SINGLE LINKED LIST

What is Linked List?
When we want to work with unknown number of data values, we use a linked list data structure to organize that data. Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".

What is Single Linked List?
Simply a list is a sequence of data, and linked list is a sequence of data linked with each other.

The formal definition of a single linked list is as follows...

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data and next. The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

The graphical representation of a node in a single linked list is as follows...

☀ In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head").
☀ Always next part (reference part) of the last node must be NULL.

Example

Operations
In a single linked list we perform the following operations...

Insertion
Deletion
Display
Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1: Include all the header files which are used in the program.
Step 2: Declare all the user defined functions.
Step 3: Define a Node structure with two members data and next
Step 4: Define a Node pointer 'head' and set it to NULL.

Step 4: Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

Inserting At Beginning of the list
Inserting At End of the list
Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1: Create a newNode with given value.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty then, set newNode→next = NULL and head = newNode.
Step 4: If it is Not Empty then, set newNode→next = head and head = newNode.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1: Create a newNode with given value and newNode → next as NULL.
Step 2: Check whether list is Empty (head == NULL).
Step 3: If it is Empty then, set head = newNode.
Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.
Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
Step 6: Set temp → next = newNode.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

Step 1: Create a newNode with given value.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty then, set newNode → next = NULL and head = newNode.
Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.
Step 5: Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
Step 6: Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.
Step 7: Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

Deleting from Beginning of the list
Deleting from End of the list

Deleting a Specific Node
Deleting from Beginning of the list
We can use the following steps to delete a node from beginning of the single linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4: Check whether list is having only one node (temp → next == NULL)
Step 5: If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)
Step 6: If it is FALSE then set head = temp → next, and delete temp.
Deleting from End of the list
We can use the following steps to delete a node from end of the single linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
Step 4: Check whether list has only one Node (temp1 → next == NULL)
Step 5: If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)
Step 6: If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)
Step 7: Finally, Set temp2 → next = NULL and delete temp1.
Deleting a Specific Node from the list
We can use the following steps to delete a specific node from the single linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
Step 4: Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
Step 5: If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.
Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
Step 7: If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).
Step 8: If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).
Step 9: If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.
Step 10: If temp1 is not first node then check whether it is last node in the list (temp1 → next

== NULL).
Step 11: If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).
Step 12: If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).
Displaying a Single Linked List
We can use the following steps to display the elements of a single linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!!' and terminate the function.
Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4: Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
Step 5: Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).
Inserting a Node at the beginning of the List

In this case newNode is inserted at the starting of the List. We have to update Next in newNode to point to the previous firstNode and also update Head to point to newNode.

Pseudocode:
firstNode = Head->Next

newNode->Next = firstNode

Head->Next = newNode
But above pseudocode can be modified to reduce the space complexity by removing the temporary variable usage as shown below
newNode->Next = Head->Next

Head->Next = newNode
Complexity:
Time Complexity: O(1)
Space Complexity: None
}

Inserting a Node at the End of the List

In order to add the node at the end of the list we have to first traverse to the end of the List. Then we have to update the Next variable in lastNode pointing to newNode.

Pseudocode:
cur = head

forever:

 if cur->Next == NULL
 break

cur->Next = newNode
Complexity:
To add a Node at the end of a list whose length is 'n'

Time Complexity: O(n)
Space Complexity: O(1)
Inserting a Node at position 'p' in the List

To add at the position 'p' we have to traverse the list until we reach the position 'p'. For this case have to maintain two pointers namely prevNode and curNode. Since Singly Linked Lists are uni-directional we have to maintain the information about previous Node in prevNode. Once we reach the position 'p' we have to modify prevNode Next pointing to newNode while newNode points to curNode.

Pseudocode:

```
curNode = head
curPos = 1

forever:

 if curPos == P || curNode == NULL
 break

 prevNode = curNode
 curNode = curNode->Next
 curPos++

if curNode != NULL:

 prevNode->Next = newNode
 newNode->Next = curNode
```

Complexity:
Time Complexity: O(n) in worst case.
Space Complexity: O(3)
Traversing a Singly Linked List
Traversing a Singly Linked List is the basic operation we should know before we do other operations like Inserting a Node or Deletion of a Node from the Singly Linked List. Let us see how to traverse Singly Linked List in Figure 1.Let us assume Head points to the first Node in the List.

Pseudocode:

```
cur = head

forever:
```

```
if cur == NULL
break

cur = cur->Next
```

Complexity:
To traverse a complete list of size 'n' it takes
Time complexity: O(n).
Space Complexity: O(1) for storing one temporary variable.
d List.
Next Article: Reversing a Singly Linked List

Deletion of a Node from a Singly Linked List
Similar to insertion we have three cases for deleting a Node from a Singly Linked List.

Deleting First Node in Singly Linked List

To complete deletion of firstNode in the list we have to change Head pointing to Next of firstNode.

Pseudocode:

```
firstNode = Head

Head = firstNode->Next

free firstNode
```

Complexity:
Time Complexity: O(1)
Space Complexity: O(1)
Deleting Last Node in the Singly Linked List

Traverse to Last Node in the List using two pointers namely prevNode and curNode. Once curNode reaches the last Node in the list point Next in prevNode to NULL and free the curNode.

Pseudocode:

```
curNode = head

forever:

 if curNode->Next == NULL
 break

 prevNode = curNode
 curNode = curNode->Next

prevNode->Next = NULL
```

free curNode
Complexity:
Time Complexity: O(n)
Space Complexity: O(1)
Deleting Node from position 'p' in the List

To delete a Node at the position 'p' we have to first traverse the list until we reach the position 'p'. For this case have to maintain two pointers namely prevNode and curNode. Since Singly Linked Lists are uni-directional we have to maintain the information about previous Node in prevNode. Once we reach the position 'p' we have to modify prevNode Next pointing to curNode Next and free curNode.

Pseudocode:

```
curNode = head
curPos = 1

forever:

 if curPos == P || curNode == NULL
 break

 prevNode = curNode
 curNode = curNode->Next
 curPos++

if curNode != NULL:

 prevNode->Next = curNode->Next
 free curNode
```

Complexity:
Time Complexity: O(n) worst case
Space Complexity: O(3)
Pseudocode of Selection Sort

SELECTION-SORT
Pseudocode :

```
1.    for j ← 1 to n-1
2.        smallest ← j
3.         for i ← j + 1 to n
4.                if A[ i ] < A[ smallest ]
5.                      smallest ← i
6.          Exchange A[ j ] ↔ A[ smallest ]
```

Time Complexity O(n).
Linear Search

Pseudocode:
Input: Array D, integer key
Output: first index of key in D, or -1 if not found

```
 For i = 0 to last index of D:
   if D[i] equals key:
     return i
 return -1
```

Time Complexity O(n)

DOUBLE LINKED LIST

What is Double Linked List?
In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows...

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...

Here, 'link1' field is used to store the address of the previous node in the sequence, 'link2' field is used to store the address of the next node in the sequence and 'data' field is used to store the actual value of that node.

Example

☀ In double linked list, the first node must be always pointed by head.
☀ Always the previous field of the first node must be NULL.
☀ Always the next field of the last node must be NULL.

Operations
In a double linked list, we perform the following operations...

Insertion
Deletion
Display

Insertion
In a double linked list, the insertion operation can be performed in three ways as follows...

Inserting At Beginning of the list
Inserting At End of the list
Inserting At Specific location in the list
Inserting At Beginning of the list
We can use the following steps to insert a new node at beginning of the double linked list...

Step 1: Create a newNode with given value and newNode → previous as NULL.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty then, assign NULL to newNode → next and newNode to head.
Step 4: If it is not Empty then, assign head to newNode → next and newNode to head.
Inserting At End of the list
We can use the following steps to insert a new node at end of the double linked list...

Step 1: Create a newNode with given value and newNode → next as NULL.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty, then assign NULL to newNode → previous and newNode to head.
Step 4: If it is not Empty, then, define a node pointer temp and initialize with head.
Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
Step 6: Assign newNode to temp → next and temp to newNode → previous.
Inserting At Specific location in the list (After a Node)
We can use the following steps to insert a new node after a node in the double linked list...

Step 1: Create a newNode with given value.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty then, assign NULL to newNode → previous & newNode → next and newNode to head.
Step 4: If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.
Step 5: Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
Step 6: Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.
Step 7: Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → next and newNode to temp2 → previous.
Deletion
In a double linked list, the deletion operation can be performed in three ways as follows...

Deleting from Beginning of the list
Deleting from End of the list
Deleting a Specific Node

Deleting from Beginning of the list
We can use the following steps to delete a node from beginning of the double linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4: Check whether list is having only one node (temp → previous is equal to temp → next)
Step 5: If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)
Step 6: If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.
Deleting from End of the list
We can use the following steps to delete a node from end of the double linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4: Check whether list has only one Node (temp → previous and temp → next both are NULL)
Step 5: If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)
Step 6: If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)
Step 7: Assign NULL to temp → previous → next and delete temp.
Deleting a Specific Node from the list
We can use the following steps to delete a specific node from the double linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.
Step 4: Keep moving the temp until it reaches to the exact node to be deleted or to the last node.
Step 5: If it is reached to the last node, then display 'Given node not found in the list! Deletion not possible!!!' and terminate the fuction.
Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
Step 7: If list has only one node and that is the node which is to be deleted then set head to NULL and delete temp (free(temp)).
Step 8: If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).
Step 9: If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL (head → previous = NULL) and delete temp.
Step 10: If temp is not the first node, then check whether it is the last node in the list (temp

→ next == NULL).
Step 11: If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp (free(temp)).
Step 12: If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next = temp → next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).
Displaying a Double Linked List
We can use the following steps to display the elements of a double linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty, then display 'List is Empty!!!' and terminate the function.
Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.
Step 4: Display 'NULL <--- '.
Step 5: Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node
Step 6: Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).
Front Insertion Pseudo Code
Here is the first pseudo code example. I have tried to write out all the logic inside of a single method so that we don’t have to jump back and forth from multiple method calls (at least until we understand the basics).

```
begin insertAtFront(data):
   if head is null:
      head = new Node(data);
      // Tail and head cannot point at the same node
      tail = null;
   else:
      if tail is null:
         tail = new Node(data);
         // Update references
         // Head --> Tail
         // Head <-- Tail
         head.setNext(tail)
         tail.setPrev(head)
      else:
         Node<T> prevHead = head;
         Node<T> newHead = new Node(data);

         // Update references
         // newHead.next --> prevHead
         // prevHead.prev <-- newHead
         newHead.setNext(prevHead)
         prevHead.setPrev(newHead)
         head = newHead
      end if else;
   end if else;
```

```
    increment size
end insertAtFront;
```

Back Insertion Pseudo Code:

Let's take a look at what all of this logic will look like in pseudo code. Note that the logic almost mirrors that of the front insertion. The only difference between the two is how we handle the reference updates when inserting a head and tail node.

Comparing the two examples will help you wrap your mind around the difference and solidify your working knowledge of the doubly linked list.

```
begin insertAtBack(T dataToInsert):
    // First element.
    if head is null:
        nodeToInsert = new Node(dataToInsert)
        head = nodeToInsert
        // Tail and head cannot point at the same node
        tail = null
    else:
        // Second element to add to list
        if tail is null:
            tail = new Node(dataToInsert)
            // Update references
            // Head --> Tail
            // Head <-- Tail
            head.setNext(this.tail)
            tail.setPrev(this.head)
        else:
            prevTail = tail
            newTail  = new Node(dataToInsert)

            // Update references
            // prevTail --> newTail
            // prevTail <-- newTail
            newTail.setPrev(prevTail)
            prevTail.setNext(newTail)
            tail = newTail
        end if else;
    end if else;
    increment size
end insertAtBack;
```

Middle Insertion Pseudo Code:

Here is the pseudo code for the middle insertion, which is pretty much inserting data based on index. In this implementation, for the sake of simplicity, we will begin iterating from the start of the linked list (the head) until we reach the desired index at which to insert the data.

As a challenge, I recommend you to implement a version that inspects whether it is faster to

iterate from the head or tail and choose the most optimal path. I have included this version in the source code, but try it on your own before looking at the solutions.

```
begin insertAfter(data, index):
    if index is less than size - 1:
        // Handle error here
    end if;
    // Get node at a specific index
    currentNode = head;
    counter = 0;

    // Iterate from the front of the list
    while currentNode is not null and counter != index:
        currentNode = currentNode.getNext()
    end while;
    // Add node
    call insertNode(data, currentNode, currentNode.getNext());
end insertAfter;
```

Below is the pseudo code for the insertNode method.

```
begin insertNode(dataToInsert, currentNode, nextNode):
    newNode = new Node(dataToInsert)
    // If next node is null, current node is the tail node
    if nextNode is null:
        tail = newNode
        tail.setPrev(currentNode)
        currentNode.setNext(newNode)
    else:
        // update references of existing node
        currentNode.setNext(newNode)
        nextNode.setPrev(newNode)
        // update references of new node
        newNode.setPrev(currentNode)
        newNode.setNext(nextNode)
    end if else;
    increment size
end insertNode;
```

Pseudo Code

Now that we have reviewed all the cases, let's take a look at the pseudo code for the entire remove operation.

Remove operation

```
begin remove(T dataToRemove):
        currentNode = head
        while current node is not null:
```

```
            currentData = currentNode.data
            if  currentData equals dataToRemove:
                call removeNode(currentNode)
            end if;
            currentNode = currentNode.next
        end while;
end remove;
removeNode operation

begin removeNode(nodeToRemove):
    prevNode = nodeToRemove.prev
    nextNode = nodeToRemove.next
    // Head node does not have a previous node
    if prevNode is null:
        head = null
        head = nextNode
        head.prev = null
    else if nextNode is null:
        // Tail does not have a next node
        tail = null
        tail = prevNode
        tail.next = null
    else:
        // Is somewhere in the middle of the linked list
        // Set current node to null
        nodeToRemove = null
        // connect the previous and next nodes together
        prevNode.next = nextNode
        nextNode.prev = prevNode
    end if else statement;
    decrement size
end removeNode;
```

SELECTION-SORT

Pseudocode :

```
1.    for j ← 1 to n-1
2.        smallest ← j
3.         for i ← j + 1 to n
4.                if A[ i ] < A[ smallest ]
5.                      smallest ← i
6.          Exchange A[ j ] ↔ A[ smallest ]
```

Time Complexity O(n).

Linear Search

Pseudocode:

```
# Input: Array D, integer key
# Output: first index of key in D, or -1 if not found
```

```
For i = 0 to last index of D:
  if D[i] equals key:
    return i
return -1
```

Time Complexity O(n).