SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2020

Topic Notes: Linear Structures

The structures we've seen so far, `Vectors`/`ArrayLists` and linked list variations, allow insertion and deletion of elements from any position in the structure. So there is an "order" to the structure, but that order does not restrict how we access or modify the structure.

There may be significant differences in efficiency when modifying or accessing the structure in a way that is perfectly legal but may be hard to implement efficiently. Moreover, it may not be clear to the user of a structure which operations are efficient and which will incur a significant cost.

Linear structures are more restricted, allowing only a single `add` and single `remove` method, neither of which allows us to specify a position within the structure, and in the case of `remove`, we also cannot specify an element to remove by value.

Why would we want to restrict our structures in such a way? It seems we would be unnecessarily limiting what a user of the structure can do. The answer is that by placing more restrictions on a structure, we can generally allow for more efficient implementation of its supported operations. Since we know how it will be used (and in fact enforce this by limiting the number of public methods), we can make sure we use an appropriate *internal representation* to ensure efficiency of the supported operations.

Linear Structures: High-Level Concepts

We will look closely at two particular highly restricted linear structures:

- *stack*: all additions and removals must come at same end of the structure, resulting in a "last-in, first-out" or "LIFO" behavior
- *queue*: all additions must happen at one end of the structure, and all removals from the other, resulting in a "first-in, first-out" or "FIFO" behavior

We will also consider a third option, which provides a bit more general but still highly restricted functionality, called a *deque*, or double-ended queue. A deque allows additions and removals at either end of the structure, but not in the middle.

Stacks

We start with *stacks*. The idea is very simple. The most recently added element in the structure is the only one which can be removed next. Hence, the common description of stacks as a *last in, first out (LIFO)* structure.

As a real-world analog, consider a stack of trays. New trays are added at the top and trays are also taken from the top when needed. (Sane) people don't go in and try to take a tray from the middle or from the bottom of the stack.

One way to describe a stack is recursively: A stack is either empty or has its top element sitting on a (smaller) stack.

All additions and deletions take place at the top of the stack.

When dealing specifically with stacks, we traditionally refer to addition as *push*, removal as *pop*, motivated by the analogy with a spring-loaded stack of trays, and the look-without-removing as a *peek*. We'll sometimes use these stack-specific and general names interchangeably (methods will exist with both names in the structure package's implementations): `add` and `push`; `remove` and `pop`; and `get` and `peek`.

So the three major operations allowed on a stack are:

- push/add
- pop/remove
- get/peek

A Java interface for a stack might look like this:

```
public interface StackInterface<E>
{
    void push(E element);
    E pop();
    E peek();
    boolean isEmpty();
    void clear();
}
```

Applications include:

- run-time stacks on computers to manage memory for method/function/procedure calls
- maze running
- computing arithmetic expressions in "postfix" notation

What would be output by this code segment?

```
myStack = new Stack<Integer>();
Stack<Integer> temp = new Stack<Integer>();

myStack.push(2);
myStack.push(4);
myStack.push(8);
myStack.push(10);
temp.push(myStack.pop());
temp.push(myStack.pop());
myStack.pop();
myStack.peek();
myStack.push(temp.pop());
myStack.push(temp.pop());
while(!myStack.isEmpty()){
  System.out.print(myStack.pop() + " ");
}
```

Queues

The other linear structure we will consider is the *queue*, a *first in, first out (FIFO)* structure.

The only way we are allowed to use a queue is by adding values to the "end of the line" and taking values out from the "front of the line".

Applications include:

- Waiting lines
- Event Queues: Keyboard and mouse events in interactive programs on time-shared computers.

The zyBook also uses the terms `push` and `pop` for the `add` and `remove` operations of queues, but that is unusual. Usually these are replaced by the queue-specific terms *enqueue* and *dequeue*, which are equivalent to `add` and `remove`. There is also a `peek`, which is again the equivalent to `get`: this returns the value that would be dequeued next without actually dequeuing it.

A more traditional queue interface might look like this:

```
public interface QueueInterface<E>
{
    void enqueue(E element);
    E dequeue();
    E getFront();
    boolean isEmpty();
    void clear();
}
```

What happens with this code segment?

```
myQueue = new Queue<Integer>();
Queue<Integer> temp = new Queue<Integer>();

myQueue.enqueue(2);
myQueue.enqueue (4);
myQueue.enqueue (8);
myQueue.enqueue (10);
temp.enqueue(myQueue.dequeue());
temp.enqueue(myQueue.dequeue());
myQueue.dequeue();
myQueue.getFront();
myQueue.enqueue(temp.dequeue());
myQueue.enqueue(temp.dequeue());
while(!myQueue.isEmpty()){
   System.out.print(myQueue.dequeue() + " ");
}
```

Deques

The *deque*, or doubly-ended queue, is a linear structure that allows elements to be added at the start or end of the structure.

Since there is not a unique `add` or `remove`, the operations are typically named `addFirst`, `addLast`, `removeFirst`, and `removeLast`.

Some common application of deques include:

- the "undo" history of an application
- tracking a web browser tab's history

In each of these cases, we would usually want the stack functionality for an undo operation or when someone hits the back button for a browser tab as in each case we would need the most recently added element. However, we would also want to be able to remove the oldest entries at some point to prevent these from becoming unreasonably large.

Implementing Linear Structures

Before we consider some specific applications of these structures, let's think about how they can be implemented efficiently.

As we think about this we reemphasize a key point. The fact that these each have a very limited set of supported operations means that we can use underlying structures that implement those

operations efficiently, and we don't care if other operations would not be efficient since we know they'd never happen!

Implementations of stacks, queues, and deques also typically include familiar additional operations like `contains` and the ability to create an iterator over the structure.

As we consider our options here, let's recall the possible underlying structures we can use to implement stacks, queues, and deques: arrays, `ArrayLists`, singly-linked lists, circular lists, and doubly-linked lists. Our goal will be to choose an *orientation* of our linear structure within the underlying structure such that the operations we need to support can be done efficiently and with as little memory overhead as possible.

When implementing stacks, we will say that elements are pushed and popped at the *top* of the stack, and the items that have been in the stack are at the *bottom* of the stack.

When implementing queues, we will say that elements are added to the *back* of the queue and removed from the *front* of the queue.

Deques are essentially a symmetric structure, so the orientation in the underlying structure is less important.

Array-based Stack Implementation

To build a stack using an array as its internal representation, we will follow the same idea as the way an `ArrayList` is built from an array: a stack containing n values will have those packed into the first n slots of the array, and there will be a variable to track how many slots are occupied with meaningful values.

We first need to decide if the orientation should place the top of the stack at the start or at the end of the occupied portion of the array. It is not a difficult decision here. If we were to place the top of the stack at the start of the array, every push operation would need to shift n values "up" and every pop operation would need to shift n values "down", so each would be $O(n)$. Placing the top at the end quickly turns each of these into an $O(1)$ operation.

The only problem is if you attempt to push an element when the array is full.

That opertion should fail, throwing an exception. This is unexpected behavior from a stack, and would be potentially problematic for users.

We can also throw an exception when trying to remove from an empty stack, but that's a misuse of the structure, not a shortcoming of our implementation.

Well, we have a nice data structure that will eliminate the "full" problem: the `ArrayList`, so...

`ArrayList`/`Vector` Stack Implementation

it could make more sense to implement this with an underlying `ArrayList` to allow unbounded growth (at cost of occasional $O(n)$ push when the internal array needs to grow).

The implementations of each of the stack operations are done almost trivially by making use of the

underlying `ArrayList` methods.

What about the complexity of the supported operations?

- All operations are $O(1)$ with exception of the occasional `push` (when the `ArrayList` needs to grow) and `clear`, which should replace all entries by `null` in order to let them be garbage-collected.

So this is very nice. It's easy to implement, building on the `ArrayList` that takes care of resizing for us. However, it has a few disadvantages:

- `add`/`push` is $O(n)$ when the `ArrayList` needs to grow.
- Space usage is proportional to the largest internal `ArrayList` needed for the life of the stack. If we place a large number of elements in the stack at some point and later will have only a few, we are still using all of that memory.

We can take care of both of these by using a linked list as our internal representation.

Linked List Stack Implementation

We considered a few types of linked lists – which are appropriate for use as the internal structure of a stack?

To decide this, we consider which operations we need. With the `ArrayList` implementation, we added things at the end, and only doubly linked lists allowed efficient deletions from the end. So a doubly linked list would work.

However, there's no rule that says the element at the top of the stack has to be at the end of the list the way it was at the end of the `ArrayList`. The restrictions on the allowed operations of a stack give us another good option. We can keep the top at the head of the list and always use `add`/`remove` operations on the first element. Singly Linked Lists are very good at these two operations.

The linked list implementation is very similar to the `ArrayList` implementation, it just orients the stack within the list so that the top of the stack is at the head of the list.

What are the complexities for our stack operations with these implementations?

- `get`, `pop`, and `isEmpty` are all $O(1)$ for the three implementations.
- `push` can be $O(n)$ in worst case for an `ArrayList`-based stack, but on average it is $O(1)$. For the other implementations, it is always $O(1)$.
- `clear` is $O(1)$ for the list-based stack and array-based stack, but $O(n)$ for the `ArrayList`-based stack, if the `ArrayList`'s `clear` method sets all array entries to `null`.

- The array-based stack uses a fixed amount of space: this wastes space if you reserve too much, while the program will fail if there is too little.
- `ArrayList`-based stack provides more flexibility, but at the cost of occasional significant delays (though average cost of `push` is $O(1)$). Also, space will never be given back once the `ArrayList` grows large at some point in the stack's lifetime.
- For the list-based stack, all operations are $O(1)$ in the worst case, but it requires $O(n)$ extra space for the links.

Linked List Queue Implementation

We now turn our attention to queue implementations.

If we want to use a linked list to implement a queue, we need to orient our queue within the list by deciding which end to add to and which end to remove from, and which of our list structures is appropriate.

Clearly, a singly linked list is not good enough, since that one only had a `head` pointer and either `add` or `remove` would need to be an $O(n)$ operation, depending on which end of the list represents the head of the queue.

A doubly-linked list would certainly work. The important operations would be $O(1)$, but that implementation has an additional space overhead for the extra references in the doubly-linked nodes.

How about a circular list? There we at least have direct (or nearly direct) access to both `head` and `tail` references.

But should we `add` to the `head` and `remove` from the `tail`, or vice-versa?

For circular lists, `add` to the `head`, `add` to the `tail` and `remove` from `head` are all $O(1)$, but `remove` from `tail` is $O(n)$. So the latter should be avoided.

We can do exactly this if our queue is oriented with the front of the queue at the head of the list and the back of the queue at the tail of the list. Now, our queue operations are $O(1)$. And the list nodes remain singly-linked.

Let's reemphasize a key point here that it is tempting to use a doubly-linked list as the underlying structure here since we know all of the add/remove operations at either end are $O(1)$. But since we know our queue does not need to support all of those operations, we can choose a lighter-weight structure, the singly-linked circular list, and by orienting the queue properly, make both our `enqueue` and `dequeue` operations efficient with one fewer reference per queue element.

`ArrayList` Queue Implementation

However we orient a queue within an `ArrayList`, either the `enqueue`/`add` or `dequeue`/`remove` operation will need to be $O(n)$.

In structure's implementation, the front of the queue is at index 0 of the underlying `ArrayList`, making the `dequeue`/`remove` method the one that's $O(n)$.

It seems like `ArrayLists` are not the best choice for implemenation of queues.

Clever Array-based Queue Implementation

We can also have an array-based queue implementation, but it is a bit trickier!

Suppose we know that our queue will never contain more than some constant number of elements. That is, we have an upper bound on the maximum size of the queue.

We can use this to solve the problem of the queue "walking" off one end of the array.

Consider a "circular" array implementation, where we maintain references (array indices, really) `head` and `tail` referring to the front and back of the queue, respectively.

We increment the `head` reference on a `dequeue` and increment the `tail` on an `enqueue`. If nothing else is done, you soon bump up against the end of the array, even if there is lots of space at the beginning (which used to hold elements which have now been removed from the queue).

To avoid this, we pretend that the array wraps around from the end back to its beginning. When our head or tail "walks off" the end of the array, we go back to beginning by modular arithmetic.

This approach does require careful consideration of some picky details.

Note in particular that logically we want to have a `head` and `tail` to track where to remove and add values, respectively, but there is no `tail` index need be stored, it is instead computed from the `head` and a stored `count`. This allows us to tell the difference between an empty and a full queue, each of which would occur when the `head` and `tail` refer to the same array element.

The complexity of operations for the array-based queue is the same as for the circular list-based queue. (Both $O(1)$)

The big disadvantage of the array-based queue is its limited size, and it is only useful in cases where we know an upper bound on the number of elements to be stored in the queue at any given time.

Deque Implementation

A deque implementation would need to support efficient adds and removes at both ends of the structure.

- The only underlying structure we have available that is capable of that, and which also would not have a capacity limit, is the doubly-linked list.
- Like with queues, any `ArrayList`-based deque would have adds and removes at one of the ends necessarily $O(n)$.
- We could augment the clever array-based queue to have deque functionality with appropriate management of both a head and a tail index. All operations would be $O(1)$, but it would have

a capacity limit.

Applications of Linear Structures

We will consider some applications of these structures in upcoming labs and problem sets. They will also be used as tools in algorithms later in this course and in many fields.