CMPSC24 Midterm-II
Spring 2018

This exam is closed book, closed notes. You may use a one-page A4 size paper containing your notes. Write your name on the your notes paper and all other sheets of the exam. No calculators or phones are allowed in the exam. **Write all your answers on the answer sheet. All exam sheets, notes paper and scratch paper must be turned in at the end of the exam.**

By signing your name below, you are asserting that all work on this exam is yours alone, and that you will not provide any information to anyone else taking the exam. In addition, you are agreeing that you will not discuss any part of this exam with anyone who is not currently taking the exam in this room until after the exam has been returned to you. This includes posting any information about this exam on Piazza or any other social media. Discussing any aspect of this exam with anyone outside of this room constitutes a violation of the academic integrity agreement for CMPSC24.

Signature: _____

Name (please print clearly):_____

Umail address: _____@umail.ucsb.edu

Perm number: _____

You have 75 minutes to complete this exam. Work to maximize points. A hint for allocating your time: if a question is worth 10 points, spend no more than 5 minutes on it if a question is worth 20 points, spend no more than 10 minutes on it etc. If you don't know the answer to a problem, move on and come back later. Most importantly, stay calm. You can do this.

**WRITE ALL YOUR ANSWERS IN THE PROVIDED ANSWER SHEET IN PEN OR DARK PENCIL!**

**DO NOT OPEN THIS EXAM UNTIL YOU ARE INSTRUCTED TO DO SO.**
**GOOD LUCK!**

Name: _____

**Part 1 [20 points] Linked-list and run time analysis**

Consider the definition of the class **`LinkedList`** used in the construction of a singly linked-list containing integer elements.

```
class LinkedList {

public:
    // ASSUME ALREADY IMPLEMENTED
    LinkedList(){head = 0; tail = 0;}     //Default constructor
    LinkedList(const LinkedList& source); //Copy constructor
    LinkedList& operator=(const LinkedList& source); //Copy assignment
    void append(int value); // Append to the end of the list

    // YOU MUST IMPLEMENT THESE:
    ~LinkedList();
    void insertFront(int value);  //insert value to the front

private:
    class Node{
       public:
            int data;          // data element
            Node* next;        // pointer to the next node in the list
            Node (const int& d) { data = d; next = 0;}
           ~Node(); //Implement the destructor
    };
    Node* head; // pointer to the first node in the linked-list
    Node* tail; // pointer to the last node in the linked-list
};
```

Assume that the following methods have already been implemented

```
    LinkedList(){head = 0; tail = 0;}     //Default constructor
    LinkedList(const LinkedList& source); //Copy constructor
    LinkedList& operator=(const LinkedList& source); //Copy assignment
    void append(int value); // Append to the end of the list
```

You may use any of the above in your implementation of the functions described on the next page. In each case derive the big-O running time of your implementation as a function of the number of nodes in the linked list, denoted as N

Name: _____

1. **(10 pts)** Implement the **`insertFront()`** method of the LinkedList class and derive the Big-O running time for your implementation. The **`insertFront()`** method takes an integer value and adds a Node with that value to the front of the linked list. The method should handle all edge cases including insertion into an empty list. Point distribution: 6 points for correctness, 2 pts for code that is readable and concise, 2 pts for deriving the Big-O running time of your implementation.

2. **(6 pts)** Implement the **`destructor`** of the **`LinkedList`** and **`Node`** classes. You may use a recursive or iterative implementation. What is the Big-O running time of your implementation of the destructor of the LinkedList?
Point distribution: 2 points for correctness of each destructor, 1 pt for code that is readable and concise, 1 pt for the Big-O running time of your implementation.

3. **(4 pts)** Consider the following two variations on the implementation of the **`append()`** method, that inserts a node at the end of the linked list. For each implementation derive the Big-O running time of in terms of the number of nodes (N) in the linked list. Provide the tightest bound.

**(a) Implementation #1**

```
LinkedList::append(int value){

      if(head == 0){
            head = new Node(value);
            tail = head;
      } else{
            tail->next = new Node(value);
            tail = tail->next;
      }

}
```

**(b) Implementation #2**

```
LinkedList::append(int value){
      if(head == 0){
            head = new Node(value);
            tail = head;
      } else{
           Node* tmp = head;
           while(tmp->next)
              tmp=tmp->next;
            tmp->next = new Node(value);
            tail = tmp;
      }
}
```

Name: _____

**Part 2: [10 points] More running time analysis**

1. **[5 pts]** Find the Big-O running time of each of the following code as a function of the input N. Justify your answer

**i.[ 2pts]**

```
int x = 10;
for(int i=1; i<N; i+=2) {
      for(int j = N; j>0; j--) {
            x++;
      }
}
```

**ii. [3 pts]**

```
int x = 10;
for(int i=1; i<N; i+=2) {
      for(int j = N; j>1; j=j/2) {
            x++;
      }
}
```

2. **[ 2 pts]** Given below is the number of primitive operations needed for two algorithms in terms of the size of the input `N` . What is the Big-O run time in each case?

i. Algorithm A: $50*N*\log(N) + 10*2^N + 250*\log(N^2)$

ii. Algorithm B: $2^{10000} + 500*N^2 + N$

3. **[3 pts]** For the following questions select the operation that is "faster" based on its Big-O running time? *Write A, B or C in your answer sheet in each case*

i. (1 pt) **Deleting a value**:

A. **Deleting** a value at the head of an unsorted linked-list
B. **Deleting** a value from a sorted array
C. Both are equally fast

ii. (1 pt) **Searching** for an element:

A. **Searching** for an element in a balanced BST
B. **Searching** for an element in a sorted array using binary search
C. Both are equally fast

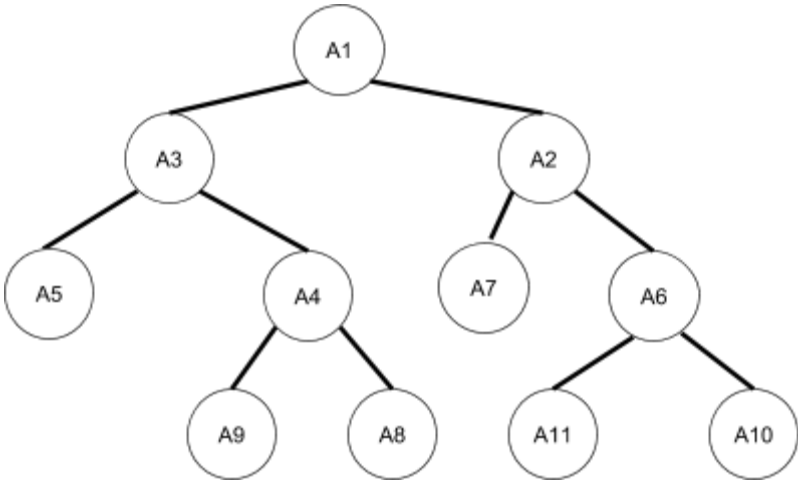
iii. (1 pt) **Finding the minimum** element:

A. **Finding the minimum element** in an sorted array that is either in ascending or descending order
B. **Finding the minimum element** in a balanced BST
C. Both are equally fast

Name: _____

**Part 3 [20 points+10 points extra credit] Binary Search Trees**

**1.[10 points]** Consider the following BST with keys labeled A1 through A11. The labels are not the actual key values. Use the properties of BSTs to answer the following questions:




i. (1 pt) Is A2 < A1 ? (Yes/No/Cannot be determined)
ii. (1 pt) Is A8 < A2 ? (Yes/No/Cannot be determined)
iii. (1 pt) What is the key with the minimum value?
iv. (1 pt) What is the key with the maximum value?
v. (1 pt) Which key is the successor of A2?
vi. (1 pt) Which key is the successor of A8?
vii. (4 pts) Write the output of printing the keys in the BST when the tree is traversed using an:
    a. Inorder traversal
    b. Postorder traversal

**2. [10 points]** Draw a Binary Search Tree that results from inserting the following integers starting with an empty tree. Insert the keys in the order provided: 500, 40, 20, 30, 10, 50, 600, 200

Name: _____

**3. [Extra credit: 10 pts] Consider the following definition of a Node in a binary search tree and a class BST that represents a binary search tree.**

```
class Node{
  public:
    int data; // data element
    Node* left; // pointer to the left child
    Node* right; //pointer to the right child
    Node* parent; //pointer to the parent node
    Node(const int& d){ data = d; left = 0; right = 0; parent=0;}
};

class BST{
  public:
    BST(){root = 0;}
   ~BST();  // destructor
    bool insert(const int& value);
    int getHeight() const;

private:
    Node* root; // pointer to the root of the tree

};
```

**[10 pts] Implement the method `getHeight()` of BST. This method returns the height of the BST. Recall that the height of an empty BST is -1, the height of a BST with just one node is 0. You must write your own private helper function that uses recursion to calculate the height of any BST. Provide the implementation of `getHeight()` and the recursive helper function. The complexity of your algorithm should not be worse that O(N), where N is the number of nodes in the BST**

Point distribution: 2 points for public method, 5 pts for correctness of recursive helper, 2 pts for code that is readable and concise, 1 pts for an implementation with running time not worse that O(N).

**THE END**

Name: _____

**Scratch Paper**