Practice Final 3

CS106B, Fall 2023

Last (**Family**) Name	
First (**Given**) Name	
Stanford E-mail	`@stanford.edu`

Exam Instructions

There are **5** questions worth a total of **95** points. Write all answers directly on the exam paper in the provided spaces for each question. Do not add or remove pages to this exam, and do not remove the staple. This printed exam is closed-book and closed-device; you may refer only to our provided reference sheet. You are required to write your SUID number in the blank at the top of each odd-numbered page.

Unless otherwise restricted in the instructors for a specific problem, you are free to use any of the CS106B libraries and classes. You don't need *#include* statements in your solutions; just assume the required header files (*vector.h*, *strlib.h*, etc.) are visible. You do not need to declare prototypes. You are free to create helper functions unless the problem states otherwise. Comments are not required, but when your code is incorrect, comments could clarify your intentions and help the graders award partial credit.

The Stanford University Honor Code (2023 Revision)

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 in any work that serves as a component of grading or evaluation, including assignments, examinations, and research.

Instructors will support this culture of academic honesty by providing clear guidance, both in their course syllabi and in response to student questions, on what constitutes permitted and unpermitted aid. Instructors will also not take unusual or unreasonable precautions to prevent academic dishonesty.

Students and instructors will also cultivate an environment conducive to academic integrity. While instructors alone set academic requirements, the Honor Code is a community undertaking that requires students and instructors to work together to ensure conditions that support academic integrity.

In signing below, I acknowledge, accept, and agree to abide by both the letter and the spirit of the Stanford Honor Code. I will not receive any unpermitted aid on this test, nor will I give any. I do not have any advance knowledge of what questions will be asked on this exam. My answers are my own work.

____

(**signature**) (required)

Problem 1: Short answer (15 points)

a) What is $\log_2$ of 1,000,000,000 ("one BILL-ionnnnnn") ?

b) We define a *pre-order word* as one whose letters can be arranged into a binary search tree of letter nodes such that a pre-order tree traversal spells out the word and an in-order tree traversal visits the letters in alphabetical order. The BST below demonstrates **"met"** is a pre-order word.

```
met
          m
        /   \
       e     t
```

If a word does not satisfy the pre-order property, it will not be possible to construct a BST whose pre-order traversal spells the word.

For each of the four words given below, either draw a BST that demonstrates it is a pre-order word or write "not possible" if no such BST exists. You will have to play with this problem (and perhaps develop a strategy as you go) to find which words are amenable to this task.

Reminder of alphabetical order: `a b c d e f g h i j k l m n o p q r s t u v w x y z`

`learn`

`hasty`

`flown`

`arches`

c) For the two questions below, use the following assumptions for the hash table implementation:

- Hash collisions are resolved using separate chaining
- An array of linked lists is used for the chains, the array has size `N`
- The hash table does not allow duplicate values
- The hash function has O(1) runtime

What is the **best-case** big-O runtime for inserting `N` distinct values into an initially empty hash table? Briefly justify your reasoning and describe an input that would be a best case.

What is the **worst-case** big-O runtime for inserting `N` distinct values into an initially empty hash table? Briefly justify your reasoning and describe an input that would be a worst case.

Problem 2: Classes (20 points)
The **RandomBag** class models a collection of names with an operation for random selection, such as might be used for a lottery. The bag capacity is set in the constructor. The **add** operation adds a name to the bag. The **choose** operation selects a name from the bag at random and removes it. Here is the public interface of the class:

```
class RandomBag {
  public:
    RandomBag(int capacity);
    ~RandomBag();

    bool isEmpty() const;
    string choose();
    void add(string name);
  private:
    ...
}
```

You are to write the complete **RandomBag** class, including the declaration of the private member variables and implementation of the constructor, destructor and member functions.

Design: A **RandomBag** holds a collection of names. The required internal representation is a dynamic array of **string**. The names in the array are in no particular order. Your design can include other member variables as needed for the operations. These additional variables must be of primitive type, not other ADTs or classes.

Constructor/destructor: The constructor creates an empty **RandomBag** of the desired capacity. You may assume the capacity is non-negative. The constructor performs all necessary allocation and initialization and the destructor performs any needed cleanup and deallocation.

Check if empty: **isEmpty()** reports whether the bag is empty. This operation must run in O(1) time.

Choose from bag: **choose()** selects a name from the bag and removes it. The chosen name is selected at random from those in the bag. An error is raised if the bag is empty. This operation must run in O(1) time.

Add to bag: **add(name)** adds a name to the bag. If the bag is already at capacity, a name is first discarded to make room for the new name. The name to be discarded is chosen at random from those in the bag. This operation must run in O(1) time.

Below is sample client use of **RandomBag**.

```
RandomBag bag(3);         // in bag: empty
bag.add("Sean");          // in bag: Sean
bag.add("Clinton");       // in bag: Sean + Clinton
cout << bag.choose();     // in bag: Clinton if Sean chosen (else vice versa)
cout << bag.choose();     // in bag: empty
bag.add("Julie");         // in bag: Julie
bag.add("Clinton");       // in bag: Julie + Clinton
bag.add("Julie");         // in bag: 2 Julie + Clinton
bag.add("Julie");         // in bag: either 3 Julie or 2 Julie + Clinton
```

Complete the **RandomBag** class implementation by declaring the private member variables.

```
class RandomBag {
  public:
    RandomBag(int capacity);
    ~RandomBag();

    bool isEmpty() const;
    string choose();
    void add(string name);

  private:
      // TODO: declare private member variables
```

Write the full implementation of the **`RandomBag`** class, including proper prototypes and bodies for all five operations: constructor, destructor, **`isEmpty`**, **`choose`**, and **`add`**.

Problem 3: Recursive backtracking (20 points)
A word *morphs* into another through a sequence of steps that replace one letter in the start word with the letter from the corresponding index in the destination word. The string formed at each step must be a valid English word. Here are two sample morph sequences:

`"serum"` to `"strut"`: `"serum" -> "strum" -> "strut"`

("serum" to "strum" changes e to t at index 1, "strum" to "strut" changes m to t at index 4)

`"read"` to `"boot"`: `"read" -> "bead" -> "beat" -> "boat" -> "boot"`

(the underlined letter indicates the change at each step)

You are to write the recursive function

```
bool canMorph(string start, string dest, Lexicon& lex)
```

The function `canMorph` takes three arguments: the start and destination words and a `Lexicon`. The function returns `true` if start can be morphed to destination through a sequence of valid words and `false` otherwise.

Here are some example calls and the expected result:

```
EXPECT( canMorph("serum", "strut", lex) );
EXPECT( canMorph("read", "boot", lex) );
EXPECT( !canMorph("quiz", "boat", lex) );
```

Specifications:

- Your algorithm must be **recursive** and must use **backtracking** to generate the result.
- For full-credit, a solution must **avoid needless inefficiency**. In particular,
  - it must stop at the first successful morph sequence found
  - it must prune exploration that cannot lead to a successful result
  - it must not copy or create any additional data structures (no vectors, arrays, lists, etc.)
- It is allowable to write a **helper function**, however, this problem can be cleanly solved without one. Any helper function is subject to these same constraints.
- You can assume the initial `start` and `dest` are **same length** and both **valid English** words.
- The function simply returns a Boolean result of whether a successful morph sequence was found. Do not print, store, or return the sequence.

```
bool canMorph(string start, string dest, Lexicon& lex)
```

Problem 4: Linked Lists (20 points)
Write the **extractStrand** function that extracts and returns a strand built from the nodes taken from the front of an input list. A strand is a sorted list that can be extended on either end.

The strand is gathered using the following process:

1. Detach the first node of input list. This node is the initial strand.
2. Consider the next node of input list and whether it can extend the strand.
   a. If value is <= first of strand, detach node and prepend to strand *or*
   b. If value is >= last of strand, detach node and append to strand
   c. If value cannot extend strand, stop here.
3. Repeat Step 2, until you stop or at the end of input list, whichever happens first.

The call **extractStrand(input)** updates its reference parameter to point to the remainder of the input list and returns the front of the strand.

Below are expected outcomes for some sample lists. You may find it helpful to trace/draw these lists on scratch paper to confirm your understanding.

input before call	**extractStrand(input)** returns	**input** after call (remainder)
2 → 5 → 8 → 4 → 9 → 1	2 → 5 → 8	4 → 9 → 1
6 → 1 → 3	1 → 6	3
7 → 4 → 9 → 4 → 3	3 → 4 → 4 → 7 → 9	nullptr
nullptr	nullptr	nullptr

Specifications:

- The function **extractStrand** operates by rewiring links between existing **ListNodes**. Do not change/swap **ListNode** data values.
- You may declare **ListNode*** pointer variables but must not allocate nor deallocate any **ListNodes** (no calls to **new** or **delete**).
- Both the strand and remainder lists should be properly null-terminated.
- You must not use additional data structures such as arrays, vectors, queues, etc.
- The function must run in at most **O(N)** time, where **N** is the length of the input list. It must make only a single traversal and stop at the first value which cannot extend the strand.

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

Please write your answer on the following page.

```
ListNode* extractStrand(ListNode*& input)
```

Problem 5: Trees (20 points)
Write the function **`deleteMin`** that finds the node in a binary search tree with the minimum value, removes that node from the tree, and returns the value.

Specifications:

- You should deallocate any nodes that are no longer used.
- The remaining values must form a valid binary search tree.
- Your solution must run in **`O(height)`** time where **`height`** is the tree height.
- You may assume the initial tree is non-empty (i.e. there will be a minimum value).
- It is your choice whether to solve using iteration or recursion.

```
struct BSTNode {
    int value;
    BSTNode *left, *right;
};
```

Please write your answer on the following page.

5a) `int` **`deleteMin`**`(BSTNode*& t)`

5b) Assume your implementation of **`deleteMin`** works correctly. Without making any edits to the code, the type of parameter **`t`** is changed from **`BSTNode*&`** to **`BSTNode*`**. Identify an input tree for which **`deleteMin`** no longer works correctly after this change. Briefly justify your answer.

This page intentionally left blank. You may use this space for scratch work.
It will not be graded unless you write a redirect from the original answer area to here.