Data Structures / Sorting

Time Complexities

Akash Anjanappa

Contents

[Stack 2](#_Toc508276465)

[Queue 3](#_Toc508276466)

[LinkedList 4](#_Toc508276467)

[Time Complexities 5](#_Toc508276468)

[Java Data Structures: 5](#_Toc508276469)

[Sorting 7](#_Toc508276470)

[Bubble sort 7](#_Toc508276471)

[Selection Sort 8](#_Toc508276472)

[Merge Sort 9](#_Toc508276473)

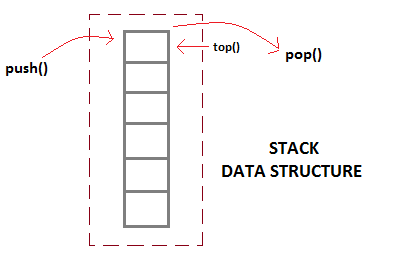
[Quick Sort 11](#_Toc508276474)

[Heap Sort 12](#_Toc508276475)

[Sorting Summary 14](#_Toc508276476)

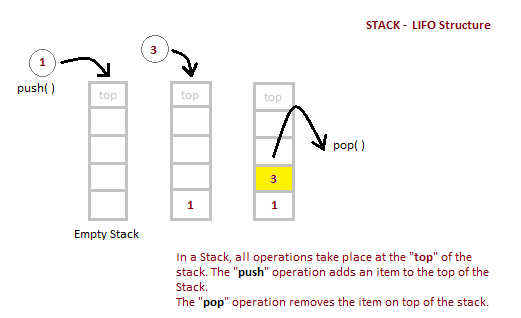
# Stack

Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.



**Implementation of Stack**

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.

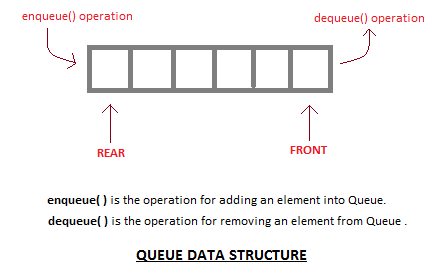


#### Analysis of Stacks

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

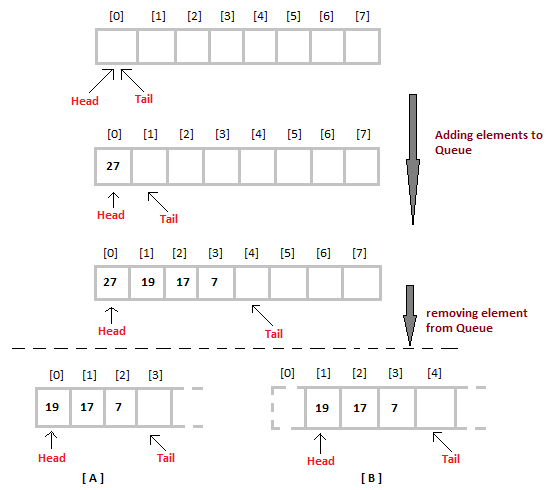
* **Push Operation: O(1)**
* **Pop Operation: O(1)**
* **Top Operation: O(1)**
* **Search Operation: O(n)**

Queue  
Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of existing element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will also be removed first.



#### Implementation of Queue

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array. Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.



**Analysis of Queue**

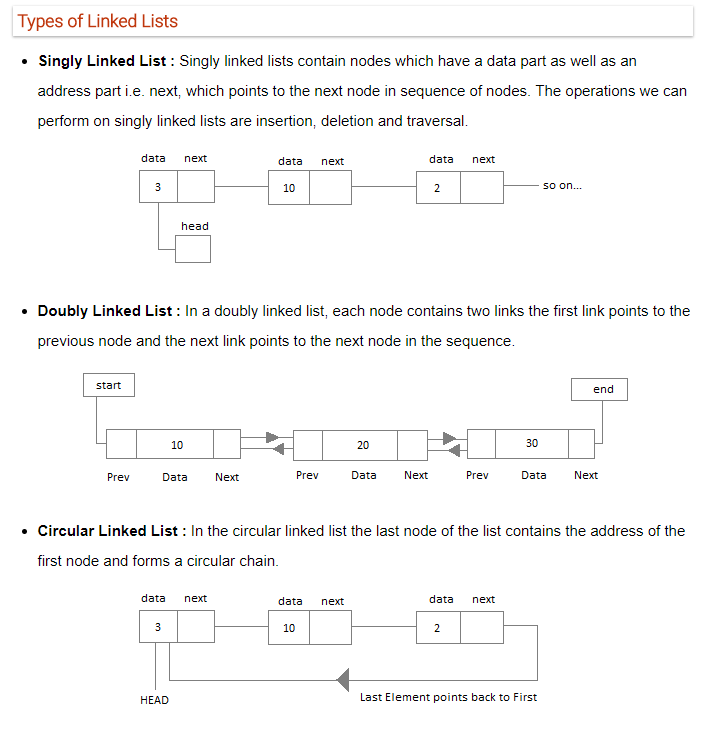
Enqueue : **O(1)**

Dequeue : **O(1)**

Size : **O(1)**

# LinkedList

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.



**Linked List:**

* **Inserting**: **O(1)**, if done at the head, **O(n)** if anywhere else since we have to reach that position by traveseing the linkedlist linearly.
* **Deleting**: **O(1)**, if done at the head, **O(n)** if anywhere else since we have to reach that position by traveseing the linkedlist linearly.
* **Searching**: **O(n)**

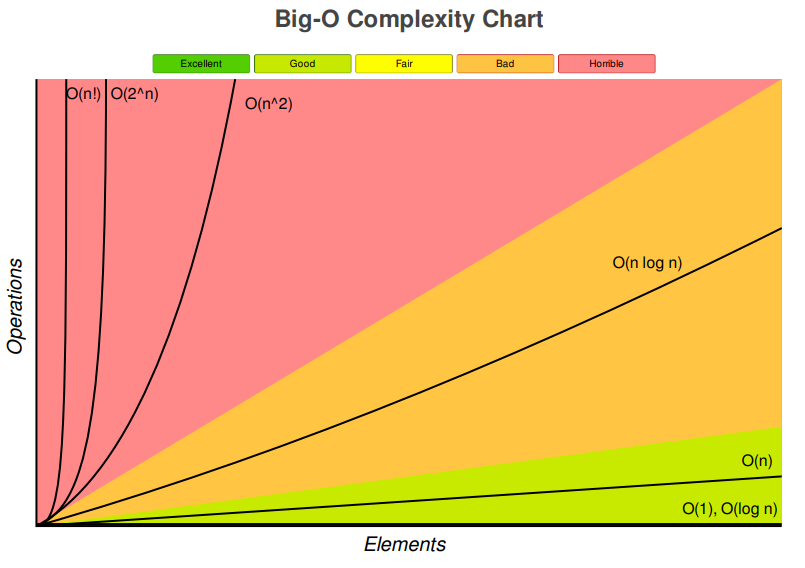
**Doubly-Linked List:**

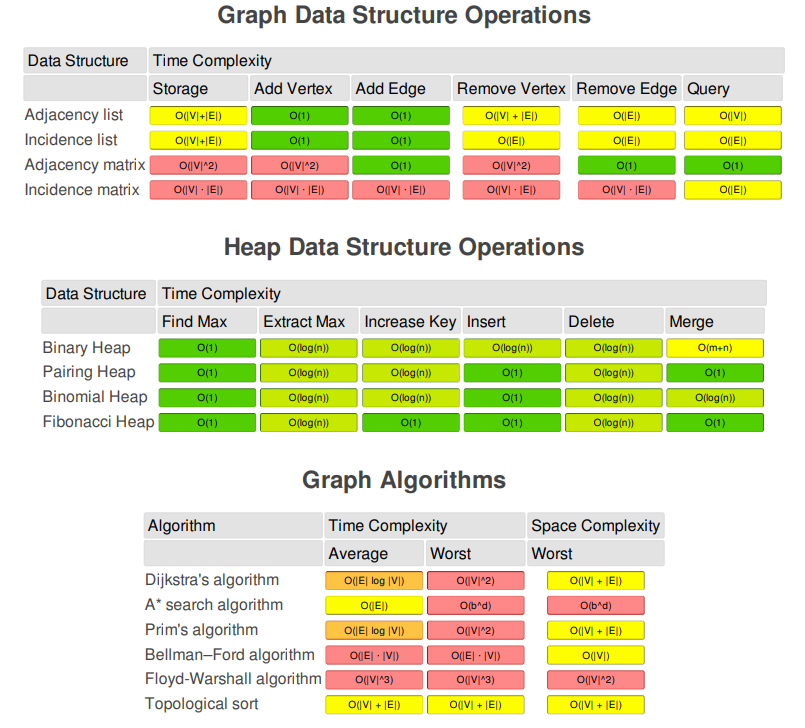
* **Inserting**: **O(1)**, if done at the head or tail, **O(n)** if anywhere else since we have to reach that position by traveseing the linkedlist linearly.
* **Deleting**: **O(1)**, if done at the head or tail, **O(n)** if anywhere else since we have to reach that position by traveseing the linkedlist linearly.
* **Searching**: **O(n)**

# Time Complexities

# Java Data Structures:

|  |  |  |
| --- | --- | --- |
| **Array List:**   * **Add**: **Amortized O(1)** * **Remove**: **O(n)** * **Contains**: **O(n)** * **Size**: **O(1)** | **Linked List:**   * **Inserting**: **O(1)**, if done at the head, **O(n)** if anywhere else. * **Deleting**: **O(1)**, if done at the head, **O(n)** if anywhere. * **Searching**: **O(n)** | **Doubly-Linked List:**   * **Inserting**: **O(1)**, if done at the head or tail, **O(n)** if anywhere else * **Deleting**: **O(1)**, if done at the head or tail, **O(n)** if anywhere else * **Searching**: **O(n)** |
| **Stack:**   * **Push**: **O(1)** * **Pop**: **O(1)** * **Top**: **O(1)** * **Search:** **O(n)** | **Queue/Deque/ Circular Queue:**   * **Insert**: **O(1)** * **Remove**: **O(1)** * **Size**: **O(1)** | **Binary Search Tree:**   * **Insert, delete and search**: Average case: **O(log n)**, Worst Case: **O(n)** |
| **Heap/PriorityQueue (min/max):**   * **Find Min/Find Max**: **O(1)** * **Insert**: **O(log n)** * **Delete Min/Delete Max**: **O(log n)** * **Extract Min/Extract Max**: **O(log n)** * **Lookup, Delete** (if at all provided): **O(n)**, we will have to scan all the elements as they are not ordered like BST | | **HashMap/ Hashtable/ HashSet:**   * **Insert/Delete**: **O(1)** amortized * **Re-size/hash**: **O(n)** * **Contains**: **O(1)** |



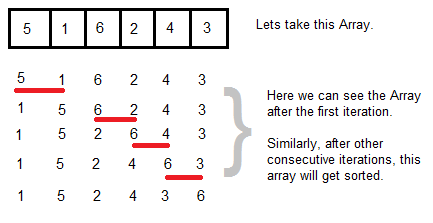


# Sorting

# Bubble sort

Is an algorithm which is used to sort **N** elements that are given in a memory for eg: an Array with **N** number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the largest element in the list bubbles up towards the last place, just like a water bubble rises up to the water surface.



int a[6] = {5, 1, 6, 2, 4, 3};

int i, j, temp;

for(i=0; i<6; i++)

{

for(j=0; j<6-i-1; j++)

{

if( a[j] > a[j+1])

{

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

//now you can print the sorted array after this

#### Complexity Analysis of Bubble Sort

**Worst Case Time Complexity : O(n^2)**

**Best Case Time Complexity : O(n)**

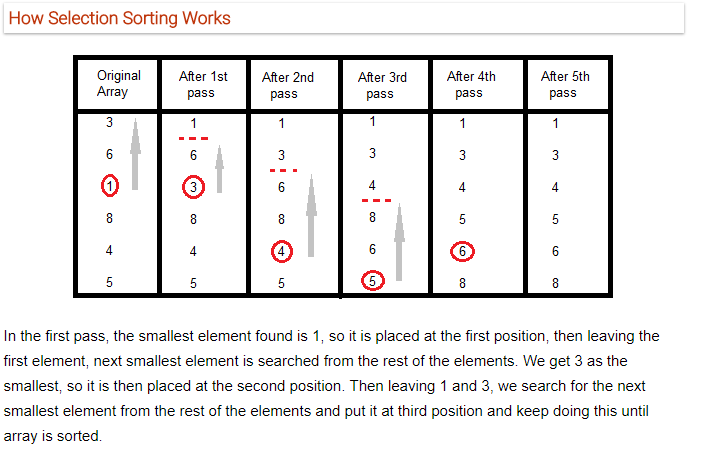
**Average Time Complexity : O(n^2)**

**Space Complexity : O(1)**

Stable algorithm

# Selection Sort

Selection sorting is conceptually the most simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.



void selectionSort(int a[], int size)

{

int i, j, min, temp;

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

{

min = i; //setting min as i

for(j=i+1; j < size; j++)

{

if(a[j] < a[min]) //if element at j is less than element at min position

{

min = j; //then set min as j

}

}

temp = a[i];

a[i] = a[min];

a[min] = temp;

}

}

**Worst Case Time Complexity : O(n2)**

**Best Case Time Complexity : O(n2)**

**Average Time Complexity : O(n2)**

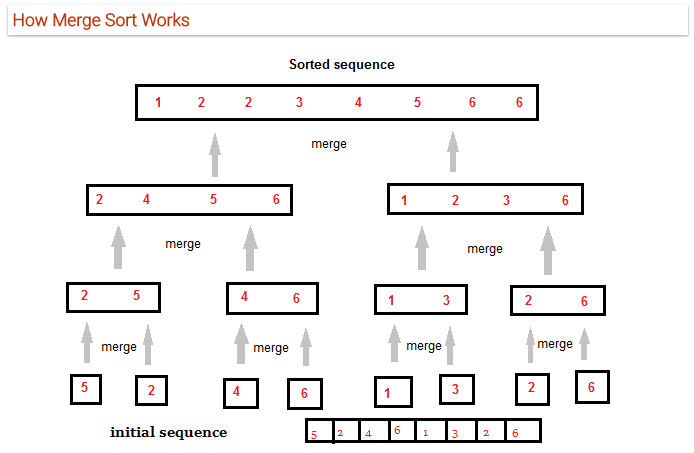
**Space Complexity : O(1)**

Algorithm is not stable: relative positions are swapped.  it might change the occurrence of two similar elements in the list while sorting.

# Merge Sort

Merge Sort follows the rule of **Divide and Conquer**. In merge sort the unsorted list is divided into N sublists, each having one element, because a list consisting of one element is always sorted. Then, it repeatedly merges these sublists, to produce new sorted sublists, and in the end, only one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of **O(n log n)**. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.



**void** mergeSort(**int** ary[], **int** low, **int** high) {  
 **if** (low < high) {  
 **int** mid = (low + high) / 2;  
 mergeSort(ary, low, mid);  
 mergeSort(ary, mid + 1, high);  
 merge(ary, low, mid, high);  
 }  
}

**void** merge(**int**[] ary, **int** low, **int** mid, **int** high) {  
 **int** i = low, j = mid + 1, k = low;  
 **int** temp[] = **new int**[high + 1];  
  
 **for** (**int** idx = low; idx <= high; idx++) {  
 temp[idx] = ary[idx];  
 }  
 **while** (i <= mid && j <= high) {  
 **if** (temp[i] <= temp[j]) {  
 ary[k++] = temp[i++];  
 } **else** {  
 ary[k++] = temp[j++];  
 }  
 }  
 **while** (i <= mid) {  
 ary[k++] = temp[i++];  
 }  
}

**Worst Case Time Complexity : O(n log n)**

**Best Case Time Complexity : O(n log n)**

**Average Time Complexity : O(n log n)**

**Space Complexity : O(n)**

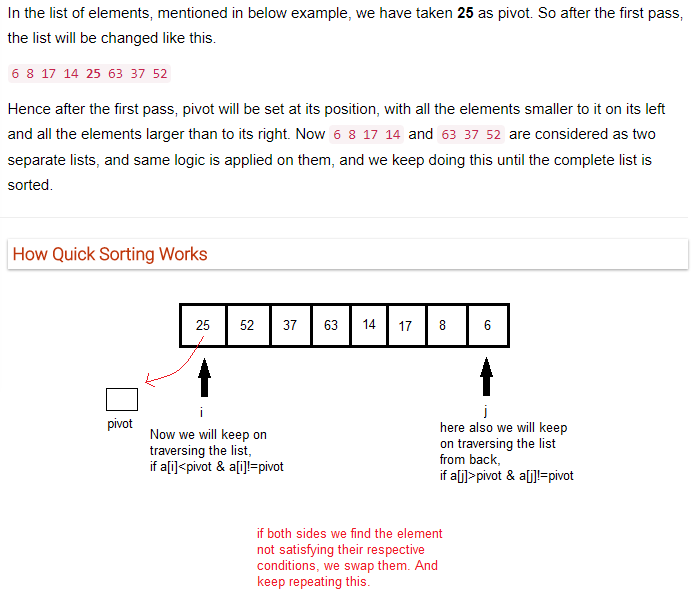
* Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
* It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.
* It is the best Sorting technique used for sorting **Linked Lists**.

stable sort

# Quick Sort

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not a stable search, but it is very fast and requires very less additional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :

1. Elements less than the Pivot element
2. Pivot element(Central element)
3. Elements greater than the pivot element



**Worst Case Time Complexity : O(n2)**

**Best Case Time Complexity : O(n log n)**

**Average Time Complexity : O(n log n)**

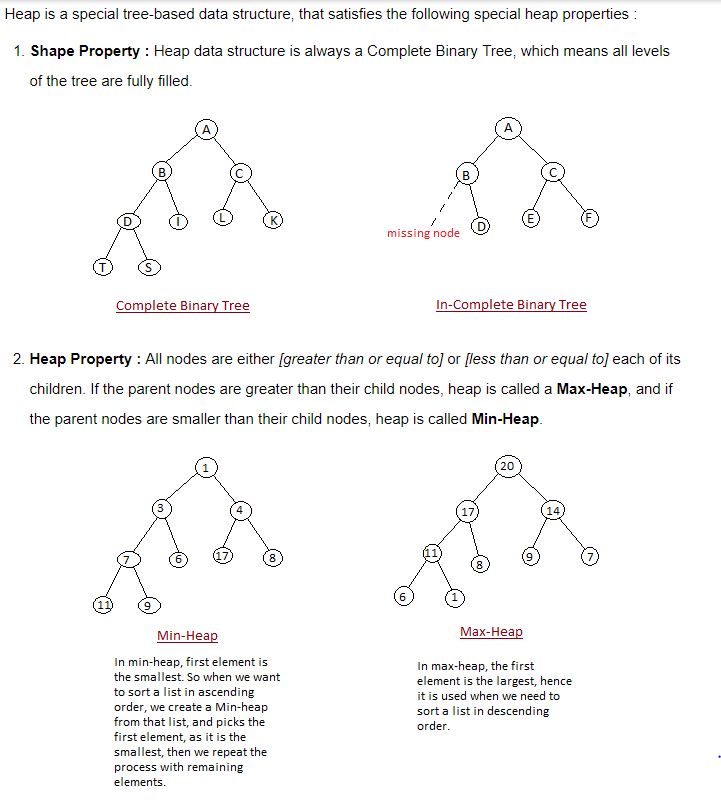
**Space Complexity : O(n log n)**

* Space required by quick sort is very less, only O(n log n) additional space is required.
* Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

# Heap Sort

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts :

* Creating a Heap of the unsorted list.
* Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.



#### How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

//function that sorts the array in ascending order using maxheap.  
void sort(int[] arr){  
 int n = arr.length;  
  
 //build the heap from bottom up first.  
 //go to the last parent in the tree and build from there.  
 for(int i=n/2-1; i>=0;i--)  
 heapify(arr,n,i);  
  
 //once max heap is found, swap the arr[0] with the last element.  
 //now again heapify the arr excluding the last element..continue this process  
 for(int i=n-1;i>=0;i--){  
 int temp = arr[0];  
 arr[0] = arr[i];  
 arr[i] = temp;  
  
 //heapify the 0th position  
 heapify(arr,i,0);  
 }  
}  
  
//maxheap  
//To heapify a subtree rooted with node i which is  
 // an index in arr[]. n is size of heap  
void heapify(int[] arr,int n,int i){  
  
 int largest = i;  
 int left = 2\*i+1; //left child  
 int right = 2\*i+2; //right child  
  
 //check if parent is the largest, else swap  
 if(left<n && arr[left]> arr[largest])  
 largest = left;  
 if(right<n && arr[right]>arr[largest])  
 largest = right;  
  
 //see if the swap needs to be done  
 if(largest != i){  
 int swap = arr[largest];  
 arr[largest] = arr[i];  
 arr[i] = swap;  
  
 //now heapify the swapped position recursively  
 heapify(arr,n,largest);  
 }  
}

**Worst Case Time Complexity : O(n log n)**

**Best Case Time Complexity : O(n log n)**

**Average Time Complexity : O(n log n)**

**Space Complexity : O(1)**

Heap sort is not a Stable sort, and requires a constant space for sorting a list.

Heap Sort is very fast and is widely used for sorting.

# Sorting Summary

