Lecture 14

Stacks

- Linear Data Structures
- Stacks
- Midterm Review
  - Problem 1
  - Problem 2

Linear Data Structures

Linear Data Structures

In a `Vector` it is possible to `add` and `remove` entries based on an index. Similarly, we can do the same thing in any array or linked list. In other words, if there are n entries, then we can add a new entry in any one of $n+1$ indices, and remove an entry from any one of n indices.

In some cases, it makes sense to have only one index where an entry can be added or removed. In other words, the `add` and `remove` methods do not have an index argument.

Although terminology isn't always consistent in computer science, we often refer to this type of data structure as a *linear data structure*.

```
GNU nano 5.8                                    Linear.java
// An interface for LIFO/FIFO structures.
// (c) 1998,2001 duane a. bailey

package structure5;

// An interface describing the behavior of linear data structures, structures that
// that have completely determined  add and remove methods.
// Linear structures are often used to store the the state of a recursively
// solved problem and stacks and queues are classic examples of such structures.
// The structure package provides several implementations of the Linear interface,
// each of which has its particular strengths and weaknesses.
public interface Linear<E> extends Structure<E>
{
    // Add a value to the structure.  The type of structure determines
    // the location of the value added.
    public void add(E value);

    // Preview the object to be removed.
    public E get();

    // Remove a value from the structure.  The particular value
    // to be removed is determined by the structure.
    public E remove();

    // Returns the number of elements in the linear structure.
    public int size();

    // Returns true iff the structure is empty.
    public boolean empty();
}
```

`Linear` interface in `Linear.java`.

Adding and Removing in Different Contexts

Consider a line of people waiting.

- Where would a new element (i.e. person) be added?
- Where would an element be removed?

Consider plates on top of each other.

- Where would a new element (i.e. plate) be added?
- Where would an element be removed?

In these situations, it makes sense to restrict how entries are added and removed.

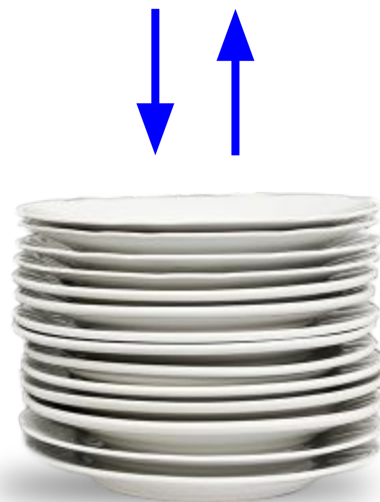
Queues and Stacks

We will consider two different data structures called “queues” and “stacks”.

- These two terms are consistent across computer science.

We’ll guide our discussion by looking at implementations of these data structures in the `structures` package.

Queue



Stack

Stacks

Stack

In a *stack*, the only element that can be accessed is the most recently added element.
We refer to it using the acronym *LIFO* for *Last-In First-Out*.

Which plate is most easily accessible?

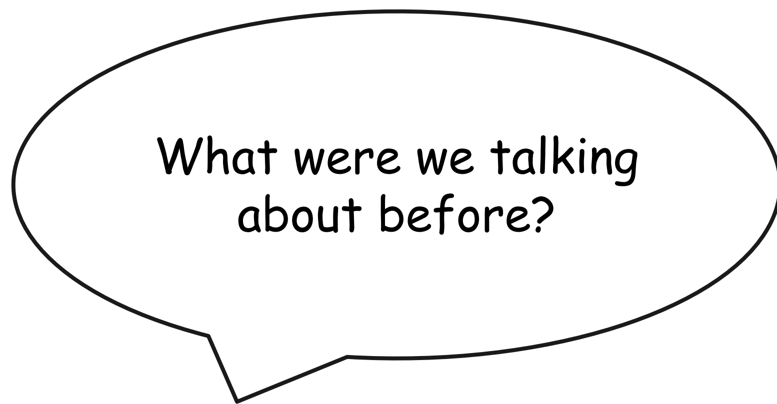
Adding to the top of the stack is called *pushing*. More generically, `Linear` has `add`.
Removing the top of the stack is called *popping*. More generically, `Linear` has `remove`.
Viewing the top of the stack is sometimes called *peeking*. More generically, `Linear` has `get`.

Metaphors for Stacks

Consider these instances of stacking.

What happens during push and pop?

Stack of dishes.



Conversation stack.

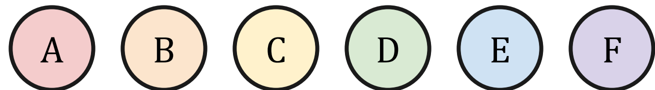
How is a conversation stack similar to a recursive call stack?

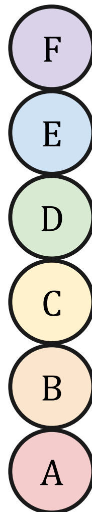
Reversing a Sequence

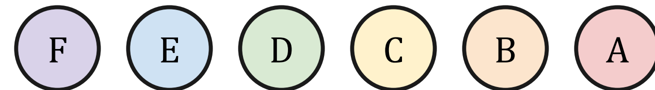
Reversing a Sequence

Suppose that we have a sequence of values, and we want to reverse the sequence.

How can we do this with a stack?







Sequence A, B, C, D, E, F

Push the values onto a stack

The order obtained by popping

1. Push every value in the sequence onto the stack.
2. Pop every value in the stack off.

Implementation in `structure`

```
 GNU nano 5.8                                    Stack.java
package structure5;

// An interface describing a Last-In, First-Out structure.
// Stacks are typically used to store the state of a recursively
// solved problem.
// The structure package provides several implementations of the Stack interface,
// each of which has its particular strengths and weaknesses.
public interface Stack<E> extends Linear<E>
{
    // Add an element from the top of the stack.
    public void add(E item);

    // Add an element to top of stack.
    public void push(E item);

    // Remove an element from the top of the stack.
    public E remove();

    // Remove an element from the top of the stack.
    public E pop();

    // Fetch a reference to the top element of the stack.
    public E get();

    // Fetch a reference to the top element of the stack.
    public E getFirst();

    // Fetch a reference to the top element of the stack.
    public E peek();

    // Returns true iff the stack is empty.  Provided for
    // compatibility with java.util.Vector.empty.
    public boolean empty();

    // Returns the number of elements in the stack.
    public int size();
}
```

`Stack` interface in `Stack.java`.

Discussion: Implementing `Stack`

We could implement the `Stack` interface in a number of different ways.

- How would you implement `Stack` using an array `A`?
- How would you implement `Stack` using a linked list?
- How would you implement `Stack` using a `Vector`?

Discuss these questions with your neighbor for 5 minutes.

Consider the following important details.

- Array implementation: Does `A[0]` contain the top or bottom of the stack? Why? Does your array resize itself? Why?
- Linked list implementation: Does `head` give the top or the bottom of the stack? Why? Would you use a singly linked or doubly linked list? Why?
- Vector implementation: Is there an advantage to using a `Vector` instead of an array?

```
// An implementation of a stack, based on array.  The head of the
// stack is stored in the first position of the array, allowing the stack to grow
// and shrink in constant time. This stack implementation is ideal for
// applications that require a stack with a known maximum size that expands
// in constant time.
public class StackArray<E> extends AbstractStack<E> implements Stack<E>
{
    // An index to the top element of the stack.
    protected int top;

    // The array of value references.  Top of the stack is higher in array.
    protected Object data[];

    // Construct a stack capable of holding at least size elements.
    public StackArray(int size) {
        data = new Object[size];
        clear();
    }

    // Remove all elements from the stack.
    public void clear() {
        top = -1;
    }

    // Add a value to the top of the stack.
    public void add(E item) {
        Assert.pre(!isFull(),"Stack is not full.");
        top++;
        data[top] = item;
    }

    // Remove a value from the top of the stack.
    public E remove() {
        Assert.pre(!isEmpty(),"Stack is not empty.");
        E result = (E)data[top];
        data[top] = null;
        top--;
        return result;
```

Implementing the `Stack` interface with an array in `StackArray.java`.

```java
// An implementation of a stack, based on extensible arrays.  The head of the
// stack is stored in the first position of the list, allowing the stack to grow
// and shrink in constant time. This stack implementation is ideal for
// applications that require a dynamically resizable stack which occasionally takes
// a time proportional to the its length to expand.
public class StackVector<E> extends AbstractStack<E> implements Stack<E>
{
    // The vector containing the stack data.
    protected Vector<E> data;

    // Construct an empty stack.
    public StackVector() {
        data = new Vector<E>();
    }

    // Construct a stack with initial capacity
    // Vector will grow if the stack fills vector.
    public StackVector(int size) {
        data = new Vector<E>(size);
    }

    // Add an element from the top of the stack.
    public void add(E item) {
        data.add(item);
    }

    // Remove an element from the top of the stack.
    public E remove() {
        return data.remove(size()-1);
    }

    // Fetch a reference to the top element of the stack.
    public E get() {
        // raise an exception if stack is already empty
        return data.get(size()-1);
    }
```

Implementing the `Stack` interface with a `Vector` in `StackVector.java`.

```java
// An implementation of a stack, based on lists.  The head of the
// stack is stored at the head of the list, allowing the stack to grow
// and shrink in constant time. This stack implementation is ideal for
// applications that require a dynamically resizable stack that expands
// in constant time.
public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    // The list that maintains the stack data.
    protected List<E> data;

    // Construct an empty stack.
    public StackList() {
        // Think about why we use singly linked lists here:
        // They're simple, and take less space.
        data = new SinglyLinkedList<E>();
    }

    // Remove all elements from the stack.
    public void clear() {
        data.clear();
    }

    // Determine if the stack is empty.
    // Provided for compatibility with java.util.Stack.empty.
    public boolean empty() {
        return data.isEmpty();
    }

    // Get a reference to the top value in the stack.
    public E get() {
        return data.getFirst();
    }

    // Add a value to the top of the stack.
    public void add(E value) {
        data.addFirst(value);
    }
```

Implementing the `Stack` interface with a linked list in `StackList.java`.

Midterm Review

Problem 1 [15 points]

Please answer the following with True or False. Justify each answer with one sentence.

a. Every type in Java is an object.

b. All objects have an `equals` method that determines if two objects have the same value.

c. Strings are sometimes mutable, sometimes not.

d. The choice to extend a `Vector` by doubling the size of the underlying array improves the performance of its `contains` method.

e. All recursive procedures have a *base case*.

Problem 2 [20 points]
String pools. When a program makes use of a large number of strings, many of which have the same value, it can be useful to use a *string pool*. A string pool is a data structure that contains *canonical* versions of every string value seen so far. When a new string is encountered, it is *interned* into the string pool using the following process:

1. The string pool is searched for a string with the same value.
2. If the string was not found, the new string is added to the pool.
3. The reference returned is the reference to the string value in the pool.

For example, in a database describing pets, the string `"labrador retriever"` might appear multiple times. We can reduce the storage required by sharing many references to a single string, `"labrador retriever"`.

There are two important observations to be made here. First, using a string pool will replace a program's duplicate string values with their canonical forms. Second, if all strings are interned then the *references of two interned strings are the same if and only if the two strings have the same value*.

In this problem, we'll write a class that describes a string pool.

a. Declare the instance variable(s) that would be necessary to keep track of (the canonical versions of) an arbitrarily large number of `String` objects:

```
public class StringPool {

}
```

b. Write a constructor that initializes your `StringPool` object:

```
public StringPool()
{
```

c. Write a `StringPool` method that interns a String, returning the canonical form of the string, `s`:

```
public String intern(String s) {

}
```

d. Describe the complexity—using "big-O" notation—of the `intern(String s)` method for your implementation. Explain your reasoning.

e. Write a static method, `same(a,b)`, that returns true exactly when two interned strings, `a` and `b`, are equal to each other.

```
public static boolean same(String a, String b)
// pre: a and b are interned Strings
```