COP 3502
Final Exam Review

Dr. Neslisah Torosdagli

Content

- 11 Questions, 110 points
- Dynamic Memory Management
- Linked Lists
- Binary Trees
- Minheaps
- Recursive
- Tries
- Algorithm Analysis
- Recurrence Relations
- Bitwise Operators
- AVL Trees
- Backtracking

Dynamic Memory Management

Consider a binary search tree, where each node contains some key integer value and data in the form of a linked list of integers. The structures are shown below: the tree nodes and list nodes are dynamically allocated. We are going to eventually upgrade the structure, and when we do so, all of the dynamically allocated memory will be deleted (including all of the linked lists). Write a function called deleteTreeList that will take in the root of the tree, freeing all the memory space that the tree previously took up. Your function should take 1 parameter: a pointer to the root. It should return a null pointer representing the now empty tree.

```
typedef struct listNode {
 int value;
 struct listNode * next;
} listNode;
typedef struct treeNode {
 struct treeNode * left;
 struct treeNode * right;
 int value;
 listNode * head;
} treeNode;
```

```
treeNode * deleteTreeList (treeNode * root)
{
    // Checking the tree is empty 1 pt
   if (root == NULL)
      return NULL; // returning NULL 1 pt

  // Loop through (iterative or recursive) list at the node 1 pt
   while (root->head != NULL) {
       listNode * tmp = root->head; // Prevent use after free 1 pt
       root->head = root->head->next;
      free(tmp); // free 1 pt
    }
    // Freeing both children 1 pt each
    root->right = deleteTreeList(root->right);
    root->left = deleteTreeList(root->left);
    // Freeing the tree's root 1 pt
    free(root); // If the root is free'd last 1 pt
    return NULL; // returning NULL at end 1 pt
}
```

Linked List

An alternate method of storing a string is to store each letter of the string in a single node of a linked list, with the first node of the list storing the first letter of the string. Using this method of storage, no null character is needed since the next field of the node storing the last letter of the string would simply be a null pointer. Write a function that takes in a pointer to a linked list storing a string and returns a pointer to a traditional C string storing the same contents. Make sure to dynamically allocate your string in the function and null terminate it before returning the pointer to the string. Assume that a function, length, exists already (that you can call in your solution), that takes in a pointer to a node and returns the length of the list it points to. The prototype for this function is provided below after the struct definition.

```
typedef struct node {
 char letter;
 struct node* next;
} node;
int length(node* head);
```

```
char* toCString(node * head) {
          // Find length of the list.
           int len = length(head); // 1 pt
           // 2 pts – allocate space.
           char* res = malloc((len+1)*sizeof(char));
           int i=0; // 1 pt – init loop variable
           // Go through list – 4 pts, 1 pt per line.
           while (head != NULL) {
                     res[i] = head->letter;
                     head = head->next;
                     i++;
           }
           // 1 pt – null terminate string.
           res[len] = '\0’;
           // 1 pt – return pointer to string.
           return res;
}
```

Binary Trees

- Know :
  - pre-order
  - in-order
  - post-order

Minheaps

Show the state of the following minheap after performing the deleteMin operation. (Instead of writing this with a pen or pencil, typing the result in text similarly to how the drawing of the heap below was constructed will suffice.)

```
      14
    /     \
 22       72
 / \       / \
38 40 80 90
```

Solution:

```
      22
    /      \
 38      72
 / \       /
90 40 80
```

Grading:

1 pt for 22 being at the root

1 pt for 38 being left of root

1 pt for 90 being left of left of root

2 pts for 40, 72 and 80 being unchanged

Recursive Coding

Consider the problem of a frog jumping out of a well. Initially, the frog is n feet below the top of the well. When the frog jumps up, it elevates u feet. If a jump gets the frog to the top of the well or past it, the frog escapes the well. If not, unfortunately, the frog slips down by d feet before clinging to the side of the well. (Note that d < u.) Write a recursive function that takes in positive integers, n, u, and d, and returns the number of times the frog must jump to get out of the well. For example, if n = 10, u = 5 and d = 3, the function should return 4. On the first jump, the frog goes from 10 feet below the top to 8 feet below (5-3 is the progress). On the second jump, the frog goes from 8 feet below the top to 6 feet below the top. On the third jump, the frog goes from 6 feet below the top to 4 feet below the top. On the last jump, since 5 feet is enough to clear the top of the well, the frog does not slip down and gets out. In this case, had n = 11, the frog would have also gotten out in 4 jumps. (Note: Although one can do some math to arrive at an O(1) solution without recursion, please use recursion to simulate the jumping process described as this is what is being tested - the ability to take a process and express it in code, recursively. Also, though this is a toy problem, it's surprisingly similar to the real life process of paying off a loan, though in the latter process, the amount you "slip down" slowly decreases, month after month.)

```
int numJumps(int n, int u, int d) {
        if (u >= n) return 1; // 2 pts
                return 1 + numJumps(n-(u-d), u, d); // 3 pts
}
```

Grading: 1 pt for checking base case, 1 pt for the return in this case.
1 pt for adding one jump, 2 pts for the recursive call

Recursive PowerOfTwo

```
int countOfOnes(unsigned int n)
{
        if (n == 0)
                return 0;
        else
                return ((n&1) + countOfOnes(n>>1));
}
int powerOfTwo(unsigned int n)
{
        return (countOfOnes(n) == 1);
}
```

Tries

Given a dictionary of words stored in a trie rooted at root, and a string, str, consider the problem of determining the length of the longest prefix of str that is also a prefix of one of the words stored in the trie. You may assume that if a link in the trie exists, a valid word is stored down that path in the trie. You may use string functions as needed, but please try to do so efficiently. (One point will be deducted forinefficient use of a particular string function.) For example, if str = “capitulate” and the trie stored the setof words {“actor”, “bank”, “cat”, “capitol”, and “caption”}, then the function should return 5 since, the first five letters of “capitulate” are the same as the first five letters of “capitol”, and no other word stored in the trie shares the first six letters with “capitulate.” Complete the code for the function that solves this problem below. root is a pointer to the root of the trie and str is a pointer to the string. You may assume that at least one word is stored in the trie. The function signature and trie node struct definition are given below.

Note that due to the function signature, you must write your code iteratively.

```
#include <string.h>
typedef struct TrieNode {
        struct TrieNode *children[26];
        int flag; // 1 if the string is in our trie, 0 otherwise
} TrieNode;

int maxPrefixMatch(TrieNode* root, char* str) {
        // Grading: 2 pts – give 1 pt if called multiple times.
        int len = strlen(str);
        // Grading: 2 pts loop
        for (int i=0; i<len; i++) {
                // 3 pts – 2 pts for statement, 1 pt for placement (before NULL check)
                root = root->children[str[i]-'a’];
                // 2 pts – 1 pt for check, 1 pt for return
                if (root == NULL) return i;
        }
        // Grading: 1 pt
        return len;
}
```

Algorithm Analysis

What is the Big-Oh memory usage for the function call createNode(N)? Please provide your answer in terms of the input parameter, N. Please justify your answer by either evaluating an appropriate recurrence relation or summation.

```
typedef struct Node Node;
struct Node { Node ** children; int val; };
Node * createNode(int N)
{
  Node * res = (Node *) malloc(sizeof(Node));
  if (N == 0)
    return res;
  res->children = (Node **) malloc(sizeof(Node*) * N);
  res->children[0] = createNode(N / 2);
  res->val = 0;
  for (int i = 0; i < N; i++)
    res->val += i;
  return res;
}
```

Algorithm Analysis

- The amount of memory produced by the function ignoring any recursive call is O(N). Taking into account the recursive call we find that the memory created fits the following recurrence relation
  - T(N) = T(N/2) + O(N)
  - T(1) = O(1)
- Solving the recurrence relation we get the following,
  - T(N) = T(N/2) + N
  - T(N/2) = T(N/4) + N/2
  - T(N) = T(N/4) + N/2 + N
  - T(N/4) = T(N/8) + N/4
  - T(N) = T(N/8) + N/4 + N/2 + N
- after k iterations $$T(N) = T\left(\frac{N}{2^k}\right) + \sum_{i=0}^{k-1}\frac{N}{2^i} \le T(1) + N\left(\frac{1}{1-\frac{1}{2}}\right) = 1 + 2N = O(N)$$
- Alternatively, a student could recognize that at each level half as much memory will be allocated, so that
- ultimately, the amount of memory allocated will be roughly N + N/2 + N/4 + …, and solve the sum.
- Grading: 8 pts to get to the correct sum (any way), 2 pts to evaluate it and give the correct answer.
- If the sum is incorrect, then max credit is 7 pts. Give partial based on work and breakdown of # of
- nodes created. If recurrence relation method is followed, 4 pts for stating the recurrence, 2 pts for
- iteration, 2 pts for its solution.

Recurrence Relations

Using the iteration technique, just solve for the next two iterations of the following recurrence relation:
$T(n) = 3T(n-1) + n^2, for\ integers\ n > 0$
$T(0) = 1$
Your answers should be of the form
$T(n) = aT(n-2) + bn^2 - cn + d$ and
$T(n) = eT(n-3) + fn^2 - gn + h$, where a, b, c, d, e, f, g, and h are positive integers.

$T(n) = 3(3T(n-2) + (n-1)^2) + n^2$
$T(n) = 9T(n-2) + 3n^2 - 6n + 3 + n^2$
$T(n) = 9T(n-2) + 4n^2 - 6n + 3$
It follows that a = 9, b = 4, c = -6, and d = 3
Now, plug in one more iteration
$T(n) = 9(3T(n-3) + (n-2)^2) + 4n^2 - 6n + 3$
$T(n) = 27T(n-3) + 9n^2 - 36n + 36 + 4n^2 - 6n + 3$
$T(n) = 27T(n-3) + 13n^2 - 42n + 39$
It follows that e = 27, f = 13, g = -42 and d = 39

Grading: 2 pts first iteration, (give 1 pt if some terms correct)
3 pts second iteration (give 1 pt if 1 or 2 terms correct,
2 pts if 3 terms correct)
Take off 1 pt if they leave 1st unsimplified: $9T(n-2) + 3(n-1)^2 + n^2$
Take off 1 pt if they leave 2nd unsimplified: $27T(n-3) + 9(n-2)^2 + 3(n-1)^2 + n^2$

Bitwise Operators

There are 20 light switches, numbered 0 to 19, each which control a single light. Initially, all of the lights the switches control are off. There are several buttons. Each button toggles several switches, when pressed. For example, if a button toggles the switches 3, 5 and 9, then pressing the button changes the state of the three switches 3, 5 and 9, leaving the other switches in the same state. (So, if lights 3 and 5 were on and light 9 was off, after the button is pressed, lights 3 and 5 would be off and light 9 would be on.) Each button can be stored in a single integer, where the kth bit is set to 1 if that button toggles the kth switch and set to 0 if pressing the button doesn't affect the kth switch. For example, the button described would be stored as the integer 552 since 2^3 + 2^5 + 2^9 = 552. Write a function that takes in an array, buttons, storing the buttons to press and an integer len, representing the length of the array buttons and returns a single integer storing the state of the lights after each of the buttons has been pressed once, assuming that all of the lights were off before any of the button presses. The format for storing the state of the lights should be identical to the format of the buttons.

```
int pressButtons(int buttons[], int len) {
}
```

```
int pressButtons(int buttons[], int len) {

        int res = 0; // 1 pt
         for (int i=0; i<len; i++) // 2 pts
                 res ^= buttons[i]; // 6 pts (partial possible)
         return res; // 1 pt
}
```

Grading Partial: 1 pt update res, 3 pts XOR, 2 pts access buttons[i] If they use | instead of ^ (-2), & instead of ^ (-3), + instead of ^ (-4).

AVL Trees

(a) (4 pts) Insert the value 6 into the following AVL tree. Clearly show where the node is inserted initially and what the tree looks like after each rotation that takes place (if any).

```
       27
    /       \
 19          29
 /   \      /  \
11   23   28 30
 /  \
10  15
```

Solution: 6 is inserted as the left child of 10. This causes a single right rotation at node 19. During the rotation, 11 moves up to take the place of 19. Node 19 moves down, and the old right child of 11 becomes the new left child of 19.

```
     27                                27
   /      \                        /          \
 19      29                      11              29
 /  \    /   \                 /     \         /    \
11 23 28 30                  10      19      28     30
 / \                        /        /   \
10 15                      6        15     23
/
6
```

Grading: 1 pt for keeping 27, 1 pt for 11, 19 and 15

(b) (4 pts) List all the integer values that would cause a double rotation if inserted into the following AVL tree. If there are no such values, say so. (Assume we do not allow the insertion of duplicate values into our AVL trees.)

```
 24
 / \
 20 32
     /
   28
```

Solution: Inserting 29, 30, or 31 would cause a double rotation.
No other values would cause a double rotation.

// 1 pt for each correct answer 1 pt for not
// listing any extra

(c) (2 pts) In big-oh notation, what is the worst-case runtime of inserting a single element into an AVL tree that already has n elements?

Solution: O(log n) Grading: 2 pts all or nothing

Backtracking

Imagine a dating website that asks each user 20 yes/no questions and stores their answers in a single
integer in between 0 and 220-1, with the answer to the kth question stored in bit 2k-1
, with 0 representing no and 1 representing yes. (So for example, if one person’s answers to questions 1, 3 and 4 were yes and the rest of the answers were no, the integer 11012 = 13 would be used to represent her 20 answers to the questions.) Consider the problem of finding the “best match” for a client out of a list of prospective matches. We consider a match A for the client to be better than match B if she shares more answers on corresponding questions with A than B. Write a function that takes in an integer representing the client’s answers, an array of integers storing the answers of potential matches, and the length of that array, which returns the index into the array storing the best match for that client. If there are multiple best
matches, you may return the index associated with any of them. A function has been given that you may call in your code, if you find it useful.

int bestMatch(int client, int* matches, int length) {}

```
int bestMatch(int client, int* matches, int length) {
        int res = 0, best = -1, i;
        for (i=0; i<length; i++) {
                int tmp = 20 - count(client ^ matches[i]);
                if (tmp > best) {
                        best = tmp;
                        res = i;
                }
        }
        return res;
}
int count(int mask) {
        int res = 0, i;
        for (i=0; i<32; i++)
        if ((mask & (1<<i)) != 0)
                res++;
        return res;
}
```