COP 3502 Section 2
Exam #2
Version B Solutions
Spring 2017
3/23/2017

Lecturer: Arup Guha

Directions: Answer all multiple choice questions on the scantron. Each question has a single correct answer. In case of ambiguities, choose the most accurate answer. Each of these questions is worth 4 points for a correct answer. Incorrect answers and questions left blank are worth 0 points. When you finish this exam, **double check that you have bubbled in your UCFID and Exam Version on the scantron** and then hand in the scantron **ONLY**.

Questions 1-3 involve converting the following infix expression to postfix using a stack of operators and parentheses:

(3 + 7) * ((1 + 8) - 2 * (16 / (4 - 2) - 5)) - 26

1) Right before the value 16 gets inserted into the expression, how many items are on the stack of operators and parentheses?

a) 2 b) 3 c) 4 **d) 5** e) None of the Above

Reason: The stack empties after it reaches the first close parenthesis. Here is what happens after that: *, (, (, + get pushed. Then when we get to the) after the 8, the + and one) get popped. Then the - gets pushed, followed by the * and then the open parenthesis. This is when we insert 16 into the expression. As you can see, there are 5 items in the stack at this point:

(
*****
-
(
*****

Stack

2) When the last close parenthesis gets processed, how many items are popped off the stack of operators and parentheses?

a) 1 b) 2 **c) 3** d) 4 e) None of the Above

Reason: The stack empties after it reaches the first close parenthesis which is identical method with above approach. Thus when the last close parenthesis get processed, *, (, -, * remained in the stack and (, -, * will be popped from the stack.

3) Which of the following is the equivalent postfix expression to the infix expression given above?

a) 3 7 + 1 8 + 2 16 4 2 - / 5 - * - * 26 -
b) 3 7 1 8 + + 2 16 4 2 - / 5 - * - 26 * -
c) 3 7 1 8 + + 2 16 4 2 - / 5 - * - * 26 -
d) 3 7 + 1 8 + 2 16 4 2 - / 5 - * - 26 * -

Reason: The stack empties after it reaches the first close parenthesis which is identical method with above approach. The only thing to remember is *,/has higher priority than +,-.

e) None of the Above

4) In class we implemented a queue using a linked list and one using an array. We tested both implementations with successive enqueues until the enqueue failed. The array implementation failed with fewer elements in the queue than the linked list version. What was our hypothesis as to why?

a) We were doubling our array size when it filled up so even though our code failed to find enough room for the doubling, there would have been room for quite a few more than one extra element. With the linked list implementation,

we only added one node at a time. It only failed after nearly all the available memory was filled.

b) I had different programs running on my computer in the background during the two separate program runs. The programs running on my computer while I ran the array code just happened to be using more memory.

c) I ran the linked list version with a special command line flag that specified a larger allocation of memory for the execution of the program.

d) Arrays are stored inefficiently in C, compared to linked list and a single array slot takes up twice the space of the corresponding linked list node.

e) None of the Above

5) A short video was shown in class to introduce the breadth first search. What was the video about?

a) Trees b) Rubix Cubes c) Shopping Lines
d) Corn Mazes e) None of the Above

Reason: The maze problem was defined as a breath first search problem using queue in a homework programming #4 as well.

Consider storing a string in a linked list where each node stores a single character. The following code implements functions similar to strlen and strcmp, as well as a function to convert a regular string to its equivalent linked list representation. Several lines of the implementation have been omitted. Questions 6 - 11 will be about these missing lines.

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

typedef struct chType {
    char c;
    struct chType* next;
} chType;

chType* convert(char* word);
int len(chType* word);
int compare(chType* left, chType* right);

chType* convert(char* word) {
    chType* front = NULL;
    int i;
    for (i=strlen(word)-1; i>=0; i--) {
        chType* tmp = malloc(sizeof(chType));
        tmp->c = word[i];
        tmp->next = front; // Q6
        front = tmp; // Q7
    }
    return front;
}

int len(chType* word) {
    if (word == NULL)
        return 0;
    return len(word->next) + 1; // Q8
}


int compare(chType* left, chType* right) {
    if (left == NULL && right == NULL)
        return 0; // Q9
    if (left == NULL)
        return -1;
    if (right == NULL)
        return 1;
    if (left->c != right->c)
        return left->c - right->c; // Q10
    return compare(left->next, right->next); // Q11
}
```

6) What expression should replace /*** Q6 ***/?

a) front->next b) tmp c) tmp->next

d) tmp->c **e) None of the Above**

Reason: We are inserting each item as we get to it to the front of the list pointed to by front. Thus, we want to link the node pointed by tmp to the node pointed to by front.

7) What expression should replace /*** Q7 ***/?

a) front->next **b) tmp** c) tmp->next

d) tmp->c e) None of the Above

Reason: Now that we've inserted a new item into the list, we must reassign the front of the list so that it's correct for the next loop iteration. Since tmp is pointing to the front of the list, we reassign front to point there.

8) What expression should replace /*** Q8 ***/?

a) len(word->next) + 1 b) 2*len(word->next) c) len(word) + 1

d) len(word->next) e) None of the Above

Reason: Recursively, the length of a string is simply one plus the length of the "rest of the string". The pointer word->next points to the rest of the string, excluding the first character. Thus, the + 1 is to count the first character in the string and the recursive call counts the rest. Add and return.

9) What expression should replace /*** Q9 ***/?

a) -1 **b) 0** c) 1 d) left->c + right->c e) None of the Above

Reason: If both pointers are NULL the strings are equal. In this case the compare function must return 0.

10) What expression should replace /*** Q10 ***/?

a) -1 b) 0 c) 1 d) left->c + right->c **e) None of the Above**

Reason: The correct answer is left->c - right->c. If the string pointed to by left comes lexicographically before the string pointed to by right, we must return a negative integer. The subtraction accomplishes this and also correctly handles the other case where the string pointed to by left comes strictly after the string pointed to by right where the first characters differ.

11) What expression should replace /*** Q11 ***/?

a) -1 b) 0 c) 1 **d) compare(left->next,right->next)** e) None of the Above

Reason: If we get here, the first characters match and the real decision is based upon the two substrings of the left and right where we clip off the first character. The recursive call compare(left->next, right->next) encapsulates that idea precisely.

12) Which of the following could represent the inorder traversal of a binary search tree?

a) 9 7 3 6 8 15 12 99
b) 99 12 15 9 8 7 6 3
c) 3 6 7 8 9 12 15 99
d) All of the Above (A, B and C)
e) None of the Above

Reason: In a binary search tree, all items less than a node get inserted to its left and all items greater than a node get inserted to its right. An inorder traversal of such a tree MUST produce a sorted list from smallest item to largest item. There is only one choice of the ones given that is sorted, choice B.

13) What is the run time of an insert inorder function on a sorted doubly linked list of n integers?

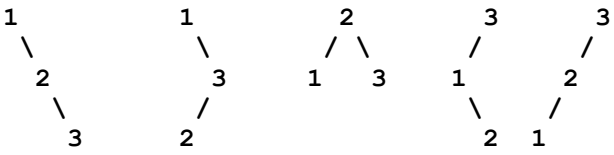
a) O(1) b) O(lg lg n) c) O(lg n) d) $O(\sqrt{n})$ **e) None of the Above**

Reason: If we must insert in order, then depending on what item we are given to insert, it could go anywhere in the list. Since we only have access to the front of the list, we could potentially iterate through the whole list to find the correct access point. But, we can always find this point and do the insertion in O(n) time, since we just need a single loop through the elements and then some constant number of statements to patch the new node into the list.

14) How many structurally different binary search trees can be created which have three nodes, storing the values 1, 2 and 3?

a) 4 **b) 5** c) 6 d) 8 e) None of the Above

Reason: There are 5. They are drawn below:



Note: In general the answer to this question for n values is C(n), where C(n) denotes the nth Catalan number. When breaking this problem down recursively, the recurrence we generate turns out to be the same recurrence that defines the Catalan numbers. For this problem, I assumed students had never heard of what a Catalan number was so my intended solution was for students to simply draw out the trees like I have done above.

15) Which of the following class examples roughly implemented a linked list of linked lists?

a) binary tree b) stack **c) Artists+CDs** d) queue e) Books

The following code implements an insert function for a binary search tree of integers, where the height of each node is stored (distance from node to furthest leaf node in its subtree). Several lines of the implementation have been omitted. Questions 16 - 21 will be about these missing lines.

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

typedef struct bintreenode {
    int data;
    int height;
    struct bintreenode* left;
    struct bintreenode* right;
} bintreenode;

bintreenode* insert(bintreenode* root, int value);
void preorder(bintreenode* root);
void freeTree(bintreenode* root);

bintreenode* insert(bintreenode* root, int value) {
    if (root == NULL) {
        bintreenode* tmp = malloc(sizeof(bintreenode)); // Q16
        tmp->data = value;
        tmp->left = NULL;
        tmp->right = NULL;
        tmp->height = 0; // Q17
        return tmp;
    }
    if (value <= root->data) {
        root->left = insert(root->left, value);
        if (root->left->height >= root->height) // Q18
            root->height = root->left->height + 1; // Q19
    }
    else {
        root->right = insert(root->right, value);
        if (/*** Q20 ***/ >= root->height)
            root->height = /*** Q21 ***/;
    }
    return root;
}
```

16) What expression should replace /*** Q16 ***/?

a) root b) malloc(sizeof(bintreenode*)) c) malloc(4)
d) malloc(sizeof(bintreenode)) e) None of the Above

Reason: To instance the temporal bintreenode node, we should allocate the memory with size of the structure, bintreenode.

17) What expression should replace /*** Q17 ***/?

a) 0 b) 1 c) 2 d) root->height e) None of the Above

Reason: The definition of the height is a distance from node to furthest leaf node in its subtree. In the condition, it says current node is NULL which means the inserted node should be a root node. Thus the height should be 0.

18) What expression should replace /*** Q18 ***/?

a) 0 **b) root->left->height** c) root->height

d) root->right->height e) None of the Above

Reason: If we look at the code, it is a case when given value is lesser than current node which means the given value should go to left side. In that case, it should check the distance of height of current node because new value was inserted and it made change for height. Therefore the condition should have a comparison between root->left->height and root->height.

19) What expression should replace /*** Q19 ***/?

a) 0 b) root->left->height c) root->height

d) root->right->height **e) None of the Above**

Reason: We hit this branch if the left subtree is deeper than the right, so we must take its height and add one. Thus, the correct expression is root->left->height + 1.

20) What expression should replace /*** Q20 ***/?

a) root->right->height b) root->left->height c) 0

d) root->height e) None of the Above

Reason: Reason: Similar to question 18, if we look at the code, it is a case when given value is larger than current node which means the given value should go to right side. In that case, it should check the distance of height of current node because new value was inserted and it made change for height. Therefore the condition should have a comparison between root->right->height and root->height

21) What expression should replace /*** Q21 ***/?

a) root->left->height **b) root->right->height+1** c) root->left->height+1

d) root->right->height e) None of the Above

Reason: This is the symmetric case to question 19, just for the right side.

22) What is the value of the following postfix expression?

3 4 5 + * 3 - 1 1 + 2 * /

a) 2 b) 3 c) 4 **d) 6** e) None of the Above

Reason: Here is the stack, step by step. Below each stack is the sign (if there was one) that was processed at that step, leading to the next stack.

```
5                    1       2
4  9       3         1   2   2   4
3  3  27  27  24  24  24  24  24  6
+  *       -       +       *   /
```

23) What is the value of the following postfix expression?

6 3 / 2 2 + * -

a) -2 b) 2 c) 4 d) 6 **e) None of the Above**

Reason: When you get to the last operator, there is only one item in the stack. If this ever happens, that means the postfix expression is invalid and doesn't have a value.

24) Let the sequence A of operations on a stack involve 3 pushes followed by 2 pops. Consider repeating the sequence A 20 times. If the stack was implemented using an array, what is the minimum size of the array necessary to ensure that these instructions are executed properly?

a) 21 **b) 22** c) 23 d) 24 e) None of the Above

Reason: For each sequence we push three times and pop 2 times, so 20 array slots are required. But if we consider 19th sequence, because the sequence will push 3 times, before popping, we need 19 + 3 = 22 spaces to store all of the items in the stack, thus the minimum array size necessary is 22.

25) Where do seahorses live?

a) mountaintops **b) the sea** c) mars d) 7-11s e) igloos