CS106B Practice Final Exam (previously: various qtrs.)

Name:	SUID (numeric):

Instructions:

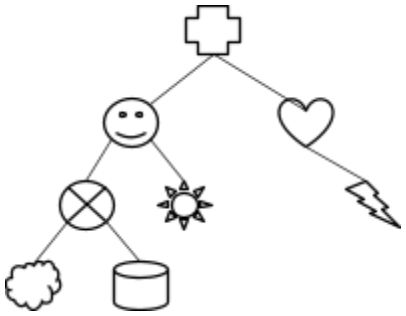
- **Stay within designated answer boxes**, or graders may not see your response on Gradescope. If you require extra space, write "continued" in the answer box, and then use designated scratch space somewhere in the exam. **No additional scratch paper.**
- **The time limit is strict.** When time is called, immediately drop your pencil/pen, and proceed to turn in your exam. No writing.
- For coding problems, **you may assume `#include` and function prototypes** already exist for you as needed.
- **Stanford Honor Code:** The Honor Code is an undertaking of the Stanford academic community, individually and collectively. Its purpose is to uphold a culture of academic honesty. Students will support this culture of academic honesty by neither giving nor accepting unpermitted academic aid on this examination.
- **Permitted and unpermitted aid:** You may have one double-sided, 8.5" × 11" page of notes. No other aids. **No electronics including laptops, phones, smart glasses/watches or earbuds within reach or sight.** Other than course staff, no consulting with other humans or AI in any way during the exam period.
- **Proctoring:** Proctors will be present in the exam room. If you have any questions or concerns during this exam, please speak with a proctor. At some point during the exam, proctors may take a few broad photos of the room, for later seating chart verification. Backpacks and phones must be left at the front of the room or with proctors. Proctors will verify your photo ID when you turn in your exam.
- If you need a **break, (e.g., bathroom)**, first give your exam to a proctor, who will log your break.

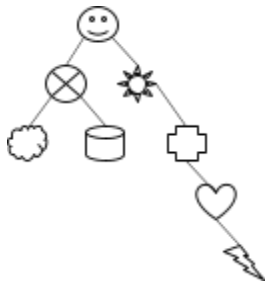
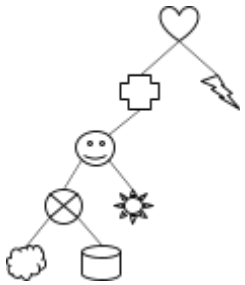
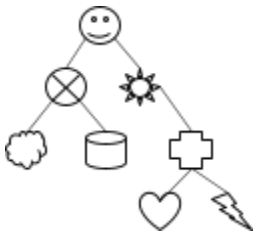
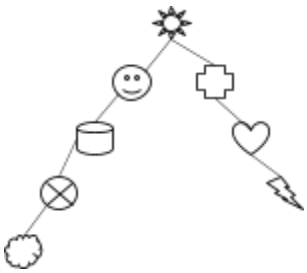
Please sign before you begin: I agree to abide by the spirit and letter of the Honor Code, and to follow the instructions above.

Signature:

1. Binary Search Trees (BSTs) (14 points)

(a) (8pts) The diagram at right represents a valid binary search tree, but we have **obscured the actual numeric keys** behind various shapes/symbols. So all you know about the keys are the greater- than and less-than relationships that you can infer from the given tree structure. For each of the following re-arranged trees, identify whether each is guaranteed to be a valid BST, guaranteed to not be a valid BST, or not enough information to say one way or the other. **Fill <u>one</u> circle for your choice under each tree.**



			
◯ Valid BST ◯ Not a valid BST ◯ Not enough information	◯ Valid BST ◯ Not a valid BST ◯ Not enough information	◯ Valid BST ◯ Not a valid BST ◯ Not enough information	◯ Valid BST ◯ Not a valid BST ◯ Not enough information

(b) (6pts) Below are lists of the numbers 1-6 shuffled three different ways. For each, indicate whether inserting keys into a BST-backed Map *in that order* would give a worst-case performance tree structure (use the usual BST algorithm with no rebalancing). **Fill <u>one</u> circle for your choice under each.**

6, 5, 4, 3, 2, 1	1, 6, 2, 5, 3, 4	4, 2, 5, 1, 6, 3
◯ Worst case structure ◯ Not worst case	◯ Worst case structure ◯ Not worst case	◯ Worst case structure ◯ Not worst case

(scratch space–end of problem 1)

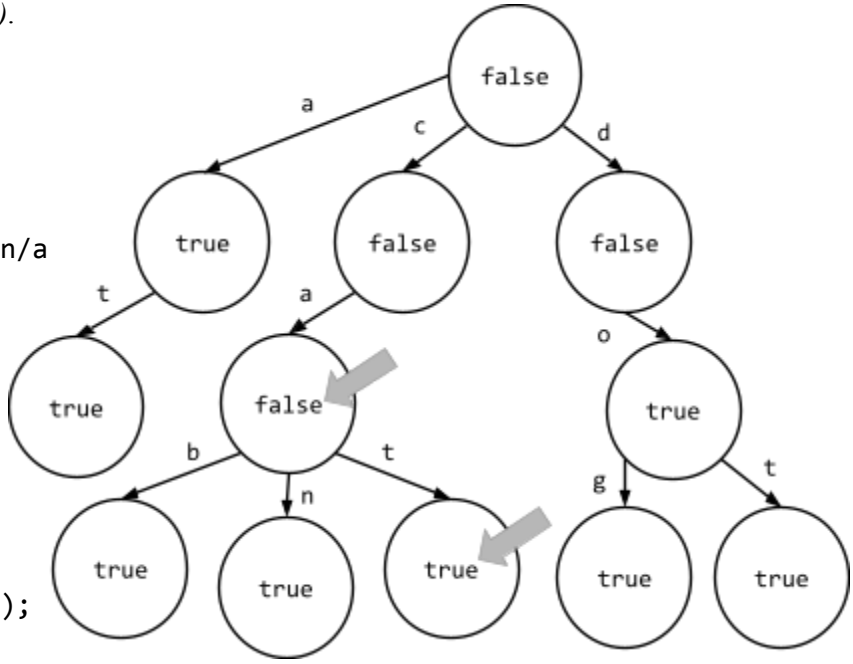
2. Classes: Implementing Lexicon (27 points)

One ADT in the Stanford Library whose implementation we didn't discuss was the `Lexicon`, which is basically equivalent in purpose to a `Set<string>`. Unlike the generic `Set` that has to be able to work with many different data types, `Lexicon` has behind the scenes optimization specifically for storing type `string`. That optimization is to use a special tree data structure where each node can have up to 26 children (!), one for each letter `a-z` (actual `Lexicon` can handle uppercase and other characters, but we will just do strings of `a-z`). Below is the `lexicon.h` file showing the planned interface *(you will not need to implement functions marked* `n/a` *on the exam, but imagine someone else is writing them)*.

```
class Lexicon {
public:
    Lexicon();
    ~Lexicon();
    bool contains(string word); // n/a
    void add(string word);
    int size(); // n/a

private:
    struct Node {
        bool  isWordEnd;
        Map<char, Node*> children;
    };
    void destructorHelper(Node* cur);

    Node* _root;   // root of the tree
    int   _size;   // number of words stored
};
```



The tree works as follows:

- Each word is represented as a tree `Node` where the `isWordEnd` field is `true`, and the letters of the word are the sequence of child pointers on the path from `_root` to that node.
- Since each node can have between 0 and 26 children, our `Node struct` avoids needing a separate field for each child pointer by using a Map where key:value pairs map characters to child `Node` pointers.
- We only store keys in a `children` Map for children that actually exist. So the size of the `children` Map for the root node in the diagram above is `3` (the keys are `'a'`, `'c'`, and `'d'`).
- **Examples:** We can see `"cat"` is stored in this tree by this process: (1) start at the `_root`, and look up `children['c']` to find a pointer to the `'c'` child and follow it, (2) go to that node's `'a'` child, and (3) go to that node's `'t'` child, and see that its `isWordEnd` field is `true`. We can see that `"ca"` is *not* a stored word, because the node we reach by following the path `_root` to `'c'` to `'a'` has `isWordEnd` set to `false`. *(See two grey arrows indicating these observations.)* This tree also stores both `"do"` and `"dog"`.

(a) (2pts) What should `_size` be for the tree shown above?

(b) (2pts) What is the **shortest** word stored in the tree above?

(c) (3pts) We begin with the **constructor**. Your tree should start by allocating one new `Node` to be the root. Your tree doesn't store any words yet, so you shouldn't add any keys to the root's `children` Map yet, and you should set the root's `isWordEnd` and the tree's `_size` accordingly for no words.

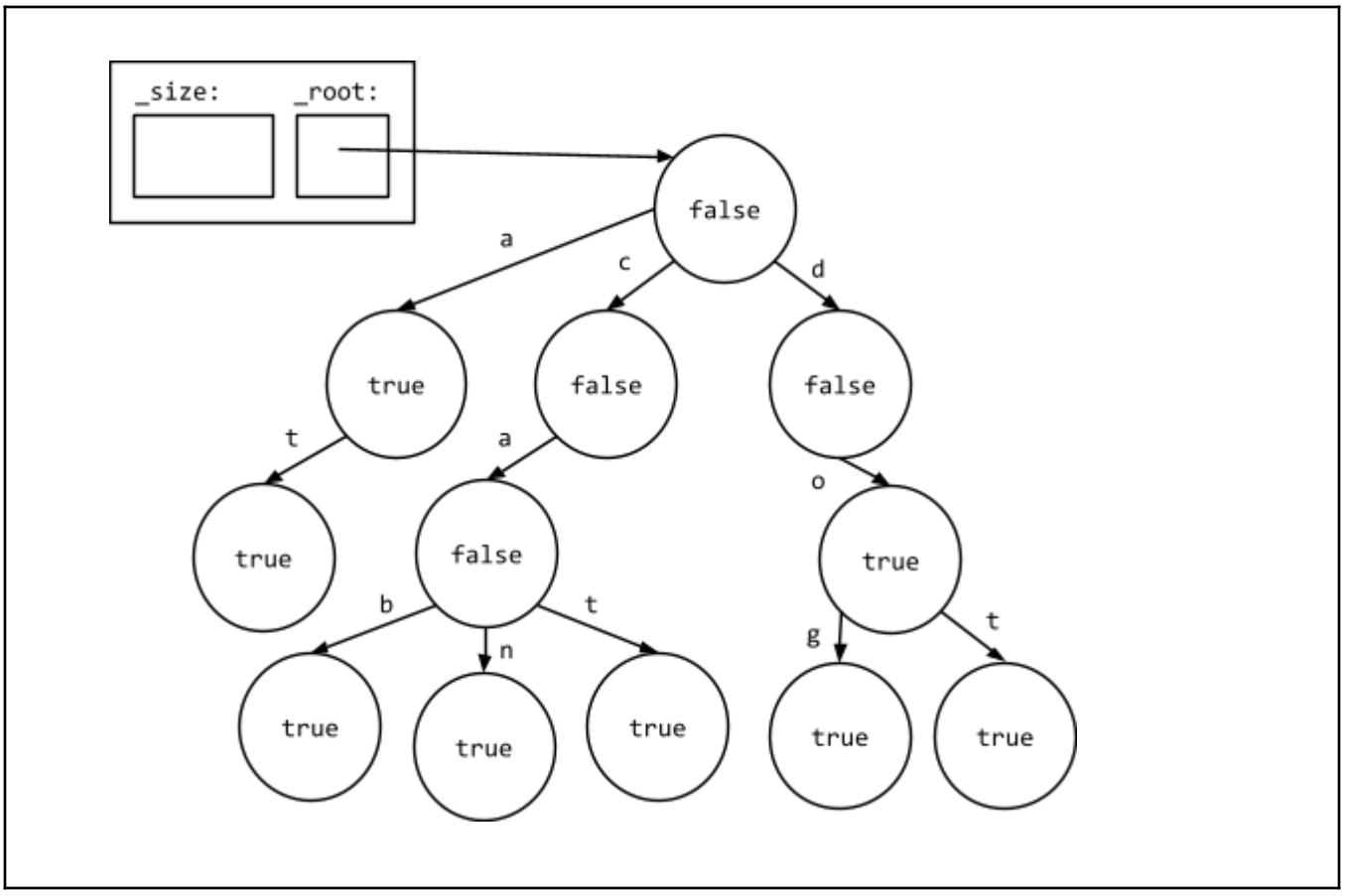
```
Lexicon::Lexicon() {



}
```

Now we will **add** words to our tree. This will initially follow a similar process to the `cat` example above. Start at `_root`, and follow the sequence of children dictated by each letter of the word. If, at some point along the path, you find that `children` does not contain the key that you need (the next letter of the word), so there is no next child pointer to follow, then you will need to allocate new `Node`(s) as needed to complete the path to the word. When you arrive at the node corresponding to the completed word, if its `isWordEnd` is `false` then set it to `true` and update `_size` *(if its* `isWordEnd` *is already true, we return without changing anything)*.

(d) (3pts) To prepare to implement `add()`, **update the drawing of this tree** and **fill in `_size`** to show the result after calling `add("dug")` and `add("cat")` on this object.



(e) (10pts) Now implement `add()`. Assume the `word` is a string of `a-z`. We recommend an iterative approach, but recursive is fine if you prefer. If recursive, feel free to make a recursive helper function *(you may assume any edits that might be needed to add the helper in .h are handled for you)*.

```
void Lexicon::add(string word) {









}
```

(f) (7pts) Finally, we implement our **destructor**. It should ensure that there are no memory leaks by deleting all dynamically allocated memory. Your solution should take a recursive tree traversal approach by **making a recursive helper function, as shown**.

```
// recursive helper function
void Lexicon::destructorHelper(Node* cur) {









}

// destructor
Lexicon::~Lexicon() {




}
```

(scratch space–end of problem 2)

3. Classes: Pointers and Linked Lists.

Write a function that, given two linked lists, will determine whether the second list is a subsequence of the first. To be a subsequence, every value of the second must appear within the first list and in the same order, but there may be additional values interspersed in the first list. By convention, we say that a list always contains itself and that the empty list is contained in any list. For the purposes of this problem, we will use the following definition:

```
struct LinkNode {
    int data;
    LinkNode* next;
};
```

Examples:

list	**sub**	**contains(list, sub)**
1 → 4 → 2 → 9	1 → 4	`true`
1 → 4 → 2 → 9	9 → 4	`false`
1 → 4 → 2 → 9	4 → 9	`true`
1 → 4 → 2 → 9	1 → 4 → 9	`true`
1 → 4 → 2 → 9	2 → 9 → 10	`false`

```
bool contains(LinkNode* list, LinkNode* sub) {









}
```

(end of problem 3)

4. Sorting

Show each step of **Selection Sort** on the vector of integers shown below. The selection sort code as it was shown in lecture is given below to refresh your memory (two print lines added for this problem):

```
void selectionSort(Vector<int>& vec) {
   cout << vec << endl; // initial contents of vector
   // already-fully-sorted section grows 1 at a time from left to right
   for (int lh = 0; lh < vec.size(); lh++) {
      int rh = lh;
      // find the min element in the entire unsorted section
      for (int i = lh + 1; i < n; i++) {
          // found new min?
          if (vec[i] < vec[rh]) rh = i;
      }
      // add min to end of sorted section
      int tmp = vec[lh];
      vec[lh] = vec[rh];
      vec[rh] = tmp;    // swap places
      cout << vec << endl; // your responses will give this output each iteration
   }
}
```

Below is the initial state of the vector of values to sort, as it would appear from the first `cout` above. You fill in the rest of the steps for each time another `cout` executes. *You may not need all the provided blanks to complete your solution.*

{5, 6, 8, 4, 2, 8, 3, 7, 1}
{ }
{ }
{ }
{ }
{ }
{ }
{ }
{ }
{ }
{ }

5. Big-O

Give the Big-O analysis as we learned in class (i.e., tight bound worst-case runtime complexity class) for each of the following code fragments, **in terms of the variable `N`**. As a reminder, when doing Big-O analysis, we write a simple expression, such as $O(N^2)$ or O(log N), *not a precise count*. Write your answer in the boxes on the right side. For each problem, the N that you use for your analysis is defined as the value of the variable `N` shown in the code. If running the code will make an infinite loop or stack overflow, **write a large X in the answer box**.

	Code Fragment	**Answer**

(a)	`void functionA(int N) {`
	`  return functionA(N - 1) + functionA(N - 1);`
	`}`
		O( )

(b)	`// Assume that str has nonzero length and starts out containing`
	`// no 'X' characters.`
	`void functionE(string& str) {`
	`  int N = str.length();`
	`  if (str[N - 1] == 'X') {`
	`  return;`
	`  }`
	`  for (int i = 0; i < N; i++) {`
	`  if (str[i] != 'X') {`
	`  str[i] = 'X';`
	`  break;`
	`  }`
	`  }`
	`  functionE(str);`
	`}`
		O( )

(scratch space–end of problem 5)

(scratch space–end of exam)