Contents

Chapter 1

Linked List vs Array

Difficulty Level: Rookie

Both Arrays and Linked List can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Following are the points in favour of Linked Lists.

(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, and in practical uses, upper limit is rarely reached.

(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 shifted.

For example, suppose 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.

So Linked list provides following two advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion

Linked lists have following drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2) Extra memory space for a pointer is required with each element of the list.
3) Arrays have better cache locality that can make a pretty big difference in performance.

Please also see this thread.

References:
http://cslibrary.stanford.edu/103/LinkedListBasics.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/linked-list-vs-array/

Chapter 2

How to write C functions that modify head pointer of a Linked List?

Consider simple representation (without any dummy node) of Linked List. Functions that operate on such Linked lists can be divided in two categories:

1) Functions that do not modify the head pointer: Examples of such functions include, printing a linked list, updating data members of nodes like adding given a value to all nodes, or some other operation which access/update data of nodes
It is generally easy to decide prototype of functions of this category. We can always pass head pointer as an argument and traverse/update the list. For example, the following function that adds x to data members of all nodes.

```
void addXtoList(struct node *node, int x)
{
    while(node != NULL)
    {
        node->data = node->data + x;
        node = node->next;
    }
}
```

2) Functions that modify the head pointer: Examples include, inserting a node at the beginning (head pointer is always modified in this function), inserting a node at the end (head pointer is modified only when the first node is being inserted), deleting a given node (head pointer is modified when the deleted node is first node). There may be different ways to update the head pointer in these functions. Let us discuss these ways using following simple problem:

"Given a linked list, write a function deleteFirst() that deletes the first node of a given linked list. For example, if the list is 1->2->3->4, then it should be modified to 2->3->4

Algorithm to solve the problem is a simple 3 step process: (a) Store the head pointer (b) change the head pointer to point to next node (c) delete the previous head node.
Following are different ways to update head pointer in deleteFirst() so that the list is updated everywhere.

2.1) *Make head pointer global:* We can make the head pointer global so that it can be accessed and updated in our function. Following is C code that uses global head pointer.

```
// global head pointer
struct node *head = NULL;

// function to delete first node: uses approach 2.1
// See http://ideone.com/ClfQB for complete program and output
void deleteFirst()
{
    if(head != NULL)
    {
       // store the old value of head pointer
       struct node *temp = head;

       // Change head pointer to point to next node
       head = head->next;

       // delete memory allocated for the previous head node
       free(temp);
    }
}
```

See thisfor complete running program that uses above function.

This is not a recommended way as it has many problems like following:
a) head is globally accessible, so it can be modified anywhere in your project and may lead to unpredictable results.
b) If there are multiple linked lists, then multiple global head pointers with different names are needed.

See thisto know all reasons why should we avoid global variables in our projects.

2.2) *Return head pointer:* We can write deleteFirst() in such a way that it returns the modified head pointer. Whoever is using this function, have to use the returned value to update the head node.

```
// function to delete first node: uses approach 2.2
// See http://ideone.com/P5oLe for complete program and output
struct node *deleteFirst(struct node *head)
{
    if(head != NULL)
    {
       // store the old value of head pointer
       struct node *temp = head;

       // Change head pointer to point to next node
       head = head->next;

       // delete memory allocated for the previous head node
       free(temp);
    }

    return head;
}
```

See thisfor complete program and output.
This approach is much better than the previous 1. There is only one issue with this, if user misses to assign the returned value to head, then things become messy. C/C++ compilers allows to call a function without assigning the returned value.

```
head = deleteFirst(head);  // proper use of deleteFirst()
deleteFirst(head);  // improper use of deleteFirst(), allowed by compiler
```

2.3) *Use Double Pointer:* This approach follows the simple C rule: *if you want to modify local variable of one function inside another function, pass pointer to that variable.* So we can pass pointer to the head pointer to modify the head pointer in our deleteFirst() function.

```
// function to delete first node: uses approach 2.3
// See http://ideone.com/9GwTb for complete program and output
void deleteFirst(struct node **head_ref)
{
    if(*head_ref != NULL)
    {
       // store the old value of pointer to head pointer
       struct node *temp = *head_ref;

       // Change head pointer to point to next node
       *head_ref = (*head_ref)->next;

       // delete memory allocated for the previous head node
       free(temp);
    }
}
```

See this for complete program and output.
This approach seems to be the best among all three as there are less chances of having problems.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/how-to-write-functions-that-modify-the-head-pointer-of-a-linked-list/

Chapter 3

Swap nodes in a linked list without swapping data

Given a linked list and two keys in it, swap nodes for two given keys. Nodes should be swapped by changing links. Swapping data of nodes may be expensive in many situations when data contains many fields.

It may be assumed that all keys in linked list are distinct.

Examples:

```
Input:  10->15->12->13->20->14,  x = 12, y = 20
Output: 10->15->20->13->12->14

Input:  10->15->12->13->20->14,  x = 10, y = 20
Output: 20->15->12->13->10->14

Input:  10->15->12->13->20->14,  x = 12, y = 13
Output: 10->15->13->12->20->14
```

This may look a simple problem, but is interesting question as it has following cases to be handled.
1) x and y may or may not be adjacent.
2) Either x or y may be a head node.
3) Either x or y may be last node.
4) x and/or y may not be present in linked list.

How to write a clean working code that handles all of the above possibilities.

We strongly recommend to minimize your browser and try this yourself first.

The idea it to first search x and y in given linked list. If any of them is not present, then return. While searching for x and y, keep track of current and previous pointers. First change next of previous pointers, then change next of current pointers. Following is C implementation of this approach.

```
/* This program swaps the nodes of linked list rather
   than swapping the field from the nodes.*/

#include<stdio.h>
```

```
#include<stdlib.h>

/* A linked list node */
struct node
{
    int data;
    struct node *next;
};

/* Function to swap nodes x and y in linked list by
   changing links */
void swapNodes(struct node **head_ref, int x, int y)
{
   // Nothing to do if x and y are same
   if (x == y) return;

   // Search for x (keep track of prevX and CurrX
   struct node *prevX = NULL, *currX = *head_ref;
   while (currX && currX->data != x)
   {
       prevX = currX;
       currX = currX->next;
   }

   // Search for y (keep track of prevY and CurrY
   struct node *prevY = NULL, *currY = *head_ref;
   while (currY && currY->data != y)
   {
       prevY = currY;
       currY = currY->next;
   }

   // If either x or y is not present, nothing to do
   if (currX == NULL || currY == NULL)
       return;

   // If x is not head of linked list
   if (prevX != NULL)
       prevX->next = currY;
   else // Else make y as new head
       *head_ref = currY;

   // If y is not head of linked list
   if (prevY != NULL)
       prevY->next = currX;
   else  // Else make x as new head
       *head_ref = currX;

   // Swap next pointers
   struct node *temp = currY->next;
   currY->next = currX->next;
   currX->next  = temp;
}
```

```
/* Function to add a node at the begining of List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Druver program to test above function */
int main()
{
    struct node *start = NULL;

    /* The constructed linked list is:
     1->2->3->4->5->6->7 */
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before calling  swapNodes() ");
    printList(start);

    swapNodes(&start, 4, 3);

    printf("\n Linked list after calling  swapNodes() ");
    printList(start);

    return 0;
}
```

Output:

```
Linked list before calling  swapNodes() 1 2 3 4 5 6 7
Linked list after calling  swapNodes() 1 2 4 3 5 6 7
```

Optimizations: The above code can be optimized to search x and y in single traversal. Two loops are used to keep program simple.

This article is contributed by **Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/swap-nodes-in-a-linked-list-without-swapping-data/

Category: Linked Lists

Chapter 4

Write a function to get Nth node in a Linked List

Write a GetNth() function that takes a linked list and an integer index and returns the data value stored in the node at that index position.

Algorithm:

```
1. Initialize count = 0
2. Loop through the link list
     a. if count is equal to the passed index then return current
         node
     b. Increment count
     c. change current to point to next of the current.
```

Implementation:

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
    of a list and an int, push a new node on the front
    of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
```

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

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Takes head pointer of the linked list and index
    as arguments and return data at index*/
int GetNth(struct node* head, int index)
{
    struct node* current = head;
    int count = 0; /* the index of the node we're currently
                  looking at */
    while (current != NULL)
    {
       if (count == index)
          return(current->data);
       count++;
       current = current->next;
    }

    /* if we get to this line, the caller was asking
       for a non-existent element so we assert fail */
    assert(0);
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
     1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    /* Check the count function */
    printf("Element at index 3 is %d", GetNth(head, 3));
    getchar();
}
```

Time Complexity: O(n)

Source

http://www.geeksforgeeks.org/write-a-function-to-get-nth-node-in-a-linked-list/

Category: Linked Lists Tags: GetNth, Linked Lists

Chapter 5

Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?

A **simple solution** is to traverse the linked list until you find the node you want to delete. But this solution requires pointer to the head node which contradicts the problem statement.

Fast solution is to copy the data from the next node to the node to be deleted and delete the next node. Something like following.

```
    struct node *temp  = node_ptr->next;
    node_ptr->data  = temp->data;
    node_ptr->next  = temp->next;
    free(temp);
```

Program:

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
    of a list and an int, push a new node on the front
    of the list. */
void push(struct node** head_ref, int new_data)
{
   /* allocate node */
```

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

   /* put in the data  */
   new_node->data  = new_data;

   /* link the old list off the new node */
   new_node->next = (*head_ref);

   /* move the head to point to the new node */
   (*head_ref)    = new_node;
}

void printList(struct node *head)
{
   struct node *temp = head;
   while(temp != NULL)
   {
      printf("%d  ", temp->data);
      temp = temp->next;
   }
}

void deleteNode(struct node *node_ptr)
{
   struct node *temp = node_ptr->next;
   node_ptr->data    = temp->data;
   node_ptr->next    = temp->next;
   free(temp);
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
    1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    printf("\n Before deleting \n");
    printList(head);

    /* I m deleting the head itself.
        You can check for more cases */
    deleteNode(head);

    printf("\n After deleting \n");
    printList(head);
```

```
    getchar();
    return 0;
}
```

This solution doesn't work if the node to be deleted is the last node of the list. To make this solution work we can mark the end node as a dummy node. But the programs/functions that are using this function should also be modified.

You can try this problem for doubly linked list.

Source

http://www.geeksforgeeks.org/given-only-a-pointer-to-a-node-to-be-deleted-in-a-singly-linked-list-how-do-you-delete-it/

Category: Linked Lists Tags: Linked Lists, loop

Chapter 6

Write a C function to print the middle of a given linked list

Method 1:
Traverse the whole linked list and count the no. of nodes. Now traverse the list again till count/2 and return the node at count/2.

Method 2:
Traverse linked list using two pointers. Move one pointer by one and other pointer by two. When the fast pointer reaches end slow pointer will reach middle of the linked list.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to get the middle of the linked list*/
void printMiddle(struct node *head)
{
    struct node *slow_ptr = head;
    struct node *fast_ptr = head;

    if (head!=NULL)
    {
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;
            slow_ptr = slow_ptr->next;
        }
        printf("The middle element is [%d]\n\n", slow_ptr->data);
    }
}
```

```
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%d->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    for (i=5; i>0; i--)
    {
        push(&head, i);
        printList(head);
        printMiddle(head);
    }

    return 0;
}
```

Output:

```
 5->NULL
The middle element is [5]

4->5->NULL
The middle element is [5]
```

```
3->4->5->NULL
The middle element is [4]

2->3->4->5->NULL
The middle element is [4]

1->2->3->4->5->NULL
The middle element is [3]
```

Method 3:
Initialize mid element as head and initialize a counter as 0. Traverse the list from head, while traversing increment the counter and change mid to mid->next whenever the counter is odd. So the mid will move only half of the total length of the list.
Thanks to Narendra Kangralkar for suggesting this method.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to get the middle of the linked list*/
void printMiddle(struct node *head)
{
    int count = 0;
    struct node *mid = head;

    while (head != NULL)
    {
        /* update mid, when 'count' is odd number */
        if (count & 1)
            mid = mid->next;

        ++count;
        head = head->next;
    }

    /* if empty list is provided */
    if (mid != NULL)
        printf("The middle element is [%d]\n\n", mid->data);
}


void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));
```

```
    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%d->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    for (i=5; i>0; i--)
    {
        push(&head, i);
        printList(head);
        printMiddle(head);
    }

    return 0;
}
```

Output:

```
5->NULL
The middle element is [5]

4->5->NULL
The middle element is [5]

3->4->5->NULL
The middle element is [4]

2->3->4->5->NULL
The middle element is [4]
```

```
1->2->3->4->5->NULL
The middle element is [3]
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/write-a-c-function-to-print-the-middle-of-the-linked-list/

Category: Linked Lists

Chapter 7

Nth node from the end of a Linked List

Given a Linked List and a number n, write a function that returns the value at the nth node from end of the Linked List.

Method 1 (Use length of linked list)
1) Calculate the length of Linked List. Let the length be len.
2) Print the (len – n + 1)th node from the begining of the Linked List.

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

/* Link list node */
struct node
{
  int data;
  struct node* next;
};

/* Function to get the nth node from the last of a linked list*/
void printNthFromLast(struct node* head, int n)
{
    int len = 0, i;
    struct node *temp = head;

    // 1) count the number of nodes in Linked List
    while (temp != NULL)
    {
        temp = temp->next;
        len++;
    }

    // check if value of n is not more than length of the linked list
    if (len < n)
      return;

    temp = head;
```

```
    // 2) get the (n-len+1)th node from the begining
    for (i = 1; i < len-n+1; i++)
       temp = temp->next;

    printf ("%d", temp->data);

    return;
}

void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  // create linked 35->15->4->20
  push(&head, 20);
  push(&head, 4);
  push(&head, 15);
  push(&head, 35);

  printNthFromLast(head, 5);
  getchar();
  return 0;
}
```

Following is a recursive C code for the same method. Thanks to Anuj Bansal for providing following code.

```
void printNthFromLast(struct node* head, int n)
{
    static int i = 0;
    if(head == NULL)
       return;
    printNthFromLast(head->next, n);
    if(++i == n)
```

```
       printf("%d", head->data);
}
```

Time Complexity: O(n) where n is the length of linked list.

Method 2 (Use two pointers)
Maintain two pointers – reference pointer and main pointer. Initialize both reference and main pointers to head. First move reference pointer to n nodes from head. Now move both pointers one by one until reference pointer reaches end. Now main pointer will point to nth node from the end. Return main pointer.

Implementation:

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

/* Link list node */
struct node
{
  int data;
  struct node* next;
};

/* Function to get the nth node from the last of a linked list*/
void printNthFromLast(struct node *head, int n)
{
  struct node *main_ptr = head;
  struct node *ref_ptr = head;

  int count = 0;
  if(head != NULL)
  {
     while( count < n )
     {
        if(ref_ptr == NULL)
        {
           printf("%d is greater than the no. of "
                    "nodes in list", n);
           return;
        }
        ref_ptr = ref_ptr->next;
        count++;
     } /* End of while*/

     while(ref_ptr != NULL)
     {
        main_ptr = main_ptr->next;
        ref_ptr  = ref_ptr->next;
     }
     printf("Node no. %d from last is %d ",
               n, main_ptr->data);
  }
}
```

```
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;
  push(&head, 20);
  push(&head, 4);
  push(&head, 15);

  printNthFromLast(head, 3);
  getchar();
}
```

Time Complexity: O(n) where n is the length of linked list.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/nth-node-from-the-end-of-a-linked-list/

Category: Linked Lists Tags: Linked Lists

Chapter 8

Write a function to delete a Linked List

Algorithm: Iterate through the linked list and delete all the nodes one by one. Main point here is not to access next of the current pointer if current pointer is deleted.

Implementation:

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to delete the entire linked list */
void deleteList(struct node** head_ref)
{
   /* deref head_ref to get the real head */
   struct node* current = *head_ref;
   struct node* next;

   while (current != NULL)
   {
       next = current->next;
       free(current);
       current = next;
   }

   /* deref head_ref to affect the real head back
      in the caller. */
   *head_ref = NULL;
}
```

```
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
     1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    printf("\n Deleting linked list");
    deleteList(&head);

    printf("\n Linked list deleted");
    getchar();
}
```

Time Complexity: O(n)
Space Complexity: O(1)

Source

http://www.geeksforgeeks.org/write-a-function-to-delete-a-linked-list/

Category: Linked Lists Tags: Delete a Linked List, Linked Lists

Chapter 9

Write a function that counts the number of times a given int occurs in a Linked List

Here is a solution.

Algorithm:

```
1. Initialize count as zero.
2. Loop through each element of linked list:
     a) If element data is equal to the passed number then
        increment the count.
3. Return count.
```

Implementation:

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
```

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

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Counts the no. of occurences of a node
   (search_for) in a linked list (head)*/
int count(struct node* head, int search_for)
{
    struct node* current = head;
    int count = 0;
    while (current != NULL)
    {
        if (current->data == search_for)
           count++;
        current = current->next;
    }
    return count;
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
     1->2->1->3->1  */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    printf("count of 1 is %d", count(head, 1));
    getchar();
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Source

http://www.geeksforgeeks.org/write-a-function-that-counts-the-number-of-times-a-given-int-occurs-in-a-linked-list/

Category: Linked Lists Tags: Linked Lists

Chapter 10

Write a function to reverse a linked list

Iterative Method
Iterate trough the linked list. In loop, change next to prev, prev to current and current to next.

Implementation of Iterative Method

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to reverse the linked list */
static void reverse(struct node** head_ref)
{
    struct node* prev   = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
```

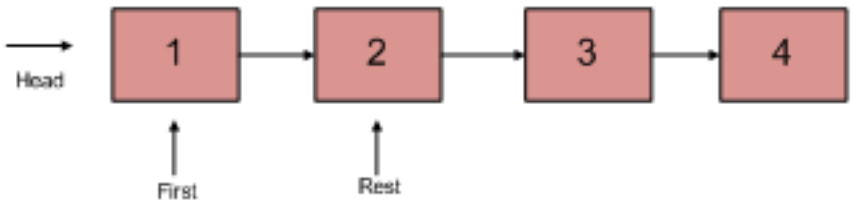
```
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d  ", temp->data);
        temp = temp->next;
    }
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

     push(&head, 20);
     push(&head, 4);
     push(&head, 15);
     push(&head, 85);

     printList(head);
     reverse(&head);
     printf("\n Reversed Linked list \n");
     printList(head);
     getchar();
}
```

Time Complexity: O(n)
Space Complexity: O(1)

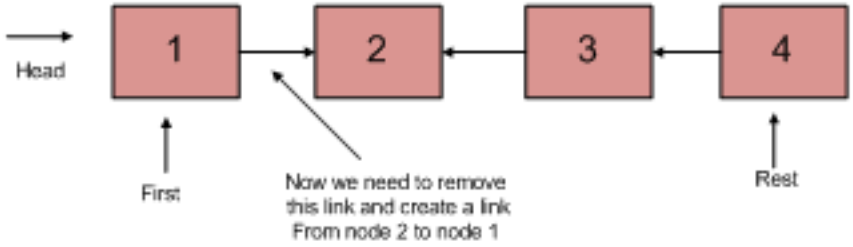
Recursive Method:

```
   1) Divide the list in two parts - first node and rest of the linked list.
   2) Call reverse for the rest of the linked list.
   3) Link rest to first.
   4) Fix head pointer
```

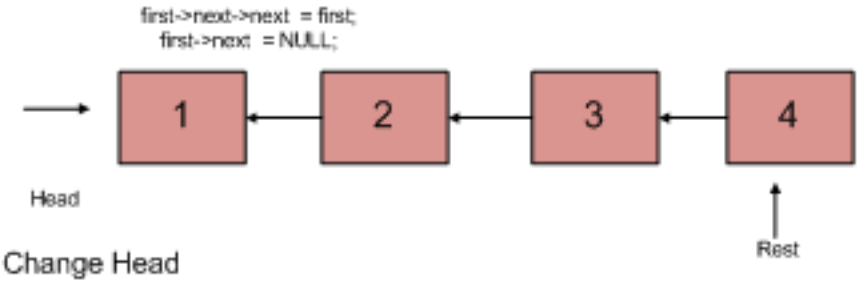
Divide the List in two parts



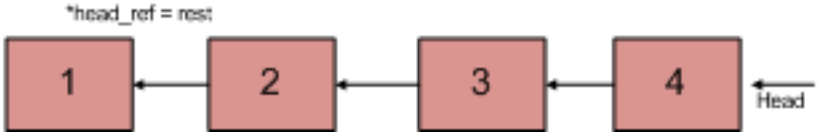
Reverse Rest



Link Rest to First



Change Head




```
void recursiveReverse(struct node** head_ref)
{
    struct node* first;
    struct node* rest;

    /* empty list */
    if (*head_ref == NULL)
       return;

    /* suppose first = {1, 2, 3}, rest = {2, 3} */
    first = *head_ref;
    rest  = first->next;

    /* List has only one node */
    if (rest == NULL)
       return;
```

```
    /* reverse the rest list and put the first element at the end */
    recursiveReverse(&rest);
    first->next->next  = first;

    /* tricky step -- see the diagram */
    first->next  = NULL;

    /* fix the head pointer */
    *head_ref = rest;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

A Simpler and Tail Recursive Method
Below is C++ implementation of this method.

```
// A simple and tail recursive C++ program to reverse
// a linked list
#include<bits/stdc++.h>
using namespace std;

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

void reverseUtil(node *curr, node *prev, node **head);

// This function mainly calls reverseUtil()
// with prev as NULL
void reverse(node **head)
{
    if (!head)
        return;
    reverseUtil(*head, NULL, head);
}

// A simple and tail recursive function to reverse
// a linked list.  prev is passed as NULL initially.
void reverseUtil(node *curr, node *prev, node **head)
{
    /* If last node mark it head*/
    if (!curr->next)
    {
        *head = curr;

        /* Update next to prev node */
        curr->next = prev;
        return;
    }
```

```
    /* Save curr->next node for recursive call */
    node *next = curr->next;

    /* and update next ..*/
    curr->next = prev;

    reverseUtil(next, curr, head);
}

// A utility function to create a new node
node *newNode(int key)
{
    node *temp = new node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printlist(node *head)
{
    while(head != NULL)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

// Driver program to test above functions
int main()
{
    node *head1 = newNode(1);
    head1->next = newNode(2);
    head1->next->next = newNode(2);
    head1->next->next->next = newNode(4);
    head1->next->next->next->next = newNode(5);
    head1->next->next->next->next->next = newNode(6);
    head1->next->next->next->next->next->next = newNode(7);
    head1->next->next->next->next->next->next->next = newNode(8);
    cout << "Given linked list\n";
    printlist(head1);
    reverse(&head1);
    cout << "\nReversed linked list\n";
    printlist(head1);
    return 0;
}
```

Output:

```
 Given linked list
1 2 2 4 5 6 7 8
```

```
Reversed linked list
8 7 6 5 4 2 2 1
```

Thanks to Gaurav Ahirwar for suggesting this solution.

References:
http://cslibrary.stanford.edu/105/LinkedListProblems.pdf

Source

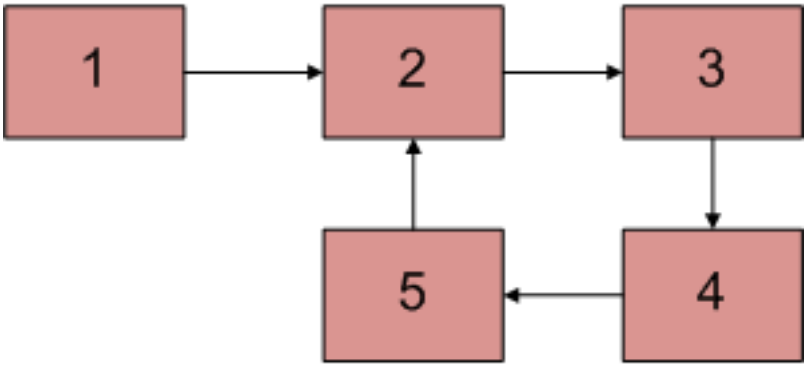
http://www.geeksforgeeks.org/write-a-function-to-reverse-the-nodes-of-a-linked-list/

Category: Linked Lists

Chapter 11

Write a C function to detect loop in a linked list

Below diagram shows a linked list with a loop



Following are different ways of doing this
Use Hashing:
Traverse the list one by one and keep putting the node addresses in a Hash Table. At any point, if NULL is reached then return false and if next of current node points to any of the previously stored nodes in Hash then return true.

Mark Visited Nodes:
This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the linked list and keep marking visited nodes. If you see a visited node again then there is a loop. This solution works in O(n) but requires additional information with each node.
A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Just store the addresses of visited nodes in a hash and if you see an address that already exists in hash then there is a loop.

Floyd's Cycle-Finding Algorithm:
This is the fastest method. Traverse linked list using two pointers. Move one pointer by one and other pointer by two. If these pointers meet at some node then there is a loop. If pointers do not meet then linked list doesn't have loop.

Implementation of Floyd's Cycle-Finding Algorithm:

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

/* Link list node */
struct node
{
  int data;
  struct node* next;
};

void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
  (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

int detectloop(struct node *list)
{
  struct node  *slow_p = list, *fast_p = list;

  while(slow_p && fast_p &&
          fast_p->next )
  {
    slow_p = slow_p->next;
    fast_p  = fast_p->next->next;
    if (slow_p == fast_p)
    {
       printf("Found Loop");
       return 1;
    }
  }
  return 0;
}

/* Drier program to test above function*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  push(&head, 20);
  push(&head, 4);
  push(&head, 15);
  push(&head, 10);
```

```
  /* Create a loop for testing */
  head->next->next->next->next = head;
  detectloop(head);

  getchar();
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)

References:
http://en.wikipedia.org/wiki/Cycle_detection
http://ostermiller.org/find_loop_singly_linked_list.html

Source

http://www.geeksforgeeks.org/write-a-c-function-to-detect-loop-in-a-linked-list/

Category: Linked Lists Tags: Linked Lists, loop

Chapter 12

Function to check if a singly linked list is palindrome

Given a singly linked list of characters, write a function that returns true if the given list is palindrome, else false.

METHOD 1 (Use a Stack)
A simple solution is to use a stack of list nodes. This mainly involves three steps.
1) Traverse the given list from head to tail and push every visited node to stack.
2) Traverse the list again. For every visited node, pop a node from stack and compare data of popped node with currently visited node.
3) If all nodes matched, then return true, else false.

Time complexity of above method is O(n), but it requires O(n) extra space. Following methods solve this with constant extra space.

METHOD 2 (By reversing the list)
This method takes O(n) time and O(1) extra space.
1) Get the middle of the linked list.
2) Reverse the second half of the linked list.
3) Check if the first half and second half are identical.
4) Construct the original linked list by reversing the second half again and attaching it back to the first half

To divide the list in two halves, method 2 of thispost is used.
When number of nodes are even, the first and second half contain exactly half nodes. The challenging thing in this method is to handle the case when number of nodes are odd. We don't want the middle node as part

of any of the lists as we are going to compare them for equality. For odd case, we use a separate variable 'midnode'.

```
/* Program to check if a linked list is palindrome */
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Link list node */
struct node
{
    char data;
    struct node* next;
};

void reverse(struct node**);
bool compareLists(struct node*, struct node *);

/* Function to check if given linked list is
  palindrome or not */
bool isPalindrome(struct node *head)
{
    struct node *slow_ptr = head, *fast_ptr = head;
    struct node *second_half, *prev_of_slow_ptr = head;
    struct node *midnode = NULL;  // To handle odd size list
    bool res = true; // initialize result

    if (head!=NULL && head->next!=NULL)
    {
        /* Get the middle of the list. Move slow_ptr by 1
          and fast_ptrr by 2, slow_ptr will have the middle
          node */
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;

            /*We need previous of the slow_ptr for
             linked lists  with odd elements */
            prev_of_slow_ptr = slow_ptr;
            slow_ptr = slow_ptr->next;
        }


        /* fast_ptr would become NULL when there are even elements in list.
           And not NULL for odd elements. We need to skip the middle node
           for odd case and store it somewhere so that we can restore the
           original list*/
        if (fast_ptr != NULL)
        {
            midnode = slow_ptr;
            slow_ptr = slow_ptr->next;
        }
```

```
        // Now reverse the second half and compare it with first half
        second_half = slow_ptr;
        prev_of_slow_ptr->next = NULL; // NULL terminate first half
        reverse(&second_half);  // Reverse the second half
        res = compareLists(head, second_half); // compare

        /* Construct the original list back */
         reverse(&second_half); // Reverse the second half again
         if (midnode != NULL)  // If there was a mid node (odd size case) which
                               // was not part of either first half or second half.
         {
            prev_of_slow_ptr->next = midnode;
            midnode->next = second_half;
         }
         else  prev_of_slow_ptr->next = second_half;
    }
    return res;
}

/* Function to reverse the linked list  Note that this
    function may change the head */
void reverse(struct node** head_ref)
{
    struct node* prev   = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to check if two input lists have same data*/
bool compareLists(struct node* head1, struct node *head2)
{
    struct node* temp1 = head1;
    struct node* temp2 = head2;

    while (temp1 && temp2)
    {
        if (temp1->data == temp2->data)
        {
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        else return 0;
    }

    /* Both are empty reurn 1*/
    if (temp1 == NULL && temp2 == NULL)
```

```
        return 1;

    /* Will reach here when one is NULL
      and other is not */
    return 0;
}

/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref)    = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%c->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}


/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    char str[] = "abacaba";
    int i;

    for (i = 0; str[i] != '\0'; i++)
    {
       push(&head, str[i]);
       printList(head);
       isPalindrome(head)? printf("Is Palindrome\n\n"):
                           printf("Not Palindrome\n\n");
    }

    return 0;
```

```
}
```

Output:

```
 a->NULL
Palindrome

b->a->NULL
Not Palindrome

a->b->a->NULL
Is Palindrome

c->a->b->a->NULL
Not Palindrome

a->c->a->b->a->NULL
Not Palindrome

b->a->c->a->b->a->NULL
Not Palindrome

a->b->a->c->a->b->a->NULL
Is Palindrome
```

Time Complexity O(n)
Auxiliary Space: O(1)

METHOD 3 (Using Recursion)
Use two pointers left and right. Move right and left using recursion and check for following in each recursive call.
1) Sub-list is palindrome.
2) Value at current left and right are matching.

If both above conditions are true then return true.

The idea is to use function call stack as container. Recursively traverse till the end of list. When we return from last NULL, we will be at last node. The last node to be compared with first node of list.

In order to access first node of list, we need list head to be available in the last call of recursion. Hence we pass head also to the recursive function. If they both match we need to compare (2, n-2) nodes. Again when recursion falls back to (n-2)nd node, we need reference to 2nd node from head. We advance the head pointer in previous call, to refer to next node in the list.

However, the trick in identifying double pointer. Passing single pointer is as good as pass-by-value, and we will pass the same pointer again and again. We need to pass the address of head pointer for reflecting the changes in parent recursive calls.

Thanks to Sharad Chandra for suggesting this approach.

```
// Recursive program to check if a given linked list is palindrome
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

```
/* Link list node */
struct node
{
    char data;
    struct node* next;
};

// Initial parameters to this function are &head and head
bool isPalindromeUtil(struct node **left, struct  node *right)
{
   /* stop recursion when right becomes NULL */
   if (right == NULL)
      return true;

   /* If sub-list is not palindrome then no need to
       check for current left and right, return false */
   bool isp = isPalindromeUtil(left, right->next);
   if (isp == false)
      return false;

   /* Check values at current left and right */
   bool isp1 = (right->data == (*left)->data);

   /* Move left to next node */
   *left = (*left)->next;

   return isp1;
}

// A wrapper over isPalindromeUtil()
bool isPalindrome(struct node *head)
{
   isPalindromeUtil(&head, head);
}

/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref)    = new_node;
}
```

```
// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%c->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    char str[] = "abacaba";
    int i;

    for (i = 0; str[i] != '\0'; i++)
    {
       push(&head, str[i]);
       printList(head);
       isPalindrome(head)? printf("Is Palindrome\n\n"):
                           printf("Not Palindrome\n\n");
    }

    return 0;
}
```

Output:

```
 a->NULL
Not Palindrome

b->a->NULL
Not Palindrome

a->b->a->NULL
Is Palindrome

c->a->b->a->NULL
Not Palindrome

a->c->a->b->a->NULL
Not Palindrome

b->a->c->a->b->a->NULL
Not Palindrome

a->b->a->c->a->b->a->NULL
Is Palindrome
```

Time Complexity: O(n)
Auxiliary Space: O(n) if Function Call Stack size is considered, otherwise O(1).

Please comment if you find any bug in the programs/algorithms or a better way to do the same.

Source

http://www.geeksforgeeks.org/function-to-check-if-a-singly-linked-list-is-palindrome/

Chapter 13

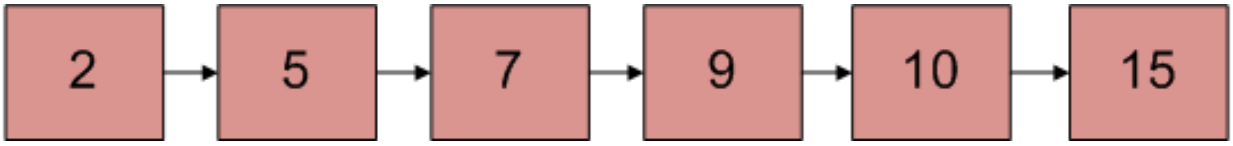
Given a linked list which is sorted, how will you insert in sorted way

Algorithm:
Let input linked list is sorted in increasing order.

```
1) If Linked list is empty then make the node as head and return it.
2) If value of the node to be inserted is smaller than value of head node
    then insert the node at start and make it head.
3) In a loop, find the appropriate node after which the input node (let 9) is
    to be inserted. To find the appropriate node start from head, keep moving
    until you reach a node GN (10 in the below diagram) who's value is
    greater than the input node. The node just before GN is the appropriate
    node (7).
4) Insert the node (9) after the appropriate node (7) found in step 3.
```

Initial Linked List

Linked List after insertion of 9



Implementation:

```
/* Program to insert in a sorted list */
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function to insert a new_node in a list. Note that this
  function expects a pointer to head_ref as this can modify the
  head of the input linked list (similar to push())*/
void sortedInsert(struct node** head_ref, struct node* new_node)
{
    struct node* current;
    /* Special case for the head end */
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        current = *head_ref;
        while (current->next!=NULL &&
               current->next->data < new_node->data)
        {
            current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST sortedInsert */

/* A utility function to create a new node */
struct node *newNode(int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;
    new_node->next =  NULL;

    return new_node;
}

/* Function to print linked list */
```

```
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d  ", temp->data);
        temp = temp->next;
    }
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    struct node *new_node = newNode(5);
    sortedInsert(&head, new_node);
    new_node = newNode(10);
    sortedInsert(&head, new_node);
    new_node = newNode(7);
    sortedInsert(&head, new_node);
    new_node = newNode(3);
    sortedInsert(&head, new_node);
    new_node = newNode(1);
    sortedInsert(&head, new_node);
    new_node = newNode(9);
    sortedInsert(&head, new_node);
    printf("\n Created Linked List\n");
    printList(head);

    getchar();
    return 0;
}
```

Shorter Implementation using double pointers
Thanks to Murat M Ozturk for providing this solution. Please see Murat M Ozturk's comment below for complete function. The code uses double pointer to keep track of the next pointer of the previous node (after which new node is being inserted).

Note that below line in code changes *current* to have address of next pointer in a node.

```
current = &((*current)->next);
```

Also, note below comments.

```
new_node->next = *current; /* Copies the value-at-address current to new_node's next pointer*/

*current = new_node;  /* Fix next pointer of the node (using it's address) after which new_node is being in
```

Time Complexity: O(n)

References:
http://cslibrary.stanford.edu/105/LinkedListProblems.pdf

Source

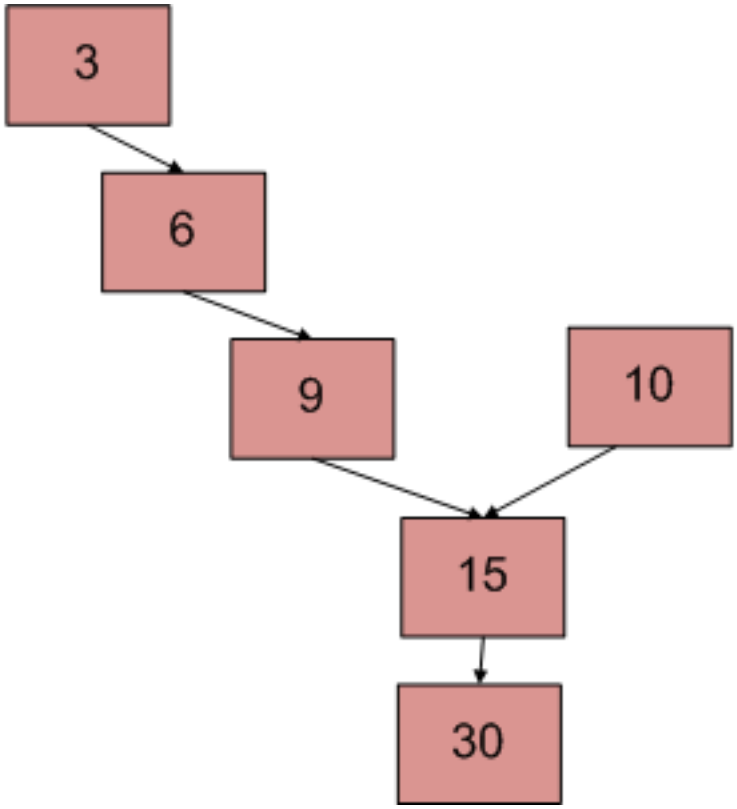
http://www.geeksforgeeks.org/given-a-linked-list-which-is-sorted-how-will-you-insert-in-sorted-way/

Category: Linked Lists

Chapter 14

Write a function to get the intersection point of two Linked Lists.

There are two singly linked lists in a system. By some programming error the end node of one of the linked list got linked into the second list, forming a inverted Y shaped list. Write a program to get the point where two linked list merge.



Above diagram shows an example with two linked list having 15 as intersection point.

Method 1(Simply use two loops)
Use 2 nested for loops. Outer loop will be for each node of the 1st list and inner loop will be for 2nd list. In the inner loop, check if any of nodes of 2nd list is same as the current node of first linked list. Time complexity of this method will be O(mn) where m and n are the number of nodes in two lists.

Method 2 (Mark Visited Nodes)

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the first linked list and keep marking visited nodes. Now traverse second linked list, If you see a visited node again then there is an intersection point, return the intersecting node. This solution works in O(m+n) but requires additional information with each node. A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Traverse the first linked list and store the addresses of visited nodes in a hash. Now traverse the second linked list and if you see an address that already exists in hash then return the intersecting node.

Method 3(Using difference of node counts)
1) Get count of the nodes in first list, let count be c1.
2) Get count of the nodes in second list, let count be c2.
3) Get the difference of counts d = abs(c1 – c2)
4) Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes.
5) Then we can traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

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

/* Link list node */
struct node
{
  int data;
  struct node* next;
};

/* Function to get the counts of node in a linked list */
int getCount(struct node* head);

/* function to get the intersection point of two linked
   lists head1 and head2 where head1 has d more nodes than
   head2 */
int _getIntesectionNode(int d, struct node* head1, struct node* head2);

/* function to get the intersection point of two linked
   lists head1 and head2 */
int getIntesectionNode(struct node* head1, struct node* head2)
{
  int c1 = getCount(head1);
  int c2 = getCount(head2);
  int d;

  if(c1 > c2)
  {
    d = c1 - c2;
    return _getIntesectionNode(d, head1, head2);
  }
  else
  {
    d = c2 - c1;
    return _getIntesectionNode(d, head2, head1);
  }
}
```

```
/* function to get the intersection point of two linked
   lists head1 and head2 where head1 has d more nodes than
   head2 */
int _getIntesectionNode(int d, struct node* head1, struct node* head2)
{
  int i;
  struct node* current1 = head1;
  struct node* current2 = head2;

  for(i = 0; i < d; i++)
  {
    if(current1 == NULL)
    {  return -1; }
    current1 = current1->next;
  }

  while(current1 !=  NULL && current2 != NULL)
  {
    if(current1 == current2)
      return current1->data;
    current1= current1->next;
    current2= current2->next;
  }

  return -1;
}

/* Takes head pointer of the linked list and
   returns the count of nodes in the list */
int getCount(struct node* head)
{
  struct node* current = head;
  int count = 0;

  while (current != NULL)
  {
    count++;
    current = current->next;
  }

  return count;
}

/* IGNORE THE BELOW LINES OF CODE. THESE LINES
   ARE JUST TO QUICKLY TEST THE ABOVE FUNCTION */
int main()
{
  /*
    Create two linked lists

    1st 3->6->9->15->30
    2nd 10->15->30
```

```
    15 is the intersection point
  */

  struct node* newNode;
  struct node* head1 =
            (struct node*) malloc(sizeof(struct node));
  head1->data  = 10;

  struct node* head2 =
            (struct node*) malloc(sizeof(struct node));
  head2->data  = 3;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 6;
  head2->next = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 9;
  head2->next->next = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 15;
  head1->next = newNode;
  head2->next->next->next  = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 30;
  head1->next->next= newNode;

  head1->next->next->next = NULL;

  printf("\n The node of intersection is %d \n",
          getIntesectionNode(head1, head2));

  getchar();
}
```

Time Complexity: O(m+n)
Auxiliary Space: O(1)

Method 4(Make circle in first list)
Thanks to Saravanan Man for providing below solution.
1. Traverse the first linked list(count the elements) and make a circular linked list. (Remember last node so that we can break the circle later on).
2. Now view the problem as find the loop in the second linked list. So the problem is solved.
3. Since we already know the length of the loop(size of first linked list) we can traverse those many number of nodes in second list, and then start another pointer from the beginning of second list. we have to traverse until they are equal, and that is the required intersection point.
4. remove the circle from the linked list.

Time Complexity: O(m+n)
Auxiliary Space: O(1)

Method 5 (Reverse the first list and make equations)

Thanks to Saravanan Mani for providing this method.

```
1) Let X be the length of the first linked list until intersection point.
   Let Y be the length of the second linked list until the intersection point.
   Let Z be the length of the linked list from intersection point to End of
   the linked list including the intersection node.
   We Have
           X + Z = C1;
           Y + Z = C2;
2) Reverse first linked list.
3) Traverse Second linked list. Let C3 be the length of second list - 1.
     Now we have
        X + Y = C3
     We have 3 linear equations. By solving them, we get
       X = (C1 + C3 - C2)/2;
       Y = (C2 + C3 - C1)/2;
       Z = (C1 + C2 - C3)/2;
      WE GOT THE INTERSECTION POINT.
4)  Reverse first linked list.
```

Advantage: No Comparison of pointers.
Disadvantage : Modifying linked list(Reversing list).

Time complexity: O(m+n)
Auxiliary Space: O(1)

Method 6 (Traverse both lists and compare addresses of last nodes) This method is only to detect if there is an intersection point or not. (Thanks to NeoTheSaviour for suggesting this)

```
1) Traverse the list 1, store the last node address
2) Traverse the list 2, store the last node address.
3) If nodes stored in 1 and 2 are same then they are intersecting.
```

Time complexity of this method is O(m+n) and used Auxiliary space is O(1)

Method 7 (Use Hashing)
Basically we need to find common node of two linked lists. So we hash all nodes of first list and then check second list.
1) Create an empty hash table such that node address is used as key and a binary value present/absent is used as value.
2) Traverse the first linked list and insert all nodes' addresses in hash table.
3) Traverse the second list. For every node check if it is present in hash table. If we find a node in hash table, return the node.

Please write comments if you find any bug in the above algorithm or a better way to solve the same problem.

Source

http://www.geeksforgeeks.org/write-a-function-to-get-the-intersection-point-of-two-linked-lists/

Chapter 15

Write a recursive function to print reverse of a Linked List

Note that the question is only about printing the reverse. To reverse the list itself see this

Difficulty Level: Rookie

Algorithm

```
printReverse(head)
  1. call print reverse for hed->next
  2. print head->data
```

Implementation:

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to reverse the linked list */
void printReverse(struct node* head)
{
  // Base case
  if(head == NULL)
    return;

  // print the list after head node
  printReverse(head->next);
```

```
  // After everything else is printed, print head
  printf("%d  ", head->data);
}

/*UTILITY FUNCTIONS*/
/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{

  struct node* head = NULL;

  push(&head, 1);
  push(&head, 2);
  push(&head, 3);
  push(&head, 4);

  printReverse(head);
  getchar();
}
```

Time Complexity: O(n)

Source

http://www.geeksforgeeks.org/write-a-recursive-function-to-print-reverse-of-a-linked-list/

Category: Linked Lists

Chapter 16

Remove duplicates from a sorted linked list

Write a removeDuplicates() function which takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60.

Algorithm:
Traverse the list from the head (or start) node. While traversing, compare each node with its next node. If data of next node is same as current node then delete the next node. Before we delete a node, we need to store next pointer of the node

Implementation:
Functions other than removeDuplicates() are just to create a linked linked list and test removeDuplicates().

```
/*Program to remove duplicates from a sorted linked list */
#include<stdio.h>
#include<stlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* The function removes duplicates from a sorted list */
void removeDuplicates(struct node* head)
{
  /* Pointer to traverse the linked list */
  struct node* current = head;

  /* Pointer to store the next pointer of a node to be deleted*/
  struct node* next_next;

  /* do nothing if the list is empty */
  if(current == NULL)
```

```
     return;

  /* Traverse the list till last node */
  while(current->next != NULL)
  {
   /* Compare current node with next node */
    if(current->data == current->next->data)
    {
       /*The sequence of steps is important*/
      next_next = current->next->next;
      free(current->next);
      current->next = next_next;
    }
    else /* This is tricky: only advance if no deletion */
    {
      current = current->next;
    }
  }
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
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;
```

```
  /* Let us create a sorted linked list to test the functions
   Created linked list will be 11->11->11->13->13->20 */
  push(&head, 20);
  push(&head, 13);
  push(&head, 13);
  push(&head, 11);
  push(&head, 11);
  push(&head, 11);

  printf("\n Linked list before duplicate removal  ");
  printList(head);

  /* Remove duplicates from linked list */
  removeDuplicates(head);

  printf("\n Linked list after duplicate removal ");
  printList(head);

  getchar();
}
```

Time Complexity: O(n) where n is number of nodes in the given linked list.

References:

cslibrary.stanford.edu/105/LinkedListProblems.pdf

Source

http://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/

Category: Linked Lists

Chapter 17

Remove duplicates from an unsorted linked list

Write a removeDuplicates() function which takes a list and deletes any duplicate nodes from the list. The list is not sorted.

For example if the linked list is 12->11->12->21->41->43->21 then removeDuplicates() should convert the list to 12->11->21->41->43.

METHOD 1 (Using two loops)
This is the simple way where two loops are used. Outer loop is used to pick the elements one by one and inner loop compares the picked element with rest of the elements.

Thanks to Gaurav Saxena for his help in writing this code.

```
/* Program to remove duplicates in an unsorted array */

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

/* A linked list node */
struct node
{
 int data;
 struct node *next;
};

/* Function to remove duplicates from a unsorted linked list */
void removeDuplicates(struct node *start)
{
  struct node *ptr1, *ptr2, *dup;
  ptr1 = start;

  /* Pick elements one by one */
  while(ptr1 != NULL && ptr1->next != NULL)
  {
     ptr2 = ptr1;

     /* Compare the picked element with rest of the elements */
```

```
      while(ptr2->next != NULL)
      {
        /* If duplicate then delete it */
        if(ptr1->data == ptr2->next->data)
        {
           /* sequence of steps is important here */
           dup = ptr2->next;
           ptr2->next = ptr2->next->next;
           free(dup);
        }
        else /* This is tricky */
        {
           ptr2 = ptr2->next;
        }
      }
      ptr1 = ptr1->next;
  }
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data);

/* Function to print nodes in a given linked list */
void printList(struct node *node);

/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   10->12->11->11->12->11->10*/
  push(&start, 10);
  push(&start, 11);
  push(&start, 12);
  push(&start, 11);
  push(&start, 11);
  push(&start, 12);
  push(&start, 10);

  printf("\n Linked list before removing duplicates ");
  printList(start);

  removeDuplicates(start);

  printf("\n Linked list after removing duplicates ");
  printList(start);

  getchar();
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
```

```
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}
```

Time Complexity: O(n^2)

METHOD 2 (Use Sorting)
In general, Merge Sort is the best suited sorting algorithm for sorting linked lists efficiently.
1) Sort the elements using Merge Sort. We will soon be writing a post about sorting a linked list. O(nLogn)
2) Remove duplicates in linear time using the algorithm for removing duplicates in sorted Linked List. O(n)

Please note that this method doesn't preserve the original order of elements.

Time Complexity: O(nLogn)

METHOD 3 (Use Hashing)
We traverse the link list from head to end. For every newly encountered element, we check whether it is in the hash table: if yes, we remove it; otherwise we put it in the hash table.

Thanks to bearwang for suggesting this method.

Time Complexity: O(n) on average (assuming that hash table access time is O(1) on average).

Please write comments if you find any of the above explanations/algorithms incorrect, or a better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/remove-duplicates-from-an-unsorted-linked-list/

Category: Linked Lists

Chapter 18

Pairwise swap elements of a given linked list

Given a singly linked list, write a function to swap elements pairwise. For example, if the linked list is 1->2->3->4->5 then the function should change it to 2->1->4->3->5, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5.

METHOD 1 (Iterative)
Start from the head node and traverse the list. While traversing swap data of each node with its next node's data.

```
/* Program to pairwise swap elements in a given linked list */
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
  int data;
  struct node *next;
};

/*Function to swap two integers at addresses a and b */
void swap(int *a, int *b);

/* Function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node *head)
{
  struct node *temp = head;

   /* Traverse further only if there are at-least two nodes left */
  while (temp != NULL && temp->next != NULL)
  {
    /* Swap data of node with its next node's data */
    swap(&temp->data, &temp->next->data);

    /* Move temp by 2 for the next pair */
    temp = temp->next->next;
```

```
  }
}

/* UTILITY FUNCTIONS */
/* Function to swap two integers */
void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while (node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   1->2->3->4->5 */
  push(&start, 5);
  push(&start, 4);
  push(&start, 3);
  push(&start, 2);
  push(&start, 1);

  printf("\n Linked list before calling  pairWiseSwap() ");
```

```
  printList(start);

  pairWiseSwap(start);

  printf("\n Linked list after calling  pairWiseSwap() ");
  printList(start);

  getchar();
  return 0;
}
```

Time complexity: O(n)

METHOD 2 (Recursive)
If there are 2 or more than 2 nodes in Linked List then swap the first two nodes and recursively call for rest of the list.

```
/* Recursive function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node *head)
{
  /* There must be at-least two nodes in the list */
  if(head != NULL && head->next != NULL)
  {
    /* Swap the node's data with data of next node */
    swap(&head->data, &head->next->data);

    /* Call pairWiseSwap() for rest of the list */
    pairWiseSwap(head->next->next);
  }
}
```

Time complexity: O(n)

The solution provided there swaps data of nodes. If data contains many fields, there will be many swap operations. See thisfor an implementation that changes links rather than swapping data.

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/pairwise-swap-elements-of-a-given-linked-list/

Category: Linked Lists

Chapter 19

Practice questions for Linked List and Recursion

Assume the structure of a Linked List node is as follows.

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

Explain the functionality of following C functions.

1. What does the following function do for a given Linked List?

```
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}
```

fun1() prints the given Linked List in reverse manner. For Linked List 1->2->3->4->5, fun1() prints 5->4->3->2->1.

2. What does the following function do for a given Linked List ?

```
void fun2(struct node* head)
{
  if(head== NULL)
    return;
```

```
  printf("%d  ", head->data);

  if(head->next != NULL )
    fun2(head->next->next);
  printf("%d  ", head->data);
}
```

fun2() prints alternate nodes of the given Linked List, first from head to end, and then from end to head. If Linked List has even number of nodes, then fun2() skips the last node. For Linked List 1->2->3->4->5, fun2() prints 1 3 5 5 3 1. For Linked List 1->2->3->4->5->6, fun2() prints 1 3 5 5 3 1.

Below is a complete running program to test above functions.

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

/* A linked list node */
struct node
{
  int data;
  struct node *next;
};


/* Prints a linked list in reverse manner */
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}

/* prints alternate nodes of a Linked List, first
  from head to end, and then from end to head. */
void fun2(struct node* start)
{
  if(start == NULL)
    return;
  printf("%d  ", start->data);

  if(start->next != NULL )
    fun2(start->next->next);
  printf("%d  ", start->data);
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
```

```
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above functions */
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  /* Using push() to construct below list
    1->2->3->4->5  */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);

  printf("\n Output of fun1() for list 1->2->3->4->5 \n");
  fun1(head);

  printf("\n Output of fun2() for list 1->2->3->4->5 \n");
  fun2(head);

  getchar();
  return 0;
}
```

Please write comments if you find any of the answers/explanations incorrect, or you want to share more information about the topics discussed above.

Source

http://www.geeksforgeeks.org/practice-questions-for-linked-list-and-recursion/

Category: Linked Lists Tags: Recursion

Chapter 20

Move last element to front of a given Linked List

Write a C function that moves last element to front in a given Singly Linked List. For example, if the given Linked List is 1->2->3->4->5, then the function should change the list to 5->1->2->3->4.

Algorithm:
Traverse the list till last node. Use two pointers: one to store the address of last node and other for address of second last node. After the end of loop do following operations.
i) Make second last as last (secLast->next = NULL).
ii) Set next of last as head (last->next = *head_ref).
iii) Make last as head (*head_ref = last)

```
/* Program to move last element to front in a given linked list */
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
 int data;
 struct node *next;
};

/* We are using a double pointer head_ref here because we change
   head of the linked list inside this function.*/
void moveToFront(struct node **head_ref)
{
  /* If linked list is empty, or it contains only one node,
    then nothing needs to be done, simply return */
  if(*head_ref == NULL || (*head_ref)->next == NULL)
    return;

  /* Initialize second last and last pointers */
  struct node *secLast = NULL;
  struct node *last = *head_ref;

  /*After this loop secLast contains address of second last
```

```
  node and last contains address of last node in Linked List */
  while(last->next != NULL)
  {
    secLast = last;
    last = last->next;
  }

  /* Set the next of second last as NULL */
  secLast->next = NULL;

  /* Set next of last as head node */
  last->next = *head_ref;

  /* Change the head pointer to point to last node now */
  *head_ref = last;
}

/* UTILITY FUNCTIONS */
/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}


/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   1->2->3->4->5 */
  push(&start, 5);
```

```
  push(&start, 4);
  push(&start, 3);
  push(&start, 2);
  push(&start, 1);

  printf("\n Linked list before moving last to front ");
  printList(start);

  moveToFront(&start);

  printf("\n Linked list after removing last to front ");
  printList(start);

  getchar();
}
```

Time Complexity: O(n) where n is the number of nodes in the given Linked List.

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/move-last-element-to-front-of-a-given-linked-list/

Category: Linked Lists

Chapter 21

Intersection of two Sorted Linked Lists

Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed.

For example, let the first linked list be 1->2->3->4->6 and second linked list be 2->4->6->8, then your function should create and return a third list as 2->4->6.

Method 1 (Using Dummy Node)
The strategy here uses a temporary dummy node as the start of the result list. The pointer tail always points to the last node in the result list, so appending new nodes is easy. The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either 'a' or 'b', and adding it to tail. When we are done, the result is in dummy.next.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

void push(struct node** head_ref, int new_data);

/*This solution uses the temporary dummy to build up the result list */
struct node* sortedIntersect(struct node* a, struct node* b)
{
  struct node dummy;
  struct node* tail = &dummy;
  dummy.next = NULL;

  /* Once one or the other list runs out -- we're done */
  while (a != NULL && b != NULL)
  {
```

```
    if (a->data == b->data)
    {
       push((&tail->next), a->data);
       tail = tail->next;
       a = a->next;
       b = b->next;
    }
    else if (a->data < b->data) /* advance the smaller list */
       a = a->next;
    else
       b = b->next;
  }
  return(dummy.next);
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
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 lists */
  struct node* a = NULL;
  struct node* b = NULL;
  struct node *intersect = NULL;

  /* Let us create the first sorted linked list to test the functions
   Created linked list will be 1->2->3->4->5->6 */
  push(&a, 6);
```

```
  push(&a, 5);
  push(&a, 4);
  push(&a, 3);
  push(&a, 2);
  push(&a, 1);

  /* Let us create the second sorted linked list
   Created linked list will be 2->4->6->8 */
  push(&b, 8);
  push(&b, 6);
  push(&b, 4);
  push(&b, 2);

  /* Find the intersection two linked lists */
  intersect = sortedIntersect(a, b);

  printf("\n Linked list containing common items of a & b \n ");
  printList(intersect);

  getchar();
}
```

Time Complexity: O(m+n) where m and n are number of nodes in first and second linked lists respectively.

Method 2 (Using Local References)
This solution is structurally very similar to the above, but it avoids using a dummy node Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** "reference" strategy can be used

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

void push(struct node** head_ref, int new_data);

/* This solution uses the local reference */
struct node* sortedIntersect(struct node* a, struct node* b)
{
  struct node* result = NULL;
  struct node** lastPtrRef = &result;

  /* Advance comparing the first nodes in both lists.
    When one or the other list runs out, we're done. */
  while (a!=NULL && b!=NULL)
  {
```

```
    if (a->data == b->data)
    {
      /* found a node for the intersection */
      push(lastPtrRef, a->data);
      lastPtrRef = &((*lastPtrRef)->next);
      a = a->next;
      b = b->next;
    }
    else if (a->data < b->data)
      a=a->next;       /* advance the smaller list */
    else
      b=b->next;
  }
  return(result);
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
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 lists */
  struct node* a = NULL;
  struct node* b = NULL;
  struct node *intersect = NULL;

  /* Let us create the first sorted linked list to test the functions
   Created linked list will be 1->2->3->4->5->6 */
```

```
  push(&a, 6);
  push(&a, 5);
  push(&a, 4);
  push(&a, 3);
  push(&a, 2);
  push(&a, 1);

  /* Let us create the second sorted linked list
   Created linked list will be 2->4->6->8 */
  push(&b, 8);
  push(&b, 6);
  push(&b, 4);
  push(&b, 2);

  /* Find the intersection two linked lists */
  intersect = sortedIntersect(a, b);

  printf("\n Linked list containing common items of a & b \n ");
  printList(intersect);

  getchar();
}
```

Time Complexity: O(m+n) where m and n are number of nodes in first and second linked lists respectively.

Method 3 (Recursive)
Below is the recursive implementation of sortedIntersect().

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

struct node *sortedIntersect(struct node *a, struct node *b)
{
    /* base case */
    if (a == NULL || b == NULL)
        return NULL;

    /* If both lists are non-empty */

    /* advance the smaller list and call recursively */
    if (a->data < b->data)
        return sortedIntersect(a->next, b);

    if (a->data > b->data)
        return sortedIntersect(a, b->next);
```

```
    // Below lines are executed only when a->data == b->data
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->data = a->data;

    /* advance both lists and call recursively */
    temp->next = sortedIntersect(a->next, b->next);
    return temp;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
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 lists */
    struct node* a = NULL;
    struct node* b = NULL;
    struct node *intersect = NULL;

    /* Let us create the first sorted linked list to test the functions
     Created linked list will be 1->2->3->4->5->6 */
    push(&a, 6);
    push(&a, 5);
    push(&a, 4);
    push(&a, 3);
    push(&a, 2);
    push(&a, 1);
```

```
    /* Let us create the second sorted linked list
     Created linked list will be 2->4->6->8 */
    push(&b, 8);
    push(&b, 6);
    push(&b, 4);
    push(&b, 2);

    /* Find the intersection two linked lists */
    intersect = sortedIntersect(a, b);

    printf("\n Linked list containing common items of a & b \n ");
    printList(intersect);

    return 0;
}
```

Time Complexity: O(m+n) where m and n are number of nodes in first and second linked lists respectively.

Please write comments if you find the above codes/algorithms incorrect, or find better ways to solve the same problem.

References:

cslibrary.stanford.edu/105/LinkedListProblems.pdf

Source

http://www.geeksforgeeks.org/intersection-of-two-sorted-linked-lists/

Category: Linked Lists

Post navigation

← Maximum width of a binary tree How are variables scoped in C – Static or Dynamic? →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 22

Delete alternate nodes of a Linked List

Given a Singly Linked List, starting from the second node delete all alternate nodes of it. For example, if the given linked list is 1->2->3->4->5 then your function should convert it to 1->3->5, and if the given linked list is 1->2->3->4 then convert it to 1->3.

Method 1 (Iterative)
Keep track of previous of the node to be deleted. First change the next link of previous node and then free the memory allocated for the node.

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

/* A linked list node */
struct node
{
  int data;
  struct node *next;
};

/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct node *head)
{
  if (head == NULL)
    return;

   /* Initialize prev and node to be deleted */
  struct node *prev = head;
  struct node *node = head->next;

  while (prev != NULL && node != NULL)
  {
     /* Change next link of previous node */
     prev->next = node->next;

     /* Free memory */
     free(node);
```

```
     /* Update prev and node */
     prev = prev->next;
     if(prev != NULL)
        node = prev->next;
  }
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Drier program to test above functions */
int main()
{
  int arr[100];

  /* Start with the empty list */
  struct node* head = NULL;

  /* Using push() to construct below list
    1->2->3->4->5  */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);
```

```
  printf("\n List before calling deleteAlt() ");
  printList(head);

  deleteAlt(head);

  printf("\n List after calling deleteAlt() ");
  printList(head);

  getchar();
  return 0;
}
```

Time Complexity: O(n) where n is the number of nodes in the given Linked List.

Method 2 (Recursive)
Recursive code uses the same approach as method 1. The recursive coed is simple and short, but causes O(n) recursive function calls for a linked list of size n.

```
/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct node *head)
{
  if (head == NULL)
    return;

  struct node *node = head->next;

  if (node == NULL)
    return;

  /* Change the next link of head */
  head->next = node->next;

  /* free memory allocated for node */
  free(node);

  /* Recursively call for the new next of head */
  deleteAlt(head->next);
}
```

Time Complexity: O(n)

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/delete-alternate-nodes-of-a-linked-list/

Category: Linked Lists

Chapter 23

Alternating split of a given Singly Linked List

Write a function AlternatingSplit() that takes one list and divides up its nodes to make two smaller lists 'a' and 'b'. The sublists should be made from alternating elements in the original list. So if the original list is 0->1->0->1->0->1 then one sublist should be 0->0->0 and the other should be 1->1->1.

Method 1(Simple)
The simplest approach iterates over the source list and pull nodes off the source and alternately put them at the front (or beginning) of 'a' and b'. The only strange part is that the nodes will be in the reverse order that they occurred in the source list. Method 2 inserts the node at the end by keeping track of last node in sublists.

```
/*Program to alternatively split a linked list into two halves */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* pull off the front node of the source and put it in dest */
void MoveNode(struct node** destRef, struct node** sourceRef) ;

/* Given the source list, split its nodes into two shorter lists.
  If we number the elements 0, 1, 2, ... then all the even elements
  should go in the first list, and all the odd elements in the second.
  The elements in the new lists may be in any order. */
void AlternatingSplit(struct node* source, struct node** aRef,
                            struct node** bRef)
{
  /* split the nodes of source to these 'a' and 'b' lists */
  struct node* a = NULL;
  struct node* b = NULL;
```

```
  struct node* current = source;
  while (current != NULL)
  {
    MoveNode(&a, &current); /* Move a node to list 'a' */
    if (current != NULL)
    {
       MoveNode(&b, &current); /* Move a node to list 'b' */
    }
  }
  *aRef = a;
  *bRef = b;
}

/* Take the node from the front of the source, and move it to the front of the dest.
   It is an error to call this with the source list empty.

   Before calling MoveNode():
   source == {1, 2, 3}
   dest == {1, 2, 3}

   Affter calling MoveNode():
   source == {2, 3}
   dest == {1, 1, 2, 3}
*/
void MoveNode(struct node** destRef, struct node** sourceRef)
{
  /* the front source node  */
  struct node* newNode = *sourceRef;
  assert(newNode != NULL);

  /* Advance the source pointer */
  *sourceRef = newNode->next;

  /* Link the old dest off the new node */
  newNode->next = *destRef;

  /* Move dest to point to the new node */
  *destRef = newNode;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);
```

```
  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
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;
  struct node* a = NULL;
  struct node* b = NULL;

  /* Let us create a sorted linked list to test the functions
   Created linked list will be 0->1->2->3->4->5 */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);
  push(&head, 0);

  printf("\n Original linked List:  ");
  printList(head);

  /* Remove duplicates from linked list */
  AlternatingSplit(head, &a, &b);

  printf("\n Resultant Linked List 'a' ");
  printList(a);

  printf("\n Resultant Linked List 'b' ");
  printList(b);

  getchar();
  return 0;
}
```

Time Complexity: O(n) where n is number of node in the given linked list.

Method 2(Using Dummy Nodes)
Here is an alternative approach which builds the sub-lists in the same order as the source list. The code uses a temporary dummy header nodes for the 'a' and 'b' lists as they are being built. Each sublist has a "tail" pointer which points to its current last node — that way new nodes can be appended to the end of

each list easily. The dummy nodes give the tail pointers something to point to initially. The dummy nodes are efficient in this case because they are temporary and allocated in the stack. Alternately, local "reference pointers" (which always points to the last pointer in the list instead of to the last node) could be used to avoid Dummy nodes.

```
void AlternatingSplit(struct node* source, struct node** aRef,
                              struct node** bRef)
{
  struct node aDummy;
  struct node* aTail = &aDummy; /* points to the last node in 'a' */
  struct node bDummy;
  struct node* bTail = &bDummy; /* points to the last node in 'b' */
  struct node* current = source;
  aDummy.next = NULL;
  bDummy.next = NULL;
  while (current != NULL)
  {
    MoveNode(&(aTail->next), &current); /* add at 'a' tail */
    aTail = aTail->next; /* advance the 'a' tail */
    if (current != NULL)
    {
      MoveNode(&(bTail->next), &current);
      bTail = bTail->next;
    }
  }
  *aRef = aDummy.next;
  *bRef = bDummy.next;
}
```

Time Complexity: O(n) where n is number of node in the given linked list.

Source: http://cslibrary.stanford.edu/105/LinkedListProblems.pdf

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/alternating-split-of-a-given-singly-linked-list/

Category: Linked Lists

Post navigation

← Delete alternate nodes of a Linked List Output of C Programs | Set 14 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 24

Merge two sorted linked lists

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list. The new list should be made by splicing
together the nodes of the first two lists.

For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20.

There are many cases to deal with: either 'a' or 'b' may be empty, during processing either 'a' or 'b' may run out first, and finally there's the problem of starting the result list empty, and building it up while going through 'a' and 'b'.

Method 1 (Using Dummy Nodes)
The strategy here uses a temporary dummy node as the start of the result list. The pointer Tail always points to the last node in the result list, so appending new nodes is easy.
The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either 'a' or 'b', and adding it to tail. When
we are done, the result is in dummy.next.

```
/*Program to alternatively split a linked list into two halves */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* pull off the front node of the source and put it in dest */
void MoveNode(struct node** destRef, struct node** sourceRef);

/* Takes two lists sorted in increasing order, and splices their nodes together to make one big sorted list wh
struct node* SortedMerge(struct node* a, struct node* b)
{
```

```
   /* a dummy first node to hang the result on */
   struct node dummy;

   /* tail points to the last result node  */
   struct node* tail = &dummy;

   /* so tail->next is the place to add new nodes
     to the result. */
   dummy.next = NULL;
   while(1)
   {
      if(a == NULL)
      {
         /* if either list runs out, use the other list */
         tail->next = b;
         break;
      }
      else if (b == NULL)
      {
         tail->next = a;
         break;
      }
      if (a->data <= b->data)
      {
         MoveNode(&(tail->next), &a);
      }
      else
     {
        MoveNode(&(tail->next), &b);
     }
     tail = tail->next;
  }
  return(dummy.next);
}


/* UTILITY FUNCTIONS */
/*MoveNode() function takes the node from the front of the source, and move it to the front of the dest.
   It is an error to call this with the source list empty.

   Before calling MoveNode():
   source == {1, 2, 3}
   dest == {1, 2, 3}

   Affter calling MoveNode():
   source == {2, 3}
   dest == {1, 1, 2, 3}
*/
void MoveNode(struct node** destRef, struct node** sourceRef)
{
  /* the front source node  */
  struct node* newNode = *sourceRef;
  assert(newNode != NULL);
```

```
  /* Advance the source pointer */
  *sourceRef = newNode->next;

  /* Link the old dest off the new node */
  newNode->next = *destRef;

  /* Move dest to point to the new node */
  *destRef = newNode;
}


/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
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* res = NULL;
  struct node* a = NULL;
  struct node* b = NULL;

  /* Let us create two sorted linked lists to test the functions
   Created lists shall be a: 5->10->15,  b: 2->3->20 */
  push(&a, 15);
  push(&a, 10);
  push(&a, 5);

  push(&b, 20);
  push(&b, 3);
```

```
  push(&b, 2);

  /* Remove duplicates from linked list */
  res = SortedMerge(a, b);

  printf("\n Merged Linked List is: \n");
  printList(res);

  getchar();
  return 0;
}
```

Method 2 (Using Local References)
This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** "reference" strategy can be used (see Section 1 for details).

```
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* point to the last result pointer */
  struct node** lastPtrRef = &result;

  while(1)
  {
    if (a == NULL)
    {
      *lastPtrRef = b;
       break;
    }
    else if (b==NULL)
    {
       *lastPtrRef = a;
       break;
    }
    if(a->data <= b->data)
    {
      MoveNode(lastPtrRef, &a);
    }
    else
    {
      MoveNode(lastPtrRef, &b);
    }

    /* tricky: advance to point to the next ".next" field */
    lastPtrRef = &((*lastPtrRef)->next);
  }
  return(result);
}
```

Method 3 (Using Recursion)
Merge is one of those nice recursive problems where the recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists.

```
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* Base cases */
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
  return(result);
}
```

Source: http://cslibrary.stanford.edu/105/LinkedListProblems.pdf

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/merge-two-sorted-linked-lists/

Category: Linked Lists

Post navigation

← Total number of possible Binary Search Trees with n keys A nested loop puzzle →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 25

Identical Linked Lists

Two Linked Lists are identical when they have same data and arrangement of data is also same. For example Linked lists a (1->2->3) and b(1->2->3) are identical. . Write a function to check if the given two linked lists are identical.

Method 1 (Iterative)
To identify if two lists are identical, we need to traverse both lists simultaneously, and while traversing we need to compare data.

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

/* Structure for a linked list node */
struct node
{
  int data;
  struct node *next;
};

/* returns 1 if linked lists a and b are identical, otherwise 0 */
bool areIdentical(struct node *a, struct node *b)
{
  while(1)
  {
    /* base case */
    if(a == NULL && b == NULL)
    {  return 1; }
    if(a == NULL && b != NULL)
    {  return 0; }
    if(a != NULL && b == NULL)
    {  return 0; }
    if(a->data != b->data)
    {  return 0; }

    /* If we reach here, then a and b are not NULL and their
       data is same, so move to next nodes in both lists */
    a = a->next;
    b = b->next;
```

```
  }
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Druver program to test above function */
int main()
{
  struct node *a = NULL;
  struct node *b = NULL;

  /* The constructed linked lists are :
   a: 3->2->1
   b: 3->2->1 */
  push(&a, 1);
  push(&a, 2);
  push(&a, 3);

  push(&b, 1);
  push(&b, 2);
  push(&b, 3);

  if(areIdentical(a, b) == 1)
    printf(" Linked Lists are identical ");
  else
    printf(" Linked Lists are not identical ");

  getchar();
  return 0;
}
```

Method 2 (Recursive)
Recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists

```
bool areIdentical(struct node *a, struct node *b)
{
  if (a == NULL && b == NULL)
  {  return 1;  }
  if (a == NULL && b != NULL)
  {  return 0;  }
  if (a != NULL && b == NULL)
  {  return 0;  }
  if (a->data != b->data)
  {  return 0;  }

  /* If we reach here, then a and b are not NULL and their
       data is same, so move to next nodes in both lists */
  return areIdentical(a->next, b->next);
}
```

Time Complexity: O(n) for both iterative and recursive versions. n is the length of the smaller list among a and b.

Please write comments if you find the above codes/algorithms incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/identical-linked-lists/

Category: Linked Lists

Chapter 26

Merge Sort for Linked Lists

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

```
MergeSort(headRef)
1) If head is NULL or there is only one element in the Linked List
    then return.
2) Else divide the linked list into two halves.
      FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
      MergeSort(a);
      MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
     *headRef = SortedMerge(a, b);
```

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef);
```

```
/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
  struct node* head = *headRef;
  struct node* a;
  struct node* b;

  /* Base case -- length 0 or 1 */
  if ((head == NULL) || (head->next == NULL))
  {
    return;
  }

  /* Split head into 'a' and 'b' sublists */
  FrontBackSplit(head, &a, &b);

  /* Recursively sort the sublists */
  MergeSort(&a);
  MergeSort(&b);

  /* answer = merge the two sorted lists together */
  *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
   function */
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* Base cases */
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
  return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
     and return the two lists using the reference parameters.
```

```
     If the length is odd, the extra node should go in the front list.
     Uses the fast/slow pointer strategy.  */
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef)
{
  struct node* fast;
  struct node* slow;
  if (source==NULL || source->next==NULL)
  {
    /* length < 2 cases */
    *frontRef = source;
    *backRef = NULL;
  }
  else
  {
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL)
    {
      fast = fast->next;
      if (fast != NULL)
      {
        slow = slow->next;
        fast = fast->next;
      }
    }

    /* 'slow' is before the midpoint in the list, so split it in two
      at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
  }
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));
```

```
  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* res = NULL;
  struct node* a = NULL;

  /* Let us create a unsorted linked lists to test the functions
   Created lists shall be a: 2->3->20->5->10->15 */
  push(&a, 15);
  push(&a, 10);
  push(&a, 5);
  push(&a, 20);
  push(&a, 3);
  push(&a, 2);

  /* Sort the above created Linked List */
  MergeSort(&a);

  printf("\n Sorted Linked List is: \n");
  printList(a);

  getchar();
  return 0;
}
```

Time Complexity: O(nLogn)

Sources:
http://en.wikipedia.org/wiki/Merge_sort
http://cslibrary.stanford.edu/105/LinkedListProblems.pdf

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/merge-sort-for-linked-list/

Category: Linked Lists

Post navigation

← Identical Linked Lists Segregate Even and Odd numbers →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 27

Reverse a Linked List in groups of given size

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

```
Example:
Inputs:  1->2->3->4->5->6->7->8->NULL and k = 3
Output:  3->2->1->6->5->4->8->7->NULL.

Inputs:   1->2->3->4->5->6->7->8->NULL and k = 5
Output:  5->4->3->2->1->8->7->6->NULL.
```

Algorithm: *reverse(head, k)*
1) Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be *next* and pointer to the previous node be *prev*. See this post for reversing a linked list.
2) *head->next = reverse(next, k)* /* Recursively call for rest of the list and link the two sub-lists */
3) return *prev* /* *prev* becomes the new head of the list (see the diagrams of iterative method of this post) */

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Reverses the linked list in groups of size k and returns the
   pointer to the new head node. */
struct node *reverse (struct node *head, int k)
{
    struct node* current = head;
```

```
    struct node* next = NULL;
    struct node* prev = NULL;
    int count = 0;

    /*reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
       next  = current->next;
       current->next = prev;
       prev = current;
       current = next;
       count++;
    }

    /* next is now a pointer to (k+1)th node
       Recursively call for the list starting from current.
       And make rest of the list as next of first node */
    if(next !=  NULL)
    {  head->next = reverse(next, k); }

    /* prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
```

```
{
    /* Start with the empty list */
    struct node* head = NULL;

     /* Created Linked list is 1->2->3->4->5->6->7->8 */
     push(&head, 8);
     push(&head, 7);
     push(&head, 6);
     push(&head, 5);
     push(&head, 4);
     push(&head, 3);
     push(&head, 2);
     push(&head, 1);

     printf("\n Given linked list \n");
     printList(head);
     head = reverse(head, 3);

     printf("\n Reversed Linked list \n");
     printList(head);

     getchar();
     return(0);
}
```

Time Complexity: O(n) where n is the number of nodes in the given list.

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/reverse-a-list-in-groups-of-given-size/

Category: Linked Lists

Chapter 28

Reverse alternate K nodes in a Singly Linked List

Given a linked list, write a function to reverse every alternate k nodes (where k is an input to the function) in an efficient way. Give the complexity of your algorithm.

```
Example:
Inputs:   1->2->3->4->5->6->7->8->9->NULL and k = 3
Output:   3->2->1->4->5->6->9->8->7->NULL.
```

Method 1 (Process 2k nodes and recursively call for rest of the list)
This method is basically an extension of the method discussed in thispost.

```
kAltReverse(struct node *head, int k)
  1)  Reverse first k nodes.
  2)  In the modified list head points to the kth node.  So change next
       of head to (k+1)th node
  3)  Move the current pointer to skip next k nodes.
  4)  Call the kAltReverse() recursively for rest of the n - 2k nodes.
  5)  Return new head of the list.
```

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Reverses alternate k nodes and
```

```
   returns the pointer to the new head node */
struct node *kAltReverse(struct node *head, int k)
{
    struct node* current = head;
    struct node* next;
    struct node* prev = NULL;
    int count = 0;

    /*1) reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
       next  = current->next;
       current->next = prev;
       prev = current;
       current = next;
       count++;
    }

    /* 2) Now head points to the kth node.  So change next
       of head to (k+1)th node*/
    if(head != NULL)
      head->next = current;

    /* 3) We do not want to reverse next k nodes. So move the current
        pointer to skip next k nodes */
    count = 0;
    while(count < k-1 && current != NULL )
    {
      current = current->next;
      count++;
    }

    /* 4) Recursively call for the list starting from current->next.
       And make rest of the list as next of first node */
    if(current !=  NULL)
       current->next = kAltReverse(current->next, k);

    /* 5) prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);
```

```
    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    int count = 0;
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
        count++;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    // create a list 1->2->3->4->5...... ->20
    for(int i = 20; i > 0; i--)
      push(&head, i);

     printf("\n Given linked list \n");
     printList(head);
     head = kAltReverse(head, 3);

     printf("\n Modified Linked list \n");
     printList(head);

     getchar();
     return(0);
}
```

Output:
Given linked list
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Modified Linked list
3 2 1 4 5 6 9 8 7 10 11 12 15 14 13 16 17 18 20 19

Time Complexity: O(n)

Method 2 (Process k nodes and recursively call for rest of the list)
The method 1 reverses the first k node and then moves the pointer to k nodes ahead. So method 1 uses two while loops and processes 2k nodes in one recursive call.
This method processes only k nodes in a recursive call. It uses a third bool parameter b which decides whether to reverse the k elements or simply move the pointer.

```
_kAltReverse(struct node *head, int k, bool b)
```

```
  1)  If b is true, then reverse first k nodes.
  2)  If b is false, then move the pointer k nodes ahead.
  3)  Call the kAltReverse() recursively for rest of the n - k nodes and link
       rest of the modified list with end of first k nodes.
  4)  Return new head of the list.


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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Helper function for kAltReverse() */
struct node * _kAltReverse(struct node *node, int k, bool b);

/* Alternatively reverses the given linked list in groups of
   given size k. */
struct node *kAltReverse(struct node *head, int k)
{
  return _kAltReverse(head, k, true);
}

/*  Helper function for kAltReverse().  It reverses k nodes of the list only if
    the third parameter b is passed as true, otherwise moves the pointer k
    nodes ahead and recursively calls iteself  */
struct node * _kAltReverse(struct node *node, int k, bool b)
{
   if(node == NULL)
       return NULL;

   int count = 1;
   struct node *prev = NULL;
   struct node  *current = node;
   struct node *next;

   /* The loop serves two purposes
      1) If b is true, then it reverses the k nodes
      2) If b is false, then it moves the current pointer */
   while(current != NULL && count <= k)
   {
       next = current->next;

       /* Reverse the nodes only if b is true*/
       if(b == true)
          current->next = prev;

       prev = current;
       current = next;
       count++;
```

```
   }

   /* 3) If b is true, then node is the kth node.
       So attach rest of the list after node.
     4) After attaching, return the new head */
   if(b == true)
   {
        node->next = _kAltReverse(current,k,!b);
        return prev;
   }

   /* If b is not true, then attach rest of the list after prev.
     So attach rest of the list after prev */
   else
   {
        prev->next = _kAltReverse(current, k, !b);
        return node;
   }
}


/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    int count = 0;
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
        count++;
    }
}

/* Drier program to test above function*/
int main(void)
{
```

```
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    // create a list 1->2->3->4->5...... ->20
    for(i = 20; i > 0; i--)
      push(&head, i);

    printf("\n Given linked list \n");
    printList(head);
    head = kAltReverse(head, 3);

    printf("\n Modified Linked list \n");
    printList(head);

    getchar();
    return(0);
}
```

Output:
Given linked list
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Modified Linked list
3 2 1 4 5 6 9 8 7 10 11 12 15 14 13 16 17 18 20 19

Time Complexity: O(n)

Source:
http://geeksforgeeks.org/forum/topic/amazon-interview-question-2

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/reverse-alternate-k-nodes-in-a-singly-linked-list/

Category: Linked Lists

Post navigation

← Program to count number of set bits in an (big) array Next Greater Element →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 29

Delete nodes which have a greater value on right side

Given a singly linked list, remove all the nodes which have a greater value on right side.

Examples:
a) The list 12->15->10->11->5->6->2->3->NULL should be changed to 15->11->6->3->NULL. Note that 12, 10, 5 and 2 have been deleted because there is a greater value on the right side.

When we examine 12, we see that after 12 there is one node with value greater than 12 (i.e. 15), so we delete 12.
When we examine 15, we find no node after 15 that has value greater than 15 so we keep this node.
When we go like this, we get 15->6->3

b) The list 10->20->30->40->50->60->NULL should be changed to 60->NULL. Note that 10, 20, 30, 40 and 50 have been deleted because they all have a greater value on the right side.

c) The list 60->50->40->30->20->10->NULL should not be changed.

Method 1 (Simple)
Use two loops. In the outer loop, pick nodes of the linked list one by one. In the inner loop, check if there exist a node whose value is greater than the picked node. If there exists a node whose value is greater, then delete the picked node.
Time Complexity: O(n^2)

Method 2 (Use Reverse)
Thanks to Parasfor providing the below algorithm.
1. Reverse the list.
2. Traverse the reversed list. Keep max till now. If next node R.Srinivasan for providing below code.

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

/* structure of a linked list node */
struct node
{
     int data;
     struct node *next;
};
```

```
/* prototype for utility functions */
void reverseList(struct node **headref);
void _delLesserNodes(struct node *head);

/* Deletes nodes which have a node with greater value node
  on left side */
void delLesserNodes(struct node **head_ref)
{
    /* 1) Reverse the linked list */
    reverseList(head_ref);

    /* 2) In the reversed list, delete nodes which have a node
       with greater value node on left side. Note that head
       node is never deleted because it is the leftmost node.*/
    _delLesserNodes(*head_ref);

    /* 3) Reverse the linked list again to retain the
       original order */
    reverseList(head_ref);
}

/* Deletes nodes which have greater value node(s) on left side */
void _delLesserNodes(struct node *head)
{
     struct node *current = head;

     /* Initialize max */
     struct node *maxnode = head;
     struct node *temp;

     while (current != NULL && current->next != NULL)
     {
         /* If current is smaller than max, then delete current */
         if(current->next->data < maxnode->data)
         {
             temp = current->next;
             current->next = temp->next;
             free(temp);
         }

         /* If current is greater than max, then update max and
            move current */
         else
         {
             current = current->next;
             maxnode = current;
         }

     }
}

/* Utility function to insert a node at the begining */
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;
}

/* Utility function to reverse a linked list */
void reverseList(struct node **headref)
{
     struct node *current = *headref;
     struct node *prev = NULL;
     struct node *next;
     while(current != NULL)
     {
          next = current->next;
          current->next = prev;
          prev = current;
          current = next;
     }
     *headref = prev;
}

/* Utility function to print a linked list */
void printList(struct node *head)
{
     while(head!=NULL)
     {
        printf("%d ",head->data);
        head=head->next;
     }
     printf("\n");
}

/* Driver program to test above functions */
int main()
{
    struct node *head = NULL;

    /* Create following linked list
      12->15->10->11->5->6->2->3 */
    push(&head,3);
    push(&head,2);
    push(&head,6);
    push(&head,5);
    push(&head,11);
    push(&head,10);
    push(&head,15);
    push(&head,12);

    printf("Given Linked List: ");
    printList(head);

    delLesserNodes(&head);
```

```
    printf("\nModified Linked List: ");
    printList(head);

    getchar();
    return 0;
}
```

Output:

```
Given Linked List: 12 15 10 11 5 6 2 3
Modified Linked List: 15 11 6 3
```

Source:

http://geeksforgeeks.org/forum/topic/amazon-interview-question-for-software-engineerdeveloper-about-linked-lists-6

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/delete-nodes-which-have-a-greater-value-on-right-side/

Chapter 30

Segregate even and odd nodes in a Linked List

Given a Linked List of integers, write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers same.

Examples:
Input: 17->15->8->12->10->5->4->1->7->6->NULL
Output: 8->12->10->4->6->17->15->5->1->7->NULL

Input: 8->12->10->5->4->1->6->NULL
Output: 8->12->10->4->6->5->1->NULL

// If all numbers are even then do not change the list
Input: 8->12->10->NULL
Output: 8->12->10->NULL

// If all numbers are odd then do not change the list
Input: 1->3->5->7->NULL
Output: 1->3->5->7->NULL

Method 1
The idea is to get pointer to the last node of list. And then traverse the list starting from the head node and move the odd valued nodes from their current position to end of the list.

Thanks to blunderboy for suggesting this method.

Algorithm:
…1) Get pointer to the last node.
…2) Move all the odd nodes to the end.
……..a) Consider all odd nodes before the first even node and move them to end.
……..b) Change the head pointer to point to the first even node.
……..b) Consider all odd nodes after the first even node and move them to the end.

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

/* a node of the singly linked list */
struct node
{
    int data;
```

```
    struct node *next;
};

void segregateEvenOdd(struct node **head_ref)
{
    struct node *end = *head_ref;

    struct node *prev = NULL;
    struct node *curr = *head_ref;

    /* Get pointer to the last node */
    while (end->next != NULL)
        end = end->next;

    struct node *new_end = end;

    /* Consider all odd nodes before the first even node
       and move then after end */
    while (curr->data %2 != 0 && curr != end)
    {
        new_end->next = curr;
        curr = curr->next;
        new_end->next->next = NULL;
        new_end = new_end->next;
    }

    // 10->8->17->17->15
    /* Do following steps only if there is any even node */
    if (curr->data%2 == 0)
    {
        /* Change the head pointer to point to first even node */
        *head_ref = curr;

        /* now current points to the first even node */
        while (curr != end)
        {
            if ( (curr->data)%2 == 0 )
            {
                prev = curr;
                curr = curr->next;
            }
            else
            {
                /* break the link between prev and current */
                prev->next = curr->next;

                /* Make next of curr as NULL  */
                curr->next = NULL;

                /* Move curr to end */
                new_end->next = curr;

                /* make curr as new end of list */
                new_end = curr;
```

```
                /* Update current pointer to next of the moved node */
                curr = prev->next;
            }
        }
    }

    /* We must have prev set before executing lines following this
       statement */
    else prev = curr;

    /* If there are more than 1 odd nodes and end of original list is
      odd then move this node to end to maintain same order of odd
      numbers in modified list */
    if (new_end!=end && (end->data)%2 != 0)
    {
        prev->next = end->next;
        end->next = NULL;
        new_end->next = end;
    }
    return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
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;

    /* Let us create a sample linked list as following
      0->2->4->6->8->10->11 */

    push(&head, 11);
    push(&head, 10);
    push(&head, 8);
    push(&head, 6);
    push(&head, 4);
    push(&head, 2);
    push(&head, 0);

    printf("\n Original Linked list ");
    printList(head);

    segregateEvenOdd(&head);

    printf("\n Modified Linked list ");
    printList(head);

    return 0;
}
```

Output:

```
 Original Linked list 0 2 4 6 8 10 11
 Modified Linked list 0 2 4 6 8 10 11
```

Time complexity: O(n)

Method 2
The idea is to split the linked list into two: one containing all even nodes and other containing all odd nodes. And finally attach the odd node linked list after the even node linked list.
To split the Linked List, traverse the original Linked List and move all odd nodes to a separate Linked List of all odd nodes. At the end of loop, the original list will have all the even nodes and the odd node list will have all the odd nodes. To keep the ordering of all nodes same, we must insert all the odd nodes at the end of the odd node list. And to do that in constant time, we must keep track of last pointer in the odd node list.

Time complexity: O(n)

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

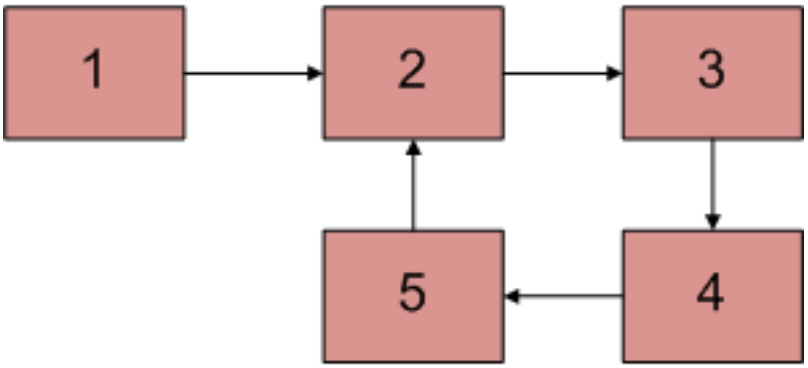
Source

http://www.geeksforgeeks.org/segregate-even-and-odd-elements-in-a-linked-list/

Chapter 31

Detect and Remove Loop in a Linked List

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.



We recommend to read following post as a prerequisite.

Write a C function to detect loop in a linked list

Before trying to remove the loop, we must detect it. Techniques discussed in the above post can be used to detect loop. To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone.
We can easily use Hashing or Visited node techniques (discussed in the above mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.
We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

Method 1 (Check one by one)
We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say ptr2. Then we start from the head of the Linked List

and check for nodes one by one if they are reachable from ptr2. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node  *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

/* Function to remove loop.
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
   struct node *ptr1;
   struct node *ptr2;

   /* Set a pointer to the beging of the Linked List and
```

```
      move it one by one to find the first node which is
      part of the Linked List */
   ptr1 = head;
   while (1)
   {
     /* Now start a pointer from loop_node and check if it ever
       reaches ptr2 */
     ptr2 = loop_node;
     while (ptr2->next != loop_node && ptr2->next != ptr1)
         ptr2 = ptr2->next;

     /* If ptr2 reahced ptr1 then there is a loop. So break the
        loop */
     if (ptr2->next == ptr1)
        break;

     /* If ptr2 did't reach ptr1 then try the next node after ptr1 */
     ptr1 = ptr1->next;
   }

   /* After the end of loop ptr2 is the last node of the loop. So
     make next of ptr2 as NULL */
   ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Drier program to test above function*/
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;
```

```
    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}
```

Output:

```
 Linked List after removing loop
50 20 15 4 10
```

Method 2 (Better Solution)
This method is also dependent on Floyd's Cycle detection algorithm.
1) Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.
2) Count the number of nodes in loop. Let the count be k.
3) Fix one pointer to the head and another to kth node from head.
4) Move both pointers at the same pace, they will meet at loop starting node.
5) Get pointer to the last node of loop and make next of it as NULL.

Thanks to WgpShashank for suggesting this method.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node  *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
```

```
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

/* Function to remove loop.
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1 = loop_node;
    struct node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for (i = 0; i < k; i++)
      ptr2 = ptr2->next;

    /*  Move both pointers at the same pace,
      they will meet at loop starting node */
    while (ptr2 != ptr1)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    ptr2 = ptr2->next;
    while (ptr2->next != ptr1)
       ptr2 = ptr2->next;

    /* Set the next node of the loop ending node
      to fix the loop */
    ptr2->next = NULL;
}

/* Function to print linked list */
```

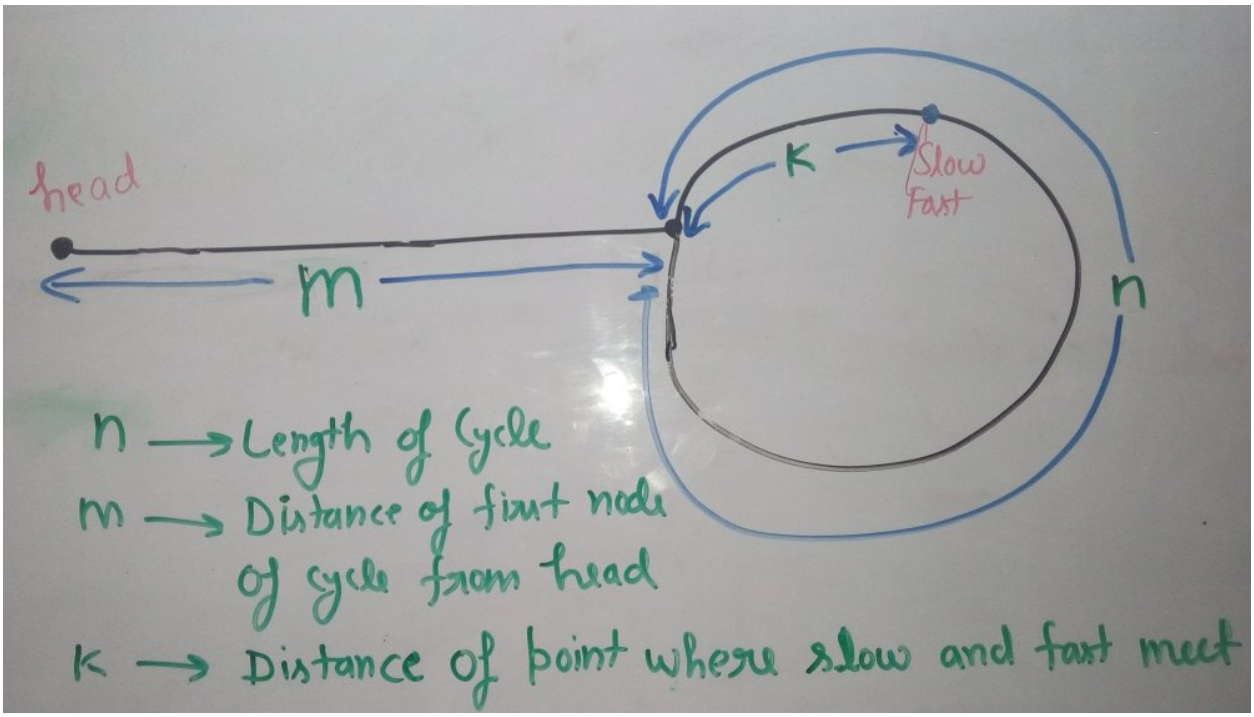
```
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Drier program to test above function*/
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}
```

Output:

```
 Linked List after removing loop
50 20 15 4 10
```

Method 3 (Optimized Method 2: Without Counting Nodes in Loop)
We do not need to count number of nodes in Loop. After detecting the loop, if we start slow pointer from head and move both slow and fast pointers at same speed until fast don't meet, they would meet at the beginning of linked list.

How does this work?
Let slow and fast meet at some point after Floyd's Cycle finding algorithm. Below diagram shows the situation when cycle is found.



We can conclude below from above diagram

```
Distance traveled by fast pointer = 2 * (Distance traveled
                                         by slow pointer)

(m + n*x + k) = 2*(m + n*y + k)

Note that before meeting the point shown above, fast
was moving at twice speed.

x -->  Number of complete cyclic rounds made by
       fast pointer before they meet first time

y -->  Number of complete cyclic rounds made by
       slow pointer before they meet first time
```

From above equation, we can conclude below

```
    m + k = (x-2y)*n

Which means m+k is a multiple of n.
```

So if we start moving both pointers again at **same speed** such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of linked list (has made m steps). Fast pointer would have made also moved m steps as they

are now moving same pace. Since m+k is a multiple of n and fast starts from k, they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after m steps.

```
// C++ program to detect and remove loop
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int key;
    struct Node *next;
};

Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printList(Node *head)
{
    while (head != NULL)
    {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

void detectAndRemoveLoop(Node *head)
{
    Node *slow = head;
    Node *fast = head->next;

    // Search for loop using slow and fast pointers
    while (fast && fast->next)
    {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    /* If loop exists */
    if (slow == fast)
    {
        slow = head;
        while (slow != fast->next)
        {
            slow = slow->next;
```

```
            fast = fast->next;
        }

        /* since fast->next is the looping point */
        fast->next = NULL; /* remove loop */
    }
}

/* Drier program to test above function*/
int main()
{
    Node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    return 0;
}
```

Output:

```
 Linked List after removing loop
50 20 15 4 10
```

Thanks to Gaurav Ahirwar for suggesting above solution.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/detect-and-remove-loop-in-a-linked-list/

Chapter 32

Add two numbers represented by linked lists | Set 1

Given two numbers represented by two lists, write a function that returns sum list. The sum list is list representation of addition of two input numbers.

Example 1

```
Input:
  First List: 5->6->3  // represents number 365
  Second List: 8->4->2 //  represents number 248
Output
  Resultant list: 3->1->6  // represents number 613
```

Example 2

```
Input:
  First List: 7->5->9->4->6  // represents number 64957
  Second List: 8->4 //  represents number 48
Output
  Resultant list: 5->0->0->5->6  // represents number 65005
```

Solution
Traverse both lists. One by one pick nodes of both lists and add the values. If sum is more than 10 then make carry as 1 and reduce sum. If one list has more elements than the other then consider remaining values of this list as 0. Following is C implementation of this approach.

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

/* Linked list node */
struct node
```

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

/* Function to create a new node with given data */
struct node *newNode(int data)
{
    struct node *new_node = (struct node *) malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = newNode(new_data);

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Adds contents of two linked lists and return the head node of resultant list */
struct node* addTwoLists (struct node* first, struct node* second)
{
    struct node* res = NULL; // res is head node of the resultant list
    struct node *temp, *prev = NULL;
    int carry = 0, sum;

    while (first != NULL || second != NULL) //while both lists exist
    {
        // Calculate value of next digit in resultant list.
        // The next digit is sum of following things
        // (i)  Carry
        // (ii) Next digit of first list (if there is a next digit)
        // (ii) Next digit of second list (if there is a next digit)
        sum = carry + (first? first->data: 0) + (second? second->data: 0);

        // update carry for next calulation
        carry = (sum >= 10)? 1 : 0;

        // update sum if it is greater than 10
        sum = sum % 10;

        // Create a new node with sum as data
        temp = newNode(sum);

        // if this is the first node then set it as head of the resultant list
        if(res == NULL)
```

```
            res = temp;
        else // If this is not the first node then connect it to the rest.
            prev->next = temp;

        // Set prev for next insertion
        prev  = temp;

        // Move first and second pointers to next nodes
        if (first) first = first->next;
        if (second) second = second->next;
    }

    if (carry > 0)
      temp->next = newNode(carry);

    // return head of the resultant list
    return res;
}

// A utility function to print a linked list
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Drier program to test above function */
int main(void)
{
    struct node* res = NULL;
    struct node* first = NULL;
    struct node* second = NULL;

    // create first list 7->5->9->4->6
    push(&first, 6);
    push(&first, 4);
    push(&first, 9);
    push(&first, 5);
    push(&first, 7);
    printf("First List is ");
    printList(first);

    // create second list 8->4
    push(&second, 4);
    push(&second, 8);
    printf("Second List is ");
    printList(second);

    // Add the two lists and see result
    res = addTwoLists(first, second);
```

```
    printf("Resultant list is ");
    printList(res);

   return 0;
}
```

Output:

```
First List is 7 5 9 4 6
Second List is 8 4
Resultant list is 5 0 0 5 6
```

Time Complexity: O(m + n) where m and n are number of nodes in first and second lists respectively.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/add-two-numbers-represented-by-linked-lists/

Category: Linked Lists

Chapter 33

Delete a given node in Linked List under given constraints

Given a Singly Linked List, write a function to delete a given node. Your function must follow following constraints:
1) It must accept pointer to the start node as first parameter and node to be deleted as second parameter i.e., pointer to head node is not global.
2) It should not return pointer to the head node.
3) It should not accept pointer to pointer to head node.

You may assume that the Linked List never becomes empty.

Let the function name be deleteNode(). In a straightforward implementation, the function needs to modify head pointer when the node to be deleted is first node. As discussed in previous post, when a function modifies the head pointer, the function must use one of the given approaches, we can't use any of those approaches here.

Solution
We explicitly handle the case when node to be deleted is first node, we copy the data of next node to head and delete the next node. The cases when deleted node is not the head node can be handled normally by finding the previous node and changing next of previous node. Following is C implementation.

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

/* structure of a linked list node */
struct node
{
    int data;
    struct node *next;
};

void deleteNode(struct node *head, struct node *n)
{
    // When node to be deleted is head node
    if(head == n)
    {
        if(head->next == NULL)
        {
```

```
            printf("There is only one node. The list can't be made empty ");
            return;
        }

        /* Copy the data of next node to head */
        head->data = head->next->data;

        // store address of next node
        n = head->next;

        // Remove the link of next node
        head->next = head->next->next;

        // free memory
        free(n);

        return;
    }


    // When not first node, follow the normal deletion process

    // find the previous node
    struct node *prev = head;
    while(prev->next != NULL && prev->next != n)
        prev = prev->next;

    // Check if node really exists in Linked List
    if(prev->next == NULL)
    {
        printf("\n Given node is not present in Linked List");
        return;
    }

    // Remove node from Linked List
    prev->next = prev->next->next;

    // Free memory
    free(n);

    return;
}

/* Utility function to insert a node at the begining */
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;
}

/* Utility function to print a linked list */
```

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

/* Driver program to test above functions */
int main()
{
    struct node *head = NULL;

    /* Create following linked list
      12->15->10->11->5->6->2->3 */
    push(&head,3);
    push(&head,2);
    push(&head,6);
    push(&head,5);
    push(&head,11);
    push(&head,10);
    push(&head,15);
    push(&head,12);

    printf("Given Linked List: ");
    printList(head);

    /* Let us delete the node with value 10 */
    printf("\nDeleting node %d: ", head->next->next->data);
    deleteNode(head, head->next->next);

    printf("\nModified Linked List: ");
    printList(head);

    /* Let us delete the the first node */
    printf("\nDeleting first node ");
    deleteNode(head, head);

    printf("\nModified Linked List: ");
    printList(head);

    getchar();
    return 0;
}
```

Output:

```
Given Linked List: 12 15 10 11 5 6 2 3
```

```
Deleting node 10:
Modified Linked List: 12 15 11 5 6 2 3

Deleting first node
Modified Linked List: 15 11 5 6 2 3
```

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/delete-a-given-node-in-linked-list-under-given-constraints/

Category: Linked Lists

Chapter 34

Union and Intersection of two Linked Lists

Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elments in output lists doesn't matter.

Example:

```
Input:
   List1: 10->15->4->20
   lsit2:  8->4->2->10
Output:
   Intersection List: 4->10
   Union List: 2->8->20->4->15->10
```

Method 1 (Simple)
Following are simple algorithms to get union and intersection lists respectively.

Intersection (list1, list2)
Initialize result list as NULL. Traverse list1 and look for its each element in list2, if the element is present in list2, then add the element to result.

Union *(list1, list2):*
Initialize result list as NULL. Traverse list1 and add all of its elements to the result.
Traverse list2. If an element of list2 is already present in result then do not insert it to result, otherwise insert.

This method assumes that there are no duplicates in the given lists.

Thanks to Shekhufor suggesting this method. Following is C implementation of this method.

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

/* Link list node */
struct node
{
```

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

/* A utility function to insert a node at the begining of a linked list*/
void push (struct node** head_ref, int new_data);

/* A utilty function to chec if given data is present in a list */
bool isPresent (struct node *head, int data);

/* Function to get union of two linked lists head1 and head2 */
struct node *getUnion (struct node *head1, struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1, *t2 = head2;

    // Insert all elements of list1 to the result list
    while (t1 != NULL)
    {
        push(&result, t1->data);
        t1 = t1->next;
    }

    // Insert those elements of list2 which are not present in result list
    while (t2 != NULL)
    {
        if (!isPresent(result, t2->data))
            push(&result, t2->data);
        t2 = t2->next;
    }

    return result;
}

/* Function to get intersection of two linked lists head1 and head2 */
struct node *getIntersection (struct node *head1, struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1;

    // Traverse list1 and search each element of it in list2. If the element
    // is present in list 2, then insert the element to result
    while (t1 != NULL)
    {
        if (isPresent(head2, t1->data))
            push (&result, t1->data);
        t1 = t1->next;
    }

    return result;
}

/* A utility function to insert a node at the begining of a linked list*/
void push (struct node** head_ref, int new_data)
```

```
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* A utility function to print a linked list*/
void printList (struct node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}

/* A utilty function that returns true if data is present in linked list
else return false */
bool isPresent (struct node *head, int data)
{
    struct node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
        t = t->next;
    }
    return 0;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head1 = NULL;
    struct node* head2 = NULL;
    struct node* intersecn = NULL;
    struct node* unin = NULL;

    /*create a linked lits 10->15->5->20 */
    push (&head1, 20);
    push (&head1, 4);
    push (&head1, 15);
    push (&head1, 10);
```

```
    /*create a linked lits 8->4->2->10 */
    push (&head2, 10);
    push (&head2, 2);
    push (&head2, 4);
    push (&head2, 8);

    intersecn = getIntersection (head1, head2);
    unin = getUnion (head1, head2);

    printf ("\n First list is \n");
    printList (head1);

    printf ("\n Second list is \n");
    printList (head2);

    printf ("\n Intersection list is \n");
    printList (intersecn);

    printf ("\n Union list is \n");
    printList (unin);

    return 0;
}
```

Output:

```
 First list is
10 15 4 20
 Second list is
8 4 2 10
 Intersection list is
4 10
 Union list is
2 8 20 4 15 10
```

Time Complexity: O(mn) for both union and intersection operations. Here m is the number of elements in first list and n is the number of elements in second list.

Method 2 (Use Merge Sort)
In this method, algorithms for Union and Intersection are very similar. First we sort the given lists, then we traverse the sorted lists to get union and intersection.
Following are the steps to be followed to get union and intersection lists.

1) Sort the first Linked List using merge sort. This step takes O(mLogm) time. Refer this post for details of this step.
2) Sort the second Linked List using merge sort. This step takes O(nLogn) time. Refer this post for details of this step.
3) Linearly scan both sorted lists to get the union and intersection. This step takes O(m + n) time. This step can be implemented using the same algorithm as sorted arrays algorithm discussed here.

Time complexity of this method is O(mLogm + nLogn) which is better than method 1's time complexity.

Method 3 (Use Hashing)
Union (list1, list2)
Initialize the result list as NULL and create an empty hash table. Traverse both lists one by one, for each element being visited, look the element in hash table. If the element is not present, then insert the element to result list. If the element is present, then ignore it.

Intersection (list1, list2)
Initialize the result list as NULL and create an empty hash table. Traverse list1. For each element being visited in list1, insert the element in hash table. Traverse list2, for each element being visited in list2, look the element in hash table. If the element is present, then insert the element to result list. If the element is not present, then ignore it.

Both of the above methods assume that there are no duplicates.

Time complexity of this method depends on the hashing technique used and the distribution of elements in input lists. In practical, this approach may turn out to be better than above 2 methods.

Source: http://geeksforgeeks.org/forum/topic/union-intersection-of-unsorted-lists

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/union-and-intersection-of-two-linked-lists/

Chapter 35

Find a triplet from three linked lists with sum equal to a given number

Given three linked lists, say a, b and c, find one node from each list such that the sum of the values of the nodes is equal to a given number.
For example, if the three linked lists are 12->6->29, 23->5->8 and 90->20->59, and the given number is 101, the output should be tripel "6 5 90 .

In the following solutions, size of all three linked lists is assumed same for simplicity of analysis. The following solutions work for linked lists of different sizes also.

A simple method to solve this problem is to run three nested loops. The outermost loop picks an element from list a, the middle loop picks an element from b and the innermost loop picks from c. The innermost loop also checks whether the sum of values of current nodes of a, b and c is equal to given number. The time complexity of this method will be O(n^3).

Sorting can be used to reduce the time complexity to O(n*n). Following are the detailed steps.
1) Sort list b in ascending order, and list c in descending order.
2) After the b and c are sorted, one by one pick an element from list a and find the pair by traversing both b and c. See isSumSorted() in the following code. The idea is similar to Quadratic algorithm of 3 sum problem.

Following code implements step 2 only. The solution can be easily modified for unsorted lists by adding the merge sort code discussed here.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* A utility function to insert a node at the begining of a linked list*/
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
```

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

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* A function to chech if there are three elements in a, b and c whose sum
   is equal to givenNumber.  The function assumes that the list b is sorted
   in ascending order and c is sorted in descending order. */
bool isSumSorted(struct node *headA, struct node *headB,
                 struct node *headC, int givenNumber)
{
    struct node *a = headA;

    // Traverse through all nodes of a
    while (a != NULL)
    {
        struct node *b = headB;
        struct node *c = headC;

        // For every node of list a, prick two nodes from lists b abd c
        while (b != NULL && c != NULL)
        {
            // If this a triplet with given sum, print it and return true
            int sum = a->data + b->data + c->data;
            if (sum == givenNumber)
            {
               printf ("Triplet Found: %d %d %d ", a->data, b->data, c->data);
               return true;
            }

            // If sum of this triplet is smaller, look for greater values in b
            else if (sum < givenNumber)
                b = b->next;
            else // If sum is greater, look for smaller values in c
                c = c->next;
        }
        a = a->next;  // Move ahead in list a
    }

    printf ("No such triplet");
    return false;
}


/* Drier program to test above function*/
int main()
```

```
{
    /* Start with the empty list */
    struct node* headA = NULL;
    struct node* headB = NULL;
    struct node* headC = NULL;

    /*create a linked list 'a' 10->15->5->20 */
    push (&headA, 20);
    push (&headA, 4);
    push (&headA, 15);
    push (&headA, 10);

    /*create a sorted linked list 'b' 2->4->9->10 */
    push (&headB, 10);
    push (&headB, 9);
    push (&headB, 4);
    push (&headB, 2);

    /*create another sorted linked list 'c' 8->4->2->1 */
    push (&headC, 1);
    push (&headC, 2);
    push (&headC, 4);
    push (&headC, 8);

    int givenNumber = 25;

    isSumSorted (headA, headB, headC, givenNumber);

    return 0;
}
```

Output:

```
 Triplet Found: 15 2 8
```

Time complexity: The linked lists b and c can be sorted in O(nLogn) time using Merge Sort (See this). The step 2 takes O(n*n) time. So the overall time complexity is O(nlogn) + O(nlogn) + O(n*n) = O(n*n).

In this approach, the linked lists b and c are sorted first, so their original order will be lost. If we want to retain the original order of b and c, we can create copy of b and c.

This article is compiled by **Abhinav Priyadarshi** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-a-triplet-from-three-linked-lists-with-sum-equal-to-a-given-number/

Chapter 36

Rotate a Linked List

Given a singly linked list, rotate the linked list counter-clockwise by k nodes. Where k is a given positive integer. For example, if the given linked list is 10->20->30->40->50->60 and k is 4, the list should be modified to 50->60->10->20->30->40. Assume that k is smaller than the count of nodes in linked list.

To rotate the linked list, we need to change next of kth node to NULL, next of last node to previous head node, and finally change head to (k+1)th node. So we need to get hold of three nodes: kth node, (k+1)th node and last node.
Traverse the list from beginning and stop at kth node. Store pointer to kth node. We can get (k+1)th node using kthNode->next. Keep traversing till end and store pointer to last node also. Finally, change pointers as stated above.

```
// Program to rotate a linked list counter clock wise
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

// This function rotates a linked list counter-clockwise and updates the head.
// The function assumes that k is smaller than size of linked list. It doesn't
// modify the list if k is greater than or equal to size
void rotate (struct node **head_ref, int k)
{
     if (k == 0)
       return;

    // Let us understand the below code for example k = 4 and
    // list = 10->20->30->40->50->60.
    struct node* current = *head_ref;

    // current will either point to kth or NULL after this loop.
    //  current will point to node 40 in the above example
    int count = 1;
```

```
    while (count < k && current != NULL)
    {
        current = current->next;
        count++;
    }

    // If current is NULL, k is greater than or equal to count
    // of nodes in linked list. Don't change the list in this case
    if (current == NULL)
        return;

    // current points to kth node. Store it in a variable.
    // kthNode points to node 40 in the above example
    struct node *kthNode = current;

    // current will point to last node after this loop
    // current will point to node 60 in the above example
    while (current->next != NULL)
        current = current->next;

    // Change next of last node to previous head
    // Next of 60 is now changed to node 10
    current->next = *head_ref;

    // Change head to (k+1)th node
    // head is now changed to node 50
    *head_ref = kthNode->next;

    // change next of kth node to NULL
    // next of 40 is now NULL
    kthNode->next = NULL;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
```

```
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    // create a list 10->20->30->40->50->60
    for (int i = 60; i > 0; i -= 10)
        push(&head, i);

    printf("Given linked list \n");
    printList(head);
    rotate(&head, 4);

    printf("\nRotated Linked list \n");
    printList(head);

    return (0);
}
```

Output:

```
 Given linked list
10  20  30  40  50  60
Rotated Linked list
50  60  10  20  30  40
```

Time Complexity: O(n) where n is the number of nodes in Linked List. The code traverses the linked list only once.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

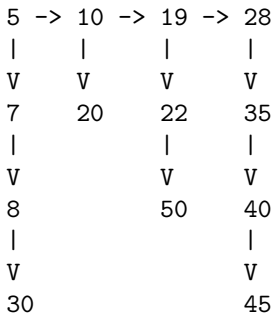
http://www.geeksforgeeks.org/rotate-a-linked-list/

Category: Linked Lists

Chapter 37

Flattening a Linked List

Given a linked list where every node represents a linked list and contains two pointers of its type:
(i) Pointer to next node in the main list (we call it 'right' pointer in below code)
(ii) Pointer to a linked list where this node is head (we call it 'down' pointer in below code).
All linked lists are sorted. See the following example



Write a function flatten() to flatten the lists into a single linked list. The flattened linked list should also be sorted. For example, for the above input list, output list should be 5->7->8->10->19->20->22->28->30->35->40->45->50.

The idea is to use Merge() process of merge sort for linked lists. We use merge() to merge lists one by one. We recursively merge() the current list with already flattened list.
The down pointer is used to link nodes of the flattened list.

Following is C implementation.

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

// A Linked List Node
typedef struct Node
{
    int data;
    struct Node *right;
    struct Node *down;
```

```
} Node;

/* A utility function to insert a new node at the begining
   of linked list */
void push (Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = (Node *) malloc(sizeof(Node));
    new_node->right = NULL;

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->down = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in the flattened linked list */
void printList(Node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->down;
    }
}

// A utility function to merge two sorted linked lists
Node* merge( Node* a, Node* b )
{
    // If first list is empty, the second list is result
    if (a == NULL)
        return b;

    // If second list is empty, the second list is result
    if (b == NULL)
        return a;

    // Compare the data members of head nodes of both lists
    // and put the smaller one in result
    Node* result;
    if( a->data < b->data )
    {
        result = a;
        result->down = merge( a->down, b );
    }
    else
    {
        result = b;
        result->down = merge( a, b->down );
    }
```

```
    return result;
}

// The main function that flattens a given linked list
Node* flatten (Node* root)
{
    // Base cases
    if ( root == NULL || root->right == NULL )
        return root;

    // Merge this list with the list on right side
    return merge( root, flatten(root->right) );
}

// Driver program to test above functions
int main()
{
    Node* root = NULL;

    /* Let us create the following linked list
       5 -> 10 -> 19 -> 28
       |    |     |     |
       V    V     V     V
       7    20    22    35
       |          |     |
       V          V     V
       8          50    40
       |                |
       V                V
       30               45
    */
    push( &root, 30 );
    push( &root, 8 );
    push( &root, 7 );
    push( &root, 5 );

    push( &( root->right ), 20 );
    push( &( root->right ), 10 );

    push( &( root->right->right ), 50 );
    push( &( root->right->right ), 22 );
    push( &( root->right->right ), 19 );

    push( &( root->right->right->right ), 45 );
    push( &( root->right->right->right ), 40 );
    push( &( root->right->right->right ), 35 );
    push( &( root->right->right->right ), 20 );

    // Let us flatten the list
    root = flatten(root);

    // Let us print the flatened linked list
    printList(root);
```

```
    return 0;
}
```

Output:

```
 5 7 8 10 19 20 20 22 30 35 40 45 50
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/flattening-a-linked-list/

Category: Linked Lists

Chapter 38

Add two numbers represented by linked lists | Set 2

Given two numbers represented by two linked lists, write a function that returns sum list. The sum list is linked list representation of addition of two input numbers. It is not allowed to modify the lists. Also, not allowed to use explicit extra space (Hint: Use Recursion).

Example

```
Input:
  First List: 5->6->3  // represents number 563
  Second List: 8->4->2 //  represents number 842
Output
  Resultant list: 1->4->0->5  // represents number 1405
```

We have discussed a solution herewhich is for linked lists where least significant digit is first node of lists and most significant digit is last node. In this problem, most significant node is first node and least significant digit is last node and we are not allowed to modify the lists. Recursion is used here to calculate sum from right to left.

Following are the steps.
1) Calculate sizes of given two linked lists.
2) If sizes are same, then calculate sum using recursion. Hold all nodes in recursion call stack till the rightmost node, calculate sum of rightmost nodes and forward carry to left side.
3) If size is not same, then follow below steps:
....**a)** Calculate difference of sizes of two linked lists. Let the difference be *diff*
....**b)** Move *diff* nodes ahead in the bigger linked list. Now use step 2 to calculate sum of smaller list and right sub-list (of same size) of larger list. Also, store the carry of this sum.
....**c)** Calculate sum of the carry (calculated in previous step) with the remaining left sub-list of larger list. Nodes of this sum are added at the beginning of sum list obtained previous step.

Following is C implementation of the above approach.

```
// A recursive program to add two linked lists

#include <stdio.h>
```

```
#include <stdlib.h>

// A linked List Node
struct node
{
    int data;
    struct node* next;
};

typedef struct node node;

/* A utility function to insert a node at the beginning of linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* A utility function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
    printf("\n");
}

// A utility function to swap two pointers
void swapPointer( node** a, node** b )
{
    node* t = *a;
    *a = *b;
    *b = t;
}

/* A utility function to get size of linked list */
int getSize(struct node *node)
{
    int size = 0;
    while (node != NULL)
    {
        node = node->next;
        size++;
```

```
    }
    return size;
}

// Adds two linked lists of same size represented by head1 and head2 and returns
// head of the resultant linked list. Carry is propagated while returning from
// the recursion
node* addSameSize(node* head1, node* head2, int* carry)
{
    // Since the function assumes linked lists are of same size,
    // check any of the two head pointers
    if (head1 == NULL)
        return NULL;

    int sum;

    // Allocate memory for sum node of current two nodes
    node* result = (node *)malloc(sizeof(node));

    // Recursively add remaining nodes and get the carry
    result->next = addSameSize(head1->next, head2->next, carry);

    // add digits of current nodes and propagated carry
    sum = head1->data + head2->data + *carry;
    *carry = sum / 10;
    sum = sum % 10;

    // Assigne the sum to current node of resultant list
    result->data = sum;

    return result;
}

// This function is called after the smaller list is added to the bigger
// lists's sublist of same size.  Once the right sublist is added, the carry
// must be added toe left side of larger list to get the final result.
void addCarryToRemaining(node* head1, node* cur, int* carry, node** result)
{
    int sum;

    // If diff. number of nodes are not traversed, add carry
    if (head1 != cur)
    {
        addCarryToRemaining(head1->next, cur, carry, result);

        sum = head1->data + *carry;
        *carry = sum/10;
        sum %= 10;

        // add this node to the front of the result
        push(result, sum);
    }
}
```

```
// The main function that adds two linked lists represented by head1 and head2.
// The sum of two lists is stored in a list referred by result
void addList(node* head1, node* head2, node** result)
{
    node *cur;

    // first list is empty
    if (head1 == NULL)
    {
        *result = head2;
        return;
    }

    // second list is empty
    else if (head2 == NULL)
    {
        *result = head1;
        return;
    }

    int size1 = getSize(head1);
    int size2 = getSize(head2) ;

    int carry = 0;

    // Add same size lists
    if (size1 == size2)
        *result = addSameSize(head1, head2, &carry);

    else
    {
        int diff = abs(size1 - size2);

        // First list should always be larger than second list.
        // If not, swap pointers
        if (size1 < size2)
            swapPointer(&head1, &head2);

        // move diff. number of nodes in first list
        for (cur = head1; diff--; cur = cur->next);

        // get addition of same size lists
        *result = addSameSize(cur, head2, &carry);

        // get addition of remaining first list and carry
        addCarryToRemaining(head1, cur, &carry, result);
    }

    // if some carry is still there, add a new node to the fron of
    // the result list. e.g. 999 and 87
    if (carry)
        push(result, carry);
}
```

```
// Driver program to test above functions
int main()
{
    node *head1 = NULL, *head2 = NULL, *result = NULL;

    int arr1[] = {9, 9, 9};
    int arr2[] = {1, 8};

    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);

    // Create first list as 9->9->9
    int i;
    for (i = size1-1; i >= 0; --i)
        push(&head1, arr1[i]);

    // Create second list as 1->8
    for (i = size2-1; i >= 0; --i)
        push(&head2, arr2[i]);

    addList(head1, head2, &result);

    printList(result);

    return 0;
}
```

Output:

```
 1  0  1  7
```

Time Complexity: O(m+n) where m and n are the sizes of given two linked lists.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/sum-of-two-linked-lists/

Category: Linked Lists

Post navigation

← Microsoft Interview | Set 9 Output of C++ Program | Set 18 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 39

Sort a linked list of 0s, 1s and 2s

Given a linked list of 0s, 1s and 2s, sort it.

Source: Microsoft Interview | Set 1

Following steps can be used to sort the given linked list.
1) Traverse the list and count the number of 0s, 1s and 2s. Let the counts be n1, n2 and n3 respectively.
2) Traverse the list again, fill the first n1 nodes with 0, then n2 nodes with 1 and finally n3 nodes with 2.

```
// Program to sort a linked list 0s, 1s or 2s
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

// Function to sort a linked list of 0s, 1s and 2s
void sortList(struct node *head)
{
    int count[3] = {0, 0, 0};  // Initialize count of '0', '1' and '2' as 0
    struct node *ptr = head;

    /* count total number of '0', '1' and '2'
     * count[0] will store total number of '0's
     * count[1] will store total number of '1's
     * count[2] will store total number of '2's  */
    while (ptr != NULL)
    {
        count[ptr->data] += 1;
        ptr = ptr->next;
    }

    int i = 0;
    ptr = head;
```

```
    /* Let say count[0] = n1, count[1] = n2 and count[2] = n3
     * now start traversing list from head node,
     * 1) fill the list with 0, till n1 > 0
     * 2) fill the list with 1, till n2 > 0
     * 3) fill the list with 2, till n3 > 0  */
    while (ptr != NULL)
    {
        if (count[i] == 0)
            ++i;
        else
        {
            ptr->data = i;
            --count[i];
            ptr = ptr->next;
        }
    }
}

/* Function to push a node */
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Drier program to test above function*/
int main(void)
{
    struct node *head = NULL;
    push(&head, 0);
    push(&head, 1);
    push(&head, 0);
    push(&head, 2);
```

```
    push(&head, 1);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);
    push(&head, 2);

    printf("Linked List Before Sorting\n");
    printList(head);

    sortList(head);

    printf("Linked List After Sorting\n");
    printList(head);

    return 0;
}
```

Output:

```
 Linked List Before Sorting
2  1  2  1  1  2  0  1  0
Linked List After Sorting
0  0  1  1  1  1  2  2  2
```

Time Complexity: O(n)
Auxiliary Space: O(1)

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/sort-a-linked-list-of-0s-1s-or-2s/

Category: Linked Lists

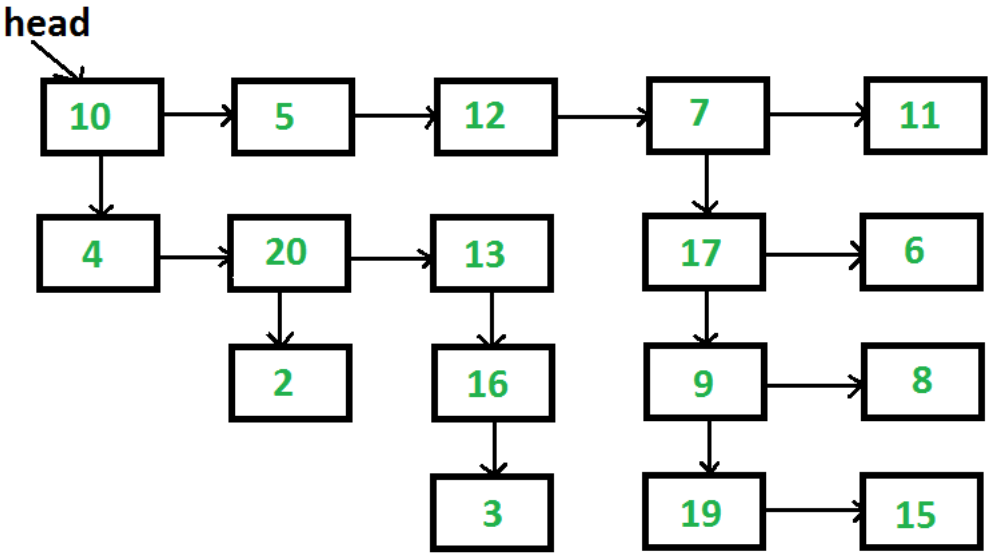
Chapter 40

Flatten a multilevel linked list

Given a linked list where in addition to the next pointer, each node has a child pointer, which may or may not point to a separate list. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure, as shown in below figure.You are given the head of the first level of the list. Flatten the list so that all the nodes appear in a single-level linked list. You need to flatten the list in way that all nodes at first level should come first, then nodes of second level, and so on.

Each node is a C struct with the following definition.

```
struct list
{
    int data;
    struct list *next;
    struct list *child;
};
```



The above list should be converted to 10->5->12->7->11->4->20->13->17->6->2->16->9->8->3->19->15

The problem clearly say that we need to flatten level by level. The idea of solution is, we start from first level, process all nodes one by one, if a node has a child, then we append the child at the end of list, otherwise we don't do anything. After the first level is processed, all next level nodes will be appended after first level. Same process is followed for the appended nodes.

```
1) Take "cur" pointer, which will point to head of the fist level of the list
2) Take "tail" pointer, which will point to end of the first level of the list
3) Repeat the below procedure while "curr" is not NULL.
    I) if current node has a child then
    a) append this new child list to the "tail"
        tail->next = cur->child
    b) find the last node of new child list and update "tail"
        tmp = cur->child;
        while (tmp->next != NULL)
            tmp = tmp->next;
        tail = tmp;
    II) move to the next node. i.e. cur = cur->next
```

Following is C implementation of the above algorithm.

```
// Program to flatten list with next and child pointers
#include <stdio.h>
#include <stdlib.h>

// Macro to find number of elements in array
#define SIZE(arr) (sizeof(arr)/sizeof(arr[0]))

// A linked list node has data, next pointer and child pointer
struct node
{
    int data;
    struct node *next;
    struct node *child;
};

// A utility function to create a linked list with n nodes. The data
// of nodes is taken from arr[].  All child pointers are set as NULL
struct node *createList(int *arr, int n)
{
    struct node *head = NULL;
    struct node *p;

    int i;
    for (i = 0; i < n; ++i) {
        if (head == NULL)
            head = p = (struct node *)malloc(sizeof(*p));
        else {
            p->next = (struct node *)malloc(sizeof(*p));
            p = p->next;
        }
```

```
        p->data = arr[i];
        p->next = p->child = NULL;
    }
    return head;
}

// A utility function to print all nodes of a linked list
void printList(struct node *head)
{
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

// This function creates the input list.  The created list is same
// as shown in the above figure
struct node *createList(void)
{
    int arr1[] = {10, 5, 12, 7, 11};
    int arr2[] = {4, 20, 13};
    int arr3[] = {17, 6};
    int arr4[] = {9, 8};
    int arr5[] = {19, 15};
    int arr6[] = {2};
    int arr7[] = {16};
    int arr8[] = {3};

    /* create 8 linked lists */
    struct node *head1 = createList(arr1, SIZE(arr1));
    struct node *head2 = createList(arr2, SIZE(arr2));
    struct node *head3 = createList(arr3, SIZE(arr3));
    struct node *head4 = createList(arr4, SIZE(arr4));
    struct node *head5 = createList(arr5, SIZE(arr5));
    struct node *head6 = createList(arr6, SIZE(arr6));
    struct node *head7 = createList(arr7, SIZE(arr7));
    struct node *head8 = createList(arr8, SIZE(arr8));


    /* modify child pointers to create the list shown above */
    head1->child = head2;
    head1->next->next->next->child = head3;
    head3->child = head4;
    head4->child = head5;
    head2->next->child = head6;
    head2->next->next->child = head7;
    head7->child = head8;


    /* Return head pointer of first linked list.  Note that all nodes are
       reachable from head1 */
    return head1;
}
```

```
/* The main function that flattens a multilevel linked list */
void flattenList(struct node *head)
{
    /*Base case*/
    if (head == NULL)
       return;

    struct node *tmp;

    /* Find tail node of first level linked list */
    struct node *tail = head;
    while (tail->next != NULL)
        tail = tail->next;

    // One by one traverse through all nodes of first level
    // linked list till we reach the tail node
    struct node *cur = head;
    while (cur != tail)
    {
        // If current node has a child
        if (cur->child)
        {
            // then append the child at the end of current list
            tail->next = cur->child;

            // and update the tail to new last node
            tmp = cur->child;
            while (tmp->next)
                tmp = tmp->next;
            tail = tmp;
        }

        // Change current node
        cur = cur->next;
    }
}

// A driver program to test above functions
int main(void)
{
    struct node *head = NULL;
    head = createList();
    flattenList(head);
    printList(head);
    return 0;
}
```

Output:

```
 10 5 12 7 11 4 20 13 17 6 2 16 9 8 3 19 15
```

Time Complexity: Since every node is visited at most twice, the time complexity is O(n) where n is the

number of nodes in given linked list.

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/flatten-a-linked-list-with-next-and-child-pointers/

Category: Linked Lists

Post navigation

← Microsoft Interview | 15 Iterative Postorder Traversal | Set 1 (Using Two Stacks) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 41

Delete N nodes after M nodes of a linked list

Given a linked list and two integers M and N. Traverse the linked list such that you retain M nodes then delete next N nodes, continue the same till end of the linked list.

Difficulty Level: Rookie

Examples:

```
Input:
M = 2, N = 2
Linked List: 1->2->3->4->5->6->7->8
Output:
Linked List: 1->2->5->6

Input:
M = 3, N = 2
Linked List: 1->2->3->4->5->6->7->8->9->10
Output:
Linked List: 1->2->3->6->7->8

Input:
M = 1, N = 1
Linked List: 1->2->3->4->5->6->7->8->9->10
Output:
Linked List: 1->3->5->7->9
```

The main part of the problem is to maintain proper links between nodes, make sure that all corner cases are handled. Following is C implementation of function skipMdeleteN() that skips M nodes and delete N nodes till end of list. It is assumed that M cannot be 0.

```
// C program to delete N nodes after M nodes of a linked list
#include <stdio.h>
#include <stdlib.h>

// A linked list node
```

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

/* Function to insert a node at the beginning */
void push(struct node ** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)  = new_node;
}

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Function to skip M nodes and then delete N nodes of the linked list.
void skipMdeleteN(struct node  *head, int M, int N)
{
    struct node *curr = head, *t;
    int count;

    // The main loop that traverses through the whole list
    while (curr)
    {
        // Skip M nodes
        for (count = 1; count<M && curr!= NULL; count++)
            curr = curr->next;

        // If we reached end of list, then return
        if (curr == NULL)
            return;

        // Start from next node and delete N nodes
        t = curr->next;
        for (count = 1; count<=N && t!= NULL; count++)
```

```
        {
            struct node *temp = t;
            t = t->next;
            free(temp);
        }
        curr->next = t; // Link the previous list with remaining nodes

        // Set current pointer for next iteration
        curr = t;
    }
}

// Driver program to test above functions
int main()
{
    /* Create following linked list
      1->2->3->4->5->6->7->8->9->10 */
    struct node* head = NULL;
    int M=2, N=3;
    push(&head, 10);
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("M = %d, N = %d \nGiven Linked list is :\n", M, N);
    printList(head);

    skipMdeleteN(head, M, N);

    printf("\nLinked list after deletion is :\n");
    printList(head);

    return 0;
}
```

Output:

```
 M = 2, N = 3
Given Linked list is :
1 2 3 4 5 6 7 8 9 10

Linked list after deletion is :
1 2 6 7
```

Time Complexity: O(n) where n is number of nodes in linked list.

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/delete-n-nodes-after-m-nodes-of-a-linked-list/

Category: Linked Lists

Chapter 42

QuickSort on Singly Linked List

QuickSort on Doubly Linked List is discussed here. QuickSort on Singly linked list was given as an exercise. Following is C++ implementation for same. The important things about implementation are, it changes pointers rather swapping data and time complexity is same as the implementation for Doubly Linked List.
In **partition()**, we consider last element as pivot. We traverse through the current list and if a node has value greater than pivot, we move it after tail. If the node has smaller value, we keep it at its current position.
In **QuickSortRecur()**, we first call partition() which places pivot at correct position and returns pivot. After pivot is placed at correct position, we find tail node of left side (list before pivot) and recur for left list. Finally, we recur for right list.

```
// C++ program for Quick Sort on Singly Linled List
#include <iostream>
#include <cstdio>
using namespace std;

/* a node of the singly linked list */
struct node
{
    int data;
    struct node *next;
};

/* A utility function to insert a node at the beginning of linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = new node;

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}
```

```
/* A utility function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
    printf("\n");
}

// Returns the last node of the list
struct node *getTail(struct node *cur)
{
    while (cur != NULL && cur->next != NULL)
        cur = cur->next;
    return cur;
}

// Partitions the list taking the last element as the pivot
struct node *partition(struct node *head, struct node *end,
                       struct node **newHead, struct node **newEnd)
{
    struct node *pivot = end;
    struct node *prev = NULL, *cur = head, *tail = pivot;

    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot)
    {
        if (cur->data < pivot->data)
        {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        }
        else // If cur node is greater than pivot
        {
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }
```

```
    // If the pivot data is the smallest element in the current list,
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;

    // Update newEnd to the current last node
    (*newEnd) = tail;

    // Return the pivot node
    return pivot;
}


//here the sorting happens exclusive of the end node
struct node *quickSortRecur(struct node *head, struct node *end)
{
    // base condition
    if (!head || head == end)
        return head;

    node *newHead = NULL, *newEnd = NULL;

    // Partition the list, newHead and newEnd will be updated
    // by the partition function
    struct node *pivot = partition(head, end, &newHead, &newEnd);

    // If pivot is the smallest element - no need to recur for
    // the left part.
    if (newHead != pivot)
    {
        // Set the node before the pivot node as NULL
        struct node *tmp = newHead;
        while (tmp->next != pivot)
            tmp = tmp->next;
        tmp->next = NULL;

        // Recur for the list before pivot
        newHead = quickSortRecur(newHead, tmp);

        // Change next of last node of the left half to pivot
        tmp = getTail(newHead);
        tmp->next =  pivot;
    }

    // Recur for the list after the pivot element
    pivot->next = quickSortRecur(pivot->next, newEnd);

    return newHead;
}

// The main function for quick sort. This is a wrapper over recursive
// function quickSortRecur()
void quickSort(struct node **headRef)
{
```

```
    (*headRef) = quickSortRecur(*headRef, getTail(*headRef));
    return;
}

// Driver program to test above functions
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(&a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

Output:

```
 Linked List before sorting
30  3  4  20  5
Linked List after sorting
3  4  5  20  30
```

This article is contributed by **Balasubramanian.N**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/quicksort-on-singly-linked-list/

Category: Linked Lists

Post navigation

← [TopTalent.in] Interview with Manpreet Who Got Offers From Amazon, Hoppr, Browserstack, Reliance via TopTalent.in Print all possible strings of length k that can be formed from a set of n characters →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 43

Merge a linked list into another linked list at alternate positions

Given two linked lists, insert nodes of second list into first list at alternate positions of first list.
For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty. The nodes of second list should only be inserted when there are positions available. For example, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Use of extra space is not allowed (Not allowed to create additional nodes), i.e., insertion must be done in-place. Expected time complexity is O(n) where n is number of nodes in first list.

The idea is to run a loop while there are available positions in first loop and insert nodes of second list by changing pointers. Following is C implementation of this approach.

```
// C implementation of above program.
#include <stdio.h>
#include <stdlib.h>

// A nexted list node
struct node
{
    int data;
    struct node *next;
};

/* Function to insert a node at the beginning */
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;
}

/* Utility function to print a singly linked list */
void printList(struct node *head)
{
```

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

// Main function that inserts nodes of linked list q into p at alternate
// positions. Since head of first list never changes and head of second list
// may change, we need single pointer for first list and double pointer for
// second list.
void merge(struct node *p, struct node **q)
{
     struct node *p_curr = p, *q_curr = *q;
     struct node *p_next, *q_next;

     // While therre are avialable positions in p
     while (p_curr != NULL && q_curr != NULL)
     {
         // Save next pointers
         p_next = p_curr->next;
         q_next = q_curr->next;

         // Make q_curr as next of p_curr
         q_curr->next = p_next;  // Change next pointer of q_curr
         p_curr->next = q_curr;  // Change next pointer of p_curr

         // Update current pointers for next iteration
         p_curr = p_next;
         q_curr = q_next;
    }

    *q = q_curr; // Update head pointer of second list
}

// Driver program to test above functions
int main()
{
     struct node *p = NULL, *q = NULL;
     push(&p, 3);
     push(&p, 2);
     push(&p, 1);
     printf("First Linked List:\n");
     printList(p);

     push(&q, 8);
     push(&q, 7);
     push(&q, 6);
     push(&q, 5);
     push(&q, 4);
     printf("Second Linked List:\n");
     printList(q);
```

```
    merge(p, &q);

    printf("Modified First Linked List:\n");
    printList(p);

    printf("Modified Second Linked List:\n");
    printList(q);

    getchar();
    return 0;
}
```

Output:

```
 First Linked List:
1 2 3
Second Linked List:
4 5 6 7 8
Modified First Linked List:
1 4 2 5 3 6
Modified Second Linked List:
7 8
```

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/merge-a-linked-list-into-another-linked-list-at-alternate-positions/

Category: Linked Lists

Chapter 44

Pairwise swap elements of a given linked list by changing links

Given a singly linked list, write a function to swap elements pairwise. For example, if the linked list is 1->2->3->4->5->6->7 then the function should change it to 2->1->4->3->6->5->7, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5

This problem has been discussed here. The solution provided there swaps data of nodes. If data contains many fields, there will be many swap operations. So changing links is a better idea in general. Following is a C implementation that changes links instead of swapping data.

```
/* This program swaps the nodes of linked list rather than swapping the
field from the nodes.
Imagine a case where a node contains many fields, there will be plenty
of unnecessary swap calls. */

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

/* A linked list node */
struct node
{
    int data;
    struct node *next;
};

/* Function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node **head)
{
    // If linked list is empty or there is only one node in list
    if (*head == NULL || (*head)->next == NULL)
        return;

    // Initialize previous and current pointers
    struct node *prev = *head;
    struct node *curr = (*head)->next;
```

```
    *head = curr;  // Change head before proceeeding

    // Traverse the list
    while (true)
    {
        struct node *next = curr->next;
        curr->next = prev; // Change next of current as previous node

        // If next NULL or next is the last node
        if (next == NULL || next->next == NULL)
        {
            prev->next = next;
            break;
        }

        // Change next of previous to next next
        prev->next = next->next;

        // Update previous and curr
        prev = next;
        curr = prev->next;
    }
}

/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Druver program to test above function */
int main()
{
```

```
    struct node *start = NULL;

    /* The constructed linked list is:
     1->2->3->4->5->6->7 */
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before calling  pairWiseSwap() ");
    printList(start);

    pairWiseSwap(&start);

    printf("\n Linked list after calling  pairWiseSwap() ");
    printList(start);

    getchar();
    return 0;
}
```

Output:

```
 Linked list before calling  pairWiseSwap() 1 2 3 4 5 6 7
 Linked list after calling  pairWiseSwap() 2 1 4 3 6 5 7
```

Time Complexity: Time complexity of the above program is O(n) where n is the number of nodes in a given linked list. The while loop does a traversal of the given linked list.

Following is **recursive implementation** of the same approach. We change first two nodes and recur for the remaining list. Thanks to geek and omer salem for suggesting this method.

```
/* This program swaps the nodes of linked list rather than swapping the
field from the nodes.
Imagine a case where a node contains many fields, there will be plenty
of unnecessary swap calls. */

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

/* A linked list node */
struct node
{
    int data;
    struct node *next;
};
```

```
/* Function to pairwise swap elements of a linked list.
   It returns head of the modified list, so return value
   of this node must be assigned */
struct node *pairWiseSwap(struct node* head)
{
    // Base Case: The list is empty or has only one node
    if (head == NULL || head->next == NULL)
        return head;

    // Store head of list after two nodes
    struct node* remaing = head->next->next;

    // Change head
    struct node* newhead = head->next;

    // Change next of second node
    head->next->next = head;

    // Recur for remaining list and change next of head
    head->next = pairWiseSwap(remaing);

    // Return new head of modified list
    return newhead;
}

/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Druver program to test above function */
int main()
```

```
{
    struct node *start = NULL;

    /* The constructed linked list is:
     1->2->3->4->5->6->7 */
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before calling  pairWiseSwap() ");
    printList(start);

    start = pairWiseSwap(start);  // NOTE THIS CHANGE

    printf("\n Linked list after calling  pairWiseSwap() ");
    printList(start);

    return 0;
}
```

```
 Linked list before calling  pairWiseSwap() 1 2 3 4 5 6 7
 Linked list after calling  pairWiseSwap() 2 1 4 3 6 5 7
```

This article is contributed by **Gautam Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/pairwise-swap-elements-of-a-given-linked-list-by-changing-links/

Category: Linked Lists Tags: Linked Lists

Chapter 45

Given a linked list of line segments, remove middle points

Given a linked list of co-ordinates where adjacent points either form a vertical line or a horizontal line. Delete points from the linked list which are in the middle of a horizontal or vertical line.

Examples:

```
Input:  (0,10)->(1,10)->(5,10)->(7,10)
                                  |
                                (7,5)->(20,5)->(40,5)
Output: Linked List should be changed to following
        (0,10)->(7,10)
                  |
                (7,5)->(40,5)
The given linked list represents a horizontal line from (0,10)
to (7, 10) followed by a vertical line from (7, 10) to (7, 5),
followed by a horizontal line from (7, 5) to (40, 5).

Input:     (2,3)->(4,3)->(6,3)->(10,3)->(12,3)
Output: Linked List should be changed to following
    (2,3)->(12,3)
There is only one vertical line, so all middle points are removed.
```

Source: Microsoft Interview Experience

We strongly recommend to minimize the browser and try this yourself first.

The idea is to keep track of current node, next node and next-next node. While the next node is same as next-next node, keep deleting the next node. In this complete procedure we need to keep an eye on shifting of pointers and checking for NULL values.

Following is C implementation of above idea.

```
// C program to remove intermediate points in a linked list that represents
// horizontal and vertical line segments
#include <stdio.h>
```

```
#include <stdlib.h>

// Node has 3 fields including x, y coordinates and a pointer to next node
struct node
{
    int x, y;
    struct node *next;
};

/* Function to insert a node at the beginning */
void push(struct node ** head_ref, int x,int y)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->x  = x;
    new_node->y  = y;
    new_node->next = (*head_ref);
    (*head_ref)  = new_node;
}

/* Utility function to print a singly linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while (temp != NULL)
    {
        printf("(%d,%d)-> ", temp->x,temp->y);
        temp = temp->next;
    }
    printf("\n");

}

// Utility function to remove Next from linked list and link nodes
// after it to head
void deleteNode(struct node *head, struct node *Next)
{
    head->next = Next->next;
    Next->next = NULL;
    free(Next);
}

// This function deletes middle nodes in a sequence of horizontal and
// vertical line segments represented by linked list.
struct node* deleteMiddle(struct node *head)
{
    // If only one node or no node...Return back
    if (head==NULL || head->next ==NULL || head->next->next==NULL)
        return head;

    struct node* Next = head->next;
    struct node *NextNext = Next->next ;

    // Check if this is a vertical line or horizontal line
    if (head->x == Next->x)
```

```
    {
        // Find middle nodes with same x value, and delete them
        while (NextNext !=NULL && Next->x==NextNext->x)
        {
            deleteNode(head, Next);

            // Update Next and NextNext for next iteration
            Next = NextNext;
            NextNext = NextNext->next;
        }
    }
    else if (head->y==Next->y) // If horizontal line
    {
        // Find middle nodes with same y value, and delete them
        while (NextNext !=NULL && Next->y==NextNext->y)
        {
            deleteNode(head, Next);

            // Update Next and NextNext for next iteration
            Next = NextNext;
            NextNext = NextNext->next;
        }
    }
    else  // Adjacent points must have either same x or same y
    {
        puts("Given linked list is not valid");
        return NULL;
    }

    // Recur for next segment
    deleteMiddle(head->next);

    return head;
}

// Driver program to tsst above functions
int main()
{
    struct node *head = NULL;

    push(&head, 40,5);
    push(&head, 20,5);
    push(&head, 10,5);
    push(&head, 10,8);
    push(&head, 10,10);
    push(&head, 3,10);
    push(&head, 1,10);
    push(&head, 0,10);
    printf("Given Linked List: \n");
    printList(head);

    if (deleteMiddle(head) != NULL);
    {
        printf("Modified Linked List: \n");
```

```
        printList(head);
    }
    return 0;
}
```

Output:

```
 Given Linked List:
(0,10)-> (1,10)-> (3,10)-> (10,10)-> (10,8)-> (10,5)-> (20,5)-> (40,5)->
Modified Linked List:
(0,10)-> (10,10)-> (10,5)-> (40,5)->
```

Time Complexity of the above solution is O(n) where n is number of nodes in given linked list.

Exercise:
The above code is recursive, write an iterative code for the same problem.

This article is contributed by Sanket Jain. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/given-linked-list-line-segments-remove-middle-points/

Category: Linked Lists

Post navigation

← Given n appointments, find all conflicting appointments Bharti SoftBank (Portal Team) Interview Experience (Off-Campus) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 46

Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes

Given two sorted linked lists, construct a linked list that contains maximum sum path from start to end. The result list may contain nodes from both input lists. When constructing the result list, we may switch to the other input list only at the point of intersection (which mean the two node with the same value in the lists). You are allowed to use O(1) extra space.

```
Input:
List1 =  1->3->30->90->120->240->511
List2 =  0->3->12->32->90->125->240->249

Output: Following is maximum sum linked list out of two input lists
list =  1->3->12->32->90->125->240->511
we switch at 3 and 240 to get above maximum sum linked list
```

We strongly recommend to minimize the browser and try this yourself first.

The idea here in the below solution is to adjust next pointers after common nodes.
1. Start with head of both linked lists and find first common node. Use merging technique of sorted linked list for that.
2. Keep track of sum of the elements too while doing this and set head of result list based on greater sum till first common node.
3. After this till the current pointers of both lists don't become NULL we need to adjust the next of prev pointers based on greater sum.

This way it can be done in-place with constant extra space.
Time complexity of the below solution is O(n).

```
// C++ program to construct the maximum sum linked
// list out of two given sorted lists
#include<iostream>
using namespace std;
```

```
//A linked list node
struct Node
{
    int data; //data belong to that node
    Node *next; //next pointer
};

// Push the data to the head of the linked list
void push(Node **head, int data)
{
    //Alocation memory to the new node
    Node *newnode = new Node;

    //Assigning data to the new node
    newnode->data = data;

    //Adjusting next pointer of the new node
    newnode->next = *head;

    //New node becomes the head of the list
    *head = newnode;
}

// Method that adjusts the pointers and prints the final list
void finalMaxSumList(Node *a, Node *b)
{
    Node *result = NULL;

    // Assigning pre and cur to the head of the
    // linked list.
    Node *pre1 = a, *curr1 = a;
    Node *pre2 = b, *curr2 = b;

    // Till either of the current pointers is not
    // NULL execute the loop
    while (curr1 != NULL || curr2 != NULL)
    {
        // Keeping 2 local variables at the start of every
        // loop run to keep track of the sum between pre
        // and cur pointer elements.
        int sum1 = 0, sum2 = 0;

        // Calculating sum by traversing the nodes of linked
        // list as the merging of two linked list.  The loop
        // stops at a common node
        while (curr1!=NULL && curr2!=NULL && curr1->data!=curr2->data)
        {
            if (curr1->data < curr2->data)
            {
                sum1 += curr1->data;
                curr1 = curr1->next;
            }
            else // (curr2->data < curr1->data)
```

```
        {
            sum2 += curr2->data;
            curr2 = curr2->next;
        }
    }

    // If either of current pointers becomes NULL
    // carry on the sum calculation for other one.
    if (curr1 == NULL)
    {
        while (curr2 != NULL)
        {
            sum2 += curr2->data;
            curr2 = curr2->next;
        }
    }
    if (curr2 == NULL)
    {
        while (curr1 != NULL)
        {
            sum1 += curr1->data;
            curr1 = curr1->next;
        }
    }

    // First time adjustment of resultant head based on
    // the maximum sum.
    if (pre1 == a && pre2 == b)
        result = (sum1 > sum2)? pre1 : pre2;

    // If pre1 and pre2 don't contain the head pointers of
    // lists adjust the next pointers of previous pointers.
    else
    {
        if (sum1 > sum2)
            pre2->next = pre1->next;
        else
            pre1->next = pre2->next;
    }

    // Adjusting previous pointers
    pre1 = curr1, pre2 = curr2;

    // If curr1 is not NULL move to the next.
    if (curr1)
        curr1 = curr1->next;
    // If curr2 is not NULL move to the next.
    if (curr2)
        curr2 = curr2->next;
}

// Print the resultant list.
while (result != NULL)
{
```

```
        cout << result->data << " ";
        result = result->next;
    }
}

//Main driver program
int main()
{
    //Linked List 1 : 1->3->30->90->110->120->NULL
    //Linked List 2 : 0->3->12->32->90->100->120->130->NULL
    Node *head1 = NULL, *head2 = NULL;
    push(&head1, 120);
    push(&head1, 110);
    push(&head1, 90);
    push(&head1, 30);
    push(&head1, 3);
    push(&head1, 1);

    push(&head2, 130);
    push(&head2, 120);
    push(&head2, 100);
    push(&head2, 90);
    push(&head2, 32);
    push(&head2, 12);
    push(&head2, 3);
    push(&head2, 0);

    finalMaxSumList(head1, head2);
    return 0;
}
```

Output:

```
1 3 12 32 90 110 120 130
```

Time complexity = O(n) where n is the length of bigger linked list
Auxiliary space = O(1)
However a problem in this solution is that the original lists are changed.

Exercise
1. Try this problem when auxiliary space is not a constraint.
2. Try this problem when we don't modify the actual list and create the resultant list.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/maximum-sum-linked-list-two-sorted-linked-lists-common-nodes/

Category: Linked Lists

Chapter 47

Clone a linked list with next and random pointer | Set 2

We have already discussed 2 different ways to clone a linked list. In thispost, one more simple method to clone a linked list is discussed.

The idea is to use Hashing. Below is algorithm.
1. Traverse the original linked list and make a copy in terms of data.
2. Make a hash map of key value pair with original linked list node and copied linked list node.
3. Traverse the original linked list again and using the hash map adjust the next and random reference of cloned linked list nodes.

A Java based approach of the above idea is below.

```
// Java program to clone a linked list with random pointers
import java.util.HashMap;
import java.util.Map;

// Linked List Node class
class Node
{
    int data;//Node data
    Node next, random;//Next and random reference

    //Node constructor
    public Node(int data)
    {
        this.data = data;
        this.next = this.random = null;
    }
}

// linked list class
class LinkedList
{
    Node head;//Linked list head reference

    // Linked list constructor
```

```
public LinkedList(Node head)
{
    this.head = head;
}

// push method to put data always at the head
// in the linked list.
public void push(int data)
{
    Node node = new Node(data);
    node.next = this.head;
    this.head = node;
}

// Method to print the list.
void print()
{
    Node temp = head;
    while (temp != null)
    {
        Node random = temp.random;
        int randomData = (random != null)? random.data: -1;
        System.out.println("Data = " + temp.data +
                           ", Random data = "+ randomData);
        temp = temp.next;
    }
}

// Actual clone method which returns head
// reference of cloned linked list.
public LinkedList clone()
{
    // Initialize two references, one with original
    // list's head.
    Node origCurr = this.head, cloneCurr = null;

    // Hash map which contains node to node mapping of
    // original and clone linked list.
    Map<Node, Node> map = new HashMap<Node, Node>();

    // Traverse the original list and make a copy of that
    // in the clone linked list.
    while (origCurr != null)
    {
        cloneCurr = new Node(origCurr.data);
        map.put(origCurr, cloneCurr);
        origCurr = origCurr.next;
    }

    // Adjusting the original list reference again.
    origCurr = this.head;

    // Traversal of original list again to adjust the next
    // and random references of clone list using hash map.
```

```
        while (origCurr != null)
        {
            cloneCurr = map.get(origCurr);
            cloneCurr.next = map.get(origCurr.next);
            cloneCurr.random = map.get(origCurr.random);
            origCurr = origCurr.next;
        }

        //return the head reference of the clone list.
        return new LinkedList(map.get(this.head));
    }
}

// Driver Class
class Main
{
    // Main method.
    public static void main(String[] args)
    {
        // Pushing data in the linked list.
        LinkedList list = new LinkedList(new Node(5));
        list.push(4);
        list.push(3);
        list.push(2);
        list.push(1);

        // Setting up random references.
        list.head.random = list.head.next.next;
        list.head.next.random =
            list.head.next.next.next;
        list.head.next.next.random =
            list.head.next.next.next.next;
        list.head.next.next.next.random =
            list.head.next.next.next.next.next;
        list.head.next.next.next.next.random =
            list.head.next;

        // Making a clone of the original linked list.
        LinkedList clone = list.clone();

        // Print the original and cloned linked list.
        System.out.println("Original linked list");
        list.print();
        System.out.println("\nCloned linked list");
        clone.print();
    }
}
```

Output:

```
Original linked list
Data = 1, Random data = 3
Data = 2, Random data = 4
```

```
Data = 3, Random data = 5
Data = 4, Random data = -1
Data = 5, Random data = 2

Cloned linked list
Data = 1, Random data = 3
Data = 2, Random data = 4
Data = 3, Random data = 5
Data = 4, Random data = -1
Data = 5, Random data = 2
```

Time complexity : O(n)
Auxiliary space : O(n)

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

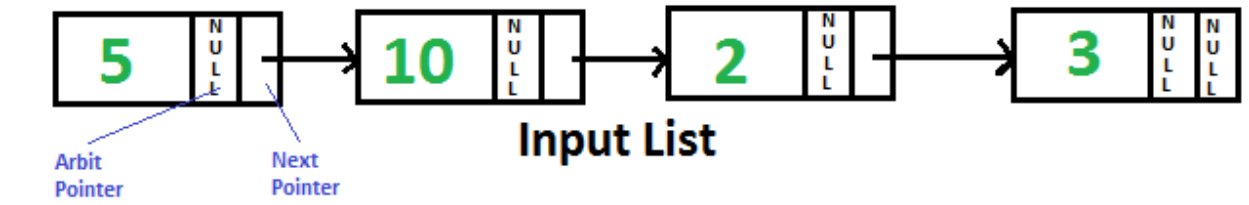
http://www.geeksforgeeks.org/clone-linked-list-next-arbit-pointer-set-2/

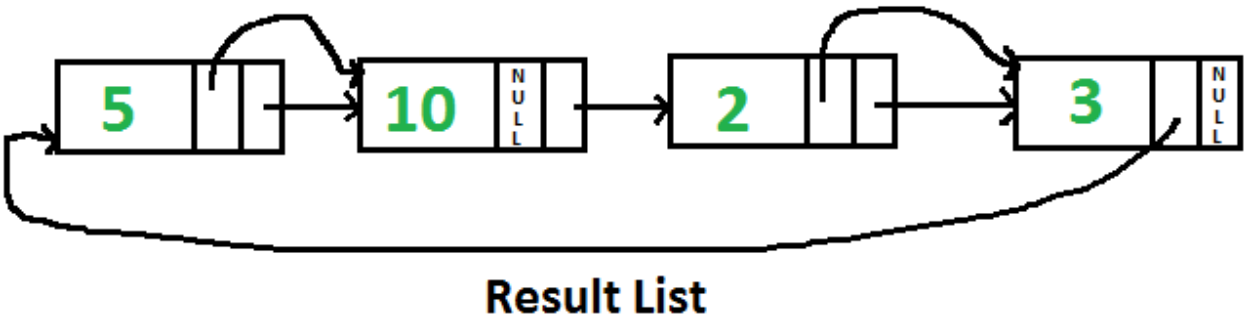
Category: Linked Lists

Chapter 48

Point to next higher value node in a linked list with an arbitrary pointer

Given singly linked list with every node having an additional "arbitrary" pointer that currently points to NULL. Need to make the "arbitrary" pointer point to the next higher value node.





We strongly recommend to minimize your browser and try this yourself first

A **Simple Solution** is to traverse all nodes one by one, for every node, find the node which has next greater value of current node and change the next pointer. Time Complexity of this solution is $O(n^2$.

An **Efficient Solution** works in O(nLogn) time. The idea is to use Merge Sort for linked list.
1) Traverse input list and copy next pointer to arbit pointer for every node.
2) Do Merge Sort for the linked list formed by arbit pointers.

Below is C implementation of above idea. All of the merger sort functions are taken from here. The taken functions are modified here so that they work on arbit pointers instead of next pointers.

```
// C program to populate arbit pointers to next higher value
// using merge sort
```

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

/* Link list node */
struct node
{
    int data;
    struct node* next, *arbit;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef);

/* sorts the linked list formed by arbit pointers
  (does not change next pointer or data) */
void MergeSort(struct node** headRef)
{
    struct node* head = *headRef;
    struct node* a, *b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->arbit == NULL))
        return;

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
   function */
struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if (b==NULL)
        return (a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->arbit = SortedMerge(a->arbit, b);
```

```
    }
    else
    {
        result = b;
        result->arbit = SortedMerge(a, b->arbit);
    }

    return (result);
}

/* Split the nodes of the given list into front and back halves,
   and return the two lists using the reference parameters.
   If the length is odd, the extra node should go in the front list.
   Uses the fast/slow pointer strategy.  */
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef)
{
    struct node* fast, *slow;

    if (source==NULL || source->arbit==NULL)
    {
        /* length < 2 cases */
        *frontRef = source;
        *backRef = NULL;
        return;
    }

    slow = source,  fast = source->arbit;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL)
    {
        fast = fast->arbit;
        if (fast != NULL)
        {
            slow = slow->arbit;
            fast = fast->arbit;
        }
    }

    /* 'slow' is before the midpoint in the list, so split it in two
      at that point. */
    *frontRef = source;
    *backRef = slow->arbit;
    slow->arbit = NULL;
}

/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));
```

```
    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    new_node->arbit = NULL;

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

// Utility function to print result linked list
void printListafter(struct node *node, struct node *anode)
{
    printf("Traversal using Next Pointer\n");
    while (node!=NULL)
    {
        printf("%d, ", node->data);
        node = node->next;
    }

    printf("\nTraversal using Arbit Pointer\n");
    while (anode!=NULL)
    {
        printf("%d, ", anode->data);
        anode = anode->arbit;
    }
}

// This function populates arbit pointer in every node to the
// next higher value. And returns pointer to the node with
// minimum value
struct node* populateArbit(struct node *head)
{
    // Copy next pointers to arbit pointers
    struct node *temp = head;
    while (temp != NULL)
    {
        temp->arbit = temp->next;
        temp = temp->next;
    }

    // Do merge sort for arbitrary pointers
    MergeSort(&head);

    // Return head of arbitrary pointer linked list
    return head;
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
```

```
    struct node* head = NULL;

    /* Let us create the list shown above */
    push(&head, 3);
    push(&head, 2);
    push(&head, 10);
    push(&head, 5);

    /* Sort the above created Linked List */
    struct node *ahead = populateArbit(head);

    printf("\nResult Linked List is: \n");
    printListafter(head, ahead);

    getchar();
    return 0;
}
```

Output:

```
Result Linked List is:
Traversal using Next Pointer
5, 10, 2, 3,
Traversal using Arbit Pointer
2, 3, 5, 10,
```

This article is contributed by **Saurabh Bansal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/point-to-next-higher-value-node-in-a-linked-list-with-an-arbitrary-pointer/

Category: Linked Lists

Post navigation

← Pythagorean Triplet in an array Flipkart Interview Experience | Set 22 (For SDE 2) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 49

Rearrange a given linked list in-place.

Given a singly linked list L_0 -> L_1 -> … -> L_{n-1} -> L_n. Rearrange the nodes in the list so that the new formed list is : L_0 -> L_n -> L_1 -> L_{n-1} -> L_2 -> L_{n-2} …

You are required do this in-place without altering the nodes' values.

```
Examples:
Input:  1 -> 2 -> 3 -> 4
Output: 1 -> 4 -> 2 -> 3

Input:  1 -> 2 -> 3 -> 4 -> 5
Output: 1 -> 5 -> 2 -> 4 -> 3
```

We strongly recommend you to minimize your browser and try this yourself first.

Simple Solution

```
1) Initialize current node as head.
2) While next of current node is not null, do following
    a) Find the last node, remove it from end and insert it as next
       of current node.
    b) Move current to next to next of current
```

Time complexity of the above simple solution is $O(n^2)$ where n is number of nodes in linked list.

Efficient Solution:

```
1) Find the middle point using tortoise and hare method.
2) Split the linked list in two halves using found middle point in step 1.
3) Reverse the second half.
4) Do alternate merge of first and second halves.
```

Time Complexity of this solution is O(n).

Below is C++ implementation of this method.

```
// C++ program to rearrange a linked list in-place
#include<bits/stdc++.h>
using namespace std;

// Linkedlist Node structure
struct Node
{
    int data;
    struct Node *next;
};

// Function to create newNode in a linkedlist
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// Function to reverse the linked list
void reverselist(Node **head)
{
    // Initialize prev and current pointers
    Node *prev = NULL, *curr = *head, *next;

    while (curr)
    {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }

    *head = prev;
}

// Function to print the linked list
void printlist(Node *head)
{
    while (head != NULL)
    {
        cout << head->data << " ";
        if(head->next) cout << "-> ";
        head = head->next;
    }
    cout << endl;
}

// Function to rearrange a linked list
void rearrange(Node **head)
{
    // 1) Find the muddle point using tortoise and hare method
```

```
    Node *slow = *head, *fast = slow->next;
    while (fast && fast->next)
    {
        slow = slow->next;
        fast = fast->next->next;
    }

    // 2) Split the linked list in two halves
    // head1, head of first half    1 -> 2
    // head2, head of second half   3 -> 4
    Node *head1 = *head;
    Node *head2 = slow->next;
    slow->next = NULL;

    // 3) Reverse the second half, i.e.,  4 -> 3
    reverselist(&head2);

    // 4) Merge alternate nodes
    *head = newNode(0); // Assign dummy Node

    // curr is the pointer to this dummy Node, which will
    // be used to form the new list
    Node *curr = *head;
    while (head1 || head2)
    {
        // First add the element from list
        if (head1)
        {
            curr->next = head1;
            curr = curr->next;
            head1 = head1->next;
        }

        // Then add the element from second list
        if (head2)
        {
            curr->next = head2;
            curr = curr->next;
            head2 = head2->next;
        }
    }

    // Assign the head of the new list to head pointer
    *head = (*head)->next;
}

// Driver program
int main()
{
    Node *head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);
```

```
    printlist(head);    // Print original list
    rearrange(&head);       // Modify the list
    printlist(head);    // Print modified list
    return 0;
}
```

Output:

```
1 -> 2 -> 3 -> 4 -> 5
1 -> 5 -> 2 -> 4 -> 3
```

Thanks to Gaurav Ahirwar for suggesting above approach.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/rearrange-a-given-linked-list-in-place/

Category: Linked Lists

Chapter 50

Sort a linked list that is sorted alternating ascending and descending orders?

Given a Linked List. The Linked List is in alternating ascending and descending orders. Sort the list efficiently.

Example:

```
 Input List:   10->40->53->30->67->12->89->NULL
Output List:  10->12->30->43->53->67->89->NULL
```

Source: http://qa.geeksforgeeks.org/616/linked-that-sorted-alternating-ascending-descending-orders

We strongly recommend you to minimize your browser and try this yourself first.

A **Simple Solution** is to use Merge Sort for linked List. This solution takes O(n Log n) time.

An **Efficient Solution** works in O(n) time. Below are all steps.
1. Separate two lists.
2. Reverse the one with descending order
3. Merge both lists. Below is C++ implementation based on above idea.

Below is C++ implementation of above algorithm.

```
// Sort a linked list that is alternatively sorted
// in increasing and decreasing order
#include<bits/stdc++.h>
using namespace std;

// Linked list node
struct Node
{
    int data;
    struct Node *next;
};

Node *mergelist(Node *head1, Node *head2);
```

```
void splitList(Node *head, Node **Ahead, Node **Dhead);
void reverselist(Node *&head);

// This is the main function that sorts the
// linked list
void sort(Node **head)
{
    // Split the list into lists
    Node *Ahead, *Dhead;
    splitList(*head, &Ahead, &Dhead);

    // Reverse the descending linked list
    reverselist(Dhead);

    // Merge the two linked lists
    *head = mergelist(Ahead, Dhead);
}

// A utility function to create a new node
Node* newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to reverse a linked list
void reverselist(Node *&head)
{
    Node* prev = NULL, *curr = head, *next;
    while (curr)
    {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    head = prev;
}

// A utility function to print a linked list
void printlist(Node *head)
{
    while (head != NULL)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

// A utility function to merge two sorted linked lists
Node *mergelist(Node *head1, Node *head2)
```

```
{
    // Base cases
    if (!head1) return head2;
    if (!head2) return head1;

    Node *temp = NULL;
    if (head1->data < head2->data)
    {
        temp = head1;
        head1->next = mergelist(head1->next, head2);
    }
    else
    {
        temp = head2;
        head2->next = mergelist(head1, head2->next);
    }
    return temp;
}

// This function alternatively splits a linked list with head
// as head into two:
// For example, 10->20->30->15->40->7 is splitted into 10->30->40
// and 20->15->7
// "Ahead" is reference to head of ascending linked list
// "Dhead" is reference to head of descending linked list
void splitList(Node *head, Node **Ahead, Node **Dhead)
{
    // Create two dummy nodes to initialize heads of two linked list
    *Ahead = newNode(0);
    *Dhead = newNode(0);

    Node *ascn = *Ahead;
    Node *dscn = *Dhead;
    Node *curr = head;

    // Link alternate nodes
    while (curr)
    {
        // Link alternate nodes of ascending linked list
        ascn->next = curr;
        ascn = ascn->next;
        curr = curr->next;

        // Link alternate nodes of descending linked list
        if (curr)
        {
            dscn->next = curr;
            dscn = dscn->next;
            curr = curr->next;
        }
    }

    ascn->next = NULL;
    dscn->next = NULL;
```

```
    *Ahead = (*Ahead)->next;
    *Dhead = (*Dhead)->next;
}

// Driver program to test above function
int main()
{
    Node *head = newNode(10);
    head->next = newNode(40);
    head->next->next = newNode(53);
    head->next->next->next = newNode(30);
    head->next->next->next->next = newNode(67);
    head->next->next->next->next->next = newNode(12);
    head->next->next->next->next->next->next = newNode(89);

    cout << "Given Linked List is " << endl;
    printlist(head);

    sort(&head);

    cout << "\nSorted Linked List is " << endl;
    printlist(head);

    return 0;
}
```

Output:

```
 Given Linked List is
10 40 53 30 67 12 89

Sorted Linked List is
10 12 30 40 53 67 89
```

Thanks to Gaurav Ahirwar for suggesting this method here.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/how-to-sort-a-linked-list-that-is-sorted-alternating-ascending-and-descending-orders/

Category: Linked Lists

Chapter 51

Select a Random Node from a Singly Linked List

Given a singly linked list, select a random node from linked list (the probability of picking a node should be 1/N if there are N nodes in list). You are given a random number generator.

Below is a Simple Solution
1) Count number of nodes by traversing the list.
2) Traverse the list again and select every node with probability 1/N. The selection can be done by generating a random number from 0 to N-i for i'th node, and selecting the i'th node node only if generated number is equal to 0 (or any other fixed number from 0 to N-i).

We get uniform probabilities with above schemes.

```
i = 1, probability of selecting first node = 1/N
i = 2, probability of selecting second node =
                   [probability that first node is not selected] *
                   [probability that second node is selected]
                  = ((N-1)/N)* 1/(N-1)
                  = 1/N
```

Similarly, probabilities of other selecting other nodes is 1/N

The above solution requires two traversals of linked list.

How to select a random node with only one traversal allowed?
The idea is to use Reservoir Sampling. Following are the steps. This is a simpler version of Reservoir Sampling as we need to select only one key instead of k keys.

```
(1) Initialize result as first node
   result = head->key
(2) Initialize n = 2
(3) Now one by one consider all nodes from 2nd node onward.
    (3.a) Generate a random number from 0 to n-1.
         Let the generated random number is j.
    (3.b) If j is equal to 0 (we could choose other fixed number
          between 0 to n-1), then replace result with current node.
    (3.c) n = n+1
    (3.d) current = current->next
```

Below is C implementation of above algorithm.

```
/* C program to randomly select a node from a singly
   linked list */
#include<stdio.h>
#include<stdlib.h>
#include <time.h>

/* Link list node */
struct node
{
    int key;
    struct node* next;
};

// A reservoir sampling based function to print a
// random node from a linked list
void printRandom(struct node *head)
{
    // IF list is empty
    if (head == NULL)
       return;

    // Use a different seed value so that we don't get
    // same result each time we run this program
    srand(time(NULL));

    // Initialize result as first node
    int result = head->key;

    // Iterate from the (k+1)th element to nth element
    struct node *current = head;
    int n;
    for (n=2; current!=NULL; n++)
    {
        // change result with probability 1/n
        if (rand() % n == 0)
           result = current->key;

        // Move to next node
        current = current->next;
    }

    printf("Randomly selected key is %d\n", result);
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST  */

/* A utility function to create a new node */
struct node *newNode(int new_key)
{
    /* allocate node */
    struct node* new_node =
```

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

    /* put in the key  */
    new_node->key  = new_key;
    new_node->next =  NULL;

    return new_node;
}


/* A utility function to insert a node at the beginning
  of linked list */
void push(struct node** head_ref, int new_key)
{
    /* allocate node */
    struct node* new_node = new node;

    /* put in the key  */
    new_node->key  = new_key;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}


// Driver program to test above functions
int main()
{
    struct node *head = NULL;
    push(&head, 5);
    push(&head, 20);
    push(&head, 4);
    push(&head, 3);
    push(&head, 30);

    printRandom(head);

    return 0;
}


 Randomly selected key is 4
```

Note that the above program is based on outcome of a random function and may produce different output.

How does this work?
Let there be total N nodes in list. It is easier to understand from last node.

The probability that last node is result simply 1/N [For last or N'th node, we generate a random number between 0 to N-1 and make last node as result if the generated number is 0 (or any other fixed number]

The probability that second last node is result should also be 1/N.

```
The probability that the second last node is result
          = [Probability that the second last node replaces result] X
            [Probability that the last node doesn't replace the result]
          = [1 / (N-1)] * [(N-1)/N]
          = 1/N
```

Similarly we can show probability for 3^{rd} last node and other nodes.

This article is contributed by **Rajeev**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/select-a-random-node-from-a-singly-linked-list/

Chapter 52

Why Quick Sort preferred for Arrays and Merge Sort for Linked Lists?

Why is Quick Sortpreferred for arrays?
Below are recursive and iterative implementations of Quick Sort and Merge Sort for arrays.

Recursive Quick Sort for array.
Iterative Quick Sort for arrays.
Recursive Merge Sort for arrays
Iterative Merge Sort for arrays

Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why is Merge Sortpreferred for Linked Lists?
Below are implementations of Quicksort and Mergesort for singly and doubly linked lists.

Quick Sort for Doubly Linked List

Quick Sort for Singly Linked List
Merge Sort for Singly Linked List

Merge Sort for Doubly Linked List

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

Thanks to **Sayan Mukhopadhyay** for providing initial draft for above article. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/why-quick-sort-preferred-for-arrays-and-merge-sort-for-linked-lists/

Chapter 53

Generic Linked List in C

Unlike C++ and Java, C doesn't support generics. How to create a linked list in C that can be used for any data type? In C, we can use void pointer and function pointer to implement the same functionality. The great thing about void pointer is it can be used to point to any data type. Also, size of all types of pointers is always is same, so we can always allocate a linked list node. Function pointer is needed process actual content stored at address pointed by void pointer.

Following is a sample C code to demonstrate working of generic linked list.

```
// C program for generic linked list
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
    // Any data type can be stored in this node
    void  *data;

    struct node *next;
};

/* Function to add a node at the beginning of Linked List.
   This function expects a pointer to the data to be added
   and size of the data type */
void push(struct node** head_ref, void *new_data, size_t data_size)
{
    // Allocate memory for node
    struct node* new_node = (struct node*)malloc(sizeof(struct node));

    new_node->data  = malloc(data_size);
    new_node->next = (*head_ref);

    // Copy contents of new_data to newly allocated memory.
    // Assumption: char takes 1 byte.
    int i;
    for (i=0; i<data_size; i++)
        *(char *)(new_node->data + i) = *(char *)(new_data + i);
```

```
    // Change head pointer as new node is added at the beginning
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list. fpitr is used
   to access the function to be used for printing current node data.
   Note that different data types need different specifier in printf() */
void printList(struct node *node, void (*fptr)(void *))
{
    while (node != NULL)
    {
        (*fptr)(node->data);
        node = node->next;
    }
}

// Function to print an integer
void printInt(void *n)
{
   printf(" %d", *(int *)n);
}

// Function to print a float
void printFloat(void *f)
{
   printf(" %f", *(float *)f);
}

/* Driver program to test above function */
int main()
{
    struct node *start = NULL;

    // Create and print an int linked list
    unsigned int_size = sizeof(int);
    int arr[] = {10, 20, 30, 40, 50}, i;
    for (i=4; i>=0; i--)
       push(&start, &arr[i], int_size);
    printf("Created integer linked list is \n");
    printList(start, printInt);

    // Create and print a float linked list
    unsigned float_size = sizeof(float);
    start = NULL;
    float arr2[] = {10.1, 20.2, 30.3, 40.4, 50.5};
    for (i=4; i>=0; i--)
       push(&start, &arr2[i], float_size);
    printf("\n\nCreated float linked list is \n");
    printList(start, printFloat);

    return 0;
}
```

Output:

```
 Created integer linked list is
 10 20 30 40 50

Created float linked list is
 10.100000 20.200001 30.299999 40.400002 50.500000
```

This article is contributed by **Himanshu Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

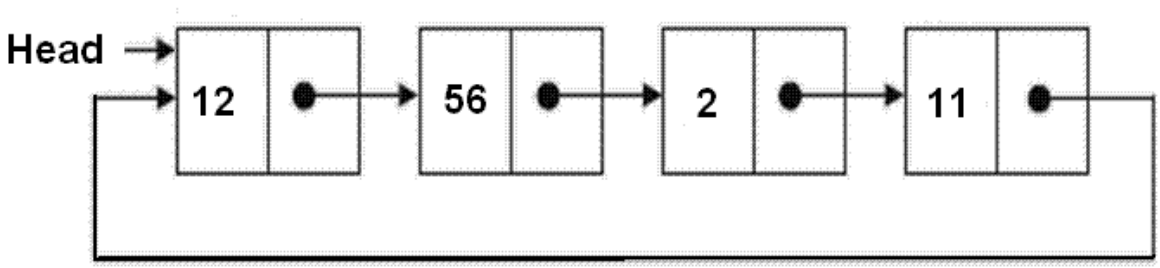
http://www.geeksforgeeks.org/generic-linked-list-in-c-2/

Category: C/C++ Puzzles Linked Lists

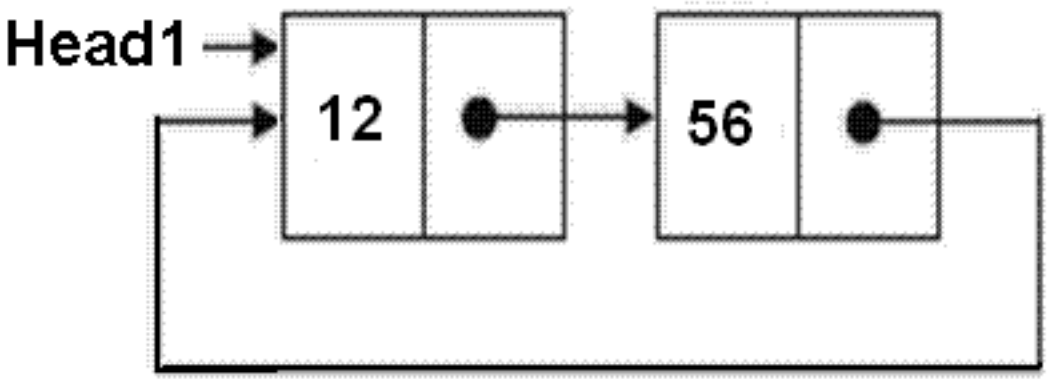
Chapter 54

Split a Circular Linked List into two halves

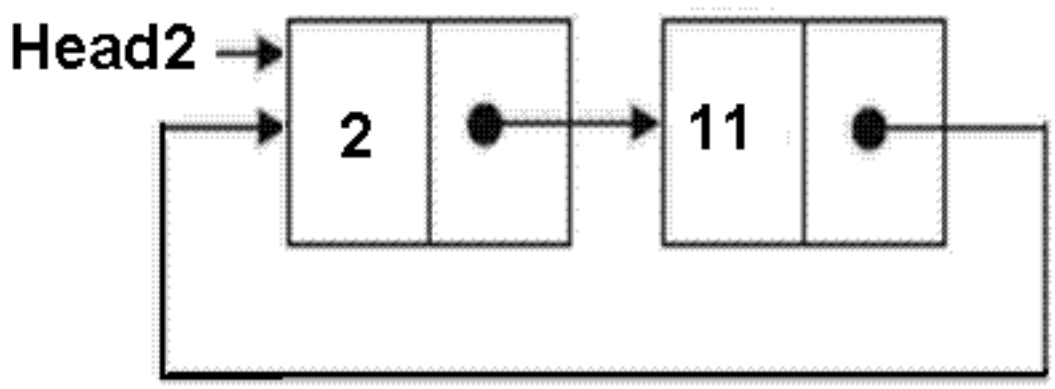
Asked by Bharani



Original Linked List



Result Linked List 1



Result Linked List 2

Thanks to Geek4u for suggesting the algorithm.
1) Store the mid and last pointers of the circular linked list using tortoise and hare algorithm.
2) Make the second half circular.
3) Make the first half circular.
4) Set head (or start) pointers of the two linked lists.

In the below implementation, if there are odd nodes in the given circular linked list then the first result list has 1 more node than the second result list.

```
/* Program to split a circular linked list into two halves */
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct node
{
  int data;
  struct node *next;
};

/* Function to split a list (starting with head) into two lists.
   head1_ref and head2_ref are references to head nodes of
    the two resultant linked lists */
void splitList(struct node *head, struct node **head1_ref,
                                          struct node **head2_ref)
{
  struct node *slow_ptr = head;
  struct node *fast_ptr = head;

  if(head == NULL)
    return;

  /* If there are odd nodes in the circular list then
     fast_ptr->next becomes head and for even nodes
     fast_ptr->next->next becomes head */
```

```
  while(fast_ptr->next != head &&
        fast_ptr->next->next != head)
  {
     fast_ptr = fast_ptr->next->next;
     slow_ptr = slow_ptr->next;
  }

 /* If there are even elements in list then move fast_ptr */
  if(fast_ptr->next->next == head)
    fast_ptr = fast_ptr->next;

  /* Set the head pointer of first half */
  *head1_ref = head;

  /* Set the head pointer of second half */
  if(head->next != head)
    *head2_ref = slow_ptr->next;

  /* Make second half circular */
  fast_ptr->next = slow_ptr->next;

  /* Make first half circular */
  slow_ptr->next = head;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the begining of a Circular
   linked lsit */
void push(struct node **head_ref, int data)
{
  struct node *ptr1 = (struct node *)malloc(sizeof(struct node));
  struct node *temp = *head_ref;
  ptr1->data = data;
  ptr1->next = *head_ref;

  /* If linked list is not NULL then set the next of
    last node */
  if(*head_ref != NULL)
  {
    while(temp->next != *head_ref)
      temp = temp->next;
    temp->next = ptr1;
  }
  else
     ptr1->next = ptr1; /*For the first node */

  *head_ref = ptr1;
}

/* Function to print nodes in a given Circular linked list */
void printList(struct node *head)
{
  struct node *temp = head;
  if(head != NULL)
```

```
  {
    printf("\n");
    do {
      printf("%d ", temp->data);
      temp = temp->next;
    } while(temp != head);
  }
}

/* Driver program to test above functions */
int main()
{
  int list_size, i;

  /* Initialize lists as empty */
  struct node *head = NULL;
  struct node *head1 = NULL;
  struct node *head2 = NULL;

  /* Created linked list will be 12->56->2->11 */
  push(&head, 12);
  push(&head, 56);
  push(&head, 2);
  push(&head, 11);

  printf("Original Circular Linked List");
  printList(head);

  /* Split the list */
  splitList(head, &head1, &head2);

  printf("\nFirst Circular Linked List");
  printList(head1);

  printf("\nSecond Circular Linked List");
  printList(head2);

  getchar();
  return 0;
}
```

Time Complexity: O(n)

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

Source

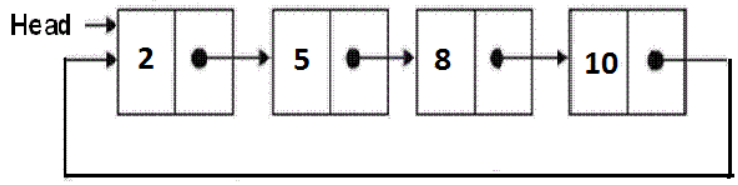
http://www.geeksforgeeks.org/split-a-circular-linked-list-into-two-halves/

Category: Linked Lists

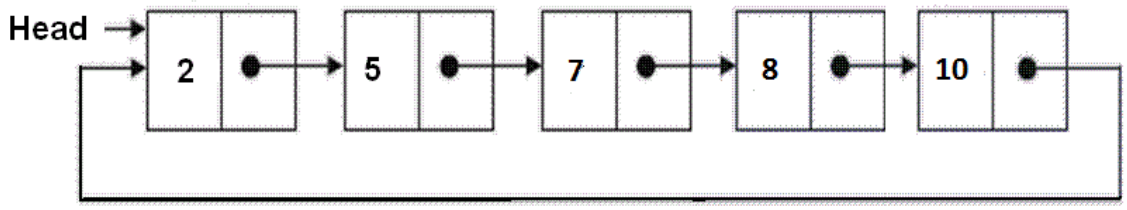
Chapter 55

Sorted insert for circular linked list

Difficulty Level: Rookie
Write a C function to insert a new value in a sorted Circular Linked List (CLL). For example, if the input CLL is following.



After insertion of **7**, the above CLL should be changed to following



Algorithm:
Allocate memory for the newly inserted node and put data in the newly allocated node. Let the pointer to the new node be new_node. After memory allocation, following are the three cases that need to be handled.

```
1) Linked List is empty:
    a)  since new_node is the only node in CLL, make a self loop.
          new_node->next = new_node;
    b) change the head pointer to point to new node.
          *head_ref = new_node;
2) New node is to be inserted just before the head node:
  (a) Find out the last node using a loop.
         while(current->next != *head_ref)
            current = current->next;
  (b) Change the next of last node.
```

```
        current->next = new_node;
  (c) Change next of new node to point to head.
        new_node->next = *head_ref;
  (d) change the head pointer to point to new node.
        *head_ref = new_node;
3) New node is to be  inserted somewhere after the head:
   (a) Locate the node after which new node is to be inserted.
        while ( current->next!= *head_ref &&
            current->next->data data)
        {   current = current->next;   }
   (b) Make next of new_node as next of the located pointer
        new_node->next = current->next;
   (c) Change the next of the located pointer
        current->next = new_node;



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

/* structure for a node */
struct node
{
  int data;
  struct node *next;
};

/* function to insert a new_node in a list in sorted way.
   Note that this function expects a pointer to head node
   as this can modify the head of the input linked list */
void sortedInsert(struct node** head_ref, struct node* new_node)
{
  struct node* current = *head_ref;

  // Case 1 of the above algo
  if (current == NULL)
  {
     new_node->next = new_node;
     *head_ref = new_node;
  }

  // Case 2 of the above algo
  else if (current->data >= new_node->data)
  {
    /* If value is smaller than head's value then
      we need to change next of last node */
    while(current->next != *head_ref)
        current = current->next;
    current->next = new_node;
    new_node->next = *head_ref;
    *head_ref = new_node;
  }

  // Case 3 of the above algo
```

```
  else
  {
    /* Locate the node before the point of insertion */
    while (current->next!= *head_ref && current->next->data < new_node->data)
      current = current->next;

    new_node->next = current->next;
    current->next = new_node;
  }
}

/* Function to print nodes in a given linked list */
void printList(struct node *start)
{
  struct node *temp;

  if(start != NULL)
  {
    temp = start;
    printf("\n");
    do {
      printf("%d ", temp->data);
      temp = temp->next;
    } while(temp != start);
  }
}

/* Driver program to test above functions */
int main()
{
  int arr[] = {12, 56, 2, 11, 1, 90};
  int list_size, i;

  /* start with empty linked list */
  struct node *start = NULL;
  struct node *temp;

  /* Create linked list from the array arr[].
    Created linked list will be 1->2->11->56->12 */
  for(i = 0; i< 6; i++)
  {
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = arr[i];
    sortedInsert(&start, temp);
  }

  printList(start);
  getchar();
  return 0;
}
```

Output:
1 2 11 12 56 90

Time Complexity: O(n) where n is the number of nodes in the given linked list.

Case 2 of the above algorithm/code can be optimized. Please see thiscomment from Pavan. To implement the suggested change we need to modify the case 2 to following.

```
// Case 2 of the above algo
else if (current->data >= new_node->data)
{
  // swap the data part of head node and new node
 swap(&(current->data), &(new_node->data));  // assuming that we have a function swap(int *, int *)

  new_node->next = (*head_ref)->next;
  (*head_ref)->next = new_node;
}
```

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/sorted-insert-for-circular-linked-list/

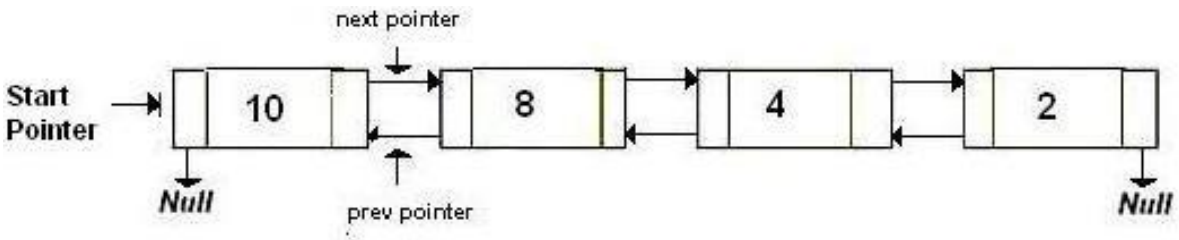
Category: Linked Lists

Chapter 56

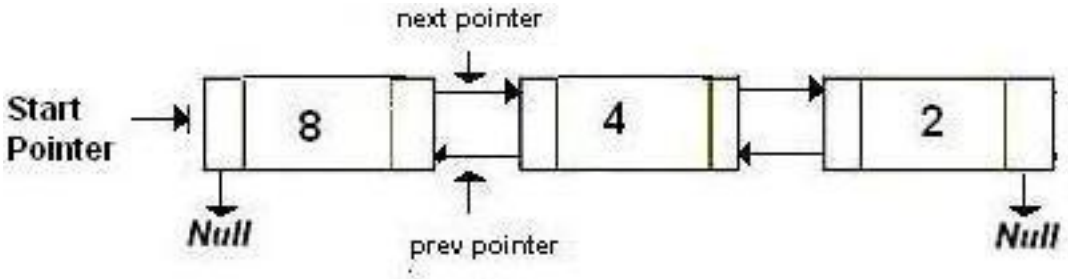
Delete a node in a Doubly Linked List

Write a function to delete a given node in a doubly linked list.

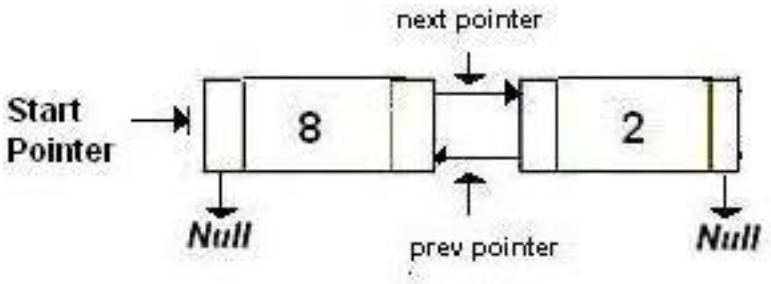
(a) Original Doubly Linked List



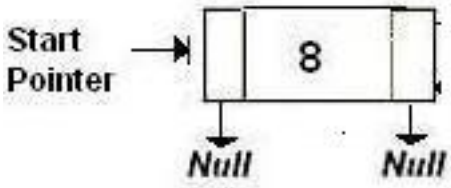
(a) After deletion of head node



(a) After deletion of middle node



(a) After deletion of last node



Algorithm
Let the node to be deleted is *del*.
1) If node to be deleted is head node, then change the head pointer to next current head.
2) Set *next* of previous to *del*, if previous to *del* exixts.
3) Set *prev* of next to *del*, if next to *del* exixts.

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

/* a node of the doubly linked list */
struct node
{
  int data;
  struct node *next;
  struct node *prev;
};

/* Function to delete a node in a Doubly Linked List.
   head_ref --> pointer to head node pointer.
   del  -->  pointer to node to be deleted. */
void deleteNode(struct node **head_ref, struct node *del)
{
  /* base case */
  if(*head_ref == NULL || del == NULL)
    return;

  /* If node to be deleted is head node */
  if(*head_ref == del)
    *head_ref = del->next;
```

```
  /* Change next only if node to be deleted is NOT the last node */
  if(del->next != NULL)
    del->next->prev = del->prev;

  /* Change prev only if node to be deleted is NOT the first node */
  if(del->prev != NULL)
    del->prev->next = del->next;

  /* Finally, free the memory occupied by del*/
  free(del);
  return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
       (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* since we are adding at the begining,
    prev is always NULL */
  new_node->prev = NULL;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* change prev of head node to new node */
  if((*head_ref) !=  NULL)
   (*head_ref)->prev = new_node ;

  /* move the head to point to the new node */
   (*head_ref)    = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
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;

  /* Let us create the doubly linked list 10<->8<->4<->2 */
  push(&head, 2);
  push(&head, 4);
  push(&head, 8);
  push(&head, 10);

  printf("\n Original Linked list ");
  printList(head);

  /* delete nodes from the doubly linked list */
  deleteNode(&head, head);  /*delete first node*/
  deleteNode(&head, head->next);  /*delete middle node*/
  deleteNode(&head, head->next);  /*delete last node*/

  /* Modified linked list will be NULL<-8->NULL */
  printf("\n Modified Linked list ");
  printList(head);

  getchar();
}
```

Time Complexity: O(1)
Time Complexity: O(1)

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/delete-a-node-in-a-doubly-linked-list/

Category: Linked Lists

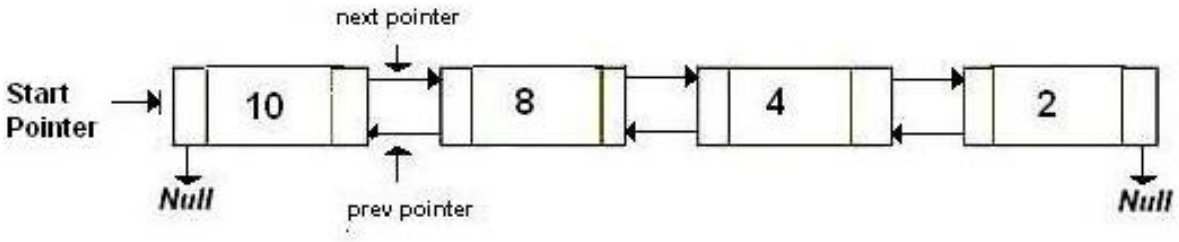
Chapter 57

Reverse a Doubly Linked List

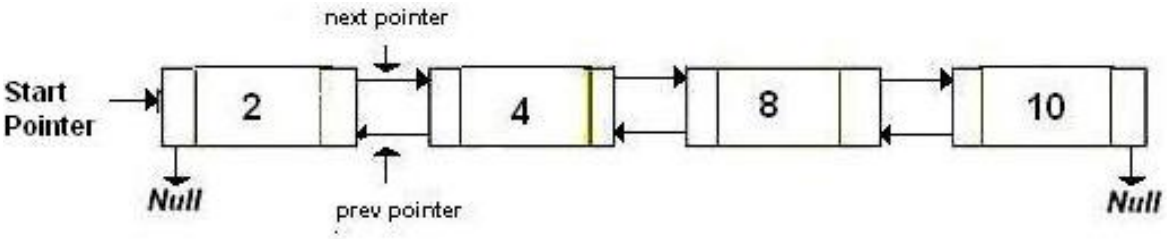
Write a C function to reverse a given Doubly Linked List

See below diagrams for example.

(a) Original Doubly Linked List



(b) Reversed Doubly Linked List



Here is a simple method for reversing a Doubly Linked List. All we need to do is swap prev and next pointers for all nodes, change prev of the head (or start) and change the head pointer in the end.

```
/* Program to reverse a doubly linked list */
#include <stdio.h>
#include <stdlib.h>
```

```
/* a node of the doubly linked list */
struct node
{
  int data;
  struct node *next;
  struct node *prev;
};

/* Function to reverse a Doubly Linked List */
void reverse(struct node **head_ref)
{
     struct node *temp = NULL;
     struct node *current = *head_ref;

     /* swap next and prev for all nodes of
       doubly linked list */
     while (current !=  NULL)
     {
       temp = current->prev;
       current->prev = current->next;
       current->next = temp;
       current = current->prev;
     }

     /* Before changing head, check for the cases like empty
        list and list with only one node */
     if(temp != NULL )
        *head_ref = temp->prev;
}



/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* since we are adding at the begining,
      prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if((*head_ref) !=  NULL)
      (*head_ref)->prev = new_node ;
```

```
    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
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;

  /* Let us create a sorted linked list to test the functions
   Created linked list will be 10->8->4->2 */
  push(&head, 2);
  push(&head, 4);
  push(&head, 8);
  push(&head, 10);

  printf("\n Original Linked list ");
  printList(head);

  /* Reverse doubly linked list */
  reverse(&head);

  printf("\n Reversed Linked list ");
  printList(head);

  getchar();
}
```

Time Complexity: O(n)

We can also swap data instead of pointers to reverse the Doubly Linked List. Method used for reversing array can be used to swap data. Swapping data can be costly compared to pointers if size of data item(s) is more.

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/reverse-a-doubly-linked-list/

Category: Linked Lists

Chapter 58

The Great Tree-List Recursion Problem.

Asked by Varun Bhatia.

Question:
Write a recursive function treeToList(Node root) that takes an ordered binary tree and rearranges the internal pointers to make a circular doubly linked list out of the tree nodes. The"previous" pointers should be stored in the "small" field and the "next" pointers should be stored in the "large" field. The list should be arranged so that the nodes are in increasing order. Return the head pointer to the new list.

This is very well explained and implemented at http://cslibrary.stanford.edu/109/TreeListRecursion.html

Source

http://www.geeksforgeeks.org/the-great-tree-list-recursion-problem/

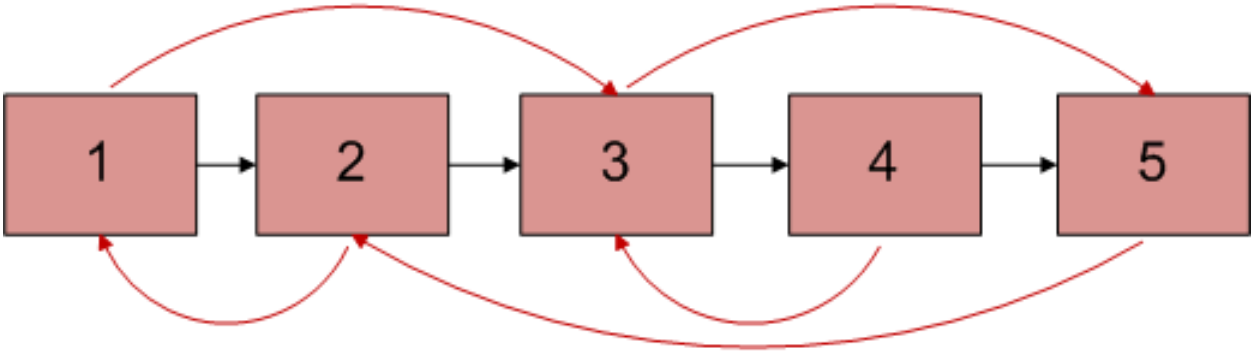
Category: Linked Lists Trees

Chapter 59

Clone a linked list with next and random pointer | Set 1

You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link list. The second pointer however CAN point to any node in the list and not just the previous node. Now write a program in **O(n) time** to duplicate this list. That is, write a program which will create a copy of this list.

Let us call the second pointer as arbit pointer as it can point to any arbitrary node in the linked list.



Arbitrary pointers are shown in red and next pointers in black

Figure 1

Method 1 (Uses O(n) extra space)
This method stores the next and arbitrary mappings (of original list) in an array first, then modifies the original Linked List (to create copy), creates a copy. And finally restores the original list.

1) Create all nodes in copy linked list using next pointers.
3) Store the node and its next pointer mappings of original linked list.
3) Change next pointer of all nodes in original linked list to point to the corresponding node in copy linked list.
Following diagram shows status of both Linked Lists after above 3 steps. The red arrow shows arbit pointers and black arrow shows next pointers.

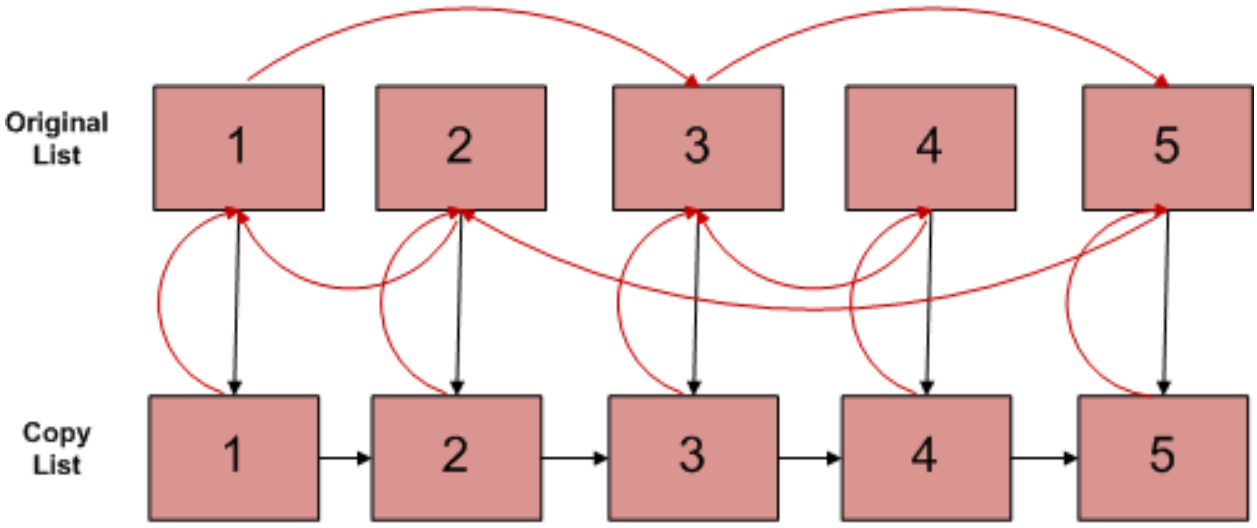


Figure 2

4) Change the arbit pointer of all nodes in copy linked list to point to corresponding node in original linked list.
5) Now construct the arbit pointer in copy linked list as below and restore the next pointer of nodes in the original linked list.

```
 copy_list_node->arbit =
              copy_list_node->arbit->arbit->next;
copy_list_node = copy_list_node->next;
```

6) Restore the next pointers in original linked list from the stored mappings(in step 2).

Time Complexity: O(n)
Auxiliary Space: O(n)

Method 2 (Uses Constant Extra Space)
Thanks to Saravanan Mani for providing this solution. This solution works using constant space.
1) Create the copy of node 1 and insert it between node 1 & node 2 in original Linked List, create the copy of 2 and insert it between 2 & 3.. Continue in this fashion, add the copy of N afte the Nth node
2) Now copy the arbitrary link in this fashion

```
    original->next->arbitrary = original->arbitrary->next;  /*TRAVERSE
TWO NODES*/
```

This works because original->next is nothing but copy of original and Original->arbitrary->next is nothing but copy of arbitrary.
3) Now restore the original and copy linked lists in this fashion in a single loop.

```
    original->next = original->next->next;
    copy->next = copy->next->next;
```

4) Make sure that last element of original->next is NULL.

Time Complexity: O(n)
Auxiliary Space: O(1)

Refer Following Post for Hashing based Implementation.

Clone a linked list with next and random pointer | Set 2

Asked by Varun Bhatia. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/a-linked-list-with-next-and-arbit-pointer/

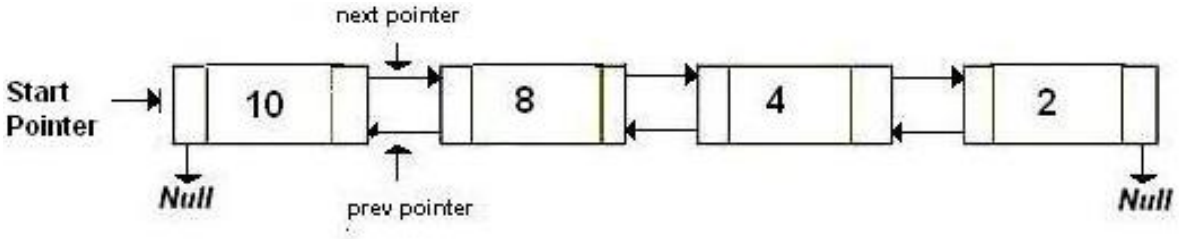
Chapter 60

QuickSort on Doubly Linked List

Following is a typical recursive implementation of QuickSort for arrays. The implementation uses last element as pivot.

```
/* A typical recursive implementation of Quicksort for array*/

/* This function takes last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than
   pivot) to left of pivot and all greater elements to right of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h- 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l  --> Starting index, h  --> Ending index */
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        int p = partition(A, l, h); /* Partitioning index */
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}
```

Can we use same algorithm for Linked List?
Following is C++ implementation for doubly linked list. The idea is simple, we first find out pointer to last node. Once we have pointer to last node, we can recursively sort the linked list using pointers to first and last nodes of linked list, similar to the above recursive function where we pass indexes of first and last array elements. The partition function for linked list is also similar to partition for arrays. Instead of returning index of the pivot element, it returns pointer to the pivot element. In the following implementation, quickSort() is just a wrapper function, the main recursive function is _quickSort() which is similar to quickSort() for array implementation.




```
// A C++ program to sort a linked list using Quicksort
#include <iostream>
#include <stdio.h>
using namespace std;

/* a node of the doubly linked list */
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* A utility function to swap two elements */
void swap ( int* a, int* b )
{   int t = *a;      *a = *b;       *b = t;   }

// A utility function to find last node of linked list
struct node *lastNode(node *root)
{
    while (root && root->next)
        root = root->next;
    return root;
}

/* Considers last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than
   pivot) to left of pivot and all greater elements to right of pivot */
node* partition(node *l, node *h)
{
    // set pivot as h element
    int x  = h->data;

    // similar to i = l-1 for array implementation
    node *i = l->prev;
```

```
    // Similar to "for (int j = l; j <= h- 1; j++)"
    for (node *j = l; j != h; j = j->next)
    {
        if (j->data <= x)
        {
            // Similar to i++ for array
            i = (i == NULL)? l : i->next;

            swap(&(i->data), &(j->data));
        }
    }
    i = (i == NULL)? l : i->next; // Similar to i++
    swap(&(i->data), &(h->data));
    return i;
}

/* A recursive implementation of quicksort for linked list */
void _quickSort(struct node* l, struct node *h)
{
    if (h != NULL && l != h && l != h->next)
    {
        struct node *p = partition(l, h);
        _quickSort(l, p->prev);
        _quickSort(p->next, h);
    }
}

// The main function to sort a linked list. It mainly calls _quickSort()
void quickSort(struct node *head)
{
    // Find last node
    struct node *h = lastNode(head);

    // Call the recursive QuickSort
    _quickSort(head, h);
}

// A utility function to print contents of arr
void printList(struct node *head)
{
    while (head)
    {
        cout << head->data << "  ";
        head = head->next;
    }
    cout << endl;
}

/* Function to insert a node at the beginging of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = new node;     /* allocate node */
    new_node->data  = new_data;
```

```
    /* since we are adding at the begining, prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) !=  NULL)  (*head_ref)->prev = new_node ;

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Driver program to test above function */
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

Output :

```
 Linked List before sorting
30  3  4  20  5
Linked List after sorting
3  4  5  20  30
```

Time Complexity: Time complexity of the above implementation is same as time complexity of QuickSort() for arrays. It takes O(n^2) time in worst case and O(nLogn) in average and best cases. The worst case occurs when the linked list is already sorted.

Can we implement random quick sort for linked list?
Quicksort can be implemented for Linked List only when we can pick a fixed point as pivot (like last element in above implementation). Random QuickSort cannot be efficiently implemented for Linked Lists by picking random pivot.

Exercise:
The above implementation is for doubly linked list. Modify it for singly linked list. Note that we don't have

prev pointer in singly linked list.
Refer QuickSort on Singly Linked Listfor solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/quicksort-for-linked-list/

Category: Linked Lists

Post navigation

← B-Tree | Set 2 (Insert) Longest prefix matching – A Trie based solution in Java →

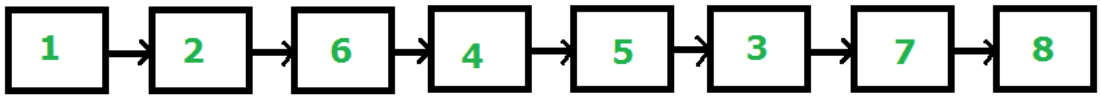
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 61

Swap Kth node from beginning with Kth node from end in a Linked List

Given a singly linked list, swap kth node from beginning with kth node from end. **Swapping of data is not allowed, only pointers should be changed.** This requirement may be logical in many situations where the linked list data part is huge (For example student details line Name, RollNo, Address, ..etc). The pointers are always fixed (4 bytes for most of the compilers).

For k = 3, the above list should be changed to following (6 and 3 are swapped)



The problem seems simple at first look, but it has many interesting cases.

Let X be the kth node from beginning and Y be the kth node from end. Following are the interesting cases that must be handled.
1) Y is next to X
2) X is next to Y
3) X and Y are same
4) X and Y don't exist (k is more than number of nodes in linked list)

We strongly recommend you to try it yourself first, then see the below solution. It will be a good exercise of pointers.

```
// A C++ program to swap Kth node from beginning with kth node from end
#include <iostream>
#include <stdlib.h>
using namespace std;

// A Linked List node
```

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

/* Utility function to insert a node at the beginning */
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;
}

/* Utility function for displaying linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

/* Utility function for calculating length of linked list */
int countNodes(struct node *s)
{
    int count = 0;
    while (s != NULL)
    {
        count++;
        s = s->next;
    }
    return count;
}

/* Function for swapping kth nodes from both ends of linked list */
void swapKth(struct node **head_ref, int k)
{
    // Count nodes in linked list
    int n = countNodes(*head_ref);

    // Check if k is valid
    if (n < k)  return;

    // If x (kth node from start) and y(kth node from end) are same
    if (2*k - 1 == n) return;

    // Find the kth node from beginning of linked list. We also find
    // previous of kth node because we need to update next pointer of
    // the previous.
    node *x = *head_ref;
```

```
    node *x_prev = NULL;
    for (int i = 1; i < k; i++)
    {
        x_prev = x;
        x = x->next;
    }

    // Similarly, find the kth node from end and its previous. kth node
    // from end is (n-k+1)th node from beginning
    node *y = *head_ref;
    node *y_prev = NULL;
    for (int i = 1; i < n-k+1; i++)
    {
        y_prev = y;
        y = y->next;
    }

    // If x_prev exists, then new next of it will be y. Consider the case
    // when y->next is x, in this case, x_prev and y are same. So the statement
    // "x_prev->next = y" creates a self loop. This self loop will be broken
    // when we change y->next.
    if (x_prev)
        x_prev->next = y;

    // Same thing applies to y_prev
    if (y_prev)
        y_prev->next = x;

    // Swap next pointers of x and y. These statements also break self
    // loop if x->next is y or y->next is x
    node *temp = x->next;
    x->next = y->next;
    y->next = temp;

    // Change head pointers when k is 1 or n
    if (k == 1)
        *head_ref = y;
    if (k == n)
        *head_ref = x;
}

// Driver program to test above functions
int main()
{
    // Let us create the following linked list for testing
    // 1->2->3->4->5->6->7->8
    struct node *head = NULL;
    for (int i = 8; i >= 1; i--)
       push(&head, i);

    cout << "Original Linked List: ";
    printList(head);

    for (int k = 1; k < 10; k++)
```

```
    {
        swapKth(&head, k);
        cout << "\nModified List for k = " << k << endl;
        printList(head);
    }

    return 0;
}
```

Output:

```
 Original Linked List: 1 2 3 4 5 6 7 8

Modified List for k = 1
8 2 3 4 5 6 7 1

Modified List for k = 2
8 7 3 4 5 6 2 1

Modified List for k = 3
8 7 6 4 5 3 2 1

Modified List for k = 4
8 7 6 5 4 3 2 1

Modified List for k = 5
8 7 6 4 5 3 2 1

Modified List for k = 6
8 7 3 4 5 6 2 1

Modified List for k = 7
8 2 3 4 5 6 7 1

Modified List for k = 8
1 2 3 4 5 6 7 8

Modified List for k = 9
1 2 3 4 5 6 7 8
```

Please note that the above code runs three separate loops to count nodes, find x and x prev, and to find y and y_prev. These three things can be done in a single loop. The code uses three loops to keep things simple and readable.

Thanks to Chandra Prakash for initial solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/swap-kth-node-from-beginning-with-kth-node-from-end-in-a-linked-list/

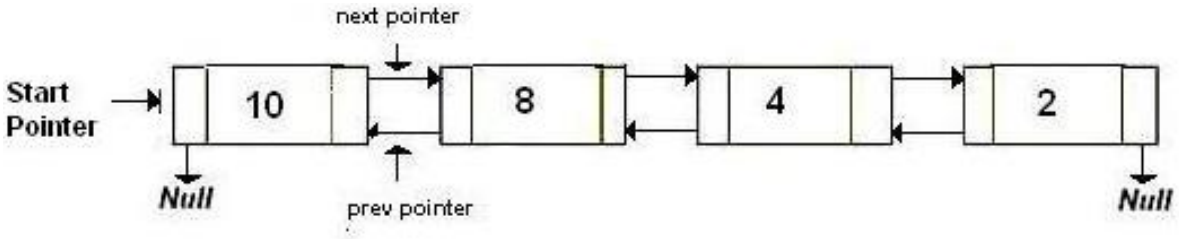
Category: Linked Lists

Chapter 62

Merge Sort for Doubly Linked List

Given a doubly linked list, write a function to sort the doubly linked list in increasing order using merge sort.

For example, the following doubly linked list should be changed to 24810



We strongly recommend to minimize your browser and try this yourself first.
Merge sort for singly linked list is already discussed. The important change here is to modify the previous pointers also when merging two lists.

Below is C implementation of merge sort for doubly linked list.

```
// C program for merge sort on doubly linked list
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next, *prev;
};

struct node *split(struct node *head);

// Function to merge two linked lists
struct node *merge(struct node *first, struct node *second)
{
    // If first linked list is empty
    if (!first)
        return second;
```

```
    // If second linked list is empty
    if (!second)
        return first;

    // Pick the smaller value
    if (first->data < second->data)
    {
        first->next = merge(first->next,second);
        first->next->prev = first;
        first->prev = NULL;
        return first;
    }
    else
    {
        second->next = merge(first,second->next);
        second->next->prev = second;
        second->prev = NULL;
        return second;
    }
}

// Function to do merge sort
struct node *mergeSort(struct node *head)
{
    if (!head || !head->next)
        return head;
    struct node *second = split(head);

    // Recur for left and right halves
    head = mergeSort(head);
    second = mergeSort(second);

    // Merge the two sorted halves
    return merge(head,second);
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct node **head, int data)
{
    struct node *temp =
        (struct node *)malloc(sizeof(struct node));
    temp->data = data;
    temp->next = temp->prev = NULL;
    if (!(*head))
        (*head) = temp;
    else
    {
        temp->next = *head;
        (*head)->prev = temp;
        (*head) = temp;
    }
}
```

```
// A utility function to print a doubly linked list in
// both forward and backward directions
void print(struct node *head)
{
    struct node *temp = head;
    printf("Forward Traversal using next poitner\n");
    while (head)
    {
        printf("%d ",head->data);
        temp = head;
        head = head->next;
    }
    printf("\nBackword Traversal using prev pointer\n");
    while (temp)
    {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
}

// Utility function to swap two integers
void swap(int *A, int *B)
{
    int temp = *A;
    *A = *B;
    *B = temp;
}

// Split a doubly linked list (DLL) into 2 DLLs of
// half sizes
struct node *split(struct node *head)
{
    struct node *fast = head,*slow = head;
    while (fast->next && fast->next->next)
    {
        fast = fast->next->next;
        slow = slow->next;
    }
    struct node *temp = slow->next;
    slow->next = NULL;
    return temp;
}

// Driver program
int main(void)
{
    struct node *head = NULL;
    insert(&head,5);
    insert(&head,20);
    insert(&head,4);
    insert(&head,3);
    insert(&head,30);
    insert(&head,10);
    printf("Linked List before sorting\n");
```

```
    print(head);
    head = mergeSort(head);
    printf("\n\nLinked List after sorting\n");
    print(head);
    return 0;
}
```

Output:

```
 Linked List before sorting
Forward Traversal using next pointer
10 30 3 4 20 5
Backward Traversal using prev pointer
5 20 4 3 30 10

Linked List after sorting
Forward Traversal using next pointer
3 4 5 10 20 30
Backward Traversal using prev pointer
30 20 10 5 4 3
```

Thanks to Goku for providing above implementation in a comment here.

Time Complexity: Time complexity of the above implementation is same as time complexity of MergeSort for arrays. It takes Θ(nLogn) time.

You may also like to see QuickSort for doubly linked list

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/

Category: Linked Lists