COMP9123 Data structures and Algorithms

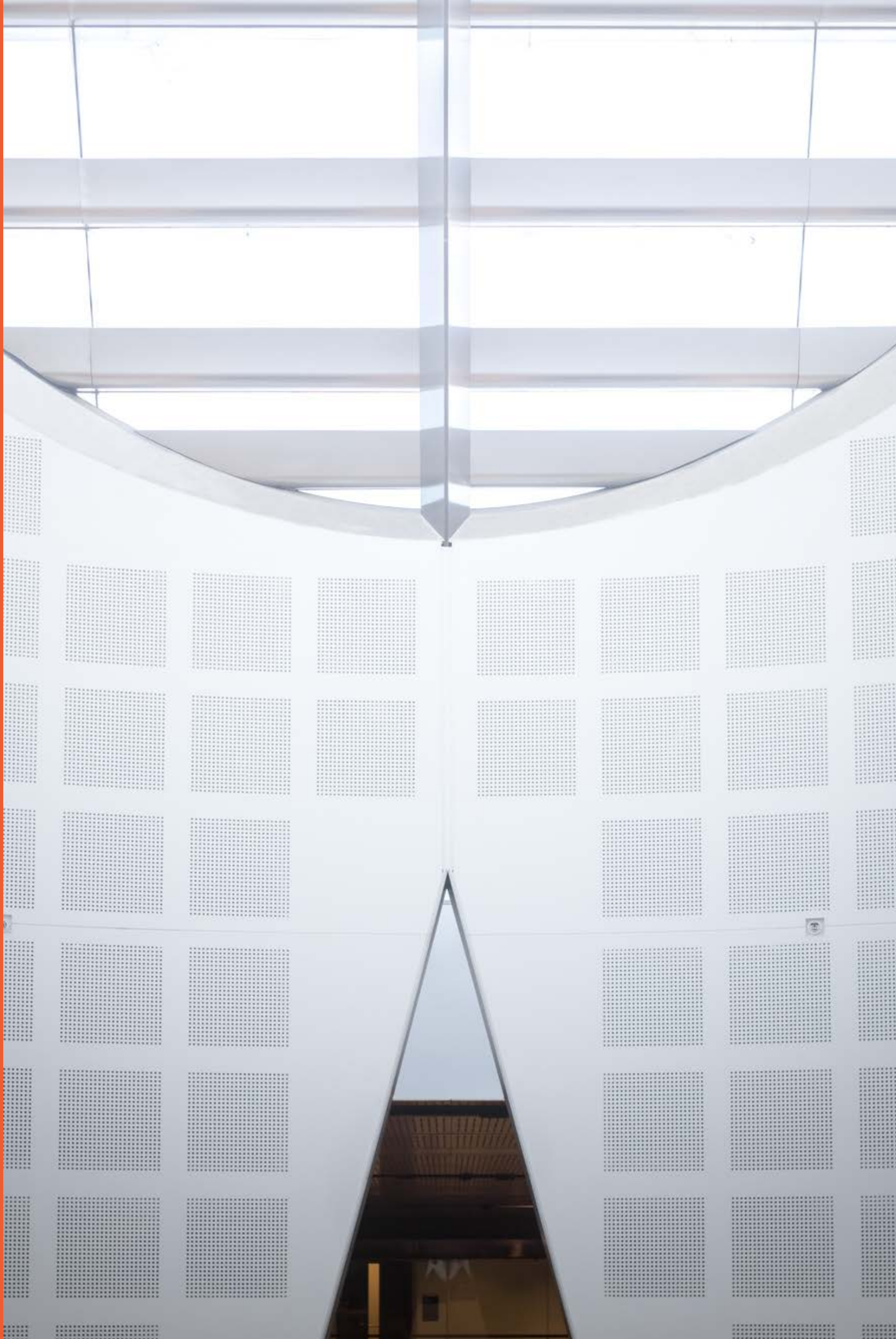
Lecture 13: Review

Lecturer: Dr. Karlos Ishac

Co-ordinator: Dr. Ravihansa Rajapakse
School of Computer Science

Some content is taken from the textbook publisher Wiley and previous Co-ordinator Dr. Andre van Renssen.

THE UNIVERSITY OF SYDNEY



Week 13 Tutorials

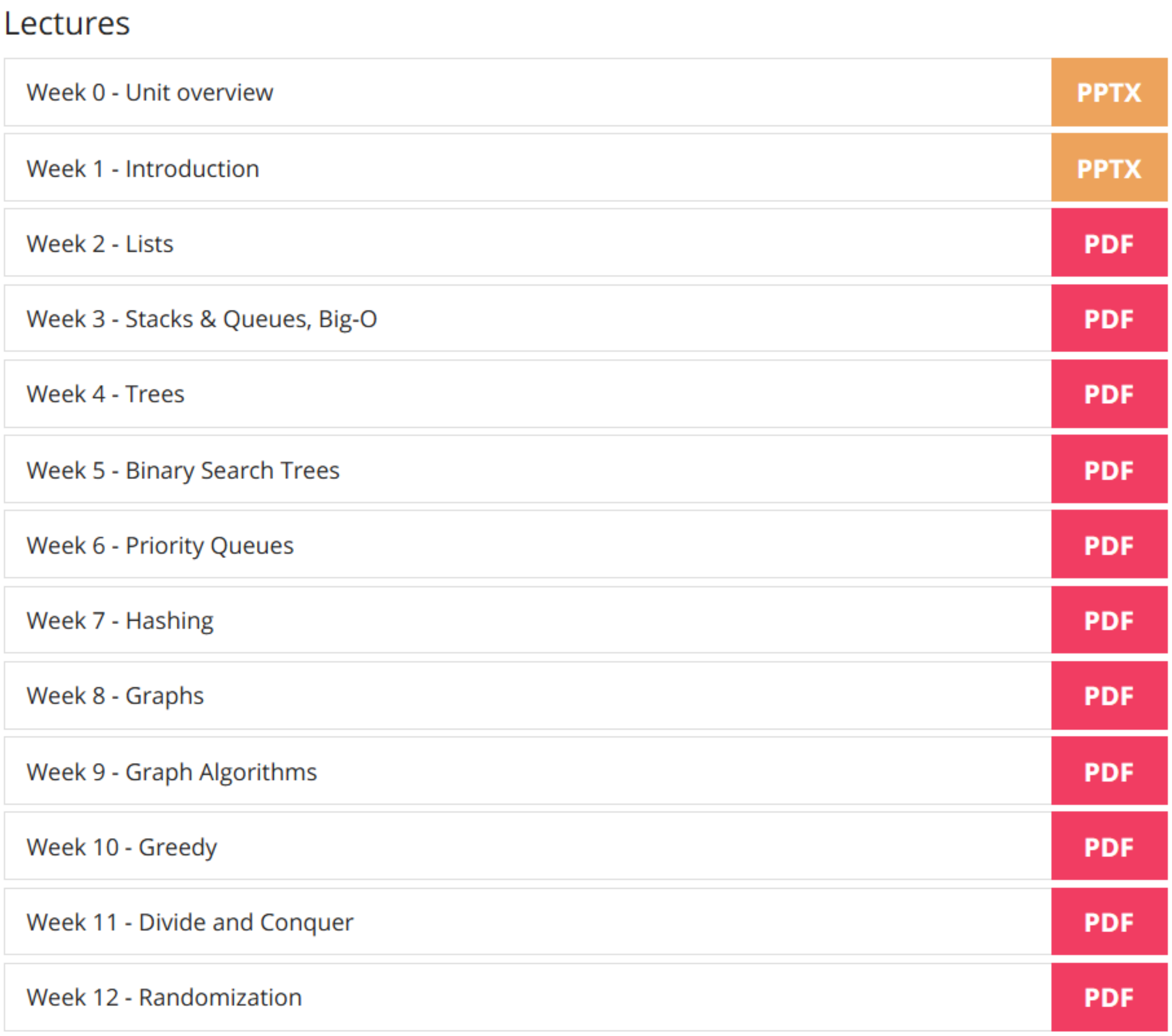
There is a review tutorial sheet. This is on purpose, so that you have the time to ask questions to your tutor.

Make the most of this!

Use this opportunity, as after this week we'll have reduced presence on Ed.

Looking back

It's been quite
a journey!



Lectures

Week 0 - Unit overview	PPTX
Week 1 - Introduction	PPTX
Week 2 - Lists	PDF
Week 3 - Stacks & Queues, Big-O	PDF
Week 4 - Trees	PDF
Week 5 - Binary Search Trees	PDF
Week 6 - Priority Queues	PDF
Week 7 - Hashing	PDF
Week 8 - Graphs	PDF
Week 9 - Graph Algorithms	PDF
Week 10 - Greedy	PDF
Week 11 - Divide and Conquer	PDF
Week 12 - Randomization	PDF

What is examinable?

Everything from the lectures, the referenced sections of the textbooks, the tutorials, the quizzes, the assignments. Exceptions to this rule:

- when explicitly labeled as non-examinable.
- probabilistic analysis of randomized algorithms

In general, if it happened during this unit, you are expected to know about it!

Focus on the things we put most emphasis on, as seen in tutorials and assignments

Final Exam Structure

2 hours writing plus 10 minutes reading time

4 questions worth in total 60 points – each worth 15 points

Worth 50% of overall COMP9123 grade

Final exam has a 40% barrier

Restricted Open Book:

- You can bring 1 A4 double-sided cheatsheet (handwritten or printed)
- A non-programmable calculator is allowed (although it is very unlikely you would require this).

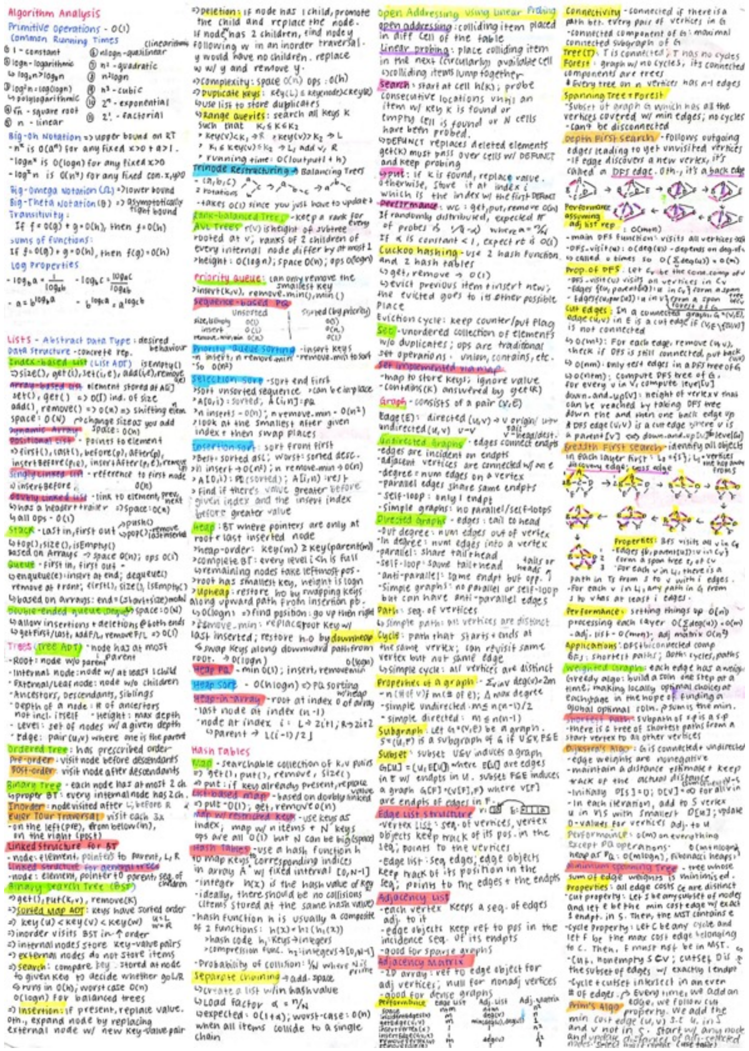
Question Structure

Design or modify an ADT/algorithm that solves a problem

Remember to:

- Describe your approach for each answer
- Prove correctness (if asked)
- Analyze complexity (if there's a space requirement, don't forget to analyze this as well)

Neatest summary page we came across



Algorithm Analysis
Primitive operations - O(1)
Common Running Times
① 1 - constant ⑥ nlogn - quasilinear (linearithmic)
② logn - logarithmic ⑦ n² - quadratic
↳ $\log_a n > \log_b n$ ⑧ n²logn
③ log²n = log(logn) ⑨ n³ - cubic
↳ polylogarithmic ⑩ 2^n - exponential
④ √n - square root ⑪ 2! - factorial
⑤ n - linear
Big-Oh Notation => upper bound on RT
- n^x is $O(a^n)$ for any fixed x>0 + a>1.
- $\log n^x$ is O(logn) for any fixed x>0
- $\log^x n$ is $O(n^y)$ for any fixed con. x,y>0
Big-Omega Notation (Ω) => lower bound
Big-Theta Notation (Θ) => asymptotically tight bound
Transitivity:
If f = O(g) + g = O(h), then f = O(h)
Sums of functions:
If f = O(g) + g = O(h), then f(g) = O(h)
Log Properties
- $\log_b a = \frac{1}{\log_a b}$ - $\log_b c = \frac{\log_a c}{\log_a b}$
- $a = b^{\log_b a}$ - $b^{\log_c a} = a^{\log_c b}$

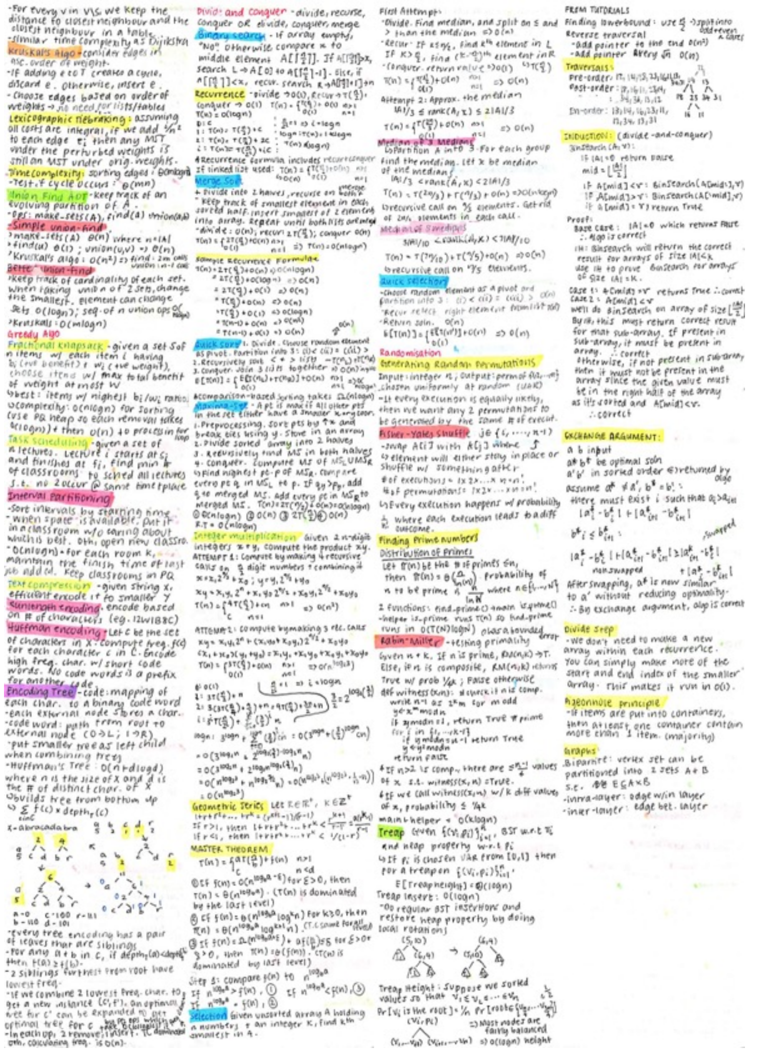
Lists - Abstract Data Type: desired behaviour
Data Structure - concrete rep.
Index-based List (List ADT) isEmpty()
=> size(), get(i), set(i,e), add(i,e), remove(i,e)
Array-based List element stored at A[i]
set(), get() => O(1) ind. of size
add(), remove() => O(n) => shifting n elem
space: O(N) ↳ change size as you add
Dynamic Array space: O(n)
Positional List - points to element
=> first(), last(), before(p), after(p), insertBefore(p,e), insertAfter(p,e), remove(p)
Singly Linked List - reference to first node
↳ insertBefore O(n)
Doubly Linked List - link to element, prev, next
↳ has a header + trailer => space: O(n)
↳ all ops - O(1)
Stack - Last in, first out ↳ push() ↳ pop() - remove last element
↳ top(), size(), isEmpty()
Based on Arrays -> space O(n); ops O(1)
Queue - First in, first out -
↳ enqueue(e): insert at end; dequeue() remove at front; first(), size(), isEmpty()
↳ based on arrays: end = (start+size) mod N
Double-ended queue (Deque) space: O(N)
↳ allow insertions + deletions @ both ends
↳ getFirst/Last, addF/L, removeF/L => O(1)
Trees (Tree ADT) - node has at most 1 parent
- Root: node w/o parent
- Internal node: node w/ at least 1 child
- External/Leaf node: node w/o children
- Ancestors, Descendants, Siblings
- Depth of a node: # of ancestors not incl. itself - Height: max depth
- Level: set of nodes w/ a given depth
- Edge: pair (u,v) where one is the parent
Ordered Tree: has prescribed order
Pre-order: visit node before descendants
Post-order: visit node after descendants
Binary Tree - each node has at most 2 ch.
↳ proper BT: every internal node has 2 ch.
Inorder: node visited after L, before R
Euler Tour Traversal: visit each 3x
- on the left (pre), from below (in), on the right (post)
Linked Structure for BT
- node: element, pointers to parent, L, R
Linked Structure for general trees
- node: element, pointer to parent, seq. of children
Binary Search Tree (BST)
=> get(), put(k,v), remove(k)
=> Sorted Map ADT: keys have sorted order
=> key(u) < key(v) < key(w) u=L w=R
=> inorder visits BST in ↑ order
=> internal nodes store key-value pairs
=> external nodes do not store items
=> Search: compare key stored at node to given key to decide whether go L/R
↳ runs in O(h); worst case O(n) O(logn) for balanced trees
=> Insertion: if present, replace value. Oth., expand node by replacing external node w/ new key-value pair
=> Deletion: if node has 1 child, promote the child and replace the node.
If node w has 2 children, find node y following w in an inorder traversal. y would have no children. replace w w/ y and remove y.
=> Complexity: space O(n) ops: O(h)
=> Duplicate keys: key(L) ≤ key(node) < key(R)
↳ use list to store duplicates
↳ Range queries: search all keys k such that $k_1 \le k \le k_2$
> key(v) < k_1 → R > key(v) > k_2 → L
> k_1 ≤ key(v) ≤ k_2 → L, add v, R
> running time: O(|output| + h)
Trinode Restructuring → Balancing Trees
- (a,b,c) 2 rotations
- takes O(1) since you just have to update
Rank-balanced Trees - keep a rank for every
AVL Trees r(v) is height of subtree rooted at v; ranks of 2 children of every internal node differ by at most 1
> height: O(log n); space O(n); ops O(logn)
Priority Queue: can only remove the smallest key
> insert(k,v), remove_min(), min()
Sequence-based PQ

	Unsorted	Sorted (by priority)
size, isEmpty	O(1)	O(1)
insert	O(1)	O(n)
remove_min, min	O(n)	O(1)

Priority Queue Sorting - insert keys - remove_min to sort
- n insert, n remove_min
- So O(n²)
Selection Sort - sort end first
> sort unsorted sequence > can be in place
> A[0,i): sorted; A[i,n]: PQ
> n inserts - O(n); n remove_min - O(n²)
> look at the smallest after given index + then swap places
Insertion Sort: sort from first
> Best: sorted asc; Worst: sorted desc.
> n insert → O(n²); n remove_min → O(n)
> A[0,i): PQ (sorted); A[i,n): rest
> Find if there's value greater before given index and the insert index before greater value
Heap: BT where pointers are only at root + last inserted node
> heap-order: key(m) ≥ key(parent(m))
> complete BT: every level i < h is full
↳ remaining nodes take leftmost pos.
> root has smallest key, height is logn
> Upheap: restore h-o by swapping keys along upward path from insertion pt.
↳ O(logn) => find position: go up then right
> remove_min: replace root key w/ last inserted; restore h-o by downheap
↳ swap keys along downward path from root. → O(logn) O(logn)
Heap PQ - min O(1); insert, removemin
Heap Sort - O(nlogn) => PQ sorting w/ heap
Heap-in-Array - root at index 0 of array
- last node at index (n-1)
- node at index i: L → 2i+1; R → 2i+2
↳ parent → ⌊(i-1)/2⌋
Hash Tables
Map - searchable collection of k,v pairs
=> get(), put(), remove, size()
=> put: if key already present, replace value
List-based Map - based on doubly linked
=> put - O(1); get, remove O(n)
Map w/ restricted keys - use keys as index; map w/ n items + N keys ops are all O(1) but N can be big (space)
Hash Tables - use a hash function h to map keys to corresponding indices in array A w/ fixed interval [0,N-1]
- integer h(x) is the hash value of key
- ideally, there should be no collisions (items stored at the same hash value)
- hash function h is usually a composite of 2 functions: $h(x) = h_2(h_1(x))$
> hash code h_1: keys → integers
> compression func. h_2: integers → [0,N-1]
- Probability of collision: 1/N where N is prime
Separate Chaining → add. space
↳ create a list w/in hash value
↳ Load factor α = n/N
↳ expected: O(1+α); worst-case: O(n) when all items collide to a single chain
Open Addressing using Linear Probing
open addressing: colliding item placed in diff cell of the table
Linear probing: place colliding item in the next (circularly) available cell
↳ colliding items lump together
Search: start at cell h(k); probe consecutive locations until an item w/ key k is found or empty cell is found or N cells have been probed.
↳ DEFUNCT replaces deleted elements
get(k) must pass over cells w/ DEFUNCT and keep probing
↳ put: if k is found, replace value. Otherwise, store it at index i which is the index w/ the first DEFUNCT
Performance: WC: get, put, remove O(n)
If randomly distributed, expected # of probes is 1/(1-α) where α = n/N
If α is constant < 1, expect rt is O(1)
Cuckoo hashing - use 2 hash function and 2 hash tables
↳ get, remove → O(1)
↳ evict previous item + insert new; the evicted goes to its other possible place
Eviction cycle: keep counter/put flag
Set - unordered collection of elements w/o duplicates; ops are traditional set operations: union, contains, etc.
Set implemented via map
- map to store keys; ignore value
- contains(k) answered by get(k)
Graph - consists of a pair (V,E)
Edge(E): directed (u,v) → v origin/ u→v tail; undirected (u,v) v−v v - head/dest.
Undirected Graphs - edges connect endpts
- edges are incident on endpts
- adjacent vertices are connected w/ an e
- degree = num edges on a vertex
- parallel edges share same endpts
- self-loop: only 1 endpt
- simple graphs: no parallel/self-loops
Directed Graphs - edges: tail to head
- out degree: num edges out of vertex
- in degree: num edges into a vertex
- parallel: share tail+head
- self-loop: same tail+head tails or heads
- anti-parallel: same endpt but opp.
- simple graphs: no parallel or self-loop but can have anti-parallel edges
Path: seq. of vertices
↳ simple path: all vertices are distinct
Cycle: path that starts + ends at the same vertex; can revisit same vertex but not same edge
↳ simple cycle: all vertices are distinct
Properties of a graph - $\sum_{v \in V} \deg(v) = 2m$
- n (# of V); m (# of E); Δ max degree
- simple undirected: m ≤ n(n-1)/2
- simple directed: m ≤ n(n-1)
Subgraph: Let G = (V,E) be a graph. S = (U,F) is a subgraph of G if U ⊆ V, F ⊆ E
Subset: subset U ⊆ V induces a graph G[U] = (U, E[U]) where E[U] are edges in E w/ endpts in U. subset F ⊆ E induces a graph G[F] = (V[F], F) where V[F] are endpts of edges in F.
Edge List Structure V: E:
- Vertex list: seq. of vertices, vertex objects keep track of its pos. in the seq.; points to the vertices
- Edge list: seq. edges; edge objects keep track of its position in the seq.; points to the edges + the endpts
Adjacency List
- each vertex keeps a seq. of edges adj. to it
- edge objects keep ref to pos in the incidence seq. of its endpts
- good for sparse graphs
Adjacency Matrix
- 2D array: ref to edge object for adj vertices; null for nonadj vertices
- good for dense graphs

Performance	Edge List	Adj. List	Adj. Matrix
space	n+m	n+m	n²
incidentEdges(v)	m	deg(v)	n
getEdge(u,v)	m	min(deg(u),deg(v))	1
insertVertex(x)	1	1	n²
insertEdge(u,v,x)	1	1	1
removeVertex(v)	m	deg(v)	n²
removeEdge(e)	1	1	1

Connectivity - connected if there is a path bet. every pair of vertices in G
- connected component of G: maximal connected subgraph of G
Tree (T): T is connected, T has no cycles
Forest: graph w/ no cycles, its connected components are trees
* Every tree on n vertices has n-1 edges
Spanning Tree + Forest
- Subset of graph G which has all the vertices covered w/ min edges; no cycles
- can't be disconnected
Depth First Search - follows outgoing edges leading to yet unvisited vertices
- if edge discovers a new vertex, it's called a DFS edge. Oth., it's a back edge
Performance assuming adj list rep: O(m+n)
- main DFS function: visits all vertices O(n)
- DFS_visit(u): O(deg(u)) - depends on deg of u
↳ called n times so $O(\sum_u \deg(u)) = O(m)$
Prop. of DFS: Let C_v be the conn. comp of v
- DFS_visit(u) visits all vertices in C_v
- Edges {(u, parent[u]): u in C_v} form a span tree of C_v
- Edges {(u,parent[u]): u in V} form a span forest of G
Cut Edges: In a connected graph G = (V,E), edge (u,v) in E is a cut edge if (V, E∖{(u,v)}) is not connected
↳ O(m²): For each edge, remove (u,v), check if DFS is still connected, put back (u,v)
↳ O(nm): only test edges in a DFS tree of G
↳ O(n+m): compute DFS tree of G. For every v in V, compute level[v]
down_and_up[v]: height of vertex v that can be reached by taking DFS tree down the and then one back edge up
A DFS edge (u,v) is a cut edge where v is a parent[v] and down_and_up[v] > level[u]
Breadth First Search - identify all objects in each layer first: L_0 = {s}; L_1 = vertices one hop away from s
discovery edge; cross edge
Properties: BFS visits all v in C_s
- Edges {(v, parent[v]): v in C_s} form a span tree T_s of C_s
- For each v in L_i, there is a path in T_s from s to v with i edges.
- For each v in L_i, any path in G from s to v has at least i edges.
Performance: setting things up O(n); processing each layer $O(\sum \deg(u)) = O(m)$
- adj. list - O(m+n); Adj matrix O(n²)
Applications: DFS: biconnected comp; BFS: shortest paths; Both: cycles, paths
Weighted Graphs: each edge has a weight
Greedy algo: build a soln one step at a time, making locally optimal choices at each stage in the hope of finding a global optimal soln. p sum is the min.
Shortest Path: subpath of s.p is a s.p
- there is a tree of shortest paths from a start vertex to all other vertices
Dijkstra's Algo: G is connected + undirected
- edge weights are nonnegative
- maintain a distance estimate + keep track of the actual distance V-S
- Initially D[s] = 0; D[v] = ∞ for all v in
- In each iteration, add to S vertex u in V∖S with smallest D[u]; update D-values for vertices adj. to u
Performance: O(m) on everything except PQ operations: O((m+n)logn)
heap as PQ: O(mlogn); Fibonacci heaps:
Minimum Spanning Tree - tree whose sum of edge weights is minimised.
Properties: all edge costs c_e are distinct
- Cut property: Let S be any subset of nodes and let e be the min cost edge w/ exactly 1 endpt. 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, f must not be in MST.
- Cut, nonempty S ⊂ V; cutset D is the subset of edges w/ exactly 1 endpt
- Cycle + cutset intersect in an even # of edges. ↳ Every time, we add an edge, we follow cut
Prim's Algo property. We add the min cost edge (u,v) s.t. u in S and v not in S. Start w/ any node and update distances of non-selected nodes. Select [illegible] (use table)



- For every v in V∖S, we keep the distance to closest neighbour and the closest neighbour in a table.
- similar time complexity as Dijkstra
Kruskal's Algo - consider edges in asc. order of weight.
- If adding e to T creates a cycle, discard e. Otherwise, insert e.
- Choose edges based on order of weights → no need for lists/tables
Lexicographic Tiebreaking: assuming all costs are integral, if we add i/n² to each edge e_i then any MST under the perturbed weights is still an MST under orig. weights.
Time complexity: sorting edges: Θ(mlogn)
- Test if cycle occurs: Θ(mn)
Union Find ADT - keep track of an evolving partition of A.
- Ops: make_sets(A), find(a), union(a,b)
Simple union-find
> make_sets(A) Θ(n) where n = |A|
> find(u) Θ(1); union(u,v) => Θ(n)
> Kruskal's algo: O(n²) => find: 2m calls union: n-1 calls
Better union-find
> keep track of cardinality of each set. when taking union of 2 sets, change the smallest. element can change sets O(logn); seq. of n union ops O(nlogn)
> Kruskals: O(mlogn)
Greedy Algo
Fractional Knapsack - given a set S of n items w/ each item i having b_i (+ve benefit) + w_i (+ve weight), choose items w/ max total benefit of weight at most W
↳ best: items w/ highest b_i/w_i ratio
↳ complexity: O(nlogn) for sorting (use PQ heap so each removal takes O(logn)) + then O(n) to process in for loop
Task Scheduling - given a set of n lectures. Lecture i starts at s_i and finishes at f_i. find min # of classrooms to sched all lectures s.t. no 2 occur @ same time + place
Interval Partitioning
- Sort intervals by starting time
- when space is available, put it in a classroom w/o caring about which is best. Oth., open new classro.
- O(nlogn) - for each room k, maintain the finish time of last job added. Keep classrooms in PQ
Text Compression - given string X, efficient encode it to smaller Y
Runlength encoding. encode based on # of characters (eg. 12W1B8C)
Huffman encoding - Let C be the set of characters in X. compute freq. f(c) for each character c in C. Encode high freq. char. w/ short code words. No code words is a prefix for another code.
Encoding Tree - code: mapping of each char. to a binary code word
- each external node stores a char.
- code word: path from root to external node (0→L; 1→R)
- put smaller tree as left child when combining trees
- Huffman's Tree: O(n + dlogd) where n is the size of X and d is the # of distinct char. of X
↳ builds tree from bottom up
↳ $\sum_{c \in C} f(c) \times \mathrm{depth}_T(c)$
X = abracadabra a 5 b 2 c 1 d 1 r 2
a - 0 c - 100 r - 11
b - 110 d - 101
- every tree encoding has a pair of leaves that are siblings
- For any a + b in C, if $\mathrm{depth}_T(a) < \mathrm{depth}_T(b)$ then f(a) ≥ f(b).
- 2 siblings furthest from root have lowest freq.
- If we combine 2 lowest freq. char. to get a new instance (C', f'), an optimal tree for C' can be expanded to get optimal tree for C
- In each op, 2 remove, 1 insert. TC dominated by PQ ops which is O(nlogn); oth. calculating freq. is O(n).
Divide and Conquer - divide, recurse, conquer OR divide, conquer, merge
Binary search - If array empty, "No". Otherwise compare x to middle element A[⌈n/2⌉]. If A[⌈n/2⌉] > x, search L → A[0] to A[⌈n/2⌉-1]. Else, if A[⌈n/2⌉] < x, recur. search R → A[⌈n/2⌉+1] to n
Recurrence - divide → O(1), Recur → T(n/2), conquer → O(1) $T(n) = \begin{cases} T(n/2) + O(1) & n>1 \\ O(1) & n=1 \end{cases}$
T(n) = O(logn)
0: c
1: T(n) = T(n/2) + c
2: T(n) = T(n/4) + 2c
i: T(n) = T(n/2^i) + ic
: n/2^i = 1 => i = logn
: logn: T(n) = 1 + clogn
: T(n) ∈ O(logn)
* Recurrence formula includes recur + conquer
If linked list used: $T(n) = \begin{cases} T(n/2) + O(n) & n>1 \\ O(1) & n=1 \end{cases}$ → O(n)
Merge Sort
+ Divide into 2 halves, recurse on both + merge
- Keep track of smallest element in each sorted half. insert smallest of 2 elements into array. Repeat until both lists are empty
- divide: O(n); recur 2T(n/2); conquer O(n)
$T(n) = \begin{cases} 2T(n/2) + O(n) & n>1 \\ O(1) & n=1 \end{cases}$ => T(n) = O(nlogn)
Sample Recurrence Formulae
T(n) = 2T(n/2) + O(n) => O(nlogn)
= 2T(n/2) + O(logn) => O(n)
= 2T(n/2) + O(1) => O(n)
= T(n/2) + O(n) => O(n)
= T(n/2) + O(1) => O(logn)
= T(n-1) + O(n) => O(n²)
= T(n-1) + O(1) => O(n)
Quick Sort 1. Divide. Choose random element as pivot. Partition into 3: (i) < (ii) = (iii) >
2. Recursively sort < + > lists → $T(n_L) + T(n_R)$
3. Conquer: Join 3 lists together => O(n) w/ linked list
$E[T(n)] = \begin{cases} E[T(n_L) + T(n_R)] + O(n) & n>1 \\ O(1) & n=1 \end{cases}$ => O(nlogn)
* comparison-based sorting takes Ω(nlogn)
Maxima-Set - A pt is max if all other pts in the set either have a smaller x or y coor.
1. Preprocessing. Sort pts by ↑x and break ties using y. Store in an array
2. Divide sorted array into 2 halves
3. Recursively find MS in both halves
4. Conquer. Compute MS of $MS_L \cup MS_R$
↳ Find highest pt. p of MS_R. Compare every pt q in MS_L to p. If $q_y > p_y$, add q to merged MS. Add every pt in MS_R to merged MS. T(n) = 2T(n/2) + O(n) = O(nlogn)
① O(nlogn) ② O(n) ③ 2T(n/2) ④ O(n)
R.T = O(nlogn)
Integer multiplication Given 2 n-digit integers x + y, compute the product xy.
ATTEMPT 1: Compute by making 4 recursive calls on n/2 digit numbers + combining it
$x = x_1 2^{n/2} + x_0$; $y = y_1 2^{n/2} + y_0$
$xy = x_1y_1 2^n + x_1y_0 2^{n/2} + x_0y_1 2^{n/2} + x_0y_0$
$T(n) = \begin{cases} 4T(n/2) + cn & n>1 \\ c & n=1 \end{cases}$ => O(n²)
ATTEMPT 2: compute by making 3 rec. calls
$xy = x_1y_1 2^n + (x_1y_0 + x_0y_1) 2^{n/2} + x_0y_0$
$(x_1 + x_0)(y_1 + y_0) = x_1y_0 + x_1y_0 + x_0y_1 + x_0y_0$
$T(n) = \begin{cases} 3T(n/2) + O(n) & n>1 \\ O(1) & n=1 \end{cases}$ => $O(n^{\log_2 3})$
0: O(1)
1: 3T(n/2) + n
2: 3(3T(n/4) + n/2) + n = 9T(n/4) + 3/2 n + n
i: $3^i T(n/2^i) + \sum_{j=0}^{i-1} (3/2)^j n$
n/2^i = 1 => i = logn
3/2 = $2^{\log_2(3/2)}$
logn: $3^{\log n} + \sum_{j=0}^{\log n} (3/2)^j cn = O(3^{\log n} + (3/2)^{\log n} cn)$
$= O(3^{\log_2 n} + 2^{\log_2(3/2) \cdot \log_2 n} n)$
$= O(3^{\log_2 n} + 2^{\log_2 n^{\log_2(3/2)}} n)$
$= O(n^{\log_2 3} + n^{\log_2 3/2} n) = O(n^{\log_2 3} + (n^{\log_2 3} \cdot \frac{1}{n} \cdot n))$
$= O(n^{\log_2 3})$
Geometric Series Let r ∈ ℝ⁺, k ∈ ℤ⁺
$1 + r + r^2 + \dots + r^k = (r^{k+1} - 1)/(r-1)$
If r > 1, then $1 + r + r^2 + \dots + r^k < \frac{r^{k+1}}{r-1}$
If r < 1, then $1 + r + r^2 + \dots + r^k < 1/(1-r)$
MASTER THEOREM
$T(n) = \begin{cases} aT(n/b) + f(n) & n>1 \\ c & n<d \end{cases}$
① If $f(n) = O(n^{\log_b a - \epsilon})$ for ε > 0, then $T(n) = \Theta(n^{\log_b a})$. (T(n) is dominated by the last level)
② If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for k ≥ 0, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$. (T.C same for all levels)
③ If $f(n) = \Omega(n^{\log_b a + \epsilon})$ + af(n/b) ≤ δf(n) for ε > 0 or δ > 0, then T(n) = Θ(f(n)). (T(n) is dominated by last level)
Step 1: compare f(n) to $n^{\log_b a}$
If $n^{\log_b a}$ > f(n), ① If $n^{\log_b a}$ < f(n), ③
If $n^{\log_b a}$ = f(n), ②
Selection Given unsorted array A holding n numbers + an integer k, find kth smallest in A.
First Attempt:
- Divide. Find median, and split on ≤ and > than the median => O(n)
- Recur: If k ≤ n/2, find kth element in L. If k > n/2, find (k - n/2)th element in R ↳ T(n/2)
- Conquer. return value => O(1)
$T(n) = \begin{cases} T(n/2) + O(n) & n>1 \\ O(1) & n=1 \end{cases}$ => O(n)
Attempt 2: Approx. the median
|A|/3 ≤ rank(A, x) ≤ 2|A|/3
$T(n) = \begin{cases} T(2n/3) + O(n) & n>1 \\ O(1) & n=1 \end{cases}$ => O(n)
Median of 3 Medians
↳ Partition A into 3. For each group find the median. Let x be median of the medians
|A|/3 < rank(A, x) < 2|A|/3
T(n) = T(2n/3) + T(n/3) + O(n) => O(nlogn)
↳ recursive call on n/3 elements. Get rid of 2n/3 elements in each call.
Median of 5 medians
3|A|/10 < rank(A, x) < 7|A|/10
T(n) = T(7n/10) + T(n/5) + O(n) => O(n)
↳ recursive call on n/5 elements.
Quick Selection
- Choose random element as a pivot and partition into 3: (i) < (ii) = (iii) > O(n)
- Recur select right element from list
- Return soln. O(n)
$E[T(n)] = \begin{cases} E[T(n')] + O(n) \\ O(1) \end{cases}$ => O(n)
Randomisation
Generating Random Permutations
Input: integer n; Output: perm of {0, ..., n-1} chosen uniformly at random (UAR)
- If every execution is equally likely, then we want any 2 permutations to be generated by the same # of execut.
Fisher-Yates Shuffle j ∈ {i, ..., n-1}
- swap A[i] with A[j] where j
↳ element will either stay in place or shuffle w/ something other
# of executions = 1×2×...×n = n!
# of permutations = 1×2×...×n = n!
↳ Every execution happens w/ probability 1/n! where each execution leads to a diff outcome.
Finding Prime numbers
Distribution of Primes
Let π(n) be the # of primes ≤ n, then $\pi(n) = \Theta(\frac{n}{\ln(n)})$. Probability of n to be prime is $\frac{1}{\ln N}$ where n ∈ {1, ..., N}
2 functions: find_prime() + main is_prime()
- helper is_prime runs T(n) so find_prime runs in O(T(N)logN) plus a bounded error
Rabin-Miller - testing primality
Given n + k. If n is prime, RM(n,k) → T. Else, if n is composite, RM(n,k) returns True w/ prob 1/4^k; False otherwise
def witness(x,n): # check if n is comp.
write n-1 as $2^k m$ for m odd
y ← x^m mod n
if y mod n = 1, return True # prime
for i in {1, ..., k-1}
if y mod n = n-1 return True
y ← y² mod n
return False
* If n > 2 is comp., there are ≤ (n-1)/4 values of x s.t. witness(x,n) = True.
* If we call witness(x,n) w/ k diff values of x, probability ≤ 1/4^k
main + helper = O(klogn)
Treap Given $\{(v_i, p_i)\}_{i=1}^n$, BST w.r.t v_i and heap property w.r.t p_i
↳ If p_i is chosen UAR from [0,1] then for a treap on $\{(v_i, p_i)\}_{i=1}^n$,
E[Treap height] = Θ(logn)
Treap insert: O(logn)
- Do regular BST insertions and restore heap property by doing local rotations
(5,10) (6,4) → (6,4) (5,10)
Treap Height: Suppose we sorted values so that $v_1 \le v_2 \le \dots \le v_n$
Pr[v_i is the root] = 1/n Pr[root ∈ {$v_{n/4}$, ..., $v_{3n/4}$}] = 1/2
=> Most nodes are fairly balanced
=> O(logn) height
FROM TUTORIALS
Finding lowerbound: use n/2 → split into odd + even cases
Reverse traversal
- add pointer to the end O(n²)
- add pointer every √n O(n)
Traversals:
Pre-order: 17, 14, 18, 23, 16, 11, 34, 13, 31
Post-order: 18, 16, 11, 23, 14, 34, 31, 13, 17
In-order: 18, 14, 16, 23, 11, 17, 34, 13, 31
INDUCTION: (divide-and-conquer)
BinSearch(A, v):
if |A| = 0 return False
mid = ⌊|A|/2⌋
if A[mid] < v: BinSearch(A[mid:], v)
if A[mid] > v: BinSearch(A[:mid], v)
if A[mid] = v return True
Proof:
Base case: |A| = 0 which returns False
∴ Algo is correct
IH: BinSearch will return the correct result for arrays of size |A| < k
Use IH to prove BinSearch for arrays of size |A| = k.
Case 1: A[mid] = v returns True ∴ correct
Case 2: A[mid] < v
we'll do BinSearch on array of size ⌊|A|/2⌋
By IH, this must return correct result for that sub-array. If present in sub-array, it must be present in array. ∴ correct
Otherwise, if not present in sub-array, then it must not be present in the array since the given value must be in the right half of the array as it's sorted and A[mid] < v.
∴ correct
EXCHANGE ARGUMENT:
a b input
a* b* be optimal soln
a' b' in sorted order ⇐ returned by algo
assume a* ≠ a', b* = b'.
there must exist i such that $a_i > a_{i+1}$
$|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}|$
$b^*_i \le b^*_{i+1}$
$|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}| \ge |a^*_{i+1} - b^*_i| + |a^*_i - b^*_{i+1}|$
nonswapped swapped
After swapping, a* is now similar to a' without reducing optimality.
∴ By exchange argument, algo is correct
Divide Step
- We don't need to make a new array within each recurrence. You can simply make note of the start and end index of the smaller array. This makes it run in O(1).
Pigeonhole Principle
- If items are put into containers, then at least one container contain more than 1 item. (majority)
Graphs
Bipartite: vertex set can be partitioned into 2 sets A + B s.t. E ⊆ A×B
- intra-layer: edge w/in layer
- inter-layer: edge bet. layer

Do’s and Don’ts

Exam is in person.

Check your exam timetable for details on the venue!

Restricted open book exam:

- Making the **cheatsheet yourself is highly recommended**
- **Never** copy text verbatim from anywhere, including the slides (this is grounds for academic dishonesty case). If you refer to anything from the permitted material, write in your own words

Start your submission with your student ID

- Don’t include your name

Do’s and Don’ts

Ensure to write your answers only in the provided boxes in the given booklet. Anything you write outside these boxes may not be detected properly during the scanning process.

- You do not need to fill the entire space for a given question, as extra space is allocated intentionally.
- If you run out of space, first indicate this in the original space given, and then use the blank pages at the end of the paper and clearly name the question and sub-section.
- You could also use these blank pages for rough work.
- Do not attach any extra paper to the provided booklet.

Exam technique

Read all questions to see which ones you can answer quickly

Plan how you will allocate time (wisely)

Start with easy problems and move to harder ones

Write clearly and efficiently

- Start with outline/bullet points, then expand if you have time
- No need for fancy style or overly formal

Pragmatic Advice

- It's a good idea to check the exam venue ahead of time
- Plan to arrive ahead of time (don't rely on public transport running smoothly on the day of the exam)
- Bring water, spare pens, and ID
- Start by writing your student ID. Do not write your name on the exam (marking is anonymous)
- Breathe and relax
- Follow the instruction of the invigilator

More info:

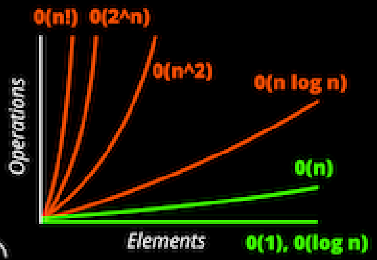
https://www.sydney.edu.au/students/exams/in-person.html

LEGEND

TIME Complexity VS. **SPACE** Complexity

Good Fair Bad

Good Fair Bad



<BIG-O-CHEATSHEET>

</>

www.bigocheatsheet.com

DATA STRUCTURE Operations

DATA Structure	**TIME** Complexity: Average				Worst				**SPACE** Complexity: Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

ARRAY SORTING Algorithms

ARRAY Algorithms	**TIME** Complexity: Best	Average	Worst	**SPACE** Complexity: Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(1)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)t	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

How to make your USS feedback count

Your Unit of Study Survey (USS) feedback is confidential.

It's a way to share what you enjoyed and found most useful in your learning, and to provide constructive feedback. It's also a way to 'pay it forward' for the students coming behind you, so that their learning experience in this class is as good, or even better, than your own.

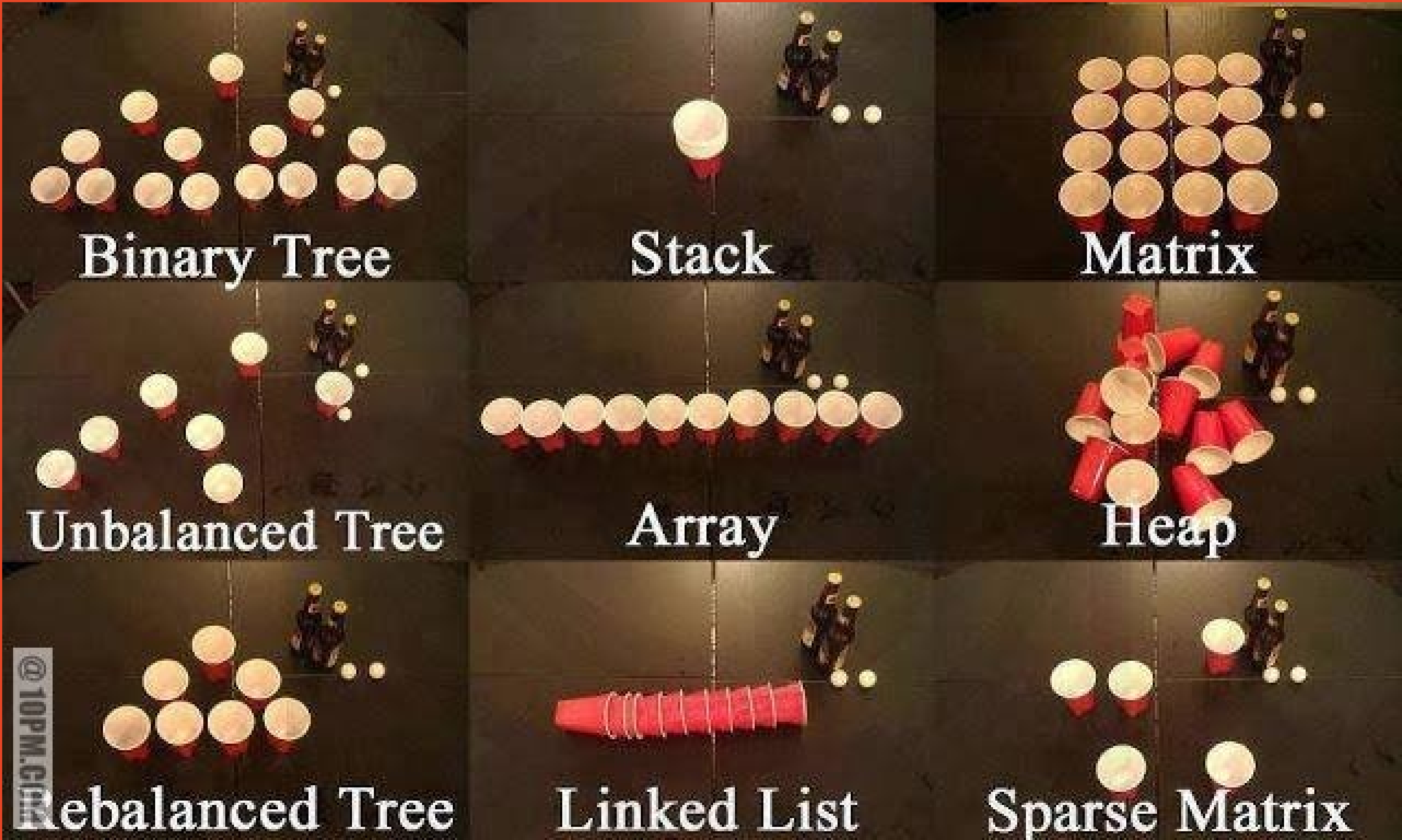
When you complete your USS survey (https://student-surveys.sydney.edu.au), please:

Be specific.
Which class tasks, assessments or other activities helped you to learn? *Why* were they helpful? Which one(s) *didn't* help you to learn? *Why* didn't they work for you?

Be constructive.
What practical changes can you suggest to class tasks, assessments or other activities, to help the next class learn better?

Be relevant.
Imagine you are the teacher. What sort of feedback would you find most useful to help make your teaching more effective?

A Brief Review



Big O Examples

(5) //Loop

```
for x in 1…N:
    statements // O(1)
end
```

(6) //Complex Loop

```
for x in 1…N:
    statements // O(n)
end
```

(7) //Nested Loop

```
for x in 1…N:
    for y in 1…M:
        statements // O(1)
    end
end
```

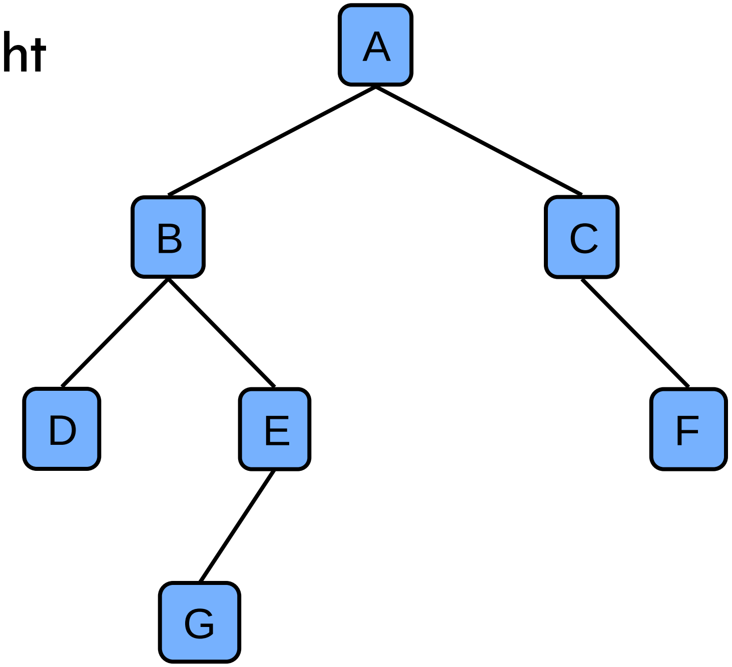
(8) //Complex Nested Loop

```
for x in 1…N:
    for y in 1…M:
        statements // O(n^2)
    end
end
```

Binary Trees

A binary tree is an ordered tree with the following properties:

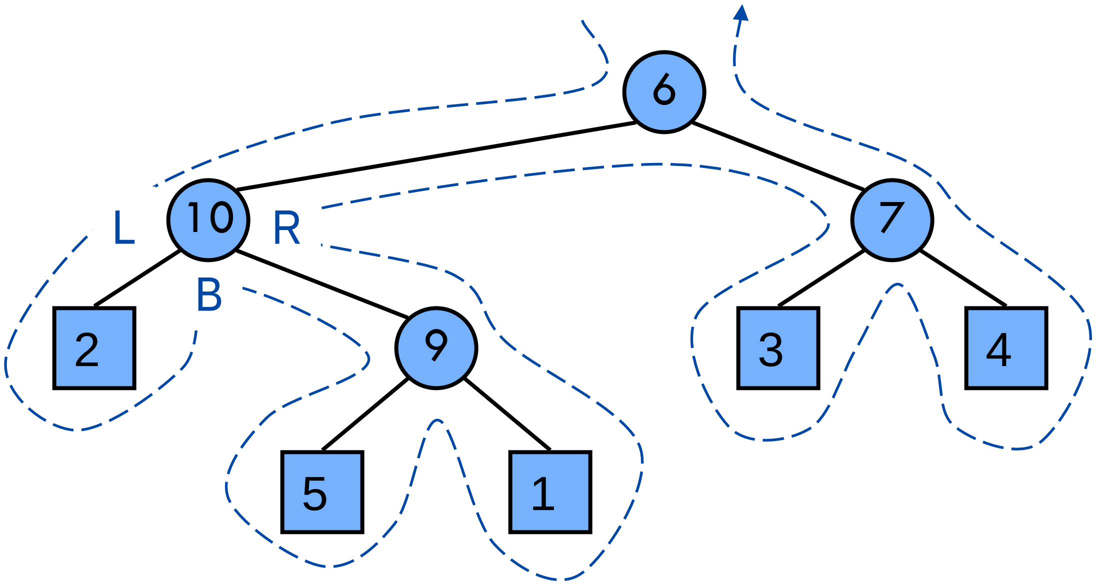
- Each internal node has at most two children
- Each child node is labeled as a left child or a right child
- Child ordering is left followed by right



The right/left subtree is the subtree root at the right/left child.

We say the tree is proper if every internal node has two children

Euler Tour Traversal



6,10,2,2,2,10,9,5,5,5,9,1,1,1,9,10,6,7,3,3,3,7,4,4,4,7,6

Preorder (first visit): 6, 10, 2, 9, 5, 1, 7, 3, 4

Inorder (second visit): 2, 10, 5, 9, 1, 6, 3, 7, 4

Postorder (third visit): 2, 5, 1, 9, 10, 3, 4, 7, 6

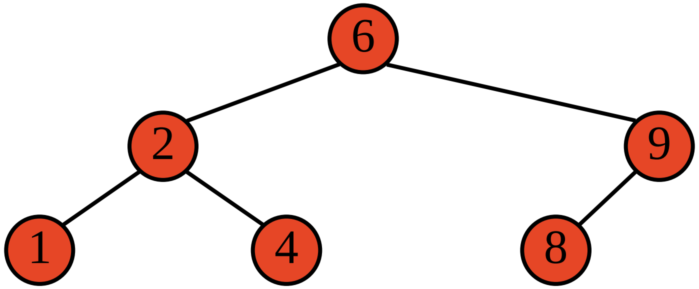
Binary Search Trees (BST)

A binary search tree is a binary tree storing keys (or key-value pairs) satisfying the following BST property

For any node v in the tree and
any node u in the left subtree of v and
any node w in the right subtree of v,

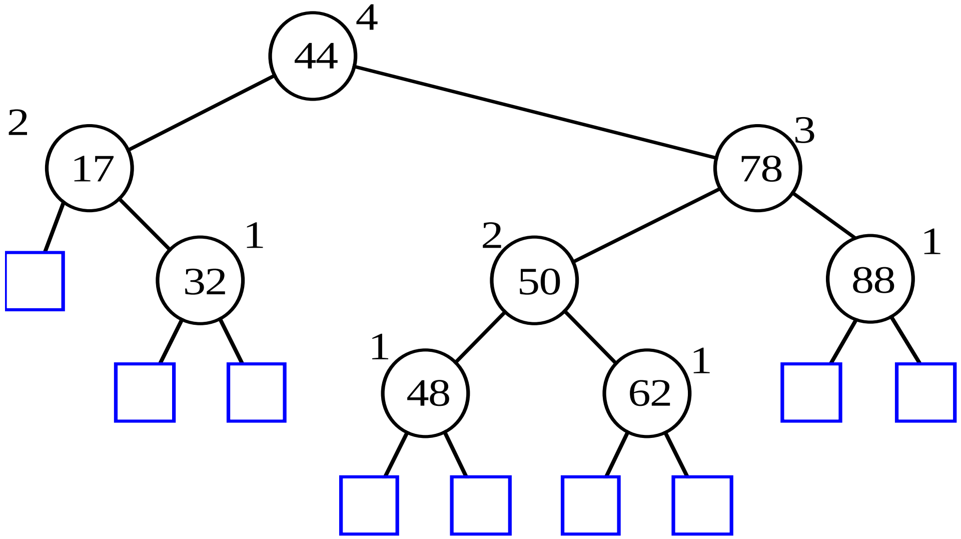
key(u) < key(v) < key(w)

Note that an inorder traversal of a binary search tree visits the keys in increasing order.



AVL Tree Definition

AVL trees are rank-balanced trees, where r(v) is its height of the subtree rooted at v



Balance constraint: The ranks of the two children of every internal node differ by at most 1.

Fact: The height of an AVL tree storing n keys is O(log n).

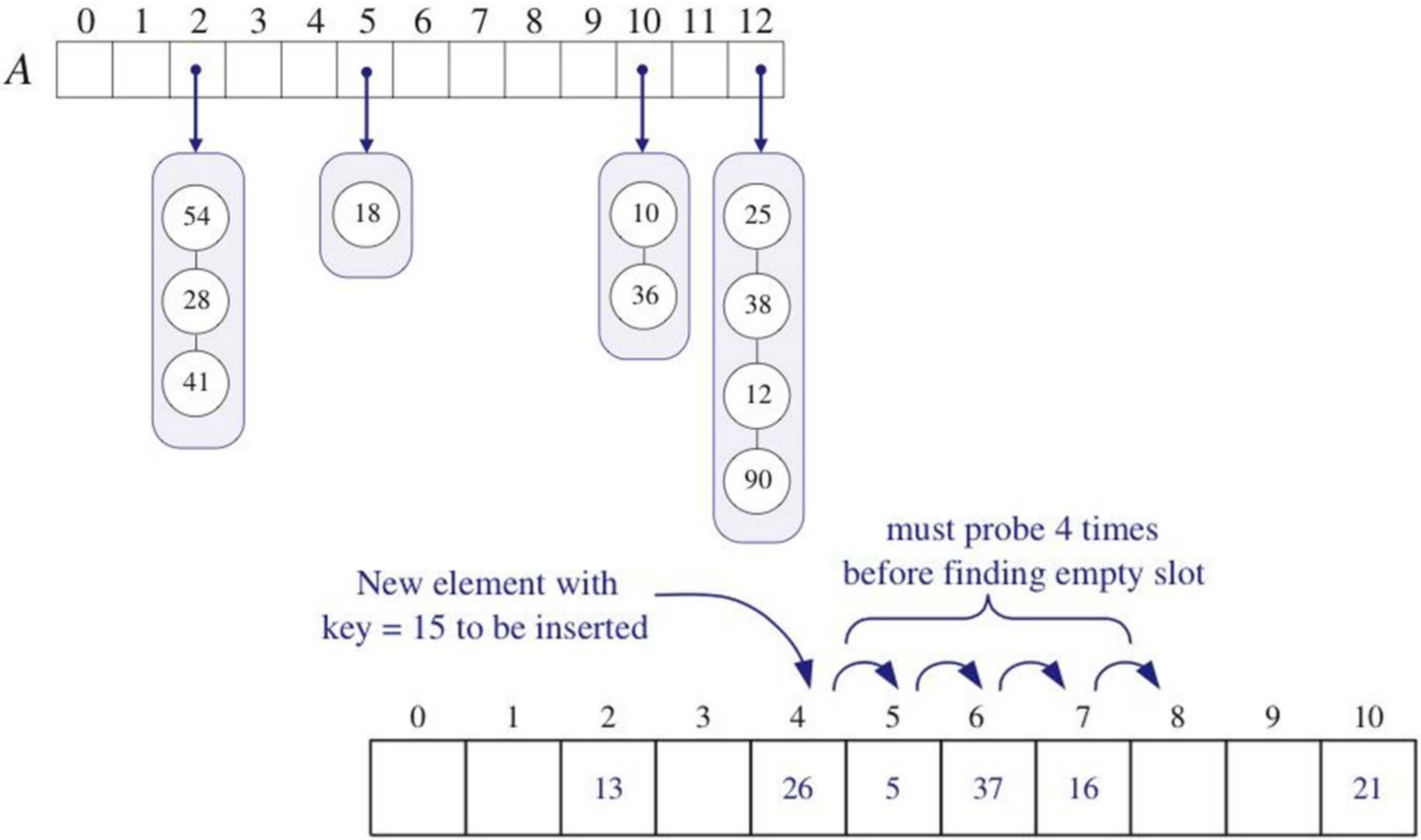
Selection-Sort

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Insertion-Sort

By Swfung8 - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=14961606

Chaining versus probing


A
0 1 2 3 4 5 6 7 8 9 10 11 12
54
28
41
18
10
36
25
38
12
90
New element with key = 15 to be inserted
must probe 4 times before finding empty slot

0	1	2	3	4	5	6	7	8	9	10
		13		26	5	37	16			21

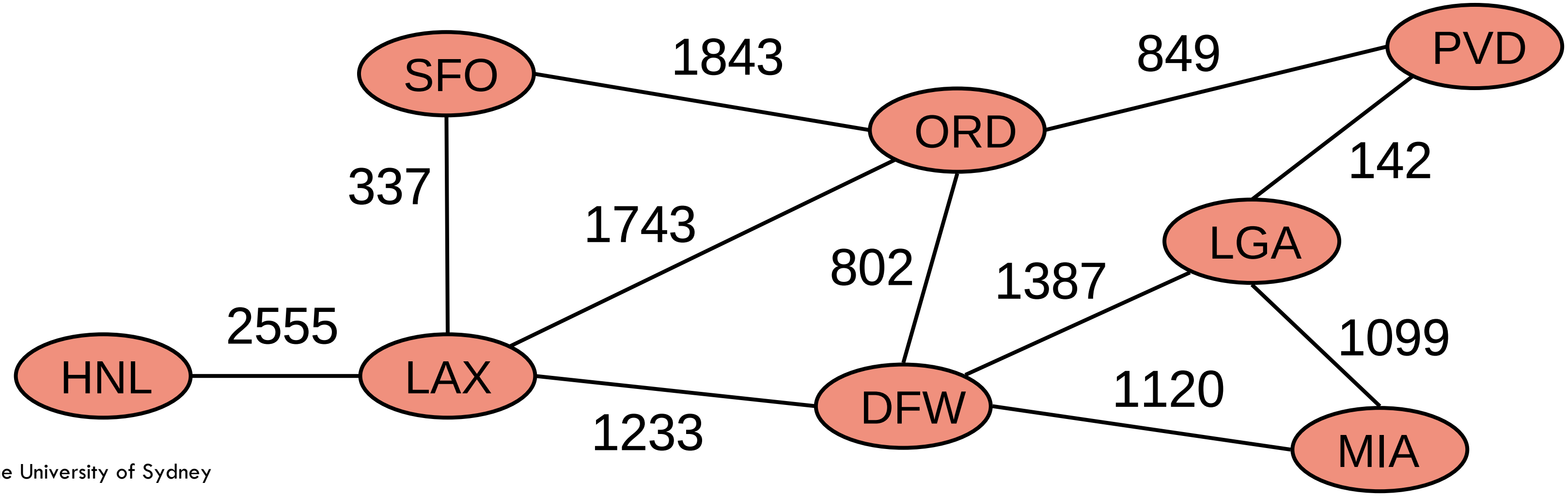
Graphs

A graph **G** is a pair (***V***, ***E***), where

- ***V*** is a set of nodes, called vertices
- ***E*** is a collection of pairs of vertices, called edges

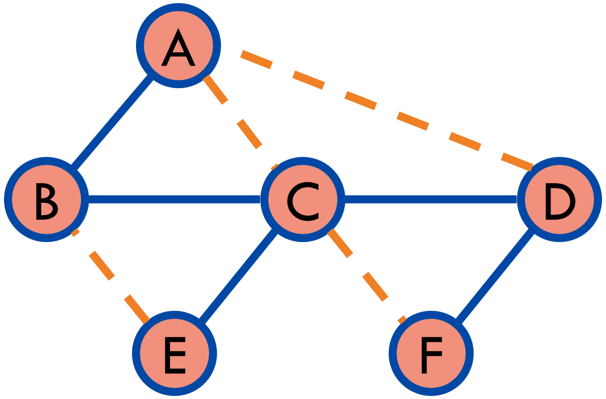
Example:

- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route

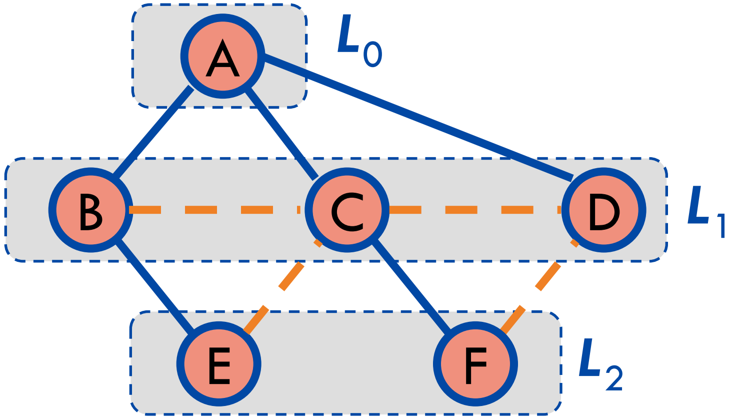


DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Cut edges	√	



DFS



BFS

DFS Visualization (click here)

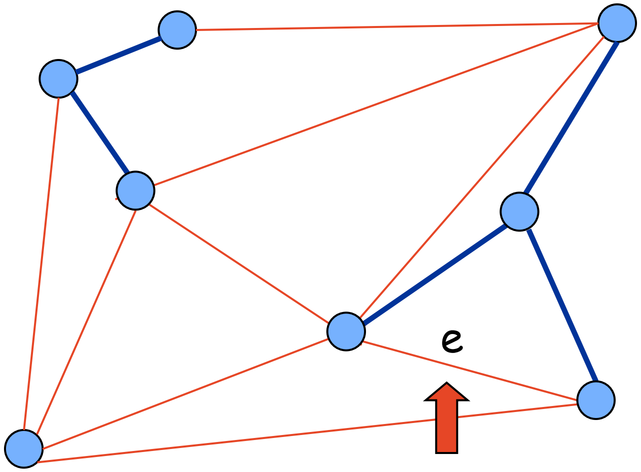
BFS Visualization (click here)

Kruskal's Algorithm

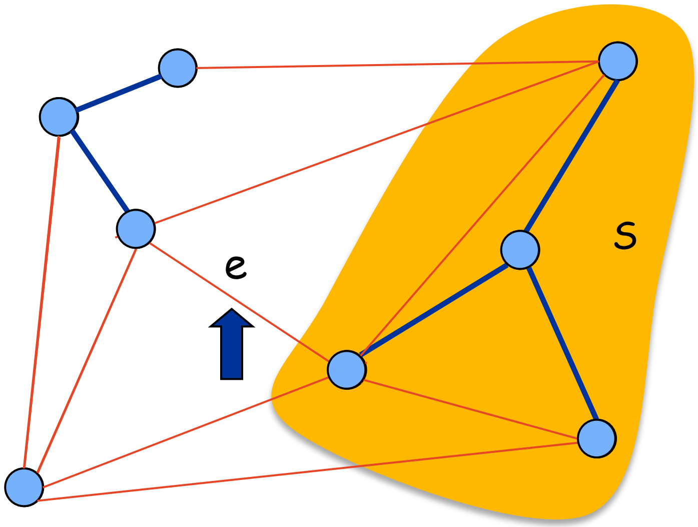
Consider edges in ascending order of weight.

Case 1: If adding e to T creates a cycle, discard e according to cycle property.

Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.

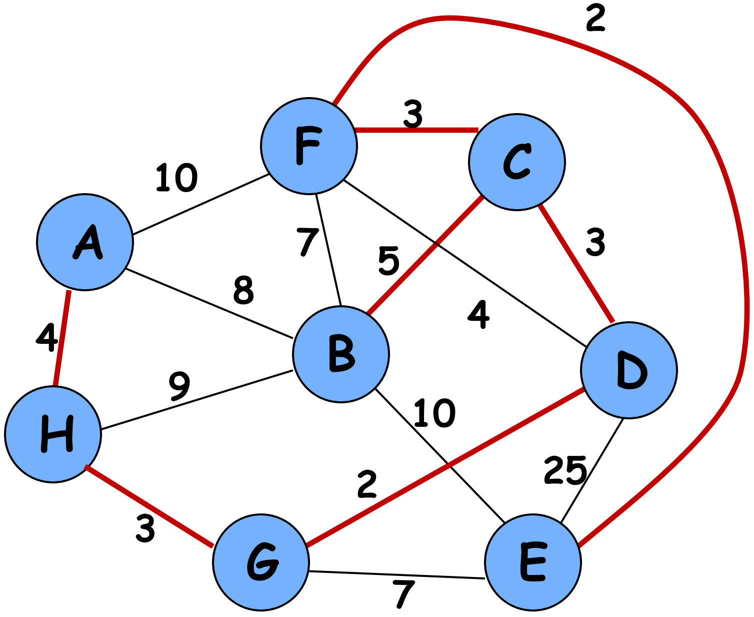


Case 1



Case 2

Walk-Through



What’s “best”?

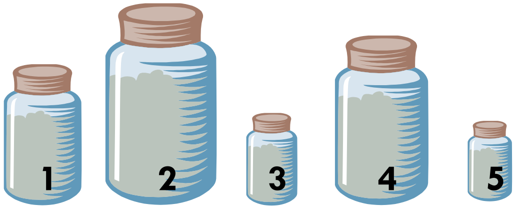
Greedy choice: Keep taking the “best” item.

[benefit/weight]: Select items with highest benefit to weight ratio.

1 ml of 5 → $50
2 ml of 3 → $40
6 ml of 4 → $30
1 ml of 2 → $4

Total value: $124

Items:



“knapsack”

10 ml

Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	$12	$32	$40	$30	$50
Benefit/ml:	3	4	20	5	50

Divide and Conquer

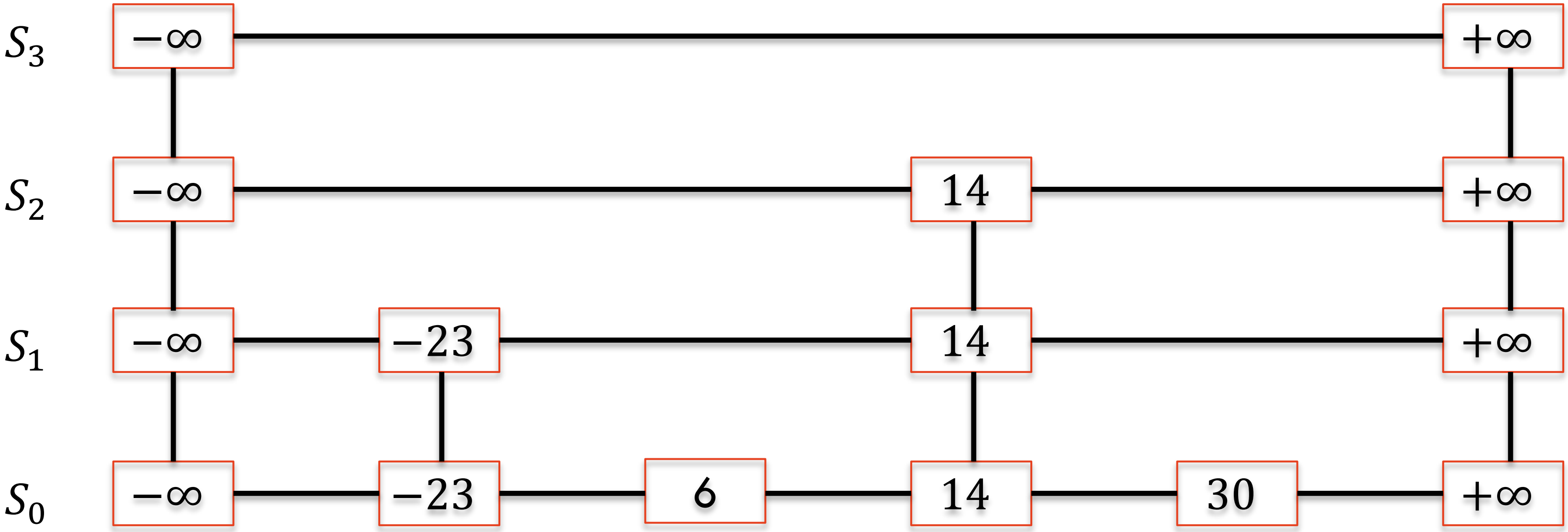
Divide and Conquer algorithms can normally be broken into these three parts:

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.

Skip lists

Leveled structure, where every level is a subset of the one below it.

Next level's elements determined by coin flips.



I CAN
T
DO IT