Main ideas

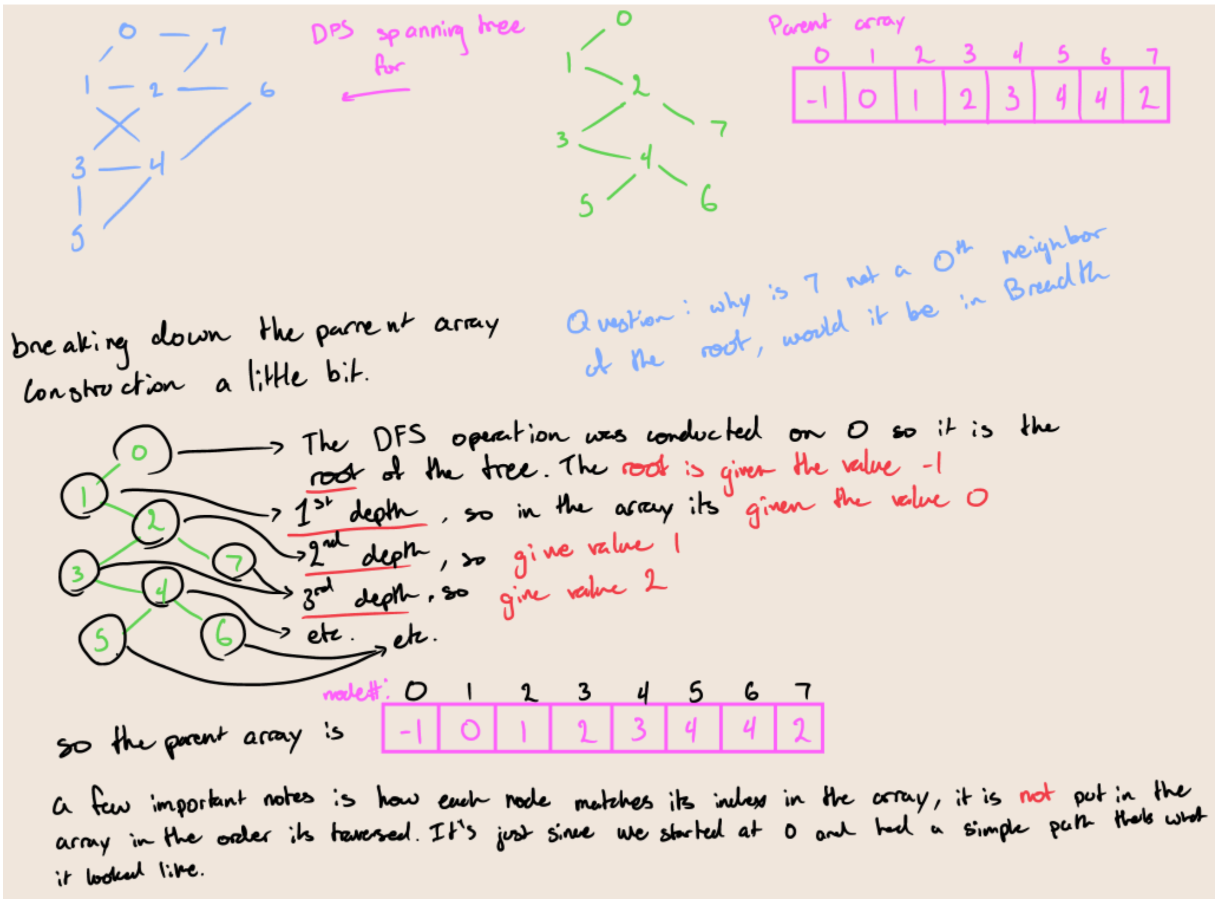
- Many questions will be mechanics of stuff
    - i.e. learn how to do stuff not just what it is
- Go through exam 1 and 2 as well as quizzes
- Question on formally defined big O
    - The mathematical definition. The answer to this is on one of the quizzes

Graphs Chapter

1. How many ways a graph can be stored?
    - Adjacency List
    - Adjacency Matrix
        - C++ definition of a matrix
        - Compressed sparse row
        - Row-major ordering
        - Column-major ordering
        - Sparse matrix representation arrays of arrays
        - Arrays of arrays
        - Upper triangular can be used with an undirected graph so less data has to be used since the upper and lower triangles are symmetrical
            - This is literally just collecting the top triangle instead of the whole matrix to limit space used. For smaller graphs it doesn't make a lot of since, but as graphs grow it becomes more and more useful
    - In class he mentioned these as well, but generally the above implementations are more reliable
        - degree (valency) of the graph
            - Degree is simply the number of nodes a graph has
            - This isn't a bad way of storage as it is a very small space complexity, but to traverse it a matrix needs to be made which is $O(n * n)$ complexity anyways, and the same space complexity if not more so its not really viable
            - But for the sake of compression and decompression it is good
            - Along with that its the one he said in class so its good to keep it in mind, but its like basically adjacency matrix
        - does edge exist
            - This is also basically just an adjacency matrix since its also just showing

    edge relationships
  - see all neighbors
    - This is basically adjacency list as it just shows all the neighbors of each node

2. Compare and contrast each of the ways in terms of storage and time to perform operations
   - Adjacency list and matrix(O(n^2)) are both very good ways of storage with their own advantages and disadvantages
   - The space complexity of a list is $O(n+m)$, where n is nodes and m is edges, where the space complexity of a graph is $O(n*n)$
   - The other significant difference between adjacency lists and adjacency matrices is in the efficiency of the operations they perform.
     - In an adjacency list, the neighbors of each vertex may be listed efficiently, in time proportional to the degree of the vertex.
       - In an adjacency matrix, this operation takes time proportional to the number of vertices in the graph, which may be significantly higher than the degree.
     - On the other hand, the adjacency matrix allows testing whether two vertices are adjacent to each other in constant time (O(n)); the adjacency list is slower to support this operation.
3. Algorithms for DFS and BFS trees?
   - Both algorithms are essentially a loop that goes through the graph visiting each neighbor along the way
   - DFS uses a stack to do this, when the end of a path is reached the stack is used to backtrack by popping until another path is revealed
   - The BFS uses a very similar procedure but by instead using a queue, this causes a search that "asynchronously" searches all possible paths without backtracking
     - I use quotes because its not happening at the same time, but just appears to as nodes from all paths are added to the same queue to be explored at the same time
   - Here are the algorithms for either operation can be found here
     - Chapter 11 - Graphs
4. Given a graph perform operations on the graph like finding the DFS and BFS tree
   - Just traverse the graph either adding to a stack then backtracking or by adding to a queue
   - An important note is that the nodes are just kinda picked when deciding between multiple neighbors, the choice would be specified by the algorithm, but it was never specified by the professor so just don't worry about the which needs to be chosen

5. What are applications of DFS and BFS trees?
    - DFS
        - Construct spanning tree
        - Can detect connection of graph
        - Can locate the number of connected components
        - Can detect a cycle in the graph
    - BFS
        - The breadth first search is useful in determining the shortest path from a given vertex to all the vertices of an unweighted graph.
6. How are the trees stored in a parent array data structure
    - 

0	1	2	3	4	5	6	7
-1	0	1	2	3	4	4	2

breaking down the parent array construction a little bit.

Question: why is 7 not a 0th neighbor of the root, would it be in Breadth

The DFS operation was conducted on 0 so it is the root of the tree. The root is given the value -1

1st depth, so in the array its given the value 0

2nd depth, so give value 1

3rd depth, so give value 2

etc. etc.

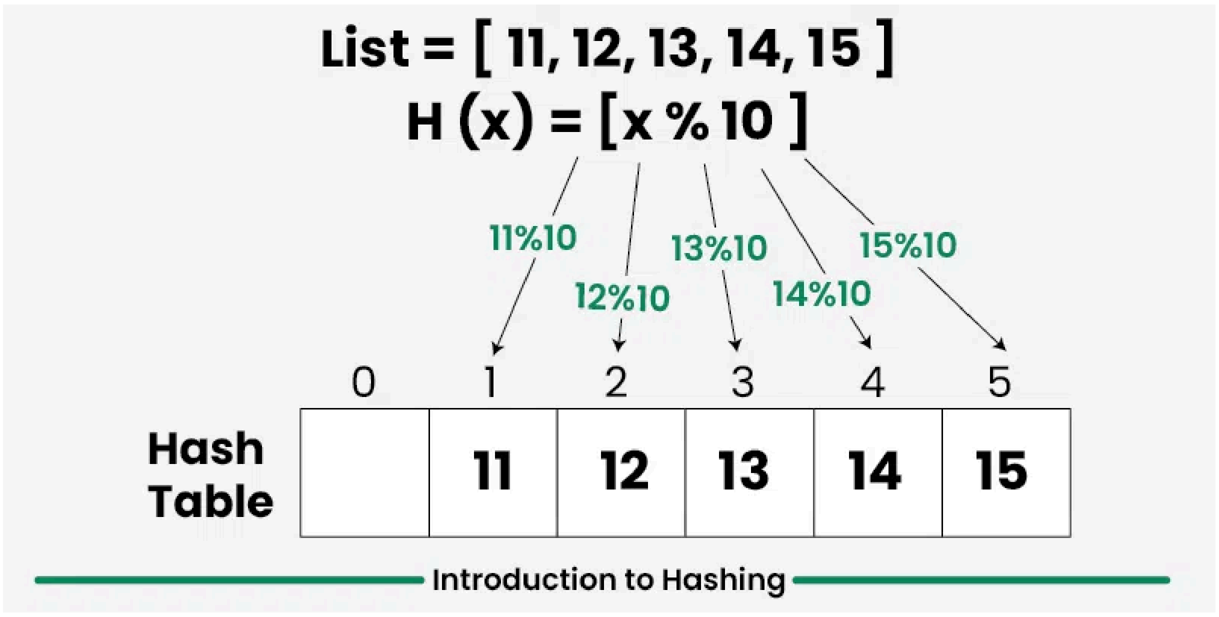
so the parent array is

node#:	0	1	2	3	4	5	6	7
	-1	0	1	2	3	4	4	2

a few important notes is how each node matches its index in the array, it is not put in the array in the order its traversed. It's just since we started at 0 and had a simple path thats what it looked like.

Hashing

1. Given a set of values and a hash function, give the values after the hash function is computed
    - Quite simple, the element of the set is can simple be considered as the x value in the function, execute the hashing function and the output is the hashed value, there are then collision which will be talked about next
    - example of hash function:

- 

2. What are the ways collision is handled
    - There are many different methods for avoiding collisions in a hash table
    - The first of which is simply omitting the colliding element, this is obviously not very good iv every element needs to be in the table, but in certain use cases it is efficient and practical
    - The next method is *chained hashing*, there are 2 approaches to chaining: separate chaining and coalesced chaining
        - First up is sperate chaining
            - Each slot of the hash table act as the head of a linked list (chain) when an element collides at the slot it is simply added to the chain
            - This chain is usually just kept unordered and accessed linearly, but if the user want to sort it by some frequency to increase access efficiency that is also possible
        - Next is coalesced chaining
            - This is very similar to separate chaining with the difference that all the elements are stored in the array and no through chains
            - When a collision occurs the first open position from the back of the array is found and the spot with the original collision stores a value of that new index
                - in illustrations it often shows arrows instead of values, but its the same thing
            - If a collision has already occurred on a given index it is traced to the new index which will then point to the next, this process can go on for the entire array if necessary, it would be super inefficient, but it would work
    - The most common method is probing. There are 2 main types of probing, *linear probing, quadratic probing, and double hashing*. Both of them work basically the

same (explained later)

3. What is a open addressing scheme - Linear Probing an quadratic probing
    - **Linear probing** is simply doing the hash function once then if there's a collision the table is iterated until an empty slot is found
    - **Quadratic probing** is very similar to linear probing but instead of just incrementing by 1 its increments by i^2 where i is the iteration(so by 1, then 4 then 9 then 16 and so on)
    - **Double hashing** is simply using 2 different hash functions to hash a value using the following formula:
        - $(hash1(key) + i * hash2(key)) \ \% \ tableSize$
        - With this formula hash1 is hashing the function and hash2 is the hash that determines how it is incremented, while i is the iteration, so after every increment it is increased by 1 changing its position
4. What is a perfect hash function?
    - A "perfect hash function" is a hash function that maps a set of distinct keys to a range of integers with **absolutely no collisions**

Sorting

1. Quick sort - Partitioning based on pivot element - Algorithm and how to do this?
    - quick sort is a type of comparison functions which operates by choosing a pivot element then splitting all of the lower and higher elements in respect to the pivot, the pivot is then moved to the middle, then each half is partitioned and quicksort is operated on each subgroup until the list is sorted
    - **Worst case time complexity:** $O(n^2)$
    - **Best case time complexity:** $O(nlogn)$
    - **Space complexity**: $O(log(n))$
    - There is also the important topics of selecting the pivot point as this is an important choice in maintaining efficiency of the function
        - There is the simple choice of just selecting the first last middle or even a random value, this simply picks a pivot point without any care of efficiency, if unlucky this could pick a pivot point that makes the algorithm have extremely poor efficiency, but generally its fine as its just O(1)
        - There is also the option of using 'Median of three' (Mo3) which is simply taking the median of the first middle and last element, this gives a higher chance of getting a more optimal pivot point, while still not guaranteed it is a better chance
            - Mo3 does have the issue of requiring some computation to pick the pivot, and for large data sets this can be kinda intense. So even though it does give a better pivot it should be used with caution

2. Merge sort - Algorithm, show the splitting and merging
    - Merge sort is a comparison based algorithm that works by breaking the list into n subsets then sorting them as they merge, the sorted sets then continue to be combined until the full sorted sets is created
    - **Best worst and average case time complexity:** O(nlogn)
    - **Space complexity**: $O(n)$
3. Heap sort - complexity and algorithm
    - Heap sort is pretty simple it is simply creating a min or max heap (talked about later), then talking the top element of the tree and putting it in the front or the back of the array (front for min, back for max)
    - **Best worst and average case time complexity:** O(nlogn)
    - **Space complexity**: $O(1)$
4. Straight selection sort - Complexity and algorithm
    - Selection sort is also a comparison based algorithm that uses a sort of imaginary subarray at the beginning of the array that contains all the sorted elements
    - When completing the sort the lowest element is added to this pseudo array, the array continues to be traversed until all elements are in the sorted portion leaving the array fully sorted
    - **The time complexity for all cases:** $O(n^2)$
        - So its good for smaller less complex data sets but is worse for more complicated sets
    - **Space complexity:** $O(1)$
5. Insertion sort - Complexity and algorithm
    - Insertion sort is also a comparison based algorithm that takes value "out" of the array then compares the value to the rest of the array until it finds its spot then it *inserts itself*
    - This algorithm also has the imaginary array going on but its the unsorted part instead of the sorted part.
    - **Best case:** $O(n)$
    - **Worst and average case:** $O(n^2)$
    - **Space complexity**: $O(1)$
6. Simple bucket sort - complexity and problems with it
    - Simple bucket is a distribution sort, but it can also be a comparison sort based on how the buckets are sorted.
    - Simple bucket sort works by creating a bucket system for the array to be split up into, the elements of the array are then scattered into their bucket. Each bucket is then sorted before it is combined back together.

- The actual computational complexity is widely dependent on the sorting algorithm used to sort each bucket, but the complexities are as follows
- **Best case and average case:** $O(n+k)$
  - k is the number of buckets made
- **Worst:** $O(n^2)$
- **Space complexity**: $O(n)$
  - Since it has to create the buckets

7. Known the worst case and best case complexities of the algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	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(n)
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)	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)

8. Worst case and best case inputs of the algorithm
   - He never really talked about this, but generally some algorithms are better with small sets other larger sets. And along with that some sets have the worst case when its reversed while other algorithm's worsts case is something else
9. A balanced binary tree such as AVL and red-black trees can be used for sorting. How and complexity?
   - $O(nlogn)$ for insertion, then InOrder traversal to get sorted list

Priority search tree

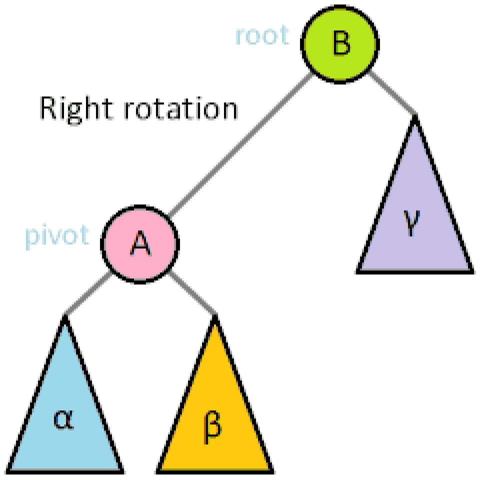
1. What are the operations that can be performed on a heap data structure? complexity of these operations?
   - **getMin()**: returns root of a min/max heap, **Time complexity is** $O(1)$

- **extractMin/Max()**: Removes the minimum/maximum element from the min/max heap, after removal heapify() is called to maintain the heap, so the **time complexity is** $O(logn)$
- **decrease/increaseKey()**: Decreases/Increases the value of the key, this is also a **time complexity of** $O(logn)$ since if the decrease/increase of the key breaks the rules of the heap then the heap will need to be traversed to fix it
- **insert()**: insert a new node into the tree; a node is added to the end of the heap, if it doesn't break any rules everything is fine, but other wise the heap will have to be traversed to fix the problem, so the **time complexity is** $O(logn)$
- **delete()**: Deletion occurs by calling increase/decreaseKey() until the node is at the root of the heap then extractMin/Max() is called to remove the node, so the **time complexity is** $O(logn)$

2. Define min or Max heap completely: structure (Complete binary tree) and ordering properly
    - A min and max heap are both complete binary trees where the min or max value is the root of the tree
        - A complete binary tree is a tree filled from left to right, top to bottom. Every layer of the tree must be full besides the last layer which still must be filled from left to right
3. Given a set of numbers, construct a min-heap, and max-heap, and a min-max-heap
    - just add to the tree from left to right, if there is an error in the properties of the respective tree then performs swaps to maintain order
4. Why is a min-max-heap better than a min-heap or max-heap or a balanced binary tree such as AVL
    - it allows $O(1)$ access to both the min and max elements where the other two heaps only have $O(n)$ for their non-specialty , and balanced trees require $O(logn)$ to find the min or max in a data set
5. Given a heap; (min or max), perform operations in a sequence: delete min, delete max, insert, and so on
    - Ways to do this are already described in the operations sections (question 1)
6. How to construct a min-heap from a binary search tree? complexity
    - Create an array of size n, where n is the number of nodes in the given binary search tree
    - Perform the **InOrder traversal** to get the sorted list and copy it to the array
    - Then preform the **preOrder traversal** of the tree. While traversing copy the values from the array to the tree
        - So when the root is visited take that index from the array and set that node to it (that makes no sense)

- Basically the list is sorted from least to greatest so when traversing the tree in preOrder the node is overridden with the sorted value leaving a tree that is sorted by min order

Self-adjusting binary trees

1. C++ code for zig and zag operations
   - A **zig rotation** is a clockwise rotation of a tree or a **Left rotation**
   - A **zag rotation** is a counter clockwise rotation of a tree or a **right rotation**
   - 

   - Here is the code for the **zig operation**

```
// Zig operation: Perform a right rotation
TreeNode* zig(TreeNode* x) {
    TreeNode* p = x->parent;

    if (!p) return x; // If no parent, can't perform zig

    TreeNode* g = p->parent; // Grandparent
    p->left = x->right;
    if (x->right) x->right->parent = p;

    x->right = p;
    p->parent = x;

    x->parent = g;
    if (g) {
        if (g->left == p) g->left = x;
        else g->right = x;
    }
```

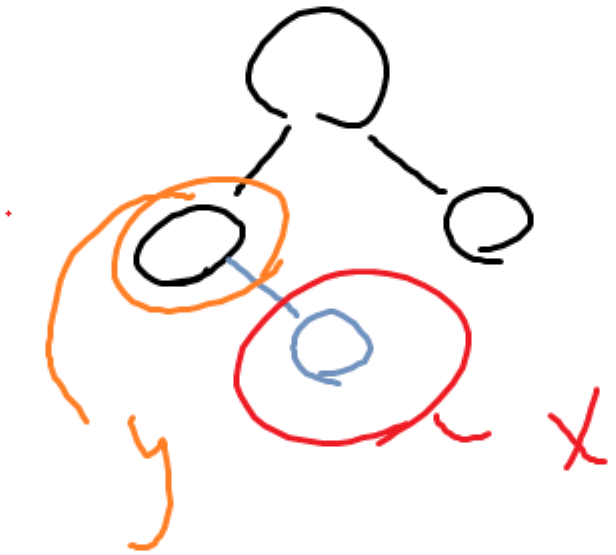
```
        return x; // Return new root of the rotated subtree
}
```

- and the zag operation

```
// Zag operation: Perform a left rotation
TreeNode* zag(TreeNode* x) {
    TreeNode* p = x->parent;

    if (!p) return x; // If no parent, can't perform zag

    TreeNode* g = p->parent; // Grandparent
    p->right = x->left;
    if (x->left) x->left->parent = p;

    x->left = p;
    p->parent = x;

    x->parent = g;
    if (g) {
        if (g->left == p) g->left = x;
        else g->right = x;
    }

    return x; // Return new root of the rotated subtree
}
```

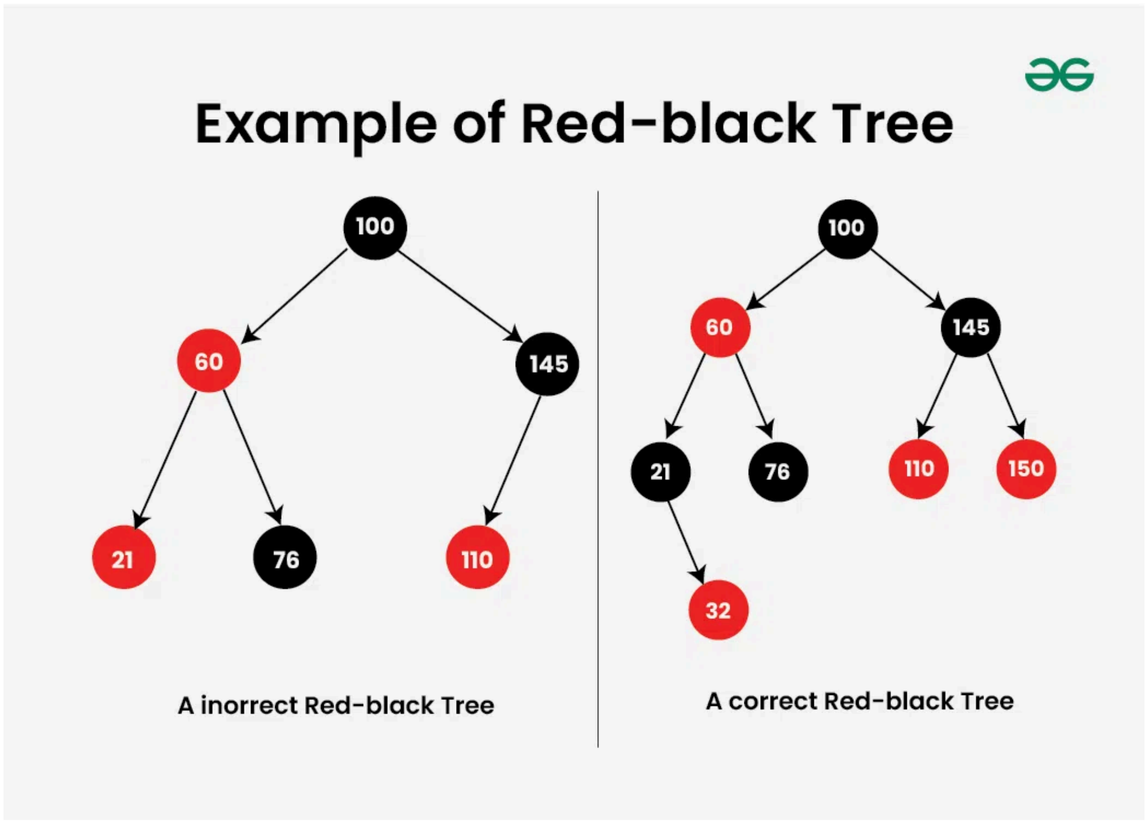
- This was made by chat-gpt so take it with a grain of salt, but its probably right

2. Define what a splay tree is. Why is a splay tree useful? Given a tree perform a splaying node in the tree
    - Splay tree definition (directly from notes):
        - It is a BST
        - Find(x)
            - If x is a found then bring x to the root of the tree with rotations
            - If x is not found then bring node y (node y is the parent of the location of where node x would be if it existed) to the root of the tree



-
  - x is the node that doesn't exist and y is the parent of the nonexistent node
- Insert(x)
  - Insert x and bring it to the root of the tree with rotations
- Remove(x)
  - Remove x and bring parent of x to the root of the tree with rotations
- A splay tree is a binary search tree that uses rotations to make recently accessed elements quicker to access again through the process of splaying, which is the main reason it's useful
- Along with that they can still be balanced, but more than likely the splay tree will not be balanced as its more concerned about **enabling easier access to specific nodes than making even access to all nodes**
  - So the worst case for operations is $O(n)$, but the best is still $O(logn)$ like other trees
- The primary operation of a splay tree is splaying, which is just using a series of rotations to bring the desired node to the root
  - There are two types of splaying the first is **bottom up splaying**:
    - Bottom up splaying is done by first finding the target node through a search, then rotations are done to bring the target node to the root of the tree
    - The main notes about bottom up splaying is that it requires a double pass (searching for node then applying rotations). Along with that is is slightly better than the alternative because it has a slightly simpler application
  - The next type is top down splaying:

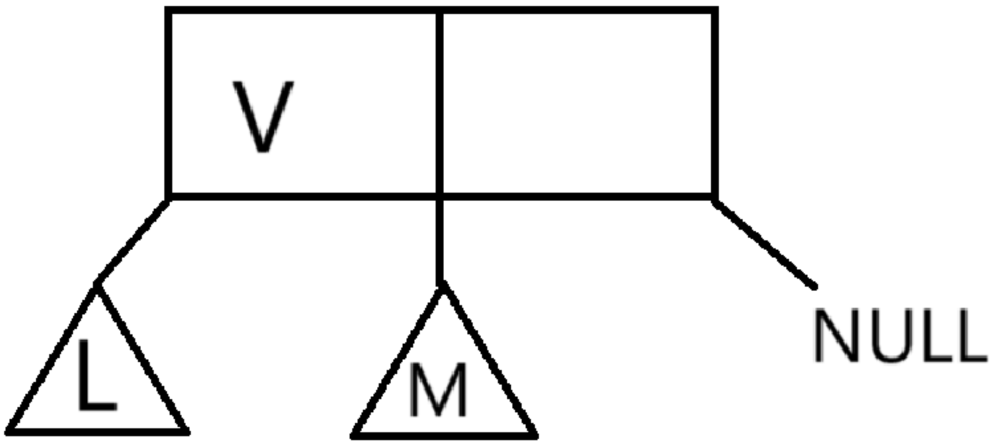
        - Top down splaying is done by doing the rotations on the tree while searching for the node
        - Main notes about top down splaying is that it only requires one pass to splay the node, but it does require a more complex operation.
    - This is likely going to disrupt the balance of the tree but that isn't an issue since a balanced tree is not the goal in a spay tree
3. Define a red-black tree Given a BST, color the nods the BST so that if satisfying the definition of the RB-tree
    - Definition from notes:
        - Version of a binary search tree
        - Nodes are colored either red or black
        - A node colored red does not have any children colored red
        - The number of black nodes from the root to any leaf node is the same



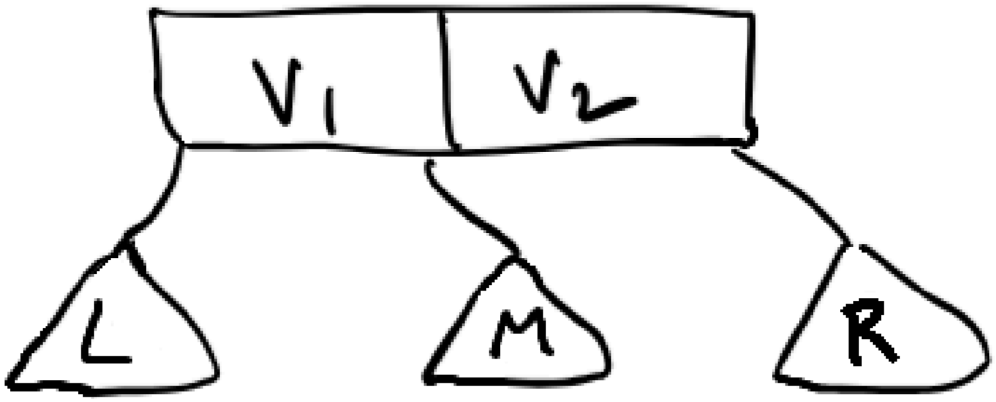
4. Define a AVL. Given a BST, determine the balance factor of each node in the tree
    - Definition from notes:
        - Version of Binary search tree
        - Each node x has a balance factor
            - the balance factor is found by the **height of the left subtree - the height of the right subtree** $(h_2 - h_1)$
            - The balance factor is good if its either -1, 0, or 1, anything else is an unbalanced tree, which is then handled through rotations

- When the balance factor is outside of the acceptable range it typically means there are one of 4 violations, which are then handled by different forms of rotations
    - Left left violation -> single zig rotations
    - right right violations -> single zag rotations
    - right left violations -> zag zig rotation
    - left right violation -> zig zag violation
        - Remember when doing double rotations the second rotation is done first. So for a zag zig rotations (aka left right rotation), the node is first rotated left, **then** rotated right

5. Given a AVL tree, perform additions in a sequence, write down the violations if any and perform rotations to fix it
    - Specifics of the procedure is essentially described in the above question; simple add the next value in accordance to BST properties then check for balanced height, if unbalanced preform proper operation
6. Define a 2-3 tree data structure. Write down C++ class definition of the 2-3 data structure.
    - 2-3 trees, although they look very similar to BSTs they do not fall under that category. This is due to the unique nature of the trees structure. 2-3 trees have a 2 node and 3 child structure, this creates a unique tree were it in a way expands upward instead of expanding downward. I'll elaborate later
    - There are a few rules a 2-3 takes on and they are as follows:
        - 2-node (2 children):
            - 1 value and 2 children (left and middle, right child is null)
            - Values in the left tree are less than the single value in the node, while the node in the middle is greater than the single value
            - L < V < M



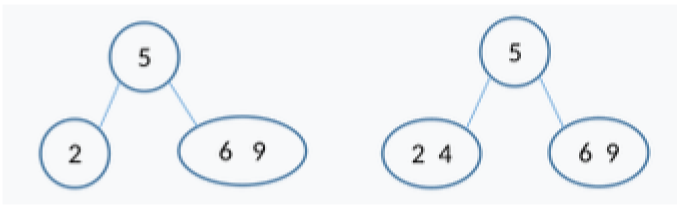
        - 3-node:
            - 2 values and 3 children (left middle and right)

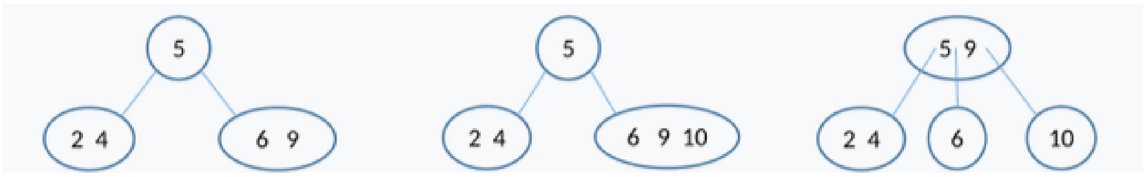


- If a node is a leaf node than L,M, and R are all null
- If a node is a non-leaf node, then it needs to follow the rules of the 2-node or 3-node version
- All leaf nodes have to be a the same level
- Here is the rough C++ class for 2-3 trees

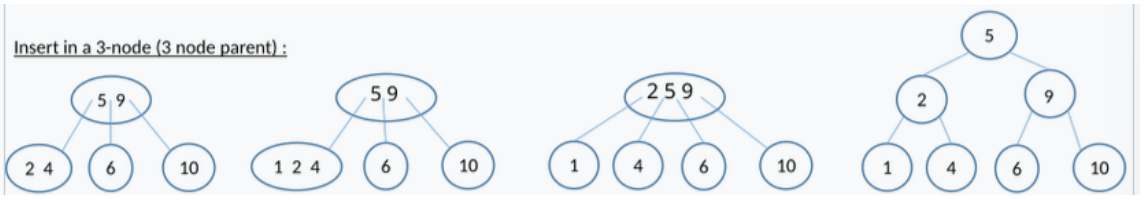
```
class twoThreeTree {
  public:
    twoThreeTree(void);      // Constructor
    ~twoThreeTree(void);     // Destructor


    void add(int item);      // Adds an item
    void search(int item);   //
    Searches for an item

  private:
    twoThreeNode *root;      // Pointer to root node

    // Private helper functions go here
};

class twoThreeNode {
  public:
    int firstData, secondData; // Two data fields

    // The three child nodes
    twoThreeNode *first, *second, *third;

    // This next one is quite useful. It aids
    // moving around the tree. It is a pointer
    // to the parent of the current node.
    twoThreeNode *parent;
};
```

- He doesn't have the method implementations anywhere, but its basically just M-trees from that one project so use your imagination

7. you will be given a 2-3 data structure and you will be asked to perform a sequence of insertions and show the resulting tree after each insertion
    - insertions on a 2-3 tree are kinda strange, as I said earlier they kinda expand upward instead of downward.
    - The easiest case of insertion is when a subtree only has 2 nodes, and the node that needs to be inserted into only has one value. In this case the new element can simply be added to the node and no extra work has to be done
    - 

    - The next case is still pretty simple but it requires a little more work; this case begins as a 2 node, but the node that needs to be inserted into already has 2 values, so when the value is inserted it would then have 3 which is not allowed. In this case the middle of the 3 values is **popped** into the parent node making it a 3 node, but the rule of a 3 node is it has to have ... 3 nodes. So the node were the value was popped from is split up into two nodes where the smaller value is now the middle node and the larger value is now the right node.
    - 

    - The next case begins with a 3 node 2-3 tree, where the node where the incoming value needs to be added to only has 1 value. In this case the value can simply be added with no rearranging. Its the same concept as the first case, but there are just more nodes now
    - The last and most complicated case is a 3-node, whose desired child node already ahs 2 values. In this case the value is added resulting in an error, the middle value is then popped, which puts 3 into the parent node and 4 nodes that are children. Then the parent node's middle value is popped creating a new parent. The picture makes more sense.

- 

8. Why is 2-3 tree better than an AVL or RB-tree
    - Really aren't any sourcing praising 2-3 trees for anything more than their them being a good teaching concept :)
    - But one possible benefit is that that tree never has to do rotations it just auto balances based on the rules, so its organization is slightly simpler

Binary trees

1. Given a set of numbers construct a binary search tree with minimum height
    - minimum height $h = log_2(n + 1)$
    - There are many ways to construct a tree with minimum height the, the main way is just by using rotations, but given this is in the binary trees section I don't really know If that's the answer he wants here
2. What are the operations that can be performed on a binary tee? Note binary Tree not a binary search tree
    - The operations he gave us in the notes are:
        - root() -> gets root
        - size() -> gets size (number of nodes)
        - LC() -> gets left child
        - RC() -> gets right child
        - isLeaf() -> checks if node is a leaf node (no children)
        - height() -> gets height of the tree (recursively calls left and right child, and adds 1, the result will be the height of the longest path. Its lowkey clever)
        - setLeft() -> sets left
        - setRight() -> sets right
            - for anything referring to left and right nodes its kinda just getters and setters since its just the data of the given node
    - There are also the operations:
        - insert(node x) -> adds a new node to the tree at the next available spot to preserve the full tree (since its not a BST it doesn't actually need to be sorted)
        - delete(node x) (aka remove) -> delete given node and distribute its children accordingly if it has any
        - Traversal methods, there's a lot so I'll list them here then explain it later

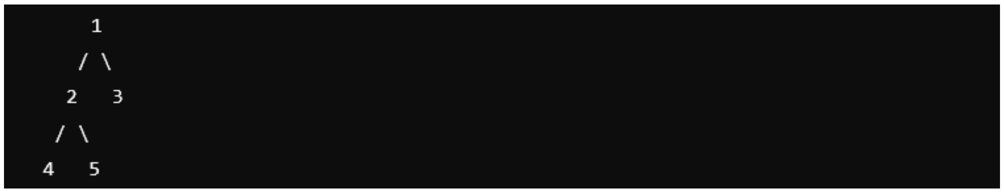
        - pre-order, in-order, post-order, depth-first order, and breadth-first order. Those last two aren't gonna be described, but they were explained along time ago so just go back there

3. Algorithms for preorder, InOrder, and post order traversals
    - All three of these algorithms are pretty much the exact same but the three lines are just moved around. The three elements are:
        - **Print the value of the root**
        - **process the left subtree**
        - **process the right subtree**
    - Preorder traversal is:
        - print root
        - process left
        - process right
    - Inorder traversal is:
        - process left
        - print root
        - process right
        - This traversal print the tree in sorted order if the binary tree is properly sorted, which is very useful in a lot of methods
    - Postorder traversal is:
        - Process left
        - Process right
        - print root
    - The actual algorithm is as follows (only showing preorder):

    ```
    template <class DT> //DT is just an unestablished data type
    void BT<DT>::Preorder(){
        if(_info != null) { // base case - if leaf node it stops the
    recursion
            cout << *_info; // print root
            (*LC).Preorder(); // recursive call to process left subtree
            (*RC).Preorder(); // THEN right subtree
        }
    }
    ```

4. Given a preorder and InOrder, construct the unique binary tree
    - The general idea of this is to use the preorder list to get the root of the tree and the subtrees, while using the inorder list to organize the subtrees
    - Here is the instruction given from geeks for geeks:

        - Take the **first** element of the **pre-order array** and create **root** node.
        - Find the index of this node in the **in-order** array.
        - Create the **left subtree** using the elements present on left side of root node in in-order array.
        - Similarly create the **right subtree** using the elements present on the right side of the root node in in-order array.
5. The number of ways a binary tree can be stored - 6 different ways
    1. Array representation
        - Like every source does it different but its chill
        - He basically has an array that stores 3 values per node. I can't really tell if there's any rhyme or reason to the indexes, but there probably is. Anyway the three values are _info, RC, and LC. The values stored in RC and LC are the indexes of where they are stored in the array, this allows everything to remain connected. Also in leaf nodes the RC and LC are marked as -1
    2. Linked representation
        - Basically just a linked list with 2 pointers instead of 1
    3. Parent Array representation
        - Parent array created by DFS, this is covered in graphs section I think
    4. Child-sibling representation
        - This isn't very useful in simple binary trees, but is a lot more useful in k-ary, trees since they can have more than 2 per node
        - But anyway this is basically a way of simplifying a tree by having a node point to its left child and any sibling it has, so its simply a step that percusses one of the other steps ... idk
            - Like sources were straight up saying its useful for converting more complex trees into binary trees which are good for easier storage, so ya
    5. Traversal (inorder and preorder)
        - Literally what question 4 is doing
    6. Generalized list representation
        - What I could find is as follows:
            - ['A', ['B', None, None],['C', None, None]]
            - The nodes in this representation are A, B, and C, where A is the root and B and C are leaf nodes. The main reason this is useful as its a simple representation of what nodes are children of who. I'm assuming this is a whole different data structure, but its still a good representation
        - Another example is:
            - [1, [2, [4, None, None], [5, None, None]], [3, None, None]]

            - In this example the nodes are 1,2,3,4,5, where 2 and 3 are children of 1 and 4 and 5 are children of 2. This can be understood based on the positioning of brackets, its a little convoluted, but it kinda makes sense
            - Here is a picture of the tree
                - 

```
      1
     / \
    2   3
   / \
  4   5
```

6. What are conditions to perform a removal from a binary search tree
    - The first two cases are very simple:
        - Case 1 : The node has no children (its a leaf node)
            - just delete the node
        - Case 2 : The node has one child
            - replace the node with the child
        - Case 3 : Node has two children (stuff actually happens)
            - Find the **in-order predecessor** (largest node in the left subtree) or **in-order successor** (smallest node in the right subtree)
            - Replace the node's value with the in-order predecessor/successor value
            - Recursively delete the in-order predecessor/successor node, which will be a simpler case (either a lead or a node with one child)

Stacks and queues

- What are the operations that can be performed on a stack and queue data structure?
- Compare and contrast linked list and array implementations of the stack and queue data structure
- Algorithm for infix to postfix conversation
- Convert an infix expression to a postfix expression
- Radix sort - Show the steps and what data structure is used for Radix sort

Linked List

- C++ class definition for a linked list
- Describe an array implementation of a linked list
- Write an algorithm to reverse a linked list

Arrays, matrices, and Vectors

- Know the binary search algorithm and what is the complexity

- Given a sorted array and a number (value) show the order of comparison as part of the binary search
- How many ways a matrix can be sorted? 6 ways. Describe them

---

Chapter 2 Algorithm and recursion

Chapter 3 Arrays, Strings, and vectors

Chapter 4 Linked list

Chapter 5 Stacks and queues

Chapters 6 Single dimensional Binary Trees

Chapter 7 Self-Modifying Search Trees

Chapter 8 Priority Search Trees

Chapter 9 Sorting

Chapter 10 Hashing

Chapter 11 Graph