Practice Final 7

CS106B, Spring 2025

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

Exam Instructions

There are **5** questions worth a total of **105** 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)

Proposed Time Allocation

Question #1 (Classes and Dynamic Memory Management)	60 minutes
Question #2 (Backtracking with Linked Lists)	30 minutes
Question #3 (Tree Traversals)	10 minutes
Question #4 (Binary Tree Coding)	30 minutes
Question #5 (Hash Tables and Graph Algorithms)	10 minutes
Flex Time (Allocate as Needed)	40 minutes

Notes and Advice

1. This is a loose outline. Your time allocation might look very different from what we have above.
2. The times above are based on a 3-hour exam. If you have a different time limit, scale as needed.
3. Don’t panic if reading Question #1 takes some time. That’s normal. We have budgeted for that.
4. You might want to start by glancing through the entire exam to gauge how long you think each question will take.
5. Work diligently.

Preliminaries for Question #1

Before we dig into Question #1, we need to establish the following:

1. Definition of "Path String"

A "path string" is a string made up entirely of 'L' and 'R' characters. It defines a path where, starting from the root of some binary tree, we go left every time we see an 'L' character in our string and right every time we see an 'R' character. For example, consider the following tree:

```
      4
     / \
   18   15
  /    /
32    42
       \
       137
```

For this tree, the path string "RLR" gets us to node 137: from the root, we go **r**ight (to 15), then **l**eft (to 42), then **r**ight (to 137). Similarly, the string "LL" gets us to node 32, and the empty string ("") gets us to node 4 (the root node).

2. Array Representation for Binary Trees

Note that if we want to represent a binary tree using an array, we can use the same indexing scheme that we saw with the array-based representation of minheaps: the root is at index 0, the root's left child is at index 1, the root's right child is at index 2, the next level of nodes occupy cells 3, 4, 5, and 6, and so on. Using this convention, the array representation of the tree above is as follows:

4	18	15	32		42							137
0	1	2	3	4	5	6	7	8	9	10	11	12

Notice that – unlike a minheap – the above tree is not *complete*; there are gaps in the tree as we read each level from left to right. As a result, our array representation of this tree has unused (wasted) cells. The wasted space makes this a poor choice for representing arbitrary binary trees, but we'll use it throughout Question #1 anyway because it creates an interesting playground for us to explore various concepts covered in class this quarter.

3. Child Index Formulas

Given a node stored at index *i* in our array, we can find the index of its left child and right child using the following formulas:

$leftChild(i) = 2i + 1$

$rightChild(i) = 2i + 2$

1. Classes and Dynamic Memory Management (40 pts) In this problem, you will implement a class that uses a dynamically allocated array to store a binary tree. This class will use the same array indexing scheme that we saw with the array-based representation of minheaps, but it will not require our binary trees to be complete.

The class definition is below. Specifications for the functions you must implement are given on the pages that follow.

```
static const int kEmptyCell = -2147483648;
static const int kInitialCapacity = 10;

class BadTree {
public:
   BadTree();
   ~BadTree();

   // uses the path string to find the index where ‘value’ should be stored in our
   // array; expands the array if necessary; see following pages for more detail
   void set(string pathString, int value);

   // returns, in O(1) time, the # of nodes in the tree (# of occupied cells in _array)
   int size() const;

   // returns, in O(1) time, the # of unused cells in _array
   int wastedSpace() const;

private:
   // the dynamically allocated array containing our integer elements
   // fill in the box with the appropriate data type for this variable
   [                          ] _array;

   // expands _array to the given length; see following pages for more detail
   void expandArray(int newLength);

   // use this space to declare any other private variables you need; please follow
   // established conventions for any variable names; DO NOT ADD NEW MEMBER FUNCTIONS
    
};
```

(**Read these requirements carefully!**) Implement this bare-bones *BadTree* class:[1]

- The data type for *_array* (a dynamically allocated integer array) should go in the first box on the previous page.
- The declarations of any additional private member variables should go in the second box on the previous page.
- You may add new member **variables** to this class, but not new member **functions**.
- The full implementation of the member functions should go in the boxes on the following pages.
- Throughout this problem, you must take care to avoid memory leaks.
- You cannot use a node struct or other auxiliary data structures. This class must use an array to store its binary tree.
- You cannot use recursion in this problem. All your functions must be **iterative**.
- We discourage the use of helper functions in this problem beyond the required functions we have listed. Additional helper functions might make sense from a design perspective (e.g., readability, code re-use, and good decomposition practices), but avoiding them will actually help simplify a key aspect of the *set()* function that you have to write.

The *set()* function is described on the following page, but here are some example usages:

- `set("LL", 99)`
  For the tree below, this would change the 32 to a 99.
- `set("", 7)`
  For the tree below, this would change the value at the root node from 4 to 7.
- `set("LLR", 55)`
  For the tree below, this would give 32 a new right child containing the value 55. Note that this new node ends up at index 8 in the array, which is a valid index, and so the array does **not** have to expand.
- `set("RLRR", 75)`
  For the tree below, this would give 137 a new right child containing the value 75. Note that this new node ends up at index 26 in the array, and so the array **must expand** to length 27 to accommodate this new element.
- `set("RRR", 10)`
  For the tree below, this would throw an error since there is no node at "RR", and so it cannot be given a right child.

```
        4
       / \
     18   15
    /    /
  32    42
         \
          137
```

4	18	15	32		42							137
0	1	2	3	4	5	6	7	8	9	10	11	12

1 This is a "bad tree" because the array could potentially have a LOT of wasted space. It's also a bad tree because we can only add elements to it; it doesn't have functions to search for elements or even retrieve the value at a given node.

Here are the specific functions you must implement:

- `BadTree()`

  Constructor function. Dynamically allocate an integer array whose length is *kInitialCapacity*. Fill all the cells with *kEmptyCell*. Beyond that, be sure to initialize any private member variables you created to ensure the rest of the functions you write in this problem can work as intended.

- `~BadTree()`

  Destructor function. Perform any necessary tasks to ensure we have no memory leaks when a *BadTree* object is deleted or goes out of scope.

- `void set(string pathString, int value);`

  Compute the index that *pathString* leads to, and place *value* at that index in the array. If the index is out of bounds for our current array, call *expandArray()* to expand the array so that it is just long enough for this new element to fit. The *pathString* can land at an empty node – as with the *set("LLR", 55)* example on the previous page. However, if it tries to *skip over* an empty node – as with the *set("RRR", 10)* example on the previous page – throw an error using the *error()* function (and you must find a way to ensure that your function does not call *expandArray()* in this case). Be sure this function updates any private member variables as needed, and be careful that you never access an out-of-bounds array index in this function. You may assume *pathString* contains only 'L' and 'R' characters (no need to worry about it containing other characters).

- `int size() const;`

  In O(1) time, return the number of nodes in the *BadTree* (i.e., the number of occupied cells in *_array*).

- `int wastedSpace() const;`

  In O(1) time, return the number of unused cells in *_array*.

- `void expandArray(int newLength);`

  Expand the integer array so that its new length is *newLength*. Specifically:

  - Dynamically allocate a new integer array whose length is *newLength*.
  - Copy elements from the old array into this new array.
  - Ensure that *_array* points to the new array and that the function has no memory leaks.
  - Fill any unused cells in the new array with *kEmptyCell*.
  - If *newLength* is negative, zero, or not larger than the length of the existing *_array*, use *error()* to throw an error.

Write your implementations of the BadTree member functions on the following pages.

Write your implementation of the *BadTree* class member functions on this and the following two pages.

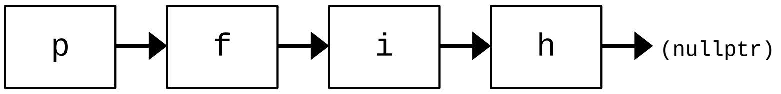
Write your implementation of the *BadTree* class member functions on this and the two surrounding pages.

Write your implementation of the *BadTree* class member functions on this and the previous two pages.

2. Backtracking with Linked Lists (20 pts)

Write a recursive backtracking function that takes the head of a linked list of characters (*head*) and an integer (*k*) and produces all valid English words that can be created by either adding or subtracting *k* from each character in the linked list. The function should add all such strings to a set of strings that is passed to the function by reference (*results*), and each call should return the number of valid English words it found down that branch of recursive calls. You can assume the function receives a lexicon (*lexy*) that contains a list of valid English words.

For example, suppose we're given the following linked list:



If we pass this list to our function with *k* = 3, our *results* set should ultimately end up with the following strings:

{"mile", "milk", "silk"}

Notice that "milk" is created by subtracting 3 from 'p', adding 3 to 'f', adding 3 to 'i', and adding 3 to 'h'. Notice that we are allowed to add *k* to some of the characters and subtract *k* from others, but we are not allowed to leave a character unmodified or change it by any amount other than exactly *k*. Notice also that all the words we generate must use the letters from *all* nodes in the linked list.

In solving this problem, you must abide by the following guidelines and restrictions:

- Your algorithm must be **recursive** and must use **backtracking** techniques to generate its results.
- In particular, you should only explore strings that could potentially lead to valid results. As soon as it becomes clear that a string you have generated cannot lead to any valid results, stop exploring that dead-end path.
- You must not create any instances of auxiliary data structures!
- Do not modify the contents or structure of the linked list.
- We have written a wrapper function for you. You only need to write the recursive helper function.
- You cannot modify the function signatures we have provided.
- You must do all your work in a single recursive function. Do not write additional helper functions.
- You may assume your function is given a lexicon with all valid English words, all in lowercase.
- Recall that you can add and subtract integers from characters (e.g., `'p' – 3` will give you the character `'m'`).
- You may assume every node in the linked list you receive contains a single lowercase alphabetic character.
- Do not worry about what happens if *k* is too large. Assume that that if we add or subtract a large value of *k*, we will still get some sort of character to add to our string, and if it's non-alphabetic, the lexicon will take care of rejecting the resulting string.

```
// This is the wrapper function that will call
// your recursive helper function. Do not modify.
void perturb(Node *head, int k) {
    Set<string> results;
    Lexicon lexy("EnglishWords.txt");

    int count = perturb(head, k, "", lexy, results);

    cout << "Found " << count << " result(s)." << endl;
    for (string s : results) {
         cout << " - " << s << endl;
    }
}
```

```
// This is the node struct
// for this problem.
struct Node {
    char data;
    Node *next;
};
```

```
int perturb(Node *head, int k, string soFar, Lexicon& lexy, Set<string>& results) {
```

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page.

Your code for Question #2 should fit in the box on the previous page.

3. Tree Traversals (10 pts)

We've seen *preorder*, *postorder*, and *inorder* traversals, which process all the nodes/subtrees in a binary tree in the following orders:

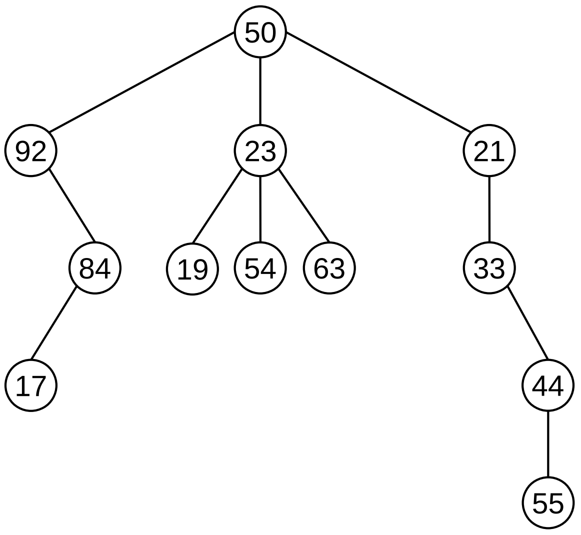
- Preorder: ROOT – LEFT – RIGHT
- Postorder: LEFT – RIGHT – ROOT
- Inorder: LEFT – ROOT – RIGHT

In this problem, we introduce a traversal algorithm that operates on **ternary** trees. A ternary tree is a tree where our nodes have three child pointers, labeled *left*, *middle*, and *right*. The node struct is as follows:

```
struct Node
{
    int data;
    Node *left;
    Node *middle;
    Node *right;
};
```

Let's define a new traversal algorithm called "wacky" that processes all the nodes/subtrees in a ternary tree in the following order:

- Wacky: MIDDLE – ROOT – RIGHT – LEFT



In this box, give the wacky traversal for the tree above. Note that the angle of each edge in this tree conveys whether a node is a left, middle, or right child.

Made a mistake in the box above? Cross is out and use this box instead!

4. Binary Tree Coding (20 pts)

Write a **<u>recursive</u>** function that takes the root of a binary tree (*root*), an integer that is set to the value in the original root of the tree (*target*), and a reference to a set of node pointers (*results*). The function must break up the tree so that every node that contains the *target* value (which is guaranteed to be equal to the tree's original root value) is disconnected and becomes the root of its own separate tree; the function must also add the roots of all the resulting trees to the *results* set.

For example, consider the following tree:

```
         27
       /    \
     15      13
    /  \       \
  27    50     22
 /  \         /  \
27  10       27   19
 \            \    \
 12            5   27
```

Given that the tree contains 27 at its root, we split off any subtrees rooted at 27. This yields five separate trees:

```
    27           27           27           27           27
  /    \          \            \            \
15     13         10           12             5
  \      \
  50     22
           \
           19
```

When the function is finished, the *results* set should contain the roots of those five separate trees.

To be clear, you can't just add those five node pointers to the set; you must also break the links so that these become five *separate* trees (e.g., the left child of 15 becomes a null pointer; it shouldn't still point to the 27 that serves as its left child in the original diagram).

(**Read these requirements carefully!**) Your solution to this problem must meet the following criteria:

- Your function must be **<u>recursive</u>**.
- Do not allocate any new nodes or deallocate any nodes!
- Be careful not to dereference any null pointers!
- Do not edit the *data* field of any nodes.
- We have written a wrapper function for you. You only need to write the recursive helper function.
- You cannot modify the function signatures we have provided.
- You must do all your work in a single recursive function. Do not write additional helper functions.
- You should not add any null pointers to the *results* set.
- Your solution should traverse the tree exactly once. Do not traverse the tree multiple times.

Hint: When you land on a node that needs to become the root of its own tree, it's too late to disconnect it from its parent node. For example, when you encounter a node like 15 in the tree above, you probably want to peek ahead and see that it needs to be detached from its left child.

```
// This is the wrapper function that will call
// your recursive helper function. Do not modify.
Set<Node*> treeCut(Node *root) {
    Set<Node*> results;
    if (root == nullptr) return results;
    treeCut(root, root->data, results);
    return results;
}
```

```
// This is the node struct for this problem.
struct Node {
    int data;
    Node *left;
    Node *right;
};
```

```
// ‘target’ is specified by the wrapper function and should not change as you make recursive calls.
void treeCut(Node *root, int target, Set<Node*>& results) {
```

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page.

Your code for Question #4 should fit in the box on the previous page.

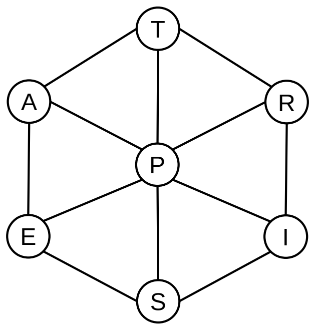
5. Hash Tables and Graph Algorithms (15 pts)

a) Add the characters to the hash table below, one by one in the order given, using linear probing and the given hash codes. Each cell is designed to hold a single character, and all 12 cells will be occupied in the end. You may assume our hash table allows us to insert the same character multiple times (i.e., it doesn't prevent the insertion of duplicates).

‘O’ (hash code: 9)
‘D’ (hash code: 8)
‘D’ (hash code: 8)
‘C’ (hash code: 4)
‘A’ (hash code: 4)
‘R’ (hash code: 0)
‘–’ (hash code: 7)
‘O’ (hash code: 9)
‘L’ (hash code: 4)
‘I’ (hash code: 3)
‘A’ (hash code: 4)
‘D’ (hash code: 8)

0	1	2	3	4	5	6	7	8	9	10	11

b) Indicate whether each of the following is a valid BFS for the given graph:



________ P A R T I E S
(yes/no)

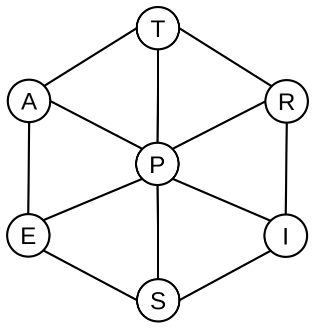
________ P I R A T E S
(yes/no)

________ P A S T I E R
(yes/no)

________ T R A I P S E
(yes/no)

________ T R A P I E S
(yes/no)

c) Indicate whether each of the following is a valid DFS for the given graph (same graph as above):



________ P A R T I E S
(yes/no)

________ T A P R I S E
(yes/no)

________ T R A P I S E
(yes/no)

________ T R I P S E A
(yes/no)

________ T R I P E A S
(yes/no)

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page.