Lecture Summary

In this lecture, we followed up in more depth about linked lists. We covered a recursive definition, as well as an asymptotic analysis of the running time of different operations. We also learned about the concept of Abstract Data Types, and their C++ implementation using classes and objects.

1 Definition of Linked Lists

While we know intuitively what a linked list is based on our discussion so far, we haven’t defined it formally. In other words, we haven’t yet said exactly how to distinguish a “Linked List” from a “Not Linked List”. In trying to come up with a definition, our intuition tells us that it is a bunch of items, where each item has associated data and points to another item. But first of all, one of the items doesn’t point to anything. And second, this would include a lot of nodes, all of which point to one other node. That’s not what we want. We may try something like “a bunch of item each of which points to 1 other node, and is pointed to by 1 other node, except for one node that points to no other node and 1 node which is not pointer to by any other.” This would be correct, but getting a little cumbersome.

To keep the definition short yet clear, we can draw on recursive definitions which we saw a few lectures ago, and define Linked Lists as follows:

- The empty list is a linked list.
- A node that points to a “shorter” linked list is a linked list.

We need the adjective “shorter” to exclude, for example, a one-node list where the node points to itself; in that case, it would be pointing to a linked list of the same length. (Remember that for recursive functions and definitions, we should only reference “smaller” objects, which we do so explicitly here.)

2 Runtime Analysis: An Overview

We want to analyze how long the different operations take for linked lists, in particular adding, removing, and looking up elements. This will be a recurring theme throughout the semester: when we design and implement data structures, we need to understand how efficiently they support different operations; this will let us decide which data structure is right for the particular task we have to perform.

When analyzing the running time of certain operations, there are a lot of factors at play: How fast is the machine? What other processes are running? For that reason, just measurements alone don’t tell us the whole story. While measuring running time empirically for a bunch of inputs is important, it must be complemented by a theoretical analysis.

Which running time should we measure/analyze? There are probably three natural candidates: worst case, best case, and average case. Best case clearly does not make sense; imagine promising a prospective customer that if only he feeds your program the one input it was optimized for, it will run fast enough. Not a good selling point. Forget that the notion of “best case” exists! Average case is more meaningful. But it also has downsides. Imagine that you are writing the software for a self-driving car, which needs to react to a car in front of yours breaking. Suppose that 99% of the time, the software reacts in 0.01s, and 1% of the
time, it takes 30s. On average, it takes much less than half a second. But I’m not sure you’d like your car to crash 1% of the time when someone breaks in front of you. There are cases when average case analysis is the right thing, and you might see it in CS270 or more advanced algorithms classes.

But for now, the only thing we care about is “worst case”. If you do a worst-case analysis, you can confidently promise your customers (or collaborators) that your code never takes more than a certain amount of time. That’s the kind of useful guarantee you want.

Next, we need to think about how exactly to report those worst-case running times. Which machine? Clearly, that can make a difference, as some computers are (much) faster than others. But in the end, it’s not a property of our algorithm or data structure, so maybe we shouldn’t measure performance in seconds. How about we count operations that are performed? That sounds better, but it raises the question what an “operation” is. Different processors differ in whether they have a few simple operations (this is called RISC) from which you build up complex ones, or implement many complex operations. For instance, is a[i] ++ one operation? Technically, we first have to read the value of i, then use it to read the value of a[i], then increment it, and then write it back to its memory position.

This kind of fine-grained analysis is quite cumbersome, and also does not tell us much about whether a data structure is actually good. In order not to waste time on unimportant details, almost all theoretical analyses of running times of data structures and algorithms are carried out using big-O notation. Notice that big-O notation is not tied to running time of algorithms — you can apply it to many other functions as well. It’s just that this is the area where computer scientists need it most frequently. Here is a refresher on what you should have learned by now in CSCI170 about big-O notation:

- When we say that a function (call it $T(n)$) is $O(f(n))$, we mean that $T(n) \leq c \cdot f(n)$ for all $n$. So $T$ is never bigger than $f$ times a constant.
- When we say that $T$ is $\Omega(g(n))$, we mean that $T(n) \geq c' \cdot g(n)$ for all $n$. So $T$ is never less than $g$ times a constant $c' > 0$.
- Finally we say that $T$ is $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$. That means that up to constant factors, $T$ and $f$ are the same function.

Big-O allows us to be sloppy in a precise sense. We know exactly what we are ignoring (constant factors, and thus also lower-order terms) and what we are keeping: the growth of $T$ as a function of $n$. The latter is the important part: we want to know how well our data structure performs as the size (e.g., number of items stored in it) grows large. That’s exactly what big-O notation is there for. Whenever you analyze any kind of algorithm or data structure, we recommend that you don’t say that something takes “$n$ steps”, and instead say “$T(n)$ steps” (or “$O(n)$ steps”). This makes clear that you understand that “steps” is an amorphous concept, and are focusing on the big picture.

Understanding big-O analysis for worst-case running time tends to be pretty straightforward. You go through your code and count the number of operations executed in the worst case, as a function of $n$, and along the way drop constant factors and terms that grow less fast than the dominant term. That way, you arrive at an upper bound in big-O notation for the worst-case running time.

Students generally seem to be more mystified by using big-O notation with worst-case running time. After all, big-O gives a lower bound, so the natural (but wrong!) assumption is that this must involve proving that the running time is always at least some function $g(n)$. The mistake here is in the word “always”. If we write “always”, we are implicitly talking about the best-case running time. Or rather, the confusion probably arises because we substituted “running time” for “worst-case running time”, i.e., we are trying to prove upper and lower bounds on the “running time”. When you try to prove a lower bound on the worst-case running time, you need to show that the worst case takes at least some number $g(n)$ of steps. To do that, it is enough to show that there is one case of size $n$ (for each $n$) on which your algorithm takes $\Omega(g(n))$ steps. We recommend that you re-read the previous paragraph a few times, to make sure it sinks in correctly.

Why would we actually care about big-O analysis? Suppose that we had two data structures to choose from for a particular task we need to perform, and for both, we have shown that the worst-case running time
is $O(n^2)$, so it never takes more than a constant times $n^2$ steps to execute the task. So far, the two data structures look the same. Now suppose that for the first one, we also know that it is $\Omega(n^2)$, which means that we have (at least) one example on which the task does take (a constant times) $n^2$ steps. For the second alternative, we only have a lower bound of $\Omega(n)$. So for the first data structure, we know pretty much the entire story. For the second one, there are multiple alternatives:

- Our big-O analysis was not careful enough, and the task is actually performed in $O(n)$ steps. In that case, it would be much better than the alternative.
- Our big-$\Omega$ analysis was not careful enough, and there are inputs on which it does take $\Omega(n^2)$ steps. Then, it seems that the two data structures are comparable, and we should look for other clues, like how easy to implement they are, or how big the hidden constants are. (This is where actual measurements will be helpful.)
- Something in between: maybe both our upper and lower bound are wrong, and the correct answer was $\Theta(n^{1.5})$.

For each function (and thus also for the worst-case running time of each algorithm), there is a correct answer: some $f$ such that the running time is $\Theta(f(n))$. Sometimes, determining this $f$ can be quite complex, which is why sometimes, we have upper and lower bounds that do not match. In this class, typically, it will not be too hard. As an upshot, when you have a $\Theta(f(n))$ type of bound on your worst-case running time, you know that you have a near-complete understanding of how efficient your algorithm is.

To summarize once more: when you say that an algorithm’s worst-case running time is $O(f(n))$, you say that it can never take more than $c \cdot f(n)$ steps. When you say that it is $\Omega(g(n))$, you say that there are some cases when it is as bad as $c' \cdot g(n)$ steps.

In class, someone asked whether it is possible that the big-$\Omega$ bound is larger than the big-O bound, for instance, could an algorithm have a running time of $O(n^2)$ and $\Omega(n^3)$? The answer is “No”, as this would mean that it takes at most $n^2$ and at least $n^3$ steps in some cases.

### 3 Analysis of Linked List operations

Let’s now analyze how long the three main operations `add`, `lookup/find`, and `remove` take. We will express the running time of these functions in terms of the length of the list, which we denote by $n$.

**Add:** Let’s say that we add elements to the end of the list, like we have so far in class. If we have a `tail` pointer, we just execute a constant number of steps to generate a new object and set the links between it and the `tail` pointer. So it takes a constant number of steps, written as $O(1)$. Since it also takes at least one step (in fact, always, but in particular in the worst case), it is also $\Omega(1)$, so in total, the time is $\Theta(1)$.

If we do not have a `tail` pointer, we first have to find the end of the list, starting from `head`. We can certainly accomplish that by performing a constant amount of work (pointer update) for each element of the list, which gives us $O(n)$. On the other hand, we also do have to go through the entire list (in fact, every time, so this is also a best-case time), so the time is $\Omega(1)$, so in total, the time is $\Theta(1)$. After we have found the tail of the list, we still have to spend the $\Theta(1)$ time to generate and link the new element, so in total, we get $\Theta(n) + \Theta(1) = \Theta(n)$. This tells us that it is very important to maintain a `tail` pointer.

**Lookup/Find:** Let’s say that our implementation just returns whether or not the element is in the list, but doesn’t actually return the element. (Not that it matters much).

First, one way to find the element is to go through the entire list and check each element. For each element, we only do $O(1)$ work, so the total time is $O(n)$, an upper bound in all cases.

For a lower bound, notice that if the element is the last element of the list (pretty bad case) or not in the list (another pretty bad case — we aren’t sure which is actually the worst case), then we do in
fact do constant amounts of work for each of \( n \) elements, so the time is \( \Omega(n) \). In summary, the time to lookup/find an element is \( \Theta(n) \).

Notice that another “pretty bad” case is when the element is in the second half of the list. Then, we scan through at least \( n/2 = \Omega(n) \) elements, doing constant amounts of work for each. This also shows the lower bound of \( \Omega(n) \). The point of mentioning this is that we don’t always have to identify an absolute worst case, just one that is bad enough to match the upper bound, which here was \( O(n) \).

**Remove:** To remove an element, we first find it in the list in the same way as the previous case, which we just saw takes \( \Theta(n) \). Once we have found the element, we just have to unlink it from the list (which involves updating a constant number of pointers) and delete it, which takes \( \Theta(1) \). So that part is nice and fast (you can’t be faster than constant time). The total time is \( \Theta(n) + \Theta(1) = \Theta(n) \), which is not that great; but at least we know where the bottleneck is.

### 4 Abstract Data Types and Object-Oriented Design

If we take a step back from what we have been doing with Linked Lists, we can think about the functionality they enable. Let’s say we look at the version we just analyzed, which allows us to do three things:

1. Add an integer \( n \) to the list.
2. Check whether the list contains the integer \( n \).
3. Remove an integer \( n \) from the list.

If all we care about is being able to perform these three operations, it doesn’t matter so much whether we implement them using linked lists, or perhaps arrays, or vectors, or some other method. What we really want is some data structure that stores data internally, and allows us to use these three operations. In other words, we focus on what we want to do, not how exactly it is done.

This is the idea behind specifying an *Abstract Data Type* (ADT). It is defined entirely by the operations it supports, such as the ones above. For instance, the operations above define a very rudimentary version of what the textbook calls a *Bag* data type. (It is also quite similar to what is called *Container* in many languages.)

There may be many different ways of implementing a Bag type, but that is a separate question. In this class, of course, we will focus on both: learning to specify and abstract operations, and learning how to actually implement them.

The notion of an Abstract Data Type goes hand in hand with Object-Oriented Design. In Object-Oriented Design of programs, we group together data items with the operations that work on them into *classes*; classes are data types, and we can then generate instances of these classes called *objects*. The objects have both the data inside them and the functions that operate on the data.

When specifying a class, we usually take good care to keep two things separate: (1) the *specification* of the functions that other classes can call. This is like the specification of an abstract data type, and usually given in a header (.h) file. (2) the *implementation* of how the functions are actually doing their job, and how data items are stored internally. This is in a .cpp file, and hidden from other classes. Hiding it allows us to change the implementation without needing to change the implementation of other pieces of code that use our classes. We will see way more about classes and object-oriented design in the next few lectures.