CS106B
J Zelenski

Handout #36
Mar 10, 2008

Final Practice

Final exam: Friday March 21 12:15-3:15pm
Location TBD (but definitely not Terman ☺)

This is the university-scheduled time for our exam and we are looking forward to seeing all your smiling faces. If you are one of the few students who approached us at the start of the quarter about an unavoidable conflict, please follow up with head TA Jason about the alternate exam Thursday 12:15pm. There will be absolutely no other alternates.

SCPD students: Local SCPD students are asked to attend the regular on-campus exam. This ensures you will be able to ask questions and receive any clarifications that might come up during the exam. Folks outside the Bay Area will take the exam at your local site on Friday. Your SCPD site coordinator will administer the exam. Please send e-mail to Jason to initiate the arrangements.

Coverage

The final is comprehensive and covers material from the entire quarter, but will tend to focus on the topics covered after the midterm. This means you should come prepared for nitty-gritty implementation-side work (pointers, linked lists, trees, graphs, templates, and so on).

It will be a 3-hour exam. It will be open book /open note, but no electronics.

This handout is intended to give you practice solving problems that are comparable to those which will appear on the exam. These problems were chosen from recent exams as fairly representative in terms of format, content, and difficulty.

We highly recommend working through the problems in test-like conditions to prepare for the actual exam. To encourage this, we won't even give out the solutions until next class. Many of our section problems have been taken from previous exams and chapter exercises from the reader often make appearances in same or similar forms on exams, so both of those resources are a valuable source of study material as well.

Be sure to bring the reader with you to the exam. We won't repeat the standard class definitions on the exam, so the reader appendix will come in handy for looking up the library interfaces.

(FYI: All the space usually left for answers was removed in order to conserve trees).

Problem 1: Templates and callback functions

a) Write the generic function **`KeysForMaxValue`** that searches for the maximum value in a map and returns the associated key(s). The function is given a map and a callback function to compare values and returns a vector containing one or more keys that are paired with the largest value in the map. The comparison callback function has the standard form: it takes two arguments and returns a negative result if the first is less than the second, zero if the two are equal, and positive otherwise. If the client doesn't provide a comparison function, a default one that applies the built-in relational operators should be used. This function should be written as a template and should work for maps storing any type of values. You may assume that the map has at least one entry.

As an example, suppose that you have initialized the map **`ages`** like this:

```
Map<int> ages;
ages.add("Zinnia", 3);
ages.add("Rein", 4);
ages.add("Kalev", 2)
ages.add("Ian", 4);
```

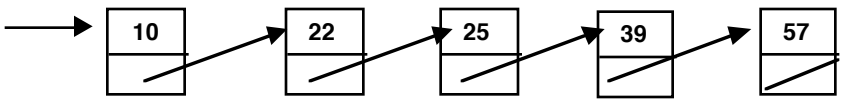
Calling **`KeysForMaxValue(ages)`** returns a vector containing **`{"Rein", "Ian"}`**.

b) Use the **`KeysForMaxValue`** function from part (a) to implement the **`MostFrequentSeed`** function that returns the most frequently occurring seed from a Markov model. The Markov model is represented as in the Random Writer assignment: a map where each key is a seed and its value is a vector of characters, one following each occurrence of the seed within the input text. If there is a tie for most frequent, return a random choice from the most frequent seeds. You should use the **`KeysForMaxValue`** function and can assume it works correctly.

```
string MostFrequentSeed(Map<Vector<char> > & model)
```

Problem 2: Linked lists

The reader discusses two different Set implementations, one layered on the BST class template and another using bit vectors. For this problem you will consider instead using a sorted linked list. Here is a diagram of a linked list set containing the integer elements {10, 22, 25, 39, 57}.



The standard **`Set`** class template interface is given in the reader appendix. Using an internal data structure of a sorted linked list, you will implement the **`contains`** and **`unionWith`** member functions. Remember that you are writing these functions as the **implementer** of the **`Set`** class and thus have direct access to all its internal data structures. Your implementation must adhere to the following constraints:

- The elements are internally stored using a singly-linked list in ascending sorted order.
- The list does not use a dummy cell.
- The number of elements is cached in a data member to avoid traversing the list to count.
- **`contains`** runs in O(N) time where N is the number of elements in the set.
- **`unionWith`** must run in O(N) time where N is the number of elements in the input sets.
- **`unionWith`** should not call any other member functions. This is necessary in order to run in linear time.
- The **`Set`** class must work for elements of any type.

The interface for the linked list version of the **Set** class template is given below:

```
template <typename ElemType>
  class Set {
    public:
      /* Constructor: Set
       * Usage: Set<student> set(CmpStudents);
       * -------------------------------------
       * The constructor initializes an empty set.
       */
      Set(int (cmpfn)(ElemType, ElemType) = OperatorCmp);

      /* Member function: contains
       * Usage: if (set.contains(elem))...
       * ---------------------------------
       * This function returns true if the element is in this set.
       */
      bool contains(ElemType elem);

      /* Member function: unionWith
       * Usage: s1.unionWith(s2);
       * ------------------------
       * This function updates s1 to include all elements in s2.
       */
      void unionWith(Set & otherSet);

    // other public member functions not shown, no changes made

    private:
      struct cellT {
        ElemType value;
        cellT *next;
      };

      cellT *head;
      int (*cmp)(ElemType, ElemType);
      int nElements;
  };
```

You are given the implementation of the **Set** constructor:

```
template <typename ElemType>
  Set<ElemType>::Set(int (*cmpfn)(ElemType, ElemType))
 {
     cmp = cmpfn;
     head = NULL;
    nElements = 0;
 }

// reminders: must run in O(N) time
//            returns true if element in set, false otherwise
template <typename ElemType>
  bool Set<ElemType>::contains(ElemType elem)


// reminders: must run in O(N) time
//            cannot call other public member functions
//            updates receiver set, does not modify other set
template <typename ElemType>
 void Set<ElemType>::unionWith(Set &other)
```

Problem 3: Trees
When implementing a binary search tree, an important efficiency concern is taking care that the tree remains balanced to ensure logarithmic performance for insert and lookup. In lecture, we briefly mentioned self-adjusting trees such as AVL that continually make minor rearrangements to prevent the tree from becoming lopsided. An alternative strategy is to wait until a problem is observed and then perform a tree rebalancing.

Write the function **`Rebalance`** that rebalances a binary search tree. After rebalancing, the tree will contain the same values as before and is still a valid binary search tree, but its nodes have been re-organized into a balanced configuration.

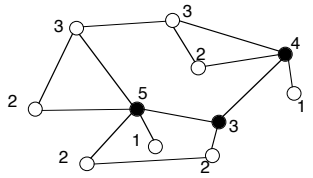
A few **very important** notes:

- Rebalancing must take at most O(N) time where N is the number of nodes in the tree.
- A straightforward way to meet the performance requirement is to transfer the nodes into a Vector in sorted order and then construct the balanced tree from the Vector. Be sure to consider how you can easily fill the Vector in sorted order and how having the nodes in sorted order enables you to directly and efficiently build a balanced tree.
- You are strongly encouraged to decompose your solution into helper functions. Below, we give suggested prototypes for two helpers (one for filling the vector, another for building a tree from a vector). You are free to change/augment our suggested decomposition.
- No nodes are created or destroyed, the existing nodes are just rearranged. This means you should not **`new`** or **`delete`** nodes.

```
struct nodeT {
    string data;
    nodeT *left, *right;
};
                                // suggested prototypes for helper functions
void FillVector(nodeT *t, Vector<nodeT *> &v);
nodeT * BuildTree(Vector<nodeT *> &v, int startIndex, int stopIndex);

void Rebalance(nodeT *& t)
```

Problem 4: Graphs and graph algorithms
A *dominating set* of a graph is a subset of the nodes such that those nodes along with their immediate neighbors constitute all graph nodes. That is, every node in the graph is either in the dominating set or is a neighbor of a node in the dominating set. In the graph diagrammed below, the black nodes mark a dominating set for the graph. Other dominating sets are also possible. (Each node is labeled with the number of neighbors to facilitate tracing the algorithm below.)



Often the goal is to find the absolute minimum size dominating set**,** but it is known that this is a computationally difficult task – too expensive for most graphs. Here is a greedy approximation algorithm to build a small dominating set (not guaranteed to produce the optimal result):

- The set starts empty
- Consider each graph node in order of decreasing number of neighbors (e.g. start by examining the node with the most neighbors)
- If the node is not redundant with the set, add it to the set (a node would be redundant if it and all its neighbors are already dominated by the set)

- Continue until the set dominates the entire graph

Implement the function **`FindSmallDomSet`** that takes the set of graph nodes and returns a subset of nodes that forms a small dominating set as found by the above algorithm.

Hints:
- The **`PQueue`** class should be used to process nodes in order.
- When ordering nodes by number of neighbors, ties can be broken arbitrarily.
- The high-level **`Set`** class operations will be handy here.

```
struct node {
   string name;
   Set<node *> connectedTo;
};

Set<node *> FindSmallDomSet(Set<node *> & allNodes)
```

Problem 5: Class design
The **`Calendar`** class organizes a personal event schedule. Here are specifications for a Calendar:

- A Calendar tracks events scheduled according to date. Each event stores its time, duration, title, and a description.
- The two most frequently performed operations are:
  1. **`displayByDate`**: For a given date, display all events scheduled on that date in time order.
  2. **`findByKeyword`**: For a given keyword, display a list of all events that contain that keyword in its description.
- The Calendar must also allow adding new events, but it is not as critical to optimize this operation since it occurs much less frequently than the above operations.
- It is desirable to be frugal with space where feasible.

For this problem, you will consider various Calendar implementation decisions. You will choose data structures and describe the algorithmic processes for the operations. You will *not* write code to implement the operations.

When asked to make and justify a choice, your justification should address relevant design issues such as big-O efficiency, memory use, ease of coding, scalability, and so on. Your answer will be evaluated on the data structure design and **specifics** of the implementation. You should clearly describe a **successful and sensible** strategy for meeting the requirements and provide **solid justification** for your choices, noting alternative strategies you considered and why they were rejected. Don't obsess over perfectly composed English; short bullets are fine.

Unless you state otherwise, we will assume these implementations for the standard classes:

Vector	array (dynamically resized)
Grid	2-dimensional array
Stack, Queue	linked list
Map	hash table (dynamic rehashing)
BST	balanced binary search tree
Set	BST
PQueue	binary heap

The basic information for an event is collected into this record:

```
struct Event {
  string title;
  Time time;          // uses a custom Time class
  int duration;
  string description; // paragraph of information about the event
};
```

a) The **displayByDate** operation displays all events scheduled on a given date in order of time**.** Below are two usage scenarios for which you are to choose an appropriate data structure for storing events within the Calendar. For each scenario, give C++ declarations of the private data members for the Calendar class. Provide any necessary details so that a competent programmer would understand how data is organized within your structure(s) and how **displayByDate** will be implemented to meet its specification.

Scenario 1: The Calendar tracks events for exactly one calendar year, from January 1 to December 31. Most dates have events scheduled, a few dates are empty. Provide an appropriate data structure for storing events that allows **displayByDate** to run in O(N) time where N is the number of events scheduled on that date. Briefly justify the appropriateness of your choice.

Scenario 2: The Calendar tracks events over an indeterminate range, which could span several years or decades. Many dates are empty, others have several events. Provide an appropriate data structure for storing events that allows **displayByDate** to run in O(lgD + N) time or better where N is the number of events scheduled on that date and D is the number of non-empty dates in the calendar. Briefly justify the appropriateness of your choice.

b) The **findByKeyword** operation displays all events that contain a given keyword within the event description. It is required to run in O(N) time where N is the number of matching events. Provide C++ declarations of additional private data members that are needed, along with any necessary details to understand how the structure(s) are used. Sketch the implementation of **findByKeyword** and justify how this implementation meets the running time specification.

c) Recurring events: You are adding a Calendar feature to support events that repeat, such as a weekly Monday meeting or a happy hour on the 1st of each month. It is desirable that when editing such an event (such as to change its time or description) that all occurrences update as well. Describe an implementation strategy for storing and manipulating recurrent events that allows you to efficiently implement this update operation.

Problem 6: Short answer

a) Quicksort and Mergesort both have O(NlgN) performance in the average case. Give one reason/situation to prefer using Quicksort instead of Mergesort and vice versa.

b) Draw the binary search tree that results from inserting the 10 numbers in this sequence **{24, 8, 16, 30, 1, 9, 42, 25, 18, 29}**.

c) True or false: A pre-order traversal of a heap visits the entries in reverse sorted order. Briefly justify why this is true or provide a counter-example.

d) List two of the techniques used in the implementation of the Boggle lexicon to improve its space and/or time efficiency.