Introduction to Trees:

- Arrays, stacks, Queues and Linked lists are known as linear data structures because elements are arranged in a linear fashion (One dimensional representation).
- Another data structure is tree, where elements appear in a non linear fashion, which requires two dimensional representations.

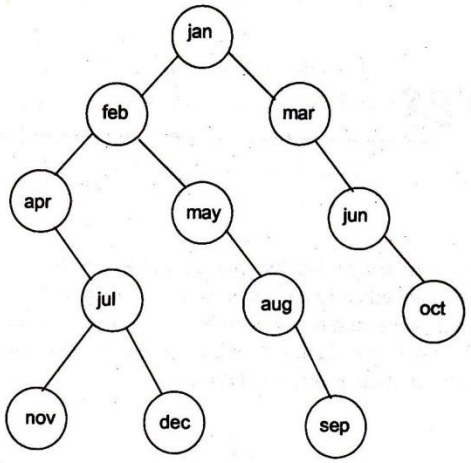


Figure (f) : Tree

- **Tree:** A tree is a non-linear data structure in which items are arranged in a sorted sequence. It is used to represent hierarchical relationship existing amongst several data items.
- A tree is a finite set of one or more nodes such that
  i. There is a specially designated node called the root,
  ii. Remaining nodes are partitioned into n (n > 0) disjoint sets T1,T2,T3,.....,Tn, where each Ti (i=1,2,.....,n) is a tree; T1,T2,.....,Tn are called sub trees of the root.

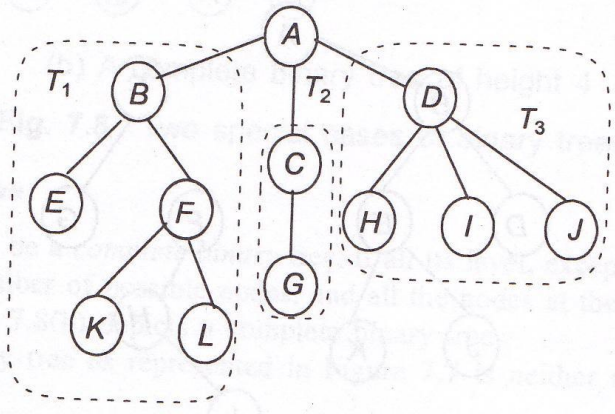


Fig (g): Tree

Basic Terminologies:

1) **Root:** It is specially designed data item in a tree. It is the first in the hierarchical arrangement of data items.

Ex: From above tree, "Jan" is a root node.

2) **Node:** Each data item in a tree is called node. It is the basic structure in a tree. It stores the actual data and links to another node.

Fig (h) represents the structure of the node.

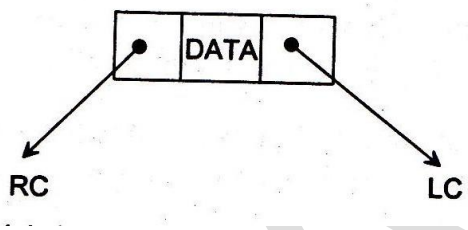


Fig (h): Structure of Node

3) **Degree of Node:** It is the number of sub trees of a node in a given tree.

4) **Parent:** parent of the node is the immediate predecessor of a node. Here, X is the parent of Y and Z as shown in fig (i).

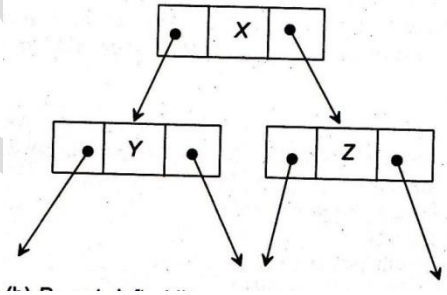


Fig (i): Parent, Left Child and Right Child

5) **Leaf or Terminal node:** The node which is at the end and which does not have any child is called leaf node. In fig (j), H, I, K, L and M are the leaf nodes. Leaf node is also known as terminal node which contains degree zero.

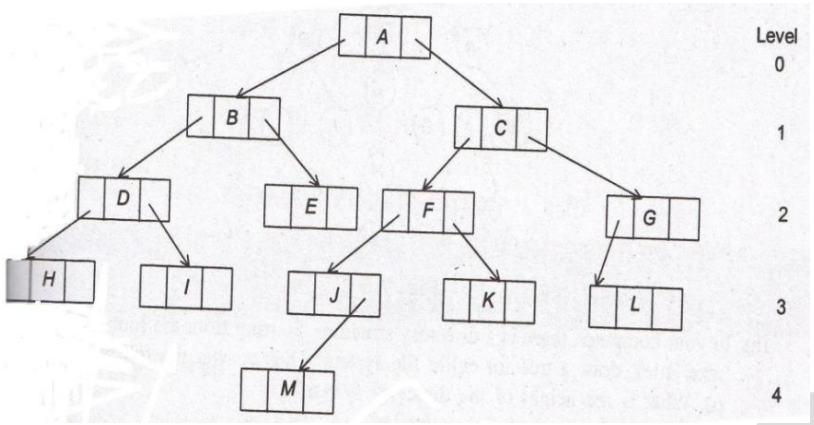
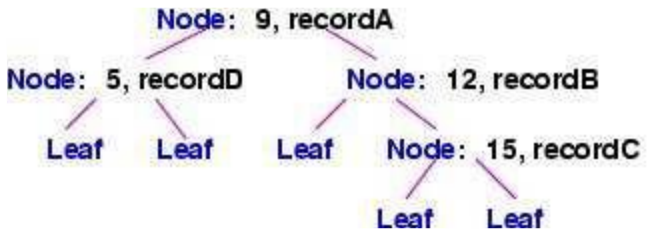


Fig (j): Tree

6) **Level:** Level is a rank in the hierarchy. The root node is always at level 0. If a node is at level L then its child is at L+1 and parent is at level L-1. This is true for all nodes except the root node.
7) **Sibling:** The nodes which have the same parent are called siblings. They are also called brothers. From figure (j) J and K are siblings.
8) **Height:** The maximum no. of nodes that is possible in a path starting from root node to a leaf node is called height of a tree. It is the maximum level of any node in a given tree. The term height is also used to denote the depth.
9) **Branch:** Branch means a link between parent and its child.
10) **Edge:** It is connecting line of two nodes. The line drawn from one node to another node is called an edge.
11) **Path:** It is a sequence of consecutive edges from the source node to destination node.
12) **Forest:** it is a set of disjoint trees. In a given tree, if you remove its root node then it becomes a forest.
13) **Directed Tree:** It is an acyclic diagraph which has one node called its root whose in degree is 0 while other nodes have in degree 1.

Applications of Tree:

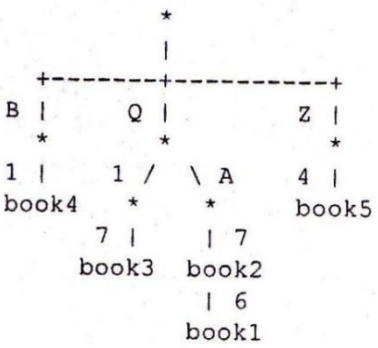
1. Trees can hold objects that are sorted by their keys. The nodes are ordered so that all keys in a node's left sub tree are less than the key of the object at the node, and all keys in a node's right sub tree are greater than the key of the object at the node. Here is an example of a tree of records, where each record is stored with its integer key in a tree node:



2. Trees can hold objects that are located by keys that are sequences. For example, we might have some books with these Library of Congress catalog numbers:

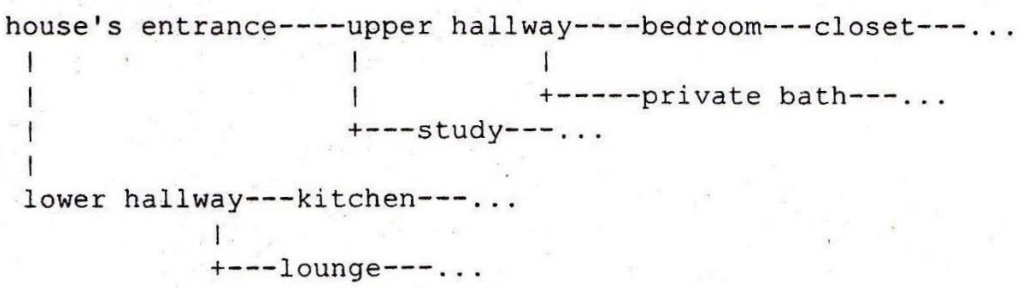
```
QA76  book1
QA7   book2
Q17   book3
B1    book4
Z4    book5
```

The books' keys are sequences, and the sequences label the branches of a tree that holds the books:



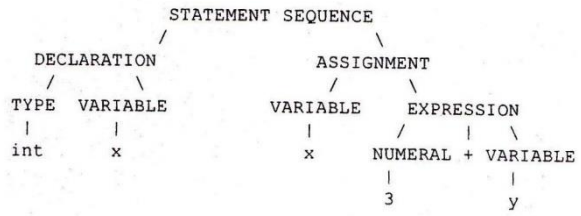
Books can be stored at nodes or leaves, and not all nodes hold a book (e.g., Q1). This tree is called a *spelling tree*, and it has the advantage that the insertion and retrieval time of an object is related only to the length of the key.

3. A tree can represent a structured object, such as a house that must be explored by a robot or a human player in an adventure game:



4. Trees are used to represent phrase structure of sentences, which is crucial to language processing programs. Here is the phrase structure tree (``parse tree'') for the Java statements.

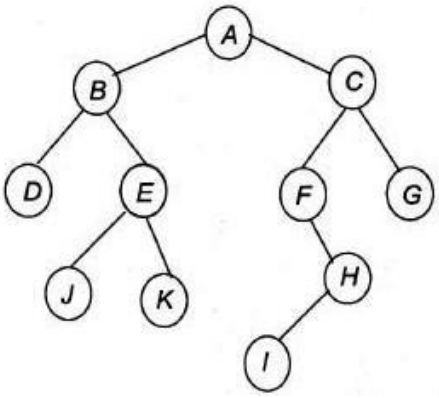
```
int x;
x = 3 + y;
```



5. An operating system maintains a disk's file system as a tree, where file folders act as tree nodes.

Introduction to binary tree:

- One of the most important of all tree structures is the binary tree.
- In a binary tree, each node has no more than two child nodes, each of which may be a leaf node.
- In a binary tree each node has two sub trees known as the left sub tree and the right sub tree.
- A binary tree T is a finite set of nodes such that
  i. T is empty (called empty binary tree) or
  ii. T contains a specially designated node called the root of T, and the remaining nodes of T form two disjoint binary trees T1 and T2 which are called left sub tree and the right sub tree respectively.



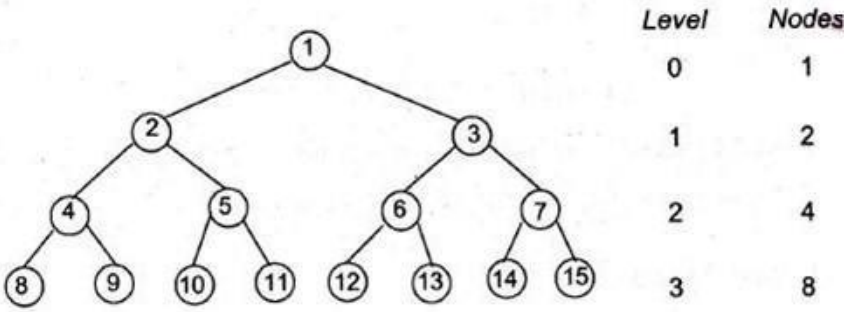
- A tree can never be empty but binary tree may be empty. In case of a binary tree, a node may have at most two children, whereas in case of a tree, a node may have any number of children.

Types of Binary trees:

- There are two various types of binary tree
  1. Full binary tree.
  2. Complete binary tree.

1. Full binary tree:

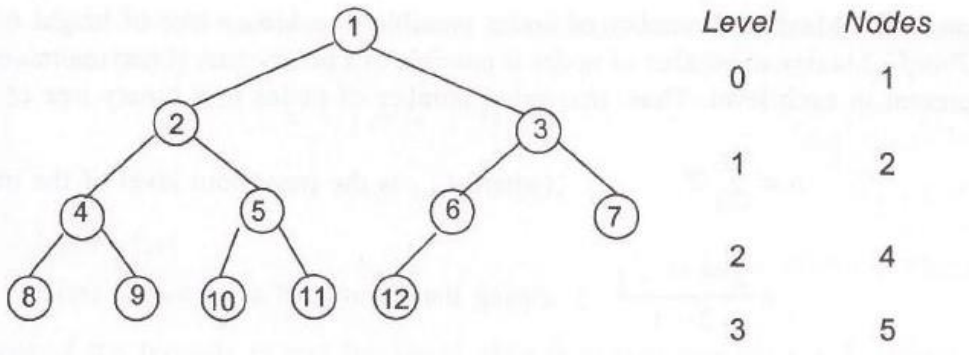
- A binary tree is a full binary tree, if it contains maximum possible number of nodes in all level. Figure shows such a tree with height 4.



(a) A full binary tree of height 4

2. Complete binary tree:

- A binary tree is said to be a complete binary tree if all its level, except possibly the last level, have the maximum number of possible nodes, and all the nodes at the last level appear as far as possible.



A complete binary tree of height 4

Representation of a binary tree:

Sometime it is required to maintain binary tree in the computer memory. There are two techniques used to maintain binary tree in the memory.

1. Linear (Sequential) representation.
2. Linked list representation.

Linear (Sequential) representation of a binary tree:

- This is easiest method to implement.
- This type of representation is static it means once the memory is allocated, the size of the tree will be restricted as the memory permits.
- In this representation, the nodes are stored level by level, starting from the zero level where only root node is present.
- Root node is stored in the first memory location.
- A linear storage representation can be done by **one dimensional array**. The **size** of one dimensional array can be calculated by using following formula.

Formula:

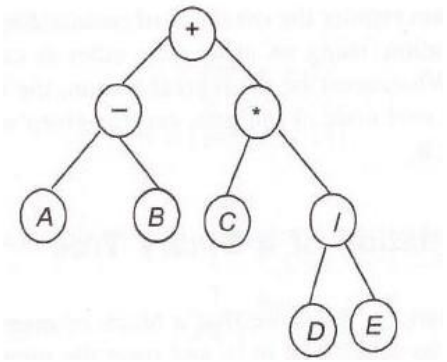
Array size = $2^{d+1} - 1$

Where d is the depth of the tree. Depth of tree is the maximum level of any node in the tree.

All the elements of binary tree can be stored into one-dimensional array by following **rules**.

1. Root node is stored at position 1st location of array.
2. If the node is stored at position N then its left child node is stored at position 2*N and its right child node is at 2*N + 1.

Example: Consider following binary tree.



A binary tree

Here, root "+" is at level 0, nodes "*" and "-" are at level 1, nodes "A","B","C","/" are at level 2 while nodes "D" and "E" are stored at level 3. So depth of this tree is 3. So size of array is calculated as below:

Array size = $2^{d+1} - 1$ $= 2^{3+1} - 1$ $= 2^4 - 1$ $= 16 - 1$ $= 15$

All the elements are stored in array as follow:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+	–	*	A	B	C	/	.	.	.	.	.	.	D	E

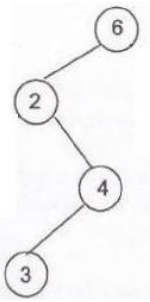
A sequential representation of the binary tree

- **Advantages:**
  1. It is simple and easy to implement.
  2. The main advantage of this method is its simplicity.
  3. It is possible to find parent node for any child node. (Position of a parent node = position of child node / 2)
  4. Any node can be accessed from any other node by calculating the index and this is efficient from execution.
  5. Data are stored only without any pointers to their successor or ancestor.
  6. Programming language, where dynamic memory allocation is not possible, array representation is the best option to store a tree.

7. This is an ideal representation for complete binary tree where space is not wasted.

- **Disadvantages:**

1. Other than full binary trees, majority of the array entries may be empty. For example, skewed binary tree where more than 50% of space is unutilized.



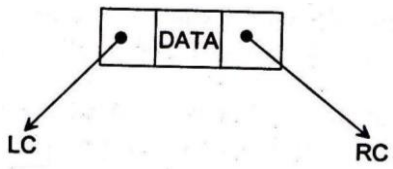
A skew binary tree

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Info	6	2	.	.	4	.	.	.	.	3	.	.	.	.	.

2. It allows only static representation. It is not possible to enhance the tree structure if the array size is limited.
3. Inserting a new node to it or deleting a node from it requires considerable data movement of array which is time consuming process.

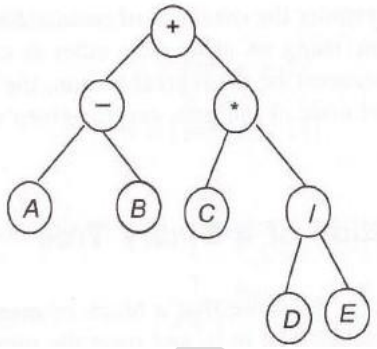
Linked representation of binary trees:

- In a binary tree each node may have maximum 2 child nodes.
- In linked representation, a node has two pointers fields and one information field. Each of pointer field contains the address of left or right child node. An information field contains the actual data about node. When node has no child the corresponding pointer fields are null.
- In linked representation the structures of a node shown in figure.

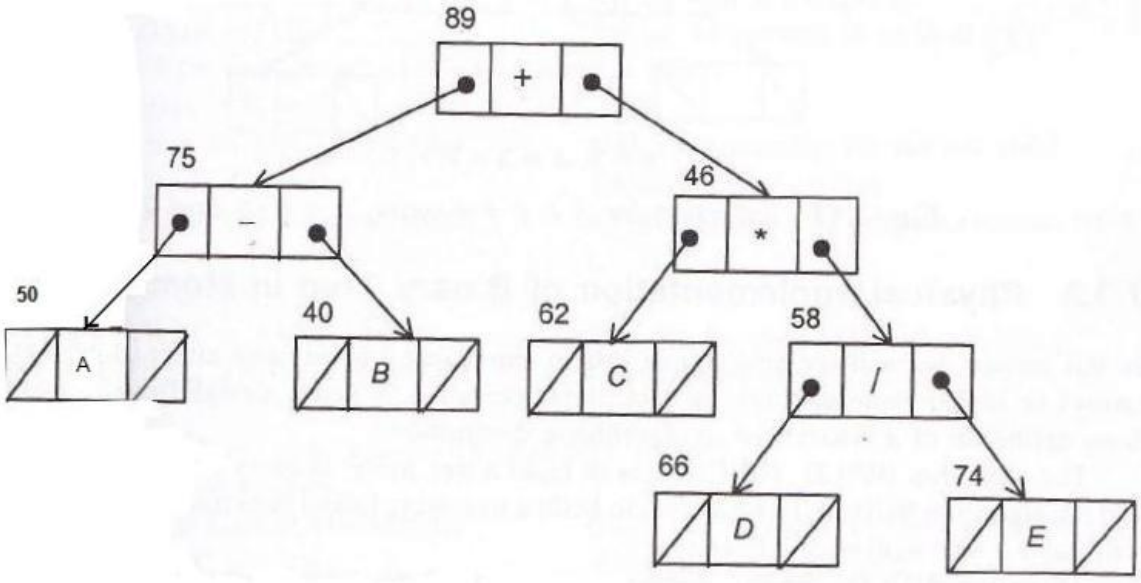


Structure of a node

- Here, LC and RC are two link fields to store the address of left child and right child of a node; DATA is the information content of the node.
- With this representation, if one knows the address of the root node then from it any other node can be accessed.



A binary tree



Logical view of the linked representation of a binary tree

- **Advantages:**
  1. It allows the dynamic memory allocation. So size of the tree can be changed whenever necessary.
  2. It is more efficient way for insertion and deletion operation.
  3. Insertions / deletions do not require data movement. It needs only rearrangement of few pointers. So it is faster than linear representations.

- **Disadvantages:**
  1. This representation requires extra memory to maintain the pointers. Some pointers with null values are also stored in memory. So more memory requires compare to linear representation.
  2. Wastage of memory space in null pointers.
  3. It is difficult to find parent node from the given child node.
  4. It is difficult to implement in old languages (such as FORTRAN, BASIC) that do not support dynamic storage techniques.

Traversal of Binary Tree:

- It is a most common operation performed on tree data structure.
- It is a procedure by which each node in the tree is processed exactly once in a systematic manner.
- All nodes are connected via edges (links) therefore we always start from the root (head) node. That is, we cannot randomly access a node in a tree.
- There are three ways which we use to traverse a tree −
  1) In-order Traversal
  2) Pre-order Traversal
  3) Post-order Traversal
- Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

Inorder traversal of a binary tree:

- With this traversal, before visiting the root node, left sub-tree of the root node is to be visited then root node and after the visit of the root node right sub-tree of the root node will be visited. Visit of both sub-trees again will be in the same fashion. Such traversal can be stated as below:
  a) Traverse the left sub-tree of the root node R is inorder.
  b) Visit the root node R.
  c) Traverse the right sub-tree of the root node R is inorder.

- Out of these steps, step 1 and 3 are defined recursively. Following is the algorithm INORDER to implement the above definition.
- **Algorithm: INORDER(ROOT)**
  - **Input:** ROOT is the pointer to the root node of the binary tree.
  - **Output:** Visiting of the all nodes in inorder fashion.
  - **Data Structure:** Linked structure of binary tree.
  - **Steps:**

1. ptr = ROOT // Start from ROOT
2. If (ptr != NULL) then // If it is not an empty node
   a. INORDER(ptr.LC) // Traverse the left sub-tree
   b. VISIT(ptr) // Visit the node
   c. INORDER(ptr.RC) // Traverse the right sub-tree
3. End if
4. Stop

Preorder traversal of a binary tree:

- In this traversal, root is visited first then left sub-tree in pre-order fashion and then right sub-tree in pre-order fashion. It can be defined as below:
  a) Visit the root node R.
  b) Traverse the left sub-tree of R in preorder.
  c) Traverse the right sub-tree of R in preorder.
- The algorithm is presented as below:
- **Algorithm: PREORDER(ROOT)**
  - **Input:** ROOT is the pointer root node of the binary tree.
  - **Output:** Visiting of all the nodes in preorder fashion.
  - **Data structure:** Linked structure of binary tree.
  - **Steps:**

1. ptr = ROOT // Start from ROOT
2. If (ptr != NULL) then // If it is not an empty node
   a. VISIT(ptr) // Visit the node
   b. PREORDER(ptr.LC) // Traverse the left sub-tree
   c. PREORDER(ptr.RC) // Traverse the right sub-tree
3. End if

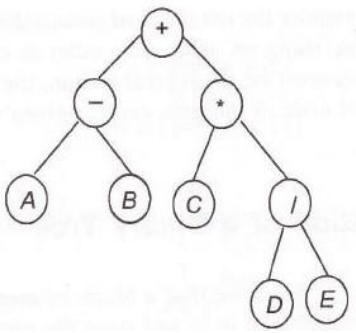
4. Stop

Postorder traversal of a binary tree:

- Here, first visit the left sub-tree and then right sub-tree and at last is the root. Definition for this is as stated below:
  a) Traverse the left sub-tree of the root R in post order.
  b) Traverse the right sub-tree of the root R in post order.
  c) Visit the root node R.
- **Algorithm: POSTORDER(ROOT)**
  - **Input:** ROOT is the pointer root node of the binary tree.
  - **Output:** Visiting of all the nodes in postorder fashion.
  - **Data structure:** Linked structure of binary tree.
  - **Steps:**

1. ptr = ROOT // Start from ROOT
2. If (ptr != NULL) then // If it is not an empty node
   a. POSTORDER(ptr.LC) // Traverse the left sub-tree
   b. POSTORDER(ptr.RC) // Traverse the right sub-tree
   c. VISIT(ptr) // Visit the node
3. End if
4. Stop

Example: Write a preorder, inorder and postorder traversal of a given binary tree.



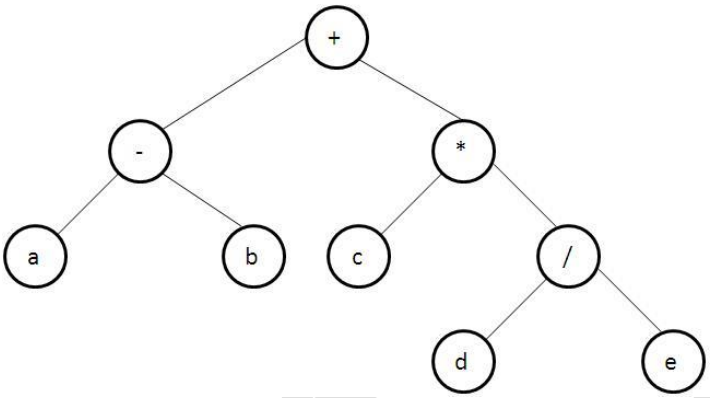
Pre order: + - A B * C / D E

In order: A – B + C * D / E

Post order: A B – C D E / * +

Example: Draw a binary tree for following expression and write down pre-order, post order and inorder traversal of a binary tree.

a-b+c*d/e



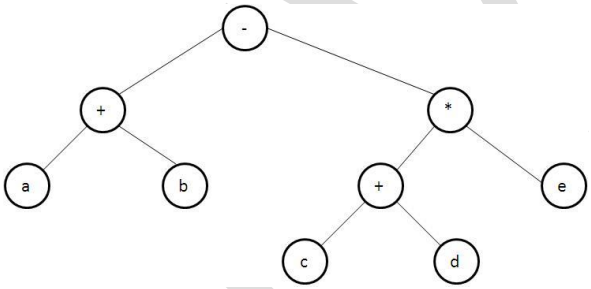
Pre order: + - a b * c / d e

In order: a – b + c * d / e

Post order: a b – c d e / * +

Example: Draw a binary tree for following expression and write down pre-order, post order and inorder traversal of a binary tree.

(a+b)-(c+d)*e



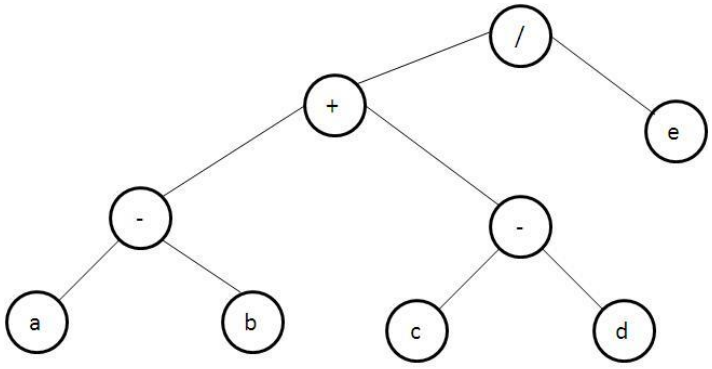
Pre order: - + a b * + c d e

In order: a + b - c + d * e

Post order: a b + c d + e * -

Example: Draw a binary tree for following expression and write down pre-order, post order and inorder traversal of a binary tree.

(a-b)*(c-d)/e



Pre order: / + - a b – c d e

In order: a - b + c - d / e

Post order: a b – c d - + e /

Formation of binary tree from its traversals:

- Sometime it is required to construct a binary tree if its traversals are known. From a single traversal, it is not possible to construct unique binary tree, however, if two traversals are known then corresponding tree can be drawn uniquely.
- Basic principles for formation are stated as below:
  1. If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node.
  2. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified.
  3. Same technique can be applied repeatedly to form sub-trees.

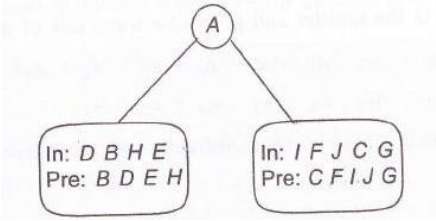
Example – 1: Suppose inorder and preorder traversals of a binary tree are as below:

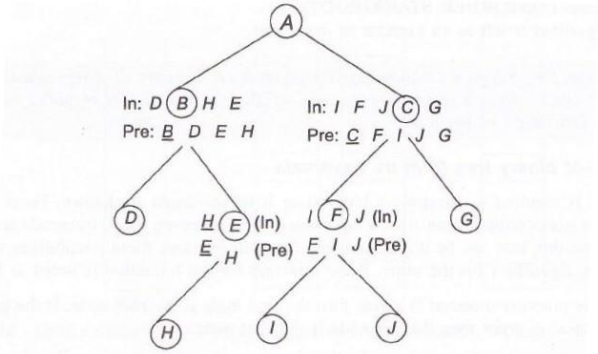
Inorder	D	B	H	E	A	I	F	J	C	G
preorder	A	B	D	E	H	C	F	I	J	G

We have to construct the binary tree.

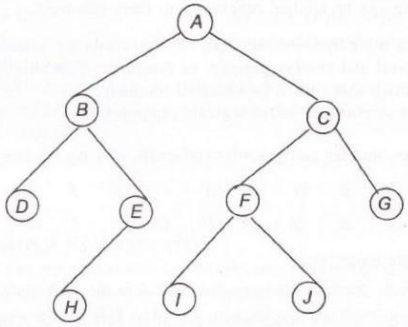
1. From the preorder traversal, it is evident that A is the root node.
2. In inorder traversal, all the nodes which are at the left side of a belong to the left sub-tree and those are at right side of A belong to the right sub-tree.

3. Now the problem reduces to form sub-trees and the same procedure can be applied repeatedly.





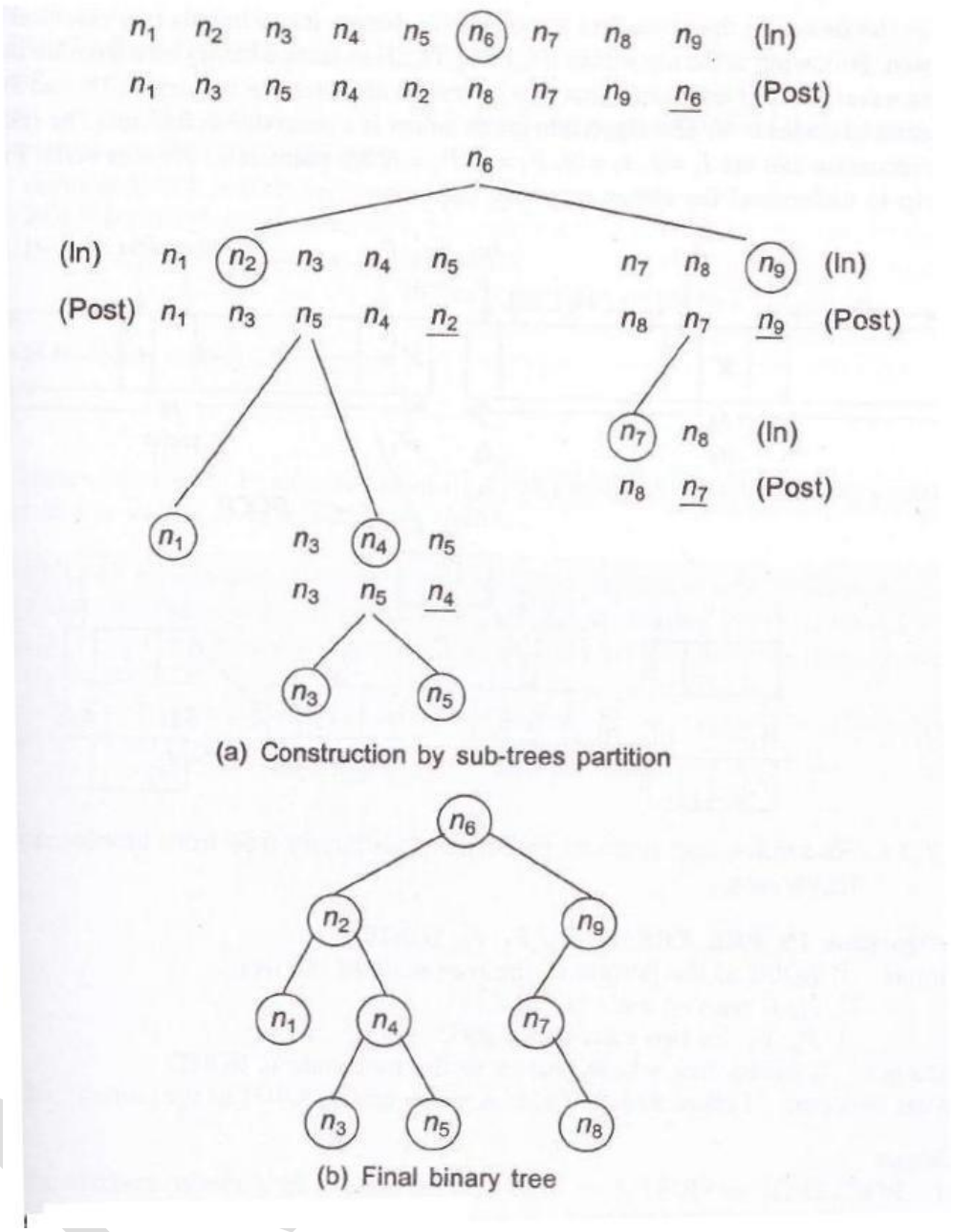
(b) Repeated application of sub-trees formation



(c) Final binary tree from its inorder and preorder traversals

Example – 2: Suppose inorder and postorder traversals of a binary tree are as below:

In order	n1	n2	n3	n4	n5	n6	n7	n8	n9
Post order	n1	n3	n5	n4	n2	n8	n7	n9	n6

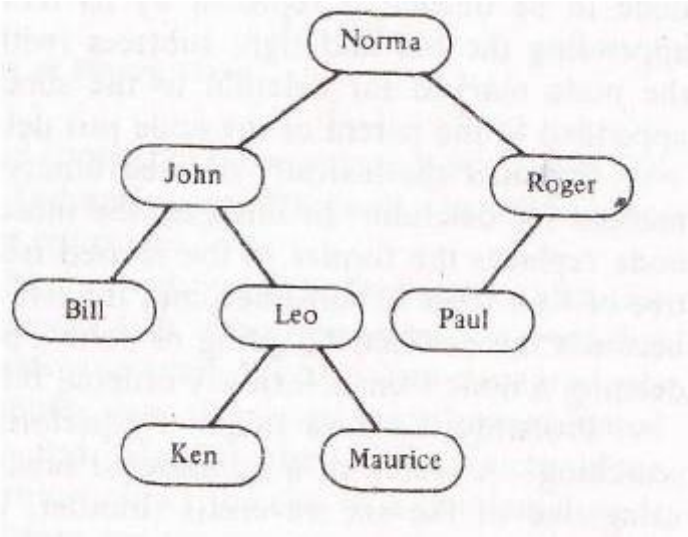


Lexically ordered binary tree:

- A binary tree T is known as lexically ordered binary tree if each node N of T satisfies the following property:
  - The value at N is greater than value in the right sub tree of N and is less than every value in the left sub tree of N.

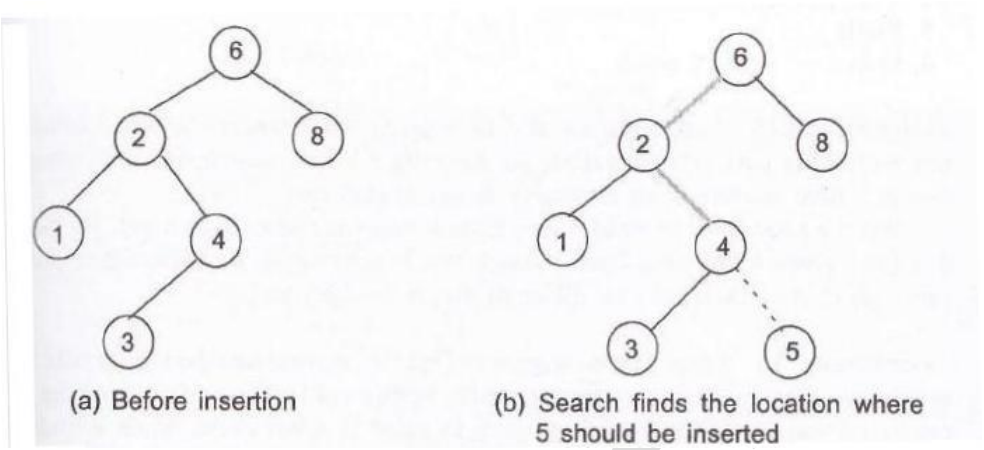
Insertion in a lexically ordered binary tree:

- The creation of a binary tree could be based on the information associated with each node. This ordering could be numerical or it could be a list of names to be kept in lexicographical order as shown in figure.



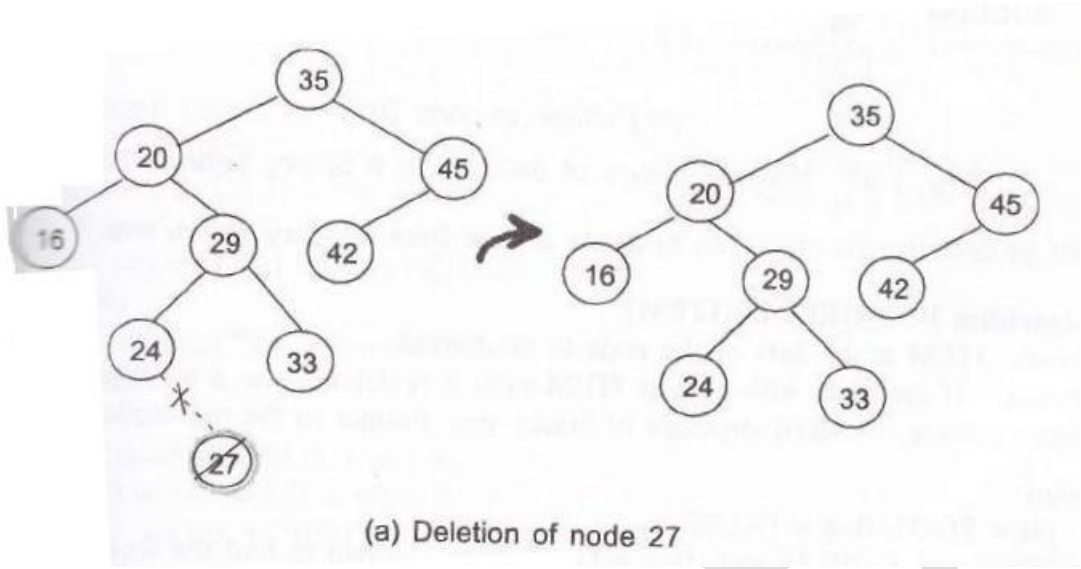
Lexically ordered tree

- The left sub-tree of a tree is to contain nodes whose associated names are lexically less than the name associated with the root node of the tree (or sub-tree).
- Similarly, the right sub-tree of the tree is to contain nodes whose associated names are lexically greater than the name associated with the root node of the tree(or sub-tree).
- There are two cases for insertion
    1. Insertion into an empty tree results in appending new nodes or root of the tree.
    2. Inserting a new node into non empty tree.
- To insert a node with data, say ITEM, into a tree, the tree is to be searched starting from the root node. If item is found, do nothing, otherwise ITEM is to be inserted at the dead end where search halts.
- Figure shows the insertion of 5 into a binary tree. Here, search proceeds starting from root node as 6 – 2 – 4 then halts when it finds that right child is null (dead end).
- This simply that if 5 occurs, then it should occur in the right part of the node 4. So, 5 should be inserted as the right child of 4.



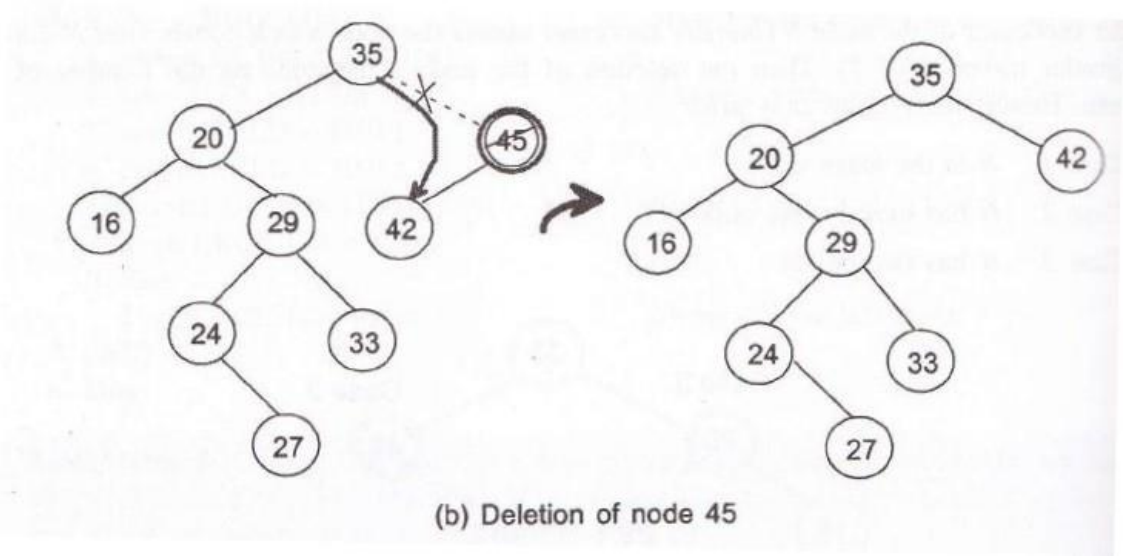
Deletion in a lexically ordered binary tree:

- Suppose T is a binary search tree, and ITEM is the information given which has to be deleted from T if it exists in the tree. Suppose N be the node which contains the information ITEM. Let us assume PARENT(N) denotes the parent node of N and SUCC(N) denotes the inorder successor of the node N (inorder successor means the node which comes after N during the inorder traversal of T). Then the deletion of the node N depends on the number of its children. Hence three cases may arise:
  1) Case 1**:** N is the leaf node.
  2) Case 2: N has exactly one child.
  3) Case 3: N has two child.
- **Case - 1:** N is deleted from T by simply setting the pointer of N in the parent node PARENT (N) by NULL value. See figure (a).



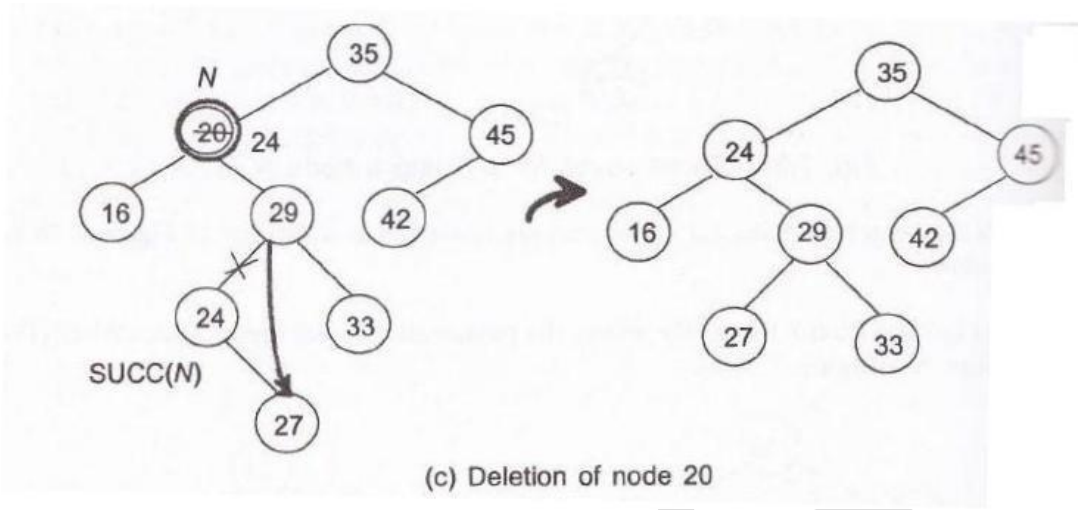
(a) Deletion of node 27

- **Case - 2:** N is deleted from T by simply replacing the pointer of N in PARENT (N) by the pointer of only child of N. See figure (b).



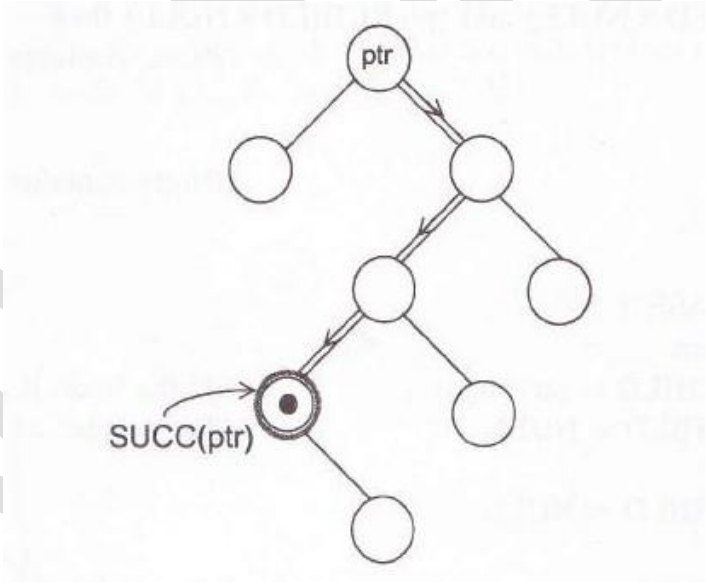
(b) Deletion of node 45

- **Case - 3:** N is deleted from T by first deleting SUCC (N) from T and then replaces the data content in node N by the data content in node SUCC (N). See figure (c).



(c) Deletion of node 20

- Assume that the function SUCC(ptr) which returns pointer to the inorder successor of the node ptr. It can be verified that inorder successor of ptr always occurs in the right sub-tree of ptr, and inorder successor of ptr does not have left child.



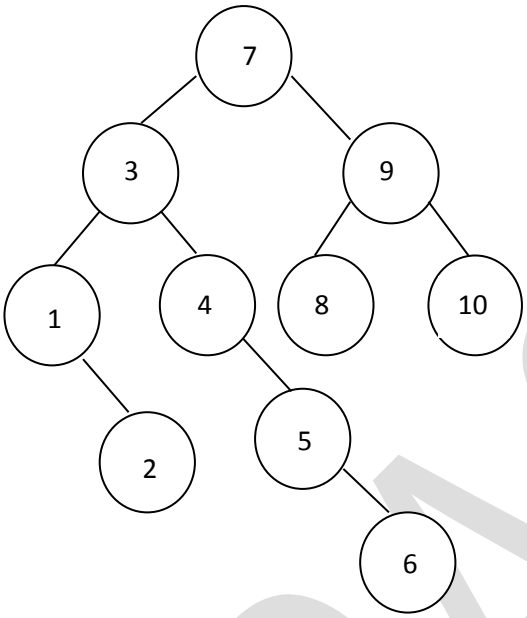
In order successor of a binary tree

Binary Search Tree:

A binary search tree is a binary tree which is either empty or satisfies the following rules:

a) The value of the key in the left child or left subtree is less than the value of the root.
b) The value of the key in the right child or right subtree is more than or equal to the value of the tree.

c) All the sub-trees of the left and right children observe the two rules.



SYBCA SEM - IV
US04CBCA25 (DATA STRUCTURES – II)
Unit – II Linked List

LINKED LISTS

An array is a data structure where elements are stored in consecutive memory locations. In order to occupy the adjacent space, a block of memory that is required for the array should be allocated beforehand. Once memory is allocated it cannot be extended any more. This is why array is known as a static data structure.

In contrast to this, linked list is called dynamic data structure where the amount of memory required can be varied during its use. In linked list, the adjacency between the elements is maintained by means of links or pointers.

A link or pointer actually is the address (memory location) of the subsequent element. Thus, in a linked list, data (actual content} and link (to point to the next data) both are required to be maintained.

An element in a linked list is specially termed as node, which can be viewed as shown in Figure 3.1. A node consists of two fields:

- DATA (to store the actual information) and
- LINK (to point to the next node).

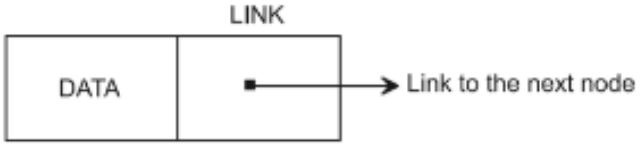


Figure 3.1 Node: an element in a linked list.

DEFINITION

A linked list is an ordered collection of finite, homogeneous data elements celled nodes where the linear order is maintained by means of links or pointers.

Depending on the requirements the pointers are maintained, and accordingly linked list can be classified into three major groups:

- single linked list,
- circular linked list, and
- double linked list.

SINGLE LINKED LIST

In a single linked list each node contains only one link which points the subsequent node in the list. Figure 3.2 shows a linked list with six nodes

Here, N1, N2. . . .. N6 are the constituent nodes in the list. HEADER is an empty node (having data element NULL) and only used to store a pointer to the first node N1.

Thus, if one knows the address of the HEADER node from the link field of this node, the next node can be traced, and so on.

This means that starting from the first node one can reach to the last node whose link field does not contain any address but has a null value.

Note that in a single linked list one can move from left to right only; this is why a single linked list is also called **one way list.**

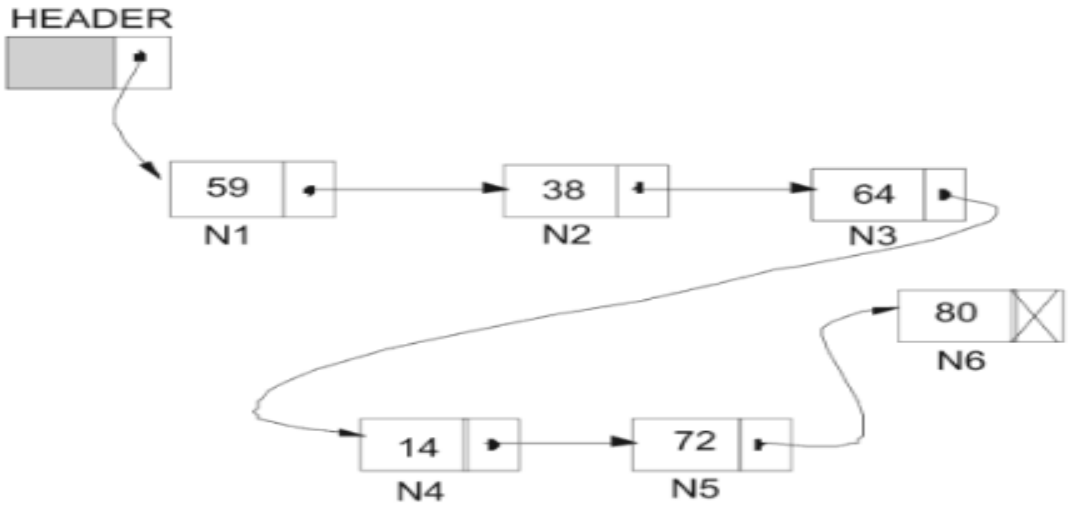


Figure 3.2 A single linked list with six nodes.

CIRCULAR LINKED LIST

In our previous discussion, we have noticed that in single linked list, the link field of the last node is null (hereafter a single linked list may be read as ordinary linked list), but a number of advantages can be gained if we utilize this link field to store the pointer of the header node.

A linked list where the last node points the header node is called circular linked list. Figure 3.3 shows a pictorial representation of a circular linked list.

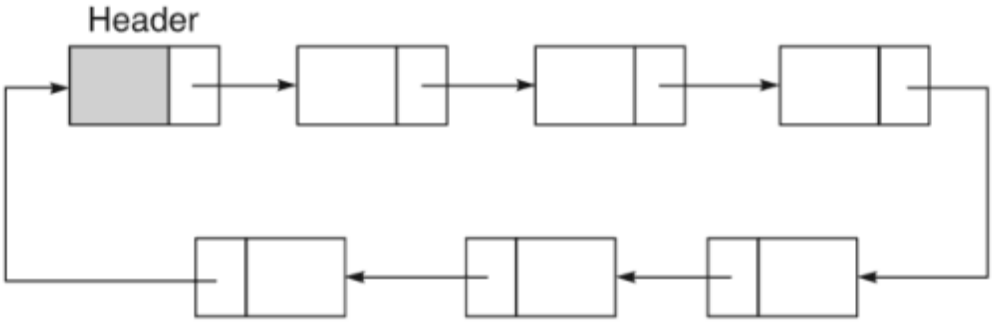


Figure 3.8 A circular linked list.

DOUBLE LINKED LISTS

In a single linked list one can move beginning from the header node to any node in one direction only (from left to right). This is why a single linked list is also termed a one-way list.

On the other hand, a double linked list is a two-way list because one can move in either direction, either from left to right or from right to left.

This is accomplished by maintaining two link fields instead of one as in a single linked list. A structure of a node for a double linked list is represented as in Figure 3.10.

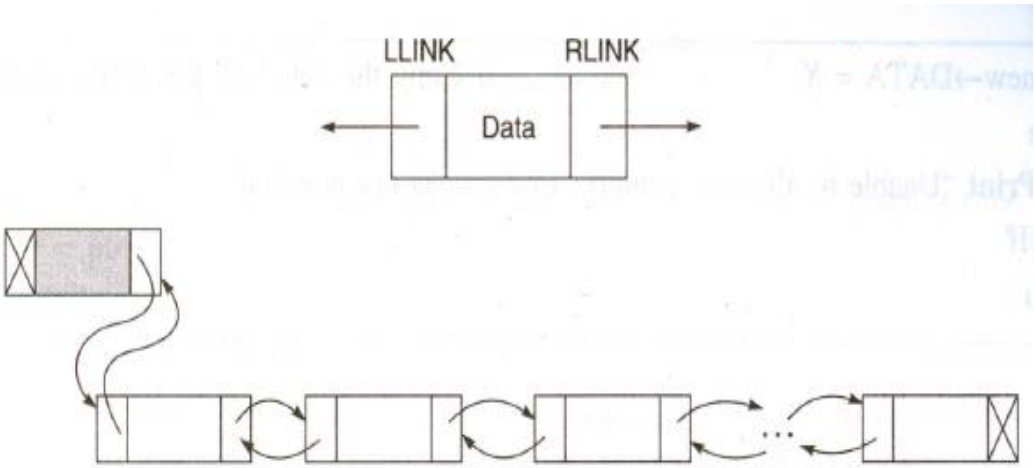


Figure 3.10 Structure of a node and a double linked list.

From the figure, it can be noticed that two fields, namely RLINK and LLINK, point to the nodes on the right side and left side of the node, respectively.

Thus, every node, except the header node and the last node, points to its immediate predecessor and immediate successor.

CIRCULAR DOUBLE LINKED LISTS

The advantage of both double linked list and circular linked list are incorporated into another type of list structure called circular double linked list and it is known to be the best of its kind.

Figure 3.13 shows a schematic presentation of a circular double linked list.

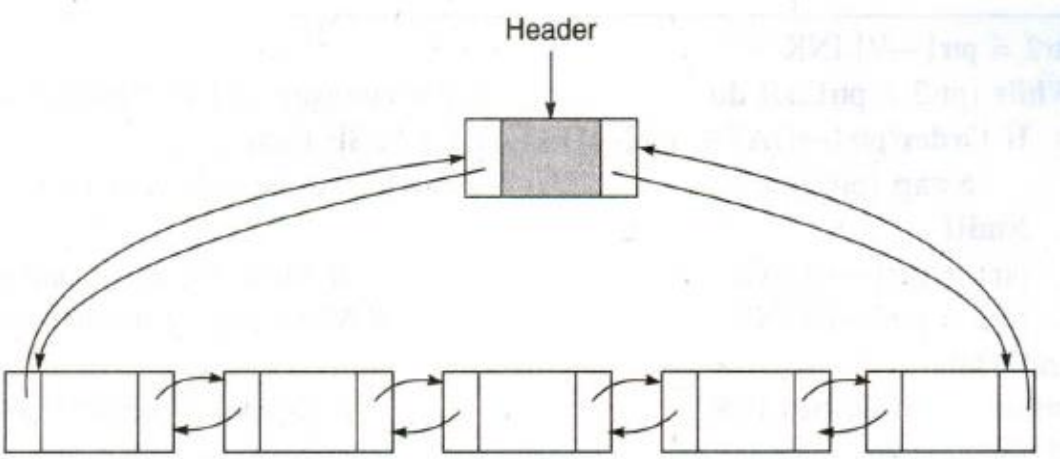


Figure 3.13 A circular double linked list.

Here note that the

- RLINK (right link) of the rightmost node and LLINK (left link) of the leftmost node contain the address of the header node;
- again the RLINK and LLINK of the header node contain the address of the rightmost node and the leftmost node, respectively.

An empty circular double linked list is represented as shown in figure 3.14. In case of an empty list, both LLINK and RLINK of the header node point to itself.

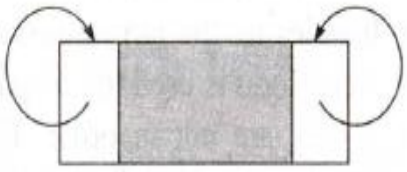


Figure 3.14 An empty circular double linked list.

*INSERTING A NODE INTO A SINGLE LINKED LIST

There are various positions where a node can be inserted:
1. Insert at front (as a first element)
2. Insert at end (as a last element)
3. Insert at any other position

Inserting of a node at the front of a single linked list

The algorithm InsertFront_SL is used to insert a node at the front of a single linked list. .

Algorithm InsertFront_SL

Input:	HEADER is the pointer to the header node and X is the data of the node to be inserted.
Output:	A single linked list with newly inserted node at the front of the list.
Data structures:	A single linked list whose address of the starting node is known from the HEADER.

Steps:

1. new = **GetNode**(NODE) // Get a memory block of type NODE and store its pointer in new
2. **If** (new = NULL) **then** // Memory manager returns NULL on searching the memory bank
3. **Print** "Memory underflow: No insertion"
4. Exit
5. **Else** // Memory is available and get a node from memory bank
6. new → LINK = HEADER → LINK // Change of pointer 1 as shown in Fig.3.5(a)
7. new → DATA = X // Copy the data X to newly availed node
8. HEADER → LINK = new // Change of pointer 2 as shown in Fig.3.5(a)
9. **EndIf**
10. **Stop**

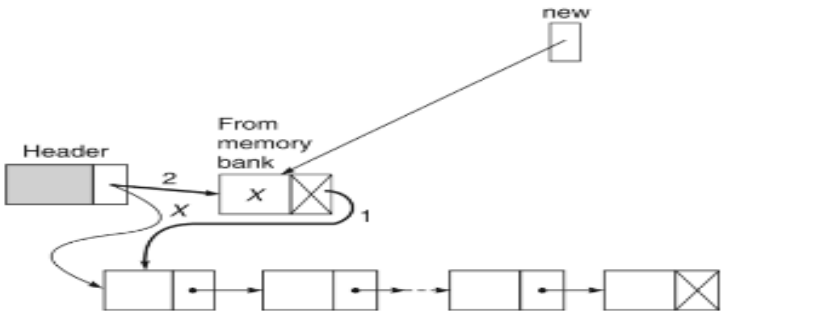


Figure 3.5(a) Inserting a node in the front of a single linked list.

Inserting a node at the end of a single linked list

The algorithm InsertEnd_SL is used to insert a node at the end of a single linked list.

Algorithm InsertEnd_SL

Input:	HEADER is the pointer to the header node and X is the data of the node to be inserted.
Output:	A single linked list with newly inserted node having data X at the end of the list.
Data structures:	A single linked list whose address of the starting node is known from the HEADER.

Steps:

```
1.  new = GetNode(NODE)          // Get a memory block of type NODE and
                                    return its pointer as new
2.  If (new = NULL) then         // Unable to allocate memory for a node
3.      Print "Memory is insufficient: Insertion is not possible"
4.      Exit                     // Quit the program
5.  Else                         // Move to the end of the given list and then insert
6.      ptr = HEADER             // Start from the HEADER node
7.      While ( ptr → LINK ≠ NULL ) do   // Move to the end
8.          ptr = ptr → LINK     // Change pointer to the next node
9.      EndWhile
10.     ptr → LINK = new         // Change the link field of last node:
                                    Pointer 1 as in Fig.3.5(b)
11.     new → DATA = X           // Copy the content X into the new node
12. EndIf
13. Stop
```

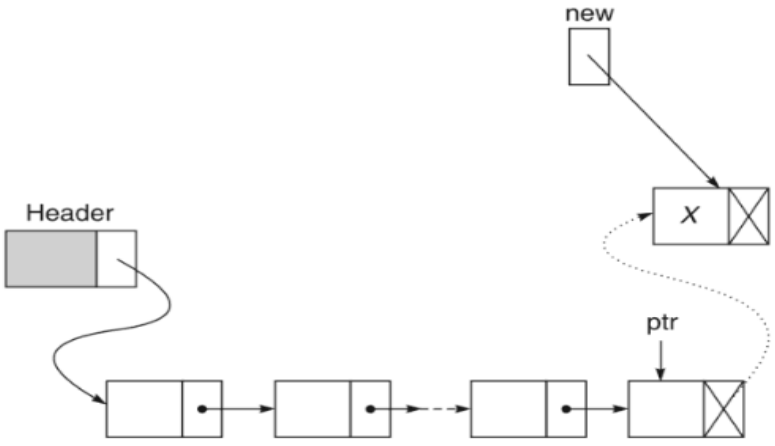


Figure 3.5(b) Inserting a node at the end of a single linked list.

Inserting a node into a single linked list at any position in the list

The algorithm InsertAny_SL is used to insert a node into a single linked list at any position in the list.

Algorithm InsertAny_SL

Input:	HEADER is the pointer to the header node, X is the data of the node to be inserted, KEY being the data of the key node after which the node has to be inserted.
Output:	A single linked list enriched with newly inserted node having data X after the node with data KEY.
Data structures:	A single linked list whose address of the starting node is known from the HEADER.

Steps:

```
1.  new = GetNode(NODE)                    // Get a memory block of type NODE and
                                           return its pointer as new
2.  If (new = NULL) then                   // Unable to allocate memory for a node
3.      Print "Memory is insufficient: Insertion is not possible"
4.      Exit                               // Quit the program
5.  Else                                   // Move to the end of the given list and then insert
6.      ptr = HEADER                       // Start from the HEADER node
7.      While (ptr→LINK ≠ NULL) and (ptr→LINK ≠ NULL) do  // Move to the node
                                           // having data as KEY or at the end if KEY is not in the list
8.          ptr = ptr → LINK
9.      EndWhile
10.     If (ptr → LINK = NULL ) then       // Search fails to find the key

11.         Print "KEY is not available in the list"
12.         Exit
13.     Else
14.         new → LINK = ptr → LINK        // Change of pointer 1 as shown
                                           in Fig.3.5(c)
15.         new → DATA = X                 // Copy the content X into the new node
16.         ptr → LINK = new               // Change the pointer 2 as in Fig.3.5(c)
17.     EndIf
18. EndIf
19. Stop
```

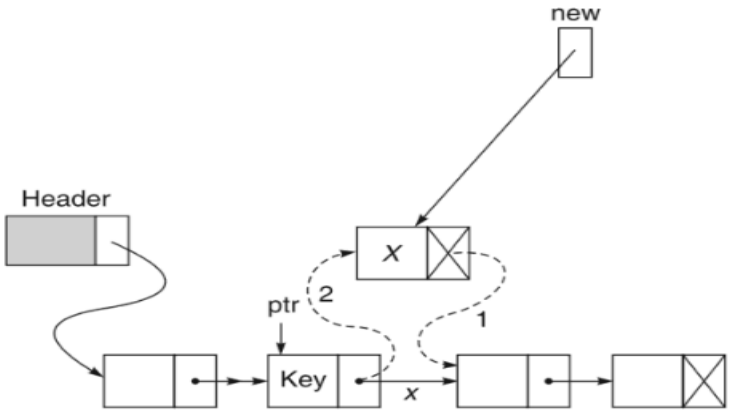


Figure 3.5(c) Inserting a node at any position in a single linked list.

*DELETION OF A NODE FROM A SINGLE LINKED LIST

Like insertions, there are also three cases of deletion:

1. Deleting from the front of the list
2. Deleting from the end of the list
3. Deleting from any position in the list.

Deleting the node at the front of a single linked list

The algorithm DeleteFront_SL is used to delete the node at the front of a single linked list. Such a delete operation is explained in figure 3.6(a).

Algorithm DeleteFront_SL

Input:	HEADER is the pointer to the header node.
Output:	A single linked list after eliminating the node at the front of the list.
Data structures:	A single linked list whose address of the starting node is known from the HEADER.

Steps:

1. ptr = HEADER → LINK // Pointer to the first node
2. **If** (ptr = NULL) **then** // If the list is empty
3. **Print** "The list is empty: No deletion"
4. Exit // Quit the program
5. **Else** // The list is not empty
6. ptr1 = ptr → LINK // ptr1 is the pointer to the second node, if any
7. HEADER → LINK = ptr1 // Next node becomes the first node as in Fig. 3.6(a)
8. **ReturnNode** (ptr) // Deleted node is freed to the memory bank for future use
9. **EndIf**
10. **Stop**

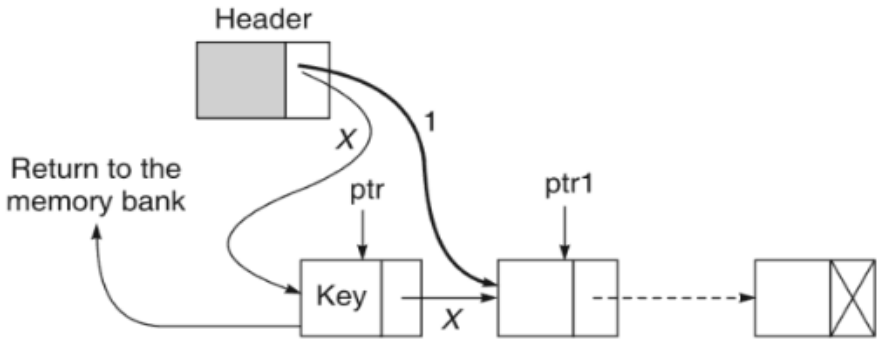


Figure 3.6(a) Deleting the node at the front of a single linked list.

Deleting the node at the end of a single linked list

The algorithm DeleteEnd_SL is used to delete the node at the end of a single linked list. This is shown in figure 3.6(b).

Algorithm DeleteEnd_SL

Input:	HEADER is the pointer to the header node.
Output:	A single linked list after eliminating the node at the end of the list.
Data structures:	A single linked list whose address of the starting node is known from the HEADER.

Steps:

1. ptr = HEADER // Move from the header node
2. **If** (ptr → LINK = NULL) **then**
3. **Print** "The list is empty: No deletion possible"
4. Exit // Quit the program
5. **Else**
6. **While** (ptr → LINK ≠ NULL) **do** // Go to the last node
7. ptr1 = ptr // To store the previous pointer
8. ptr = ptr → LINK // Move to the next
9. **EndWhile**
10. ptr1 → LINK = NULL // Last but one node becomes the last node as in Fig. 3.6(b)
11. **ReturnNode** (ptr) // Deleted node is returned to the memory bank for future use
12. **EndIf**
13. **Stop**

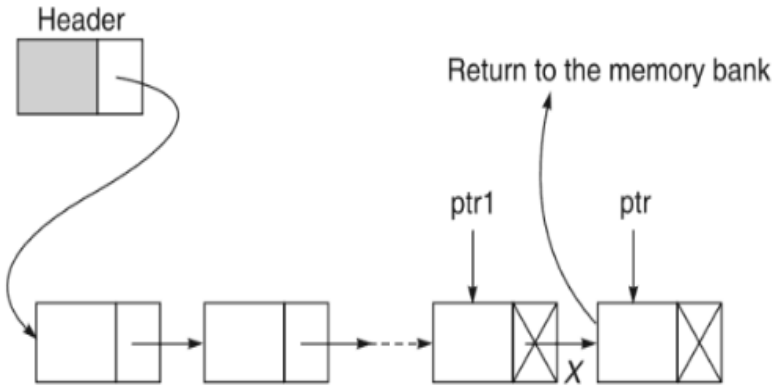


Figure 3.6(b) Deleting the node at the end of a single linked list.

Deleting the node from any position of a single linked list

The algorithm DeleteAny_SL is used to delete a node from any position in a single linked list. This is illustrated in figure 3.6(c).

Algorithm DeleteAny_SL

Input:	HEADER is the pointer to the header node, KEY is the data content of the node to be deleted
Output:	A single linked list except the node with data content as KEY.
Data structures:	A single linked list whose address of the starting node is known from the HEADER.

Steps:

```
1.  ptr1 = HEADER                                   // Start from the header node
2.  ptr = ptr1 → LINK                               // This points to the first node, if any
3.  While ( ptr ≠ NULL ) do
4.      If ( ptr → DATA ≠ KEY ) then                // If found the key
5.          ptr1 = ptr                  // Keep a track of the pointer of the previous node
6.          ptr = ptr → LINK                        // Move to the next
7.      Else
8.          ptr1 → LINK = ptr → LINK    // Link field of the predecessor is to point the
                                        // successor of node under deletion, Fig. 3.6(c)
9.          ReturnNode (ptr)    // Return the deleted node is returned to the memory bank
10.         Exit                                    // Exit the program
11.     EndIf
12. EndWhile
13. If ( ptr = NULL ) then              // When the desired node is not available in the list
14.     Print "Node with KEY does not exist: No deletion"
15. EndIf
16. Stop
```

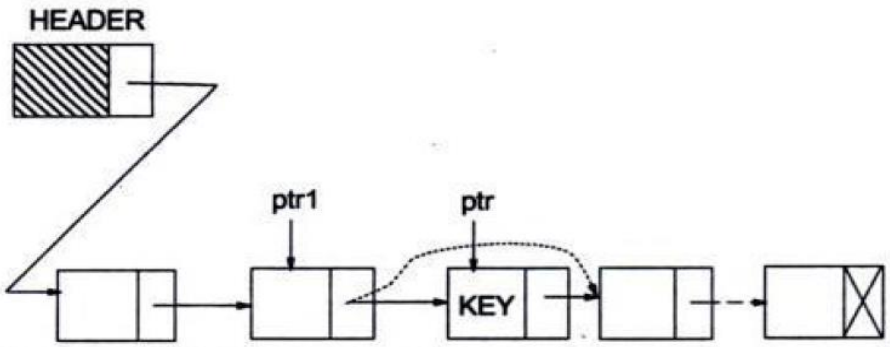


Fig. 3.6(c) Deletion of a node at any position in a single linked list.

Unit – 3

Sorting: It is the process to arrange the data in ascending or descending order.

Categories of Sorting:

Sorting can be classified into two broad categories:

1. Internal sorting and
2. External sorting

Internal sort: When a set of data is small enough such that entire sorting can be performed in a computer's internal storage (primary memory) then the sorting is called internal sort.

External sort: Sorting of a large set of data, which is stored in low speed computer's external memory (such as hard disk, magnetic tape, etc) is called external sort. It involves large amount of data transfer between external memory (low speed) and main memory (high speed).

Sorting Technique: A technique that arranges the data in ascending or descending order.

Types of Sorting Techniques:

The sorting techniques are:

1. Selection sort
2. Bubble sort
3. Insertion sort
4. Shell sort
5. Quick sort
6. Merge Sort

Bubble Sort

An algorithm to sort the data into ascending order using Bubble sort.

A	:	Vector (Array)
N	:	Number of elements in a vector
PASS	:	Pass counter
LAST	:	Position of last unsorted element
I	:	Index (subscript) used for vector elements
EXCHS	:	Used to count number of exchanges done in any pass

Step -1 : [Initialize **]**

LAST ⟵ N

Step -2 : [Loop on pass index **]**

Repeat through step – 5 for PASS = 1, 2, 3, ... N - 1

Step -3 : [Initialize exchange variable EXCHS **]**

EXCHS ⟵ 0

Step -4 : [Perform pair wise comparison **]**

Repeat for I = 1, 2, ... LAST – 1

IF (A[I] > A[I+1])

THEN

A[I] ⟷ A[I+1]

EXCHS ⟵ EXCHS + 1

Step -5 : [Exchange made or not **]**

IF (EXCHS = 0)

THEN

Return (Mission accomplish, return early)

ELSE

LAST ⟵ LAST – 1 (Reduce size of unsorted array)

Step -6 : [Finished **]**

Return

Bubble sort dry run

Given array: 5, 1, 4, 2, 8

Pass-1

5	1	1	1
1	5	4	4
4	4	5	2
2	2	2	5
8	8	8	8

Pass-2

1	1
4	2
2	4
5	5
8	8

Pass-3

1
2
4
5
8

Advantages of Bubble sort

1. Easy to understand.
2. Easy to implement.
3. Better algorithm for almost sorted data.

Disadvantages of Bubble sort

Large amount of data movement required if data is in random order or reverse sorted order.

Quick Sort

An algorithm to sort the data into ascending order using Quick sort.

A	:	Vector (Array)
N	:	Number of elements in a vector
LB	:	Lower bounds
UB	:	Upper bounds
I and J	:	Index (subscript) used for vector elements
KEY	:	Key value
FLAG	:	Boolean variable

QUICK_SORT(A, LB, UB)

Step -1 : [Initialize **]**

FLAG ⟵ true

Step -2 : [Perform sort **]**

If (LB < UB)

Then

I ⟵ LB

J ⟵ UB + 1

KEY ⟵ A[LB]

Repeat while FLAG

I ⟵ I + 1

Repeat while A [I] < KEY

I ⟵ I + 1

J ⟵ J – 1

Repeat while A[J] > KEY

J ⟵ J – 1

If (I < J)

Then

A [I] ⟷ A [J]

Else

FLAG ⟵ false

A [LB] ⟷ A [J]

Call QUICK_SORT(A, LB, J – 1)

Call QUICK_SORT(A, J + 1, UB)

Step -3 : [Finished **]**

Return

Merge Sort

Algorithm: SIMPLE_MERGE (K, FIRST, SECOND, THIRD)

This algorithm sorts elements into ascending order.

K	Vector (Array) contains two ordered arrays
TEMP	Temporary vector
FIRST	Position of first element of First vector in K vector
SECOND	Position of first element of Second vector in K vector
THIRD	Position of last element of Second vector in K vector
I	Index (subscript) used for first vector elements
J	Index (subscript) used for second vector elements
L	Index (subscript) used for TEMP vector elements

1.	[Initialize] I←FIRST J ←SECOND L ←0

2	[Compare corresponding elements and output the smallest]
	Repeat while I < SECOND and J ≤ THIRD
	If K[I] ≤ K [J] then
	L ← L + 1
	TEMP [L] ← K[I]
	I←I + 1
	Else
	L ← L + 1
	TEMP [L] ← K[J]
	J ← J + 1

3.	[Copy the remaining unprocessed elements in output area]
	If I≥ SECOND then
	Repeat while J ≤ THIRD
	L ← L + 1
	TEMP [L] ← K[J]
	J ← J + 1
	Else
	Repeat while I< SECOND
	L ← L + 1
	TEMP [L] ← K[I]
	I←I + 1

4.	[Copy elements of temporary vector into original area] Repeat for I = 1, 2, ……., L K [FIRST – 1 + I] ←TEMP [I]
5.	[Finished] Return

Merge sort dry run

Given array: 11, 23, 42, 9, 25

Array-1	Array-2	Temp array
11 →	~~9~~	9
23	25	
42		

~~11~~ →	25	9
23		11
42		

~~23~~ →	25	9
42		11
		23

42 →	~~25~~	9
		11
		23
		25
		42

Advantages of Merge sort

Easy to merge already sorted lists into a new sorted list with merge sort.

Disadvantages of Merge sort

Merge sort requires extra storage space for temporary vector.

Applications of sorting:

Sorting algorithms are essential in a broad variety of applications.

1. Commercial computing
2. Search for information
3. Operations research
4. Event-driven simulation
5. Numerical computations
6. String processing algorithms
7. Records with multiple keys
8. Display Google Page Rank results
9. Find the median
10. Frequency distribution and find the mode
11. Find the closest pair
12. Identify statistical outliers
13. Find duplicates in a mailing list
14. Organize an MP3 library
15. Element uniqueness
16. Stability

Searching: It is the process to find the particular element from the array.

Searching Technique: A technique that find the particular element from the array.

The searching techniques are:

1. Linear Searching or Sequential search
2. Binary Search

LINEAR SEARCH

This algorithm searches an element from unordered / ordered vector.

A : Vector (Array) consist N elements
N : Number of elements in a vector
X : Element to be searched

Step -1 : [Initialize **]**

I ⟵ 1

A [N + 1] ⟵ X

Step -2 : [Perform Search **]**

Repeat while A [I] <> X

I ⟵ I + 1

Step -3 : [Successful search or not **]**

```
IF ( I = N + 1 )
THEN
        WRITE ('UNSUCESSFUL SEARCH ')
        Return ( 0 )
ELSE
        WRITE ('SUCESSFUL SEARCH ')
        Return ( I )
```

Advantages of Linear search

1. Linear searching is the basic and simple method of searching.
2. Easy to implement.
3. Useful for searching an element in an unordered or ordered list.

Disadvantages of Linear search

Linear search is time consuming.

BINARY SEARCH

BINARY_SEARCH(A, N, X) : This algorithm is used to find a particular elements X from the array A which is consisting of N elements using binary search. It returns the index of the array if the element is found, otherwise returns 0.

Step -1 : [Initialize **]**

LOW ⟵ 1

HIGH ⟵ N

Step -2 : [Perform Search **]**

Repeat through step – 4 while LOW <= HIGH

Step -3 : [Calculate MIDDLE **]**

MIDDLE ⟵ (LOW + HIGH) / 2

Step -4 : [Compare **]**

```
IF ( X < A [ MIDDLE ] )
THEN
        HIGH ⟵ MIDDLE – 1
ELSE
        IF ( X > A [ MIDDLE ] )
        THEN
                LOW ⟵ MIDDLE + 1
        ELSE
                WRITE ('SUCCESSFUL SEARCH')
                Return ( MIDDLE )
```

Step -5 : [Unsuccessful search **]**

```
WRITE ('UNSUCCESSFUL SEARCH')
Return ( 0 )
```

Advantages of Binay search

1. Binary search is very efficient algorithm.
2. Require fewer number of comparisons as compared to Linear search.

Disadvantages of Binary search

1. Binary search is not useful when the array elements are frequently changed.
2. Array must be sorted to perform binary search.

Applications of searching:

1. Search algorithms can be used to find solutions or objects with specified properties and constraints in a large solution search space or among a collection of objects.

2. There are Search algorithms which are designed for the prospective quantum computer.

3. In text editors, we might want to search through a very large document for the occurrence of a given string.

4. In text retrieval tools, we may want to search through thousands of such documents.

5. String matching algorithms as part of a more complex algorithm (e.g., the Unix program ``diff'' that works out the differences between two similar text files).

6. To search in binary strings (ie, sequences of 0s and 1s). For example the ``pbm'' graphics format is based on sequences of 1s and 0s.

7. Implementing a "switch() ... case:" construct in a virtual machine where the case labels are individual integers.

If you have 100 cases, you can find the correct entry in 6 to 7 steps using binary search, whereas sequence of conditional branches takes on average 50 comparisons.

8. Binary search is now used in 99% of 3D games and applications.

Space is divided into a tree structure and a binary search is used to retrieve which subdivisions to display according to a 3D position and camera.

9. Binary search offers a feature of finding non-exact matches (closest matches).

Sorting V/s Searching:

	Sorting		Searching
1.	It is the process to arrange the data in ascending or descending order.	1.	It is the process to find the particular element from the array.
2.	The most common sorting algorithms are: • Bubble Sort • Insertion Sort • Selection Sort • Quick Sort • Merge Sort • Shell Sort	2.	The most common searching algorithms are: • Linear search • Binary search • Interpolation search
3.	This is the operation of arranging the elements of a table into some sequential order	3.	This is the process by which one searches the group of elements for the desired element.
4.	Sorting returns an array with the elements arranged in ascending or descending order.	4.	Searching returns the position of the searched array
5.	Output of sorting algorithms is sorted elements.	5.	Output of searching algorithm is successful or unsuccessful search.
6.	After performing sorting, searching becomes easy.	6.	Without performing sorting, searching becomes difficult.
7.	After performing sorting techniques, the position of data elements or data records are changed.	7.	After performing searching techniques, the position of data elements or data records are not changed.

---------- X ----------

Class: SYBCA (SEM-IV)
Subject: US04CBCA25 (DATA STRUCTURES – II)

***UNIT – 4 (FILE ORGANIZATION - II)**

**HASHING FUNCTION

INTRODUCTION

Hashing function transfers the key of the record into a direct address by applying it to a predetermined formula.

This **key-to-address** transformation is defined as a mapping or a hashing function H, which will map the key space (K) into an address space (A).

That is, given a key value, a hashing function H produces a table address or location of the corresponding record.

The function generates this address by performing some simple arithmetical or logical operations on the key or some part of the key.

The key space is usually much larger than the address space, many keys will be mapped to the same address. Such a many-to-one mapping results in **collisions** between records.

Collision-resolution technique is required to resolve these collisions.

Definition: A **collision** is said to have occurred when two keys are hashed to the same address and the two keys are called **synonyms**.

Hashing function fall into two classes:

1. Distribution independent and
2. Distribution dependent method

A distribution independent hashing function does not use the distribution of the keys of a table in computing the position of a record.

A distribution dependent hashing function is obtained by examining the subset of keys corresponding to known records.

Desired properties of hashing function

1. Speed
2. Generation of addresses uniformly
3. Easily computable
4. Minimize the number of collisions

Different types of hashing function

1) Division/Remainder Method
2) Mid Square Method

3) Folding Method
4) Digit Analysis Method
5) Length-Dependent Method

1) Division / Remainder Method

It is one of the first hashing functions and most widely accepted method. It is defined as: **H(x) = x mod m + 1** for some integer divisor m.

Operator mod means modulo operator. For example if x=35 and m=11 then

H(35) = 35 mod 11 + 1 = 2 + 1 = 3

This method gives a hash value which belong to set {1, 2, ……,m}.

In mapping key to address, the division method preserves uniformity in a key set to certain extent. Keys which are close to each other or clustered are mapped to unique addresses.

For example, for m=31, the keys 1000, 1001, ……,1010 are mapped to addresses 9, 10, ….,19.

This uniformity is disadvantage if two or more clusters are mapped to same addresses.

For example, another cluster of keys is 2300, 2301,……, 2313 are also mapped to addresses 7,8,……, 20. There are many collisions with keys from the cluster starting at 1000.

The reason for this phenomenon is that keys in the two clusters gives same remainder when divided by m=31.

In general, it is uncommon for a number of keys to give same remainder when **m is a large prime number**. In practice it has been found that odd divisors without factors less than 20 are also satisfactory.

2) Mid Square Method

Another hashing function that has been widely used in many applications is the “Middle of Square” method.

In this method the key is multiplied by itself and its address is obtained by selecting an appropriate number of bits or digits from the middle of the square.

Usually the number of bits or digits chosen depends on the table size and consequently, can fit into one computer word of memory.
The same positions in the square must be used for all the products.

Example Consider a six digit key 123456. Squaring this key will result in 15241383936. If a 3 digit address is required, position 5 to 7 can be chosen. It will give address 138.

This method has certain drawbacks, but gives good result for certain key sets.

3) Folding Method

In this method the key is split (partitioned) into a number of parts. Each part has same length as the required address with the possible exception of the last part.

The parts are then added together ignoring the final carry to form address.

Example Assume that key is 356942781 and it is to be transformed into a 3 digit address..

In the **fold shifting method**: 356, 942 and 781 are added to give 079.

A variation of the basic method will be reversing of digits in outermost parts. This variation is called **fold boundary method**. In this example, 653, 942 and 187 are added to give 782.

4) Digit Analysis Method

This method forms addresses by selecting and shifting digits or bits of the original key. This function is distribution-dependent.

For e.g. a key 7546123 is transformed to the address 2164 by selecting digits in positions 3 to 6 and reversing their order.

For a given key set, the same positions in the key and the same rearrangement pattern must be used consistently.

Initially, an analysis on a sample of the key set is performed to determine which key positions should be used in forming an address.

Digit positions having the most uniform distributions (i.e. the smallest peaks & valleys) are selected.

Example Consider the digit analysis of the sample key set.

A total of 5000 ten-digit keys are analyzed to determine which key positions should be used in forming addresses in the address space. {0, 1, - - - - - -, 9999}.

Positions 2, 4, 5 & 9 have the most uniform distribution of digits, so they are selected.

For e.g. key 1234567890 is transformed to the address 9542 by selecting digits in positions 2, 4, 5, 9 & by reversing their order.

This hashing transformation technique has been used in conjunction with static key sets (i.e. Key sets that do not change over time.)

5) Length Dependent Method

Another hashing technique which has been used in table handling applications is called the length dependent method.

In this method the length of the key is used along with some portion of the key to produce

- either a table address directly or
- an intermediate key which can be used for example with division remainder method to produce final table address.

One function has produced very good results. Add the internal binary presentation of first and last characters and the length of key multiplied by 16.

Example The key PARTNO becomes 215 + 214 + (6 X 16) = 525 using EBCDIC representations. (In EBCDIC – decimal value of P is 215 and O is 214)

If we treat 525 as an intermediate key and apply the divison method with divisor 49, the address will be 36.

****COLLISION-RESOLUTION TECHNIQUES**

***INTRODUCTION**

Hashing function can map several keys into the same address.

When this situation occurs, the colliding records must be sorted and accessed as determined by **"collision-resolution technique".**

There are 2 broad classes of such technique:

1) Open addressing and
2) Chaining.

The general objective of collision resolution techniques is to try to place colliding records elsewhere in the table.

This requires investigation of series of table positions until empty position is found to accommodate a colliding record.

.

1) OPEN ADDRESSING

With open addressing, if a record with key x is mapped to an address location d and this location is already occupied, then other location in the table are examined until a free location is found for the new record.

If a record with key K_i is deleted then K_i is set to a special value DELETE which is not equal to value of any key.

1. Linear Probing

One of the simplest techniques for resolving collisions is to use following sequence of table locations of m entries:

d, d+1, …..m-1, m, 1, 2, …..,d-1

An unoccupied record location is always found if at least one is available. Otherwise, the search will be unsuccessful after examining m locations.

When accessing (retrieving) a particular record, the same sequence of locations is examined

- until that record is found or
- until an unoccupied (empty) record position is found. Here, the desired record is not in the table so the search fails.

This collision-resolution technique is called **linear probing.**

Example Assume that insertions are performed in following order:

NODE, STORAGE, AN, ADD, FUNCTION, B, BRAND and PARAMETER.

The name NODE is mapped into 1.
The name STORAGE is mapped into 2.
The names AN and ADD are mapped into 3.
The names FUNCTION, B, BRAND and PARAMETER are mapped into 9.

			Number of probes
R_1	NODE		1
R_2	STORAGE		1
R_3	AN		1
R_4	ADD		2
R_5	PARAMETER		8
R_6	Empty		
R_7	Empty		
R_8	Empty		
R_9	FUNCTION		1
R_{10}	B		2
R_{11}	BRAND		3

Above figure represents structure with m=11.

- The first 3 keys are placed in single probe.
- ADD must go into position 4 instead of 3 because 3 is already occupied. So number of probes will be 2.
- FUNCTION is placed in position 9 in 1 probe.
- B and BRAND take 2 and 3 probes respectively.
- PARAMETER will be placed into position 5 after 8 probes because positions 9, 10, 11, 1, 2, 3 and 4 are occupied.

Drawback of Linear Probing

- Deletion from table is difficult to perform.

- Clustering effect. The trend is for long sequences of occupied positions (filled slots) to become longer. This is known as **Primary Clustering**.

2. Random Probing

Effect of primary clustering can be reduced by selecting a different probing technique called **Random probing**.

This method generates a random sequence of positions rather than an ordered sequence like linear probing method.

The random sequence generated must contain every position between 1 and m exactly once. A table is full when first duplicate position is generated.

An example of random number generator: $Y \leftarrow (y + c) \bmod m$

Where
- y is the initial position number of random sequence.
- c and m are integers that are relatively prime to each other (i.e. their greatest common divisor is 1).

For example m=11, c=5 and starting with initial value y=2; generates the sequence 7, 1, 6, 0, 5, 10, 4, 9, 3, 8, and 2.

Adding 1 to each number of this sequence transforms all numbers so that they fall in interval [1, 11]

Drawback of Random Probing

Random probing has reduced the problem of primary clustering but all types of clustering have not eliminated.

Clustering occurs when 2 keys are hashed into the same value.

In such case, same sequence of positions is generated for both keys by random probe method. This is known as **Secondary Clustering**.

3. Double Hashing or Rehashing - Solution for Secondary Clustering

To solve secondary clustering problem, second hashing function is used. This second function is independent of first function.

For example, if H_1 is first hashing function where

$H_1(x_1) = H_1(x_2) = i$ for $x_1 \neq x_2$ and i is the hash value.

Means for two keys x_1 and x_2, we have same hash value.

Now, we have second hashing function H_2 such that
$H_2(x_1) \neq H_2(x_2)$

When $x_1 \neq x_2$ and $H_1(x_1) = H_1(x_2)$ we can use $H_2(x_1)$ or $H_2(x_2)$ as the value of parameter c in the random probe method.

The two random sequences generated by this method are different when H_1 and H_2 are independent. The effect of secondary clustering can be reduced.

This variation of open addressing is known as **Double Hashing or Rehashing**.

Example H1(x) = x mod m and H2(x) = (x mod (m-2)) + 1

Table size m=11 and key value x=75, $H_1(75)=9$ and $H_2(75)=4$.

Now Y ⟵ (y + c) mod m with y=9 and c=4
produce the sequence 9, 2, 6, 10, 3, 7, 0, 4, 8, 1, 5

If we chose key 42 then $H_1(42)=9$ same as $H_1(75)=9$ but $H_2(42)=7$

Here Y ⟵ (y + c) mod m with y=9 and c=7
produce the sequence 9, 5, 1, 8, 4, 0, 7, 3, 10, 6, 2

$H_1(75) = H_1(42) = 9$ but two random sequences generated are different.

Difficulties with Open addressing method

- List of colliding records for different hash values become intermixed. So it requires more probes.
- Unable to handle table overflow situation.
- Physical deletion of record is difficult.

2) CHAINING

1. Separate Chaining

One of the most popular methods of handling overflow records is called **separate chaining or chaining with separate lists.**

In this method the colliding records are chained into a special overflow area which is separate from prime area.

Prime area contains the part of table into which records are initially hashed. A separate linked list is maintained for each set of colliding records.

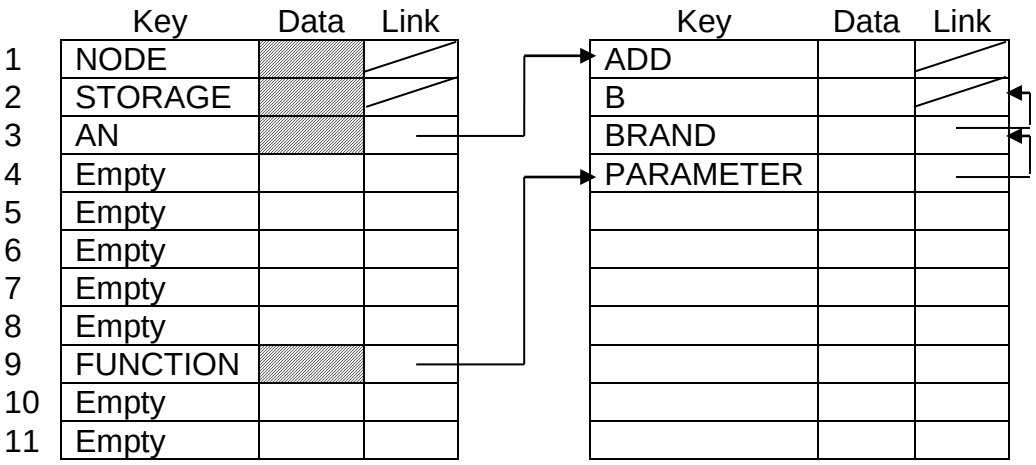
Therefore, a pointer field is required for each record in the primary and overflow areas.

Consider m =11 and n=9. Assume that insertions are performed in following order:

NODE, STORAGE, AN, ADD, FUNCTION, B, BRAND and PARAMETER.

Colliding records are in each linked list are not kept in alphabetical order.

When a new colliding record is entered in the overflow area, it is placed at the front of those records in the appropriate linked list of the overflow area.



Primary area Overflow area

2. Independent Chaining

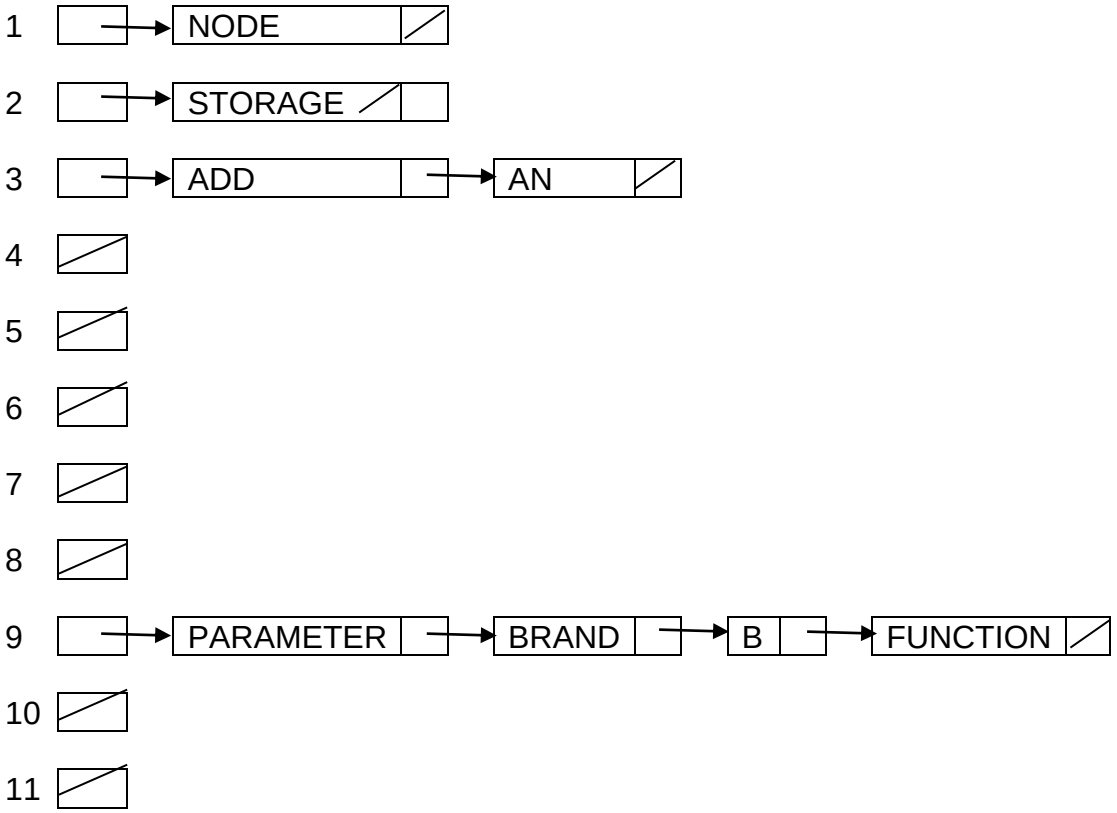
More efficient representation of separate chaining

In this representation, all the records reside in the overflow area while the prime area contains only pointers.

For a table with large records, this approach results in a heavily packed overflow area. But the hash table can be made large without wasting much storage.

Assume that insertions are performed in following order:

NODE, STORAGE, AN, ADD, FUNCTION, B, BRAND and PARAMETER.



****DIRECT / RANDOM FILES**

Random or direct organization of a file implies that a particular record can be accessed directly, without scanning other records that precede it in the file.

Example-1

Consider a customer billing system.

To accommodate any type of on line processing like checking the bill amount or verifying list of goods purchased so far etc., individual customer records must be accessed directly.

It is also desirable that records are ordered sequentially by account number. This is necessary to generate monthly customer bills based on receipts which are received in batches from point of sale.

In near future, company may want to remove this system of filling purchase slip at point of sale and then sending slips to main office for computer processing.

A simple but more expensive approach is: A purchase or return is posted against a customer account immediately via on-line terminals operated by point-of-sale clerks.

The on-line terminals are remote from the computer, yet tied directly to it via communication lines.

When purchases and returns are handled at the point of sale, it is required to batch all the customer receipts and sequentially process them against the account via the merge update procedure.

When the need for sequential processing is eliminated, we can design a system that requires only capability of direct access.

Example-2

In a railway reservation system, passengers demand reservation for any train, for any date and for any kind of seats (2 tier AC, 3 tier AC, Second Class etc) in random order.

Here, the train data are required to be accessed randomly.

STRUCTURE OF DIRECT FILES

In a direct (random) file, a transformation or mapping is done from the key of a record to the address of the storage location at which that record is to be located in the file.

One mechanism used for transformation is hashing algorithm.

Hashing algorithm has 2 components:

1. Hashing function which defines a mapping from key space to address space and
2. Collision-resolution technique which resolves conflicts that occur when more than one record is mapped to the same table location.

Hashing algorithms used for direct files are very similar to those used for tables.

Main conceptual differences are due to physical characteristics of external storage which are different from tables as tables are directly addressable.

The time to access a record in a table in main memory is in microseconds while time to access a record in external memory is in milliseconds.

How records are stored in Direct file organization

Records in a file are stored in **buckets** in direct file organization.

Each bucket contains b record locations, as opposed to just one record in a location of a hash table.

The number of records in a bucket is called the **bucket capacity**.

To find a particular record,

- the bucket must be located in which the record resides
- the contents of bucket is brought into a buffer in memory and then
- the desired record is extracted from buffer.

Let us define an **address space** A of size m such that A = {C+1,C+2,.. , C+m }, where C is an integer constant.

Then mb records can be accommodated by A.

Consider a key set S = {X_1,X_2, . . . ,X_n} which is a subset of the set K of possible keys which is called **key space**.

If the size of K is equal to the number of record locations in A and the key is consecutive then a transformation can be defined which assigns exactly b keys from K to each bucket of A.

This type of one-to-one transformation is called **direct addressing**.

In most situations, S is a small subset of K.

So indirect addressing is implemented. i.e. S is mapped into A with possibility that many records will be assigned to same bucket and bucket overflow occurs.

When this happens, bucket overflow handling technique must be used to store overflow records.

PROCESSING DIRECT FILES

Processing of a direct file is dependent on how the key set for the records is transformed into external device addresses.

Direct files are primarily processed directly. That is, a key is mapped to an address.

And depending on the nature of the file transaction; a record is created, deleted, updated or accessed.

A record may be at that address or possibly at some subsequent (successive) address if a collision takes place.

Subsequent address is determined by the overflow handling technique which is adopted.

**INDEXED SEQUENTIAL FILES

In a customer billing system, inquiry about account status of any customer is required to be handled at any time. To handle such on-line query quickly, ability to go directly to required record must be available.

Also we need to process monthly reports by accessing customer records in sequential fashion.

So, the file structure is required which must support both direct and sequential access. Indexed sequential file organization supports both direct and sequential access.

STRUCTURE OF INDEXED SEQUENTIAL FILES

An important aspect affecting the file structure is the type of physical medium on which the file resides.

The capability of directly accessing a record is based on a key (or unique index). It can only be achieved if the external storage device used supports this type of access.

EXTERNAL STORAGE DEVICES

Devices such as card readers and tape units allow the access of a particular record only after reading all the other records that physically appear before a desired record in the file.

So direct record access is impossible for these types of devices.

The types of external storage devices that support both direct and sequential access are magnetic drums and disk units.

The file-structure concepts relating to indexed sequential files are best illustrated when we consider a magnetic disk as the storage medium.

In fact, because of their low price / performance ratio and large total storage capacity, disks are generally chosen when using indexed sequential files.

2 TYPES OF INDEXED SEQUENTIAL FILE ORGANIZATIONS

1) used by IBM
2) used by CDC

IBM INDEXED SEQUENTIAL FILE

An IBM's indexed sequential file consists of 3 separate areas:

1. **Prime area**
2. **Index area**
3. **Overflow area**

1. Prime area

The prime area is an area into which data records are written when the file is first created.

The file is created sequentially, that is, by writing records in the prime area in a sequence dictated by the lexical ordering of the keys of the records.

The writing process starts at the second track of a particular cylinder say the n^{th} cylinder, of a disk.

When this cylinder is filled, writing process continues on the second track of the next $(n+1)^{th}$ cylinder. It continues in this fashion until the file's creation is completed.

If the newly created file is accessed sequentially according to the key item, the records are processed in the order in which they were written.

2. Index area

The second important area of an indexed sequential file is the index area. It is created automatically by the data-management routines in the operating system.

A number of index levels may be involved in an indexed sequential file.

1) **Track index**
2) **Cylinder index**
3) **Master index**

1) Track index

The lowest level of index is the track index.

It is always written on the first track (named track 0) of the cylinders for the indexed sequential file.

The track index contains two entries for each prime track of the cylinder – **a normal entry and an overflow entry**.

The **normal entry** is composed of

- Address of the prime track to which the entry is associated and
- Highest value of the keys for the records stored on track.

- If there are no overflow records, the **overflow entry** is set equal to the normal entry.

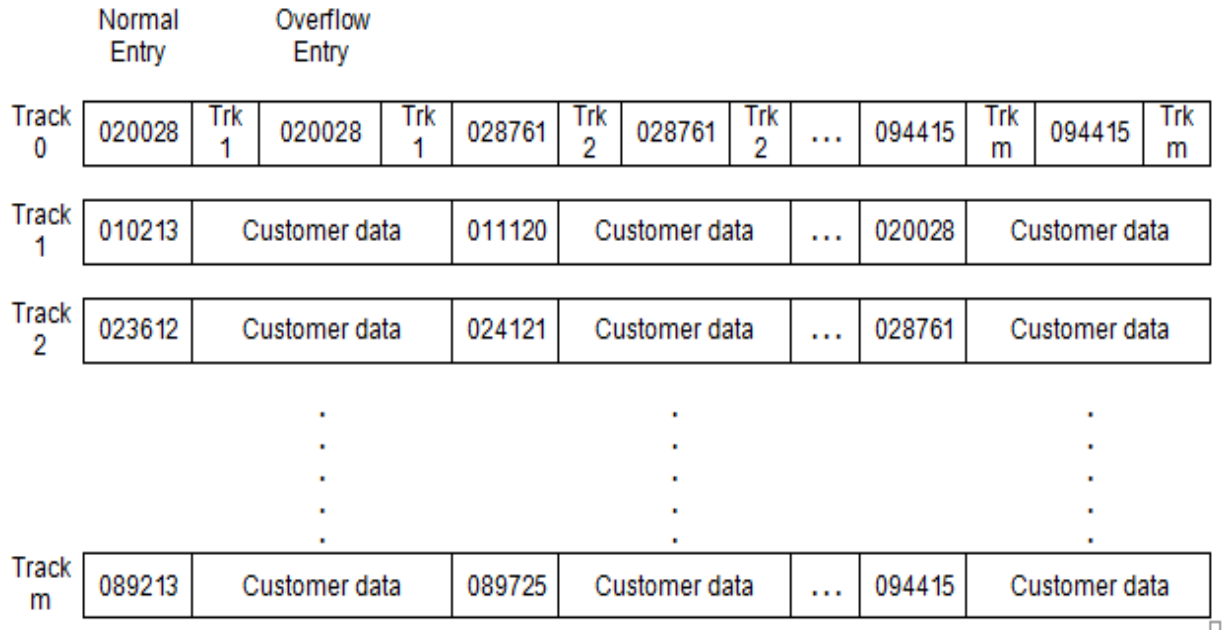


Figure- Track index and prime area of an indexed sequential file

Figure illustrates the file structure of an indexed sequential file of customer records in which the key item is an account number.

Only one cylinder is shown with a prime area of m tracks.

2) Cylinder index

Track index describes the storage of records on the tracks of a cylinder.

Same way, the cylinder index indicates how records are distributed over a number of cylinders.

A cylinder index references a track index—one cylinder index entry per track index.

3) Master index

A final level of indexing exists in this hierarchical indexing structure.

A master index is used for an extremely large file where a search of the cylinder index too time consuming.

This index forms the root node of the tree of indices used in indexed sequential file.

Relationships between the different levels of index

Following Fig-5 explains relationships between the different levels of index.

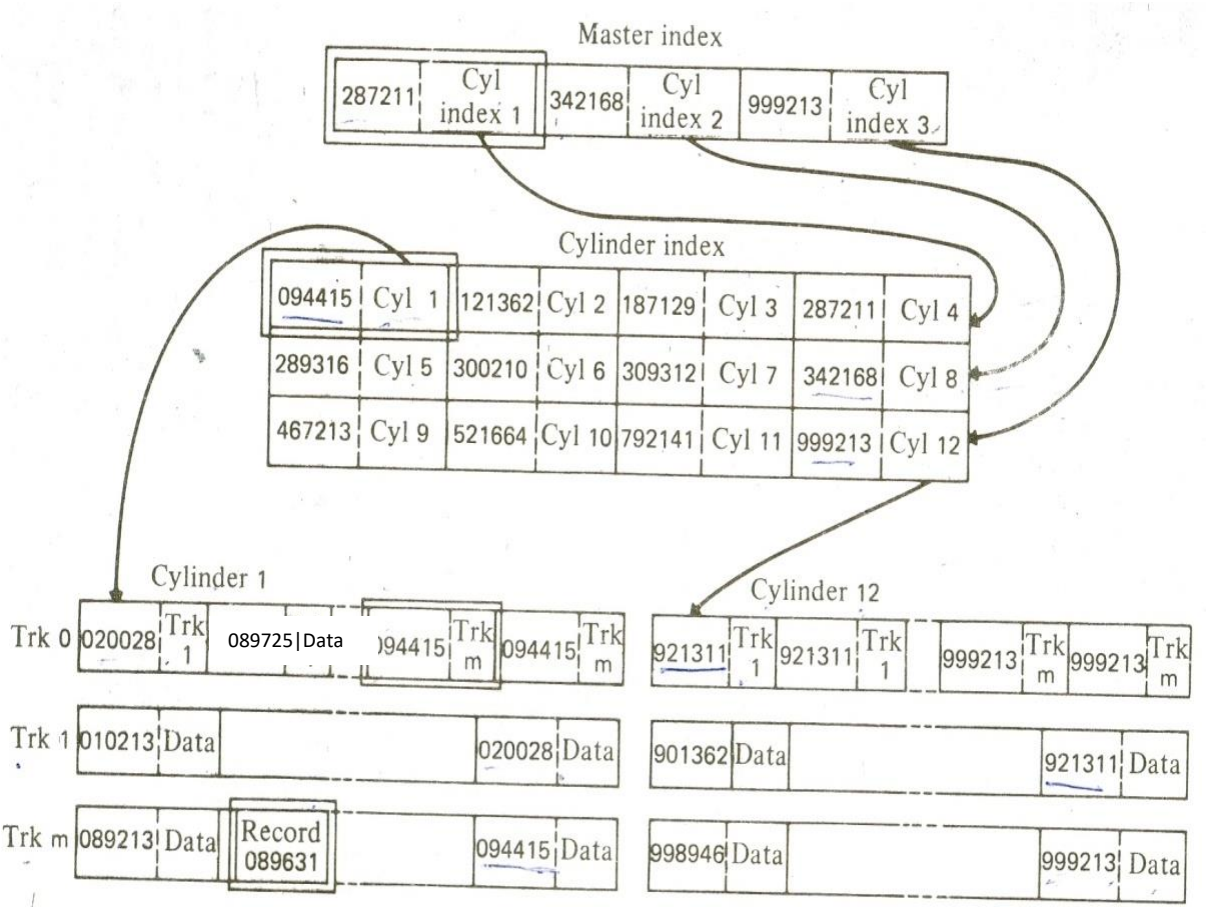


Figure- Relationships between the different levels of index

This example assumes that there are 12 cylinders on which records are stored.

Track 0 of every cylinder is the track index of a particular cylinder. It contains m entries because there are m prime tracks per cylinder.

Cylinder index contains 12 entries – one per cylinder.

Master index contains 3 entries – one per cylinder index.

Searching a record from Indexed Sequential file

- Locating the record corresponding to the customer with account number **089725** involves a search of the master index to find the proper cylinder index with which the record is associated (e.g., cylinder index 1).
- Next, a search is made of the cylinder index to find the cylinder on which the record is located (e.g. cylinder 1).
- A search of the track index produces the track number on which the record resides (e.g., track m).
- Finally, a search of the track is required to locate the desired record.

Adding a record to Indexed Sequential file

If records are added to a sequential file, a new sequential tile must be created. We can use the same approach when handling additions for an indexed sequential file.

It is possible to access records directly in an indexed sequential file.

So this type of file is generally used in a more volatile and quick-response demanding environment, i.e., an environment in which many additions and deletions arise from on-line transactions or small batches of transactions.

Such deletions and additions must be immediately reflected in the flit one cannot wait until the end of months.

3. Overflow area

The problems of adding records are handled by creating an overflow area or areas, usually on the same device on which the file resides.

1) **Cylinder overflow area**
2) **Independent overflow area**

1) Cylinder overflow area

A cylinder overflow area is a number of dedicated tracks on a cylinder that contains a number of prime-area tracks.

Addition of a record

If an overflow is created in the prime-area tracks of the cylinder through the addition of a record, then the overflow records are stored in the cylinder overflow area.

An overflow record is identical to a prime record, except that a track / record address field is added to the end of the record.

We assume that one track contains only three records (unrealistic example).

Example: Addition of a record with key 026924

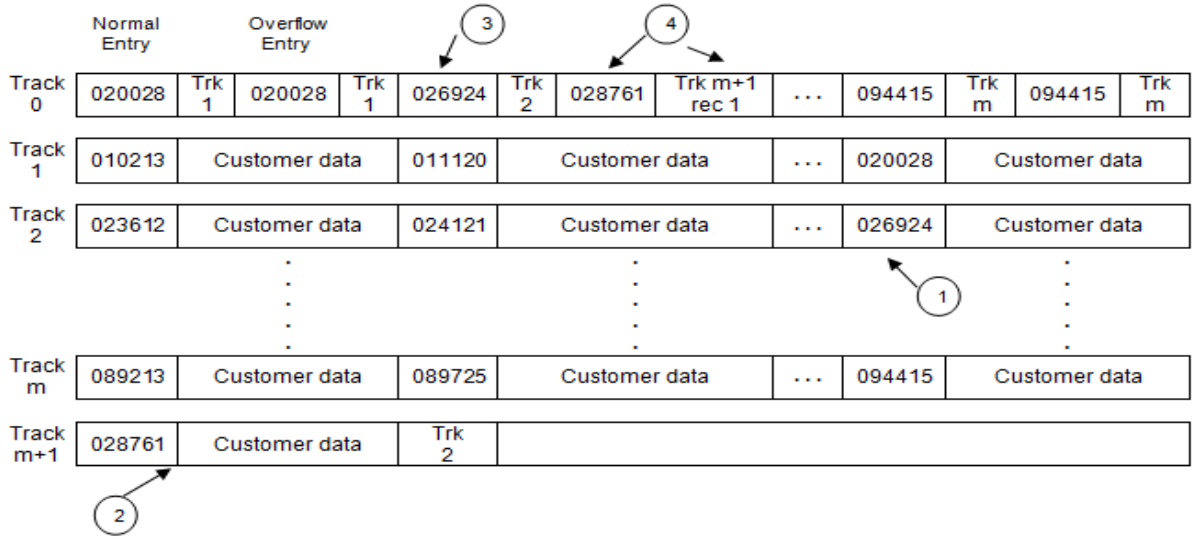


Figure - Effects of an overflow in an indexed sequential file

The addition of a record with key 026924

- Initially, a customer record with an account number of **026924** is added to the file, as depicted in Fig. 6(a) - ①.

- The record with account number 028761 must be moved to the cylinder overflow area at track m + 1.

  Track 2 is placed in the link field of over flow record 028761.

  This is done to preserve the sequential ordering of records in track 2 of the prime area. Fig. 6(a) - ②

  This change demands **two other alterations** to the file.

- **First**, the normal entry in the track index for track 2 must be changed from 028761 to 026924. Because 026924 is now the highest key value for the track. Fig. 6(a) - ③

- **Secondly**,

  - ❖ Overflow entry is adjusted so that its first subentry contains the largest key value of any record for track 2 (i.e.. the value 028761). Fig. 6(a) - ④
  - ❖ And the second entry is set to the track / record address of the overflow record with the smallest key value for track 2. Fig. 6(a) - ④

Example: Addition of a record with key 021008

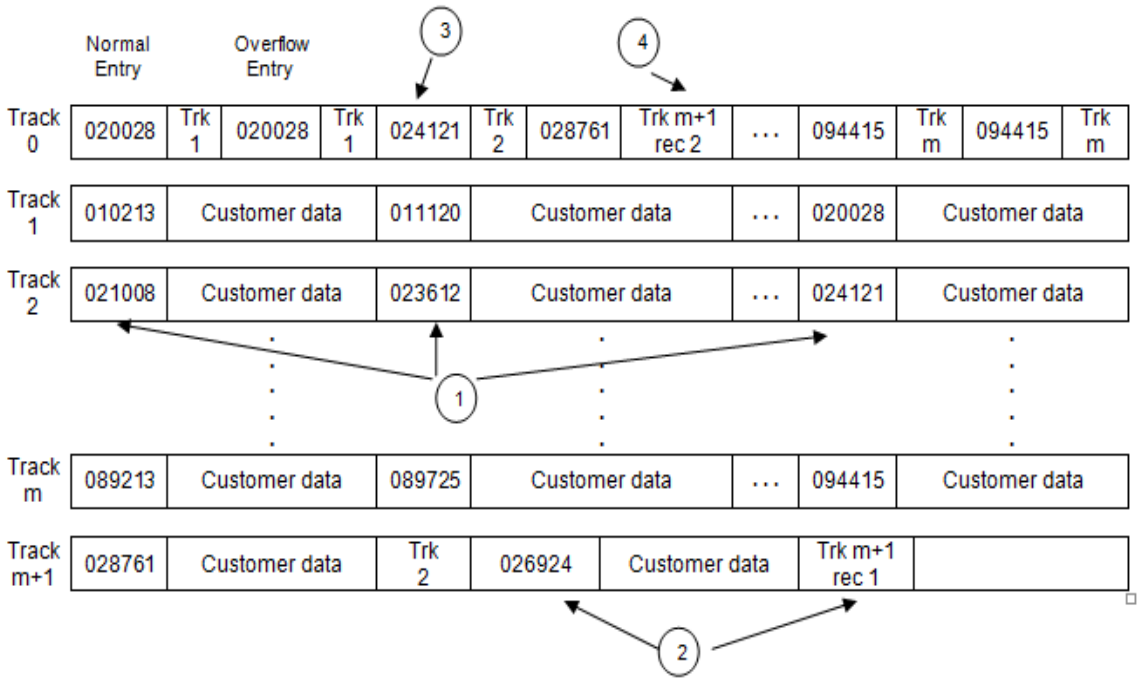


Figure - Effects of an overflow in an indexed sequential file

The addition of a record with key 021008

- Record with key 023612 and 024121 will be shifted one record position on right side to accommodate record with key 021008. Fig. 6(b) - ①

- Record with the key value 026924 becomes an overflow record with the addition of the record with a key value of 021008.

  The track / address field is set to point to the record with key value 028761. Fig. 6(b) - ②

- Normal entry in the track index for track 2 must be changed from 026924 to 024121. Because 024121 is now the highest key value for the track. Fig. 6(b) - ③

- Second entry of Overflow entry for track 2 is set point to the overflow record with the next largest key value in the list of overflow record for track 2. Fig. 6(b) - ④

An overflow record is same as prime record except that a track/record address field is added to the end of record.

This track / record address field contains a pointer to the overflow record with the next largest key value in the list of overflow records for a particular track.

So when the record with key 026924 becomes an overflow record after addition of record with key value 021008, the track/address field is set to point to the record with key value 028761.

In general, there may be a number of cylinder overflow tracks, and the overflow records for each track are grouped together in a linked list.

he head of the link list is given by the track/record address in the track index.

The final record in the linked list is specified by placing the number of the associated prime track in the track / record address field of the overflow record (track 2 is placed in linked field of overflow record 028761).

2) Independent overflow area

As more and more records are added to the indexed sequential file the cylinder overflow area becomes full.

When this happens, further overflow records are transferred to an **independent** overflow area.

This can be done if such an area is specified when the file is created. The independent overflow area resides on a cylinder or cylinders apart from any prime-area cylinder.

Overflow records are linked together in the same manner as they are in the cylinder overflow area.

Note, however for disks with movable heads, the use of independent overflow areas should be discouraged because significant number of seeks are generated when the access arm is moved between the prime and independent overflow areas.

Deletion of a record

In IBM indexed sequential organization, deleted records are not physically removed from a file, but are just marked as deleted by placing ‘11111111’ B (all ones) in the first byte of the record.

If a new record is added later which has the same key as a record previously deleted, then the space occupied by a deleted record is recovered.

Reorganization of a file

Records which are placed in an overflow area are never moved back into the prime area because of a deletion. Only by reorganizing the file can an overflow record be placed in the prime area.

Reorganization is achieved by sequentially copying the records of the file into a temporary file and then recreating the file by sequentially copying the records back into the original file.

Because the retrieval of overflow records can carry a large overhead, the amount of disorganization in an indexed sequential file should be monitored closely.

A good rule of thumb when using movable head disk is to reorganize when records must be placed in the independent overflow area.

PROCESSING INDEXED SEQUENTIAL FILES

Organization of an indexed sequential file is much more complex than a sequential file.

Because of this complexity, most operating systems provide access facilities or methods to handle the file structure changes. These changes result from the insertion and deletion of records.

The **main advantage of an indexed sequential file** is that records can be processed either sequentially or directly.

The sequential processing of an indexed sequential file is logically identical to the sequential processing of a sequential file. i.e., records are processed in a sequence determined by the index item.

The types of transactions that are performed are the reading, alteration, addition, and deletion of records.

These are accomplished at a user level with READ, WRITE, and REWRITE statements.

Types of transactions and the operations used to effect these transaction types are usually the same for the sequential processing of sequential and indexed sequential flies.

But the manner in which the records are accessed is substantially different, due to the differences in the file structures.