UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **M. Brudno**
Summer 2000

Midterm 2

READ THIS PAGE FIRST. *Please do not discuss this exam before Friday morning with people who haven't taken it.* Your exam should contain 5 problems on 7 pages. Officially, it is worth 25 points.

This is an open-book test. You have one hour and twenty minutes to complete it. You may consult any books, notes, or other inanimate objects (other than computers) available to you. You may use any program text supplied in lectures, problem sets, or solutions. Please write your answers in the spaces provided in the test. Make sure to put your name, login, and discussion section in the space provided below. Put your login and initials *clearly* on each page of this test and on any additional sheets of paper you use for your answers.

Don't panic. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious. Should you feel an attack of anxiety coming on, feel free to jump up and run around outside Dwinelle once or twice.

Your name: ______________________ Login: __________

Discussion Section time: ______________ TA: ______________

1. __________/4

2. __________/4

3. __________/

4. __________/9

5. __________/8

TOT __________/25

Throughout the test assume that the following definitions are available. Note that the List class now has a static append method.

```
/** Standard linked list */
class List {
  public Object head;
  public List tail;

  public List(Object head, List tail)
  {
    this.head = head; this.tail = tail;
  }

  public List(Object head) { this(head, null); }

  /** Appends B to the end of A, returning the result */
  public static List append(List A, List B) {
    if (A == null) return B;
    if (B == null) return A;
    A.tail = append(A.tail,B);
    return A;
  }
}

class ListQueue {
  /** Makes a new ListQueue */
  public ListQueue() {...}
  /** True if the queue is empty */
  public boolean empty() {...}
  /** Remove the top element and return it */
  public Object remove() {...}
  /** Inserts o into the end of the queue */
  public void insert(Object o) {...}
}

interface Enumeration {
  /** True if this Enumeration has more elements.
   */
  boolean hasMoreElements() {...}
  /** Returns the next element of this enumeration and advances the
   *  Enumeration to the following element.
   */
  Object nextElement() {...}
}
```

1. (4 points / 10 minutes) You are building a treap using int labels and the hash function $H(x) = x^2$ % 17 (the underlying heap is a max-heap). Pick a 5 element subset from the set $\{1, 2, ...10\}$ so that if starting from the empty treap you insert these 5 elements **in order** into your treap, it will have maximal height. Draw the corresponding treap.
Subset: Treap:

```
1,2,3,9,10          The treap is a linked list to the left:
1 <- 2 <- 3 <- 9 <- 10
1.5pts for right subset, 2.5 for treap corresponding to your subset.
```

For convinience, $1\%17 = 1$; $4\%17 = 4$; $9\%17 = 9$; $16\%17 = 16$; $25\%17 = 8$; $36\%17 = 2$; $49\%17 = 15$; $64\%17 = 13$; $81\%17 = 13$; $100\%17 = 15$

2. (4 points / 5 minutes) For Homework 5 you had to come up with an algorithm for the "majority" problem: given an array of k elements, determine whether any particular element constitutes a majority, that is whether $> k/2$ elements are equal to it. The elements are not comparable (you cannot compare them for $>$ or $<$, only for $==$). The following is our solution:

```
Make a new stack.
Repeat the following for each element of the array:
  If the stack is empty, push the element onto the stack.
  Else, If it is equal to the element on top of the stack push it as well.
  Otherwise pop the top element off the stack and throw it out, together
     with the current element.
Once you are done, if the stack is empty return false.
If it is not, pop it, and compare the element to every element in the
  array, counting the number of matches. If it is > k/2, return true,
  else false.
```

Explain how to modify our solution (or provide your own) so that it uses a constant amount of extra space, while still running in O(k) time.

```
Instead of using a stack use a single location to store an element and a
counter. The key observation is that all items on the stack are always
equal.
4pts for right answer, 1pt for a constant space non-working answer, 1pt
for a non-constant space working answer unless it is just like ours.
```

3. (1 point / 0 minutes)
Please provide the next line in this sequence:

```
1
11
21
1211
111221
```

312211 (half-point for putting down 42).

4. (9 points / 30 minutes)
You have been hired by CNN to write an election monitor software to be used for all future elections in all countries. They provide you with the following class definitions:

```
class Vote {
  ...
  public String getCandidate() {...}
  ...
}
```

They want you to be able to support the following functions as efficiently as is possible:

```
class ElectionMonitor {
  /** Record this vote for the election */
  registerVote(Vote v) {...}
  /** Return the current Leader */
  String getLeader() {..}
  /** Returns a List of candidates in order of votes, from
   *  largest to smallest */
  List currentResults() {...}
}
```

In particular, they want to make sure getLeader() works very quickly, as everyone always wants to know who is winning, and want the registerVote() method to also work fast. currentResults, however, will not be called often and hence it is possible that it is slower. `Do not` assume that the number of candidates running for office is small, or anything else, unless it is explictly stated.

a. (6 pts) Explain (in English) how you will design the ElectionMonitor class. You can assume the existence of all data structures we discussed in class, you `do not` have to explain how any of the standard methods (e.g. heap construction) work. Be specific, however, about which data structures you are using and how these data structures are interconnected. `Make sure to explain what the running time of the three methods above will be using your implementation.`

```
Use a hash table to map candidates to votes and a special pointer to the
current leader (with name and # of votes).
For registerVote look up candidate in hashtable and increment the # of votes.
If it becomes larger then the # of votes of the leader, change the leader.
Running time O(1).
getLeader return the leader O(1)
currentResults: throw the elements of the hashtable into an array and sort
it with any nlgn algorithm. O(nlgn).
Other solutions exist (using trees, etc.) For full credit, you needed to
get the first 2 in O(lg n) and the last one in n lg n. Using treaps is not a
good idea because the number of votes is not really random enough.
```

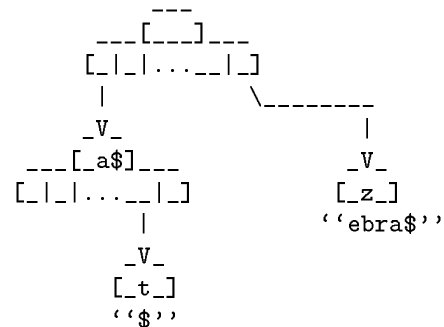
b. (3 pts) Because votes are being cast all the time around the country, your system should be multi-threaded to handle all of the incoming votes. Because your program may be busy doing something else and not have time to process the votes immediately, these votes will be stored in a queue until you get to them. Your co-worker writes the following code:

```
class VoteQueue {
  /** ListQueue is on page 2 */
  private ListQueue myqueue = new ListQueue();

  /** True if this queue is empty */
  synchronized boolean isEmpty() { return myqueue.empty(); }

  /** Adds v to the end of this queue */
  synchronized void addVote(Vote v) {
    myqueue.insert(v);
    notify();
  }

  /** removes the first vote from the queue and returns it */
  synchronized Vote getVote() { return myqueue.remove(); }
}

class ElectionMonitor {
  VoteQueue myvotes;
  ...
  synchronized void processQueue() {
    while (true) {
      while (myvotes.isEmpty()) {
        try {
          wait();
        }
        catch (InterruptedException e) { return; } // Just quit if interrupted
      }
      registerVote(myvotes.getVote());
    }
  }
  ...
}
```

The idea is that many threads with votes could come in at any point and add votes to the VoteQueue. Other threads will process the votes in the queue using the processQueue method. There are two bugs/inefficencies. Explain how to fix them.

```
1. processQueue shouldn't be sunchronized som many threads can run it
2. wait and notify are called on different objects, so waiting threads
will never wake up.
```

5. (8 points / 30 minutes) The data type Trie was discussed in lecture. For this question, instead of saving the whole string in the leaf nodes, we will save just the suffix corresponding to the leaf in order to conserve space; a trie with the strings `a, at, zebra` will look like the follwoing:



You have the following definitions:

```
/** A retrieval tree node */
abstract class TrieNode {
  /** True if this node is a leaf node */
  abstract boolean isLeaf();
  /** Returns the letter to which this node corresponds*/
  char getLetter() {...}
  /** Returns the size of the alphabet of this trie */
  int alphaSize() {...}
  /** An enumeration of all of the Strings in this trie */
  Enumeration allStrings() { return new TrieEnum(this); } // NOTE!!!
}

class InternalTrieNode extends TrieNode {
  boolean isLeaf() { return false; }
  /** returns the nth child of this Trie or null if none */
  TrieNode getChild(int n) {...}
  /** returns true if this node is also a terminal node for a String in the
   *  trie (i.e. it has the ''$'' terminator in it) */
  boolean isTerminal() {...}
}

class LeafTrieNode extends TrieNode {
  boolean isLeaf() { return true; }
  /** returns the suffix stored in this TrieNode without the terminator */
  String getSuffix() {...}
}
```

Assume that all of the methods above are already written. You are to supply the definition of the class `TrieEnum` used for the allStrings() method. For your reference there is a definition of Enumeration on page 2. Although you should feel free to use any algorithm, recursion tends to be best suited for this problem. Also using helper methods can make your code less obfuscated and easier to write.

Continued on next page

```
class TrieEnum implements Enumeration {
  ListQueue L;

  TrieEnum(TrieNode t) {
    processTrie(t, "");
  }

  void processTrie(TrieNode T, String prefix) {
    if (T.isLeaf) {
       L.insert(prefix + T.getLetter() + ((LeafTrieNode) T).getSuffix());
       return;
    }
    if (T.isTerminal)
       L.insert(prefix + T.getLetter());
    for (int i = 0; i < T.alphaSize(); i++)
      if (((InternalTrieNode)T).getChild(i) != null)
        processTrie(T.getChild(i), prefix + T.getLetter())
  }

  boolean hasMoreElements() { return !L.isEmpty(); }
  boolean nextElement() { return L.remove(); }
}
```

```
The grading varied depending on the number of bugs. Major things were
forgetting to pass down the prefix (-2) doing only limited depth (-4)
and others. Minor errors (null checks, casting) cost you .5 pts. It is
very difficult (but possible) to generate the strings one at a time
instead of pre-processing. Most who took that route didn't get far.
```