CS106X
Winter 2018

Handout 35S
March 12th, 2018

CS106X Midterm Examination Solution

Once again, your awesome course staff spent the week grading your exams, and I'm happy to share that they're graded! You can visit Gradescope to see how everything turned out.

As promised, the midterm focused on pointers, memory allocation, and linked structures, and deemphasized algorithmic novelty so that I could really make sure to drill the material you need to master to fully appreciate C and C++ and move forward to CS107.

The exam median was a 20.5 out of 25, the mean was a 19.14, and the standard deviation was 7.13. There were no perfect scores, but one person managed to eke out a 31.5! That means 31.5 maps up to a 100, 20.5 maps up to an 80, 9.5 maps up to a 50, and everything else scales up accordingly.

As always, if you have questions about how your midterm was graded, or you feel some of your work was overlooked, you're more than welcome to request a regrade. All regrade requests must go through Jerry, however, and you must come see me in person sometime next quarter during office hours or during some scheduled time. Please don't be shy about asking for a regrade request, since the exam is too large a part of your final grade, and I want everyone moving forward trusting his or her exam was graded properly.

		Score	Grader
1. **`mergeLists`** Memory Trace	[7]	______	______
2. Exponential Trees	[20]	______	______
3. Short Answer Questions	[8]	______	______
Total	**[35]**	______	______

Summary of Relevant Data Types

```
class string {
   bool empty() const;
   int size() const;
   int find(char ch) const; // returns string::npos on failure
   int find(char ch, int start) const; // returns string::npos on failure
   string substr(int start) const;
   string substr(int start, int length) const;
   char& operator[](int index);
   const char& operator[](int index) const;
};

class Vector {
   bool isEmpty() const;
   Type& operator[](int pos);
   const Type& operator[](int pos) const;
   void insert(int pos, const Type& value);
}

class Stack {
   bool isEmpty() const;
   void push(const Type& elem);
   const Type& peek() const;
   Type pop();
};

class Map {
   bool isEmpty() const;
   int size() const;
   bool containsKey(const Key& key) const;
   Value& operator[](const Key& key);
   const Value& operator[](const Key& key) const;
};

class Set {
   bool isEmpty() const;
   int size() const;
   void add(const Type& elem); // operator+= also adds elements
   bool contains(const Type& elem) const;
};

class Lexicon {
   bool contains(const string& str);
   bool containsPrefix(const string& str);
   void add(const string& word);
};
```

Solution 1: `mergeLists` Memory Trace [8 points]

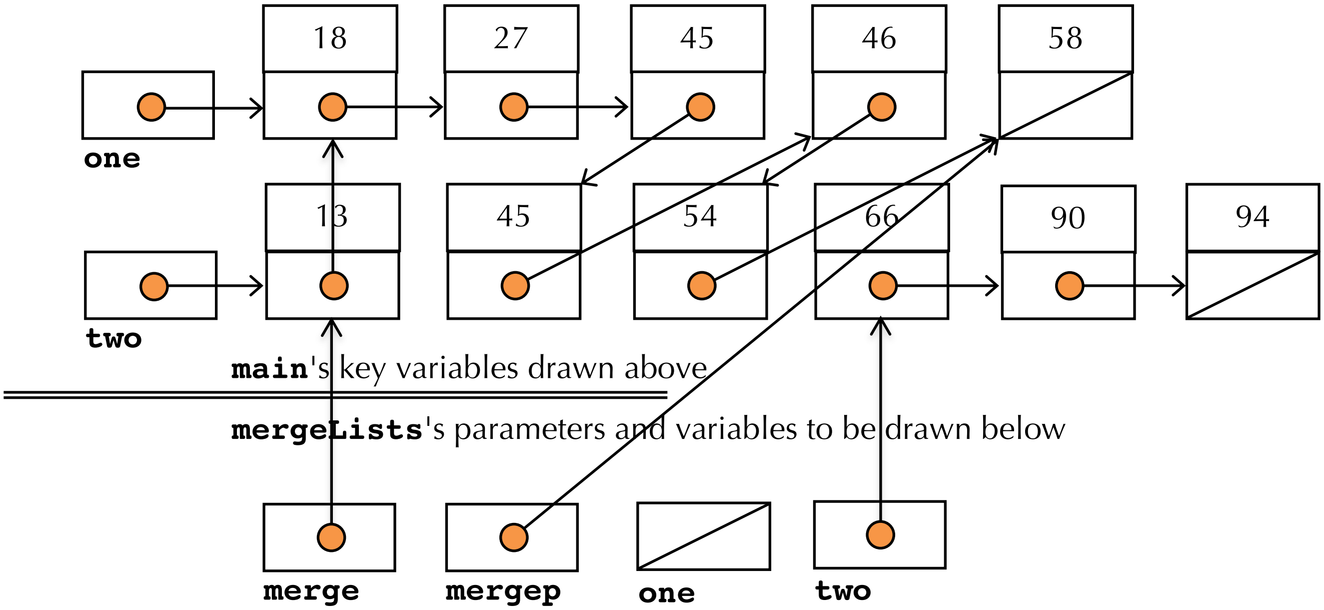
Consider the **`mergeLists`** presented below. Given two sorted linked lists of potentially different lengths, **`mergeLists`** cannibalizes the two incoming lists to construct a single one that is the sorted merge of the two originals.

```
struct node {
   int value;
   node *next;
};

static node *mergeLists(node *one, node *two) {
   node *merge = NULL;
   node **mergep = &merge;

   while (one != NULL && two != NULL) {
      if (one->value <= two->value) {
         *mergep = one;
         one = one->next;
      } else {
         *mergep = two;
         two = two->next;
      }
      mergep = &((*mergep)->next);
   }

   if (one != NULL) *mergep = one;
   else *mergep = two;
   return merge;
}
```

Assume you've been given the following **`main`** program where **`constructList`** builds the lists, as depicted on the next page:

```
int main() {
   Vector<int> v1, v2;
   v1 += 18, 27, 45, 46, 58;
   v2 += 13, 45, 54, 66, 90, 94;
   node *one = constructList(v1);
   node *two = constructList(v2);
   node *merge = mergeLists(one, two);
   freeList(merge);
   return 0;
}
```

In the space provided, draw the state of memory **at the moment the `while` loop test within the call to `mergeLists` fails**. Understand that the **`one`** and **`two`** variables drawn for you are those owned by **`main`**, so you're to draw all four parameters (**`one`**, **`two`**, **`merge`**, and **`mergep`**) for the **`mergeLists`** call, cannibalize the existing lists in your memory diagram by (very neatly) updating all four parameters and all **`next`** pointers as they're updated. Do not redraw nodes, since they're retained by the merge. Your final drawing should be beautiful enough to sit in a special exhibit at the Louvre, and it should convey a clear understanding of linked structures and pointers.

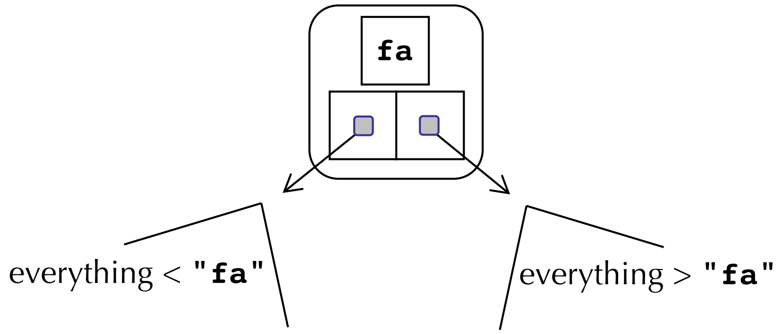


Problem 1 Criteria: 7 points

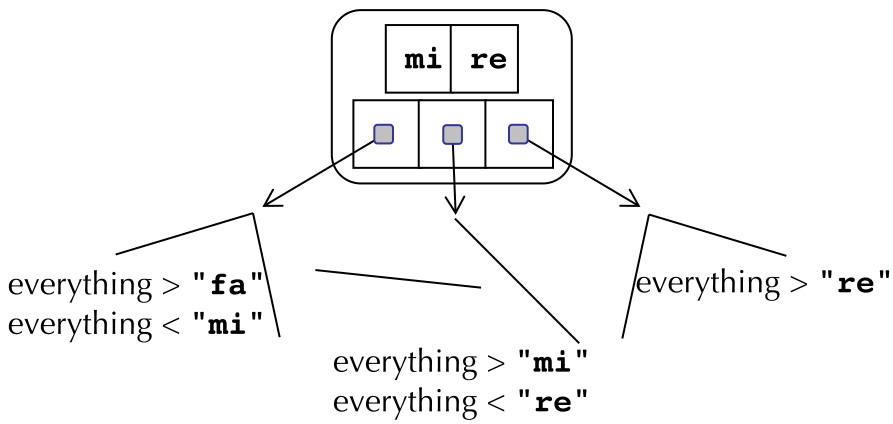
- **merge** addresses the node housing the 13: 1 point
- Original nodes are threaded together in sorted order, save for the chasm between the 58 and the 66, which has yet to be established: 3 points
  - Everything is linked in sorted order: 1 point
  - 45 in upper list comes before 45 in lower list (algorithm biases towards **one**'s nodes): 1 point
  - 58 is not linked to the 66: 1 point
- **mergep** addresses what's unambiguously the next field of the 58 node (allow any reasonable drawing, provided the arrow hits the **NULL** field and not the 58): 1 point
- **one** is **NULL** (not a pointer to some **NULL**, but **NULL**): 1 point
- **two** addresses the node surrounding the 66: 1 point

Solution 2: Exponential Trees [20 points]

Exponential trees are similar to binary search trees, except that the **depth** of the node in the tree dictates how many elements it can store. The root of the tree is at depth 1, so it contains 1 element and two children. The root of a tree storing strings might look like this:



If completely full, a node at depth 2—perhaps the right child of the above root node—might look like this:



Generally speaking, a node at depth **`d`** can accommodate up to **`d`** elements. Those **`d`** elements are stored in sorted order within a **`Vector<string>`**, and they also serve to distribute all child elements across the **`d + 1`** sub-trees.

We can use the following data structure to build up and manage an exponential tree:

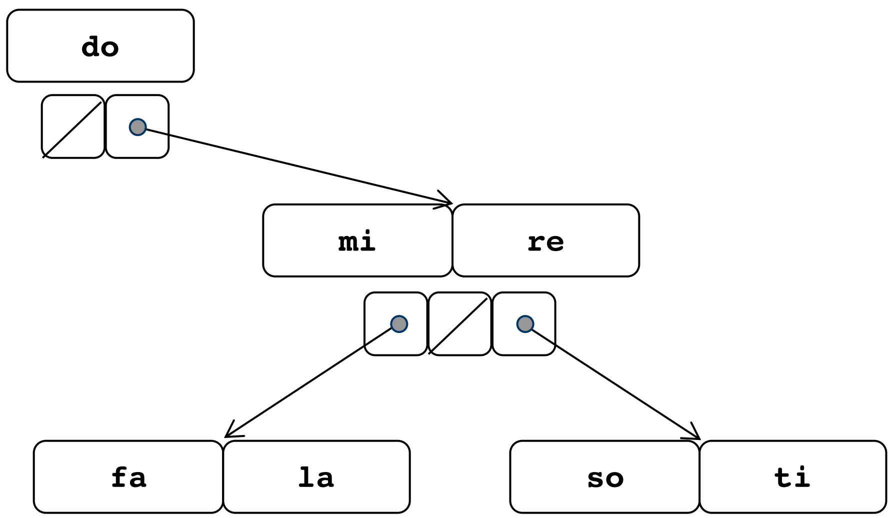
```
struct expnode {
   int depth;              // depth of the node within the tree
   Vector<string> values;  // stores up to depth keys in sorted order
   expnode **children;     // set to NULL until node is saturated.
};
```

- Each node must keep track of its **`depth`**, because the depth alone decides how many elements it can hold, and how many sub-trees it can support.

- The string values are stored in the **values** vector, which maintains all of the strings it's storing in sorted order. We use a **Vector<string>** instead of an exposed array, because the number of elements stored can vary from **0** to **depth**.
- **children** is a dynamically allocated array of pointers to sub-trees. The **children** pointer is maintained to be **NULL** until the **values** vector is full, at which point the **children** pointer is set to be a dynamically allocated array of **depth + 1** pointers, all initially set to **NULL**. Any future insertions that pass through the node will actually result in an insertion into one of **depth + 1** sub-trees.

a. [2 points] Draw the exponential tree that results from inserting the following strings in the specified left-to-right order:

"do" "re" "mi" "fa" "so" "la" "ti"



b. [6 points] Implement the **`expTreeContains`** predicate function, which given the root of an exponential tree and a string, returns **`true`** if and only if the supplied string is present somewhere in the tree, and **`false`** otherwise. Your function should only visit nodes that lead to the string of interest. Your implementation can rely on the implementation of **`find`**, which accepts a sorted string vector and a new string **`value`** and returns the smallest index within the vector where **`value`** can be inserted while maintaining sorted order. You may implement this either iteratively or recursively.

```
static int find(const Vector<string>& v, const string& value) {
   for (int pos = 0; pos < v.size(); pos++) {
      if (value <= v[pos]) {
         return pos;
      }
   }

   return v.size();
}


static bool expTreeContains(const expnode *root, const string& value) {
   const expnode *curr = root;
   while (curr != NULL) {
      int pos = find(curr->values, value);
      if (pos < curr->values.size() && curr->values[pos] == value) return true;
      curr = curr->children == NULL ? NULL : curr->children[pos];
   }
   return false;
}
```

Criteria for Problem 2b: 6 points

- Returns **`false`** on the empty tree, or the search eventually arrives at some other **`NULL`** pointer: 2 points (**`while`** loop test, or recursive base case)
- Correctly calls **`find`** (1 point) and:
  - Returns **`true`** if a match is found: 1 point
  - Identifies the index of the child that, if present, should be descended into: 1 point
- Correctly updates **`curr`**: 1 point

c. [8 points] Write the **`expTreeInsert`** function, which takes the root of an exponential tree [by reference] and the value to be inserted, and updates the tree to include the specified value, allocating and initializing new **`expnode`**s and arrays of **`expnode *`**s as needed. Ensure that you never extend a **`values`** vector beyond a length that matches the node's **`depth`**. Feel free to rely on **`find`** from part b. **You must implement this using iteration, without recursion.**

```
static expnode *createExpNode(int depth) {
   expnode *node = new expnode;
   node->depth = depth;
   node->children = NULL;
   return node;
}

static void allocateChildren(expnode *node) {
   node->children = new expnode *[node->depth + 1];
   for (int i = 0; i < node->depth + 1; i++) node->children[i] = NULL;
}

static void expTreeInsert(expnode *& root, const string& value) {
   int depth = 1;
   expnode **currp = &root;
   while (true) {
      if (*currp == NULL) *currp = createExpNode(depth);
      expnode *curr = *currp;
      assert(curr->values.size() <= depth);
      int pos = find(curr->values, value);
      if (curr->values.size() < depth) {
         curr->values.insert(pos, value);
         if (curr->values.size() == depth) allocateChildren(curr);
         return;
      }
      currp = &curr->children[pos];
      depth++;
   }
}
```

Criteria for Problem 2c: 8 points

- Introduces the notion of depth, and promotes **`depth`** with each iteration: 1 point
- Introduces my equivalent of **`currp`** and properly initializes it: 1 point
- Manages to update root to be non-**`NULL`** whenever it is **`NULL`**. Notice that I don't add the value immediately; instead, I continue and let the core of the implementation manage that part from me, knowing the vector isn't saturated. The allocation and the initialization of the depth and children fields absolutely has to happen when root is **`NULL`**: 1 point
- Properly calls find to **`find`** the insertion or descend point: 1 point
- Explicitly check to see if **`values`** is saturated, and if not, inserts the new key into the correct position: 1 point
- If **`values`** becomes saturated, immediately allocate **`children`** and set all **`expnode *`**s to **`NULL`**: 2 points
- Reframe the next iteration in terms of the correct **`expnode **`**: 1 point

d. [4 points] Finally, write the **expNodeDispose** function, which recursively disposes of the entire tree rooted at the specified address.

```
static void expTreeDispose(expnode *root) {
   if (root == NULL) return;
   if (root->children != NULL) {
      for (int i = 0; i < root->depth + 1; i++)
         expTreeDispose(root->children[i]);
      delete[] root->children;
   }
   delete root;
}
```

Criteria for Part 2d: 4 points

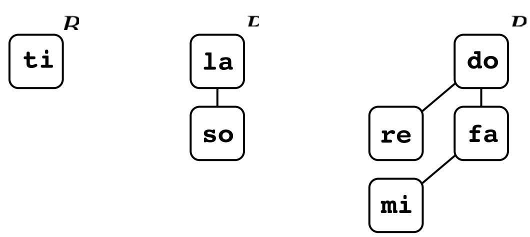
- Recursively calls **expTreeDispose** against all children: 1 point
- Is careful to only dereference the **root** and **root->children** if they're non-**NULL**: 1 point (0 points if there are any **NULL** pointer dereference issues)
- Properly disposes of the children array after all children have been disposed of: 1 point
- Disposes of the root node itself, and does so last: 1 point

Solution 3: Short Answer Questions [8 points]

Unless otherwise noted, your answers to the following questions should be 75 words or fewer. **Responses longer than the permitted length will receive 0 points.** You needn't write in complete sentences provided it's clear what you're saying. Full credit will only be given to the best of responses. Just because everything you write is true doesn't mean you get all the points.

a. [2 points] Assignment 5's **`BinomialHeapPQueue`** implementation was constrained to internally manage a vector of binomial heaps. Assume the strings **`"do"`**, **`"re"`**, **`"mi"`**, **`"fa"`**, **`"so"`**, **`"la"`**, and **`"ti"`** are inserted into an instance of a fully operational **`BinomialHeapPQueue`**. Draw a vector of valid binomial heaps that might contribute to the internal state of the priority queue. (Recall that **`extractMin`** removes and returns the lexicographically smallest string, so your binomial heaps need to be structured accordingly.)

The size of the priority queue would be 7, which is better viewed as 111 in binary. That means the priority queue would be backed by three binomial heaps of order 0, 1, and 2, occupying indices 0, 1, and 2 in a **`Vector<node *>`**. We'll accept trio of valid binomial heaps that stores the seven keys. (Only the picture is required. The narrative above is not.)



b. [2 points] Explain why Assignment 6's Huffman compression algorithm does a better job of compressing data when the Huffman encoding tree is unbalanced (i.e. far from structurally symmetrical).

You want asymmetric encoding trees, because that means that the most common character is embedded within the shallowest leaf node of the encoding tree, and the corresponding binary encoding is very short. That means some other characters have very long encodings, but presumably we don't care, because those characters occur much less frequently.

c. [2 points] We implemented the **Map** abstraction (**put**, **get**, **operator[]**, etc.) two different ways: one relied on hashing and hash tables, and a second relied on binary search trees. Describe one clear advantage that each implementation has over the other.

- Advantage of hash tables over BSTs: constant time insertion and lookup for hash tables, not the case with BSTs.
- Advantage of BSTs over hash tables: iteration presents keys in sorted order, hash tables do not.

d. [2 points] The **Lex** class we implemented in lecture relied on a trie for the internal representation, although we could have relied on a binary search tree instead. Describe one clear advantage of using tries instead of binary search trees and describe one advantage of using binary search trees over tries.

- Advantage of trie over BST: natural support for **containsPrefix**, and search time for **contains** and **containsPrefix** is proportional to the size of the string, not the size of the lexicon itself.
- Advantage of BST over trie: much lower memory footprint.

e. [0 points] What was your favorite assignment this quarter, and why?

The correct answer is Assignment 5 **PriorityQueue**. Hope that's what you wrote.