MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

THÈSES CANADIENNES

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

**LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE**

A NEW CLASS OF PRIORITY QUEUE ORGANIZATIONS

By
Azmeena Hasham, B. Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
July 19, 1986

© copyright
1986, Azmeena Hasham

deletions and updates, and supports order statistics queries involving predetermined ranks.

An order statistics tree is organized as a structure comprising of an m -dimensional vector V and $m+1$ MinMax heaps $\{H_0, H_1, \dots, H_m\}$. An element $v[i]$ is the i^{th} smallest integer drawn from a linearly ordered set of size m . A heap H_i contains items having keys whose ranks lie between $v[i]$ and $v[i+1]$ (where $v[0] = -\infty$ and $v[m+1] = +\infty$).

An order statistics tree on n items can be constructed in linear time and allows the following operations: i) find an item whose key has rank K_i (rank query), in constant time; ii) find all items whose keys have ranks K_i to K_{i+m} (range query), in $O(m)$ time; iii) delete an item whose key has rank K_i , in $O(\log n)$ time; and iv) insert a new item into the tree, in $O(\log n)$ time.

The data structure provides an efficient solution to the dynamically changing file problem known as the *course-marks problem*. The problem typically involves maintaining a list of students enrolled in a course and their grades. The order statistics tree can be used to support a predefined set of operations on the file. The advantage of the order statistics tree over tabular and binary search tree organizations is that it can be implemented using either implicit or dynamic representations and gives improved worst case performance.

2.3. Priority Queue Structures

Several priority queue implementations are based on min-ordered trees. In this section, we briefly review these data structures. We assume that each tree is accessible by a pointer to the root.

2.3.1. Leftist Tree

A *leftist tree* [Cran72, Knut73] is a linked binary tree with a key and distance associated with every internal node. A leaf or external node is represented by a null pointer and by convention, its key value is $+\infty$ and its distance value is 0. Then a *leftist tree* is

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

A New Class of Priority Queue Organizations

submitted by Azmeena Hasham, B Sc.
in partial fulfilment of the requirements for
the degree of Master of Computer Science

Jörg-Rüdiger Sack
Thesis Supervisor

J. R. Pugh

Chairman, School of Computer Science

Carleton University
Ottawa, Ontario

September 17/86
Date

ABSTRACT

A new organization for a set of keys, called *minmax-ordering*, allows both the minimum and maximum keys to be found easily. This thesis explores some theoretical and practical implications of minmax-ordering. Recently, an efficient data structure based on minmax-ordering was proposed to implement a double-ended priority queue. The structure is known as the MinMax heap. Lower bounds are derived for the operations i) to construct a MinMax heap; ii) to convert a MinMax heap into a MaxMin heap; and iii) to merge two MinMax heaps into a MinMax heap. It is found that MinMax heaps are not sublinearly mergeable. Improved upper bounds for the convert and merge operations are also derived.

Another minmax-ordered structure, called a MinMax leftist tree, is introduced. The structure allows efficient insertions, but costly deletions in the worst case. The complexity of the deletion algorithm is dependent on the height of the tree.

Finally, a new structure which uses minmax-ordering is presented to implement a mergeable double-ended priority queue.

ACKNOWLEDGEMENTS

My greatest debt is to my supervisor, Professor Jorg-Rudiger Sack. This thesis would not have been possible without his unstinted support, insightful observations and enthusiasm. My sincere thanks to him.

A note of special thanks goes to Professor Michael Atkinson for his keen interest and encouragement; to Dr. Thomas Strothotte who offered many helpful suggestions; and to Professor David Avis for valuable discussions. I would also like to thank my dear friends at Carleton University, especially Hin Wai Chang and Yanni Jew, for their criticisms on preliminary versions of this thesis.

This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada under Grant No. A0392.

TABLE OF CONTENTS

Page

ACCEPTANCE SHEET	ii
ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
TABLE OF CONTENTS	v
LIST OF TABLES	viii
LIST OF FIGURES	ix

<i>Chapter</i>	1. INTRODUCTION	
	1.1. Priority Queues and Double-Ended Priority Queues	1
	1.2. MinMax-Ordering	3
	1.3. Lower Bounds and Upper Bounds	7
	1.4. Outline of Thesis and Summary of Results	9
 <i>Chapter</i>	 2. DOUBLE-ENDED PRIORITY QUEUE AND PRIORITY QUEUE STRUCTURES	
	2.1. Double-Ended Priority Queue Structures	11
	2.1.1. Sorted and Unsorted Lists	11
	2.1.2. Williams' Twin Heap	12
	2.1.3. P-tree	13
	2.1.4. Van Emde Boas Priority Queue	14
	2.1.5. Search Tree	15
	2.1.6. MinMax Heap	16
	2.2. Double-Ended Priority Queue Applications	21
	2.2.1. External Quicksort	21
	2.2.2. Mmm-heap	22
	2.2.3. Order Statistics Tree	22
	2.3. Priority Queue Structures	23
	2.3.1. Leftist Tree	23
	2.3.2. Binomial Queue	25
	2.3.3. Pagoda	27
	2.3.4. F-Heap	28

	Page
Chapter 3. LOWER BOUNDS FOR THE MINMAX HEAP	
3.1. Model of Computation	30
3.2. Lower Bounds for MinMax Heap Construction	30
3.2.1. An Information Theoretic Lower Bound	31
3.2.2. An Adversary Lower Bound	32
3.3. Converting a MinMax Heap	34
3.3.1. A Lower Bound	34
3.3.2. An Upper Bound	39
3.4. Merging Perfect MinMax Heaps of Equal Size	42
 Chapter 4. ALGORITHMS FOR MERGING MINMAX HEAPS	
4.1. The Merge Operation	46
4.2. Merging Algorithm A: Subheaps	47
4.2.1. Merging Perfect MinMax Heaps of Equal Size	47
4.2.2. Merging Perfect MinMax Heaps of Different Sizes	47
4.2.3. Merging a Perfect into a Non-Perfect MinMax Heap	48
4.2.4. Merging Non-Perfect Heaps	50
4.2.5. Complexity Analysis for Algorithm A	50
4.3. Merging Algorithm B: Heaps in Heaps	51
4.3.1. MinMax Binomial Heap Forest	51
4.3.2. Valid Descriptors	53
4.3.3. Overview of Algorithm B	54
4.3.4. Complexity Analysis for Algorithm B	56
 Chapter 5. MINMAX LEFTIST TREE	
5.1. Definition and Representation	57
5.2. Construction Algorithm	58
5.3. Insertion Algorithm	58
5.4. Conversion Algorithm	64
5.5. Merging Algorithm	67
5.6. Deletion Algorithm	69

	Page
Chapter 6. MERGEABLE DOUBLE-ENDED PRIORITY QUEUES	
6.1. Converting MinMax-Ordered Trees	71
6.1.1. A MinMax-Ordered Tree with Unbounded Fanout	71
6.1.2. A MinMax-Ordered Tree with Bounded Fanout	71
6.1.3. A MinMax-Ordered Binary Tree of Given Shape	72
6.2. MinMax-Ordered Double-Ended Priority Queues	72
6.2.1. MinMax Binomial Queues	73
6.2.2. MinMax Pagodas	73
6.2.3. MinMax Fibonacci Heaps	73
6.3. Mergeable Min-Ordered and Max-Ordered Queues	73
6.3.1. Williams' Twin Heap	74
6.3.2. Min and Max Heaps	77
6.3.3. Min and Max Binomial Queues	78
6.4. A New Mergeable Double-Ended Priority Queue	79
6.4.1. A Review of Binomial Heaps	79
6.4.2. MinMax-Ordered Queue	80
6.4.3. Operations on the MinMax-Ordered Queue	80
6.4.4. Implementing the MinMax-Ordered Queue	82
 Chapter 7. CONCLUSIONS	
7.1. Summary of Results	84
7.2. Further Research	85
 REFERENCES	 87

LIST OF TABLES

Table	Description	Page
2.1	Summary of Double-Ended Priority Queue Structures	20
2.2	Worst Case Complexities for Leftist Trees	25
2.3	Worst Case Complexities for Binomial Queues	26
2.4	Average and Worst Case Complexities for Pagodas	28
2.5	Amortized Complexities for Fibonacci Heaps	29
6.1	Worst Case Complexities for MinMax-Ordered Queues	83

LIST OF FIGURES

Figure	Description	Page
1.1	Examples of Heaps	4
1.2	An Example of a MinMax-Ordered Tree	5
1.3	Transformation of Problems	9
2.1	An Example of a MinMax Heap	17
2.2	An Example of a Leftist Tree	24
2.3	An Example of a Min-Ordered B_3 Tree	25
2.4	A Forest for a Binomial Queue on 11 Nodes	26
2.5	An Example of a Pagoda	27
3.1	Labelling a MinMax-Ordered Tree	32
3.2	Labelling a MinMax Heap for Conversion	36
3.3	Hasse Diagram for Converted Heap	37
3.4	Merging Perfect MinMax Heaps of Equal Size	44
4.1	Merging MinMax Heaps of Different Sizes	48
4.2	Merging a Perfect 3-Heap into a Non-Perfect 22-Heap	49
4.3	A Binomial Heap	51
4.4	BHF corresponding to the 25-Heap	53
5.1	An Example of a MinMax Leftist Tree	58
5.2	Linking Right Paths Circularly	63
5.3	Converting a MinMax Binary Tree	64
5.4	A Worst Case for the Conversion Algorithm	67
5.5	A Worst Case for the Deletion Algorithm	70

Chapter 1

INTRODUCTION

1.1. Priority Queues and Double-Ended Priority Queues

A queue or waiting line is encountered frequently whenever demands occur in production, transportation, communication, computers or other service systems. The order in which items are removed from the queue may be determined by their time of arrival (first-in first-out or last-in first-out), randomly or according to a priority. The last case illustrates an important and widely used concept in computer science, namely that of a *priority queue*.

An obvious application of the priority queue is in task scheduling problems. In particular, tasks whose urgency is far greater than that of other tasks must be completed first; a first come, first served basis is not always desirable. Hence, tasks must be assigned priorities and those with highest priorities are executed first.

In computer applications of the priority queue, an item is assigned a priority by associating a key with it. The key reflects the relative ability of the item to get out of the queue quickly. The item with the smallest key has the highest priority and is the first to be removed from the queue. For example, if items consist of edges of a graph and the key of each edge is determined by the length of the edge, then shortest edges are successively removed. Formally, a priority queue is a data type for maintaining a collection of items,

each having an associated key from a linearly ordered set, such that the item with the smallest key is the one that is removed next from the queue. If Q is a priority queue and x is an item containing a key, then the two characteristic operations of the priority queue are defined as:

Insert (x, Q) add item x to Q

DeleteMin (Q) remove the item containing the smallest key from Q and return the removed item

These actions are referred to informally as *insertion* and *deletion*.

A priority queue is *stable* if items with equal keys are deleted first-in first-out.

A *mergeable priority queue* is a priority queue with the additional property that two disjoint queues can be combined efficiently into a single queue. The operation

Merge (P, Q) return a queue containing all items in P and Q . The operation destroys P and Q

is defined when P and Q are mergeable priority queues; this operation is informally known as *merging* (or *melding*) P and Q . Of course, any pair of priority queues can be merged using repeated applications of Insert, but we reserve the qualification "mergeable" for those priority queues which can be merged quickly. In other words, merging should not require examining a positive fraction of the items in the queue in the worst case.

Finally, it may be more efficient to construct a priority queue from a fixed set S of items without using repeated insertions. The operation

Construct(S) return a new queue containing all items in S

is informally known as *construction*.

Certain applications need a more powerful data type than the priority queue. One such application is an external sorting routine, called *external quicksort*. Briefly, the routine maintains a number of items in memory which are read from both ends of an unsorted file, and writes items with the smallest and largest keys to the file. The routine requires that the following operations be performed repeatedly and efficiently: insertion of an item containing a key into a queue; deletion of an item with the minimum key from the queue; and deletion of an item with the maximum key from the queue. The data type which supports these operations is known as a *double-ended priority queue* (or *priority deque*).

1.2. MinMax-Ordering

A tree is said to be *min-ordered* if the key of every node is smaller than or equal to the keys of its descendant nodes (if any). The reverse order relation holds in a tree which is *max-ordered*. Most implementations of the priority queue are either min-ordered or max-ordered trees in which the degrees (number of children) of all internal nodes are bounded by some function.

Example 1.1. A well-known priority queue structure is the heap [Will64]. A heap is a binary tree with the *heap-shape*[†] property, that is, all leaves on the last level are as far to the left as possible, and all internal nodes (except possibly the last one) have out-degree 2; in addition, it is either min-ordered or max-ordered. A heap that is min-ordered is called a

[†] As a result of the heap-shape, in a heap on n nodes, no node is at a distance of more than $\log_2 n$ from the root. Furthermore, an array implementation is suitable for the heap. (All logarithms in this thesis are to base 2.)

Min heap and similarly, one that is max-ordered is called a Max heap. Figure 1.1 illustrates a Min heap and Max heap. ♦

Figure 1.1. Examples of Heaps

Consequently, in a priority queue implemented as a min-ordered tree, it is easy to find the minimum key since it is located at the root. However, finding the maximum key involves a linear number of steps in the worst case since the key of each leaf must be examined. In order to find both the minimum and maximum keys efficiently in a tree, it is necessary to establish an alternative order relation between a node and its descendants.

Definition. A tree is *minmax-ordered* if the following properties hold:

- i) keys on even levels are smaller than or equal to keys of all descendant nodes; and
- ii) keys on odd levels are larger than or equal to keys of all descendant nodes.

(The level of a node gives the length of the path from the node to the root [Knut73].)

A tree structure that is minmax-ordered is called a MinMax structure. From now on, we will assume that an item contains only a key, unless otherwise stated, and use the terms item and key interchangeably whenever the meaning is clear from the context.

Example 1.2. It is clear from the example of Figure 1.2 that in a tree that is minmax-ordered, the minimum key is at the root and the maximum key is the largest of the children of the root. ♦

Figure 1.2. An Example of a MinMax-Ordered Tree

The strategy of alternating between min levels and max levels in a minmax-ordered tree is a general one and can be applied to all min-ordered trees. In particular, for a tree in which the degree of all internal nodes is bounded by some small constant, both the minimum and maximum keys can be found efficiently. The strategy is also applicable to those min-ordered trees in which the degree of an internal node is bounded by some function of the size of the tree, with a corresponding loss of efficiency for finding the minimum and maximum keys.

We can further generalize on the minmax-ordering. Instead of alternating min levels and max levels in the tree, we can alternate between an α number of min levels and β number of max levels.

The minmax-ordering strategy was successfully applied to heaps by Atkinson et. al. [ASSS84] to obtain an elegant double-ended priority queue structure. The following double-ended priority queue operations are defined for a MinMax queue Q and an item x :

FindMin (Q)	find the item containing the smallest key in Q and return the item
FindMax (Q)	find the item containing the largest key in Q and return the item
Insert (x, Q)	add item x to Q
DeleteMin (Q)	remove the item containing the smallest key from Q and return the removed item
DeleteMax (Q)	remove the item containing the largest key from Q and return the removed item

Additionally, we may want to be able to construct a queue from a fixed set S of items,

Construct (S)	return a new queue containing all items in S
---------------	--

or merge two MinMax queues P and Q into a MinMax queue,

Merge (P, Q)	return a queue containing all items in P and Q. The operation destroys P and Q
--------------	--

or split a MinMax queue P into MinMax queues P and Q of sizes $n - m$ and m respectively,

Split (P, Q, m)	add m items from P to a new queue Q and return P and Q
-----------------	--

or convert a MinMax queue Q into a MaxMin queue,

Convert (Q)	reorder items in Q to obtain a MaxMin queue and return it. The operation destroys Q
-------------	---

For the remainder of this thesis, a ~~deletion~~ will refer to the operations DeleteMin and DeleteMax.

1.3. Lower Bounds and Upper Bounds

A fundamental question in the design of any algorithm which solves a given problem concerns the intrinsic difficulty of the problem. Knowing how fast a problem can be solved, we can evaluate an existing algorithm for the problem, or determine if a fast algorithm for the problem can be found.

Informally speaking, a *lower bound* for a problem places a lower limit on the complexity of all algorithms which solve the problem, that is, no algorithm runs faster than the lower bound. More precisely however, a lower bound is stated within a *model of computation* that specifies permissible operations and a measure of complexity, for example, the operations to be counted. A lower bound result applies to only those algorithms which belong to the class of algorithms described by the model of computation.

The complexity of the best algorithm for a problem gives an *upper bound* for the problem. If the complexity of the algorithm is the same as the lower bound, the algorithm is *optimal* and the lower bound is said to be *right*. It may be that a lower bound is so small that no algorithm has been found which runs within that time limit; in this case, an improved lower bound is conceivable.

The remaining section provides a theoretical framework for the derivation of lower bounds for comparison-based problems. One technique for deriving lower bounds involves constructing a *decision tree* for an input of a given size. Each interior node in the tree is labelled by a comparison; the descendants of any node represent comparisons consistent with the outcome of the comparison made at the node. For example, suppose a node denotes a comparison between two keys such as "is $K_i < K_j$?". Then the subtrees of the node represent comparisons based on the orderings that $K_i < K_j$ and $K_i \geq K_j$. The objective is to minimize the number of comparisons needed to find a solution and so, redundant comparisons are not permitted. The leaves correspond to all possible solutions

that are obtained based on outcomes of comparisons made along any path in the tree. The height of the decision tree gives a lower bound on the number of comparisons required in the worst case, and is called an *information theoretic lower bound*.

Another technique for deriving lower bounds employs an *adversary argument*. The idea is to construct an adversary which determines the outcome of each comparison made by an algorithm. The adversary forces the algorithm to make a rather large number of comparisons in order to find a solution. The adversary can be used to show that if enough comparisons are not made by the algorithm, an input can be arranged so that an incorrect solution is obtained. In general, adversary methods give better lower bounds than information theoretic methods.

It is sometimes possible to relate the difficulty of two problems by use of a technique of *transformation of problems*. Specifically, suppose problem A is transformed into an input for problem B, problem B is solved, and the output of problem B is transformed into a solution for problem A. If problem B can be solved in $O(T_B)$ time and the transformations take $O(\tau)$ time, then problem A can be solved in $O(T_B) + O(\tau)$ time.

Transformations give us a method for solving problem A. On the other hand, if the lower bound for problem A is known, this method can be used to obtain a lower bound for problem B. If problem A requires at least $\Omega(T_A)$ time and the transformations take $O(\tau)$ time, then the time needed to solve problem B is at least $\Omega(T_A) - O(\tau)$. If any algorithm to solve problem B beat this bound, we would have a method which solves problem A and runs faster than its lower bound - a contradiction. Figure 1.3 shows the relationship between lower bounds and upper bounds for problems A and B.

If problem A is transformable to problem B and vice versa, the two problems are said to be *equivalent*. The lower bounds for both problems, neglecting the overhead of transformations, are the same.

Figure 1.3. Transformation of Problems

1.4. Outline of Thesis and Summary of Results

The minmax-ordering, when applied to the traditional heap, gives an efficient double-ended priority queue structure [ASSS84]. The structure, called a MinMax heap, can be implemented using arrays or pointers. While it is inefficient to merge array implementations of the MinMax heap due to the number of data movements involved, the question of whether a pointer-based implementation of the MinMax heap is mergeable or not remains unsolved. For a long time, merging Min heaps was also considered a hard problem. However, two new algorithms were recently proposed which merge Min heaps of sizes m and n using at most $O(\log m \log n)$ comparisons [SaSt85, StSa85].

A natural extension to the work of [ASSS84] is to apply the minmax-ordering to other min-ordered priority queue structures. Our emphasis is on developing algorithms to perform the double-ended priority queue operations that are asymptotically efficient in the worst case.

Chapter 2 surveys various representations of the double-ended priority queue that have appeared in the literature, and describes a few of its varied applications. A brief review of min-ordered implementations of the priority queue is also given.

Chapter 3 presents lower bounds on the number of comparisons required in the worst case to construct, convert and merge MinMax heaps. An information theoretic

argument establishes that $1.3644 \dots n + O(\log n)$ comparisons are necessary to construct a MinMax heap on n keys. An adversary-based argument improves this lower bound to approximately $3n/2$ comparisons. A lower bound of approximately $n/6$ is obtained for the number of comparisons required to convert a MinMax heap into a MaxMin heap. It is shown via transformation of problems that the lower bound for conversion can be applied to the problem of merging two equal-sized perfect MinMax heaps, each on $n/2$ keys, into a MinMax heap on n keys. The lower bound for merging enables us to settle the question of mergeability for the MinMax heap. An upper bound to convert a MinMax heap into a MaxMin heap is also established; it is shown that $n/3$ comparisons are sufficient for conversion.

Two new algorithms for merging MinMax heaps are presented in chapter 4. The algorithms improve the upper bound for the merging problem; two MinMax heaps of sizes m and n ($m \leq n$) can be merged in $O(m + \log m \log n)$ time.

In chapter 5, the minmax-ordering is applied to another priority queue representation, called the leftist tree. Algorithms to construct, insert, convert, merge and delete are developed for the new data structure, called the MinMax leftist tree. The MinMax leftist tree can be constructed in linear time. An insertion can be performed in logarithmic time. However, algorithms to perform the convert, merge and delete operations are costly in the worst case as their execution time complexities are dependent on the height of the tree.

Chapter 6 discusses the merging problem for minmax-ordered data structures and introduces a new double-ended priority queue, called the minmax-ordered queue. Minmax-ordered queues of size n can be merged and split efficiently in $O(\log^2 n)$ time.

Finally, we present our conclusions and list problems for further research in chapter 7.

Chapter 2

DOUBLE-ENDED PRIORITY QUEUE AND PRIORITY QUEUE STRUCTURES

2.1. Double-Ended Priority Queue Structures

A variety of data structures for implementing the double-ended priority queue appear in the literature. In this section, we describe these data structures and the associated operations.

2.1.1. Sorted and Unsorted Lists

A *sorted doubly linked list* is one of the simplest double-ended priority queue organizations. Items are deleted from either end of the list. An insertion is done after a sequential or binary search finds the correct location for the new key.

A sorted list is in general inefficient for insertions. However, it may be a good choice in the following situations: i) very few items in the list; ii) special distributions which produce insertions near the beginning or end of the list; or iii) no insertions at all in which case all items are available and sorted before any deletion is done.

An *unsorted linked list*, at the other extreme, provides very easy insertion of items but costly deletions. An unsorted list may be a good choice if there are very few deletions to be performed. A pointer-based implementation of an unsorted list gives a mergeable double-ended priority queue since two lists can be merged in constant time.

One way to reduce the time for insertions and deletions is to reduce the length of the list to be scanned. Some methods involving the use of indices which guarantee good time complexity in the average case and worst case are discussed in [NeTe78].

2.1.2. Williams' Twin Heap

A min-ordered heap, or Min heap, allows efficient retrieval of the minimum key while a max-ordered heap, or Max heap, allows efficient retrieval of the maximum key. Williams' proposed a scheme [Knut73, pp. 619] based on the Min heap and Max heap so that the minimum and maximum keys could be found easily in the same structure. Items are divided into two sets of equal sizes from which a Min heap A and Max heap B are constructed. Suppose heaps A and B are implemented as arrays. The following property always holds for keys of A and B:

$$a[i] \leq b[i], \quad 1 \leq i \leq n/2. \quad (1)$$

This arrangement contains an even number of items; to handle an odd number of items, we simply store one item in a temporary location. It follows by transitivity that (1) holds for all nodes if and only if it holds for all the leaves.

One implementation of the construct operation for the *twin heap* finds the upper median, and constructs a Min heap on keys that are smaller than the median and a Max heap on the remaining keys. The median-find algorithm for n keys requires $3n$ comparisons [SPP76]. Each heap is constructed using at most n comparisons. So, the total number of comparisons performed by the construction algorithm is $5n$.

The insertion and deletion algorithms are similar to those for traditional heaps. Items are inserted in pairs since A and B must be of the same size. The smaller key is inserted into the Min heap and the larger key into the Max heap. For insertion, the position of the new item in the heap is found by searching from the last leaf up toward the root. If (1)

(1) does not hold after insertion, the last leaves of both heaps are swapped and one heap is reordered for a second time.

A deletion is accomplished by removing a root and promoting items while searching down from the root for the new position of the last leaf. In this way, the last leaf "trickles down" towards the leaves to its correct position. If the last leaf trickles down all the way to the bottom of the heap and (1) does not hold, corresponding leaves in both heaps are swapped and the larger heap (whose root was not deleted) is reordered. Finally, since A and B must be of the same size, the deletion is followed by an insertion if an extra item is available, or the last leaf of the larger heap is moved to a temporary location.

The twin heap has the advantage that it can be stored *implicitly* [MuSu80] in an array implementation. However, the maximum size of the heap must be known in advance and contiguous storage space must be available. The twin heap is not stable. In chapter 6, we will show that the structure is mergeable.

2.1.3. P-tree

A *p-tree* (or *priority tree*) [JoDa75] is a binary tree with the following ordering constraint: it is either an empty or sorted sequence of nodes (the so-called left path), such that with each node of the left path except the last one, there is associated a *p-tree* (the so-called right subtree). The nodes of the right subtree associated with a node x on the left path, are ranked between x and the left child of x .

Consequently, the minimum and maximum keys are found on the left path, at the root and left leaf respectively. Furthermore, each node in the *p-tree* with a right child also has a left child. The left leaf has no right descendants.

For insertion, nodes on the left path are scanned to determine if a new item should be inserted into a right subtree. If the new item contains the maximum key, it becomes the

left leaf. If the new item contains the minimum key, it becomes the root of the p-tree. The expected number of comparisons required to insert an item containing a random key into a random p-tree on n nodes is $O(\log^2 n)$.

Items containing the minimum and maximum keys can be deleted from a p-tree without performing any comparisons. If the root containing the minimum key is removed, all right path nodes are promoted one place towards the root and the last left subtree, if nonempty, is appended to the right path. If the left leaf containing the maximum key is removed, the right subtree of its parent, if nonempty, is appended to the left path.

Although the worst case number of comparisons for insertion is linear in the size of the p-tree, deletion requires only pointer manipulations. In addition, the p-tree is stable. However, it is not mergeable since in the worst case the operation is equivalent to merging two sorted linear lists.

2.1.4. Van Emde Boas Priority Queue

The *van Emde Boas data structure* [BKZ77], as it is often referred to, supports a large set of operations, including all the operations of the double-ended priority queue. The keys in the data structure are selected from a fixed universe U of keys consisting of the integers $1, \dots, N$. Each of the double-ended priority queue operations can be performed in worst case processing time of $O(\log \log N)$.

The queue is implemented as a structure composed of a queue of queues. Assume that N is not prime and $N = u \cdot v$. Decompose the universe $\{1, \dots, N\}$ into subsets V_i , each of size v , and $V_i = \{(i-1) \cdot v + 1, \dots, i \cdot v\}$, $i = 1, \dots, u$. A subset S of U is completely determined by its intersections $S_i = S \cap V_i$. In other words, keys in S are represented by "marking" corresponding members in V_i and thus obtaining sets S_i . Consider as well the set T of the indices of the nonempty sets S_i , i.e. $T = \{i \mid S_i \text{ is nonempty}\}$.

nonempty}. The double-ended priority queue operations are performed by manipulating the set T and some set S_i . To insert a new key into the queue for example, first insert the key into the set S_i to which it should belong and next insert its index into T if S_i was empty before. To delete a key from the queue, delete it from S_i and if S_i becomes empty, delete the corresponding index from T . To find the minimum key, find the minimum index i in T and return the minimum key in the corresponding set S_i . The maximum key is found in a similar fashion.

The queue of queues which supports T is called the top of the queue and the queues which support S_i are called bottom queues. Clearly, we need u bottom queues of size v and a single top queue of size u . The sizes of the top queue and bottom queues are chosen so that the data structures which support them can be manipulated in $O(\log \log N)$ worst case time per operation and require at most $O(N)$ storage space. Suitable representations are described in [Boas77, Gonn84].

The van Emde Boas priority queue has several disadvantages. The keys in the queue are limited to subsets of the universe $\{1, \dots, N\}$. Therefore, it cannot be used in situations where the priority queue consists of keys whose values are not known in advance. Secondly, the storage requirements are proportional to the size of the universe N while $n \leq N$ is the actual subset of keys included in the queue. Additionally, special care must be taken to maintain multiple copies of a key in the queue.

2.1.5. Search Tree

A *binary search tree* is a binary tree with the property that for any node x , the keys in the left subtree of x are smaller than the key of x and likewise, keys in the right subtree are larger than the key of x . Any implementation of the binary search tree can be used to represent the double-ended priority queue. The minimum key is located at the leftmost

node and the maximum key at the rightmost node in the tree. A balanced binary search tree such as the *AVL tree* [Knut73] allows insertions and deletions to be performed on a tree on n nodes in $O(\log n)$ time.

Another balanced tree, called the *2-3 tree* [AHU74], can be used to implement a mergeable double-ended priority queue. Every interior node in a 2-3 tree has two or three children, and all leaves appear on the same level. Insertion, deletion and merging can be performed in $O(\log n)$ time on a 2-3 tree on n nodes.

Search trees used as priority queues have the advantage that they are stable. However, non-balanced search trees can degenerate into skewed trees after a sequence of insertions and deletions and exhibit inefficient worst case behaviour. Balanced search trees on n items can be created in $O(n \log n)$ time and although they support all the double-ended priority queue operations, the algorithms tend to be rather involved. The 2-3 tree requires more storage if keys are not contained in internal nodes.

2.1.6. MinMax Heap

A *MinMax heap* [ASSS84] is a binary tree that is heap-shaped and minmax-ordered. The MinMax heap differs from the traditional heap only in the ordering of the keys. The conceptual simplicity of the data structure makes it highly suitable for implementing the double-ended priority queue.

Example 2.1. An example of a MinMax heap on 15 keys is given in Figure 2.1. Note that keys on alternate levels on any path are in sorted ascending or descending order. ♦

Node Format :

KEY	
LEFT	RIGHT

Figure 2.1. An Example of a MinMax Heap

The MinMax heap is constructed bottom-up [Floy64] so that the subtrees of any node are valid MinMax or MaxMin heaps, while the key of the node itself may not necessarily be in its correct position. The key is recursively "trickled down" until the minmax- or maxmin-ordering has been established. The procedure must differentiate between whether a key on a min level or max level is being examined. Algorithm 2.1 describes the TrickleDown procedure.

For deletion, the item is removed and replaced with the last leaf of the heap. The minmax-order is then re-established by applying the TrickleDown procedure.

An insertion involves placing a new item into the first available leaf position and re-establishing the ordering on the path from the new leaf to the root. The first step is to determine whether the new key should be placed at a min level or max level and the item is swapped with its parent if necessary. Then only grandparents along the path to the root need be examined. The new item is swapped with its grandparent until it has "bubbled up" to its correct position.

In fact, since keys on alternate levels on any path in the tree are in sorted order, the position of the new key is found using $O(\log \log n)$ comparisons by performing a binary search on the path from the new leaf to the root. However, the overall complexity of the

Algorithm 2.1. TrickleDown

Input. MinMax or MaxMin heap, *heap*, with an incorrect key at the root

Output. MinMax or MaxMin heap, *heap*

```

procedure TrickleDown (heap)
    if ROOT (heap) is on a min level then
        TrickleDownMin(heap)
    else
        TrickleDownMax(heap)

```

end;

Algorithm 2.1 a). TrickleDownMin

Input. MinMax heap, *heap*, with an incorrect key at the root.

Output. MinMax heap, *heap*.

```

procedure TrickleDownMin (heap)
    if ROOT (heap) has children then
        m := smallest of children and grandchildren (if any) of ROOT (heap)
        if m is a grandchild of ROOT (heap) then
            if KEY (m) < KEY (ROOT (heap)) then
                swap KEY (m) and KEY (ROOT (heap))
            if KEY (m) > KEY (PARENT (m)) then
                swap KEY (m) and KEY (PARENT (m))
            TrickleDownMin(m)
        endif
    else { m is a child of ROOT (heap) }
        if KEY (m) < KEY (ROOT (heap)) then
            swap KEY (m) and KEY (ROOT (heap))
        endif
    end;

```

Procedure TrickleDownMax is similar except that the relational operators are reversed.

algorithm does not change as the number of data movements remain unaltered.

A MinMax heap on n keys is constructed using at most $(7n)/3$ comparisons (cf. Williams' twin heap); the minimum and maximum keys are found in constant time; and the operations of deleting the minimum, deleting the maximum and inserting a new key into the queue are performed in $O(\log n)$ time. The data structure can be stored implicitly if its maximum size is known in advance and contiguous storage space is available. However, it is not stable and we will show that it is not a mergeable queue.

Summary

Table 2.1 summarizes the various double-ended priority queue structures discussed in this section, assumptions about the universal set from which keys are drawn, and the worst case number of key comparisons required to perform the double-ended priority queue operations.

Double-Ended Priority Queue Structure	Type of Universe	Worst case key comparisons to process an operation on sets of size n	
		Insertion	Deletion
Sorted list	Arbitrary ordered set	$O(n)^{\dagger}$	$O(1)$
Unsorted list	Arbitrary ordered set	$O(1)$	$O(n)$
William's twin heap	Arbitrary ordered set	$O(\log \log n)$	$O(\log n)$
P-tree	Arbitrary ordered set	$O(n)$	$O(1)$
Van Emde Boas priority queue	Integers 1 to N	$O(\log \log N)$	$O(\log \log N)$
Unbalanced search tree	Arbitrary ordered set	$O(n)$	$O(n)$
Balanced search tree	Arbitrary ordered set	$O(\log n)$	$O(\log n)$
MinMax heap	Arbitrary ordered set	$O(\log \log n)$	$O(\log n)$

Table 2.1. Summary of Double-Ended Priority Queue Structures

[†] For a pointer-based implementation (extra storage would be required to perform a binary search).

2.2. Double-Ended Priority Queue Applications

The double-ended priority queue supports all the operations of the priority queue and hence all of its diverse applications [Brow77]. In addition, the double-ended priority queue arises in certain problems and we present some of them here to illustrate its varied applications. In particular, a fast double-ended priority queue structure can improve the efficiency of some algorithms.

2.2.1. External Quicksort

An important application of the double-ended priority queue is in *external quicksort* [Gonn84]. External quicksort is an ideal sorting routine for direct access files. The sorting procedure does not require any additional file storage since records are sorted "in place". The basic algorithm is the same as internal quicksort which uses an arbitrarily chosen record as a pivot to split the file into two parts containing records with keys smaller and larger than that of the pivot. Thus the original problem is reduced to two simpler problems, namely to sort the subfile to the left and to sort the subfile to the right of the pivot. The procedure is applied to each subfile until the problem has been solved. With external quicksort however, instead of a single pivot, the procedure keeps several pivots in memory which are updated as the splitting phase progresses. The goal is to produce an even split and place as many records as possible in their final position.

To sort a file, records are read from both ends of the file and these positions may then be written to. The pivots in memory are maintained so that at the end of each pass through an unsorted file, they are written as close to the centre of the file as possible. Therefore, as the splitting phase progresses, either the minimum or maximum pivot is written to the file and replaced by a new record which is read in. The reading of records is done as balanced as possible with respect to the writing positions. A double-ended priority queue used to store the pivots allows both the minimum and maximum to be found and

extracted efficiently. In addition, a new record which is read in can be inserted efficiently into the queue

2.2.2. Mmm-heap

An application of the MinMax heap is a data structure called the *mmm-heap* (or *MinMaxMedian heap*) [ASSS84]. The mmm-heap, in addition to supporting all the operations of the double-ended priority queue, allows successive medians to be found and deleted efficiently.

An mmm-heap on n items is a binary tree with the properties that i) the median of all keys is located at the root; and ii) its left subtree is a MinMax heap H_l of size $\lceil (n-1)/2 \rceil$, containing keys smaller than the median, and the right subtree is a MaxMin Heap H_r of size $\lfloor (n-1)/2 \rfloor$ containing keys larger than the median.

Property ii) implies that we must have $|H_l| = |H_r|$ or $|H_l| = |H_r| + 1$ since we are interested in the upper median. Any insertion or deletion should observe this balance. If the median is deleted, either the maximum item in H_l or the minimum item in H_r replaces the root, depending on which maintains the balance. Thus, operations on the mmm-heap manipulate either the MinMax heap H_l or MaxMin heap H_r and are performed in $O(\log n)$ time.

Other structures including balanced search trees can be used to find and delete the item containing the median of a set of keys. However, the mmm-heap has the advantage that it can be constructed in linear time and stored implicitly.

2.2.3. Order Statistics Tree

Yet another application of the MinMax heap is the composite data structure called the *order statistics tree* [ASSS85]. The data structure maintains a set of items under insertions,

deletions and updates, and supports order statistics queries involving predetermined ranks.

An order statistics tree is organized as a structure comprising of an m -dimensional vector V and $m+1$ MinMax heaps $\{H_0, H_1, \dots, H_m\}$. An element $v[i]$ is the i^{th} smallest integer drawn from a linearly ordered set of size m . A heap H_i contains items having keys whose ranks lie between $v[i]$ and $v[i+1]$ (where $v[0] = -\infty$ and $v[m+1] = +\infty$)

An order statistics tree on n items can be constructed in linear time and allows the following operations: i) find an item whose key has rank K_i (rank query), in constant time; ii) find all items whose keys have ranks K_i to K_{i+m} (range query), in $O(m)$ time; iii) delete an item whose key has rank K_i , in $O(\log n)$ time; and iv) insert a new item into the tree, in $O(\log n)$ time.

The data structure provides an efficient solution to the dynamically changing file problem known as the *course-marks problem*. The problem typically involves maintaining a list of students enrolled in a course and their grades. The order statistics tree can be used to support a predefined set of operations on the file. The advantage of the order statistics tree over tabular and binary search tree organizations is that it can be implemented using either implicit or dynamic representations and gives improved worst case performance.

2.3. Priority Queue Structures

Several priority queue implementations are based on min-ordered trees. In this section, we briefly review these data structures. We assume that each tree is accessible by a pointer to the root.

2.3.1. Leftist Tree

A *leftist tree* [Cran72, Knut73] is a linked binary tree with a key and distance associated with every internal node. A leaf or external node is represented by a null pointer and by convention, its key value is $+\infty$ and its distance value is 0. Then a *leftist tree* is

defined as a min-ordered binary tree in which the distance of an internal node is equal to the length of the shortest path from that node to a leaf; moreover, the shortest path from any node to a leaf is always to the right. Consequently, the shortest path from the root to a leaf is given by the rightmost path in the tree which is of length at most $O(\log n)$ in a tree on n nodes

Example 2.2. Figure 2.2 gives an example of a leftist tree. The binary tree is called *leftist* because it is generally heavier on the left. ♦

Node Format :

KEY	DISTANCE
LEFT	RIGHT

Figure 2.2. An Example of a Leftist Tree

The basic operation on the leftist tree is merging. The merging of two disjoint trees requires examining nodes on the rightmost paths only. Thus, two leftist trees with a total of n nodes can be merged in $O(\log n)$ time. An insertion is accomplished by merging a single node into the tree; a deletion is performed by removing the root and merging its left and right subtrees.

Operation	Time to process an operation on sets of size n
Construct	$O(n)$
Merge	$O(\log n)$
Insert	$O(\log n)$
FindMin	$O(1)$
DeleteMin	$O(\log n)$

Table 2.2. Worst Case Complexities for Leftist Trees

2.3.2. Binomial Queue

A *binomial queue* [Vuil78, Brow77] is represented as a forest of min-ordered B_k trees. A B_k tree is of size 2^k and consists of a root having k descendants: one B_0 , one B_1 , ..., one B_{k-1} tree.

Example 2.3. Figure 2.3 illustrates a min-ordered B_3 tree. ♦

Figure 2.3. An Example of a Min-Ordered B_3 Tree

Two equal sized B_k trees are merged into a single B_{k+1} tree using one comparison. The B_k tree with the larger root is made an offspring of the other B_k tree. Consequently, a

min-ordered B_k tree is constructed using $2^k - 1$ comparisons by repeated pairwise merging. This construction is optimal.

The B_k trees in a forest for a binomial queue of size n correspond to 1's in the binary decomposition of n . Therefore, a forest contains at most $O(\log n)$ B_k trees

Example 2.4. A binomial queue on $n = 11$ nodes is represented by $\{B_3, B_1, B_0\}$ and corresponds to the binary decomposition of 11 which is 1011_2 . Figure 2.4 illustrates the associated forest. ♦

Example 2.4. A Forest for a Binomial Queue on 11 Nodes

Operation	Time to process an operation on sets of size n
Construct	$O(n)$
Merge	$O(\log n)$
Insert	$O(\log n)$
FindMin	$O(\log n)$
DeleteMin	$O(\log n)$

Table 2.3. Worst Case Complexities for Binomial Queues

Two binomial queues are merged into a single queue by merging all equal sized B_k

trees in a process identical to binary addition. An insertion is accomplished by merging a new item into the queue. The minimum key of a binomial queue is found by inspecting the roots of all its B_k trees. A root is deleted by removing it from its B_k tree and merging its subtrees with the other B_k trees in the forest.

2.3.3. Pagoda

A *pagoda* [FVV78] is a min-ordered binary tree with the following pointer organization: i) the left and right links of the root point to the leftmost and rightmost nodes respectively; ii) the right link of a right descendant points to its parent and the left link to its leftmost descendant; and iii) the left link of a left descendant points to its parent and the right link to its rightmost descendant.

Example 2.5. An example of a pagoda showing the pointer organization is illustrated in Figure 2.5. Note that the left and right links of a pagoda on a single node point to the node itself. ♦

Figure 2.5. An Example of a Pagoda

The basic operation on a pagoda is merging two disjoint pagodas. Merging pagodas is done bottom-up with the rightmost path of one being merged with the leftmost path of the other. An insertion is achieved by merging a pagoda on the new node into the original pagoda; a deletion is done by removing the root and merging its left and right subtrees.

Operation	Time to process an operation on sets of size n	
	Average case	Worst case
Construct	$2n - 2H_n$	$2n - 3$
Merge	$4H_n - 2H_{2n}$	$2n - 1$
Insert	$2 - \frac{2}{n}$	$n - 1$
FindMin	1	1
DeleteMin	$2H_n - 4 + \frac{2}{n}$	$n - 2$

$$H_n = 1 + 1/2 + \dots + 1/n$$

Table 2.4. Average and Worst Case Complexities for Pagodas.

2.3.4. F-heap

An *F-heap* (or *Fibonacci heap*) [FrTa84] is represented as a forest of min-ordered trees. The number of trees in the forest is constrained by the way in which the trees are manipulated. Any node in a tree of size n has at most $O(\log n)$ children as a result.

A fundamental operation on the trees is called *linking*: two trees are combined by adding the tree with the smaller root as a new subtree of the larger root (cf. merging B_k trees).

An insertion into an F-heap is achieved by simply adding the new item to the forest of trees. The minimum item is found by keeping track of the tree whose root contains the minimum key. A deletion of a root involves removing it from its tree and adding its

subtrees to the forest. Then all trees whose roots have the same number of children are linked together until there are no more trees in the forest whose roots have the same number of offspring. F-heaps have efficient amortized time complexities

Operation	Time to process an operation on sets of size n
Insert	$O(1)$
FindMin	$O(1)$
DeleteMin	$O(\log n)$

Table 2.5. Amortized Complexities for Fibonacci Heaps

Chapter 3

LOWER BOUNDS FOR THE MINMAX HEAP

3.1. Model of Computation

In this chapter, we present lower bound results for the operations to construct a MinMax heap on a set of keys, to convert a MinMax heap into a MaxMin heap, and to merge two MinMax heaps into a MinMax heap. An upper bound for converting a MinMax heap into a MaxMin heap is also established. Our model of computation is a random access machine or RAM [AHU74] whose memory contains one key per storage location. The basic operation employed is the 2-way branch comparison between the keys. The keys are drawn from a linearly ordered set. The measure of complexity is the number of comparisons required in the worst case to solve a problem. We assume that all keys are distinct so that there are exactly two possible outcomes from any comparison between two different keys.

3.2. Lower Bounds for MinMax Heap Construction

In this section, we establish lower bounds on the number of comparisons necessary, in the worst case, to construct a MinMax heap. First, we present an information theoretic lower bound. Then, we give an improved lower bound based on an adversary lower bound for finding the minimum and maximum of a set of keys.

3.2.1. An Information Theoretic Lower Bound

To derive the information theoretic lower bound for MinMax heap construction, we must find $H(n)$, the number of MinMax heaps on a set of n keys. Since we are considering sets with distinct key values, we may without loss of generality represent any sequence of n keys as a permutation of the numbers $\{1, 2, \dots, n\}$. If the input to a MinMax heap construction algorithm is a random permutation of $\{1, 2, \dots, n\}$, then any one of the $H(n)$ possible heaps is a likely solution. Lemma 3.1 enumerates $H(n)$

Lemma 3.1. The number of ways to label the nodes of a given binary tree with the keys $\{1, 2, \dots, n\}$ so that the tree is minmax-ordered is

$$t_n = \frac{n!}{s_1 s_2 \dots s_n}$$

where s_i is the size (number of nodes) of the subtree rooted at node i .

Proof. In any minmax-ordered tree on n keys, the minimum key must be at the root. The remaining $n - 1$ keys can label the left and right subtrees, of sizes p and q respectively, with $\binom{n-1}{p}$ combinations. The subtrees are maxmin-ordered, but clearly, the number of maxmin-ordered subtrees is equal to the number of minmax-ordered subtrees. So if t_i denotes the number of ways to label a tree of size i so that it is minmax-ordered, then $t_n = \binom{n-1}{p} t_p t_q$. A similar argument shows that there are also t_n ways to label the tree so that it is min-ordered. Furthermore, for min-ordered trees, it has been shown that $t_n = n! / \prod_{i=1}^n s_i$ where s_i is the size of the subtree rooted at node i [Knut73]. ♦

Example 3.1. In the MinMax tree below,

Figure 3.1. Labelling a MinMax-Ordered Tree

the number of ways to label the nodes is $10!/(10 \cdot 5 \cdot 4 \cdot 1 \cdot 3 \cdot 1 \cdot 2 \cdot 1 \cdot 1 \cdot 1) = 9 \cdot 8 \cdot 7 \cdot 6$ ♦

Theorem 3.1. The number of comparisons required in the worst case to construct a MinMax heap on a set of n keys is at least $\log(n!/(s_1 s_2 \cdots s_n)) = 1.3644 \dots n + O(\log n)$ where s_i is the size of the subtree rooted at node i .

Proof. Consider a binary decision tree for an input corresponding to a set of size n . Its leaves are the $H(n)$ possible outputs that can be obtained from a MinMax heap construction algorithm. Each path in the tree represents the comparisons made to arrive at a solution. Therefore, the binary tree of minimum height gives the number of comparisons necessary, in the worst case, to construct a MinMax heap. The information theoretic lower bound follows from the result of [GoMu82]. ♦

3.2.2. An Adversary Lower Bound

We now present an improved lower bound for MinMax heap construction based on the number of comparisons needed to find the minimum and maximum of a set of keys. Lemmas 3.2 [Baas78] and 3.3 [Atki77] use an adversary argument to prove the number of

comparisons needed to find the extrema of a set.

Lemma 3.2. [Baas78] The number of comparisons required in the worst case to find the minimum (or maximum) of a set on n keys is at least $n - 1$.

Proof. Suppose an algorithm uses fewer than $n - 1$ comparisons to determine the minimum. Then there are at least two keys in the set which are not known to be larger than any of the other keys. These keys are candidates for the solution. Arrange the input to the algorithm so that the larger key is output as a solution, thus proving the algorithm false. ♦

Lemma 3.3. [Atki77] The number of comparisons required in the worst case to find the minimum and maximum of a set on n keys is at least $(3n)/2 - 2$ if n is even and at least $(3(n-1))/2$ if n is odd.

Proof. Let the candidates for the minimum and maximum belong to set M_n and set M_x respectively. Then any comparison between keys in $M_n \cap M_x$ causes both M_n and M_x to be reduced by one. Any other comparison reduces the size of either M_n or M_x by at most one. Since initially $M_n \cap M_x$ contains all n keys, at most $\lfloor n/2 \rfloor$ comparisons diminish both M_n and M_x . Further comparisons within M_n and M_x are required to determine the minimum and maximum respectively. So from Lemma 3.2, it follows that the total number of comparisons required is at least $2(n - \lfloor n/2 \rfloor - 1) + \lfloor n/2 \rfloor$. ♦

Theorem 3.2. The number of comparisons required in the worst case to construct a MinMax heap on a set of n keys is at least $(3n)/2 - 3$ if n is even and $(3(n-1))/2 - 1$ if n is odd.

Proof. Clearly, with only one comparison, we can find both the minimum and maximum keys in the MinMax heap. Therefore, any algorithm for constructing the MinMax heap must make at least one less than the number of comparisons needed to find both the minimum and maximum of a set. Otherwise, we would have a method of finding the minimum and maximum of a set of keys using fewer comparisons, thus contradicting Lemma 3.3. ♦

3.3. Converting a MinMax Heap

We noted previously that in a MinMax heap, the maximum key is found using one comparison while in a MaxMin heap, the maximum key is located at the root and therefore, readily available. A trivial lower bound establishes that at least one comparison is needed to convert a MinMax heap into a MaxMin heap. In this section, we give a nontrivial lower bound and an upper bound for converting a MinMax heap into a MaxMin heap.

3.3.1. A Lower Bound

A MinMax heap is *perfect* if all its leaves are on the same level. A perfect MinMax heap of height h has exactly $2^{h+1} - 1$ nodes. We wish to find the number of comparisons needed to convert a perfect MinMax heap into a MaxMin heap. A lower bound for the conversion problem can thus be established.

Consider a perfect MinMax heap of odd height. Let the nodes from min levels and max levels belong to set M_n and set M_x respectively. Then $|M_n| = m$ and $|M_x| = 2m$, that is, there are twice as many nodes on max levels than on min levels. This means that after conversion, at least m nodes which were initially in M_x are now in M_n . Similarly, when the perfect MinMax heap is of even height, then $|M_n| = 2m + 1$ and $|M_x| = m$. So at least $m + 1$ nodes from M_n are in M_x after conversion. Furthermore, note that approximately $3/4$ of all nodes in a perfect heap are at the last two levels. Intuitively, it would seem that

the comparisons performed between keys at the last two levels in the heap during conversion, dominate the total conversion time.

To quantify these comparisons, define a perfect MinMax heap in which all nodes at the last two levels are labelled with distinct key values in the range $(-\infty, +\infty)$. All other nodes are labelled with $-\infty$'s and $+\infty$'s, alternating from level to level. (See Figure 3.2 a)). We shall refer to this embedding of keys in a $-\infty, +\infty$ heap as L .

Denote the set of keys at the last two levels in the heap by S . Assume that the heap is of odd height. Then to find the maximum key in S , it is necessary only to examine keys at the last level in the heap. This is because any key on the second-last level is known to be smaller than two other keys in S . This follows easily from the definition of the minmax-ordering.

Now convert the MinMax heap into a MaxMin heap using any algorithm A . To find the maximum key in S given the MaxMin heap, we would perform comparisons between keys of S located in the heap so that i) if key k is at a max level, then there is no other key from S at a max level on the path from the root to k ; and ii) if key k is at a min level, then there is no other key from S which is at a max level on the path from the root to k , or in the subtree rooted at k . In all other cases, a key k is known to be smaller than at least one other key in S . Therefore, it cannot be a candidate for the maximum.

Say there were x candidates for the maximum initially, and y candidates for the maximum after conversion ($x > y$). To find the maximum of the x keys, convert the MinMax heap into a MaxMin heap using any algorithm A and determine the maximum of the y keys using $y - 1$ comparisons. Then algorithm A must have performed at least $x - y$ comparisons. If fewer comparisons were made by A , we would have a method to find the maximum of x keys using fewer than $x - 1$ comparisons, contradicting Lemma 3.2. To establish a lower bound for converting a perfect MinMax heap into a MaxMin heap, we

- a) Before Conversion. The maximum of all keys *at* the last two level in the heap must be located at one of the highlighted node positions.

- b) After Conversion. The maximum of all keys *from* the last two levels in the heap must be located at one of the highlighted node positions.

Figure 3.2. Labelling a MinMax Heap for Conversion

must minimize $x - y$. In other words, we must maximize y , the number of candidates for the maximum key in the converted heap.

Example 3.2. Figure 3.2 illustrates a perfect MinMax heap of height 5, labelled by L , before and after conversion. The candidates for the maximum of all the keys initially at the last two levels in the heap, are highlighted. In this case, there are 32 candidates for the maximum before conversion, and 21 candidates after conversion. ♦

Figure 3.3. Hasse Diagram for Converted Heap

Earlier, we identified the locations of keys which were candidates for the maximum in the converted heap. In the Hasse diagram associated with the heap, a key k is a candidate for the maximum if nodes on all paths from the maximal node to k are labelled with $+\infty$'s. As an example, see Figure 3.3 which illustrates the Hasse diagram for the MaxMin heap of Figure 3.2 b). This characterization can be used to count the largest

number of candidates possible for the maximum (and minimum) key in a converted heap of odd (and even) height. Thus y is maximized by considering the placements of $+\infty$'s (and $-\infty$'s) in the converted heap.

Lemma 3.4. Consider a perfect MinMax heap of size $n = 2^{h+1} - 1$ labelled by L . Denote the set of keys which are initially at the last two levels in the heap by S . After the MinMax heap has been converted into a MaxMin heap, at most $(n - 1)/3$ comparisons are required to find the minimum key in S if h is even, and at most $(n - 3)/3$ comparisons are required to find the maximum key in S if h is odd.

Proof. Assume h is odd. The $-\infty$'s and $+\infty$'s in the converted heap must be arranged in such a way as to maximize the number of candidates for the maximum key. Arrange the $-\infty$'s and $+\infty$'s so that all candidates occur at the shortest possible distances from the root. A simple counting argument shows that all candidates occur at levels $h - 1$ and h . By inspecting the Hasse diagram associated with the heap, we easily obtain the following: i) a $+\infty$ at a distance of $h - 1$ from the root is moved to a min level at the expense of at least 2 candidates; and ii) a $+\infty$ at a distance less than $h - 1$ from the root is moved to a min level at the expense of at least 3 candidates. Therefore, the configurations with the largest number of candidates for the maximum place $+\infty$'s in the converted heap at alternate levels and as close to the root as possible. It follows that we can have at most $n/3$ candidates for the maximum.

A similar argument is applied to find the minimum key in S if h is even, giving at most $(n + 2)/3$ candidates for the minimum.

Finally, Lemma 3.2 gives the number of comparisons required. ♦

Theorem 3.3. The number of comparisons required in the worst case to convert a perfect MinMax heap of size $n = 2^{h+1} - 1$ into a MaxMin heap is at least $(n - 1)/6$ if h is even, and at least $(n + 3)/6$ if h is odd.

Proof. Assume h is odd. Consider a perfect MinMax heap labelled by L . Denote the set of keys at the last two levels in the heap by S . Then the maximum of all keys in S is found by inspecting keys at the last level. So initially $(n - 1)/2$ comparisons are needed to find the maximum. After the heap has been converted, by Lemma 3.4, at most $(n - 3)/3$ comparisons are needed to find the maximum key in S . Therefore, any algorithm for converting a MinMax heap into a MaxMin heap must perform at least $(n + 3)/6$ comparisons. Otherwise, we would have a method of finding the maximum of $(n + 1)/2$ keys using fewer than $(n - 1)/2$ comparisons, contradicting Lemma 3.2.

A similar argument for finding the minimum key in S shows that at least $(n - 1)/6$ comparisons are needed by any algorithm to convert a perfect MinMax heap when h is even. ♦

Example 3.3. With reference to Figure 3.2, consider the problem of finding the maximum of all the keys initially located at the last two levels in the MinMax heap. There are 32 candidates for the maximum before conversion, and *at most* 21 candidates after conversion. Therefore, any algorithm which converts a perfect MinMax heap of height 5 into a MaxMin heap must make at least 11 comparisons. ♦

3.3.2. An Upper Bound

The following algorithm to convert a MinMax heap into a MaxMin heap reorders the keys in the heap and at the same time maintains the heap-shape. The root is swapped with

its larger child so that the maximum and minimum keys are correctly located with respect to the maxmin-ordering. However, we now have a MaxMin heap as a subtree of the maximum key and two MinMax heaps as subtrees of the minimum key, all of which must be converted. Algorithm 3.1 describes the conversion algorithm.

Algorithm 3.1. Conversion Algorithm for the MinMax Heap

Input. MinMax heap, *heap*.

Output. MinMax heap converted into a MaxMin heap, *heap*.

```

procedure ConvertMinMaxHeap (heap)
  if (heap contains more than one node) then
    m := maximum child of ROOT (heap)
    p := subtree of ROOT (heap) which is not rooted at m
    swap KEY (ROOT (heap)) and KEY (m)
    ConvertMaxMinHeap (p)
    ConvertMinMaxHeap (LEFT (m))
    ConvertMinMaxHeap (RIGHT (m))
  endif
end;

```

Procedure ConvertMaxMinHeap which converts a MaxMin heap is similar except that the relational operators are reversed.

Correctness of Conversion Algorithm for MinMax Heaps

The proof of correctness of Algorithm 3.1 is given by induction on the height h of the MinMax (or MaxMin) heap to be converted. The algorithm is obviously correct for the trivial cases where the MinMax or MaxMin heap to be converted is empty ($h = -1$) or consists of a single node ($h = 0$). The induction hypothesis assumes that all MinMax and

MaxMin heaps of depths up to $h \geq 0$ are correctly converted by the algorithm.

To convert a MinMax heap of height $h + 1$, the key of the larger child of the root is swapped with the key of the root. The algorithm then converts a MaxMin heap of height at most h and two MinMax heaps of heights at most $h - 1$. By the induction hypothesis, all of the heaps are correctly converted. So both subtrees of the root are MinMax heaps. One of the subtrees of the root is simply the converted MaxMin heap. The other is rooted at the minimum key which was originally at the root and its subtrees are the converted MinMax heaps. Furthermore, the key of the root is the maximum key in the heap; hence the tree is a MaxMin heap as it is both heap-shaped and maxmin-ordered. ♦

Theorem 3.4 Algorithm 3.1 converts a MinMax heap on n keys into a MaxMin heap using at most $n/3$ comparisons.

Proof. Let $C(h)$ be the total number of comparisons performed by Algorithm 3.1 to convert a perfect MinMax heap of size $n = 2^{h+1} - 1$. Then

$$C(h) = C(h - 1) + 2C(h - 2) + 1, \text{ for } h \geq 2$$

$$C(0) = 0; C(1) = 1$$

and, in fact

$$C(h) = 2^{h+1}/3 + (-1)^{h+1}/6 - 1/2.$$

So $C(h) = (n - 1)/3$ if h is even and $C(h) = n/3$ if h is odd. This implies that at most $n/3$ comparisons are required to convert a MinMax heap on n keys into a MaxMin heap. (Note that $(7n)/3$ comparisons are required in the worst case to reconstruct a MaxMin heap).

Algorithm 3.1 swaps the keys of the root and one of its children after performing each comparison. Therefore, the number of data movements is at most $2C(h)$. ♦

3.4. Merging Perfect MinMax Heaps of Equal Size

Is the MinMax heap mergeable? An answer to this question might be found if a lower bound is established for the merge operation. In this section, we present a lower bound by considering the problem of merging two perfect MinMax heaps, each on $n/2$ keys, into a MinMax heap on n keys. Our objective is to show that any conversion problem is transformable to a merging problem. Then the lower bound result for the conversion problem which was derived in the previous section can be used to obtain a lower bound for the merging problem.

A MinMax heap is converted into a MaxMin heap via merging as follows. Merge the subtrees of the root, which are MaxMin heaps, into a MaxMin heap. Then insert the root into the merged heap to obtain the solution. Algorithm 3.2 describes this procedure.

Insertion of the root into the MaxMin heap does not require any comparisons. This is because the key of the root is known to be the smallest. A "hole" is created in the MaxMin heap for the root by "pushing down" nodes at alternate levels on the path to the first available leaf position. The push down involves reinserting a node two levels farther down the tree; the leaf position is filled by its parent or grandparent node. The path to the leaf is given by the binary representation of the size of the MinMax heap. The high order bit is ignored; bit 0 stands for "go to left child of node" and bit 1 stands for "go to right child of node".

Algorithm 3.2. Convert MinMax Heap via Merging

Input. MinMax heap, *heap*.

Output. MinMax heap converted into a MaxMin heap.

```

function Convert (heap)
  if heap is empty then return heap
  else
     $t :=$  merge left and right subtrees of ROOT (heap) into a MaxMin heap
    insert ROOT (heap) into  $t$ 
    return  $t$ 
  endif
end;

```

Lemma 3.5. Suppose at least T_C comparisons are required in the worst case to convert a perfect MinMax heap on $n + 1$ keys into a MaxMin heap. Then the number of comparisons required in the worst case to merge two perfect MinMax heaps, each on $n/2$ keys, into a MinMax heap on n keys, is at least T_C .

Proof. We have shown that the conversion problem is transformable to the merging problem. In particular, if Algorithm 3.2 converts a perfect MinMax heap, two perfect MaxMin heaps of equal size are merged together. The solution to the conversion problem is obtained by inserting the root into the merged heap, without any comparisons. Since the lower bound for the conversion problem is T_C comparisons, any algorithm which solves the problem must require at least T_C comparisons. Therefore, the merging problem must require at least T_C comparisons. ♦

In a brief diversion, let us look at how a lower bound for merging perfect equal-sized MinMax heaps can be used to derive a lower bound for a MinMax heap conversion problem. Once again, the method of transformations is employed. Two perfect equal-sized MinMax heaps are merged into a single MinMax heap via conversion in the following way. Remove the last leaf of one of the MinMax heaps to obtain a perfect

heap, a non-perfect heap and an extra node. Let the node become the root of a binary tree whose left subtree is the perfect heap and whose right subtree is the non-perfect heap. The binary tree is heap-shaped, and additionally, maxmin-ordered except at the root. The order is restored by allowing the root to trickle down to its correct position (see section 2.1.6). The resulting MaxMin heap is converted into a MinMax heap to obtain the solution to the merging problem. Algorithm 3.3 describes the merging algorithm.

Example 3.4. Figure 3.4 illustrates the procedure for merging two perfect MinMax heaps of equal size into a heap-shaped binary tree. ♦

Figure 3.4. Merging Perfect MinMax Heaps of Equal Size

Algorithm 3.3. Merge Perfect MinMax Heaps of Equal Size via Conversion

Input. Two perfect MinMax heaps of equal size, p and q .

Output. A MinMax heap containing all items in p and q .

function Merge (p, q)

if p and q are empty **then return** p

else

 ROOT ($tree$) := last leaf of q

 LEFT (ROOT ($tree$)) := p

 RIGHT (ROOT ($tree$)) := q without last leaf

 TrickleDownMax ($tree$)

return Convert($tree$)


```

endif
end;

```

Lemma 3.6. Suppose at least T_m comparisons are required in the worst case to merge two perfect MinMax heaps, each on $n/2$ keys, into a MinMax heap on n keys. Then the number of comparisons required in the worst case to convert a MinMax heap on n keys into a MaxMin heap, is at least $T_m - O(\log n)$

Proof. Algorithm 3.3 combines two perfect equal-sized MinMax heaps into a MaxMin heap using at most $O(\log n)$ comparisons. Converting the MaxMin heap gives the solution to the merging problem. The lower bound for the merging problem is T_m comparisons, so any algorithm which solves this problem requires at least T_m comparisons. Therefore, the conversion problem must require at least $T_m - O(\log n)$ comparisons. ♦

Theorem 3.5. The number of comparisons required in the worst case to merge two perfect MinMax heaps, each on $n/2$ keys, into a MinMax heap on $n = 2^{h+1} - 2$ keys, is at least $n/6$ if h is even, and at least $(n + 4)/6$ if h is odd.

Proof. The proof is easily derived using Theorem 3.3 and Lemma 3.5. ♦

Corollary 3.1. The MinMax heap is not sublinearly mergeable.

Proof. The proof follows from Theorem 3.5. ♦

ALGORITHMS FOR MERGING MINMAX HEAPS

4.1. The Merge Operation

In the previous chapter, we derived a linear time lower bound for the merge operation on MinMax heaps. We will now study algorithms for merging two MinMax heaps into a MinMax heap which achieve this time bound. We define the merge operation on two MinMax heaps, called N-heap and M-heap, of sizes n and m respectively ($n \geq m$). The result of merging these two heaps is a MinMax heap, called (N+M)-heap, containing $n+m$ items from the N-heap and M-heap. To merge the heaps, two methods should be apparent:

- a) Insert the items of the M-heap one by one into the N-heap. This requires at most $O(m \log \log (n+m))$ comparisons and $O(m \log (n+m))$ data movements.
- b) Construct a new heap ignoring the structure in both heaps. This requires $O(n+m)$ comparisons and data movements in the worst case.

We are primarily interested in merging algorithms which take advantage of the structure in the MinMax heap. To this end, we present two sets of algorithms, which we shall refer to as Merging Algorithm A and Merging Algorithm B. Both algorithms perform better than the methods above in almost all cases.

4.2. Merging Algorithm A: Subheaps

An algorithm for merging traditional heaps was recently introduced by Sack and Strothotte [SaSt85]. Their algorithm employs the fact that perfect subheaps of equal sizes can be merged efficiently (i.e. using fewer comparisons). We shall use a similar strategy to obtain a more efficient merging algorithm for MinMax heaps.

4.2.1. Merging Perfect MinMax Heaps of Equal Size

Recall the algorithm for merging two perfect equal-sized MinMax heaps presented in section 3.4. The algorithm extracts the last leaf of one of the heaps and makes the N -heap its left child and the $(N-1)$ -heap its right child. The heap-shaped binary tree obtained in this way is then maxmin-ordered by allowing the root to trickle down towards the leaves. This is necessary since the extracted item may not contain the maximum key. The solution to the merging problem is given by converting the MaxMin heap into a MinMax heap. This is the basic method used in the following sections to develop an algorithm for merging MinMax heaps of arbitrary sizes.

Note that two perfect MinMax heaps of equal size can be combined into a MaxMin heap in logarithmic time in a pointer-based implementation. In an array implementation, at most $O(\log n)$ comparisons and $O(n)$ data movements are needed.

4.2.2. Merging Perfect MinMax Heaps of Different Sizes

The algorithm for merging perfect MinMax heaps of different sizes is most clearly described with reference to Figure 4.1. The algorithm finds the left subheap, K -heap, of size m in the N -heap as illustrated in Figure 4.1 a). Assume for now that the K -heap is minmax-ordered. The M -heap is merged into the K -heap using the algorithm of the previous section. The height of the merged heap is exactly one more than that of the K -heap and therefore the tree of Figure 4.1 b) is heap-shaped.

Figure 4.1. Merging MinMax Heaps of Different Sizes

However, the $(N+M)$ -tree is not necessarily minmax-ordered. The M-heap may contain keys which are larger than keys on max levels in the N-heap, and smaller than keys on min levels in the N-heap. Thus the minmax-ordering may be violated when the result of merging the K-heap and M-heap is placed into the N-heap. To ensure that this situation does not arise, each key on the path from N-heap's root to K-heap's root is compared with the minimum or maximum in the M-heap as appropriate. If any key on a min level in the N-heap is larger than the minimum in the M-heap, the two keys are swapped. The M-heap is then reordered by trickling down the new key. A similar process is applied to keys on max levels in the N-heap. This procedure guarantees the minmax-ordering in the $(N+M)$ -heap and is called *walk down*.

If the K-heap is maxmin-ordered, it is converted *before* it is merged with the M-heap. In this case, the two perfect equal-sized MinMax heaps are efficiently combined into a MaxMin heap.

4.2.3. Merging a Perfect into a Non-Perfect MinMax Heap

Now consider the problem of merging a non-perfect N-heap and a perfect M-heap into an $(N+M)$ -heap. Define *slots* to be those new leaf positions in the N-heap which get occupied as a result of the merge operation. A node p *covers* a group of slots if all slots

are descendants of p . The *lowest common ancestor* (LCA) of a group of slots covers all slots in a heap, and the subtree rooted at LCA, called L-heap, is of minimum height. In Figure 4.2 for example, the LCA is the root.

Figure 4.2. Merging a Perfect 3-Heap into a Non-Perfect 22-Heap

The merging algorithm locates the LCA of all m slots in the N-heap, and applies a walk down from the root to the LCA. Assume that the L-heap is minmax-ordered. If the L-heap is of the same height as the M-heap, then the algorithm of section 4.2.1 is used to merge the L-heap and M-heap. Otherwise, let h be the height of the M-heap. Two subtrees rooted at nodes p_l and p_r in the N-heap are found so that i) the subtree rooted at p_l (called p_l -heap) has height $h+1$; ii) the subtree rooted at p_r (called p_r -heap) has height h ; and iii) p_l and p_r together cover all slots. The existence of p_l and p_r is guaranteed in [SaSt85].

The algorithm now applies a *double walk down* from LCA to p_l , updating the M-heap and p_r -heap; and from LCA to p_r , updating the M-heap and p_l -heap. The p_r -heap and M-heap are then merged using a procedure analogous to that described in section 4.2.1. However, this time it is the last leaf of the p_l -heap that is extracted and made the root. The result is a perfect heap rooted at p_r of height $h+1$. Exchanging the heaps rooted at p_l and p_r completes the merge procedure.

4.2.4. Merging Non-Perfect Heaps

The reader will note that a non-perfect MinMax heap is generally composed of a root, a perfect MaxMin heap and a non-perfect MaxMin heap. In order to obtain two distinct MaxMin heaps from a MinMax heap, the root is removed and inserted into the non-perfect MaxMin heap. This process redefines the split operation of chapter 1.

An algorithm for merging non-perfect heaps can now be readily described. The smaller non-perfect heap is repeatedly split as above. Each perfect heap so obtained is merged into the larger heap using the strategies of previous sections.

A more efficient algorithm initially finds the LCA in the larger N-heap and performs a walk down from the root to the LCA. As before, two nodes p_l and p_r are then found which together cover all slots. The smaller M-heap is now split and its perfect piece is merged into the p_l -heap or p_r -heap, depending on which has enough slots. The procedure is recursively applied to the non-perfect heap, and p_l -heap and p_r -heap. As always, we must keep track of the (minmax- or maxmin-) ordering of the heaps to be merged, and of the ordering desired. If both heaps are not ordered the same way, one of them is converted so that the heaps can be combined efficiently. Note that the height of the heaps to be merged is reduced by at least one at each recursive step.

4.2.5. Complexity Analysis for Algorithm A

We analyse the time taken to merge a non-perfect M-heap into a non-perfect N-heap. The path from the root to the LCA is of length at most $O(\log(n/m))$ and each trickle down in the M-heap takes at most $O(\log m)$ time. Therefore, the time required to perform the walk down to the LCA is at most $O(\log(n/m) \log m)$. The remaining steps are independent of n and are performed recursively. Splitting the M-heap requires only data movements and can be performed in at most $O(\log m)$ time. To merge perfect heaps of size m requires

a trickle down and a conversion (see section 3.3.2), which can be performed in at most $O(m + \log m)$ time. Since the height of the M-heap decreases by at least one at each recursive step, these steps are repeated $O(\log m)$ times. So the total time taken by the recursion is at most $\sum_{i=0, 1, \dots, \log m} O(m/2^i + \log(m/2^i)) \leq O(m + \log^2 m)$.

Thus, in the worst case, $O(m + \log n \log m)$ comparisons and data movements are performed by Algorithm A.

4.3. Merging Algorithm B: Heaps in Heaps

A unifying approach to merging traditional heaps, presented by Strothotte and Sack [StSa85], shows that heaps are in fact composed of simpler structures, called binomial heaps. Equal-sized binomial heaps have the appealing property that they can be merged efficiently. The algorithm presented in this section uses a similar technique to merge MinMax heaps.

4.3.1. MinMax Binomial Heap Forest

A *MinMax binomial heap* is defined as a binary tree on 2^k ($k \geq 0$) keys in which the root containing the minimum key has exactly one child. In addition, the subtree rooted at the child is a perfect MaxMin heap on the remaining $2^k - 1$ keys. Thus the tree is minmax-ordered. A *MaxMin binomial heap* is maxmin-ordered and is defined similarly.

Example 4.1. Figure 4.3 illustrates the shape of a binomial heap. ♦

Figure 4.3. A Binomial Heap

Algorithm 4.1 shows how two equal-sized, MinMax binomial heaps can be merged into a MaxMin binomial heap. The operation has logarithmic time worst case complexity.

Algorithm 4.1. Merging Equal-Sized MinMax Binomial Heaps

Input. Equal-sized MinMax binomial heap, p and q .

Output. A MaxMin binomial heap containing all items in p and q .

```

function Merge ( $p, q$ )
    if KEY (ROOT ( $p$ )) > KEY (ROOT ( $q$ )) then
        swap trees  $p$  and  $q$ 
    make child of  $p$  a second child of ROOT ( $q$ )
    make  $q$  an only child of ROOT ( $p$ )
    TrickleDown ( $p$ )
    return  $p$ 
end;
```

The TrickleDown procedure for MinMax binomial heaps is a slightly modified version of the TrickleDown procedure for MinMax heaps described in section 2.1.6.

A MinMax heap is in fact an ordered collection of MinMax and MaxMin binomial heaps. In particular, each node on the path from the root to the last leaf in a MinMax heap is a root of some MinMax or MaxMin binomial heap. We refer to such an ordered collection as a *MinMax binomial heap forest* or $BHF = (B_p, B_{p-1}, \dots, B_0)$ satisfying the following properties:

- a) Each B_i ($i = 0, 1, \dots, p$) is either a MinMax or MaxMin binomial heap.
- b) If KEY (B) denotes the key at the root of B , then

$$\text{KEY}(B_i) \leq \text{KEY}(B_{i-1}) \text{ if } p - i \text{ is even; and}$$

$$\text{KEY}(B_i) \geq \text{KEY}(B_{i-1}) \text{ if } p - i \text{ is odd.}$$
- c) If SIZE (B) denotes the size of B , then $\text{SIZE}(B_i) \geq \text{SIZE}(B_{i-1})$.

A MinMax binomial heap forest or BHF is said to *correspond* to a MinMax heap H if i) BHF contains all the keys in H ; and ii) if node p has child q in BHF, then this relation also holds in H .

Example 4.2. Figure 4.4 illustrates the BHF corresponding to the 25-heap. Note that each binomial heap in the BHF is rooted on the path from the root to the last leaf in the heap. ♦

Figure 4.4. BHF corresponding to the 25-Heap

4.3.2. Valid Descriptors

We associate with every BHF a *descriptor* $D = (d_p, d_{p-1}, \dots, d_0)$ where d_i gives the number of binomial heaps of size 2^i in a BHF ($i = 0, 1, \dots, p$). A descriptor is *valid* if i) the partial sums $\sum_{(i=0, 1, \dots, j)} d_i$ are equal to $j+1$ or $j+2$; and ii) $0 \leq d_i \leq 2$.

Any BHF corresponding to a MinMax heap has a valid descriptor. Furthermore, the descriptor is unique. Thus two MinMax heaps can be merged together by constructing a BHF with a valid descriptor for the merged heap.

Example 4.3. The descriptor associated with the 25-heap of Figure 4.4 is $(1, 0, 1, 2, 1)$. This means that the BHF for the 25-heap contains the following binomial heaps: one of size 2^4 ; zero of size 2^3 ; one of size 2^2 ; two of size 2^1 ; and one of size 2^0 . It can easily be

verified that the descriptor is valid. ♦

4.3.3. Overview of Algorithm B

Algorithm B uses the binomial heap structure of MinMax heaps to merge together an N-heap and M-heap. Algorithm 4.2 details the merging procedure.

Step 1 of the algorithm ensures that all the roots of MinMax binomial heaps of heights $h = \log n, \log n - 1, \dots, \log (n/m) - 1$, are smaller than or equal to all keys in binomial heaps of heights $\leq h$. Similarly, all the roots of MaxMin binomial heaps of heights $h = \log n, \log n - 1, \dots, \log (n/m) - 1$, are greater than or equal to all keys in binomial heaps of heights $\leq h$.

Step 2 gives a collection of binomial heaps of heights $h = \log m, \log m - 1, \dots, 0$, having the smallest and the largest of all keys in binomial heaps of heights $\leq h$. (Therefore, a BHF can be obtained by converting appropriate binomial heaps so that their roots are minmax-ordered.)

Finally, step 3 creates a BHF with a valid descriptor for the (N+M)-heap from the collection of binomial heaps obtained in steps 1 and 2. Each component of the valid descriptor $D = \{d_i\}$ ($i = 0, 1, \dots, \log (n+m)$) is examined in turn. The number of binomial heaps of size 2^i needed for the BHF and their (minmax- or maxmin-) ordering is thus determined. If the ordering is incorrect, the binomial heap is simply converted before it is added to the BHF. If the binomial heaps are not needed in the BHF, they are merged using Algorithm 4.1. An inductive argument shows that it is always possible to create a BHF for the (N+M)-heap in this way [StSa85].

Algorithm 4.2. Merging Algorithm B

Input. An N-heap and M-heap, of sizes n and m respectively.

Output. An (N+M)-heap, containing all items in N-heap and M-heap.

- Step 0.* Find the BHF's, $nB = (nB_p, nB_{p-1}, \dots, nB_0)$ and $mB = (mB_q, mB_{q-1}, \dots, mB_0)$, corresponding to the N-heap and M-heap respectively.
- Step 1.* Set k to the smallest index for which $SIZE(nB_i) \geq m$. Inspect binomial heaps nB_i ($i = p, p-1, \dots, k$) in turn. If nB_i is minmax-ordered, swap the minimum keys of nB_i and M-heap if the key of M-heap is smaller; if nB_i is maxmin-ordered, swap the maximum keys of nB_i and M-heap if the key of M-heap is larger. Restore the order in M-heap using the TrickleDown procedure.
- Step 2.* for the remaining binomial heaps ($h = \log m, \log m - 1, \dots, 0$) do
 $S := \{ \text{all binomial heaps of height } h \text{ in } nB \text{ and } kB \}$
 Sort S in ascending order on minimum key and let the result be T , and
 in descending order on maximum key and let the result be T'
 Assume $T = T'$ (if not, swap keys and reorder)
 Set nH_s and mH_t to the largest subheaps of height $< h$ in N-heap and
 M-heap respectively
 for all the binomial heaps in $T = \{T_i\}$ ($i = 1, 2, \dots, |T|$) do
 TrickleDownMin(T_i, nH_s)
 TrickleDownMin(T_i, mH_t)
 TrickleDownMax(T_i, nH_s)
 TrickleDownMax(T_i, mH_t)
 endfor
 endfor;
- Step 3.* Assume that $U = \{U_i\}$ ($i = 0, 1, \dots, r$) contains all binomial heaps obtained in steps 1 and 2, and B is the BHF for the (N+M)-heap
 for all binomial heaps U_i ($i = 0, 1, \dots, r$) do
 if B needs a binomial heap of size 2^i then
 Add U_i to B if it is correctly ordered; or convert it first
 else Convert U_{i+1} or U_i if they are not ordered in the same way
 $U_{i+1} = \text{Merge}(U_{i+1}, U_i)$
 endif
 endfor;

4.3.4. Complexity Analysis for Algorithm B

In Algorithm 4.2, step 0 creates BHF's for N-heap and M-heap in $O(\log n + \log m)$ time. Step 1 compares keys of $O(\log(n/m))$ roots. If the keys are swapped after performing a comparison, it is the (minmax or maxmin-) ordering of the smaller binomial heap that must be restored. This means that each trickle down takes at most $O(\log m)$ time. So step 1 can be performed in $O(\log(n/m) \log m)$ time.

In step 2, binomial heaps of heights up to $O(\log m)$ are sorted by minimum and maximum keys. It follows from the properties of a valid descriptor that there are at most 4 binomial heaps of a given height. Therefore, a trickle down is performed in at most $O(\log m)$ time for each of the $O(\log m)$ binomial heaps. Therefore, step 2 requires at most $O(\log^2 m)$ time.

Finally, in step 3, binomial heaps are converted and merged as necessary to create a BHF with a valid descriptor. At most $O(\log m)$ binomial heaps of sizes $O(m/2^i)$, $i = 0, 1, \dots, O(\log m)$, are merged in pairs. Two binomial heaps, each of size m , are merged in at most $O(m + \log m)$ time; step 3 can therefore be performed in $O(m + \log^2 m)$ time.

So, in the worst case, $O(m + \log n \log m)$ comparisons and data movements are performed by Algorithm B.

Chapter 5

MINMAX LEFTIST TREE

5.1. Definition and Representation

The *MinMax leftist tree* is inspired by the leftist tree data structure of section 2.3.1. A binary tree is said to be *leftist* if the shortest path from any node in the tree to a leaf is to the right. Leftist trees are generally "heavier" on the left, hence their name. A MinMax leftist tree is defined as a binary tree that is both leftist and minmax-ordered.

A MinMax leftist tree is represented as a linked binary tree in which every internal node is associated with a key and distance. A leaf or external node is represented by a null pointer. The distance of an internal node gives the length of the shortest path from that node to a leaf. If we adopt the convention that the distance of a leaf is equal to 0, then the distance of an internal node x , $\text{DISTANCE}(x)$, is given by

$$\text{DISTANCE}(x) = \text{DISTANCE}(\text{right child of } x) + 1.$$

The shortest path from any node to a leaf is obtained by following right links. A MinMax leftist tree can be linear with all right links being null, or balanced in which case it may resemble a MinMax heap.

Example 5.1. Figure 5.1 illustrates a MinMax leftist tree. Note that each subtree is either a MinMax or MaxMin leftist tree. ♦

Node Format :

KEY	DISTANCE
LEFT	RIGHT

Figure 5.1. An Example of a MinMax Leftist Tree

5.2. Construction Algorithm

In the previous section, we noted that a MinMax leftist tree can be heap-shaped. Therefore, it is possible to construct a MinMax leftist tree using an adaptation of the linear time MinMax heap construction algorithm (see section 2.1.6). The distances of nodes at each level are initialized at the same time as the tree is being constructed bottom-up. The following theorem is easily obtained.

Theorem 5.1. A MinMax leftist tree on n nodes is constructed in $O(n)$ time.

5.3. Insertion Algorithm

An insertion into a MinMax leftist tree consists of finding the position of the new node on a path from the root to a leaf; creating a "hole" in the tree for the new node by "pushing down" nodes at alternate levels; and, after inserting the new node into the tree, adjusting the distances of nodes on the path in bottom-up fashion. It is best to do an

insertion towards the rightmost leaf since it is closest to the root.

Algorithm 5.1 describes the insertion algorithm. Procedure PushDown reinserts a node farther down the rightmost path. Procedure FixDistance recomputes the distances of nodes on the insertion path and makes the tree leftist by swapping left and right subtrees as necessary

Algorithm 5.1. Insertion Algorithm

Input A MinMax leftist tree, *tree*, and a new node, *new*, not in the tree.

Output A revised MinMax leftist tree, *tree*, with the inserted node.

procedure Insert (*tree*, *new*)

if *tree* is empty **then**

tree := *new*

 DISTANCE (*tree*) := 1

else

if ROOT (*tree*) is on a min-level **then**

if KEY (ROOT (*tree*)) ≤ KEY (*new*) **then**

 Insert(RIGHT (*tree*), *new*)

else { > }

 PushDown(RIGHT (*tree*), ROOT (*tree*))

 replace root of *tree* with *new*

endif

else { ROOT (*tree*) is on a max-level }

if KEY (ROOT (*tree*)) ≥ KEY (*new*) **then**

 Insert (RIGHT (*tree*), *new*)

else { < }

 PushDown(RIGHT (*tree*), ROOT (*tree*))

 replace root of *tree* with *new*

endif

 FixDistance(*tree*)

endif

end;

Algorithm 5.1 a). PushDown

Input. A MinMax or MaxMin leftist tree, *tree*, and a node, *node*, to be pushed down.

Output. A revised MinMax or MaxMin leftist tree, *tree*, with the inserted node

procedure PushDown(*tree*, *node*)

if *tree* is empty **then**

tree := *node*

 DISTANCE (*tree*) := 1

else

if RIGHT (*tree*) is empty **then**

 PushDown (RIGHT (*tree*), *node*)

else { push down nodes at alternate levels }

 swap right child of *tree* and *node*

 PushDown (subtree rooted at rightmost grandchild of *tree*, *node*)

 FixDistance (RIGHT (*tree*))

endif

 FixDistance (*tree*)

endif

end;

Algorithm 5.1 b). FixDistance

Input. A minmax-ordered tree, *tree*.

Output. A MinMax leftist tree, *tree*.

procedure FixDistance (*tree*)

if DISTANCE (left child of *tree*) < DISTANCE (right child of *tree*) **then**

 swap left and right subtrees of *tree*

 DISTANCE (*tree*) := DISTANCE (RIGHT (*tree*)) + 1

end;

Correctness of Insertion Algorithm

The correctness of Algorithm 5.1 for the trivial case of an insertion into an empty tree is obvious. So assume that a node is inserted into a nonempty MinMax leftist tree.

We will first prove that the tree obtained after an insertion is minmax-ordered.

The position of the new node is found so that the path from the root to the new node is minmax-ordered. In addition, the subtree rooted at the new node is obtained by pushing down nodes on the rightmost path (starting at the level of the new node) to lower levels. Therefore, to show that the revised tree is minmax-ordered, we must show that the subtree rooted at the new node is appropriately ordered.

Suppose l is a min level. Let $\text{node}(l)$ denote the rightmost node at level l , and $\text{key}(l)$ denote its key. When $\text{node}(l)$ is pushed down, it is replaced by $\text{node}(l - 2)$ or by the new node, neither of which has a key larger than $\text{key}(l)$. The following two cases show that the subtree rooted at l is minmax-ordered when $\text{node}(l)$ is reinserted in place of a node farther down the rightmost path.

Case 1. A node from level l replaces a node at level $l + 1$. Since $\text{node}(l + 1)$ is null, the subtree rooted at level l is obviously minmax-ordered.

Case 2. A node from level l replaces a node at level $l + 2$. All keys in the new subtree are at least as large as $\text{key}(l)$. Since $\text{key}(l) \leq \text{key}(l + 2)$, the subtree rooted at $l + 2$ is minmax-ordered. Also, $\text{key}(l + 1) \geq \text{key}(l + 2)$, and $\text{key}(l + 1)$ is the largest in its subtree. So, the subtree rooted at level l is minmax-ordered.

A similar argument is applied when l is a max level. It follows that the revised tree is minmax-ordered.

We will now prove that the tree is also leftist. Denote the rightmost path in the tree by P . An insertion into the tree increases the length of P by exactly one. Moreover, the left subtree of any node on P is unaffected by the insertion. Also, note that any subtree of a leftist tree must be leftist. The algorithm adjusts only distances of nodes on P , by moving from the rightmost node up towards the root. The algorithm adjusts the distance of any node x by swapping its subtrees if required and updating its distance. So, if both

subtrees of x were leftist, after adjusting the distance of x , the subtree rooted at x is also leftist. The proof is completed by induction on the length of P . ♦

Theorem 5.2. Algorithm 5.2 inserts a new node into a MinMax leftist tree with n nodes in at most $O(\log n)$ time.

Proof. The definition of a MinMax leftist tree implies the existence of at least 2^d leaves below a node with distance d ; otherwise there would be a shorter path from that node to a leaf. Thus the rightmost path in a MinMax leftist tree of size n contains at most $\lfloor \log(n+1) \rfloor$ nodes. A new node is inserted into a MinMax leftist tree by traversing this path, and therefore, the worst case occurs when the tree is balanced.

The complexity of Algorithm 5.2 is determined by the worst case number of key comparisons, pointer manipulations and distance updates that are performed. The number of comparisons is at most $\lfloor \log(n+1) \rfloor$; the comparisons are used to establish the position of the node in the tree. Pointer manipulations affect nodes on the rightmost path and so, $O(\log n)$ pointer manipulations are sufficient. Note that the distance of a node does not change after its subtrees have been swapped. Consequently, the insertion algorithm can be modified so that distances are updated only if no children are swapped. In the worst case, the distances of $\lfloor \log(n+1) \rfloor$ nodes on the rightmost path are incremented by one. ♦

Additional Remarks on the Insertion Algorithm

It is possible to determine whether a new node should be inserted at a min level or max level in a MinMax leftist tree provided that the key of the rightmost node is known. The node is inserted at a min level if its key is smaller than the key of the rightmost node, and at a max level otherwise. Therefore, it is necessary only to examine nodes lying on

alternate levels on the path from the root to the rightmost leaf. In a linked binary tree implementation, the rightmost node is located in logarithmic time by traversing the rightmost path.

The rightmost node is accessed in $O(1)$ time by storing with each node x an extra pointer $NEXT(x)$, defined as the $PARENT(x)$ if x is a right child and the rightmost node in the subtree rooted at x if x is a left child or the root. Thus right paths are circularly linked as illustrated in Figure 5.2. An insertion is done bottom-up (cf. insertion into MinMax heap) and requires approximately $(\log n)/2$ comparisons in the worst case. However, there are three pointers per node instead of two, and the extra pointer must be updated each time children are swapped. (The asymptotic running times of operations described in later sections are unaffected by this change in the data structure.)

Figure 5.2. Linking Right Paths Circularly

In fact, the position of the new node can be found by applying a binary search since keys on alternate levels on any path are in sorted ascending or descending order. The number of comparisons is reduced to $O(\log \log n)$. However, an additional $O(\log n)$ of array storage is required to index the keys. The method is useful if comparisons are expensive as in the case of extremely large key values.

The overall complexity of the insertion algorithm does not change following these

improvements. This is due to the fact that $O(\log n)$ pointer manipulations and distance updates are required in the worst case for an insertion. It should be noted that the methods described above to search for the position of a new node are applicable to all minmax-ordered tree structures.

5.4. Conversion Algorithm

A general procedure for converting a MinMax binary tree into a MaxMin binary tree uses *rotations* as illustrated in Figure 5.3. A single rotation makes the maximum node the new root. But the new subtree of the minimum node is now minmax-ordered instead of maxmin-ordered (T_2 in Figure 5.3 b)). A MaxMin binary tree is obtained by recursively converting this subtree.

Figure 5.3. Converting a MinMax Binary Tree

The algorithm to convert a MinMax leftist tree into a MaxMin leftist tree proceeds similarly. The main difference is that the minimum node with one of its subtrees always replaces the right subtree of the maximum node. So, it is the right subtree of the maximum node that gets recursively converted by the algorithm. We choose to convert the right subtree over the left subtree because of the leftist property of the tree. Additionally,

distances of the minimum and maximum nodes are adjusted. Algorithm 5.2 describes the procedure for converting a MinMax leftist tree into a MaxMin leftist tree. Procedure FixDistance adjusts the distances of the nodes.

Algorithm 5.2. Conversion Algorithm

Input. MinMax leftist tree, *tree*.

Output MinMax leftist tree converted into a MaxMin leftist tree, *tree*.

```

procedure Convert (tree)
    if tree contains zero or one node then
        return tree
    else
        { Let minimum be the root of the tree and }
        { maximum be the larger child of the root }
        if maximum is the left child of the root then
            LEFT (minimum) := Convert (RIGHT (maximum))
        else { maximum is the right child of the root }
            RIGHT (minimum) := Convert (RIGHT (maximum))
        RIGHT (maximum) := tree rooted at minimum
        FixDistance (minimum)
        FixDistance (maximum)
        return tree rooted at maximum
    endif
end;

```

Correctness of the Conversion Algorithm /

The proof of correctness of Algorithm 5.2 is by induction on the height h of the MinMax leftist tree to be converted. The algorithm is obviously correct for the trivial cases where the tree is empty ($h = -1$) or consists of a single node ($h = 0$). The induction hypothesis is that all trees of heights up to $h \geq 0$ are correctly converted.

Suppose a tree of height $h + 1$ is converted in which the left child of the root has the maximum key. Then the tree converted via a recursive call to the conversion algorithm has height at most $h - 1$. By the induction hypothesis, the converted tree is a MaxMin leftist tree and becomes the left subtree of the old root containing the minimum key. The right subtree of the old root is also a MaxMin leftist tree. A call to procedure FixDistance swaps the left and right subtrees of the old root if the distance of the right child is larger and updates its distance. So in the converted tree, it follows that both subtrees of the root containing the maximum key are MinMax leftist trees. A second call to procedure FixDistance adjusts the distance of the root. So the converted tree is a MaxMin leftist tree.

A similar argument is used if the right child of the root has the maximum key. ♦

Theorem 5.3. Algorithm 5.2 converts a MinMax leftist tree of height h into a MaxMin leftist tree in at most $O(h)$ time.

Proof. The complexity of Algorithm 5.2 is proportional to the length of the path traversed by the algorithm. ♦

A worst case for Algorithm 5.2 maximizes the length of the path traversed by the algorithm. From the definition of the MinMax leftist tree, it is clear that this path lies to the left. In fact, the leftmost path traversed by the algorithm is described by $(0101 \dots)$ where 0 represents "go to left child of node" and 1 represents "go to right child of node". In the worst case, the length of this path is $O(n)$ as illustrated in Example 5.2.

Example 5.2. The leftmost path traversed by Algorithm 5.2 is highlighted for the MinMax leftist tree illustrated in Figure 5.4. The path length is $O(n)$.


```

procedure MergeMin ( $p, q$ );
    if  $p$  is nonempty then
        if  $q$  is empty then  $q := p$ 
        else
            if KEY (ROOT ( $p$ )) < KEY (ROOT ( $q$ )) then
                swap trees  $p$  and  $q$  {  $q$  contains the minimum key }
             $t := \text{Convert} (p)$ 
            MergeMax ( $t$ , RIGHT ( $q$ ))
            FixDistance ( $q$ )
        endif
    end;

```

Procedure MergeMax which merges two MaxMin leftist trees is similar.

Correctness of Merging Algorithm

The merging algorithm for MinMax leftist trees employs conversion to ensure that the nodes lying on the merged path are minmax-ordered. The height of a tree may remain unchanged, or increase or decrease by one after it has been converted. Thus, the proof of correctness involves a double induction.

Let $m(k, l)$ with $0 \leq k \leq l$, denote the merge operation on two trees of heights k and l . Furthermore, assume that the minimum or maximum of all keys is always found in the first tree. We will prove the correctness of the merging algorithm for $m(k, l)$ by induction on k . The basis, $k = 0$, is easy since the algorithm is obviously correct for $m(0, l)$. Now for the induction on k , suppose that the algorithm is correct for $m(k, l)$ with $k < K$ and for all l . We must show that the algorithm is correct for $m(k, l)$ for all k and l . This is done by induction on l .

For the basis of the induction on l , it is obvious that the algorithm is correct for $m(K, 0)$. By the induction hypothesis for k , we also know that the algorithm is correct for

$m(K-1, l)$. Now to complete the induction step on k , we must do the inductive step for l .

To solve $m(K, l)$ for $l > 1$, the algorithm first locates the minimum key. By our initial assumption, this key is in the first tree. The second tree is converted and merged into the right subtree of the first tree. But the subtree is of height at most $K-1$, and by the induction hypothesis, this step is solved correctly. So, in the tree obtained after solving $m(K, l)$, the root contains the minimum key and both its subtrees are MaxMin leftist trees; the distance of the root is recomputed to make the tree leftist. Therefore, the algorithm is correct for $m(k, l)$ for all k and l . ♦

Theorem 5.4. Algorithm 5.3 merges two MinMax leftist trees, each of height h , into a MinMax leftist tree in at most $O(h^2)$ time. In particular, Algorithm 5.3 merges two balanced MinMax leftist trees, each on $n/2$ nodes, into a MinMax leftist tree on n nodes, in at most $O(\log^2 n)$ time.

Proof. The complexity of Algorithm 5.3 is dominated by the time required to perform all the conversions. Since the maximum height of the trees is h , at most $O(h)$ trees are merged; so at most $O(h)$ conversions are performed. By Theorem 5.3, each conversion takes $O(h)$ time in the worst case. Therefore, the worst case complexity of the algorithm is $O(h^2)$. ♦

5.6. Deletion Algorithm

Two different methods can be used to perform deletions on a MinMax leftist tree. The first involves removing the node containing the minimum or maximum key, and promoting nodes towards the empty "hole" in the tree as in MinMax heaps (see section 2.1.6). Thus, a deletion is processed in time proportional to the maximum height of the

tree, that is, $O(n)$. For example, suppose that the smallest keys in the MinMax leftist tree of Figure 5.5 are found on the leftmost branch at min levels. Then, roughly one-third of the nodes are promoted along the leftmost path if the root is deleted.

Figure 5.5. A Worst Case for the Deletion Algorithm

In the second method, the two subtrees of the deleted node are merged and the resulting tree is converted. The conversion is necessary since the subtrees are not ordered in the same way as the original tree. If the node containing the maximum key is removed from the tree, the converted tree replaces the deleted node and its subtrees. The worst case complexity of this algorithm is the same as the merging algorithm, that is, $O(n^2)$.

Chapter 6

MERGEABLE DOUBLE-ENDED PRIORITY QUEUES

6.1. Converting MinMax-Ordered Trees

In previous chapters, we learnt that MinMax heaps and MinMax leftist trees cannot be merged in less than linear time in the worst case. Furthermore, the merge operation can be done efficiently only if the convert operation can be done efficiently. In this section, we shall discuss algorithms to convert minmax-ordered trees into maxmin-ordered trees.

6.1.1. A MinMax-Ordered Tree with Unbounded Fanout

Consider a minmax-ordered tree t in which any node can have an unlimited number of children. To convert t , locate the maximum child of the root and remove its subtree t' . Make t an offspring of t' .

The complexity of the algorithm is proportional to the degree of the root of t . Note that any conversion algorithm must find the maximum key. Therefore, this algorithm is optimal.

6.1.2. A MinMax-Ordered Tree with Bounded Fanout

Suppose any node in a minmax-ordered tree is of degree at most d . Furthermore, the tree is of unbounded height h . We discuss two ways to convert the tree. The first method removes a leaf l and makes it the root of the tree. To establish a maxmin-ordering

in the tree, find the largest grandchild of l and swap it with l . In this way, l trickles down to its correct position. The worst case complexity of this algorithm is $O(d^2h)$.

In the second method, locate the maximum key in the minmax-ordered tree t by inspecting the children of the root. Remove the subtree rooted at the maximum node. If the maximum node has fewer than d children, make t its offspring. Otherwise, replace the last subtree s of the maximum node with t . Now the original root (at t) has at most $d - 1$ children. Make s an offspring of the original root. However, s is minmax-ordered instead of maxmin-ordered and so, it is recursively converted by the algorithm. Therefore, the algorithm requires at most $O(dh)$ time.

6.1.3. A MinMax-Ordered Binary Tree of Given Shape

A minmax-ordered binary tree of given shape is converted into a maxmin-ordered binary tree of the same shape in a recursive procedure. Swap the minimum and maximum keys so that they are correctly located with respect to the maxmin-ordering. However, the second subtree of the minimum (if it exists) and the subtrees of the maximum must now also be converted. This algorithm takes linear time in the worst case.

6.2. MinMax-Ordered Double-Ended Priority Queues

In section 2.3, we presented min-ordered priority queue structures, including binomial queues, pagodas and fibonacci heaps. It appears that minmax-ordered versions of these structures cannot be used to implement efficient double-ended priority queues. The main problem is merging (or converting) minmax-ordered queues in less than linear time. This section briefly describes the merge operation for these queues.

6.2.1. MinMax Binomial Queues

The basic operation on binomial queues is merging equal-sized B_k trees. Two MinMax B_k trees are merged into a MinMax B_k tree as follows. The B_k tree with the larger root is converted so that it is maxmin-ordered and made an offspring of the B_k tree with the smaller root. However, the conversion problem is hard, rendering the MinMax binomial queue unsuitable for implementing the double-ended priority queue.

6.2.2. MinMax Pagodas

Insertions and deletions on pagodas are achieved by merging disjoint pagodas. Merging pagodas is done bottom-up with the rightmost path of one being merged with the leftmost path of the other. The average length of these paths in a pagoda of size n is $\sum_{i=1, 2, \dots, n} i^{-1}$. Merging minmax-ordered trees along a path is difficult since expensive conversions are required to ensure that each subtree along the merged path is ordered appropriately (minmax- or maxmin-ordered).

6.2.3. MinMax Fibonacci Heaps

A deletion in F-heaps corresponds to linking equal-sized trees. A fundamental property of F-heaps is that a sequence of $n+1$ insertions and one deletion gives a binomial queue of size n . Two MinMax F-heaps can be linked by converting the F-heap with the larger root and making it a subtree of the F-heap with the smaller root. As with binomial queues, converting MinMax F-heaps into MaxMin F-heaps is hard.

6.3. Mergeable Min-Ordered and Max-Ordered Queues

In this section, we will show how traditional min- and max-ordered structures can be used to implement mergeable double-ended priority queues.

6.3.1. Williams' Twin Heap

It has been shown in [SaSt85] that Min heaps (and Max heaps) of sizes m and n can be merged using $O(\log m \log n)$ comparisons in the worst case. We shall employ a similar technique to merge Williams' twin heap (see section 2.1.2). Nontrivial modifications to the algorithm of [SaSt85] are required to achieve the same time complexity. (See also section 4.2).

A twin heap on n keys consists of a Min heap A and Max heap B of equal size. An additional restriction is placed on keys of A and B . If any two keys a and b are identically situated in heaps A and B respectively (for example, in locations $a[i]$ and $b[i]$ in array implementations of A and B), then

$$a \leq b \quad (1)$$

In pointer-based representations of twin heaps, we assume that a and b are doubly linked.

Merging Perfect Heaps of Equal Size

Perfect equal-sized twin heaps are combined by merging the two Min heaps and two Max heaps. To merge the Min heaps, say A_1 and A_2 , remove the last leaf of A_2 and make A_1 its left subtree and A_2 its right subtree. The min-ordering is established in the resulting heap-shaped binary tree by allowing the root to trickle down to its correct position (cf. deletion). The Max heaps are merged similarly.

If a root trickles down all the way to the bottom in either heap, it is possible that (1) does not hold after the merge operation. In this case, the twin heap order is restored by swapping at most 4 keys located at corresponding leaf positions in both heaps. The heaps are then reordered as described in section 2.1.2. Therefore, perfect equal-sized twin heaps can be merged using a logarithmic number of comparisons.

Merging Perfect Heaps of Different Sizes

Now consider the problem of merging perfect twin heaps of sizes $2n$ and $2m$ ($n \geq m$). The Min heaps, A_1 and A_2 , are combined in the following way: Locate the rightmost subtree of the same size as A_2 in A_1 ; call the subtree K_A . Keys in A_1 on the path from the root to K_A are successively compared with the minimum key in A_2 . If the key in A_2 is smaller, the two keys are swapped and A_2 is reordered. This procedure is called *walk down*.

Suppose the Max heaps associated with A_1 and A_2 are called B_1 and B_2 respectively. The merging algorithm must always ensure that (1) holds. Suppose a new key, inserted into A_2 during walk down, trickles down all the way to the bottom in A_2 . Then, corresponding leaves in A_2 and B_2 are compared, and if necessary, swapped and B_2 is reordered.

Denote the rightmost subheap of size B_2 in B_1 by K_B . A walk down to K_B is also repeated on B_1 and B_2 , this time updating A_2 so that (1) holds. The walk down in both heaps guarantees that keys in A_1 on the path to K_A are no larger than keys in A_2 , and similarly, keys in B_1 on the path to K_B are no smaller than keys in B_2 .

Finally, A_2 is merged into K_A and B_2 is merged into K_B using the procedure for merging perfect equal-sized heaps. The algorithm for merging perfect heaps of different sizes requires at most $O(\log n \log m)$ comparisons.

Merging a Perfect into a Non-Perfect Heap

To merge a perfect heap of size m into a non-perfect heap of size n ($n \geq m$), first we find the *lowest common ancestor* (or *LCA*) in the larger heap. The *LCA* covers all new leaves in the larger heap, that is, all leaf positions which get occupied as a result of merging the smaller heap into the larger heap are descendants of *LCA*. The subtree rooted

at LCA, called L-heap, is of minimum height.

Let the height of the perfect heap be h . Then it is always possible to find two nodes, p_l and p_r , in the larger heap so that i) the subtree rooted at p_l (called p_l -heap) has height $h+1$; ii) the subtree rooted at p_r (called p_r -heap) has height h ; and iii) p_l and p_r together cover all new leaves.

The merging algorithm for twin heaps can now be outlined. A walk down is performed to LCA in A_1 and B_1 in the same manner as described previously. Suppose the height of the L-heaps is also h . Then, the perfect heaps, A_2 and B_2 , are merged into L-heaps in A_1 and B_1 respectively using a procedure analogous to merging perfect equal-sized heaps.

If the L-heaps are not of height h , then nodes p_l^A and p_r^A in A_1 , and nodes p_l^B and p_r^B in B_1 are found. A_2 is merged into the perfect p_r^A -heap as described before, except that this time it is the last leaf of p_l^A -heap that is removed and made the root. The result is a perfect heap of height $h+1$. Similarly, p_r^B -heap and B_2 are combined.

A walk down is now performed from LCA to p_l^A , updating p_r^A - and p_r^B -heaps; from LCA to p_l^B , also updating p_r^A - and p_r^B -heaps; from LCA to p_r^A , updating p_l^A - and p_l^B -heaps; and from LCA to p_r^B , also updating p_l^A - and p_l^B -heaps. The result of the merge operation is obtained by swapping p_l^A - and p_r^A -heaps, and p_l^B - and p_r^B -heaps. The algorithm requires at most $O(\log m \log n)$ comparisons.

Merging Non-Perfect Heaps

A non-perfect heap is split into a perfect and non-perfect heap by removing the root and inserting it into one of its subtrees at the first available leaf position. The insertion does not cost any comparisons since it is known that the key must occupy the root position.

The algorithm to merge non-perfect heaps finds the LCA in the larger heaps, A_1 and B_1 , and performs a walk down to LCA as usual. The smaller heaps, A_2 and B_2 , are then repeatedly split and the perfect heaps so obtained are merged into the subtrees rooted at (updated) LCA using any one of the above methods. Note that the height of the perfect heap decreases by at least one at each recursive step. Therefore, the merging algorithm requires at most $O(\log m \log n)$ comparisons.

6.3.2. Min and Max Heaps

A mergeable double-ended priority queue structure can be obtained from the traditional heap structure at the expense of additional storage [WiSa86]. A Min heap and Max heap are constructed on two copies of a set of items. (Alternatively, the heaps can be stored with pointers to single copies of the items.) In addition, the copies of each item in the Min heap and Max heap are doubly linked so that their locations can be obtained easily from one another.

Insertions and deletions are done in the usual way except that both heaps must be maintained. For insertion, two copies of the new item are made; one is inserted into the Min heap, and another into the Max heap. Pointers to each copy of the new item are updated so that the copies are doubly linked.

To delete the minimum item for example, the root of the Min heap is deleted. The copy of the minimum item is then extracted from the Max heap. The resulting "hole" is as usual replaced by the last leaf l in the Max heap. Since the minimum item was removed, all keys below l must be larger than or equal to the key of l . Therefore, the order in the Max heap is reestablished by allowing l to "bubble up" (cf. insertion) towards the root to its correct position. The position of l can also be found by performing a binary search.

To merge heaps on $2n$ and $2m$ items, the two Min heaps and two Max heaps are

merged. Any one of the algorithms of [SaSt85] or [StSa85] is used.

The insertions and deletions require at most logarithmic time. The merge operation can be performed efficiently in at most $O(\log m \log n)$ time if a pointer-based implementation is used.

6.3.3. Min and Max Binomial Queues

Another mergeable double-ended priority queue structure is obtained using binomial queues (see section 2.3.2). The strategy described above for heaps is used. A Min binomial queue and Max binomial queue are constructed for the same set of items. Items in both queues are doubly linked.

To insert an item in the queue, one copy of the item is merged into the Min binomial queue, and the second into the Max binomial queue. The new copies are doubly linked.

To delete the minimum node, the item with the minimum key is located in the Min binomial queue by comparing the roots of all B_k trees in the queue. The minimum node is removed from the Min binomial queue by merging its subtrees into the queue. The minimum node is also removed from the Max binomial queue. The resulting "hole" is filled by the root r of the smallest B_k tree in the queue. The subtrees of r are added to the queue. The order in the B_k tree containing r is restored by allowing r to "bubble up" to its correct position. The maximum node is deleted similarly.

Two queues of sizes $2m$ and $2n$ are combined by merging the two Min binomial queues and two Max binomial queues.

Insertions, deletions and merging can be performed in at most logarithmic time. Finding the minimum node or maximum node also requires logarithmic time. A suitable implementation allows any node in the binomial queues to easily access its parent and children.

6.4. A New Mergeable Double-Ended Priority Queue

We have seen that the minmax-ordering cannot easily be extended to all min-ordered tree structures. The problem is typically the conversion problem which takes linear time in the worst case. In this section, we introduce a new structure, called *minmax-ordered queue*, to represent the double-ended priority queue. The new structure is flexible enough to permit efficient merging.

6.4.1. A Review of Binomial Heaps

The minmax-ordered queue is based on MinMax and MaxMin binomial heaps (see section 4.3.1). Recall that a MinMax binomial heap is defined as a minmax-ordered binary tree on 2^k keys in which the root contains the minimum key and the offspring of the root is a perfect MaxMin heap on $2^k - 1$ keys. A MaxMin binomial heap is defined analogously.

Two equal-sized MinMax binomial heaps can be merged in logarithmic time into a MaxMin binomial heap (refer to Algorithm 4.1). Furthermore, a MinMax binomial heap can be split into two equal-sized MaxMin binomial heaps in logarithmic time. Algorithm 6.1 describes the split operation.

Algorithm 6.1. Splitting a MinMax Binomial Heap

Input. A MinMax binomial heap, p .

Output. Equal-sized MaxMin binomial heaps, p and q .

procedure Split (p, q)

if p is of size > 1 **then**

$q :=$ child of ROOT (p)

 make right child of q an only child of ROOT (p)

 TrickleDown (p)

endif

end;

The TrickleDown procedure for MaxMin binomial heaps is similar to the TrickleDown procedure of section 2.1.6 for MinMax heaps.

6.4.2. MinMax-Ordered Queue

A minmax-ordered queue of size n is defined as a forest of MM_k trees ($k = 0, 1, \dots, \lfloor \log n \rfloor$). An MM_k tree is either a minmax- or maxmin-ordered binomial heap of size 2^k . An additional constraint on the structure is that there is at most one minmax-ordered and one maxmin-ordered binomial heap of a given size in the forest.

Therefore, there are at most $\lfloor \log n \rfloor$ MM_k trees in a minmax-ordered queue and the largest MM_k tree is of height at most $\lfloor \log n \rfloor$. However, the representation for a queue of size n is not unique.

Example 6.1. A minmax-ordered queue of size $n = 11$ is represented as $\{MM_3, MM_1, MM_0\}$ or $\{MM_2, MM_2, MM_1, MM_0\}$. In the last case, one MM_2 is minmax-ordered and the other maxmin-ordered. ♦

6.4.3. Operations on the MinMax-Ordered Queue

A minmax-ordered queue of size n is constructed by repeated pairwise merging. Equal-sized binomial heaps which are ordered (minmax- or maxmin-) in the same way are merged until there is at most one minmax- and one maxmin-ordered binomial heap of any size remaining. Since MM_k trees of size k can be merged in $O(\log k)$ time, it follows that the construction requires at most $O(n)$ time.

Alternatively, the following lemma can be applied to construct the queue.

Lemma 6.1. Any MinMax heap is uniquely represented by a forest of MM_k trees.

Proof. The proof follows from the fact that any MinMax heap can be described by a valid descriptor (see chapter 4 and [StSa85]). Moreover, there are at most two binomial heaps of the same size, and their roots lie adjacently on the path to the last leaf. Since binomial heaps on any path are alternately minmax- and maxmin-ordered, no two binomial heaps of the same size are ordered in the same way. ♦

So to construct a minmax-ordered queue, first construct a MinMax heap on n keys in $O(n)$ time. Each MM_k tree is rooted on the path leading to last leaf in the heap. The minmax-ordered queue is therefore obtained, without performing any comparisons, in additional $O(\log n)$ time.

Two minmax-ordered queues of sizes m and n ($m \leq n$) are merged in at most $O(\log^2 n)$ time. Equal-sized MM_k trees are merged in the usual way, starting from the smallest. An insertion is accomplished by merging a queue on the new key into the original queue. This operation also takes at most $O(\log^2 n)$ time.

The minimum or maximum key is deleted from an MM_k tree by splitting the MM_k tree containing the desired key until two MM_1 trees are obtained. The MM_1 tree with the key to be deleted is removed. All trees obtained from splitting are merged into the queue. Splitting an MM_k tree in this way gives $k + 1$ trees and takes at most $O(k^2)$ time. Therefore, a deletion is achieved in $O(\log^2 n)$ time.

In chapter 1, we defined the split operation on a queue of size n to obtain two queues of sizes $n - m$ and m , where m can be chosen arbitrarily ($m \leq n$). The operation can be performed on a minmax-ordered queue in at most $O(\log^2 n)$ time by Algorithm 6.2.

Algorithm 6.2. Split MinMax-Ordered Queue into two Queues of Arbitrary Sizes

Input. A minmax-ordered queue on n keys, p , and size of split, m .

Output. Two minmax-ordered queues, p and q , of sizes $n - m$ and m respectively.

```

procedure SplitArbitrary ( $p, q, m$ )
     $i := m$ 
    repeat
         $t :=$  (largest binomial heap in  $p$  of size  $\leq i$ ) or
              (smallest binomial heap in  $p$  of size  $> i$ )
        if SIZE ( $t$ )  $\geq i$  then
            split  $t$  into equal-sized binomial heaps and add to  $p$ 
        else
            move  $t$  from  $p$  to  $q$ 
             $i := i - \text{SIZE} (t)$ 
        endif
    until  $i = 0$ ;
    validate ( $q$ )      /* merge equal-sized  $\text{MM}_k$  trees */
end;

```

To find the minimum key in the queue for example, inspect the roots of all minmax-ordered MM_k trees and the children of roots of all maxmin-ordered MM_k trees. The procedure takes at most $O(\log n)$ time. In fact, the minimum and maximum keys can be found in constant time by constructing a MinMax heap on keys which are candidates for the minimum and maximum. The heap is of size at most $2 \lfloor \log n \rfloor$ and can be constructed in $O(\log n)$ time. The heap is updated during insertions and deletions in $O(\log \log n)$ time. The worst case complexities of the other operations are not affected by this change in the data structure.

6.4.4. Implementing the MinMax-Ordered Queue

A minmax-ordered queue is implemented as a collection of binary trees representing binomial heaps. Each node in a binary tree has pointers to its parent and to its children. A MinMax heap accesses the binomial heaps by pointers to the minimum and maximum keys. In addition, we maintain an array containing information about the type (minmax- or

maxmin-ordered), size and number of binomial heaps of given type and size in the queue.

During the merge operation, the information stored in the array is scanned, starting with the smallest binomial heaps. Two binomial heaps of the same type and size are merged into a larger binomial heap.

Our primary interest in the minmax-ordered queue stems from the fact that it allows the merge and split operations to be done in less than linear time. It is likely that the algorithms presented in this section can be made more efficient.

Summary

— We have obtained a conceptually simple data structure using minmax-ordering. The structure gives efficient worst case time complexities for the double-ended priority queue operations. In addition, the structure supports the merge and split operations.

Operation	Time to process an operation on sets of size n
Construct	$O(n)$
Merge	$O(\log^2 n)$
Insert	$O(\log^2 n)$
FindMin FindMax	$O(1)$
DeleteMin DeleteMax	$O(\log^2 n)$
Split	$O(\log^2 n)$

Table 6.1. Worst Case Complexities for MinMax-Ordered Queues

Chapter 7

CONCLUSIONS

7.1. Summary of Results

Lower bounds on the number of key comparisons required, in the worst case, were established for the operations i) to construct a MinMax heap on a given set of keys; ii) to convert a MinMax heap into a MaxMin heap, and iii) to merge two MinMax heaps into a MinMax heap. It was found that an important difference between traditional heaps and MinMax heaps lies in the time needed to perform the merge operation. While traditional heaps are mergeable, it was shown that merging MinMax heaps requires a linear number of comparisons. Even the seemingly simple task of converting a MinMax heap into a MaxMin heap requires a linear number of comparisons.

An algorithm for converting MinMax heaps into MaxMin heaps was presented. The algorithm improves the upper bound for the conversion problem by a constant factor. Two algorithms for merging MinMax heaps were also developed. The first algorithm shows how perfect heaps can be used as building blocks in merging MinMax heaps of arbitrary sizes. The second algorithm uses binomial heaps to merge two MinMax heaps.

A new minmax-ordered data structure, called a MinMax leftist tree, was introduced. A MinMax leftist tree obeys an explicit balance condition that indirectly bounds the length of the rightmost path. A MinMax leftist tree on n keys is constructed in $O(n)$ time.

An insertion takes at most $O(\log n)$ time. However, a deletion takes $O(n)$ time in the worst case as the complexity of the deletion algorithm is proportional to the height of the tree.

Finally, a new representation of a mergeable double-ended priority queue, based on minmax-ordering, was presented. The new queue is composed of a forest of MinMax and MaxMin binomial heaps, and has the appealing property that it can be merged and split efficiently.

7.2. Further Research

In this thesis, we have concentrated on minmax-ordered data structures to implement efficient mergeable double-ended priority queues. Many areas for further research remain.

In chapter 2, we gave a comprehensive list of the known double-ended priority queue structures, and subsequently, introduced new representations. It would be instructive to compare execution times for these data structures.

New applications of the minmax-ordering may also be investigated. Some applications were presented in chapter 2. Recently, the minmax-ordering was also used to solve geometric problems [KNOW86].

Another area of research involves the average case behaviour of the MinMax leftist tree. The execution time complexities of algorithms to delete, convert and merge for the MinMax leftist tree are dependent on the height of the tree. Therefore, it would be useful to consider the average height of the tree. Also, it might be possible to extend the work of Sleator and Tarjan [SlTa86], on amortized time complexities for *self-adjusting* Min leftist trees, to minmax-ordered versions of the data structure.

In most applications, we wish to perform not just a single operation but a sequence of operations. Batched operations on traditional heaps have been studied. The aim is to reduce the average time per operation over a worst case sequence of operations, so that we

may obtain an overall time bound much less than the worst case time per operation multiplied by the number of operations. Batched operations on the MinMax heap might also be investigated.

The minmax-ordering can be generalized. A bit may be associated with each level in a tree to indicate whether it is a min or max level. In fact, a bit may be associated with each node to indicate its level type. It would be interesting to know how many order relations are contained in such a representation. In particular, data (partial orders) may be represented compactly in this way and thus this ordering may be of practical value, if storage is a factor. ⁶

2 OF 2 DE 2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

REFERENCES

- [AHU74] A. V. Aho, J. E. Hopcroft and J. D. Ullman (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.
- [Atk77] M. D. Atkinson (1977/78), "Mathematics in the service of computer programming", *Mathematical Spectrum* 10:1, 6-11.
- [ASSS84] M. D. Atkinson, J.-R. Sack, N. Santoro and Th. Strothotte (July 1984), "An efficient, implicit double-ended priority queue", Technical Report 55, School of Computer Science, Carleton University. Also to appear in *Comm. ACM*.
- [ASSS85] M. D. Atkinson, J.-R. Sack, N. Santoro and Th. Strothotte (March 1985), "MinMaxheaps, orderstatisticstrees and their application to the coursemarks problem", *Proc. 19th Annual Conference on Information Sciences and Systems*, Baltimore, 160-165.
- [Baas78] S. Baase (1978), *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, Mass.
- [BKZ77] P. van Emde Boas, R. Kaas and E. Zijlstra (1977), "Design and implementation of an efficient priority queue", *Mathematical Systems Theory* 10, 99-127.
- [Boas77] P. van Emde Boas (June 1977), "Preserving order in a forest in less than logarithmic time and linear space", *Information Processing Letters* 6:3, 80-82.
- [Brow77] M. R. Brown (1977), "An analysis of a practical and nearly optimal priority queue", Ph. D. Thesis STAN-CS-77-600, Computer Science Department, Stanford University.

- [Cran72] C. A. Crane (1972), "Linear lists and priority queues as balanced binary trees", Ph. D. Thesis STAN-CS-72-259, Computer Science Department, Stanford University.
- [Floy64] R. W. Floyd (December 1964), "Algorithm 245: Treesort3", *Comm ACM* 7:12, 701.
- [FrTa84] M. L. Fredman and R. E. Tarjan (January 1984), "Fibonacci heaps and their uses in improved network optimization algorithms", submitted for publication.
- [FVV78] J. Francon, G. Viennot and J. Vuillemin (1978), "Description and analysis of an efficient priority queue representation", *Proc. 19th Annual Symposium on Foundations of Computer Science*, Ann Arbor, 1-7.
- [GoMu82] G. H. Gonnet and J. I. Munro (July 1982), "Heaps on heaps", *Proc. ICALP 9*, Aarhus, 282-291.
- [Gonn84] G. H. Gonnet (1984), *Handbook of Algorithms and Data Structures*, Addison-Wesley, London.
- [JoDa75] A. Jonassen and O.-J. Dahl (1975), "Analysis of an algorithm for priority queue administration", *BIT* 15, 409-422.
- [KNOW86] R. Klein, O. Nurmi, T. Ottmann and D. Wood (June 1986), "Optimal dynamic solutions for windowing problems", *Proc. 2nd Annual Symposium on Computational Geometry*, Yorktown Heights, 109-115.
- [Knut73] D. E. Knuth (1973), *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass.
- [MuSu80] J. I. Munro and H. Suwanda (October 1980), "Implicit data structures for fast search and update", *J. Computer and System Sciences* 21:2, 236-250.

-
- [NeTe78] O. Nevalainen and J. Teuhola, "The efficiency of two indexed priority queue algorithms", *BIT* **18**, 320-333.
- [SaSt85] J.-R. Sack and Th. Strothotte (1985), "An algorithm for merging heaps", *Acta Informatica* **22**, 171-186.
- [SlTa86] D. D. Sleator and R. E. Tarjan (February 1986), "Self-adjusting heaps", *SIAM J. Computing* **15**:1, 52-69.
- [SPP76] A. Schönhage, M. Paterson and N. Pippenger (1976), "Finding the median", *J. Computer and System Sciences* **13**, 184-199.
- [StSa85] Th. Strothotte and J.-R. Sack (December 1985), "Heaps in heaps", *Congressus Numerantium* **49**, 223-235.
- [Vuil78] J. Vuillemin (April 1978), "A data structure for manipulating priority queues", *Comm. ACM* **21**:4, 309-314.
- [Will64] J. W. J. Williams (June 1964), "Algorithm 232: Heapsort", *Comm. ACM* **7**:6, 347-348.
- [WiSa86] R. Winger and J.-R. Sack (1986), personal communication.

END

13.08.87

FIN