Study Guide Week II:

Chapter 5 – Abstraction and Abstract Data Types

**Abstraction and Information Hiding – See Study Guide I.**

Quick Review: These are the processes of identifying and hiding unimportant aspects of an object or model. It then leaves only essential features and elements visible and available.

Abstraction can be equated to an Atlas. The deeper you go in, the more detailed things become.

Another term to think of is *encapsulation.*

**Encapsulation** – packaging, placing items into a unit or capsule. The purpose of this process is to isolate the extent of the task being performed and keep the implementation of the task specific. This allows us to keep our programs and software in small, digestible, easily usable chunks. Different programming languages hve different techniques for encapsulation.

Some languages (but not C) use something called an *interface.*

**Interface –** a collection of functions that serve a common purpose. These are commonly used as the prototypes in a class or of a function, sometimes called a signature. For example:

int addDivideSum(int a, int b, int c);

The function would then be filled in in the source of the code like such:

Int addDivideSum(int a, int b, int c)

{

int result;

c = a + b;

result = c / b;

return result;

}

Abstract Data Types:

Often termed collections or containers. There are a large number of collection/container types.

**Stack** – think of a stack of plates. Last In, First Out. We can only take from the top.

**Queue** – Think of a line, first in, first out.

**Deque** (pronounced deck) – A two sided line – whatever is in the front or in the back can be removed.

We use these metaphors, and then the behavior of the ADT that determines the type of the collection, not necessarily for the operations. And how it is utilized is called the implementation.

Other Classic Abstract Data Types:

**Bag** – an unordered collection. Think of a bag of marbles.

**Set** – An extension of the bag, which adds restrictions to the data set. Can help prevent repeats or restrict what sort of data can be accessed.

Linear Collections (See above)

**Priority Queue** – values in order of importance. Think of a to-do list.

**Map/Dictionary** – a pair of elements called a key and a value are used. Think of dictionaries, like the name.

ADT Implementations –

The most basic form is called an array. Arrays are a fixed block of memory, with adjacent cells holding memory in the collection.

They have a limited size, so they have a close relative called the dynamic array which increases the size and capacity of a container. Keep in mind, size and capacity are two distinct features. Size is the actual amount of data elements in a container, capacity is the number of elements can HOLD.

Another alternative to an array is a **linked list**. This refers to an array-like structure or list where each element points to the next in the sequence, and they do not necessarily need to be stored in adjacent or subsequent memory locations.

Both arrays and linked lists suffer from the linear organization, which frequently makes their execution time a matter of O(n). If we have one of these or several of these and they grow very large, it can become a very time consuming process to search from them.

A method to alleviate this issue is to use what is called a **binary tree**.

A search in a binary tree can be performed by moving from the top to bottom, making it much faster to look for an element.

*Hash Table* – this is a combination of an array and linked list. Elements are assigned and placed into another position in their array, called its *BUCKET*.

As seen on the worksheets, we can use #DEFINE to give a name and value pair, and this can help us circumvent some of the limitations for the separate value pairs.

#define TYPE double – for example. Thus, we can use TYPE to represent the type of value of the container. This helps us get around the limitations of C and include certain symbolic things that might C would be otherwise unable to comprehend.

Sizeof – a useful method to help in memory allocation. (malloc()) – if you need 10 spaces reserved, for example you would do int array = malloc(sizeof(TYPE) \* 10);

1. What is abstraction? Give three examples of abstraction from real life.

Abstraction is removing or compressing of unimportant (at that moment) information and hiding it from view.

For example, Atlases, Dictionaries, and Encyclopedias are all abstractions.

1. What is information hiding? How is it related to abstraction?

Information hiding conceals less important or pertinent elements from view. It relates to abstraction as it works for computer science because it leaves only the most important aspects of the program visible to the user and programmer.

1. How is encapsulation related to abstraction?

Encapsulation is limiting the focus of a method or in higher level languages, a class, and it helps by keeping the information set in one place, returning only pertinent data necessary for the next step of a program.

The key consequence of this process is that the encapsulation can be viewed in two ways, from the inside and from the outside. The outside view is often a description of the task being performed, while the inside view includes the implementation of the task.

4. Explain how a function can be viewed as a type of encapsulation. What information is

being hidden or abstracted away?

A function is a form of encapsulation, as it has internal information that is not shared with anything else in the program while a process is run through. The code outside the package only needs to know a return rather than the internal workings.

5. What makes an ADT description abstract? How is it different from a function

signature or an interface?

An ADT description usually defines what is going to go into the data object or collections of data, rather than actual parameter processes of the program like you find in a signature or interface.

6. Come up with another example from everyday life that illustrates the behavior of

each of the six classic abstractions (bag, stack, queue, deque, priority queue, map).

Bag: Bag of marbles

Stack: Pile of dishes

Queue: The line at a movie theater or theme park.

Deque: Peas in a straw, or a Chinese finger trap.

Priority Queue: A to-do list broken down by urgency

Map: A dictionary or atlas.

7. For each of the following situations, describe what type of collection seems most

appropriate, and why. Is order important? Is time of insertion important?

1. The names of students enrolled in a class.

“Bag” or “List” – order of insertion and time is not as important as other data types.

1. Files being held until they can be printed on a printer.

“Queue” – time of insertion is critical and should be as low as computationally possible. Order is also critical. First in, first out. Time of insertion is rigid. One after the other.

1. URLs for recently visited web pages in a browser.

“Stack”- time of insertion depends, order is important. Last in, first out. Rigid time of insertion.

1. Names of patients waiting for an operating room in a hospital emergency ward.

“Priority Queue” – time of insertion is critical, because if the queue needs to be re-arranged and patients with certain qualities are needed to be prioritized, o(n) may be unacceptable. We want to reduce time. Also, if we are talking about literal time, then possibly not as higher priority patients get moved to the front of the queue.

1. Names and associated Employee records in a company database.

“Map” – time of insertion is not critical, given that the employee is looked up by search function. Order also is unimportant.

8. In what ways is a set similar to a bag? In what ways are they different?

Both are unordered, but a set will have certain restrictions. A bag is thoroughly unorganized and without restriction.

9. In what ways is a priority queue similar to a queue? In what ways are they different?

A priority queue maintains values of importance, whereas a queue is simply whatever or whomever shows up first. A priority list jumps things with higher priority to the front of the queue.

10. Once you have completed worksheet 14, answer this question and the ones that follow. What is a dynamic array?

A dynamic array is an array that has no set size or capacity and expands to meet the needs of the computation being performed. Whereas the capacity of an array is restricted by its size, the capacity and size of the array are divorced from each other. There is also the use of a pointer to serve as a reference to the memory being stored.

11. What does the term capacity refer to in a dynamic array?

The amount of elements that can actually be held in the dynamic array.

12. What does the term size refer to in a dynamic array?

The current number of elements in the dynamic array.

13. Can you describe the set of legitimate subscript positions in a dynamic array? Is this different from the set of legal positions in an ordinary array?

Legitimate subscript positions in a dynamic array are anything that does not exceed the size of the array. Otherwise, the computer will try to fill it in itself. A fixed array would simply return an error.

14. How does the add function in a dynamic array respond if the user enters more values than the current capacity of the array?

Assuming the dynamic array is set up to handle capacity overflow, it would need to store the original members of the array, replace them in a new array that has an increased capacity, and then delete the current array. This results in an O(n) execution time issue.

15. Suppose the user enters 17 numbers into a dynamic array that was initialized with a capacity of 5? What will be the final length of the data array? How many times will it have been expanded?

It depends on the array size, and how it increases its capacity. If it does it incidentally (by number), it will have 17, but most logically, it would double its size each time capacity is reached or increase by a nominal increment (IE 5 or 10), so odds are highly likely it would be 20.

16. What is the algorithmic execution time for the \_dyArrayDoubleCapacity, if n represents the size of the final data array?

O(n)+ based on the number of elements that have to be processed.

17. Based on your answer to the previous question, what is the worst case algorithmic complexity of the function dyArrayAdd?

O(n)+ depending on what the issue is.

**Chapter 6: Stacks**

By now Stacks should be fairly standard and familiar territory. Remember, that stacks are like a pile of plates. And the only item we are allowed access to is the one at the very top. A stack follows LIFO, or Last-in-First-Out principles.

We are well aware that in real life, you can lift up a stack of dishes and get the plate you want, but let’s forget that for now and remember we have two actions with a stack. Put something in, take something out. Whatever the last thing you put in was, will be the first thing you will take out.

Refer to the list of commands that are common for stacks in the review sheet to know their purpose, but most stacks are interacted strictly with:

Push(newEntry), Pop(), Top(), and isEmpty(). Most of these are fairly self explanatory, but merit attention.

Stacks exist in nearly every programming language, and every single high level one. How you interact with the stacks varies from language to language, but the effects are fundamentally the same. The key difference is the key or command word used to interact with the samesaid stack.

Stacks work by adding an object or bit of data on top of the previously entered piece of data. To remove data, you take the top item off. Then you can add more, remove more, or leave it alone as you so choose. However, in a rigid stack, it will only remove the top most object.

**Question**: The user of a web browser can also move to a new page by selecting a

hyperlink. In fact, this is probably more common than using either the back or forward

buttons. When this happens how should the contents of the back and forward stacks be

changed?

When you hit forward and back, the current page works as a node to hold the current “data”. When this happens, you swap positions on the top of the stack with the back and forward stacks.

**Question**: Web browsers often provide a *history* feature, which records all web pages

accessed in the recent past. How is this different from the back stack? Describe how the

history should change when each of the three following conditions occurs: (a) when the

user moves to a new page by pressing a hyperlink, (b) when the user restores an old page by pressing the back button, and (c) when the user moves forward by pressing the

forward button.

1. By clicking the hyperlink, the user will “push” a new page to the top of the stack.
2. When the user moves backwards, the current page is put into the top of the ‘forward’ stack. This stack can be completely eliminated if the user clicks a different hyperlink.
3. When the user moves back, the previous website is removed from the memory “node” and pushed to the top of the forward stack.

**Question**: What should be the effect if the user enters a backspace key and there are no characters in the input?

You should get a null error.

**Question**: What should (or what does) happen if there is no available space in memory

for a new activation record? What condition does this most likely represent?

This will likely result in a fault or program termination. This represents a stack overflow.

**Question**: Show the state of the stack after each character is read in the following

expression: ( a b { c } d ( [ e [ f ] g ] ) ( j ) )

Some important applications for stacks include character balancing and polish notation. If you took CS162 last term or CS271, you might remember this.

It is represented in the stack by groupings and applying the operator to the grouped numbers.

Polish notation is also called postfix notation. It is surprisingly efficient once you understand it.

Conversion of Infix (3+4) to post-fix 34+ is actually quite easy in terms of stack usage.

Left parenthesis: push it to the stack

Operand: write to output and manipulate the integers/floats/whathaveyou

Right Parenthesis: Pop stack until corresponding left parenthesis is found. When stack is empty, terminate/report error. Otherwise, just keep going.

This may sound very complex, but when it’s broken down into composite parts, it’ll be quite simple.

One of the most important concepts covered in this section is what is called *amortized* execution time. When dealing with amortized execution time, we are actually dealing with the infrequent worst case scenario, when we make assumptions of the faster average execution time. Notation, based on Worksheets 15 and 16, shows that *amortized execution time* is written as such: O(n)+.

Amortization usually occurs when we are dealing with dynamic arrays or other dynamic data types. When the capacity of such a structure is reached, then we have to rearrange and replace the information contained in the ADT. When that happens, we have to slow down to make sure that everything is done –again-, in a logical and memory safe way.

Another important concept is *linked list array* or *linked list implementation.*

**IMPORTANT: This is an important anecdotal note based on the experience of some CS/EECS friends of mine – know linked list arrays backwards, forwards, inside out, and upside down. If you are going to work anywhere with high level programming, they WILL ask you questions regarding Linked Lists. -- Andrew**

This is an alternative implementation of a dynamic array. Each container within the array is a reference to a collection. And each link maintains two data field – one is the actually value or information and a reference to another link. The last link stores a null value.

One of the major advantages of a linked list is that it will keep growing as large as necessary and the increase in items is O(1), making it preferable to a dynamic array in many ways as you NEVER EVER have to copy an entire data block.

They can be challenging; however, to set up. There are head and tail, and intermediate nodes. In other words, it a lot of pointers.

**Memory Management:**

The linked list and dynamic array are two ways to approach problems involving memory management. When we are dealing with memory, we have to take into consideration the amount of actual physical memory a computer has. While this is becoming less and less of a problem as computers with upwards of 8 GB of RAM and incredibly powerful processors are becoming commonplace and cheap, good memory management is always a critical aspect of programming.

Memory leaks, which many popular programs and applications such as Firefox, Microsoft Word, Chrome, Facebook, games such as World of Warcraft, Skyrim, etc. have caused diminished performance and ultimately program or system freezes if not addressed or the application memory is maxed out. The most frequent result of a memory leak of that magnitude is program termination, but if that is not handled correctly, it may result in a systems crash. This was an issue on older computers; however, it still does occur on newer machines.

Given that C does not have garbage collection, the onus on memory management is on us. We should free and deallocate any used memory when necessary. In other words, clean up your mess – *the kindergarten principle*.

1. What are the defining characteristics of the stack abstraction?

Last In First Out Implementation.

2. Explain the meaning of the term LIFO. Explain why FILO might have been equally

appropriate.

Last in, first out. First in, last out effectively means the same thing. The first thing to go into a stack, especially if more things are added to the stack, it will be the last thing to be removed when the stack is terminated or made empty. Any LiFO and FILO behavior is totally stack legitimate if the behavior is maintained.

1. Give some examples of stacks found in real life.

A stack of dishes, web browser histories (backwards and forward), the Towers of Hanoi, etc.

1. Explain how the use of an activation record stack simplifies the allocation of memory for program variables. Explain how an activation record stack make it possible to perform memory allocation for recursive procedures

.

5. Evaluate the following postfix polish expressions:

a. 2 3 + 5 9 - \* = (2 + 3) \* (9 -5 ) = 20 or -20.

b. 2 3 5 9 + - \*

6. How is the memory representation of a linked list different from that of a Dynamic

Array?

A dynamic array requires an o(N) execution time action to increase its capacity after it has been reaching. A linked list always grows at a constant rate when things are added to it. Whereas a dynamic array keeps blocks of information stored in specific arrays, a linked list does not require arrays and simply holds the information in memory w/o arrays.

7. What information is stored in a link?

The link maintains two fields, one with a value and another with a reference to the other link.

8. How do you access the first element in a linked list? How would you get to the second element?

A linked list is not necessarily sequential in memory, but there is always a reference value to the next node. By accessing the node, you get to the second element.

9. In the last link, what value is stored in the next field?

A null value field.

10. What is the big-Oh complexity of pushing a value on the front of a linked list stack?

O(1)

Of popping?

O(1) – all data handled in a linked list runs as a constant time.

11. What would eventually happen if the pop function did not free the first link in the

list?

A memory leak. Eventually you would hit a segmentation fault.

12. Suppose you wanted to test your linked list class. What would be some boundary value test cases? Develop a test harness program and execute your code with these test cases.