Available Online at www.ijcsmc.com/journal

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 7.056

IJCSMC, Vol. 11, Issue. 7, July 2022, pg.83 – 97

A New Vista of Performing Insertion and Deletion in Linked Lists

Arijeet Banerjee[1]; Pijush Kanti Kumar[2]
[1]Undergrad, Information Technology, Government College of Engineering and Textile Technology, Serampore-712201, India
[2]Asst. Prof, Information Technology, Government College of Engineering and Textile Technology, Serampore-712201, India
[1] arijeetbanerjee98@gmail.com; [2] pijush752000@yahoo.com
DOI: https://doi.org/10.47760/ijcsmc.2022.v11i07.008

Abstract: Insertion and deletion in linked lists. Linked lists are an example of linear data structures. They are made up of nodes which are connected to each other. The data can be accessed by sequential manner. In this paper we will create an abstract model of linked list where the creation of a linked list of n nodes will take a time complexity of O(n) and the insertion and deletion will take a time complexity of O(1) or constant time at the tail position and head position and also reduce the runtime of insertion and deletion at middle or any index in between to some extent. This abstract model of the linked list can also be used as a stack and a queue because it will contain all properties of stack and queue data structures.
Keywords: Linked list, PushAtBack, PushAtFront, PushAtIndex, PopAtIndex, PopAtBack, PopAtFront, size

1. RESEARCH OBJECTIVES

To discuss on unique ways to perform insertion and deletion in a linked list and reducing their runtime. The creation of a linked list of n nodes will take a runtime of O(n^2) if we traverse the list while inserting nodes. In this paper we will reduce the time complexity of creation to O(n). The insertion and the deletion of the linked list will take a runtime of O(n) at

the tail position if we traverse the list but this can be reduced to O(1) or constant time. This abstract model will also serve as a template for data structures like stack and queue.

2. LITERATURE REVIEW

We will use several algorithms to perform the insertion and the deletion in the linked lists. The runtime of an algorithm for a specific input depends on the number of operations executed. The more the number of operations the more will be the runtime of that algorithm. The time complexity is the description of the asymptotic behavior of the running time as the input tends to a very large number like infinity. We can express the algorithmic complexity using Big – O notation. O(1) means a constant time function which is the best possible runtime of an algorithm. O(n) means a linear time function which takes order of n and O(n^2) is a quadratic time function which is of order N squared. The Big O asymptotic notation basically gives us the upper bound idea. Mathematically we can say a function f(n) is said to be O(g(n)) if there exists a positive integer n0 and a positive constant c, such that for all n > n0 f(n) <= c.g(n). In this paper we will create an abstracted model of linked list and try to reduce the time complexity of the following operations. The creation of the linked list takes O(n^2) time for creation in general. We can reduce the time complexity to O(n) using some techniques which will be discussed in this paper. Similarly, insertion and deletion methods can also be optimised to avoid entire traversal and reduce runtime. The linked list is a collection of nodes which are connected to each other. In C language we can use structure to represent the node. The node will be of self-referencing type. It will contain one data type and two others for storing the previous and next node address. The abstracted model of the linked list that we will be using here will be sometimes similarly to the double linked list but different from the perspectives of algorithms used for insertion and deletion. The detailed analysis will be discussed here in this paper. Also, in this paper we are assuming that malloc function under standard library <stdlib.h> is never giving NULL and there is no memory exhaustion.

3. INTRODUCTION TO NODE

The nodes will be of self-referencing type. In C language we can create the node as shown below.

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

The nodes will contain three fields. One is for the data field and the rest two is storing address of another nodes. Hence, we call these self-referencing nodes. Fig1 shows the structure of the node.

Fig.1 Detailed structure of a node

Struct node * back stores the address of the previous node and struct node * next stores the address of the next node.

Before implementing the linked-list we will use two pointers which are head and tail. Head and tail are two pointers which are initially pointing to NULL.

4. INTRODUCTION TO THE ABSTRACT MODEL OF LINKED LIST

In this abstract model of linked list, we will create some new functions for insertion and deletion of nodes at various locations in lesser runtime. This abstracted model is somewhat similar to a double linked list with the below operations.

PushAtFront(data) operation takes a data as an argument and inserts it at the starting position of the linked list and all successive insertions are performed from the beginning.

PushAtBack(data) operation takes a data as an argument and inserts it at the ending position of the linked list and all successive insertions are performed from the end.

PushAtIndex(position,data) operation takes two arguments from the user. One is the data for the node and another is the position where the node has to be inserted. It inserts the node at that desired position.

PopAtFront() operation deletes the first node of the linked list.

PopAtBack() operation deletes the last node of the linked list.

PopAtIndex(position) operation takes the position as the argument and deletes the node at that specified position.

Size() operation returns the size of the linked list.

Createlist() creates a list of n nodes taking each data from the user.

These operations are meant for better working of the linked list with a higher level of abstraction. In this paper all the working of the above functions will be discussed. We will use a variable count to keep track of the size of the list. Count is a global variable which increments by one when a node is inserted and decrements by one when a node is deleted.

© 2022, IJCSMC All Rights Reserved

4.1 PUSHATFRONT OPERATION: EXPLANATION AND IMPLEMENTATION

PushAtFront operation means a new node will be inserted from the start. Hence, if the linked list contains the following data: 2 3 4 5. PushAtFront(8) will make the linked list 8 2 3 4 5.

The functions work in the following manner. At first a temp node is created. The standard library <stdlib.h> is used to allocate dynamic memory using malloc. Below is a short snippet of the code.

```
struct node *temp = (struct node *)malloc(sizeof(struct node));
```

This temp node contains three parts. One part is for storing the data and the other two for storing addresses. Initially we set both the address to NULL. Whatever data is to be stored is stored in the data part of the node. Now after this temp node is created, we check if the linked list is empty or not.

If the linked list is empty, we assign the temp node as the only node in the linked list with both the head and the tail pointers pointing to this node. Below is the figure of the situation.

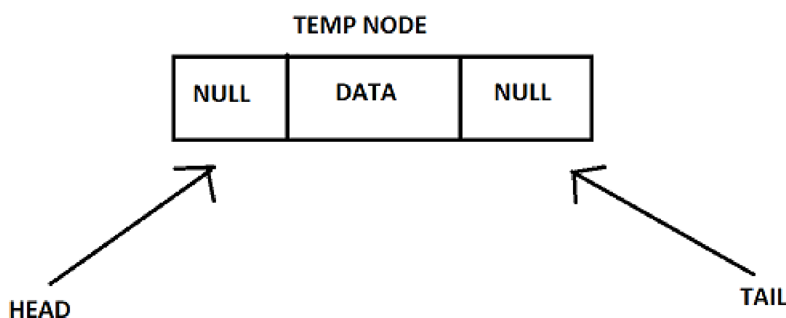

Fig 2. Abstracted model of this linked list containing only one node

Above was the case if the linked list was initially empty and then just one node is inserted, but in case the linked list is not empty, the following algorithm is performed. At first a temp node is created. The standard library <stdlib.h> is used to allocate dynamic memory using malloc.

Below is a short snippet of the code.

```
struct node *temp = (struct node *)malloc(sizeof(struct node));
```

This temp node contains three parts. One part is for storing the data and the other two for storing addresses. Initially we set both the address to NULL. Whatever data is to be stored is

© 2022, IJCSMC All Rights Reserved

stored in the data part of the node. This data can be given by the user. We update the next pointer of this newly created temp node to the head and the back pointer to Null. The back pointer of the head node is now pointed to temp. In this way the node is added to the linked list. We update the head node to temp node. The tail pointer remains where it was initially.

Fig 3 shows us two linked lists before and after insertion of the temp node by pushAtFront operation. The tail pointer of the linked list remains wherever it was before the operation. The head pointer changes every time and as a new node is inserted from the front of the linked list head is then updated as that new node. The time complexity of this operation is O(1). The PushAtFront operation takes constant time. Even if there are ten thousand nodes in the linked list it would take the same time as there is one node in the linked list. The source code of the operation has been implemented in C language below. Also, we are assuming that malloc returns the specified block of memory upon calling and there is no exhaustion of heap memory.

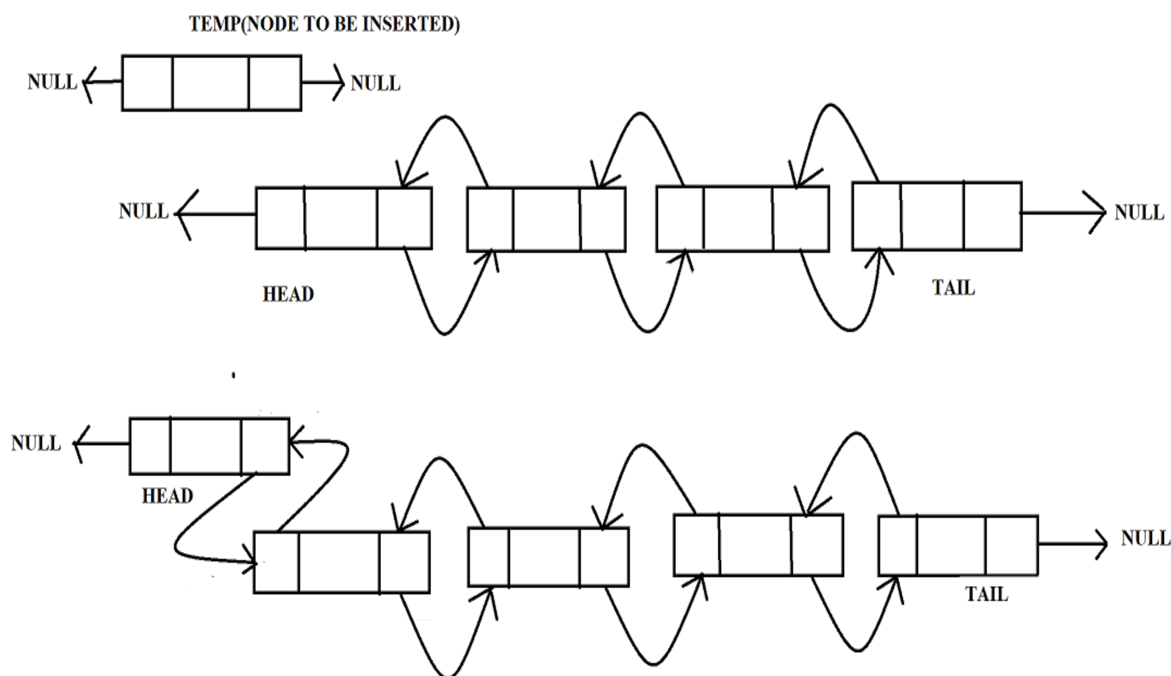

Fig.3 Before insertion and after insertion of a linked list

```
void pushAtFront(int value)
{
   struct node *temp = (struct node *)malloc(sizeof(struct node));
   // Assuming that malloc returns the desired block of memory and there is no exhaustion of
memory
   temp->data = value;
   temp->next = NULL;
   temp->back = NULL;
```

© 2022, IJCSMC All Rights Reserved

```
    if (head == NULL)
    {
        head = temp;
        tail = temp;
    }
    else
    {
        temp->next = head;
        temp->back = NULL;
        head->back = temp;
        head = temp;
    }
    count++;
}
```

4.2. POPATFRONT OPERATION: EXPLANATION AND IMPLEMENTATION

PopAtFront means the nodes will be popped or deleted from the start. Hence, if the linked list contains the following data 2 3 4 5. PopAtFront() will make the linked list 3 4 5. Thus, it will delete the node with data 2. If the size of the linked is 0 or it is empty then the function PopAtFront() will simply display a message "The list is already empty" else it will delete the node using the below algorithm.

First, we point a new pointer say "F" to head, then we will simply update the head pointer to its next node and free the pointer "F" using Free function under "stdlib.h" module. This will delete the first node of the list or infact pop it. We then assign back pointer of the head to NULL again. If there is only one node, then we assign head and tail pointers to NULL and free that one node using free() function. Fig 4 shows us the visual representation.

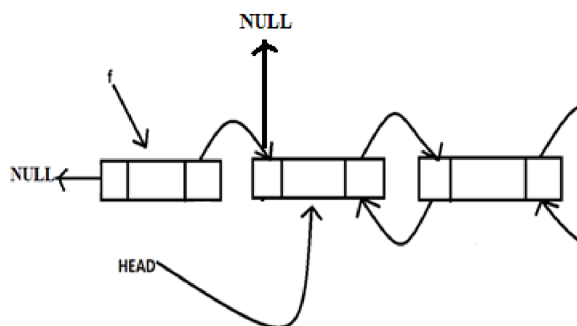

Fig.4 Deletion of the first node

The above algorithm for PopATFront has been implemented in C language below.

```
void popAtFront()
{
    if (count != 0)
```

```
    {
        struct node *f = head;
        if (count == 1)
        {
            head = NULL;
            tail = NULL;
            free(f);
            count--;
        }
        else
        {
            head = head->next;
            head->back = NULL;
            count--;
            free(f);
        }
    }
    else
    {
        printf("Sorry List is already empty\n");
    }
}
```

The time complexity for popAtFront operation will take O(1) time same as pushAtFront operation. Thus, we can pop from the beginning in constant time of the linked list.

4.3 PUSHATBACK OPERATION: EXPLANATION AND IMPLEMENTATION

The PushAtBack operation will work in the following manner. Say the linked list contains data as follows: 4 4 5 3 2. Now PushAtBack(6) will take argument 6 as an int value which will inserted from the back of the linked list. The updated linked list will be 4 4 5 3 2 6. Similarly like PushAtFront function, if the linked list is initially empty, we would create a temp node with both of its pointer head and tail set to NULL and then assign head and tail both to the temp node. Now suppose another node has to be entered from the back. Firstly, a temp node will be created. Tail pointer is also pointing to our first node which is also the last node as there is just one single node in the linked list. We will make the back pointer of newly created temp node to now point to tail and the next pointer of our node which is being pointed by tail in the linked list to point to temp and then update tail to temp. This will simply update the linked list by adding a new node and tail will point towards the last node. Let's say a third node has to be inserted in the linked list. Tail is pointing the last node right now. We will create another temp node and then update temp->back to tail and tail->next to temp and

© 2022, IJCSMC All Rights Reserved

lastly, we will again update the tail to temp so that tail always points to the last node. So, if this goes on, we can observe that since tail is already pointing to the last node of the list, there is no need for traversing the entire list to perform insertion at end. The head pointer remains where it was before and after the insertion. The tail pointer gets updated every time the pushAtBack operation is performed.

In fig.5 a fifth node is being inserted in the linked list. The time complexity of this insertion will be O(1). Thus we can perform insertion at back in O(1) time complexity, without the need for traversing the entire linked and insert. All successive insertions will take O(1) time in the linked list. This algorithm will drastically reduce the time complexity to constant time making this abstract model of linked list better than normal linked lists. This algorithm has been implemented in C language below.

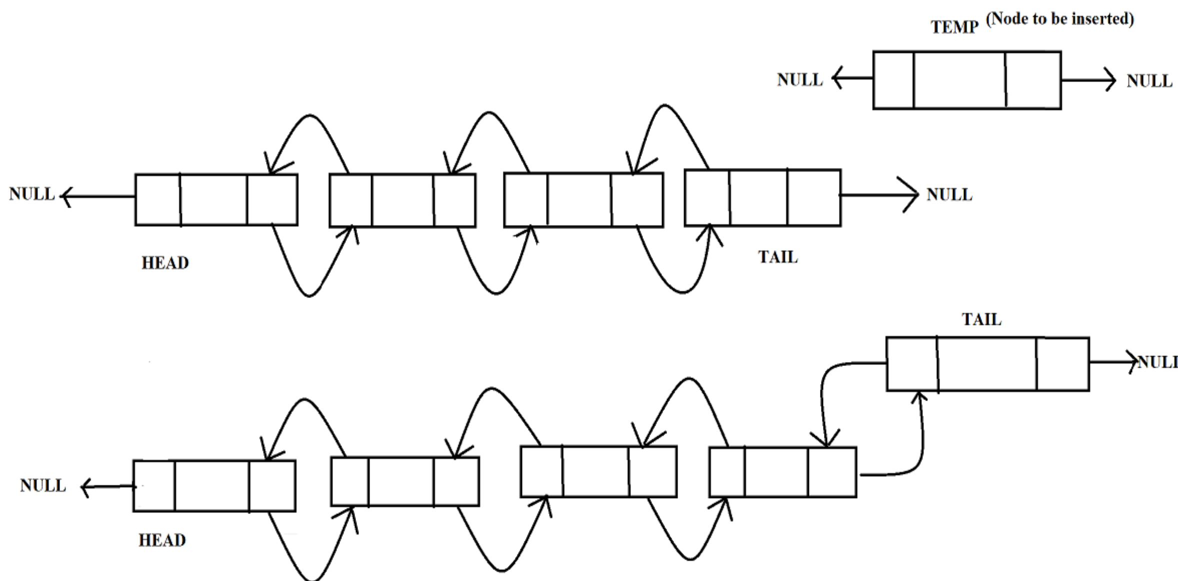

Fig.5. Insertion at tail

```
void pushAtBack(int value)
{
   struct node *temp = (struct node *)malloc(sizeof(struct node));
   temp->data = value;
   temp->next = NULL;
   if (head == NULL)
   {
      head = temp;
      tail = temp;
   }
   else
   {
      temp->back = tail;
      tail->next = temp;
```

```
        tail = temp;
    }
    count++;
}
```

4.4 POPATBACK OPERATION: EXPLANATION AND IMPLEMENTATION

Suppose the linked list contains the elements 3 4 5 6 9. Now popAtBack operation will delete the last element of the linked list and the new linked list will become 3 4 5 6. Hence the last element of the linked list will be popped. However, this pop operation will be somewhat different from the popAtFirst. At first the size of the linked list has to be checked. If the size of the linked list is 0, which means the linked list is empty then the function popAtBack will simply print "Sorry list is already empty." Now if the linked list is not empty, we create a new pointer say "f". We point this new pointer "f" to the node tail is pointing which is infact the node which has to be deleted. We point the tail to its previous node. Since the last node will contain the address of the second last node this is done in constant time. After tail is updated, tail->next is pointed to NULL and then we free this pointer "f" with free function under standard library <stdlib.h>. This will delete the last node of the linked list. The tail is now updated to the last node of the linked list. Similarly, another node will be deleted and tail will be updated again. Thus, we can see in order to delete the last node we don't need to traverse the entire linked list and delete the last node. We can simply take advantage of the tail pointer and delete the last node in O(1) time complexity or constant time. Now there is a corner case. If there is just one node in the linked list and popAtBack is performed. Head and tail are both now pointing to that single node. We update tail to tail->back which makes tail point to null, but our head pointer still points to the node which will be freed. So now we update head to Null also to avoid dangling pointers.

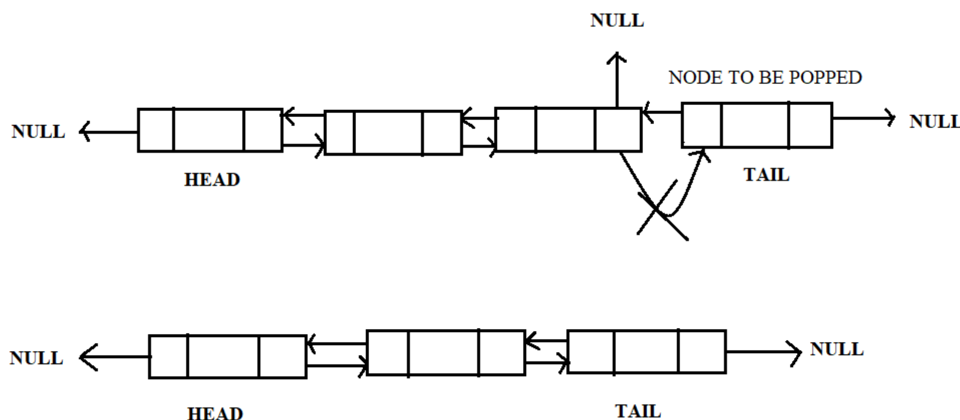

Fig.6 The list before and after deletion

The source code for the above function written in C is

```
void popAtBack()
{
   if (count != 0)
   {
      if (count == 1)
      {
         struct node *f = tail;
         tail = NULL;
         head = NULL;
         free(f);
         count--;
      }
      else
      {
         struct node *f = tail;
         tail = tail->back;
         tail->next = NULL;
         count--;
         free(f);
      }
   }
   else
   {
      printf("Sorry List is already empty\n");
   }
}
```

The time complexity of the deletion from end is O(1). Thus, in constant time we can delete from the end of the linked list without traversing the entire linked list.

PushAtFront, PopAtFront, PushAtBack and PopAtback functions can perform their functions in O(1) time constant time. Even if the linked list contains 100 nodes or 1000 nodes or millions of nodes, the insertion and deletion from both tail and front would take constant time.

4.5 PUSHATINDEX OPERATION: EXPLANATION AND IMPLEMENTATION

We can push a node at a desired index. Say our linked list was 2 3 5 7 8. Now push at index(1,8) will make the new linked list 2 8 3 5 7 8. So function Pushatindex takes 2 inputs as arguments i.e. position and data. The function inserts a node with the data at the desired position. If the index is 0 this means that we need to perform inserting a node in the first position. We can simple call our PushAtFront() function for this operation. If the index is n where n is the number of nodes, we can again simple call out function PushatBack().

© 2022, IJCSMC All Rights Reserved

Now there are two probabilities of inserting a new node. In the first scenario the index can lie between 0 and n/2 or the first half where n is the number of nodes. In this case the node has to be inserted by traversing from front to that desired position and then create a new node and insert it in between. In the second scenario, the index can lie between n/2+1 and n-1 or the second half. Now in order to insert a node in this range we can do back traversal from the tail pointer and then insert the node. By performing the above algorithm, we don't need to do entire traversal in worst cases. If the position to be deleted is near the tail pointer, we traverse from the tail else we traverse from the head.

A sample C code is attached below:

```
void push_at_index(int index, int value)
{
   if (index == 0)
   {
      pushAtFront(value);
   }
   else if (index == count)
   {
      pushAtBack(value);
   }
   else if (index <= count / 2)
   {
      struct node *ptr = head;
      struct node *temp = (struct node *)malloc(sizeof(struct node));
      temp->data = value;
      temp->next = NULL;
      temp->back = NULL;
      for (int i = 0; i < index; i++)
      {
         ptr = ptr->next;
      }
      temp->back = ptr->back;
      temp->next = ptr;
      ptr->back->next = temp;
      ptr->back = temp;
      count++;
   }
   else if (index > count/2 && index < count)
   {
      struct node *ptr = tail;
      struct node *temp = (struct node *)malloc(sizeof(struct node));
      temp->data = value;
      temp->next = NULL;
      temp->back = NULL;
      for (int i = 0; i < count - index - 1; i++)
```

© 2022, IJCSMC All Rights Reserved

```
        {
           ptr = ptr->back;
        }
        temp->back = ptr->back;
        temp->next = ptr;
        ptr->back->next = temp;
        ptr->back = temp;
        count++;
    }
    else
    {
        printf("Sorry Index is out of range!!\n");
    }
}
```

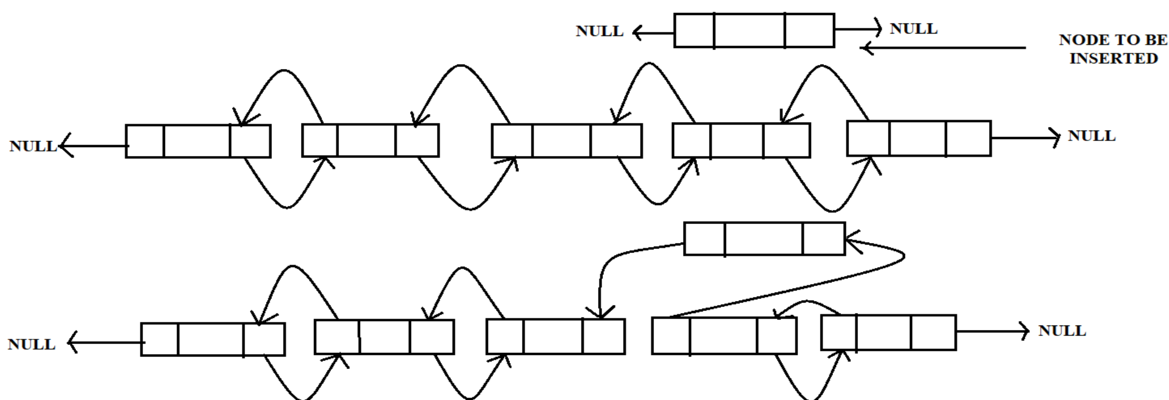

Fig.7 Insertion of a node in the middle

4.6 POPATINDEX OPERATION: EXPLANATION AND IMPLEMENTATION

The PopAtIndex will take an index as argument from the user and delete that node at that position. If the node is at index 0 or index n-1 where n is the number of nodes, then we can simply call the PopAtFront and PopAtBack index respectively. Else just like the pushAtIndex function we can do traversal from the beginning or traversal from the end depending on whether the index lies in range 1 to n/2 which is the first half of the list or from n/2+1 to n-2 which is the second half of the list respectively. In such cases, we won't be traversing the entire array in worst cases and can reduce the time complexity to some extent. The same concept of traversal is used here as in PushAtIndex. If the node to be deleted is near the tail pointer, then we can traverse from the back, else we can traverse from the front. Fig. 8 shows the visual representation of the scenario. To delete the node, we will simple iterate to that node and apply the algorithm: ptr->back->next=ptr->next and ptr->next->back=ptr->back. Then we shall free the pointer ptr. This will delete the node. We will not need to traverse the entire list even in worse cases using this algorithm.

© 2022, IJCSMC All Rights Reserved

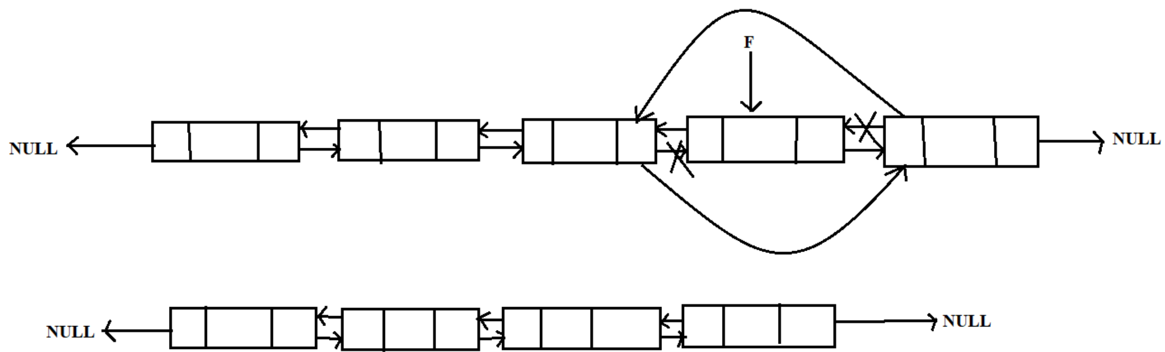

Fig.8 The node pointed by F is being deleted

The following C code has been shown below:

```
void popAtIndex(int index)
{
   if (index==0)
   {
      popAtFront();
   }
   else if (index==count-1)
   {
      popAtBack();
   }
   else if (index <= count/2)
   {
      struct node * ptr = head;
      for (int i = 0; i < index; i++)
      {
         ptr=ptr->next;
      }
      ptr->back->next=ptr->next;
      ptr->next->back=ptr->back;
      free(ptr);
      count--;
   }
   else if (index >count/2 && index < count )
   {
      struct node * ptr = tail;
      for (int i = 0; i < count - index - 1; i++)
      {
         ptr=ptr->back;
      }
      ptr->back->next=ptr->next;
      ptr->next->back=ptr->back;
      free(ptr);
```

© 2022, IJCSMC All Rights Reserved

```
      count--;
   }
   else
   {
      printf("Sorry Index is out of range!!\n");
   }
}
```

4.6 SIZE OPERATION: EXPLANATION AND IMPLEMENTATION

The size of the linked list can be calculated with the help of a global variable. With each pushAtBack, PushAtfront and PushAtIndex operations the size of the linked list increases by one, so we increase the count by 1. In case of the popAtfirst, popatback and popAtIndex operations the size of the linked list decreases by 1 so we decrease the count by 1. So we can get the size of the linked list in O(1) time complexity with the help of a global variable.

5. CREATION OF A LIST: EXPLANATION AND IMPLEMENTATION

If n data are to be stored in the linked list, we can use the concept of n push_at_back operations to create the list. One push_Pop will take O(1) operation so to create a list with n nodes it will take O(n) time complexity. Say, the list is empty initially. Lets PushAtBack(5), this will make the list NULL<- 5 ->NULL, then PushAtBack(7) will make the list NULL<- 5 <-> 7-> NULL. Again PushAtBack(51) will make the list NULL <- 5 <-> 7 <-> 51 -> NULL.

Similarly, we can create n number of nodes to create the list. We can create the list in O(n) time complexity where n is the number of nodes. The sample C code for this algorithm is show below.

```
void createlist()
{
   printf("Enter the number of nodes:");
   int n;
   scanf("%d",&n);
   for (int i = 0; i < n; i++)
   {
       int data;
       printf("Enter data:");
       scanf("%d",&data);
       pushAtBack(data);
   }
   display_linked_list();
}
```

© 2022, IJCSMC All Rights Reserved

6. CONCLUSIONS AND FUTURE WORK

In this abstract model of linked list, we were able to create the linked list in O(n) where n is the number of nodes. Furthermore we created functions like pushatback, pushatfront, popatfront and popatback to perform insertion at back, insertion at front, delete at front and delete at back respectively in O(1) time complexity or constant time. We have also optimised insertion in middle and deletion in middle. Furthermore, we have added properties of data structure like stack and queue. Stack is LIFO (Last In First Out) data structure. We can implement a stack with this abstract model of linked list. PushAtBack() operation will act as the push operation for the stack and PopAtBack will act as the pop operation for the stack. Queue is FIFO(First In First Out) data structure. Similarly, the PushAtBack and PopAtFront operation will act as enqueue and dequeue operations for a queue respectively. These operations will also be running in constant time.

However, each node contains two address fields other than data field. Therefore, due to this reason, size of the list increases. Also, operations like insertion and deletion in the middle are optimised but still they take linear run time complexity. In future, more optimised algorithms are be created to make operations in middle run in constant time complexity and reduce the space occupied by nodes.

REFERENCES

[1] Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
[2] https://www.geeksforgeeks.org/data-structures/linked-list/
[3] https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm
[4] https://www.javatpoint.com/singly-linked-list
[5] https://en.wikipedia.org/wiki/Linked_list
[6] https://opendatastructures.org/ods-java/3_Linked_Lists.html
[7] Review of Shortest Path Algorithm (ijcsmc.com)

© 2022, IJCSMC All Rights Reserved