
IT2070 – Data Structures and Algorithms

Introduction

Subject Group

Malabe Campus

- Dr. Dinuka R. Wijendra - LIC
- Ms. Jenny Krishara
- Ms. Thamali Kelegama
- Ms. Poorna Panduwawela

Metro Campus

- Ms. Thamali Kelegama

Kandy Center

- Ms. Chathuni Thilakarathne

Matara Center

- Ms. Bhagyanie Chathurika

Jaffna Center

- Ms. Sangeetha ArunPirakash

Teaching Methods

- Lectures – 2 hours/week
- Tutorials – 2 hour/week (WD)
- Labs – 2 hours /week

Student Evaluation

- Assessments (Two exams) - 40 %
- Final Examination - 60 %

Lectures will cover

Data Structures

- Stack data structure
- Queue data structure
- Linked list data structure
- Tree data structure

Algorithms

- Asymptotic Notations
- Algorithm designing techniques
- Searching and Sorting algorithms

Tutorials and Labs will cover

- Solve problems using the knowledge acquired in the lecture
- Get hands on experience in writing programs
 - Java
 - Python

Data Structures and Algorithms

- **Data Structures**

- Data structure is an arrangement of data in a computer's memory or sometimes on a disk.

- Ex: stacks, queues, linked lists, trees

- **Algorithms**

- Algorithms manipulate the data in these structures in various ways.

- Ex: searching and sorting algorithms

Data Structures and Algorithms

- **Usage of data structures**
 - Real world data storage
 - Real world modeling
 - queue, can model customers waiting in line
 - graphs, can represent airline routes between cities
 - Programmers Tools
 - stacks, queues are used to facilitate some other operations

Data Structures and Algorithms

Algorithms

Algorithm is a well defined computational procedure that takes some value or set of values as input and produce some value or set of values as output.

An algorithm should be

- correct.
- unambiguous.
- give the correct solution for all cases.
- simple.
- terminate.

Academic Integrity Policy

Are you aware that following are not accepted in SLIIT???

Plagiarism - using work and ideas of other individuals intentionally or unintentionally

Collusion - preparing individual assignments together and submitting similar work for assessment.

Cheating - obtaining or giving assistance during the course of an examination or assessment without approval

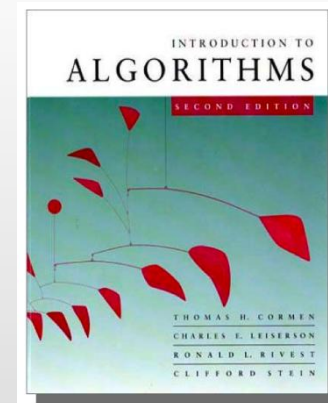
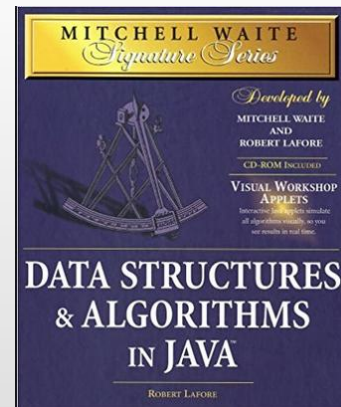
Falsification – providing fabricated information or making use of such materials

From year 2018 the committing above offenses come with serious consequences !

See General support section of Courseweb for full information.

References

1. Mitchell Waite, Robert Lafore, Data Structures and Algorithms in Java, 2nd Edition, Waite Group Press, 1998.
2. T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.



IT2070 – Data Structures and Algorithms

Lecture 01

Introduction to Stack

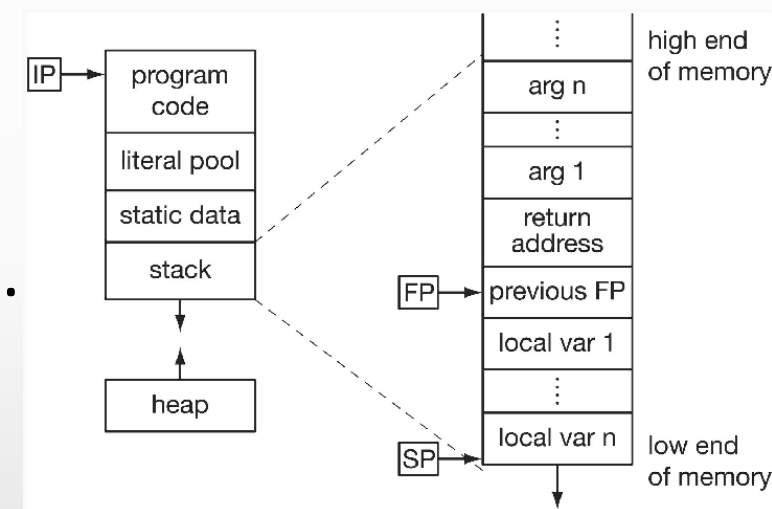
Stack



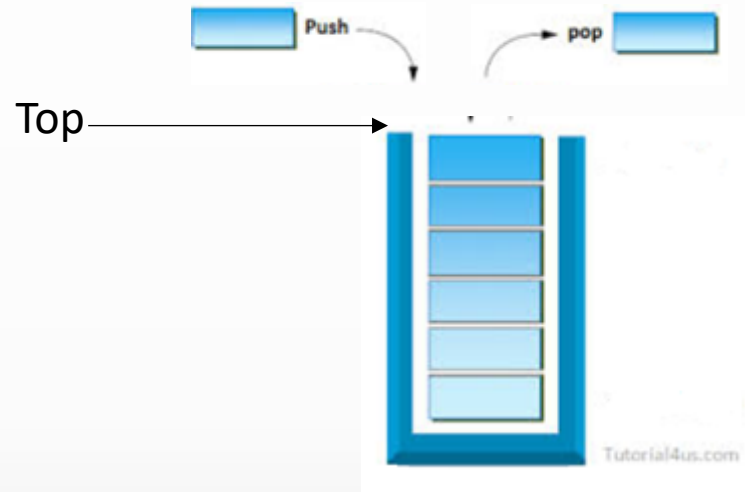
- Allows access to only one data item; the last item inserted
- If you remove this item, then you can access the next-to-last item inserted

Application of Stacks

- String Reverse
- Page visited history in Web browser.
- Undo sequence of text editor.
- Recursive function calling.
- Auxiliary data structure for Algorithms.
- Stack in memory for a process

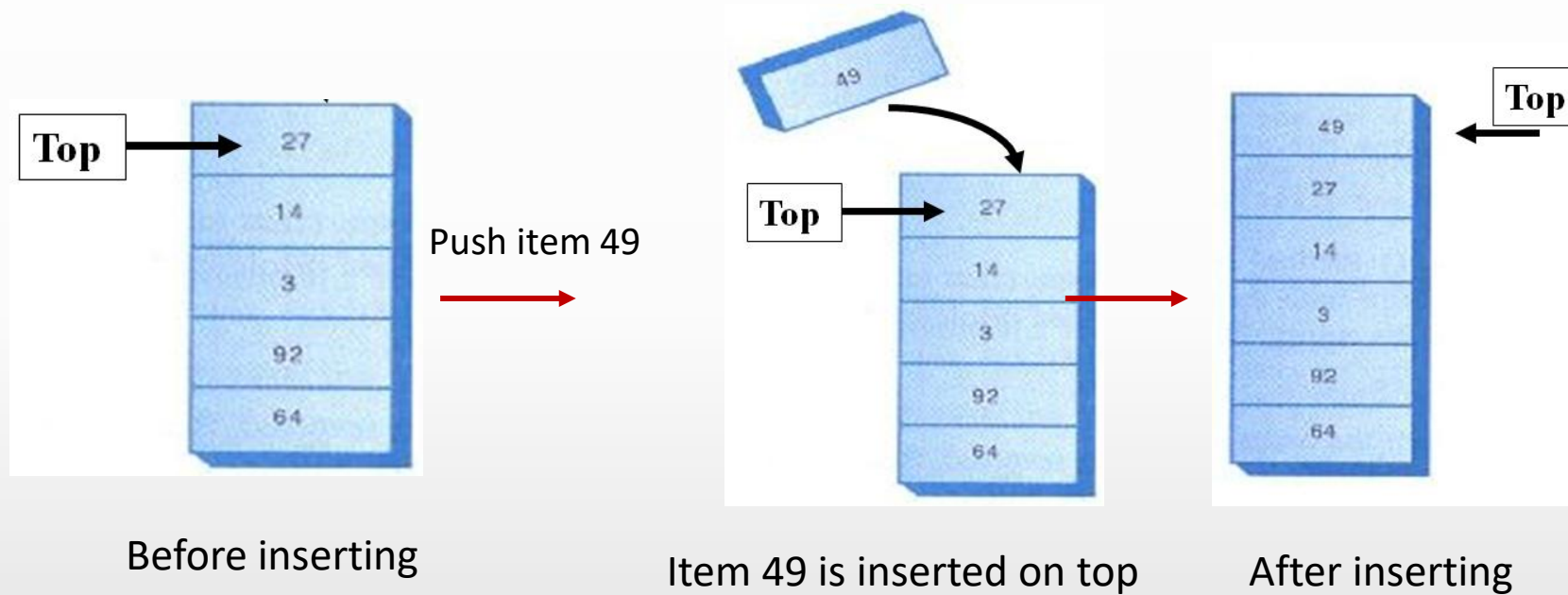


Stack

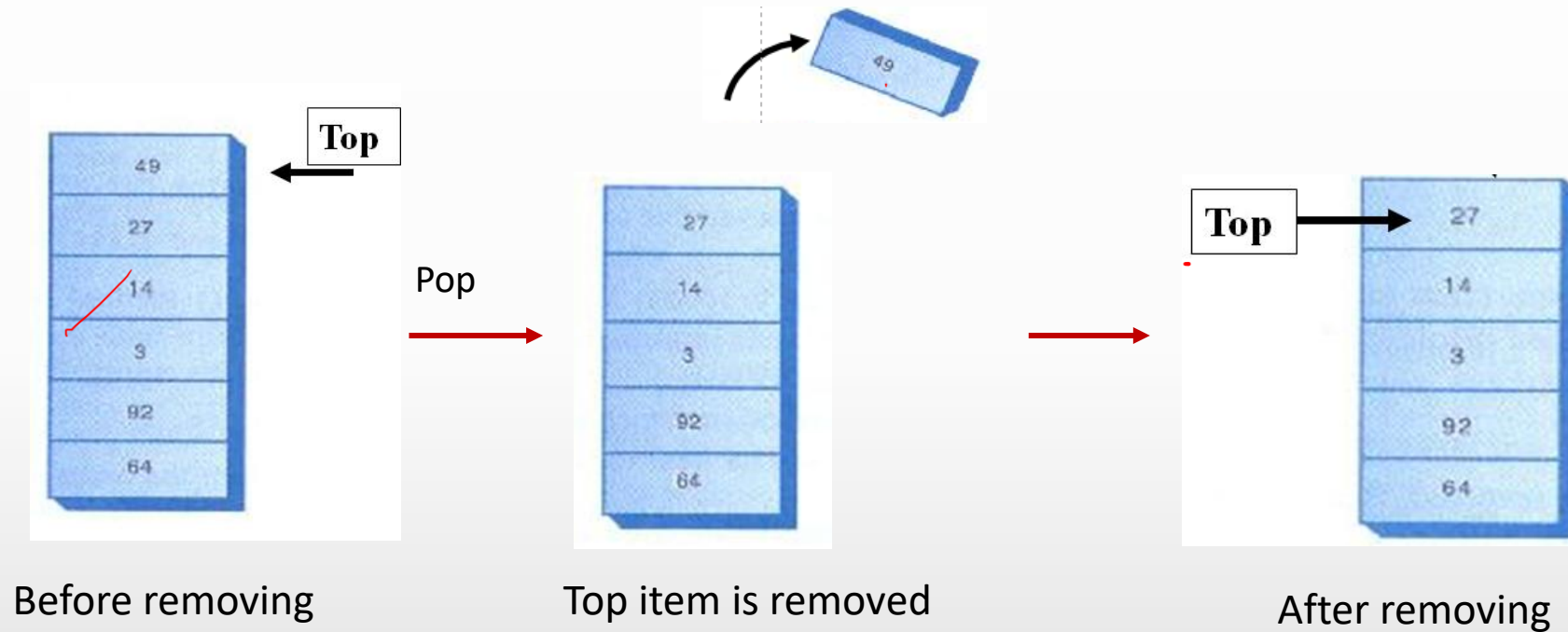


- In a stack all insertions and deletions are made at one end (Top). Insertions and deletions are restricted from the Middle and at the End of a Stack
- Adding an item is called Push
- Removing an item is called Pop
- Elements are removed from a Stack in the reverse order of that in which the elements were inserted into the Stack
- The elements are inserted and removed according to the Last-In-First-Out (LIFO) principle.

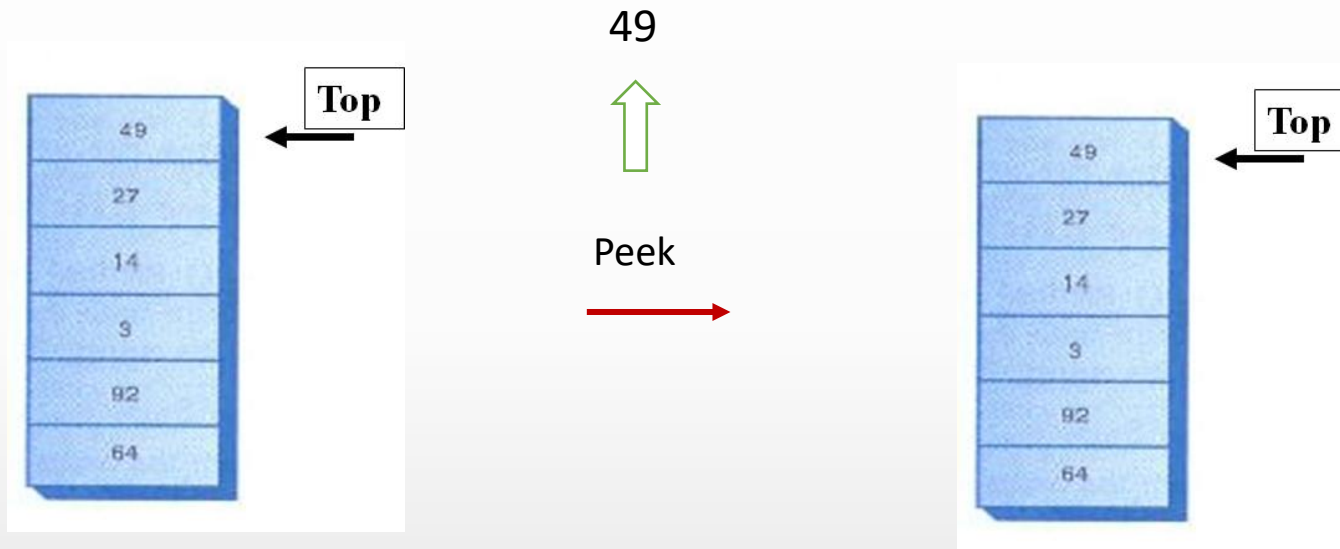
Stack - Push



Stack - Pop



Stack - Peek

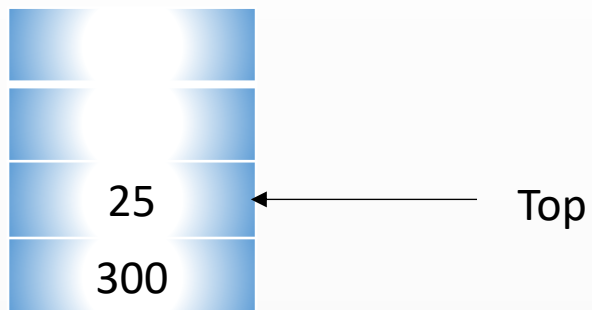


Stack remains the same

Peek is used to read the value from the top of the stack without removing it. You can peek only the Top item, all the other items are invisible to the stack user.

Question

Draw the stack frame after performing the below operations to the stack given below.



- i) Push item 50
- ii) Push item 500
- iii) Peek
- iv) Push item 100
- v) Pop
- vi) Pop
- vii) Pop
- viii) Pop



Uses of Stack

- The stack operations are built into the microprocessor.
- When a method is called, its return address and arguments are pushed onto a stack, and when it returns they're popped off.

Stack - Implementation

Stack implementation using an **array**

- Constructor creates a new stack of a size specified in its argument.
- Variable *top*, which stores the index of the item on the top of the stack.

```
class StackX {  
  
    private int maxSize; // size of stack array  
    private double[] stackArray;  
    private int top;      //top of the stack  
  
    public StackX(int s) { // constructor  
  
        maxSize = s;      // set array size  
        stackArray = new double[maxSize];  
        top = -1;         // no items  
    }  
  
    .....  
    .....  
}
```

Stack – Implementation - push

```
class StackX{

    private int maxSize; // size of stack array
    private double[] stackArray;
    private int top;      //top of the stack

    public StackX(int s) { // constructor

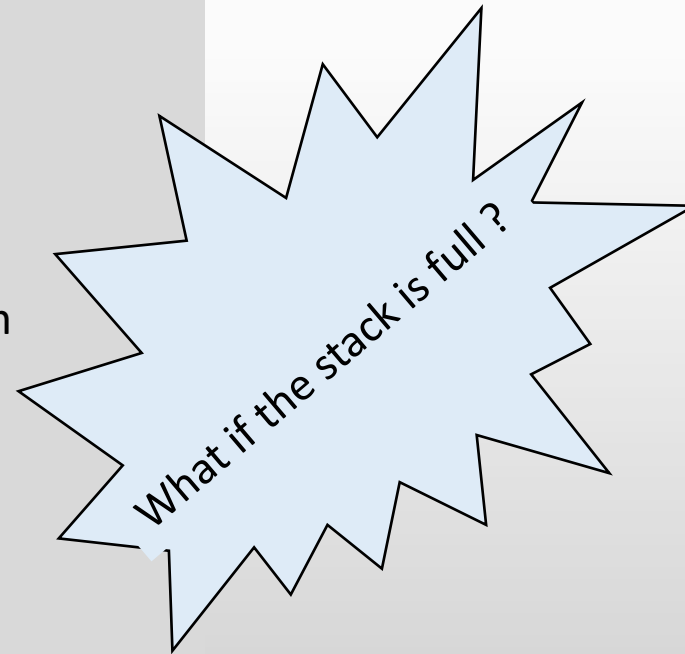
        maxSize = s;      // set array size
        stackArray = new double[maxSize];
        top = -1;          // no items
    }

    public void push(double j) {

        // increment top
        // insert item
    }
}
```

Stack – Implementation - push

```
class StackX {  
    private int maxSize; // size of stack array  
    private double[] stackArray;  
    private int top;      //top of the stack  
  
    public StackX(int s) { // constructor  
  
        maxSize = s;      // set array size  
        stackArray = new double[maxSize];  
        top = -1;          // no items  
    }  
    public void push(double j) {  
  
        // increment top. insert item  
        stackArray[++top] = j;  
    }  
}
```



Stack – Implementation - push

```
class StackX
{
    private int maxSize; // size of stack array
    private double[] stackArray;
    private int top;      //top of the stack

    public StackX(int s) { // constructor

        maxSize = s;      // set array size
        stackArray = new double[maxSize];
        top = -1;          // no items
    }

    public void push(double j) {

        // check whether stack is full
        if (top == maxSize - 1)
            System.out.println("Stack is full");
        else
            stackArray[++top] = j;
    }
}
```


Stack – Implementation – pop/peek

```
class StackX
{
    private int maxSize; // size of stack array
    private double[] stackArray;
    private int top; //top of the stack

    public StackX(int s) { // constructor

        maxSize = s; // set array size
        stackArray = new double[maxSize];
        top = -1; // no items
    }

    public void push(double j) {

        // check whether stack is full
        if (top == maxSize - 1)
            System.out.println("Stack is full");
        else
            stackArray[++top] = j;
    }
}
```

```
public double pop() {
    // check whether stack is empty
    // if not
    // access item and decrement top
}

public double peek() {

    // check whether stack is empty
    // if not
    // access item
}
```

Stack – Implementation – pop/peek

```
class StackX {  
    private int maxSize; // size of stack array  
    private double[] stackArray;  
    private int top; //top of the stack  
  
    public StackX(int s) { // constructor  
  
        maxSize = s; // set array size  
        stackArray = new double[maxSize];  
        top = -1; // no items  
    }  
    public void push(double j) {  
  
        // check whether stack is full  
        if (top == maxSize - 1)  
            System.out.println("Stack is full");  
        else  
            stackArray[++top] = j;  
    }  
}
```

```
    public double pop() {  
        if (top == -1)  
            return -99;  
        else  
            return stackArray[top--];  
    }  
  
    public double peek() {  
        if (top == -1)  
            return -99;  
        else  
            return stackArray[top];  
    }  
}
```

Question

isEmpty() method returns true if the stack is empty and isFull() method return true if the Stack is full.

Implement isEmpty() and isFull() methods of the stack class.

Creating a stack

Question

Using the implemented StackX class, Write a program to create a stack with maximum size 10 and insert the following items to the stack.

30 80 100 25

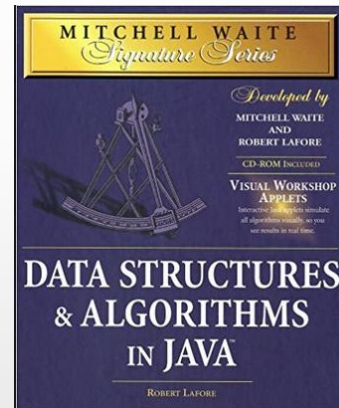
Delete all the items from the stack and display the deleted items.

Creating a stack

```
class StackApp {  
    public static void main(String[] args) {  
        StackX theStack = new StackX(10); // create a stack with max size 10  
  
        theStack.push(30); // insert given items  
        theStack.push(80);  
        theStack.push(100);  
        theStack.push(25);  
  
        while( !theStack.isEmpty() ) { // until it is empty, delete item from stack  
  
            double val = theStack.pop();  
            System.out.print(val);  
            System.out.print(" ");  
        }  
    }  
} // end of class
```

References

1. Mitchell Waite, Robert Lafore, Data Structures and Algorithms in Java, 2nd Edition, Waite Group Press, 1998.

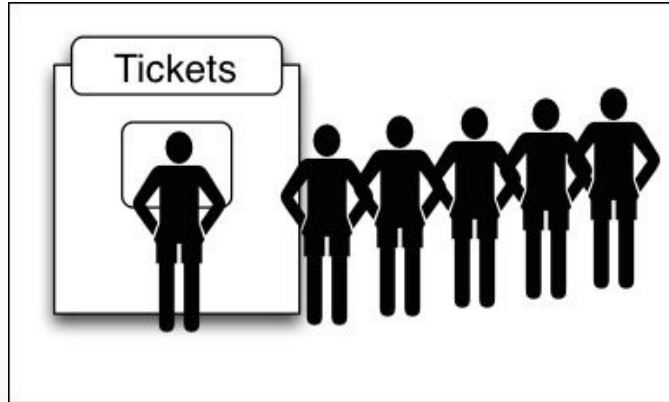


IT2070 – Data Structures and Algorithms

Lecture 02

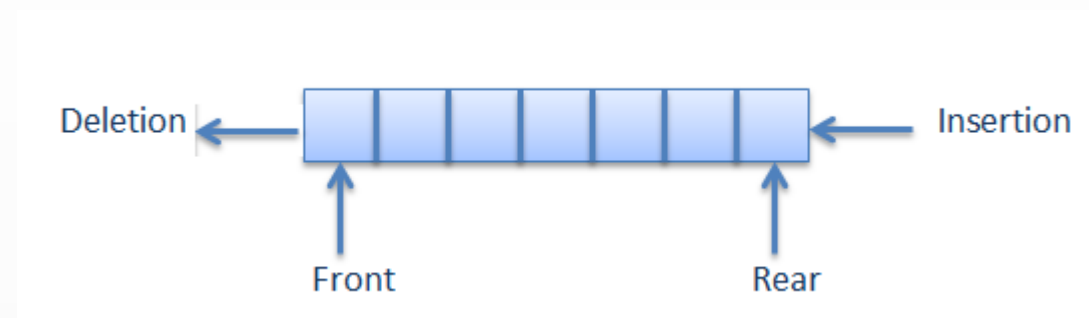
Introduction to Queue

Queues



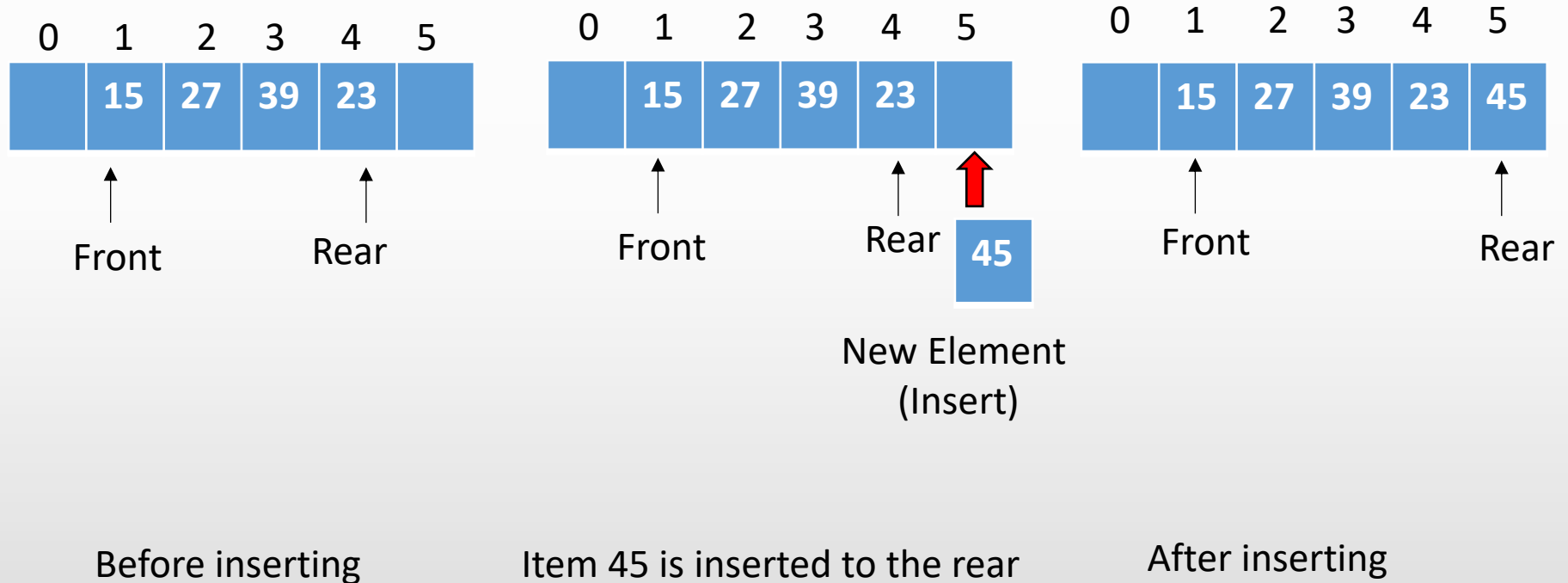
- Imagine a queue in real life
- The first item inserted is the first item to be removed

Queues

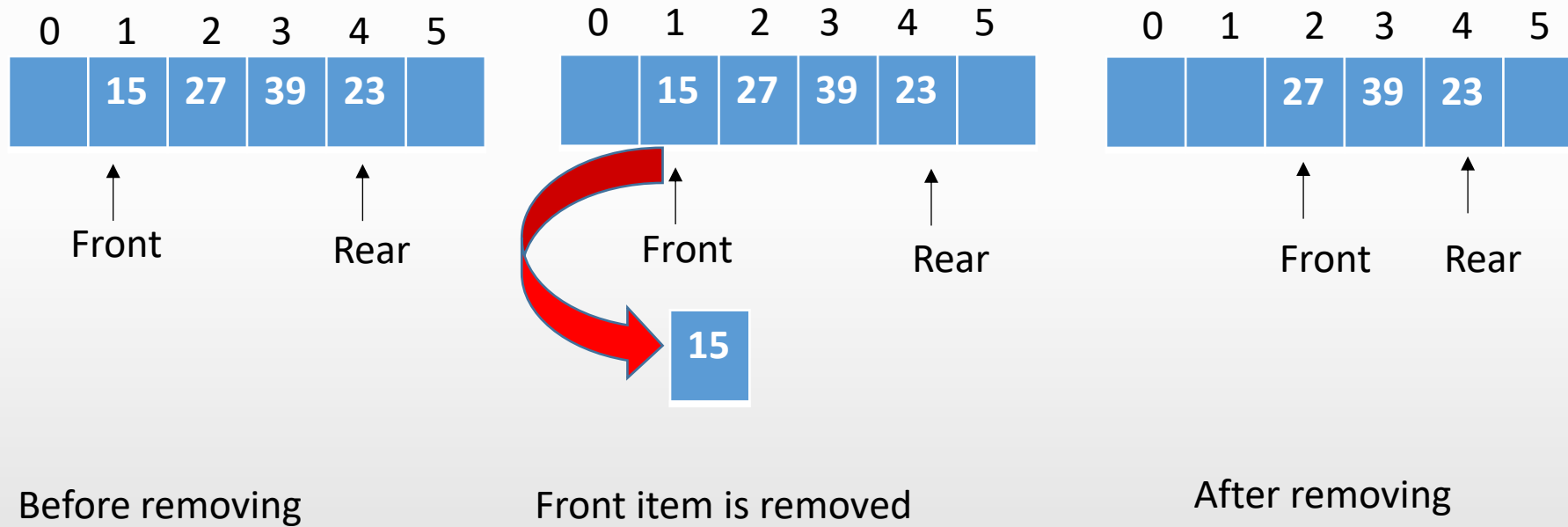


- In a queue all insertions are made at the **Rear** end and deletions are made at the **Front** end.
- Insertions and deletions are restricted from the Middle of the Queue.
- Adding an item is called **insert**
- Removing an item is called **remove**
- The elements are inserted and removed according to the **First-In-First-Out (FIFO)** principle.

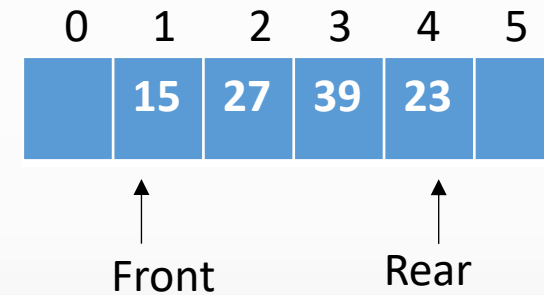
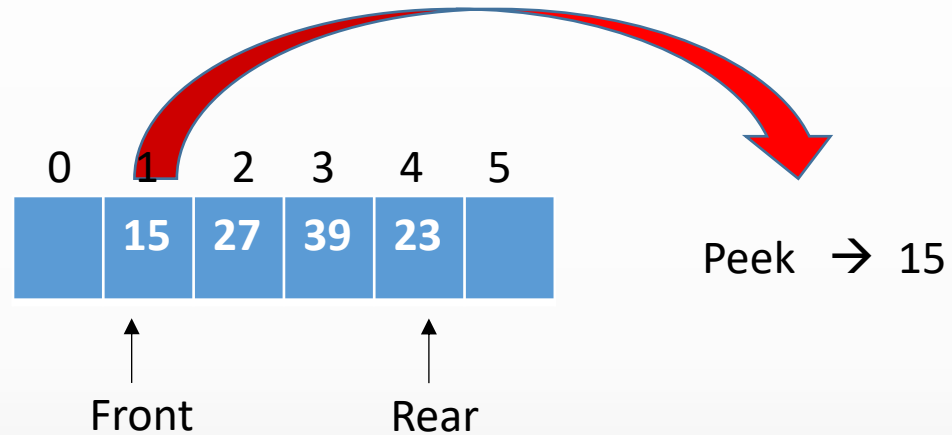
Queue - Insert



Queue - Remove



Queue - PeekFront

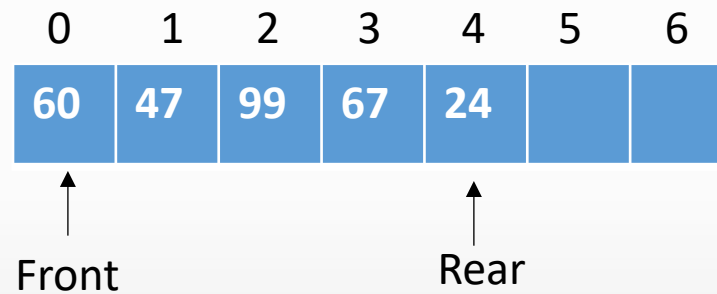


Queue remains the same

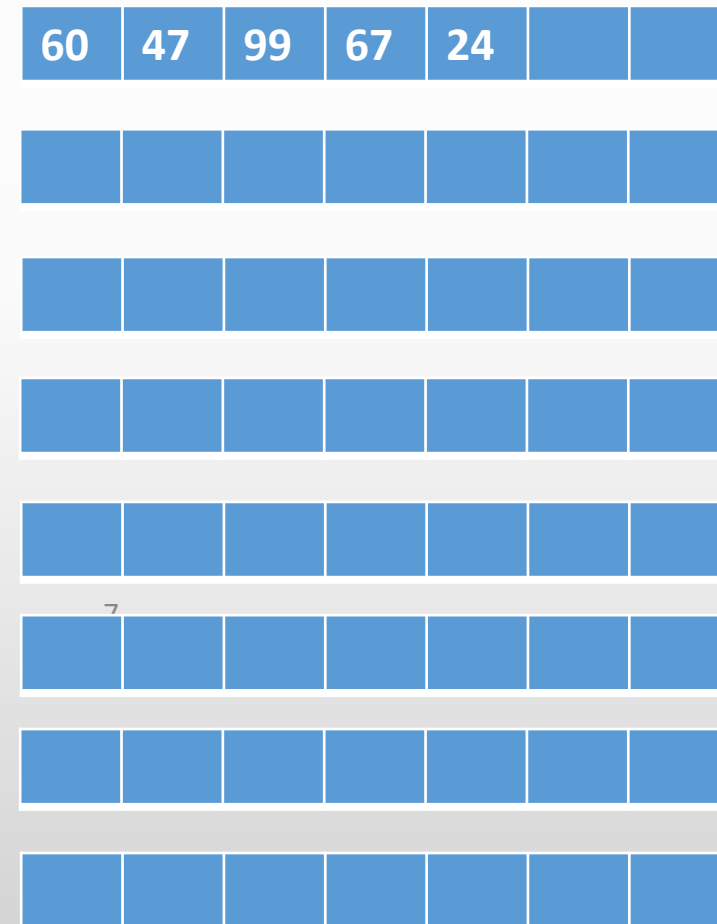
Peek is used to read the value from the Front of the queue without removing it. You can peek only the Front item, all the other items are invisible to the queue user.

Question 01

Draw the Queue frame after performing the below operations to the queue given below.



- i) Insert item 33
- ii) Insert item 53
- iii) peekFront
- iv) remove
- v) remove
- vi) remove
- vii) remove
- viii) remove



Uses of Queue

- There are various queues quietly doing their job in a computer's operating system.
 - printer queue
 - stores keystroke data as you type at the keyboard
 - pipeline

Queue - Implementation

Queue implementation using an **array** with restricted access

- Constructor creates a new Queue of a size specified in its argument.
- Variable **front**, which stores the index of the item on the front of the queue.
- Variable **rear**, which stores the index of the item on the end of the queue.
- Variable *nItems*, which stores the total number of the items in the queue.


```
class QueueX {  
    private int maxSize; // size of queue array  
    private int [] queArray;  
    private int front;    //front of the queue  
    private int rear;     //rear of the queue  
    private int nItems; //no of items of the queue  
  
    public QueueX (int s) { // constructor  
  
        maxSize = s;    // set array size  
        queArray = new int[maxSize];  
        front = 0;  
        rear = -1;  
        nItems = 0;      // no items  
    }  
    .....  
    .....  
}
```

Queue – Implementation - insert

```
class QueueX {  
    private int maxSize; // size of queue array  
    private int [] queArray;  
    private int front;    //front of the queue  
    private int rear;     //rear of the queue  
    private int nItems; //no of items of the queue  
  
    public QueueX(int s) { // constructor  
  
        maxSize = s;    // set array size  
        queArray = new int [maxSize];  
        front = 0;  
        rear = -1;  
        nItems = 0;      // no items  
    }  
    public void insert(int j) {  
        // increment rear  
        // insert an item  
    }  
}
```


Queue – Implementation - insert

```
class QueueX {  
    private int maxSize; // size of queue array  
    private int [] queArray;  
    private int front;    //front of the queue  
    private int rear;    //rear of the queue  
    private int nItems; //no of items of the queue  
  
    public QueueX(int s) { // constructor  
  
        maxSize = s;    // set array size  
        queArray = new int [maxSize];  
        front = 0;  
        rear = -1;  
        nItems = 0;    // no items  
    }  
  
    public void insert(int j) {  
        // increment rear and insert an item  
        queArray[++rear] = j;  
        nItems++;  
    }  
}
```



What if the queue is full ?

Queue – Implementation - insert

```
class QueueX {  
    private int maxSize; // size of queue array  
    private int [] queArray;  
    private int front;    //front of the queue  
    private int rear;    //rear of the queue  
    private int nItems; //no of items of the queue  
  
    public QueueX(int s) { // constructor  
  
        maxSize = s;    // set array size  
        queArray = new int [maxSize];  
        front = 0;  
        rear = -1;  
        nItems = 0;    // no items  
    }  
}
```

```
    public void insert(int j) {  
        // check whether queue is full  
        if (rear == maxSize - 1)  
            System.out.println("Queue is full");  
        else {  
            queArray[++rear] = j;  
            nItems++;  
        }  
    }  
}
```

Queue – Implementation – remove/peekFront

```
class QueueX {  
    private int maxSize; // size of queue array  
    private int [] queArray;  
    private int front;    //front of the queue  
    private int rear;    //rear of the queue  
    private int nItems; //no of items of the queue  
  
    public QueueX(int s) { // constructor  
  
        maxSize = s;    // set array size  
        queArray = new int [maxSize];  
        front = 0;  
        rear = -1;  
        nItems = 0;    // no items  
    }  
    public void insert(int j) {  
  
        // check whether queue is full  
        if (rear == maxSize - 1)  
            System.out.println("Queue is full");  
        else {  
  
            queArray[++rear] = j;  
            nItems++;  
  
        }  
    }  
}
```

```
public int remove() {  
    // check whether queue is empty  
    // if not  
    // access item and increment front  
}
```

```
public int peekFront() {  
    // check whether queue is empty  
    // if not  
    // access item  
}
```

Queue – Implementation – remove

```
class QueueX{
    private int maxSize; // size of queue array
    private int [] queArray;
    private int front; //front of the queue
    private int rear; //rear of the queue
    private int nItems; //no of items of the queue

    public QueueX(int s) { // constructor

        maxSize = s; // set array size
        queArray = new int [maxSize];
        front = 0;
        rear = -1;
        nItems = 0; // no items
    }

    public void insert(int j) {

        // check whether queue is full
        if (rear == maxSize - 1)
            System.out.println("Queue is full");
        else {

            queArray[++rear] = j;
            nItems++;
        }
    }
}
```

```
public int remove() {
    if (nItems == 0) {
        System.out.println("Queue is empty");
        return -99;
    }
    else {
        nItems--;
        return queArray[front++];
    }
}
```

Queue – Implementation – peekFront

```
class QueueX{
    private int maxSize; // size of queue array
    private int [] queArray;
    private int front; //front of the queue
    private int rear; //rear of the queue
    private int nItems; //no of items of the queue

    public QueueX(int s) { // constructor

        maxSize = s; // set array size
        queArray = new int [maxSize];
        front = 0;
        rear = -1;
        nItems = 0; // no items
    }

    public void insert(int j) {

        // check whether queue is full
        if (rear == maxSize - 1)
            System.out.println("Queue is full");
        else {

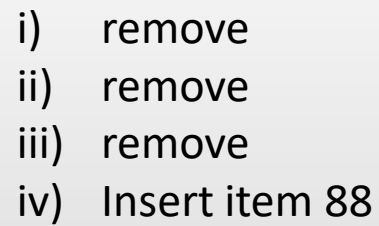
            queArray[++rear] = j;
            nItems++;
        }
    }
}
```

```
public int peekFront() {
    if (nItems == 0) {
        System.out.println("Queue is empty");
        return -99;
    }
    else {
        return queArray[front];
    }
}
```

Question 02

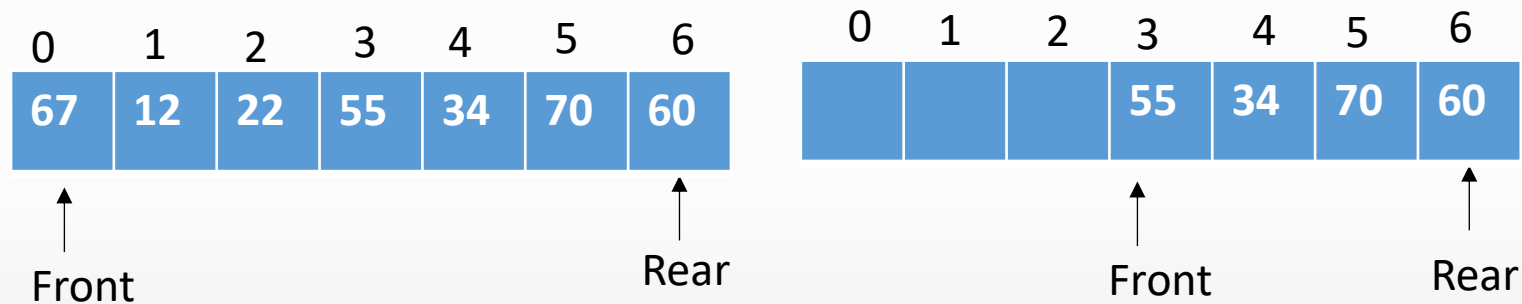
isEmpty() method of the Queue class returns true if the Queue is empty and isFull() method returns true if the Queue is full.

Implement isEmpty() and isFull() methods of the Queue class.



Question 03 Contd..

Draw the Queue frame after performing the below operations to the queue given below.



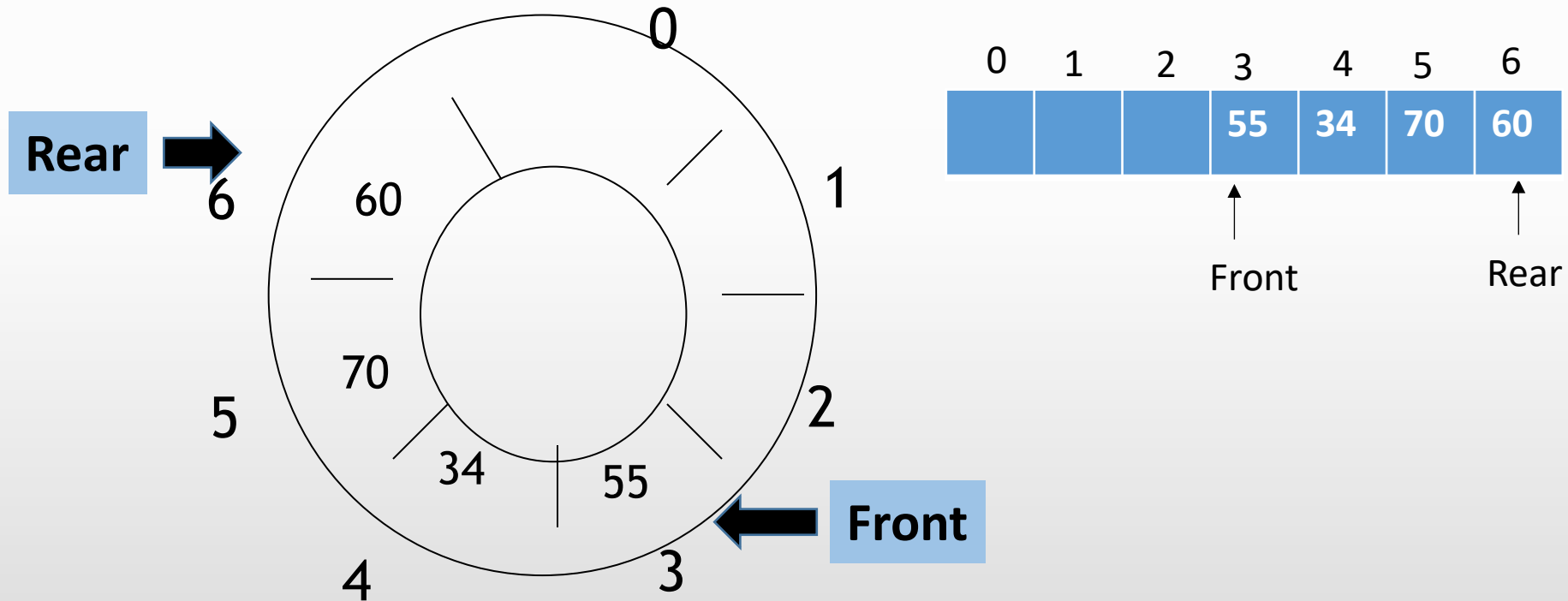
- i) remove
- ii) remove
- iii) remove
- iv) Insert item 88

Although the queue is not full we cannot insert more elements.

Any Suggestions?

How to overcome this situation??

We can use a Circular Queue



Circular Queue

- Circular queues are queues that wrap around themselves.
- These are also called ring buffers.
- The problem in using the linear queue can be overcome by using circular queue.
- When we want to insert a new element we can insert it at the beginning of the queue.

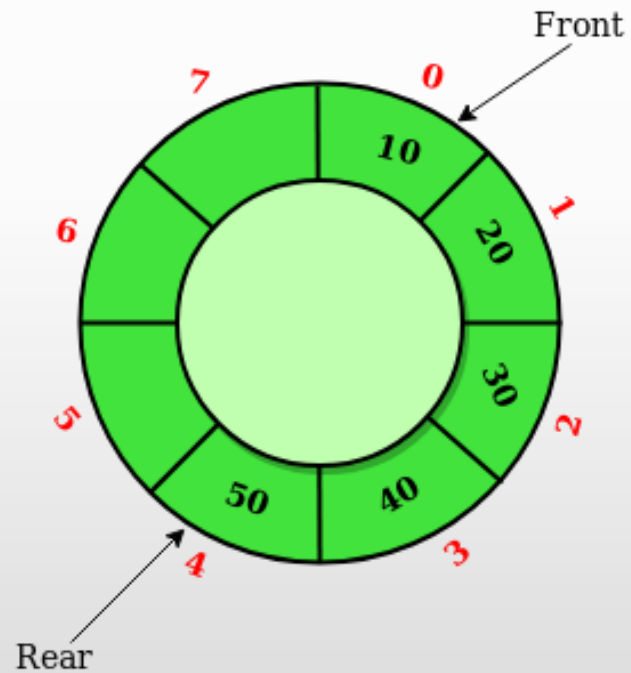
i.e. if the queue is not full we can make the rear start from the beginning by wrapping around

If rear was 3 then the next element should be stored in index 4

If rear was 7 then the next element should be stored in index 0

Question 04

Draw the Queue frame after performing the below operations to the circular queue given below.



- i) insert(14);
- ii) insert(29);
- iii) insert(33);
- iv) insert(88);
- v) peekFront();
- vi) remove();
- vii) remove();
- viii) insert(90);
- ix) insert(100);
- x) peekFront();

Inserting an element to Linear Queue

```
public void insert(int j) {  
    // check whether queue is full  
    if ( rear == maxSize - 1)  
        System.out.println("Queue is full");  
    else {  
        queArray[++rear] = j;  
        nItems++;  
    }  
}
```

Inserting an element to Circular Queue

```
public void insert(int j) {  
    // check whether queue is full  
    if (nItems == maxSize)  
        System.out.println("Queue is full");  
    else {  
        if(rear == maxSize - 1)  
            rear = -1;  
  
        queArray[++rear] = j;  
  
        nItems++;  
    }  
}
```

Removing an element from Linear Queue

```
public int remove() {  
    // check whether queue is empty  
    if ( nItems == 0)  
        System.out.println("Queue is empty");  
    else {  
        nItems--;  
        return queArray[front++];  
    }  
}
```

Removing an element from Circular Queue

```
public int remove() {  
    // check whether queue is empty  
    if ( nItems == 0)  
        System.out.println("Queue is empty");  
    else {  
        int temp = queArray[front++];  
        if (front == maxSize)  
            front = 0;  
  
        nItems--;  
        return temp;  
    }  
}
```

Question 05

Implement `isFull()`, `isEmpty()` and `peekFront()` methods of the Circular Queue class.

Question 06

Creating a Queue

Using the implemented QueueX class, Write a program to create a queue with maximum size 10 and insert the following items to the queue.

10 25 55 65 85

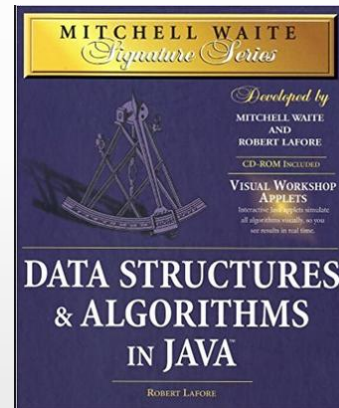
Delete all the items from the queue and display the deleted items.

Creating a Queue

```
class QueueApp {  
    public static void main(String[] args) {  
        QueueX theQueue = new QueueX(10); // create a queue with max size 10  
  
        theQueue.insert(10); // insert given items  
        theQueue.insert(25);  
        theQueue.insert(55);  
        theQueue.insert(65);  
        theQueue.insert(85);  
  
        while( !theQueue.isEmpty() ) { // until it is empty, delete item from queue  
  
            int val = theQueue.remove();  
            System.out.print(val);  
            System.out.print(" ");  
        }  
    }  
} // end of class
```


References

1. Mitchell Waite, Robert Lafore, Data Structures and Algorithms in Java, 2nd Edition, Waite Group Press, 1998.



IT2070 – Data Structures and Algorithms

Lecture 03

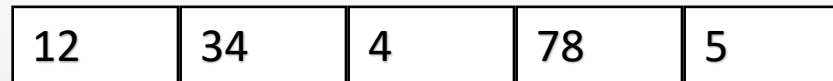
Linked Lists

Ways in which linked lists differ from arrays

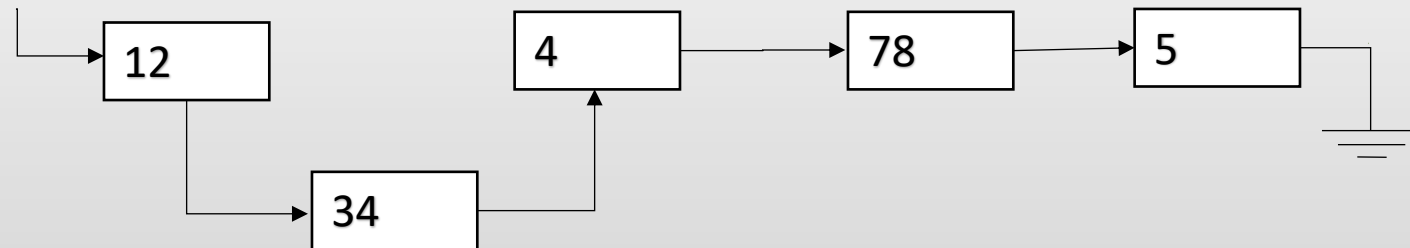
Array – each item occupies a particular position and can be directly accessed using an index number.

Linked list – need to follow along the chain of element to find a particular element. A data item cannot be accessed directly.

Array →



Linked List →



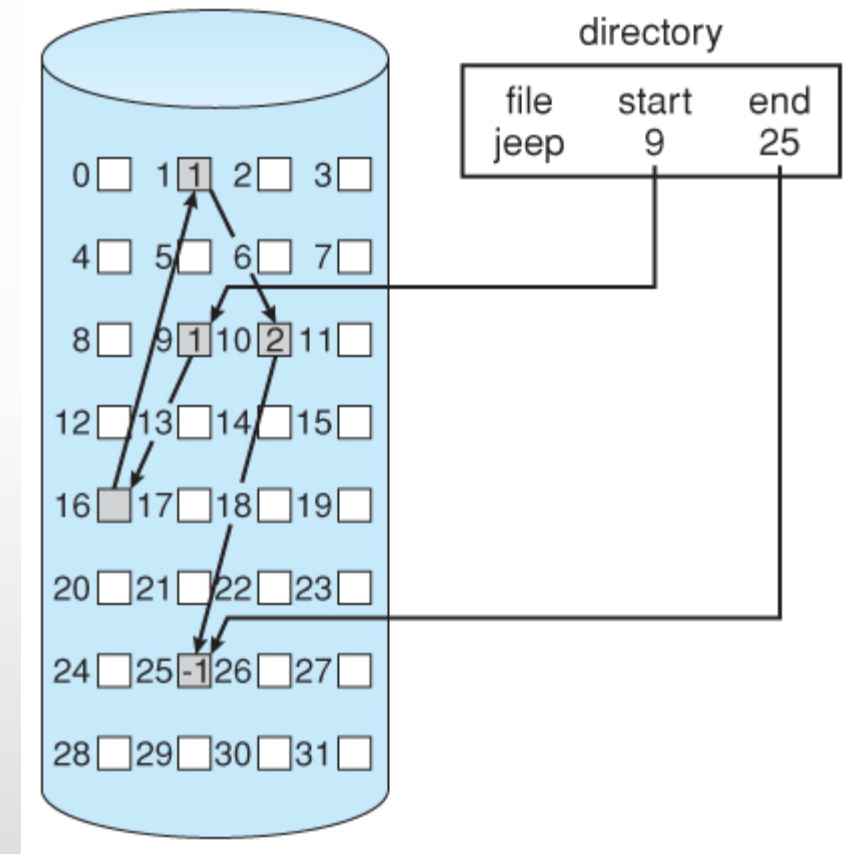
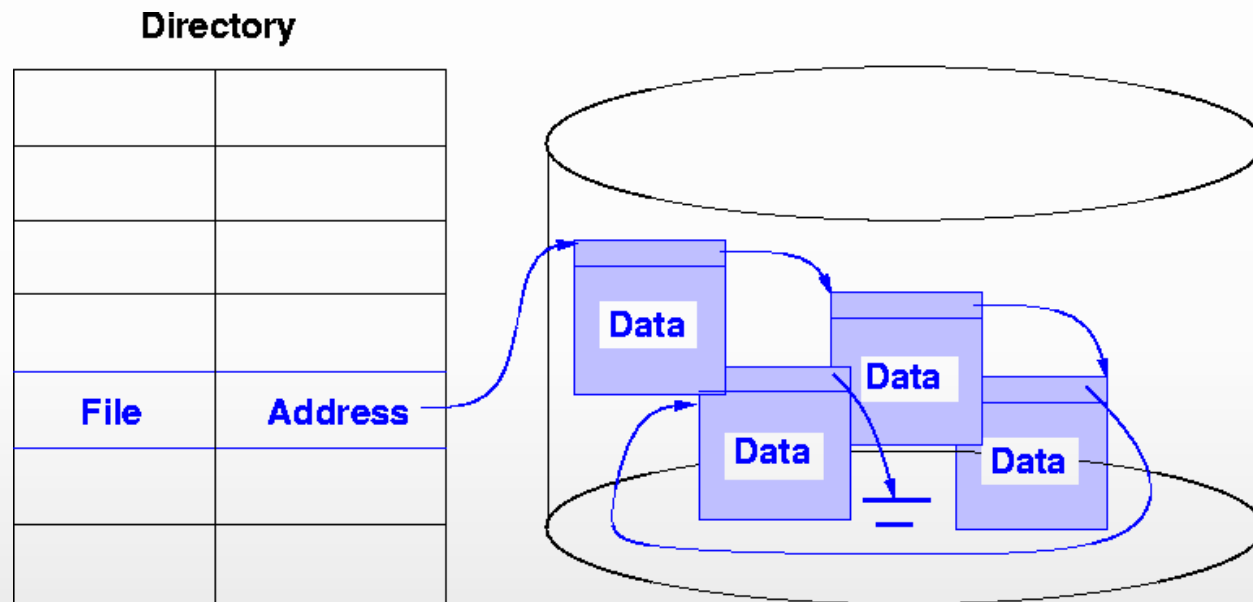
Applications of linked list in real world-

- *Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.
- *Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- *Music Player* – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Applications of linked list in computer science –

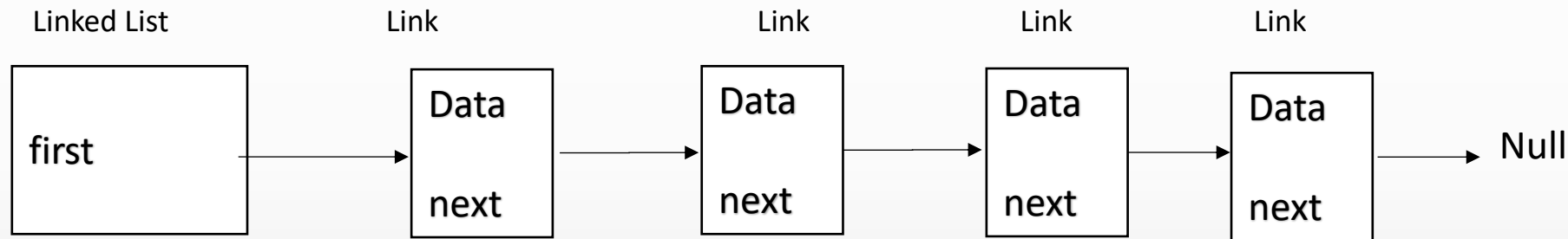
- Implementation of stacks and queues
- Implementation of graphs : Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names

Linked Allocation in File System



Linked List

Linked lists are probably the second most commonly used general purpose storage structures after arrays.



- In a linked list each data item is embedded in a link.
- There are many similar links.
- Each link object contains a reference to the next link in the list.
- In a typical application there would be many more data items in a link.

Operations

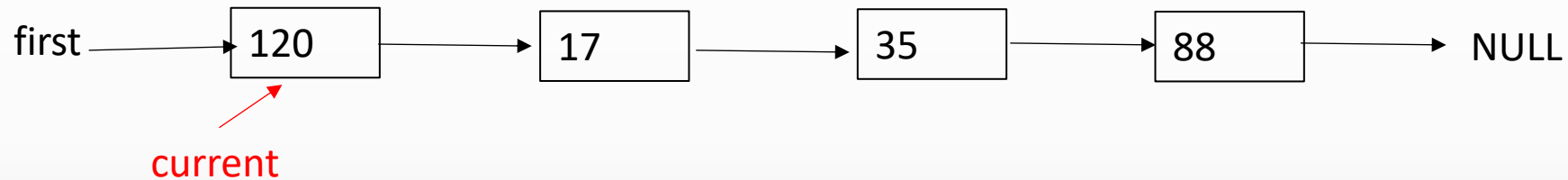
- Mainly the following operations can be performed on a linked list.
 - Find
Find a link with a specified key value.
 - Insert
Insert links anywhere in the list.
 - Delete
Delete a link with the specified value.

Operations - Find

Start with the first item, go to the second link, then the third, until you find what you are looking for.

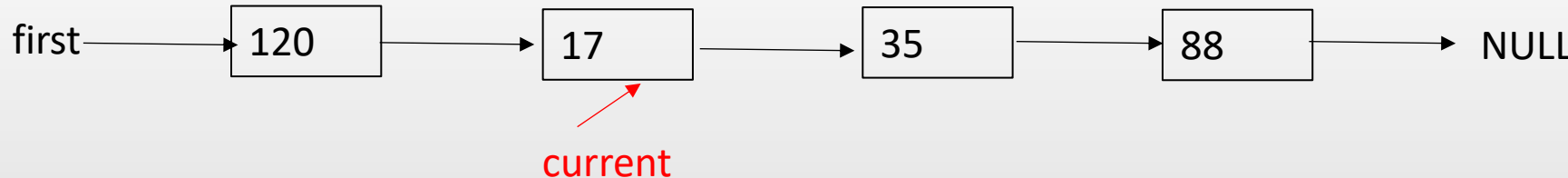
Ex: Find Item 35

Step 1 :



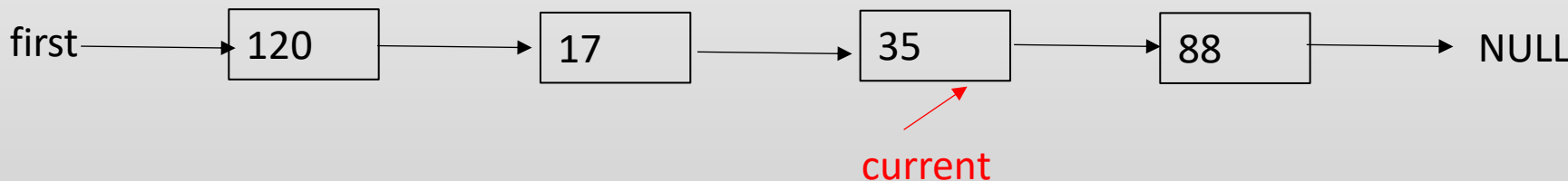
**Found ?
No**

Step 2 :



**Found ?
No**

Step 3 :

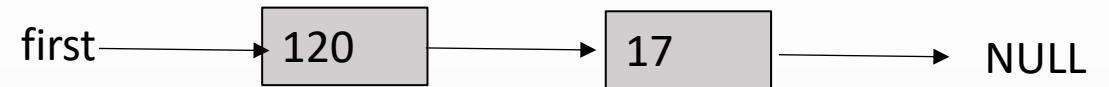


**Found ?
Yes**

Operations – Insert

Inserting an item at the beginning of the list

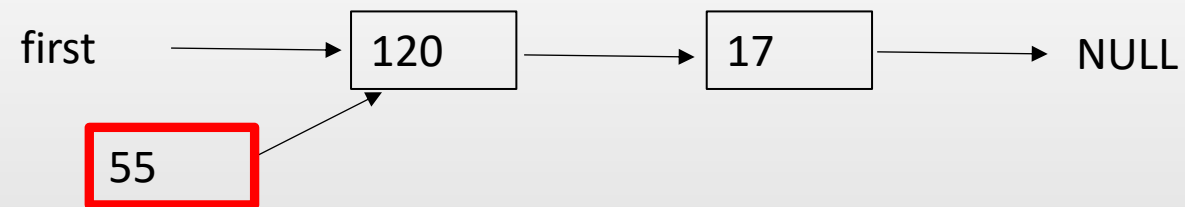
Before inserting



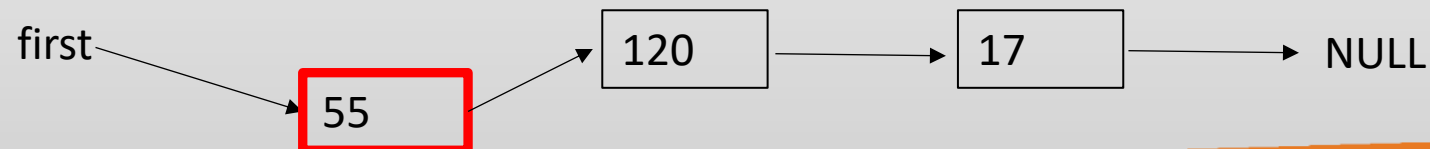
Step 1 : create a new link

55

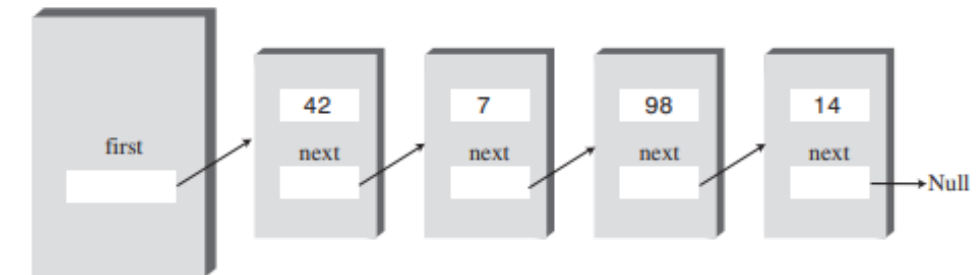
Step 2 : 'next' field of the new link points to the old first link



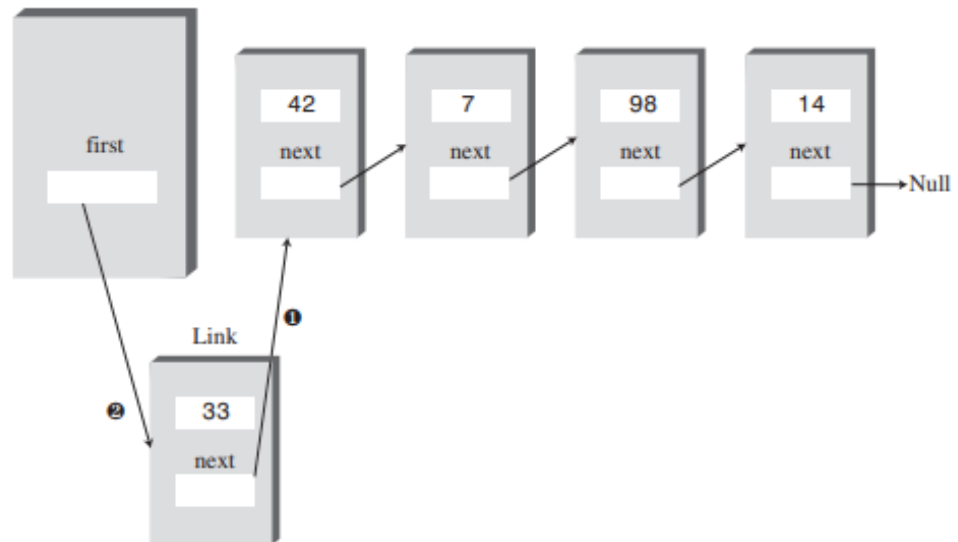
Step 3 : 'first' points to the newly created link



InsertFirst()



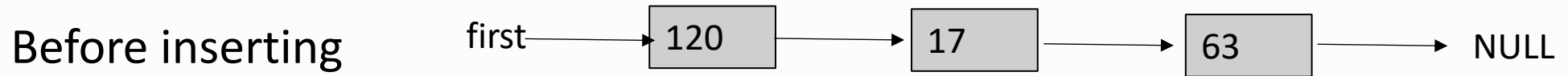
a) Before Insertion



b) After Insertion

Operations - Insert

Inserting an item in the middle of the list



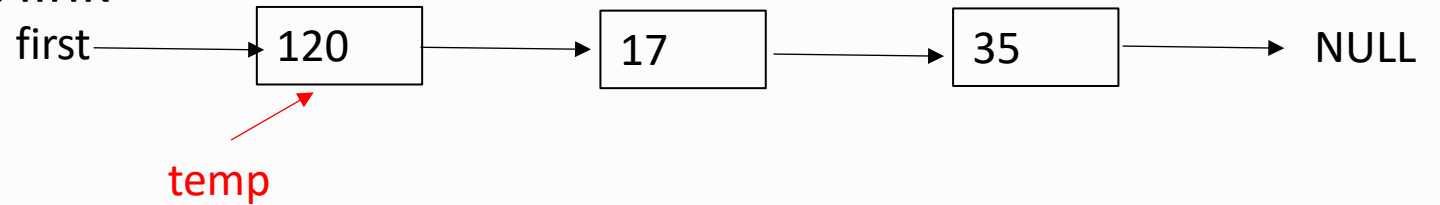
Question:

What steps need to be followed if a new link is inserted after the link '17' ?

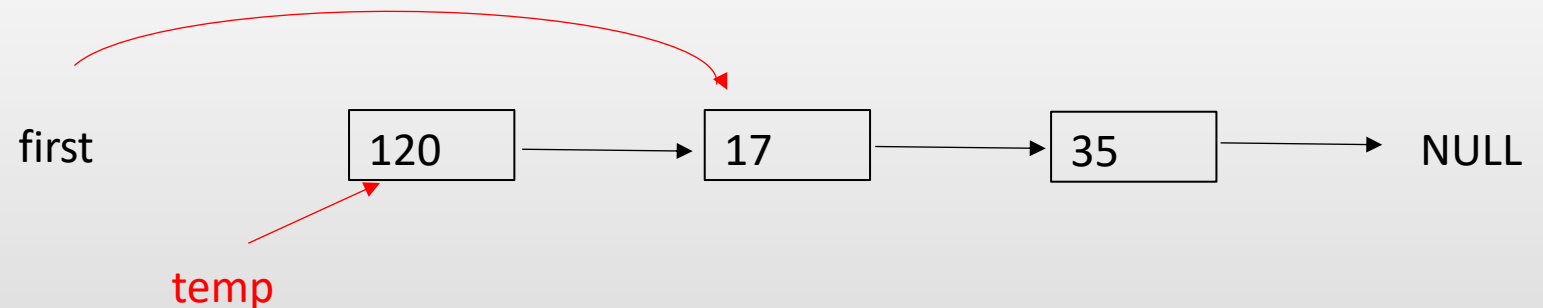
Operations - Delete

Deleting an item from the beginning of the list

Step 1 : Save reference to first link

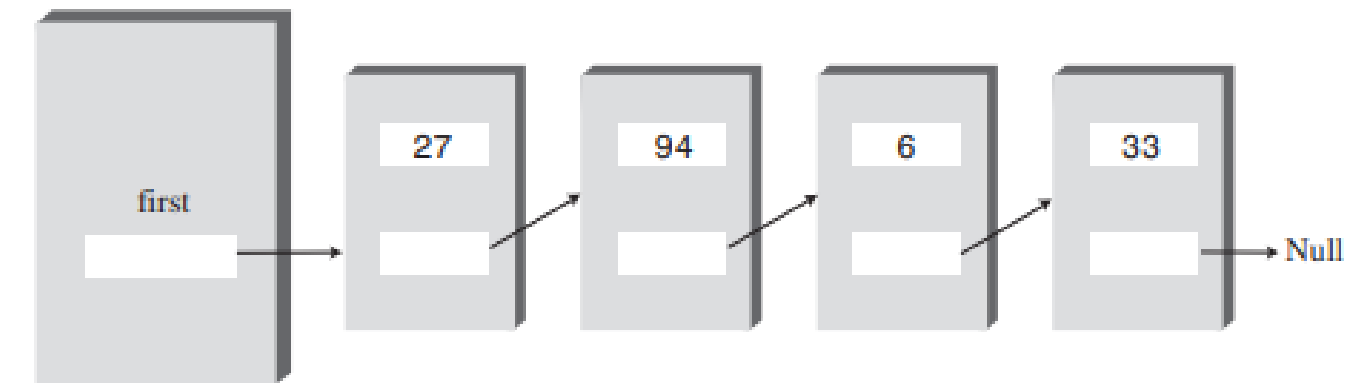


Step 2 : Disconnect the first link by rerouting first to point to the second link

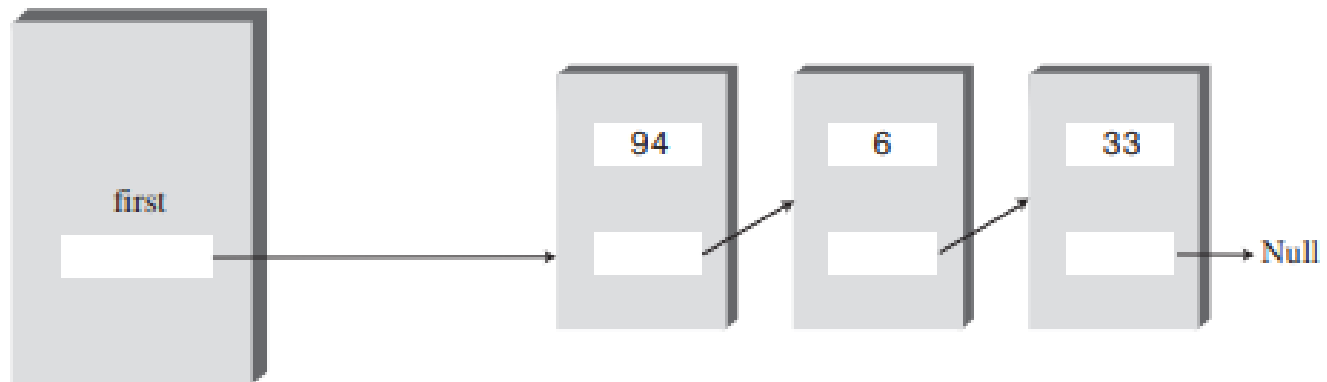


Step 3 : Return the deleted link (temp)

deleteFirst()



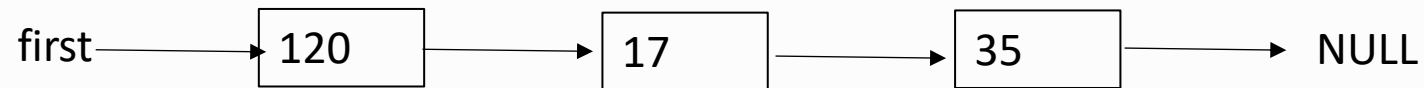
a) Before Deletion



b) After Deletion

Operations - Delete

Deleting a given item from the list



Question:

What steps need to be followed to delete the link '17' ?

Linked List - Implementation

Link Class

- In a linked list, a link is an object of a class called something like “**Link**”.
- There are many similar links in a linked list.
- Each link contains Data Items and a reference to the next link in the list.

```
class Link {  
    public int iData; // data item  
    public Link next; // reference to the next link  
  
    public Link(int id) { // constructor  
  
        iData = id;  
        next = null;  
    }  
    public void displayLink() { // display data item  
  
        System.out.println(iData);  
    }  
}
```


Linked List - Implementation

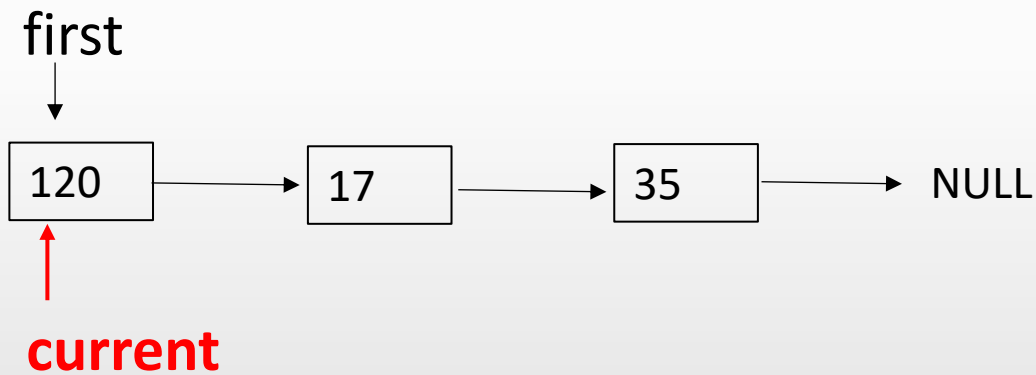
Link List Class

- The LinkedList class contains only one data item, a reference to the first link on the list called **'first'**.
- It is possible to find the other links by following the chain of references from **'first'**, using each link's next field.

```
class LinkedList {  
    private Link first;  
  
    public LinkedList() {    //constructor  
  
        first = null;  
    }  
    public boolean isEmpty() { // true if list is empty  
  
        return (first == null);  
    }  
    // .....          other methods  
}
```

Linked List - Implementation

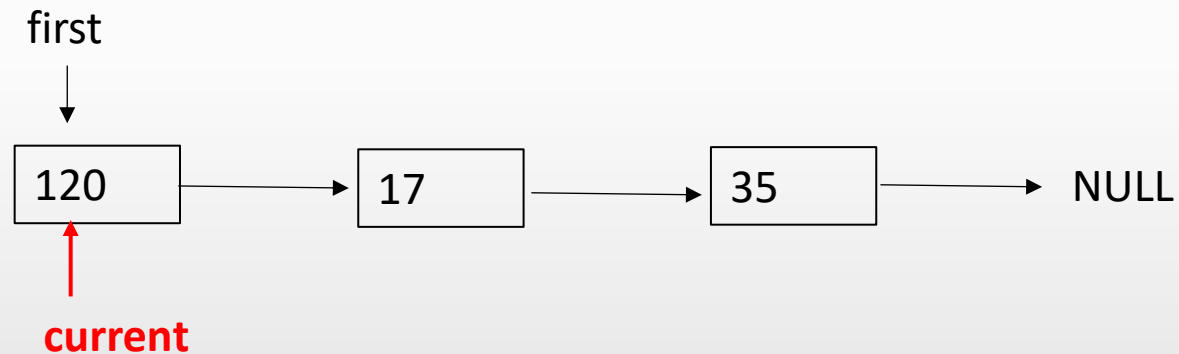
Link List Class – Contd.



```
class LinkedList {  
    private Link first;  
    public LinkedList() { //constructor  
        first = null;  
    }  
    public boolean isEmpty() { // true if list is empty  
        return (first == null);  
    }  
  
    public void displayList() {  
        Link current = first;  
        while (current != null) {  
            current.displayLink();  
            current = current.next;  
        }  
        System.out.println(" ");  
    }  
}
```

Linked List - Implementation

Link List Class – Contd.



```
class LinkedList {  
    private Link first;  
    public LinkedList() { //constructor  
        first == null;  
    }  
    public boolean isEmpty() { // true if list is empty  
        return (first == null);  
    }  
    public void displayList() {  
        Link current = first;  
        while (current != null) {  
            current.displayLink();  
            current = current.next;  
        }  
        System.out.println(" ");  
    }  
}
```

Linked List - Implementation

Link List Class – Contd.

```
class LinkedList {  
    private Link first;  
    public LinkedList() { //constructor  
  
        first = null;  
    }  
    public boolean isEmpty() { // true if list is empty  
  
        return (first == null);  
    }  
    public void displayList() {  
  
        Link current = first;  
        while (current != null) {  
  
            current.displayLink();  
            current = current.next;  
        }  
        System.out.println("");  
    }  
}
```

```
// insertFirst Method  
public void insertFirst(int id) {  
    Link newLink = new Link(id);  
    newLink.next = first;  
    first = newLink;  
}
```

```
// deleteFirst Method  
public Link deleteFirst() {  
    Link temp = first;  
    first = first.next;  
    return temp;  
}
```

Question 1

Write a program to

- i) Create a new linked list and insert four new links.
- ii) Display the list.
- iii) Remove the items one by one until the list is empty.

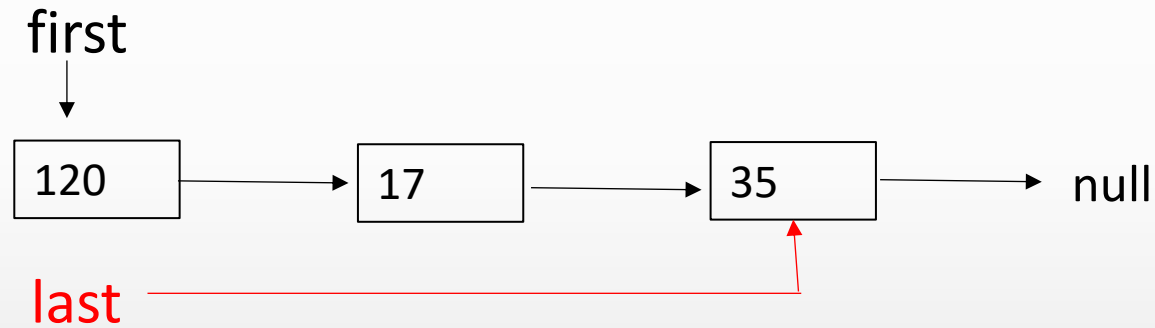
(Use the LinkList class created)

Answer1

```
class myList {  
    public static void main(String[] args)    {  
        LinkedList theList = new LinkedList(); // create a new list  
  
        theList.insertFirst(23); // insert four items  
        theList.insertFirst(89);  
        theList.insertFirst(12);  
        theList.insertFirst(55);  
  
        theList.displayList(); //display the list  
  
        while( !theList.isEmpty() ) { // delete item one by one  
  
            Link aLink = theList.deleteFirst();  
            System.out.print("Deleted ");  
            aLink.displayLink();  
        }  
    }  
}
```

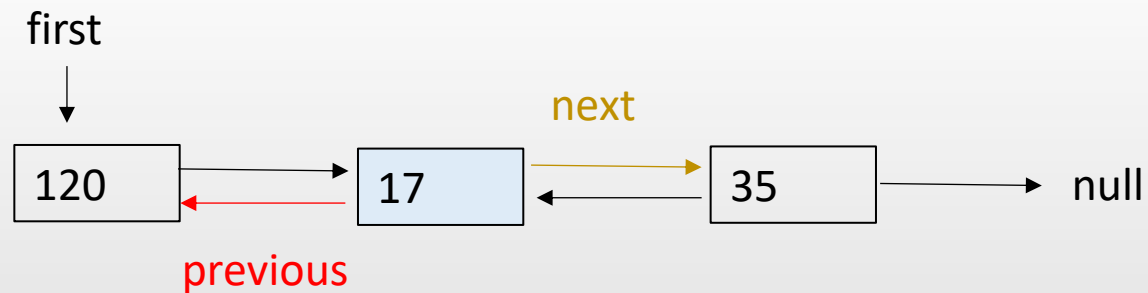
Double-Ended List

A double-ended list is similar to an ordinary linked list with an additional reference to the last link.



Doubly Linked List

A doubly linked list allows to traverse backwards as well as forward through the list. Each link has two references.



References

Mitchell Waite, Robert Lafore, Data Structures and Algorithms in Java, 2nd Edition, Waite Group Press, 1998.

Tree Data Structure

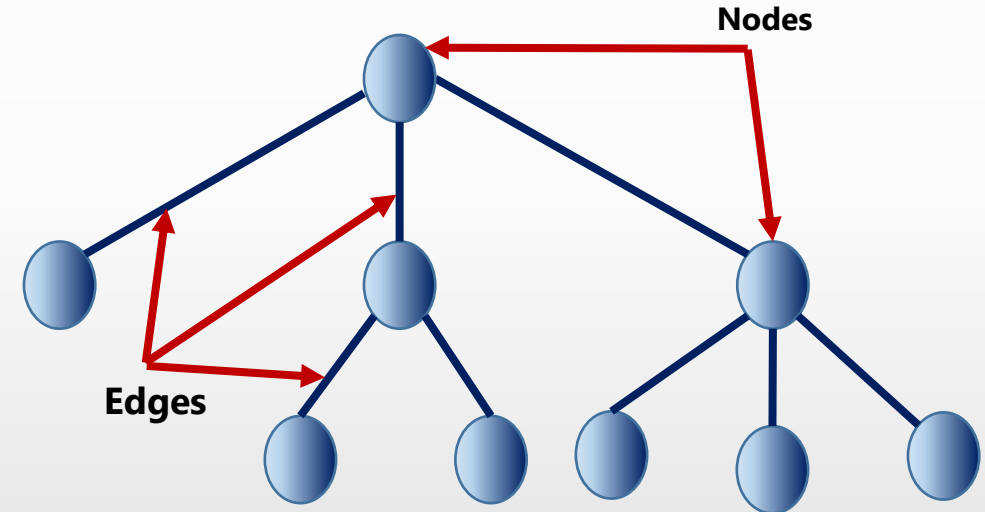
What is a tree?

A tree consist of nodes connected by edges.

In a picture of a tree the nodes are represented as circles and the edges as lines connecting the circles.

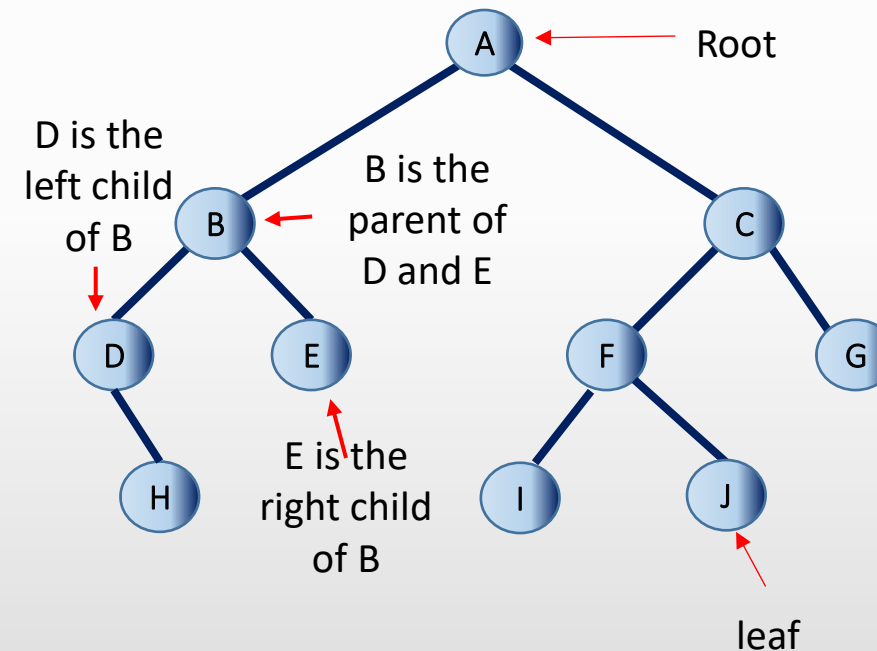
In a tree, the nodes represent the data items and the edges represent the way the nodes are related.

A tree with nodes which has maximum of two children is called a **binary tree**.



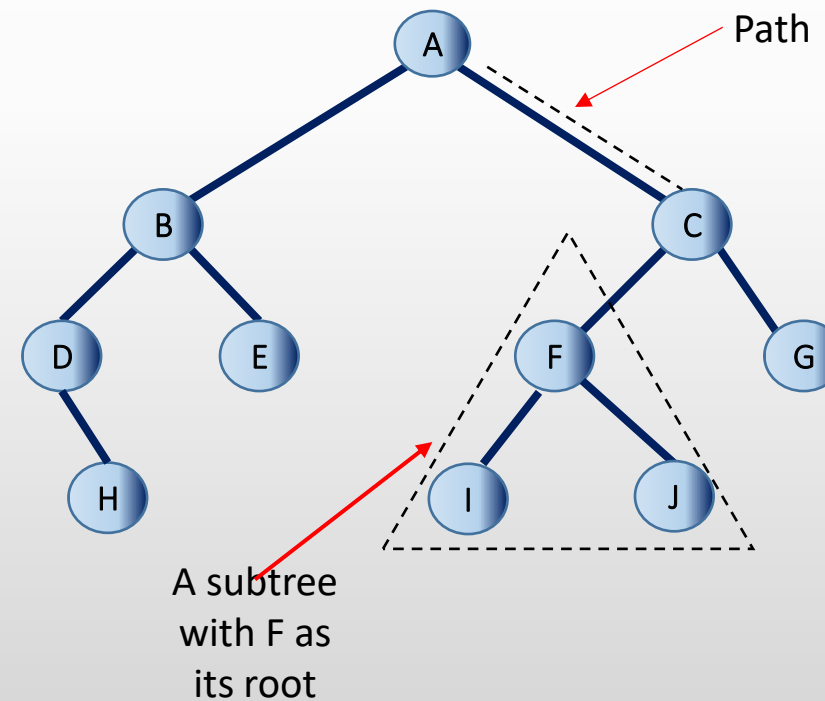
What is root, parent and children in a tree?

- The node at the top of the tree is called root.
- Any node which has exactly one edge running upwards to other node is called a child
- The two children of each node in a binary tree are called left child and right child.
- Any node which has one or more lines running downwards to other nodes is called a parent
- A node that has no children is called a leaf node or leaf



What is path and subtree in a tree

- Sequence of nodes from one node to another along the edges is called a path.
- Any node which consist of its children and it's children's children and so on is called a sub tree



Key value of a node

- Each node in a tree stores objects containing information.
- Therefore one data item is usually designated as a key value.
- The key value is used to search for a item or to perform other operations on it.
- eg: person object – social security number
car parts object– part number

Binary Search Tree

- Binary Search Tree

- is a tree that has at most two children.

- a node's left child must have a key less than its parent and node's right child must have a key greater than or equal to its parent.

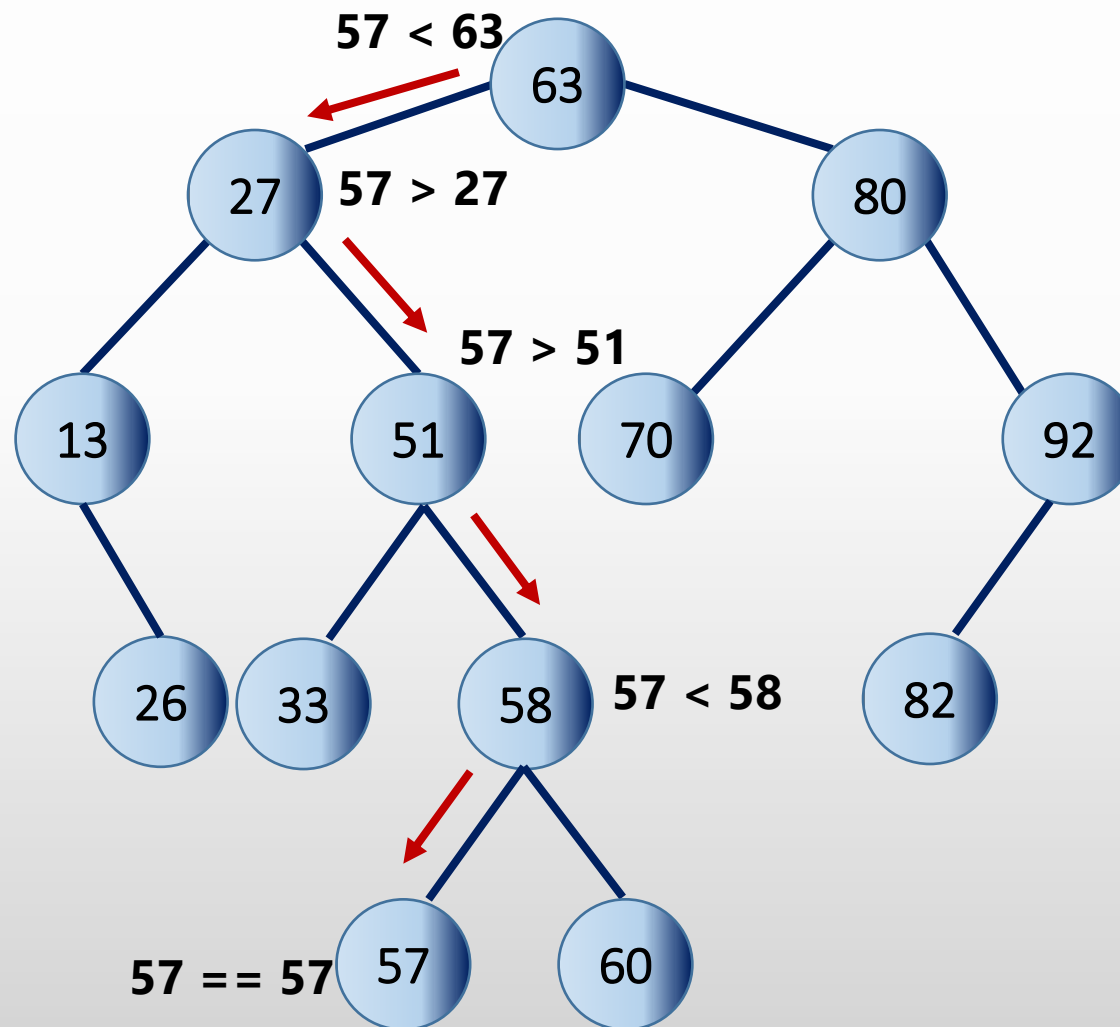
Operations of Binary Search Tree

- There are four main operations perform in a binary search tree
 - Find – find a node with a given key
 - Insert – insert a new node
 - Delete – delete a node
 - Traverse – visit all the nodes

Operations - Find

- Find always start at the root.
- Compare the key value with the value at root.
- If the key value is less, then compare with the value at the left child of root.
- If the key value is higher, then compare with the value at the right child of root
- Repeat this, until the key value is found or reach to a leaf node.

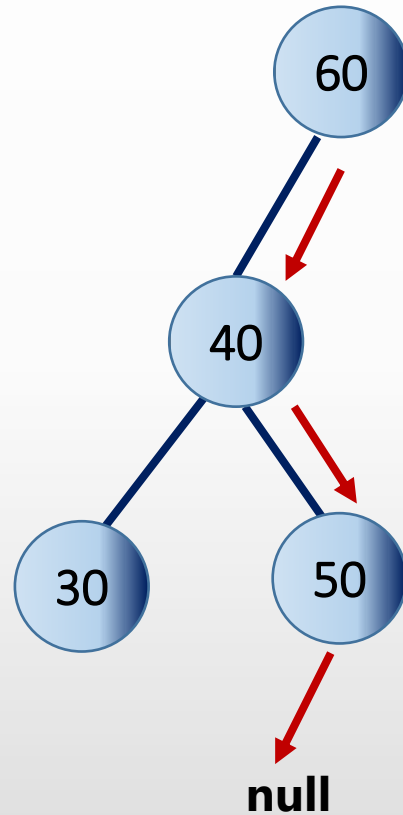
Find value 57



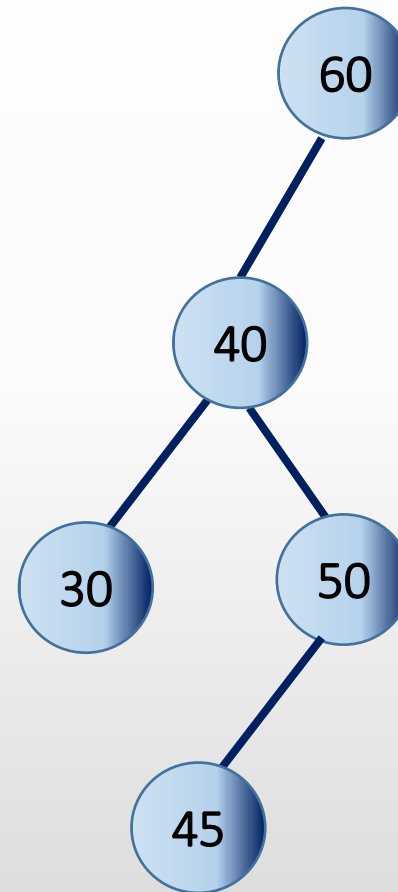
Operations - Insert

- Create a new node.
- Find the place (parent) to insert a new node.
- When the parent is found, the new node is connected as its left or right child, depending on whether the new node's key is less than or greater than that of the parent.

Insert value 45



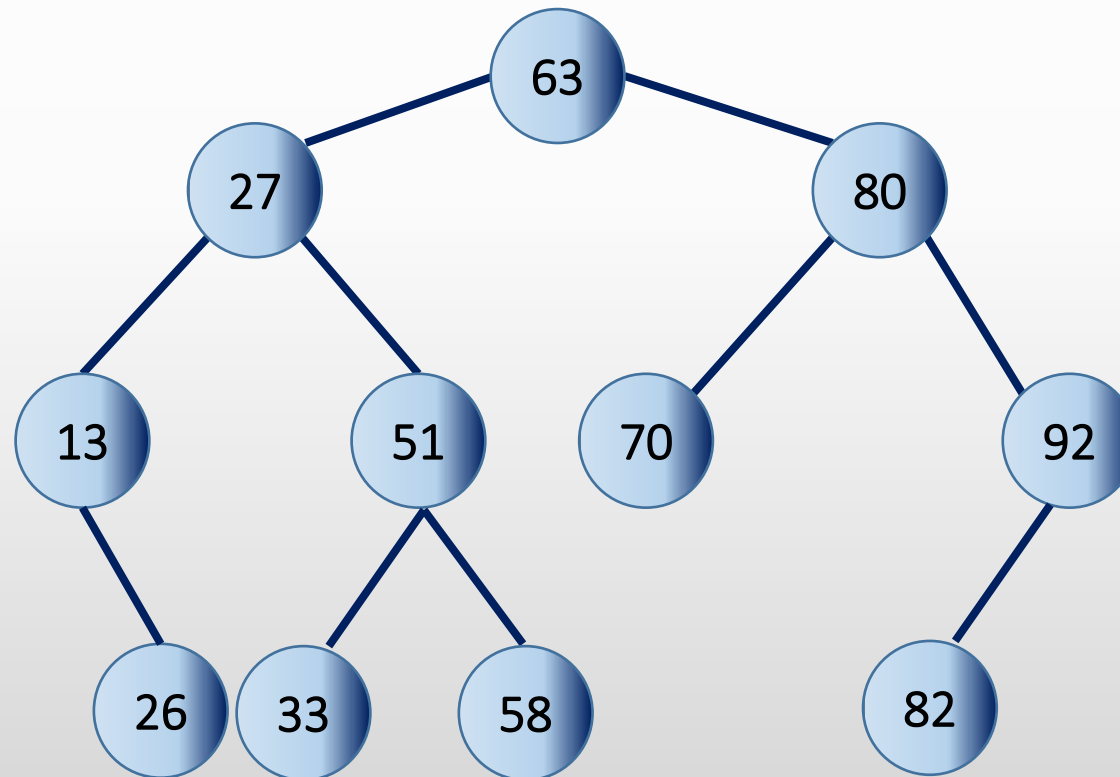
a) Before insertion



b) After insertion

Question 1

- Draw a tree after inserting number 8



BFS vs DFS for Binary Tree

A Tree is typically traversed in two ways:

- Breadth First Traversal (Or Level Order Traversal)
- Depth First Traversals
 1. Inorder Traversal (Left-Root-Right)
 2. Preorder Traversal (Root-Left-Right)
 3. Postorder Traversal (Left-Right-Root)

BFS and DFSs of above Tree

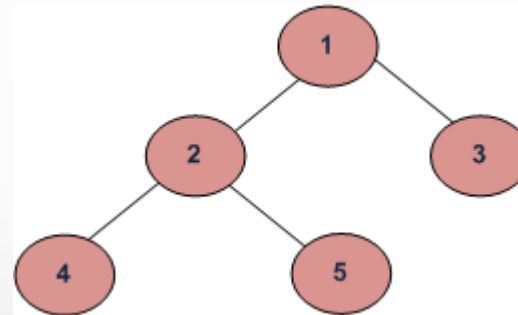
Breadth First Traversal : 1 2 3 4 5

Depth First Traversals:

Preorder Traversal : 1 2 4 5 3

Inorder Traversal : 4 2 5 1 3

Postorder Traversal : 4 5 2 3 1



Traversing

- Traverse a tree means to visit all the nodes in some specified order.
- There are three common ways to traverse a tree
 - Pre order
 - In order
 - Post order

InOrder(root) visits nodes in the following order:

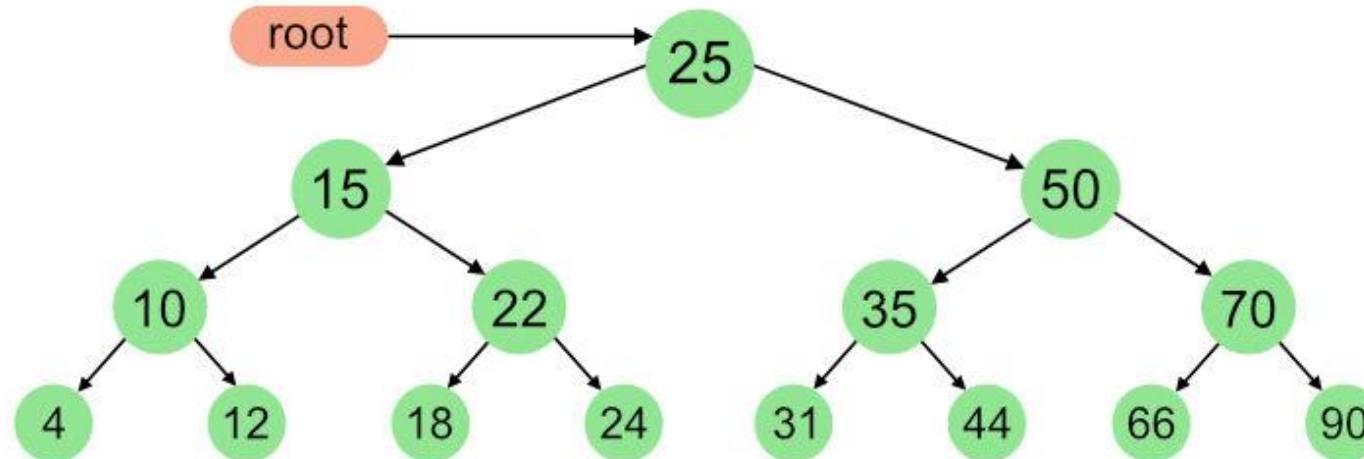
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

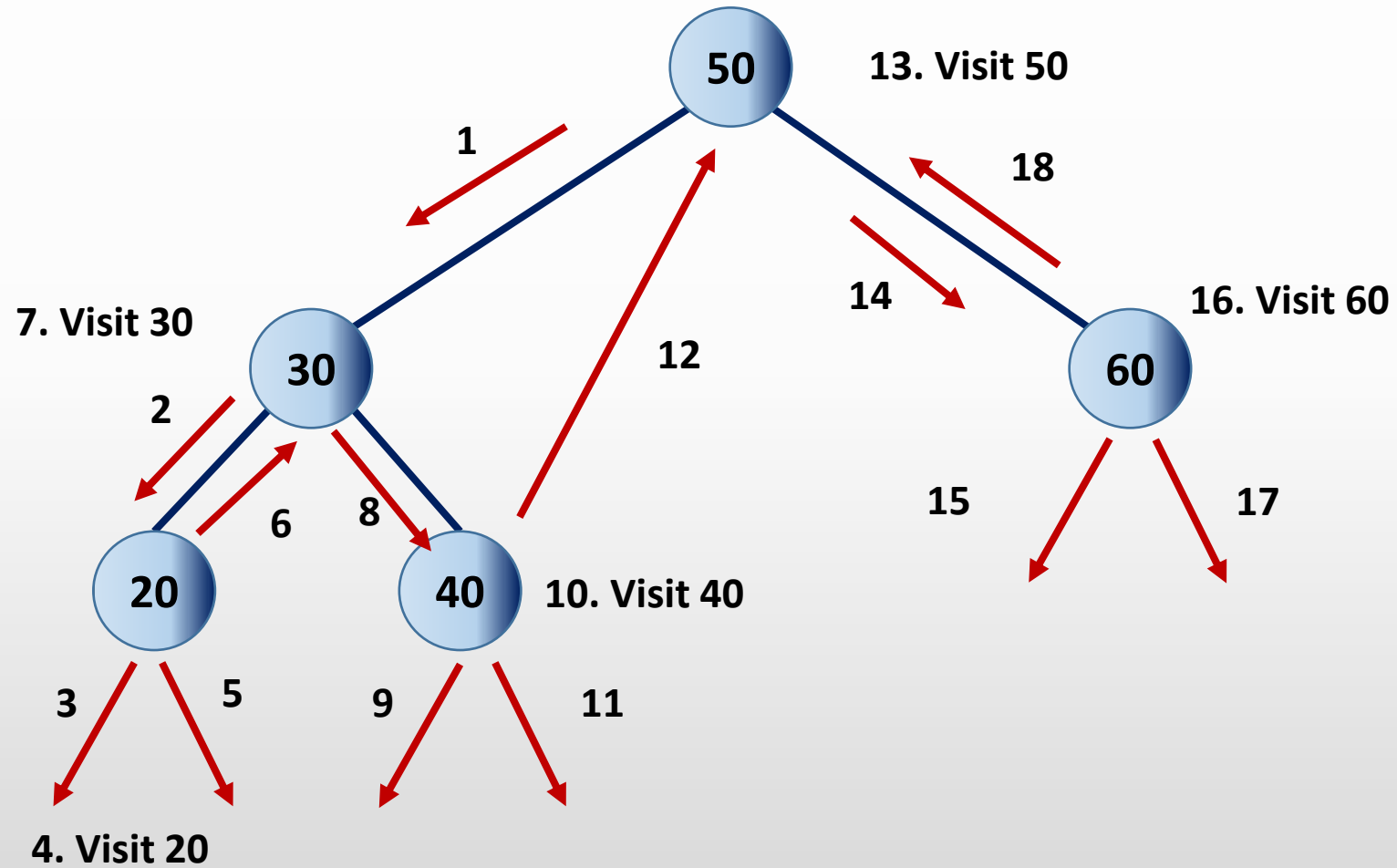
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Inorder traversing

- Call itself to traverse the node's left subtree
- Visit the node
- Call itself to traverse the node's right subtree

Inorder traversing cont...

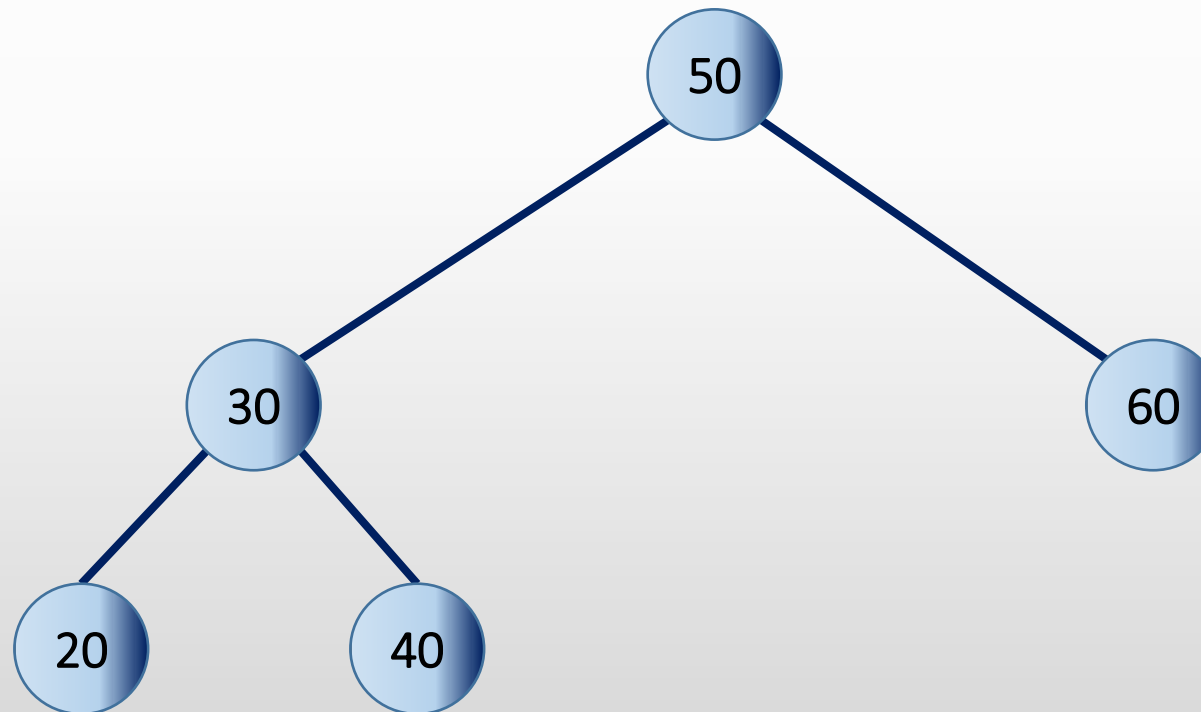


Preorder traversing

- Visit the node
- Call itself to traverse the node's left subtree
- Call itself to traverse the node's right subtree

Question 2

- Write the output, if the following tree is traverse in preorder.

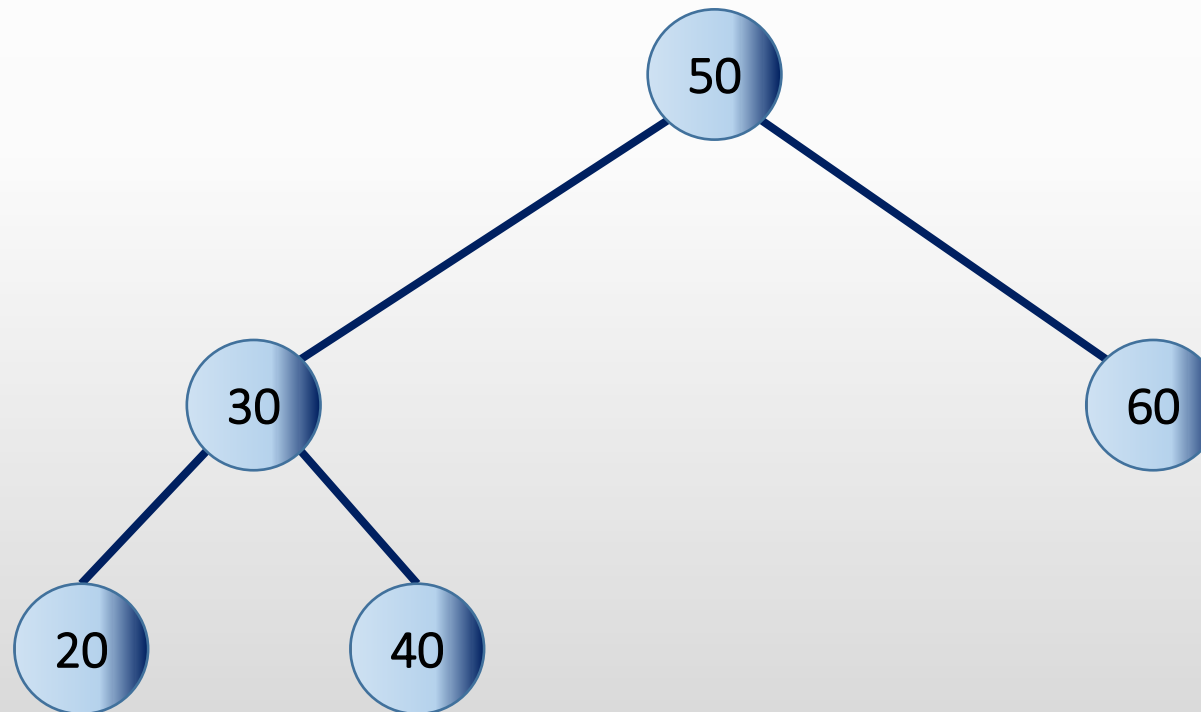


Postorder traversing

- Call itself to traverse the node's left subtree
- Call itself to traverse the node's right subtree
- Visit the node

Question 3

- Write the output, if the following tree is traverse in postorder.



Binary search tree - Implementation

Node Class

Node contains the information about an object.

Each node should have a key, data and reference to left and right child.

```
class Node
{
    public int iData; // data item (used as key value)
    public double dData; //other data
    public Node leftChild; // this node's left child
    public Node rightChild; //this node's right child

    public void displayNode( ){
        System.out.print("{");
        System.out.print(iData);
        System.out.print(", ");
        System.out.print(dData);
        System.out.print( " }");
    }
}
```


Binary search tree - Implementation

Tree Class

```
class Tree
{
    private Node root; // first node of tree

    public Tree(){
        root = null;
    }
    public void insert (int id, double dd){
    }
    public boolean delete (int id){
    }
    public Node find (int key){
    }
}
```

Binary search tree - Implementation

Tree Class – Find Method

```
class Tree
{
    private Node root;
    public Tree(){
        root = null;
    }
    public void insert (int id, double dd){
    }
    public void delete (int id){
    }
    public Node find(int key){
        Node current = root;
        while (current.iData != key)
        {
            if(key < current.iData)
                current = current.leftChild;
            else
                current = current.rightChild;
            if (current == null)
                return null;
        }
        return current;
    }
}
```

Binary search tree - Implementation

```

class Tree{
    private Node root;
    .....
    .....

    public void insert ( int id , double dd){
        Node newNode = new Node();
        newNode.iData = id;
        newNode.dData = dd;
        if (root == null) // no node in root
            root = newNode;
        else // root occupied
        {
            Node current = root; //start at root
            Node parent;
            while (true)
            {
                parent = current;
            }
        }
    }
}
    
```

```

        if (id < current.iData) // go left
        {
            current = current.leftChild;
            if (current == null) {
                parent.leftChild = newNode;
                return;
            }
        }
        else // go right
        {
            current = current.rightChild;
            if (current == null){
                parent.rightChild = newNode;
                return;
            }
        }
    }
}
}
}
    
```

Binary search tree - Implementation

Tree Class – Inorder Traversing Method

```
private void inOrder(Node localRoot)
{
    if (localRoot != null)
    {
        inOrder(localRoot.leftChild);
        localRoot.displayNode();
        inOrder(localRoot.rightChild);
    }
}
```

Binary search tree - Implementation

Tree Class – Preorder Traversing Method

```
private void preOrder(Node localRoot)
{
    if (localRoot != null)
    {
        localRoot.displayNode();
        preOrder(localRoot.leftChild);
        preorder(localRoot.rightChild);
    }
}
```

Binary search tree - Implementation

Tree Class – Postorder Traversing Method

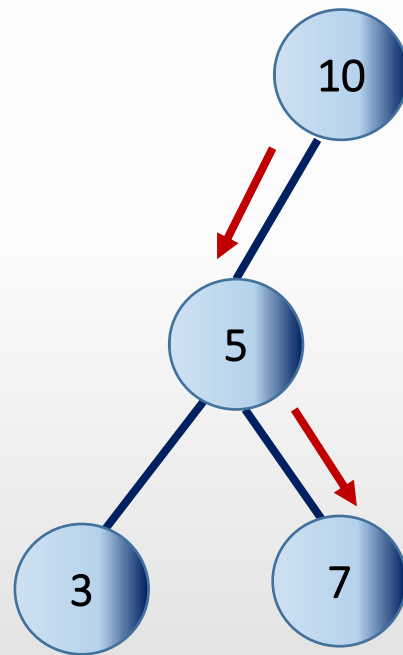
```
private void postOrder(Node localRoot)
{
    if (localRoot != null)
    {
        postOrder(localRoot.leftChild);
        postOrder(localRoot.rightChild);
        localRoot.displayNode();
    }
}
```

Operations - Delete

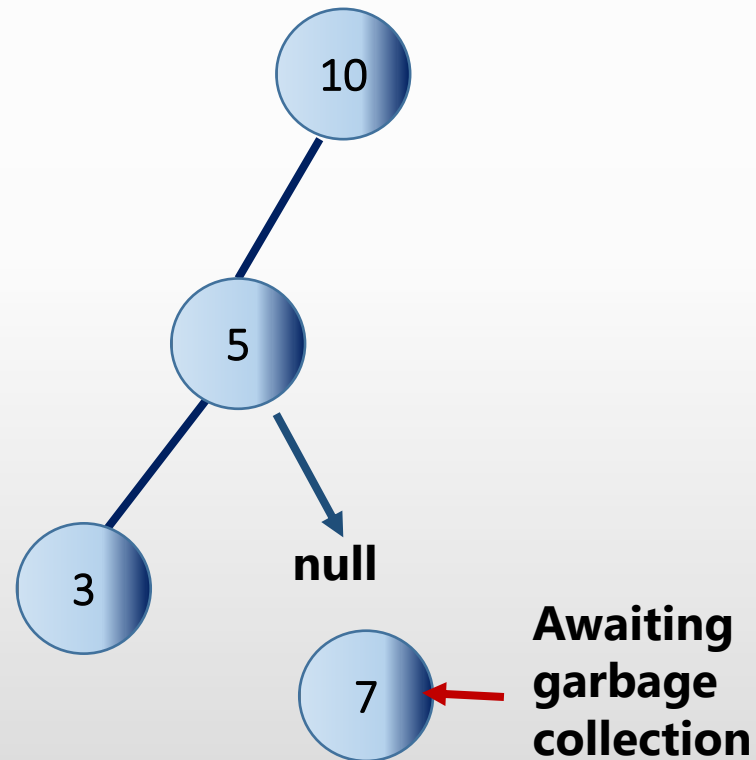
- First find the node to be deleted.
- If the node to be deleted is found there are three cases to be considered.
Whether
 - The node to be deleted is a leaf
 - The node to be deleted has one child
 - The node to be deleted has two children.

Case 1 : The node to be deleted has no children

Delete 7



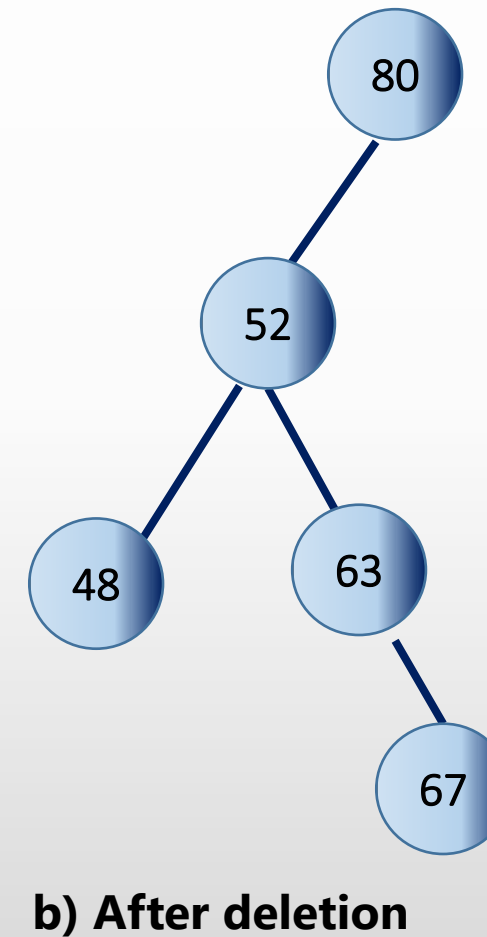
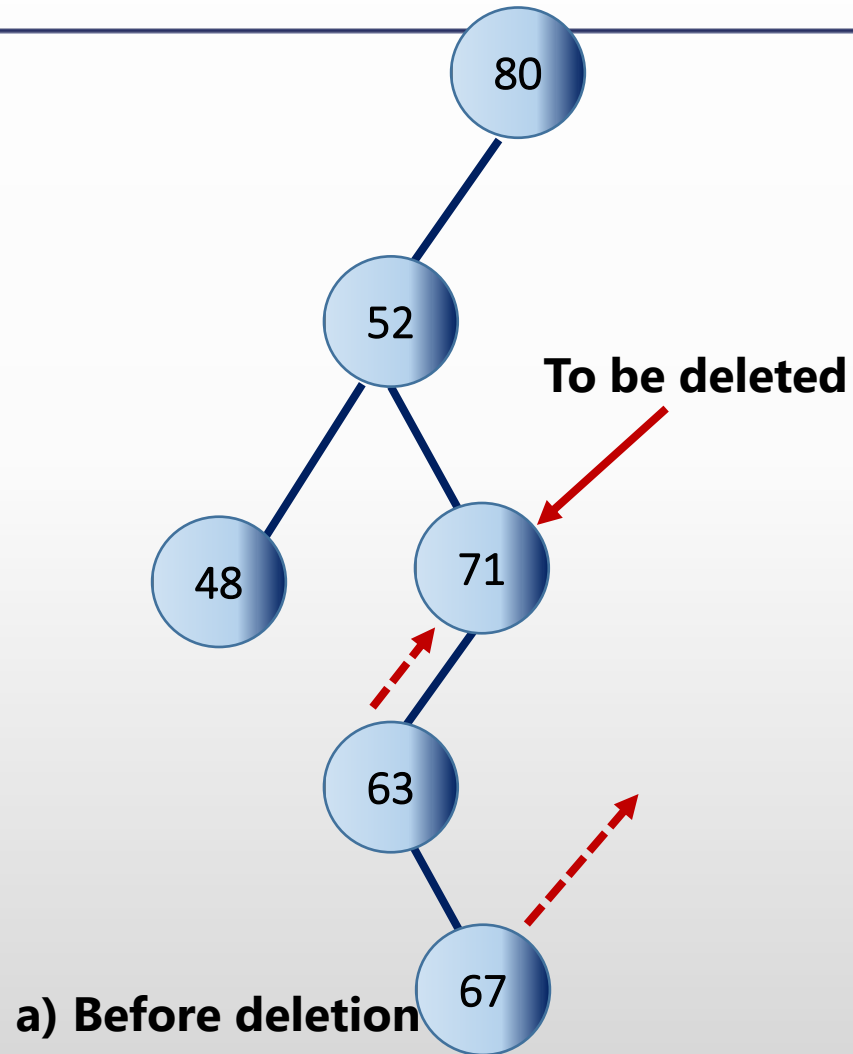
a) Before deletion



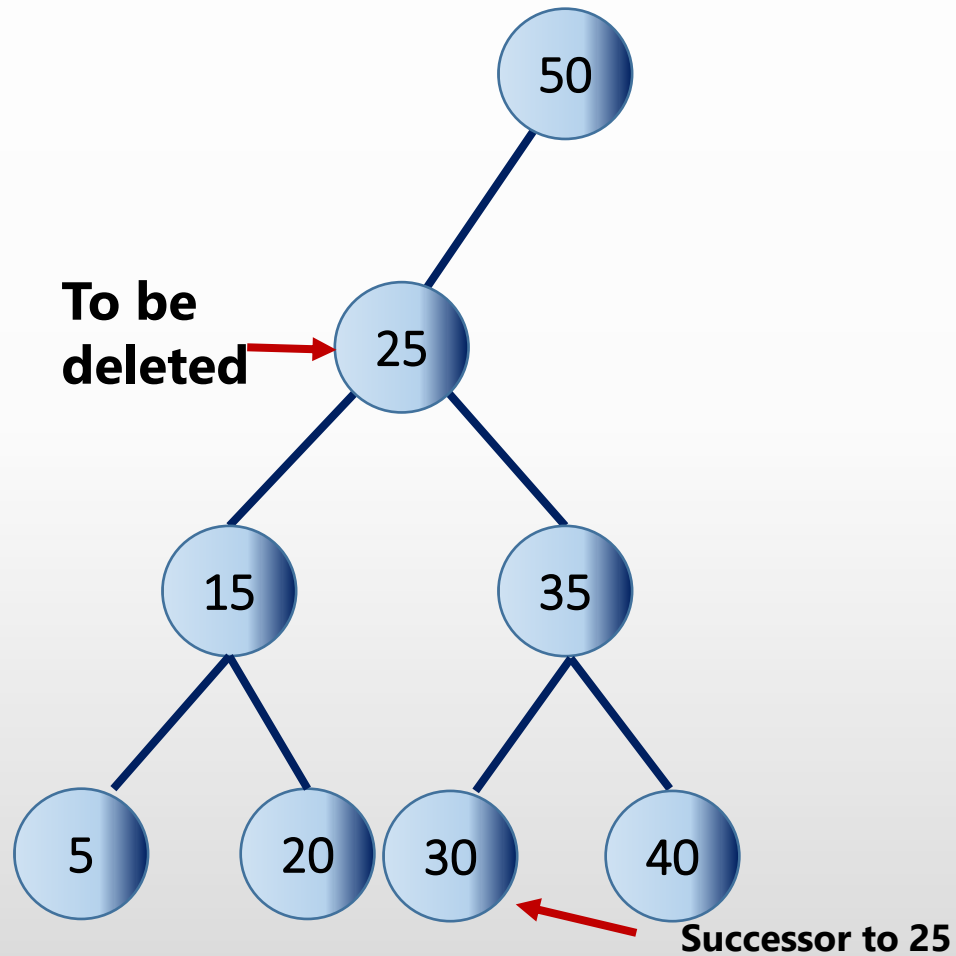
b) After deletion

Awaiting
garbage
collection

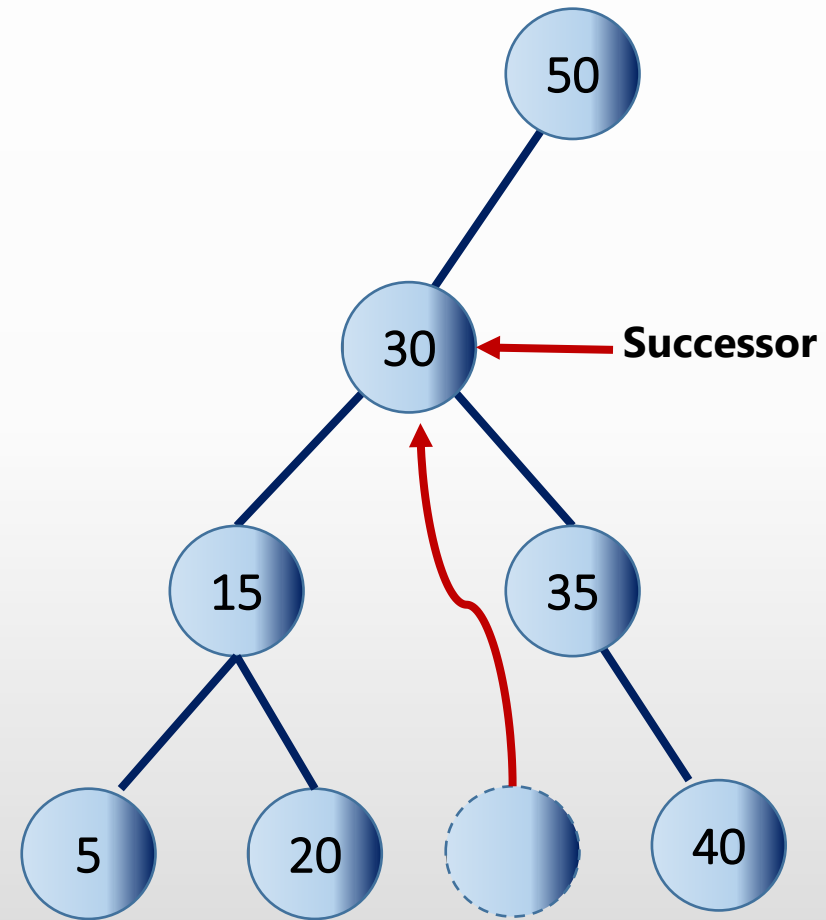
Case 2 : The node to be deleted has one child



Case 3 : The node to be deleted has two children



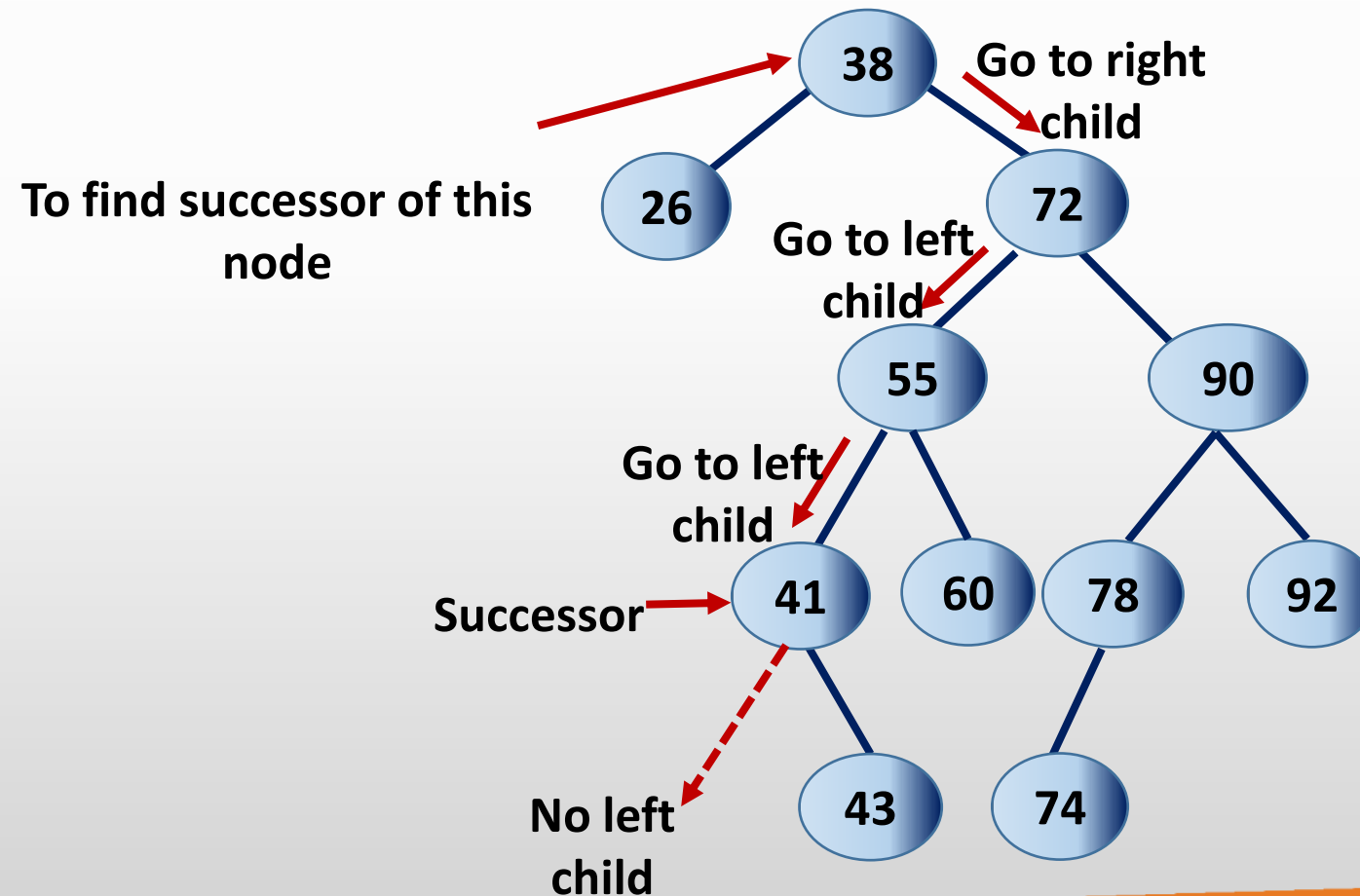
a) Before deletion



b) After deletion

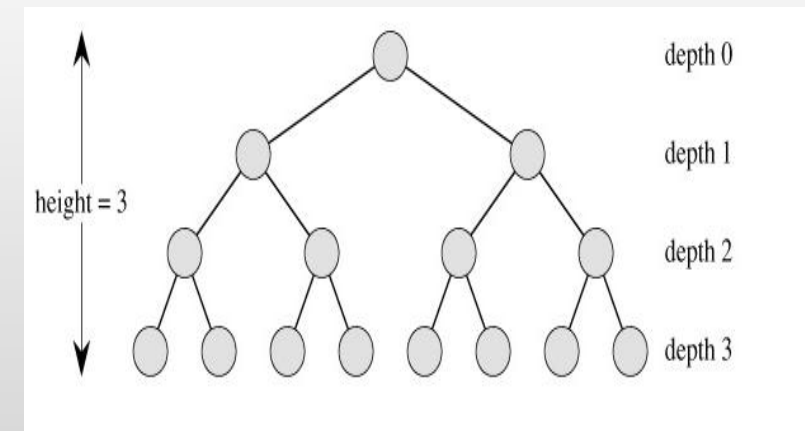
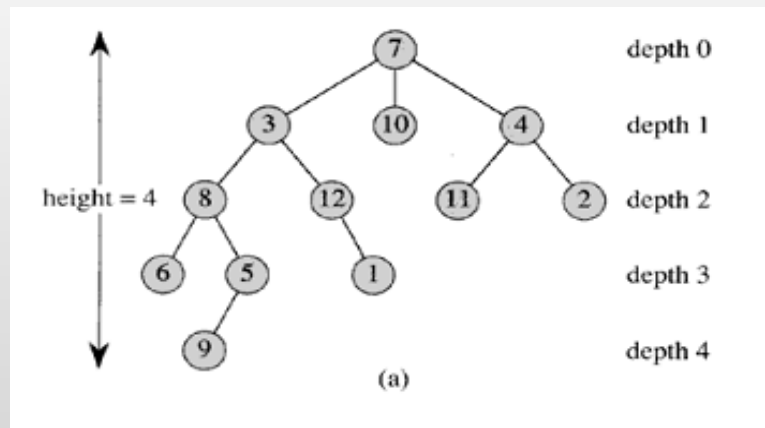
How to find a successor of a node?

- In a Binary Search Tree, successor of a node is a node with next-highest key.



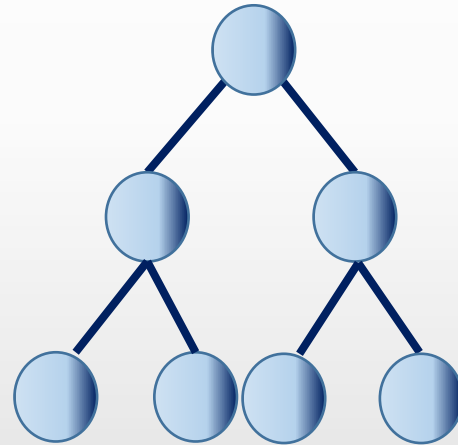
Tree Terminology

- **Degree of a node:** The number of children it has
- **Depth:** The depth of x in a tree is the length of the path from the root to a node x .
- **Height:** The largest depth of any node in a tree.



Full Binary Tree

- A Full binary tree of height ***h*** that contains exactly $2^{h+1}-1$ nodes



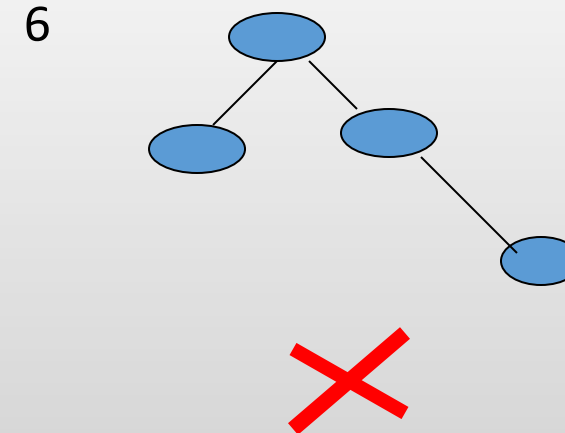
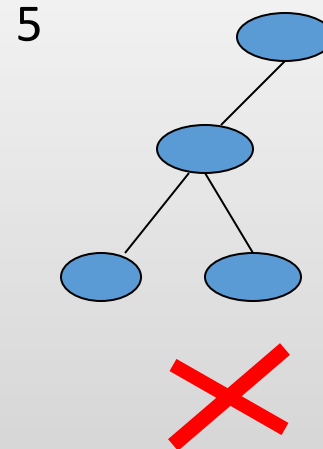
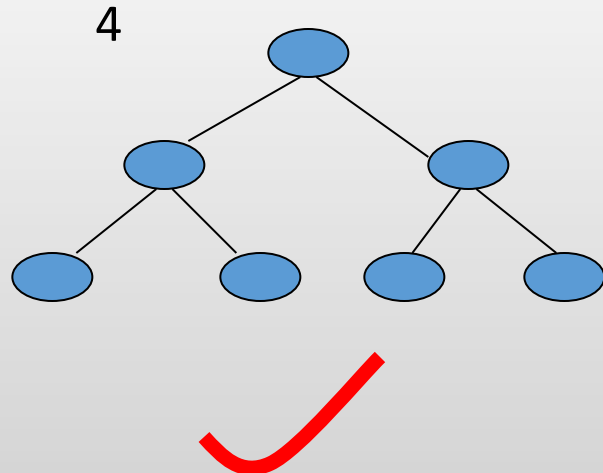
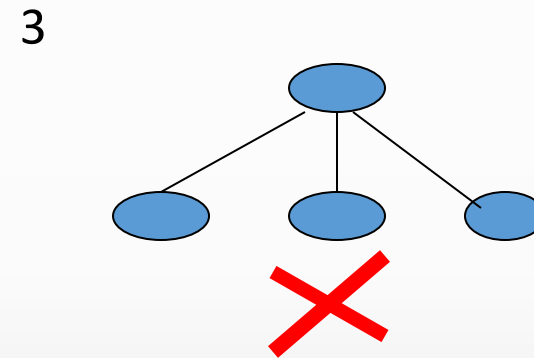
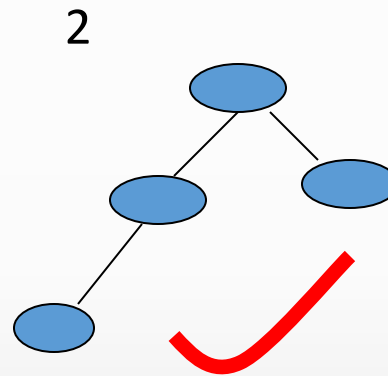
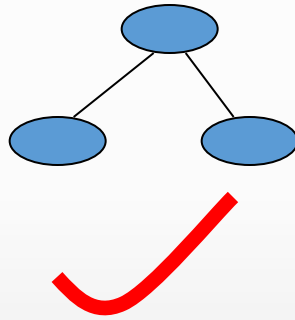
Height, ***h*** = 2, nodes = $2^{2+1}-1=7$

Complete Binary Tree

- It is a Binary tree where each node is either a leaf or has degree ≤ 2 .
- Completely filled, except possibly for the bottom level
- Each level is filled from **left to right**
- All nodes at the lowest level are as far to the left as possible
- Full binary tree is also a complete binary tree

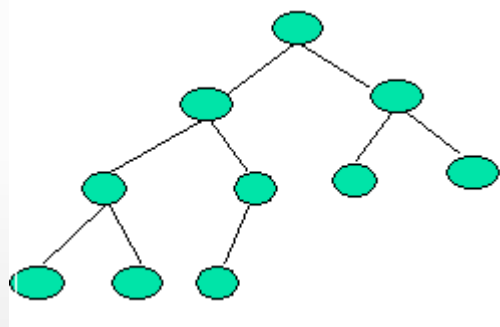
Question

- Find Complete Binary Trees



Height of a complete binary tree

- Height of a complete binary tree that contains n elements is $\lfloor \log_2(n) \rfloor$
- Example



Above is a Complete Binary Tree with height = 3

No of nodes: $n = 10$

$$\text{Height} = \lfloor \log_2(n) \rfloor = \lfloor \log_2(10) \rfloor = 3$$