# Array

## Properties

* Linear data structure
* Contiguous and allows indexed access to its contents
* Fixed size

# LinkedList

## General

* Linear data structure.
* Elements are not stored at contiguous locations
* The elements are linked using pointers.
* + Dynamic Size
* + Easy insertion/Deletion
* - Random Access
* - Extra storage requirement for pointers
* If the problem demands you to use no extra space, then you :
  + CANNOT CREATE NEW NODES
  + CANNOT USE THE RECURSION STACK (with the exception of TAIL RECURSION)

### Types

* Circular
* Doubly

### Sorting

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

# Stack

## General

* LIFO
* Unfair way of processing in real life.
* Linear data structure which follows a particular order in which the operations are performed.
* The order may be LIFO(Last In First Out).
* Mainly the following three basic operations are performed in the stack:
* Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
* Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
* Peek: Get the topmost item.
* Stack of plates. The plate which is at the top is the first one to be removed, i.e.
* the plate which has been placed at the bottom-most position remains in the stack for the longest period of time.
* So, it can be simply seen to follow LIFO order.
* Implementation:
* There are two ways to implement a stack:
  + Using array
  + Using linked list

# Queue

## General

* FIFO
* Fair way to process items
* The order is First In First Out (FIFO).
* A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.
* The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations

* Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
* Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
* Front: Get the front item from queue.
* Rear: Get the last item from queue.

Applications

* Queue is used when things don’t have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search.
* 1) When a resource is shared among multiple consumers.

Examples include CPU scheduling, Disk Scheduling.

* 2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes.

Examples include IO Buffers, pipes, file IO, etc

# Tree

## Purpose

To store hierarchical data

The difference between a graph and a tree is:

* For a tree there is only one sequence of edges between any two vertices; no such guaranty is made for a graph
* Directed acyclic graph is a tree

## Performance

Access/Search time : LL > Trees > Arrays

Insertion/Deletion : Arrays > Trees > LL

+ No upper limit on the number of nodes

## Applications

Maintain hierarchical data

Decision maker

Routing algorithms

Easy to search info

## Binary Tree properties

Maximum number of nodes at any level L : 2^(L-1)

Maximum number of nodes in a tree of height H : 2^H - 1

Minimum possible height of a tree w/ N nodes : log(N + 1) // Base 2

Levels in a tree w/ L leaves : logL + 1

Number of leaf nodes = number of internal nodes (w/ 2 children) + 1

## Types

Complete Binary Tree : All levels are completely filled except last level

Level L has 2^(L-1) nodes

Perfect (Full) Binary Tree : All nodes have 2 children and leaves are at same level

A full tree with L levels has 2^L-1 nodes

BST : left < parent < right. Duplicate keys are not allowed

Minheap : Complete binary tree with parent <= left and right

Maxheap : Complete binary tree with parent >= left and right

## Properties

### Diameter

The number of nodes on the longest path between two leaves in the tree.

aka WIDTH

### Depth

The number of edges from the node to the tree's root node. A root node will have a depth of 0.

### Height

The number of edges on the longest path from the node to a leaf. A leaf node will have a height of 0.

HEIGHT OF A TREE = HEIGHT OF THE ROOT NODE = DEPTH OF THE DEEPEST NODE

DEPTH : Mark the first level as 0 and count downwards

HEIGHT : Mark the last level as 0 and count upwards

LEVEL : Mark the root as level 1 and count downwards

### Size

Number of nodes in the tree

Number of unlabeled binary trees : T(n) = (2n)! / (n+1)!n!

Number of labeled binary trees : T(n) = (2n)! / (n+1)!

## Traversals

Traversals are named based on the position of the ROOT

### Depth First Search

INorder, so ROOT comes in the middle (IN) LEFT -> ROOT -> RIGHT

POSTorder, so ROOT comes in the end (POST) LEFT -> RIGHT -> ROOT

PREorder, so ROOT comes in the beginning (PRE) ROOT -> LEFT -> RIGHT

### Breadth First Search

Level Order : Prints level by level

The following combinations can uniquely identify a tree :

Inorder and Preorder.

Inorder and Postorder.

Inorder and Level-order.

And following do not.

Postorder and Preorder.

Preorder and Level-order.

Postorder and Level-order.

INORDER SUCCESSOR : The next node to be visited during an inorder traversal.

Inorder traversal of a BST sorts the nodes of the tree

# Graph

## General

* Complete graph with n vertices can have n(n-1)/2 edges. Performance, hence will peak at O(n^2)

### Representations

#### Adjacency Matrix

* int[][] graph

#### Adjacency List

* List<Edge> graph = new List[vertices]
* This is an array of lists.
* Array is of size = # vertices. Ergo, we have a list for each vertex.
* Edge is an object that holds the destination vertex and the weight that make up an edge
* e.g. graph[0] = Arrays.asList( new Edge[ ] { new Edge(1, 40), new Edge(2, 30) } )
  + The above statement adds two edges, 0 to 1 and 0 to 2 of weights 40 and 30 resp.

### Traversals

Iteration + Queue => BFS

Recursion or Stack => DFS

Dfs goes as deep as possible then scans the next node and goes deep again

Requires less memory as you are not storing all the child pointers at each level

Bfs scans all nodes on a level and moves to the second level

e.g. in a family tree, to find the person who's still alive(will most likely be at the bottom of the tree) use Dfs and To find the person who's long gone, use bfs

#### Depth First Search

* Uses a STACK to go down and scan the deepest item first
* Just like a stack operation (LIFO), the last adjacent vertex of the currently visited node is visited first
* Analogy : You are doing a task and immediately another on comes up. You freeze the one you were doing and move on to the new one. If another one comes, that is worked up on instantly.
* Train of thoughts!!
* Performance is O( V + E )
  + All the edges of all the vertices need to be visited
* The typical DFS algorithm using stack will traverse all vertices as long as a connection exists.
* If there exists more than one connected component, then DFS has to be run by taking each vertex as the starting vertex, until all of them are visited
* All the tree traversals are DFS

##### Applications

* Cycle detection : If a visited vertex is revisited, a cycle is present in the graph
* Path finding : Useful to detect a path between two vertices, u and v. Perform DFS. Once v is seen on top of the stack, print the stack contents to get the path
* Topological Sorting
* Strongly connected components : A strongly connected graph has a path from every vertex to every other vertex
* Solving Mazes
* <http://www.geeksforgeeks.org/applications-of-depth-first-search/>

#### Breadth First Search

* Uses a QUEUE to perform a broader search on the graph
* A vertex is visited, its adjacent vertices are enqueued and processed individually
* Just like a Queue operation (FIFO)
* This is the Level order traversal of a tree
* Performance is O( V + E )
* One important observation about BFS is, the path used in BFS always has least number of edges between any two vertices.
* So if all edges are of same weight, we can use BFS to find the shortest path.

##### Applications

* For an unweighted graph (or for a graph with equal weights, BFS gives the shortest path)
* To find neighboring items, like
* Cycle detection in an undirected graph
* <http://www.geeksforgeeks.org/applications-of-breadth-first-traversal/>

### Cycle Detection

* Undirected Graphs
  + DFS and BFS work like a charm in detecting cycles here
  + Ideally if a visited node is visited again, it’s a cycle
* Directed Graphs
  + BFS fails here as it might mistakingly report a cycle when there is’nt any
  + A -> B, A -> C, C -> B does not a have a cycle. But BFS will report a cycle
  + This is because for directed graphs, a visited node may be visited again, but by chosing a different path. So along with the visited nodes, the stacked nodes should be remembered as well. Hence DFS is preferred.

# Sorting and Searching

Visualizations : https://visualgo.net/sorting

## Binary\_Search

* Sort the array, get the mid
* If the mid value is greater than item, move left
* If the mid value is less than the item, move right
* It they are equal, return mid or boolean
* Use the shift operators for getting the middle.
  + >>> and >> are right shift operators
  + >> is arithmetic hence the sign of the number is preserved
  + >>> is logical (unsigned), so the sign is ignored
* This is due to the chance of overflow when the array size is so large (MAX VALUE). Doing

low + high will cause overflow and can give a negative result!, hence incorrect mid value

* TIME : O(logn)

private boolean binarySearchRecursive**(**int**[]** array**,** int low**,** int high**,** int item**){**

**if(**low **<=** high**)** **{**

int mid **=** **(**low **+** high**)>>**1**;**

**if(**array**[**mid**]** **<** item**)**

**return** **false** **||** binarySearchRecursive**(**array**,** mid**+**1**,** high**,** item**);**

**else** **if(**array**[**mid**]** **>** item**)**

**return** **false** **||** binarySearchRecursive**(**array**,** low**,** mid**-**1**,** item**);**

**return** **true;**

**}**

**return** **false;**

**}**

private int binarySearchIterative**(**int**[]** array**,**int size**,**int item**){**

int low **=** 0**;**

int high **=** size**-**1**;**

int mid**;**

**while(**low **<=** high**){**

mid **=** **(**low **+** high**)>>>**1**;**

**if(**array**[**mid**]** **==** item**)**

**return** mid**;**

**else** **if(**array**[**mid**]** **<** item**)**

low **=** mid**+**1**;**

**else**

high **=** mid**-**1**;**

**}**

**return** **-**1**;**

**}**

## Bubble\_Sort

* Iterate the array (i loop)
* For each element, j loop will compare it with adjacent element and swap if needed
* After i'th iteration, i'th largest will be at the end of the array, so the rest of the iterations can skip this element, hence n-i
* If no swaps were made, meaning the array got sorted, end the iterations
* size - i - 1 is the index of the last unsorted element
* INVARIANT : n-i ... n is always sorted
* BEST TIME : O(n) [ Array is already sorted]
* WORST TIME : O(n^2) [ Random array, reverse sorted array ]
* AVERAGE TIME : O(n^2)
* SPACE : O(1)

public void sort**(**int**[]** array**){**

int size **=** array**.**length**;**

boolean swaps **=** **true;**

**for(**int i **=** 0**;** i **<** size **&&** swaps**;** i**++){**

swaps **=** **false;**

**for(**int j **=** 0**;** j **<** size**-**i**-**1**;** j**++)**

**if(**array**[**j**]** **>** array**[**j**+**1**])**

swaps **=** swapUsingXOR**(**array**,** j**,** j**+**1**);**

**}**

**}**

## Heap\_Sort

* Sort method
  + Each parent node at position i in a heap has children at positions 2\*i and 2\*i+1,
  + only if these positions do not exceed the length of the array.
  + Since the sink() algorithm swaps parents with their children, only length/2 iterations are needed.
  + i.e. Element at n/2 will be the parent to the last elements of the array
  + Moreover, the heap is constructed from bottom to top. Therefore, the iteration starts at n/2 going up, i.e. towards position 0.
  + So, starting from n/2 sink or heapify all elements till 0
  + As long as the array size is >=0, swap the first and last elements, reduce the overall size by 1 and sink the first element
* Sink method
  + Since we are sinking element at i, it's children can be at 2\*i and 2\*i+1, hence we repeat this as long as 2\*i<=n
  + Find out if any of the children is larger than the parent -
  + Here we are comparing the left and right child, we pick the child with the larger value
  + Then compare the chosen child with the papa and swap if needed
  + If none are larger than parent, break (coz he's in the right spot)
  + If either is larger, swap it with the parent and sink a level down (i=j) to verify the grandchildren
* BEST TIME : O(nlogn)
* WORST TIME : O(nlogn)
* AVERAGE TIME : O(nlogn)
* SPACE : O(1)

private void heapSort**(**int**[]** array**){**

int arraySize **=** array**.**length**-**1**;**

**for(**int i **=** arraySize**/**2**;** i **>=** 0**;** i**--)**

sink**(**array**,** i**,** arraySize**);**

**while(**arraySize **>=** 0**){**

swap**(**array**,** 0**,** arraySize**);**

sink**(**array**,** 0**,** **--**arraySize**);**

**}**

**}**

private void sink**(**int**[]** array**,** int parentIndex**,** int heapSize**)** **{**

**while(**2**\***parentIndex **<=** heapSize**){**

int childIndex **=** 2**\***parentIndex**;**

**if(**childIndex **<** heapSize **&&** array**[**childIndex**]** **<** array**[**childIndex**+**1**])**

childIndex**++;**

**if(**array**[**parentIndex**]** **>=** array**[**childIndex**])**

**break;**

swap**(**array**,** parentIndex**,** childIndex**);**

parentIndex **=** childIndex**;**

**}**

**}**

## Insertion\_Sort

* V1 uses SWAPS
  + Iterate through the array (i loop)
  + Ignore the first element as it is assumed to be sorted
  + For each element, scan right to left
  + Swap if previous element is larger than current element
  + If there're no larger elements to the left of the current element, no swaps are needed (that element is already in the right spot)
* V2 uses COPYING
  + Iterate through the array (i loop)
  + Ignore the first element as it is assumed to be sorted
  + Save the current item (This item needs to be inserted in its proper location in the sorted left sub array)
  + For each element, scan right to left ( j loop)
  + If previous element is larger than item, copy and replace current element
  + When j loop ends, replace element a[j] with item
* INVARIANT : a[1....i] is sorted
* BEST TIME : O(n) [Already sorted / Almost sorted]
* WORST TIME : O(n^2)
* AVERAGE TIME : O(n^2) [ Random array, reverse sorted array ]
* SPACE : O(1)

private void insertionSortV1**(**int**[]** array**){**

**for(**int i **=** 1**;** i **<** array**.**length**;** i**++){**

**for(**int j **=** i**;** j **>** 0**;** j**--)**

**if(**array**[**j**-**1**]** **>** array**[**j**])**

swap**(**array**,** j**,** j**-**1**);**

**else**

**break;**

**}**

**}**

private void insertionSortV2**(**int**[]** array**){**

**for(**int i **=** 1**;** i **<** array**.**length**;** i**++){**

int item **=** array**[**i**];**

int j **=** i**;**

**for(** **;** j **>** 0**;** j**--)**

**if(**array**[**j**-**1**]** **>** item**)**

array**[**j**]** **=** array**[**j**-**1**];**

**else**

**break;**

array**[**j**]** **=** item**;**

**}**

**}**

## Knuth\_Shuffle

Pick an element, generate a random number between 0 and the index of this element, insert this element at the generated index

private static void knuthShuffle**(**int**[]** array**)** **{**

**for(**int i **=** 0**;** i **<** array**.**length**;** i**++)**

swap**(**array**,** i**,** **new** Random**().**nextInt**(**i**+**1**));**

**}**

**}**

## Merge\_Sort

* MergeSort method
  + Stop recursion if high<=low (duh!)
  + Calculate mid use mid = low + (high-low)/2 or (low + high)>>1
  + Call mergeSort for sub arrays [low,mid] and [mid+1,high]
  + Merge the sub arrays
* Merge method
  + Prepare right and left arrays
  + Left array is from low to mid, so size = (mid-low)+1
  + Right array is from mid+1 to high, so size = (high-(mid+1))+1 => high-mid
  + Fill left and right arrays with values from the original array
  + For left array, the values start from low+i index of original array are copied (Coz left array starts at low)
  + For right array, values start from (mid+1)+i (Coz right array starts at mid)
  + Iterate both arrays and copy the least value onto the actual array
  + Fill remaining values onto the array
* Average case : O(nlogn)
* Worst case : O(nlogn)
* Best case : O(nlogn)
* Auxiliary Space : O(n)

private void mergeSort**(**int**[]** array**,** int low**,** int high**){**

**if(**low **<** high**){**

int middle **=** **(**low **+** high**)>>**1**;**

mergeSort**(**array**,** low**,** middle**);**

mergeSort**(**array**,** middle**+**1**,** high**);**

merge**(**array**,** low**,** middle**,** high**);**

**}**

**}**

private static void merge**(**int**[]** array**,** int low**,** int middle**,** int high**)** **{**

int sizeOfLeft **=** middle**-**low**+**1**;**

int**[]** leftArray **=** **new** int**[**sizeOfLeft**];**

System**.**arraycopy**(**array**,** low**,** leftArray**,** 0**,** sizeOfLeft**);**

int sizeOfRight **=** high**-**middle**;**

int**[]** rightArray **=** **new** int**[**sizeOfRight**];**

System**.**arraycopy**(**array**,** middle**+**1**,** rightArray**,** 0**,** sizeOfRight**);**

int i **=** 0**;**

int j **=** 0**;**

int k **=** low**;**

**while(**i **<** sizeOfLeft **&&** j **<** sizeOfRight**)**

array**[**k**++]** **=** leftArray**[**i**]** **<=** rightArray**[**j**]?** leftArray**[**i**++]** **:** rightArray**[**j**++];**

**while(**i **<** sizeOfLeft**)**

array**[**k**++]** **=** leftArray**[**i**++];**

**while(**j **<** sizeOfRight**)**

array**[**k**++]** **=** rightArray**[**j**++];**

**}**

## Quick\_Sort

* Take an item as pivot. After one "partitioning", the pivot will be in the right spot, so all items to
* its left are < and right are > it.
* Hoare partition deals with this logic. First item of the partition is chosen as pivot.
* Scan right until a value > pivot is seen.
* Scan left until a value < pivot is seen.
* If the indices cross, return rightindex.
* Else swap the items at these indices
* Best case happens when the partitioning is even

private void quickSort**(**int**[]** array**,**int low**,**int high**){**

**if(**low **<** high**){**

int middle **=** partition\_Hoare**(**array**,** low**,** high**);**

quickSort**(**array**,** low**,** middle**);**

quickSort**(**array**,** middle**+**1**,** high**);**

**}**

**}**

private int partition\_Hoare**(**int**[]** array**,** int low**,** int high**)** **{**

int pivot **=** array**[**low**];**

int leftIndex **=** low **-** 1**;**

int rightIndex **=** high **+** 1**;**

**while(true){**

**while(**array**[++**leftIndex**]** **<** pivot**);**

**while(**array**[--**rightIndex**]** **>** pivot**);**

**if(**leftIndex **>=** rightIndex**)**

**return** rightIndex**;**

**else**

swap**(**array**,** leftIndex**,** rightIndex**);**

**}**

**}**

private int partition\_Lomuto**(**int**[]** array**,** int low**,** int high**)** **{**

int pivot **=** array**[**high**];**

int i **=** low**-**1**;**

**for(**int j**=**low**;**j **<** high**;**j**++)**

**if(**array**[**j**]** **<=**pivot**){**

i**++;**

swap**(**array**,**i**,**j**);**

**}**

swap**(**array**,**i**+**1**,**high**);**

**return** i**+**1**;**

**}**

## Selection\_Sort

* Iterate through the array (i loop)
* Assume i-th element is the least (min)
* Iterate through the rest of the array (j loop and it starts at i)
* If any of the remaining values is less that the assumed least value, update min
* After iterating through the rest of the array (j loop ends),swap the value at min with the value at i
* INVARIANT - a[min] is the least element among a[i...n]
* INVARIANT - a[1...i] is sorted
* Best case : O(n2)
* Worst case : O(n2)
* Average case : O(n2)

public static void selectionSort**(**int**[]** array**){**

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

int min **=** i**;**

**for(**int j **=** i**;** j **<** array**.**length**;** j**++)**

min **=** array**[**j**]** **<** array**[**min**]?** j **:** min**;**

swap**(**array**,** i**,** min**);**

**}**

**}**

## Shell\_Sort

* h sort the array
* h=1 => Insertion sort
* Value of h is a value in the sequence 3x+1 such that it is less than length of array/3
* Decrement by h

public static void shellSort**(**int**[]** a**){**

int h **=** 1**;**

**while(**h**<**a**.**length**/**3**)**

h**=**3**\***h**+**1**;**

**while(**h**>=**1**){**

**for(**int i**=**h**;**i**<**a**.**length**;**i**++){**

**for(**int j**=**i**;**j**>=**h**;**j**-=**h**)**

**if(**a**[**j**-**1**]>**a**[**j**])**

swap**(**a**,** j**,** j**-**h**);**

**}**

h**/=**3**;**

**}**

**}**

# Dynamic Programming

## General

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again

### Overlapping Subproblems

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn’t have common subproblems. If we take example of recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

|  |
| --- |
| /\* simple recursive program for Fibonacci numbers \*/  int fib(int n)  {     if ( n <= 1 )        return n;     return fib(n-1) + fib(n-2);  } |

Recursion tree for execution of *fib(5)*

fib(5)

/ \

fib(4) fib(3)

/ \ / \

fib(3) fib(2) fib(2) fib(1)

/ \ / \ / \

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

/ \

fib(1) fib(0)

We can see that the function f(3) is being called 2 times. If we would have stored the value of f(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:  
a) Memoization (Top Down)  
b) Tabulation (Bottom Up)

**a) Memoization (Top Down):**The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

/\* memoized function for nth Fibonacci number \*/

int fib(int n)

{

   if (lookup[n] == NIL)

   {

      if (n <= 1)

         lookup[n] = n;

      else

         lookup[n] = fib(n-1) + fib(n-2);

   }

   return lookup[n];

}

**b) Tabulation (Bottom Up):**The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottom-up.

/\* tabulated function for nth Fibonacci number \*/

int fib(int n)

{

  int f[n+1];

  int i;

  f[0] = 0;   f[1] = 1;

  for (i = 2; i <= n; i++)

      f[i] = f[i-1] + f[i-2];

  return f[n];

}

Both Tabulated and Memoized store the solutions of subproblems. In Memoized version, table is filled on demand while in Tabulated version, starting from the first entry, all entries are filled one by one. Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version

### Optimal Substructure

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

For example, the Shortest Path problem has following optimal substructure property:  
If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn’t have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the CLRS book. There are two longest paths from q to t: q→r→t and q→s→t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.

