YEAH A5: Linked Lists!

CS106B Summer '21
Jin-Hee Lee, Grant Bishko, Lauren Saue-Fletcher
Assignment 5 is due Tuesday, August 3 at 11:59 pm PDT

The grace period extends until Thursday, August 5 at 11:59 pm PDT
Linked Lists

1. Warmup
2. Labyrinth
3. Sorting Linked Lists
Linked Lists

1. Warmup
2. Labyrinth
3. Sorting Linked Lists
Before anything, we're going to practice catching mistakes with memory and pointers.

As I'm sure we've all experienced in PQ, pointers can get tricky.

Lucky for us, this warmup walks us through how to handle different kinds of trickiness!
Memory Debugging

A memory error, (such as trying to access garbage memory, dereference a null pointer) can cause our program to crash and make a big fuss.

A memory leak, however, can pass on silently and is therefore more difficult to catch.

So, why is it a problem?

Well, memory leaks occur when we ask for new memory, but don't delete it when we're done using it.

Like any relationship, we can't keep taking without giving anything back -- that's just poor memory management! (And if we keep doing this, we'll have no more memory to take!)

We can use SimpleTest to observe the differences between how a memory error and a memory leak manifests itself in our testing harness.
Warmup Short Answers

The expectation for these questions is simply to...

- Observe
- Report what you see!
Questions on the warm-up?
Linked Lists

1. Warmup
2. Labyrinth
3. Sorting Linked Lists
Linked Lists

1. Warmup
2. Labyrinth
3. Sorting Linked Lists
In this exercise, you are stuck in a labyrinth.

To escape, you need 3 items:

1. Spellbook
2. Potion
3. Wand

You'll start off by being plopped in a random location in this labyrinth represented by a linked list of MazeCell structs.
```c
struct MazeCell {
    string contains;  // Either '', "Spellbook", "Potion", or "Wand"
    MazeCell *north;  // The cell to the north, or nullptr if can't go north.
    MazeCell *south;  // The cell to the south, or nullptr if can't go south.
    MazeCell *east;   // The cell to the east, or nullptr if can't go east.
    MazeCell *west;   // The cell to the west, or nullptr if can't go west.
};
```
Escaping the Labyrinth!

After being plopped in a starting location, your job is to roam around the labyrinth until we've collected all 3 items.

(Then, we can cast the spell and escape!)

(We'll come back to this after...)
**bool isPathToFreedom(MazeCell* start, string path, Set<string> needed)**

- **start**: pointer to a MazeCell representing the starting point
- **path**: a string made up of only 'N', 'S', 'E', 'W'
- **needed**: the items you need to collect
  - For example, in the labyrinth we saw, this would be {"Spellbook", "Potion", "Wand"}
  - Be sure to check items in the set generally (don't assume it will always be those 3 items, don't manually check for a certain string)

- **Returns true if...**
  - The path itself is legal (doesn't make any invalid moves from one cell to the next)
  - All items from the needed set have been seen (we've been to that cell)
    - *After picking up all items, any remaining steps are ignored*
bool isPathToFreedom(MazeCell* start, string path, Set<string> needed)

Some notes as you implement:

- We guarantee that start will not be nullptr
- *** A path cannot take a step that follows the nullptr! This is an invalid move. ***
- Call error() if there are any invalid characters in your path (not NSEW)
- *** Implement iteratively ***
- The start location can be any place in the labyrinth.
- The set can contain any set of items.
- The path can visit the same place more than once (this is a-okay!)
- There can be multiple cells with a needed item, and you can go to any to pick it up.
- *** Do not allocate any new MazeCells.***
  - You can use a MazeCell* without using the keyword new
Escaping Your PERSONAL Labyrinth!

After being plopped in a starting location, your job is to roam around the labyrinth until we've collected all 3 items.

(Then, we can cast the spell and escape!)
Escaping Your PERSONAL Labyrinth!

At the top of the `labyrinth.cpp` file, enter your name as the value of `kYourName`

```cpp
const string kYourName = "NAME!";
```

Then, scroll down all the way to the final test case and set a breakpoint somewhere in this test.

It's time to explore.
When you first hit your breakpoint, you'll find this in your variables pane:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[statics]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>allThree</td>
<td>&lt;3 items&gt;</td>
<td>Set&lt;std::str</td>
</tr>
<tr>
<td>startLocatio...</td>
<td>@0x115fcbee0</td>
<td>MazeCell</td>
</tr>
<tr>
<td>contents</td>
<td></td>
<td>std::string</td>
</tr>
<tr>
<td>east</td>
<td>0x0</td>
<td>MazeCell *</td>
</tr>
<tr>
<td>north</td>
<td>0x0</td>
<td>MazeCell *</td>
</tr>
<tr>
<td>south</td>
<td>@0x115fb5f60</td>
<td>MazeCell</td>
</tr>
<tr>
<td>west</td>
<td>0x0</td>
<td>MazeCell *</td>
</tr>
</tbody>
</table>

Here, it looks like:

- There are no contents in this cell
- south is the only valid move I can make, since every other direction is null.
Now, I've opened up the `south` pointer, which reveals the cell that's south of my starting location.

From here, I can see that:

- There are no contents in this cell
- `east`, `north`, or `south`, since they all contain valid memory addresses.
Using the Debugger

Now, I need to open up east, north, and south.

And this process continues!
IT'S A SECRET

A VERY BIG SECRET
Best Advice for Labyrinth

Draw out each MazeCell the labyrinth as you explore.

- Does this cell have an item?
- Which directions are valid from this cell?
  - i.e., which pointers are not nullptr?

You should be able to keep opening up the debugger and draw each cell and its directions patiently to fully unravel the labyrinth.

From there, you should be able to trace out a valid path to reach ~freedom~

Btw... we love the labyrinth!!! One of the coolest assignment tasks, imo.
Questions on labyrinth?
Linked Lists

1. Warmup
2. Labyrinth
3. Sorting Linked Lists
Sorting Linked Lists

Linked lists are awesome because we can sort them pretty efficiently!

Re-wiring is the name of the game!
Utility Functions!

For this assignment, you will be writing your own utility functions to help with testing!! This is a SUPER important skill to develop!
Utility Functions!

For this assignment, you will be writing your own utility functions to help with testing!! This is a SUPER important skill to develop!

1. `void printList(ListNode* front)`
2. `void deallocateList(ListNode* front)`
3. `ListNode* createList(Vector<int> values)`
4. `bool areEquivalent(ListNode* front, Vector<int> v)`

Careful for memory bugs!!!
Allocating/Deallocating Memory

Your TESTS are the ones that allocate and deallocate memory in this part of the assignment. This is where you need to be careful! Your actual sorting algorithms should only rearrange the lists, not create/delete nodes.
Allocating/Deallocating Memory

Deallocating memory can be tricky when we're rearranging nodes
Allocating/Deallocating Memory

Deallocating memory can be tricky when we're rearranging nodes
Allocating/Deallocating Memory

Deallocating memory can be tricky when we’re rearranging nodes

Not both! We’re deleting nodes, NOT pointers
Allocating/Deallocating Memory

Deallocating memory can be tricky when we're rearranging nodes.

Also not both here -> because list2 is already part of list1
Suggested Test Cases

- Sorting an empty list
- Sorting a single-element list
- Sorting a list in already sorted or reverse sorted order
- Sorting a list that contain some duplicate elements or all duplicate elements
- Sorting a list that contains positive and negative numbers
- Sorting a list of random values, randomly organized
- Sorting a very long list
Your task: void quickSort(ListNode*& list)

QuickSort is a recursive “divide and conquer” algorithm:

1) Partition: Divides the imputed list into sublists
   - Sublists that are: Less than, greater than, equal to a chosen “pivot” node
2) Recursively sort the less than/ greater than sublists
3) Concatenate: Joins the sorted sublists into one combined list
Incremental Testing!!!

Test EVERY SINGLE FUNCTION thoroughly before moving on.

Do not write a ton of code before testing - trust us it will not be a fun time :/
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists
Partition

1) Choose first element of list as “pivot”
2) Divide elements of list into 3 sublists: elements that are less than, greater than, or equal to that pivot
3) Redistribute those nodes into the 3 sublists

Tips:
- DO NOT MAKE NEW NODES HERE. REWIRE THINGS INSTEAD
- Use pass-by-reference parameters here since partition can only return 1 thing, but you are creating 3 lists!
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists

Do we need to recursively sort equal?
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists

Do we need to recursively sort equal?
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists
Concatenate

- Combine your lists into one! Attach one list to follow at the end of another.
- Hint: try making it only work with 2 parameters, and call your concatenate function more than once!
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists
Quick Example

1) Partition
2) Recursively sort
3) Concatenate lists
Requirements for Sort Routines

- Your sorts must operate **solely by rewiring the links** between existing ListNode.
  - Do not allocate or deallocate any ListNode (i.e. no calls to new or delete).
  - Do not replace/swap the ListNode data values.
  - You may declare ListNode * pointers to be used temporarily when changing links between ListNode.
- Do not use a sort function or library to arrange the elements, nor any algorithms from the STL C++ framework.
- Do not create any temporary or auxiliary data structures (array, Vector, Map, etc.).
- Both of your sorts should be capable of sorting an input list of random values of effectively any length.
Assignment tips

- Incremental testing
- Draw lots of pictures to plan out each part before implementing it
- Use the debugger to compare expected behavior to actual behavior
Questions?
You got this!!