

Function to insert node in the end of the List,

```
struct Node *addAfter(struct Node *last, int data, int item)
```

```
{
    if (last == NULL)
        return NULL;
    struct Node *temp, *p;
    p = last -> next;
    // Searching the item.
    do
    {
        if (p -> data == item)
        {
            temp = (struct Node *)malloc(sizeof(struct Node));
            // Assigning the data.
            temp -> data = data;
            // Adjusting the links.
            temp -> next = p -> next;
            // Adding newly allocated node after p.
            p -> next = temp;
            // Checking for the last node.
            if (p == last)
                last = temp;
            return last;
        }
        p = p -> next;
    } while (p != last -> next);

    cout << item << " not present in the list." << endl;
    return last;
}
```

**Module-2:**  
**Lecture-11**

**Memory Allocation-**

Whenever a new node is created, memory is allocated by the system. This memory is taken from list of those memory locations which are free i.e. not allocated. This list is called AVAIL List. Similarly, whenever a node is deleted, the deleted space becomes reusable and is added to the list of unused space i.e. to AVAIL List. This unused space can be used in future for memory allocation.

Memory allocation is of two types-

1. Static Memory Allocation
2. Dynamic Memory Allocation

**1. Static Memory Allocation:**

When memory is allocated during compilation time, it is called 'Static Memory Allocation'. This memory is fixed and cannot be increased or decreased after allocation. If more memory is allocated than requirement, then memory is wasted. If less memory is allocated than requirement, then program will not run successfully. So exact memory requirements must be known in advance.

**2. Dynamic Memory Allocation:**

When memory is allocated during run/execution time, it is called 'Dynamic Memory Allocation'. This memory is not fixed and is allocated according to our requirements. Thus in it there is no wastage of memory. So there is no need to know exact memory requirements in advance.

**Garbage Collection-**

Whenever a node is deleted, some memory space becomes reusable. This memory space should be available for future use. One way to do this is to immediately insert the free space into availability list. But this method may be time consuming for the operating system. So another method is used which is called 'Garbage Collection'. This method is described below: In this method the OS collects the deleted space time to time onto the availability list. This process happens in two steps. In first step, the OS goes through all the lists and tags all those cells which are currently being used. In the second step, the

OS goes through all the lists again and collects untagged space and adds this collected space to availability list. The garbage collection may occur when small amount of free space is left in the system or no free space is left in the system or when CPU is idle and has time to do the garbage collection.

### **Compaction**

One preferable solution to garbage collection is compaction. The process of moving all marked nodes to one end of memory and all available memory to other end is called compaction. Algorithm which performs compaction is called compacting algorithm.

## Lecture-12

### **Infix to Postfix Conversion**

```
1  #include<stdio.h>
2  char stack[20];
3  int top = -1;
4  void push(char x)
5  {
6      stack[++top] = x;
7  }
8
9  char pop()
10 {
11     if(top == -1)
12         return -1;
13     else
14         return stack[top--];
15 }
16
17 int priority(char x)
18 {
19     if(x == '(')
20         return 0;
21     if(x == '+' || x == '-')
22         return 1;
23     if(x == '*' || x == '/')
24         return 2;
25 }
26
27 main()
28 {
29     char exp[20];
30     char *e, x;
31     printf("Enter the expression :: ");
32     scanf("%s",exp);
33     e = exp;
34     while(*e != '\0')
35     {
36         if(isalnum(*e))
37             printf("%c",*e);
38         else if(*e == '(')
39             push(*e);
40         else if(*e == ')')
41             {
```

```

42         while((x = pop()) != '(')
43             printf("%c", x);
44     }
45     else
46     {
47         while(priority(stack[top]) >= priority(*e))
48             printf("%c",pop());
49         push(*e);
50     }
51     e++;
52 }
53 while(top != -1)
54 {
55     printf("%c",pop());
56 }
57 }

```

OUTPUT:

Enter the expression :: a+b\*c  
abc\*+

Enter the expression :: (a+b)\*c+(d-a)  
ab+c\*da-+

## Evaluate POSTFIX Expression Using Stack

```
1  #include<stdio.h>
2  int stack[20];
3  int top = -1;
4  void push(int x)
5  {
6      stack[++top] = x;
7  }
8
9  int pop()
10 {
11     return stack[top--];
12 }
13
14 int main()
15 {
16     char exp[20];
17     char *e;
18     int n1,n2,n3,num;
19     printf("Enter the expression :: ");
20     scanf("%s",exp);
21     e = exp;
22     while(*e != '\0')
23     {
24         if(isdigit(*e))
```

```
25     {
26         num = *e - 48;
27         push(num);
28     }
29     else
30     {
31         n1 = pop();
32         n2 = pop();
33         switch(*e)
34         {
35             case '+':
36                 {
37                     n3 = n1 + n2;
38                 break;
39                 }
40             case '-':
41                 {
42                     n3 = n2 - n1;
43                 break;
44                 }
45             case '*':
46                 {
47                     n3 = n1 * n2;
48                 break;
49                 }
```

```
50         case '/':
51         {
52             n3 = n2 / n1;
53             break;
54         }
55     }
56     push(n3);
57 }
58 e++;
59 }
60 printf("\nThe result of expression %s = %d\n\n",exp,pop());
61 return 0;
62
63 }
64
```

**Output:**

Enter the expression :: 245+\*

The result of expression 245+\* = 18

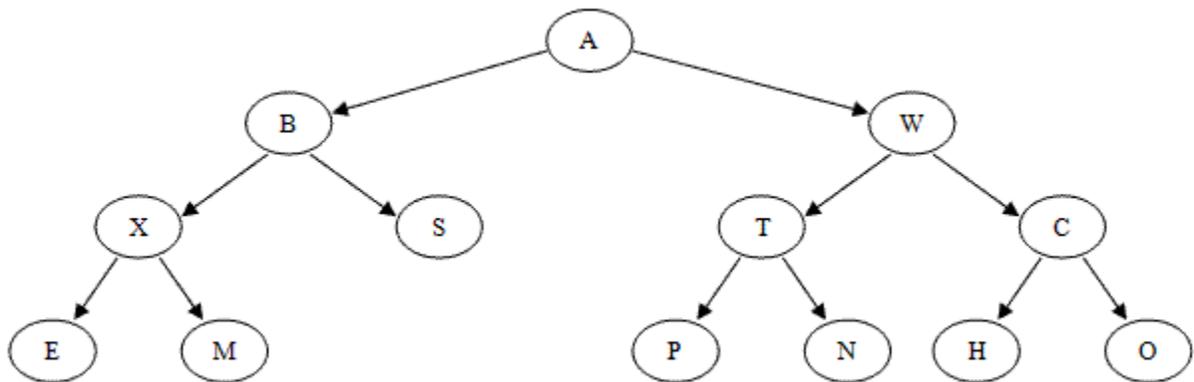
## Lecture-13

### Binary Tree

A *binary tree* consists of a finite set of nodes that is either empty, or consists of one specially designated node called the *root* of the binary tree, and the elements of two disjoint binary trees called the *left subtree* and *right subtree* of the root.

Note that the definition above is recursive: we have defined a binary tree in terms of binary trees. This is appropriate since recursion is an innate characteristic of tree structures.

**Diagram 1: A binary tree**



### Binary Tree Terminology

Tree terminology is generally derived from the terminology of family trees (specifically, the type of family tree called a *lineal chart*).

- Each root is said to be the *parent* of the roots of its subtrees.
- Two nodes with the same parent are said to be *siblings*; they are the *children* of their parent.
- The root node has no parent.
- A great deal of tree processing takes advantage of the relationship between a parent and its children, and we commonly say a *directed edge* (or simply an *edge*) extends from a parent to its children. Thus edges connect a root with the roots of each subtree. An *undirected edge* extends in both directions between a parent and a child.

- *Grandparent* and *grandchild* relations can be defined in a similar manner; we could also extend this terminology further if we wished (designating nodes as cousins, as an uncle or aunt, etc.).

## Other Tree Terms

- The number of subtrees of a node is called the *degree* of the node. In a binary tree, all nodes have degree 0, 1, or 2.
- A node of degree zero is called a *terminal node* or *leaf node*.
- A non-leaf node is often called a *branch node*.
- The *degree of a tree* is the maximum degree of a node in the tree. A binary tree is degree 2.
- A *directed path* from node  $n_1$  to  $n_k$  is defined as a sequence of nodes  $n_1, n_2, \dots, n_k$  such that  $n_i$  is the parent of  $n_{i+1}$  for  $1 \leq i < k$ . An *undirected path* is a similar sequence of undirected edges. The length of this path is the number of edges on the path, namely  $k - 1$  (i.e., the number of nodes  $- 1$ ). There is a path of length zero from every node to itself. Notice that in a binary tree there is exactly one path from the root to each node.
- The *level* or *depth* of a node with respect to a tree is defined recursively: the level of the root is zero; and the level of any other node is one higher than that of its parent. Or to put it another way, the level or depth of a node  $n_i$  is the length of the unique path from the root to  $n_i$ .
- The *height* of  $n_i$  is the length of the longest path from  $n_i$  to a leaf. Thus all leaves in the tree are at height 0.
- The *height of a tree* is equal to the height of the root. The *depth of a tree* is equal to the level or depth of the deepest leaf; this is always equal to the height of the tree.
- If there is a directed path from  $n_1$  to  $n_2$ , then  $n_1$  is an ancestor of  $n_2$  and  $n_2$  is a descendant of  $n_1$ .

## Lecture-14

### Special Forms of Binary Trees

There are a few special forms of binary tree worth mentioning.

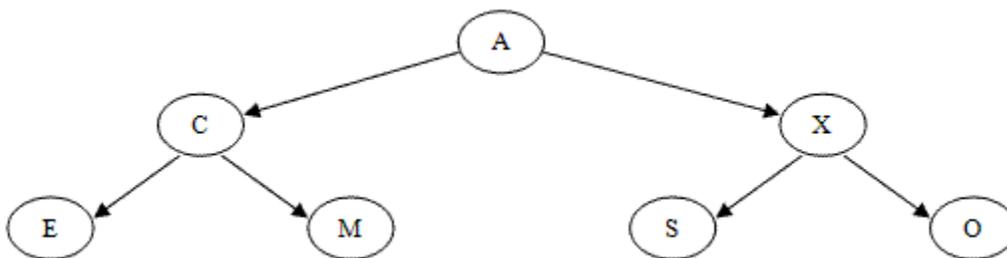
If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a *strictly binary tree*. Or, to put it another way, all of the nodes in a strictly binary tree are of degree zero or two, never degree one. A strictly binary tree with  $N$  leaves always contains  $2N - 1$  nodes.

Some texts call this a "full" binary tree.

A *complete binary tree* of depth  $d$  is the strictly binary tree all of whose leaves are at level  $d$ .

The total number of nodes in a complete binary tree of depth  $d$  equals  $2^{d+1} - 1$ . Since all leaves in such a tree are at level  $d$ , the tree contains  $2^d$  leaves and, therefore,  $2^d - 1$  internal nodes.

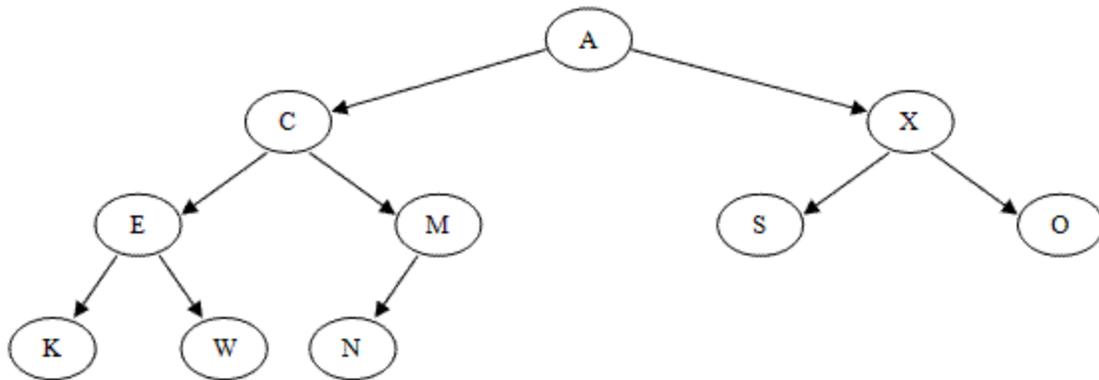
#### Diagram 2: A complete binary tree



A binary tree of depth  $d$  is an *almost complete binary tree* if:

- Each leaf in the tree is either at level  $d$  or at level  $d - 1$ .
- For any node  $n_d$  in the tree with a right descendant at level  $d$ , all the left descendants of  $n_d$  that are leaves are also at level  $d$ .

#### Diagram 3: An almost complete binary tree



An almost complete strictly binary tree with  $N$  leaves has  $2N - 1$  nodes (as does any other strictly binary tree). An almost complete binary tree with  $N$  leaves that is not strictly binary has  $2N$  nodes. There are two distinct almost complete binary trees with  $N$  leaves, one of which is strictly binary and one of which is not.

There is only a single almost complete binary tree with  $N$  nodes. This tree is strictly binary if and only if  $N$  is odd.

## Representing Binary Trees in Memory

### Array Representation

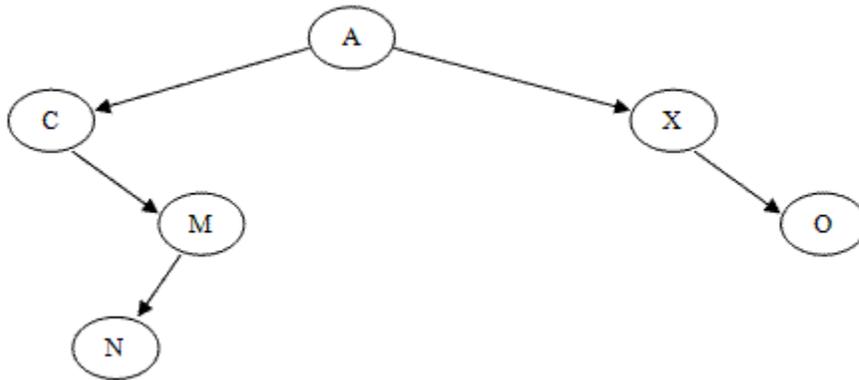
For a complete or almost complete binary tree, storing the binary tree as an array may be a good choice.

One way to do this is to store the root of the tree in the first element of the array. Then, for each node in the tree that is stored at subscript  $k$ , the node's left child can be stored at subscript  $2k+1$  and the right child can be stored at subscript  $2k+2$ . For example, the almost complete binary tree shown in **Diagram 2** can be stored in an array like so:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| A   | C   | X   | E   | M   | S   | O   | K   | W   | N   |

However, if this scheme is used to store a binary tree that is not complete or almost complete, we can end up with a great deal of wasted space in the array.

For example, the following binary tree

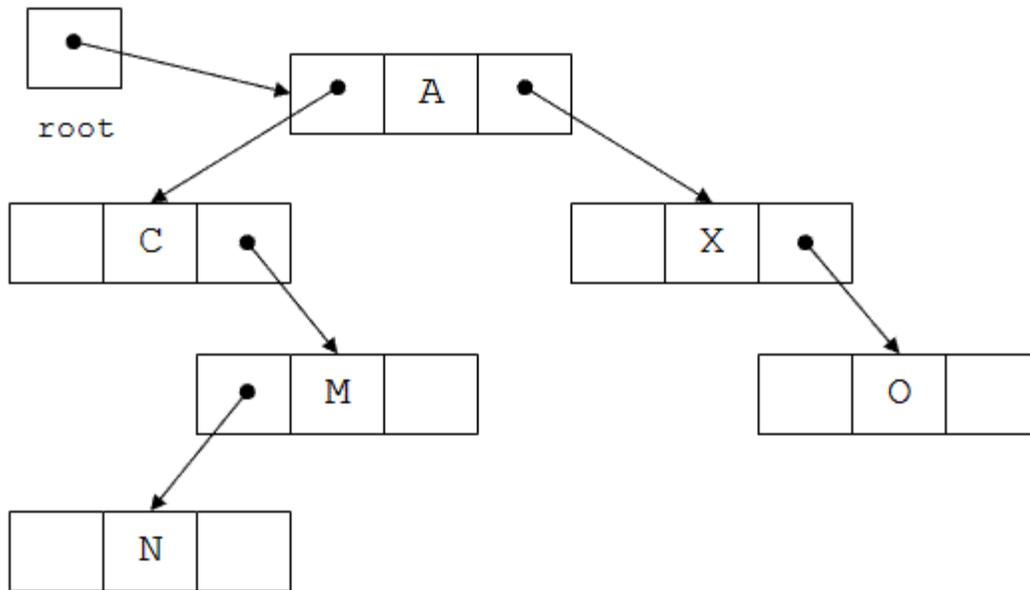


would be stored using this technique like so:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| A   | C   | X   |     | M   |     | O   |     |     | N   |

### ***Linked Representation***

If a binary tree is not complete or almost complete, a better choice for storing it is to use a linked representation similar to the linked list structures covered earlier in the semester:



Each tree node has two pointers (usually named left and right). The tree class has a pointer to the root node of the tree (labeled root in the diagram above).

Any pointer in the tree structure that does not point to a node will normally contain the value NULL. A linked tree with  $N$  nodes will always contain  $N + 1$  null links.

## Lecture-15

### Tree Traversal:

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) 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 –

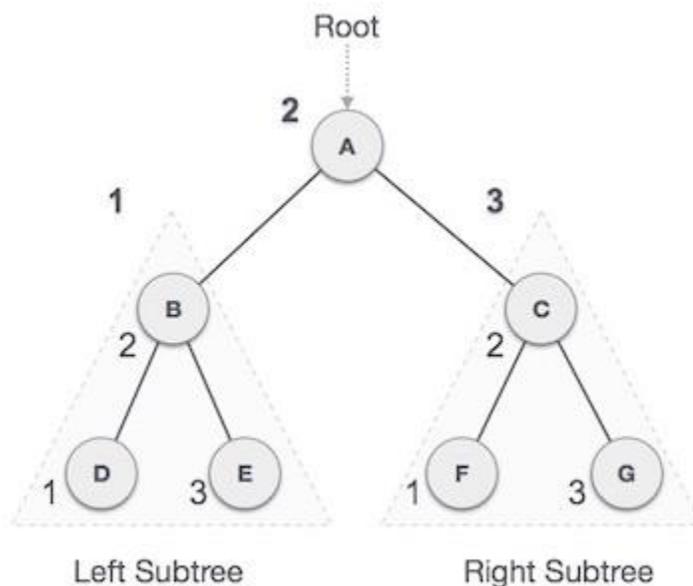
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

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.

### In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

**$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$**

## Algorithm

Until all nodes are traversed –

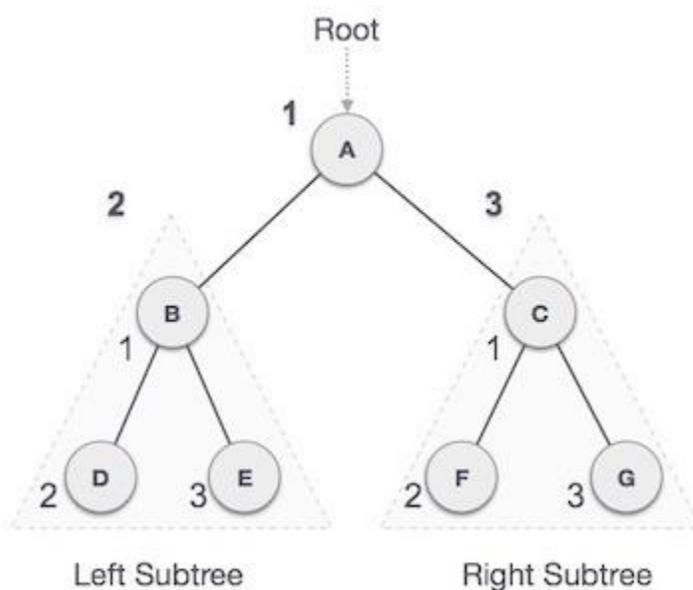
**Step 1** – Recursively traverse left subtree.

**Step 2** – Visit root node.

**Step 3** – Recursively traverse right subtree.

### Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

**A → B → D → E → C → F → G**

## Algorithm

Until all nodes are traversed –

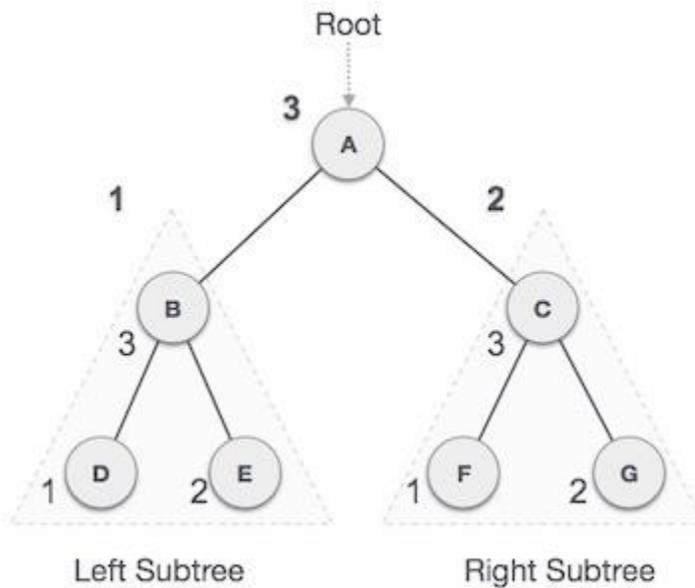
**Step 1** – Visit root node.

**Step 2** – Recursively traverse left subtree.

**Step 3** – Recursively traverse right subtree.

## Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

**$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$**

### Algorithm

Until all nodes are traversed –

**Step 1** – Recursively traverse left subtree.

**Step 2** – Recursively traverse right subtree.

**Step 3** – Visit root node.

## Lecture-16

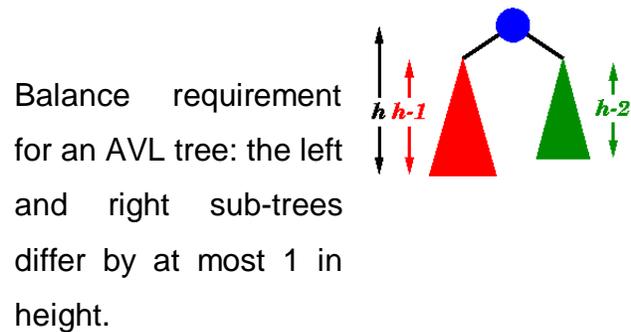
### AVL Trees

An **AVL tree** is another balanced binary search tree. Named after their inventors, **Adelson-Velskii** and **Landis**, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an  $O(\log n)$  search time. Addition and deletion operations also take  $O(\log n)$  time.

#### Definition of an AVL tree

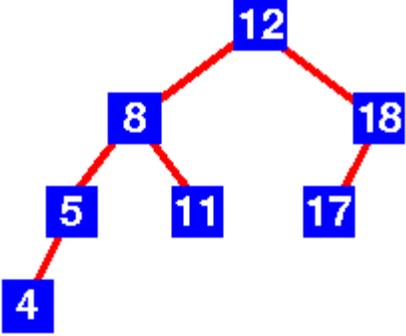
An AVL tree is a binary search tree which has the following properties:

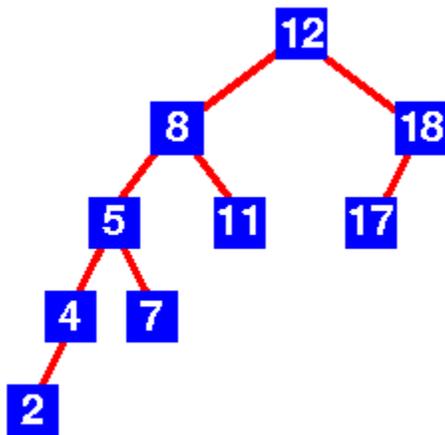
1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.



You need to be careful with this definition: it permits some apparently unbalanced trees!

For example, here are some trees:

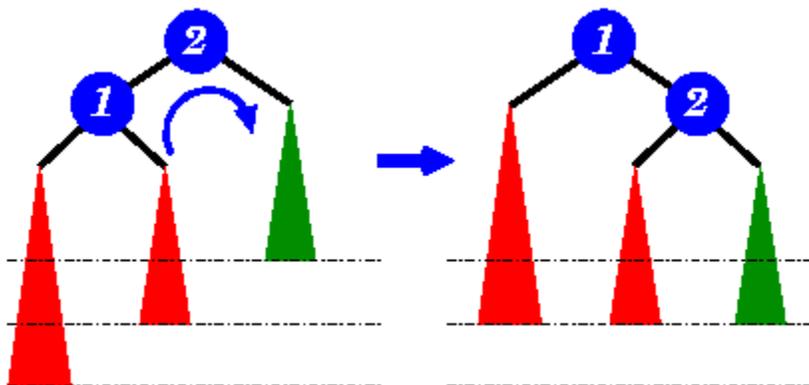
| Tree  | AVL tree?   |
|---|---|
|  | Yes<br>Examination shows that each left sub-tree has a height 1 greater than each right sub-tree. |



No  
 Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2

### Insertion

As with the red-black tree, insertion is somewhat complex and involves a number of cases. Implementations of AVL tree insertion may be found in many textbooks: they rely on adding an extra attribute, the **balance factor** to each node. This factor indicates whether the tree is *left-heavy* (the height of the left sub-tree is 1 greater than the right sub-tree), *balanced* (both sub-trees are the same height) or *right-heavy* (the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.



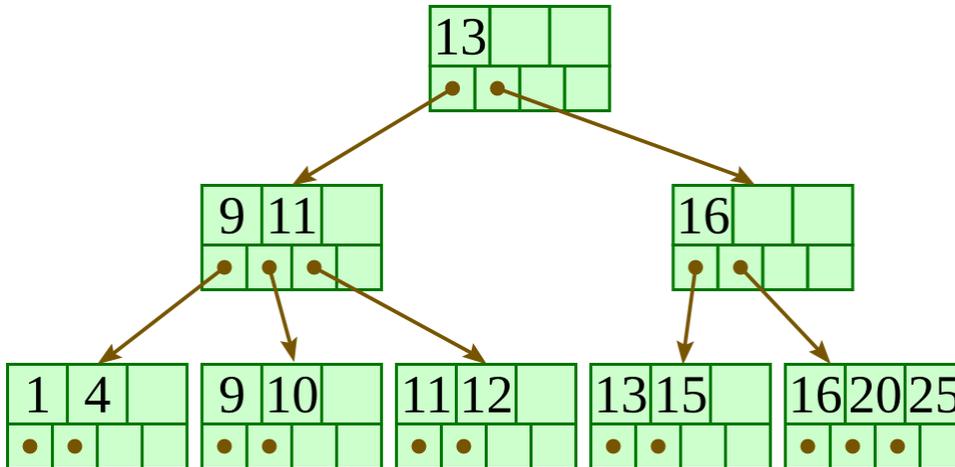
A new item has been added to the left subtree of node 1, causing its height to become 2 greater than 2's right subtree (shown in green). A right-rotation is performed to correct the imbalance.

## Lecture-17

### B+-tree

In B+-tree, each node stores up to  $d$  references to children and up to  $d - 1$  keys. Each reference is considered “between” two of the node's keys; it references the root of a subtree for which all values are between these two keys.

Here is a fairly small tree using 4 as our value for  $d$ .



A B+-tree requires that each leaf be the same distance from the root, as in this picture, where searching for any of the 11 values (all listed on the bottom level) will involve loading three nodes from the disk (the root block, a second-level block, and a leaf).

In practice,  $d$  will be larger — as large, in fact, as it takes to fill a disk block. Suppose a block is 4KB, our keys are 4-byte integers, and each reference is a 6-byte file offset. Then we'd choose  $d$  to be the largest value so that  $4(d - 1) + 6d \leq 4096$ ; solving this inequality for  $d$ , we end up with  $d \leq 410$ , so we'd use 410 for  $d$ . As you can see,  $d$  can be large.

A B+-tree maintains the following invariants:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node  $N$  with  $k$  being the number of keys in  $N$ : all keys in the first child's subtree are less than  $N$ 's first key; and all keys in the  $i$ th child's subtree ( $2 \leq i \leq k$ ) are between the  $(i - 1)$ th key of  $n$  and the  $i$ th key of  $n$ .
- The root has at least two children.
- Every non-leaf, non-root node has at least  $\text{floor}(d / 2)$  children.

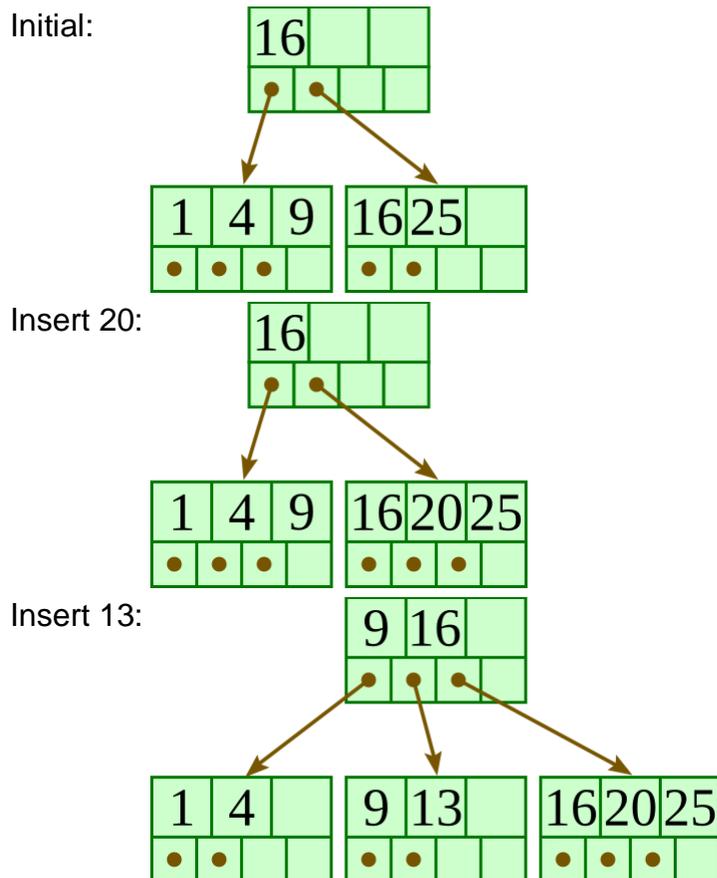
- Each leaf contains at least  $\text{floor}(d / 2)$  keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.

In our examples, we'll continue to use 4 for  $d$ . Looking at our invariants, this requires that each leaf have at least two keys, and each internal node to have at least two children (and thus at least one key).

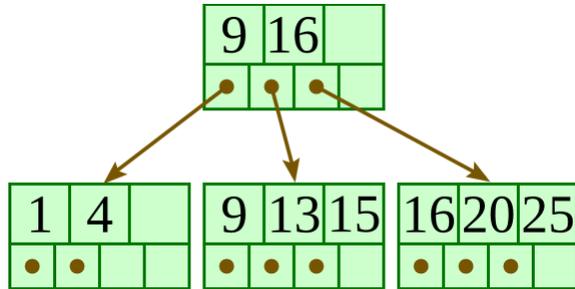
## 2. Insertion algorithm

Descend to the leaf where the key fits.

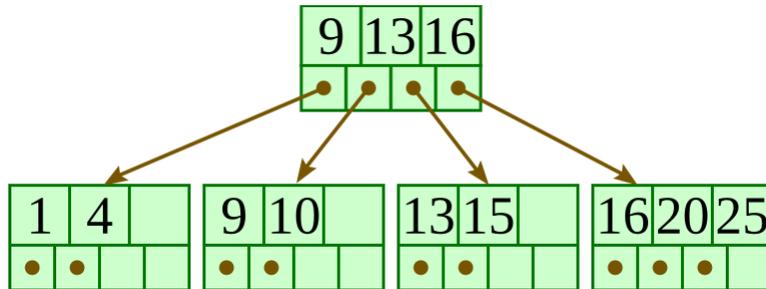
1. If the node has an empty space, insert the key/reference pair into the node.
2. If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. If the node is a leaf, take a copy of the minimum value in the second of these two nodes and repeat this insertion algorithm to insert it into the parent node. If the node is a non-leaf, exclude the middle value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.



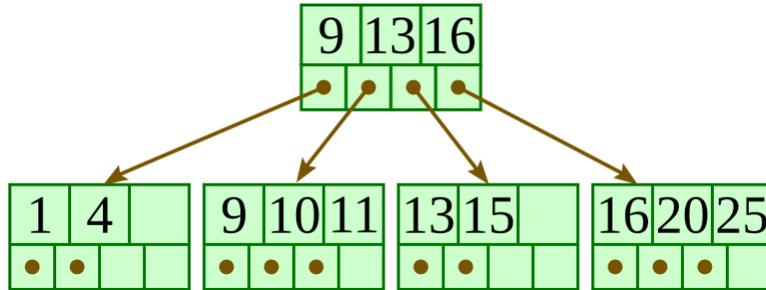
Insert 15:



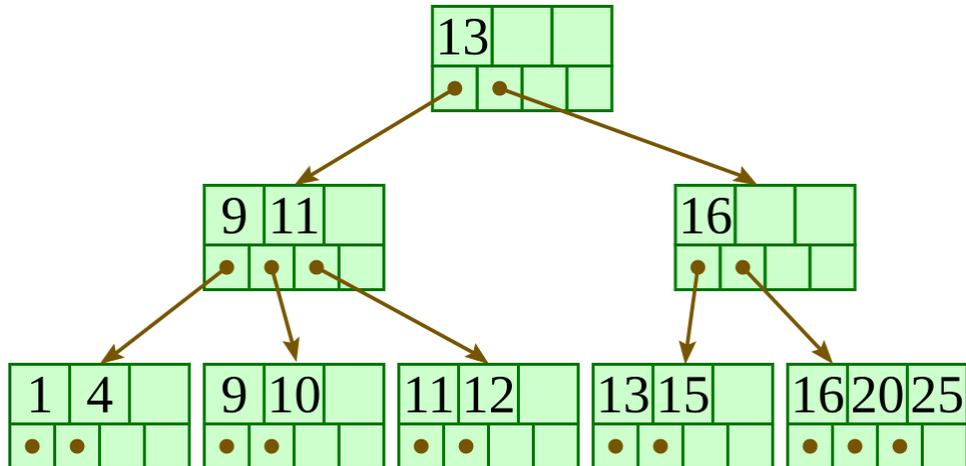
Insert 10:



Insert 11:



Insert 12:



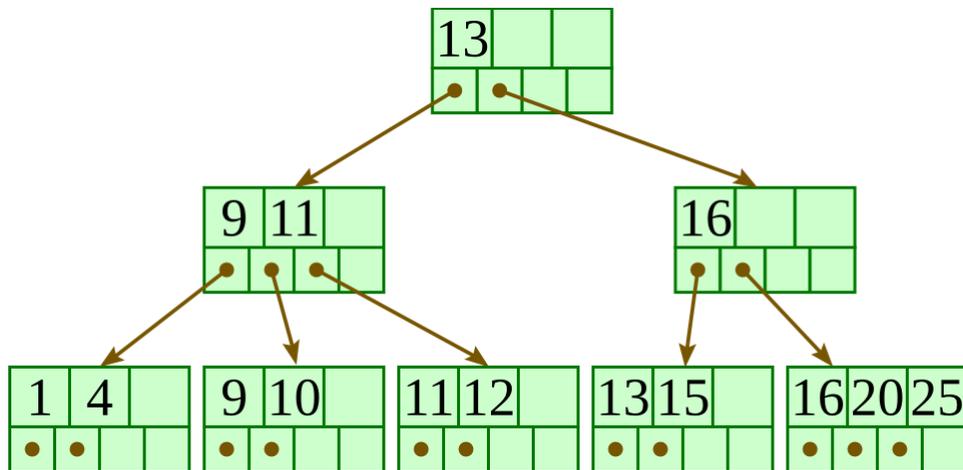
### 3. Deletion algorithm

Descend to the leaf where the key exists.

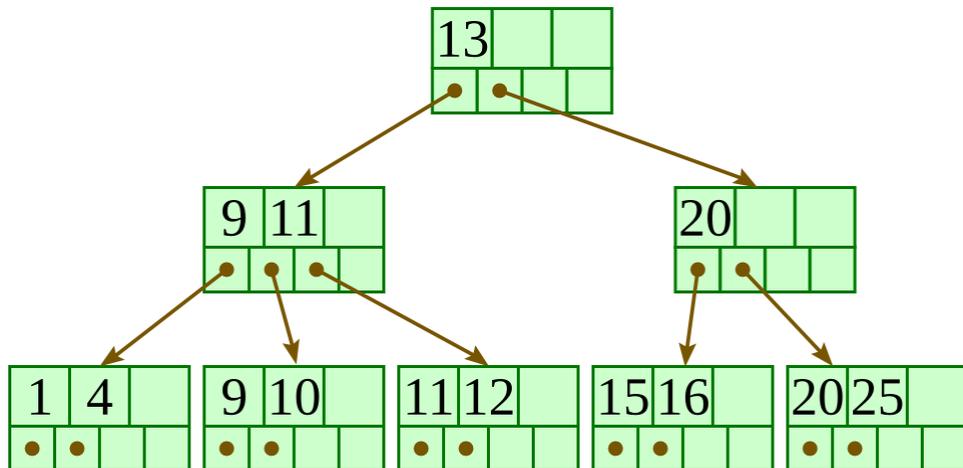
1. Remove the required key and associated reference from the node.
2. If the node still has enough keys and references to satisfy the invariants, stop.

3. If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.
4. If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

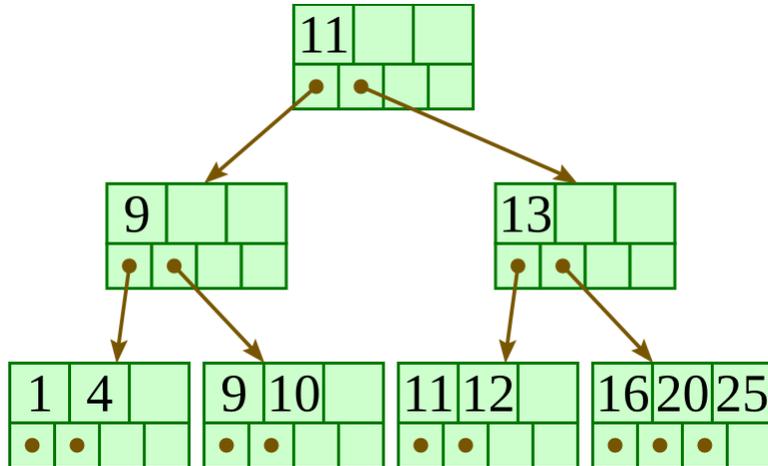
Initial:



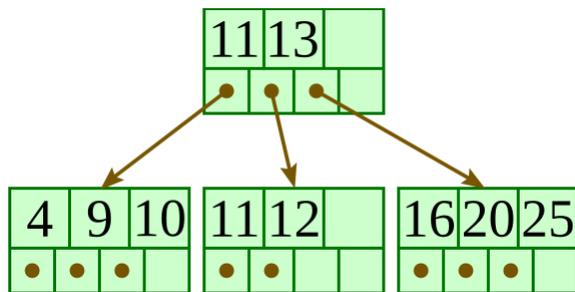
Delete 13:



Delete 15:

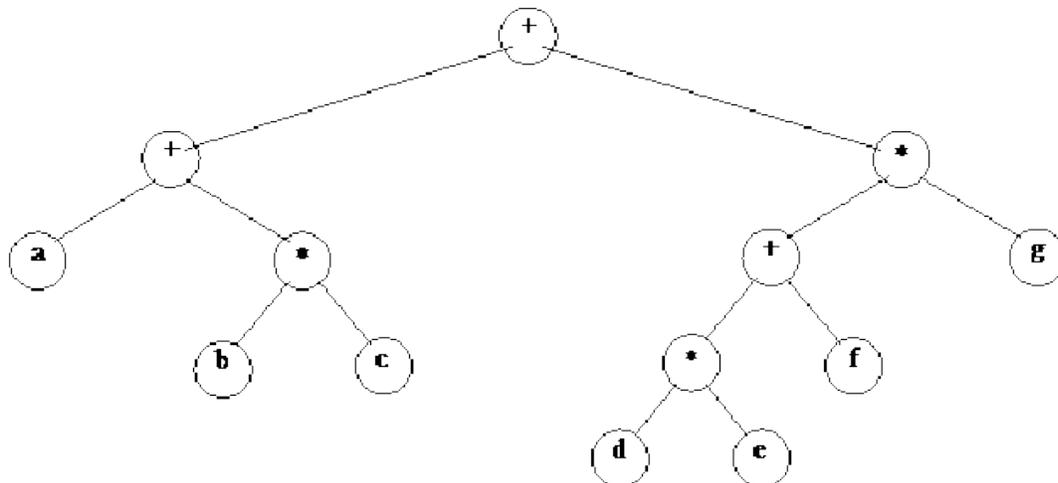


Delete 1:



### Expression Trees:

Trees are used in many other ways in the computer science. Compilers and database are two major examples in this regard. In case of compilers, when the languages are translated into machine language, tree-like structures are used. We have also seen an example of expression tree comprising the mathematical expression. Let's have more discussion on the expression trees. We will see what are the benefits of expression trees and how can we build an expression tree. Following is the figure of an expression tree.

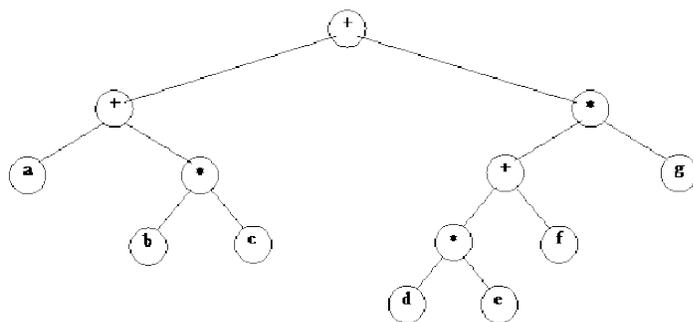


In the above tree, the expression on the left side is  $a + b * c$  while on the right side, we have  $d * e + f * g$ . If you look at the figure, it becomes evident that the inner nodes contain operators while leaf nodes have operands. We know that there are two types of nodes in the tree i.e. inner nodes and leaf nodes. The leaf nodes are such nodes which have left and right subtrees as null. You will find these at the bottom level of the tree. The leaf nodes are connected with the inner nodes. So in trees, we have some inner nodes and some leaf nodes.

In the above diagram, all the inner nodes (the nodes which have either left or right child or both) have operators. In this case, we have  $+$  or  $*$  as operators. Whereas leaf nodes contain operands only i.e.  $a, b, c, d, e, f, g$ . This tree is binary as the operators are binary. We have discussed the evaluation of postfix and infix expressions and have seen that the binary operators need two operands. In the infix expressions, one operand is on the left side of the operator and the other is on the right side. Suppose, if we have  $+$  operator, it will be written as  $2 + 4$ . However, in case of multiplication, we will write as  $5 * 6$ . We may have unary operators like negation ( $-$ ) or in Boolean expression we have NOT. In this example, there are all the binary operators. Therefore, this tree is a binary tree. This is not the Binary Search Tree. In BST, the values on the left side of the nodes are smaller and the values on the right side are greater than the node. Therefore, this is not a BST. Here we have an expression tree with no sorting process involved.

This is not necessary that expression tree is always binary tree. Suppose we have a unary operator like negation. In this case, we have a node which has ( $-$ ) in it and there is only one leaf node under it. It means just negate that operand.

Let's talk about the traversal of the expression tree. The inorder traversal may be executed here.



**Inorder traversal yields:  $a + b * c + d * e + f * g$**

## Lecture-18

### Binary Search Tree (BST)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

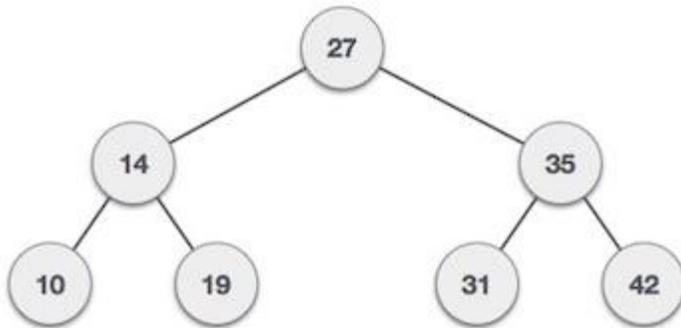
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

$$\text{left\_subtree (keys)} \leq \text{node (key)} \leq \text{right\_subtree (keys)}$$

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree –

- **Search** – Searches an element in a tree.
- **Insert** – Inserts an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {
    int data;
    struct node *leftChild;
    struct node *rightChild;
};
```

#### Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

#### Algorithm

```
struct node* search(int data){
    struct node *current = root;
    printf("Visiting elements: ");

    while(current->data != data){

        if(current != NULL) {
            printf("%d ",current->data);

            //go to left tree
            if(current->data > data){
                current = current->leftChild;
            } //else go to right tree
            else {
                current = current->rightChild;
            }

            //not found
```

```

    if(current == NULL){
        return NULL;
    }
}
}

return current;
}

```

### **Insert Operation**

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

### **Algorithm**

```

void insert(int data) {
    struct node *tempNode = (struct node*) malloc(sizeof(struct node));
    struct node *current;
    struct node *parent;

    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;

    //if tree is empty
    if(root == NULL) {
        root = tempNode;
    } else {
        current = root;
        parent = NULL;
    }
}

```

```
while(1) {
    parent = current;

    //go to left of the tree
    if(data < parent->data) {
        current = current->leftChild;
        //insert to the left

        if(current == NULL) {
            parent->leftChild = tempNode;
            return;
        }
    } //go to right of the tree
    else {
        current = current->rightChild;

        //insert to the right
        if(current == NULL) {
            parent->rightChild = tempNode;
            return;
        }
    }
}
}
```

**Module-3:**  
**Lecture-19**

**Graphs Terminology**

A **graph** consists of:

- A set,  $V$ , of **vertices** (nodes)
- A collection,  $E$ , of pairs of vertices from  $V$  called **edges** (arcs)

**Edges**, also called arcs, are represented by  $(u, v)$  and are either:

**Directed** if the pairs are ordered  $(u, v)$

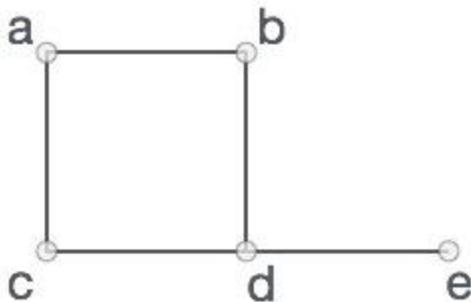
$u$  the **origin**

$v$  the **destination**

**Undirected** if the pairs are unordered

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets  $(V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

$V = \{a, b, c, d, e\}$

$E = \{ab, ac, bd, cd, de\}$

Then a **graph** can be:

**Directed graph (di-graph)** if all the edges are directed

**Undirected graph** (graph) if all the edges are undirected

**Mixed graph** if edges are both directed or undirected

**Illustrate terms on graphs**

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

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

An edge is **incident** on a vertex if the vertex is an endpoint of the edge.

**Outgoing edges** of a vertex are directed edges that the vertex is the origin.

**Incoming edges** of a vertex are directed edges that the vertex is the destination.

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

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

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

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

**Self-loop** is an edge with the end vertices the same vertex

**Simple graphs** have **no** parallel edges or self-loops

**Properties**

If graph,  $G$ , has  $m$  edges **then**  $\sum_{v \in G} \text{deg}(v) = 2m$

If a di-graph,  $G$ , has  $m$  edges **then**

$\sum_{v \in G} \text{indeg}(v) = m = \sum_{v \in G} \text{outdeg}(v)$

If a **simple** graph,  $G$ , has  $m$  edges and  $n$  vertices:

If  $G$  is also directed **then**  $m \leq n(n-1)$

If  $G$  is also undirected **then**  $m \leq n(n-1)/2$

So a simple graph with  $n$  vertices has  $O(n^2)$  edges at most

**More Terminology**

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

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

**Simple path** is a path with distinct vertices.

**Directed path** is a path of only directed edges

**Directed cycle** is a cycle of only directed edges.

**Sub-graph** is a subset of vertices and edges.

**Spanning sub-graph** contains all the vertices.

**Connected graph** has all pairs of vertices connected by at least one path.

**Connected component** is the maximal connected sub-graph of a disconnected graph.

**Forest** is a graph without cycles.

**Tree** is a connected forest (previous type of trees are called rooted trees, these are free trees)

**Spanning tree** is a spanning subgraph that is also a tree.

**More Properties**

If  $G$  is an **undirected** graph with  $n$  vertices and  $m$  edges:

- If  $G$  is connected **then**  $m \geq n - 1$
- If  $G$  is a tree **then**  $m = n - 1$
- If  $G$  is a forest **then**  $m \leq n - 1$

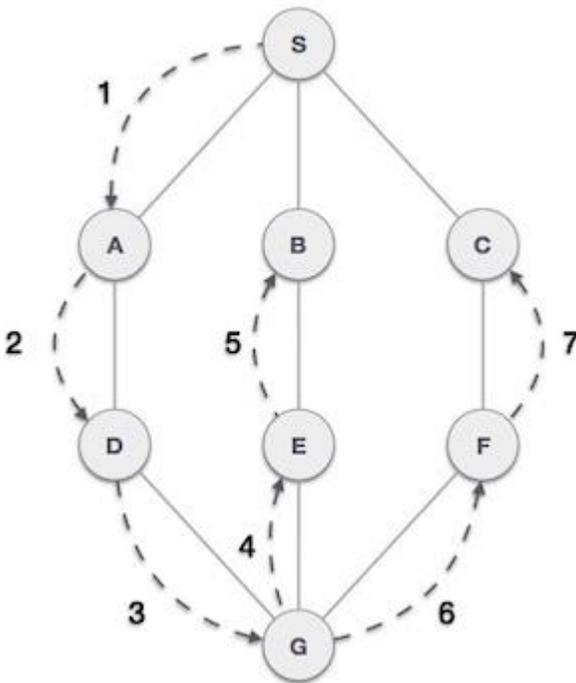
**Graph Traversal:**

1. Depth First Search
2. Breadth First Search

## Lecture-20

### Depth First Search:

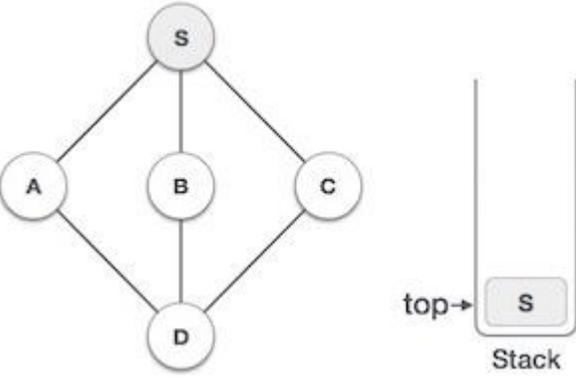
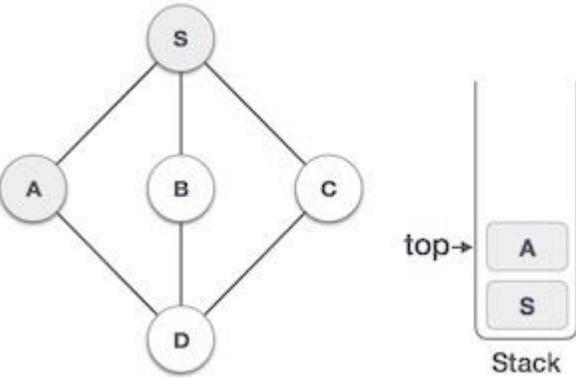
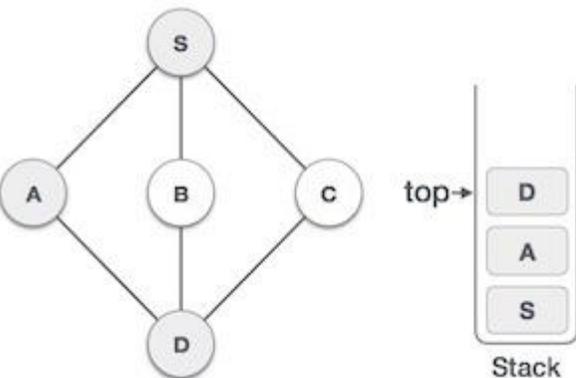
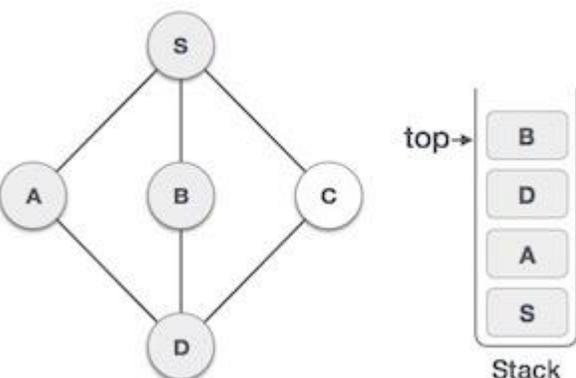
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

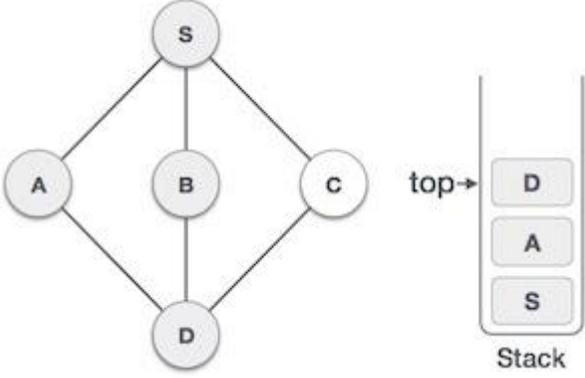
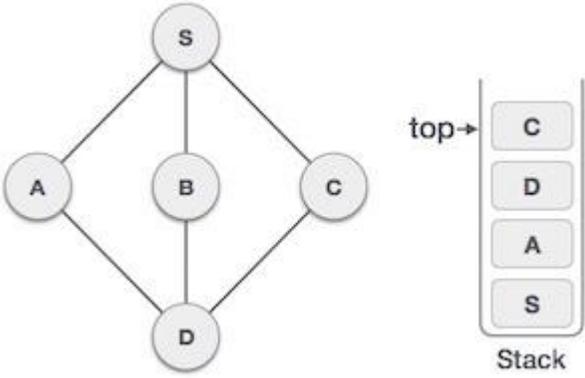


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

| Step | Traversal | Description           |
|------|-----------|-----------------------|
| 1    |           | Initialize the stack. |

|   |   |   |
|---|---|---|
| 2 |    | <p>Mark <b>S</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>S</b>. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.</p> |
| 3 |   | <p>Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>A</b>. Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.</p>                  |
| 4 |  | <p>Visit <b>D</b> and mark it as visited and put onto the stack. Here, we have <b>B</b> and <b>C</b> nodes, which are adjacent to <b>D</b> and both are unvisited. However, we shall again choose in an alphabetical order.</p>   |
| 5 |  | <p>We choose <b>B</b>, mark it as visited and put onto the stack. Here <b>B</b> does not have any unvisited adjacent node. So, we pop <b>B</b> from the stack.</p>  |

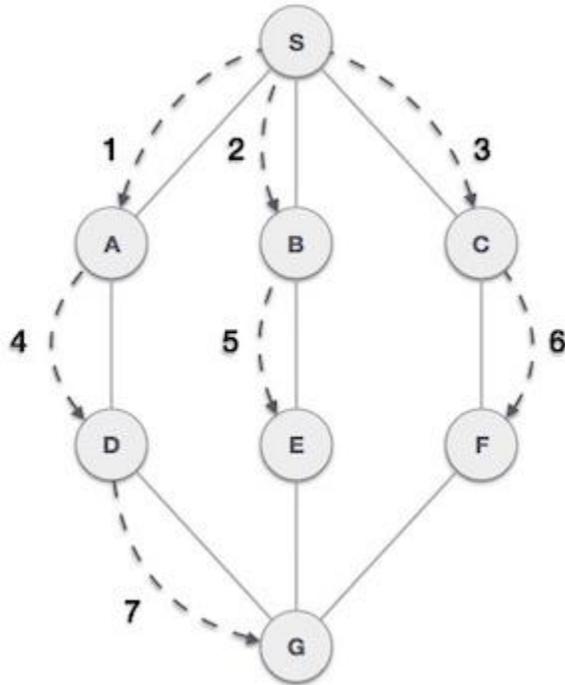
|   |  |  |
|---|--|--|
| 6 |   | <p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of the stack.</p> |
| 7 |  | <p>Only unvisited adjacent node is from <b>D</b> is <b>C</b> now. So we visit <b>C</b>, mark it as visited and put it onto the stack.</p>                    |

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

## Lecture-21

### Breadth First Search

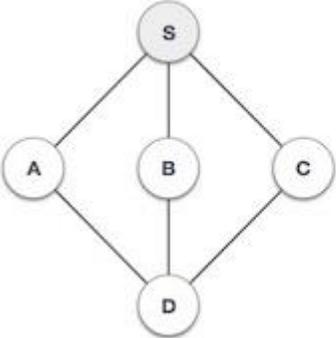
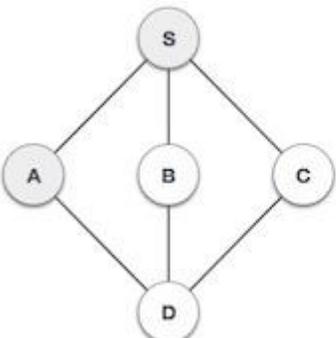
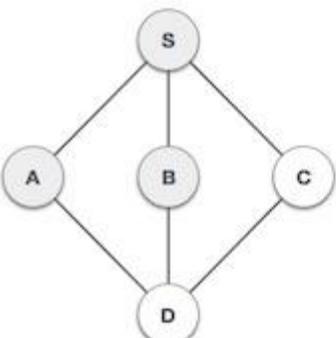
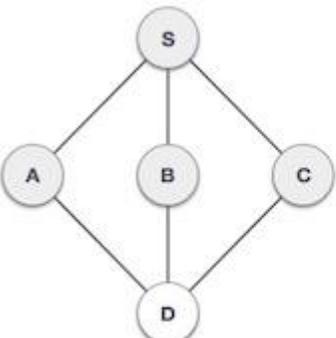
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

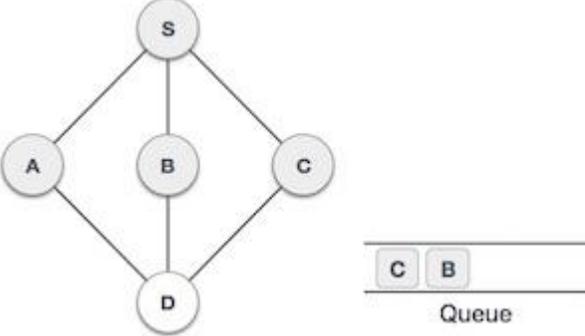
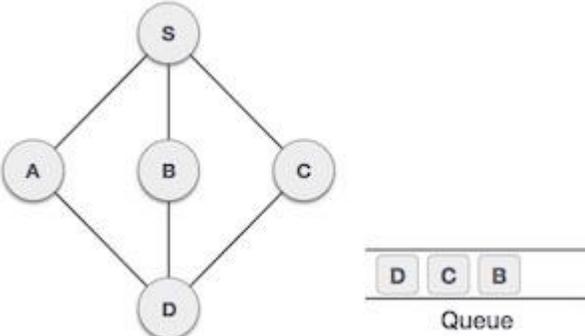


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

| Step | Traversal | Description           |
|------|-----------|-----------------------|
| 1    |           | Initialize the queue. |

|   |  |   |
|---|--|---|
| 2 |  <div style="text-align: right; margin-top: 10px;"> <hr style="width: 100px; border: 0.5px solid black;"/> <hr style="width: 100px; border: 0.5px solid black;"/> <p style="text-align: center;">Queue</p> </div>   | <p>We start from visiting <b>S</b>(starting node), and mark it as visited.</p>  |
| 3 |  <div style="text-align: right; margin-top: 10px;"> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-right: 5px;">A</div> <hr style="width: 100px; border: 0.5px solid black;"/> <hr style="width: 100px; border: 0.5px solid black;"/> <p style="text-align: center;">Queue</p> </div>   | <p>We then see an unvisited adjacent node from <b>S</b>. In this example, we have three nodes but alphabetically we choose <b>A</b>, mark it as visited and enqueue it.</p> |
| 4 |  <div style="text-align: right; margin-top: 10px;"> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-right: 5px;">B</div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-right: 5px;">A</div> <hr style="width: 100px; border: 0.5px solid black;"/> <hr style="width: 100px; border: 0.5px solid black;"/> <p style="text-align: center;">Queue</p> </div>  | <p>Next, the unvisited adjacent node from <b>S</b> is <b>B</b>. We mark it as visited and enqueue it.</p>   |
| 5 |  <div style="text-align: right; margin-top: 10px;"> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-right: 5px;">C</div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-right: 5px;">B</div> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-right: 5px;">A</div> <hr style="width: 100px; border: 0.5px solid black;"/> <hr style="width: 100px; border: 0.5px solid black;"/> <p style="text-align: center;">Queue</p> </div> | <p>Next, the unvisited adjacent node from <b>S</b> is <b>C</b>. We mark it as visited and enqueue it.</p>   |

|   |   |   |
|---|---|---|
| 6 |  | <p>Now, <b>S</b> is left with no unvisited adjacent nodes. So, we dequeue and find <b>A</b>.</p>        |
| 7 |  | <p>From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it as visited and enqueue it.</p> |

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

## Lecture-22

### Graph representation

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

#### Adjacency matrix

An adjacency matrix is a  $V \times V$  binary matrix **A**. Element  $A_{i,j}$  is 1 if there is an edge from vertex  $i$  to vertex  $j$  else  $A_{i,j}$  is 0.

**Note:** A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

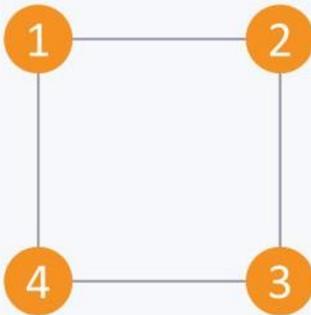
The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in  $A_{i,j}$ , the weight or cost of the edge will be stored.

In an undirected graph, if  $A_{i,j} = 1$ , then  $A_{j,i} = 1$ . In a directed graph, if  $A_{i,j} = 1$ , then  $A_{j,i}$  may or may not be 1.

Adjacency matrix provides **constant time access ( $O(1)$ )** to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is  $O(V^2)$ .

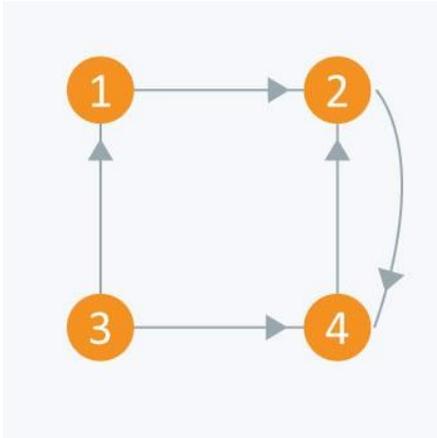
The adjacency matrix of the following graph is:

```
i/j : 1 2 3 4
1 : 0 1 0 1
2 : 1 0 1 0
3 : 0 1 0 1
4 : 1 0 1 0
```



The adjacency matrix of the following graph is:

```
i/j: 1 2 3 4
1 : 0 1 0 0
2 : 0 0 0 1
3 : 1 0 0 1
4 : 0 1 0 0
```



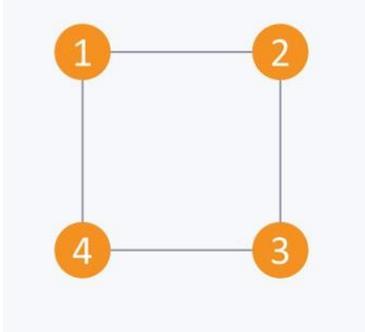
### Adjacency list

The other way to represent a graph is by using an adjacency list. An adjacency list is an array  $A$  of separate lists. Each element of the array  $A_i$  is a list, which contains all the vertices that are adjacent to vertex  $i$ .

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex  $j$  is in list  $A_i$  then vertex  $i$  will be in list  $A_j$ .

The space complexity of adjacency list is  $O(V + E)$  because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.

**Note:** A sparse matrix is a matrix in which most of the elements are zero, whereas a dense matrix is a matrix in which most of the elements are non-zero.



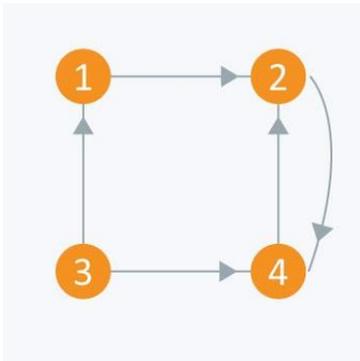
Consider the same undirected graph from an adjacency matrix. The adjacency list of the graph is as follows:

$A_1 \rightarrow 2 \rightarrow 4$

$A_2 \rightarrow 1 \rightarrow 3$

$A_3 \rightarrow 2 \rightarrow 4$

$A4 \rightarrow 1 \rightarrow 3$



Consider the same directed graph from an adjacency matrix. The adjacency list of the graph is as follows:

$A1 \rightarrow 2$

$A2 \rightarrow 4$

$A3 \rightarrow 1 \rightarrow 4$

$A4 \rightarrow 2$

## Lecture-23

### Topological Sorting:

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge  $uv$ , vertex  $u$  comes before  $v$  in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

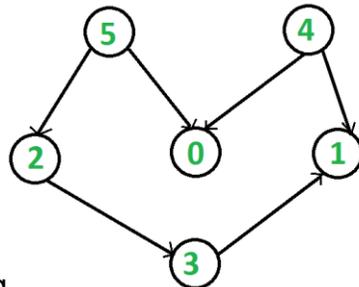
For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).

### Algorithm to find Topological Sorting:

In **DFS**, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

### Topological Sorting vs Depth First Traversal (DFS):

In **DFS**, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike **DFS**, the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the shown graph is "5 2 3 1 0 4", but it is not a



topological sorting

### Dynamic Programming

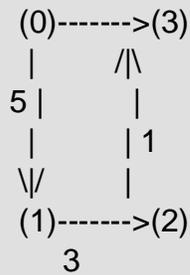
The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

#### Input:

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

which represents the following graph



Note that the value of  $\text{graph}[i][j]$  is 0 if  $i$  is equal to  $j$   
 And  $\text{graph}[i][j]$  is INF (infinite) if there is no edge from vertex  $i$  to  $j$ .

**Output:**

Shortest distance matrix

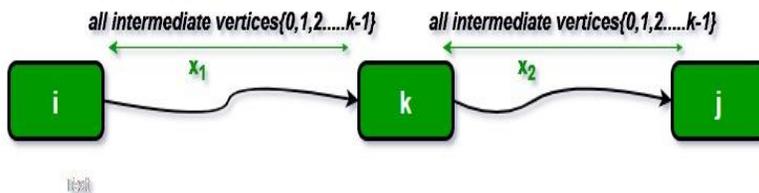
|     |     |     |   |
|-----|-----|-----|---|
| 0   | 5   | 8   | 9 |
| INF | 0   | 3   | 4 |
| INF | INF | 0   | 1 |
| INF | INF | INF | 0 |

**Floyd Warshall Algorithm**

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

- 1)  $k$  is not an intermediate vertex in shortest path from  $i$  to  $j$ . We keep the value of  $\text{dist}[i][j]$  as it is.
- 2)  $k$  is an intermediate vertex in shortest path from  $i$  to  $j$ . We update the value of  $\text{dist}[i][j]$  as  $\text{dist}[i][k] + \text{dist}[k][j]$ .

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



## Lecture-24

### Bubble Sort

We take an unsorted array for our example. Bubble sort takes  $O(n^2)$  time so we're keeping it short and precise.



Bubble sort starts with very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.



We know then that 10 is smaller 35. Hence they are not sorted.



We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –



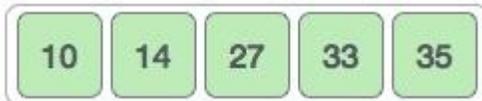
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

### Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```

begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort

```

### Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows –

```

procedure bubbleSort( list : array of items )

  loop = list.count;

  for i = 0 to loop-1 do:
    swapped = false

```

```
for j = 0 to loop-1 do:  
  
    /* compare the adjacent elements */  
    if list[j] > list[j+1] then  
        /* swap them */  
        swap( list[j], list[j+1] )  
        swapped = true  
    end if  
  
end for  
  
/*if no number was swapped that means  
array is sorted now, break the loop.*/  
  
if(not swapped) then  
    break  
end if  
  
end for  
  
end procedure return list
```

## Lecture-25

### Insertion Sort

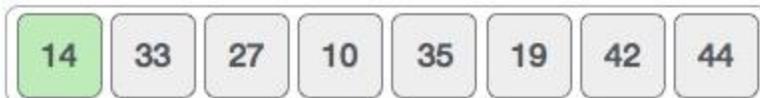
We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



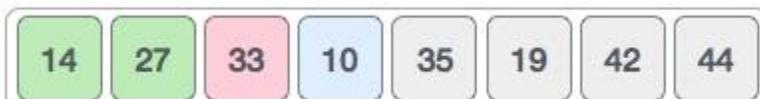
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

## Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

- Step 1** – If it is the first element, it is already sorted. return 1;
- Step 2** – Pick next element
- Step 3** – Compare with all elements in the sorted sub-list
- Step 4** – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
- Step 5** – Insert the value
- Step 6** – Repeat until list is sorted

## Pseudocode

```
procedure insertionSort( A : array of items )
  int holePosition
  int valueToInsert
  for i = 1 to length(A) inclusive do:
    valueToInsert = A[i]
    holePosition = i

    while holePosition > 0 and A[holePosition-1] > valueToInsert do:
      A[holePosition] = A[holePosition-1]
      holePosition = holePosition - 1
    end while
    A[holePosition] = valueToInsert
  end for
end procedure
```

## Lecture-26

### Selection Sort

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



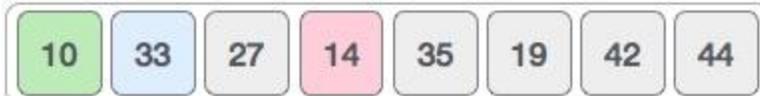
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

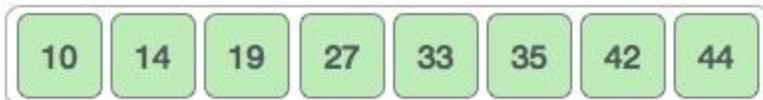
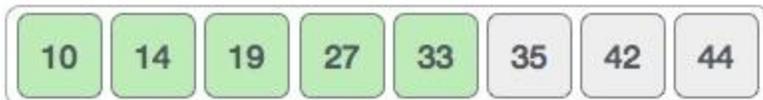
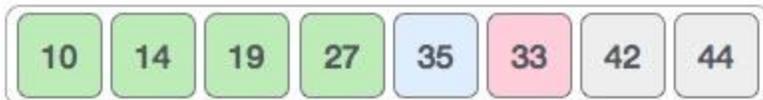
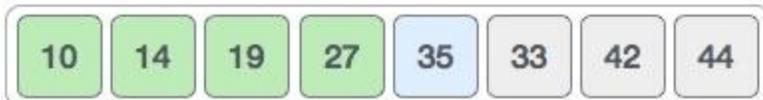


After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –



Now, let us learn some programming aspects of selection sort.

### Algorithm

- Step 1** – Set MIN to location 0
- Step 2** – Search the minimum element in the list
- Step 3** – Swap with value at location MIN
- Step 4** – Increment MIN to point to next element
- Step 5** – Repeat until list is sorted

## Pseudocode

```
procedure selection sort
  list : array of items
  n    : size of list

  for i = 1 to n - 1
    /* set current element as minimum*/
    min = i

    /* check the element to be minimum */

    for j = i+1 to n
      if list[j] < list[min] then
        min = j;
      end if
    end for

    /* swap the minimum element with the current element*/
    if indexMin != i then
      swap list[min] and list[i]
    end if
  end for

end procedure
```

## Lecture-27

### Merge Sort

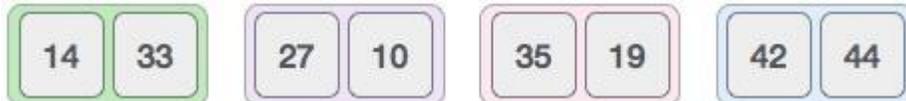
To understand merge sort, we take an unsorted array as the following –



We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

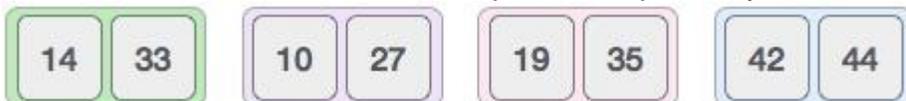


We further divide these arrays and we achieve atomic value which can no more be divided.



Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this –



Now we should learn some programming aspects of merge sorting.

### Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

**Step 1** – if it is only one element in the list it is already sorted, return.

**Step 2** – divide the list recursively into two halves until it can no more be divided.

**Step 3** – merge the smaller lists into new list in sorted order.

Merge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort( var a as array )
  if ( n == 1 ) return a
  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]
  l1 = mergesort( l1 )
  l2 = mergesort( l2 )
  return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )
  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while

  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while
  return c
end procedure
```

## Lecture-28

### Quick sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of  $O(n^2)$ , where  $n$  is the number of items.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.

### Unsorted Array



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

- Step 1** – Choose the highest index value has pivot
- Step 2** – Take two variables to point left and right of the list excluding pivot
- Step 3** – left points to the low index
- Step 4** – right points to the high
- Step 5** – while value at left is less than pivot move right
- Step 6** – while value at right is greater than pivot move left
- Step 7** – if both step 5 and step 6 does not match swap left and right
- Step 8** – if  $left \geq right$ , the point where they met is new pivot

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as –

```
function partitionFunc(left, right, pivot)
    leftPointer = left
    rightPointer = right - 1

    while True do
        while A[++leftPointer] < pivot do
```

```

    //do-nothing
end while

while rightPointer > 0 && A[--rightPointer] > pivot do
    //do-nothing
end while

if leftPointer >= rightPointer
    break
else
    swap leftPointer,rightPointer
end if

end while

swap leftPointer,right
return leftPointer

end function

```

#### Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows –

- Step 1** – Make the right-most index value pivot
- Step 2** – partition the array using pivot value
- Step 3** – quicksort left partition recursively
- Step 4** – quicksort right partition recursively

#### Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm –

```

procedure quickSort(left, right)

if right-left <= 0
    return
else
    pivot = A[right]
    partition = partitionFunc(left, right, pivot)
    quickSort(left,partition-1)
    quickSort(partition+1,right)
end if
end procedure

```

## Lecture-29

### Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

### What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

### Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index  $I$ , the left child can be calculated by  $2 * I + 1$  and right child by  $2 * I + 2$  (assuming the indexing starts at 0).

### Heap Sort Algorithm for sorting in increasing order:

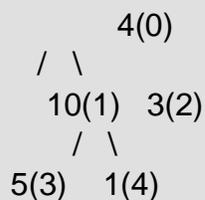
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

### How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

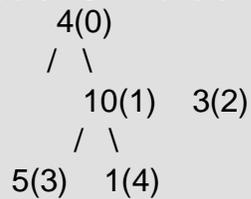
Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1



The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:



Applying heapify procedure to index 0:



The heapify procedure calls itself recursively to build heap in top down manner.

### Radix Sort

The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is  $\Omega(n \log n)$ , i.e., they cannot do better than  $n \log n$ .

Counting sort is a linear time sorting algorithm that sort in  $O(n+k)$  time when elements are in range from 1 to k.

***What if the elements are in range from 1 to  $n^2$ ?***

We can't use counting sort because counting sort will take  $O(n^2)$  which is worse than comparison based sorting algorithms. Can we sort such an array in linear time? Radix Sort is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

## Lecture-30

### Radix Sort

1) Do following for each digit  $i$  where  $i$  varies from least significant digit to the most significant digit.

.....**a)** Sort input array using counting sort (or any stable sort) according to the  $i$ 'th digit.

#### **Example:**

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [\*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [\*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

#### ***What is the running time of Radix Sort?***

Let there be  $d$  digits in input integers. Radix Sort takes  $O(d*(n+b))$  time where  $b$  is the base for representing numbers, for example, for decimal system,  $b$  is 10. What is the value of  $d$ ? If  $k$  is the maximum possible value, then  $d$  would be  $O(\log_b(k))$ . So overall time complexity is  $O((n+b) * \log_b(k))$ . Which looks more than the time complexity of comparison based sorting algorithms for a large  $k$ . Let us first limit  $k$ . Let  $k \leq n^c$  where  $c$  is a constant. In that case, the complexity becomes  $O(n \log_b(n))$ . But it still doesn't beat comparison based sorting algorithms.

### Linear Search

Linear search is to check each element one by one in sequence. The following method `linearSearch()` searches a target in an array and returns the index of the target; if not found, it returns -1, which indicates an invalid index.

```
1  int linearSearch(int arr[], int target)
2  {
3      for (int i = 0; i < arr.length; i++)
4          {
5              if (arr[i] == target)
6                  return i;
7          }
8      return -1;
9  }
```

Linear search loops through each element in the array; each loop body takes constant time. Therefore, it runs in linear time  $O(n)$ .

## Lecture-31

### Binary Search

For sorted arrays, *binary search* is more efficient than linear search. The process starts from the middle of the input array:

- If the target equals the element in the middle, return its index.
- If the target is larger than the element in the middle, search the right half.
- If the target is smaller, search the left half.

In the following `binarySearch()` method, the two index variables `first` and `last` indicates the searching boundary at each round.

```
1  int binarySearch(int arr[], int target)
2  {
3      int first = 0, last = arr.length - 1;
4
5      while (first <= last)
6      {
7          int mid = (first + last) / 2;
8          if (target == arr[mid])
9              return mid;
10         if (target > arr[mid])
11             first = mid + 1;
12         else
13             last = mid - 1;
14     }
15     return -1;
16 }
```

```
1  arr: {3, 9, 10, 27, 38, 43, 82}
```

```
2
```

```
3  target: 10
```

```
4  first: 0, last: 6, mid: 3, arr[mid]: 27 -- go left
```

```
5  first: 0, last: 2, mid: 1, arr[mid]: 9 -- go right
```

```
6  first: 2, last: 2, mid: 2, arr[mid]: 10 -- found
```

```
7
```

```
8  target: 40
```

```
9  first: 0, last: 6, mid: 3, arr[mid]: 27 -- go right
```

```
10 first: 4, last: 6, mid: 5, arr[mid]: 43 -- go left
```

```
11 first: 4, last: 4, mid: 4, arr[mid]: 38 -- go right
```

```
12 first: 5, last: 4 -- not found
```

Binary search divides the array in the middle at each round of the loop. Suppose the array has length  $n$  and the loop runs in  $t$  rounds, then we have  $n * (1/2)^t = 1$  since at each round the array length is divided by 2. Thus  $t = \log(n)$ . At each round, the loop body takes constant time. Therefore, binary search runs in logarithmic time  $O(\log n)$ .

The following code implements binary search using recursion. To call the method, we need provide with the boundary indexes, for example,  
`binarySearch(arr, 0, arr.length - 1, target);`

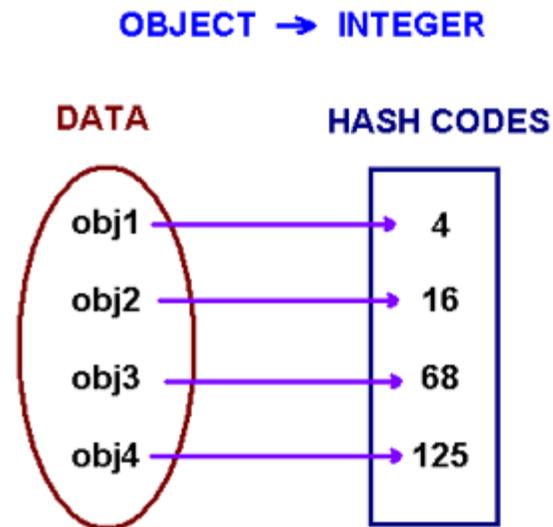
```
1
2  binarySearch(int arr[], int first, int last, int target)
3  {
4      if (first > last)
5          return -1;
6
7      int mid = (first + last) / 2;
8
9      if (target == arr[mid])
10         return mid;
11     if (target > arr[mid])
12         return binarySearch(arr, mid + 1, last, target);
13     // target < arr[mid]
14     return binarySearch(arr, first, mid - 1, target);
15 }
```

## Lecture-32

### Hashing

#### Introduction

The problem at hand is to speed up searching. Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worst-case runtime complexity to  $O(\log n)$ . We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have that magic function that would tell us the index for a given value. With this magic function our search is reduced to just one probe, giving us a constant runtime  $O(1)$ . Such a function is called a **hash function**. A hash function is a function which when given a key, generates an address in the table.



The example of a hash function is a *book call number*. Each book in the library has a *unique* call number. A call number is like an address: it tells us where the book is located in the library. Many academic libraries in the United States, use Library of Congress Classification for call numbers. This system uses a combination of letters and numbers to arrange materials by subjects.

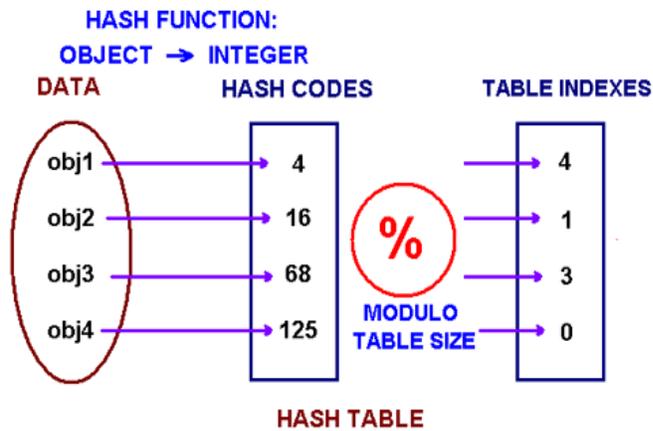
A hash function that returns a unique hash number is called a **universal hash function**. In practice it is extremely hard to assign unique numbers to objects. The latter is always possible only if you know (or approximate) the number of objects to be processed.

Thus, we say that our hash function has the following properties

- it always returns a number for an object.
- two equal objects will always have the same number
- two unequal objects not always have different numbers

The procedure of storing objects using a hash function is the following.

Create an array of size  $M$ . Choose a hash function  $h$ , that is a mapping from objects into integers  $0, 1, \dots, M-1$ . Put these objects into an array at indexes computed via the hash function  $index = h(object)$ . Such array is called a **hash table**.



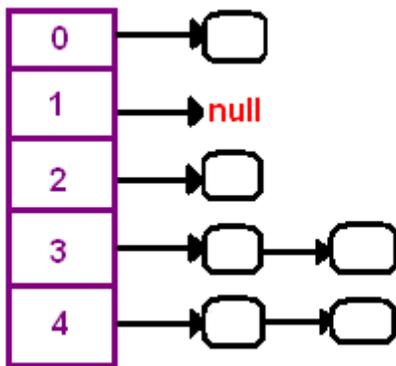
| OBJ4 | OBJ2 |   | OBJ3 | OBJ1 |
|------|------|---|------|------|
| 0    | 1    | 2 | 3    | 4    |

## Collisions

When we put objects into a hashtable, it is possible that different objects (by the *equals()* method) might have the same hashcode. This is called a **collision**. Here is the example of collision. Two different strings "Aa" and "BB" have the same key: .

$$\text{"Aa"} = \text{'A'} * 31 + \text{'a'} = 2112$$

$$\text{"BB"} = \text{'B'} * 31 + \text{'B'} = 2112$$



How to resolve collisions? Where do we put the second and subsequent values that hash to this same location? There are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list! A hash table then is an array of lists!! This technique is called a *separate chaining* collision resolution.

The big attraction of using a hash table is a constant-time performance for the basic operations add, remove, contains, size. Though, because of collisions, we cannot guarantee the constant runtime in the worst-case. Why? Imagine that all our objects collide into the same index. Then searching for one of them will be equivalent to searching in a list, that takes a liner runtime. However, we can guarantee an expected constant runtime, if we make sure that our lists won't become too long. This is usually implemented by maintaining a *load factor* that keeps a track of the average length of lists. If a load factor approaches a set in advanced threshold, we create a bigger array and *rehash* all elements from the old table into the new one.

Another technique of collision resolution is a *linear probing*. If we cannot insert at index  $k$ , we try the next slot  $k+1$ . If that one is occupied, we go to  $k+2$ , and so on.

## Lecture-33

### Hashing Functions

Choosing a good hashing function,  $h(\mathbf{k})$ , is essential for hash-table based searching.  $h$  should distribute the elements of our collection as uniformly as possible to the "slots" of the hash table. The key criterion is that there should be a minimum number of collisions.

If the probability that a key,  $\mathbf{k}$ , occurs in our collection is  $P(\mathbf{k})$ , then if there are  $m$  slots in our hash table, a *uniform hashing function*,  $h(\mathbf{k})$ , would ensure:

$$\sum_{\mathbf{k}|h(\mathbf{k})=0} P(\mathbf{k}) = \sum_{\mathbf{k}|h(\mathbf{k})=1} P(\mathbf{k}) = \dots = \sum_{\mathbf{k}|h(\mathbf{k})=m-1} P(\mathbf{k}) = \frac{1}{m}$$

Sometimes, this is easy to ensure. For example, if the keys are randomly distributed in  $(0,r]$ , then,

$$h(\mathbf{k}) = \text{floor}((m\mathbf{k})/r)$$

will provide uniform hashing.

#### **Mapping keys to natural numbers**

Most hashing functions will first map the keys to some set of natural numbers, say  $(0,r]$ . There are many ways to do this, for example if the key is a string of ASCII characters, we can simply add the ASCII representations of the characters mod 255 to produce a number in  $(0,255)$  - or we could **xor** them, or we could add them in pairs mod  $2^{16}-1$ , or ...

Having mapped the keys to a set of natural numbers, we then have a number of possibilities.

1. Use a **mod** function:

$$h(\mathbf{k}) = \mathbf{k} \bmod m.$$

When using this method, we usually avoid certain values of  $m$ . Powers of 2 are usually avoided, for  $\mathbf{k} \bmod 2^b$  simply selects the  $b$  low order bits of  $\mathbf{k}$ . Unless we know that all the  $2^b$  possible values of the lower order bits are equally likely, this will not be a good choice, because some bits of the key are not used in the hash function.

Prime numbers which are close to powers of 2 seem to be generally good choices for  $m$ .

For example, if we have 4000 elements, and we have chosen an overflow table organization, but wish to have the probability of collisions quite low, then we might choose  $m = 4093$ . (4093 is the largest prime less than  $4096 = 2^{12}$ .)

2. Use the multiplication method:
  - o Multiply the key by a constant  $A$ ,  $0 < A < 1$ ,
  - o Extract the fractional part of the product,
  - o Multiply this value by  $m$ .

Thus the hash function is:

$$h(\mathbf{k}) = \text{floor}(m * (\mathbf{k}A - \text{floor}(\mathbf{k}A)))$$

In this case, the value of  $m$  is not critical and we typically choose a power of 2 so that we can get the following efficient procedure on most digital computers:

- Choose  $m = 2^p$ .
- Multiply the  $w$  bits of  $k$  by  $\text{floor}(A * 2^w)$  to obtain a  $2w$  bit product.
- Extract the  $p$  most significant bits of the lower half of this product.

It seems that:

$$A = (\text{sqrt}(5)-1)/2 = 0.6180339887$$

is a good choice (see Knuth, "Sorting and Searching", v. 3 of "The Art of Computer Programming").

3. Use universal hashing:

A malicious adversary can always choose the keys so that they all hash to the same slot, leading to an average  $O(n)$  retrieval time. Universal hashing seeks to avoid this by choosing the hashing function randomly from a collection of hash functions (cf Cormen *et al*, p 229- ). This makes the probability that the hash function will generate poor behaviour small and produces good average performance.