LINKED LIST

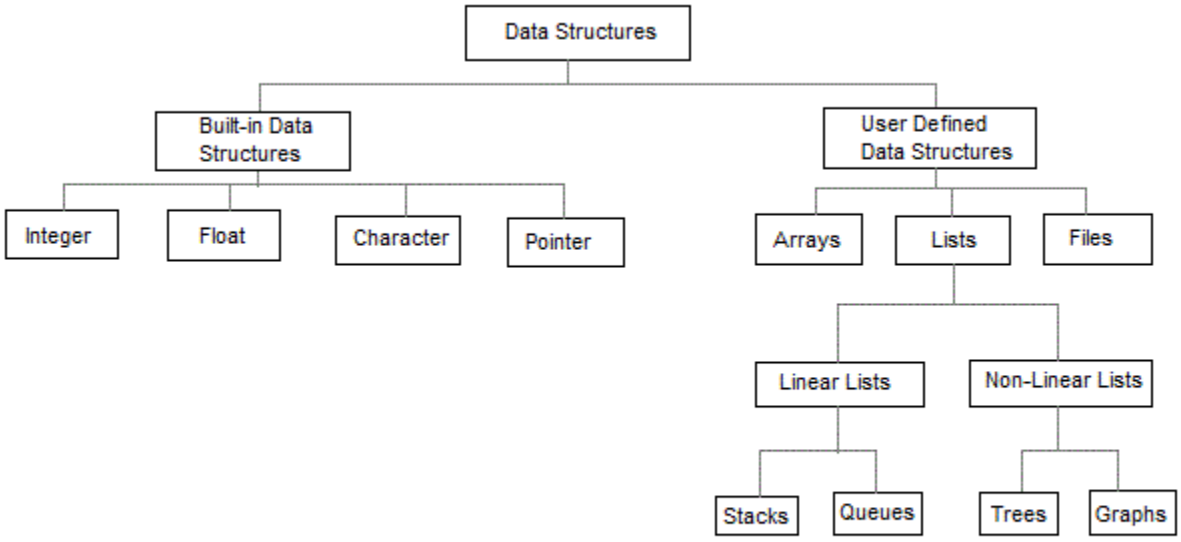
Basic types of Data Structures

As we discussed above, anything that can store data can be called as a data strucure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



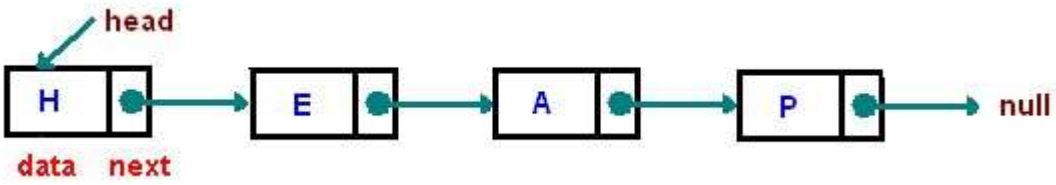
INTRODUCTION TO DATA STRUCTURES

Introduction to Linked Lists

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.

One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions. In this chapter we consider another data structure called Linked Lists that addresses some of the limitations of arrays.

A linked list is a linear data structure where each element is a separate object.



Each element (we will call it a **node**) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a reference to the next node.

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

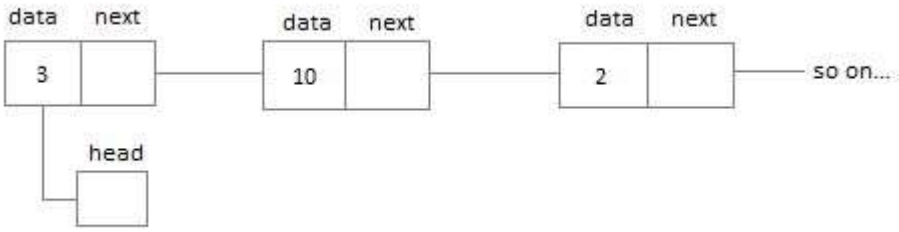
Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don’t need to know the size in advance.

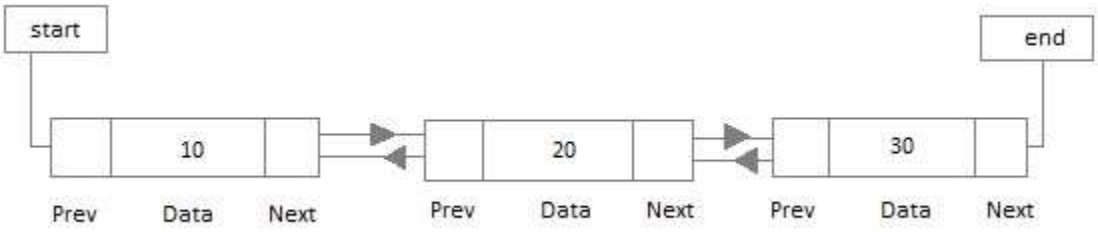
Types of Linked Lists

- **Singly Linked List :** Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in

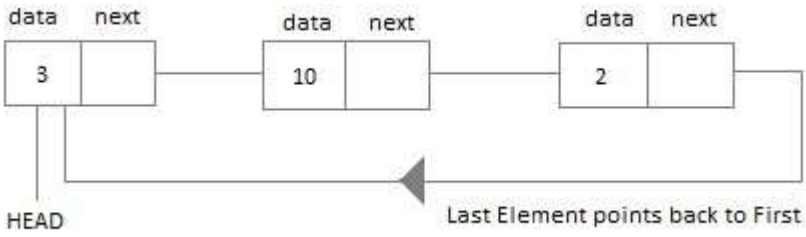
sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



- **Doubly Linked List :** In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



- **Circular Linked List :** In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



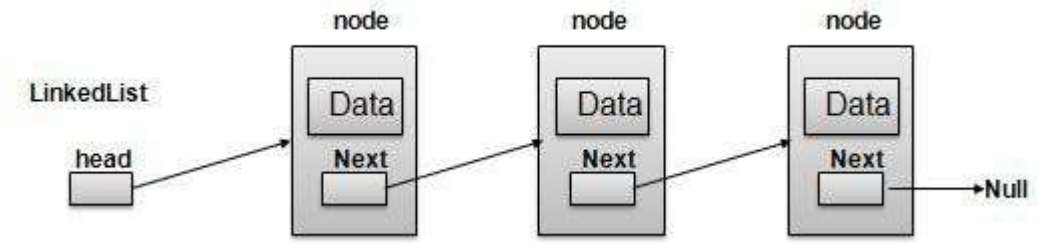
Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List.

- **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



As per above shown illustration, following are the important points to be considered.

- 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.

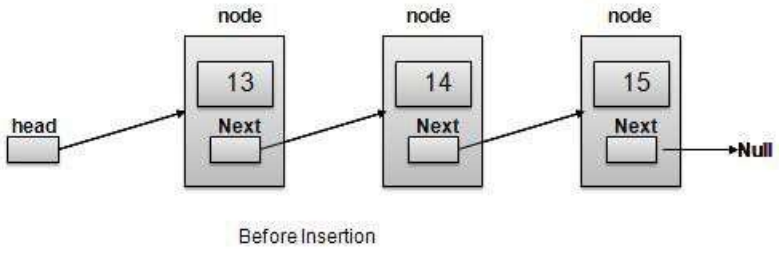
- **Display** – displaying complete list.
- **Search** – search an element using given key.

- **Delete** – delete an element using given key.

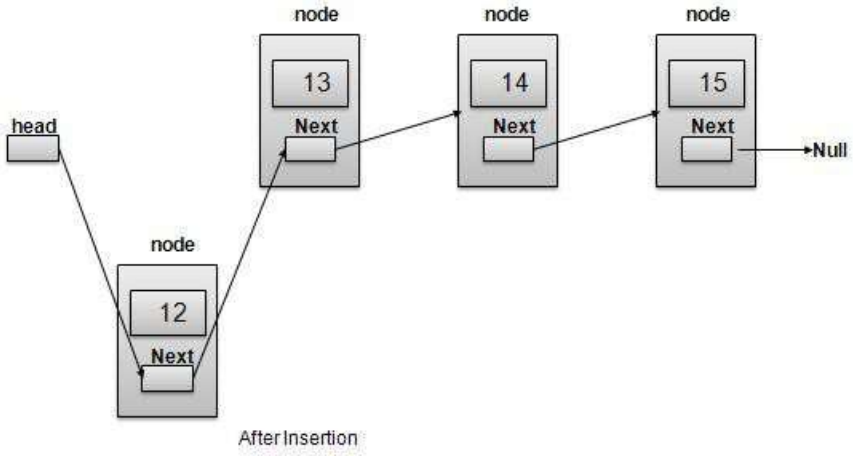
Insertion Operation

Insertion is a three step process –

- Create a new Link with provided data.
- Point New Link to old First Link.
- Point First Link to this New Link.



Before Insertion



After Insertion

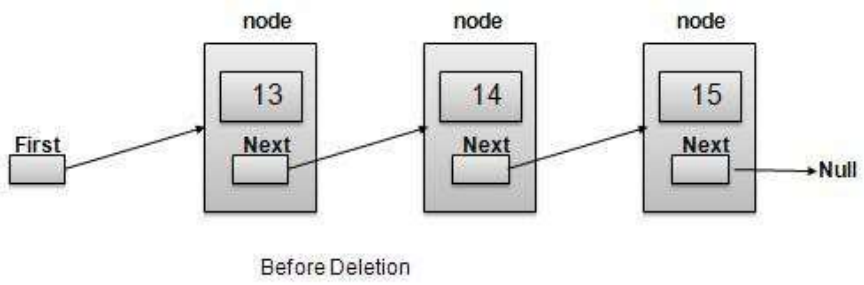
```
//insert link at the first location
void insertFirst(int key, int data){
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;
   //point it to old first node
```

```
   link->next = head;
   //point first to new first node
   head = link;
}
```

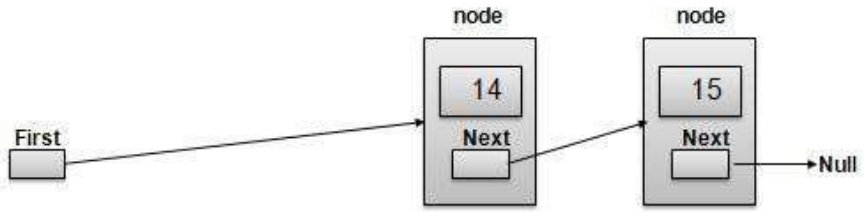
Deletion Operation

Deletion is a two step process –

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



Before Deletion



After Deletion

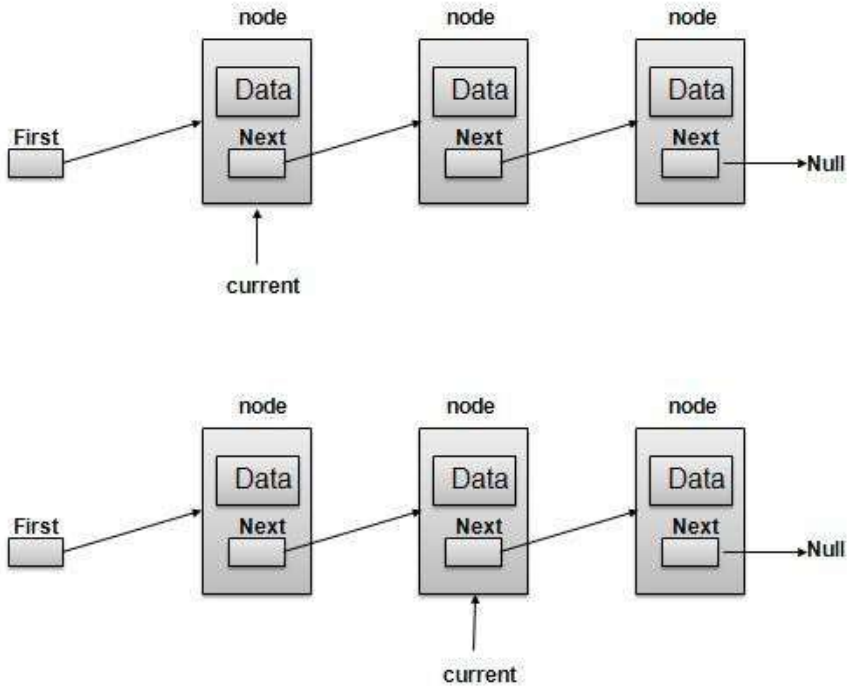
```
//delete first item
struct node* deleteFirst(){
   //save reference to first link
   struct node *tempLink = head;
   //mark next to first link as first
   head = head->next;
```

```
   //return the deleted link
   return tempLink;
}
```

Navigation Operation

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

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



Note −

```
//display the list
void printList(){
   struct node *ptr = head;
```

```
   printf("\n[ ");
   //start from the beginning
   while(ptr != NULL){
      printf("(%d,%d) ",ptr->key,ptr->data);
      ptr = ptr->next;
   }
   printf(" ]");
}
```

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.

Sort Operation

We've used bubble sort to sort a list.

```
void sort(){
   int i, j, k, tempKey, tempData ;
   struct node *current;
   struct  node  *next;
   int size = length();
   k = size ;
   for ( i = 0 ; i < size - 1 ; i++, k-- ) {
      current = head ;
      next = head->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.

```
void reverse(struct node** head_ref) {
  struct node* prev   = NULL;
  struct node* current = *head_ref;
  struct node* next;
  while (current != NULL) {
    next  = current->next;
    current->next = prev;
    prev = current;
    current = next;
  }
```

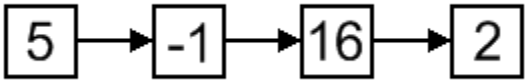
```
    *head_ref = prev;
}
```

Singly-linked list

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 to keep 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.

Example

Sketchy, singly-linked list can be shown like this:



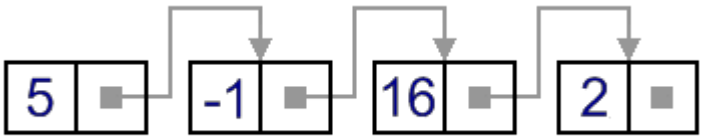
Each cell is called a **node** of a singly-linked list. First node is called **head** and it's a dedicated node. By knowing it, we can access every other node in the list. Sometimes, last node, called **tail**, is also stored in order to speed up add operation.

Singly-linked list. Internal representation.

Every node of a singly-linked list contains following information:

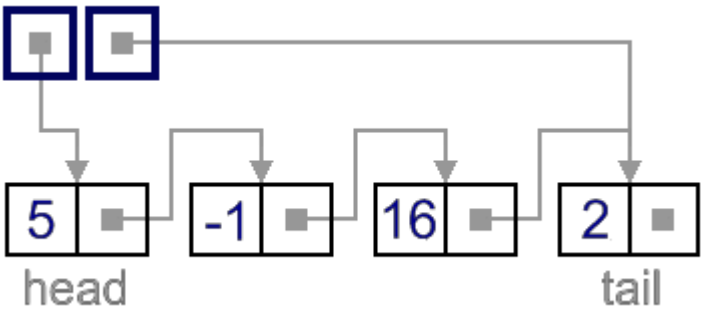
- a value (user's data);
- a link to the next element (auxiliary data).

Sketchy, it can be shown like this:



First node called **head** and no other node points to it. Link to the head is usually stored it the class, which provides an interface to the resulting data structure. For empty list, head is set to *NULL*.

Also, it makes sense to store a link to the last node, called **tail**. Though no node in the list can be accessed from the tail (because we can move forward only), it can accelerate an add operation, when adding to the end of the list. When list is big, it reduces add operation complexity essentially, while memory overhead is insignificant. Below you can see another picture, which shows the whole singly-linked list internal representation:




```
class SinglyLinkedListNode {
public:
    int value; SinglyLinkedListNode
    *next; SinglyLinkedListNode(int
    value) {
        this->value = value;
        next = NULL;
    }
};
class SinglyLinkedList {
private:
    SinglyLinkedListNode *head;
```

```
    SinglyLinkedListNode *tail;
public:
    SinglyLinkedList() {
        head = NULL;
        tail = NULL;
    }
}
```

Singly-linked list. Traversal.

Assume, that we have a list with some nodes. Traversal is the very basic operation, which presents as a part in almost every operation on a singly-linked list. For instance, algorithm may traverse a singly-linked list to find a value, find a position for insertion, etc. For a singly-linked list, only forward direction traversal is possible.

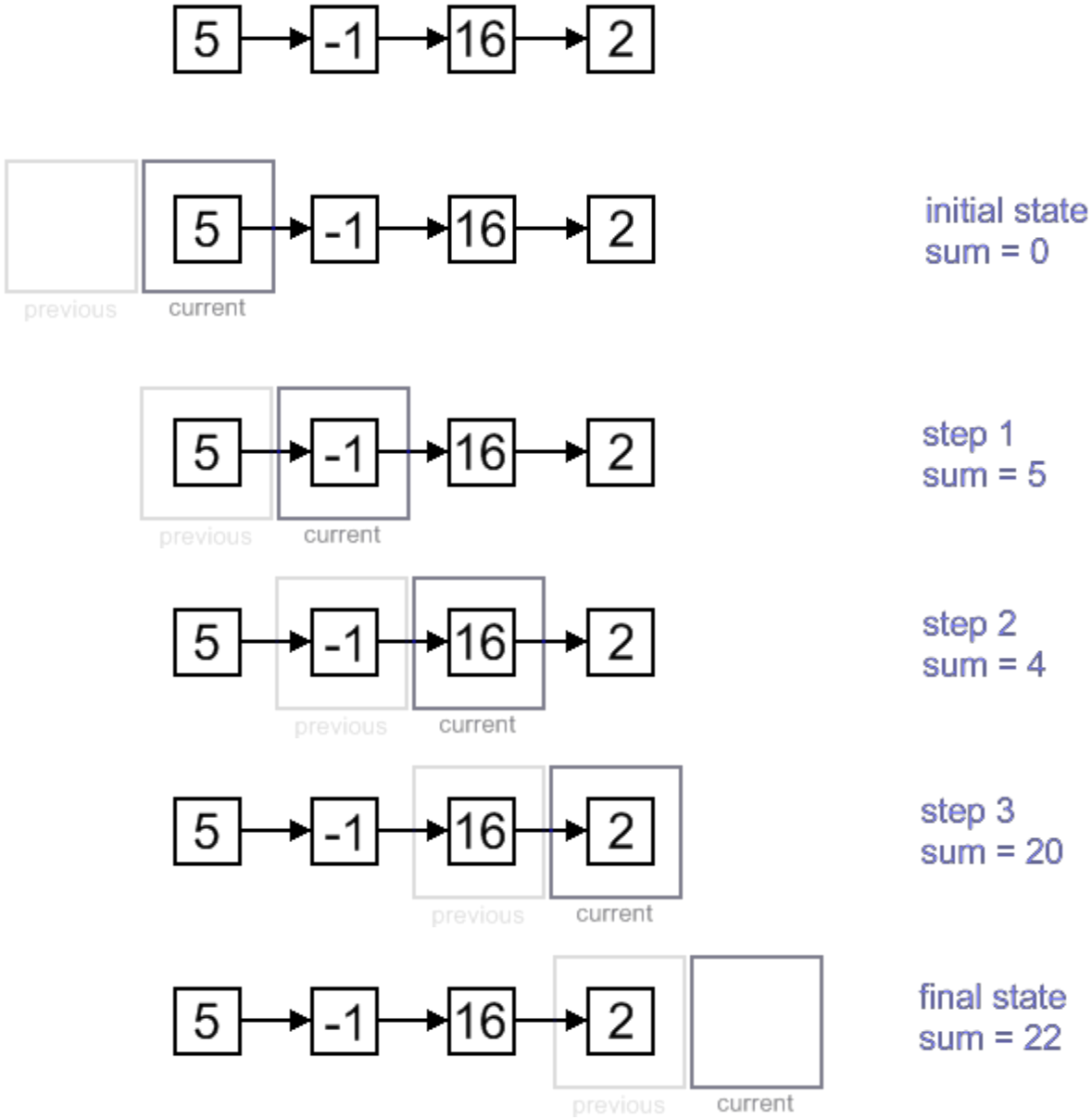
Traversal algorithm

Beginning from the head,

1. check, if the end of a list hasn't been reached yet;
2. do some actions with the current node, which is specific for particular algorithm;
3. current node becomes previous and next node becomes current. Go to the step 1.

Example

As for example, let us see an example of summing up values in a singly-linked list.



For some algorithms tracking the previous node is essential, but for some, like an example, it's unnecessary. We show a common case here and concrete algorithm can be adjusted to meet it's individual requirements.

```
int SinglyLinkedList::traverse() {
    int sum = 0;
    SinglyLinkedListNode *current = head;
```

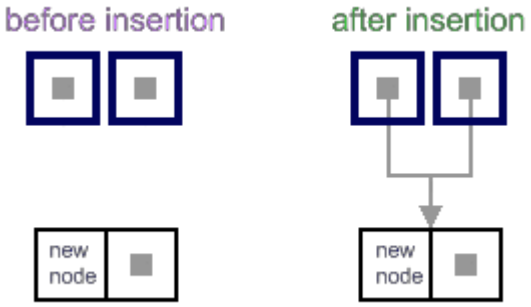
```
    SinglyLinkedListNode *previous = NULL;
    while (current != NULL) {
        sum += current->value;
        previous = current;
        current = current->next;
    }
    return sum;
}
```

Singly-linked list. 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.

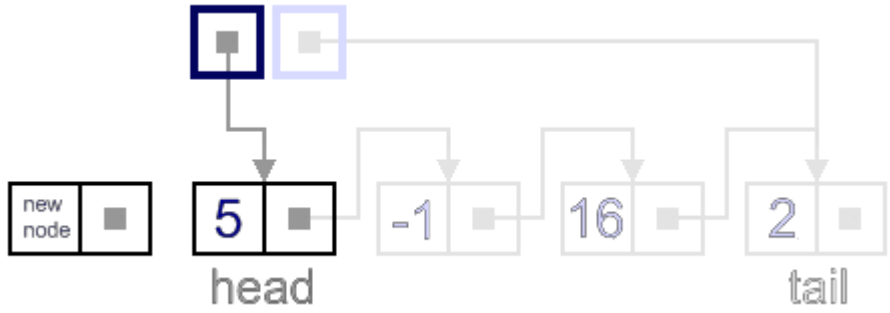
Empty list case

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.



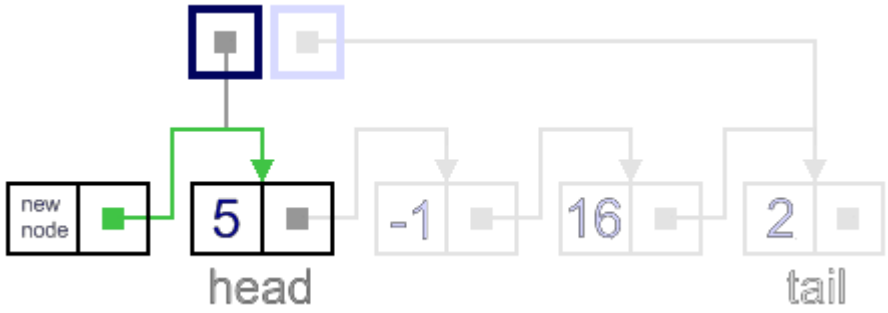
Add first

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

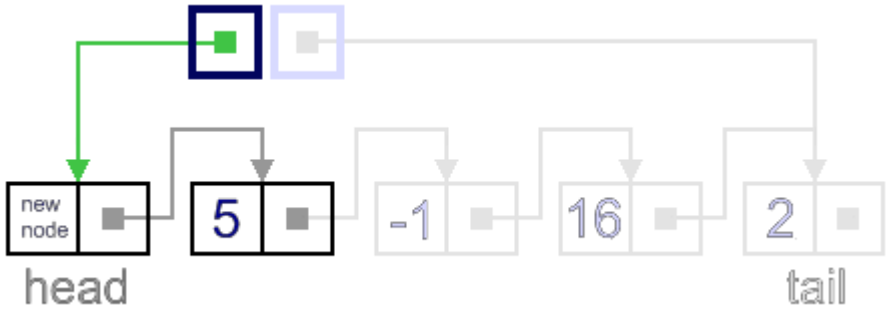


It can be done in two steps:

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

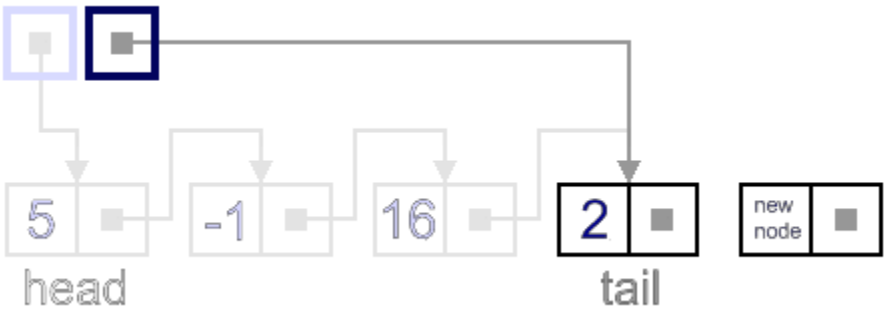


2. Update head link to point to the new node.



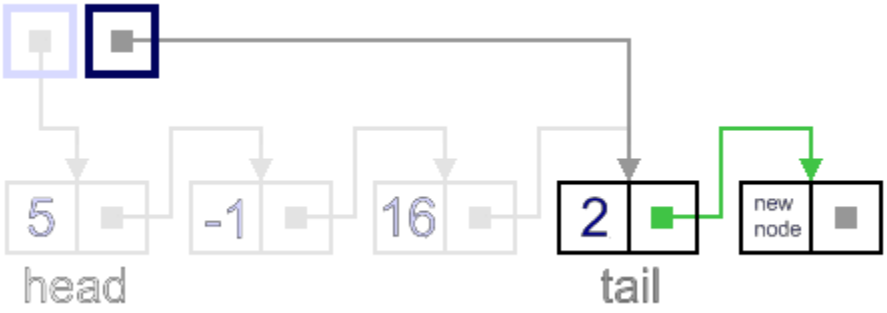
Add last

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

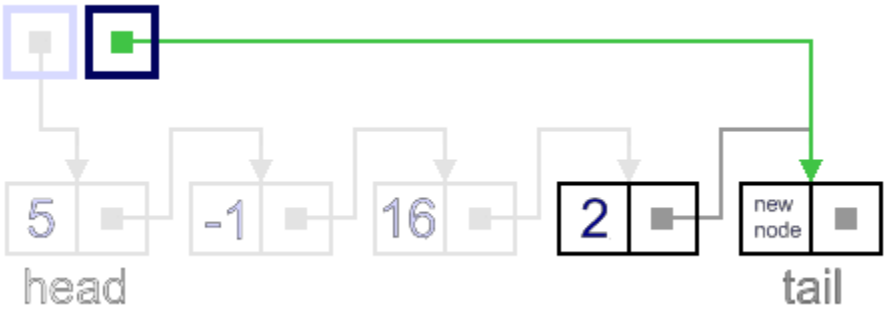


It can be done in two steps:

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

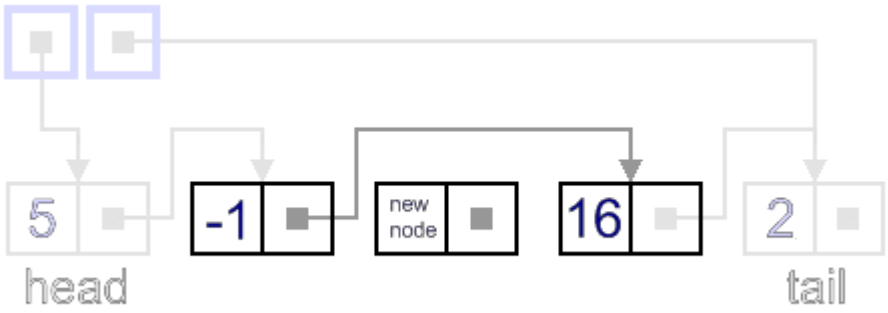


2. Update tail link to point to the new node.



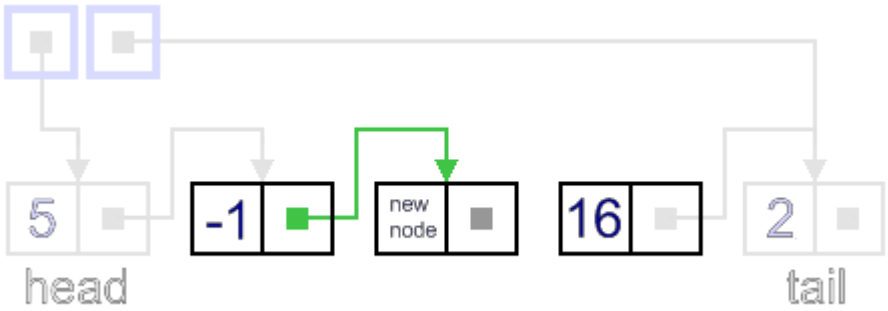
General case

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.

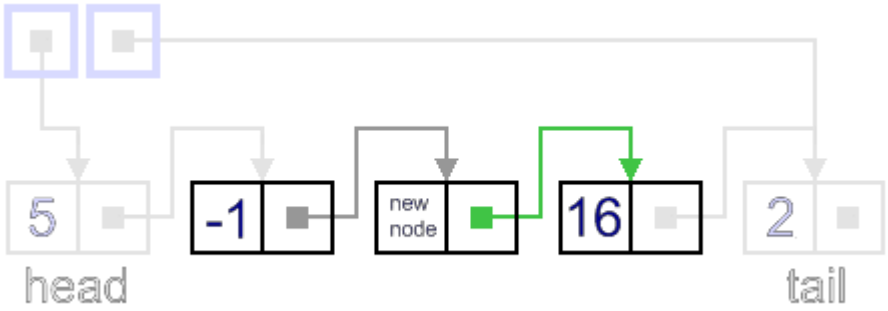


Such an insert can be done in two steps:

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.



Algorithm to insert node in single Linked List

```
void SinglyLinkedList::addLast(SinglyLinkedListNode *newNode) {
    if (newNode == NULL)
        return;
    else {
```

```
        newNode->next = NULL;
        if (head == NULL) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
    }
}

void SinglyLinkedList::addFirst(SinglyLinkedListNode *newNode) {
    if (newNode == NULL)
        return;
    else {
        if (head == NULL) {
            newNode->next = NULL;
            head = newNode;
            tail = newNode;
        } else {
            newNode->next = head;
            head = newNode;
        }
    }
}
```

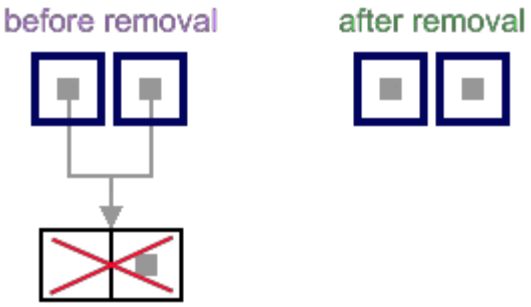
```
void SinglyLinkedList::insertAfter(SinglyLinkedListNode *previous,
        SinglyLinkedListNode *newNode) {
    if (newNode == NULL)
        return;
    else {
        if (previous == NULL)
            addFirst(newNode);
        else if (previous == tail)
            addLast(newNode);
        else {
            SinglyLinkedListNode *next = previous->next;
            previous->next = newNode;
            newNode->next = next;
        }
    }
}
```

Singly-linked list. 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).

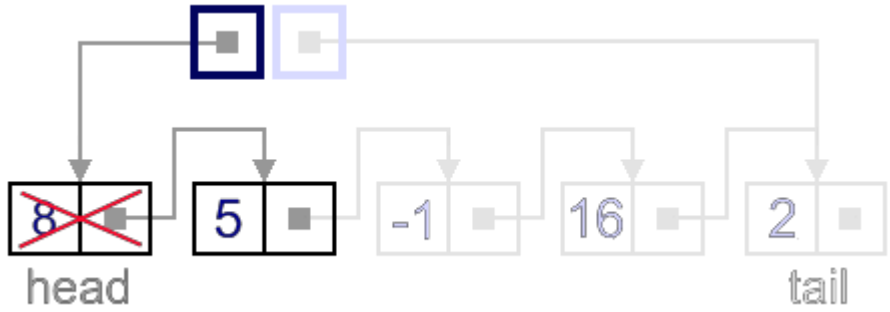
List has only one node

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*.



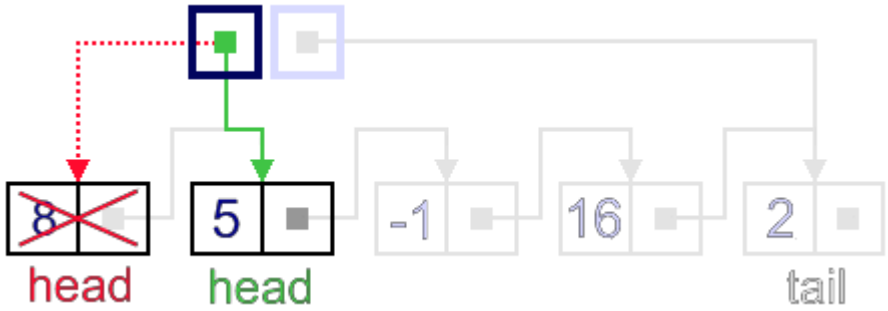
Remove first

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

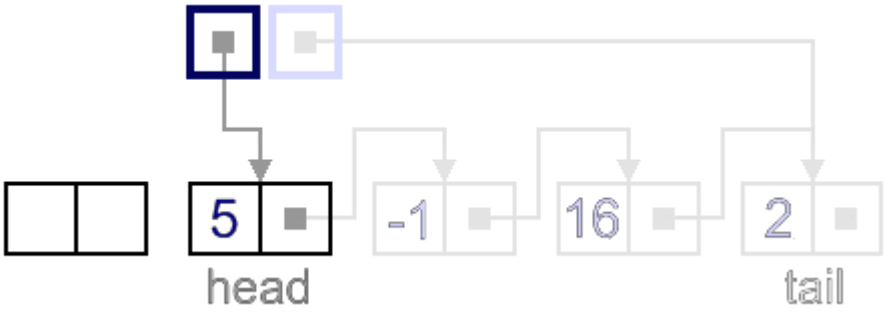


It can be done in two steps:

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

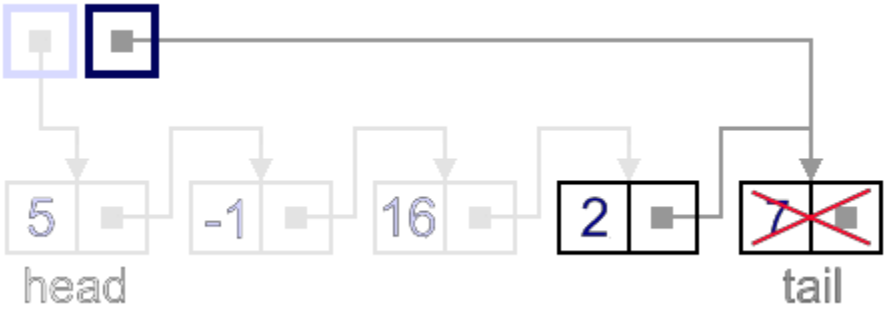


2. Dispose removed node.



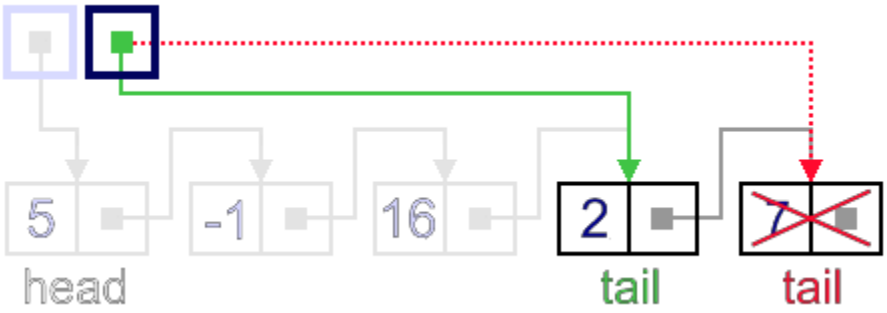
Remove last

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

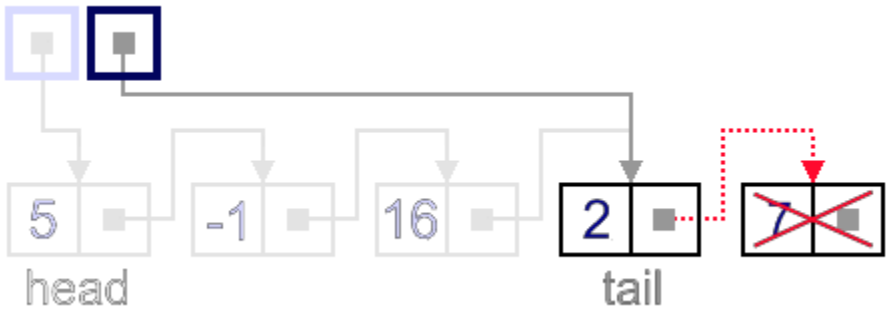


It can be done in three steps:

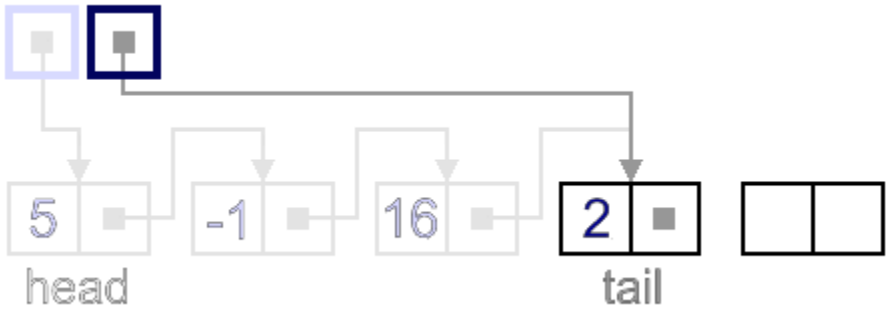
1. Update tail link to point to the node, before the tail. In order to find it, list should be traversed first, beginning from the head.



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

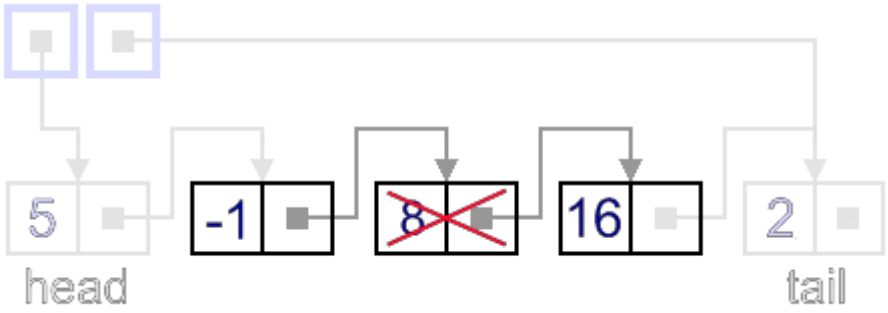


3. Dispose removed node.



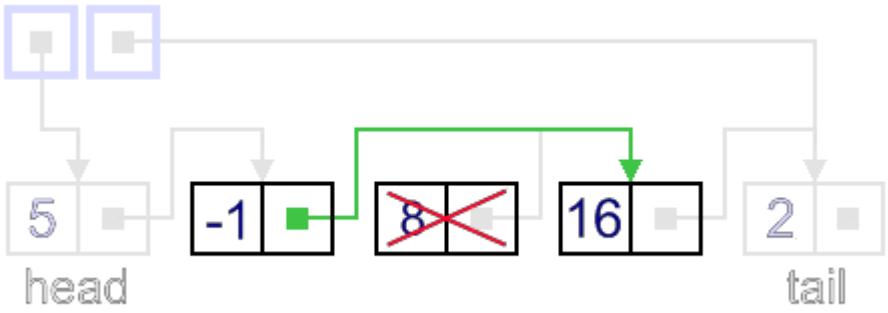
General case

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

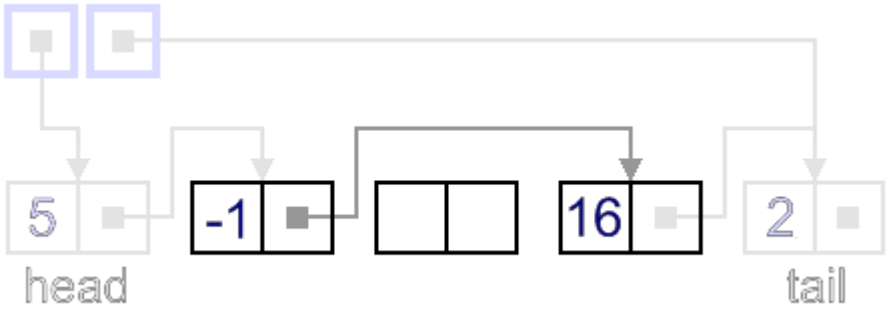


Such a removal can be done in two steps:

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



2. Dispose removed node.



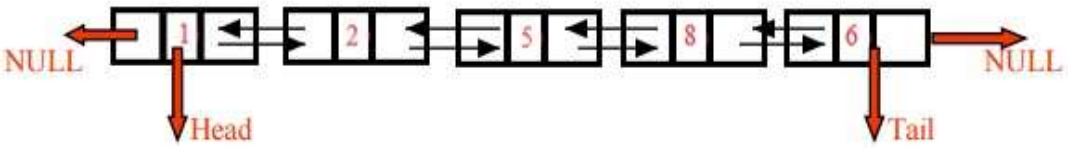

```
void SinglyLinkedList::removeFirst() {
    if (head == NULL)
        return;
    else {
        SinglyLinkedListNode *removedNode;
        removedNode = head;
        if (head == tail) {
            head = NULL;
            tail = NULL;
        } else {
            head = head->next;
        }
        delete removedNode;
```

```
    }
}

void SinglyLinkedList::removeLast() {
    if (tail == NULL)
        return;
    else {
        SinglyLinkedListNode *removedNode;
        removedNode = tail;
        if (head == tail) {
            head = NULL;
            tail = NULL;
        } else {
            SinglyLinkedListNode *previousToTail = head;
            while (previousToTail->next != tail)
                previousToTail = previousToTail->next;
            tail = previousToTail;
            tail->next = NULL;
        }
        delete removedNode;
    }
}

void SinglyLinkedList::removeNext(SinglyLinkedListNode *previous) {
    if (previous == NULL)
```

```
        removeFirst();
    else if (previous->next == tail) {
        SinglyLinkedListNode *removedNode = previous->next;
        tail = previous;
        tail->next = NULL;
        delete removedNode;
    } else if (previous == tail)
        return;
    else {
        SinglyLinkedListNode *removedNode = previous->next;
        previous->next = removedNode->next;
        delete removedNode;
    }
}
```

Doubly-Linked List

Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contain two references (or links) – one to the previous node and other to the next node. The previous link of the first node and the next link of the last node points to NULL. In comparison to singly-linked list, doubly-linked list requires handling of more pointers but less information is required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.



Basic operations of a singly-linked list are:

1. Insert – Inserts a new element at the end of the list.
2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list.

Algorithm:

The node of a linked list is a structure with fields data (which stored the value of the node), *previous (which is a pointer of type node that stores the address of the previous node) and *next (which is a pointer of type node that stores the address of the next node).

Two nodes *start (which always points to the first node of the linked list) and *temp (which is used to point to the last node of the linked list) are initialized. Initially temp = start, temp->prevoius = NULL and temp->next = NULL. Here, we take the first node as a dummy node. The first node does not contain data, but it used because to avoid handling special cases in insert and delete functions.

Functions –

1. Insert – This function takes the start node and data to be inserted as arguments. New node is inserted at the end so, iterate through the list till we encounter the last node. Then, allocate memory for the new node and put data in it. Lastly, store the address of the previous node in the previous field of the new node and address in the next field of the new node as NULL.
2. Delete - This function takes the start node (as pointer) and data to be deleted as arguments. Firstly, go to the node for which the node next to it has to be deleted,
   If that node points to NULL (i.e. pointer->next=NULL) then the element to be deleted is not present in the list.

Else, now pointer points to a node and the node next to it has to be removed, declare a temporary node (temp) which points to the node which has to be removed. Store the address of the node next to the temporary node in the next field of the node pointer (pointer->next = temp->next) and also link the pointer and the node next to the node to be deleted (temp->prev = pointer). Thus, by breaking the link we removed the node which is next to the pointer (which is also temp). Because we deleted the node, we no longer require the memory used for it, free() will deallocate the memory.

3. Find - This function takes the start node (as pointer) and data value of the node (key) to be found as arguments. First node is dummy node so, start with the second node. Iterate through the entire linked list and search for the key.

   Until next field of the pointer is equal to NULL, check if pointer->data = key. If it is then the key is found else, move to the next node and search (pointer = pointer -> next). If key is not found return 0, else return 1.

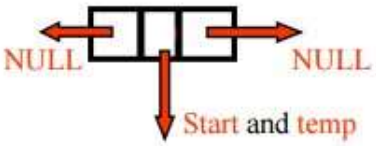
4. Print - function takes the start node (as pointer) as an argument. If pointer = NULL, then there is no element in the list. Else, print the data value of the node (pointer->data) and move to the next node by recursively calling the print function with pointer->next sent as an argument.

Performance:

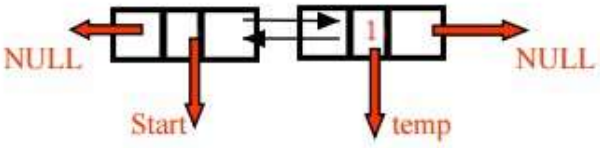
1. The advantage of a singly linked list is that we don't need to keep track of the previous node for traversal or no need of traversing the whole list for finding the previous node.
2. The disadvantage is that more pointers needs to be handled and more link need to updated.

Example:

Initially

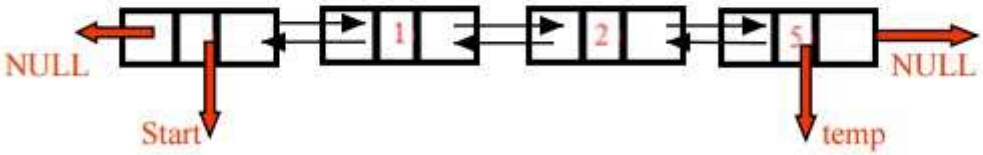


Insert(Start,1) - A new node with data 1 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.

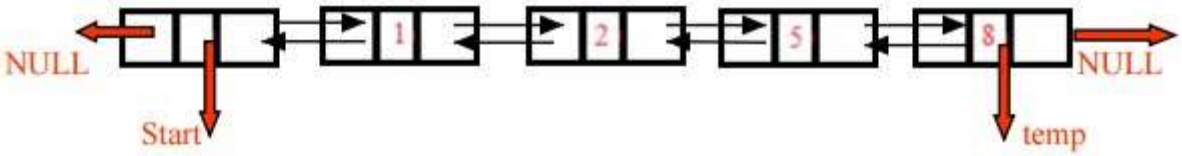


Insert(Start,2) - A new node with data 2 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.

Insert(Start,5) - A new node with data 5 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.

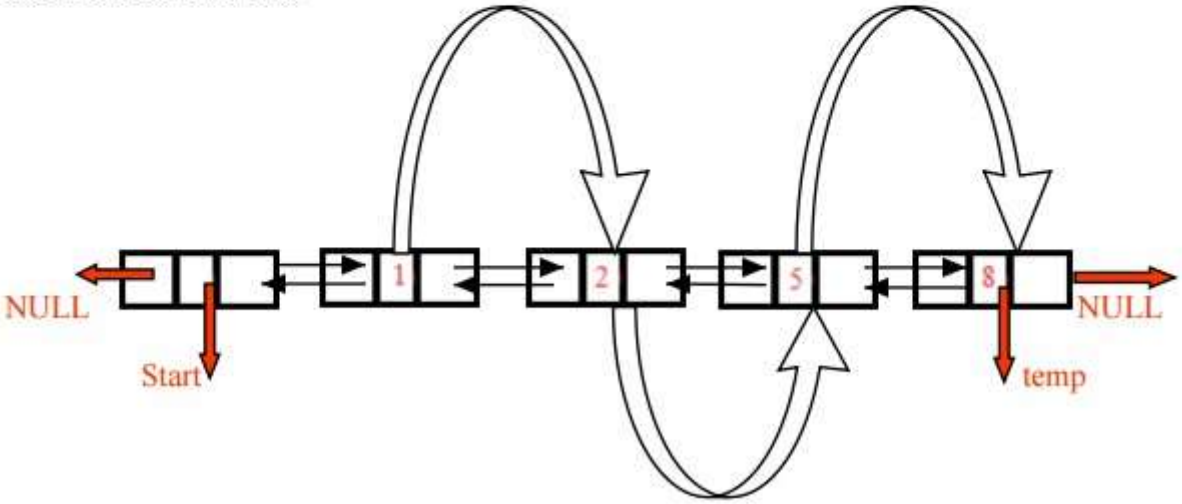


Insert(Start,8) - A new node with data 8 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.

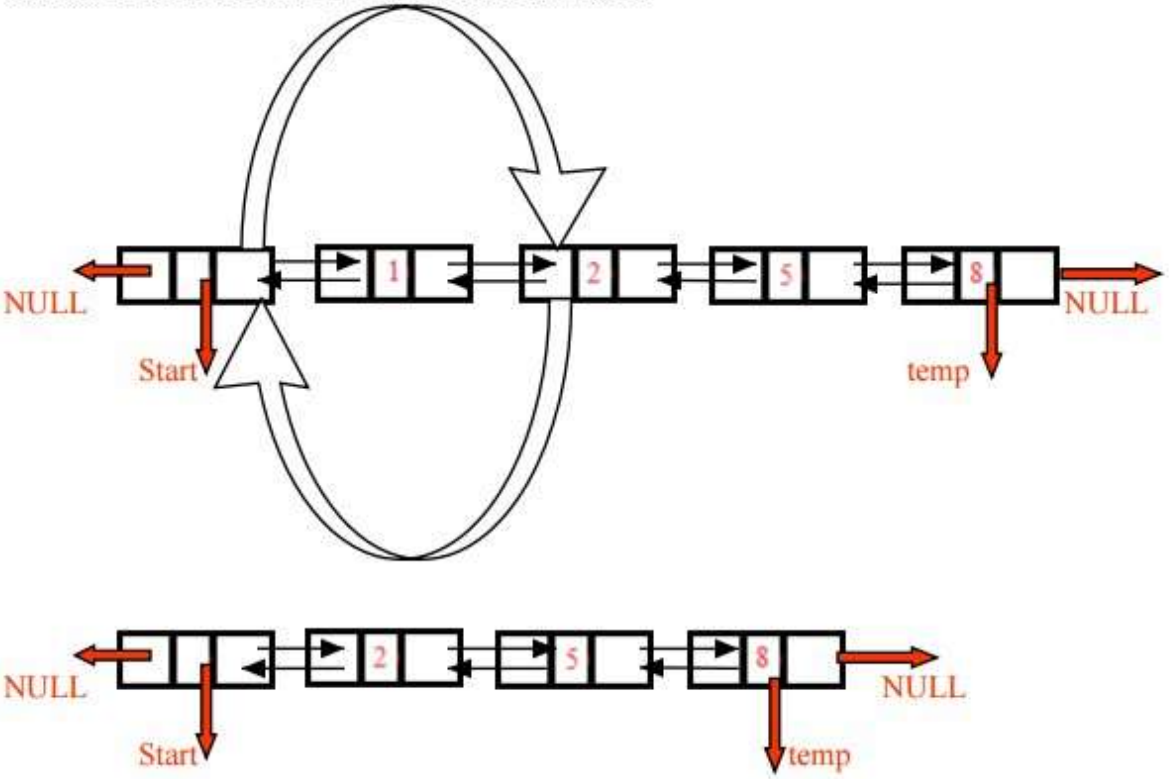


Print(start) 1, 2, 5, 8

To print start from the first node of the list and move to the next with the help of the address stored in the next field.

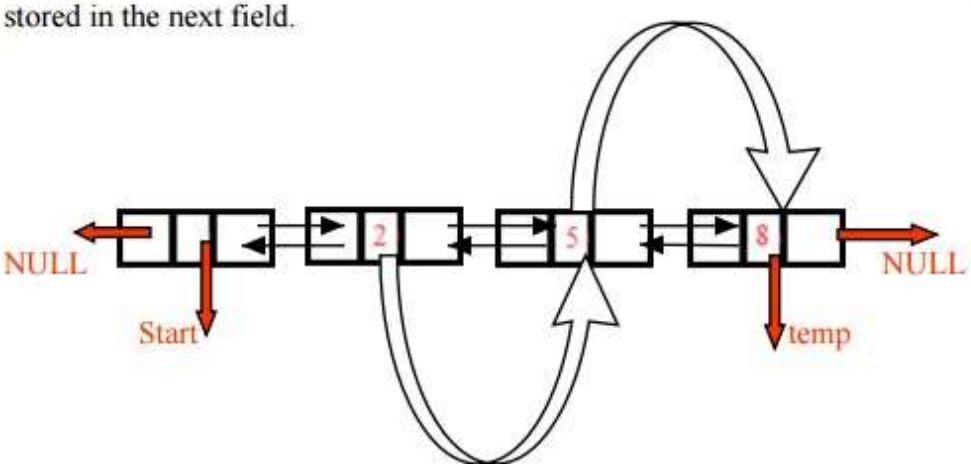


Delete(start,1) - A node with data 1 is found, the next and previous fields are updated to store the NULL value. The next field of previous node is updated to store the address of node next to the deleted node. The previous field of the node next to the deleted one is updated to store the address of the node that is before the deleted node.



Find(start,10) 'Element Not Found'

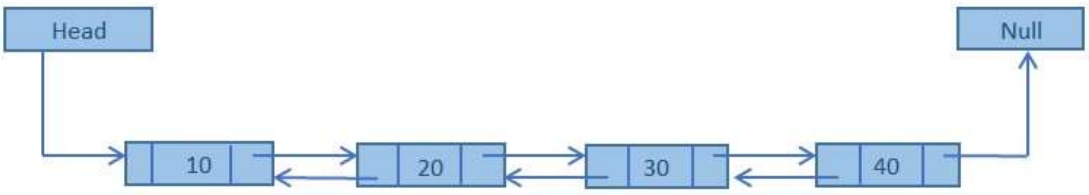
To print start from the first node of the list and move to the next with the help of the address stored in the next field.



Doubly Linked List

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.



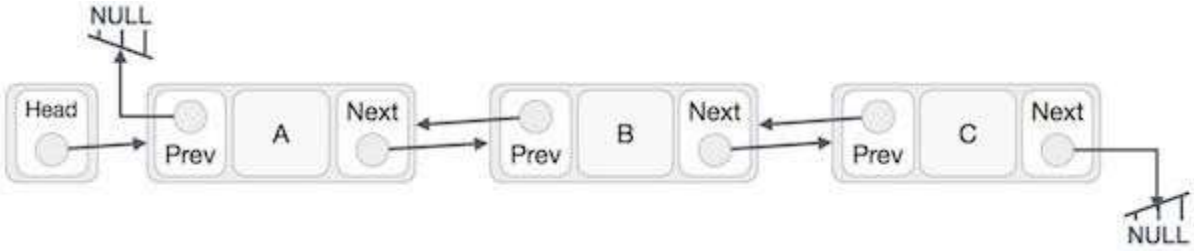
- NEXT holds the memory location of the Previous Node in the List.
- PREV holds the memory location of the Next Node in the List.
- DATA holds Data.

Some Points About Doubly Linked-List

- Doubly Linked List are more convenient than Singly Linked List since we maintain links for bi-directional traversing
- We can traverse in both directions and display the contents in the whole List.
  In Doubly Linked List we can traverse from Head to Tail as well as Tail to
  Head.
- Each Node contains two fields, called Links , that are references to the previous and to the Next Node in the sequence of Nodes.

- The previous link of the first node and the next link of the last node points to NULL.

Doubly Linked List Representation



As per above shown illustration, following are the important points to be considered.

- 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
void insertFirst(int key, int data) {
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;
   if(isEmpty()) {
      //make it the last link
      last = link;
   }else {
      //update first prev link
      head->prev = link;
   }
   //point it to old first link
   link->next = head;
   //point first to new first link
   head = link;
}
```

Deletion Operation

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

```
//delete first item
struct node* deleteFirst() {
   //save reference to first link
   struct node *tempLink = head;
```

```
   //if only one link
   if(head->next == NULL) {
      last = NULL;
   }else {
      head->next->prev = NULL;
   }
   head = head->next;
   //return the deleted link
   return tempLink;
}
```

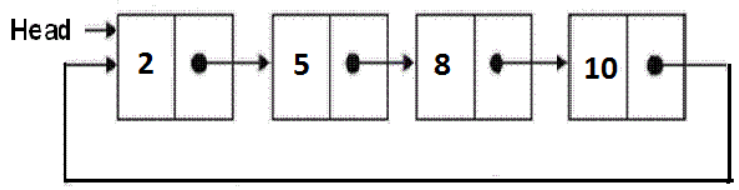
Insertion at End Operation

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

```
//insert link at the last location
void insertLast(int key, int data) {
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;
   if(isEmpty()) {
      //make it the last link
      last = link;
   }else {
      //make link a new last link
```

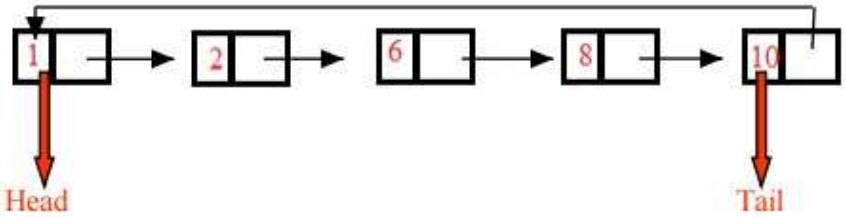
```
        last->next = link;
        //mark old last node as prev of new link
        link->prev = last;
    }
    //point last to new last node
    last = link;
}
```

Circular Linked List



Circular Linked List

Circular linked list is a more complicated linked data structure. In this the elements can be placed anywhere in the heap memory unlike array which uses contiguous locations. Nodes in a linked list are linked together using a next field, which stores the address of the next node in the next field of the previous node i.e. each node of the list refers to its successor and the last node points back to the first node unlike singly linked list. It has a dynamic size, which can be determined only at run time.



Basic operations of a singly-linked list are:

1. Insert – Inserts a new element at the end of the list.
2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list.

Algorithm:

The node of a linked list is a structure with fields data (which stored the value of the node) and *next (which is a pointer of type node that stores the address of the next node).

Two nodes *start (which always points to the first node of the linked list) and *temp (which is used to point to the last node of the linked list) are initialized. Initially temp = start and temp->next = start. Here, we take the first node as a dummy node. The first node does not contain data, but it used because to avoid handling special cases in insert and delete functions.

Functions –

1. Insert – This function takes the start node and data to be inserted as arguments. New node is inserted at the end so, iterate through the list till we encounter the last node. Then, allocate memory for the new node and put data in it. Lastly, store the address of the first node (start) in the next field of the new node.
2. Delete - This function takes the start node (as pointer) and data to be deleted as arguments. Firstly, go to the node for which the node next to it has to be deleted,
   If that node points to NULL (i.e. pointer->next=NULL) then the element to be deleted is not present in the list.
   Else, now pointer points to a node and the node next to it has to be removed, declare a temporary node (temp) which points to the node which has to be

removed. Store the address of the node next to the temporary node in the next field of the node pointer (pointer->next = temp->next). Thus, by breaking the link we removed the node which is next to the pointer (which is also temp). Because we deleted the node, we no longer require the memory used for it, free() will deallocate the memory.

3. Find - This function takes the start node (as pointer) and data value of the node (key) to be found as arguments. A pointer start of type node is declared, which points to the head node of the list (node *start = pointer). First node is dummy node so, start with the second node. Iterate through the entire linked list and search for the key.

   Until next field of the pointer is equal to start, check if pointer->data = key. If it is then the key is found else, move to the next node and search (pointer = pointer -> next). If key is not found return 0, else return 1.

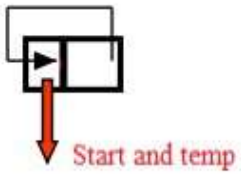
4. Print - function takes the start node (as start) and the next node (as pointer) as arguments. If pointer = start, then there is no element in the list. Else, print the data value of the node (pointer->data) and move to the next node by recursively calling the print function with pointer->next sent as an argument.

Performance:

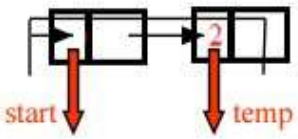
1. The advantage is that we no longer need both a head and tail variable to keep track of the list. Even if only a single variable is used, both the first and the last list elements can be found in constant time. Also, for implementing queues we will only need one pointer namely tail, to locate both head and tail.
2. The disadvantage is that the algorithms have become more complicated.

Example:

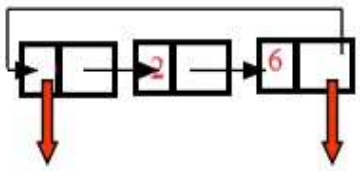
Initially



Insert(Start,2) - A new node with data 2 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.

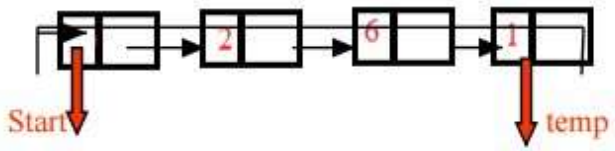


Insert(Start,6) - A new node with data 6 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.

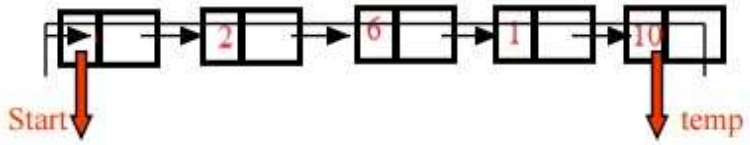


Start temp

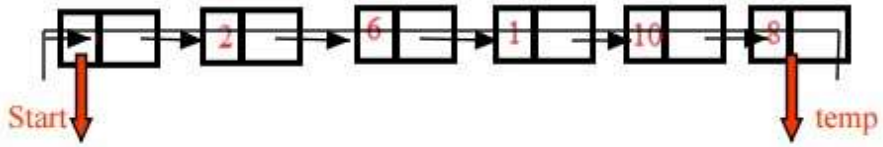
Insert(Start,1) – A new node with data 1 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.



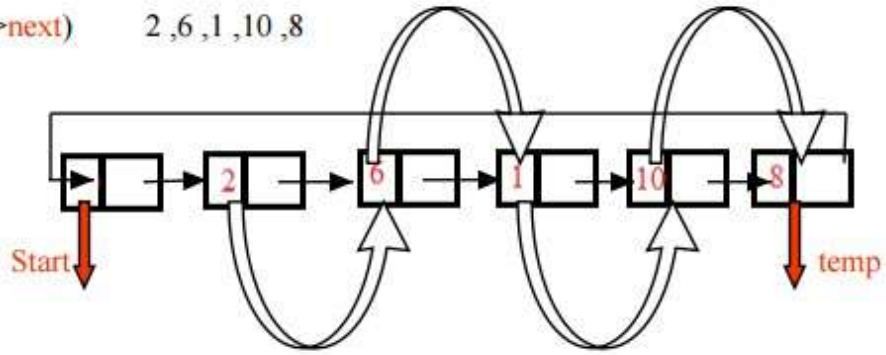
Insert(Start,10) – A new node with data 6 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.



Insert(Start,8) - A new node with data 6 is inserted and the next field is updated to store the address of start node. The next field of previous node is updated to store the address of new node.

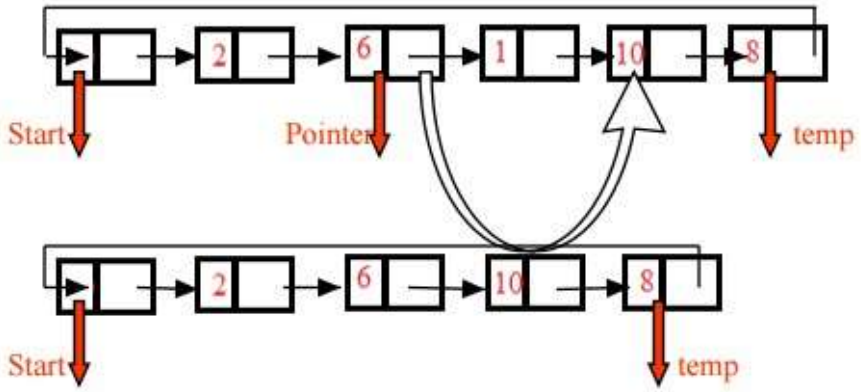


Print(start->next) 2 ,6 ,1 ,10 ,8

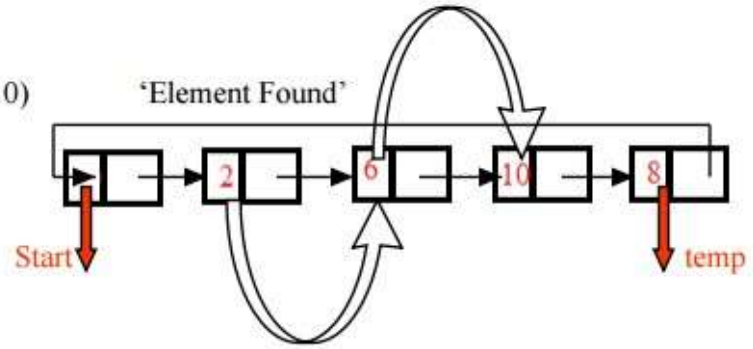


To print start from the first node of the list and move to the next with the help of the address stored in the next field.

Delete(start,1) - A node with data 1 is found and the next field is updated to store the NULL value. The next field of previous node is updated to store the address of node next to the deleted node.



Find(start,10) 'Element Found'



To find, start from the first node of the list and move to the next with the help of the address stored in the next field.

Insertion In Circular Linked List

There are three situation for inserting element in Circular linked list.

- Insertion at the front of Circular linked list.
- Insertion in the middle of the Circular linked list.
- Insertion at the end of the Circular linked list.

Insertion at the front of Circular linked list

Procedure for insertion a node at the beginning of list

Step1. Create the new node

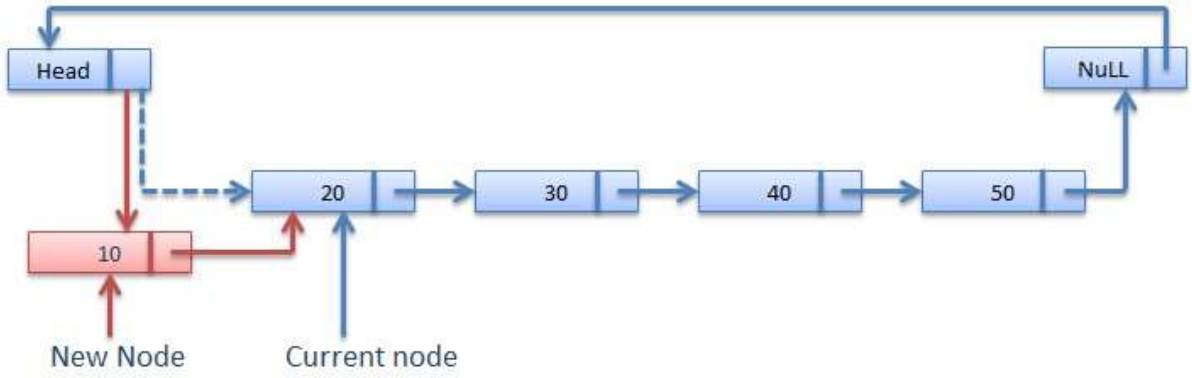
Step2. Set the new node's next to itself (circular!)

Step3. If the list is empty,return new node.

Step4. Set our new node's next to the front.

Step5. Set tail's next to our new node.

Step6. Return the end of the list.



Algorithm for Insertion at the front of Circular linked list

```
node* AddFront(node* tail, int num)
{
        node *temp = (node*)malloc(sizeof(node));
        temp->data = num;
        temp->next = temp;
        if (tail == NULL)
                return temp;
        temp->next = tail->next;
        tail->next = temp;
```

return tail;

}

Insertion in the middle of the Circular linked list

InsertAtlocDll(info,next,start,end,loc,size)

set nloc = loc-1 , n=1

create a new node and address in assigned to ptr.

check[overflow] if(ptr=NULL)

write:overflow and exit

set Info[ptr]=item;

if(start=NULL)

set next[ptr] = NULL

set start = ptr

else if(nloc<=size)

repeat steps a and b while(n != nloc)

a. loc = next[loc]

b. n = n+1

[end while]

next[ptr] = next[loc]

next[loc] = ptr

else

set last = start;

repeat step (a) while(next[last]!= NULL)

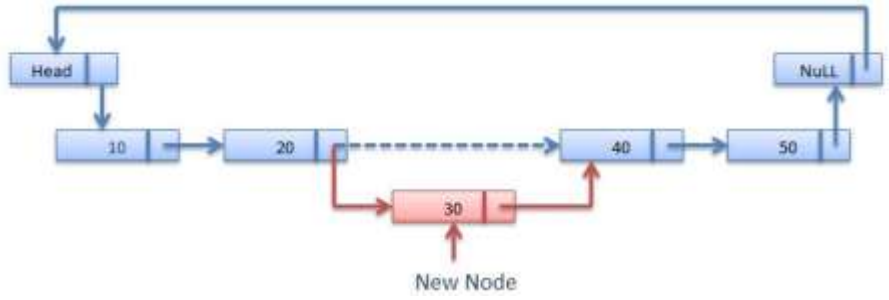
a. last=next[last]

[end while]

last->next = ptr ;

[end if]

Exit.



Insertion at the end of Circular linked list

Procedure for insertion a node at the end of list

Step1. Create the new node

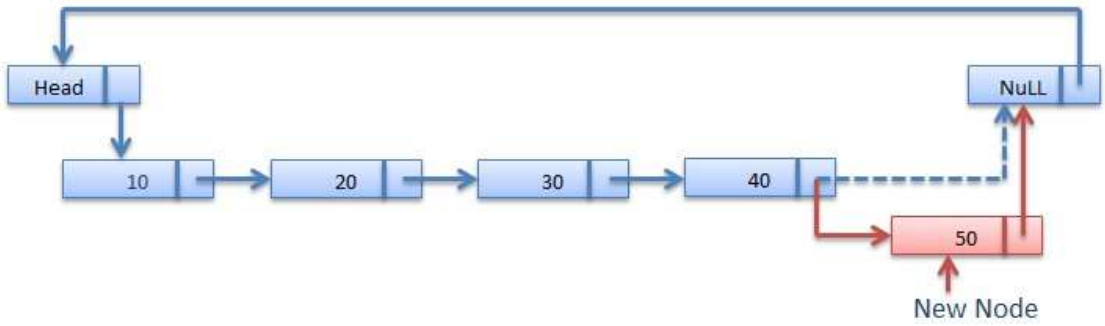
Step2. Set the new node's next to itself (circular!)

Step3. If the list is empty,return new node.

Step4. Set our new node's next to the front.

Step5. Set tail's next to our new node.

Step6. Return the end of the list.



Algorithm for Insertion at the End of Circular linked list

node AddEnd(node* tail, int num)*

{

```
        node *temp = (node*)malloc(sizeof(node));
        temp->data = num;
        temp->next = temp;
        if (tail == NULL)
                return temp;
        temp->next = tail->next;
        tail->next = temp;
        return temp;
}
```