `Insert()` adds a single new root with no children to the forest, possibly updating the pointer to the minimum root. `Merge()` combines two forests into one, again possibly updating the minimum pointer. `ExtractMin()` removes the smallest element, makes all its children new roots in the forest, finds the next smallest element, and then repeatedly combines roots of the same degree. `DecreaseKey()` is supported with a marking scheme. Each non-root node can either be marked or unmarked. If a node x is marked, this implies x has lost a child. When a marked node x loses a second child, x is removed from the list of children stored at its parent, $p(x)$, the subtree rooted at x is made a new root, and x is unmarked. The parent $p(x)$ is then either marked or also removed to form a new root, recursively. `DecreaseKey()` of a node x simply makes x a new unmarked root and processes its parent $p(x)$ via the marking scheme. `Delete()` calls `DecreaseKey()` on the element and decreases the key to $-\infty$.

It is perhaps surprising that a tree T of a Fibonacci heap satisfies $D(T, k) = O(k \log(n/k))$. The bound is easier to show on a binomial heap [41], since a binomial tree of degree k has exactly 2^k nodes. Fibonacci heaps have a more-flexible structure, in particular giving only *lower bounds* on the degree of child nodes. We cannot find a fixed relation between the degree of the node and the size of the subtree rooted at such node. In order to bound the runtime of the selection algorithm, it will require bounding the total degree sum by showing that a large degree sum implies more than n nodes in the Fibonacci heap. The fundamental property used is that children of a node have exponentially-increasing subtree size. In Fibonacci heaps, this is due to the following lemma.

Lemma 5 ([20])**.** *Let x be any node in a Fibonacci heap. Arrange the children of x in the order they were linked to x, from earliest to latest. Then the ith child of x has a degree of at least $i - 2$.*

Lemma 5 can be used to prove the following corollary.

Corollary 6 ([20])**.** *A node of degree k in a Fibonacci heap has at least $F_{k+2} \geq \phi^k$ descendants, including itself, where F_k is the kth Fibonacci number and $\phi = (1 + \sqrt{5})/2$ is the golden ratio.*

We give the main theorem of the section below.

Theorem 7. *Fibonacci heaps support `SelectK(k)`, `ExtractK(k)`, and `Delete(`$e_1, \ldots, e_k$`)` in $O(k \log(n/k))$ amortized time.*

Proof. We first create a heap-ordered tree T_{big} from the collection of roots stored in the Fibonacci heap by creating a dummy node d with value $-\infty$ and linking all roots below it. We then perform `Soft-Select-Heapify(`d`)` [29] to select the $k+1$ smallest elements from T_{big}. We show $D(T_{big}, k) = O(t + k\log(n/k))$, where t is the number of roots the Fibonacci heap contains at the time of selection.

Consider the subtree T_{select} of T_{big} of selected nodes. Subtree T_{select} contributes $O(k)$ to the degree of the selected nodes in tree T_{big}. Let us thus ignore this contribution and consider only unselected children of selected nodes. As there are a total $n+1$ nodes in T_{big}, we will maximize the sum of unselected children by attaching subtrees of smallest size to every selected node other than d, so that we may attach as many of them as possible. By Lemma 5 and Corollary 6, the first two unselected subtrees we attach must be of degree at least 0, containing 1 element, then the third must be of degree 1, containing 2 elements, and the jth subtree must be of degree at least $j-2$, containing at least ϕ^{j-2} elements. The number of attached subtrees, ignoring the contribution of T_{select}, is maximized when we attach the same number i to each selected node. Solving for i in

$$k + k\sum_{j=2}^{i} \phi^{j-2} \geq n - k,$$

we can determine

$$i \leq \frac{\log(1 + \frac{(\phi-1)(n-2k)}{k})}{\log \phi} + 1.$$

It follows that $i \leq \log_\phi(n/k) + 1$. Adding back the contribution of T_{select} affects the above analysis by no more than k, so that in total each selected node has degree $O(\log(n/k))$, besides dummy node d which has degree t. Structural limitations imposed by the particular tree T_{big} can only make average degree decrease by replacing smaller subtrees with larger subtrees. It thus follows that $D(T_{big}, k) = O(t + k\log(n/k))$ and by Theorem 4, selection in T_{big} takes $O(t + k\log(n/k))$ time.

To complete operation `SelectK(`k`)`, we reduce t to at most $\log n$ by repeatedly combining roots of equal degree. By the potential function of the Fibonacci heap [20], this releases t units of potential and achieves amortized $O(k\log(n/k))$ time selection. To complete operation `ExtractK(`k`)`, we first remove the k smallest elements. This leaves $O(t+k\log(n/k))$ independent trees. We then again repeatedly combine roots of equal degree, resulting in at most $\log n$ independent trees. The amortized time complexity of `ExtractK(`k`)` is thus also $O(k\log(n/k))$.

The above degree bounds did not require the subtree T_{select} be minimal, or even connected. Our argument shows the total sum of degrees of any k nodes in a Fibonacci heap is $O(k\log(n/k))$. This allows an $O(k\log(n/k))$ time `Delete(`$e_1, \ldots, e_k$`)` operation. For each i, we remove the subtree rooted at e_i and perform cascading cuts until an ancestor of e_i is not marked or a root is reached, as in the decrease-key operation. The final unmarked ancestor, if not a root, is then marked and all children of e_i are added to the collection of roots. The total number of children is $O(k\log(n/k))$ and each cascading cut operation takes $O(1)$ amortized time per e_i. Thus the entire deletion can be performed in $O(k\log(n/k))$ amortized time. □

2.4 The Brodal queue selection

Brodal queues [3] are much more complicated than Fibonacci heaps, but allow all operations in worst-case time. Nodes are stored in a tree T_1 or possibly another tree T_2 which is incrementally merged into T_1. Each node has a non-negative integer rank assigned to it. For each node x, we use $p(x)$ as the parent of x, $r(x)$ as the rank[1] of x, and $n_i(x)$ as the number of children of rank i that x has. Finally, we use t_i as the root of T_i. Nodes which satisfy heap order, i.e. they are larger than their parents, are called good nodes. Nodes which are not good are called *violating* nodes.

Brodal queues use 13 invariants to maintain the structure. The invariants can be classified into three sets S, O and R respectively. The invariants in the set S apply to all nodes in the Brodal queue, while the invariants in the set R apply to the root of the trees. The invariants in set O control the violations.

For any node x, the following invariants are satisfied.

S1 : If x is a leaf, then $r(x) = 0$,

S2 : $r(x) < r(p(x))$,

S3 : if $r(x) > 0$, then $n_{r(x)-1}(x) \geq 2$,

S4 : $n_i(x) \in \{0, 2, 3, \ldots, 7\}$,

S5 : $T_2 = \varnothing$ or $r(t_1) \leq r(t_2)$.

Each node x has rank $r(x)$ and at most 7 children of any rank less than $r(x)$ (Invariants S2 and S4). Additionally, x cannot have a single child of any rank (S4) and must have at least two children of rank $r(x) - 1$ (S3). Leaves have rank 0 (S1).

Beside the good nodes, Brodal queues also allow violating nodes. To keep track of the violating nodes, each node x is associated with two subsets $V(x)$ and $W(x)$. If a node y is smaller than its parent $p(y)$, the violation is stored in a violating set $V(x)$ or $W(x)$ for some node $x \leq y$. Despite the fact that each node x might need to maintain non-empty subsets $V(x)$ and $W(x)$, elements will only be inserted to either $V(t_1)$ or $W(t_1)$ (t_1 is the root of T_1). The V sets take care of large violations, i.e. violations that have rank larger than $r(t_1)$ when they are created will be added to $V(t_1)$. The W sets handle smaller violations, i.e. violations that have rank less than or equal to $r(t_1)$. We use $w_i(x)$ to denote the number of nodes of rank i in the set $W(x)$. Brodal uses the constant α for the number of large violations that can be created between two increases in the rank of t_1. In the original paper [3], Brodal uses the following set of invariants to maintain the violations.

O1 : $t_1 = \min T_1 \cup T_2$,

[1] Rank is used here as an assigned parameter. The term rank is used for different concepts in chapter 2 and 3.

O2 : if $y \in V(x) \cup W(x)$, then $y \geq x$,

O3 : if $y < p(y)$, then an $x \neq y$ exists such that $y \in V(x) \cup W(x)$,

O4 : $w_i(x) \leq 6$,

O5 : if $V(x) = (y_{|V(x)|}, \ldots, y_2, y_1)$, then $r(y_i) \geq \lfloor (i-1)/\alpha \rfloor$ for $i = 1, 2, \ldots, |V(x)|$, where α is a constant.

From the above invariants, we can obtain the upper bound of the size of V and W sets. The set $W(x)$ contains at most 6 violations of any rank (O4) and the set $V(x)$ contains at most $m\alpha$ violations of rank less than m, where α is a constant (O5).

At the roots t_1 of T_1 and t_2 of T_2, invariants are stronger. Each root t_j has at least two children of every rank (R1).

R1 : $n_i(t_j) \in \{2, 3, \ldots, 7\}$ for $i = 0, 1, \ldots, r(t_j) - 1$,

R2 : $|V(t_1)| \leq \alpha r(t_1)$,

R3 : if $y \in W(t_1)$, then $r(y) < r(t_1)$.

A *guide* data structure [30, 17] is used to efficiently manage the invariants R1 and O4. Given a sequence of integer variables $x_k, x_{k-1}, \ldots, x_1$ and $x_i \leq T$ for some threshold T, we can only perform `Reduce(i)` operations on the sequence which decrease x_i by at least two and increase x_{i+1} by at most one. The x_is can be forced to increase and decrease by one, but for each change in an x_i we are allowed to do $O(1)$ `Reduce` operations to prevent any x_i from exceeding T. The guide data structure will tell us which operations to perform in $O(1)$ time.

Notice that, if we have at least three nodes of the same rank i, we can combine those nodes and create a new node with rank $i+1$. Similarly, we can also split a node of rank $i+1$ to several nodes of rank at most i. We call the first operation *linking* and the latter *delinking*. For each node, if we count the number of children of each rank and list the number from highest rank to lowest, we obtain a sequence of integer variables $x_k, x_{k-1}, \ldots, x_1$. We can perform linking and delinking operation for the children of the same node, and we can reflect the changes of the rank in the above sequence by performing `Reduce(i)` operations. We can use two instances of the guide data structure for each node to maintain both a lower and upper bound on the number of children of each rank so that addition and removal of a child of any rank can be supported at the roots t_1 and t_2 in $O(1)$ worst-case time. A guide data structure is also used to maintain the upper bound given in Invariant O4 on $W(t_1)$. Two violations of the same rank r can be reduced to at most one of rank $r+1$ in worst-case $O(1)$ time.

Brodal queue [3] selection will differ from Fibonacci heap selection in a couple ways. The most important is handling violating nodes. We treat violating nodes with on-the-fly conversion into a proper heap-ordered tree. The second is that the rigid rank structure of

Brodal queues can actually allow us to simplify the proof of degree bounds compared to the approach taken in the previous section. Namely, rather than lower bounding the sizes of unselected subtrees, we can directly upper bound the number of nodes of high degree. We first give a lemma from [3][2].

Lemma 8 (Brodal [3])**.** *A node x with rank $r(x)$ in a Brodal queue has subtree of size at least $2^{r(x)+1} - 1$.*

Lemma 8 implies the maximum rank r in a Brodal queue is at most $\log_2(n)$.

We can use S2 and Lemma 8 to get a bound on the number of nodes in a Brodal queue of a particular rank.

Lemma 9. *A Brodal queue has at most one node of rank $\lceil \log_2(n) \rceil$, two nodes of rank $\lceil \log_2(n) \rceil - 1$, and in general at most 2^i nodes of rank $\lceil \log_2(n) \rceil - i$, where $i \leq \lceil \log_2(n) \rceil$.*

Proof. By S2, nodes of the same rank cannot be descendants of each other. This implies their subtrees are disjoint. By Lemma 8, a node of rank $\lceil \log_2(n) \rceil - i$ has at least $n/2^{i-1} - 1 \geq n/2^i$ descendants. Thus there can only be 2^i such nodes. □

Theorem 10. *Brodal queues support `SelectK(k)`, `ExtractK(k)`, and `Delete`$(e_1, \ldots, e_k)$ in $O(k \log(n/k))$ worst-case time.*

Proof. As in the Brodal queue delete minimum operation, we first empty tree T_2 by moving all children of t_2 to T_1 and making t_2 a rank 0 child of T_1. We then call `Soft-Select-Heapify`(t_1) to select the k smallest nodes in T_1.

Whenever we reach a node y, we consider nodes of $V(y)$ and $W(y)$ to be children of y in the selection algorithm. From invariant O2, we know that $x \geq y$. It is thus possible to reach a node x via its proper parent or a node y in which $x \in V(y) \cup W(y)$. Upon encountering x, we check to make sure it was not already selected. Since x is in the violating set $V(y)$ or $W(y)$ of at most one node y, the total extra work for encountering x on two paths is $O(1)$, totaling $O(k)$ for all k selected nodes. Algorithm `Soft-Select-Heapify()` can process the selection by considering the heap to be constructed in this way since the tree T we construct on the fly still remains heap order.

We now consider the cost of the selection. We must show $D(T, k) = O(k \log(n/k))$, where T is the heap-ordered tree we construct on the fly. Observe that the degree of a node x in T is the sum of the number of violating nodes in its sets $V(x)$ and $W(x)$ and the number of its proper children. Observe that the number of proper children of a node x is bounded by $7r(x)$. Thus to bound the number of proper children of selected nodes, it suffices to find an upper bound on the total sum of ranks of selected nodes.

Let the ranks of the k selected nodes in non-increasing order be $r_1, \ldots, r_k$. Then by Lemma 9, $r_1 \leq \lceil \log_2(n) \rceil$, $r_2, r_3 \leq \lceil \log_2(n) \rceil - 1$, $r_4, \ldots, r_7 \leq \lceil \log_2(n) \rceil - 2$, and $r_k \leq \lceil \log_2(n) \rceil - \lfloor \log_2(k) \rfloor \leq \log_2(n/k) + 2$. The total sum of ranks is thus at most

$$2k + \sum_{i=0}^{\infty} \frac{k}{2^i} (\log_2(n/k) + i) = k \log_2(n/k) + 4k.$$

[2]While this is not an explicit lemma in [3], it is explicitly stated and proven on page 2.

It follows the total sum of proper children of selected nodes is $O(k \log(n/k))$.

We can apply a similar strategy to bound the total number of nodes in violating sets V and W of selected nodes. Divide the nodes into two categories: those with rank greater than or equal to $\log_2(n) - \log_2(k)$ and those with rank less than $\log_2(n) - \log_2(k)$. By Lemma 9 again and by employing the above argument, there are at most $O(k)$ nodes in the first category.

We can bound the number of nodes in the second category with invariants O4 and O5. Invariant O4 states that the W sets of selected nodes contain at most 6 nodes per rank. Invariant O5 states that V sets of selected nodes contain at most $m\alpha$ nodes of rank less than m. Together they imply a bound of $(6 + \alpha) \log_2(n/k)$ nodes in total in the second category. As $\alpha = O(1)$, this implies the total number of nodes in violating sets of selected nodes is $O(k \log(n/k))$. Since the number of proper children of selected nodes is also $O(k \log(n/k))$, we have $D(T, k) = O(k \log(n/k))$. By Theorem 4, this implies the heap-ordered tree selection takes $O(k \log(n/k))$ worst-case time.

This shows `SelectK(k)` takes $O(k \log(n/k))$ worst-case time on a Brodal queue. To prove $O(k \log(n/k))$ worst-case runtime for `ExtractK(k)`, we must rebuild the Brodal queue with the k smallest nodes removed. We can do so as follows. We remove nodes other than t_1 one-by-one. Consider the process for a node x. First, we remove x. This may cause $p(x)$ to have only one child of rank $r(x)$, violating S4. If $r(p(x)) > r(x) + 1$, we can remove the other rank $r(x)$ node from $p(x)$ and make it a child of t_1. Otherwise, we can find a rank $r(x)$ child of the root and replace x with it. This may create a violation; if so, we can add it to $W(t_1)$. We then add the proper children of x below t_1. Further, we must deal with $V(x)$ and $W(x)$. We can simply add them to $W(t_1)$.

We can bound the total cost of adding children of removed nodes below t_1 as the above analysis gives for the bound on the number of proper children, at $O(k \log(n/k))$. Similarly, we can bound the number of violations created at $O(k)$ and the number of violations added to $W(t_1)$ at $O(k \log(n/k))$, again following the above analysis. At this point we can then remove t_1, following the extract minimum procedure stated in `DeleteMin(Q)` of [3]. The total time taken to remove the k smallest elements is $O(k \log(n/k))$.

As in the proof of Theorem 7, we again do not need the property that the k nodes removed in `ExtractK(k)` are minimal. We can thus support `Delete(`$e_1, \ldots, e_k$`)` as follows. We again first empty tree T_2 by moving all children of t_2 to T_1 and making t_2 a rank 0 child of T_1. If any $e_i = t_1$, we remove each node one-by-one as in `ExtractK(k)`, but save t_1 for the final removal as in `DeleteMin(Q)` of [3]. Otherwise, we remove each node e_i one-by-one exactly as in the above procedure for `ExtractK(k)`, skipping the final `DeleteMin(Q)` procedure of [3]. Either way the above analysis indicates the total cost will be $O(k \log(n/k))$ worst-case time.

□

Chapter 3

Optimal Lazy Search Trees

Lazy search trees [37] are comparison-based data structures that support the following operations on a dynamic set S with $|S| = n$. (It is straightforward to extend data structures to multisets.) Lazy search trees are designed for scenarios where the number of insertions is larger than the number of queries. The element of rank r is the rth smallest element in the set S. The operations of lazy search trees are referred to as a *sorted dictionary*.

Lazy search trees support the following operations that change the set S.

- `Construction(`S`)` := Construct a sorted dictionary on the set S.
- `Insert(`e`)` := Add element e to S; (this increments n).
- `Delete(`e`)` := Delete e from S, with a pointer to e; (this decrements n).
- `ChangeKey(`e`,` v`)` := Change the key of the element e (with pointer to it) to v.
- `Split(`r`)` := Split S at rank r, returning two sorted dictionaries T_1 and T_2 of r and $n - r$ elements, respectively, such that for all $x \in T_1$, $y \in T_2$, $x \leq y$.
- `Merge(`T_1`,` T_2`)` := Merge sorted dictionaries T_1 and T_2 and return the result, given that for all $x \in T_1$, $y \in T_2$, $x \leq y$.

Lazy search trees support `RankBasedQuery()` to get information about the set. Informally, a rank-based query is a query computable in $O(\log n)$ time on a (possibly augmented) binary search tree and in $O(n)$ time on an unsorted array. The permitted queries include:

- `rank(k)` := Return the rank of key k in the set S.
- `select(r)` := Return the element of rank r in S.
- `contains(k)` := Return true if exists an element $e \in S$ with key k; otherwise return false.
- `successor(k)` := Return the successor of the element e in set S.

- `predecessor(k)` := Return the predecessor of the element e in set S.
- `minimum()` := Return the minimum element of set S
- `maximum()` := Return the maximum element of set S.

For more formal definitions on the permitted queries, see the original lazy search trees paper [37].

Lazy search trees seek to improve the $O(\log n)$ time per-operation complexity given by binary search trees as a sorted dictionary by employing a fine-grained complexity analysis, not unlike that done in dynamic optimality literature [39, 42, 10, 9, 26, 12, 11]. Instead of sorting elements upon insertion, sorting is delayed until query operations. Elements are stored in a partition into gaps $\Delta_1, \ldots, \Delta_m$ such that for $x \in \Delta_i$ and $y \in \Delta_{i+1}$, $x \leq y$. Inserted elements are placed into a gap respecting the key-order partition. Upon query, the gap Δ_i containing rank r is split into two gaps Δ'_i and Δ'_{i+1} such that $|\Delta'_i| + \sum_{j=1}^{i-1} |\Delta_j| = r$ and for $x \in \Delta'_i$, $y \in \Delta'_{i+1}$, $x \leq y$.

The results of [37] are `Construction(`S`)` in $O(n)$ time where $|S| = n$, `Insert()` into a gap Δ_i in $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|, \log q))$ worst-case time (q is the number of queries), `RankBasedQuery()` that splits a gap Δ_i into a smaller gap of size k and a larger gap of size $|\Delta_i| - k$ in $O(k \log(|\Delta_i|/k) + \log n)$ amortized time, `Delete()` in $O(\log n)$ worst-case time, `Split(`r`)` in time as in `RankBasedQuery(`r`)`, and `Merge()` in $O(\log n)$ worst-case time.

The performance of `ChangeKey(`e`,` v`)` is not as easily stated. In general it can be supported in $O(\log n)$ worst-case time as in deletion and reinsertion. More efficient runtimes are possible dependent on the rank of the element e and the nature of the gap Δ_i to which e belongs (and remains after the key-change).

Each gap is either *zero-sided*, *left-sided*, *right-sided*, or *two-sided*. If no queries have occurred, the single gap Δ_1 is zero-sided; if queries have occurred only on the left boundary of Δ_i it is left-sided, only on the right boundary it is right-sided, and otherwise (the typical case) it is two-sided. A zero-sided gap Δ_i supports any key-change operation within Δ_i in $O(1)$ time. A left-sided gap Δ_i supports decrease-key within Δ_i in $O(\min(\log q, \log\log|\Delta_i|))$ time. A right-sided gap Δ_i supports increase-key within Δ_i in the same time complexity. A two-sided gap Δ_i supports decrease-key on elements less than or equal to the median of Δ_i and increase-key on elements larger than or equal to the median of Δ_i, again in the same time complexity.

As previously stated, if we take $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$, lazy search trees serve an operation sequence of n insertions and q distinct queries in $O(B + \min(n \log\log n, n \log q))$ time, where $\Omega(B + n)$ is a lower bound [37]. By using selectable heaps in place of the interval data structure in [37], we can achieve three new results for lazy search trees.

Theorem 11. *1. Lazy search trees can support insertion into gap Δ_i in $O(\log(n/|\Delta_i|))$ time and change-key in $O(1)$ time instead of $O(\min(\log q, \log\log|\Delta_i|))$ time in the conditions stated above, while matching previous time bounds for all other operations. Taking $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$, lazy search trees serve a sequence of n insertions and q distinct queries in $O(B + n)$ time, which is optimal.*

2. *Split of a two-sided gap Δ_i can be done in worst-case instead of amortized time. Split of a left-sided or right-sided gap Δ_i can be done in worst-case instead of amortized time if the larger resulting gap is left-sided or right-sided, respectively.*

3. *Merge of two left-sided gaps or two right-sided gaps can be performed in $O(1)$ amortized or worst-case time. Alternatively, change-key can be made $O(\log n)$ time, the merge of any two gaps can be supported in $O(1)$ time, and all operations of the lazy search tree can be made worst-case.*

Proof. We create a new interval data structure[1] based on selectable heaps. If the gap is zero-sided, the interval data structure is an unsorted array or linked list. If the gap is left-sided, the interval data structure is a selectable min-heap. If the gap is right-sided, the interval data structure is a selectable max-heap. Otherwise, the gap is two-sided, and we partition the elements roughly into thirds. The smallest third of the elements we make into a min-heap, the largest into a max-heap, and the middle third we keep unsorted. We maintain that each third contains at least a $1/3 - \epsilon$ for $0 < \epsilon < 1/6$ fraction of the total elements in the gap. We can maintain static separator elements between the thirds to ensure a valid partition.

As in [37], insertion first locates the gap Δ_i in which the inserted element belongs in $O(\log(n/|\Delta_i|))$ worst-case time. Insertion within the interval data structure then considers which third to place the element, if applicable, then does so in $O(1)$ worst-case time if the selectable heap is Fibonacci [20] or Brodal [3] (or if it is an unsorted array). Change-key is supported as decrease-key in a left-sided heap or increase-key in a right-sided heap, in $O(1)$ time. For a two-sided gap, any element in the middle third can have its key increased or decreased via removal and re-insertion into the proper third in $O(1)$ time, otherwise the min-heap supports decrease-key and the max-heap increase-key, ultimately satisfying the necessary constraints for efficient change-key, in $O(1)$ time. If a Brodal queue is used, the complexity is worst-case.

Query works as follows. As in [37], first the gap Δ_i in which $r \in \Delta_i$ is found in $O(\log n)$ time. If Δ_i is zero-sided we amortize the work against the total number of elements in zero-sided gaps, as in [37], performing the operation in $O(1)$ amortized time. If Δ_i is two-sided, then it should be possible to determine which third the query rank r falls into, if applicable, in $O(1)$ time. If in the middle third, we answer the query in $O(|\Delta_i|)$ time, rebuilding Δ_i into Δ'_i and Δ'_{i+1} as previously described. Otherwise, we must perform selection in a min- or max-heap; without loss of generality, assume it is a min-heap. We repeat the following. We select the smallest 2^j elements into a set X and the smallest 2^{j+1} elements into a set Y, starting at $j = 0$. By Definition 6 in the original paper [37], we can determine which of X or Y contains r. If it is Y, we continue with $j \leftarrow j + 1$; otherwise, we stop and answer the query. We break Δ_i into Δ'_i and Δ'_{i+1} so that $|\Delta'_i| + \sum_{j=1}^{i-1} |\Delta_i| = r$, by extracting $k = r - \sum_{j=1}^{i-1} |\Delta_i|$ elements from the heap. We make Δ'_i into a new two-sided gap in $O(k)$ time. The existing structure of gap Δ_i becomes Δ'_{i+1}.

[1] The term "interval data structure" was used in [37] due to the representation of elements within a gap into a second-level key-order partition into intervals, analogous to the gaps on the first level. The "interval data structure" discussed herein is based on selectable heaps, so at this point the name is a misnomer, but we maintain the nomenclature for consistency.

The time complexity of query can be proven as follows. The time taken for the selections, by Theorems 7 and 10, is proportional to no more than $\sum_{i=0}^{\infty} k/2^i \log(n2^i/k) = O(k \log(n/k))$. Extraction similarly takes $O(k \log(n/k))$ time. If a Brodal queue is used, the time bound is worst-case. However, k may be larger than $|\Delta_i|/2$ if Δ_i was a left-sided or right-sided gap. In this case we can amortized against the total number of elements in one-sided gaps, as is done in [37], to perform the operation in $O(1)$ amortized time.

Operation `Construction(S)` can be completed via insert. Operation `Delete(`e`)` can be performed as priority queue deletion, in $O(\log n)$ time. Operation `Split(`r`)` is performed as query and then an operation on the biased search tree gap data structure, as in [37]. Finally, `Merge(`T_1`,` T_2`)` similarly occurs at the gap level.

One detail remains, which is the maintenance of the partition into thirds in a two-sided gap into fractions of size at least $1/3 - \epsilon$ of the total gap size. For every k elements inserted, removed, or key-changed in an operation, we perform $O(k)$ work towards building a new copy of the interval data structure with a more-accurate partition, as described in [36]. After the new version is constructed it is caught up to the operations performed during construction at twice the pace they occur. When it is caught up the current data structure is thrown away and construction on a new data structure begins. This allows worst-case time maintenance of the partition while keeping all operation complexity the same.

The above shows parts 1 and 2 of Theorem 11. We now consider part 3. Brodal and Fibonacci heaps support $O(1)$ time merge, in worst-case and amortized time, respectively. As left-sided and right-sided gaps are just selectable heaps, we can simply merge the heaps. For the alternative approach, we forget about left-sided, right-sided, or two-sided gaps and store all elements of a gap in both a min- and max-heap, with each element containing a pointer to the other in the opposite heap. All operations work as above, except now for query, when we extract elements from the min- or max-heap, we also delete the elements in the other heap. As stated in Theorems 7 and 10, deletion of the k elements can be completed in $O(k \log(n/k))$ amortized or worst-case time, respectively. Now, since any two gaps has both a min- and a max-heap, we can support the merge of two arbitrary gaps in $O(1)$ time by simply merging the two min- and max-heaps. Change-key is supported only by deletion and re-insertion, in $O(\log n)$ time. All operations can be made worst-case via use of a Brodal queue. □

With the help of Theorem 11 we can improve lazy search trees in the following ways:

1. Lazy search trees can achieve optimal $O(B + n)$ time performance over a sequence of n insertions and q distinct queries, lowering insertion complexity into gap Δ_i from $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|, \log q))$ to $O(\log(n/|\Delta_i|))$. This lowers the time complexity for n uniformly-distributed insertions and q interspersed queries for k consecutive keys to $O(n \log q + qk \log n)$ and improves insertion and decrease-key complexity as a priority queue to $O(1)$ time. (This answers open problem 2 from [37].)

2. Lazy search trees can be made to support $O(1)$ time merge when used as a priority queue, among other situations. (This answers open problem 4 from [37].)

3. Queries in a lazy search tree can be made worst-case in the general case of two-sided gaps. (This addresses open problem 5 from [37]; a fully-general worst-case time solution does not appear possible while keeping change-key in the exact model given in [37]. We do offer a worst-case time solution with fully-general gap merge and change-key supported as deletion and re-insertion in $O(\log n)$ time.)

However, we give worst-case performance only in the general case of two-sided gaps. It appears this may be necessary if change-key is to be supported as a decrease-key operation for all elements in a left-sided gap (analogously, increase-key in a right-sided gap), which is what provides optimal performance as a priority queue. Specifically, the performance of change-key and the ability to perform quick splitting of a gap are coupled. If a lazy search tree is used as a min-heap, allowing decrease-key of all elements, then we cannot find the maximum element in $O(\log n)$ worst-case time. We need to rebuild the data structure so that ranks close to the maximum and minimum can be found efficiently, which necessarily requires the change-key operation to be decrease-key on elements with rank closer to the minimum and increase-key on elements with rank closer to the maximum.

Finally, we observe that the alternative insertion complexity of $O(\log q)$ is also achieved in the version of lazy search trees stated herein. The number of elements in the gap data structure is bounded by q [37], and insertion into the interval data structure based on selectable heaps takes $O(1)$ time. However, $O(\log q)$ vs. $O(\log(n/|\Delta_i|))$ time insertion does not impact overall time complexity on any sequence of operations, so we leave it out of the statement of Theorem 11.

Chapter 4

Conclusion

In this thesis we have shown that the $O(\log n)$ time extract-minimum function of efficient priority queues can be generalized to the extraction of the k smallest elements in $O(k \log(n/k))$ time. We apply selectable heaps to lazy search trees [37], giving an optimal data structure in the gap model, adding a merge function when used as a priority queue, and providing worst-case runtimes in the general case of two-sided gaps. Any further theoretical improvement in lazy search trees would require abstraction to an even more fine-grained model.

We believe by using selectable heaps, we can achieve a more straightforward approach to lazy search trees than that of [37]. With an understanding of Fibonacci heaps [20] or Brodal queues [3], the technical arguments required herein are slightly less involved. However, the approach of [37] based on first principles has its own merit. In [37], the data structure satisfies an $O(\min(n, q))$ pointer bound, where n is the number of elements and q the number of queries. Further, the simple priority queue developed which supports extraction natively is surely more practical. In [37], it is shown in the experiments that a rudimentary implementation of the described data structure with the number of insertions approaches $n \geq 1\,000\,000$ can outperform binary search trees in following two ways: when the number of queries q is way smaller than the number of insertions n ($q \leq \sqrt{n}$), or when the queries ask for consecutive elements in the tree in increasing order. In contrast, considering the complicated structures of Fibonacci heaps [20] or Brodal queues [3], the lazy search tree improvement discussed herein are likely to mostly be of theoretical interest.

For future work, it would be interesting to see if selectable heaps have further applications outside of the straightforward transitive applications through lazy search trees, such as an optimal online multiple selection algorithm [13]. Although of relatively low theoretical import, it would also be interesting to see if selection can be supported on a priority queue with optimal worst-case guarantees in the pointer machine model, such as strict Fibonacci heaps [5]. Such a data structure would allow a lazy search tree with worst-case guarantees on the pointer machine, whereas the version discussed here requires internal use of arrays. Finally, it may be possible to support `SelectK(`k`)` in $O(k)$ time, as do lazy search tree priority queues, but while retaining optimal time bounds for the remaining operations.

References

[1] Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jon Sorenson. Near-optimal online multiselection in internal and external memory. *Journal of Discrete Algorithms*, 36:3–17, 2016. WALCOM 2015.

[2] Gerth Stølting Brodal. Fast meldable priority queues. In *Workshop on Algorithms and Data Structures (WADS)*, pages 282–290. Springer, 1995.

[3] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Symposium on Discrete Algorithms (SODA)*. SIAM, 1996.

[4] Gerth Stølting Brodal. A survey on priority queues. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 150–163. Springer, 2013.

[5] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict Fibonacci heaps. In *Symposium on Theory of Computing (STOC)*. ACM, 2012.

[6] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.

[7] Timothy M. Chan. Quake heaps: A simple alternative to Fibonacci heaps. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 27–32. Springer, 2009.

[8] Bernard Chazelle. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027, 2000.

[9] Richard Cole. On the dynamic finger conjecture for splay trees. part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[10] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part I: Splay sorting $\log n$-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[11] Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Patrascu. The geometry of binary search trees. In *Symposium on Discrete Algorithms (SODA)*, pages 496–505. SIAM, 2009.

[12] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality–almost. *Siam Journal of Computing*, 37(1):240–251, 2007.

[13] David Dobkin and J. Ian Munro. Optimal time minimal space selection algorithms. *Journal of the Association for Computing Machinery*, 28(3):454–461, 1981.

[14] James R. Driscoll, Harold N. Gabow, Ruth Shairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(1):343–354, 1988.

[15] Amr Elmasry. Pairing heaps with $O(\log \log n)$ decrease cost. In *Symposium on Discrete Algorithms (SODA)*. SIAM, 2009.

[16] Amr Elmasry. The violation heap: A relaxed Fibonacci-like heap. *Discrete Mathematics Algorithms and Applications*, 2(4):493–503, 2010.

[17] Amr Elmasry, Claus Jensen, and Jyrki Katajainen. On the power of structural violations in priority queues. In *Proceedings of the thirteenth Australasian symposium on Theory of computing*. ACM, 2007.

[18] Robert W Floyd. Algorithm 245: treesort. *Communications of the ACM*, 7(12):701, 1964.

[19] Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197–214, 1993.

[20] Michael Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery*, 34(3):596–615, 1987.

[21] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[22] Gaston H. Gonnet and J. Ian Munro. Heaps on heaps. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*, volume 140 of *Lecture Notes in Computer Science*, pages 282–291. Springer, 1982.

[23] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-pairing heaps. *SIAM Journal on Computing*, 40(6):1463–1485, 2011.

[24] Thomas Dueholm Hansen, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Hollow heaps. *ACM Transactions on Algorithms*, 13(3):1–27, 2017.

[25] Peter Høyer. A general technique for implementation of efficient priority queues. In *Proceedings of the Third Israel Symposium on the Theory of Computing and Systems*. IEEE, 1995.

[26] John Iacono. Alternatives to splay trees with $o(\log n)$ worst-case access time. In *Symposium on Discrete Algorithms (SODA)*, pages 516–522. SIAM, 2001.

[27] Donald B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3):53–57, 1974.

[28] Kanela Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 103–114. Springer, 2005.

[29] Haim Kaplan, László Kozma, Or Zamir, and Uri Zwick. Selection from Heaps, Row-Sorted Matrices, and $X + Y$ Using Soft Heaps. In Jeremy T. Fineman and Michael Mitzenmacher, editors, *2nd Symposium on Simplicity in Algorithms (SOSA 2019)*, volume 69 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:21, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[30] Haim Kaplan and Robert E. Tarjan. New heap data structures. Technical report, Princeton University, 1999.

[31] Haim Kaplan and Robert Endre Tarjan. Thin heaps, thick heaps. *ACM Transactions on Algorithms*, 4(1), 2008.

[32] Haim Kaplan, Robert Endre Tarjan, and Uri Zwick. Soft heaps simplified. *SIAM Journal on Computing*, 42(4):1660–1673, 2013.

[33] C. M. Khoong and H. W. Leong. Double-ended binomial queues. In *Proceedings of the 4th International Symposium on Algorithms and Computation*. Springer, 1993.

[34] Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. A back-to-basics empirical study of priority queues. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*, pages 61–72. ACM, 2014.

[35] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *J. Comput. Syst. Sci.*, 21(2):236–250, 1980.

[36] Mark H Overmars. *The design of dynamic data structures*, volume 156. Springer Science & Business Media, 1983.

[37] Bryce Sandlund and Sebastian Wild. Lazy search trees. In *Proceedings of the 61st Annual Symposium on Foundations of Computer Science*, 2020.

[38] Bryce Sandlund and Lingyi Zhang. Selectable heaps and optimal lazy search trees, 2020.

[39] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary seach trees. *Journal of the ACM*, 32(3):652–686, 1985.

[40] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[41] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[42] Robert E. Wilber. Lower bounds for accessing binary search trees with rotations. *Siam Journal of Computing*, 18(1):56–69, 1989.

[43] J. W. J. Williams. Algorithm 232 - heapsort. *Communications of the ACM*, 7(6):347–348, 1964.