CS 1332 - Week 01

Monday

Topics

I. Syllabus review

Wednesday

Topics

I. 1331 Review
   A. Point Class

```
public class Point {
	public int x = 0;
	public int y = 0; // Note B
	public Point(int a, int b) {
		x = a; // OR this.x = a
		y = b; // OR this.y = b
	}
	public Point(int a) {
		this(a, 0); // gives a default value of 0 to y
	}
}
```

   B. Public vs. private visibility
      1. When deciding visibility of a variable, ask: what do we want to access and what do we want to restrict access to?
      2. If x and y are public, we can directly access them from other classes.
         a) Point point = new Point(1, 2);
         b) point.x = 3; // this works because x is public
      3. If x and y are private, we can only access them from this class and can allow access from other classes by writing getters/setters.
         a) Point point = new Point(1, 2);
         b) point.setX(3); // point.x would not work since x is private
   C. Constructor chaining
      1. Calling another constructor within a constructor
      2. Good for allowing flexible instantiation based on the info you do have
      3. Good for setting default values when only some of the info is provided

II. Arrays
   A. Storage: allocate contiguous space in memory
   B. Pros:
      1. Flexible in what they can store - primitives, Objects, references, etc.
      2. Constant time access when index is known
         a) Accessing when index is not known (searching) → O(n)

C. Cons:
    1. If you have only allocated enough space for n items, and you want to add n+1, you must regrow/resize the array.
        a) Time complexity of adding the n+1 element: O(n) → you must copy over all n elements from the original array
        b) Space complexity of adding the n + 1 element: O(2n) → you want to double the capacity of the array

0	1	2
Tim	Alok	Adrianna

→ resize

0	1	2	3	4	5	6
Tim	Alok	Adrianna	Elena			

        c) Space vs. Time - memory is cheap on modern systems; we trade space for time by making a larger array so we resize less often
    2. Memory is static - we must manually resize and move the data to a larger chunk of memory if we exceed the capacity of the current array
        a) Solution - ArrayList!

III. ArrayList
A. Coding demo - how you've probably used an ArrayList before

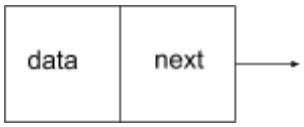
```
ArrayList<String> A2 = new ArrayList<String>();
A2.add("Tim"); // adds one element
```

B. The ArrayList is backed by an Array
    1. ArrayLists should not have null spaces between data elements, because of this, we must often shift data to fill null spaces after remove operations
C. Operation efficiencies:
    1. adding to the front - O(n) → must shift elements to make space
    2. adding to the back - amortized O(1) → no need to shift, but *amortized* because of need to perform an O(n) resize operation after n add operations
        a) amortized - when an "expensive" operation happens infrequently enough that we can "average" it over the runtimes of the less expensive operations that happen more frequently
    3. removing from the front - O(n) → must shift elements to fill empty space
    4. removing from the back - O(1)
    5. adding/removing at a given index - O(n) → shift data around the index
    6. accessing at a given index - O(1) → the ArrayList is backed by an Array
D. Cons:
    1. Cannot store primitives (ints, booleans, etc.)
    2. Still requires O(n) resizing

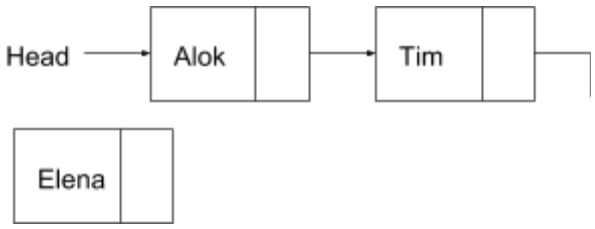
E. Pros:
    1. Data elements are stored contiguous in memory
    2. Dynamic memory - even though we resize the backing array behind the scenes, we still consider the ArrayList to be dynamic because users can add as much data as they want without having to call a resize method on the ArrayList.

IV. LinkedList

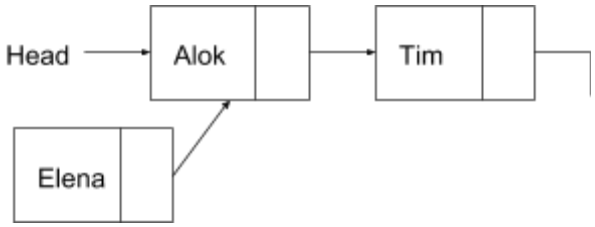
A. Utilize structures called nodes which contain data and a next pointer



B. To keep track of the list, you always need a head reference
    1. If the list is empty, the head is null
    2. To add:
        a) Create a new node containing the data you want to add



        b) Set the new node's next to the current head



        c) Set the head to the new node

C. Coding demo

```
public class SinglyLinkedList {
      private Node head;
      public SinglyLinkedList() {
            head = null; // default, set head to null
      }

      public void addToFront(int data) {
            head = new Node(data, head);
      }
```

```
public String toString() {
      Node current = head;
      String answer = "";
      while (current != null) {
            answer += current + " "; // record current
            current = current.next; // move current
      }
      return answer;
}

private class Node { // private inner class
      private int data;
      private Node next;
      private Node(int data, Node next) { // full cstr
            this.data = data;
            this.next = next;
      }
      private Node(int data) { // we only know data
            this(data, null); // constructor chaining
      }
      public String toString() {
            return (String)data;
      }
}
```

D. Writing the toString():
    1. The node toString() should just print out the data
    2. To iterate through the SLL:
        a) Create a “current” node variable to traverse through the list
        b) Set current equal to head (our starting point)
            (1) If we traversed with the head pointer itself, we’d lose access to the front of the list
        c) While current is NOT null, append the node to the string we’ll return and set current equal to current’s next to move it along

E. Operations:
    1. addToFront() - O(1)
        a) Create the new node
        b) Point the new node’s next to the head
            (1) If the list is empty, the head is null which is what we would want any way if we were adding to an empty list
        c) Set the new node to be the head
    2. addToBack() - O(n)
        a) Iterate until current’s next is null, not until current is null
        b) We need access to the previous node to add one after it

Friday

Topics

I. Singly Linked List/Linked List Review
   A. Adding to the back
      1. We need to iterate to the back, but can't mess with the head.
      2. Edge case → if the head is null, point the head to the new node
      3. General case → create "current" node, set it equal to head, and then loop while current's next is not null. If next is null, you are sitting at the last node and all you need to do is set the next pointer to the new node.
   B. Checking if empty → if the head is null
   C. Removing from the front
      1. Save the data from the head for returning
      2. Set head equal to head's next
      3. There is no need to explicitly delete the first node, because Java has garbage collection!
   D. Removing from the back
      1. Set current equal to head
      2. Loop until current's next's next is null (stop just before the last node)
      3. Edge cases → size is 0 or 1
      4. General case → works for size > 1
   E. SLL extensions
      1. Keep a size variable
         a) Have a size instance variable which is incremented during add operations and decremented during remove operations
         b) Have a method for returning the size
         c) Makes checking for edge cases much easier
      2. Tail reference
         a) Points to the last node in the list
         b) Makes adding to the back easier
            (1) Simply set the tail's next reference to the new node
         c) Does no improve removing from the back → we still have to traverse to the node BEFORE the tail
         d) Edge cases → when size = 0, point the head and tail at new node
      3. Generic SLL
         a) Allow the nodes to contain any data (not just ints)
         b) We don't want to have to write multiple SLLs for every data type, we want it to be versatile and be able to handle any type of data
         c) We'll change the SLL, the Node class and variables, and all parameters we previously has as ints to a generic type
            (1) Users can now specify the type of data the SLL holds:

```
SinglyLinkedList<String> sll =
            new SinglyLinkedList<>();
```

d) Code demo for a generic SLL:

```
public class SinglyLinkedList<Type> {
	private Node<Type> head;
	public SinglyLinkedList() {
		head = null;
	}

	public void addToFront(Type data) {
		head = new Node<>(data, head);
	}

	private class Node<Type> {
		private Type data;
		private Node<Type> next;
		private Node(Type d, Node<Type> n) {
			this.data = d;
			this.next = n;
		}
	}
}
```

F. Solving the problem of removing from the back → make it doubly linked!

G. Taking it one step further → circularly linked!