CS106B Autumn 2022

Practice Final

_____

(Print name **legibly**)

_____

(SUID number)

Exam instructions: Write all answers directly on the exam. This printed exam is **closed-book** and **closed-device**; you may refer only to your one letter-sized page of prepared notes and the provided reference sheet. You are required to write your SUID number in the blank at the top of each odd numbered page.

C++ coding guidelines: Unless otherwise restricted in the instructions for a specific problem, you are free to use any of the CS106 libraries and classes. You don't need `#include` statements in your solutions, just assume the required `vector`, `strlib`, etc. header files 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 earn partial credit.

THE STANFORD UNIVERSITY HONOR CODE

A. The Honor Code is an undertaking of the students, individually and collectively:

(1) that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;

(2) that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.

B. The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid as far as practicable, academic procedures that create temptations to violate the Honor Code.

C. While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

I acknowledge and accept the Honor Code.

_____

(signature)

Problem 1: Recursive backtracking

There is much speculation about the matching process for assigning frosh roommates: is it by compatibility in sleep schedule? by taste in music? by height? We have no insider info, but that won't stop us from trying to code it up.

You are given a helper function **`int score(string, string)`** that takes two student names and returns the compatibility score of this match. A score of 1 indicates these students are a perfect match and 0 is an okay match. Although ResEd would love for all matches to be perfect, they will accept up to a certain number of okay matches.

This table shows the compatibility scores for six students:

	Ari	**Ben**	**Chi**	**Dag**	**Eva**	**Flo**
Ari		1	0	1	1	1
Ben	1		0	0	1	0
Chi	0	0		0	1	0
Dag	1	0	0		0	0
Eva	1	1	1	0		0
Flo	1	0	0	0	0	

Note: ***`score(Ari, Ben) == score(Ben, Ari)`*** *by symmetry. Calling* ***`score`*** *for student with self will raise an error.*

Finding an acceptable pairing of the students is an excellent use of recursive backtracking! To find a pairing for these six students with the goal to allow at most one okay match, you could match **`Ari`** with **`Ben`** (perfect), **`Chi`** with **`Eva`** (perfect), and **`Dag`** with **`Flo`** (okay). If asked to satisfy the more restrictive goal of zero okay matches, no acceptable pairing is possible.

Write the function **`canPair`** that uses backtracking to find whether an acceptable pairing can be achieved. The two parameters are a **`Vector`** of student names and **`maxOk`**, the number of okay matches allowed. An acceptable pairing has at most **`maxOk`** number of okay matches. The function returns **`true`** if an acceptable pairing was found and **`false`** otherwise.

Your code should obey the following constraints:

- Your overall approach **must use recursive backtracking** to explore possible pairings.
- You do not have to find the optimal pairing, just any acceptable pairing.
- The function result is simply true or false, to report whether an acceptable pairing was found. Do **not print or store** the pairings.
- For full-credit, a solution must **avoid needless inefficiency**. In particular, it must stop at the first acceptable pairing found, must prune exploration that cannot lead to an acceptable pairing, and must not re-explore previously examined options
- You can assume the vector contains an even number of students, each student name is unique, and the **`score`** function works correctly for all students named in the vector.
- 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.

```
bool canPair(Vector<string> students, int maxOk)
```

Problem 2: Classes

The **`CallHistory`** class tracks your phone's call log using a fixed-size list of recently received calls. Here is the public portion of the class declaration:

```
class CallHistory {
public:
    CallHistory(int capacity);  // construct empty history of capacity
    ~CallHistory();             // destructor
    void add(string caller);    // add received call
    string getRecent(int n);    // retrieve from history
    int getTotalCalls();        // total count of all received calls
```

Your job is to write the complete **`CallHistory`** class, including the declaration of the private member variables and full implementation of the constructor, destructor and member functions.

Design: All of the **`CallHistory`** operations **must run in O(1) time**. The required internal representation for the list of recent calls is a **dynamic array of strings**. You may include additional member variables as needed to support the operations. These variables must be of primitive type, not other ADTs or classes.

Constructor/destructor: The constructor creates an empty history of the requested capacity. The array is allocated to the proper size and all member variables are properly initialized. The destructor performs any needed cleanup and deallocation.

Add to history: **`add(name)`** updates the history to include this most recent call. Below is some sample client code. The comment on each line shows the internal array after this update.

```
CallHistory history(4);   // - | - | - | - (capacity 4, all unfilled)
history.add("Cynthia");   // Cynthia | -    | -     | -
history.add("Neel");      // Cynthia | Neel | -     | -
history.add("Julie");     // Cynthia | Neel | Julie | -
history.add("Neel");      // Cynthia | Neel | Julie | Neel
```

Note that the array capacity is set in the constructor and does not enlarge. When adding to an array that is already filled to capacity, a new entry replaces the oldest entry. In the code below, adding "Michael" overwrites "Cynthia" and adding "Karel" then overwrites "Neel".

```
history.add("Michael");   // Michael | Neel  | Julie | Neel
history.add("Karel");     // Michael | Karel | Julie | Neel
```

Retrieve from history: **`getRecent(n)`** returns an entry from the recent calls. The most recent entry is n of 0, n of 1 refers to the entry previous to the most recent, and so on. If n is not valid (i.e. not within the range of recent calls for this **`CallHistory`**), the function raises an error.

Total calls: **`getTotalCalls`**() returns the count of received calls over the entire lifetime of this **`CallHistory`**. Note that this is not the same as capacity. In the client code above, the capacity is 4, the total number of calls received thus far is 6.

Complete the **`CallHistory`** class declaration started below by filling in the private section.

```
class CallHistory {
public:
    CallHistory(int capacity);  // construct empty history of capacity
    ~CallHistory();             // destructor
    void add(string caller);    // add received call
    string getRecent(int n);    // retrieve from history
    int getTotalCalls();        // total count of all received calls
private:
```

Write the full implementation of the **`CallHistory`** class, including proper prototypes and bodies for all five operations: constructor, destructor, add, getRecent, and getTotalCalls.

Problem 3: Linked Lists

Write the **`transform`** function to modify a linked list following this process:

- Traverse the list from front to back.
- For each node **`cur`**, if the data value of **`cur`** is smaller than the data value of the current list front, move node **`cur`** to become the new front of the list.

The table below demonstrates the expected result for various lists. You may find it helpful to trace/diagram these examples on scratch paper to confirm your understanding.

`list`	**`list` after call `transform(list)`**
`5 -> 6 -> 3 -> 1 -> 2`	`1 -> 3 -> 5 -> 6 -> 2`
`2 -> 8 -> 4`	`2 -> 8 -> 4`
`9 -> 12 -> 3 -> 3`	`3 -> 9 -> 12 -> 3`
`nullptr`	`nullptr`

Your code should obey the following constraints:

- Your function must operate solely by rewiring links between existing **`ListNodes`**. You may create **`ListNode*`** pointer variables but must not create nor deallocate any **`ListNodes`** (no calls to **`new`** or **`delete`**). Do not replace/swap **`ListNode`** data values.
- You must not use additional data structures such as arrays, vectors, queues, etc.
- Your code must run in at most **`O(N)`** time, where **`N`** is the length of the list, and must make only a single traversal over the list.

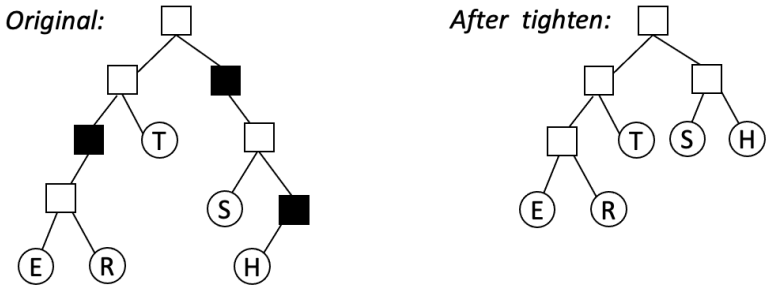
```
void transform(ListNode*& front)
```

Problem 4: Trees

In a properly constructed Huffman encoding tree, each node has either two or zero children. An improper node with only one child extends the encoding length of its subtree for no benefit. You are to write the **`tighten`** function that excises these improper nodes.

`tighten` operates by traversing the tree, identifying those nodes that are improper, and removing them. Any node with only one child is detached from the tree and deallocated; it is replaced in the tree with its child.

The diagram below shows a tree before and after **`tighten`**. Interior nodes are drawn as boxes; the three black boxes are the improper nodes that were removed.



Your code should obey the following constraints:

- You must not use new to allocate any **`Nodes`**. You should deallocate any nodes that are no longer used.
- Your solution must run in **`O(N)`** time where **`N`** is the number of nodes and can make only one traversal over the tree.

```
struct Node {
    char ch;
    Node *left, *right;
};

void tighten(Node*& t)
```

Problem 6: Short answer

Sorting. The Apple Mail team has received a bug report of a program crash when sorting mailboxes. The user says that sort mostly works correctly, but has observed an occasional crash.

The QA (quality assurance) engineer attempts to reproduce the behavior on a test mailbox of 1000 messages. Despite trying all variants of sorting, no problem surfaces. Next they try on a very large mailbox. They first sort by date newest to oldest (ok), then sort by size (ok), then sort by sender (ok), then back to sorting by date (ok), then reverse sort by date oldest to newest…. BOOM! Repeating this same sequence will crash during this last sort every time.

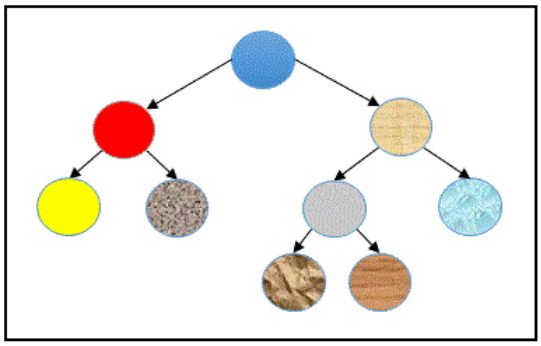
The sort routine being used is a correct implementation of Quicksort on linked lists that uses the first element as the pivot, just as you wrote for Assignment 6.

The QA team asks for your help. Explain the likely root cause and relate it to the observed behavior, specifically why it crashed on this particular sort operation for this mailbox and not in the other situations.

Binary heap You are adding a new member function **`changePriority`** to your **`PQHeap`** class from Assignment 5. A call to **`pq.changePriority(label, newpriority)`** finds an existing **`DataPoint`** in the priority queue with the specified label and changes its priority to the new value. Your goal is to make the **`changePriority`** operation work correctly with the **`PQHeap's`** existing design (binary min-heap stored in array), leveraging available operations where possible. Describe how you will implement the operation and give its Big-O running time.

Binary Search Trees.The diagram below is a valid binary search tree (BST). The exact keys are hidden, but the nodes have colors/textures so you can tell them apart.

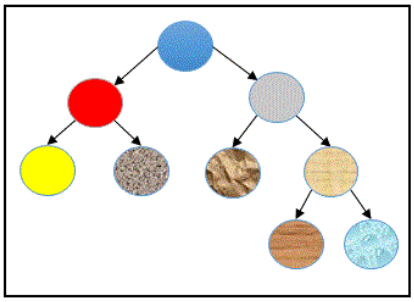
Original BST:



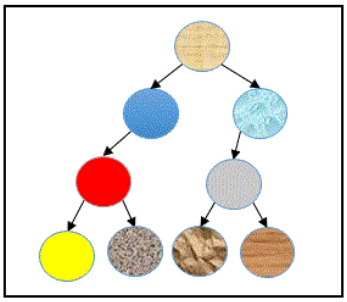
Recall that two BSTs containing the same keys can have different shapes depending on the order in which the keys were inserted.

The diagram below show three variant trees containing the same keys as the original BST. Which of these trees are also valid BSTs?

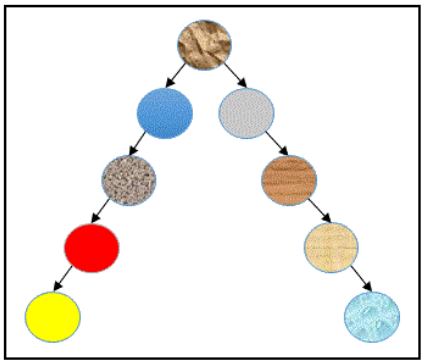
Variant 1:



Variant 2:



Variant 3:



For each variant, indicate whether it is also a valid BST and briefly justify your reasoning.