Practice Final Exam

Starter Files

The starter files for this practice exam are available here:

Practice Exam Starter Files

Problem One: All Squared Away

Here's a little number puzzle: can you arrange the numbers 1, 2, 3, …, and 15 in a sequence such that the sum of any two adjacent numbers is a perfect square? (A ***perfect square*** is a number that's the square of an integer. So, for example, $16 = 4^2$ is a perfect square, and $1 = 1^2$ is a perfect square, but 19 isn't a perfect square). There is indeed a way to do this, which is shown here:

```
8,  1,  15,  10,  6,  3,  13,  12,  4,  5,  11,  14,  2,  7,  9
```

However, you *can't* do this with the numbers 1, 2, 3, …, 18, and you *can't* do this with the numbers 1, 2, 3, …, 19, but you *can* do this with the numbers 1, 2, 3, …, 23.

Open the file **`AllSquaredAway.cpp`** and implement a function

```
bool hasSquareSequence(int n, Vector<int>& sequence);
```

that takes as input a number *n*, then determines whether there is some way to arrange the numbers 1, 2, 3, …, n such that each number appears exactly once and the sum of any two adjacent numbers is a perfect square. If so, the function should return **`true`** and update the outparameter **sequence** to contain one such way of arranging the numbers in this way. If not, the function should return **`false`**, and it can do whatever it likes to the **sequence** parameter.

For convenience, we have provided you with a function

```
bool isPerfectSquare(int value);
```

that takes as input an integer and returns whether that integer is a perfect square. (Please do not modify this function.)

You are encouraged to write your own **STUDENT_TEST** cases in **`AllSquaredAway.cpp`**, but you are not required to do so. Our provided tests are available in **`AllSquaredAwayTests.cpp`**.

Some notes on this problem:

- You must implement this function recursively – that's kinda what we're testing here – but you are welcome to make the **hasSquareSequence** function a wrapper.
- If the input value *n* is zero or negative, your function should call **`error()`** to report an error. However, an input with *n* = 1 is permissible and should produce the sequence of just the number 1 on its own.
- There are no constraints on what the **sequence** parameter may hold when your function is first called. It could be empty, or it could contain a gigantic list of random numbers.
- Do not modify the signatures of **hasSquareSequence** or **`isPerfectSquare`**. Those functions must have the exact parameters and return values shown here.
- Your solution does not have to be as efficient as possible, but should be fast enough to pass all the stress tests in the appropriate amount of time (plus several stress tests we will use when grading). However, simply passing the provided stress tests does not guarantee that your code is sufficiently efficient.

Solution

Problem Two: Big-O Analysis Revisited

We've provided you a collection of five functions in **`BigOFunctions.cpp`**, along with the runtime plotter you had in Assignment 5. For each of those functions, determine its big-O time complexity, and ***briefly explain your reasoning*** (at most fifty words per answer). Place your answers in the file **`BigOAnswers.txt`**.

Some notes on this problem:

- You do not need to – and in fact, should not – write any code here.
- The advice from Assignment 5 about the runtime plotter is still applicable here – watch out for noise due to background processes running, understand that very small runtimes can be heavily influenced by noise, etc.
- Feel free to consult the Stanford C++ Library Documentation for information about the big-O costs of different operations on different types.

Solution

Problem Three: Leapfrog Hashing

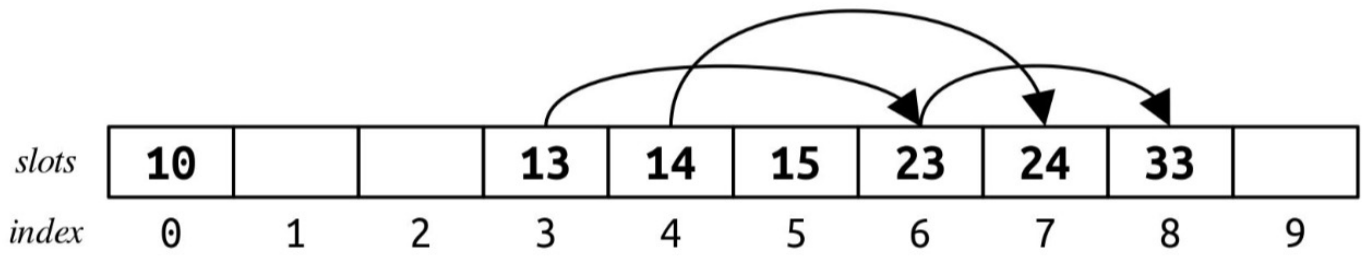
In this problem, you'll implement a type of hash table called a ***leapfrog hash table*** that's a hybrid between a chained hash table and a linear probing hash table.

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 for leapfrog hashing 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 Leapfrog Hashing

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.

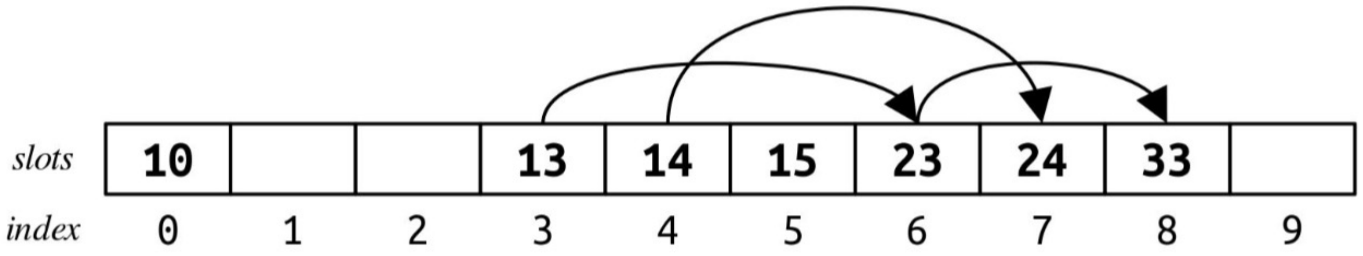
Next, suppose we want to look up 16. We begin by jumping to slot 6, which holds 23 (not what we're looking for), so we follow the link to slot 8. Slot 8 contains 33, which isn't what we're looking for, and since it has no outgoing link we stop searching and report that the table doesn't contain 16.

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

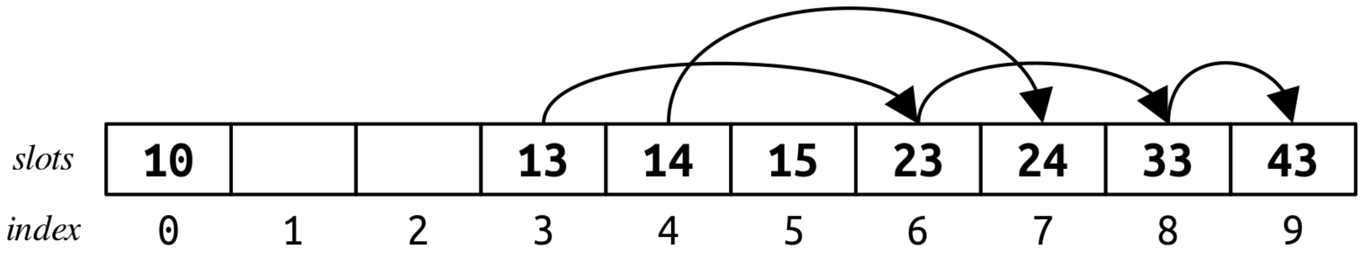
Insertions in leapfrog hashing working as follows. We begin by first checking that the item we want to insert isn't in the table; if it is, then we don't need to do anything. Otherwise, the item isn't present, and we need to add it. We jump to the slot given by the item's hash code, then follow the links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

For example, suppose we want to insert 43 into this table:

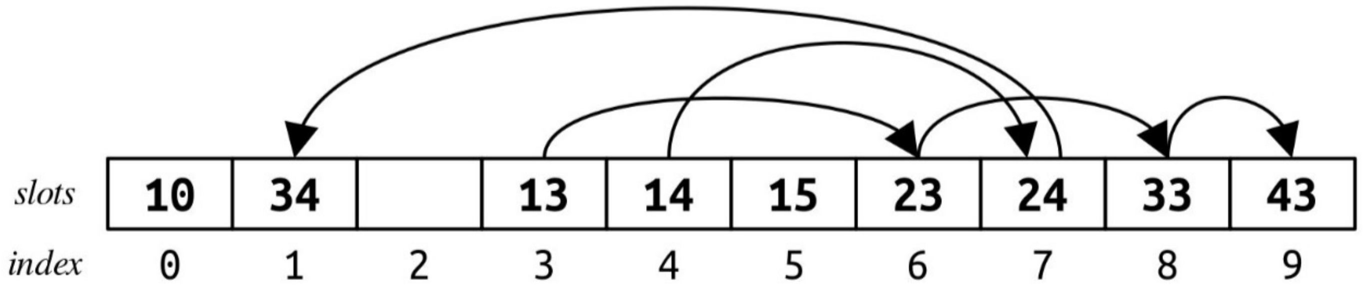


Item 43 has hash code 3, so we begin in slot 3. We follow links from slot 3 to slot 6, then from slot 6 to slot 8, stopping in slot 8 because there is no outgoing link.

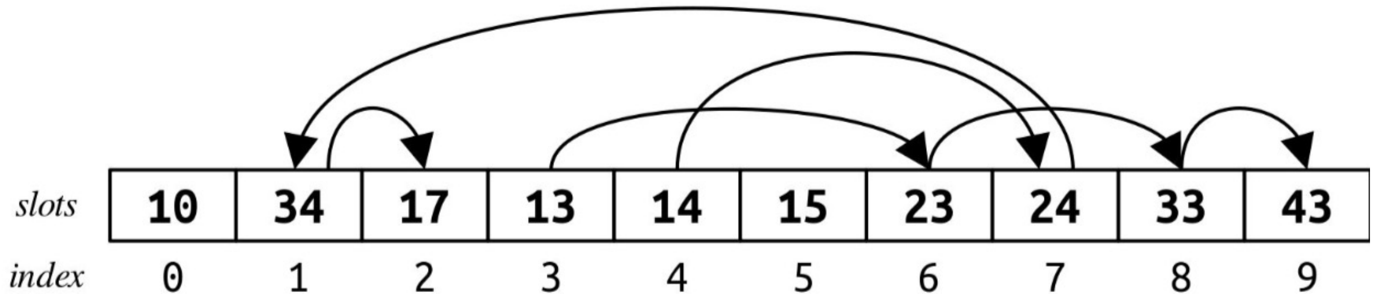
At this point, we need to find another free spot in the table at which to insert 43. To do so, we use a regular linear probing search starting in the slot just after the one we ended in (wrapping around, of course) until we find a free slot. Our chain ended in slot 8, so we start a linear probing search 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

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.

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

    int  size() const;
    bool isEmpty() const;

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

    void printDebugInfo() const;

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;
};
```

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

All your code should go in the files **LeapfrogHashTable.h** and **LeapfrogHashTable.cpp**. You are encouraged to write your own **STUDENT_TEST** cases in **LeapfrogHashTable.cpp**, but you are not required to do so. Our provided tests are available in **LeapfrogHashTableTests.cpp**.

Some notes on this problem:

- An excellent warmup before you start coding things up: look at the very first image of a leapfrog hash table from this section. In what order did we insert those items into the hash table? Is there exactly one way to do it, or are there many? You do not need to submit your answers to this question; it's just a warmup to help solidify your understanding of how leapfrog hashing works. However, we definitely recommend doing this before you write any code to make sure you're rock solid on how the procedures work!
- As with Assignment 7, you must do all of your own memory management, and you must not use any of the standard container types (**Vector**, **Map**, **Stack**, etc.)
- As with Assignment 7, your hash table should have a fixed size given by the number of slots specified by the hash function in the constructor. You should not rehash even if the table starts to get full. In particular, insertions should fail if the table is full.
- As with Assignment 7, your implementations of **contains()** and **insert()** must not be recursive, since that might result in stack overflows on large inputs.

- The **`size()`** and **`isEmpty()`** functions should run in time O(1).
- We have provided a set of tests for **`LeapfrogHashTable`** that are not as extensive as the ones from Assignment 7. You are encouraged to write your own tests, but you are not required to do so.
- You can do whatever you'd like with **`printDebugInfo`**. We won't be calling this function when grading your submission.
- Do not change any of the signatures of the functions from **`LeapfrogHashTable`**, edit the **`SlotType`** or **`Slot`** types, or remove the **`Slot* elems`** data member. You may, however, add additional private data members or private member functions.
- Your code should be fast enough to pass all the provided stress tests (plus several we will use during grading). However, simply passing the provided stress tests does not guarantee that your code is sufficiently efficient.

Solution