CS106B

Winter 2023

Practice Final Exam 2

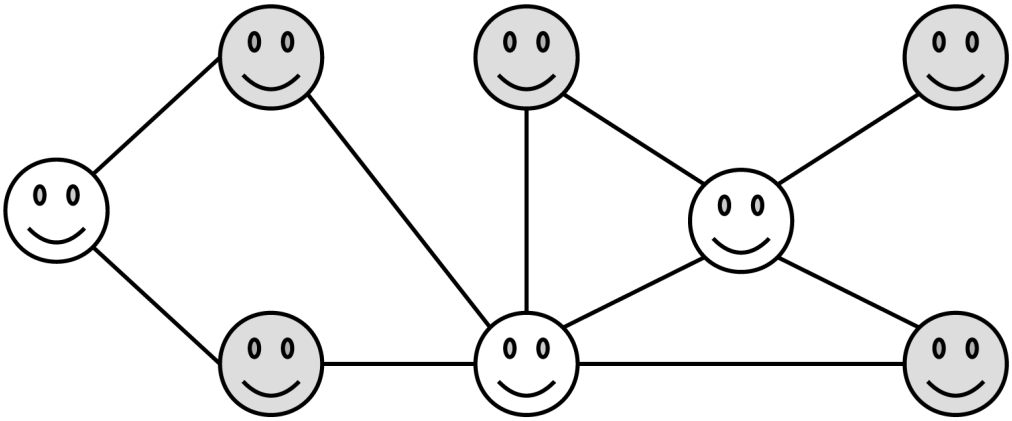
This is a printable version of the second set of practice problems from the practice final exam. It's designed to have the same format and style as the actual CS106B final exam.

Solutions can be found by visiting the Additional Practice Problems resource on the CS106B website and looking up solutions to the corresponding problems.

Problem One: Avoiding Sampling Bias

One of the risks that comes up when conducting field surveys is *sampling bias*, that you accidentally survey a bunch of people with similar backgrounds, tastes, and life experiences and therefore end up with a highly skewed view of what people think, like, and feel. There are a number of ways to try to control for this. One option that's viable given online social networks is to find a group of people of which no two are Facebook friends, then administer the survey to them. Since two people who are a part of some similar organization or group are likely to be Facebook friends, this way of sampling people ensures that you get a fairly wide distribution.

For example, in the social network shown below (with lines representing friendships), the folks shaded in gray would be an unbiased group, as no two of them are friends of one another.



Your task is to write a function

```
Set<string> largestUnbiasedGroupIn(const Map<string, Set<string>>& network);
```

that takes as input a `Map` representing Facebook friendships (described later) and returns the largest group of people you can survey, subject to the restriction that you can't survey any two people who are friends.

The `network` parameter is similar to the road network from the Disaster Preparation assignment. Each key is a person, and each person's associated value is the set of all the people they're Facebook friends with. You can assume that friendship is symmetric, so if person *A* is a friend of person *B*, then person *B* is a friend of person *A*. Similarly, you can assume that no one is friends with themselves.

Some other things to keep in mind:

- You need to use recursion to solve this problem – that's what we're testing here.
- Your solution must not work by simply generating all possible groups of people and then checking at the end which ones are valid (i.e. whether no two people in the group are Facebook friends). Along the lines of the Disaster Preparation problem, this approach is far too slow to be practical.

```
Set<string> largestUnbiasedGroupIn(const Map<string, Set<string>>& network) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

Additional space for your answer to Problem One, if you need it.

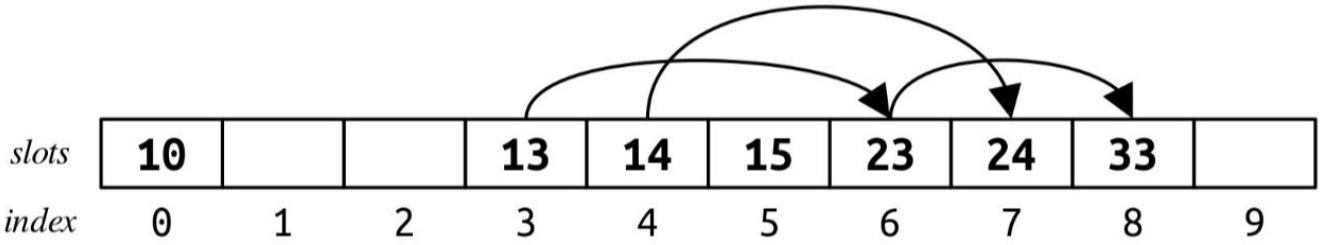
Problem Two: Leapfrog Hashing

In this problem, you'll implement a type of hash table called a ***leapfrog hash table*** that's a hybrid between a chained hash table and a linear probing hash table. Leapfrog hashing, like linear probing and Robin Hood hashing, is an open-addressing table, which means the hash table is made of a collection of individual slots, each of which can either be empty or hold an item. The slot type is as follows:

```
struct Slot {
    std::string value;
    SlotType type; // SlotType::EMPTY or SlotType::FILLED
    int link;
};
```

This slot type is the same as what you'd find in standard linear probing with one major change: the `link` field. This field can hold one of two values: either the constant `NOT_LINKED`, indicating that the slot is not linked to another slot, or the index of another slot in the table.

We can visualize a leapfrog hash table by drawing it out like a regular linear probing table, with the addition of links between slots. For example, here's a sample leapfrog hash table. Here, we're using a (bad, just for expository purposes) hash function that hashes each number to its last digit:



In this picture, if there's a link from one slot to another, we've drawn an arrow from the first slot to the slot it links to. Empty slots, and slots whose link is `NOT_LINKED`, are drawn with no outgoing arrows.

Lookups in a leapfrog hash table work as follows. As in linear probing or Robin Hood hashing, we begin by hashing the item in question to find its home slot. If that slot is empty, then the element is definitely not in the table and we can stop looking. If that slot is full and contains the item we're looking up, great! We've found it. Otherwise, the element might still be in the table, but just not in the position it wants to be in, so we will need to search for it.

This is where the link field comes into play. Rather than using the linear probing technique, we'll instead move from this slot to the slot given by the link field, seeing if the element is there. We'll keep following the links from one slot to the next until either (1) we find the element or (2) we need to follow a link pointer from a slot whose link field is the constant `NOT_LINKED`, indicating that the slot has no outgoing link. (This behavior of jumping around the table is where the name "leapfrog hashing" comes from.) The sequence of slots we visit this way is called a chain.

For example, suppose we want to look up 33 in the above table. We begin by hashing 33 to get slot 3, which is our starting slot. That slot is full, but contains a value other than 33 (specifically, 13). We therefore follow its link to slot 6. Slot 6 has 23 in it, which isn't what we're looking for, so we follow its link to slot 8. Slot 8 contains 33, so we've found the item we're looking for.

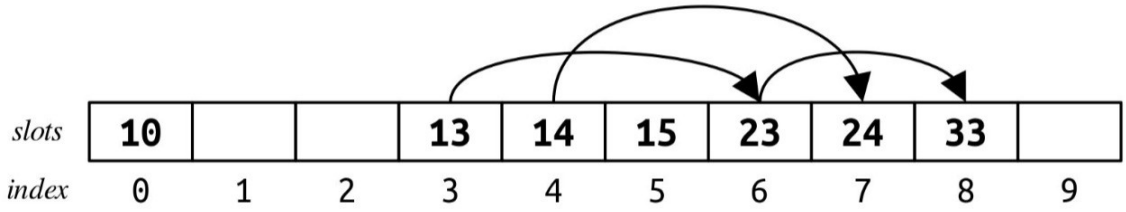
As another example, suppose we're looking for 17, which is not in the table. We begin by hashing 17 to get slot 7, which currently holds 24. That's not what we're looking for. Since slot 7 has no outgoing link, we stop searching and report that 17 is not in the table.

To look up 10, we hash to slot 0, find that 10 is present, and therefore signal that 10 is in the table.

Next, suppose we want to look up 16. We begin by jumping to slot 6, which holds 23 (not what we're looking for), so we follow the link to slot 8. Slot 8 contains 33, which isn't what we're looking for, and since it has no outgoing link we stop searching and report that the table doesn't contain 16.

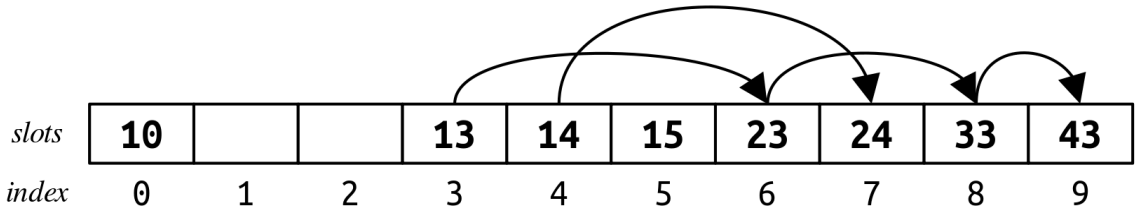
Finally, suppose we want to look up 11. We jump to slot 1, find that it's empty, and therefore immediately know that 11 isn't in the table.

Insertions in leapfrog hashing working as follows. We begin by first checking that the item we want to insert isn't in the table; if it is, then we don't need to do anything. Otherwise, the item isn't present, and we need to add it. We jump to the slot given by the item's hash code, then follow the links until we come to the last slot in the chain. For example, suppose we want to insert 43 into this table:

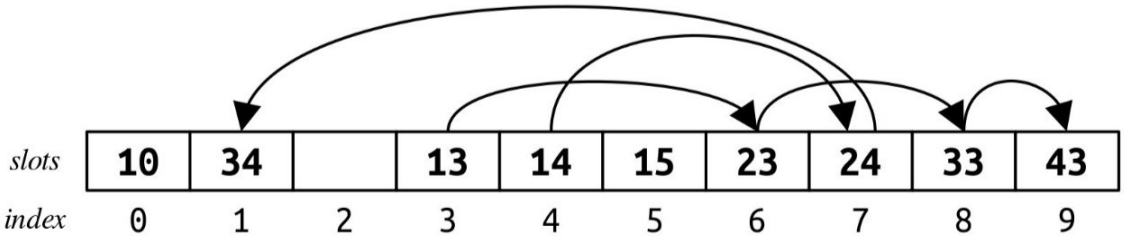


Item 43 has hash code 3, so we begin in slot 3. We follow links from slot 3 to slot 6, then from slot 6 to slot 8, stopping in slot 8 because there is no outgoing link.

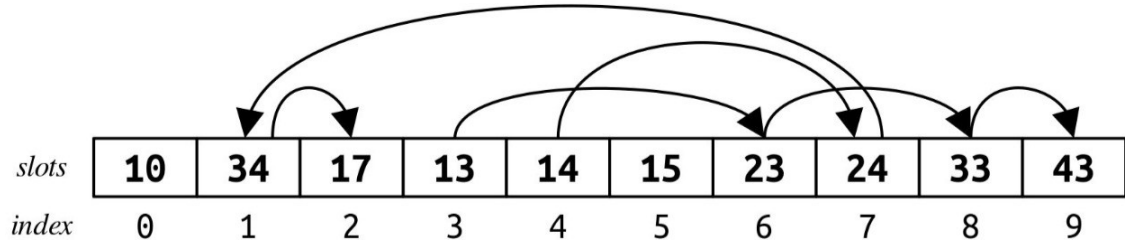
At this point, we need to find another free spot in the table at which to insert 43. To do so, we use a regular linear probing search starting in the slot just after the one we ended in (wrapping around, of course) until we find a free slot. Our chain ended in slot 8, so we start a linear probing search from slot 9. That slot is empty, so we place 43 there, linking slot 8 to slot 9. Since 43 is now at the end of its chain, we set its link to `NOT_LINKED`, since nothing comes after it. This is shown here:



Now, suppose we insert 34. We jump to slot 4 to start, then follow links until we reach the end of the chain. That takes us to slot 7. From here, we do linear probing, starting in slot 8, to find a free slot. That wraps us around the table to slot 1. We put 34 there, adding a link from slot 7 to slot 1, as shown here:



You might have noticed that, at least in this table, all items in a chain have the same hash code (13, 23, 33, and 43; 14, 24, and 34). But that's just a coincidence and isn't guaranteed to be true. For example, suppose we insert 17. We jump to slot 7, then follow the link to slot 1, then use linear probing to place 17 into slot 2. That's shown here:



To recap, here's how the lookup and insertion procedures work:

- ***Lookup:*** Jump to the slot given by the item's hash code, and stop if the slot is empty. Otherwise, follow links forward until you either find the item or run out of links to follow.
- ***Insert:*** Proceed as with a lookup. If you don't find the item, use linear probing starting from right after the last slot visited until an empty slot is found, placing the item there and updating links as appropriate.

There are some edge cases to watch out for when dealing with insertions. You have to make sure not to insert the same item twice, not to insert into a full table, and to handle the case where the initial slot is empty.

Your task is to implement the LeapfrogHashTable class shown below. Since we only discussed lookups and insertions, you do not need to handle removal of elements.

```
class LeapfrogHashTable {
public:
    LeapfrogHashTable(HashFunction<std::string> hashFn);
    ~LeapfrogHashTable();

    bool contains(const std::string& value) const;
    bool insert(const std::string& value);

private:
    enum class SlotType {
        FILLED,
        EMPTY
    };

    static const int NOT_LINKED = /* something that isn't a valid link index */;

    struct Slot {
        std::string value;
        SlotType type;
        int link;
    };

    Slot* elems;

    /* Plus anything else you'd like. */
};
```

You should not change or remove any of the code given here, but you are welcome to add to it.

```
/* Use the space here to write out the contents of the private section of the
 * LeapfrogHashTable class. We've copied what is already there, and you are
 * welcome to add to this.
 */
private:
    enum class SlotType {
        FILLED,
        EMPTY
    };

    static const int NOT_LINKED = /* something that isn't a valid link index */;

    struct Slot {
        std::string value;
        SlotType type;
        int link;
    };

    Slot* elems;
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work.

```
/* Implement the constructor here. */
LeapfrogHashTable::LeapfrogHashTable(HashFunction<string> hashFn) {









}

/* Implement the destructor here. */
LeapfrogHashTable::~LeapfrogHashTable() {









}
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's more space for your answer on the next page.

```
/* Implement contains here. */
bool LeapfrogHashTable::contains(const string& elem) const {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's more space for your answer on the next page.

```
/* Implement insert here. */
bool LeapfrogHashTable::insert(const string& elem) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work.

Problem Three: Data Structure Sleuthing

Below are four functions. We picked one of those functions and ran it on many different values of n. We captured the output of that function on each of those inputs, along with the runtime. That information is printed in the table to the right.

```
int function1(int n) {
   Stack<int> values;
   for (int i = 0; i < n; i++) {
      values.push(i);
   }

   int result;
   while (!values.isEmpty()) {
      result = values.pop();
   }

   return result;
}
```

```
int function2(int n) {
   Queue<int> values;
   for (int i = 0; i < n; i++) {
      values.enqueue(i);
   }

   int result;
   while (!values.isEmpty()) {
      result = values.dequeue();
   }

   return result;
}
```

```
int function3(int n) {
   Set<int> values;
   for (int i = 0; i < n; i++) {
      values.add(i);
   }

   int result;
   for (int value: values) {
       result = value;
   }

   return result;
}
```

```
int function4(int n) {
   Vector<int> values;
   for (int i = 0; i < n; i++) {
      values.add(i);
   }

   int result;
   while (!values.isEmpty()) {
      result = values[0];
      values.remove(0);
   }

   return result;
}
```

n	*Time*	*Return Value*
100,000	0.137s	99999
200,000	0.274s	199999
300,000	0.511s	299999
400,000	0.549s	399999
500,000	0.786s	499999
600,000	0.923s	599999
700,000	0.960s	699999
800,000	1.198s	799999
900,000	1.335s	899999
1,000,000	1.472s	999999

Answer the following questions.

i. For each of these pieces of code, tell us its big-O runtime as a function of n. No justification is required.

We will grade whatever you write in this box. Write your scratchwork elsewhere.

ii. For each of these pieces of code, tell us whether that function could have given rise to the return values reported in the rightmost column of the table. No justification is required.

We will grade whatever you write in this box. Write your scratchwork elsewhere.

iii. Which piece of code did we run? How do you know? Justify you answer in at most fifty words.

We will grade whatever you write in this box. Write your scratchwork elsewhere.

Problem Four: The Classic Interview Question

Here's a classic interview question that's so overused that it's almost an in-joke among software engineers. You are given the following `Cell` type, which represents a cell in a singly-linked list:

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

Write a function

```
void reverse(Cell*& list);
```

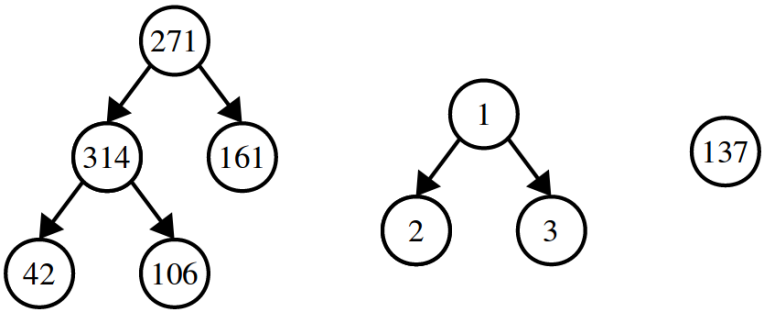
that reverses a linked list. Your function should work by rewiring the pointers within the list rather than changing the value fields of any of the cells. Your function should run in time O(n) and you must not use any container types (e.g. `Vector`, `Stack`, `string`, etc.) in the course of solving this problem.

We will grade whatever you write in this box. Use the remaining space on the page for scratch work. There's more space on the next page, if you need it.

Extra space for your answer to Problem Four, if you need it.

Problem Five: Agglomerative Clustering

A ***full binary tree*** is a binary tree structure where each node either has two children or no children. The Huffman encoding trees you worked with Assignment 8 are an example of full binary trees, and here are a few more.



Note that full binary trees are not necessarily binary *search* trees:

Let's imagine that we have a type representing a node in a full binary tree, which is shown here:

```
struct Node {
    double value;  // The value stored in this node

    Node* left;    // Standard left and right child pointers
    Node* right;
};
```

Your first task in this problem is to write a function

```
Set<double> leavesOf(Node* root);
```

that takes as input a pointer to the root of a full binary tree, then returns a set of all the values stored in the *leaves* of that tree. For example, calling this function on the leftmost tree would return a set containing {42, 106, 161}, calling this function on the tree in the middle would return {2, 3}, and calling this function on the tree on the right would return {137}.

Some notes:

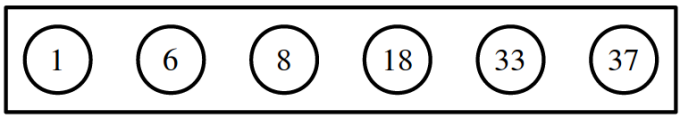
- You can assume that the pointer to the root of the tree is not null.
- You should completely ignore the values stored at the intermediary nodes.

```
struct Node {
    double value;  // The value stored in this node

    Node* left;    // Standard left and right child pointers
    Node* right;
};

Set<double> leavesOf(Node* root) {
```

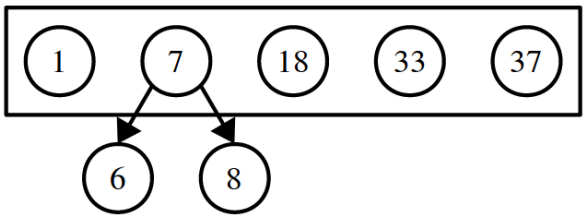
We will grade whatever you write in this box.
Use the surrounding space for scratch work.

The second part of this problem explores an algorithm called ***agglomerative clustering*** that, given a collection of data points, groups similar data points together into clusters of similar values. The algorithm is similar to Huffman encoding in that it works by starting with a bunch of singleton nodes and assembling them into full binary trees. For the purposes of this problem, we'll cluster a group of `doubles`.

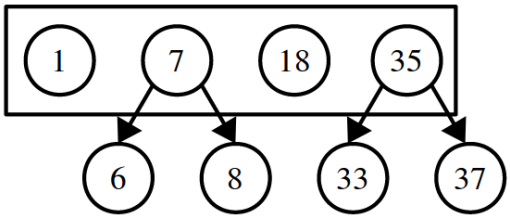
The first step in running agglomerative clustering is to create singleton trees for each of the data points. For example, given the numbers 1, 6, 8, 18, 33, and 37, we'd begin with the following trees:



We now choose the two trees whose root nodes' values are closest to one another. Here, we pick the trees holding 6 and 8. We then merge those two trees into a single tree, and give the root of that tree the average value of all its leaves. Here, the leaves are 6 and 8, so the new tree has root value 7:

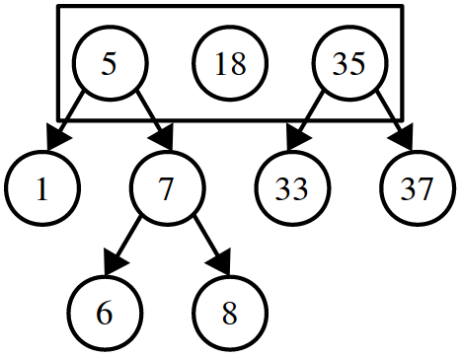


We repeat this process, again selecting the two trees whose root values are as close as possible. In this case, that would be 33 and 37, which get merged into a new tree:



Notice that the new root node has the value 35, the average value of its leaves.

On this next step, we'll find that the trees with the two closest roots are the ones with roots 1 and 7. We merge those trees into a new tree. As before, the root of this tree is then assigned the average value of all the leaves. The leaves have values 1, 6, and 8, so the new root gets the value 5. Here's the result:



As with Huffman encoding, each step in this process reduces the number of trees by one. Unlike Huffman encoding, though, we'll typically stop this algorithm before it's reduced everything down to a single tree; we'll have a separate parameter to the overall function that tells us how many trees to hand back.

If we stop here, we have three clusters: the cluster {1, 6, 8}, the cluster {18}, and the cluster {33, 37}, which you can see by looking at the leaves of the resulting trees.

Write a function

```
Set<Node*> cluster(const Set<double>& values, int numClusters);
```

that takes as input a set of values and a desired number of clusters, then runs the agglomerative clustering algorithm described on the previous page to form the specified number of clusters. This function should then return a `Set<Node*>` containing pointers to the roots of the trees formed this way. Some notes:

- You can assume the number of clusters is less than or equal to the number of values and greater than or equal to one (the algorithm only works on values in those ranges.)
- Unlike Huffman encoding, ***don't use a priority queue to determine which trees to merge;*** it doesn't work well here. It's fine to iterate over all pairs of trees to see which two are closest.
- If there's a tie between which pair of trees is closest, break the tie however you'd like.
- You'll almost certainly want to use the `leavesOf` function you wrote in the first part of this function in the course of writing up your solution.

```
/* You can assume this function works correctly. */
Set<double> leavesOf(Node* root);

/* Feel free to use this helper function to determine the absolute value of the
 * difference between two numbers.
 */
double distance(double a, double b);

Set<Node*> cluster(const Set<double>& values, int numClusters) {
```

We will grade whatever you write in this box. Anything else will be considered scratch work. There's extra space for your answer on the next page.

Extra space for your answer to Problem Five, if you need it.