**Data Structure**

**Data Structure**

* + A data structure is a storage that is used to to store and organize data.
  + It is a way of arranging data on a computer so that iit can be accessed and updated efficiently.
  + A data structure is not only used for organizing the data it is also used for processing retrieving and storing data.

**Classificaton of Data Structure**

There are two type of data structure.

1. Linear data structure

1.1 Static Data Structure

1.1.1 Array

1.2 Dynamic Data Structure

1.2.1 Queue

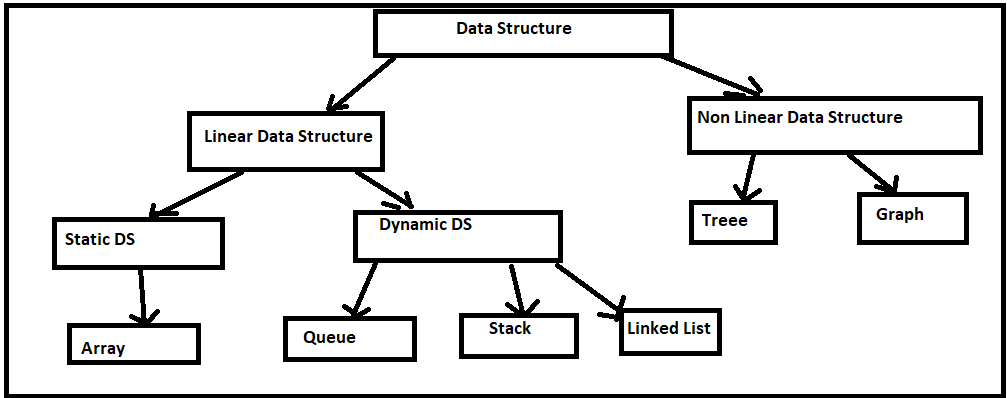
1.2.2 Stack

1.2.3 Linked List

2. Non Linear Data Structue

2.1 Tree

2.2 Graph



**Linear Data Structure**

* + Data Structure in which data elements are arranged sequentially or linearly, where each elements is attached to its previous and next adjucent elements is called a linear data structure.

**Example**=> Array, Stack, Queue, Linked List etc.

**Static Data Structure**

* + Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.

**Example**=> Array

**Dynamic Data Structure**

* + In dynamic data structure the size is not defined. It can be randomly updated during the runtime which may be considered efficient concerning the memory(space) complexity of the code.

**Example**=> Queue, Stack, Linked List

**Non Linear Data Structure**

* + Data Structure where data elements are not placed sequentially are called Non-Linear Data Structure. We can't traverse all the elements in a single run only.

**Example**=> Tree and Graph

**LINKED LIST**

A Linked List is a**linear data structure** which looks like a chain of nodes, where each node is a different element. Unlike Arrays, Linked List elements are not stored at a contiguous location.

**Why linked list data structure needed?**

Here are a few advantages of a linked list that is listed below, it will help you understand why it is necessary to know.

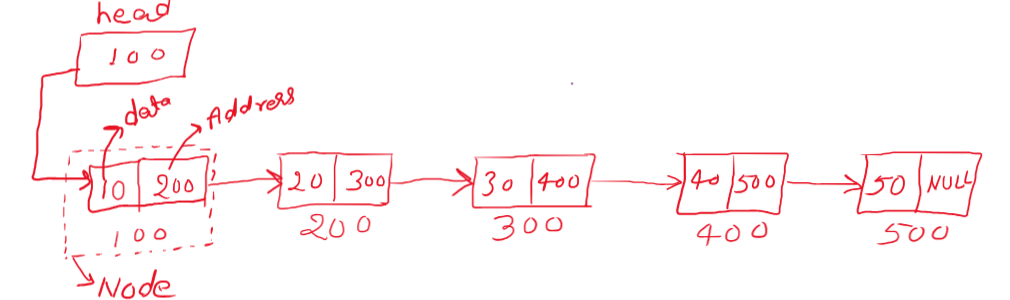
* **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
* **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
* **Efficient Memory Utilization:**As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
* **Implementation:**Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

**There are three type of linked list**

* + Singly Linked List
  + Doubly Linked List
  + Circular Linked List

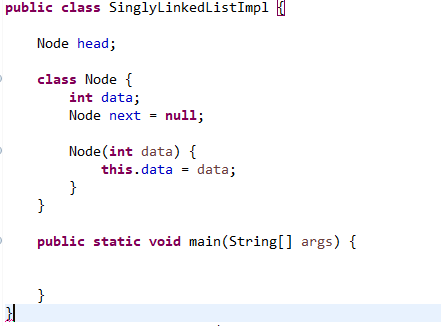
**Singly Linked List**

A **singly linked list** is a linear data structure in which the elements are not stored in contiguous memory locations and each element is connected only to its next element using a pointer.



**How to create list**

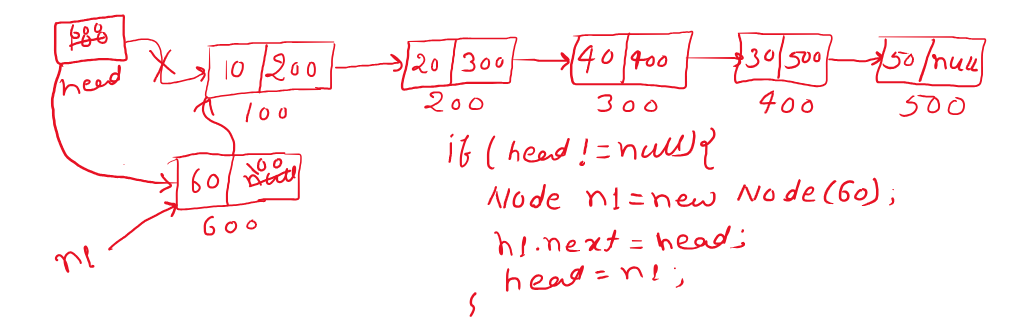
**Node Structure**

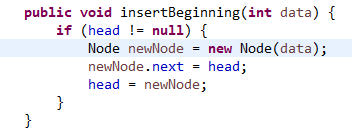
****

**Insertion of the Singly Linked List**

* At the front of the linked list
* After a given node.
* At the end of the linked list

## [How to Insert a Node at the Front/Beginning of Linked List](https://www.geeksforgeeks.org/insert-a-node-at-front-beginning-of-a-linked-list/)

* Make the first node of Linked List linked to the new node
* Remove the head from the original first node of Linked List
* Make the new node as the Head of the Linked List.

****

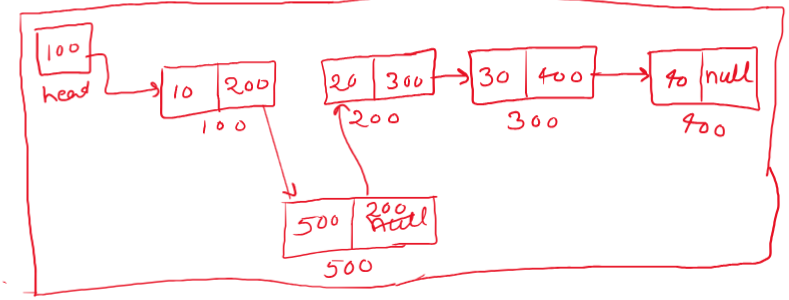
**Complexity Analysis:**

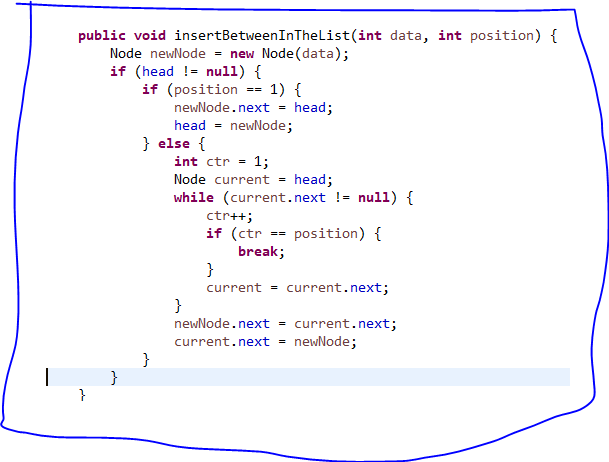
* **Time Complexity:** O(1), We have a pointer to the head and we can directly attach a node and change the pointer. So the Time complexity of inserting a node at the head position is O(1) as it does a constant amount of work.
* **Auxiliary Space:**O(1)

## How to Insert a Node Between of the List

In this list need to add new node between of the list

* + create the current node from head.
  + Do the while loop based on current node. If while loop iteration is equal of position the break the while loop
  + Assign current into current of next node.
  + Create the new node
  + Current.next address assign into the new node of next.
  + Assign new node address into the current.next





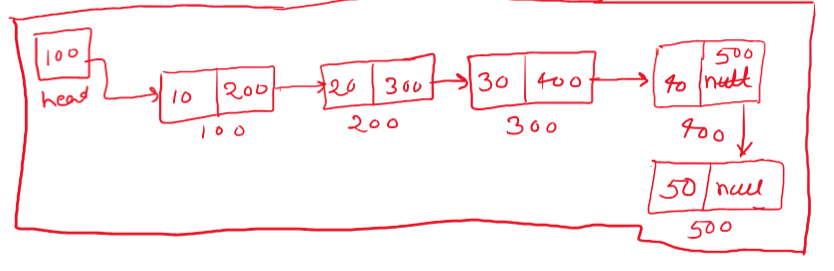
[How to Insert a Node at the End of Linked List](https://www.geeksforgeeks.org/insert-node-at-the-end-of-a-linked-list/)

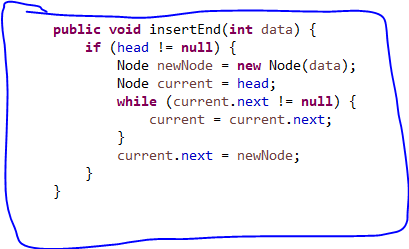
**Approach:**

To insert a node at the end of a Linked List, we need to:

* + Go to the last node of the Linked List
  + Change the next pointer of last node from NULL to the new node
  + Make the next pointer of new node as NULL to show the end of Linked List

After insertion of new node in the last of the list





**Full Code of Linked List Operation**

**public** **class** **SinglyLinkedListImpl** {

Node head;

**class** Node {

**int** data;

Node next = **null**;

Node(**int** data) {

**this**.data = data;

}

}

**public void createList**(**int** data) {

Node newNode = **new** Node(data);

**if** (head == **null**) {

head = newNode;

} **else** {

Node current = head;

**while** (current.next != **null**) {

current = current.next;

}

current.next = newNode;

}

}

**public** **void** **displayList**() {

**if** (head != **null**) {

Node current = head;

**while** (current != **null**) {

System.***out***.print(current.data + " ");

current = current.next;

}

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

}

}

**public** **void** **insertBeginning**(**int** data) {

**if** (head != **null**) {

Node newNode = **new** Node(data);

newNode.next = head;

head = newNode;

}

}

**public** **void** **insertEnd**(**int** data) {

**if** (head != **null**) {

Node newNode = **new** Node(data);

Node current = head;

**while** (current.next != **null**) {

current = current.next;

}

current.next = newNode;

}

}

**public** **void** **insertBetweenInTheList**(**int** data, **int** position) {

Node newNode = **new** Node(data);

**if** (head != **null**) {

**if** (position == 1) {

newNode.next = head;

head = newNode;

} **else** {

**int** ctr = 1;

Node current = head;

**while** (current.next != **null**) {

ctr++;

**if** (ctr == position) {

**break**;

}

current = current.next;

}

newNode.next = current.next;

current.next = newNode;

}

}

}

**public** **void** **reverseList**() {

Node prev = **null**;

Node current = head;

Node next = **null**;

**while** (current != **null**) {

next = current.next;

current.next = prev;

prev = current;

current = next;

}

**if**(prev != **null**) {

head = prev;

}

displayList();

}

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

SinglyLinkedListImpl list = **new** SinglyLinkedListImpl();

**for** (**int** i = 1; i <= 10; i++) {

list.createList(i);

}

list.displayList();

System.***out***.println("insert beginning of the list");

list.insertBeginning(1 );

list.displayList();

System.***out***.println("insert end of the list");

list.insertEnd(12);

list.displayList();

System.***out***.println("insert between of the list");

list.insertBetweenInTheList(13, 1);

list.displayList();

System.***out***.println("reverse of the list");

list.reverseList();

}

}

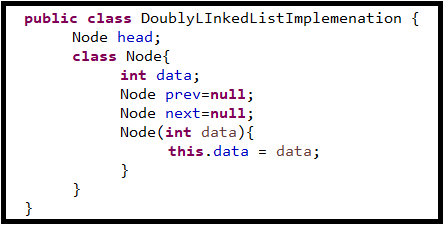
**DOUBLY LINKED LIST**

**What is Doubly Linked List?**

A **doubly linked list** (DLL) is a special type of linked list in which each node contains a pointer to the previous node as well as the next node of the linked list.



**Doubly Linked List Node Structure**



## ****Advantages of Doubly Linked List over the singly linked list:****

* A DLL can be traversed in both forward and backward directions.
* The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
* We can quickly insert a new node before a given node.
* In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

## ****Disadvantages of Doubly Linked List over the singly linked list:****

* Every node of DLL Requires extra space for a previous pointer.
* All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with the next pointers. For example in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

## Applications of Doubly Linked List:

* It is used by web browsers for backward and forward navigation of web pages
* LRU ( Least Recently Used ) / MRU ( Most Recently Used ) Cache are constructed using Doubly Linked Lists.
* Used by various applications to maintain undo and redo functionalities.
* In Operating Systems, a doubly linked list is maintained by thread scheduler to keep track of processes that are being executed at that time.

**Creation of Doubly Linked List**

* + Allocate the new node.
  + If head is null then assign new node address to head.
  + If head is not null then take the current node and assign the head address.
  + Iterate current node till last node until current of next is null.
  + Assign the address of new node in side current of next node.
  + Assign current address inside the new node of prev.

**public** **void** createDoublyList(**int** data) {

Node newNode = **new** Node(data);

**if**(head == **null**) {

head = newNode;

}**else** {

Node current = head;

**while**(current.next != **null**) {

current = current.next;

}

current.next=newNode;

newNode.prev = current;

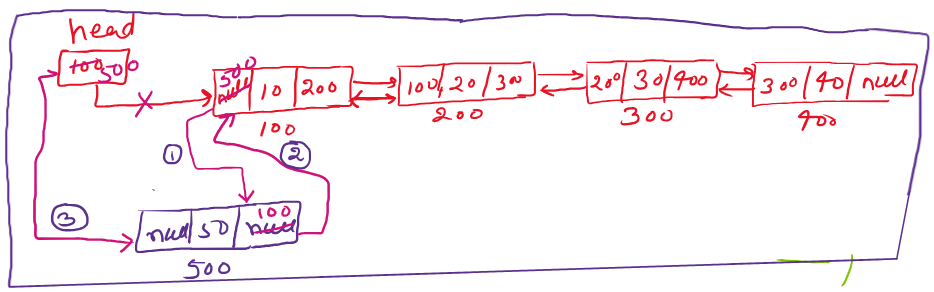
}

}

**Insertion of the Doubly Linked List**

* + Insert beginning of the Doubly Linked List.
  + Insert last of the Doubly Linked List.
  + Insert between of the Doubly Linked List.

**Insert beginning of the Doubly Linked List**

* Create the new node and assign the data.
* Check if head is null or not.
* If head is null the assign new node address into the head.
* If head is not null then assign head address into the new node of the next.
* Assign new node address into the head of the prev.
* Assign new node addres into the head.

**public** **void** insertBeginningOfTheDoublyList(**int** data) {

Node newNode = **new** Node(data);

**if**(head == **null**) {

head = newNode;

}**else** {

newNode.next = head;

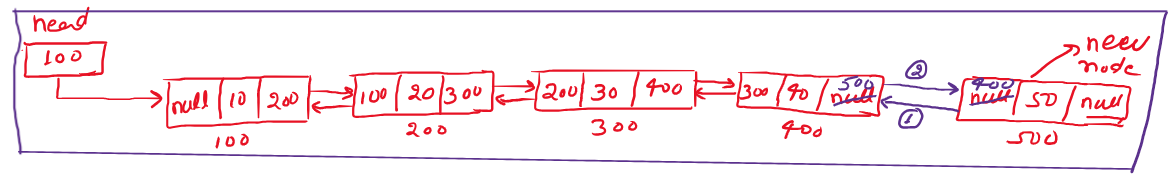
head.prev = newNode;

head = newNode;

}

}

**Insert last of the Doubly Linked List**

* Create the new node and assign the data.
* Check if head is null or not.
* If head is null the assign new node address into the head.
  + If head is not null the take the current node and assign the head address.
  + Iterate the current node until current node of the next is null.
  + Assign the new node address into the current node of the next,
  + Assign the current node address into the new node of the prev.

**public** **void** insertLastOfTheDoublyList(**int** data) {

Node newNode = **new** Node(data);

**if**(head == **null**) {

head = newNode;

}**else** {

Node current = head;

**while**(current.next != **null**) {

current = current.next;

}

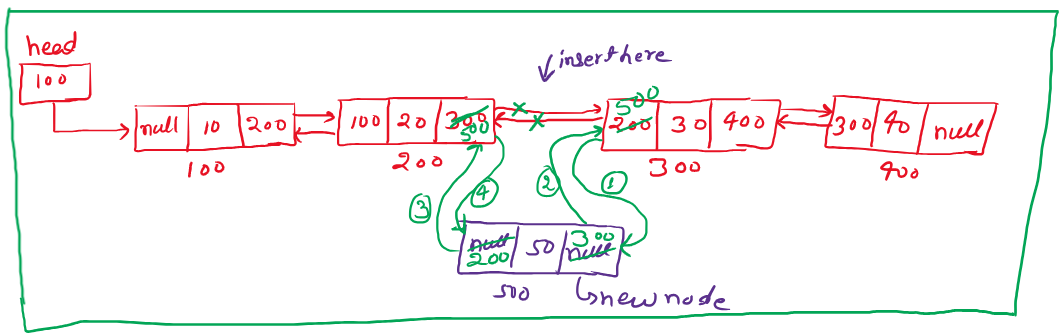
current.next = newNode;

newNode.prev = current;

}

}

**Insert between of the Doubly Linked List**

* Create the new node and assign the data.
  + Take the position where you want to insert the new node.
  + Take the counter variable and assign the default value one.
  + Take the current node and assign the head address.
  + Iterate the current node until last node of next is null.
  + Then check if counter value is equal to the position.
  + If counter value is not equal to the position then increment the counter and assign current of next address into the current.
  + If counter value is equal of position then break the loop.
  + Then assign the current node of next address into the new node of next.
  + Assign the new node address into the current node of next of prev.
  + Assign current node address into the new node of prev.
  + Assign new node address into the current node of the next.

**public** **void** insertBetweenOfTheList(**int** data, **int** pos) {

Node newNode = **new** Node(data);

**int** ctr=1;

Node current = head;

**while**(current.next != **null**) {

**if**(ctr == pos) {

**break**;

}

ctr++;

current = current.next;

}

newNode.next = current.next;

current.next.prev = newNode;

newNode.prev = current;

current.next = newNode;

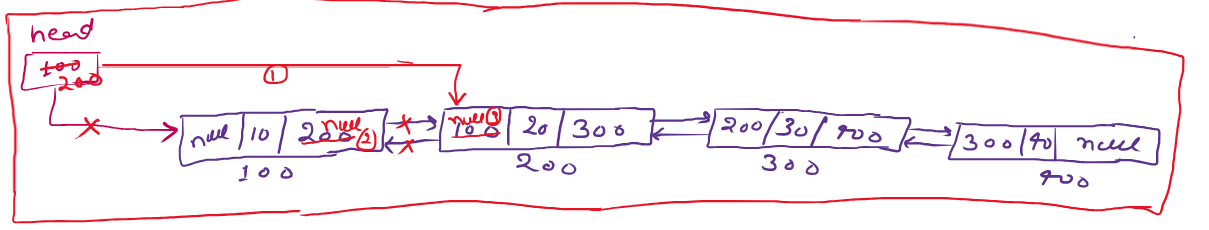
}

**DELETION OF DOUBLY LINKED LIST**

* + Delete node from beginning of the doubly linked list.
  + Delete node from last of the doubly linked list.
  + Delete node from between of the doubly linked list.

**Delete node from beginning of the doubly linked list**

* + Check head is null of or not.
  + If head is null then print list is empty.
  + If head is not null the assign next node address of head in the head.
  + Make the pev node of head of the next node is null.
  + Make the prev node of head is null.



**public** **void** deleteBeginningOfThedoublyList() {

**if**(head != **null**) {

head = head.next;

head.prev.next=**null**;

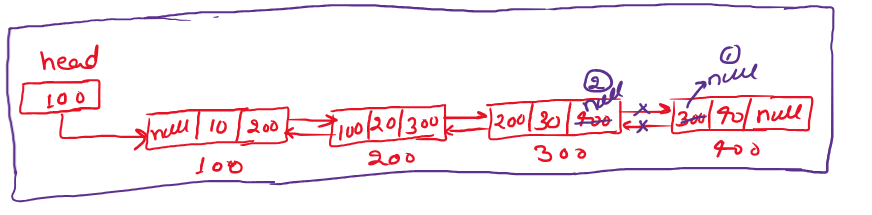
head.prev = **null**;

}

}

**Delete node from last of the doubly linked list**

* + Check head is null or not
  + If head is null the print list is empty.
  + If head is not null then take the current node from head.
  + Traverse the current node until next of next node of the current is null.
  + Make the prev node of the next node of current is null.
  + Make the next node of the current is null.



**public** **void** deleteEndOfTheDoublyList() {

**if**(head != **null**) {

Node current = head;

**while**(current.next.next != **null**) {

current = current.next;

}

current.next.prev = **null**;

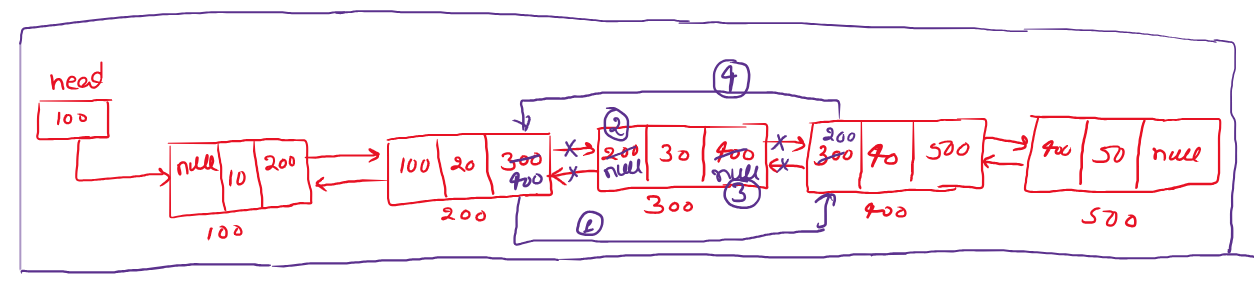
current.next = **null**;

}

}

**Delete node from between of the doubly linked list**

* + Check if head is null or not.
  + If head is null then list is empty.
  + If head is not null then take current node form head and take the counter and position (which position you want to delete the node).
  + Traverse current node until current node is null.
  + If counter is equal to position then break the while loop otherwise assign the address of the next node of current in the current.
  + Assign the address of next node of the next node of the current into the next node of the current.
  + Make the prev node of the prev of the next node of the current is null.
  + Make the next node of the prev of the next node of the current is null.
  + Assign the current node address into the prev node of the next node of the current.



**public** **void** deleteBetweenOfTheDoublyList(**int** position) {

**if**(head != **null**) {

Node current = head;

**int** counter =1;

**while**(current != **null**) {

counter++;

**if**(counter == position) {

**break**;

}

current = current.next;

}

current.next = current.next.next;

current.next.prev.prev = **null**;

current.next.prev.next = **null**;

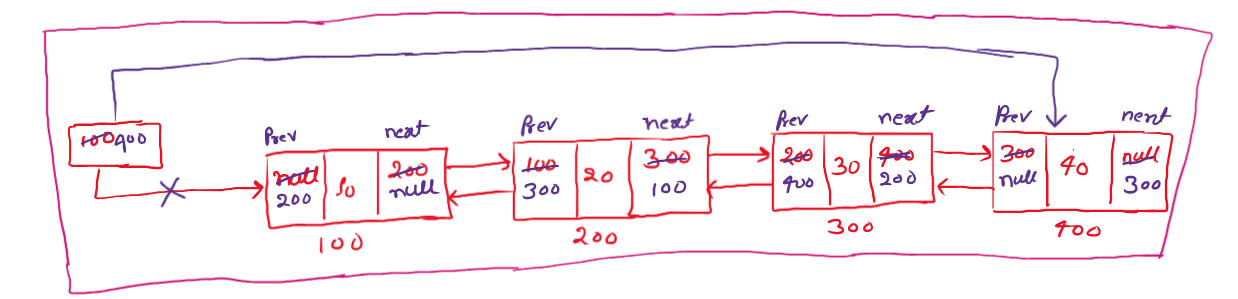
current.next.prev = current;

}

}

**Reverse of the Doubly Linked List**

* + Check of head is null or not.
  + If head is null the print list is empty.
  + If head is not null then take the current node from head.
  + Take temp node and assign the value is null.
  + Traverse the current node until current node is null.
  + Assign the prev node address of the current into the temp
  + Assign next node address of the current into the prev node of the current.
  + Assign the temp node address into the next node of the current.
  + Assign the prev node address of the current into the current node.



**public** **void** reverseDoublyLinkedlist() {

**if**(head != **null**) {

Node current = head;

Node temp = **null**;

**while**(current != **null**) {

temp = current.prev;

current.prev = current.next;

current.next = temp;

current = current.prev;

}

**if**(temp != **null**) {

head = temp.prev;

}

}

}

**STACK**

* + Stack is a linear data structure used for storing data.
  + A stack is a an order list in which insertion and deletion are done at one end

called top.

* + The last element inserted is the first none to be deleted.
  + Stack is called LastInFirstOut(LIFO) or FirstInLastOut(FILO).
  + when the element is inserted in the stack then it called **PUSH** and when

element is removed from the stack is called **POP**.

* + If we are trying to pop from empty stack is called **underflow** and if we are

trying to push an element in a full stack is called **overflow**.

* + If stack is empty then **TOP** is -1.

**Operation on the stack**

* + void push(int data) : insert data into the stack.
  + int pop(): Removes and returns the last inserted element from the stack.
  + int TOP(): Returns the last inserted element without removing it.
  + int size(): Return the number of the elements stored in the stack.
  + boolean isEmptyStack(): Indicates whether any elements are stored in the

stack or not.

* + boolean isFullStack(): indicated whether the stack is full or not.

**Application**

Following are some of the application in which stacks play an important role

**Direct Application**

* + Balance of Symbols
  + Infix to postfix conversion
  + Evaluation of postfix expression.
  + Implementing function call (including recursion)
  + Finding of spans(Finding spans in stock markets)
  + Page visited history in a web browsers(back button)
  + Undo sequence in a text editor.
  + Matching tags in HTML and XML.

**Indirect Application**

* + Auxiliary data structure for oter alogrithms. (Example : Tree traversal

algorithm)

* + Component of other data structure. (Example : simulating ques)

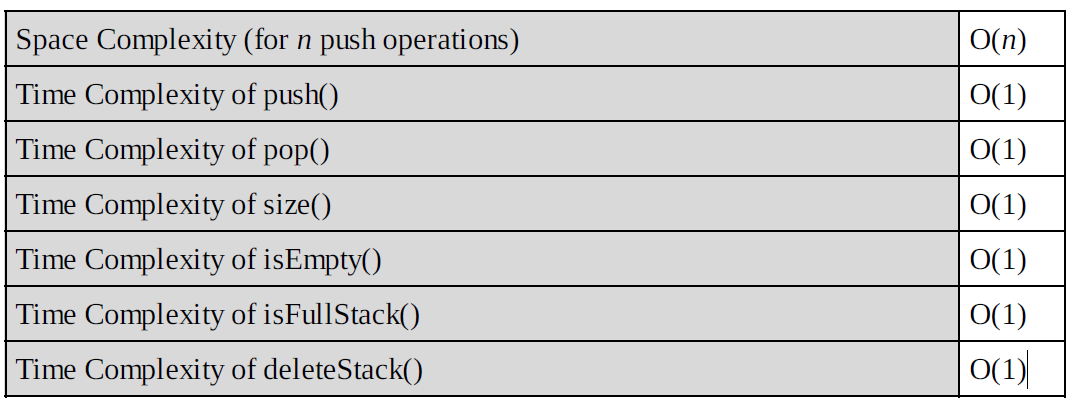
**Stack Implementation**

There are many ways of implementing stack ADT. below are the commonly used methods.

* + Simple array based implementation.(stack with array)
  + Dynamic array based implementation. (stack with dynamic array)
  + Linked Lists implementation. (Stack with linked list)

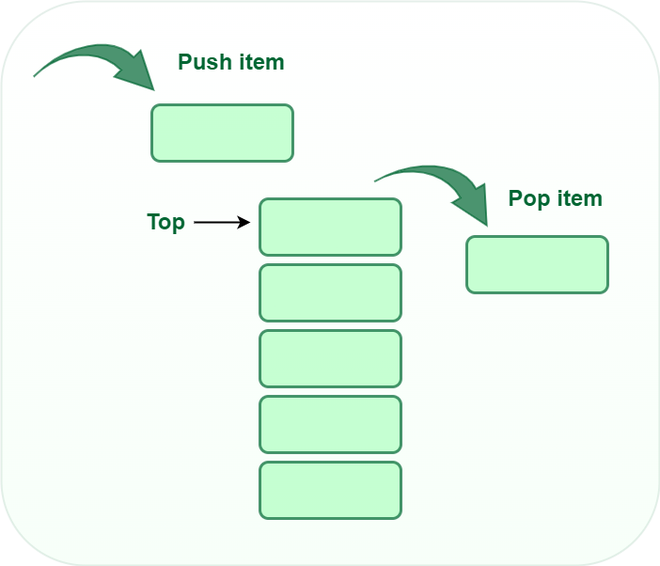
**Performance & Limitations**

**Performance:**

Let *n* be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

**Limitations:**

* The maximum size of the stack must first be defined and it cannot be changed.
* Trying to push a new element into a full stack causes an implementation-specific exception.

**STACK WITH ARRAY**

**Push:**

Adds an item to the stack. If the stack is full, then it is said to be an**Overflow condition.**

**Algorithm for push:**

begin

if stack is full

return

endif

else

increment top

stack[top] assign value

end else

end procedure

**=======================================================**

**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.**

**Algorithm for pop:**

begin

if stack is empty

return

endif

else

store value of stack[top]

decrement top

return value

end else

end procedure

**=======================================================**

**Top:**

Returns the top element of the stack.

**Algorithm for Top:**

begin

return stack[top]

end procedure

**=======================================================**

**isEmpty:**

Returns true if the stack is empty, else false.

**Algorithm for isEmpty**:

begin

if top < 1

return true

else

return false

end procedure