Data Structures & Algorithms

Lecture 1: Computer Architecture

**TL/DR**

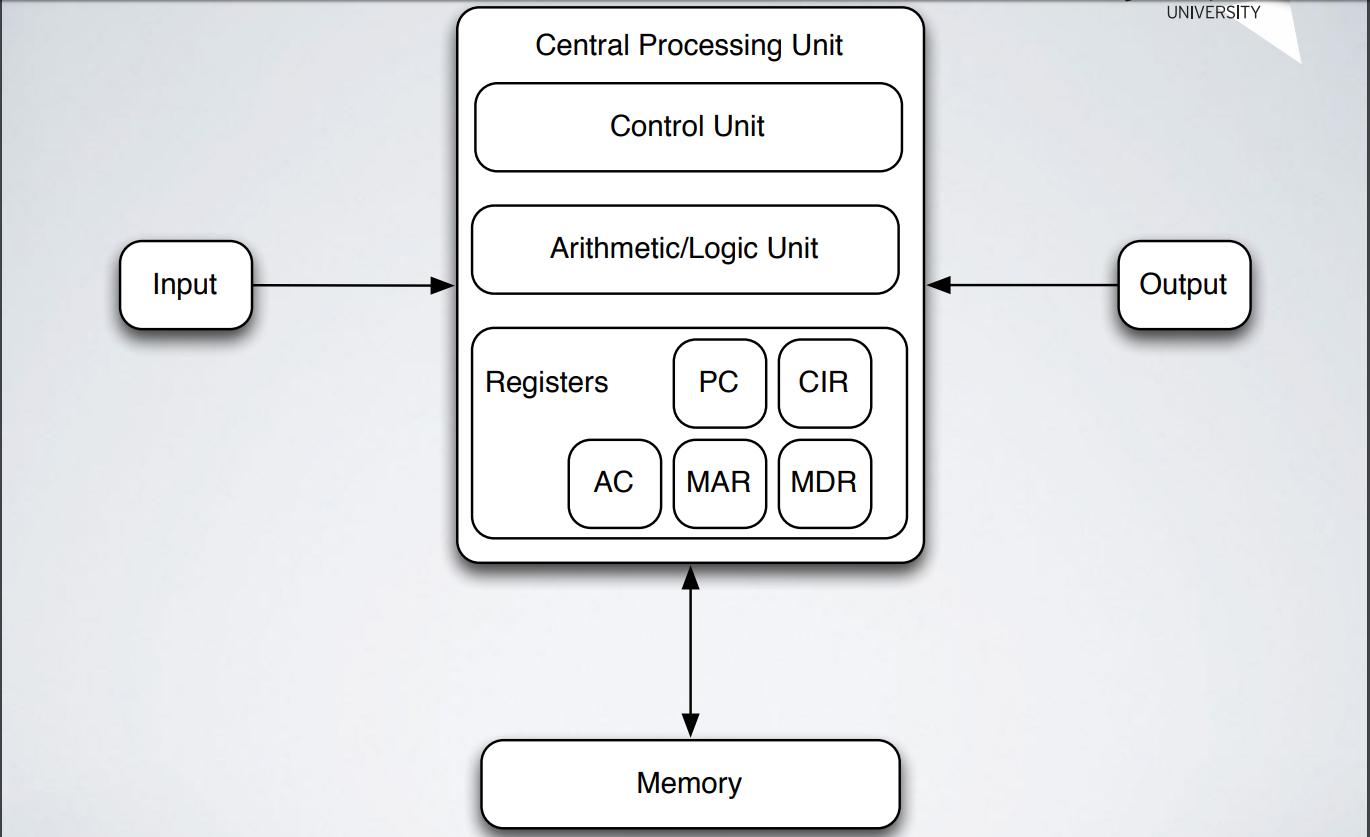
There are many Data Structures & Algorithms, as many as there are ways to organise & manipulate computer memory, but they are specified against an abstract machine and optimised for a specific architecture/implementation

**Data structures and algorithms**

* Data Structures are organised collections of data
  + How that organisation is achieved differentiates data structures
* Algorithms are recipes for manipulating
  + A series of operations that transform our initial data into a desired form

**John Von Neumann Architecture**

* Introduced in 1945
* Based on the first draft of the EDVAC machine (Early electronic computer)
* Still an accurate abstraction of modern computer hardware
* Design of a digital architecture is made up of
  + Central Processing Unit (CPU)
  + Arithmetic Logic Unit (ALU)
  + Processor Registers
  + Control Unit (containing instruction register and program counter)
  + Memory (storing data & instructions)
  + Mass storage
  + Input/Output (IO) Mechanisms
* All connected by busses



**Von Neumann Bottleneck**

* There is klimited bandwidth to move data from one location to another – if there is too much data then the limiting factor on speed is the bus – this is a known limitation of many computer architectures.

**Specification Vs Optimisation**

* Data structures and algorithms are specified against an abstract machine but optimised for a specific architecture/implementation
  + Consider the difference between describing an implementation of an array for an abstract machine versus implementing that array from scratch on a given piece of hardware, or adding an array implementation to an existing language

**Storing Data In Memory**

* Many types of hardware implementations of memory
* We often “idealise” the architecture & e.g. assume infinite memory (although we only use a finite amount at a time)
* Generally, memory is made up of discrete cells which individually store data.
* Cells are arranged in regular pattern and of fixed size
* Addressable – each cell can be uniquely referred to

**Data & Types**

* Depending upon your language, you may have some mix of:
  + Primitives [integers, characters, floats, doubles, strings, references]
  + Composite types [arrays, structs, records]
  + Linear types [arrays, lists, vectors, matrices]
  + Associative types [dictionary’s, maps, sets, tuples]
  + Abstract data types [Lists, stacks, queues, dequeues, trees, graphs]
  + Concrete data types (may or may not be identical to the abstract types)

**Pragmatism & Language Design**

* For several reasons, whilst a data structure has a theoretical shape the implementation must take account of practical and design issues:
  + Avoiding duplication (e.g. Python has lists but no more primitive collections like arrays)
  + Reusing existing data structures (the behaviour of stacks, queues, deques can all be achieved using the Python list methods)
  + Fitting with design of the language
  + Optimisation
* Sometimes we need to consider behaviour (practical), behaviour (ideal), performance trade-offs between the two.

**Structs**

* A composite type (in contrast to primitives)
* Aggregation of multiple (potentially differing) primitive datatypes into a single memory block that is referenced by a single variable
* Can contain pointers to other structures (used to build linked data structures)

**Sequential/Linear Structures**

* A way to organise primitive datatypes in relation to each other – as various kinds of sequences
* Could use individual items of data (e.g. variables), but this quickly becomes problematic
* Arrays {single/doubly}, Vectors, Matrices, (Linked) Lists

**Arrays**

* Composite datatype
* Linear data structure – elements form a sequence
* Static data structure
* Collection of values of the same type
* Stored contiguously in memory
* Can build an array of most primitive datatypes (including structs)
* Limitations:
* Basic, fixed size
* Limitations

**Algorithms**

* Can be used to do computation, process data, calculate results, reason
* Effective algorithms are expressed within a finite time and space using a well-defined language
  + 1. Start with an initial state & (possibly empty) input
  + 2. Proceed through a finite number of intermediate states by executing instructions
  + 3. To produce an output and termination state.

**Costs Of Computation**

* There are costs involved in computation:
  + Storing data uses memory
  + Finding data uses CPU
  + Moving data around uses CPU & memory
* All take **time** (abstraction from CPU usage)
* All use **space** (abstraction from memory usage)
* Data structures & algorithms is concerned with evaluating & trading off between time & space usage

Lecture 2: Algorithms & Complexity

**Algorithms**

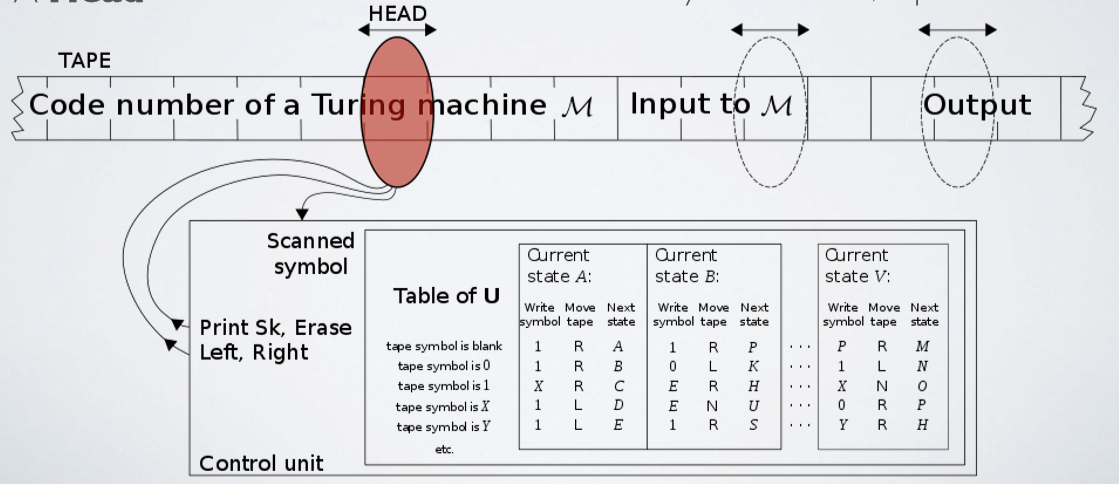
* We have algorithms for
  + Calculating results
  + Data processing
* They are also used in Artificial Intelligence – uses algorithms (Path-finding, Machine Learning, Neural Networks)
* An effective method expressed within a finite amount of space & time using a well-defined formal language for calculating a function.
* When executed there are a finite number of well-defined successive states that eventually produce an output and the computation terminates at a final ending state

**Effective Methods**

* Finite number of exact finite instructions
* When applied to a problem from its class:
  + It always terminates after a finite number of steps
  + It always produces a correct answer

**Turing Machines**

* An abstract machine that manipulates symbols on a strip of tape according to a set of rules
* A **Tape** of infinite length
* A **state register**
* A **finite Table of instructions**
* A **Head**
* Read tape, decode information, act on any instructions, repeat.



**Halting Problem**

* Determining from a description of an arbitrary program and input whether the program will finish running or continue forever.
* Phrased in terms of Turing machines:
  + Given a description of a turing machine and initial input, asks whether the program, when executed on the input, will halt (complete) or continue forever.
  + Benn shown that not possible to construct a turing machine that can answer this question.
    - e.g. have a function halts() into which we pass a program. Function then returns true if halts & false otherwise
  + Only way to know for certain is the run the machine & see what happens. If it halts then you know it halts,otherwise…?
  + Example of an undecidable or non-computable problem

**Classifying Algorithms**

**Big Oh Notation**

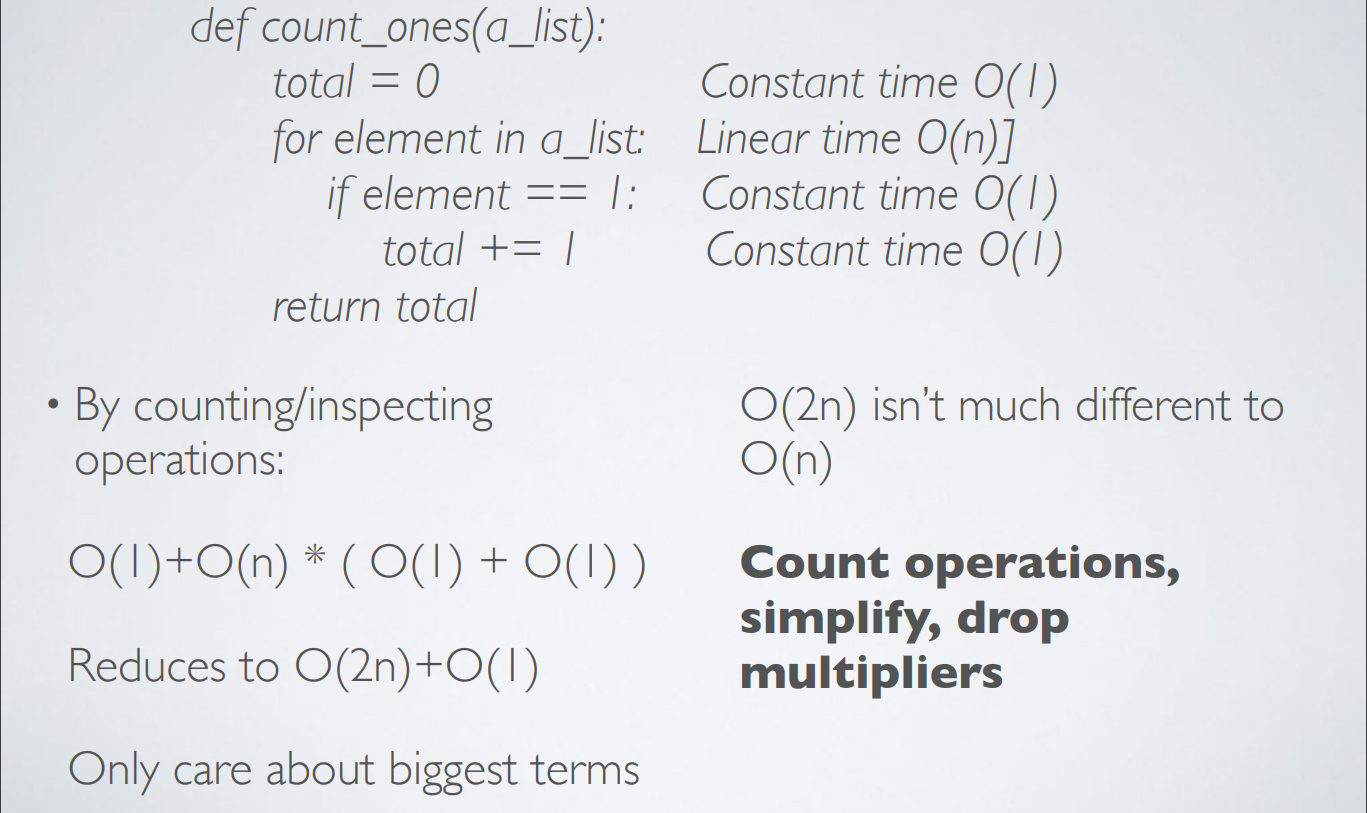
* Big Oh is just fancy sounding words for insight & practices that many progressional developers know and use
* Big Oh refers to the “order” associated with the performance, i.e. the degree of complexity, so O(n) is read “The order of n”
* O really refers to the Order function
* A function’s Big Oh notation is generally determined by how it responds to different inputs
  + E.g. How much slower is this function if we give it 1,00m00 items instead of 1 item?
* Essentially we are approximating orders of magnitude
  + i.e. Does the algorithm run in constant time, linear time, quadratic time, logarithmic time?
* This lets us predict how a given algorithm will perform for a given input size

**Other Notations**

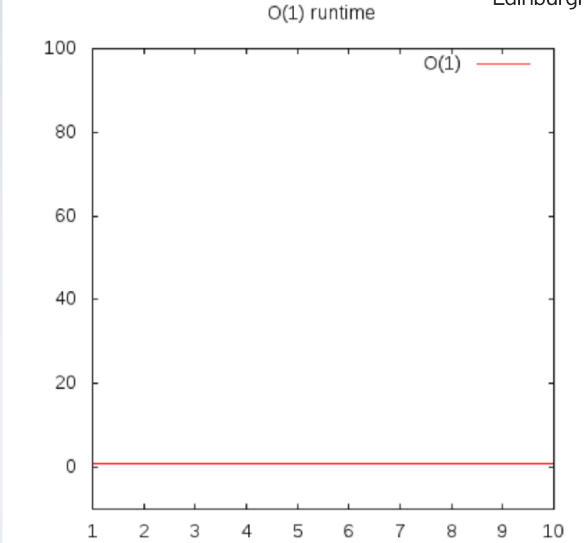
* Big Oh gives the upper bound
* Big Ω (Omega) gives the lower bound
* There is Big Ɵ (Theta) notation to asymptotically bound the growth to within constant factors above and below
* Important because a single notation doesn’t always give the full story
* Each notation can also be used to reason about best, worst, & average cases

**Calculating #1**

* We take measurements of how an algorithm performs
* Graph the results (where n the number of items corresponds to the x axis)
* Match the curve to known performance curves
* We graph the n in O(n) where n corresponds to x axis

**Calculating #2**

**Constant Time**

* An algorithm runs in constant time if it requires the same amount of time regardless of input size.
* Big Oh Notation/Complexity is O(I)
* No matter how big the input will always take the same amount of time
* Example: Access any element of an array, push & pop to a fixed size stack, Enqueue to & dequeue from a fixed size queue

**Linear Time**

* An algorithm runs in linear time if the time it takes to execute is directly proportional to input size
* Complexity is O(n)
* Examples:
  + Array: Linear search, Traversal, Find Minimum
  + ArrayList: Contains
  + Queue: Contains
* Call with, e.g. item\_in\_list(2, [1,2,3])
* If we graph the time it takes the function with different sized inputs (arrays) we’d see that this approximately corresponds to the number of items in the array

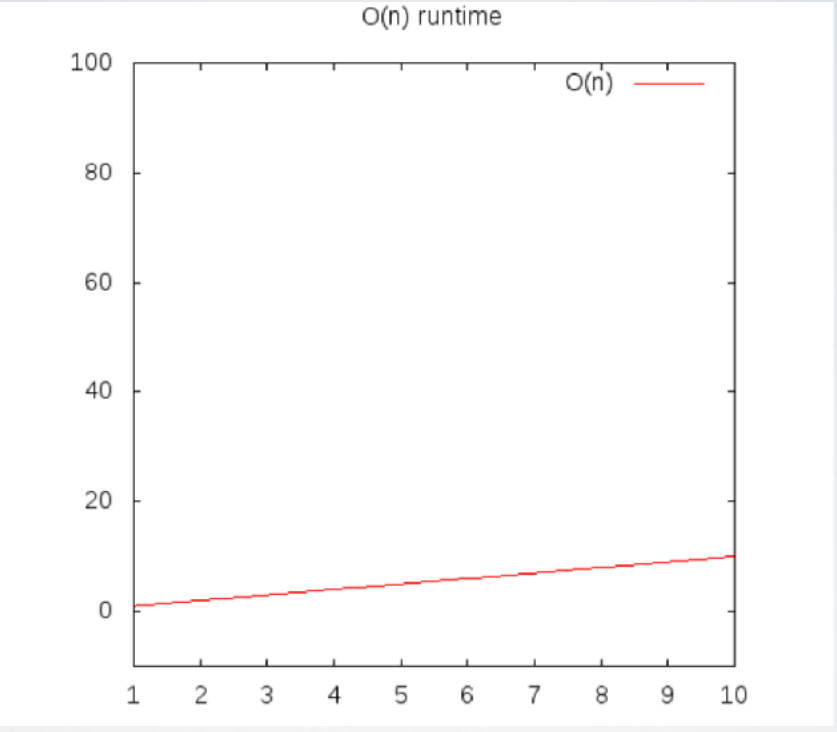
def item\_in\_list(to\_check, the\_list):

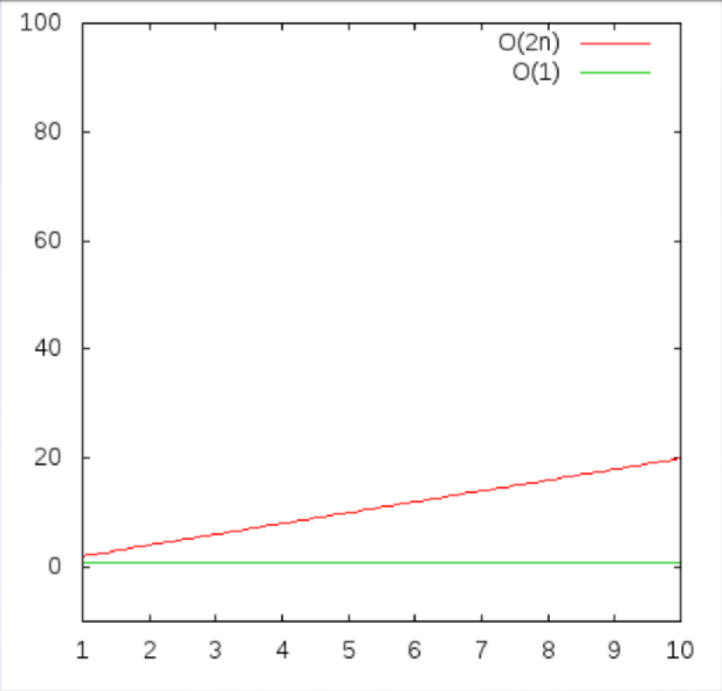
for item in the\_list:

if to\_check == item:

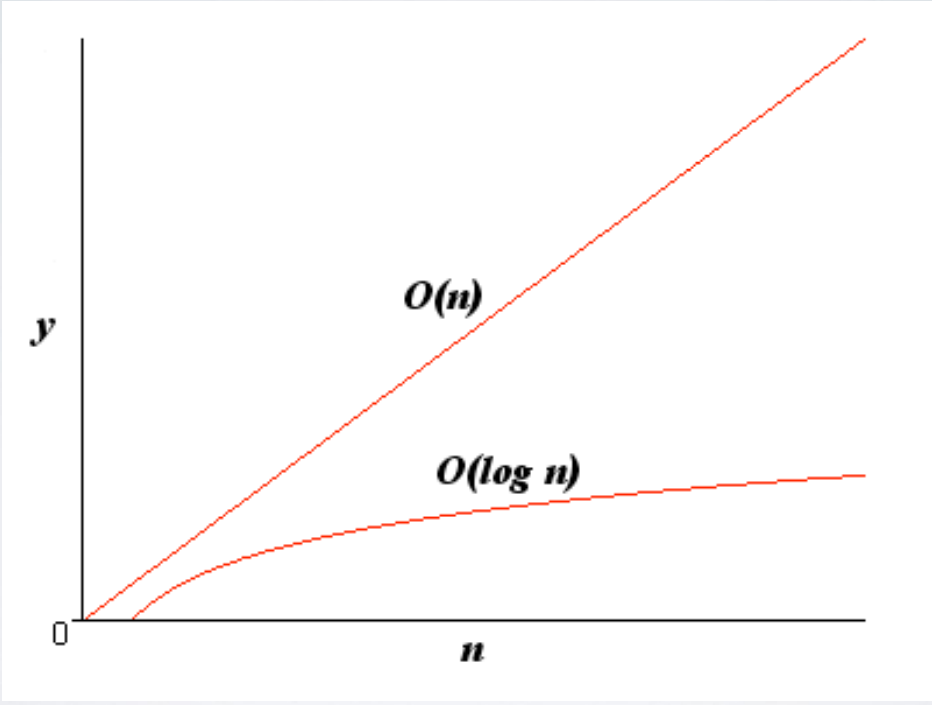
return True

return False



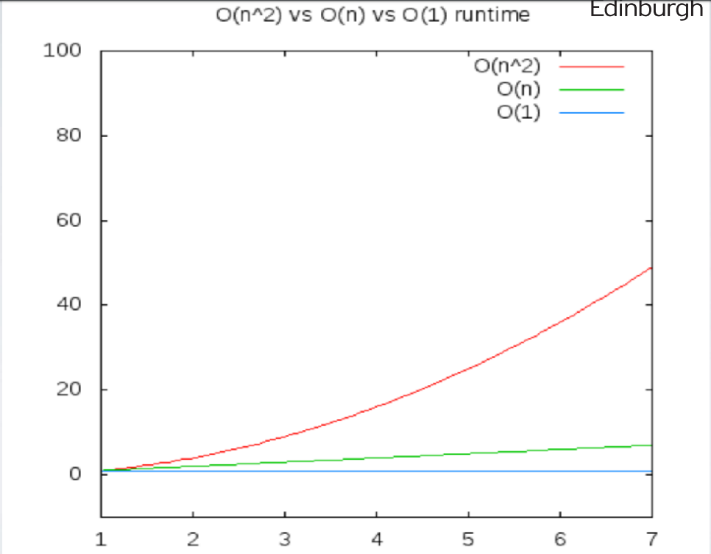
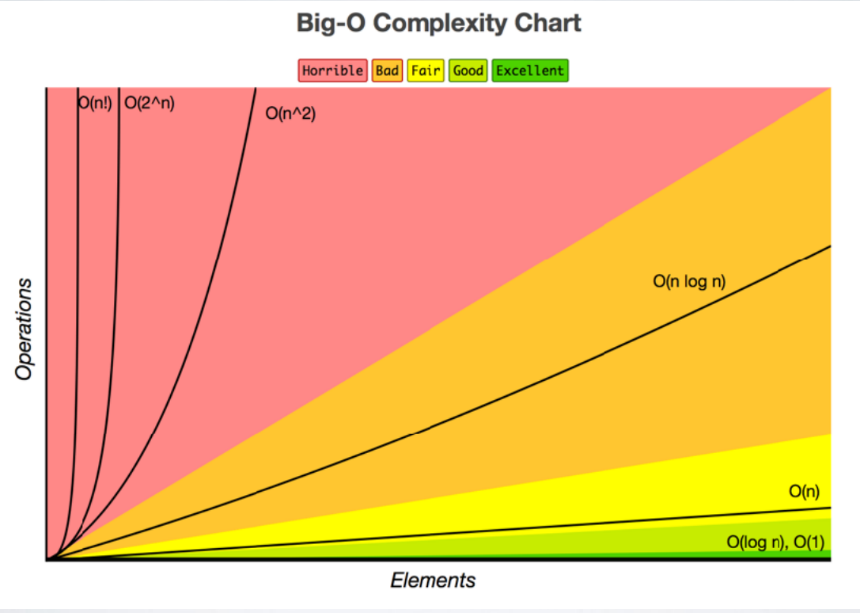


**Logarithmic Time**

* If the execution time is proportional to the logarithm of the input size.
* A common attribute of algorithms with logarithmic running times is that there is often a choice of new element on which to perform an action and only one needs to be chosen
* Example: Binary Search
* Classical “divide & conquer” scenarios

**Quadratic Time**

* An algorithm runs in quadratic time if its execution time is proportional to the square of the input size
* Given a list e.g. [1,2,3] get back all combinations:
  + [(1,1) (1,2), (1,3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
  + For every item, n, in the list we have to do n operations
  + n \* n == n^2, i.e. O(n^2)
  + Example: Bubble, Selection, & Insertion sorts



Lecture 3 Data Structures

**Data and memory**

* Memory organised logically as a huge, sequence of buckets
* Data of various types occupies various numbers of these buckets
* Each bucket is individual, it is not aware of it’s surroundings – it doesn’t know whether the data in the next bucket is related in any way
* So we must mange not only our data, but also how it is organised in the buckets, and how these buckets relate to one another

**Organising data and memory**

* How to organise buckets
* Simplest way is to group together similar buckets. Decide how many buckets we need for our data. Allocate those buckets as a group and keep track of the first bucket.
* If we know which is our first bucket and how may we have we should be ok
* Major Drawback: Need to group buckets together – in the real world this means allocating a contiguous series of buckets (memory locations)

**Non-Contiguous Memory**

* The only with current hardware architecture is to allocate whatever memory we have non-contiguously, i.e wherever it can be located, then to keep track of that.
* Many data structures are merely patterns for organising and tracking memory allocations
* Get’s around the major drawback of arrays at the expense of complexity.

**Types Of Data Structure**

* Aggregates: Structs, Unions
* Linear (Sequential): Arrays, Linked Lists, Stacks, Queues, Deques
* Linear (Associative): Dictionaries
* Non-Linear: (Binary) Trees, Graphs

**Aggregates**

* A way to cluster arbitrary collections of data together so that they can be treated as a related group.
* Is easier to handle a group of things, that might be of different datatypes, but that need to be treated as logically similar
* Contents of a struct highly dependent upon speciﬁc programming problem, e.g.
  + Modelling a person, might group personal information about them such as Name, Date of Birth, Favourite Star Trek Movie, &c.
* In other languages, these are data that we might group together in a class

**Sequential Structures**

* A way to organise primitive datatypes in relation to each other – no branches
* Arrays are our core sequential data structure
  + Can use arrays to implement other kinds of data structures using contiguous memory or can replace array with a non-contiguous structure
* Leads to two forms:
  + Contiguous: Each element of the sequence is next to a neighbour until you get to the end
  + Non-Contiguous: Each element of the sequence stores both it own data & metadata about the location of the next & possibly also the previous element in the sequence

**Sequential APIS**

* If we have a sequence of elements then its shapes suggests natural ways to interact with it,
  + e.g. from each end (assume one end is the start/head & the other is the end/tail)
  + If we start at one end, than can move from one element to the next until we reach the other end
* By restricting how we interact with the sequence we get the behaviour of various data structures, e.g. stack, queue, deque (logical variations on the theme of the sequential API)

**Associative Structures**

* Hash tables are an example of an associative structure
* Might be aware of them under different names, e.g. map, dictionary, associative array
* A collection of pairs comprising a key and a value
* Given the key, can retrieve the value

**Non-Linear Structures**

* Instead of pointing to next & previous, lets:
  + Point to children – leads to a form of a data tree structure
  + Arbitrarily point to other elements (possibly including itself) – leads to a graph (network) data structure
    - NB. Various forms of trees and graphs depending on restrictions on the links
    - A tree is a sub-type of graph in which there are no-cycles

**Stacks**

* Think of a stack of trays in a canteen - this is an almost archetypal physical real-world implementation of a stack.
* Stacks are collections that are based on sequential structures so member elements are organised as a sequence
* They have a well defined interface: The stack has a top & a bottom. Items can be pushed on to the top of the stack and popped off of the top also.
* Often have non-essential operations implemented, e.g. “peek” - observe but don’t remove
* Items within the stack are not accessible within the definition of the stack API until they are at the top - then they can be popped (NB. pragmatism)
* last item in is the first item out (LIFO)
* Stacks don’t specify how the underlying collection should be implemented:
  + An array is an easy approach - limited size, need reorganising
  + Linked Lists are an alternative - increased flexibility & complexity

**Queues**

* Just like stacks, based on a sequential collection of data but the interface, how we can use that data is different.
  + Queue has a front and a back
  + Elements are enqueued (added to the back) and dequeued (removed or “serviced” from the front)
* Leads to a first in first out (FIFO) data structure - elements are dealt with in the order in which they arrive

**Linked Lists**

* An alternative to the dynamic array that overcomes some of the limitations of the basic array
  + but introduces further complications
* A list created by linking discrete items of data together so that they are spread across memory
  + Start with one item, the head of the list
  + Each item “points” to the next item
  + Move through the sequence by accessing the current item & following the link to the next item
* Two types doubly and singly linked lists:
  + Singly Linked List - Move through the sequence by accessing the current item & following the link to the next item
  + Doubly linked list - Each element stores it’s own data and a pointer to both the next element and the previous element. Can traverse the list in either direction.

**Linked List Characteristics**

* Constant time insertion and deletion - no reallocation or reorganisation needed - because we’ve removed the contiguous constraint on memory layout
* Dynamic can grow or shrink as necessary
* Use more memory than arrays
* No random access to or efficient indexing of data in the list

**Accessing Data**

* Until now, once data has been added to a data structure we have had two basic ways to retrieve it:
  + 1.By index - when a structure supports direct indexing
  + 2.By value - search through the structure until we find our target value. then retrieve that value & any associated data (assume a struct)

**Keys & Values**

* Think of keys as really flexible array indices…
* instead of just a number it can be useful to use real data as an index and associate that index with a further set of data (the values associated with the key)
* Assume the key is unique and maps to one set of data

**Key:Value Stores**

* Key:value stores (databases) are very popular ways to store data as an alternative to relational, graph, document, or column stores
* Examples include: Redis, Riak, Project Voldemort, Berkeley DB, Memcached, Dynamo, &c

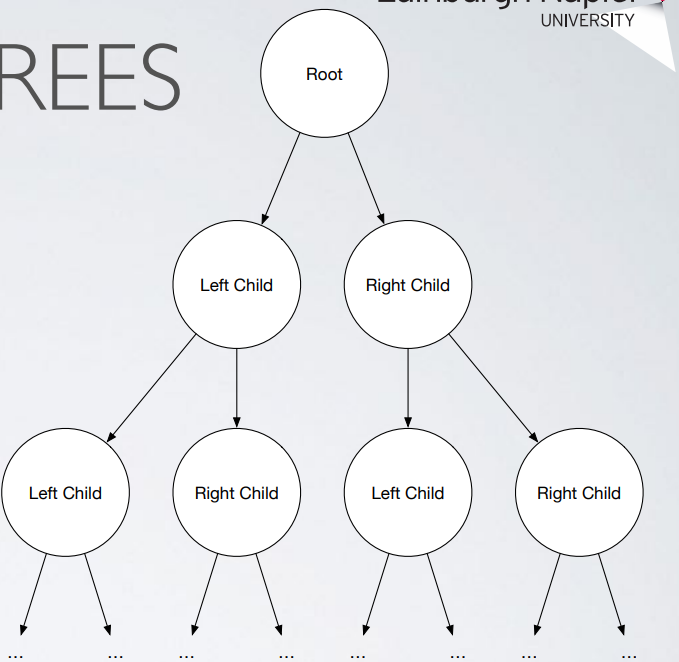
**Key:Value Data Structures**

* Not just databases
* Basic data structures provide key:value structure as well
  + e.g. HashTable (map, dictionary, symbol table, associative array)
  + Appears in many languages (Python, Java, JavaScript, &c.)
* The collection comprises a set of key:value pairs such that each possible key appears at most once in the collection

**Associative Arrays**

* Typically support the following API:
  + Add a key:value pair to the collection
  + Remove a key:value pair from the collection
  + Modify an existing key:value pair within the collection
  + Lookup the value associated with a particular key
* • N.B. The value doesn’t have to be a basic data type, can be a more complex structure.

**Trees**

* Hierarchical
* Start with root node
* Each node can have children
* e.g. up to two children (left child & right child) gives a binary tree
* Various types of tree available allowing various numbers of children, being constructed in various ways, being balanced, ordered

**Graphs**

* Another non-linear data structure used a lot on mathematics
* Also non-hierarchical
  + Whilst tree has a root node, there is no root node in a graph
  + Any node in a graph can point to any other node
  + Formally:
* A set of nodes (vertices) & a set of edges
* The edges connect the nodes
* Nodes & Edges can also store information, e.g. label/weight/ length/distance on edges

**Directed and undirected**

* Edges can have a direction, e.g. pointing from one node to another – directed
* or else they don’t - undirected graphs have edges that merely link nodes together

Lecture 4: Algorithms and Searching

**Classifying algorithms**

* Can be classified on the basis of various characteristics
* These can help folk to see how various algorithms group together or are differentiated
* Classification by:
  + purpose | implementation | design paradigm | probabilistic/heuristic paradigm
* Algorithms can be classified on the basis of how they are internally organised
  + on the main implementational process that is used to get from the start state to the termination state

**Implementation**

* Recursive or Iterative - Algorithm repeatedly calls itself until a certain condition is matched. Common in functional programming (e.g. Haskell, Erlang)
* Logical or procedural - A logical algorithm has problem expressed in terms of axioms, to which rules are applied to deduce a solution (e.g. Prolog)
* Serial or parallel - Serial algorithms execute one step at a time whereas a parallel algorithm allows multiple steps to occur together (which can take advantage of multiple CPUs, Cores, Threads, etc.)
* Deterministic or non-deterministic - Deterministic algorithms solve problem using a predefined process at each step whereas non-deterministic perform a best guess (guided by heuristics)

**Design Approaches**

* Divide & Conquer - Repeatedly reduce problem into smaller independent instances of same problem (usually recursive) until small enough to solve easily, e.g. binary search algorithm
* Dynamic Programming - If optimal solution can be constructed from overlapping optimal solutions to sub-problems then can avoid recomputing solutions (related to D&C above). Memoization is a useful related technique (storing results of expensive calculations), e.g.
* Greedy Methods - Similar to dynamic programming but solutions to sub-problems don’t need to be known at each stage. Instead make a greedy choice of what looks the best solution at the moment, e.g. Kruskal’s (graph) algorithm
* Linear Programming - Express problem as a set of linear inequalities then attempt to maximise or minimise the inputs
* Reduction (transform & conquer) - Solve problem by transforming into another problem, e.g. find median in an unsorted list; transform into sorted list then take middle value; Goal: to find simplest transformation possible
* Graphs - Model problem as a graph then apply a graph exploration algorithm
* Machine Learning - New chart entry this year ;) - Increasingly used to tackle problems. Usually data intensive + lots of different sub-approaches

**Probabilistic & Heuristic Approaches**

* Probabilistic Search - Build probabilistic model of candidate solutions for a given search space
* Genetic - Find solutions by taking inspiration from biological or evolutionary processes, e.g. define generation, randomly mutate, test against benchmark, allow percentage of best performers to survive to next generation, repeat
* Heuristic - Find an acceptable approximate solution rather than an optimal solution, e.g. if time or resource mean optimal is not practical

**Searching**

* Any algorithm to retrieve information stored in a data structure
  + e.g. Finding data in collections like arrays, lists, &c.
* Searching for & retrieving data becomes an acute problem at scale
  + e.g. Search engines return results in a split second (user expectation) but have to get those results from perhaps billions of pages

**Search Keys**

* An algorithm is used to find items that have specific properties within a collection of items
* Items might be
  + basic data-types or objects
  + database records or structs
  + elements of a search space
* The datum that defines the search target is known as the search key

**Search strategies**

* Search algorithms implement search strategies
* Basic differentiation based upon the data being searched:
  + Is the data organised in any way - if it is then this influences how we can approach finding a particular element within the collection
  + What else do we know about the data? - sometimes the nature of the data and it’s sorting can give us a little boost in our search strategies

**Sorted Vs Unsorted**

* Strategies that work on unsorted data are simple and robust but can have very poor performance characteristics
* Most efficient search strategies rely on sorted data

**Linear Search**

* Simplest Search algorithm
* List need not be ordered
* Essentially a brute force or exhaustive search
* Trade-off between set-up/sort time & search time for other algorithms versus cost of searching with linear search

**Binary Search**

* Find the position of a target value within a sorted array
* Compare target to value of middle element of array
  + If target is equal then search is finished
  + If target < element then repeat in lower half of array
  + If target > element then repeat in upper half of array
* Each iteration will half the search space

**Jump Search**

* Algorithm for sorted arrays
* Check fewer elements by skipping a fixed number of elements instead of searching all elements
* Because array is sorted our comparison tells us if we’ve jumped too far
* Can then perform linear search of the interval
* Time complexity is O(√n) which is between linear search, O(n), and Binary Search, O(Log n)
* Advantage of jump search is that it only needs to jump back once so if the jumping back operation is costly then jump search can become more efficient than binary search (limited circumstances)

**Interpolation search**

* Sorted arrays
* Uses a simple formula to calculate a probe position on each iteration instead of just using the middle element. Formula relies on the distribution assumption above.

**Exponential search**

* Misleading name (actually runs in O(Log n) time)
* Useful when you need to search unbounded data, e.g. size of array is infinite
* Can work better than Binary search in a bounded array when the target is closer to the first element

Lecture 5 Searching & Sorting Algorithms

**Comparison Relations**

* Various comparison relations for data can make sense
* But numeric and lexicographical order are obvious choices for many sequences, strings, tuples, etc.

**Stable Sorting**

* Sorting algorithms can be distinguished on the basis of whether the sorts that they generate are stable
  + That is, whether repeated elements in the input appear in the same order in the output

**Bogo Sort**

* Generate all permutations of your data until you get the right (sorted) one
* Pretty bad performance: O((n+1)!) in average & worst cases
* Relies on the idea that there is some probability of getting the right permutation at each try

**Bubble Sort**

* Type of comparison sort
* Simple (but inefficient)
* On each pass, sorts largest value to end of collection
* However: can efficiently check if data is sorted:

Lecture 6 Hash Table & Hashing

**Hash Tables**

* A hash table consists of an array in which data is accessed by a special index called a key.
* Establish a mapping between the set of all possible keys and indices in the array.
  + a hash function, h(k).
  + h(name) -> index of phone number in array.
* Use this to perform constant time searches!

**Good hashing**

* In most applications, number of positions in hash table much less than universe of possible keys
* unfortunately, some keys will map to the same index = a collision
* We want to minimise collisions (uniform hashing)
* But collusions will still occur

**The Hard work With Hash tables**

* Mapping keys evenly around the table
  + approximating uniform hashing
* Dealing with collisions
  + chained hash tables
  + open-addressed hash tables

**Chained hash tables**

* Chained hash tables are arrays of lined lists
* When a Collison occurs, simply place the new item at the head of the corresponding linked lists.

**Dealing with collisions**

* Easy, but if an excessive number of collisons occurs at a specific position, the bucket becomes longer and longer
* Worst case we go from O(1) to O(n) for retrieving an item
* Ideally, we want all buckets to grow at the same rate

**Keeping an eye on the load factor**

* Load Factor = number of elements in table/number of positions into which elements may be hashed.
* In chained hash tables, this gives the maximum number of elements we can expect to encounter in a bucket assuming uniform hashing.
  + e.g. with 1699 buckets and 3198 elements inserted, the load factor is....
  + 3198 / 1699 = 2
* But remember, in reality uniform hashing is only approximated...

**Hash functions**

* A good hash function should approximate uniform hashing.
* h(k) = x, where x is the hash value of k
* Most methods assume k to be an integer (and x must be!)
* f k isn’t an integer, we can coerce it to be one…

**Division method hash functions**

* Once we have our key k as an integer, an easy way to map it into one of m positions in the table is:
* h(k) = k mod m
* But, we need to avoid values of m that are powers of 2.
  + Because if m = 2p, then h(k) becomes just the p lowest-order bits of k.
* Choosing our table size, m
  + Typically we set m to be a prime number not too close to a power of 2.
  + e.g. if we expect to insert around 4500 elements into a chained hash table, we might choose m=1699
    - why?
    - it’s a good prime between 2^10 and 2^11.
    - it results in a load factor of 4500/ 1699 (approx. 2.6)

**Open-Addressed Hash Tables**

* All elements reside directly in the table itself (no linked lists).
* We need another way of resolving collisions,
* We resolve collisions by **probing** the table until we find an empty slot.
  + Go to index h(k), then probe until we find a free slot (insertion) or we find the item (retrieval or deletion)

**Performance**

* Goal is to minimise the number of probes we have to do.
* This depends on
  + The load factors
  + How uniform h(k) is
* The load factor of an open-addressed hash table cannot be > 1.

**Linear probing**

* The simplest sort of probing you can think of
* When a collision occurs store the item in the next empty slot in the table.
* More formally: h(k,i) = (h’(k) + i) mod m
  + where h’(k) is our plain old hash function, i is the number of times the table has been probed, and m is the size of the table.

(Examples in the lecture slides)

**Disadvantage of Linear Probing**

* Suffers from primary clustering
  + Large chains of occupied positions develop as the table becomes more and more full.

**Quadratic probing**

* Reduce linear clustering by going up in increments of i ^2 instead of increments of i.
  + probe positions 1 long, then 4 along, then 9 along…
* Performs better than linear probing as clustering is less severe

**Universal hashing**

* Generates hashing functions randomly at run time.
  + So that no particular set of keys is likely to produce a bad distribution of elements in the hash table.
* Even hashing the same set of keys during different executions may lead to different numbers of collisions.

(Review questions also in lecture slides)

**Cryptographic hash functions**

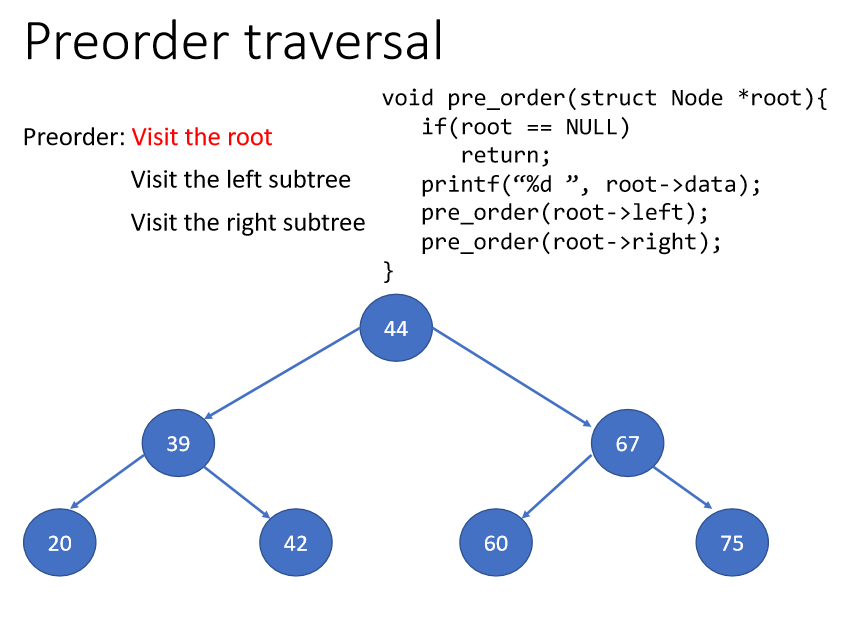
* In general, hash functions map data of arbitrary size down to data of a fixed size.
* In security applications, we want to map data to a bit string (hash value) of a fixed size
* And make it a one-way function so its very difficult to go back from the hash value to the original data

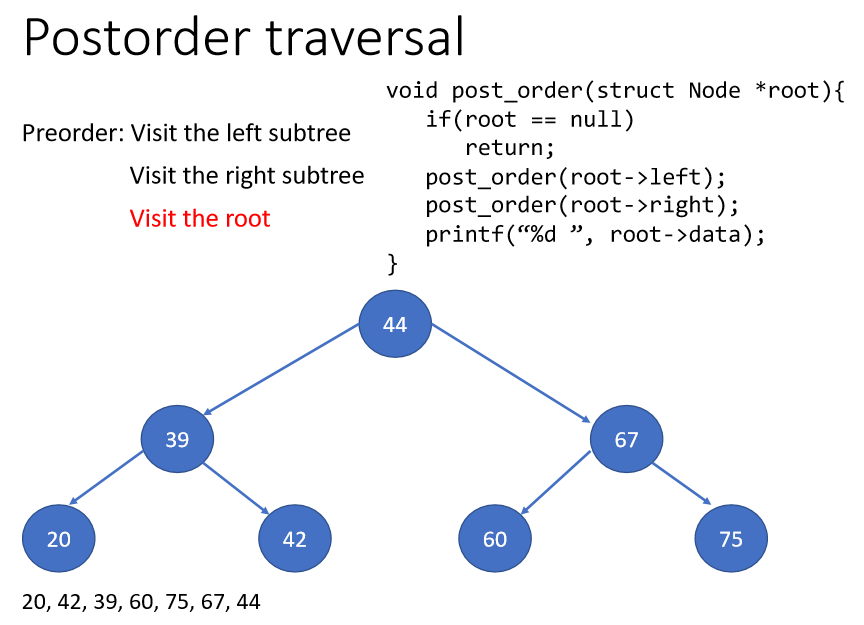
Lecture 8 Binary tree algorithms

**Trees**

* A tree consists of nodes
* Node at the top of the hierarchy is the root
* Nodes below the root are its children
* Each node has one parent
* But a node can have many children – the maximum number is the tree’s branching factor

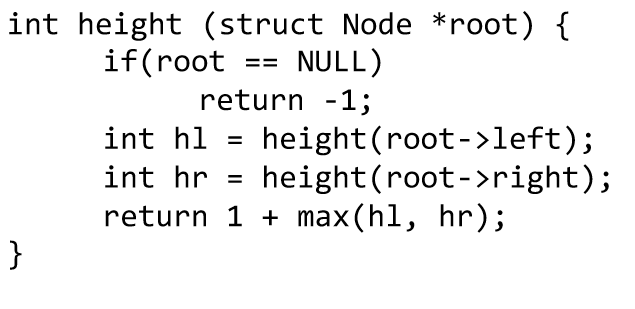
**Traversing trees recursively**





**The height of a tree**

* For a tree with just one node, the height is 0.
* The height of a null tree is -1.
* The height of a tree with two levels is 1.

Recursive algorithm to calculate the height of a tree.

**Expression trees**

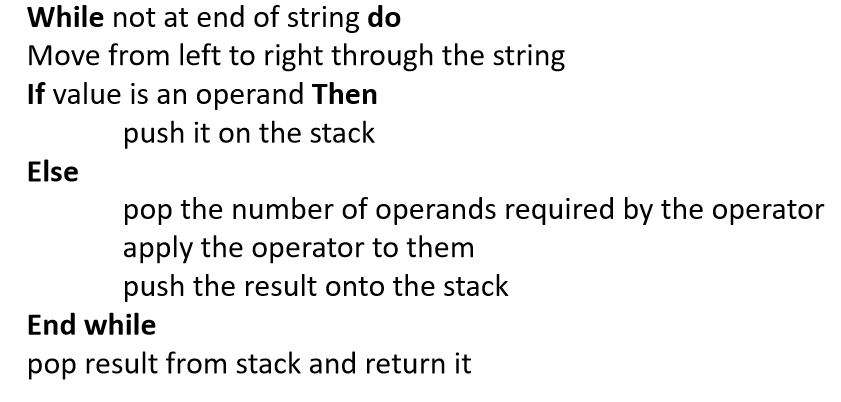
* Consider A \* (B + C) /D.
* This is written in infix notation. Problem: This needs rules about precedence in the language, and brackets to override these.
* Prefix (polish) notation : /\* A + BCD
  + (/ (\* A(+BC)) D). Operators written before their operands, evaluated left to right. No brackets needed.
* Postfix(reverse polish) notation : A B C + \*D/
  + ((A(BC+) \*)D/). Operators written after their operands, evaluated left to right. No brackets needed.

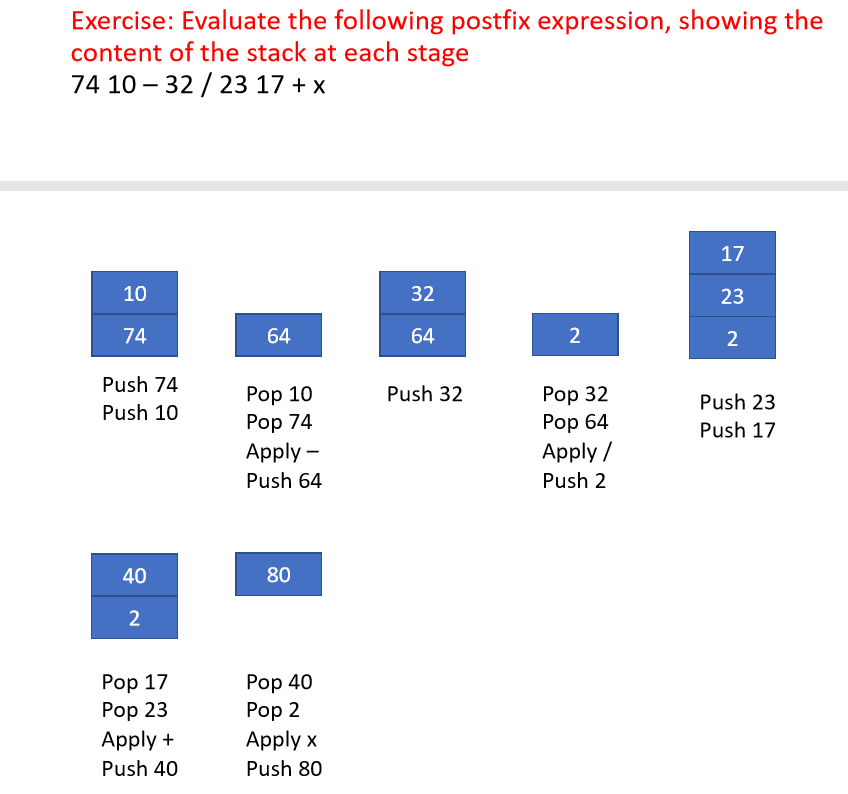
**Converting between the notations**

* Insert all the implicit brackets showing the order of evaluation:
* (A \* (B + C)) /D)
* To get a prefix expression, simply move each operator to beside its left bracket:
  + (/(\*A(+BC))D)
* To get a postfix expression, simply move each operator to beside its right bracket:
  + (A(BC +) \*)D /)

**Evaluating a postfix expression**

* To evaluate a postfix expression, we apply each operator to the operands immediately preceding it.
* They are easy to evaluate using an abstract stack machine

**Algorithm to evaluate a postfix expression**



**Stack machines**

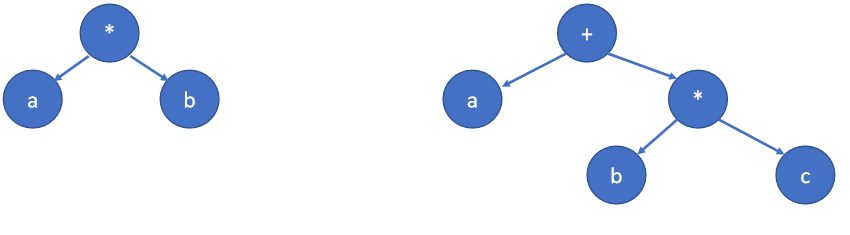
* A type of (virtual) computer where most instructions operate on a stack rather than registers.
* Instructions take their operands from the top most values on the stack.
* Very fast to access operands
* Example : The java virtual machine.

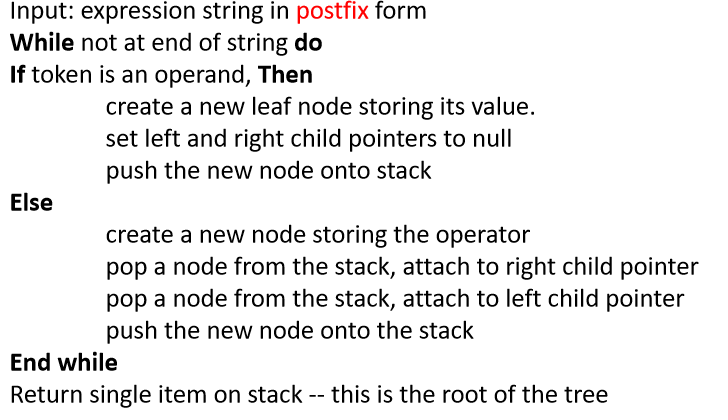
**Evaluating a prefix expression**

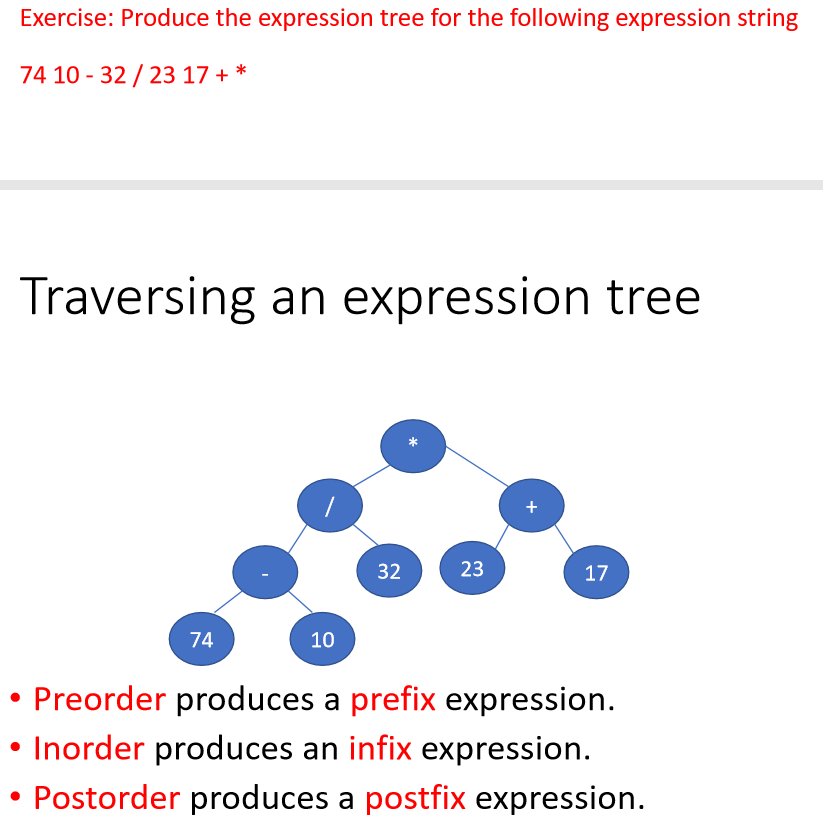
* Easier to evaluate the string from right to left.
* You can then push operators onto the stack as before, pop them when you encounter an operator, apply the operator to them, and push the result back onto the stack.

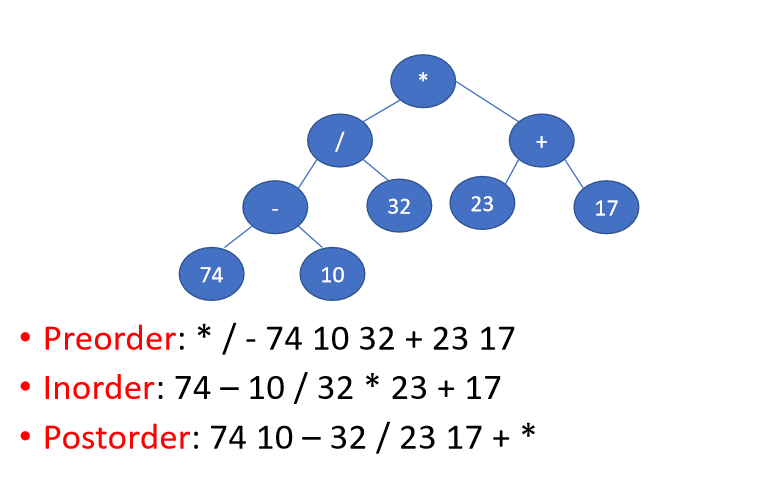
**Expression Trees**

* A compiler uses a binary tree to represent an arithmetic expression.
* Expression trees contain operators and terminal values.

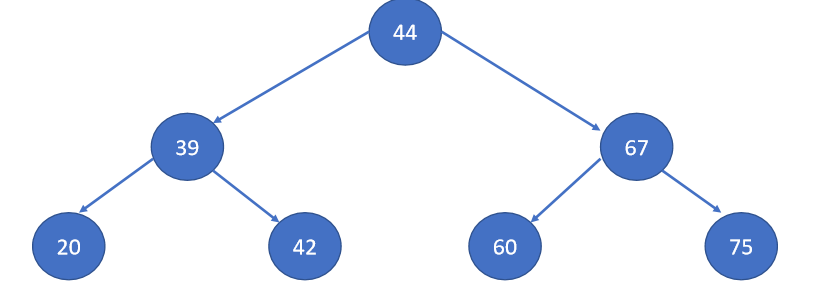


**Building an expression tree**





**Binary search trees**

* Organise data for efficient searching
* Everything to the left of the root is smaller than it
* Everything to the right of the root is greater than it
* Apply this rule recursively

**Efficiency**

* In the worst case we only end up searching one branch.
* If a tree is balanced, will be O(Log2n)
* If unbalanced, can drop to O(n)