Computer Science 210:

Data Structures
Summary

• Today
  • arrays
  • arrays of objects
  • in-class: add an entry into an array

• READING:
  • GT textbook chapter 3.1
Arrays

- The most common thing you want to do when programming is handle a bunch of data.

- The basic way to do this is the array

```java
int[] a;
//declare a to be an array; a is null
a = new int[10];
//create a: allocate space to hold 10 integers and assign the reference to
//this memory to a
```

- Accessing an array:
  ```java
  a[0], a[1]...a[9]
  a.length
  ```

- Assigning arrays
  ```java
  int[] a = new int[10];
  int[] b;
  b = a;
  ```

- Today we’ll see a general example of arrays, namely arrays of objects.
suppose we have a class that stores game entries that looks like this

```java
public class GameEntry {

    protected String name;  // name of the person earning this score
    protected int score;    // the score value

    /** Constructor to create a game entry */
    public GameEntry(String n, int s) {
        name = n;
        score = s;
    }

    /** Retrieves the name field */
    public String getName() {
        return name;
    }

    /** Retrieves the score field */
    public int getScore() {
        return score;
    }

    /** Returns a string representation of this entry */
    public String toString() {
        return "(" + name + ", " + score + ")";
    }
}
```
• Suppose we want to store high scores for a video games. But we don’t want to store ALL entries. We want store the top 10 highest entries.

• We are going to provide this functionality through a class called Scores

• class Scores needs to store
  • maximum nb of entries
    • in our case 10
    • this should be a constant
  • actual number of entries
  • the entries
    • array of GameEntries

• class Scores needs to provide an insert method that inserts a GameEntry while maintaining the invariant that entries[] represents the top 10 scores seen so far

• To make things easier (for the user, that is), we’re going to maintain entries[] in order of scores
  • decreasing order (why is it better than increasing?)
/** Class for storing high scores in an array in non-decreasing order. */
public class Scores {

    public static final int maxEntries = 10; // number of high scores we keep
    protected int numEntries; // number of actual entries
    protected GameEntry[] entries; // array of game entries (names & scores)

    /** Default constructor */
    public Scores() {
        entries = new GameEntry[maxEntries];
        numEntries = 0;
    }

    /** Returns a string representation of the high scores list */
    public String toString() {
        String s = "["
        for (int i = 0; i < numEntries; i++) {
            if (i > 0) s += ", "; // separate entries by commas
            s += entries[i];
        }
        return s + "]";
    }

    .......
}
public void insert(GameEntry e)

• How do we want this to behave?
  • if entries[] has space:
    • insert e in the right spot; shift things to the right; increment numEntries
  • if entries[] is full:
    • if e is smaller than all scores, do nothing
    • else
      • find the right spot to insert e
      • shift everything to the right one position (thus the last entry is over-written)

• class-work: come up with an implementation of insert
  • works on all cases
  • simple to read
public void insert(GameEntry e) {

    int newScore = e.getScore();
    if (numEntries == MAX_ENTRIES) {
        //if array is full
        if (newScore < entries[numEntries-1].getScore()) return;
    } else numEntries++;

    //if we are here, e needs to be inserted; numEntries includes the new entry
    //start from end and shift entries to the right until finding an entry that’s smaller
    int i = numEntries-1;
    while (i > 0 && entry[i-1].getScore() < newScore) {
        entry[i] = entry[i-1];
        i--;
    }
    //entry[i-1] is the first entry that’s larger than newScore
    //entry[i] has been copied to the right, so all we need to do is replace it
    entry[i] = e;
}
public void insert(GameEntry e) {

    int newScore = e.getScore();
    if (numEntries == MAX_ENTRIES) {
        //if array is full
        if (newScore < entries[numEntries-1].getScore()) return;
    } else numEntries++;

    //if we are here, e needs to be inserted; numEntries includes the new entry
    //start from end and shift entries to the right until finding an entry that’s smaller
    int i = numEntries-1;
    while (i>0 && entry[i-1].getScore() < newScore) {
        entry[i] = entry[i-1];
        i--;
    }

    //entry[i] is the first entry that’s larger than newScore; it has been copied to the
    //right, so all we need to do is replace it
    entry[i] = e;
}

Easy to read ===> easy to write, check that it works, implement, debug
public void remove(int i)

• What should this do?
  • action: remove entry i
  • if i is outside the bounds, print some error message (or throw an exception)
  • otherwise shift all entries to the right of i one position to the left, and decrement numEntries
Remove an entry from Scores

• public void remove(int i)
  • action: remove entry i
  • if i is outside the bounds, print some error message (or throw an exception)
  • otherwise shift all entries to the right of i one position to the left, and decrement numEntries

```java
public void remove (int i) {
    if (i < 0 || i >= numEntries) {
        System.out.println("remove: invalid index");
        exit(1);
    }
    //if we are here then i is a valid index
    //shift everything one position to the left; be careful with last
    //element
    for (j = i; j < numEntries-1; j++)
        entries[j] = entries[j+1];
    numEntries--;
}
```
Arrays in Java

- Java provides a number of built-in methods for performing common tasks on arrays.
- `java.util.Arrays`
  - `equals(a, b)`;
    - performs an element-by-element comparison of `a` and `b` and returns true if all elements are equal.
  - `binarySearch(a, val)`
  - `toString(a)`
  - `sort(a)`

- Note: All static methods
  - Why? So that you can use them without having to instantiate an object.

```java
import java.util.Arrays;
...
int[] a = new int[100];
//assign values to a ...
//...
System.out.print("the arrays is: " + Arrays.toString(a));
Arrays.sort(a);
System.out.print("The sorted arrays is: " + Arrays.toString(a));
```
2D-arrays

- `int[][] a;`
- `int a = new int[3][5];`
- // `a` is an array of 3 rows; each row is an array of 5 columns

```
[3][5] array:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>a[1]</td>
<td>a[2]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- `a[0].length` is 5
- `a[1].length` is 5
- `a[2].length` is 5
- `a.length` is 3
3D-arrays

- int [[[]]] a;
- a = new int [3][4][5];
- a is an array of 3 elements; each element of a is a 2D-array [4][5]
- a.length is 3
- a[0].length is 4
- a[0][0].length is 5
Insertion sort

• Sorting:
  • given an array $a$ of $n$ comparable objects
  • re-arrange $a$ so that $a[0] \leq a[1] \leq a[2] \ldots$

• One of the most fundamental problems

• One solution to sorting: INSERTION sort

```c
for (int i=1; i< n; i++)
    //invariant: $a[0..i-1]$ sorted
    insert $a[i]$ into the right position of $a[0..i-1]$
    //invariant: $a[0..i]$ sorted
```

Note: Think in terms of invariants when you write solutions. They are your proof of correctness.

• Exercise: write code for inserting $a[i]$ into $a[0..i-1]$