Practice Final 3

CS106B, Summer 2024

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

Exam Instructions

There are **6** questions worth a total of **110** 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)

1. Recursive Backtracking (20 pts)

Before we dig into this question, let's define the "distance" between two lowercase alphabetic characters as how many letters apart they are from one another in the alphabet (without ever wrapping around from the end of the alphabet back to the beginning). So, for example, the distance between 'a' and 'a' is 0, the distance between 'a' and 'b' is 1, the distance between 'a' and 'c' is 2, the distance between 'a' and 'z' is 25, and so on. Note that the distance between two letters is always non-negative, and the order of the arguments does not matter. So, the distance between 'z' and 'a' is equal to the distance between 'a' and 'z', which is 25.

The goal of this problem is to write a backtracking function that prints all strings that meet the following conditions:

- The string is a valid English word that is made up entirely of lowercase alphabetic characters (no uppercase or non-alphabetic characters).
- No two consecutive characters in the string have a distance greater than some maximum that will be given as a parameter to your function (*consecDistMax*).
- The sum of the distances between all consecutive characters in the string does not exceed some separate maximum that will be given to your function as another parameter (*totalDistMax*).

For example, if we called this function with *consecDistMax* = 3 and *totalDistMax* = 4, some of the words we would print are given below. The following diagrams show the distance between all consecutive characters in these strings (none of which exceed 3), as well as the total sum of all those distances for each string (none of which exceed 4).

```
h   i   g   h
  1   2   1     (total: 1 + 2 + 1 = 4)
```

```
e   g   g
  2   0     (total: 2 + 0 = 2)
```

```
l   o   o   p
  3   0   1     (total: 3 + 0 + 1 = 4)
```

```
a
(total: 0)
```

In contrast (again with *consecDistMax* = 3 and *totalDistMax* = 4), we would **not** print the following. One has a pair of consecutive characters whose distance exceeds 3. The other has a total sum of distances that exceeds 4.

```
s   o
  4       (total: 4)
```

```
t   u   t   u   s
  1   1   1   2       (total: 5)
```

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.
- You must not create any instances of auxiliary data structures!
- You will need to write a helper function.
- You may assume your function is given a lexicon with all valid English words. See function signature below.

- Aside from the lexicon, please pass all parameters by **value** to help keep your solution clean and tidy.
- 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.
  - ***Hint:** There are at least two situations that should cause us to give up early and stop making recursive calls without having found a valid solution on a given path.*
- You should find a way to avoid unnecessarily looping through an entire string with each function call.
- To help keep distance calculations clean, you may assume the existence of an absolute value function, *abs()*, that takes a single integer argument and returns its absolute value. For example, *abs*(12) and *abs*(-12) both return 12.

```
void printAccordionWords(int consecDistMax, int totalDistMax, Lexicon& lexy) {
```

2. Classes (20 pts) In this problem, you'll develop a vector-based data structure that supports insertion and search operations, and which switches between two modes depending on which operation is performed more frequently:

- **Insertion-heavy mode.** We enter this mode when our client has performed more insertions than search operations on this data structure (or if the two are equally frequent). In this mode, we optimize for efficient, O(1) insertions at the sacrifice of our search runtime: we don't worry about keeping our vector sorted when inserting new elements, but that means our search function must perform a slow, expensive linear search.
- **Search-heavy mode.** We enter this mode when our client has performed more search operations than insertions. In this mode, our insertion algorithm ensures our vector is sorted at all times, which results in worst-case insertion runtimes of O(n), but we gain from that the ability to perform fast binary search operations.

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

```
class SearchSack {
public:
  SearchSack();
  void add(int elt);
  bool contains(int elt);

private:
   // the vector containing our elements
   Vector<int> _v;

   // adds element to vector; assumes vector is already sorted; maintains sorted order
   void sortedAdd(int elt);

   // adds element to vector in O(1) time with no regard for sortedness
   void unsortedAdd(int elt);

   // returns index of given element if found, -1 otherwise
   int linearSearch(int elt);

   // assumes vector is sorted; returns index of given element if found, -1 otherwise
   int binarySearch(int elt);

   // assumes vector is sorted; returns index where element should be inserted in order
   // to preserve sortedness of vector
   int getInsertionIndex(int elt);

   // use this space to declare any other private variables you need

};
```

Implement this bare-bones *SearchSack* class:

- The declaration of any additional private member variables should go in the box on the previous page.
- The full implementation of the member functions should go on the following two pages.

We enter into "search-heavy mode" if the number of search operations we perform ever exceeds the number of insertions, tracked over the entire lifespan of the data structure. Otherwise, we should be in "insertion-heavy mode."

Here are the specific functions you must implement:

- `SearchSack()`

  Constructor function. Simply initialize any member variables as appropriate to ensure the rest of the functions you write in this problem work as intended.

- **`void`** `add(`**`int`** `elt)`

  First, perform any bookkeeping necessary to help keep track of whether we're in insertion-heavy or search-heavy mode. Then call either *sortedAdd()* or *unsortedAdd()*, depending which mode we're in.

- **`bool`** `contains(`**`int`** `elt)`

  First, perform any bookkeeping necessary to help keep track of whether we're in insertion-heavy or search-heavy mode. If this search operation causes us to flip from insertion-heavy to search-heavy mode, sort the vector immediately. Next, call either *linearSearch()* or *binarySearch()*, depending which mode we're in. You may assume *linearSearch()* and *binarySearch()* are already implemented within this class and return integers as described above.

- **`void`** `unsortedAdd(`**`int`** `elt)`

  Insert the given element into the vector in O(1) time with no regard for sortedness.

- **`void`** `sortedAdd(`**`int`** `elt)`

  Insert the given element into the vector in its sorted position. You may assume the vector is already sorted any time we call this function. You may also assume the *getInsertionIndex(int elt)* function described on pg. 4 is already implemented within this class and will give the exact index where this element needs to be inserted in order to maintain sortedness of the vector. Any elements at or after that index must move over to make room for the new one, and you need to be sure to do that in a way that does not attempt to access any out-of-bounds vector indices.

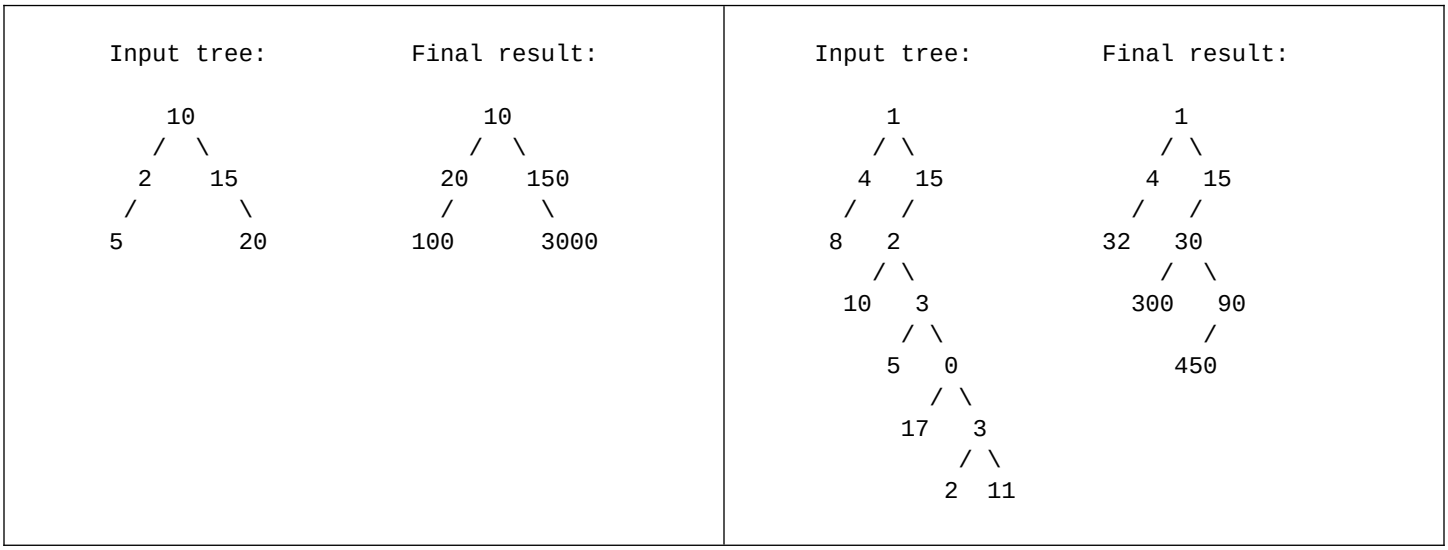
Write your implementations of the SearchSack
member functions on the following pages.

Write your implementation of the *SearchSack* class member functions on this and the following page.

Write your implementation of the *SearchSack* class member functions on this and the previous page.

3. Binary Trees (20 pts)

Write a function called *propagate()* that takes the root of a binary tree and modifies all nodes in that tree so that their values are multiplied by the products of all the values in their ancestor nodes, but removes any nodes that would end up having a value of zero as the result of that operation. For example:



Some restrictions:

- Your algorithm must be recursive.
- Do not create any instances of auxiliary data structures!
- You will need to write at least one helper function (and a second one might help make this easier to solve).
- Your solution should traverse the tree exactly once. Do not traverse the tree multiple times.
- Any nodes you remove from the tree should be deleted to avoid memory leaks.
- Do not create any new nodes or rearrange the nodes that are not being deleted.

The *Node* struct and function signature are as follows:

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

void propagate(Node*& root);
```

Place your answer on the following page.

```
void propagate(Node*& root) {
```

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

We will not grade anything on this page.

4. Linked Lists (20 pts)

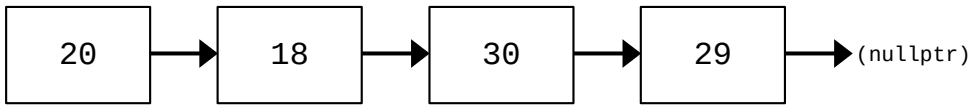
a) Draw a linked list with **at least three nodes** (but no more than **six nodes**) that would cause a segmentation fault if we were to pass its head to the following function. Your linked list must be valid and well formed and cause a crash because of the faulty logic in the following function, not because the linked list itself is somehow busted, invalid, or circular.

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

int foo(Node *head)
{
  if (head == nullptr)
  {
    return 106;
  }
  else if (head->data > 106 || head->data <= 0)
  {
    return head->data * 107 + head->data;
  }
  else if (head->data % 10 == 5)
  {
     return head->data * foo(head->next->next);
  }
  else
  {
    return foo(head->next) + head->data * 109;
  }
}
```

b) Consider the following code, and then answer the questions on the following page.

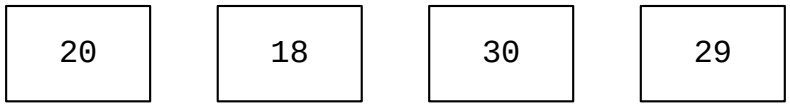
```
struct Node {
  int data;
  Node *next;

  Node(int data) {
    this->data = data;
    this->next = nullptr;
  }
};

void mystery(Node *head) {
  Stack<Node*> nodez1;
  Stack<Node*> nodez2;

  Node *temp1 = head;
  Node *temp2 = nullptr;

  while (temp1 != nullptr) {
    nodez1.push(temp1);
    nodez2.push(temp2);

    temp2 = temp1;
    temp1 = temp1->next;
  }

  Node *newHead = nullptr;
  Node *newTail = nullptr;

  while (!nodez1.isEmpty()) {
    newTail = nodez1.pop();
    newTail->next = nodez2.pop();

    if (newHead == nullptr) {
      newHead = newTail;
    }
  }

  head = newHead;

  // *** CHECKPOINT ONE ***
}

int main() {
  Node *head = new Node(20);
  head->next = new Node(18);
  head->next->next = new Node(30);
  head->next->next->next = new Node(29);

  mystery(head);

  // *** CHECKPOINT TWO ***

  return 0;
}
```

i) The linked list we create in *main()* initially looks like this:



Draw what the linked list in our *mystery()* function looks like when we hit the line labeled "**`*** CHECKPOINT ONE ***`**" (after setting *head* = *newHead*, but before returning from the *mystery()* function):

ii) Consider the state of our program when we hit the line labeled "**`*** CHECKPOINT TWO ***`**" (after returning from the *mystery()* function, but before we return from *main()*). Indicate whether each of the following nodes from our original linked list has been orphaned at that point (i.e., whether it has become unreachable) or not. Fill in the blank beneath each of the nodes below with either "yes" (the node is orphaned) or "no" (not orphaned).

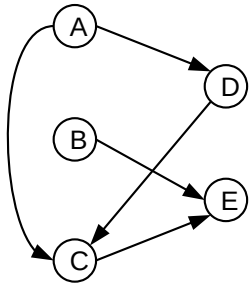


_____ _____ _____ _____ ← orphaned?

yes/no yes/no yes/no yes/no

5. Graphs (20 pts)

a) Indicate whether each of the following is a valid topological sort for the given graph:



_____ B A D E C (yes/no)

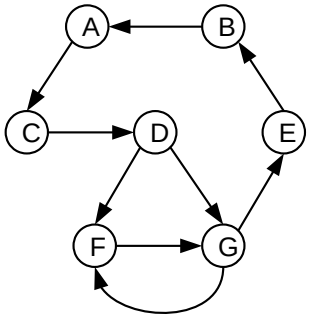
_____ A D C B E (yes/no)

_____ A B C D E (yes/no)

_____ A D B C E (yes/no)

_____ B A C D E (yes/no)

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



_____ D G E F B A C (yes/no)

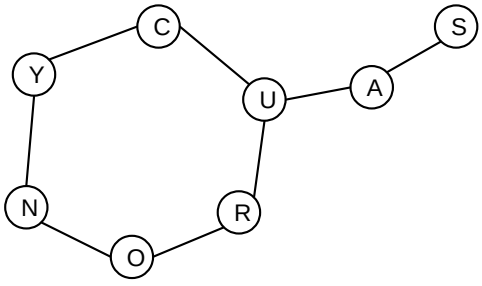
_____ D G F E B A C (yes/no)

_____ D F G E B A C (yes/no)

_____ F D G E B A C (yes/no)

_____ F G E B A C D (yes/no)

c) In the box, list all valid DFS traversals starting at node S in the graph to the right.

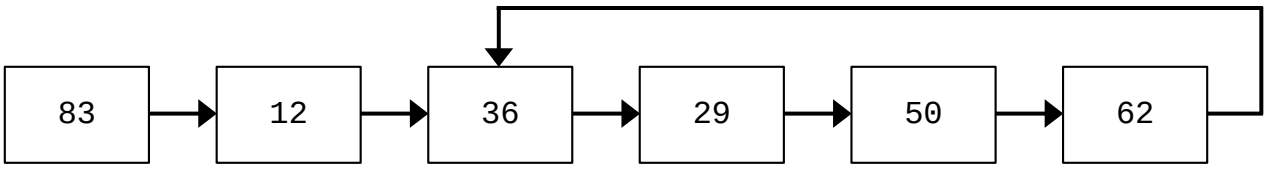


d) In the box, list all valid BFS traversals starting at node S in the graph from part (c).

6. Problem Solving (10 pts)

Write a function that takes the head of a linked list and determines whether it contains a cycle. If so, return *true*. Otherwise, return *false*.

In solving this problem, you cannot use the tortoise and the hare approach (fast and slow pointers) covered in class, or any variation thereof. You must come up with another approach. Avoid segmentation faults, and ensure your function will not get stuck in an infinite loop if it encounters a linked list with a cycle. Note also that a cycle might not involve the head of the linked list, as depicted in the following diagram:



Your solution must be efficient both in terms of runtime and memory usage (but if you have to prioritize one or the other, prioritize a faster runtime). All work must be completed within a single function (no helper functions).

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

bool hasCycle(Node *head) {
```

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

We will not grade anything on this page.