COP 3502 Exam #2 Review Questions

Linked Lists

1. Write a ***recursive*** function, `equal`, that takes in pointers to two linked lists and returns 1 if the two lists are equal and 0 otherwise. For two lists to be equal, they have to have the same number of elements, and each corresponding element must be equal. (For example, the lists 3, 9, 5 and 3, 9, 5 are equal, but the lists 3, 4, 7 and 3, 7, 4 are not equal and the lists 2, 9, 1 and 2, 9, 1, 4 are not equal.)

```
typedef struct node {
    int data;
    struct node* next;
} node;

int equal(node* listA, node* listB) {









}
```

2. Write a **recursive** function that takes in a pointer to the front of a linked list of integers and returns the sum of every other node, starting with the second. If the list is empty or contains one element, return 0. (Note: If you write an iterative function, the maximum credit awarded will be 6 points.) Use the struct definition and function prototype provided below:

```
struct ll {
    int data;
    struct ll* next;
};

int sumEveryOther(struct ll* front) {









}
```

Stacks

1. The function that pops an integer off the stack is defined as:

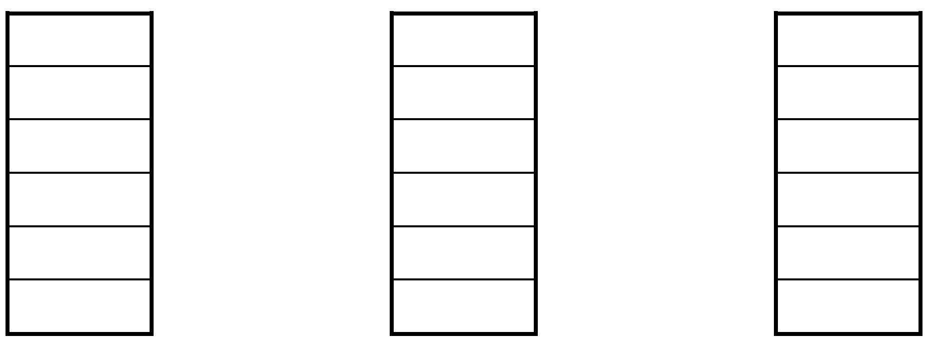
list_node *pop(list_node *p, int *val)
(A list_node contains two fields: next and data)

a) Assuming that we are using the most straightforward implementation of the stack, when calling pop what should be passed into the variable p?

b. What does this function return?

2. Utilize an operand stack to evaluate the following postfix expression. Show the contents of the stack at each of the indicated points:

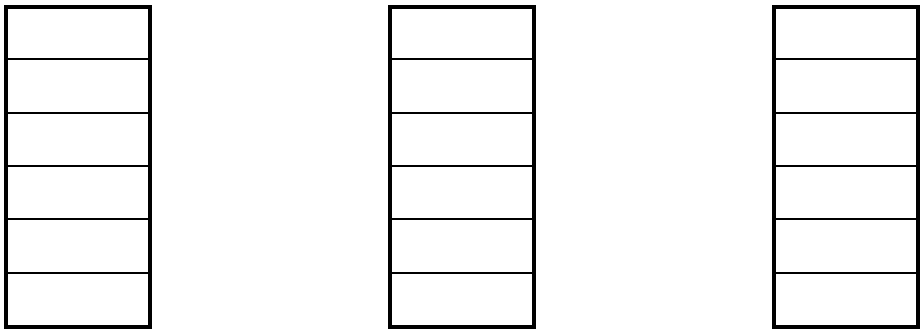
```
                        1                2          3
2   5   4   +   3   6   –   /   7   8   +   5   /   +   *
```



1 2 3

3. Utilize an operand stack to evaluate the following postfix expression. Show the contents of the stack at each of the indicated points:

```
                      1             2                3
3   6   10   2   3   +   /   +   *   4   8   2   /   1   +   *   -
```



4. An incomplete version of a linked list implementation of a stack is below. Fill in the code for each function that is empty.

```
struct stack {
  int data;
  struct stack *next;
};

void init(struct stack *front) {
  front = NULL;
}

// Returns a pointer to the new front of the stack. Assume that allocating
// space for the new node is always successful.
struct stack* push(struct stack *front, int num) {









}

// Returns the new front of the stack and sets the variable pointed to
// by ptrTop to the value popped from the top of the stack. If no such
// value exists, NULL is returned and the the variable ptrTop is pointing
// to is set to 0.
struct stack* pop(struct stack *front, int* ptrTop) {

  struct stack *temp;
  *ptrTop = 0;
  if (front != NULL) {
    temp = front;
    front = front->next;
    *ptrTop = temp->data;
    free(temp);
    return front;
  }
  else
    return NULL;
```

```
}

// Returns 1 if the stack is empty, 0 otherwise.
int empty(struct stack *front) {




}
```

Queues

1. Complete the program below so that it reads in a file ("lines.txt") storing information about enqueues and dequeues into two queues and prints out one line for each enqueue and dequeue. You are guaranteed that all operations specified are possible, so you don't need to check to see if there is enough memory for an enqueue or if a dequeue is being performed on an empty queue. The file format is as follows: the first line contains a single positive integer, *n*, representing the number of operations in the file. Each of the following *n* lines will either contain instructions for an enqueue or a dequeue. The first number on each line will contain an integer, 1 or 2, specifying which queue is being operate on. The second number on each line will contain a second integer, 1 or 2, specifying either an enqueue (1) or dequeue(2). If the second integer is 1 (enqueue), then a third positive integer will be on the line specifying the number to be enqueued. For each action, output a line with one of the two following formats:

```
Enqueue <item> into line <linenumber>.
Dequeue <item> out of line <linenumber>.
```

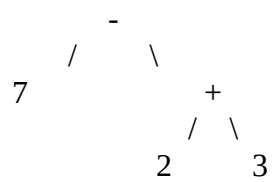
Fill in the scaffold below to complete the task:

```
#include <stdio.h>

void enqueue(struct queue* q, int item);
int dequeue(struct queue* q);
void initialize(struct queue* q);

int main() {

    struct queue q1, q2;
    initialize(___);
    initialize(___);
    FILE* ifp = fopen("lines.txt", "r");
    int i, n;
    fscanf(ifp, "%d", &n);

    for (i=0; i<n; i++) {

        int q_num, op;
        fscanf(ifp, "%d%d", &q_num, &op);

        if (op == 1) {
```

```
        }
        else {









        }
    }
    fclose(ifp);
    return 0;
}
```

Binary Trees & Binary Search Trees

1. A binary tree can store an arithmetic expression as follows: All leaf nodes store operands, while all non-leaf nodes store operators. Each non-leaf node must have two children. The value stored in such a tree is as follows: if it's a leaf node, the value is just the value stored in the node. Otherwise, the result of carrying out the operation stored in the root with the value of the left and right trees, in that order. For example, the tree



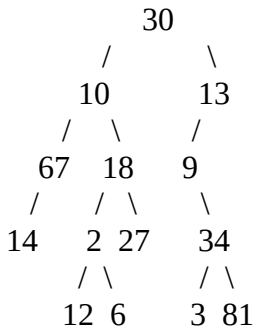
evaluates to 2, since the left of the root evaluates to 7 and the right evaluates to 5.

Consider storing expressions with only positive numbers in such a tree. If the integer -1 is stored in a node, this indicates addition and if the integer -2 is store in a node, this indicates subtraction. Complete the function below so that it returns the value of a tree stored in this manner. You may assume that the pointer passed to the function points to a non-empty valid expression tree, as described above. Use the struct provided.

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

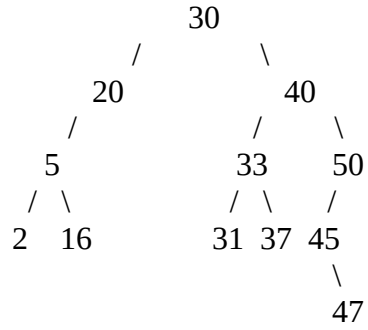
```
int evaluate(struct bintreenode* root) {




}
```

2. Determine the preorder, inorder and postorder traversals of the following binary tree:

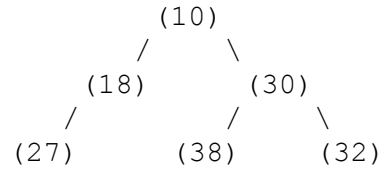


Preorder:
Inorder:
Postorder:

3. What is the result of running a post-order traversal on the binary tree shown below:



4. Consider the following figure, then answer the questions below.



a) Is this a tree (Y/N)?
b) Is this a binary tree (Y/N)?
c) Is this a binary search tree (Y/N)?
d) Is this an AVL tree (Y/N)?

5. Consider inserting the following elements into an initially empty binary search tree, in this order: 10, 5, 3, 8, 9, 1, 2, 7, 4 and 6. What is the height of the resulting tree?

AVL Trees

1. Insert the following items into an AVL tree, in the order shown: 5, 10, 15, 30, 20 and 17. Put a box around what the tree looks like after each insert. Thus, you should have 6 boxes around valid AVL trees with 1, 2, 3, 4, 5, and 6 nodes respectively.

2. Show the final state of an AVL Tree after the following inserts have been made (in this order shown).

50, 30, 10, 20, 3, 14