Name: ID#

COMPUTING II
FINAL EXAM

The total number of points is 110. You are not allowed to use any books, notes, calculator, or electronic devices. Write your answer carefully and clearly. Incorrect answers will receive little to no points. When you are asked to write a program the program should be clear and include indentations and comments to make it easier to read. ***Be sure to state any assumptions on which you have based your answers.***

Problem Number(s)	**Possible Points**	**Earned Points**
1	15	
2	10	
3	24	
4	6	
5	8	
6	15	
7	8	
8	6	
9	10	
10	8	
	TOTAL POINTS 110	

Sorting Algorithms

1. Define the following sorting algorithms using ***text*** and/or ***pseudo code***.
   i) Merge Sort **(5 points)**

https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/tutorial/

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.
Idea:
Divide the unsorted list into sublists, each containing element.
Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. will now convert into lists of size 2.
Repeat the process till a single sorted list of obtained.
While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

ii) Heapsort**(5 points)**

A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a heap. The heap itself has, by definition, the largest value at the top of the tree, so the heap sort algorithm must also reverse the order. It does this with the following steps:
1. Remove the topmost item (the largest) and replace it with the rightmost leaf. The topmost item is stored in an array.

2. Re-establish the heap.

3. Repeat steps 1 and 2 until there are no more items left in the heap.

The sorted elements are now stored in an array.

A heap sort is especially efficient for data that is already stored in a binary tree. In most cases, however, the quick sort algorithm is more efficient.

iii) Insertion Sort**(5 points)**

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.

It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

Quicksort

2. Quicksort is an efficient sorting algorithm based on partitioning array data into smaller arrays with a reference to a pivot element, i.e., after an initial pivot element is chosen, the resulting array is such that elements before the pivot are either larger or smaller than the pivot and vice-versa on the other side. There are various techniques to divide (partition) the array with reference to the chosen pivot.

   i) Assuming that the first element of an array is chosen as the pivot, describe a strategy to partition the array such that elements before the pivot are smaller than those on the right **(5 points)**. Please use clear plain English possibly written point by point or use pseudocode.

https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/tutorial/

Step 1 Choose the highest index value has pivot
Step 2 Take two variables to point left and right of the list excluding pivot
Step 3 left points to the low index
Step 4 right points to the high
Step 5 while value at left is less than pivot move right
Step 6 while value at right is greater than pivot move left
Step 7 if both step 5 and step 6 does not match swap left and right
Step 8 if left right, the point where they met is new pivot

   ii) Based on your strategy in (i) above, show the result of partitioning the array below by taking the **N** at the left as the pivot (**5 points**).

N	E	W	P	A	R	T	I	T	I	O	N	Q	U	E	S	T	I	O	N

Linked Lists

3. For this question please ***do not add code*** to test if `malloc` ***fails.***

Given the following structure and assuming a `Node` pointer called head is initialized as `NULL`:

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

a. Write a function `head_insert` that given a pointer to the head node pointer and an integer item will insert the item into the list using head insertion (**8 points**):

```
void head_insert(Node** head, int item)
```

```
void head_insert(Node** pHead, int item)
{
    Node* temp = (Node*)malloc(sizeof(Node));
    if (temp == NULL){
        printf("Failed to allocate memory for a new node.\n");
        exit(1);
    }
    temp->data = item;
    temp->next = *pHead;
    *pHead = temp;
}
```

b. Write a recursive function **`tail_insert`** that given a pointer to the head node pointer and an integer item, will insert the item into the list using tail insertion(**8 points**).

```
void tail_insert(Node** head, int item)
```

```
// Function to insert a new node at the
// end of linked list using recursion.
void tail_insert(Node** head, int item)
{
    // If linked list is empty, create a
    // new node (Assuming newNode() allocates
    // a new node with given data)
    if (head == NULL)
        head->data = item;
        head->next = NULL;

    // If we have not reached end, keep traversing
    // recursively.
    else
        head->next = tail_insert(head->next, data);

    return;
}
```

c. Write an iterative (***nonrecursive***) function **`tail_insert`** that given a pointer to the head node pointer and an integer item, will insert the item into the list using tail insertion (**8 points**):

```
void tail_insert(Node** head, int item)
```

```
void tail_insert(Node** head, int item)
{
    if (*head == NULL){
        *head = (Node*)malloc(sizeof(Node));
        (*head)->data = item;
        (*head)->next = NULL;
    }
    else {
        Node* temp = *head;
        while (temp->next != NULL){
            temp = temp->next;
        }
        temp->next = (Node*)malloc(sizeof(Node));
        temp->next->data = item;
        temp->next->next = NULL;
    }
}
```

Trees

4. (6 points) What are the main properties of:

a. Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

The left subtree of a node contains only nodes with keys lesser than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
The left and right subtree each must also be a binary search tree.

b. Max Heap

A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

the min-heap property: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
the max-heap property: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.`

c. AVL Tree

AVL tree is a type of binary search tree in which at any given node, absolute difference between heights of left sub-tree and right sub-tree cannot be greater than 1. This property of the AVL tree helps to keep the tree height balanced.

Binary tree

5. (8 points) Given the following preorder and inorder traversals for an unknown binary tree, determine the exact tree that would generate these traversals and then draw that tree. Once you have generated the tree be sure to check your work.
   Preorder: {7, 10, 6, 1, 5, 9, 2, 12}
   Inorder: {6, 10, 1, 7, 9, 12, 2, 5}

Binary tree

6. Given the following binary tree and the binary tree node, implement a recursive function, **`identical_trees_nodes()`**, that checks if two trees are identical (**15 points**).

```
typedef struct binaryNode BinaryNode, *BinaryNode_Ptr;
struct binaryNode
{
      BinaryNode_Ptr left;
      BinaryNode_Ptr right;
      int item;
};
struct binaryTree
{
      BinaryNode_Ptr root;
};
typedef struct binaryTree BinaryTree, * BinaryTree_Ptr;
```

The function identical_trees will call the recursive function identical_trees_nodes

```
int identical_trees(BinaryTree_Ptr tree1, BinaryTree_Ptr tree2)
{
      return identical_trees_nodes(tree1->root, tree2->root);
}
```

You will need to write ***the function declaration*** and ***definition*** for **`identical_trees_nodes()`**.

```
int identical_trees_nodes(node* a, node* b)
{
    /*1. both empty */
    if (a == NULL && b == NULL)
        return 1;
    /* 2. both non-empty -> compare them */
    if (a != NULL && b != NULL)
    {
        return
        (
            a->data == b->data &&
            identical_trees_nodes(a->left, b->left) &&
            identical_trees_nodes(a->right, b->right)
        );
    }

    /* 3. one empty, one not -> false */
    return 0;
}
```

Max Heap

7. Given the following array of numbers; show the final max-heap and the final corresponding array resulting from using the fix-down or heapify approach. Show all work **(8 points).**

1	10	3	40	12	2	33	11

AVL Tree

8. Assuming the following values arrive in the order they appear and are inserted into a binary search tree implemented as an AVL tree, show the resulting tree. Show all work for partial credit opportunities (**6 points**).

   1, 2, 3, 7, 6, 5

Dynamic median-finding

9. You need to support a client that reads a huge stream of numbers that are all different and needs to keep track of the median element in the entire stream seen so far. For example, if the client gives you the numbers 2 9 7 4 1 and then asks for the median, you must return 4, and if the client then adds the numbers 6 8 5 and again asks for the median, you must return 5 or 6. There are three requirements: First, you have only constant extra space (beyond what is needed to store the numbers themselves). Second, you must return the median in constant time. Third, you must process the Nth element in time proportional to log N.

   i) Is it possible to discard some portion of the input, such that your algorithm finds the median element accurately even in the future? **(1 points)**

   ii) Assume that you have seen N numbers and know that the median is of those numbers is v. Which of the following is true of the median when you process the (N+1)st number? **(2 points)**

      A. It does not change.
      B. It is the largest of the numbers smaller than v.
      C. It is the smallest of the numbers larger than v.
      D. Either A, B or C
      E. Either B. or C, but not A.
      F. It could be any of the numbers seen so far.

   iii) Which data structure can support inserting numbers in logarithmic time and returning the maximum in constant time, using only a constant amount of extra space (beyond what is needed to store the numbers)? **(2 points)**

   iv) Describe how this problem can be solved. Just use plain English to describe how you would go about solving it **(5 points).**

TRUE/FALSE

10. Answer the following questions with either true or false. Assume there are n elements in the data structure. No explanation necessary **(8 points)**

i) One can implement a stack based on a linked list so that ***each individual*** push/pop operation is time O(1).

ii) One can implement a stack (of unbounded size) based on an array so that each individual push/pop operation is time O(1).

iii) One can reverse the order of the elements in a linked list in time O(n).

iv) It is possible to append two linked lists in time O(1).

v) Adding an element to a heap has worst-case time complexity ***O(log(n)).***

vi) In a circular doubly linked list with 10 nodes, we will need to change 4 links if we want to delete a node other than the head node.

vii) Returning the maximum element in a max-heap (but not deleting it from the heap) can be done in time O(1).

viii) When we use a max heap to implement a priority queue, the time complexity of both the add and delete operations are O(n).

SCRATCH PAD