Misc

- Algorithm = A series of steps that when followed solve an algorithmic problem in finite time (finishes in a finite number of steps)
- Data structure = a systematic way of storing and organising a collection of data
- **Stirling's approximation: $n! \geq (n/e)^n$**
  - Hence, $\log(n!) \geq n \log(n/e)$

Algorithms

Asymptotic analysis: Definitions

- **$f(n) \in O(g(n))$ iff $\exists c, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}_{>n0} [f(n) \leq c \cdot g(n)]$** — g(n) is an asymptotic upper bound for f(n)
  - Typical used for worst case complexity — O for wOrst
- **$f(n) \in \Omega(g(n))$ iff $\exists c, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}_{>n0} [f(n) \geq c \cdot g(n)]$** — g(n) is an asymptotic lower bound for f(n)
  - Typically used for best case complexity — has a line on the bottom, is a lower bound
- **$f(n) \in \Theta(g(n))$ iff $\exists c_0, c_1, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}_{>n0} [c_0 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)]$** — f(n) is asymptotically equal to g(n)
  - Typically used for average case complexity — has a line in the middle, is the average

Asymptotic analysis: Theorems

- Theorem: $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
  Proof: $g(n) \in \Omega(f(n))$ iff $\exists c, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}_{>n0} [g(n) \geq c \cdot f(n)]$
  iff $\exists c, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}>n_0 [f(n) \leq (1/c) \cdot g(n)]$ iff $f(n) \in O(g(n))$
- Theorem: $g(n) \in \Theta(f(n))$ iff $f(n) \in \Theta(g(n))$
  Proof: $g(n) \in \Theta(f(n))$ iff $\exists c_0`, c_1`, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}_{>n0} [c_0` \cdot f(n) \leq g(n) \leq c_1` \cdot f(n)]$
  $f(n) \in \Theta(g(n))$ iff $\exists c_0, c_1, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}_{>n0} [c_0 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)]$
  iff $c_0, c_1, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}_{>n0} [g(n) \leq (1/c_0) \cdot f(n) \wedge (1/c_1) \cdot f(n) \leq g(n)]$ iff $g(n) \in \Theta(f(n))$ [by taking $c_0`=1/c_1$, $c_1`=1/c_0$]
- Theorem: $[f(n) \in O(g(n)) \wedge g(n) \in O(f(n))]$ iff $[g(n) \in \Theta(f(n)) \wedge f(n) \in \Theta(g(n))]$
  Proof: As $g(n) \in \Theta(f(n))$ iff $f(n) \in \Theta(g(n))$, it suffices to show that $[f(n) \in O(g(n)) \wedge g(n) \in O(f(n))]$ iff $f(n) \in \Theta(g(n))$
  $f(n) \in \Theta(g(n))$ iff $\exists c_0, c_1, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}_{>n0} [c_0 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)]$
  $f(n) \in O(g(n))$ iff $\exists c_a, n_a \in \mathbb{R}: \forall n \in \mathbb{N}_{>na} [f(n) \leq c_a \cdot g(n)]$
  $g(n) \in O(f(n))$ iff $\exists c_b, n_b \in \mathbb{R}: \forall n \in \mathbb{N}_{>nb} [g(n) \leq c_b \cdot f(n)]$ iff $\exists c_b, n_0 \in \mathbb{R}: \forall n \in \mathbb{N}_{>n0} [(1/c_b) \cdot g(n) \leq f(n)]$
  Take $c_0=1/c_b$, $c_1=c_a$, $n_0=\max(\{n_a, n_n\})$

Recursion

- Linear recursion = when a function makes one recursive call in the general case e.g. binary search
- Binary recursion = when a function makes two recursive calls in the general case e.g. naive fibonacci
- Multiple recursion = when a function makes more than two recursive calls in the general case

Arrays

Arrays

- **Array = sequenced fixed size collection of variables all of the same type**
- Accessing an array element is O(1) time
- **Creating an array of length n is O(n) time**
- **Adding an element to index i in an array with n cells in use is O(n−i) time as n−i entries** (A[i] through to A[n−1]) **must be moved one space forward** to make room and A[i] must be set
  - Prepending an element is O(n)
  - **Appending an element is O(1)**
- **Removing the element at index i from an array with n cells in use is O(n−i) time as** A[i] must be cleared must be cleared and **n−i−1 entries** ((A[i+1] through to A[n−1]) **must be moved one space backwards** to move the gap to the end of the array
- Resizing an array of size n to be size N is O(n+N) time
  1. Create a new array B, of size N — O(N)
  2. Copy elements of A into B — O(n)
  3. Change references to A to point to B — O(1)
  (4. Deallocate memory of A — O(n))

Lists

Singly linked lists

- **Singly linked list = a series of nodes and a head pointer (a pointer to the first node) where each node stores a data value and a pointer to the next node**
- **Accessing the ith element is O(i)**
- **Inserting an element at position i is O(i)**
  - Prepending (insert at head) an element is O(1)
  - **Appending (insert at tail) an element is O(n) in general case but O(1) if a pointer to tail is stored**
- **Removing ith element is O(i)** (removing tail is O(n) even if a tail pointer exists)

Doubly linked lists

- **Doubly linked list = a series of nodes and a header pointer and a trailer pointer where each node stores a data value and pointers to the previous and next node** (except for header and trailer which only have a next and previous pointer respectively)
- **Header and trailer don't store any data — they only point to the head and tail (the first and last node with data) respectively**
- **Accessing the ith element by index (if a size attribute is kept updated) is min({O(i), O(n−i)})**
  - This is asymptotically equal to the O(i) of the singly linked list as $n=ci$ for some positive real c so $n-i=(c-1)i$
- **Inserting at index i (if a size attribute is kept updated) is min({O(i), O(n−i)})**
- **Removing at index i (if a size attribute is kept updated) is min({O(i), O(n−i)})**

Skip lists

- Sorting a linked list does not improve the bad search time, skip lists address this
- **0th list contains all the elements in the list and special head and tail elements** (acting as minimal and maximal respectively)
- **For 0<i<h, ith list is a random subset of the i−1th list with head and tail both included** — each element has a ½ chance of being included
- **hth list contains only head and tail**
- **To search for an element, start at first element of top (hth) list**
  - **If next element == target, return**
  - **If next element < target, move pointer to next element of current list**
  - **If next element > target, move to the same element in the list below**
- Nodes in each list are quad-nodes — store pointers to next and prev in current list and the element containing the same value in the lists immediately above and below — prev is not really necessary
- **Search and hence insertion and deletion is O(n) in worst case and O(log n) in average case**
- **Requires O(n) space in average case**

Skip lists: Insertion and deletion

- Insert: Using similar algorithm to search, find and store pointers to the largest element in each list that is smaller than what is being inserted and from there carry out standard linked list inserts
  - **Increment a counter every time we get a 0 from round(Math.random()) and add to that many lists starting at bottom — if ≥ h, make new lists**
- To delete: Using similar algorithm to search, find the element before the element to be removed in each skip list and from there carry out standard linked list removals
  - **If we end up with several lists that only contain minimal and maximal, remove the excess ones**

Skip lists: Probabilistic analysis

- Expectation of size of i^{th} sublist = $n/2^i$, expectation of total space = sum from j=0 to inf of $(n/2^j) = 2n \in O(n)$
  - n is a constant and $1/2^i$ is a geometric series
- Search is O(number of dropdowns + number of scan forwards) time in all cases
- Number of drop-downs is O(height)
- Worst case for search is target is only in 0th list (or not present at all) and height is kn (assuming we do not allow height to exceed kn for some k) — O(n+kn)=O(n)
- For a non-negative random variable X, $P(X \geq a) \leq (E(X))/a$, for a binomial variable E(X)=np. Hence, probability the ith list has at least one item $\leq n/2^i$
  - Skip list with n entries has height of at most $k\log_2(n)$ with probability at most $1-n/n^k = 1-1/n^{k-1}$ — height is O(log n) with high probability
- Number of scan forwards is O(log n) with high probability as on average as scan forward by 2 (1/p) per list before dropping down
- Hence, search (and so insertion and deletion) is O(log n + log n)=O(log n) with high probability

Sets

Stacks

Queues

Array implementation

- Store a pointer to the front, store the number of cells in use, and know the size of the array
- Dequeue:

```
ret = A[front]
front++
front %= capacity
size--
return ret
```

- Enqueue:

```
A[(front + size) % capacity] = val
size++
```

Priority queue

- **Behaves like a series of queues where a dequeue dequeues from the highest priority non−empty queue**
  - Can be used to implement a standard queue/stack by giving elements decreasing/increasing priorities respectively
  - Charis uses smaller priorities are "higher" — all this flips
- **Implemented by storing priority value pairs**
- **Can implement using unsorted list**
  - **Enqueue: Add to head** — O(1)
  - **Dequeue: Iterate over entire list once, return most recently (as this will be first in as we are appending to head) encountered value with maximal priority** — O(n)
- **Can implement using sorted list**
  - **Enqueue: Iterate over list until first node with priority less than what is being added is encountered. Insert the new node as predecessor of that node and insert at tail if there was no such node** — O(n)
  - **To dequeue, pop from head** — O(1)

Sorting using priority queues

- Inserting elements into a priority queue with priorities equal to the negation of their values then dequeuing will sort the elements into ascending order
- For unsorted list implementation, $O(n^2)$ as dequeue n times and $n+(n-1)+...+1=O(n^2)$
  - **This is selection sort**
  - Every case is worst case
- For sorted list implementation, $O(n^2)$ as enqueue n times and $1+2+...+n=O(n^2)$
  - **This is insertion sort**
  - Best case is array is in descending order as then adding to head every time
  - Worst case is array is in ascending order as then adding to tail every time

Hashmaps

Hash tables

- An array is created that stores key value pairs and a hash function is used to map keys to indices
- **Hash function is a composition of two functions — first a hash code function is applied then a compression function is applied**
- **Hash code maps keys to some interval of integers**
- **Compression maps the range of the hash code to integers 0 to N−1 where N is the capacity of the array**

Hash code algorithms

- Java's default for Object — Cast memory address of key to integer
  - Equal keys could have distinct memory addresses
- Integer casting — Cast bitstring of key to integer
  - Can cause overflows
- Component sum — Cast chunks (to avoid overflows) of the key to integer then sum
  - Same chunks in different order give same hash (e.g. string with byte chunks hashes anagrams to same)
- Polynomial accumulation — Cast chunks of the key to integer, use as coefficients of a polynomial with degree 1 less than number of chunks, then evaluate polynomial at some (usually prime) x
  - **Horner's algorithm for evaluating an nth-degree polynomial** $f(x) = a_0 + a_1x + \ldots + a_nx^n$ **in O(n) time:**

    $\mathbf{p_0(x) = a_n}$

    $\mathbf{p_i(x) = a_{n-i} + xp_{i-1}(x)}$

    $\mathbf{f(x)=p_n(x)}$

Compression function algorithms

- **Division — h(y) = y mod N where N is capacity of array** — N is usually prime
- **MAD (Multiply, Add, Divide) — h(y) = (ay+b) mod N where N is capacity of array and prime, a and b are naturals, and a mod N ≠ 0**

Collision resolution: Separate chaining

- **Each cell of array stores a linked list**
- On get carry out a linear search of the linked list at that cell for the key value pair with the desired key
- Simple to implement
- Increases memory use
- O(N) as worst case everything collides

Open addressing

- If a collision occurs, place the new value in a different cell
- No additional memory required
- All collisions increase number of steps required whereas in separate chaining only collisions with the current item increase number of steps required
- Linear probing — iterate circularly starting at the value of the hash until you find a free cell
  - O(N) as worst case it is in the cell preceding the cell it hashes to
    - On get, carry out a circular linear search returning null if found an empty cell or have checked every cell
      - On remove cells are set to a special value that indicates that they used to contain a value instead of emptying them
  - E(number of probes for one insertion) = 1/(1−load factor) where load factor = (number of stored values)/(capacity)
    - A probe = getting value of one cell

Graphs

Terminology

- Edges are parallel if they are between the same nodes (and in the same direction if directed)
- An edge is a self loop if it is between a node and itself
- A graph is simple if it contains no self-loops and no parallel edges
- **Degree of a vertex = number of edges incident to it — self loop adds 2**
- **A path is a sequence of vertices and edges $v_1e_1v_2e_2\ldots e_{n-1}v_n$**
- **Simple path = path where no edge and no vertex is repeated**
- **A cycle is a path with $v_n=v_1$**
- **Simple cycle = cycle where no edge is repeated and no vertex apart from v_1 is repeated**
- **Length of a path (and hence a cycle) = number of edges**
- **A spanning subgraph S of G is a subgraph of G with the same vertex set as G**
- A spanning tree is a spanning subgraph that is a tree
- A spanning forest is a spanning subgraph that is a forest

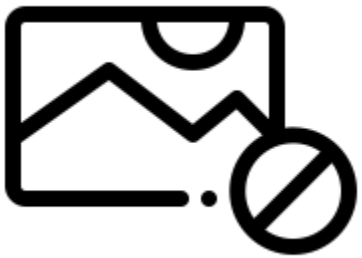
Theorems about graphs

- **Handshaking lemma: For any graph, $\Sigma_{v \in V} \deg(v) = 2|E|$**
  Proof: LHS = number of endpoints of edges. Every edge has two endpoints.
  - Hence, every graph contains an even number of nodes with odd degree
- **Theorem: Let m = number of edges and n = number of vertices. In a simple undirected graph, $m \leq n(n-1)/2$**
  Proof:
  For a complete directed graph with self-loops and without parallel edges we have $E=V^2$ so $m=n^2$.
  There are n self loops in the aforementioned graph. Hence, $m=n^2-n$ in a complete directed simple graph.
  Each edge in this graph has a corresponding edge in the other direction.
  Hence $m=(n^2-n)/2$ in a complete undirected simple graph.
  If not complete m will be lower

Implementations (Charis' versions)

- **Edge list**
  - **Vertex objects only store an element**
  - **Store a list of edge objects (that store an origin vertex object and a destination vertex object** and possibly a weight**)**
- **Adjacency list**
  - **Store a list of vertex objects (that store an element and a list containing all the edge objects that are incident on the vertex))** — store two lists (one for edges it is origin of and one for edges it is destination of) for each vertex for a directed graph
  - **Edge objects store either nothing or only a weight**
- **Adjacency matrix**
  - **2D array**
  - **Each vertex is assigned an integer — indices of column and row of element determine the vertices at the start and end**
    - **Each element is null for no edge, 1 for edge (or weight of edge in weighted graph)**
      - **Can have edge objects that store their weight as the elements of the array instead** if weights will need to be changed

Performance of implementations



Graph traversals: Depth first search

```
depthFirstSearch(G, u)
    return dfs(G, u, new Set())
dfs(G, u, visited)
    visited.add(u)
    for e in G.getEdgesWithSource(u)
        v = e.getDestination()
        if not visited.contains(v)
            return dfs(G, v, visited)
    return visited
```

- **Uses a stack** (implicitly)
- **Returns all nodes that are in the same connected component as u**
  - If e is stored for each node, they form a spanning tree of the connected component (as the search visits every node in the connected component exactly once)
- Time complexity depends on the graph implementation

Graph traversals: Breadth first search

```
breadthFirstSearch(G, start)
    toExplore = new Queue()
    // Explored nodes and nodes that have been queued to be explored
    known = new Set()
    toExplore.enqueue(start)
    known.add(start)
    while not toExplore.isEmpty()
        u = toExplore.dequeue()
        for e in G.getEdgesWithSource(u))
            v = e.getDestination()
            if not known.contains(v)
                toExplore.enqueue(v)
                known.add(v)
    return known;
```

- **Uses a queue**
- **Returns all nodes that are in the same connected component as u**
  - If e is stored for each node, they form a spanning tree of the connected component (as the search visits every node in the connected component exactly once)
    - Any path in the tree will be a shortest (in terms of number of edges used) path between those nodes in the graph — doesn't apply to dfs

Dijkstra's shortest path algorithm

```
for node in current.get_neighbours() start
    if node in visited start
        do nothing
    end
    else start
        potential_cost = current.get_weight_of_edge_to(node) + current.cost
        if node.cost > potential_cost start
            node.cost = potential_cost
            node.previous = current
        end
    end
end
```

- Takes a weighted graph as input
  - Can be directed or undirected but must be weighted
  - Edges that are not possible paths should have a weight of infinity when carrying out by hand and MAXINT when implementing in a programming language
  - Weights don't have to be distances!
- Guaranteed to find the path with the least total weight
- **Inital state is: start node has cost 0 and all other nodes have cost infinity**
  - Cost is not the same as weight
- The bulk of the algorithm only cares about the start point — it is trivial to swap out the destination you want to get to

Dijkstra in words

1. Set all previous to null and all distance to infinity
2. Set start node distance to zero
3. While there are unvisited vertices
    a. Pick the unvisited vertex with the lowest distance
    b. Calculate the *potential_distance* of each neighbor (that isn't already visited) — the distance from the start to the neighbour going through the picked vertex
        i. *Potential_distance = distance of vertex + weight of edge linking vertex to neigbour*
        ii. Update the neighbor's *distance* and *previous* if *potential_distance* is smaller
    c. Mark node as visited

Trees

Misc

- **Level/depth of a node = number of edges on the shortest path from it to the root or equivalently number of ancestors on the shortest path from it to the root**
- **Height of tree = depth of the node with maximum depth**
- **Internal node = a node with at least one child**
- **External node (leaf) = a node with no children**
- Ancestors of a node = parent, grandparent etc.
- Descendants of a node = child, grandchild etc

Binary trees

- Binary tree = a tree in which each node has at most two children
- **Proper/perfect binary tree = a tree in which each internal node has exactly two children**
- **Complete binary tree = a tree in which each internal node either has exactly two children or only has a left child and the left child is an external node**
- Let n be the number of nodes, e be the number of external nodes, i be the number of internal nodes, and h be the height. Then, in a proper binary tree:
  - **e = i + 1** — every time a node gains children, two external nodes are added and one external node becomes an internal node. In base case, e=1, i=0
  - n = i+e — every node is either an internal node or an external node
  - **$e = 2^h$ as there are 2^i nodes at level i**
  - Combinations of the above
  - n = sum from i=0 to i=h of $2^i = 2^{h+1} - 1$

Binary trees: Implementations

- **A binary tree can be implemented as node objects that store their value and pointers to their parent, left child, and right child** — many algorithms do not require parent pointer
- **A binary tree can be implemented as an array of values**
  - **Root has index 0**
  - **Index of left child of node with index p is 2p + 1**
  - **Index of right child of node with index p is 2p + 2**
  - Space required depends on how balanced the tree is, let n be number of nodes
    - Worst case (every internal node only has a right child) — $2^{n+1} - 1 \in O(2^n)$
      - Solving $x_{n+2}=2x_{n+1}+0x_n+2$, $x_0=0$ gives index of $2^{n+1} - 2$, it is necessary to add 1 to find size due to zero indexing
    - Best case — $n \in \Omega(n)$

Binary (min-)heaps

- Complete binary tree that satisfies the heap-order property
  - **Heap-order property: For every node v, key(v) ≥ key(parent(v))** — this does not apply to the root as it doesn't have a parent
    - If the keys are priorities, can be used to implement a priority queue (where minimal priority is considered "highest")
- **A heap storing n keys has height O(log n)**

  As is a complete binary tree, for $0 \leq i \leq h-1$ there are 2^i keys with depth i and there is at least one key with depth h

  $n \geq 2^0 + 2^1 + \ldots + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h$

  $h \leq \log n$
- **A heap storing n keys can be stored using an array of size n** (as is balanced)
- **The last node is the rightmost node with depth of the height of the heap** — if an array is used is the last node in the array

Heap algorithms

- **Insert: insert a new node at the next free index in the array and restore the heap-order property**
  - **upheap restores the heap-order property after an insertion by swapping a node with it's parent if they do not obey the heap ordering property and working its way up — stops once has reached a node that doesn't need swapping**
    - Average and worst case time is O(height)=O(log n)
- **To remove the root, move the last node to the root and restore the heap-order property**
  - **downheap algorithm restores the heap-order property after a deletion by swapping a parent with it's child if they do not obey the heap ordering property and working its way down — stops once has reached a node that doesn't need swapping**
    - Average and worst case time is O(height)=O(log n)
- **Heap sort — Add elements to a heap with key=value then remove root repeatedly, the elements will come off in ascending order**
  - **Requires n inserts (each of which is O(log n)) and n removes (each of which is O(log n))** — O(n log n)

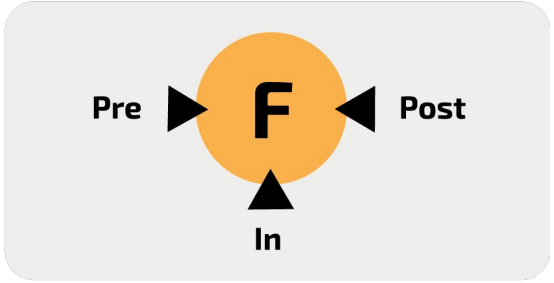
Binary search trees

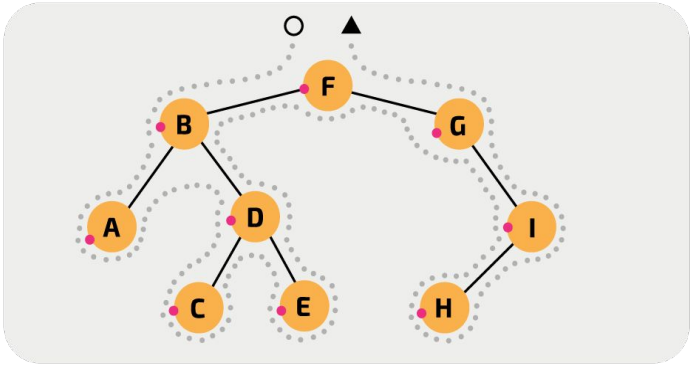
- **A binary search tree is a binary tree that stores key value pairs at internal nodes, stores nothing at external nodes, and obeys the following ordering property:**
  - **For each internal node, keys in left subtree are all < key of node and keys in right subtree are all > key of node**
- **To insert: Search for the key and update the leaf the search reaches to contain the key and give it empty children**
- **To remove:**
  - **Search for the key to find the node**
  - **If the node has one leaf child: Change parent to point to non-leaf child then delete node and leaf child**
  - **If the node has two leaf children: Change parent to point to either one of them then delete the node and the unused leaf**
  - **If both children are internal nodes: Find the first internal node in an in order traversal of the right subtree** (hence finding the node with the smallest key that is greater than the key being removed) **and update the node to be removed to contain its key-value pair then remove the original instance of it instead** (causing recursion until a node with a leaf child is encountered)
- **Search, insert, and remove are all O(h) — h is O(log n) in best case and O(n) in worst case**

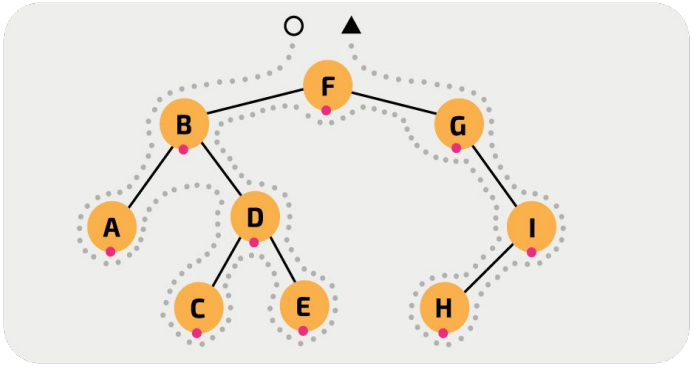
Tree traversals

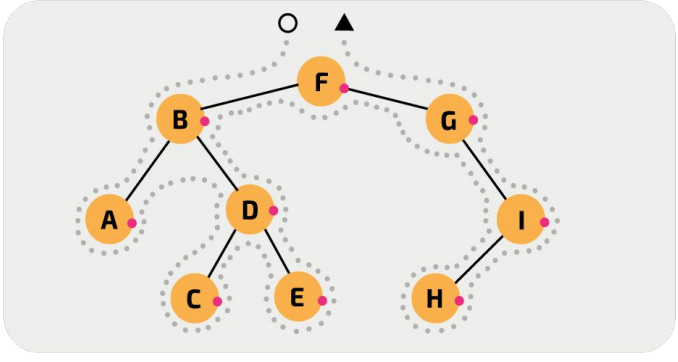
- A pre-order traversal is a depth-first-search that visits a node when it first sees it
- An in-order traversal is a depth-first-search that visits a node when it has visited all of the left-hand subtree and none of the right hand subtree — only applies to binary trees
    - Treesort: Create a binary search tree then do an in-order traversal
- A post-order traversal is a depth-first-search that visits a node when it is seeing it for the last time

Traversal by hand

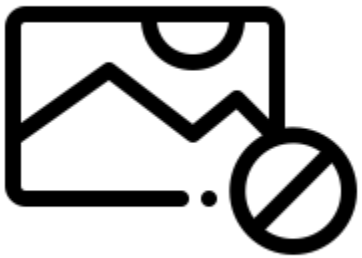








Expression Trees



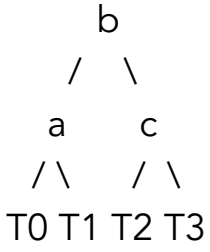
- Inorder traversal with (before left subtree and) after right subtree gives original expression (possibly with some extra brackets that don't change meaning)
- Evaluation: Setting each node from bottom to top to its operator applied to its children (which are numbers) gives a number, number at root is the result
    - As we want bottom to top we use a post order traversal

AVL Trees

- An AVL tree is a self-balancing binary search tree
- **An AVL tree ensures the height balance property is maintained**
  - **The heights of the left and right subtrees differ by at most 1**
- Theorem: The height of an AVL tree storing n keys is O(log n)
- Proof:
  Let n(h) = minimum number of internal nodes in an AVL trees of height h. We will use induction on h.
  Base cases: n(1)=1, n(2)=2
  For n>2, n(h)=1+n(h−1)+n(h−2) (using height balance property)
  We known n(h) is an increasing function, so n(h−1) > n(h−2), hence n(h) > 1 + n(h−2) + n(h−2)
  We know that 2n(h−2) \ geq 0, hence 2n(h−2) +1 > 2n(h−2)
  Hence, n(h) > 2n(h−2)
  Hence, n(h) > 2n(h−2) > 4n(h−4) > 8n(h−6) > … > $2^{h/2-1}$n(h—(2(h/2−1)) = $2^{h/2-1}$n(2) = $2^{h/2}$
  Hence, n(h) > $2^{h/2}$
  So $\log_2$(n(h)) > h/2
  So h < 2$\log_2$(n(h))
  Hence, h is O(log n)
- **To search: Search as normal for a BST**
  - **O(log n) in worst case**
- **To insert: Insert as normal for a BST, then restructure**
- **To remove: Remove as normal for a BST, then restructure**

AVL Trees: Trinode restructuring

- Only works if only one node has been added/removed and the tree was balanced prior to that
- **Let w=node that has been added/removed and let z=deepest unbalanced node on the simple path from w to the root, y=child of z with larger height, x=child of y with larger height**
- **Let a b c be the inorder listing of x y z**
- Let T0, T1, T2, T3 be the inorder listing of the subtrees of x y z that are not rooted at x, y, or z
  - Some subtrees may be empty
- **Replace the subtree rooted at z with a subtree rooted by b containing all the same nodes and obeying the key order property of BSTs**



- **Insertion only requires one restructure but removal may require many**
  - **Insertion: After restructuring after we know the height balance property has been restored**
  - **Removal: After restructuring, check if height balance property is still violated, if it is repeat using new w=parent of current w**, continue in this manner until no longer violated

AVL Trees: Complexity Analysis

- O(n) space
- Single restructure is O(1) time
- Search is O(log n) time in average and worst case
- Insertion is O(log n) time in average and worst case
- Removal is O(log n) time in average and worst case