Module 3

Multilevel Indexing and B- Trees

R. Bayer and E. McCreight announced B-Trees to the world in 1972.

Why B-Trees came into existence :

When index file is too large, it has to be stored on secondary storage. It has mainly two problems –

- → Searching the index using binary search requires multiple access to secondary memory.
- → Insertion and deletion must be done to secondary memory and sorting of data is to be done.

Indexing with Binary Search Tree

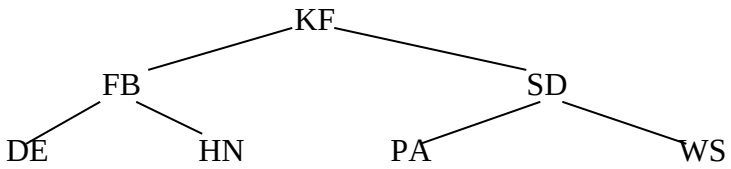
To overcome the 2nd problem, the keys can be represented in Binary Search Tree.

If the keys are in sorted list, the middle element is taken as the root and a balanced tree is formed.

Eg: If the list is –

DE, FB, HN, KF, PA, SD, WS.

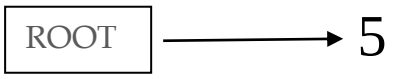
If KF is taken as root, a balanced tree is obtained,



Using tree structure - there is no problem of sorting the file (when new records are added). The new key is simply linked to the appropriate leaf node.

In a Binary Search Tree, each node will have three informations – key value, left link, right link

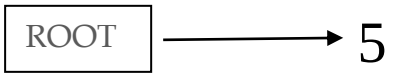
The above tree is represented in memory as follows:



	KEY	LEFT CHILD	RIGHT CHILD
0	**FB**	**2**	**4**
1	**PA**		
2	**DE**		
3	**SD**	**1**	**6**
4	**HN**		
5	**KF**	**0**	**3**
6	**WS**		

Suppose a key 'XF' is added to the tree, the key will be attached to the rightmost subtree ie. As the right child of 'WF'.

Record contents for a linked representation of the binary tree will be –



	KEY	LEFT CHILD	RIGHT CHILD
0	**FB**	**2**	**4**
1	**PA**		
2	**DE**		
3	**SD**	**1**	**6**
4	**HN**		
5	**KF**	**0**	**3**
6	**WS**		**7**
7	**XF**		

As key are added to the Binary Search Tree, it may become imbalanced. **A tree is balanced, when the difference in height of the shortest path to the tree and the longest path of the tree is not more than one level.**

or

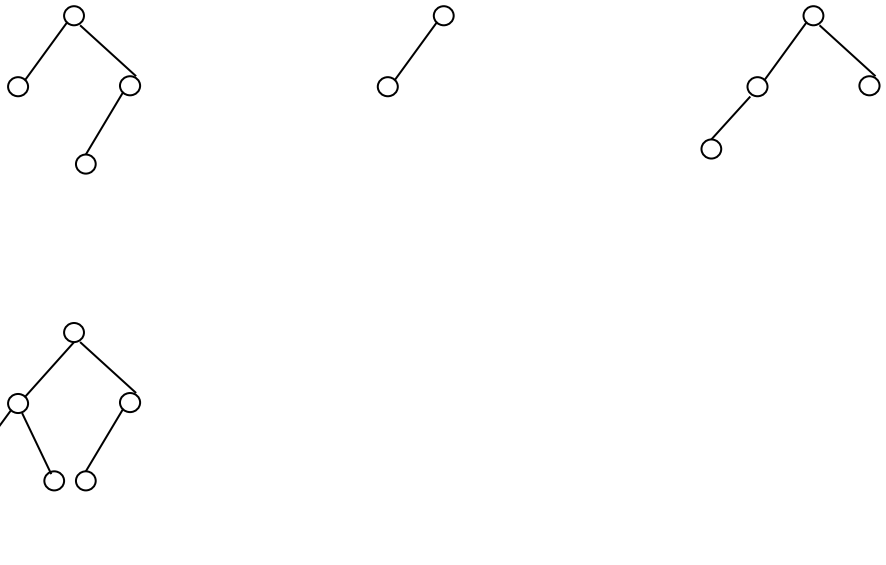
A tree is balanced when all the leaves of the tree are at the same level or differ by one level.

AVL trees –

Suppose the keys are added in sorted order to a binary tree, the tree grows in one direction only. To handle such imbalances in tree a new class of trees known as AVL trees were developed.

G.M.Adel'son, Vel'skii and E.M.Landis, first defined the AVL trees. An AVL tree is a height balanced-1 tree or HB(1) tree, were the maximum allowable height difference between any two leaves is one.

Eg: of AVL tree



The two features of AVL trees are –

→ Minimum level of searching performance.

→ Maintaining AVL trees- when new nodes are added, any of the four possible rotations occur in the tree, depending on the growth of the tree.

The four possible rotations are –

Left – Left rotation
Right – Right rotation } Single Rotation

Left – Right rotation
Right – Left rotation } Double Rotation

For a complete balanced tree, the worst-case search to find a key is $\log_2(N+1)$

For an AVL tree, the worst-case search is $1.44 \log_2(N+2)$

So AVL tree search is very near to complete binary tree search. So, AVL tree helps in faster searching, and also it is always balanced. Thus avoiding the use of indexing.

In the two problems discussed-

- Searching requires many seeks.
- Keeping an index in sorted order is expensive.

The 2nd is solved by using AVL trees. Now, the number of seeks in binary search can be avoided by using paged binary trees.

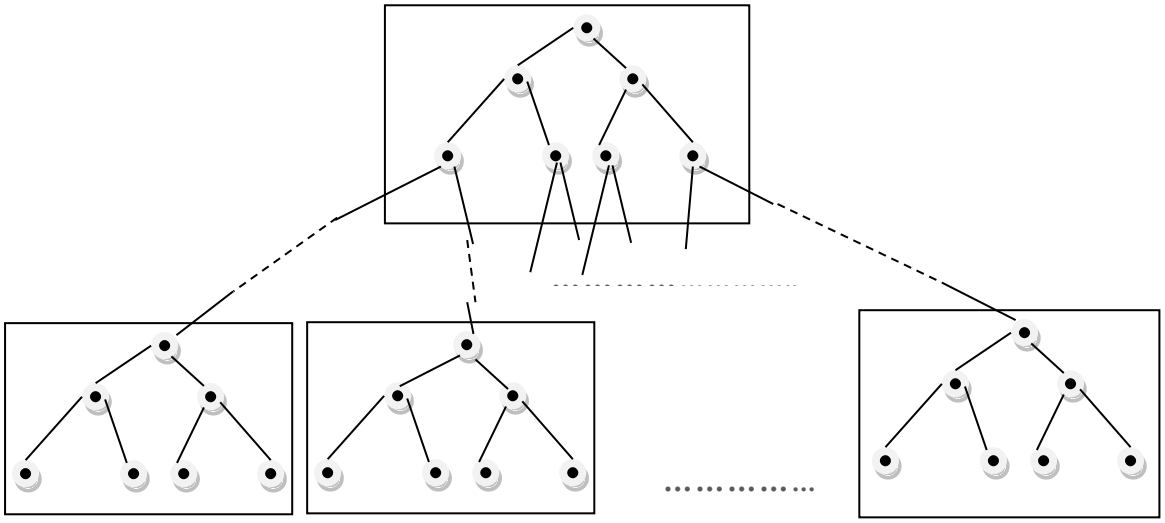
Paged Binary Trees –

The main problem with binary search tree is that the disk utilization is extremely inefficient. Each disk read has only one node of data consisting of only three useful pieces of information - the key, address of the left and right sub trees.

Each read operation, reads one page of data ie. about 512 bytes. One page consists of one node of information (only the key, left address and right address), the remaining space is wasted.

This can be solved by storing more than one node in a single page. So that in one seek that many nodes of information can be retrieved.

For eg., if seven nodes are stored in a page, by accessing this page, we get the information of seven nodes in a single access. A single access will return the information required for next move also, thus reducing the cost of disk access.



Thus in one access we get the information of seven nodes and in two access we get the information of 63 nodes.

Worst case comparison between binary tree and paged binary tree is shown below:

Binary tree	: $\log_2(N+1)$	N- No of keys
Paged binary tree	: $\log_{K+1}(N+1)$	N- No. of keys and K – no. of keys in a page

If the no of keys (N) is 13421

Binary search requires $\log_2(13421+1) = 14$ seeks

Paged binary tree requires $\log_{7+1}(13421+1) = 5$ seeks

Advantage:

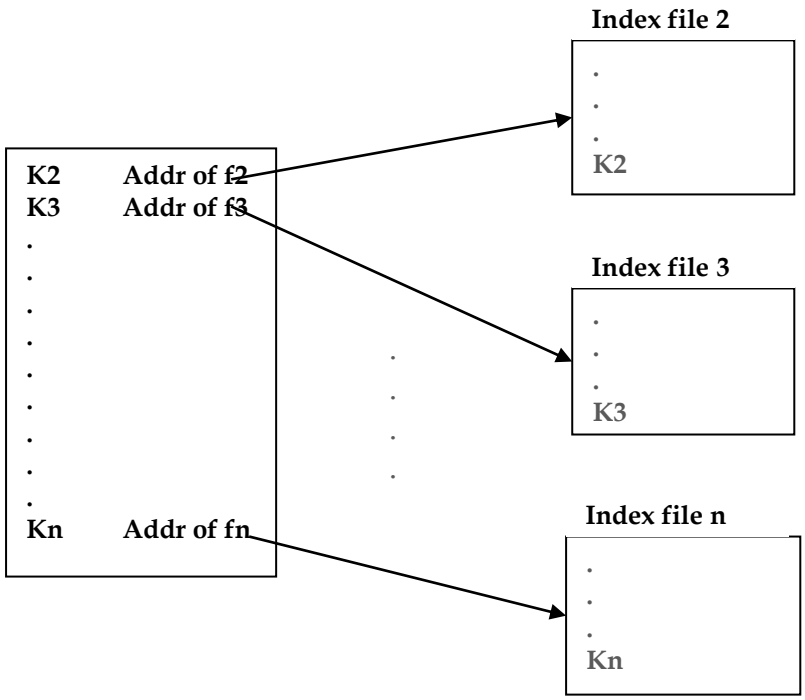
Seek time required to access many nodes is reduced compared to binary search tree.

Disadvantage:

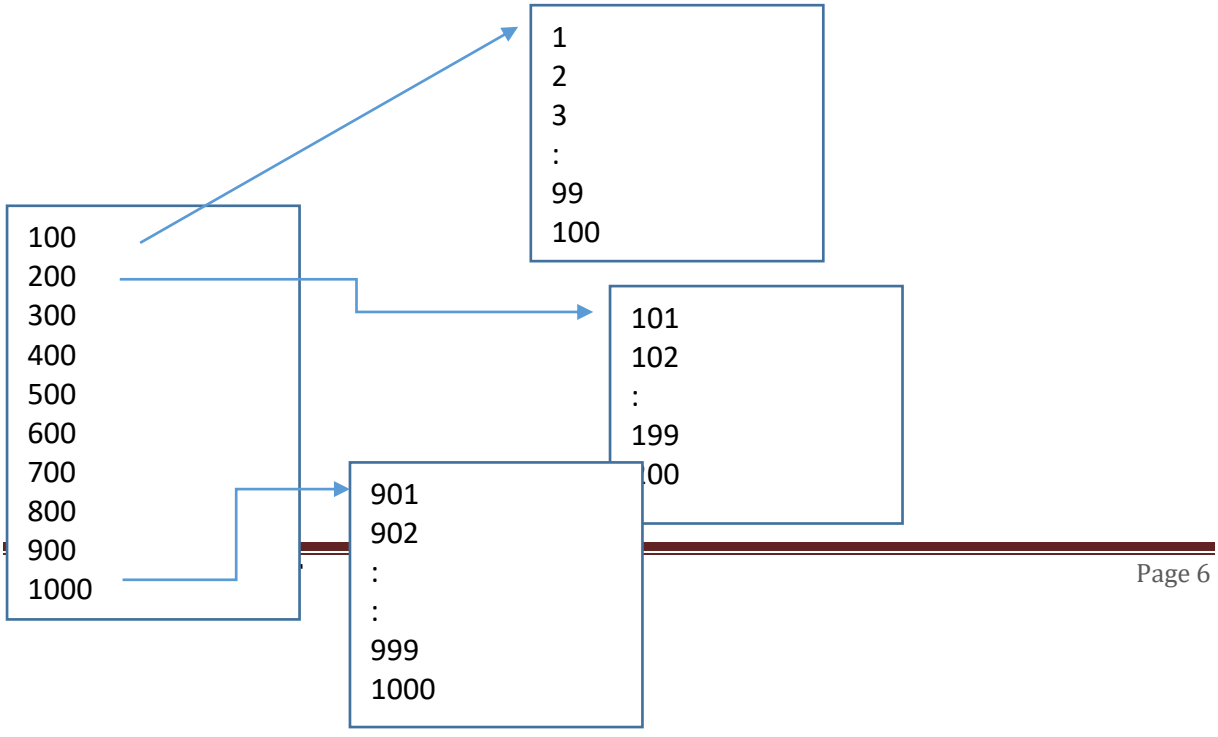
- Most of the information read is unused. All the data in page is read and thus there is wastage of transfer time.
- The paged tree formed will not be balanced.
- It is difficult to use the AVL tree techniques (rotations) in paged binary tree.
- The root page key cannot be found at the beginning and hence the tree will not be balanced.

Multilevel indexing

- Multilevel index is index of the index file.
- Since index records form a sorted list of keys, one of the keys in each index record (usually largest key) is taken as the key of the whole record.



Multilevel indexes help to reduce, the number of disk accesses and their overhead space costs are minimal. An index file to another index file is multilevel indexing.



B- Trees: Working up form bottom

B-Trees are multilevel indexes that solve the problem of linear cost of insertion and deletion. Each node of a B-tree is an index record. Each of these node has a same maximum number of key-reference pairs, called the order of B-tree.

Order of a B-tree is the maximum no. of keys in a node of a tree.
Or
Maximum no. of descendants from a node. It is usually denoted by 'm'.

While inserting a new key to a node –

- **If the node is not full, simply insert the key.**
- **If the node is full, the node is split into two nodes, each with half of the keys.**

  Since a new node is created, the largest key of this new node must be added into the next higher level node. This process is called promotion of the key.

  This promotion may cause an overflow at that higher level, which inturn causes the split of this node and again key promotion to the next higher level. This continous addition of keys, may also lead to the splitting of root node. In this way, B-tree grows up from the leaves.

Formal definition of B-tree properties –

Properties of a B-tree of order m, are –

1. Every page has a maximum of 'm' descendants.
2. Every page, except for the root and the leaves, has at least $\lceil m/2 \rceil$ descendants.
3. The root has at least two descendants.
4. All the leaves appear on the same level.
5. The leaf level forms a complete, ordered index of the associated data file.

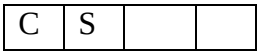
Creation of B-Tree –

If the order of the B-Tree is 'm', each node in a B-tree can store maximum of m keys.

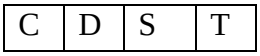
- Initially the keys are added to the node.
- When the node is full (overflow), it is split into two nodes, each containing half of the keys.
- When the new node is created, the largest key of the new nodes must be inserted into the higher level node – key promotion.
- Continue the process of insertion, splitting and key promotion until the elements to be inserted into the B-tree is complete.

Eg: Suppose the list is
C S D T A M P I B W N G U R .

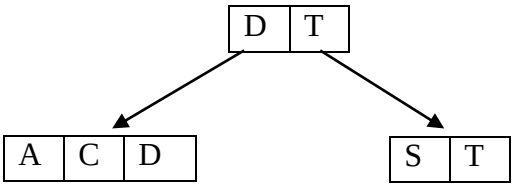
1)



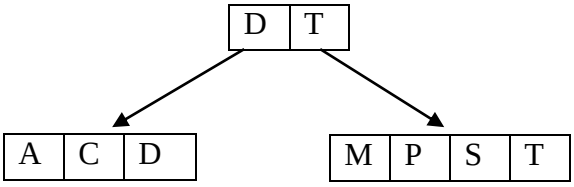
2)



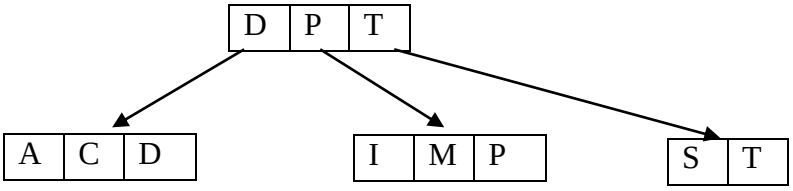
3) Insertion of 'A', causes the node to split and create a higher level node – containg the largest key from both nodes.



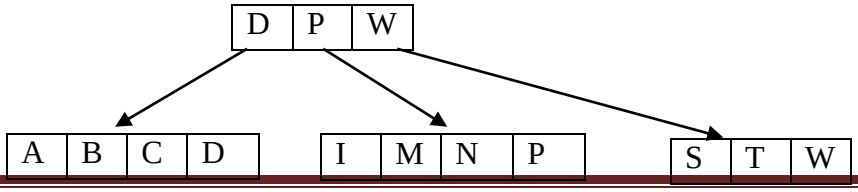
4) After addition of 'M; & 'P'.



5) The addition of 'I' in the right leaf node, causes splitting of that node. The node is split into two nodes having keys I,M,P and ST. The key P is promoted to higher level node.

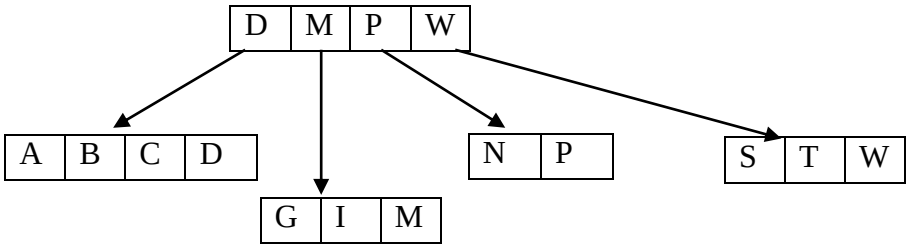


6) After insertion of B, W, N into leaf nodes -

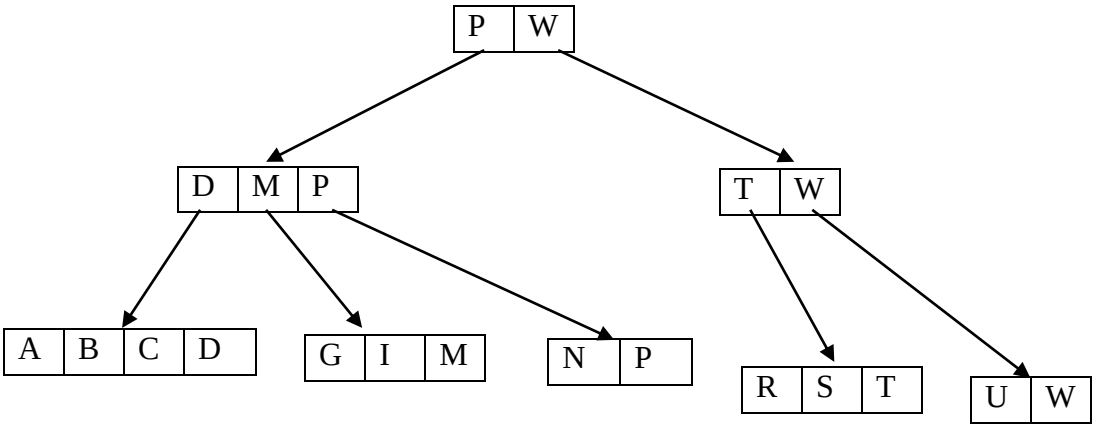


When ‘W’ is added, as ‘W’ is the highest key in that node, it is added to the higher level node.

7) Insertion of ‘G’ in the 2nd leaf node, splits the node into two (as the number of keys will be greater than 4). An key is added to the root node. Thus the root node becomes full.



8) Insertion of U & R, causes the rightmost leaf node to split. This adds up another key to the root node, and thus causes the splitting of root. Thus the tree grows to another level, a new root node is formed.



Worst case search depth –

Worst case occurs when every page of the tree has only the minimum number of descendants. For a B- tree of order ‘m’, the minimum number of descendants from the root page is 2. The minimum number of descendants from intermediate node is ⌈m/2⌉. The general expression of the relation between depth and the minimum number of descendants are –

Level	Minimum no. of descendants
1(root)	2
2	2 x ⌈m/2⌉

3	$2 \times \lceil m/2 \rceil \times \lceil m/2 \rceil$ or $2 \lceil m/2 \rceil^2$
4	$2 \times \lceil m/2 \rceil \times \lceil m/2 \rceil \times \lceil m/2 \rceil$ or $2 \lceil m/2 \rceil^3$
.	
..	
.	
d	$2 \times \lceil m/2 \rceil^{d-1}$

So, the minimum no. of descendants at level d is $2 \times \lceil m/2 \rceil^{d-1}$. If there are N keys at the leaf level, at depth 'd', then

$$N \geq 2 \times \lceil m/2 \rceil^{d-1}$$

$$N/2 \geq \lceil m/2 \rceil^{d-1}$$

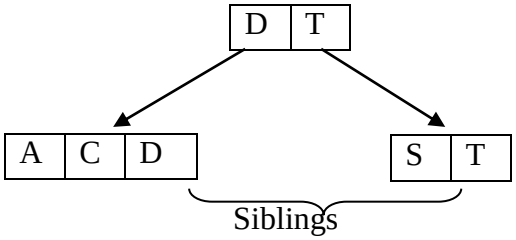
Or

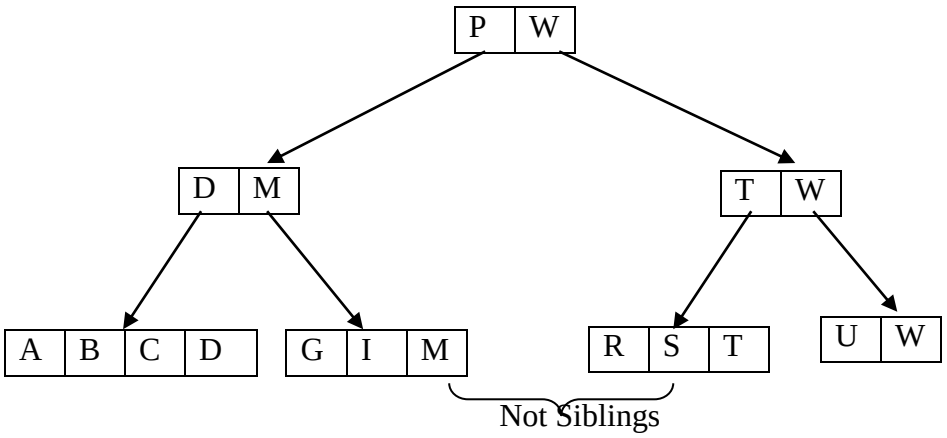
$$d \leq 1 + \log_{\lceil m/2 \rceil}(N/2).$$

Note: C++ code for B – Tree insertrion() and search() functions – Find at the end of notes

Deletion, Merging and Redistribution

The deletion of key from a node , affects the sibling node. Nodes that have the same parent node are called sibling nodes.





The rules for deleting a key '**k**' from a node '**n**' in a B-tree are as follows –

1) If node ' **n**' has more than minimum number of keys –
   a) If 'k' is not the largest in 'n' - simply delete 'k' from 'n'.
   b) If 'k' is the largest in 'n' – delete 'k' and modify the higher level indexes to reflect the new largest key in 'n'.

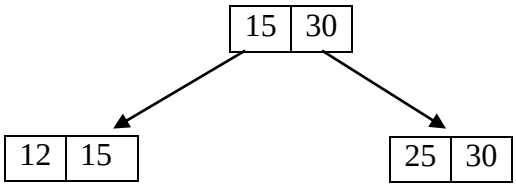
2) If '**n**' has exactly minimum number of keys and one of the sibling of 'n' has –
   a) Few enough keys (min. no. of keys only), merge 'n' with its sibling and delete a key from the parent node.
   b) Extra keys (more than min.), redistribute by moving some keys from a sibling to 'n', and modify the higher level indexes to reflect the new largest keys in the affected nodes.

Merge – After deleting a key, if the node has less than minimum number of keys (unstable), an underflow of node occurs. So the keys of this node can be put into a sibling node (iff there is enough space). This process is called merging.

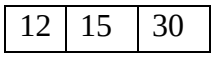
Redistribution - After deleting a key from a node, if underflow occurs in a node(the node has less than minimum number of keys) and if there is no of space for merging the keys among the siblings, ie. if the siblings are full.
The keys are distributed among the siblings, this process is called redistribution.

Redistribution differs from both splitting (creating a node) and merging (deleting a node). Here the number of nodes in a tree is not changed.

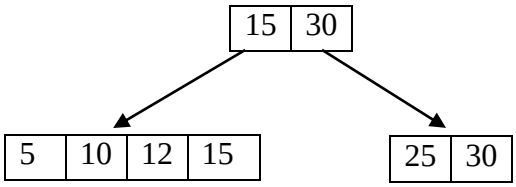
Example of deletion in B-Tree
For merging:



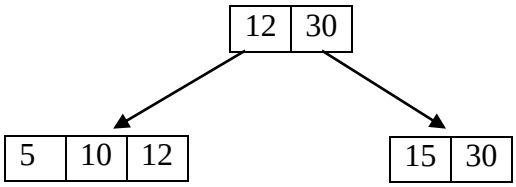
Suppose key to be deleted is '25', then the node is merged to form a single node.



For redistribution:



Suppose key to be deleted is '25'. As 1st node is full, redistribution takes place.



Redistribution during insertion : A way to improve storage utilization

During insertion, when a node is split, a new node is created. Thus using more memory space. This can be avoided by postponing the creation of new node. Rather than dividing (splitting) the node to create two pages, redistribute the keys among the existing node only.

The use of redistribution in place of splitting tends the B-tree to be more efficient in utilization of space.

B* trees

Knuth in 1973, extended the idea of redistribution in insertion and thus formed a B* tree.

By postponing the splitting by redistribution, a condition comes when the node has to be splitted. In this stage atleast one of its sibling is full. So instead of splitting one-to-two, the nodes are split as two-to-three ie. two sibling nodes that are almost full is split to form three nodes, such that each node is 2/3rd full, rather than ½ full. Thus making a B* tree .

The properties of B* tree are –

1. Every page has a maximum of 'm' descendants.
2. Every page except for the root has atleast $\lceil (2m-1)/3 \rceil$ descendants.
3. The root has atleast two descendants.

4. All the leaves appear on the same level.

Buffering of pages: Virtual B-trees

While using B-Trees, sometimes it is not possible to hold the entire tree in memory. So a page buffer can be created to hold two or more pages of B-tree.

A B-tree which uses memory buffer to hold more B-tree pages in memory is called virtual B-tree.

When a page is requested, the page is accessed from the memory (iff present), there by avoiding disk access. Buffering works only if it is likely to request a page that is in buffer. But some times the requested page may not be there in buffer. **This situation were the requested page is not in the buffer (memory) is called page fault**. Then the process is accessed from the disk.

There are two causes for page fault –

1. The page is never accessed before.
2. The page was in the buffer but has been replaced with a new page.

The first case is unavoidable, as the page is not read even ones, the page cannot be in memory. But the second case can be minimized.

One approach is to replace the page that was least recently used. This method is called **LRU replacement**. The page to be replaced is the one that has gone the longest time without

a request. LRU replacement is based on the assumption that, it is more likely to need a page that is used recently.(least recently page is not used.)

Another approach is **'Replacement Based on Page Height'** – this approach always retain the pages that occur at the highest levels of the tree. If a large amount of buffer is present, even the second level of the tree can be placed in buffer.

Important questions:

1. Write a note on AVL Trees
2. What are paged binary trees? Explain the problems associated with paged binary trees.
3. Write the adv. and disadv. of paged binary tree

4. Give the formal definition of properties of B-Tree. Why is it called as “Bottom-up” tree.
5. Mention the four properties of B* trees
6. What are the two major drawbacks with binary search to search a simple sorted index on secondary storage.
7. Show the B-Tree of order 4 that result from loading the following sets of keys in order i) CGJXNSUOAEBHIF ii)CSDAMPIBWNGURKE
8. Explain with an example the creation of B-trees.
9. Explain the following with respect to B-Tree:
10. i) Worst-case search depth ii) Redistribution during insertion.
11. What is multilevel indexing? Explain the concept of B - Trees in multilevel indexing with an example.
12. Explain deletion, Merging and redistribution of elements in B - Tree.
13. What are B-trees? Explain in detail an object oriented representation of B-trees.
14. Explain the virtual B-tree and how to overcome page fault.
15. What is redistribution? Explain redistribution during deletion and insertion of key in a B- tree node.
16. Derive an expression for worst search depth in B-tree.
17. How is storage utilization improved by redistribution of keys during insertion
18. How is B* tree different from B-Tree. List the properties of B*trees.

C++ code for B Tree Creation and searching

B Tree Creation

B Tree node structure

```
struct node
{  int level;
   int usage;
   int numbers[order+1];
   struct node* children[order+1];
   struct node* parent;
};
```

```
void insertnode()
{
        int num,i,j,k,poi;
        cout << "enter number to insert into btree\n";
        cin >> num;
        if (head == NULL)
        {  head=getnode();
```

```
   head->usage=1;
   head->numbers[0]=num;
   return;
}
else
{
        cur=head;
          while (cur != NULL)
          {
               for(i=0,j=0;i<=(cur->usage)-1;i++)
                    if (num > cur->numbers[i]) j++ ;
                prev=cur;
                if (j == cur->usage) cur=cur->children[j-1];
                else cur=cur->children[j];
          }
  cur=prev;
  poi=j;
  if (cur->numbers[poi] == num)
 {
        cout << "element already present\n";
        return;
 }
 if (poi != cur->usage)
 {
        for(i=cur->usage;i>=poi+1;i--)
        {
              cur->numbers[i]=cur->numbers[i-1];
               cur->children[i]=cur->children[i-1];
        }
}
 cur->numbers[poi]=num;
 cur->children[poi]=NULL;
 cur->usage++;

 while(cur!=NULL)
 {
    if (cur->usage <= order)
   {
        if (cur->parent == NULL) break;
        up=cur->parent; i=0;
        while (up->children[i] != cur) i++;
```

```
        up->numbers[i]=cur->numbers[(cur->usage)-1];

       cur=up;
       continue;
 }

temp=getnode();
for(k=min+1,j=0;k<=order;k++,j++)
{
       temp->numbers[j]=cur->numbers[k];
       cur->numbers[k]=999;
       temp->children[j]=cur->children[k];
       cur->children[k]=NULL;
}

cur->usage=min+1;
temp->usage=order-(cur->usage)+1;
temp->level=cur->level;
if (cur->children[0] != NULL)
{
       for (i=0;i<=(cur->usage)-1;i++) (cur->children[i])->parent=cur;
       for(i=0;i<=(temp->usage)-1;i++) (temp->children[i])->parent=temp;
}

if (cur->parent == NULL)
{  up=getnode();
   up->level=(cur->level)+1;
   up->usage=2;
   up->numbers[0]=cur->numbers[(cur->usage)-1];
   up->numbers[1]=temp->numbers[(temp->usage)-1];
   up->children[0]=cur;
   up->children[1]=temp;
   cur->parent = temp->parent = up;
   head=up;
   return;
 }
 else
 {  up=cur->parent;
    temp->parent=up;
    newup=getnode();
```

```
                i=0;
                while (up->numbers[i] < cur->numbers[0])
                {   newup->numbers[i] = up->numbers[i];
                   newup->children[i] = up->children[i];
                   i++;
                }

                newup->numbers[i]= cur->numbers[(cur->usage)-1];
                newup->children[i] = cur;
                i++;
                newup->numbers[i]= temp->numbers[(temp->usage)-1];
                newup->children[i] = temp;
                i++;

                j=0;
                while (up->numbers[j] <= temp->numbers[(temp->usage)-1]) j++;
                while (j <= (up->usage)-1)
                {   newup->numbers[i] = up->numbers[j];
                   newup->children[i] = up->children[j];
                   i++,j++;
                }

                up->usage=i;
                newup->usage=i;
                for(i=0;i<=(newup->usage)-1;i++)
                {       up->numbers[i] = newup->numbers[i];
                        up->children[i] = newup->children[i];
                }

                free(newup);
                cur=up;
                continue;
            }
        }
    }
}
```

B Tree Search

```
void searchnode()
{
        int num,i,j,poi;
        cout << "enter number to search\n";
        cin >> num;
        cur=head;
        while (cur != NULL)
        {  for(i=0,j=0;i<=(cur->usage)-1;i++)
            if (num > cur->numbers[i]) j++ ;
            prev=cur;
            if (j == cur->usage) cur=cur->children[j-1];
            else cur=cur->children[j];
        }
        cur=prev;
        poi=j;
        if (cur->numbers[poi] == num)
                cout << "search success\n";
        else
                cout << "search failure\n";
}
```