Worksheet 3

Overview

The workshop for Week 4 will start with a tutorial on binary search tree insertion and how to keep a node counter updated once an item is inserted.

Tutorial Questions

Question 3.1 Given the following input:

- 8 4 9 11 6 7 1 5 3 14 10 13 2

Write the insertion function used to get the binary search tree shown below. You may find it useful to draw what you want to do for a few steps on paper first before trying to write the code for it.

NOTE: You will also have to implement the freeTree function to resolve the leak checking functionality if you choose to complete this exercise in the notebook. You can also utilise the fflush(stdout) command after the tree drawing to ensure the output happens before the program ends.

```
/* llqueue.h */
/* This is used for printing the tree. It is an implementation of an abstract
    data structure called a queue, which we will cover next week. You can ignore
    llqueue.h and llqueue.c for the moment and trust they do what they say
    they do. */
struct llqueue;

/* Creates a new, empty queue. */
struct llqueue *newQueue();

/* Adds the given item to the queue, allocating if needed. */
void queue(struct llqueue **queue, void *item);

/* Takes the next item from the queue. */
void *dequeue(struct llqueue *queue);

/* Returns 1 if the queue is empty, 0 otherwise. */
int empty(struct llqueue *queue);

/* llqueue.c  */
#include <stdio.h>
#include <stdlib.h>
/* #include "llqueue.h" */
#include <assert.h>

struct llqueue {
    struct llitem *first;
    struct llitem *last;
};

struct llitem {
    struct llitem *next;
    void *item;
};

struct llqueue *newQueue(){
    struct llqueue *ret = (struct llqueue *) malloc(sizeof(struct llqueue));
    assert(ret);

    ret->first = NULL;
    ret->last = NULL;
    return ret;
}
```

```
/* Adds the given item to the queue, allocating if needed. */
void queue(struct llqueue **queue, void *item){
    if(!queue){
        return;
    }
    if(!*queue){
        *queue = newQueue();
    }
    struct llitem *newItem = (struct llitem *) malloc(sizeof(struct llitem));
    assert(newItem);
    newItem->next = NULL;
    newItem->item = item;
    if(!(*queue)->first){
        (*queue)->first = newItem;
    } else {
        (*queue)->last->next = newItem;
    }
    (*queue)->last = newItem;
}

void *dequeue(struct llqueue *queue){
    if(!queue || !queue->first){
        fprintf(stderr, "Failed to take item from queue, queue empty.\n");
    }
    void *retItem = queue->first->item;

    struct llitem *newFirst = queue->first->next;

    if(queue->first->next == NULL){
        queue->last = NULL;
    }

    free(queue->first);

    queue->first = newFirst;

    return retItem;
}

int empty(struct llqueue *queue){
    if(! queue || ! queue->first){
        return 1;
    } else {
        return 0;
    }
}

/* bst.h */
struct bst;

/*
    The function you are to write. Takes a parent pointer (null for the root),
    and returns the tree with the child in the right position. Returns the
    item in a new tree with null left/right pointers.
*/
struct bst *bstInsert(struct bst *parent, int data);

/* Draws the tree. You will need to change this if your bst uses different names. */
/* You needn't understand how this works, but you're welcome to try. */
void drawTree(struct bst *parent);

/*
    Frees the given tree, doing nothing on an empty tree, freeing child trees
    (recursively) first, then freeing the passed parent.
*/
void freeTree(struct bst *parent);

/* bst.c */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
/* #include "llqueue.h" */
/* #include "bst.h" */
```

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

/*
    The function you are to write. Takes a parent pointer (null for the root),
    and returns the tree with the child in the right position. Returns the
    item in a new tree with null left/right pointers.
*/
struct bst *bstInsert(struct bst *parent, int data){
    /* Write this function. */







}

void freeTree(struct bst *parent){
    if(! parent){
        return;
    }
    /* Fill in function according to function description. */
}

/* Calculate the number of spaces which need to appear before and after the
    data point at a given depth to allow a single character to occur in all
    children below it. */
int spacesAtDepth(int depth);

/* Calculate the depth to the deepest child of a given node. */
int countDepth(struct bst *parent);

/* Draws the tree. You will need to change this if your bst uses different names. */
/* You needn't understand how this works, but you're welcome to try. */
void drawTree(struct bst *parent){
    int i;
    if(!parent){
        /* Done, no tree to print. */
        return;
    }

    struct llqueue *currentQueue = newQueue();
    struct llqueue *nextQueue = newQueue();
    /* Used for swapping. */
    struct llqueue *tempQueue;
    /* Current node being processed */
    struct bst *current;

    /* The depth of the parent, used to work out where to place the value. */
    int depth = countDepth(parent);

    /* The number of spaces needed at the current level before and after each
        data value. */
    int spaces = spacesAtDepth(depth);


    /* Add the parent node to the queue. */
    queue(&currentQueue, parent);

    while(!empty(currentQueue) && depth >= 0){
        current = (struct bst *) dequeue(currentQueue);
        for(i = 0; i < spaces; i++){
            printf("  ");
        }
        if(current){
            printf("%2d",current->data);
        } else {
            printf("  ");
        }
```

```
        for(i = 0; i < spaces; i++){
            printf("  ");
        }
        /* Account for parent's space */
        printf("  ");

        /* Queue any children for next row */
        if(current && current->left){
            queue(&nextQueue, current->left);
        } else {
            /* Mark empty space so spacing stays consistent. */
            queue(&nextQueue, NULL);
        }

        if(current && current->right){
            queue(&nextQueue, current->right);
        } else {
            /* Mark empty space so spacing stays consistent. */
            queue(&nextQueue, NULL);
        }

        if(empty(currentQueue)){
            /* Start new row. */
            printf("\n");
            /* Update depth information. */
            depth--;
            spaces = spacesAtDepth(depth);

            /* Swap the new row to the current row. */
            tempQueue = currentQueue;
            currentQueue = nextQueue;
            nextQueue = tempQueue;
        }
    }
    /* If we reach depth 0, the queue may still have contents
        we must empty first. */
    while(!empty(currentQueue)){
        current = (struct bst *) dequeue(currentQueue);
    }
    if(tempQueue){
        free(nextQueue);
    }
    if(currentQueue){
        free(currentQueue);
    }
}

int countDepth(struct bst *parent){
    int leftDepth;
    int rightDepth;
    if(!parent){
        /* Here we assume a leaf node is at depth -1, other choices are possible. */
        return -1;
    }
    leftDepth = countDepth(parent->left);
    rightDepth = countDepth(parent->right);
    if(leftDepth > rightDepth){
        return leftDepth + 1;
    } else {
        return rightDepth + 1;
    }
}

int spacesAtDepth(int depth){
    int cDepth = depth;
    int result = 1;
    while(cDepth > 0){
        result = 2*result + 1;
        cDepth--;
    }
    return result;
}

/* main.c */
```

```
//%stdin: "8 4 9 11 6 7 1 5 3 14 10 13 2"
#include <stdio.h>
#include <stdlib.h>
/* #include "bst.h" */

int main(int argc, char **argv){
    int inputData;
    struct bst *bst = NULL;
    while(scanf("%d",&inputData) > 0){
        bst = bstInsert(bst, inputData);
    }

    drawTree(bst);

    freeTree(bst);

    return 0;
}
```

The printout of your final step should look like:
8
4 9
1 6 11
3 5 7 10 14
2 13
Question 3.2 In some circumstances, it might be useful to have a doubly-linked tree, i.e each node has a parent pointer (root pointer is initialised to NULL), as well as two child pointers. Assume each node has a counter and a depth variable. After each insertion, some counters get updated using the following assignment:
node.counter = node.left.depth - node.right.depth; // or node->counter = node->left->depth - node->right->depth;
Taking your code from the previous question, add the missing variables and show how you can use the doubly-linked tree to keep the counters up to date after a node is *inserted*. You may find it useful to replace the printed value in the draw tree function with the counter instead of current->data.

Programming Exercises

Programming 3.1 For this exercise, you will write the missing components from the first exercise,

- the *header for the binary search tree*,
- the *main function*,
  and a *value lookup function*, which will allow you to perform a lookup for a particular value and print *how many comparisons were needed to find the item* or *that it does not exist in the tree*.

If you have time, you should also try writing a function that *frees all the memory used in your tree*.

You will practice writing a *makefile* for this exercise and practice debugging your program if a bug exists.

For this, we will get you to use **GDB** and **Valgrind**. The **GDB reference card** file will likely be useful, as well as the **Valgrind tutorial** on the LMS under Resources. Valgrind is a great tool to help find memory leaks!

NOTE: You will get much more useful debugging information by compiling with *-g*, this adds additional debugging information which tends to be extremely important. Add this to all your compilations.

Checking Memory in Valgrind:

You can use the following command to check for errors you've made in memory allocation using valgrind on the university servers.
valgrind --tool=memcheck program_name
--tool=memcheck specifies that valgrind should use the memory error checker, there are other tools, such as Cachegrind, Callgrind, Helgrind, DRD, Massif, DHAT, BBV, Lackey and (technically) Nulgrind. You can find what these do at The Valgrind

Manual.

program_name is where you should place your program. Unlike gdb, you pass all your arguments when you call valgrind, so you would also add your arguments (and input redirections) here.

If you want a detailed description of memory leaks, add the option *--leak-check=yes* before your program's name (and arguments).

Quick summary of basic commands for GDB

To open your program using gdb, you can use:
gdb program_name
where *program_name* is the name of your program.

Once you're inside gdb, you can use:

- b filename.c:lineNum (to set a breakpoint at lineNum in filename.c)
- r (to run your program, reminder: you can essentially treat r/run as your program's name, so additional arguments and input/output redirections can come after the r command)
- CTRL+c (to stop your program executing)
- c (continue)
- n (next)
- s (step)
- d #number (disable breakpoint #number for now)
- p name_variable (prints the value of the variable with the given name)
- bt (backtrace, shows the program stack, what functions were called to get to where you are currently)
- q (quit, exits gdb)
- start (essentially sets a temporary breakpoint at the main function, starts your program running and then stops it)
- info locals (gives information about all local variables in the scope you're in)
- finish (finish the function you're currently in)
- command #number (set some gdb commands to run every time you hit breakpoint #number, end this list with end)

gdb has a huge suite of additional commands (including a Python interpreter, remote debugging, the ability to set variables to different values while running, time travel and functions), so you will probably find it useful to gradually transfer a lot of your usual debugging techniques over to gdb.

Once you've created a *breakpoint* and begun *run*ning your program, you can visualise your code.

Visualisation:

- Ctrl-x followed by 1 (visualise the code you're debugging)
- Ctrl-x followed by o (change focus of the screen from code to gdb terminal)
- Ctrl-x followed by a (kill visualisation of code)

Programming 3.2 If you are confident with the previous exercise, try to *maintain a balanced tree while inserting*, using rotations (utilise the counter variable and parent for this). You will very likely find drawing out the rotations is useful. As a hint, only your insert function will need to change.

Programming Challenge 3.1 You've been put in charge of part of the UN dance party, a formal dance party celebrating the first year in quite some time without any country changing their flag. As part of this party, you have a number of guests of honour coming. Your guests of honour may be on peaceful terms, but a few countries are still not on particularly good terms and have said, in no uncertain terms, that they do not want to be partnered with certain countries for PR purposes. Knowing you're an expert programmer taking an impressive sounding "Algorithms and Data Structures" subject, the UN has asked you to write a program to find a set of pairings of dancers that does not violate any of the pairs which don't want to be paired. [1]

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#define GRUDGECOUNT 5
#define NATIONS 8
#define PARTNERS 2
#define GROUPS NATIONS/PARTNERS

/*
    Returns 1 if any pair in the array given violates
    the grudges given (that is, any pair matches any
    grudge in any order). Returns 0 otherwise.
    It is assumed that no grudges occur with someone
    and themself, though the function will still
    return a correct result in that case by some
    definitions, that's not too important for this
    problem.
*/
int conflict(int grudges[GRUDGECOUNT][PARTNERS], int pairs[GROUPS][PARTNERS]);

int main(int argc, char **argv){
    int grudges[GRUDGECOUNT][PARTNERS];
    int pairs[GROUPS][PARTNERS];
    int i, j;

    /*
        Grudges are:
        3 and 4
        5 and 6
        1 and 3
        2 and 3
        3 and 6
        Where, for sanity, indices start from 0
    */

    grudges[0][0] = 3; grudges[0][1] = 4;
    grudges[1][0] = 5; grudges[1][1] = 6;
    grudges[2][0] = 1; grudges[2][1] = 3;
    grudges[3][0] = 2; grudges[3][1] = 3;
    grudges[4][0] = 3; grudges[4][1] = 6;

    /* Find pairs using some algorithm. */


    /* ------------------------------- */
    for(i = 0; i < GROUPS; i++){
        printf("Partners %d: ",i);
        for(j = 0; j < PARTNERS; j++){
            if(j != 0){
                printf(" and ");
            }
            printf("%d",pairs[i][j]);
        }
        printf("\n");
    }

    return 0;
}



int conflict(int grudges[GRUDGECOUNT][PARTNERS], int pairs[GROUPS][PARTNERS]){
    int i,j,k,l;
    int count;
    for(i = 0; i < GROUPS; i++){
        /* Check not all partners in ith group not a grudges group */
        for(j = 0; j < GRUDGECOUNT; j++){
            count = 0;
            /* Accumulate all partners. */
            for(k = 0; k < PARTNERS; k++){
                for(l = 0; l < PARTNERS; l++){
                   if(grudges[j][k] == pairs[i][l]){
                       count++;
                       break;
                   }
                }
```

```
                if(count >= PARTNERS){
                    return 1;
                }
            }
        }
    }
    /* Found no conflicts */
    return 0;
}
```

[1]: This problem w as adapted from tuenti challenge 8 ([https://contest.tuenti.net/Challenges?id=10](Problem 10)), a 2018 programming challenge put together by Madrid-based company Tuenti. Some of those w ho did w ell in the competition ended up w orking at the company as a result of their performance. The challenge itself is a bit more involved, but may not be too much more difficult.