Data Structure and Algorithm

Apurba Sarkar

IIEST Shibpur

July 19, 2017

Motivation

- Computers do store, retrieve, and process a large amount of data.
- If the data is stored in well organized way on storage media and in computer's memory, it can be accessed quickly for processing and the user is provided fast response.

What/Why Data Structures?

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

Data structure, Algorithm, Function

- A data structure should be seen as a logical concept that must address two fundamental concerns.
  - First, how the data will be stored, and
  - Second, what operations will be performed on it?

Data structure, Algorithm, Function

- Data structure is a scheme for data organization.
- The functional definition of a data structure should be independent of its implementation.
- The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation.
- The implementation part is left to the developers who decide which technology better suits to their project needs.
- For example, a stack ADT is a structure which supports operations such as push() and pop().
  - A stack can be implemented in a number of ways, for example using an array or using a linked list.

Data structure, Algorithm, Function

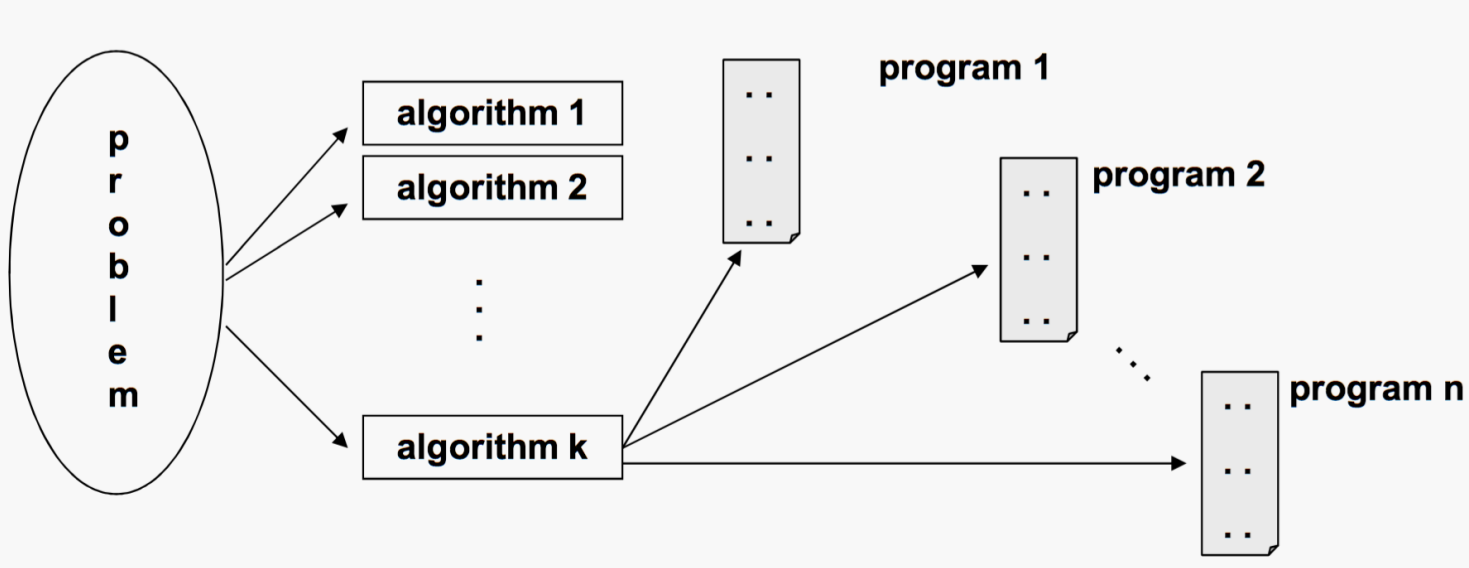
- Along with data structures, in real life, problem solving is done with help of Algorithms.
- An algorithm is a step by step process to solve a problem.
  - finiteness: The algorithm must always terminate after a finite number of steps.
  - definiteness: Each step must be precisely defined;
  - input: An algorithm has zero or more inputs, taken from a specified set of objects.
  - output: An algorithm has one or more outputs, which have a specified relation to the inputs.
  - effectiveness: its operations must be basic enough to be able to be done exactly and in a finite length of time by, for example, somebody using pencil and paper.
- In programming, algorithms are implemented in form of methods or functions or routines.

Data structure, Algorithm, Function

- To get a problem solved we not only want algorithm but also an efficient algorithm.
- One criteria of efficiency is time taken by the algorithm, another could be the memory it takes at run time.
- We can have more than one algorithm for the same problem.
- The best algorithm is the one which has a fine balance between time taken and memory consumption.
- Sadly, Best rarely exists.😕
- We generally give more priority to the time taken by the algorithm rather than the memory it consumes.

Problems vs Algorithms vs Programs

For each problem or class of problems, there may be many different algorithms.
For each algorithm, there may be many different implementations (programs).



Expressing Algorithms

- natural language: usually verbose and ambiguous
- flow charts: avoid most (if not all) issues of ambiguity;
- pseudo-code: also avoids most issues of ambiguity; vaguely resembles common elements of programming languages; no particular agreement on syntax
- programming language: tend to require expressing low-level details that are not necessary for a high-level understanding

Testing Correctness

How do we know whether an algorithm is actually correct?

- First, the logical analysis of the problem we performed in order to design the algorithm should give us confidence that we have identified a valid procedure for finding a solution.
- Second, we can test the algorithm by choosing different sets of input values and checking to see if the resulting solution does, in fact, work.
- BUT... no matter how much testing we do, unless there are only a finite number of possible input values for the algorithm, testing can never prove that the algorithm produces correct results in all cases.

Testing Correctness

How do we know whether an algorithm is actually correct?

- We can attempt to construct a formal, mathematical proof that, if the algorithm is given valid input values then the results obtained from the algorithm must be a solution to the problem
- We should expect that such a proof be provided for every algorithm.
- In the absence of such a proof, we should view the algorithm as nothing more than a heuristic procedure, which may or may not yield the expected results.

Classification of Data Structures

- Data structures can be broadly classified in two categories.
  - Linear data structures: Arrays, linked lists, stacks, and queues
  - Hierarchical data structures: trees, graphs and heaps
- Every data structure has its own strengths, and weaknesses.
- Every data structure specially suits to specific problem depending upon the operations performed and the data organization.

Arrays

- Arrays are by far the most common data structure used to store data.
- Arrays are generally statically implemented data structures by some programming languages.
- The size of this data structure must be known at compile time and cannot be altered at run time.
- Although, few programming languages implement arrays as objects and give the programmer a way to alter the size of them at run time.

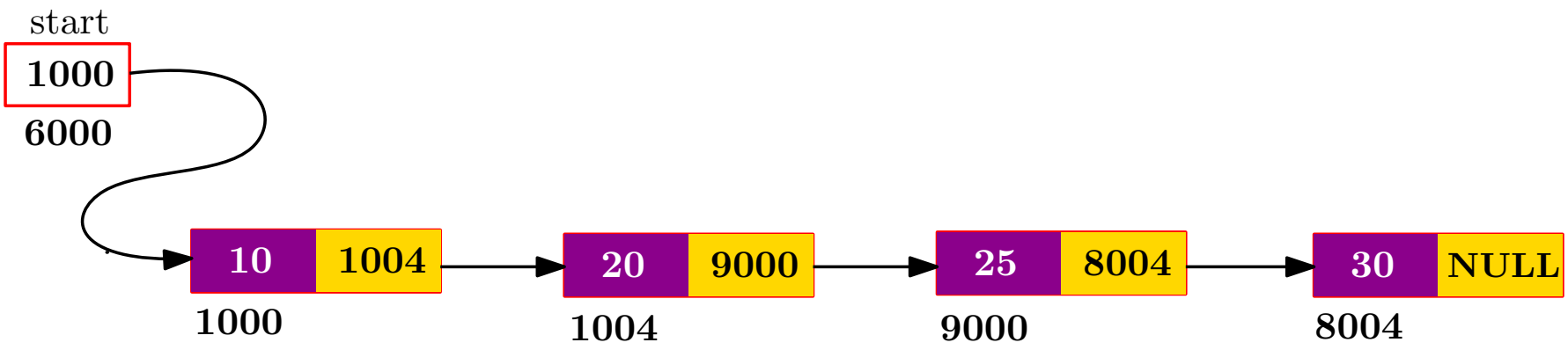
Arrays

- Disadvantages:
  - Inserting an item to an array
  - Deleting an item from the array.
  - The time taken by insert operation depends on how big the array is, and at which position the item is being inserted.
- Advantages:
  - Searching: If the array is unsorted then search operation is also proved costly and takes $O(N)$ time in worst case, where N is size of the array.
  - Searching: If the array is sorted then search performance is improved magically and takes $O(\lg N)$ time in worst case.

Linked List

- Linked list data structure provides better memory management than arrays.
- Linked list is allocated memory at run time, so, there is no waste of memory.
- Advantages:
  - Linked list is proved to be a useful data structure when the number of elements to be stored is not known ahead of time.
- Disadvantages:
  - Performance wise Linked list is slower than array because there is no direct access to linked list elements.
- There are many flavors of Linked list: linear, circular, doubly, and doubly circular.

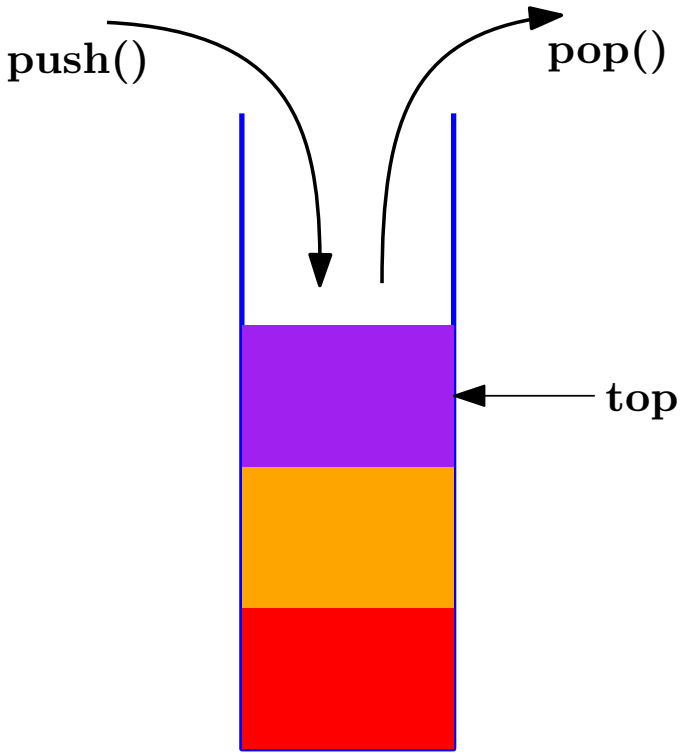
Linked List


start
1000
6000
10
1004
1000
20
9000
1004
25
8004
9000
30
NULL
8004

Stack

- Stack is a last-in-first-out strategy data structure.
- The element stored in last will be removed first.
- Stack has specific but very useful applications;
  - Solving Recursion - recursive calls are placed onto a stack, and removed from there once they are processed.
  - Evaluating post-fix expressions
  - Solving Towers of Hanoi
  - Backtracking
  - Depth-first search
  - Converting a decimal number into a binary number

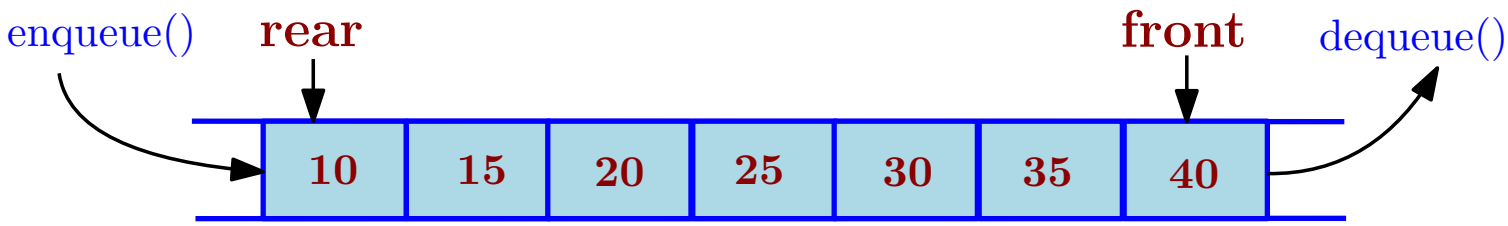
Stack



Queue

- Queue is a first-in-first-out data structure.
- The element that is added to the queue data structure first, will be removed from the queue first.
- Dequeue, priority queue, and circular queue are the variants of queue data structure.
- Queue has the following application uses:
  - Access to shared resources (e.g., printer).
  - Multiprogramming (ready queue).
  - Message queue.

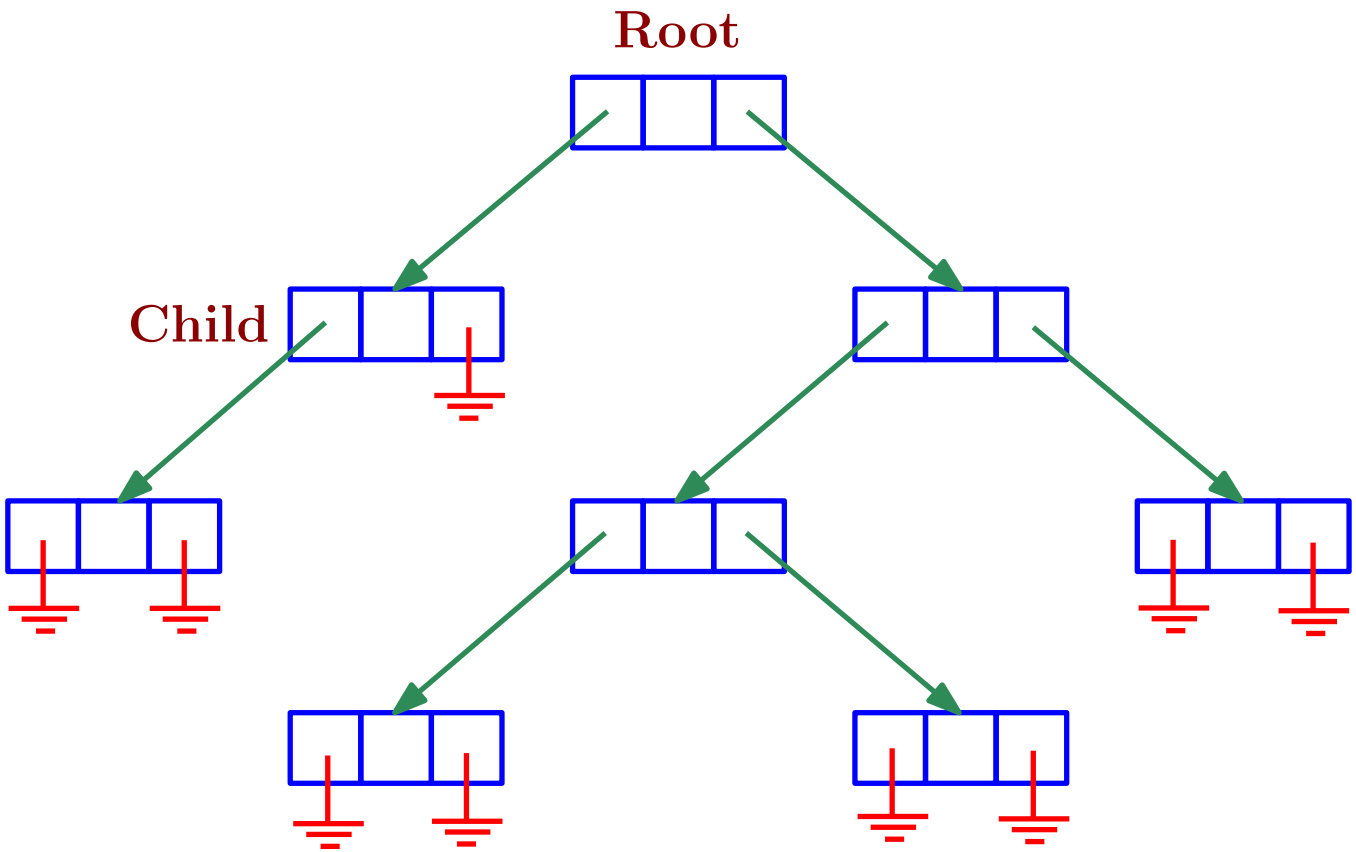
Queue



Trees

- Tree is a hierarchical data structure.
- The very top element of a tree is called the root of the tree.
- Except the root element every element in a tree has a parent, and zero or more children.
- All elements in the left sub-tree come before the root in sorting order, and all those in the right sub-tree come after the root.
- Tree is the most useful data structure when you have hierarchical information to store.
  - directory structure of a file system.
- Some variants of tree data structures are Red-black tree, threaded binary tree, AVL tree, etc.

Trees



Heap

- Heap is a binary tree that stores a collection of keys by satisfying heap property.
- Heap Property:
  - There are two flavours of heap data structure (Max Heap and Mean Heap).
  - For max heap, each node should be greater than or equal to each of its children.
  - For min heap, each node should be smaller than or equal to each of its children.
- Heap data structure is usually used to implement priority queues.

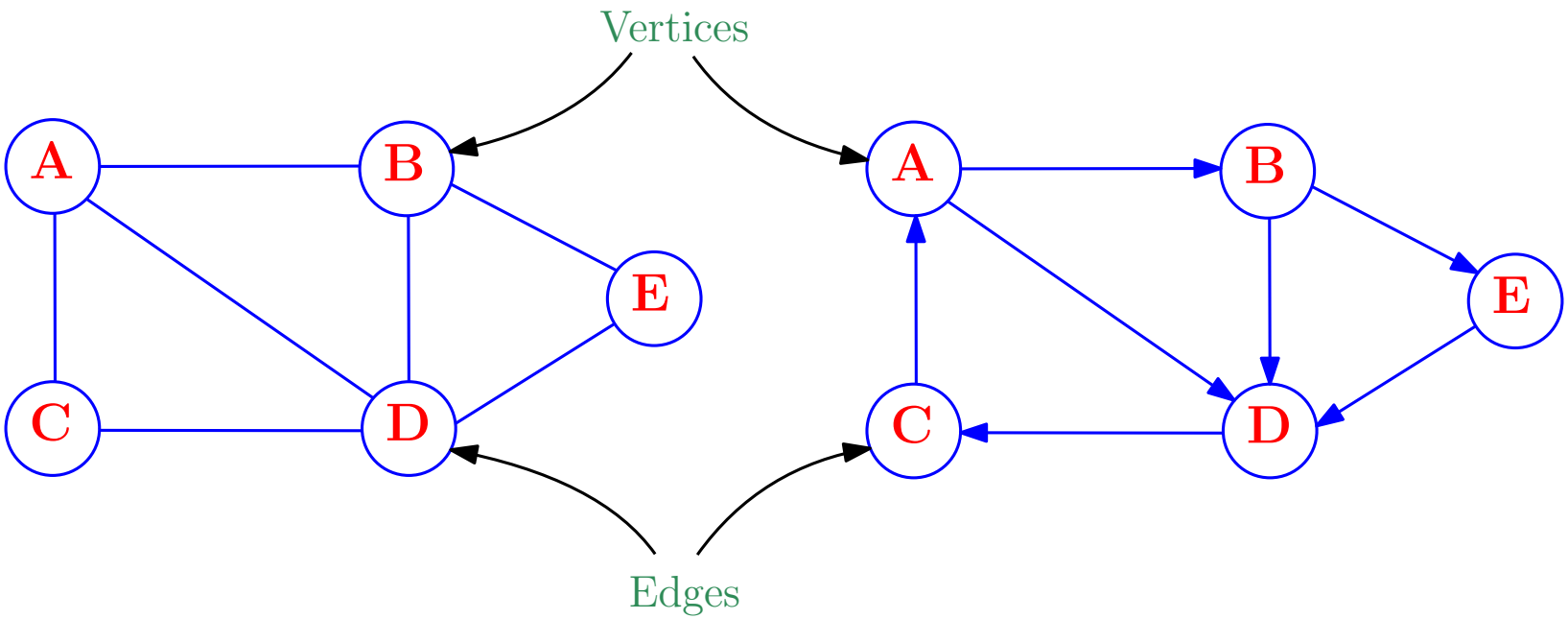
Hash Table

- Hash Table is a data structure that stores data in form of key-element pairs.
- A key is a non-null value which is mapped to an element.
- The element is accessed on the basis of the key associated with it.
- Hash Table is a useful data structure for implementing dictionary.

Graph

- Graph is a networked data structure that connects a collection of nodes called vertices, by connections, called edges.
- An edge can be seen as a path or communication link between two nodes.
- edges can be either directed or undirected.
- In a directed path you can move in one direction only, while in an undirected path the movement is possible in both directions.

Graph



a. Undirected Graph

b. Directed Graph

Abstract Data Types and Data Structures

- Often, these terms are used as synonyms. But its better to think of them this way:
  - An Abstract Data Type (ADT) represents a particular set of behaviours.
    - You can formally define (i.e., using mathematical logic) what an ADT is/does.
    - e.g., a Stack is a list implements a LIFO policy on additions/deletions.
  - A data structure is more concrete. Typically, it is a technique or strategy for implementing an ADT.
    - Use a linked list or an array to implement a stack class.
  - Going one level lower, we get into particulars of programming languages and libraries
    - Use java.lang.Vector or java.util.Stack or a C library from STL.

Abstract Data Types and Data Structures

- Some common ADTs that all trained programmers know about:
  - stack, queue, priority queue, dictionary, sequence, set.
- Some common data structures used to implement those ADTS:
  - array, linked list, hash table (open, closed, circular hashing)
  - trees (binary search trees, heaps, AVL trees, 2-3 trees, tries, red/black trees, B-trees)

Stack Implementation

```
class stack {
    private int maxSize, top = -1;
    public static final int DefaultMaxSize = 100;
    private Object [] store;

    public stack () {
        maxSize = DefaultMaxSize;
        store = new Object [maxSize];
    }

    public stack (int desiredMaxSize) {
        maxSize = desiredMaxSize;
        store = new Object [maxSize];
    }

    public void push (Object newVal) {
        if (top < maxSize -1) {
            top++;
            store[top]=newVal;
        } else {
            System.err.println ("Sorry, stack is full.");
        }
    }
    :
```

Whats Wrong with this implementation?

There's something unsatisfying about this implementation.

- We have to state, in advance, exactly how many elements we will need. [Or just use the provided default of 100].
  - If we go over that number of elements at any one point in time, the stack stops working.
  - If we use a big desiredMaxSize, then we waste space most of the time. [What if we need several stacks?]
- This approach uses a static approach to resource allocation:
  - We have to decide before creating the object what its size will be.
  - We are not allowed to change the size during its lifetime.
- Yes, all resources are ultimately finite. But we would like a little more flexibility and reasonable use of resources.

A Dynamic Approach to Resource Allocation

- What if we could design a stack that uses exactly "enough" storage at any given time?
- Dynamically allocated structures can grow and shrink as needed throughout their lifetime.
- A common opposite forces in CS in static vs. dynamic allocation.

A Linked List Approach

- The stack is a set of nodes linked together.
  - Each nodes has a "value" plus a link variable (a reference to another node).
  - The "value" could be a simple value (e.g., an int), a set of values, or a reference to another object (e.g., an employee record).
- When you want to push a new value:
  1. Create a new node via new.
  2. Set the value of the new node.
  3. Set its link field to point to the previous top node.
  4. Reset top to point to the new node.

Linked Implementation of Stack

```
public interface Stack {
    public abstract void push (Object element);
    public abstract Object pop ();
    public abstract boolean isEmpty ();
    public abstract int size ();
}

class Node {
    // Use package-level visibility.
    Object value;
    Node next;

    public Node (Object value, Node next) {
        this.next = next;
        this.value = value;
    }
}

public class LinkedStack implements Stack {
    private int numElements;
    private Node first;

    public LinkedStack () {
        numElements = 0;
        first = null;
    }

    public void push (Object element) {
        Node newNode = new Node (element, first);
        first = newNode;
        numElements++;
    }
```

Linked Implementation of Stack

```
// Still inside LinkedStack ...

public Object pop () {
    Object returnVal;
    if (numElements > 0) {
        returnVal = first.value;
        first = first.next;
        numElements--;
    } else {
        System.out.println ("Sorry, stack is empty.");
        returnVal = null;
    }
    return returnVal;
}

public boolean isEmpty () {
    return numElements == 0;
}

public int size () {
    return numElements;
}
```

Linked Implementation of Stack

```
    // Still inside LinkedStack ...

    public static void main (String [] args) {
        LinkedStack s1 = new LinkedStack();
        s1.push ("hello");
        s1.push ("there");
        s1.push ("world");
        System.out.println ("There are now " + s1.size()
                + " elements.");
        String s;
        s = (String) s1.pop();
        System.out.println ("Popped: " + s);
        s = (String) s1.pop();
        System.out.println ("Popped: " + s);
        s = (String) s1.pop();
        System.out.println ("Popped: " + s);
        s = (String) s1.pop();
        System.out.println ("Should be empty now.");
    }
}
```

Array Implementation of Stack

Array implementation restated slightly as implementing the Stack interface:

```
public class ArrayStack implements Stack {
    private int maxSize;
    public static final int DefaultMaxSize = 100;
    private int top = -1;
    private Object [] store;

    public ArrayStack () {
        maxSize = DefaultMaxSize;
        store = new Object [maxSize];
    }

    public ArrayStack (int desiredMaxSize) {
        maxSize = desiredMaxSize;
        store = new Object [maxSize];
    }

    public void push (Object newVal) {
        if (top < maxSize -1) {
            top++;
            store[top]=newVal;
        } else {
            System.err.println ("Sorry, stack is full.");
        }
    }
```

Array Implementation of Stack

```
    // Still inside ArrayStack ...

    public Object pop () {
        Object ans = null;
        if (top >= 0) {
            ans = store[top];
            top--;
        } else {
            System.err.println ("Sorry, stack is empty.");
        }
        return ans;
    }

    public boolean isEmpty () {
        return size() == 0;
    }

    public int size () {
        return top + 1;
    }
}
```

Thank You

for your attention.