EECE.3220: Data Structures

Fall 2019

Exam 2 Solution

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;
}
```

Loop visits all list[] elements, with ½ the values going to each stack. i always goes up by 3, using modulo arithmetic to stay < 10. So, after this loop:
Sasquatch = { 2019, 450, 30, 86, 3220 }
Seymour = { 23, -2, 11, -15, 6 }

Loop alternates operations: If i is even, it pops the top value off Sasquatch If i is odd, it pushes the top value from Seymour to Sasquatch, then pops that value off Seymour
So, after the 1st iteration (which removes 2019 from Sasquatch), it alternates between moving a value from Seymour to Sasquatch, then popping that value off Sasquatch anyway. The last value to move (11) is not removed.

Final program output is on the next page.

Output for Question 1a

```
Sasquatch:
2019
450
30
86
3220

Seymour:
23
-2
11
-15
6

Sasquatch:
11
450
30
86
3220

Seymour:
-15
6
```

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.

Other solutions may work for each of these problems, but my solutions are shown below. Both functions are essentially linked list (stack) traversals. The `size()` function counts every node it visits, while the `getBottom()` function simply navigates to the last node (as long as the stack isn't empty).

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

   n = 0;          // Start count at 0. Ensures right value returned if
                   //   stack empty, too

   // Traverse stack and count every node
   // Wrote traversal as a for loop, just to show something different
   //   than what we covered in class
   for (ptr = top; ptr != NULL; ptr = ptr->getNext())
      n++;

   return n;
}
```

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 (empty()) {
      // 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
      ptr = top;
      while (ptr->next != NULL)          // Look ahead one node--ptr
         ptr = ptr->getNext();           //  should point to last node
                                         //  when loop is done
      return ptr->getVal();
   }
}
```

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;
```

Loop adds characters from s1 starting with last and working toward first, so Quirrell = { 'd', 'e', 's', 's', 'e', 'r', 't', 's'}

```
   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();
   }
```

Loop does following:
Add copy of front to back
Enqueue every other char from s2 (i, s, c, s)
Dequeue the two front chars

The loop above has 4 iterations, so the queue changes as shown below:

{ 'd', 'e', 's', 's', 'e', 'r', 't', 's'} → (Initial state)
{ 's', 's', 'e', 'r', 't', 's', 'd', 'i'} → (After 1st iter)
{ 'e', 'r', 't', 's', 'd', 'i', 's', 's'} → (After 2nd iter)
{ 't', 's', 'd', 'i', 's', 's', 'e', 'c'} → (After 3rd iter)
{ 'd', 'i', 's', 's', 'e', 'c', 't', 's'} (After 4th iter)

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

   return 0;
}
```

OUTPUT:

```
Q = d e s s e r t s
Now, Q = d i s s e c t s
```

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.

The parts in red make the choice above false, for two reasons:

- *Subtracting 1 from `nItems` every time an item is added would make that variable equal the negative of the queue capacity when full …*
- *… if `nItems` was a signed integer. As an unsigned value, what typically happens when executing 0 – 1 is the result "wraps around" to the maximum possible unsigned value ($2^{32} - 1$)*

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 (*this != rhs) {

      // Store data from calling object in temp
      while (!empty()) {
         temp.push(top->getVal());     // Can't just copy—need to put
         pop();                        //  data back in original order
      }

      // Copy data from RHS to calling object—shouldn't need loop!
      *this = rhs;                  // Can just copy here—want data from
                                    //   rhs in same order in calling obj.

      // Copy data from temp back to calling object
      while (!temp.empty()) {
         push(temp.getTop());
         temp.pop();
      }
   }
   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
   left = front;
   right = rhs.front;

   // Loop as long as you haven't reached the back of either queue
   while (left != back && right != rhs.back) {

      // If you find a mismatch, queues aren't equal
      if (list[left] != rhs.list[right])
            return false;

      // Otherwise, increment both indexes appropriately
      left = (left + 1) % cap;
      right = (right + 1) % rhs.cap;
   }

   // If you reach the back of both queues, must be equal
   if (left == back && right == rhs.back)
      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?"

All of the above are "correct" … but it's probably (v).

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
   temp = rhs;          // Copy rhs to temp--since we can't access node
                        // contents, will pop data off temp as we compare

   i = tos;      // Start i at top and work our way back

   // Loop to compare stack elements
   // Repeat as long as we haven't hit end of either stack
   while (i != -1 && !temp.empty()) {

      // Mismatch found--stacks don't match
      if (list[i] != temp.getTop())
         return true;

      // Get to next value
      i--;
      temp.pop();
   }

   // 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
   // If we hit the end of both stacks at the same time, they match
   if (i == -1 && temp.empty())
      return false;

   // Otherwise, one holds less data than the other--no match
   else
      return true;
}
```