CS18000: Problem Solving and Object-Oriented Programming

Dynamic Data Structures
Video 1
Implementing an ArrayList
Dynamic Data Structures

Dynamic Arrays
Linked Lists

“Algorithms + Data Structures = Programs”
Niklaus Wirth
Some Definitions

• *data structure*: a way to organize, store, and retrieve information in a program

• *dynamic data structure*: a data structure whose memory use can grow and shrink as necessary to store the information being maintained
ArrayList

• Like an array, but grows as elements are added
• How can we write our own version of ArrayList?
• Use what you’ve got:
  – Fixed-sized arrays
  – Abstraction through class definition
  – Accessor and mutator methods
Implementing an ArrayList

• Hide details inside class (ArrayList)
• Use underlying fixed-size array
• Keep track of
  – actual number of elements currently stored vs.
  – capacity of the underlying array
• Use accessor and mutator methods to control access, enforcing the abstraction
Expanding the Underlying Array

• If actual number of elements becomes larger than current capacity...
• Allocate new underlying array
• Copy old array elements into it
• Free old array (and elements it references)
What Methods in ArrayList?

- Allocate new instances
  
  ```java
  ArrayList a = new ArrayList()
  ```
- Add elements to the end of the array
  
  ```java
  a.add("hello");
  a.add("there");
  ```
- Replace (set) a specific element in the array
  
  ```java
  a.set(0, "world");
  ```
- Get a specific element of the array
  
  ```java
  System.out.println(a.get(1));
  ```
- Note: The underlying implementation is hidden
Example: ArrayList (1)

```java
public class ArrayList {
    private String[] strings;
    private int size;

    public ArrayList() {
        strings = new String[10];
        size = 0;
    }

    public String get(int i) {
        return strings[i];
    }

    public void set(int i, String s) {
        strings[i] = s;
    }
}
```
public void add(String string) {
    if (size >= strings.length)
        reallocate();
    strings[size++] = string;
}

private void reallocate() {
    String[] newstrings = new String[strings.length * 2];
    for (int i = 0; i < size; i++)
        newstrings[i] = strings[i];
    strings = newstrings;
}
Video 2
Linked Lists
What’s Wrong with this ArrayList

• Holds only a single type
  – Elements are Strings
  – What if we want to store ints (Integers) or Trees?
  – Must create a new ArrayList class for each type, or use “generics”

• Efficiency considerations
  – Doubling each time can mean substantial wasted space
  – Performance hit every time new internal array is allocated and must be initialized from old internal array

• An Alternate Approach: Linked List
A Linked List

head

value
"hello"
link

value
"there"
link

value
"world"
link
null
Linked List: Another View

LinkedIn

Node objects

value next value next value next value next

"fleas" "has" "dog" "my"

head size 4
Creating a LinkedList Class

• “Outer” (LinkedList) class:
  – Implements accessor and mutator methods like ArrayList
  – Keeps “head pointer” to head of linked list
  – Keeps “size” variable to track the size

• “Inner” (nested) class
  – Implements individual nodes
  – Each node is linked to another node (except the last)
Adding to a Linked List

- Variable head points to "head node"
- Create new node with link field pointing to current "head node"
- Update head with new node just created

- Creates linked list "backwards"
Walking a Linked List

• Start “current node” at “head node”
• While “current node” is not null
  – “Visit” it (access value stored there)
  – Replace “current node” with link to next node
• When “current node” is null, we’ve reached the end of the list
Other Linked List Operations

• Get $i^{th}$ element of list
  – “Walk” through $i$ nodes
  – May be slow for long lists
• Use “tail pointer” to append to end of list
• Use multiple pointers to insert in middle of list
LinkedList: Basic Definition

```java
public class LinkedList {
    private Node head;

    private class Node {
        String value;
        Node link;
    }
}
```

• Need accessor and mutator methods to use!
Video 1
Implementing a Linked List
LinkedList: Operations

• LinkedList() null constructor
• void add(String s)
  add String s to the linked list
• String[] toArray()
  create and return an array with all elements
• int getSize()
  returns the number of items currently in the linked list
Problem: ProcessFile

- Read lines from a file one at a time
- Unknown number of lines
- Store in a data structure (linked list)
- Create array with one element per line
Example: ProcessFile

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ProcessFile {
    public static void main(String[] args) throws FileNotFoundException {
        LinkedList list = new LinkedList();

        Scanner in = new Scanner(new File("dickens-tale-of-two-cities.txt"));
        while (in.hasNextLine()) {
            list.add(in.nextLine());
        }

        String[] array = list.toArray();
        System.out.printf("read %d lines\n", list.getSize());
        for (int i = 0; i < array.length; i++)
            System.out.println(array[i]);
    }
}
Example: LinkedList Version 1 (1)

public class LinkedList {
    private Node head;
    private int size;

    private class Node {
        String value;
        Node link;

        Node(String value) {
            this.value = value;
            size++;
        }
    }

    public LinkedList() {
        head = null;
        size = 0;
    }
    // continued...
// ...continued
public void add(String s) {
    Node node = new Node(s);
    node.link = head;
    head = node;
}
public int getSize() {
    return size;
}
public String[] toArray() {
    String[] array = new String[size];
    int i = 0;
    Node node = head;
    while (node != null) {
        array[i++] = node.value;
        node = node.link;
    }
    return array;
};
Linked List Version 1: Problem

- List created in reverse order
- Head points to last element added
- Array elements in reverse order
- Solution:
  - Add tail pointer
  - As nodes are added, update tail rather than head
- With head and tail pointer, linked list can have nodes added at either end
head

```
[ ] ← [ "A" ] → [ "B" ] → [ "C" (null) ]
```

tail

```
public class LinkedList {
    private Node head;
    private Node tail;
    private int size;

    private class Node {
        String value;
        Node link;

        Node(String value) {
            this.value = value;
            size++;
        }
    }

    // continued...
}
Example: LinkedList Version 2 (2)

// ... continued

public LinkedList() {
    head = tail = null;
    size = 0;
}

public void add(String s) {
    Node n = new Node(s);
    if (head == null)
        head = n;
    if (tail != null)
        tail.link = n;
    tail = n;
}

// getSize and toArray unchanged from version 1

head: null

A

n

head

A

null

null

n

B

null
Video 2
Stacks and Queues
Dynamic Data Structures

Stacks

Queues
What methods does a data structure provide?

• When asking “What methods?” ...
  – ArrayList: add, get, set, ...
  – LinkedList: add, toArray, ...

• ... answers reveal the “features” of the data structure...

• How is the data *accessed*?

• Not how those features are *implemented*?

• Implementation is hidden or “abstract”
Abstract Data Type (ADT)

• A description of the behavior of a data type (class) without specifying an implementation of that behavior
• User of data type unaware of implementation details
• User does not (cannot) make assumptions about the implementation
• Gives implementer of ADT maximum flexibility
Two Common ADTs

• Stack (LIFO)
  – Models a “stack of plates” (for example)
  – Operations:
    • push an element on the stack
    • pop an element off the stack
    • isEmpty: check if the stack is empty

• Queue (FIFO)
  – Models a “line of people” (for example)
  – Operations:
    • put an element at the end of the queue
    • get next element from the front of the queue
    • isEmpty: check if the queue is empty
Stack Uses

• RPN expression evaluation (coming up)
  Same as used in Java Virtual Machine (JVM)
• Managing the storage of local variables for methods being executed
• Undo operations
Queue Uses

- Files sent to a printer
- Threads (or processes) waiting for access to the CPU (or a core)
- Network packets waiting for transmission
- Producer/consumer relationships:
  - producers generate requests,
  - enqueue them,
  - consumers remove from queue and process them
Problem: RPN Evaluation

• Input is a string of space-separated numbers and operators
• \((3 \times 5 - 5) \times (3 + 5) / 2 = 40\)
• Treat as mathematical expression in “Reverse-Polish Notation” (RPN)
• Reminder: similar to Java byte code
• Example:
  \(3 \ 5 \ \ast \ 5 \ - \ 3 \ 5 \ + \ \ast \ 2 \ /\)
  \(= \ 40\)
• Use a stack to carry out the computation
\[
\begin{align*}
3 \times 5 &= 15 - 5 \\
3 + 5 &= 10 \times 8 \\
80 \div 2 &= 40
\end{align*}
\]
public class Evaluator {
    public static int evaluate(String s) {
        Stack stack = new LinkedListStack();
        String[] tokens = s.split(" ");
        for (String token : tokens) {
            if (token.matches("[0-9]"))
                stack.push(Integer.parseInt(token));
            else { // note non-standard compact formatting to fit slide...
                int op2 = stack.pop(); int op1 = stack.pop();
                if (token.equals("+")) stack.push(op1 + op2);
                else if (token.equals("-")) stack.push(op1 - op2);
                else if (token.equals("*")) stack.push(op1 * op2);
                else if (token.equals("/")) stack.push(op1 / op2);
                else throw new RuntimeException("unknown operator");
            }
        }
        return stack.pop();
    }
}
Example: RPN Evaluator (2)

// main method...

public static void main(String[] args) {
    String s = "3 5 * 5 - 3 5 + * 2 /";
    System.out.printf("%s = %d\n", s, evaluate(s));
}

Video 1
Implementing a Stack
Example: Stack Interface

```java
public interface Stack {
    boolean isEmpty();  // true if there are no elements on the stack
    void push(int value);   // push element on the top of the stack
    int pop();             // remove element from the top of the stack
}
```
Example: LinkedListStack

```java
public class LinkedListStack implements Stack {
    private class Node {
        int value;
        Node link;
    }

    private Node head = null;

    public void push(int x) {
        Node n = new Node();
        n.value = x;
        n.link = head;
        head = n;
    }

    // continued...
}
```
head \rightarrow 8 \rightarrow 4 \text{ (null)}

n \rightarrow 21 \text{ (null)}
public int pop() {
    if (isEmpty())
        throw new RuntimeException("Can't pop empty stack");
    int value = head.value;
    head = head.link;
    return value;
}

public boolean isEmpty() {
    return head == null;
}
}
head: 21 \rightarrow 8 \rightarrow 4 \rightarrow \text{null}

value: 21
Stack Implementation with Array

• Use a static array that is “big enough” or a dynamic array like an ArrayList
• Maintain a separate “size” variable that tracks the current stack size
• isEmpty: size == 0
• push: array[size++] = value
• pop: return array[--size]
• Add error checking for empty stack and overflow
<table>
<thead>
<tr>
<th>array</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stacks in Java Utils Package

• Basic stack...
  http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

• Generalized stack...
  http://docs.oracle.com/javase/8/docs/api/java/util/Deque.html
  – Deque (pronounced “deck”): “double-ended queue”
  – Provides methods for both stack-access and queue access
Video 2
Implementing a Queue
The Queue ADT

• Java operations
  – add
  – remove
  – peek

• Other names
  – enqueue
  – dequeue
  – front
Example: Queue Interface

```java
public interface Queue {
    boolean isEmpty();    // true if there are no elements in the queue
    void add(int value);  // add element to the end of the queue
    int remove();         // remove element from front of the queue
    int peek();           // “peek” at front element
}
```
Example: LinkedListQueue (1)

```java
public class LinkedListQueue implements Queue {
    private class Node {
        int value;
        Node link;

        Node(int value) {
            this.value = value;
        }
    }

    private Node head = null;
    private Node tail = null;

    public boolean isEmpty() {
        return head == null;
    }

    // continued...
}
```
// continued...

public void add(int value) {
    Node node = new Node(value);
    if (isEmpty()) {
        head = tail = node;
    } else {
        tail.link = node;
        tail = node;
    }
}

public int remove() {
    int value = head.value;
    head = head.link;
    if (head == null)
        tail = null;
    return value;
}

// continued...
head

\[ \rightarrow \]

\[ 4 \rightarrow 8 \rightarrow 21 \rightarrow \text{null} \]

tail \[ \rightarrow 1 \]
Example: LinkedListQueue (3)

// continued...

    public int peek() {
        return head.value;
    }

// continued...
Same Concepts, Many Names

• Stack...
  – push, pop, top
  – push, pop, peek

• Queue...
  – put, get, peek
  – enqueue, dequeue, front

• Deque...
  – Stack: addFirst, removeFirst, peekFirst
  – Queue: addLast, removeFirst, peekFirst
Video 3
Generic Classes
Generic Classes

• Given the need for a ...
  – Stack of Integers
  – Stack of Strings
  – Stack of Trees
• Would like to just implement once
• Allow the type to be a parameter
• Generics allow <T> notation to indicate a “type parameter”
• Examples:
  var s = new Stack<Integer>();
  var list = new ArrayList<String>();
Examples from the Java Library

- ArrayList (already seen)
- Stack, Queue (interface), Deque (interface)
- LinkedList (implements Stack, Queue, Deque)
- HashMap (very useful!)
HashMap

• Stores “values” referenced by a “key”
• A “dictionary” data structure
  – “word” -> “definition”
  – We say, “maps keys to values”
• Also known as an “associative array”
• HashMap<K,V>
  – K: type of the “key”
  – V: type of the “value”

var map = new HashMap<String, Tree>();
map.put("elm", new Tree("elm", 34.5));
map.put("maple", new Tree("maple", 14.2));
Tree t = map.get("elm");
Making Your Own Generic Class (1)

ArrayList<String> = new ArrayList<String> ();

ArrayList<Integer> = new ArrayList<Integer> ();

Note: Java does not allow a generic array. So, we must use an array of Objects and cast each of the values to the generic type as needed.
Making Your Own Generic Class (2)

```java
public class ArrayList<Ty> {
    private Object[] values;
    private int size;

    public ArrayList() {
        values = new Object[10];
        size = 0;
    }

    public Ty get(int i) {
        return (Ty) values[i];
    }

    public void set(int i, Ty s) {
        values[i] = s;
    }
}
```
public void add(Ty item) {
    if (size >= values.length)
        reallocate();
    values[size++] = item;
}

private void reallocate() {
    Object[] newvalues = new Object[values.length * 2];
    for (int i = 0; i < size; i++)
        newvalues[i] = values[i];
    values = newvalues;
}