Computer Science 501
Data Structures & Algorithms
The College of Saint Rose
Fall 2015

Topic Notes: Linear Structures

The structures we've seen so far, `Vectors` and linked lists, 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.

Why would we want to restrict our structures in such a way? More restrictions on a structure generally allows 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.

The basic operations on our linear structure are specified in the `Linear` interface in the structure package.

```
public interface Linear<E> extends Structure<E>
{  // we have size, isEmpty, & clear from Structure

    public void add(E value);
    // pre: value is non-null
    // post: the value is added to the collection,
    //       the replacement policy not specified.

    public E remove();
    // pre: structure is not empty.
    // post: removes an object from container

    public E get();
    // pre: structure is not empty.
    // post: returns ref to next object to be removed.
}
```

We will look at two particular highly restricted linear structures:

- *Stack*: all additions and deletions at same end: LIFO (last-in, first-out)

- *Queue*: all additions at one end, all deletions from the other: FIFO (first-in, first-out)

Stacks

We start with stacks. The idea is very simple. 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.

Stacks can be described 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.

We traditionally refer to addition as *push* and removal as *pop*, motivated by the analogy with a spring-loaded stack of trays. Here we'll use both names interchangeably (and provide both methods) for the `add` or `push` and `remove` or `pop`. There are also two names for the operation of looking at the element on top of the stack without removing it: `get` or `peek`.

So the three major operations allowed on a stack are:

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

The structure package includes an interface to which all stack implementations must adhere:

See Structure Source:
`/home/cs501/src/structure5/Stack.java`

Applications of Stacks

Stacks are LIFO ("last in, first out"), making them an appropriate structure for maintaining "Undo" information in various programs. Consider your favorite text editor (emacs) or word processor.

Computing Arithmetic Expressions

Some machines have stack-based machine language instructions. Such machines require all arithmetic calculations to take place using data on the stack and that the result is placed on the stack at the end.

Such a machine might have instructions such as:

- `PUSH A` – push the value from variable `A` onto the stack.
- `POP A` – pop a value from the stack and store it in variable `A`.

- `ADD` – pop the top two elements from the stack, add them and push the result back onto the stack.
- `SUB`, `MULT`, `DIV`, etc. are similar.

We will see in an upcoming lab assignment that the PostScript language (understood by many printers) is a stack-based language.

On such a machine, to calculate

`A = X * Y + Z * W`

one would need to use the following sequence:

```
PUSH X
PUSH Y
MULT
PUSH Z
PUSH W
MULT
ADD
POP A
```

How would you generate this code?

The ideas is to write the expression in postfix form: each operator comes after its operands.

For example, `2 + 3` becomes `2 3 +`

There are three primary ways to represent an arithmetic expression:

- infix notation: `(2 + 3) * 4 - 5`
- (Polish) postfix notation: `2 3 + 4 * 5 -`
- (Polish) prefix notation: `- * + 2 3 4 5`

In general, to convert an infix expression to postfix notation:

1. Write the expression fully parenthesized.
2. Recursively move operators after operands, dropping parentheses when done.

For example:

`X * Y + Z * W` ⟶ `(X * Y) + (Z * W)` ⟶ `(X Y *) (Z W *) +` ⟶ `X Y * Z W * +`

Note: in Polish notation, parentheses are no longer needed.

Once in postfix, it's easy to generate code as follows:

- If we encounter an operand, generate a `PUSH` command
- If we encounter an operator, generate the command to do that operation.

Our expression above will compile to

```
PUSH X
PUSH Y
MULT
...
```

You will do something very similar to this in a lab soon.

Run-Time Stacks on Computers

Probably the most common use of stacks is one we use pretty much every time we run a program, but rarely think about – the *call stack* of a program.

The text illustrates how the call stack works for a fairly complex recursive method. In class, we will look at a simpler case – a binary search.

Recall the binary search on arrays that we looked at a while back.

See Example:
`/home/cs501/examples/BinSearch`

What's really happening when we run this program?

First, `main` gets called and its parmeters and local variables are allocated on the stack:`args`, `size`, `orderedInts`, `i`.

We then allocate the arrays and fill them in. The memory from these is not part of the call stack. It is retrieved from a separate part of memory called the *heap*. We won't worry about the details of that at the moment.

Then, we make a call to `search`. Its *formal parameters* (`elts`, `findElt`) are allocated on the stack and are initialized to the values specified by the caller as *actual parameters*.

This is more formally known as a *call frame* or an *activation record* and keeps track of some other things, like where to return to when the method returns.

Next, `binsearch` is called and its parameters and local variables are allocated: `elts`, `low`, `high`, `findElt`, `mid`.

Then it gets called again and a new activation record is created: a brand new copy of its parameters and local variables.

The recursive calls continue on, piling up activation records on the stack. When methods return, we pop the activation records off the stack and continue doing what we were doing before the method was called. We know where to go back by the return address that was allocated as part of the activation record.

The text shows how you can change the recursive quicksort method into an iterative quicksort procedure by building your own call frames and putting them on a stack. It's worth reading, but we won't go over it in class.

Stack Implementations

The structure package also includes an abstract class `AbstractStack` which provides some of the methods with duplicate names, so they need only be implemented once in an actual stack implementation. Thus, our actual implementations do not need to provide `push`, `pop`, and `peek`, but only `add`, `remove`, and `get`.

As we saw earlier, stacks are essentially restricted lists and therefore have similar representations.

Array-based implementation

Since all operations are at the top of the stack, an array implementation is reasonable. One is provided in the class `StackArray` in structure.

```
public class StackArray<E> implements Stack<E>
{
    protected int top;
    protected E[] data;
    ...
```

The array implementation keeps the bottom of the stack at the beginning of the array. It grows toward the end of the array. We define `top` to be the index of the element currently at the top of the stack.

So what does `top` need to be set to when a stack is created? It should be -1! There is no element at the top of the stack in this case, so we point to the place where there would be one.

We can pretty easily develop some of this implementation:

```
public StackArray(int size) {
        data = (E[])new Object[size];
        clear();
    }

    public void clear() {
        top = -1;
    }

    public void add(E item) {
        Assert.pre(!isFull(),"Stack is not full.");
        top++;
        data[top] = item;
    }

    public E remove() {
        Assert.pre(!isEmpty(),"Stack is not empty.");
        E result = data[top];
        data[top] = null;
        top--;
        return result;
    }

    public E get() {
        // raise an exception if stack is already empty
        Assert.pre(!isEmpty(),"Stack is not empty.");
        return data[top];
    }

    public int size() {
        return top+1;
    }

    public boolean isEmpty() {
        return size() == 0;
    }

    public boolean isFull() {
        return top == (data.length-1);
    }
```

See Structure Source:
`/home/cs501/src/structure5/StackArray.java`

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

```
    Assert.pre(!isFull(),"Stack is full.");
```

will 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.

Thus it could make more sense to implement this with `Vector` to allow unbounded growth (at cost of occasional $\Theta(n)$ delays on a push).

The basics of this implementation:

```
    protected Vector<E> data;

    public StackVector() {
        data = new Vector<E>();
    }

   public void add(E item) {
        data.add(item);
    }

    public E remove() {
        return data.remove(size()-1);
    }

    public E get() {
        // raise an exception if stack is already empty
        return data.get(size()-1);
    }

    public boolean isEmpty() {
        return size() == 0;
    }

    public int size() {
        return data.size();
    }

    public void clear() {
        data.clear();
    }
```

See Structure Source:
`/home/cs501/src/structure5/StackVector.java`

What about the complexity of the supported operations?

- All operations are $\Theta(1)$ with exception of the occasional `push` (when the `Vector` 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 `Vectors` that takes care of resizing for us. However, it has a few disadvantages:

- `add`/`push` is $\Theta(n)$ when the `Vector` needs to grow.
- Space usage is proportional to the largest internal `Vector` 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 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 `Vector` 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 `Vector`. 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. `SinglyLinkedLists` are very good at these two operations.

The linked list implementation is very similar to the `Vector` implementation, it just puts things at the other end.

See Structure Source:
`/home/cs501/src/structure5/StackList.java`

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

- `get`, `pop`, and `isEmpty` are all $\Theta(1)$ for the three implementations.
- `push` can be $\Theta(n)$ in worst case for `StackVector`, but on average it is $\Theta(1)$. For the other implementations, it is always $\Theta(1)$.
- `clear` is $\Theta(1)$ for `StackList`, $\Theta(n)$ for `StackArray` and `StackVector`.
- `StackArray` uses a fixed amount of space: this wastes space if you reserve too much, while the program will fail if there is too little.
- `StackVector` provides more flexibility, but at the cost of occasional significant delays (though average cost of `push` is $\Theta(1)$). Also, space will never be given back once the `Vector` grows large at some point in the stack's lifetime.
- For `StackList`, all operations are $\Theta(1)$ in the worst case, but it requires $\Theta(n)$ extra space for the links.

Queues

The other linear structure we will consider is the *queue*, a FIFO ("first in-first out") 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 Java, the Mac, or time-shared computers.

The `Queue` interface:

See Structure Source:
`/home/cs501/src/structure5/Queue.java`

Rather than `push` and `pop`, we have the queue-specific terms `enqueue` and `dequeue`, which are equivalent to `add` and `remove`. There is also a `peek`, which is the equivalent to `get`: this tells us the value that would be dequeued without actually dequeuing it.

Queue Implementations

As with stacks, we start with an abstract class, `AbstractQueue` that will take care of the queue-specific methods that can be implemented as calls to other methods. We will not need to worry about these in our actual implementations.

Linked List implementation

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

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

A `DoublyLinkedList` would certainly work. The important operations would be $\Theta(1)$, but that implementation has an additional $\Theta(n)$ space overhead above and beyond the references needed to store our `SinglyLinkedLists`.

How about a `CircularList`? 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 $\Theta(1)$, but `remove` from `tail` is $\Theta(n)$. So the latter should be avoided.

We can do exactly this if our queue does its `enqueue`/`add` to the `tail` and its `dequeue`/`remove` from the `head`.

All of the operations are $\Theta(1)$.

See Structure Source:
`/home/cs501/src/structure5/QueueList.java`

`Vector` implementation

We can implement a queue using a `Vector` with the `head` at index 0 and `tail` moving to the right as items are enqueued. Addition of new elements is done using the `Vector`'s `add` method, while `dequeues` set the `Vector` entry at the `head` slot to `null` and increment `head` (move it one place to the right). This allows both `add`/`enqueue` and `remove`/`dequeue` to be $\Theta(1)$.

However, there is a big problem: `enqueues` and `dequeues` will result in elements removed from the left side and added to the right side. Thus the queue will "walk" off to the right over time, even if it remains the same size (what happens after 100 `enqueues` and 100 `dequeues`?). Yes, the `Vector` will keep growing, but each time it grows, it will cost twice as much. And we never give back the memory.

One possibility to improve this a bit is to reset `head` to 0 each time our queue is empty, but this will not help for queue usages where this case is unlikely.

Alternatively, we could change the `dequeue` so that the element at the `head` (index 0) is actually removed. However, this would make the `dequeue` method $\Theta(n)$. [Note: This is the actual implementation.]

See Structure Source:
`/home/cs501/src/structure5/QueueVector.java`

Given that `add`/`remove` from the front of a `Vector` either must waste space or have $\Theta(n)$ operations at least some of the time, it seems like `Vectors` might not be the best choice for our queues.

Clever array 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. This is 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) referring to the `head` and `tail` of the queue.

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 we walk off end of the array, we go back to beginning:

```
    index = (index + 1) mod arraysize
```

Notice that the representation of a full queue and an empty queue can have identical values for front and rear.

Once we have this idea down, it remains to worry about some picky details.

- `head` refers to the next slot where we can find an element for a `dequeue`.
- Should `tail` refer to the slot containing the most recently `enqueue`d item or the empty slot where the next `enqueue`d item should be replaced?
- In either of these cases, how would we be able to tell the difference between an empty queue and a full queue?

The solution used by the `QueueArray` is to keep track of the `head` and a `count` of the number of items currently in the queue. Note that `head` and `count` together take the place of `tail`. When we need the `tail` reference, we compute it from `head` and `count`:

```
  int tail = (head + count) % data.length;
```

See Structure Source:
`/home/cs501/src/structure5/QueueArray.java`

Alternatively, we could keep track of the `head` and `tail` rather than a `count`. To be able to determine whether a queue is full or empty requires that we always leave an empty slot and that we keep the `tail` reference pointing to the slot where the next element to be `enqueue`d will be placed. Here, an empty queue will have `head == tail`. The queue is full if `head = (tail + 1) % data.length`. When this is true, there would still be one empty slot in the queue, but this is the cost to be able to determine whether the queue is empty or full without the `count`.

The complexity of operations for the `QueueArray` is the same as for `QueueList`. (Both $\Theta(1)$)

The big disadvantage of the `QueueArray` 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.

Final note: the `QueueArray` does not bother to set the array entry for `dequeue`d element to `null`. Similarly for items removed with `clear`. This saves a bit of time, but the Java garbage collector would not be able to clean up the `dequeue`d elements even if they are not in use elsewhere, not until the slot is reused after the queue contents wrap around through the array. It would probably make sense to be more consistent with `Vector` and clean up these references.

Stack/Queue example: Running a Maze

The text has a section about using stacks and queues to find a path through a maze. We won't go through that in great detail (at least not by tracing the details of the code) but it's a good example of where stacks and queues are useful.

I have original text example copied and slightly modified (with some example input mazes):

See Example:
`/home/cs501/examples/MazeRunner`

At any given time, the stack in this example contains a collection of places (moves) we still need to try, given where we've already been.

We start (appropriately) at the start location by `push`ing its location onto the stack. Each step of the solution process involves a `pop` of the next move to make. If we haven't arrived at the goal position, we look in each of the 4 directions, and if we have not yet been there and there is no wall, we `push` that location to be tried later.

How does this work for a maze with no internal walls?

Now what happens if we make the stack into a queue? How will we proceed on the maze with no internal walls?

Using a stack will result in a *depth first* search for the goal. Using a queue will result in a *breadth first* search.

We will see stacks and queues along the way later in the semester, and you are using stacks in the postscript lab assignment.