# Merge Sort Docs

The merge sort algorithm recursively merges and sorts a list. First, it checks whether the list has one or more elements. If the list has fewer than two elements, the algorithm returns the list because it is already sorted. Otherwise, it splits the list into halves. Then, the algorithm uses recursion to sort each half. Finally, it combines both halves, producing a single, sorted list. I created a merge sort algorithm for linked lists.

# Programmer's Guide

## void moveNodeIntoSortedList(LinkedList<Data> &list, LinkedList<Data> &sortedItems, Node<Data> \*node)

Moves a node from one linked list to another by inserting the node data, then deleting the old node.

Params

* LinkedList<Data> &list: the list to move the node from
* LinkedList<Data> &sortedItems: the sorted items
* Node<Data> \*node: the node to move

## LinkedList<Data> &getListWithRemainingElements(LinkedList<Data> &leftList, LinkedList<Data> &rightList)

Returns the list with remaining elements

Returns: LinkedList<Data>

Params:

* LinkedList<Data> &leftList: the left list
* LinkedList<Data> &rightList: the right list

## void moveRemainingElements(LinkedList<Data> &sortedItems, LinkedList<Data> &leftList, LinkedList<Data> &rightList)

Gets the list that still has nodes, then moves its nodes into the sorted list.

Params:

* LinkedList<Data> &sortedItems: the sorted items so far
* LinkedList<Data> &leftList: left list
* LinkedList<Data> &rightList: right list

## Data getDefaultValue(LinkedList<Data> &leftList)

Gets the default value used for the head and tail's value.

Returns: Data

Params:

* LinkedList<Data> &list: the list to get the value from

## LinkedList<Data> mergeLists(LinkedList<Data> &leftList, LinkedList<Data> &rightList)

Merges and sorts two lists together.

Returns: LinkedList<Data>

Params:

* LinkedList<Data> &leftList: the left list
* LinkedList<Data> &rightList: the right list

## bool getIfShouldContinueLookingForMiddleNode(LinkedList<Data> &list, Node<Data> \*leftNode, Node<Data> \*rightNode)

Returns a boolean indicating whether the left and right cursors point to the middle node.

Returns: bool

Params:

* LinkedList<Data> &list: the list to get the middle node from
* Node<Data> \*leftNode: the left cursor
* Node<Data> \*rightNode: the right cursor

## Node<Data> \*getMiddleNode(LinkedList<Data> &list)

Returns the middle node.

Returns: Node<Data>

Params:

* LinkedList<Data> &list: the list to get the middle node from

## bool getIfListIsSorted(LinkedList<Data> &list)

Returns whether the list's length is either 0 or 1, meaning the list is already sorted.

Returns: bool

Params:

* LinkedList<Data> &list: the list to check

## LinkedList<Data> getLeftList(Data defaultNodeValue, LinkedList<Data> &list, Node<Data> \*middleNode)

Returns the left half of the list

Returns: LinkedList<Data>

Params

* Data defaultNodeValue: the default value of the head and tail
* LinkedList<Data> &list: the list to split
* Node<Data> \*middleNode: the middle node of the list

## LinkedList<Data> getRightList(Data defaultNodeValue, LinkedList<Data> &list, Node<Data> \*middleNode)

Returns the right half of the list

Returns: LinkedList<Data>

Params

* Data defaultNodeValue: the default value of the head and tail
* LinkedList<Data> &list: the list to split
* Node<Data> \*middleNode: the middle node of the list

## LinkedList<Data> mergeSort(LinkedList<Data> &list)

Sorts a linked list by merging and sorting recursively.

Returns: LinkedList<Data>

Params:

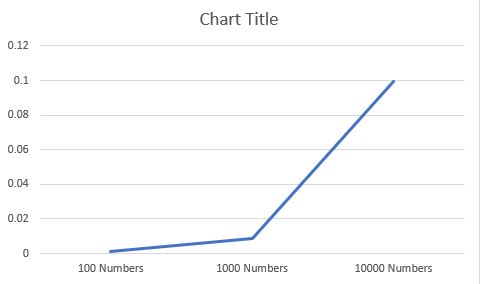
* LinkedList<Data> &list: the list to sort

Example:

LinkedList<int> linkedList{0};  
mergeSort(linkedList);

# Analysis





My merge sort function has a time complexity of O(n log n). When the algorithm calls getMiddleNode, it iterates over the list until the cursors reach the middle. Therefore, getMiddleNode has a time complexity of O(n). Then, the algorithm uses recursion to sort the left and right lists. Since mergeSort halves the lists until they can no longer be divided, mergeSort will call itself up to log n times. Finally, mergeLists merges the left and right lists by iterating over each of their elements. Since both lists combined have n nodes, mergeLists has a time complexity of O(n). When you combine these time complexities, mergeSort has a time complexity of O(n log n).

The measured time complexity does not match the actual time complexity. Sorting 1000 numbers should have taken more than 0.01 seconds if the time complexity were greater than linear. However, the algorithm only took about 0.008 seconds on average. The measured results do not match the expected results probably because the dataset is too small.

According to *Time Complexities of all Sorting Algorithms*, selection sort performs best for small datasets (about 100 numbers) because it has a quadratic time complexity (Geeks for Geeks, 2021). Insertion sort and bubble sort perform best for medium datasets (about 1000 numbers) because they have a best-case time complexity of O(n) (Geeks for Geeks, 2021). Finally, merge sort performs best for all datasets because its best, worst, and average time complexities are O(n log n) (Geeks for Geeks, 2021).

# References

Geeks for Geeks. (2021, June 28). *Time complexities of all sorting algorithms*. GeeksforGeeks. https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/.