Ex No: 4 Implementation of Linked List ADT

Linked list is a very important dynamic data structure. Basically, there are two types of linked list, singly-linked list and doubly-linked list. In a singly-linked list every element contains some data and a link to the next element, which allows keeping the structure. On the other hand, every node in a doubly-linked list also contains a link to the previous node. Linked list can be an underlying data structure to implement stack, queue or sorted list.

Each cell is called a node of a singly-linked list. First node is called head and it's a dedicated node. By knowing it, we can access every other node in the list. Sometimes, last node, called tail, is also stored in order to speed up add operation. For empty list, head is set to NULL. Every node of a singly-linked list contains following information:

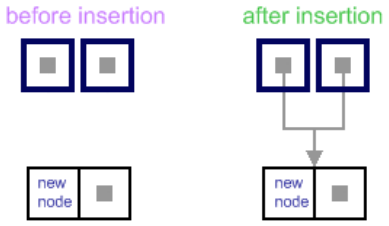
- a value (user's data);
- a link to the next element (auxiliary data).

Addition (insertion) operation:

Insertion into a singly-linked list has two special cases. It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list). In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list. There is a description of all these cases below.

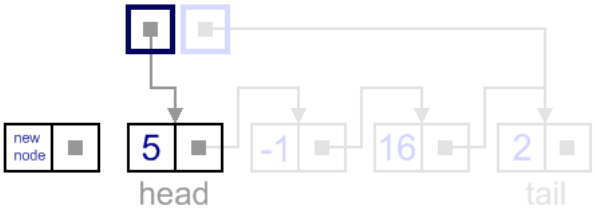
Empty list case

When list is empty, which is indicated by (head == NULL) condition, the insertion is quite simple. Algorithm sets both head and tail to point to the new node.



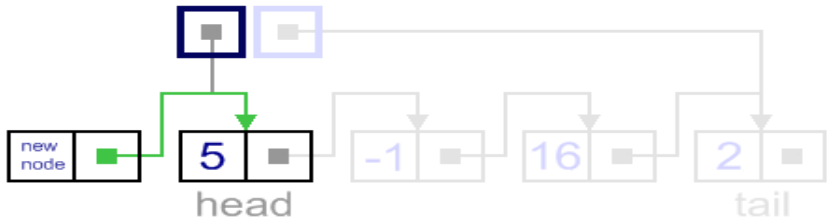
Add first

In this case, new node is inserted right before the current head node.

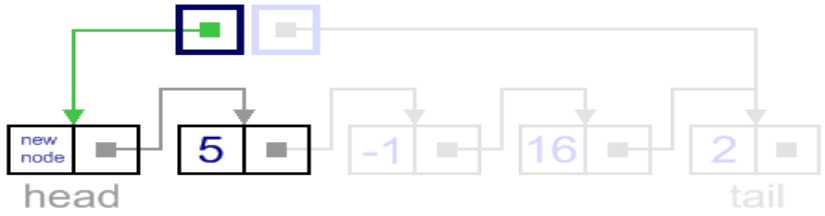


It can be done in two steps:

1. Update the next link of a new node, to point to the current head node.



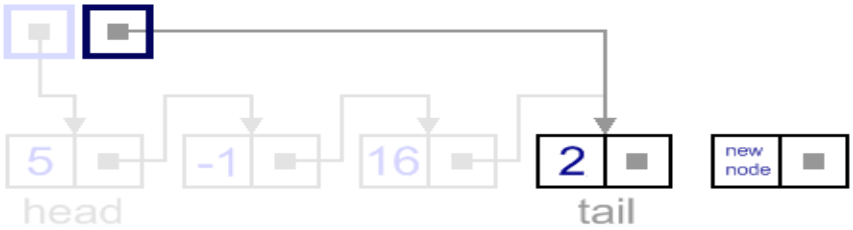
2. Update head link to point to the new node.




```
procedure addFirst(SinglyLinkedListNode newNode)
{
        if (newNode == null)
            return;
        else {
            if (head == null) {
                newNode->next = null;
                head = newNode;
                tail = newNode;
            } else {
                newNode->next = head;
                head = newNode;
            }
        }
}
```

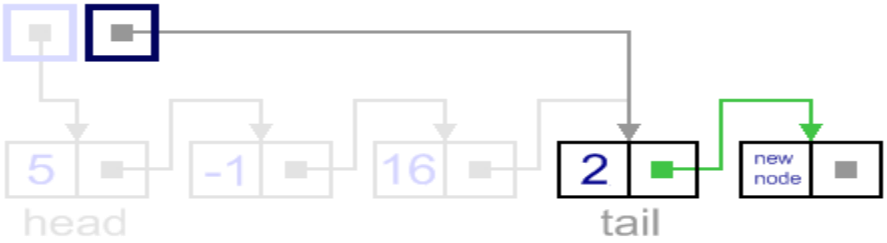
Add last

In this case, new node is inserted right after the current tail node.

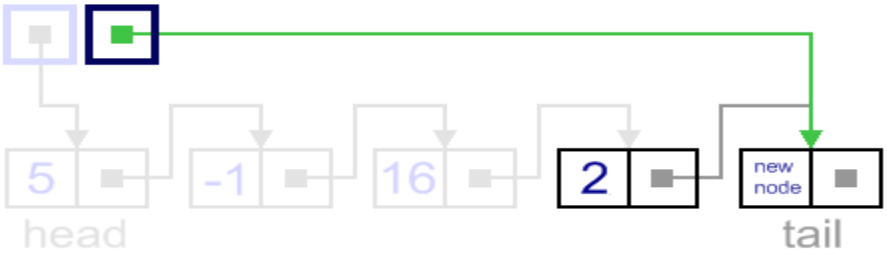


It can be done in two steps:

1. Update the next link of the current tail node, to point to the new node.



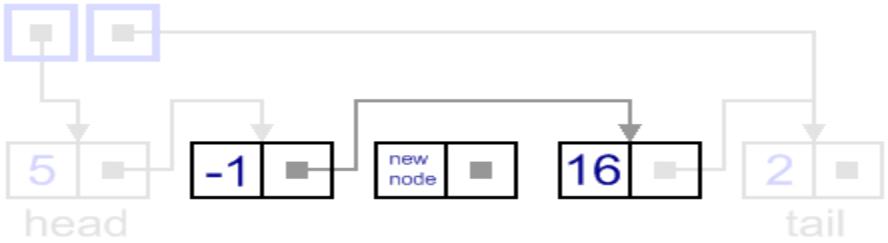
2. Update tail link to point to the new node.




```
procedure addLast(SinglyLinkedListNode newNode)
{
    if (newNode == null)
       return;
    else {
       newNode->next = null;
       if (head == null) {
       head = newNode;
       tail = newNode;
       } else {
             tail->next = newNode;
             tail = newNode;
             }
        }
}
```

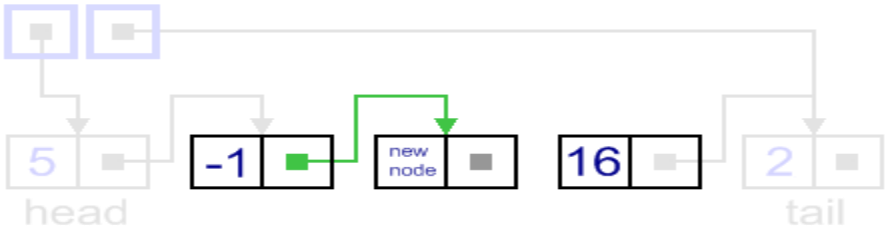
General case

In general case, new node is always inserted between two nodes, which are already in the list. Head and tail links are not updated in this case.

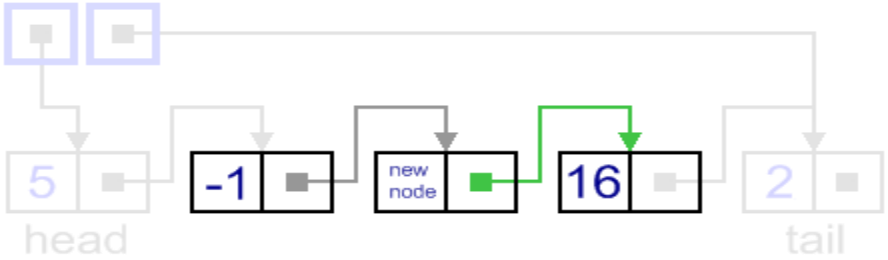


Such an insert can be done in two steps:

1. Update link of the "previous" node, to point to the new node.



2. Update link of the new node, to point to the "next" node.




```
procedure insertAfter(SinglyLinkedListNode previous,
            SinglyLinkedListNode newNode)
{
        if (newNode == null)
            return;
        else {
            if (previous == null)
                addFirst(newNode);
            else if (previous == tail)
                addLast(newNode);
            else {
                SinglyLinkedListNode temp = previous->next;
                previous->next = newNode;
                newNode->next = temp;
            }
        }
}
```

Removal (deletion) operation.

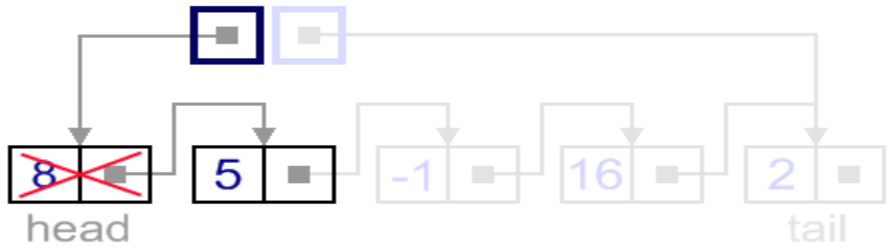
There are four cases, which can occur while removing the node. These cases are similar to the cases in add operation. We have the same four situations, but the order of algorithm actions is opposite. Notice, that removal algorithm includes the disposal of the deleted node, which may be unnecessary in languages with automatic garbage collection (i.e., Java).

List has only one node

When list has only one node, which is indicated by the condition, that the head points to the same node as the tail, the removal is quite simple. Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to NULL.

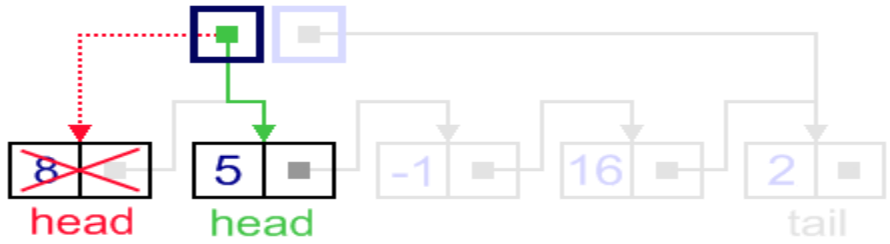
Remove first

In this case, first node (current head node) is removed from the list.

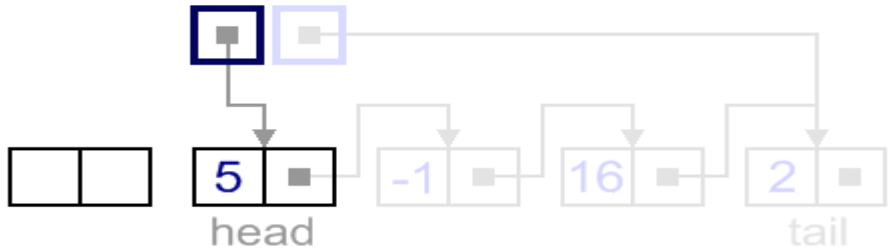


It can be done in two steps:

1. Update head link to point to the node, next to the head.



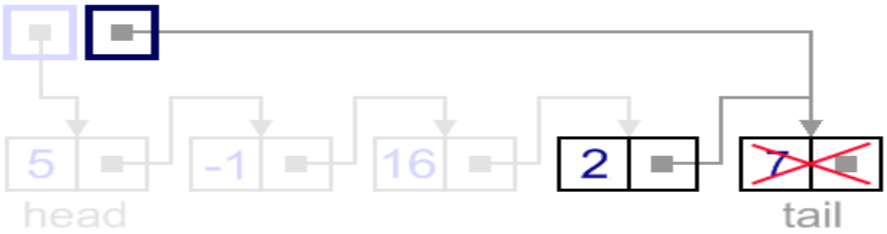
2. Dispose removed node



```
procedure removeFirst()
{
        if (head == null)
            return;
         else {
            if (head == tail) {
                head = null;
                tail = null;
            } else {
                head = head->next;
            }
        }
}
```

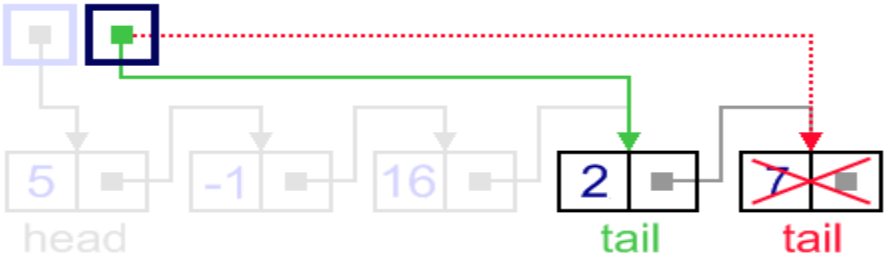
Remove last

In this case, last node (current tail node) is removed from the list. This operation is a bit trickier, than removing the first node, because algorithm should find a node, which is previous to the tail first.

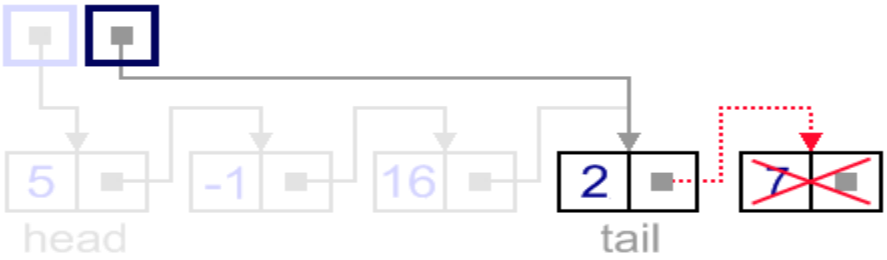


It can be done in three steps:

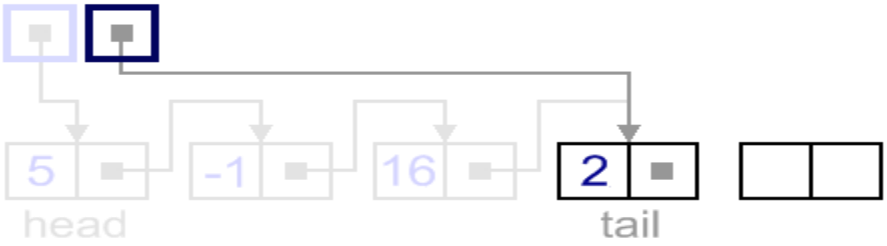
1. Update tail link to point to the node, before the tail. In order to find it, list should be traversed first, beginning from the head.



2. Set next link of the new tail to NULL.



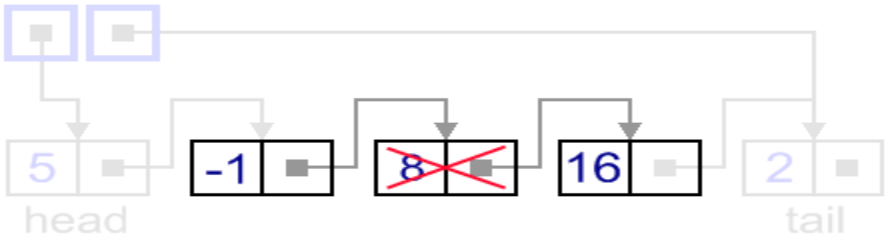
3. Dispose removed node.




```
procedure void removeLast()
{
        if (tail == null)
            return;
        else {
            if (head == tail) {
                head = null;
                tail = null;
            } else {
                SinglyLinkedListNode tmp = head;
                while (tmp->next != tail)
                    tmp = tmp->next;
                tail = tmp;
                tail->next = null;
            }
        }
}
```

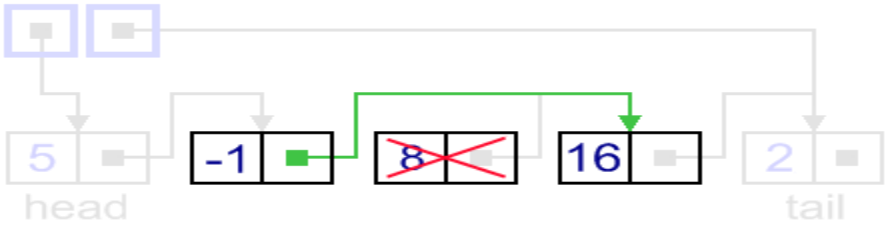
General case

In general case, node to be removed is always located between two list nodes. Head and tail links are not updated in this case.

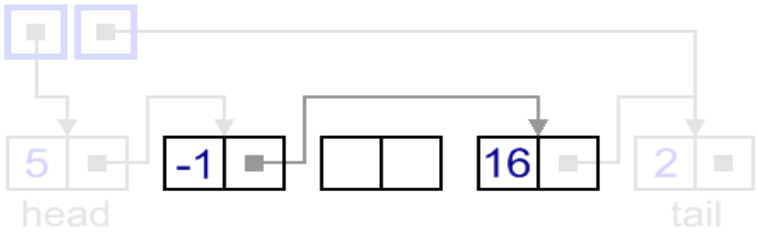


Such a removal can be done in two steps:

1. Update next link of the previous node, to point to the next node, relative to the removed node.



2. Dispose removed node




```
procedure void removeNext(SinglyLinkedListNode previous)
{
        if (previous == null)
            removeFirst();
        else if (previous->next == tail) {
            tail = previous;
            tail.next = null;
        } else {
            SinglyLinkedListNode tmp = previous->next;
            previous->next = tmp->next;
            free(tmp);
        }
    }
}
```

Sample Questions, but not limited to

1. Let L1 and L2 be two independent lists. Write procedure mergeLists() such that L2 comes after L1. For example, L1 = { 1, 4, 7} L2 = { 2, 3, 9} Result = {1, 4, 7, 2, 3, 9}
2. Let L be a singly linked list. Write procedure countNodes() such that it returns number of nodes present in the list. If L = { 1, 4, 7, 2, 3, 9 } Result = 6
3. Write a procedure getNthItem() such that it takes a linked list and an integer index and returns the data value stored in the node at that index position. For ex. If the list contains {42, 13, 666} and the integer value is 1, then getNthItem() should return 13. Assume the index ranges from [0 .. N-1]. If not, return error status.
4. L1 and L2 are two sorted list. Merge two lists.

5. Number of occurrences of given item in the list
6. FrontBackSplit() - {1,2,3,4,5} => {1,2,3} and {4,5} based on the length
7. RemoveDuplicates() in sorted list . Should have one traversal
8. MoveNode - remove from one list, append it to the first of the other node (similar to pop() of stack)
9. AlternateSplit() - {a, b, a, b, a} => {a,a,a} and {b,b}
10. ShuffleMerge() - {1, 2, 3} and {7, 13, 1} => {1, 7, 2, 13, 3, 1}
11. L1 U L2
12. L1 ∩ L2
13. Reverse singly linked list
14. Write a function to count the number of distinct elements in a list.
15. Write a function to change the element at position p in a list L to x. The function is undefined if L does not currently have any element at position p.
16. Substring (L, p, q) : extracts the substring of L from position p to position q (excluding it).
17. first_occurrence (L , x) : returns the position at which x occurs first in L, undefined if x does not occur.

Assessment Rubrics for Ex 4:

Parameters	**Max Marks**	**Actual Marks**
Linked List ADT procedure with diagrams	15	
Linked List application using ADT*(atleast two)*	15	
Viva (Online Test)	10	
Adherence to the template for documentation (Record)	10	
Total	50	
Staff Signature		