Data Structures

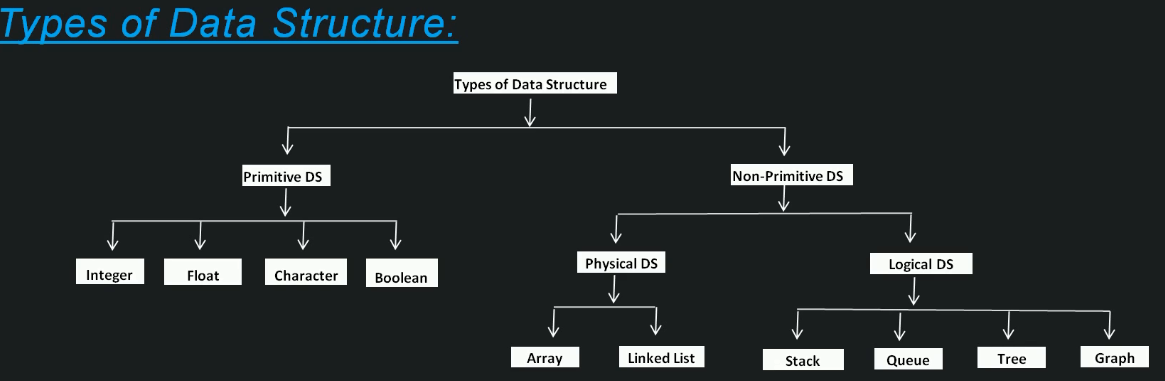
And

Algorithms

What is Data Structure?

Data Structure is way to ‘organize data’ that enables to be processed in an efficient time.

Types of Data Structures



What is Algorithm?

Algorithm is a set of rules to be followed to solve a problem

Algorithm Runtime Analysis

Algorithm runtime analysis is a study about a given algorithm runtime, by identifying its behavior as the input size for the algorithm increases .In in lay man’s language ‘how much time will the given algorithm will take to run’

Notations for Algorithm Runtime Analysis

There are 3 notations for ‘Algorithm runtime analysis ’

* Omega(Ω) – Best Case
  + This notation gives the tighter lower bound of a given algorithm
  + For any given input, running time of a given algorithm will not be less than the given time
  + This is the Best Case runtime of a given algorithm
* Big-o (O) – Worst Case
  + This notation gives the tighter upper bound of a given algorithm
  + For any given input, running time of given algorithm will not be more than the given time
  + This is the Worst Case runtime of a given algorithm
* Theta (θ) – Average Case
  + This notation decides whether upper bound and lower bound of a given algorithm are same or not
  + For any given input, running time of a given algorithm will on an average be equal to the given time
  + This is the Average Case runtime of given algorithm

Array

**Trapping Rain Water**

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

  
The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. **Thanks Marcos** for contributing this image!

**Example:**

**Input:** [0,1,0,2,1,0,1,3,2,1,2,1]

**Output:** 6

class Solution {

public int trap(int[] height) {

int totalValue = 0;

if(height == null || height.length ==0){

return totalValue;

}

int leftHighest[] = new int[height.length+1];

leftHighest[0] =0;

for(int i=0;i<height.length;i++){

leftHighest[i+1] = Math.max(leftHighest[i],height[i]);

}

int rightHighest = 0;

for(int i=height.length-1;i>=0;i--){

rightHighest = Math.max(rightHighest,height[i]);

int computeMinHeight = Math.min(leftHighest[i],rightHighest);

int currentTrap = computeMinHeight- height[i];

currentTrap = currentTrap >0 ? currentTrap : 0;

totalValue = totalValue + currentTrap;

}

return totalValue;

}

}

**Largest Rectangle in Histogram**

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

  
Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].

  
The largest rectangle is shown in the shaded area, which has area = 10 unit.

**Example:**

**Input:** [2,1,5,6,2,3]

**Output:** 10

class Solution {

public int largestRectangleArea(int[] heights) {

int largetRectArea = 0;

if(heights == null || heights.length ==0 ){

return largetRectArea;

}

Stack<Integer> stack = new Stack<Integer>();

stack.push(-1);

int currentBarHeight = 0;

for(int i=0;i<heights.length;i++){

currentBarHeight = heights[i];

while(stack.peek() != -1 && currentBarHeight < heights[stack.peek()]){

int height = heights[stack.pop()];

int width = i-stack.peek()-1;

int rectArea = height \* width;

largetRectArea = Math.max(rectArea,largetRectArea);

}

stack.push(i);

}

while(stack.peek() != -1){

int height = heights[stack.pop()];

int width = heights.length-stack.peek()-1;

int rectArea = height \* width;

largetRectArea = Math.max(rectArea,largetRectArea);

}

return largetRectArea;

}

}

Linked List

Stack

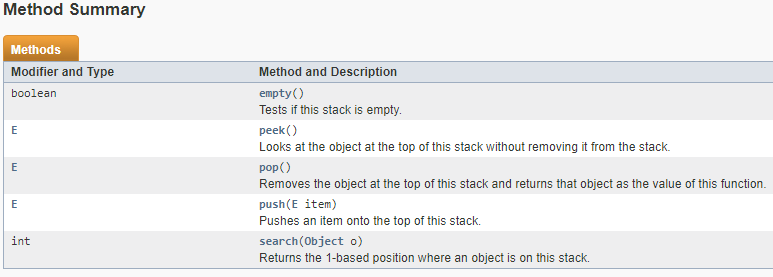
A stack is a basic data structure that can be **logically** thought as linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack.

The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it. This structure is used all throughout programming.

The basic implementation of a stack is also called a **“Last In First Out**” structure; however there are different variations of stack implementations.

There are basically three operations that can be performed on stacks. They are:

* inserting (“pushing”) an item into a stack
* deleting (“popping”) an item from the stack
* displaying the contents of the top item of the stack (“peeking”)



The Java Stack class, java.util.Stack, is a classical stack data structure. You can push elements to the top of a Java Stack and pop them again, meaning read and remove the elements from the top of the stack.

The Java Stack class actually implements the [**Java List**](http://tutorials.jenkov.com/java-collections/list.html) interface, but you rarely use a Stack as a List - except perhaps if you need to inspect all elements currently stored on the stack

Stack stack = new Stack();

Stack stack = new Stack();

stack.push("1");

String topElement = stack.peek();

Stack stack = new Stack();

stack.push("1");

stack.push("2");

stack.push("3");

**int index = stack.search("3");** //index = 1

Stack stack = new Stack();

stack.push("123");

stack.push("456");

stack.push("789");

Iterator iterator = stack.iterator();

while(iterator.hasNext()){

Object value = iterator.next();

}

Stack stack = new Stack();

stack.push("A");

stack.push("B");

stack.push("C");

Stream stream = stack.stream();

stream.forEach((element) -> {

System.out.println(element); // print element

});

Queue

A queue is an abstract data type or a linear data structure, in which the first element is **inserted** from one end (the “**tail**”), and the **deletion** of existing element takes place from the other end (the “**head**”). A queue is a “**First In First Ou**t” structure. "First In First Out" means that elements put in the queue first will come out first, and elements put in the queue last will come out last.

An example of a queue are lines of people waiting. The first person in the line goes first, and the last person in the line goes last.

The process of adding an element to a queue is called “**enqueuing**” and the process of removing an element from a queue is called “**dequeuing**”

The Java Queue interface, java.util.Queue represents a data structure designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue. This is similar to how a queue in a supermarket works.

The Java Queue interface is a subtype of the [**Java Collection**](http://tutorials.jenkov.com/java-collections/collection.html) interface. It represents an ordered sequence of objects just like a [**Java List**](http://tutorials.jenkov.com/java-collections/list.html), but its intended use is slightly different.

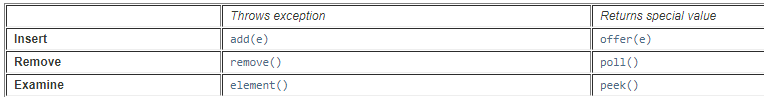
Being a Collection subtype all methods in the Collection interface are also available in the Queue interface.

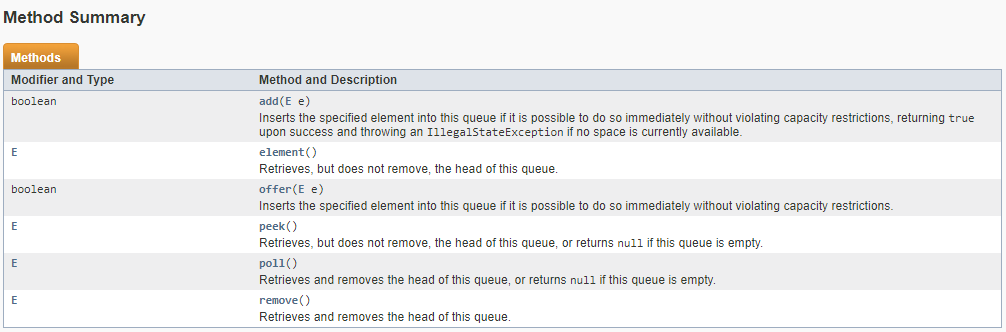
Since Queue is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Queue implementations in the Java Collections API:

* java.util.LinkedList
* java.util.PriorityQueue

LinkedList is a pretty standard queue implementation.

PriorityQueue stores its elements internally according to their natural order (if they implement Comparable), or according to a Comparator passed to the PriorityQueue.





Queue queueA = new LinkedList();

Queue queueB = new PriorityQueue();

Tree

Graphs

Hashing

Sorting

Magic Framework

Greedy Algorithm

Divide & Conquer Algorithms

Dynamic Programming