Introduction to Data Structures

Albert Gural

October 28, 2011

1 Introduction

When trying to convert from an algorithm to the actual code, one important aspect to consider is how to store and manipulate data. Specifically you need to be able to figure out what data structure to use for such a purpose. Often, you will be able to use pre-written data structures in an API for your program, although sometimes you may want to define your own.

For this lecture, I will partially cover two common programming language's syntaxes - `C/C++` and `Java`. Because most of you are probably more familiar with `Java` (thanks to the APCS curriculum), I'll focus a bit more on syntax and examples with `Java`.

2 Declaring and Defining Variables

Whenever you want to use a data structure, you will first need to declare it:

```
int i;
```

then define it:

```
i = 10;
```

These statements can be combined into a single statement:

```
int i = 10;
```

Note that in `C++`, you don't need to define a variable (but not defining a variable is very dangerous - it could lead to spurious data).

3 Which Structure to Use

This is a very important question to ask yourself when writing your program. In order to figure this out, you'll need to first consider what the structure must be capable of doing. Since you all attended our lecture on complexity last week, you'll also need to know the efficiencies of certain operations like insertion and deletion from the end or middle of the data structure. (Note: the distinction between "end" and "middle" is because some data structures aren't as easily indexed, and so inserting at the middle, for example, could take longer than $O(1)$.)

...and now for the actual data structures...

4 Primitives and Strings

Primitive data types are the most common ones used in any program. They are generally very fast, and proficiency in most of the primitive types is essential to success in programming.

4.1 Numbers

type (Java/C++)	size	function	range
boolean/bool	1B	Flag	true/false
char	1B	Character	'' - '$\breve{2}$55'
short	2B	Short Integer	$(-2^{15}) - (2^{15} - 1)$
int	4B	Integer	$(-2^{31}) - (2^{31} - 1)$
long/long long	8B	Long Integer	$(-2^{63}) - (2^{63} - 1)$
float/double	4B	Floating Point Number	$(-2^{31}) - (2^{31} - 1)$
double/long double	8B	Double Precision Float	$(-2^{31}) - (2^{31} - 1)$

4.2 Strings

Strings can be thought of simply as an array of characters - although the two aren't always exactly synonymous. However, you don't have to declare a string with the same syntax as that of an array. The general way to declare a string is to delineate the string with quotes:

```
String s = ''Hello World!'';
```

In contest programming, strings are useful for outputting debug data as well as essential for certain problems that ask for non-numeric output.

4.3 Containers

Often, primitive data structures are not powerful enough to accomplish the entire programming task. For the rest, you'll need more powerful data structures. What follows in this lecture are a series of data structures that are containers of the primitive types we've seen so far.

5 Arrays

An array is an indexed collection of (identical type) data structures. For example, a character array of size 10 is made by allocating 10 bytes of continguous stack space. The ith byte can be indexed by adding i to the address of the first byte for the array. In `C++`, you can see this in action with pointers. In `Java` and in `C++`, you can index an element with the bracket operator. Let's look at a specific example in `Java`.

```
int[] squares = new int[10];
for(int i = 0; i < squares.length; i++)
    squares[i] = i * i;
```

`squares`	0	1	4	9	16	25	36	49	64	81

We can notice several things about how arrays are declared and instantiated. Note that arrays are 0-indexed. That means the first element is `array[0]`, not `array[1]`. In `Java`, the array's size is given by `array.length`. Another way to instantiate arrays is the following:

```
int[] squares = {0, 1, 4, 9, 16, 25, 36, 49, 64, 81};
```

In this way, we are explicitly giving the values in the array. Notice, however, that this version is a bit less flexible. You have to know exactly what you want to place in the array at compile time.

5.1 Higher Dimensional Arrays

Suppose instead of an array of integers, we used an array of arrays. In a sense, this would give a higher dimensionality. For example, instead of just a 10-length array, we could have a 10×10 array of elements. Declaring and instantiating such a structure is exactly how you might expect:

```
int[][] timesTable = new int[10][10];
for(int i = 0; i < timesTable.length; i++)
    for(int j = 0; j < timesTable[0].length; j++)
        timesTable[i][j] = i * j;
```

Notice a few things. For this two dimensional array, we use double bracket operators. Really, this is just an `int[]` array, which is given as `int[][]`. Also notice that we use `array.length` for the highest level, and `array[0].length` for the next highest level. (Why is this?)

5.2 Important Facts

Arrays are a nice data structure because of how simple and light weight they are. They're useful for storing large amounts of data in any number of dimensions.

However, arrays do have certain limitations. You can only store one type of data in all cells of an array. For example, you can't have an array containing an integer, character, and string (unless you do something crazy like make an array of Objects...). Another problem is that you can't resize an array. Once it's made, the size is fixed. However, you can always make a new array that's bigger and copy the elements in from the old array, at the expense of some time.

6 Lists

6.1 Linkedlists

One problem with arrays is that inserting elements takes $O(n)$ time. An alternate structure is to have each element of data stored in some container that points to "the next" container.

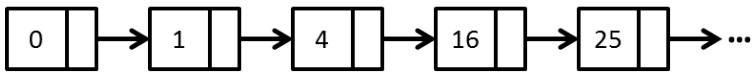

Figure 1: Linked List Storing Squares Data

So instead of finding the fifth element by indexing `array[4]`, you would start at some head container and go to the container it points to, then the container that that one points to, etc, four times. The beauty is that once we're at this point, we can easily insert an element by changing two pointers.

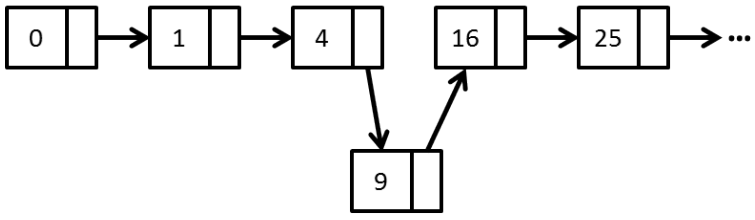

Figure 2: Linked List Inserting an Element

6.2 Arraylists / Vectors

Arraylists try to combine the best of array's fast indexing with linked lists's fast insertion at the end. In essense, an Arraylist is just an array. But instead of being of just the right size for the data, it is purposely made larger. That way, when data is added, the array doesn't need to keep resizing. As long as the number of resizes is kept to a minimum, appending elements to Arraylists can be thought of as a constant-time operation. Inserting and deleting from the middle of an arraylist, however, is still $O(n)$.

7 Queues and Stacks

We already learned a bit about queues and stacks from the Basic Graph Theory lecture. Both structures are often implemented off of a Linked List data structure because we only care about insertion and deletion (pushing and popping) from the ends of the structure.

7.1 Queue

A Queue is a FIFO (first-in first-out) structure. Similarly to how restaurants take orders, the first orders in a queue get processed first, while the later ones are always processed after earlier ones. Hopefully this visual helps explain the process:

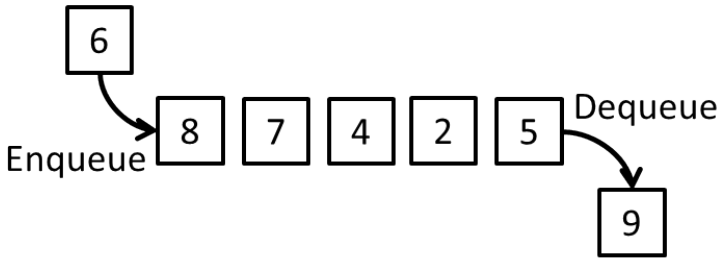

Figure 3: A Queue Data Structure

7.2 Priority Queues

A Priority Queue is like a queue in that you push elements in and there's a perfered element to get popped first. However, unlike ordinary queues, priority queues pop the element with highest priority. (Internally, a Priority Queue is based on a heap, not a linked list, but you won't need to know that). Hopefully this image will help explain Priority Queues:

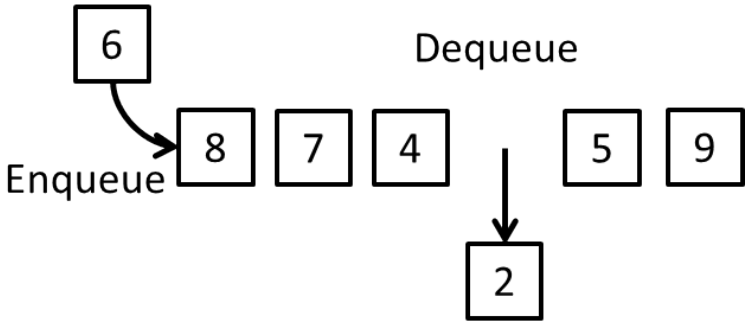

Figure 4: A Priority Queue Data Structure

7.3 Stacks

A Stack is a LIFO (last-in first-out) structure. Similarly to how people take plates from a stack of plates, the last plate put in a stack (the one on top) is the first plate removed. This visual should help explain the process:

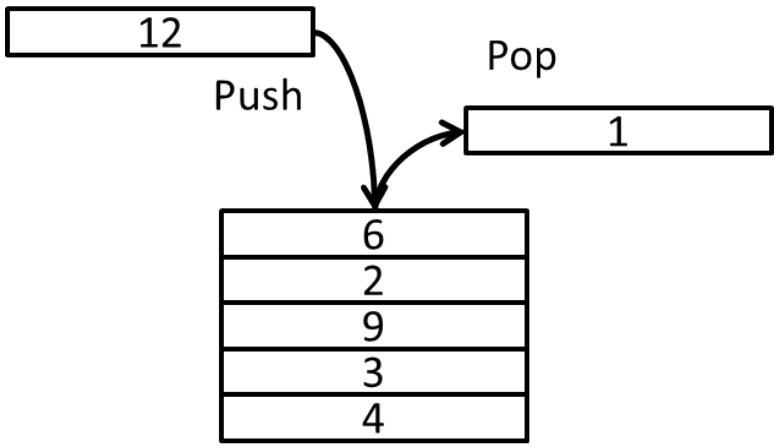

Figure 5: A Stack Data Structure

8 Other and Custom Data Structures

There are many other data structures. The most useful data structures that haven't been covered in this lecture are probably sets and maps. See if you can understand how they work and how to use them. Of course, there are even more advanced data structures.

Besides using premade data structures, it's also very important to be able to make your own. For example, if you wanted to have an array that stored an integer and a string paired together for each element, you would need to make a new data structure to hold an integer and a string, and make an array of those data types. In `Java`, you have to make a new class:

```
class MyDataStructure {
    int myID;
    String myName;

    public MyDataStructure(int id, String name) {
        myID = id;
        myName = name;
    }

    public int getID() {
        return myID;
    }

    \\ more stuff...
}
```

In `C++`, if you're just pairing a couple of elements together, you can use the `Pair` structure:

```
pair<int, string> myDataStruct;
myDataStruct.first = 1;
myDataStruct.second = "Hi!";
```

For more complicated structures, you could create a class, but the preferred method is to use a `struct`.

```
struct myDataStruct {
  int myID;
```

```
  string myName;
  double myCode;
  char myChar;
};
```

Make sure to remember to include the closing semicolon. Well, with that, it's time to look at a summary of all the data structures.

9 Summary of Data Structures

Here is a Big-O efficiency chart for various actions on the basic container (storying data type E) data structures we've seen so far.

1. Indexing Element
2. Modifying Element
3. Inserting (end)
4. Deleting (end)
5. Inserting (middle)
6. Deleting (middle)
7. Searching for an Element

Big-O of Various Operations on Various Data Structures

Type	1	2	3	4	5	6	7
Array	1	1	n	1	n	n	n
Linked List	n	n	1	1	n	n	n
Array List	1	1	1	1	n	n	n
Queue	/	/	1	1	/	/	/
Priority Queue	/	/	$\log n$	$\log n$	/	/	/
Stack	/	/	1	1	/	/	/

10 Problems

1. Suppose we have a graph and are given as input all the edges that connect nodes a and b by a distance of d. Devise a data structure to store this data.
2. We're trying to use floodfill to determine the size of a given flood on some field. What data structure should we use? How should we utilize it?
3. Using these data structures to your full advantage, devise an $O(n \log n)$ sorting algorithm, without using recursion. What is the *space* complexity of your algorithm?
4. Postfix and Prefix notations are alternatives to the more common Infix notation used in math. In prefix notation, 2 + 3 would be written as + 2 3. Given some prefix expression, devise an algorithm to evaluate the given expression. What data structure should you use?
5. You want to write a program to calculate the nth Fibonacci number ($1 \leq n \leq 60$). What data types should you use in your program?