Reminder: TA Hour Policies

- If you hit a bug, you must be able to show significant effort of fixing it on your own before seeing a TA
  - If not, TAs will turn you away
- TAs will only help you with the question entered in SignMeUp
  - The question should be specific (e.g., doodlejump platforms is bad)
- To be fair to other students in line, each TA hours visit is capped at 15 minutes
- If you do not show up within 5 minutes of your name being called, you will lose your spot in line
  - You can remove yourself from the line by logging into SignMeUp and clicking "remove"
  - Doing so is in your best interest because if you don't, you will get turned away and have to wait an hour to sign up again (as per the SignMeUp policy)

Reminder: CS15 Collaboration Policy

- We have caught more collaboration policy violations
  - MOSS is excellent at detecting it
- If you are ever in doubt about what is or isn't allowed, ask a TA
- Remember: labs are collaborative (while you are in your lab section), no other form of assignment is

Abstract Data Types (1/2)

- To use a method, need to know its essentials: signature and return type
  - Additionally, documentation tells us purpose, error conditions, what resources (such as classes and packages) the method needs, etc.
  - Set of signatures and return types for an entire class designed to store and manage data is called an Abstract Data Type (ADT) – in Java, ADT’s are supported by the interface feature of Java
  - We don’t know anything about its implementation – encapsulation

1. This is an informal definition. ADT also has a mathematical definition.
Abstract Data Types (2/2)

- In these linked list lectures, show both how to implement various list ADTs using linked nodes, and then use those ADTs with simple programs to demonstrate their use.
  - ArrayLists and NodeList are ADTs that adhere to Java's list interface
- Note: full description of an ADT is sometimes called an API (Application Program Interface)
  - term "Application Program Interface" coined by former undergraduate Ira Cotton in 1968

Stacks

- Stack has special requirements for insertion and deletion called push and pop
- Instead of being able to insert and delete nodes from anywhere in the list, can only add and delete nodes from top of Stack
  - LIFO (Last In, First Out)
- We'll implement a stack with a linked list and then use it in a simple demo app

Stack Constructor

- When generic Stack is instantiated, it contains an empty MyLinkedList

  ```java
  public class Stack<Type> {
      private MyLinkedList<Type> _list;
      public Stack() {
          _list = new MyLinkedList<Type>();
      }
  } /* other methods elided */
  ```

Methods of a Stack

- Add element to top of stack
- Remove element from top of stack
- Returns whether stack has elements
- Returns number of elements in stack

  ```java
  public void push(Type el) { //elided }
  public Type pop() { //elided }
  public boolean isEmpty() { //elided }
  public int size() { //elided }
  ```
Pushing an Object

// in the Stack<Type> class ...
public Node<Type> push(Type newData) {
    return _list.addFirst(newData);
}

• When an element is pushed, it is always added to front of list
• Let's see what this does...

Popping an Object

// in the Stack<Type> class ...
public Type pop() {
    return _list.removeFirst();
}

• When popping an element, element is always removed from top of Stack, so call removeFirst on MyLinkedList
• removeFirst returns element removed, and Stack in turn returns it
• Remember that removeFirst method of MyLinkedList first checks to see if list is empty
• Let's see what this does...

isEmpty

• Stack will be empty if _list is empty
• Returns a boolean that is true if Stack is empty and false otherwise

size

• Size of Stack will be the size of the Linked List that it contains
• Size is updated whenever a Node is added to or deleted from _list during push and pop methods
First Example: Execution Stacks

- Each method has an Activation Record (AR)
  - contains an execution pointer to instruction to be executed next in method
  - also contains all local variables and parameters of method
- When methods execute and call other methods, Java uses a Stack to track these calls
  - when a method calls another method, Java adds activation record of called method to Stack
  - when new method is finished, its AR is removed from Stack, and previous method is continued
  - method could be different or a recursively called done

Execution Stacks

Stack Trace

- When an exception is thrown in a program, get a long list of methods and line numbers known as a stack trace
  - Exception in thread "main" <exception name> at <class>.<method>(<class>.java:<line >) ...
- A stack trace prints out all methods currently on execution stack
- If exception is thrown during execution of recursive method, prints all calls to recursive method
Bootstrapping ADT’s

- In effect, this stack ADT is implemented as a thin wrapper over a Linked List ADT, but user has no knowledge of that
- Could also implement it with an array or ArrayList, but that would be more inefficient due to constant data movement
- We’ll use the same technique to implement a Queue

What are Queues?

- Similar to stacks, but elements are removed in different order
  - information retrieved in the same order it was stored
  - FIFO: First In, First Out (as opposed to stacks, which are LIFO: Last In, First Out)
- Examples:
  - standing on line at the checkout counter or movie theater
  - waitlist for TA hours

Enqueuing and Dequeuing

- Enqueuing: adds a node
- Dequeuing: removes a node

Enqueuing and Dequeuing

- Enqueuing: adds a node
- Dequeuing: removes a node
Enqueuing and Dequeuing

- Enqueuing: adds a node
- Dequeuing: removes a node

Before Dequeuing

1 2 3 head of queue
6 7 8 tail of queue

After Dequeuing

2 3 4 head of queue
5 6 7 tail of queue

Our Queue

- Let's use another Linked List to help us make our Queue
- Contain a MyLinkedList within Queue class
  - enqueue will add to the end of MyLinkedList
  - dequeue will remove the first element in MyLinkedList

```java
public class Queue<Type> {
    private MyLinkedList<Type> _list;
    public Queue() {
        _list = new MyLinkedList<Type>();
    }
    // Other methods elided
}
```

enqueue

- Just call _list's addLast method!
- This will add node to end of _list

```java
public void enqueue(Type newNode) {
    _list.addLast(newNode);
}
```
**dequeue**

- We want first node in `_list`
- Use `_list`'s `removeFirst` method!

```java
public Type dequeue() {
    return _list.removeFirst();
}
```

- What if `_list` is empty? There will be nothing to dequeue!
- Our `MyLinkedList` class's `removeFirst()` method returns `null` in this case, so `dequeue` does as well.

---

**isEmpty() and size()**

- As with Stacks, very simple methods; just delegate to `MyLinkedList`

```java
public int size() {
    return _list.size();
}

public boolean isEmpty() {
    return _list.isEmpty();
}
```

---

**Exercise 1 (1/5)**

- How can we use a Stack to reverse a Linked List?

- Linked List: Troy, Gabriella, Ryan, Sharpay

---

**Exercise 1 (2/5)**

- Solution:
  - while Linked List is not empty, remove from Linked List and push elements onto Stack
  - then, while Stack is not empty, pop elements from Stack and add to Linked List
Exercise 1 (3/5)

while(!_list.isEmpty()) {
    stack.push(_list.removeFirst());
}

Exercise 1 (4/5)

while(!stack.isEmpty()){
    _list.addLast(stack.pop());
}

Exercise 2 (1/2)

- Check for balanced parentheses in a given string
- Balanced: [()]{}[]
- Not balanced: []

Exercise 2 (2/2)

- Go through every character, if it is a starting bracket, push it onto the stack
- If it is a closing bracket, pop from the stack
- The bracket you pop should be the opening bracket that corresponds to the closing bracket you are looking at
  - if it is not, return false
- If you get through every character and you haven't returned false, check if stack is empty
- If it is, the brackets are balanced!
Problem 2 Pseudocode

for each bracket in string:
    if it is a starting bracket:
        push it onto stack
    if it is a closing bracket:
        pop from the stack
        if the popped character is not the matching opening bracket:
            return false
    if stack is empty:
        return true

for each character:
    if it is a starting bracket:
        push it onto stack
    if it is a closing bracket:
        pop from the stack
        if the popped character is not the matching opening bracket:
            return false
    if stack is empty:
        return true
for each character:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
if stack is empty
  return true

[ ( ) ]

Stack

for each character:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
if stack is empty
  return true

[ ]

Stack

for each character:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
  if stack is empty
    return true

[ ( ) ]

Stack

Announcements
  • DoodleJump is due tomorrow at 10:00pm
    – Late handin Sunday at 10:00pm
  • Tetris goes out today
    – on-time handin 11/22
  • Homework 3 goes out today
    – on-time handin 11/15 (no early or late handin)
  • Start EARLY
    – TA hours get very long right before the deadline