Lecture 5: DLLists, Arrays

9/4/2020

Summary of SLLists So Far

One Downside of SLLists

- Inserting at the back of an SLList is much slower than the front

Improvement #7: Fast addLast

- Suppose we want to support add, get, and remove operations, will having a `last` pointer result for fast operations on long lists?

Why a Last Pointer Isn't Enough

`.last` is not enough

- The `remove` operation will still be slow. Requires setting the second to last node's pointer to null, and `last` to the second to last node

Improvement #7: .last and ???

- We added .last. What other changes might we make so that remove is also fast?
  - Add backwards links from every node
  - This yields a "doubly linked list" or DLList, as opposed to our earlier "singly linked list" or SLList

Doubly Linked Lists (Naive)

- Reverse pointers allow all operations (add, get, remove) to be fast
  - We call such a list a "doubly linked list" or DLList
- This approach has an annoying special case: `last` sometimes points to the sentinel, and sometimes points at a "real" node

Doubly Linked Lists (Double Sentinel)

- One solution: have two sentinels
  - One that is always at the front and one that is always at the back

Doubly Linked Lists (Circular Sentinel)

- A single sentinel is both at the front **and** and the back

Improvement #8: Fancier Sentinel Nodes

- While fast, adding `.last` and `.prev` introduces lots of special cases
- To avoid these, either:
  - Add an additional `sentBack` sentinel at the end of the list

    - Make your linked list circular (highly recommended for project 1), with a single sentinel in the middle

Generic Lists

Integer Only Lists

- One issue with our list classes: They only support integers

```
public class SLList<LochNess> {
    private class StuffNode {
        public LochNess item;
        public StuffNode next;

        public StuffNode(LochNess i, StuffNode n) {
            item = i;
            next = n;
        }
    }

    private StuffNode first;
    private int size;

    ...
    ...
    ...
}

public class SLListLauncher {
    public static void main(String[] args) {
        SLList<String> s1 = new SLList<>("bone");
        s1.addFirst("thugs");
    }
}
```

SLists

- Java allows us to defer type selection until declaration

Generics

- Rules for project 1
    - In the .java file implementing your data structure, specify your "generic type" only once at the very top of the file
    - In the .java files that use your data structure, specify desired type once:
        - Write out desired type during declaration
        - Use the empty diamond operator <> during instantiation
    - When declaring or instantiating your data structure, use the reference type:
        - int: Integer
        - double: Double

    - char: Character
    - boolean: Boolean
    - long: Long

Array Overview

Getting Memory Boxes

- To store information, we need memory boxes, which we can get in Java by declaring variables or instantiating objects. Examples:
  - `int x;`
  - `Walrus w1;`
  - `Walrus w2 = new Walrus(30, 5.6);`
- Arrays are a special kind of object which consists of a **numbered** sequence of memory boxes
  - To get ith item of array A, use A[i]
  - Unlike **class** instances which have **named** memory boxes

Arrays

- Arrays consist of:
  - A fixed integer **length**
  - A sequence of N memory boxes where **N=length** such that:
    - All of the boxes hold the same type of value (and have the same # of bits)
    - The boxes are numbered 0 through length-1
- Like instances of class:
  - You get one reference when it's created
  - If you reassign all variables containing that reference, you can never get the array back
- Unlike classes, arrays do not have methods

Arrays

- Like classes, arrays are instantiated with `new`
- Three valid notations:
  - `y = new int[3];`
    - Creates array containing 3 int boxes (32 x 3 = 96 bits total)
    - Each container gets a default value
      - The default value for `int` is `0`
      - The default value for `String` is `null` (holds string references)
  - `x = new int[]{1, 2, 3, 4, 5};`
  - `int[] w = {9, 10, 11, 12, 13};`
    - Can omit the new if you are also declaring a variable
- All three notations create an array, which we saw on the last side comprises:
  - A **length** field
  - A sequence of **No boxes** where **N = length**

Array Basics:

```
int[] z = null;
int[] x, y;
x = new int[]{1, 2, 3, 4, 5};
y = x;
x = new int[]{-1, 2, 5, 4, 99};
y = new int[3];
z = new int[0];
int xL = x.length;

String[] s = new String[6];
s[4] = "ketchup";
s[x[3] - x[1]] = "muffins";

int[] b = {9, 10, 11};
System.arraycopy(b, 0, x, 3, 2);
```

Array Copy

- Two ways to copy arrays:
  - Item by item using a loop
  - Using arraycopy. Takes 5 parameters:
    - Source array
    - Start position in source
    - Target array
    - Start position in target
    - Number to copy

```
System.arraycopy(b, 0, x, 3, 2);
(In Python): x[3:5] = b[0:2]
```

- arraycopy is (likely to be) faster, particularly for larger arrays. Comre compact code
  - Code is (arguably) harder to read

2D Arrays

Arrays of Array Addresses

```
int[][] pascalsTriangle;
pascalsTriangle = new int[4][];
int[] rowZero = pascalsTriangle[0];

pascalsTriangle[0] = new int[]{1};
pascalsTriangle[1] = new int[]{1, 1};
pascalsTriangle[2] = new int[]{1, 2, 1};
pascalsTriangle[3] = new int[]{1, 3, 3, 1};
int[] rowTwo = pascalsTriangle[2];
rowTwo[1] = -5;
```

```
int[][] matrix;
matrix = new int[4][];
matrix = new int[4][4];

int[][] pascalAgain = new int[][]{{1}, {1, 1}, {1, 2, 1}, {1, 3, 3, 1}};
```

- Syntax for arrays of arrays can be a bit confounding. You'll learn through practice
- `int[][] pascalsTriangle;`
  - Array of int array references
- `pascalsTriangle = new int[4][];`
  - Create four boxes, each can store an int array reference
- `pascalsTriangle[2] = new int[]{1, 2, 1};`
  - Create a new array with three boxes, storing integers 1, 2, 1, respectively. Store a reference to this array in pascalsTriangle in box #2
- `matrix = new int[4][];`
  - Creates 1 total array
- `matrix = new int[4][4];`
  - Creates 5 total arrays

Arrays vs. Classes

Arrays vs. Classes

- Arrays and Classes can both be used to organize a bunch of memory boxes
  - Array boxes are accessed using [] notation
  - Class boxes are accessed using dot notation
  - Array boxes must all be of the same type
  - Class boxes may be of different types
  - Both have a fixed number of boxes
- Array indices can be computed at runtime
- Class member variable names CANNOT be computed and used at runtime
  - Dot notation does not work
  - [] notation also does not work

Another view

- The only (easy) way to access a member of a class is with hard-coded dot notation

```
int k = x[indexOfInterest];

double m = p.fieldOfInterest;  // Won't work
double z = p[fieldOfInterest];  // Won't work
// No (sane) way to use field of interest

double w = p.mass;  // Works fine
```