NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

SEARCH TREES IN C++

Graduation Project
COM-400

Student: Mohammed Imran Khan(990829)

Supervisor: Assist.Professor Dr.Firudin Muradov

NICOSIA-2003

TABLE OF CONTENTS

NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

SEARCH TREES IN C++

Graduation Project
COM-400

Student: Mohammed Imran Khan(990829)

Supervisor: Assist.Professor Dr.Firudin Muradov

NICOSIA-2003

TABLE OF CONTENTS

ACKNOWLEDGEMENT

All praise is for Allah. I will begin my acknowledgement by first of all thanking Assist.Prof.Dr Firudin Muradav who guided me through the tough task I understood while making this project.

I would also like to thank my parents for supporting me for my studies here. Due to their endless efforts and prayers for my success, I was able to reach my goal.

ABSTRACT

There are four major types of search trees. First type is **Binary Search Trees**. These trees provide an asymptotic performance that is comparable to that of skip lists. The expected complexity of a search, insert, or delete operation is O(log$n$), while the worst case complexity is $\Theta(n)$.

The second type of search trees is **AVL Trees.** AVL trees perform at most one rotation following an insert and O(log$n$) rotations following a delete.

The third type of search trees is **Red-Black Trees**. The run-time performance of these trees are slightly slower than AVL trees. red-black trees perform a single rotation following an insert or delete, the overall insert or delete time remains O(log$n$) if we use a red-black tree to represent a priority search tree.

The fourth type of search trees is **B-Trees**. These are search trees of higher degree. For larger dictionaries (called **external dictionaries** or **files**) that must reside on a disk, we can get improved performance from this type.

INTRODUCTION

Computer programming requires sophisticated tools to cope with the complexity of real applications, and only practice with these tools builds skill in their use. Many of the algorithms and data structures possess an intrinsic elegance, a simplicity that covers the range and power of their applicability.

The program development process requires to represent data in an effective way and develop a suitable step-by-step procedure (or algorithm), which can be implemented as a computer program.

The world of data structures has a wide variety of trees. In the present project we discuss search trees. It develops tree structures suitable for the representation of a dictionary.

We begin chapter 1 by examining binary search trees. These trees provide an asymptotic performance that is comparable to that of skip lists. The expected complexity of a search, insert, or delete operation is O(log$n$), while the worst case complexity is $\Theta(n)$. Next two chapters we consider two of the many known varieties of balance trees: AVL and red-black trees. When either AVL or red-black trees are used, a search, insert, or delete can be performed in logarithmic time (expected and worst case). The actual run-time performance of both structures is similar, with AVL trees generally being slightly faster. Both balanced tree structures use "rotations" to maintain balance. AVL trees perform at most one rotation following an insert and O(log$n$) rotations following a delete. However, red-black trees perform a single rotation following either an insert or delete. This difference is not important in most applications where a rotation takes $\Theta(1)$ time. It does, however, become important in advanced applications where rotation cannot be performed in constant time. One such application is priority search trees are used to represent elements with two-dimensional keys. In this case each key is a pair (x,y). A priority search tree is simultaneously a priority queue on y and a search tree on x. When rotations are performed in these trees, each has a cost of O(log$n$). Since red-black trees perform a single rotation following an insert or delete, the overall insert or delete time remains O(logn) if we use a red-black tree to represent a priority search tree. When we use an AVL tree, the time for the delete operation becomes O(log$n$).

Although AVL and red-black trees provide good performance when the dictionary being represented is sufficiently small to fit in our computer's memory, they are quite inadequate for larger dictionaries. When the dictionary resides on disk, we need to use search trees with a much higher degree and hence a much smaller height. An example of such a search tree, the B-tree, is also considered in the 4th chapter.

Three applications of search trees are developed in the applications section. The first is the computation of a histogram. The second application is the implementation of the best-fit approximation method for the NP-hard bin-packing problem of Section 5.1.

CHAPTER 1
BINARY SEARCH TREES

1.1.Definition

A **binary search tree** is a binary tree that may be empty. A nonempty binary search tree satisfies the following properties:

1.Every element has a key (or value) and no two elements have the same key; therefore, all keys are distinct.

2.The keys (if any) in the left subtree of the root are smaller than the key in the root.

3.The keys (if any) in the right subtree of the root are larger than the key in the root.

4.The left and right subtrees of the root are also binary search trees.

There is some redundancy in this definition. Properties 2,3, and 4 together imply that the keys must be distinct. Therefore, property 1 can be replaced by the following property: The root has a key. The preceding definition is, however, clearer than the nonredundant version.

Some binary trees in which the elements have distinct keys appear in Figure 1.1. the number inside a node is the element key. The tree of Figure 1.1(a) is not a binary search tree despite the fact that it satisfies properties 1, 2, and 3. The right subtree fails to satisfy property 4. This subtree is not a binary search tree as its right subtree has a key value (22) that is smaller than that in the subtrees' root (25). The binary trees of Figures 1.1(b) and (c) are binary search trees.

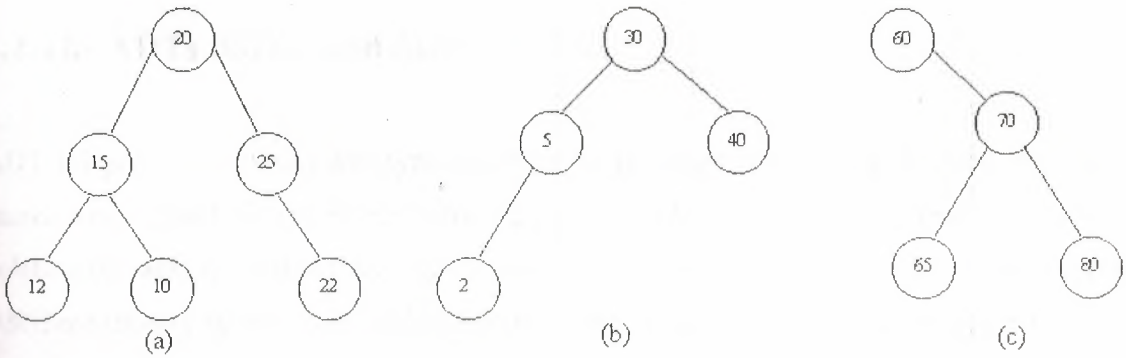


Figure 1.1 Binary trees

We can remove the requirement that all elements in a binary search tree need distinct keys. Now we replace smaller in property 2 by ≤ and larger in property 3 by ≥; the resulting tree is called **binary search tree with duplicates.**

An **indexed binary search tree** is derived from an ordinary binary search tree by adding the field LeftSize to each tree node. This field gives the number of elements in the node's left subtree plus one. Figure 1.2 shows two indexed binary search trees. The number inside a node is the element key, while that outside is the value of LeftSize. Notice that LeftSize also gives the rank of an element with respect to the elements in the subtree. For example, in the tree of figure 11.2(a), the elements (in sorted order) in the subtree with root 20 are 12, 15, 18, 20, 25 and 30. The rank of the root is four (i.e., the element in the root is the fourth element in sorted order). In the subtree with root 25, the elements (in sorted order) are 25 and 30, so the rank of 25 is one and its LeftSize value is 1.

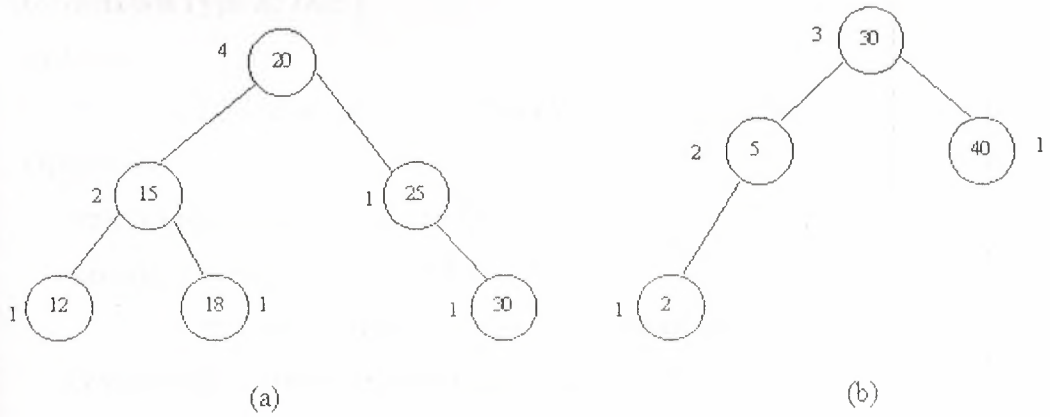


Figure 1.2 Indexed binary search trees

1.2.The ADTs *BSTree* and *IndexedBSTree*

ADT 1.1 gives the abstract data type specification for binary search tree. An indexed binary search tree supports all the binary search tree operations. In addition, it supports search and deletion by rank. Its abstract data type specification is given in ADT 1.2. The abstract types *DBSTree* (binary search trees with duplicates) and *DIndexedBSTree* may be specified in a similar way.

AbstractDataType *BSTree* {

Instances

Binary trees, each node has an element with a key field; all keys are distinct; keys in the left subtree of any node are smaller than the key in the node; those in the right subtree are larger.

Operations

Create(); create an empty binary search tree

Search(k,e): return in e the element with key k

return false if the operation fails, return true if it succeeds

insert(e): insert element e into the search tree

Delete(k,e): delete the element with key k and return it in e

Ascend(): Output all elements in ascending order of key

}

ADT 1.1 Abstract data type specification of a binary search tree

AbstractDataType *BSTree* {

Instances

Same as for BSTree except that each node has a LeftSize field.

Operations

Create(); create an empty indexed binary search tree

Search(k,e): return in e the element with key k

return false if the operation fails, return true if it succeeds

IndexSearch(k,e): return in e the kth element

insert(e): insert element e into the search tree

Delete(k,e): delete the element with key k and return it in e

Ascend(): Output all elements in ascending order of key

}

ADT 1.2 Abstract data type specification of an indexed binary search tree

1.3.The Class BSTree

Since the number of elements in a binary search tree as well as its shape changes as operations are performed, a binary search tree is represented using the linked

representation. We can greatly simplify the task of developing the class BSTree if we define this class as a derived class of BinaryTree(see Appendix A program A.2) as is done in Program 1.1. Since BSTree is a derived class of BinaryTree, it inherits all members of BinaryTree. However, it has access only to the public and protected members. To access the private member root of BinaryTree, we need to make BSTree a friend of BinaryTree.

```
template<class E, class K>
class BSTree : public BinaryTree<E>  {
  public:
      bool Search(const K& k, E& e) const;
      BSTree<E,K>&  Insert (const E& e)
      BSTree<E,K>&  Delete (const K& k, E& e);
      Void Ascend ()  {InOutput();}
};
```

Program 1.1 Class definition for binary search trees

Notice that the elements of a binary tree can be output in ascending order by invoking the inorder output function InOutput defined in Section 8.9 for binary trees. This function outputs the elements in the left subtree (i.e, smaller elements), then the root element, and finally those in the right subtree (i.e, larger elements). The time complexity is $\Theta(n)$ for an n-element tree.

1.4.Searching

Suppose we wish to search for an element with key k. We begin at the root. If the root is NULL (i.e, zero), the search key contains no elements and the search is unsuccessful. Otherwise, we compare k with the key in the root. If k is less than the key in the root, then no element in the right subtree can have key value k and only the left subtree is to be searched. If k is larger than the key in the root, only the right subtree needs to be searched. The subtrees may be searched similarly. Program 1.2 gives the code. The time complexity is O(h) where h is the height of the tree being searched.

```
template<class E,  class K>
bool BSTree<E,K>::Search (const K& k, E &e) const
(// Search for element that matches k.
   // pointer p starts at the root and moves through the tree looking for an element with
```

```
    //key k
    BinaryTreeNode<E> *p = root;
    while (p)  // examine p->data
       if (k < p->data) p= p->LeftChild;
       else if (k > p->data) p = p->RightChild;
            else (// found element
                    e = p->data;
                    return true;}
    return false;
}
```

Program 1.2 Search a binary search tree

We can perform an indexed search in an indexed binary search tree in a similar manner. Suppose we are looking for the third element in the tree of Figure 1.2(a). The LeftSize field of the root is 4. So the third element is in the left subtree. The root of the left subtree has LeftSize = 2. So the third element is the smallest element in its right subtree. Since the root of this right subtree has LeftSize = 1, the desired element is located hare. The time complexity of an indexed search is also O(h).

1.5.Inserting an Element

To insert a new element e into a binary search tree, we must verify that its key is different from those of existing elements by performing a search for an element with the same key as that of e. If the search is unsuccessful, then the element is inserted at the point the search terminated. For instance, to insert an element with key 80 into the tree of Figure 1.1(b), we first search for 80. This search terminates unsuccessfully, and the last node examined is the one with key 40. The new element is inserted as the right child of this node. The resulting search tree appears in Figure 1.3(a). Figure 1.3(b) shows the result of inserting key 35 into the search tree of figure 1.3(a). Program 1.3 implements this insert strategy.

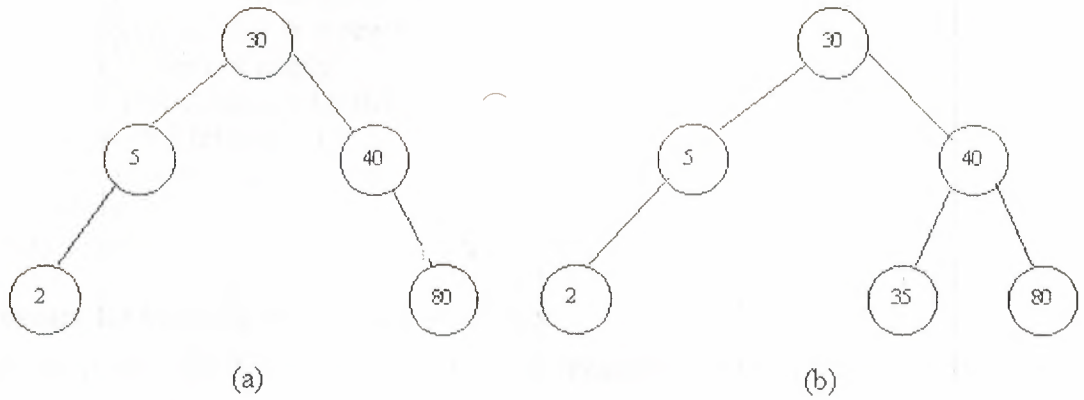


Figure 1.3 Inserting into a binary search tree

When inserting into an indexed binary search tree, we use a procedure similar to Program 1.3. This time, though, we also need to update LeftSize fields on the path from the root to the newly inserted node. Nevertheless, the insertion can be performed in O(h) time where h is the height of the search tree.

1.6.Deleting an Element

For deletion we consider the three possibilities for the node p that contains the element to be deleted: (1) p is a leaf, (2) p has exactly one nonempty subtree, and (3) p has exactly two nonempty subtrees.

Case (1) is handled by discarding the leaf node. To delete 35 from the tree of Figure 1.3(b), the left-child field of its parent is set to zero and the node discarded. The resulting tree appears in Figure 1.3(a). To delete the 80 from this

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Insert(const E& e)
{//Insert e if not duplicate.
   BinaryTreeNode<E> *p = root,   //search pointer
                     *pp = 0;     //parent of p
   //find place to insert
   while (p) {// examine p->data
      pp = p;
     //move p to a child
     if (e < p->data) p= p->LeftChild;
     else if (e > p->data)  p = p->RightChild;
        else throw BadInput();  //duplicate
     }
```

```
    //get a node for e and attach to pp
    BinaryTreeNode<E> *r = new BinaryTreeNode<E> (e);
    if (root) {// tree not empty
       If (e < pp->data) pp->LeftChild = r;
       else pp->RightChild = r;}
    else // insertion into empty tree
       root = r;
    return *this;
}
```

Program 1.3 Inserting into a binary search tree

tree, the right-child field of 40 is set to zero, obtaining the tree of Figure 1.1(b), and the node containing 80 is discarded.

Next consider the case when the element to be deleted is in a node p that has only one nonempty subtree. If p has no parent(i.e, it is the root), node p is discarded and the root of its single subtree becomes the new search tree root. If p has parent pp,then we change the pointer from pp so that it points to p's only child and then delete the node p. For instance, if we wish to delete the element with key 5 from the tree of Figure 1.3(b), we change the left-child field of its parent (i.e, the node containing 30) to point to the node containing the 2.

Finally, to delete an element in a node that has two nonempty subtrees, we replace this element with either the largest element in its left subtree or the smallest element in its right subtree. Suppose we wish to delete the element with key 40 from the tree of Figure 1.4(a). Either the largest element (35) from its left subtree or the smallest (60) from its right subtree can replace this element. If we opt for the smallest element in the right subtree, we move the element with key 60 to the node from which the 40 was deleted, and the leaf from which the 60 is moved is deleted. The resulting tree appears in Figure 1.4(b).

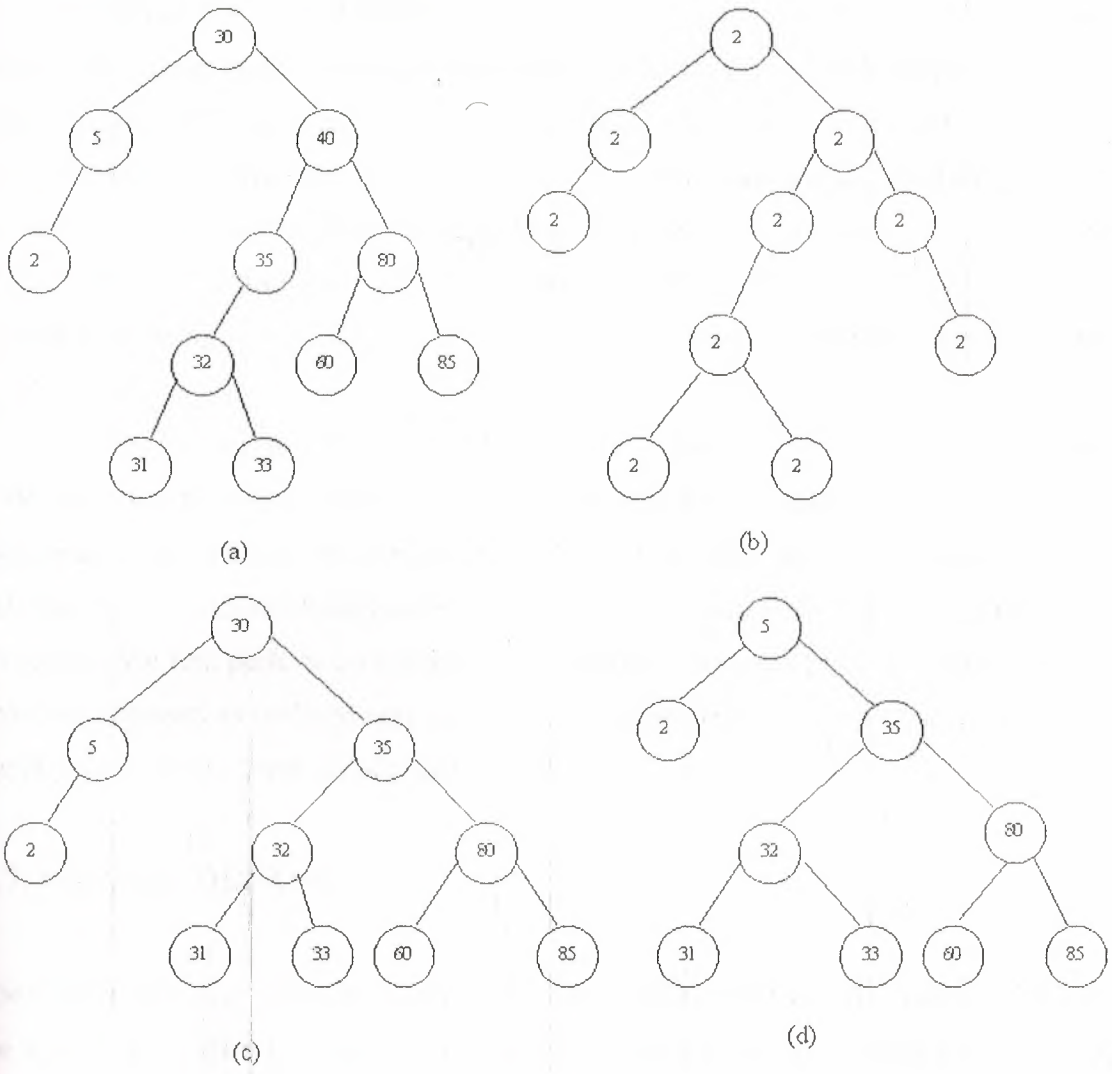


Figure 1.4 Deletion from a binary search tree

Suppose, instead, that when deleting 40 from the search tree of Figure 1.4(a), we had opted for the largest element in the left subtree of 40. This element has key 35 and is in a node of degree 1. we move the element into the node that currently contains 40 and the left-child pointer of this node changes to point to the lone child of the node from which the 35 was moved. The result is shown in Figure 1.4(c).

As another example, consider the deletion of 30 from the tree of Figure 1.4(c). We may replace this element with either the 5 or 31. if we opt for the 5, then since 5 is currently in a node with degree 1, we change the left-child pointer of the parent node to pint to the lone child. The result is the tree of Figure 1.4(d). If we had opted to replace 30 with 31, then since 31 is in a leaf, we need merely delete this leaf.

Notice that the element with smallest key in the right subtree (as well as that with largest key in the right subtree) is guaranteed to be in a node with either zero or one nonempty subtree. Note also that we can find the largest element in the left subtree of a node by moving to the root of that subtree and then following a sequence of right-child pointers until we reach a node whose right-child pointer is 0. Similarly, we can find the smallest element in the right subtree of a node by moving to the root of the right subtree and then following a sequence of left-child pointers until we reach a node whose left-child pointer is 0.

Program 1.4 implements the deletion strategy outlined above. When deleting from a node with two nonempty subtrees, this code always uses the largest element in the left subtree as a replacement. The complexity of this code is O(h). We can perform an indexed deletion from an indexed binary search tree in the same amount of time using an analogous procedure. We first perform an indexed search to locate the element to be deleted. Next we delete the element as outlined here and update LeftSize fields on the path from the root to the physically deleted node as necessary.

1.7.The Class DBSTree

The class for the case when the binary search tree is permitted to contain elements that have the same key is called DBSTree. We can implement this class by changing the while loop of BSTree::Insert (Program 1.3) to that shown in Program 1.5. No other changes are needed.

1.8.Height of Binary Search Tree

Unless care is taken, the height of binary search tree with n elements can become as large as n. The tree height becomes this large, for instance, when Program 1.3 is used to insert elements with keys [1, 2, 3, ... , n], in this order, into an initially empty binary search tree. As a result, a search, insert, or delete operation on a binary search tree takes O(n) time. This time is no better than the times for these operations using unordered chain. However, we can show that when insertions and deletions are made at random using Program 1.3 and

1.4, the height of the binary search tree is O(logn) on the average. As a result, the expected time for each of the search tree operations is O(logn).

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Delete (const K& k, E& e)
{// Delete element with key k and put it in e.
    // set p to point to node with key k
    BinayrTreeNode<E> *p = root,  //search pointer
                      *pp = 0;  //parent of p
    while (p && p->data != k) {//move to a child of p
       pp = p;
       if (k < p->data) p = p->LeftChild;
       else p = p->LeftChild;
       }
    if (!p) throw BadInput();  //no element with key k
    e = p->data;  //save element to delete
    //restructure tree
    //handle case when p has two children
    if (p->LeftChild && p->RightChild) {//two children
       //convert to zero or one child case
       //find largest element in left subtree of p
       BinaryTreeNode<E> *s = p->LeftChild,
                         *ps = p;  //parent of s
       while (s->RightChild) {//move to larger element
          ps = s;
          s = s->RightChild;}
       //move largest from s to p
       p->data = s->data;

       p=s;
       pp = ps;}
    //p has at most on child
    //save child pointer in c
    BinaryTreeNode,E> *c;
    if (p->LeftChild) c = p->LeftChild;
    else c = p->RightChild;
    //delete p
    if (p == root) root = c;
    else {//is p left or right child of pp?
          if (p == pp->LeftChild)
             pp->LeftChild = c;
          else pp->RightChild = c;}
    delete p;
    return *this;
}
```

Program1.4 Deleting from a binary search tree

```
while (p) {
    pp = p;
    if (e <= p->data) p = p->LeftChild;
    else p = p->RightChild;
    }
```

Program 1.5 New while loop for Program 1.3 to permit duplicates

CHAPTER 2
AVL TREES

2.1.Definition

We can guarantee O(log$n$) performance for each search tree operation by ensuring that the search tree height is always O(log$n$). Trees with a worst-case height of O(log$n$) are called **balanced trees.** One of the more popular balanced trees, known as an **AVL tree**, was introduced in 1962 by Adelson-Velskii and Landis.

Definition An empty binary is an AVL tree. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is an AVL tree if(1) T_L and T_R are AVL trees and (2) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively.

An **AVL search tree** is a binary search tree that is also an AVL tree. Trees (a) and (b) of Figure 1.1 are AVL trees, while tree (c) is not. Tree (a) is not an AVL search tree, as its not a binary search tree. Tree (b) is an AVL search tree. The trees of Figure 1.3 are AVL search trees.

An **indexed AVL search tree** is an indexed binary search tree that is also an AVL tree. Both the search trees of Figure 1.2 are indexed AVL search trees. In the remainder of this chapter, we shall not consider indexed AVL search trees explicitly. However, the techniques we develop carry over in a rather straight forward manner to such trees.

If we are to use AVL search trees to represent a dictionary and perform each dictionary operation in logarithmic time, then we must establish the following properties:

1.The height of an AVL tree with n elements/nodes is O(log$n$).

2.For every value of n, $n \geq 0$, there exists an AVL tree. (Otherwise, some insertions cannot leave behind an AVL tree, as no such tree exists for the current number of elements.)

3.An n-element AVL search tree can be searched in O(*height*) = O(log$n$) time.

4.A new element can be inserted into an n element AVL search tree so that in the result is an $n + 1$ element AVL tree and such an insertion can be done in O(log$n$) time.

5.An element can be deleted from an n-element AVL search tree so that the result is an n - 1 element AVL tree and such a deletion can be done in O(log$n$) time.

Property 2 follows from property 4, so we shall not show property 2 explicitly. Properties 1, 2, 3, and 5 are established in the following subsections.

2.2.Height of an AVL Tree

We shall obtain a bound on the height of an AVL tree that has n nodes in it. Let N_h be the minimum number of nodes in an AVL tree of height h. In the worst case the height of one of the subtrees is h - 1 , and the height of the other is h - 2. Both these subtrees are also AVL trees. Hence

$$N_h = N_h\text{-}1 + N_h\text{-}2 + 1,\ N_0 = 0, \text{ and } N_1 = 1$$

Notice the similarity between this definition for N_h and the definition of the Fibonacci numbers.

$$F_n = F_{n-1} + F_{n-2}, F_0 = 0, and\ F_1 = 1$$

From Fibonacci number theory we know that $F_h \approx \Phi^h / \sqrt{5}$ where $\Phi = (1+\sqrt{5})/2$. Hence $N_h \approx \Phi^{h+2} / \sqrt{5} - 1$. If there are n nodes in the tree, then its height h is at most $\log_\Phi(\sqrt{5}(n+1)) - 2 \approx 1.44 \log_2(n+2) = O(\log n)$.

2.3.Representation of an AVL Tree

AVL trees are normally represented using the linked representation using the linked representation scheme for binary trees. However, to facilitate insertion and deletion, a balance factor *bf* is associated with each node. The balance factor *bf(x)* of a node x is defined to be

height of left subtree of x – height of right subtree of x

From the definition of an AVL tree, it follows that the permissible balance factors are -1, 0, and 1. Figure 2.1 shows two AVL search trees and the balance factors for each node.

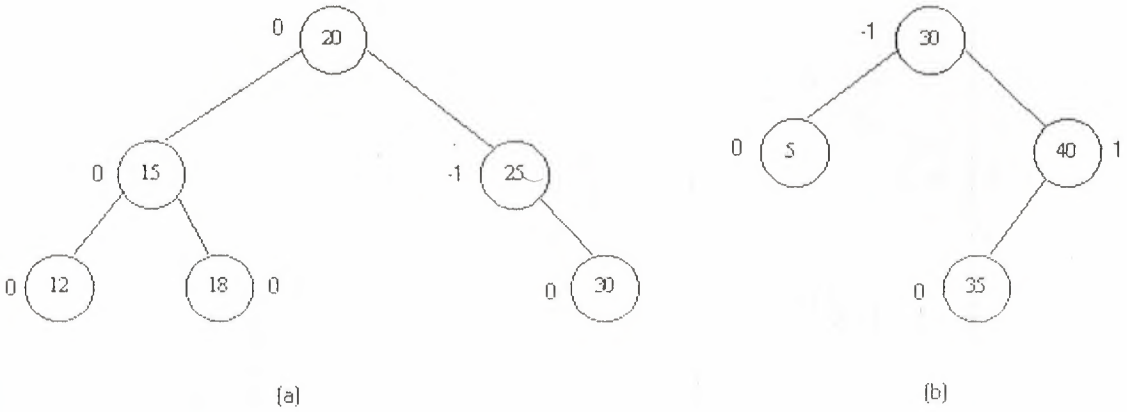


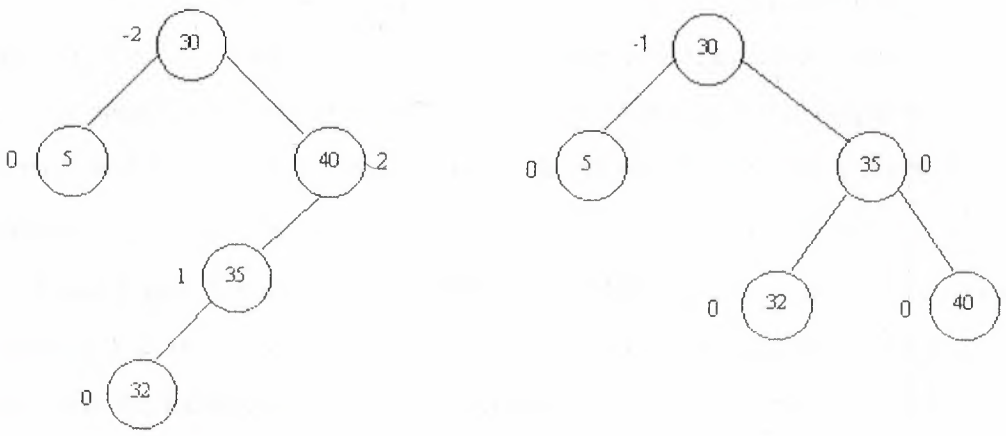
Figure 2.1 AVL search trees

2.4.Searching an AVL Search Tree

To search an AVL search tree, we may use the code of Program 1.2 without change. Since the height of an AVL tree with n elements is O(log*n*), the search time is O(*n*log*n*).

2.5.Inserting into an AVL Search Tree

If we use the strategy of Program 1.3 to insert an element into an AVL search tree, the tree following the insertion may no longer be AVL. For instance, when an element with key 32 is inserted into the AVL tree of Figure 2.1(b), the new search tree is the one shown in Figure 2.2(a). since this tree contains nodes with balance factors other than -1, 0, and 1, it is not an AVL tree. When an insertion into an AVL tree using the strategy of Program 1.3 results in a search tree that has one or more nodes with balance factors other than -1, 0, and 1, the resulting search tree is **unbalanced.** We can restore balance (i.e., make all balance factors -1, 0, and 1) by shifting some of the subtrees of the unbalanced tree as in Figure 2.2(b).



(a)Immediatley after insertion (b)Following rebalancing

Figure 2.2 Sample insertion into an AVL search tree

Before examining the subtree movement needed to restore balance, let us make some observations about the unbalanced tree that results from an insertion.

I1:In the unbalanced tree the balance factors are limited to -2, -1, 0, 1, and 2.

I2:A node with balance factor 2 had a balance factor 1 before the insertion. Similarly, a node with balance factor -2 had a balance factor -1 before the insertion.

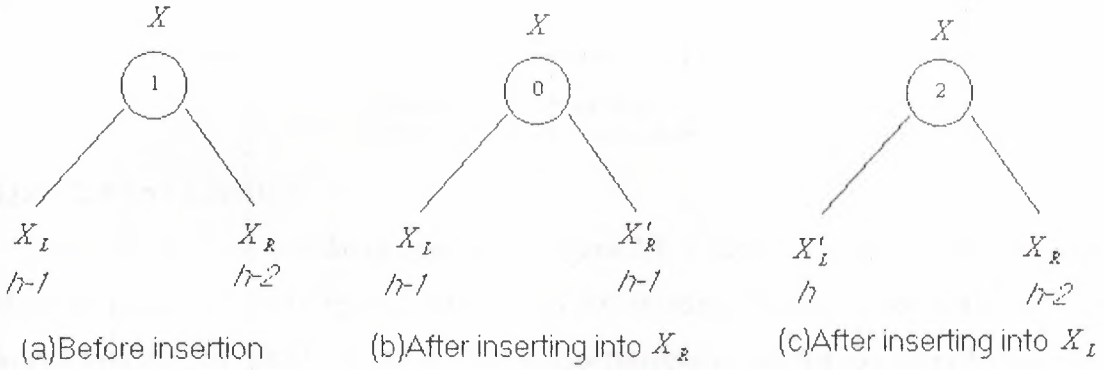
I3:The balance factors of only those nodes on the path from the root to the newly inserted node can change as a result of the insertion.

I4:Let *A* denote the nearest ancestor of the newly inserted node whose balance factor is either -2 or 2. (In the case of Figure 2.2(a), the *A* node is the node with key 40.) The balance factor of all nodes on the path from A to the newly inserted node was 0 prior to the insertion.

Node *A* (see I4) may be identified while we are moving down from the root searching for the place to insert the new element. From I2 it follows that *bf(A)* was either -1 or 1 prior to the insertion. Let *X* denote the last node encountered that has such a balance factor. When inserting 32 into the AVL tree of Figure 2.1(b), *X* is the node with key 40; when inserting 22, 28, or 50 into the AVL tree of Figure 2.1(a), *X* is the node with key 25; and when inserting 10, 14, 16 or 19 into the AVL tree of Figure 2.1(a), there is no node *X*.

When node *X* does not exist, all nodes on the path from the root to the newly inserted node have balance factor 0 prior to the insertion. The tree cannot be unbalanced following the insertion because an insertion changes balance factors by -1, 0, or 1, and only balance

factors on the path from the root may change. Therefore, if the tree is unbalanced following the insertion, X exists. If $bf(X) = 0$ after the insertion, then the height of the subtree with root X is the same before and after the insertion. For example, if this height was h before the insertion and if $bf(X)$ was 1, the height of its left subtree X_L was h-1 and that of its right subtree X_R was h-2 before the insertion must be made in X_R resulting in an X'_R of height h-1 (see Figure 2.3(b)). The height X'_R must increase to h-1 as all balance factors on the path from X to the newly inserted code were 0 prior to the insertion. The height of X remains h and the balance factors of the ancestors of X are the same before and after the insertion, so the tree is balanced.



Balance factor of X is inside the node
Subtree heights are below subtree names.

Figure 2.3 Inserting into an AVL search tree

The only way the tree can become unbalanced is if the insertion causes $bf(X)$ to change from -1 to -2 or from 1 to 2. For the latter case to occur, the insertion must be made in the left subtree X_L of X (see Figure 2.3(c)). Now the height of X'_L must become h (as all balance factors on the path from X to the newly inserted node were 0 prior to the insertion). Therefore, the A node referred to in observation I4 is X.

When the A node has been identified, the imbalance at A can be classified as either an L (the newly inserted node is in the left subtree of A) or R type imbalance. This imbalance classification may be refined by determining which grandchild of A is on the path to the newly inserted node. Notice that such a grandchild exists, as the height of the subtree of A that contains the new node must be at least 2 for the balance factor of A to be -2 or 2. With

this refinement of the imbalance classification, the imbalance at A is of the types LL (new node is in the left subtree of the left subtree of A), LR (new node is in the right subtree of the left subtree of A), RR, and RL.

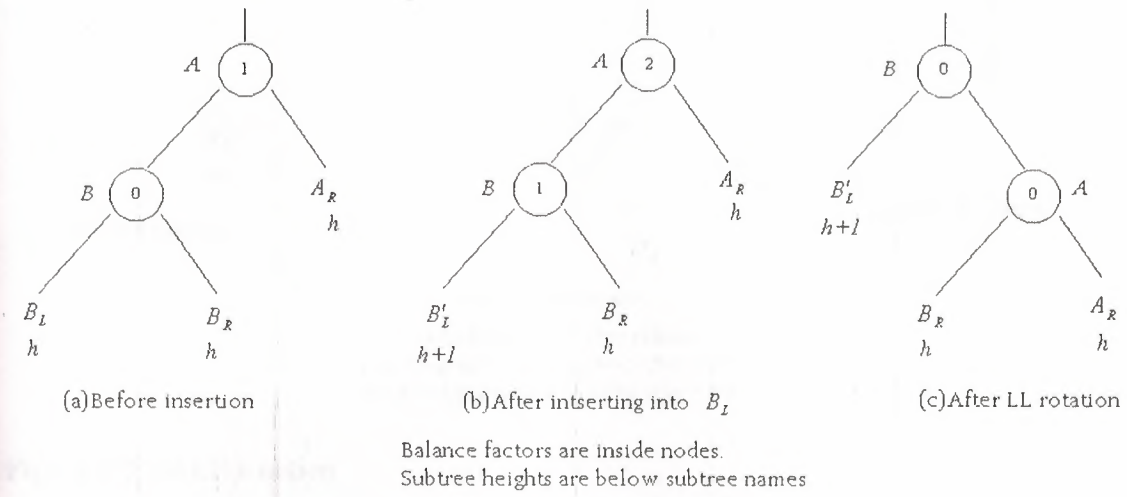


Figure 2.4 An LL rotation

A generic LL type imbalance appears in Figure 2.4. Figure 2.4(a) shows the conditions before the insertion, and Figure 2.4(b) shows the situation following the insertion of an element into the left subtree B_L of B. The subtree movement needed to restore balance at A appears in Figure 2.4(c). B becomes the root of the subtree that A was previously root of, B'_L remains the left subtree of B, A becomes the root of B's right subtree of A, and the right subtree of A is unchanged. The balance factors of nodes in B'_L that are on the path from B to the newly inserted node change as does the balance factor of A. The remaining balance factors are the same as before the rotation. Since the heights of the subtrees of Figures 2.4(a) and (c) are the same, the balance factors of the ancestors (if any) of this subtree are the same as before the insertion. So no nodes with a balance factor other than -1, 0, or 1 remain. A single LL rotation has rebalanced the entire tree! You may verify that the rebalanced tree is indeed a binary search tree.

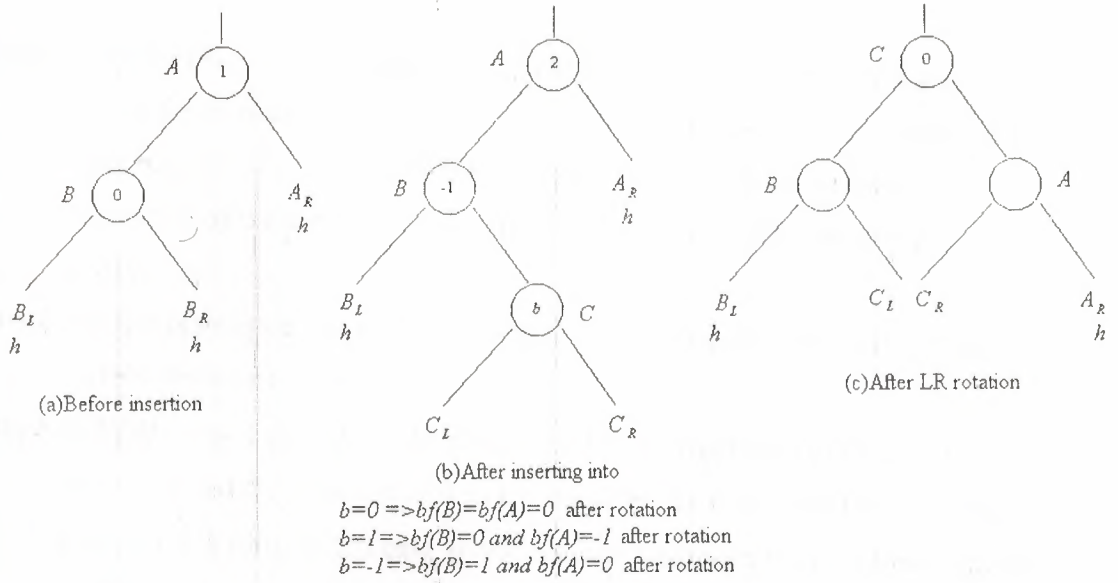


Figure 2.5 An LR rotation

Figure 2.5 shows a generic LR type imbalance. Since the insertion took place in the right subtree of B, this subtree cannot be empty following the insertion; therefore, node C exists. However, its subtrees C_L and C_R may be empty. The rearrangement of subtrees needed to rebalance appears in Figure 2.5(c). The values *bf(B)* and *bf(A)* following the rearrangement depend on the value, b, of *bf(C)* just after the insertion but before the rearrangement. The figure gives these values as a function of b. The rearranged subtree is seen to be a binary search tree. Also, since the heights of the subtrees of Figures 2.5(a) and (c) are the same, the balance factors of their ancestors (if any) are the same before and after the insertion. So a single LR rotation at A rebalances the entire tree.

The cases RR and RL are symmetric to the ones we just seen. The transformations done to remedy LL and RR imbalances are often called **single rotations**, while those done for LR and RL imbalances are called **double rotations**. The transformations for an LR imbalance can be viewed as an RR rotation followed by an LL rotation, while that for an RL imbalance can be viewed as an LL rotation followed by an RR rotation.

The steps in the AVL search-tree-insertion algorithm that results from our discussion appear in Figure 2.6. These steps can be refined into C++ code that has a complexity of O(*height*) = O(log*n*). *Notice that a single rotation is sufficient to restore balance if the insertion causes imbalance.*

Step1: Find the place to insert the new element by following a path from the root as in a search for an element with the same key. During this process, keep track of the most recently seen node with balance factor -1 or 1. Let this node be A. If an element with the same key is found, the insert fails and the remaining steps are not performed.

Step2: If there is no node A, then make another pass from the root, updating balance factors. Terminate following this pass.

Step3: If $bf(A) = 1$ and the new node was inserted in the right subtree of A or if $bf(A) = -1$ and the insertion took place in the left subtree, then the new balance factor of A is zero. In this case update balance factors on the path from A to the new node and terminate.

Step4: Classify the imbalance at A and perform the appropriate rotation. Change balance Factors as required by the rotation as well as those of nodes on the path from the new subtree root to the newly inserted node.

Figure 2.6 Steps for AVL search tee insertion

2.6.Deletion from an AVL Search Tree

To delete an element from an AVL search tree, we proceed as in Program 1.4. Let q be the parent of the node that was physically deleted. For example, if the element with key 25 is deleted from the tree of Figure 2.1(a), the node containing this element is deleted and the right-child pointer from the root diverted to the only child of the deleted node. The parent of the deleted node is the root, so q is the root. If instead, the element with key 15 is deleted, its spot is used by the element with key 12 and the node previously containing this element deleted. Now q is the node that originally contained 15(i.e., the left child of the root). Since the balance factors of some (or all) of the nodes on the path from the root to q have changed as a result of the deletion, we retrace this path backwards from q towards the root.

If the deletion took place from the left subtree of q, $bf(q)$ decreases by 1, and if it took place from the right subtree, $bf(q)$ increases by 1. We may make the following observations.

D1:If the new balance factor of q is 0, its height has decreased by 1, and we need to change the balance factor of its parent (if any) and possibly those of its other ancestors.

D2:If the new balance factor of q is either -1 or 1, its height is the same as before the deletion and the balance factors of its ancestors are unchanged.

D3:If the new balance factor of q is either -2 or 2, the tree is unbalanced at q.

Since balance factor changes may propagate up the tree along the path from q to the root (see observation D2), it is possible for the balance factor of a node on this path to become -2 or 2. Let A be the first such node on this path. To restore balance at node A, we classify the type of imbalance. The imbalance is of type L if the deletion took place from A's left subtree. Otherwise, it is of type R. If $bf(A) = 2$ after the deletion, it must have been 1 before. So A has left subtree with root B. A type R imbalance is sub classified into the types R0, R1, and R-1 depending on $bf(B)$. The type R-1, for instance, refers to the case when the deletion took place from the right subtree of A and $bf(B) = -1$. In a similar manner type L imbalances are sub classified into types L0, L1, and L-1.

An R0 imbalance at A is rectified by performing the rotation shown in Figure 2.7. Notice that the height of the shown subtree was h+2 before the deletion and is h+2 after. So the balance factors of the remaining nodes on the path to the root are unchanged. As a result, the entire tree has been rebalanced.

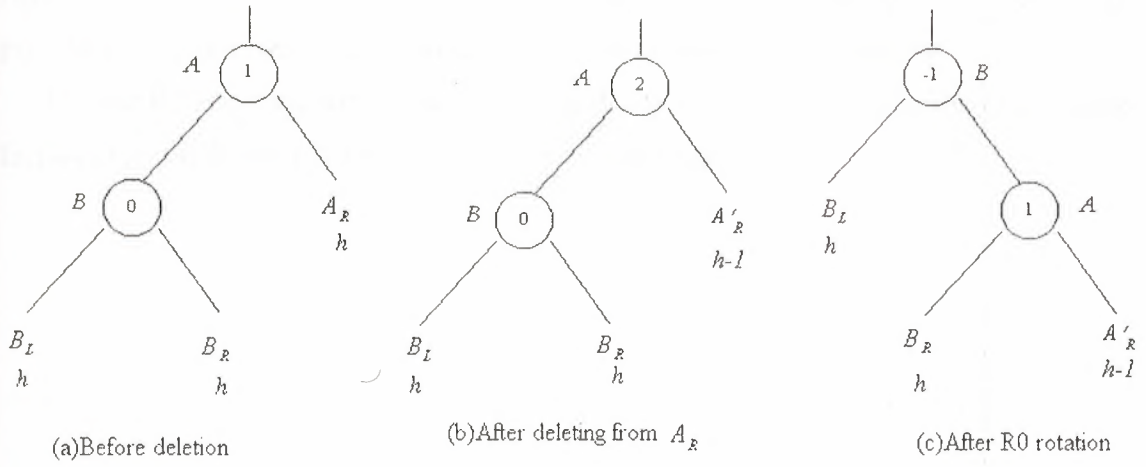


Figure 2.7 An R0 rotation (single rotation)

Figure 2.8 shows how to handle an R1 imbalance. While the pointer changes are the same as for an R0 imbalance, the new balance factors for A and B are different and the height of the subtree following the rotation is now h+1, which is one less than before the deletion. So if A is not the root, the balance factors of some of its ancestors will change and further rotations may be necessary. Following an R1 rotation, we must continue to examine nodes on the path to the root. *Unlike the case of an insertion, one rotation may not suffice to restore balance following a deletion. The number of rotations needed is* O(log*n*).

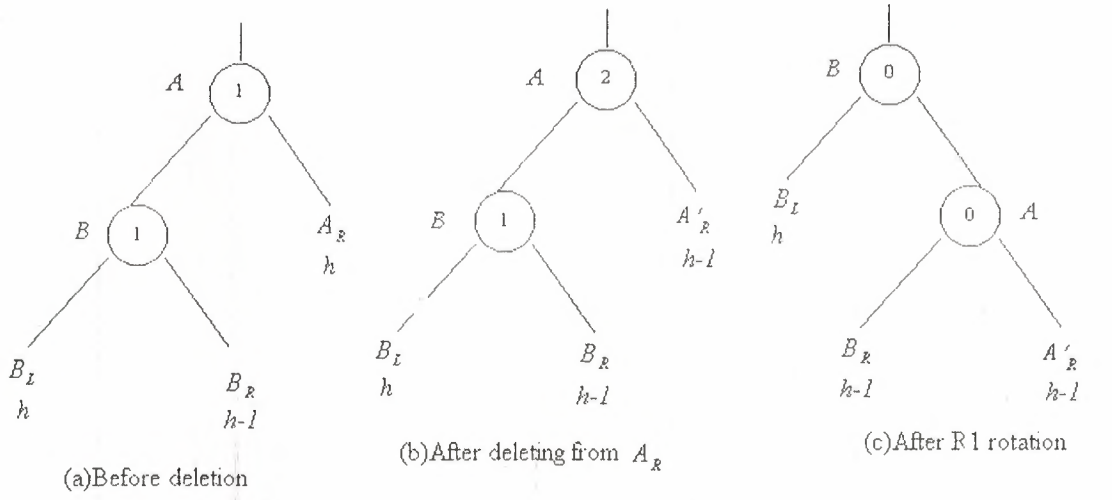


Figure 2.8 An R1 rotation (single rotation)

The transformation needed when the imbalance is of type R-1 appears in Figure 2.10. The balance factors of A and B following the rotation depend on the balance factor b of the

right of *B*. This rotation leaves behind a subtree of height h+1, while the subtree height prior to the deletion was h+2. So we need to continue on the path to the root.

LL and R1 rotations are identical; LL and R0 rotations differ only in the final balance factors of *A* and *B*; and LR and R-1 rotations are identical.

CHAPTER 3
RED-BLACK TREE

3.1.Definition

A **red-black tree** is a binary search tree in which every node is colored either red or black. The remaining properties satisfied by a red-black tree are best stated in terms of the corresponding extended binary tree. We obtain an extended binary tree from a regular binary tree by replacing every null pointer with an external node. The additional properties are

RB1. The root and all external nodes are colored black.

RB2. No root-to-external-node path has two consecutive red nodes.

RB3. All root-to-external-node paths have the same number of black nodes.

An equivalent definition arises from assigning colors to the pointers between a node and its children. The pointer from a parent to a black child is black and to a red child is red. Additionally,

RB1′. Pointers from an internal node to an external node are black.

RB2′. No root-to-external-node path has two consecutive red pointers.

RB3′. All root-to-external-nodes paths have the same number of black pointers.

Notice that is we know the pointer colors, we can deduce the node colors and vice-versa. In the red-black tree of Figure 3.1, the external nodes are solid squares, black pointers are think lines, and red pointers are thin lines. The colors of the nodes may be deduced from the pointer colors and property RB1. The nodes with 5, 50, 62, and 70 are red, as they have red pointers from their parents. The remaining nodes are black. Notice that every path from the root to an external node has exactly two black pointers and three black nodes (including the root and the external node); no such path has two consecutive red nodes or pointers.

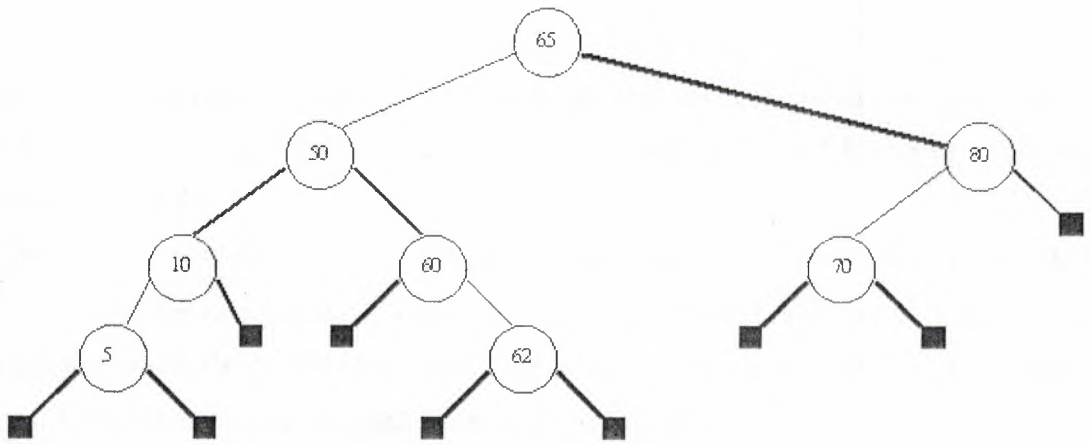


Figure 3.1 A red-black tree

Let the **rank** of a node in a red-black tree be the number of black pointers on any path from the node to any external node in its subtree. So the rank of an external node is zero. The rank of the root of Figure 3.1 is 2, that of its left child is 2, and of its right child is 1.

Lemma 3.1 Let the length of a root-to-external-node path be the number of pointers on the path. If P and Q are two root-to-external-node paths in a red-black tree, then $length(P) \leq 2length(Q)$.

Proof Consider any red-black tree. Suppose that the rank of the root is r. From RB1′ the last pointer on each root-to-external-node path is black. From RB2′ no such path has two consecutive red pointers. So each red pointer is followed by a black pointer. As a result, each root-to-external-node path has between r and $2r$ pointers, so $length(P) \leq 2length(Q)$. To see that upper bound is possible, consider the red-black tree of Figure 3.1. The path from the root to the left child of 5 has length 4, while that to the right child of 80 has length 2.

Lemma 3.2 Let h be the height of a red-black tree (excluding the external nodes), let n be the number of internal nodes in the tree, and let r be the rank of the root.

(a) $h \leq 2r$

(b) $n \geq 2^r - 1$

(c) $h \leq 2\log_2(n+1)$

Proof From the proof of Lemma 3.1, we know that no root-to-external-node path has length > $2r$, so $h \leq 2r$. (The height of the red-black tree of Figure 3.1 with external nodes removed is $2r = 4$.)

Since the rank of the root is r, there are no external nodes at levels 1 through r, so there are $2^r - 1$ internal nodes at these levels. Consequently, the total number of internal nodes is at least this much. (In the red-black tree of Figure 3.1, levels 1 and 2 have $3 = 2^2 - 1$ internal nodes. There are additional internal nodes at levels 3 and 4.)

From (b) it follows that $r \leq \log_2(n+1)$. This inequality together with (a) yields (c).

Since the height of a red-black tree is at most $2\log_2(n+1)$, search, insert, and delete algorithms that work in O(h) time have complexity O(log$n$).

Notice that the worst-case height of a red-black tree is more that the worst-case height (approximately $1.44\log_2(n+2)$) of an AVL tree with the same number of (internal) nodes.

3.2.Representation of a Red-Black Tree

Although it is convenient to include external nodes when defining red-black trees, in an implementation, null or zero pointers, rather than physical nodes, represent these codes. Further, since pointer and node colors are closely related, with each node we need to store only its color or the color of the two pointers to its children. Node colors require just one additional bit per node, while pointer colors require two. Since both schemes require almost the same amount of space, we may choose between them on the basis of actual run times of the resulting red-black tree algorithms.

In our discussion of the insert and delete operations, we shall explicitly state the needed color changes only for the nodes. The corresponding pointer color changes may be inferred.

3.3.Searching a Red-Black Tree

We can search a red-black tree with the code we used to search an ordinary binary search tree (Program 1.2). This code has complexity O(h), which is O(log$n$) for a red-black tree. Since we use the same code to search ordinary binary search trees, AVL trees, and red-black trees and since the worst-case height of an AVL tree is least, we expect AVL trees to show the best worst-case performance in applications where search is dominant operation.

3.4.Inserting into Red-Black Tree

Elements may be inserted using the strategy used for ordinary binary trees (Program 1.3). When the new node is attached to the red-black tree, we need to assign it a color. If the tree was empty before the insertion, then the new node is the root and must be colored black (see property RB1). Suppose the tree was not empty prior to the insertion. If the new node is given the color black, then we will have an extra black node on paths from the root to the external nodes that are children of the new node. On the other hand, if the new node is assigned the color red, then we might have two consecutive red nodes. Making the new node black is guaranteed to cause a violation of property RB3, while making the new node red may or may not violate property RB2. We shall make the new node red.

If making the new node red causes a violation of property RB2, we shall say that the tree has become imbalanced. The nature of the imbalance is classified by examining the new node u, its parent pu, and the grandparent gu of u. Observe that since property RB2 has been violated, we have two consecutive red nodes. One of these red nodes is u, and other must be its parent; therefore, pu exists. Since pu is red, it cannot be the root (as the root is black by property RB1); u must have a grandparent, gu, which must be black (property RB2). When pu is the left child of gu, u is the left child of pu, and the other child of gu is black (this case includes the case when the other child of gu is an external node), the imbalance is of type LLb. The other imbalance types are LLr (pu is the left child of gu, u is the left child of pu, the other child of gu is red), LRb (pu is the left child of gu, u is the right child of pu, the other child of gu is black), LLb, LRr, RRb, RRr, RLb, and RLr.

Imbalances of type XYr (X and Y may be L or R) are handled by changing colors, while those of type XYb require a rotation. When we change a color, the RB2 voilation may propagate two levels up the tree. In this case we will need to reclassify at the new level, with the new *u* being the former *gu*, and apply the transformations again. When a rotation is done, the RB2 violation is taken care of, and no further work is needed.

Figure 3.2 shows the color changes performed for LLr and LRr imbalances; these color changes are identical. Black nodes are shaded dark, while red ones are shaded light. In Figure 3.2(a), for example, *gu* is black, while *pu* and *u* are red; the pointers from *gu* to its left and right children are red; gu_R is the right subtree of *gu*; and pu_R is the right subtree of *pu*. Both LLr and LRr color changes require us to change the color of *pu* and of the right child of *gu* from red to black. Additionally, we change the color of *gu* from black to red provided *gu* is not the root. Since this color change is not done when *gu* is the root, the number of black nodes on all root-to-external-node paths increases by one when *gu* is the root of the red-black tree.

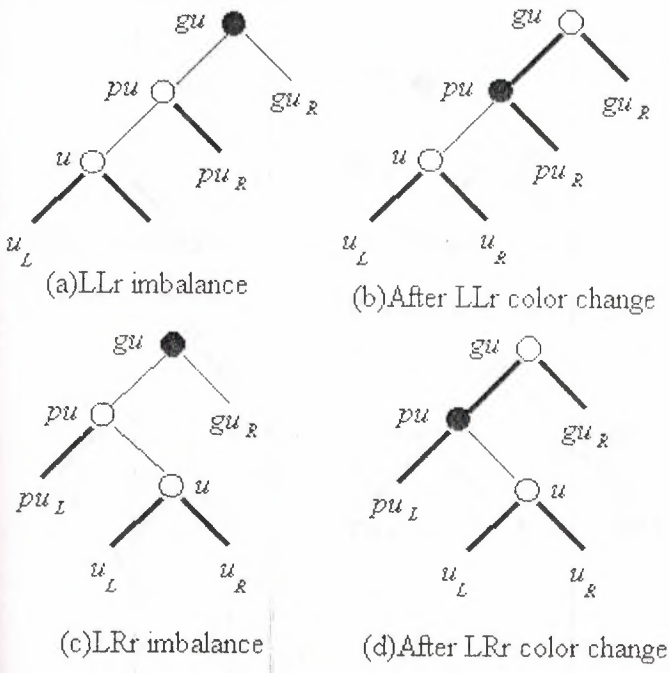


Figure 3.2 LLr and LRr color changes

If changing the color of *gu* to red causes an imbalance, *gu* becomes the new *u* node, its parent becomes the new *pu*, and its grandparent becomes the new *gu* and we continue to

rebalance. If *gu* is the root or if the color change does not cause an RB2 violation at *gu*, we are done.

Figure 3.3 shows the rotations performed to handle LLb and LRb imbalances. In Figures 3.3(a) and (b), *u* is the root of pu_L. Notice the similarity between these rotations and the LL (refer to Figure 2.4) and LR (refer to Figure 2.5) rotations used to handle an imbalance following an insertion in an AVL tree. The pointer changes are the same. In the case of an LLb rotation, for example, in addition to pointer changes, we need to change the color of *gu* from black to red and of *pu* from red to black.

In examining the node (or pointer) colors after the rotations of Figure 3.3, we see that the number of black nodes (or pointers) on all root-to-external-node paths is unchanged. Further, the root of the involved subtree (*gu* before the rotation and *pu* after) is black following the rotation; therefore, two consecutive red nodes cannot exist on the path from the tree root to the new *pu*. Consequently, no additional rebalancing work is to be done. *A single rotation (preceded possibly by O(logn) color changes) suffices to restore balance following an insertion!*

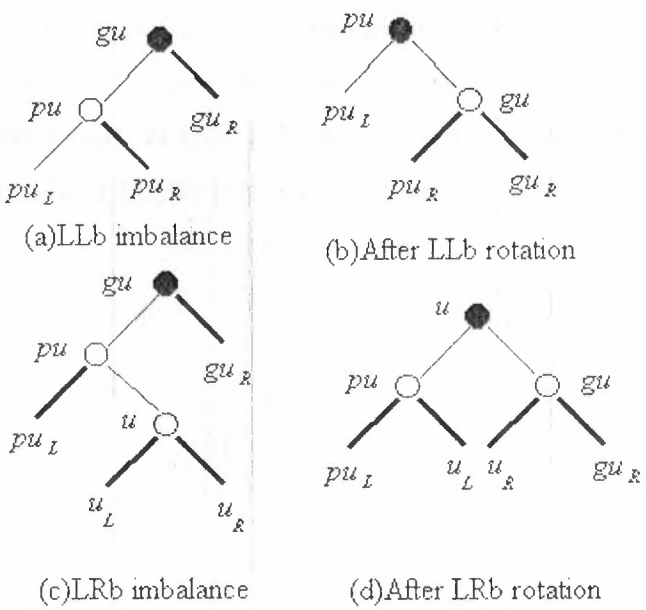


Figure 3.3 LLb and LRb rotations for red-black insertion

Example 3.1 Consider the red-black tree of figure 3.4(a). This figure shows only pointer colors; node colors may be inferred from the pointer colors and the knowledge that the root

color is always black. External nodes are shown for convenience. In an actual implementation, the shown black pointers to external nodes are simple null ore zero, and external nodes are not represented. Notice that all root-to-external-node paths have two black pointers.

To insert 70 into this red-black tree, we use the algorithm of Program 1.3. The new node is added as the left child of 80. Since the insertion is done into a nonempty tree, the new node is assigned the color red. So the pointer to it from its parent (80) is also red. This insertion does not result in a violation of property RB2, and no remedial action is necessary. Notice that the number of black pointers on all root-to-external-node paths is the same as before the insertion.

Next, insert 60 into the tree of Figure 3.4(b). The algorithm of Program 1.3 attaches a new node as the left child of 70 as in Figure 3.4(c). The new node is red, and the pointer to it is also red. The new node is the *u* node, its parent (70) is *pu*, and its grandparent (80) is *gu*. Since *pu* and *u* are red, we have an imbalance. This imbalance is classified as an LLr imbalance (as *pu* is the left child of *gu*, *u* is the left child of *pu*, and the other child of *gu* is red). When the LLr color change of Figure 3.2(a) and (b) is performed, we get the tree of Figure 3.4(d). Now *u*, *pu*, and *gu* are each moved two levels up the tree. The node with 80 is the new *u* node, the root becomes *pu*, and *gu* is zero. Since there is no *gu* node, we cannot have an RB2 imbalance at this location and we are done. All root-to-external-node paths have exactly two black pointers.

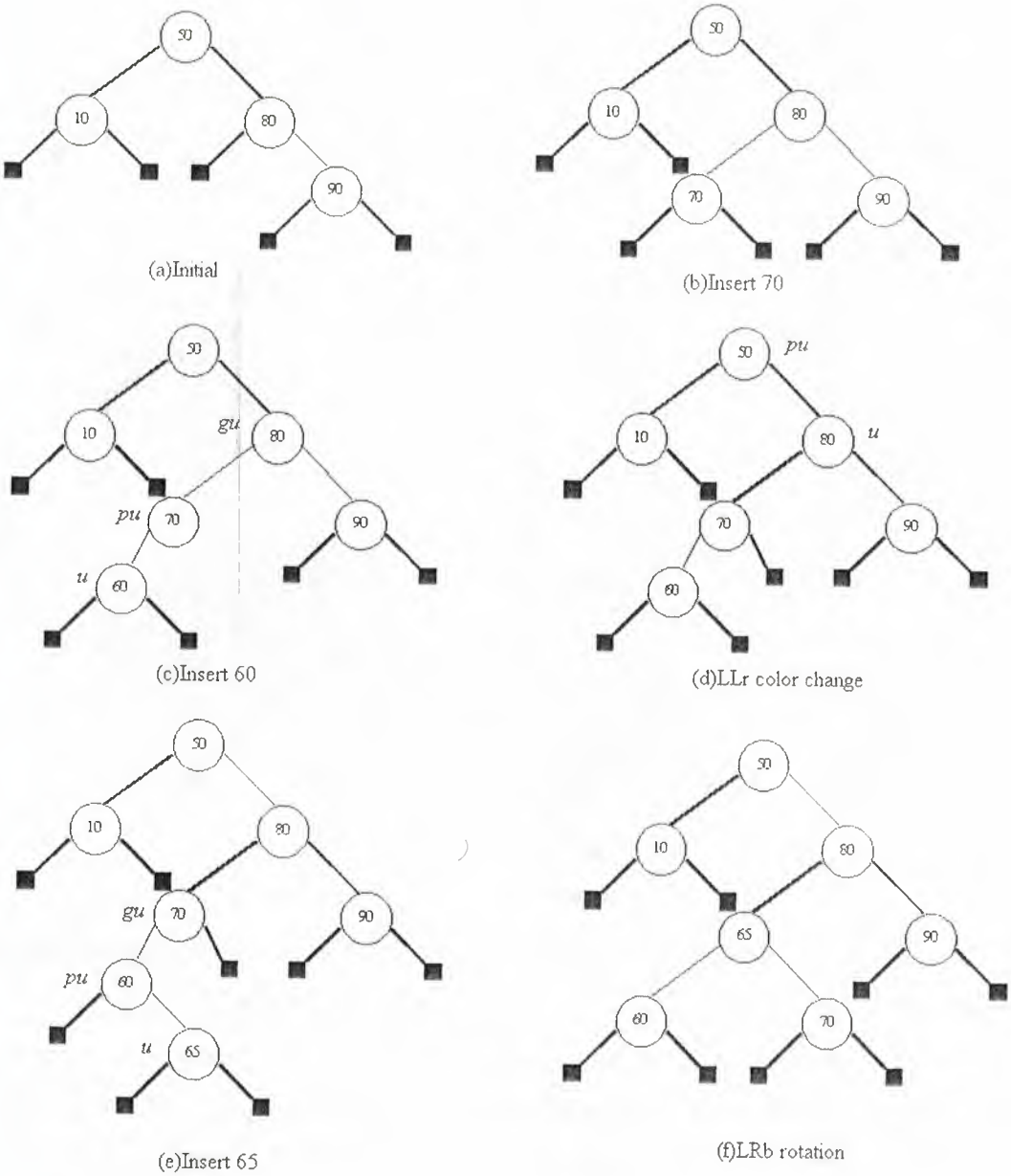
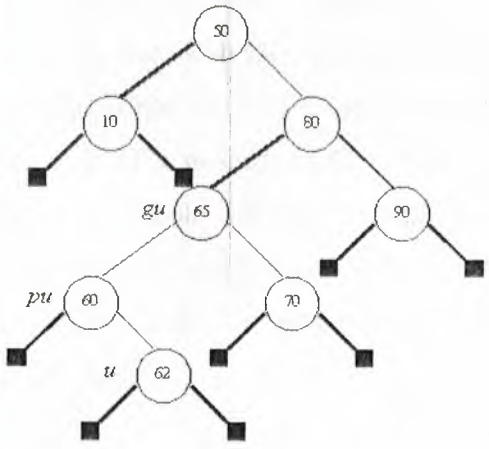
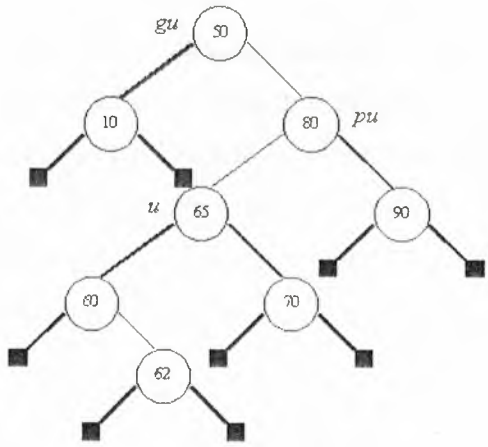


Figure 3.4 Insertion into a red-black tree(continues)

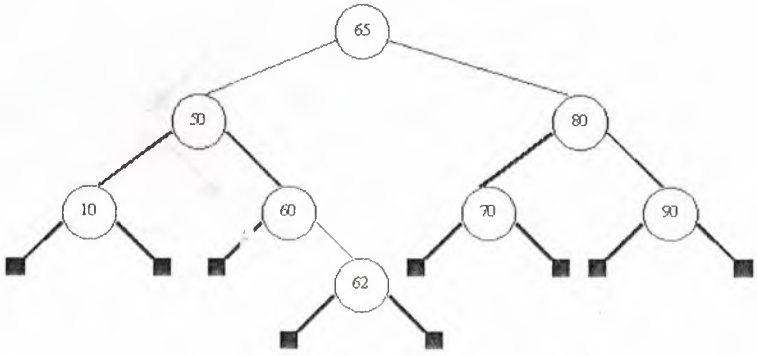
Now insert 65 into the tree of Figure 3.4(d). The result appears in Figure 3.4(e). The new node is the *u* node. Its parent and grandparent are, respectively, the *pu* and *gu* nodes. We have an LRb imbalance that requires us to perform the rotation of Figure 3.3(c) and (d). The result is the tree of Figure 3.4(f).



(g)Insert 62



(h)LRr color change



(i)RLb rotation

Figure 3.4 Insertion into a red-black tree (concluded

Finally, insert 62 to obtain the tree of Figure 3.4(g). We have an LRr imbalance that requires a color change. The resulting tree and the new *u*, *pu*, and *gu* nodes appear in Figure 3.4(h). The color change just performed has caused an RLb imbalance two levels up, we now need to perform an RLb rotation. The rotation results in the tree of Figure 3.4(i). Following a rotation, no further work is needed and we are done.

3.5.Deletion from a Red-Black Tree

Deletion are performed by first using the deletion algorithm for ordinary binary search trees (Program 1.4) and then performing remedial color changes and a single rotation if necessary. Consider the red-black tree of Figure 3.5(a). If Program 1.4 is used to delete 70,

we get the tree of Figure 3.5(b). (If pointer colors are represented, we will also need to change the color of 90's left pointer to get this tree). When 90 is deleted from tree(a), tree(c) results. (If pointer colors are used, the right-pointer color of 65 will need to be changed to get this tree.) The deletion of 65 from tree(a) results in tree (d). (Again, if pointer colors are used, a pointer-color change is needed.) Let y denote the node that takes the place of the physically deleted node. The y nodes for the deletion examples appear in Figure 3.5. In the case of Figure 3.5(b), for example, the left child of 90 was deleted. Its new left child is the external node y.

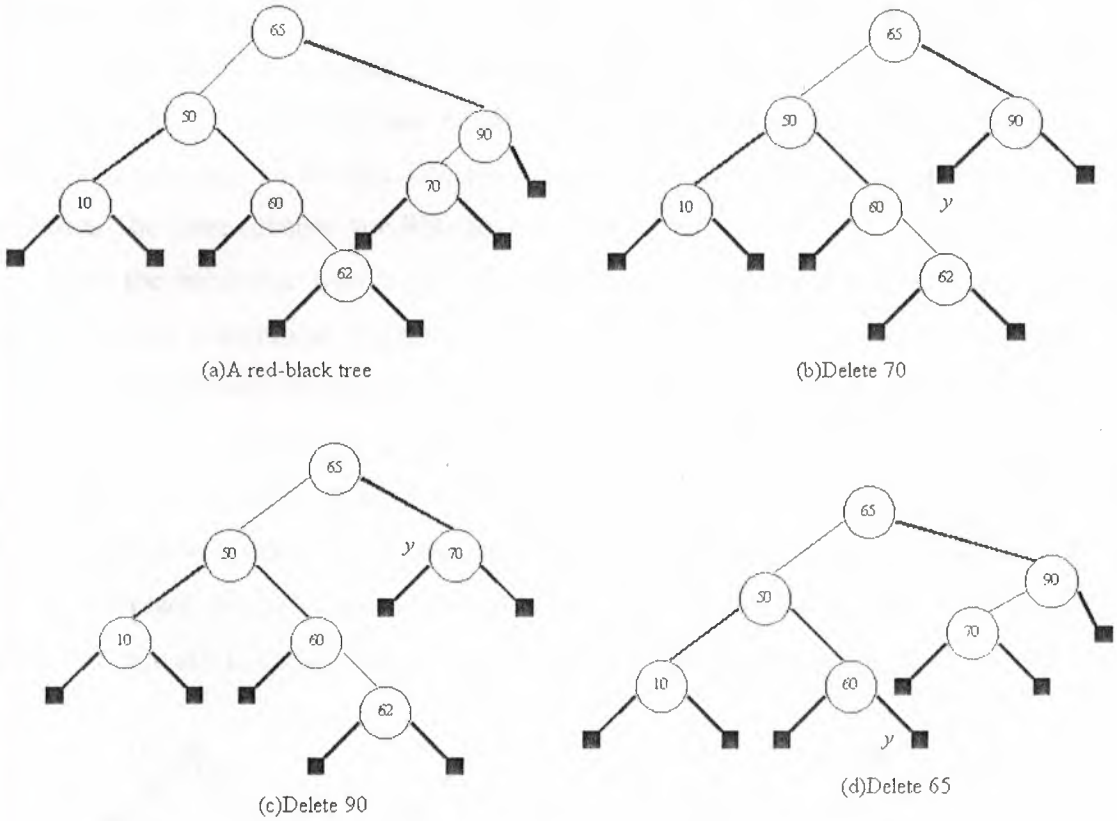


Figure 3.5 Red-black deletion examples

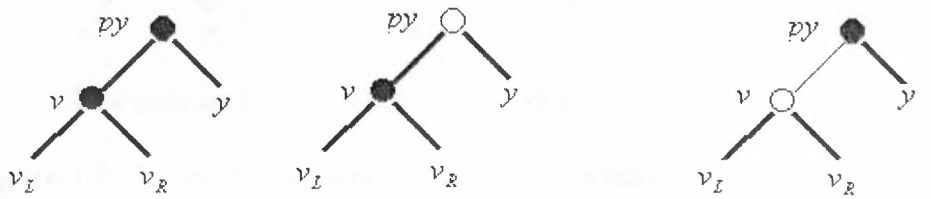
In the case of tree(b), the deleted node (i.e., the one that contained 70 in tree (a)) was red. Its deletion does not affect the number of black nodes on root-to-external-node paths, and no remedial work is necessary. In tree (c) the deleted node (i.e., the one with 90 in tree (a)) was black, and the number of black nodes (and hence pointers) on paths from the root-to-external-nodes in y is one less than before. Since y is not the new root, an RB3 violation occurs. In tree (d) the deleted node was red, and no RB3 violation occurs. *An RB3 violation*

occurs only when the deleted node was black and y is not the root of the resulting tree. No other red-black property violations are possible following a deletion using Program 1.4.

When an RB3 violation occurs, the subtree rooted at y is one black node (or equivalently, one black pointer) deficient; therefore, the number of black nodes (and hence pointers) on paths from the root to external nodes in the subtree y is one less than on paths to other external nodes. We shall say that the tree has become **unbalanced**. We classify the nature of the imbalance by identifying the parent py and sibling v of y. When y is the right child of py, the imbalance is of type R. Otherwise, it is of type L. Observe that since y is one black node deficient, v cannot b an external node. If v is a black node, the imbalance is of type Lb or Rb. When v is red, the imbalance is of type Lr or Rr.

First, consider an Rb imbalance. Imbalances of type Lb are handled in a similar way. Rb imbalances may be divided into three subcases on the basis of the number of v's red children. The three subcases are Rb0, Rb1, and Rb2.

When the imbalance type is Rb0, a color change is performed (Figure 3.6). Figure 3.6 shows the two possibilities for the color of py. If py was black prior to the color change, then the color change causes the subtree rooted at py to be one black node deficient. Also, in Figure 3.6(b) the number of black nodes on paths to external nodes in v is one less than before the color change. Therefore, regardless of whether the path goes to an external node in v or y, following the color change, it is one black node deficient. If py is the root of the whole red-black tree, nothing more is to be done. If it is not, then py becomes the new y; the imbalance at y is reclassified, and appropriate remedial action occurs at this new y.



(a)Rb0 imbalance (b)Rb0 color change

Figure 3.6 Rb0 color change for red-black deletion

When py was red before the color change, the number of black nodes on paths to external nodes in y increases by one but is unchanged for those in v. The entire tree becomes balanced, and we are done.

Rotations are performed when the imbalance type is Rb1 or Rb2. These rotations appear in Figure 3.7. An unshaded node denotes a node that may be either red or black. The color of such node is not changed as a result of the rotation. Therefore, in Figure 3.7(b) the root of the shown subtree has the same color before and after the rotation-the color of v in (b) is the same as that of py in (a). You should verify that following the rotation the number of black nodes (and hence black pointers) on paths from the root to the remaining external nodes. As a result, a rotation rebalances the tree, and no further work is to be done.

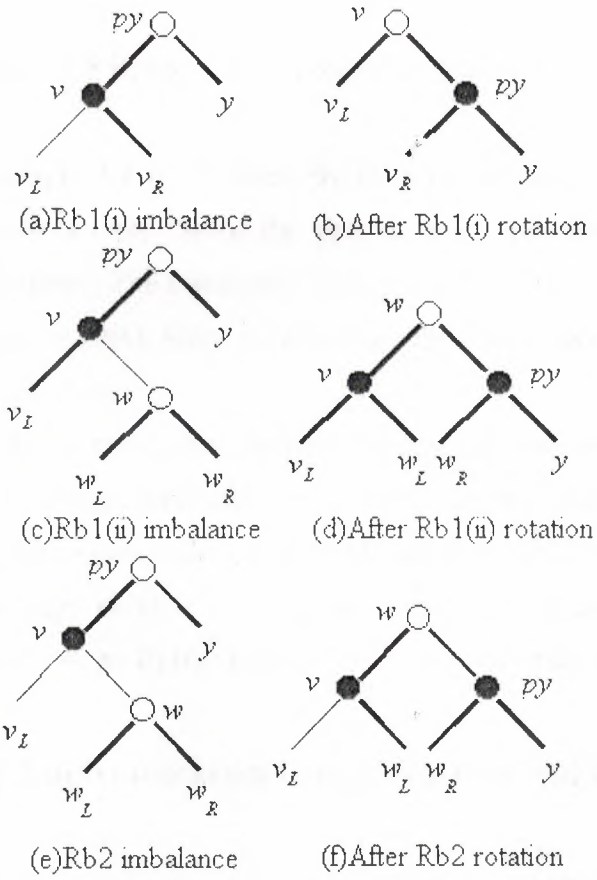
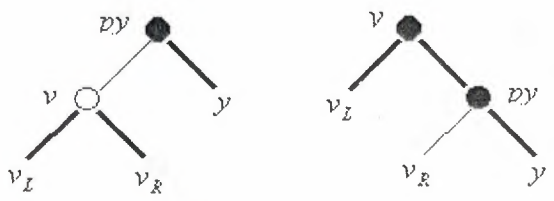


Figure 3.7 Rb1 and Rb2 rotations for red-black deletion

Next, consider imbalances of type Rr. The case of Lr imbalances is symmetric. Since y is one black node deficient and v is red, both v_L and v_R have at least one black node that is not an external node; therefore, both children of v are internal nodes. Rr imbalances may be subdivided into three cases according to the number of red children (0, 1, or 2) that v's right child has. All three cases of an Rr imbalance are handled by a rotation. The rotations appear

in Figures 3.8 and 3.9. Once again, you can verify that the shown rotation restores balance to the entire tree.



(a)Rr0 imbalance (b)Rr0 rotation

Figure 3.8 Rr0 rotation for red-black deletion

Example 3.2 If we delete 90 from the red-black tree of Figure 3.4(i), we get the tree of Figure 3.10(a). Since the deleted node was not the root and was black, we have an imbalance. The imbalance is of type Rb0, and a color change is performed to get the tree of Figure 3.10(b). Since *py* was originally a red node, this color change rebalances the tree and we are done.

If we now delete 80 from the tree (b), tree (c) results. A red node was deleted, so the tree remains balanced. When 70 is deleted from tree (c), we get tree (d). This time a nonroot black node was deleted, and the tree is unbalanced. The imbalance type is Rr1(ii) (the right child *w* of *v* has one red pointer, which is itself the right-child pointer of *w*). Following an Rr1(ii) rotation, tree (e) is obtained. This tree is balanced.

3.6.Implementation Considerations and Complexity

The remedial action taken to rebalance a red-black tree following an insertion or deletion requires us to move back on the path taken from the root to the point of insertion or deletion. This backward movement is easy to do if each node has a parent field in addition to data, left child, and color fields. An alternative to adding a parent field to each node is to save, on a stack, pointers to nodes encountered on the downward path from the root to the point of insertion/deletion. Now we may move back toward the root by performing deletes from the stack of saved pointers. For an *n*-element red-black tree, the addition of parent fields increases the space requirements by $\Theta(n)$, while the use of a stack increases the space

requirements by $\Theta(\log n)$. Although the stack scheme is more efficient on space requirements, the parent-pointer scheme runs slightly faster.

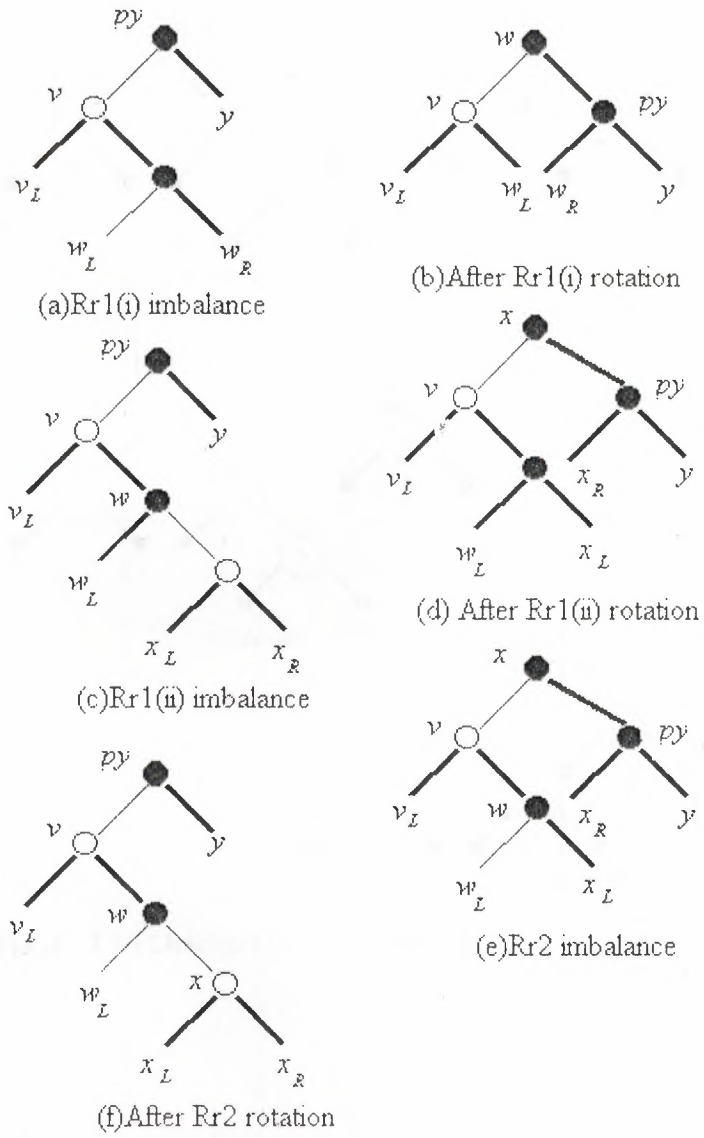


Figure 3.9 Rr1 and Rr2 rotations for red-black deletion

Since the color changes performed following an insert or delete may propagate back towards the root, O(log*n*) of these color changes may be performed. Rotations, on the other hand, guarantee to rebalance the tree. As a result, at most one rotation may be performed following each insert/delete. The time needed for each color change or rotation operation is $\Theta(1)$, the total time needed to insert/delete is O(log*n*).

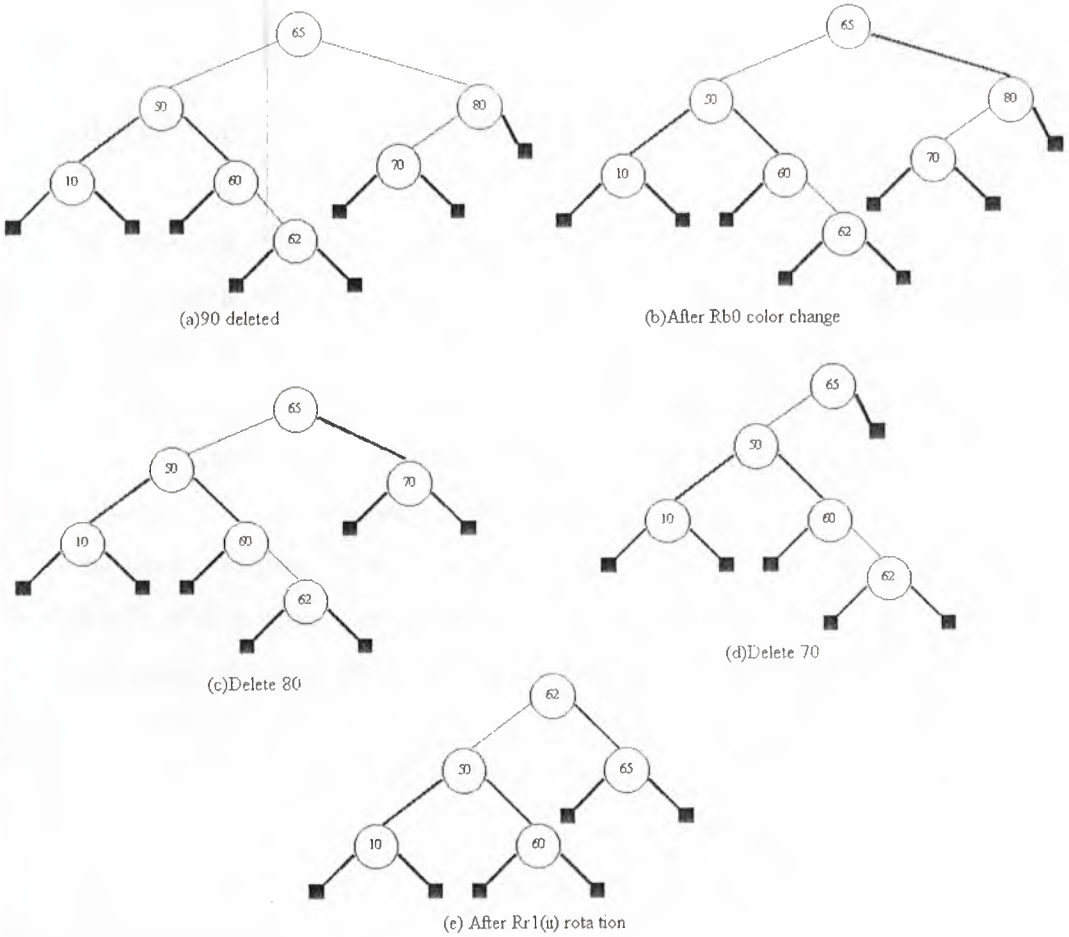


Figure 3.10 Deletion from a red-black tree

CHAPTER 4
B-TREES

4.1.Indexed Sequential Access Method

AVL and red-black trees guarantee good performance when the dictionary is small enough to reside in internal memory. For larger dictionaries (called **external dictionaries** or **files**) that must reside on a disk, we can get improved performance using search trees of higher degree. Before we jump into the study of high these high degree search trees, let us take a look at the popular **indexed sequential access method** (ISAM) for external dictionaries. This method provides good sequential and random access.

In the ISAM method the available disk space is divided into blocks, a block being the smallest unit of disk space that will be input or output. Typically, a block is one track long and can be input or output with a single seek and latency delay. The dictionary elements are packed into the blocks in ascending order and the blocks are used in an order that minimizes the delay in going from one block to next.

For sequential access the blocks are input in order, and the elements in each block are retrieved in ascending order. If each block contains m elements, the number of disk accesses per element is $1/m$.

To support random access, an index is maintained. This index contains the largest key in each block. Since the index contains only as many keys as there are blocks and since a block generally houses may elements (i.e., m is usually large), the index is generally small enough to reside in internal memory. To perform a random access of an element with key k, the index is searched for the single block that can contain the corresponding element; this block is retrieved from the disk and searched internally for the desired element. As a result, a single disk access is sufficient to perform a random access.

This technique may be extended to larger dictionaries that span several disks. Now the elements are assigned to disks in ascending order and then to blocks within a disk also in ascending order. Each disk maintains a block index that retains the largest key in each block. Additionally, an overall disk index maintains the largest key in each disk. This index generally resides in memory.

To perform a random access, the disk index is searched to determine the single disk that the desired record might reside on. Next, the block index for this disk is retrieved from the appropriate disk and searched for the block that is to block that is to be fetched from the disk. The block is then fetched and searched internally. In the extended scheme a random access requires two disk accesses (one to fetch a block index and another to fetch a block).

Since the ISAM method is essentially a formula based representation scheme, it runs into difficulty when inserts and deletes are performed. We can partially alleviate this difficulty by leaving space in each block so that a few inserts can be performed without moving elements across block boundaries. Similarly, we can leave empty space in the block after deletes, rather than perform an expensive shift of the elements across block boundaries to use the new free space.

4.2. *m*-way Search Trees

Definition An *m*-way search tree may be empty. If it is not empty, it is a tree that satisfies the following properties:

1.In the corresponding extended search tree (obtained by replacing zero pointers with external nodes), each internal node has up to m children and between 1 and $m - 1$ elements. (External nodes contain no elements and have no children.)

2. Every node with p elements has exactly $p + 1$ children.

3.Consider any node with p elements. Let $k_1,...,k_p$ be the keys of these elements. The elements are ordered so that $k_1 < k_2 < ... < k_p$. Let $c_0, c_1,..., c_p$ be the $p + 1$ children of the of the node.The elements in the subtree with root c_0 have keys smaller than k_1, those in the subtree with root c_p have keys larger than k_p, and those in the subtree with root c_i have keys larger than k_i but smaller than k_{i+1}, $1 \leq i < p$.

Although it is useful to include external nodes when defining an *m*-way search tree, external nodes are not physically represented in actual implementations. Rather, a null or zero pointers appears wherever there would otherwise be an external node.

Figure 4.1 shows a seven-way search tree. External nodes are shown as solid squares. All other nodes are internal nodes. The root has two elements (with keys 10 and 80) and

three children. The middle child of the root has six elements and seven children; six of these children are external nodes.

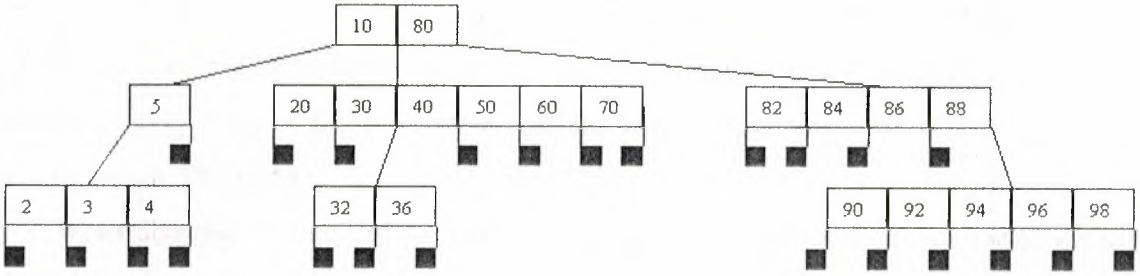


Figure 4.1 A seven-way search tree

Searching an *m*-Way Search Tree

To search the seven-way search tree in Figure 4.1 for an element with key 31, we begin at the root. Since 31 lies between 10 and 80, we follow the middle pointer. (By definition, all elements in the first subtree have key < 10, and all in the third one have key > 80.) The root of the middle subtree is searched. Since $k_2 < 31 < k_3$, we move into the first subtree. Now we determine that $31 < k_1$. This move causes us to fall off the tree; that is, we reach an external node. We conclude that the search tree contains no element with key 31.

Inserting into an m-Way Search Tree

If we wish to insert an element with key 31, we search for 31 as above and fall off the tree at the node [32,36]. Since this node can hold up to six elements (each node of a seven-way search tree can have up to six elements), the new element may be inserted as the first one in the node.

To insert an element with key 65, we search for 65 and fall off the tree by moving to the sixth subtree of the node [20,30,40,50,60,70]. This node cannot accommodate additional elements, and a new node is obtained. The new element is put into this node, and the new node becomes the sixth child of [20,30,40,50,60,70].

Deleting from an m-Way Search Tree

To delete the element with key 20 from the search tree of Figure 4.1, we first perform a search. The element is the first element in the element in the middle child of the root. Since $k_1 = 20$ and $c_0 = c_1 = 0$, we may simply delete the element from the node. The new middle child of the 84, we first locate the element. It is the second element in the third child of the root. Since $c_1 = c_2 = 0$, the element may be deleted from this node and the new node configuration [82,86,88].

When deleting the element with key 5, we have to do more work. Since the element to be deleted is the first one in its node and since at least one of its neighboring children (these children are c_0 and c_1) is nonzero, we need to replace the deleted element with an element from a nonempty neighboring subtree. From the left neighboring subtree (c_0), we may move up the element with largest key (i.e., the element with key 4).

To delete the element with key 10 from the root of Figure 4.1, we may replace this element with either the largest element in c_0 or the smallest element in c_1. If we opt to replace it with the largest in c_0, then the element with key 5 moves up and we need to find a replacement for this element in its original node. The element with key 4 is moved up.

Height of an *m*-Way Search Tree

An *m*-way search tree of height *h* (excluding external nodes) may have as few as *h* elements (one node per level and one element per node) and as many as $m^h - 1$. The upper bound is achieved by an *m*-way search tree of height *h* in which each node at levels 1 through $h - 1$ has exactly *m* children and nodes at level *h* have no children. Such a tree has $\sum_{i=0}^{h-1} m^i = (m^h - 1)/(m - 1)$ nodes. Since each of these nodes has $m - 1$ elements, the number of elements is $m^h - 1$.

As the number of elements in an m – way search tree of height h ranges from a low of *h* to a high of $m^h - 1$, the height of an m – way search tree with n elements ranges from a low of $\log_m(n+1)$ to a high of *n*.

A 200 – way search tree of height 5, for example, can hold $32*10^{10}-1$ elements but might told as few as 5. Equivalently, a 200 – way search tree with $32*10^{10}-1$ elements has a height between 5 and $32*10^{10}-1$. When the search tree resides on the disk, the search, insert, and delete times are dominated by the number of disk accesses made (under the assumption that each node is no longer than a disk block). Since the number of disk accesses needed for the search, insert, and delete operations are O(h) where h is the tree height, we need to ensure that the height is close to $\log_m(n+1)$. This assurance is provided by balanced m – way search trees.

4.3. B-Trees of Order *m*

Definition A **B – Tree of order** m is an m – way search tree. If the B – tree is not empty, the corresponding extended tree satisfies the following properties:

1. The root has at least two children.
2. All internal nodes other than the root have at least [m/2] children.
3. All external nodes are at the same level.

The seven – way search tree of Figure 4. 1 is not a B - tree of order 7, as it contains external nodes at more than one level (levels 3 and 4). Even if all its external nodes were at the same level, it would fail to be a B – tree of order 7 as it contains nonroot internal nodes with two (node [5]) and three (node [32,36]) children. Nonroot internal nodes in a B – tree of order 7 must have at least [7/2] = 4 children. A B – tree of order 7 appears in Figure 4.2. All external nodes are at level 3, the root has three children, and all remaining internal nodes have at least four children. Additionally, it is a seven – way search tree.

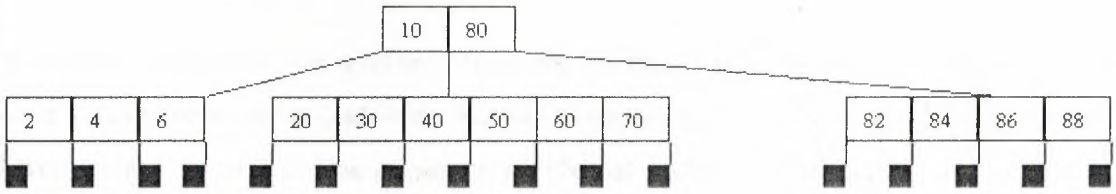


Figure 4.2 A B-tree of order 7

In a B – tree of order 2, no internal node has more than two children. Since an internal node must have at least two children, all internal nodes of a B – tree of order 2 have exactly

two children. This observation coupled with the requirement that all external nodes be on the same level implies that B – trees of order 2 are full binary trees. As such, these trees exist only when the number of elements is $2^h - 1$ for some integer h.

In a B – tree of order 3, internal nodes have either two or three children. So a B – tree of order 3 is also known as a 2 – 3 tree. Since internal nodes in B – trees of order four must have two, three, or four children, these B – trees are also referred to as 2-3-4 (or simply 2,4) trees. A 2 – 3 tree appears in Figure 4.3. Even though this tree has no internal node with four children, it is also an example of a 2-3-4 tree. To build a 2-3-4 tree in which at least one internal node has four children, simply add elements with keys 14 and 16 into the left child of 20.

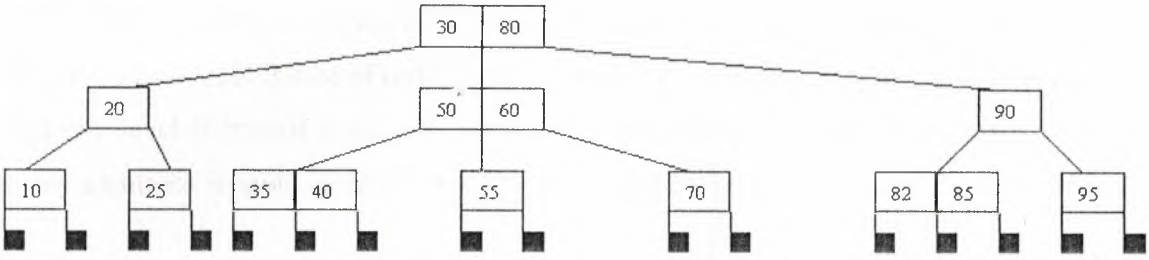


Figure 4.3 A 2-3 tree or B-tree order 3

4.4. Height of a B-Tree

Lemma 4.1 Let T be a B-tree of order m and height h. Let $d = [m/2]$ and let n be the number of elements in T.

a) $2d^{h-1} - 1 \le n \le m^h - 1$

b) $\log_m(n+1) \le h \le \log_d(n+1/2) + 1$

Proof The upper bound on n follows from the fact that T is an m – way search tree. For the lower bound the lower bound, note that the external nodes of the corresponding extended B-tree are at level $h + 1$. The minimum number of nodes on levels 1, 2, 3, 4, …, $h + 1$ is $1,2,2d,2d^2,...,2d^{h-1}$, so the minimum number of external nodes is one more than the number of elements

$n \ge 2d^{h-1} - 1$

(b)follows directly from (a).

From Lemma 4.1, it follows that a B-tree of order 200 and height 3 has at least 19,999 elements, and one of the same order and height 5 has at least $2*10^8 - 1$ elements. Consequently, if a B-tree of order 200 or more is used, the tree height is quite small even when the number of elements is rather large. In practice, the B-tree order is determined by the disk block size and the size of individual elements. There is no advantage to using a node size smaller than the disk block size, as each disk access reads or writes one block. Using a larger node size involves multiple disk accesses, each accompanied by a seek and latency delay, so there is no advantage to making the node size larger than one block.

Although in actual applications the B-tree order is large, our examples use a small m because a two-level B-tree of order m has at least $2d - 1$ elements. When m is 200, d is 100 and two-level B-tree of order 200 has at least 199 elements. Manipulating trees with this many elements is quite cumbersom . Our examples involve 2-3 trees and B-trees of order 7.

4.5. Searching a B-Tree

A B-tree is searched using the same algorithm as used for an m-way search tree. Since all internal nodes on some root-to-external-node path may be retrieved during the search, the number of disk accesses is at most h (h is the height of the B-tree).

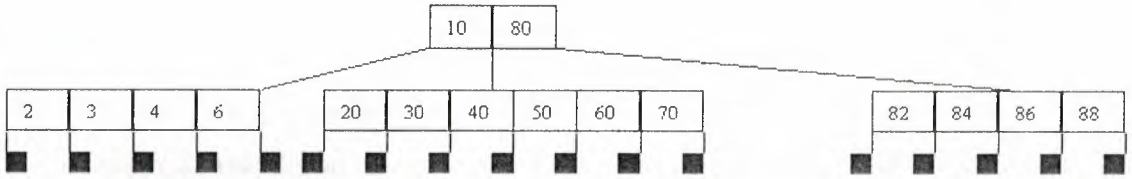
4.6. Inserting into a B-Tree

To insert an element into a B-tree, we first search for the presence of an element with the same key. If such an element is found, the insert fails because duplicates are not permitted. When the search is unsuccessful, we attempt to insert the new element into the last internal node encountered on the search path. For example, when inserting an element with key 3 into the B-tree of Figure 4.3, we examine the root and its left child. We fall off the tree at the second external node of the left child. We fall off the tree at the second external node of the left child. Since the left child currently has three elements and can hold up to six, the

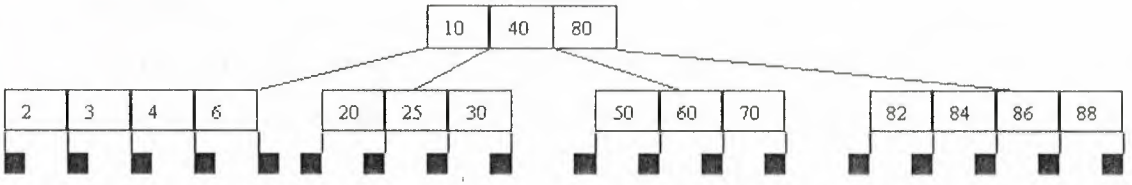
new element may be inserted into this node. The result is the B-tree of Figure 4.4(a). Two disk accesses are made to read in the root and its left child. An additional access is necessary to write out the modified left child.

Next, let us try to insert an element with key 25 into the B-tree of Figure 4.4(a). This element is to go into the node [20,30,40,50,60,70]. However, this node is full. *When the new element needs to go into a full node, the full node is split.* Let P be the full node. Insert the new element e together with a null pointer into P to get an overall node with m elements and $m + 1$ children. Denote this overall node as

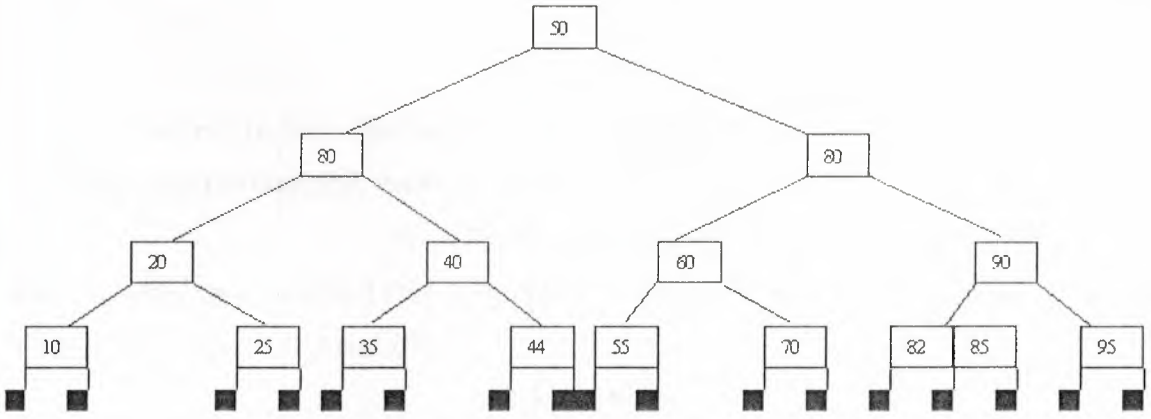
$$m, c_0, (e_1, c_1), \ldots, (e_m, c_m)$$



(a)Insert 3 into Figure 4.2



(b)Insert 25 into a



(c)Insert 44 into Figure 4.3

Figure 4.4 Inserting into B-tree

where the $e_i s$ are the elements and the $c_i s$ are the children pointers. The node is split around element e_d where $d = \lceil m/2 \rceil$. Elements to the left of this one remain in P, and those

to the right move into a new node Q. The pair (e_d, Q) is inserted into the parent of P. The format of the new P and Q is

$$P: d-1, c_0, (e_1, c_1), \ldots, (e_{d-1}, c_{d-1})$$

$$Q: m-d, c_d, (e_{d+1}, c_{d+1}), \ldots, (e_m, c_m)$$

Notice that the number of children of both P and Q is at least d.

In our example, the overall node is

7,0 (20,0), (25,0), (30,0), (40,0), (50,0), (60,0), (70,0)

and $d = 4$. Splitting around e_4 yields the two nodes

P: 3, 0, (20,0), (25,0), (30,0)

Q: 3, 0, (50,0), (60,0), (70,0)

When the pair (40,Q) is inserted into the parent of P, we get the B-tree of Figure 4.4(b).

To insert 25 into 4.4(a), we need to get the root and its middle child from the disk. Then we write to disk the two split nodes and the modified root. The total number of disk accesses is five.

As a final example, consider inserting an element with key 44 into the 2-3 tree of Figure 4.5. This element goes into the node [35,40]. Since the node is full, we get the overall node

3, 0, (35,0), (40,0), (44,0)

Splitting around $e_d = e_2$ yields the two nodes

P: 1, 0, (35,0)

Q: 1, 0, (44,0)

When we attempt to insert the pair (40,Q) into the parent A of P, we see that this node is full. Following the insertion, we get the overall node

A: 3, P, (40,Q), (50,C), (60,D)

where C and D are pointers to the nodes [55] and [70]. The overall node A is split to create a new node B. The new A and B are

A: 1, P, (40,Q)

B: 1, C, (60,D)

Now we need to insert the pair (50,B) into the root. Prior to this insertion, the root has the format

R: 2, S, (30,A), (80,T)

where S and T are, respectively, pointers to the first and third subtrees of the root. Following the insertion of the pair (50,B), we get the overall node

R: 3, S, (30,A), (50,B), (80,T)

This node is split around the element with key 50 to create a new R and a new node U as below:

R: 1, S, (30,A)

U:1, B, (80,T)

The pair (50,U) would normally be inserted into the parent of R. However, since R has no parent, we create a new root with the format

1, R, (50,U)

The resulting 2-3 tree appears in Figure 4.4(c).

The disk accesses are made to read in nodes [30,80], [50,60], and [35,40]. For each node that splits, two accesses are made to write the modified node and the newly created node. In our case three nodes are split, so six write accesses are made. Finally, a new root is created and written out. This write takes an additional disk access. The total number of disk accesses is 10.

When an insertion causes s nodes to split, the number of disk accesses is h (to read in the nodes on the search path) + $2s$(to write out the two split parts of each node that is split) + 1 (to write the new root or the node into which an insertion that does not result in a split is made). Therefore, the number of disk accesses needed for an insertion is $h + 2s + 1$, which is at most $3h + 1$.

4.7. Deletion from a B-Tree

Deletion is first divided into two cases: (1) the element to be deleted in a node whose children are external nodes (i.e., the element is in a leaf), and (2) the element is to be deleted element from a nonleaf. Case (2) is transformed into case (1) by replacing the deleted element with either the largest element in its left-neighboring subtree or the smallest element in its right-neighboring subtree. The replacing element is guaranteed to be in a leaf.

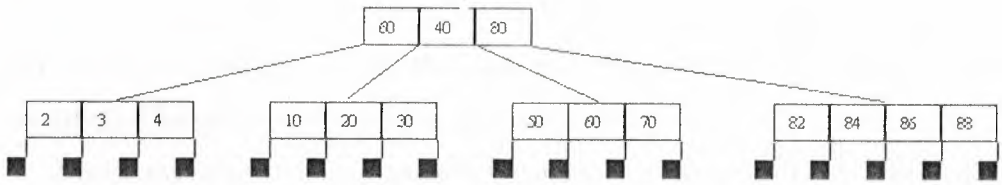
Consider deleting the element with key 80 from the B-tree of Figure 4.4(a). Since the element is not in a leaf, we find a suitable replacement. The possibilities are the element with key 70 (i.e., the largest element in the left-neighboring subtree) and 82 (i.e., the smallest element in the right-neighboring subtree). When we use the 70, the problem of deleting this element from the leaf [20,30,40,50,60,70] remains.

If we are to delete the element with key 80 from the 2-3 tree of Figure 4.4(c), we replace it with either the element with key 70 or that with 82. If we select the 82, the problem of deleting 82 from the leaf [82,85] remains.

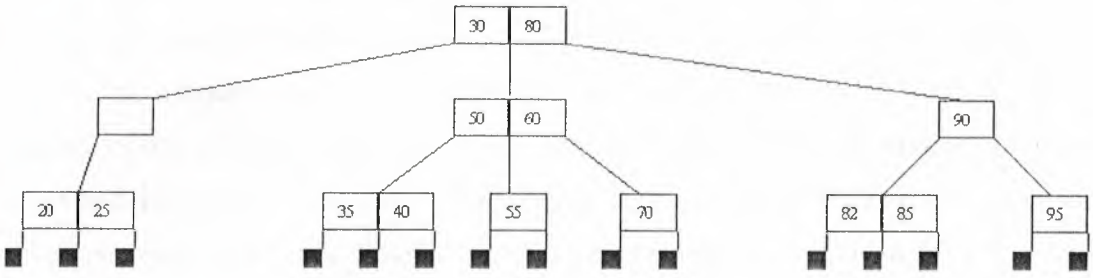
Since case (2) may be transformed into case (1) quite easily, we concern ourselves with case (1) only. To delete an element from a leaf that contains more than the minimum number of elements (1 if the leaf is also the root and than the minimum number of elements (1 if the leaf is also the root and $\lceil m/2 \rceil - 1$ if it is not) requires us to simply write out the modified node. (In case this node is the root, the B-tree becomes empty.) To delete 50 from the B-tree of Figure 4.4(a), we write out the modified node [20,30,40,60,70], and to delete 85 from the 2-3 tree of Figure 4.4(c), we write out the node [82]. Both cases require h disk accesses to follow the search path down to the leaf and an additional access to write out the modified version of the leaf that contained the deleted element.

When the element is being deleted from a nonroot node that has exactly the minimum number of elements, we try to replace the deleted element with an element from its nearest-left or –right sibling. Notice that every node other than the root has either a nearest-left sibling or nearest-right sibling or both. For example, suppose we wish to delete 25 from the B-tree of Figure 4.4(b). This deletion leaves behind the node [20,30], which has just two elements. However, since this node is a nonroot node of a B-tree of order 7, it must contain at least three elements. Its nearest-left sibling, [2,3,4,6], has an extra element. The largest element from here is moved to the parent node, and the intervening element (i.e., with key 10) is moved down to create the B-tree of Figure 4.5(a). The number of disk accesses is 2 (to go from the root to the leaf that contains 25) + 1 (to read in the nearest-left sibling of this leaf) + 3 (to write out the changed leaf, its sibling, and its parent) = 6.

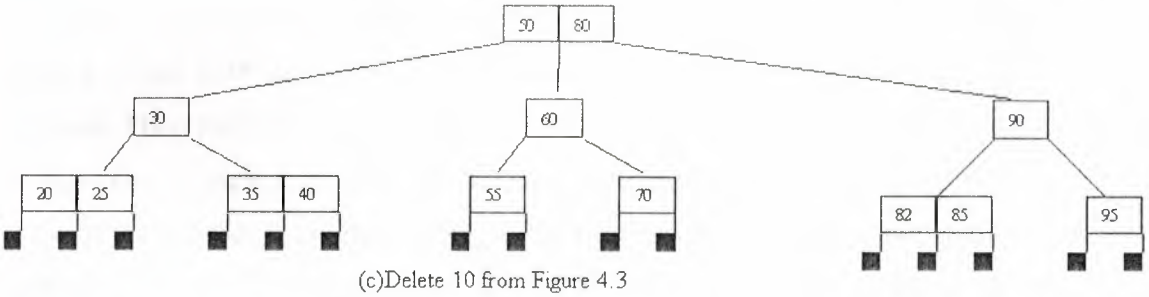
Suppose that instead of checking the nearest-left sibling of [20,30], we had checked its nearest-right sibling [50,60,70]. Since this node has only three elements, we cannot delete an element. (If the node had four or more, we would has moved its smallest element to the



(a)Delete 25 from Figure 4.4(b)



(b)After merging at leaf level



(c)Delete 10 from Figure 4.3

Figure 4.5 Deleting from a B-tree

parent and moved the element in the parent that lies between these two siblings into the leaf that is one element short.) Now, we can proceed to check the nearest-left sibling of [20,30]. Performing this check requires an additional disk access, and we are not certain that this nearest sibling will have an extra element. *In the interest of keeping the worst-case disk access count low, we shall check only one of the nearest siblings of a node that is one element short.*

When the nearest sibling that is checked has no extra elements, we merge the two siblings with the element between them in the parent into a single node. Since the siblings have $d-2$ and $d-1$ elements each, the merged node has $2d-2$ elements. As $2d-2$ equals $m-1$ when m is odd and $m-2$ when m is even, there is enough space in a node to hold this many elements.

In our example the siblings [20,30] and [50,60,70] and the element with key 40 are merged into single node [20,30,40,50,60,70]. The resulting B-tree is that of Figure 4.4(a).

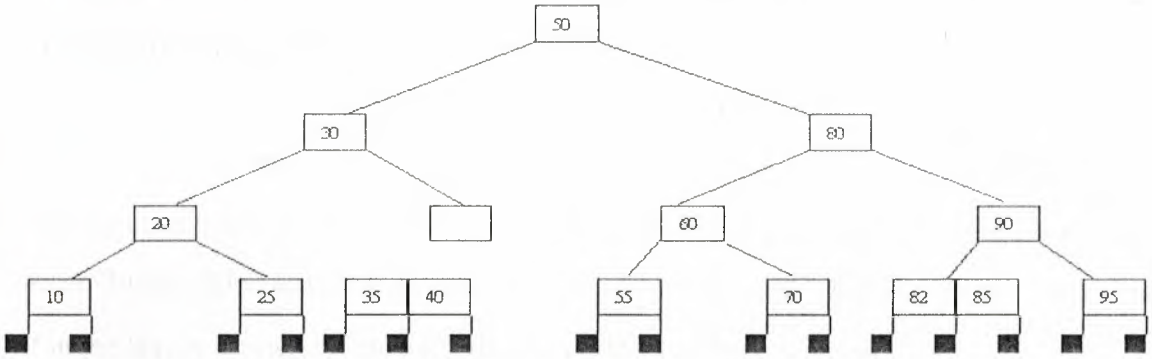
This deletion requires two disk accesses to get to the node [20,25,30], another access to read in its nearest-right sibling, then two more accesses to write out the two nodes that are modified. The total number of disk accesses if five.

Notice that since merging reduces the number of elements in the parent node, the parent may end up being one element short. If the parent becomes one element short, we will need to check the parent's nearest sibling and either get an element from there or merge with it. If we get an element from the nearest-right (-left) sibling, then the left-most (right-most) subtree of this sibling is also taken. If we merge, the grandparent may become one element short and the process will need to be applied at the grandparent. At worst, the shortage will propagate back to the root. When the root is one element short, it is empty. The empty root is discarded, and the tree height decreases by one.

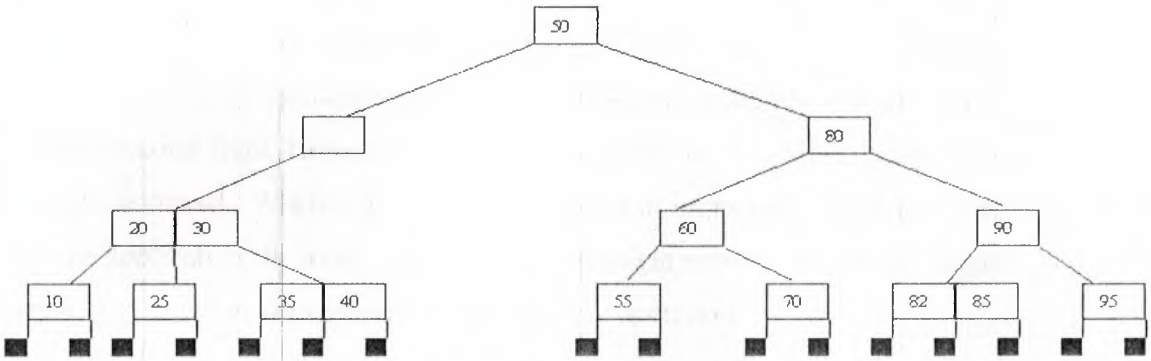
Suppose we wish to delete 10 from the 2-3 tree of Figure 4.5. This deletion leaves behind a leaf with zero elements. Its nearest-right sibling [25] does not have an extra element. Therefore, the two sibling leaves and the in-between element in the parent (10) are merged into a single node. The new tree structure appears in Figure 4.5(b). We now have a node at level 2 that is an element short. Its nearest-right sibling has an extra element. The left-most element (i.e., the one with key 50) moves to parent and the element with key 30 moves down. The resulting 2-3 tree appears in Figure 4.5(c). Notice that the left subtree of the former [50,60] has moved also. This deletion took three read accesses to get to the nearest-right siblings of the level 3 and 2 nodes; and four write accesses to write out the four nodes at levels 1, 2, and 3 that were modified. The total number of disk accesses is nine.

As a final example, consider the deletion of 44 from the 2-3 tree of Figure 4.4(c). When the 44 is removed from the leaf it is in, this leaf becomes short one element. Its nearest-left sibling does not have an extra element, and so the two siblings together with the in-between element in the parent are merged to get the tree of Figure 4.6(a). We now have a node at level 3 that is one element short. Its nearest-left sibling is examined and found to have no extra elements, so the two siblings and the in-between element in their parent are merged. The tree of Figure 4.5(b) is obtained. Now we have a level 2 node that is one element short. Its nearest-right sibling has no extra elements, and we perform another merge to get the tree of Figure 4.6(c). Now the root is an element short. Since the root becomes an element short

only when it is empty, the root is discarded. The final 2-3 tree is shown in Figure 4.6(d). Notice that when the root is discarded, the tree height reduces by one.



(a)After merging at leaf level



(b)After merging at level 3

We need four disk accesses to find the leaf that contains the element to be deleted, three nearest-sibling accesses, and three write accesses. The total number is 10.

The worst case for a deletion from a B-tree of height h is when merges take place at levels h, $h-1$,..., and 3, and at level 2, we get an element from a nearest sibling. The worst-case disk access count is $3h$; (h reads to find the leaf with the element to be deleted) + ($h-1$ reads to get nearest siblings at levels 2 through h) + ($h-2$ writes of merged nodes at level 3 through h) + (3 writes for the modified root and two level 2 nodes).

4.8. Node Structure

Our discussion has assumed a node structure of the form

$$s, c_0, (e_1, c_1), (e_2, c_2), \ldots, (e_s, c_s)$$

where s is the number of elements in the node, the $e_i s$ are the elements in ascending order of key, and the $c_i s$ are children pointers. When the element size is large relative to the size of a key, we may use the node structure.

$$s, c_0, (k_1, c_1, p_1), (k_2, c_2, p_2), \ldots, (k_s, c_s, p_s)$$

where the $k_i s$ are the element keys and the $p_i s$ are the disk locations of the corresponding elements. By using this structure, we can use a B-tree of a higher order. An even higher-order B-tree, called a B′-tree, becomes possible if nonleaf nodes contain no p_i pointers and if in the leaves we replace the null children pointers with p_i pointers.

Another possibility is to use a balanced binary search tree to represent the contents of each node. Using a balanced binary search tree in this way reduces the permissible order of the B-tree, as with each element we need a left- and right-child pointer as well as a balance factor or color field. However, the CPU time spent inserting/deleting an element into/from a node decreases. Whether this approach results in improved overall performance depends on the application. In some cases a smaller m might increase the B-tree height, resulting in more disk accesses for each search/insert/delete operation

CHAPTER 5
APPLICATIONS

5.1.Histogramming

In the histogramming problem we start with a collection of n keys and must output a list of the distinct keys and the number of times (i.e., frequency) each occurs in the collection. Figure 5.1 gives an example with 10 keys. The problem input appears in Figure 5.1(a), and the histogram is presented in Figure 5.1(b) as a table and as a bar chart in Figure 5.1(c). Histogramming is commonly performed to determine the distribution of data. For example, we may histogram the scores on a test, the gray-scale values in an image, the cars registered in Gainesville (the key being the manufacturer), and the highest degree earned by persons living in Los Angeles.

When the key values are integers in the range zero through r and r is reasonably small, the histogram can be computed in linear time by a rather simple procedure (Program 5.1) that uses the array element h[i] to determine the frequency of the key i. Other integral key types may be mapped into this range to use Program 5.1. For example, if the keys are lowercase letters, we may use the mapping [a, b,..., z] = [0, 1, ... , 25].

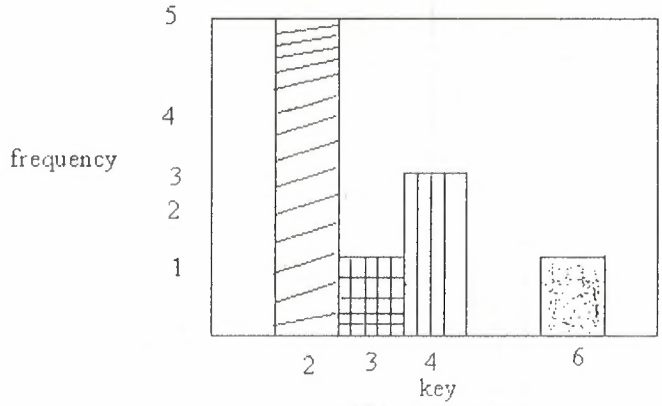
Program 5.1 becomes infeasible when the key range is very large as well as when the key type is not integral (for example, when the keys are real numbers). Suppose we are determining the frequency with which different words occur in a text. The number of possible different words in very large compared to the number that might actually appear in the text. In such a situation, we may sort the keys and then use a simple left-to-right scan to determine the number of keys for each distinct key value. The sort can be accomplished in $O(n\log n)$ time, and the ensuing left-to-right scan takes $\Theta(n)$ time; the overall complexity is $O(n\log n)$. This solution can be improved upon when the number m of distinct keys is small when compared to n. By using balanced search trees such as AVL and red-black trees, we can solve the histogramming problem in $O(n\log m)$ time. Furthermore, the balanced search tree solution requires only the distinct keys to be stored in memory. Therefore, this solution is appropriate even in situations when n is so large that we do not have enough memory to accommodate all keys (provided, of course, there is enough memory for the distinct keys).

n = 10 ; keys=[2,4,2,2,3,4,2,6,4,2]

(a)Input

key	frequency
2	5
3	1
4	3
6	1

(b)Output histogramming example



(c)Histogram plot

Figure 5.1 Histogramming example

This solution we describe uses a binary search tree and so has expected complexity O(*n*log*m*). By replacing the binary search tree used in this solution with a balanced search tree, the claimed complexity is achieved. In our binary search tree solution, we extend the class BSTree by adding the public member

BSTree<E,K>& InsertVisit(const E& e, void (*Visit) (E& u));

that inserts element e into the search tree provided no element with key equal to e.key exists. In case such an element u exists, function visit is invoked. The code for this member may be obtained from that for Insert (Program 1.3) by replacing the line

else throw BadInput(); //duplicate key

with the lines

```
void main(void)
{// Histogram of nonnegative integer values.
   int n,  // number of elements
       r;  // values between 0 and r
   cout << "Enter number of elements and range"
        << endl;
   cin >> n >> r;
   // create histogram array h
   int *h;
   try {h = new int[r + 1];}
   catch (NoMem)
```

```
    {cout << "range is too large" << endl;
     exit(1);}

  // initialize array h to zero
  for (int i = 0; i <= r; i++)
     h[i] = 0;
  // input data and compute histogram
  for (int i = 1; i <= n; i++) {
     int key;  // input value
     cout << "Enter element" << i << endl;
     cin >> key;
     h[key]++;
     }
  // output histogram
  cout << "Distinct elements and frequencies are"
       << endln;
  for (int i = 0; i <= r; i++)
     if (h[i]) cout << I << "  " << h[i] << endln; }
```

Program 5.1 Simple histogramming program

```
else {Visit(p->data);
return *this;};
```

Program 5.2 gives the code for the new histogramming program. During an element visit, its frequency count is incremented by one.

```
class eType {
    friend void main(void);
    friend void Add1(eType&);
    friend ostream& operator << (ostream&, eType);
    public:
        operator int() const {return key;}
    private:
        int key,  // element value
            count; // frequency
};
ostream& operator<<(ostream& out, eType x)
   {out << x.key << "  " x.count << "  "; return out;}
void Add1(eType& e) {e.count++;}
void main(void)
{ // Histogram using a search tree.
  BSTree<eType,int> T;
  int n; // number of elements
  cout << "Enter number of elements" << endl;
  cin >> n;
```

```
  // input elements and enter into tree
  for (int i = 1; i <= n; i++) {
     eType e; // input element
     cout << "Enter element" << i << endl;
     cin >> e.key;
     e.count = 1;
     // put e in tree unless match already there
     // in latter case increase count by 1
     try {T.InsertVisit (e, Add1);}
     catch (NoMem)
           {cout << "Out of memory" << endl;
            exit(1);}
     }
  // output distinct elements and their counts
  cout << "Distinct elements and frequencies are" << endl;
  T.Ascend();
}
```

Program 5.2 Histogramming using a search tree

5.2. Best-Fit Bin Packing

By using a balanced search tree, we can implement the best-fit method to pack n objects into bins of capacity c to run in $O(n\log n)$ time. The search tree will contain one element for each bin that is currently in use and has nonzero available capacity. Suppose that when object i is to be packed, there are nine bins (a through i) in use that still have some space left. Let the available capacity of these bins be 1, 3, 12, 6, 8, 1, 20, 6, and 5, respectively. Notice that it is possible for two or more bins to have the same available capacity. The nine bins may be stored in a binary search tree with duplicates (i.e., a member of either DBSTree or DAVLtree), using as key the available capacity of a bin.

Figure 5.2 shows a possible binary search tree for the nine bins. For each bin, the available capacity is shown inside a node, and the bin name, outside. This tree is also an AVL tree. If the object i that is to be packed requires s[i] = 4 units, we can find the bin that provides the best fit by starting at the root of the tree of Figure 5.2. The root tells us bin h has an available capacity of six. Since object i fits into this bin, bin h becomes the candidate for the best bin. Also, since the capacity of all bins in the right subtree is at least six, we

need not look at the bins in this subtree in our quest for the best bin. The search proceeds to the left subtree. The capacity of bin *b* isn't adequate to accommodate our object, so the search for the best bin moves into the right subtree of bin b. The bin, bin *i*, at the root of this subtree has enough capacity, and it becomes the new candidate for the best bin. From here, the search moves into the left subtree of bin *i*. Since this subtree is empty, there are no better candidate bins and bin *i* is selected.

As another example of the sear h for the best bin, suppose s[i] = 7. The search again starts at the root. The root bin, bin h, does not have enough capacity for this object, so our quest for a bin moves into the right subtree. Bin *c* has enough capacity and becomes the new candidate bin. From here we move into *c*'s left subtree and examine bin *d*. It does not have enough capacity to accommodate the object, so we continue with the right subtree of *d*. Bin *e* has enough capacity and becomes the new candidate bin. We then move into its left subtree, which is empty. The search terminates.

When we find the best bin, we can delete it from the search tree, reduce its capacity by s[i], and reinsert it (unless its remaining capacity is zero). If we do not find a bin with enough capacity, we can start a new bin.

To implement this scheme, we can use the class DBSTree to obtain O(*n*log*n*) expected performance or the class DAVLtree for O(*n*log*n*) performance in all instances. In either case we need to extend the class definition to include a public member FindGE(k,kout) that finds the smallest bin capacity Kout that is ≥ k. This member takes the from given in Program 5.3. Its complexity is O(*height*). The code for the class AVLtree is identical.

```
template<class E,class K>
bool DBSTree<E,K>::FindGE(const K& k, K& Kout) const
{// Find smallest element with value >= k.
   BinaryTreeNode<E> *p = root, // search pointer
                     *s = 0; // pointer to smallest
                             // >= k found so far
   // search the tree
 while (p) {
   // is p a candidate?
   if (k <= p->data) {// yes
      s = p;  // p is a better candidate than s
      // smaller elements in left subtree only
      p = p->LeftChild;}
   else // no, p->data too small, try right subtree
```

```
            p = p->RightChild;
        }
    if (!s) return false; // not found
    kout = s->data;
    return true;
}
```

Program 5.3 Finding the smallest key ≥ k

Program 5.4 packs n objects into bins using best-fit strategy.

```
class BinNode{
    friend void BestFitPack(int *, int, int);
    friend ostream& operator<< (ostream&, BinNode);
    public:
        operator int() const {return avail;}
    private:
        int ID,    // bin identifier
            avail; // available capacity
};
ostream& operator<<(ostream& out, BinNode x)
    {out << "Bin " << x.ID << " " << x.avail;
     return out;}
void BestFitPack(int s[], int n, int c)
{
     int b = 0;                          // number of bins used
     DBSTree<BinNode, int> T;  // tree of bin capacities
     // pack objects one by one
     for (int i = 1; i <= n; i++)  {// pack object i
          int k;            // best fit bin number
          BinNode e;  // corresponding node
          if (T.FindGE(s[i], k)) // find best bin
              T.Delete(k, e);       // remove best bin from tree
          else {// no bin large enough start a new bin
                e = *(new BinNode);
                e.ID = ++b;
                e.avail = c;}
           cout << "Pack object " << i << " in bin "
                << e.ID << endl;
          // update available capacity and put bin in tree unless avail capacity is zero
          e.avail -= s[i];
          if (e.avail) T.Insert(e);
          }
}
```

Program 5.4 Bin packing using best fit

CONCLUSION

The world of data structures has a wide variety of trees. In the present project we discussed search trees. It develops tree structures suitable for the representation of a dictionary. We examined search trees. These trees provide an asymptotic performance that is comparable to that of skip lists. The expected complexity of a search, insert, or delete operation is $O(\log n)$, while the worst case complexity is $\Theta(n)$. AVL trees perform at most one rotation following an insert and $O(\log n)$ rotations following a delete. However, red-black trees perform a single rotation following either an insert or delete. AVL and red-black trees guarantee good performance when the dictionary is small enough to reside in internal memory. For larger dictionaries (called **external dictionaries** or **files**) that must reside on a disk, we can get improved performance using search trees of higher degree that is B-Trees.

REFERENCES

[1] Sartaj Sahni, Data Structures, Algorithms & Applications in C++, New York, NY , 1998

[2] E. Horowitz, S. Sahni, and D. Mehta, W. H. Freeman, *Fundamentals of Data Structures in C++*, New York, NY, 1994.

[3] R. Bayer, papers "Symmetric Binary B-Trees: *Data Structures and Maintenance*", *Acta Informatica*, 1, 1972, 290-360, *www.c++journal.com*

APPENDIX A

```
template <class T>
class BinaryTreeNode{

public:
        BinaryTreeNode() {LeftChild=RightChild=0;}
        BinaryTreeNode(const T& e)
        {
                data = e;
                LeftChild=RightChild=0;}
        BinaryTreeNode(const T& e, BinaryTreeNode *l, BinaryTreeNode *r)
        {
                data = e;
                LeftChild=l;
                RightChild=r;}
private:
        T data;
        BinaryTreeNode<T> *LeftChild,
                          *RightChild;
};

template <class T>
void InOrder(BinaryTreeNode<T> *t)
{
        if (t){
                InOrder(t->LeftChild);
             Visit(t);
                InOrder(t->RightChild);
}
}
template <class T>
void PostOrder(BinaryTreeNode<T> *t)
{
        if (t) {
                PostOrder(t->LeftChild);
                PostOrder(t->RightChild);
                Visit(t);
        }
}
template <class T>
void Infix(BinaryTreeNode<T> *t)
{
```

```
        if (t) {cout << '(';
        Infix(t->LeftChild);
        cout<< t->data;
        Infix(t->RightChild);
        cout << ')';}
}
template <class T>
void LevelOrder(BinaryTreeNode<T> *t)
{ while(t) {
        Visit(t);
        if(t->LeftChild) Q.Add(t->LeftChild);
        if(t->RightChild) Q.Add(t->RightChild);
        try{Q.Delete(t);}
        catch (OutOfBounds) {return;}
}
}
```

Program A.1 Node class for linked list and Implementation of its public members

```
#include <iostream.h>
#include "BinaryTreeNode.cpp"
template<class T>
class BinaryTree:public BinaryTreeNode<T>{
        public:
                BinaryTree() {root = 0;};
                ~BinaryTree(){};
                bool IsEmpty() const
                {return ((root)?false : true);}
                bool Root(T& x) const;
                void MakeTree(const T& element,
                        BinaryTree<T>& left, BinaryTree<T>& right);
                void BreakTree(T& element, BinaryTree<T>& left, BinaryTree<T>&
right);
                void PreOrder(void(*Visit)(BinaryTreeNode<T> *u))
                {PreOrder(Visit, root);}
                void InOrder(void(*Visit)(BinaryTreeNode<T> *u))
                {InOrder(Visit,root);}
                void PostOrder
                        (void(*Visit)(BinaryTreeNode<T> *u))
                {PostOrder(Visit,root);}
                void LevelOrder
                        (void(*Visit)(BinaryTreeNode<T> *u));
                void PreOutput()
                {PreOrder(Output,root);cout<<endl;}
                void InOutput()
                {InOrder(Output,root);cout << endl;}
                void PostOutput()
                {PostOrder(Output,root);cout << endl;}
```

```
        void LevelOutput()
        {LevelOrder(Output);cout << endl;}
    private:
        BinaryTreeNode<T> *root;
        void PreOrder(void(*Visit)
            (BinaryTreeNode<T> *u), BinaryTreeNode<T> *t);
        void InOrder (void(*Visit)
            (BinaryTreeNode<T> *u), BinaryTreeNode<T> *t);
        void PostOrder(void(*Visit)
            (BinaryTreeNode<T> *u), BinaryTreeNode<T> *t);
        static void Output(BinaryTreeNode<T> *t)
        {cout << t->data << " ";}
};
template<class T>
bool BinaryTree<T>::Root(T& x) const
{
    if (root) {x=root->data;
    return true;}
    else return false;
}
template<class T>
void BinaryTree<T>::MakeTree(const T& element,
                                        BinaryTree<T>& left,
BinaryTree<T>& right)
{
    root = new BinaryTreeNode<T>
        (element, left.root, right.root);
    left.root=right.root=0;
}
template<class T>
void BinaryTree<T>::BreakTree(T& element, BinaryTree<T>& left, BinaryTree<T>&
right){
if (!root) throw BadInput();
element=root->data;
left.root=root->LeftChild;
right.root=root->RightChild;
delete.root;
root=0;
}
template<class T>
void BinaryTree<T>::PreOrder(void(*Visit)(BinaryTreeNode<T> *u),
                                        BinaryTreeNode<T> *t)
{
    if (t) {Visit(t);
    PreOrder(Visit, t->LeftChild);
    PreOrder(Visit, t->RightChild);
    }
```

```
}


template <class T>
void BinaryTree<T>::InOrder(void(*Visit)(BinaryTreeNode<T> *u),BinaryTreeNode<T>
*t)
{
        if (t) {InOrder(Visit, t->LeftChild);
        Visit(t);
        InOrder(Visit, t->RightChild);
        }
}
template <class T>
void BinaryTree<T>::PostOrder(void(*Visit)(BinaryTreeNode<T> *u),
                                                        BinaryTreeNode<T> *t)
{
        if (t) {PostOrder(Visit, t->LeftChild);
             PostOrder(Visit, t->RightChild);
        }
}
template <class T>
void BinaryTree<T>::LevelOrder(void(*Visit)(BinaryTreeNode<T> *u))
{
        LinkedQueue<BinaryTreeNode<T>*> Q;
        BinaryTreeNode<T> *t;
        t=root;
        while (t) {
                Visit(t);
                if (t->LeftChild) Q.Add(t->LeftChild);
                if (t->RightChild) Q.Add(t->RightChild);
                try {Q.Delete(t);}
                catch (OutOfBounds) {return;}
        }
}
int count=0;
BinaryTree<int> a,x,y,z;
template <class T>
void ct(BinaryTreeNode<T> *t) {count++;}
void main(void)
{
        y.MakeTree(1,a,a);
        z.MakeTree(2,a,a);
        x.MakeTree(3,y,z);
        y.MakeTree(4,x,a);
        y.PreOrder(ct);
        y.PreOutput();
        cout << "\n";
```

```
        cout << count << endl;
}
```

Program A.2 Binary tree complete Program

A.1. Output

The Output of the above is

4 3 1 2

4

cout << count statement gives out 4. y.PreOutput() gives out 4 3 2 1 as output.