Preconditions for True Fun

- "You must be present and undistracted in order to have fun, because flow is a foundational element of fun, and flow requires total absorption."
- "If you feel judged, either by yourself or by someone else, you will not be able to have fun. Likewise, it's difficult to have fun if you feel like the other people around you aren't also having fun, or if there's a wet blanket or spoilsport present."
- "You (and your companions) must be fully invested in and connected with the activity or people you're with (or both)."
- "When the stakes are too high, the fun runs away."
- "… it is remarkable how consistently members of the Fun Squad mentioned other people when they reminisced about times that were truly fun – even if they were self-described introverts."

-- The Power of Fun: How to Feel Alive Again by Price

CSCI 136: Data Structures and Advanced Programming

Lecture 17

Linear structures, part 2

Instructor: Kelly Shaw

Williams

Topics

- Stack data structure
- Queue ADT
- Queue data structure
- Resubmission procedure

Your to-dos

1. Lab 6 (partner lab), **due Tuesday 11/1 by 10pm**. (two weeks!)
2. Read **before Fri**: Bailey, Ch 8-8.3.

Announcements

- Colloquium: **What I Did Last Summer (Research)**, 2:35pm in Wege Auditorium with **cookies**.
- Practice midterm posted on the course website.
    - **Bring questions** to class on Monday for review!
- TA feedback.

Announcements

Please **consider being a TA** next semester (especially for this class!)

Applications **due Friday, October 28**.

https://csci.williams.edu/tatutor-application/

Stack ADT

Stack ADT

A **stack** is an **abstract data type** that stores a collection of **any type of element**. A stack **restricts which elements are accessible**: elements may only be added and removed from the "**top**" of the collection. The "**push**" operation places an element onto the top of the stack while a "**pop**" operation removes an element from the top.

Stack implementations

StackArray

A **StackArray** is a stack implemented using an **array** for element storage.

Pros: **push** and **pop** are **O(1)** operations.

Cons: data structure has a maximum **capacity**.

Stack implementations

StackVector

A **StackVector** is a stack implemented using a **Vector** for element storage.

Pros: **push** and **pop** are amortized **O(1)** operations. There is no maximum capacity.

Cons: Most of the time, ops take **O(1)** time, but occasionally--when the underlying array needs to grow--an **O(n)** cost is incurred. This may be fine for most applications, but if the application cannot tolerate wide variation in time, this is a bad choice.

Also, unless the underlying array is completely full, Vectors **waste some space**.

Stack implementations

StackList

A **StackList** is a stack implemented using a **List** (usu. **SLL**) for element storage.

Pros: **push** and **pop** are **O(1)** operations. There is no maximum capacity, and no wasted space. **push** and **pop** costs are predictable (always the same), unlike **StackVector**.

Cons: because of the way computer hardware is implemented, a **StackList**'s constant-time cost is likely to be much higher than a **StackVector**'s. So a **StackList**'s performance may be **more predictable** than a **StackVector**, but it will likely be **slower on average**.

Let's look at **StackList**

```java
package structure5;
import java.util.Iterator;

public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    protected List<E> data;

    public StackList() {
        data = new SinglyLinkedList<E>();
    }

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

    public boolean empty() {
        return data.isEmpty();
    }

    public Iterator<E> iterator() {
        return data.iterator();
    }

    public E get() {
        return data.getFirst();
    }

    public void add(E value) {
        data.addFirst(value);
    }

    public E remove() {
        return data.removeFirst();
    }

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

    public String toString() {
        return "<StackList: "+data+">";
    }
}
```

Uses an SLL for storage.


```java
package structure5;
import java.util.Iterator;

public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    protected List<E> data;

    public StackList() {
        data = new SinglyLinkedList<E>();
    }

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

    public boolean empty() {
        return data.isEmpty();
    }

    public Iterator<E> iterator() {
        return data.iterator();
    }

    public E get() {
        return data.getFirst();
    }

    public void add(E value) {
        data.addFirst(value);
    }

    public E remove() {
        return data.removeFirst();
    }

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

    public String toString() {
        return "<StackList: "+data+">";
    }
}
```

Uses an SLL for storage.

Adding an element puts it at the front of the list.


```java
package structure5;
import java.util.Iterator;

public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    protected List<E> data;

    public StackList() {
        data = new SinglyLinkedList<E>();
    }

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

    public boolean empty() {
        return data.isEmpty();
    }

    public Iterator<E> iterator() {
        return data.iterator();
    }

    public E get() {
        return data.getFirst();
    }

    public void add(E value) {
        data.addFirst(value);
    }

    public E remove() {
        return data.removeFirst();
    }

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

    public String toString() {
        return "<StackList: "+data+">";
    }
}
```

Uses an SLL for storage.

Adding an element puts it at the front of the list.

Wait! What about push?

push just calls add.

```java
package structure5;

public abstract class AbstractStack<E> extends AbstractLinear<E> implements Stack<E>
{
    public void push(E item)
    {
        add(item);
    }

    public E pop()
    {
        return remove();
    }

    @Deprecated public E getFirst()
    {
        return get();
    }

    public E peek()
    {
        return get();
    }
}
```

push just calls add.

```java
package structure5;

public abstract class AbstractStack<E> extends AbstractLinear<E> implements Stack<E>
{
    public void push(E item)
    {
        add(item);
    }

    public E pop()
    {
        return remove();
    }

    @Deprecated public E getFirst()
    {
        return get();
    }

    public E peek()
    {
        return get();
    }
}
```

pop just calls remove.

Uses an SLL for storage.

Removing an element removes the first element in the list.

```java
package structure5;
import java.util.Iterator;

public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    protected List<E> data;

    public StackList() {
        data = new SinglyLinkedList<E>();
    }
    
    public void clear() {
        data.clear();
    }

    public boolean empty() {
        return data.isEmpty();
    }

    public Iterator<E> iterator() {
        return data.iterator();
    }

    public E get() {
        return data.getFirst();
    }

    public void add(E value) {
        data.addFirst(value);
    }

    public E remove() {
        return data.removeFirst();
    }

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

    public String toString() {
        return "<StackList: "+data+">";
    }
}
```

Adding an element puts it at the front of the list.

Queue ADT

A **queue** is an **abstract data type** that stores a collection of **any type of element**. A queue **restricts which elements are accessible**: elements may only be added to the "**end**" of the collection and elements may only be removed from the "**front**" of a collection. The "**enqueue**" operation places an element at the end of a queue while a "**dequeue**" operation removes an element from the front.

Queue ADT

Queue ADT

Also sometimes referred to as a **FIFO**: **"first in, first out."**

(a stack would be an annoying way to process a line at Starbucks!)

Frequently used as a **buffer** to hold work **to do later**.

We also frequently include a "**peek**" operation that lets us look at an element on the top of a queue without removing it, and "**size**" and "**isEmpty**" operations that let us check how many elements are stored and whether a queue stores zero elements, respectively.

Queue implementations

QueueArray

A **QueueArray** is a queue implemented using an **array** for element storage.

Pros: **enqueue** and **dequeue** are **O(1)** operations.

Cons: data structure has a maximum **capacity**.

Queue implementations

QueueVector

A **QueueVector** is a queue implemented using a **Vector** for element storage.

Pros: **enqueue** and **dequeue** are amortized **O(1)** operations. There is no maximum capacity.

Cons: Most of the time, they take **O(1)** time, but occasionally--when the underlying array needs to grow--an **O(n)** cost is incurred. This may be fine for most applications, but if the application cannot tolerate wide variation in time, this is a bad choice. Also, unless the underlying array is completely full, **Vectors waste some space**.

Queue implementations

QueueList

A **QueueList** is a queue implemented using a **List** (usu. **DLL** or **CL**) for element storage.

Pros: enqueue and dequeue are **O(1)** operations. There is no maximum capacity. **enqueue** and **dequeue** costs are predictable (always the same), unlike **QueueVector**.

Cons: because of the way computer hardware is implemented, a **QueueList**'s constant-time cost is likely to be much higher than a **QueueVector**'s. So a **QueueList**'s performance may be more predictable than a **QueueVector**, but it will likely be slower on average.

Other queue-like ADTs

One very useful and interesting variant of the Queue ADT is the **Priority Queue ADT**. We'll talk about priority queues after the midterm!

Resubmission procedure

Resubmission procedure

Remember: the goal of this course is mastery.

Resubmission procedure

Allows you to earn **up to 50% of the lost points**.

E.g., **if you got a 50%** on the midterm, **you can get a 75%** on resubmission.

Midterm is 25% of your final grade. **This is worth doing!**

Resubmission procedure

1. You have **until the end of reading period**.
2. Resubmission **must include both** the **original work** and the **new submission**.
3. Must be accompanied by an **explanation document**, written in plain English.

Resubmission procedure

Explanation document **must identify**:

1. **What** the mistake is.
2. **How** you fixed the mistake.
3. **Why** the new version is correct.

Resubmission procedure

Resubmit code **electronically** (i.e., using git).

Resubmit exam **on paper** (i.e., hand it to me or put in mailbox).

Resubmission procedure

Sample from CS334:

2. Troubleshooting

My fix was slightly wrong. Righr before calling $random\_string()$, I added

```
char * arrarr[i] = malloc(sizeof(char)*MAXLEN);
```

when what I should have added is

```
arrarr[i] = malloc(sizeof(char)*MAXLEN);
mcheck(arrarr[i]);
```

There is no need for "char *" because I am not declaring $arrarr$.

I got my explanation and drawing wrong. In my drawing, I had $arrarr[i]$ pointing back to a call stack because I thought the program would automatically allocate memory on a call stack if we did not $malloc()$. What I should have said is that without allocating sub-array $arrarr[i]$, the address currently living in the sub-array is arbitrary so the value referred to by the sub array is also arbitrary. When we call $memset()$ or manipulating $arrarr[i]$ in $random\_string()$, we are likely to get memory errors. Below is what I should have drawn.