Announcements

• precall 2 is due now!
• lab 2 will have you time some algorithms on a variety of input sizes, then plot the growth rate of those algorithms to see if you can "guess" their running times.
• no class on Friday (Alexa is out of town)
• rearranged the schedule a bit so that you'd have enough done for next week's problem set.

ArrayList Running Times

Now that you've finished your MyArrayList, let's look at the RT of its methods.

```java
public class MyArrayList<T> extends AbstractList<T> {
    T[] data;
    int size;

    // What is the running time of this constructor, as a function of "startSize"?
    public MyArrayList( int startSize ) {
        int capacity = 2;
        while( startSize > capacity ) {
            capacity = capacity*2;
        }
        data = (T[]) new Object[capacity];
    }
    public MyArrayList() { this(2); }
}
```

Next, some easy ones:

```java
T[] data
int size

int size() {
    return size
}

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

clear()
size = 0
data = (T[]) new Object[2] // allows for garbage collection

T get(int index) {
    if( (index < 0) || (index >= size) ) throw Exception
    return data[index]
}
```

Next, some more difficult:

```java

public MyArrayList(T[] other) {
    data = (T[]) other;
}

private T[] copyArray() {
    T[] tempArray = (T[]) new Object[data.length];
    for( int i = 0; i < data.length; i++ ) {
        tempArray[i] = data[i];
    }
    data = Object.cloneArray(T[], tempArray);  // Java 5 only
    return tempArray;
}
```

First, the constructors:

Now that you've finished your MyArrayList, let's look at the RT of its methods.

ArrayList Running Times
ArrayList Running Times

The declaration is implementation independent:

- The declared operations:
  - the data type:
  - the data type:

This is an example of an abstract data type (ADT). We've defined both:

- the ADT;
- the ADT implementation.

For this ADT, you'll see how we implement all in O(1) time.

### Variable

The empty ADT - clear all items from the stack.

- The number of elements in the stack
- the flag (true or false) indicating if the stack is empty or not
- the top of the stack
- the data on the top of the stack
- the data on the top of the stack
- the data on the top of the stack

### Methods

#### set(int index, T element)

If (index < 0) || (index >= size) throw Exception

```java
ret = data[index]
data[index] = element
return ret
```

#### remove(int index)

If (index < 0) || (index >= size) throw Exception

```java
ret = data[index]
for (i = index; i < size-1; i++)
data[i] = data[i+1]
size--
return ret
```

### Getting more tricky:

#### resize()

```java
newdata = new array of size [data.length*2]
for (int i = 0; i < data.length; i++)
    newdata[i] = data[i]
data = newdata
```

### add(int index

If (index < 0) || (index > size) throw Exception

```java
if (index > data.length) resize()
data[index] = element
size++
```

### add(T element)

```java
add(size, element)
return true
```

#### amortized analysis

Over n adds to average out resize:

```java
// amortized running times over n adds to average out resizes
```

### ArrayList Running Times Summary

- int size() -- O(1)
- boolean isEmpty() -- O(1)
- void clear() -- O(1)
- T get(int index) -- O(1)
- T set(int index, T element) -- O(1)
- T remove(int index) -- O(n) (worst case: index=0 & need shift)
- add(int index, T element) -- O(n) (worst case: same as remove)
- add(T element) -- always adding to end... so O(1)...
  unless there is a resize requiring O(n) work.
  but by double resizing, over n adds we only
  do O(1)+O(2)+O(4)+O(16)+...+O(n)=O(n) work.
- Thus, add is
  amortized
  (i.e. average) O(1)

In summary, the arraylist running times are:

#### Stacks

A stack is a data structure that represents a collection of items ordered last in first out (access is restricted to most recently added item).

- Supported Stack Operations:
  - void push( item ) -- add an item (to the top of the stack)
  - item pop() -- remove and return the item on the top of the stack
  - item top() -- return (but don't remove) the item on the top of the stack
  - int size() -- return the number of elements in the stack
  - boolean isEmpty() -- return whether the stack is empty of any items
  - void clear() -- clear all items from the stack

Yes, this looks access to data, but we implement all in O(1) time.

This is an example of an abstract data type (ADT). We've defined both:

- the data type;
- the allowable operations.

The definition is implementation independent.

Some uses for a stack are...
Array Implementation of a Stack

Suppose we implement a stack using an array as the "backing storage":

```
T[] stack     // where to store the items
int top       // the index of the next free spot, also size
```

```c
size()         return queue.length
isEmpty()     return size == 0
```

The running time of all the operations are $O(1)$ (one is amortized $O(1)$)

```
push (item) { stack[items] ++; return;} // Add item (to the end of the stack)
pop() { if (!isEmpty()) return stack[--items]; else return null; } // Remove and return the item at the front of the stack
top() { if (isEmpty()) throw Exception; return stack[items]; } // What to do if stack is empty?
```

Some uses for a queue are...

- `poll` (dequeue) - Dequeue (or first item from the queue)
- `removeIf` (dequeue) - Remove from the queue items that match a condition
- `peek` (dequeue) - Return (but don't remove) the item at the front of the queue
- `removeIf` (dequeue) - Remove and return the item from the front of the queue
- `pull` (dequeue) - Remove and return the item from the front of the queue
- `peek` (dequeue) - Get a view of the item at the front of the queue

A queue is a data structure that represents a collection of items ordered first-in-first-out (FIFO). It is used in various applications, including file systems, operating systems, and more. Queues are particularly useful when you need to process items in the order they arrive, such as in a printer where jobs are processed one after another.
A way of overcoming fixed size of an array is to use a linked list. A linked list is a collection of nodes where each node contains an item and a pointer to the next node.

To implement a queue with these linked nodes, we need access to both the "front" and the "back" of the queue. To implement a queue with these linked nodes, we need access to both the top of the stack and the bottom of the stack.

Node<T>
private T data;
private Node<T> next;

// What is a good constructor?

// And now your stack can look like this:
Node<T> top
int size
void push(T item)
top = new Node<T>(item, this.top)
size++

// Does this work if the stack is empty?

In lab 3 you will figure out the rest of the linked implementation of a queue.

Node<T>
private T data;
private Node<T> next;
Node<T> front // Do these need initial values? If so, what?
Node<T> back
int size

dequeue(T item)
if (isEmpty()) throw exception
T ret = front.getData()
size--
front = front.getNext()
return ret

What is a good constructor?

Node<T> top
void push(T item)
Node<T> node

In this chapter we will face the implementation of a stack and a queue.

A linked implementation of a Stack

In this chapter we will face the implementation of a stack and a queue.

A linked implementation of a Queue