# Data Structures and Algorithms

## Week 1- General

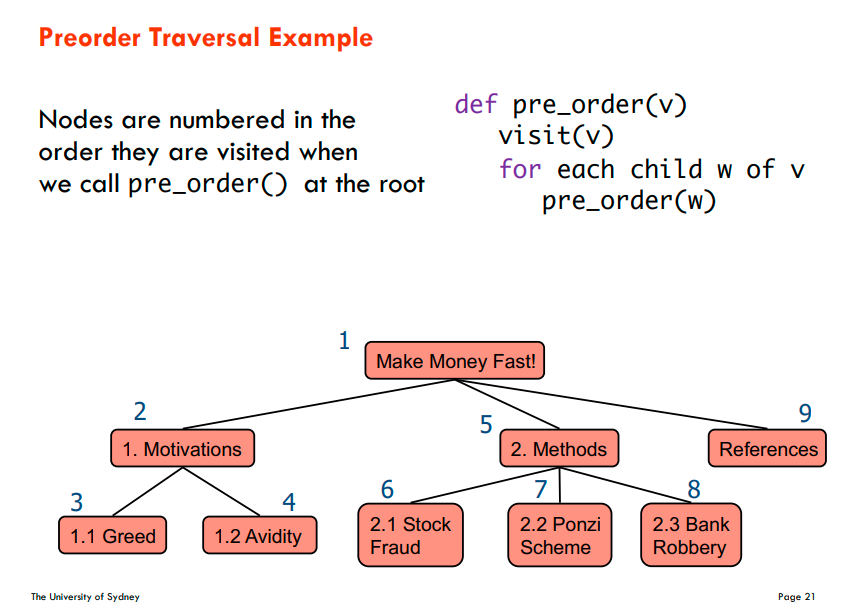
* We will have a problem set: a computational problem. We will design an algorithm. And then we will have to analyse its complexity and correctness, justifying its efficiency and resolution to the computational problem.
* An algorithm is **constant time** if you can say exactly how many iterations it will take, while it is not constant size if the number of iterations depends on the input.
* An algorithm is **efficient if it runs in polynomial time**; that is, its running time is **O(n^k)** for k > 0 rather than **O(a^n) for a > 0 (exponential).** Another example is **logarithmic (O(logn)).**
* If **T(n)** is the exact number of effort that an algorithm does, then **O(n) is the ‘asymptotic growth analysis’** which gives us a less exact but more digestible analysis by focussing on that which dominates the running time.
* Definitions:
* We say that T(n) = O(f(n)) if there exist n0, c > 0 such that T(n) ≤ cf(n) for all n > n0. T(n) = 32n 2 + 17n + 32 T(n) is O(n 2 ) and O(n 3 ), but not O(n). (upper-bound: worst case)
* We say that T(n) = Ω(f(n)) if there exist n0, c > 0 such that T(n) ≥ cf(n) for all n > n0. T(n) = 32n 2 + 17n + 32 T(n) is Ω(n 2 ) and Ω(n), but not Ω(n 3 ). (lower-bound: best case)
* We say that T(n) = Θ(f(n)) if T(n) = O(f(n)) and T(n) = Ω(f(n)). (bound: lower-bound=upper-bound case)

## Week 2- Lists

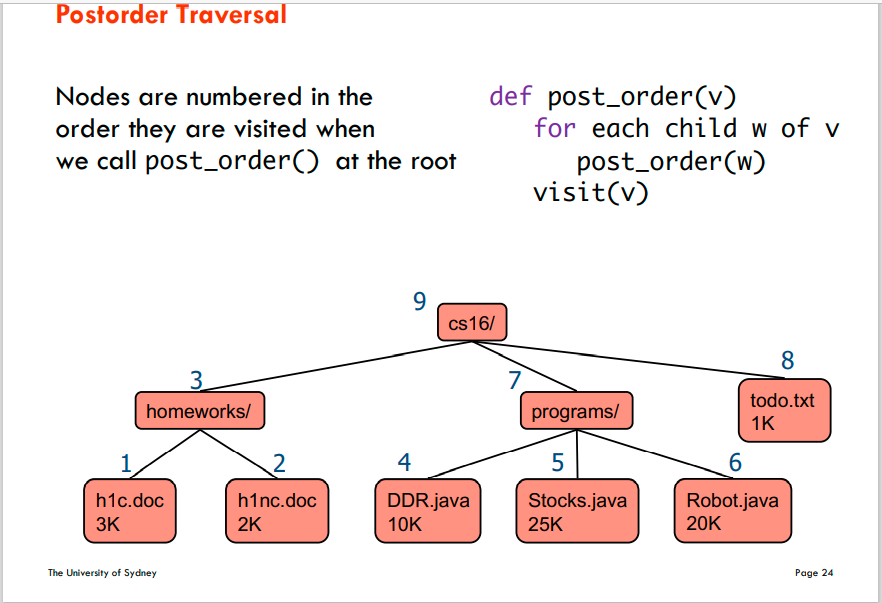
* **ADT:** Abstract Data Type, a data type that is defined by its functionality and what functions it needs to support, rather than its data type in the program/language.
* **Difference between n and N:** n is the size of the list/array, N is the size of the memory allocated to the list/array.
* We can have **array-based** lists (A.get(i) where I is the index) or **positional lists** (e.g single or doubly linked lists where it’s just A.get(element))
* Positional list ADT: changing the list changes are based on an index/position/container given. E.g insert\_before.
* **Stack:** last-in-first-out (LIFO). When push(k) is called the element is placed at the end of the ‘array’ and is the first to be removed when pop() is called. Space is O(N) and operations are O(1).
* **Queue:** first-in-first-out (FIFO). When enqueue(e) is called the element is placed at the end of the ‘array’ and dequeue() takes away the element at A[0]. Space is O(N) and operations are O(1). Be aware of when array must double when n = N.
* Arrays are generally faster if you know that you need to access elements by index a lot or are iterating over the list a lot, otherwise the linked lists are much better since their memory allocation is more flexible (not contiguous) i.e. efficient deletion, insertion, lower space used as no maximum to space.

## Week 3- Trees

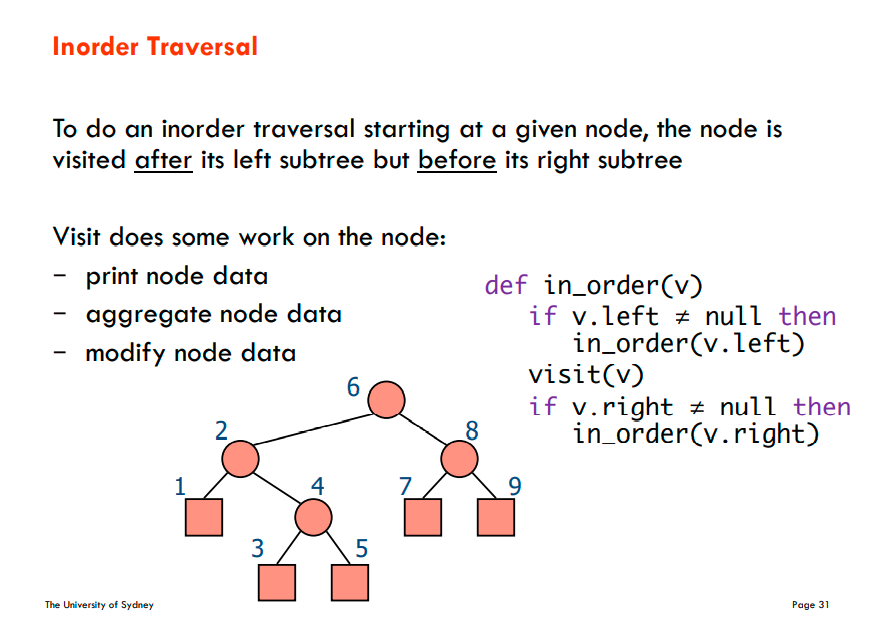
* In a tree, a ‘**root**’ is a node without a **parent**, an ‘**internal node’** is a node with at least one child, and an ‘**external node/leaf’** is a node with no children. We can also define ‘**ancestors**’, ‘**descendants**’, and ‘**siblings**’.
* The ‘**depth(x)**’ is the number of ancestors of x. The **‘level’** refers to the set of roots with the same number of ancestors. The **‘height’** is the maximum depth of the tree.
* A ‘**subtree**’ is a tree made up of some node and its descendants. An ‘**edge**’ is a pair of nodes where one node is a parent of the other. A ‘**path**’ is a sequence of nodes such that 2 consecutive nodes in the tree have an edge.
* A ‘**binary tree’** is a tree where each internal node has at most two children. It is ‘**proper’** if each node has either no or two children.
* Pre-order traversal:



* Post-order traversal:

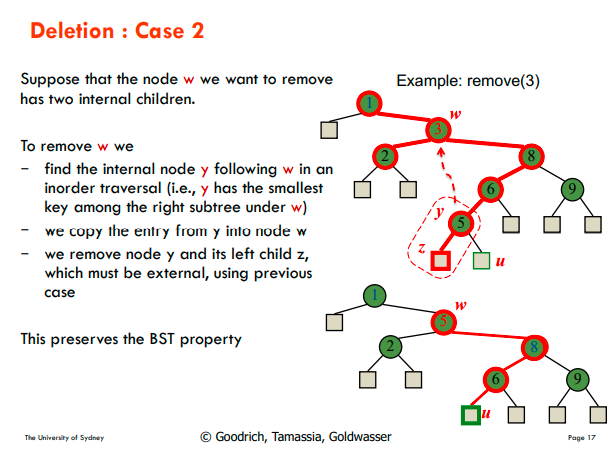


* In-order traversal:



## Week 4- Binary Search Trees

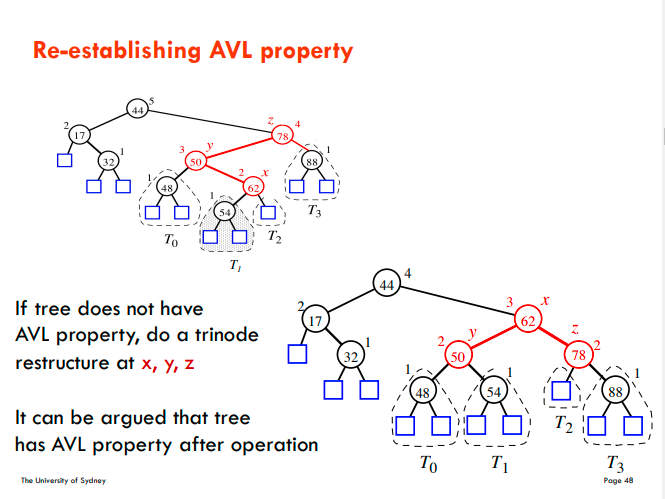
* A **Binary search tree** is laid out so that all left children are lower than their parent and all right children are higher than their parent. They are also balanced so that their height is logn by trying to give each node at least two children.
* **All external nodes are null** and our actual data is in internal notes. This is because we can have an easy check where, if we’ve reached a null node, our search has failed and we can return that failure.
* Remember that nodes in a tree are key-value pairs, so the **put(k, o)** operation first searches for the key in O(logn) time then either inserts it or replaces its value. If it’s inserted, it will replace an external node and it will now need to have its children be assigned to null.
* With **remove(k),** either the key is not there, it has one internal child, or it has two internal children. The first case is simple. The second case simply requires us to assign that internal child to k’s original position. The third case is as such:



* A range query operates by making three different decisions at each node:

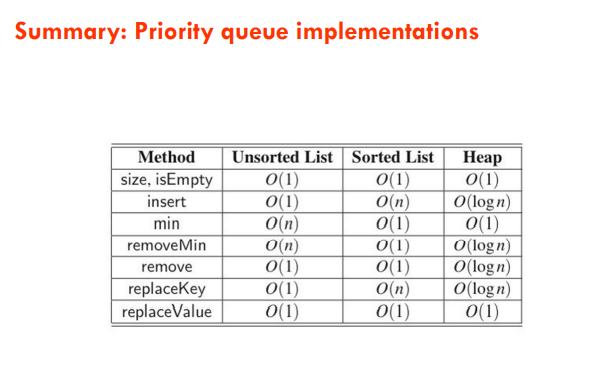
1. If the node is within the given range, then recurse on both of its children since both of them can be in the range.
2. If the node is below the range, then only recurse on its right children.
3. If the node is above the range, the only recurse on its left children.

* Therefore, since we only spend O(1) time per node we visit, the total running time of range search is O(|output| + tree height).
* **AVL trees** have a **height of logn** by ensuring that the depth of the two children of every internal node **differ by at most 1**.
* Maintain the height at each node by, whenever you insert or remove a node, recompute the height of it and its children which takes O(logn).



## Week 5- Priority Queues

* A **priority queue** is a special type of map/dictionary where we can only remove the **smallest or highest** key-value pair.
* We can have an **unsorted implementation** where we **insert in O(1)** time but find the min and use the **remove\_min() method in O(n).** A sorted implementation is the opposite where we sort as we insert so that takes O(n) time but remove\_min() just takes O(1).
* **Selection-sort:** a priority queue algorithm where you sort an array by choosing the minimum of the entire unordered array each time and placing it at the front of the array. Takes O(n^2) time whether in unsorted implementation or sorted implementation.
* **Insertion-sort:** a priority queue algorithm where you sort an array by choosing the minimum as you iterate through the array and moving that to the front. Takes O(n^2) time whether in unsorted implementation or sorted implementation.
* A **heap** is a binary tree in which each child has a **key higher than its parents’**. A **complete binary tree** has every level except the last being full (2^I nodes) and all the nodes at depth(h) take the leftmost positions. ***The root is the minimum****.*
* **Insertion**s can maintain a heap by inserting at the next available position in the bottom level, and then **upheaping**. The node is swapped with its ancestor until its ancestor is less than it. O(logn).
* We **find the position for insertion** by starting from the last inserted node and going up until either we reach a left child with no right sibling or we reach the root, in which case we go down left until we reach a lead (page 255).
* When we **remove the minimum (the root)**, we replace it with the last node and reassigning our pointer to our previous last node. Then we **downheap** by swapping the node along its smallest child until our node has no children that are lower than it.
* We have to find the previous last node by going up until we reach a right child with no left sibling or the root. If we reach the root then we need to go down right until we reach a leaf (page 260)
* Heaps are like priority queues in that you can only remove by taking out the minimum/maximum, and rather than having insert = O(1) and remove\_min() = O(n) or vice versa, both operations are just O(logn).
* Hence heap-sort takes O(nlogn) since it’s n remove\_min operations.
* Heap-in-array implementation: page 263.



## Week 6- Hash Tables

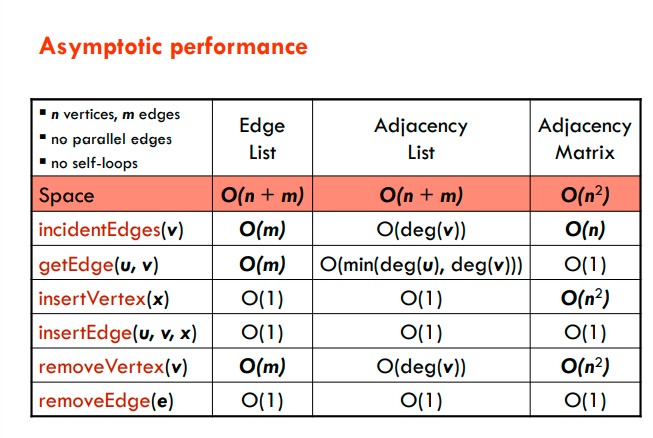
* Unsorted maps have O(1) time for put if the key doesn’t exist yet, but in the worst case **put, get and remove require O(n)** since we have to search the entire ADT for the element.
* If we place elements in an array where the index is the value and the value is the key (i.e. D: 4 is at A[4] = D, then we can make all these operations O(1), but of course this means that we have an array where N >> n. O(N) space complexity. Also does not work for non-int values.
* Enter, **Hash tables:** values are placed in an array according to an index calculated by running the value through a **hash function e.g h(x) = x mod N.** Range of h() is 0: N-1.
* **Horner’s algorithm:** h(k) = x1 a^d-1 + x2^ad-2 + … + xd-1 a + xd where you multiply a digit/character of a key by some value a ^ k.length. e.g h(“cat”) = c \* 2^2 + a \*2^1 + t\*2^0.
* **A collection of hash functions can also be used to assign hash table index.** This collection is said to be **2-universal** if a hash function chosen **uniformly at random (UAR)** if Pr[h(i) = h(j)] <= 1/N.
* **The load factor** is the parameter n/N, usually written as *a*.
* To deal with collisions, we either use:

1. **Separate chaining**: each cell points to a linked list holding all the entries including duplicates. Can have O(n) time if all items collide in a single chain (pg 295).
2. **Linear probing:** inserts by skipping over full spots to place in the next available spot. Worst case is O(n) time as can all try to place in the same spot. Fact: Assuming hash values are uniformly randomly distributed, expected number of probes for each get and put is 1/(1-a) where a = n/N is the load factor of the hash table. Thus, if the load factor is a constant < 1 then the **expected running time for the get and put operations is O(1**)
3. **Cuckoo hashing:** Although in practice this turns out to be a little bit slower, it’s still preferred since it’s **worst-case is always O(1).** This is made up of two arrays where an element has two options to be placed in. If both its positions are full, this starts an eviction cycle until an alternate element has another position in the hash tables. Only need to check two positions each time so O(1).

## Week 7: Graphs

* **G = (V, E)** where V is a set of nodes called vertices and E is a collection of pairs of vertices, called edges.
* **Parallel edges** share same endpoints e.g., h and i are parallel
* **Self-loop** have only one endpoint e.g., j is a self-loop
* Simple graphs have no parallel or self-loops
* A **simple path** is one where all vertices are **distinct**
* A **tree’s** vertices are all connected. A **forest** is made up of disconnected trees.
* **A spanning tree is** a connected subgraph on the same vertex set
* Graphs can be recorded in an:

1. **Edge list**: a list of pairs of vertices which represent edges.
2. **Adjacency list:** vertices keep track of which vertices have edges into them.
3. **Adjacency matrix:** Each index is assigned to a vertex, and a value in a 2d nxn array means that there is an edge between the corresponding indices.



* **Depth-first search (DFS):** We start from any vertex and recurse over its children. We keep recursing along children until we can only go back to already visited vertices, in which case we go back up the stack and onto the next children (pg 343) O(n + m).
* **Breadth-first search (BFS):** visits each node by visiting the graph by levels, where each next recursion is on all the children of the next node (page 354).
* Identifying cut edges O(n + m) by doing a DFS and at each step finding the vertex’s highest node and then one edge back up.

## Week 8- Shortest Paths and MST

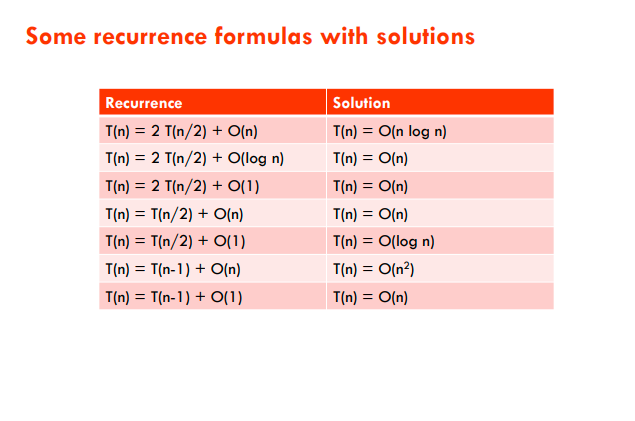
* **Dijkstra’s algorithm** finds the **shortest path** between two nodes by finding the shortest path from the starting node to every other node. First initialise the shortest distance to each points as infinity, and then recurse to each incident vertex and update their shortest paths. Keep visiting the nodes, adding the parent’s shortest distance with the edge’s weight and assigning it to the vertex if it’s the minimum. (page 372) O(m + nlogn) with heap and O(m + nlogn) with Fibonacci heap.
* **A minimum spanning tree (MST)** is the spanning tree with **lowest cumulative edge weights.**
* Cut property: Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.
* Cycle property: Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.
* **Prim’s Algorithm:** go through the graph and choose the shortest edge that maintains the above properties each time (pg 405). O(m + nlogn) with heap and O(m + nlogn) with Fibonacci heap.
* **Kruskal’s algorithm:** Add edges in increasing order of weight so long as they don’t form a cycle (pg 424). O(mn)
* To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to **break any ties**. E.g if we add i/n2 to each edge e.

## Week 9- Greedy algorithms

* A class of algorithms where we build a solution one step at a time making locally optimal choices at each stage in the hope of finding a global optimum solution.
* The **fractional knapsack problem** involves a set S of n items where each item has a benefit and a weight. Each item needs to be able to be divided and is positive. Benefit/weight O(nlogn) just to sort the items.
* **Task scheduling:** the minimum number of ‘lecture halls’ to assign the lectures is the depth. Sort in increasing start time order than make greedy choice for each lecture.
* **Text compression uses Huffman encoding** by placing the most frequent characters earlier up in a tree and the less frequently used lower in the tree. Their positions are encoded as 0s and 1s where a 0 is a left on the tree and a 1 is a right. (Page 463)

## Week 10- Divide and Conquer

* Usually recursive solutions.
* 1. Divide If it is a base case, solve directly, otherwise break up the problem into several parts.
* 2. Recur/Delegate Recursively solve each part [each sub-problem].
* 3. Conquer Combine the solutions of each part into the overall solution.
* Example: **binary search algorithm** finds an element in a sorted array by checking the middle of the array for equality, and then recursing on the left or right half depending on whether the element is higher or lower. O(logn)
* **Unrolling** is achieved by … Make sure the k is the number that we divide the array by each time.
* **Merge-sort** requires you to half the arrays in half until you can sort just one or two elements at a time, then you can choose pointers across two arrays and choose the lowest elements each time to sort into the new array. O(nlogn).



* **Quick-sort** O(nlogn) (page 524)

## Week 11- Divide and Conquer 2

* **Maxima-set:** the set of points in an x-y plane for which every other point is either smaller in x-value or smaller in y-value.
* **To solve maxima-**set, sort points by x-coord and break ties by using y-coord. Recur and find the Maxima set of each half, then check the maxima-set of the left half by comparing each of those points against the highest point in the right half (pg 543) O(nlogn).
* The **integer multiplication** problem is a way of showing unrolling as well as a way to multiply large numbers in less than O(n^2) time. It works out to O(n^logn3) because T(n) = 3T(n/2) + O(n). (pg 555).
* We can find the **kth smallest integer in an unsorted array** by dividing the array in 5 parts each time and approximating the median of each array, which solves to O(n) rather than the O(nlogn) we would’ve gotten with simple sorting. (page 571)

