Lesson 22: Sorting Algorithms, II

Professor Abdul-Quader

Computer Science II

Exercise: implement the merge algorithm

Part 1: Implement the merge algorithm. Recall: given two **sorted** lists, we want to create a new list that contains all the elements of the two lists in sorted order.

```
public int[] merge(int[] list1, int[] list2);
```

Make sure your method works: write a test case in the `main` method and print out the resulting merged list. (Use the pseudocode we worked on in the last video.)

Mergesort pseudocode

Recall: the merge sort algorithm works as follows:

- Split the list in half.
- Recursively merge sort each list.
- Merge the two halves.

Write pseudocode for the merge sort algorithm and write pseudocode for it. (No need to re-write the `merge` method.)

Mergesort running time

To analyze the running time, suppose $T(n)$ is the number of steps the merge sort algorithm takes on a list of size n. Then notice:

$$T(n) = 2T(n/2) + O(n)$$

For simplicity, we will just assume $T(n) = 2T(n/2) + n$. Notice: if $n = 2^3$:

$$\begin{aligned} T(2^3) &= 2T(2^2) + 2^3 \\ &= 2(2T(2^1) + 2^2) + 2^3 \\ &= 2^2 T(2^1) + 2 * 2^3 \\ &= 2^2(2T(1) + 2^1) + 2 * 2^3 \\ &= 2^3 T(1) + 3 * 2^3 \end{aligned}$$

Now notice that $T(1)$ (the base case) is $O(1)$, and if $n = 2^3$, then $3 = \log_2 n$.

Exercise: implement the merge sort algorithm

Part 2: implement the recursive merge sort algorithm. Use the existing code that you wrote for `merge`. Test it out with a main method.

```
public int[] mergeSort(int[] list);
```

Project 4

Final project of the semester: Sorted List implementation.

You are meant to implement a list structure with the methods:

- insert(): adds an element to the list.
- get(): gets the i^{th} smallest element of the list.
- size(): returns the size of the list.

You can decide how to implement this. For example, you can keep the elements of the list in sorted order every time you insert (which would make `get` easier), or you can make `insert` easier by just adding to the end of the list, but make `get` harder.

Trade-offs

Suppose we keep the elements sorted every time we insert. What would the best running times be for the insert and get methods?

Trade-offs

Suppose we keep the elements sorted every time we insert. What would the best running times be for the insert and get methods?

Now suppose we just add to the end of the list. Its running time is (amortized) O(1). How would we implement `get`?

Trade-offs

Suppose we keep the elements sorted every time we insert. What would the best running times be for the insert and get methods?

Now suppose we just add to the end of the list. Its running time is (amortized) O(1). How would we implement `get`?

There is a data structure one can use to get (average-case) $O(\log n)$ insertion and $O(\log n)$ get (not covered in this class, but will be discussed in MAT 3710: Data Structures).

Data structures: Linked List

A **data structure** is a way of organizing data in memory. We have seen two important kinds of data structures already: arrays and ArrayList.

A **linked list** is another kind of list structure.

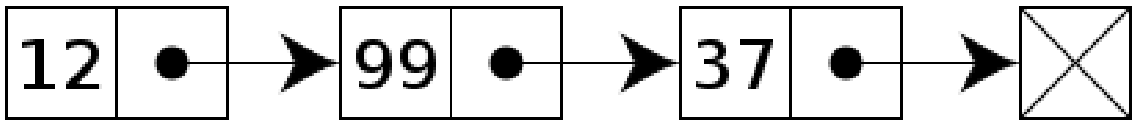

Figure: Singly linked list diagram, *Wikimedia Commons*

The data is organized into **nodes**. Each node has a data item and a link to the next node. (The last node links to `null`.)