COMP110: Principles of Computing

8: Data Structures I

Administration

- Research Journal Peer Review is **today**
- Final deadline is **soon (check MyFalmouth)**

FALMOUTH UNIVERSITY

Arrays and lists

Memory allocation — recap

- Memory is allocated in **blocks**
- The program specifies the size, in bytes, of the block it wants
- The OS allocates a **contiguous** block of that size
- The program owns that block until it frees it
- Blocks can be allocated and deallocated at will, but can **never grow or shrink**

Collection types

- Memory management is hard and programmers are lazy
- Collections are an **abstraction**
  - Hide the details of memory allocation, and allow the programmer to write simpler code
- Collections are an **encapsulation**
  - Bundle together the data's representation in memory along with the algorithms for accessing it

Arrays

- An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- Each array element has an **index**, starting from zero
- Given the address of the 0th element, it is easy to find the ith element:

$$\text{address}_i = \text{address}_0 + (i \times \text{elementSize})$$

- E.g. if the array starts at address 1000 and each element is 4 bytes, the 3rd element is at address $1000 + 4 \times 3 = 1012$
- Accessing an array element is **constant time** $O(1)$

Lists

- An array is a block of memory, so its size is **fixed** once created
- A **list** is a variable size array
- When the list needs to change size, it **creates** a new array, **copies** the contents of the old array, and **deletes** the old array

Arrays and lists in C#

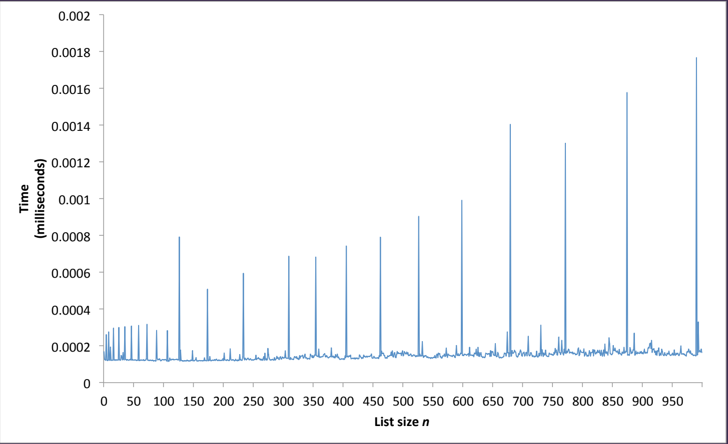
```
int[] myArray = new int[10];

int[] myOtherArray = new int[] { 2, 3, 5, 7, 11 };

List<int> myList = new List<int>();

List<int> myOtherList = new List<int> { 2, 3, 5, 8, 13 };
```

Time taken to append an element to a list of size *n*



Operations on lists

- **Appending** to a list is **amortised constant time**
  - Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size
- **Inserting** anywhere other than the end is **linear time**
  - Can't just insert new bytes into a memory block — need to move all subsequent list elements to make room
- Similarly, **deleting** anything other than the last element is **linear time**

Basic data structures in C#

Classes and interfaces

- A **class** in C# defines constructors, destructor, methods, properties, fields, ...
- An **interface** defines methods and properties which a class can implement
- An interface is a little like a fully abstract class
- A class in C# can only **inherit** from one **class**, but can **implement** several **interfaces**

IEnumerable

- Most collection types in C# implement the `IEnumerable<ElementType>` interface
- This interface allows us to iterate through the elements in the collection, from beginning to end
- C# provides the `foreach` loop as a convenient way of using this
- $\Longrightarrow$ any class which implements `IEnumerable<ElementType>` can be used with a `foreach` loop

Arrays

```
int[] myArray = new int[10];
int[] anotherArray = new int[] { 123, 456, 789 };
```

- `int[]` is an array of `int`s
- In general, `ElementType[]` is an array of values of type `ElementType`
- Size of the array is set on initialisation with `new`
- Array **cannot change size** after initialisation
- Use `myArray[i]` to get/set the `i`th element (starting at 0)
- Use `myArray.Length` to get the number of elements

Multi-dimensional arrays

```
int[,] myGrid = new int[20, 15];
```

- `int[,]` is a 2-dimensional array of `int`s
- Use `myArray[x, y]` to get/set elements
- Use `myArray.GetLength(0)`,`myArray.GetLength(1)` to get the “width” and “height”
- Similarly `int[,,]` is a 3-dimensional array, etc.

Lists

```
using System.Collections.Generic;

List<int> myList = new List<int>();
List<int> anotherList = new List<int> { 1, 2, 3, 4 };
```

- Like a list in Python, but can only store values of the specified type (here `int`)
- Append elements with `myList.Add()`
- Get the number of elements with `myList.Count`

Strings

```
string myString = "Hello, world!";
```

- `string` can be thought of as a collection
- In particular, it implements `IEnumerable<char>`
- So for example we can iterate over the characters in a string:

```
foreach (char c in myString)
{
    Console.WriteLine(c);
}
```

Strings are immutable

- Strings are **immutable** in C#
- This means that the contents of a string **cannot be changed** once it is created
- But wait... we change strings all the time, don't we?

```
string myString = "Hello ";
myString += "world";
```

- This isn't changing the string, it's creating a new one and throwing the old one away!
- Hence building a long string by appending can be slow (appending strings is $O(n)$)
- C# has a **mutable** string type: `StringBuilder`

Dictionaries

- Dictionaries are **associative maps**
- A dictionary maps **keys** to **values**
- Takes two generic parameters: the **key type** and the **value type**
- A dictionary is implemented as a **hash table**

Using dictionaries

```
var age = new Dictionary<string, int> {
    ["Alice"] = 23,
    ["Bob"] = 36,
    ["Charlie"] = 27
};
```

Access values using `[]`:

```
Console.WriteLine(age["Alice"]);  // prints 23
age["Bob"] = 40;          // overwriting an existing item
age["Denise"] = 21;       // adding a new item
age.Add("Emily", 29);     // adding a new item -- will raise
                          // an error if already present
```

Iterating over dictionaries

- `Dictionary<Key, Value>` implements `IEnumerable<KeyValuePair<Key, Value>>`
- `KeyValuePair<Key, Value>` stores `Key` and `Value`

```
foreach (var keyValue in age)
{
    Console.WriteLine("{0} is {1} years old",
                      kv.Key, kv.Value);
}
```

- (C# tip: the `var` keyword lets the compiler automatically determine the appropriate type to use for a variable)
- Dictionaries are **unordered** — avoid assuming that `foreach` will see the elements in any particular order!

Hash sets

- Sets are **unordered** collections of **unique** elements
  - Sets **cannot** contain **duplicate** elements
  - Attempting to `Add` an element already present in the set does nothing
- `HashSet`s are like `Dictionary`s without the values, just the keys
- As discussed in Week 5, certain operations are much more efficient (constant time) on hash sets than on lists

Using sets

```
var numbers = new HashSet<int>{1, 4, 9, 16, 25};
```

Add and remove members with `Add` and `Remove` methods

```
numbers.Add(36);
numbers.Remove(4);
```

Test membership with `Contains`

```
if (numbers.Contains(9))
    Console.WriteLine("Set contains 9");
```

FALMOUTH UNIVERSITY

Stacks and queues

Stacks and queues

- A **stack** is a **last-in first-out (LIFO)** data structure
- Items can be **pushed** to the **top** of the stack
- Items can be **popped** from the **top** of the stack

- A **queue** is a **first-in first-out (FIFO)** data structure
- Items can be **enqueued** to the **back** of the queue
- Items can be **dequeued** from the **front** of the queue

Implementing stacks

- Stacks can be implemented efficiently as lists
- Top of stack = end of list
- To push an element, use `Add` — $O(1)$ complexity
- To pop an element we can do something like this:

```
x = myStack[myStack.Count - 1];
myStack.RemoveAt(myStack.Count - 1);
```

- This is also $O(1)$

Implementing queues

- Queues can be implemented as lists, but not efficiently
- End of list = back of queue
- Enqueue using `Add` — $O(1)$ complexity
- Dequeue by retrieving and removing from beginning of list:

```
x = myQueue[0];
myQueue.RemoveAt(0);
```

- This is $O(n)$

Implementing queues

- End of list = front of queue
- Dequeue is like popping from end of list — $O(1)$ complexity
- Enqueue using `Insert(0, x)` — $O(n)$ complexity

Using stacks and queues

- C# has `Stack` and `Queue` classes which you should use instead of trying to use a list
- Python has `deque` (double-ended queue) which can work as either a stack or a list

Stacks and function calls

- Stacks are used to implement **nested function calls**
- Each invocation of a function has a **stack frame**
- This specifies information like **local variable values** and **return address**
- Calling a function **pushes** a new frame onto the stack
- Returning from a function **pops** the top frame off the stack
- Hence the term **stack trace** when using the debugger or looking at error logs