# EECE.3220: Data Structures

Fall 2019

Exam 2
November 7, 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 before the start of the exam to avoid distracting other students.

The exam contains 4 sections for a total of 100 points, as well as a 10-point extra credit question. 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:

- Several questions (1b, 1c, 3a, 3b) require you to complete a short function. In some cases, we have provided comments to describe what your function should do and written some of the code.
    - You can complete the function using only the variables that have been declared, but you may declare and use other variables if you want.
- The exam contains several multiple-choice questions. Questions 2b, 4a, 4c, and 4d each have at least one correct answer, but may have more than one correct answer. Question 4b has exactly one correct answer.
    - Again, make sure you read these questions carefully. If just one part of a statement is false, that statement isn't a correct answer to the question being asked!

You will have 2 hours to complete this exam.

| | |
|---|---|
| S1: Stacks | / 36 |
| S2: Queues | / 16 |
| S3: Operator overloading | / 32 |
| S4: Multiple choice: templates, linked lists | / 16 |
| **TOTAL SCORE** | / 100 |
| **S5: EXTRA CREDIT** | / 10 |

1. (36 points) ***Stacks***

a. (16 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. The next page is left blank to allow you extra space to solve this problem.

This code uses the linked `Stack` shown on the extra sheet, but it works with any `Stack` class template.

```
int main() {
   Stack<int> Sasquatch;
   Stack<int> Seymour;
   int list[10] = { 3220, 450, 11, 6, 2019, -2, 86, 23, 30, -15};
   int i, num;

   i = 0;
   for (num = 0; num < 10; num++) {
      if (list[i] > 25)
         Sasquatch.push(list[i]);
      else
         Seymour.push(list[i]);
      i = (i + 3) % 10;
   }

   cout << "Sasquatch:\n" << Sasquatch;
   cout << "\nSeymour:\n" << Seymour;

   for (i = 0; i < 6; i++) {
      if (i % 2 == 1) {
         Sasquatch.push(Seymour.getTop());
         Seymour.pop();
      }
      else
         Sasquatch.pop();
   }

   cout << "\nSasquatch:\n" << Sasquatch;
   cout << "\nSeymour:\n" << Seymour;

   return 0;
}
```

***Extra space to solve Question 1a***

1 (continued)

Write a definition for each of the short functions defined below. Assume you are using the linked `Stack` and `Node` classes shown on the extra handout provided with the exam.

b. (10 points) `size()`: Returns the number of nodes in the stack

```
template <class T>
unsigned Stack<T>::size() {
   unsigned n;         // # of values
   Node<T> *ptr;       // Node pointer









}
```

c. (10 points) `getBottom()`: Return value in bottom (last) node of stack (if that node exists)

```
template <class T>
T Stack<T>::getBottom() {
   Node<T> *ptr;       // Node pointer

   if (_____________________________________________________________) {
      // Function should return garbage value here. You do not have to
      //   write that code--just identify the condition that causes
      //   this section to execute and write it on the blank line above
   }
   else {        // Find bottom of stack and return value in that node









   }
}
```

2. (16 points) ***Queues***

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 function uses the array-based `Queue` defined on the extra sheet, although it would work with any `Queue` class template. Assume the << operator prints all items in the queue from front to back on a single line, with one space between each item and a newline at the end.

```
int main() {
   Queue <char>Quirrell;
   string s1 = "stressed";
   string s2 = "insects";
   int i, j;

   for (i = s1.size() - 1; i >= 0; i--)
      Quirrell.enqueue(s1.at(i));            // Remember, s1.at(i) is
                                             //  similar to s1[i]
   cout << "Q = " << Quirrell;

   for (i = 0; i < s2.size(); i += 2) {
      Quirrell.enqueue(Quirrell.getFront());
      Quirrell.enqueue(s2.at(i));
      for (j = 0; j < 2; j++)
           Quirrell.dequeue();
   }

   cout << "Now, Q = " << Quirrell;

   return 0;
}
```

2 (continued)

b. (4 points) Which of the following choices describe a valid way to ensure that you can tell the difference between the empty and full states of an array-based queue? **This question has at least one correct answer but may have more than one correct answer! Circle ALL choices that correctly answer the question.**

i. Leave one spot in the array empty, so the queue is empty when the front and back indexes are equal, and full when the back index is one spot behind the front index.

ii. Add a Boolean variable, `isEmpty`, to the `Queue` class. `isEmpty` is initialized to `true`, set to `false` when the first object is added to the queue, and set to `true` when the last object is removed from the queue. The front and back indexes are equal if the queue is empty or full, but `isEmpty` allows you to tell the difference between these two states.

iii. Add an unsigned variable, `nItems`, to the `Queue` class. `nItems` is initialized to 0, we subtract 1 from `nItems` every time an object is added to the queue, and we add 1 to `nItems` every time an object is removed from the queue. The queue is empty when `nItems` equals 0, and full when `nItems` equals the capacity of the queue.

iv. All these choices are incorrect—there's no good way to tell the difference between the empty and full states.

3. (32 points) ***Operator overloading***

For each part of this section, complete the overloaded operator as described.

a. (16 points) `Stack<T> &Stack<T>::operator +=(const Stack<T> &rhs)`

Augmented assignment to "add" stack on right to calling object. Data from calling object is at top of updated stack. So, if `Stack<int> S1` holds `{5, 3}` (with 5 at top) and `Stack<int> S2` holds `{2, 4}` (with 2 at top), then after executing `S1 += S2`, `S1` will hold `{5, 3, 2, 4}`.

This function uses the linked `Stack` and `Node` classes shown on the extra sheet.

```
template <class T>
Stack<T> &Stack<T>::operator +=(const Stack<T> &rhs) {
   Stack<T> temp;           // Temporary stack
   Node<T> *p;              // Node pointer

   // Condition true if no self-assignment (calling obj. isn't also rhs)

   if (_____________________________________________________) {

      // Store data from calling object in temp



      while (_____________________________________________________) {









      }

      // Copy data from RHS to calling object—shouldn't need loop!



      // Copy data from temp back to calling object

      while (_____________________________________________________) {









      }
   }
   return *this;      // Return calling object
}
```

3 (continued)

b. (16 points) `bool Queue<T>::operator ==(const Queue<T> &rhs)`

Comparison operator that returns true if data in two `Queue` objects is the same and false otherwise. (The `Queue` capacities do not have to match, only the data contained in both queues.)

This function uses the array-based `Queue` class shown on the extra sheet.

```
template <class T>
bool Queue<T>::operator ==(const Queue<T> &rhs) {
   int left, right;   // Indexes into Queues on left/right of operator

   // Initialize both indexes









   // Loop as long as you haven't reached the back of either queue

   while (_____________________________________________________________) {

      // If you find a mismatch, queues aren't equal

      if (_____________________________________________________________)
            return false;

      // Otherwise, increment both indexes appropriately









   }

   // If you reach the back of both queues, must be equal

   if (_____________________________________________________________)
      return true;

   // Otherwise, one queue is shorter than other and they don't match
   else
      return false;
}
```

4. (16 points, 4 points each) ***Multiple choice: templates, linked lists***

a. Which of the following statements about programming with templates is true? **This question has at least one correct answer but may have more than one correct answer! Circle ALL choices that correctly answer the question.**

i. To call a function with template arguments, the programmer must explicitly write out the type of each argument—for example, `f1(<int>x, <int>y);`

ii. To create an object of a templated class, the programmer must explicitly declare the type of data being stored in that object—for example, `Stack <double> S1;`

iii. Member functions of a class template should be defined in the same file as the class definition.

iv. Class templates can have member functions that are not template functions.

*Parts b and c of this section refer to the `LList` class defined on the extra sheet.*

b. The linked list `insert()` function discussed in class contains the following loop for finding the predecessor and successor of the node to be inserted into the list. The lines are numbered for the sake of this question:

```
while (curr != NULL &&                  (1)
      curr->val < newNode->val)         (2)
{
   prev = curr;                         (3)
   curr = curr->next;                   (4)
}
```

Which of the following changes to this loop would change the list order so its contents are sorted from highest to lowest value? **This question has exactly one correct answer.**

i. Change line (1) to: `while (curr <= NULL &&`

ii. Change line (2) to: `newNode->val < curr->val)`

iii. Change line (3) to: `curr = prev;`

iv. Change line (4) to: `curr = prev->next;`

4 (continued)

*Parts b and c of this section refer to the* `LList` *class defined on the extra sheet.*

c. Which of the following choices represent "special cases" that must be accounted for when writing linked list functions? **This question has at least one correct answer but may have more than one correct answer! Circle ALL choices that correctly answer the question.**

i. Adding a node after the current last node in the list.

ii. Adding a node before the current first node in the list.

iii. Removing the current last node in the list.

iv. Removing the current first node in the list.

d. Which of the following statements accurately reflect your opinion(s)? Circle all that apply (but please don't waste too much time on this "question")!

i. "This course is moving too quickly."

ii. "This course is moving too slowly."

iii. "I've attended very few lectures, so I don't really know what the pace of the course is."

iv. "I hope the next exam is as easy as this question."

v. "Is a course with no programming assignments really a programming course?"

5. (10 points) ***EXTRA CREDIT***

Complete the overloaded operator described below:

```
bool AStack<T>::operator !=(const Stack<T> &rhs);
```

Compares an array-based stack (calling object) to a linked stack (argument `rhs`), returning true if the stacks do not contain the same data and false otherwise. (Yes, it's possible to compare objects of different types.)

This question refers to both stack classes (`AStack` and `Stack`) defined on the extra sheet.

```
template <class T>
bool AStack<T>::operator !=(const Stack<T> &rhs) {
   Stack <T> temp;          // Temporary stack--may be helpful,
                            //   since you can't modify rhs
   int i;                   // Index into array-based stack
                            // DECLARE OTHER VARIABLES IF YOU WANT


   // Initialize local variables as needed









   // Loop to compare stack elements

   while (_____________________________________________________________) {









   }

   // If we reach the end of the loop, must determine if stacks are
   //   same length or different
   // Return true in one case, false in the other




}
```