The function inorder() is-

```
void inorder(struct node *ptr)
{
        int i;
        if(ptr!=NULL)
        {
                for(i=0; i<ptr->count; i++)
                {
                        inorder(ptr->child[i]);
                        printf("%d\t",ptr->key[i+1]);
                }
                inorder(ptr->child[i]);
        }
}/*End of inorder()*/
```

This function prints the inorder traversal of the B-tree. In inorder traversal, key is processed after its left subtree and before its right subtree. The left subtrees of the keys are processed in the first recursive call. The rightmost subtree of the current node is traversed by the second recursive call outside the for loop.

The function display() is-

```
void display(struct node *ptr,int blanks)
{
        if(ptr)
        {
                int i;
                for(i=1; i<=blanks; i++)
                        printf(" ");
                for(i=1; i<=ptr->count; i++)
                        printf("%d ",ptr->key[i]);
                printf("\n");
                for(i=0; i<=ptr->count; i++)
                        display(ptr->child[i],blanks+10);
        }
}/*End of display()*/
```

6.22 B+ tree

A disadvantage of B tree is inefficient sequential access. If we want to display the data in ascending order of keys, we can do an inorder traversal but it is time consuming, let us see the reason for it. While doing inorder traversal, we have to go up and down the tree several times, i.e. the nodes have to be accessed several times. Whenever an internal node is accessed, only one element from it is displayed and we have to go to another node. We know that each node of a B tree represents a disk block and so moving from one node to another means moving from one disk block to another which is time consuming. So for efficient sequential access, the number of node accesses should be as few as possible.

B+ tree which is a variation of B tree, is well suited for sequential access. The two main differences in B tree and B+ tree are-

(i) In B tree, all the nodes contain keys, their corresponding data items (records or pointers to records), and child pointers but in B+ tree the structures of leaf nodes and internal nodes are different. The internal nodes store only keys and child pointers while the leaf nodes store keys and their corresponding data items. So the data items are present only in the leaf nodes. The keys in the leaf nodes may also be present in the internal nodes.

(ii) In B+ tree, each leaf node has a pointer that points to the next leaf node i.e. all leaf nodes form a linked list.

The figure 6.163 shows a B tree and the figure 6.164 shows a B+ tree containing the same data.

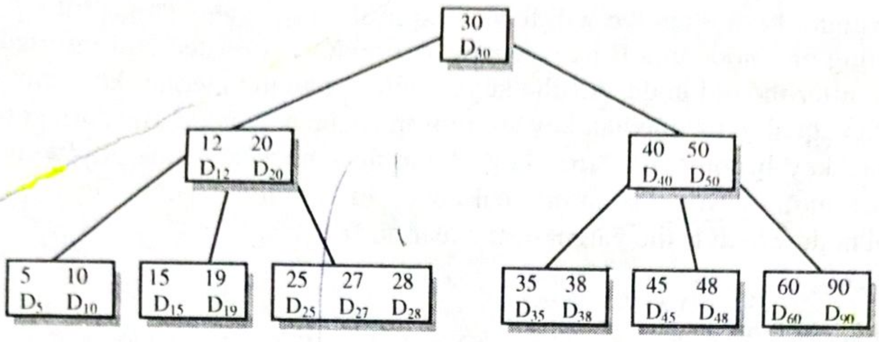
Figure 6.163

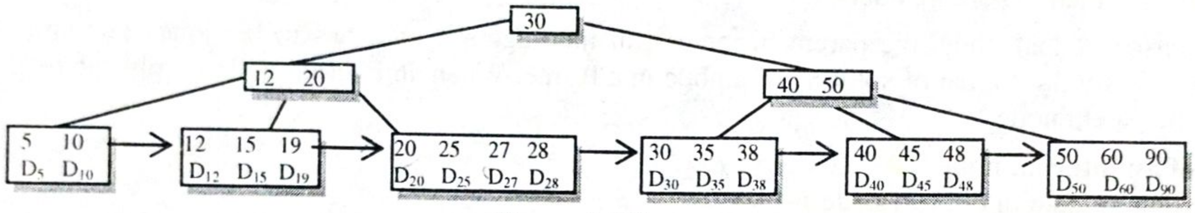
Figure 6.164

The alphabet 'D' with subscript shown under the key value represents the data item. While discussing B tree we had not shown this data item in the figures so that the figures remain small.

In B+ tree, the internal nodes contain only keys and pointers to child nodes, while the leaf nodes contain keys, data items and pointer to next leaf node. The internal nodes are used as index for searching data and so they are also called index nodes. The leaf nodes contain data items and they are also called data nodes. The index nodes form an index set while the data nodes form a sequence set. All the leaf nodes form a linked list and this feature of B+ tree helps in sequential access i.e. we can search for a key and then access all the keys following it in a sequential manner. Traversing all the leaves from left to right gives us all the data in ascending order. So both random and sequential accesses are simultaneously possible in B+ tree.

6.22.1 Searching

In B tree our search terminated when we found the key, but this will not be the case in B+ tree. In a B+ tree, it is possible that a key is present in the internal node but is not present in the leaf node. This happens because when any data item is deleted from the leaf node, the corresponding key is not deleted from the internal node. So presence of a key in an internal node does not indicate that the corresponding data item will be present in the leaf node. Hence the searching process will not stop if we find a key in an internal node but it will continue till we find the key in the leaf node. The data items are stored only in the leaf nodes, so we have to go to the leaf nodes to access the data.

Suppose we want to search the key 20 in the B+ tree of figure 6.164. Searching will start from the root node so first we look at the node [30], and since 20 < 30, we'll move to left child which is [12, 20]. The key value is equal to 20 so we'll move to the right child which is the leaf node and there we find the key 20 with its data item.

B+ tree supports efficient range queries i.e. we can access all data in a given range. For this we need to search the starting key of the range and then sequentially traverse the leaf nodes till we get the end key of the range.

6.21.2 Insertion

First a search is performed and if the key is not present in the leaf node then we can have two cases depending on whether the leaf node has maximum keys or not.

If the leaf node has less than maximum keys, then key and data are simply inserted in the leaf node in ordered manner and the index set is not changed.

If the leaf node has maximum keys, then we will have to split the leaf node. The splitting of a leaf node is slightly different from splitting of a node in a B tree. A new leaf node is allocated and inserted in the sequence set(linked list of leaf nodes) after the old node. All the keys smaller than the median key remain in the old leaf node, all the keys greater than equal to the median key are moved to the new node, the corresponding data items are also moved. The median key becomes the first key of the new node and this key(without data item) is copied(not moved) to the parent node which is an internal node. So now this median key is present both in the leaf node and in the internal node which is the parent of the leaf node.

Splitting of a leaf node
keys < median remain in old leaf node
keys >= median go to new leaf node
Median key is copied to parent node.

If after splitting of leaf node, the parent becomes full then again a split has to be done. The splitting of an internal node is similar to that of splitting of a node in a B tree. When an internal node is split the median key is moved to the parent node.

Splitting of an internal node
keys < median remain in old leaf node
keys > median go to new leaf node
Median key is moved to parent node.

This splitting continues till we get a non full parent node. If root node is split then a new root node has to be allocated.

Suppose we have to insert data with keys 42 and 24 in B+ tree of figure 6.164.

The key 42 can be simply inserted in the leaf node [40, 45, 48]. After inserting 24 in the tree, we get an overflow leaf node [20, 24, 25, 27, 28] which needs to be splitted. A new leaf node is allocated and keys 25, 27, 28 with data items are moved to this node. The median key 25 is copied to the parent node and is present in the leaf node also.

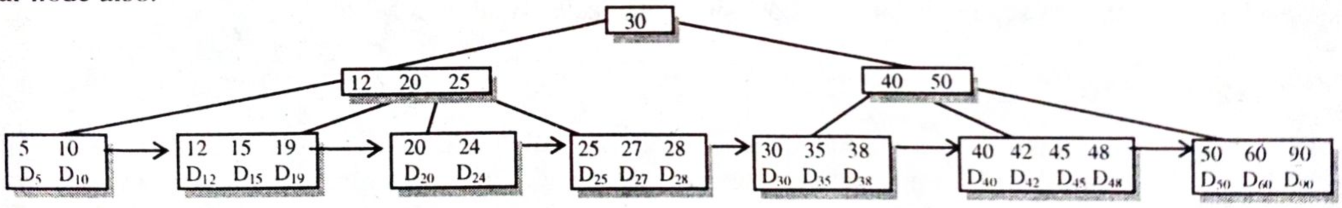
30
12 20 25
40 50
5 10 D_5 D_{10}
12 15 19 D_{12} D_{15} D_{19}
20 24 D_{20} D_{24}
25 27 28 D_{25} D_{27} D_{28}
30 35 38 D_{30} D_{35} D_{38}
40 42 45 48 D_{40} D_{42} D_{45} D_{48}
50 60 90 D_{50} D_{60} D_{90}

Figure 6.165

6.21.3 Deletion

First a search is performed and if the key is present in the leaf, then we can have two cases depending on whether the leaf node has minimum keys or more than that.

If the leaf node has more than minimum elements then we can simply delete the key and its data item, and move other elements of the leaf node if required. In this case, the index set is not changed i.e. if the key is present in any internal node also then it is not deleted from there. This is because the key still serves as a separator key between its left and right children.

If the key is present in a leaf which has minimum number of nodes then we have two cases-

(A) If any one of the siblings has more than minimum nodes then a key is borrowed from it and the separator key in the parent node is updated accordingly.

If we borrow from left sibling then, then rightmost key(with data item) of left sibling is moved to the underflow node. Now this new leftmost key in the underflow node becomes the new separator key.

If we borrow from right sibling then, then leftmost key(with data item) of right sibling is moved to the underflow node. Now the key which is leftmost in right sibling becomes the new separator key.

(B) If both siblings have minimum nodes then we need to merge the underflow leaf node with its sibling. This is done by moving the keys(with data) of underflow leaf node to the sibling node and deleting the underflow leaf node. The separator key of the underflow node and its sibling is deleted from the parent node, and the corresponding child pointer in parent node is also removed.

The merging of leaf nodes may result in an underflow parent node which is an internal node. For internal nodes borrowing and merging is performed in same manner as in B tree.

(i) Delete data with keys 12, 38, 40, 50 from B+ tree of figure 6.164.

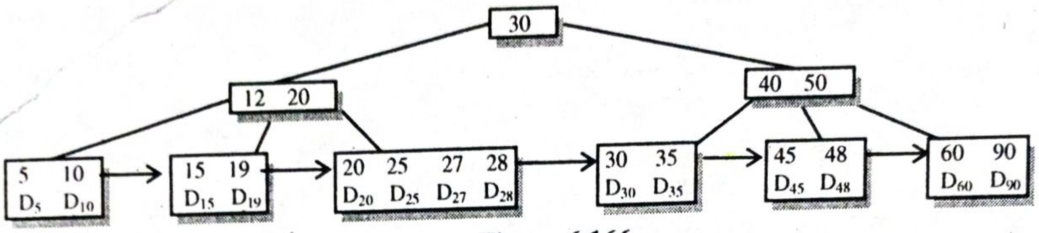

Figure 6.166

(ii) Delete 15 from B+ tree of figure 6.166(borrow from right sibling).

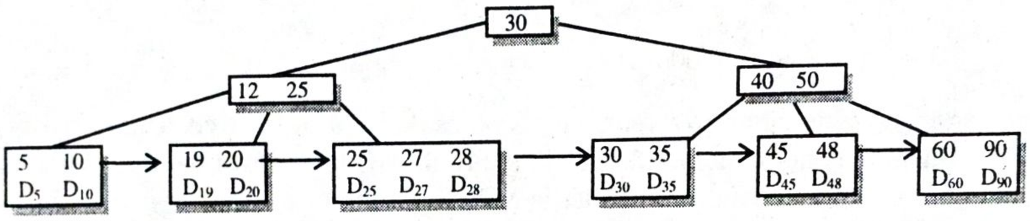

Figure 6.167

(iii) Delete 48 from B+ tree of figure 6.167.

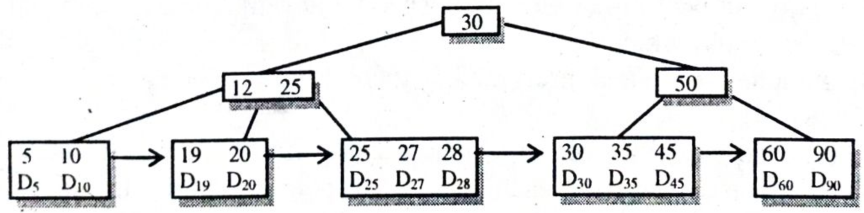

Figure 6.168

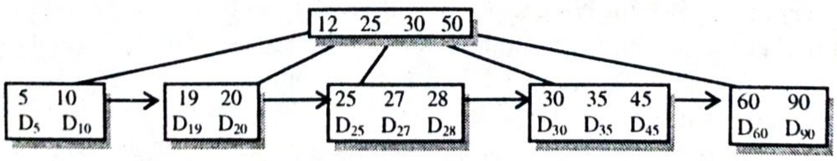

Figure 6.169

6.22 Digital Search Trees

Digital search tree is a binary tree in which a key and data pair is stored in every node, and the position of keys is determined by their binary representation.

The procedure of insertion and searching is similar to that of binary search tree, but with a small difference. In binary search tree, the decision to move to left or right subtree was made by comparing the given key with the key in the current node, while here this decision is made by a bit in the key. The bits in the given key are scanned from left to right and when we have 0 bit we move to the left subtree and when we have a 1 bit we move to the right subtree.