Lecture 12

Bit Manipulation

We can use a bit vector to represent membership in a subset. Assume we have a set of three items – "`a`," "`b`," and "`c`." We can represent a subset of this set as a 3 bit integer. We assign "`a`" to the leftmost bit, "`b`" to the middle bit, and "`c`" to the rightmost bit.

`0 1 0` - subset {`b`}
`1 0 1` - subset {`a, c`}
`0 0 0` - subset {} (i.e., the empty set $\emptyset$)

If we want to test for membership in our bitvector, we can use a combination of the Java bit-shift operators, and bitwise `and` and `or`.

Left shift "<< – shifts all bits in the value to the left one digit. e.g.,

$5 = 0101$, $5 << 1 = 1010$
$1 = 0001$, $1 << 0 = 0001$
$1 = 0001$, $1 << 3 = 1000$

Note that `int`s are 32 bits wide. Bits that are shifted past the 32^{nd} bit are lost.

Right shift ">> – shifts all bits in the value to the right one digit.

$5 = 0101$, $5 >> 2 = 0001$
$5 = 0101$, $1 >> 3 = 0000$

As before, bits shifted past the first bit are lost.

Bitwise `and` "&" – returns result of `and`-ing bits in same locations together.

```
0110
1101
----
0100
```

You can think of the ones as true values, and the zeros as false values.

Bitwise `or` "|" – returns the result of `or`-ing bits in same locations together.

```
0110
1101
----
1111
```

So how do we tell if the i^{th} item is in the set?

Shift the value representing the subset to the right by the appropriate number of bits, then & the value by one. If the n^{th} bit was one, then the result of the `and` will be 1. Otherwise, it will be 0.

e.g., is the 3^{rd} element in the subset represented by bitvector 10110101?

10110101 >> 3 = 00010110
00010110 & 00000001 = 00000000 NO!

Remember we number from [0..n-1], so we have the 0^{th} bit, then the 1^{st} bit, then the 2^{nd}, then the 3^{rd}, and so on.

Stacks

Stacks are called LIFO (Last In, First Out) structures.

We use stacks for lots of things, most importantly is probably the call stack in most computer systems. When we make a new method call, we create a new stack frame, and push it onto the call stack. The stack frame holds our local variables, as well as pointers to the calling methods stack frame, and other information.

Stacks are simpler data structures, they support a very small number of operations. At least:

- `push(E value)` – add a value to the top of the stack.
- `pop()` – remove and return the value on the top of the stack.

Typically, also:

- `peek()` – look at, but don't remove, top of stack.
- `empty()` – true, iff no items on stack.

From the operations, we see that we only have access to the top of the stack at any given time. That is, if something is on the stack, but not the first item, we would need to remove, or `pop`, items off the stack until we got to the one we wanted.

The bailey library implementation of stacks starts with the interface `Stack.java`. Since this extends `Linear`, we have some additional methods to implement, like `add` and `remove`.

Bailey uses an abstract class called `AbstractStack.java` to make this easier. It defines `push` and `pop` in terms of `add` and `remove`. This means that the implementation of `push` and `pop` will be the same regardless of implementation. By defining it in an abstract class, different implementations can `extend` the abstract class, and not have to repeat this work.

The linked list implementation of a stack in the Bailey library uses a standard singly linked list implementation.

`get()` is `getFirst()` from `SinglyLinkedList`. `add()` is `addFirst()` from `SinglyLinkedList`. `remove` is `removeFirst()` from `SinglyLinkedList`. By using the "First" versions for all operations, we can guarantee that we uphold the definition of the Stack.

For the array based implementation, a `top` pointer, which is just an integer representing the index of the top item on the stack, is used. That is all we need, sine we only access the top of the stack.

Let's look at run-times for the basic operations:

	StackList	StackArray
push	O(1)	O(1)*
pop	O(1)	O(1)
peek	O(1)	O(1)
empty	O(1)	O(1)

We have seen in class how a linked list can add and remove from the front of the list using the `head` reference in constant time. `peek` only requires looking at the `head`'s value, and `empty` only requires testing `head` against `null`.

For our array based implementation, since we are adding to the end of the array, and we have direct access to each element in the array, we can add and remove in constant time. Likewise, `peek` is just getting the value at a specified index. When `top` holds the value -1, we know that the stack is empty.

*One problem with the array implementation is that the array can get full. This would seem to give an advantage to the linked list implementation. We can resolve this by creating more space, like the `ArrayList` class does. Of course, this means that any single add operation may be $O(n)$ to copy the values to the new array, but will still have $O(1)$ ammortized time.