csci 210: Data Structures

Stacks and Queues

Summary

- Topics
  - Stacks and Queues as abstract data types (ADT)
  - Implementations
    - arrays
    - linked lists
  - Analysis and comparison
  - Applications: searching with stacks and queues
    - In-class problem: missionary and cannibals
    - In-class problem: finding way out of a maze
  - Searching a solution space: Depth-first and breadth-first search (DFS, BFS)

Stacks and Queues

- Fundamental "abstract" data types
  - we think of them conceptually in terms of their interface and functionality
  - we use them as building blocks in problems without pinning down an implementation (the implementation may vary)

- Interface:
  - Stacks and Queues handle a collection of elements
  - Operations:
    - insert(e)
    - remove()
    - isEmpty()
    - getSize()

In what order?

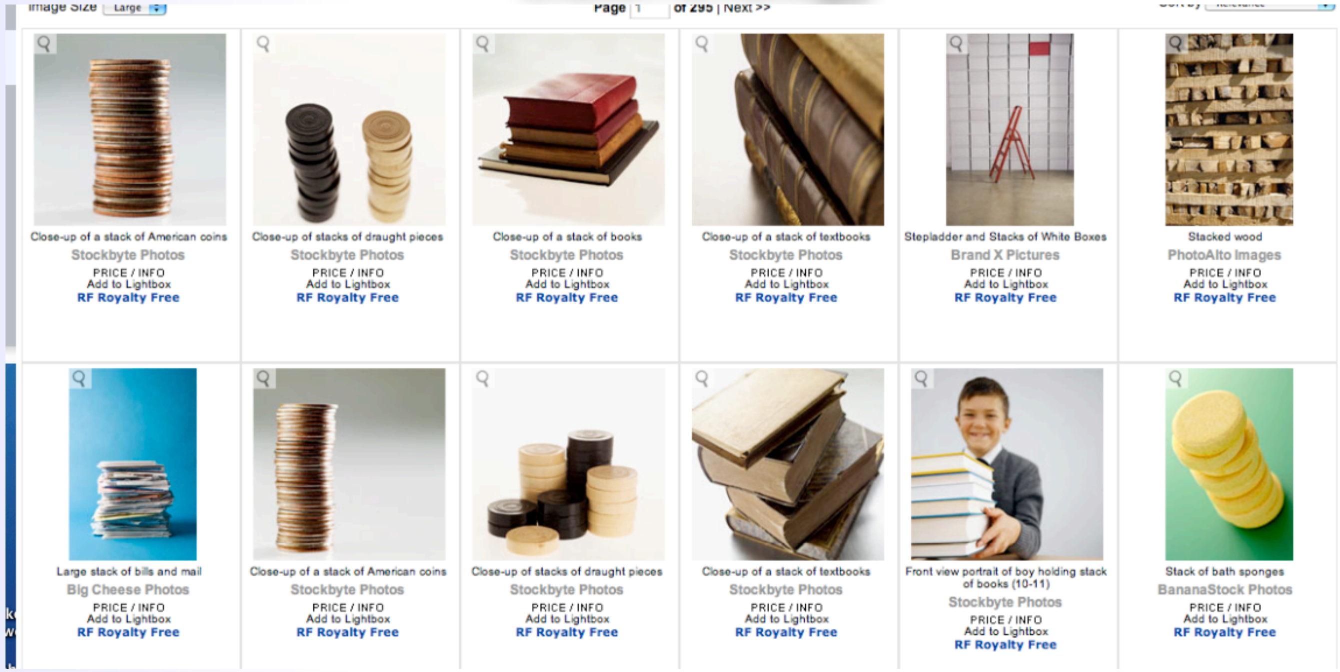
Stacks

- 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 at other
- first-in-first-out (FIFO)

Stack analogy



Stack interface

- push(e) : insert element e (at top of stack)
- pop() : delete and return the top of stack (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 (at end of queue)
- dequeue(): delete and return the front of queue (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 all the time.

- Stacks
  - 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
    - the context of a method: its parameters, local variables, what needs to be returned, and where to return (the instruction to be executed upon return)
    - when the function returns, its frame is popped, the context is reset to the previous method (now on top of the stack) and teh program continues by executing the previously suspended method

Applications

- Are stacks and queues useful?
  - YES. They come up all the time.

- Queues
  - Queue of processes waiting to be processed
    - for e.g. the queue of processes to be scheduled on the CPU.
    - the process at front is dequeued and processed. New processes are added at the end of the queue.

  - Round-robin scheduling: iterate through a set of processes in a circular manner and service each element:
    - 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

```
import java.util.Stack;

//a stack of integers
Stack<Integer>  st = new Stack<Integer>();
st.push (Integer(3)) ;
st.push (Integer(5)) ;
st.push (Integer(2));
//print the top
System.out.print(st.peek());
st.pop();
st.pop();
st.pop();


//a stack of Strings
Stack<String> st = new Stack<String>();
...
```

Using Stacks

```
import java.util.Stack;

//a stack of integers
Stack<Integer> st = new Stack<Integer>();
st.push (Integer(3)) ;
st.push (Integer(5)) ;
st.push (Integer(2));
//print the top
System.out.print(st.peek());
st.pop();
st.pop();
st.pop();


//a stack of Strings
Stack<String> st = new Stack<String>();
...
```

generic type

class Stack uses *generics*

Stacks

- a 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()

Implementing a Stack

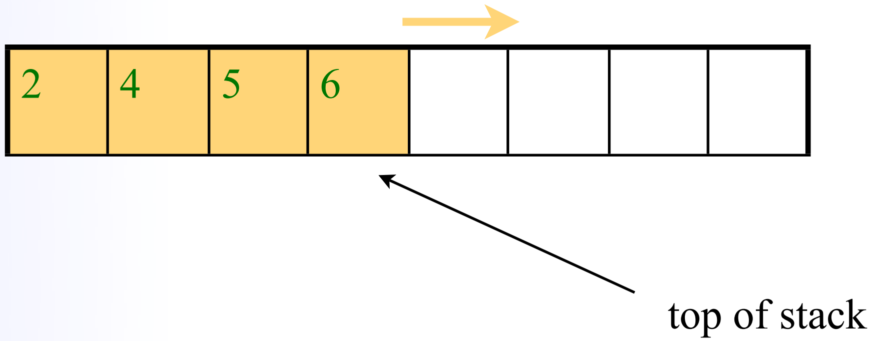
- A Stack interface
- Implementing a Stack with arrays
- Implementing a Stack with linked lists
- Analysis, comparison

```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;
}
```

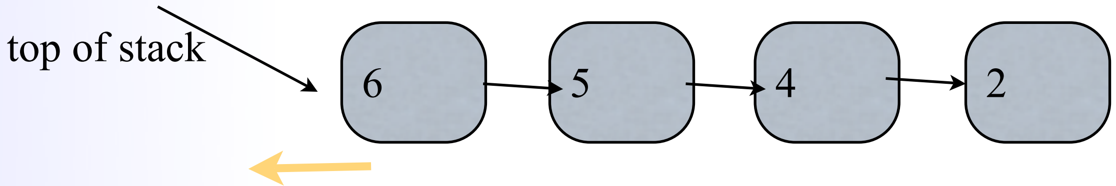
Implementing a Stack

- 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



Implementing Stacks

- Exercise: Sketch each implementation

```
public class StackWithArray<E> implements Stack {

 ....

}
```

- Efficiency ?
- Compare ?

Stack: 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)

Implementing a Queue

- A Queue interface
- Implementing a Queue with arrays
- Implementing a Queue with linked lists
- Analysis, comparison

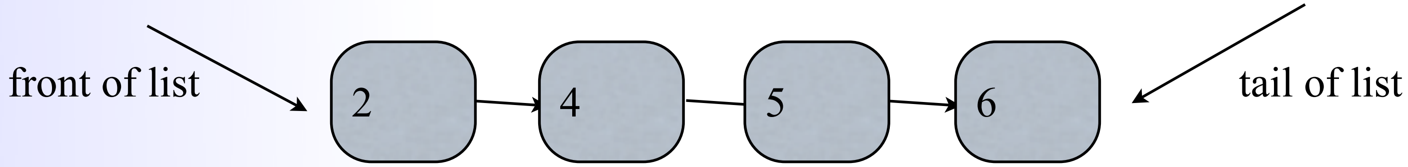
A Queue Interface

```java
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)

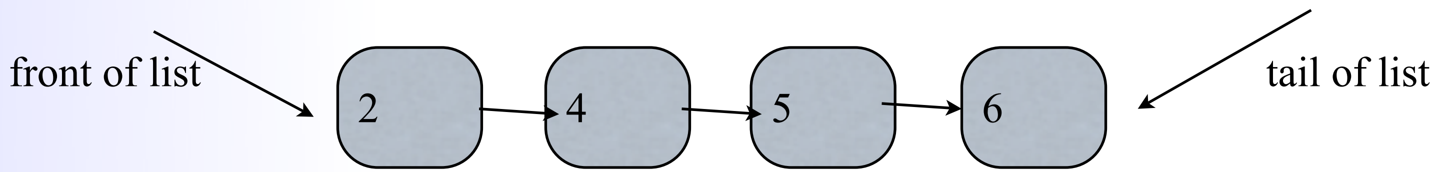


  - Exercise: sketch implementations
  - Analysis?

Queue Implementations

- Queue with arrays
  - need to shift elements on inserts ==> insert not O(1)

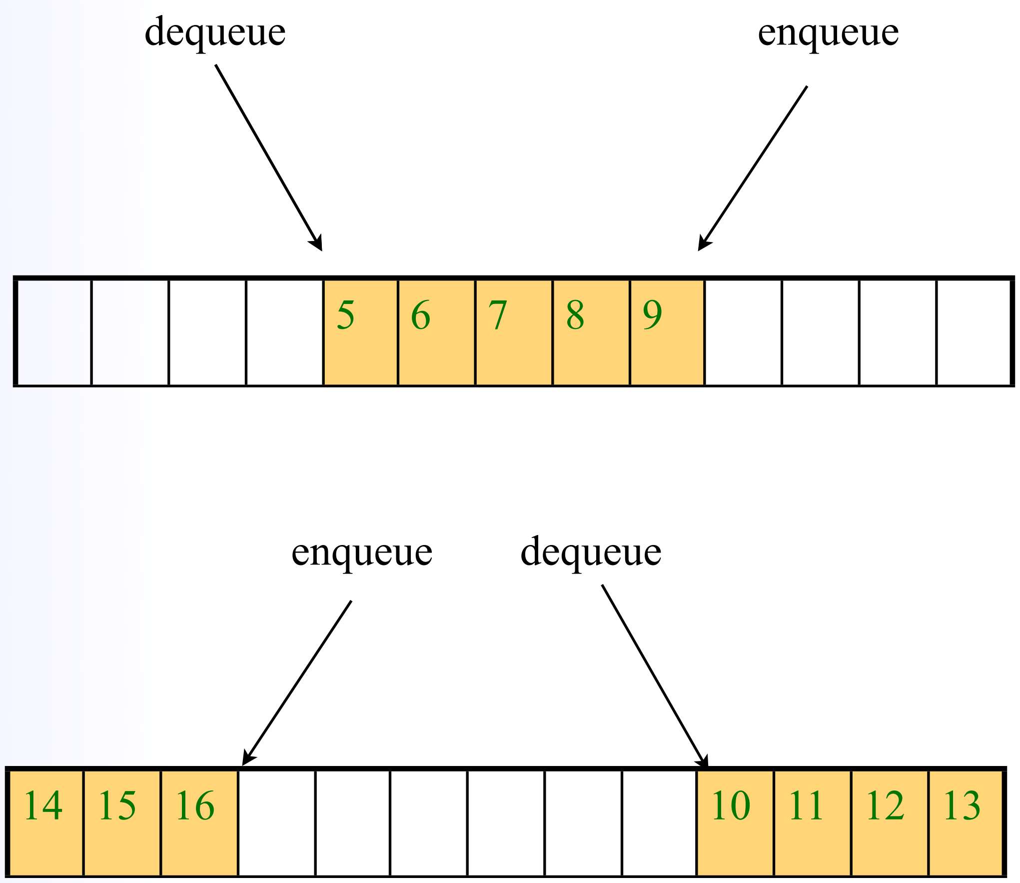
- Queue with linked-list



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



- Exercise: sketch implementation