CS211 Spring 2007
Prelim 2
April 17, 2007

Solutions

Instructions

Write your name and Cornell netid above. There are 5 questions on 8 numbered pages. Check now that you have all the pages. Write your answers in the space provided. Indicate your answer clearly. Use the back of the pages for workspace. Ambiguous answers will be considered incorrect. The exam is closed book and closed notes. Do not begin until instructed. You have 90 minutes. **Good luck!**

	1	2	3	4	5	Σ
Score	/20	/40	/20	/10	/10	/100
Grader						

1. (20 points) The expression $8-2*3$ evaluates to 2, since it is understood as $8-(2*3)$, following the convention that multiplication has precedence over subtraction. If we wish the operators to be evaluated in a different order, say as $(8-2)*3=18$, we must include the parentheses explicitly. Expressions of this form containing operators such as $+, *, -$ that appear between operands (the values they act upon) are called *infix* expressions. To evaluate such expressions, you either need to know the precedence ordering of the operators, or the expression must be parenthesized to help you interpret which operation to evaluate first.

   An alternative notation used to represent expressions is *postfix notation*. You do not need to learn any operator precedence rules to evaluate postfix expressions. You also do not need to use parentheses. In any postfix expression, all operations appear *after* their corresponding operands, unlike infix, where they appear between their operands. And unlike infix, the order of evaluation of the operators is uniquely determined by the postfix expression. Due to its unambiguous interpretation, postfix notation is convenient to use as an intermediate representation in compilers and calculators.

   Some examples:

Infix expression	Postfix expression	Value
$2+3$	$2\ 3\ +$	5
$7*8$	$7\ 8\ *$	56
$6/2+5$	$6\ 2\ /\ 5\ +$	8
$5+6/2$	$5\ 6\ 2\ /\ +$	8

   To evaluate a postfix expression, say for example $7\ 3\ -\ 5\ 8\ +\ *$, you process the expression from left to right. Whenever you see an operator, you apply it on the two most recently seen operands, compute the result, and replace the operator and two operands by the result in the expression. For example, in the expression $7\ 3\ -\ 5\ 8\ +\ *$, when we encounter the $-$, we compute $7-3$, then replace the subexpression $7\ 3\ -$ by 4. The whole expression now looks like $4\ 5\ 8\ +\ *$.

   We continue processing from left to right. This time, $+$ is the first operator we encounter. We apply $+$ to the two most recently seen operands, namely 5 and 8, which gives $4\ 13\ *$. Finally, this expression evaluates to $13*4$, which is 52.

(Questions on next page)

(a) Describe how to evaluate a postfix expression using a stack. Use English or high-level pseudocode, not Java. Assume that the operands and operators are available as a sequence of parsed tokens.

Read the tokens from left to right. When you see an operand, push it on the stack. When you see an operator, pop the two top operands off the top of the stack, perform the operation on those two operands, and push the result back on the stack.

(b) Evaluate the following expression, showing the state of the stack at each step.
6 5 * 7 3 − 4 8 + * +

6
6 5
30
30 7
30 7 3
30 4
30 4 4
30 4 4 8
30 4 12
30 48
78

2. (40 points) Imagine you want to build an online anagram service where people submit a word or phrase and they get back a list of English words that are anagrams[1] of the string they submitted. In this exercise, we will just stick to single English words. Say you have a list of m different English words, namely your English dictionary. You would like to solve the following problem: given a string s, find all words in the dictionary that are anagrams of s.

   Your friend Leet Haxor from Elbonian State University suggests the following approach. He has written a Java method `isAnagram(String s, String t)` that determines whether `s` is an anagram of `t`. Here is his code:

```
1  public static boolean anagram(String s, String acc, String t) {
2     boolean found;
3     for (int i = 0; i < s.length(); i++) {
4        String u = s.substring(0, i) + s.substring(i + 1);
5        String v = s.substring(i, i + 1) + acc;
6        found = anagram(u, v, t);
7        if (found || (u.length() == 0 && t.equals(v)))
8           return true;
9     }
10    return false;
11 }
12 public static boolean isAnagram(String s, String t) {
13    return anagram(s, "", t);
14 }
```

   (Recall that `s.substring(i,j)` returns the substring of `s` consisting of all characters starting at position `i` up to but not including position `j`, and `s.substring(i)` returns the substring of `s` consisting of all characters starting at position `i` up to the end of the string.)

   Leet suggests that given a string s and the English dictionary d you can solve the problem with the following code:

```
for (String w : d)
   if (isAnagram(s, w))
      System.out.println(w);
```

[1] An *anagram* is a word or phrase obtained by rearranging the letters of another word or phrase. For example, Clint Eastwood = Old West Action, The countryside = No City Dust Here, Evangelist = Evil's Agent, The Morse Code = Here Come Dots, Slot Machines = Cash Lost in'em, Heavy Rain = Hire a Navy, and Desperation = A Rope Ends It.

(a) Prove that Leet's approach is extremely bad. In particular, prove by induction that the comparison in line 3 of Leet's code will be executed $n!$ times for a string of length n. *Hint.* Write down a recurrence describing the running time $T(n)$ of the recursive method `anagram` on inputs of length n.

On an input of length n, the test in line 3 is executed n times, and the recursive call in line 6 calls the method n times with an argument of size $n-1$. Thus the running time is at least $T(n) \geq n + n \cdot T(n-1)$.

We show by induction on n that $T(n) \geq n!$. Basis: $T(1) \geq 1$. Induction step: $T(n) \geq n + n \cdot T(n-1) > n \cdot T(n-1) \geq n \cdot (n-1)!$ (by the induction hypothesis) $= n!$.

(b) Describe an alternative approach that decides if `s` is an anagram of `t` in $O(n \log n)$ time, where n is the length of `s` and `t`.

Sort the characters in `s` and `t` and check if they give the same sequence of sorted characters. The sorting takes $O(n \log n)$ time and the comparison takes $O(n)$ time, so $O(n \log n)$ in all.

(c) If the number of words in the dictionary is m, what is the running time of the algorithm that uses (b) to find all anagrams of `s` in the dictionary, assuming that you are not allowed to preprocess or modify the dictionary?

$O(mn \log n)$

(d) Suggest a way to preprocess/organize the dictionary so that after preprocessing, finding all k anagrams of a given `s` takes $O(n \log n + k)$ time. What is the complexity of preprocessing the dictionary?

Create a `HashMap` that maps strings t of sorted characters to a `HashSet` of all words in the dictionary that give t when their characters are sorted. For each word s in the dictionary, sort its characters to give a string t, then enter s in the `HashSet` associated with t in the `HashMap`. After preprocessing, to find all anagrams of a string s, sort its characters to give a sorted string t, then retrieve the `HashSet` associated with t. This actually takes $O(n \log n)$ time.

The complexity of preprocessing is $O(mn \log n)$.

3. (20 points) Suppose we have a priority queue of elements whose priorities are integers. The usual priority queue operations are

`void insert(Object obj, int pr)`	insert element `obj` with priority `pr`
`Object extractMin()`	extract the minimum-priority element.

We could implement such a priority queue as an array-based heap whose elements are instances of the class

```
class PriorityQueueElement {
   Object data;
   int priority;
}
```

As shown in class, this implementation permits the operations `insert` and `extractMin` to be performed in time $O(\log n)$ (assuming the capacity of the array is large enough that it never has to be resized).

Now suppose we want to extend our priority queue with a new operation

```
resetPriority(PriorityQueueElement pqe, int newPriority)
```

that allows us to change the priority of the given `PriorityQueueElement` to the value of `newPriority`, even after it has already been inserted in the priority queue. We would still like all of the operations, including the new `resetPriority` operation, to be $O(\log n)$-time. Describe briefly how this can be done.

Hint. Changing the priority of an element may cause the heap order to be violated, so you will need to readjust the heap after this operation. Also, you will need to find the given `PriorityQueueElement` in the heap, but you do not have time for a linear search. However, you are allowed to add a field to the `PriorityQueueElement` class to avoid this.

Include a new field `int index` in the `PriorityQueueElement` class that always gives the current location of the object in the heap array. Thus the location of the element can be found in constant time, there is no need to search for it. When changing the priority of an element, the heap order may be violated. If the priority is decreased, the element may have to be swapped with its parent, and so on up the tree, until the heap order is again satisfied. If the priority is increased, it may have to be swapped down with its left or right child, whichever has the smaller `priority`.

While swapping up or down, the `index` field can be adjusted accordingly, so as always to give the current location in the heap array. This takes constant time per swap, so the worst-case time is still $O(\log n)$ for each operation.

4. (10 points) Recall that in a binary search tree, the invariant is that the root of any subtree has value larger than any node in its left subtree and smaller than any node in its right subtree. Write a recursive instance method `Pair bst()` that starts at the root of a binary search tree and verifies that this invariant holds. The method should return a `Pair` consisting of the smallest and largest value in the tree if the tree satisfies the invariant, `null` if not.

```
class Node {
   int value;
   Node leftChild; //may be null
   Node rightChild; //may be null

   Pair bst() {
      Pair lp = null, rp = null;
      if (leftChild != null) {
         lp = leftChild.bst();
         if (lp == null || lp.largest > value) return null;
      }
      if (rightChild != null) {
         rp = rightChild.bst();
         if (rp == null || rp.smallest < value) return null;
      }
      int smallest = leftChild != null? lp.smallest : value;
      int largest = rightChild != null? rp.largest : value;
      return new Pair(smallest, largest);
   }
}

class Pair {
   int smallest;
   int largest;

   Pair(int s, int l) {
      smallest = s; largest = l;
   }
}
```

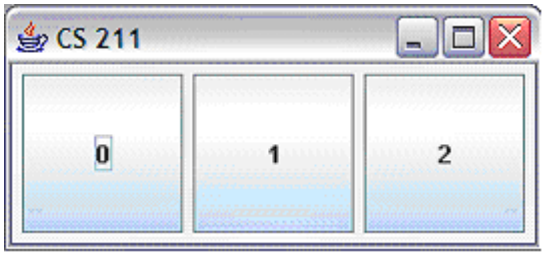
5. (10 points)

Consider the following GUI code:

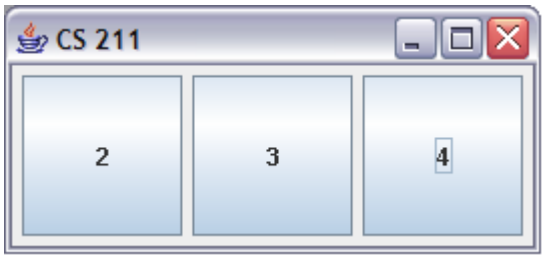
```
final JFrame frame = new JFrame();
frame.setTitle("CS 211");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setLayout(new FlowLayout());

final JButton[] buttons = new JButton[3];

ActionListener buttonActionListener = new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      JButton pushedButton = (JButton)e.getSource();
      int j = Integer.parseInt(pushedButton.getText());
      for (JButton button : buttons) {
         int i = Integer.parseInt(button.getText());
         button.setText(String.valueOf(i + j));
      }
   }
};

for (int i = 0; i < buttons.length; i++) {
   buttons[i] = new JButton(String.valueOf(i));
   buttons[i].setPreferredSize(new Dimension(80, 80));
   buttons[i].addActionListener(buttonActionListener);
   frame.add(buttons[i]);
}

frame.setLocation(200, 200);
frame.pack();
frame.setVisible(true);
```

(Questions are on the following page)

(a) Draw the window as it initially appears.



(b) Draw the window as it would appear after clicking on the rightmost button.



End of Exam