csci 210: Data Structures

Stacks and Queues

Summary

- Topics
  - stacks and queues as abstract data types
  - implementations
    - arrays
    - linked lists
  - analysis and comparison
  - application: searching with stacks and queues
    - Problem: missionary and cannibals
    - Problem: finding way out of a maze
    - depth-first and breadth-first search

- READING:
  - GT textbook chapter 5

Stacks and Queues

- Fundamental “abstract” data types
    - abstract, i.e. we think of their interface and functionality; the implementation may vary

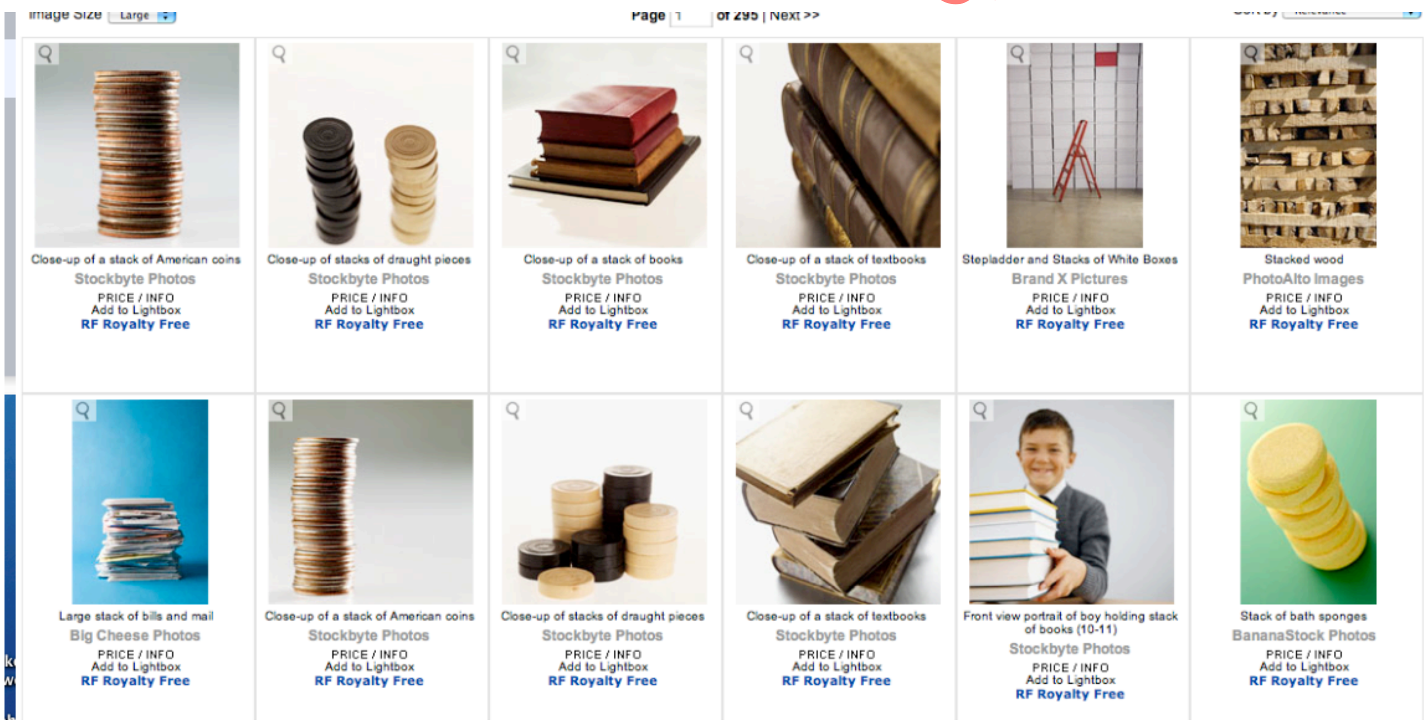
- Interface:
    - stacks and queues handle a collection of elements
    - operations:

        - insert(e)
        - remove()
        - isEmpty()
        - getSize()

- Stacks
    - only last element can be deleted
    - ==>insert and delete at one end
    - last-in-first-out (LIFO)

Queues
- only first element can be deleted
- ==> insert at one end, delete from the other end
- First-in-first-out (FIFO)

Stack analogy



Stack interface

- push(e)
  - insert element e
- pop()
  - delete and return the last inserted element
- size()
  - return the number of elements in the queue
- isEmpty()
  - return true if queue is empty

Queue Analogy

Queue interface

- enqueue(e)
  - insert element e
- dequeue()
  - delete and return the first inserted element
- size()
  - return the number of elements in the queue
- isEmpty()
  - return true if queue is empty

Applications

- Are stacks and queues useful?
  - YES. They come up in many problems.

- Stacks
  - Internet Web browsers store the addresses of recently visited sites on a stack. Each time the visits a new site ==> pushed on the stack. Browsers allow to “pop” back to previously visited site.
  - The undo-mechanism in an editor. The changes are kept in a stack. When the user presses “undo” the stack of changes is popped.
  - The function-call mechanism
    - the active (called but not completed) functions are kept on a stack
    - each time a function is called, a new frame describing its context is pushed onto the stack
    - when the function returns, its frame is popped, and the context is reset to the previous method (now on top of the stack)
- Queues
  - queue of processes waiting to be processed; for e.g. the queue of processes to be scheduled on the CPU
  - round-robin scheduling: iterate through a set of processes in a circular manner and service each element:
    - e.g. the process at front is dequeued, allowed to run for some CPU cycles, and then enqueued at the end of the queue

Using Stacks

- java.util.Stack

Constructor Summary
Stack()   Creates an empty Stack.

Method Summary	
boolean	**empty**() Tests if this stack is empty.
Object	**peek**() Looks at the object at the top of this stack without removing it from the stack.
Object	**pop**() Removes the object at the top of this stack and returns that object as the value of this function.
Object	**push**(Object item) Pushes an item onto the top of this stack.
int	**search**(Object o) Returns the 1-based position where an object is on this stack.

Using Stacks

```
Stack<Integer>  st = new Stack<Integer>();

s.push (3) ;

s.push (5) ;

s.push (2);

//print the top
System.out.print(s.peek());

s.pop();

s.pop();

s.pop();
```

A Stack Interface

- stack can contain elements of arbitrary type E
- use *generics:* define Stack in terms of a generic element type E

```
Stack<E> {

}...
```

- when instantiating Stack, specify E

```
Stack<String> st;
```

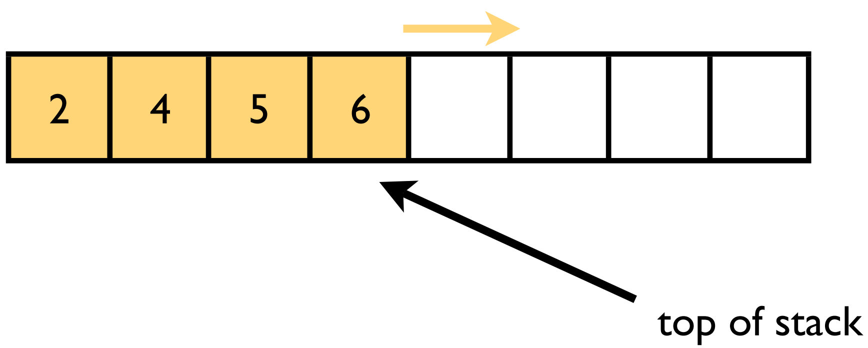
- Note: could use Object, but then need to cast every pop()

```java
 /**
  * Interface for a stack: a collection of objects that are inserted
  * and removed according to the last-in first-out principle.  This
  * interface includes the main methods of java.util.Stack.
  */
public interface Stack<E> {
 /**
  * Return the number of elements in the stack.
  * @return number of elements in the stack.
  */
  public int size();
 /**
  * Return whether the stack is empty.
  * @return true if the stack is empty, false otherwise.
  */
  public boolean isEmpty();
 /**
  * Inspect the element at the top of the stack.
  * @return top element in the stack.
  * @exception EmptyStackException if the stack is empty.
  */
  public E top()
    throws EmptyStackException;
 /**
  * Insert an element at the top of the stack.
  * @param element to be inserted.
  */
  public void push (E element);
 /**
  * Remove the top element from the stack.
  * @return element removed.
  * @exception EmptyStackException if the stack is empty.
  */
  public E pop()
    throws EmptyStackException;
}
```

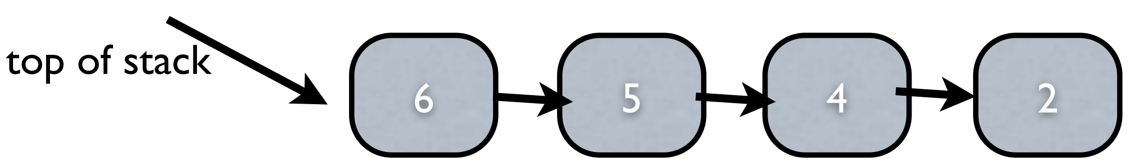
Stack Implementations

- Stacks can be implemented efficiently with both
  - arrays
  - linked lists

- Array implementation of a Stack



- Linked-list implementation of a stack
  - a linked list provides fast inserts and deletes at head
    - ==> keep top of stack at front



Next..

- Provide sketch for each implementation
- Analyze efficiency
- Compare

Arrays vs Linked-List Implementations

- Array
  - simple and efficient
  - assume a fixed capacity for array
    - if CAP is too small, can reallocate, but expensive
    - if CAP is too large, space waste
- Lists
  - no size limitation
  - extra space per element
- Summary:
  - when know the max. number of element, use arrays

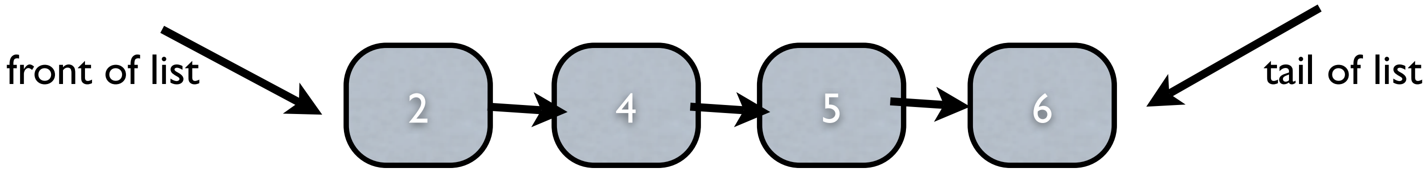
Method	**Time**
size()	O(1)
isEmpty()	O(1)
top	O(1)
push	O(1)
pop	O(1)

A Queue Interface

```
public interface Queue<E> {
 /**
  * Returns the number of elements in the queue.
  * @return number of elements in the queue.
  */
  public int size();
 /**
  * Returns whether the queue is empty.
  * @return true if the queue is empty, false otherwise.
  */
  public boolean isEmpty();
 /**
  * Inspects the element at the front of the queue.
  * @return element at the front of the queue.
  * @exception EmptyQueueException if the queue is empty.
  */
  public E front() throws EmptyQueueException;
 /**
  * Inserts an element at the rear of the queue.
  * @param element new element to be inserted.
  */
  public void enqueue (E element);
 /**
  * Removes the element at the front of the queue.
  * @return element removed.
  * @exception EmptyQueueException if the queue is empty.
  */
  public E dequeue() throws EmptyQueueException;
}
```

Queue Implementations

- Queue with arrays
  - say we insert at front and delete at end
  - need to shift elements on inserts ==> insert not O(1)
- Queue with linked-list
  - in a singly linked-list can delete at front and insert at end in O(1)



- Exercise: sketch implementation

Method	Time
size()	O(1)
isEmpty()	O(1)
front	O(1)
enqueue	O(1)
dequeue	O(1)

Queue with a Circular Array

- A queue can be implemented efficiently with a circular array if we know the maximum number of elements in the queue at any time