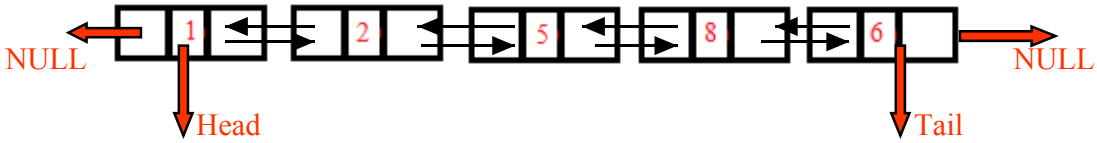
Doubly-Linked List

Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contains 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 Doubly-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 that 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.

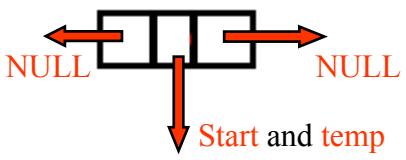
3. **Find** - This function takes the start node (as pointer) and data value of the node (key) to be found as arguments. First node is dummy node so, start with the second node. Iterate through the entire linked list and search for the key.Until next field of the pointer is equal to NULL, check if pointer->data = 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:

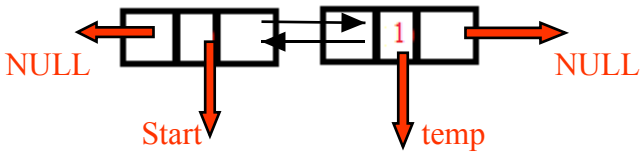
1. The advantage of a singly linked list is that we don't need to keep track of the previous node for traversal or no need of traversing the whole list for finding the previous node.
2. The disadvantage is that more pointers need to be handled and more links need to update.

Example: Initially

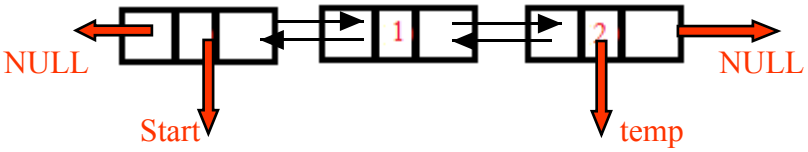
```
typedef struct Node {
      int data;
      struct Node *next;
      struct Node *prev;
} node;

node *start, *temp;
```



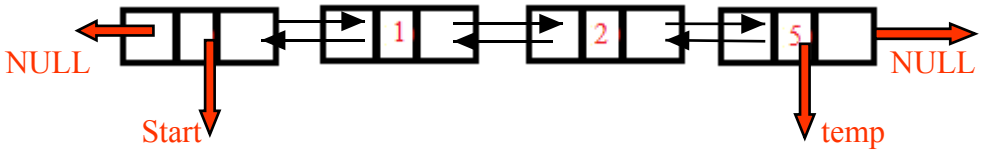
Insert(Start,1) - A new node with data 1 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.



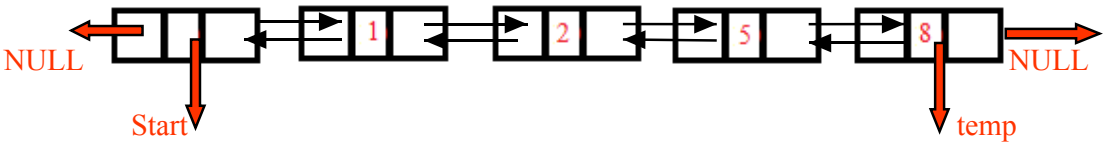
Insert(Start,2) - A new node with data 2 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.



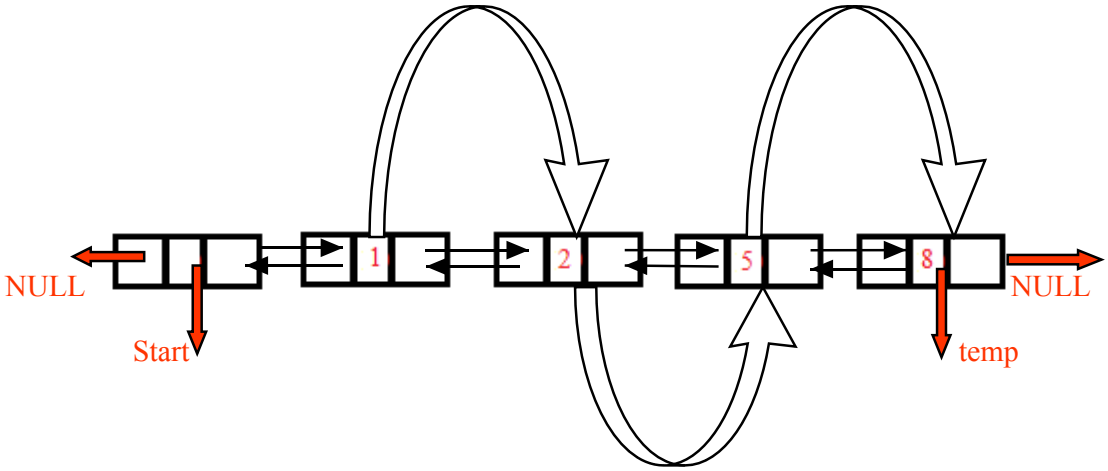
Insert(Start,5) - A new node with data 5 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.



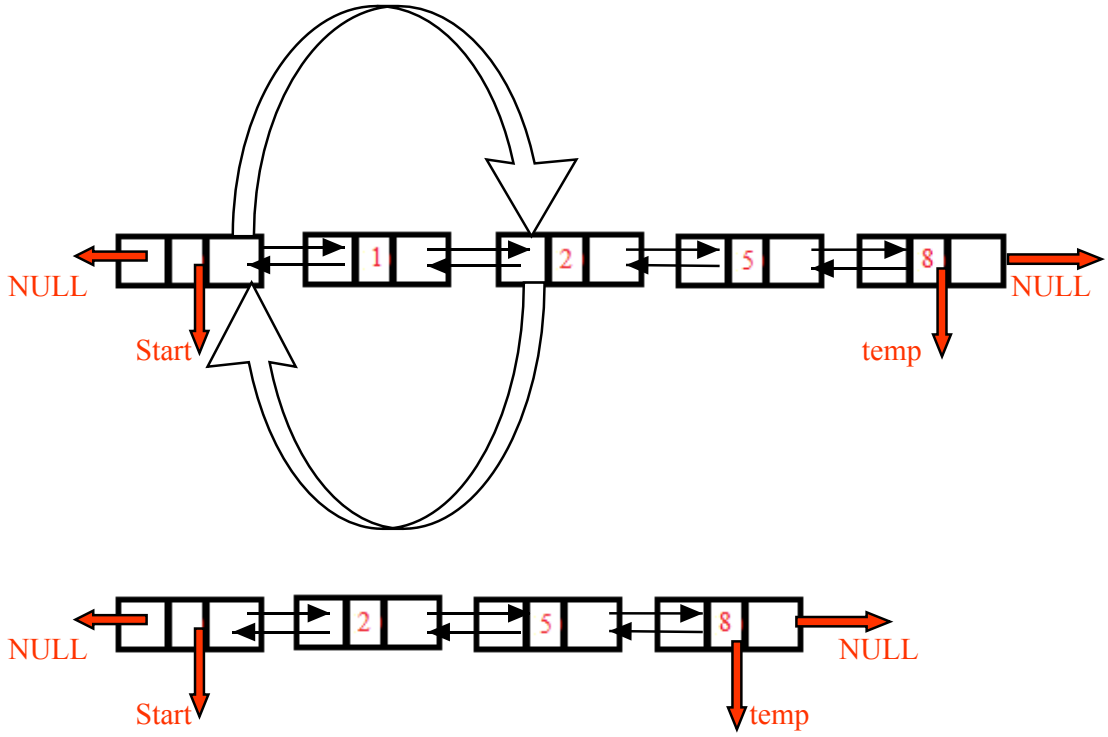
Insert(Start,8) - A new node with data 8 is inserted, the next field is updated to NULL and the previous field is updated to store the address of the previous node. The next field of previous node is updated to store the address of new node.



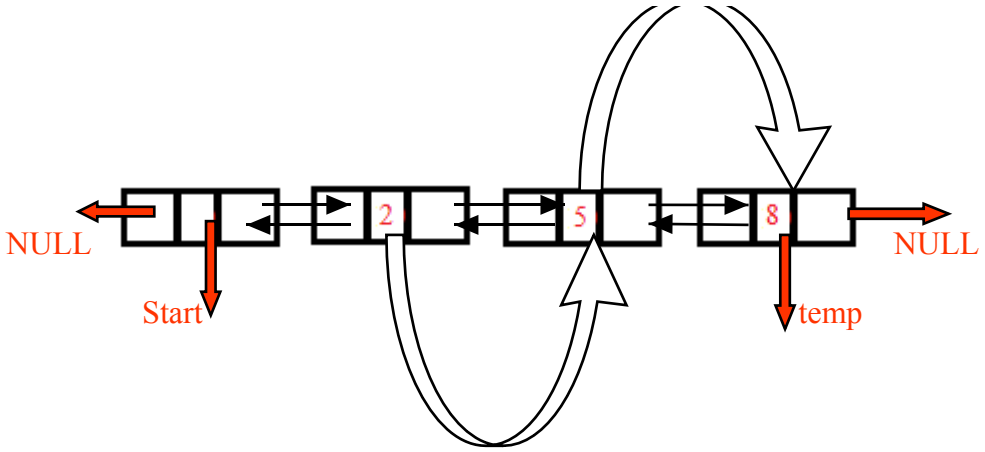
Print(start)- 1, 2, 5, 8 To print start from the first node of the list and move to the next with the help of the address stored in the next field.



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' stored in the next field.



Double Linked List - C Program source code

```
#include<stdio.h>
#include<stdlib.h>

typedef struct Node {
	int data;
	struct Node *next;
	struct Node *prev;
} node;

void insert(node *pointer, int data) {
/* Iterate through the list till we encounter the last node.*/
	while (pointer->next != NULL) {
		pointer = pointer->next;
	}
	/* Allocate memory for the new node and put data in it.*/
	pointer->next = (node *) malloc(sizeof(node));
	(pointer->next)->prev = pointer;
	pointer = pointer->next;
	pointer->data = data;
	pointer->next = NULL;
}

int find(node *pointer, int key) {
	pointer = pointer->next; //First node is dummy node.
/*Iterate through the linked list and search for the key.*/
	while (pointer != NULL) {
		if (pointer->data == key) {
			return 1; //key is found.
		}
		pointer = pointer->next; //Search in the next node.
	}
	return 0;	/*Key is not found */
}
```

```c
void delete_node(node *pointer, int data) {
/* Go to the node for which the node next to it has to be
deleted */
  while(pointer->next !=NULL && (pointer->next)->data!=data)
      {
      pointer = pointer->next;
      }
  if (pointer->next == NULL) {
      printf("Element %d is not present in the list\n", data);
      return;
      }
/* Now pointer points to a node and the node next to it has to
be removed */
      node *temp;
      temp = pointer->next;
/*temp points to the node which has to be removed*/
      pointer->next = temp->next;
      temp->prev = pointer;
/*We removed the node which is next to the pointer (which is
also temp) */
      free(temp);
/* Beacuse we deleted the node, we no longer require the memory
used for it. free() will deallocate the memory. */
      return;
}

void print(node *pointer) {
      if (pointer == NULL) {
            return;
      }
      printf("%d ", pointer->data);
      print(pointer->next);
}

int main() {
/* start always points to the first node of the linked list.
temp is used to point to the last node of the linked list.*/
      node *start, *temp;
      start = (node *) malloc(sizeof(node));
      temp = start;
      temp->next = NULL;
      temp->prev = NULL;

/* Here in this code, 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.*/
```

```
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Print\n");
        printf("4. Find\n");
        while (1) {
                int query;
                scanf("%d", &query);
                if (query == 1) {
                        int data;
                        scanf("%d", &data);
                        insert(start, data);
                } else if (query == 2) {
                        int data;
                        scanf("%d", &data);
                        delete_node(start, data);
                } else if (query == 3) {
                        printf("The list is ");
                        print(start->next);
                        printf("\n");
                } else if (query == 4) {
                        int data;
                        scanf("%d", &data);
                        int status = find(start, data);
                        if (status) {
                                printf("Element Found\n");
                        } else {
                                printf("Element Not Found\n");

                        }
                }
        }
}
```