COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Sort linked lists

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Sorting a Singly Linked List

Given a linked list, the task to sort this linked list in ascending or descending order can be solved by many sorting techniques.

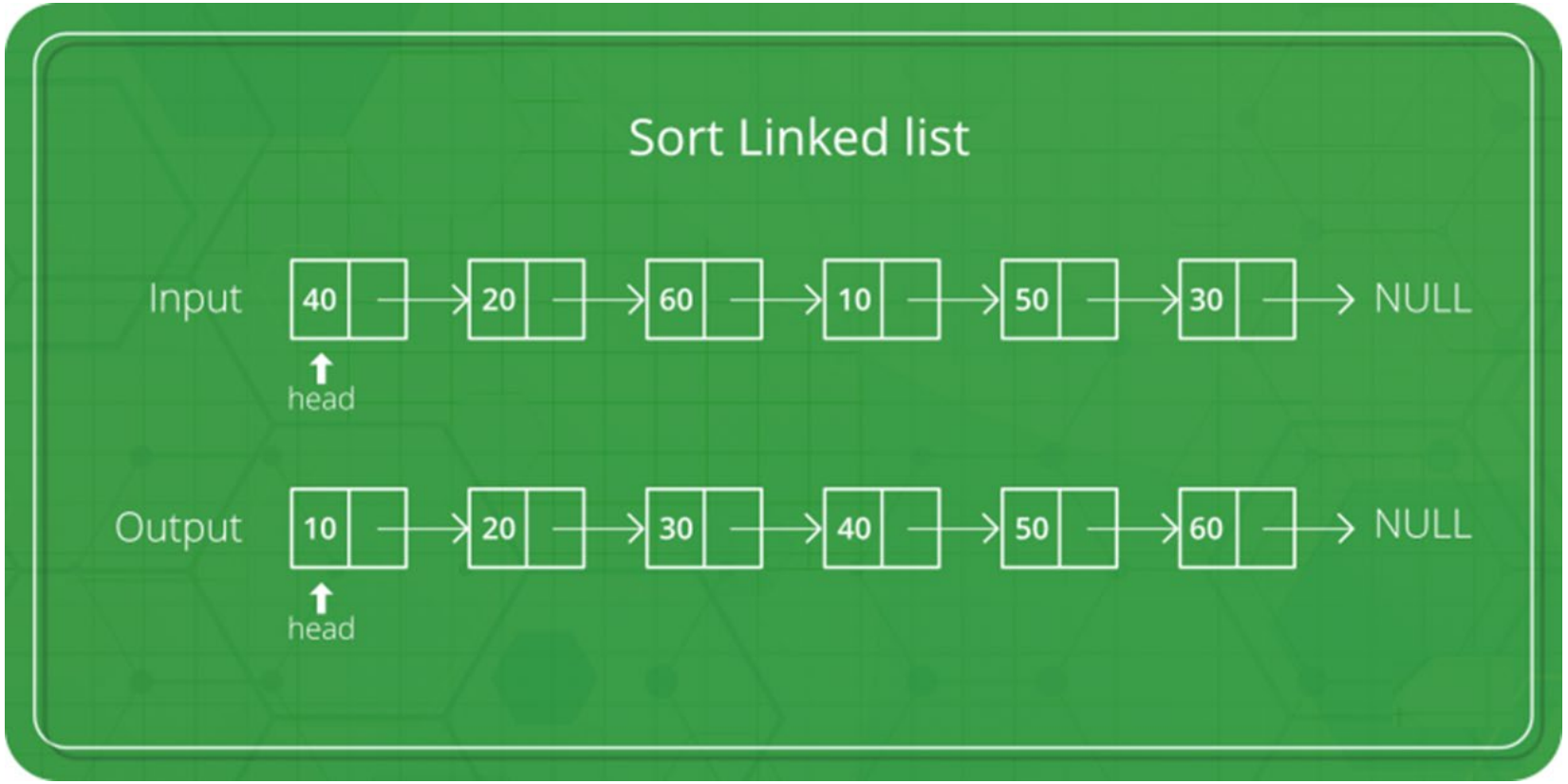
Examples:

Input: 10->30->20->5

Output: 5->10->20->30

Input: 20->4->3

Output: 3->4->20



Sorting techniques

- Bubble sort
- Insertion sort
- Quick sort
- Merge sort

Bubble sort algorithm

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

In Bubble Sort algorithm,

traverse from left and compare adjacent elements and the higher one is placed at right side.
In this way, the largest element is moved to the rightmost end at first.
This process is then continued to find the second largest and place it and so on until the data is sorted.

Bubble sort algorithm


```
// A version of Bubble Sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                SWAP(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false) // If no two elements were swapped by inner loop, then break
            break;
    }
}
```

```
#define SWAP(a, b) \
        int temp = a; \
        a = b; \
        b = temp;
```

```
#define SWAP(a, b) \
        do { \
                int temp = a; \
                a = b; \
                b = temp; \
        } while (0)
```

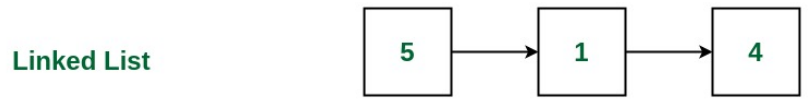
Recursive bubble sort algorithm

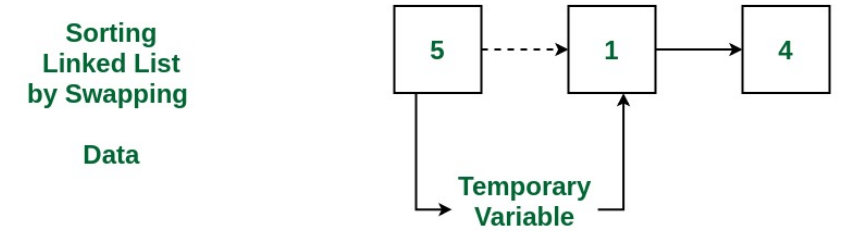

```
void bubbleSort(int arr[], int n)
{
   // Base case
   if (n == 1)
      return;
   int count = 0;
   // One pass of bubble sort. After this pass, the largest element is moved (or bubbled) to end.
   for (int i=0; i<n-1; i++)
      if (arr[i] > arr[i+1]){
         SWAP(arr[i], arr[i+1]);
         count++;
      }
   // Check if any recursion happens or not, if any recursion does not happen then return
   if (count==0)
     return;
   // Largest element is fixed,
   // recur for remaining array
   bubbleSort(arr, n-1);
}
```

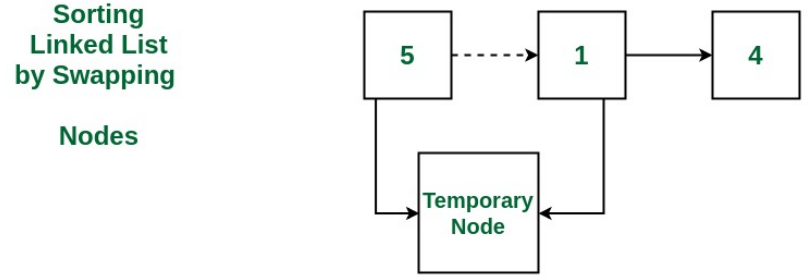
Bubble sort algorithm

Method 1: Sort Linked List using Bubble Sort
Get the linked list to be sorted and apply Bubble Sort to this linked list, in which, while comparing the two adjacent nodes, actual nodes are swapped instead of just swapping the data.
You can print the sorted list to check that is an ordered list.

Time complexity: O(n ^ 2)
Auxiliary Space: O(1)







Linked list bubble sort algorithm


```
int bubbleSort(struct Node** ptr) // A version of Bubble Sort
{
  struct Node* prev;
  struct Node* current;
  struct Node* tmp;
  bool swapped = true;

  if (*ptr == NULL || (*ptr)->next == NULL)
     return(0);
  while (swapped)
  {
     swapped = false;
     current = (*ptr)->next;
     if ((*ptr)->data > current->data)
     {
        swapped = true;
        prev = *ptr;
        *ptr = current;
        prev->next = current->next;
        current->next = prev;
     }
     prev = *ptr;
     current = prev->next->next;
     while (current != NULL)
     {
        if (prev->next->data > current->data)
        {
           swapped = true;
           tmp = prev->next;
           prev->next = current;
           tmp->next = current->next;
           current->next = tmp;
        }
        prev = prev->next;
        current = current->next;
     }
     if (!swapped)
        break;
  }
  return(0);
}
```

Insertion sort algorithm

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.
Consider an example:
arr[]: {18, 15, 23, 6, 9}

18	15	23	6	9

Insertion sort algorithm

First pass

Initially, the first two elements of the array are compared in insertion sort.

18	**15**	23	6	9

Here, 18 is greater than 15 hence they are not in the ascending order and 18 is not at its correct position. Thus, swap 18 and 15.

So, for now 15 is stored in a sorted sub-array.

15	**18**	23	6	9

Second Pass

Now, move to the next two elements and compare them

15	**18**	**23**	6	9

Insertion sort algorithm

Here, 23 is greater than 18, thus both elements seems to be in ascending order, hence, no swapping will occur. 18 also stored in a sorted sub-array along with 15

Third Pass

Now, two elements are present in the sorted sub-array which are 15 and 18

Moving forward to the next two elements which are 23 and 6

15	18	**23**	**6**	9

Both 6 and 23 are not present at their correct place so swap them

15	18	**6**	**23**	9

After swapping, elements 18 and 6 are not sorted, thus swap again

15	**6**	**18**	23	9

Insertion sort algorithm

Here, again 15 and 6 are not sorted, hence swap again

6	**15**	18	23	9

Here, 6 is at its correct position

Fourth Pass

Now, the elements which are present in the sorted sub-array are 6, 15 and 18

Moving to the next two elements 23 and 9

6	15	18	**23**	**9**

They are not sorted, thus perform swap between both

6	15	18	**9**	**23**

Now, 9 is smaller than 18, hence, swap again

6	15	**9**	**18**	23

Insertion sort algorithm

Here, also swapping makes 15 and 9 unsorted hence, swap again

6	**9**	**15**	18	23

Finally, the array is completely sorted.

Insertion sort algorithm


```
// Function to sort an array using insertion sort
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1], that are greater than key,
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Insertion sort algorithm

Method 2: Sort Linked List using Insertion Sort
Below is a simple insertion sort algorithm for a linked list.

- Create an empty sorted (or result) list
- Traverse the given list, do following for every node.
  - Insert current node in sorted way in sorted or result list.
- Change head of given linked list to head of sorted (or result) list.

Time complexity: O(n ^ 2)
Auxiliary Space: O(1)

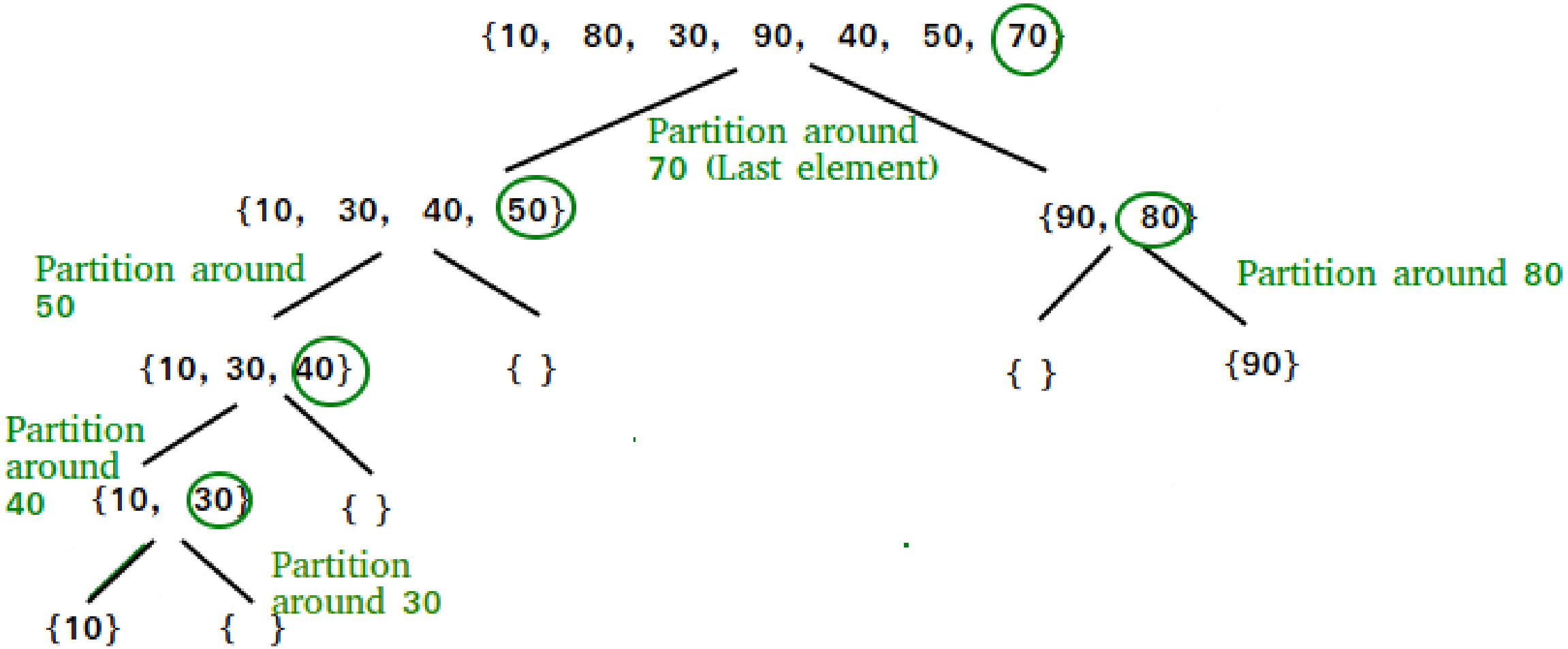
QuickSort algorithm

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort work?
The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

QuickSort algorithm



QuickSort algorithm

Choice of Pivot:

There are many different choices for picking pivots.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick the middle as the pivot.

QuickSort algorithm


```
int partition(int arr[],int low,int high)
{
 //choose the pivot
 int pivot=arr[high];
 //index of smaller element and indicate the right position of pivot found so far
 int i=(low-1);
 for(int j=low;j<=high;j++)
 {
  //if current element is smaller than the pivot
  if (arr[j]<pivot)
  {
   //Increment index of smaller element
   i++;
   SWAP(arr[i],arr[j]);
  }
 }
 SWAP(arr[i+1],arr[high]);
 return (i+1);
}
```

10, 80, 30, 90, 40, 50, 70
pivot = 70
j=0, i=-1, 10 is < 70, SWAP(10,10)
j=1, i=0, 80 is not < 70
j=2, i=0, 30 is < 70, SWAP(80,30)
10, 30, 80, 90, 40, 50, 70
j=3, i=1, 90 is not < 70
j=4, i=1, 40 is < 70, SWAP(80, 40)
10, 30, 40, 90, 80, 50, 70
j=5, i=2, 50 is < 70, SWAP(90, 50)
10, 30, 40, 50, 80, 90, 70
j=6, i=3, 70 is not < 70
SWAP(80, 70)
10, 30, 40, 50, 70, 90, 80
return(4)

QuickSort algorithm


```
void quickSort(int arr[],int low,int high)
{
  // when low is less than high
  if (low<high)
  {
    // pi is the partition return index of pivot

    int pi=partition(arr,low,high);

    //recursion Call
    //smaller element than pivot goes left and higher element goes right
    quickSort(arr,low,pi-1);
    quickSort(arr,pi+1,high);
  }
}
```

pi = 4
10, 30, 40, 50, 70, 90, 80
quickSort(arr, 0, 3)
quickSort(arr, 5, 6)

QuickSort algorithm

Method 3: Sort Linked List using Quick Sort
Call the partition function to get a pivot node placed at its correct position
In the partition function, the last element is considered a pivot
Then traverse the current list and if a node has a value greater than the pivot, then move it after the tail. If the node has a smaller value, then keep it at its current position.
Return pivot node
Find the tail node of the list which is on the left side of the pivot and recur for the left list
Similarly, after the left side, recur for the list on the right side of the pivot
Now return the head of the linked list after joining the left and the right list, as the whole linked list is now sorted. Time complexity: O(n ^ 2)
Auxiliary Space: O(1)

QuickSort algorithm

- This function partitions the list into two parts around a pivot.
- Nodes smaller than the pivot are moved to the front, and larger nodes are moved to the back.

```
// Function to partition the list and return the pivot node
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;

    // Nodes smaller than the pivot will be moved to the front
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // Move node to the new head
            if ((*newHead) == NULL) {
                (*newHead) = cur;
            }
            prev = cur;
            cur = cur->next;
        }
```

QuickSort algorithm


```
else {
        // Move node to the end of the list
        if (prev) {
            prev->next = cur->next;
        }
        struct Node* temp = cur->next;
        cur->next = NULL;
        tail->next = cur;
        tail = cur;
        cur = temp;
      }
   }
 // If the new head hasn't been updated, it is the pivot
   if ((*newHead) == NULL) {
      (*newHead) = pivot;
   }
   // Update the new end to the current tail
   (*newEnd) = tail;
   return pivot;
}
```

MergeSort algorithm

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

How does Merge Sort work?

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

MergeSort algorithm

Method 4: Sort Linked List using Merge Sort

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

MergeSort algorithm

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

MergeSort algorithm

Split the List

- Splits the linked list into two halves.
- Uses the slow and fast pointer approach to find the middle of the list.

```
void splitList(struct Node* source, struct Node** frontRef, struct Node** backRef) {
    struct Node* slow;
    struct Node* fast;
    slow = source;
    fast = source->next;
    // Move fast two nodes and slow one node
    while (fast != NULL) {
        fast = fast->next;
        if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
        }
    }
    // Split the list into two halves
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
}
```

MergeSort algorithm


```
void mergeSort(struct Node** headRef) {
    struct Node* head = headRef;
    struct Node *a;
    struct Node* b;
    // Base case: if the list is empty or has a single element
    if ((head == NULL) || (head->next == NULL)) {
       return;
    }

    // Split the list into two halves
    splitList(head, &a, &b);

    // Recursively sort the two halves
    mergeSort(&a);
    mergeSort(&b);

    // Merge the sorted halves
    *headRef = sortedMerge(a, b);
}
```

MergeSort algorithm


```
struct Node* sortedMerge(struct Node* a, struct Node* b) {
    struct Node* result = NULL;

    if (a == NULL)
        return b;
    else if (b == NULL)
        return a;

    if (a->data <= b->data) {
        result = a;
        result->next = sortedMerge(a->next, b);
    } else {
        result = b;
        result->next = sortedMerge(a, b->next);
    }

    return result;
}
```