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

The game of Boggle is played using sixteen letter cubes. A letter cube is represented as a string of length 6, one character on each of the six cube faces. Given a target word and a collection of letter cubes, you want to spell the word using letters from the cubes, where each cube can be used at most once.

For example, given the letter cubes `{"etaoin", "shrdlu", "delwvy"}`, you can spell `"hey"` and `"law"` but neither "`weld`" nor "`toe`".

```
bool canSpell(string word, Vector<string>& cubes);
```

The function `canSpell` has two parameters: a word and a `Vector` of letter cubes. The function returns `true` if possible to spell the word from the cubes and `false` otherwise.

Specifications:

- Your overall approach **must use recursive backtracking**.
- For full-credit, a solution must **avoid needless inefficiency**. In particular
  - it must stop if the word has been spelled
  - it must prune exploration that cannot lead to a successful result
  - it must not re-explore previously examined options
  - it must not copy or create any data structures (no additional vectors, arrays, lists, etc. beyond the one vector passed by reference)
- 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.

Please write your answer on the following page.

```
bool canSpell(string word, Vector<string>& cubes)
```

This page intentionally left blank.
You may use this space for scratch work (it will not be graded).

Problem 2: Classes

The **`RangeSet`** class represents a set of integer values drawn from a fixed-size range. The min and max values of the range are configured in the constructor. Only values within the specified range can be added to the set.

```
class RangeSet {
public:
  RangeSet(int minValue, int maxValue);
  ~RangeSet();
  bool add(int value);
  int count() const;
private:
  int getIndex(int value) const;
```

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

Design: A **`RangeSet`** holds a set of values drawn from a specified range. The required internal representation is a dynamic array of **`bool`**. Each array index corresponds to a value within the range. The array element at an index is **`true`** if its corresponding value is contained in the set and **`false`** otherwise. 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 **`RangeSet`** over the range of values from min to max, **inclusive**. The range must include at least one value to be valid; otherwise the constructor raises an error. The constructor performs all necessary allocation and initialization and the destructor performs any needed cleanup and deallocation.

Determine array index for a given value: The member function **`getIndex`** is a private helper that returns the corresponding array index for a value or -1 if value is not within the range. For example, for the **`RangeSet(3, 6)`**, a call to **`getIndex(4)`** returns 1, **`getIndex(6)`** returns 3, and **`getIndex(2)`** returns -1.

Add a value: **`add(value)`** adds a value to the set. This operation must run in O(1) time.

- If the value is not within the range or is already contained in the set, the set is unchanged and returns **`false`**.
- If the value is within the range and not already contained in the set, the value is added to the set and returns **`true`**.
- Hint: **`getIndex`** can be useful when implementing **`add`**.

Return count of values: **`count()`** returns the count of values contained in the set. This operation must run in O(1) time.

Below is sample client use of **RangeSet**. The comments show the contents of the internal array after each line of code executes.

```
RangeSet rs(7, 11);           // [ false, false, false, false, false ]
cout << rs.count() << endl;   // prints 0
rs.add(8);                    // [ false, true,  false, false, false ]
rs.add(11);                   // [ false, true,  false, false, true  ]
cout << rs.count() << endl;   // prints 2
RangeSet badset(5, 2);        // Error!
```

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

```
class RangeSet {
public:
  RangeSet(int minValue, int maxValue); // Constructor
  ~RangeSet();                          // Destructor
  bool add(int value);                  // Add value to set
  int count() const;                    // Return count of values in set
private:
  int getIndex(int value) const;  // get array index corresponding to value
```

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

Write the full implementation of the **`RangeSet`** class, including proper prototypes and bodies for all five operations: constructor, destructor, **`add`**, **`count`**, and **`getIndex`**.

Problem 3: Linked Lists

In Assignment 6, you implemented the **`runSort`** algorithm which repeatedly split a single sorted run from the input list and combined it with the output list using binary merge. An alternate implementation of **`runSort`** instead combines all runs using *k-way merge*, a generalization of binary merge. Here is the algorithm:

Step 1) **Separate** (divide input into sorted runs)
- Split a sorted run from input list using provided **`splitRun`** helper function
- Enqueue run in priority queue with "priority" equal to the value of its front node
- Repeat until input list is empty, all runs now in the priority queue

Step 2) **K-way Merge** (combine runs into sorted output)
- Dequeue from priority queue to access the run with the smallest front value
- Detach front node of run and efficiently append node to output list
- If run remainder is non-empty, re-enqueue it with priority value of now-front node
- Repeat until all runs have been combined into output, priority queue now empty

The **`runSort`** function is given below.

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

void runSort(ListNode*& front) {
    PriorityQueue<ListNode*> runs;

    separate(front, runs);
    front = kWayMerge(runs);
}
```

You are to write the functions **`separate`** and **`kWayMerge`** that work as described in Steps 1 and 2.

Specifications:

- The functions **`separate`** and **`kWayMerge`** operate by rewiring links between existing **`ListNode`**s. Do not change/swap **`ListNode`** data values.
- You may declare **`ListNode*`** pointer variables but must not allocate nor deallocate any **`ListNode`**s (no calls to **`new`** or **`delete`**).
- Do not copy or create other data structures (no additional queues, vectors, arrays, etc. beyond the one priority queue declared in **`runSort`**).

```
// The splitRun helper detaches a sorted run from the front of the input
// list. It returns a pointer to the remainder of input list or nullptr if
// this was the final run.
// Example: given input {3 -> 5 -> 5 -> 4 -> 10 -> 4} splitRun detaches
// the run {3 -> 5 -> 5} and returns remainder list {4 -> 10 -> 4}.
// The runtime is O(len) where len is the length of the detached run.

ListNode* splitRun(ListNode* input);
```

a) Write the function **`separate`** to divide the list into sorted runs. (This is Step 1 on page 8). You should call the **`splitRun`** helper; the code is not shown but you can assume the function works as described above and is correctly implemented.

```
void separate(ListNode* front, PriorityQueue<ListNode*>& runs) {
```

b) Write the function **`kWayMerge`** to merge the sorted runs. (This is Step 2 on page 8).

```
ListNode *kWayMerge(PriorityQueue<ListNode*>& runs) {
```

c) What is the **best-case** big-O runtime of this version of **`runSort`** in terms of **`N`**, the length of the input list? Briefly justify your reasoning and identify which kinds of inputs are best-case.

d) What is the **worst-case** big-O runtime of this version of **`runSort`** in terms of **`N`**, the length of the input list? Briefly justify your reasoning and identify which kinds of inputs are worst-case.

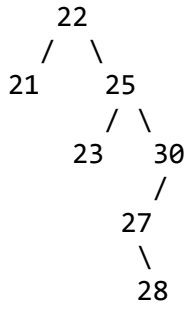
Problem 4: Trees

A tree is *balanced* if its left and right subtrees are balanced trees whose heights differ by at most one. A binary search tree must be balanced to achieve **`O(logN)`** runtime for insert and find. In lecture, we discussed self-adjusting trees such as AVL that make frequent minor rearrangements to maintain tree balance. An alternative approach is to watch and wait and if a significant problem develops, then perform a complete tree rebalance.

The **`rebalance`** function given below is an O(N) operation that dismantles an existing tree into a sorted vector of values and rebuilds a new balanced tree from the vector. You are to implement the two helper functions used in the rebalance operation: **`dismantle`** and **`rebuild`**.

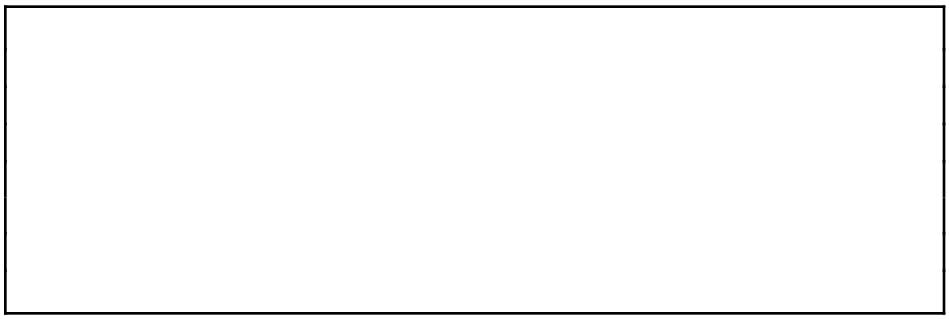
```
struct BSTNode {
   int value;
   BSTNode *left;
   BSTNode *right;
};

BSTNode* rebalance(BSTNode* t) {
   Vector<int> values;

   dismantle(t, values);    // fill vector with sorted values
   return rebuild(values);  // create balanced tree from vector
}
```

a) Trace the operation of **`rebalance`** on the tree below. Show the vector contents and draw the resulting balanced tree.



Contents of vector:

Resulting tree:



b) Write the helper function **`dismantle`** to place the values into a vector and deallocate the tree. The function arguments are a pointer to the root node of the tree and an empty vector. The function fills the vector with the values from an inorder tree traversal and deallocates all nodes. **`dismantle`** must traverse the tree **only once**.

```
void dismantle(BSTNode* t, Vector<int>& v)
```

c) Write the helper function **`rebuild`** to construct a new balanced binary search tree from the values in a sorted vector.

Specifications:

- The vector is in sorted order, which allows you to directly and efficiently build a balanced tree. Consider which vector index to use as the root value and which indexes would belong in the left subtree and which in the right subtree.
  Hint: recall how the binary search algorithm subdivides a vector into subranges.
- If possible to build more than one balanced tree, you are free to return any of them.
- Your function should not traverse any path in the tree more than once (i.e. you cannot rebuild the tree by repeatedly inserting values).
- Do not copy or create other data structures (no additional vectors, arrays, stack, etc. beyond the one vector passed by reference).
- You will need a helper/wrapper function.

```
BSTNode* rebuild(Vector<int>& v)
```