Bootcamp

Data Structures and Algorithms

- **Rama Bhadra Raju**
  IITM Alum | Data Scientist @Cargill

Data Types

Primitive Datatypes - are typically types that are built-in or basic to a language implementation.

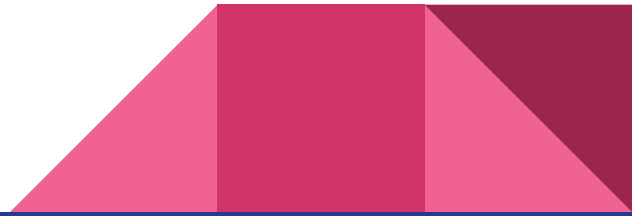
Eg- int, float, char, bool, pointers

Abstract Datatypes - Any data type that does not expatiate on the concrete representation of the data. Instead, a formal specification based on the data type's operations is used to describe it.

Eg- List ADT, Stack ADT

Data structure is a particular way of organizing data in a computer so that it can be used effectively.

Eg - Array List, Linked list



Time complexity

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

```
int array[]= {1,2,3,4,5};
//func to retrieve first element in array
public int getFirstElement()
{
return array[0];
}
```

No. of computations to perform getFirstElement() is **constant**

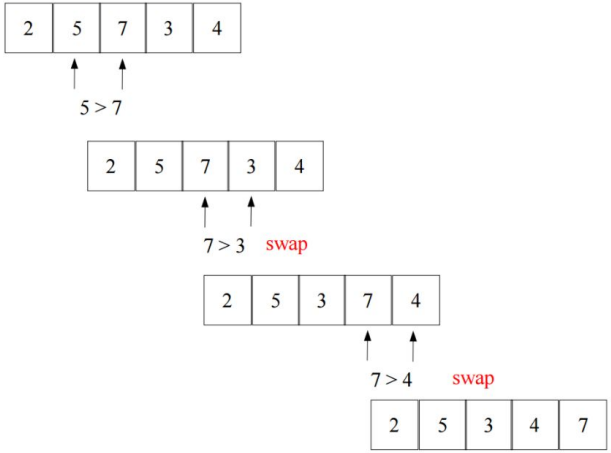
Time complexity

```
int array[]= {1,2,3,4,5};
//func to find sum of array elements
public int arraySum(int arr[])
{
int sum = 0; // initialize sum
for (int i = 0; i < arr.length; i++){
    sum +=  arr[i];
}
return sum;
}
```

No. of computations to perform arraySum() $\propto$ array length

Time complexity

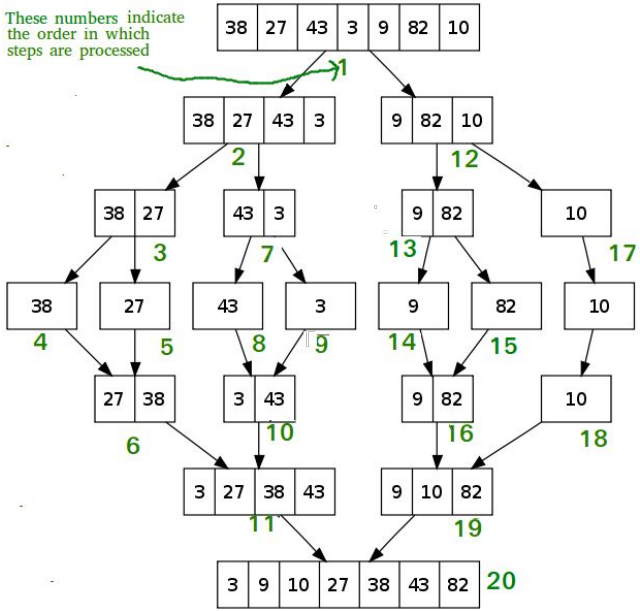
Bubble sort algorithm




```
int n = arr.length;
for (int i = 0; i < n-1; i++)
      for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1]){
                  int temp = arr[j];
                  arr[j] = arr[j+1];
                  arr[j+1] = temp;
                }
        }
   }
```

No. of computations to perform Bubble sort $\propto N^2$

Merge sort Time complexity



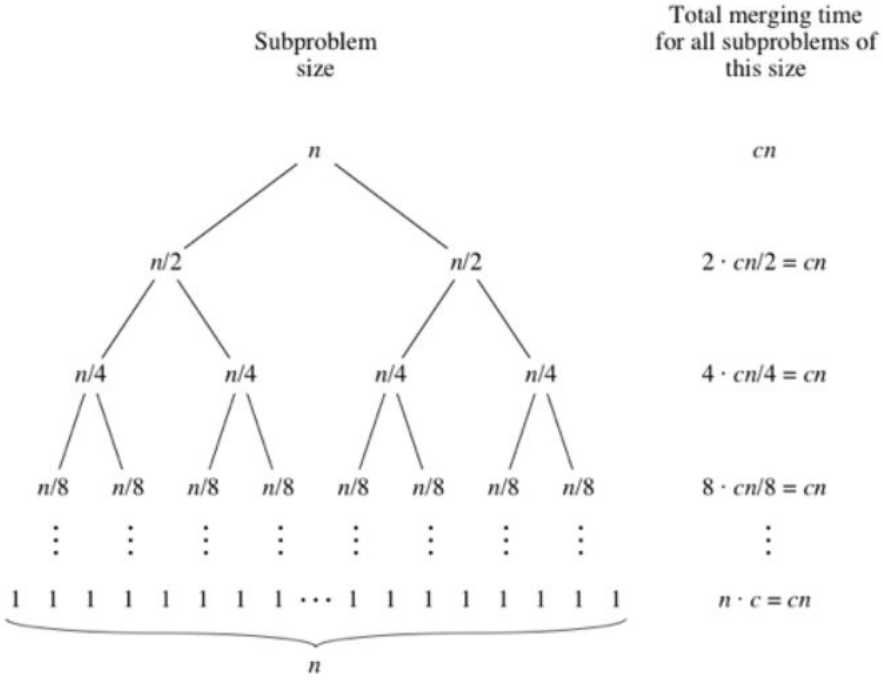
No. of computations to perform Merge sort $\propto$ **(n*Log n)**

How to compare time Complexities of two algorithms?

Merge sort Time complexity

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = l+ (r-l)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
            Call merge(arr, l, m, r)
```

Merge sort Time complexity

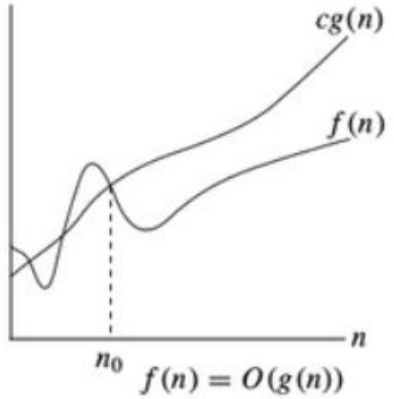


Big O notation

Let g and f be functions from the set of natural numbers to itself. The function f(n) = O(g(n)) such that,

if there exists a positive integer n_0 and a positive constant c, such that

$$f(n) \leq c.g(n) \quad \forall \; n \geq n_0$$



Big-O Notation gives us the Upper Bound Idea, mathematically.

Big O notation

```
statement;

for(i=0; i < N; i++)
{
    statement;
}

for(i=0; i < N; i++)
{
    for(j=0; j < N;j++)
    {
    statement;
    }
}
```

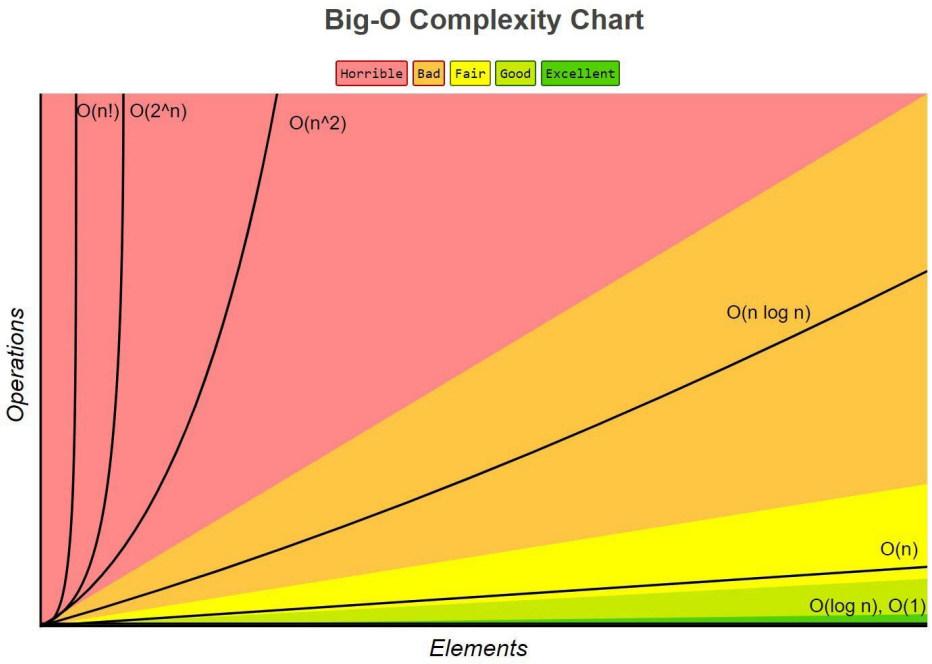
Actual time complexity of this code be

$$f(n) = c_1 + c_2 * N + c_3 * N^2$$

But in Big O notation,
Time Complexity is $O(N^2)$

Writing BigO for any function

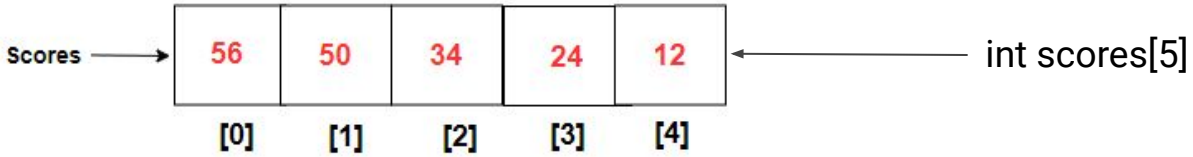
1. Drop lower growth rate terms
2. Drop constant multipliers



Notation	Type	Examples	Description
O(1)	Constant	Hash table access	Remains constant regardless of the size of the data set
O(log n)	Logarithmic	Binary search of a sorted table	Increases by a constant. If n doubles, the time to perform increases by a constant, smaller than n amount
O(< n)	Sublinear	Search using parallel processing	Performs at less than linear and more than logarithmic levels
O(n)	Linear	Finding an item in an unsorted list	Increases in proportion to n. If n doubles, the time to perform doubles
O(n log(n))	n log(n)	Quicksort, Merge Sort	Increases at a multiple of a constant
$O(n^2)$	Quadratic	Bubble sort	Increases in proportion to the product of n*n
$O(c^n)$	Exponential	Travelling salesman problem solved using dynamic programming	Increases based on the exponent n of a constant c
O(n!)	Factorial	Travelling salesman problem solved using brute force	Increases in proportion to the product of all numbers included (e.g., 1*2*3*4...)

Static Array

Static arrays are fixed in size i.e. when declared, a constant contiguous memory is allocated and does not change its size for the rest of the program during run time.

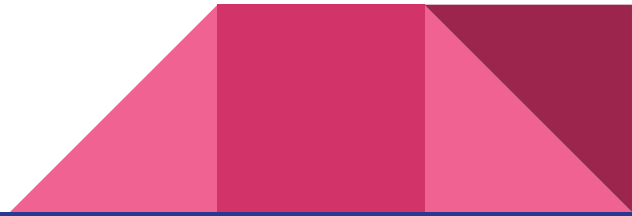


*Integer index starts at 0 and goes up one at a time. Hence, it is called zero-based indexing

*With index nos, we can easily access an element easily
Scores[0] = 56
Scores[1] = 50
Scores[2] = 34

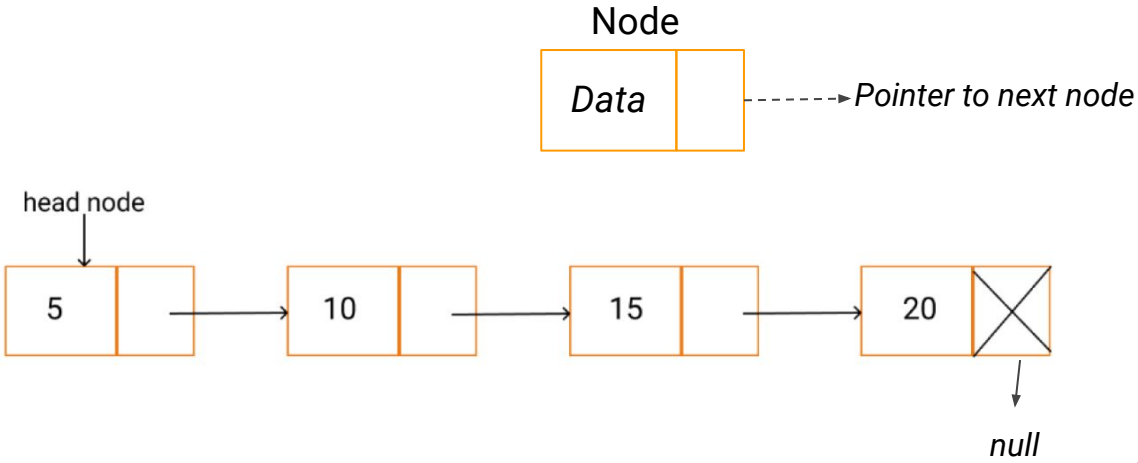
Dynamic Array

- **Dynamic array** (vector in C++, ArrayList in Java) automatically grows when we try to make an insertion when there is no more space left for the new item. Usually the area doubles in size.
- Usually involves three main functions
  1. push()
  2. pop()
  3. size()



Linked lists

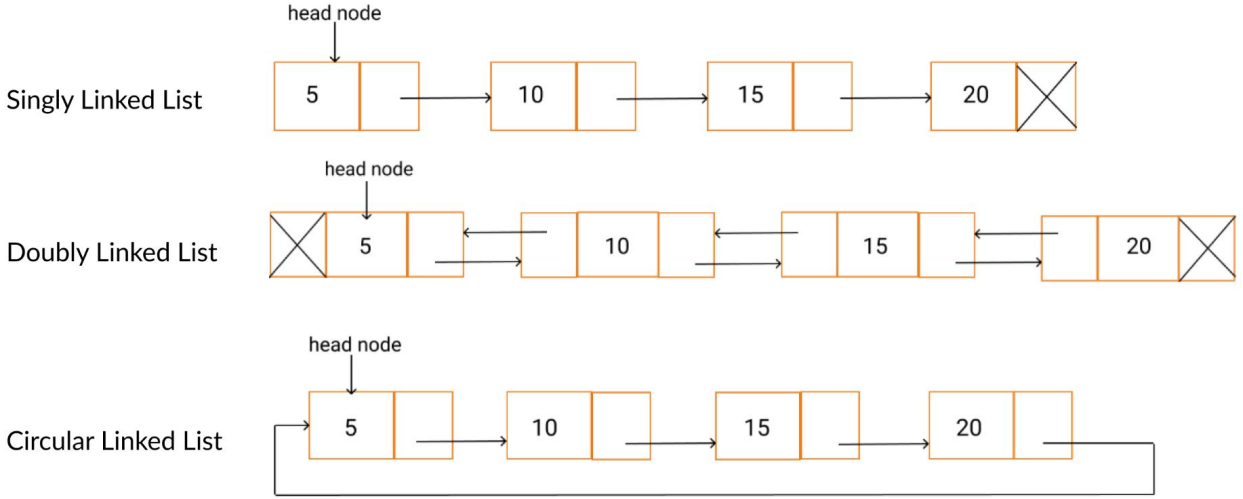
- Linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.
- *Linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.*



Linked list Structure

```
// A linked list node
static class Node {
        int data;
        Node next;
        // Constructor
        Node(int d)
        {
            data = d;
            next =  null;
        }
    }
```

Linked lists



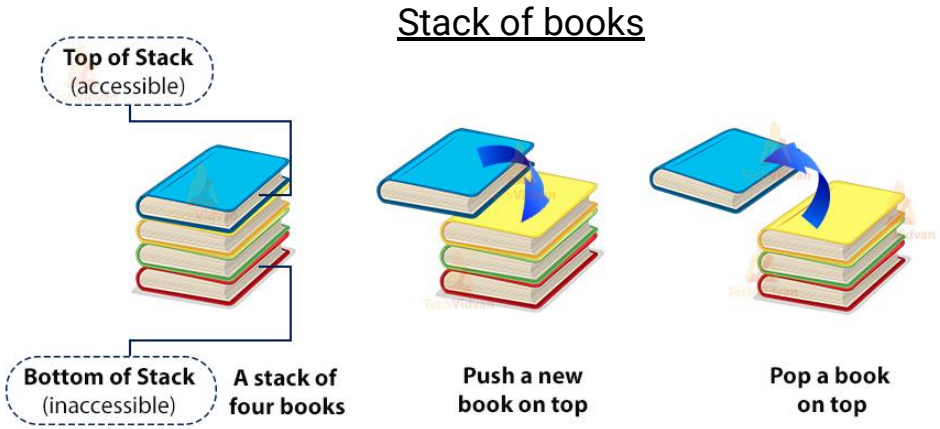
Types of Linked lists:

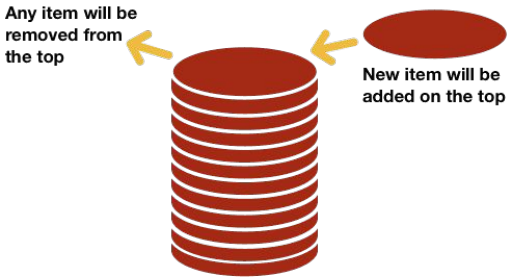
1. **Single Linked List -** *node contains a data field and a pointer to the next node*
2. **Circular Linked List** - *a linked list where all nodes are connected to form a circle*
3. **Doubly Linked List** - *contains an extra pointer, typically called previous pointer, together with next pointer and data*

Linkedlist vs ArrayList Time Complexity

Operation	LinkedList time complexity	ArrayList time complexity	Preferred
Insert at last index	O(1)	O(1) (If array copy operation is Considered then O(N))	LinkedList
Insert at given index	O(N)	O(N)	LinkedList
Search by value	O(N)	O(N)	ArrayList
Get by index	O(N)	O(1)	ArrayList
Remove by value	O(N)	O(N)	LinkedList
Remove by index	O(N)	O(N)	LinkedList

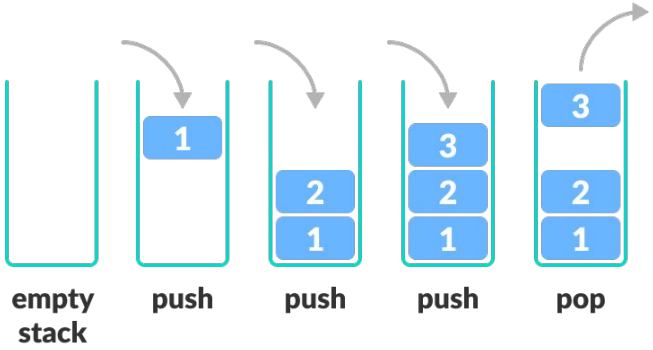
Stack





- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order is **LIFO(**Last In First Out) or **FILO**(First In Last Out).

Stack operations

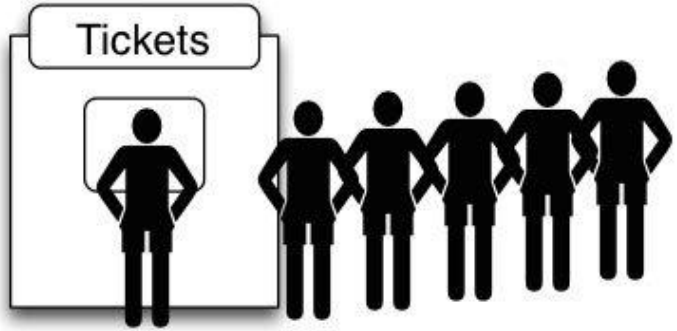


Mainly the following basic operations are performed in the stack

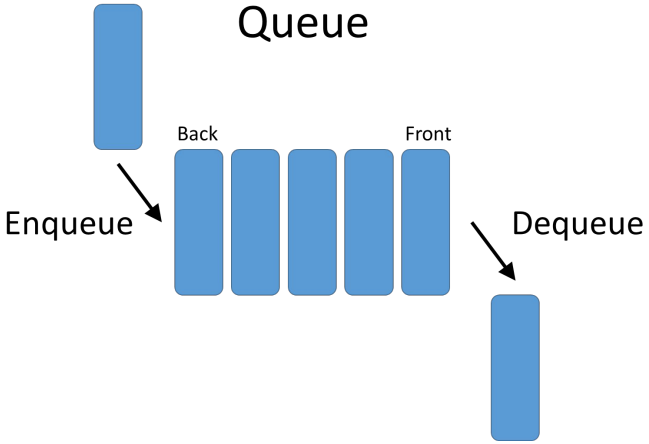
1. **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
2. **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
3. **Peek or Top:** Returns top element of stack.
4. **isEmpty:** Returns true if stack is empty, else false

Queue

- Queue is a linear structure which follows a particular order in which the operations are performed.
- The order is **First In First Out** (FIFO)



Queue operations

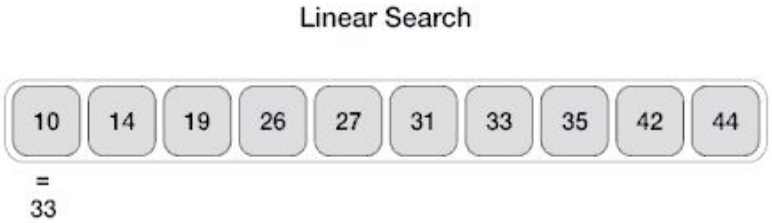


Mainly the following basic operations are performed in the stack

1. **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
2. **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.
3. **Front:** Get the front item from queue.
4. **Rear:** Get the last item from queue.

Binary Search Algorithm

- Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].
- Sorted array - elements arranged in numerical order



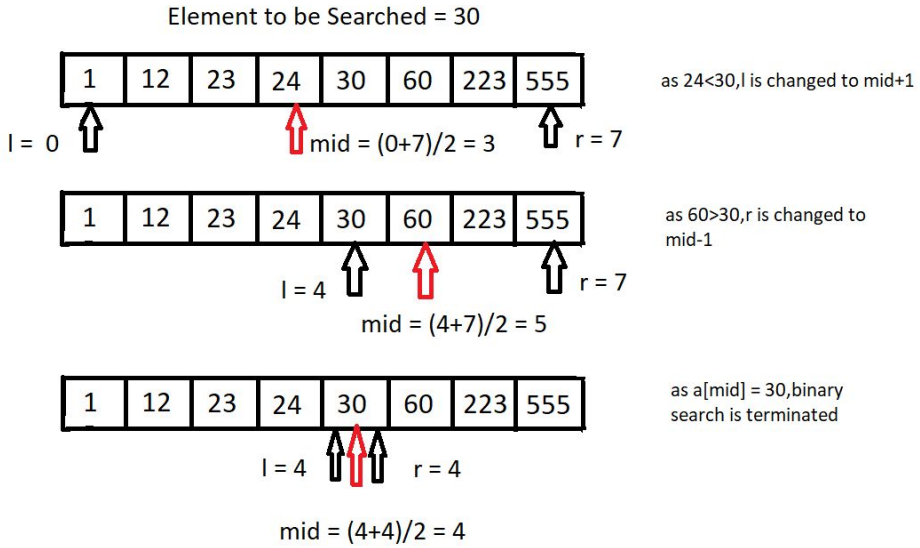
★ Simple approach is to do a **Linear Search** - *Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the array*

★ Time complexity is **O(n)**

Binary Search perform same task with less time complexity **O(log n)**

Binary Search Algorithm

Binary Search: Search a sorted array by repeatedly dividing the search interval in half.



Steps

1. Compare x with the middle element
2. If x matches with the middle element, we return the mid index
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half
4. Else (x is smaller) recur for the left half

References

- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort
- https://dzone.com/articles/performance-analysis-of-arraylist-and-linkedlist-i
- Linked List | Set 1 (Introduction) - GeeksforGeeks
- https://www.geeksforgeeks.org/how-to-implement-our-own-dynamic-array-class-in-java/
- https://www.javatpoint.com/dynamic-array-in-java
- https://www.sanfoundry.com
- stackoverflow

Thank You