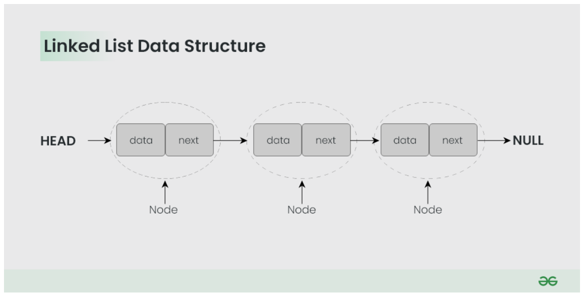
What is a Linked List?

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:



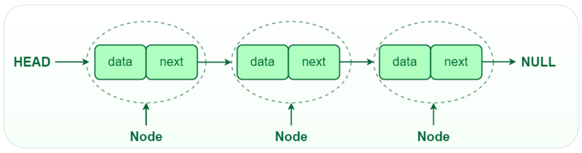
In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Why is a linked list data structure needed?
Here are a few advantages of a linked list that is listed below, it will help you understand why it is necessary to know.

- Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
- Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Types of linked lists:
There are mainly three types of linked lists:
- Single-linked list
- Double linked list
- Circular linked list

1. Singly-linked list
Traversal of items can be done in the forward direction only due to the linking of every node to its next node.

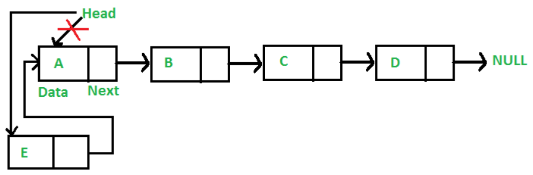


```
// A Single linked list node
class Node {
public:
    int data;
    Node* next;
};
```

How to Insert a Node at the Front/Beginning of Linked List

To insert a node at the start/beginning/front of a Linked List, we need to:

- Make the first node of Linked List linked to the new node
- Remove the head from the original first node of Linked List
- Make the new node as the Head of the Linked List.



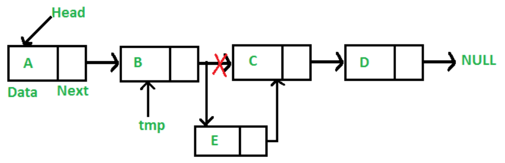
```
void push(Node** head_ref, int new_data)
{
    // 1. allocate node
    Node* new_node = new Node();
    // 2. put in the data
    new_node->data = new_data;
    // 3. Make next of new node as head
    new_node->next = (*head_ref);
    // 4. Move the head to point to
    // the new node
    (*head_ref) = new_node;
}
```

How to Insert a Node after a Given Node in Linked List

To insert a node after a given node in a Linked List, we need to:

- Check if the given node exists or not.
    - If it do not exists,
        - terminate the process.
    - If the given node exists,
        - Make the element to be inserted as a new node
        - Change the next pointer of given node to the new node

- Now shift the original next pointer of given node to the next pointer of new node

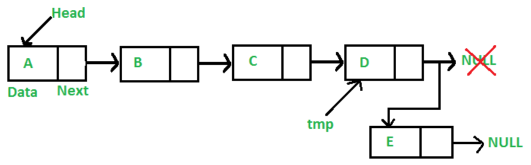


```
void insertAfter(Node* prev_node, int new_data)
{
    // 1. Check if the given prev_node is NULL
    if (prev_node == NULL) {
        cout << "The given previous node cannot be NULL";
        return;
    }
    // 2. Allocate new node
    Node* new_node = new Node();
    // 3. Put in the data
    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;
}
```

How to Insert a Node at the End of Linked List

To insert a node at the end of a Linked List, we need to:

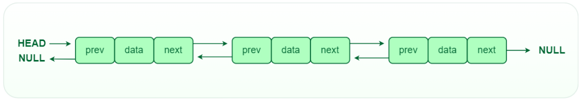
- Go to the last node of the Linked List
- Change the next pointer of last node from NULL to the new node
- Make the next pointer of new node as NULL to show the end of Linked List



```
void append(Node** head_ref, int new_data)
{
    // 1. allocate node
    Node* new_node = new Node();
    // Used in step 5
    Node* last = *head_ref;
    // 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;
}
```

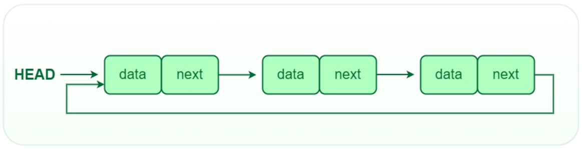
The -> symbol is an operator to select an element from a data structure pointed to by a pointer. So suppose you have a pointer defined as mystruct *p and it points to a mystruct instantiation. Suppose also that mystruct declares a variable i of, say, type int .

2. Doubly linked list
Traversal of items can be done in both forward and backward directions as every node contains an additional prev pointer that points to the previous node.



```
/* Node of a doubly linked list */
class Node {
public:
    int data;
    Node* next; // Pointer to next node in DLL
    Node* prev; // Pointer to previous node in DLL
};
```

3. Circular linked lists
A circular linked list is a type of linked list in which the first and the last nodes are also connected to each other to form a circle, there is no NULL at the end.


HEAD
data
next
data
next
data
next

Deletion in Linked List
Delete from a Linked List:-
You can delete an element in a list from:
1) Delete from Beginning:
Point head to the next node i.e. second node
```
    temp = head
    head = head->next
```
Make sure to free unused memory
```
    free(temp); or delete temp;
```

2) Delete from End:
Point head to the previous element i.e. last second element
```
    Change next pointer to null
    struct node *end = head;
    struct node *prev = NULL;
    while(end->next)
    {
        prev = end;
        end = end->next;
    }
    prev->next = NULL;
```

Make sure to free unused memory
```
    free(end); or delete end;
```

3) Delete from Middle:
Keeps track of pointer before node to delete and pointer to node to delete
```
    temp = head;
    prev = head;
    for(int i = 0; i < position; i++)
    {
        if(i == 0 && position == 1)
            head = head->next;
            free(temp)
        else
        {
            if (i == position - 1 && temp)
            {
                prev->next = temp->next;
                free(temp);
            }
            else
            {
                prev = temp;
                if(prev == NULL) // position was greater than number of nodes in the list
                    break;
                temp = temp->next;
            }
        }
    }
```

Iterative Method to delete an element from the linked list:
To delete a node from the linked list, we need to do the following steps:

- Find the previous node of the node to be deleted.
- Change the next of the previous node.
- Free memory for the node to be deleted.

Linked List Operations: Traverse, Insert and Delete
Traversal - access each element of the linked list
Insertion - adds a new element to the linked list
Deletion - removes the existing elements
Search - find a node in the linked list
Sort - sort the nodes of the linked list

Things to Remember about Linked List

- head points to the first node of the linked list
- next pointer of the last node is NULL, so if the next current node is NULL, we have reached the end of the linked list.

In all of the examples, we will assume that the linked list has three nodes 1 --->2 --->3 with node structure as below:

```
struct node {
  int data;
  struct node *next;
};
```

Traverse a Linked List

1. Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.
2. When temp is NULL, we know that we have reached the end of the linked list so we get out of the while loop.

```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL) {
  printf("%d --->",temp->data);
  temp = temp->next;
}
```

The output of this program will be:

```
List elements are -
1 --->2 --->3 --->
```

Insert Elements to a Linked List

1. Insert at the beginning

1. Allocate memory for new node
2. Store data
3. Change next of new node to point to head
4. Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

2. Insert at the End

1. Allocate memory for new node
2. Store data
3. Traverse to last node
4. Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;

struct node *temp = head;
while(temp->next != NULL){
  temp = temp->next;
}

temp->next = newNode;
```

3. Insert at the Middle

1. Allocate memory and store data for new node
2. Traverse to node just before the required position of new node
3. Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
```

```
struct node *temp = head;

for(int i=2; i < position; i++) {
  if(temp->next != NULL) {
    temp = temp->next;
  }
}
newNode->next = temp->next;
temp->next = newNode;
```

Delete from a Linked List
You can delete either from the beginning, end or from a particular position.

1. Delete from beginning

1. Point head to the second node

```
head = head->next;
```

2. Delete from end

1. Traverse to second last element
2. Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

3. Delete from middle

1. Traverse to element before the element to be deleted
2. Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
  if(temp->next!=NULL) {
    temp = temp->next;
  }
}

temp->next = temp->next->next;
```

Search an Element on a Linked List
You can search an element on a linked list using a loop using the following steps. We are finding item on a linked list.

1. Make head as the current node.

2. Run a loop until the current node is NULL because the last element points to NULL.
3. In each iteration, check if the key of the node is equal to item. If it the key matches the item, return true otherwise return false.

```
// Search a node
bool searchNode(struct Node** head_ref, int key) {
  struct Node* current = *head_ref;

  while (current != NULL) {
    if (current->data == key) return true;
      current = current->next;
  }
  return false;
}
```

Sort Elements of a Linked List
We will use a simple sorting algorithm, Bubble Sort, to sort the elements of a linked list in ascending order below.

1. Make the head as the current node and create another node index for later use.
2. If the head is null, return.
3. Else, run a loop till the last node (i.e. NULL).
4. In each iteration, follow the following step 5-6.
5. Store the next node of current in the index.
6. Check if the data of the current node is greater than the next node. If it is greater, swap current and index.

```
// Sort the linked list
void sortLinkedList(struct Node** head_ref) {
  struct Node *current = *head_ref, *index = NULL;
  int temp;

  if (head_ref == NULL) {
    return;
  } else {
    while (current != NULL) {
      // index points to the node next to current
      index = current->next;

          while (index != NULL) {
        if (current->data > index->data) {
          temp = current->data;
          current->data = index->data;
          index->data = temp;
            }
```

```
              index = index->next;
            }
            current = current->next;
    }
  }
}
—-----------------------------------------------------------
// Linked list operations in C++
#include <stdlib.h>
#include <iostream>
using namespace std;

// Create a node
struct Node {
  int data;
  struct Node* next;
};

void insertAtBeginning(struct Node** head_ref, int new_data) {
  // Allocate memory to a node
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

  // insert the data
  new_node->data = new_data;
  new_node->next = (*head_ref);

  // Move head to new node
  (*head_ref) = new_node;
}

// Insert a node after a node
void insertAfter(struct Node* prev_node, int new_data) {
  if (prev_node == NULL) {
  cout << "the given previous node cannot be NULL";
  return;
  }

  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  new_node->data = new_data;
  new_node->next = prev_node->next;
  prev_node->next = new_node;
}

// Insert at the end
```

```
void insertAtEnd(struct Node** head_ref, int new_data) {
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  struct Node* last = *head_ref; /* used in step 5*/

  new_node->data = new_data;
  new_node->next = NULL;

  if (*head_ref == NULL) {
  *head_ref = new_node;
  return;
  }

  while (last->next != NULL) last = last->next;

  last->next = new_node;
  return;
}

// Delete a node
void deleteNode(struct Node** head_ref, int key) {
  struct Node *temp = *head_ref, *prev;

  if (temp != NULL && temp->data == key) {
  *head_ref = temp->next;
  free(temp);
  return;
  }
  // Find the key to be deleted
  while (temp != NULL && temp->data != key) {
  prev = temp;
  temp = temp->next;
  }

  // If the key is not present
  if (temp == NULL) return;

  // Remove the node
  prev->next = temp->next;

  free(temp);
}

// Search a node
bool searchNode(struct Node** head_ref, int key) {
  struct Node* current = *head_ref;

  while (current != NULL) {
  if (current->data == key) return true;
  current = current->next;
  }
  return false;
}

// Sort the linked list
void sortLinkedList(struct Node** head_ref) {
  struct Node *current = *head_ref, *index = NULL;
  int temp;

  if (head_ref == NULL) {
  return;
  } else {
  while (current != NULL) {
    // index points to the node next to current
    index = current->next;

    while (index != NULL) {
    if (current->data > index->data) {
      temp = current->data;
      current->data = index->data;
      index->data = temp;
    }
    index = index->next;
    }
    current = current->next;
  }
  }
}

// Print the linked list
void printList(struct Node* node) {
  while (node != NULL) {
  cout << node->data << " ";
  node = node->next;
  }
}

// Driver program
int main() {
```

```
struct Node* head = NULL;

insertAtEnd(&head, 1);
insertAtBeginning(&head, 2);
insertAtBeginning(&head, 3);
insertAtEnd(&head, 4);
insertAfter(head->next, 5);

cout << "Linked list: ";
printList(head);

cout << "\nAfter deleting an element: ";
deleteNode(&head, 3);
printList(head);

int item_to_find = 3;
if (searchNode(&head, item_to_find)) {
cout << endl << item_to_find << " is found";
} else {
cout << endl << item_to_find << " is not found";
}

sortLinkedList(&head);
cout << "\nSorted List: ";
printList(head);}
```