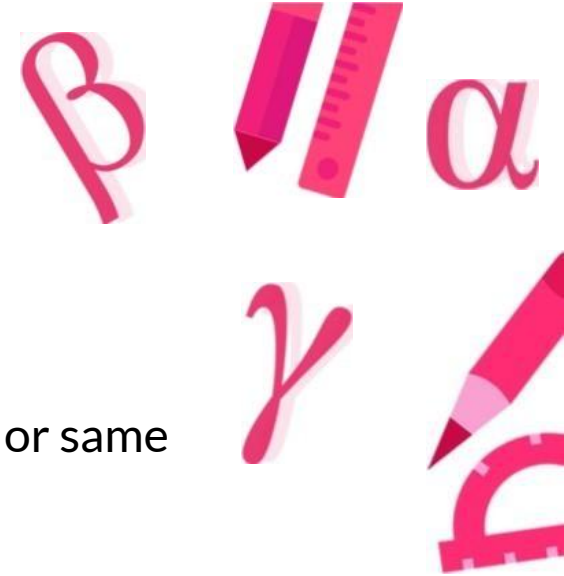
OBJECT ORIENTED ANALYSIS & DESIGN
DATA STRUCTURES & ALGORITHMS

Introduction to Data Structures

LearnOA®



Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.

- It means an array can contain one type of data only, either all integer, all float-point number or all character.
- Simply, declaration of array is as follows:

  int arr[10]

- Where int specifies the data type or type of elements arrays stores.

- "arr" is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Arrays

Following are some of the concepts to be remembered about arrays:

- The individual element of an array can be accessed by specifying name of the array, following by index or subscript inside square brackets.
- The first element of the array has index zero[0]. It means the first element and last element will be specified as:arr[0] & arr[9] Respectively.
- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

  (Upperbound-lowerbound)+1

Arrays

- **For the above array it would be**

  (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.

- Array can always be read or written through loop. If we read a one-dimensional array it require one loop for reading and other for writing the array.

Arrays

```
import java.util.Scanner;
public class Main
{
	public static void main(String[] args) {
		Scanner sc =  new Scanner(System.in);


		int n ;
		System.out.println("enter size of array");
		n = sc.nextInt();
		int a[] = new int[n];
		System.out.println("enter array elements");
		for(int i=0;i<n;i++)
		{a[i]=sc.nextInt();
		}

		for(int i=0;i<n;i++)
		{System.out.println(a[i]);
		}
}
}
```

Arrays

- If we are reading or writing two-dimensional array it would require two loops. And similarly the array of a N dimension would required N loops.

- **Some common operation performed on array are:**
  - ✔ Creation of an array
  - ✔ Traversing an array

Arrays

- Insertion of new element
- Deletion of required element
- Modification of an element
- Merging of arrays

Array Usage – A Perspective

Consider the following example:

```
import java.util.Scanner;
public class Main
{
	public static void main(String[] args) {
		Scanner sc =  new Scanner(System.in);

		int n ;
		System.out.println("enter size of array");
		n = sc.nextInt();
		int a[] = new int[n];
		System.out.println("enter array elements");
		for(int i=0;i<n;i++)
		{a[i]=sc.nextInt();
		}

		for(int i=0;i<n;i++)
		{System.out.println(a[i]);
		}
}
```

Array Usage – A Perspective

- When this program is compiled, the compiler estimates the amount of memory required for the variables, and also the instructions defined by you as part of the program.
- The compiler writes this information into the header of the executable file that it creates. When the executable is loaded into memory at runtime, the specified amount of memory is set aside.
- A part of the memory allocated to the program is in an area of memory called a runtime stack or the call stack. Once the size of the stack is fixed, it cannot be changed dynamically.

Array Usage – A Perspective

- Therefore, arrays present the classic problem of the programmer having allocated too few elements in the array at the time of writing the program, and then finding at run time that more values are required to be stored in the array than what had originally been defined.

- The other extreme is of the programmer allocating too many elements in the array and then finding at run time that not many values need to be stored in the array thereby resulting in wastage of precious memory.

Array Usage – A Perspective

- Moreover, array manipulation (in terms of insertion and deletion of elements from the array) is more complex and tedious, and a better alternative to all this is to go for dynamic data structures.

- Before venturing into dynamic variables, or dynamic data structures, it would be prudent at this juncture to differentiate between stack and heap variables, and their characteristics.

- Let us begin by understanding the concept of lifetime of variables.

Lifetime Of A Variable

- Is a run-time concept

- Is a period of time during which a variable has memory space associated with it
  - ✔ begins when space is allocated
  - ✔ ends when space is de-allocated

- Has three categories
  - ✔ static - start to end of program execution
  - ✔ automatic (stack) - start to end of declaring function's execution
  - ✔ heap (variable declared dynamic at runtime, and also de-allocated dynamically at runtime.

Java Strings

With Strings in Java you can perform various operations, some of which are:

Search → The quick brown fox jumps over the lazy dog

Create Substring → The quick brown fox

Create new strings → The quick brown fox jumps over the lazy dog suddenly

Linked Lists

Self-Referential Structures

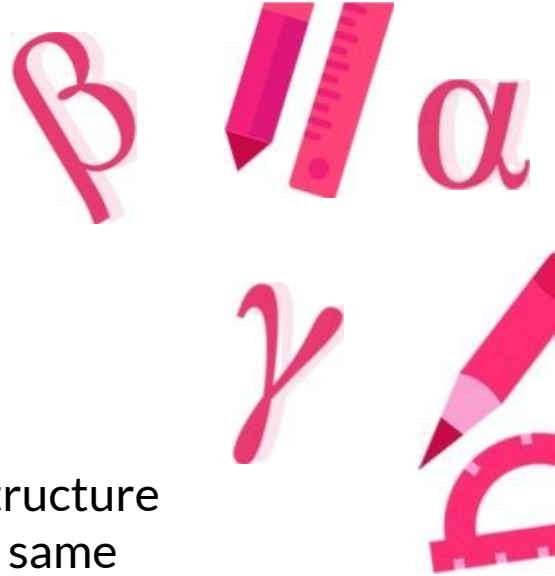
- Suppose, you have been given a task to store a list of marks. The size of the list is not known.
- If it were known, then it would have facilitated the creation of an array of the said number of elements and have the marks entered into it.
- Elements of an array are contiguously located, and therefore, array manipulation is easy using an integer variable as a subscript, or using pointer arithmetic.
- However, when runtime variables of a particular type are declared on the heap, let's say a structure type in which we are going to store the marks, each variable of the structure type marks will be located at a different memory location on the heap, and not contiguously located.

Self-Referential Structures

- Therefore, these variables cannot be processed the way arrays are processed, i.e., using a subscript, or using pointer arithmetic.

- An answer to this is a self-referential structure.

Self-Referential Structures

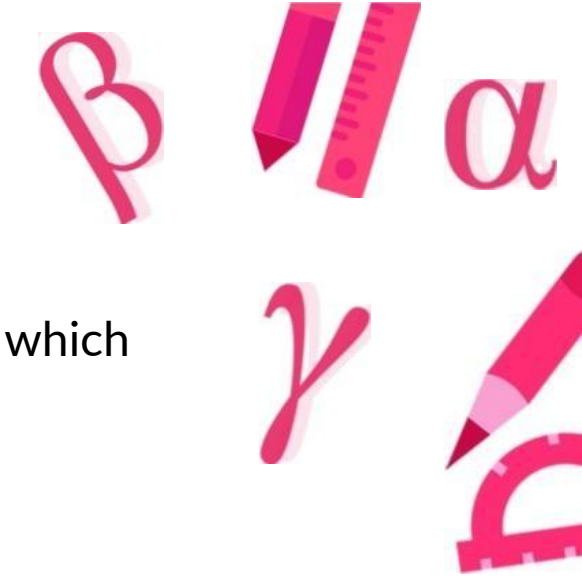


A self-referential structure is so defined that one of the elements of the structure variable is able to reference another subsequent structure variable of the same type, wherever it may be located on the heap.

In other words, each variable maintains a link to another variable of the same type, thus forming a non-contiguous, loosely linked data structure.

This self-referential data structure is also called a linked list.

Declaring a Linked List

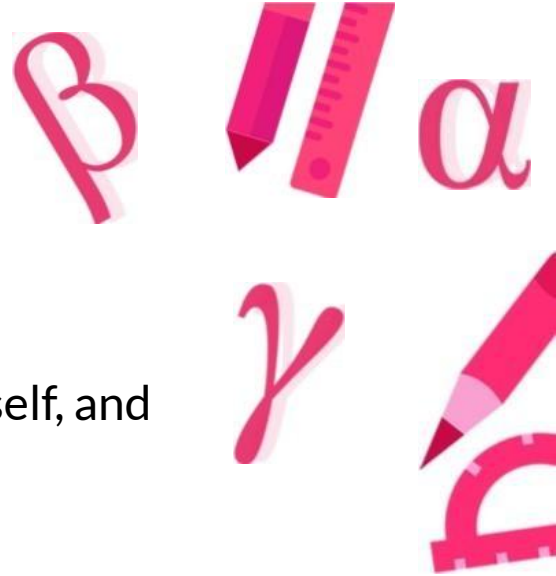


Let us define a self-referential structure to store a list of marks the size of which may not be known.

```
public class SinglyLinkedList {
   class Node{
      int data;
      Node next;

      public Node(int data) {
         this.data = data;
         this.next = null;
      }
   }
```

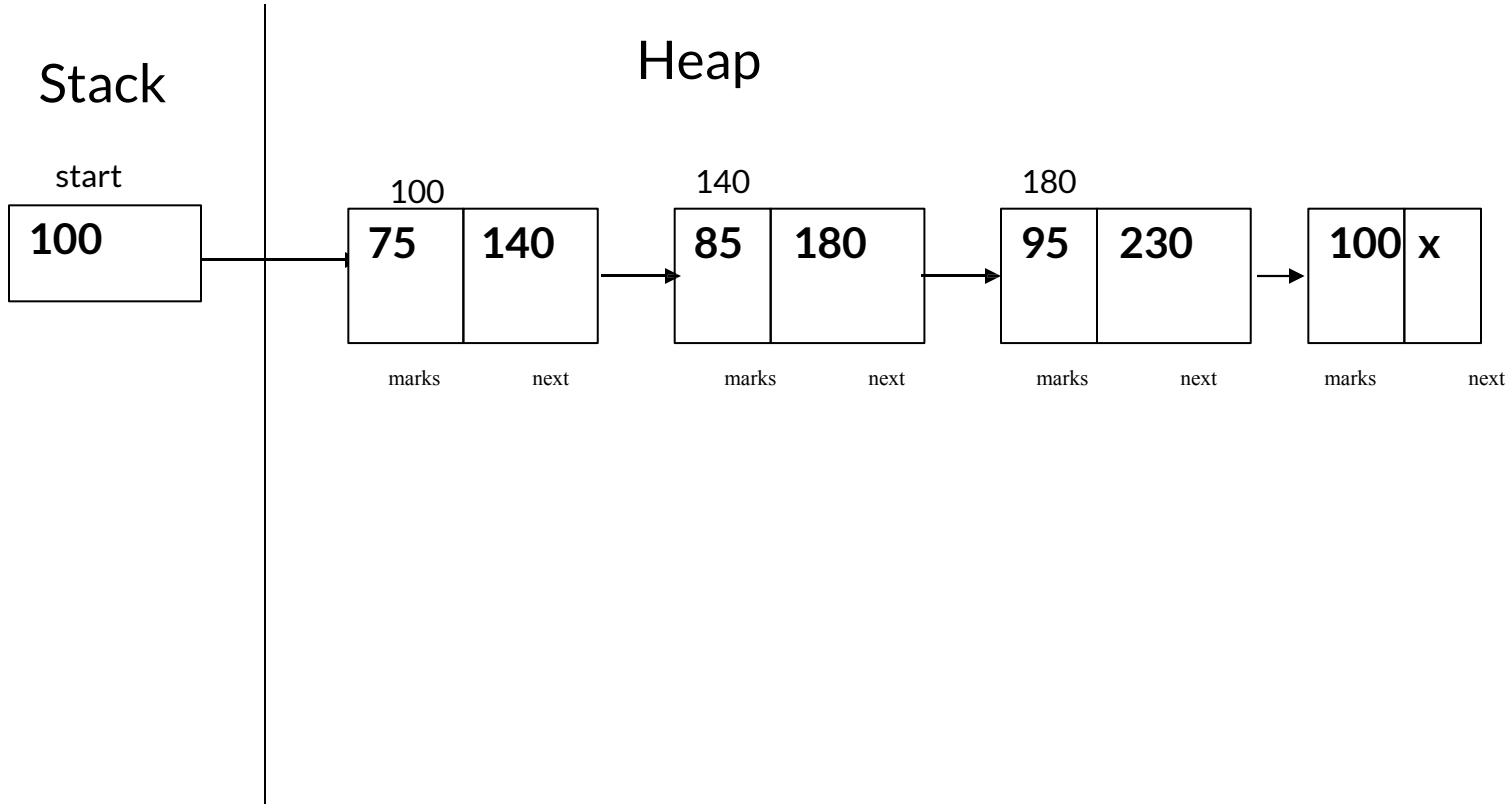
We have defined a structure of type marks_list. It consists of two elements, one integer element marks and the other element, a pointer next, which is a pointer to a structure of the same type, i.e., of type marks_list itself.

Declaring a Linked List



- Therefore, a part of the structure is referencing a structure type of itself, and hence the name **self-referential structure**.
- Such data structures are also popularly known as **linked lists**, since each structure variable contains a link to other structure variables of the same type.
- One can visualize a linked list as shown in the following slide:

Visualizing a Linked List



Creating a Sorted Linked List

```
public class SinglyLinkedList {
//Represent a node of the singly linked list
   class Node{
      int data;
      Node next;

      public Node(int data) {
         this.data = data;
         this.next = null;
      }
   }

   //Represent the head and tail of the singly linked list
   public Node head = null;
   public Node tail = null;

   //addNode() will add a new node to the list
   public void addNode(int data) {
      //Create a new node
      Node newNode = new Node(data);
```

Creating a Sorted Linked List

```
    if(head == null) {

        head = newNode;
        tail = newNode;
    }
    else {
                tail.next = newNode;

        tail = newNode;
    }
}
```

Creating a Sorted Linked List

```
public void display() {
   //Node current will point to head
   Node current = head;

   if(head == null) {
      System.out.println("List is empty");
      return;
   }
   System.out.println("Nodes of singly linked list: ");
   while(current != null) {
```

Creating a Sorted Linked List

```
        System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {

        SinglyLinkedList sList = new SinglyLinkedList();

        //Add nodes to the list
        sList.addNode(1);
        sList.addNode(2);
        sList.addNode(3);
        sList.addNode(4);

        //Displays the nodes present in the list
        sList.display();
    }
}
```

Doubly Linked Lists

Need For a Doubly Linked List

- The disadvantage with a singly linked list is that traversal is possible in only direction, i.e., from the beginning of the list till the end.
- If the value to be searched in a linked list is toward the end of the list, the search time would be higher in the case of a singly linked list.
- It would have been efficient had it been possible to search for a value in a linked list from the end of the list.

Properties of a Doubly Linked List

- This would be possible only if we have a doubly linked list.
- In a doubly linked list, each node has two pointers, one say, **next** pointing to the next node in the list, and another say, **prior** pointing to the previous node in the list.
- Therefore, traversing a doubly linked list in either direction is possible, from the start to the end using next, and from the end of the list to the beginning of the list using prior.

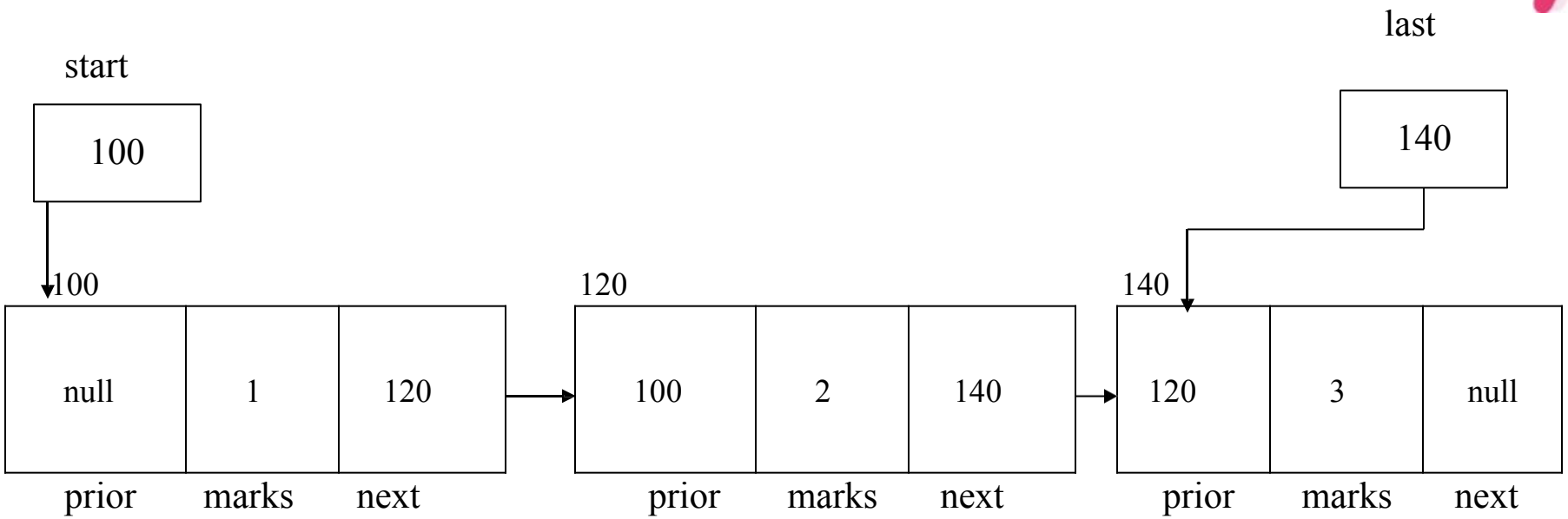
Properties of a Doubly Linked List

- In a doubly linked list, the **prior pointer of the first node will be null**, as there is no node before the first node.
- In a doubly linked list, the **next pointer of the last node will be null**, as there is no node after this list.
- Bidirectional traversal of a doubly linked list is useful for implementing **page up**, and **page down** functionality when using doubly linked lists to create editors.
- A doubly linked list would have two pointers, **start** and **last** to facilitate traversal from the beginning and end of the list respectively.

Declaration of a Doubly Linked List

```
public class DoublyLinkedList {
  // node creation
  Node head;
  class Node {
    int data;
    Node prev;
    Node next;

    Node(int d) {
      data = d;
    }
  }
```

Visualizing a Doubly Linked List



Creating a Sorted Doubly Linked List

```
public class DoublyLinkedList {
  // node creation
  Node head;
  class Node {
    int data;
    Node prev;
    Node next;

    Node(int d) {
      data = d;
    }
  }
```

Creating a Sorted Doubly Linked List

```
// insert node at the front
 public void insertFront(int data) {
   // allocate memory for newNode and assign data to newNode
   Node newNode = new Node(data);
   // make newNode as a head
   newNode.next = head;
newNode.prev = null;
if (head != null)
     head.prev = newNode;

   // head points to newNode
   head = newNode;
 }
```

Creating a Sorted Doubly Linked List

```
public void insertAfter(Node prev_node, int data) {
   if (prev_node == null) {
     System.out.println("previous node cannot be null");
     return;
   }
Node new_node = new Node(data);
 new_node.next = prev_node.next;
  prev_node.next = new_node;
 new_node.prev = prev_node;
 if (new_node.next != null)
     new_node.next.prev = new_node;
  }
```

Creating a Sorted Doubly Linked List

```
void insertEnd(int data) {
    Node new_node = new Node(data);

Node temp = head;
    new_node.next = null;

if (head == null) {
      new_node.prev = null;
      head = new_node;
      return;
    }

while (temp.next != null)
      temp = temp.next;

temp.next = new_node;

new_node.prev = temp;
  }
```

Creating a Sorted Doubly Linked List

```
 void deleteNode(Node del_node) {
   if (head == null || del_node == null) {
     return;
   }
if (head == del_node) {
     head = del_node.next;
   }
 if (del_node.next != null) {
     del_node.next.prev = del_node.prev;
   }
 if (del_node.prev != null) {
     del_node.prev.next = del_node.next;
   }
 }
```

Creating a Sorted Doubly Linked List

```
public void printlist(Node node) {
  Node last = null;
  while (node != null) {
    System.out.print(node.data + "->");
    last = node;
    node = node.next;
  }
  System.out.println();
 }
```

Creating a Sorted Linked List

```
 public static void main(String[] args) {
    DoublyLinkedList doubly_ll = new DoublyLinkedList();

    doubly_ll.insertEnd(5);
    doubly_ll.insertFront(1);
    doubly_ll.insertFront(6);
    doubly_ll.insertEnd(9);

doubly_ll.insertAfter(doubly_ll.head, 11);

doubly_ll.insertAfter(doubly_ll.head.next, 11);

    doubly_ll.printlist(doubly_ll.head);

doubly_ll.deleteNode(doubly_ll.head.next.next.next.next.next);

    doubly_ll.printlist(doubly_ll.head);
  }
}
```

Stack

What is a Stack?

A stack is a data structure in which insertions and deletions can only be done at the top.

A common example of a stack, which permits the selection of only its end elements, is a stack of books.

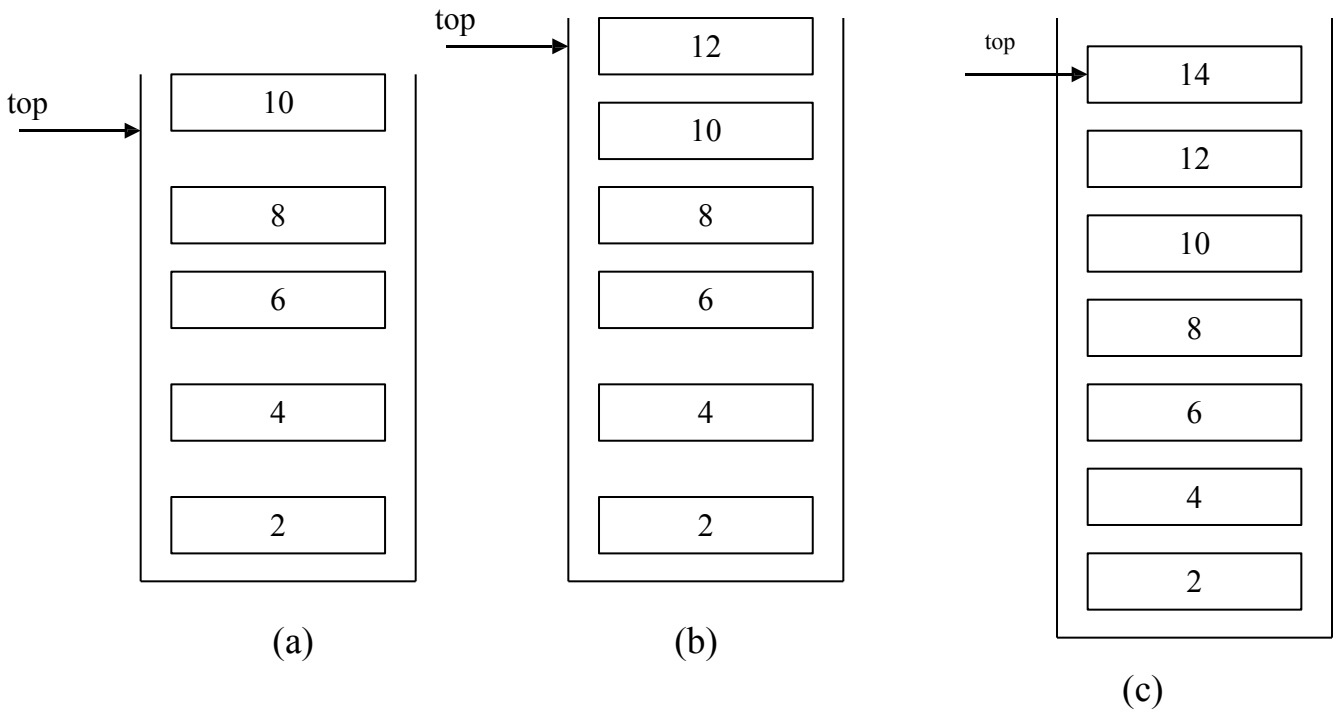
A person desiring a book can pick up only the book at the top, and if one wants to place a plate on the pile of plates, one has to place it on the top.

What is a Stack?

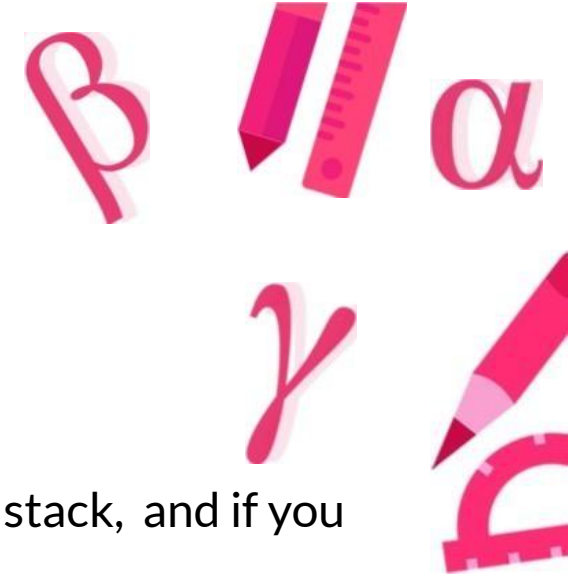
- You would have noticed that whenever a book is placed on top of the stack, the top of the stack moves upward to correspond to the new element (the stack grows).

- And, whenever a book is picked, or removed from the top of the stack, the top of the stack moves downward to correspond to the new highest element (the stack shrinks).

What is a Stack?

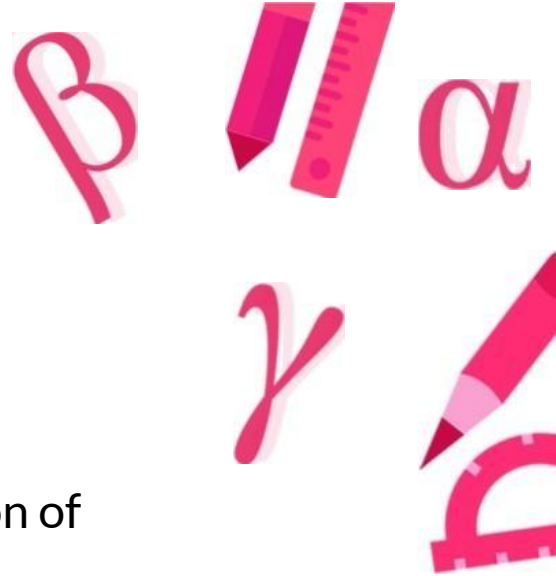


Characteristics of a Stack



- When you add an element to the stack, you say that you **push** it on the stack, and if you delete an element from a stack, you say that you **pop** it from the stack.
- Since the last item pushed into the stack is always the first item to be popped from the stack, a stack is also called as a **Last In, First Out or a LIFO structure**.
- Unlike an array that is static in terms of its size, **a stack is a dynamic data structure.**

Characteristics of a Stack



- Since the definition of the stack provides for the insertion and deletion of nodes into it, a stack can grow and shrink dynamically at runtime.

- An ideal implementation of a stack would be a special kind of linked list in which insertions and deletions can only be done at the top of the list.

Operations on Stacks

Some of the typical operations performed on stacks are:

- **create (s)** – to create s as an empty stack
- **push (s, i)** – to insert the element i on top of the stack **s**
- **pop (s)** – to remove the top element of the stack and to return the removed element as a function value.
- **top (s)** – to return the top element of stack(s)
- **empty(s)** – to check whether the stack is empty or not. It returns true if the stack is empty, and returns false otherwise.

Implementation of Stacks

- An array can be used to implement a stack.
- But since array size is defined at compile time, it cannot grow dynamically at runtime, and therefore, an attempt to insert an element into a array implementation of a stack that is already full causes a stack overflow.
- A stack, by definition, is a data structure that cannot be full since it can dynamically grow and shrink at runtime.

Implementation of Stacks

- An ideal implementation for a stack is a linked list that can dynamically grow and shrink at runtime.
- Since you are going to employ a variation of a linked list that functions as a stack, you need to employ an additional pointer (top) that always points to the first node in the stack, or the top of the stack.
- It is using top that a node will either be inserted at the beginning or top of the stack (push a node into the stack), or deleted from the top of the stack (popping a node at the top or beginning of the stack).

Code Implementing for a Stack

The push( ) and the pop( ) operations on a stack are analogous to the insert-first-node and delete-first-node operations on a linked list that functions as a stack.

```
class Stack
{
    private int arr[];
    private int top;
    private int capacity;

    // Constructor to initialize the stack
    Stack(int size)
    {
        arr = new int[size];
        capacity = size;
        top = -1;
    }
```

Code Implementing for a Stack

```
public void push(int x)
{
    if (isFull())
    {
        System.out.println("Overflow\nProgram Terminated\n");
        System.exit(-1);
    }

    System.out.println("Inserting " + x);
    arr[++top] = x;
}
```

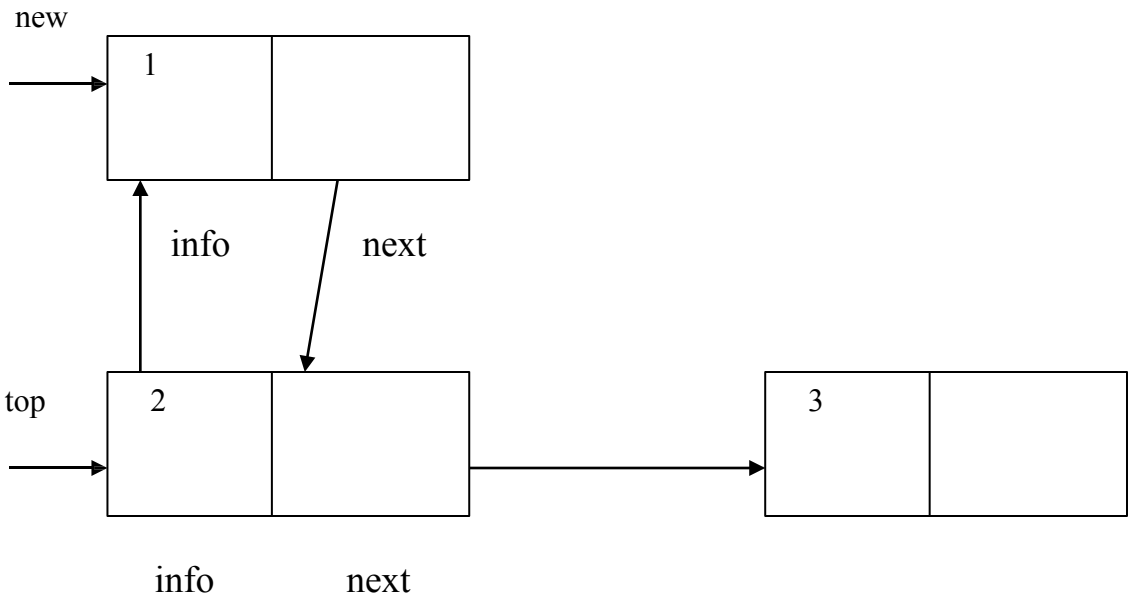
Code Implementing for a Stack

```
public int pop()
   {
      if (isEmpty())
      {
         System.out.println("Underflow\nProgram Terminated");
         System.exit(-1);
      }
      System.out.println("Removing " + peek()
      return arr[top--];
   }
```

Implementing push( )

```
public void push(int x)
  {
     if (isFull())
     {
        System.out.println("Overflow\nProgram Terminated\n");
        System.exit(-1);
     }

     System.out.println("Inserting " + x);
     arr[++top] = x;
  }
```

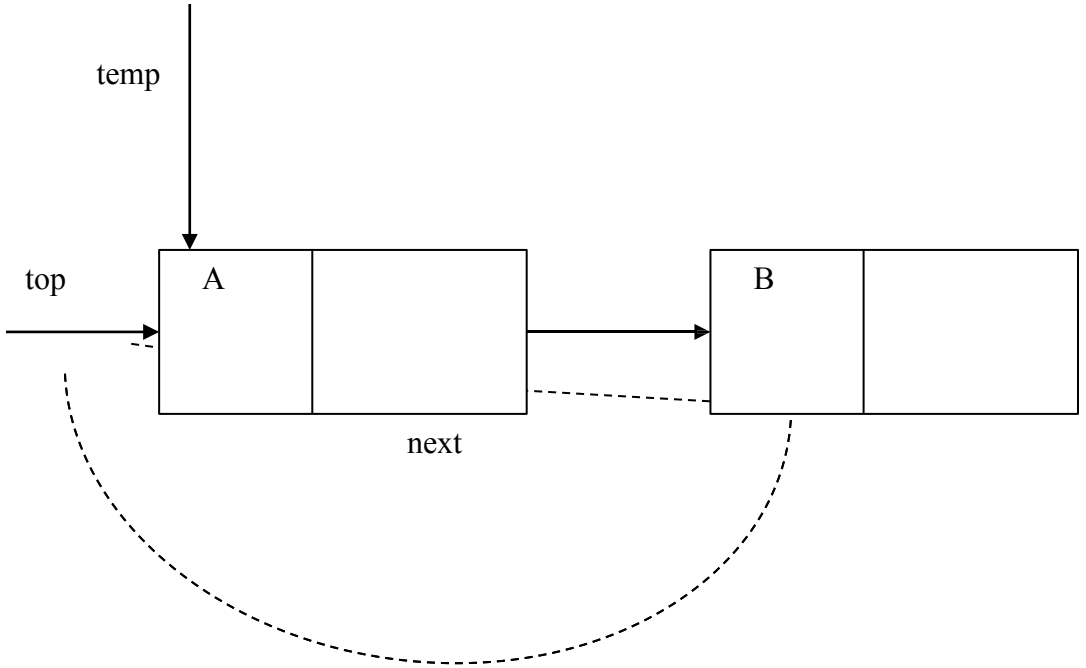
A View of the Stack After Insertion



Implementing pop( )

```
public int pop()
   {
      if (isEmpty())
      {
         System.out.println("Underflow\nProgram Terminated");
         System.exit(-1);
      }
      System.out.println("Removing " + peek()
      return arr[top--];
   }
```

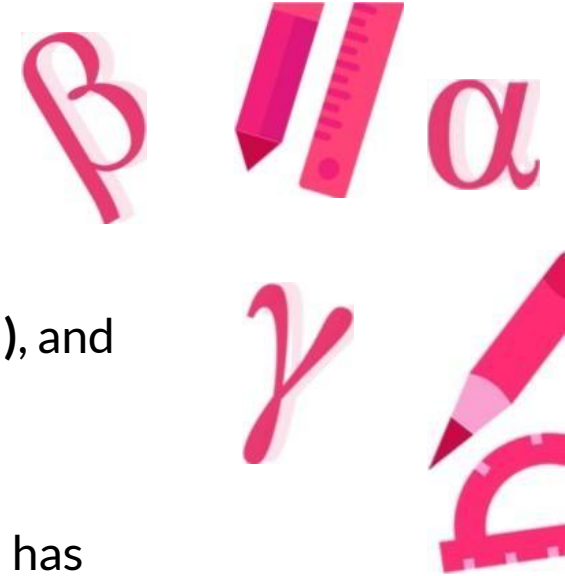
A View of the Stack After Deletion



Applications of Stacks

- **As a stack is a LIFO structure, it is an appropriate data structure for applications in which information must be saved and later retrieved in reverse order.**
- Consider what happens within a computer when function main( ) calls another function.
- How does a program remember where to resume execution from after returning from a function call?
- From where does it pick up the values of the local variables in the function main( ) after returning from the subprogram?

Applications of Stacks (Example)



- As an example, let **main( )** call **a( )**. Function **a( )**, in turn, calls function **b( )**, and function **b( )** in turn invokes function **c( )**.
- **main( )** is the first one to execute, but it is the last one to finish, after **a( )** has finished and returned.
- **a( )** cannot finish its work until **b( )** has finished and returned. **b( )** cannot finish its work until **c( )** has finished and returned.
- When **a( )** is called, its calling information is pushed on to the stack (calling information consists of the address of the return instruction in **main( )** after **a( )** was called, and the local variables and parameter declarations in **main( )**.
- When **b( )** is called from **a( )**, b( )'s calling information is pushed onto the stack (calling information consists of the address of the return instruction in **a( )** after **b( )** was called, and the local variables and parameter declarations in **a( )**).

Applications of Stacks

- Then, when **b( )** calls **c( )**, **c( )**'s calling information is pushed onto the stack (calling information consists of the address of the return instruction in **b( )** after **c( )** was called, and the local variables and parameter declarations in **b( )**).
- When **c( )** finishes execution, the information needed to return to **b( )** is retrieved by popping the stack.
- Then, when **b( )** finishes execution, its return address is popped from the stack to return to **a( )**
- Finally, when **a( )** completes, the stack is again popped to get back to **main( )**.
- When **main( )** finishes, the stack becomes empty.
- Thus, a stack plays an important role in function calls.

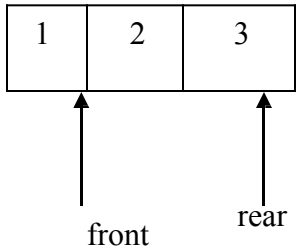
Queue

Defining a Queue

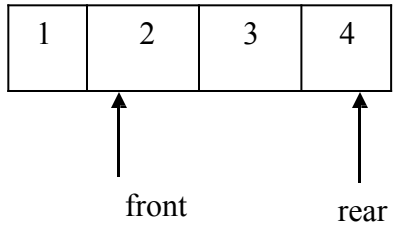
- A Queue is a data structure in which elements are added at one end (called the **rear**), and elements are removed from the other end (called the **front**).
- You come across a number of examples of a queue in real life situations.
- For example, consider a line of students at a fee counter. Whenever a student enters the queue, he stands at the end of the queue (analogous to the addition of nodes to the queue)
- Every time the student at the front of the queue deposits the fee, he leaves the queue (analogous to deleting nodes from a queue).
- The student who comes first in the queue is the one who leaves the queue first.

Therefore, a queue is commonly called a first-in-first-out or a FIFO data structure.

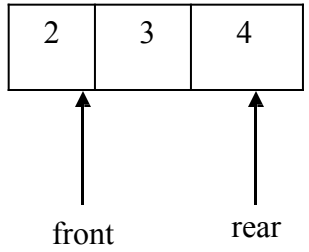
Queue Insertions and Deletions



(a)



(b)



(c)

Queue Operations

- To complete this definition of a queue, you must specify all the operations that it permits.
- The first step you must perform in working with any queue is to initialize the queue for further use. Other important operations are to add an element, and to delete an element from a queue.
- Adding an element is popularly known as ENQueue and deleting an element is know as DEQueue. The following slide lists operations typically performed on a queue.

Queue Operations

create(q) – which creates q as an empty queue

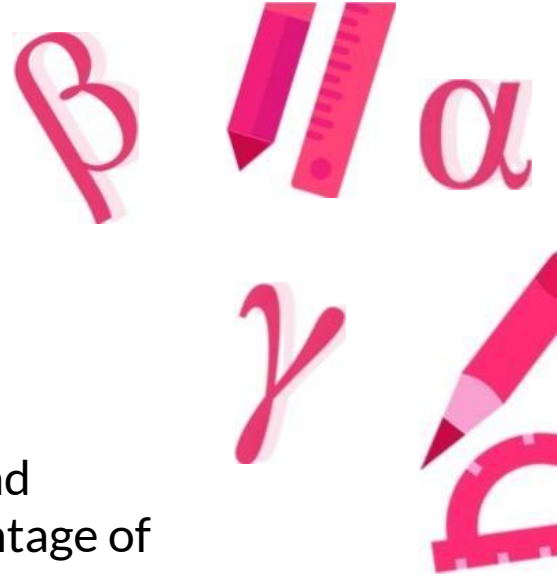
enq(i) – adds the element i to the rear of the queue and returns the new queue

deq(q) – removes the element at the front end of the queue (q) and returns the resulting queue as well as the removed element

empty (q) – it checks the queue (q) whether it is empty or not. It returns true if the queue is empty and returns false otherwise
front(q) – returns the front element of the queue without changing the

queue queuesize (q) – returns the number of entries in the queue

Implementing Queues



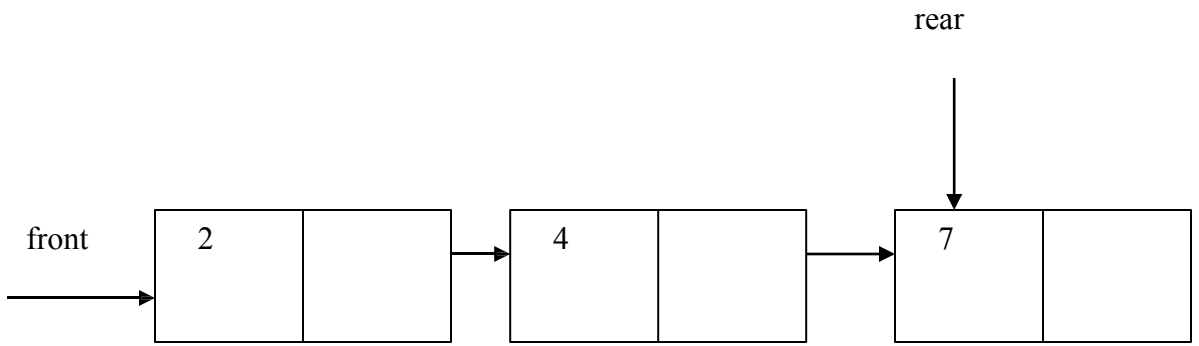
- Linked lists offer a flexible implementation of a queue since insertion and deletion of elements into a list are simple, and a linked list has the advantage of dynamically growing or shrinking at runtime.

- Having a list that has the functionality of a queue implies that insertions to the list can only be done at the rear of the list, and deletions to the list can only be done at front of the list.

Implementing Queues

- Queue functionality of a linked list can be achieved by having two pointers **front** and **rear**, pointing to the first element, and the last element of the queue respectively.

- The following figure gives a visual depiction of linked list implementation of a queue.



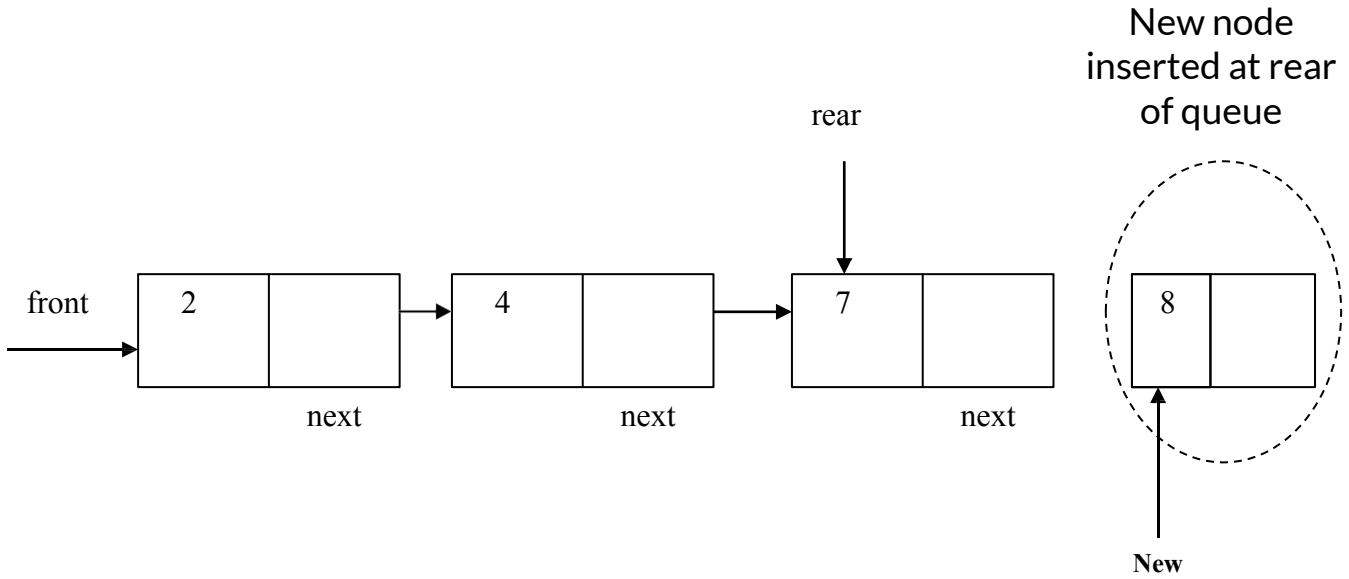
Queue Declaration & Operations

```
class Queue
{
    private int[] arr;      // array to store queue elements
    private int front;      // front points to the front element in the queue
    private int rear;       // rear points to the last element in the queue
    private int capacity;   // maximum capacity of the queue
    private int count;      // current size of the queue

      Queue(int size)
    {
        arr = new int[size];
        capacity = size;
        front = 0;
        rear = -1;
        count = 0;
    }
```

Insertion into a Queue

```
public void enqueue(int item)
{
    if (isFull())
    {
        System.out.println("Overflow\nProgram Terminated");
        System.exit(-1);
    }
    System.out.println("Inserting " + item);
rear = (rear + 1) % capacity;
    arr[rear] = item;
    count++;
```

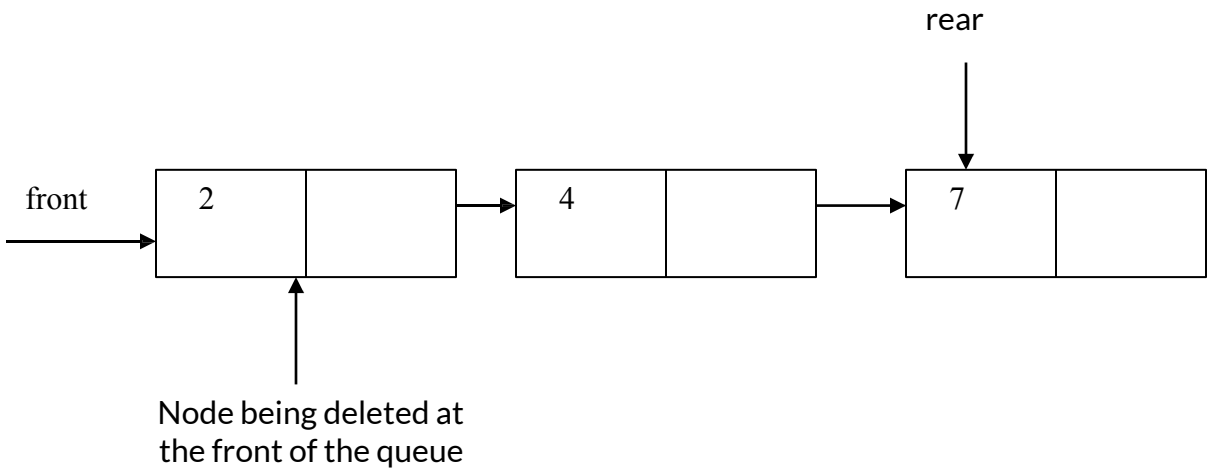
Insertion into a Queue



Deletion from a Queue

```
public int dequeue()
   {
      if (isEmpty())
      {
         System.out.println("Underflow\nProgram Terminated");
         System.exit(-1);
      }

      int x = arr[front];

      System.out.println("Removing " + x);

      front = (front + 1) % capacity;
      count--;

      return x;
   }
```

Deletion of a Node From a Queue



Applications of Queues

- Queues are very useful in a time-sharing multi-user operating system where many users share the CPU simultaneously.

Whenever a user requests the CPU to run a particular program, the operating system adds the request (by first converting the program into a process that is a running instance of the program, and then assigning the process an ID).

This process ID is then added at the end of the queue of jobs waiting to be executed.

Applications of Queues

- Whenever the CPU is free, it executes the job that is at the front of the job queue.

- Similarly, there are queues for shared I/O devices. Each device maintains its own queue of requests.

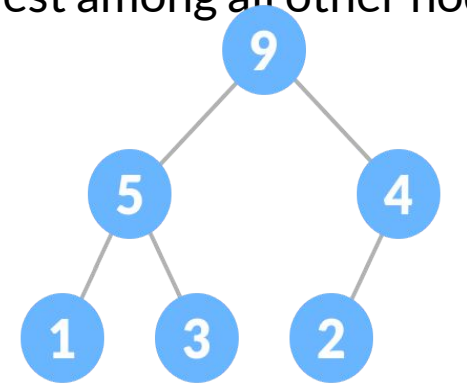
An example is a print queue on a network printer, which queues up print jobs issued by various users on the network.
The first print request is the first one to be processed. New print requests are added at the end of the queue.

Heap

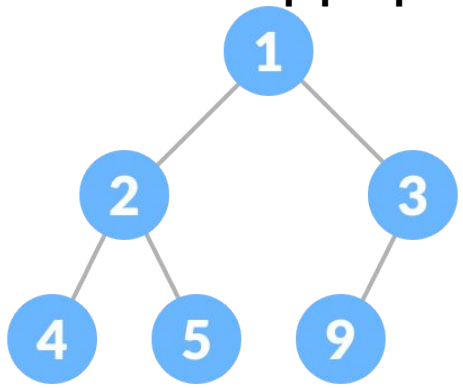
Heap data structure is a complete binary tree that satisfies **the heap property**, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called **max heap property**.

- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called **min heap property**.



Max heap



Min heap

Maps

A *Map* is a type of fast key lookup data structure that offers a flexible means of indexing into its individual elements. Unlike most array data structures that only allow access to the elements by means of integer indices, the indices for a Map can be nearly any scalar numeric value or a character vector.

Indices into the elements of a Map are called *keys*. These keys, along with the data *values* associated with them, are stored within the Map. Each entry of a Map contains exactly one unique key and its corresponding value.

Implementing maps

```
import java.util.Map;
import java.util.HashMap;
class Main {
   public static void main(String[] args) {
Map<String, Integer> numbers = new HashMap<>();
       numbers.put("One", 1);
       numbers.put("Two", 2);
 System.out.println("Map: " + numbers);
System.out.println("Keys: " + numbers.keySet());
System.out.println("Values: " + numbers.values());
System.out.println("Entries: " + numbers.entrySet());
int value = numbers.remove("Two");
       System.out.println("Removed Value: " + value);
   }
}
```

Dictionaries

Collection of pairs. (key, value)

Each pair has a unique key.

Operations.

Get(theKey) Delete(theKey) Insert(theKey, theValue)

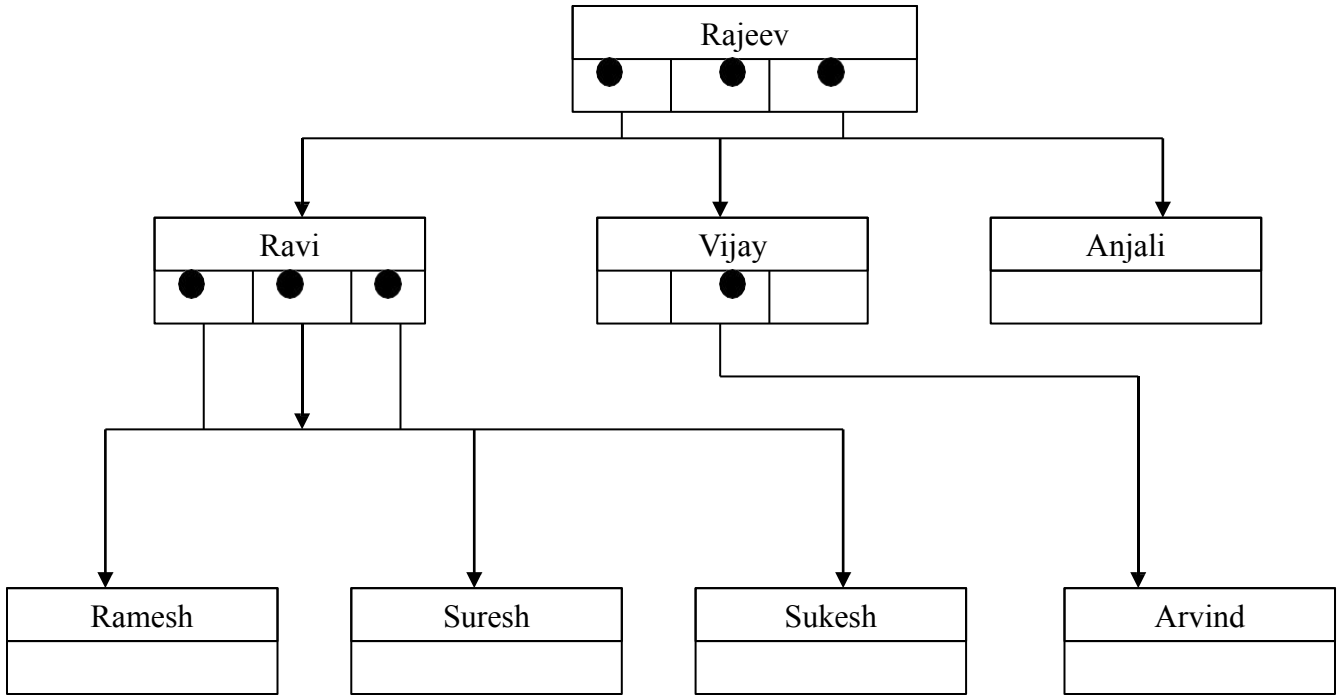
Implementing dictionaries

```
import java.util.*;
public class InsertElementExample
{
public static void main(String args[])
{
//creating a dictionary
Dictionary dict = new Hashtable();
//adding values in the dictionary
dict.put(101, "Sydney");
dict.put(102, "Brisbane");
dict.put(103, "Melbourne");
dict.put(104, "Perth");
dict.put(105, "Lismore");
dict.put(106, "Mount Gambier");
dict.put(107, "Nelson Bay");
dict.put(108, "Canberra");
//prints keys and corresponding values
System.out.println(dict);
}
}
```

Trees

- Compared to linked lists that are linear data structures, **trees are non-linear data structures**.
- In a linked list, each node has a link which points to another node.
- In a tree structure, however, each node may point to several nodes, which may in turn point to several other nodes.
- Thus, a tree is a very flexible and a powerful data structure that can be used for a wide variety of applications.

Trees



Trees

- A tree consists of a collection of nodes that are connected to each other.
- **A tree contains a unique first element known as the root, which is shown at the top of the tree structure.**
- A node which points to other nodes is said to be the parent of the nodes to which it is pointing, and the nodes that the parent node points to are called the children, or child nodes of the parent node.

Binary Trees

Binary Tree

- If you can introduce a restriction that each node can have a maximum of two children or two child nodes, then you can have a binary tree.
- You can give a formal definition of a binary tree as a tree which is either empty or consists of a root node together with two nodes, each of which in turn forms a subtree.
- You therefore have a left subtree and a right subtree under the root node.

Binary Search Tree

- A complete binary tree can be defined as one whose non-leaf nodes have non empty left and right subtrees and all leaves are at the same level.

- This is also called as a balanced binary tree. If a binary tree has the property that all elements in the left subtree of a node n are less than the contents of n, and all elements in the right subtree are greater than the contents of n, such a tree is called a binary search tree.

Implementing trees

```
public class BinarySearchTree {
    //Represent the node of binary tree
    public static class Node{
        int data;
        Node left;
        Node right;
        public Node(int data){
            //Assign data to the new node, set left and righ
t children to null
            this.data = data;
            this.left = null;
            this.right = null;
            }
        }
```

Implementing trees

```
public Node root;
    public BinarySearchTree(){
        root = null;
    }
    //factorial() will calculate the factorial of given numb
er
    public int factorial(int num) {
        int fact = 1;
        if(num == 0)
            return 1;
        else {
            while(num > 1) {
                fact = fact * num;
                num--;
            }
            return fact;
        }
    }
```

Implementing trees

```
 public int numOfBST(int key) {

        int catalanNumber = factorial(2 * key)/(factorial(key + 1) * factorial(key));

        return catalanNumber;
    }
    public static void main(String[] args) {
        BinarySearchTree bt = new BinarySearchTree();
        //Display total number of possible binary search tree with key 5
        System.out.println("Total number of possible Binary Search Trees with gi
ven key: " + bt.numOfBST(5));
     }
}
```

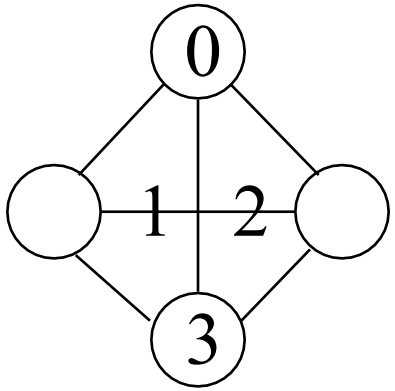
Graph

- A graph G consists of two sets
  - a finite, nonempty set of vertices V(G)
  - a finite, possible empty set of edges E(G)
  - G(V,E) represents a graph

- An undirected graph is one in which the pair of vertices in a edge is unordered, (v0, v1) = (v1,v0)

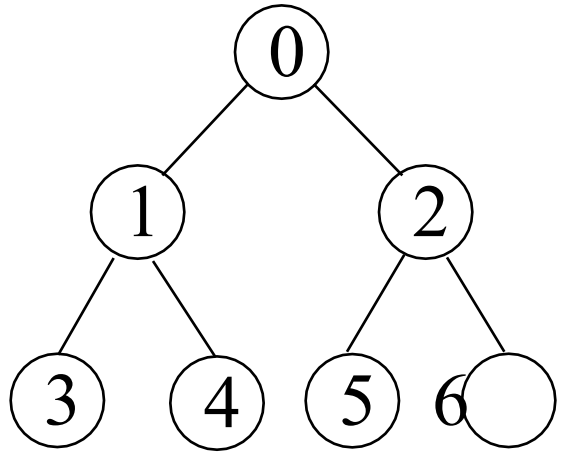
- A directed graph is one in which each edge is a directed pair of vertices, <v0, v1> != <v1,v0>

Examples for Graph



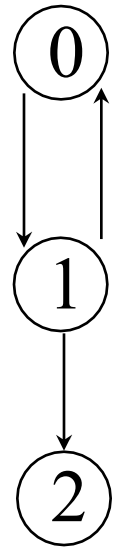
G_1

complete graph



G_2

incomplete graph



G_3

V(G1)={0,1,2,3} E(G1)={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}
V(G2)={0,1,2,3,4,5,6} E(G2)={(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)}
V(G3)={0,1,2} E(G3)={<0,1>,<1,0>,<1,2>}

complete undirected graph: n(n-1)/2 edges
complete directed graph: n(n-1) edges

Some Graph Operations

- Traversal

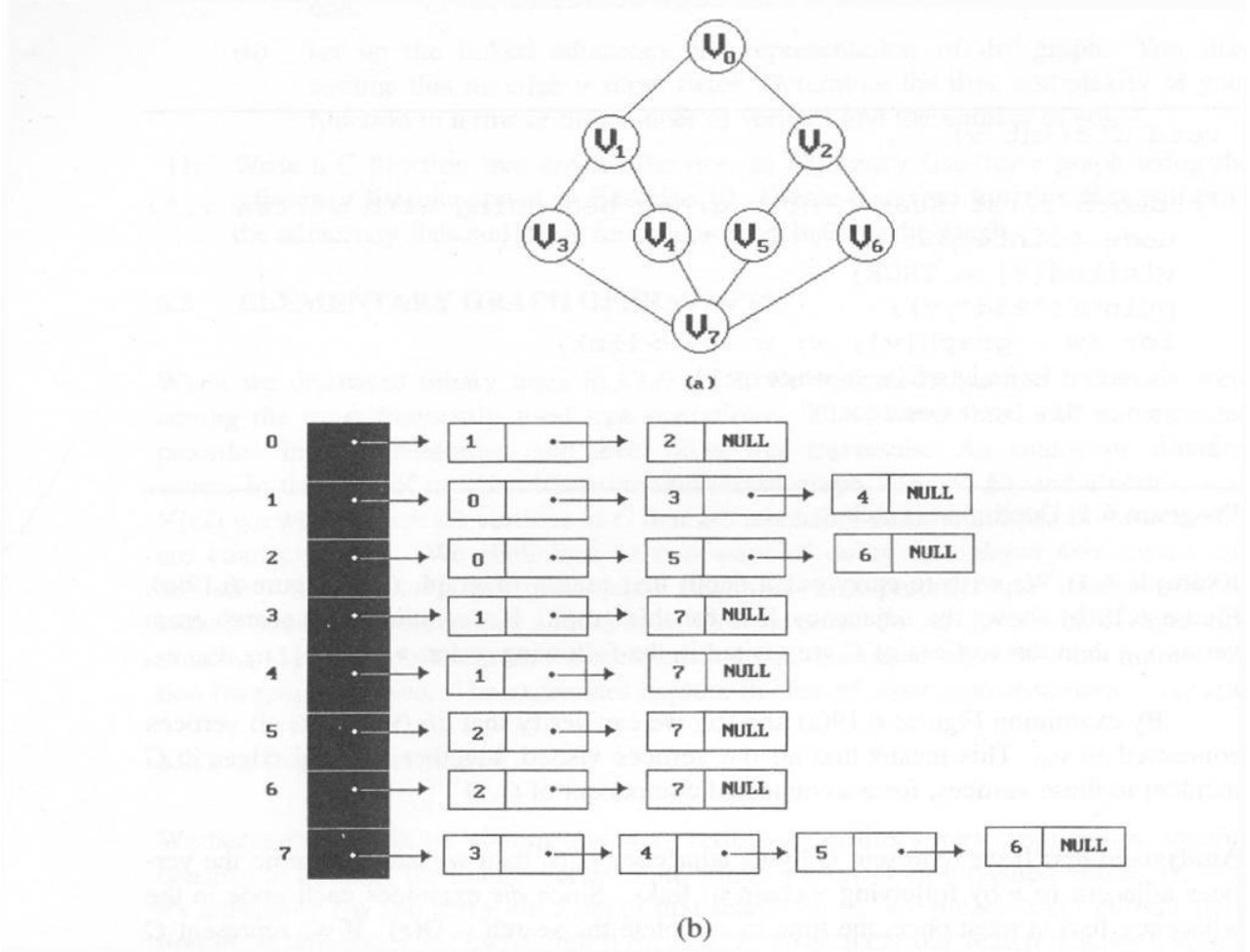
Given G=(V,E) and vertex v, find all w∈V, such that w connects v.
Depth First Search (DFS) preorder tree traversal
Breadth First Search (BFS) level order tree traversal

- Connected Components

- Spanning Trees

Graph G and its adjacency lists

depth first search: v0, v1, v3, v7, v4, v5, v2, v6



breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

Implementing Graphs

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

// A class to store a graph edge
class Edge
{
    int src, dest;

    Edge(int src, int dest)
    {
        this.src = src;
        this.dest = dest;
    }
}
```

Implementing Graphs

```
class Graph
{
    List<List<Integer>> adjList = new ArrayList<>();
     public Graph(List<Edge> edges)
    {
        int n = 0;
        for (Edge e: edges) {
            n = Integer.max(n, Integer.max(e.src, e.dest));
        }
                for (int i = 0; i <= n; i++) {
            adjList.add(i, new ArrayList<>());
        }
                for (Edge current: edges)
        {
            adjList.get(current.src).add(current.dest);

 }
   }
```

Implementing Graphs

```
public static void printGraph(Graph graph)
   {
       int src = 0;
       int n = graph.adjList.size();

       while (src < n)
       {
           // print current vertex and all its neighboring
vertices
           for (int dest: graph.adjList.get(src)) {
               System.out.print("(" + src + " ——> " + dest +
")\t");
           }
           System.out.println();
           src++;
       }
   }
}
```

Implementing Graphs

```
class Main
{
    public static void main (String[] args)
    {

        List<Edge> edges = Arrays.asList(new Edge(0, 1), new Edge(1, 2),
                new Edge(2, 0), new Edge(2, 1), new Edge(3, 2),
                new Edge(4, 5), new Edge(5, 4));

        Graph graph = new Graph(edges);

        Graph.printGraph(graph);
    }
}
```

Thank You!