3.7 Trees

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any

number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The Fig 3.1 shows a tree and a non-tree.

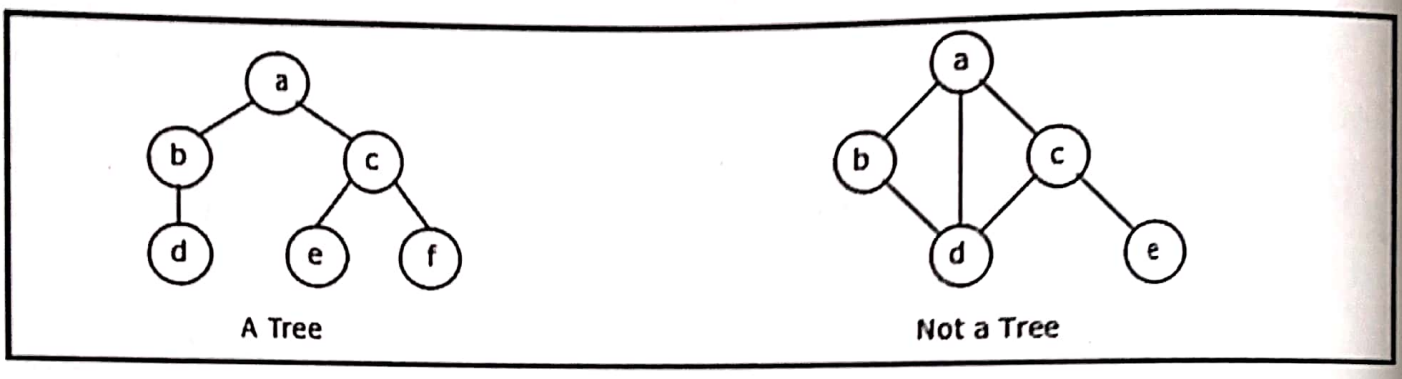


Fig 3.1 Example of Tree and Not a tree

In a tree data structure, there is no distinction between the various children of a node i.e., none is the "first child" or "last child". A tree in which such distinctions are made is called an **ordered tree**, and data structures built on them are called **ordered tree data structures**. Ordered trees are by far the commonest form of tree data structure.

3.8 Binary Tree

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

A tree with no nodes is called as a **null** tree. A binary tree is shown in Fig 3.2.

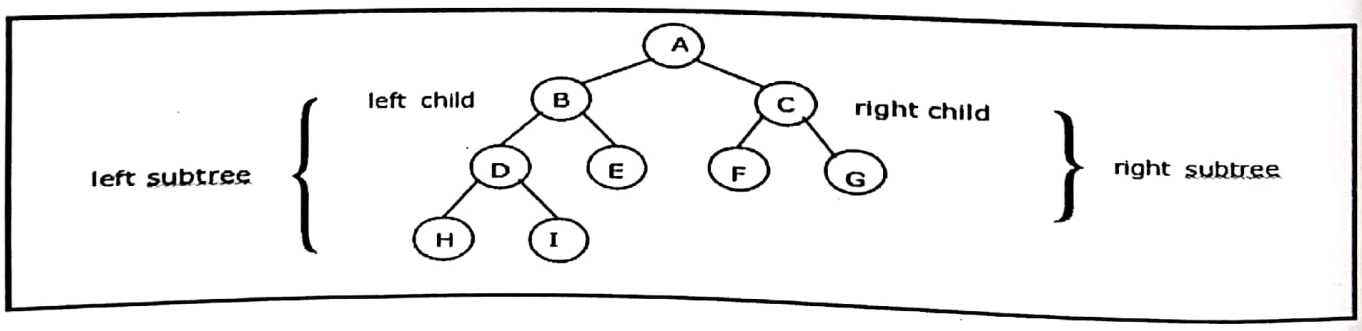


Fig 3.2 Binary Tree

Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

3.9 Tree Terminology

- **Leaf node**

A node with no children is called a *leaf* (or *external node*). A node which is not a leaf is called an *internal node.*

- **Path**

A sequence of nodes $n_1, n_2, \ldots, n_k$, such that n_i is the parent of n_{i+1} for $i = 1, 2, \ldots, k - 1$. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

For the tree shown in Fig 3.2, the path between A and I is A, B, D, I.

- **Siblings**

The children of the same parent are called siblings.

For the tree shown in Fig 3.2, F and G are the siblings of the parent node C and H and I are the siblings of the parent node D.

- **Ancestor and Descendent**

If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A.

- **Subtree**

Any node of a tree, with all of its descendants is a subtree.

- **Level**

The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent. For example, in the binary tree of Fig 3.2 node F is at level 2 and node H is at level 3. *The maximum number of nodes at any level is*

2^n.

- **Height**

The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. The height of the tree of Fig 3.2 is 3.

- **Depth**

The depth of a node is the number of nodes along the path from the root to that node. For instance, node 'C' in Fig 3.2 has a depth of 1.

- **Assigning level numbers and Numbering of nodes for a binary tree:**

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent. For example, see Fig 3.3.

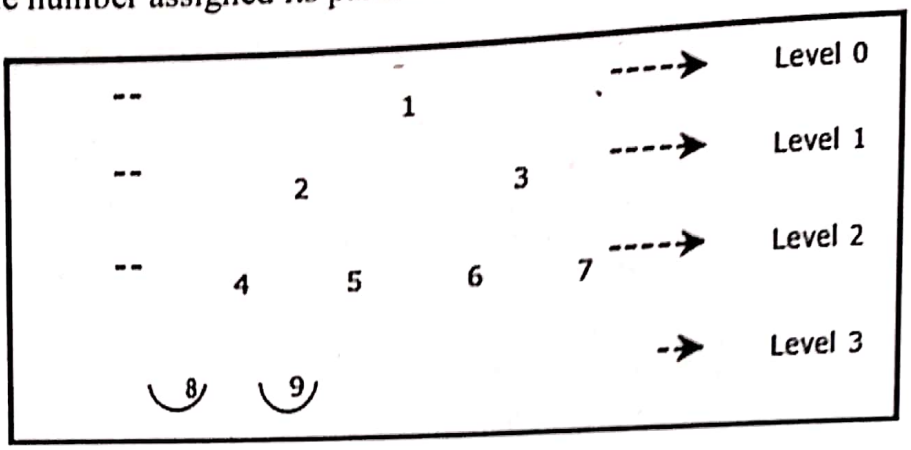


Fig 3.3. Level by level numbering of binary tree

- **Properties of binary trees:**

Some of the important properties of a binary tree are as follows:

If h = height of a binary tree, then

Maximum number of leaves = 2^h

Maximum number of nodes = $2^{h+1} - 1$

If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.

Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.

The total number of edges in a full binary tree with n node is n - 1.

- **Strictly Binary tree:**

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed as strictly binary tree. Thus the tree of Fig 3.4(a) is strictly binary. A strictly binary tree with n leaves always contains 2n - 1 nodes.

- **Full Binary tree:**

A full binary tree of height h has all its leaves at level h. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height h has $2^{h+1} - 1$ nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h. Fig 3.4(d) illustrates the full binary tree containing 15 nodes and of height 3.

A full binary tree of height h contains 2^h leaves and, $2^h - 1$ non-leaf nodes.

Thus by induction, total number of nodes (tn) = $\sum_{l=0}^{h} 2^l = 2^{h+1}$

-1.

$l = 0$

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

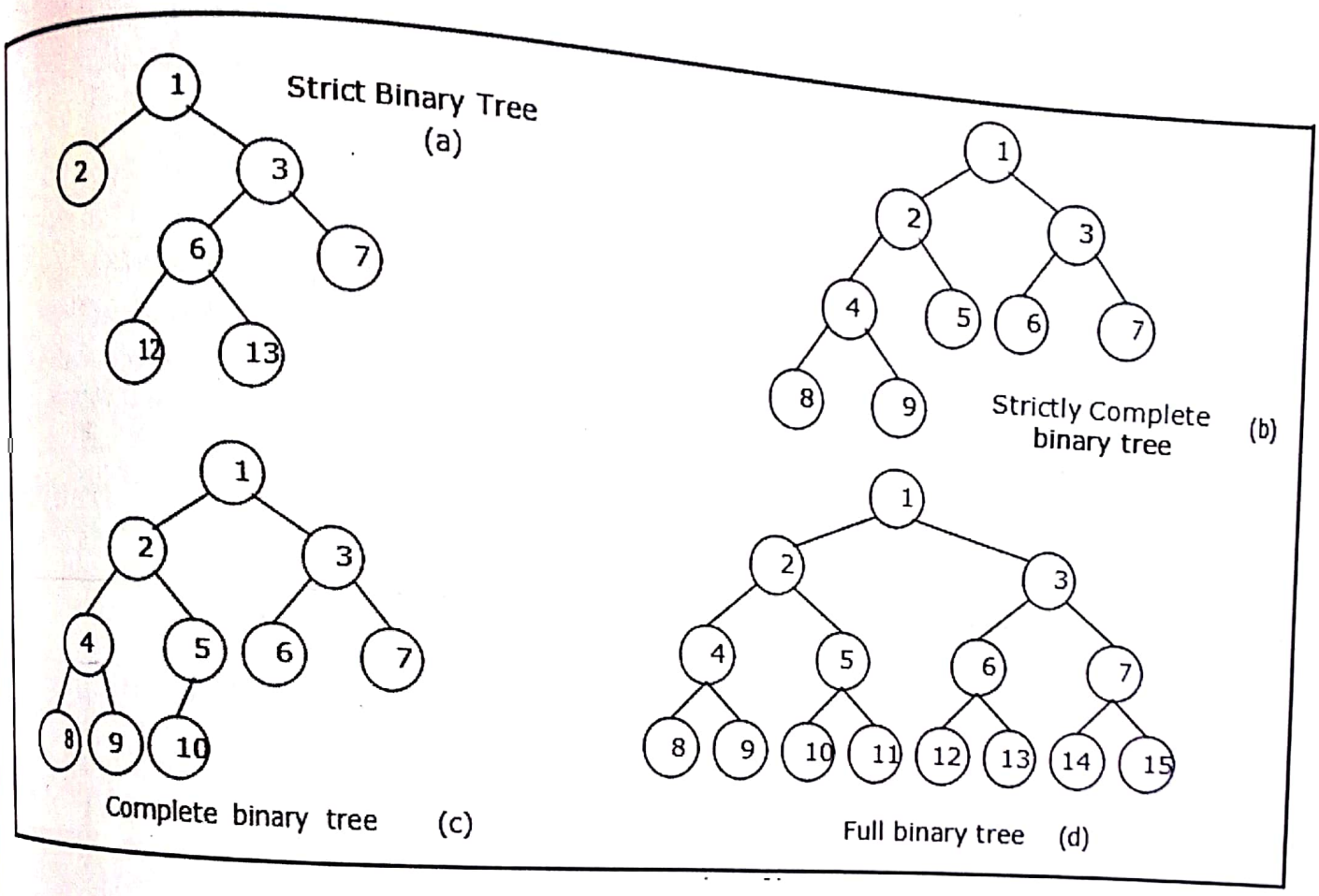


Fig 3.4 Examples of binary trees

- **Complete Binary tree:**

A binary tree with n nodes is said to be **complete** if it contains all the first n nodes of the above numbering scheme. Fig 3.5 shows examples of complete and incomplete binary trees.

A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right.

A complete binary tree with n leaves that is *not strictly* binary has 2n nodes. For example, the tree of Fig 3.4(c) is a complete binary tree having 5 leaves and 10 nodes.

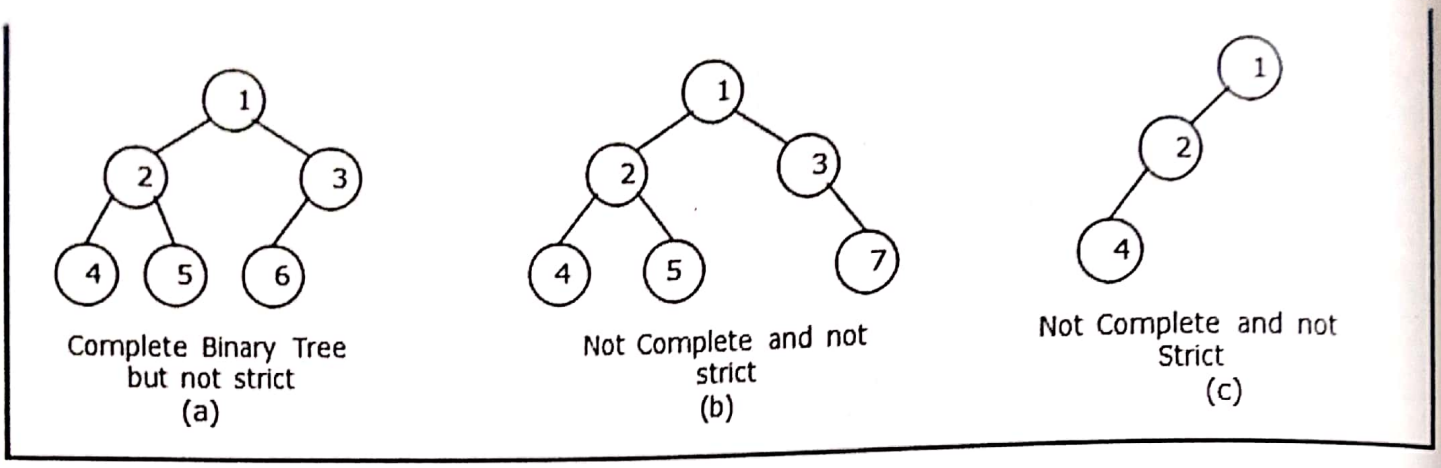


Fig 3.5. Examples of complete and incomplete binary trees

- **Internal and external nodes:**

We define two terms: Internal nodes and external nodes. An internal node is a tree node having at least one–key and possibly some children. It is sometimes convenient to have another types of nodes, called an external node, and pretend that all null child links point to such a node. An external node doesn't exist, but serves as a conceptual place holder for nodes to be inserted.

We draw internal nodes using circles, with letters as labels. External nodes are denoted by squares. The square node version is sometimes called an extended binary tree. A binary tree with n internal nodes has n+1 external nodes. Fig 3.6 shows a sample tree illustrating both internal and external nodes.

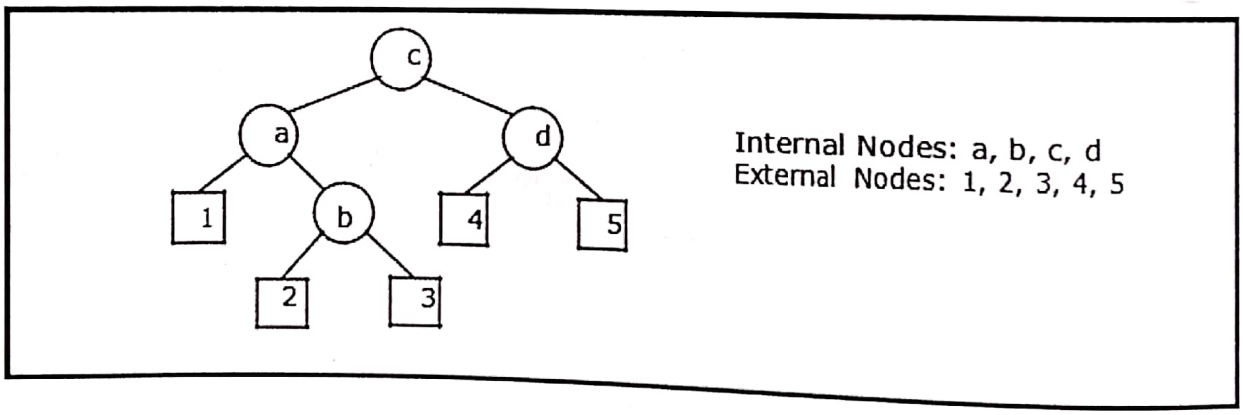


Fig 3.6. Internal and external nodes

3.10 Data Structures for Binary Tree

- Arrays; especially suited for complete and full binary trees.
- Pointer-based.

3.10.1 Array-based Implementation

Binary trees can also be stored in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index i, its children are found at indices $2i+1$ and $2i+2$, while its parent (if any) is found at index *floor((i-1)/2)* (assuming the root of the tree stored in the array at an index zero).

3.11 Binary Tree Traversal Techniques

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are three common ways to traverse a binary tree:

Preorder
Inorder
Postorder

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

3.11.1 Recursive Traversal Algorithms:

- **Inorder Traversal**

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

Visit the left subtree, using inorder.
Visit the root.
Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)

{
    if(root != NULL)
    {
        inorder(root->lchild);
        print root -> data;

        inorder(root->rchild);
    }
}
```

- **Preorder Traversal**

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

Visit the root.
Visit the left subtree, using preorder.
Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
void preorder(node *root)

{
    if( root != NULL )
    {
        print root -> data;
        preorder (root -> lchild);
        preorder (root -> rchild);
    }
```

- **Postorder Traversal**

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

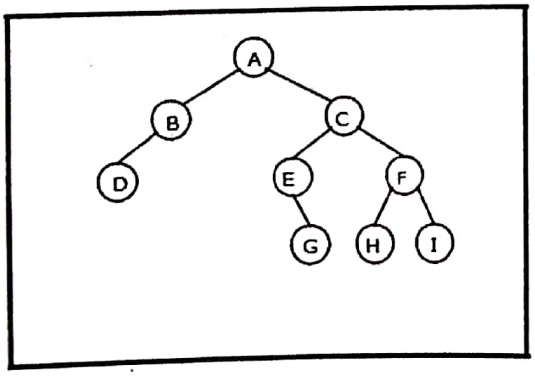
Visit the left subtree, using postorder.
Visit the right subtree, using postorder
Visit the root.

The algorithm for postorder traversal is as follows:

```
void postorder(node *root)
{
    if( root != NULL )
    {
        postorder (root -> lchild);
        postorder (root -> rchild);
        print (root -> data);
    }
}
```

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



Preo rder traversal yields: A, B, D, C , E, G , F , H, I

Posto rder traversal yields: D, B, G , E, H, I, F , C , A

Ino rder traversal yields: D, B, A, E, G , C , H, F , I

Level o rder traversal yields: A, B, C , D, E, F , G , H, I

3.15 Binary Search Tree

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

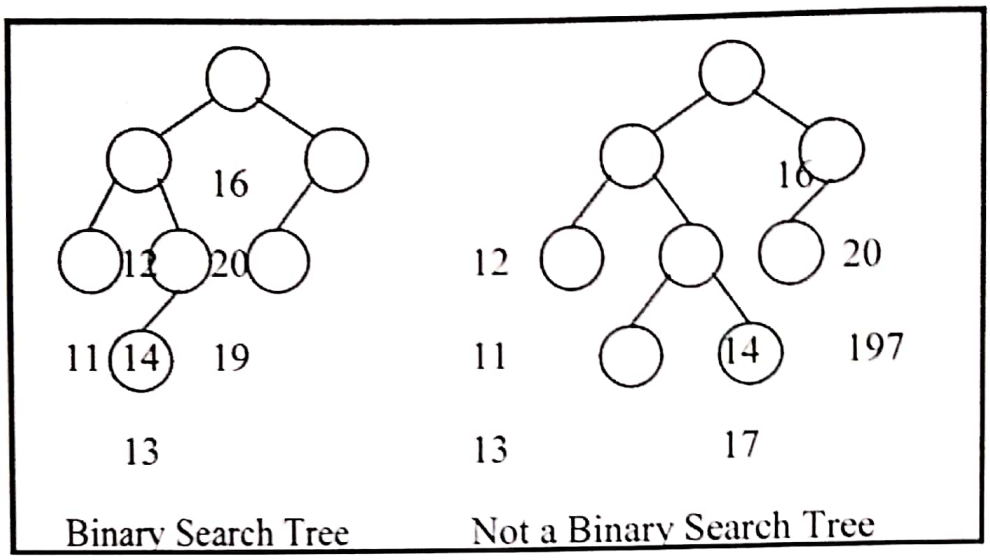
Every element has a key and no two elements have the same key.

The keys in the left subtree are smaller than the key in the root.

The keys in the right subtree are larger than the key in the root.

The left and right subtrees are also binary search trees.

Fig 3.7(a) is a binary search tree, whereas Fig 3.7(b) is not a binary search tree.



(a) (b)

Fig 3.7. Examples of binary search

3.16 General Trees (m-ary tree)

If in a tree, the outdegree of every node is less than or equal to ***m***, the tree is called general tree. The general tree is also called as an *m-ary* tree. If the outdegree of every node is exactly equal to m or zero then the tree is called *a full or complete m-ary* tree. For m = 2, the trees are called *binary* and *full binary trees*.

Differences between trees and binary trees:

TREE	BINARY TREE
Each element in a tree can have any number of subtrees.	Each element in a binary tree has at most two subtrees
The subtrees in a tree are unordered.	The subtrees of each element in a binary tree are ordered.

3.17 AVL Trees

An **AVL tree** (Adelson-Velskii and Landis' tree, named after the inventors) is a self-balancing binary search tree. It was the first such data structure to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all

take $O(\log n)$ time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

The balance factor is calculated as follows: balanceFactor = height(left subtree) - height(right subtree). For each node checked, if the balance factor remains −1, 0, or +1 then no rotations are necessary. However, if balance factor becomes less than -1 or greater than +1, the subtree rooted at this node is unbalanced. In other words,

An AVL tree is a binary arch tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.

2. Every sub-tree is an AVL tree.

Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root)<keys(right)). 1) Left Rotation 2) Right Rotation

```
T1, T2 and T3 are subtrees of the tree rooted with y (on left
side)
or x (on right side)
                                                  x
          y                                      / \
         / \       Right Rotation              T1   y
        x   T3     - - - - - - - >                 / \
       / \         < - - - - - - -                T2  T3
     T1   T2       Left Rotation
Keys in both of the above trees follow the following order
     keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

Steps to follow for insertion Let the newly inserted node be w

1)Perform standard BST insert for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There

can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways.

Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

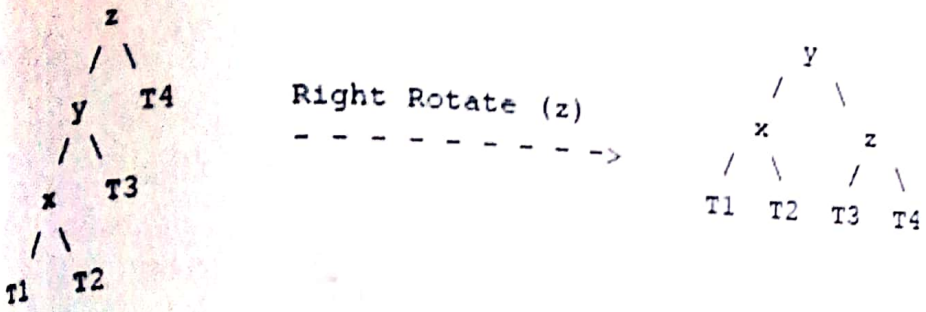
c) y is right child of z and x is right child of y (Right Right Case)

d) y is right child of z and x is left child of y (Right Left Case)

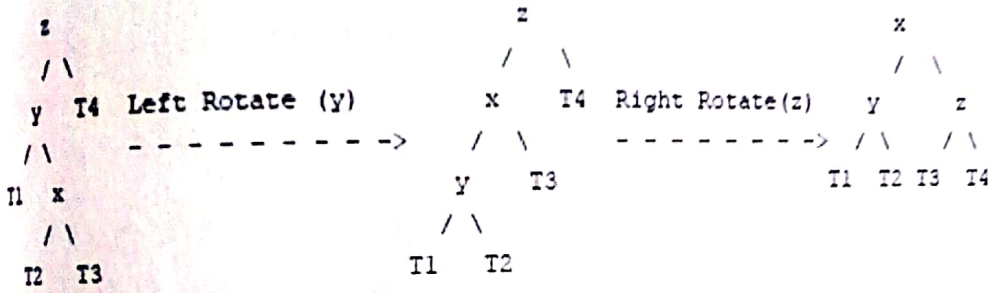
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before

a) Left Left Case

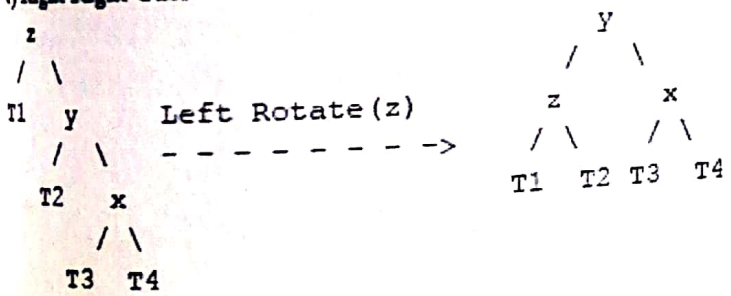
T1, T2, T3 and T4 are subtrees.



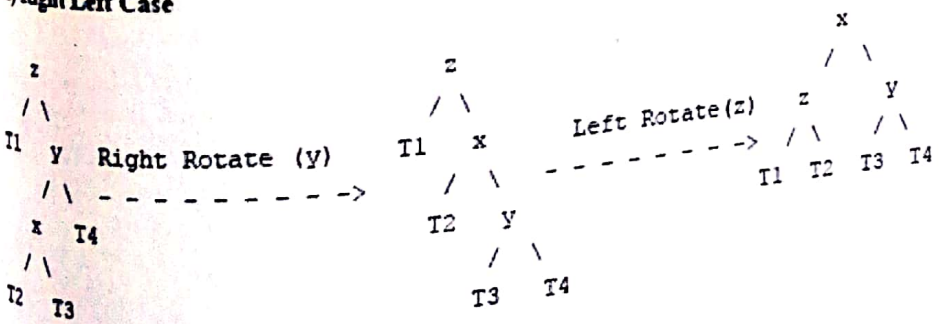
b) Left Right Case



c) Right Right Case



d) Right Left Case



Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

To make sure that the given tree remains AVL after every deletion, we must augment standard BST delete operation to perform some re-balancing as described above in insetion. w be the node to be deleted

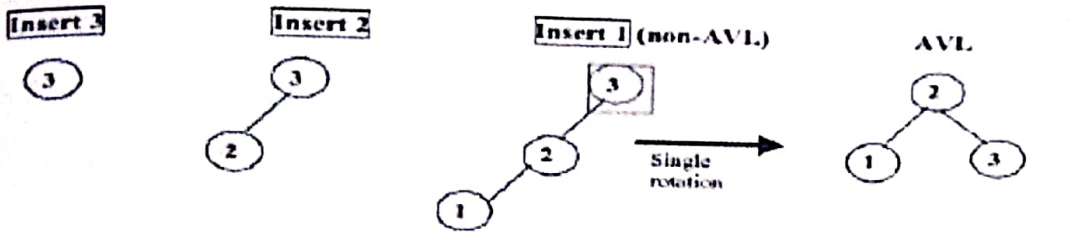
1) Perform standard BST delete for w.

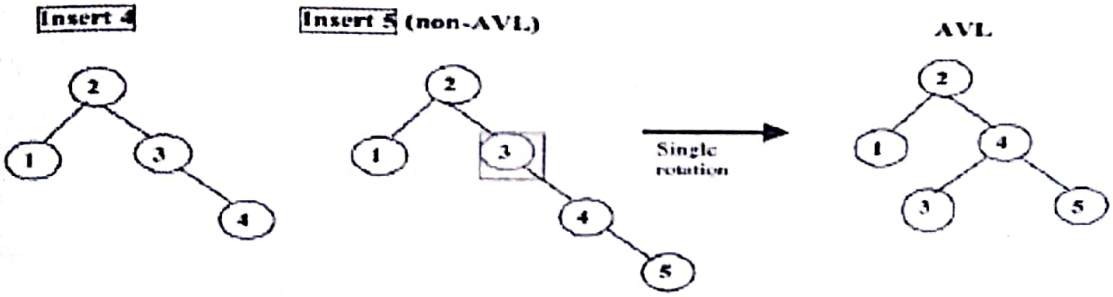
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.

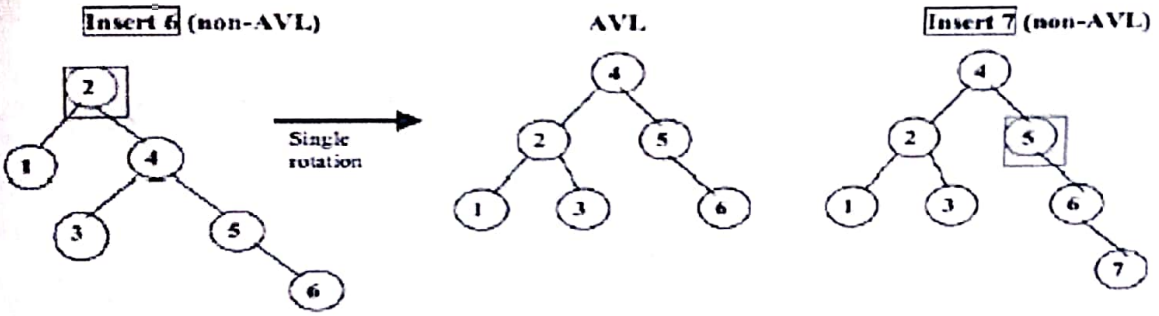
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z as explained above.

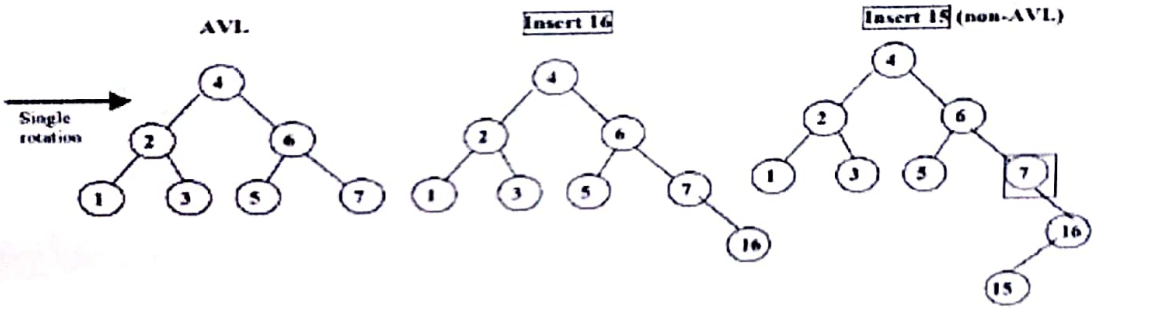
Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, may have to fix ancestors of z as well. Time Complexity: The rotation operations (left and rig rotate) take constant time as only few pointers are being changed there. Updating the height ar getting the balance factor also take constant time. So the time complexity of AVL delete remai same as BST delete which is O(h)

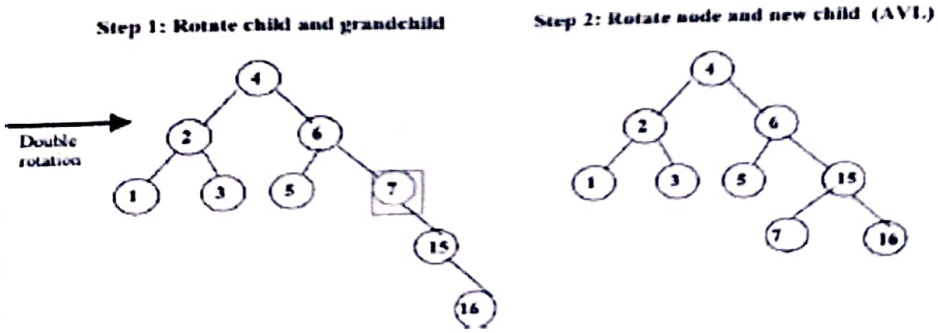
where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So tim complexity of AVL delete is O(Logn).For eg:

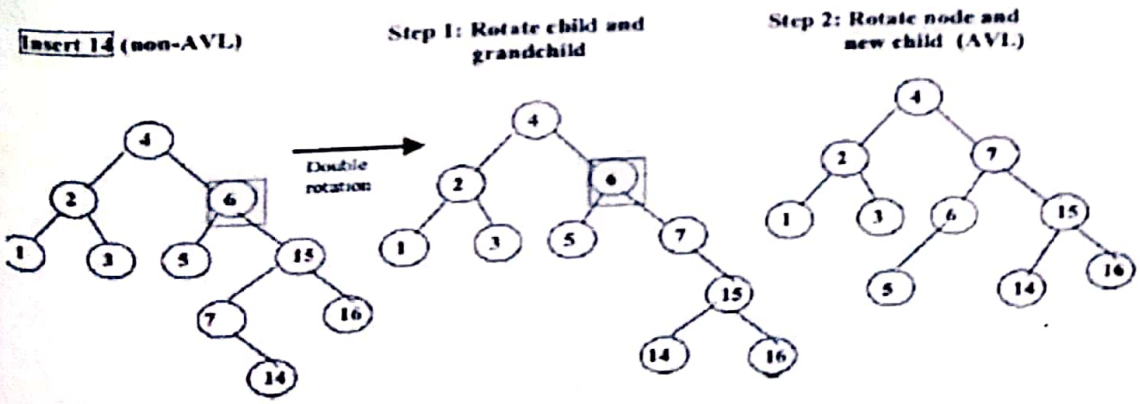










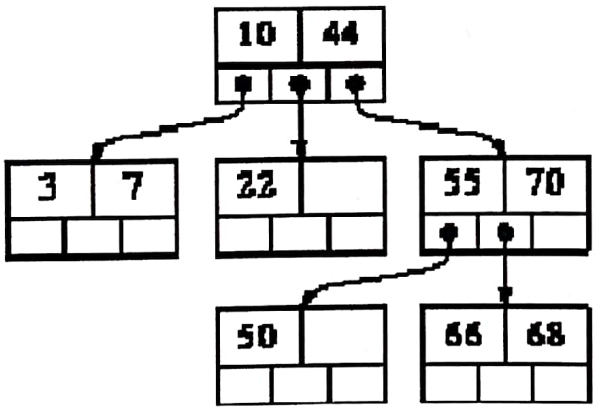


3.18 M-WAY Search Trees

A binary search tree has *one* value in each node and *two* subtrees. This notion easily generalizes to an M-way search tree, which has (M-1) values per node and M subtrees. M is called the *degree* of the tree. A binary search tree, therefore, has degree 2. In fact, it is not necessary for every node to contain exactly (M-1) values and have exactly M subtrees. In an M-way subtree a node can have anywhere from 1 to (M-1) values, and the number of (non- empty) subtrees can range from 0 (for a leaf) to 1+(the number of values). M is thus a *fixed upper limit* on how much data can be stored in a node.

The values in a node are stored in ascending order, V1 < V2 < ... Vk (k <= M-1) and the subtrees are placed between adjacent values, with one additional subtree at each end. We can thus associate with each value a `left' and `right' subtree, with the right subtree of Vi being the same as the left subtree of V(i+1). All the values in V1's left subtree are less than V1 , all the values in Vk's subtree are greater than Vk; and all the values in the subtree between V(i) and (i+1) are greater than V(i) and less than V(i+1).

For example, here is a 3-way search tree:



In the examples it will be convenient to illustrate M-way trees using a small value of M. But in practice, M is usually very large. Each node corresponds to a physical block on disk, and M represents the maximum number of data items that can be stored in a single block. The algorithm for searching for a value in an M-way search tree is the obvious generalization of the algorithm for searching in a binary search tree. If we are searching for value X and currently at node consisting of values V1...Vk, there are four possible cases that can arise:

1. If X < V1, recursively search for X in V1's left subtree.

2. If X > Vk, recursively search for X in Vk's right subtree.

3. If X=Vi, for some i, then we are done (X has been found).

4. the only remaining possibility is that, for some i, Vi < X < V(i+1). In this case recursively search for X in the subtree that is in between Vi and V(i+1).

For example, suppose we were searching for 68 in the tree above. At the root, case (2) would apply, so we would continue the search in V2's right subtree. At the root of this subtree, case (4) applies, 68 is between V1=55 and V2=70, so we would continue to search in the subtree between them. Now case (3) applies, 68=V2, so we are done. If we had been searching for 69, exactly the same processing would have occurred down to the last node. At that point, case (2) would apply, but the subtree we want to search in is empty. Therefore we conclude that 69 is not in the tree.

3.19 B-trees: Perfectly Height-balanced M-way search trees

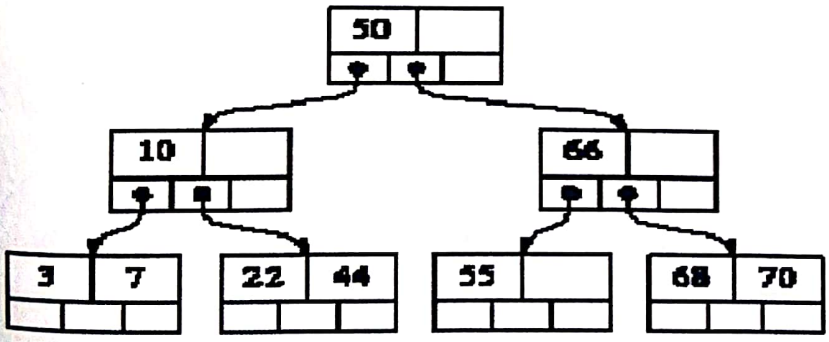
A B-tree is an M-way search tree with two special properties:

1. It is perfectly balanced: every leaf node is at the same depth.

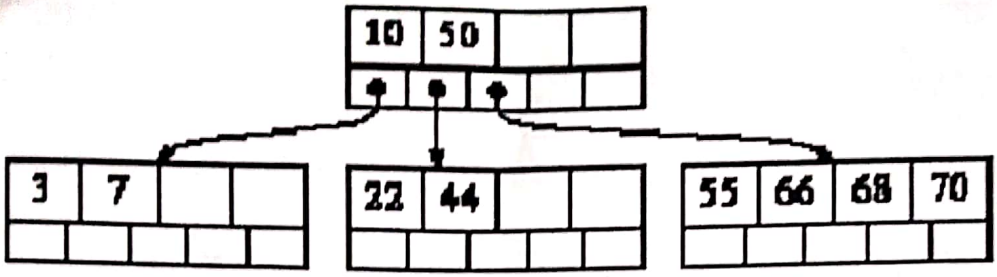
2. Every node, except perhaps the root, is at least half-full, i.e. contains M/2 or more values

(of course, it cannot contain more than M-1 values). The root may have any number of values (1 to M-1).

The 3-way search tree above is clearly *not* a B-tree. Here is a 3-way B-tree containing the same values:



And here is a 5-way B-tree (each node other than the root must contain between 2 and 4 values):



Insertion into a B-Tree

To insert value X into a B-tree, there are 3 steps:

1. using the SEARCH procedure for M-way trees (described above) find the leaf node to which X should be added.

2. add X to this node in the appropriate place among the values already there. Being a leaf node there are no subtrees to worry about.

3. if there are M-1 or fewer values in the node after adding X, then we are finished. If there are M nodes after adding X, we say the node has *overflowed*. To repair this, we split the node into three parts:

Left:

the first (M-1)/2 values

Middle:

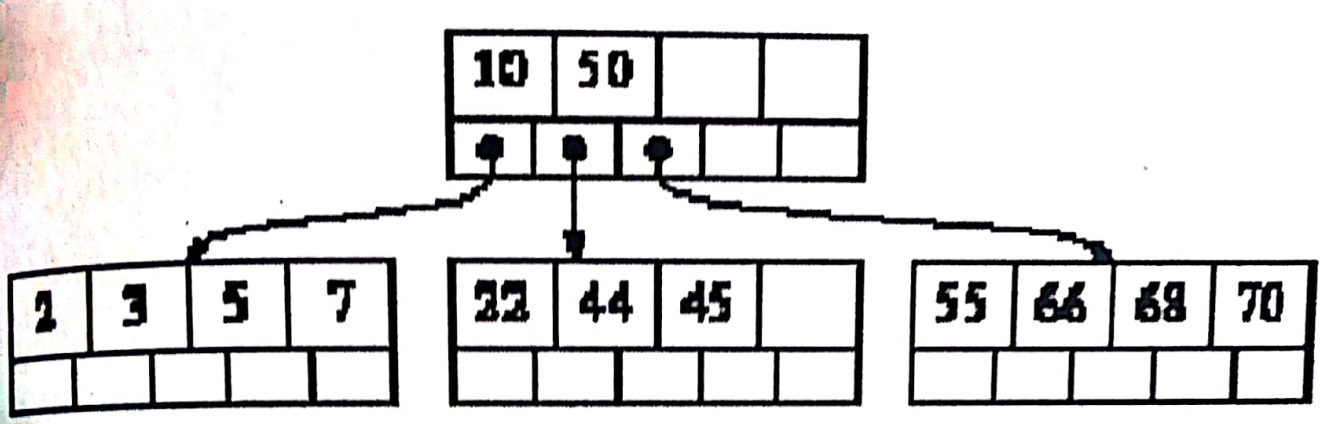
the middle value (position 1+((M-1)/2)

Right:

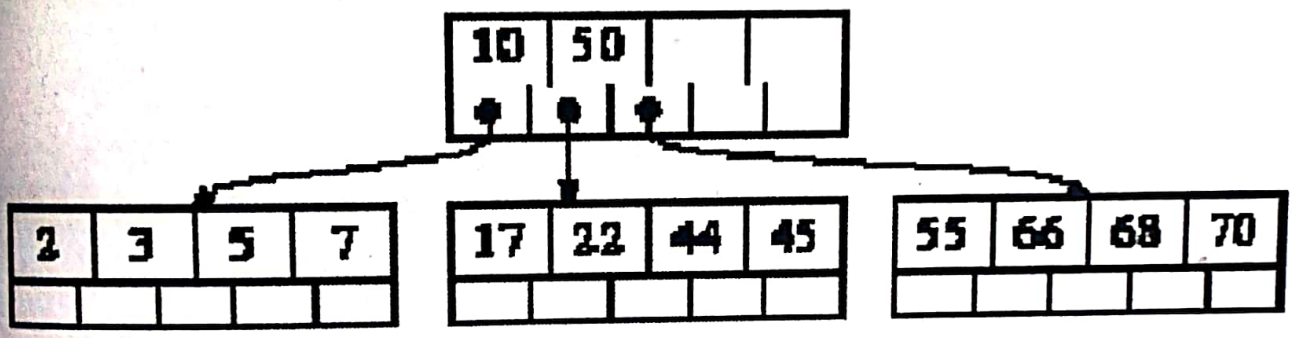
the last (M-1)/2 values

For example, let's do a sequence of insertions into this B-tree (M=5, so each node other than the root must contain between 2 and 4 values):

For example, let's do a sequence of insertions into this B-tree (M=5, so each node other than the root must contain between 2 and 4 values):



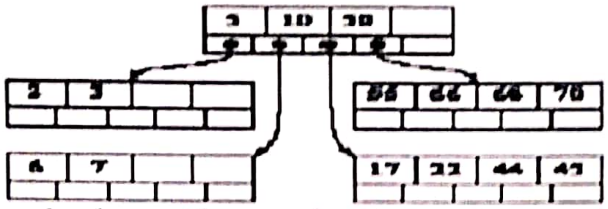
Insert 17: Add it to the middle leaf. No overflow, so we're done.



Insert 6: Add it to the leftmost leaf. That overflows, so we split it:

- Left = [2 3]
- Middle = 5
- Right = [6 7]

Left and Right become nodes; Middle is added to the node above with Left and Right as its children.

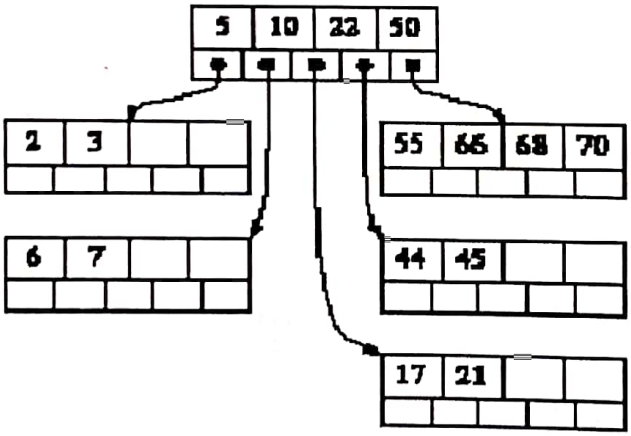


The node above (the root in this small example) does not overflow, so we are done.

Insert 21: Add it to the middle leaf. That overflows, so we split it:

- left = [17 21]
- Middle = 22
- Right = [44 45]

Left and Right become nodes; Middle is added to the node above with Left and Right as its children.

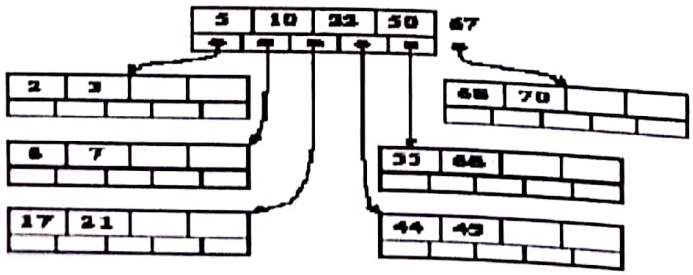


The node above (the root in this small example) does not overflow, so we are done.

Insert 67: Add it to the rightmost leaf. That overflows, so we split it:

- Left = [55 66]
- Middle = 67
- Right = [68 70]

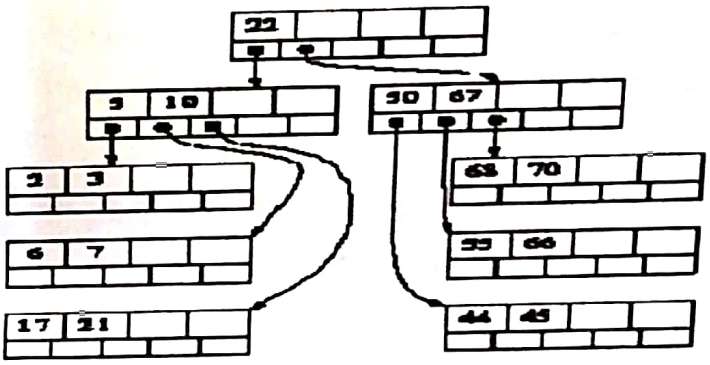
Left and Right become nodes; Middle is added to the node above with Left and Right as its children.



But now the node above does overflow. So it is split in exactly the same manner:

- Left = [5 10] (along with their children)
- Middle = 22
- Right = [50 67] (along with their children)

Left and Right become nodes, the children of Middle. If this were not the root, Middle would be added to the node above and the process repeated. If there is no node above, as in this example, a new root is created with Middle as its only value.



Deleting a Value from a B-Tree

In B-tree if the value to be deleted does not occur in a leaf, we replace it with the largest value in its left subtree and then proceed to delete that value from the node that originally contained it. For example, if we wished to delete 67 from the above tree, we would find the largest value in 67's left subtree, 66, replace 67 with 66, and then delete the occurrence of 66 in the left subtree.

In a B-tree, the largest value in any value's left subtree is guaranteed to be in leaf. Therefore

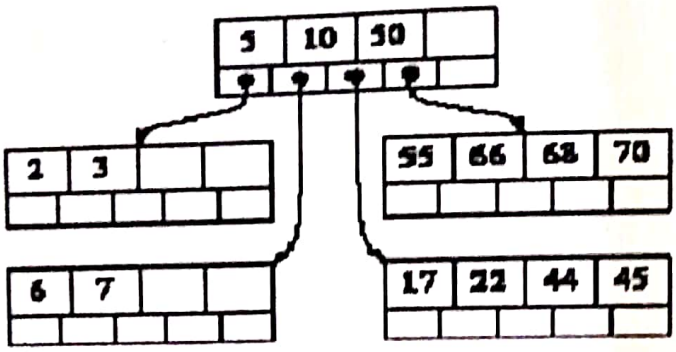
wherever the value to be deleted initially resides, the following deletion algorithm always begins at a leaf.

To delete value X from a B-tree, starting at a leaf node, there are 2 steps:

1. Remove X from the current node. Being a leaf node there are no subtrees to worry about.

2. Removing X might cause the node containing it to have *too few* values.

Remember that we require the root to have at least 1 value in it and all other nodes to have atleast (M-1)/2 values in them. If the node has too few values, we say it has *underflowed*. e.g. deleting 6 from this B-tree (of degree 5):



Removing 6 causes the node it is in to underflow, as it now contains just 1 value (7). Our strategy for fixing this is to try to 'borrow' values from a neighbouring node. We join together the current node and its more populous neighbour to form a 'combined node' - and we must also include in the combined node the value in the parent node that is in between these two nodes.

In this example, we join node [7] with its more populous neighbour [17 22 44 45] and put '10' in between them, to create

[7 10 17 22 44 45]

How many values might there be in this combined node?

· The parent node contributes 1 value.

· The node that underflowed contributes exactly (M-1)/2-1 values.

· The neighbouring node contributes somewhere between (M-1)/2 and (M-1) values.

The treatment of the combined node is different depending on whether the neighbouring contributes exactly (M-1)/2 values or more than this number.

Case 1: Suppose that the neighbouring node contains more than (M-1)/2 values. In this case, the total number of values in the combined node is strictly greater than 1 + ((M-1)/2 - 1) + ((M-1)/2), i.e. it is strictly greater than (M-1). So it must contain M values or more.

We split the combined node into three pieces: Left, Middle, and Right, where Middle is a single value in the very middle of the combined node. Because the combined node has M values or more, Left and Right are guaranteed to have (M-1)/2 values each, and therefore are legitimate nodes. We replace the value we borrowed from the parent with Middle and we use Left and Right as its two children. In this case the parent's size does not change, so we are completely finished.

This is what happens in our example of deleting 6 from the tree above. The combined node [7 10 17 22 44 45] contains more than 5 values, so we split it into:

· Left = [7 10]

Middle = 17

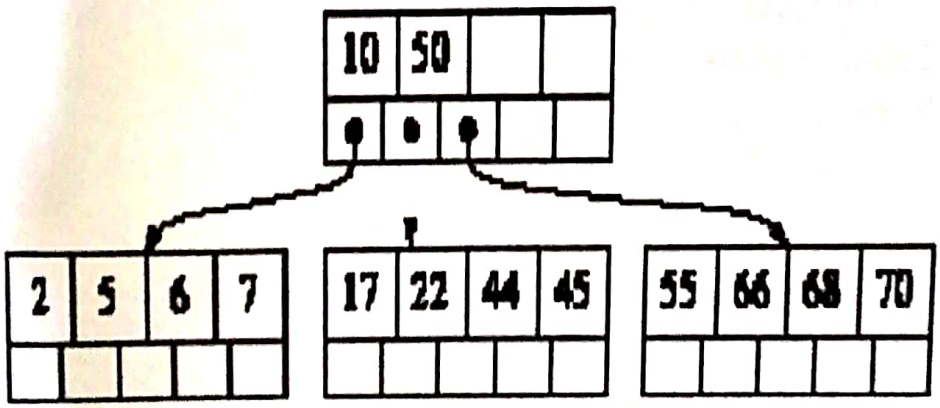
Right = [22 44 45]

Then put Middle into the parent node (in the position where the '10' had been) with Left and Right as its children

Case 2: Suppose, on the other hand, that the neighbouring node contains exactly (M-1)/2 values. Then the total number of values in the combined node is 1 + ((M-1)/2 - 1) + ((M-1)/2) =(M-1)

In this case the combined node contains the right number of values to be treated as a node. So we make it into a node and remove from the parent node the value that has been incorporated into the new, combined node. As a concrete example of this case, suppose that, in the above tree, we had deleted 3 instead of 6. The node [2 3] underflows when 3 is removed. It would be combined with its more populous neighbour [6 7] and the intervening value from the parent (5) to create the combined node [2 5 6 7]. This contains 4 values, so it can be used without further processing.

The result would be:

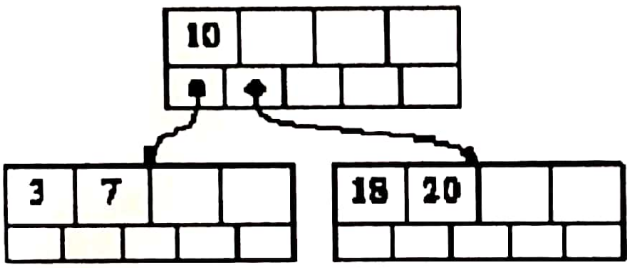


It is very important to note that the parent node now has one fewer value. This might cause *it* to underflow - imagine that 5 had been the *only* value in the parent node. If the parent node underflows, it would be treated in exactly the same way - combined with *its* more populous neighbor etc. The underflow processing repeats at successive levels until no underflow occurs or until the root underflows.

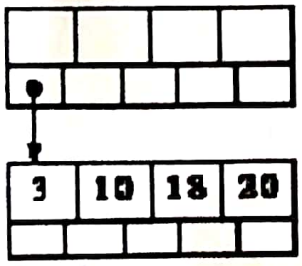
Now let us consider the root. For the root to underflow, it must have originally contained just one value, which now has been removed. If the root was also a leaf, then there is no problem: in this case the tree has become completely empty.

If the root is not a leaf, it must originally have had two subtrees (because it originally contained one value).The deletion process always starts at a leaf and therefore the only way the root could have its value removed is through the Case 2. The root's two children have been combined, along with the root's only value to form a single node. But if the root's two children are now a single node, then *that node* can be used as the new root, and the current root (which has underflowed) can simply be deleted.

suppose we delete 7 from this B-tree (M=5):



The node [3 7] would underflow, and the combined node [3 10 18 20] would be created. This has 4 values, which is acceptable when M=5. So it would be kept as a node, and `10' would be removed from the parent node - the root. This is the only circumstance in which underflow can occur in a root that is not a leaf. The situation is this:



Clearly, the current root node, now empty, can be deleted and its child used as the new root.

3.20 B+ Tree

A B+ tree is an n-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children. A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, file systems. This is primarily because unlike binary search trees, B+ trees have very high fanout (number of pointers to child nodes in a node, typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

4.1 Introduction to Sorting

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

Roll No.

Name

Age

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Sorting Efficiency

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters.

First parameter is the execution time of program, which means time taken for execution of program.

Second is the space, which means space taken by the program.

4.2 Types of Sorting

An internal sort is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory.

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub files are combined into a single larger file.

We can say a sorting algorithm is stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

4.2.4 Quick Sort

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of Divide and Conquer (also called partition-exchange sort). This algorithm divides the list into three main parts

Elements less than the Pivot element

Pivot element

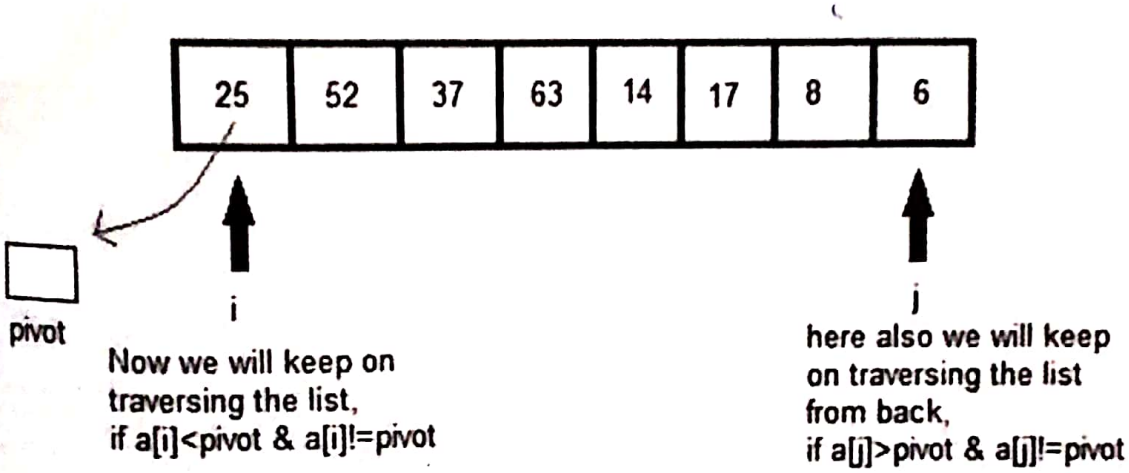
Elements greater than the pivot element

How Quick Sort Works

In the list of elements, mentioned in below example, we have taken 25 as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.



if both sides we find the element not satisfying their respective conditions, we swap them. And keep repeating this.

DIVIDE AND CONQUER - QUICK SORT

Quick Sort Algorithm

```
void quicksort (int a[], int low, int high);
int partition(int a[], int low, int high);
int a[5] = {55, 1, 78, 13, 45};
void main( )
{
clrscr();
int i, n;
printf("\nOriginal array");
```

```
for(i=0; i < 5; i++)
printf("%4d", a[i]);
quicksort(a, 0, 4);
printf("\nThe sorted array is \n");
for(i=0; i < 5; i++)
printf("%4d", a[i]);
getch();
}
void quicksort (int a[], int low, int high)
{
int j;
if (low < high)
{
j = partition(a,low, high);
quicksort (a, low, j-1);
quicksort(a, j+1, high);
}
}
int partition(int a[], int low, int high)
{
int i, j, temp, key;
key = a[low];
i = low + 1;
j = high;
while (1)
{
while (i < high && key >= a[i])
```

```
i++;
while (key < a[j])
j--;
if (i < j)
{
temp = a[i];
a[i] = a[j];
a[j] = temp;
}
else
{
temp = a[low];
a[low] = a[j];
a[j] = temp;
}
return j;
}
}
```

Complexity of Quick Sort Algorithm

The Worst Case occurs when the list is sorted. Then the first element will require n comparisons to recognize that it remains in the first position. Furthermore, the first sublist will be empty, but the second sublist will have n-1 elements. Accordingly the second element require n-1 comparisons to recognize that it remains in the second position and so on.

$F(n)= n+(n-1)+(n-2)+\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots+2+1=n(n+1)/2 = O(n^2)$

Algorithm	Worst Case	Average Case	Best Case

Quick Sort	$n(n+1)/2 = O(n^2)$	O(n logn)	O(n logn)

4.2.5 Merge Sort

Merge Sort follows the rule of Divide and Conquer. But it doesn't divide the list into two halves. In merge sort the unsorted list is divided into N sub lists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sub lists, to produce new sorted sub lists, and at lasts one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of O(n log n). It is also a stable sort, which means the equal elements are ordered in the same order in the sorted list.

How Merge Sort Works

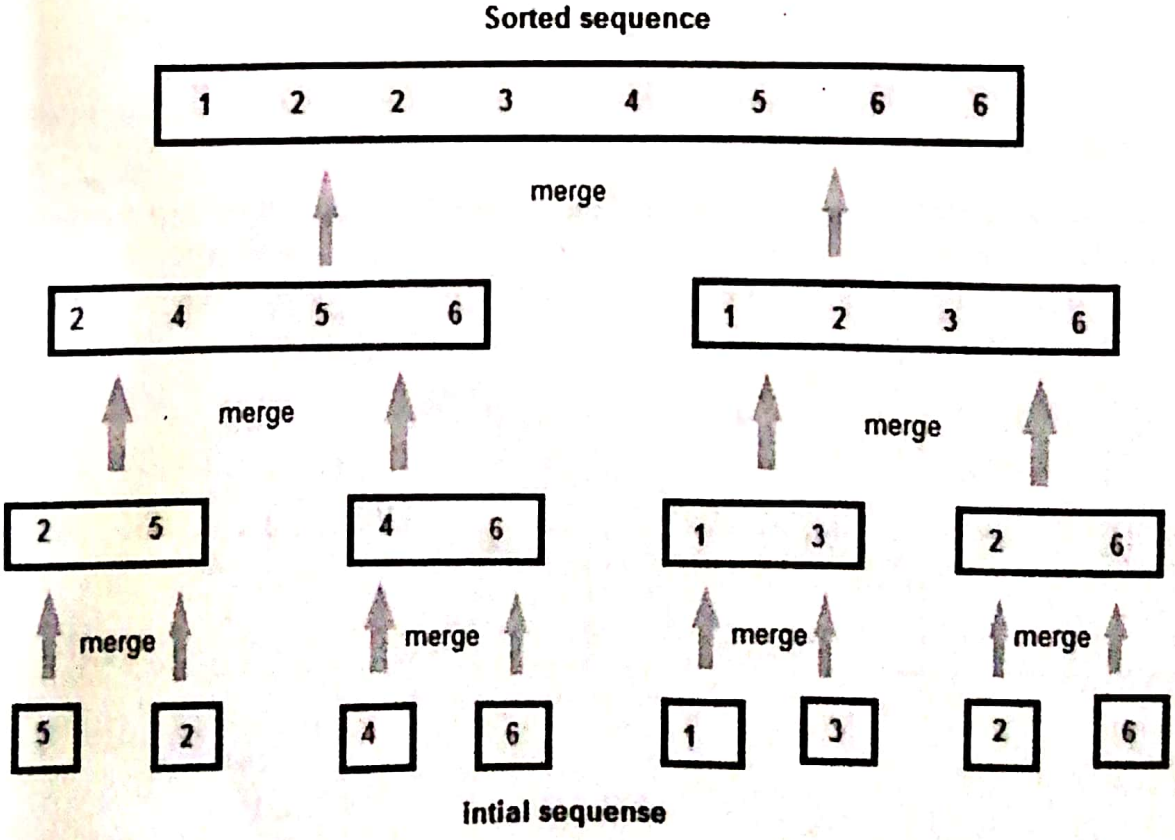
Suppose the array A contains 8 elements, each pass of the merge-sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows.

PASS 1. Merge each pair of elements to obtain the list of sorted pairs.

PASS 2. Merge each pair of pairs to obtain the list of sorted quadruplets.

PASS 3. Merge each pair of sorted quadruplets to obtain the two sorted subarrays.

PASS 4. Merge the two sorted subarrays to obtain the single sorted array.



Merge Sort Algorithm

```
int array[MAX];
void merge(int low, int mid, int high )
{
int temp[MAX];
int i = low;
int j = mid +1 ;
int k = low ;
while( (i <= mid) && (j <=high) )
{
if (array[i] <= array[ j])
temp[k++] = array[i++] ;
else
temp[k++] = array[ j++] ;
}
while( i <= mid )
temp[k++]=array[i++];
while( j <= high )
temp[k++]=array[j++];

for (i= low; i <= high ; i++)
array[i]=temp[i];
}
void merge_sort(int low, int high )
{
int mid;
if ( low != high )
{
mid = (low+high)/2;
merge_sort( low , mid );
merge_sort( mid+1, high );
merge(low, mid, high );
}
}
void main()
{
int i,n;
clrscr();
printf ("\nEnter the number of elements :");
scanf ("%d",&n);
for (i=0;i<n;i++)
{
printf ("\nEnter element %d :",i+1);
scanf ("%d",&array[i]);
```

```
}
printf ("\nUnsorted list is :\n");
for ( i = 0 ; i<n ; i++)
printf ("%d", array[i]);
merge_sort( 0, n-1 );
printf ("\nSorted list is :\n");
for ( i = 0 ; i<n ; i++)
printf ("%d", array[i]);
getch();
}
```

Complexity of Merge Sort Algorithm

Let f(n) denote the number of comparisons needed to sort an n-element array A using merge-sort algorithm. The algorithm requires at most logn passes. Each pass merges a total of n elements and each pass require at most n comparisons. Thus for both the worst and average case

$F(n) \leq n \log n$

Thus the time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Algorithm	Worst Case	Average Case	Best Case
Merge Sort	O(n logn)	O(n logn)	O(n logn)

4.2.6 Heap Sort

Heap Sort is one of the best sorting methods being in -place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts Creating a Heap of the unsorted list.

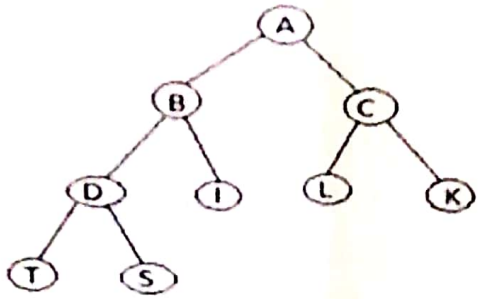
Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. What is a Heap?

Heap is a special tree-based data structure that satisfies the following special heap properties **Shape Property**: Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

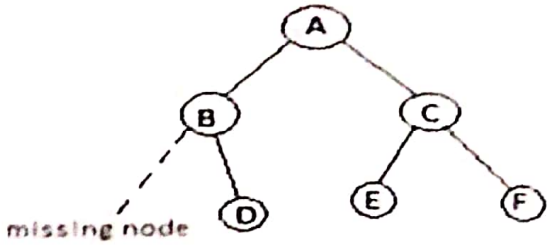
Heap Property: All nodes are either greater than or equal to or less than or equal to each of its children. If the parent nodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are smaller than their child nodes, heap is called Min-Heap.

How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.



Complete Binary Tree



In-Complete Binary Tree

Heap Sort Algorithm

- HEAPSORT(A)

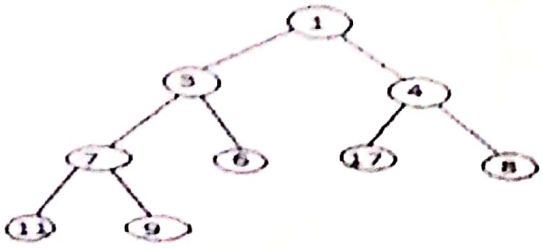
```
1.    BUILD-MAX-HEAP(A)
2.    for i ← length[A] downto 2
3.    do exchange A[1] ↔ A[i ]
4.    heap-size[A] ← heap-size[A] - 1
5.    MAX-HEAPIFY(A, 1)
```

- BUILD-MAX-HEAP(A)

```
1.    heap-size[A] ← length[A]
2.    for i ← length[A]/2 downto 1
3.    do MAX-HEAPIFY(A, i )
```

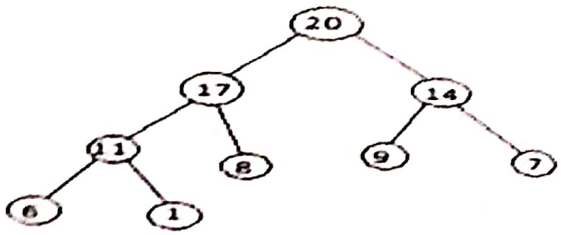
- MAX-HEAPIFY(A, i)

```
1.    l ← LEFT(i )
2.    r ← RIGHT(i )
3.    if l ≤ heap-size[A] and A[l] > A[i ]
4.    then largest ←l
5.    else largest ←i
6.    if r ≤ heap-size[A] and A[r] > A[largest]
7.    then largest ←r
8.    if largest = i
9.    then exchange A[i ] ↔ A[largest]
10.   MAX-HEAPIFY(A, largest)
```



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Complexity of Heap Sort Algorithm

The heap sort algorithm is applied to an array A with n elements. The algorithm has two phases, and we analyze the complexity of each phae separately.

Phase 1. Suppose H is a heap. The number of comparisons to find the appropriate place of a new element item in H cannot exceed the depth of H. Since H is complete tree, its depth is bounded by $\log_2 m$ where m is the number of elements in H. Accordingly, the total number g(n) of comparisons to insert the n elements of A into H is bounded as

$$g(n) \leq n \log_2 n$$

Phase 2. If H is a complete tree with m elements, the left and right subtrees of H are heaps and L is the root of H Reheaping uses 4 comparisons to move the node L one step down the tree H. Since the depth cannot exceeds $\log_2 m$, it uses $4\log_2 m$ comparisons to find the appropriate place of L in the tree H.

$$h(n) \leq 4n\log_2 n$$

Thus each phase requires time proportional to $n\log_2 n$, the running time to sort n elements array A would be $n\log_2 n$.

4.3 Hashing

4.3.1 Hash Function

A hash function is a function that:

1. When applied to an Object, returns a number
2. When applied to equal Objects, returns the same number for each
3. When applied to unequal Objects, is very unlikely to return the same number for each.

Suppose we were to come up with a "magic function" that, given a value to search for, would tell us exactly where in the array to look

1. If it's in that location, it's in the array
2. If it's not in that location, it's not in the array

A hash function is a function that makes hash of its inputs. Suppose our hash function gave us the following values:

hashCode("apple") = 5

hashCode("watermelon") = 3

hashCode("grapes") = 8

hashCode("cantaloupe") = 7

hashCode("kiwi") = 0

hashCode("strawberry") = 9

hashCode("mango") = 6

hashCode("banana") = 2

Sometimes we want a map—a way of looking up one thing based on the value of another

- We use a key to find a place in the map
- The associated value is the information we are trying to look up

Hashing works the same for both sets and maps

A perfect hash function would tell us exactly where to look

In general, the best we can do is a function that tells us where to start looking!

4.3.2 Collision Resolution Techniques

When two values hash to the same array location, this is called a collision

Collisions are normally treated as "first come, first served"—the first value that hashes to the location gets it

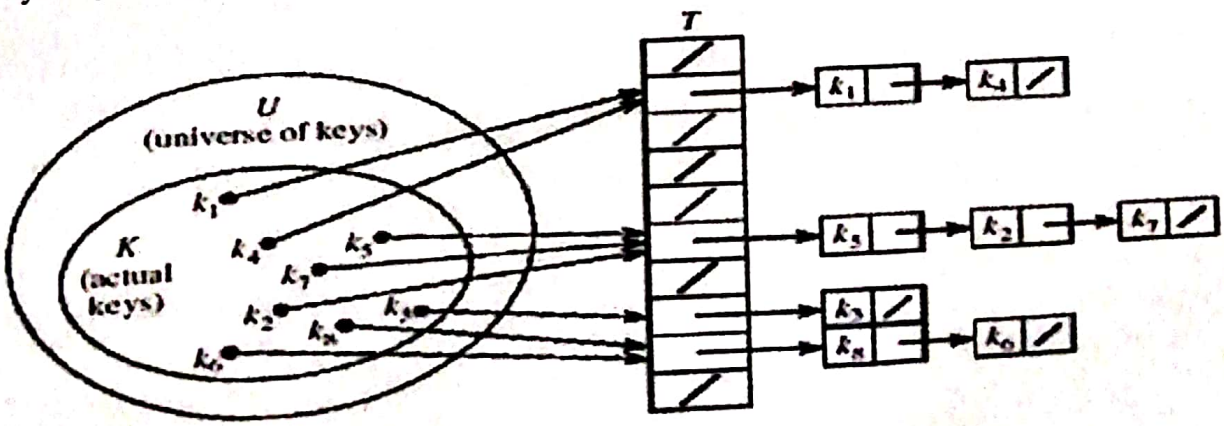
We have to find something to do with the second and subsequent values that hash to this same location.

There are two broad ways of collision resolution:

1. Separate Chaining: An array of linked list implementation.
2. Open Addressing: Array-based implementation
   - (i) Linear probing (linear search)
   - (ii) Quadratic probing (nonlinear search)
   - (iii) Double hashing (uses two hash functions)

Separate Chaining

- The hash table is implemented as an array of linked lists.
- Inserting an item, r, which hashes at index i is simply insertion into the linked list at position i.
- Synonyms are chained in the same linked list



- Retrieval of an item, r, with hash address, i, is simply retrieval from the linked list at position i.
- Deletion of an item, r, with hash address, i, is simply deleting r from the linked list at position i

Example: Load the keys 23, 13, 21, 14, 7, 8, and 15 , in this order, in a hash table of size 7

using separate chaining with the hash function: h(key) = key % 7

h(23) = 23 % 7 = 2

h(13) = 13 % 7 = 6

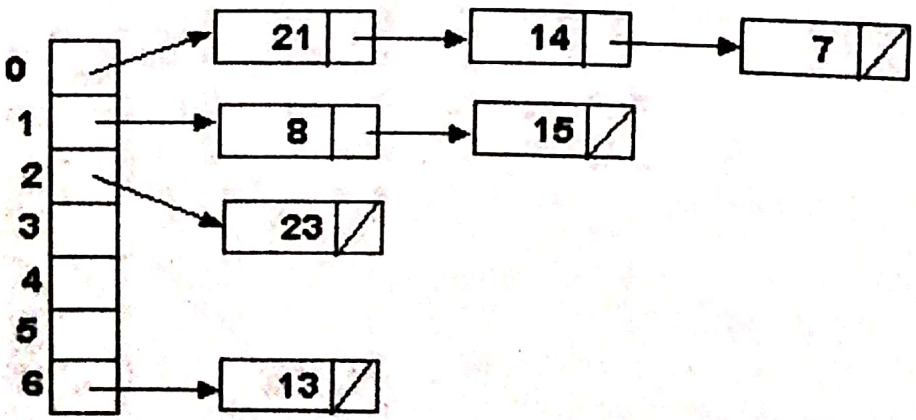
h(21) = 21 % 7 = 0

h(14) = 14 % 7 = 0 collision

h(7) = 7 % 7 = 0 collision

h(8) = 8 % 7 = 1

h(15) = 15 % 7 = 1 collision



4.1 Introduction

A **graph** is a mathematical structure consisting of a set of vertices (also called nodes) $\{v_1,v_2,\ldots,v_n\}$ and a set of edges $\{e_1,e_2,\ldots,e_n\}$. An edge is a pair of vertices $\{v_i,v_j\}$ $i,j \in \{1\ldots n\}$. The two vertices are called the edge *endpoints*. Graphs are ubiquitous in computer science. ⊆ Formally: G = (V, E), where V is a set and V × V

They are used to model real-world systems such as the Internet (each node represents a router and each edge represents a connection between routers); airline connections (each node is an airport and each edge is a flight); or a city road network (each node represents an intersection and each edge represents a block). The wireframe drawings in computer graphics are another example of graphs.

A graph may be either *undirected* or *directed*. Intuitively, an undirected edge models a "two-way" or "duplex" connection between its endpoints, while a directed edge is a one-way connection, and is typically drawn as an arrow. A directed edge is often called an *arc*. Mathematically, an undirected edge is an unordered pair of vertices, and an arc is an ordered pair. The maximum number of edges in an undirected graph without a self-loop is $n(n - 1)/2$ while a directed graph can have at most n^2 edges ∈ ⟺ ∈

G = (V, E) undirected if for all v, w V: (v, w) E (w, v) E.

Otherwise directed.For eg.

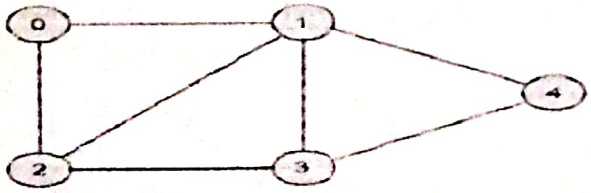


Fig4.1:Undirect Graph

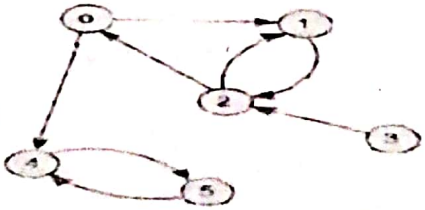


Fig4.2:Direct Graph

Graphs can be classified by whether or not their edges have **weights**. In **Weighted graph**, edges have a weight. Weight typically shows cost of traversing

Example: weights are distances between cities

In **Un-weighted graph**, edges have no weight. Edges simply show connections.

4.2 Terminology

- A **graph** consists of:
- A set, V, of **vertices** (nodes)

- A collection, E, of pairs of vertices from V called **edges** (arcs)
- **Edges**, also called arcs, are represented by **(*u*, *v*)** and are either:
- **Directed** if the pairs are ordered **(*u*, *v*)**
- *u* the **origin**
- *v* the **destination**

- **Undirected End-vertices** of an edge are the **endpoints** of the edge.

- Two vertices are **adjacent** if they are endpoints of the same edge.

- An edge is **incident** on a vertex if the vertex is an endpoint of the edge.
- **Outgoing edges** of a vertex are directed edges that the vertex is the origin. **Incoming edges** of a vertex are directed edges that the vertex is the destination.

- **Degree** of a vertex, *v*, denoted *deg*(*v*) is the number of incident edges.

- **Out-degree**, *outdeg*(*v*), is the number of outgoing edges.
- **In-degree**, *indeg*(*v*), is the number of incoming edges.

- **Parallel edges** or multiple edges are edges of the same type and end-vertices

- **Self-loop** is an edge with the end vertices the same vertex
- **Simple graphs** have **no** parallel edges or self-loops

- **Path** is a sequence of alternating vetches and edges such that each successive vertex is connected by the edge. Frequently only the vertices are listed especially if there are no paralleledges.

- **Cycle** is a path that starts and end at the same vertex.

- **Simple path** is a path with distinct vertices.
- **Directed path** is a path of only directed edges
- **Directed cycle** is a cycle of only directed edges.
- **Sub-graph** is a subset of vertices and edges.
- **Spanning sub-graph** contains all the vertices.

- **Connected graph** has all pairs of vertices connected by at least one path.
- **Connected component** is the maximal connected sub-graph of a unconnected graph.
- **Forest** is a graph without cycles.
- **Tree** is a connected forest (previous type of trees are called rooted trees, these are free trees)

- **Spanning tree** is a spanning sub graph that is also a tree.

4.3 Graph Representations

There are two standard ways of maintaining a graph G in the memory of a computer.

1. The sequential representation

2. The linked representation

4.3.1sequential Representation of Graphs Adjacency Matrix

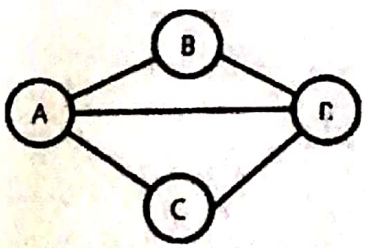
Representation

An **adjacency matrix** is one of the two common ways to represent a graph. The adjacency matrix shows which nodes are **adjacent** to one another. Two nodes are adjacent if there is an edge connecting them. In the case of a directed graph, if node j is adjacent to node i, there is an edge from i to j . In other words, if j is adjacent to i, you can get from i to j by traversing one edge. For a given graph with n nodes, the adjacency matrix will have dimensions of nxn. For an unweighted graph, the adjacency matrix will be populated with Boolean values.

For any given node i, you can determine its adjacent nodes by looking at row (i,[1...n]) adjacency matrix. A value of true at (i,j) indicates that there is an edge from node i to node j, and false indicating no edge. In an undirected graph, the values of (i,j) and (j,i)will be equal. In a weighted graph, the boolean values will be replaced by the weight of the edge connecting the two nodes, with a special value that indicates the absence of an edge.

The memory use of an adjacency matrix is $O(n^2)$.

- Example undirected graph (assume self-edges not allowed):



A B C D

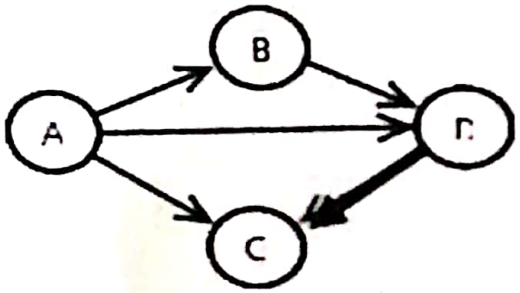
A 0 1 1 1

B 1 0 ∞ 1

C 1 ∞ 0 1

D 1 1 1 0

- Example directed graph (assume self-edges allowed):



A B C D

A ∞ 1 1 1

B ∞ ∞ ∞ 1

C ∞ ∞ ∞ ∞

D ∞ ∞ 1 ∞

1.8 Searching

1.8.1 Linear Search

Linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest search algorithm; it is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list; and so is its expected cost, if all list elements are equally likely to be searched for. Therefore, if the list has more than a few elements, other methods (such as binary search or hashing) will be faster, but they also impose additional requirements.

1.8.1.1 How Linear Search works

Linear search in an array is usually programmed by stepping up an index variable until it reaches the last index. This normally requires two comparisons for each list item: one to check whether the index has reached the end of the array, and another one to check whether the item has the desired value.

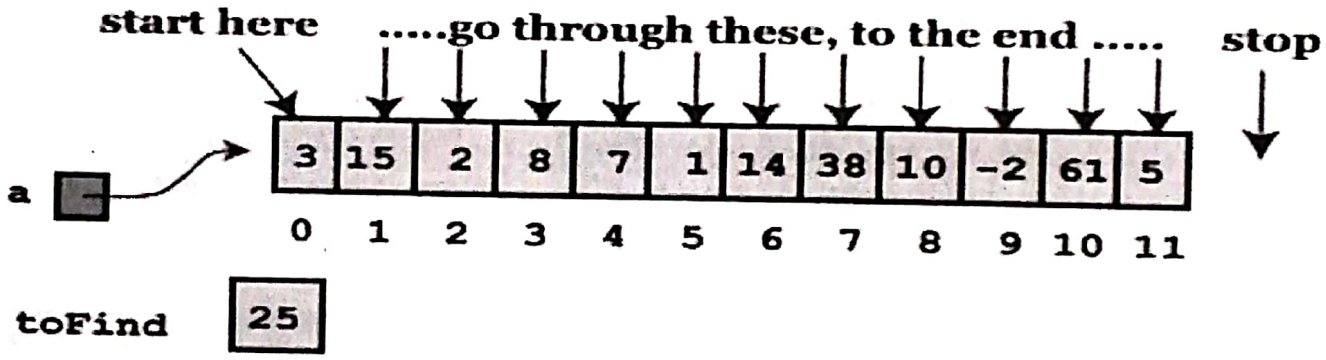


Fig 1.16 Linear Search

1.8.1.2 Linear Search Algorithm

1. Repeat For J = 1 to N

2. If (ITEM == A[J]) Then

3. Print: ITEM found at location J

4. Return [End of If]

   [End of For Loop]

5. If (J > N) Then

6. Print: ITEM doesn't exist

   [End of If]

1.8.1.3 CODE

```
int a[10],i,n,m,c=0, x;
printf("Enter the size of an array: ");
scanf("%d",&n);

printf("Enter the elements of the array: ");
for(i=0;i<=n-1;i++){
    scanf("%d",&a[i]);
}
printf("Enter the number to be search: ");
scanf("%d",&m);
for(i=0;i<=n-1;i++){
    if(a[i]==m){
      x=I;
       c=1;
       break;
    }
}
if(c==0)
    printf("The number is not in the list");
else
    printf("The number is found at location %d", x);
}
```

1.8.1.4 Complexity of linear Search

Linear search on a list of n elements. In the worst case, the search must visit every element onc
This happens when the value being searched for is either the last element in the list, or is not i
the list. However, on average, assuming the value searched for is in the list and each list elemer
is equally likely to be the value searched for, the search visits only n/2 elements. In best case th
array is already sorted i.e O(1)

Table 1.1 Complexity of linear Search

Algorithm	Worst Case	Average Case	Best Case
Linear Search	O(n)	O(n)	O(1)

1.8.2 Binary Search

A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

1.8.2.1 How Binary Search Works

Searching a sorted collection is a common task. A dictionary is a sorted list of word definitions. Given a word, one can find its definition. A telephone book is a sorted list of people's names, addresses, and telephone numbers. Knowing someone's name allows one to quickly find their telephone number and address.

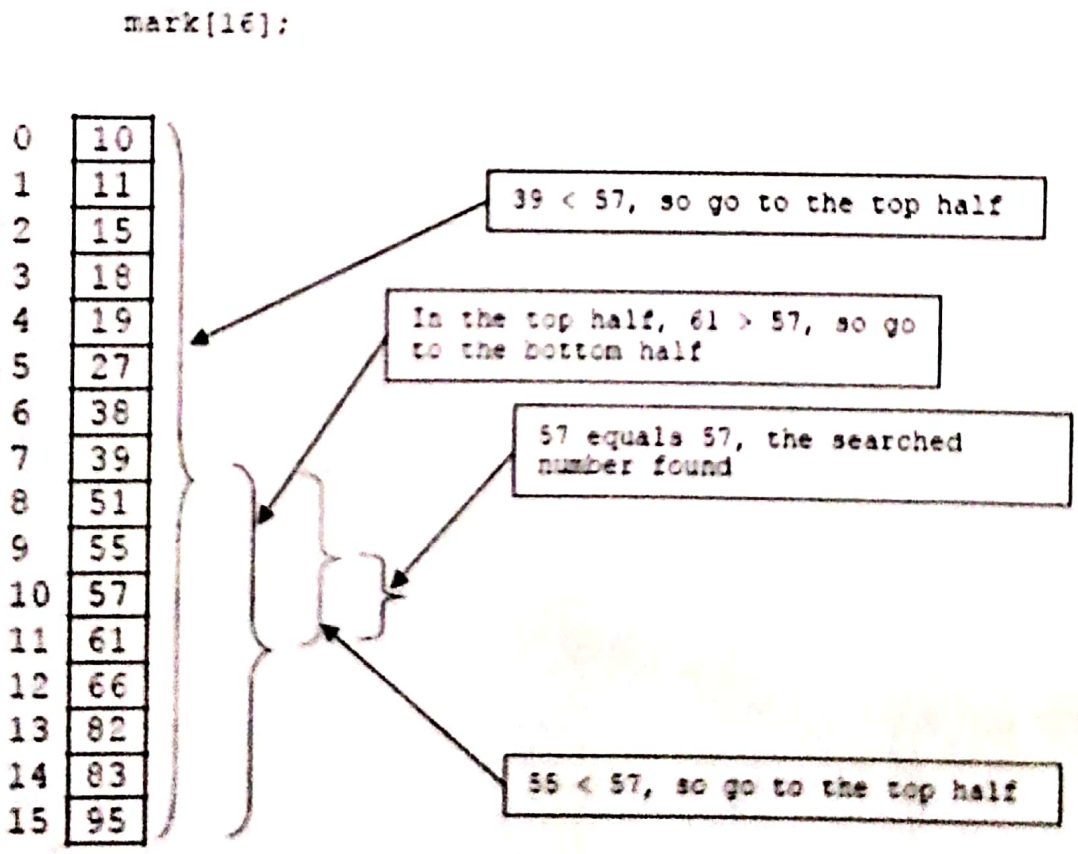


Fig 1.17 Binary Search

1.8.2.2 Binary Search Algorithm

1. Set BEG = 1 and END = N
2. Set MID = (BEG + END) / 2
3. Repeat step 4 to 8 While (BEG <= END) and (A[MID] ≠ ITEM)
4. If (ITEM < A[MID]) Then

5. Set END = MID – 1
6. Else
7. Set BEG = MID + 1
[End of If]

8. Set MID = (BEG + END) / 2
9. If (A[MID] == ITEM) Then
10. Print: ITEM exists at location MID
11. Else
12. Print: ITEM doesn't exist
[End of If]

13. Exit

1.8.2.3 CODE

```
#include<stdio.h>
void main()
{
   int list[50];
   int size,i,pos=-1,val;
   int lb=0,ub,low,high,mid;

   printf("Enter the size for the list: ");
   scanf("%d",&size);
   ub=size-1;

   printf("Enter the element's in the list ( in ascending order ),\n");
   for(i=0;i<size;i++)
   {
      printf("Element %d: ",i+1);
      scanf("%d",&list[i]);
   }

   system("cls");
   printf("Enter the value you want to search: ");
   scanf("%d",&val);

   low=lb; high=ub;

   while(low<=high)
   {
      mid=(low+high)/2;

      if(list[mid]==val)
      {
         pos=mid;
         break;
      }
      else
      {
```

```
        if(list[mid]>val)
          high=mid-1;
        else
          low=mid+1;
      }
  }

  if(pos != -1)
    printf("The number is found at %d",pos+1);
  else
    printf("Number not found in the list.");


}
```

1.8.2.4 Complexity of Binary Search

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time.

Algorithm	Worst Case	Average Case	Best Case
Binary Search	O(nlogn)	O(nlogn)	O(1)

9.7 RADIX SORT

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. (Here the radix is 26, the 26 letters of the alphabet.) Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes.

The radix sort is the method used by a card sorter. A card sorter contains 13 receiving pockets labeled as follows:

9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 11, 12, R (reject)

Each pocket other than R corresponds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the obvious way and hence use only the first 10 pockets of the sorter. The sorter uses a radix reverse-digit sort on numbers. That is, suppose a card sorter is given a collection of cards where each card contains a 3-digit number punched in columns 1 to 3. The cards are first sorted according to the units digit. On the second pass, the cards are sorted according to the tens digit. On the third and last pass, the cards are sorted according to the hundreds digit. We illustrate with an example.

Example 9.8

Suppose 9 cards are punched as follows:

348, 143, 361, 423, 538, 128, 321, 543, 366

Given to a card sorter, the numbers would be sorted in three phases, as pictured in Fig. 9.6:

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(a) First pass

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543		366			
366										
348					348					
538				538						
128			128							

(b) Second pass

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128				538				
538										
143		143				543				
543										
348				348						
361				361						
366				366						

(c) Third pass

Fig. 9.6

(a) In the first pass, the units digits are sorted into pockets. (The pockets are pictured upside down, so 348 is at the bottom of pocket 8.) The cards are collected pocket by pocket, from pocket 9 to pocket 0. (Note that 361 will now be at the bottom of the pile and 128 at the top of the pile.) The cards are now reinput to the sorter.

(b) In the second pass, the tens digits are sorted into pockets. Again the cards are collected pocket by pocket and reinput to the sorter.

(c) In the third and final pass, the hundreds digits are sorted into pockets.

When the cards are collected after the third pass, the numbers are in the following order:

128, 143, 321, 348, 361, 366, 423, 538, 543

Thus the cards are now sorted.

The number C of comparisons needed to sort nine such 3-digit numbers is bounded as follows:

$$C \leq 9*3*10$$

The 9 comes from the nine cards, the 3 comes from the three digits in each number, and the 10 comes from radix $d = 10$ digits.

Complexity of Radix Sort

Suppose a list A of n items $A_1, A_2, \ldots, A_n$ is given. Let d denote the radix (e.g., $d = 10$ for decimal digits, $d = 26$ for letters and $d = 2$ for bits), and suppose each item A_i is represented by means of s of the digits:

$$A_i = d_{i1}d_{i2} \ldots d_{is}$$

The radix sort algorithm will require s passes, the number of digits in each item. Pass K will compare each d_{iK} with each of the d digits. Hence the number $C(n)$ of comparisons for the algorithm is bounded as follows:

$$C(n) \leq d*s*n$$

Although d is independent of n, the number s does depend on n. In the worst case, $s = n$, so $C(n) = O(n^2)$. In the best case, $s = \log_d n$, so $C(n) = O(n \log n)$. In other words, radix sort performs well only when the number s of digits in the representation of the A_i's is small.

Another drawback of radix sort is that one may need $d*n$ memory locations. This comes from the fact that all the items may be "sent to the same pocket" during a given pass. This drawback may be minimized by using linked lists rather than arrays to store the items during a given pass. However, one will still require $2*n$ memory locations.

File Organizations refers to the ways or methods used to store and access records of the file on storage medium. **File access** refers to the way records are found. If every operation associated with the records of a file, that is, retrieval, update, insertion & deletion is based only on the primary key value, then the associated access method is called primary access method.

Otherwise, if some operations involves accessing records based also on secondary key values, then the associated access method is called a secondary access method.

FILE ORGANIZATION SELECTION

File organization selection depends primarily on the type of operations that are to be allowed. We give a chart, which shows the classification of access methods of file organizations.

SERIAL FILE ORGANIZATION

In this organization, records are stored in order of their occurrence. In this organization records are not stored according to specified primary key value. To access any record, we start with first record and go on searching to second record and so on until we get the required record.

Generally, magnetic tape is used as storage media for this organization. This file organization is not of much importance and is generally not considered.

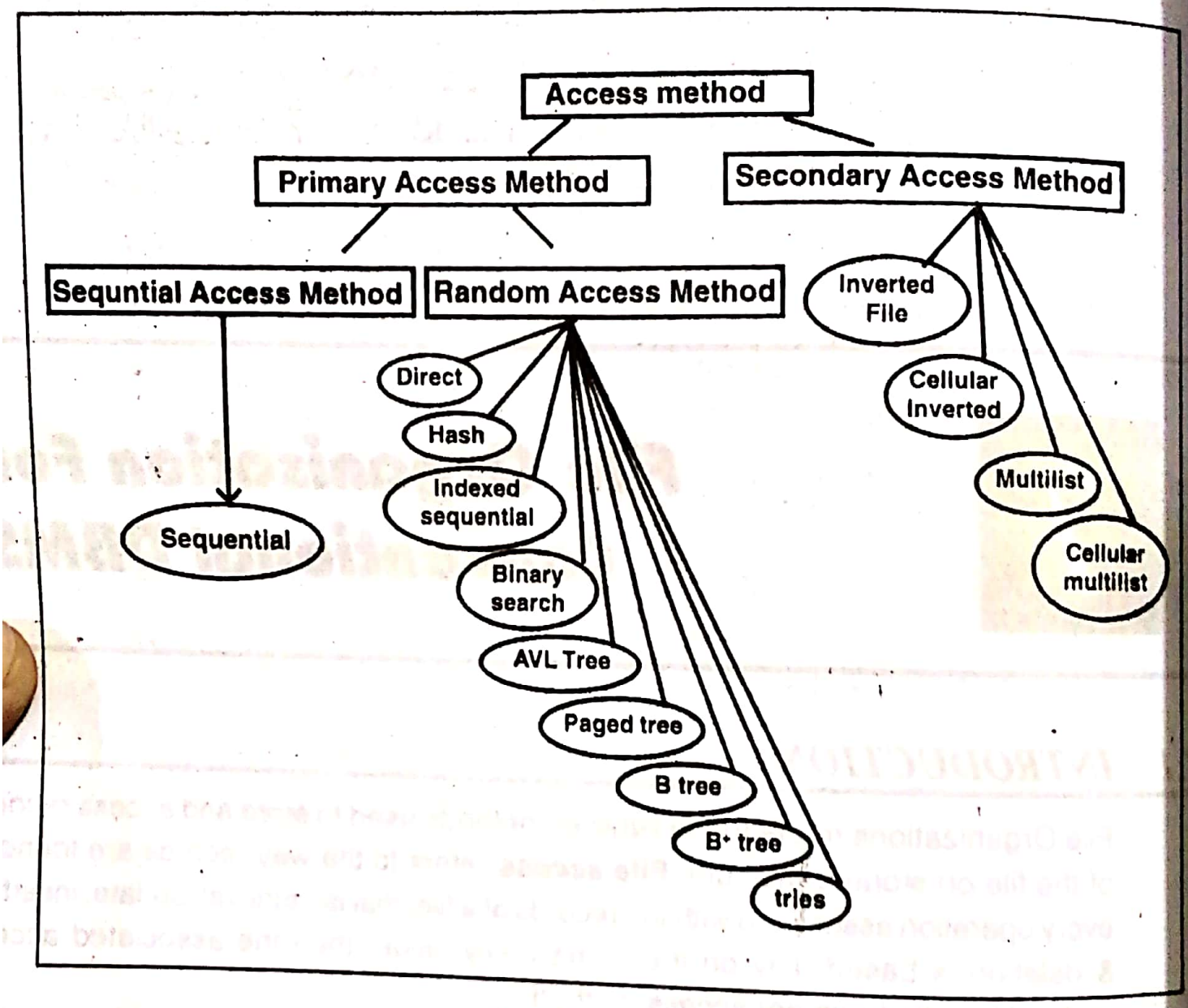


FIGURE 7.1

SEQUENTIAL FILE ORGANIZATION

Under this file organization, records are written in the same sequence in which they are collected but they are written on the disk according to the ascending or descending order of the record key. To access records in sequential files, it is necessary to read the file from beginning to examine each record in sequence until the desired record is located.

In a sequentially organized file, the records are written consecutively when the file is created and must be accessed consecutively when the file is later used for input as shown in fig. 7.2.

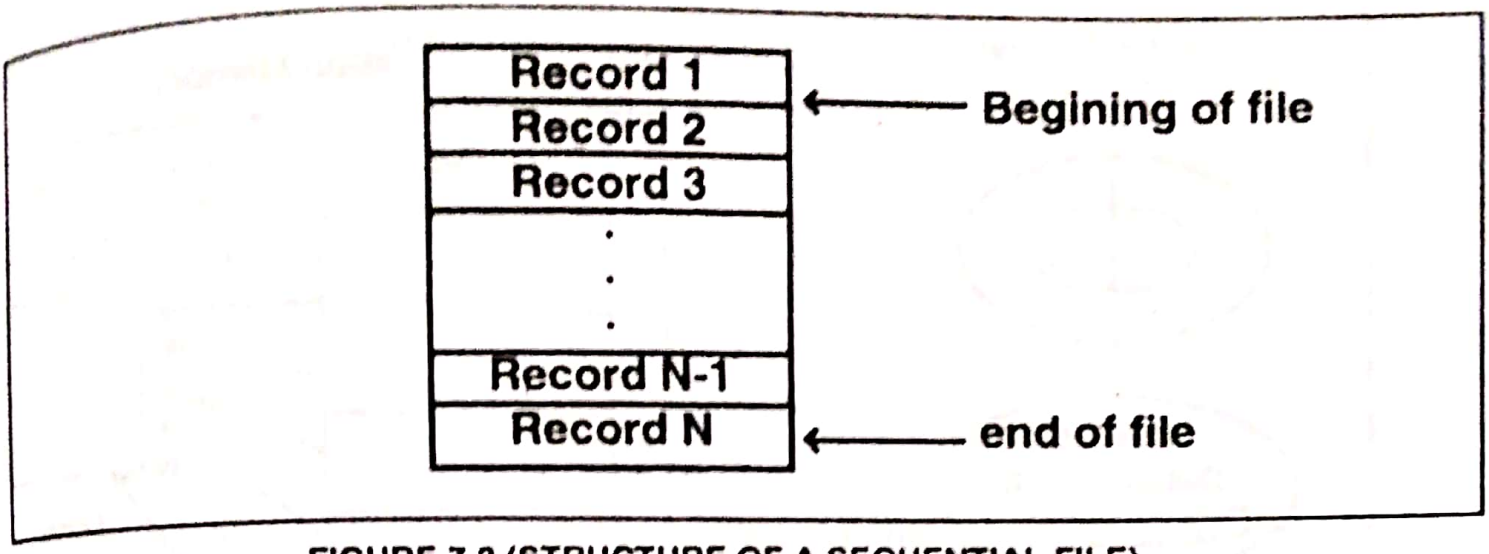


FIGURE 7.2 (STRUCTURE OF A SEQUENTIAL FILE)

All types of external storage devices support a sequential file organization. Devices which are strictly sequential in nature are magnetic tape, paper tape, readers, card readers, tape cassetts & line printers. A sequential file is physically placed on disk by storing the sequence of records in adjacent locations on track. If file is larger than amount of space on track, then records are stored on adjacent tracks.

It is often advantageous to group a number of logical records into block. To handle the read & write command issued for a particular device, we use buffer between external storage & the data area of a program (Note a buffer is a section of main memory which is equal in size to the maximum size of a block of logical records used by a program). The data management routine of a operating system use buffers for the blocking & deblocking of records.

To understand the concept of blocking and deblocking of records using a buffer, consider the READ command. When the first READ statement is execute, a block of records is moved from external storage to a buffer. The first record in the block is then transferred to the program's data area as shown in fig. 7.3. For each subsequent execution of READ statements, the next successive record in the buffer is transmitted to the data area.

Only after every record in the buffer has been moved to the data area, in response to READ statements, does the next READ statement cause another block to be transferred to the buffer from external storage. The new records in the buffer are moved to the data area, and this entire process is repeated for each block that is read.

In this same way, WRITE statements cause the transfer of program data to the buffer. When the buffer becomes full, then the block is written on the external device.

7.6 DIRECT (OR RANDOM) FILE ORGANISATION

It offers an effective way to organize data when there is a need to access individual records directly. In a direct file, a transformation or mapping is made from the key of a record to the address of the storage location. One mechanism used. for generation this transformation is called a hashing algorithm In the previous section, we studied the hashing algorithm and collision resolution techniques. Direct files are stored on DASD (Direct Access storage Devices).

7.6.1 Direct (or random) File Operations

Various direct file operations are as follows

(I) **Creation of Direct Files :**

For creating a direct file, a hashing algorithms and collision handling algorithm must be specified by programmer. Using the given hashing algorithm, the primary key value of input record is translated into physical address and then the record is written into the location corresponding to the physical address. This process continues until all the input records have been written to the storage medium.

(II) **Retrieving from Direct file (searching) :**

To retrieve (or search or read) a record from a direct file, user must supply the primary key value of that record. Using the given hashing algorithm, the primary key value is translated into physical address and program retrieves the record from that address. If there is no record at that address, then the desired record is not present in the file. If there is a record other than the desired record at the transformed address, the specified collision method is used to retrieve the desired record.

(III) **Updating a Direct file :**

To add, delete or modify the record, primary key value of the record is translated into physical address using hashing algorithm first. Then

(a) **Addition of a new record :** The record is inserted at the calculated physical location. If the calculated physical location is not vacant, specified collision techniquesis used to insert the record.

(b) **Deletion of an Existing record :** The record is searched at the calculated physical location and thus the record-status-code is changed to deleted or vacant. If the record to be deleted does not exist in the file then error message should be displayed.

(c) **Modification of a record :** The record is retrieved from the calculated physical location and specified modifications are made. The modified records rewritten to the file in the same location.

7.6.2 Advantages

1. Fast retrieval of records
2. Immediate updating of several files on a result of a single transaction is possible
3. There is no need to sort transactions.
4. Very suitable for ad-hoc and query requirements.
5. It is also possible to process direct file records sequentially in a record key sequence.
6. Updating of any record or group of records does not require the rewriting of the entire file.

7.6.3 Disadvantages

1. Expensive hardware & software resources are required.
2. Some wastage of storage space is involved, so less efficient in the use of storage space.
3. Algorithm to access a record may entail complex calculation
4. Records in the online files may be exposed to loss and security risks and costly backup procedure need to be established for record re-construction.
5. Addition and deletion of records are somewhat difficult to carry out.
6. Special security measures are necessary for online direct files that are accessible for several stations.

7 INDEXED SEQUENTIAL FILE ORGANISATION (ISAM)

The techinique is also known as Indexed Sequential Access Method (ISAM). This approach combines the advantages of both sequential & direct access file organisation. In this organisation, the records are organised in sequence but direct access to individual records is possible through an index. This index is analogous to a card catalog in a library.

The external storage devices that support both direct & sequential access are magnetic drums & magnetic disk. So an indexed sequential file is stored on to direct access storage. Indexed sequential files are composed of three areas.

(i) **Prime Area :** The prime area contains records of the file when the file is created or reorganized. Records in the prime area are in order by key.

(ii) **Overflow Area :** Records are placed in overflow area when additions to the file cannot fitted into the prime area.

(iii) **Index Area :** The indexes are used to locate a particular record for random processing. There are three types of indexes area:

(a) **Track Index :** The track index contains the address of the track to which the entry to associated & of the highest value of the keys for the records stored on that track.

(b) **Cylinder Index :** If file is large, the track index is long. To reduce the search time on the track index, another index called the cylinder index, is created. This index shows the highest key on each cylinder.

(c) **Master Index :** A master index is used for an extremely large file where a search of the cylinder index is too time consuming.

7.7.1 Operation of Index sequential file6

To understand the processing of an indexed sequential file. Let us consider an example. Suppose we have 20 records with primary key values stored in different tracks as shown in fig. 7.15. Note that each record has a unique key & that records are in order by key.

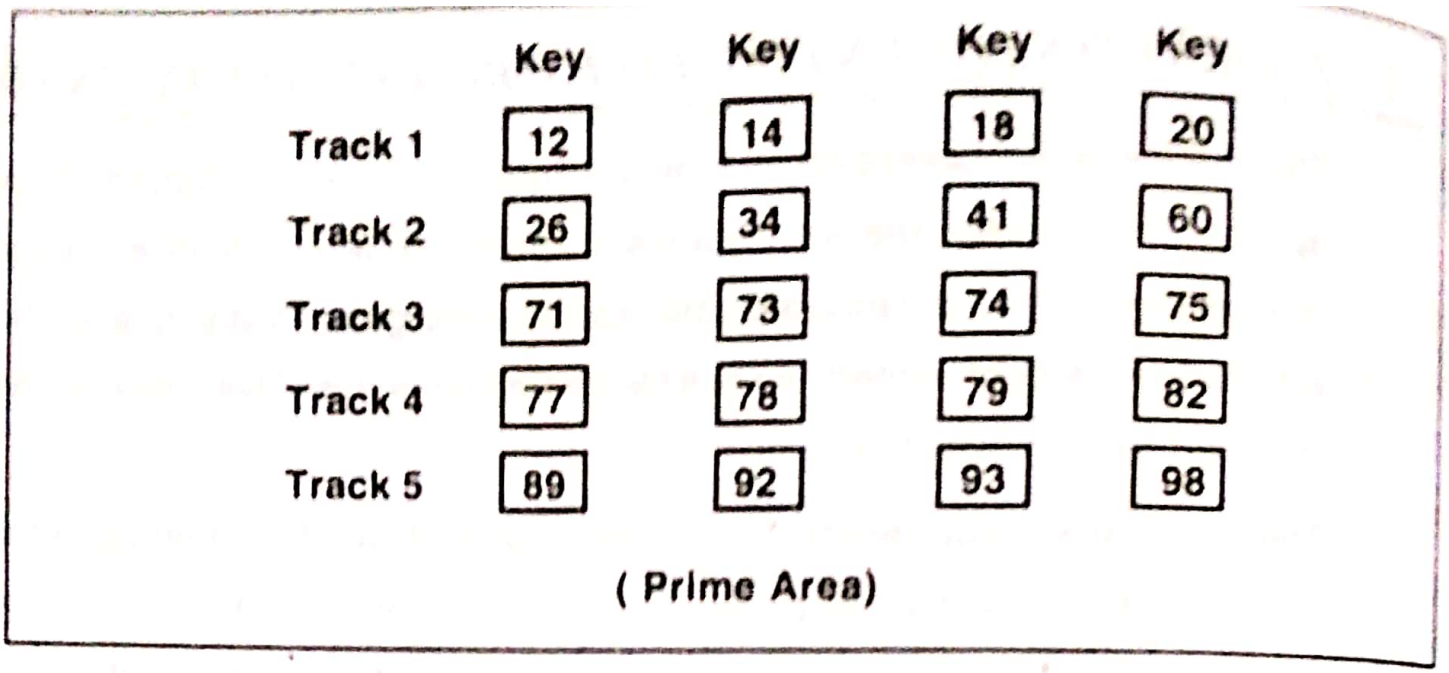


FIGURE 7.15

The track index for the above arrangement is shown in fig. 7.16. The track index contains the value of the highest key on each track.

Track No.	Highest Key on Track
1	20
2	60
3	75
4	82
5	98

FIGURE 7.16 TRACK INDEX

(1) **Creating Indexed Sequential File :** When an indexed sequential file is created all records are written into the prime area in sequence by key. The indexes are generated at this time.

(2) **Retrieving from Indexed Sequential File :** Suppose it is desired to read record 79. One way to access this record is to sequentially read the records in a file until record 79 is found. This may be a very slow process.

A faster way of finding record 79 is to use the track index.

The highest key of track 3 is 75, & the highest key on track 4 is 82. Therefore, if record 79 exist, it must be on track 4. Consequently, the next step is to read track 4 to find record 79.

Using the track index does not eliminate sequential searching, it just reduces the magnitude of the task. Instead of searching a large file of records, we are able to search a table that points toward the desired record location.

MULTI-LIST ORGANIZATION

The other fundamental approach to providing the linkage between an index and the file of data records is called multi-list organization. Like an inverted file, a multi-list file maintains an index for each secondary key. There is one entry in the secondary key index for each value that the secondary key presently has in the data file. The entry in the multi-list index for a key value has just one pointer to the first data record with the key value. The data record contains a pointer to the next data record with that key value and so forth. Thus there is a linked list of data records for each value of the secondary key. Multi-list chains usually are bi-directional and occasionally are circular to improve update performance.

Example : Fig. 7.23 and 7.24 show the multi-list indexes for secondary key sex and salary respectively.

Sex	Address
M	1
F	4

FIGURE 7.23

Salary	Address
9000	1
10000	5
11000	2

FIGURE 7.24

Fig. 7.25 shows the corresponding data file.

Employee

Record Address	Emp#	NAME	Sex	Next (for sex)	Salary	Next (for salary)
1	500	Amit	M	2	9000	3
2	600	Vikas	M	3	11000	4
3	700	Aman	M	6	9000	0
4	800	Savita	F	5	11000	6
5	900	Suman	F	0	10000	0
6	950	Mohan	M	0	11000	0

FIGURE 7.25

, One attractive characteristic of the multi-list approach is that index entries can be fixed length. Each value has associated with it just one pointer.

In multi-list approach, to respond to the following queries :

(1) How many employees are there with salary = 9000 ?

(2) How many employees are there with sex = 'M' ?

(3) How many employees are there with sex = 'F' ?

(4) How many employees are there with salary > 10000

The data records must be accessed. In order to be able to answer such queries (i.e. counting queries), each value entry in the multi-list index may have the length of the entry's linked list of records also. Fig. 7.26 & Fig. 7.27 show this variation for the sex and salary indexes.

Sex	Address	Length
M	1	4
F	4	2

FIGURE 7.26

Salary	Address	Length
9000	1	2
10000	5	1
11000	2	3

FIGURE 7.27

Two variations on the basic multi-list structure are :

(a) Controlled-length multi-list file :

In this file, a maximum lengths is imposed upon the linked lists of data records. If a secondary key value is possessed by more data records than that length allows, then the key value would appear more than once in the index and there would be more than one linked list of data records with that value.

(b) Cellular multi-list file :

In this file, the list structures are determined in part by the characteristics of storage. For example, a cell could be defined as a cylinder, as a track or an a page. A linked list is not allowed to cross cell boundaries. If there are four tracks that have the same secondary value, then there would be four index entries and four linked lists for that value. This multi-list variation can be useful to reducing disk arm movement requirements and I/O accesses.

Overview of Physical Storage Media

Several types of data storage exist in most computer systems. These storage media are classified by the speed with which data can be accessed, by the cost per unit of data to buy the medium, and by the medium's reliability. Among the media typically available are these:

• **Cache**. The cache is the fastest and most costly form of storage. Cache memory is small; its use is managed by the computer system hardware. We shall not be concerned about managing cache storage in the database system.

• **Main memory**. The storage medium used for data that are available to be operated on is main memory. The general-purpose machine instructions operate on main memory. Although main memory may contain many megabytes of data, or even gigabytes of data in large server systems, it is generally too small (or too expensive) for storing the entire database. The contents of main memory are usually lost if a power failure or system crash occurs.

• **Flash memory**. Also known as *electrically erasable programmable read-only memory* (*EEPROM*), flash memory differs from main memory in that data survive power failure. Reading data from flash memory takes less than 100 nanoseconds (a nanosecond is 1/1000 of a microsecond), which is roughly as fast as reading data from main memory. However, writing data to flash memory is more complicated — data can be written once, which takes about 4 to 10 microseconds, but cannot be overwritten directly. To overwrite memory that has been written already, we have to erase an entire bank of memory at once; it is then ready to be written again. A drawback of flash memory is that it can support only a limited number of erase cycles, ranging from 10,000 to 1 mil- lion. Flash memory has found popularity as a

• **Magnetic-disk storage**. The primary medium for the long-term on-line storage of data is the magnetic disk. Usually, the entire database is stored on magnetic disk. The system must move the data from disk to main memory so that they can be accessed. After the system has performed the designated operations, the data that have been modified must be written to disk.

• **Optical storage**. The most popular forms of optical storage are the *compact disk* (CD), which can hold about 640 megabytes of data, and the *digital video disk* (DVD) which can hold 4.7 or 8.5 gigabytes of data per side of the disk (or up to 17 gigabytes on a two-sided disk). Data are stored optically on a disk, and are read by a laser. The optical disks used in read-only compact disks (CD-ROM) or read-only digital video disk (DVD-ROM) cannot be written, but are supplied with data prerecorded.

There are "record-once" versions of compact disk (called CD-R) and digital video disk (called DVD-R), which can be written only once; such disks are also called **write-once, read-many** (WORM) disks. There are also "multiple-write" versions of compact disk (called CD-RW) and digital video disk (DVD-RW and DVD-RAM), which can be written multiple times. Recordable compact disks are magnetic – optical storage devices that use optical means to read magnetically encoded data. Such disks are useful for archival storage of data as well as distribution of data.

Jukebox systems contain a few drives and numerous disks that can be loaded into one of the drives automatically (by a robot arm) on demand.

• **Tape storage**. Tape storage is used primarily for backup and archival data. Although magnetic tape is much cheaper than disks, access to data is much slower, because the tape must be accessed sequentially from the beginning. For this reason, tape storage is referred to as **sequential-access** storage. In contrast, disk storage is referred to as **direct-access** storage because it is possible to read data from any location on disk.

Tapes have a high capacity (40 gigabyte to 300 gigabytes tapes are currently available), and can be removed from the tape drive, so they are well suited to cheap archival storage. Tape jukeboxes are used to hold exceptionally large collections of data, such as remote-sensing data from satellites, which could include as much as hundreds of terabytes (1 terabyte = 1012 bytes), or even a petabyte (1 petabyte = 1015 bytes) of data.

Floyd Warshall Algorithm | DP-16

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

Input:

```
       graph[][] = { {0,   5,  INF, 10},
                     {INF,  0,  3,  INF},
                     {INF, INF, 0,   1},
                     {INF, INF, INF, 0} }
```

which represents the following graph

```
             10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3
```

Note that the value of graph[i][j] is 0 if i is equal to j

And graph[i][j] is INF (infinite) if there is no edge from verte

Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.

Time Complexity: O(V^3)

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

```
#include

#define INF INT_MAX
.........................
if ( dist[i][k] != INF &&
     dist[k][j] != INF &&
     dist[i][k] + dist[k][j] < dist[i][j]
    )
 dist[i][j] = dist[i][k] + dist[k][j];
.........................
```

Dijkstra's Algorithm

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest - path estimate, insert u into S and relaxes all edges leaving u.

Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the **greedy strategy**.

Analysis: The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big - O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract - Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is O ($|V^2|+|E|=O(V^2)$).

Example: