Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.

1. The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2. Inserting a new element in an array of elements is expensive; because room has to be created for the new elements and to create room existing elements have to be shifted.

For example, in a system if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

**Advantages over arrays**

1. Dynamic size
2. Ease of insertion/deletion

**Drawbacks:**

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we can’t do binary search with linked lists efficiently with its default implementation.
2. Extra memory space for a pointer is required with each element of the list.
3. Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.

Each node in a list consists of at least two parts:

1. Data
2. Pointer (or Reference) to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with integer data.

In C++ or Java, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

struct Node {

    int data;

    struct Node\* next;

};

**First Simple Linked List in C**

Let us create a simple linked list with 3 nodes.

// A simple C program to introduce

// a linked list

#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* next;

};

// Program to create a simple linked

// list with 3 nodes

int main()

{

    struct Node\* head = NULL;

    struct Node\* second = NULL;

    struct Node\* third = NULL;

    // allocate 3 nodes in the heap

    head = (struct Node\*)malloc(sizeof(struct Node));

    second = (struct Node\*)malloc(sizeof(struct Node));

    third = (struct Node\*)malloc(sizeof(struct Node));

    /\* Three blocks have been allocated dynamically.

     We have pointers to these three blocks as head,

     second and third

       head           second           third

        |                |               |

        |                |               |

    +---+-----+     +----+----+     +----+----+

    | #  | #  |     | #  | #  |     |  # |  # |

    +---+-----+     +----+----+     +----+----+

   # represents any random value.

   Data is random because we haven’t assigned

   anything yet  \*/

    head->data = 1; // assign data in first node

    head->next = second; // Link first node with

    // the second node

    /\* data has been assigned to the data part of the first

     block (block pointed by the head). And next

     pointer of first block points to second.

     So they both are linked.

       head          second         third

        |              |              |

        |              |              |

    +---+---+     +----+----+     +-----+----+

    | 1  | o----->| #  | #  |     |  #  | #  |

    +---+---+     +----+----+     +-----+----+

  \*/

    // assign data to second node

    second->data = 2;

    // Link second node with the third node

    second->next = third;

    /\* data has been assigned to the data part of the second

     block (block pointed by second). And next

     pointer of the second block points to the third

     block. So all three blocks are linked.

       head         second         third

        |             |             |

        |             |             |

    +---+---+     +---+---+     +----+----+

    | 1  | o----->| 2 | o-----> |  # |  # |

    +---+---+     +---+---+     +----+----+      \*/

    third->data = 3; // assign data to third node

    third->next = NULL;

    /\* data has been assigned to data part of third

    block (block pointed by third). And next pointer

    of the third block is made NULL to indicate

    that the linked list is terminated here.

     We have the linked list ready.

           head

             |

             |

        +---+---+     +---+---+       +----+------+

        | 1  | o----->|  2  | o-----> |  3 | NULL |

        +---+---+     +---+---+       +----+------+

    Note that only head is sufficient to represent

    the whole list.  We can traverse the complete

    list by following next pointers.    \*/

    return 0;

}

**Linked List Traversal**

In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general-purpose function printList() that prints any given list.

// A simple C++ program for traversal of a linked list

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

    int data;

    Node\* next;

};

// This function prints contents of linked list

// starting from the given node

void printList(Node\* n)

{

    while (n != NULL) {

        cout << n->data << " ";

        n = n->next;

    }

}

// Driver code

int main()

{

    Node\* head = NULL;

    Node\* second = NULL;

    Node\* third = NULL;

    // allocate 3 nodes in the heap

    head = new Node();

    second = new Node();

    third = new Node();

    head->data = 1; // assign data in first node

    head->next = second; // Link first node with second

    second->data = 2; // assign data to second node

    second->next = third;

    third->data = 3; // assign data to third node

    third->next = NULL;

    printList(head);

    return 0;

}

**Output:**

1 2 3

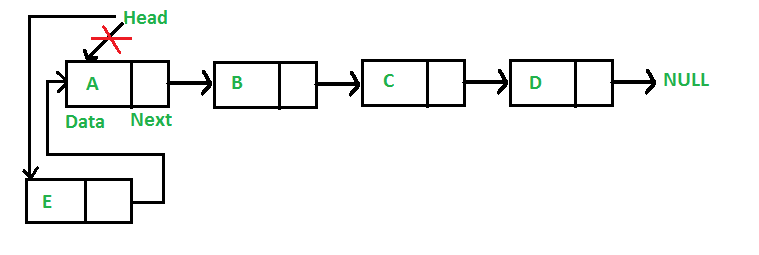
**Inserting a Node**

A node can be added in three ways

1. At the front of the linked list
2. After a given node.
3. At the end of the linked list.

**Add a node at the front: (A 4 steps process)**

The new node is always added before the head of the given Linked List. Any newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node.



Following are the 4 steps to add at the front.

/\* Given a reference (pointer to pointer)

to the head of a list and an int,

inserts a new node on the front of the list. \*/

void push(Node\*\* head\_ref, int new\_data)

{

    /\* 1. allocate node \*/

    Node\* new\_node = new Node();

    /\* 2. put in the data \*/

    new\_node->data = new\_data;

    /\* 3. Make next of new node as head \*/

    new\_node->next = (\*head\_ref);

    /\* 4. move the head to point to the new node \*/

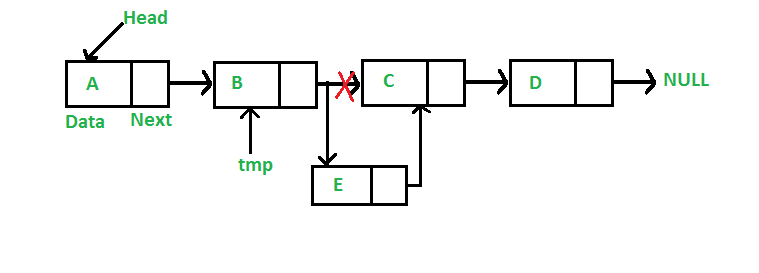
    (\*head\_ref) = new\_node;

}

Time complexity of push() is O(1) as it does constant amount of work.

**Add a node after a given node. (5 steps process)**

We are given pointer to a node, and the new node is inserted after the given node.



/\* Given a node prev\_node, insert a new node after the given

   prev\_node \*/

void insertAfter(struct Node\* prev\_node, int new\_data)

{

    /\*1. check if the given prev\_node is NULL \*/

    if (prev\_node == NULL)

    {

       printf("the given previous node cannot be NULL");

       return;

    }

    /\* 2. allocate new node \*/

    struct Node\* new\_node =(struct Node\*) malloc(sizeof(struct Node));

    /\* 3. put in the data  \*/

    new\_node->data  = new\_data;

    /\* 4. Make next of new node as next of prev\_node \*/

    new\_node->next = prev\_node->next;

    /\* 5. move the next of prev\_node as new\_node \*/

    prev\_node->next = new\_node;

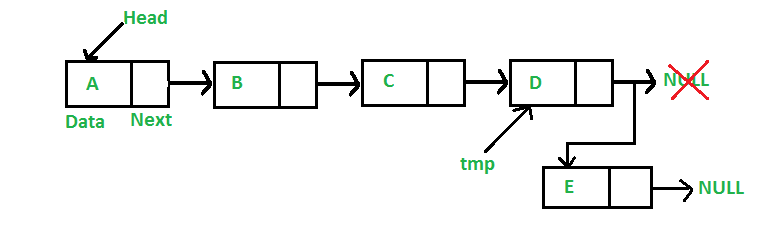
}

Time complexity of insertAfter() is O(1) as it does constant amount of work.

**Add a node at the end: (6 steps process)**

The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.



Following are the 6 steps to add node at the end.

/\* Given a reference (pointer to pointer) to the head

   of a list and an int, appends a new node at the end  \*/

void append(struct Node\*\* head\_ref, int new\_data)

{

    /\* 1. allocate node \*/

    struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

    struct Node \*last = \*head\_ref;  /\* used in step 5\*/

    /\* 2. put in the data  \*/

    new\_node->data  = new\_data;

    /\* 3. This new node is going to be the last node, so make next

          of it as NULL\*/

    new\_node->next = NULL;

    /\* 4. If the Linked List is empty, then make the new node as head \*/

    if (\*head\_ref == NULL)

    {

       \*head\_ref = new\_node;

       return;

    }

    /\* 5. Else traverse till the last node \*/

    while (last->next != NULL)

        last = last->next;

    /\* 6. Change the next of last node \*/

    last->next = new\_node;

    return;

}

Time complexity of append is O(n) where n is the number of nodes in linked list. Since there is a loop from head to end, the function does O(n) work.

This method can also be optimized to work in O(1) by keeping an extra pointer to tail of linked list.

Following is a complete program that used all of the above methods to create a linked list.

// A complete working C program to demonstrate all insertion methods

// on Linked List

#include <stdio.h>

#include <stdlib.h>

// A linked list node

struct Node

{

  int data;

  struct Node \*next;

};

/\* Given a reference (pointer to pointer) to the head of a list and

   an int, inserts a new node on the front of the list. \*/

void push(struct Node\*\* head\_ref, int new\_data)

{

    /\* 1. allocate node \*/

    struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

    /\* 2. put in the data  \*/

    new\_node->data  = new\_data;

    /\* 3. Make next of new node as head \*/

    new\_node->next = (\*head\_ref);

    /\* 4. move the head to point to the new node \*/

    (\*head\_ref)    = new\_node;

}

/\* Given a node prev\_node, insert a new node after the given

   prev\_node \*/

void insertAfter(struct Node\* prev\_node, int new\_data)

{

    /\*1. check if the given prev\_node is NULL \*/

    if (prev\_node == NULL)

    {

      printf("the given previous node cannot be NULL");

      return;

    }

    /\* 2. allocate new node \*/

    struct Node\* new\_node =(struct Node\*) malloc(sizeof(struct Node));

    /\* 3. put in the data  \*/

    new\_node->data  = new\_data;

    /\* 4. Make next of new node as next of prev\_node \*/

    new\_node->next = prev\_node->next;

    /\* 5. move the next of prev\_node as new\_node \*/

    prev\_node->next = new\_node;

}

/\* Given a reference (pointer to pointer) to the head

   of a list and an int, appends a new node at the end  \*/

void append(struct Node\*\* head\_ref, int new\_data)

{

    /\* 1. allocate node \*/

    struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

    struct Node \*last = \*head\_ref;  /\* used in step 5\*/

    /\* 2. put in the data  \*/

    new\_node->data  = new\_data;

    /\* 3. This new node is going to be the last node, so make next of

          it as NULL\*/

    new\_node->next = NULL;

    /\* 4. If the Linked List is empty, then make the new node as head \*/

    if (\*head\_ref == NULL)

    {

       \*head\_ref = new\_node;

       return;

    }

    /\* 5. Else traverse till the last node \*/

    while (last->next != NULL)

        last = last->next;

    /\* 6. Change the next of last node \*/

    last->next = new\_node;

    return;

}

// This function prints contents of linked list starting from head

void printList(struct Node \*node)

{

  while (node != NULL)

  {

     printf(" %d ", node->data);

     node = node->next;

  }

}

/\* Driver program to test above functions\*/

int main()

{

  /\* Start with the empty list \*/

  struct Node\* head = NULL;

  // Insert 6.  So linked list becomes 6->NULL

  append(&head, 6);

  // Insert 7 at the beginning. So linked list becomes 7->6->NULL

  push(&head, 7);

  // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL

  push(&head, 1);

  // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL

  append(&head, 4);

  // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL

  insertAfter(head->next, 8);

  printf("\n Created Linked list is: ");

  printList(head);

  return 0;

}

**Output:**

Created Linked list is: 1 7 8 6 4

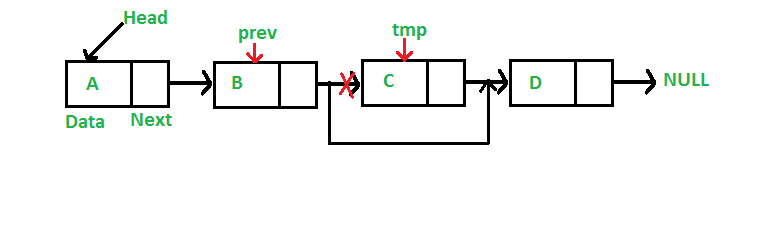
**Deleting a node**

Let us formulate the problem statement to understand the deletion process.

**Given a ‘key’, delete the first occurrence of this key in linked list.**

To delete a node from linked list, we need to do following steps:

1. Find previous node of the node to be deleted.
2. Change the next of previous node.
3. Free memory for the node to be deleted.



Since every node of linked list is dynamically allocated using malloc() in C, we need to call free() for freeing memory allocated for the node to be deleted.

// A complete working C program to demonstrate deletion in singly

// linked list

#include <stdio.h>

#include <stdlib.h>

// A linked list node

struct Node

{

    int data;

    struct Node \*next;

};

/\* Given a reference (pointer to pointer) to the head of a list

   and an int, inserts a new node on the front of the list. \*/

void push(struct Node\*\* head\_ref, int new\_data)

{

    struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

    new\_node->data  = new\_data;

    new\_node->next = (\*head\_ref);

    (\*head\_ref)    = new\_node;

}

/\* Given a reference (pointer to pointer) to the head of a list

   and a key, deletes the first occurrence of key in linked list \*/

void deleteNode(struct Node \*\*head\_ref, int key)

{

    // Store head node

    struct Node\* temp = \*head\_ref, \*prev;

    // If head node itself holds the key to be deleted

    if (temp != NULL && temp->data == key)

    {

        \*head\_ref = temp->next;   // Changed head

        free(temp);               // free old head

        return;

    }

    // Search for the key to be deleted, keep track of the

    // previous node as we need to change 'prev->next'

    while (temp != NULL && temp->data != key)

    {

        prev = temp;

        temp = temp->next;

    }

    // If key was not present in linked list

    if (temp == NULL) return;

    // Unlink the node from linked list

    prev->next = temp->next;

    free(temp);  // Free memory

}

// This function prints contents of linked list starting from

// the given node

void printList(struct Node \*node)

{

    while (node != NULL)

    {

        printf(" %d ", node->data);

        node = node->next;

    }

}

/\* Drier program to test above functions\*/

int main()

{

    /\* Start with the empty list \*/

    struct Node\* head = NULL;

    push(&head, 7);

    push(&head, 1);

    push(&head, 3);

    push(&head, 2);

    puts("Created Linked List: ");

    printList(head);

    deleteNode(&head, 1);

    puts("\nLinked List after Deletion of 1: ");

    printList(head);

    return 0;

}

**Output:**

Created Linked List:

2 3 1 7

Linked List after Deletion of 1:

2 3 7

**Delete a Linked List node at a given position**

Given a singly linked list and a position, delete a linked list node at the given position.

**Example:**

Input: position = 1, Linked List = 8->2->3->1->7

Output: Linked List = 8->3->1->7

Input: position = 0, Linked List = 8->2->3->1->7

Output: Linked List = 2->3->1->7

If node to be deleted is root, simply delete it. To delete a middle node, we must have pointer to the node previous to the node to be deleted. So if positions is not zero, we run a loop position-1 times and get pointer to the previous node.

Below is the implementation of above idea.

// A complete working C program to delete a node in a linked list

// at a given position

#include <stdio.h>

#include <stdlib.h>

// A linked list node

struct Node

{

    int data;

    struct Node \*next;

};

/\* Given a reference (pointer to pointer) to the head of a list

   and an int, inserts a new node on the front of the list. \*/

void push(struct Node\*\* head\_ref, int new\_data)

{

    struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

    new\_node->data  = new\_data;

    new\_node->next = (\*head\_ref);

    (\*head\_ref)    = new\_node;

}

/\* Given a reference (pointer to pointer) to the head of a list

   and a position, deletes the node at the given position \*/

void deleteNode(struct Node \*\*head\_ref, int position)

{

   // If linked list is empty

   if (\*head\_ref == NULL)

      return;

   // Store head node

   struct Node\* temp = \*head\_ref;

    // If head needs to be removed

    if (position == 0)

    {

        \*head\_ref = temp->next;   // Change head

        free(temp);               // free old head

        return;

    }

    // Find previous node of the node to be deleted

    for (int i=0; temp!=NULL && i<position-1; i++)

         temp = temp->next;

    // If position is more than number of ndoes

    if (temp == NULL || temp->next == NULL)

         return;

    // Node temp->next is the node to be deleted

    // Store pointer to the next of node to be deleted

    struct Node \*next = temp->next->next;

    // Unlink the node from linked list

    free(temp->next);  // Free memory

    temp->next = next;  // Unlink the deleted node from list

}

// This function prints contents of linked list starting from

// the given node

void printList(struct Node \*node)

{

    while (node != NULL)

    {

        printf(" %d ", node->data);

        node = node->next;

    }

}

/\* Drier program to test above functions\*/

int main()

{

    /\* Start with the empty list \*/

    struct Node\* head = NULL;

    push(&head, 7);

    push(&head, 1);

    push(&head, 3);

    push(&head, 2);

    push(&head, 8);

    puts("Created Linked List: ");

    printList(head);

    deleteNode(&head, 4);

    puts("\nLinked List after Deletion at position 4: ");

    printList(head);

    return 0;

}

**Output:**

Created Linked List:

8 2 3 1 7

Linked List after Deletion at position 4:

8 2 3 1