CS 151

ArrayLists, Stacks & Queues
Announcements

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

First, the constructors:

```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)

void 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]
```
ArrayList Running Times

Getting more tricky:

T[] data
int size

T set(int index, T element)
    if( (index < 0) || (index >= size) ) throw Exception
    ret = data[index]
    data[index] = element
    return ret

T remove(int index)
    if( (index < 0) || (index >= size) ) throw Exception
    ret = data[index]
    for(i = index; i < size-1; i++)
        data[i]=data[i+1]
    size--
    return ret
All remaining methods make use of the resize() method:

```java
void resize()
    newdata = new array of size [data.length*2]
    for( int i=0; i < data.length; i++ )
        newdata[i] = data[i]
    data = newdata
    // This is O(data.length) in the worst case

void add(int index, T element)
    if( (index < 0) || (index > size) ) throw Exception
    if( index >= data.length ) resize()
    data[index] = element
    size++

boolean add(T element)
    add(size, element)
    return true
    // Use amortized analysis over n adds to average out resizes
```
In summary, the ArrayList running times are:

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)
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 limits access to data, but we’ll implement all in O(1) time.

This is an example of an abstract data type (ADT). We’ve defined both
• the data type, and
• 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”:

```java
T[] stack     // where to store the items
int top       // the index of the next free spot, also size
size()        // the index of the next free spot, also size
isEmpty()     // the index of the next free spot, also size
clear()       // the index of the next free spot, also size
top()         // what to do if stack is empty?
pop()         // what to do if stack is empty?
push( item )  // what to do if stack is full?
```

The running time of all the operations are $O(1)$ (one is amortized $O(1)$)
A queue is a data structure that represents a collection of items ordered First In First Out (access is restricted to least recently added item.)

Stack Operations:
• `void enqueue(item)` – add an item (to the end of the queue)
• `item dequeue()` – remove and return the item at the front of the queue
• `item front()` – return (but don’t remove) the item at the front of the stack
• `int size()` – return the number of elements in the queue
• `boolean isEmpty()` – return whether the queue is empty of any items
• `void clear()` – clear all items from the queue

Some uses for a queue are ...
Array Implementation of a Queue

Suppose we implement a queue using an array as the “backing storage”:

T[] queue     // where to store the items
int size      // the index of the next open spot, also size

When we dequeue (from the front of the array), should we shift all the elements forward? If so, this is an O(n) operation, possibly every time.

Could avoid the shifting by using two pointers into our array: one for the front, and one for the back:

T[] queue     // where to store the items
int front     // the index of the first element in the queue
int back      // the index of the last element in the queue
int size      // could probably calculate this from front & back
  // but this is just easier to keep around

What happens when back gets to the end? (Double) resize?

But then we could be resizing the array even when there is only one element in it. And we’ll keep doing this, which is a total waste of space.
Circular Array Implementation of a Queue

Fix this by using a circular array, i.e. use modular arithmetic to wrap around

T[] queue    // where to store the items
int front    // the index of the first element in the queue
int back     // the index of the last element in the queue
int size     // number of items in the queue

size()      -- O(1)
isEmpty()   -- O(1)
clear()     -- O(1)
front()     -- O(1)
    if isEmpty() throw Exception
        return queue[front]
dequeue()
    tmp = front()
    front = (front+1) % queue.length, size--
    return tmp
enqueue( item )
    if size == queue.length then doubleResize()
    back = (back+1) % queue.length, size++
    queue[back] = item
A way of overcoming fixed size of an array is to use a linked list: a list of nodes, where each node contains an item and a pointer to the next node.

```java
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?
```
Linked Implementation of a Stack

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

    public Node( T data, Node<T> next )
        this.data = data
        this.next = next
    private T getData() return data
    private Node<T> getNext() return next

Node<T> top
int size

T pop()
    if( isEmpty() ) throw exception
    T ret = top.getData()
    size--
    top = top.getNext()
    return ret
To implement a queue with these linked nodes, we need access to both the “front” and the “back” of the 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

enqueue( item )
    if( isEmpty() )
        back = new Node<T>( item, null )
        front = back
    else
        back.next = new Node<T>(item, null)
        back = back.next
    size++

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