Exploring advanced programming concepts

Sjors van Gelderen

January 22, 2017

Contents

1 Introduction

In this document I describe the materials studied during my graduation phase. The aim of the project was to prepare for professionally teaching students selective aspects of computer science. These aspects include:

- The C# and Python 3 programming languages
- Object-oriented design patterns
- Data structures, algorithms and complexity analysis

I have provided many example implementations in a variety of programming languages. All provided complexities in this document and related implementations are worst-case complexities.

1.1 Primary languages

Here is a brief description of the main languages used.

1.1.1 C#

C# is Microsoft's most popular .NET programming language. The language strikes me as a much higher level variant of C++. With its rich .NET ecosystem, modern features (LINQ, anonymous methods, tasks, etcetera), C# is certainly a powerful programming language.

Sources C# code is provided in the subdirectory *csharp/*.

1.1.2 Python 3

Advantages of this language include its concise syntax and its portability. Because of Python 3's high-level nature, the programmer can focus exclusively on the actual logic of an algorithm, rather than the low-level memory management involved. This is removes a significant source of potential distraction. Unfortunately, the absence of a strict compiler means frequent run-time debugging sessions are necessary.

Sources Python 3 code is provided in the subdirectory *python_3/*.

1.1.3 F#

Being a more recent addition to the .NET family of programming languages, F# has not quite gained the popularity of C#. The F# programming language enables the programmer to tackle problems using high-level concepts such as recursion, higher order functions, partial application and currying, computation expressions (syntactic sugar for monads) and more.

Sources F# code is provided in the subdirectory *fsharp/*.

1.2 Secondary languages

These languages were only used sparingly, out of curiosity rather than necessity.

1.2.1 C

Virtually any programmer will at some point in their career encounter this venerable, fast and portable programming language. Because this low-level language doesn't use a garbage collector, the programmer must exercise great caution with the manual management of memory. Many security problems that affect us today are a direct consequence of a failure to do so. This language is well-suited to studying the low-level implementation of algorithms and data structures.

Sources C code is provided in the subdirectory *bonus/c/*.

1.2.2 Rust

Developed by Mozilla, Rust aims to be a modern solution for safe, asynchronous programming. With default immutable variables, as well as borrowing and lifetimes, the compiler makes it very difficult indeed to write a program that contains run-time errors relating to incorrect memory access.

Sources Rust code is provided in the subdirectory *bonus/rust/*.

1.2.3 Chicken Scheme

The LISP family of programming languages has two major dialects; these being Common LISP and Scheme. Chicken Scheme is a modern implementation of the Scheme dialect. It has a very minimalistic syntax, revolving around the use of parentheses and prefix notation. Chicken Scheme is a functional programming language.

Sources Chicken Scheme code is provided in the subdirectory *bonus/chicken_scheme/*.

2 Analysis

2.1 Empirical analysis

Empirical analysis refers to inferring a program's expected performance based on measurements taken while running a program with various configurations. A typical scenario involves the measurement of performance in terms of running time by the use of a 'stopwatch'.

2.2 Complexity analysis

Complexity analysis is different from empirical analysis in that it uses reasoning - rather than experiment - to determine a program's expected performance.

3 Data structures

As the manner in which data is stored affects the efficiency of and compatibility with a given algorithm, it is appropriate to discuss the studied data structures now.

3.1 Array

Among the most common data structures for collections is the array. All elements in the array are stored contiguously in memory, and may be accessed in constant time($O(1)$) through their respective indices. Arrays have excellent cache-alignment, making them very fast indeed.

The location of an element in an array at some given index i will be

$$base\_address\_of\_array + i * s$$

In which s is the size of a single element. This size depends on the data type of the array.

3.1.1 Size

Fixed size By default, arrays are generally of *fixed size*; meaning the array will take up constant space (*O(1)*). Since less elements might be stored in the array than its total capacity allows for, space might be wasted on unused memory. The amount of elements that are actually being used is called the *logical size* of the array.

Dynamic size When an array is of dynamic size, the programmer must carefully specify the amount by which the array will be resized. If the programmer adds more elements than some given threshold will allow, the array must perform a resize operation. Since the resize operation is costly (typically $O(n)$), frequent resizes ought to be avoided. In some implementations, dynamic size arrays will also shrink when the logical size becomes less than a given threshold.

3.1.2 Dimensionality

Arrays may have more than one dimension. Such a construction may also be called a matrix. Elements in the multidimensional array may be accessed with multiple indices describing the relevant coordinates.

3.2 Linked list

The *linked list* is a linear, dynamic size data structure for storing collections of elements. Contrary to arrays, elements are not guaranteed to be stored contiguously. This makes it possible to store elements in a fragmented fashion, which is useful when the collection is larger than any available contiguous block of memory. Since elements in the linked list are stored in a fragmented fashion, direct access through indices as done with the array becomes impossible.

The linked list consists of several segments, each of which has up to two references to other elements in the sequence. The last element of the sequence will point to an empty segment, which may be called a *null link*. Traversing the linked list is a linear time ($O(n)$) operation.

Singly-linked and doubly-linked The *singly-linked* list consists of segments that contain a value and a reference to the next segment in the sequence. The *doubly-linked* list is the same as the singly-linked list, except each segment also has a reference to the previous segment in the sequence. Whether the list is singly-linked or doubly-linked affects the traversal process, since where singly-linked lists only allow forward traversal, doubly-linked lists also allow backward traversal.

Insert Elements are inserted at the root of the list. Since no traversal is required to locate the root, the insertion operation is one of constant time $O(1)$. The inserted element becomes the new root, and is given a reference to the next segment. This next segment is the old root segment.

When inserting at the end - rather than the beginning - of a list, the list must be traversed until a given segment references a null link as the next element in the sequence. The reference to the empty segment may be replaced with a reference the element to be inserted. Due to the traversal, the worst case complexity of such an insertion is of linear time $O(n)$. If the list is *doubly-linked*, the new element's *previous* variable should reference the last element of the current list.

Delete Locating the element to be deleted will in the worst case require a complete traversal of the list, which is a linear time ($O(n)$ operation. If the current segment's *next* variable references the element to be deleted, it must be changed to point to the segment in the sequence that succeeds the element to be deleted (which may be *null*). If the root element is deleted, the next segment in the list becomes the root.

3.3 Stack

The *stack* is a *LIFO* (Last In, First Out) data structure. Data is always inserted on and removed from the top of the stack. Only the top element of the stack may be inspected at any time. Regardless of which structure is used to store the stack contents, the interface must always offer the following operations: *push*, *pop* and *peek*.

Push The *push operation* puts a new element on top of the stack. This is a constant time ($O(1)$) operation. If the stack has a size limit, this operation throws a *stack overflow* exception when the stack's capacity is exceeded.

Pop The pop operation removes the top element from the stack. This is a constant time ($O(1)$) operation. If the stack has no elements, this operation will throw a *stack underflow* or *invalid operation* exception, as there are no elements to remove.

Peek The peek operation gives the programmer access to the current top element of the stack. This is a constant time ($O(1)$) operation. If the stack is empty, this operation will throw an *invalid operation* exception, as there are no elements on the stack to reveal.

Sources Stack implementations are provided at the following locations:

- *csharp/data_structures/stack/*
- *python_3/data_structures/*
- *fsharp/data_structures/*
- *bonus/c/data_structures/*

3.4 Queue

Contrary to the stack, the queue is a *FIFO* (First In, First Out) data structure. Elements are inserted at the back of the queue, and removed from the front of the queue.

Enqueue The *enqueue* operation adds a new element at the front of the queue. This is a constant time ($O(1)$) operation.

Dequeue The *dequeue* operation removes the element at the back of the queue. This is a constant time ($O(1)$) operation for arrays, and a linear time ($O(n)$) for linked-lists.

IsEmpty The *is_empty* operation returns whether the queue is empty or not. This is a constant time ($O(1)$) operation. It can be used to prevent queue underflow.

Rear and front The *rear* and *front* operations provide access to the front and rear elements of the queue. These operations are optional and as such are not provided in all implementations. Both operations are constant time ($O(1)$) operations for arrays. The rear operation is linear time ($O(n)$) and the front operation is constant time ($O(1)$) for linked-lists.

Sources Queue implementations are provided at the following locations:

- *csharp/data_structures/queue/*
- *python_3/data_structures/*
- *fsharp/data_structures/*
- *bonus/c/data_structures/*

3.5 Hashmap

This data structure - also known as a *dictionary* or *hash table* - is used to store a list of key-value pairs. This kind of structure is also called an *associative array*. The data structure consists of a list or array of *buckets* or *slots* containing one or more elements. The amount of elements that may be stored in a given bucket depends on the implementation of said bucket. The underlying data structure may, for example, be a fixed-size array or a linked-list.

Of primary interest is the way the indices of the relevant bucket are determined. Unlike standard arrays, indices need not be integers. Indices may instead be derived from any kind of data type. This is done using a *hash function* (hence, the name of the data structure).

The hash function derives an integer from the provided data. The modulo operator then restricts the provided integer to a range that fits in the array or list.

```
hash_value = hash_function(data)
hash_map_index = hash_value % hash_map_size
```

To determine whether the number of buckets is appropriate for the amount of elements to be inserted, the load factor may be calculated. The load factor is calculated as follows:

$$number\_of\_entries / number\_of\_buckets$$

A large load factor is indicative of an inappropriate amount of buckets. If the load factor is too small, however, it is possible a lot of buckets are unnecessarily taking up space. Some implementations of hash maps will resize when the load factor reaches a certain threshold. This is a costly operation, seeing as all elements will have to be redistributed according to the new size.

It is possible that two different pieces of data generate the same hash map index. Such a situation is called a *collision*. If there are many collisions, the distribution of the values will be poor. As a consequence, the data structure will be inefficient. If all keys are known before the data is inserted into the hash map, it is possible to generate a perfect hashing function. This means each piece of data provided will result in a unique hash map index.

There exist several solutions to the collision problem, three of which are described in the next sections.

3.5.1 Linear and quadratic probing

When performing *linear probing*, a collision causes the hash map index to be increased successively by one, until an empty bucket is found. Similarly, *quadratic probing* successively increases the hash map index using values from a quadratic polynomial to find a new hash map index for the colliding element. Usually, both of these probing methods will wrap around if the resulting hash map index is outside of the range of the bucket array. These methods may cause elements to occupy buckets that would otherwise be populated by other pieces of data.

In addition, these probing methods may cause infinite loops when an acceptable bucket is never found.

3.5.2 Dynamic size buckets

Another solution to the collision problem is to allow multiple values per bucket. Such a solution could be implemented by making each bucket into a dynamic size array or linked-list. If hash function or load-factor is bad, this implementation may result in similar efficiency to simply using a single dynamic size array or linked list.

3.6 Tree

Trees are hierarchical data structures consisting of *linked nodes*. Unless the tree is empty, there will be a top node called the *root node*. From this node spring all the *subtrees*.

Any node other than the root node of the tree is called a *child node*. The predecessor of a child node is called its *parent*. The root node is the only node that does not have a parent.

Nodes that don't have any children are called *leaves*, expanding on the analogy of the tree. Any node that has at least one child is an *internal node*.

A tree is said to be *balanced* if the nodes are distributed evenly among the subtrees in the tree. A tree might be so poorly balanced that the nodes are organized much like a *linked list*. In this case, it's possible that the tree was not a suitable data structure for the relevant problem.

The *depth* of a node is the amount of *edges* that must be traversed from the root to reach it. A collection of edges to traverse is called a *path*, the length of which is the amount of edges. All nodes of the same depth are said to be on the same *level*.

In this section are described a number of data structures that may be called trees.

3.6.1 Binary tree

The binary tree - also called *bifurcating arborescence* - consists of nodes containing at most one parent, and at most two children. There is no particular property governing where new elements are inserted. This is because the binary tree does not inherently imply any particular sorting order.

A binary tree is said to be *full* if all of its leaves are on the same level, and each internal node has two children. This is different from the notion of completeness, as a *complete* binary tree is a tree where each level other than possibly the last is fully populated, and all nodes are as far to the left as possible.

Sources A binary tree implementation is provided at the following location:

- *fsharp/data_structures/*

3.6.2 Binary heap

This complete binary tree maintains the *heap property*, which is said to be *max* or *min*. A heap that satisfies the *max heap property* is called a *max heap*. Accordingly, a heap that satisfies the *min heap property* is called a *min heap*. The property determines the order in which the elements are stored inside the heap.

In a max heap each key must be greater than every key stored inside its children. Conversely, in a min heap each key must be less than every key stored inside its children.

Insert The *insert* operation adds a new key at the end of the heap. If the heap property is violated, it may be restored by letting the key 'swim' to the correct position. When 'swimming', the key is compared to its parent. If the a violation of the heap property is found, the key and its parent are swapped. This process is repeated until the heap property is no longer violated. Note that the heap property is also restored if this key becomes the new root of the heap.

Extract and heapify The *extract* operation removes the root node from the binary heap. It does this by replacing the root key with the last key in the heap. After this step, the *heapify* operation is used to restore the heap property if it is be violated by the new root. In such case, the root is compared to its children. Depending on the relevant property, the root is swapped with either its smaller child in a min-heap, or its larger child in a max-heap. The process is then repeated from the old root's new location, which is the index of the child it was swapped with. Eventually, the heap property must be restored.

Sources Heap implementations are provided at the following locations:

- *csharp/data_structures/heap/*
- *python_3/data_structures/*
- *fsharp/data_structures/* (At the time of writing incomplete)