EECE.3220: Data Structures

Spring 2019

Exam 3 Solution

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;
   Next loop alternates taking values from Quahog & Quincy,
      printing front value of each as it goes through loop and
      stopping once one is emptied
   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;
}
```

After loop, queue values are:
Quahog: 1.1, 2.2, 3.3, 4.4, 5.5
Quincy: 9.9, 8.8, 7.7, 6.6, 5.5

OUTPUT:

```
1.1 9.9
2.2 9.9
2.2 8.8
3.3 8.8
3.3 7.7
4.4 7.7
4.4 6.6
5.5 6.6
5.5 5.5
Quahog empty
```

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() {
   if (back >= front)
      return back - front;
   else
      return (back + cap) - front;
}
```

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) {
   int i;

   if (!empty()) {
      for (i = front; i != back; i = (i + 1) % cap)
         out << list[i] << ' ';
      out << '\n';
   }
   else {
      out << "Queue empty\n";
   }
}
```

Note: the original solution said nothing about handling an empty queue, so I didn't penalize people who left that code out. I really needed to specify the problem better—it's possible for the display function to handle an empty queue, or it's possible for the caller to check the empty condition before calling, like this (assuming you've declared `Queue Q`):

```
if (!Q.empty())
   Q.display(cout);
```

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. **This question may have more than one correct answer—choose all answers you believe are correct.**

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.

You can make a pretty good argument that (v) is a correct answer as well, but choices (i)-(iv) were all cases we discussed in class. Frankly, as long as you're consistent across all of your Queue functions, any pair of initial values for array indexes will work.

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
   if (empty())
      return false;

   // Otherwise, traverse queue and compare consecutive values
   // If all pairs of values are in order, queue is in order
   p = front->next;
   val = front->data;

   while (p != NULL) {

      // If out-of-order pair found, queue isn't in order
      if (val < p->data)
         return false;

      // Otherwise, update variables to move to next node
      val = p->data;
      p = p->next;
   }

   // If you get through whole queue, must be ordered
   return true;

}
```

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++) {   At the end of this loop:
      L1.insert(v);              L1: -0.8, -0.2, 0.1, 0.4, 1.6
      Q1.enqueue(v);             Q1: 0.1, -0.2, 0.4, -0.8, 1.6
      v = v * -2;
   }

   while (!Q1.empty()) {            This loop prints Q1,
      cout << Q1.getFront() << " ";   doubles front value and
      L1.insert(Q1.getFront() * 2);   adds it to L1, then
      Q1.dequeue();                   removes value from Q1
   }
   After loop, L1: -1.6 -0.8 -0.4 -0.2 0.1 0.2 0.4 0.8 1.6 3.2

   cout << "\n";
   L1.display(cout);     // Prints list contents from first to
                         //   last node

   return 0;
}
```

OUTPUT:

```
0.1 -0.2 0.4 -0.8 1.6
-1.6 -0.8 -0.4 -0.2 0.1 0.2 0.4 0.8 1.6 3.2
```

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.

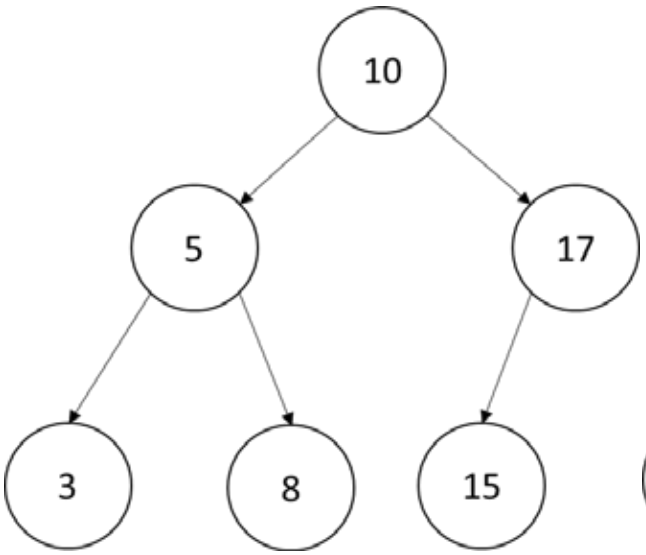
```
bool LList::inList(double val) {
  Node *p;          // Pointer to current node

  // Start at first node
  p = first;

  // Traverse list until either:
  //  (1) You reach a NULL pointer, or
  //  (2) You reach a node with a value > val, which means
  //       val can't be in list (assuming list ordered lo->hi)
  while (p != NULL && val >= p->val) {

    // If you find a match, return true
    if (val == p->val)
      return true;

    // Otherwise (implied else), move to next node
    p = p->next;
  }

  // If you reach this line, must have exited loop without match
  return false;
}
```

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

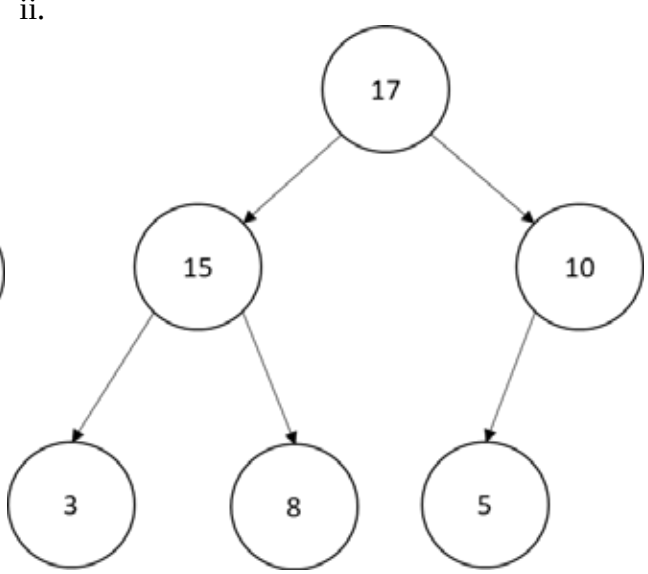
a. Which of the trees below are binary search trees? **Circle ALL correct answers—there may be more than one.**

SOLUTION: (i) and (iv) are BSTs. I tricked myself with (iii)—I intended to make it a BST, but most of the values in the right subtree of 10 are less than 10—so I didn't penalize anyone who chose it. (ii) is a heap, which relates directly to part (b) …

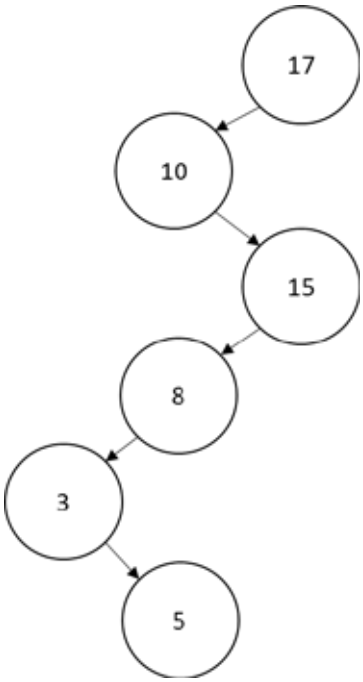
i.



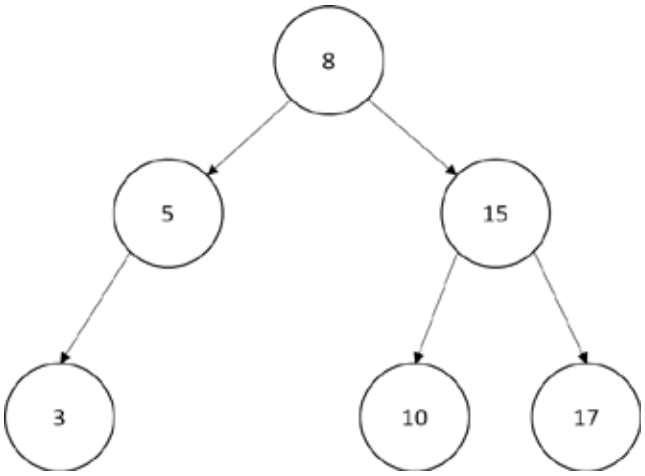
ii.



iii.



iv.

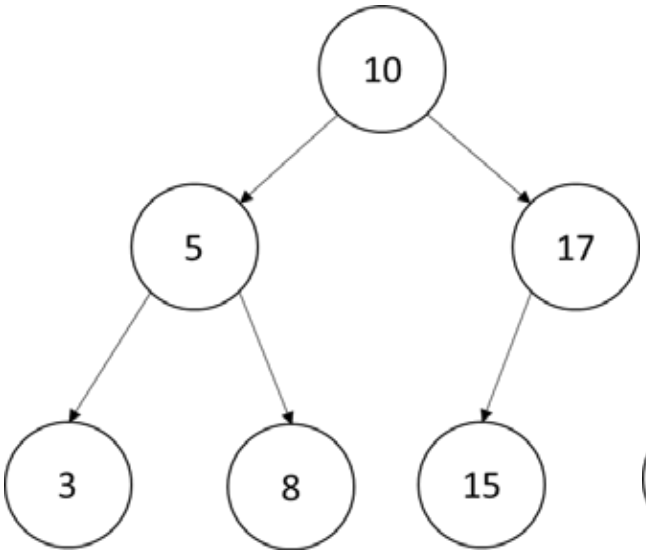


4 (continued)

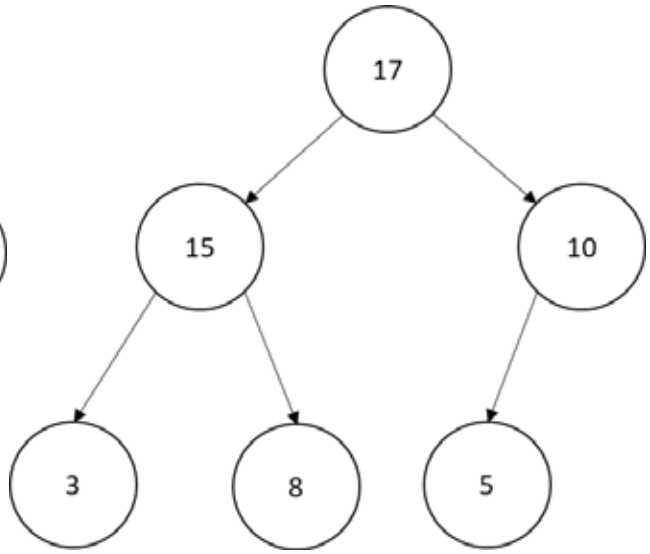
b. Which of the trees below are heaps? **Circle ALL correct answers—there may be more than one.**

SOLUTION: As noted in part (a), (ii) is a heap. The other three choices aren't—the greatest value in (i) isn't at the root of the tree, and (iii) and (iv) aren't complete trees.

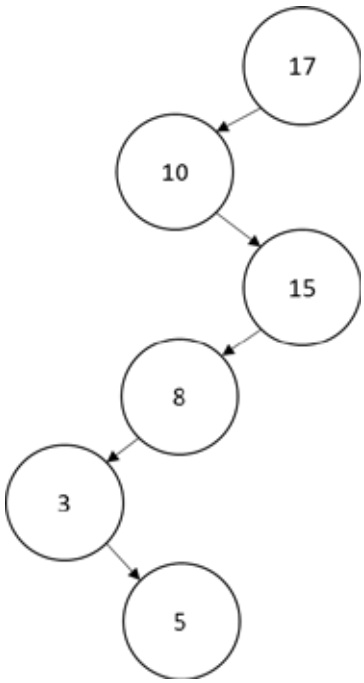
i.



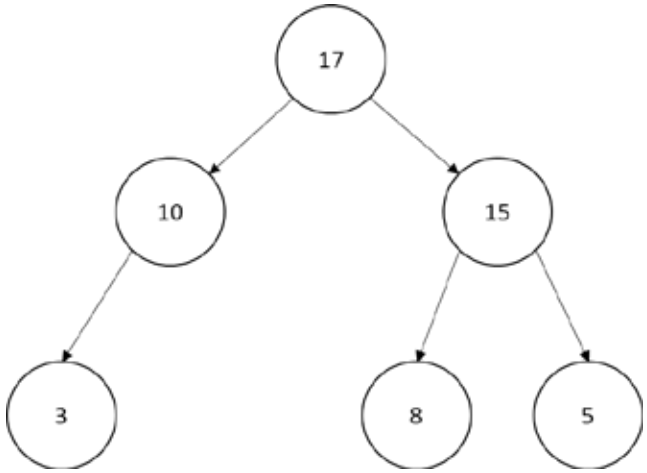
ii.



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.***

Note: As mentioned in class, all priority queue functions (at least the ones we briefly discussed, which include these four functions) can be implemented using heap functions.

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). ***(It's O(n), since the worst case is effectively a linked list.)***

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. ***(While this answer seems pretty reasonable, it's not actually correct.)***

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):

A quick note to future students studying this old exam solution: if there's a question like this on your exam, answering both parts of the question will not get you extra points. Please just follow the directions.

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)).

Solution: *The highest value in a heap is always stored at the root of the tree, which means it can be accessed in a constant amount of time.*

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.

Solution: *In the worst case for a BST, every node has at most one child and they're all chained together (like choice (iii) in Questions 4a and 4b). The worst-case BST organization is therefore effectively a linked list, which has a worst-case search time of O(n) (search every node in the list).*

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.

Solution: *The proper answer is "no"—you also need to know the queue capacity to make such a solution work. However, since it wasn't my intent to be that tricky—I forgot about the capacity when I was writing the problem, since I assumed you knew the capacity is part of any array-based queue implementation—I allowed anything like the following response:*

This implementation is absolutely possible, since you can use a combination of the back index and the number of values in the queue to find the front index—if N is the number of values in the queue, front is N spots "behind" back (using modulo arithmetic).

Assume that, initially, back = 0 and N = 0, and that the array inside the queue is called list. The enqueue and dequeue functions would therefore behave as described in the pseudocode below:

```
void enqueue(QueueElement val) {
   if (N == queue capacity)
      Queue is full, add nothing and indicate error

   else {
      list[back] = val;
      back = (back + 1) % (queue capacity);
      N++;
   }
}

void dequeue() {
   if (N == 0)
      Queue is empty, remove nothing and indicate error

   else
      N--;      // You don't actually need to find the front to
                //   dequeue, since dequeueing simply removes—
                //   not returns—the front item
}
```