COMP110: Principles of Computing

7: Data Structures I

Assignments

- Peer review **next Thursday**
- Please come to the session with a **draft** of your research journal

- Worksheet 5 due **tomorrow**
- **No new worksheet** this week — work on your research journal instead

FALMOUTH UNIVERSITY

Turing machines

Turing machines

- Introduced in 1936 by Alan Turing
- Theoretical model of a "computer"
    - I.e. a machine that carries out computations (calculations)

Turing machine

- Has a finite number of **states**
- Has an infinite **tape**
- Each space on the tape holds a **symbol** from a finite **alphabet**
- Has a **tape head** pointing at one space on the tape
- Has a transition table which, given:
  - The current state
  - The symbol under the tape head

  specifies:
  - A new state
  - A new symbol to write to the tape, overwriting the current symbol
  - Where to move the tape head: one space to the left, or one space to the right

The Church-Turing Thesis

- If a calculation can be carried out by a mechanical process at all, then it can be carried out by a Turing machine
- I.e. a Turing machine is the most "powerful" computer possible, in terms of what is possible or impossible to compute
- A machine, language or system is **Turing complete** if it can simulate a Turing machine

Computability

Computability theory

- Let A and B be **sets** of elements
  - NB: A may be **infinite**
- A function $f : A \to B$ is **computable** if there exists a Turing machine which computes f
  - I.e. given an encoding of $a \in A$ as input, the Turing machine outputs an encoding of $f(a)$

An uncomputable function

The **halting problem**

- A = the set of all Turing machines (encoded as transition tables)
- $B = \{\text{true}, \text{false}\}$
- $f(a) = \begin{cases} \text{true} & \text{if } a \text{ halts in finite time on all inputs} \\ \text{false} & \text{otherwise} \end{cases}$
- There is **no** Turing machine that computes f
- f is **uncomputable**

Halting revisited

- Write a software tool that, given a Python program, predicts whether that program can go into an infinite loop
- Your tool must work for **all** Python programs
- Is this possible?

FALMOUTH UNIVERSITY

Arrays and lists

Memory allocation

- 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
- Forgetting to free a block is called a **memory leak** (rare in Python/C#, but a common bug in C++)
- Blocks can be allocated and deallocated at will, but can **never grow or shrink**

Containers

- Memory management is hard and programmers are lazy
- Containers are an **abstraction**
  - Hide the details of memory allocation, and allow the programmer to write simpler code
- Containers 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 *i*th element:

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

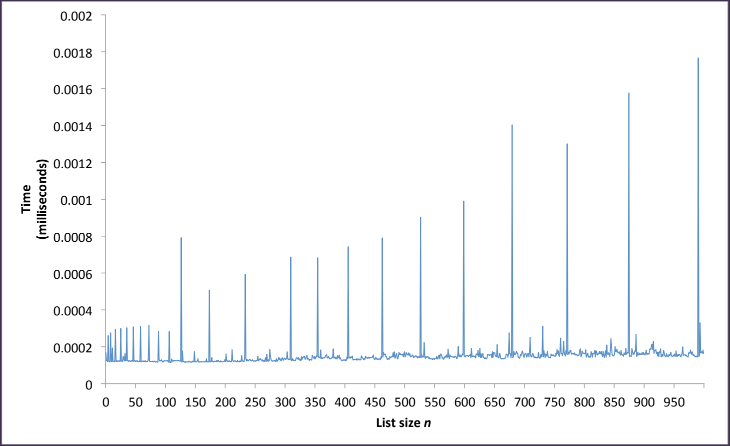
- E.g. if the array starts at address 1000 and each element is 4 bytes, the 3rd element is at address $1000 + 4 \times 3 = 1012$
- Accessing an array element is **constant time** $O(1)$

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
- Implementation details: `http://www.laurentluce.com/posts/python-list-implementation/`

FALMOUTH UNIVERSITY

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**

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

Stacks in Python

- Stacks can be implemented efficiently as lists
- `append` method adds an element to the end of the list
  - What is the time complexity?
- `pop` method removes and returns the last element of the list
  - What is the time complexity?

Queues in Python

- Queues can be implemented as lists, but not efficiently
- Could use `append(item)` to enqueue and `pop(0)` to dequeue
  - What is the time complexity of `pop(0)`?
- Could use `insert(0, item)` to enqueue and `pop()` to dequeue
  - What is the time complexity of `insert(0, item)`?
- `deque` (from the `collections` module) implements an efficient **double-ended queue**
- Provides methods `append`, `appendleft`, `pop`, `popleft`
  - All of which are $O(1)$

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

FALMOUTH UNIVERSITY

Pass by reference

References

- Our picture of a variable: a labelled box containing a value
- For "plain old data" (e.g. numbers), this is accurate
- For **objects** (i.e. instances of classes), variables actually hold **references** (a.k.a. **pointers**)
- It is possible (indeed common) to have **multiple references** to the same underlying object

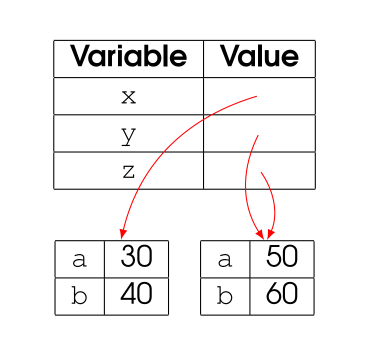
The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value	
x	a	30
	b	40
y	a	50
	b	60
z	a	50
	b	60

The right picture

```python
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```



Values and references

Socrative room code: `FALCOMPED`

```
a = 10
b = a
a = 20
print("a:", a)
print("b:", b)
```

Values and references

Socrative room code: `FALCOMPED`

```
class X:
    def __init__(self, value):
        self.value = value

a = X(10)
b = a
a.value = 20
print("a:", a.value)
print("b:", b.value)
```

Values and references

Socrative room code: `FALCOMPED`

```
class X:
    def __init__(self, value):
        self.value = value

a = X(10)
b = X(10)
a.value = 20
print("a:", a.value)
print("b:", b.value)
```

Pass by value

In **function parameters**, "plain old data" is passed by **value**

```
def double(x):
    x *= 2

a = 7
double(a)
print(a)
```

`double` does not actually do anything, as `x` is just a local copy of whatever is passed in!

Pass by reference

However, instances are passed by **reference**

```
class Box:
    def __init__(self, v):
        self.value = v

def double(x):
    x.value *= 2

a = Box(7)
double(a)
print(a.value)
```

`double` now has an effect, as `x` gets a reference to the `Box` instance

Lists are objects too

```
a = ["Hello"]
b = a
b.append("world")
print(a)   # ["Hello", "world"]
```

... which means you should be careful when passing lists into functions, because the function might actually change the list!

References can be circular

```python
class X:
    pass

foo = X()
foo.x = foo
foo.y = "Hello"

print(foo.x.x.x.x.x.y)
```

References and pointers

- Some languages (e.g. C, C++) use **pointers**
- Pointers are a type of reference, and have the same semantics
- References in other languages (e.g. C#, Python) are implemented using pointers
- C++ also has something called references, which are similar but different (pointers can be **retargeted** whilst references cannot)

Pointers

- Recall that memory is a series of 1-byte locations, each with a numeric **address**
- A **pointer** to something is simply the **address** at which it starts
- When allocating a block of memory, the OS returns a pointer to the start of the block
- When the memory is freed, any pointers into it are said to be **dangling**
- If the memory is subsequently reused for something else, those pointers could end up pointing to random data
- Again this is not really possible in Python/C#, but a common source of bugs in C/C++