POTENTIAL FUNCTIONS IN AMORTIZED COMPUTATIONAL COMPLEXITY ANALYSIS

By

RUO-LING HU

Bachelor of Science
Shanghai Institute of Mechanical Engineering
Shanghai, China
1984

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1988

POTENTIAL FUNCTIONS IN AMORTIZED COMPUTATIONAL COMPLEXITY ANALYSIS

Thesis Approved:

Donald D Fisher
Thesis Adviser

[illegible]

M. J. Folk

Norman N. Durham
Dean of the Graduate College

ACKNOWLEDGEMENTS

It is a pleasure and good fortune to have studied in the Department of Computing and Information Sciences, at Oklahoma State University.

I wish to express my most deep and sincere gratitude to Dr.Donald.D. Fisher, my advisor, for his encouragement and confidence in me throughout this study. His continuous guidance based on his rich experience has made a major contribution to my career. His support and understanding are forever appreciated.

I also extend my heartfelt thanks to Dr. G.E. Hedrick and Dr. M.J. Folk for serving as part of my advisory committee, for their valuable comments, encouragement, and advisements.

Finally, I want to send my gratefulness and thanks to my parents for their support, encouragement and patience.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Computer science can be defined as the study of data, its representation, and its transformation by a digital computer [1]. Although there are many things involved in writing computer programs to solve an application problem, the building blocks of most computer programs [2], namely, the data structures and algorithms are most important, and affect the computational efficiency significantly.

In fact, there is an intimate connection between the structuring of data and the synthesis of algorithms. The structures are collections of variables,connected in various ways; while an algorithm is a finite set of instructions which accomplish a particular task. An algorithm may operate on multiple data structures; on the other hand, more than one algorithm may operate on a data structure. Therefore, for any given problem, we will specify a suitable data structure and give an applicable algorithm which operates on that structure. Actually, the data structure and the algorithm are so tightly connected that they should be considered as a unit, neither one making sense without the other.

In the last two decades the study of data structures and algorithm analysis has been very active. The development of new data structures and new techniques for analyzing algorithms has greatly helped the growth in the fields of network and graph theory [3], database design [4-11], and file and index organizations [12-14], etc. At the same time, the structure performance and algorithm complexity have been widely evaluated and discussed [15-27], in order that some efficient algorithms can be found corresponding to some significant structures in applications. Thus, the analysis of complexity (more specifically, the complexity of a given algorithm on a specified data structure) is always an important research topic in computing and data processing.

Complexity Analysis

Basically, there have been two ways to evaluate the complexity of an algorithm that occurs with operations on a data structure [3]. First, we measure the complexity by the length of the program. This measure has interesting theoretical uses [28,29], however, it is static in nature since the complexity is independent of the input data. On the other hand, the performance of common data structures are often evaluated and compared based on their responses to certain data operations. The required running time or storage space, which are functions of the input size, serve as a dynamic measure of the algorithm on a data structure. In practice, the running time is more significant in cost and complexity analysis, because most of the algorithms have a space bound that is a linear function of the input data [3].

Running Time Analysis

If we use a parameter n to characterize the inputs and/or outputs, then the running time of an algorithm can be represented by a function, f(n) [1].

Definition: $f(n) = O(g(n))$ iff there exist two constants c and n_0 such that

$$|f(n)| <= c * |g(n)|, \text{ for all } n >= n_0,$$

where g(n) is also a function of parameter n. Common function classes for g(n) include polynomial, logarithmic, exponential, and products of members from different classes as discussed by Horowitz et. al. [1] and Aho et. al. [2].

When we say that the computing time of an algorithm is $O(g(n))$ we mean that its execution of n data operations takes no more than a positive constant times the g(n) for n sufficiently large. Thus, the running time function f(n) provides a mathematical description of the data structual performance and algorithm efficiency.

However, it is important to know that the running time estimate depends strongly on the analysis technique applied. Different methods result in the variation of complexity estimation. Tarjan [3] presented three analysis methods of running time available for different applications: the worst-case, the average-case, and the amortized analysis.

The worst-case method has been widely applied in complexity analysis because it is simple and it provides a performance guarantee. For a worst-case estimate, the running time is calculated as the sum of the worst-case times of all the individual operations. But this measure is not so realistic and it may give an overly pessimistic estimate of the actual performance if the worst case occurs rarely.

The average-case analysis of running time is another common method. The analysis is carried out based on probabilistic assumptions and the running time is averaged over the possible operations. However, such an analysis is generally much harder than the worst-case method because of the difficult in determining the probability distribution that accurately reflects reality.

In most data structure applications, some particular algorithms, such as a sequence of search, insertion, deletion, and updating, are repeatly applied. For these situations, the amortized analysis by which we average the running time per operation over a worst-case sequence of operations, combines aspects of the two former methods and has proven to be more realistic than worst-case analysis and more robust than average-case analysis.

Amortized Analysis

The amortized running time analysis is a newly developed technique. It is especially appropriate in complexity analysis of a variety of data structures, where a sequence of operations rather than a single operation is always performed and the total running time for that operation sequence is of interest.

The concept of amortization was well explained by Tarjan [30]. He used the " banker's " view and the "physicist's" view to analyze the running time of operations on a data structure. In the former case, the computer user acts as a customer of a bank. Initially there is no credit in the user's account. The credits reflecting the amortized operation time are then deposited into or withdrawn from the account according to the operation sequence. Finally, an account balance is available from which the upper and lower bounds of performance of that data structure can be obtained. While from the physicist's view, the change of states in a data structure due to a sequence of operations can be imagined as the energy change in a water tank system [31], where a pump is used to increase the potential energy by pumping water to a higher level and a generator is used to produce electrical energy by allowing the higher level water to flow down. It is clearly seen that a potential function is generally needed in amortization analysis, to present either the account balance in the banker's view or the system potential energy in the physicist's view. The major advantages of the amortization analysis is that it not only provides more exact measures of running time for known algorithms but also suggests some new algorithms, which might be more efficient in an amortized rather than a worst-case sense. In last five years, this method has been applied to evaluate a number of data structures and algorithms, such as list data structures, paging rules [30,32], various balanced search trees [33-36], heap operations [37,38], and database view schemas [39], etc.

A thorough literature survey shows, however, that several classes of potential functions exist and that the potential functions were chosen freely in each application. For example, we can use the number of inversions in updating as the potential function for list structures and use either the height or the number of internal nodes for different tree structures. No paper has been found which concerns the principles and/or limitations in selecting the potential function for common data structures. It was believed [30] that the more astute the choice, the more informative the amortized time estimate. Such an arbitrariness, however, probably makes difficulties in complexity comparison and thus obstructs the application of this new method. Therefore, there is a growing demand from the designers and users, who conduct complexity analysis, to systematically analyze the potential function, since it plays the most important role in determining the amortized running time.

Research Objective

The objective of this research is twofold: to explore the possibilities as well as the limitations of different potential functions in amortized analysis, and to evaluate the amortized performance by using these potential functions for typical accessing and updating algorithms on several data structures. The idea is to identify classes of potential functions which best support complexity analysis for each specified data structure. This investigation is the first known attempt to discuss the role of potential functions. Furthermore, it will also provide us added insight into the performances of some useful data structures.

Fisher [33] summarized Tarjan's theory and suggested three promising potential functions from many candidates, for easy implementation and calculation. They are the rank, the sum children including itself, and the log sum. Fisher and Hu [34] also proposed to use the height as structure potential. These four classes of potential functions are defined and analyzed for each of the following data structures: linear lists, balanced search trees (including the HB[1]-tree, Red-Black tree, and B-tree), and heaps (including the pairing heap and self-adjusting heap). In each case, the typical operation algorithms are presented. Then the amortized complexities are analyzed based on each potential function. By comparing the limitations of these potential functions, an optimal procedure of amortization analysis is possible for each data structure analyzed.

Chapter II reviews previous work on amortization for relevant data structures. The four selected potential functions are presented in chapter III. Chapters IV, V, and VI develop amortized measures for typical operation algorithms in lists, balanced search trees, and heap structures, respectively. In chapter VII, the influence of potential functions on structure performance is discussed. Also, the amortized computational complexities of these data structures are compared with the worst-case results in applications. Chapter VIII includes conclusions and suggestions for further study.

CHAPTER II

LITERATURE SURVEY

In this chapter, we survey investigations on amortized performance analysis. Since most fundamental work in this area was developed by Tarjan, we first outline his theory of amortization. On this base, we review the amortization in several data structures. Finally, we summarize the potential functions analyzed in different situations.

Amortized Computational Complexity

Amortization is a new method in analysis of computational complexity for data structures. According to Tarjan [3, 30], amortization can be defined as an "averaging" algorithm that averages the running times of operations in a sequence over the sequence and is appropriate in many uses of data structures where particular algorithms are repeatedly applied.

As stated in the chapter I, Tarjan used two views, the banker's view and the physicist's view, to explain the concept of amortization. In both cases, the state of a physical data structure can be represented by a variable called the potential function. This function, viewed as either the current account balance or the potential energy level, is operation-dependent and, thus, can be described mathematically.

Tarjan defined such a potential function ϕ that maps any configuration D of a data structure into a real number $\phi(D)$ called the potential of D. The amortized time of an operation is:

$$a = t + (\phi(D') - \phi(D)) \tag{2.1}$$

where t = the actual time of the operation

D = data structure configuration before operation

D' = data structure configuration after operation

With this definition, the total amortized cost due to any sequence of k operations can be obtained as

$$\sum_{i=1}^{k} a_i = \sum_{i=1}^{k} t_i + (\phi_f - \phi_0) \tag{2.2}$$

where a_i = amortized time of the ith operation

t_i = actual time of the ith operation

ϕ_0 = data structure potential before the k operations reflecting the initial configuration

ϕ_f = data structure potential after the k operations reflecting the final configuration

Eq.(2.2) states that the total amortized time (or the sum of amortized times) equals the total time of the k operations plus the net increase in data structure potential, caused by the operations, from the initial to the final configuration. In most cases of interest the initial potential is zero and the potential is always nonnegative. In such a situation the total amortized time serves as an upper bound of the total running time.

Amortized Updating Cost

The amortized method is receiving more attention today. In last five years, many cost-related problems have been analyzed by using the amortized concept, including updating cost of "Move-to-Front" algorithm in linear lists[30,32,44], updating in (2,3) trees, (a,b) trees, and Red-Black trees [17,19,32,36], sequential insertions into AVL-trees [35,41] and B-trees [33,34], pairing heaps [37] and self-organizing heaps [38,39], and for view complexity of relational databases [40].

Linear List Amortization

A linear list is a simple data structure. Tarjan [30] discussed the amortized behavior on a stack, a simple form of the list structure, from the physicist's view. The potential of a stack can be defined as the number of items contained in the stack. Suppose initially the stack contains i items and then K pops and one push are carried out. The stack potential changes from i to (i - K + 1). Thus, m such operations take at most

$$((K + 1) + (i - K + 1) - i) * m = 2 * m \quad \text{(steps)}$$

which gives an amortized time $O(1)$ per operation.

Many researchers analyzed the cost of self-organizing (or self-adjusting) sequential operating algorithms for the list. Tarjan [30] and Sleator [32] considered an abstract data structure consisting of a table of n items, under the sequential operations of accessing k specified items. This problem is also called the dictionary problem. We assume that the data table is represented by a linear list whose items are in arbitrary order. Three kinds of operations are usually allowed:

OP_1. access (i): locate the ith item in the list
OP_2. insert (i): insert the ith item into the list
OP_3. delete (i): delete the ith item from the list

To access an item, we scan the list from the front until locating the item (Fig.1a). To insert an item, we scan the entire list to verify that the item is not already present and then insert it at the rear of the list (Fig.1b). To delete an item, we scan the list from the front to find the item and then delete it (Fig.1c). Thus, the cost of the various operations can be defined as follows. Accessing or deleting the ith item in the list costs i; inserting a new item costs n+1, where n is the size of the list before the insertion.

After each operation, we may want to rearrange the list by swapping pairs of consecutive items in order to speed up later operations and reduce the total running time. There are three proposed swapping heuristics:

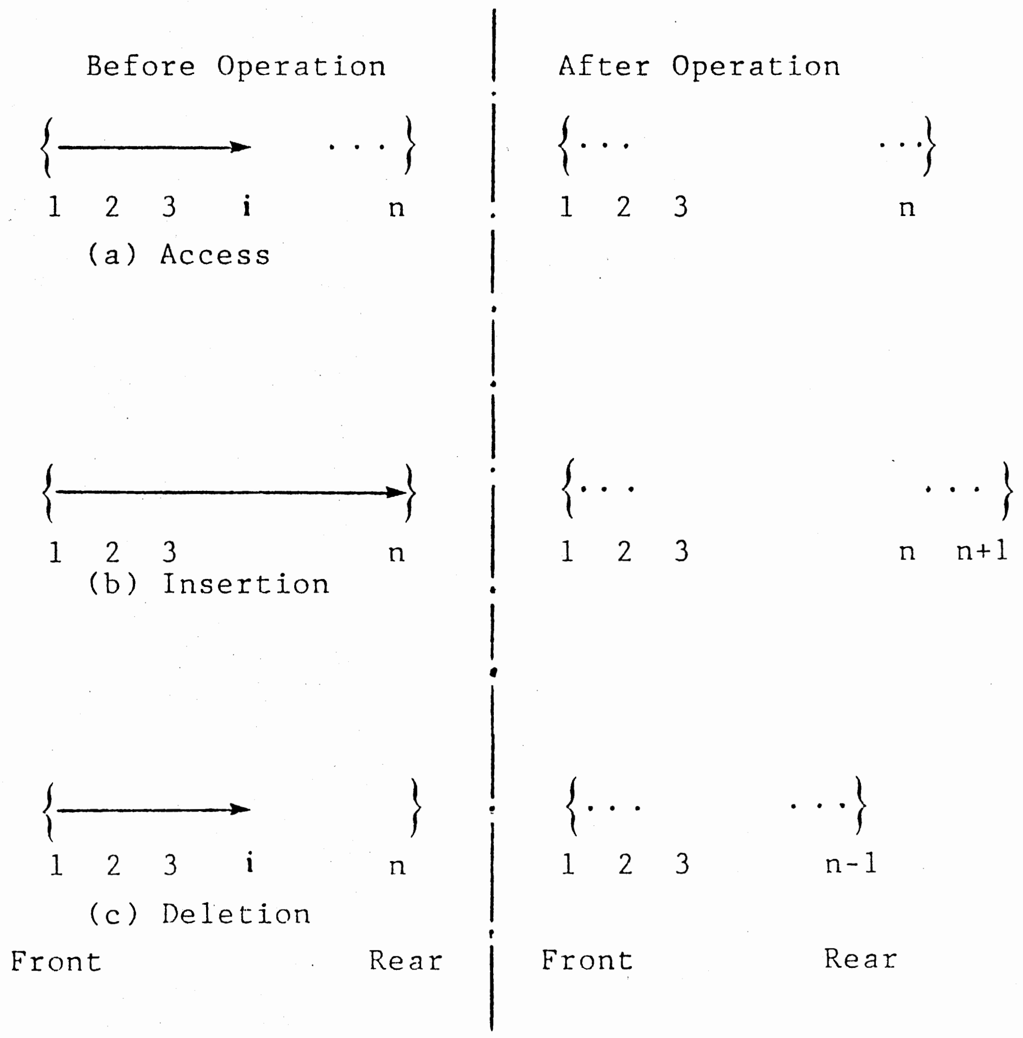
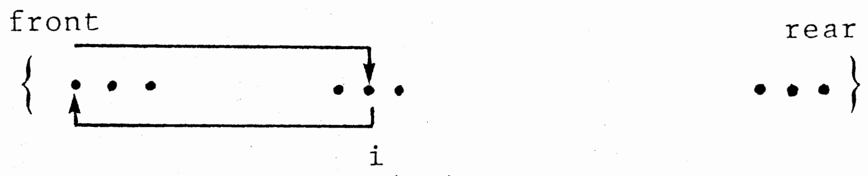
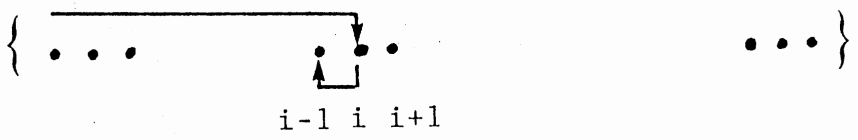


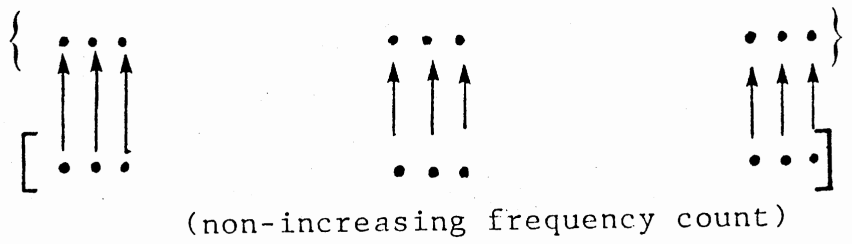
Figure 1. Linear List Operations



(a) Move-to-Front (MF)



(b) Transpose (T)



(c) Frequency Count (FC)

Figure 2. Three Swapping Rules in Linear List

1. Move-to-Front (MF): after accessing or inserting an item, move it to the front of the list, without changing the relative order of the other items (Fig.2a).
2. Transpose (T): after accessing or inserting any item other than the first on the list, exchange it with the immediately preceding item (Fig.2b).
3. Frequency Count (FC): maintain a frequency count for each item (initially set to zero); increase the count of an item by 1 whenever it is inserted or accessed; reduce its count to zero when it is deleted; maintain the list so that the items are in nonincreasing order by cumulative frequency count (Fig.2c).

The swapping cost is defined as follows. Immediately after an access or insertion of an item i, allow i to be moved freely to any position closer to the front of the list. Any other exchange, called a paid exchange, costs 1 unit. Our final goal is to find an optimal algorithm (operation plus swapping), from the three list maintainance heuristics described above.

Rivest [42] showed that transpose costs less than move-to front in a list with n fixed items and without exchanges. However, Bitner [15] discovered that move-to-front performs much better in practice. He suggested that move-to-front converges much faster to its asymptotic behavior if the list is random initially. Bentley and McGeoch [43] tested the various update rules empirically on real data. Their tests showed that transpose is inferior to frequency count but move-to-front is competitive with frequency count and sometimes better. This reveals that some real sequences have a high locality of reference, which move-to-front, but not frequency count, exploits. Bentley and McGeoch also studied the amortized complexity for cases of a fixed list of n items on which only accesses are permitted. They compared the three updating rules (MF,T,and FC) with a static algorithm, called decreasing frequency (DF), which minimizes the total access cost among algorithms that do not rearrange the list. Let s be any sequence of access operations and C(s) be the total cost of all the accesses. Bentley and McGeoch proved that $C_{MF}(s) <= 2 * C_{DF}(s)$ if MF's initial list contains the items in order by first access. Frequency count but not transpose shares this property. Sleator and Tarjan [32] applied the concept of potential function, as described in the previous section, to generalize Bentley and McGeoch's results.

Consider running an arbitrary algorithm A and the move-to-front heuristic MF in parallel on an arbitrary operation sequence, starting with the same initial list for both methods. We define as the potential function the number of inversions in MF's list with respect to A's list. For any two lists containing the same items, an inversion in one list with respect to the other is an unordered pair of items, i, j, such that i occurs anywhere before j in MF's list and anywhere after j in A's list (as shown in Fig.3a), or that i occurs before j in A's list and after j in MF's list (as shown in Fig.3b). Here we use a standard example list to illustrate the calculation of list potential function. In Fig.3c, two lists contain the same items but are under different algorithms, so the item orders are different. The inversion number for each item j (j=1,...,n) in MF's list can be calculated by using either the method shown in Fig.3a or that in Fig.3b. We call the inversion number before j as the potential of item j, then the sum of all the item potentials is the list potential. For the standard example, both methods lead to a list potential of 20.

If the initial A's and MF's lists are empty, the initial MF's configuration has zero potential (zero inversion); and the final configuration has a nonnegative potential (more inversions). Thus, by Eq.(2.2), we know that the actual cost to MF of a sequence of k operations is bounded by the sum of the operations'amortized times. That is

$$\sum_{i=1}^{k} t_i = \sum_{i=1}^{k} a_i - \phi_f + \phi_0 \;<=\; \sum_{i=1}^{k} a_i \qquad (2.3)$$

Using number of inversions as the potential function, we have the amortized operation times for the MF rule as follows.

$$\text{A} \quad (\qquad\qquad\qquad j \; . \; \overbrace{. \; . \qquad .}^{i} \; . \;)$$

$$\text{MF} \quad (\; . \; . \; \underbrace{. \qquad\qquad .}_{i} \; . \; . \; j \qquad\qquad)$$

(a) i before j in MF's and after j in A's

$$\text{A} \quad (. \; \overbrace{. \; . \qquad .}^{i} \; . \; j \qquad\qquad\qquad)$$

$$\text{MF} \quad (\qquad\qquad\qquad j \; \underbrace{. \; . \; .}_{i} \;)$$

(b) i after j in MF's and before j in A's

A	(a	e	j	o	g	l	n	r	t	x)	
MF	(l	e	n	o	x	j	a	t	g	r)	
by method (a)											
#	0	1	0	2	0	4	6	1	4	2	$\Sigma\# = \phi(D) = 20$
by method (b)											
#	5	1	4	2	5	1	0	2	0	0	$\Sigma\# = \phi(D) = 20$

(c) item and list potentials

Figure 3. List Potentials in terms of Number of Inversions

access i : $a_i <= 2 * i_A - 1$ (2.4)

delete i : $a_i <= i_A <= 2 * i_A - 1$ (2.5)

insert i : $a_i <= 2 * (n+1) - 1 = 2 * i_A' - 1$ (2.6)

where i_A = position of item i in A's list

i_A'= position of item i in A's list after insertion

n = list size before insertion

To prove these amortized costs (inequalities 2.4, 2.5, and 2.6), we consider an access by both algorithms A and MF to an item i. Let the position of i be i_A in A's list and be i_{MF} in MF's list (Fig.4a). Also let x_i be the inversion number (potential) of i. Then the number of items preceding i in both lists is $(i_{MF} - 1 - x_i)$. This not only means that moving i to the front of MF's list creats $(i_{MF} - 1 - x_i)$ inversions and destroys x_i other inversions, but also indicates that $(i_{MF} - x_i) <= i_A$. Thus, the amortized time for access operation is proved as

$$
\begin{aligned}
a_i &= t_i + \phi(D') - \phi(D) \\
&= i_{MF} + (\phi(D) + i_{MF} - 1 - x_i - x_i) - \phi(D) \\
&= 2 * (i_{MF} - x_i) - 1 \\
&<= 2 * i_A - 1
\end{aligned}
$$

In the standard example, we access item a, so i_{MF}=7, i_A=1, x_i=6; after MF of a, the list potential changes from 20 to 14 (Fig.4b). Then the amortized access time is

$$
\begin{aligned}
a_i &= t_i + \Delta\phi = i_{MF} - 6 = 7 - 6 = 1 \\
&<= 2 * i_A - 1 = 1
\end{aligned}
$$

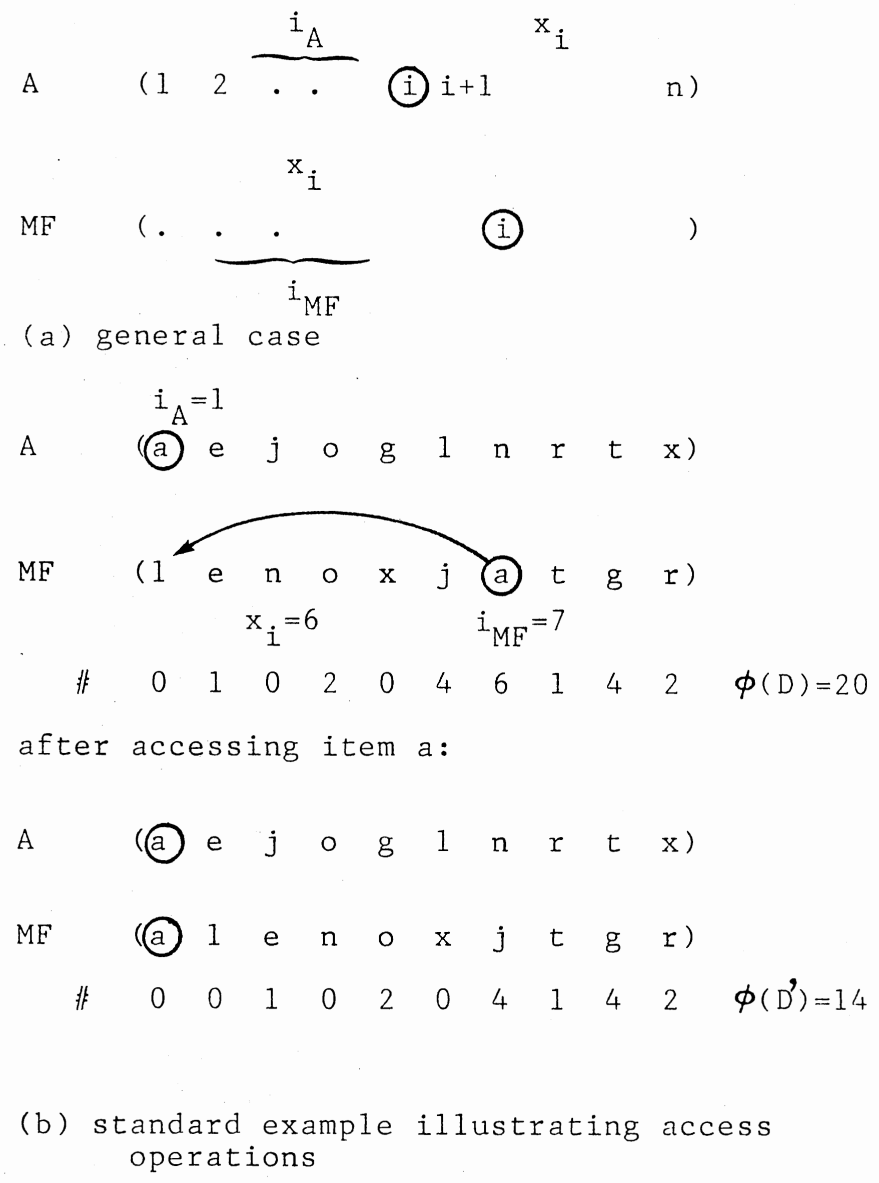


Figure 4. List Potential Changes due to the Access Operation

The above analysis for access operation applies virtually unchanged to a deletion and an insertion. Deletion creates no inversions but still destroys x_i inversions, so

$$a_i = t_i + \Delta\phi = i_{MF} - x_i$$
$$<= i_A <= 2 * i_A - 1$$

In the case of an insertion, i_A=n changes to i_A'=n+1, the variation of inversions is the same as that in an access operation, so

$$a_i <= 2 * (n+1) - 1 = 2 * i_A' - 1$$

Inequalities (2.4), (2.5), and (2.6) show that the amortized time per operation a_i is at most $(2 * i_A - 1)$. Furthermore, when A does a free exchange (move i in A's list to left/front), the inversion number reduces at least one; when A does a paid exchange (move i in A's list to right /rear), the inversion number increases at most one. Thus, for any algorithm A and any sequence of k operations starting with the empty list,

$$C_{MF} <= 2 * C_A + X_A - F_A - k \qquad (2.7)$$

where C = total operation cost excluding paid exchanges

X = number of paid exchanges

F = number of free exchanges

Using the standard example, we insert the ten items sequentially into an initially empty list. C_A is the sum from one to ten and equals 110. k is ten. If A does no exchange, then $X_A = F_A = 0$, potential increase is 20. So,

$$C_{MF} = \sum a_i = \sum t_i + \Delta\phi = 110 + 20$$
$$<= 2 * C_A - k = 220 - 10$$

If A does X_A free exchanges and F_A paid exchanges, then

$$C_{MF} = \sum a_i = 110 + 20 + X_A - F_A$$
$$<= 2 * C_A + X_A - F_A - k$$

Since A can be any algorithm including algorithms that base their behavior on advance knowledge of the entire sequence of operations, we know that the performance of MF for the current list condition is within a factor of 2 of that by an optimum algorithm. Relation (2.7) is a very strong result, which implies the average-case optimality result for move-to-front heuristics. No analogus result holds for transpose or for frequency count.

By using the same potential function, the amortized costs of MF for several different situations are also available. When the initial set is nonempty, and MF and A begin with different lists, the cost bound is

$$C_{MF} <= 2 C_A + X_A - F_A - k + I \qquad (2.8)$$

where I = initial number of inversions, $<= \binom{n}{2}$

If the insertion position i rather than the length n of the list is the cost of an insertion, the result is similar to relation (2.7).

Also, we consider the modified MF method by which the accessed or inserted item at position i is moved at least (i/d - 1) units closer to the front, then the cost bound is

$$C_{MF} <= d * (2 * C_A + X_A - F_A - k) \tag{2.9}$$

where d = integer larger than or equal to 1.

Finally, if we count free exchanges in the MF, the cost bound is

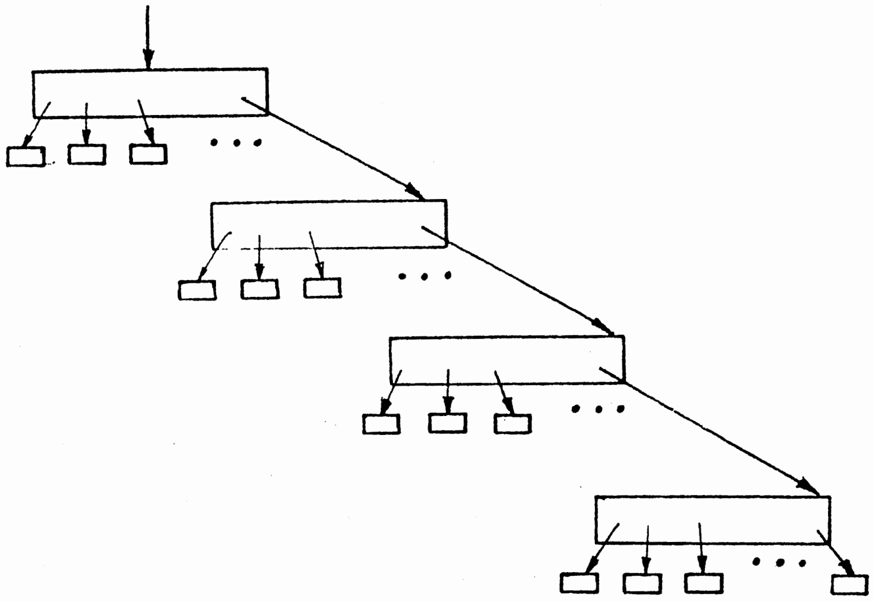
$$C_{MF} = 2 * \sum_{i=1}^{k} a_i$$

$$<= 4 * C_A + 2 * X_A - 2 * F_A - 2 * k \tag{2.10}$$

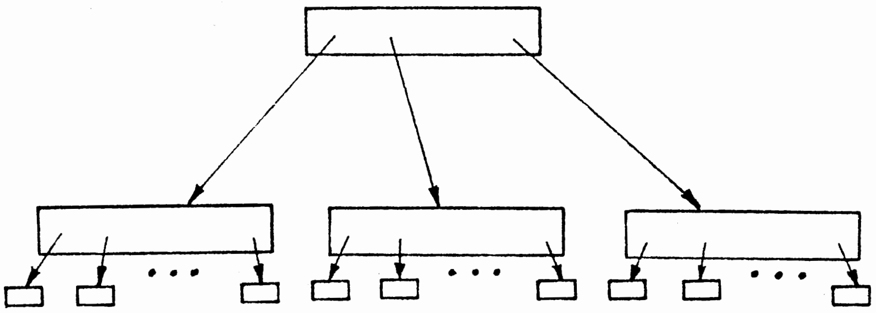
The above results show that, by amortized complexity analysis, the move-to-front is generally the best rule for linear list sequential operations as it has a total cost within a constant factor of the minimum cost. The constant is 4 if we do not allow free exchanges in updating or 2 if we do. Since the theoretical results are well supported by Bentley and McGeoch's experiments, Sleator and Tarjan believed that the amortized complexity is a more robust and more realistic measure for all list update rules than asymptortic average-case complexity.

Search Tree Amortization

Search trees provide a way of representing static or dynamically changing sorted data sets. If the ordering among items is important, a search tree is the data structure of



(a)



(b)

Figure 5. Unbalanced(a) and Balanced(b) Search Trees [16]

choice [44]. Search trees can be unbalanced or balanced, as shown in Figs.5a and 5b, respectively. While an unbalanced tree has some long paths and some short ones, a balanced tree has all leaves at approximately the same depth. The standard way to make search tree operations efficient in the worst case is to impose a balance condition that forces the depth of an n-node tree to be $O(\log n)$. This requires strong local balance information at each tree node and rebalancing the tree after (or during) each update operation. In the so-called standard balanced trees, the update transformations all take place along a single path in the tree, from the root to the leaf. Therefore, the worst-case time for an insertion or deletion is $\Omega(\log n)$. In this section, we briefly survey previous results concerning amortization of some search tree structures.

<u>(2,3)-Trees</u>

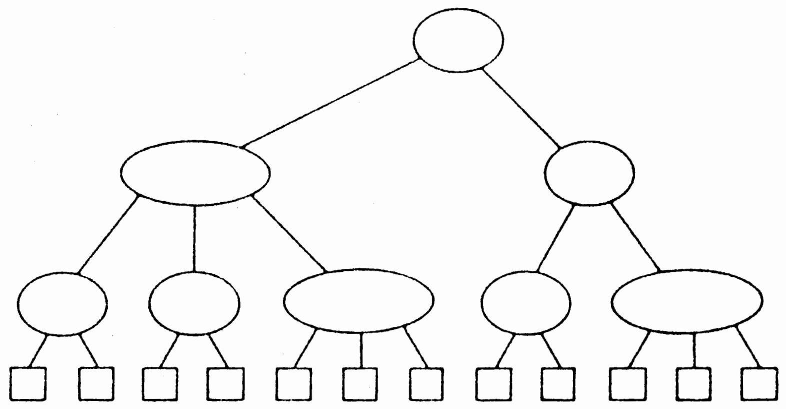
A (2,3)-tree is a height-balanced tree such that 2 or 3-way branching takes place at every internal node, and all external nodes occur on the same level. An internal node with 2-way branching is called a 2-node, and one with 3-way branching a 3-node. The (2,3) tree is a data structure which allows both fast accessing and fast updating of sorted information [17]. An example of a (2,3) tree is given in Fig.6a. It is easy to see that the height h of a (2,3)-tree with n external nodes can be expressed as

$$\log_3(n) < h < \log_2(n) \tag{2.11}$$

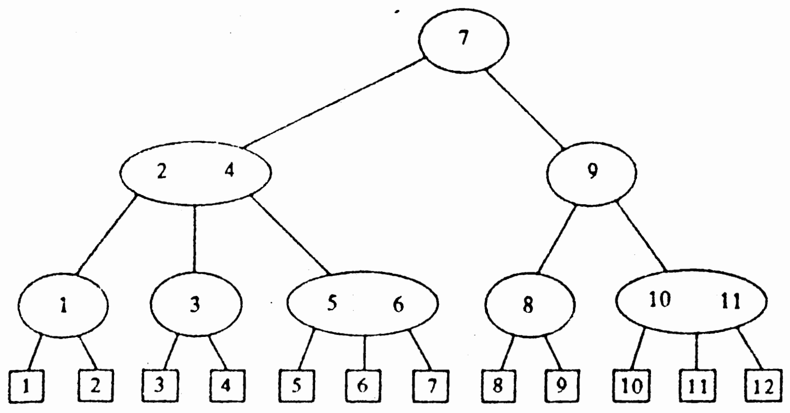
Brown and Tarjan [17] analyzed the cost of sequences of operations on (2,3)-trees. Generally, (2,3)-trees are used to represent a sorted list of length n so that a search for any item in the list takes $O(\log n)$ steps or unit times. Once the position to insert a new item or to delete an old one has been found through a search, the insertion or deletion can be performed in additional $O(\log n)$ steps. But there are several applications of (2,3)-trees in which the regularity of successive insertions or deletions allow searches to proceed faster than $\Omega(\log n)$, such as to implement a priority queue in a sorted list.

One way to represent a sorted list using a (2,3)-tree is shown in Fig.6b. The elements of the list are assigned to the external nodes of the tree, with key values of the list elements increasing from left to right. Keys from the list elements are also assigned to internal nodes of the tree in a "symmetric" order analogous to that of binary search trees. More precisely, each internal node is assigned one key for each of its subtrees other than the rightmost, this key being the largest which appears in an external node of the subtree. Therefore, each key except the largest is shown in an internal node, and by starting from the root of the tree we can locate any element of the list in $O(\log n)$ steps, using a generalization of binary tree search.

We define insertion as the addition of a new external node at a given position in the tree, excluding the search



(a) a (2,3)-tree

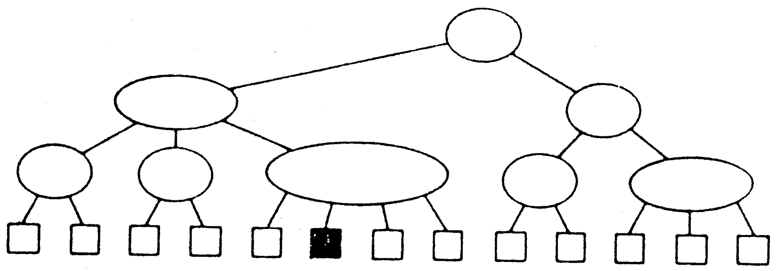


(b) a (2,3)-tree for sorted lists

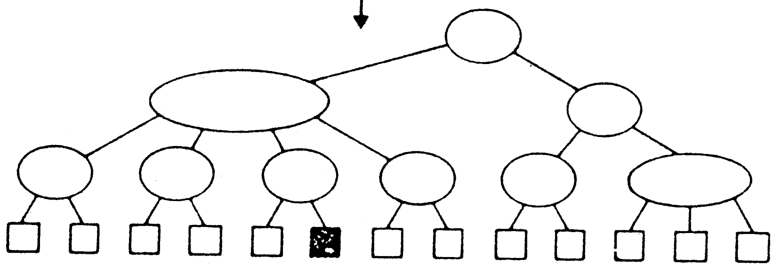
Figure 6. Illustration of a (2,3)-Tree [17]

operation. Insertion is accomplished by a sequence of node expansions and splittings, as shown by the example in Fig.7. When a new external node is attached to a terminal node p, this node expands to accomdate the extra edge. If p was a 2-node prior to the expansion, it is now a 3-node, and the insertion is complete. If p was a 3-node prior to expansion, it is now a 4-node (Fig.7a), which is not allowed in a (2,3)-tree; therefore, p is split into a pair of 2-nodes (Fig.7b). This split causes an expansion of p's parent (Fig.7c), and the process repeats until either a 2-node expands into a 3-node or the root is split. Any individual insertion into a (2,3)-tree initially of size n can cause up to $\log_2(n)$ splittings of internal nodes to take place. Therefore, if k consecutive insertions are made, the total number of splits is bounded by $O(k*\log_2(n))$ in the worst-case. On the other hand, in the amortized case, the k insertions require a maximum of $(\lceil n/2\rceil+k)$ splits, because each split generates a new internal node and the number of internal nodes is initially at least $\lceil (n-1)/2\rceil$ and finally at most $(n+k)-1$. More precisely, if the positions of the newly-inserted nodes in the resulting tree are $p_1 < p_2 < p_3 < \ldots < p_k$, then the number of node splittings which take place during the insertions is bounded by

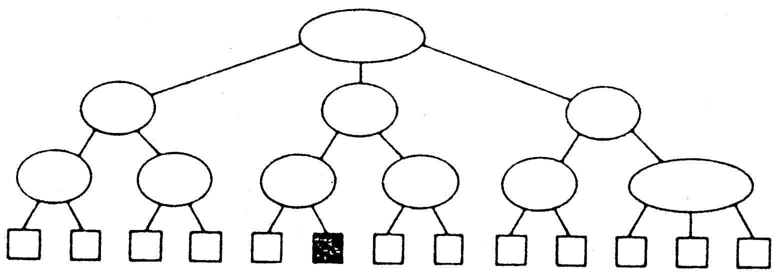
$$C_s <= 2(\lceil \log_2(n+k)\rceil + \sum_{i=1}^{k} \lceil \log_2(p_i - p_{i-1} + 1)\rceil) \quad (2.12)$$



(a) Insert

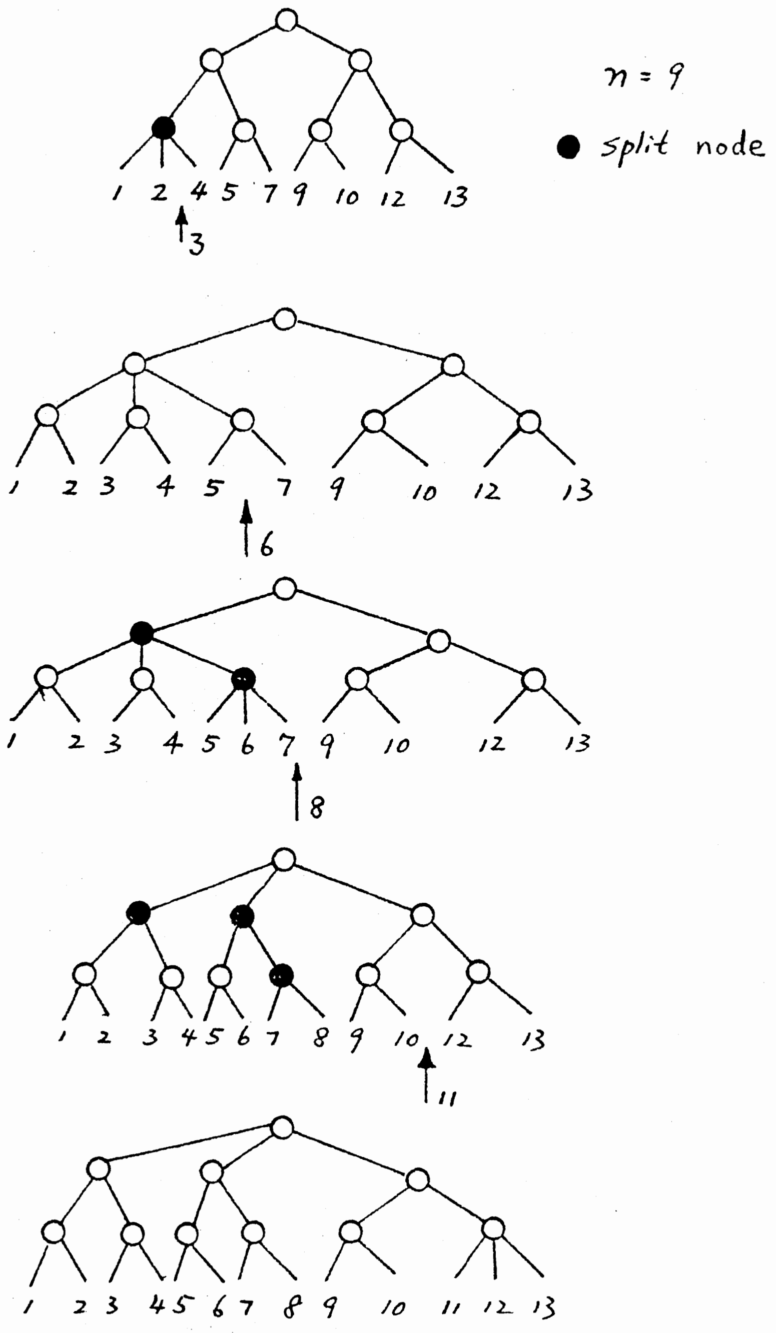


(b) Split



(c) Expansion in parent nodes

Figure 7. Insertion into a (2,3)-Tree [17]



(d) sequential insertions of 3,6,8,and 11

Figure 7. (Continued)

Assuming the tree in Fig.7 originally has 9 nodes. Four new nodes 3, 6, 8, and 11 are inserted sequentially, as shown in Fig.7d. Then

$$C_s \approx [9/2] + 4$$

or

$$C_s <= 2(\log_2(9+4))+\log_2(6-3+1)+\log_2(8-6+1)+\log_2(11-8+1)$$
$$<= 13$$
$$<= 15 \text{ (worst-case, } k*\log_2(n))$$

Similarly, we define the deletion operation as the elimination of a specified external node from the (2,3)-tree, as shown by Fig.8. The first step of a deletion is to remove the external node being deleted (Fig.8a). If the parent of this node was a 3-node before the deletion, it becomes a 2-node and the operation is complete. If the parent was a 2-node, it is now a 1-node (Fig.8b), which is not allowed in a (2,3)-tree; hence some additional changes are required to restore the tree. If the 1-node is the root of the tree, it can simply be deleted; if the 1-node has a 3-node as a parent or as a sibling, then a local rearrange-ment eliminates the 1-node and completes the deletion. Otherwise we fuse the 1-node with its sibling 2-node; this creates a 3-node with a 1-node as parent (Fig.8c). We then must repeat the transformations until the 1-node is eliminated (Fig.8d). Consider a sequence of k (k <= n) deletions from a (2,3)-tree of a initial size n, the number of node fusings which takes place during the deletions is bounded by

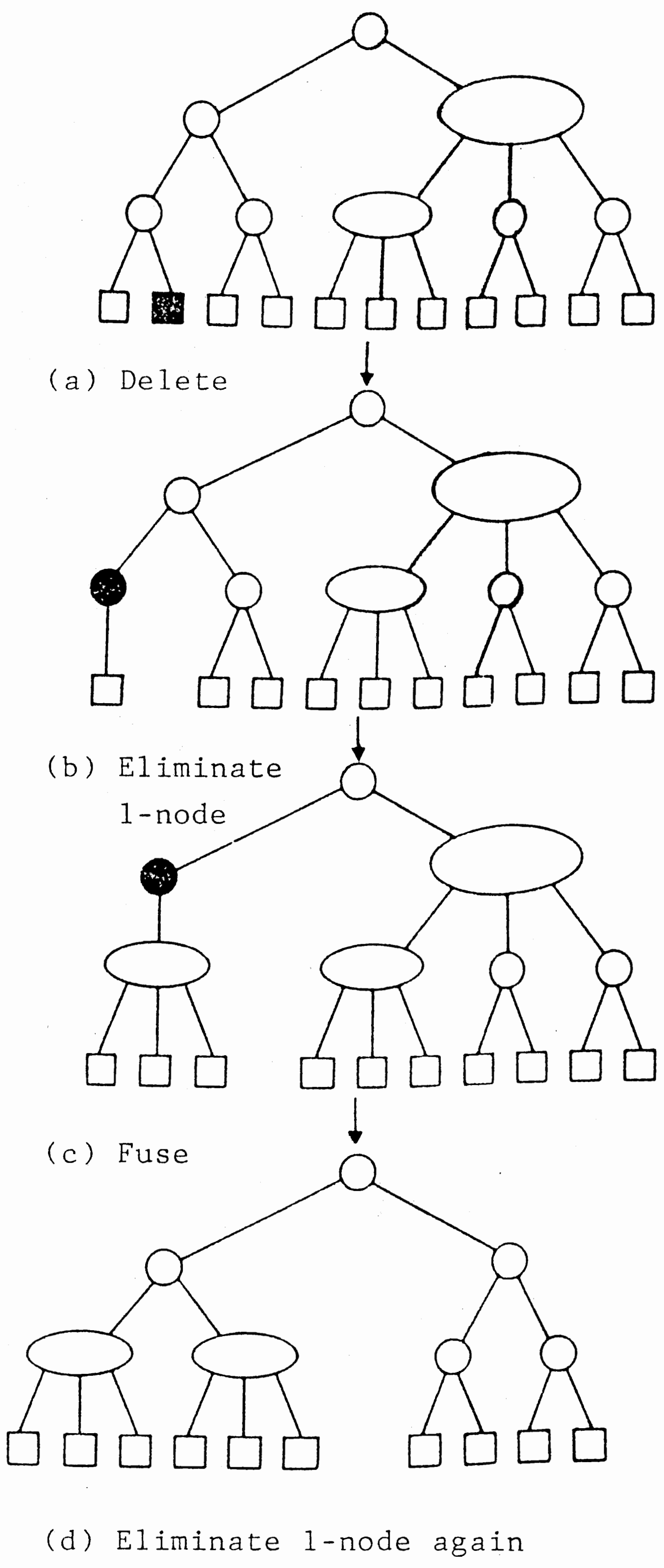


Figure 8. Deletion in A (2,3)-Tree [17]

$$C_f <= 2([\log_2(n)] + \sum_{i=1}^{k} [\log_2(p_i - p_{i-1} + 1)]) \qquad (2.13)$$

When both insertions and deletions are present in a sequence of operations, there are cases in which $\Omega(\log n)$ steps are required for each operation in the sequence. Brown and Tarjan considered the case that k insertions and p deletions take place in separate parts of the n-node tree. They divided the sequence of p deletions into disjoint epochs. Intuitively epochs represent intervals during which insertions do not interact directly with deletions. Let p_i denote the number of deletions during the ith epoch, k_i the number of insertions during this epoch, and m_i the tree size at the start of the epoch. The first deletion of epoch i costs $O(\log m_i)$; the final p_i-1 deletions cost $O(p_i+\log(m_i))$ since they operate on a section of the left path that is unaffected by insertions. Hence the total cost of the deletions in epoch i is $O(l_i+\log(m_i))$. While the insertion cost is $k_{i-1} = O(\log(m_i))$. It means the cost at i epoch is $(p_i + k_{i-1})$, so the total cost of these epochs is

$$C_T = O(k + p) \qquad (2.14)$$

Eq.(2.14) gives an $O(1)$ amortized time per update, which shows that the $O(\log(n))$ worst-case bound on individual insertions and deletions in a (2,3)-tree is overly pessimistic. But this bound does not hold for intermixed insertions and deletions. The potential function was not discussed in this analysis.

(a,b)-Trees

An (a,b)-tree is a data structure more general than the (2,3)-trees. Each node of an (a,b)-tree has at least a and at most b sons. Huddleston and Mehlhorn [19] called this data structure as a class of weak B-trees, and they analyzed the cost of sequences of intermixed insertions and deletions when the (a,b)-trees satisfy the condition of b >= 2a. Although they thought their result was the worst case cost, Huddleston and Mehlhorn did amortized analysis by using a bank savings account paradigm to simulate the operation sequence.

Insertion and deletion into (a,b)-trees are quite similar to the corresponding operations in (2,3)-trees and B-trees. An insertion means the addition of a new leaf (external node) at a given position in the tree and it is accomplished by a sequence of node expansions and node splittings, terminating when the (a,b)-tree is balanced. For a = 2 and b = 4, the insertion of a new rightmost leaf into a (2,4)-tree is shown in Fig.9. A deletion means the pruning of an existing leaf at a given position in the tree followed by a sequence of node shrinkings and node fusings and ended by possibly one node sharing. The deletion of a leftmost leaf in a (2,4)-tree is shown in Fig.10. Deletion has two parameters, the sharing threshold t which specifies when to

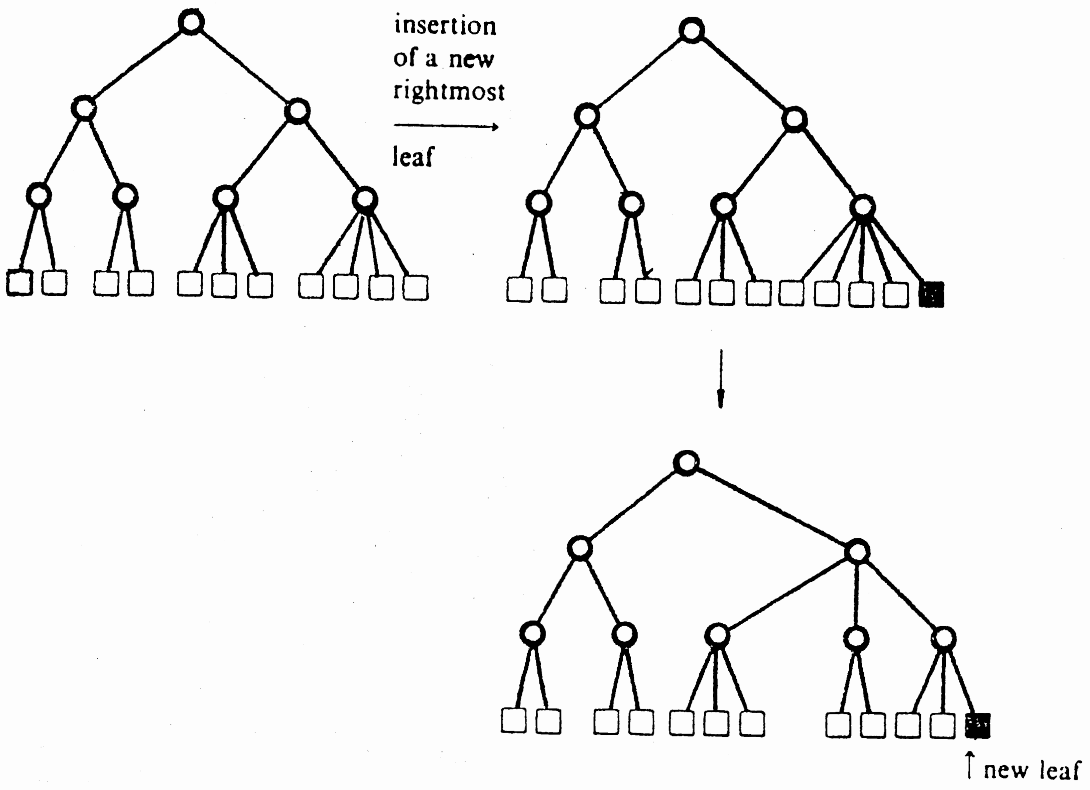


Figure 9. Insertion into an (a,b)-Tree
(a=2, b=4) [19]

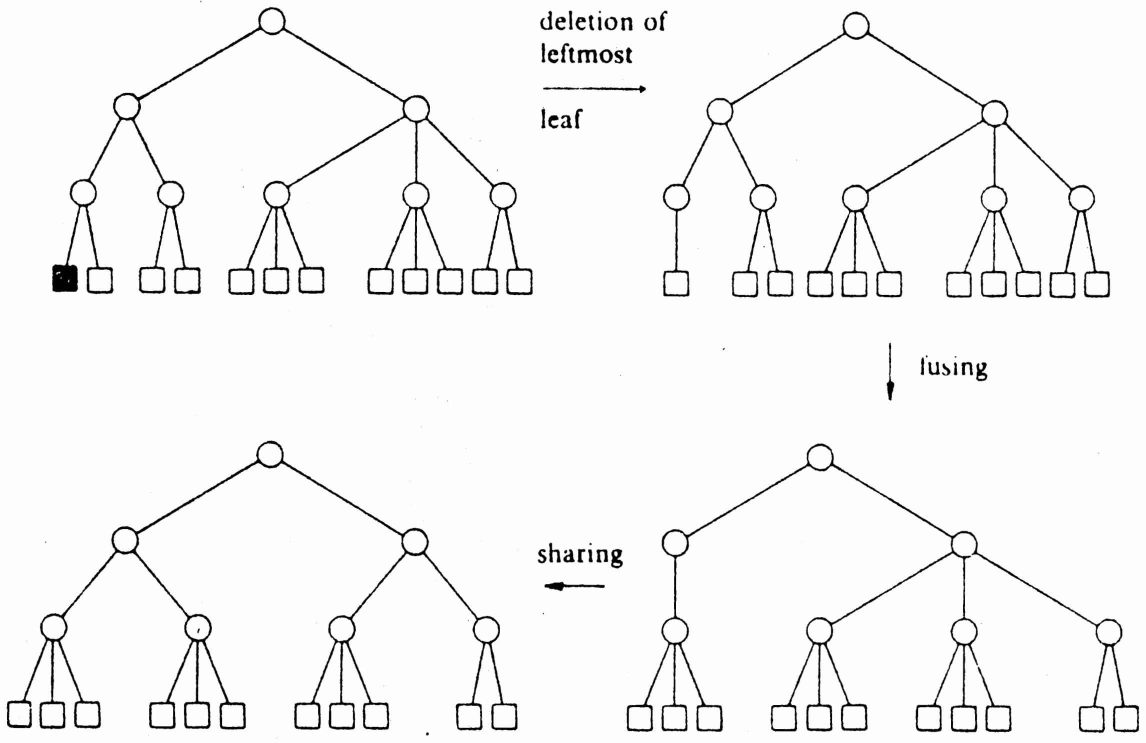


Figure 10. Deletion in an (a,b)-Tree
(a=2, b=4) [19]

fuse or share, and s which specifies how many sons to shift when sharing. In (a,b)-trees, any values of the parameters t and s in ($0 <= t <= b+1-2a$) and ($1 <= s <= t+1$) gives a correct rebalancing algorithm. Huddleston and Mehlhorn [19] considered two algorithms as follows.

Algorithm 1: uses $s = [(p+1)/2]$ and $t = p+s-1$ thus moves the arities of the balanced nodes as far away from the critical values a and b as possible and invests in the future (p is the hysteresis of (a,b)-trees, $p = [b/2] - a$).

Algorithm 2: uses $s = 1$ and $t = 0$ thus shares whenever possible and terminates rebalancing as soon as possible.

We define a savings account V for any (a,b)-tree T. Then V(T) is the sum of all V(x), where x is a node of T. Then, consider an arbitrary sequence of k intermixed insertions and deletions into an (a,b)-tree. Since initially the tree is empty, $V(T_0) = 0$. By algorithm 1, after k'th insertion or deletion, we have

$$0 <= V(T_k) <= k - p * B_k \qquad (2.15)$$

where B_k = the total number of rebalancing operations (splittings, fusings, and sharings)

Thus,

$$B_k >= (1/p) * k \qquad (2.16)$$

By algorithm 2, the account balance after the k'th operation satisfies

$$0 <= V(T_k) <= k - (2/3) * B_k \qquad (2.17)$$

Thus,

$$B_k >= (3/2) * k \qquad (2.18)$$

In either case, the amortized number of node splittings and fusings (or say the average cost of rebalancing) is $O(1)$ per operation instead of $O(\log n)$, provided that b >= 2a and that the (a,b)-tree is initially empty.

Red-Black Trees

A red-black tree is a binary search tree in which each internal node can be thought of as either red or black. The tree is balanced by satisfying the following color constraints:

1. all leaves (external nodes) are black;
2. black constraint: all paths from the root to a leaf contain the same number of black nodes;
3. red constraint: every red node except for the root has a black parent node.

Fig.11 presents a typical red-black search tree. The required rebalancing can be performed by either a bottom-up algorithm or a top-down algorithm.

Tarjan [36] and Chen [31] indicated two main advantages of the top-down update method, compared with the bottom-up

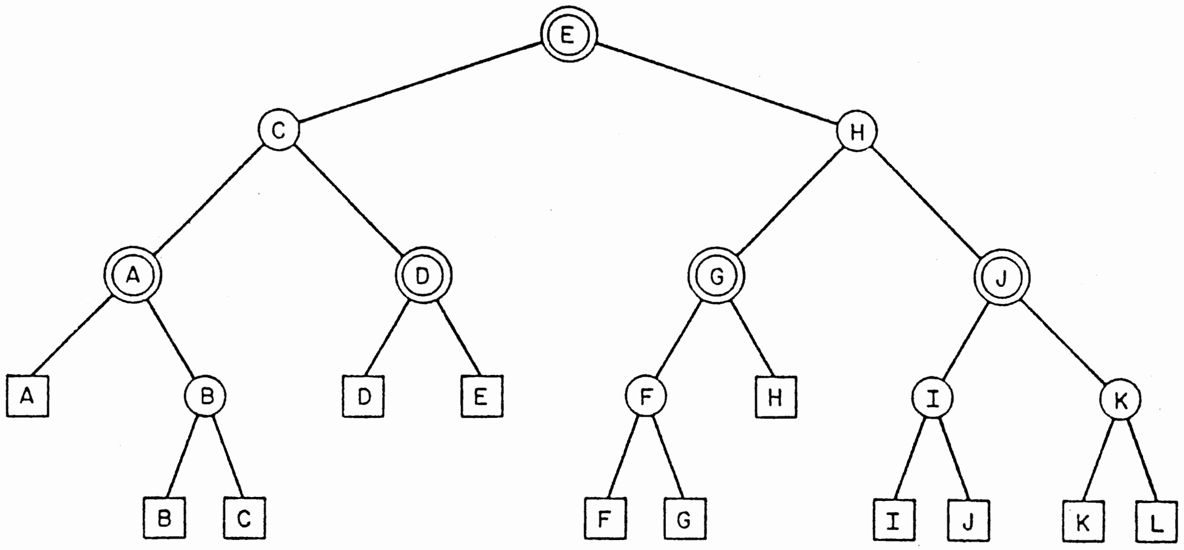


Figure 11. A Red-Black Search Tree [36]

algorithm. However, only the algorithm proposed by Tarjan [30,36], which is globally top-down but locally bottom-up, requires $O(1)$ rotations and color changes in the amortized case for top-down insertion and deletion in red-black trees.

Tarjan used both the banker's view [30,31] and the physicist's potential in analyzing the insertion and deletion updating. This method proceeds from the root down along the access path, while maintaining the invariant that the "starred" node is black and has at least one black son. To do this, we first set the root as the "starred" node X and change it to black if it is red or change both its children to black if they are both red. Then we walk from this node X down along the access path, until one of the following cases is encountered.

Case (1): a black node, say Y, with a black son is reached; then replace the "starred" node X by Y and walk down again.

Case (2): an external node is reached, then perform the bottom-up insertion until the "starred" node X is reached. For bottom-up insertion, we first replace the appropriate leaf by an internal node having two leaves, one containing the new item to be inserted and the other containing the item in the replaced node. The key of the new internal node is the minimum of the items in its two children. We color the new internal node red (See Fig.12a). This procedure preserves the black constraint but may violate the red con-

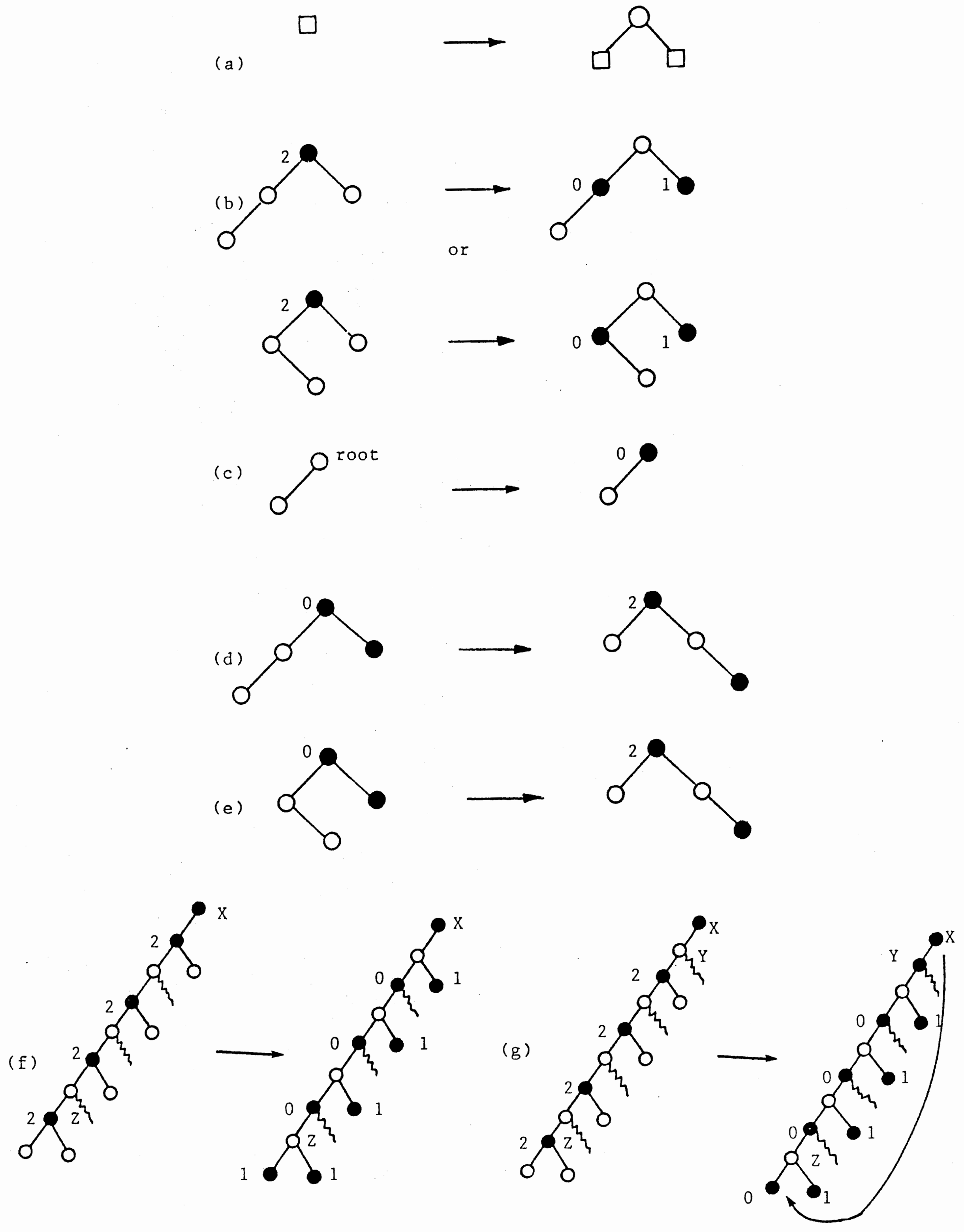


Figure 12. Insertion Operations into Red-Black Trees [36]

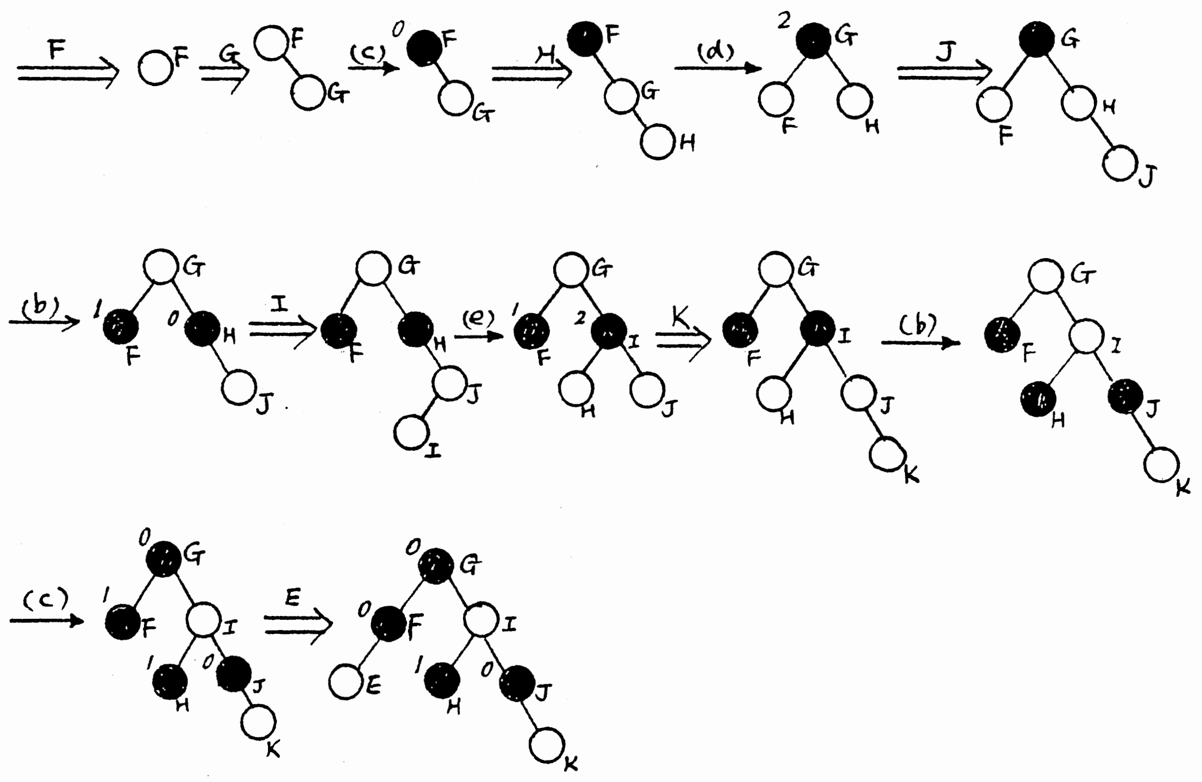
straint. So we go up to check X's parent P(X), P(X)'s sibling P'(X), and X's grandparent G(X). If a red node X has both red P(X) and P'(X), we color them both black and color the G(X) red (Fig.12b). This will cause a new violation of the red constraint if G(X)'s parent is red. We just repeat the recoloring step until no new violation is created or a red node x has a red parent P(X) that is the toot or whose sibling is black. To eliminate the last violation we apply the appropriate one of the transformations in Figures 12c, 12d and 12e.

Case (3): four successive black nodes, each with two red children, are reached along the top-down access path (Figs.12f and 12g). Let Z be the bottom-most black node. Color Z red and its two sons black, and then proceed the bottom-up insertion steps. Two possible situations are shown in Figs.12f and 12g, respectively. In Fig.12f, only color changes occur and this takes three applications of Fig.12b; while in Fig.12g, color changes followed possibly by a rotation and color change as in Fig.12c or 12d. The top-down procedure is continued by replacing the "starred" node X by the child of Z along the access path and continuing to walk down.

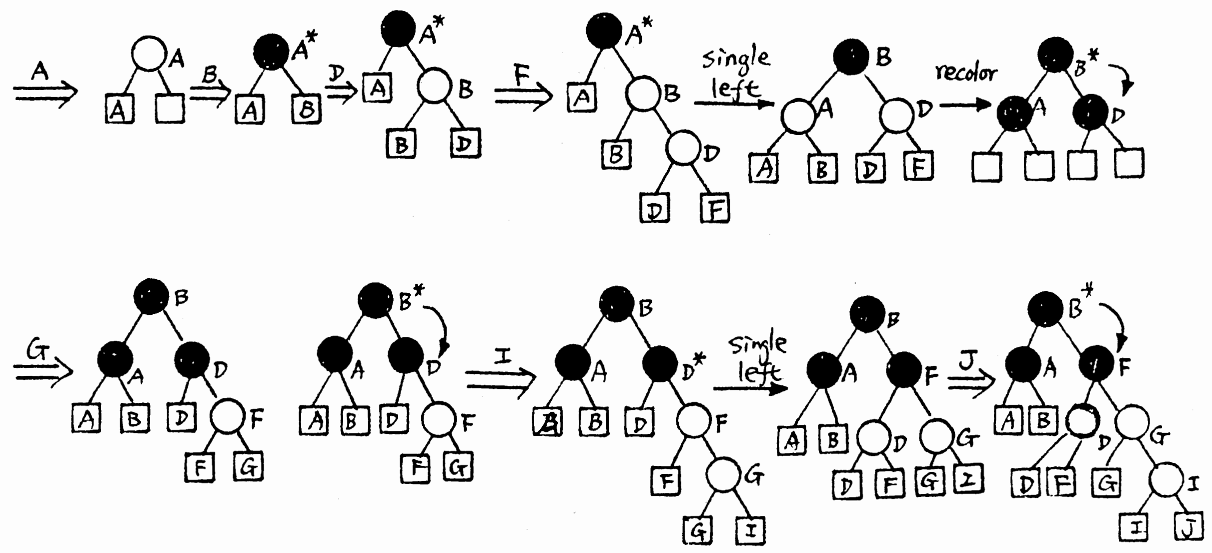
Now we can discuss the amortized complexity bound for the above algorithm in a red-black tree. From the banker's view, we assign 0, 1, or 2 credits to each black node according to whether it has 1, 0, or 2 red children, respectively. Only black nodes have credits. While from the physicist's view, we call the assigned credits for each black node the potential of that node. Then the savings account balance is the sum of credits deposited or withdrawn, or the potential of the tree is the sum of the potentials of all its nodes. The total actual cost of a sequence of update operations (algorithm complexity) is the total amortized cost minus the net increase in account balance (or tree potential) over the sequence.

Consider the account balance and potential values in insertion process. Case (1) does not change the tree, so the account balance (or tree potential) does not change either. In case (2), the insertion of a new item means to withdraw one credit (decrease potential by one), and the recolor and move-up (Fig.12g) means to withdraw one credit (decrease potential by one). In case (3), the single rotation (Fig.12d) or the double rotation (Fig.12e) spends at most two credits (deposit potential by two). Initially the account balance has $O(n)$ credits ($O(n)$ tree potential). Afterwards, a constant number of rotations and/or color changes is performed in case (2) or case (3), which means that the amortized cost of an insertion is $O(1)$ if we ignore the constant factor [36]. Two examples are presented in Figs.13a and 13b.

Deletion in a red-black tree is more complicated. However, with the help of the short node concept introduced by Tarjan [36], we can perform the top-down deletion very simi-



(a) bottom-up insertions



(b) top-down insertions

Figure 13. Example Insertions into Red-Black Trees

lar to insertion and conclude the same result that the amortized cost of a deletion is $O(1)$.

<u>HB[1]-Trees</u>

Another kind of balanced binary search tree is height balanced trees (HB[k]-trees) which are balanced by their height. The generalized case is an HB[k]-tree, which has the property that, for every node, the heights of the right and left subtrees differ by at most an integer k. An HB[1]-tree is also called an AVL-tree. The update algorithms for standard HB[k]-trees and AVL-trees are discussed by Knuth [45] and Chen [31], for weight leaf AVL-tree by Vaishnavi [41]. However, amortization was not used to analyze the algorithms.

Mehlhorn and Tsakalidis [35] analyzed the amortized behavior of AVL-trees under sequences of insertions by investigating the termination segment of the search paths during rebalancing. Their analysis can be described as follows. Let v be any internal node in an AVL-tree and let height(v) be the length of the longest path from v to a leaf. Let L(v) and R(v) be the left and right subtrees of the AVL-tree, respectively, so that v is the root of both L(v) and R(v), as illustrated in Fig.14a. If we define a parameter hb(v) to represent the height balance of node v, that is

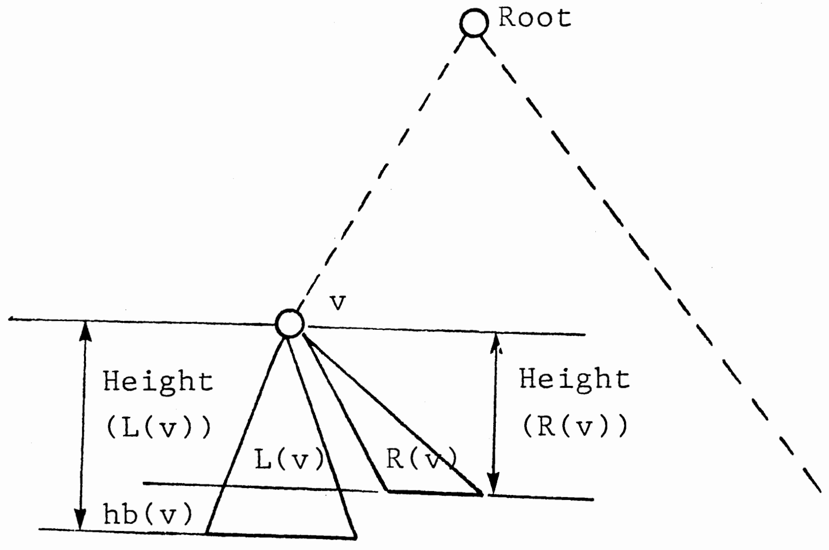
$$hb(v) = Height(R(v)) - Height(L(v)) \tag{2.19}$$

Then according to the definition of an AVL-tree, hb(v) will be +1, 0, or -1. We say node v is balanced (unbalanced) if hb(v) is 0 (+1 or -1).

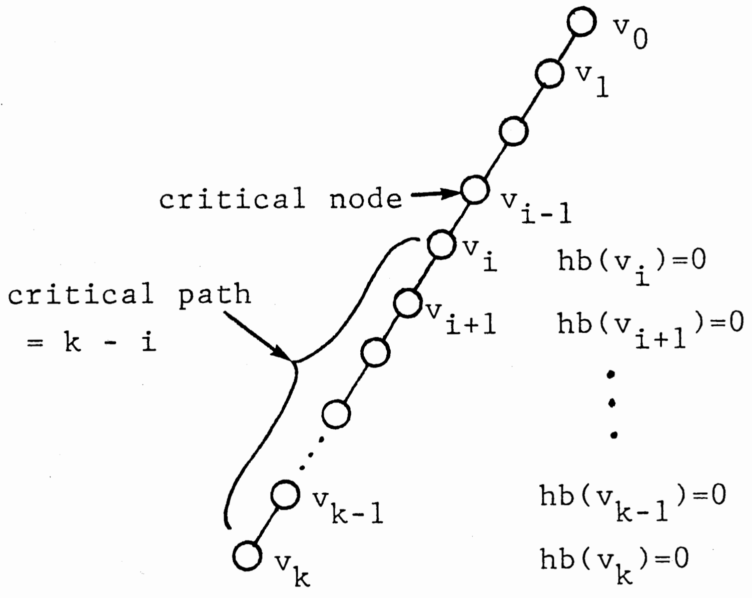
In addition, for every insertion, we specify a critical node which is the last unbalanced node on the search path from the root. Let v_0, v_1, ..., v_k be a path from the root v_0 to a leaf v_k in an AVL-tree (Fig.14b); i be the minimal node position such that $hb(v_i)=hb(v_{i+1})=...=hb(v_k)=0$. Then, for i >= 1, node v_{i-1} is the critical node of the path, below which every node on the path is balanced. The critical path is from v_i to v_k with a length of (k-i). For i = 0 there is no critical node and the insertion causes a height increase of the tree.

Basically, for each insertion, there are three steps associated with five operations (OP_1, OP_2,..., OP_5), and the total numbers of each operation are X_1, X_2, ..., X_5. When a new leaf is inserted at a given location, the old leaf at that location is replaced by an internal node with two leaves. Then the nodes on the critical path are changed from balanced to unbalanced (from hb=0 to hb=+1 or -1). This step (balance change, OP_1) is called A_6 by Knuth [45]. Finally, rebalancing is completed by absorption (OP_2), single rotation (OP_3) or double rotation (OP_4) at the critical node v_{i-1} (if i >= 1), or by tree height increase (OP_5) if no critical node exists.

Similar to that described for (2,3), (a,b), and red-black trees, we consider a sequence of k insertions into an



(a) hb(v) in AVL-Tree



(b) critical node and critical path

Figure 14. An HB[1]-Tree

initially empty AVL-tree T_0. After the ith insertion including necessary rebalancings ($0 <= i <= k$), the tree is T_i. Every insertion terminates in an absorption (OP_2), single rotation (OP_3) or double rotation (OP_4), or height increase (OP_5), that is

$$X_2 + X_3 + X_4 + X_5 = k \tag{2.20}$$

where x_j = the total number of OP_j ($j = 2,3,4,5$).

Therefore, the operation cost depends mainly upon X_1, which is the total number of node balance changes in step A_6. To estimate X_1, we first express X_1 in terms of the total length of all the critical paths d_i ($0 <= i <= k$),

$$X_1 = \sum_{i=1}^{k} d_i \tag{2.21}$$

Then we define the total number of unbalanced nodes in AVL-tree T_i to be $Val(T_i)$, which can be estimated for operations OP_1, OP_2, OP_3, and OP_4. Suppose an AVL-tree T_{i-1} is changed to T_i, after the ith insertion into the left subtree of the critical node v_{i-1} and followed by one of the following operations,

OP_2: absorption operation as shown in Fig.15a, the number of unbalanced nodes in tree T_i is

$$Val(T_i) = Val(T_{i-1}) + d_i - 1 \tag{2.22}$$

OP_3: single rotation as shown in Fig.15b,

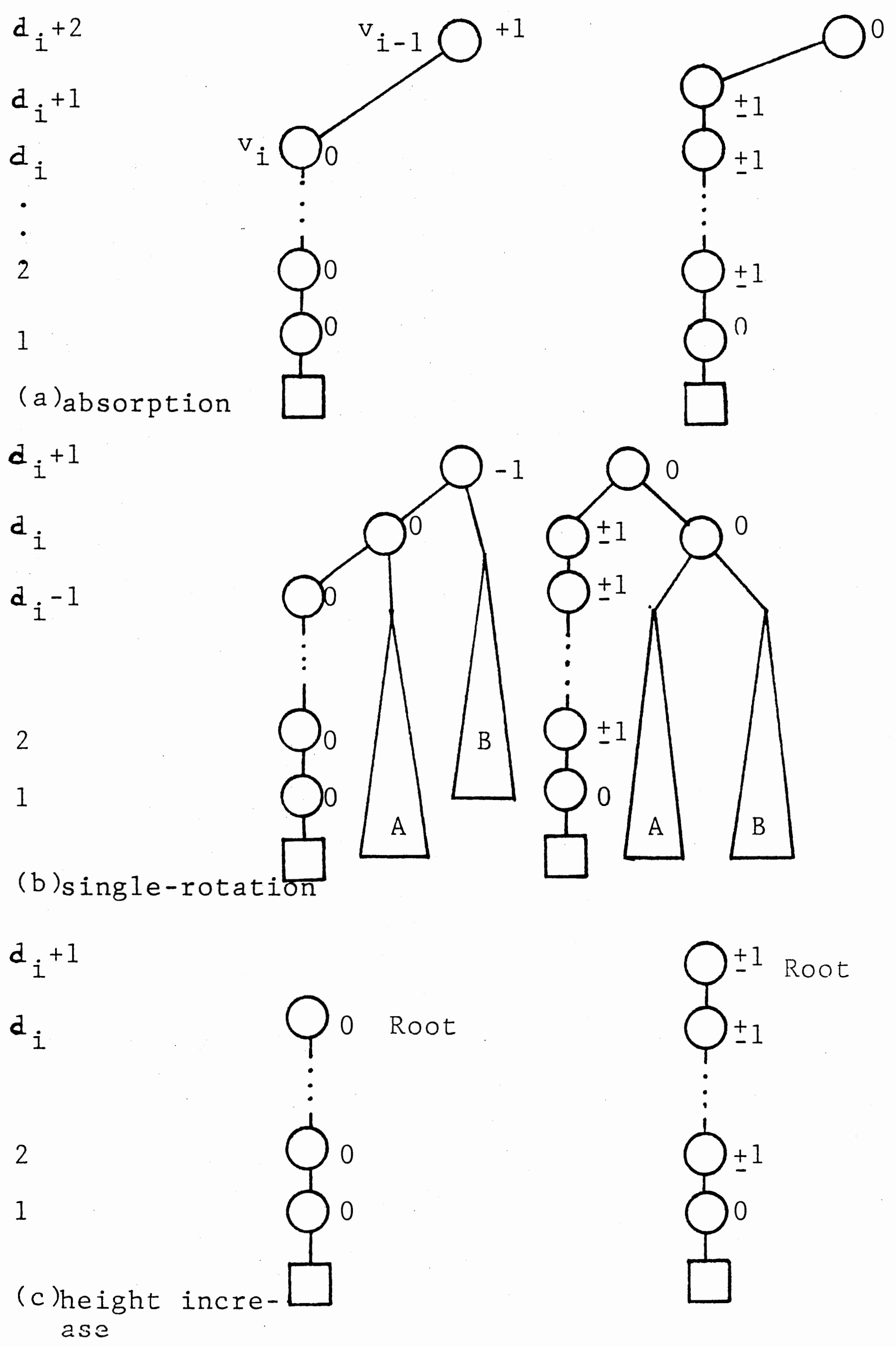
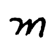
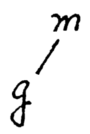
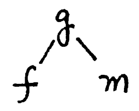
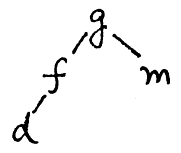
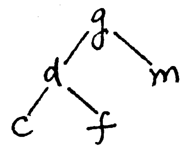
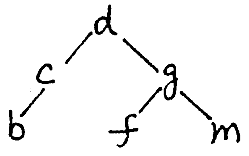
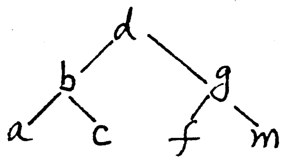


Figure 15. Calculation of Unbalanced Nodes in AVL-Trees [35]

	Sequential Number	Key	Operation	d_i	$Val(T_i)$
	1	m	OP5	1	$V(T_0)+d_1=1$
	2	g	OP5	2	$V(T_1)+d_2=3$
	3	f	OP3	2	$V(T_2)+d_3-2=3$
	4	d	OP5	3	$V(T_3)+d_4=6$
	5	c	OP3	2	$V(T_4)+d_5-2=6$
	6	b	OP3	3	$V(T_5)+d_6-2=7$
	7	a	OP3	2	$V(T_6)+d_7-2=7$

(d) Example of sequential insertions

Figure 15. (Continued)

$$Val(T_i) = Val(T_{i-1}) + d_i - 2 \tag{2.23}$$

OP_4: double rotation which is analogous to OP_3.

OP_5: height increase as shown in Fig.15c,

$$Val(T_i) = Val(T_{i-1}) + d_i \tag{2.24}$$

(here d_i equals to the root height of tree T_i).

Thus, after applying the recursive Eqs.(2.22), (2.23), and (2.24) k times and combining Eqs.(2.20) and (2.21), we have the total rebalancing number (length) X_1 after k insertions,

$$X_1 = Val(T_k) + k + (X_3 + X_4 - X_5) \tag{2.25}$$

Knuth [45] proved that the number of unbalanced nodes in an AVL-tree with k leaves is about 0.618(k-1), hence

$$X_1 <= 2.618\ k \tag{2.26}$$

This result implies that the amortized cost is $O(1)$ per insertion into an AVL-tree.

Heap Amortization

The heap or priority queue is a more complicated tree structure consisting of a collection of items, each with an associated real-valued key [3]. Items in the tree are arranged in heap-order (Fig.16) so that the root of the tree always contains an item of minimum key and this item can be found in $O(1)$ time by accessing the root (operation

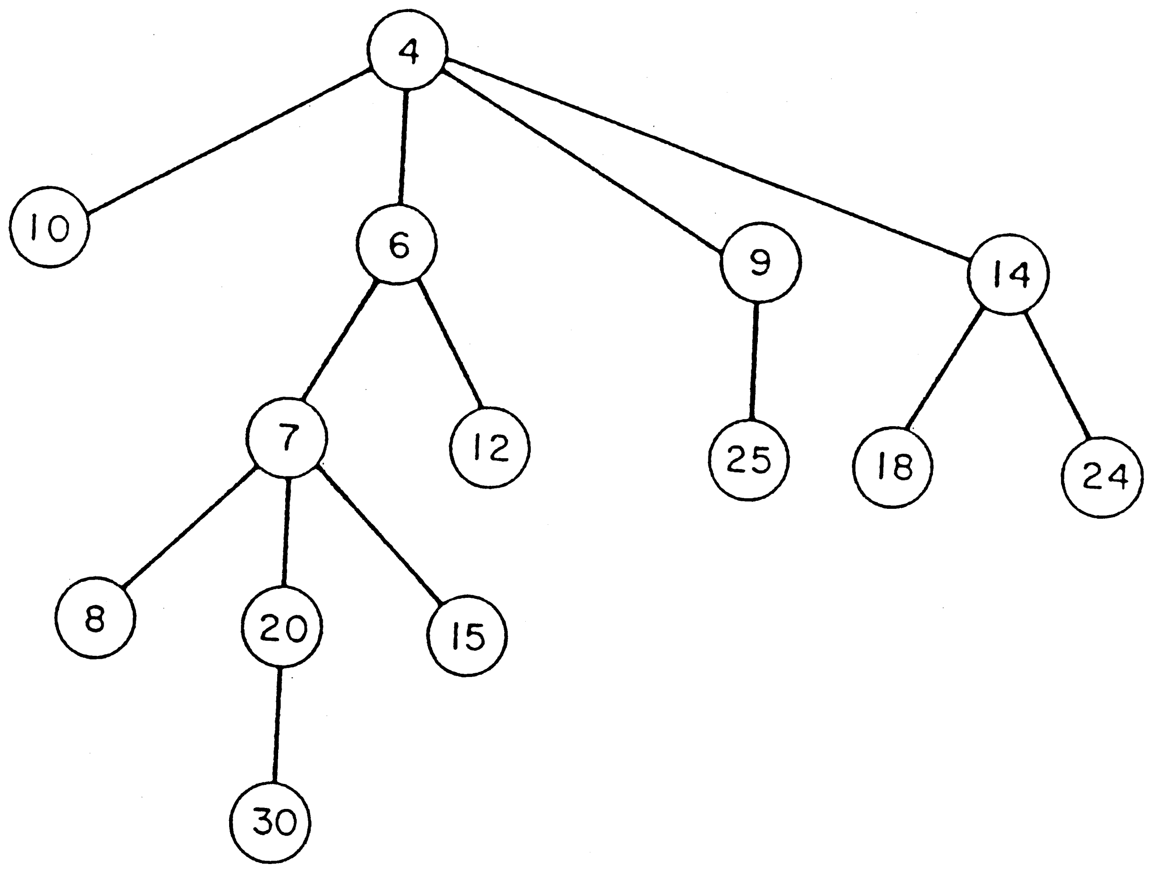


Figure 16. An Endogenous Heap-Order Tree [37]

FINDMIN). Other possible heap operations include INSERT, DELETEMIN, MAKEHEAP, DELETE, and MELD, etc. Fisher [33] summarized several heap implementations for different applications, such as

1. the d-heap for use when few unmeldable heaps are needed,
2. the leftist heap for use when several meldable heaps are needed,
3. the Fibonacci heap for use in improved network optimization algorithms,
4. the pairing heap which is an improvment of the Fibonacci heap and is self-adjusting in nature,
5. the self-adjusting heap, also called skew heap, for simple, fast, and space-saving applications.

Although the operations of these heaps have been well discussed previously, the complexity analysis from the view of amortization is very limited, covering only the pairing heap [37] and self-adjusting [38].

Pairing Heaps

The pairing heap is a new heap form developed by Fredman, Sedgewick, Sleator, and Tarjan, intended to be competitive with the Fibonacci heap in efficiency and easy to implement and fast in practice. The basic operations in a pairing heap h are as follows.

OP_1. MAKEHEAP(h): create a new empty heap called h.

OP_2. FINDMIN(h) : return the root of the heap h.

OP_3. INSERT(x,h): construct a one node tree containing x and link it with h.

OP_4. MELD(h_1,h_2): return the heap formed by linking two heaps h_1 and h_2. (h_1 and h_2 are destroyed)

OP_5. DECREASEKEY(b,x,h): decrease the key of item x in h by subtracting the positive real number b.

OP_6. DELETEMIN(h): delete an item of minimum key from h and return it (if h empty, return null).

OP_7. DELETE(x,h): delete item x from h which contains the x previously.

The pairing heap algorithm is obtained by using the pairing method to combine trees and choosing the trees to be paired carefully. We arrange the children of each node in the order that they were attached by linking operations, with the first child being the one most recently attached. This ordering of children is independent of key order in the heap. To perform the operation OP_4 (DELETEMIN), we first remove the root and link the first and second remaining subtrees, then link the third and fourth subtrees, and so on. If the original root had an odd number of subtrees, one remains unlinked. Then we link each remaining subtree to the last one, working from the next-to-last back to the first, in the opposite order to that of the pairing pass, as shown in Fig.17a. In this way, the pairing heap can provide as efficient updating as that provided by Fibonacci heaps. The pairing heap can also be built up in another way as illustrated in Fig.17b. After the first pairing path the subtrees are connected backwards. The resulting heap is tall and akinny so that the following DELETEMIN operation is even faster. We call these two pairing ways as method A and B, respectively.

Fredman et al [37] used the potential technique and provided a partial amortized analysis for those operations on pairing heaps. Their results were described by Fisher [33] in more detail. Define the potential of a node x with d children in a n-node heap h to be

$$\varphi_x = 1 - \min \{ d_x, \sqrt{n_x} \} \tag{2.27}$$

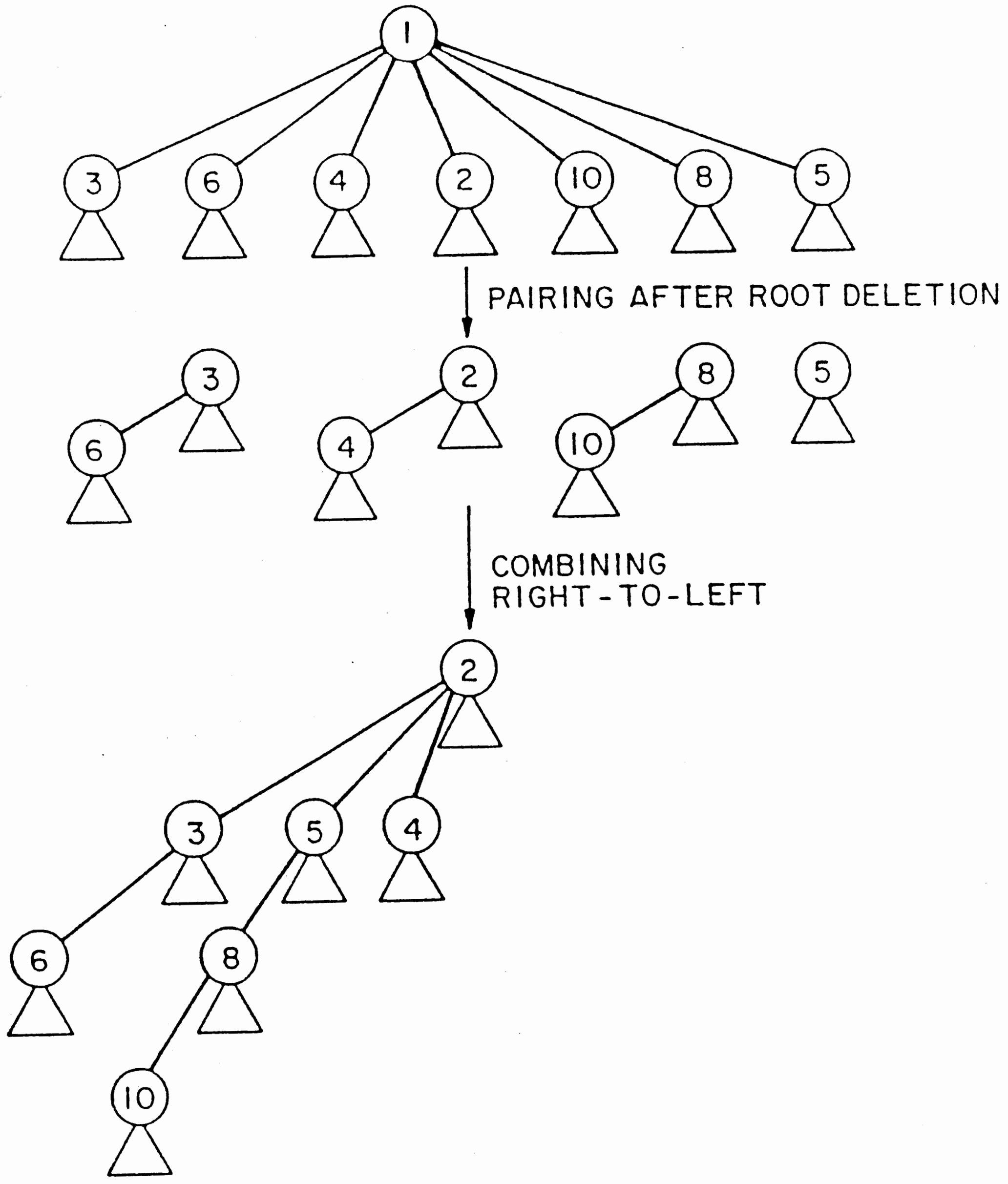
where d_x = the number of successors of node x,

n_x = the number of internal nodes in the subtree rooted at node x.

The potential for h is

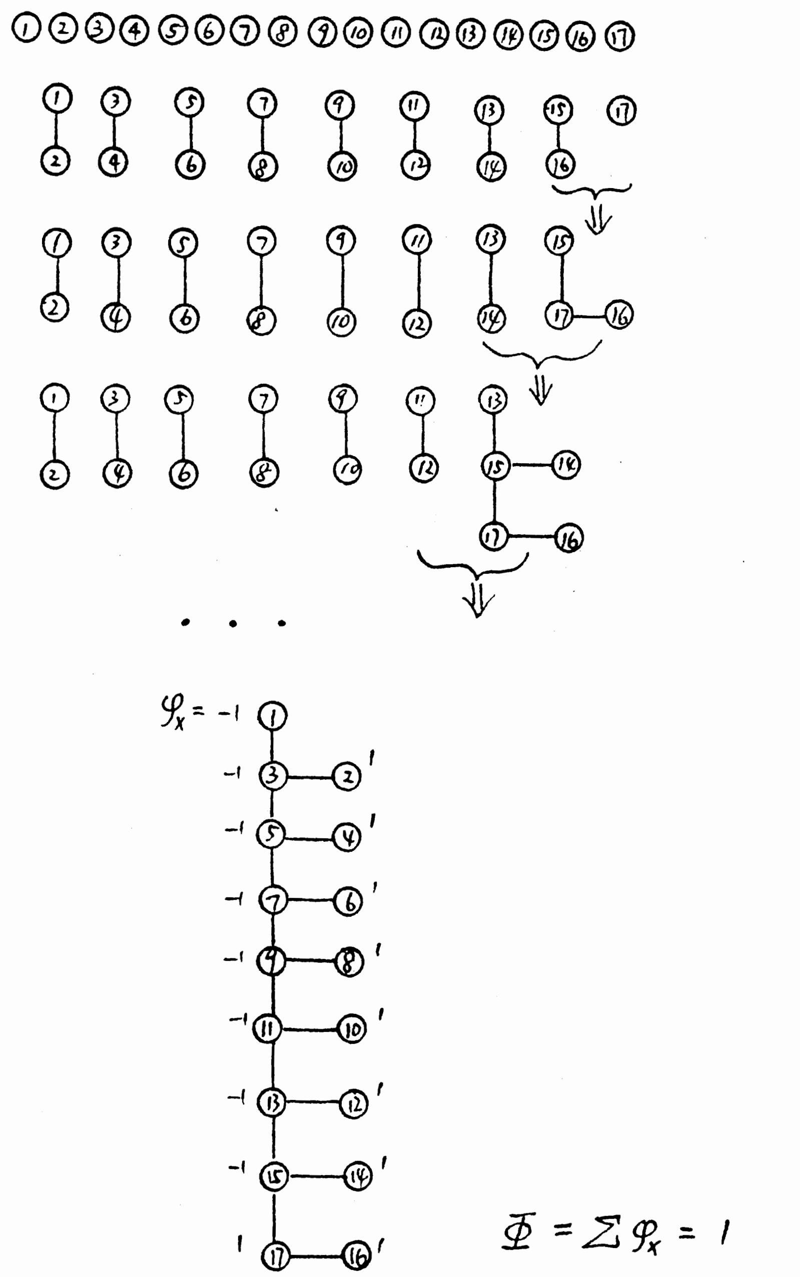
$$\phi = \sum_{x \in h} \varphi_x = \sum_{x \in h} (1 - \min \{ d_x, \sqrt{n_x} \}) \tag{2.28}$$

The potential function of both the nodes and the heap are indicated in Fig.17b and 17c for a sample heap of 17 items, using either method A or method B. Now we use this potential function to analyze the amortized cost of the DELETEMIN operation, which includes the actual time and potential change during this process. Suppose K trees remain

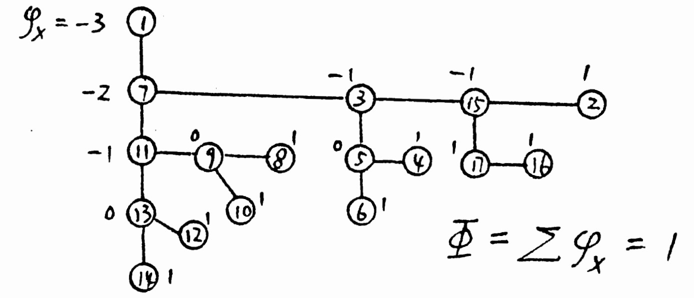


(a) A DELETEMIN Operation in a Pairing Heap [37]

Figure 17. Pairing Heap Operation



(b) Potential function by pairing method B



(c) Potential function by method A

Figure 17. Continued.

after the DELETEMIN. The actual time of the DELETEMIN is $O(K)$. We ignore the constant factor and have the time as

$$t = 1 + \lfloor K / 2 \rfloor = 1 + (\text{ number of Link operations in Pairing path }) \tag{2.29}$$

The potential change due to the DELETEMIN operation includes two parts : removing the root node causes at most $2\sqrt{n}$ potential increase and linking trees causes at most $(\lfloor K/2 \rfloor - \sqrt{n})$ potential decrease. Therefore, the amortized time of DELETEMIN is

$$\begin{aligned} a &= t + (\phi_f - \phi_o) \\ &= O(1 + \lfloor K / 2 \rfloor + 2\sqrt{n} + (\sqrt{n} - \lfloor K / 2 \rfloor)) \\ &= O(\sqrt{n}) \end{aligned} \tag{2.30}$$

It is seen that the number of subsequent relinks after deleting the minimum item 1 in Fig.17b is different than that in Fig.17c, although the heap potential function is the same in both cases. Using method B results in less subtrees so that the relink is faster; while the shorter, fatter heap produced by method A has more subtrees to be linked, leading to a pessimistic amortized time $O(\sqrt{n})$. Therefore, Fredman et. al. tried another form, the log size, as the potential function. Let the size S(x) of a node x in a binary tree be the number of nodes including x itself in its subtree. Then we use the log S(x) to represent the potential of node x so that every node in an n-node tree has a potential between 0 and log n, as shown in Fig.18a. The potential of a heap h (a set of trees) is the total potentials of all the nodes in the tree.

The amortized costs of most heap operations except the DELETEMIN (OP_6) can be easily estimated based on this potential function. MAKEHEAP and FINDMIN have $O(1)$ amortized time bound because the heap potential is not changed by these operations. INSERT, MELD, and DECREASEKEY have $O(\log n)$ amortized time bound, since each of these operations causes the potential to increase at most $O(1+\log n)$ (Fig.18b).

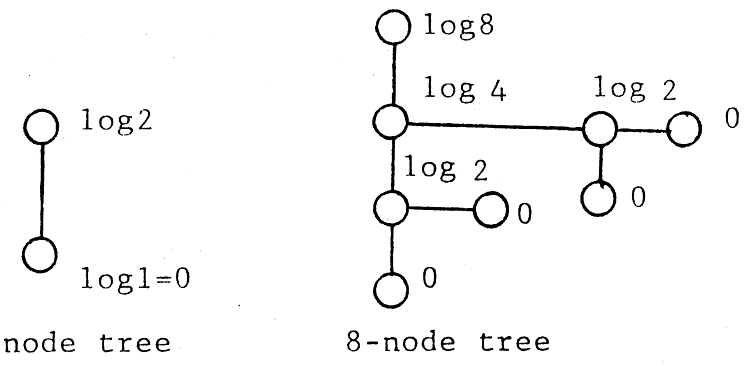
To consider the DELETEMIN operation (removing the root, then pairwise combining the remaining trees), we still use Eq.(2.29) to estimate the actual time; but the potential change due to links in this operation is analyzed as follows.

We first estimate the potential change caused by the first-pass link (pairing), illustrtated in Fig.19. In this figure, A, B, and C are subtrees; a, b, c, x, and y are nodes.

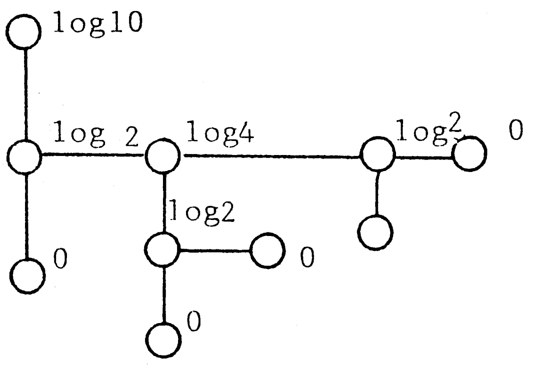
The initial potential is

$$\begin{aligned}\phi_o = {} & \log (S(a)) + \log (S(b)) + \log (S(c)) \\ & + \log (1 + S(b) + S(c)) \\ & + \log (1 + S(a) + 1 + S(b) + S(c)) \qquad (2.31)\end{aligned}$$

The final potential is,



(a) node potentials



(b) potential increase due to MELD

Figure 18. Log Size as Potential Function in Pairing Heaps [33]

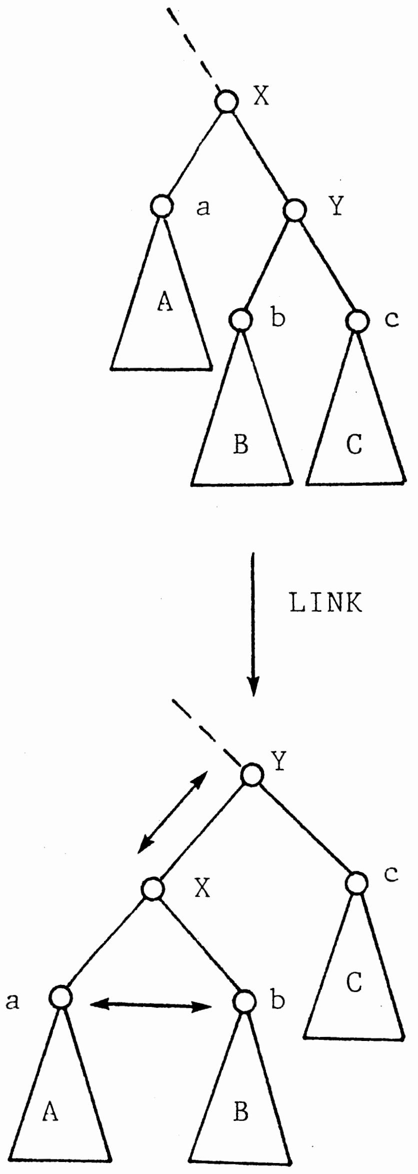


Figure 19. LINK during DELETEMIN [37]

$$\phi_f = \log (S(a)) + \log (S(b)) + \log (S(c)) + \log (1 + S(a) + S(b)) + \log (1 + S(c) + 1 + S(a) + A(b)) \qquad (2.32)$$

Therefore, the increase of potential is

$$\phi_f - \phi_o = \log (1 + S(a) + S(b)) - \log (1 + S(b) + S(c)) \qquad (2.33)$$

This potential increase is proved [33] to be bounded by

$$\phi_f - \phi_o <= 2 * \log (S(a) + S(b) + S(c) + 2) - 2 * \log (S(c)) - 2 \qquad (2.34)$$

In case that the last link operation during the first pass has an empty subtree c, the potential increase is bounded by

$$\phi_f - \phi_o = \log (1 + S(a) + S(b)) - \log (1 + S(b)) <= 2 * \log (S(a) + S(b) + 2) \qquad (2.35)$$

We sum the bound (2.34) of potential increase over K first-pass links and obtain a total bound of 2*(log(n)-2*K-2).

The other potential changes that take place during the DELETEMIN are a decrease of log(n) due to removing the original root node and an increase of at most log(n-1) due to the second pass (combining remaining trees). Since the number of links performed during second-pass is at most as great as that during the first-pass, we obtain the amortized time of DELETEMIN as

TABLE I

AMORTIZED COSTS OF PAIRING HEAP OPERATIONS

	Operation	Conjectural estimation	Weaker estimation
OP_1	Make heap	O(1)	O(1)
OP_2	Find min	O(1)	O(1)
OP_3	Insert	O(1)	O(log n)
OP_4	Meld	O(1)	O(log n)
OP_5	Decrease key	O(1)	O(log n)
OP_6	Delete min	O(log n)	O(log n)
OP_7	Delete	O(log n)	O(log n)

$$
\begin{aligned}
a &= t + \phi_f - \phi_o \\
&<= (1 + 2*K + 2*\log n - 2*K + 2 - \log n + \log(n - 1)) \\
&= O(\log n) \qquad (2.36)
\end{aligned}
$$

The amortized costs of pairing heap operations are listed in Table I. Also listed is a conjectural result which gives $O(1)$ for INSERT, MELD, and DECREASEKEY.

Self-adjusting heaps

Standard kinds of data structures, such as the many varieties of balanced trees, are specifically designed so that the worst-case time per operation is small. Such efficiency is achieved by imposing an explicit structure constraint that must be maintained during update. However, maintaining such a structural constraint consumes both running time and storage space, and even tends to produce complicated updating algorithms with many cases. With the self-adjusting data structure, during each access or update operation we allow the data structure to be in an arbitrary state and adjust the structure in a simple, uniform way instead of imposing any explicit structural constraint [38]. Fisher discussed both the advantages and the disadvantages in using the self-adjusting data structure. One thing important is that the self-adjusting data structure is efficient in only an amortized sense. That means if the amortized running time is the complexity measure of interest (as it is always true in application), we can guarantee that the self-adjusting structure is at least as efficient as balanced structures.

Sleator and Tarjan [38] analyzed amortized behavior of a self-adjusting form of heap, called a skew heap. The operations allowed are MAKEHEAP, FINDMIN, INSERT, DELETEMIN, and MELD, with the same definitions as those described for the pairing heap. However, the fundamental operation on skew heaps is melding, which combines two disjoint heaps into one. To perform MELD in a self-adjusting heap as shown in Fig.20, we merge the right paths of the two heaps and then swap the left and right children of every node on the merge path except the lowest. This makes the potentially long right path formed by the merge into a left path.

To analyze the amortized time for skew heap operations, Sleator and Tarjan used the node weight to define the potential. So the potential of a skew heap is the total number of right heavy nodes it contains. The meanings of heavy and right are explained as follows.

For any node x in a binary tree, we define the weight wt(x) as the number of descendants of x (including x itself). Any nonroot node x is heavy if wt(x) > wt(p(x))/2, or it is light if otherwise. A nonroot node is right if it is a right child, or it is left otherwise.

With these definitions, we know that any node has at most one heavy child and that any path in a skew heap, and in particular any path traversed during melding, contains only O(log n) light nodes.

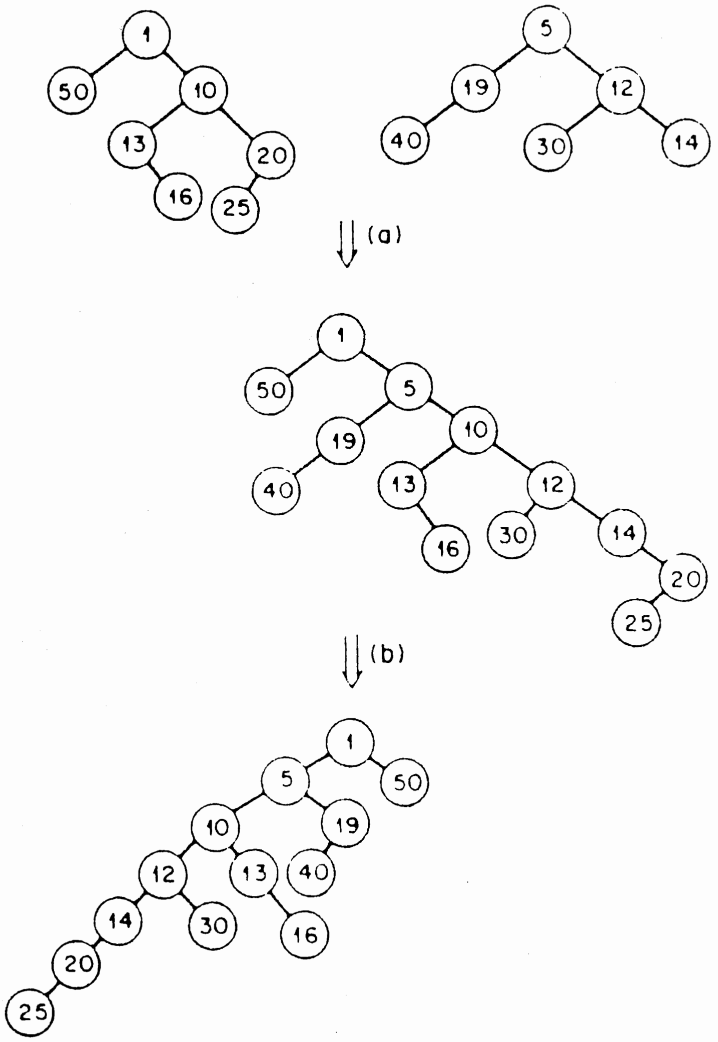


Figure 20. A MELD of Two Skew Heaps [38]

Similar to the analysis for operations in pairing heaps, we estinate that the amortized time of a MAKEHEAP or a FINDMIN operation in self-adjusting heaps is $O(1)$, since the heap potential does not change. But the times of the other four operations depend on the cost of MELD.

Consider a meld of two heaps h_1 and h_2, containing n_1 and n_2 items, respectively. Let n be the total item number in the two heaps. To measure the MELD time, we charge one unit per node on the merge path so that the amortized time of a MELD operation is the number of nodes on the merge path plus the potential change. Since the number of light nodes on the right path is at most $\lfloor \log n_1 \rfloor$ for h_1 and at most $\lfloor \log n_2 \rfloor$ for h_2 (Fig.21), the number of nodes on the merge path is bounded by

$$\begin{aligned} t &<= 2 + \lfloor \log n_1 \rfloor + K_1 + \lfloor \log n_2 \rfloor + K_2 \\ &<= 1 + 2 * \lfloor \log n \rfloor + K_1 + K_2 \qquad (2.37) \end{aligned}$$

where K_1 = number of heavy nodes on right path of h_1

K_2 = number of heavy nodes on right path of h_2

Obviously, the number of heavy nodes on the right path after melding is at most $\lfloor \log n \rfloor$. Hence, the amortized time of a MELD is given by

$$\begin{aligned} a &= t + \phi_f - \phi_o \\ &<= (1 + 2*\lfloor \log n \rfloor + K_1 + K_2) \\ &\qquad + (\lfloor \log n \rfloor - K_1 - K_2) = O(\log n) \qquad (2.38) \end{aligned}$$

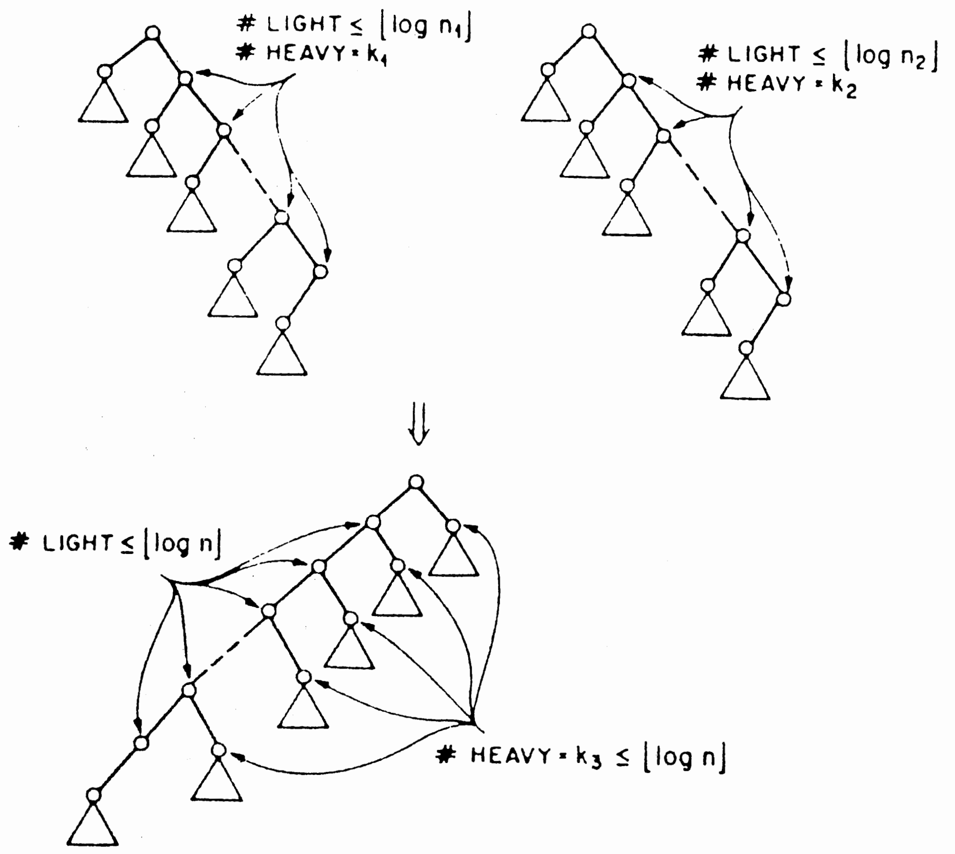


Figure 21. Analysis of Right Heavy Nodes in MELD [38]

TABLE II

AMORTIZED COSTS OF SELF-ADJUSTING HEAP OPERATIONS

	Operation	Bottom-Up	Top-Down
OP_1	Make heap	O(1)	O(1)
OP_2	Find min	O(1)	O(1)
OP_3	Insert	O(1)	O(log n)
OP_4	Meld	O(1)	O(log n)
OP_6	Delete min	O(log n)	O(log n)
OP_7	Delete	O(log n)	O(log n)

The bound for INSERT or DELETEMIN follows immediately from the bound of MELD and is alse $O(\log n)$.

Sleator and Tarjan also developed a bottom-up melding algorithm in self-adjusting heaps, which reduces the amortized time of INSERT and MELD to $O(1)$. But the bottom-up skew heap requires a more complicated potential function than the one defined in the above heap algorithm (the top-down skew heap). The heap potential is defined to be the number of right heavy nodes in the heap plus twice the number of right light nodes on the major path (the right path descending from the root) and the minor path (the right path descending from the left child of the root).

Table II summarizes the amortized complexity for both top-down and bottom-up skew heaps. These results scratch the possibility of using different potential functions to study the amortized behaviors for self-adjusting heaps.

CHAPTER III

POTENTIAL FUNCTIONS

The thorough literature survey in chapter II shows that the amortized analysis is an appreciable measure of algorithm complexity for data structures, because in most applications of data structures a sequence of operations rather than a single operation is frequently encountered. Amortization is a performance evaluation method that sums the total cost of the sequential operations and then averages the running time per operation over the sequence. This method can yield complexity measures both more realistic and more robust than that of the worst-case and average-case methods. However, the time complexity for a single operation from the sequence can be much greater than the amortized time which implies that amortized bounds may not be appropriate for real time applications.

Although the definition of amortization is simple and straightforward, the amortized costs of updating operations in different data structures have been determined quite differently, such as the epoch interval approach in (2,3)-trees, the savings account method in (a,b)-trees, and the total terminal segment length in AVL-trees, etc. Among all the published investigations, Tarjan's theory is most fundamental and mathematically precise considered either from banker's view or from physicist's view. The general approach is to define a potential function ϕ(D) which reflects the configuration of a data structure and then to estimate the amortized cost by calculating the potential change due to the operation. The amortized time thus provides an upper bound on the actual running time.

The utility of this method, however, strongly depends on the ability to choose a potential function that results in small amortized times for the operations. Although in recent years, several potential functions have been successfully used by Tarjan et. al., such as the inversion numbers in self-adjusting lists, node credits in red-black trees, log size in pairing heaps, and number of right heavy nodes in self-adjusting heaps, the discussion of selecting a suitable potential function for each structure is limited. The potential function method has not been widely applied to search trees. To measure the performance accurately and to promote the amortization method, the possibilities and limitations of various potential functions for different data structures need to be explored.

Fisher [33] summarized Tarjan's theory and indicated that, among many possible ones, there are three classes of potential functions which have more promise because of easy implementation and calculation. These three potential functions are the rank of item or node, sum children, and log sum children. Another potential function, the height in structure, was also proposed by Fisher and Hu [34] for the same reason. Let x be an item or a node in a data structure T, the four potential functions can be described as follows:

(PF1) rank(x)

Define a rank(x) for each x and use it to represent the potential of x (Fig.22a), then the potential of structure T is the sum of all the item potentials and expressed as

$$\phi_R(T) = \sum_{i=1}^{n} \mathrm{Rank}(x_i) \tag{3.1}$$

This is a generized method since the rank(x) can be defined in any possible ways such that the potential function can be easily determined. Some examples are the access frequency, inversion number, height difference, etc.

(PF2) sum x's children including x itself

This method is a special case of the rank method. For each x, find out its size S(x), where S(x) is the number of x's children including x itself, and use S(x) to represent the potential of x (Fig.22b). Then T's potential is the sum of all the potentials of x and expressed as

$$\phi_S(T) = \sum_{i=1}^{n} S(x_i) \tag{3.2}$$

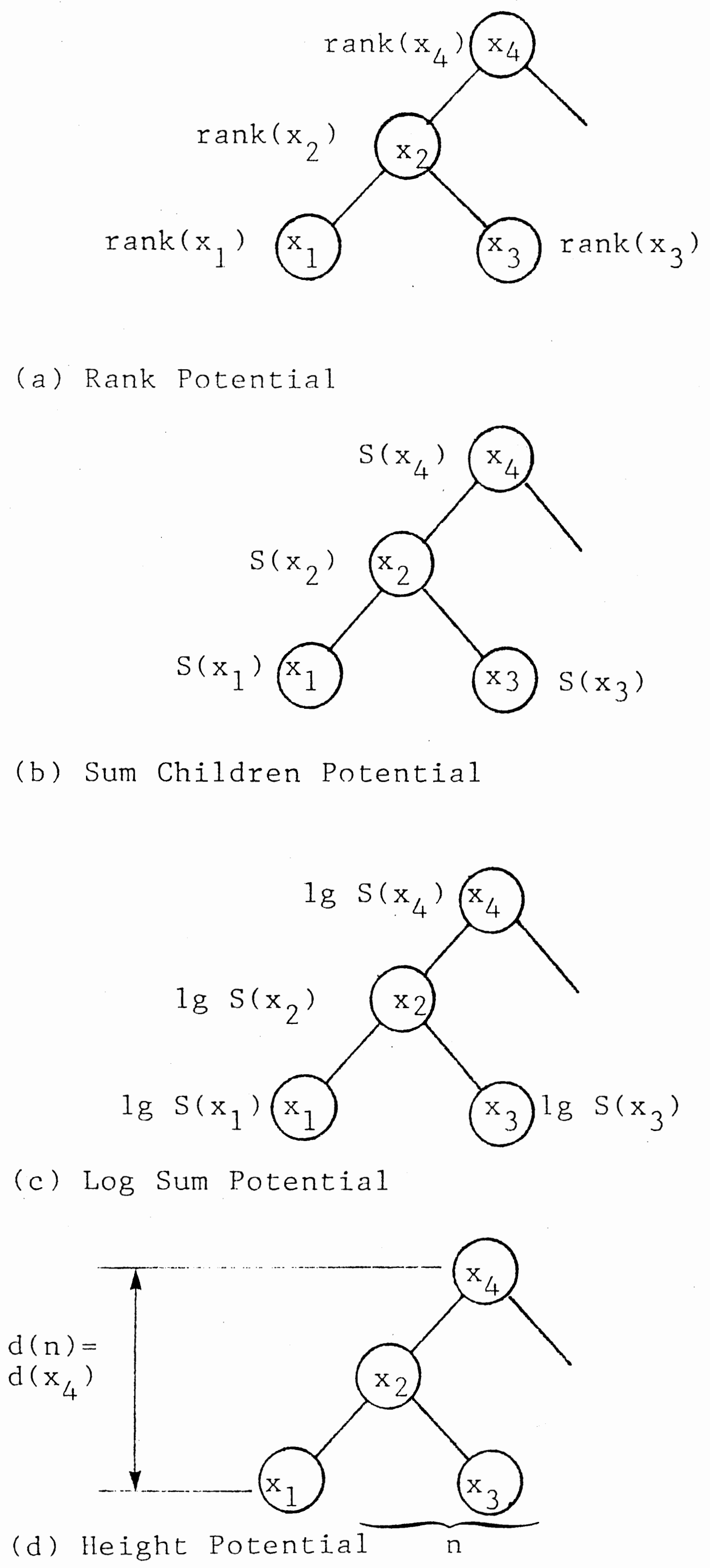


Figure 22. Four Selected Potential Functions

(PF3) log sum

Use log S(x) instead of S(x) to represent the potential of x, where S(x) is the size of x defined as above(Fig.22c); and the potential of T is expressed as

$$\phi_L(T) = \sum_{i=1}^{n} \log S(x_i) \tag{3.3}$$

(PF4) height

Use the structure height d(n) to represent the potential of T, where d(n) is a function of items or nodes in the structure (Fig.22d).

$$\phi_H(T) = d(n) \tag{3.4}$$

In the next three chapters, the amortized complexities of updating operations in linear list, balanced search trees, and heaps are analyzed based on these four potential functions.

CHAPTER IV

LINEAR LIST DATA STRUCTURES

Operations

A list is a sequence of arbitrary elements, some of which may be repeated. We represent a list q by an array [x] and denote the q by

$$q = [\ x_1,\ x_2,\ \dots\ ,\ x_n\ ]$$

where x_i = arbitrary array element representing an item i in the list

Element x_1 is the head of the list (first entered element) and x_n is the tail (last entered element). Both x_1 and x_n are ends of list q. There are three fundamental operations on lists [3]:

Access: given a q and an integer i, access and return the ith element $q[i]=x_i$; if $i<1$ or $i>n$, return Null;

Sublist: given a q and two integers i and j within the range of 1 to n, return $q[i,\dots,j]=x_i,\dots,x_j$;

Concatenation: given two lists $q=[x_1,\dots,x_n]$ and $r=[y_1,\dots,y_m]$, then return $q\&r = [x_1,\dots,x_n,y_1,\dots,y_m]$.

The insertion and deletion in lists can be performed by appropriate combinations of sublist and concatenation.

Especially important are six special cases of the three basic operations that manipulate the two ends of a list:

Access head: given a q, access and return $q[1]=x_1$;

Push: add a new item p to the tail of q, and return

$q\&p = [x_1,...,x_n,p]$;

Pop: remove the tail item $q[n]=x_n$, return $q=[x_1,..,x_{n-1}]$.

Access tail: access and return $q[n]=x_n$;

Inject: add a new item p to the head of q, and return

$p\&q = [p,x_1,...x_n]$;

Eject: remove the head item $q[1]=x_1$, return $q=[x_2,..,x_n]$.

A linear list can be called a stack, a queue, a deque, or an output-restricted deque depending on the operations allowed by the structure. The various list types are defined as follows.

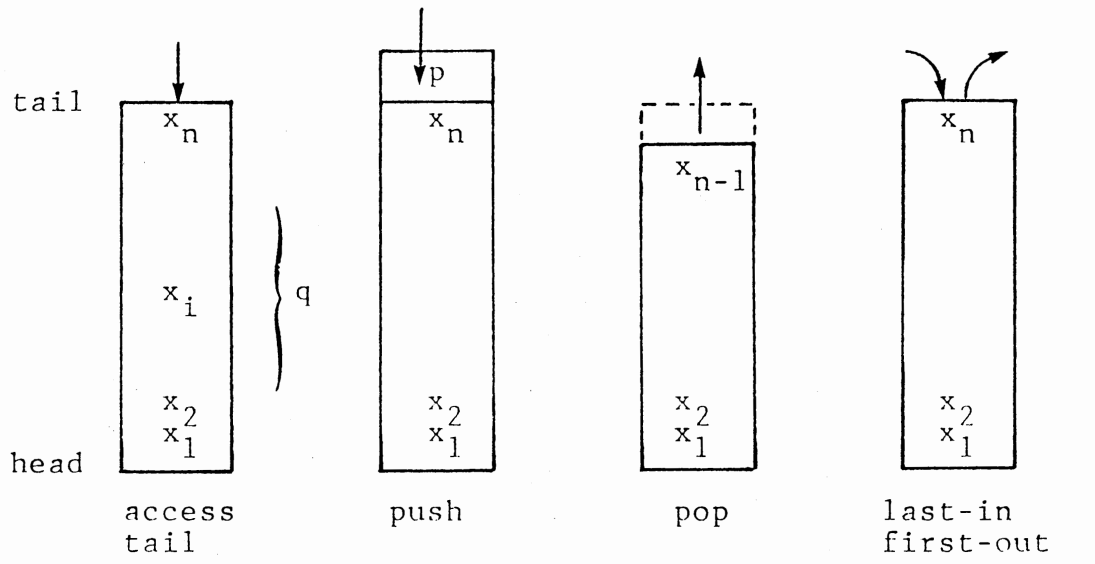
Stack: allows access tail, push, and pop (Fig.23a).

Queue: allows access tail, inject, and pop (Fig.23b).

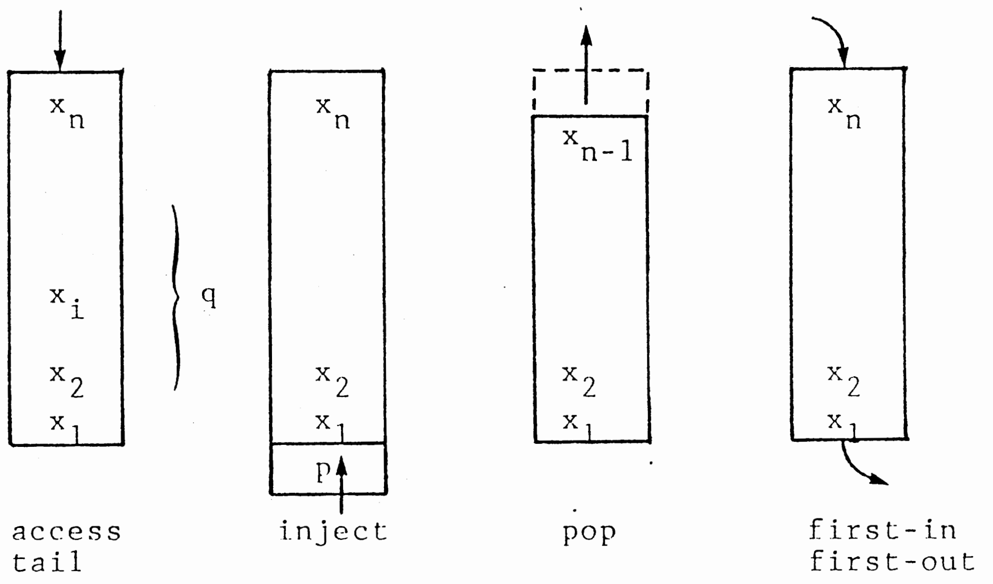
Deque: allows all six operations.

Output-restricted Deque: allows all operations except eject.

To concentrate on the amortized complexity, we analyze the stack and queue only. For insertion and deletion, the stack acts in a last-in/first-out manner; while the queue functions in a first-in/first-out manner. But these two structures allow similar updating operations, such as access tail, push(at the tail) or inject(at the head), and pop. If



(a) Stack Structure



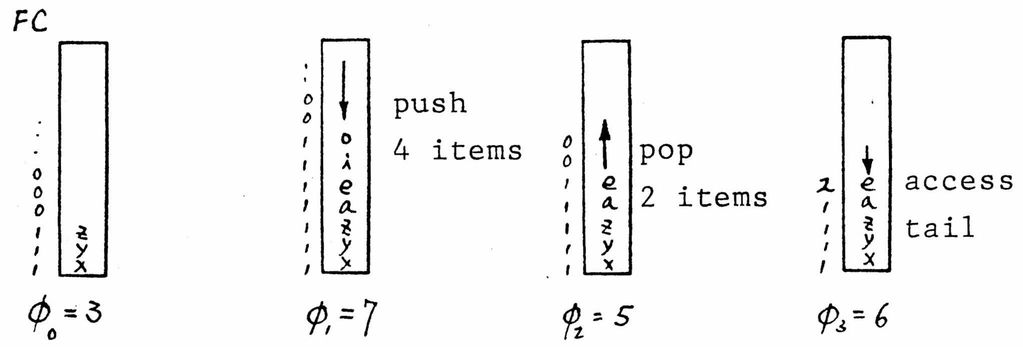
(b) Queue Structure

Figure 23. Two Linear Lists: Stack(a) and Queue(b)

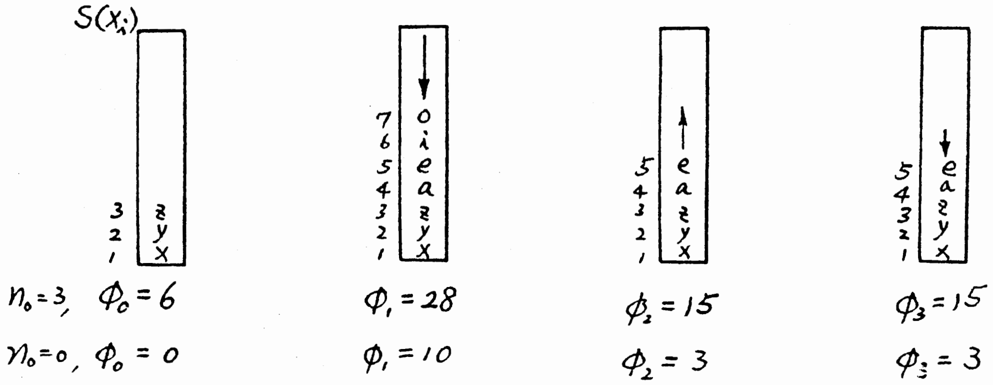
we allow swapping after updating, the list becomes a self-adjusting structure.

Amortized Complexity

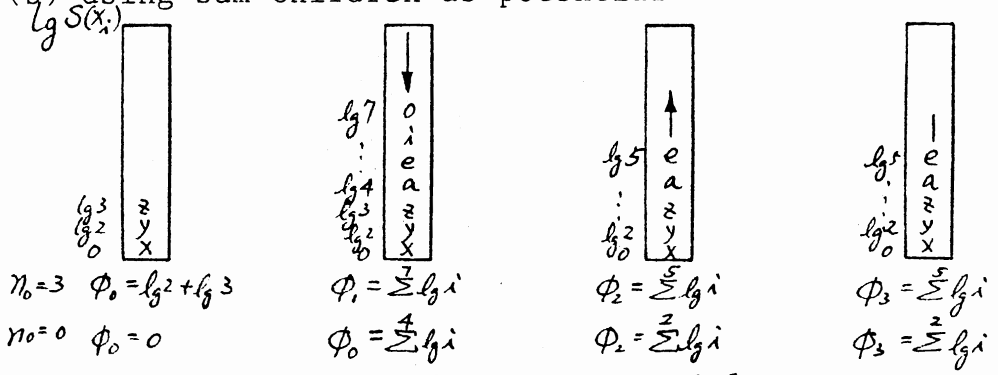
With the array representation of a stack (or queue) illustrated in Fig.23, it is clear that each operation takes $O(1)$ time, no matter what average method is applied. We are interested in the amortized costs for a sequential operations. First we use the rank potential to analyze the cost. One kind of rank is the inversion number which serves as the list potential in Tarjan's analysis. However, this rank method needs the comparison between two algorithms (such as the move-to-front and an optimal algorithm) and leads to a relative cost estimation. To simplify our analysis and to determine the absolute cost, we use the frequency count as the item rank. We maintain a frequency count (FC) for each item in the list, as shown in Fig.24a. Initially the FC is set to zero. Increase the count of an item by one whenever it is accessed or pushed (injected); reset its count to zero when it is popped. This count represents the rank of the item, and the list potential is the sum of all the item ranks. Assume a sequence of operations consists of k pushes (injects), p pops, and t operations of access tail; further, assume the sequential operations are carried out in an arbitrary time interval. Then, the potential change after the operations can be expressed by Eq.(4.1),



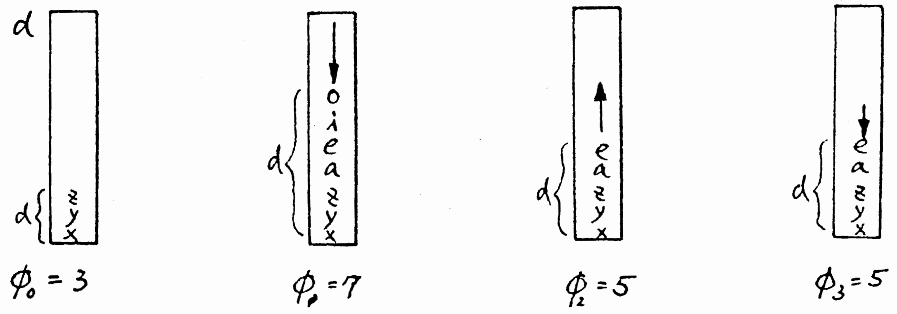
(a) using FC as item rank to represent potential



(b) using Sum Children as potential


$\lg S(x_i)$
$n_0 = 3$ $\Phi_0 = \lg 2 + \lg 3$
$n_0 = 0$ $\Phi_0 = 0$
$\Phi_1 = \sum^{7} \lg i$
$\Phi_0 = \sum^{4} \lg i$
$\Phi_2 = \sum^{5} \lg i$
$\Phi_2 = \sum^{2} \lg i$
$\Phi_3 = \sum^{5} \lg i$
$\Phi_3 = \sum^{2} \lg i$

(c) using Log Sum Children as potential



(d) using Height as potential

Figure 24. Example of Four Potential Functions in List Operations

$$\phi_R = \phi_R(D') - \phi_R(D)$$
$$= k - p + t \qquad (4.1)$$

where $\phi_R(D)$ = list potential before operations

$\phi_R(D')$ = list potential after operations

Eq.(4.1) implies that an amortized time per operation is $O(1)$ which is independent of the initial stack configuration, since

$$a_R = (\sum t_i + \Delta\phi)/(\sum t_i)$$
$$= (k+p+t+k-p+t)/(k+p+t) <= 2 \qquad (4.2)$$

An example is that we sequentially insert four items (a,e,i, and o) into a stack, then pop the last two items, and finally access the tail (Fig.24a). The stack initially contains three items: x, y, and z. The initial potential is assumed to be 3. The total potential increases 3 and the amortized cost is

$$\sum_{i=1}^{7} a_i = \sum_{i=1}^{7} t_i + \Delta\phi = 4 + 2 + 1 + 3 = 10$$

which gives an amortized cost $O(1)$ per operation.

The methods of sum children and log sum children, as described in chapter II, are more complicated here, because any item x_i's size $S(x_i)$ is dependent upon its position i in the stack (Fig.24b and c).In the case of using sum children, the potential change due to updating is

$$\phi_S = \phi_S(D') - \phi_S(D)$$

$$= \sum_{i=1}^{n_f} i - \sum_{i=1}^{n_0} i$$

$$= (n_f*(n_f+1)-n_0*(n_0+1))/2$$

$$= (k-p)*(k-p+2*n_0+1)/2 \qquad (4.3)$$

where n_0 = initial item number in stack

n_f = final item number in stack

$Z = n_f - n_0 = k-p >= 1.$

The amortized time per operation is

$$a_S = 1 + (k-p)*(k-p+2*n_0+1)/(2*(k+p+t))$$

$$= 1 + (Z+2*n_0+1)/(2*(1+2*p/Z+t/Z)) \qquad (4.4)$$

$$= f_1(\ n_0,\ Z\ )$$

Eq.(4.4) shows that the amortized cost is not a constant but a function of initial configuration n_0 and actual operations Z. Eq.(4.4) is calculated based on different Z (1 to 1000) and p (10 to 200) values, with n_0=0. Results are shown in Fig.25. The a_S increases rapidly with Z, the difference between push and pop frequencies. The example illustrated in Fig.24b shows that the potential increment is (15-6)=9 when n_0=3. But it will be only 3 if n_0=0.

In the case of using log sum as potential function, the potential change is

$$\begin{aligned}\phi_L &= \phi_L(D') - \phi_L(D) \\ &= \sum_{i=1}^{n_f} \lg i - \sum_{i=1}^{n_0} \lg i \\ &= \lg(\, n_f! \,/\, n_0! \,) \end{aligned} \tag{4.5}$$

Thus, the amortized time a is determined by Eq.(4.6),

$$\begin{aligned} a_L &= 1 + \lg((n_0+Z)!/n_0!)/(k+p+t) \\ &= f_2(\, n_0,\, Z \,) \end{aligned} \tag{4.6}$$

Eq.(4.5) is also calculated for Z values up to 1000 with n_0=0. The computational results, plotted on Fig.25, show that this method provides a much better estimation than that by sum children method. However, the cost is still a function of parameters n_0, Z, and p. When either large numbers of operations occur, or the initial stack size is large, the amortized estimate is pessimistic.

Now we consider the potential function expressed in terms of height. We use the number of items contained by the list to represent the height (potential) (Fig.24d). So the potential change after k pushes, p pops, and t access tails, is

$$\begin{aligned}\phi_H &= \phi_H(D') - \phi_H(D) \\ &= k - p \end{aligned} \tag{4.7}$$

and the amortized time per operation is

$$a_H = 1 + (k-p)/(k+p+t) <= 2 \tag{4.8}$$

which gives an $O(1)$ complexity.

Results from the above four potential analyses are listed in Table III for comparison. It is seen that both the

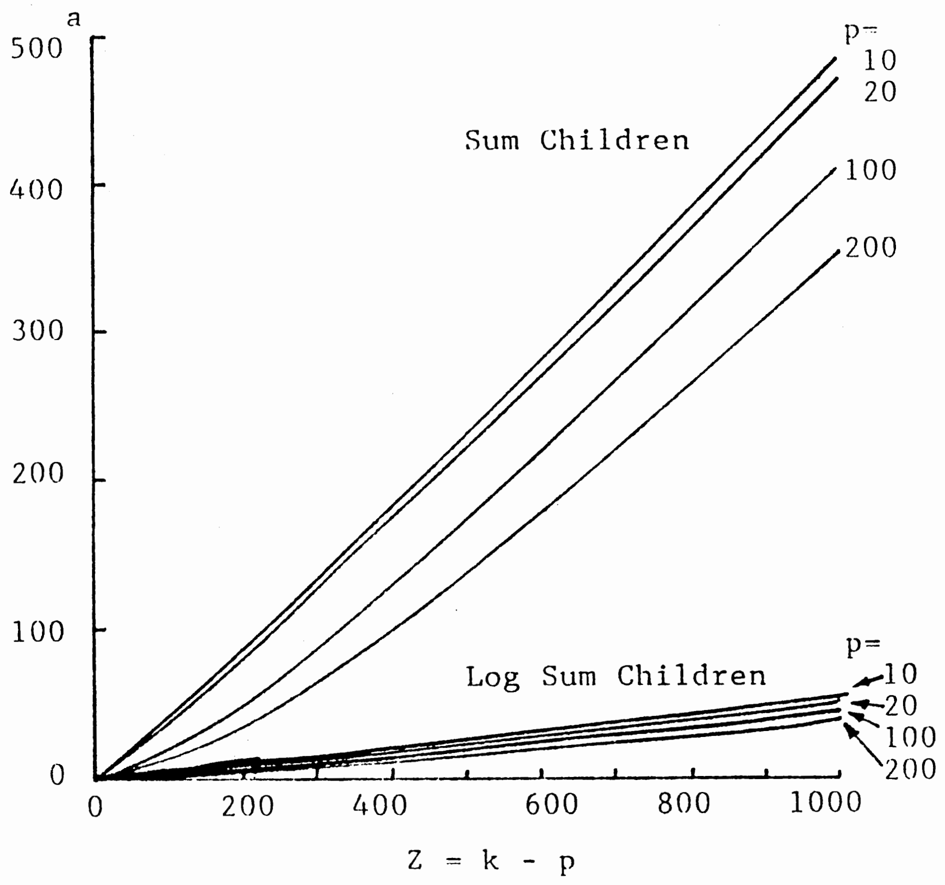


Figure 25. Amortized Time vs. List Operations when using Sum Children and Log Sum Methods

sum children and the log sum potentials are not suitable for list structures; while the other two methods, the rank (using frequency count) and the height (using item number), can provide $O(1)$ complexity estimations for sequential operations in stacks or queues.

We can use the standard data set analyzed in chapter II to further demonstrate these results. That is, we sequentially insert a set of items, a, e, i, o, g, l, n, r, t, and x, into an initially empty stack. In this case, k equals 10. The amortized time per insertion, therefore, is 7 by using sum children as potential (Eq.4.4); while the other three potentials give the same cost, 2, under this condition. The example results are also listed in Table III to compare with the predicted costs.

Finally, we use the rank potential to analyze the move-to-front algorithm for the example in Fig.24. After each push (inject) or access tail, the frequency count increases one; the item just accessed or pushed is then moved to the head, this transaction also adds one to the frequency count (Fig.26). Thus,

$$C_{MF} = \sum a_i = \sum t_i + \Delta\Phi = 7 + 7 = 14$$
$$<= 2*C_A = 20$$

where $C_A = 10$, the total cost of algorithm without move-to-front.

This result agrees with Tarjan's analysis which is mentioned in chapter II.

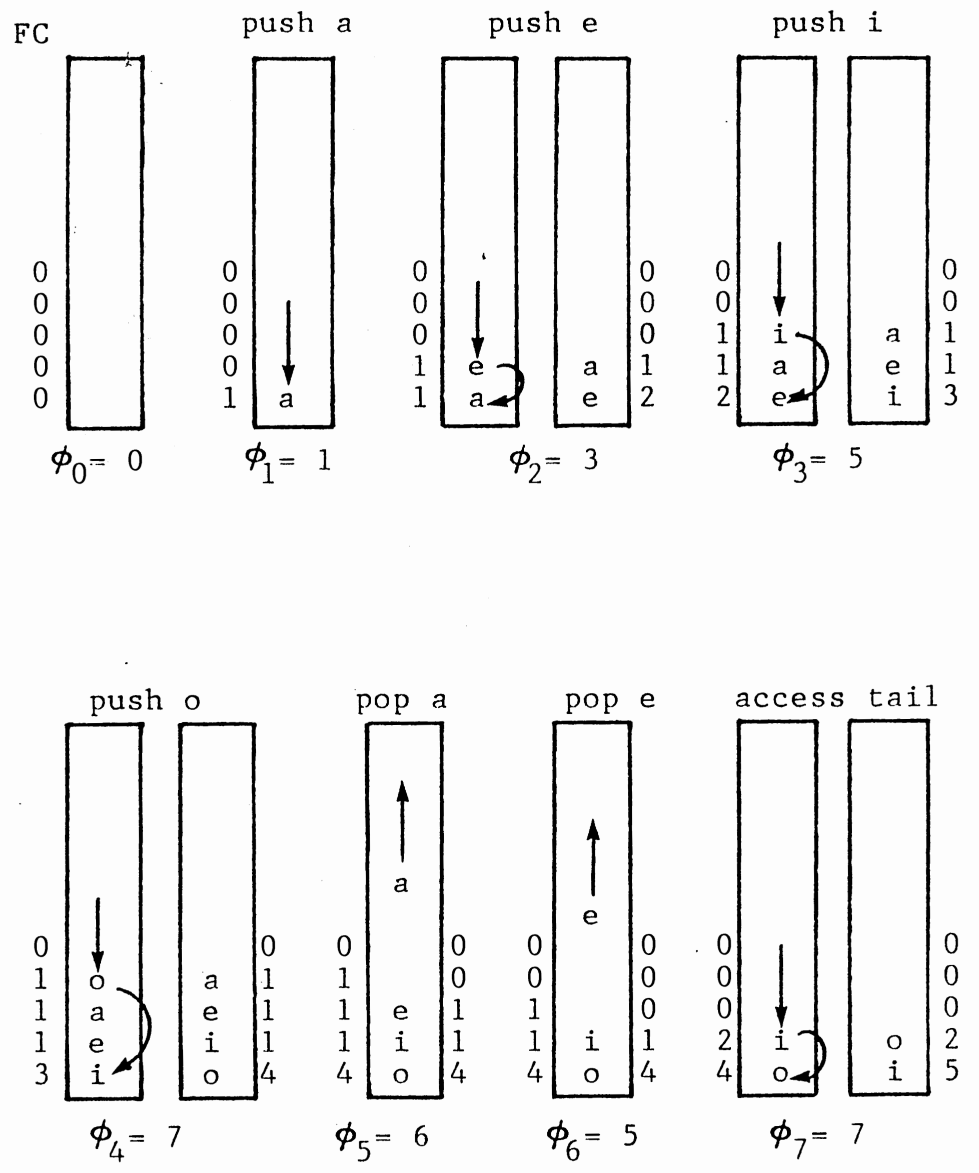


Figure 26. Using Rank(FC) Potential to Analyze the Move-to-Front Algorithm

TABLE III

AMORTIZED COMPLEXITY OF LIST OPERATIONS

Potential Functions	Potential Change	Amortized Time per Operation	EXAMPLE Predicted	Actual
Φ_R	$k - p + t$	$O(1)$	2.0	1.3
Φ_S	$\frac{(k-p)(k-p+2n_0+1)}{2}$	$f_1(n_0,k-p)$	11	7
Φ_L	$\lg((n_0+k-p)!/n_0!)$	$f_2(n_0,k-p)$	2	2
Φ_H	$k - p$	$O(1)$	2	2

CHAPTER V

BALANCED SEARCH TREES

A search tree containing one item per internal node is usually used to represent a sorted data set. Basic operations for maintaining a sorted set are access, insertion, and deletion. In this chapter, we analyze the amortized complexity of three typical different balanced search trees -- the HB[1]-tree, red-black tree, and B-tree under a sequence of insertions. The HB[1]-tree is sometimes called as an AVL-tree, and the result is easy to extend to generalized HB[k]-trees. The red-black tree is also a binary structure, but is balanced by color constraints. The B-tree is a very effective structure for file organization. It is not a binary tree but has variable node degrees (each node can have more than one key). For each tree structure, we examine the amortized cost of rebalancing in bottom-up insertions by using the four potential functions. This analysis method can be similarly applied to top-down algorithms and other operations.

HB[1]-Trees

Operations

The structure of an HB[1]-tree (AVL-tree) was well described by Knuth [45]. We use the same definition as that stated in chapter II for AVL-trees, and use the updating algorithm described by Chen [31]. An HB[1]-tree satisfies:

1. $hb(v) = |Height\ (R(v)) - Height\ (L(v))| <= 1$ where v is any internal node);
2. right and left subtrees R(v) and L(v), respectively, are also HB[1]-trees.

To insert a new item into an HB[1]-tree, first we carry out a binary search to find out the proper place for it, and attach a new leaf node in which the new item is stored (Fig.27a). The hb of the new node is always set to zero. Then we perform necessary rebalancing steps if there is any violation of the height balance constraints. The bottom-up algorithm of rebalancing an HB[1]-tree ($k = 1$) after an insertion is illustrated in Fig.27 and explained as follows.

1. Examine whether the current node v is a critical node (a critical node satisfies $|hb(v)| = k + 1$). If it is true, go to step (3); otherwise, go to step (2).
2. Examine whether v's parent P(v) satisfies $k >= hb\ (P(v)) > 0$ and v is the left child of P(v) (or symmetric variant). If the test is true, update

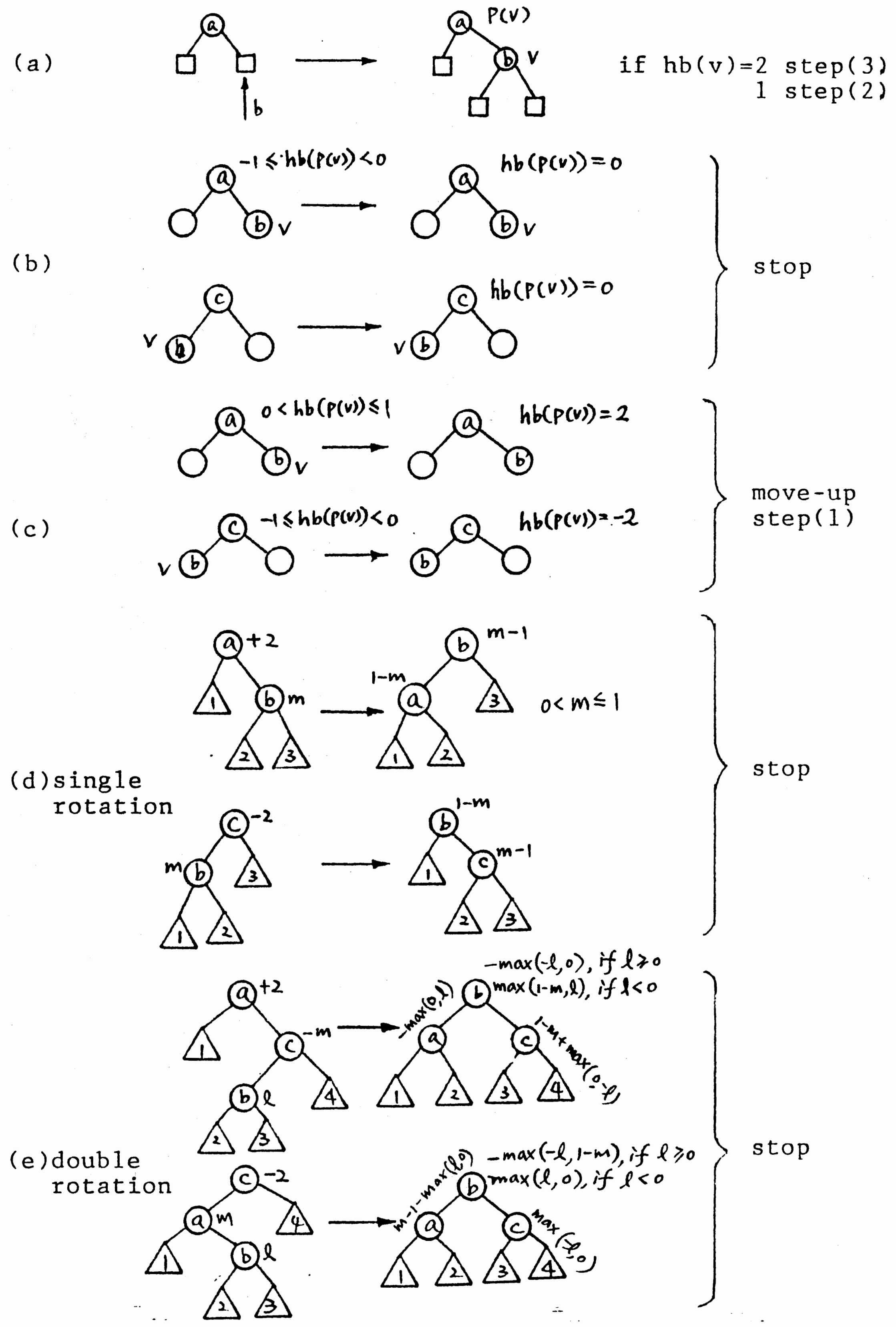


Figure 27. Bottom-Up Insertion Algorithm for AVL-Trees

hb(P(v)) and stop (Fig.27b); otherwise, update hb(P(v)), let P(v) be current node v and go back to step (1). (Fig. 27c).

3. Examine whether $hb(v) = k + 1$ and $k >= hb(R(v)) > 0$ (or symmetric variant). If the test is true, perform a single rotation, update hb(v) and stop (Fig. 27d); otherwise, perform a double rotation, update hb(v), and stop (Fig. 27e).

The above procedures are for rebalancing general HB[k]-trees and apply to AVL-trees with $k = 1$.

<u>Amortized Complexity</u>

As we did for the list structure, we analyze the amortized cost of a sequence of insertions based on four potential functions which reflect the structure configuration.

First, we need to estimate two variables in a binary tree of n leaves (or internal nodes): m which is the maximum number of nodes at each level of the tree and h which is the maximum number of levels in the tree, or say height. These maximums provides an upper bound in our amortized analysis. Here we assume a positive integer j can be determined such that $2^{j-1} <= n < 2^j$. Consider a binary tree of n arbitrary leaves (Fig.28). From the definition of binary trees, it is known that the n leaves at base level can construct at most $\lceil n / 2^1 \rceil$ internal nodes at the first level, and these first level nodes can construct at most $\lceil n / 2^2 \rceil$ nodes at

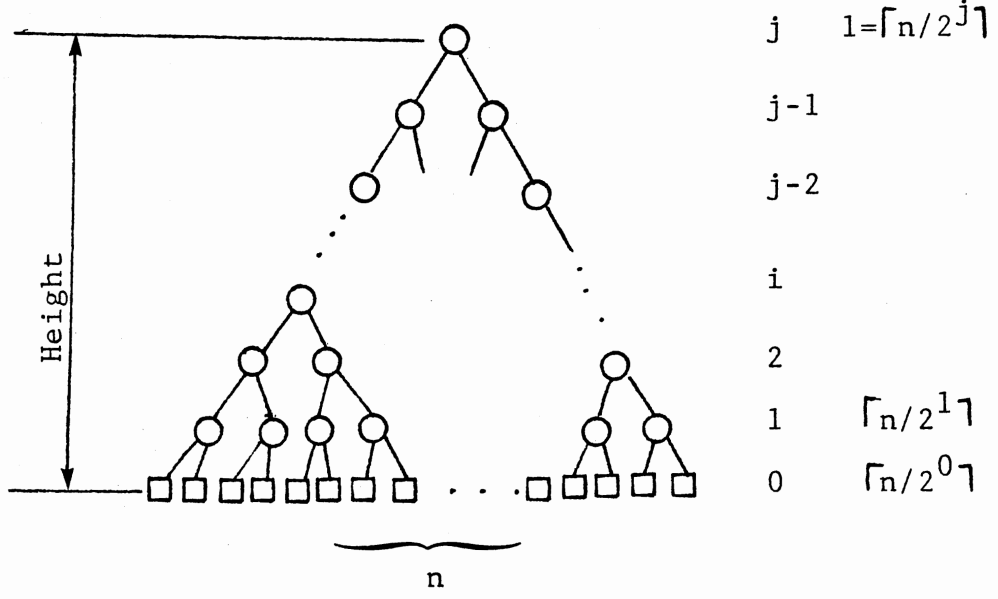


Figure 28. Height Estimation of an AVL-Tree

second level. While at the heighest level, only one root node exists, that is $m = 1 = \lceil n / 2^j \rceil$. Therefore, the maximum node number at any level i ($0 <= i <= j$) can be written as

$$m_i <= \lceil n / 2^i \rceil \qquad (5.1)$$

Since the level i increases from 0 to j, it implies that the tree height is bounded by

$$h = j + 1 \qquad (5.2)$$

where j satisfies that

$$j - 1 <= \log_2 n < j \qquad (5.3)$$

For the rank potential function, we define the level of each node in an HB[1]-tree as its rank. That is, the rank of the root is j and the rank of any node at level i is i. Thus, the rank of a tree with n items is the sum of all n node ranks and can be calculated as follows.

$$\begin{aligned} \phi_R &= \sum_{i=1}^{j} \lceil n / 2^i \rceil * i \\ &<= n * j * \sum_{i=1}^{j} (1 / 2^i) = n * j * (1 - 1/2^j) \\ &<= n * j \end{aligned} \qquad (5.4)$$

Then the amortized cost per insertion is bounded by Eq.(5.5)

$$a_R = (\sum t_i + \Delta\phi_R) / n$$
$$<= 1 + j$$
$$= O(\log_2 n) \qquad (5.5)$$

We examine this result by inserting the sample data set [a, e, i, o, g, l, n, r, t, x]. The process of step-by-step insertion and rebalancing is illustrated in Fig.29. The tree potential function is also listed. As a result, the total potential increase is 37, and the amortized cost is

$$\sum_{i=1}^{10} a_i = \sum_{i=1}^{10} t_i + \Delta\phi_r = 10 + 21 = 31$$

which gives an per insertion cost of $O(\log_2 n)$.

To apply the potential functions of sum children and log sum children, we also need to estimate the maximum size of a node. It is easy to determine in a binary tree, as the node at level i has at most two children nodes at level (i-1). Since the size of node x is the sum of its children including itself, then for i = 0, 1, 2, ..., j,

$$S_i <= 2 * S_{i-1} + 1 = 2 * (2^i - 1) + 1$$
$$= 2^{i+1} - 1 \qquad (5.6)$$

Combining the upper bounds of m, h, and S_i, we have the potential function estimation in terms of sum children for n sequential insertions into an HB[1]-tree,

$$\phi_S <= \sum_{i=0}^{j} \lceil n / 2^i \rceil * (2^{i+1} - 1) \qquad (5.7)$$

where j is determined by (5.3).

The amortized cost per insertion thus is

$$a_S <= 1 + (\sum_{i=0}^{j} \lceil n / 2^i \rceil * (2^{i+1} - 1)) / n$$

$$= 1 + 2 * j <= 1 + 2 * (\log_2 n + 1)$$

$$= O(\log_2 n) \qquad (5.8)$$

The case of using log sum as potential function can be similarly analyzed. The only difference here is the maximum size of each node at level i is given by

$$\lg S_i <= \lg (2^{i+1} - 1) \qquad (5.9)$$

Thus,the tree-potential increment after n insertions is

$$\Delta \phi_L = \phi_L <= \sum_{i=0}^{j} \lceil n / 2^i \rceil * \lg (2^{i+1} - 1) \qquad (5.10)$$

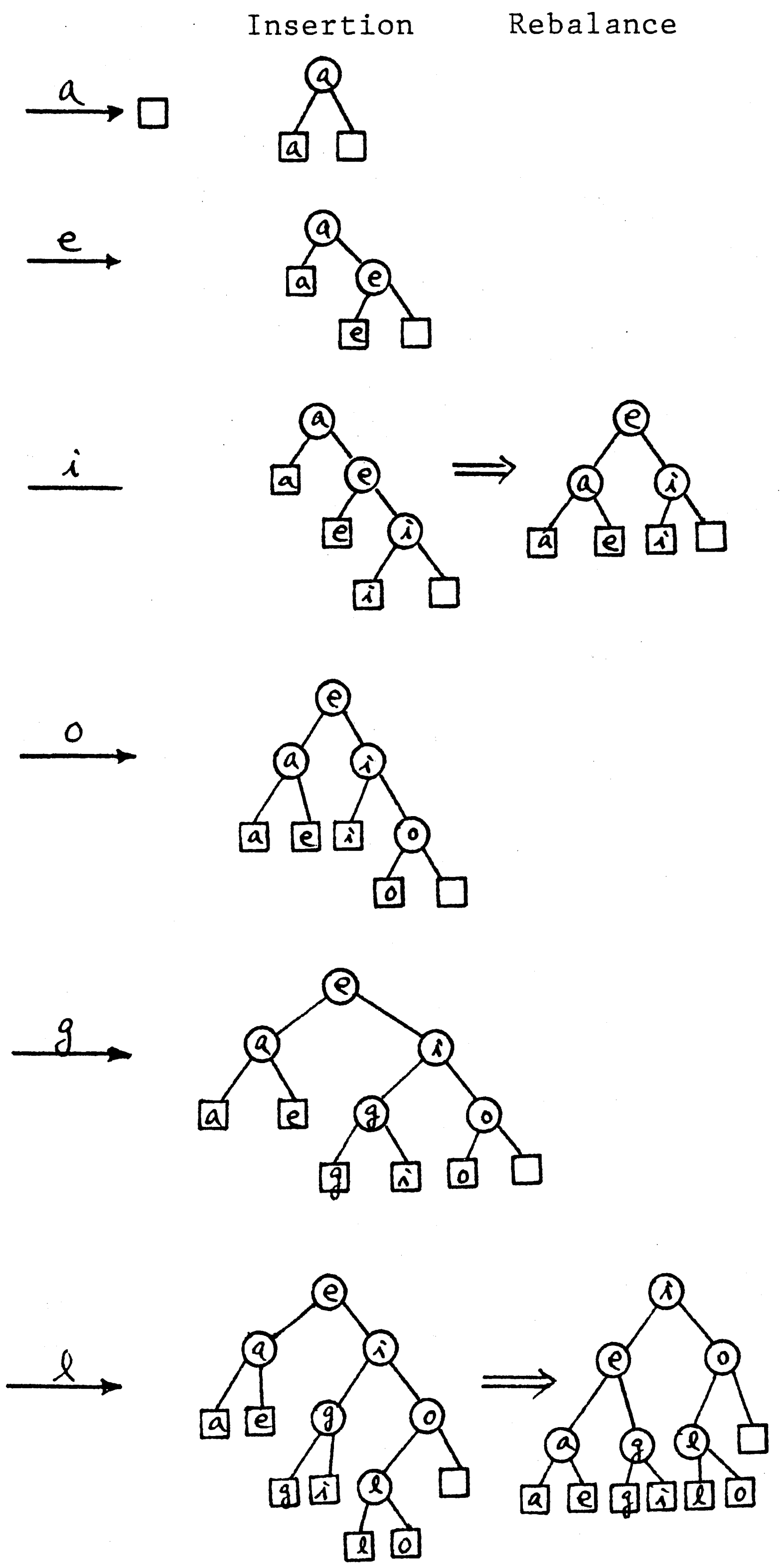
which gives an amortized cost per insertion as

$$a_L <= 1 + \sum_{i=0}^{j} (\lg(2^{i+1} - 1) / 2^i)$$

$$<= 1 + \sum_{i=0}^{j} (\lg 2^{i+1} / 2^i)$$

$$= O(1) \qquad (5.11)$$

The last case of potential function is to use the height. From the upper bound of h (Eqs.(5.2) and (5.3)), we directly obtain



Φ_R	Φ_S	Φ_L	Φ_H
1	3	0.3	2
3	8	0.9	3
4	14	1.55	3
8	19	1.98	4
9	29	3.03	4
10	38	4.05	4

Figure 29. Example of Sequential Insertions in AVL-Tree

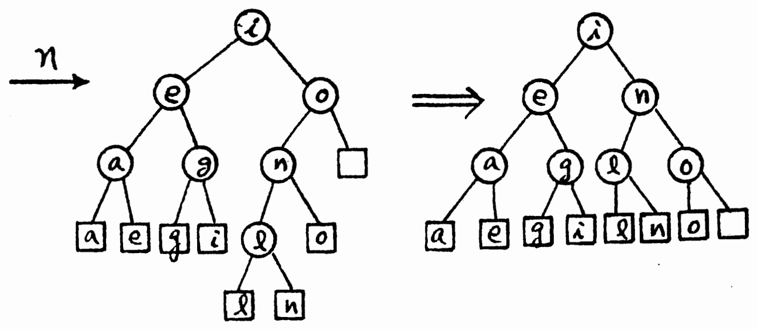
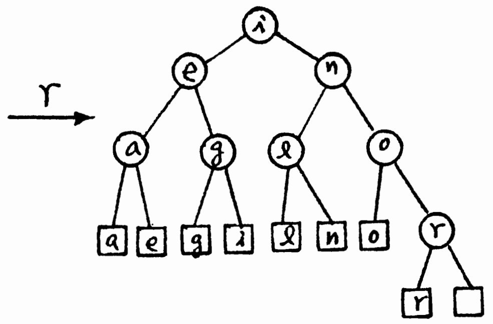
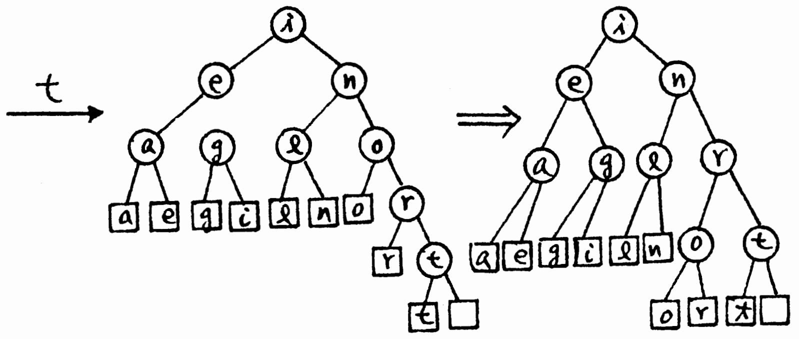
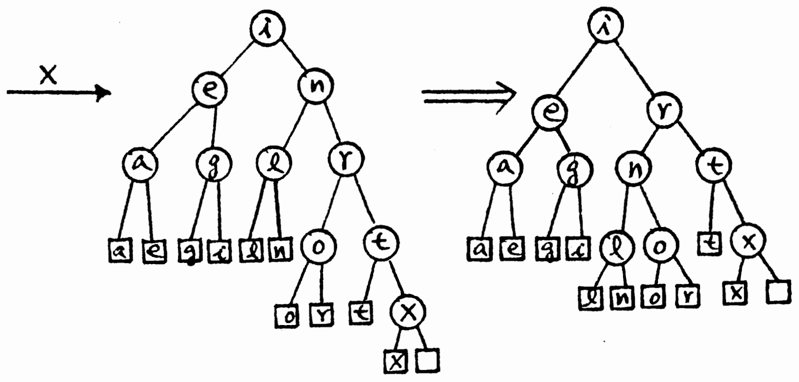
	Φ_R	Φ_S	Φ_L	Φ_H
	11	45	4.5	4
	19	54	5.28	5
	20	64	6.08	5
	21	74	6.88	5

Figure 29. Continued

TABLE IV

AMORTIZED COMPLEXITY OF SEQUENTIAL INSERTIONS IN HB[1]-TREES

Potential Functions	Potential Change	Amortized Time per Operation	EXAMPLE Predicted	Actual
Φ_R	$\sum_{i=1}^{j} \lceil n/2^i \rceil * i$	$O(1)$	4.32	3.1
Φ_S	$\sum_{i=1}^{j} \lceil n/2^i \rceil * (2^{i+1}-1)$	$O(\log_2 n)$	8.64	8.1
Φ_L	$\sum_{i=1}^{j} \lceil n/2^i \rceil * \lg(2^{i+1}-1)$	$O(1)$	2	1.65
Φ_H	$\lceil \log_2 n \rceil + 2$	$O(1)$	2	1.5

$$\Phi_H = h <= j + 1$$
$$<= \lceil \log_2 n \rceil + 2 \qquad (5.12)$$

Therefore, the per operation cost is

$$a_H <= 1 + (\lceil \log_2 n \rceil + 2) / n <= 3$$
$$= O(1) \qquad (5.13)$$

The above analysis is summarized and listed in Table IV. We use the standard data set to illustrate these complexity estimations for AVL-trees. The variation in each of the four potential functions is indicated in Fig.29 for every updating step. The amortized time per insertion is 8.1, 1.68 and 1.5 when using sum children, log sum, and height potential, respectively, as compared with the predicted values in Table IV.

Red-Black Trees

Operations

A red-black tree is a binary search tree which is balanced according to three color constraints defined in chapter II. The procedure of bottom-up insertion in red-black trees has been well described by Fisher [33] and Chen [31]. Here we restate it as follows.

The first step in insertion into a red-black tree is to search at the leaf level and find a proper position for the new item to be inserted as we did in AVL-trees. Then we replace the external node by a subtree consisting of one internal node and two external children, one containing the new item and the other containing the item in the original external node. The key of the new internal node is the smaller one of its two children. This insertion step is represented by case (a) in Fig.30.

After the new item is inserted, the tree may need to be rebalanced to satisfy the color balance condition. The new internal node v is always colored red. This preserves the black constraint but may violate red constraint. So if v's parent P(v) is black, recoloring stops. Otherwise, perform the following two color rebalancing steps according to four possible cases (b), (c), (d), or (e).

1. if the red node v has a red parent P(v), whose sibling P'(v) is also red, then recolor both P(v) and P'(v) black, also recolor v's grandparent P(P(v)) red (case (b) in Fig.30). However, if P(P(P(v))) is still red, then let P(P(v)) be node v and repeat this step until no red violation occurs or conditions in step 2 are met.
2. if the red node v has a red parent P(v) which is just the root node,color P(v) black and stop (case(c) in Fig.30); otherwise, it means that red node v has a red P(v) and a black P'(v). If v is the left child of P(v) and P(v) has a right black P'(v) (or symmetric variant), perform a single right rotation and stop (case (d) in Fig.30). If v is the right child of P(v) and P(v) has a right black P'(v) (or symmetric variant), perform a double right rotation and stop (case (e) in Fig.30).

Amortized Complexity

It is seen from the above procedures that the bottom-up insertion terminates in case (c), (d), and (e). Only case(b) may be not terminating and recoloring can at most occur along a path from a node at base level until the root node, or can terminate in case (d) or (e).

Tarjan [36] once used a special case of the potential for amortized analysis of red-black trees. In his method, each black internal node is assigned a potential of one, zero, or two if the node has no, one, or two red children, respectively. Then the potential of a tree is the sum of the potentials of its black nodes. Here we refine Tarjan's result by calling such black node potential as the rank of the black node. With this potential definition, we know that the rank of a subtree can increase at most two in either case (d) or case (e); decrease one every time for case (b); and remain unchanged for case (c). These changes of subtree rank are indicated beside the black nodes in Fig.30.

From the operation procedures described above, we analyze the rank potential change in each possible recoloring process as follows.

1. Case (b) repeated m times and stop at case (c). This leads to a decrease in tree rank potential. That is

$$\Phi(D') - \Phi(D) = m * (-1) + 0 \qquad (5.14)$$

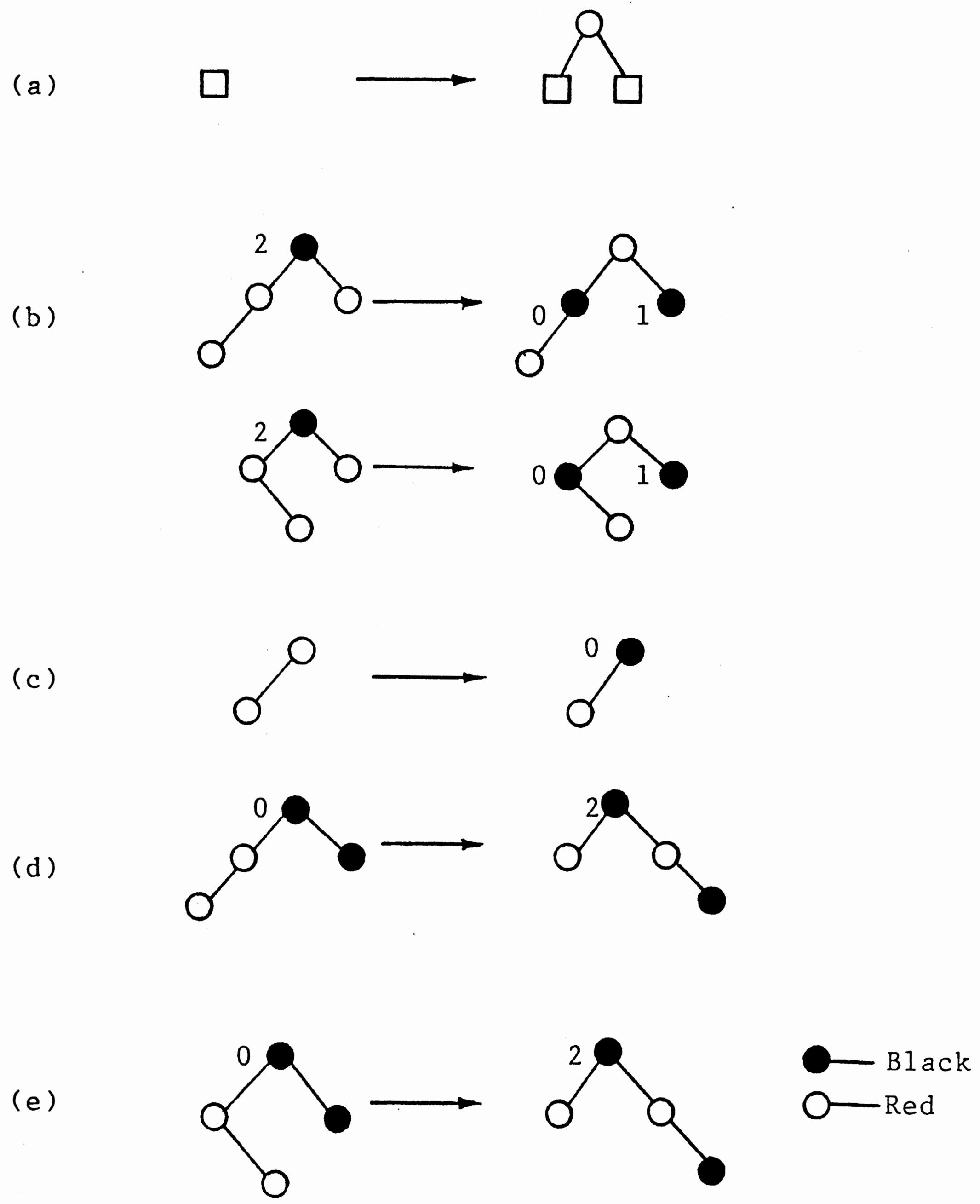


Figure 30. Bottom-Up Insertion Algorithm for Red-Black Trees

2. Case (b) repeated m times and stop at either case (d) or case (e). The rank increment is at most one when m = 1. That is

$$\phi(D') - \phi(D) = (-1) + 2 = 1 \tag{5.15}$$

Since in n sequential insertions there are at most n such recoloring cases, the total rank increment is bounded by n,

$$\phi_R <= n \tag{5.16}$$

Thus, the amortized cost per operation is given by

$$\begin{aligned} a_R &= (\sum t_i + \Delta\phi_R) / n \quad <= 2 \\ &= O(1) \end{aligned} \tag{5.17}$$

The other three classes of potential function are all height-related. The maximum node number and maximum node size at each level are the same as that in AVL-trees. But the height difference between two subtrees of any node in a red-black tree is at most 2. We consider the worst case, all insertions occur at a regular position, say the leftmost of a tree as shown in Fig.31. The recoloring process terminates in case (c), (b), or (d). Because of the red constraint, there is at most one insertion which is rebalanced without applying case (d) and results in a subtree height difference of two, such as the 6th insertion in Fig.31. One more insertion at this position certainly leads to case (d)

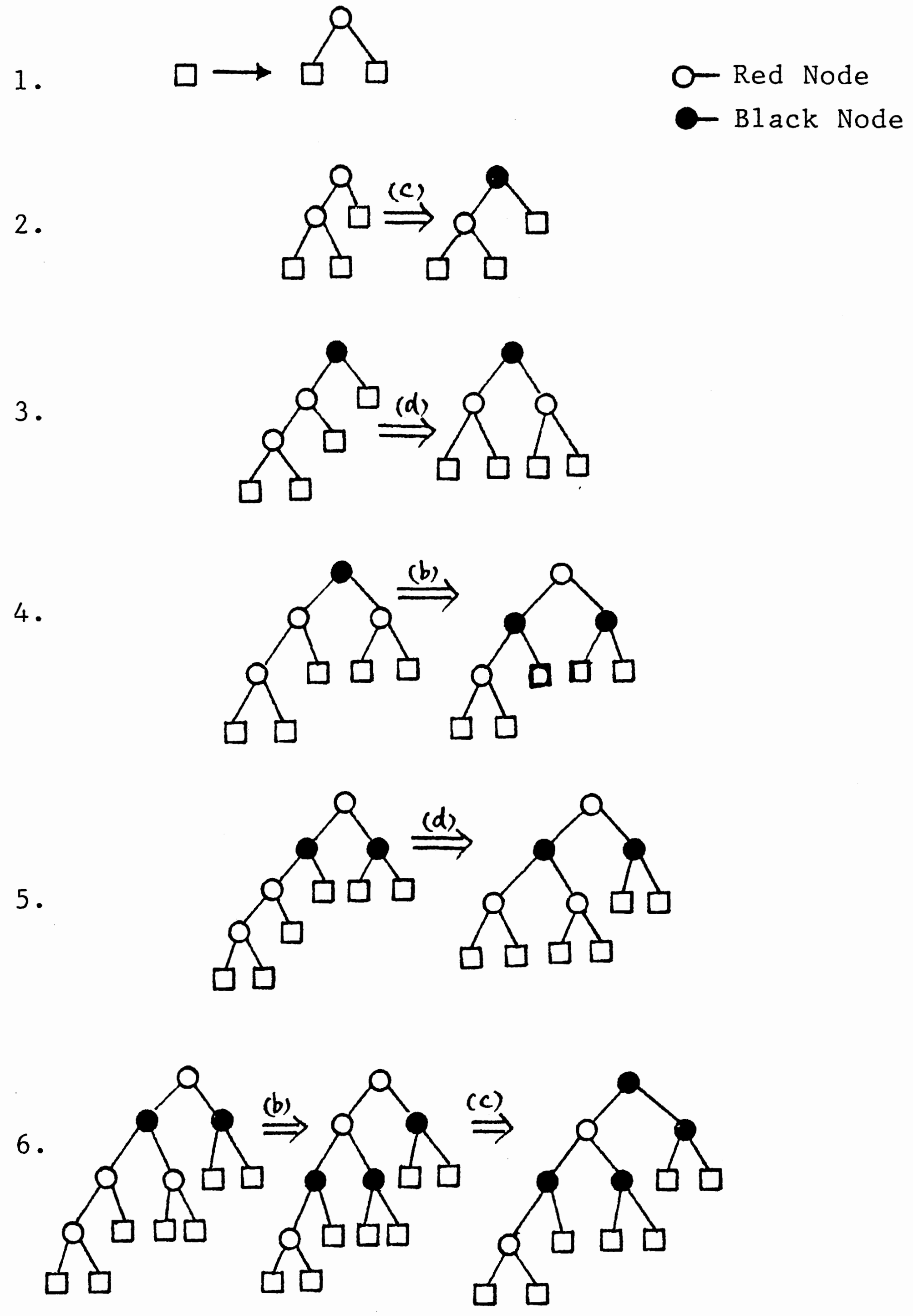


Figure 31. The Leftmost Insertion into an Red-Black Tree

and each application of a single rotation in case (d) reduces the height of this subtree by one. Thus, the maximum height difference is two, and the red-black tree height is bounded by

$$h <= j + 2 \qquad (5.18)$$

where j is defined by inequality (5.5)

Combining the h (maximum number of levels) with the upper bounds of node number and node size at each level, the tree potential in terms of sum children can be obtained as

$$\Phi_s <= \sum_{i=0}^{j+1} \lceil n / 2^i \rceil * (2^{i+1} - 1) \qquad (5.19)$$

and the amortized cast per insertion is

$$\begin{aligned} a_s &<= 1 + (\sum_{i=0}^{j+1} \lceil n / 2^i \rceil * (2^{i+1} - 1)) / n \\ &<= 1 + 2 * (j + 1) \\ &<= 1 + 2 * (\log_2 n + 2) \\ &= O(\log_2 n) \qquad (5.20) \end{aligned}$$

In the case of using log sum children, the tree potential increment after n insertions is bounded by

$$\Phi_L <= \sum_{i=0}^{j+1} \lceil n / 2^i \rceil * \lg (2^{i+1} - 1) \qquad (5.21)$$

and the amortized cost per insertion is

$$a_L <= 1 + \sum_{i=0}^{j+1} (\lg(2^{i+1} - 1) / 2^i)$$
$$<= 1 + \sum_{i=0}^{j+1} (\lg 2^{i+1} / 2^i)$$
$$= O(1) \qquad (5.22)$$

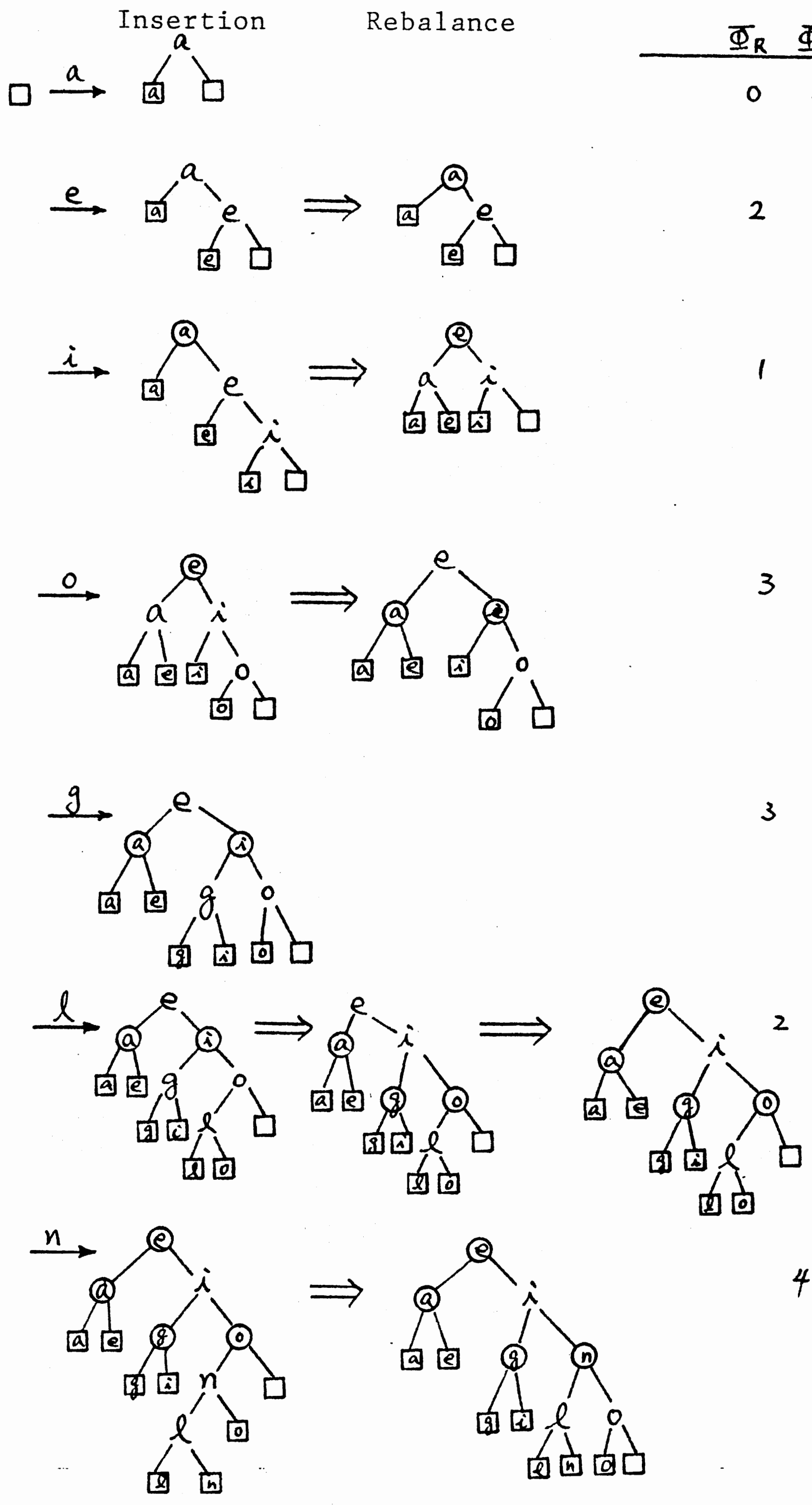
Based on the analysis of maximum height of a red-black tree built up by n sequential insertions, we have

$$\Phi_H = h <= j + 2$$
$$<= \lceil \log_2 n \rceil + 3 \qquad (5.23)$$

and the amortized cost per insertion is

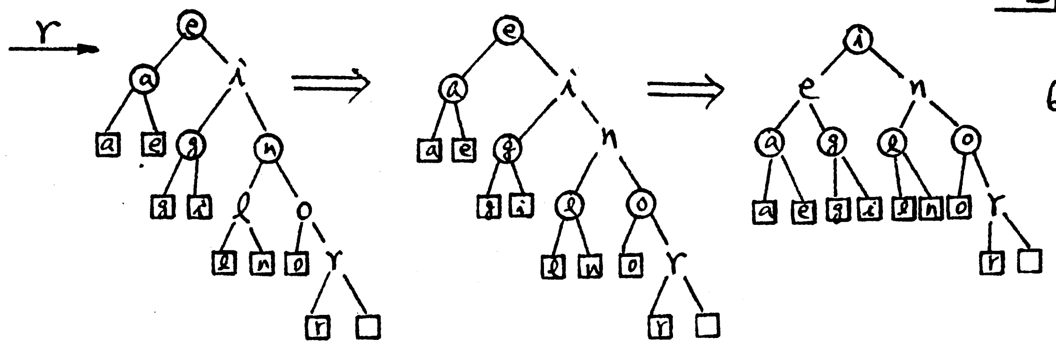
$$a_H <= 1 + (\lceil \log_2 n \rceil + 3) / n <= 4$$
$$= O(1) \qquad (5.24)$$

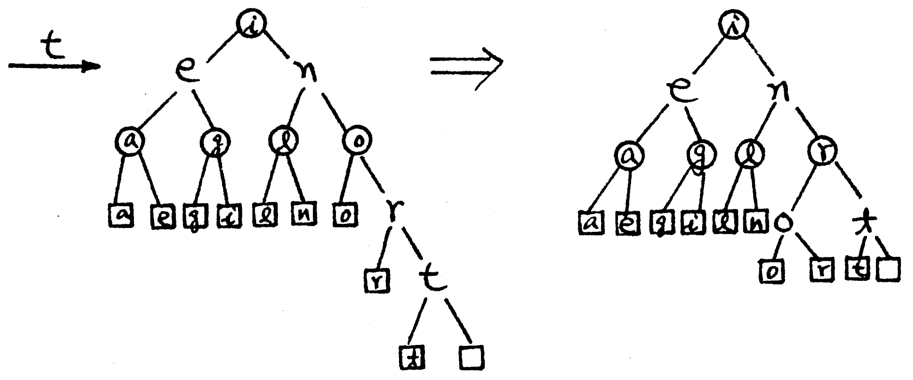
The amortized complexity of n sequential insertions into a red-black tree is summarized in Table V for later comparison. The insertion of our example data set is demonstrated in Fig.32, with the four potential function values indicated at each step. The total potential change and amortized time per insertion are also calculated. The a is 1.7, 8.2, 1.79, and 1.6 in the four cases, respectively, below that predicted values shown in Table V.

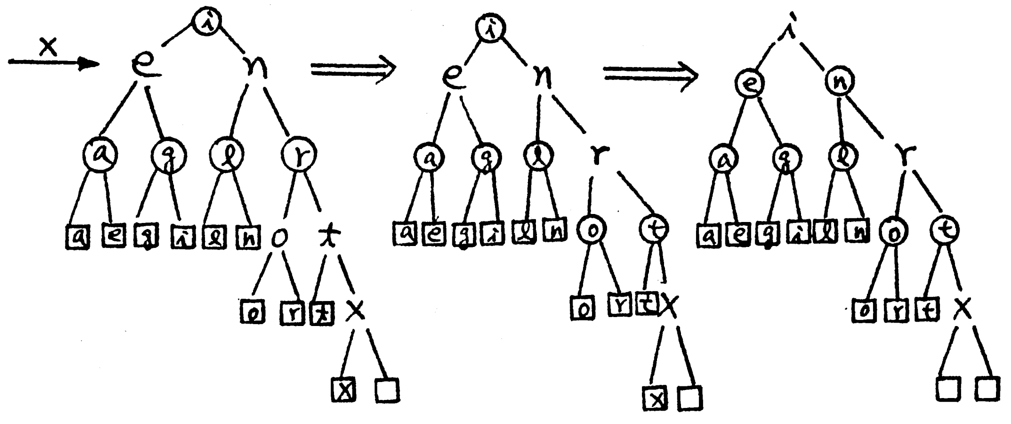


Φ_R	Φ_S	Φ_L	Φ_H
0	3	0.3	2
2	8	0.9	3
1	14	1.55	3
3	19	1.98	4
3	29	3.03	4
2	38	4.05	5
4	48	4.65	5

Figure 32. Example of Sequential Insertions into a Red-Black Tree







Φ_R	Φ_S	Φ_L	Φ_H
6	54	5.28	5
7	64	6.08	5
7	75	7.94	6

Figure 32. Continued

TABLE V

AMORTIZED COMPLEXITY OF SEQUENTIAL INSERTIONS IN RED-BLACK TREES

Potential Functions	Potential Change	Amortized Time per Operation	EXAMPLE Predicted	Actual
Φ_R	n	$O(1)$	2	1.7
Φ_s	$\sum_{0}^{j+1} \lceil n/2^i \rceil * (2^{i+1}-1)$	$O(\log_2 n)$	11	8.2
Φ_L	$\sum_{0}^{j+1} \lceil n/2^i \rceil * \lg(2^{i+1}-1)$	$O(1)$	2	1.8
Φ_H	$\lceil \log_2 n \rceil + 3$	$O(1)$	4	1.6

B - Trees

Operations

The B-tree data structure proposed by Bayer and McCreight [46] is an effective method for organizing an external file. Comer [16] indicated that B-trees have become a standard for file organization. However, variations of B-trees abound, and the literature on B-tree is not uniform in its use of terms relating to B-trees. Since the B-tree is a multiway search tree instead of binary tree as described previously, here we need to restate the definitions of order and leaf (external node) of a B-tree, which were addressed in [45] and [46], respectively. Then we can describe the properties of a B-tree formally defined by Folk [14].

Definition : the order of a B-tree is the maximum number of children that a node can have; the leaf (external node) in a B-tree is a node at the lowest level of keys. Then a B-tree of order m satisfied the following six properties.

1. Each internal node has at most m children.
2. Each internal node except the leaves and possibly the root has at least $\lceil m / 2 \rceil$ children.
3. The root node has at least two children (unless it is a leaf).
4. All the leaves appear on the same (the lowest) level.
5. A nonleaf node (internal node) with k children contains

(k - 1) keys.

6. Each leaf contains at least ($\lceil m / 2 \rceil$ - 1) keys but no more than (m - 1) keys.

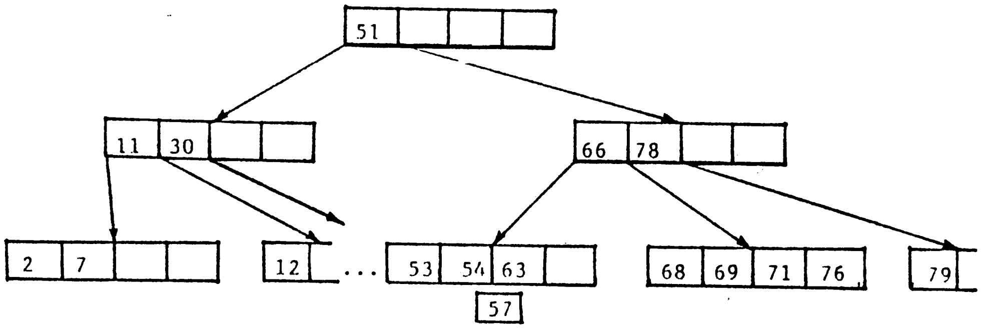
In a B-tree, a node consists of an ordered sequence of keys and a set of pointers. A bottom-up insertion of a new key requires a two-step process. First, a search proceeds from the root to locate the proper leaf position for insertion. Then the insertion is performed, and balance is restored by a procedure which moves from the leaf back toward the root.

An example of inserting a new key is illustrated in Fig.33, where each node in a B-tree of order 5 contains between ($\lceil 5/2 \rceil$ - 1) and (5 - 1) keys. From Fig.33, it is seen that when inserting the key " 57 " the search terminates unsuccessfully at the fourth leaf. Since the leaf can accommodate another key, the new key "57" is simply inserted, yielding the B-tree shown in Fig.33b. If a key "72" were inserted, however, complications would arise because the appropriate leaf node is full. Overfull nodes can be restructured by equalization, either one way or two way, followed by node splitting if the equlization is unsuccessful. Another approach is to split the overfull node directly. The left (or right) adjacent leaf node (one way equalization) is checked, if the node has some spaces, then the key from parent moves into the "unfull" sibling node and keys from the overfull node are distributed among the two nodes with the appropriate key placed into the parent node as shown in Fig.33c and 33d. The equalization procedure delays spliting until the sibling node(s) is (are) full. This strategy increases space utilization and minimizes the number of levels in the tree. Node splitting as means of restructuring overfull nodes splits the right(left) half off as new node, and sends the median key upward one level for parent node insertion, which serves as a separator presented in Fig.34. Usually the parent node will accommodate an additional key, thus the insertion process terminates. If the parent node happens to be full too, then the restructuring process is applied again. In the worst case, splitting propagates all the way to the root and the tree grows in height by one level.

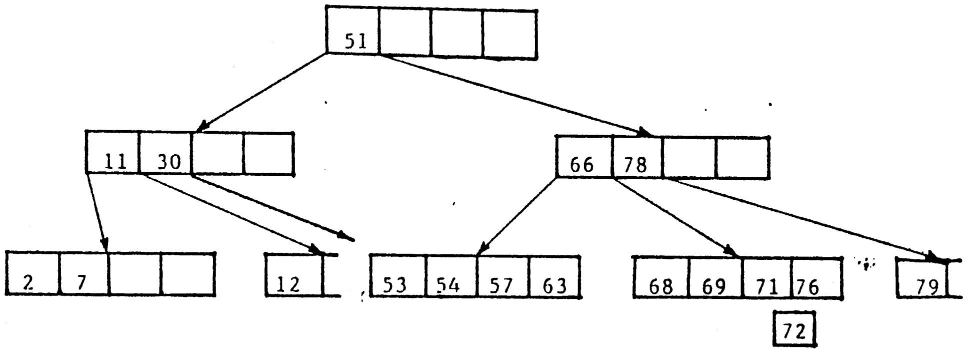
Amortized Complexity

From the definition of B-trees, it is known that each internal node in a B-tree can contain several children (at least $\lceil m/2 \rceil$ and at most m) instead of two in a binary tree. This uncertainty makes it difficult to use either for HB[1]-trees or for red-black trees as node rank in B-trees. Here, we try to modify another rank definition proposed by Tarjan [30] and others [33] for binary trees and heap structures. This node rank method is defined as follows.

$$\text{rank}(v) = \begin{cases} 0, & \text{if } v \text{ is an external node,} \\ 1 + \min\{\text{ranks of } v\text{'s children}\}, & \text{otherwise} \end{cases} \qquad (5.25)$$

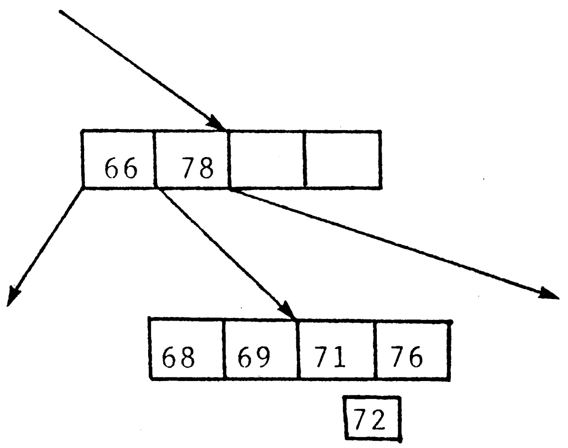


(a) A B-tree of Order 5. Number 57 is being inserted.

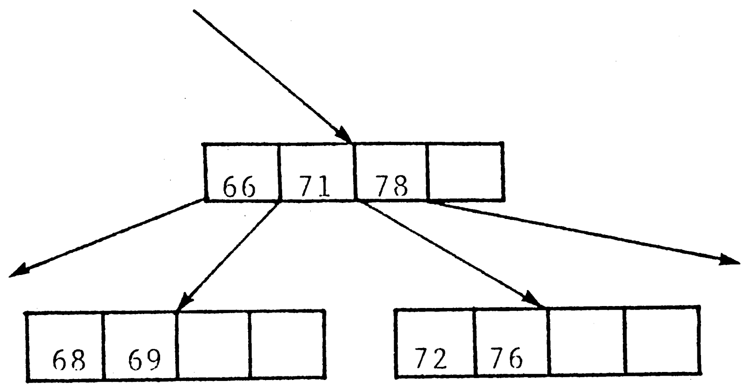


(b) The Same Tree. Number 72 is being Inserted.

Figure 33. Illustration of Single Insertion into a B-Tree



(a) A leaf and its ancestor in a B-Tree before spliting.



(b) The same subtree after insertion of "72".

Figure 34. Process of Splitting

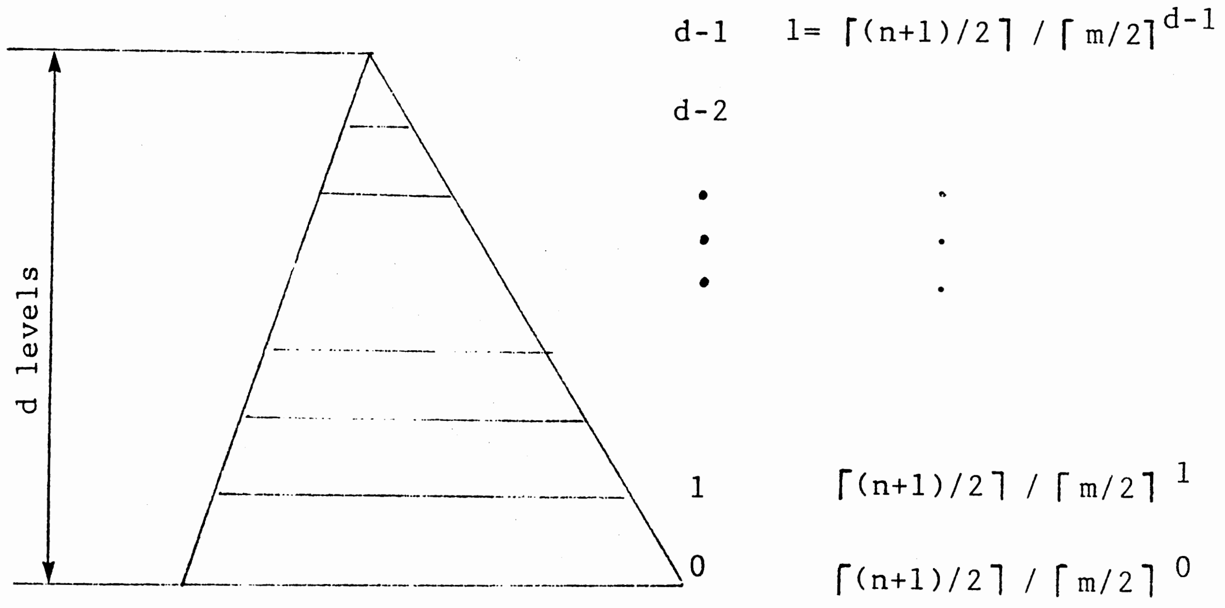


Figure 35. Height Calculation in B-Trees

That is, the rank of any node except leaves is the minimum of its children's ranks plus one (to reflect itself). Thus, the node rank can be easily determined as each node at level i will gave a rank of i based on this definition. We construct a B-tree shown in Fig. 35, where the leaf level is level zero and the root level is level (d - 1), making a total of d levels in height. To obtain the tree rank potential, we need to determine the height of a B-tree containing n keys. We use the following calculation for the consistency of notation in this study and reach the same answer given by Folk et. al. [14].

Consider that a B-tree with n keys has (n + 1) descendants from its leaf level. There are at most $\lceil (n+1) / 2 \rceil$ leaves. The maximum height occurs when each internal node (including the root) has minimum number of keys, leading to the maximum node number at level i is

$$Z_i = \lceil (n+1) / 2 \rceil / (\lceil m / 2 \rceil^i) \qquad (5.26)$$

Hence, at the root level,

$$1 <= Z_{d-1} = \lceil (n+1) / 2 \rceil / (\lceil m / 2 \rceil^{d-1}) \qquad (5.27)$$

which gives the maximum height

$$d <= 1 + \log_{\lceil m/2 \rceil} ((n+1) / 2) \qquad (5.28)$$

From this height upper bound, we can estimate the four potential functions in a B-tree built up by n sequential insertions. Using the node rank defined above, the tree potential is bounded by

$$\begin{aligned}\phi_R &<= \sum_{i=0}^{d-i} z_i * i \\ &<= ((n + 1) / 2) * (d - 1) / (m / 2) \\ &<= (n + 1) * (d - 1) / m \qquad (5.29)\end{aligned}$$

where d is the height, a function of tree order m and key number n expressed by inequality (5.28),

The amortized cost per operation is

$$a_R <= 1 + (d - 1) / m = O(1) \quad (\text{if } m >> d) \qquad (5.30)$$

In the case of using sum children method, the maximum size of each node at level i is

$$s_i <= \sum_{k=0}^{i} m^k \qquad (5.31)$$

Therefore, the tree potential is

$$\begin{aligned}\Phi_S &<= \sum_{i=0}^{d-1} z_i * s_i * (m - 1) \\ &<= ((n + 1)/2) \sum_{i=0}^{d-1} (m^i - 1)/(m - 1) \\ &<= (n + 1) * 2^{d-1} \qquad (5.32)\end{aligned}$$

and the amortized cost per insertion is

$$\begin{aligned}a_S &<= 1 + 2^{d-1} + 2^{d-1} / n \\ &= O(\log_2 n) \qquad (5.33)\end{aligned}$$

Using the log sum method, the tree potential becomes

$$\Phi_L <= \sum_{i=0}^{d-1} z_i * \lg \left(\sum_{k=0}^{i} m^k * (m - 1) \right)$$

$$<= ((n + 1) / 2) * \lg m * \sum_{i=0}^{d-1} i / (\lceil m/2 \rceil^i)$$

$$<= (n + 1) * \lg m * (d - 1) \qquad (5.34)$$

and the amortized cost per operation is bounded by

$$a_L <= 1 + \lg m * (d - 1) * (1 + 1/n)$$

$$= O(1) \qquad (5.35)$$

The last case of potential function is to use the tree height expressed by inequality (5.28),

$$\Phi_H = d <= 1 + \log_{\lceil m / 2 \rceil} ((n + 1) / 2) \qquad (5.36)$$

Thus, the per operation cost is bounded by

$$a_H <= 1 + d / n = O(1) \qquad (\text{if } n >> d) \qquad (5.37)$$

The above analysis indicates that the amortized complexity for B-tree is basically not a constant but varies with the tree size n and tree rank m (shown in Table VI). The values of a_R (5.30), a_S (5.33), a_L (5.35), and a_H (5.37) are calculated for wide ranges of n and m, and plotted on Fig.36 to show the dependence of amortized complexity on these tree parameters. It is found that the tree height has most promise because of its stability over a wide range of n.

The standard data set is also used to check the cost estimation. The process of sequential insertions of these set is shown in Fig.37 with the values of four potential functions calculated at each step. The actual costs are all below the estimated costs, as compared in Table VI.

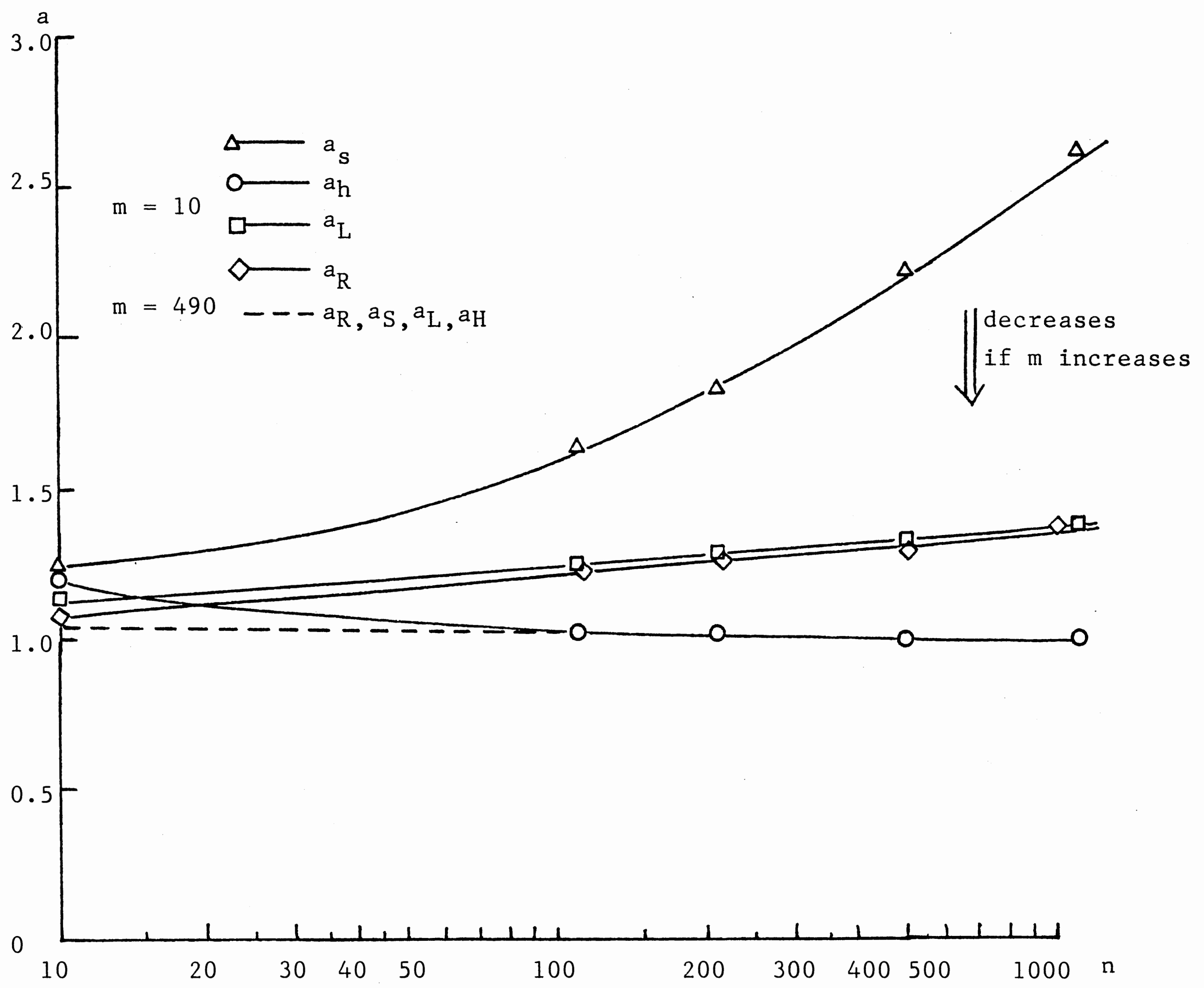
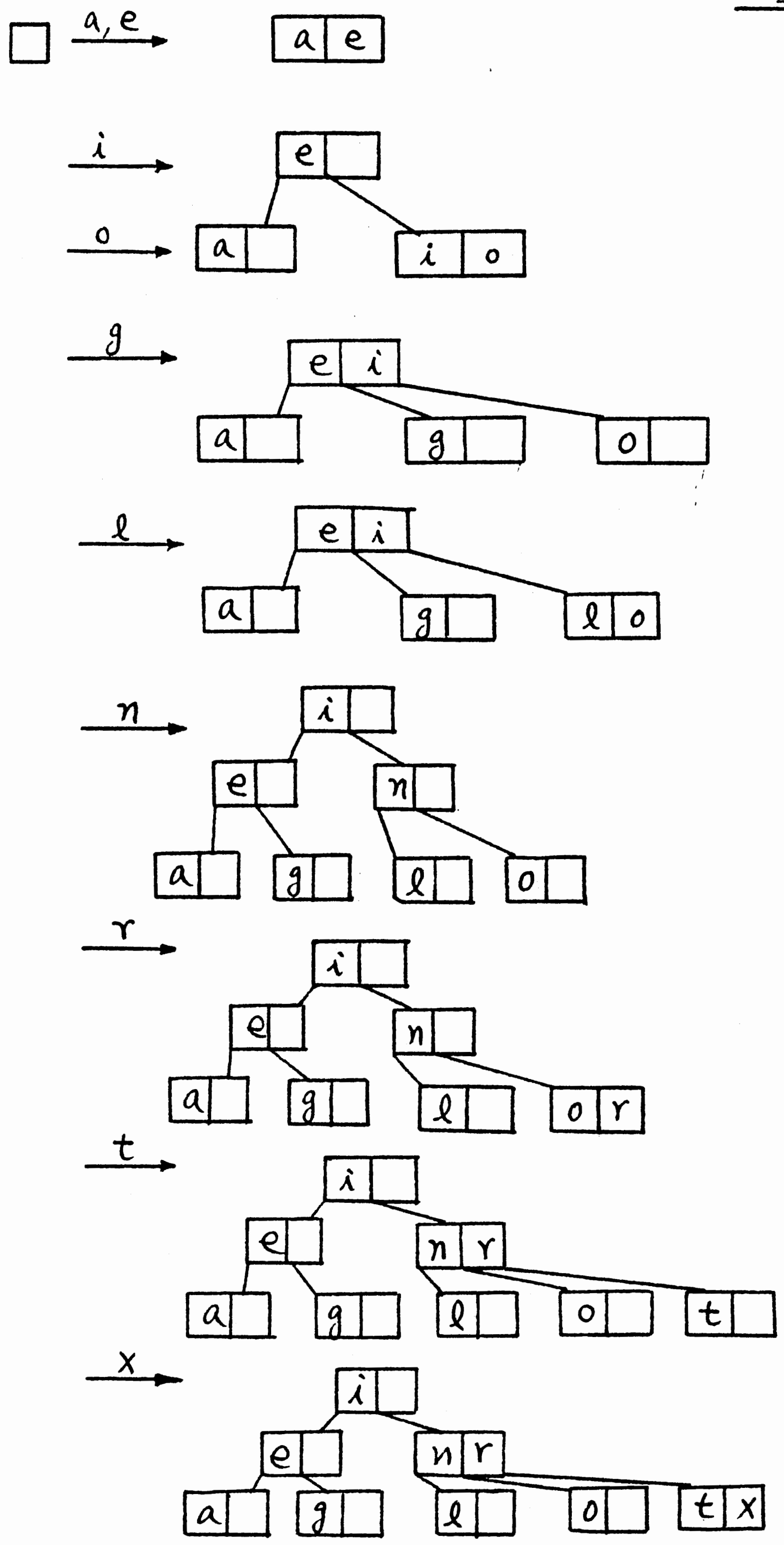


Figure 36. Effect of Parameters n and m on Amortized Behavior of B-Trees



Φ_R	Φ_S	Φ_L	Φ_H
0	1	0	1
1	5	0,47	2
1	7	0,602	2
1	7	0,602	2
4	17	1.8	3
4	17	1.8	3
4	19	2	3
4	19	2	3

Figure 37. Example of Sequential Insertions into a B-Tree (m=3)

TABLE VI

AMORTIZED COMPLEXITY OF SEQUENTIAL INSERTIONS IN B-TREES

Potential Functions	Potential Change	Amortized Time per Operation	EXAMPLE	
			Predicted	Actual
Φ_R	$(n+1)*(d-1)/m$	$O(1)$	1.819	1.4
Φ_S	$(n+1)*2^{d-1}$	$O(\log_2 n)$	3.75	2.9
Φ_L	$(n+1)*\lg m*(d-1)/m$	$O(1)$	1.43	1.2
Φ_H	d	$O(1)$	1.35	1.3

CHARTER VI

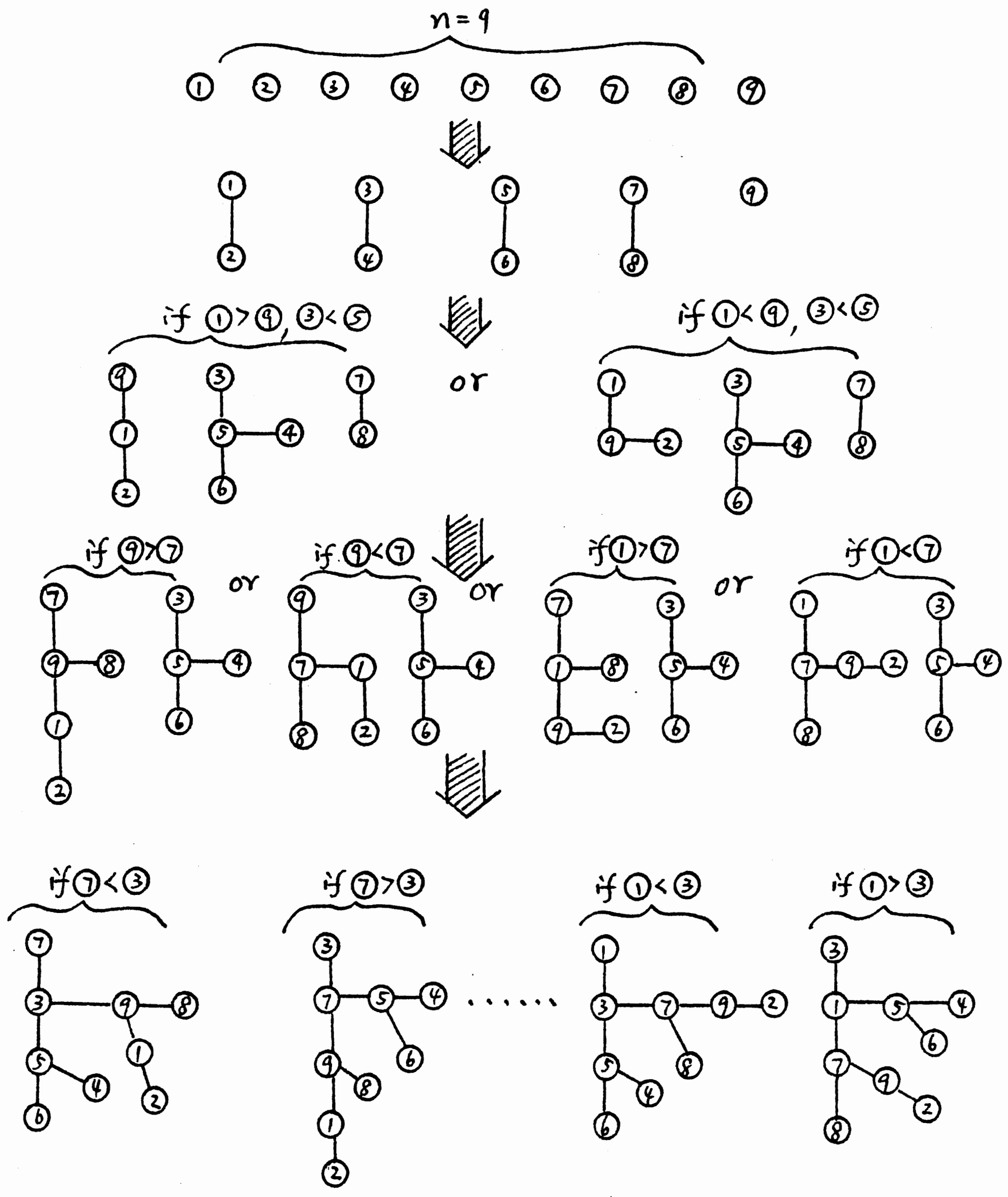
HEAP STRUCTURES

A heap is an abstract data structure consisting of a collection of items, each with an associated real-valued key. Five different heap implementations were presented in chapter II. In this chapter, we examine the amortized complexity of sequential operations in two other heap structures: the pairing heap and the self-adjusting heap. Although the amortized behavior of both these heaps has been analyzed in the past [33, 37, 38], the potential function used previously is different from the four that we are discussing now and seems more complicated. Furthermore, since one potential function was chosen for each heap, the effect of potential functions on the amortized complexity is not available. These tasks become the target of this study.

Pairing Heaps

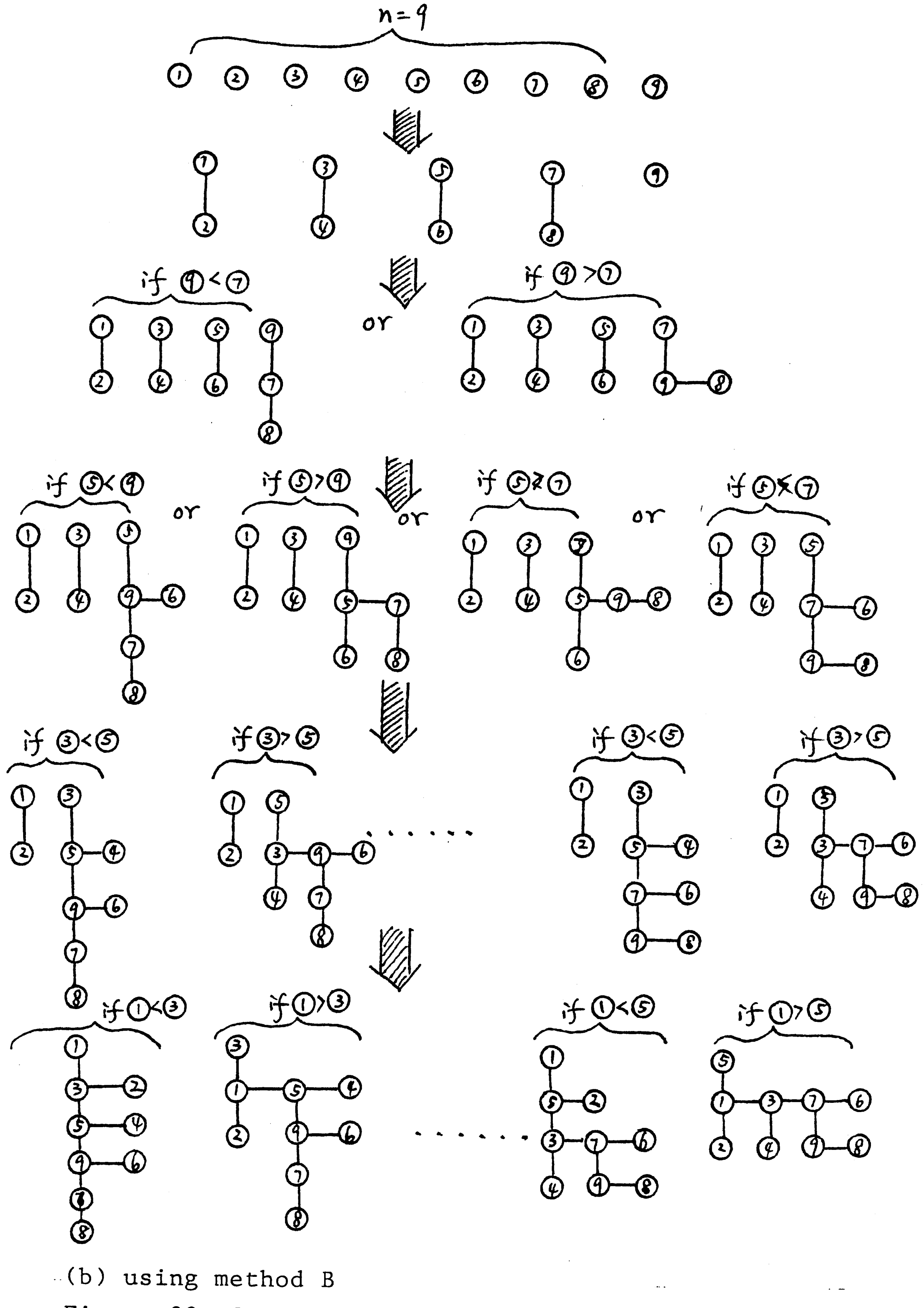
Operations

The pairing heap can be represented by an endogenous heap-ordered tree. It is an improvement of the Fibonacci heap, obtained by using the pairing method to combine trees and choosing the trees to be paired carefully. Like other heap structures, a pairing heap allows fundamental operations, such as MAKEHEAP, FINDMIN, INSERT, MELD, DECREASEKEY, and DELETE. The insertion that we are especially interested in at this stage is a process which first constructs a one node tree to contain the new item being inserted, and then links (or melds) it with another heap. In Chapter II, we already presented two pairing ways: method A and B. Specifically, for a set of items to be inserted sequentially into a pairing heap using method A, individual items are pairly linked and then these pairs are melded together. This process is illustrated in Fig.38a, where n arbitrary items are linked into $\lceil n/2 \rceil$ pairs (if n is an odd number, one item remains unlinked). Each pair is a subtree of the final heap. We link the first and second subtrees, then the third and fourth, and so on. By using method B, as shown in Fig.38b, after the first pairing path, the subtrees are linked pairly from the rear to the front. In either cases, totally $\lceil n / 2^2 \rceil$ higher subtrees can be constructed in this step. After several repeated steps, one heap containing all the n items is built up. We call the resulting data structure the pairing heap. It is seen that the linking is an essential part of the insertion operation. When we combine two heap-ordered subtrees, we use linking by making the root of smaller key the parent of the root of larger key to ensure that the final tree is in heap-order.



(a) using method A

Figure 38. A Pairing Heap Built up by n Items



(b) using method B

Figure 38. Continued.

Amortized Complexity

According to the above pairing procedures, we can analyze the amortized complexity for inserting n items into a heap by knowing the relation between the heap size and the link frequency. We consider the final heap containing n items is built up by $\lceil n/2 \rceil$ pairly linking of subtrees. As illustrated in Fig.39a, an individual subtree grows from level 0 (as an one node tree) to level d (final heap). The maximum node number for a subtree at level i is 2^i, while there are maximum $\lceil n / 2^i \rceil$ such subtrees. Since at level d there is only one final tree left, the maximum heap depth is

$$d <= \lceil \log_2 n \rceil + 1 \qquad (6.1)$$

We define the rank of each node in a n-item heap as its level rank, as we did for B-trees. In a n-item heap there are $(\lceil n/2 \rceil - 1)$ nodes at the second level after $\lceil n/2 \rceil$ pairly linking. One node becomes the root. Thus the heap potential in terms of rank is bounded by

$$\begin{aligned} \emptyset_R &<= d + (\lceil n/2 \rceil - 1) * (d-1) + \lceil n/2 \rceil * (d-2) \\ &<= d + 2 * (\lceil n/2 \rceil) * (d - 1) - (d - 1) \\ &= 2 * (\lceil n / 2 \rceil) * (d - 1) + 1 \qquad (6.2) \end{aligned}$$

and the amortized cost per operation is

$$\begin{aligned} a_R &<= 1 + d/n + d - 1 = d + d/n \\ &= O(d) \\ &= O(\log_2 n) \qquad (6.3) \end{aligned}$$

The method of sum children, and consequently the log sum approach, is more difficult here because the size of each subtree needs to be determined iteratively. In Fig.39a, the maximum size of a subtree at each level is indicated. Suppose we have an arbitrary tree T_k consisting of a root and three subtrees T_a, T_b, and T_c (with size S_a, S_b, and S_c), shown in Fig.39b. The root rank is

$$S_{root} = S_a + S_b + S_c + 1 = 2^d \qquad (6.4)$$

and the total rank potential of T_k is

$$S_k = \sum S_j + S_{root} \qquad (6.5)$$

where j = a, b, c.

This means the size of a tree of level d can be expressed based on the sizes of lower level trees, as shown in Fig.39a,

$$\begin{aligned} \sum S_j &= \sum_{i=1}^{d} (2^{d-i} * 2^{i-1}) \\ &= \sum_{i=1}^{d} 2^{d-1} \\ &= d * 2^{d-1} \end{aligned} \qquad (6.6)$$

Substituting (6.6) into (6.5), and considering the total size of a d-level heap, we obtain the potential as

$$\begin{aligned} \Phi_S = S_d &<= d * 2^{d-1} + 2^d \\ &= (d + 2) * 2^{d-1} \end{aligned} \qquad (6.7)$$

Therefore, the amortized time per operation is

$$a_S <= 1 + (d + 2) * 2^{d-1} / n$$
$$= O(d)$$
$$= O(\log_2 n) \quad (6.8)$$

Consequentially, the potential in terms of log sum children including itself can be obtained,

$$\Phi_L <= \sum_{i=1}^{d} 2^{d-i} * lg(2^{i-1}) + lg(2^d)$$
$$<= 2^d * lg2 * \sum_{i=1}^{d} (i-1)/2^i + d * lg2$$
$$= lg2 * (2^{d+2} + d) \quad (6.9)$$

and the amortized time per operation is

$$a_L <= 1 + lg2 * (2^{d+2} + d) / n$$
$$= O(1) \quad (6.10)$$

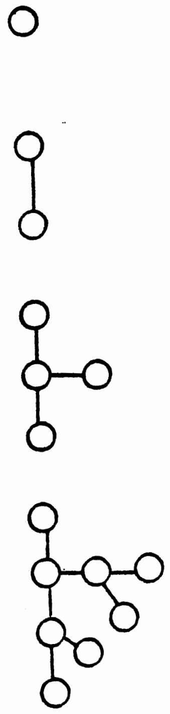
Finally, the potential in terms of heap height is determined by relation (6.1),

$$\Phi_H = d = \lceil \log_2 n \rceil + 1 \quad (6.11)$$

and the amortized time per operation is

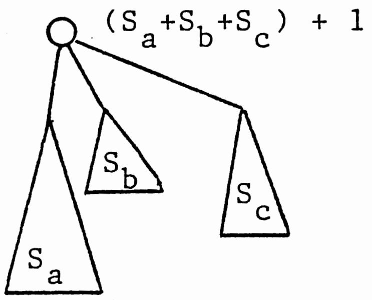
$$a_H <= 1 + \lceil \log_2 n \rceil / n + 1/n$$
$$= O(1) \quad (6.12)$$

Results from the above analysis agree with that obtained by Fredman, Sedgewick, Sleator, and Tarjan [37], to



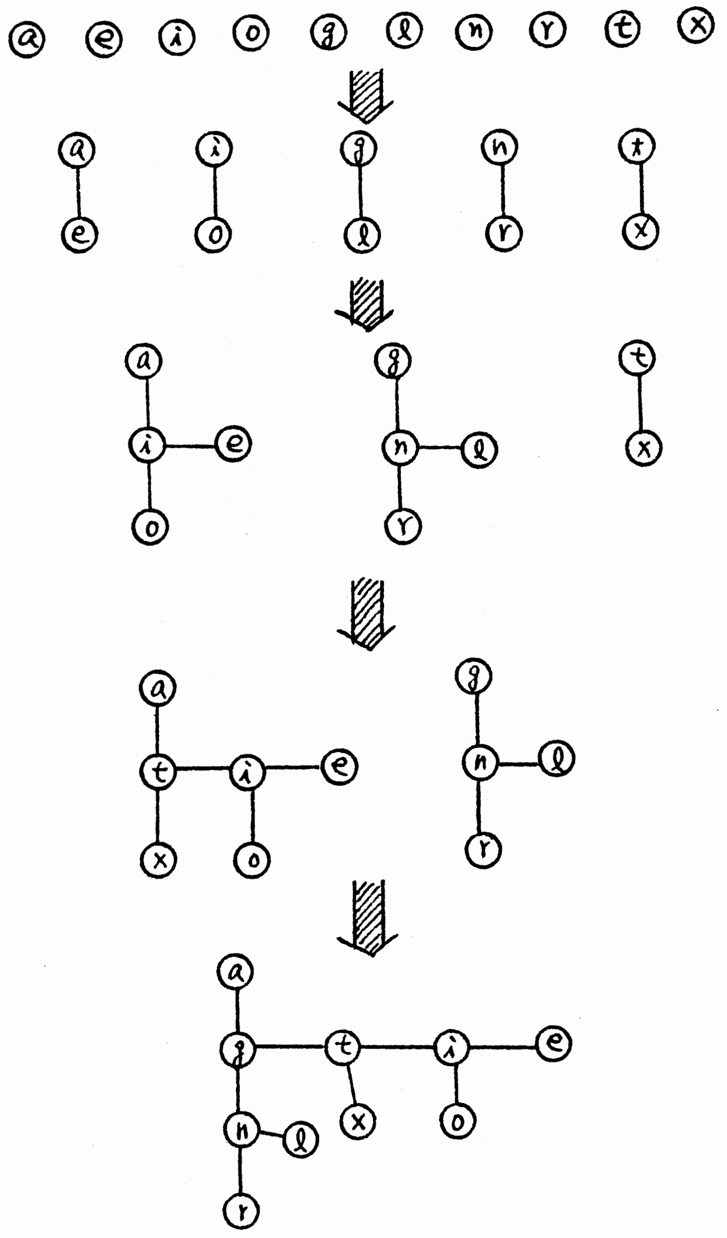
level	Max.node # per tree
0	$2^0=1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
⋮	⋮

(a) growth of individual trees



(b) a tree T_k

Figure 39. Size of Subtrees in Pairing Heaps



Φ_R	Φ_S	Φ_L	Φ_H
10	10	0	1
15	15	1.5	2
19	21	2.3	3
19	24	2.5	3
25	33	2.8	4

Figure 40. Example of Insertions into a Pairing Heap

TABLE VII

AMORTIZED COMPLEXITY OF SEQUENTIAL INSERTIONS IN PAIRING HEAPS

Potential Functions	Potential Change	Amortized Time per Operation	EXAMPLE Predicted	Actual
Φ_R	$2*\lceil n/2\rceil*(d-1)+1$	$O(\log_2 n)$	3.6	3.5
Φ_S	$(d+2) * 2^{d-1}$	$O(\log_2 n)$	4.4	4.3
Φ_L	$\lg 2 * (2^{d+2} + d)$	$O(1)$	2.1	1.2
Φ_H	d	$O(1)$	1.43	1.4

show that the amortized complexity for insertion in pairing heaps is $O(\log_2 n)$ in most cases. However, the present study further shows that by using either the log sum or the height potential, the insert costs $O(1)$ which is a conjecture unproved by Fredman et al. (shown in Table VII). We demonstrate our analysis by using the standard data set (Fig.40). The four potentials are calculated at each step. The final actual cost in each case is also listed in Table VII to compare with the theore-tical estimate.

Self-Adjusting Heaps

The self-adjusting heap is a data structure that guarantees efficiency on basis of amortized complexity measure by adjusting the structure during each update operation instead of maintaining an explicit balance constraint. The pairing heap also can be considered as a form of self-adjusting structure due to the pairing step in updating. However, a typical self-adjusting heap analyzed here is the one recently developed by Sleator and Tarjan [38], and called skew heap which swaps the structure after every updating to speed up later operations, just like the move-to front algorithm in a linear list.

Operations

The skew heap can be represented by a heap-ordered binary tree. That is, the nodes in a binary tree are ordered in such a way that if P(v) is the parent of v, then key(P(v)) < key(v). A skew heap allows basic operations such as MAKEHEAP, FINDMIN, INSERT, DELETEMIN, and MELD. It also can be updated in either a top-down approach or a bottom-up approach. In this section we examine the effect of potential functions on skew heaps and use the top-down insertion as our analysis target. The process of insertion in a top-down skew heap is best described as follows.

1. Make item v into a one node heap h.
2. Meld h_1 with another heap h_2 by
   (a) merging the right paths of h_1 and h_2 in heap order,
   (b) Swapping the left and right children of every node on the merge path except the lowest.

Therefore, the meld operation is the major part of an insertion process and is shown in Fig.41, where the merge path is indicated.

Amortized Complexity

Sleator and Tarjan [38] used a potential function of heavy right nodes to show that the insertion costs $(\log_2 n)$ in top-down skew heap, because the length of the right path in a skew heap (which is a self-adjusting form of the leftist trees) of n nodes is at most $\lceil \log_2 n \rceil$. The time for the meld is bounded by a constant times the length of the merge path. Actually the depth of a skew heap in the illness case after n insertions is bounded by

$$d <= n \qquad (6.13)$$

Assume that the root is at level 1 and the lowest level is d. The maximum number of nodes at any level i, therefore, is 2^{i-1}. If we define the rank of a node by the number of levels where the node exists, then the root rank is d; the node at lowest level ranks 1; and the node at level i is (d - i + 1). Thus, the total rank potential of the heap is obtained as

$$\begin{aligned} \phi_R &<= \sum_{i=0}^{d} 2^{i-1} * (d - i + 1) \\ &= (d + 1) \sum_{i=1}^{d} 2^{i-1} - \sum_{i=1}^{d} i * 2^{i-1} \\ &= \sum_{i=1}^{d} i * 2^{d-i} = 2^d * \sum_{i=1}^{d} i / 2^i \\ &<= 2^d \end{aligned} \qquad (6.14)$$

This gives the amortized cost per operation as

$$\begin{aligned} a_R &<= 1 + 2^d / n \\ &= O(\log_2 n) \end{aligned} \qquad (6.15)$$

Since the skew heap is represented by a binary tree, the maximum size of a node at level i (refer Fig.29) can be expressed as

$$S_i = 2 * S_{i-1} + 1 = 2^{i+1} - 1 \qquad (6.16)$$

Therefore, the total heap size is bounded by

$$\Phi_S <= \sum_{i=0}^{d} \lceil n / 2^i \rceil * (2^{i+1} - 1) \tag{6.17}$$

which implies that the amortized time per operation is

$$\begin{aligned} a_S <= 1 + 2 * d &= O(d) \\ &= O(n) \end{aligned} \tag{6.18}$$

While the total heap potential in terms of log sum children is bounded by

$$\Phi_L <= \sum_{i=0}^{d} \lceil n / 2^i \rceil * \lg(2^{i+1} - 1) \tag{6.19}$$

which gives the amortized time as

$$\begin{aligned} a_L &<= 1 + \sum_{i=0}^{d} \lg(2^{i+1} - 1) / 2^i \\ &<= 1 + \sum_{i=0}^{d} \lg(2^{i+1}) / 2^i \\ &= O(1) \end{aligned} \tag{6.20}$$

Finally, we use the height (6.13) to estimate the heap potential after n insertions,

$$\Phi_H = d <= n \tag{6.21}$$

Thus, the amortized time per insertion is

$$\begin{aligned} a_H <= 1 + n / n &<= 2 \\ &= O(1) \end{aligned} \tag{6.22}$$

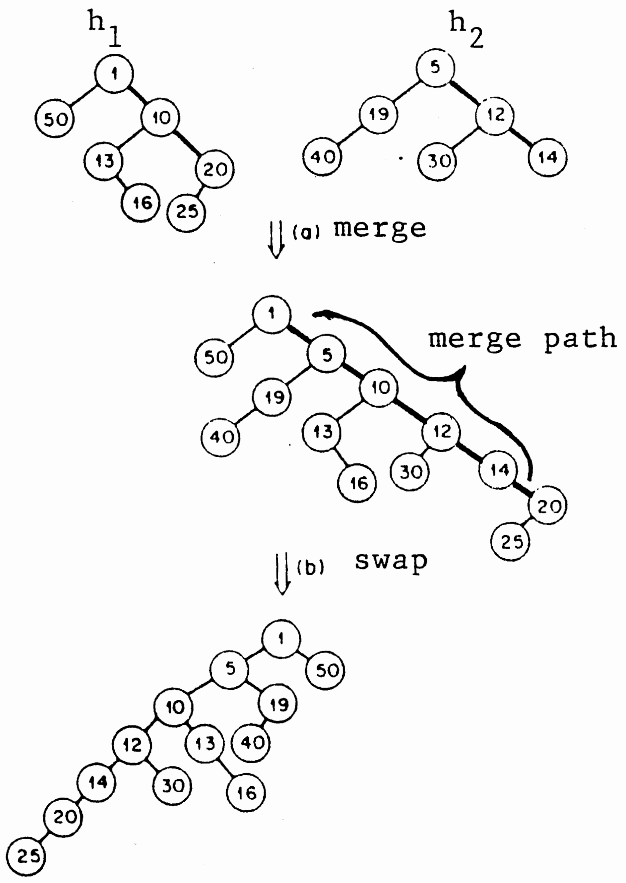
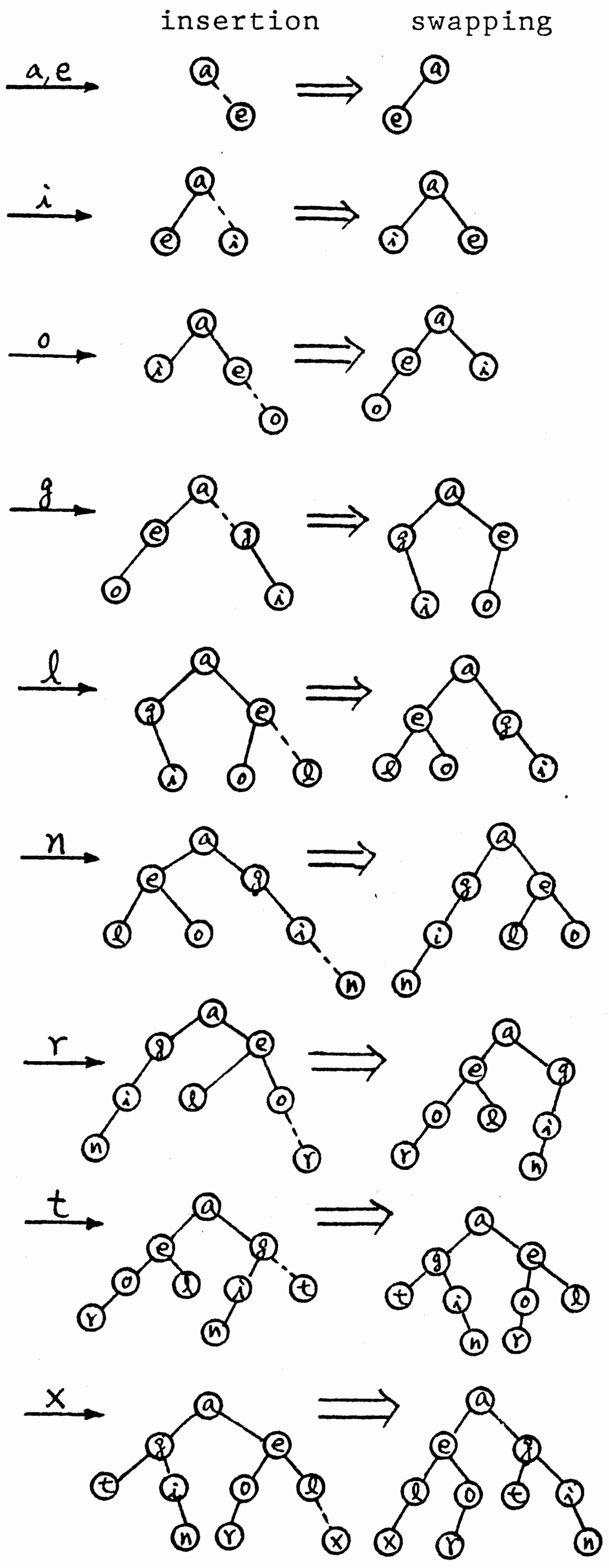


Figure 41. Meld Operation in a Self-Adjusting Heap



Φ_R	Φ_S	Φ_L	Φ_H
3	3	0,3	2
4	5	0,47	2
8	8	0.9	3
9	11	1,3	3
10	14	1,56	3
17	18	2.1	4
18	22	2.58	4
20	27	2.76	4
21	29	3.2	4

Figure 42. Example of Insertions into a Self-Adjusting Heap

TABLE VIII

AMORTIZED COMPLEXITY OF SEQUENTIAL INSERTIONS IN SELF-ADJUSTING HEAPS

			EXAMPLE	
Potential Functions	Potential Change	Amortized Time per Operation	Predicted	Actual
Φ_R	2^d	$O(\log_2 n)$	4.46	3.1
Φ_s	$\sum_{0}^{d} \lvert n/2^i \rvert * (2^{i+1}-1)$	$O(n)$	4.46	3.9
Φ_L	$\sum_{0}^{d} \lvert n/2^i \rvert * \lg(2^{i+1}-1)$	$O(1)$	1.5	1.3
Φ_H	d	$O(1)$	1.45	1.4

An example of a self-adjusting heap is built up by sequentially inserting the standard data set which consists of ten items (Fig.42). The four potential functions are calculated at each step. In each case the predicted cost is compared with the actual value as shown in Table VIII. It is found that there is a good match between the two.

CHAPTER VII

COMPARISON AND DISCUSSION

In chapter IV, V, and VI, we analyzed the amortized computational complexity of some basic updating algorithms on linear lists, balanced search trees, and heap data structures, respectively. The main tool is to define a potential function that can map the configuration of the data structure into a real number before and after operations. Then the amortized cost of an operation can be determined based on the variation of the potential function. Four classes of potential functions have been applied in the analysis of the same operation, each gives an amortized complexity as summarized by a table for the corresponding data structure. To further understand the amortized analysis for algorithms and data structures, we compare these results and discuss the effect of potential function on amortized analysis. In addition the amortized costs are compared with that from worst-case analysis.

We summarize the amortized cost per operation in Table IX. For linear list structure, the 'a' reflects the computational complexity of an arbitrary sequence of operations including push (or inject), pop, and access on a stack (or a queue). For both balanced search trees and heap structures, the 'a' is the complexity of a sequence of insertions into the AVL-trees (HB[1]-trees), red-black trees, B-trees, pairing heaps, and self-adjusting heaps. The rebalance in these trees and heaps is performed by bottom-up algorithm except in the self-adjusting heap in which the top-down algorithm is employed. From Table IX, we have the following comparisons.

1. The potential function used in amortized analysis does affect the computation complexity. Among the four potential functions, the log sum and the structure height are more promising than the other two for they provide the smallest amortized time of $O(1)$ for sequential operations in both the tree and heap structures analyzed. In most tree structures, the height potential can be easily determined. However, it may be hard to estimate for some structures, such as the self-adjusting heaps. In this case, other potential functions should be considered, especially the log sum potential.
2. The rank potential we analyzed gives good results for linear list (frequency count), red-black tree (number of red sons), and B-tree (1 + min. children's rank). This method is more flexible since its utility depends on the choice of the rank of each node. Sometimes this potential function is the same as the sum children and the log sum children, if we define the node rank in that way. But we choose the rank method because it is general and can include many other variables, such as

the right heavy nodes, etc.

3. The sum children including itself and the log sum children are two similar potential functions. These two potentials are not suitable for linear list operations because they give the amortized times that are not constant but vary with the operation frequencies. The advantage of using these two methods is that the node size(sum children including the node itself, or log this size) usually can be determined easily.
4. Our results are compared with that obtained by Tarjan et. al. and shown in Table IX. It can be seen that all the amortized costs based on the four potential functions are in agreement with Tarjan's results except the a_R (using frequency count as rank potential) for HB[1]-trees. The potential of both the log sum and the structure height even give better cost estimations for HB[1]-trees, pairing heaps, and self-adjusting heaps.
5. The correctness of our results are also demonstrated by using a standard data set for each data structure. From the comparison between the actual value and the cost predicted based on each potential function, we find that the current analysis of potential function does provide a correct measure of amortized complexity.
6. In worst-case analysis, we sum the worst-case times of the individual operations. In conventional applications a search for any item in a n-node binary tree, which has a depth of at least $\log_2 n$, takes $O(\log n)$ steps. Once

the position to insert a new item or delete an old one has been found, the insertion or deletion can be completed in $O(\log n)$ additional steps for rebalancing, either starting from the root and proceeding towards the leaves (top-down algorithm) or starting from the leaf level and walking up towards the root (bottom-up algorithm). Therefore, one can guarantee an $O(\log n)$ worst-case complexity for sequential insertions. According to the theory of amortization, the amortized cost is not only an upper bound of actual operation time but also an estimation better than the worst-case measure. This is proved by the present study, because the results shown in Table IX are $O(1)$ in many cases and at most $O(\log n)$ in remaining cases.

7. Therefore, amortized analysis provides a suitable approach for estimating the performance of sequential operation algorithms in data structures. The basic procedure of amortized analysis consists of choosing a proper potential function which can reflect the configuration of the structure at any operation interval and determining the total increment in potential function. Based on the present study, the structure height is an optimal class of potential function to be chosen.

TABLE IX

SUMMARY OF EFFECT OF POTENTIAL FUNCTIONS ON AMORTIZED COMPLEXITY FOR SEQUENTIAL INSERTIONS IN DATA STRUCTURES

Amortized Cost		a_R	a_S	a_L	a_H	Tarjan's
Data Struc.		Rank	Sum children	log sum	Height	results
LIST	Linear list	O(1)	$f_1(n_0,k-p)$	$f_2(n_0,k-p)$	O(1)	O(1)
BALAN- CED SEARCH TREES	HB[1]	O(1)	$O(\log_2 n)$	O(1)	O(1)	$O(\log_2 n)$
	Red-Black	O(1)	$O(\log_2 n)$	O(1)	O(1)	O(1)
	B	O(1)	$O(\log_2 n)$	O(1)	O(1)	--
HEAPS	Pair-ing	$O(\log_2 n)$	$O(\log_2 n)$	O(1)	O(1)	$O(\log_2 n)$
	Self-Adj.	$O(\log_2 n)$	O(n)	O(1)	O(1)	$O(\log_2 n)$

CHAPTER VIII

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FURTHER STUDY

Summary and Conclusions

Using amortized computational complexity to judge the performance of data structure algorithms is a new approach developed by Tarjan and promoted by many others. It is motivated by the observations that in most data structure applications a sequence of operations are encountered and that there are usually some correlations among these operations. By averaging the running time per operation over a worst-case sequence of operations (reflected by data structure configuration), it is possible to obtain a time bound smaller than the worst-case cost and, consequently, to provide a more accurate estimation on dynamic performance of data structures.

In the present study, the situation of amortization in linear lists, balanced search trees, and heap structures is thoroughly reviewed. Four classes of potential functions in these data structures are analyzed and the corresponding amortized costs are calculated. Several conclusions from this study are summarized as follows.

1. It is found that among all the published investigations on amortization, Tarjan's theory is the most fundamental and precise one that can be formulated mathematically and applied to any data structures.

2. The concept of potential function proposed by Tarjan for physicist's view analysis can also be extended to the banker's view in order to have an uniform analysis. The potential function is useful because it can reflect the change of data structure configuration caused by operations; however, it has not been analyzed carefully in the past. The level of difficulty of amortization depends on the potential function used in analysis.

3. Four potential functions -- node rank, sum children including itself, log sum, and structure height -- are proposed and analyzed in the present study for each of the three data structures : lists, trees, and heaps. In each case, the potential function is formulated and the amortized cost per operation is calculated.

4. The results show that the log sum and the height are the two better ones among the four potential functions for the tree and heap structures analyzed. For any structure whose height is determinable, the height method is to be considered. The sum children potential usually gives the worst estimation, because of its overestimation and unconstant characteristics. The efficiency of node rank depends on the definition of rank. This approach is

flexible but may fail if the rank is determined incorrectly.

5. A standard data set is used as an example for the amortized analysis of each case. The actual amortized costs are well below the predicted values.

6. The amortized analysis based on the four potential functions agree with Tarjan's result for most cases. The results from the log sum and the height potential are even better than Tarjan's estimation for HB[1]-trees, pairing heaps, and self-adjusting heaps.

7. We must be careful in choosing the potential function for a specific data structure and its updating algorithm.

8. This study also proves that the amortized cost based on potential function analysis is a measure better than that by worst-case analysis because the former is at most (log n) in some cases but (1) in most other cases analyzed.

Suggestions

The most obvious suggestion for further study is to analyze more complicated operation sequences that are realistic in data structure applications, such as the intermixed operations including arbitrary access, insertion, and deletion in trees and arbitrary combination of meld, delete min, insertion, etc. in heaps. Although the analysis method can possibly be similar,there will be more derivation effort and more intense mathematical analysis.

Another suggestion is to explore the possibility of other classes of potential functions. It will be a great achievement to have a more clear guidance for choosing the potential function in major data structures.

Finally, there are some other data structures (such as the self-adjusting trees) and algorithms (such as the top-down algorithm in trees and bottom-up algorithm in heaps) to be analyzed from the view of amortization. The choice of potential function to use in these studies is also important and to be investigated.

REFERENCES

1. Horowitz,E. and S.Sahni, Fundamentals of Data Structures in Pascal, 2nd Ed., Computer Sci. Press, Inc., 1987.

2. Aho, A.V., J.E.Hopcroft, and J.D.Ullman, Data Structures and Algorithms, Addison-Wesley, 1983.

3. Tarjan, R.E., Data Structures and Network Algorithms, Soc.Industrial and Applied Mathematics, PA, 1983.

4. Navathe, S.B. and J.P.Fry, "Restructuring for Large Databases: Three levels of Abstraction", ACM Trans. Database Systems, 1, 2 (1976), 138-158.

5. Bancilhon, F. and N.Spyratos, "Update Semantics of Relational Views", ACM Trans. Database Systems, 6, 4 (1981), 557-575.

6. Lehman, P.L. and S.B.Yao, "Efficient Locking for Concurrent Operations on B-Trees", ACM Trans. Database Systems, 6, 4 (1981), 650-670.

7. Roussopoulos, N.,"View Indexing in Relational Database", ACM Trans. Database Systems, 7, 2 (1982), 258-290.

8. Hawryszkiewycz, I.T., Database Analysis and Design, Science Research Associate, Inc., 1984.

9. Smith, H.C., "Database Design: Composing Fully Normalized Tables from a Rigorous Dependency Diagram", Comm. of ACM, 28, 8 (1985), 826-838.

10. Teorey, T.J. and J.P. Fry, Design of Database Structures, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

11. Date, C.J., An Introduction to Database Systems, Vol. I, Fourth Ed., Addison-Wesley Publishing Company, 1986.

12. Kruse, R.L., Data Structures and Program Design, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

13. Golden, D. and M. Pechura, "The Structure of Microcomputer File Systems", Comm. of ACM, 29, 3 (1986), 222-230.

14. Folk, M. J. and B. Zoellick, File Structures : A Conceptual Toolkit, Addison-Wesley Publishing Company, Inc., 1987.

15. Bitner, J.R., "Heuristics that Dynamically Organize Data Structures", SIAM J. Comput., 8, 1 (1979), 81-110.

16. Comer, D., "The Ubiquitous B-Tree", Computing Surveys, 11, 2 (1979), 121-137.

17. Brown, M.R. and R.E. Tarjan, "Design and Analysis of a Data Structure for Representing Sorted Lists", SIAM J. COMPUT., 9, 3 (1980), 594-614.

18. Rosenberg, A.L. and L. Snyder, "Time- and Space-Optimality in B-Tree", ACM Trans. on Database Systems, 6, 1 (1981), 174-193.

19. Huddleston, S. and K. Mehlhorn, "A New Data Structure for Representing Sorted lists", Acta Informatica, 17, 1 (1982), 157-184.

20. Christodoulakis, S., "Implications of Certain Assumptions in Database Performance Evaluation", ACM Trans. Database Systems, 9, 2 (1984), 163-186.

21. Manber, U. and R.E.Ladner, "Concurrency Control in a Dynamic Search Structure", ACM Trans.Database Systems, 9, 3 (1984), 439-355.

22. Schkolnick, M. and P. Tiberio, "Estimating the Cost of Updates in a Relational Database", ACM Trans. Database Systems, 10, 2 (1985), 163-179.

23. Sarnak, N. and R.E. Tarjan, "Planar Point Location using Persistent Search Trees", Comm. of ACM, 29, 7 (1986), 669-679.

24. Mahmond, H.M., "On the Average Internal Path Length of m-ary Search Trees", Acta Informatica, 23, 1 (1986), 111-117.

25. Bender, E.A., C.E.Praeger, and N.C.Wormald, "Optimal Worst Case Trees", Acta Informatica, 24, 2 (1987), 475-489.

26. Shmueli, O. and A. Itai, "Complexity of Views : Tree and Cyclic Schemas", SIAM J. COMPUT., 16, 1 (1987), 17-37.

27. Fedorowicz, J., "Database Performance Evaluation in an Indexed File Environment", ACM Trans.Database Systems, 12, 1 (1987), 85-110.

28. Chaitin, G.J., "On the Length of Programs for Computing Finite Binary Sequences", J.Assoc. Comput. Math., 13, 2 (1966), 547-569.

29. Paul, W.J., J.I.Seiferas, and J.Simon, "An Information-Theoretic Approach to Time Bounds for On-Line Computation", J.Comput. System Sci., 23, 1 (1981), 108-126.

30. Tarjan, R.E., "Amortized Computational Complexity", SIAM J. ALG. DISC. MATH, 6, 2 (1985), 306-318.

31. Chen, Y.K., "Search Tree Data Structures and Their Applications", (Unpub. M.S. thesis, Oklahoma State University, 1987.)

32. Sleator, D.D. and R.E. Tarjan, "Amortized Efficiency of List Update and Paging Rules", Communications of the ACM, 28, 2 (1985), 202-208.

33. Fisher, D.D., Data Structures (COMSC 5413) Class-Notes, Department of Computing and Information Sciences, Oklahoma State University, Fall 1987.

34. Hu, R.L., "Dynamic View on Worst-Case and Amortized Complexity for B-Tree under Sequential Insertions", Project Report, COMSC 5413, OSU, Fall 1987.

35. Mehlhorn, K. and A. Tsakalidis, "An Amortized Analysis of Insertions into AVL-Trees", SIAM J. COMPUT, 15, 1 (1986), 23-33.

36. Tarjan, R.E., "Efficient Top-Down Updating of Red-Black Trees", Tech. Rept. CS-TR-013-86, Princeton University, June, 1985.

37. Fredman, M.L., R.Sedgewick, D.D.Sleator, and R.E.Tarjan, "The Pairing Heap: A New Form of Self-Adjusting Heap", Technical Report, CS-TR-008-85, Computer Sci. Dept., Princeton University, July, 1985.

38. Sleator, D.D. and R.E.Tarjan, "Self-Adjusting Heaps", SIAM J. COMPUT., 15, 1 (1986), 52-69.

39. Bentley, J.L. and C.C.McGeoch, "Amortized Analysis of Self-Organizing Sequential Search Heuristics", Comm. of ACM, 28, 4 (1985), 404-411.

40. Shmueli, O. and A. Itai, "Complexity of Views : Tree and Cyclic Schemas", SIAM J. COMPUT., 16, 1(1987), 17-37.

41. Vaishnavi, V.K., "Weighted Leaf AVL-Trees", SIAM J. COMPUT., 16, 3 (1987), 503-537.

42. Rivest, R., "On Self-Organizing Sequential Search

Heuristics", Comm. of ACM, 19, 2 (1976), 82-110.

43. Bentley, J.L. and C.C.McGeoch, "Worst-Case Analysis of Self-Organizing Sequential Search Heuristics", Proc. 20th Allerton Conf. on Comm., Control, and Comput., (University of Illinois, Urbana-Champaign, Oct.6-8, 1982), 1983, 452-461.

44. Tarjan, R.E., "Algorithm Design", Comm. of ACM, 30, 3 (1987), 205-212.

45. Knuth, D., The Art of Computer Programming, Vol. 3 : Sorting and Searching, Addison-Wesley Publ. Co., Reading, Mass., 1973.

46. Bayer, R. and E. McCreight,"Organization and Maintenance of Large Ordered Indexes", Acta Informatica, 1, 3 (1972), 290-306.

VITA

Ruo-Ling Hu

Candidate for the Degree of

Master of Science

Thesis: AMORTIZED ANALYSIS OF COMPUTATIONAL COMPLEXITY FOR DATA STRUCTURES

Major field: Computing and Information Sciences

Biographical:

Personal Data: Born in Shanghai, China, the daughter of Ji-Deng Hu and Wan-Zheng Zhu. Married.

Education: Graduated from the 55th High School, Shanghai, China; received Bachelor of Science Degree in Engineering from the Shanghai Institute of Mechanical Engineering in August, 1984; received the Master of Science degree in Physics from Oklahoma State University in May, 1987; completed requirements for the degree of Master of Science in Computing and Information Sciences at Oklahoma State University in December, 1988.

Professional Experience: Associate Engineer, Gansu Optical Instruments General Plant, Gansu, China, 1/1967-5/1980; Design Engineer,Hanjiang Precision Machine Tool Research Institute, Hanzhong, China, 8/1980-5/1984.
Teaching Assistant and Research Assistant, at the Department of Physics, Oklahoma State University, 8/1984-5/1987.