SLIIT
Discover Your Future

IT2070 – Data Structures and Algorithms

Introduction

Subject Group

Malabe Campus

- Ms.Namalie Walgampaya
- Ms. Dinuka Wijendra

Metro Campus

- Dr. Jeewanee Bamunusinghe

Kandy Center

- Ms. Chathurika Pinnaduwage

Matara Center

- Mr.Ravi Supunya

Teaching Methods

- Lectures – 2 hours/week
- Tutorials – 1 hour/week
- 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

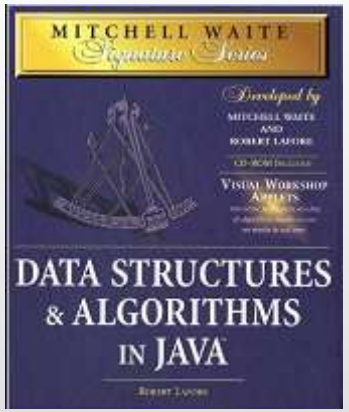
From year 2018 the committing above offenses come with serious consequences !

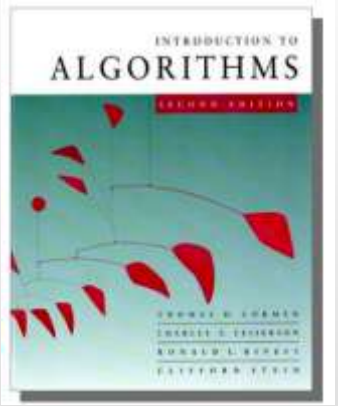
See General support section of Courseweb for full information.

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.





SLIIT
Discover Your Future

IT2070 – Data Structures and Algorithms

Lecture 01

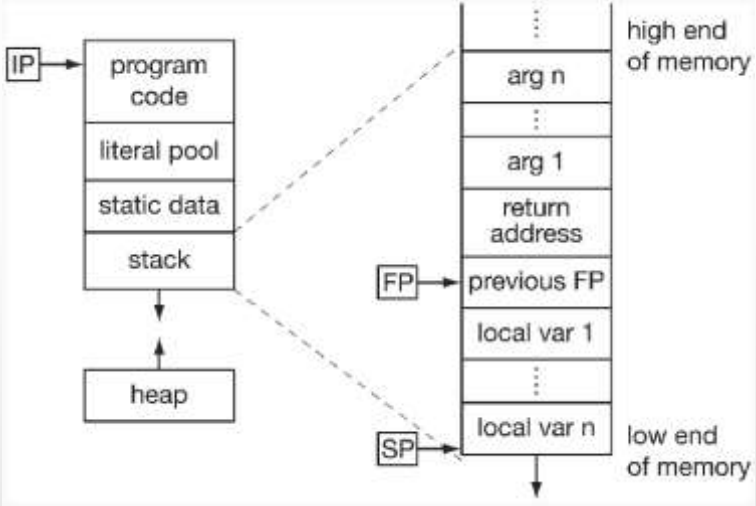
Introduction to Stack

Stack

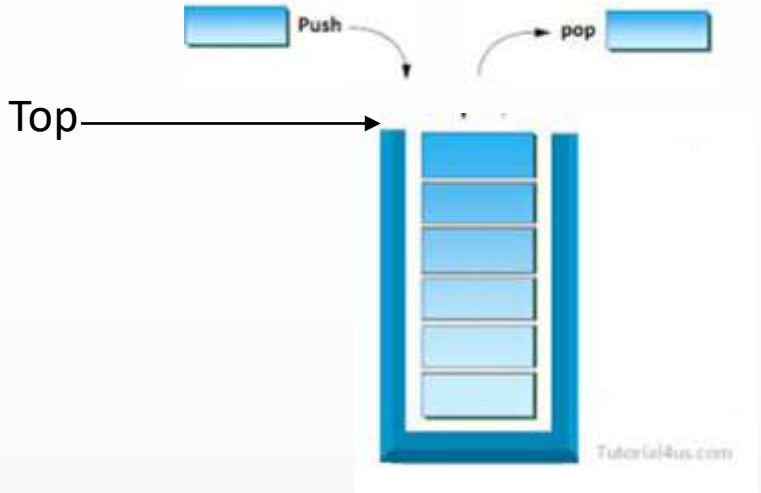
- Allows access to only one data item; the last item inserted
- If you remove this item, then you can access the next-to-last item inserted

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

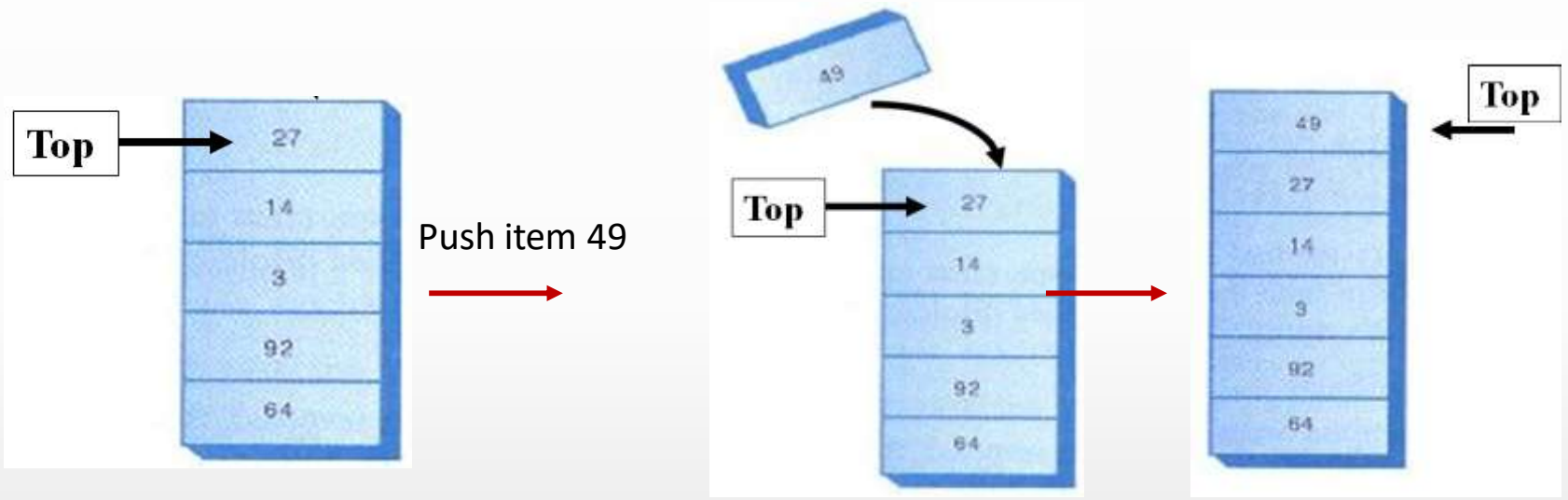


- In a stack all insertions and deletions are made at one end (Top). Insertions and deletions are restricted from the Middle and at the End of a Stack

- 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

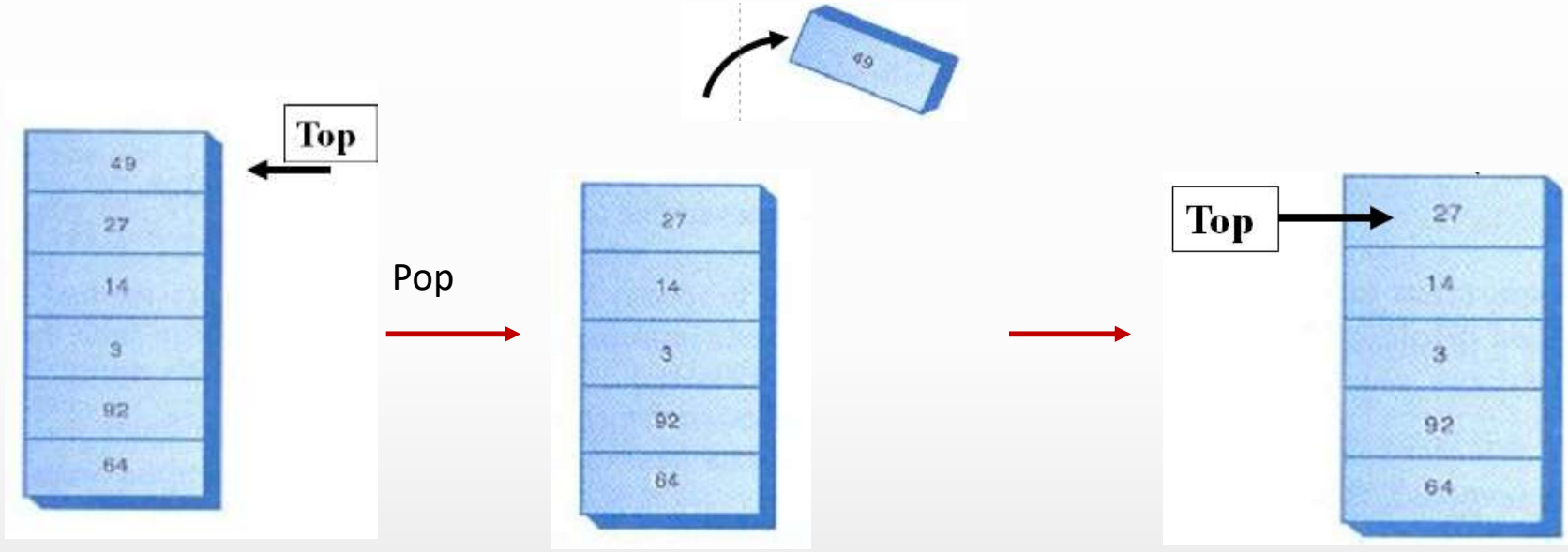


Before inserting

Item 49 is inserted on top

After inserting

Stack - Pop

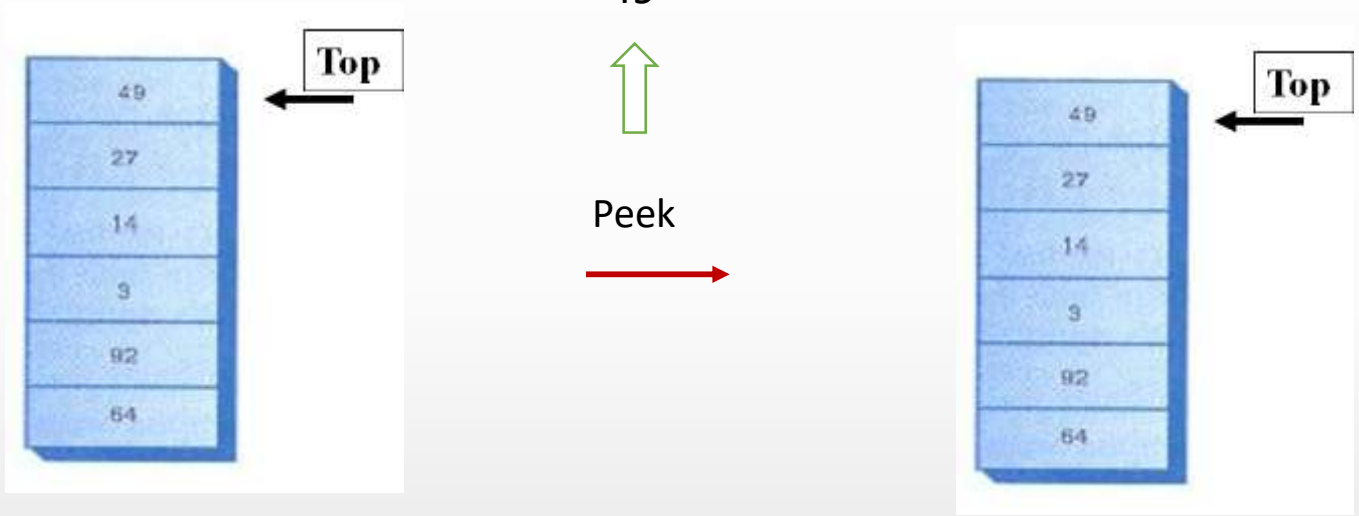


Before removing

Top item is removed

After removing

Stack - Peek

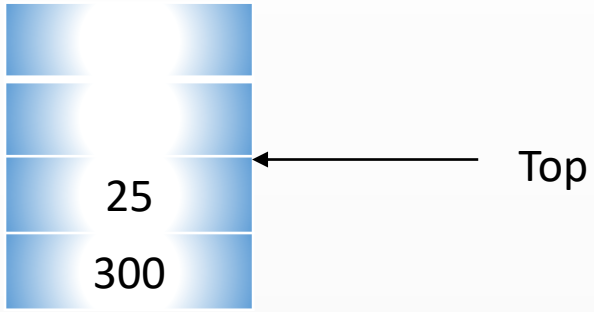


Stack remains the same

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

    publc 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

    publc 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

      publc 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

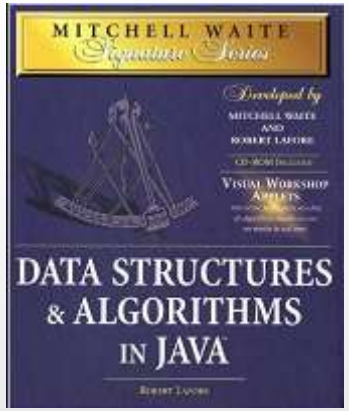
Delete all the items from the stack and display the deleted items.

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.

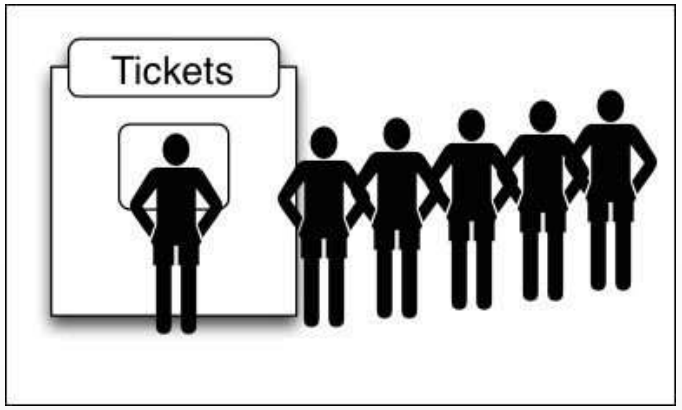


IT2070 – Data Structures and Algorithms

Lecture 02

Introduction to Queue

Queues



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

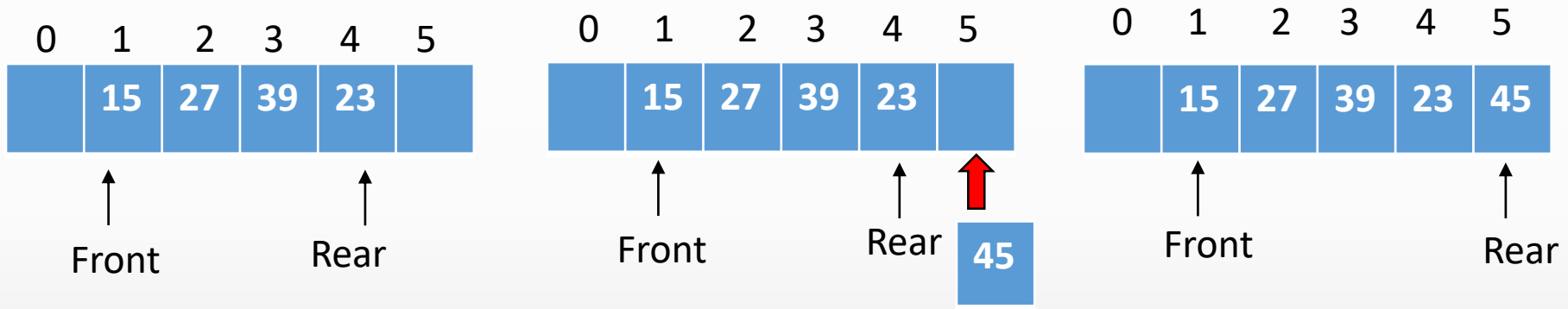
Queues

- 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

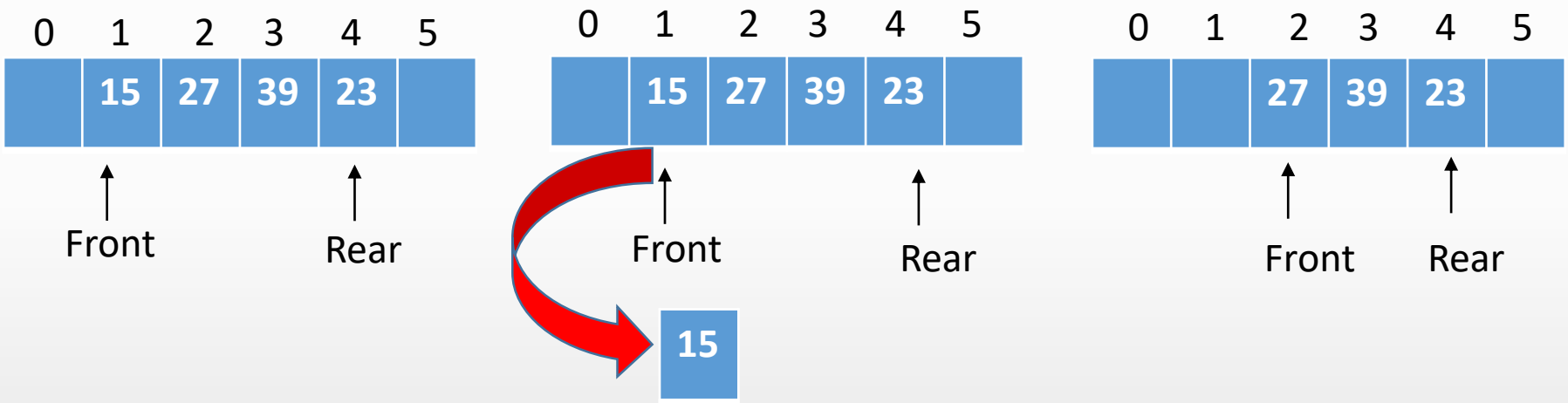


Before inserting

Item 45 is inserted to the rear

After inserting

Queue - Remove

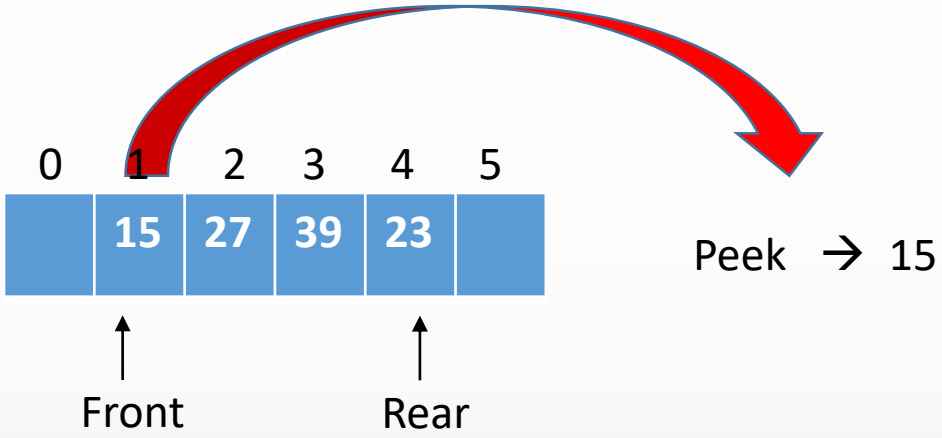


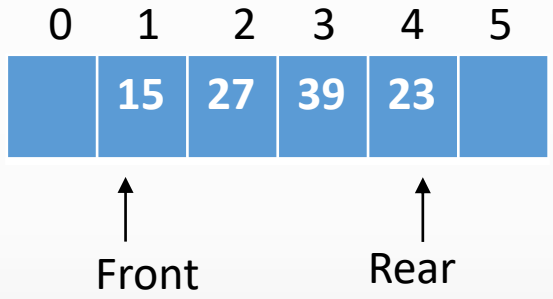
Before removing

Front item is removed

After removing

Queue - PeekFront



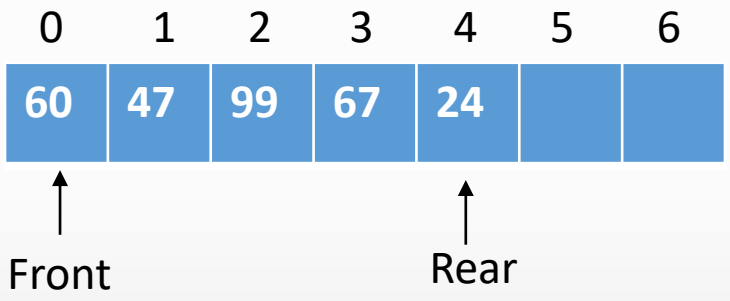


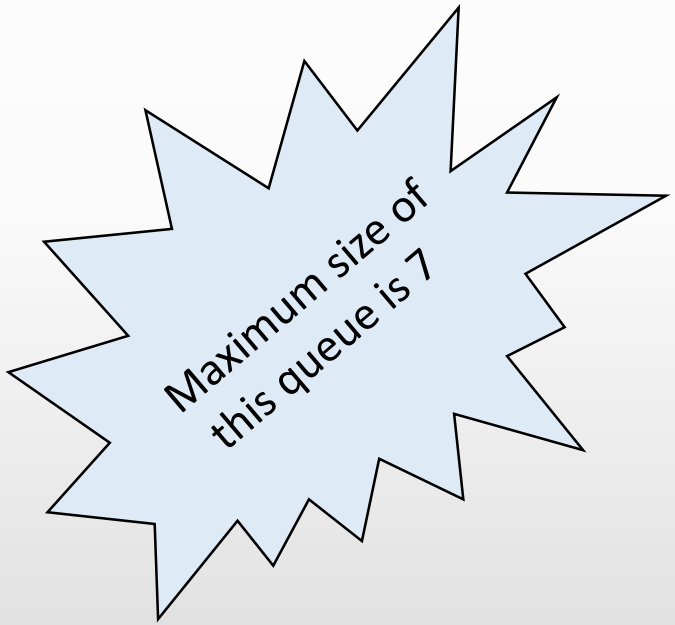
Queue remains the same

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.

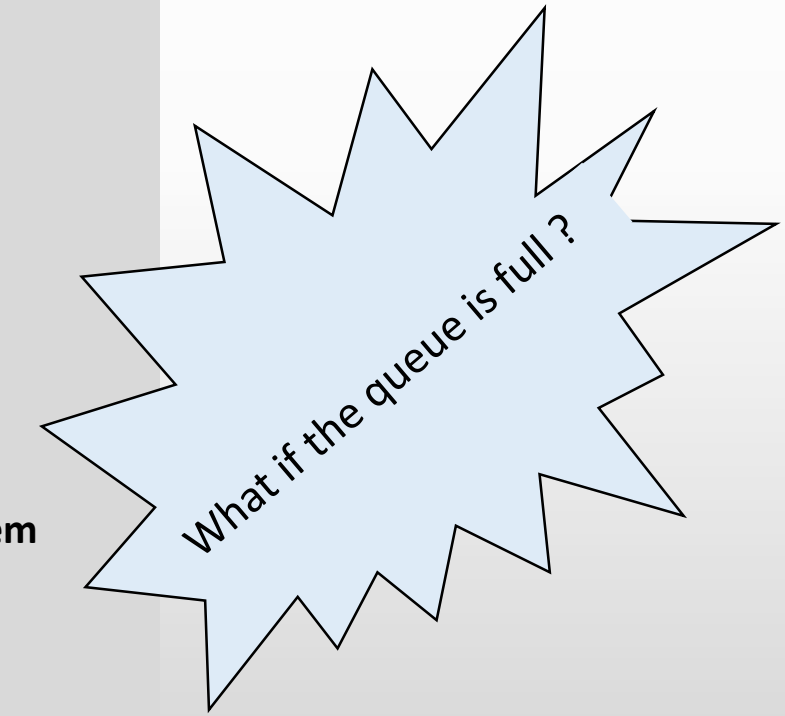
```
class QueueX {
    private int maxSize;    // size of queue array
    private int [] queArray;
    private int front;          //front of the queue
    private int rear;          //rear of the queue
    private int nItems;  //no of items of the queue

    publc 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

      publc 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

        publc 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++;
      }
}
```



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

        publc 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

      publc 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

        publc 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

        publc 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];
     }
}
```

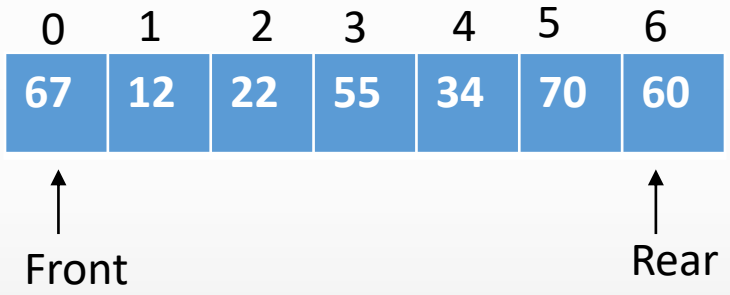
Question 02

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

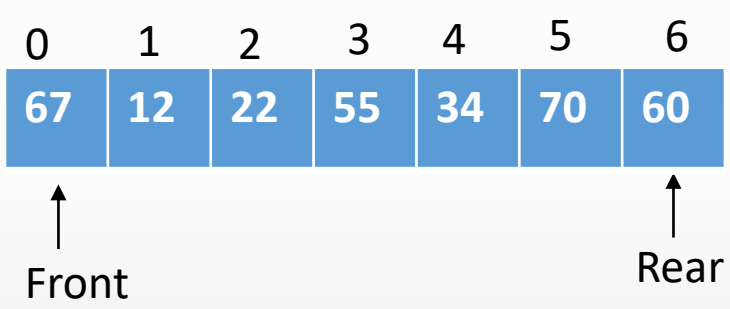
Draw the Queue frame after performing the below operations to the queue given below.

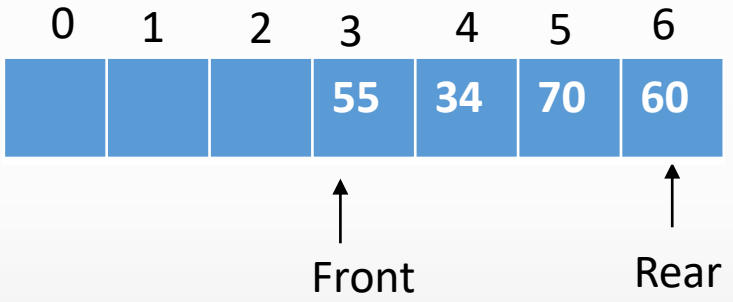


i) remove
ii) remove
iii) remove
iv) Insert item 88

Question 03 Contd..

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





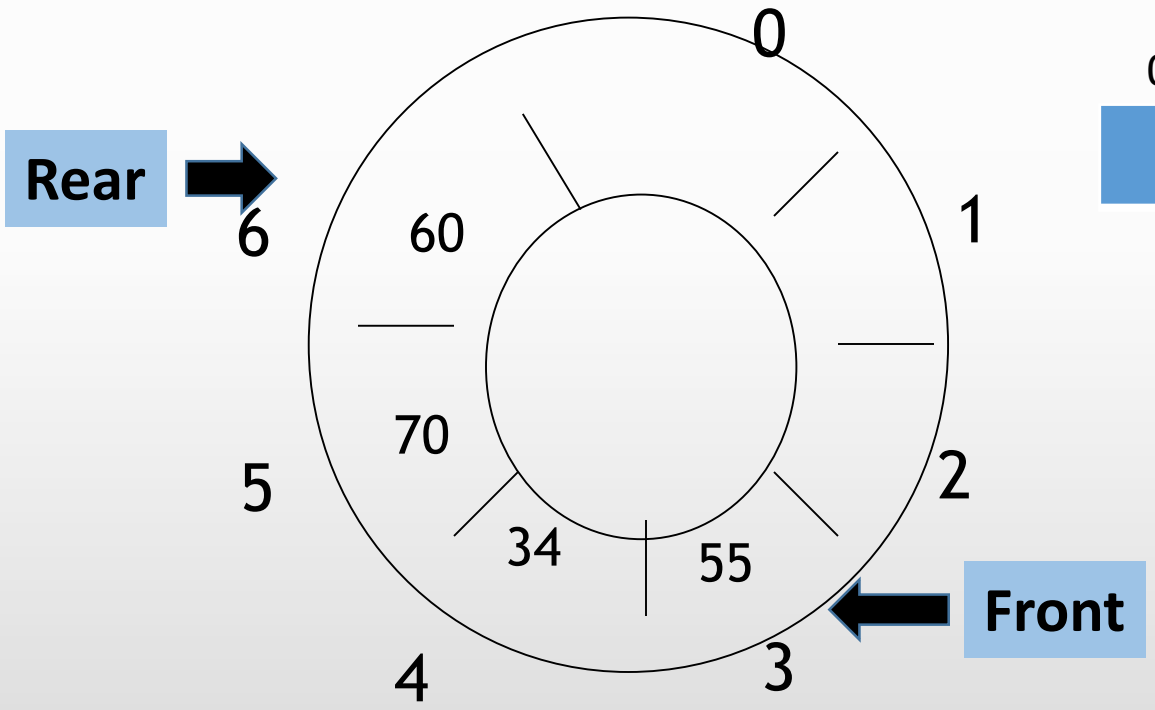
i) remove
ii) remove
iii) remove
iv) Insert item 88

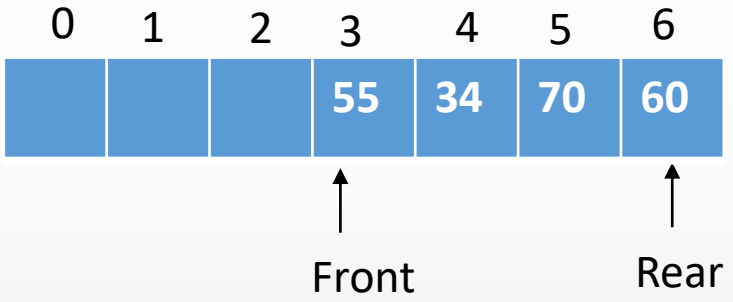
Although the queue is not full we cannot insert more elements.

Any Suggestions?

How to overcome this situation??

We can use a Circular Queue





Circular Queue

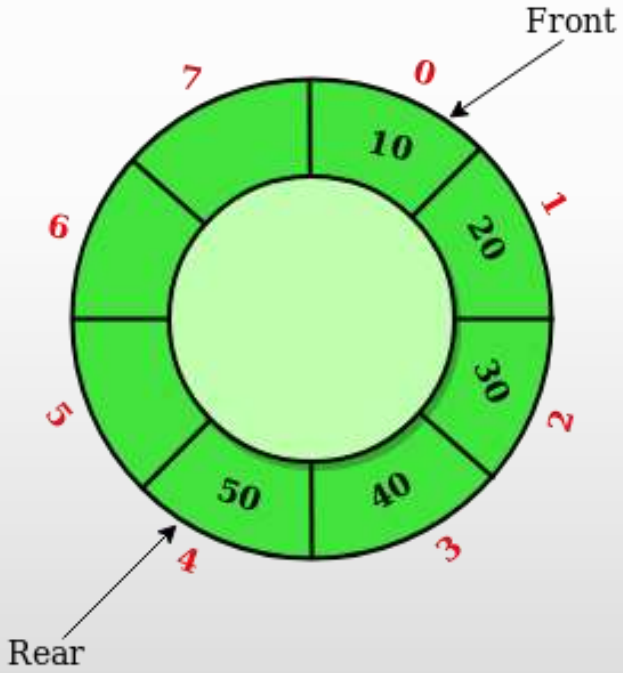
- Circular queues are queues that wrap around themselves.
- These are also called ring buffers.
- The problem in using the linear queue can be overcome by using circular queue.
- When we want to insert a new element we can insert it at the beginning of the queue.

  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

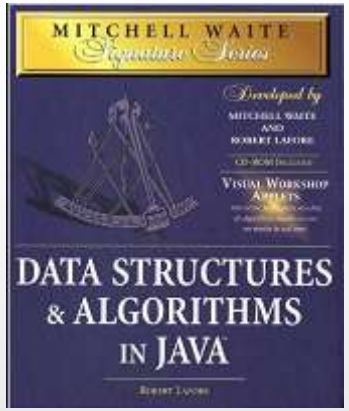
Delete all the items from the queue and display the deleted items.

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.



Linked Lists

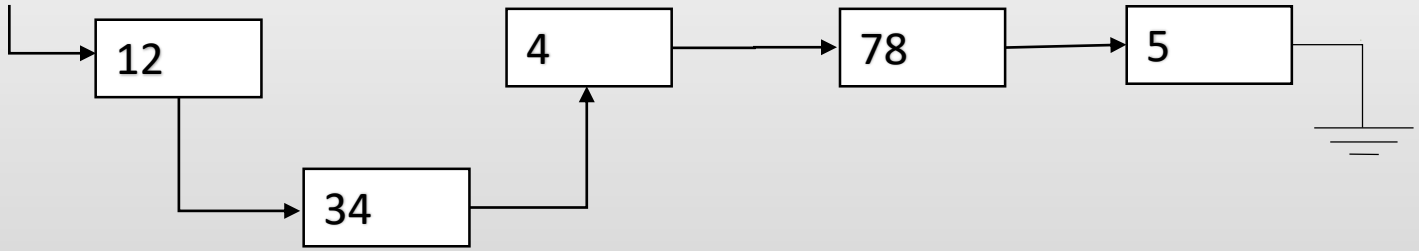
Ways in which linked lists differ from arrays

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 →



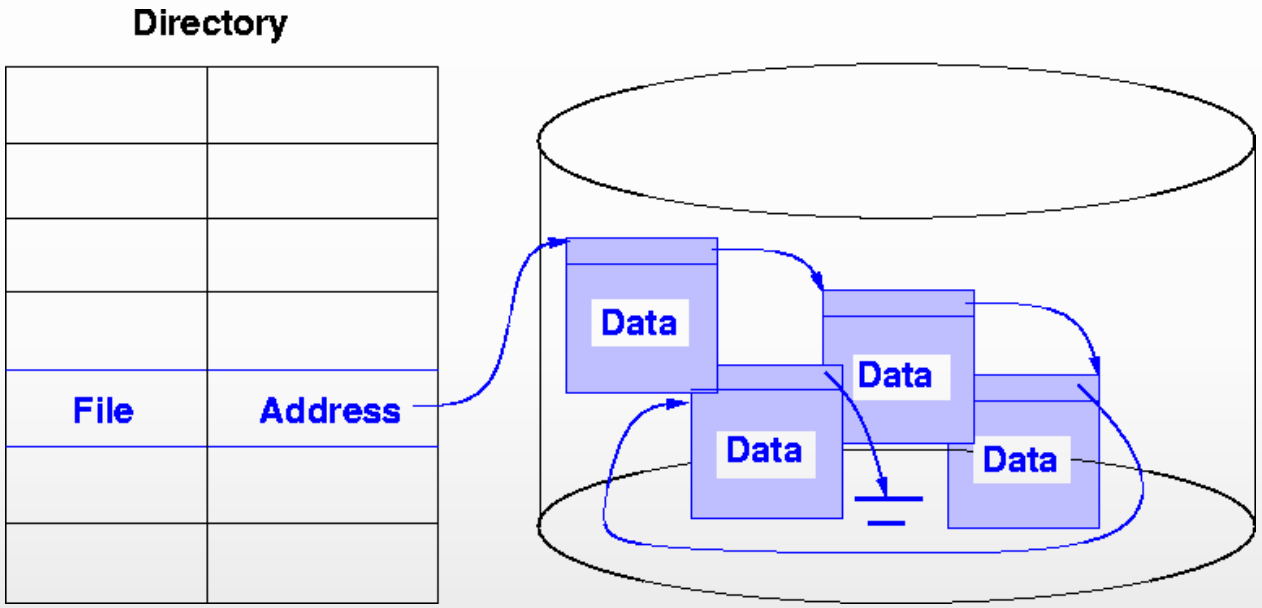
Applications of linked list in real world-

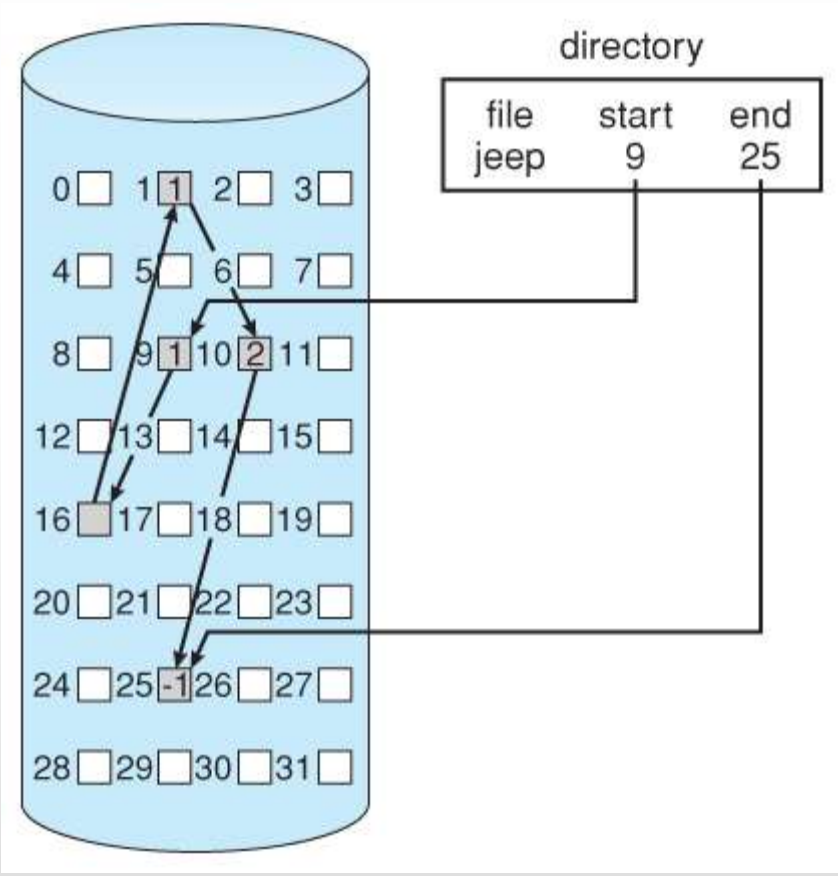
- *Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.
- *Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- *Music Player* – 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 is uses linked list to store adjacent vertices.
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names

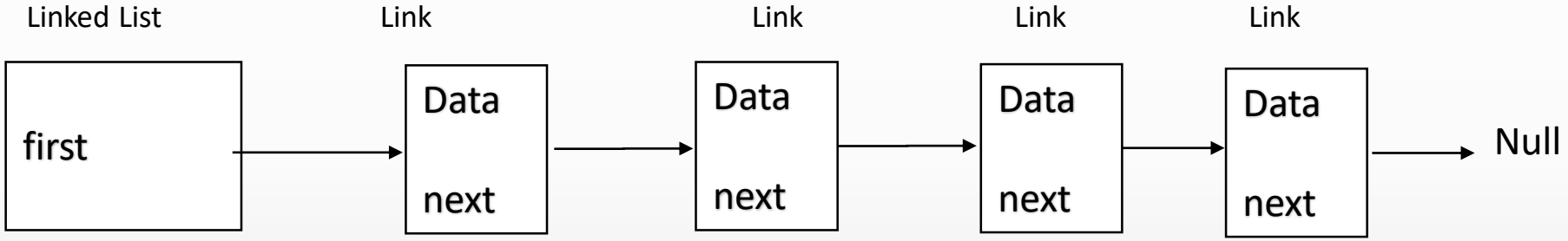
Linked Allocation in File System





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.

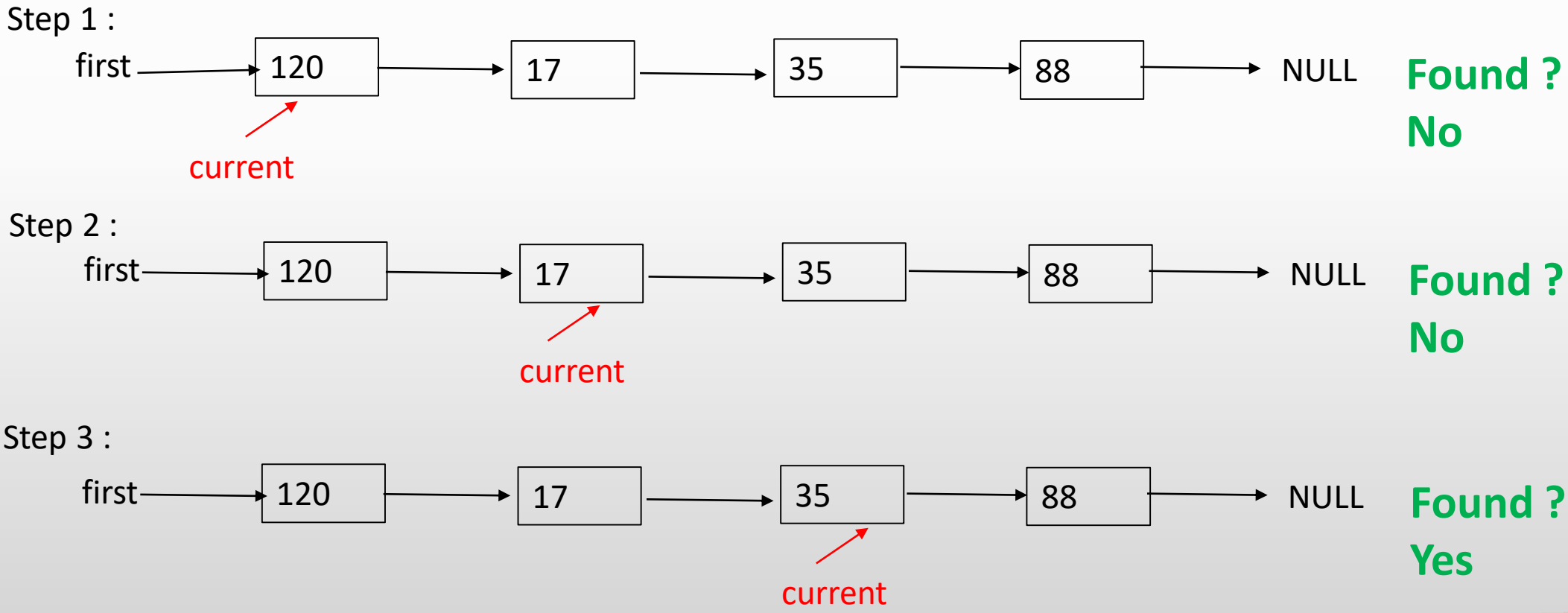
  - Delete

    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.

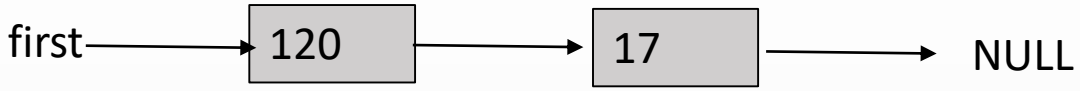
Ex: **Find Item 35**



Operations – Insert

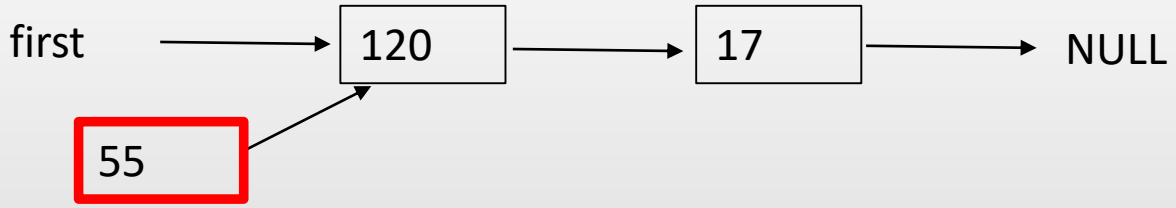
Inserting an item at the beginning of the list

Before inserting

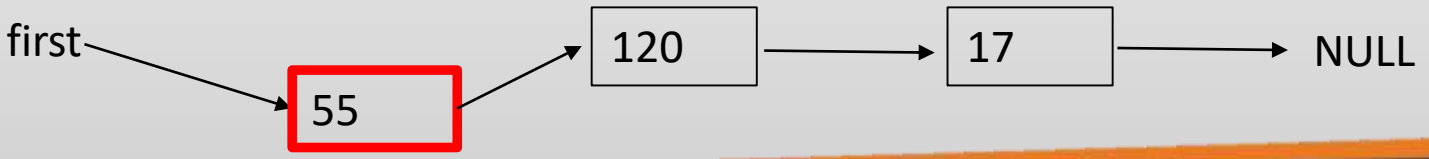


Step 1 : create a new link

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

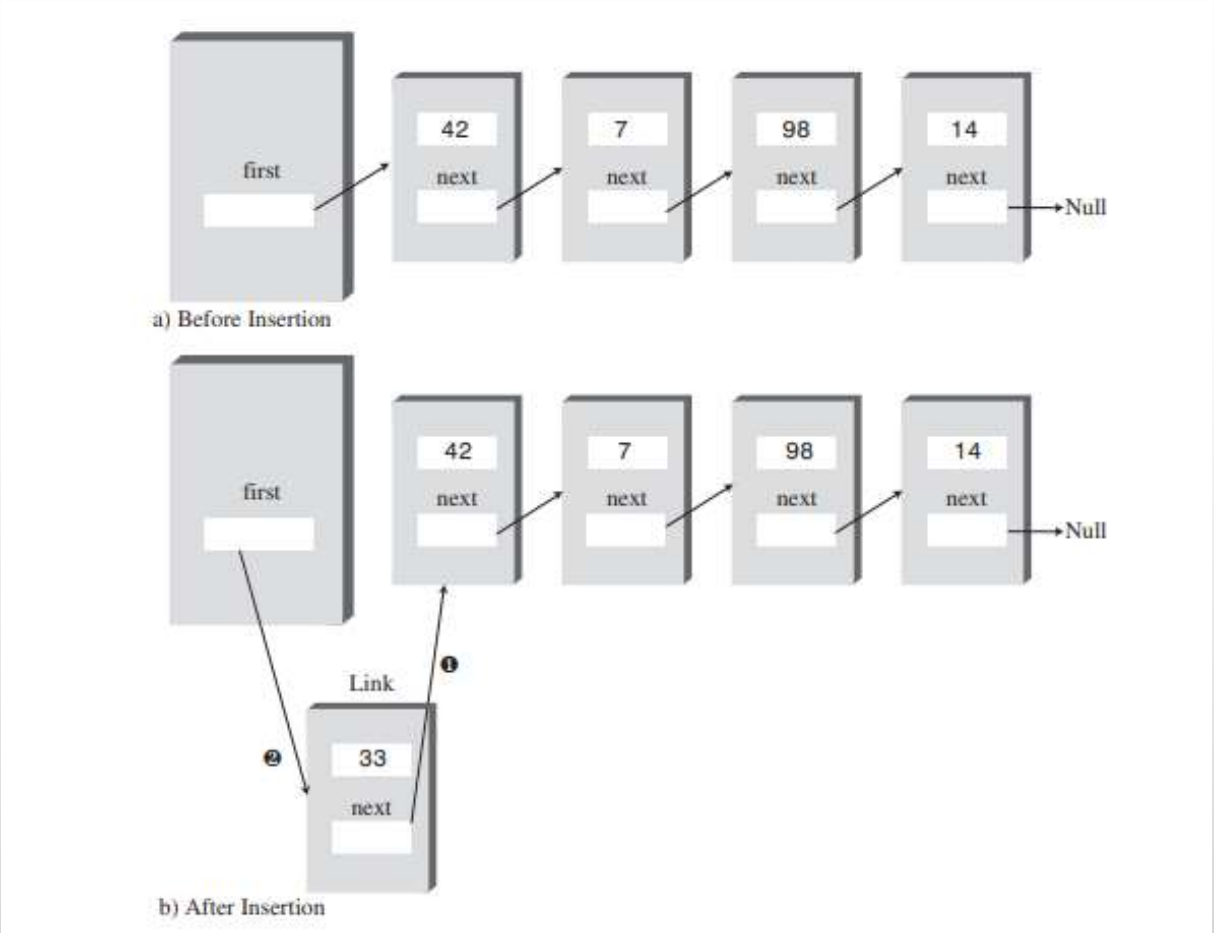


Step 3 : ‘first’ points to the newly created link



SLIIT
Discover Your Future

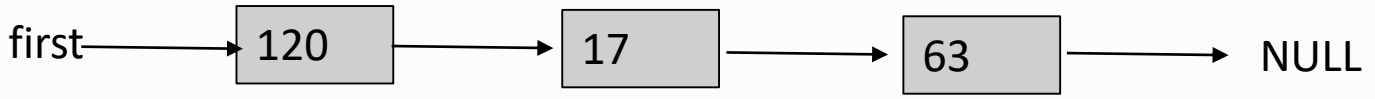
InsertFirst()



Operations - Insert

Inserting an item in the middle of the list

Before inserting



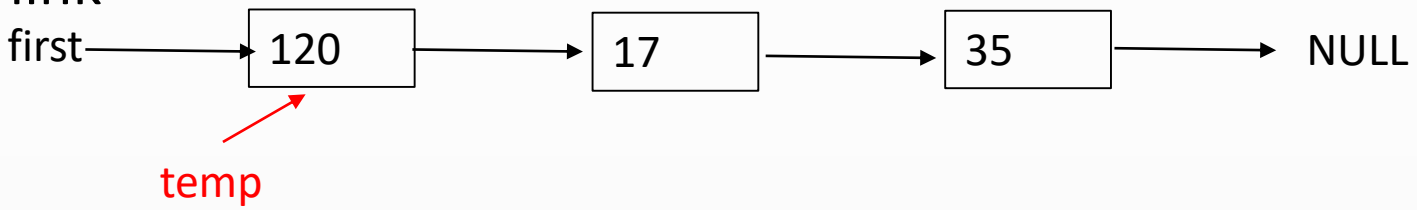
Question:

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

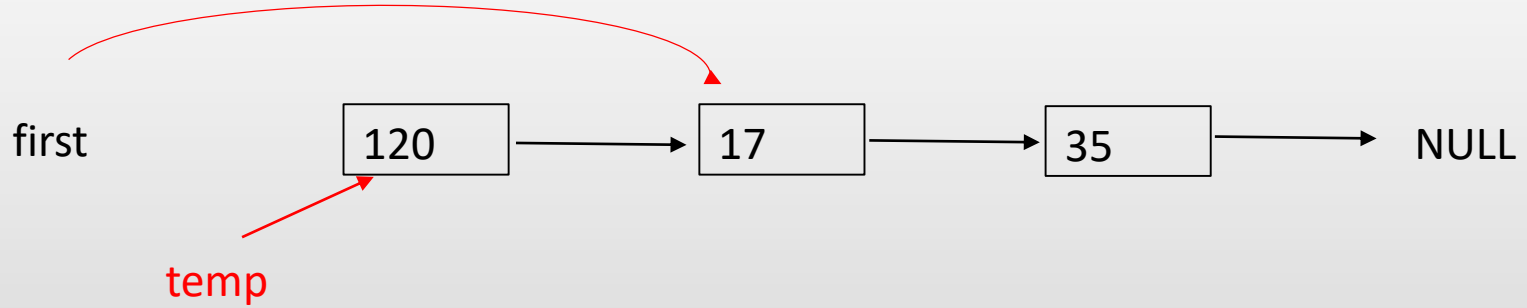
Operations - Delete

Deleting an item from the beginning of the list

Step 1 : Save reference to first link

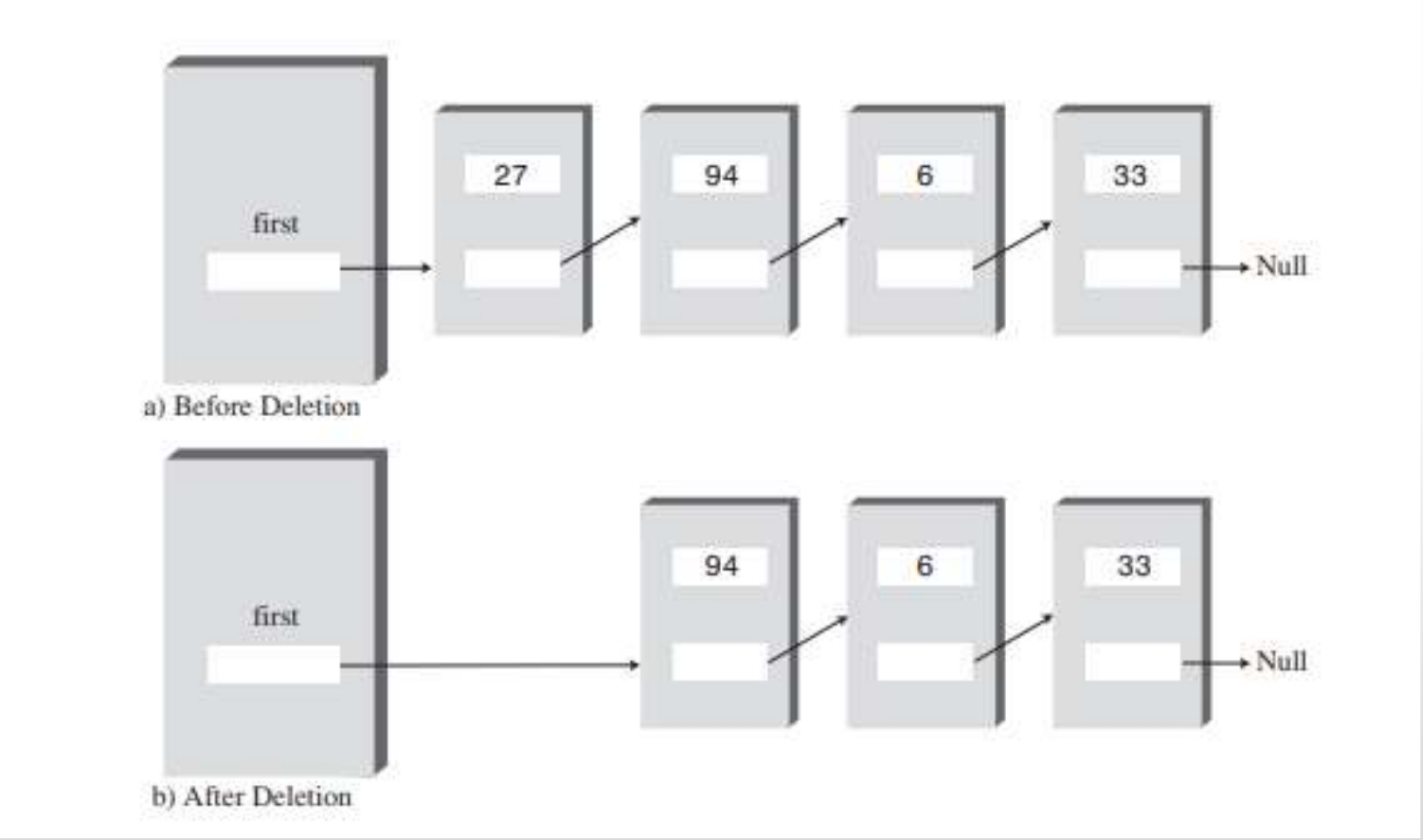


Step 2 : Disconnect the first link by rerouting first to point to the second link



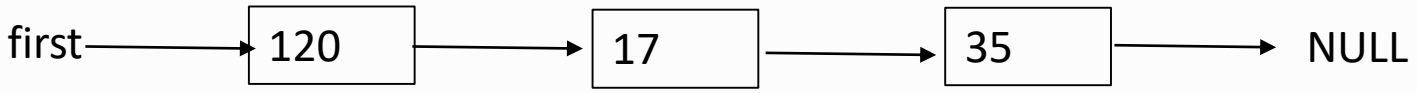
Step 3 : Return the deleted link (temp)

deleteFirst()



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

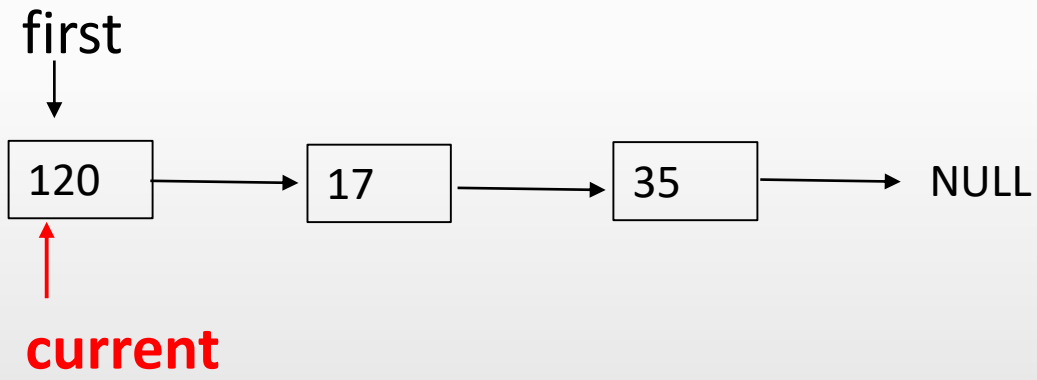
Link List Class

- The LinkList 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 LinkList {
    private Link first;

    public LinkList() {     //constructor

            first = null;
    }
    public boolean isEmpty() {  // true if list is empty

            return (first == null);
    }
    // ……….        other methods
}
```

Linked List - Implementation

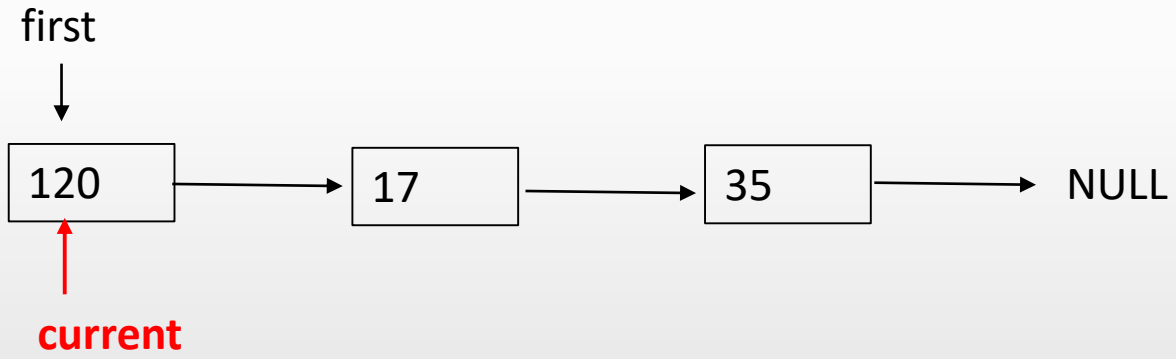
Link List Class – Contd.




```
class LinkList {
    private Link first;
    public  LinkList() { //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 LinkList {
    private Link first;
    public LinkList()  {  //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 LinkList {
    private Link first;
    public LinkList() {  //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.

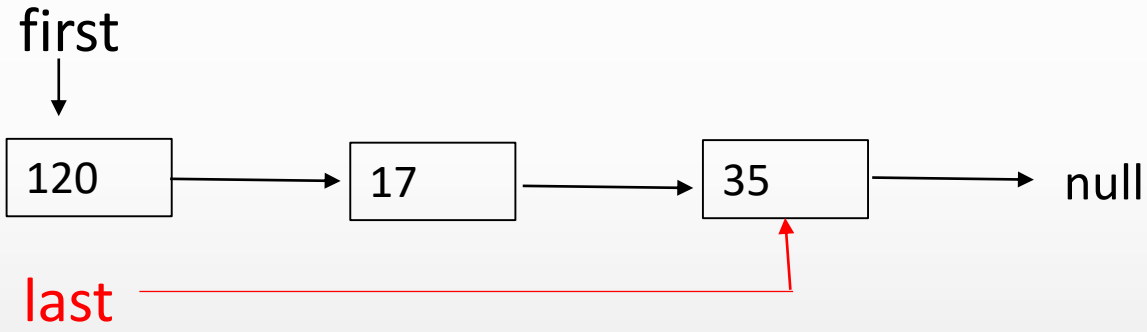
(Use the LinkList class created)

Answer1

```
class  myList {
      public static void main(String[] args)       {
            LinkList theList  = new LinkList();    // 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();
            }
      }
}
```

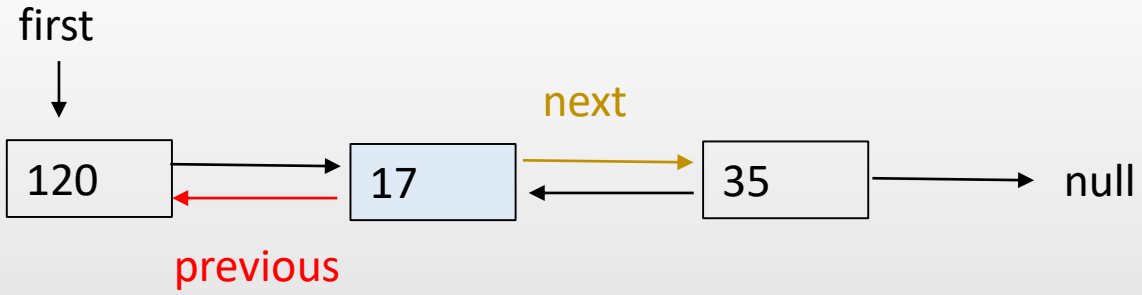
Double-Ended List

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

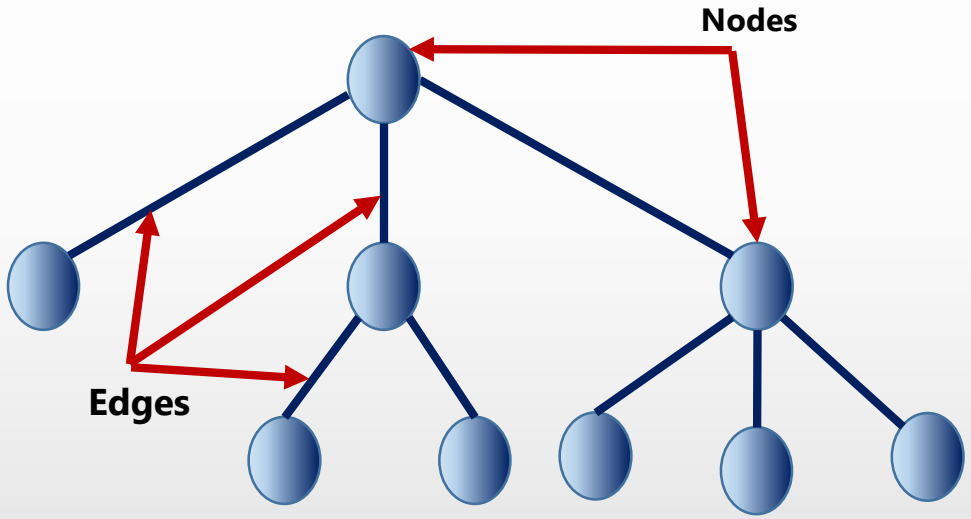
What is a tree?

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.

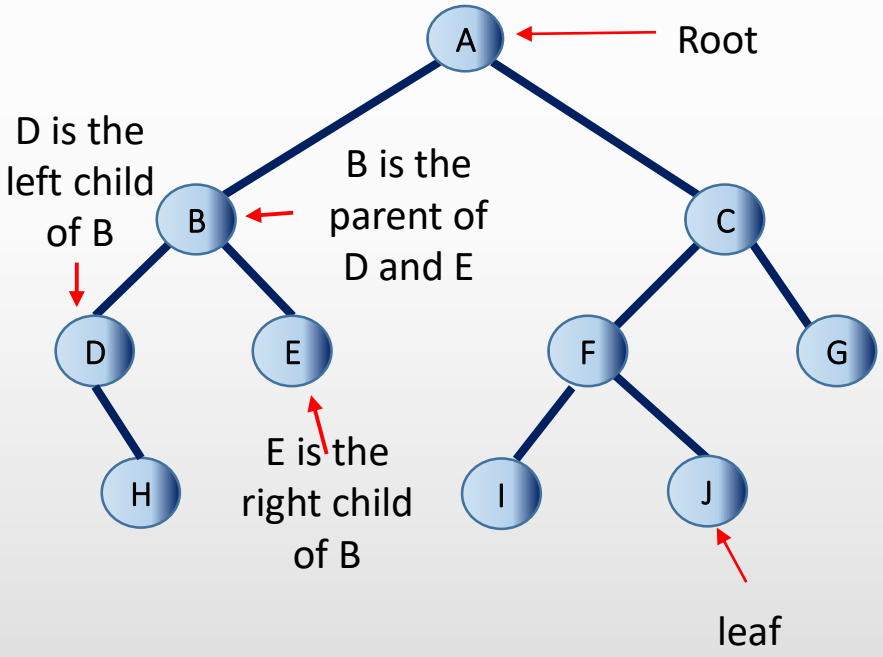
In a tree, the nodes represent the data items and the edges represent the way the nodes are related.

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



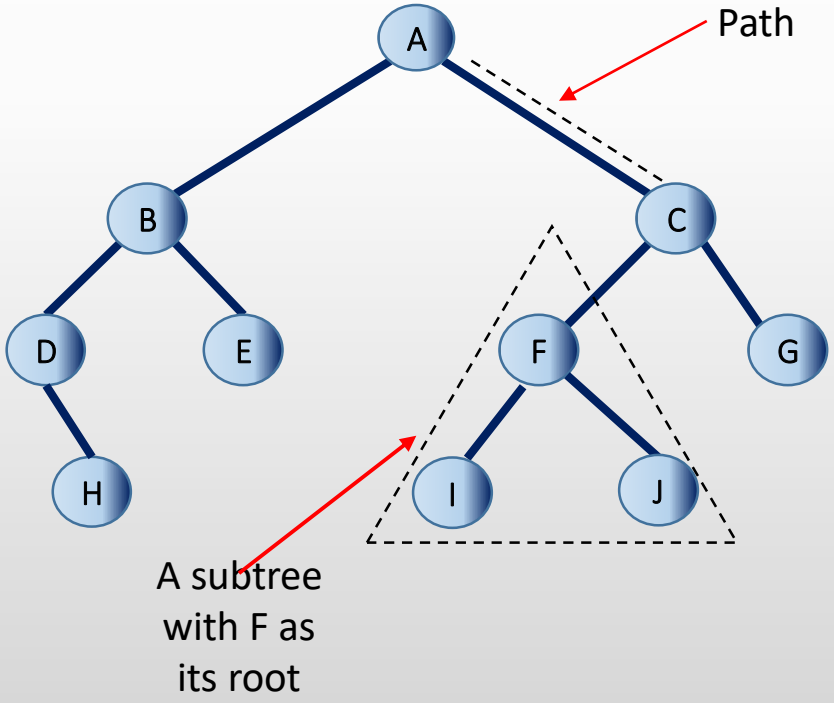
What is root, parent and children in a tree?

- 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 is 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.

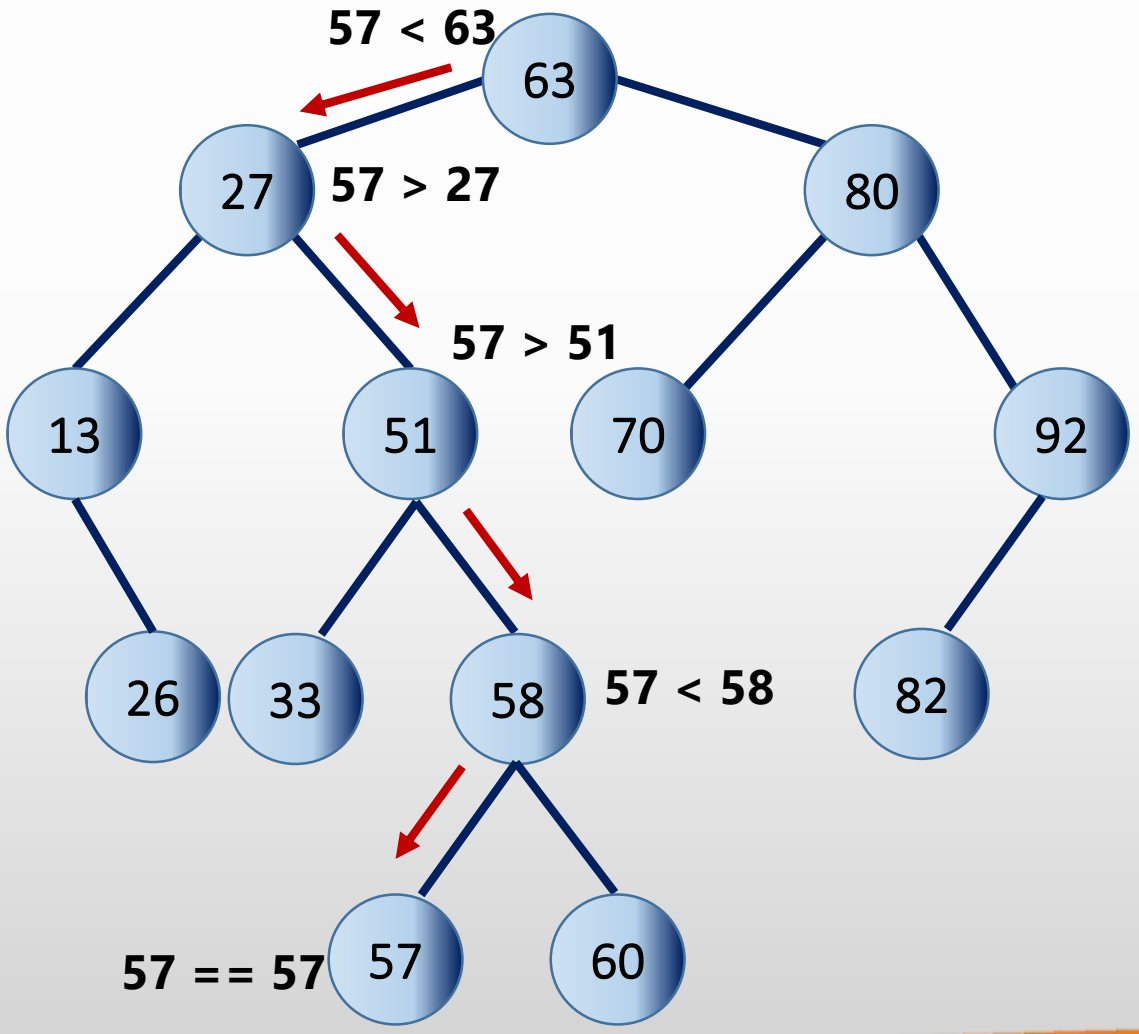
Operations of Binary Search Tree

- 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.

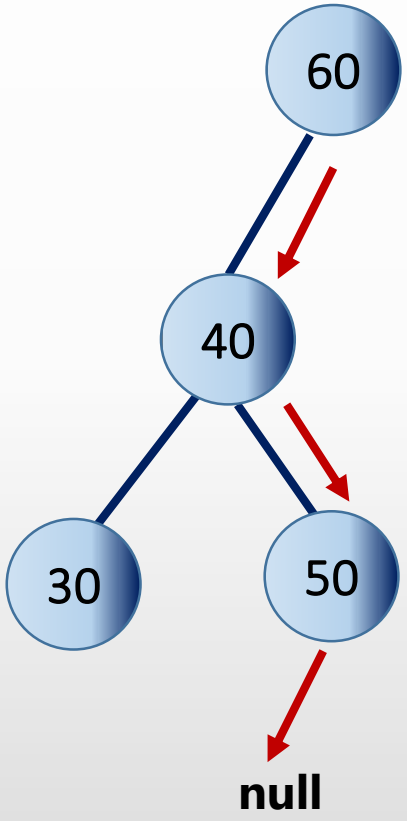
Find value 57



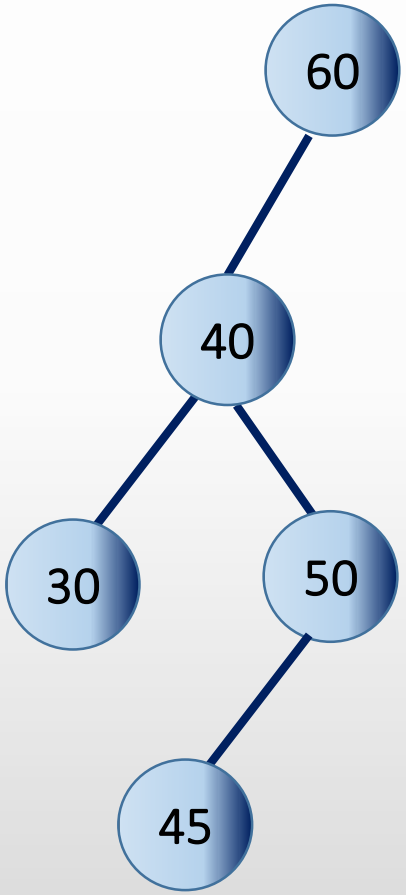
Operations - Insert

- 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



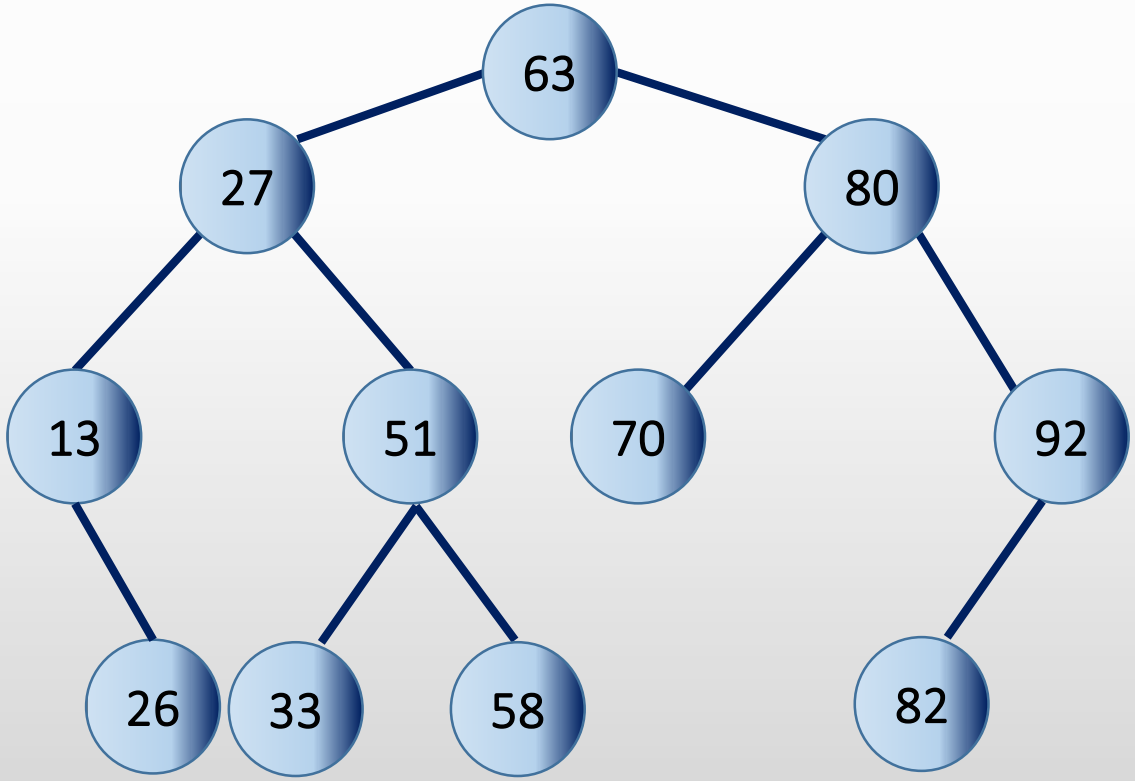
a) Before insertion



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

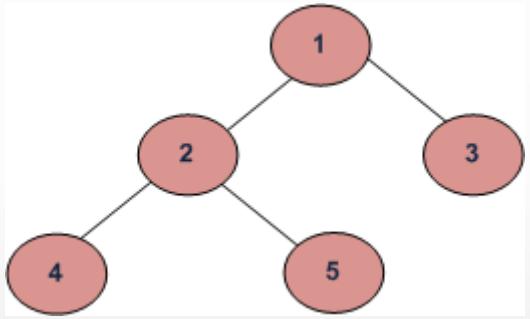
Breadth First Traversal : 1 2 3 4 5

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:

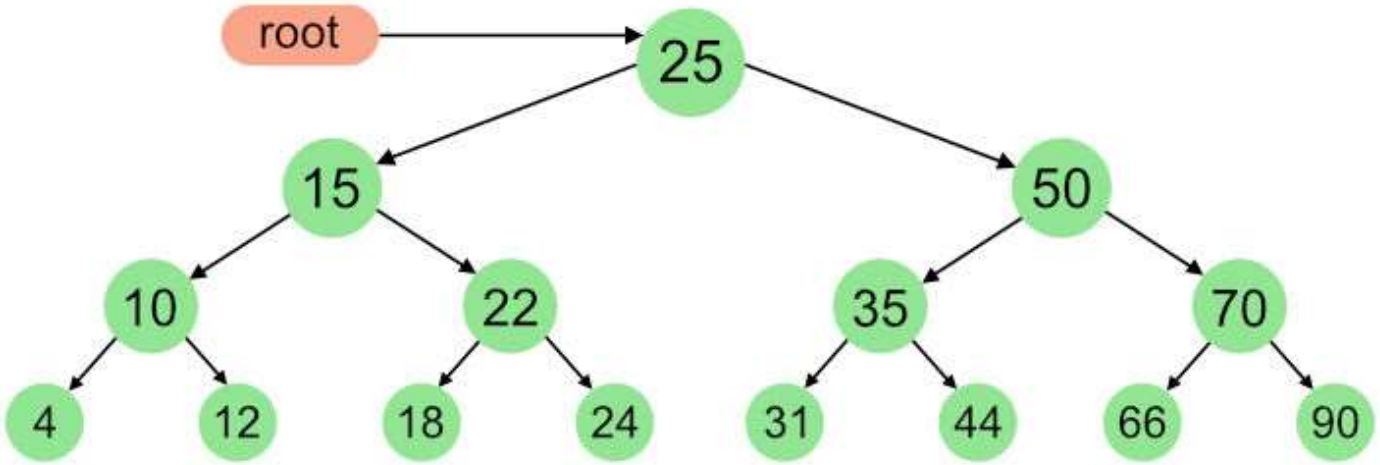
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

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:

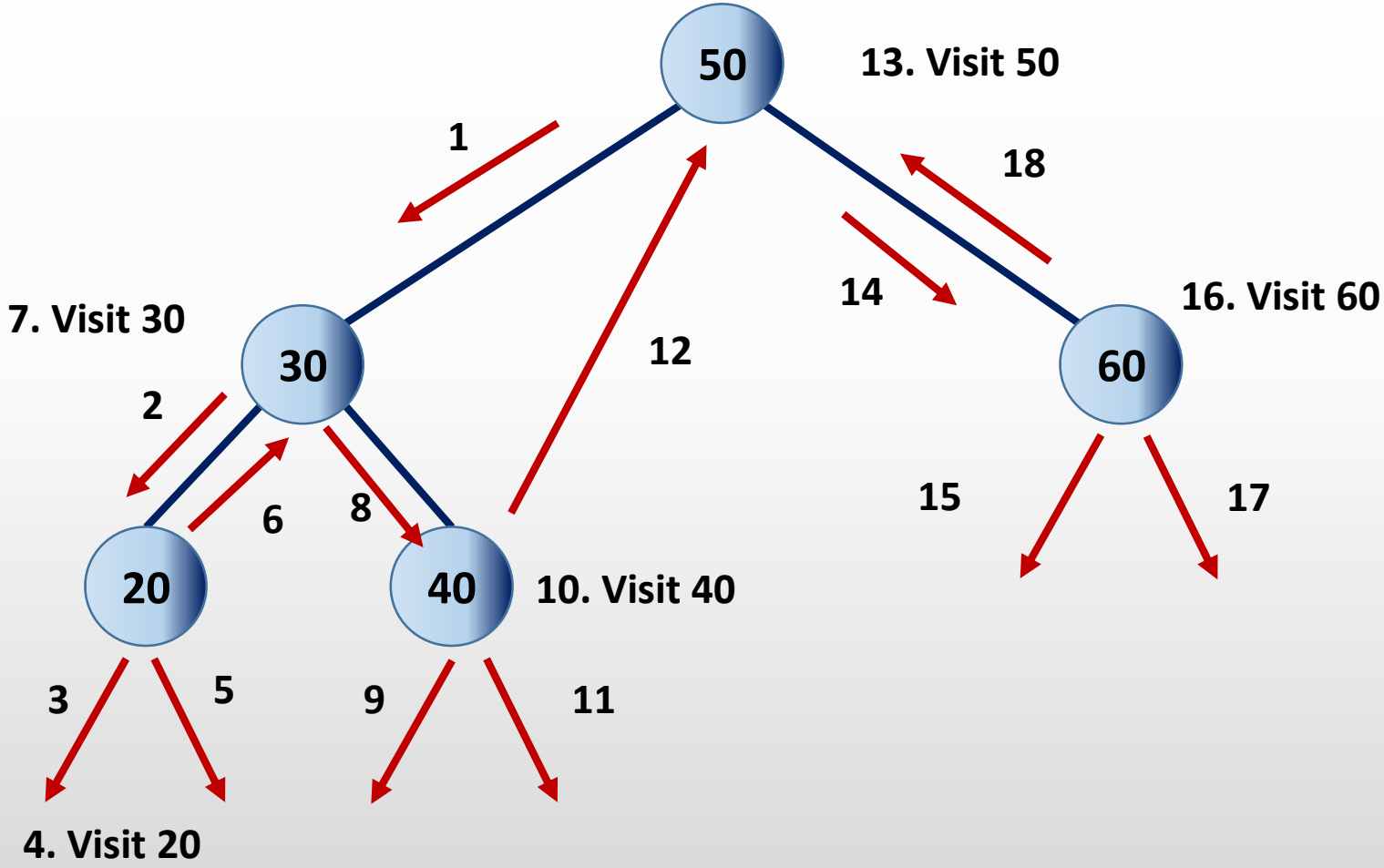
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



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…

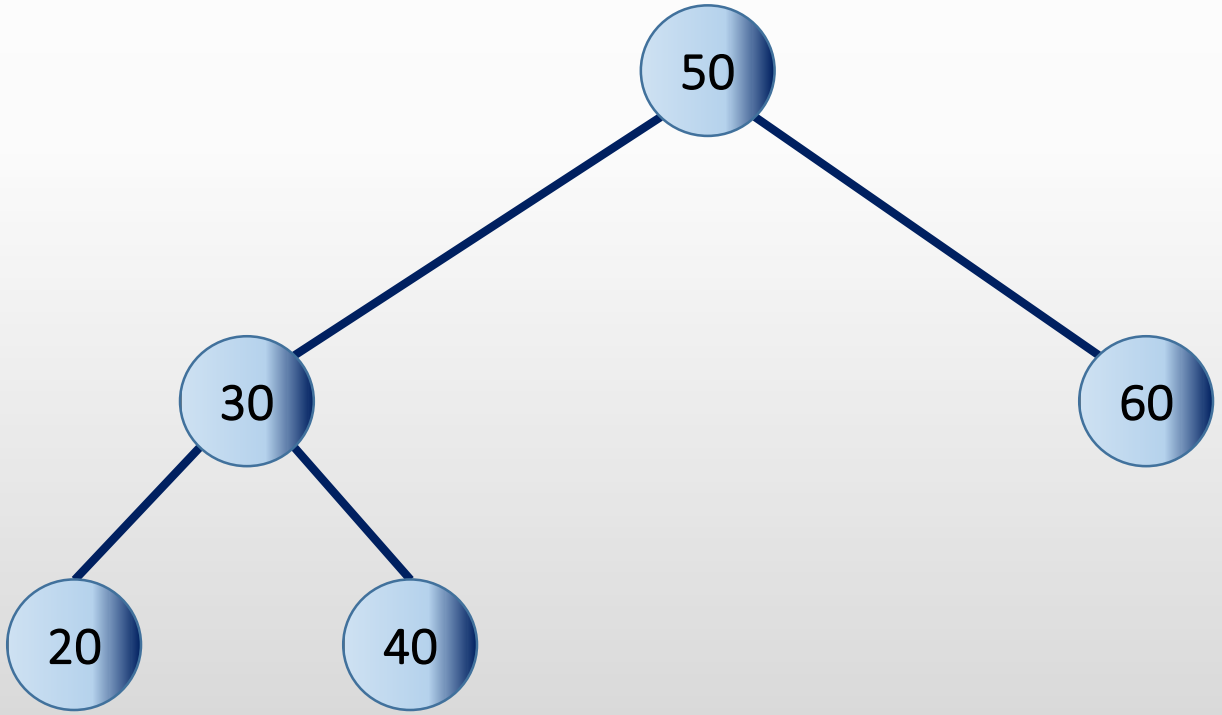


Preorder traversing

- 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.

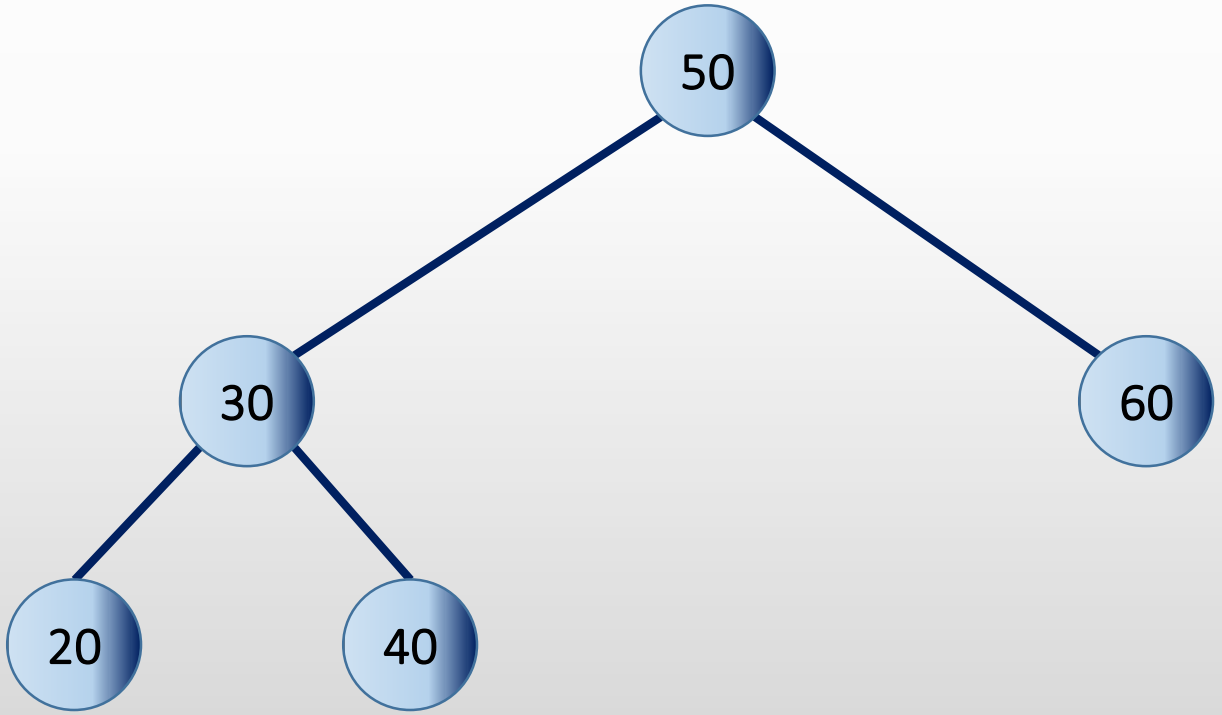


Postorder traversing

- 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 ab
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

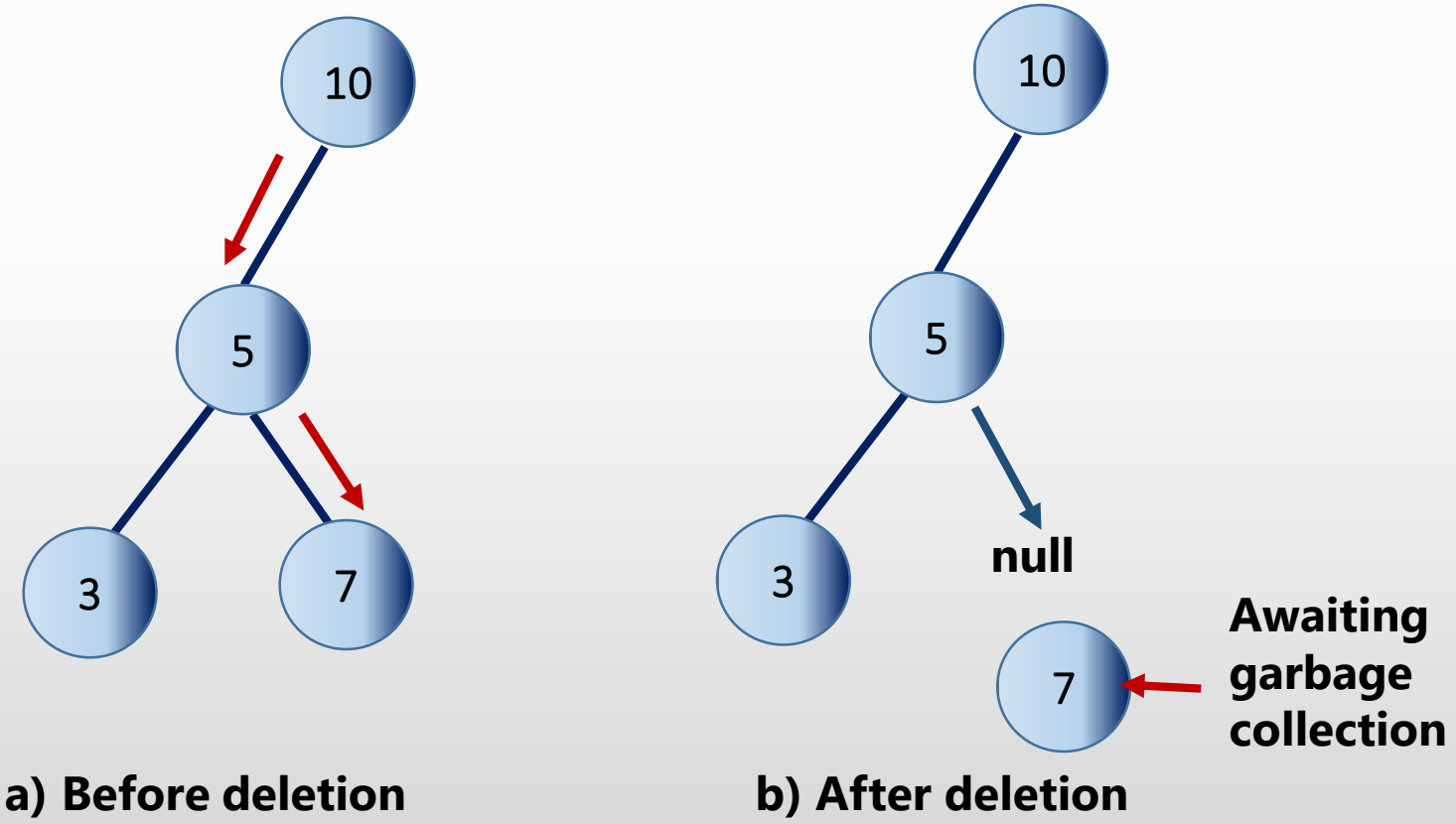
```
private void postOrder(Node localRoot)
{
        if (localRoot != null)
        {
                postOrder(localRoot.leftChild);
                postOrder(localRoot.rightChild);
                localRoot.displayNode();
        }
}
```

Operations - Delete

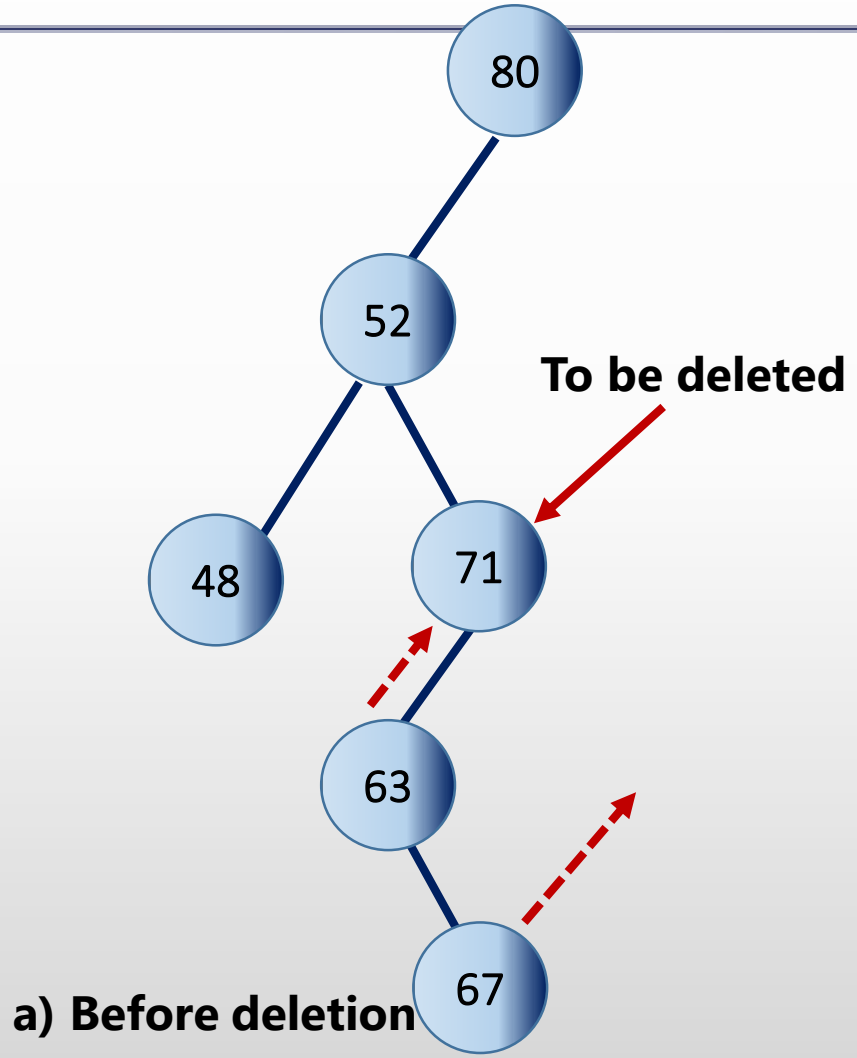
- First find the node to be deleted.
- If the node to be deleted is found there are three cases to be considered. Whether
  - The node to be deleted is a leaf
  - The node to be deleted has one child
  - The node to be deleted has two children.

Case 1 : The node to be deleted has no children

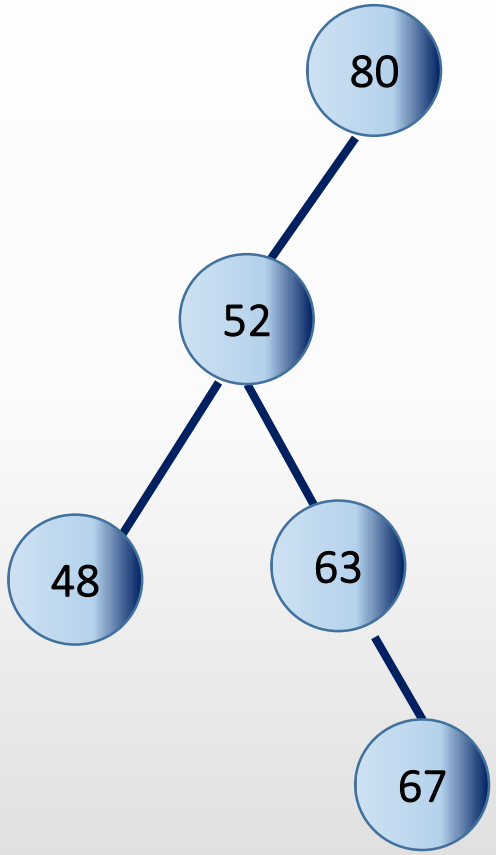
Delete 7



Case 2 : The node to be deleted has one child

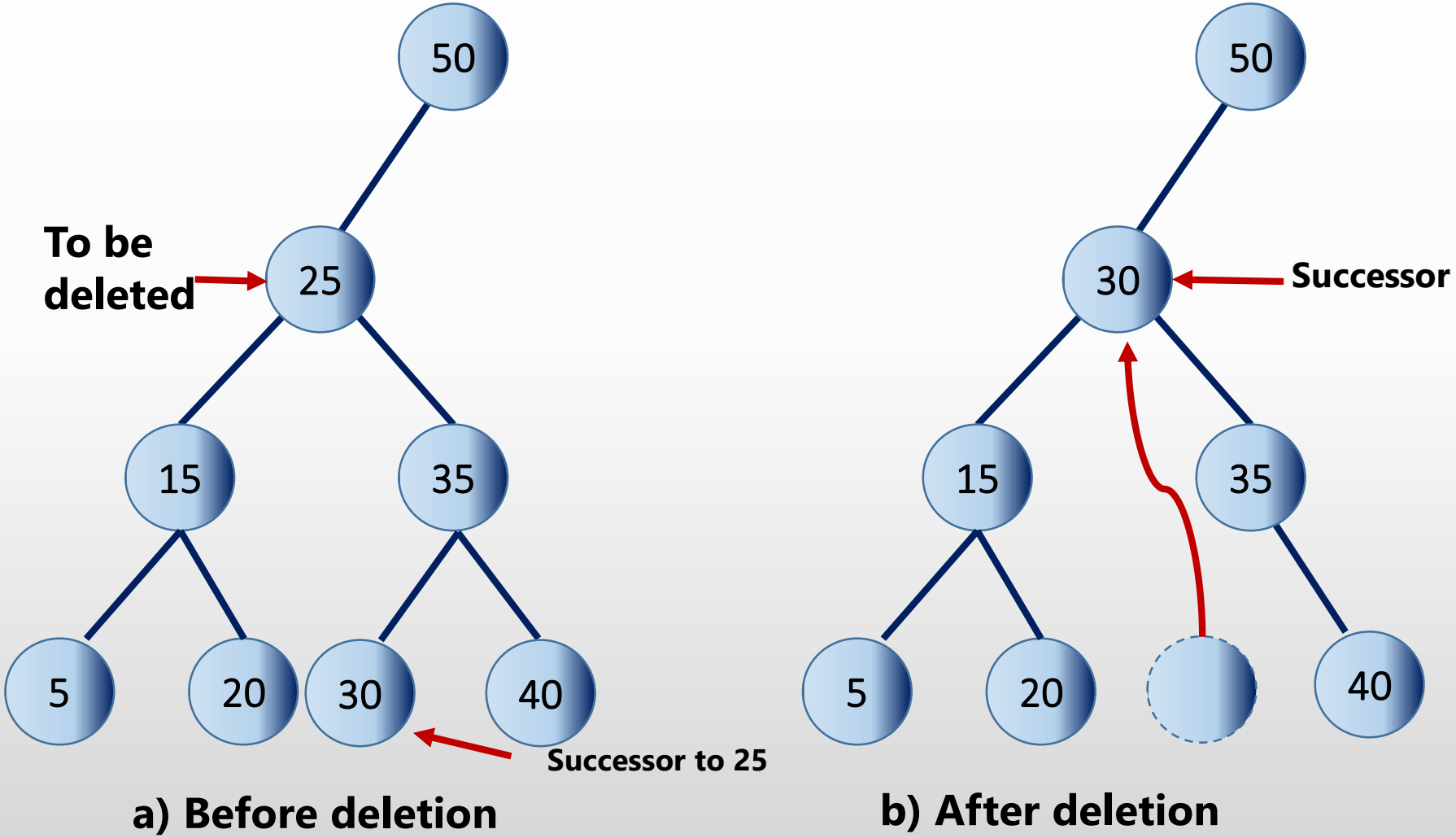


a) Before deletion



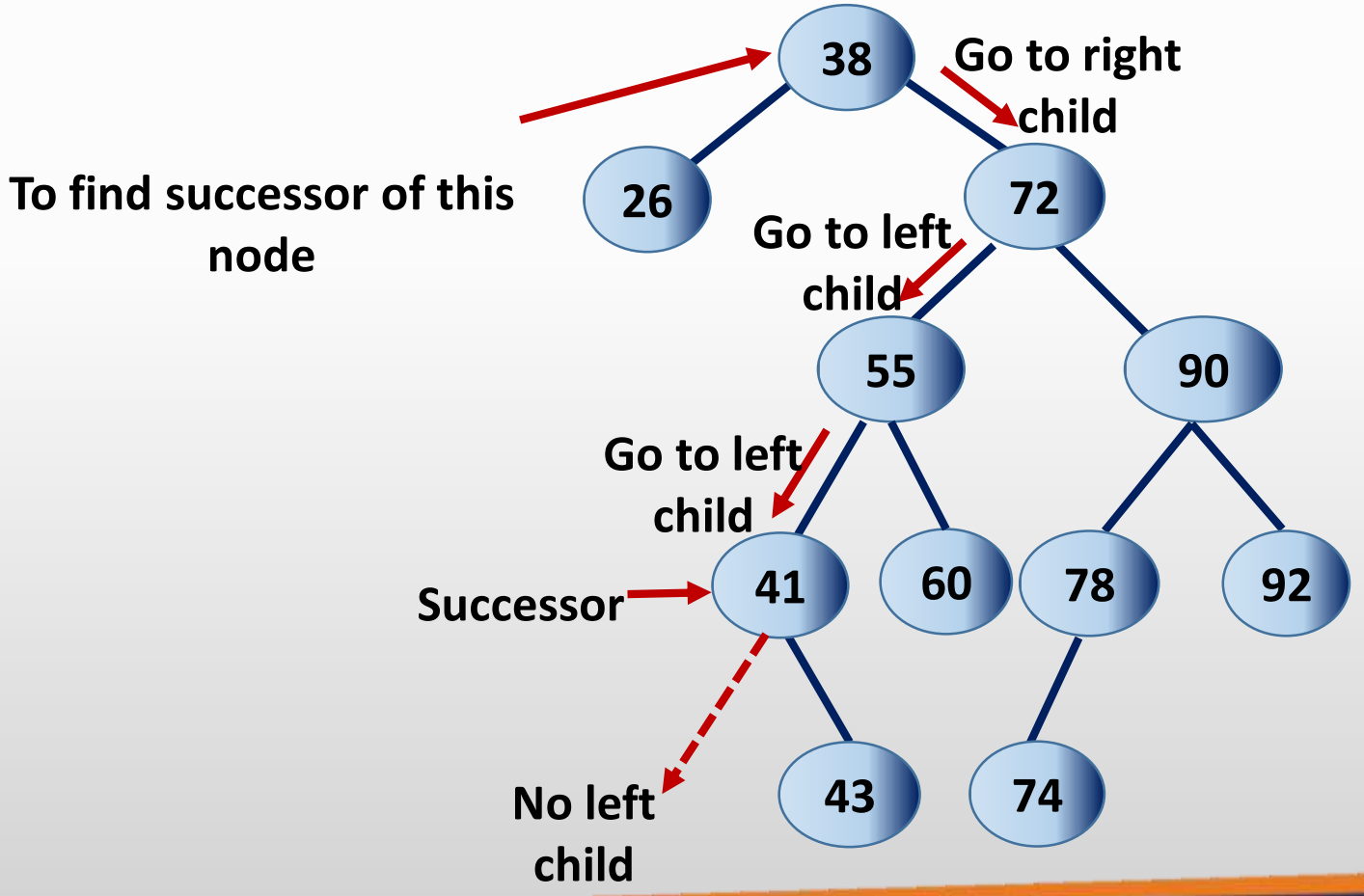
b) After deletion

Case 3 : The node to be deleted has two children



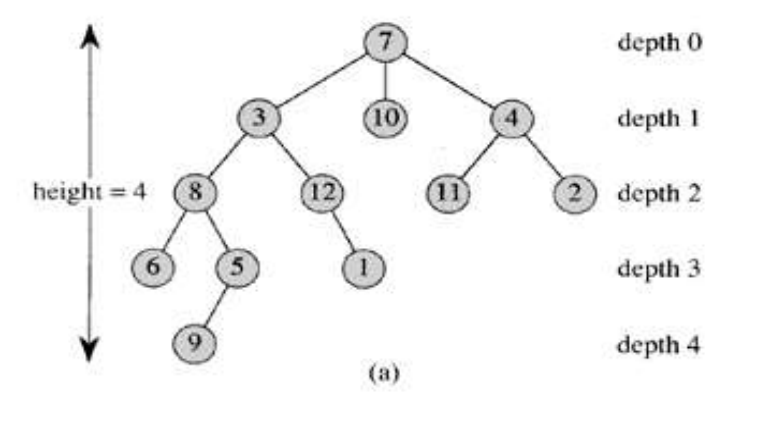
How to find a successor of a node?

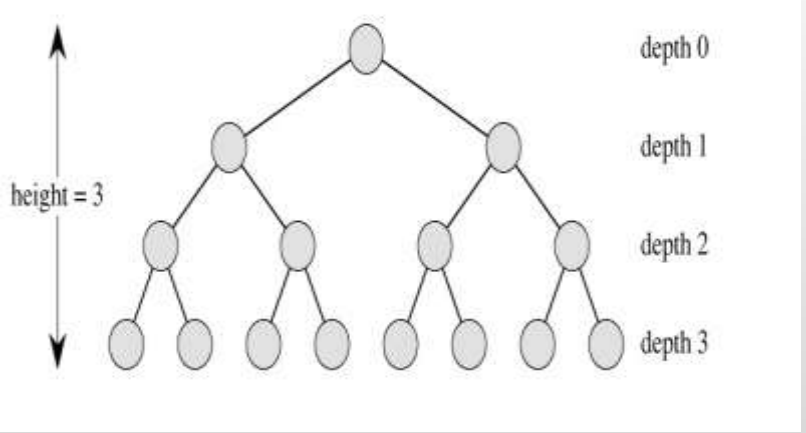
- In a Binary Search Tree, successor of a node is a node with next-highest key.



Tree Terminology

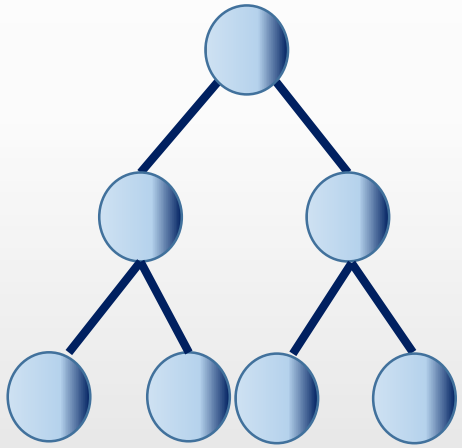
- **Degree of a node:** The number of children it has
- **Depth**: The depth of x in a tree is the length of the path from the root to a node x.
- **Height**: The largest depth of any node in a tree.





Full Binary Tree

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



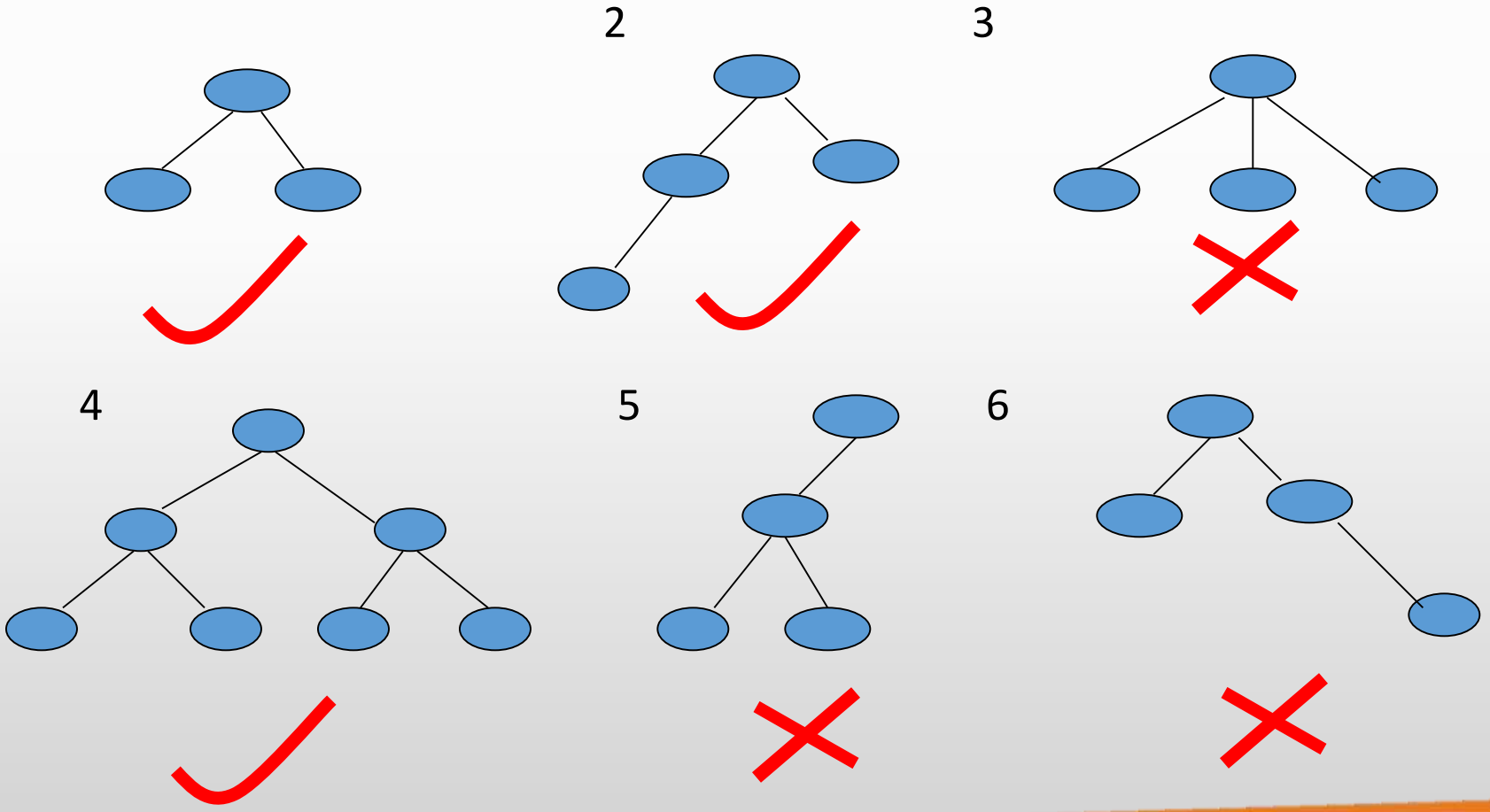
Height, ***h*** = 2, nodes = $2^{2+1}-1 = 7$

Complete Binary Tree

- It is a Binary tree where each node is either a leaf or has degree $\leq$ **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

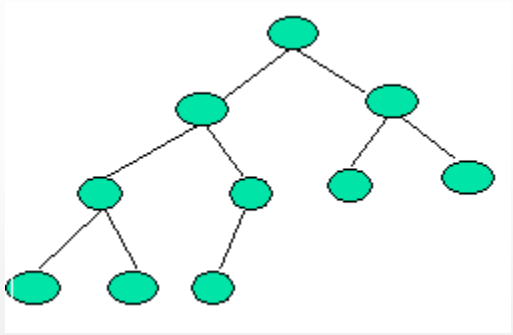
Question

- Find Complete Binary Trees



Height of a complete binary tree

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



Above is a Complete Binary Tree with height = 3

No of nodes: $n = 10$

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

SLIIT
Discover Your Future

IT2070 – Data Structures and Algorithms

Lecture 05

Introduction to Recursion

Recursion –Example 1

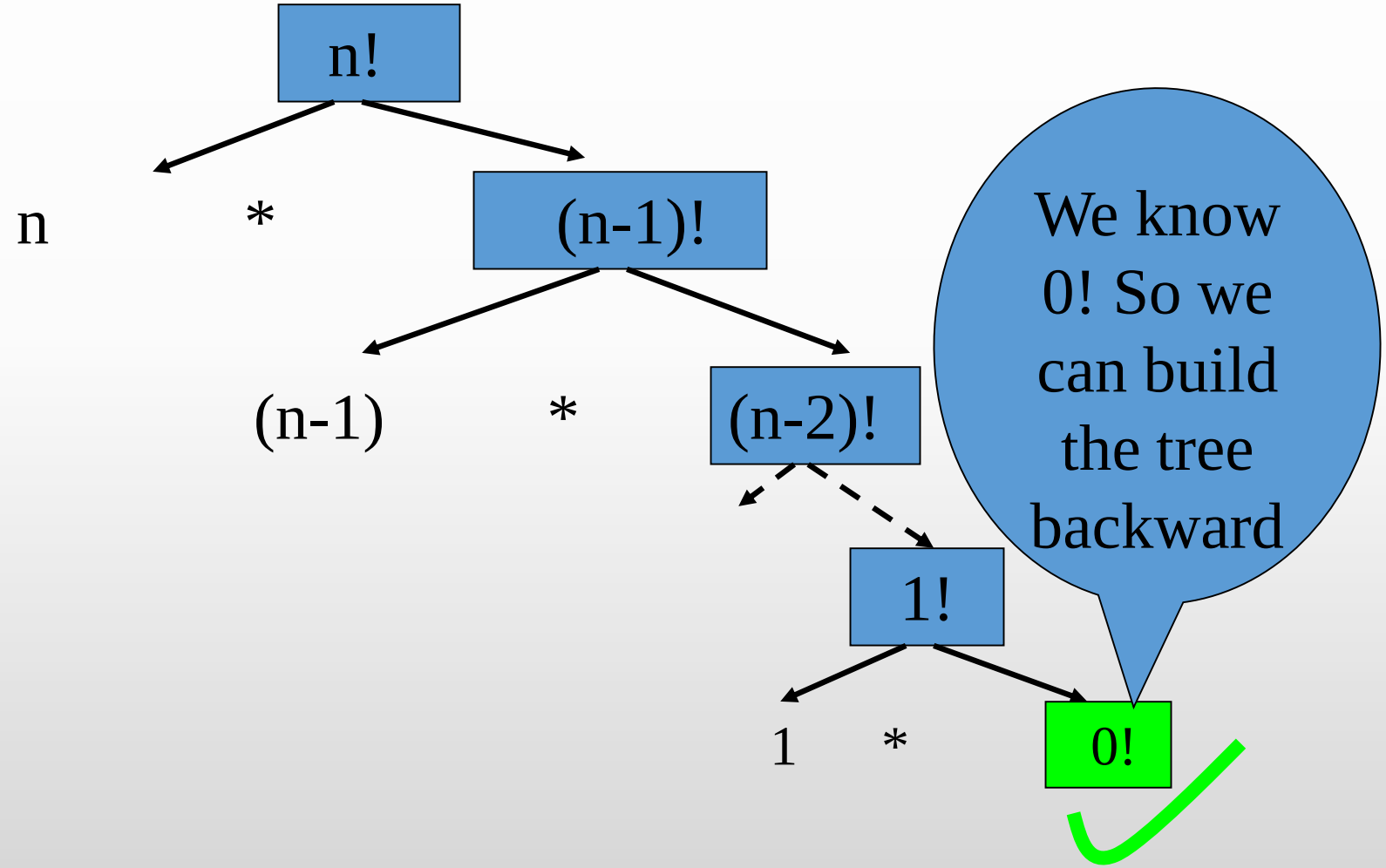
Factorial

$n! = n * (n-1) * (n-2) * \ldots * 2 * 1$, and that $0! = 1$.

A recursive definition is

$$(n)! = \begin{cases} n * (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

Factorial -A graphical view

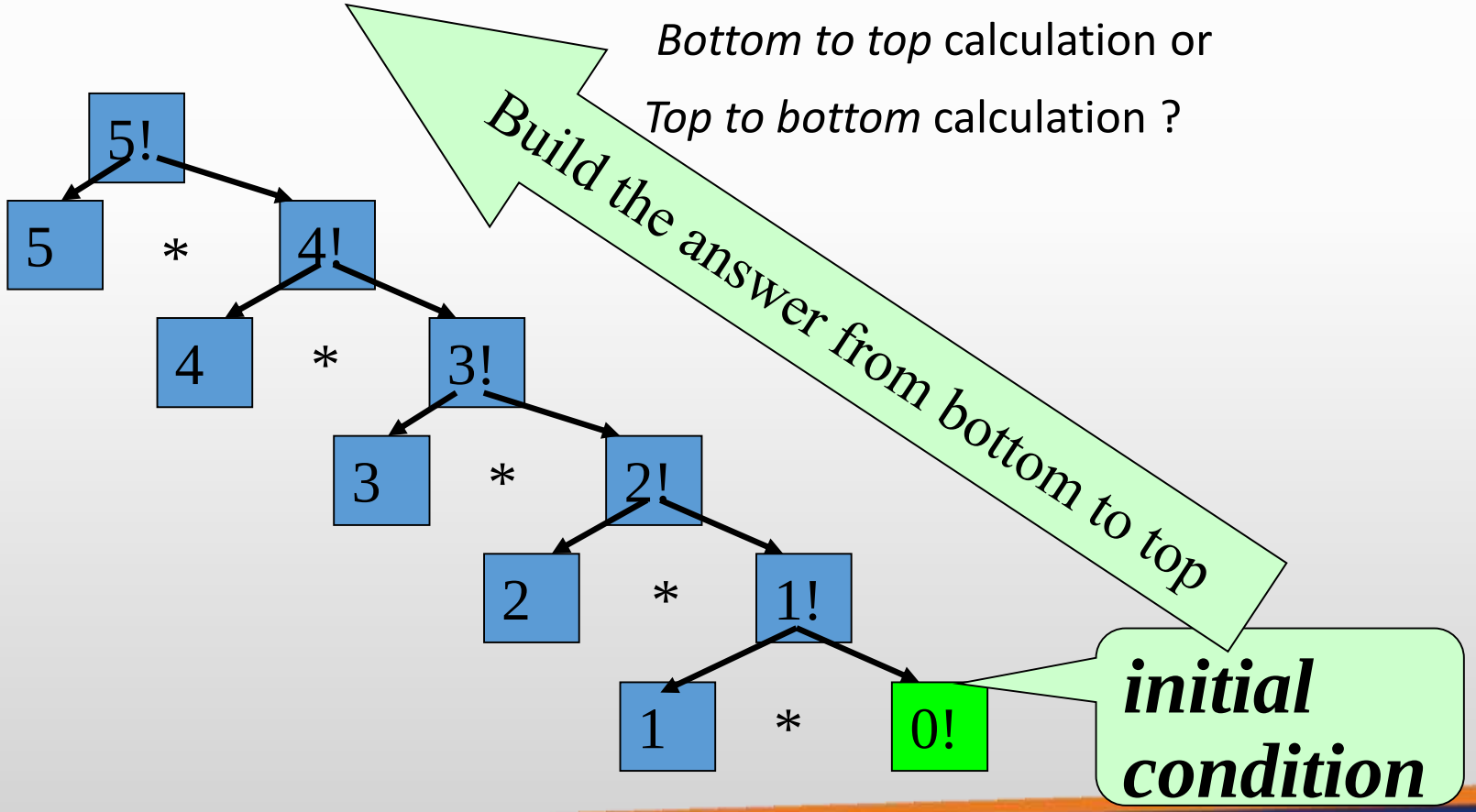


Exercise

- Draw the recursive tree for 5!
- How does it calculate 5! ? Is it:

Bottom to top calculation or

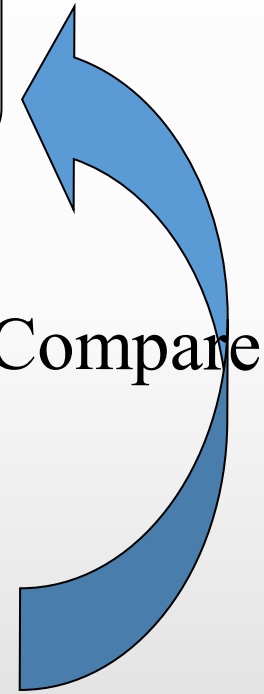
Top to bottom calculation ?



Factorial(contd.)

$$(n)! = \begin{cases} n * (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return (n * factorial(n-1));
}
```



Recursion

What is recursion?

A function that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-call is a ***recursive*** function.

Recurrence equation

- Mathematical function that defines the running time of recursive functions.

- This describes the overall running time on a problem of size *n* in terms of the running time on smaller inputs.

Ex: $T(N) = T(N-1) + b$

$T(N) = T(N/2) + c$

Recurrence - Example1

Find the Running time of the following function

```
int factorial(int n) {
    if (n == 0)              //A
        return 1;            //B
    else
        return (n * factorial(n -1));     //C
}
```

Statement A takes time a → for the conditional evaluation

Statement B takes time b → for the return assignment

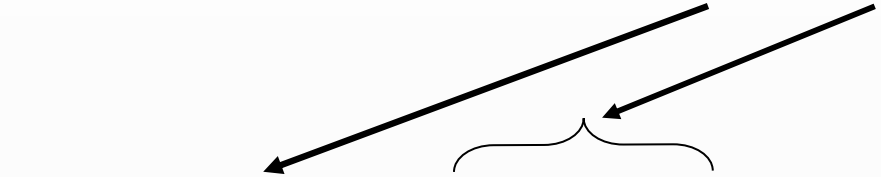
Statement C takes time:

c → for the operations(multiplication & return)

T(n-1) → to determine (n-1)!

Recurrence - Example1 (Contd.)

T(n) = Time to execute A & C



- This method is called iteration method (or repeated substitution)

Exercise

- Solve the recurrence

  T(n) = T(n/2) + 2

  You are given that

  n = 16 and

  T(1) = 1

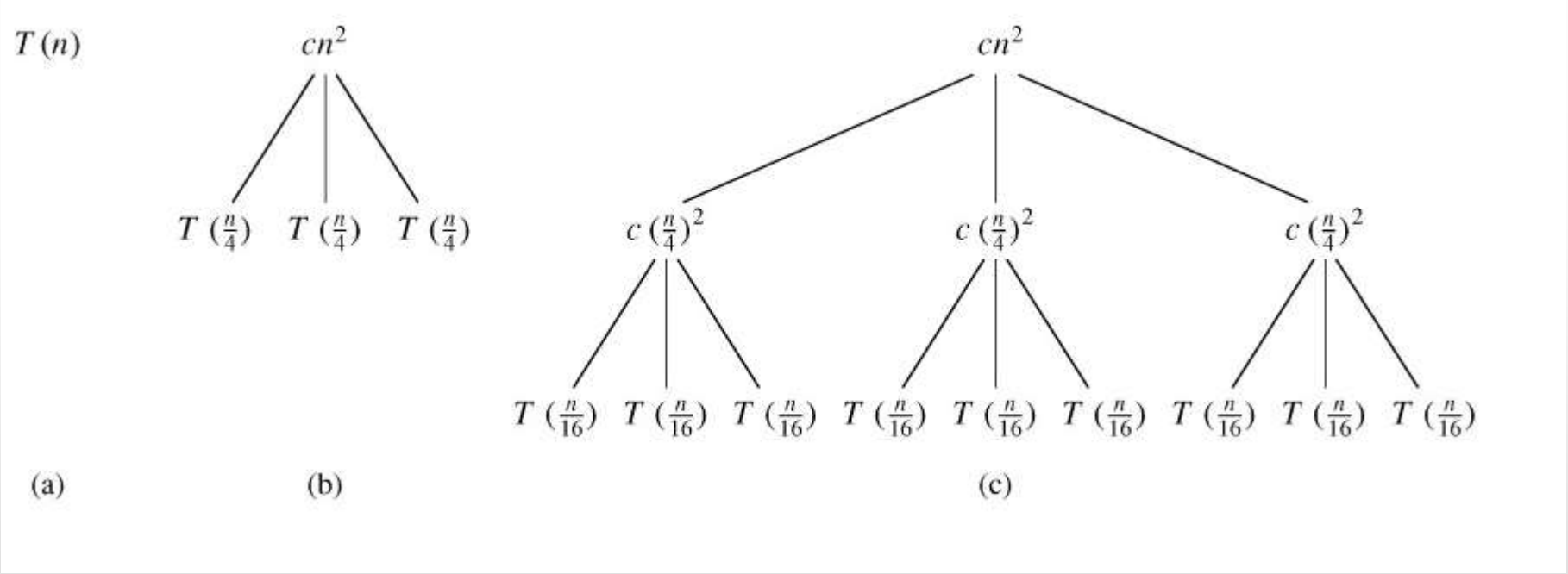
Finding a solution to a recurrence.

- Other methods
  - Recursion tree.
  - Master Theorem.

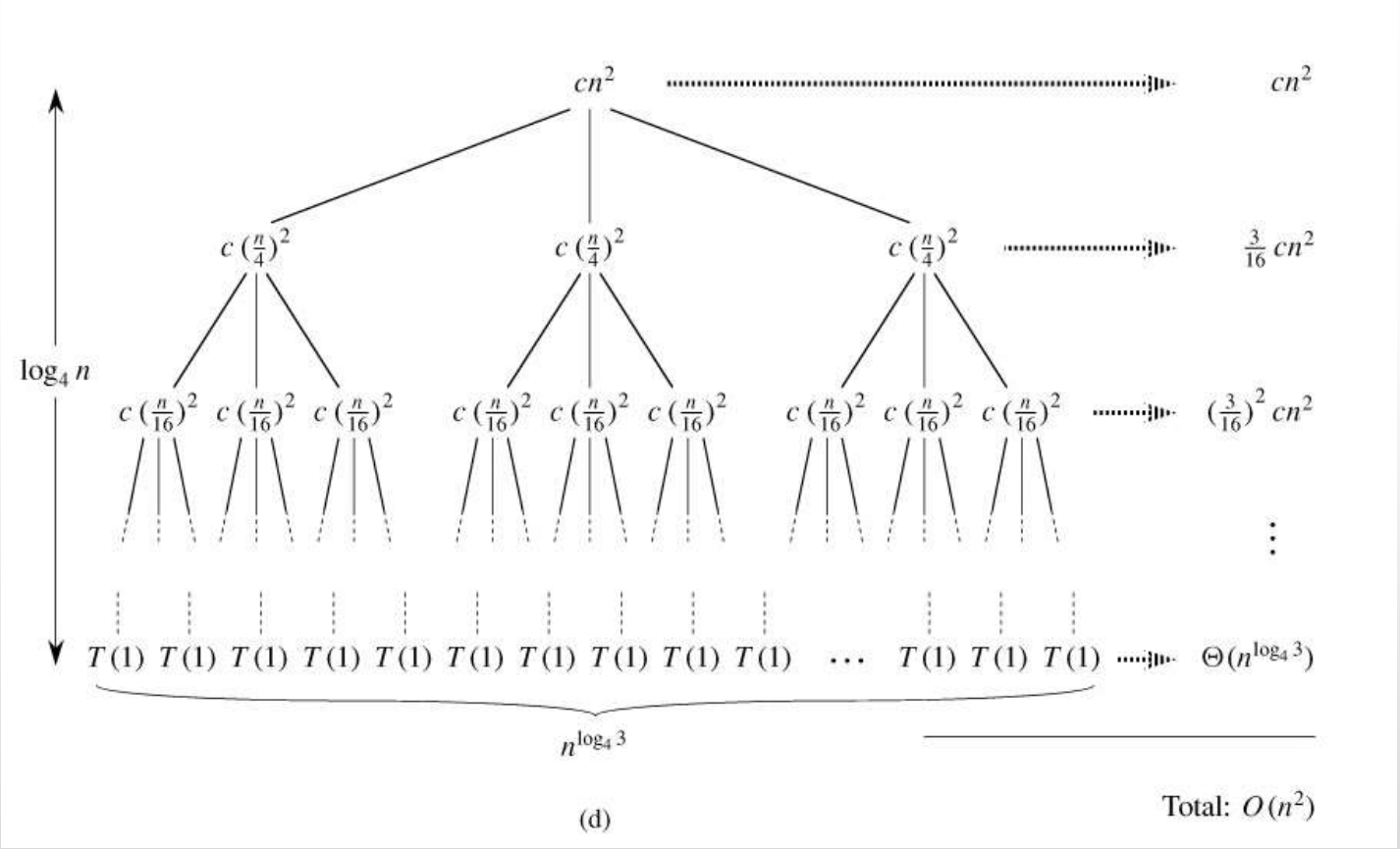
The recursion-tree method

- Although the substitution method can provide a succinct proof that a solution to a recurrence is correct, it is sometimes difficult to come up with a good guess. Drawing out a recursion tree,is a straightforward way to devise a good guess. In a **recursion tree**, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.
- A recursion tree is best used to generate a good guess, which is then verified by the substitution method.

Recursion tree for $T(n) = 3T(n/4) + cn^2$



Recursion tree for $T(n) = 3T(n/4) + cn^2$



The Master Method

- The Master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$, and $f(n)$ is an asymptotically positive function.

The recurrence describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time T (n/b). The cost of dividing the problem and combining the results of the subproblems is described by the function f (n).

The master theorem

- The master method depends on the following theorem.
- Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

The master theorem

Compare $n^{\log_b a}$ vs. $f(n)$:

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.
($f(n)$ is polynomially smaller than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(n^{\log_b a})$.

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$.

($f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.)
Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
(Intuitively: cost is $n^{\log_b a} \lg^k n$ at each level, and there are $\Theta(\lg n)$ levels.)
Simple case: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.
($f(n)$ is polynomially greater than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(f(n))$.
(Intuitively: cost is dominated by root.)

The master theorem

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \rightarrow f(n) < n^{\log_b a} \\ \Theta\left(n^{\log_b a} \lg n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \rightarrow f(n) = n^{\log_b a} \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \rightarrow f(n) > n^{\log_b a} \\ & \text{if } af(n/b) \le cf(n) \text{ for } c < 1 \text{ and large } n \end{cases}$$

Master Theorem – Case 1 example

Give tight asymptotic bound for

$$T(n) = 9T(n/3) + n$$

Solution:

a=9, b=3, and $f(n) = n$.

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

$$f(n) = O(n^{\log_3 9 - \varepsilon}) \text{ for } \varepsilon = 1 \text{ or } f(n) < n^{\log_3 9} \rightarrow \text{case1}$$

$$\therefore T(n) = \Theta(n^2)$$

Master Theorem – Case 2 example

Give tight asymptotic bound for

$$T(n) = T(2n/3) + 1$$

Solution:

a=1, b=3/2, and $f(n) = 1$.

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

$$f(n) = \Theta(n^{\log_b a}) \text{ or } f(n) = n^{\log_b a} \rightarrow \text{case } 2$$

$$\therefore T(n) = \Theta(\log n)$$

Master Theorem – Case 3 example

- Give tight asymptotic bound for
- $T(n) = 3T(n/4) + n \log n$
- **Solution:**
- a=3, b=4, and $f(n) = n \log n$

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

$$f(n) = \Omega(n^{\log_4 3+\varepsilon}), \text{for } \varepsilon \approx 0.2 \text{or } f(n) > n^{\log_4 3} \rightarrow \text{case3}$$

$$Note: n \lg n \geq c.n^{\log_4 3}.n^{0.2}$$

Exercises.

- Use the master method to give tight asymptotic bounds for the following recurrences.
  1. $T(n) = 4T(n/2) + n$.
  2. $T(n) = 4T(n/2) + n^2$.
  3. $T(n) = 4T(n/2) + n^3$.
- Use the master method to show that the solution to the binary-search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(\lg n)$.

SLIIT
Discover Your Future

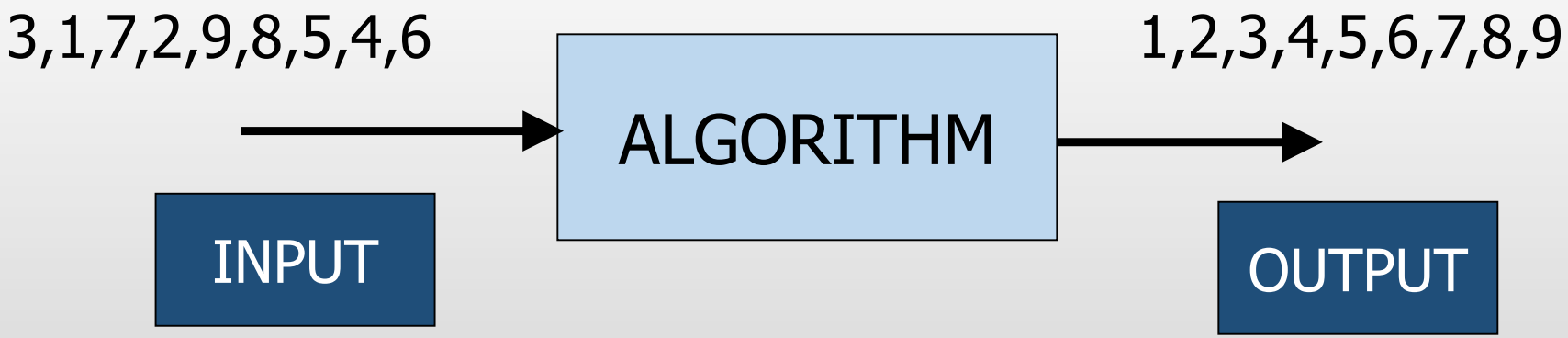
IT2070 – Data Structures and Algorithms

Lecture 06

Introduction to Algorithms

ALGORITHMS

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

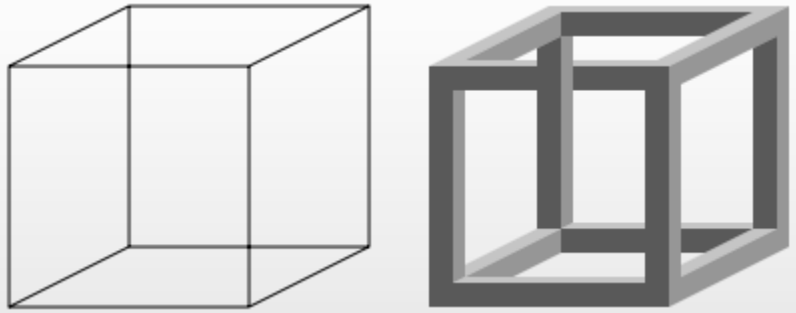


ALGORITHM (Contd.)

1. Get the smallest value from the input.
2. Remove it and output.
3. Repeat above 1,2 for remaining input until there is no item in the input.

Properties of an Algorithm.

- Be correct.
- Be unambiguous.
- Give the correct solution for all cases.
- Be simple.
- It must terminate.



Necker_cube_and_impossible_cube

Source:http://en.wikipedia.org/wiki/Ambiguity#Mathematical_interpretation_of_ambiguity

Applications of Algorithms

- Data retrieval
- Network routing
- Sorting
- Searching
- Shortest paths in a graph

Pseudocode

- Method of writing down a algorithm.
- Easy to read and understand.
- Just like other programming language.

  - More expressive method.
  - Does not concern with the technique of software engineering.

Pseudocode Conventions.

- English.
- Indentation.
- Separate line for each instruction.
- Looping constructs and conditional constructs.
- // indicate a comment line.
- = indicate the assignment.

Pseudocode Conventions.

- Array elements are accessed by specifying the array name followed by the index in the square bracket.
- The notation ".." is used to indicate a range of values within the array.

Ex:

A[1..i] indicates the sub array of A consisting of elements A[1] , A[2] , .. , A[i].

Analysis of Algorithms

Idea is to predict the resource usage.

- Memory
- Logic Gates
- **Computational Time**

Why do we need an analysis?

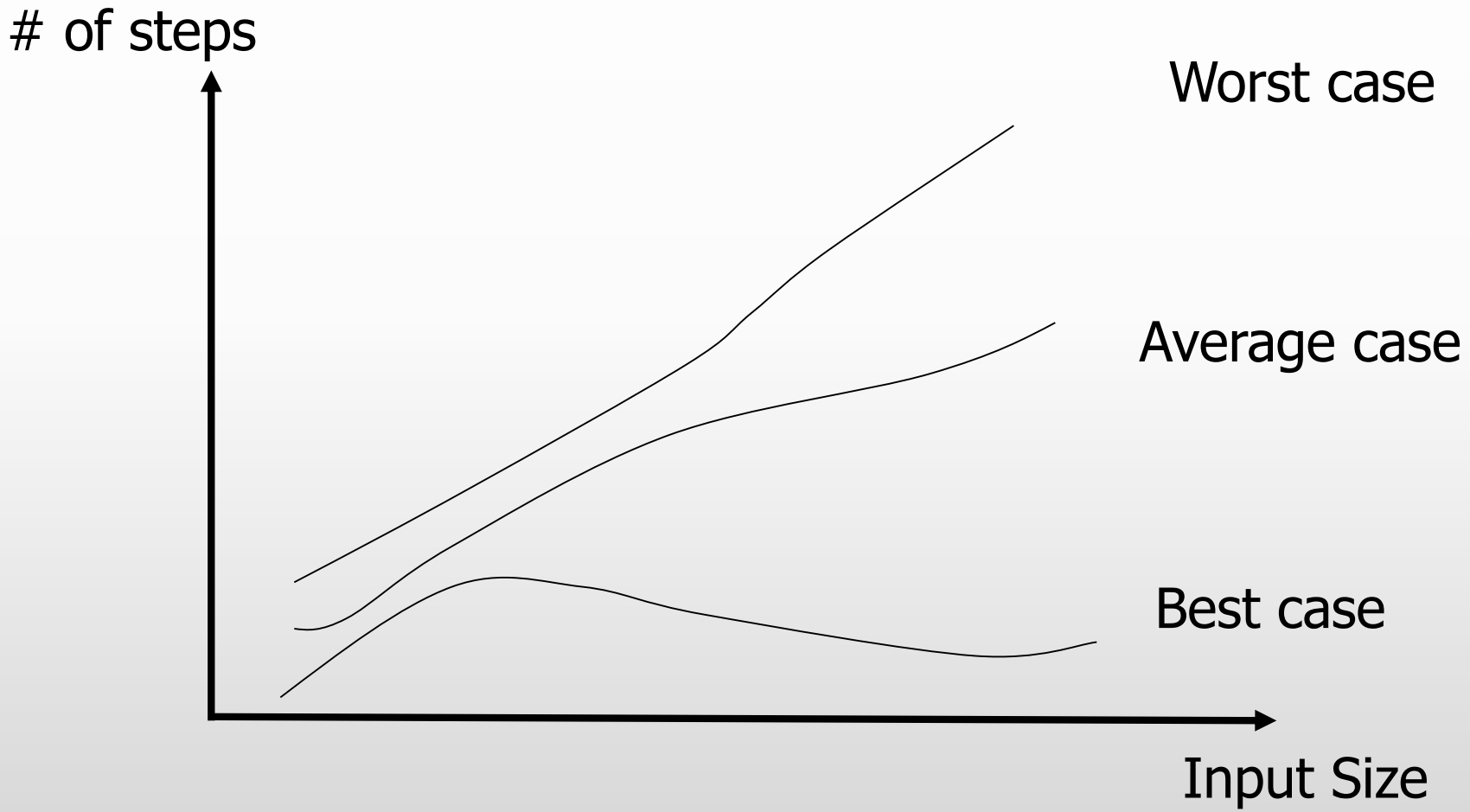
- To compare
- Predict the growth of run time

Worst, Best and Average case.

Running time will depend on the chosen instance characteristics.

- **Best case**:
  Minimum number of steps taken on any instance of size n.
- **Worst case:**
  Maximum number of steps taken on any instance of size n.
- **Average case:**
  An average number of steps taken on any instance of size n.

Worst,Best and Average case(Contd.)



Analysis Methods

- Operation Count Methods
- Step Count Method(RAM Model)
- Exact Analysis
- Asymptotic Notations

Operation count

- Methods for time complexity analysis.
- Select one or more operations such as add, multiply and compare.
- Operation count considers the **time spent on chosen operations** but not all.

Step Count (RAM Model)

- Assume a generic one processor.
- Instructions are executed one after another, with no concurrent operations.
- +, - , =, it takes exactly one step.
- Each memory access takes exactly 1 step.
- **Running Time = Sum of the steps.**

RAM Model Analysis.

Example1:

n = 100	1step
n = n + 100	2steps
Print n	1step

Steps = 4

Example2:

$sum = 0$ ..... 1 assignment

$\text{for } i = 1 \text{ to } n$ ..... n+1 assignments, n+1 comparisons, n additions

$sum = sum + A[i]$ ..... n assignments, n additions, n memory accesses

Steps = 6n+3

Question 01

- Using RAM model analysis, find out the no of steps needed to display the numbers from 1 to 10.

i = 1 → 1 step

While i <=10 → 11 steps

print i → 10 steps

i = i + 1 → 10 + 10 = 20 steps

Steps = 42

Question 02

- Using RAM model analysis, find out the no of steps needed to display the numbers from 10 to 20.

i = 10 → 1 step

While i <= 20 → 12 steps (Hint :20 – 10 + 2 = 12)

print i → 11 steps

i = i + 1 → 11 + 11 = 22 steps

Steps = 46

Question 03

- Using RAM model analysis, find out the no of steps needed to display the even numbers from 10 to 20.

for i = 10 to 20 → (12+ 12 + 11) steps = 35 steps

if i % 2 == 0 → 2 * 11 = 22 steps

print i → 6 steps

Steps = 63

Problems with RAM Model

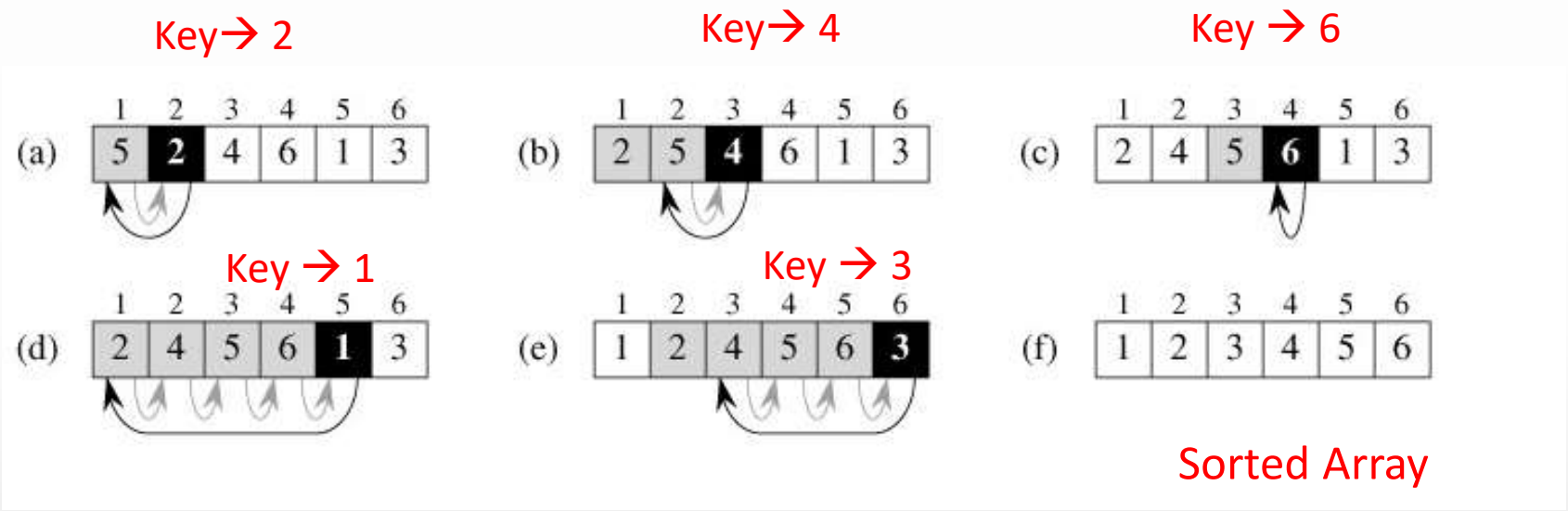
- Differ number of steps with different architecture.

  eg: sum = sum + A[i] is a one step in the CISC processor.

- It is difficult to count the exact number of steps in the algorithm.

  eg: See the insertion sort , efficient algorithm for sorting small number of elements.

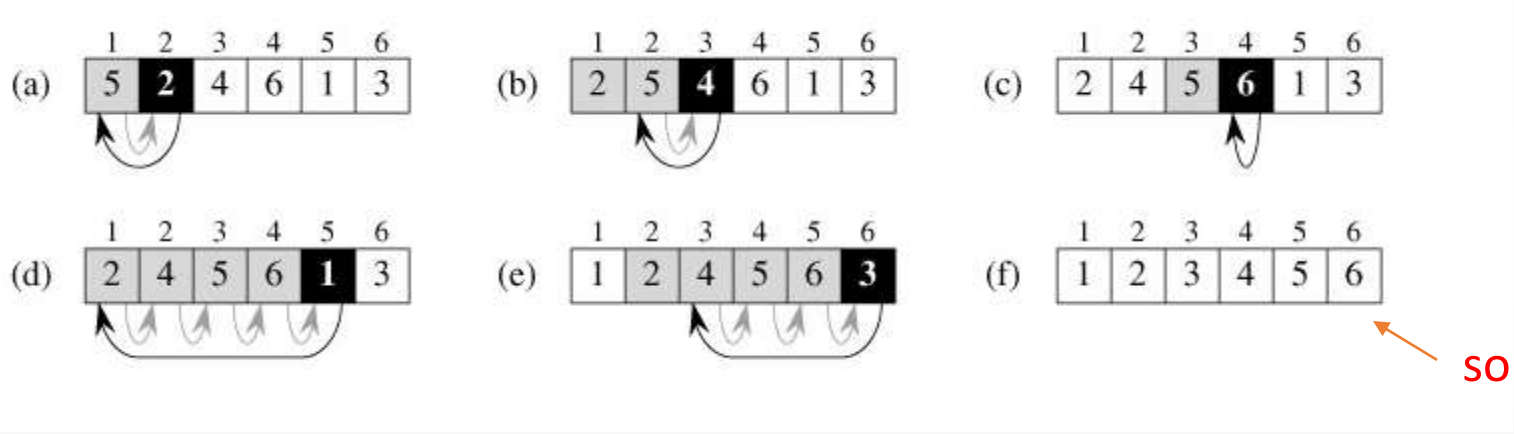
Insertion sort



Pseudocode for insertion sort.

```
INSERTION-SORT(A)
1 for j = 2 to A.length
2         key = A[j]
3         // Insert A[j] into the sorted sequence A[1..j-1]
4          i = j - 1
5          While i > 0 and A[i] > key
6                    A[i+1] = A[i]
7                     i = i-1
8          A[i+1]  = key
```

Insertion sort - Example



- *(a)-(e)* The iterations of the *for* loop → lines 1-8.
- In each iteration, the black rectangle holds the key taken from *A*[*j*],
- Key is compared with the values in shaded rectangles to its left → line 5.
- Shaded arrows show array values moved one position to the right → line 6,
- Black arrows indicate where the key is moved to → line 8.

Exact analysis of Insertion sort

- Time taken for the algorithm will depend on the input size (number of elements of the array)

Running Time (Time complexity):

This is the number of primitive operations or steps executed through an algorithm given a particular input.

Running Time : T(n)

	INSERTION-SORT(A)	Cost	Times
1	**for** j = 2 **to** A.length	c_1	**n**
2	key = A[j]	c_2	**n-1**
3	// Insert A[j] into the sorted // sequence A[1..j-1]	**0**	**n-1**
4	i = j – 1	c_4	**n-1**
5	**While** i > 0 **and** A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
6	A[i+1] = A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	i = i-1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	A[i+1] = key	c_8	**n-1**

i^{th} line takes time c_i where c_i is a constant.

For each j=2,3,…,n , t_j be the number of times the while loop is executed for that value of j

Running Time(contd.)

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} t_j$$

$$+ c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1)$$

- Best Case (Array is in sorted order)
  - $T(n) \rightarrow an+b$

- Worst Case (Array is in reverse sorted order)
  - $T(n) \rightarrow cn^2 + dn + e$

Worst Case T(n) $\rightarrow cn^2 + dn + e$

Worst case: The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.

Worst Case T(n) $\rightarrow cn^2 + dn + e$

- Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

  [The parentheses around the summation are not strictly necessary. They are there for clarity, but it might be a good idea to remind the students that the meaning of the expression would be the same even without the parentheses.]

- Letting $k = j - 1$, we see that $\sum_{j=2}^{n}(j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.

- Running time is

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) .
\end{aligned}
$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

Asymptotic Notations

- RAM Model has some problems.
- Exact analysis is very complicated.

Therefore we move to **asymptotic notation**

- Here we focus on determining the biggest term in the complexity function.
- Sufficiently large size of n.

Asymptotic Notations(Contd.)

- There are three notations.

O - Notation

Θ - Notation

Ω - Notation

Big O - Notation

- Introduced by Paul Bechman in 1892.
- We use Big O-notation to give an upper bound on a function.

Definition:

$O(g(n))$ = { $f(n)$: there exist positive constants c and n_o such that

$0 \leq f(n) \leq cg(n)$ for all $n \geq n_o$}.

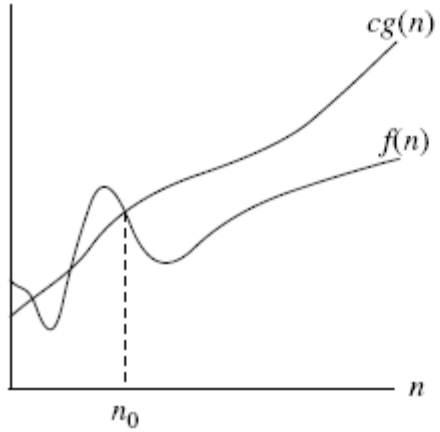
Eg: What is the big O value of f(n)=2n + 6 ?

c = 4

n_o = 3

g(n)=n therefore

f(n) = O(n)



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.
If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$

Back to the example

- Alternative calculation:

	cost	times
sum = 0	c_1	1
for *i* = 1 to *n*	c_2	*n*+1
sum = sum + A[*i*]	c_3	*n*

$$T(n) = c_1 + c_2\,(n+1) + c_3\,n$$

$$= (c_1 + c_2) + (c_2 + c_3)\,n$$

$$= c_4 + c_5\,n \quad \rightarrow O(n)$$

Proof: $c_4 + c_5\,n \le c\,n$ → TRUE for $n \ge 1$ and $c \ge c_4 + c_5$

Big O – Notation(Contd.)

Assignment (s = 1)

Addition (s+1) O(1)

Multiplication (s*2)

Comparison (S<10)

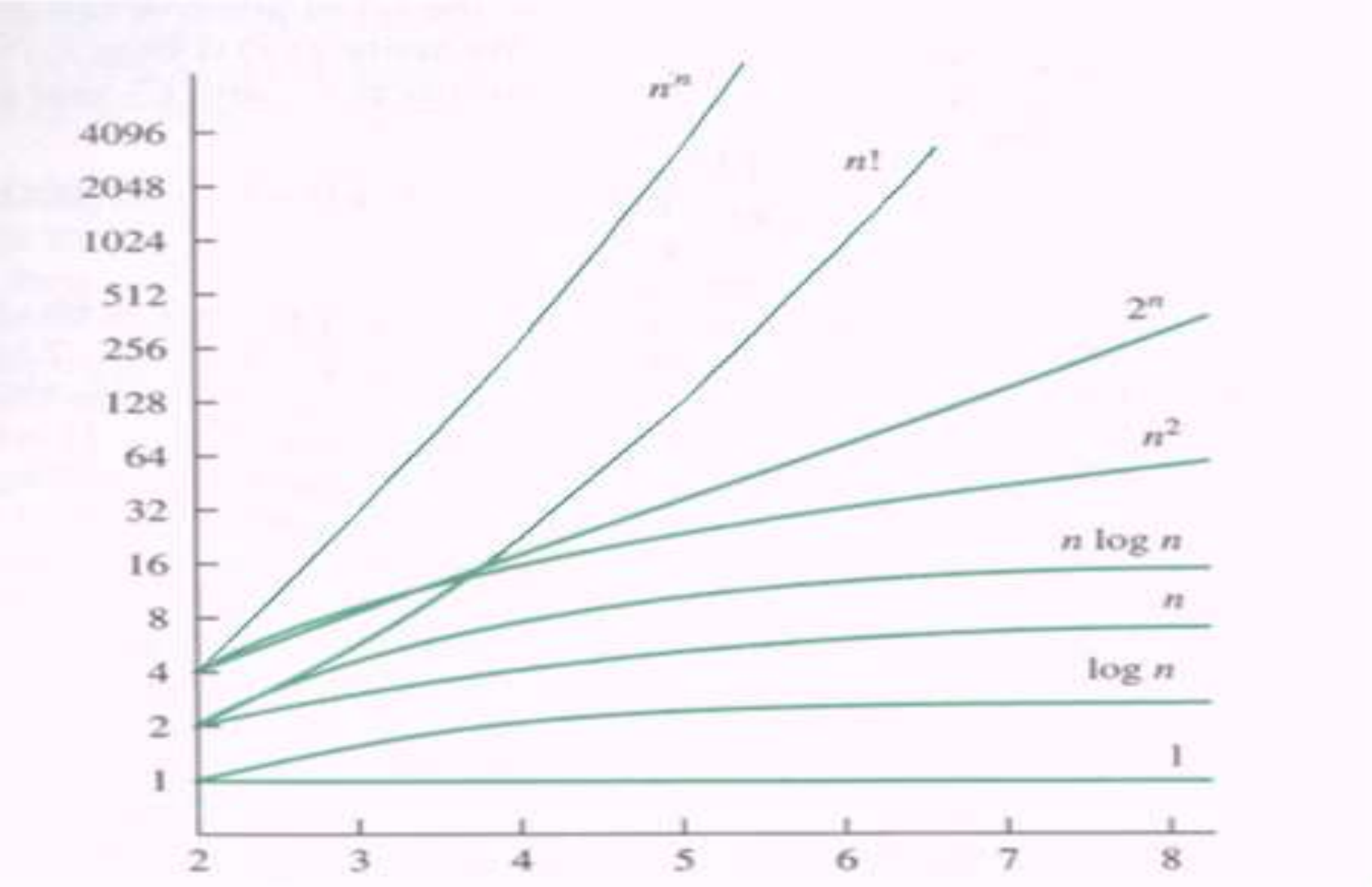
Question

- Find the Big O value for following fragment of code.

```
for i = 1 to n
        for j = 1 to i
                Print j
```

$O(n^2)$

Graphs of functions



n	$\log n$	n	$n\log n$	n^2	n^3	2^n
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,094	262,144	$1.84 * 10^{19}$
128	7	128	896	16,384	2,097,152	$3.40 * 10^{38}$
256	8	256	2,048	65,536	16,777,216	$1.15 * 10^{77}$
512	9	512	4,608	262,144	134,217,728	$1.34 * 10^{154}$
1024	10	1,024	10,240	1,048,576	1,073,741,824	$1.79 * 10^{308}$

Big O – Notation(Contd.)

- Find the Big O value for the following functions.

  (i) $T(n)= 3 +5n + 3n^2$

  (ii) $f(n)= 2^n + n^2 +8n +7$

  (iii) $T(n)= n + \log n +6$

Answers:

(i) $O(n^2)$

(ii) $O(2^n)$

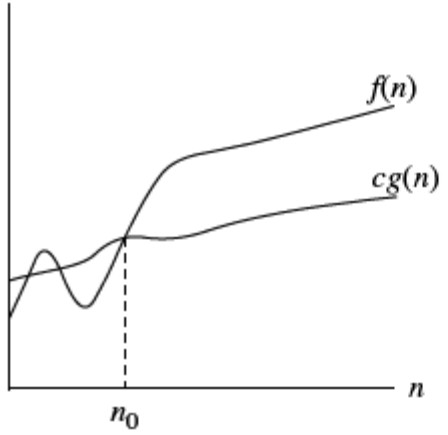
(iii) $O(n)$

Ω - Notation

- Provides the lower bound of the function.

Definition:

$\Omega(g(n))$ = { f(n) : there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_o$}



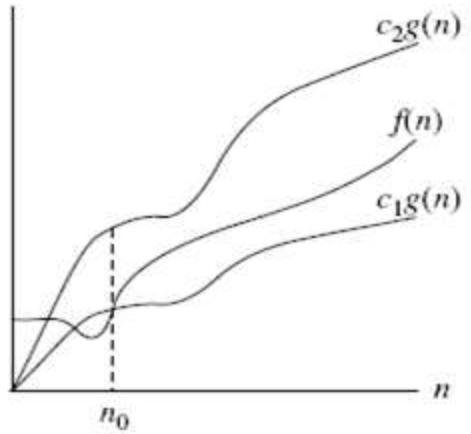
$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Θ - Notation

- This is used when the function f can be bounded both from above and below by the same function g.

Definition:

$\Theta(g(n))$ ={ f(n): there exist positive constant c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ }



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Summary

- What is an algorithm?
- Properties of an algorithm.
- Design methods.
- Pseudocode.
- Analysis(Operation count & Step count, RAM model).
- Insertion Sort.
- Asymptotic Notation

References

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, Clifford Stein Introduction to Algorithms,3rd Edition, MIT Press, 2009.

SLIIT
Discover Your Future

IT2070 – Data Structures and Algorithms

Lecture 07
Introduction to Divider and Conquer Method

Today's Lecture

- Divide and Conquer
- Applications
  - Quick Sort
  - Merge Sort
- Analysis

Divide and Conquer

Divide: Break the problem into sub problem recursively.

Conquer: Solve each sub problems.

Combine: All the solutions of sub problems are combined to get the solution of the original problem.

Applications

- Quick Sort
  - More work on divide phase.
  - Less work for others.
- Merge Sort
  - Vice versa of Quick sort.

Quick Sort (contd.)

Divide: Partition (rearrange) the array A[p..r] into two (possibly empty) sub arrays A[p..q - 1] and A[q + 1..r]

- Each element of A[p..q - 1] is less than or equal to A[q]
- Each element of A[q + 1..r]is greater than or equal to A[q].
- Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays A[p..q -1] and A[q +1..r] by recursive calls to quicksort.

Combine: Since the sub arrays are sorted in place, no work is needed to combine them: the entire array A[p..r] is now sorted.

Quick Sort procedure

Input: Unsorted Array (A,p,r)

Output: Sorted sub array A(1..r)

```
QUICKSORT (A,p,r)
1 if p < r
2     q = PARTITION(A, p, r)
3     QUICKSORT (A, p, q-1)
4     QUICKSORT (A, q+1, r)
```

To sort an entire array A, the initial call is

QUICKSORT(A, 1, A.length).

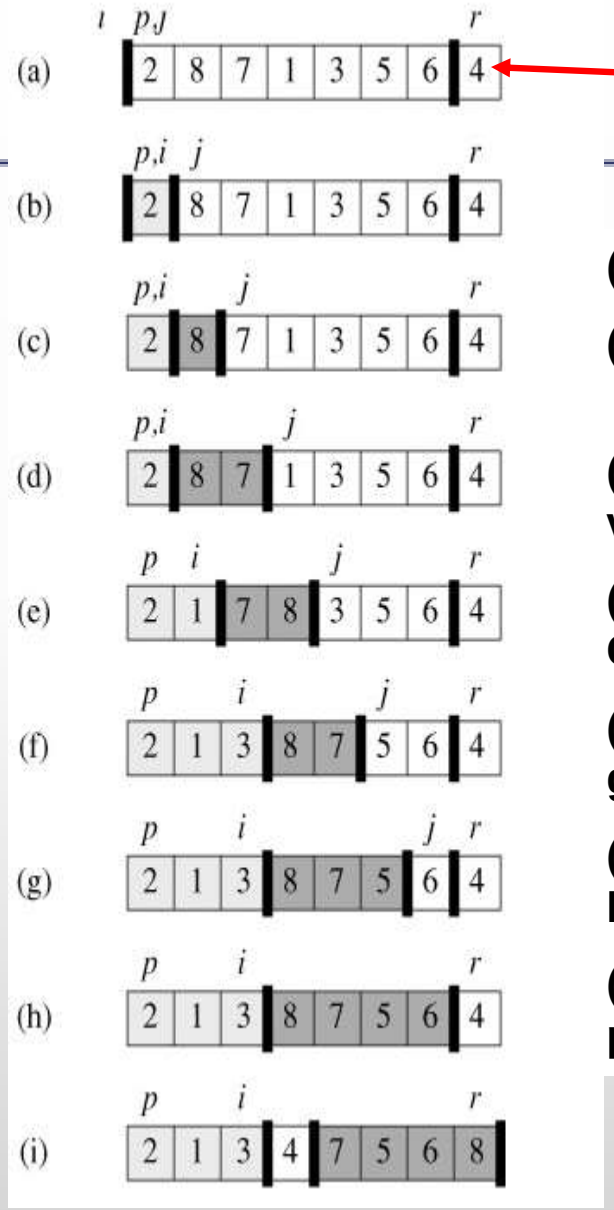
Partition Algorithm

PARTITION(*A*, *p*, *r*)

```
1    x = A[r]
2    i = p - 1
3    for j = p to r - 1
4         if A[j] ≤ x
5              i = i + 1
6              exchange A[i] with A[j]
7     exchange A[i + 1] with A[r]
8     return i + 1
```

The key to the algorithm is the PARTITION procedure, which rearranges the sub array A [p..r] in place.

Operation of PARTITION on an 8-element array.



Element x = A[r] is the pivot element

(a) The initial array

(b) The value 2 is "swapped with itself" and put in the partition of smaller values.

(c)-(d) The values 8 and 7 are added to the partition of larger values.

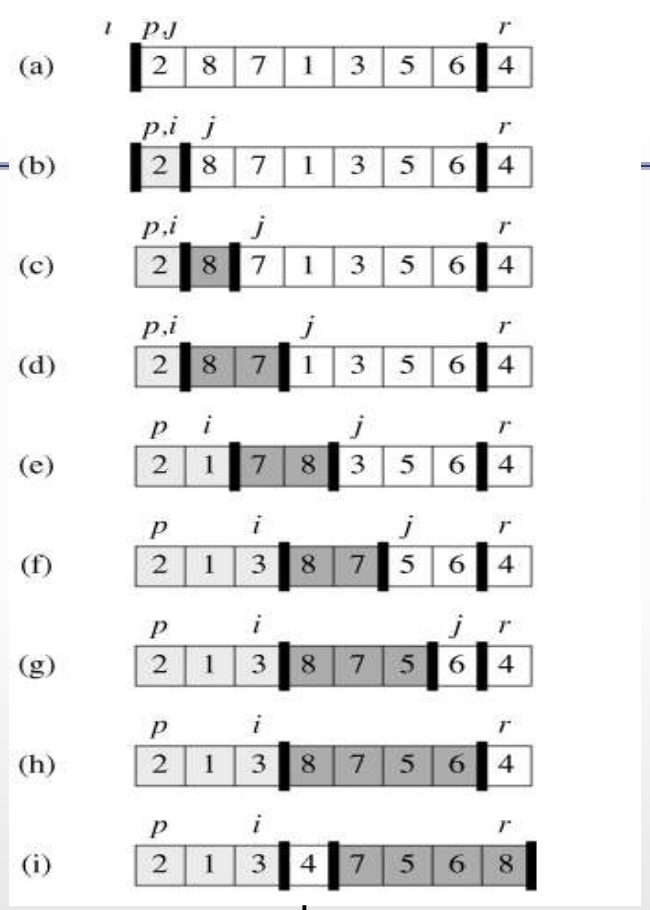
(e) The values 1 and 8 are swapped, and the smaller partition Grows.

(f) The values 3 and 7 are swapped, and the smaller partition grows.

(g)-(h) The larger partition grows to include 5 and 6 and the loop terminates.

(i) Pivot element is swapped so that it lies between the two partitions.

Operation of PARTITION on an 8-element array.



first partition with values <= pivot

second partition with values > pivot

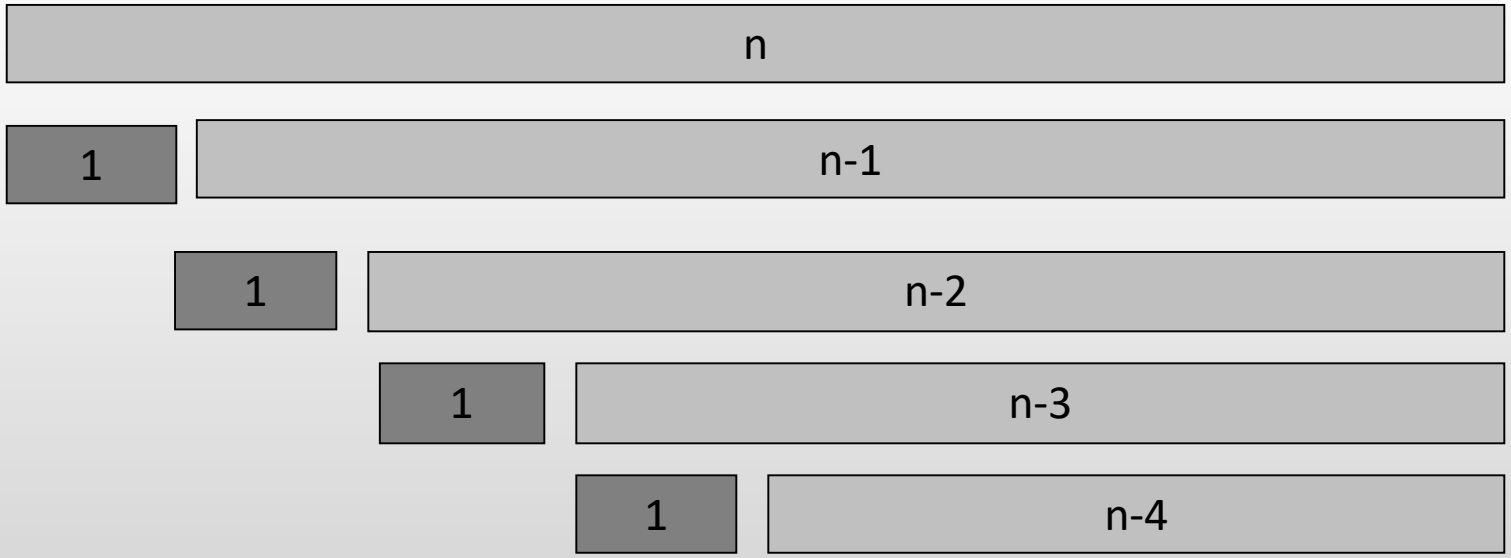
white element - pivot.

Analysis of Quick sort

The running time of quick sort depends on the partitioning of the sub arrays:

(a) Worst case partitioning (Unbalanced partitioning)

- Worst case occurs when the sub arrays are completely unbalanced. i.e. 0 elements in one sub array and $n - 1$ elements in the other sub array



Analysis of Quick sort

Worst case partitioning (Repeated Substituted method)

- Partitioning → $\Theta(n)$
- Recursive call on an array of size 0 → $T(0) = \Theta(1)$
- Recursive call on an array of size (n-1) → $T(n-1)$

Therefore **Recurrence** Equation is

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

$$= \underbrace{T(n-1)} + \Theta(n)$$

$$= T(n-2) + \Theta(n-1) + \Theta(n)$$

$$\ldots\ldots\ldots\ldots\ldots$$

$$= T(0) + \Theta(1) + \Theta(2) + \ldots + \Theta(n-1) + \Theta(n)$$

$$= \sum_{k=1}^{n} (\Theta(k)) \qquad = \Theta \sum_{k=1}^{n} k \quad = \quad \Theta(n^2)$$

Worst case Running Time is $\Theta(n^2)$

Analysis of Quick sort

(b) Best case partitioning

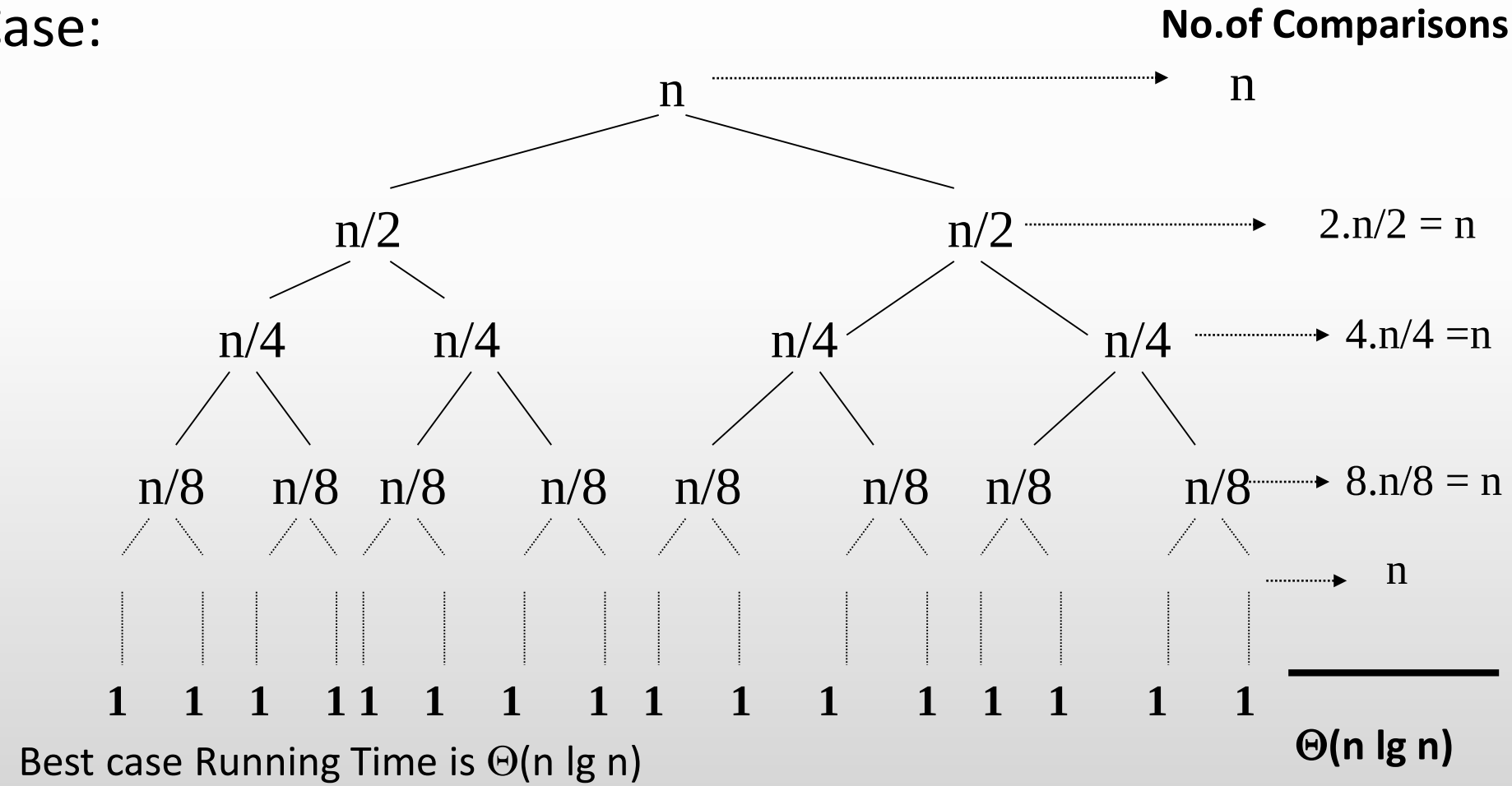
Best case occurs when PARTITION produces two sub arrays , one is of size (n-1)/2 and the other is of size (n-1)/2. In this case, quicksort runs much faster.

Recurrence equation is

$$T(n) = 2T(n/2) + \Theta(n)$$

Analysis of Quick Sort (with recursion tree)

Best Case:



Best case Running Time is Θ(n lg n)

Merge sort

Merge Sort is a sorting algorithm based on divide and conquer.

Its worst-case running time has a lower order of growth than insertion sort.

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide:** Divide the *n*-element sequence to be sorted into two subsequences of *n*/2 elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

Merge sort

Divide by splitting into two subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$, where q is the halfway point of $A[p\,..\,r]$.

Conquer by recursively sorting the two subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$.

Combine by merging the two sorted subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ to produce a single sorted subarray $A[p\,..\,r]$.

To accomplish this step, we'll define a procedure MERGE*(A, p, q, r)*.

Merge sort procedure

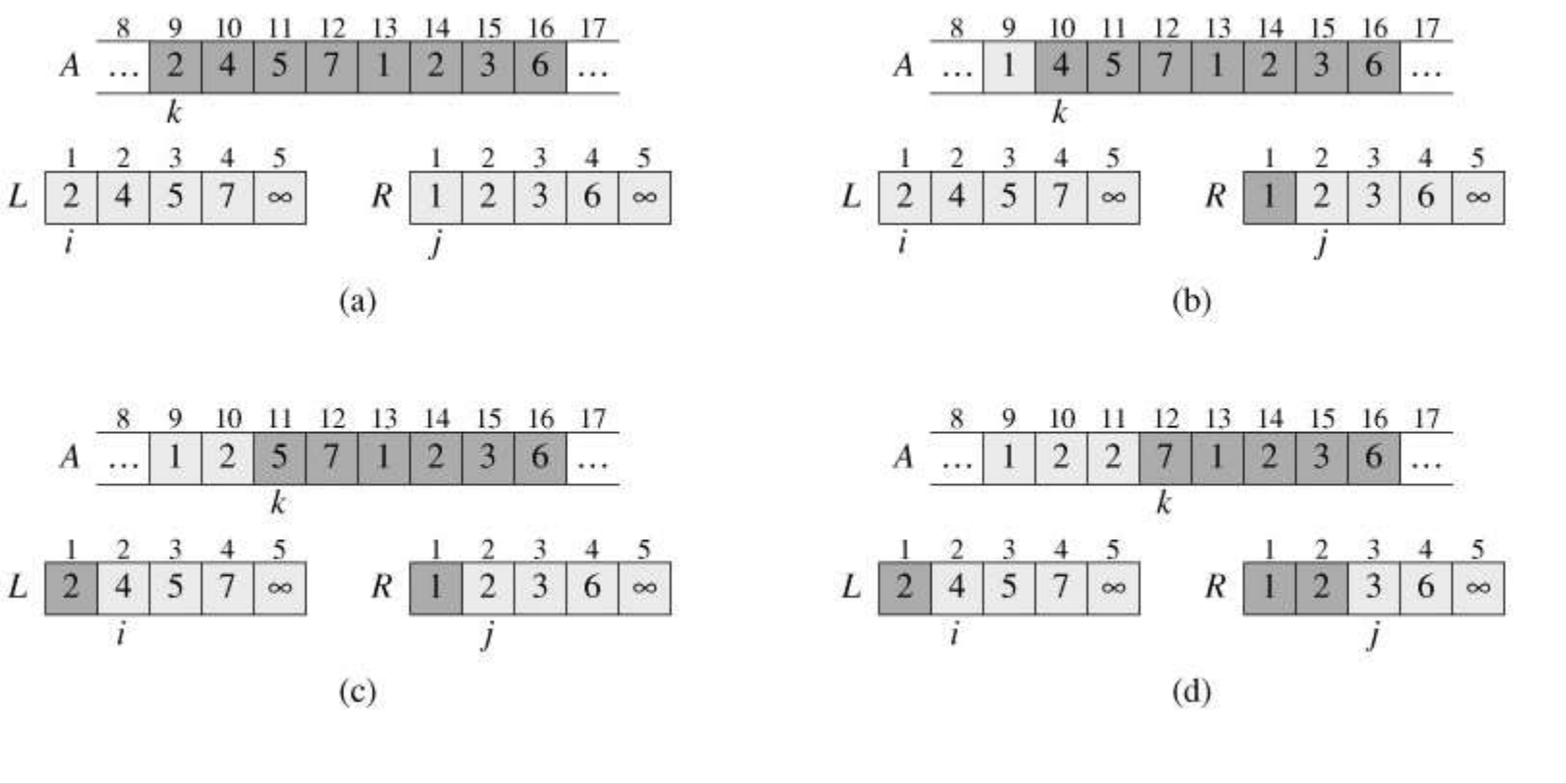
Input : A an array in the range 1 to n.
Output : Sorted array A.

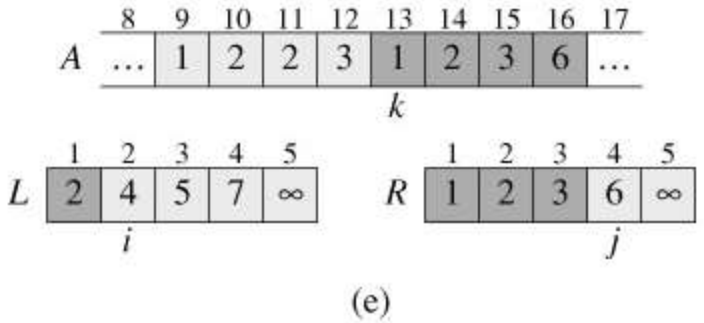
```
MERGESORT (A, p, r)
1.  if p < r
2.        q = ⌊(p+r )/2⌋
3.     MERGESORT (A, p, q)
4.     MERGESORT (A, q+1, r)
5.        MERGE (A, p, q, r)
```

Merge procedure

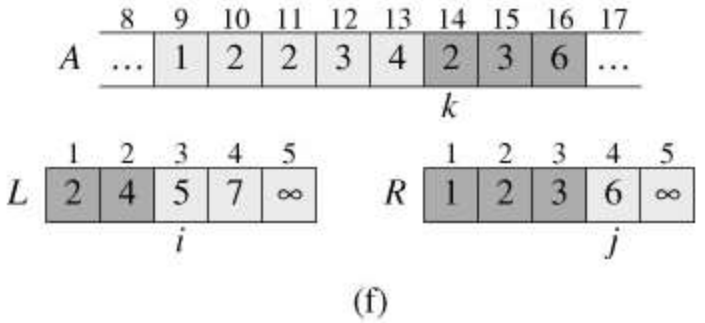
```
MERGE(A, p, q, r)
1     n1 = q - p + 1
2     n2 = r - q
3     create arrays L[1.. n1 + 1] and R[1.. n2 + 1]
4     for i = 1 to n1
5           L[i] = A[p + i - 1]
6      for j = 1 to n2
7           R[j] = A[q + j]
8     L[n1 + 1] = ∞
9     R[n2 + 1] = ∞
10    i = 1
11    j = 1
12    for k = p to r
13          if L[i] ≤ R[j]
14                A[k] = L[i]
15                i = i + 1
16          else  A[k] = R[j]
17                j = j + 1
```

Illustration when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$

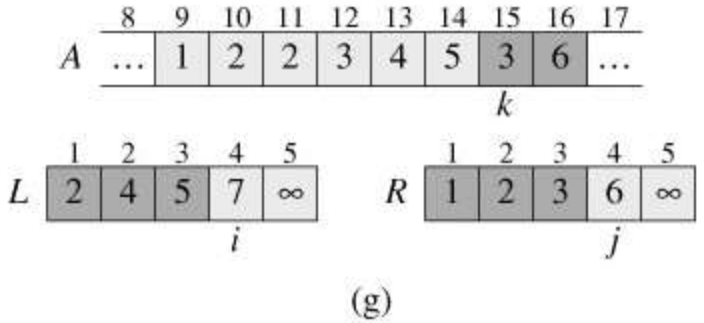



8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

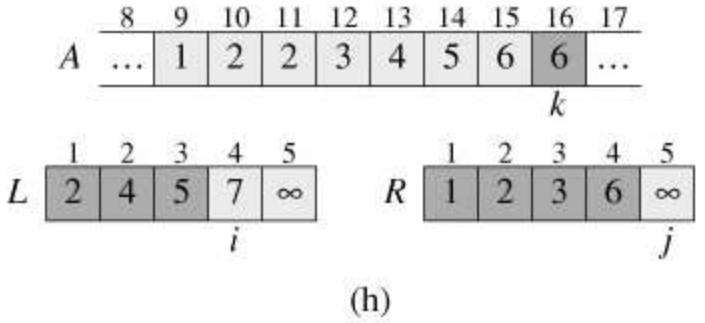
(e)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

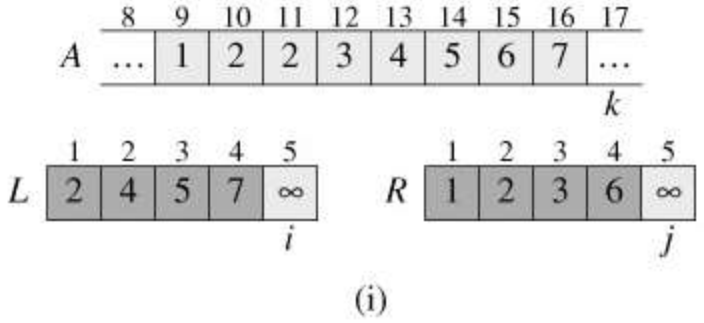
(f)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(g)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(h)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(i)

Analysis of Merge Sort

- To find the middle of the sub array will take $\Theta(1)$.
- To recursively solve each sub problem will take 2T(n/2).
- To combine sub arrays will take $\Theta(n)$.

Therefore **$T(n)=2T(n/2)+\Theta(n)+\Theta(1)$**

We can ignore $\Theta(1)$ term.

$T(n)=2T(n/2)+\Theta(n)$

Analysis of Merge Sort

$T(n) = 2T(n/2) + cn$

$2T(n/2) = 4T(n/4) + 2cn/2$

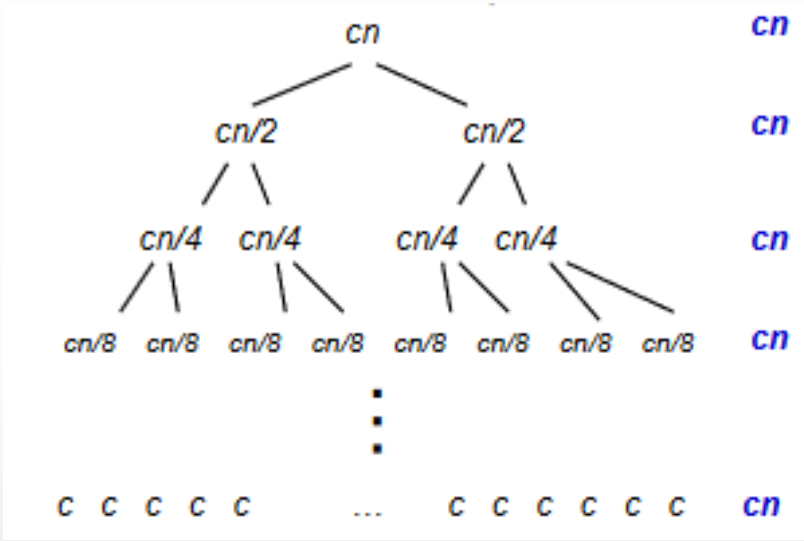
$4T(n/4) = 8T(n/8) + 4cn/4$

-

-

-

$T(2) = nT(1) + (n/2)\, c*2$



---

add and cancel:

$$T(n) = nT(1) + cn + cn + \ldots + cn$$

$$= nT(1) + cn * \log_2 n = \Theta(n \log n)$$

Summary.

- Divide and conquer method.
- Quicksort algorithm.
- Quicksort analysis.
- Mergesort algorithm.
- Mergesort analysis.

IT2070 – Data Structures and Algorithms

Lecture 08

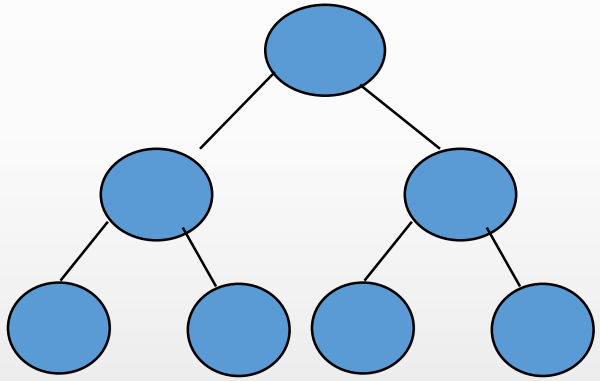
Heap Sort Algorithms

Contents for Today

- **Tree**
- **Binary Tree**
- **Complete Binary Tree**
- **Heaps**
- **Heap Algorithms**
  - Maintaining Heap Property
  - Building Heaps
  - HeapSort Algorithms

Height of a Full Binary Tree

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

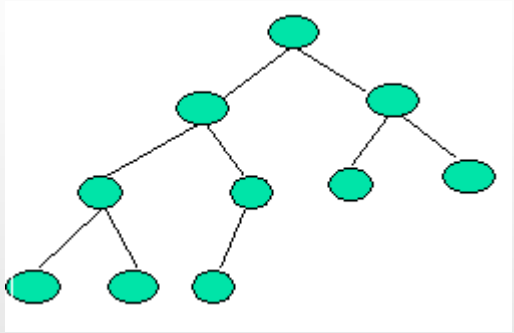


Height, ***h*** = 2, nodes = $2^{2+1}-1 = 7$

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: $n = 10$
- Height = $\lfloor \log_2(n) \rfloor = \lfloor \log_2(10) \rfloor = 3$

Heaps

Heap is an array object that can be viewed as a complete binary tree. There are two kinds of heaps

max heaps and **min heaps**

In both case, values in the nodes satisfy **Heap Property** which depend on the kind of heap

max-heap → max-heap property:

The value of each node is greater than or equal to those of its children.

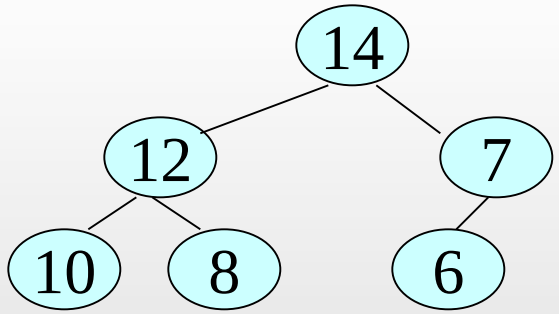
min-heap → min-heap property:

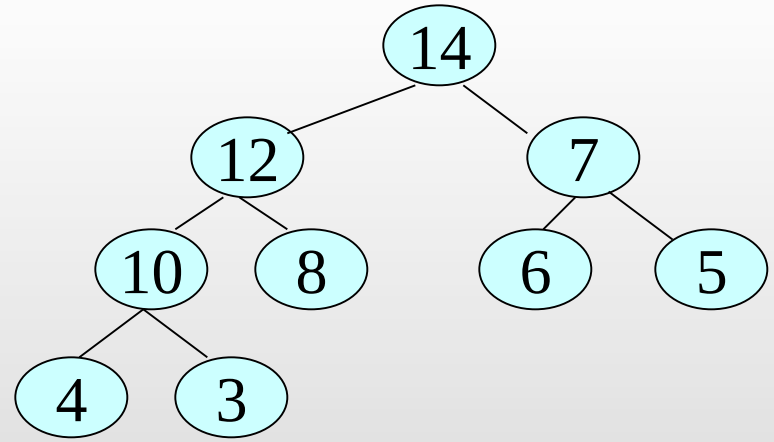
The value of each node is less than or equal to those of its children.

Max-heaps are used in heapsort algorithm

Heaps

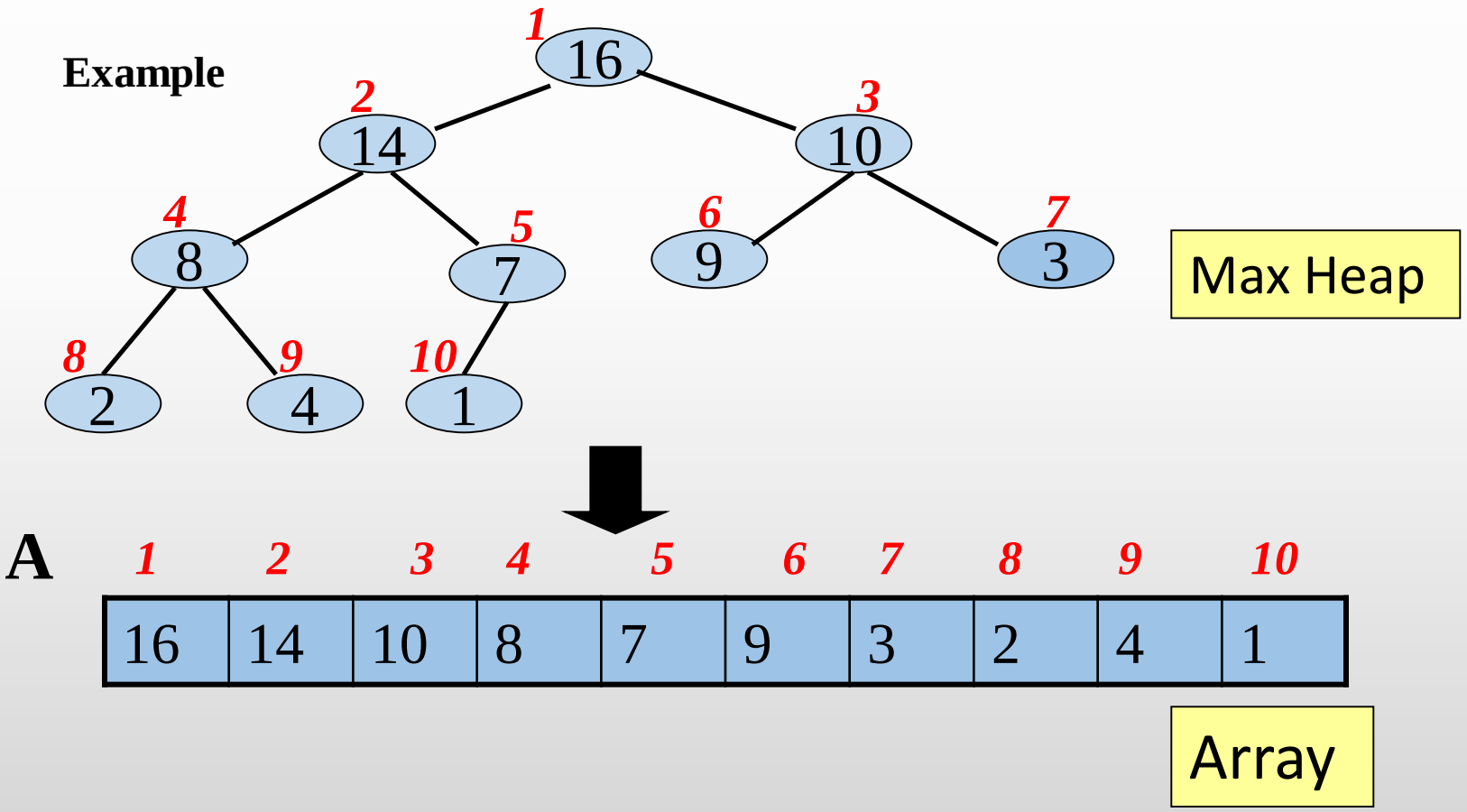
Complete Binary Tree with the **max-heap property - examples**





Heaps (contd.)

- A heap can be represented in a one-dimensional array



Heaps (contd.)

After representing a heap using an array **A**

- Root of the tree is **A[1]**

Given node with index *i*,

- ***PARENT(i)*** is the index of parent of ***i***;

$$PARENT(i) = \lfloor i/2 \rfloor$$

- ***LEFT_CHILD(i)*** is the index of left child of ***i*** ;

$$LEFT\_CHILD(i) = 2 \times i$$

- ***RIGHT_CHILD(i)*** is the index of right child of ***i***;

$$RIGHT\_CHILD(i) = 2 \times i + 1$$

Heap Algorithms

- **MAX_HEAPIFY**:

  To maintain max-heap property

  $$A[PARENT(i)] \geq A[i]$$

- **BUILD_MAX_HEAP**

  To build max-heap from an unsorted input array

- **HEAPSORT**

  Sorts an array in place.

MAX_HEAPIFY

- The MAX_HEAPIFY algorithm checks the heap elements for violation of the heap property and restores max-heap property;

$$A[PARENT(i)] \geq A[i]$$

- **Input**: An array A and index *i* to the array. *i* =1 if we want to heapify the whole tree. Subtrees rooted at *LEFT_CHILD(i)* and *RIGHT_CHILD(i)* are heaps

- **Output**: The elements of array A forming subtree rooted at *i* satisfy the heap property.

Maintaining the Heap Property

MAX_HEAPIFY (*A*,*i*)

1. *l* = LEFT(*i*);
2. *r* = RIGHT(*i*);
3. if *l* ≤ A.heap_size and A[*l*] > A[*i*]
4. largest = *l*;
5. else largest = *i*;
6. if *r* ≤ A.heap_size and A[*r*] > A[*largest*]
7. largest = *r*;
8. if *largest* ≠ *i*
9. exchange A[*i*] with A[*largest*]
10. **MAX_HEAPIFY** (A,*largest*)

SLIIT
Discover Your Future

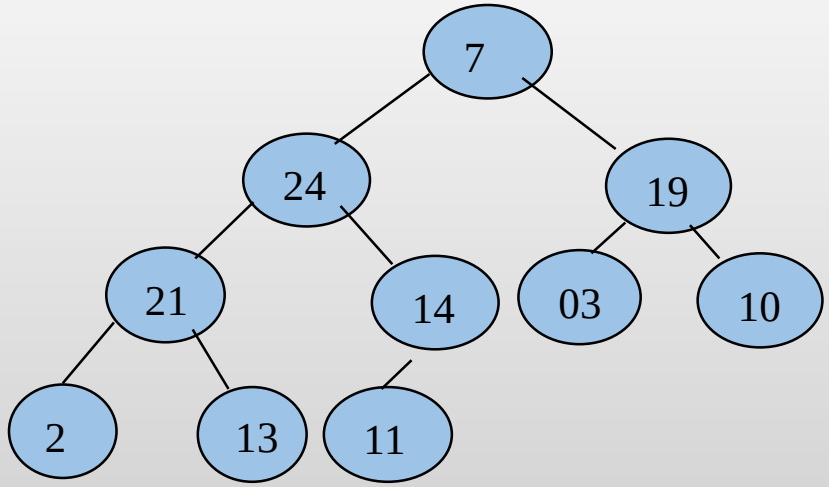
Example

You are given the following array

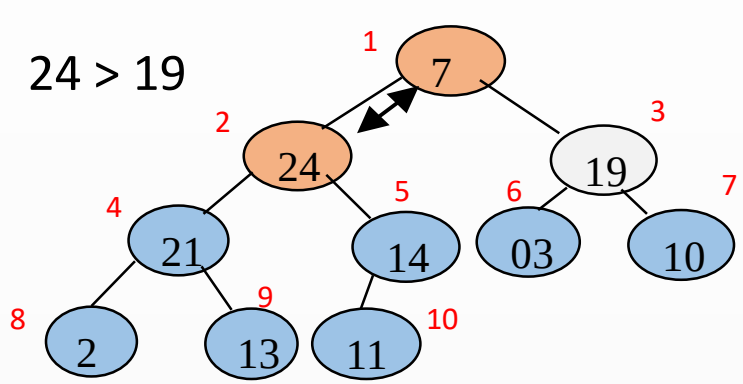
A	1	2	3	4	5	6	7	8	9	10
	7	24	19	21	14	03	10	2	13	11

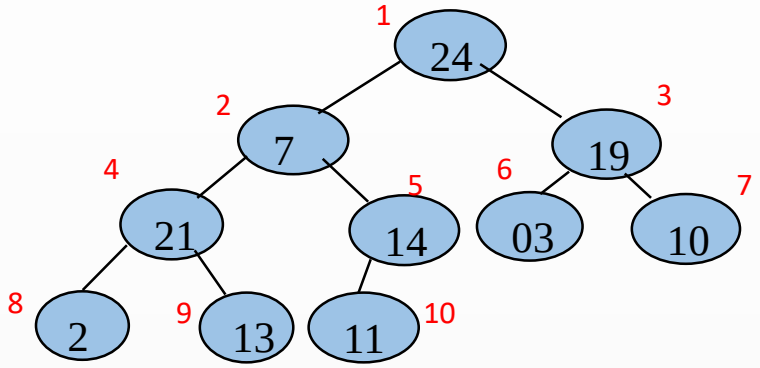
Now we are going to maintain the max-heap property

Drawing a heap would make our work easy

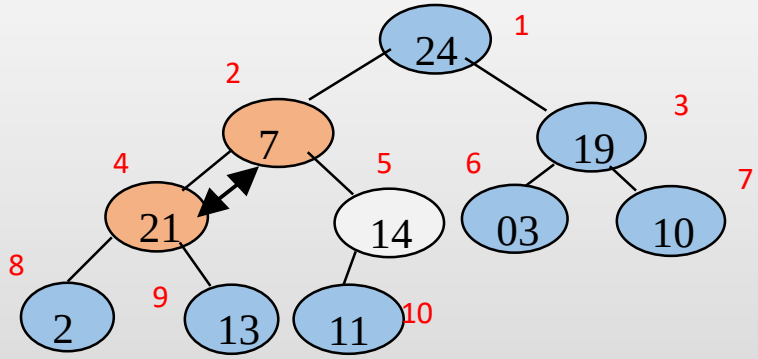


MAX_HEAPIFY (*A*,1)

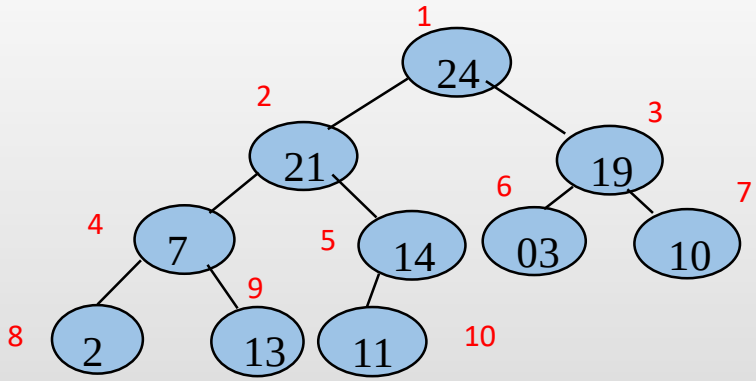




MAX_HEAPIFY (*A*,2)



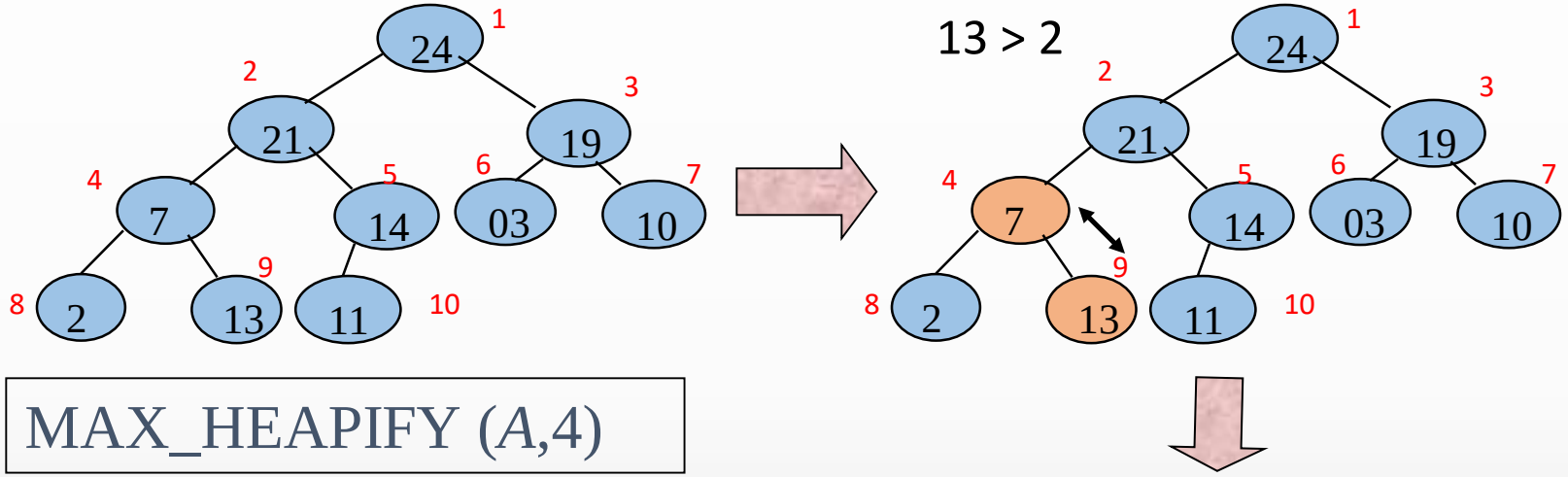
21 > 14



MAX_HEAPIFY (*A*,4)

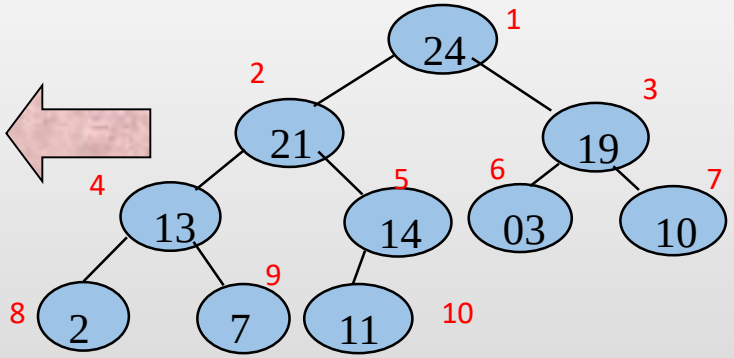
To be contd.

MAX_HEAPIFY (*A*,1) (contd.)



MAX_HEAPIFY (*A*,4)

Important point Although we represent this process using a heap actually all the task in done on the input array

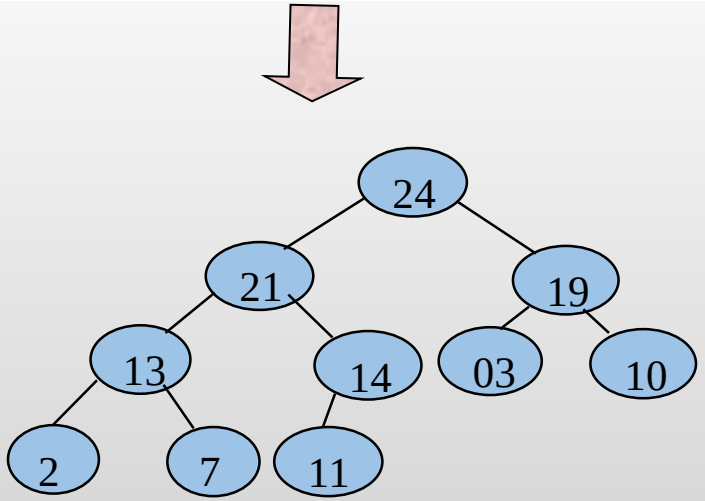


Resulting Heap

SLIIT
Discover Your Future

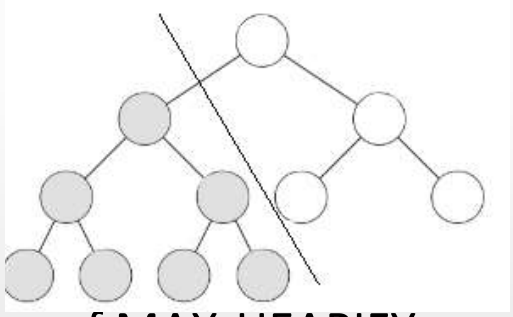
Array view of MAX_HEAPIFY Algorithm

7	**24**	**19**	**21**	**14**	**03**	**10**	**02**	**13**	**11**
24	**7**	**19**	**21**	**14**	**03**	**10**	**02**	**13**	**11**
24	**21**	**19**	**07**	**14**	**03**	**10**	**02**	**13**	**11**
24	**21**	**19**	**13**	**14**	**03**	**10**	**02**	**07**	**11**



Analysis of Heapify Algorithm.

- The running time of MAX-HEAPIFY on a subtree of size *n* rooted at given node *i* is the Θ(1) time plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node *i*.
- The children's subtrees -the worst case occurs when the last row of the tree is exactly half full



- Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height *h* as $O(h)$.

The solution to this recurrence is

$$T(n) = O(\lg n)$$

BUILD_HEAP

Input : An array A of size *n* = A.length , A.heap_size
Output : A heap of size *n*

```
BUILD_MAX_HEAP (A)

1.      A.heap_size =A.length
2.      for i = ⌊A.length/2⌋ downto 1
3.              MAX_HEAPIFY(A,i)
```

Exercise: We are given the following unordered array to build the heap.

A

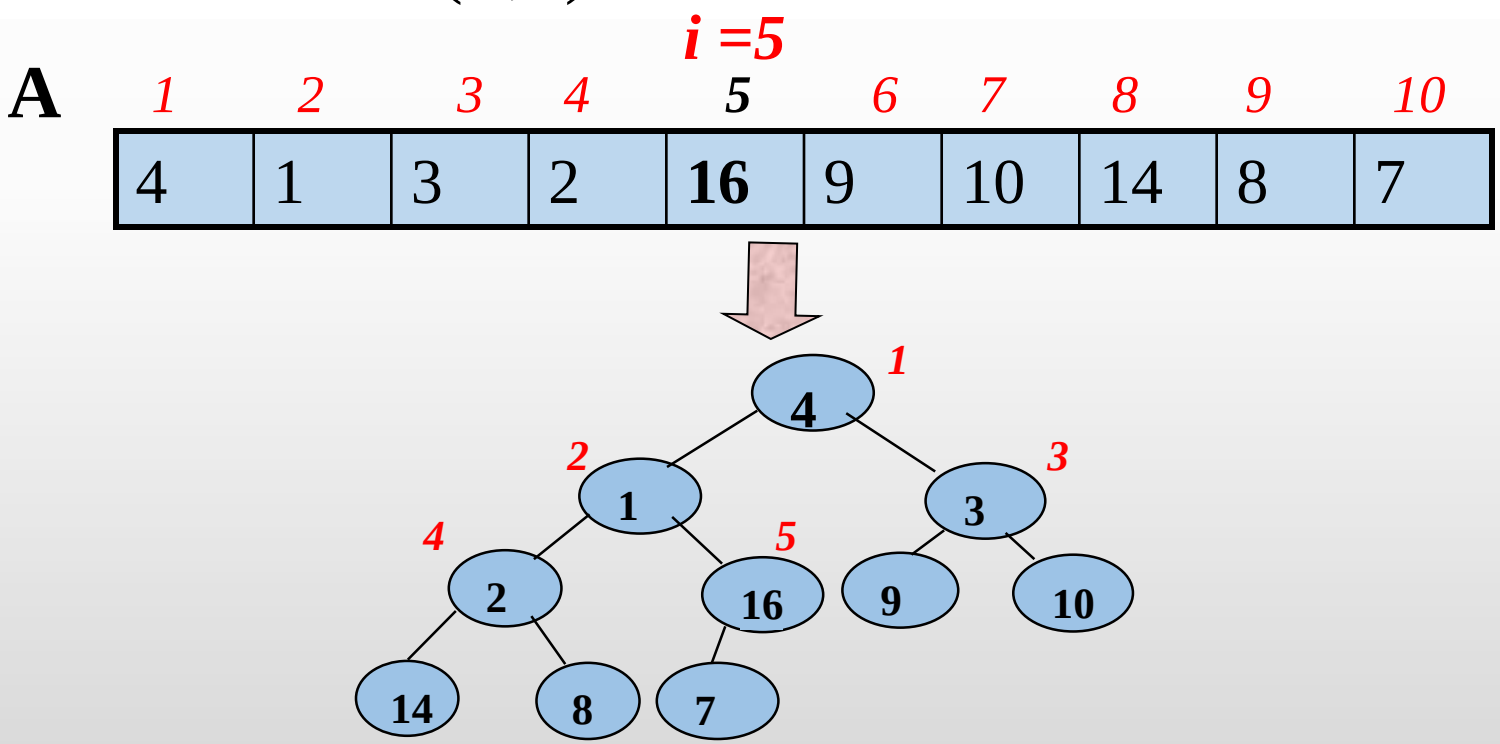
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Solution

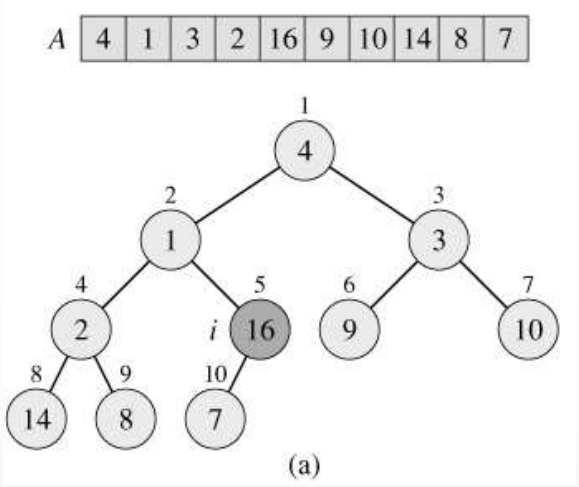
Step1

$i = \lfloor length[A]/2 \rfloor = \lfloor 10/2 \rfloor = 5$

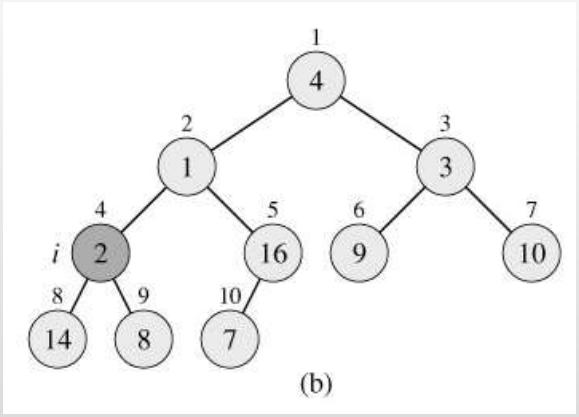
MAX_HEAPIFY(A,5)



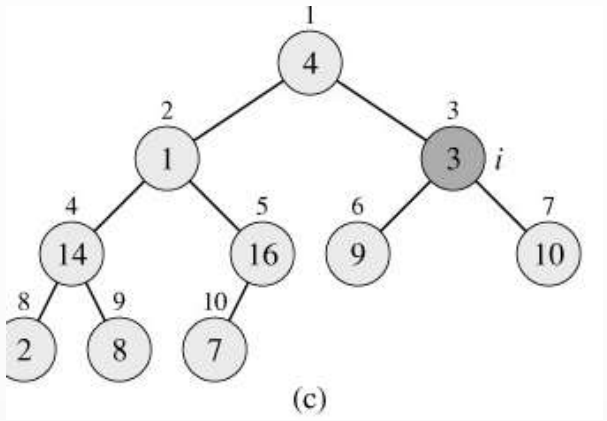
Solution (Contd.)



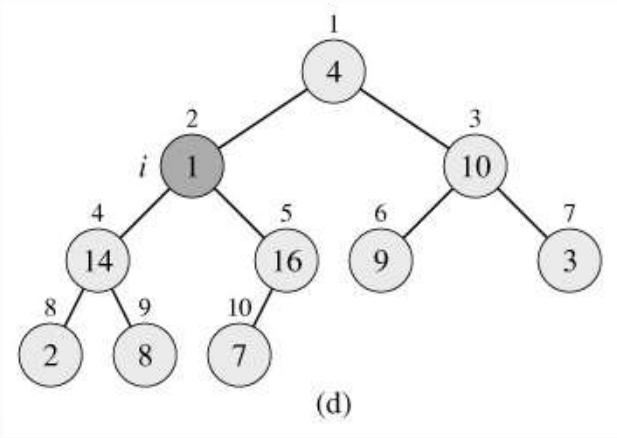
loop index *i* refers to node 5



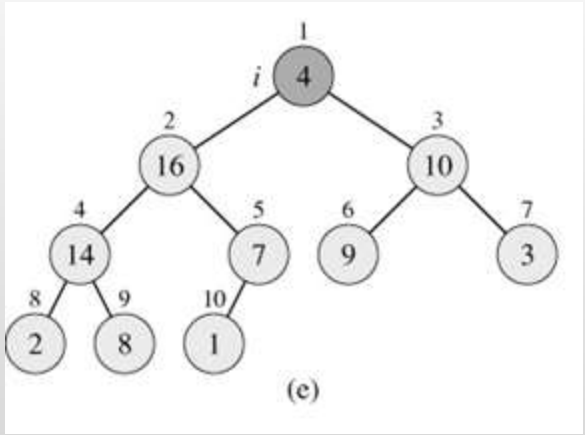
loop index *i* for the next iteration refers to node 4



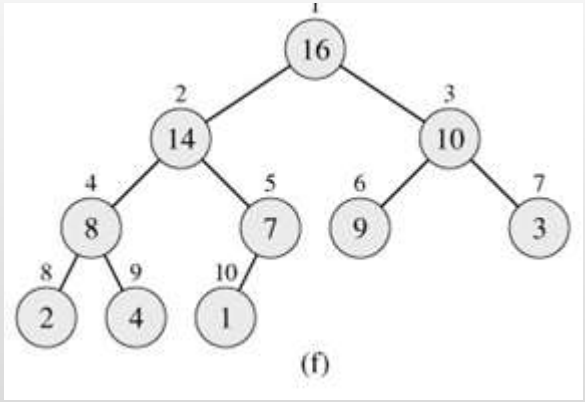
loop index *i* refers to node 3



loop index *i* refers to node 2



loop index *i* refers to node 1



max-heap after BUILD-MAX-HEAP finishes.

Analysis of Build Max Heap Algorithm.

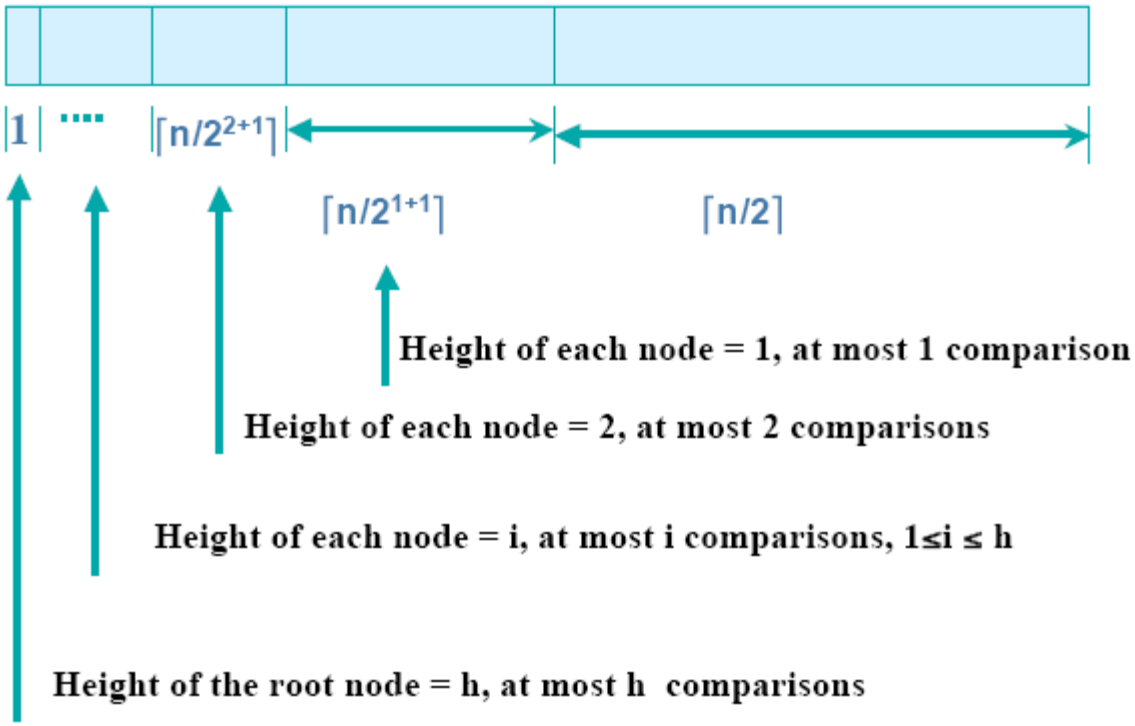
Time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.

- *n*-element heap has height $\lfloor \lg n \rfloor$

  and

- at most $\lceil n/2^{h+1} \rceil$ nodes of any height *h*.

Analysis of Build Max Heap Algorithm.



Complexity analysis of Build-Heap (1)

- For each height $0 < h \leq \lg n$, the number of nodes in the tree is at most $n/2^{h+1}$
- For each node, the amount of work is proportional to its height h, $O(h) \rightarrow n/2^{h+1} .O(h)$
- Summing over all heights, we obtain:

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil .O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil \right)$$

Complexity analysis of Build-Heap (2)

- We use the fact that $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad for\ |x|<1$

$$\sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil = \frac{1/2}{(1-1/2)^2} = 2$$

- Therefore:

$$T(n) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil \right) = O\left(n \sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil \right) = O(n)$$

- Building a heap takes only linear time and space!

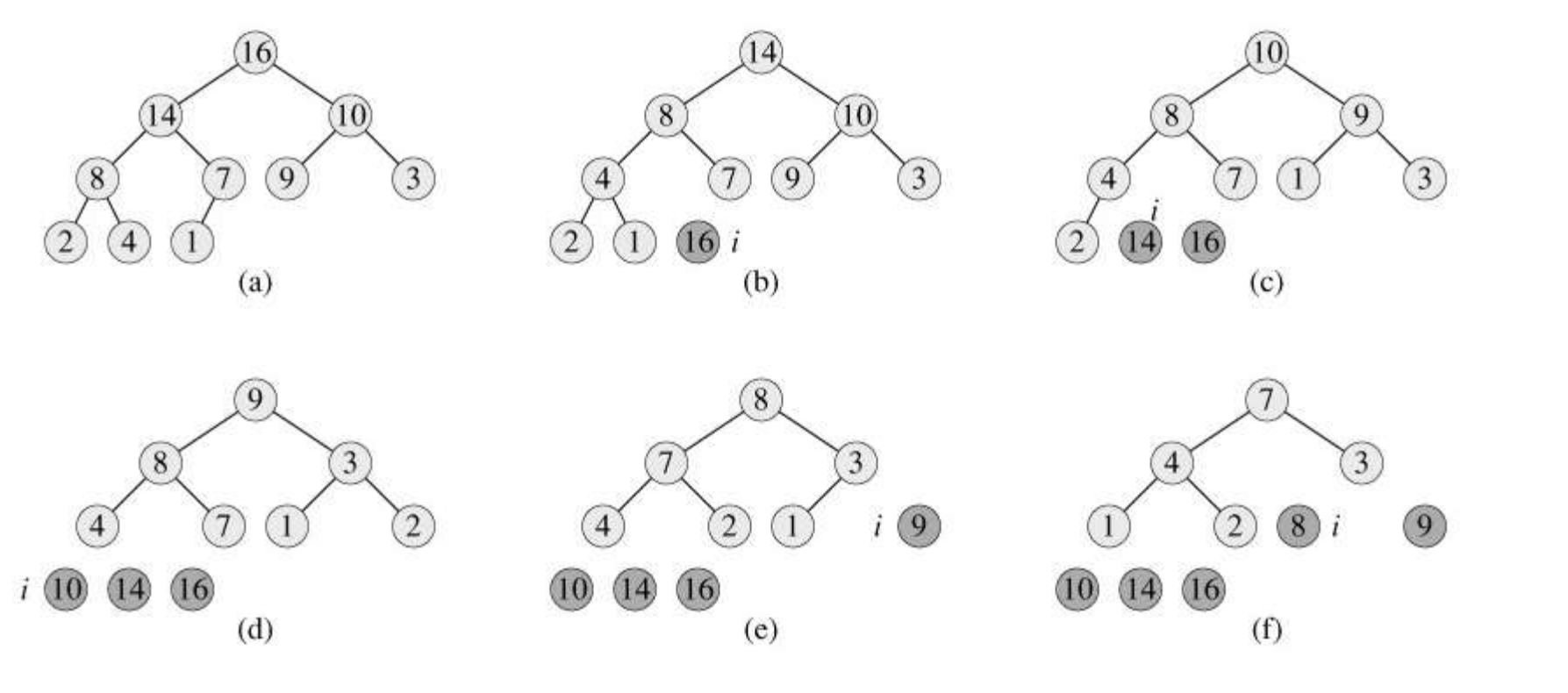
The HEAPSORT Algorithm

Input : Array A[1…n], n = A.length
Output : Sorted array A[1…n]

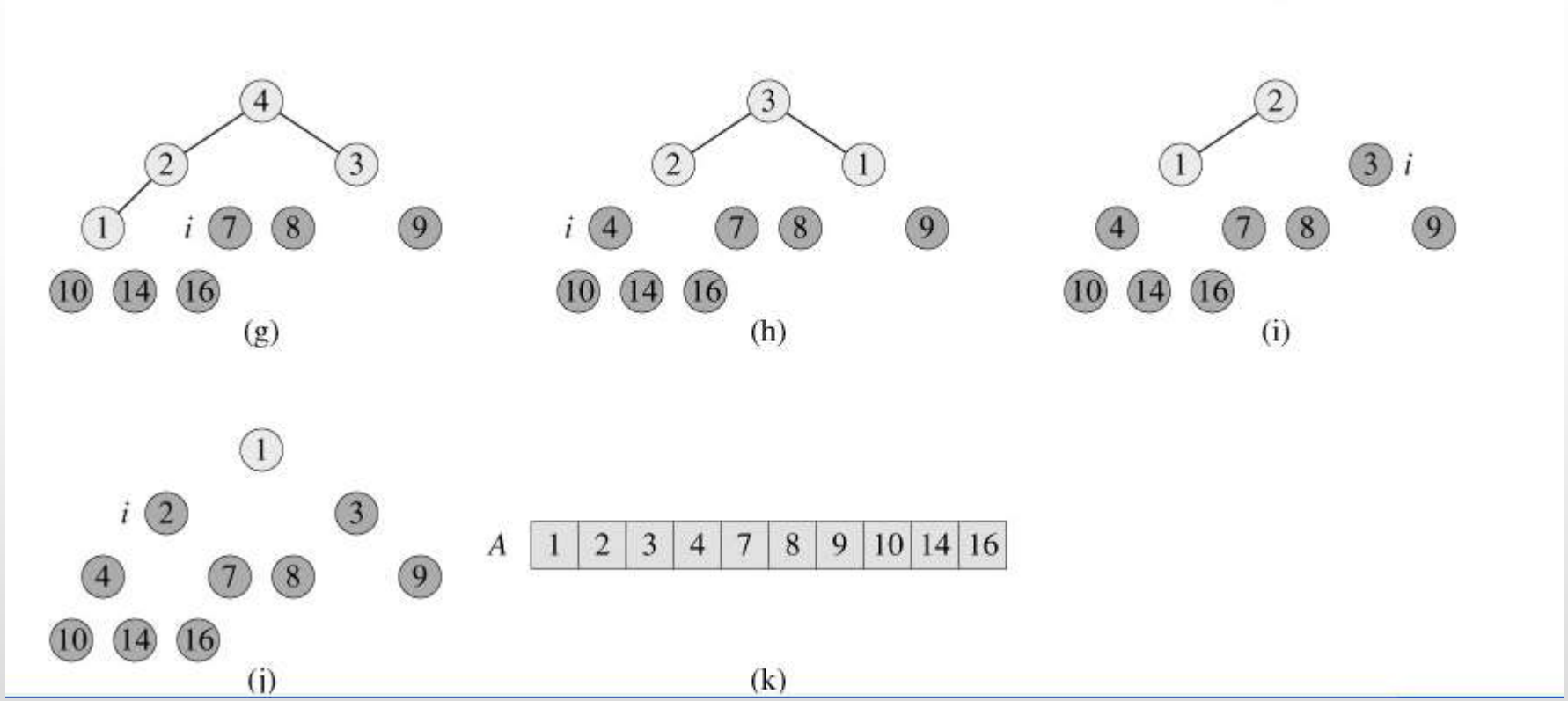
```
HEAPSORT(A)
1.      BUILD_MAX_HEAP[A]
2.      for  i = A.length down to 2
3.              exchange A[1] with A[i]
4.              A.heap_size = A.heap_size - 1;
5.              MAX_HEAPIFY(A, 1)
```

SLIIT
Discover Your Future

The operation of HEAPSORT.



The operation of HEAPSORT.



Heapsort Complexity

Running Time:

Step1 : BUILD_MAX_HEAP takes O(n)

Step 2 to 5 : MAX_HEAPIFY takes O(log n) and there are (n -1) calls

Running Time is O(n log n)

Priority Queues

- Heap data structure itself has many uses.
- One of the most popular applications of a heap: its use as an efficient **priority queue**.
- As with heaps, there are two kinds of priority queues:
  - max-priority queues
  - min-priority queues
- We will focus here on how to implement **max-priority queues**, which are in turn based on max-heaps

Priority queues.

- ***priority queue*** is a data structure for maintaining a set *S* of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations.

- INSERT(*S*, *x*) inserts the element *x* into the set *S*. This operation could be written as $S = S \cup \{x\}$.

- EXTRACT-MAX(*S*) removes and returns the element of *S* with the largest key.

Priority queues.

- One application of max-priority queues is to schedule jobs on a shared computer.

The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

HEAP_EXTRACT_MAX

HEAP_EXTRACT_MAX(A[1 .. n])

This will remove the maximum element from heap and return it

Input : heap(A)

Output : Maximum element or root, heap(A[1..n-1])

1. if A.heap_size >= 1
2. max = A[1]
3. A[1] = A[A.heap_size]
4. A.heap_size = A.heap_size -1
5. MAX_HEAPIFY(A,1)
6. return max

Running time : O(log n)

HEAP_INSERT

HEAP_INSERT(A, key)

This will add a new element to the heap

Input : heap(A[1..n]), key - the new element

Output : heap(A[1..n+1]), with k in the heap

1. A.heap_size = A.heap_size + 1
2. i = A.heap_size // assume A[i] = - ∞
3. while i > 1 and A[PARENT(i)] < key
4. A[i] = A[PARENT(i)]
5. i = PARENT(i)
6. A[i] = key

Running time : O(lg n)

Summary

- Complete binary Tree
- Heap property
- Heap
- Maintaining heap Property(HEAPIFY)
- Building Heaps
- HeapSort Algorithm
- Priority queues.
- Heap Extract Max.
- Heap Insert.

SLIIT
Discover Your Future

IT2070 – Data Structures and Algorithms

Lecture 09

String Matching Algorithms

Contents

- String Matching
- The naïve string matching algorithm
- The Rabin-Karp Algorithm
- Finite automata

String Matching

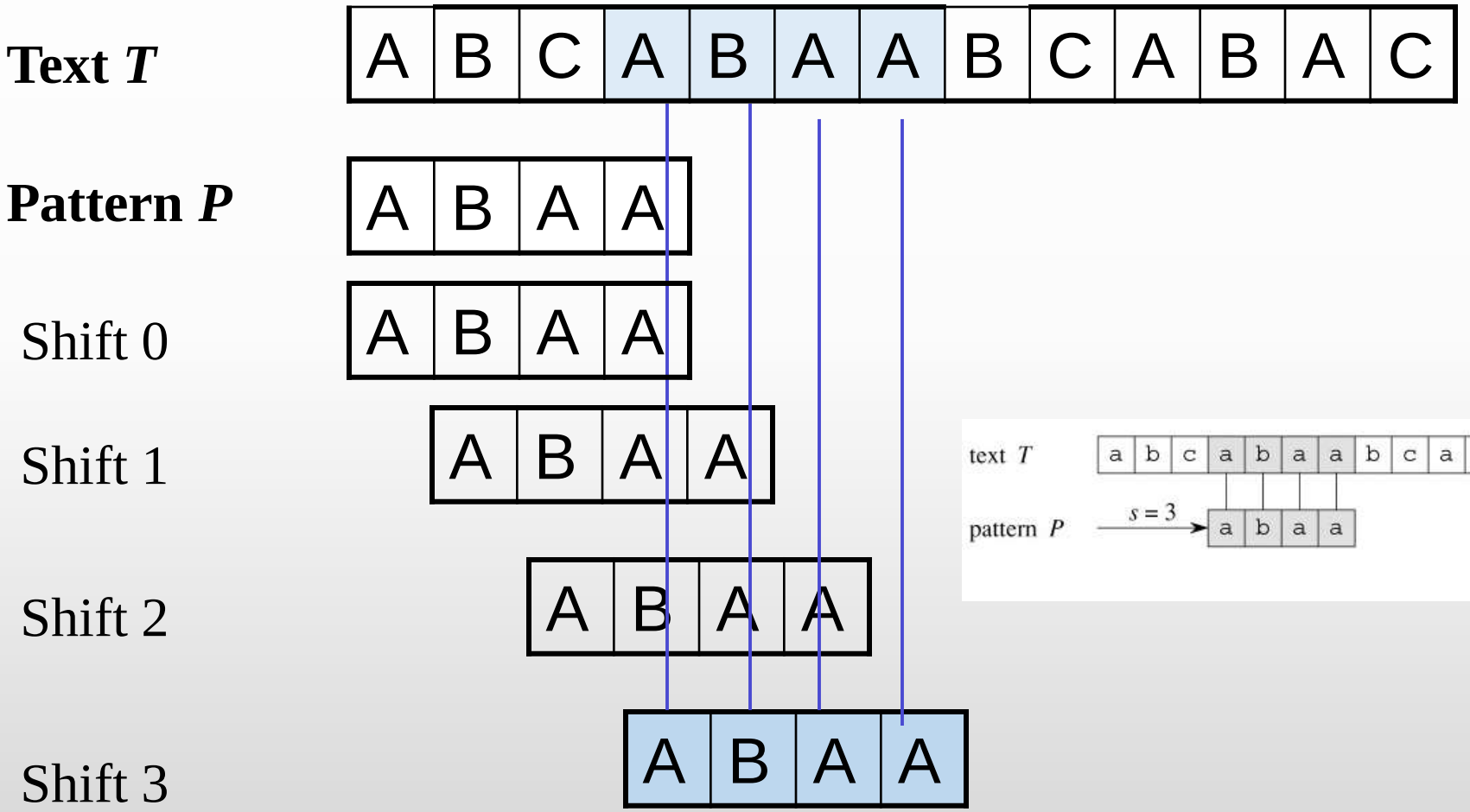
- The problem of finding occurrence(s) of a pattern string within another string or body of text.
- There are many different algorithms for efficient searching.
- String matching is a very important subject in the wider domain of text processing.

Applications

- In string matching problems, it is required to find the occurrences of a pattern in a text.

- **Applications**
  - Text processing
  - Text-editing e.g.*Find and Change* in word
  - Computer security (virus detection, password checking)
  - DNA sequence analysis.
  - Data communications (header analysis)

Example


Text T
A B C A B A A B C A B A C
Pattern P
A B A A
Shift 0
A B A A
Shift 1
A B A A
Shift 2
A B A A
Shift 3
A B A A

text T
a b c a b a a b c a b a c
s = 3
pattern P
a b a a

String Matching problem

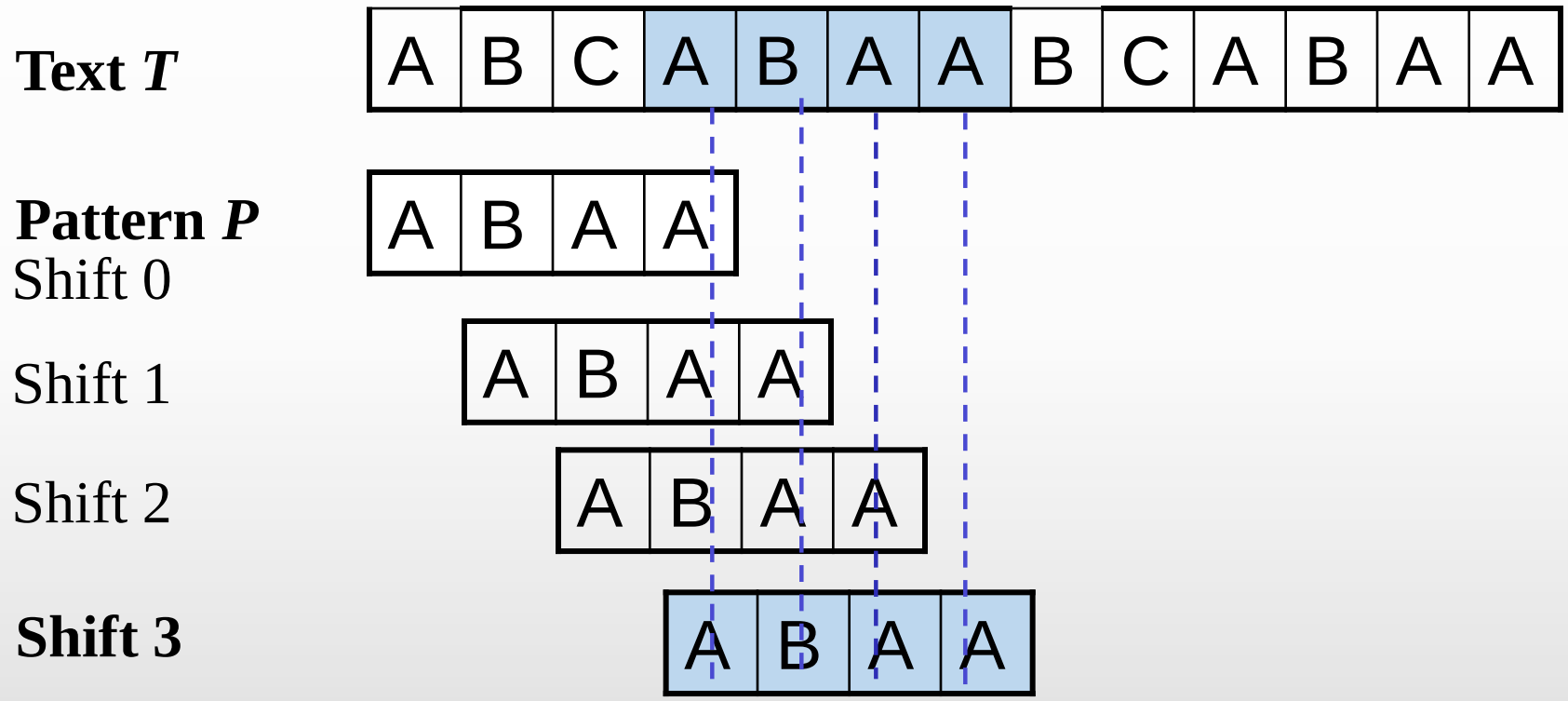
- We formulize the ***String Matching problem*** as follows.
- We assume that the text is an array **T[1..*n*]** of length ***n*** and the pattern is an array **P[1..*m*]** of length ***m***.
- We further assume that the elements of P and T are characters drawn from a finite alphabet $\sum$.
- For example we may have $\sum$={0,1} or $\sum$={a,b,..z}.
- The character arrays P and T are often called ***Strings*** of characters.

String Matching problem(contd.)

- We say that pattern ***P*** occurs with shift ***s*** in text ***T*** (or, equivalently that pattern P occurs beginning at position *s* +1 in text ***T***)

$$0 \leq s \leq n\text{-}m \text{ and } T[s+1..\ s+m] = P[1..m].$$

If P occurs with shift ***s*** in ***T***, then we call s a ***valid shift***; otherwise we call ***s*** an ***invalid shift***.



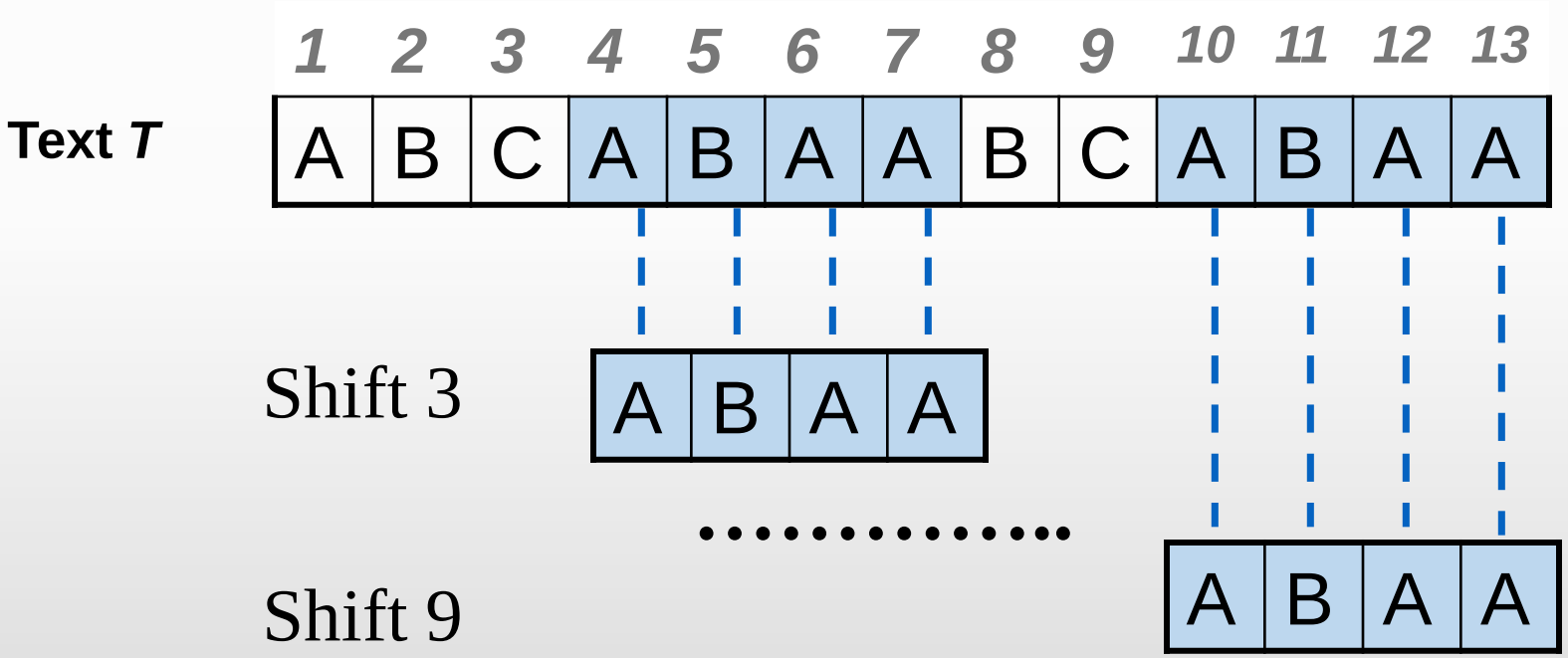
Here, n = 13 m = 4 s = 3

S = 3 is a valid shift because

0≤ **3** ≤ 13-4 and $T[3+1..3+4] = P[1..4]$.

String Matching problem(contd.)

The string-matching problem is the problem of finding all valid shifts with which a given pattern *P* occurs in a given text *T*.

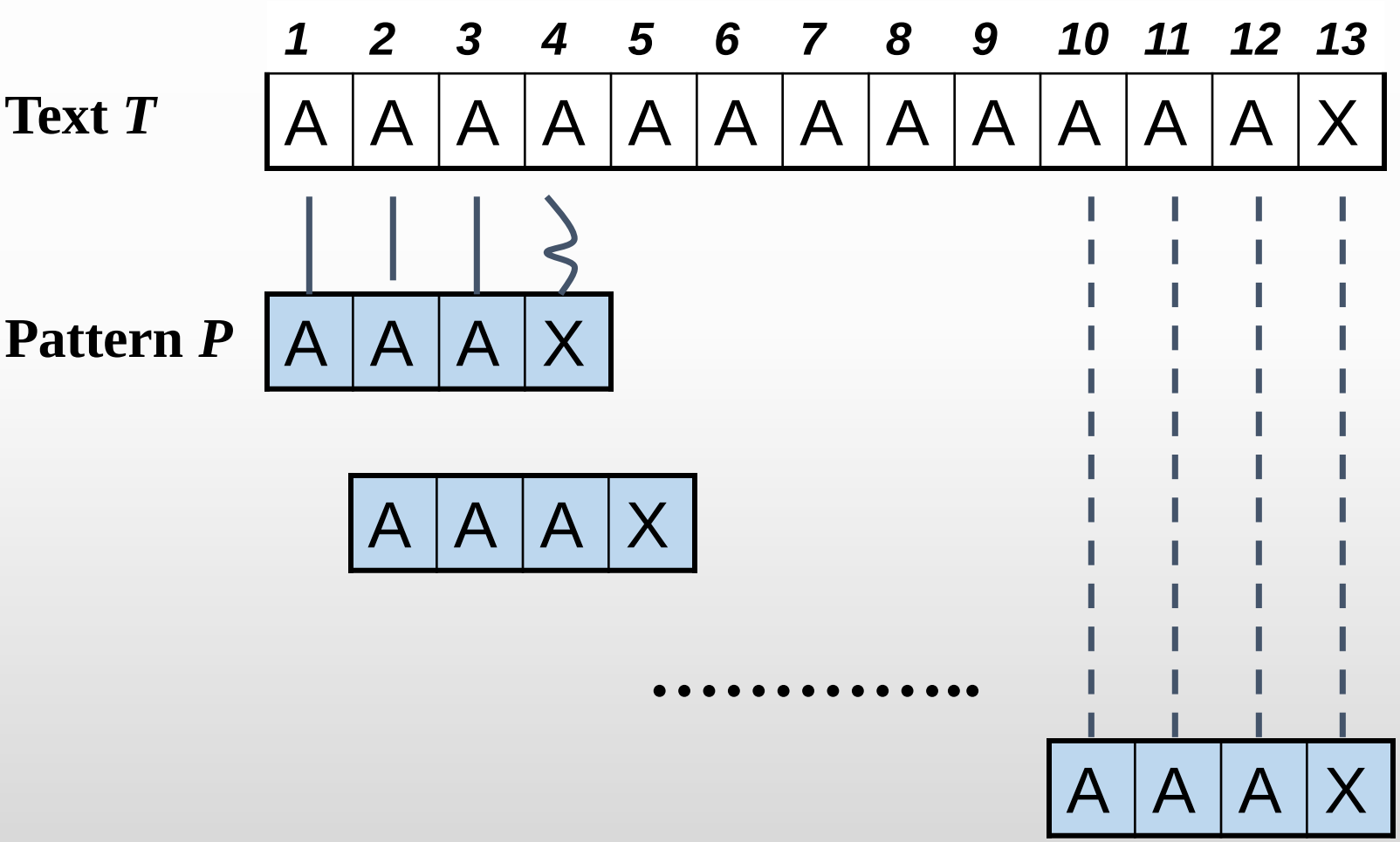


The naive string-matching algorithm

- The naïve algorithm finds all valid shifts using a loop that checks the condition P[1..m] = T[s+1..s+m] for each of the n-m+1 possible values of s.

```
Naïve-String-Matcher (T, P)
1    n = T.length
2    m = P.length
3    for s = 0 to n-m
4        if P[1..m] = T[s+1..s+m]
5                print "Pattern occurs with shift" s
```

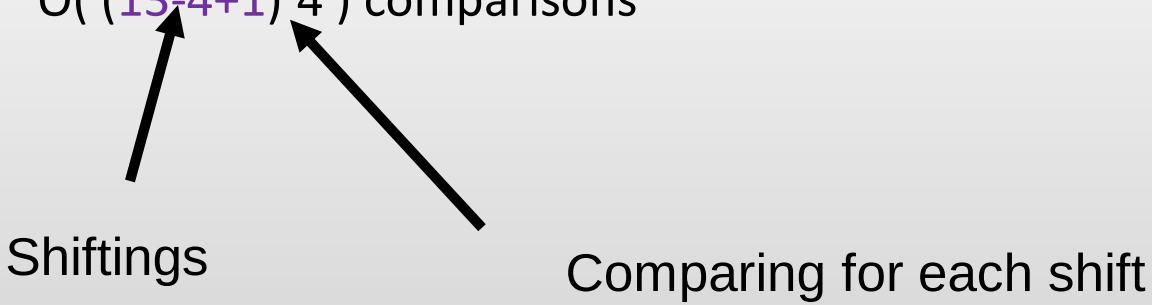
When does worst case happen?



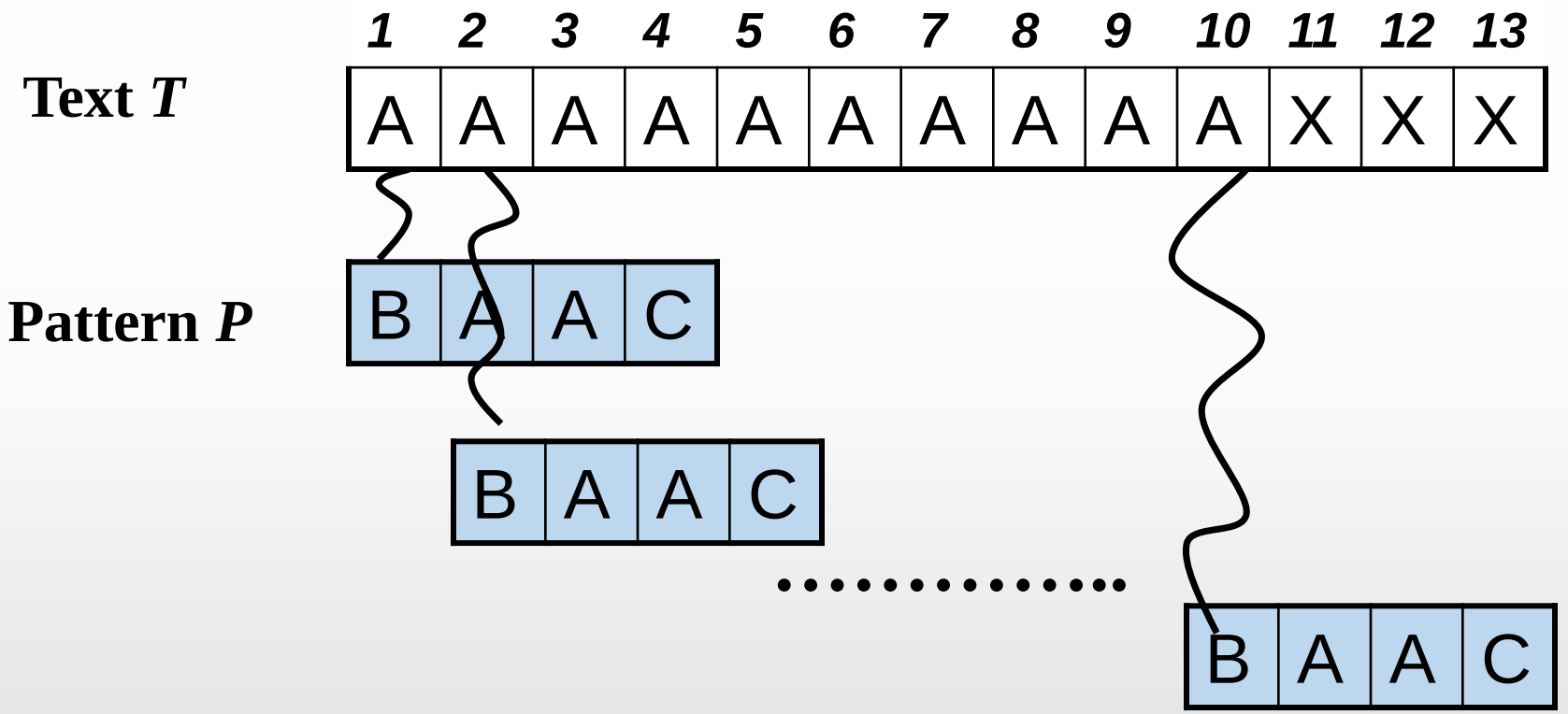
Worst case Analysis

- This algorithm takes O((n-m+1)m) in the worst case
- There are at most n-m +1 shifts
- m is to compare each string after shifting
- Therefore worst case takes O((n-m+1)m)
- E.g. In above example

  O((13-4+1) 4) comparisons



Best case analysis



No of shifts (13-4+1)

No of comparisons O((n-m+1) 1)

The Rabin-Karp Algorithm

Professor.Richard Karp **Harvard University**

Professor Michel Rabin Harvard University

Invented by Professor Richard Karp an Professor Michel Rabin in 1984.

The Rabin-Karp Algorithm

Given Text

2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

Find the occurrence of pattern

3	1	4	1	5

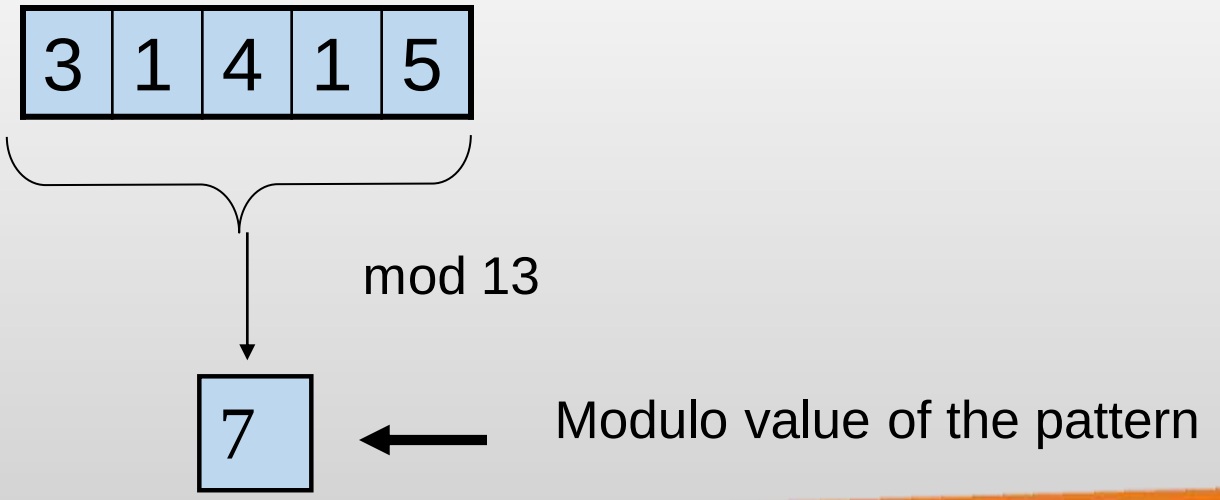
Using modulo 13

2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

3	1	4	1	5

Method

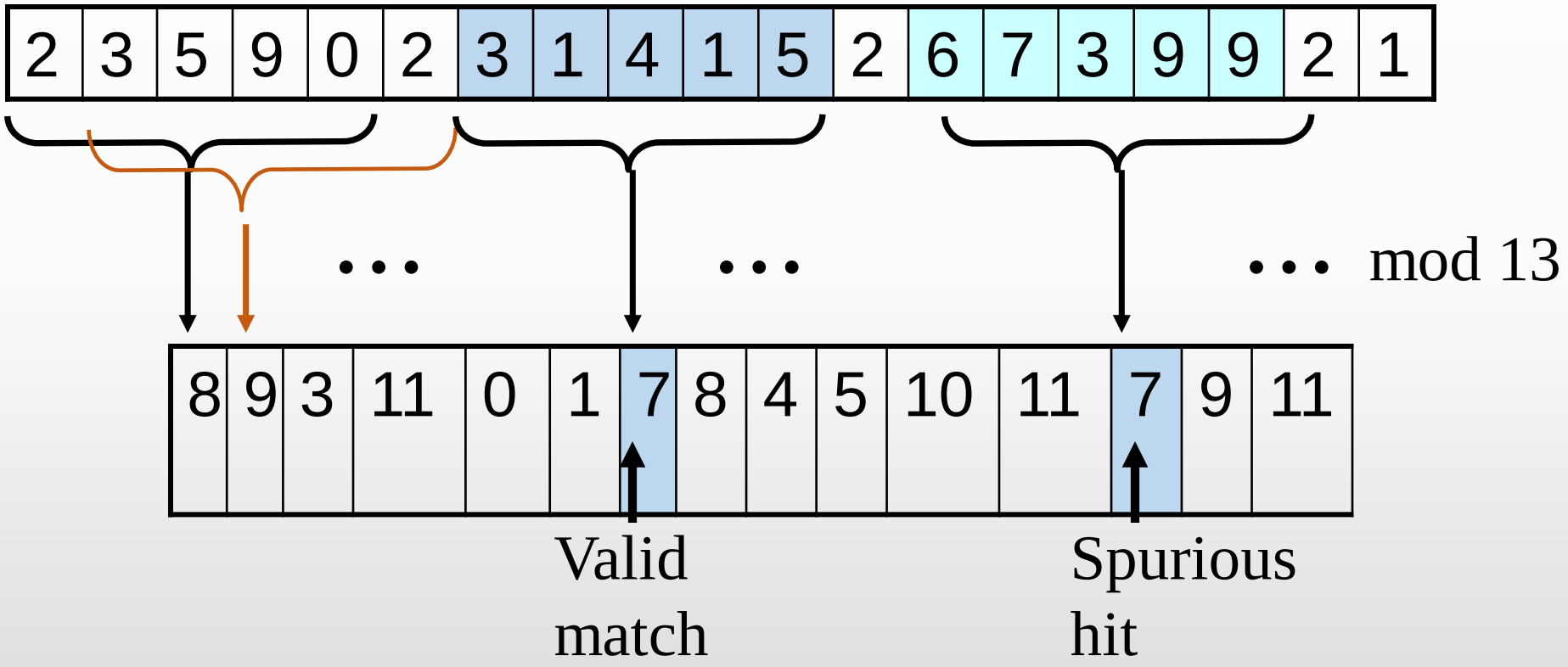
- Take the window size of the pattern
- Start from the beginning of the text taking windows
- Calculate the modulo value of that window
- Check it with the modulo value of the pattern



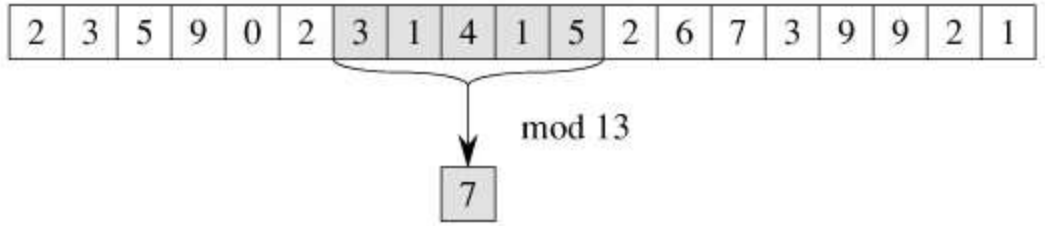
The Rabin-Karp Algorithm(contd.)

- Let us assume that the input alphabet is {0,1,2,…,9},so that each character is a decimal digit.
- We can then view a string of k consecutive characters as representing a length-k decimal number.
- The character string 31425 thus corresponds to the decimal number 31,425.

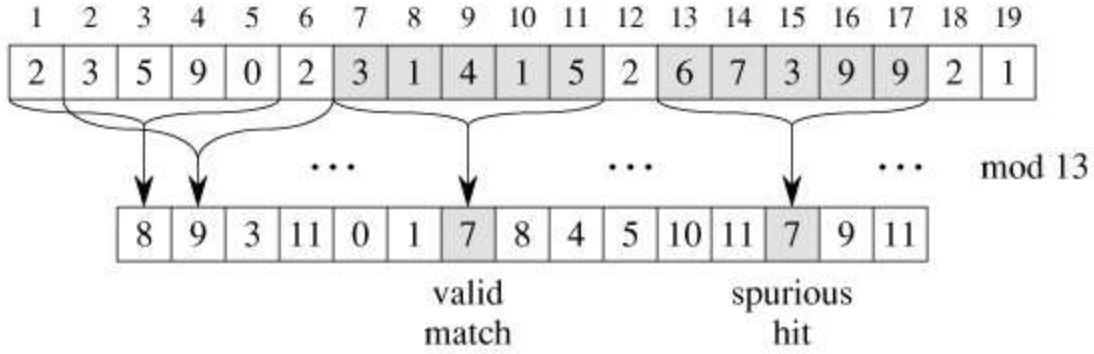
The Rabin-Karp Algorithm



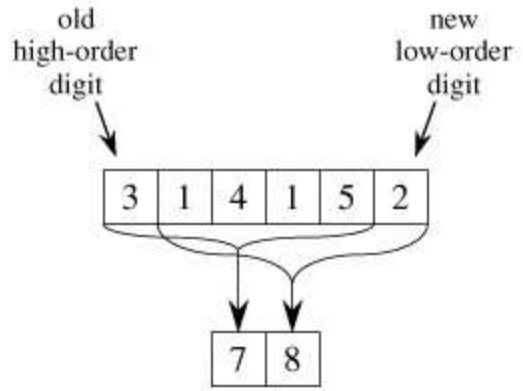
All the hits should be checked further.


2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1
mod 13
7

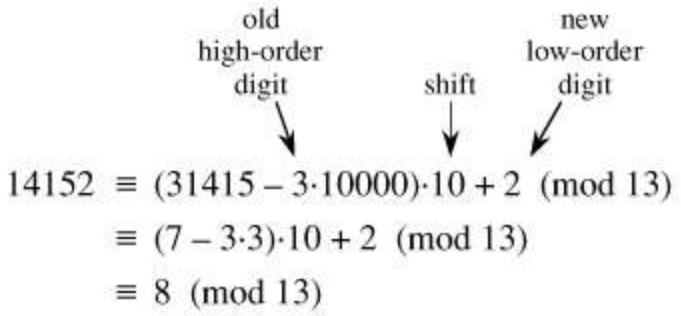
(a)


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1
mod 13
8 9 3 11 0 1 7 8 4 5 10 11 7 9 11
valid match
spurious hit

(b)


old high-order digit
new low-order digit
3 1 4 1 5 2
7 8

(c)


old high-order digit
shift
new low-order digit
14152 ≡ (31415 – 3·10000)·10 + 2 (mod 13)
≡ (7 – 3·3)·10 + 2 (mod 13)
≡ 8 (mod 13)

Analysis of Rabin-Karp

- Comparing only the modulo value does not guarantee that the exact pattern is found.
- On the other hand, if modulo values are not matched, then we definitely know that it is not the pattern.
- All the hits must be tested further to see if the hit is a valid shift or just a spurious hit.
- This testing can be done by explicitly checking the condition $P[1..m] = T[s+1..s+m]$
- If q is large enough, then we can hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

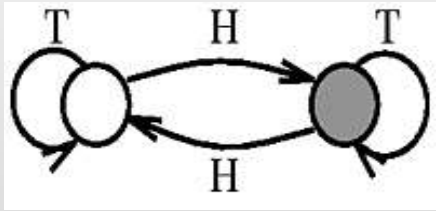
Worst case & Best case

- If all the hits are spurious hits then we have to check each of those.
- Therefore the worst case occurs when all the hits are spurious hits

  - If all the hits are neither spurious hits nor valid hits we don't have to check each of those.
  - Therefore the best case occurs when no hits occurred

String matching with finite automata

- Many string-matching algorithms build a finite automaton that scans the text string *T* for all occurrences of the pattern *P.*
- This section presents a method for building such an automaton.
- These string-matching automata are very efficient: they examine each text character *exactly once,* taking constant time per text character.

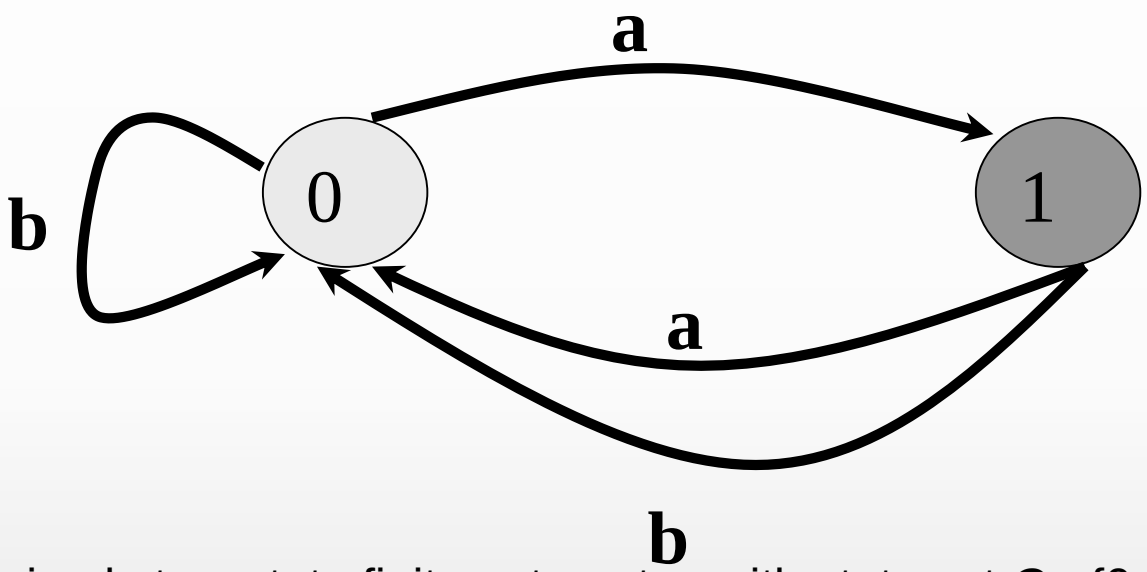


Definition of a finite automaton

A ***finite automaton M*** is a 5-tuple ***(Q, q_o, A, Σ, δ),*** where

- ***Q*** is a finite set of ***states***,
- ***q*** ε *Q* is the ***start state***,
- ***A*** $\subseteq$ *Q* is a distinguished set of ***accepting states***,
- $\sum$ is a finite ***input alphabet,***
- δ is a function from *Q* x $\sum$ into *Q,,* called the ***transition function*** *of M.*

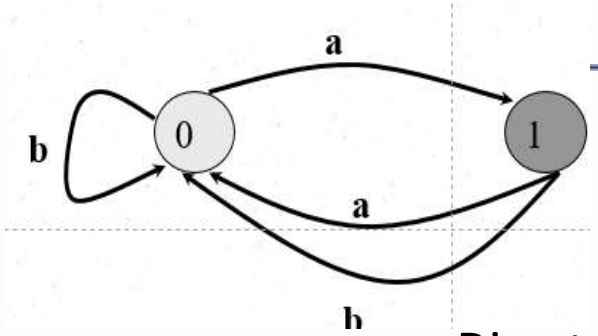
Example



state	input a	input b
0	1	0
1	0	0

A simple two-state finite automaton with state set ***Q***= {0,1},
start state q_0 = 0,
Accepting state ***A*** = 1
input alphabet ∑ = {a,b}
A tabular representation of the transition function δ

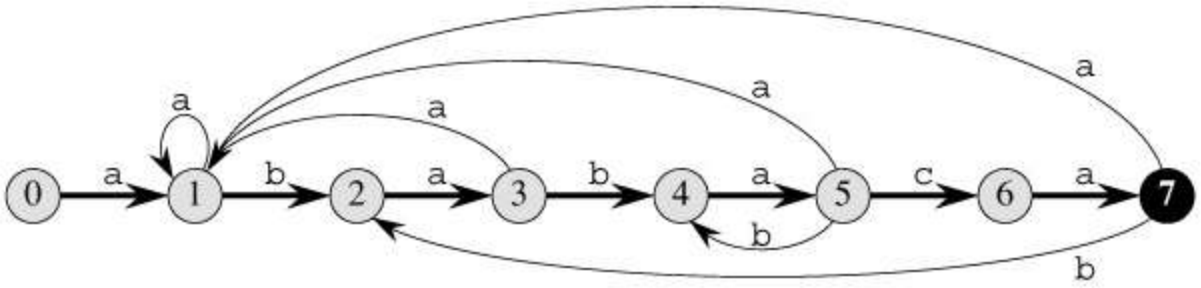
Finite automata(contd.)



- Directed edges represent transitions.
- For example, the edge from state 1 to state 0 labeled **b** indicates δ (1, b) = 0.
- This automaton accepts those strings that end in an odd number of ***a***'s.
- For example, the sequence of states this automaton enters for input **abaaa** (including the start state) is (0, 1, 0, 1, 0, 1) , and so it accepts this input.
- For input **abbaa**, the sequence of states is (0, 1,0,0, 1,0), and so it rejects this input.

Example. Accepts all strings ending in the string ababaca.

- *(a)* A state-transition diagram for the string-matching automaton that accepts all strings ending in the string ababaca. State 0 is the start state, and state 7 (shown blackened) is the only accepting state. A directed edge from state i to state j labeled a represents $\delta(i, a) = j$. The right-going edges forming the "spine" of the automaton, shown heavy in the figure, correspond to successful matches between pattern and input characters. The left-going edges correspond to failing matches. Some edges corresponding to failing matches are not shown; by convention, if a state i has no outgoing edge labeled a for some $a \in \Sigma$, then $\delta(i, a) = 0$.
- *(b)* The corresponding transition function δ, and the pattern string P = ababaca. The entries corresponding to successful matches between pattern and input characters are shown shaded.
- *(c)* The operation of the automaton on the text T = abababacaba. Under each text character $T[i]$ is given the state $\varphi(Ti)$ the automaton is in after processing the prefix Ti. One occurrence of the pattern is found, ending in position 9.



(a)

state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	**7**	2	3

(c)

Summary

- String Matching
- The naïve string matching algorithm
- The Rabin-Karp Algorithm
- Finite automata