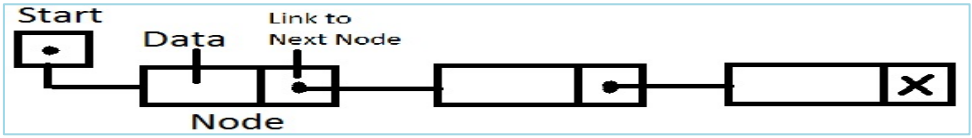
Introduction To Linked List

If the memory is allocated for the variable during the compilation (i.e. before execution) of a program, then it is fixed and cannot be changed. For example an array A[100] is declared with 100 elements, then the allocated memory is fixed and cannot decrease or increase the SIZE of the array if required. So we have to adopt an alternative strategy to allocate memory only when it is required. There is a special data structure called linked list that provides a more flexible dynamic storage system and it does not require the use of array.

Linked List

A linked list is a linear collection of specially designed data elements, called nodes, linked to one another by means of pointers. Each node is divided into two parts: the first part contains the information of the element, and the second part contains the address of the next node in the linked list. Address part of the node is also called linked or net field.



Linked List

Linked List representation in memory

Above figure shows a schematic diagram of a linked list with 3 nodes. Each node is pictured with two parts. The left part of each node contains the data items and the right part represents the address of the next node; the next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node. That is NULL pointer indicates the end of the linked list. START pointer will hold the address of the 1st node in the list START = NULL if there is no list (i.e.; NULL list or empty list).

The node of a linear linked list can be can be created using self-referential structure with the following declaration:

```
struct Node
{
  int data;           /* member variable to store the data */
  struct Node *next; /* Node pointer to store the address of a node*/
};

/* create a node in the memory */
struct Node *start = (struct node *)malloc(sizeof(struct node));
```

Advantages and Disadvantages of Linked List

Linked list have many advantages and some of them are:

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.
2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (or removed) when it is not needed.
3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many complex applications can be easily carried out with linked list.

Linked list has following disadvantages

1. More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming.

Operations on Linked List

The primitive operations performed on the linked list are as follows

1. **Creation**: Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.
2. **Insertion**: Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.
    a. At the beginning of the linked list
    b. At the end of the linked list
    c. At any specified position in between in a linked list
3. **Deletion**: Deletion operation is used to delete an item (or node) from the linked list. A node may be deleted from the
    a. Beginning of a linked list
    b. End of a linked list
    c. Specified location of the linked list
4. **Traversing**: Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, nodes only. But in doubly linked list forward and backward traversing is possible.
5. **Searching**: Traversing is the process of finding a data in the list.
6. **Concatenation**: Concatenation is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1)th node in A. After concatenation A will contain (n+m) nodes

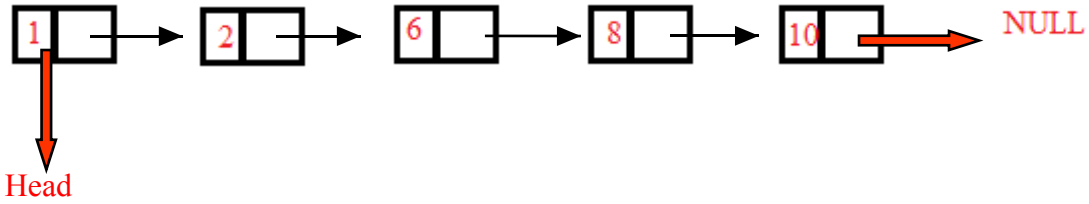
Type of Linked List

Basically we can divide the linked list into the following three types in the order in which they (or node) are arranged.

1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly Linked List

Singly linked list is the most basic 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 contains the NULL reference. It has a dynamic size, which can be determined only at run time.



Basic operations of a singly-linked list are:

Insert – Inserts a new element at the end of the list.
Delete – Deletes any node from the list.
Find – Finds any node in the list.
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 = 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

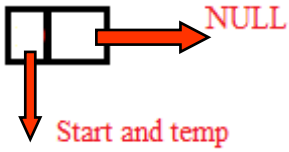
1. Insert – This function takes the start node and data to be inserted as arguments. New node is inserted at the end so, iterate through the list till we encounter the last node. Then, allocate memory for the new node and put data in it. Lastly, store the address in the next field of the new node as 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). 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:

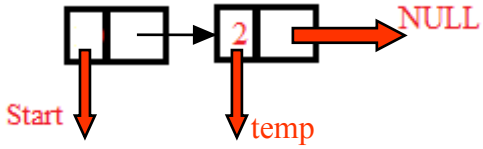
1. The advantage of a singly linked list is its ability to expand to accept virtually unlimited number of nodes in a fragmented memory environment.
2. The disadvantage is its speed. Operations in a singly-linked list are slow as it uses sequential search to locate a node.

Example:

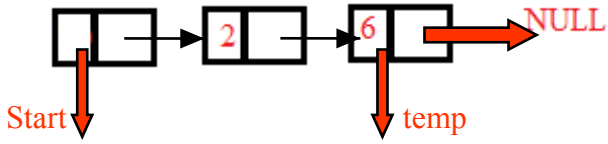
Initially



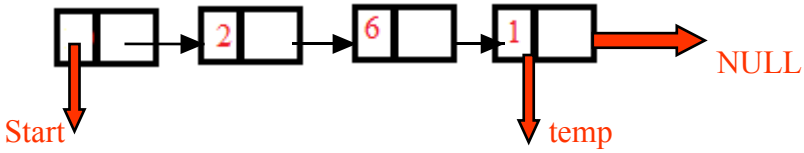
Insert(Start,2) - A new node with data 2 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



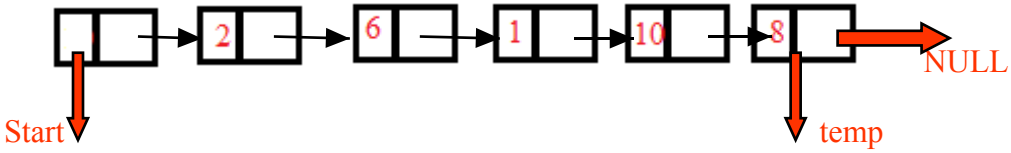
Insert(Start,6) - A new node with data 6 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



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

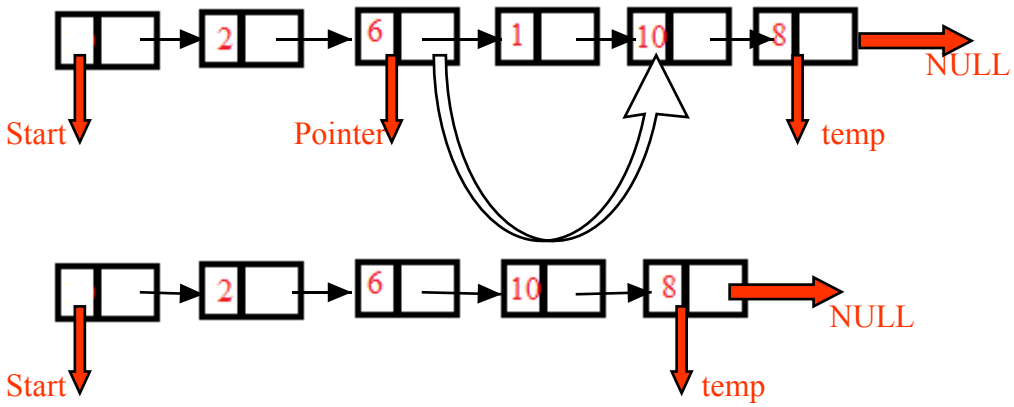


Insert(Start,10) - A new node with data 10 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.

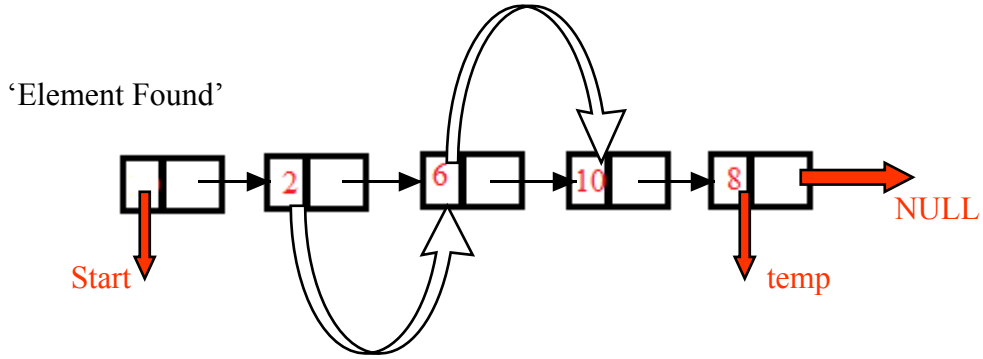


Print(start->next) - To print start from the first node of the list and move to the next with the help of the address stored in the next field. 2 ,6 ,1 ,10 ,8

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



Single Linked List - C Program source code

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

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

/* 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
in the next field of the new node as NULL. */

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 = pointer->next;
        pointer->data = data;
        pointer->next = NULL;
}
```

```
/* 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. */

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

/* 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.                */

void delete(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;
/*We removed the node which is next to the pointer(which is also
temp)*/
        free(temp);
/* Because we deleted the node, we no longer require the     memory
used for it. free() will deallocate the memory. */
        return;
}
```

```
/* 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. */

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 list.*/
        node *start,*temp;
        start = (node *)malloc(sizeof(node));
        temp = start;
        temp -> next = 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, data;
               scanf("%d",&query);

               switch(query){
                case 1:
                        scanf("%d",&data);
                        insert(start,data);
                       break;
                case 2:
                        scanf("%d",&data);
                        delete(start,data);
                       break;
                case 3:
                        printf("The list is ");
                        print(start->next);
                        printf("\n");
                       break;
                case 4:
                        scanf("%d",&data);
                        int status = find(start,data);
                        if(status)
                              printf("Element Found\n");
                        else
                              printf("Element Not Found\n");
                }
        }
}
```