Tutorial Link https://codequotient.com/tutorials/Linked-List : Operations - Deletion/5a59df31a4af025f554a0be8

TUTORIAL

Linked-List : Operations - Deletion

Chapter

1. Linked-List : Operations - Deletion

Topics

Deletion from Linked List

Let LIST be a linked list with a node N between nodes A and B. Suppose node N is to be deleted from the linked list. The deletion occurs as soon as the next pointer fields of node A points to node B. Therefore, when performing deletions, one must keep track of the address of the node which immediately precedes the node that is to be deleted.

Deletion Algorithms

We discuss two deletion algorithms.

(a) The first one deletes a node following a given node.

(b) The second one deletes the node with a given ITEM of information.

Deleting the Node Following a Given Node

Let LIST be a linked list in memory. Suppose we are given the location LOC of a node N in LIST. Furthermore, suppose we are given the location LOCP of the node preceding N or, when N is the first node, we are given LOCP=NULL. To delete the node LOC we have to change some pointers so that rest of list will operate correctly.

```
If LOCP = NULL
START = START[NEXT]    // if deleted node is first node
```

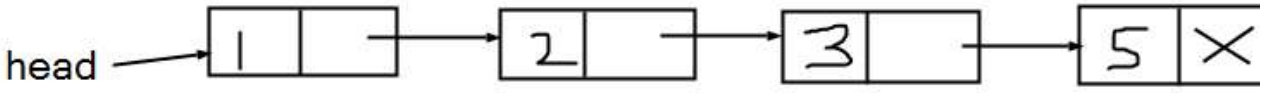
```
Else
LOCP[NEXT] = LOC[NEXT]    // if some middle node deleted
```

Deleting the Node with a Given ITEM of Information

Let LIST be a linked list in memory. Suppose we are given an ITEM of information and we want to delete from the LIST the first node N which contains ITEM. First we have to search for ITEM in LIST and find the Location of this node as well as its predecessor node. If N is the first node, we get LOCP = NULL, and if ITEM does not appear in LIST, we get LOC = NULL. Now to delete the node we can follow the same procedure as above.

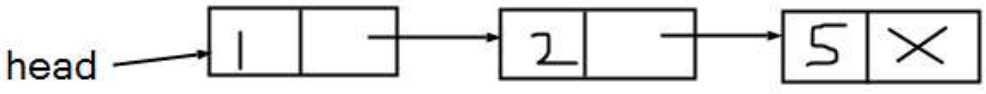
```
Search in LIST for ITEM and find LOC and LOCP
If LOC = NULL
Write “Item not in List”    // if item not found in list
If LOCP = NULL
START = START[NEXT]    // if deleted node is first node
Else
LOCP[NEXT] = LOC[NEXT]    // if some middle node deleted
```

So following figure illustrate the list before and after deletion: -



Node with DATA=3 is to be deleted

BEFORE DELETION



AFTER DELETION

Following is the implementation of these delete functions.

```c
struct Node* find(struct Node *head, int data)
```

```
{
  struct Node* temp=head;
  struct Node* tempp;
  if(head->data == data)
    return NULL;
  else
  {
    temp = head->next;
    tempp = head;
    while(temp != NULL)
    {
      if(temp->data == data)
        return tempp;
      tempp=temp;
      temp = temp->next;
    }
  }
  return NULL;
}
void deletenode(struct Node** head, struct Node* loc, struct Node* prev)
{
  if(prev ==NULL)     // loc is first node
    (*head) = (*head)->next;
  else
    prev->next = loc->next;
  free(loc);
}
void deleteitem(struct Node* head, int data)
{
  struct Node* prev;
  prev = find(head, data);
  if(prev ==NULL)     // loc is first node
    head = head->next;
  else
    prev->next = prev->next->next;
  //    free(prev->next);
}
// 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;
  }
  printf("\n");
}
int main()
{
  struct Node* head = NULL;
  printf("Linked List = ");
  printList(head);
  insert(&head, 6);    // Insert an element
  insert(&head, 5);    // Insert an element
  insert(&head, 4);    // Insert an element
  insert(&head, 3);    // Insert an element
  insert(&head, 2);    // Insert an element
  insert(&head, 1);    // Insert an element
  insert(&head, 0);    // Insert an element
  printf("Linked List = ");
  printList(head);
  deletenode(&head, head->next, head);  //delete the 2nd element
  printf("Linked List after deleting 2nd element = ");
  printList(head);
  deleteitem(head, 5);
  printf("Linked List after deleting element 5 = ");
  printList(head);
  return 0;
}

```

Java

```
static LinkList find(LinkList head, int data)
{
  LinkList temp=head;
  LinkList tempp;
  if(head.Data == data)
    return null;
  else
  {
    temp = head.next;
    tempp = head;
    while(temp != null)
    {
      if(temp.Data == data)
        return tempp;
      tempp = temp;
      temp = temp.next;
```

```
    }
  }
  return null;
}
static void deletenode(LinkList head, LinkList loc, LinkList prev)
{
  if(prev == null)     // loc is first node
    head = head.next;
  else
    prev.next = loc.next;
}
static void deleteitem(LinkList head, int data)
{
  LinkList prev;
  prev = find(head, data);
  if(prev == null)     // loc is first node
    head = head.next;
  else
    prev.next = prev.next.next;
}


public static void main(String[] args)
{
  LinkList head = null;
  int data=0;
  head = insertBeg(head, 6);    // At Beginning
  head = insertBeg(head, 5);    // At Beginning
  head = insertBeg(head, 4);    // At Beginning
  head = insertBeg(head, 3);    // At Beginning
  head = insertBeg(head, 2);    // At Beginning
  head = insertBeg(head, 1);    // At Beginning
  head = insertBeg(head, 0);    // At Beginning
  System.out.print("Linked List = ");
  traverse(head);
  deletenode(head, head.next, head);  //delete the 2nd element
  System.out.print("Linked List after deleting 2nd element = ");
  traverse(head);
  deleteitem(head, 5);
  System.out.print("Linked List after deleting element 5 = ");
  traverse(head);
}
}
```

Python 3

```
class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, new_data):
        node = Node(new_data)
        node.next = self.head
        self.head = node

    def deleteNode(self,loc,prev):
        if(prev is None):
            self.head = self.head.next;
        else:
            prev.next = loc.next;
        del loc;

    def deleteItem(self, data):
        temp = self.head
        if (temp is not None):
            if (temp.data == data):
                self.head = temp.next
                temp = None
                return

        while(temp is not None):
            if temp.data == data:
                break
            prev = temp
            temp = temp.next
        if(temp == None):
            return
        prev.next = temp.next
        temp = None

    def printList(self):
        node = self.head
        while not node is None:
            print (node.data,end=' -> '),
            node = node.next
        print();


if __name__=="__main__":
    linked_list = LinkedList();
```

```

    print('Linked List =',end=' ');
    linked_list.printList();
    linked_list.insert(6);
    linked_list.insert(5);
    linked_list.insert(4);
    linked_list.insert(3);
    linked_list.insert(2);
    linked_list.insert(1);
    linked_list.insert(0);
    print('Linked List =',end=' ');
    linked_list.printList();
    linked_list.deleteNode(linked_list.head.next,linked_list.head);
    print('Linked List =',end=' ');
    linked_list.printList();
    linked_list.deleteItem(5);
    print('Linked List =',end=' ');
    linked_list.printList();
```

C++

```
struct Node* find(struct Node *head, int data)
{
    struct Node* temp=head;
    struct Node* prev = NULL;
    if(head->data == data)
        return NULL;
    else{
        temp = head->next;
        prev = head;
        while(temp != NULL){
            if(temp->data == data)
                return prev;
            prev=temp;
            temp = temp->next;
        }
    }
  return NULL;
}
void deletenode(struct Node** head, struct Node* loc, struct Node* prev){
    if(prev ==NULL)     // loc is first node
        (*head) = (*head)->next;
    else
        prev->next = loc->next;
    delete (loc);
```

```
}

void deleteitem(struct Node* head, int data){
    struct Node* prev;
    prev = find(head, data);
    if(prev ==NULL)     // loc is first node
        head = head->next;
    else
        prev->next = prev->next->next;

}
// This function prints contents of linked list starting from head
void printList(struct Node *node){
    while (node != NULL){
        cout<<node->data<<"-> ";
        node = node->next;
    }
    cout<<endl;
}
int main(){
    struct Node* head = NULL;
    cout<<"Linked List = ";
    printList(head);
    insert(&head, 6);    // Insert an element
    insert(&head, 5);    // Insert an element
    insert(&head, 4);    // Insert an element
    insert(&head, 3);    // Insert an element
    insert(&head, 2);    // Insert an element
    insert(&head, 1);    // Insert an element
    insert(&head, 0);    // Insert an element
    cout<<"Linked List = ";
    printList(head);
    deletenode(&head, head->next, head);  //delete the 2nd element
    cout<<"Linked List after deleting 2nd element = ";
    printList(head);
    deleteitem(head, 5);
    cout<<"Linked List after deleting element 5 = ";
    printList(head);
    return 0;
}
```

```
Linked List =
Linked List = 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 ->
```

```
Linked List after deleting 2nd element = 0 -> 2 -> 3 -> 4 -> 5 -> 6 ->
Linked List after deleting element 5 = 0 -> 2 -> 3 -> 4 -> 6 ->
```

Applications

Lists are used to maintain POLYNOMIALS in the memory.

Linked-List Efficiency

Insertion and deletion at the beginning of a linked list are very fast. They involve changing only one or two references, which takes O(1) time.

Finding, deleting, or insertion next to a specific item requires searching through, on the average, half the items in the list. This requires O(N) comparisons. An array is also O(N) for these operations, but the linked list is nevertheless faster because nothing needs to be moved when an item is inserted or deleted. The increased efficiency can be significant, especially if a copy takes much longer than a comparison.

Of course, another important advantage of linked lists over arrays is that the linked list uses exactly as much memory as it needs, and can expand to fill all of the available memory. The size of an array is fixed when it's created; this usually leads to inefficiency because the array is too large, or to running out of room because the array is too small. Vectors, which are expandable arrays, may solve this problem to some extent, but they usually expand in

fixed-sized increments (such as doubling the size of the array whenever it's about to overflow). This is still not as efficient a use of memory as a linked list.