Lecture Summary

In this lecture, we explored stacks and queues in more detail. We saw what functionality they provide, how they are defined, and how they are implemented. We also saw a more detailed example of how a stack can be used to solve an algorithmic problem very elegantly.

1 Stacks

As we briefly discussed in the previous lecture, a stack is a data structure implementing “Last In First Out” (LIFO) access to data. It only allows us to look at (or remove) the most recently added element. Thus, a visual analogy is a vertical column of building blocks, one on top of another. A new element can only be added to the top, and the only element available for removal is the one on top. Stacks are used internally by the compiler for implementing recursion; generally, they are useful for computations that proceed “inside out”, as we will see below.

Let’s assume that our stack stores elements of some type T. The functionality that a stack offers us more formally is the following.

- Adding an element: void push (const T & data)
- Look at the top (most recently-added) element: T top () (also sometimes, for instance in the textbook, called T peek ()).
- Remove the top (most recently-added) element: void pop().

Notice that the only element that we can look at or remove is the one on top of the stack. Also, in practice, to make the stack easier to use, you would probably add a function bool isEmpty() which returns whether there are any elements on the stack. But the above three are really what makes a stack a stack.

As with other data structures we have seen so far, we’d like to define formally what is and isn’t a stack. As before, a recursive definition is perhaps clearest and simplest: A stack is either

1. the empty stack, or
2. of the form push(S, data) (or in object-oriented notation: S.push(data)), where S is a stack, and data is a data item.

This tells us what is and isn’t a stack, but it doesn’t tell us the semantics of the stack functions, i.e., what exactly they do. The nice thing about stacks is that they are simple enough that we can completely specify the meaning of their functions using formulas. Some people would call these formulas the stack axioms:

1. For all stacks S: s.push(data).top() = data.
2. For all stacks S: s.push(data).pop() = S.

This says that if you read the top element after pushing something on a stack, you get that element back. And if you remove the top element after pushing something on the stack, you get the original stack back. The definition does not say what happens when you call pop or top on an empty stack. (In other words, we don’t define pop or top for the base case of our recursive stack definition.) This is intentional: there is
no real “correct” meaning that we could assign those operations. When you implement a stack, of course
you will have to choose what happens, and you should document your choice. But behavior for cases that
really shouldn’t happen isn’t really something that needs to go in a high-level definition of an abstract data
type, and someone who plans to use a stack to solve a problem to solve some bigger algorithmic problems
shouldn’t rely on specific behavior for faulty inputs.

A second thing to reiterate is that it is really quite amazing that the stack functionality can be completely
specified with two lines of math, and two simple lines at that. When we get to queues later, this will not be
possible. At an intuitive level, for a stack, adding and removing come in pairs that are right next to each
other, while for queues, there can be arbitrarily many elements between adding an element and seeing it
again. This means that to specify the semantics of queue operations, one has to use much more advanced
mathematical specifications.

At an even more philosophical level, this difference is why many mathematically inclined people prefer
functional programming. Recursion by nature resembles stacks: the behavior of a function can be summar-
zied, and then substituted where it occurs. Analyzing for loops, on the other hand, requires something
called fixed point operators, which is much less intuitive. As a result, carefully thought through recursive
solutions are often much more robust.

1.1 Example

As an example of something that is really easy to do with stacks, and wouldn’t at all be obvious without
stacks or recursion, let us look at the following task: You are given a string of characters: lowercase letters
and opening and closing parentheses, brackets, and braces “( [ { } ] )”. The goal is to test if all of the
parentheses/square brackets/braces match. To illustrate the task, here are some examples:

1. “([ab]{c})” is correct: all opening/closing braces match.
2. “[ab]]” is incorrect: the closing square bracket does not match the opening parenthesis.
3. “ab{]” is incorrect: there is no opening brace matching the closing one.

We now outline an algorithm (using a stack) to recognize whether the string has matching parentheses
and brackets and braces.

1. The stack starts out empty.
2. The string is scanned character by character, left to right.
   • Each time an opening brace/bracket/parenthesis is encountered, push it onto the stack.
   • When a letter is encountered, ignore it.
   • When there is a closing brace/bracket/parenthesis, check if it matches what is on top of the stack.
     – If it does, then pop the matching brace/bracket/parenthesis off the stack.
     – If it doesn’t, then signal that this is bad input (mismatch).
     – If the stack was empty, then signal a bad input (missing opening parenthesis).
3. If the stack is not empty at the end, there was an unmatched opening parenthesis/bracket/brace, so
   signal an error.

In a sense, we can regard the above algorithm as a very rudimentary parser. It’s easy to imagine putting
numbers to add or multiply in there, or putting C++ code inside the braces. Indeed, most implementations
of parsers for languages (programming languages, logic formulas, arithmetic expressions, . . .) use stacks
either explicitly or at least implicitly (through recursion).

Looking ahead a little bit to proving correctness of algorithms formally, we would like to argue that the
algorithm does the right thing. Of course, it is our intuition that it does — otherwise, we wouldn’t have
come up with this algorithm. But sometimes, we make mistakes, not just in our implementation, but also in our logic.

There are techniques for mathematically proving that a program/algorithm is correct. These are based on phrasing axioms about what exactly certain programming constructs do, such as our axioms about the stack functions above. Whenever a program contains recursion or loops, such proofs are invariably based on using induction. For loops, a key element of a correctness proof by induction is what’s called a loop invariant: a property that will be true after each iteration of the loop, such that it’s trivial that is true at the beginning (base case), and if we can prove that it holds at the end, our program has done the correct thing. Then, proving that the invariant holds from one loop to the next is exactly the induction step of a proof by induction.

While we won’t do a full correctness proof by induction for this algorithm, for the curious student, the following should be pointed out: the algorithm runs a loop over all characters of the input string. The important part of the loop invariant is that at any time, the top of the stack is the most recent unmatched opening parenthesis/bracket/brace. If one were to really attempt the proof, this isn’t enough to make the induction step work. The actual hypothesis needed is the following: at any stage of the processing, the stack contains all currently unmatched parentheses, in right-to-left order (top-to-bottom in the stack).

### 1.2 Implementation of a general-purpose stack

A stack can be naturally (and quite easily) implemented using either a linked list or array (so long as we dynamically resize it, the way we suggested for implementing a List).

For an implementation based on linked lists, we could insert and delete either at the head or tail. Both are roughly equally easy to implement, though inserting and deleting at the head may be even easier, because even a single-linked list will do:

- To push an item create a new element, link it to the old head, and make it the new head.
- To read the top, return the head’s data item.
- To pop off the stack, remove the head from the list, and set its next element as the new head.

Notice that for none of these operations do we ever need to access any element’s previous element. We could also perform similar operations on the tail of a linked list, but in order to pop the last element, we need to know its predecessor, which will be the new tail; so we would need a doubly linked list, or to scan through the entire list to find the new tail. The latter would of course be inefficient.

An implementation using arrays is not much more difficult: we have an array $a$ and store the size of the stack in some variable, say, called $size$.

- To push an item, we write it into $a[size]$ and increment $size$. If we now exceed the allocated array size, we expand the array by allocating a new larger array and copying the data over, much like we did for the List datatype.

- To read the top, we simply return $a[size-1]$.

- To pop an element off the stack, we decrement the size.

For both implementations, the running time of the top and pop operations is clearly $O(1)$, as we only return a directly accessible element, and update only a constant number of pointers or integer variables. For push, the linked list implementation is also clearly $O(1)$. For the array implementation, it is $O(1)$ except when the array size has to increase (in which case it is $\Theta(n)$). However, if we double the array size every time the array is too small (rather than, say, incrementing it only by 1), then for every operation that takes $\Omega(n)$, the next $n$ operations will require no doubling and thus take $O(1)$. In total, those $n+1$ operations thus take $O(n)$, which means that on average, they take $O(1)$. This is an example of “amortized analysis”.

We discussed this in class in the context of this lecture, but in the class notes, we have inserted a bit of discussion in an earlier lecture on the List datatype.
2 Queues

We also had a brief preview of queues in the last lecture: they provide “First In First Out” (FIFO) access to data. When elements are added into a queue, they can be read/removed only in exactly the order in which they were added. The visual analogue is a conveyor belt, in which elements arrive at the other end for processing in exactly the order in which we place them on the belt — we often visualize queues in similar ways. The functionality of queues (storing elements of some type $T$) is the following:

- Adding an element: `void enqueue(const T & data)`.
- Look at the oldest element in the queue: `T peekfront()`.
- Remove the oldest element from the queue: `void dequeue()`.

As with stacks, you’d likely want to add a function `bool isEmpty()` in practice. Notice again the contrast to stacks: with a stack, you can only access (read/remove) the most recently added element, while a queue only allows you to access the oldest element.

Much like with stacks, we can define formally what is and isn’t a queue. A queue is either:

1. the empty queue, or
2. of the form `enqueue (Q, data)` (or `Q.enqueue(data)` in object-oriented notation), where $Q$ is a queue, and $data$ is a data item.

As we discussed above, the precise semantics of the operations on a queue are much harder to specify, and far beyond what we can do in this class. (And unfortunately, at USC, we do not have a class on semantics of programming languages.)

2.1 Implementation of a queue

Just like with a stack, a queue can be implemented using a linked list or an array.

For an implementation using linked lists, we have two choices: (1) insert at the head and read/remove at the tail, or (2) insert at the tail and read/remove at the head. There’s a slight advantage to the second version, because a singly linked list suffices: when we delete, we only need to set `head = head->next` (and deallocate the old `head`). In order to delete at the tail, we would need to know the tail’s predecessor, which would require a doubly linked list or a linear-time search.

For an implementation using arrays, compared to stacks, we now need two integer indices: one (say, `newest`) for the position of the newest element in the array, and one (say, `oldest`) for the position of the oldest element. When enqueueing a new element, we can write it into $a[\text{newest}]$ and increment `newest`. To implement `peekfront`, we can return $a[\text{oldest}]$, and to dequeue the oldest element, we can just increment `oldest` by one. As before, when enqueueing a new element and exceeding the current array size, we need to expand the array.

One downside of this implementation is that it may be quite wasteful of space. In particular if the queue is used for a long time, after a while, both `newest` and `oldest` may be large, which means that most of the array is unused. Instead, we could have the array “wrap around”, by doing calculations of the index `newest` modulo the array’s size. This makes better use of space, but we now have to be careful not to overwrite the older elements with newer elements. None of this is a huge obstacle, but one has to be a bit more careful in the implementation, which perhaps makes an implementation based on linked lists slightly more attractive.

The running time analysis is pretty much identical to the one for a stack. All operations are $O(1)$ because they just involve updates of a small number of variables. As before the only exception is the possible expansion of the array, which may take $\Theta(n)$, and which can again be amortized if the array size doubles (or gets multiplied with some number $b > 1$, rather than having something added to it) whenever the array is too small.
3 Why use restrictive data structures?

As we saw, a stack only allows access to the most recently added element, while a queue only allows access to the oldest element in the data structure. Couldn’t we have a data structure that allows access to either, as we need it?

Indeed, we can: there is a data structure called Deque, which does exactly that: it allows adding and accessing (reading/removing) elements at both ends. If you understood how to implement a stack and a queue, implementing a Deque will be easy for you. So why wouldn’t we use a Deque everywhere instead of stacks or queues?

Or, perhaps more strongly: why don’t we just use the C++ vector class for all data structures? It combines the functionality of a stack, queue, and List in the sense in which we defined it in previous lectures. So it is much more powerful.

One answer to this question is pedagogical: in this class, we want to learn how these data structures work internally, and using a powerful tool without understanding it does not contribute to learning.

A second answer arises when we have to implement the data structures ourselves: if a data structure provides more functions, then it is more work to implement, and offers more places to make implementation mistakes. So if we implement them ourselves, there’s something to be said for keeping our data structures simple. But of course, that argument does not apply when we use data structures provided in a programming language.

A third, and often appropriate, answer is that in order to implement more functions, the implementation of other functions may be less efficient. For instance, in order to allow access to each element in constant time, vector is implemented internally as an array. That may slow down the implementation of some other functions, which could be implemented faster if we didn’t also need access by index.

The fourth, and real, answer lies in our goal of writing code that we and our collaborators can understand easily in the future. If you write code in which you explicitly use a variable of type Stack, you make it clear that all your code needs is the Stack functionality, which will help others (and yourself, if you return to your code weeks or months later) parse the logic of your code. If instead, you had used Deque or Vector types, others may wonder whether somewhere buried in line 1729 of your code, you actually access specific elements, or need to access both the head and tail of your Deque. That makes it harder to understand the logic. So in general, it is a good idea to think through your code’s logic ahead of time, and only declare those data structures that you really need.

In class, someone asked whether we couldn’t just make this clear with our variable names, say, by writing something like vector<int> stack. If you do this, you might convey your intention. But of course, another interpretation would be that you had initially written Stack stack, and then later realized that you actually needed extra functionality. But because you didn’t want to rename the variable everywhere, you just changed its type to vector<int>, so even though the variable is called stack, it’s not actually used as one.

So the upshot is that you should ideally declare your variables to be of the type that you actually need, not giving yourself extra functionality you don’t intend to use. In particular, you may not find too many immediate uses for a Deque, since most of the time, you’ll want either a Stack or Queue.