Notes for Programming in C Lab Session #6

October 7, 2024

1 Introduction

The purpose of this lab session is to write some small programs that have a slightly more intense set of pointer manipulations and dynamic memory management operations than in the previous labs.

2 Overview

In this lab, you will define some functions to manipulate graphs. The graph programs themselves will not, alas, do much, but they do illustrate many of the techniques you will need to when writing more interesting C programs.

As in Lecture 6, the key data type is the *marked node*, which is a structure with a `value` field, possibly-null `left` and `right` subtree fields, and a boolean flag `marked`.

```
struct node {
  bool marked;
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node Node;
```

A pointers to a `Node` can be used to represent arbitrary graphs in memory.[1]

As we saw in lecture, it is often useful to keep track of whether or not a node has been visited or not by updating the `marked` flag. In the instructions below, an "unmarked node" is (a) a non-null node whose `marked` field is false, and (b) for which every non-null node reachable from that node is also has a false `marked` field.

Conversely, a "marked node" is taken to mean a non-null node whose `marked` field is true, and (b) for which every non-null node reachable from that node is also has a true `marked` field.

3 Instructions

1. Download the `lab6.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab6.tar.gz`.
3. This will extract the `lab6/` directory. Change into this directory using the `cd lab6/` command.

[1] Technically, these nodes can represent graphs with a maximum branching factor of 2. More general graphs can be represented by replacing the left and right fields with an array of node pointers. However, this does not add any essential difficulty, so we won't consider it in this lab.

4. In this directory, there will be files lab6.c, tree.h, and tree.c.

5. There will also be a file Makefile, which is a build script that can be invoked by running the command make (without any arguments). It will automatically invoke the compiler and build the lab6 executable.

6. You can (and should!) invoke make sane to build with the address and undefined behaviour sanitizers.

7. Run the lab6 executable, and see if your program works. The expected correct output is in a comment in the lab6.c file.

4 The Functions to Implement

4.1 Basic problems

- `int size(Node *node);`

  Given a pointer to an unmarked node node, this function returns the total number of distinct, non-null nodes reachable from node, including itself.

  If passed a null pointer, it returns 0. It also marks all of the nodes reachable from node.

- `void unmark(Node *node);`

  Given a marked node node, this function sets the marked field of node and every node reachable from it to false.

- `bool path_from(Node *node1, Node *node2);`

  Given two nodes node1 and node2, this function returns true if there is a path (via the left and right fields) of length 0 or more from node1 to node2.

  If either node1 or node2 is NULL, then this function returns false.

- `bool cyclic(Node *node);`

  This function returns true if there is a path of length 1 or more from node to itself, and false otherwise.

4.2 Challenge problems

In the Lecture, we freed the memory associated with a graph by dynamically allocating a list storing all of the reachable nodes. In this lab exercise, you have implemented the size function, which tells you the number of reachable nodes.

This means it should be possible to deallocate a graph using an array, rather than a linked list.

- `void get_nodes(Node *node, Node **dest);`

  This function receives a node pointer node, and a pointer into a buffer of node pointers dest, as arguments. The get_node function should then update the buffer with all of the unmarked nodes reachable from node via paths that only go through unmarked nodes.

- `void graph_free(Node *node);`

  This function should free a graph. It should find the nodes to deallocate by declaring an automatic array of the right size and passing a pointer into this array to get_nodes.

  Your implementation of graph_free should not be recursive, and should not allocate any memory beyond the stack allocation of the buffer storing the reachable nodes.