Programming Language Topics

Object based programming

Organizing around data types: every object in Java has a type, and these types can be either primitive or non-primitive. `char`, `int` types are primitive, whereas any other type is non-primitive. Any type that we define (like `blob.class` would be considered non-primitive, but is defined in terms of primitive types.)

Dynamic vs. Static Types

Static and Dynamic types are two different types in Java. When we write:

```
Type1 varname = new Type2
```

We say that `Type` is the object's static type and `Type2` is the object's dynamic type. The difference is how both these are read in during compile time and runtime.

Static types are only checked during compile time, whereas everything else happens during runtime. When an error would occur in compile time due to casting, then it is most likely going to be an error related to the static types of an object. Some general notes:

- casting is only for compile time, runtime has no knowledge of that.
- casting is only done so that the compiler doesn't scream at you, for having methods that don't exist within the static type of object you tried to cast it to
- the runtime always looks at the dynamic type of the variable (regardless of what you tried to cast it to) and uses the functions within that. If it cannot find that function, it goes up the hierarchy until it finds a function with the exact same signature.
    - Of course, if it cannot find it within its hierarchy, then this would be considered a runtime error.

Inheritance

Inheritance relates to extending different classes, and it's used primarily for scope of variables and functions. Take the following code:

```
class A{
    int f() ...
    int g(int x) ...
}

class B extends A {
    int f() ...
    int g() ...
}
```

- calling f() here would use f() as defined in class B
- calling g() here would use g() as defined in class B
- calling g(3) here would use g() as defined in class A, due to inheritance

Memory Models

Containers

Containers are data structures that contain data, hence their name. Each one has different uses, and each are good for certian applications. Some ones to know are: `array`, `ArrayList`, `HashMap`, `TreeMap`, `Deque`, `Set`, `HashTable`, `List`, and there's a lot more.

Pointers

Pointers are used to reference certain variables to other ones. For instance, in a linked list, the `rest` variable is usually a pointer to the next element in the linked list, and it is this chain of pointers that causes the whole thing to be called a linked list.

Numeric types

Numeric types are specifically the `int`, `double`, and other classes like this that defines numbers. We didn't really go too much into detail here, so it'll be left out for now.

Patterns

Just regex patterns.

- Explicit letters correspond to explicit letters in the regex string
- Square brackets `[]` denote everything within that clasue is considered a character
- `*` denotes 0 or more occurrences
- `+` denotes 1 or more occurrences
- `?` denotes an optional clause (either 0 or 1 times, no more)
- `-` denotes a range (i.e. `a-z` would denote all the letters from `a` to `z`.)

If you want to use these characters in your regex pattern as **characters themselves**, then you need to escape them using `\\`.

Hashing

Hasning means that we organize integers into specific bins based on a hash function $h(x)$ which is usually defined by the user. This hash function is a reasonable hash function as long as there isn't too much collision and the hash function is deterministic.

Random Numbers

Random numbers in java are deterministic, as long as a given seed is provided. The period of this random function is somewhere on the order of 2^{48} so for all intents and purposes during coding, this is random.

But in terms of hashing, if we use a random generator but giving it an input seed of x, then this is deterministic because of how the random funciton works.

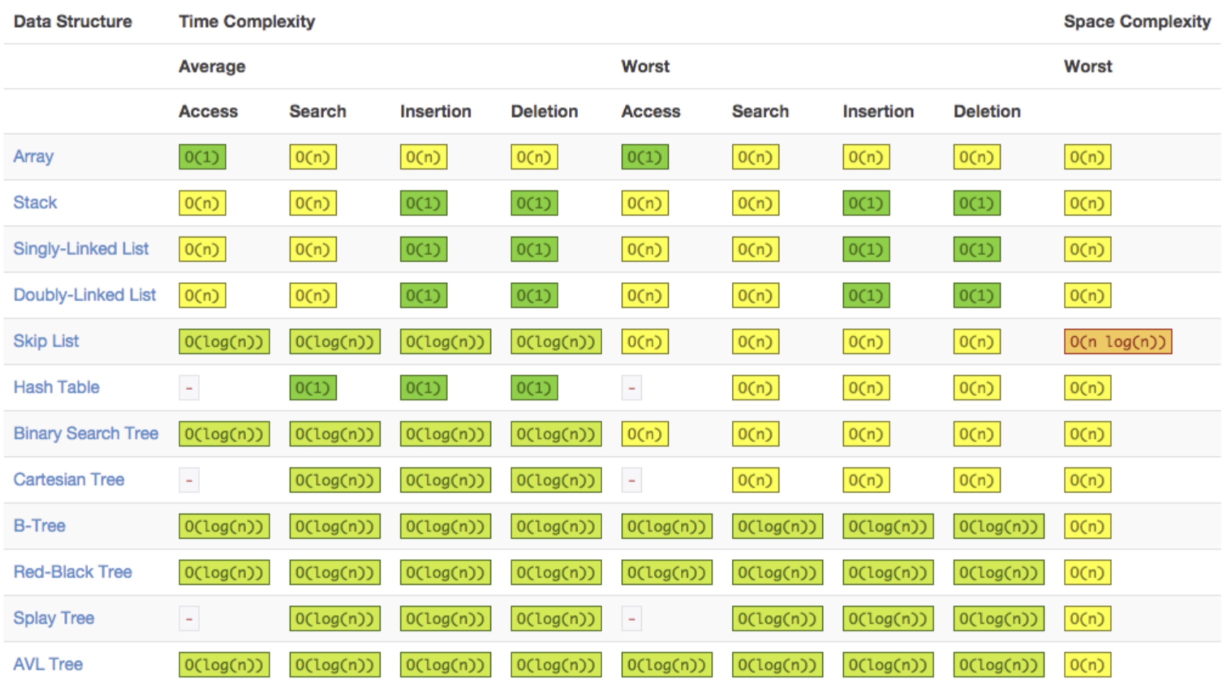
Bits

- `&` (AND) gives 1 only if both bits are 1
- `|` (OR) gives 1 yields 1 if at least one of the bits is 1.
- `^` (XOR) same thing as `|` operator, but only returns 1 if the bits are different.
- `~` (NOT) flips all digits

- `<< x` (LEFT SHIFT) shifts the bit to the left by `x` digits, filling the right with zeroes
- `>> x` (ARITHMETIC RIGHT SHIFT) shifts it by `x` digits, filling the left side with the current signed digit. (the leftmost digit)
- `>>> x` (LOGICAL RIGHT SHIFT) shifts the bits to the right, filling the left side with zeroes.

Masks are generally used in order to provide as a "marker for comparison" against a value that we might want to compare.

Data structures and their runtimes



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Analysis and Algorithmic Techniques

Asymptotic analysis

This refers specifically to answering the question "what happens when we run this program for large input values?" This is usually separated into two categories: best case and worst case scenarios. Some things to keep in mind here:

- If a `break` line exists in code, then in worst case analysis we always assume that this line is *never* reached. In best case analysis, we assume that this line is *always* reached.

Notation

$O(\cdot)$ refers to the *upper bound* limit of a program, meaning that we should expect the program to scale slower than $O(\cdot)$. Likewise, $\Omega(\cdot)$ refers to the lower bound, and $\Theta(\cdot)$ refers to the tightest bound between $\Omega(\cdot)$ and $\Theta(\cdot)$

Heaps

Heaps are data structures which enforce the fact that the top of the heap is the largest value, or smallest in the case of a min-heap.

- Insertion of a heap is $O(\lg N)$ time worst case, and $O(1)$ best case.

- Lookup times in this tree is similar to a binary search tree, since you always have to compare to the value and go down the smaller of the two branches.

Sorting Algorithms

General thing about sorting algorithms: sorts that don't change the relative order of equivalent entries (when compared to the original) are called *stable* if they preserve the general ordering of the elements that aren't being swapped. For this, everything except for quicksort and heapsort are stable.

- Stability is important becuase if we want to apply multiple sorting algorithms, we need to be mindful of which things are being swapped and which things we don't want to be swapped.

Selection sort

Choose the largest element and move it to the end of the array, by swapping it with the element currently at the end of the array. The order of elements that aren't being switched are preserved.

- If you see specific elements being replaced sequentially to the end, then it's likely going to be selection sort.
- When you look at selection sort on the exam, assume that the sort would go by the minimum value

Heapsort

Take your array, put it into a heap structure, and heapify it. Then the preorder, breadth-first traversal of this heap will be the sorted order. Then, we pop elements from the heap as necessary. Keep heapifying the heap as you remove elements.

- If you see any weird structure in the intermediate steps (like the first and the last objects coming together), then it's likely going to be a heapsort.

Merge sort

Take the array and split it into half, and apply mergesort recursively to split them up until you get to its individual units. Then, start comparing them and joining them together, by comparing the elements themselves.

- To join two arrays together, we go through the arrays element by element and append the smaller of the two, while alwyas making that comparison.
- You should see chunks that are being fused together - the indication for mergesort

Insertion sort

Start with the second element, keep swapping it to the element on the left if it is less than (or greater than depending on your sort order) it. Once thsi is done, then move onto the next element.

Quicksort

Select a pivot, and make sure that all elements to the left of the pivot are smaller than the pivot, and the ones to the right are all larger than the pivot. Repeat the same process on each side of the pivot as well.

- The pivot should be relatively obvious. Look for an element where each element to the left of it is less than and every object to the right is more.

LSD Radix sort

Sort by the least significant digit first, then go toward the most significant digit. Going from least to most works here because all the finer details of the number have been worked out by previous iteration cycles.

- For numbers that are tied (like same letter or same digit), then preserve this ordering
- This is one of the easier ones to spot - just look for the ordering of the digits (or letters). The first step of the LSD sort is always very distinctive

MSD Radix sort

Sort by the most significant digit first, then move toward the least significant digit.

- This ones also easy to spot - just look at the leading digit (or letter).

Graphs, Trees

Definitions

- Directed graphs bascially mean that each node is connected by an arrow that poitns one way, as opposed to a connection that specifies no direction.
  - Directed graphs can also only be travelled in a specific way
- A cycle is a path that leads back to itself with no repeated edges.
  - We're only dealing with simple graphs, so self-loops don't exist.
- Directed graphs are said to be *weakly connected* if the graph becomes conected when we remove the direction.
- They're called *semiconnected* if there's a one-directional path that connects all the nodes.
- They're called *strongly connected* if there are paths in both directions from one node to the other.

DAG

- Called a directed acyclic graph - in other words, a graph that contains no cycles and is directed (meaning that connections are arrows).
- Edges can have weights (can be useful to model highways, where we can denote the weight of each edge as the number of cars on a highway)

Implementation of Graphs

- Adjacency List
  - the value of a node can be a data structure (`ArrayList` or `LinkedList`) that stores all the valid adjacent nodes to that node. If none exists, then the list is empty.
- Set of Edges
  - Stored as tuples or some type of object that keeps track of two things `(A, B)`, denoting a connection between node `A` and node `B`.
- Adjacency Matrix

- A literal matrix that is denoted by `1` if two nodes share an edge, and `0` otherwise.
- Note that if you take this matrix and multiply it by its transpose that you get a matrix which tells you how many edges a certain node has, as well as which ones are connected to it.

Graph Traversals: BFS, DFS

They are graph traversal algorithms, that are similar to tree traversal algorithms that were taught earlier in the semester. As a table:

Tree Traversals	Graph Traversals
Preorder, Inorder, Postorder	DFS
Level order	BFS

The difference between trees and graphs is that grphs can have many cycles, meaning that you can start from a single node and come back to it. There may also be multiple paths from node A to B. So when we're doing traversals, we need to mark a node so that we know to not traverse it again.

DFS

Recursive

- Mark the node that you're currently at. For every unmarked adjacent node, do DFS on it.

Iterative

- Initialize a stack `s`, and put the initial node at `s`. While the stack is not empty, pop the vertex from the top of the stack. Then, for every unmarked adjacent node, mark it and put it into the stack.
    - The newly marked node is now at the bottom of the stack, so when it gets popped, the same thing continues again
    - For nodes with multiple branches, you need some kind of tiebreaker rule (usually this would be done alphabetically)
    - Since stacks are FIFO, it means that whichever one is added first gets popped first. This doesn't really create issues since tiebreaks all happen at the same level.

BFS

Just like level-order traversals in trees, visit vertices in the order of the distance to the starting point. This one is relatively self explanatory.

Traversal Algorithms + Union Find

Union Find

Basically an algorithm that allows us to join two sets of numbers (usually denotes as trees) into a single tree.

- The tree initially is arbitrarily determined
- The algorithm involves three methods: `isConnected(x, y)` and `connect(x, y)`

- `isConnected(x, y)` returns a boolean value denoting whether there is a path from node `x` to node `y`.
- `connect(x, y)` connects the two nodes together, by connecting the common root node.
- `find(x)` which returns which set a node belongs t

The actual implementation

We store an `int[]` array of values, with each array index referring to a specific node. The value at that index points to the value of the parent node in that tree. The values of the root nodes are `-1`.

When connecting two sets together (basically doing the connect function) together, we set the array value of one set's root node to the other. During union, we treat the larger set as the children of a smaller set.

Runtime

Worst case to connect everything is $O(N)$ (producing a tree of height N), where N denotes the number of nodes. This occurs when none of the nodes are initially connected together, thus requiring N connections.

Slight modification: Weighted quick union

When we connect two sets together, we first check their size (number of nodes) this time, so that the smaller set is the child of the larger set. Here, the value of the root nodes are stored as `(-1) * size of set`.

Runtime

Here, the worst case runtime would be $O(\log_2 N)$. This is the case because to increase the height by 1, we need more than just a single node in order to increase the height by 1. In fact, we need to bascially make a copy of the tree. You can think of this being the case since adding a single node just casues the node to sit on level 2, whereas doubling guarantees an extra layer being made.

Modification again: Weighted Quick Union with Path Compression

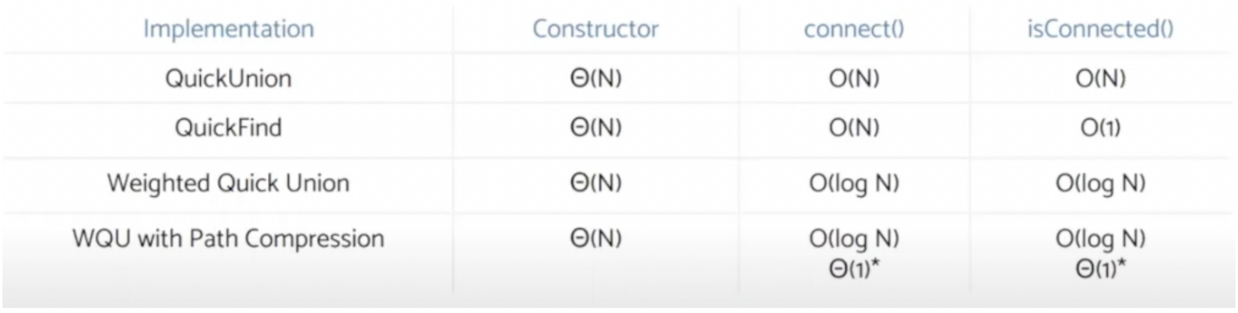
Whenever find is used, compress the path to thd node. Basically what this means is once the path from the root to the selected node is found, we make every node along that path to be a child of the root node. In other words, we connect every node to the root.

Runtime

Basically the same as weighted quick union, but there is a possibility to derive a tigehter bound using amortized runtime, giving us constant runtime.

- Due to a function called an Ackerman function.

Runtime Summary



Implementation	Constructor	connect()	isConnected()
QuickUnion	Θ(N)	O(N)	O(N)
QuickFind	Θ(N)	O(N)	O(1)
Weighted Quick Union	Θ(N)	O(log N)	O(log N)
WQU with Path Compression	Θ(N)	O(log N) Θ(1)*	O(log N) Θ(1)*

Tries

A trie is a type of search tree, usually used for word building or phrase building.

- To check if a word exists, we basically just traverse the tree in the order in the order the letters appear in the word.
    - If the final letter in the word is a marked node, then the word is contained in the trie.
- If a letter doesn't exist, then the word doesn't exist in the trie.
- Adding a word bascially means traversing down the trie until you hit a new letter, then start adding letters that branch off from that node.

Runtime

Adding a word is $\Theta(L)$ and searching for a word is $O(L)$, where L denotes the length of the word. These are the worst cases since the most computations you ever need to do is if you have to do a computation for every letter of the word, which is done L times.

- Note that there is no "best case" searching for a word since it still requires L computations to find the word letter by letter.

Topological Sort

Given a DAG (directed acyclic graph), "flatten" it in a way that it is consistent is edges. (i.e. find an ordering of the nodes that preserves its edges.)

Reverse DFS

- Reverse the direction of all edges, then perform a normal DFS starting from any node. The deepest node will be your starting point, then work your way up in the order of DFS in order to get your sort.
    - We then store this order as a list of postorder traversals (this is what DFS means)
    - If you then want to find the path to another node, perform a DFS starting at that node, and once you hit a node that you've already encountered (stored in the list), then you can stop there, then append that to the list.

Kahn's Algorithm

- Count the parents of all nodes. Keep adding nodes that have 0 parents, and as you add them update the number of parents in the children nodes to decrease by one.
- Repeat this algorithm until the entire graph is added to the list.
- The result of this is a topological sort

Runtime

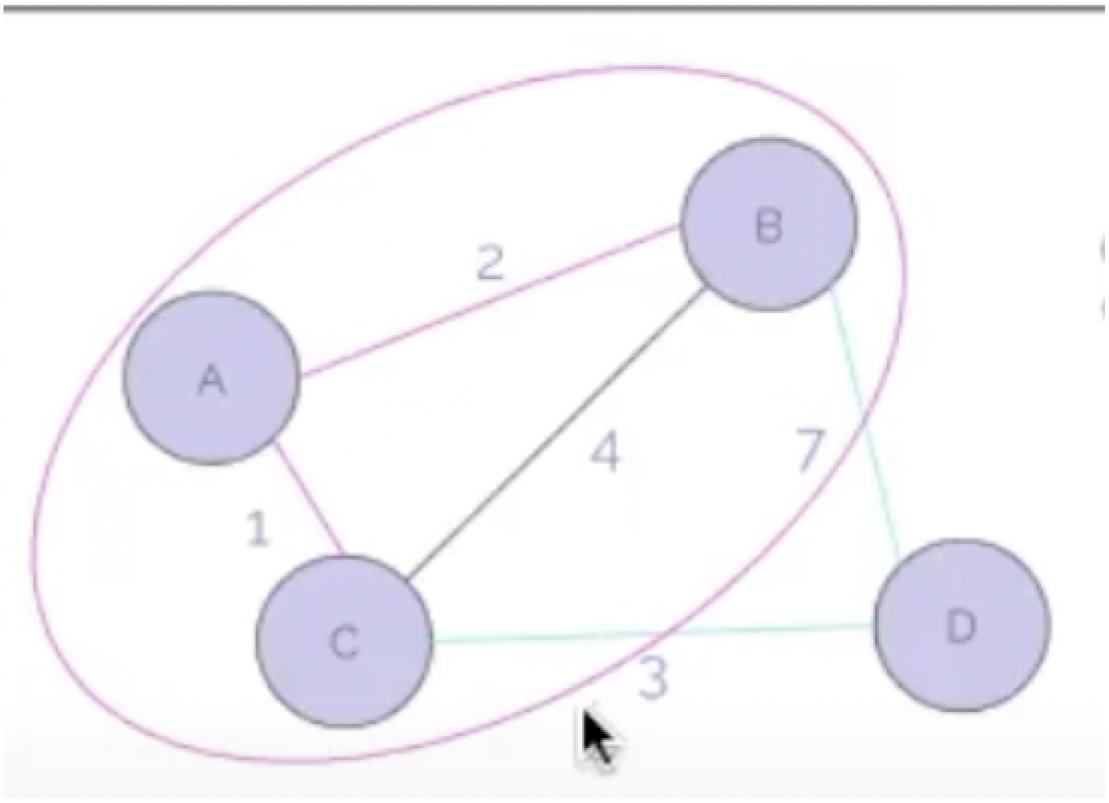
- Both algorithms here have runtime $\Theta(|E| + |V|)$ time.

- If no topological sort exists, then this means that the graph has a directed cycle
- A unique topological sort is equivalent to saying that the graph has a hamiltonian path

Prim's Algorithm

Cut Property

Cuts are edges that separate a collection of nodes from the rest of the graph.



Here, the cut that would separate the nodes A, B, C from the rest of the graphs would be (CD, BD).

The **cut property** refers to the fact that the lightest edge among the ones that are cut is in some MST (minimaum spanning tree). This makes no restriction on the largest edge in the MST.

Prim's algorithm works in the following way: First, we start with a single node and make a cut across it. (in this case it just means choose the edge with the smallest weight) Choose the smallest edge that is cut, then add the node on the other end to the tree. Repeat until all nodes have been added. The edges that were stored are the MST.

- The implementation of this is done as a priority queue, where edges are sequentially added to the priority queue and dequeud when we need to add an edge.
  - Here, the priority is reversed so that the *highest* priority is given to the edge with the *lowest* weight.
- These edges are called *fringes* in Prim's algorithm.

Kruskal's Algorithm

Alternative to Prim's algorithm to find MSTs. Kruskal's builds the MST edge by edge instead of approaching it vertex by vertex.

- It considers the edge one by one, from smallest to largest.

- We omit edges that create cycles (since that would violate the condition of an MST), and keep going until $V-1$ number of edges is selected.
    - checks whether two vertices are already connected (via a union-find method)

Manually running the algorithm

Start with the edges that have the minimum weight, regardless of where they are on the graph. If there is a tie, it really doesn't matter which one you choose. If the edges connect two vertices that aren't already connected (via ANY alternate route), connect them.

- Usually it's better to draw it out on a separate piece of graph paper.

Dijkstra's Algorithm

An algorithm that finds the shortest path between two nodes. The algorithm is run as follows:

- First add all vertices (aka nodes) to a fringe priority queue, that is ordered by all the smallest distance values.
- Pop off the smallest vertex value, and all other vertex values are initially set at ∞.
- If your current distance + the distance required to travel to that node is less than the node's distance value, then update values as necessary.
    - Updating means changing the dist value of the node that you just added to the sum of what you just added, then adding the back value of the new node to be the node you just came from.
    - Also update the nodal values in the fringe.
    - Might also need to retroactively update values once new paths are found. In this case, delete the other path that you previously created.
- rinse repeat until you have created a shortest paths tree.

Runtime

- Removing nodes from fringe: $\Theta(V \cdot \log V)$.
- Updating priorirites, changing neighboring values: $\Theta(E \cdot \log V)$.
- Total runtime: $\Theta((V+E)\log V)$.

A* Algorithm

Dijkstra's algorithm can be costly especially in large use cases since it basically explores all the possible routes from point A to B. A* Algorithm makes an improvement on this by limiting the direciton that you can move in to a single one.

A* Algorithm includes an extra step onto Dijkstra's algorithm, so that when we're comparing we also add a rough estimate of how far the current node is from the target node.

- This restricts movement to a single direction since the distance will always be larger if we're moving away.
- The estimated values will be gien to you; otherwise there's no way of knowing.

Heuristics

Basically asking how we actually make these estimates. There really isn't any good way to do this, but we should always follow the guideline that the estimate is less than the true value. If your heuristic value is zero, then you follow Dijkstra's algorithm.

- For each neighbouring node, you should expect that the heuristic value drops as you get closer to the target node.

Balanced Search Structures

B-Trees

Binary search trees are sometimes good, depending on the order in which elements are being added into them. Access time would be $\Theta(n)$ for worst cases, and $\Theta(\log n)$ for average cases.

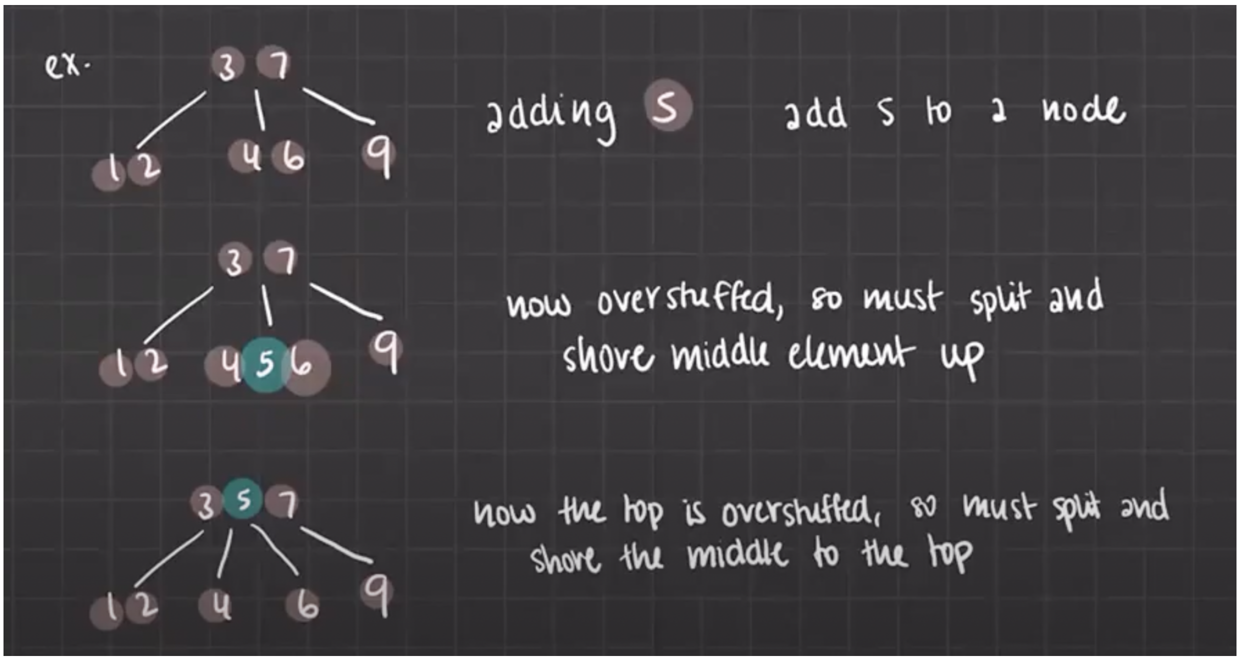
B trees work exactly like a BST tree, but for every node, we have L number of elements and $L+1$ children per node.

- Generally, we use $L = 2$ for this (2 items per node, 3 children per node)

Addition

Add a leaf to the node, and try to add it as far down as possible. If the node is filled, then push the middle one up to the parent, and split the leftover into two differnet nodes.

- Keep repeating this until all constraints are matched.


ex.
3 7
1 2
4 6
9
adding 5
add 5 to a node
now overstuffed, so must split and shove middle element up
3 5 7
4 5 6
now the top is overstuffed, so must split and shove the middle to the top

This is difficult to code, so now we use Red-Black trees as a substitute. Basically, for every node that's full, we take the higher number, and raise it. Then we color the smaller one red.

If we were to represent this as a general tree, the red and black would be on different levels.

(A, B)-trees

Here, the A denotes the minimum number of children that the tree can have, and B denotes the largest number of children that a given root node can have.

- in the example exam question, only black nodes are counted, for some reason.