CISC 3130 Midterm Review Guide

Readings

You are responsible for the following material in *Open Data Structures*:

- Chapter 1, excluding 1.4, and with only a superficial coverage of 1.3.
- Chapter 2, excluding 2.6.
- Chapter 3, excluding 3.3.5.

The general discussion of STL containers at http://www.cplusplus.com/reference/stl/, as well as the general behaviors of the stack, queue, deque, forward_list, and list STL containers.

Those parts of *C++ Annotations* we've drawn from: 12.1, 12.2, 12.4, 18.2.

Study Tactics

Work as many exercises from the book as you can, perhaps excluding 2.9–2.12 and 3.16–3.19.
Review all the questions I've asked you in this class (see Application Activities appended below).
Review all code from exercises and project assignments.
Re-read readings now that you've worked with ideas in lecture/lab.

Main Topics

C++ Concepts

I expect you to be able to read and write C++ code, including code that uses templates and public inheritance.

I expect you to be able to read and write code that uses the STL, if I provide an "API" listing of containers and methods.

Data Structures Concepts

You should be able to discuss the List, Stack, Queue, and Deque interfaces from the book, as well as their array-based and linked-list implementations. You should be able to analyze/evaluate new approaches to implementing these data structures.

You should be able to use array-based and linked-list techniques to implement operations on the data structures we've studied.

You should be able to analyze simple code to determine whether it's *O(1)*, *O(n)*, etc. You should know how the words "constant," "linear," etc apply to big-O classes. You should be able to compare algorithms from different big-O classes.

Exam Structure, Roughly

50%: short-answer/multiple-choice
30%: writing code
20%: analyzing code

1. What two operations are supported by every data structure discussed in Chapter 1?

a) get and set
b) add and remove
c) read and write
d) initialize and delete
e) print and scan

2. If all these data structures support the same operations, then why do we have so many different ones?

a) Not all data structures have correct behavior.
b) Different applications have different efficiency requirements for different operations.
c) Computer scientists have nothing better to do.
d) Different operating systems organize disk space differently.
e) All of the above

3. What is the best data structure to use to solve this problem? It should be as fast as possible and use as little storage as possible.

Read the input one line at a time and then write the lines out in reverse order, so that the last input line is printed first, then the second last input line, and so on.

a) `Stack`
b) `Queue`
c) `USet`
d) `SSet`

4. What is the best data structure to use to solve this problem? It should be as fast as possible and use as little storage as possible.

Read the input one line at a time. At any point after reading the first 42 lines, if some line is blank (i.e., a string of length 0), then output the line that occured 42 lines prior to that one. This program should be implemented so that it never stores more than 43 lines of the input at any given time.

a) `Stack`
b) `Queue`
c) `USet`
d) `SSet`

5. What is the best data structure to use to solve this problem? It should be as fast as possible and use as little storage as possible.

Read the input one line at a time and write each line to the output if it is not a duplicate of some previous input line. Note that a file with a lot of duplicate lines should not require more memory than what is required for the number of unique lines.

a) `Stack`
b) `Queue`
c) `USet`
d) `SSet`

6. A `List` is "fancier" than a `Queue`—it has more operations. Why *doesn't* `Queue` have an operation like

get(i) — return the value x_i

a) Not every implementation of `Queue` allows you to access the "ith" element.
b) It's impossible to write an implementation of `Queue` that would allow us to access the ith element
c) The designers of `Queue` didn't realize we might need the ith element.
d) A `Queue` only cares about the position where elements are added and the position where elements are removed.

7. Write pseudocode that solves the following problem:

Suppose you have a `Stack`, `s`, that supports only the `push(x)` and `pop()` operations (and `empty()`). Show how, using only a FIFO `Queue`, `q`, you can reverse the order of all elements in `s`. That is, if `s` starts with 1 at the "bottom" followed by 2, 3, up to 10 on top of the `Stack`, then after this code runs, `s` should have 1 on top and 10 at the bottom.

8. Fundamental mathematical question: if $\log_2 k = 8$, what is the value of k?

a) 3
b) 16
c) 37
d) 256
e) 1,030,492

9. Suppose this class has 40 students. How many ways could I have picked the 7 people for the first team?

a) (40 * 39 * 38 * 37 * 36* 35 * 34) / (7 * 6 * 5 * 4 * 3 * 2) = 18,643,560
b) 40 * 7 = 280
c) 40^7 = 163,840,000,000
d) 40 / 7 = about 6

10. What is the purpose of that weird "Big-Oh" notation?

a) To make sure computer science students don't have a semester without suffering.
b) To allow us to compare algorithms without having to know their implementation details.
c) To allow us to ignore how different algorithms behave on small input and focus on their performance on relatively large input.
d) To allow us to ignore inconsequential differences between algorithms.
e) All of the above.

11. Based on Figure 1.5 of the book (page 16 in the PDF), would you rather be running a O(*n*) algorithm or a O(*n* log *n*) algorithm?

a) O(n)
b) O(*n* log *n*)
c) There's really no difference.

Bonus 1: The book says that Big-Oh notation is also called "asymptotic" notation. Why is it called that?

Bonus 2: Why don't the "log" values in Big-Oh notation have an explicit base? (It's not because the book has declared "log" to mean "log base 2.")

12. The book says "Arrays are not very dynamic." What does the book mean by that?

a) It is generally inefficient to change the value of an array element.
b) It is generally inefficient to insert a value into an array.
c) Arrays are generally not exciting.
d) The size of an array does not change.
e) The memory location of an array does not change.

13. Where is it most efficient to insert a new value into an array containing `n` elements? (Assume the capacity of the array is much larger than `n`.)

a) At index `0`.
b) In the vicinity of index `n/2`.
c) At index `n`.
d) None of the above.
e) Inserting an element is equally efficient at all locations.

14. When we `resize()` an array-based data structure, what is the most expensive part of the process?

a) Calculating the value of `2*n`.
b) Allocating the new chunk of memory
c) Copying values into the new chunk of memory
d) De-allocating the old chunk of memory.

15. The book says that the `ArrayStack` operations “will run in *O(1)* amortized time.” If we loosely understand “amortized” to mean “spread out across” (like loan payments are spread out across many months), what is being “amortized” here?

a) The high cost of resizing the array is spread out across all the add/remove operations.
b) The high cost of popping elements is spread out across all the stack operations
c) The high cost of removing an element is spread out across all the add operations (you can’t remove something until after it’s been added)
d) The high cost of adding elements near the beginning of the array is spread out across all the add operations.

16. Consider the infinite array “implementation” for `ArrayQueue`. Over time, what will happen to the indices of the elements of the array where the queue is stored?

a) They will tend to stay close to 0.
b) It’s impossible to predict; it depends on the pattern of add and remove operations.
c) They will tend to increase.
d) They will tend to decrease.

17. Consider the (imaginary) case of an infinite array that’s storing a queue. The queue contains 5 elements, `j`, `k`, `l`, `m`, and `n`, stored at indexes from 27 to 31, like this

…	j	k	l	m	n	…
	27	28	29	30	31	

Now suppose that we’re using a 10-element circular array to store this queue instead—and the queue hasn’t yet had to contain more than 10 elements. How would this queue state be represented by that array?

a)

j	k	l	m	n					
0	1	2	3	4	5	6	7	8	9

b)

j	k	l						m	n
0	1	2	3	4	5	6	7	8	9

c)

			j	k	l	m	n		
0	1	2	3	4	5	6	7	8	9

d)

m	n						j	k	l
0	1	2	3	4	5	6	7	8	9

e)

					j	k	l	m	n
0	1	2	3	4	5	6	7	8	9

18. What's a *deque*?

a) It's a European term for a "deck," like a deck of cards.
b) It's a limited form of queue that only allows `remove()` operations (which are called "de-queue" for queues)
c) It stands for **d**ouble-**e**nded **que**ue.
d) Pronounced "duck-whee," it's a device like a roller-coaster that is used to entertain the ducks and geese in Central Park during the winter, thereby greatly reducing depression among quasidomesticated poultry.

19. What's the big efficiency challenge with an array-based deque?

a) Having "two ends" makes getting and setting element values much less efficient.
b) Having "two ends" makes resizing more difficult and less efficient.
c) Having "two ends" means there are four specific operations that need to be efficient.
d) Having "two ends" is a nightmare—All of the above!

20. So the `ArrayDeque` combines the circular array idea of `ArrayQueue` with the element-shifting idea of `ArrayStack`–a bit more complicated than the previous two structures. What is the impact on the O()-efficiency of the four `ArrayDeque` operations?

a) All four operations are less efficient than any of the stack or queue operations.
b) Some of the operations are less efficient than stack/queue, but some are just as efficient.
c) The deque operations are equally efficient as the stack and queue operations.
d) The relative efficiency depends on the exact state of the deque—sometimes operations are more efficient, sometimes less.

21. Which data structure is being used any time a C/C++ program is executing?

a) Stack
b) Queue
c) Deque
d) Set

22. Most modern programming languages use a variety of brackets

```
( )
{ }
[ ]
< >
```

to mark beginnings and endings. In general, there are a few common expectations about these should "behave":

i. Every closing symbol must be "matched" by a preceding opening symbol, and every opening symbol must be matched by a following closing symbol So `><` isn't acceptable, unless it's part of `<><>`.
ii. Usually, the brackets contain other text. So `<int>` is much more common than `<>`.
iii. Brackets can contain other *pairs* of brackets, but not single brackets. That is, brackets can be nested like `{()}` but can't be interleaved like `{(})`.

Write pseudocode that uses this chapter's data structures to determine whether the source code for a C++ program contains syntactically correct brackets. Use the function `char readChar()` which magically returns the next character from input, and use the function `bool isBracket(char)` which tells you whether or not a character is one of the 8 bracket characters (you do not have to write these functions! Just call them as needed). Assume that `readChar()` returns `NULL` when the end of input is reached. You may also use `bool isOpen(char)` and `bool isClosed(char)` to identify what kind of bracket you have. Note that you really don't care about the non-bracket characters, and you're not worried about the syntax of the code other than the brackets. Your code should print "yes" if the brackets are correct and "no" if they are not. Your code should produce no other output.

23. The book's `SLList` data structure includes three variables: `Node* head`, `Node* tail`, `int n`. If we implement `Stack` using `SLList`, what values correspond to an empty `Stack`?

a) `n == 0`
b) `head == tail == NULL`
c) Either a or b
d) Both a and b

24. Here's the book's implementation of `Stack.push(x)` (for a singly-linked list):

```
T push(T x) {
    Node *u = new Node(x);
    u->next = head;
    head = u;
    if (n == 0)
      tail = u;
    n++;
    return x;
  }
```

What would be the effect of exchanging the two highlighted lines?

a) The code would no longer compile.
b) The program would crash at some point during the execution of `push()`.
c) The program would crash after some subsequent call to `pop()`.
d) This would introduce a memory leak.
e) Both c and d.

25. Again, here's the book's implementation of `Stack.push(x)`:

```
T push(T x) {
    Node *u = new Node(x);
    u->next = head;
    head = u;
    if (n == 0)
      tail = u;
    n++;
    return x;
  }
```

What would be the effect of deleting the two highlighted lines?

a) The code would no longer compile.
b) The program would crash at some point during the execution of `push()`.
c) The program would crash after some subsequent call to `pop()`.
d) This would introduce a memory leak.
e) This would have no effect on `Stack` operations.

26. Here's the book's implementation of `Queue.add()`:

```
bool add(T x) {
    Node *u = new Node(x);
    if (n == 0) {
      head = u;
    } else {
      tail->next = u;
    }
    tail = u;
    n++;
    return true;
  }
```

What would be the effect of deleting the highlighted code?

a) The code would no longer compile.
b) The program would crash at some point during the execution of `add()`.
c) The program would crash during some subsequent call to `remove()`.
d) This would introduce a memory leak.
e) This would have no effect on `Queue` operations.

27. Which of these represent an “empty” doubly-linked list?

a) A single node, `dummy`, with `prev` and `next` set to `NULL`.
b) A single node, `dummy`, with `prev` and `next` pointing to `dummy`.
c) A single node, `dummy`, with the list’s `head` and `tail` pointers both pointing to `dummy`.
d) An empty node.
e) A node, `dummy`, with `prev` and `next` both pointing to another node containing the value `0`.

28. Here’s the book’s `addBefore()` implementation:

```
1       Node* addBefore(Node *target, T value) {
2           Node *toInsert = new Node;
3           toInsert->x = value;
4           toInsert->prev = target->prev;
5           toInsert->next = target;
6           toInsert->next->prev = toInsert;
7           toInsert->prev->next = toInsert;
8           n++;
9           return toInsert;
10      }
```

Which of these pairs of lines can be exchanged without affecting the list’s behavior?

a) 4 and 5
b) 5 and 6
c) 6 and 7
d) Both a and b
e) Both a and c

29. The book has a nice picture (Figure 3.3) of how to add a node to a doubly-linked list, but it doesn't have a picture for the `remove(Node *)` operation. Draw that picture in a style similar to Figure 3.3. Instead of light gray lines and black lines, use dotted lines to represent things that exist only before the operation is carried out and solid lines to represent things that exist after the operation is completed.

30. Design and implement an `SLList` method, `secondLast()`, that returns the second-to-last element of an `SLList`. (That is, if the list contains 10 elements, this method should return the value of the 9th element.) Do this without using the member variable, `n`, that keeps track of the size of the list. How should this method behave if the list contains 0 or 1 element?

31. Implement `SLList.set(i,x)`.

32. Write a method, `truncate(i)`, that truncates a `DLList` at position `i`. After executing this method, the size of the list will be `i` and it will contain only the elements at indices `0,...,  i-1`. The return value is another DLList that contains the elements at indices `i,...,n-1`. This method should run in *O(min(i, n-i)* time.