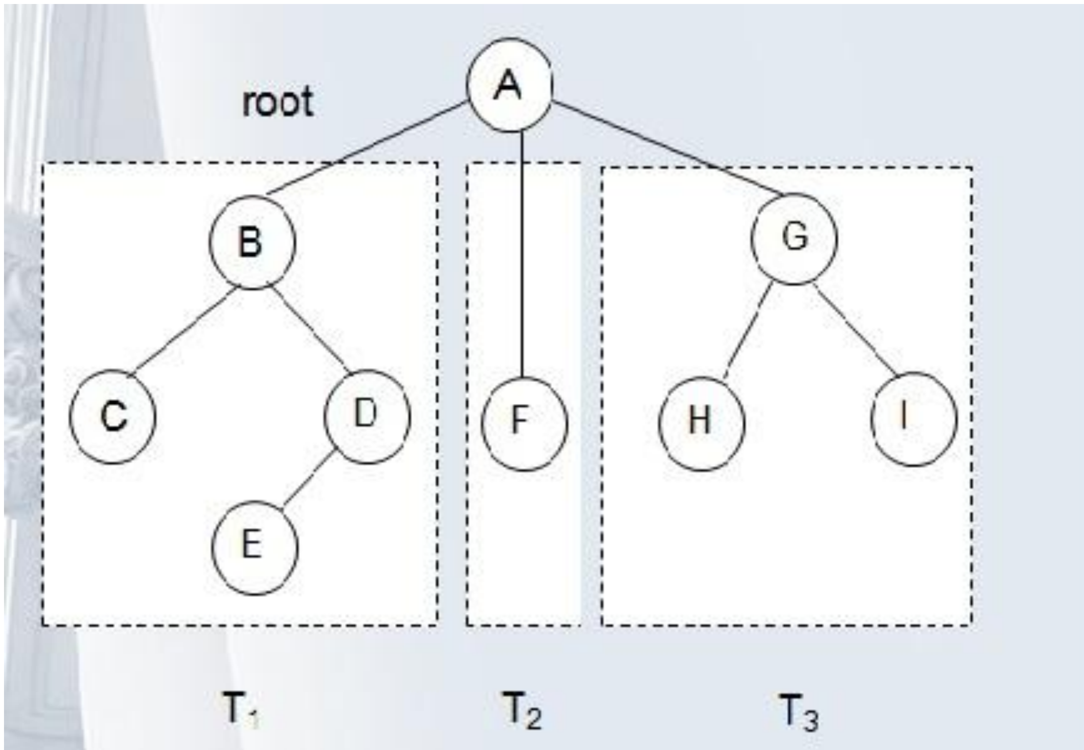
Binary Trees, Sorting and Searching

Unit 5

Introduction

- In a tree structure the data are organized in a hierarchical manner.
- **An acyclic graph is called a Tree.**
- A **rooted tree** is a finite set of one or more nodes such that
  - It has a specifically designated node called as ***root***.
  - The remaining nodes are partitioned into ***n*** disjoint sets $T_1, T_2, T_3 \ldots. T_n$ where each T_i is a tree and $T_1, T_2, T_3 \ldots. T_n$ are called ***sub-trees*** of the root.



This is a general rooted tree with root at node A.
It is **not** a Binary tree

Recursive definition of Binary Trees

- A **binary tree** is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the **root** of the tree.
- The other **two subsets are themselves Binary Trees** called the **left** and the **right subtrees** of the original tree.
- The left and right subtrees can be empty.

Binary Trees

- Each element of a binary tree is called a **Node** of the tree.
- The nodes that have no descendant are called ***leaf nodes***. i.e., The node with **no children.**
- **Few terminologies:**

  Father, child, ancestor, descendant

Definitions

- ***Node***:
  – It stores the actual data and is linked to the other node.
- ***Predecessor (Ancestor)***:
  – Every node in the tree except the ***root*** has a unique parent. All parents in the path from node till root are called the predecessors of node P.
  – In the above ex ***B*** is the predecessor of ***C, D*** and ***E***.
- ***Parent(Father)***:
  - It is the immediate predecessor of a node.
- ***Successor***:
  – Every node except the ***leaf*** nodes can have children called ***successor***.

Definitions

- ***Root***:
  - It is a node that does not have a parent.
- ***Child***:
  - If the immediate predecessor of a node is the parent of that node then the immediate successor is the child.
- **Level**
  - The **root** is at **level 0.** all the other nodes, **one more than its father.**
  - If the node is at level l, its children will be at level l+1.

Definitions

- **Height/Depth**
  - The **maximum level of any leaf** in the tree.
  - **Length of the longest path** from Root to any Leaf.

- **Strictly Binary Tree:** Every non leaf node in a binary tree has nonempty left and right subtrees. This type of tree with **n leaves always has 2n – 1 nodes**

Definitions

- **Complete Binary Tree**

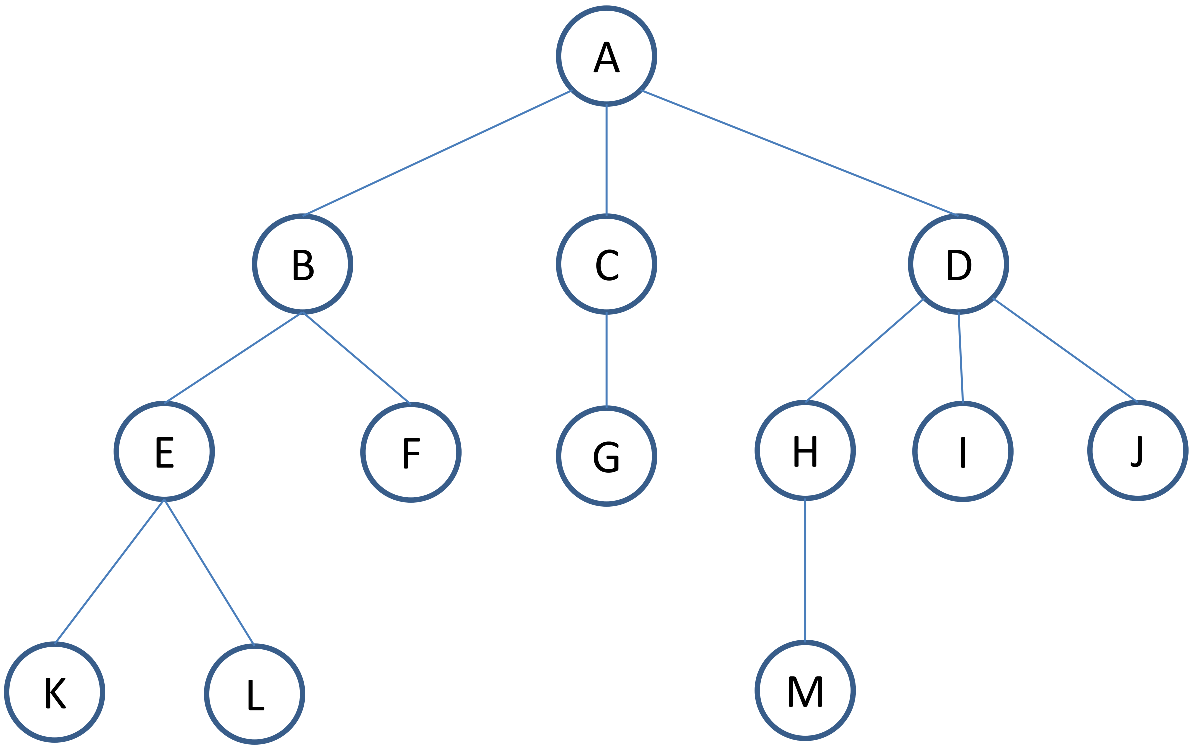
  _ It is a strictly binary tree where **all leaves are at same level.**
- If it contains **m nodes at level l,** then it contains **2m nodes at level l+1** and **2^l nodes at level l**
- **Almost complete Binary Tree**
  - any node in level less than d-1 has two children
  - In level d, if right child is there then left child has to be there

Representation of Binary Trees

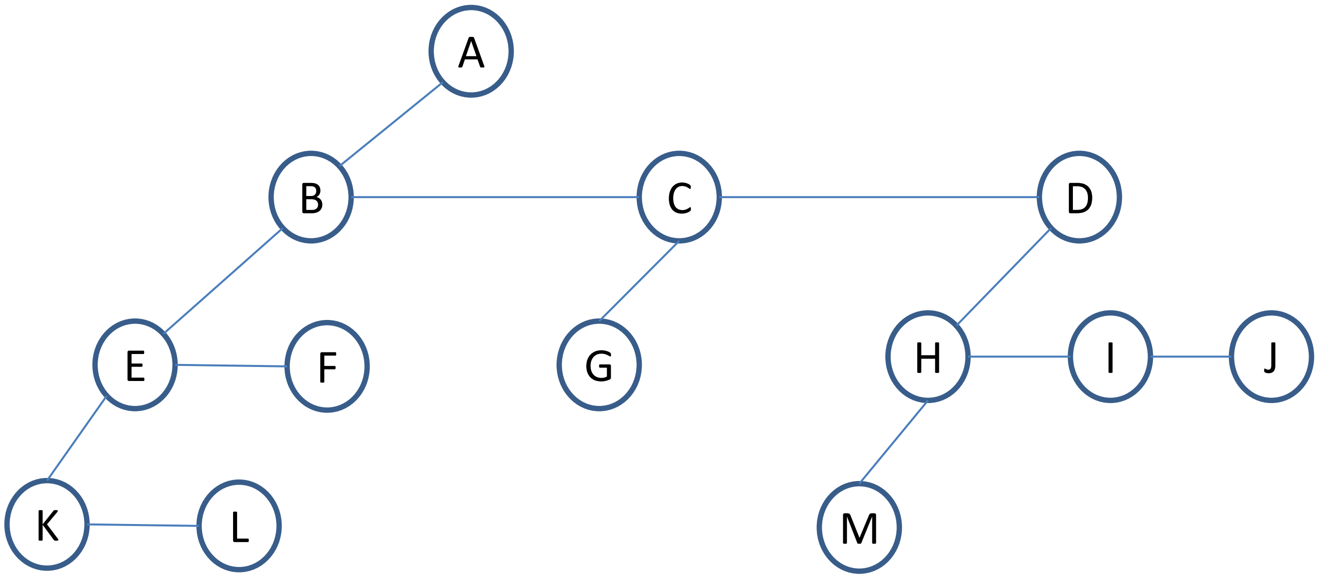
- Array Representation
- Linked list Representation
- **Using Left child – Right Sibling** Representation any general tree can be converted into Binary Tree.

General Rooted Tree - Example

Consider the following example.

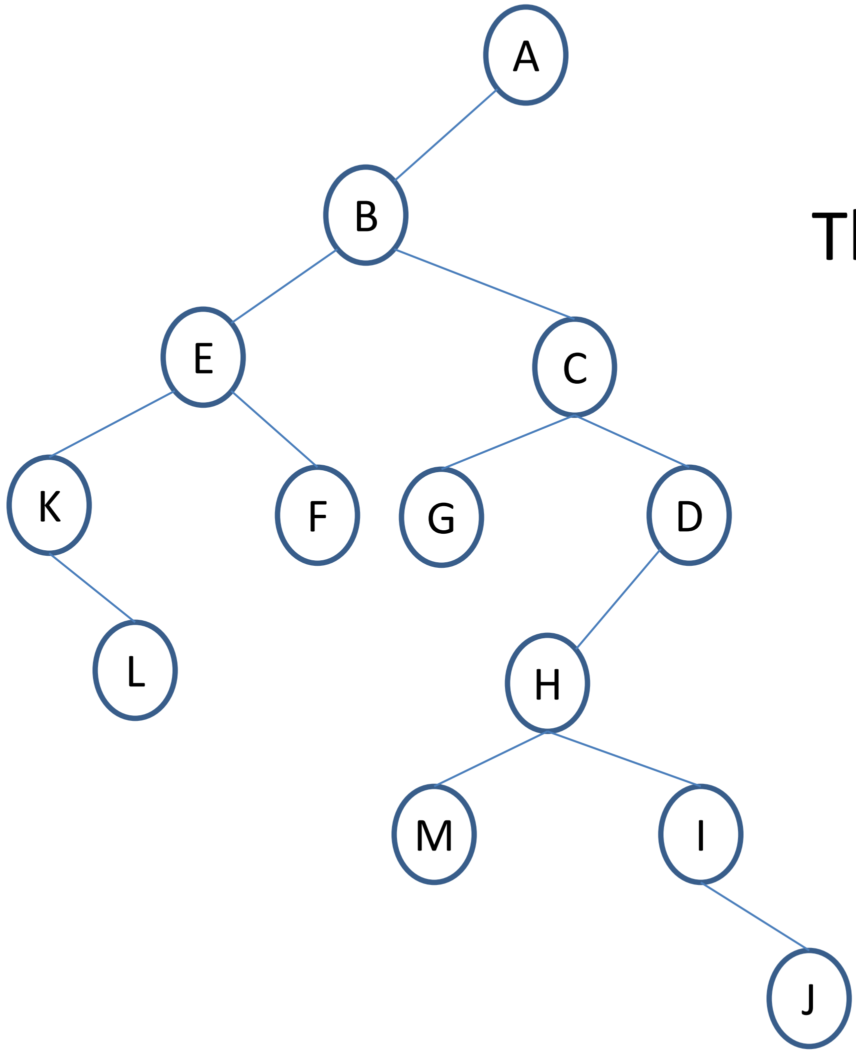


Left child – Right Sibling Representation - Example



- Rotate the right-sibling pointers in a left child right sibling tree clockwise by 45^{o}.
- The right child of the root node is always empty.

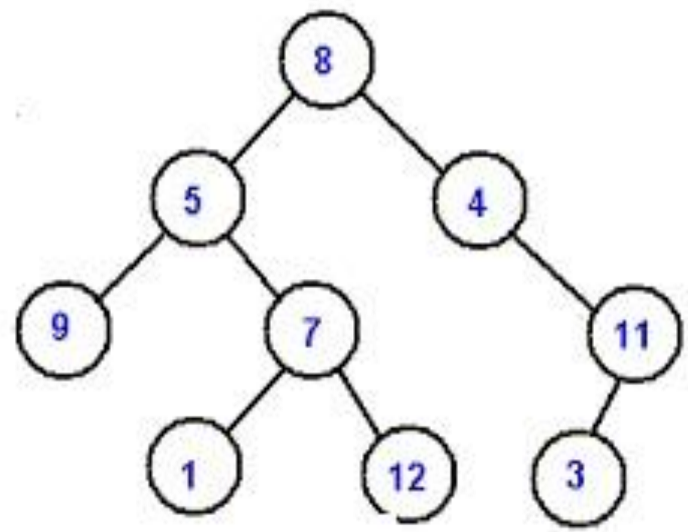
Left child – Right Sibling Representation – Contd..



This is a Binary Tree

Array representation of Binary Trees

Consider the Binary Tree in the figure. Of we allocate the index to the nodes starting from '0' from the root , left to right, then the elements can be stored in an array as follows.



Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Element	8	5	4	9	7	-	11	-	-	1	12	-	-	3	-

To write such representation, create almost complete extensions of Binary Trees

Array representation of Binary Trees – Some Properties

- In such representation, Root is at position 0
- If we know the position of a **node P**, then its **left child is at position 2P+1 and right child is at position 2P+2.**
- If we know the position of a **node P**, then its **parent** is at position **(P-1)/2**
- All **left childs** are in **Odd positions** and **right childs** are in **even positions**
- **Isleft** can be verified as **(p%2 != 0)**

Linked Representation

- Each node has three fields
  - Leftchild
  - Data
  - rightchild
- Node Structure is as follows

Left	ITEM	Right

Representation in C

```
struct node
{
    int info;
    struct node * left;
    struct node * right;
} ;
```

- typedef struct node NODE;

Create a single node Binary Tree

```
NODE * maketree(int x)
{
    NODE * p;
    p = getnode();
    p->info = x;
    p->left = p->right = NULL;
    return p;
}
```

Operations on Binary Trees

- If P is a pointer to a node in a binary tree, then

info(p) – returns the contents of the node

left(p) – returns the pointer to the left child

right(p) – returns the pointer to the right child

father(p) – returns the pointer to the parent

these functions returns NULL in case of failure.

isleft(p), isright(p) returns True if P is a left or right child respectively. Otherwise returns false.

Isleft(p) implementation

```
int isleft(NODE *p)
{  NODE * q = p->father;
    if (q == NULL)
      return 0;  // return false
    if (q->left == p)
       return 1;  // return true
   return 0;   // return false
}
```

Creating left and right childs

- Setleft(p,x) – accepts a pointer P to a binary tree node with no left child. It creates a new left child of node p with info field as x.
- Similarly, Setright(p,x) – accepts a pointer P to a binary tree node with no right child. It creates a new right child of node p with info field as x.

Applications of Binary Trees

- It is a very useful Data Structure when Two-way decisions are to be made
- Creating Expression Trees
- Tree Traversals
- Creating Binary Search Trees

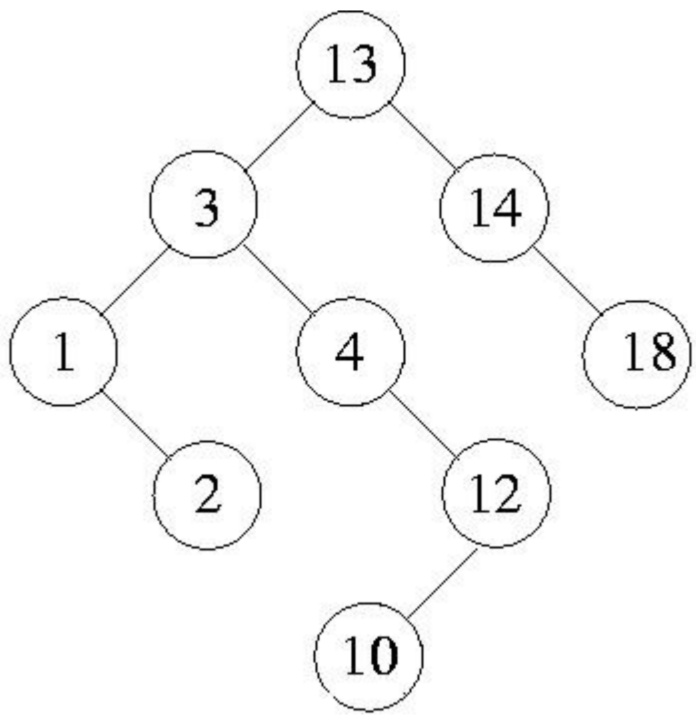
Binary Search Trees (BST)

Creating a binary-search tree of non repetitive integers:

- **First number** is placed in a node that is established as the **Root of the Binary tree** with empty left and right subtrees.
- Each **successive number** is compared with the root, if it is **smaller, examine the left subtree**. If it is **larger, examine the right subtree**(repeat the process)
- When **empty subtree** is found, **insert** the number at that position.

Creating Binary Search Trees

- Consider the list of numbers 13, 3, 4, 12, 14, 10, 18, 1, 2



Binary Search Trees (BST)

- In a **BST,** all elements in the **left subtree** of a node n, is **less than the node,** and all elements in the **right subtree** of n are **greater than or equal** to node n.
- If a BST is traversed in inorder, the centents are printed in ascending order.

Binary Tree Traversals

- **Traversal**
  - **Visiting each node in the tree once and only once.**
- A full traversal leads to a linear order for the nodes in a tree.
- There are three traversal methods

  -Inorder

  -Preorder

  -Postorder
- All are recursive in nature

Preorder Traversal

- Visit a node
- Traverse left and continue
- When you cannot continue move right and begin again.
- If not move back until you can move right and continue.

Preorder Traversal

- Visit the Root
- Traverse the left subtree in preorder
- Traverse the right subtree in preorder

Note: nothing need to be done to traverse an empty binary tree.

It is also called as **depth-first order**

Inorder and Postorder Traversal

Inorder Traversal:

- Traverse the left subtree in inorder
- Visit the Root
- Traverse the right subtree in inorder

Note: It is also called as **symmetric order**

Postorder Traversal:

- Traverse the left subtree in postorder
- Traverse the right subtree in postorder
- Visit the Root

Binary Tree Traversals in C

```
void pretrav(NODE * root)
{
     if (root != NULL)
      {
        printf("%d\n", root->info); // visit the root
        pretrav(root->left);  // Traverse left subtree
        pretrav(root->right;//Traverse right subtree
       }
      return;
 }
```

Binary Search Trees

```
// C program to demonstrate insert operation in binary
   search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
   int info;
   struct node *left, *right;
};
typedef struct node NODE;
```

Create a new node

```
// A utility function to create a new BST node
NODE * newnode(int item)
{
   NODE * temp =  (NODE *)malloc(sizeof(NODE));
   temp->info = item;
   temp->left = temp->right = NULL;
   return temp;
}
```

Inorder Traversal

```
// A utility function to do inorder traversal of BST
void inorder(NODE *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d \n", root->info);
        inorder(root->right);
    }
}
```

Insert a node

```
/* A utility function to insert a new node with given key in BST */
NODE * insert(NODE * p, int key)
{   /* If the tree is empty, return a new node */
    if (p == NULL)
        return newnode(key);
    /* Otherwise, recur down the tree */
    if (key < p->info)
        p->left  = insert(p->left, key);
    else if (key > p->info)
        p->right = insert(p->right, key);
    /* return the (unchanged) node pointer */
    return p;
}
```

Main function

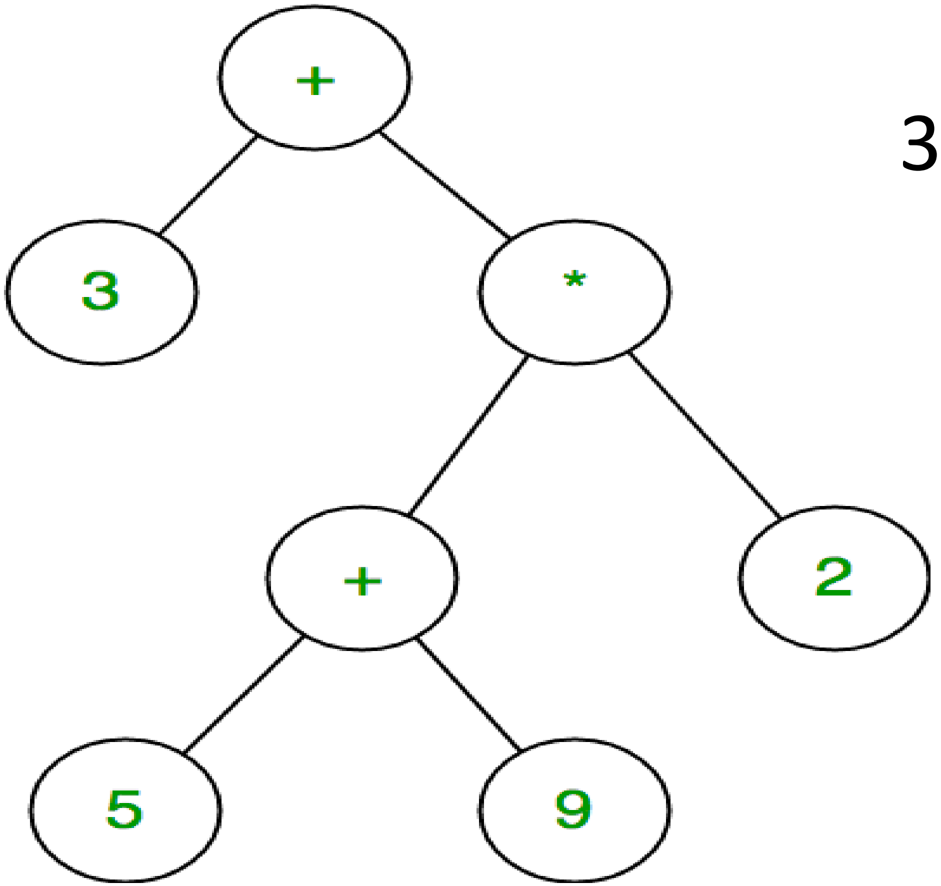
```
int main()
{
   NODE *root = NULL;
   root = insert(root, 50);
   root = insert(root, 30);
   root = insert(root, 20);
   root = insert(root, 40);
   root =  insert(root, 70);
   root = insert(root, 60);
   root = insert(root, 80);

   // print inorder traversal of the BST
   inorder(root);

   return 0;
}
```

```
/* Let us create following BST
          50
        /      \
      30      70
     /  \       /  \
   20   40  60   80 */
```

Expression Trees



3 + ((5+9)*2)

Sorting

- Why sorting?

  - subsequent **searching becomes easy.**
- **Internal sorting** (all elements are in main memory)
- **External sorting** (some of the elements are in auxiliary memory)
- Sorting algorithm is called **stable** if two elements i and j maintains their order even in sorted file

Efficiency Considerations

- Time efficiency
- Space efficiency
- Asymptotic notations
- Different sorting techniques – Bubble sort, quick sort, selection sort
- **Tree sort – create a BST and do inorder traversal**

  if input is sorted then ($O(n^2)$) or

  if input is balanced then $O(n \log n)$
- Heap Sort, Insertion sort

Simple Insertion Sort

```
void insertsort(int x[],int n)
{    int i, k,y;
     /* initially x[0] is sorted file. Insert first element of
     unsorted list into the sorted list*/
   for (k=1; k<n; k++)  {
       y = x[k];
        /* move all elements greater than y  by 1 position*/
       for (I = k-1; i>=0 && y<x[i]; i--)
            x[i+1] =x[i];
       x[i+1] = y;  // insert y at proper position
       } //end of outer for
}
```

Efficiency

- On a **sorted list** time complexity is **O(n)**
- List with **reverse order** **$O(n^2)$**
- It is **very good** algorithm for **almost sorted arrays**
- It can be still improved by treating an **array as a linked list.** Because in a list it is easy to do insertions. This is called **list insertion sort.**

Shell sort

- More improvement in insertion sort is achieved by using **diminishing increment sort also called as shell sort (name of the discoverer).**
- It sorts separate subfiles of original files which includes the kth element. Eg: if k=5 then subfile include x[0], x[5], x[10],….so on

Subfile1 – x[0], x[5], x[10], ….

Subfile2 - x[1], x[6], x[11], ….

.

.

Subfle 5 - x[4], x[9], x[14], ….

ith element of the jth subfile is x[(i-1) * k + (j-1)]

- Sort each subfiles
- Choose **new smaller value for k** and repeat the process for larger subfiles.
- Repeat this process **until k = 1**

Shell Sort

```
void shellsort(int x[],int n, int incrmnts[],int numinc)
{    int i, k,y, incr, span;
     /*  span is the size of the increment*/
 for (incr =0; incr < numinc; incr++) {
   span = incrmnts[incr];
   for (k=span; k<n; k++)  {
        y = x[k];
         /* move all elements greater than y  by 1 position*/
        for (I = k-span; i>=0 && y<x[i]; k-=span)
              x[i+span] =x[i];
        x[i+span] = y;  // insert y at proper position
        } //end of first outer for
      } //end of second outer for
  }
```

Address Calculation Sort

- Thank You