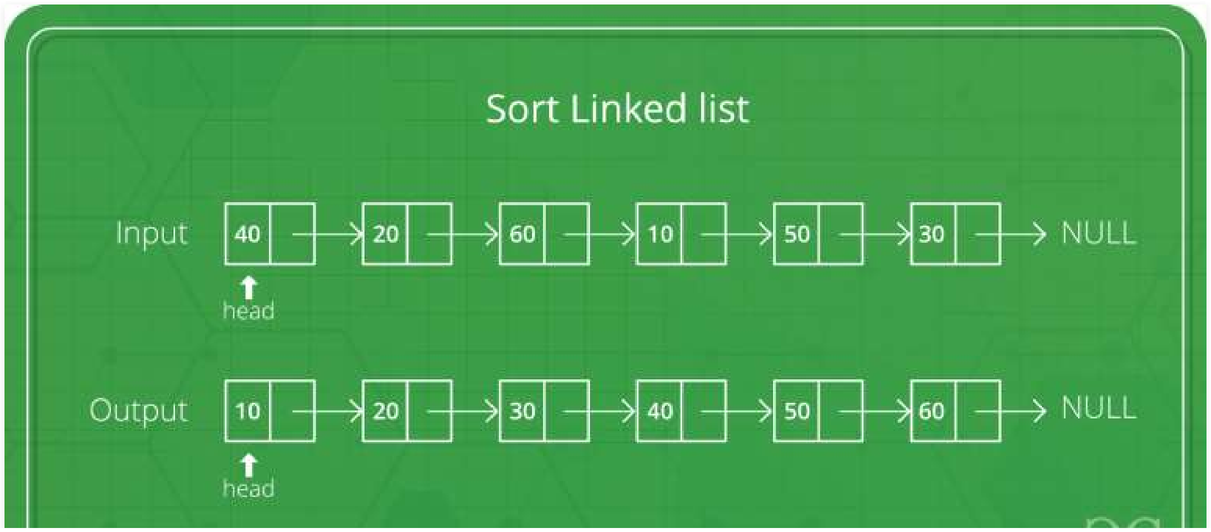
GeeksforGeeks

Related Articles Save for later

# Merge Sort for Linked Lists

Difficulty Level : Hard ● Last Updated : 26 May, 2021

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 the head by the first node of the linked list 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 the head node has to be changed if the data at the original head is not the smallest value in the linked list.

```
MergeSort(headRef)
1) If the 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);
```

Recommended: Please solve it on "***PRACTICE***" first, before moving on to the solution.

**C++**

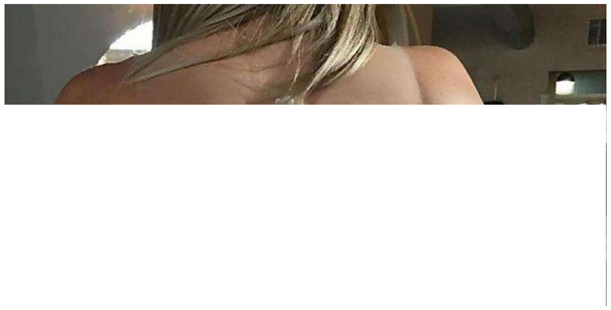
```
// C++ code for linked list merged sort
#include <bits/stdc++.h>
```

```
class Node {
public:
    int data;
    Node* next;
};

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

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

```
function */
Node* SortedMerge(Node* a, Node* b)
{
    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(Node* source,
                    Node** frontRef, Node** backRef)
{
    Node* fast;
    /* 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(Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
}

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

```
        /* 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()
    {
        /* Start with the empty list */
        Node* res = NULL;
        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);

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

        return 0;
    }
```

**Time Complexity:** O(n*log n)



**Space Complexity:** O(n*log n)

**Approach 2:** This approach is simpler and uses log n space.

mergeSort():

1. If the size of the linked list is 1 then return the head
2. Find mid using The Tortoise and The Hare Approach
3. Store the next of mid in head2 i.e. the right sub-linked list.
4. Now Make the next midpoint null.
5. Recursively call mergeSort() on both left and right sub-linked list and store the new head of the left

6. Call merge() given the arguments new heads of left and right sub-linked lists and store the final head returned after merging.
7. Return the final head of the merged linkedlist.

merge(head1, head2):

1. Take a pointer say merged to store the merged list in it and store a dummy node in it.
2. Take a pointer temp and assign merge to it.
3. If the data of head1 is less than the data of head2, then, store head1 in next of temp & move head1 to the next of head1.
4. Else store head2 in next of temp & move head2 to the next of head2.
5. Move temp to the next of temp.
6. Repeat steps 3, 4 & 5 until head1 is not equal to null and head2 is not equal to null.
7. Now add any remaining nodes of the first or the second linked list to the merged linked list.
8. Return the next of merged(that will ignore the dummy and return the head of the final merged linked list)

---

**Java**

```
// Java program for the above approach
import java.io.*;
import java.lang.*;
import java.util.*;

// Node Class
class Node {
```