# CS106X Midterm Examination

This is an open-book, open-note, closed-electronic-device exam. You needn't write **`#include`**s, and you may (and you're even encouraged to) write helper functions to help manage complexity. You needn't write prototypes for your helper functions, and you can define the helper functions either before or after the function that calls them.

Good luck!

SUNet ID: _____**`@stanford.edu`**

Section Leader: _____

Last Name: _____

First Name: _____

I accept the letter and spirit of the honor code.

(signed) _____

| | | 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);
};
```

**Problem 1: `mergeLists` Memory Trace [7 points]**

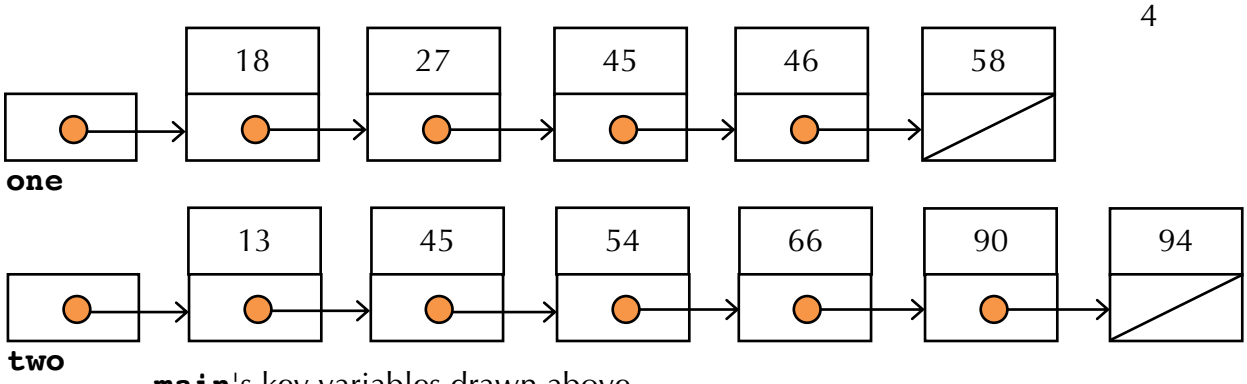
Consider the **`mergeLists`** presented below. Given two sorted linked lists of potentially different lengths, **`mergeLists`** cannibalizes the two incoming lists to construct a single one that is the sorted merge of the two originals.

```
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.



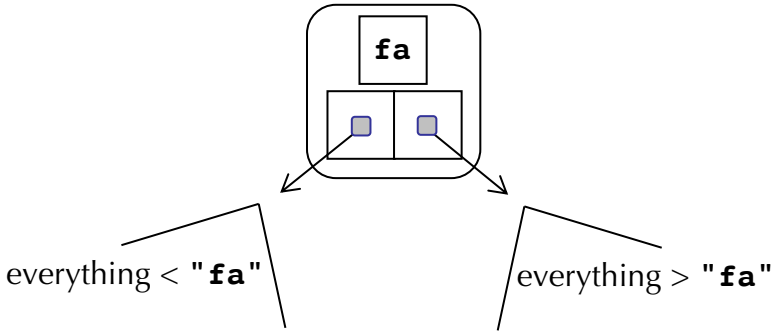

**`main`**'s key variables drawn above

---

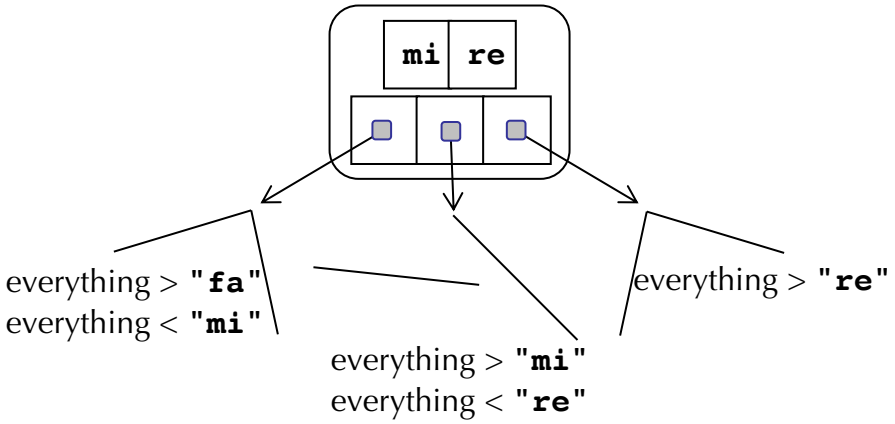
**`mergeLists`**'s parameters and variables **to be drawn below**

**Problem 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) {
```

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 void expTreeInsert(expnode *& root, const string& value) {
```

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) {
```

**Problem 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.)

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).

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.

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.

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