





Data Structures

ডেটা স্ট্রাকচার শিখি বাংলাতে...

Fibonacci Heap PM-Index Hash Table Trie Splay Tree Beap
BST Tree Heap queue Soft Heap Stack Array Heap
Soft Heap Binary Heap Weak Heap Suffix Tree
Stack Trie Beap Doubly Lined List
array Rope BIT BST Tango Tree
Array Trie Treap
Binomial Heap Tree Array
Queue Rope
Beap Heap Treap BIT queue
Suffix Array BST Tree Splay Tree Soft Heap Linked List

Agenda

-  Binary searchTree.
-  Operation on Binary searchTree.
-  **Traversing Binary Tree Recursively**
-  **Graph**

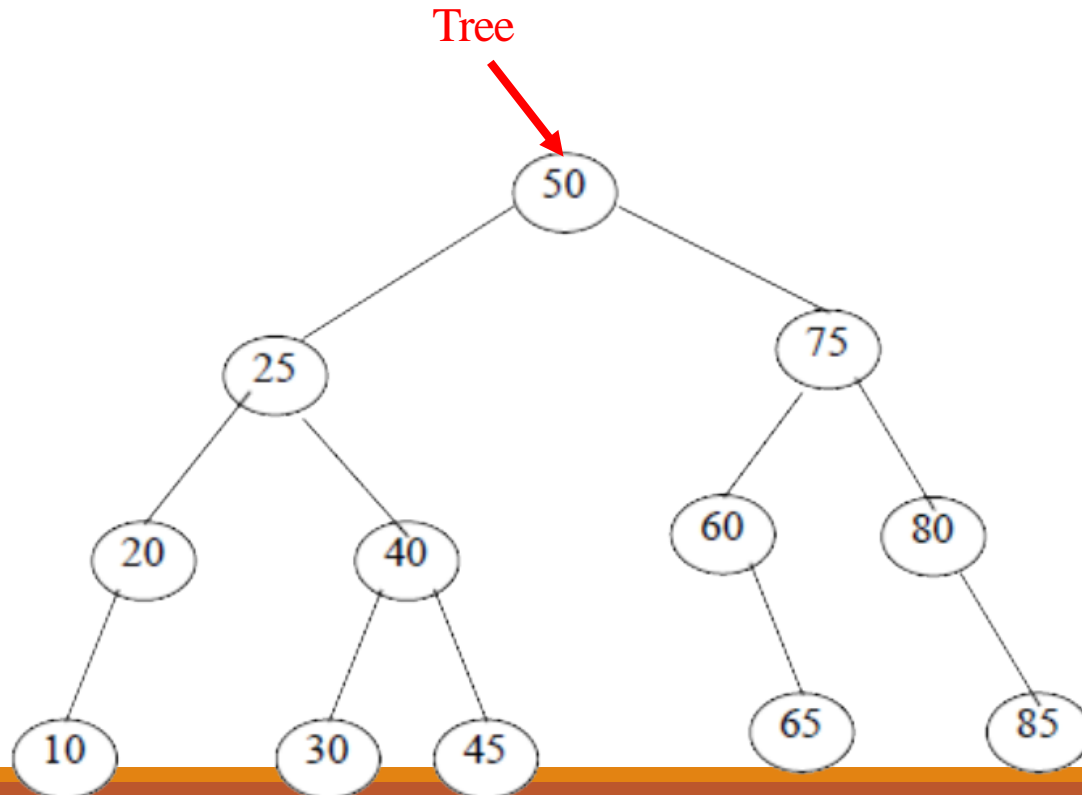
Binary Search Tree Construction

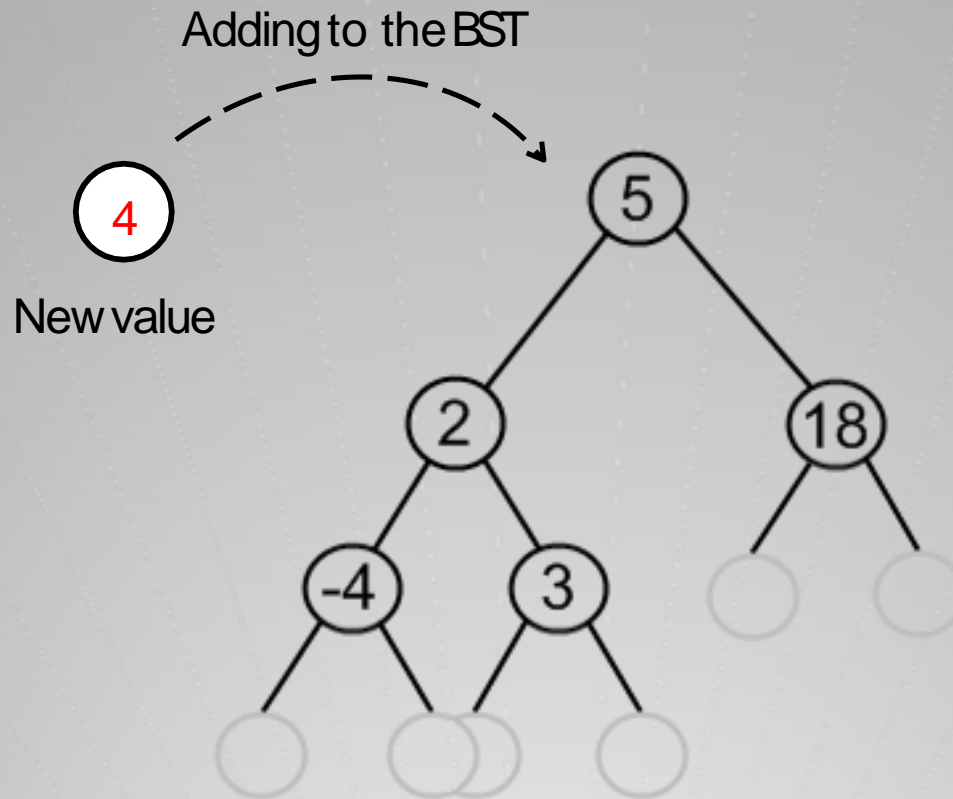
- ❑ Construct a binary search tree using the following sequence:



50,75,25,20,60,40,30,10,45,65,80,85

- ❑ The BST Tree:





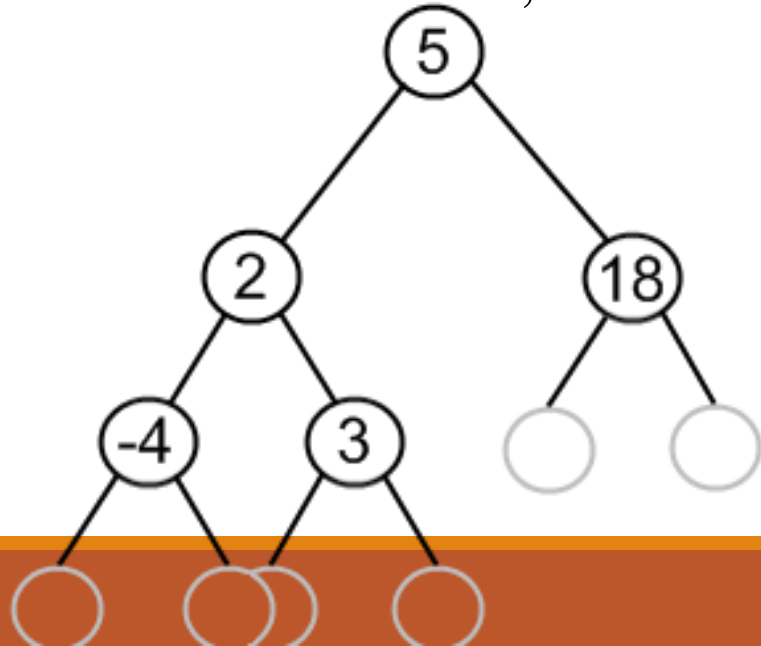
Adding a value to the BST

Adding a value to the BST

❑ Adding a value to BST can be divided into two stages:

1. **search** for a place to put a new element;
2. **insert** the new element to this place.

- ❑ If a new value **is less**, than the current node's value,
go to the **left sub-tree**,
- ❑ else
go to the **right sub-tree**



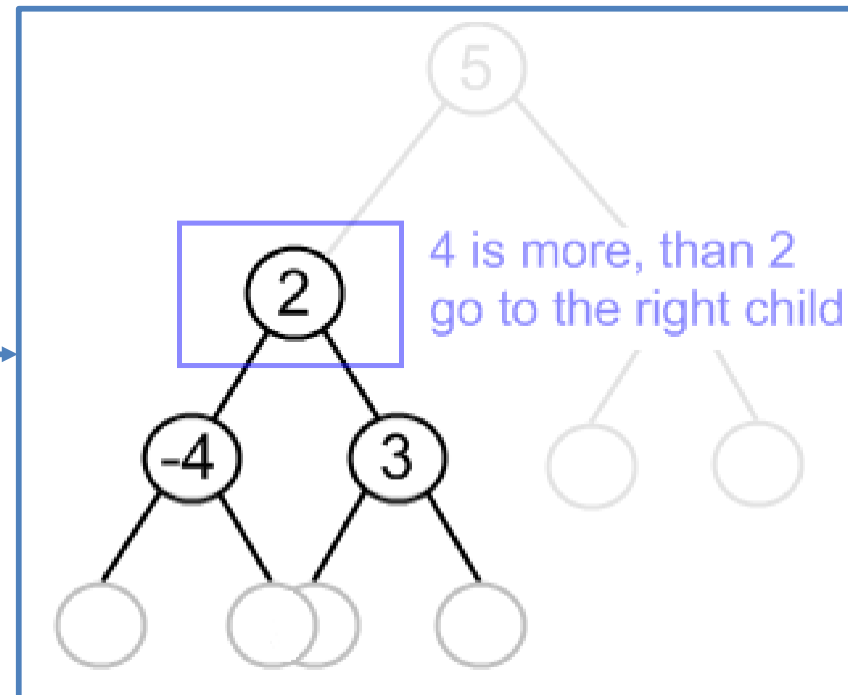
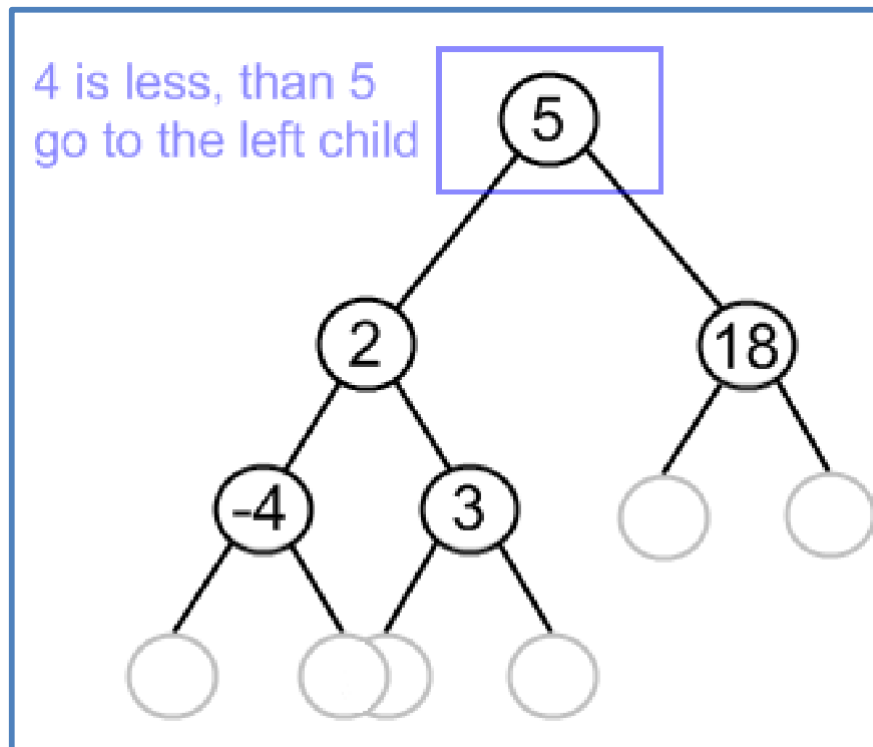
Adding a value to the BSTAlg.

❑ Starting from the root,

1. Check whether value in current node and a new value are equal. If so, duplicate is found. Otherwise
2. If a new value is less than the node's value:
 - ❑ If a current node has no left child, place for insertion has been found
 - ❑ Otherwise, handle the left child with the same algorithm.
2. If a new value is greater than the node's value:
 - ❑ If a current node has no right child, place for insertion has been found;
 - ❑ Otherwise, handle the right child with the same algorithm.

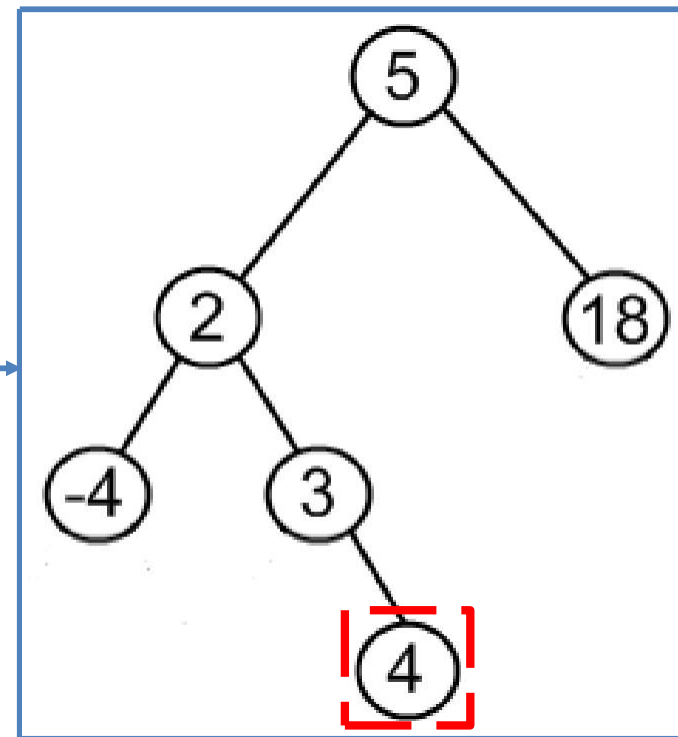
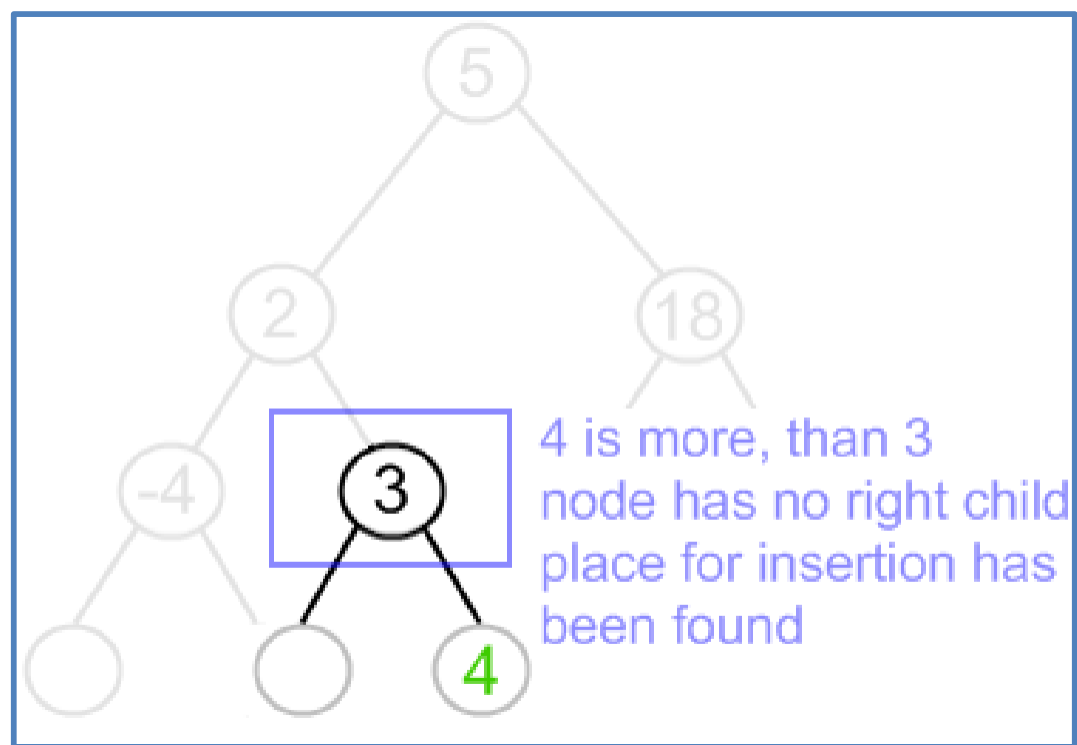
Example for Adding a value to the BST

❑ Insert 4 to the tree:



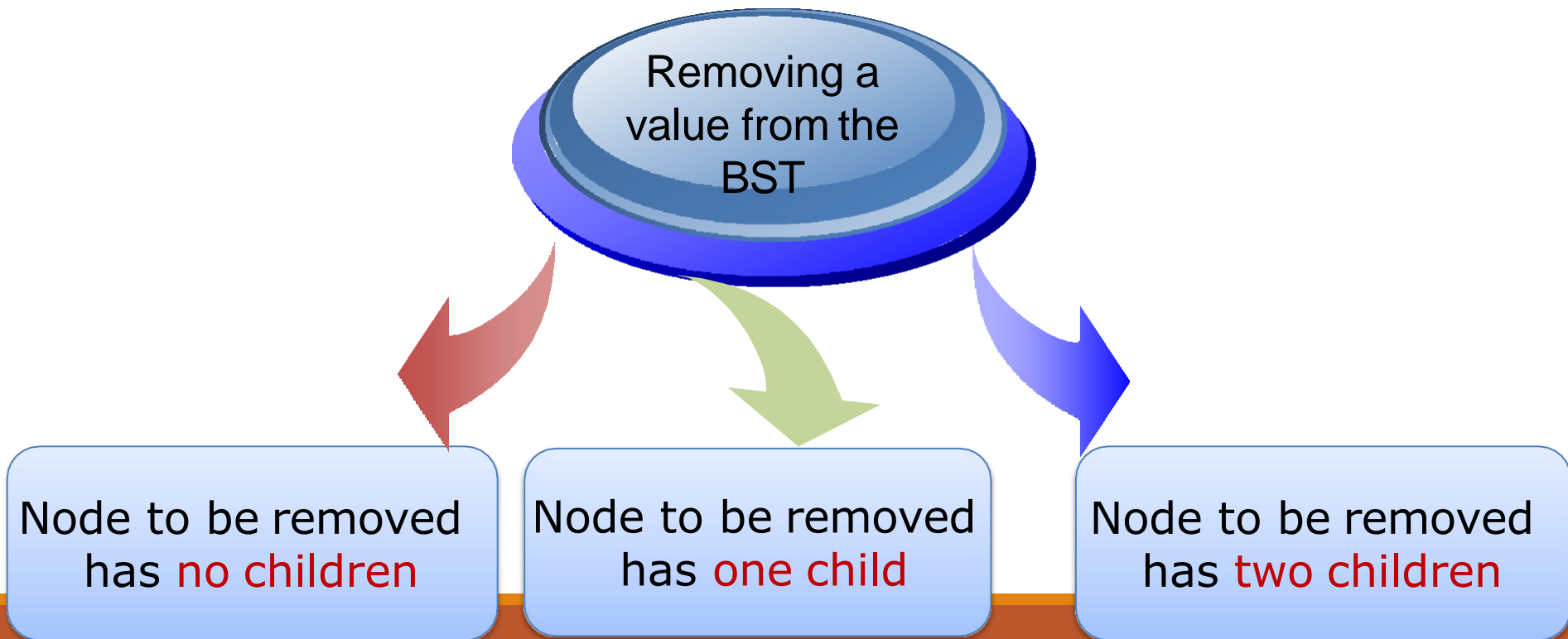
Example for Adding a value to the BST

❑ Insert 4 to the tree:



Removing a value from the BST

- ❑ Remove operation on binary search tree is more complicated, than add and search. Basically, it can be divided into two stages:
 1. search for a node to remove;
 2. if the node is found, run remove algorithm.

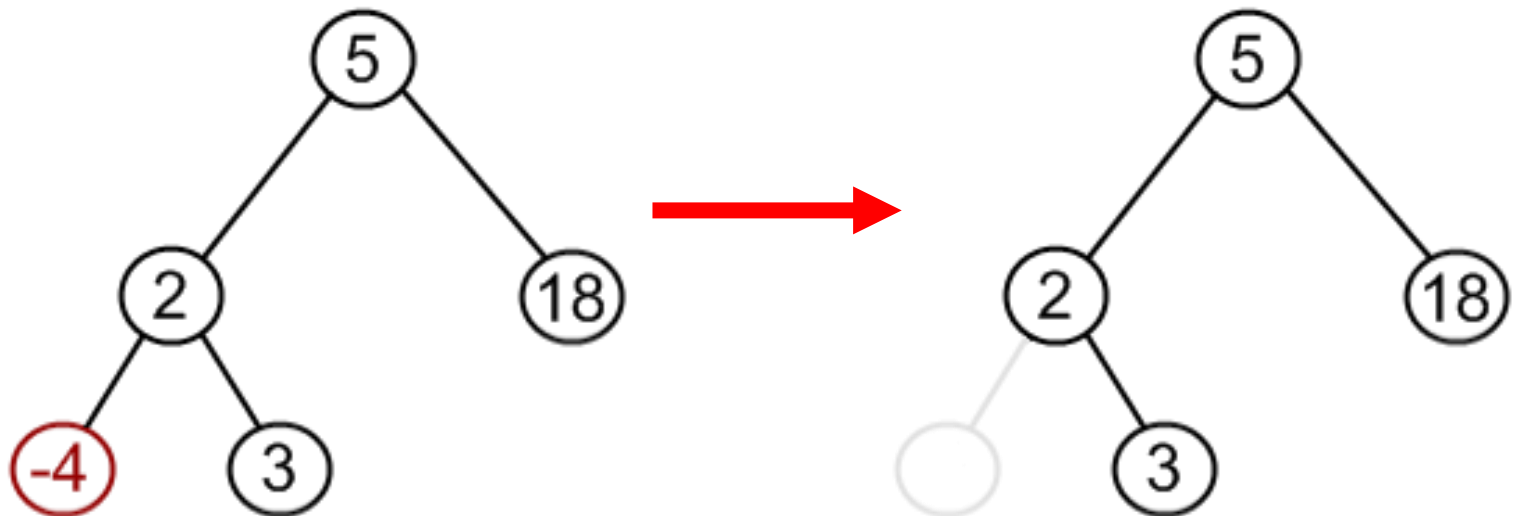


Removing a value from the BST

1. Node to be removed has no children: (Case 1)

This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

❑ Example: Remove -4 from a BST.

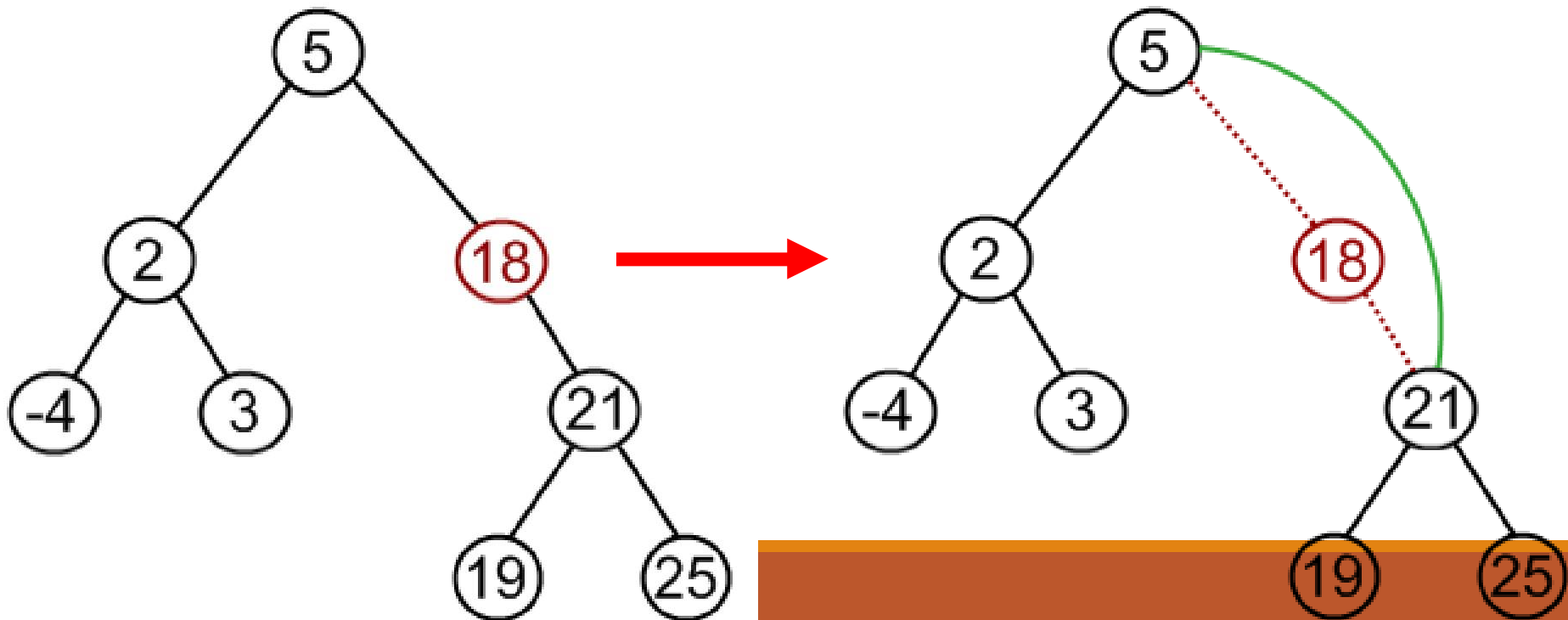


Removing a value from the BST

2. Node to be removed has one child:) **Case 2)**

□ In this case, node is cut from the tree and algorithm links single child (with its sub-tree) directly to the parent of the removed node.

□ **Example:** Remove 18 from a BST



Removing a value from the BST

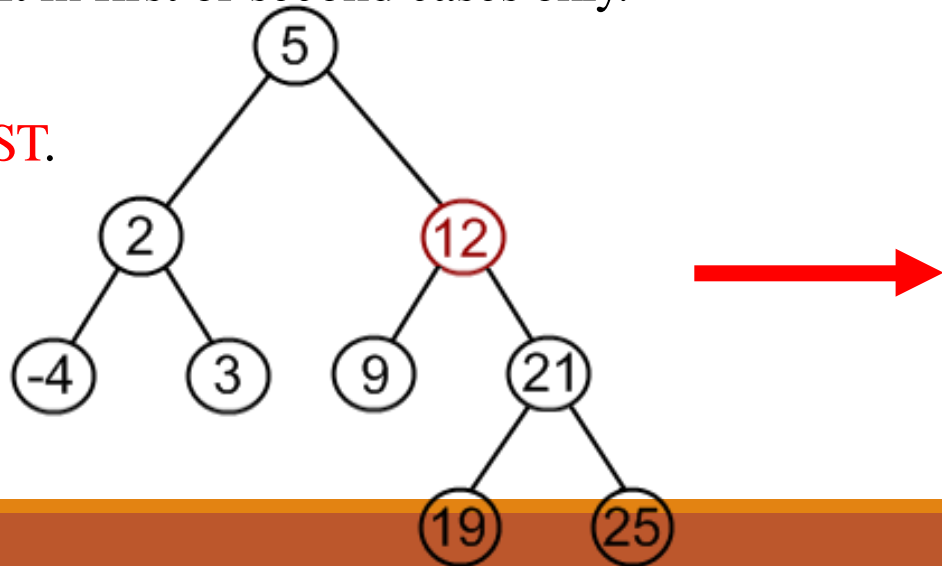
Node to be removed has two children: (Case3)

❑ This is the most complex case.

- Find a minimum value in the right sub-tree
- Replace value of the node to be removed with found minimum. Now, right sub-tree contains a duplicate!
- Apply remove to the right sub-tree to remove a duplicate.

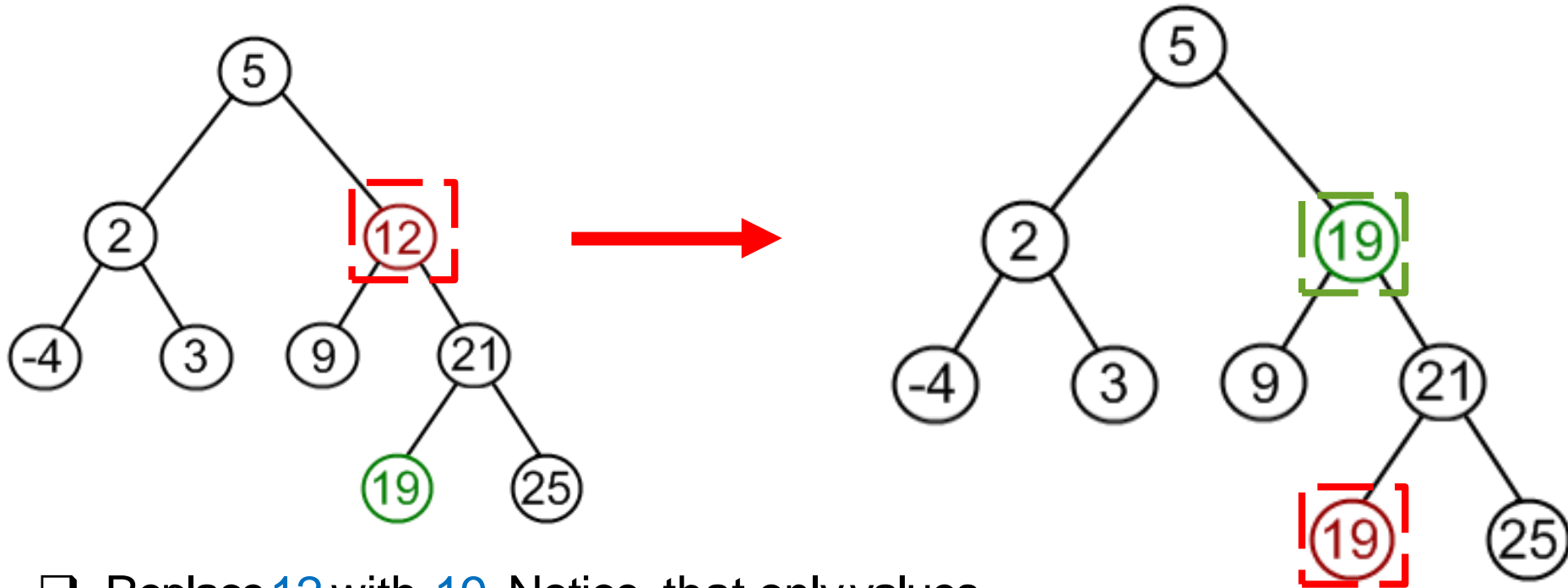
❑ Notice, that the node with minimum value has no left child and, therefore, its removal may result in first or second cases only.

❑ Example. Remove 12 from a BST.



Removing a value from the BST

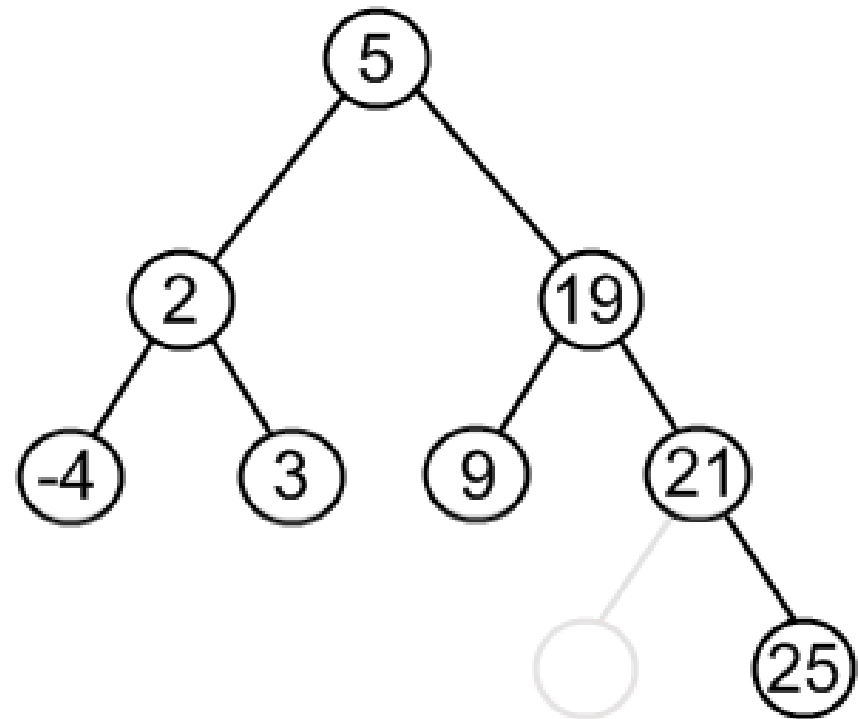
- ❑ Find minimum element in the right sub-tree of the node to be removed. In current example it is 19

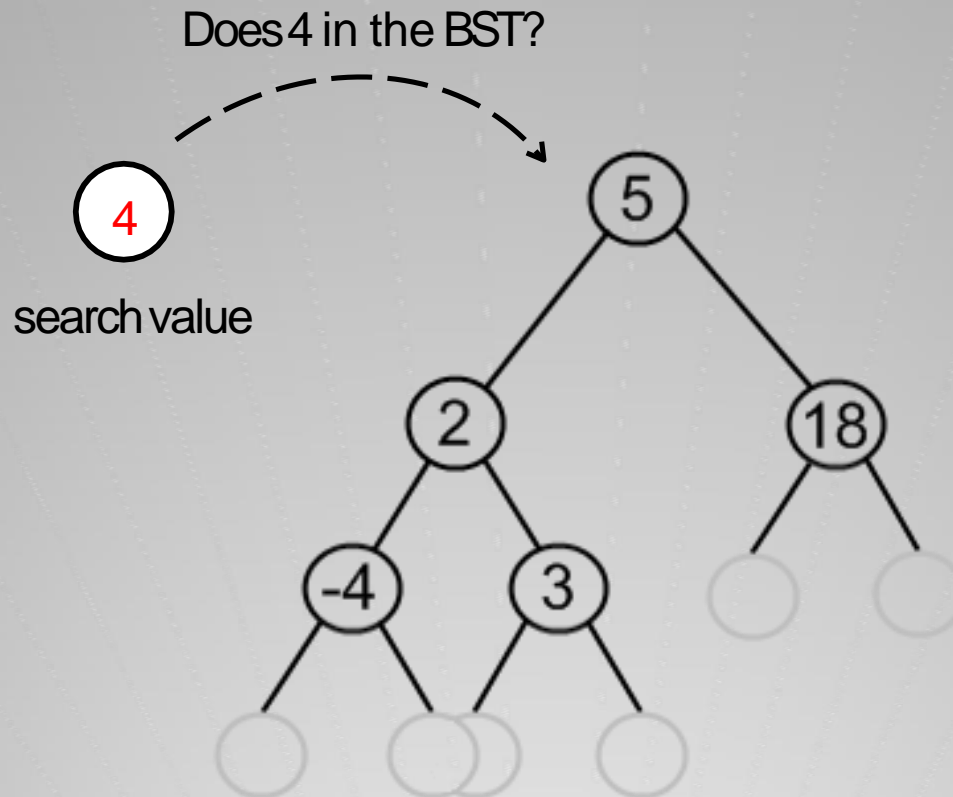


- ❑ Replace 12 with 19. Notice, that only values are replaced, not nodes.
- ❑ Now we have two nodes with the same value.

Removing a value from the BST

- ❑ Remove 19 from the left sub-tree.





Searching for a value in the BST

Searching for a value in the BST

- ❑ Searching for a value in a BST is very similar to add operation.
- ❑ Search algorithm traverses the tree "in-depth", choosing appropriate way to go, following binary search tree property and compares value of each visited node with the one, we are looking for.
- ❑ Algorithm stops in two cases:
 - ❑ A node with necessary value is found;
 - ❑ The algorithm has no way to go.

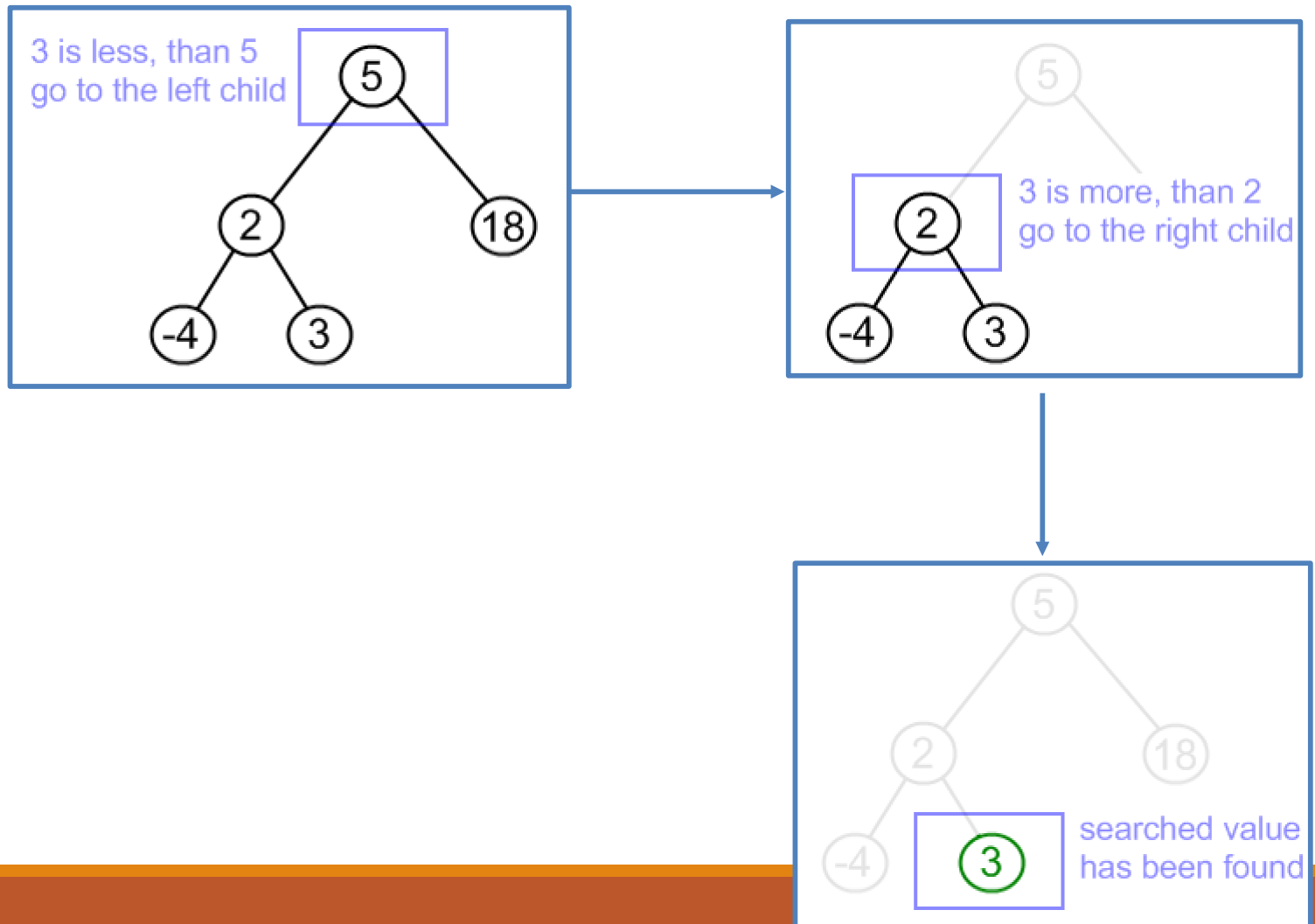
Searching for a value in the BST

□ search algorithm utilizes recursion. Starting from the root

1. check whether value in current node and searched value are equal. If so, value is found. Otherwise
2. if searched value is less than the node's value:
 1. if current node has no left child, searched value doesn't exist in the BST;
 2. otherwise, handle the left child with the same algorithm.
3. if a new value is greater than the node's value:
 1. if current node has no right child, searched value doesn't exist in the BST;
 2. otherwise, handle the right child with the same algorithm.

Example for Searching a value in the BST

- ❑ Search for 3 in the tree, shown above.



Create binary search tree:program

```
include<iostream>
```

```
using namespace std;
```

```
struct Node
```

```
{
```

```
    Node *Lchild;
```

```
    Node *Rchild;
```

```
    int data;
```

```
};
```

Create binary search tree:program

```
class binary_search_tree
{
    private:
        Node *root;

    public:
        binary_search_tree(){ root=NULL; }
        int isempty() { return(root==NULL); }
        void insert(int item);
        void displayBinaryTree();
        void printBinaryTree(Node *);
};
```

Create binary search tree:program

```
void binary_search_tree ::insert(int item)
```

```
{
```

```
Node *p=new Node;
```

```
p->data=item;
```

```
p->Lchild=NULL;
```

```
p->Rchild=NULL;
```

```
if(isempty())
```

```
root=p;
```

5 18 2 3 -4 list



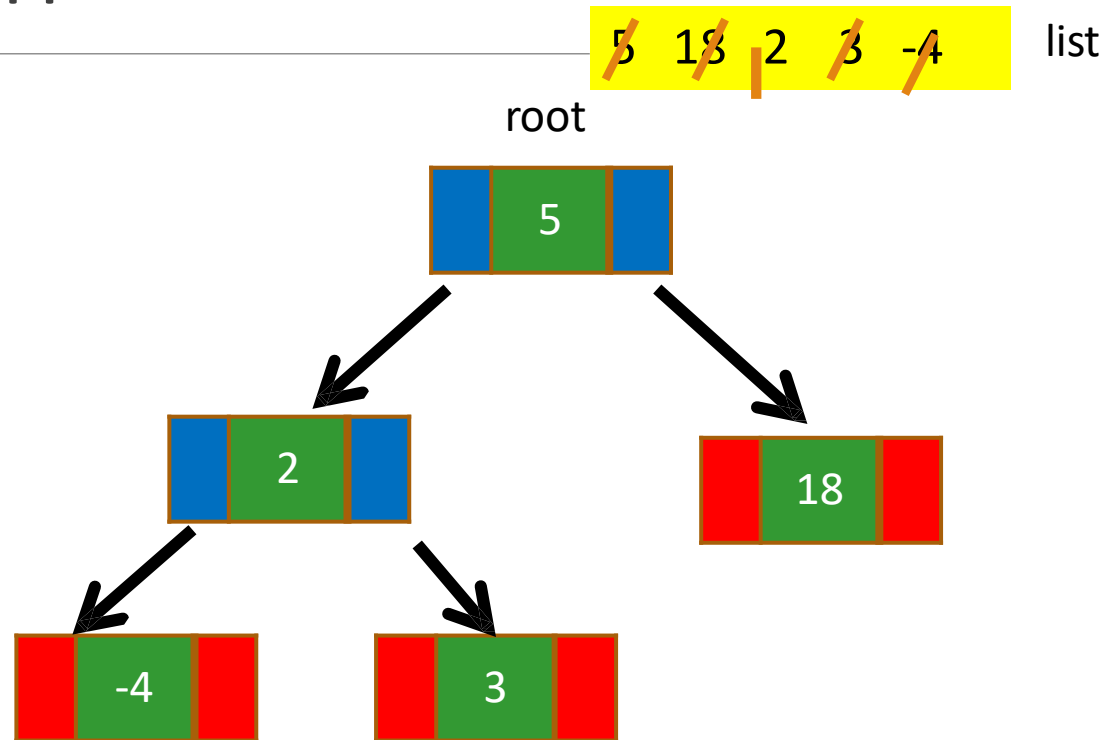
root



Create binary search tree:program

```
else{  
Node *temp=root;  
Node *parent=NULL;  
while(temp!=NULL)  
{  
parent=temp;  
if(item>temp->data)  
temp=temp->Rchild;  
else  
temp=temp->Lchild;  
}  
}
```

```
if(item<parent->data)  
parent->Lchild=p;  
else  
parent->Rchild=p;}
```



Create binary search tree:program

```
void binary_search_tree ::displayBinaryTree(){  
    printBinaryTree(root);  
}  
  
void binary_search_tree ::printBinaryTree(Node *ptr){  
    if(ptr!=NULL){  
        printBinaryTree(ptr->left);  
        cout<<" "<<ptr->data<<" ";  
        printBinaryTree(ptr->right);  
    }  
}
```

Create binary search tree:program

```
int main(){  
    binary_search_tree tree1;  
    tree1.insert(5);  
    tree1.insert(18);  
    tree1.insert(2);  
    tree1.insert(3);  
    tree1.insert(-4);  
    cout<<"Binary tree created: "<<endl;  
    tree1.displayBinaryTree();  
}
```

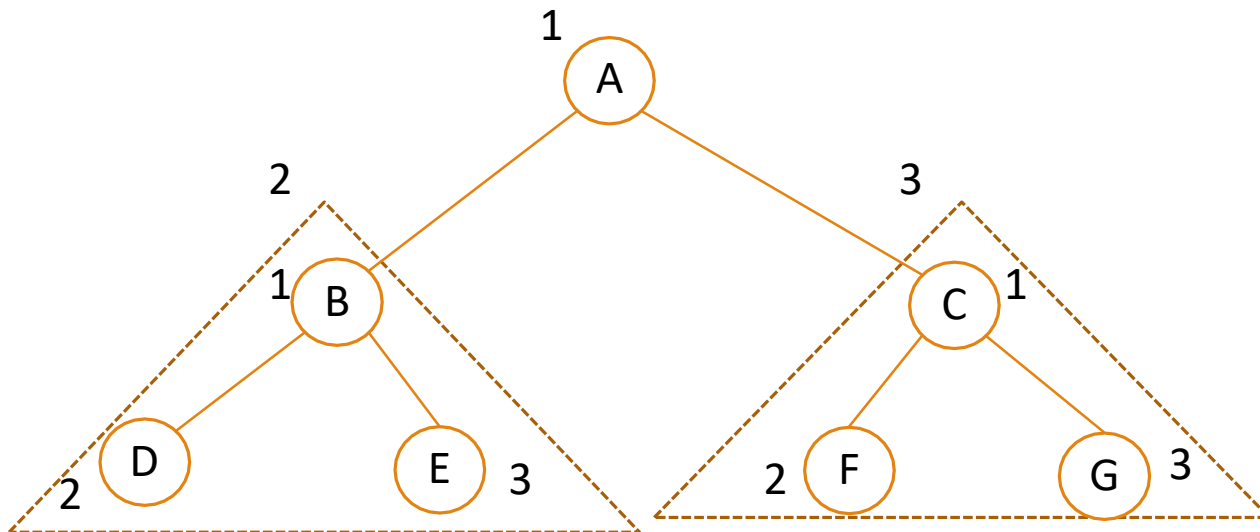

Traversing Binary Tree Recursively

Pre Order Traversal (**node**-left-right)

In order Traversal (left-**node**-right)

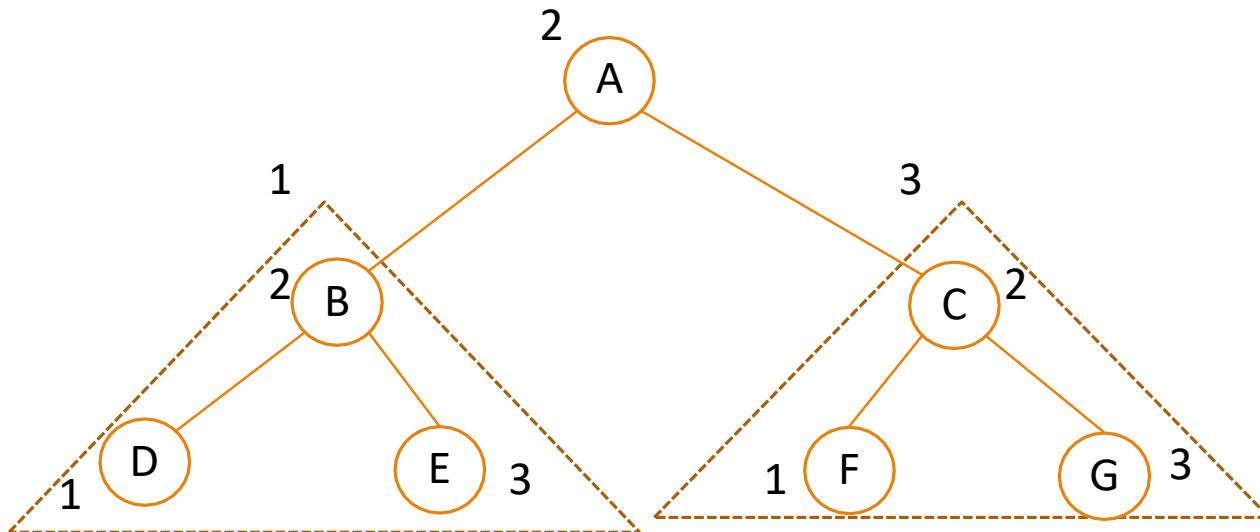
Post Order Traversal (left-right-**node**)

Pre-order Traversal



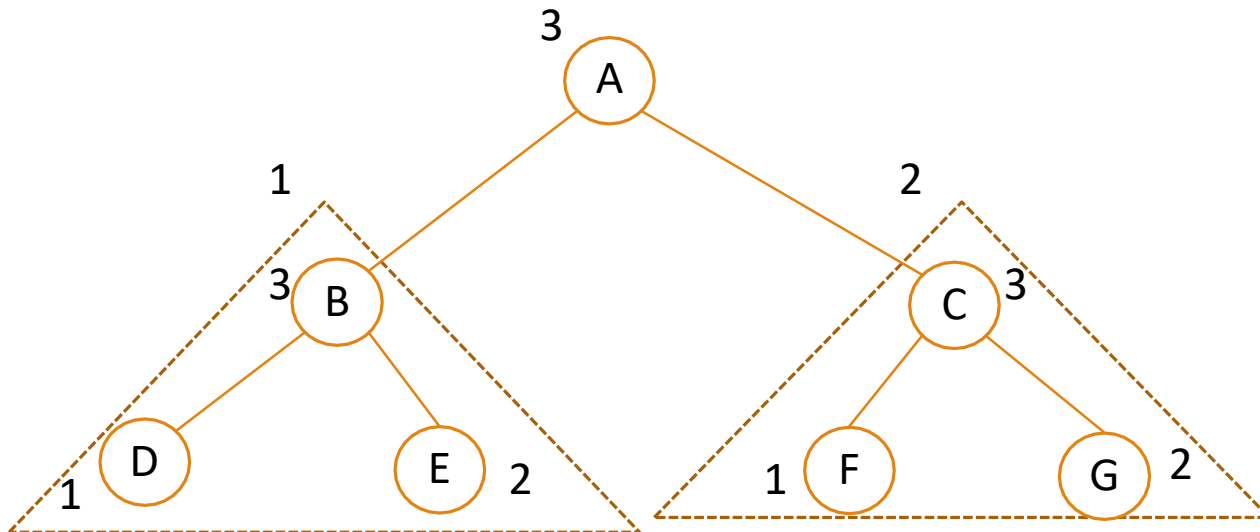
$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

In-order Traversal



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

Post-order Traversal



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

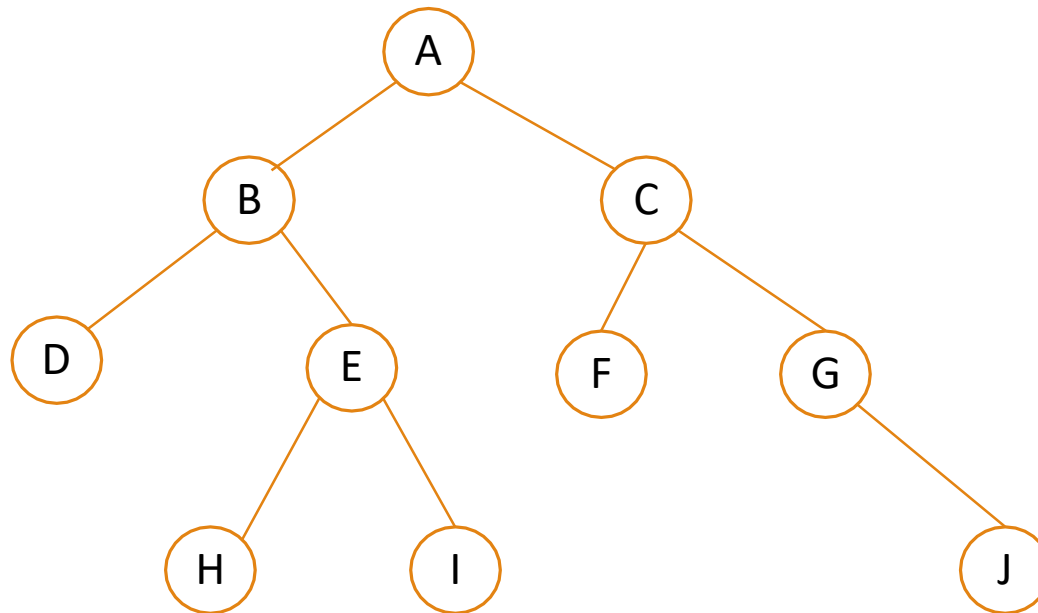
Pre Orders Traversal Recursively

To traverse a non-empty binary tree in **pre order** following steps one to be processed

1. **Visit** the **root** node
2. Traverse the **left** sub tree in preorder
3. Traverse the **right** sub tree in preorder

```
void preorder (Node * root)
{
    if (root != NULL)
    {
        cout<< root → data;
        preorder(root → Lchild);
        preorder(root → Rchild);
    }
}
```

Pre Orders Traversal Recursively



The preorder traversal of a binary tree in is A, B, D, E, H, I, C, F, G, J.

post Orders Traversal Recursively

POST ORDER TRAVERSAL RECURSIVELY

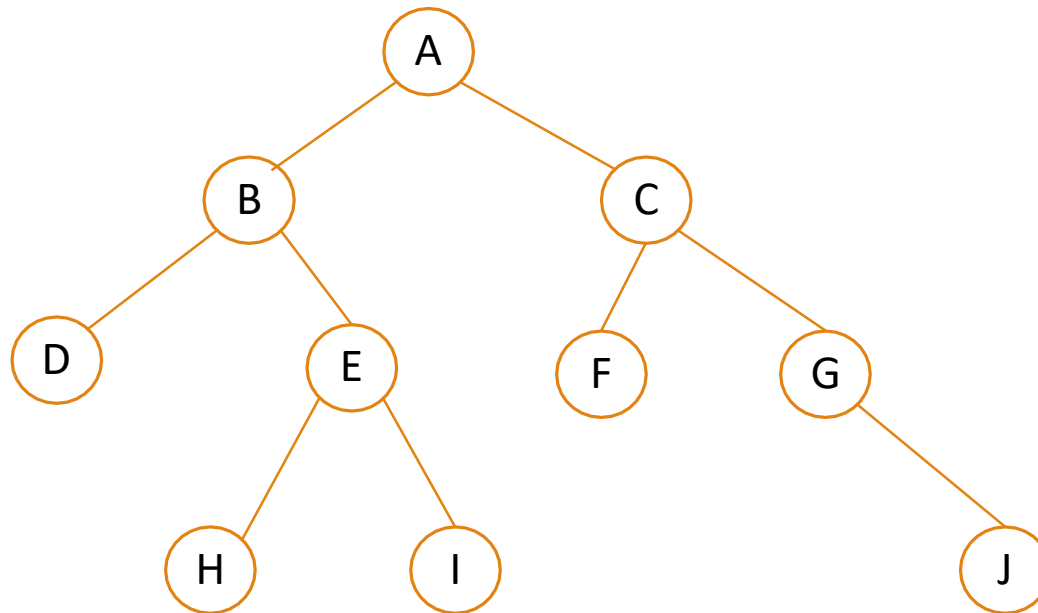
The post order traversal of a non-empty binary tree can be defined as :

1. Traverse the left sub tree in post order
2. Traverse the right sub tree in post order
3. Visit the root node

```
void postorder (Node *root)
```

```
{  
    if (root != NULL)  
    { postorder(root → Lchild);  
      postorder(root → Rchild);  
      cout<<root → data;  
    }  
}
```

post Orders Traversal Recursively



The post order traversal of a binary tree is D, H, I, E, B, F, J, G, C, A

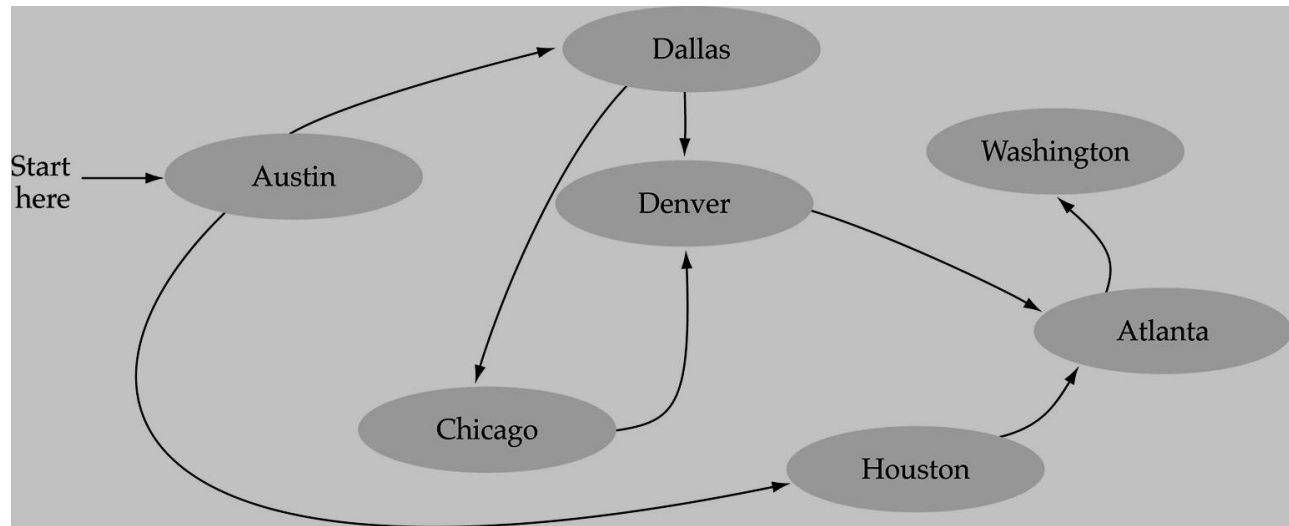
in Orders Traversal Recursively

in ORDER TRAVERSAL RECURSIVELY

The in order traversal of a non-empty binary tree can be defined as :

1. Traverse the left sub tree in post order
2. Visit the root node
3. Traverse the right sub tree in post order

```
void inorder (Node *root)
{
    if (root != NULL)
    {
        postorder(root → Lchild);
        cout<<root → data;
        postorder(root → Rchild);
    }
}
```

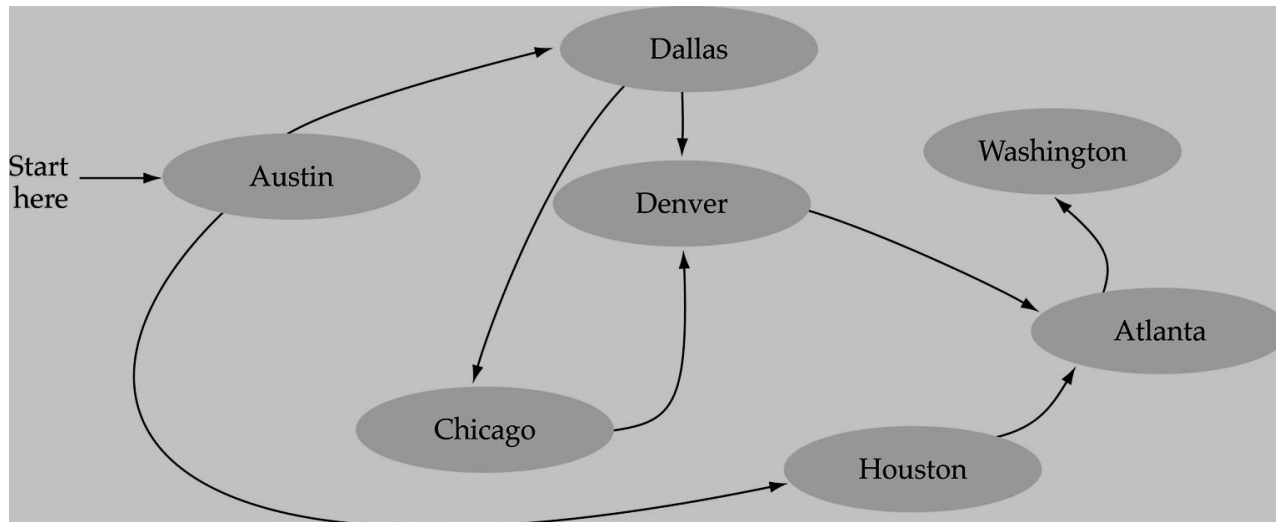


Graph

What is a graph?

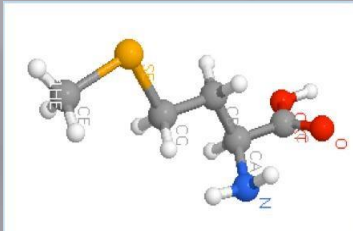
A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other

The set of edges describes relationships among the vertices

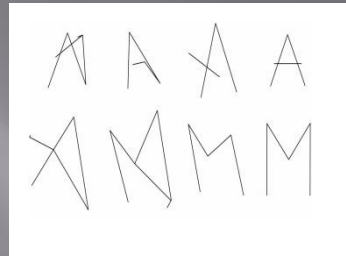


Graphs

Chemical structure



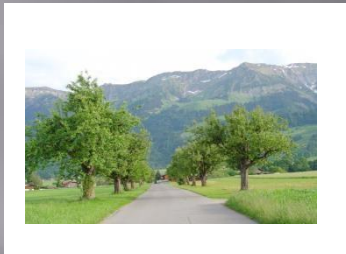
Letters



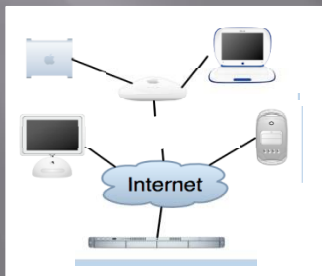
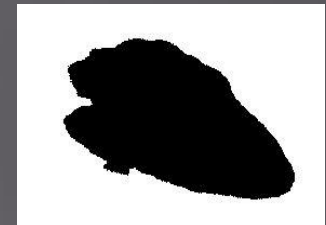
Fingerprint



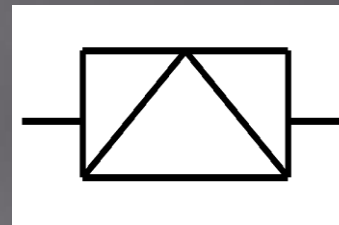
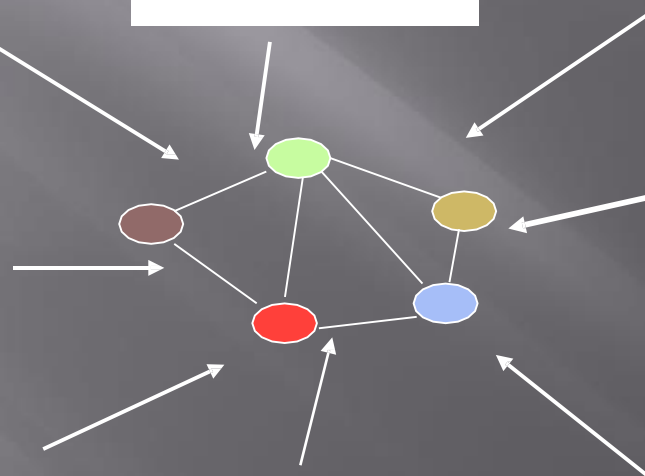
Images



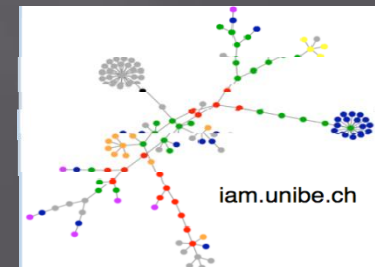
shapes



Physical Network



Graphical Symbols



Logical network

Formal definition of graphs

A graph G is defined as follows:

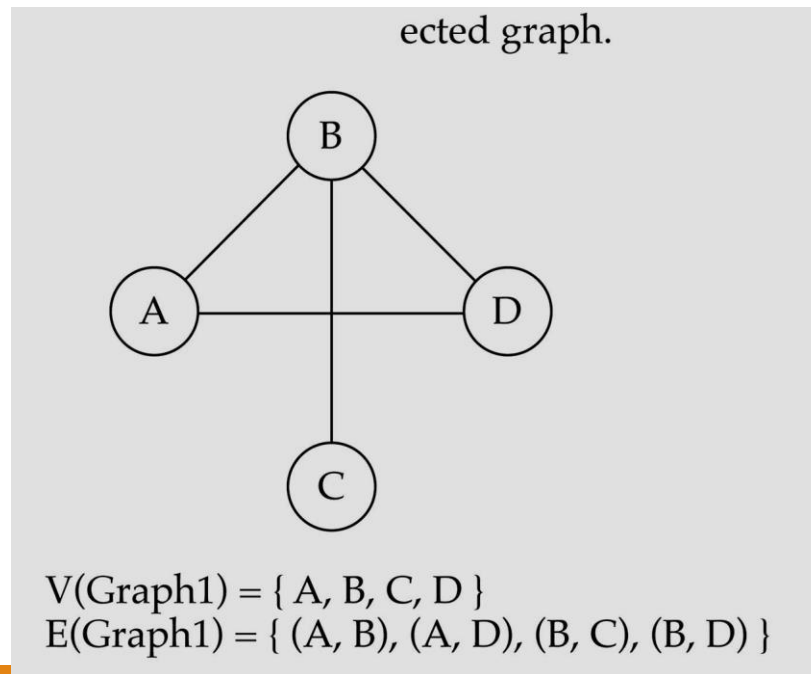
$$G=(V,E)$$

$V(G)$: a finite, nonempty set of vertices

$E(G)$: a set of edges (pairs of vertices)

Directed vs. undirected graphs

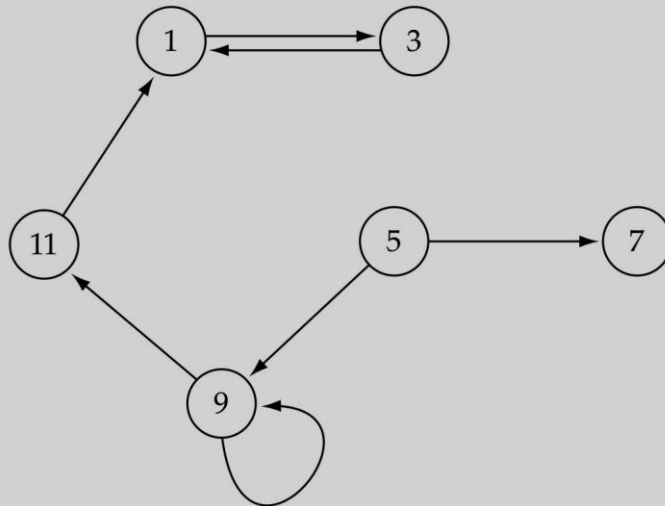
When the edges in a graph have no direction, the graph is called *undirected*



Directed vs. undirected graphs (cont.)

When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.



$V(\text{Graph2}) = \{ 1, 3, 5, 7, 9, 11 \}$

$1), (9, 9), (11, 1) \}$

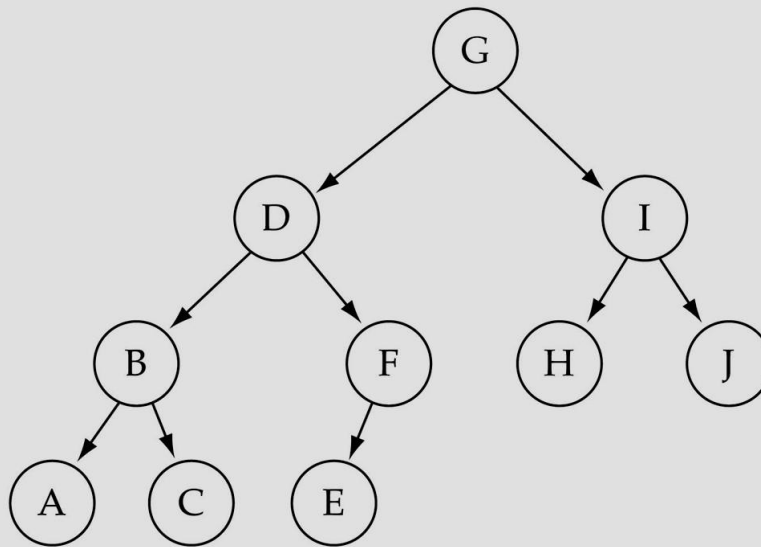
$E(\text{Graph2}) = \{(1,3) (3,1) (5,9) (9,11) (5,7)$

Warning: if the graph is directed, the order of the vertices in each edge is important !!

Trees vs graphs

Trees are special cases of graphs!!

(c) Graph3 is a directed graph.

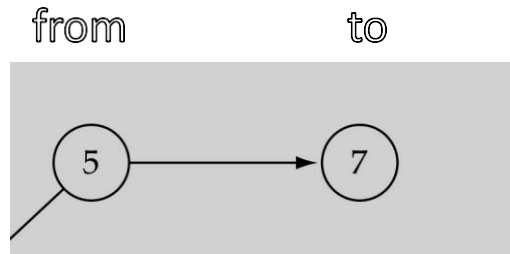


$V(\text{Graph3}) = \{ A, B, C, D, E, F, G, H, I, J \}$

$E(\text{Graph3}) = \{ (G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E) \}$

Graph terminology

Adjacent nodes: two nodes are adjacent if they are connected by an edge



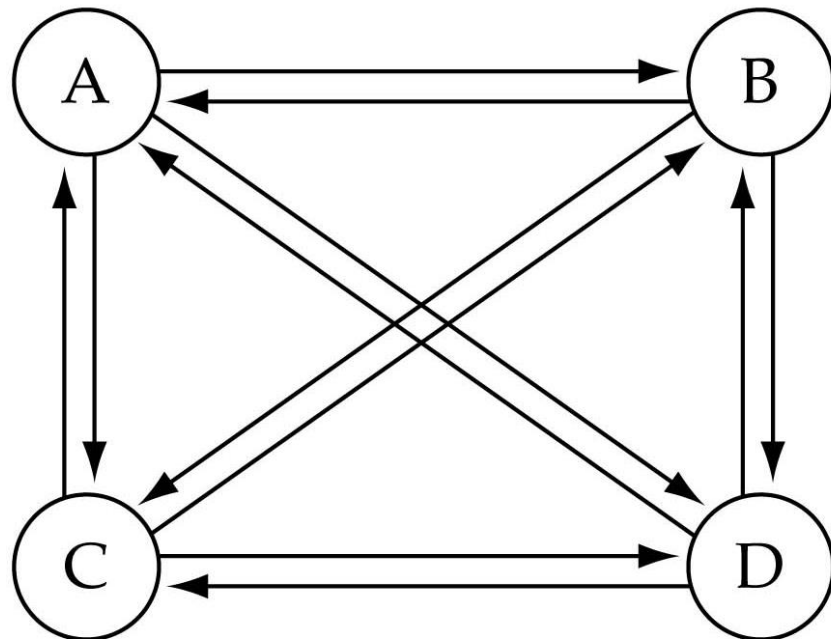
Path: a sequence of vertices that connect two nodes in a graph

Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

What is the number of edges in a complete directed graph with N vertices?

$$N * (N-1)$$

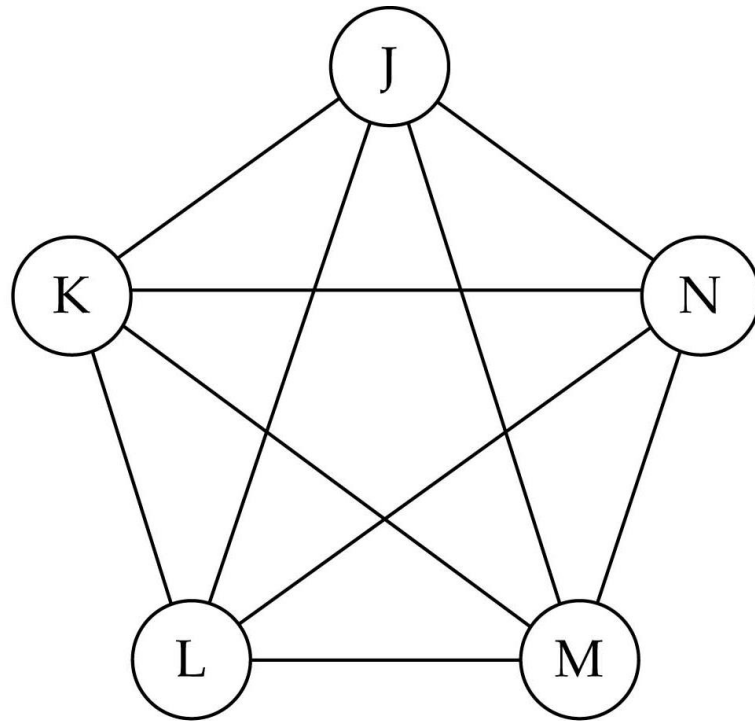


(a) Complete directed graph.

Graph terminology (cont.)

What is the number of edges in a complete undirected graph with N vertices?

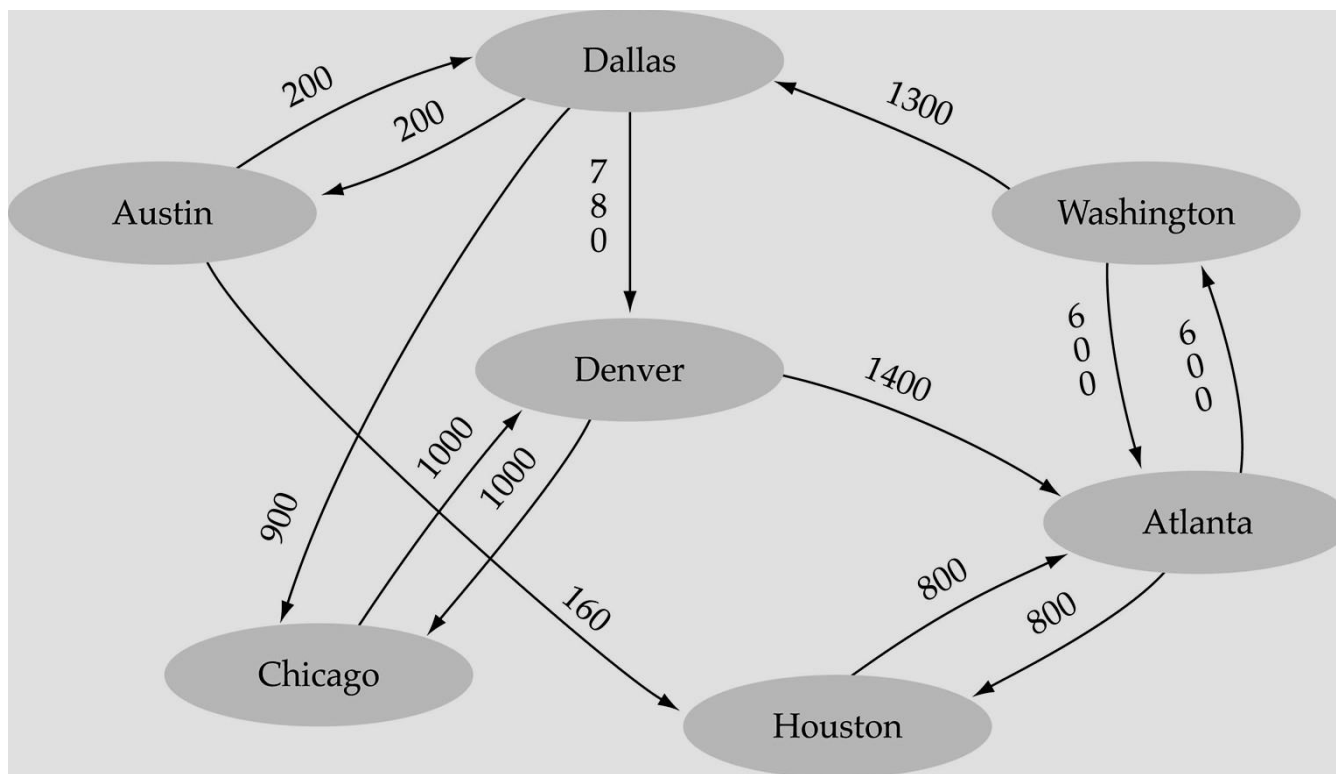
$$N * (N-1) / 2$$



(b) Complete undirected graph.

Graph terminology (cont.)

Weighted graph: a graph in which each edge carries a value

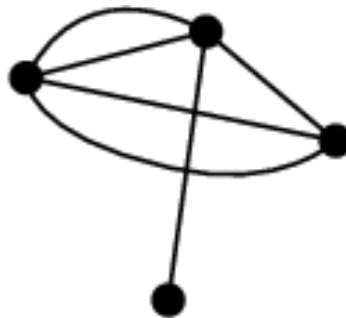


simple graph

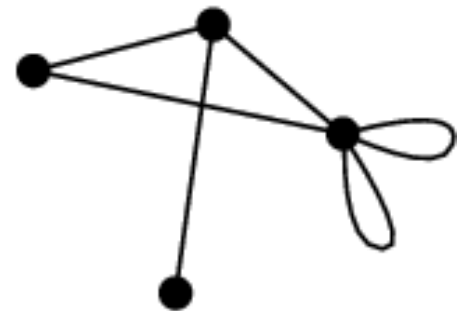
A simple graph, also called a strict graph is an unweighted undirected [graph](#) containing no [graph loops](#) or [multiple edges](#)



simple graph



*nonsimple graph
with multiple edges*



*nonsimple graph
with loops*

Connected Graph:

Connected Graph: A connected graph is the one in which there is a path between each of the vertices. This means that there is not a single vertex which is isolated or without a connecting edge.

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.

➤ for the **weighted graph** instead of storing 0 or 1 in $A_{i,j}$, the weight or cost of the edge will be stored.

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

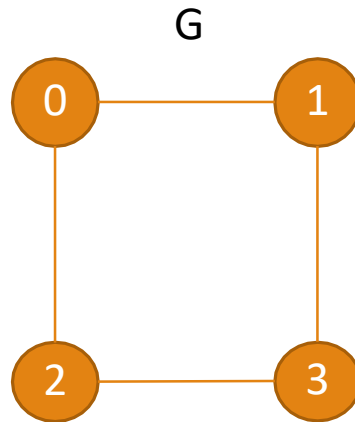
■ Adjacency list

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 .

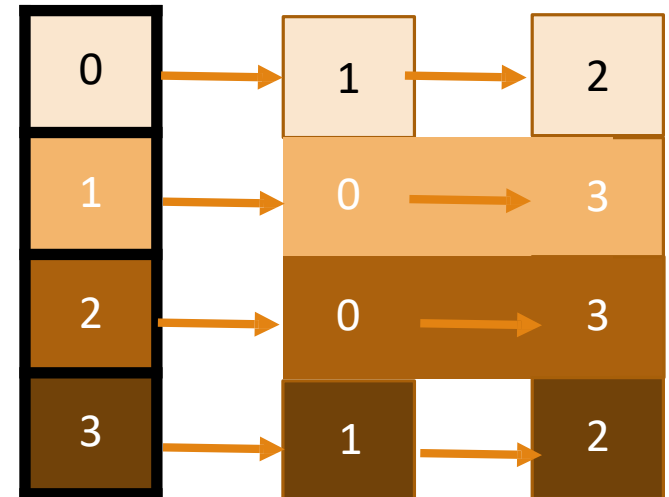
Graph representation

Adjacency matrix of Graph G

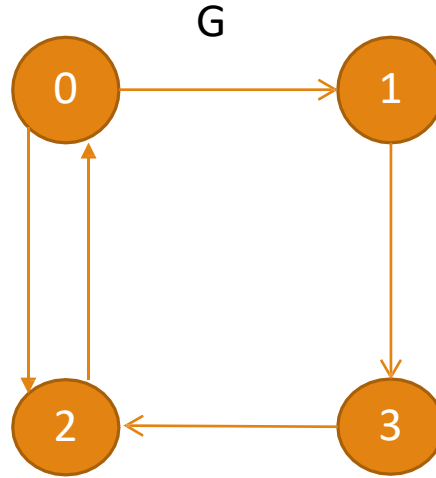
| i/j | 0 | 1 | 2 | 3 |
|-----|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 |



Adjacency List of Graph G



Graph representation



Adjacency matrix of Graph G

| i/j | 0 | 1 | 2 | 3 |
|-----|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 |

Adjacency List of Graph G

