### Algorithms and Data Structures

# BIG O

* When the number of problem statements gets halved every time, it will have O(log n) runtime
* 2^0 \* 2^1 \* 2^2 \* 2^3 …. \* 2^n = 2^(n+1) – 1
* Sum of 1 through n : 1 + 2 + 3 + 4 +….+ (n-1) + n = n(n-1)/2

# Data Structures

## Tree

A tree data structure can be defined [recursively](https://en.wikipedia.org/wiki/Recursion) (locally) as a collection of [nodes](https://en.wikipedia.org/wiki/Node_(computer_science)) (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

A tree is a data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

#### Terminology used in trees

**Root:**The top node in a tree.

**Child :** A node directly connected to another node when moving away from the Root.

**Parent:**The converse notion of a *child*.

**Siblings:**A group of nodes with the same parent.

**Descendant:**A node reachable by repeated proceeding from parent to child.

**Ancestor:**A node reachable by repeated proceeding from child to parent.

**Leaf(less commonly called External node):**A node with no children.

**Branch Internal node** A node with at least one child.

**Degree:**The number of sub trees of a node.

**Edge:** The connection between one node and another.

**Path:**A sequence of nodes and edges connecting a node with a descendant.

**Level:**The level of a node is defined by 1 + (the number of connections between the node and the root).

**Height of node:**The height of a node is the number of edges on the longest path between that node and a leaf.

**Height of tree :** The height of a tree is the height of its root node.

**Depth:**The depth of a node is the number of edges from the tree's root node to the node.

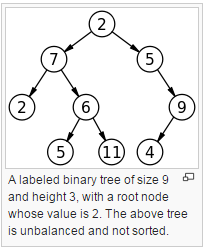
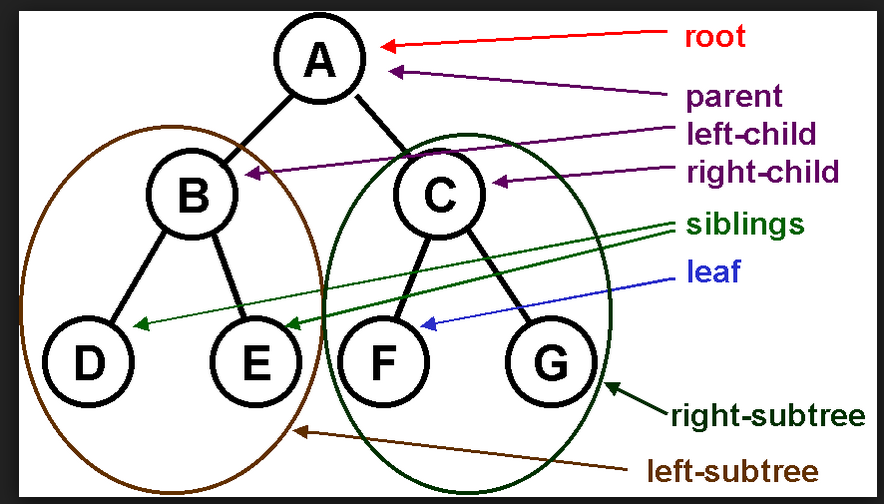
**Forest:**A forest is a set of n ≥ 0 disjoint trees.

Why Trees?  
1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer.

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).  
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).  
4. Like Linked Lists and unlike Arrays, Trees don’t have an upper limit on number of nodes as nodes are linked using pointers.

### Binary Tree

<http://www.geeksforgeeks.org/binary-tree-data-structure/>

A tree in which each node can have at most 2 children. A binary tree is a [tree](https://en.wikipedia.org/wiki/Tree_structure) [data structure](https://en.wikipedia.org/wiki/Data_structure) in which each node has at most two [children](https://en.wikipedia.org/wiki/Child_node), which are referred to as the left child and the right child.

**Structure of a node of binary tree**

class Node

{

    int key;

    Node left, right;

    public Node(int item)

    {

        key = item;

        left = right = null;

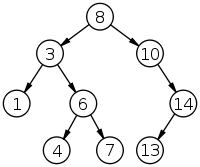
    }

}

#### Binary Search Tree

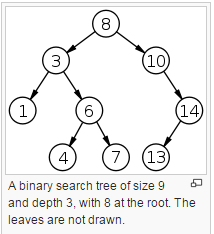
Binary Search Tree, is a node-based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys less than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.

[](http://quiz.geeksforgeeks.org/wp-content/uploads/2014/01/200px-Binary_search_tree.svg_.png)

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

* a binary tree with *n* complete levels will have 2*n*–1 elements
* Binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of [containers](https://en.wikipedia.org/wiki/Collection_(abstract_data_type)): [data structures](https://en.wikipedia.org/wiki/Data_structure) that store "items" (such as numbers, names and etc.) in [memory](https://en.wikipedia.org/wiki/Computer_memory). They allow fast lookup, addition and removal of items, and can be used to implement either [dynamic sets](https://en.wikipedia.org/wiki/Set_(abstract_data_type)) of items, or [lookup tables](https://en.wikipedia.org/wiki/Lookup_table) that allow finding an item by its key.
* Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees.



|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** |  | **Average** | **Worst Case** |
| **Space** |  | O(*n*) | O(*n*) |
| **Search** |  | O(log *n*) | O(*n*) |
| **Insert** |  | O(log *n*) | O(*n*) |
| **Delete** |  | O(log *n*) | O(*n*) |

##### Traversal of BST

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree

###### Depth First Traversals

(a) Inorder (Left, Root, Right) : 4 2 5 1 3  
(b) Preorder (Root, Left, Right) : 1 2 4 5 3  
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

###### Level Order Tree Traversal

Level order traversal of a tree is [breadth first traversal f](http://en.wikipedia.org/wiki/Breadth-first_traversal)or the tree.



*Example Tree*

Level order traversal of the above tree is 1 2 3 4 5

Searching a key  
To search a given key in Bianry Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root’s key, we recur for right subtree of root node. Otherwise we recur for left subtree.

// A utility function to search a given key in BST

Public Node search(Node root, int key)

{

    // Base Cases: root is null or key is present at root

    if (root==null || root.key==key)

        return root;

     // val is less than root's key

    if (root.key > key)

        return search(root.left, key);

     // val is greater than root's key

    return search(root.right, key);

}

Insertion of a key  
A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

// Java program to demonstrate insert operation in binary search tree

class BinarySearchTree {

/\* Class containing left and right child of current node and key value\*/

class Node {

int key;

Node left, right;

public Node(int item) {

key = item;

left = right = null;

}

}

// Root of BST

Node root;

// Constructor

BinarySearchTree() {

root = null;

}

// This method mainly calls insertRec()

void insert(int key) {

root = insertRec(root, key);

}

/\* A recursive function to insert a new key in BST \*/

Node insertRec(Node root, int key) {

/\* If the tree is empty, return a new node \*/

if (root == null) {

root = new Node(key);

return root;

}

/\* Otherwise, recur down the tree \*/

if (key < root.key)

root.left = insertRec(root.left, key);

else if (key > root.key)

root.right = insertRec(root.right, key);

/\* return the (unchanged) node pointer \*/

return root;

}

// This method mainly calls InorderRec()

void inorder() {

inorderRec(root);

}

// A utility function to do inorder traversal of BST

void inorderRec(Node root) {

if (root != null) {

inorderRec(root.left);

System.out.println(root.key);

inorderRec(root.right);

}

}

// Driver Program to test above functions

public static void main(String[] args) {

BinarySearchTree tree = new BinarySearchTree();

/\* Let us create following BST

50

/ \

30 70

/ \ / \

20 40 60 80 \*/

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

// print inorder traversal of the BST

tree.inorder();

}

}

##### **Time Complexity:**

The worst case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

##### Deleting a key

We have discussed [BST search and insert operations](http://quiz.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/). In this post, delete operation is discussed. When we delete a node, there possibilities arise.

**1)Node to be deleted is leaf:** Simply remove from the tree.

50 50

/ \ delete(20) / \

30 70 ---------> 30 70

/ \ / \ \ / \

20 40 60 80 40 60 80

**2) Node to be deleted has only one child:** Copy the child to the node and delete the child

50 50

/ \ delete(30) / \

30 70 ---------> 40 70

\ / \ / \

40 60 80 60 80

**3) Node to be deleted has two children:**Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

50 60

/ \ delete(50) / \

40 70 ---------> 40 70

/ \ \

60 80 80

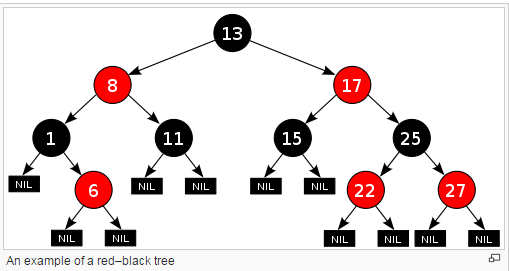
The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

##### Code ??

#### Red Black Tree

A red–black tree is a [binary search tree](https://en.wikipedia.org/wiki/Binary_search_tree) with an extra bit of data per node, its color, which can be either red or black.[[2]](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-:0-2) The extra bit of storage ensures an approximately balanced tree by constraining how nodes are colored from any path from the root to the leaf.[[2]](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-:0-2) Thus, it is a [data structure](https://en.wikipedia.org/wiki/Data_structure) which is a type of [self-balancing binary search tree](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree).

1. A node is either red or black.
2. The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
3. All leaves (NIL) are black.
4. If a node is red, then both its children are black.
5. Every [path](https://en.wikipedia.org/wiki/Path_(graph_theory)) from a given node to any of its descendant NIL nodes contains the same number of black nodes. The uniform number of black nodes in the paths from root to leaves is called the black-height of the red–black tree



|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** |  | **Average** | **Worst Case** |
| **Space** |  | O(*n*) | O(*n*) |
| **Search** |  | O(log *n*)[[1]](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-wiscurl-1) | O(log *n*)[[1]](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-wiscurl-1) |
| **Insert** |  | O(log *n*)[[1]](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-wiscurl-1) | O(log *n*)[[1]](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-wiscurl-1) |
| **Delete** |  | O(log *n*)[[1]](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-wiscurl-1) | O(log *n*)[[1]](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-wiscurl-1) |

#### Unbalanced and Balanced binary tree

#### AVL Trees

Self balancing: after insertion or deletion, balancing of tree happens

In [computer science](https://en.wikipedia.org/wiki/Computer_science), an **AVL tree** is a [self-balancing binary search tree](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree). It was the first such [data structure](https://en.wikipedia.org/wiki/Data_structure) to be invented.[[2]](https://en.wikipedia.org/wiki/AVL_tree#cite_note-2) In an AVL tree, the [heights](https://en.wikipedia.org/wiki/Tree_height) of the two [child](https://en.wikipedia.org/wiki/Child_node) subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take [O](https://en.wikipedia.org/wiki/Big_O_notation)(log *n*) time in both the average and worst cases, where *n* is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more [tree rotations](https://en.wikipedia.org/wiki/Tree_rotation).

AVL trees are often compared with [red–black trees](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree) because both support the same set of operations and take O(log *n*) time for the basic operations. For lookup-intensive applications, AVL trees are faster than red–black trees because they are more strictly balanced.[[4]](https://en.wikipedia.org/wiki/AVL_tree#cite_note-4) Similar to red–black trees, AVL trees are height-balanced.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** |  | **Average** | **Worst Case** |
| **Space** |  | O(*n*) | O(*n*) |
| **Search** |  | O(log *n*)[[1]](https://en.wikipedia.org/wiki/AVL_tree#cite_note-wiscurl-1) | O(log *n*)[[1]](https://en.wikipedia.org/wiki/AVL_tree#cite_note-wiscurl-1) |
| **Insert** |  | O(log *n*)[[1]](https://en.wikipedia.org/wiki/AVL_tree#cite_note-wiscurl-1) | O(log *n*)[[1]](https://en.wikipedia.org/wiki/AVL_tree#cite_note-wiscurl-1) |
| **Delete** |  | O(log *n*)[[1]](https://en.wikipedia.org/wiki/AVL_tree#cite_note-wiscurl-1) | O(log *n*)[[1]](https://en.wikipedia.org/wiki/AVL_tree#cite_note-wiscurl-1) |

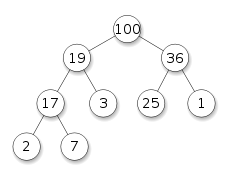
## Heap

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a **heap** is a specialized [tree](https://en.wikipedia.org/wiki/Tree_(data_structure))-based [data structure](https://en.wikipedia.org/wiki/Data_structure) that satisfies the *heap property:* If A is a parent [node](https://en.wikipedia.org/wiki/Node_(computer_science)) of B, then the *key* (the *value*) of node A is ordered with respect to the key of node B with the same ordering applying across the heap. A heap can be classified further as either a "**max heap**" or a "**min heap**".

**Max Heap** In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node.

**Min Heap**: In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node.

The heap is one maximally efficient implementation of an [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type) called a [priority queue](https://en.wikipedia.org/wiki/Priority_queue), and in fact priority queues are often referred to as "heaps", regardless of how they may be implemented. A common implementation of a heap is the [binary heap](https://en.wikipedia.org/wiki/Binary_heap), in which the tree is a complete binary tree (see figure).



Example of a complete binary max-heap with node keys being integers from 1 to 100

In a heap, the highest (or lowest) priority element is always stored at the root. A heap is not a sorted structure and can be regarded as partially ordered. As visible from the heap-diagram, there is no particular relationship among nodes on any given level, even among the siblings. When a heap is a complete binary tree, it has a smallest possible height—a heap with N nodes always has log N height. A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority.

Note that, as shown in the graphic, there is no implied ordering between siblings or cousins and no implied sequence for an [in-order traversal](https://en.wikipedia.org/wiki/Inorder_traversal) (as there would be in, e.g., a [binary search tree](https://en.wikipedia.org/wiki/Binary_search_tree)). The heap relation mentioned above applies only between nodes and their parents, grandparents, etc. The maximum number of children each node can have depends on the type of heap, but in many types it is at most two, which is known as a binary heap.

**Binary Heap:**

A **binary heap** is a [heap](https://en.wikipedia.org/wiki/Heap_(data_structure)) [data structure](https://en.wikipedia.org/wiki/Data_structure) that takes the form of a [binary tree](https://en.wikipedia.org/wiki/Binary_tree). Binary heaps are a common way of implementing [priority queues](https://en.wikipedia.org/wiki/Priority_queue).

A binary heap is defined as a binary tree with two additional constraints:[[3]](https://en.wikipedia.org/wiki/Binary_heap#cite_note-3)

* Shape property: a binary heap is a [*complete binary tree*](https://en.wikipedia.org/wiki/Complete_Binary_Tree); that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
* Heap property: the key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) the keys in the node's children, according to some [total order](https://en.wikipedia.org/wiki/Total_order).

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** |  | **Average** | **Worst Case** |
| **Space** |  | O(n) | O(n) |
| **Search** |  | O(n) | O(n) |
| **Insert** |  | O(1) | O(log n) |
| **Delete** |  | O(log n) | O(log n) |
| **Peek** |  | O(1) | O(1) |

Binomial Heap : <http://www.geeksforgeeks.org/binomial-heap-2/>

Fibonnaki heap : <http://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/>

### Heap Sort

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **heapsort** is a [comparison-based](https://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm). Heapsort can be thought of as an improved [selection sort](https://en.wikipedia.org/wiki/Selection_sort): like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. The improvement consists of the use of a [heap](https://en.wikipedia.org/wiki/Heap_(data_structure)) data structure rather than a linear-time search to find the maximum.

Although somewhat slower in practice on most machines than a well-implemented [quicksort](https://en.wikipedia.org/wiki/Quicksort), it has the advantage of a more favourable worst-case [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n* log *n*) runtime. Heapsort is an [in-place algorithm](https://en.wikipedia.org/wiki/In-place_algorithm)

Worst-case performance O(n\log n)

Best-case performance O(n\log n)

Average performance O(n\log n)

Worst-case space complexity O(1) auxiliary

## Sample Programs

### Write a Program to Find the Maximum Depth or Height of a Tree

<http://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/>

## List

### Array List

### Linked List:

#### Singly Linked list

#### Doubly Linked list

### Skip list

## Queue

A queue or FIFO (first in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: enqueue, the process of adding an element to the collection.(The element is added from the rear side) and dequeue, the process of removing the first element that was added. (The element is removed from the front side). It can be implemented by using both array and linked list.

Like [Stack](http://quiz.geeksforgeeks.org/stack-set-1/), [Queue](http://en.wikipedia.org/wiki/Queue_%28data_structure%29)is a linear structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (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 on Queue:**  
Mainly the following four basic operations are performed on queue:

**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.

[](http://quiz.geeksforgeeks.org/wp-content/uploads/2014/02/Queue.png)

**Applications of Queue:**  
Queue is used when things don’t have to be processed immediatly, but have to be processed in **F**irst **I**n**F**irst **O**ut order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search). This property of Queue makes it also useful in following kind of scenarios.

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.

See [this](http://introcs.cs.princeton.edu/43stack/)for more detailed applications of Queue and Stack.

### Code: queue implementation using array

### Code: queue implementation using linked list

### Priority Queue

Priority Queue is an extension of [queue](http://quiz.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/)with following properties.  
1) Every item has a priority associated with it.  
2) An element with high priority is dequeued before an element with low priority.  
3) If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports following operations.  
**insert(item, priority):**Inserts an item with given priority.  
**getHighestPriority():** Returns the highest priority item.  
**deleteHighestPriority():**Removes the highest priority item.

**How to implement priority queue?**  
***Using Array:***A simple implementation is to use array of following structure.

struct item {

int item;

int priority;

}

insert() operation can be implemented by adding an item at end of array in O(1) time.

getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.

deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don’t have to move items.

**Using Heaps:**  
Heap is generally preferred for priority queue implementation because heaps provide better performance compared arrays or linked list. In a Binary Heap, getHighestPriority() can be implemented in O(1) time, insert() can be implemented in O(Logn) time and deleteHighestPriority() can also be implemented in O(Logn) time.  
With [Fibonacci heap](http://en.wikipedia.org/wiki/Fibonacci_heap), insert() and getHighestPriority() can be implemented in O(1) amortized time and deleteHighestPriority() can be implemented in O(Logn) amortized time.

**Applications of Priority Queue:**  
1) CPU Scheduling  
2) Graph algorithms like [Dijkstra’s shortest path algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/), [Prim’s Minimum Spanning Tree](http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-mst-for-adjacency-list-representation/), etc  
3) All [queue applications](http://www.geeksforgeeks.org/applications-of-queue-data-structure/) where priority is involved.

### Deque

[Deque or Double Ended Queue](http://en.wikipedia.org/wiki/Double-ended_queue) is a generalized version of [Queue data structure](http://quiz.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/)that allows insert and delete at both ends.

**Operations on Deque:**  
Mainly the following four basic operations are performed on queue:

***insetFront()***: Adds an item at the front of Deque.  
***insertLast()***: Adds an item at the rear of Deque.  
***deleteFront()***: Deletes an item from front of Deque.  
***deleteLast()***: Deletes an item from rear of Deque.

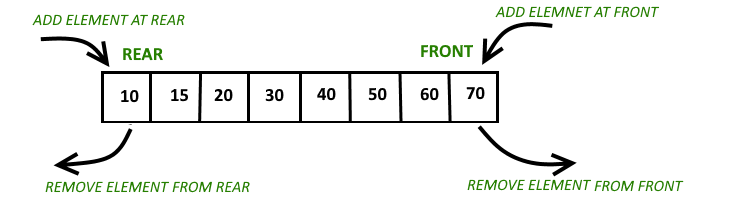
In addition to above operations, following operations are also supported  
***getFront()***: Gets the front item from queue.  
***getRear()***: Gets the last item from queue.  
***isEmpty()***: Checks whether Deque is empty or not.  
***isFull()***: Checks whether Deque is full or not.

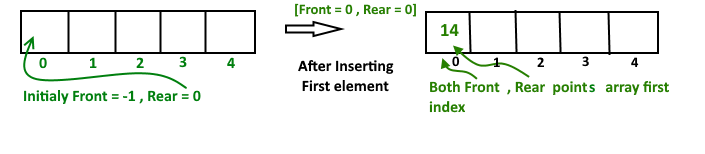
**Applications of Deque:**  
Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.  
Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque. For example see [Maximum of all subarrays of size k problem.](http://www.geeksforgeeks.org/maximum-of-all-subarrays-of-size-k/), [0-1 BFS](http://www.geeksforgeeks.org/0-1-bfs-shortest-path-binary-graph/) and [Find the first circular tour that visits all petrol pumps](http://www.geeksforgeeks.org/find-a-tour-that-visits-all-stations/).

See [wiki page](http://en.wikipedia.org/wiki/Double-ended_queue#Applications)for another example of A-Steal job scheduling algorithm where Deque is used as deletions operation is required at both ends.

**Language Support:**  
C++ STL provides implementation of Deque as [std::deque](http://www.cplusplus.com/reference/deque/deque/) and Java provides [Deque interface](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html). See [this](http://en.wikipedia.org/wiki/Double-ended_queue#Language_support)for more details.

**Implementation:**  
A Deque can be implemented either using a [doubly linked list](http://quiz.geeksforgeeks.org/doubly-linked-list/) or circular array. In both implementation, we can implement all operations in O(1) time.



**Circular array implementation deque**  
For implementing deque, we need to keep track of two indices, front and rear. We enqueue(push) an item at the rear or the front end of qedue and dequeue(pop) an item from both rear and front end.  
**Working**  
1. Create an empty array ‘arr’ of size ‘n’  
initialize **front = -1** , **rear = 0**  
Inserting First element in deque either front end or read end they both lead to the same result.  
[](http://www.contribute.geeksforgeeks.org/wp-content/uploads/deque-Copy-2.png)  
After insert **Front** Points = 0 and **Rear** points = 0  
**Insert Elements at Rear end**

a). First we check deque if Full or Not

b). IF Rear == Size-1

then reinitialize Rear = 0 ;

Else increment Rear by '1'

and push current key into Arr[ rear ] = key

Front remain same.

**Insert Elements at Front end**

a). First we check deque if Full or Not

b). IF Front == 0 || initial position, move Front

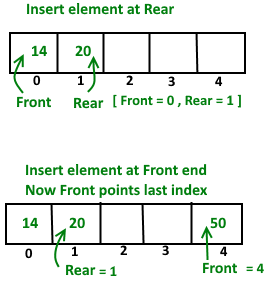
to points last index of array

front = size - 1

Else decremented front by '1' and push

current key into Arr[ Front] = key

Rear remain same.

[](http://www.geeksforgeeks.org/wp-content/uploads/deque.png)

**Delete Element From Rear end**

a). first Check deque is Empty or Not

b). If deque has only one element

front = -1 ; rear =-1 ;

Else IF Rear points to the first index of array

it's means we have to move rear to points

last index [ now first inserted element at

front end become rear end ]

rear = size-1 ;

Else || decrease rear by '1'

rear = rear-1;

**Delete Element From Front end**

a). first Check deque is Empty or Not

b). If deque has only one element

front = -1 ; rear =-1 ;

Else IF front points to the last index of the array

it's means we have no more elements in array so

we move front to points first index of array

front = 0 ;

Else || increment Front by '1'

front = front+1;

### code: Implement Queue using Stacks

### code: Find the largest multiple of 3 | (Using Queue)

### code: Implement LRU Cache

### code: Find the first circular tour that visits all petrol pumps

Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.

**1.** The amount of petrol that every petrol pump has.  
**2.** Distance from that petrol pump to the next petrol pump.

Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is O(n). Assume for 1 litre petrol, the truck can go 1 unit of distance.

For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where truck can make a circular tour is 2nd petrol pump. Output should be “start = 1” (index of 2nd petrol pump).

A **Simple Solution** is to consider every petrol pumps as starting point and see if there is a possible tour. If we find a starting point with feasible solution, we return that starting point. The worst case time complexity of this solution is O(n^2).

We can **use a Queue**to store the current tour. We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or current amount of petrol becomes negative. If the amount becomes negative, then we keep dequeueing petrol pumps till the current amount becomes positive or queue becomes empty.

Instead of creating a separate queue, we use the given array itself as queue. We maintain two index variables start and end that represent rear and front of queue.

## Stack

LIFO, push peek, pop

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last 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 or Top: Returns top element of stack.
* **isEmpty:**Returns true if stack is empty, else fals.

[](http://quiz.geeksforgeeks.org/wp-content/uploads/2013/03/stack.png)

**How to understand a stack practically?**  
There are many real life examples of stack. Consider the simple example of plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

**Time Complexities of operations on stack:**

push(), pop(), esEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

**Applications of stack:**

* [Balancing of symbols](http://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/)
* [Infix to Postfix](http://quiz.geeksforgeeks.org/stack-set-2-infix-to-postfix/) /Prefix conversion
* Redo-undo features at many places like editors, photoshop.
* Forward and backward feature in web browsers
* Used in many algorithms like [Tower of Hanoi,](http://www.geeksforgeeks.org/recursive-functions/) [tree traversals](http://www.geeksforgeeks.org/618/), [stock span problem](http://www.geeksforgeeks.org/the-stock-span-problem/), [histogram problem](http://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/).
* Other applications can be Backtracking, [Knight tour problem](http://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/), [rat in a maze](http://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/),[N queen problem](http://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/) and [sudoku solver](http://www.geeksforgeeks.org/backtracking-set-7-suduku/)

**Implementation:**  
There are two ways to implement a stack:

* Using array
* Using linked list



Simple representation of a stack runtime with *push* and *pop* operations.

**Infix expression:**The expression of the form a op b. When an operator is in-between every pair of operands.

**Postfix expression:**The expression of the form a b op. When an operator is followed for every pair of operands.

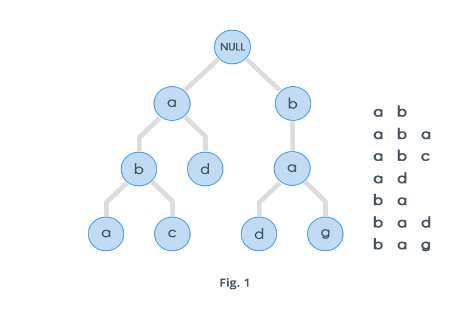
## Circular Array

## Map

## Trie

[Trie](http://en.wikipedia.org/wiki/Trie) is an efficient information re**trie**val data structure. Using trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to **M \* log N**, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in O(M) time. However the penalty is on trie storage requirements.

Every node of trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as leaf node. A trie node field value will be used to distinguish the node as leaf node (there are other uses of the value field).



Dictionary representation **:** A common application of a trie is storing a [predictive text](https://en.wikipedia.org/wiki/Predictive_text) or [autocomplete](https://en.wikipedia.org/wiki/Autocomplete) dictionary, such as found on a [mobile telephone](https://en.wikipedia.org/wiki/Mobile_telephone). Tries are also well suited for implementing approximate matching algorithms,[8] including those used in spell checking and hyphenation software.

## Set

## HashTable

## Graph

# Search Algorithm

## Linear search

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **linear search** or **sequential search** is a method for finding a target value within a [list](https://en.wikipedia.org/wiki/List_(computing)). It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.[[1]](https://en.wikipedia.org/wiki/Linear_search#cite_note-FOOTNOTEKnuth1998.C2.A76.1_.28.22Sequential_search.22.29-1)

Linear search runs in at worst [linear time](https://en.wikipedia.org/wiki/Time_complexity#linear_time) and makes at most *n* comparisons, where *n* is the length of the list. If each element is equally likely to be searched, then linear search has an average case of *n*/2 comparisons, but the average case can be affected if the search probabilities for each element vary. Linear search is rarely practical because other search algorithms and schemes, such as the [binary search algorithm](https://en.wikipedia.org/wiki/Binary_search_algorithm) and [hash tables](https://en.wikipedia.org/wiki/Hash_table), allow significantly faster searching for all but short lists.[[2]](https://en.wikipedia.org/wiki/Linear_search#cite_note-FOOTNOTEKnuth1998.C2.A76.2_.28.22Searching_by_Comparison_Of_Keys.22.29-2)

If we search for a book by Jonathan Swift by starting at the left end of

the shelf, checking book by book as we move to the right, we call that

technique ***linear search***. In terms of an array in a computer, we start

at the beginning of the array, examine each array element in turn (AOE1\_,

then AOE2\_, then AOE3\_, and so on, up through AOEn\_) and record where we

find x, if we find it at all.

### Java Code

|  |
| --- |
| public String linearSearchForValue(int value){ |

|  |  |
| --- | --- |
| 086 |  |

|  |  |
| --- | --- |
| 087 | boolean valueInArray = false; |

|  |  |
| --- | --- |
| 088 |  |

|  |  |
| --- | --- |
| 089 | String indexsWithValue = ""; |

|  |  |
| --- | --- |
| 090 |  |

|  |  |
| --- | --- |
| 091 | for(int i = 0; i < arraySize; i++){ |

|  |  |
| --- | --- |
| 092 |  |

|  |  |
| --- | --- |
| 093 | if(theArray[i] == value){ |

|  |  |
| --- | --- |
| 094 |  |

|  |  |
| --- | --- |
| 095 | valueInArray = true; |

|  |  |
| --- | --- |
| 096 |  |

|  |  |
| --- | --- |
| 097 | indexsWithValue+= i + " "; |

|  |  |
| --- | --- |
| 098 |  |

|  |  |
| --- | --- |
| 099 | } |

|  |  |
| --- | --- |
| 100 |  |

|  |  |
| --- | --- |
| 101 | printHorzArray(i, -1); |

|  |  |
| --- | --- |
| 102 |  |

|  |  |
| --- | --- |
| 103 | } |

|  |  |
| --- | --- |
| 104 |  |

|  |  |
| --- | --- |
| 105 | if(!valueInArray){ |

|  |  |
| --- | --- |
| 106 |  |

|  |  |
| --- | --- |
| 107 | indexsWithValue = "None"; |

|  |  |
| --- | --- |
| 108 |  |

|  |  |
| --- | --- |
| 109 | } |

|  |  |
| --- | --- |
| 110 |  |

|  |  |
| --- | --- |
| 111 | System.out.print("The Value was Found in the Following: " + indexsWithValue); |

|  |  |
| --- | --- |
| 112 |  |

|  |  |
| --- | --- |
| 113 | System.out.println(); |

|  |  |
| --- | --- |
| 114 |  |

|  |  |
| --- | --- |
| 115 | return indexsWithValue; |

|  |  |
| --- | --- |
| 116 |  |

|  |  |
| --- | --- |
| 117 | } |

## Binary Search

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **binary search**, also known as **half-interval search**[[1]](https://en.wikipedia.org/wiki/Binary_search_algorithm#cite_note-1) or **logarithmic search**,[[2]](https://en.wikipedia.org/wiki/Binary_search_algorithm#cite_note-FOOTNOTEKnuth1998.C2.A76.2.1_.28.22Searching_an_ordered_table.22.29.2C_subsection_.22Binary_search.22-2) is a [search algorithm](https://en.wikipedia.org/wiki/Search_algorithm) that finds the position of a target value within a [sorted array](https://en.wikipedia.org/wiki/Sorted_array).[[3]](https://en.wikipedia.org/wiki/Binary_search_algorithm#cite_note-FOOTNOTECormenLeisersonRivestStein200939-3)[[4]](https://en.wikipedia.org/wiki/Binary_search_algorithm#cite_note-4) Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful.

Binary search runs in at worst [logarithmic time](https://en.wikipedia.org/wiki/Time_complexity#logarithmic_time), making [*O*](https://en.wikipedia.org/wiki/Big_O_notation)(log *n*) comparisons, where *n* is the number of elements in the array and log is the [logarithm](https://en.wikipedia.org/wiki/Logarithm). Binary search takes only constant (*O*(1)) space, meaning that the space taken by the algorithm is the same for any number of elements in the array.[[5]](https://en.wikipedia.org/wiki/Binary_search_algorithm#cite_note-avgperf-5) Although specialized [data structures](https://en.wikipedia.org/wiki/Data_structures) designed for fast searching—such as [hash tables](https://en.wikipedia.org/wiki/Hash_tables)—can be searched more efficiently, binary search applies to a wider range of search problems.

Although the idea is simple, implementing binary search correctly requires attention to some subtleties about its exit conditions and midpoint calculation.

### Procedure[[edit](https://en.wikipedia.org/w/index.php?title=Binary_search_algorithm&action=edit&section=2)]

Given an array *A* of *n* elements with values or [records](https://en.wikipedia.org/wiki/Record_(computer_science)) *A*0 ... *An*−1, sorted such that *A*0 ≤ ... ≤ *An*−1, and target value *T*, the following [subroutine](https://en.wikipedia.org/wiki/Subroutine) uses binary search to find the index of *T* in *A*.[[6]](https://en.wikipedia.org/wiki/Binary_search_algorithm#cite_note-FOOTNOTEKnuth1998.C2.A76.2.1_.28.22Searching_an_ordered_table.22.29.2C_subsection_.22Algorithm_B.22-6)

1. Set *L* to 0 and *R* to *n* − 1.
2. If *L* > *R*, the search terminates as unsuccessful.
3. Set *m* (the position of the middle element) to the [floor](https://en.wikipedia.org/wiki/Floor_and_ceiling_functions) (the largest previous integer) of (*L* + *R*) / 2.
4. If A*m* < *T*, set L to *m* + 1 and go to step 2.
5. If A*m* > T, set R to *m* – 1 and go to step 2.
6. Now A*m* = *T*, the search is done; return *m*.

### Java code

|  |
| --- |
| // The Binary Search is quicker than the linear search |

|  |  |
| --- | --- |
| 225 | // because all the values are sorted. Because everything |

|  |  |
| --- | --- |
| 226 | // is sorted once you get to a number larger than what |

|  |  |
| --- | --- |
| 227 | // you are looking for you can stop the search. Also |

|  |  |
| --- | --- |
| 228 | // you be able to start searching from the middle |

|  |  |
| --- | --- |
| 229 | // which speeds the search. It also works best when |

|  |  |
| --- | --- |
| 230 | // there are no duplicates |

|  |  |
| --- | --- |
| 231 |  |

|  |  |
| --- | --- |
| 232 | public void binarySearchForValue(int value){ |

|  |  |
| --- | --- |
| 233 |  |

|  |  |
| --- | --- |
| 234 | int lowIndex = 0; |

|  |  |
| --- | --- |
| 235 | int highIndex = arraySize - 1; |

|  |  |
| --- | --- |
| 236 |  |

|  |  |
| --- | --- |
| 237 | while(lowIndex <= highIndex){ |

|  |  |
| --- | --- |
| 238 |  |

|  |  |
| --- | --- |
| 239 | int middleIndex = (highIndex + lowIndex) / 2; |

|  |  |
| --- | --- |
| 240 |  |

|  |  |
| --- | --- |
| 241 | if(theArray[middleIndex] < value) lowIndex = middleIndex + 1; |

|  |  |
| --- | --- |
| 242 |  |

|  |  |
| --- | --- |
| 243 | else if(theArray[middleIndex] > value) highIndex = middleIndex - 1; |

|  |  |
| --- | --- |
| 244 |  |

|  |  |
| --- | --- |
| 245 | else { |

|  |  |
| --- | --- |
| 246 |  |

|  |  |
| --- | --- |
| 247 | System.out.println("\nFound a Match for " + value + " at Index " + middleIndex); |

|  |  |
| --- | --- |
| 248 |  |

|  |  |
| --- | --- |
| 249 | lowIndex = highIndex + 1; |

|  |  |
| --- | --- |
| 250 |  |

|  |  |
| --- | --- |
| 251 | } |

|  |  |
| --- | --- |
| 252 |  |

|  |  |
| --- | --- |
| 253 | printHorzArray(middleIndex, -1); |

|  |  |
| --- | --- |
| 254 |  |

|  |  |
| --- | --- |
| 255 | } |

|  |  |
| --- | --- |
| 256 |  |

|  |  |
| --- | --- |
| 257 | } |

# Sort Algorithm

## Bubble Sort

**Bubble sort**, sometimes referred to as **sinking sort**, is a simple [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) that repeatedly steps through the list to be sorted, compares each pair of adjacent items and [swaps](https://en.wikipedia.org/wiki/Swap_(computer_science)) them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort), is named for the way smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort).[[1]](https://en.wikipedia.org/wiki/Bubble_sort#cite_note-Knuth-1) It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position. It has [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n*2) [time complexity](https://en.wikipedia.org/wiki/Time_complexity)

### Step-by-step example[[edit](https://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=4)]

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

**First Pass**

( **5** **1** 4 2 8 ) {\displaystyle \to } ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) {\displaystyle \to } ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) {\displaystyle \to } ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) {\displaystyle \to } ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass**

( **1** **4** 2 5 8 ) {\displaystyle \to } ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) {\displaystyle \to } ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) {\displaystyle \to } ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) {\displaystyle \to } ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass**

( **1** **2** 4 5 8 ) {\displaystyle \to } ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) {\displaystyle \to } ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) {\displaystyle \to } ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) {\displaystyle \to } ( 1 2 4 **5** **8** )

### Pseudocode implementation[[edit](https://en.wikipedia.org/w/index.php?title=Bubble_sort&action=edit&section=6)]

The algorithm can be expressed as (0-based array):

procedure bubbleSort( A : list of sortable items )

n = length(A)

repeat

swapped = false

for i = 1 to n-1 inclusive do

/\* if this pair is out of order \*/

if A[i-1] > A[i] then

/\* swap them and remember something changed \*/

swap( A[i-1], A[i] )

swapped = true

end if

end for

until not swapped

end procedure

### java code

|  |
| --- |
| // This bubble sort will sort everything from |

|  |  |
| --- | --- |
| 182 | // smallest to largest |

|  |  |
| --- | --- |
| 183 |  |

|  |  |
| --- | --- |
| 184 | public void bubbleSort(){ |

|  |  |
| --- | --- |
| 185 |  |

|  |  |
| --- | --- |
| 186 | // i starts at the end of the Array |

|  |  |
| --- | --- |
| 187 | // As it is decremented all indexes greater |

|  |  |
| --- | --- |
| 188 | // then it are sorted |

|  |  |
| --- | --- |
| 189 |  |

|  |  |
| --- | --- |
| 190 | for(int i = arraySize - 1; i > 1; i--){ |

|  |  |
| --- | --- |
| 191 |  |

|  |  |
| --- | --- |
| 192 | // The inner loop starts at the beginning of |

|  |  |
| --- | --- |
| 193 | // the array and compares each value next to each |

|  |  |
| --- | --- |
| 194 | // other. If the value is greater then they are |

|  |  |
| --- | --- |
| 195 | // swapped |

|  |  |
| --- | --- |
| 196 |  |

|  |  |
| --- | --- |
| 197 | for(int j = 0; j < i; j++){ |

|  |  |
| --- | --- |
| 198 |  |

|  |  |
| --- | --- |
| 199 | // To change sort to Descending change to < |

|  |  |
| --- | --- |
| 200 |  |

|  |  |
| --- | --- |
| 201 | if(theArray[j] > theArray[j + 1]){ |

|  |  |
| --- | --- |
| 202 |  |

|  |  |
| --- | --- |
| 203 | swapValues(j, j+1); |

|  |  |
| --- | --- |
| 204 |  |

|  |  |
| --- | --- |
| 205 | printHorzArray(i, j); |

|  |  |
| --- | --- |
| 206 |  |

|  |  |
| --- | --- |
| 207 | } |

|  |  |
| --- | --- |
| 208 |  |

|  |  |
| --- | --- |
| 209 | } |

|  |  |
| --- | --- |
| 210 |  |

|  |  |
| --- | --- |
| 211 | } |

|  |  |
| --- | --- |
| 212 |  |

|  |  |
| --- | --- |
| 213 | } |

|  |  |
| --- | --- |
| 214 |  |

|  |  |
| --- | --- |
| 215 | public void swapValues(int indexOne, int indexTwo){ |

|  |  |
| --- | --- |
| 216 |  |

|  |  |
| --- | --- |
| 217 | int temp = theArray[indexOne]; |

|  |  |
| --- | --- |
| 218 | theArray[indexOne] = theArray[indexTwo]; |

|  |  |
| --- | --- |
| 219 | theArray[indexTwo] = temp; |

|  |  |
| --- | --- |
| 220 |  |

|  |  |
| --- | --- |
| 221 | } |

## Selection sort

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **selection sort** is a [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm), specifically an [in-place](https://en.wikipedia.org/wiki/In-place_algorithm) [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort). It has [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n*2) [time complexity](https://en.wikipedia.org/wiki/Time_complexity), making it inefficient on large lists, and generally performs worse than the similar [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort). Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

### Example[[edit](https://en.wikipedia.org/w/index.php?title=Selection_sort&action=edit&section=1)]

Here is an example of this sort algorithm sorting five elements:

64 25 12 22 11 // this is the initial, starting state of the array

11 25 12 22 64 // sorted sublist = {11}

11 12 25 22 64 // sorted sublist = {11, 12}

11 12 22 25 64 // sorted sublist = {11, 12, 22}

11 12 22 25 64 // sorted sublist = {11, 12, 22, 25}

11 12 22 25 64 // sorted sublist = {11, 12, 22, 25, 64}

### Java code

|  |
| --- |
| // Selection sort search for the smallest number in the array |

|  |  |
| --- | --- |
| 260 | // saves it in the minimum spot and then repeats searching |

|  |  |
| --- | --- |
| 261 | // through the entire array each time |

|  |  |
| --- | --- |
| 262 |  |

|  |  |
| --- | --- |
| 263 | public void selectionSort(){ |

|  |  |
| --- | --- |
| 264 |  |

|  |  |
| --- | --- |
| 265 | for(int x=0; x < arraySize; x++){ |

|  |  |
| --- | --- |
| 266 | int minimum = x; |

|  |  |
| --- | --- |
| 267 |  |

|  |  |
| --- | --- |
| 268 | for(int y=x; y < arraySize; y++){ |

|  |  |
| --- | --- |
| 269 |  |

|  |  |
| --- | --- |
| 270 | // To change direction of sort just change |

|  |  |  |
| --- | --- | --- |
| 271 |  | // this from > to < |

|  |  |
| --- | --- |
| 272 |  |

|  |  |
| --- | --- |
| 273 | if(theArray[minimum]>theArray[y]){ |

|  |  |
| --- | --- |
| 274 | minimum = y; |

|  |  |
| --- | --- |
| 275 | } |

|  |  |
| --- | --- |
| 276 | } |

|  |  |
| --- | --- |
| 277 |  |

|  |  |
| --- | --- |
| 278 | swapValues(x, minimum); |

|  |  |
| --- | --- |
| 279 |  |

|  |  |
| --- | --- |
| 280 | printHorzArray(x, -1); |

|  |  |
| --- | --- |
| 281 | } |

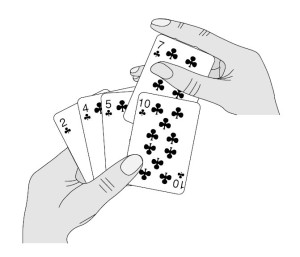
|  |  |
| --- | --- |
| 282 |  |

Ref: <https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm>

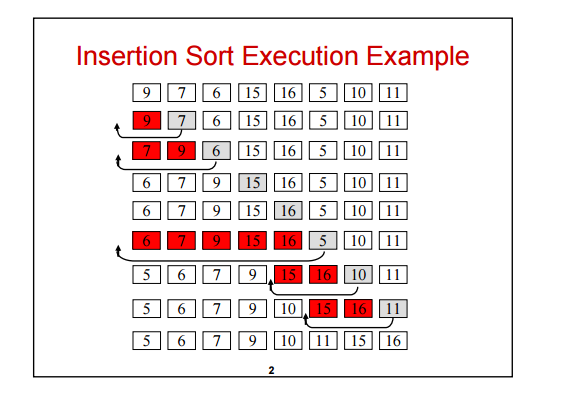
## Insertion Sort

<https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm>

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

[](http://quiz.geeksforgeeks.org/wp-content/uploads/2013/03/Insertion-Sort.jpg)

**Algorithm**  
// Sort an arr[] of size n  
insertionSort(arr, n)  
Loop from i = 1 to n-1.  
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

**Example:**  
[](http://quiz.geeksforgeeks.org/wp-content/uploads/2013/03/insertion-sort.png)  
**Another Example:**  
**12**, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 5 (Size of input array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12  
**11, 12**, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13  
**11, 12, 13**, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.  
**5, 11, 12, 13**, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.  
**5, 6, 11, 12, 13**

## Quick Sort

<https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm>

## Merge Sort

<https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm>

# Comparison of Time Complexity

## Comparison of sorting algos

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Time** | | |  | | | |
| **Sort** | **Average** | **Best** | **Worst** | **Space** | **Stability** | **Remarks** |  |
| [Bubble sort](http://www.cprogramming.com/tutorial/computersciencetheory/sorting1.html) | O(n^2) | O(n^2) | O(n^2) | Constant | Stable | Always use a modified bubble sort |  |
| [Modified Bubble sort](http://www.cprogramming.com/tutorial/computersciencetheory/sorting1.html) | O(n^2) | O(n) | O(n^2) | Constant | Stable | Stops after reaching a sorted array |  |
| [Selection Sort](http://www.cprogramming.com/tutorial/computersciencetheory/sorting2.html) | O(n^2) | O(n^2) | O(n^2) | Constant | Stable | Even a perfectly sorted input requires scanning the entire array |  |
| [Insertion Sort](http://www.cprogramming.com/tutorial/computersciencetheory/sorting2.html) | O(n^2) | O(n) | O(n^2) | Constant | Stable | In the best case (already sorted), every insert requires constant time |  |
| [Heap Sort](http://www.cprogramming.com/tutorial/computersciencetheory/heapsort.html) | O(n\*log(n)) | O(n\*log(n)) | O(n\*log(n)) | Constant | Instable | By using input array as storage for the heap, it is possible to achieve constant space |  |
| [Merge Sort](http://www.cprogramming.com/tutorial/computersciencetheory/mergesort.html) | O(n\*log(n)) | O(n\*log(n)) | O(n\*log(n)) | Depends | Stable | On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space |  |
| [Quicksort](http://www.cprogramming.com/tutorial/computersciencetheory/quicksort.html) | O(n\*log(n)) | O(n\*log(n)) | O(n^2) | Constant | Stable | Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such |  |

## Common Data Structure Operations

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|  | Average | | | | Worst | | | | Worst |
|  | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |  |
| [Array](http://en.wikipedia.org/wiki/Array_data_structure) | Θ(1) | Θ(n) | Θ(n) | Θ(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| [Stack](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| [Queue](http://en.wikipedia.org/wiki/Queue_(abstract_data_type)) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| [Singly-Linked List](http://en.wikipedia.org/wiki/Singly_linked_list#Singly_linked_lists) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| [Doubly-Linked List](http://en.wikipedia.org/wiki/Doubly_linked_list) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| [Skip List](http://en.wikipedia.org/wiki/Skip_list) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| [Hash Table](http://en.wikipedia.org/wiki/Hash_table) | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | O(n) | O(n) | O(n) | O(n) |
| [Binary Search Tree](http://en.wikipedia.org/wiki/Binary_search_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| [Cartesian Tree](https://en.wikipedia.org/wiki/Cartesian_tree) | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(n) | O(n) | O(n) | O(n) |
| [B-Tree](http://en.wikipedia.org/wiki/B_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| [Red-Black Tree](http://en.wikipedia.org/wiki/Red-black_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| [Splay Tree](https://en.wikipedia.org/wiki/Splay_tree) | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| [AVL Tree](http://en.wikipedia.org/wiki/AVL_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| [KD Tree](http://en.wikipedia.org/wiki/K-d_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |

## Array Sorting Algorithms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time Complexity | | | Space Complexity |
|  | Best | Average | Worst | Worst |
| [Quicksort](http://en.wikipedia.org/wiki/Quicksort) | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| [Mergesort](http://en.wikipedia.org/wiki/Merge_sort) | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| [Timsort](http://en.wikipedia.org/wiki/Timsort) | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |
| [Heapsort](http://en.wikipedia.org/wiki/Heapsort) | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| [Bubble Sort](http://en.wikipedia.org/wiki/Bubble_sort) | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| [Insertion Sort](http://en.wikipedia.org/wiki/Insertion_sort) | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort) | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| [Tree Sort](https://en.wikipedia.org/wiki/Tree_sort) | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |
| [Shell Sort](http://en.wikipedia.org/wiki/Shellsort) | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| [Bucket Sort](http://en.wikipedia.org/wiki/Bucket_sort) | Ω(n+k) | Θ(n+k) | O(n^2) | O(n) |
| [Radix Sort](http://en.wikipedia.org/wiki/Radix_sort) | Ω(nk) | Θ(nk) | O(nk) | O(n+k) |
| [Counting Sort](https://en.wikipedia.org/wiki/Counting_sort) | Ω(n+k) | Θ(n+k) | O(n+k) | O(k) |
| [Cubesort](https://en.wikipedia.org/wiki/Cubesort) | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |

# Dynamic programming

<https://www.youtube.com/watch?v=mmjDZGSr7EA&list=PLqM7alHXFySGbXhWx7sBJEwY2DnhDjmxm>

<http://www.geeksforgeeks.org/dynamic-programming/>

<https://www.youtube.com/playlist?list=PLqM7alHXFySGbXhWx7sBJEwY2DnhDjmxm>

## Find the nth fibonaki number:

The **Fibonacci sequence** is a **series** where the next term is the sum of pervious two terms. The first two terms of the **Fibonacci sequence** is 0 followed by 1. The **Fibonacci sequence**: 0, 1, 1, 2, 3, 5, 8, 13, 21.

**public** **class** FibonakiNumber {

**public** **static** **void** main(String[] args) {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println(*fib*(i));

}

}

**private** **static** **int** fib(**int** n) {

**if** (n == 0) {

**return** 0;

} **else** **if** (n == 1) {

**return** 1;

} **else** {

**return** *fib*(n - 1) + *fib*(n - 2);

}

}

}

## Find the Longest Increasing Subsequence



Find the combination of non consecutive numbers in an array of numbers whole sum is max.

# References

<https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm>

<http://www.newthinktank.com/2013/02/java-sort-algorithm/>

<https://www.youtube.com/watch?v=JUOyKSZScW0&index=2&list=PLGLfVvz_LVvReUrWr94U-ZMgjYTQ538nT>

sorting algos: <https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm>

quick sort: <https://interactivepython.org/runestone/static/pythonds/SortSearch/TheQuickSort.html>