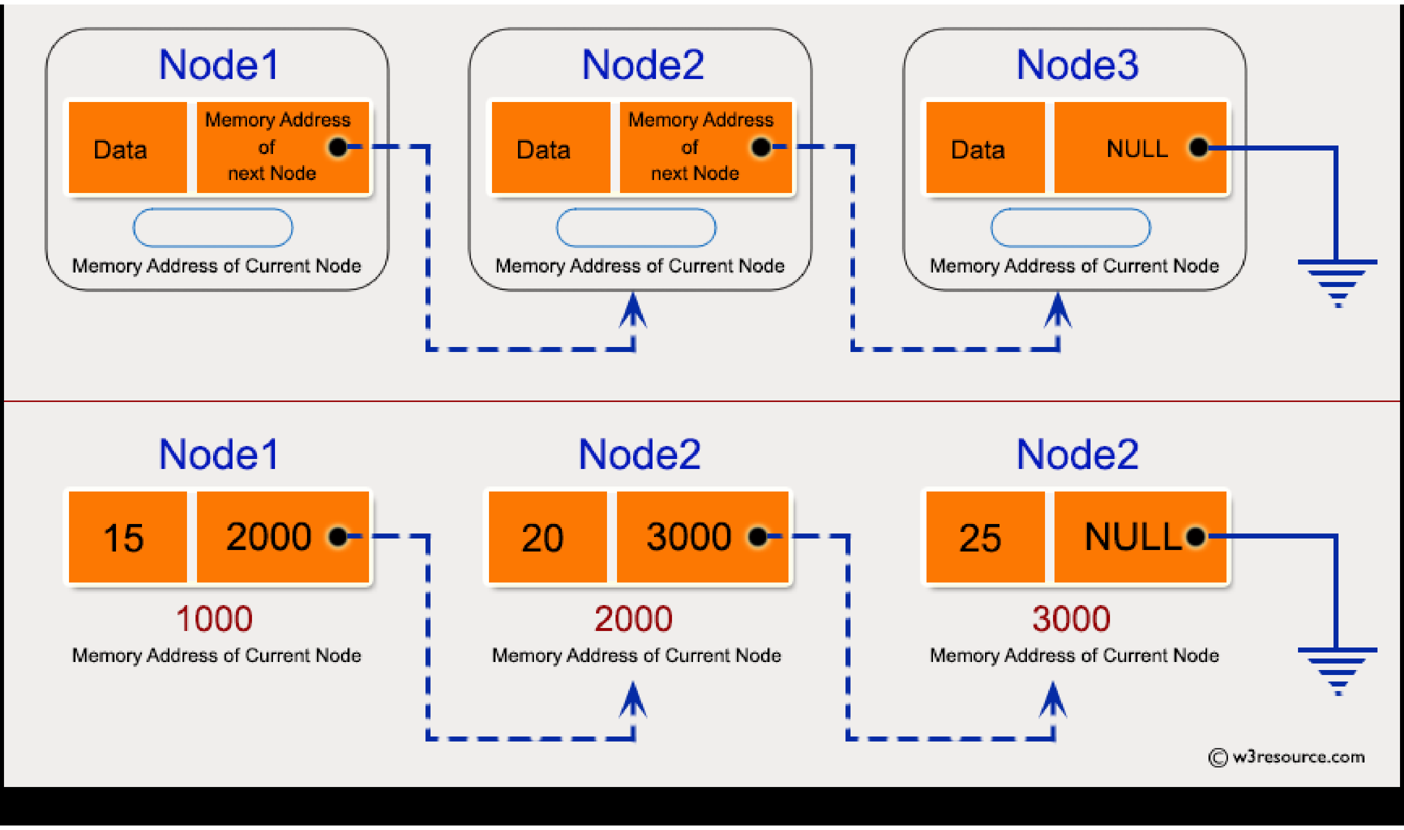
C Linked List : Exercise-1 with Solution

Write a program in C to create and display Singly Linked List.

Pictorial Presentation:


Node1
Data
Memory Address of next Node
Memory Address of Current Node
Node2
Data
Memory Address of next Node
Memory Address of Current Node
Node3
Data
NULL
Memory Address of Current Node
Node1
15
2000
1000
Memory Address of Current Node
Node2
20
3000
2000
Memory Address of Current Node
Node2
25
NULL
3000
Memory Address of Current Node
© w3resource.com

Sample Solution:

C Code:

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

struct node
{
    int num;                        //Data of the node
    struct node *nextptr;           //Address of the next node
}*stnode;

void createNodeList(int n); // function to create the list
void displayList();         // function to display the list

int main()
{
    int n;
              printf("\n\n Linked List : To create and display Singly Linked
List :\n");
              printf("--------------------------------------------------------
----\n");

    printf(" Input the number of nodes : ");
    scanf("%d", &n);
    createNodeList(n);
    printf("\n Data entered in the list : \n");
    displayList();
    return 0;
```

```
}
void createNodeList(int n)
{
    struct node *fnNode, *tmp;
    int num, i;
    stnode = (struct node *)malloc(sizeof(struct node));

    if(stnode == NULL) //check whether the fnnode is NULL and if so no memory
allocation
    {
        printf(" Memory can not be allocated.");
    }
    else
    {
// reads data for the node through keyboard

        printf(" Input data for node 1 : ");
        scanf("%d", &num);
        stnode->num = num;
        stnode->nextptr = NULL; // links the address field to NULL
        tmp = stnode;
// Creating n nodes and adding to linked list
        for(i=2; i<=n; i++)
        {
            fnNode = (struct node *)malloc(sizeof(struct node));
            if(fnNode == NULL)
            {
                printf(" Memory can not be allocated.");
                break;
            }
            else
            {
                printf(" Input data for node %d : ", i);
                scanf(" %d", &num);

                fnNode->num = num;      // links the num field of fnNode with
num
                fnNode->nextptr = NULL; // links the address field of fnNode
with NULL

                tmp->nextptr = fnNode; // links previous node i.e. tmp to the
fnNode
                tmp = tmp->nextptr;
            }
        }
    }
}
void displayList()
{
    struct node *tmp;
    if(stnode == NULL)
    {
        printf(" List is empty.");
    }
    else
    {
        tmp = stnode;
        while(tmp != NULL)
        {
            printf(" Data = %d\n", tmp->num);       // prints the data of
current node
            tmp = tmp->nextptr;                     // advances the position of
current node
        }
    }
}
```

Copy

Sample Output:

```
 Linked List : To create and display Singly Linked List :
-------------------------------------------------------------
 Input the number of nodes : 3
 Input data for node 1 : 5
 Input data for node 2 : 6
 Input data for node 3 : 7

 Data entered in the list :
 Data = 5
 Data = 6
 Data = 7
```

Linked List Operations: Traverse, Insert and Delete

In this tutorial, you will learn different operations on a linked list. Also, you will find implementation of linked list operations in C/C++, Python and Java.

There are various linked list operations that allow us to perform different actions on linked lists. For example, the insertion operation adds a new element to the linked list.
Here's a list of basic linked list operations that we will cover in this article.

- 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

Before you learn about linked list operations in detail, make sure to know about Linked List first.
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

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.
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

You can add elements to either the beginning, middle or end of the linked list.

1. Insert at the beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- 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

- Allocate memory for new node
- Store data
- Traverse to last node

- 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

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- 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

- Point head to the second node

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

2. Delete from end

- Traverse to second last element
- 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

- Traverse to element before the element to be deleted
- 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.

- Make `head` as the `current` node.
- Run a loop until the `current` node is `NULL` because the last element points to `NULL`.
- 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:

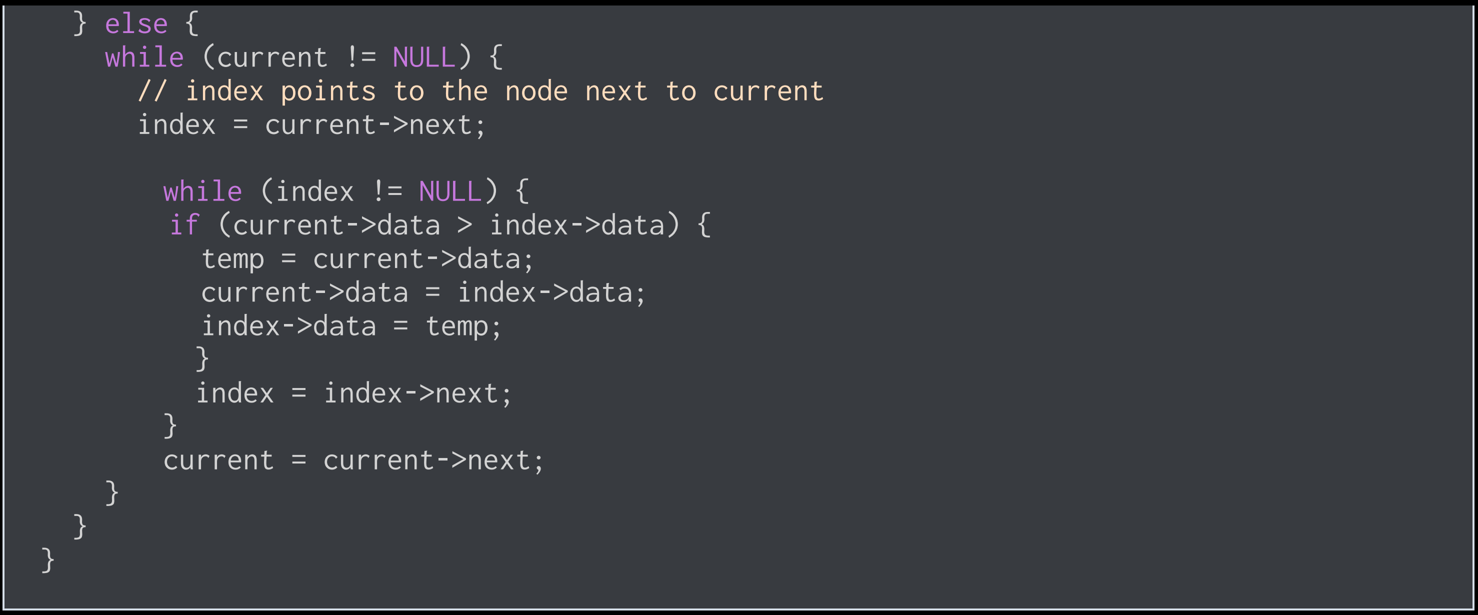
We will use a simple sorting algorithm, Bubble Sort, to sort the elements of a linked list in ascending order below.

1. Make the `head` as the `current` node and create another node `index` for later use.
2. If `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 `index`.
6. Check if the data of the current node is greater than the next node. If it is greater, swap `current` and `index`.

Check the article on bubble sort for better understanding of its working.

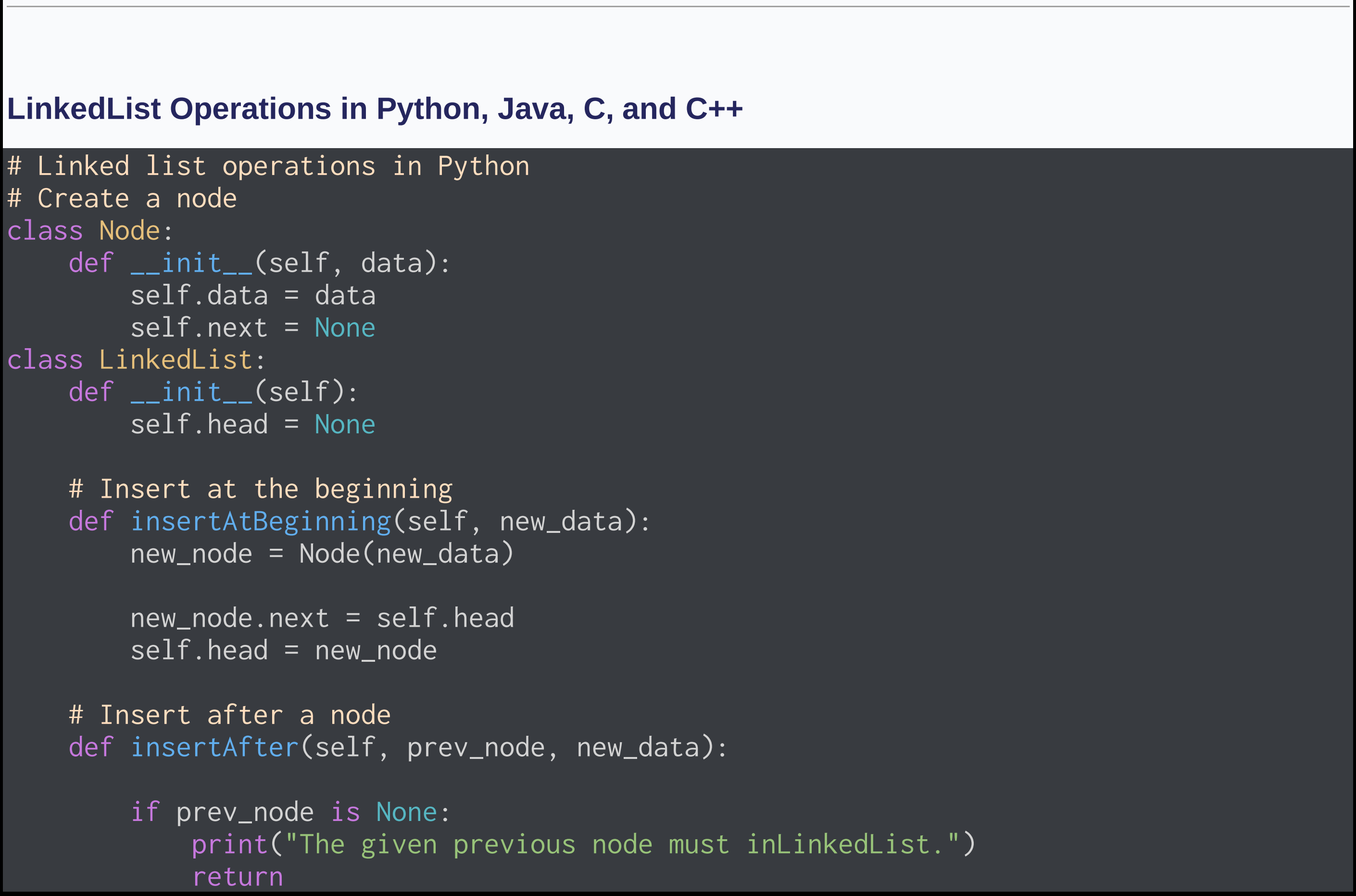
```
// Sort the linked list
void sortLinkedList(struct Node** head_ref) {
  struct Node *current = *head_ref, *index = NULL;
  int temp;

  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;
    }
  }
}
```



LinkedList Operations in Python, Java, C, and C++

```
# Linked list operations in Python
# Create a node
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None

    # Insert at the beginning
    def insertAtBeginning(self, new_data):
        new_node = Node(new_data)

        new_node.next = self.head
        self.head = new_node

    # Insert after a node
    def insertAfter(self, prev_node, new_data):

        if prev_node is None:
            print("The given previous node must inLinkedList.")
            return
```