Introduction

Data Structures are entities used in programming, which can store some form of data in some **ordered form**, which allows us to perform some **efficient processing** on them. The efficient processing can be in terms of **time, space, or both,** or it can be based on some **other factor** as a priority that is needed for some specific problem.

Data structures can be divided into 2 types:

- Primitive Data Structures: int, bool, float, etc.
- Abstract Data Structures: Tree, LinkedList, etc.

In terms of memory representation and structure, it can also be classified into 2 types:

- Linear Data Structures: Arrays.
- Non-Linear Data Structures: LinkedList, Trees, etc.

Algorithms are some well-defined sets of **instructions, steps, or logic**, written such that by following the steps, some specific tasks are accomplished. It is not the working program in itself, but rather a **well-defined series of steps**, encompassing the program's underlying logic, following which the working program can be coded down. Algorithms must satisfy the following properties:

- Input: There should be >= 0 inputs given to the algorithm to work on.
- Output: The algorithm must provide some form of output.
- Finiteness: The algorithm should have a finite number of steps.
- Definiteness: Every step of the algorithm should be well-defined and not be ambiguous.
- Correctness: The algorithm must provide a correct output.

The **efficiency** of the algorithm is also measured on various parameters, the most important of them being:

- Time Complexity.
- Space Complexity.

Algorithms are **varied and vast**, and every new program can be classified into a different category of algorithms, but some of the famous used examples are **Flows, Lowest Common Ancestor of Nodes, Sorting, Searching, etc.**

What is a Data Structure?

Data Structure is a way to organized data in such a way that it can be used efficiently. Following terms are foundation terms of a data structure.

- **Interface** − Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
- **Implementation** − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

Characteristics of a Data Structure

- **Correctness** − Data Structure implementation should implement its interface correctly.
- **Time Complexity** − Running time or execution time of operations of data structure must be as small as possible.
- **Space Complexity** − Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems applications face now-a-days.

- **Data Search** − Consider an inventory of 1 million (10^6) items of a store. If application is to search an item. It has to search item in 1 million (10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** − Processor speed although being very high, falls limited if data grows to billon records.
- **Multiple requests** − As thousands of users can search data simultaneously on a web server, even very fast server fails while searching the data.

To solve above problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be search and required data can be searched almost instantly.

Execution Time Cases

There are three cases which are usual used to compare various data structure's execution time in relative manner.

- **Worst Case** − This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst-case time is ƒ(n) then this operation will not take time more than ƒ(n) time where ƒ(n) represents function of n.
- **Average Case** − This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution then m operations will take mƒ(n) time.
- **Best Case** − This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution then actual operation may take time as random number which would be maximum as ƒ(n).

Algorithm concept

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in certain order to get the desired output. In term of data structures, following are the categories of algorithms.

- **Search** − Algorithms to search an item in a data structure.
- **Sort** − Algorithms to sort items in certain order
- **Insert** − Algorithm to insert item in a data structure
- **Update** − Algorithm to update an existing item in a data structure
- **Delete** − Algorithm to delete an existing item from a data structure

Algorithm analysis

Algorithm analysis deals with the execution time or running time of various operations of a data structure. Running time of an operation can be defined as no. of computer instructions executed per operation. As exact running time of any operation varies from one computer to another computer, we usually analyse the running time of any operation as some function of n, where n is the no. of items processed in that operation in a data structure.

Asymptotic analysis

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, running time of one operation is computed as $f(n)$ and of another operation as $g(n^2)$. Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly same if n is significantly small.

Asymptotic Notations

Following is commonly used asymptotic notations used in calculating running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete. For example, for a function *f*(n)

O(*f*(n)) = { *g*(n) : there exists c > 0 and n_0 such that *g*(n) ≤ c.*f*(n) for all n > n_0. }

Big Oh notation is used to simplify functions. For example, we can replace a specific functional equation 7nlogn + n - 1 with O(*f*(nlogn)). Consider the scenario as follows:

7nlogn +n - 1 ≤ 7nlogn + n

7nlogn +n - 1 ≤ 7nlogn + nlogn

for n ≥ 2 so that logn ≥ 1

7nlogn +n - 1 ≤ 8nlogn

It demonstrates that *f*(n) = 7nlogn + n - 1 is within the range of outout of *O*(nlogn) using constants c = 8 and n_0 = 2.

Omega Notation, Ω

The Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.

For example, for a function *f*(n)

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n_0 such that *g*(n) ≤ c.*f*(n) for all n > n_0. }

Theta Notation, θ

The θ(n) is the formal way to express both the lower bound and upper bound of an algorithm's running time. It is represented as following.

θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n_0. }

Data Structure is a way to organized data in such a way that it can be used efficiently. Following terms are basic terms of a data structure.

Data Definition

Data Definition defines a particular data with following characteristics.

- Atomic − Definition should define a single concept
- Traceable − Definition should be able to be mapped to some data element.
- Accurate − Definition should be unambiguous.
- Clear and Concise − Definition should be understandable.

Data Object

Data Object represents an object having a data.

Data Type

Data type is way to classify various types of data such as integer, string etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. Data type of two types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provides following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or other way are known as derived data types. These data types are normally built by combination of primary or built-in data types and associated operations on them. For example −

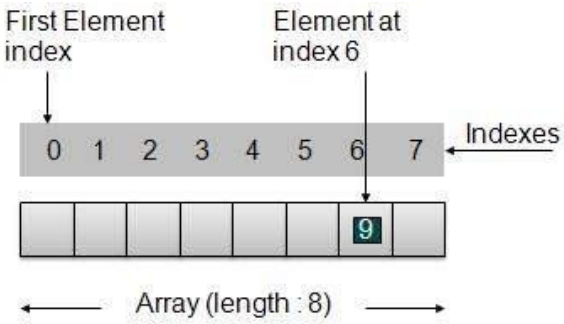
- List
- Array
- Stack
- Queue

Array Basics

Array is a container which can hold fix number of items and these items should be of same type. Most of the data structure make use of array to implement their algorithms. Following are important terms to understand the concepts of Array

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index which is used to identify the element.

Array Representation



- Index starts with 0.
- Array length is 8 which means it can store 8 elements.
- Each element can be accessed via its index. For example, we can fetch element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Insertion** − add an element at given index.
- **Deletion** − delete an element at given index.
- **Search** − search an element using given index or by value.
- **Update** − update an element at given index.

In java, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
Object	null

Linked List Basics

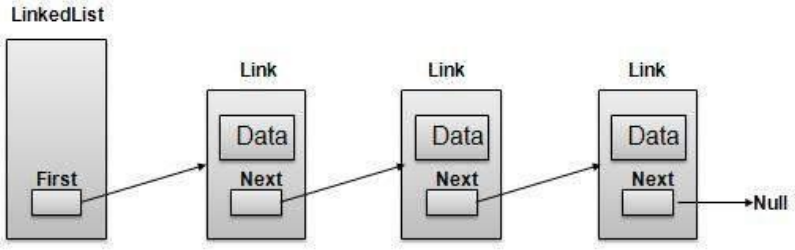
Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List.

Link − Each Link of a linked list can store a data called an element.

Next − Each Link of a linked list contain a link to next link called Next.

LinkedList − A LinkedList contains the connection link to the first Link called First.

Linked List Representation



- LinkedList contains an link element called first.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Last Link carries a Link as null to mark the end of the list.

Types of Linked List

Following are the various flavours of linked list.

- **Simple Linked List** − Item Navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward way.
- **Circular Linked List** − Last item contains link of the first element as next and and first element has link to last element as prev.

Basic Operations

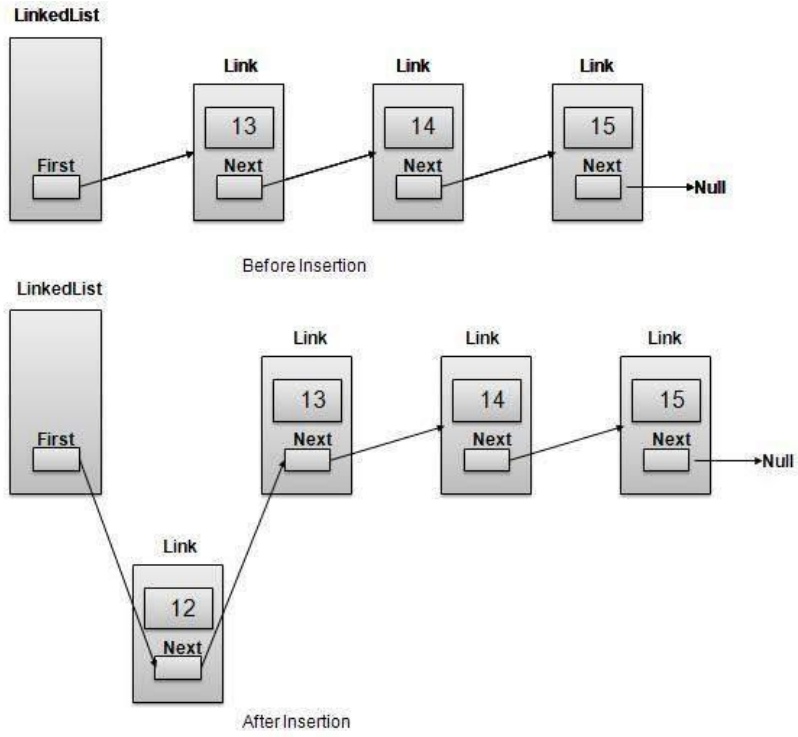
Following are the basic operations supported by a list.

- **Insertion** − add an element at the beginning of the list.
- **Deletion** − delete an element at the beginning of the list.
- **Display** − displaying complete list.
- **Search** − search an element using given key.
- **Delete** − delete an element using given key.

Insertion Operation

Insertion is a three-step process:

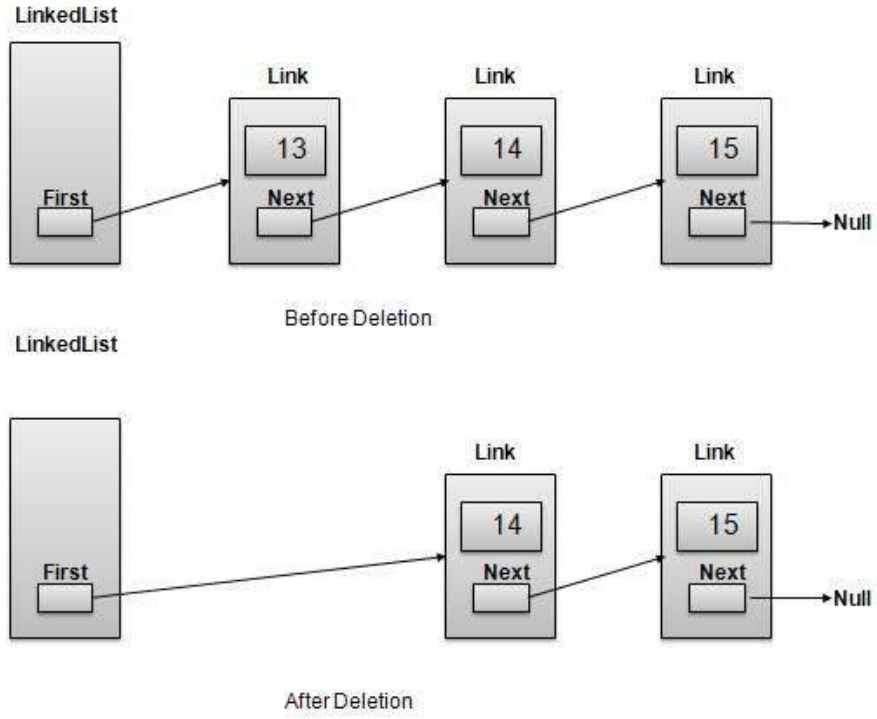
1. Create a new Link with provided data.
2. Point New Link to old First Link.
3. Point First Link to this New Link.



Deletion Operation

Deletion is a two-step process:

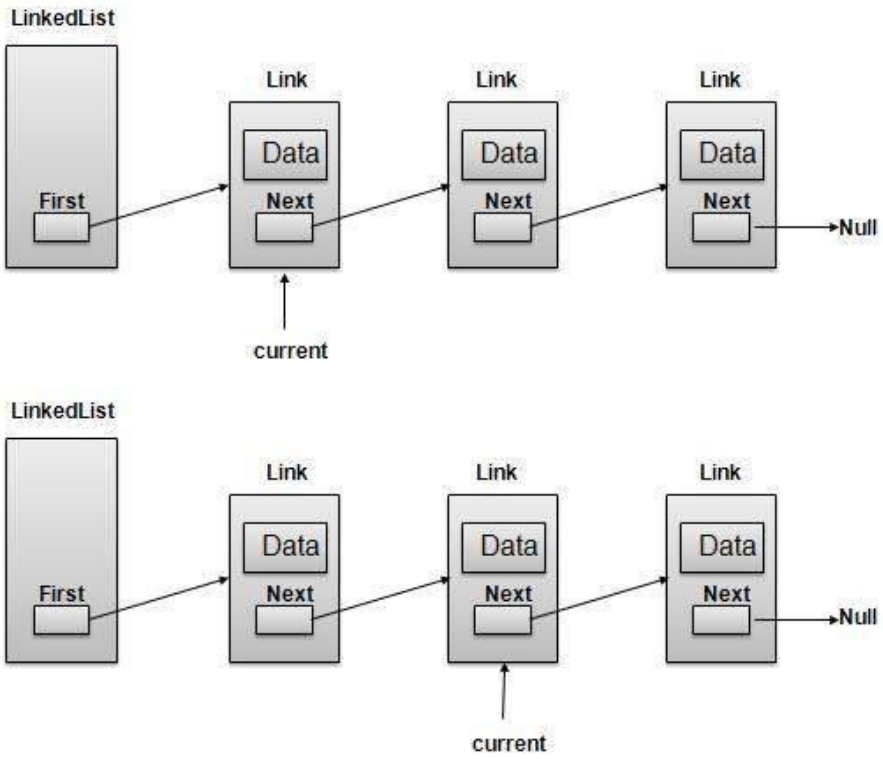
1. Get the Link pointed by First Link as Temp Link.
2. Point First Link to Temp Link's Next Link.



Navigation Operation

Navigation is a recursive step process and is basis of many operations like search, delete etc.:

1. Get the Link pointed by First Link as Current Link.
2. Check if Current Link is not null and display it.
3. Point Current Link to Next Link of Current Link and move to above step.



Advanced Operations

Following are the advanced operations specified for a list.

- **Sort** − sorting a list based on a particular order.
- **Reverse** − reversing a linked list.
- **Concatenate** − concatenate two lists.

Sort Operation

We've used bubble sort to sort a list.

```
public void sort(){

   int i, j, k, tempKey, tempData ;
   Link current,next;
   int size = length();
   k = size ;
   for ( i = 0 ; i < size - 1 ; i++, k-- ) {
      current = first ;
      next = first.next ;
      for ( j = 1 ; j < k ; j++ ) {
         if ( current.data > next.data ) {
            tempData = current.data ;
            current.data = next.data;
            next.data = tempData ;
```

```
            tempKey = current.key;
            current.key = next.key;
            next.key = tempKey;
         }
         current = current.next;
         next = next.next;
      }
   }
}
```

Reverse Operation

Following code demonstrate reversing a single linked list.

```
public LinkedList reverse() {
   LinkedList reversedlist = new LinkedList();
   Link nextLink = null;
   reversedlist.insertFirst(first.key, first.data);

   Link currentLink = first;
   // Until no more data in list,
   // insert current link before first and move ahead.
   while(currentLink.next != null){
      nextLink = currentLink.next;
      // Insert at start of new list.
      reversedlist.insertFirst(nextLink.key, nextLink.data);
      //advance to next node
      currentLink = currentLink.next;
   }
   return reversedlist;
}
```

Concatenate Operation

Following code demonstrate reversing a single linked list.

```
public void concatenate(LinkedList list){
   if(first == null){
      first = list.first;
   }
   if(list.first == null){
      return;
   }
   Link temp = first;
   while(temp.next !=null) {
      temp = temp.next;
   }
   temp.next = list.first;
}
```

Doubly Linked List Basics

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways either forward and backward easily as compared to Single Linked List. Following are important terms to understand the concepts of doubly Linked List

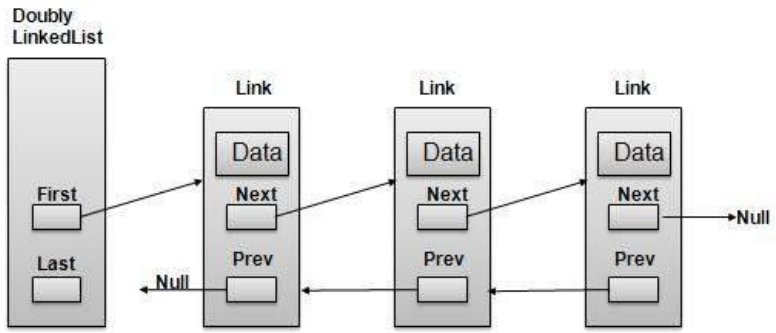
Link − Each Link of a linked list can store a data called an element.

Next − Each Link of a linked list contain a link to next link called Next.

Prev − Each Link of a linked list contain a link to previous link called Prev.

LinkedList − A LinkedList contains the connection link to the first Link called First and to the last link called Last.

Doubly Linked List Representation:



- Doubly LinkedList contains an link element called first and last.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Each Link is linked with its previous link using its prev link.
- Last Link carries a Link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by an list.

- **Insertion** − add an element at the beginning of the list.
- **Deletion** − delete an element at the beginning of the list.
- **Insert Last** − add an element in the end of the list.
- **Delete Last** − delete an element from the end of the list.
- **Insert After** − add an element after an item of the list.
- **Delete** − delete an element from the list using key.
- **Display forward** − displaying complete list in forward manner.
- **Display backward** − displaying complete list in backward manner.

Insertion Operation

Following code demonstrate insertion operation at beginning in a doubly linked list.

```
//insert link at the first location
public void insertFirst(int key, int data){
   //create a link
   Link link = new Link(key,data);

   if(isEmpty()){
      //make it the last link
      last = link;
   }else {
      //update first prev link
      first.prev = link;
   }

   //point it to old first link
   link.next = first;
   //point first to new first link
   first = link;
}
```

Deletion Operation

Following code demonstrate deletion operation at beginning in a doubly linked list.

```
//delete link at the first location
public Link deleteFirst(){
   //save reference to first link
   Link tempLink = first;
   //if only one link
   if(first.next == null){
      last = null;
   }else {
      first.next.prev = null;
   }
   first = first.next;
   //return the deleted link
   return tempLink;
}
```

Insertion at End Operation

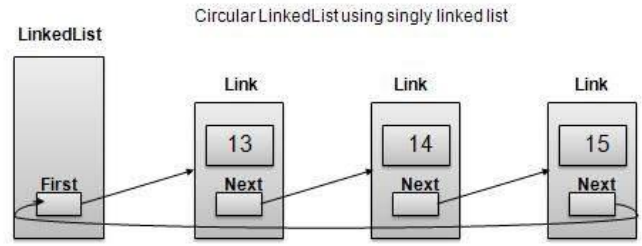
Following code demonstrate insertion operation at last position in a doubly linked list.

```
//insert link at the last location
public void insertLast(int key, int data){
   //create a link
   Link link = new Link(key,data);

   if(isEmpty()){
      //make it the last link
      last = link;
   }else {
      //make link a new last link
      last.next = link;
      //mark old last node as prev of new link
      link.prev = last;
   }

   //point last to new last node
   last = link;
}
```

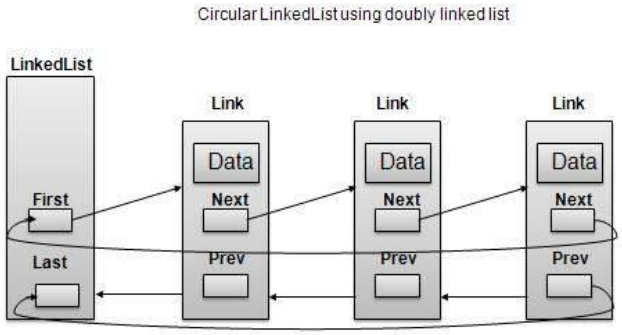
Circular Linked List Basics

Circular Linked List is a variation of Linked list in which first element points to last element and last element points to first element. Both Singly Linked List and Doubly Linked List can be made into as circular linked list

Singly Linked List as Circular



Doubly Linked List as Circular



- Last Link's next points to first link of the list in both cases of singly as well as doubly linked list.
- First Link's Prev points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- **insert** – insert an element in the start of the list.
- **delete** – insert an element from the start of the list.
- **display** – display the list.

length Operation

Following code demonstrate insertion operation at in a circular linked list based on single linked list.

```
//insert link at the first location
public void insertFirst(int key, int data){
   //create a link
   Link link = new Link(key,data);
   if (isEmpty()) {
      first  = link;
      first.next = first;
   }
   else{
      //point it to old first node
      link.next = first;
      //point first to new first node
      first = link;
   }
}
```

Deletion Operation

Following code demonstrate deletion operation at in a circular linked list based on single linked list.

```
//delete link at the first location
public Link deleteFirst(){
   //save reference to first link
   Link tempLink = first;
   //if only one link
   if(first.next == null){
      last = null;
   }else {
      first.next.prev = null;
   }
   first = first.next;
   //return the deleted link
   return tempLink;
}
```

Display List Operation

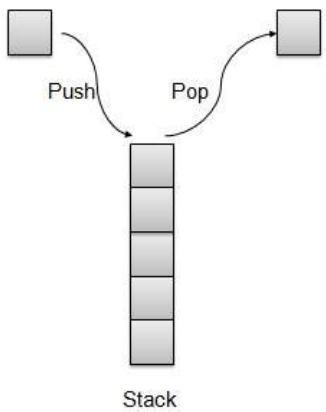
Following code demonstrate display list operation in a circular linked list.

```
public void display(){
   //start from the beginning
   Link current = first;
   //navigate till the end of the list
   System.out.print("[ ");
   if(first != null){
      while(current.next != current){
         //print data
         current.display();
         //move to next item
         current = current.next;
         System.out.print(" ");
      }
   }
   System.out.print(" ]");
}
```

Stack

Stack is kind of data structure which allows operations on data only at one end. It allows access to the last inserted data only. Stack is also called LIFO (Last In First Out) data structure and Push and Pop operations are related in such a way that only last item pushed (added to stack) can be popped (removed from the stack).

Stack Representation



Basic Operations

Following are two primary operations of a stack which are following.

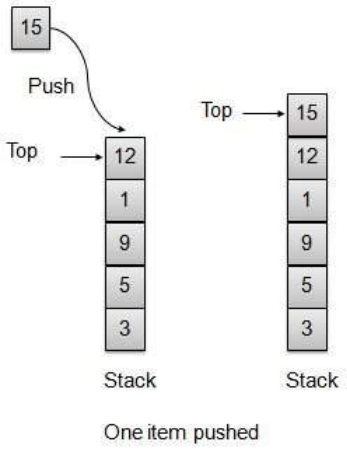
- **Push** − push an element at the top of the stack.
- **Pop** − pop an element from the top of the stack.

There is few more operations supported by stack which are following.

- **Peek** − get the top element of the stack.
- **isFull** − check if stack is full.
- **isEmpty** − check if stack is empty.

Push Operation

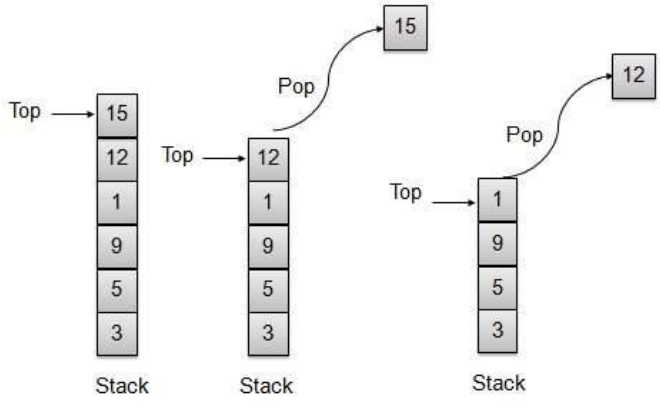
Whenever an element is pushed into stack, stack stores that element at the top of the storage and increments the top index for later use. If storage is full then an error message is usually shown.



One item pushed

Pop Operation

Whenever an element is to be popped from stack, stack retrieves the element from the top of the storage and decrements the top index for later use.



Two Items popped

Ordinary arithmetic expressions like 2*(3*4) are easier for human mind to parse but for an algorithm it would be pretty difficult to parse such an expression. To ease this difficulty, an arithmetic expression can be parsed by an algorithm using a two-step approach.

- Transform the provided arithmetic expression to postfix notation.
- Evaluate the postfix notation.

Infix Notation

Normal arithmetic expression follows Infix Notation in which operator is in between the operands. For example, A+B here A is first operand, B is second operand and + is the operator acting on the two operands.

Postfix Notation

Postfix notation varies from normal arithmetic expression or infix notation in a way that the operator follows the operands. For example, consider the following examples

Sr.No	**Infix Notation**	**Postfix Notation**
1	A+B	AB+
2	(A+B) *C	AB+C*
3	A*(B+C)	ABC+*
4	A/B+C/D	AB/CD/+
5	(A+B) *(C+D)	AB+CD+*
6	((A+B) *C)-D	AB+C*D-

Infix to Postfix Conversion

Before looking into the way to translate Infix to postfix notation, we need to consider following basics of infix expression evaluation.

- Evaluation of the infix expression starts from left to right.
- Keep precedence in mind, for example * has higher precedence over +. For example,
    - 2+3*4 = 2+12.
    - 2+3*4 = 14.
- Override precedence using brackets, For example
    - (2+3)*4 = 5*4.
    - (2+3)*4= 20.

Now let us transform a simple infix expression A+B*C into a postfix expression manually.

Step	**Character read**	**Infix Expressed parsed so far**	**Postfix expression developed so far**	**Remarks**
1	A	A	A	
2	+	A+	A	
3	B	A+B	AB	

4	*	A+B*	AB	+ can not be copied as * has higher precedence.
5	C	A+B*C	ABC	
6		A+B*C	ABC*	copy * as two operands are there B and C
7		A+B*C	ABC*+	copy + as two operands are there BC and A

Now let us transform the above infix expression A+B*C into a postfix expression using stack.

Step	**Character read**	**Infix Expressed parsed so far**	**Postfix expression developed so far**	**Stack Contents**	**Remarks**
1	A	A	A		
2	+	A+	A	+	push + operator in a stack.
3	B	A+B	AB	+	
4	*	A+B*	AB	+*	Precedence of operator * is higher than +. push * operator in the stack. Otherwise, + would pop up.
5	C	A+B*C	ABC	+*	
6		A+B*C	ABC*	+	No more operand, pop the * operator.
7		A+B*C	ABC*+		Pop the + operator.

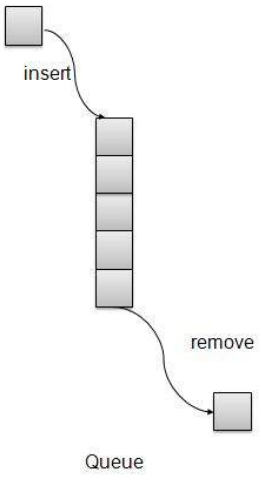
Now let us see another example, by transforming infix expression A*(B+C) into a postfix expression using stack.

Step	Character read	Infix Expressed parsed so far	Postfix expression developed so far	Stack Contents	Remarks
1	A	A	A		
2	*	A*	A	*	push * operator in a stack.
3	(	A*(	A	*(	push (in the stack.
4	B	A*(B	AB	*(	
5	+	A*(B+	AB	*(+	push + in the stack.
6	C	A*(B+C	ABC	*(+	
7	)	A*(B+C)	ABC+	*(	Pop the + operator.
8		A*(B+C)	ABC+	*	Pop the (operator.
9		A*(B+C)	ABC+*		Pop the rest of the operator(s).

Queue

Queue is kind of data structure similar to stack with primary difference that the first item inserted is the first item to be removed (FIFO - First in First Out) where stack is based on LIFO, Last in First Out principal.

Queue Representation



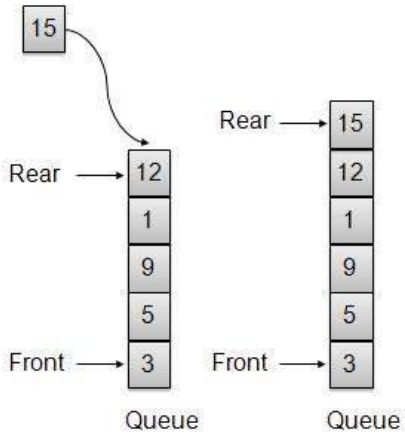
- **insert / enqueue** − add an item to the rear of the queue.
- **remove / dequeue** − remove an item from the front of the queue.

We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

Insert / Enqueue Operation

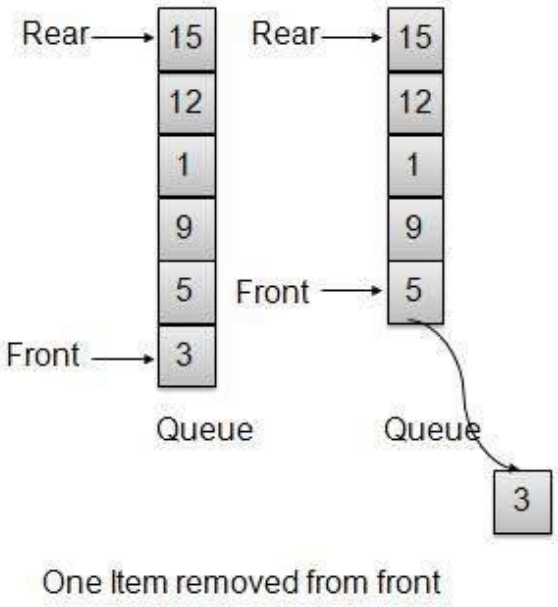
Whenever an element is inserted into queue, queue increments the rear index for later use and stores that element at the rear end of the storage. If rear end reaches to the last index and it is wrapped to the bottom location. Such an arrangement is called wrap around and such queue is circular queue. This method is also termed as enqueue operation.



One item inserted at rear end

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using front index and increments the front index. As a wraparound arrangement, if front index is more than array's max index, it is set to 0.



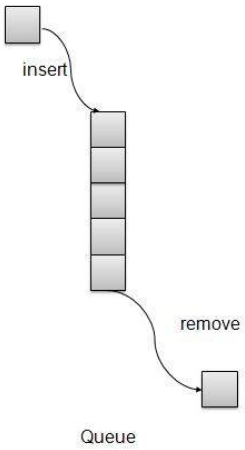
One Item removed from front

Priority Queue:

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So, we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

- **insert / enqueue** − add an item to the rear of the queue.
- **remove / dequeue** − remove an item from the front of the queue.

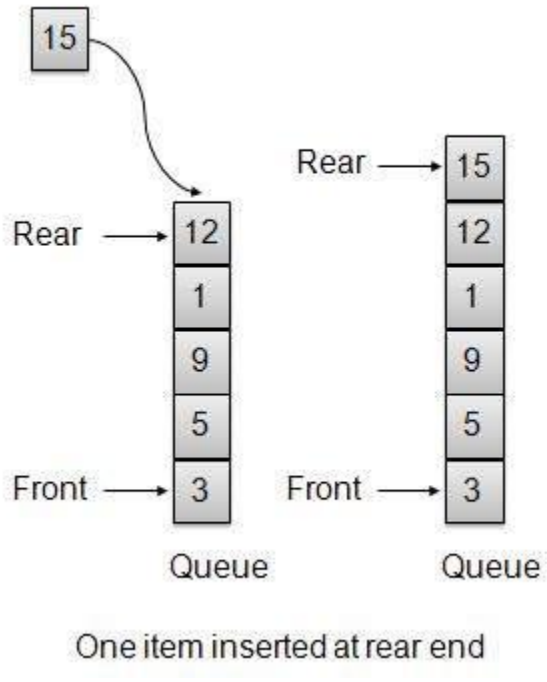
Priority Queue Representation



- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

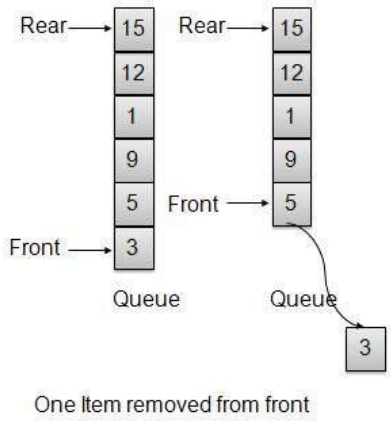
Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.



One Item removed from front

Tree

Tree represents nodes connected by edges. We'll going to discuss binary tree or binary search tree specifically.

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have two children at maximum. A binary tree has benefits of both an ordered array and a linked list as search is as quick as in sorted array and insertion or deletion operation are as fast as in linked list.

Terms

Following are important terms with respect to tree.

- **Path** − Path refers to sequence of nodes along the edges of a tree.
- **Root** − Node at the top of the tree is called root. There is only one root per tree and one path from root node to any node.
- **Parent** − Any node except root node has one edge upward to a node called parent.
- **Child** − Node below a given node connected by its edge downward is called its child node.
- **Leaf** − Node which does not have any child node is called leaf node.
- **Subtree** − Subtree represents descendents of a node.
- **Visiting** − Visiting refers to checking value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2 and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search tree exibits a special behaviour. A node's left child must have value less than its parent's value and node's right child must have value greater than it's parent value.

Binary Search Tree Representation

We're going to implement tree using node object and connecting them through references.

Basic Operations

Following are basic primary operations of a tree which are following.

- **Search** − search an element in a tree.
- **Insert** − insert an element in a tree.
- **Preorder Traversal** − traverse a tree in a preorder manner.
- **Inorder Traversal** − traverse a tree in an inorder manner.
- **Postorder Traversal** − traverse a tree in a postorder manner.

Node

Define a node having some data, references to its left and right child nodes.

```
public class Node {
        public int data;
         public Node leftChild;
         public Node rightChild;
         public Node(){}
        public void display(){
           System.out.print("("+data+ ")");
    }
}
```

Search Operation

Whenever an element is to be search. Start search from root node then if data is less than key value, search element in left subtree otherwise search element in right subtree. Follow the same algorithm for each node.

```
public Node search(int data){
  Node current = root;
  System.out.print("Visiting elements: ");
  while(current.data != data){
    if(current != null)
      System.out.print(current.data + " ");
      //go to left tree
      if(current.data > data){
        current = current.leftChild;
      }//else go to right tree
      else{
        current = current.rightChild;
      }  if(current == null){
        return null;
      }
    }
  return current;
}
```

Insert Operation

Whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left subtree and insert the data. Otherwise search empty location in right subtree and insert the data.

Preorder Traversal

It is a simple three step process.

- visit root node
- traverse left subtree
- traverse right subtree

Inorder Traversal

It is a simple three step process.

- traverse left subtree
- visit root node
- traverse right subtree

Postorder Traversal

It is a simple three step process.

- traverse left subtree
- traverse right subtree
- visit root node

Hash Table

HashTable is a datastructure in which insertion and search operations are very fast irrespective of size of the hashtable. It is nearly a constant or O(1). Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hashtable of size 20, and following items are to be stored. Item are in (key,value) format.

- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr.No.	Key	Hash	Array Index

1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique used create already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

Sr.No.	**Key**	**Hash**	**Array Index**	**After Linear Probing, Array Index**
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are basic primary operations of a hashtable which are following.

- **Search** − search an element in a hashtable.
- **Insert** − insert an element in a hashtable.
- **delete** − delete an element from a hashtable.

DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

Search Operation

Whenever an element is to be searched. Compute the hash code of the key passed and locate the element using that hashcode as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

Delete Operation

Whenever an element is to be deleted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hashtable intact.

Heap

Heap represents a special tree based data structure used to represent priority queue or for heap sort. We'll going to discuss binary heap tree specifically.

Binary heap tree can be classified as a binary tree with two constraints −

- **Completeness** − Binary heap tree is a complete binary tree except the last level which may not have all elements but elements from left to right should be filled in.
- **Heapness** − All parent nodes should be greater or smaller to their children. If parent node is to be greater than its child then it is called Max heap otherwise it is called Min heap. Max heap is used for heap sort and Min heap is used for priority queue. We're considering Min Heap and will use array implementation for the same.
- **Insert** − insert an element in a heap.
- **Get Minimum** − get minimum element from the heap.
- **Remove Minimum** − remove the minimum element from the heap

Insert Operation

- Whenever an element is to be inserted. Insert element at the end of the array. Increase the size of heap by 1.
- Heap up the element while heap property is broken. Compare element with parent's value and swap them if required.

Get Minimum

Get the first element of the array implementing the heap being root.

Remove Minimum

- Whenever an element is to be removed. Get the last element of the array and reduce size of heap by 1.
- Heap down the element while heap property is broken. Compare element with children's value and swap them if required.

Graph:

Graph is a data structure to model the mathematical graphs. It consists of a set of connected pairs called edges of vertices. We can represent a graph using an array of vertices and a two-dimensional array of edges.

Important terms

- **Vertex** – Each node of the graph is represented as a vertex. In example given below, labelled circle represents vertices. So, A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two-dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.
- **Path** – Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.
- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

Add Vertex Operation

```
//add vertex to the array of vertex
public void addVertex(char label){
   lstVertices[vertexCount++] = new Vertex(label);
}
```

Add Edge Operation

```
//add edge to edge array
public void addEdge(int start,int end){
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}
```

Display Edge Operation

```
//display the vertex
public void displayVertex(int vertexIndex){
   System.out.print(lstVertices[vertexIndex].label+" ");
}
```

Traversal Algorithms

Following are important traversal algorithms on a Graph.

- **Depth First Search** − traverses a graph in depth wards motion.
- **Breadth First Search** − traverses a graph in breadth wards motion.

Depth First Search Algorithm

Depth First Search algorithm (DFS) traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** − Visit adjacent unvisited vertex. Mark, it visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until stack is empty.

Breadth First Search Algorithm

Breadth First Search algorithm (BFS) traverses a graph in a breadth wards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

- **Rule 1** − Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until queue is empty.

Search techniques

Search refers to locating a desired element of specified properties in a collection of items. We are going to start our discussion using following commonly used and simple search algorithms.

Sr.No	Technique & Description
1	**Linear Search** Linear search searches all items and its worst execution time is n where n is the number of items.
2	**Binary Search** Binary search requires items to be in sorted order but its worst execution time is constant and is much faster than linear search.
3	**Interpolation Search** Interpolation search requires items to be in sorted order but its worst execution time is O(n) where n is the number of items and it is much faster than linear search.

Sorting techniques

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are numerical or lexicographical order.

Importance of sorting lies in the fact that data searching can be optimized to a very high level if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Some of the examples of sorting in real life scenarios are following.

- **Telephone Directory** − Telephone directory keeps telephone no. of people sorted on their names. So that names can be searched.
- **Dictionary** − Dictionary keeps words in alphabetical order so that searching of any work becomes easy.

Types of Sorting

Following is the list of popular sorting algorithms and their comparison.

Sr.No	Technique & Description
1	**Bubble Sort** Bubble sort is simple to understand and implement algorithm but is very poor in performance.
2	**Selection Sort** Selection sort as name specifies use the technique to select the required item and prepare sorted array accordingly.
3	**Insertion Sort** Insertion sort is a variation of selection sort.
4	**Shell Sort** Shell sort is an efficient version of insertion sort.
5	**Quick Sort** Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
6	**Sorting Objects** Java objects can be sorted easily using java.util.Arrays.sort()

Recursion

Recursion refers to a technique in a programming language where a function calls itself. The function which calls itself is called a recursive method.

Characteristics

A recursive function must posses the following two characteristics

- Base Case(s)
- Set of rules which leads to base case after reducing the cases.

Recursive Factorial

Factorial is one of the classical examples of recursion. Factorial is a non-negative number satisfying following conditions.

1. 0! = 1
2. 1! = 1
3. n! = n * n-1!

Recursive Fibonacci Series:

Fibonacci Series is another classical example of recursion. Fibonacci series a series of integers satisfying following conditions.

1. $F_0 = 0$
2. $F_1 = 1$
3. $F_n = F_{n-1} + F_{n-2}$

Fibonacci is represented by "F". Here Rule 1 and Rule 2 are base cases and Rule 3 are fibonnacci rules.

As an example, F_5 = 0 1 1 2 3

```
public class RecursionDemo {
  public static void main(String[] args){
    RecursionDemo recursionDemo = new RecursionDemo();
    int n = 5;
    System.out.println("Factorial of " + n + ": "
      + recursionDemo.factorial(n));
    System.out.print("Fibbonacci of " + n + ": ");
    for(int i=0;i<n;i++){
      System.out.print(recursionDemo.fibbonacci(i) +" ");
    }
  }
  private int factorial(int n){
    //base case
    if(n == 0){
      return 1;
    }else{
      return n * factorial(n-1);
    }
  }
  private int fibbonacci(int n){
    if(n ==0){
      return 0;
    }
    else if(n==1){
      return 1;
    }
```

```
    else {
      return (fibbonacci(n-1) + fibbonacci(n-2));
    }
  }
}
```

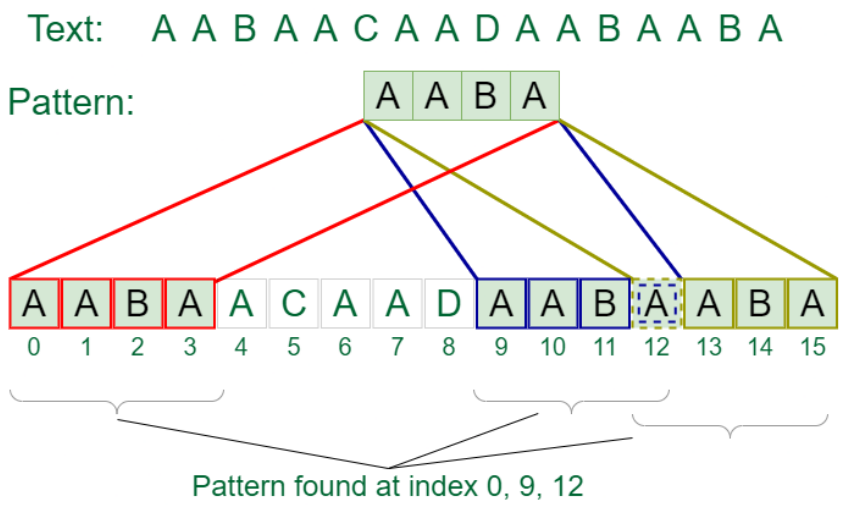
Useful Links on DSA using Java

- DSA using Java @ Wikipedia – DSA using Java, its history and various other terms has been explained in simple language.

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Heap sort

Pattern Searching Algorithms

The Pattern Searching algorithms are sometimes also referred to as String Searching Algorithms and are considered as a part of the String algorithms. These algorithms are useful in the case of searching a string within another string.

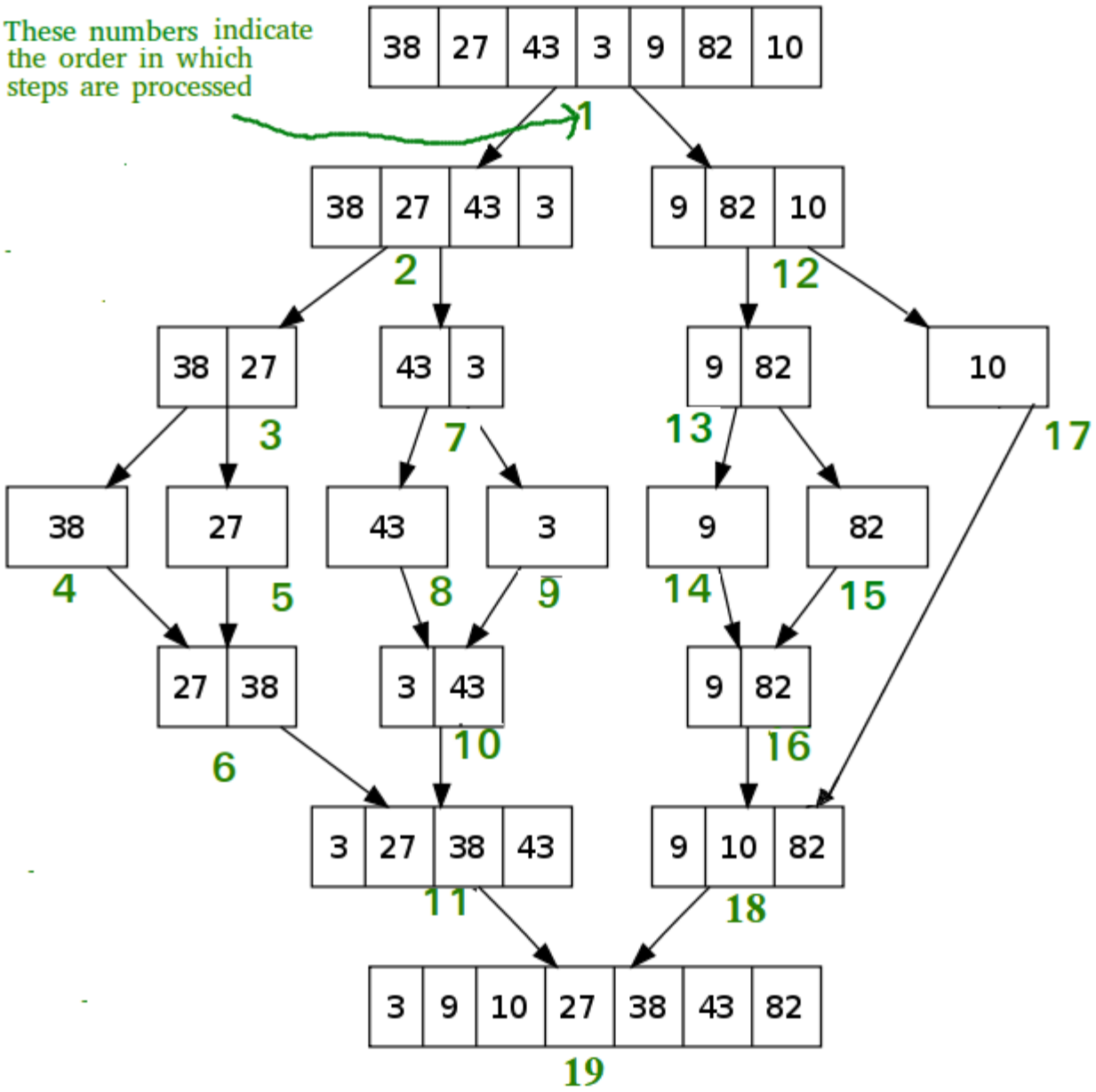


Pattern Searching Algorithms

1. Rabin Karp Algorithm
2. KMP Algorithm
3. Z Algorithm

Divide and Conquer

As the name itself suggests it's first divided into smaller sub-problems then these subproblems are solved and later on these problems are combined to get the final solution. There are so many important algorithms that work on the ***Divide and Conquer*** *strategy.*



Divide and Conquer

Some examples of Divide and Conquer algorithms are as follows:

- Quick Sort
- Merge Sort
- Matrix Multiplication

Backtracking

Backtracking *is a variation of recursion. In backtracking, we solve the sub-problem with some changes one at a time and remove that change after calculating the solution of the problem to this sub-problem. It takes every possible combination of problems in order to solve them.*

There are some standard questions on backtracking as mentioned below:

- N-Queens
- Rat in a maze
- Sudoku
- M coloring problemGreedy Algorithm

*A **greedy algorithm** is a method of solving problems with the most optimal option available. It's used in such situations where optimization is required i.e. where the maximization or the minimization is required.*

Some of the most common problems with greedy algorithms are as follows -

- Job Sequencing Problem
- Prim's algorithms for MST
- Fractional Knapsack problem

Dynamic Programming

***Dynamic programming** is one of the most important algorithms that is asked in coding interviews. Dynamic programming works on recursion. It's an optimization of recursion. Dynamic Programming can be applied to all such problems, where we have to solve a problem using its sub-problems. And the final solution is derived from the solutions of smaller sub-problems. It basically stores solutions of sub-problems and simply uses the stored result wherever required, in spite of calculating the same thing again and again.*

Some of the very important questions based on Dynamic Programming are as follows:

- 0/1 Knapsack Problem
- Longest Palindromic subsequence
- Longest common subsequence

Tree Traversals Algorithms

Tree traversal refers to the process of visiting all nodes in a tree data structure, typically in a specific order. In other words, it's the method of accessing and processing each node in a tree in a systematic way.

here are different ways to traverse a tree, but the most common ones are:

1. In-Order Traversal: In this traversal, the nodes are visited in the order of left subtree, root node, and right subtree. This type of traversal is commonly used in binary search trees.
2. Pre-Order Traversal: In this traversal, the nodes are visited in the order of root node, left subtree, and right subtree.
3. Post-Order Traversal: In this traversal, the nodes are visited in the order of left subtree, right subtree, and root node.
4. Level-Order Traversal: It involves visiting each level of the tree from left to right. In this traversal, we start from the root node, visit all the nodes on the first level, then move to the second level and visit all the nodes on that level, and so on until we reach the last level.

Graphs Algorithms

1. Breadth First Search (BFS): A graph traversal algorithm that explores all the vertices of a graph that are reachable from a given source vertex.
2. Depth First Search (DFS): A graph traversal algorithm that explores all the vertices of a graph that are reachable from a given source vertex, by going as far as possible along each branch before backtracking.
3. Dijkstra Algorithm: A shortest path algorithm that finds the shortest path between a given source vertex and all other vertices in a weighted graph.
4. Floyd Warshall Algorithm: An all-pairs shortest path algorithm that finds the shortest path between all pairs of vertices in a weighted graph.
5. Bellman-Ford Algorithm: A shortest path algorithm that can handle negative weight edges and detect negative weight cycles.
6. Prim's algorithm: A minimum spanning tree algorithm that finds the minimum spanning tree of a weighted, undirected graph.

7. Kruskal's algorithm: Another minimum spanning tree algorithm that finds the minimum spanning tree of a weighted, undirected graph.

Sliding Window

Window Sliding Technique is a computational technique that aims to reduce the use of nested loops and replace it with a single loop, thereby reducing the time complexity.

There are several types of sliding window techniques, including:

Fixed-size sliding window: In this technique, the size of the window is fixed, and we move the window by one position at a time. This is often used to perform operations like finding the maximum or minimum value in a subarray of fixed size.

Variable-size sliding window: In this technique, the size of the window changes dynamically, depending on the problem constraints. This technique is often used to find the minimum or maximum subarray whose sum is greater than or equal to a certain value.

Easy Problems

- Second Largest Element
- Third Largest Element
- Three Great Candidates
- Max Consecutive Ones
- Move All Zeroes To End
- Reverse Array in Groups
- Rotate Array
- Wave Array
- Plus One
- Stock Buy and Sell – One Transaction
- Repetitive Addition Of Digits
- Remove All Occurrences of Element
- Remove Duplicates from Sorted Array
- Alternate Positive Negative
- Insert Duplicate Element
- Array Leaders
- Missing and Repeating in Array
- Missing Ranges of Numbers

Medium Problems

- Next Permutation
- Majority Element
- Majority Element II
- Stock Buy and Sell – Multiple Transactions
- Minimize the Heights II
- Maximum Subarray Sum
- Maximum Product Subarray
- Product of Array Except Self
- Subarrays with Product Less Than K
- Split Into Three Equal Sum Segments
- Maximum Consecutive 1s After Flipping 0s
- Last Moment Before Ants Fall Out of Plank
- Find 0 with Farthest 1s in a Binary
- Intersection of Interval Lists
- Rearrange Array Elements by Sign
- Meeting Scheduler for Two Persons
- Longest Mountain Subarray
- Transform and Sort Array
- Minimum Swaps To Group All Ones
- Minimum Moves To Equalize Array
- Minimum Indices To Equal Even-Odd Sums

Hard Problems

- Trapping Rain Water
- Maximum Circular Subarray Sum
- Smallest Missing Positive Number
- Jump Game
- Closest Subsequence Sum
- Smallest Non-Representable Sum in Array
- Smallest Range Having Elements From K Lists
- Candy Distribution
- Count Subarrays with K Distinct Elements
- Next Smallest Palindrome
- Maximum Sum Among All Rotations