CMPT125, Fall 2020 - Solutions

Final Exam
December 12, 2020

Name_____

SFU ID: |__|__|__|__|__|__|__|__|__|__|

Problem 1	
Problem 2	
Problem 3	
Problem 4	
TOTAL	

Instructions:

1. You should write your solutions directly in this word file, and submit it to Coursys before December 12, 10:00pm. Submitting a pdf is also ok.

2. No late submissions, no exceptions

3. Write your name and SFU ID on the top of this page.

4. This is an open book exam. You may use textbooks, calculators, wiki, stack overflow, geeksforgeeks, etc. If you do, specify the references in your solutions.

5. Discussions with other students are not allowed. Posting questions online asking for solutions is not allowed.

6. The exam consists of four (4) problems. Each problem is worth 25 points.

7. Write your answers in the provided space.

8. Explain all your answers. Really, explain all your answers.

Good luck!

Problem 1 [25 points]

a) Consider the following function.

```
int foo(unsigned int x) {
  if (x==0)
    return 0;
  if (x==1)
    return x+1;
  unsigned int half = x/2; // rounded down
  return foo(half) + foo(x-half);
}
```

[4 points] What will be the return value of foo(8)? Show some intermediate steps of the recursion.

Answer: 16
Proof: we have foo(1) = 2
foo(2) = foo(1) + foo(1) = 2+2 = 4
foo(4) = foo(2) + foo(2) = 4+4 = 8
foo(8) = foo(4) + foo(4) = 8+8 = 16

[5 points] Use big-O notation to express the running time of foo()? Explain your answer.

On input x>0 denote the running time by T(x).
Then T(x) = 2T(x/2) + O(1), T(1) =O(1)
Claim: T(x) = O(x)
Proof by induction: Let C be such that T(x) < 2T(x/2) + C, T(1) < C
We will prove that T(x) < 2C*x-C for all x

Base case: for x=1, we have T(1) < C =2C*1-C. So the base case is ok

Induction step: suppose T(i) < 2C*i-C for all i<x, and let's prove it for x.
For x>1 we have T(x) < 2*T(x/2) + 2C
By induction hypothesis we have T(x/2)<2C*x/2 - C.
Therefore, T(x) < 2*T(x/2) +C < 2*(2C*x/2 - C)+C = 2C*x-C, as required.

[4 points] Explain the functionality of the function foo.

For the base case of x=1 the function returns 2
For x>1 the function divides x by 2, applies recursion on each of the halves, and returns the sum of the obtained values.
By induction, foo(half) = 2*half, and foo(x-half) = 2(x-half).
Therefore, for x>=0 the function simply returns 2x

[4 points] Write an equivalent function (i.e., a function with the same functionality) more efficiently.

```
int foo(unsigned int x) {
    return 2*x;
}
```

For items b and c below use the following struct

```
typedef struct {
  int x,y;
} point;
```

b) Consider the following code

```
void bar(point* p1, point* p2) {
  point p={p1->x, p2->y};
  p.x = 5;
  p.y = 6;
  *p2 = p;
  p1 = &p;
}

int main() {
  point p0 = {0,0};
  point p1 = {1,2};
  bar(&p0, &p1);
  return 0;
}
```

[4 points] Will the code above compile properly? If yes, what will be the values of p0 and p1 in the end of main()? Explain your answer.

The local variable p is set to be {0,2}
Then it is set to be {5,6}
Then *p2 gets the same values {5,6} – this affects the argument p2 passed to bar()
Then, the local copy of p1 points to p – has no effect when the function returns
ANSWER: When returned p0 is still {0,0}, and p1 is {5,6}

c) Consider the following code.

```
point* create_point(int x,int y) {
  point p={x,y};
  point* ptr = &p;
  return ptr;
}
int main() {
  point* p1 = create_point(1,2);
  point* p2 = create_point(6,7);
  printf("%d, %d, ", p1->x, p1->y);
  printf("%d, %d \n", p2->x, p2->y);
  return 0;
}
```

[4 points] Will the code compile properly? If yes, what will be the output of the following code? Explain your answer.

Create_point returns a pointer to a local variable. Hence the behaviour is undefined.

It is likely that the function will print “6, 7, 6, 7”. But this is not guaranteed.

Problem 2 [25 points]

A node in a linked list of ints is a struct containing an int and pointer to the next element in the list.

```
struct LL_node {
  int data;
  struct LL_node* next;
};
typedef struct LL_node LL_node_t;
```

In all questions below a linked list is represented by a pointer to the first element (head) of the list.

a) [5 points] Write a function in C that gets a linked list of ints and a boolean predicate on ints, and returns the number of elements in the list on which pred outputs true.

```
int count(LL_node_t* head, bool (*pred)(int)) {
int ret = 0;
for (LL_node_t* cur = head; cur; cur = cur->next) {
 if (pred(cur->data))
   ret++;
 }
 return ret;
}
```

[2 points] What is the running time of your function?

We apply pred on each node->data. Hence the running time is O(length of the list)

b) [8 points] Write a function in C that gets a linked list of ints and checks if all even numbers in the list come before all odd numbers. The running time of the function must be O(length of list).

For example, the function returns true on the following inputs:

2 → 4 → 10 → 0 → -11
4 → 3 → 5
-2 → -4
1
-6

The function needs to return false on the following inputs:

-3 → 4 → 6 → -8
7 → 10
1 → 0 → 1 → 0 → 1

```
bool evens_before_odds(LL_node_t* head) {

// idea: we first skip all evens,
// and when we see the first odd number, we don't expect more evens
// if we see even after an odd, we return false
 LL_node_t* cur = head;
 while (cur && cur->data%2 ==0) // skip all evens
   cur = cur->next;
 // here cur is odd or null
 while (cur) {
   if (cur->data %2 == 0)
     return false;
   cur = cur->next;
 }
 // if reached here, return true
 return true;
}
```

c) [10 points] Implement in C the ***rearrange*** function on a Linked List.
The function gets a pointer to the head of the list, and a pivot.
It rearranges the list so that all elements < pivot appear before all elements >=pivot.
The function returns a pointer to the new head of the list.
For example, if the input list is 5→2→1→3→3→4 and pivot = 4, then the output can be any of the following:
(1) 1 → 2 → 3 → 3 → 4 → 5 or (2) 2 → 3 → 1 → 3 → 5 → 4 or (3) 3 → 3 → 2 → 1 → 4 → 5

*You don't have to implement the rearrange algorithm for quickSort we saw in class.

```
LL_node_t* rearrange(LL_node_t* head, int pivot) {

  LL_node_t* cur = head;
  LL_node_t* less = NULL; // pointer to the sublist with data<pivot
  LL_node_t* greater =  NULL; // pointer to the sublist with data>=pivot
  LL_node_t* tmp;
  while (cur) {
    tmp = cur->next;
    if (cur->data < pivot) { // add cur to the less list
      cur->next = less;
      less = cur;
    }
    else {  // cur->data >= pivot
      cur->next = greater; // add cur to the greater list
      greater = cur;
    }
    cur = tmp;
  }

  // here less is the sublist with all elements<pivot
  // greater is the sublist with all elements>=pivot
  // next we concatenate them
  if (less == NULL)
    return greater;

  // less is non-empty
  // we iterate to the end of the list less,
  // and the tail of the list to greater
  cur = less;
  while (cur->next != NULL)
    cur = cur->next;
  cur->next = greater;
  return less;




}
```

Problem 3 [25 points]

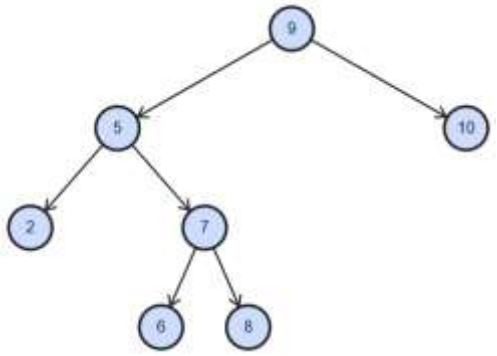
In this problem use the following struct for Binary Tree of ints.

```
struct BTnode {
  int value;
  struct BTnode* left;
  struct BTnode* right;
};
typedef struct BTnode BTnode_t;
```

a) [10 points] Write a function in C that gets a pointer to the root of a Binary Tree, and counts the number of nodes in even layers and the number of nodes in odd layers. In the tree below there are 3 nodes in even layers (9,2,7) and 4 nodes in odd layers (5,10, 6, 8).

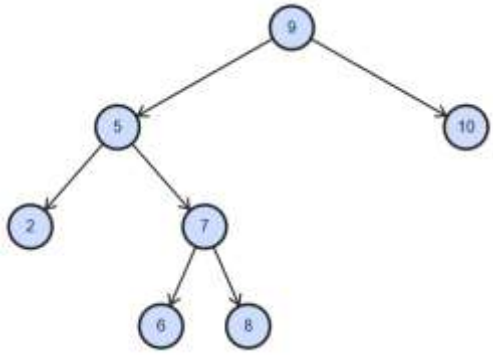
The output is stored in struct

```
typedef struct {
  int evens;
  int odds;
} pair;
```




```
pair count_evens_odds(BTnode_t* root) {
  // idea: apply recursion on left and right subtree
  // and use the obtained results to solve the problem


  if (root==NULL)
    return (pair){0,0};

  // apply recursion on left and right subtree
  pair l = count_evens_odds(root->left);
  pair r = count_evens_odds(root->right);

  // note that odd/even is switched for the children of root
  pair result = {1+l.odds+r.odds, l.evens+r.evens};
  return result;




}
```

b) [7 points] Write a function in C that gets the root of a Binary Tree and computes the sum of all numbers in all leaves of the tree. For example, for the tree below the function should output: 2+6+8+10=26



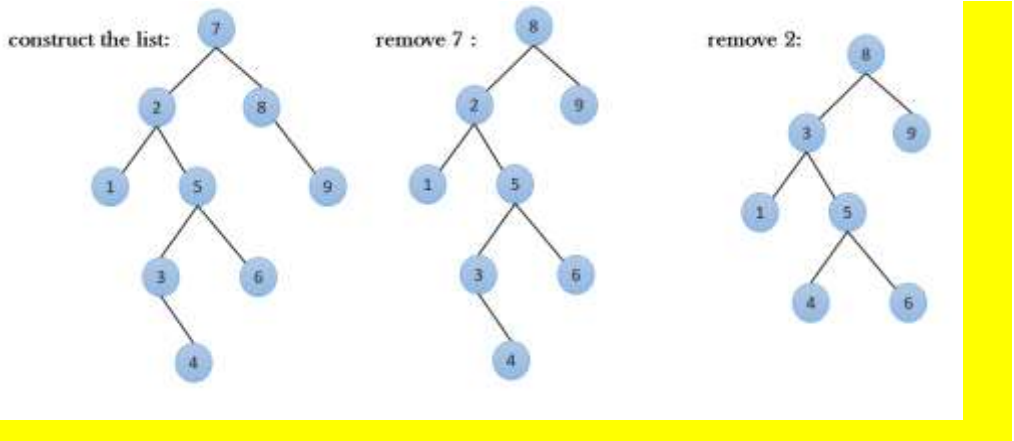

```
int sum_in_leaves(BTnode_t* root) {
  // apply recursion on left and right subtree
  // if root is a leaf, return the value
  // otherwise, compute the sum in left and right subtrees
if (root == NULL)
   return 0;

 if (root->left==NULL && root->right==NULL) // is leaf
   return root->value;
 else
   return sum_in_leaves(root->left) + sum_in_leaves(root->right);
}
```

b) [8 points total]

[4 pts] Draw the Binary Search Tree using the following sequence of insertions: 7,2,5,8,3,4,9,6,1.

[2 pts] Remove 7 from the resulting tree, and draw the new tree.

[2 pts] Remove 2 from the resulting tree, and draw the new tree.



Problem 4 [25 points]

BoundedMemoryStack is a variant of stack where the capacity of the stack is bounded by some parameter specified when creating the stack.
If the stack is full, and a new element is added, then the element that was added the earliest (the farthest in the past) is "forgotten", and the new item is added.

For example, suppose we have a stack with capacity 3. Consider the following sequence of operations
push(0) // stack is [0]
push(1) // stack is [1,0]
push(2) // stack is [2,1,0]
push(3) // stack is [3,2,1] // 0 is forgotten
pop() // -- stack is [2,1] -- returns 3
push(4) // stack is [4,2,1]
pop() // stack is [2,1] -- returns 4
pop() // stack is [1] -- returns 2
pop() // stack is empty -- returns 1

Implement this ADT in C++. Specifically, you need to implement the following methods:
Each method must run in O(1) time

```
class BoundedMemoryStack{
 public:
   BoundedMemoryStack(int capacity)
     : m_capacity(capacity), m_size(0), m_head(0)
   {
     m_storage = new int[capacity];
   }


   void push(int item) {
     m_storage[m_head] = item;
     m_head = (m_head+1) % m_capacity;
     if (m_size < m_capacity)
       m_size++;
   }


   int pop() {
     if (m_head>0)
       m_head--;
     else // m_head == 0;
       m_head = m_capacity-1;
     m_size--;

     return m_storage[m_head];
   }
```

```
   bool isEmpty() {
      return m_size==0;
   }

   ~BoundedMemoryStack() {
     delete[] m_storage;
   }

 private:
   int* m_storage;
   int m_capacity;
   int m_size;
   int m_head;
};
```