CMPT 125, Fall 2021

Final Exam - Solution
December 17, 2021

Name_________________________

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

Instructions:

1. Duration of the exam is 180 minutes.
2. Write your name and SFU ID **clearly**.
3. This is a closed book exam, no calculators, cell phones, or any other material.
4. The exam consists of four (4) problems. Each problem is worth 25 points.
5. Write your answers in the provided space.
6. There is an extra page at the end of the exam. You may use it if needed.
7. Explain all your answers.
8. Really, explain all your answers.

Good luck!

Problem 1 [25 points]

a) The function below gets as input an array of ints of length n..

```
void what(int* ar, int n) {
      if (n>1) {
          what(ar, n-1);
          char tmp = ar[n-2];
          ar[n-2] = ar[n-1];
          ar[n-1] = tmp;
      }
}
```

[6 points] What is the time complexity of what() as a function of n?. Use Big-O notation to express your answer. Explain your solution.

Answer: time complexity on an array of length n can be written as T(n) = O(1) + T(n-1). This behaves like T(n) = O(n)

[6 points] What is the effect of what() when applied on the array [1,2,3,4,5]? What is the functionality of what()? Explain your answer.

Answer: let's try to do do it backwards:

- On input [1] the array stays the same
- On input [1,2] the array becomes [2,1].
- On input [1,2,3] we make a recursive call on [1,2] and change it to [2,1], and then swap 3 and 1. So the result is [2,3,1].
- On input [1,2,3,4] we make a recursive call on [1,2,3] and change it to [2,3,1], and then swap 4 and 1. So the result is [3,2,4,1].
- On input [1,2,3,4,5] we make a recursive call on [1,2,3,4] and change it to [2,3,4,1], and then swap 4 and 1. So the result is [2,3,4,5,1].

In general, the function rotates the given array by one to the left. (Same logic as above)

b) [7 points] What will be the output of the C++ program below? Explain your answer.

```
#include <iostream>
using namespace std;

class Test {
  private:
    int m_x;

  public:
    Test() {
      m_x=0;
    }
    Test(int x) {
      m_x=x;
    }
    Test(Test& other) {
      this->m_x = other.m_x+1;
    }
    int getX() { return m_x; }
    void setX(int x) { m_x=x; }
};

int main() {
  Test t;  calls the default constructor Test()
  cout << "t = " << t.getX() << endl;
  t.setX(4);  calls the default Test(int x)
  cout << "t = " << t.getX() << endl;
  Test s(t);  calls the copy constructor
  cout << "s = " << s.getX() << endl;
  return 0;
}
```

Answer: the output is
t = 0
t = 4
s = 5

c) [6 points] Explain the difference between pointers and references in C++. Provide an example if needed.

Answer: they are very similar when passed as arguments to the function. In both cases they pass the address of the variable, and allow changing its value.
Difference are:

- A pointer can be NULL, reference must always refer to some variable
- A pointer can be reassigned to point somewhere else, reference cannot
- Pointers are a variable themselves (tamking space in the memory), references are not

Problem 2 [25 points]

A ***Doubly Linked List*** is a linked list where each node has a pointer to the next element and to the previous element.

```
struct DLL_node {
  int value;
  struct DLL_node* next;
  struct DLL_node* prev;
};
typedef struct DLL_node DLL_node_t;

typedef struct {
  DLL_node_t* head; // pointer to the first node
  DLL_node_t* tail; // pointer to the last node
} DLL_t;
```

Implement the following functions for *Doubly Linked List* of ints.
The running time in parts a-c of must be O(1).

a) [6 points] Write a function that adds a new node with a given value to the head of the list.

```
void add_to_head() {DLL_t* list, int value) {
  DLL_node* new_node = (DLL_node*) malloc(sizeof(DLL_node));
  if (!new_node) return; // malloc fail

  new_node->prev = NULL;
  new_node->next = list->head;
  new_node->value = value;
  list->head = new_node; // update the head pointer

  if (list->tail != NULL) // if list was not empty
    node_node->next->prev = new_node; // update the previous head
  else  // if list was empty
    list->tail  = new_node;
}
```

b) [6 points] Write a function that removes the first node of the list, and returns its value.

```
void remove_from_head() {DLL_t* list) {
  DLL_node* prev_head = list->head;
  list->head = prev_head->next; // update head

  if (list->head == NULL) // if list becomes empty, update the tail
       list->tail = NULL;
  else
       list->head->prev = NULL;

  free(prev_head);
}
```

c) [6 points] Write a function that removes a given node from the list.

```
// assumption node indeed belongs to the list
int remove_node() {DLL_t* list, DLL_node_t* node) {
   // update node->prev -- the vertex before node
   if (node->prev != NULL) // node is not the first in the list
       node->prev->next = node->next;
   else  // node is the first is in the list - update the head accordingly
       list->head= node->next;

   // update node->next  -- the vertex after node
   if (node->next != NULL) // node is the last in the list
       node->next->prev = node->prev;
   else  // node is last is in the list - update the tail accordingly
       list->tail = node->prev;


   int ret = node->value;

   free(node);

   return ret;
}
```

d) [7 points] Write a function that gets a Doubly Linked List and a predicate p. The function removes all nodes for which p(node->value)==false.
For example, if we apply it on the list *head→[1←→2←→5←→4←→8]←tail* with p=is_even(), then the remaining list should be *head→[2←→4←→8]←tail*.
*Remember to release the memory of the deleted nodes.

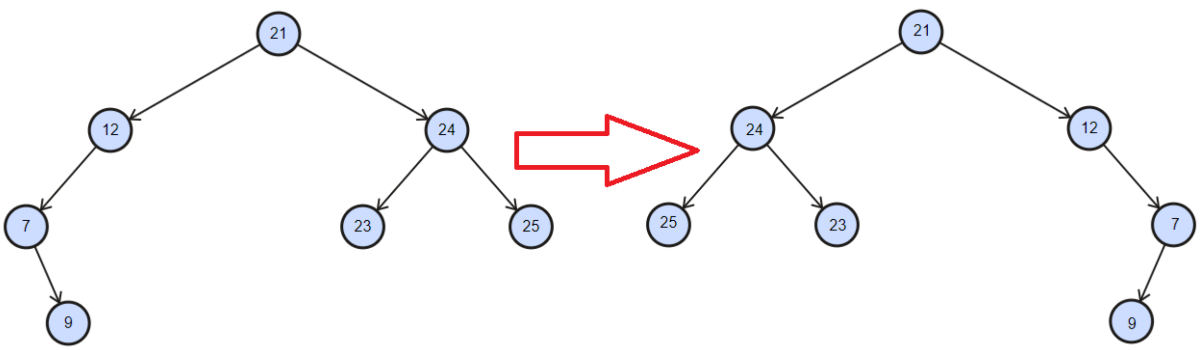
```
void filter() {DLL_t* list, bool(*p)(int)) {

   DLL_node* cur = list->head;
   DLL_node* next;

   while(cur) {
      next = cur->next; // save cur->next in case cur will be removed
      if ( !p(cur->value) )
          remove_node(list, cur);
      cur = next; // move forward in the list
   }



}
```

Problem 3 [25 points]

In this problem use the following struct representing a node in a Binary Tree of ints.

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

a) [7 points] Write an algorithm that gets a **Binary Tree** and converts it into its mirror reverse.
For example



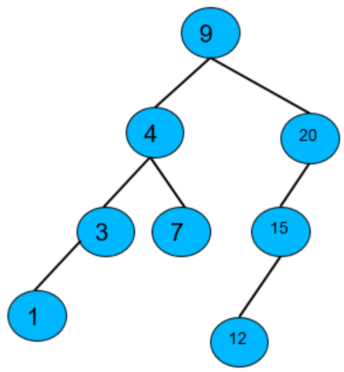

```
void mirror_tree(BTnode_t* root) {
  // idea: swap left and right child, and then apply recursion on each of the subtrees
  // it is possible to apply recursion first, and then swap the children

  if (!root)
      return;

  // swap the two children (possibly one of them is NULL)
  LL_node* tmp = root->left;
  root->left = root->right;
  root->right = tmp;

 mirror_tree(root->left);
 mirror_tree(root->right);
}
```

[3 points] Explain the running time of your function.

Each node is processed exactly once for O(1) time.

Therefore, the total time is O(size of tree)

b) [12 points] Write a function in C that gets a pointer to a node in a **Binary Search Tree**, and finds its predecessor. If the node is the minimal element in the tree, the function will return NULL. In the tree below: the predecessor of 4 is 3, the predecessor of 12 is 9, the predecessor of 7 is 4.




```
BTnode_t* find_predecessor(BTnode_t* node) {
  // idea: if node has a left child, then the predecessor is the largest node in the node->left.
  // otherwise, we can find the predecessor by going up “to the left”.

    if (node==NULL)
        return NULL;

    if (node->left) { // node has a left child
        // find maximal node in the subtree node->left
        BTnode_t* cur = node->left;
        while (cur->next != NULL)  // iterate to the max in the subtree of node>tree
            cur = cur->right;
        return cur;
    }
    else { // node doesn’t have a left child
        BTnode_t* cur = node;
        while (cur->parent && cur==cur>parent->left) // while cur is the left child of it parent
            cur = cur->parent;
         // here cur is either the right child of its parent or the root
        return cur->parent;
    }


}
```

[3 points] Explain the running time of your function.

Answer: each of the loops goes either up the tree or down the tree once.
Therefore, the total running time is at most O(depth of tree)

Problem 4 [25 points]

a) [12 points] Write a function that gets an array of ints of length n, and outputs the length of the longest contiguous increasing subsequence in O(n) time. For example, on input [6,2,5,3,6,8,9,1] the answer should be 4, as the longest increasing subsequence is [3,6,8,9].
* If your solution runs in quadratic time or slower, you will get 8 points.

```
int longest_incr_subsequence(const int* arr, int n) {

  // idea: iterate through the array if arr[i]>arr[i-1], increase the count. Otherwise, reset count.
  // update max if needed

    if (n == 0)
        return 0;

    int i;
    int count = 1;
    int max = 1;

    for (i=1; i<n; i++) {

        // we allow equalities or strictly increasing - both versions accepted for full marks
        if( arr[i] >= arr[i-1] ) {
            count++;
            if (count>max) // can be made more efficient, but it doesn't affect O() complexity
                max = count;
        }

        else  // reset count to 1, because
            count = 1;

    }
    return max;



}
```

[3 points] Explain the running time of your function.

We have a for loop with O(1) operations in each iteration. Therefore, the total time is O(n).

b) [10 points] Write a function that gets an array of digits (given as ints) of length n, and returns a string containing the largest number possible using these digits. For example:

- on input [6, 7, 5, 3, 8, 3, 0, 0] the function should return “87653300”.
- on input [1, 2, 3, 4, 4] the function should return “44321”.
- on input [0, 0, 0, 0, 0, 0] the function should return “0”

* You may assume all numbers in the array are between 0 and 9.

* Note: you need to return the number in a string because it may be too large to fit in an int/long.

** Remember to use dynamic memory allocation properly.

```
char* print_max_number(const int* arr, int n) {
  // idea: count the number of 0's, 1's, 2's....and create a string using this statistics

   int count_digits[10]; // count_digits is created on the stack - no need to free it in the end
   int i,j,k;

   for (i=0 ; i<10 ; i++)
       count_digits[i] = 0; // important to initialize;

   for (k=0 ; k<n ; k++)
       count_digits[arr[k]]++;
  // here count_digits[i] contains the number of i's in the array;

   if ( count_digits[0] == n ) { // edge case of all zeros
       char* ret = (char*) malloc(2);
       ret[0] = '0';   ret[1] = '\0';
       return ret;
   }

   char* max_number = (char*) malloc(n+1); // +1 for string terminator '\0'

   // populate max_number - probably there is a cleaner way to write it
   k = 0;
   for (i=9; i>=0; i--)
       for (j=0 ; j<count_digits[i] ; j++) {
          max_number[k] = '0'+ i;
          k++;
       }
   max_number[n] = '\0';

   return max_number ;
}
```