'CEO' is above the node 'SVP' (Senior Vice-President) and these two nodes are connected. This implies that a senior vice-president reports to the CEO. Similarly, a PL (Project Leader) reports to a PM (Project Manager). It should be noted that there is no reporting relationship between nodes in the same level. This tree originates from a unique node, 'CEO' which is the root node of this tree.

7.2 Definition of trees

A tree may be defined as a finite set "T" of one or more nodes such that: (a) there is a specially designated node called the root of the tree, (b) the remaining nodes (excluding the root) are partitioned into $n \geq 0$ disjoint sets $T_1, T_2, \ldots, T_n$ and each of these sets is a tree in turn. The trees $T_1, T_2, \ldots, T_n$ are called the sub-trees (or children) of the root. Note that the definition is recursive as the children of the root of a tree are trees once again.

In Figure 7.1, the root of the family tree is "Prakash". It has three sub-trees or children, rooted at "Anita", "Raju" and "Soumen". As the sub-trees are disjoint sets by definition, the sub-trees rooted at "Anita", "Raju" and "Soumen" in Figure 7.1 are independent and the nodes in these trees are not anyway connected to one another. It can also be said that every node in a tree is the root of some sub-tree. Therefore, Soumen is the root of a sub-tree of Prakash which has two sub-trees Partha and Mou. A node may also have an empty sub-tree as one of its child. For example, Goutam is a root of a tree with no sub-trees. Such nodes which do not have any children are called leaf or terminal nodes.

If a line is drawn from the node "Goutam" to the node "Raju" then both the nodes "Raju" and "Anita" become the parents of "Goutam". That is, more than one node becomes the parent of another node. Moreover, a cycle is formed containing four nodes – Prakash, Anita, Goutam and Raju. This scenario is depicted in Figure 7.3.

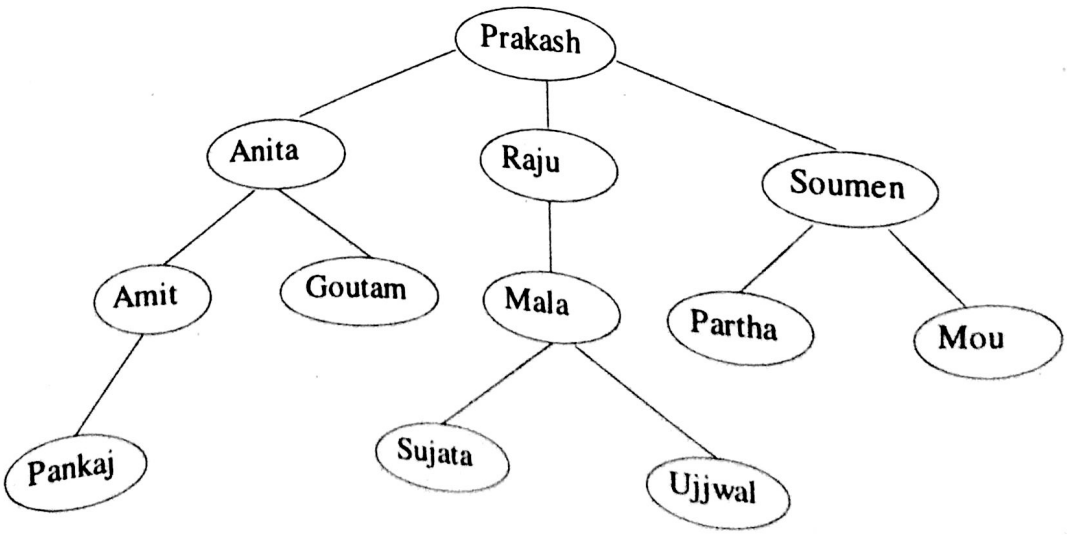


Figure 7.3 The family tree with a cycle.

Obviously, both the sub-trees rooted at "Anita" and "Raju" contain the same node "Goutam" and therefore, these sub-trees are not disjoint. According to the definition of a tree, these sub-trees must be disjoint. So, the diagram in Figure 7.3 does not represent a tree.

Generally, it is a convention to draw the root node at the top and to let the tree grow downwards. This convention is followed while drawing trees in Figures 7.1 and 7.2. However, this is merely a convention and it is not mandatory to follow this convention.

Number of nodes connected by links to a given node "n" is called the degree of the node "n". For example, degree of the node "SVP" in Figure 7.2 is 3 and degree of the node "SE" is 1. Degree of a leaf node or a terminal node is always 1. Some examples of general trees are shown in Figure 7.4.

The level of a node is recursively defined as follows. The level of the root node is defined as 0. The level of any other node is one more than the level of its parent. In Figure 7.4(b), the level of the node "D" is 2 (i.e., one higher then the level of its parent "B" which is 1).

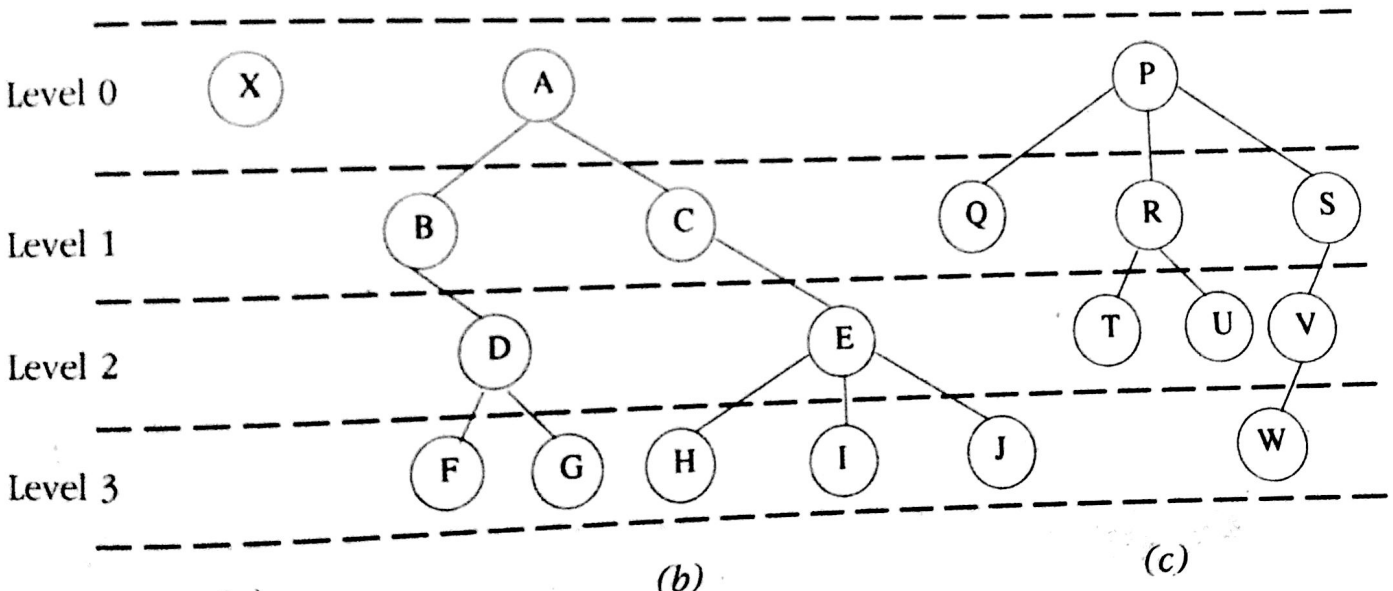


Figure 7.4 Some general trees.

If a node "y" is a child of another node "x" then "x" is said to be the parent of "y". For example, the parent of the node "E" in the tree of Figure 7.4(b) is the node "C". The links between a parent and its children are also referred to as edges or branches. An edge may be defined as a tuple (x, y) which denotes the link between the node "x" and its child "y". A collection of edges connecting one node to another is often termed as a path. For example, the collection of edges {(V, S), (S, P), (P, R), (R, U)} denotes a path from the node "V" to the node "U" in the tree of Figure 7.4(c). All nodes occurring in the path from a given node "n" to the root of a tree are called ancestors of "n".

7.3 Binary Tree

Binary tree is an important class of tree data structure in which a node can have at most two children (which are sub-trees). Moreover, children (or the sub-trees) of a node of a binary tree are ordered. One child is called the "left" child and the other is called the "right" child. An example of a binary tree is shown in Figure 7.5.

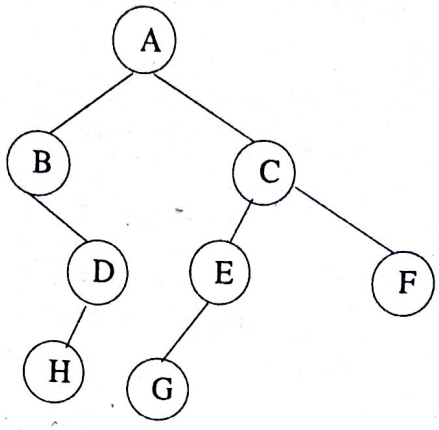


Figure 7.5 A binary tree.

In Figure 7.5, the node "A" (which is the root node) has two children "B" and "C". Similarly, each of the nodes "B", "D" and "E" has only one child "D", "H" and "G" respectively. As every node in a tree is the root of some other sub-tree, "B" and "C" are the roots of the sub-trees of the node "A". In Figure 7.5, the node "E" has a left sub-tree rooted at "G" and the node "B" has a right sub-tree rooted at "D" only. The nodes "H", "G" and "F" have no sub-trees and they are leaf nodes in this tree.

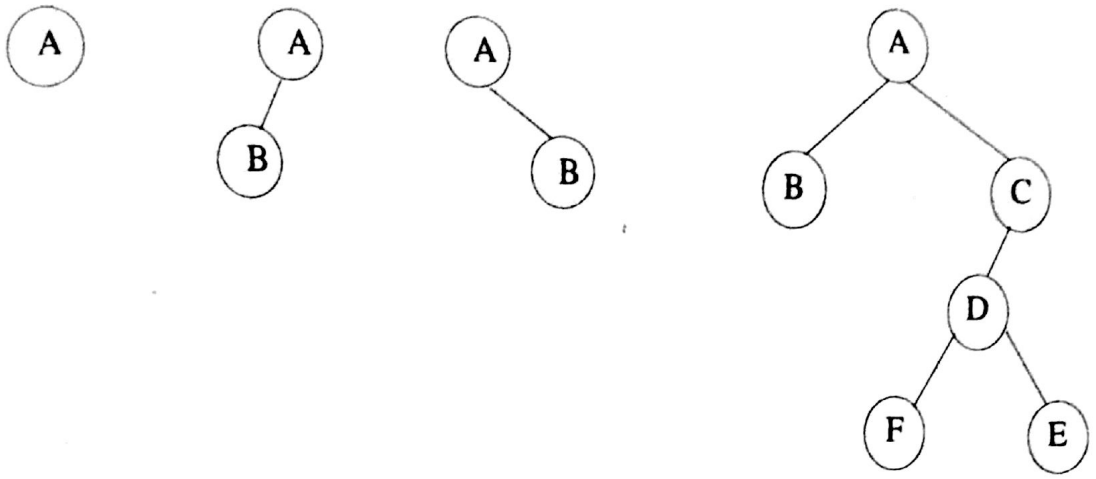
Similar to a tree, a binary tree may also be defined recursively as follows.

i) An empty tree is a binary tree.

ii) A binary tree consists of a node called "root", a left sub-tree and a right sub-tree both of which are binary trees once again.

It should be clear that, by definition, any non-empty binary tree is a tree as well. Examples of some binary trees are shown in Figure 7.6.

Note that, binary trees drawn in Figures 7.6 (ii) and 7.6 (iii) are distinct since the root of the binary tree of Figure 7.6 (ii) has an empty right sub-tree while the root of the binary tree of Figure 7.6 (iii) has an empty left sub-tree. But, the trees of Figures 7.6 (ii) & (iii) are identical, as the ordering of the sub-trees of the root is not significant for trees. Thus, it must be noted that there are fundamental differences between a binary tree and a tree.



(i) *(ii)* *(iii)* *(iv)*

Figure 7.6 Examples of binary trees.

Level of a node in a binary tree is defined in the same way as level is defined for nodes of a general tree. For example, the level of the node "D" of the tree of Figure 7.6(iv), is one higher than the level of node "C" (which is the parent of node "D"). The root node "A" has a level 0. Therefore, the level of "C" is 1 and level of "D" is 2.

Height of a binary tree

The height of a binary tree "T", denoted by height(T), is defined as follows.

height(T) = maximum level of any node of a binary tree "T".

Consider the binary tree of Figure 7.6(iv). The maximum level of any node in this tree is 4. Therefore, height of this tree is 4.

If all non-leaf nodes of a binary tree have exactly two non-empty children and the levels of all leaf nodes of a binary tree are the same then the tree is called a complete or a full binary tree. The tree shown in Figure 7.6(iv) is not a full binary

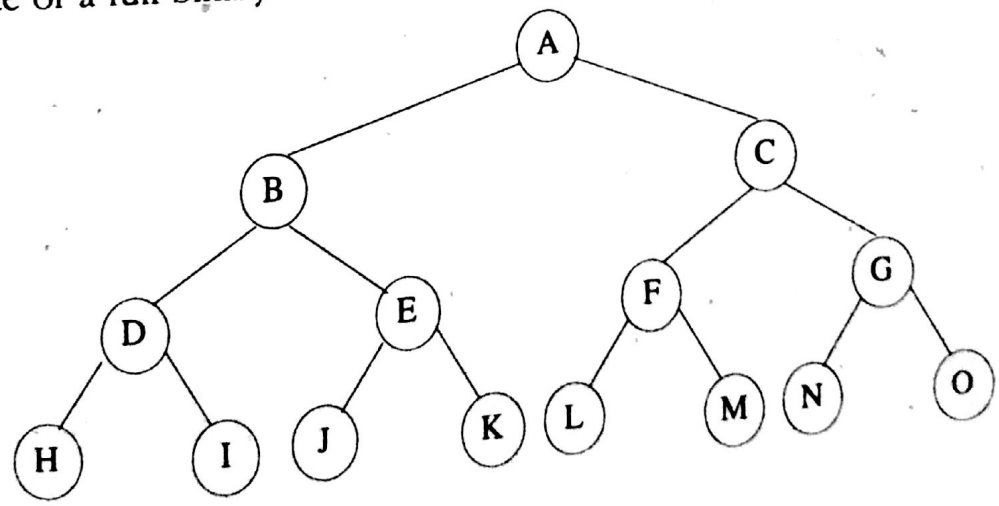


Figure 7.7 A complete binary tree of height 3.

number, the number of not well formed strings is to be excluded. One more interesting point to note is that the number of not well formed strings of "n" left and "n" right parentheses is the same as the total number of strings of (n-1) left and (n+1) right parentheses. The total number of strings of (n-1) left and (n+1) right parentheses is $^{2n}C_{n-1}$. Thus, the total number of well formed strings of "n" parentheses is

$$^{2n}C_{n} - {}^{2n}C_{n-1}$$

To complete the proof, one has to show that the number of not well formed strings of "n" left and "n" right parentheses is the same as the total number of strings of (n -1) left and (n +1) right parentheses. This can be proved by showing that any not well formed string containing "n" left and "n" right parentheses can be converted to a string of (n-1) left and (n +1) right parentheses and vice-versa. Take a string of "n" left and "n" right parentheses which is not well formed. Scan it from left to right and stop at the k-th element of the string where the number of right parentheses encountered so far outnumbers the number of left parentheses. In the remaining part of the string, replace left parentheses by right parentheses and right parentheses by left parentheses. The resultant string contains (n-1) left parentheses and (n+1) right parentheses.

The procedure to convert a string of (n-1) left and (n +1) right parentheses to a not well formed string of "n" left and "n" right parentheses is similar. Scan the given string to find out a position "k" where the number of right parentheses encountered so far outnumbers the number of left parentheses. In the remaining part of the given string exchange left for right parentheses and right for left parentheses. The resultant string contains "n" left and "n" right parentheses and the string is not well formed.

7.8 Threaded binary tree

When a binary tree is represented using pointers then pointers to empty sub-trees are set to null. That is, the "left" pointer of a node whose left child is an empty sub-tree is normally set to NULL. Similarly, the "right" pointer of a node whose right child is an empty sub-tree is also set to NULL . Thus, a large number of pointers are set to NULL. These null pointers could be used in a different and more effective way. Assume that the "left" pointer of a node "n" is set to NULL as the left child of "n" is an empty sub-tree. Then, the "left" pointer of "n" can be set to point to the inorder predecessor of "n". Similarly, if the "right" child of a node "m" is empty then the "right" pointer of "m" can be set to point to the inorder successor of "m". In the tree of Figure 7.15, links with arrow heads indicate links leading to inorder predecessors or inorder successors while other lines denote the usual links in a binary tree. Note that the links with arrows and the normal links indicate different relationship between nodes and the links are no longer used to describe only

"parent-child" relationship. Consequently, it must be understood whether the "left" or "right" link of a node "n" is leading to its children or to the inorder predecessor of "n" or to the inorder successor of "n". Two flags, "leftFlag" and "rightFlag" are to used per node to indicate the type of its "left" and "right" links. If "leftFlag" of a node is 0 then its "left" link leads to the left sub-tree of "n"; otherwise, the "left" link leads to the inorder predecessor of "n". Similarly, if "rightFlag" of a node "n" is 0 then the "right" link leads to the right sub-tree of "n"; otherwise, the "right" link leads to the in-order successor of "n". The "left" and "right" links leading to inorder predecessors or successors are called threads to distinguish them from the conventional links of a binary tree. The links are used as threads only when they would have pointed to empty sub-trees in a non-threaded binary tree.

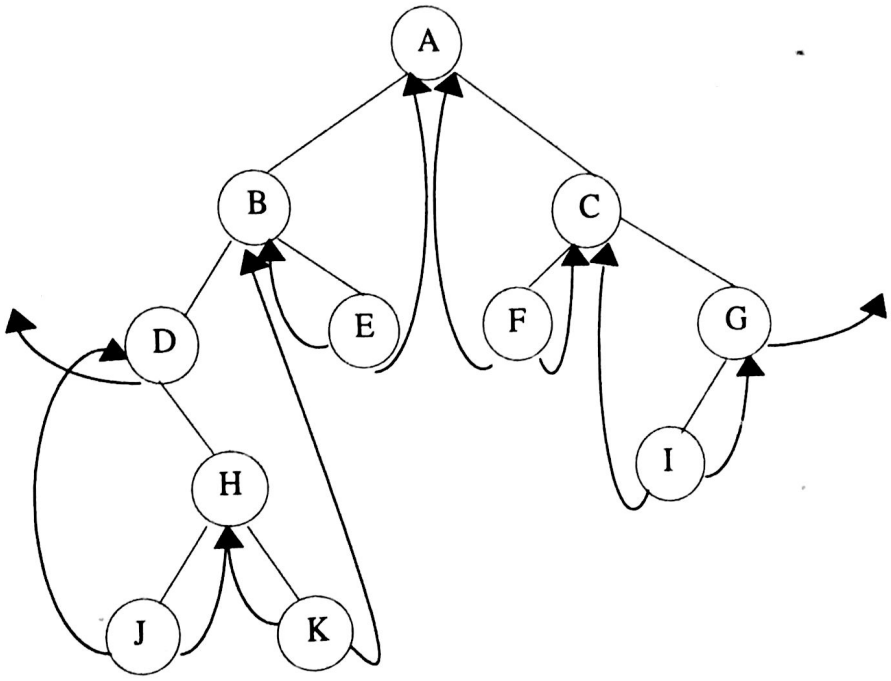
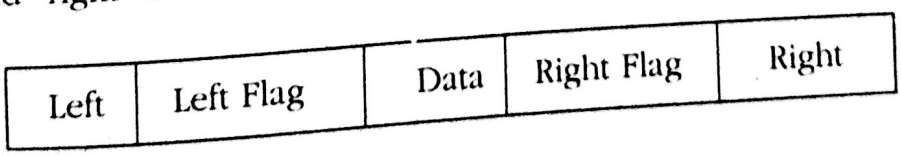


Figure 7.15 A threaded binary tree. Lines with arrows represent threads.

Every node in a threaded binary tree must contain the flags, as discussed, to indicate whether its "left" and "right" pointers are threads. Hence, the structure of a node in such trees contains two more fields in addition to the three fields: "data", "left" and "right" as shown in the following.



The type declaration for a threaded binary tree in "C" language may be done in the following way.

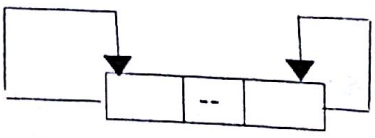
```
typedef struct bThreadNode
{
```

```
    int rightFlag, leftFlag;
    char data;
    struct bThreadNode * left, * right;
} bThreadNode;
```

Consider the tree of Figure 7.15. The lines with arrows denote threads. In a non-threaded binary tree replace these lines with arrows with links to NULL. The inorder traversal sequence of the nodes of this tree is

D, J, H, K, B, E, A, F, C, I, G

For instance, take the node "J" in the tree of Figure 7.15. Since the node "J" does not have a child, the "left and "right" pointers of this node are used as threads. The left thread is used to point to the node "D" which is the inorder predecessor of the node "J". And the right thread is used to point to the node "H" which is the inorder successor of the node "J". Consider the nodes "D" and "G" in this tree. The left and right links of these nodes are kept dangling because the node "D" does not have any inorder predecessor and the node "G" does not have any inorder successor. These pointers may be set to NULL. Another approach to get rid of such problems is to introduce a header node in a threaded binary tree as shown in the following diagram. The left and right pointers of the header node are treated as normal links (not threads) and are initialized to point to the header node itself.



The actual tree is set to be the left child of the header node. The "left" pointer of the first node and the "right" pointer of the last node in inorder sequence are set to the header node. The pointer representation of the tree of Figure 7.15 is shown in Figure 7.16.

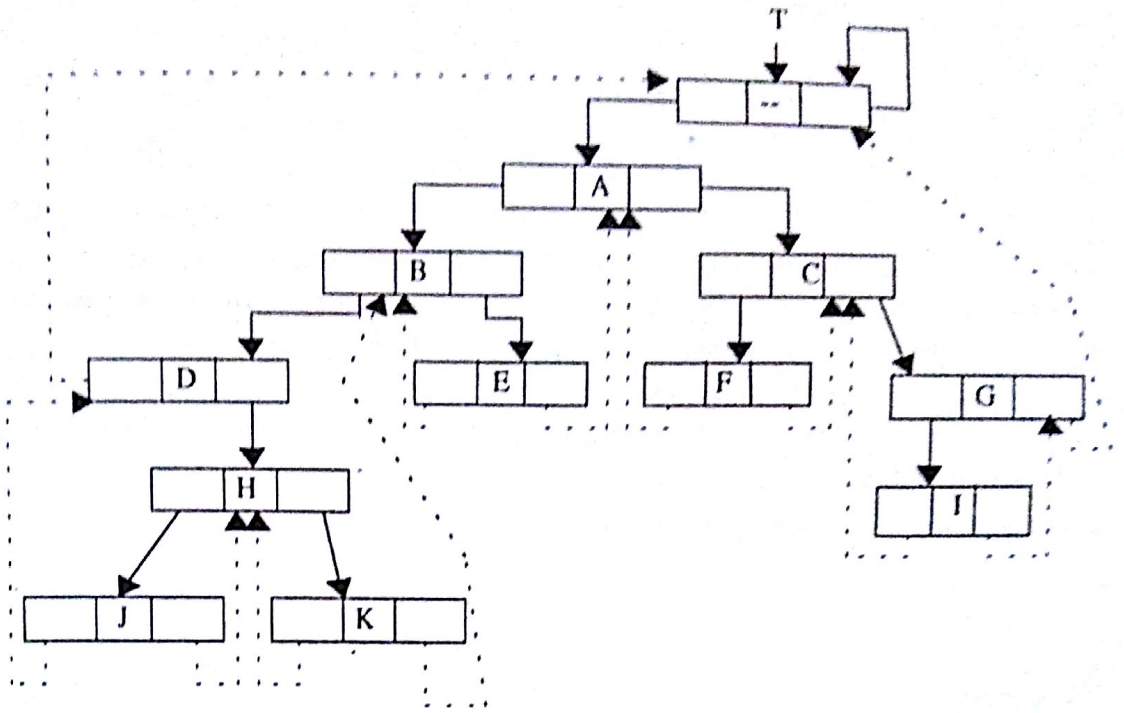


Figure 7.16 Pointer representation of a threaded binary tree including the header.

In the ensuing discussions, however, it will be assumed that the idea of the header node is not used. In stead, the left and right threads of the first and the last node in the inorder sequence are set to NULL.

It is interesting to find out how threads of all nodes can be initialized for any given unthreaded binary tree. It is also important to understand algorithms to insert a new node to a threaded binary tree or to delete a node from a threaded binary tree. Before these algorithms are described, advantages of a threaded binary tree are discussed. Observe that it becomes very easy to find out inorder successor of any node, "n", in a threaded binary tree. If the "rightFlag" of "n" is "1" then the node pointed to by its "right" pointer is the desired inorder successor. If the "rightFlag" of "n" is "0" then its inorder successor must be on the right sub-tree of "n". So, first go to the right child of "n". The first node in the inorder sequence of this sub-tree is the inorder successor of "n". The first node of the inorder sequence of any binary tree can be reached by traversing the left sub-tree from the root as long as the left sub-tree is non-empty. This leads to the following algorithm.

```
bThreadNode * inorderSuccessor(bThreadNode * n)
{
   if(n->rightFlag == 1)
         return (n->right);
   n = n->right;
   while(n->leftFlag == 0)
```

```
        n = n->left;
    return n;
}
```

A non-recursive algorithm for traversing a threaded binary tree in an inorder manner is very simple. For traversing any sub-tree rooted at "p", proceed towards left as far as possible following the "left" pointer. The last node "n" in this sequence is the first node to be visited for the tree rooted at "p". Next sub-tree to be traversed in the right child of "n" is its inorder successor "k". The algorithm just discussed to find the inorder successor of a node in a threaded binary tree can be used here to obtain "k". This step can be repeated as long as the right pointer of the node is not a thread pointng to NULL. The "C" function is presented next.

```
void inorderThreadedTree(bThreadNode * root)
{
    bThreadNode * x;
    x = root;
    while(x->leftFlag == 0)
            x = x->left;
    while(x !=NULL)
    {
            printf("%c ", x->data);
            /* The remaining part of the body of the while loop is */
            /* identical to the function "inorderSuccessor" */
            if(x->rightFlag == 1)
                    x = x->right;
            else
            {
                    x = x->right;
                    while(x->leftFlag == 0)
                            x = x->left;
            }
    }
}
```

Consider preorder traversal of a threaded binary tree. The root of the tree is visited. If there is a left child then the child is visited; otherwise, the right child is visited. This process continues until a leaf node is encountered. If the node is a leaf then

we need to go back to the first node whose right sub-tree has not been traversed. This can be done by traversing through parents of the leaf node until a node is reached which has a non-empty right sub-tree. It is amazing to note that such a node may be found by merely following the in-order successors of the leaf node. This is so because, if the inorder successor of a node "n" which does not have a right child is the node "k" then "k" must have been visited before "n" in preorder. Once again, if "t" is an inorder successor of "k" and "k" does not have a right child then "t" must have been visited before "k"(i.e. also before "n") in preorder. This reasoning can be continued until the computed inorder successor has a non-empty right sub-tree. Once an inorder successor is found which does have a right child, the right sub-tree of that node has to be visited in a preorder manner. This leads us to the following "C" function.

```
void preorderThreadedTree(bThreadNode * root)
{
   bThreadNode * x;
   x = root;
   while(x !=NULL)
   {
         printf("%c", x->data);
         if(x->leftFlag == 0)
               x = x->left;
         else if(x->rightFlag == 0)
               x = x->right;
         else
         {
               while(x && (x->rightFlag == 1))
                     x = x->right;
               if (x) x = x -> right;
         }
   }
}
```

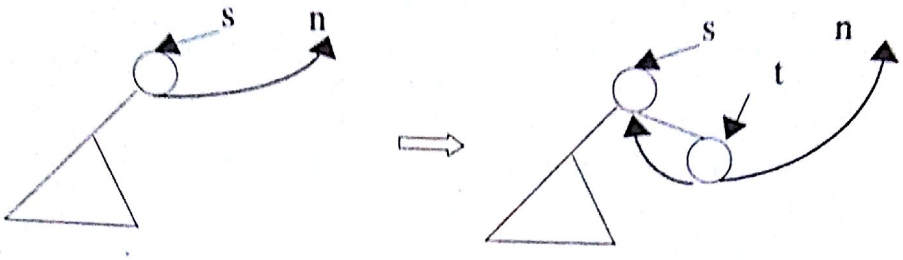
To conclude this section, it must be mentioned that traversing a threaded binary tree non-recursively in postorder fashion is also possible. However, this is more involved and explored in the exercises.

7.8.1 Insertion into a threaded binary tree

Suppose that a new node, "t", is to be inserted as the right of a node, "s", in a threaded binary tree. "s" may or may not have non-empty right sub-tree before

inserting "t" as the right child of "s". This gives rise to two cases which are discussed in the following.

Case 1: "s" does not have a non-empty right sub-tree as shown in Figure 7.17

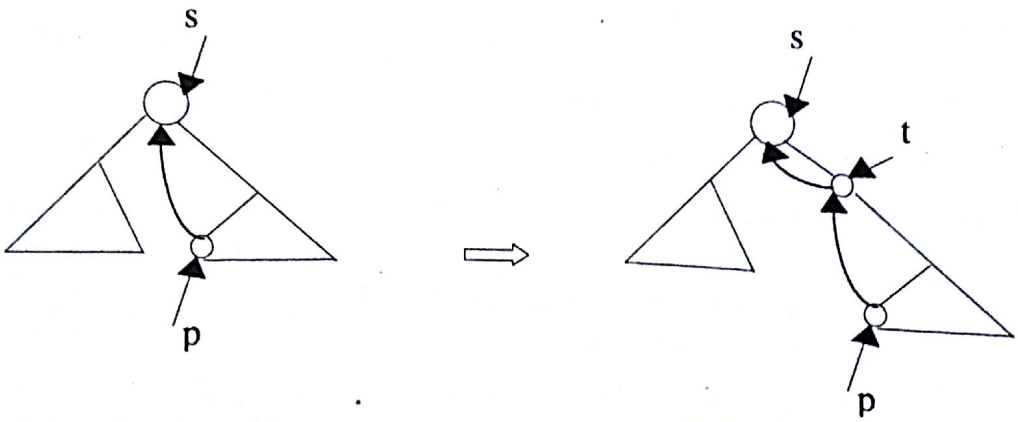


(a) Before insertion *(b) After insertion*

Figure 7.17 Illustration for insertion of a node (pointed to by "t") as the right child of a node pointed to by "s" where the right pointer of "s" is a thread before insertion. All threads are denoted by dotted arrows.

The "right" link of "s" is a thread pointing to "n" before "t" is inserted to the right of "s". After insertion, "s" becomes the inorder predecessor of "t" as both the right and left sub-trees of "t" are empty. Also, "n" becomes the inorder successor of "t".

Case 2: "s" has a non-empty right sub-tree.



(a) Before insertion *(b) After insertion*

Figure 7.18 Illustration for insertion of a node (pointed to by "t") as the right child of a node pointed to by "s". Before insertion, "s" has a right child and "p" is the inorder successor of "s".

Then there must be a node, "p", which is the inorder successor of "s". and "p" lies on the right sub-tree of "s". The "left" link of "p" must be a thread pointing to "s" as "s" is the inorder predecessor of "p". If "t" is made a right child of "s" then the right sub-tree T of "s" must be made a right sub-tree of "t". Left sub-tree of "t" will be empty and must be made a thread pointing to "s", the inorder predecessor of "t". The inorder successor of "t" is "p". So, the "left" link of "p" has to be set to point to "t".

The "C" function for the above procedure is described in the following.

```
void insertRighThreadedTree(bThreadTree * s, bThreadTree * t)
{
   bThreadTree * x;
   t->right = s->right;
   s->right = t;
   t->rightFlag = s->rightFlag;
   t->leftFlag = 1; /* thread */
   t->left = s;
   if(t->rightFlag == 0)
   {
            x = inorderSuccessor(t),
            x->left = t;
   }
}
```

A node can similarly be inserted as a left child of a node in a threaded binary tree.

7.9 Binary Search Tree

Binary search trees form an important sub-class of binary trees. In an ordinary tree, the elements are not ordered in any way. However, data are ordered in many applications. Information in the nodes in a binary search tree is maintained in some order. Usually, the nodes in a binary search tree represent some records and these records are ordered. Records are ordered on some key properties. Therefore, the information part of a node in a binary search tree is assumed to have some key attributes. For the purpose of studying binary search trees one may just concentrate on these keys. Thus, the information field of nodes in a binary tree will be referred to as just keys.

A binary search tree is a binary tree which is either empty or in which the following criteria are satisfied.

1. All keys of the left sub-tree of the root are "less than" the key in the root.
2. All keys of the right sub-tree of the root are "greater than" the key in the root.
3. The left and right sub-trees of a binary search tree are binary search trees once again.

The definition is recursive. An example of a binary search tree is given in Figure 7.19 (a).

8.4 B-Trees

The nodes in binary search trees and in AVL trees typically contain only one record. The major concern has been reduction of height of the tree for efficient operations. Sometimes data which are organized as search trees are not resident in primary memory and they are stored in secondary memory such as hard disks. Time required to access secondary memory is a thousand times more than time required to access primary memory. Also, blocks (and not words) are usually accessed from secondary memory at one shot. A block may, naturally, contain a number of data records. And in operations involving secondary memory, the main reason of inefficiency is heavy access to secondary memory. An efficient data organization in these scenarios would attempt to store multiple records in one block and to make multi-way decisions to decide the next course of action. Thus, multi-way trees are more appropriate for such situations.

Multi-way trees are generalizations of binary trees. A multi-way tree of order "m" is a tree where each node may have at most "m" children. If a node has "k" number of children ($k \leq m$) then that node must have exactly (k-1) number of keys (or records). The children partition all the keys into "k" sub-sets. Some of these sub-sets may be empty in which case the corresponding children are also empty.

Definition

A B-Tree is a multi-way search tree of order "m" such that the following properties hold.

1. All non-leaf nodes except the root must have at least $\lceil m/2 \rceil$ children and at most "m" children.
2. If a node has "t" number of children then it must have (t-1) number of keys.
3. The keys of a node "x" are sorted in non-decreasing order. If a node "x" has (t-1) keys and a key is denoted by $key_i(x)$ for i = 0 to (t-2) then,
$$key_0(x) \leq key_1(x) \leq \ldots \leq key_{t-2}(x)$$
4. The keys in a node separate the ranges of keys stored in each child of that node. Let "t" children of a node "x" be denoted by $child_i(x)$ for i = 0 to (t-1) and let k_i denote any key stored in a sub-tree $child_i(x)$. Then,
$$k_0 \leq key_0(x) \leq k_1 \leq key_1(x) \leq \ldots \leq k \leq key_{t-1}$$
5. The root may have at most "m" non-empty children but may have as few as two children if the root is not itself a leaf node. If the root is also a leaf node then it may not have any child.
6. All leaf nodes are on the same level, which defines the height of the tree.

An example of a B-tree of order "4" is shown in Figure 8.23.

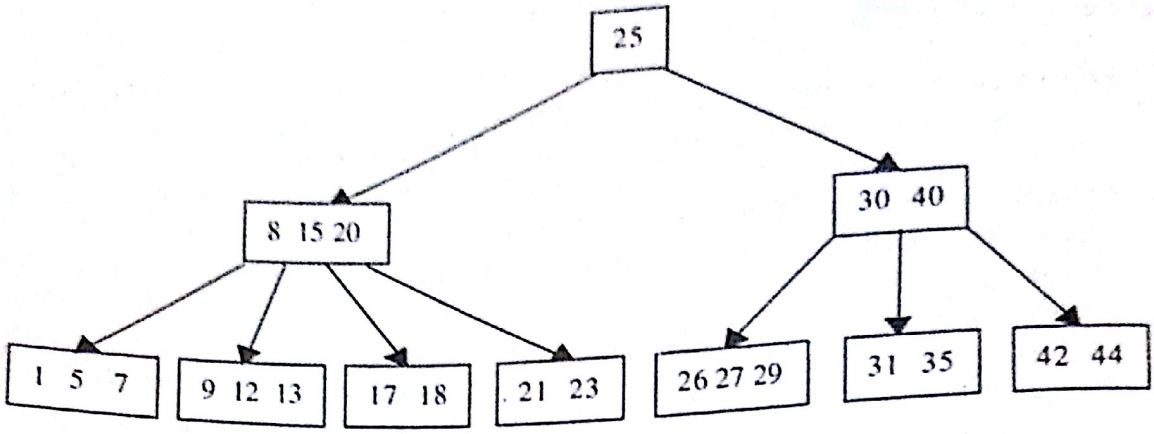


Figure 8.23 A sample B-tree of order "4".

It can be easily seen that a 2-3 tree is a B-tree of order "3". The structure required to represent one node of a B-tree of order "m" is very similar to the one defined to represent a 2-3 tree. The structure of a B-tree is presented in the following. It is assumed that the key values are integers.

```
typedef struct BTreeNode
  {
      int noOfKeys;
      struct BTreeNode *children[MAX_NO_CHILDREN + 1];
      int key[MAX_NO_CHILDREN];
      struct BTreeNode *parent;
  }BTreeNode;
```

When a variable of type BTreeNode is created, its "noOfKeys" should be initialized to 0 and all pointers of the array "children" should be initialized to NULL. The pointer "parent" is added to store the pointer to the parent node of a B-tree whenever required. This field should also be initialized to NULL.

Time complexities of most of the operations on a B-tree are proportional to its height. The following property of a B-tree analyzes the worst case height of a B-tree.

Property 1: The height, "h", of a B-tree of order "m" (m > 1) is related to the number of keys "n" stored in that B-tree according to the following inequality.

$$h \leq \log_{\lceil m/2 \rceil}((n+1)/2)$$

Proof: For a m-way B-tree of height, "h", the number of its nodes are minimized if the root contains just one key and all other nodes contain exactly ($\lceil m/2 \rceil - 1$) keys. Let "t" denote $\lceil m/2 \rceil$. In such a tree, the number of keys stored in level "0" is "1"; the number of keys stored in level "1" is (t-1)*2 = 2(t-1); the number of keys stored in level "2" = 2*t*(t-1). Thus, the minimum number of keys stored in a B-tree of height "h" is given by the following expression.

$1 + 2(t-1) + 2.t.(t-1) + 2.t.t(t-1) + \ldots + 2t^{h-1}(t-1)$

i.e., $1 + 2(t-1)\sum_{i=1..h}(t^{i-1})$

i.e., $1 + 2(t-1)(t^h-1)/(t-1)$

i.e., $1 + 2(t^h-1)$

i.e., $2t^h - 1$

i.e., $n \geq 2t^h - 1$

i.e., $h \leq \log_t((n+1)/2)$

This concludes the proof. This also means that as "m" increases, the height of the B-tree of order "m" decreases. The choice of a suitable value of "m" is very critical for the efficiency of several operations on a B-tree. In case of B-trees, the nodes typically contain many key values (basically indices to actual records) and these nodes are stored in secondary memory. Thus, the nodes are usually stored in one block so that a node can be entirely read from the secondary memory by one "read" operation. So, the maximum capacity of a node is often dictated by the size of a block in a particular machine.

8.4.1 Searching a B-tree

Searching in a B-tree is very similar to searching in a 2-3 tree. If a search is performed for a key "k" from the root "x" of a B-tree then the i-th child of "x" is chosen for next search if "k" is larger than all $key_j(x)$ for j = 0 to (i-1). If "i" is less than "t", the number of children of "x", then "k must also be less than $key_i(x)$. The search procedure then operates recursively on a child of "x" unless searching halts unsuccessfully at a null child or searching terminates successfully at any particular node. Thus, this procedure is a straight-forward generalization of the search procedure for a 2-3 Tree. The "C" function for searching for a key "k" in a B-tree rooted at the node "root" is presented next. The function returns a pointer to the node where the key, "k", is found in case of a successful search. In addition, the formal parameter, "*position", is set to the value of "i" so that key[i] is equal to "k". For unsuccessful search, the function returns NULL and sets "*position" to the largest value of "i" so that key[i] < k.

```
BTreeNode * BTreeSearch(BTreeNode * root, int k, int *position)
{
    int i = 0;
    while((i < root->noOfKeys) && (k < root->key[i]))
            i++;
    if((i < root->noOfKeys) && (k == root->key[i]))
    {
            *position = i;
            return root;
```

```
    }
    if (root->children[i] == NULL)
    {
            *position = i;
            return NULL;
    }
    return(BTreeSearch(root->children[i], k, position));
}
```

It is easy to convert this function to a non-recursive one. One can also easily verify that the worst case time complexity of the above function is equal to the height, "h", of the B-tree, which is always less than or equal to $\log_{\lceil m/2 \rceil}((n+1)/2)$. This also implies that the number of disk accesses required by the search algorithm is "h" in the worst case. It is assumed in the above analysis that the time for searching for a key value in a node is insignificant. However, if "m" is too large (making "h" too small) then the time complexity of searching for a key inside a node may become dominant. So, there is always a trade off. It can be shown that the time complexity of the function "BTreeSearch" decreases with increase in "m" upto a certain value of "m". Beyond that value of "m", even if "m" increases, the time required by the search function also increases.

8.4.2 Insertion of a key to a B-tree

The procedure to insert a given key in a B-tree essentially involves a search to find out the correct sub-tree to insert to. Finally, a point is reached where the key is to be inserted into a leaf node. If the leaf node is not yet full then adding another key does not create any problem and the algorithm ends there. For example, consider the B-tree of Figure 8.24(a) where a key "14" is to be inserted.

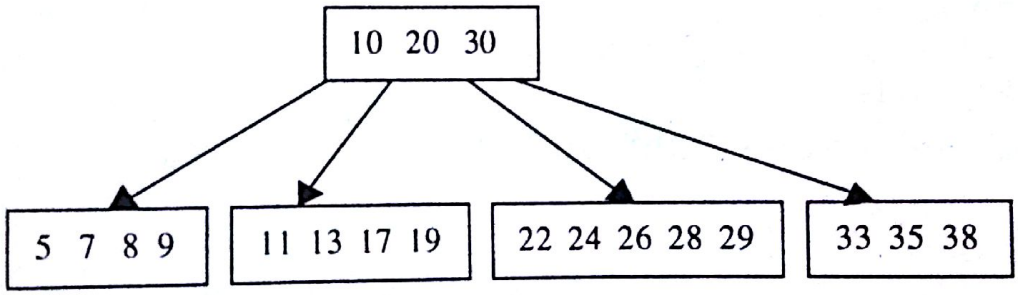


Figure 8.24(a) A sample B-tree of order "4" to which a key "14" is to be inserted.

To search for "14" from the root, it is found that the second sub-tree of the root is to be searched as $10 \leq 14 \leq 20$. Going to the second sub-tree, it is found that the node is a leaf node and hence "14" is to be inserted to this leaf node. Since the leaf node is not yet full, it is easy to put "14" in the array "key" of that node producing a B-Tree as shown in Figure 8.24(b).

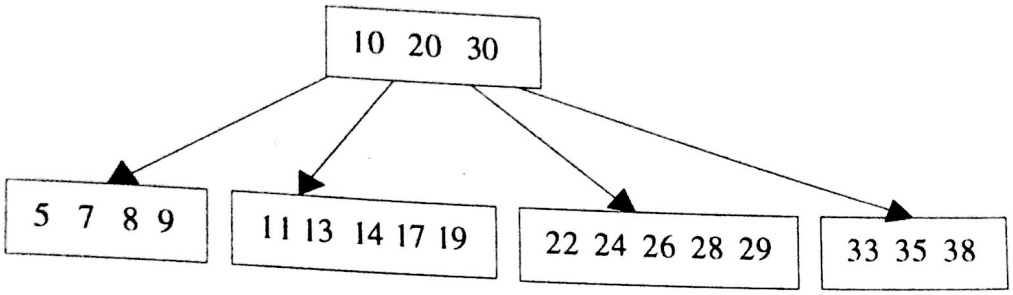
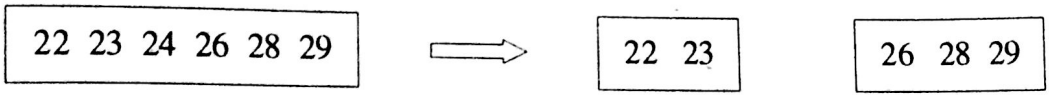


Figure 8.24(b) The B-tree after key value "14" is added to the B-tree of Figure 8.24(a).

However if "23" is to be inserted in the tree of Figure 8.24(a) then an insertion to a "full" leaf node (the 3rd child of the root) is to be done. Since, number of keys in node can not be more than "5", the leaf node must be split into two as shown in the following.



Now the middle key (say "24") is to be taken out of the left half and has to be pushed up to the root so that the left half becomes the third child of the root and the right half becomes the 4th child of the root. Thus, the new tree becomes as shown in Figure 8.24(c).

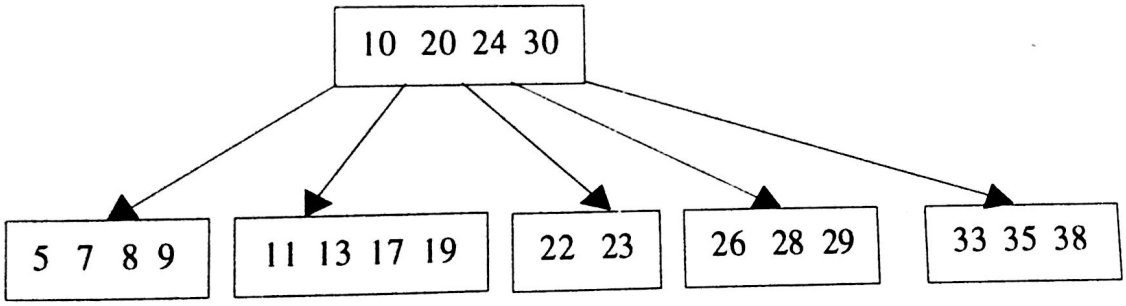


Figure 8.24(c) The B-tree after a key value "23" is added to the B-tree of Figure 8.24(a).

While trying to add "24" to the parent node, it may be found that the parent node is also full. In that case, the parent must be split again around its median key. And this process propagates upwards. The process stops either at the root or at a non-full node along the path from the point of insertion to the root. The following example elaborately illustrates the process of insertion of a key to a B-tree of degree "2". Consider a B-tree of order "4" as shown in Figure 8.25(a).

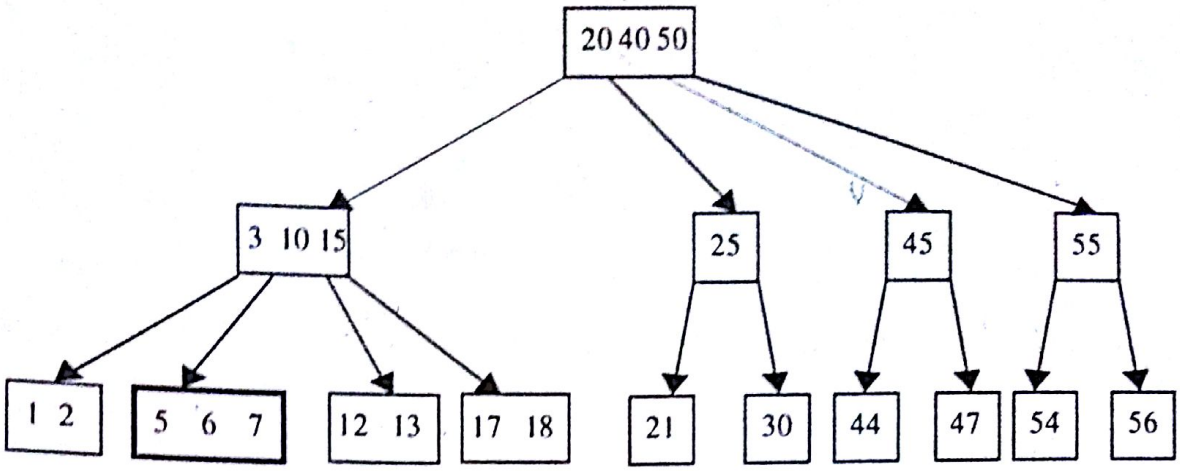


Figure 8.25(a) A sample B-tree of order "4" where "8" is to be inserted. The leaf node where insertion begins is the node (5, 6, 7), which is enclosed in a rectangle whose border is boldfaced.

Let a new key "8" be inserted to it. The first step to insertion is to find out the leaf node to which "8" is to be added. This can be accomplished by calling the function "BTreeSearch". This function will return a pointer to the leaf node and the value of "*position" would give the position where "8" is to be added in the leaf node. In this example, The rectangular node which is bordered by bold lines in Figure 8.25(a) is the node where "8" is to be added. But this node is full. So, the number of keys in this node becomes "4"after adding "8" to the array "key" of this node. As a result, the node is split after ($\lceil 4/2 \rceil$ - 1)-th (i.e., the first) key value. Thus, the original node retains the first ($\lceil m/2 \rceil$ - 1) key values and the first $\lceil m/2 \rceil$ children. A new node is created to hold the last (m-$\lceil m/2 \rceil$) key values and the last (m-$\lceil m/2 \rceil$ + 1)children. The key at index$\lceil m/2 \rceil$ is moved upwards along with a pointer to the newly created node. If these operations are applied to the leaf node in our example then the array "key" would contain the keys 5, 6, 7, 8 after inserting "8". So, the node is split. The original node contains the first ($\lceil 4/2 \rceil$ - 1) (i.e., one) element which is "5". A new node is created which would contain the last (4 - $\lceil 4/2 \rceil$) elements. So, the new node contains the keys "7" and "8". The median "6" is pushed up to be inserted into the parent of the original node. This situation is depicted in Figure 8.25(b).

```
                pos = pos+1;
                continue;
            }
            if (pos > 0)
            {
                left = p->children[pos];
                for all j = 0 to x->noOfKeys-1
                    left->key[j + ⌈m/2⌉] = x->key[j];
                for all j = 0 to left->noOfKeys
                    left->children[j + ⌈m/2⌉ - 1] = x->chldren[j];
                left->key[⌈m/2⌉-1] = p->key[pos];
                left->noOfkeys = 2*⌈m/2⌉ - 2;
                x = p;
                continue;
            }
        }
    }
```

The worst case time complexity analysis of deletion algorithm is very similar to the analysis of the insertion procedure. The number of disk accesses to delete a key from a B-tree of height "h" is (3h + 1) in the worst case.

8.5 Priority Queues

In many applications, data to be processed are not fully available. A set of records is collected, being processed and in the mean time, some more records are collected and so on. An appropriate data structure for such applications is one that supports operations to insert and delete elements efficiently. Two such data structures have already been introduced: stacks and queues. In a stack, the latest or the newest element is deleted. In a queue, the oldest element is deleted. It may also be useful to process not the oldest or the newest but the element in the current set with the highest priority. A data structure that supports efficient insertions of a new element and deletions of elements with the highest priority is known as "priority queue". In fact, a priority queue may be viewed as a generalization to "stack" or "queue" data structure. Priority queues are used for implementing job scheduling by the operating system where jobs with higher priorities are to be processed first. Another application of priority queues is simulation systems where priority might correspond to event times.

It is instructive to list down different possible operations on the priority queue data

structure. Assume that the data are a set of records where each record contains a numerical value known as its priority. Then, the following operations are to be supported.

Construct a priority queue with "N" given elements.

Add a new element to a priority queue.

Remove the element with the highest priority from a priority queue.

Replace the largest element with a new element in a priority queue.

Change the priority of an element in a priority queue.

Remove a specified element.

Join two priority queues into one.

The above descriptions of various operations on a priority queue provide a good example of the definition of an abstract data type. In fact, a closer scrutiny of these operations would reveal that many of them can be defined in terms of others. For example, the "Replace" operation is equivalent to an "Add" operation followed by a "Remove" operation. The "construct" operation can be implemented by repeated use of "Add" operation. The "change" operation is actually equivalent to "Remove" operation followed by an "Add" operation. This abstract data type can be implemented in many ways. Different implementations will lead to different performance characteristics for various operations. The simplest implementation of a priority queue can be done by using an array or using an ordered list represented by an array. However, a more efficient implementation is possible by representing a priority queue using an advanced data structure known as "heap".

8.5.1 Array implementation of a priority queue

In the ensuing discussion of priority queues, elements of the priority queue are assumed to contain only their priority values and hence, the type of elements is assumed to be integer only. The most rudimentary way to implement a priority queue of "n" elements is to store these elements in an array of integers in a completely unordered manner. Suppose that the name of the array to store the elements of the priority queue be "Elements". The algorithms for the operations "Construct", "Add", "Remove" are quite straight forward in this case and are presented in the following.

The "Construct" operation assumes that the elements of the priority queue to be constructed are available in an array, "X", and the number of elements in the array "X" is "m". Thus the algorithm involves merely copying the elements of one array "X" to the elements of another array, "Elements". In all functions, the array "Elements" and the number of elements in the priority queue, "n", are assumed to be global. It is further assumed that "n" elements of the priority queue are stored from index 0 to (n-1) in the array "Elements".

```
void Construct (int X[], int m)
```

```
    {
        for (n = 0; n < m; n++)
        {
            Elements[n] = X[n];
        }
    }
```

The "Add" operation is equally simple and involves nothing but incrementing "n" and setting the value of Elements[n]. In the following function, "val" holds the new value to be added to the queue.

```
void add (int val)
{
    Elements[n++] = val;
}
```

The time complexities of the above two functions are optimal. But such an organization of values in a priority queue leads to a very inefficient implementation of "remove" operation. To remove the largest element in the priority queue, the largest element is to be searched and Elements[n-1] is to be exchanged with the largest element and finally "n" is to be decremented. This leads to the following function.

```
int remove( )
{
    int i, max, val;
    max = 0;
    for (i = 1; i < n; i++)
        if (Elements[i] > Elements[max])
            max = i;
    val = Elements[max];
    Elements[max] = Elements[--n];
    return val;
}
```

The performance of the "remove" function may be improved if the elements of the priority queue are maintained in ascending or descending order of their priority values. Assuming that elements are stored in ascending order of their values, "remove" operation simply retrieves Elements[n-1], decrements "n" and then returns the retrieved value. But the modified algorithms to insert items into a priority queue or to construct a priority queue would become time consuming to support such an

ordered list representation of a priority queue.

8.5.2 Heaps and Priority queues

Heap is a data structure which can be used to support the operations defined for a priority queue in an efficient way. Heaps can also be used to efficiently sort a list of elements as will be discussed in chapter 9. By definition, a heap is a complete binary tree of "n" nodes satisfying "heap property". The "heap" property is that the key value of any node must be greater then (or equal to) the key values of its children. Thus, the root node of a heap contains the largest element. A heap may be recursively defined to be a complete binary tree so that

(i) The root contains the largest element and

(ii) Both the left and the right sub-trees of the root are heaps once again.

The notion of "complete" binary trees must be clarified at this point. A "full" or "completely full" binary tree is one where all leaf nodes are at the same level and all non-leaf nodes must have two children. In a "complete" binary tree, all leaf nodes must be situated at adjacent levels, "m" and (m-1), and the nodes (if any) to the left of all non-leaf nodes at level (m-1) must have two children except the last non-leaf node from left at level (m-1) which must have exactly one left child.

It has already been discussed in Chapter 7 that complete (or near complete) binary trees may be implemented by a single one-dimensional array where the nodes of a tree are numbered level by level and form left to right in the same level. If this numbering scheme starts from index "1" then the children of the i-th node in a tree are located at "2i" and (2i+1) indices of the array. Similarly, the parent of any node at the i-th index is located in the index (i/2) of the array.

Most of the operations on priority queues would first modify the structure of the underlying heap. For instance, "add" operation on a priority queue will result in insertion of an element to the heap . Heap property of some node in the original heap may be violated due to such a modification of the heap. A restructuring is, thus, necessary to restore the tree to a heap. Take, for instance, the heap of 10 elements as shown in Figure 8.28.