EECE.3220: Data Structures

Spring 2019

Exam 3
May 8, 2019

Name: _____

For this exam, you may use two 8.5" x 11" double-sided pages of notes. All electronic devices (e.g., calculators, cell phones) are prohibited. If you have a cell phone, please turn off your ringer prior to the start of the exam to avoid distracting other students.

The exam contains 3 sections for a total of 100 points, as well as a 10 point extra credit section. Please answer all questions in the spaces provided. If you need additional space, use the back of the page on which the question is written and clearly indicate that you have done so.

Please read each question carefully before you answer. In particular, note that:

- For all questions requiring you to write a function (1b, 1c, 2, 3b), please write all code necessary to complete the function. I have written some of the lines for you. Fill in your code in the blank lines and open spaces provided.
- Please read the multiple choice questions carefully. Each of the questions (1d, 4a, 4b, 4c, 4d) may have more than one correct answer.

You will have three hours to complete this exam.

S1: Array-based queues	/ 37
S2: Linked queues	/ 20
S3: Linked lists	/ 27
S4: Multiple choice	/ 16
TOTAL SCORE	/ 100
EXTRA CREDIT	/ 10

1. (37 points) ***Array-based queues***

a. (13 points) For the short program shown below, list the output exactly as it will appear on the screen. Be sure to clearly indicate spaces between characters when necessary.

You may use the available space to show your work as well as the output; just be sure to clearly mark where you show the output so that I can easily recognize your final answer.

This problem doesn't strictly require a specific queue implementation—you just need to understand queue operations to solve the problem.

```
int main() {
   Queue Quahog, Quincy;
   double arr[9] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
   int i;

   for (i = 0; i < 5; i++) {
      Quahog.enqueue(arr[i]);
      Quincy.enqueue(arr[8 - i]);
   }

   i = 10;
   while (!Quahog.empty() && !Quincy.empty()) {
      cout << Quahog.getFront() << " "
           << Quincy.getFront() << endl;
      if (i % 2 == 0)
         Quahog.dequeue();
      else
         Quincy.dequeue();
      i--;
   }

   if (Quahog.empty())
      cout << "Quahog empty" << endl;
   if (Quincy.empty())
      cout << "Quincy empty" << endl;

   return 0;
}
```

1 (continued)

This section uses the array-based Queue class defined on the extra sheet for all problems.

Complete the following functions, which you could add to the array-based `Queue` class covered in class (and shown on the extra handout).

You cannot modify the class definition. Each `Queue` object has the data members `list`, `front`, `back`, and `cap`, and assume `front` and `back` are both initialized to 0. So, the queue is empty when `front` and `back` are equal and full when `back` is one spot "behind" `front`.

b. (10 points) `unsigned Queue::numVals()`: Return the number of values currently stored in the queue.

```
unsigned Queue::numVals() {









}
```

c. (10 points) `void Queue::display(ostream &out)`: Print all values in the queue to the output stream `out`, starting with the value at the front of the queue and ending with the value at the back of the queue. The function should print a space after each value and a new line at the end of its output.

```
void Queue::display(ostream &out) {









}
```

1 (continued)

This section uses the array-based Queue class defined on the extra sheet for all problems.

d. (4 points) We discussed a few different ways of initializing the `front`/`back` index variables in an array-based queues. Which of the following choices show valid initial values for those indexes, assuming the rest of the functions are updated (if necessary) to make everything consistent. **<u>This question may have more than one correct answer—choose all answers you believe are correct.</u>**

i. `front = 0, back = 0`

ii. `front = -1, back = -1`

iii. `front = -1, back = cap – 1`

iv. `front = cap – 1, back = cap – 1`

v. The array-based queue lectures were so confusing that I have no idea how to answer this question.

2. (20 points) ***Linked queues***

This question uses the linked Queue class defined on the extra sheet.

Complete the following function, which you could add to the linked `Queue` class covered in class (and shown on the extra handout):

`bool Queue::isPriorityQ()`: Returns true if the data in the queue is ordered from highest to lowest, thus behaving like a priority queue, and false otherwise. Should return false if the queue is empty.

```
bool Queue::isPriorityQ() {
   Node *p;              // Pointer to current node
   QueueElement val;     // Value to compare to next node

   // Return false if queue is empty




   // Otherwise, traverse queue and compare consecutive values
   // If all pairs of values are in order, queue is in order




   while (_____________________________________________________) {

      // If out-of-order pair found, queue isn't in order




      // Otherwise, update variables to move to next node




   }

   // If you get through whole queue, must be ordered


}
```

3. (27 points) ***Linked lists***

a. (12 points) For the short program shown below, list the output exactly as it will appear on the screen. Be sure to clearly indicate spaces between characters when necessary.

You may use the available space to show your work as well as the output; just be sure to clearly mark where you show the output so that I can easily recognize your final answer.

This problem uses the linked list class defined on the extra sheet, while either queue implementation we discussed will work. Assume that the insert() function adds data to the linked list so the nodes are ordered from least to greatest value.

```
int main() {
   LList L1;
   Queue Q1;
   double v = 0.1;
   int i;

   for (i = 0; i < 5; i++) {
      L1.insert(v);
      Q1.enqueue(v);
      v = v * -2;
   }

   while (!Q1.empty()) {
      cout << Q1.getFront() << " ";
      L1.insert(Q1.getFront() * 2);
      Q1.dequeue();
   }
   cout << "\n";
   L1.display(cout);     // Prints list contents from first to
                         //   last node

   return 0;
}
```

3 (continued)

This question uses the linked list class defined on the extra sheet.

b. (15 points) Complete the following function, which you could add to the `LList` class shown on the extra handout:

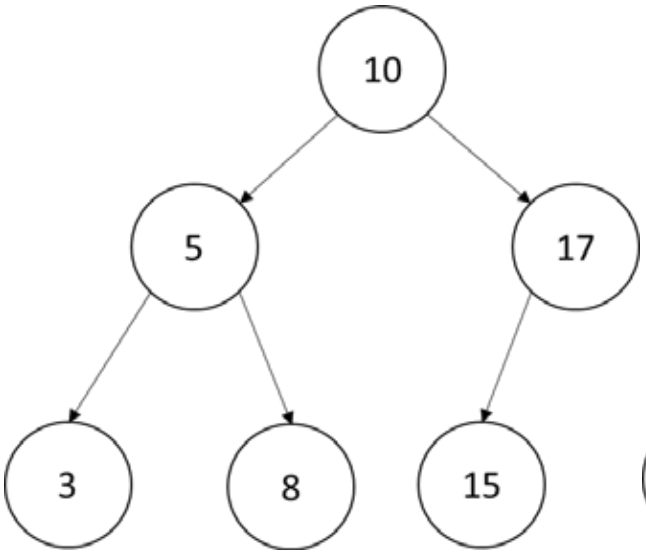
`bool LList::inList(double val)`: Returns true if a node in the list contains the value `val` and false otherwise. For full credit, your implementation should search the list until it either finds the desired value or reaches a point where the value cannot possibly be in the list, assuming the list is ordered from least to greatest value.

```
bool LList::inList(double val) {
   Node *p;          // Pointer to current node









}
```

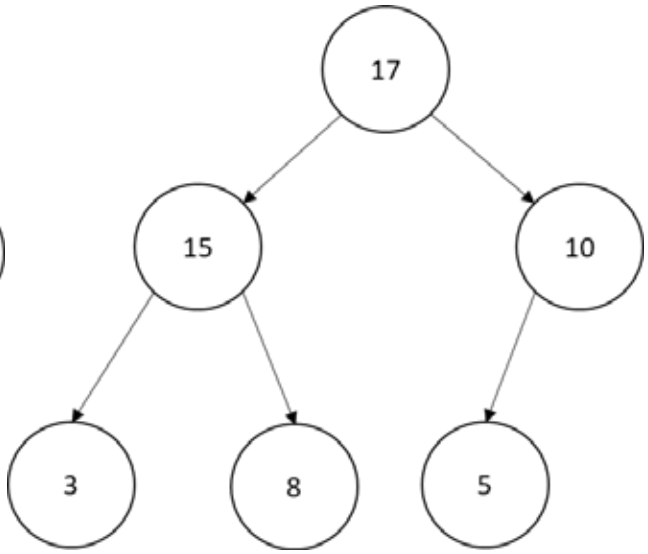
4. (16 points, 4 points each) ***Multiple choice***

a. Which of the trees below are binary search trees? **Circle ALL correct answers—there may be more than one.**

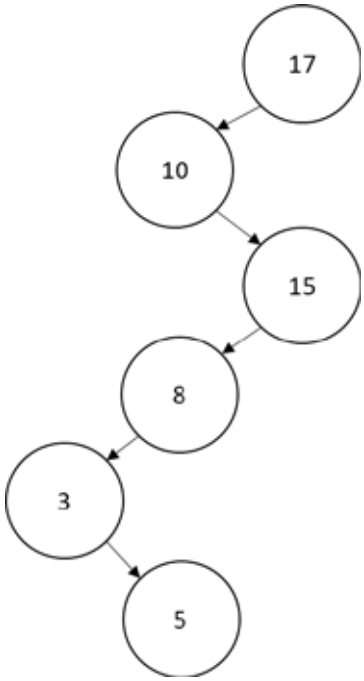
i.



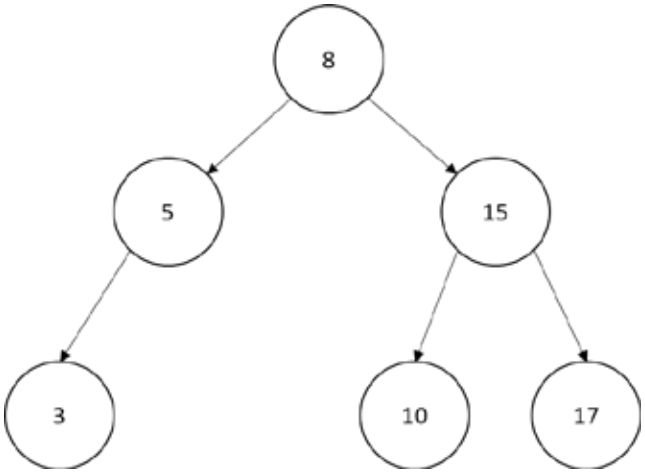
ii.



iii.



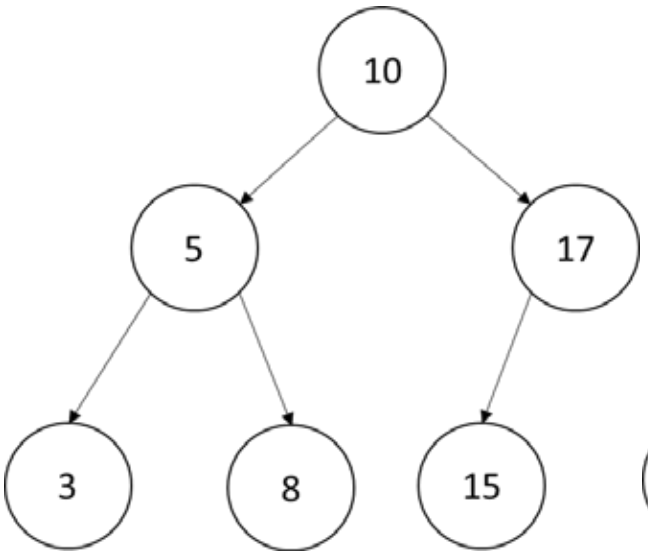
iv.



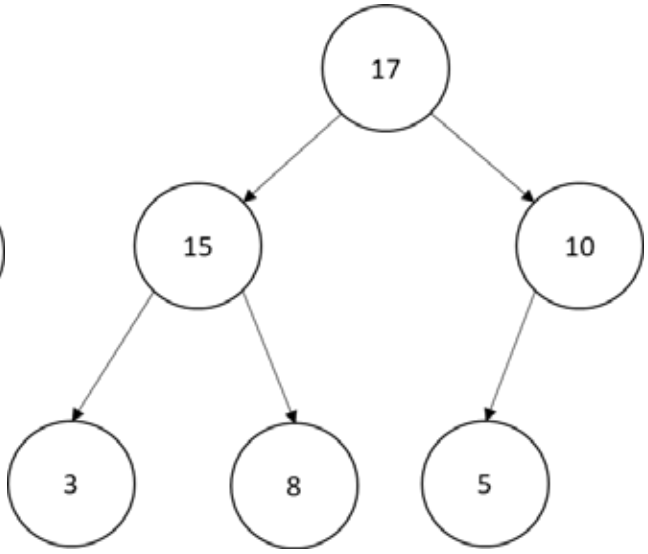
4 (continued)

b. Which of the trees below are heaps? **Circle ALL correct answers—there may be more than one.**

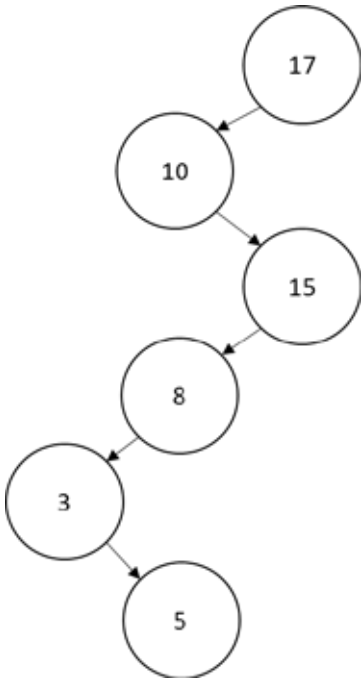
i.



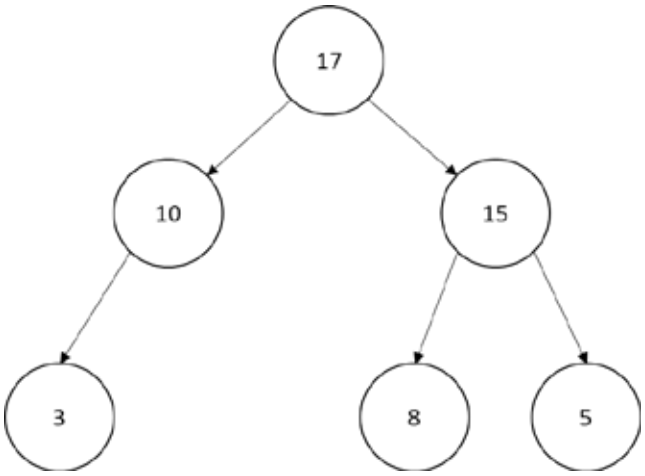
ii.



iii.



iv.



4 (continued)

c. Which of the following priority queue functions can be implemented using heap functions? **Circle ALL correct answers—there may be more than one.**

i. Insert: add an item to the priority queue such that, if it has a higher priority than other items, it will be removed from the priority queue before those items.

ii. Remove: remove the highest priority item from the priority queue.

iii. Change priority: adjust the priority of an item, then reorder the queue appropriately so that the item will be removed according to its new priority, not its old priority.

iv. Join: given two priority queues, merge them together to create one priority queue.

d. Which of the following statements about hash tables are true? **Circle ALL correct answers—there may be more than one.**

i. If data can be inserted in the table without collisions, searching a hash table is a constant time operation.

ii. The worst-case execution time of searching a hash table that uses chaining to resolve collisions is O(n log n).

iii. Double hashing describes a style of programming in which, if you fail to successfully design a good hash function for your hash table after two tries, you give up.

iv. Chaining is a technique for resolving collisions in which each hash table entry is the start of a linked list of all the values mapping to the same index.

EXTRA CREDIT

a. (5 points) Answer **only one** of the following questions (i) or (ii):

i. Explain why removing the highest value from a heap is a constant time operation (in other words, the order of magnitude of its execution time is O(1)).

ii. Explain why the worst-case search time for a binary search tree is O(n) and under what conditions a search would require searching every node in the tree.

EXTRA CREDIT (continued)

b. (5 points) Is it possible to write an array-based queue implementation that only stores one index variable, back, and the number of values currently in the queue? If so, briefly explain how you would implement the enqueue and dequeue operations. If not, explain why not.