BROWN UNDERGRADUATE COUNCIL OF STUDENTS PRESENTS

FALL POLL
LET YOUR VOICE BE HEARD

Fill out by NOV. 6, 2019 for a chance to win spring weekend tickets, gala tickets & more!

OPEN YOUR CAMERA APP TO
SCAN THE QR CODE FOR THE LINK
or go to tinyurl.com/fallpoll
CS Responsibility

Autonomous Vehicles (AV)

• Last month, National Association of City Transportation Officials published second version of Autonomous Vehicle (AV) planning guide
  o Overview of how some of the most influential cities can prepare for AVs.
  o Discussions over policy such as collection of personal data and traffic regulations.
  o AVs could potentially be more sustainable, equitable, and efficient. However, there are many considerations.

Sources:
https://news.stanford.edu/2017/05/22/stanford-scholars-researchers-discuss-key-ethical-questions-self-driving-cars-present/
https://hcri.brown.edu/2013/08/25/the-google-driverless-car-a-cool-thing-that-matters/
What level of safety is tolerable for autonomous vehicles? Are the potential conveniences worth the costs?

4 Major Concerns:

- Minimizing Risk
  - Vulnerability to adversary attacks and hacking
  - Traffic & Construction Zones
- Trolley Problem
  - autonomous vehicles act as moral agents
- Could potentially cause loss of jobs
  - 3.5 million truck drivers in the US
- Transparency & Collaboration
  - Who is designing them
  - Collaboration among disciplines needed
Cartoon Winner Demos!

Ming-May Hu

Yifan Ruan

Adwith Mukherjee

Noah Medina
Cartoon Winner Demos!

Ethan Polley

Joshua Phelps

Yuna Hiraide

Brynn Chernosky
Lecture 17

Stacks and Queues
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\(^1\) (ADT) – in Java, ADTs are supported by interfaces
  - this abstract specification tells us nothing about its implementation – encapsulation! This means that an ADT can be implemented in a variety of ways, and can be used without knowing anything about the implementation

- Can think of abstract specifications of ArrayList and LinkedList as low-level forms of ADT. If we think of their specific implementations (using arrays and linked lists of nodes resp.), they are also data structures. As the name implies, they implement Java’s List interface

- Can use lower-level ADTs in turn to create higher-level ADTs, e.g., stacks and queues, often through the wrapper/containment pattern used in this lecture

\(^1\) This is an informal definition. ADT also has a more formal, mathematical definition.
Abstract Data Types (2/2)

- These lectures show how to implement a variety of common ADTs using linked nodes, and then use those implementations with simple programs to demonstrate their use.

- 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
  - whether or not APIs should be open source was a point of contention between Google and Oracle – led to long, costly legal battle, with Google winning, arguing that they should be open source (source)
Stacks

- **Stack** has special methods for insertion and deletion, and two others for size
  - push and pop
  - isEmpty, size
- 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
Methods of a Stack (ADT spec) – much like an interface

- Add element to top of `stack`
  ```java
  void push(Type el)
  ```
- Remove element from top of `stack`
  ```java
  Type pop()
  ```
- Returns whether `stack` has any elements
  ```java
  boolean isEmpty()
  ```
- Returns number of elements in `stack`
  ```java
  int size()
  ```

Note: `public` keyword not added here and in other specs, but should add mentally as user and implementer of this ADT
Stack Constructor

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

- When using a stack, you will replace Type with the type of object your Stack will hold – enforces homogeneity

- Note the Stack ADT contains a MyLinkedList ADT (“composition” pattern), using the “wrapper” pattern to hide or modify functionality of the contained ADT and to add other methods

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

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

- Let’s see the behavior...
- When an element is pushed, it is always added to front of list
- Thus the Stack delegates to the LL to implement push
Implementing Pop

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

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

- **Stack** will be empty if **_list** is empty - delegation

- Returns a boolean that is **true** if **Stack** is empty and **false** otherwise

```java
//in the Stack<Type> class ...
public boolean isEmpty() {
    return _list.isEmpty();
}
```
size

- Size of **Stack** will be the number of elements that the Linked List contains - delegation

- Size is updated whenever a **Node** is added to or deleted from **_list** during **push** and **pop** methods

```java
// in the Stack<Type> class ...
public int size() {
    return _list.size();
}
```
push(1)

push(2)

push(3)

pop()

push(4)

pop()

pop()
Look over the following code:

```java
Stack<HeadTA> myStack = new Stack<HeadTA>();
myStack.push(htaAngel);
myStack.push(htaLucy);
myStack.pop();
myStack.push(htaNoah);
myStack.pop();
```

Who’s left in the stack?

A. htaLucy  
B. htaNoah  
C. htaAngel  
D. none of them!
Example: Execution Stacks

- Each method has an Activation Record (AR)
  - contains an execution pointer to instruction to be executed next in method – code is immutable but local variables are not
  - thus also contains all local variables and parameters of method

- When methods execute and call other methods, Java uses a Stack to keep track of the order of execution
  - 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 clone, when executable pointer points into same immutable code, but different values for variables/parameters
Execution Stacks

A calls B
B calls C
... etc.

When E finishes, its AR is popped.
Then D’s AR is popped, etc.
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 ADTs

- This implementation of the stack ADT uses a wrapper over a contained Linked List, but user of ADT has no knowledge of that.

- Could also implement it with an Array or ArrayList,
  - Array implementation could be less efficient as we would have to expand our Array as we push more objects onto the Stack.
  - User’s code would not be affected if the implementation of Stack changed (as is true for methods as well, if their semantics isn’t changed)

- We’ll use the same technique to implement a Queue
What are Queues?

● Similar to stacks, but elements are removed in different order
  o information retrieved in the same order it was stored
  o **FIFO**: First In, First Out (as opposed to stacks, which are **LIFO**: Last In, First Out)

● Examples:
  o Standing in line at the checkout counter or movie theater
  o waitlist for TA hours after randomization

Server at Seattle restaurant reminding herself what order customers get served in
Methods of a Queue (ADT spec)

- Add element to end of queue
- Remove element from beginning of queue
- Returns whether queue has any elements
- Returns number of elements in queue

void enqueue(Type el)
Type dequeue()
boolean isEmpty()
int size()
Enqueuing and Dequeueing

- Enqueuing: adds a node
- Dequeueing: removes a node

Before Enqueuing

1  2  3
head of queue  tail of queue

After Enqueuing

1  2  3  4
head of queue  tail of queue

student to add
Enqueuing and Dequeuing

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

Before Dequeuing

1 2 3 4

head of queue tail of queue

After Dequeuing

1 2 3 4

ddequeued student head of queue tail of queue
Our Queue

- Again use a wrapper for a contained Linked List. As with Stack, we’ll hide most of LL’s functionality and provide special methods that delegate the actual work to the LL.
- 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- delegation

- This will add node to end of **_list**

  ```java
  public void enqueue(Type newNode) {
    _list.addLast(newNode);
  }
  ```
We want first node in _list

Use _list’s removeFirst method - delegation

```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();
}
```
People used to be literate. You pick up a copy of Milton, and the sentences have so many nested clauses that a modern reader can hardly remember what the subject is.

It's too bad we don't live in a culture that appreciates the structure of language... the possibilities.

Meanwhile, in the CS department...

Hey Dave (which I am calling you (though I'm happy to call you something else (happy in the sense of willing) if you prefer) because Sally (from across (3 doors down (down meaning east)) the hall (though she'll be moving (different building, not different city (ha ha!)) shortly)) said you prefer it) wanna get a coffee?

Sure!
In order from head to tail, a queue contains the following: jim, dwight, pam, michael. We remove each person from the queue by calling dequeue() and then immediately push() each dequeued person onto a stack.

At the end of the process, what is the order of the stack from top to bottom?

A. jim, dwight, pam, michael  
B. jim, michael, dwight, pam  
C. michael, pam, dwight, jim  
D. It's random every time.
Exercise 1 (1/4)

- How can we use a **Stack** to reverse a Linked List?
- Linked List: Michael, Dwight, Jim, Pam
- Note: user wouldn’t see **head** and **tail** – implementation detail
Exercise 1 (2/4)

- 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 (3/4)

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

head
Michael  Dwight  Jim  tail
Pam
while(!stack.isEmpty()){
    _list.addLast(stack.pop());
}
Exercise 2 (1/2)

- Check for balanced parentheses in a given string
- 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
  - if stack is empty, return false
- 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!
Exercise 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 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 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 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

Match!  Keep going…
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

[ ( ) ]

Match! Keep going…

Stack
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

[ [ ( ) ] ]

Stack
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

[()]
Exercise 2 Actual Code

```java
for (int i=0; i<parenthesesArray.length; i++) {
    //If the element at this index is either starting bracket, push onto stack
    if (parenthesesArray[i].equals("[") || parenthesesArray[i].equals("(") ) {
        myStack.push(parenthesesArray[i]);
    }
    //If the element at this index is either closing bracket, pop off of stack
    //Note use of built-in equals() method to compare Strings- returns a boolean
    if (parenthesesArray[i].equals("]") || parenthesesArray.equals(")") ) {
        String popped = myStack.pop();
        if (parenthesesArray[i].equals("(") && !popped.equals("(" ) {
            return false;
        }
        else if (parenthesesArray[i].equals("]") && !popped.equals("[") ) {
            return false;
        }
    }
}
if (myStack.isEmpty()) {
    return true;
}
```
Exercise 3: TA Hours Line (1/2)

• Let’s model the TA hours line
• Because it is FIFO, we need to use a queue!
• What functionality do we need?
  o a method for students to be added to the line
  o a method for TAs to help the line until it is empty
Exercise 3: TA Hours Line (2/2)

• Start by initializing `_queue` and `_ta`
• Define a method for adding to hours line
  o this can be used before hours or during hours to sign up
• Define a method for seeing a student – uses CS15TA’s `help()`
• Define a method for emptying the queue
  o useful after the cutoff is set

```java
public class TAHoursLine{
    private Queue<Student> _queue;
    private CS15TA _ta;

    public TAHoursLine(CS15TA ta){
        _queue = new Queue<Student>();
        _ta = ta;
    }

    public Student addToLine(Student s){
        return _queue.enqueue(s);
    }

    public void seeStudent(){
        _ta.help(_queue.dequeue());
    }

    public void holdHoursUntilCutoff(){
        while(!_queue.isEmpty()){
            this.seeStudent();
        }
    }
}
```
Announcements

● Tetris was released over the weekend!
● Section this week is a design discussion
  o meet in your section rooms, NOT the SunLab
● Tetris deadlines:
  o Early: Thursday, 11/14
  o On-time: Saturday, 11/16
  o Late: Monday, 11/18