**Review and Research**

You can review the video of Session 5 here:

[Session 5 - Data Structures #1](https://blizzard.sharepoint.com/portals/hub/_layouts/15/pointpublishing.aspx?app=video&p=p&chid=8aa7fa80-bfa2-4021-bf44-543dba93f693&vid=33195f79-9e89-48da-a88f-4e24e5e9885e)

As always, email me at [semerson@blizzard.com](mailto:semerson@blizzard.com) if you have any questions!

Here’s the MSDN documentation for each of the data structures we discussed in class:

Queue

<https://msdn.microsoft.com/en-us/library/7977ey2c(v=vs.110).aspx>

Stack

<https://msdn.microsoft.com/en-us/library/3278tedw(v=vs.110).aspx>

HashSet

<https://msdn.microsoft.com/en-us/library/bb359438(v=vs.110).aspx>

Dictionary

<https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx>

I strongly encourage you to check out this tutorial, which describes the differences between value types and reference types in C# in a fair amount of detail. We’ll review this material in class as well:

<http://www.tutorialsteacher.com/csharp/csharp-value-type-and-reference-type>

For those interested in reading more about hash functions and the kinds of data structures that employ them, here’s a nice basic tutorial:

<https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/>

And here’s a nice beginner’s guide to Big O notation if you want to learn more:

<https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>

**Glossary**

Queue – A data structure that employs FIFO (first in, first out) data access: if we place items A, B, and C into the Queue in that order, then we begin to request items, A, B, and C will be removed from the Queue and returned to us in that order.

Stack – A data structure that employs LIFO (last in, first out) data access: if we place items A, B, and C into the Stack in that order, then we begin to request items, C, B, and A will be removed from the Stack and returned to us in that order.

Hashing function – A function that takes some form of input and, through various mathematical processes, returns a new value derived from the input. Hashing functions are used in numerous programming applications including data structure indexing, file integrity checking (see, e.g., [MD5](https://en.wikipedia.org/wiki/MD5)), and sophisticated cryptography.

Hash Set – A data structure that stores a collection of unique items by employing a hash function to produce indices at which to place them. Because this type of hash function is usually fairly simple, a Hash Set can find stored items very quickly no matter how large the set becomes.

Dictionary – A data structure that stores key / value pairs and maps a value onto a key. For example, a Dictionary<string, int> might be used to associate a person’s name with her age. There are many data structures that employ similar behavior, including maps, associative arrays, and symbol tables.

Value type – In C#, a value type is any type that holds its own data in memory. If we instance a value type object and then instance a second object and set it equal to the first, the second object will make a copy of the first object’s data and store it in memory. If you modify the first object, the second object will not be affected because they are distinct copies. A copy of this kind is called a *deep copy*, since an actual copy of the data is made.

Reference type – In C#, a reference type is any type that stores a pointer to specific data object in memory. If you instance a reference type, initialize it with a new object, and then declare a second reference type and set it equal to the first, both references will refer to the same, single object. If you modify the first reference, the changes will apply to the second reference as well. A copy of this kind is called a *shallow copy*, since we haven’t actually copied the data being referenced, only the reference itself.

Struct – In C#, a struct is similar to a class. There are a couple of differences, but probably the most important one is that a struct is a value type, whereas a class is a reference type. We generally prefer to use structs when we need an object to hold a small number of simple public fields.

Call stack – Programming languages use call stacks to keep track of where they are in execution. Each time a function calls another function, the new function is pushed onto the stack; when the program exits a function, that function is popped off of the stack and execution returns to the point in the next function down at which the exiting function was called.

Recursive function – A recursive function is a function that calls itself as part of its execution. Recursion can be tricky to reason about and takes some study and practice to master, but it’s a fundamental technique that we can use in all kinds of applications. For example, compilers like Visual Studio invariably use recursive algorithms to parse the code we write and transform it into whatever new form it needs to take.

Big O Notation – This is a shorthand that we use to describe the general complexity of an algorithm, usually relative to some variable value. For example, if an algorithm has O(1) complexity, this means that it runs at constant time and can be relied upon to stay at about the same level of complexity regardless of other factors. But if an algorithm has O(n) complexity, it runs at linear time; it becomes more complex as ‘n’ becomes larger.

**Problem Solving Techniques**

In Session 4, we discussed a number of techniques you can use when attempting to solve a problem with code. I’m including them here as well for quick reference. If you’re having difficulty working out how to complete a practice exercise, review these ideas and see if you can find ways to apply them to the problem at hand!

* Understand the problem before you begin planning a solution. Make sure you know what kind of input to expect and what should be done with that input. Think about possible edge cases and how you might deal with them. Most of all, make sure you’re solving the right problem!
* Plan before you code. It is tempting to jump straight into your development tools and begin writing code immediately, but a bit of planning will ultimately save you time, make your code less error-prone, and help prevent the frustration of not knowing how to proceed. If you find that you hit a wall when writing your code, don’t be afraid to stop and re-examine your plan (and revise it if necessary). When planning, you don’t necessarily need to focus on language features; think about the problem in real-life terms, and feel free to sketch out your ideas on paper to help make them more concrete.
* Break big problems down into smaller problems. When we implemented bubble sort in class, we first thought about how to sort a single number in an array rather than *all* the numbers. We then tackled how to repeat the process, how to manage the number of iterations in each loop, how to know when to stop, and how to swap values. By thinking about solutions to each of these smaller problems in their turn, we chip away at the larger problem until we have a workable solution.
* Use concrete examples when you plan. If you’re trying to devise a sorting algorithm, don’t just think about sorting in the abstract: Write down simple sets of actual numbers (or other values) to work with, and from those specific examples begin to create generalized solutions. Problem solving is almost always easier to do when you work with concrete values rather than jumping straight into purely abstract concepts!
* Consider extreme cases, i.e., extremely large cases or extremely small cases. This is useful for solving all kinds of problems! Consider this simple, classic riddle: You have a drawer with 10 red socks, 12 blue socks, and 5 yellow socks inside. If you begin pulling socks out of the drawer at random without looking at the color, how many socks must you take to guarantee that you grab at least one red sock? One way to easily come up with a general solution to this kind of riddle is to use an extreme case. Let’s go for an extremely small case and imagine we have exactly one sock of each color. It’s very easy to see that, in order to be sure you get the red sock, you have to take all three socks out of the drawer; until the yellow sock and blue sock have been removed you’re not guaranteed to get the red sock. Once we understand this, it’s easier to generalize for other cases: To be guaranteed a red sock, you must take the total number of yellow and blue socks, plus one more. Believe it or not, in software engineering we often solve very complex problems using just this kind of thinking! When choosing concrete examples to work with, don’t be afraid to start with extremely simple cases; the solution is usually easier to find, and will often scale up to work for more complex cases as well.

**Practice Exercises**

These exercises are grouped around concepts we’ve covered in class and range from very simple to more complex. As always, a few reminders:

1. When compiling and running these exercises in Visual Studio, be sure to do so in Debug Mode. You can do this with the hotkey combination ctl+F5.
2. These exercises are meant to be a start to your practice, but if you want more ideas, contact me and let me know. I’ll be happy to make additional suggestions!
3. Don’t be afraid to repeat an exercise several times. Repetition in your practice is very helpful, especially in the beginning when you’re trying to get comfortable with syntax as well as new concepts.
4. If you get stuck on a problem or don’t understand why something is happening, please contact me and let me know. I’ll be happy to help.
5. An exercise may occasionally require you to use a concept we haven’t covered in class; when this is the case, the exercise will be marked with an asterisk there will be a link to research you can do help you find the new information you need.
6. Don’t be afraid to use Visual Studio’s debugging features to help you understand how your program is working! This can be useful not only for fixing problems but also for gaining a better understanding of how your program is working.

*Data structures and problem solving*

NOTE: Some of these exercises are a little more complex than those in previous lessons. Read each description carefully and plan before attempting them. If the instructions for any of the exercises don’t make sense feel free to let me know and I’ll be happy to clarify!

Ex. 1. Create a struct called Person, with a public string field for name and a public unsigned int field for age. Instance a Queue of Persons and try enqueueing several Persons. Then dequeue each person and write their name and age to the console.

Ex. 2. Create a function that accepts a string as input and outputs an encrypted version of the string. Each character in the original string should be explicitly converted to an integer, decremented by 1, and converted back into a character. Additionally, the characters in the new string should be in reverse order compared to the original string (try using a Stack for this.) For example, if the input is “Penguin” then the output string should be “mhtfmdO”. Each letter has been shifted down by one and the entire string is reversed.

Ex. 3. Follow-up: Write a decryption function that, given a string encrypted using the function in Ex. 2, will return a decrypted string. If we give this function “mhtfmdO” as input it should output “Penguin”.

Ex. 4. Write a function that, given a List of strings, sorts the list into alphabetical order and removes any duplicate strings. Use a HashSet to help you with the second task.

Ex. 5. Write your own hashing function! Try a simple one that takes a string as input and returns an integer between 0 and 100, inclusive, as output. To do this, try converting each character in the string to an integer, summing all the integers, then using the remainder operator to convert the sum to a value in the specified range.

Ex. 6. Declare and initialize a new Dictionary<string, int>. Now have your program enter a loop and, each cycle, do the following:

* Ask the user to enter a person’s first name and store the name in a string (hint: Look up “Console.ReadLine()” to learn how to get user input and assign it to a variable).
* Check to see if the name exists as a key in the Dictionary.
* If not, add the name to the Dictionary and map it to an integer value that increases with each name added, starting with 0. For example, the first name added should map to value “0.” The second name added should map to value “1.” You can use a counter to keep track of these values. Have the program write a message to the console indicating that the name was added.
* If the name entered *does* exist in the Dictionary, write a message to the console that indicates that the name was found and report back the value associated with that name.
* Return to the beginning of the loop and continue.
* Hint: To keep the loop from going on forever, you can provide an escape input: For example, when the program asks the user for a name, if the string entered is “exit”, break the loop and exit the program.

Here’s a brief sample transcript of how the console might look as the program is run. User input is in italics:

Please enter a name:

*Alice*

I’ve added Alice to the Dictionary.

Please enter a name:

*Bob*

I’ve added Bob to the Dictionary.

Please enter a name:

*Alice*

Alice exists in the Dictionary with value: 0

Please enter a name:

*Bob*

Bob exists in the Dictionary with value: 1

Please enter a name:

*Exit*

Goodbye!

Ex. 7. BlizzCon is almost upon us, and your task is to create a class called TicketVendor that ensures that early buyers get BlizzCon tickets. Modify the Person struct in Ex. 1. and provide it with a public bool field called “hasTicket”. Provide your TicketVendor class with a private Queue<Person> called “ticketQueue” *and* a private List<Person> called “completedList.” Now provide your class with the following public methods:

* AddPerson - Add Persons to the queue. You might want to provide your Person struct with a constructor to make this a little easier.
* TryTicket - Attempt to give a Person in the queue a ticket. Since there are only 5 BlizzCon tickets this year (budget cuts), only the first five Persons who were added to the queue will get them. This method should dequeue a Person from the queue, and if there are any tickets left, set that Person’s “hasTicket” field to true. Otherwise set “hasTicket” to false. Regardless of whether or not the Person got a ticket, add them to the completedList when you are finished.
* Report – This method should go through each Person in the completedList (if any) and report that person’s name, age, and whether or not they received a ticket to BlizzCon.

To test your new class, instance a TicketVendor, add 10 Persons, and try to ticket all 10 of them. Then run a report and make sure the first five Persons who were added to the queue received tickets (and that nobody else did.)