A **singly linked list** is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail).

Each element in a linked list is called a **node**. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.

The first node is called the **head**; it points to the first node of the list

The last node, also sometimes called the **tail**, points to NULL

operations on single linked list

**Operations on singly linked list**

**1. Insertion**

Insertion in a singly linked list can be performed in the following ways,

* **Insertion at the start** Insertion of a new node at the start of a singly linked list is carried out in the following manner.
  + Make the new node point to HEAD.
  + Make the HEAD point to the new node.

**Inserting a new node at the start is an O(1) operation.**

void insertAtStart(Node newNode, Node head){

newNode.data = 10;

newNode.next = head;

head.next = newNode;

}

* **Insertion after some Node** Insertion of a new node after some node in a singly linked list is carried out in the following manner,
  + Reach the desired node after which the new node is to be inserted.
  + Make the new node point to the next element of the current node.
  + Make the current node point to the new node. Inserting a new node after some node is an O(N) operation.

void insertAfterTargetNode(Node newNode, Node head, int target){

newNode.data = 10;

Node temp = head;

while(temp.data != target){

temp = temp.next;

}

newNode.next = temp.next;

temp.next = newNode;

}

* **Insertion at the end** Insertion of a new node at the end of a singly linked list isperformed in te followin way,
  + Taverse the list from start and reach the last node.
  + Make the last node point to the new node.
  + Make the new node point to null, marking the end of the list. Inserting a new node at the end is an O(N) operation.

void insertAtEnd(Node newNode, Node head){

newNode.data = 10;

Node temp = head;

while(temp.next != null){

temp = temp.next;

}

temp.next = newNode;

newNode.next = null;

}

**2. Deletion**

Deletion in a singly linked list can be performed in the following ways,

* **Deletion at the start**  
  The first node of the singly linked list can be deleted as follows,
  + Make the HEAD point to its next element.

Deleting the first node of a singly linked list is an O(1) operation.

void deleteAtFirst(Node head){

head = head.next;

}

* **Deletion at the middle**

The deletion after a specific node can be formed in the following way,

* Reach the desired node after which the node is to be deleted.
* Make the current node point to the next of next element.

Deleting a node after a specific node is an O(N) operation.

void deleteAfterTarget(Node head, int target){

Node temp = head;

while(temp.data != target){

temp = temp.next;

}

temp.next= temp.next.next;

}

* **Deletion at last**

The deletion of the last node is performed in the following manner,

* Reach the second last node of th singly linke list.
* Make the second last node point null.

Deleting the last node is an O(N) operation.

void deleteLast(Node head){

Node temp = head;

while(temp.next.next != null){

temp = temp.next;

}

temp.next = null;

}

**3. Display**

To display the entire singly linked list, we need to traverse it from first to last.

In contrast to arrays, linked list nodes cannot be accessed randomly. Hence to reach the n-th element, we are bound to traverse through all (n-1) elements.

Since the entire linked list is traversed, the operation costs us O(N) time complexity. The following JAVA snippet shows how the entire singly linked list can be displayed.

void display(Node head){

Node temp = head;

while(temp != null){

System.out.println(temp.data);

temp = temp.next;

}

}

**4. Search**

To search an element in the singly linked list, we need to traverse the linked list right from the start.

At each node, we perform a lookup to determine if the target has been found, if yes, then we return the target node else we move to the next element.

In the worst case, we could end up visiting all the nodes in the list and hence searching an element in the singly linked list cost us O(N) operational time.

Node search(Node head, int target){

Node temp = head;

while(temp != null && temp.data != target){

temp = temp.next;

}

return temp;

}