IMPLICIT DATA STRUCTURES FOR THE DICTIONARY PROBLEM

by

Hendra Suwanda

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Research Report CS-80-04

January 1980

IMPLICIT DATA STRUCTURES FOR THE DICTIONARY PROBLEM

ABSTRACT

This thesis explores dynamic data structures that can be implemented without pointers. Such implicit structures keep the storage requirement minimal and therefore, they are of practical interest.

Some implicit and "semi-implicit" structures for representing a dictionary are suggested and analyzed. For implicit structures using a fixed partial order as the underlying scheme, a lower bound on the product of the search cost and the cost for deletion/insertion is given. For structures not using a fixed partial order, it is shown that the product of these costs can be reduced.

Tight lower and upper bounds for forming a new implicit structure, the beap, are given. Also, an efficient algorithm for selecting the k-th smallest element on a beap is proposed. Under a parallel environment, it is shown that beaps allow a reasonable speed-up in performing the basic operations.

ACKNOWLEDGEMENTS

I have been very fortunate to have Professor Ian Munro as my thesis supervisor. He is responsible for my interest in Data Structures. Foremost, I am deeply grateful for his friendship, enthusiasm, encouragement, and guidance. As quoted in this thesis, several results were obtained jointly with him. He read the very first draft of this thesis and improved very much on its presentation.

I also thank the other members of the examining committee, Professors Kelly Booth, Gaston Gonnet, Larry Snyder, Frank Tompa, and Dan Younger, for their numerous helpful comments and criticism. I particular, I wish to thank Professor Kelly Booth for his encouragement during my graduate studies.

I appreciate very much the typing done by Virginia Herridge and Kandry Mutheardy. Kandry also implemented the program for Tables 4.1.1 and 4.1.2. I also wish to acknowledge the assistance of Kandry and Teresa in the final preparation of this thesis.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation

Good design of data structures, and the algorithms acting upon them, is fundamental in solving set manipulation problems which have a wide range of applications [Aho74]. A particularly important set manipulation problem is the dictionary problem: maintaining a dynamic structure that can be used to process a sequence of search, insert, and delete operations over a set of linearly ordered items. The performance of a data structure is usually measured by the time required to process each allowed operation, the time to construct the data structure, and the storage needed to maintain the data structure.

Many solutions have been given for the dictionary problem [Aho74, Knuth73b]. Most of them are based on tree structures with explicit pointers representing the orderings among the keys. While the use of pointers is often crucial to the flexibility and efficiency of the algorithms to process the operations, their explicit representation often contributes heavily to the space requirement. Usually, these structures are designed to process the operations quickly rather than minimize the storage requirement.

In this thesis, we explore a class of data structures for representing a dictionary in which structural information is implicit in the way the data is stored, rather than explicit in pointers. Thus, only a simple array is needed for storing data and no extra storage is wasted for our model of computation as described in Chapter 2. Informally, we say such a data structure is implicit.

The classic example of an implicit data structure is the heap [Williams64]. A heap, containing N elements from some linearly ordered set, is stored as a one dimensional array which satisfies the heap property: $A[i] \leq A[2i]$ and $A[i] \leq A[2i+1]$ (see Fig. 1.1.1). This implicit representation of a tree permits the minimum element to be found immediately and a new element to be inserted in O(log N) steps+. Similarly, an element in a specified position may be deleted in O(log N) steps. The low costs for insertion and deletion suggest that heaps are suitable for representing dynamic structures. Furthermore, a heap of size N can be built in O(N) steps [Floyd64]. Unfortunately, the heap is a very bad representation if searches are to be performed; indeed, it is well-known that the average query requires of $\Theta(N)$ comparisons as there are about N/2 incomparable elements which reside at the leaves.

+ All logarithms are to base 2 unless otherwise stated.

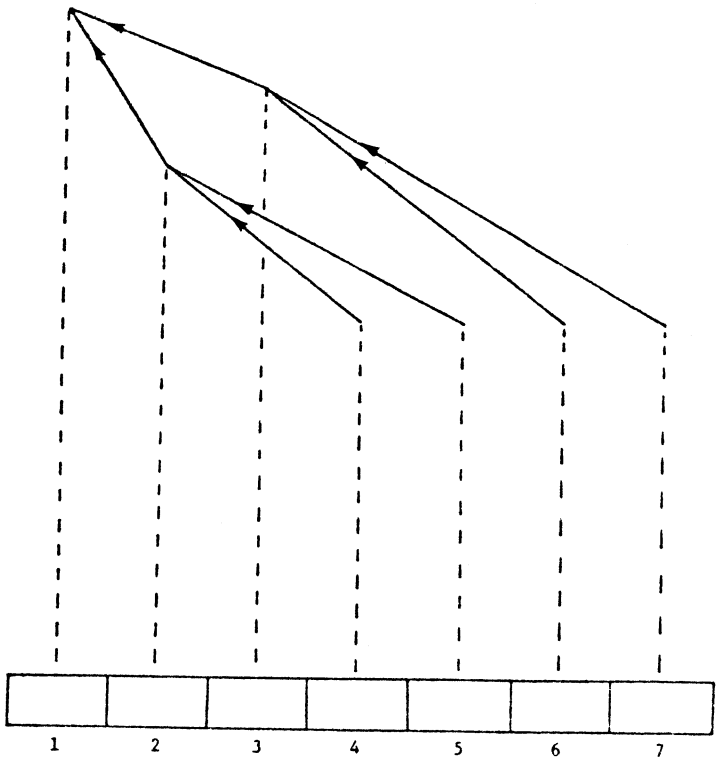


Figure 1.1.1 : A Heap as an Implicit Structure

Another example of an implicit structure is a sorted list (of size N). We can view a sorted list as being constructed by storing the median of the elements in the middle of the array, partitioning the elements into two groups (one containing elements smaller than the median and one containing elements larger than the median), repeating the same process with the smaller elements in the left part of the array and the larger ones on the right. The implicit information in this array is a binary tree corresponding to the

process of binary search (see Fig. 1.1.2). In contrast to the heap, a search can be performed in O(log N) steps. However, the construction of a sorted list requires Θ(N log N) steps, an insertion or a deletion may need Θ(N) steps (moves) to restructure the array.

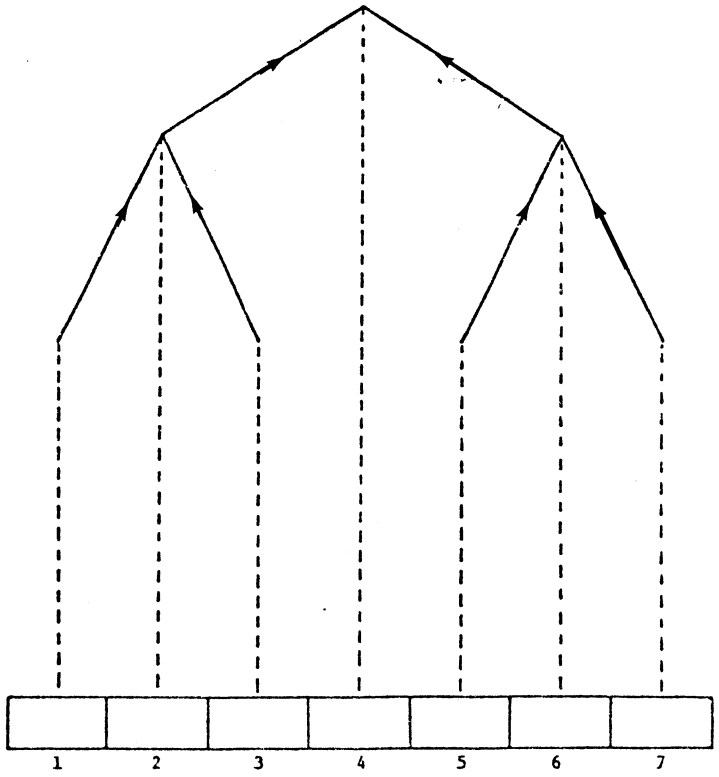


Figure 1.1.2 : A Sorted List viewed as an Implicit Structure for Performing Binary Search.

1.2 A Brief Survey of Related Work

Snyder [Snyder77] has investigated a class of data structures called generalized search-tree structures which includes most tree-like structures such as lists, binary trees, AVL-trees, and 2-3 trees. Informally, a data structure is called "unique" if the shape of the underlying tree structure (e.g. complete binary tree in a heap) depends only on the size of the structure. It is shown that if a generalized search-tree structure is unique, then at least one of the operations insert, delete, or search requires $\Omega(\sqrt{N})$ comparisons or changes (either of pointers or data). Most implementations of these structures use explicit pointers.

Bentley et al. [Bentley78, Saxe79] suggest a data structure without explicit pointers called a binomial list. The N data elements are stored in a simple array, which is divided into sorted sequences called "runs". The number of elements in each of the runs is a power of two. The set of log N run sizes provides the "binary decomposition" of the number N. For example, if the array contains 23 elements, the binomial list contains 4 runs of lengths 16, 4, 2, 1. A search can be performed by doing a binary search on each run. Insertion of a new element can be done by appending a run of length one at position N+1 followed by repeatedly merging the last 2 runs if they are of equal length. It is

shown that inserting N elements into an initially empty binomial list requires N log N - N + 1 comparisons and a search requires $(\log N)2/2$ comparisons in the worst-case. It can be shown that if N is a power of 2, then the total cost of performing an unsuccessful search on each of a sequence of structures of size 1, 2, .., N-1, is about $(N (\log N)2)/4$. This means that $O(N (\log N)2)$ comparisons are sufficient to perform any sequence of N insertions and searches on a binomial list containing O(N) elements, so that on average $O((\log N)2)$ comparisons are sufficient to perform an insertion or search. It is also shown that the bounds are within a constant factor of being optimal for these problems, provided that the only reordering used is the merging of runs, and that the cost of performing such a merge is equal to the sum of the length of the runs being merged. A method of deleting elements is also demonstrated, but this is at the cost of an extra bit of storage per data element and a small increase in the run time. Although this structure has a bad worst-case behaviour, it is very attractive for applications requiring minimum storage, good average behaviour, and few or no deletions.

Another data structure without explicit pointers has been devised by Melville and Gries [Melville78]. Unlike the structures suggested by Bentley et al. or those suggested in this thesis, gaps are introduced at regular intervals in the array. Nevertheless, the size of the array is bounded by 2N

where N is the number of elements in the structure. A new element is inserted into an appropriate gap following a search operation. It can be shown that a sequence of N insert, delete, min and search operations can be done in at most $O(N^{1+1/d})$ steps for any $d \geq 2$. However, the practical utility of the algorithms for $d > 2$ is questionable, and the use of this structure is not recommended when some bounded "response time" is required, due to bad worst-case behaviour of insert and delete operation. It should be noted that a search can be performed in O(log N) steps.

In a general structure, elements can be preprocessed into a partial order so that the subsequent searches can be performed efficiently. The trade-off between the processing cost and each search cost has been examined by Borodin et al. [Borodin79]. They show that if P(N) is the number of comparisons used to preprocess N elements such that the search can be performed in S(N) comparisons, then $P(N) + N \log S(N) \geq (1 + o(1)) N \log N$, for any comparison-based algorithm.

1.3 Thesis Outline

Our major objective in this thesis is to develop pointer free data structures with minimal storage where data is stored contiguously, so that the basic operations insert, delete, and search of a dictionary can be performed efficiently. It is our goal to provide practical data structures for application where storage is restrictive and algorithms that can be coded easily.

Chapter 2 and part of Chapter 3 of this thesis reflect the work done jointly with Professor Ian Munro.

In Chapter 2, implicit data structures for representing a dictionary are introduced. It is shown that the product of the search cost and the deletion/insertion cost is at least N (the number of the elements) for implicit structures which are based on a fixed partial order. One of the structures introduced, the biparental heap (beap)*, is shown to essentially achieve this lower bound. Indeed, $O(\sqrt{N})$ steps is sufficient to perform a search, an insertion or a deletion on a beap. Further improvement of the performance can be achieved if implicit and "semi-implicit" structures based

---

* The name shortened to bpheap and finally to beap by E.L. Robertson.

on other than a fixed partial order are used. Indeed, each of the operations insert, delete and search can be performed in $O(N^{1/3} \log N)$ steps on an implicit structure suggested in Section 2.6.

In Chapter 3, we give lower and upper bounds on forming a beap. We show that the number of comparisons required to form a beap are $1/2\ N \log N$ minus lower order terms, while $1/2\ N \log N$ plus lower order terms are sufficient. We also show how to select the k^{th} smallest element on a beap in $O(\min(k \log k, \sqrt{N} \log N))$ steps.

Chapter 4 contains a description of methods for performing insertion, deletion and search operations on beaps in a parallel processing environment. The algorithm for the parallel search is based on a divide and conquer technique. It is shown that a reasonable "speed up" can be achieved for the above operations.

CHAPTER 2

IMPLICIT DATA STRUCTURES FOR REPRESENTING A DICTIONARY

In this chapter, implicit and "semi-implicit" structures for representing a dictionary are discussed. A model of computation appropriate for this problem is defined and the notions of implicit and semi-implicit structures are made precise.

In order to search quickly in a structure some relations among the elements have to be maintained. The use of a partial order to keep this information seems rather natural and handy. The Figures 1.1.1 and 1.1.2, for example, indicate the Hasse diagrams of the partial order associated with a heap and a sorted list. More precisely, we define a partial ordering on the array locations such that the data elements occupying the locations are consistent with the partial order; that is element K1 $\leq$ element K2 if location(K1) $\leq$ location(K2) in the partial order.

The term "implicit" comes from the fact that the pointers implied by the Hasse diagram do not have to be explicitly stored in the data structure if they can be easily computed. For the data structures defined in this chapter these calculations are easy.

An implicit structure based on a partial order is explored in Section 2.2. It has a much better worst-case behaviour than a heap, sorted list, or binomial list. The product of the cost of a search and of an insertion/deletion is within a constant factor of being optimal for such a class of implicit structures. A lower bound on this a product is given in Section 2.3. This product of costs can be lowered if some scheme other than a partial ordering is used, as is demonstrated in Section 2.4. The more interesting measure, the maximum of the search and modification costs, is lowered in section 2.6, by a scheme not solely based on partial orders.

2.1. The Model of Computation

Our model is a (potentially infinite) one dimensional array in which data are always stored contiguously, even after deletions occur. We will draw no formal distinction between a pointer and an index (an integer in the range [0,N] where N is the number of elements currently in the array). A data structure is implicit if only a constant number of such integers need to be retained. In term of bits, a structure is implicit if only O(log N) bits are used. Most, although not all, of our attention will deal with structures in which N is the only such value required. We will also suggest two structures which are "semi-implicit", in that more than a constant number but o(N) pointers (indices) are kept. Our basic operations on data elements consist of making comparisons between pair of elements (with three possible outcomes <, =, and >) and swapping pairs of elements. Integer arithmetic is allowed only for manipulating indices.

Our measure of complexity is the maximum number of comparisons and swaps required to perform operations on the data structure. This worst-case analysis is in contrast with a closely related problem considered by Bentley et al. [Bentley78]. Their main concern is a good average behaviour rather than our concentration on worst-case.

2.2 The Beap, an Implicit Structure Based on a Partial Ordering

The ordering scheme presented by Bentley et al. [Bentley78] is, like the heap and the sorted list, a fixed partial order (for each N) imposed on the locations of the array. The elements occupying the locations must be consistent with the partial order. In general, the more restrictive the partial order is, the easier it will be to perform searches but the harder it will be to make changes to the structure, because of the many relations that must be maintained. The heap and the sorted list are the two rather extreme examples demonstrating the imbalance between the cost for searching and the cost for modification. In this section we present an ordering scheme which balances these costs.

As we have noted, the heap is a very poor structure upon which to perform searches. The reason for this difficulty is that as a result of each internal node having two sons, there are too many incomparable elements in the system (the N/2 leaves). At the other extreme is a sorted list, which is difficult to update because of the long chain that has to be maintained in the ordering. Our first compromise between the costs of searching and updating is to create a structure with the same parent-child relationship as a heap, in which most nodes have two children. Unlike a

heap, however, most nodes will have two parents. We called this structure a biparental heap, a name in obvious need of shortening. We will, therefore, refer to it as a beap.

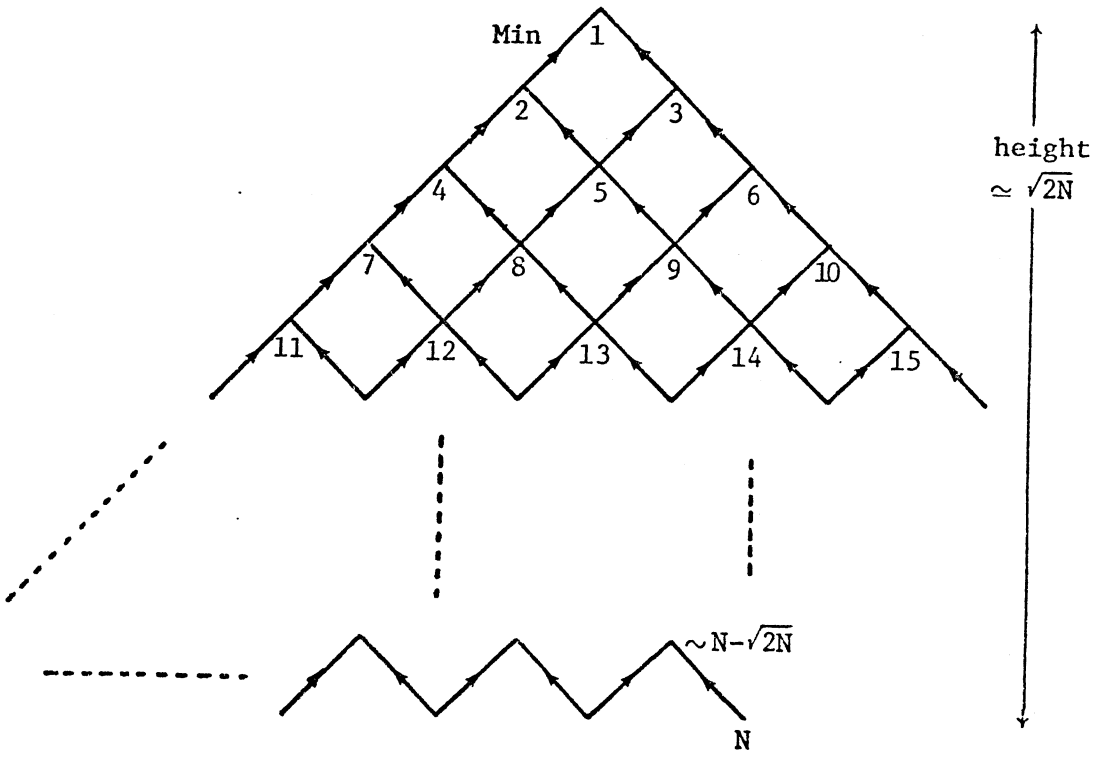


Figure 2.2.1 : A Beap.

To form the beap, the array is partitioned into roughly $\sqrt{2N}$ blocks. The i^{th} block consists of the i elements stored from position (i(i-1)/2 + 1) through position i(i+1)/2. This enables us to increase or decrease by 1 the size of the entire structure while changing the number of elements in only one block. (Indeed, this and similar blocking methods

are used in all of the structures we present.) The ordering imposed on this structure is that the k^{th} element of the j^{th} block is less than or equal to both the k^{th} and the $k+1^{st}$ elements of the block $j+1^{st}$ block. This ordering (the beap property) is illustrated in Figure 2.2.1. The numbers in the figure denote the indices of the array and an arrow may be viewed both as a pointer and as a $\leq$ relation. The structure is then analogous to a heap except that an element in our structure will typically have two parents and the height of the structure is about $\sqrt{2N}$.

Taking a slightly different point of view, one can interpret this structure as the upper left triangle of a matrix (or grid) in which locations 1, 2, 4, 7 ... form the first column and 1, 3, 6, 10 ... the first row. Furthermore, each row and each column is maintained in sorted order. Under this interpretation, the element in position (i,j) of the beap is actually stored in location $P(i,j) = 1/2\ ((i+j-2)^2 + i + 3j - 2)$ of the array. This is, in fact, the well-known diagonal pairing function. In the interest of brevity we will present several of our algorithms in terms of traversing rows and columns. Note that these manipulations can easily be performed on the structure without the awkward repeated computation of the inverses of P(i,j), and in general without the explicit evaluation of P(i,j) at each node visited.

A practical way to create the beap is by sorting the entire list. It will be shown in Chapter 3 that (N/2) log N (minus lower order terms) comparisons are necessary for its creation, and that (N/2) log N plus lower order terms are sufficient. Obviously a good sorting algorithm requires a number of comparisons within a factor of 2 of these bounds. Sorting has the practical advantage of simplicity and in fact of lower total cost than the scheme outlined in Chapter 3.

We now describe methods for performing a few basic operations on this structure.

1. Finding the minimum:

   This element is in the first location.

2. Finding the maximum:

   The maximum is in one of the last $\sqrt{2N}$ locations, all of which must be examined.

3. Insertion:

   Insertion is performed in a manner analogous to insertion into a heap. A new element is inserted into location (N+1) of the array. If this element is smaller than either of its parents, it is interchanged with the larger parent. This sifting-up process, analogous to the heap sifting, is continued until the element is in a position such that it is larger than both of its parents. Since the height of the structure is roughly $\sqrt{2N}$, one sees that at most $2\sqrt{2N}$ comparisons and $\sqrt{2N}$

swaps are performed. Furthermore, we note that if we are to insert a new element into the structure, and this element is smaller than all elements currently stored, then every element on some "parent-child" path from location 1 to one of the last $\sqrt{2N}$ locations must be moved. This condition holds regardless of the insertion scheme employed. Hence the scheme outlined minimizes the maximum number of moves performed in making an insertion into a beap.

4. Deletion:

Once we have determined the location of the element to be removed, simply move the element in position N of the array to that location. This element then filters up or down in essentially the manner used for insertions. Thus, the cost for a deletion is at most $2\sqrt{2N}$ comparisons and $\sqrt{2N}$ swaps.

5. Search:

For this operation, it is convenient to view the structure as an "upper-left" triangular matrix (see Figure 2.2.2). We start searching for an element, say x, at the top right corner of the matrix. After comparing x with the element under consideration, we will do one of the following depending upon the outcome of the comparison.

(i) If the element is too large, move left one position along the row.

(ii) If the element is too small, either move down one position along the column or, if this is not possible because we are on the diagonal, then move diagonally left and down one position.

(iii) If the element is equal to x, stop searching.

Repeating the above process, the search path will eventually terminate successfully or meet with the left side of the triangle and be unable to move. The latter condition indicates an unsuccessful search. Thus, the cost for a search is at most $2\sqrt{2N}-1$ comparisons.

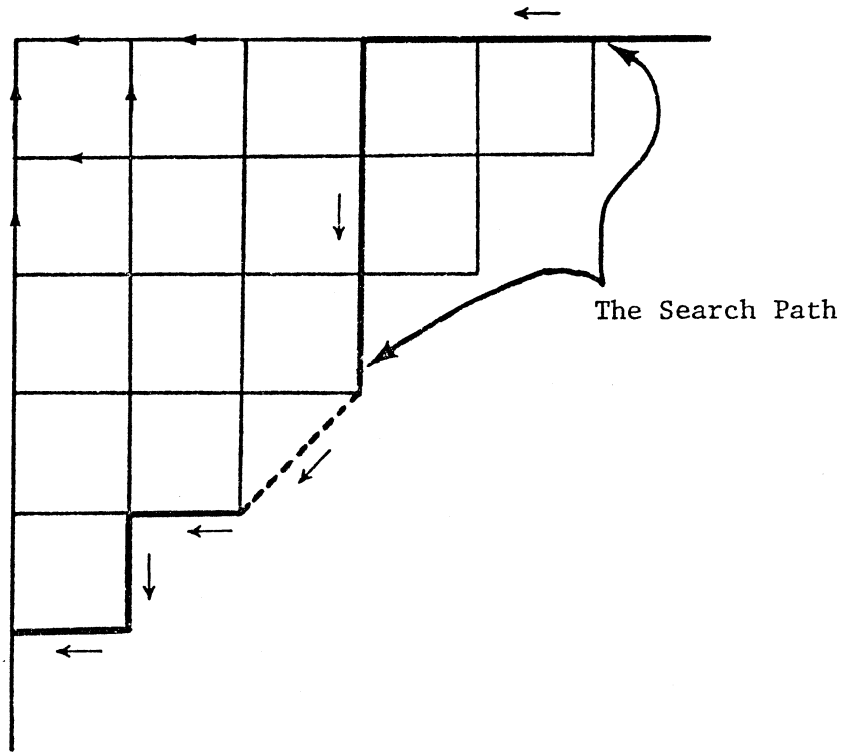


Figure 2.2.2 : A Search Path through a Beap.

The following lemma shows that this is the best search technique on beap.

Lemma 2.2.1:

In the worst-case $2\sqrt{2N}-1$ comparisons are necessary to search for an element in a beap.

Proof:

Consider the right most diagonal and super-diagonal of the structure (Figure 2.2.3). Suppose the diagonal contains the largest elements in the structure and the super-diagonal contains elements smaller than those on the diagonal but larger than any others in the rest of the structure. Note that no other information about the relative values of these elements need be known. Suppose that we are searching for an element known to be smaller than all (except perhaps one) of the elements on the diagonal, but larger than all (except perhaps one) on the superdiagonal. There is no choice but to inspect all elements in both of these blocks. □

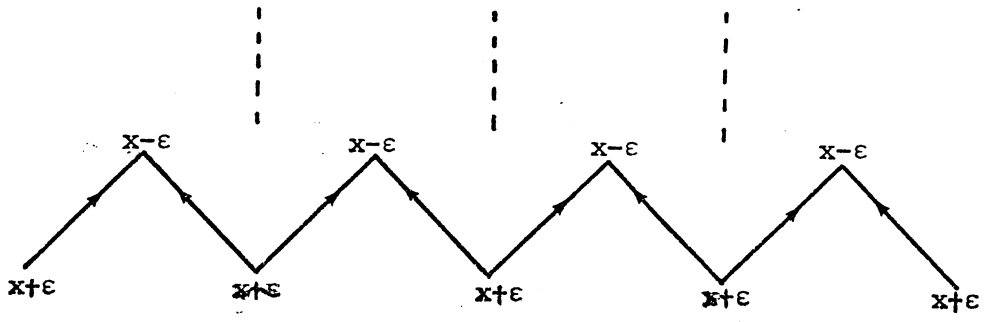


Figure 2.2.3 : Diagonal and Superdiagonal Containing Elements Near x.

In summary, we have demonstrated the following:

Theorem 2.2.2:

By storing N data elements in an array of length N ordered as a beap, retaining no additional information other than the value N, it is possible to maintain a data structure under which insertions and deletions can be performed in $2\sqrt{2N}$ comparisons and $\sqrt{2N}$ swaps, and searches in $2\sqrt{2N}-1$ comparisons.

2.3 Lower Bounds

Compared to the effort which has been devoted to developing good algorithms, little has been done on proving lower bounds for the complexity of set manipulation problems. Under certain restrictive models, however, some results have been obtained [Knuth73b, Snyder77, Tarjan77, Yao78]. A lower bound for the dictionary problem, if the dictionary is represented as a unique generalized search-tree structure, has been explored by Snyder. He shows that at least one of the operations insert, delete, or search requires $\Omega(\sqrt{N})$ comparisons or changes to pointers or data. Snyder's model differs somewhat from ours in that he assumes the use of explicit pointers in his representation. This implies, for example, that if the maximum element in a sorted list is to be replaced by one smaller than any of the elements in the list, only two pointer changes have to be made to restore the ordering. Under an implicit ordering, every element would have to be moved to preserve this property. In this sense, Snyder's lower bound can be construed as somewhat stronger than necessary for our purposes. On the other hand, he assumes the structure is effectively stored as a tree of bounded degree, and in that sense his lower bounds are too weak. We can however demonstrate the same lower bound as Snyder $\Omega(\sqrt{N})$ for the class of implicit data structures which are based solely on storing data in

some fixed partial order. This result is related to but incomparable with his. (Observe that the structures discussed in the preceding sections are based on a fixed partial order and thus are subject to this bound.)

For simplicity let us assume that we are to perform the basic operations of search and modification (a deletion followed by an insertion). We insist on pairing a deletion with an insertion only to eliminate the problem of the structure changing its size.

Theorem 2.3.1:

If the only information retained about an implicit data structure, other than N, the number of elements it contains, is a fixed partial order on the locations of the array. Then, the product of the maximum number of comparisons necessary to search for an element and the number of locations from which data must be moved (swapped) to perform a modification is at least N.

Proof:

Consider the directed acyclic graph (Hasse diagram) representing the partial order underlying the storage scheme. Let S be the largest independent set in this graph. Since the only information on which we can rely is that the elements are stored according to known partial order, it is quite possible that the values stored in the locations corresponding to S are of consecutive ranks among the values

stored in the structure. Hence in searching for a value in this range, no comparisons with any elements outside S can remove from consideration any in S. Therefore, the number of elements in S is a lower bound for the number of comparisons necessary to perform a search on the structure.

Now suppose the elements of the longest chain, C, are as small relative to the other elements in the structure as is consistent with the partial order. This means that if there are k elements anywhere in the structure which must be smaller than a given element in the chain, then the particular element is the $k+1^{st}$ smallest in the structure. Now suppose we are to replace the smallest element in C (which is the smallest element in the structure) with one greater than the largest element in C. This implies that each element in the chain is in a position requiring more specific locations be occupied by elements smaller than the given element than there exist in the entire structure. Hence, every element on the given chain must be moved (including the minimum element). Indeed, |C| swaps are required. The theorem now follows since the product of the length of the longest chain and the size of the largest independent set must be at least N (see for example, [Bondy76]). □

Corollary 2.3.2

$\Theta(\sqrt{N})$ swaps or comparisons are necessary and sufficient to perform the operations insert, delete, and search on an implicit data structure in which the only information known about the structure is a fixed partial order on the array locations and the size of the structure.

It is perhaps worth noting that a beap, which is a two dimensional grid, gives the best balance we can obtain between the cost for searching and the cost for insertion/deletion (both take $O(\sqrt{N})$ time). By going to a one dimensional grid (a sorted list), insertion and deletion will become more expensive and searching will become cheaper. Going to the obvious extension in three dimensions (see Fig. 2.3.1) or more reduces the cost of modification at the expense of retrieval cost.

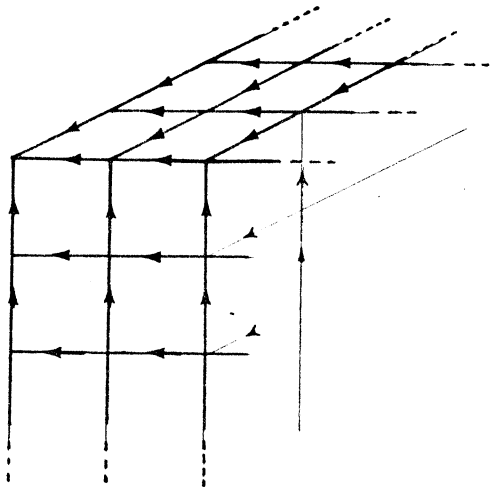


Figure 2.3.1: Three dimensional Beap.

A related result on lower bounds has been obtained by Borodin et al. [Borodin79]. They consider the cost P(N) of creating a partial order on N elements such that a search can be performed in S(N) comparisons. They show :

Lemma 2.3.3.

$$P(N) + N \log(S(N)) \geq (1 + o(1)) N \log N$$

This result complements Theorem 2.3.1 rather nicely. Combining the two and letting DI(N) to denote the number of moves required to make a delete/insert pair on a structure containing N elements, we have a lower bound on the cost P(N) such that a search can be performed in at most S(N) comparisons and a deletion/insertion pair can be done in DI(N) moves. This is shown in the following corollary :

Corollary 2.3.4:

$$P(N) + 2N \log(S(N)) + N \log(DI(N)) \geq (2 + o(1)) N \log N$$

Proof :

By Theorem 2.3.1, we get $S(N) * DI(N) \geq N$

or $\log(S(N)) + \log(DI(N)) \geq \log N$.

The corollary now follows from lemma 2.3.3. □

Looking at the proof of Lemma 2.3.3 (Lemma 4 in [Borodin79]), we note that P(N) is indeed the information theoretic lower bound on the number of comparisons required to construct a partial order. This lower bound can be ex-

pressed as log N! - log(#(N)). Combining this fact with the previous corollary, we get an interesting combinatorial view of the corollary.

Corollary 2.3.5:

$2\ N \log(S(N)) + N \log(DI(N)) - \log(\#(N)) \geq (1 + o(1))\ N \log N.$

2.4 Structures Not Using a Fixed Partial Order

In this section, we present several implicit (and "semi-implicit") structures under which the product of search time and insert (or delete) time is less than N. The main trick employed is to store each block of elements in an arbitrary cyclic shift from increasing order. We then call such a block "rotated list". An example of a rotated list is given in Figure 2.4.1.

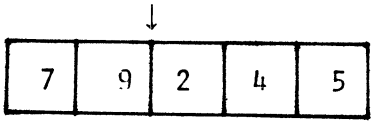


Figure 2.4.1: A Rotated List of Size 5.

2.4.1 Rotated lists

Again the array is partitioned into blocks such that the i^{th} block contains i elements. The order maintained is much more stringent than that of a beap. We insist that

(i) all elements in block i be less than or equal to all elements in block i+1;

(ii) block i is a rotated list of size i.

An example is shown in Figure 2.4.1.1.

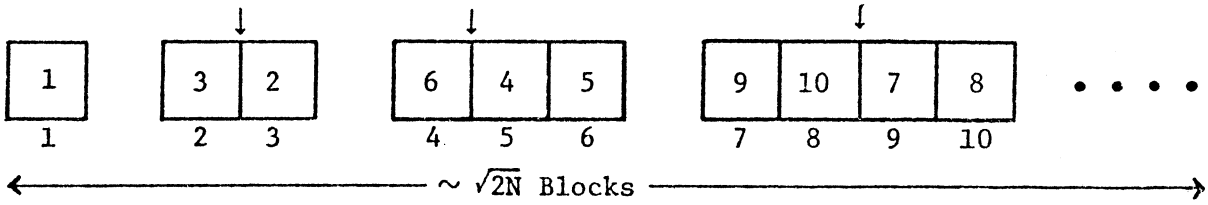


Figure 2.4.1.1: Each Block is a Rotated List.

Before discussing searches, it is helpful to describe a technique for finding the minimum of a rotated list containing distinct elements. The method used is a bisection technique. Figure 2.4.1.2 illustrates the problem of determining the minimum value in the interval [F,L]. A modified binary search procedure is described below in Pidgin-Algol :

```
procedure Block_Min (F, L)
begin
   if (L-F+1) ≤ 2 then
      return (min ( el[F], el[L])
   else begin
      M = | (F+L) / 2 |
      if el[M] < el[L] then Block_Min(F,M)
      else
         if el[M] > el[L] then Block_Min(M,L)
   end
end
```

el[M] denotes the element in position M of the el.

Figure 2.4.1.2 : Finding the Minimum in a Rotated List.

Lemma 2.3.1.1 :

If the size of the block, b, is one more than a power of two, then the procedure Block_Min requires log(b-1) + 1 comparisons. In general, at most $\lceil \log(b-1) + 1 \rceil$ comparisons are required.

Proof :

The number of comparisons satisfy the following recurrence relations :

$$T(2^x + 1) = T(2^{x-1} + 1) + 1$$

$$T(2) = 1$$

It is easy to show by induction that x+1 is the solution. For $b=2^x+1$ this yields log(b-1) + 1. □

It should be noted that the above procedure does not work if equal keys are present. For example, suppose all elements in a block are equal except the minimum of the block. In the worst case we have to use (b-1) comparisons to find the minimum. One way to overcome this problem is by adding a flag (one bit) to each array location. The flag is on if the key (element) occurs more than once in the structure. This also indicates that the number of occurrences is stored in the next location to the right (logically, not physically) of the key. The flag in the count location is also on. With little modification, the algorithm explained below can still be applied. The extra cost will not affect the total search costs dramatically. While this technique can save space if the frequency of occurrence of a key is large, the structure has to be classified since semi-implicit, since it needs an extra O(N) bits for the flags. The procedures given below operate on the assumption that there are no repeated keys.

We now describe methods for performing basic operations on rotated lists.

1. Finding the minimum.

   Again, this element is in the first array location.

2. Finding the maximum:

   The maximum is in one of the locations in the last block. By using a modified binary search, we can find the minimum element of that block; the maximum is in the immediately preceding location (or in location N, if the block minimum is in the first location within the block). Hence only $\lceil \log(\sqrt{2N}-1) \rceil + 1$ comparisons are required.

3. Search:

   Our basic approach is to perform a simple binary search until it is determined that the desired element lies in one of two consecutive blocks. Next the procedure Block-Min is performed to determine the minimum element in the larger block. Based on the outcome of a comparison between this minimum and the desired element, we either

   (i) perform a (cyclicly shifted) binary search on the larger block

or (ii) determine the position of the minimum element in the smaller block and perform a binary search on that block.

Lemma 2.4.1.1

In the worst case, searching requires at most 2 log N + O(1) comparisons.

Proof:

Let k and k+1 be the sizes of the two consecutive blocks. The number of comparisons in finding these two blocks is at most (log N - log k). Locating the minimum value in the larger block requires at most log k + 1 comparisons. Completing the search requires at most log k + 1 + log k comparisons. Thus, the entire process costs at most log N + 2 log k + 2 comparisons. This again is bounded by 2 log N + 3 comparisons. □

An alternate search technique with the same worst-case behaviour is to first do a binary search on the blocks to find two consecutive blocks that may contain the desired element and then proceed as before.

4. Insertion:

Using the basic strategy suggested for performing a search, the block into which a new element should be inserted can be found in log N + O(1) comparisons. A further $\sqrt{2N}$ (at most) moves suffice to insert the new

element into its proper position, remove the block maximum and shift the elements which lie between the new element and the former location of the block maximum. At this point, we see that for each block larger than the one in which the insertion was made, we must perform the operation of inserting a "new" minimum element and removing the old maximum. Fortunately, the new minimum can simply take the place of the old maximum and no further shifting (within blocks) is necessary. This transformation can be performed on a block of i elements in log i + O(1) comparisons and one swap. Thus, the entire task can be accomplished in about

$$\sum_{i=1}^{\sqrt{2N}} (\log i + O(1)) = \sqrt{(N/2)} \log N + O(\sqrt{N}) \text{ comparisons}$$

and $O(\sqrt{N})$ moves in the worst-case.

5. Deletion:

Deletions are performed in essentially the same way as that outlined for insertions.

6. Selecting the k^{th} smallest:

The k^{th} smallest element must lie in block $\lceil(\sqrt{1+8k} - 1)/2\rceil$. Determining the minimum element in this block, and so the value of the desired element, requires about 1/2 log k comparisons.

Summarizing the above results and observing that once the structure is formed, $O(\sqrt{N} \log N)$ comparisons are sufficient to complete a sort of the list, we have shown:

Theorem 2.4.1.2

Storing N different data elements in an array of length N, and retaining no information other than the data and the value of N, it is possible to perform searches in 2 log N comparisons and insertions and deletions in $\sqrt{N/2} \log N + O(\sqrt{N})$ comparisons and swaps. $N \log N - \Theta(N)$ comparisons are necessary and sufficient to create this structure.

2.5 Improving Insertion Time with Extra Storage

At this point, it is natural to ask whether or not we can simultaneously achieve the O(log N) search time of the rotated lists and the $O(\sqrt{N})$ modification cost of the beap. In this section we show that this is possible with o(N) additional storage.

2.5.1 With $\sqrt{2N}$ Pointers

Observe that the $O(\sqrt{N} \log N)$ behaviour of the above technique is due to the search, in each block, for the (local) maximum. By retaining a pointer to the maximum of each block, the insertion and deletion times are reduced to $O(\sqrt{N})$. Furthermore, the problem with equal keys will no longer exists.

2.5.2 Batching the Updates with a Small Auxiliary Map

Another approach is to "batch" insertions and deletions. This can be accomplished with an insertion map and a deletion map. Each map can hold up to log N pointers (indices) to the array to indicate the elements to be inserted or deleted. A total of $O((\log N)^2)$ bits are required for these maps; up to log N extra locations in the array may also be used for elements which have already been deleted.

We now describe the basic operations.

1. Insertion:

   Search the deletion map to see if the "new" element is actually in the array but to be deleted. If this is the case, we just remove the corresponding entry in the deletion map. Otherwise, we put the key in location N+1, incrementing N at the same time, and keep a new pointer to this location in the insertion map.

2. Deletion:

   Search the insertion map to see if the element to be deleted is newly inserted. If this is the case, remove the entry in the insertion map and the element in the array, close the gaps created by this removal, and decrement N. Otherwise, put a pointer to the key to be deleted in the deletiom map.

3. Search:

   We search the maps first. If there is an entry pointing to the desired element in the array, the answer will depend on where the pointer is. Otherwise, we search on the structure by using the bisection method described in the previous section. The case of equal keys has to be handled separately.

When either map is full, we sort each map according to the newly inserted and deleted keys in the array and then, make the changes in a single pass. This restructuring will re-

quire $O(\sqrt{N} \log N)$ operations and can be "time-shared" with the next log N insert/delete commands.

At this point, one is also inclined to ask whether or not both the search and modification costs may be reduced to below $O(\sqrt{N})$. The answer is in fact positive. This can be achieved by combining the ideas of a beap and the rotated sorted list as described in the next section.

2.6 A Beap of Rotated Lists

In this section we will present a structure of N elements which allows us to perform a search, a deletion or an insertion in $O(N^{1/3}\log N)$ comparisons and swaps. Again the array is partitioned into blocks. The i^{th} block is stored from location $(i-1)i(2i-1)/6 + 1$ through location $i(i+1)(2i+1)/6$, and is divided into i consecutive subblocks containing i elements each. The subblocks correspond to the elements (nodes) of the beap and each subblock is stored as a rotated list. Note that the height of the structure is about $(3N)^{1/3}$, which is equal to the number of blocks. More precisely,

(i) each subblock of block i is a rotated list of size i

(ii) all the elements of the k^{th} subblock of the j^{th} block are less then all elements of the k^{th} and $k+1^{st}$ subblocks of block $j+1$.

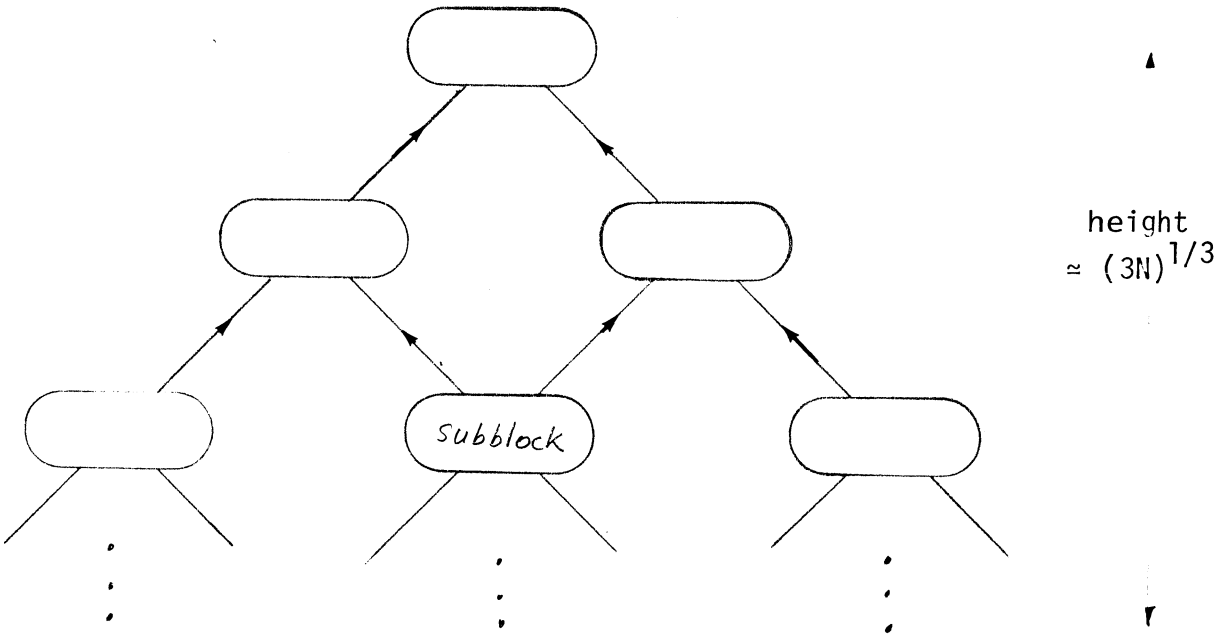


Figure 2.6.1: A Beap of Rotated List.

As in the case of a beap, we note that moving along rows and columns from subblock to subblock can be done easily without computing the pairing function and its inverses, provided three or four parameters are kept. We now describe how to perform these operations:

1. Finding the minimum:

   Again this element is in position 1.

2. Finding the maximum:

   The maximum is in one of the last $(3N)^{1/3}$ subblocks. The maximum element in a subblock can be found in $(\log N)/3 + O(1)$ comparisons, so the maximum element in the entire structure can be found in $(N/9)^{1/3} \log N + O(N^{1/3})$ comparisons.

3. Insertion:

   By combining methods for the grid and the rotated lists, we insert the new element into the $N+1^{st}$ position of the array, which is part of a subblock. Since the subblock is cyclicly sorted, about $2/3 \log N$ comparisons and $(3N)^{1/3}$ swaps are required in the worst case to insert the new element into the subblock. As in a beap, if the new element is smaller than either of the maximum elements of its parents, it is interchanged with the larger one. This process continues (as in a beap) until the imposed ordering is restored. Since the height of the structure is about $(3N)^{1/3}$, $O(N^{1/3} \log N)$ comparisons and swaps are in fact used.

4. Deletion:

   Similar to insertion.

5. Search:

Searches are performed in a manner similar to that described for the beap. The key difference is that a comparison with a single element in the grid is replaced by a (modified) binary search to find the minimum (and the maximum) of a subblock, and hence the decision to move left along the row or down along the column will need more comparisons. Again, we start searching for an element, say x, at the top right corner subblock of the matrix. After finding the minimum and maximum of the subblock under consideration, we will do the following:

(i) If x is less than the minimum element, move left one subblock along the row.

(ii) If x is larger than the maximum element, move down one subblock along the column, if this cannot be done (on diagonal) then move left and down one subblock in each direction.

(iii) If x is in the range of this subblock, then do a binary search to find x. If successful, then stop searching; otherwise, move left and down one subblock in each direction.

This process is repeated until either x is found or the required move cannot be made. In this manner a search can be performed in $O(N^{1/3} \log N)$ comparisons.

Theorem 2.6.1

Storing N different data elements in the first N locations of an array and retaining, in addition to data, only the current value of N, it is possible to perform each of the operations insert, delete and search in $O(N^{1/3} \log N)$ comparisons and swaps.

We note that the easiest way to initialize the structure is to sort the N elements of the array. The cost of doing so is within a (small) constant factor of that of optimal method. Again, the case of equal keys has to be handled with more care.

CHAPTER 3

ON FORMATION OF A BEAP AND SELECTION

The complexity of creating a beap is explored in this chapter. A lower bound based on the enumeration of beaps is derived in Section 3.1. An algorithm for forming a beap is given and analyzed in Section 3.2. This algorithm, which is based on the mode-finding algorithm [Dobkin79a], has a running time within a lower order term of the derived lower bound. A method for selecting the k^{th} smallest element on a beap is suggested in Section 3.3. It is similar to the algorithm for selecting the k^{th} element in X + Y, i.e. finding the k^{th} smallest element of $\{ x_i + y_j \mid x_i \in X, y_j \in Y, i=1,n, j=1,n\}$ [Johnson78]. It is shown that O(min(k log k, $\sqrt{N}$ log N)) are sufficient to do the selection. It should be noted that all algorithms in this chapter will need temporary working storage in addition to the storage required by the data.

3.1 A Lower Bound on Beap Formation

First we introduce the closely related notion of a Young Tableau [Knuth73b, pp. 48-67]. A Young Tableau of shape (n1,n2,...,nm), where $n1 \geq n2 \geq \ldots \geq nm \geq 0$, is an arrangement of n1+n2+...+nm distinct integers in an array of left-justified rows such that

(i) row i contains ni elements

(ii) the rows and columns of the array are sorted in increasing order.

For example, the following is a Young Tableau of shape (3,3,1):

1 3 6

2 5 7

4

A complete beap can be viewed as a Young Tableau of shape (m, m-1 , ... ,1). If the beap is not complete, it is of shape (m, m-1, ..., k, k, ..., 1). Each element in a Young Tableaux is associated with a hook, which consists of the element itself, those lying to its right, and those directly below it. An example is illustrated in Fig. 3.1.1.

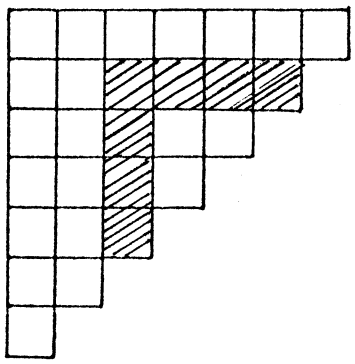


Figure 3.1.1 : A Hook of length 7.

The number of different ways to put N distinct integers in a Young Tableau of a given shape is related to the lengths of the hooks. This relation is shown in the following theorem taken from [Knuth73b, pp. 62-63].

Theorem 3.1.1:

The number of different Young Tableaux of a specific shape on N distinct integers is N! divided by the product of the hook lengths.

Since a beap is a special case of a Young Tableau, a formula for the number of different beaps on N distinct integers is easily found.

Lemma 3.1.2:

The number of Young Tableaux of shape (m, m-1, ..., 1) on N elements is #(N)

$$= \sqrt{\frac{2\pi}{A}}\ \frac{e^{(m^2/4)+(m/2)+(7/24)}}{m^{m^2+(3/2)m+(17/24)}}\ m^{(m^2/2)+(m/2)+(23/24)}\ (1+O(1/m))$$

where $m = -1/2 + \sqrt{2N}\ (1 + O(1/N)) \approx \sqrt{2N}$

and $A = 1.2824271...$ is the Glaisher's Constant.

Proof:

We need the following facts:

(i) $1 * 3 * 5 * \ldots * (2m-1) = (2^m)\ (\Gamma(m+1/2))/\Gamma(1/2)$.

(ii) $G(m) = 1^1 * 2^2 * \ldots * m^m$

$= A * m^{(m^2+m+1/6)/2} * e^{-(m^2/4)}\ (1+O(1/m))$

[Knuth73a, pp. 112 & 499].

(iii) $G(2m) = A\ (2m)^{(4m^2+2m+1/6)/2}\ e^{-m^2}\ (1+O(1/m))$.

(iv) $\sqrt{G(2m)} = \sqrt{A}\ (2m)^{(4m^2+2m+1/6)/4}\ e^{-(m^2)/2}\ (1+O(1/m))$.

(v) $\ln \Gamma(z) = (z - 1/2) \ln z - z + 1/2 \ln (2\pi) +$

$1/(12z) - 1/(360z^3) + O(1/z^5)$.

By Lemma 3.1.1, we get

$$\#(N) = N! \,/\, 1^m * 3^{m-1} * \ldots * (2m-1)^1$$

$$= N! \sqrt{\frac{1^1 * 3^3 * \ldots * (2m-1)^{2m-1}}{[1 * 3 * \ldots * (2m-1)]^{2m+1}}}$$

But,

$$1^1 * 3^3 * \ldots * (2m-1)^{2m-1} = \frac{G(2m)}{2^2 * 4^4 * \ldots * (2m)^{2m}}$$

$$= \frac{G(2m)}{1^2 * 2^4 * \ldots * m^{2m} * 2^{2+4+\ldots+2m}}$$

$$= \frac{G(2m)}{(G(m))^2 \, 2^{m(m+1)}}$$

Hence,

$$\#(N) = N! \sqrt{\frac{G(2m)}{(G(m))^2 \, 2^{m(m+1)}} \cdot \frac{(\Gamma(1/2))^{2m+1}}{2^{m(2m+1)} \, \Gamma(m+1/2)^{2m+1}}}$$

$$= N! \; \frac{\sqrt{G(2m)} \; (\Gamma(1/2))^{m+1/2}}{2^{(3m^2+2m)/2} \, G(m) \, (\Gamma(m+1/2))^{m+1/2}}$$

$[\Gamma(m+1/2)]^{m+1/2}$ can be computed as follows:

By (v), $\ln \Gamma(z) = (z-1/2) \ln z - z + 1/2 \ln (2\pi) +$

$$1/(12z) - 1/(360z^3) + O(1/z^5).$$

$$(\Gamma(z))^z = z^{z(z-1/2)} \, e^{-z^2} \, (\sqrt{2\pi})^z \, e^{1/12} \, e^{(-1/(360z^2) + O(1/z^4))}$$

$$= z^{z(z-1/2)} \, e^{-z^2} \, (\sqrt{2\pi})^z \, e^{1/12} \, (1 - 1/(360z^2) + O(1/z^4)).$$

Substituting $z = m+1/2$

$$(vi)\qquad [\Gamma(m+1/2)]^{m+1/2} = (m+1/2)^{m(m+1/2)}\, e^{-(m+1/2)^2+1/12}\, (\sqrt{2\pi})^{m+1/2} * (1 - 1/360(m+1/2)^2 + O(1/m^4))$$

It can be shown that

$$(m+1/2)^{(m+1/2)m} = m^{m(m+1/2)}\, e^{(m/2)+1/8}\, (1+O(1/m)).$$

Thus,

$$[\Gamma(m+1/2)]^{m+1/2} = m^{m(m+1/2)}\, e^{(m/2)+1/8}\, e^{-(m+1/2)^2+1/12} * (\sqrt{2\pi})^{m+1/2}\, (1 + O(1/m)).$$

N! must now be expressed as a function of m.

By (v):

$$\ln(N!) = (N+1/2)\ln N - N + \ln\sqrt{2\pi} + 1/12N - 1/(360N^3) + O(1/N^5)$$

$$= \frac{m(m+1)+1}{2} \ln \frac{m(m+1)}{2} - \frac{m(m+1)}{2} + \ln\sqrt{2\pi} + O(1/m^2)$$

$$N! = [(m.m(1+1/m))/2]^{(m^2+m+1)/2}\, e^{-m(m+1)/2}\, \sqrt{2\pi}\, (1+O(1/m^2))$$

It can be shown that

$$(1+1/m)^{(m^2+m+1)/2} = e^{m/2+1/4}\, (1+O(1/m)).$$

This gives

$$(vii)\qquad N! = (m^2/2)^{(m^2+m+1)/2}\, (e^{m/2+1/4}\sqrt{2\pi})/e^{m(m+1)/2}\, (1+O(1/m)).$$

The lemma follows from (ii), (iv), (vi), and (vii). □

Corollary 3.1.3:

$$\log (\#(N)) = (N/2) \log N - N (3 - \log e)/2 + O(\sqrt{N} \log N)$$

Theorem 3.1.4:

In the worst-case, the minimum number of comparisons required to form a complete beap is

$$N/2 \log N - 3/2 N (\log e - 1) - O(\sqrt{N} \log N)$$

Proof:

The information theoretic lower bound on the number of comparisons to form an object on N elements having #(N) different possible arrangements is $\log N! - \log(\#(N))$. This gives the theorem. □

3.2 Forming a Beap

Although sorting an array to form a beap is very practical and efficient, it leaves the question of whether or not we can form a beap in about N/2 log N comparisons rather than the N log N - O(N) required on average for a full sort. We are interested in achieving an upper bound which differs from the lower bound of N/2 log N by at most a lower order term.

The method presented here is a modification of the mode finding algorithm of Dobkin and Munro [Dobkin79a], which is in turn based on the median finding algorithm of Blum et al. [Blum73]. The basic idea in forming a beap of size N is to partition the elements into blocks of sizes 1, 2, ..., $\sqrt{2N}$ such that every element in block i is less than any element in block (i+1). Although this is clearly a stronger restriction than that of a beap, we will see that the time required to form such a structure is not much more than the lower bound given in the preceeding section.

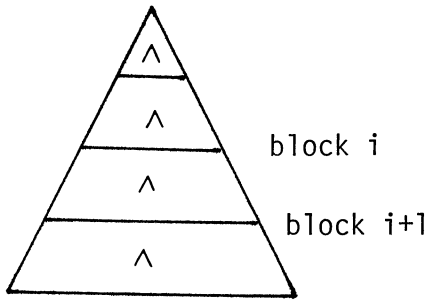


Figure 3.2.1: A Beap where each Element in Block i is smaller than any of Block i+1.

An adequate first approximation to the above situation can be achieved by selecting points, one in each block, and partioning the set of elements S about these points such that every element to the left or above each chosen point is smaller than any to the right or below the point. This is depicted in Figure 3.2.2.

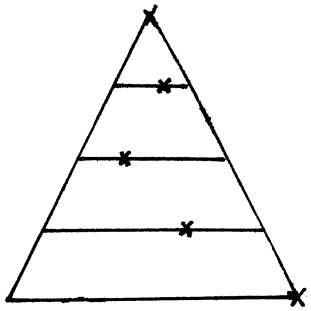


Figure 3.2.2: Elements are partitioned about the points.

The elements which are bounded by two chosen points form a segment. Thus, we would like to partition S into segments such that the endpoints of each segment reside in two consecutive blocks. These points and therefore the segments are found repetitively from an initial segment which contains all the elements, with the minimum and maximum of the elements as the endpoints. Having performed this rough partition, at most O(N) more comparisons are needed to achieve to a valid beap by performing one selection on each segment.

Following [Dobkin79a], the basic algorithm is given below; more details and analysis follow.

algorithm beap;

begin

Initially the entire list is a segment, H, with the minimum and maximum elements as its endpoints.

Split H into sublists of length $t=\lceil \log N \rceil$ and sort each sublist in increasing order. We will call each such sorted sublist a column.

while there is a segment H with end points not within two consecutive blocks (or the same block) do begin

Let |H| = h.

Using o(h) comparisons, find an element mid, such that at most $1/2 \pm o(1)$ of the elements of H are less than mid;

Split H into 2 segments H1 and H2, whose elements are < and > mid respectively. Mid is a new endpoint for both H1 and H2.

If the endpoints of H1 are not within consecutive blocks, repeatedly merge pairs of columns of H1 until the average column length is restored to about t. Do the same with H2.

end;

For each segment having endpoints in two consecutive blocks, partition the elements such that the smaller elements are moved to the smaller block and the larger ones are moved to the larger block. This is equivalent to finding 1^{st}, 3^{rd}, 6^{th}, 10^{th},... elements of the entire structure.

end.

We now describe and analyze the algorithm in more detail.

a) The Initialization:

Finding the minimum and maximum can be easily accomplished in 3N/2 comparisons. The sorting of the columns will require about N log log N comparisons.

b) The Splitting and Finding the Mid:

We follow the spirit of [Dobkin79a] in our description. Initially all columns are of the same length and we start by finding the median of the column medians and determine its rank with respect to all elements. Unfortunately, this splitting will only guarantee that the median of medians ranges from the 25^{th} to the 75^{th} percentile. Since this is too vague for our purpose, we try to get a better estimate by doing some more iterations. Inevitably, we will have to cope with columns or subcolumns of varying sizes. To be able to guarantee

that in each iteration at least 1/4 of the elements are disqualified from being the median, we will find the weighted median of the column medians as in [Dobkin79a, Johnson78]. Suppose c(1), c(2), ..., c(m) are the column medians and w(c(i)) is the weight (equal to the size of the column i) assigned to c(i). The weighted median $c(i_k)$ is the one which partition {c(i)} such that:

(ii) For any ordering $c(i_1)$, $c(i_2)$, ..., $c(i_k)$, ...,$c(i_m)$,

$$c(i_j) \leq c(i_k) \quad \text{for } j < k$$

$$c(i_j) \geq c(i_k) \quad \text{for } j > k.$$

(ii) $w(c(i_1))+ \ +...+ \ w(c(i_k)) \geq w(c(i_{k+1})) \ +...+ \ w(c(i_m))$

(iii) $w(c(i_1)) \ +...+ \ w(c(i_{k-1})) \leq w(c(i_k)) \ +...+ \ w(c(i_m))$

The weighted median can be found in time proportional to the number of elements [Dobkin79a, Johnson78].

The process to determine mid will itself be iterative. In each iteration, we find the weighted median of the sub-column medians of H that may contain the true median of H. All elements of H are partitioned about this weighted median by performing binary search on each column. This process requires O(h/t) + (h/t) log t comparisons, if the average length of the subcolumns is t. This leaves us with h/t sub-columns of various sizes, one of which contains the segment

median. The key point, of course, is that the number of elements in these subcolumns is at most about 3/4 of what it was on the previous iteration. After i iterations, the size of the subsegment containing the median is at most $h(3/4)^i$. Thus, after $\omega(1)$ iterations we can guarantee that we can produce two segments of sizes at most 1/2 + o(1) of the segment. By performing, say $O(t/(\log t)^2)$ (= $o(t/\log t)$) iterations, an adequate splitting is performed in o(h) comparisons. We refer to [Dobkin79a] for more details of the splitting.

c) The Merging

The merging process to construct H1 and H2 is exactly the same as the merging process in [Dobkin79a], where subcolumns are merged into columns with average length about t. This is easily achieved by repeatedly merging the smallest pair of subcolumns until this condition is met. A nice but unnecessary side effect of this approach is that the length of all columns will be between t/2 and 2t. Although certain elements may be involved in many comparisons, the total number of comparisons used for reconstituting the columns of H1 and H2 will be at most h (see [Dobkin79a] for details).

d) The Cost Analysis

Constructing the first segment uses only O(N) comparisons while building and sorting the columns contribute N log t = N log log N comparisons to the total cost. For ease

of analysis, we first assume that mid is the median of the elements; we will adjust the analysis later. Each time the while loop is executed, it contributes h + o(h) comparisons to the total cost, if there are merges. While the number of loop executions is of interest, the size of the segments created plays a bigger role in determining the cost contributed by this loop. The splitting process can be nicely described by a binary tree whose internal nodes are tagged with the size of the segment under consideration, representing the merge cost. It should be noted that the leaves need not be tagged since merging is not required and they do not contribute any costs. An example of such tree is given in Fig. 3.2.1.

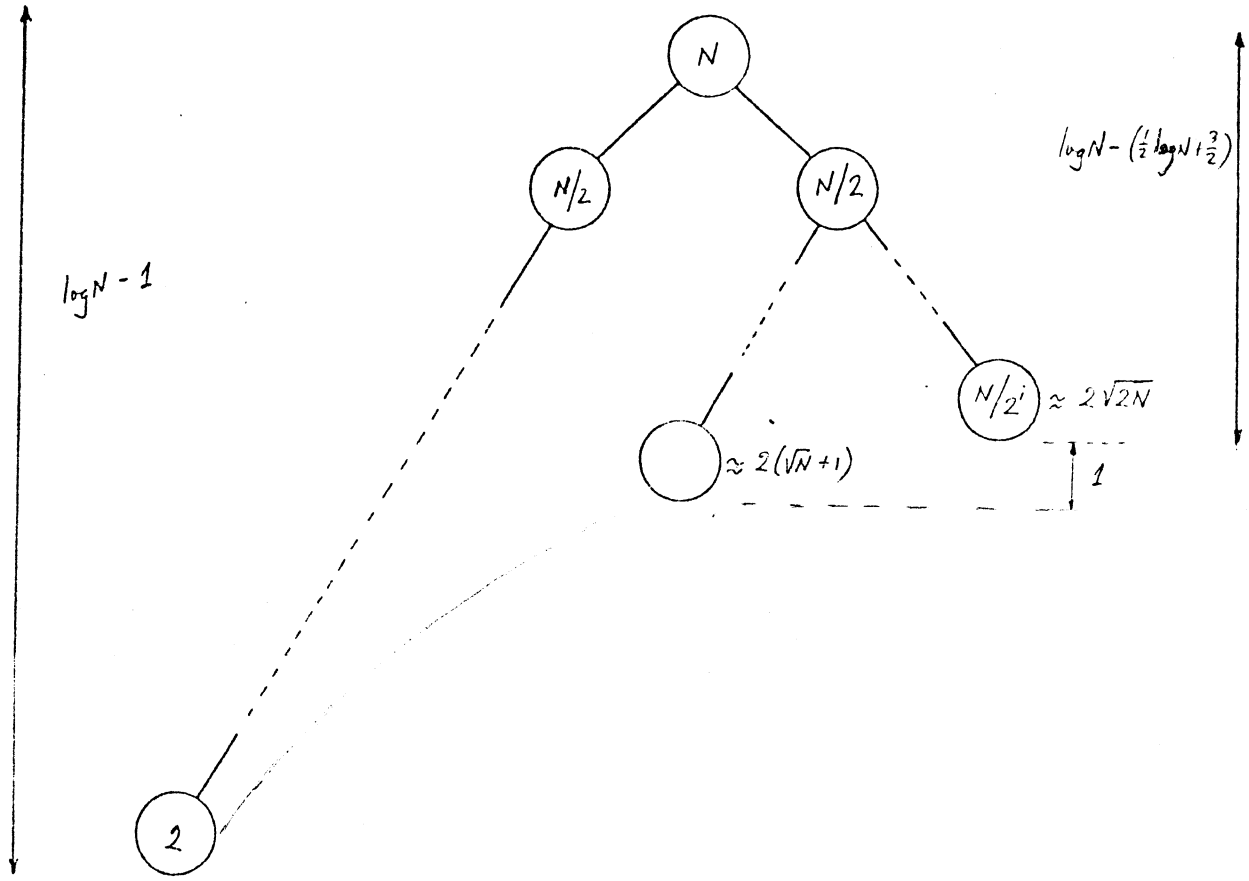


Figure 3.2.1 : A Tree Representing the Merge Cost Building a Beap.

The total merge cost would be the sum of all the numbers tagged in the internal nodes. Taking another view, we could express the sum roughly as the area shown below.

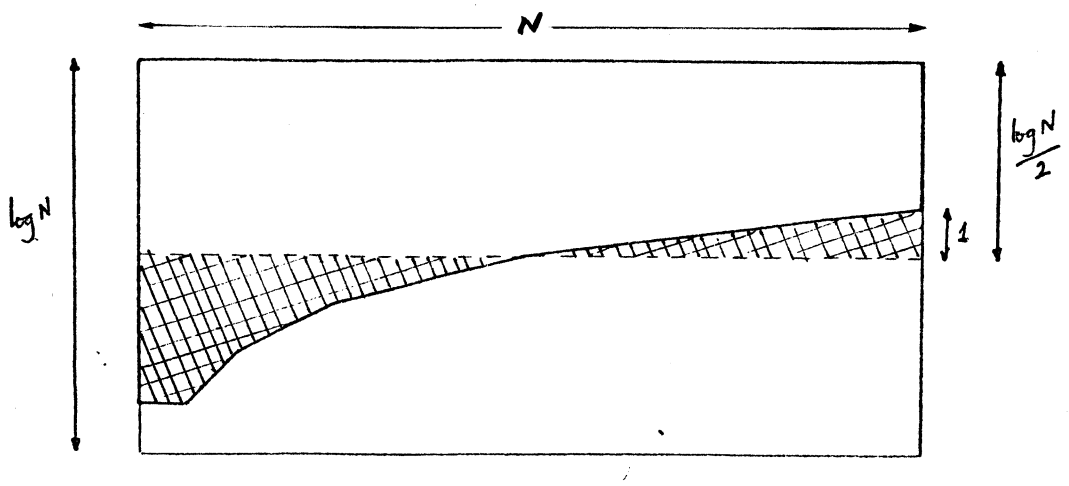


Figure 3.2.2 : Geometric View of the Merge Cost.

Since up to depth 1/2 log N - 1, every level of the tree sums up to N, we can represent the sum as a rectangle. From then on, the tree leans somewhat to the left. This effect is also depicted in Fig. 3.2.2. The shaded area on the right is N/4 and the shaded area on the left is

$$N/2\ (1/2 + 1/4 + 1/4 + 1/8 + 1/8 + 1/16 + \ldots)$$

$$= N/2\ (1/2 + 1/2 + 1/4 + 1/8 + \ldots) \leq 3/4\ N.$$

Thus, the total cost for the merging is bounded by 1/2 N log N + 1/2 N.

Since we cannot guarantee that mid is always the median, it are clear that more iterations are needed. But the crucial point is that the depth of each leaf in Fig. 3.2.1 will increase by at most o(log N), since :

$(1/2 + f(t))^x N \geq 1$

where x is the height of the tree and $f(t)=o(1)$.

Or, $\log N + x \log(1/2(1 + 2f(t)) \geq 0$

$x \leq \log N / (1-\log(1+2f(t)) \leq \log N (1+cf(t))$

for a constant $c \geq 2/(\ln 2 (1-2f(t)^2))$

Thus, $x \leq \log N (1+o(1))$.

Clearly, the while loop contributes $1/2 N \log N + o(N \log N)$ to the total cost which sums to

$1/2 N \log N + o(N \log N) + N \log \log N + O(N)$

$= 1/2 N \log N + o(N \log N)$.

The partitioning in the last statement of the algorithm can be done in O(N) comparisons. With Theorem 3.1.4, we have shown:

Theorem 3.2.1:

$1/2 N \log N + o(N \log N)$ comparisons are sufficient to build a beap of size N. Furthermore, $1/2 N \log N - O(N)$ comparisons are necessary.

3.3 Selecting the k^{th} Smallest Element

Another operation that can be performed rather efficiently on a beap is the selection of the k^{th} smallest element. Indeed, one expects some improvement over selection from an unordered set since the beap is almost as difficult to construct as it is to sort and, therefore, it contains information that can facilitate the selection process.

First, we present a simple method that works well for small values of k.

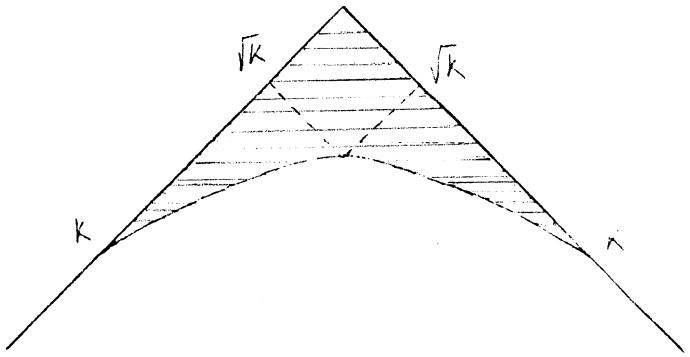


Figure 3.3.3 : Candidates For The k^{th} Smallest Element.

Lemma 3.3.1:

The k^{th} smallest element can be found in O(k ln k) comparisons.

Proof:

The k^{th} smallest element must lie in the shaded area of Fig. 3.3.3. The number of elements in this shaded area is bounded by $k + 2k \int_{\sqrt{k}}^{k} (1/x)\, dx = k + k \ln k$. Therefore, the k^{th} smallest element can be found in O(k log k) comparisons. □

While the above method is efficient for small values of k, it is very bad if k is large (e.g. k = O(N)). For this case, however, we can do better with a different algorithm presented below. It is similar to the algorithm for selecting the k^{th} element in X + Y [Johnson78].

Again, the basic method is based on the median finding algorithm. We arrange the elements in sorted columns and find the weighted median of the column medians and partition the elements around this value, removing at least 1/4 of the elements from consideration. The difference is that we have to deal with columns of varying size and that we can save some work by taking advantage of the structure.

Viewing the beap as a upper triangular matrix, we note that the columns are already in sorted order but have varying sizes. After finding the weighted median of the

column medians, we partition the elements by searching the beap for this weighted median from both the upper right corner and the lower left corner (see Fig. 3.3.4).

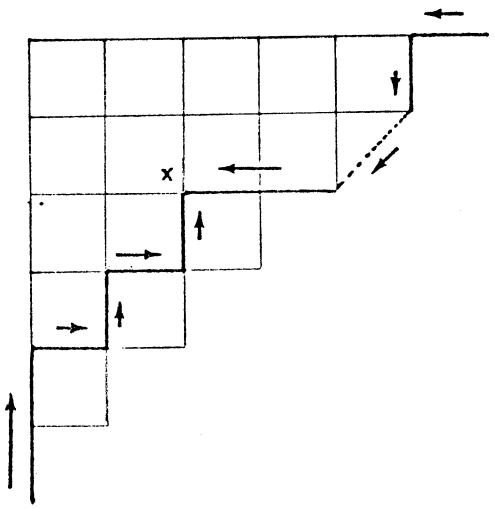


Figure 3.3.4 : The Search Paths Partitioning about the Element.

The search technique from the upper right corner has been described in Section 2.2.1. The dual search technique from the lower left corner can be described as follows:
After comparing the median of medians with the element under consideration, do one of the following:

(i) If the element is too large, move up one position along the row

(ii) If the element is too small, either move right one position along the row, or, if this is not possible (because we are on the diagonal), then move diagonally up and right one position.

(iii) If the element is equal to the median of medians, stop searching.

The paths generated by these two search processes will meet in the location containing the element being searched. This path, in fact, generates the partition about the median of medians. This process can be performed in $2\sqrt{2N}$ comparisons.

The algorithm can be written as follows:

```
Algorithm k^th smallest;
begin
    Initially S denotes the entire set of elements
    (|S|=N).
    Take the √2N sorted columns of the beap as our
    starting point.

    while |S| > √2N do
    begin
        find the weighted median of the column medians;
        partition S into S1 and S2 containing elements
        of S < and ≥ respectively;
        if |S1| ≥ k then S := S1
        else begin
            S := S2;
            k := k - |S1|

        end;

    end;

    Select the k-smallest element from S using a O(|S|)
    algorithm.
end.
```

The cost of one iteration of the while-loop is $O(\sqrt{N})$ and the while-loop is executed at most (O(log N) times, since each iteration removes at least 1/4 of the element. The selection in the last statement can be done in $O(\sqrt{N})$ comparisons. Thus, the total cost is bounded by $O(\sqrt{N} \log N)$ comparisons. We have shown the following theorem.

Theorem 3.3.2:

The k^{th} smallest element can be found in O(min(k log k, $\sqrt{N}$ log N)) comparisons on a beap of size N.

Recently Dobkin and Munro have improved the bound for selection on a beap to $O(\sqrt{N})$ comparisons [Dobkin79b]. Frederickson and Johnson [Frederickson79] have recently announced results similar to those of this chapter.

CHAPTER 4

PARALLELISM ON BEAPS

One way to increase speed in solving various problems is through parallelism. Of interest, for any given problem P, is the speed-up gained by the use of k processors. Let us define this speed-up factor as P_1 / P_k, where P_i is the worst-case complexity of P on i processors. Clearly, the maximum speed up factor achievable is k. While parallel algorithms for sorting and graph-theoretic problems have received much attention [Batcher68, Gavril75, Hirschberg78, Hirschberg79, Knuth73b, Preparata78, Regbati78, Valiant75], little has been done in devising dynamic data structures that allow parallelism. The main reasons are, probably, that serial techniques such as hashing and tree-structures are extremely efficient and there is an expectation of a low speed-up factor based on one of the few known bounds, a speed-up factor of only log(k+1) for searching a sorted list [Borodin72, Knuth73b]. This pessimism is, however, not justified for the structures we have been studying. As we have seen, because of a time-space trade-off, the performance of the basic operations for an implicit structure for the dictionary problem is rather slow compared to an explicit tree representation. The opportunity for utilizing parallelism is, then, much greater for implicit representations than for the explicit ones. We will show that this

optimism is justified by demonstrating a speed-up factor of about 2k/3 for performing the basic operations.

We will adopt Valiant's model of parallel computation [Valiant75]; k processors perform comparisons or swaps simultaneously on a shared memory. It should be noted here that in the algorithms proposed there will be no access conflicts as the processors will act on different parts of the memory.

Divide and conquer algorithms have been successfully applied in devising parallel algorithms for sorting [Hirschberg78, Gavril75, Preparata78, Valiant75]. Continuing that approach, we will first describe an alternate searching method for beaps based on divide and conquer.

4.1 A Divide and Conquer Technique for Searching

It is convenient first to describe a search procedure for finding an element, say x, in an arbitrary m by n rectangular array in which the rows and columns are maintained in increasing order. The following search procedure is then applied :

- Take the middle column (the $\lfloor n/2 \rfloor$-th) of the rectangular array and use binary search to find a pair of consecutive elements which are less and greater than x, respectively (see Fig. 4.1.1). The (roughly)

(mn/2) elements in the shaded area are discarded.

- The search is recursively continued on the two remaining structures until either m or n is reduced to 1. At this point, a simple binary search is employed.

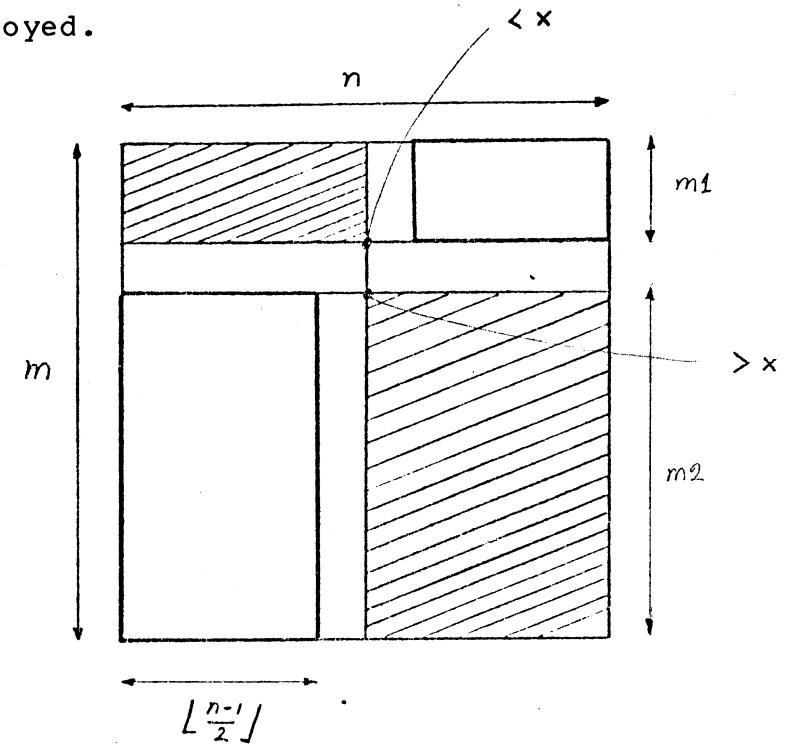


Figure 4.1.1 : Divide and Conquer Method.

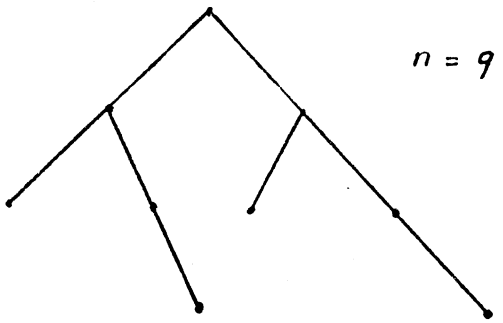


Figure 4.1.2 : Column Tree.

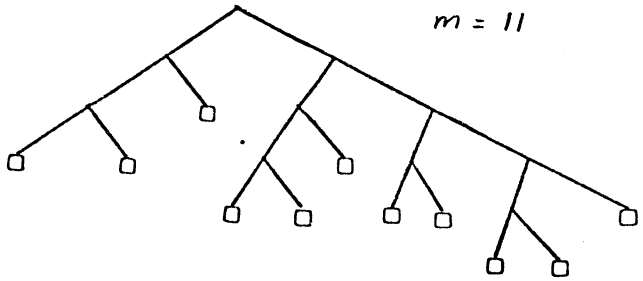


Figure 4.1.3 : Row Tree.

This process of divide and conquer can be nicely described by two binary trees, a column-tree and a row-tree. They describe the columns and rows splitting in the iterations. The column tree is fixed for a given n since the columns are always split into two equal parts. Actually, it corresponds to the process of binary search on n elements. Therefore, it is balanced, with the exception that some nodes on the next to the lowest level may have only one son, and the number of nodes in each subtree is equal to the number of column of the rectangle under consideration. As an example, the case n = 9 is depicted in Fig. 4.1.2. Unlike the column-tree, the row-tree describing the row splits does not have a fixed shape. It depends on how the rows are divided during the binary search of the middle columns. The number of leaves in each subtree is equal to the number of rows in the corresponding rectangle under consideration. An example of a row-tree is given in Fig. 4.1.3. In the first iteration, it is assumed that the rows are divided into two parts containing 3 and 8 rows respectively. Superimposing these trees, we can see that the splitting process terminates whenever the leaves of either tree are reached. This tells us that the number of either the rows or the columns of the rectangle under consideration is equal to one. Now binary search can be employed to search for the element x.

The number of comparisons in the worst-case to find an element in any m by n grid can be described by looking at the column- and row-trees or by the following recurrence relation:

$$T(m,n) = T(m1, \lfloor (n-1)/2 \rfloor) + T(m2, \lceil (n-1)/2 \rceil) + \lceil \log(m+1) \rceil$$

where $m1 + m2 = m$, $m,n \geq 2$, and

$$T(1,n) = \lceil \log(n+1) \rceil$$

$$T(m,1) = \lceil \log(m+1) \rceil$$

We first look at the solutions for some special cases.

Lemma 4.1.1:

For $m = 2^j$, $n = 2^i - 1$, $n \geq 2m - 1$ and assuming that $m1 = m2 = m/2$, (i.e. the rows are always partitioned in the middle), the solution of the above recurrence relation is

$$T(2^j, 2^i - 1) = 2^j(i-j) + 3*2^j - (j+3)$$

or

$$T(m,n) = m (\log ((n+1)/m))) + 3m - (\log m + 3)$$

Proof:

By induction on (j,i)

Basis : for (0,1) : $T(1,1) = 2^0 * 1 + 3 - 3 = 1$

Induction Hypothesis : Assume true for $(j1, i1) \leq (j,i)$

Induction Step :

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

(by induction hypothesis)

$= 2j(i-1-j+1) + 3*2^j - 2(j-1+3) + j + 1$

$= 2^j(i-j) + 3*2^j - (j + 3)$ □

For $n \leq 2m - 1$, a similar result is shown in the following lemma.

Lemma 4.1.2:

For $m = 2^j$, $n = 2^i - 1$, but $n \leq 2m - 1$ and assuming that $m1 = m2 = m/2$, the solution of the above recurrence relation is

$T(m, n) = n (\log (m/(n+1)) + 3) - \log (n + 1)$

Proof :

Similar to Lemma 4.1.1. □

It is interesting to note that the solutions of the recurrence relation under the above assumptions depend primarily on the smaller of the two sides of the rectangle. This justifies our search method of taking the middle column regardless of m and n. If $n = 2m - 1 = 2^i - 1$, the column and row-trees are equivalent and the solution of the recurrence relation under the above assumptions is

$T(m,n) = 2n - \log(n+1) = 4m - \log m - 3 \leq 3\sqrt{mn}$.

The following lemma tells us about the monotonicity of the solutions.

Lemma 4.1.3 :

For a fixed N = mn,

(i) $T(m,n) = m(\log ((n+1)/m) + 3) - (\log m + 3)$ is monotonic increasing for $m \geq 2$

(ii) $T(m,n) = n(\log (m/(n+1)) + 3) - \log (n+1)$ is monotonic increasing for $n \geq 2$

Proof:

(i) Consider

$$f(m) = m \left(\log\left(\frac{(N/m) + 1}{m}\right) + 3 \right) - (\log m + 3)$$

$$= m (\log ((N+m)/m^2) + 3) - (\log m + 3)$$

$$\frac{df}{dm} = \log \left(\frac{N+m}{m^2}\right) + 3 + m\left(\frac{1}{\ln 2}\,\frac{m^2}{N+m}\,\frac{1 - 2m(N+m)}{m^4}\right) - \frac{1}{m \ln 2}$$

$$= \log ((N+m)/m^2) + 3 + (1/\ln 2)((1 - 2m(N+m))/((N+m)m)) - 1/(m \ln 2)$$

$$= \log ((N+m)/m^2) + 3 + 1/(m\,(N+m)\,\ln 2) - 2/(\ln 2) - 1/(m \ln 2)$$

$$\geq 1 + 3 + 1.44/((N+m)m) - 2.88 - 1.44/m$$

since $(2m - 1) \leq n$ or $m^2 \leq (N+m)/2$

$$> 0 \text{ for } m > 2$$

(ii) Consider

$$g(n) = n\,(\log\,((N/n)/(n+1)) + 3) - \log\,(n+1)$$

$$= n\,(\log\, N/(n^2+n) + 3) - \log\,(n+1)$$

$$\frac{dg}{dn} = \log\,\left(\frac{N}{n^2+n}\right) + 3 +$$

$$n\left(\frac{1}{\ln 2}\;\frac{n^2+n}{N}\;\frac{-(2n+1)}{(n^2+n)^2}\right) - \left(\frac{1}{\ln 2}\;\frac{1}{(n+1)}\right)$$

$$= \log\,\left(\frac{N}{n^2+n}\right) + 3 - \frac{n}{(\ln 2}\;\frac{(2n+1)}{N*n*(n+1)} - \frac{1}{(\ln 2)(n+1)}$$

$$= \log\,(N/(n^2+n)) + 3 -$$

$$(2n+1)/((\ln 2)*N*(n+1)) - 1/((n+1)\ln 2)$$

since $n \leq (2m - 1)$ or $n^2+n \leq 2N$

$$(N/(n^2+n)) \geq 1/2$$

$$> -1 + 3 - 2.88/N - 1.44/(n+1)$$

$$> 0 \quad \text{for } n \geq 2$$

□

Lemma 4.1.3 tells us that for m, n > 2, m = 2j, n = $2^{i}-1$ and assuming that m1 = m2 = m/2, the maximum comparisons are made when n = 2m-1, and this number is bounded by $3\sqrt{mn}$.

The case m = n is of particular interest as the beap has this shape. Again, assuming that m = 2^{j} and that the split is always into two equal parts, the cost for searching can easily be calculated by considering the row- and column-trees as shown in Fig. 4.1.4.

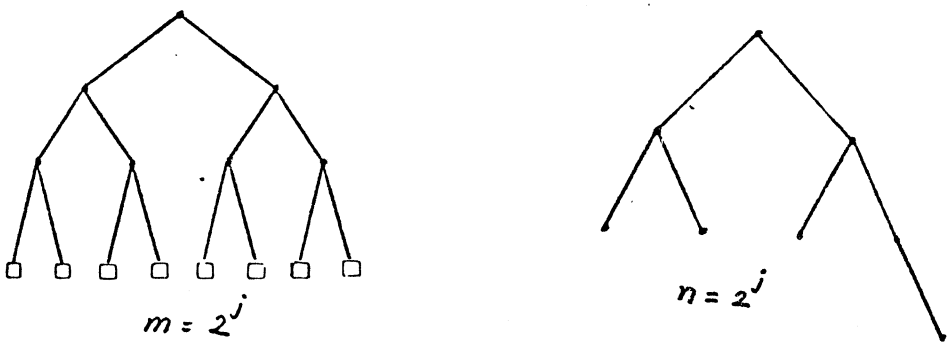


Figure 4.1.4 : Row- and Column-Trees.

It can be shown from the trees that

$$T(2^j, 2^j) = T(2^j, 2^j-1) + 1$$
$$= (n-1)*3 - \log n + 1$$
$$= 3n - \log n - 2$$
$$= 3\sqrt{mn} - (1/2) \log (mn) - 2$$

In general, assuming that $m = 2j$, $n = 2j-1$, and $m1 = m2 = m/2$, a formula for adding k new rows can be found as follows:

(a) $n \leq 2m - 1$, $1 \leq k \leq 2^i$

$$T(2j, 2i-1) = (2i-1) ((j-i) + 3) - i$$
$$T(2^j, (2^i-1)+k) = (2^i-1) (j-i+3) - i + k (j - i + 1)$$
$$T(m,n1) = (n1-k) (j-i+3) - i + k (j-i+1)$$
$$= n1 (j-i+3) - k(j-i) - 3k - i + k(j-i) + k$$
$$= n1 (j-i+3) - i - 2k$$
$$T(m,n1) = n1 (\lfloor \log (m/(n+1)) \rfloor + 3) -$$
$$\lfloor \log (n1+1) \rfloor - 2(n-2\lfloor \log (n+1) \rfloor +1)$$

$$2i-1 \leq n < 2i+1-2$$

(b) $n \geq 2m-1$

(i) $1 \leq k \leq 2^j$

$$T(2j, 2i-1) = 2j(i-j) + 3*2j - (j+3)$$
$$T(2^j, (2^i-1)+k) = 2^j(i-j) + 3*2^j - (j+3) + k$$

$$T(m,n) = m \lfloor \log ((n+1)/m) \rfloor + 3m - (\log m + 3) + (n - 2\lfloor \log(n+1) \rfloor + 1)$$

$$2^{i-1} \leq n1 \leq 2^{i-1}+m.$$

(ii) $2i-1 + m \leq n1 \leq 2i+1-1$

$$T(m,n1) = m (\lceil \log((n1+1)/m) \rceil + 3) - (\log m + 3)$$

This means that adding the last column will not increase the search-cost in the worst-case.

The dual problem of adding new rows is more difficult to analyze and not pursued here.

So far we have analyzed the case where the rows are divided into two equal parts. Another case that can be easily analyzed is when m1 = 1 and m2 = m - 1, that is, the row-tree has degenerated as shown in Fig. 4.1.5.

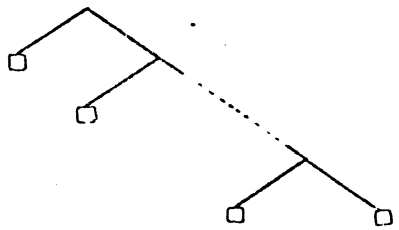


Figure 4.1.5 : Degenerate Row-Tree.

Again, assuming n = 2i - 1, the search cost can be found as follows:

(i) $m \leq \log (n+1) = i$

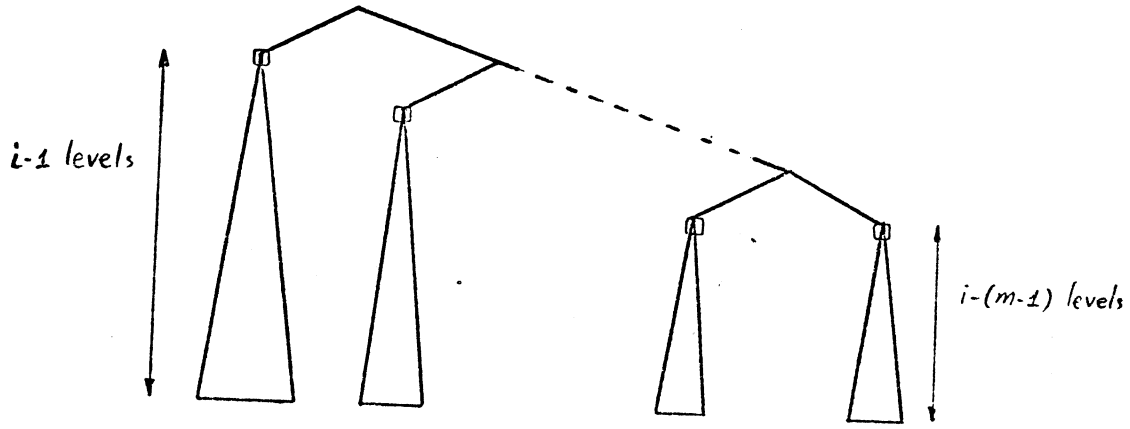


Figure 4.1.6 : Row- and Column-Trees.

The search cost is

$$T(m,n) = (i-1 + (i-2) + \ldots + (i-(m-1)) + (i-(m-1)) +$$

$$\lceil \log(m+1) \rceil + \lceil \log m \rceil + \ldots + \lceil \log 2 \rceil$$

$$= \frac{(m-1)\ (i-1+i-m+1)}{2} + i - (m-1) + \sum_{k=2}^{m+1} \lceil \log (k+1) \rceil$$

$$= \frac{(m-1)\ (2i-m)}{2} + i - (m-1) + \sum_{k=3}^{m+2} \lceil \log k \rceil$$

Using the fact that

$$\sum_{k=1}^{m} \lceil \log k \rceil = m \lceil \log m \rceil - 2^{\lceil \log m \rceil} + 1$$

[Knuth73b, pp. 184]

$$T(m,n) = \frac{(m-1)(2i-m)}{2} + i - (m-1) + \sum_{k=1}^{m+2} \lceil \log k \rceil -$$

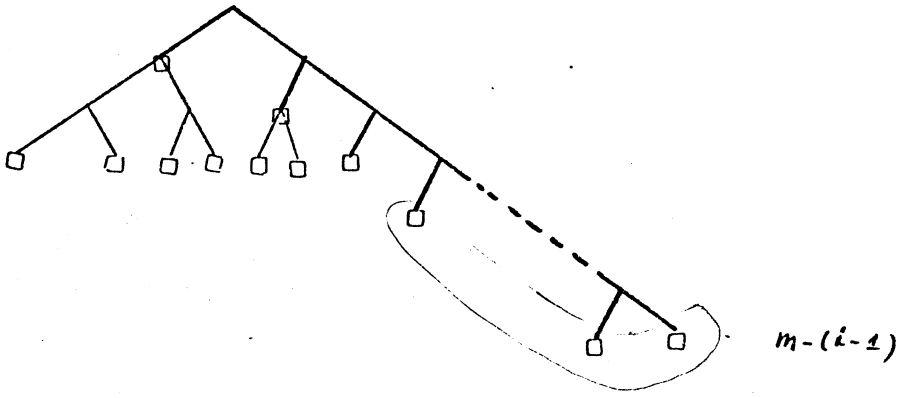
$$\lceil \log 1 \rceil - \lceil \log 2 \rceil$$

$$= \frac{(m-1)(2i-m)}{2} + i - (m-1) + (m+2)|\log (m+2)| -$$

$$2^{\lceil \log(m+2) \rceil} + 1 - 1$$

$$= O(m \log(n+1)) = O((\log n)^2).$$

(ii) $m \geq \log(n+1) = i$



The search cost is

$$T(m,n) = (i-1) + \ldots + 1 + \lceil \log (m+1) \rceil + \ldots + \lceil \log (m-(i-2)) \rceil + \log (m-(i-1))$$

$$\leq ((i-1)i)/2 + (i-1) \lceil \log (m+1) \rceil$$

$$= O((\log n)(\log m))$$

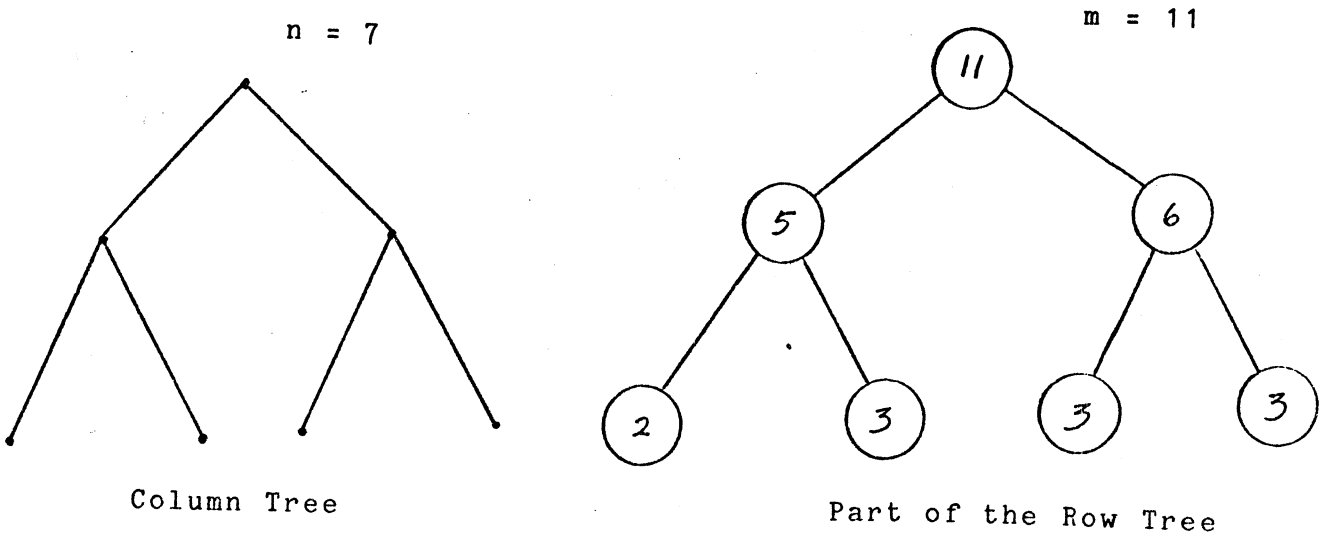
$$= O((\log N)2) \text{ where } N = mn.$$

It is conjectured that this is the best case of splitting the rows. This is consistent with the result of lemma 4.1.3 which tells us that the search cost is maximized if $n \approx 2m$.

The fact that taking $m1 = \lfloor m/2 \rfloor$ and $m2 = \lceil m/2 \rceil$ does not maximize the search cost can be seen in the following examples:

Example 1: $n = 7$, $m = 11$

a)

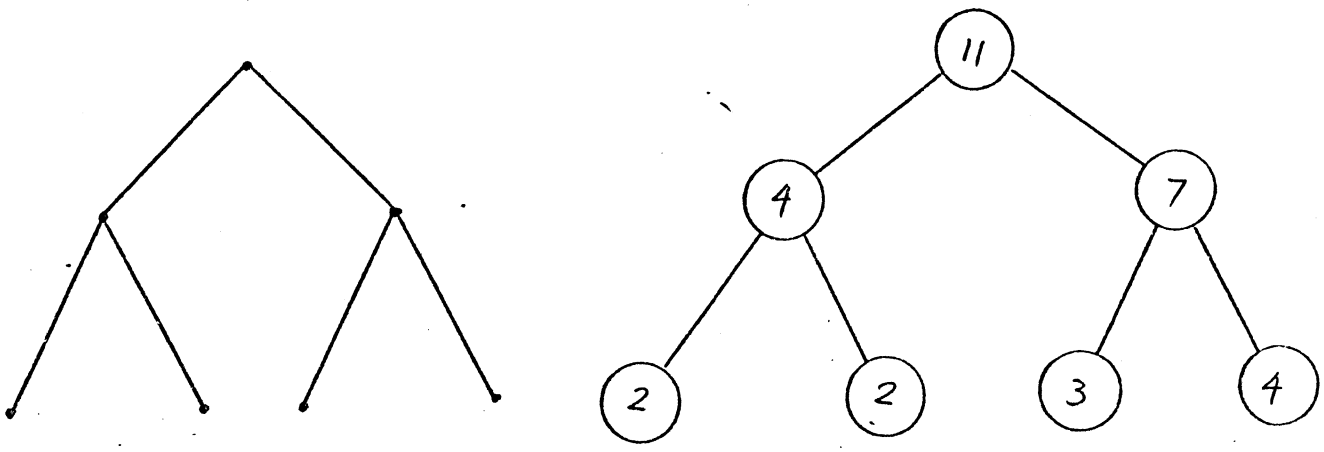


Column Tree

Part of the Row Tree

search cost = 18

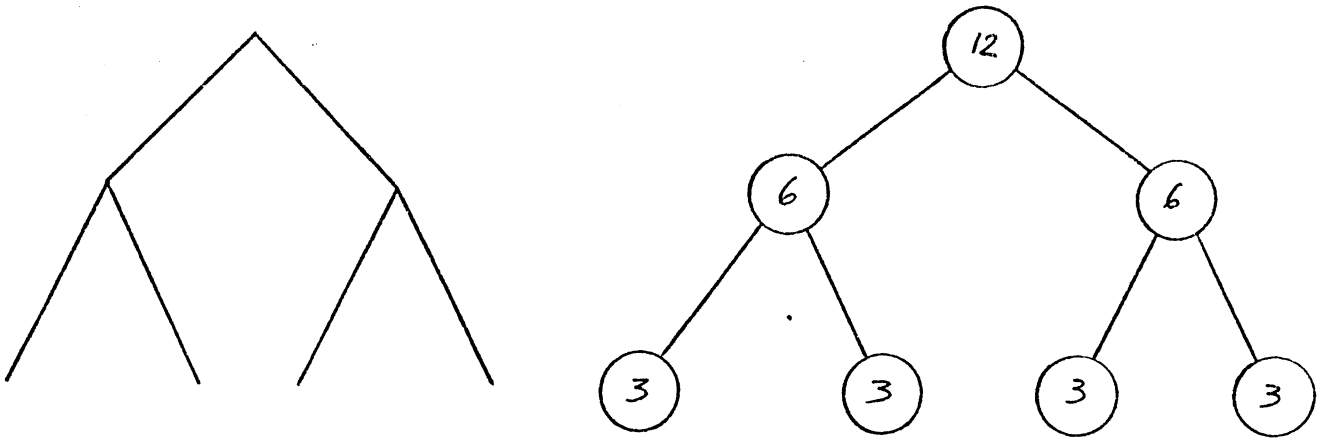
b)



search cost = 19

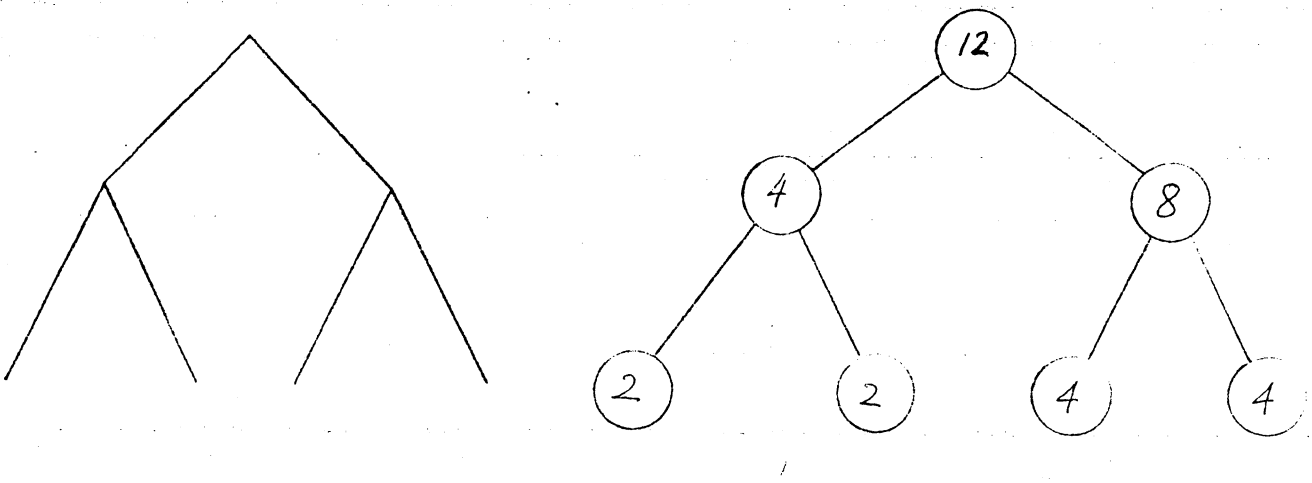
Example 2: n = 7, m = 12

a)



search cost = 18

b)



search cost = 21

One explanation of this fact is that although $2 \log (6+1)$ is larger than $\log (4+1) + \log (8+1)$, $2\lceil \log (6+1) \rceil$ is smaller than $\lceil \log (4+1) \rceil + \lceil \log (8+1) \rceil$. It seems that the ceiling operator plays an important role.

The worst-case complexity of the search cost for small values of m and n can be easily found by a dynamic programming approach. These values are given in Table 4.1.1. It is interesting to note that they are almost symmetrical. This reaffirms that the search cost does not depend upon the side on which the binary searches are performed. These costs are bounded by $3\sqrt{mn} - \log m - 1$ as

shown in table 4.1.2.

m \ n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5
2	2	3	4	5	6	6	6	7	8	8	8	8	8	8	8	9
3	2	4	5	6	7	8	8	9	10	11	11	11	11	11	11	12
4	3	5	7	8	9	10	11	12	13	14	15	15	15	15	15	16
5	3	6	7	9	10	12	12	13	14	15	16	17	17	17	17	18
6	3	6	8	10	11	13	14	15	16	17	18	19	19	20	20	21
7	3	6	8	11	12	14	15	16	17	18	19	21	21	22	22	23
8	4	7	10	12	14	16	18	19	20	21	22	23	24	25	26	27
9	4	8	10	13	15	17	18	20	21	22	23	25	26	27	27	28
10	4	8	10	13	16	17	19	21	22	23	24	26	28	28	29	30
11	4	8	10	14	16	18	19	22	23	24	25	27	29	30	30	31
12	4	8	11	14	16	19	21	23	24	26	27	29	30	32	33	34
13	4	8	11	15	17	20	21	24	25	27	28	30	32	34	34	35
14	4	8	11	15	18	20	22	24	26	28	29	31	33	35	36	37
15	4	8	11	15	18	20	22	25	27	29	30	32	34	36	37	38
16	5	9	13	16	19	22	25	27	29	31	33	35	37	39	41	42

Table 4.1.1 : The Search Costs (Worst-Case Behaviour).

m \ n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2.0	3.2	4.2	5.0	5.7	6.3	6.9	7.5	8.0	8.5	8.9	9.4	9.8	10.2	10.6	11.0
2	2.2	4.0	5.3	6.5	7.5	8.4	9.2	10.0	10.7	11.4	12.1	12.7	13.3	13.9	14.4	15.0
3	2.6	4.8	6.4	7.8	9.0	10.1	11.2	12.1	13.0	13.8	14.6	15.4	16.2	16.9	17.5	18.2
4	3.0	5.5	7.4	9.0	10.4	11.7	12.9	14.0	15.0	16.0	16.9	17.8	18.6	19.4	20.2	21.0
5	3.4	6.2	8.3	10.1	11.7	13.1	14.4	15.7	16.8	17.9	18.9	19.9	20.9	21.8	22.7	23.5
6	3.8	6.8	9.1	11.1	12.8	14.4	15.9	17.2	18.5	19.7	20.8	21.9	22.9	23.9	24.9	25.8
7	4.1	7.4	9.9	12.1	13.9	15.6	17.2	18.6	20.0	21.3	22.5	23.7	24.8	25.9	26.9	27.9
8	4.5	8.0	10.7	13.0	15.0	16.8	18.4	20.0	21.5	22.8	24.1	25.4	26.6	27.7	28.9	29.9
9	4.8	8.6	11.4	13.8	16.0	17.9	19.6	21.3	22.8	24.3	25.7	27.0	28.3	29.5	30.7	31.8
10	5.2	9.1	12.1	14.7	16.9	18.9	20.8	22.5	24.1	25.7	27.1	28.5	29.9	31.2	32.4	33.6
11	5.5	9.6	12.8	15.4	17.8	19.9	21.9	23.7	25.4	27.0	28.5	30.0	31.4	32.8	34.1	35.3
12	5.8	10.1	13.4	16.2	18.7	20.9	22.9	24.8	26.6	28.3	29.9	31.4	32.9	34.3	35.7	37.0
13	6.1	10.6	14.0	16.9	19.5	21.8	23.9	25.9	27.7	29.5	31.2	32.8	34.3	35.8	37.2	38.6
14	6.4	11.1	14.6	17.6	20.3	22.7	24.9	26.9	28.9	30.7	32.4	34.1	35.7	37.2	38.7	40.1
15	6.7	11.5	15.2	18.3	21.1	23.6	25.8	28.0	29.9	31.8	33.6	35.3	37.0	38.6	40.1	41.6
16	7.0	12.0	15.8	19.0	21.8	24.4	26.7	28.9	31.0	32.9	34.8	36.6	38.3	39.9	41.5	43.0

Table 4.1.2 : $3\sqrt{mn}$ - log m - 1.

It is still an open question which shape of the row-tree maximizes the search cost. However, we note that the "even split" is at least close to the worst-case as we show in the following theorem.

Theorem 4.1.4:

Let T(m,n) be defined as

$$T(m,n) \leq T(m1, \lfloor (n-1)/2 \rfloor) + T(m2, \lceil (n-1)/2 \rceil) + \lceil \log(m+1) \rceil$$

$$T(1,n) \leq \lceil \log(n+1) \rceil \qquad m1+m2 = m$$

$$T(m,1) \leq \lceil \log(m+1) \rceil$$

Then, $T(m,n) \leq 3\sqrt{mn} - \log m - 1$

Proof:

By induction on (m,n).

Base: By Tables 4.1.1 and 4.1.2.

Ind. Hypothesis: Assume $T(m',n') \leq 3\sqrt{m'n'} - \log m' - 1$ for all $(m',n') < (m,n)$.

Ind. step:

$$T(m,n) \leq (3\sqrt{m1 \lfloor (n-1)/2 \rfloor} - \log m1 - 1) +$$
$$(3\sqrt{m2 \lceil (n-1)/2 \rceil} - \log m2 - 1) + |\log (m+1)|$$

$$\leq 3\sqrt{m1\ n/2} - \log m1 - 1 +$$
$$3\sqrt{m2\ n/2} - \log m2 - 1 + \lceil \log (m+1) \rceil$$

Consider

$$f(m1) = 3\sqrt{m1\ n/2} - \log m1 + 3\sqrt{m2\ n/2} - \log m2$$

$$= 3\sqrt{m1\ n/2} - \log m1 + 3\sqrt{(m-m1)\ n/2} - \log (m-m1)$$

$$\frac{df}{dm1} = \frac{3\sqrt{n/2}}{2\sqrt{m1}} + \frac{1}{m1 \ln 2} - \frac{3\sqrt{n/2}}{2\sqrt{(m-m1)}} + \frac{1}{(m - m1) \ln 2}$$

$$= 0 \text{ for } m1 = m/2$$

$$\frac{d^2f}{dm1^2} = -\left(\frac{3/2\sqrt{n/2}}{2\ m1^{3/2}}\right) + \frac{1}{m1^2 \ln 2} - \frac{3/2\sqrt{n/2}}{2(m-m1)^{3/2}} + \frac{1}{(\ln 2)\ (m-m1)^2}$$

$$= -\left(\frac{3\sqrt{n}}{4\sqrt{2}}\right)\left(\frac{1}{m1\sqrt{m1}} + \frac{1}{(m-1)\sqrt{m-1}}\right) + \frac{1}{\ln 2}\left(\frac{1}{m1^2} + \frac{1}{(m-m1)^2}\right)$$

$$\left.\frac{d^2f}{dm1^2}\right|_{m1=m/2} = -\left(\frac{3\sqrt{n}}{4\sqrt{2}}\right)\left(\frac{2}{(m/2)\sqrt{m/2}}\right) + \frac{1}{\ln 2}\,\frac{8}{m^2}$$

$$= -3n/(m\sqrt{m}) + (8/(\ln 2))\ (1/m^2)$$

$$< 0 \text{ for } m \geq 15 \quad n \geq 1$$

Thus, f(m1) is maximized if m1 = m2 = m/2 for $m \geq 15$.
Going back to the recurrence relation,

$$T(m,n) \leq 2*3\sqrt{(m/2)(n/2)} - 2 \log (m/2) - 2 + \lceil \log (m+1) \rceil$$

$$\leq 3\sqrt{mn} - 2 \log m - 4 + \log m + 1$$

$$\leq 3\sqrt{mn} - \log m - 3$$

$$\leq 3\sqrt{mn} - \log m - 1$$

□

We note that if N = mn, at most $3\sqrt{N}$ comparisons are required to find an arbitrary element. Given a search procedure for a rectangular grid, we can easily develop a similar technique for a beap, i.e. the upper triangular half of an n x n grid. The easiest method is to imagine that "virtual elements" having the value $+\infty$ are in the lower triangle which would complete a square. Note that this leads to a $3\sqrt{2N}$ solution. By careful implementation, i.e. by checking if the element under consideration is in the upper or lower triangle, which can be done easily by using the indices of the element in the matrix, further saving of comparisons between keys can be achieved at the cost of more comparisons between indices.

4.2 Parallel Algorithms for Beaps

4.2.1 Parallel Searches

Having explored and analyzed the divide and conquer algorithm for searching in the last section, we can adopt it to a parallel search algorithm in a natural way. From the fact that the rectangles resulting from the splitting are independent, different processors can be assigned to perform the search. The problem of how to assign these processors immediately arises as we try to keep the processors busy all the time. The easiest way is to put new rectangles produced by the splittings into a queue and assign these to available processors. In the analysis we assume that k is much smaller than m or n (in fact, $k < \min(\log m, \log n)$). We first look at some special cases:

Lemma 4.2.1.1 :

For $k=2^s$, $m=2^j$, $n=2^i-1$ and assuming that the rows are always divided into 2 equal parts, then

$$T_k(m,n) \leq 3/k \sqrt{mn} + \log k \log m.$$

Proof :

Consider the row- and column-tree. Until level log k - 1, not all processors can be utilized simultaneously. After this level, each level can be processed k times faster.

Lemma 4.2.1.2 :

With the same assumption as above, except that k is not a power of two,

$$T_k(m,n) \leq 3/k \sqrt{mn} + \log k \log m + \log m.$$

Proof :

After the $\lceil \log k \rceil^{th}$ level, there are at most k consecutive levels whose number of nodes (rectangles) is divisible by k (since $2^{x+1}+\ldots+2^{x+d}$ is divisible by k for some $d < k$ and $x \geq \lceil \log k \rceil$). This group of levels can be speeded up by a factor k with a 1 step penalty per level.

For the general case, we have the following theorem :

Theorem 4.2.1.3 :

$$T_k(m,n) \leq 3/k \sqrt{mn} + (k+1) \log m.$$

Proof:

Case (1) : Assume that once all processors are busy, they remain so except at the final steps. It can be seen that the "full" speed up can be achieved except at the beginning and the last steps. The bound then follows immediately. It should be noted that this bound is quite pessimistic.

Case (2) : Assume that at most $p < k$ processors are busy all time. Since the depth of the superimposing trees

is at most min(m, log n), the total cost is bounded by $(\log mn)^2 + \log mn$ which is below the bound stated in the theorem.

Case (3) : Assume that for some period all processors are busy, then some processors remain idle. Again, from case (2), idleness of the processors tells us that some subproblems can be solved faster than the case if all processors were active. Thus, the bound follows.

Again, as in the case of the sequential search, we can imagine a beap as a rectangle (with $+\infty$ in the lower triangle). This gives the following corollary :

Corollary 4.2.1.2 :

For $k < \log N$, searching an arbitrary element on a beap can be done in $3/k \sqrt{2N} + (k+1) \log m$ steps under k-paralellism. Thus, a speed up factor of approximately $2k/3$ can be achieved under k-parallelism.

4.2.2 Parallel Insertion and Deletion

For the sake of completeness, we include this section which is taken from [Munro79]. Parallel insertions can be performed directly on the beap. It is divided into k groups of equal height as shown in Fig. 4.2.2.1.

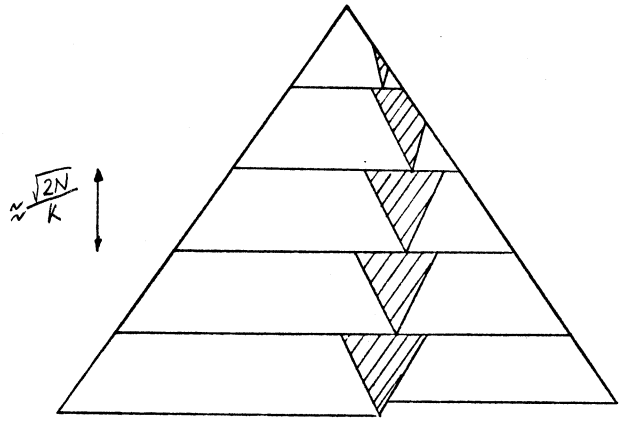


Figure 4.2.2.1 : A Beap Divided into k Groups.

We note that an element has at most i+1 ancestors at the ith preceeding generation or next i higher level (two parents, three grandparents, etc.). A new element is inserted into the location (N+1). All the ancestors of that element in the next $\sqrt{2N}/k$ higher level can be determined by index manipulation. We then assign all k processors to find the maximum of these ancestors. This can be done in at most $\sqrt{2N}/k^2 + O(\log \log k)$ comparisons [Valiant75]. If this maximum is bigger than the new element, we then swap these two elements. We repeat the same "jumping" process until the new element is bigger than all its ancestors in the next

$\sqrt{2N}/k$ higher level. These preliminary steps can be done in at most $\sqrt{2N}/k + O(k \log \log k)$ steps. At this time an element within each group may not satisfy the beap property. The point is that because of the preliminary steps they can only filter up by at most $\sqrt{2N}/k$ levels. We then assign each processor to each group to restore the order simultaneously. it can be seen that at most $2\sqrt{2N}/k$ steps are required. Hence, a total of $3\sqrt{2N}/k + O(k \log \log k)$ steps are sufficient to insert a new element. We have shown:

Theorem 4.2.2.1:

For $k < N$, inserting a new element into a beap of size N can be done in $\lceil 3\sqrt{2N}/k \rceil + O(k \log \log k)$ steps under k-parallelism, in addition to index manipulation for finding the ancestors.

Corollary 4.2.2.2:

A speed up factor of approximately 2k/3 can be achieved in parallel insertion under k-parallelism.

By noting that an element has at most i+1 descendants in the next i lower level (ith succeeding generation), parallel deletions can be handled in an analogous manner.

CHAPTER 5

SUMMARY, CONCLUSIONS, AND FUTURE RESEARCH

5.1 Summary

In chapter 1, we described the motivation for this thesis, previous related research, and an overview of this thesis.

In Chapter 2, we showed that the product of the search cost and the cost for deletion/insertion is at least N, the number of the elements, for implicit data structures which use a fixed partial order as this underlying scheme. We also suggested a structure that "essentially" achieves this bound. We showed that the product of these costs can be reduced if schemes other than a fixed partial order are used. Indeed, $O(N^{1/3} \log N)$ steps are sufficient to perform a search, an insertion, or a deletion.

In Chapter 3, we gave the lower and upper bounds on forming a beap. We showed that the number of comparisons required to form a biparental heap are 1/2 N log N minus lower order terms, while 1/2 N log N plus lower order terms are sufficient. We also showed how to select the kth smallest element in such a structure in O(min(k log k, N log N)) steps.

In Chapter 4, we showed that a speed up of approximately 2k/3 can be achieved in performing the basic operations on a biparental heap if k-parallelism is allowed.

5.2 Conclusions

We have suggested solutions to the dictionary problem with regard to the worst-case complexity by using implicit data structures. Although the algorithm suggested have not been implemented yet, we expect that they can be coded easily.

We have contributed to study of the time-space trade-offs for data structures representing a dictionary in our model of computation. Table 5.2.1 summarizes this trade-off assuming that a key requires the same amount of storage as an index does.

Array Size	Cost of Searching	Cost of Insertion/Deletion
3N (tree-structures)	$O(\log N)$	$O(\log N)$
2N (array with gaps)	$O(\log N)$	$O(N^{1+1/d})$, $d \geq 2$ (a sequence of N operations)
$N + \sqrt{2N}$ (Rotated Lists + pointers)	$O(\log N)$	$O(\sqrt{N})$
N (Rotated Lists)	$O(\log N)$	$O(\sqrt{N} \log N)$
N (Beap of Rotated Lists)	$O(N^{1/3} \log N)$	$O(N^{1/3} \log N)$

Table 5.2.1 : Time-Space Trade-Offs for Structures Containing N elements.

Although we have addressed to the solution of the dictionary problem, there are other potential useful applications of the structures discussed in this thesis:

a) All structures we suggested are suitable for representing priority queue since the minimum element can be found in constant time. Furthermore, if two beaps are placed "back to back" as a "twin-beap" in a similar way as "twin-heap" [Knuth73b, exercise 5.2.3.31], both the minimum and maximum elements can be found in constant

time. The cost for the three basic operations remains as before.

b) Selecting the k^{th} smallest element can be done in $O(\sqrt{N})$ time on a beap and in O(log N) time on rotated lists. It is also possible on these structures to locate all keys which lie between two given values k1 and k2 (range searching) in sublinear time by partitioning the elements around k1 and k2.

For the rest of this section, we try to make brief comparisons between the tree structures (with explicit pointers) and the implicit structures and between the hash table and implicit structures under the assumption that there is a memory manager that allocates a contiguous array of a requested size.

a) Tree Structures vs Implicit Structures

Both share the same problem in allocating storage for the structure; it is not obvious how much storage should be allocated. However, tree structures are more flexible in the sense that an arbitrary number of trees can be "grown" in the same memory space and therefore, tree structures have the potential in increasing the memory utilization. In contrast, only 2 implicit structures can be maintained in the array to ensure that overflow will occur only when total size of the structures exceeds the total space.

Other than the facts mentioned above, the time-space trade-off dominates the difference between the two structures. It should be noted, however, that rotated lists are attractive for applications where the number of searches is much bigger than the number of insertions and deletions.

b) Hash Tables vs Implicit Structures

While implicit structures such as heaps or rotated lists can answer the range queries including the nearest neighbour queries efficiently, hash tables cannot answer the same queries in any reasonable manner. Implicit structures are more flexible in the sense that two structures can be maintained in an array. They are also more manageable when overflow occurs.

A brief comment about deletions is in order. When chaining is used with separate lists for each cell of the table, deletions will cause no problem. However, the search time is proportional to N when the size of the table is fixed and N becomes large (re-hashing is required to avoid this).

If linear probing is used, deletion will cause no degradation in performance. However, a search on a full table costs $\Theta(\sqrt{N})$ comparisons on the average (although a search on 99% full table will take a constant time on the average).

If open addressing is used, for full table, any sophisticated scheme known including the double hashing costs $\Omega(N)$ for a deletion unless flag indicating "dirty" cell is used. This method is workable only when enough insertions are made so that the dirty cells are reused; otherwise, since the dirty cells will never become empty again, they can degrade the performance drastically.

In summary, an implicit structure such as a rotated list is a better structure than the hash table if the structure is required to be dynamic, if range queries are to be made, or the minimum is to be found efficiently, or if we must be able to respond in guaranteed time when searching for an arbitrary element.

In conclusion, we feel we have shown the class of implicit data structures to be of practical as well as theoretical interest.

5.3 Future Research

There are several problems left open in this thesis. The most interesting problem is, of course, concerning the lower bounds on general implicit structures. A special case of this would be to determine the lower bounds for rotated lists embedded in a partial order, which are still unknown.

The sizes of the blocks on the beap or rotated list play a major role in the performance of the operations. This sequence of blocksizes (called the triangular number) and others are used in transforming static data structures to dynamic structures [Saxe79]. It would be interesting if this useful method of blocking can be applied to other problems. Also, whether or not rotated lists and others will have an application for secondary memories (including bubble memories) remains to be seen.

The method described in Chapter 3 to form a beap can be generalized to arbitrary partial order. The question of whether this method of forming a partial order requires a number of comparisons differing from the information theoretic lower bound by only a lower bound term is currently under investigation.

The reliability issue of implicit structures compared to structures with explicit pointers also provides potential for future work.

REFERENCES

[Aho74]
Aho, A.V., J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974.

[Batcher68]
Batcher, K.E., "Sorting Networks and Their Applications", Proc. AFIPS 1968 SJCC, Vol 32, pp. 307-314

[Bentley78]
Bentley, J.L., D. Detig, L. Guibas, J. Saxe, "An Optimal Data Structure for Minimal Storage Dynamic Searching", unpublished manuscript.

[Blum72]
Blum, M., R.W. Floyd, V. Pratt, R.L. Rivest, R.E. Tarjan, "Time Bounds for Selection", JCSS 7 (1973), pp. 448-461.

[Bondy76]
Bondy, J.A., U.S.R. Murty, Graph Theory With Applications, American Elsevier Publishing Co., 1976.

[Borodin72]
Borodin, A.B., J.I. Munro, Notes on "Efficient and Optimal Algorithms", Department of Computer Science, University of Toronto, 1972.

[Borodin79]
Borodin, A.B., L.J. Guibas, N.A. Lynch, A.C. Yao, "Efficient Searching via Partial Ordering", unpublished manuscript, 1979.

[Dobkin79a]
Dobkin D., J.I. Munro, "Determining the Mode", Research Report CS-79-27, Dept. of Computer Science, University of Waterloo, 1979.

[Dobkin79b]
Dobkin D., J.I. Munro, private communication, 1979.

[Floyd64]
Floyd, R.W., "Algorithm 245 : Treesort 3", CACM 7 (1964), pp. 701.

[Frederickson79]
Frederickson, G., D.B. Johnson, private communication, 1979.

[Gavril75]
Gavril, F., "Merging With Parallel Processors", CACM 18(1975) pp. 588-591.

[Hirschberg78]
Hirschberg, D.S., "Fast Parallel Sorting Algorithms", CACM 21 (1978) pp. 657-661.

[Hirschberg79]
Hirschberg, D.S., A.K. Chandra, D.V. Sarwate, "Computing Connected Components on Parallel Computers", CACM 22 (1979) pp. 461-464.

[Johnson78]
Johnson, D.B., T. Mizoguchi, "Determining the k-th Element in X + Y and X1 + X2 + ... + Xm", SICOMP 7 (1978), pp. 147-153.

[Knuth73a]
Knuth, D.E., The Art of Computer Programming,
Vol. I : Fundamental Algorithms, Second Edition, Addison Wesley, 1973.

[Knuth73b]
Knuth, D.E., The Art of Computer Programming,
Vol. III : Sorting and Searching, Addison Wesley, 1973.

[Melville78]
Melville, R., D. Gries, "Sorting and Searching Using Controlled Density Arrays", TR 78-362, Computer Science Dept., Cornell University, 1978.

[Munro79]
Munro, J.I., E.L. Robertson, "Parallel Algorithms and Serial Data Structures", Proc. 17-th Annual Allerton Conference on Communication, Control, and Computing (1979).

[Preparata78]
Preparata, F.P., "New Parallel-Sorting Schemes", IEEE Trans. on Computers. C-27 (1978), pp. 669-673.

[Regbati78]
Regbati, E., D.G. Corneil, "Parallel Computations in Graph Theory", SICOMP 7 (1978), pp. 230-237.

[Saxe79]
Saxe, J.B., J.L. Bentley, "Transforming Static Data Structures to Dynamic Structures", Proc. 20-th IEEE Symposium on FOCS (1979), pp. 148-168.

[Snyder77]
Snyder, L., "On Uniquely Representable Data Structures", Proc. 18-th IEEE Symposium on FOCS (1977), pp. 142-146.

[Tarjan77]
Tarjan, R.E., "Reference Machines Require Non-Linear Time to Maintain Disjoint Sets", Proc. 9-th Annual ACM STOC (1977), pp. 18-29.

[Valiant75]
Valiant, L.G., "Parallelism in Comparison Problems", SICOMP 4 (1975), pp. 348-355.

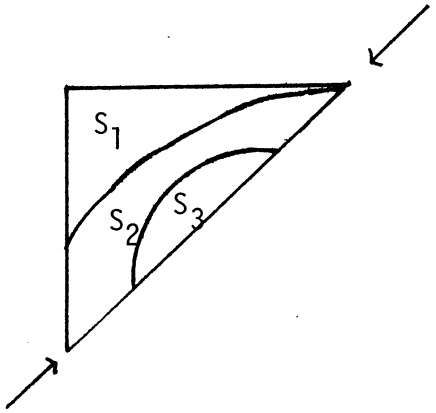
[Williams64]
Williams, J.W.J., "Algorithm 232 : Heapsort", CACM 7 (1964), pp. 347-348.

[Yao78]
Yao, A.C., "Should Tables Be Sorted?", Proc. 19-th IEEE Symposium on FOCS (1978), pp. 22-27.

Addendum to CS-80-04 (page 63)

The algorithm k^{th} smallest described in page 63 was intended for beaps containing distinct elements. If repeated keys are present in the beap, the algorithm needs a slight modification in partitioning the beap.

Here we partition S into S_1, S_2 and S_3 containing elements of S $<$, $=$, and $>$ respectively. This can be achieved by searching the value of (weighted median - ε) and (weighted median + ε) from upper right corner and lower left corner respectively as depicted below.



If $|S_1| \geq k$ then repeat the same process with S_2. Otherwise, if if $|S_1| + |S_2| \geq k$ then the weighted median is the k^{th} smallest element. Otherwise, assign $k := k - |S_1| - |S_2|$ and repeat the same process with S_3.

Note that this process is repeated until the test in while statement fails.