

Module 3

**

Binary Search Tree

- Ordered binary trees
- Nodes can have two subtrees
- Items in the left subtree are smaller than the given node
- Items in the right subtree are larger than the given node
- When deleting a node, find the smallest bigger node to replace with (look below the deleted node on the right subtree)
- A tree should be height-balanced
- Not a long line of nodes on one side (this is bad)

Balanced Search Trees

- Guaranteed height: $O(\log n)$ for n items)

Red Black Tree









- Properties
- Every node is either red or black
- The root and leaves (NIL) are always black
- If a node is red, its children are black
- All paths from any node to its NIL descendants contain same number of black nodes
- Put black NIL nodes in all the places for the leaves where there is no actual node
- Height guaranteed to be $\log n$
- A node requires one extra bit of storage to store the color
- The longest path (from the root to the farthest NIL) is no more than twice the shortest path (from root to the nearest NIL)
- Major operations
- Search
- Insert
- Complexity
- Fix violation
- $O(\log n)$
- Color red
- $O(1)$
- Rotate

- $O(1)$
- Total
- $O(\log n)$
- There will only be insertions on quizzes and exams
- Steps
- Insert Z and color it red
- Recolor and/or rotate nodes to fix violation of property
- Which property could be violated
- If a node is red, its children are black
- The root and leaves (NIL) are black
- Case 0: Z = root
- Recolor the node to be black
- Case 1: Z.uncle = red
- Recolor Z's parent, grandparent, and uncle to be opposite
- Case 2: Z.uncle = black (triangle)
- It is a triangle if Z is
 - Left child of right child
 - Right child of left child
- Rotate Z.parent in the opposite direction of Z
- Case 3: Z.uncle = black (line)
- It is a line if Z is
 - Left child of left child
 - Right child of right child
- Rotate Z.grandparent in the opposite direction of Z
- Recolor Z's original parent and grandparent
- Remove
- Rotation
- Needed for insert and remove
- Left rotation
 - Right child comes up with its entire subtree
 - Parent comes down to the left
- Right rotation
 - Left child comes up with its entire subtree
 - Parent comes down to the right
 - The left child's subtree moves to the right??
- Alter the structure of a tree by re-arranging subtrees
- Complexity is constant $O(1)$
- Rotation balances the tree
- Self balancing

- Tree is balanced if all properties of the rb tree is satisfied
- Important relationships
- Z is the node that we will be doing things to (insertion, deletion, rotation, etc)
- Z's parent is A
- Z's grandparent is B
- Z's uncle is D

B trees

- Self-balancing search tree
- Every node
- Can contain multiple keys
- Can have more than two children
- The number of keys in a node and the number of children for a node depends on the order of the B tree
- Every B tree has an order
- Order m = max number of children
- Ex. -tree of order 4 contains a maximum of 3 key values in a node and a maximum of 4 children for a node
- Properties
- All leaf nodes must be at the same level
- All nodes except root must have at least $\lceil m/2 \rceil - 1$ keys and maximum of $m-1$ keys
- All non-leaf nodes except root (i.e. all internal nodes) must have at least $m/2$ children
- If the root not is a non-leaf node, then it must have at least 2 children
- A non leaf node with $n-1$ keys must have n number of children
- All the key values in a node must be in ascending order
- Operations
- Search
- Similar to BST
- Search process starts from the root node but here we make an m way decision every time
- Where 'm' is the total number of children the node has
- In the B-tree, the search operation is performed with $O(\log n)$ time complexity
- Algorithm
- Read the search element from the user
- Compare the search element with the first key value of root node in the tree
- If both are matched, then display "given node is found!" and terminate the function
- If both are not matched, then check whether search element is smaller or larger than that key value
- If search element is smaller, then continue the search process in the left subtree

- If search element is larger, then compare the search element with next key value in the same node and repeat steps 3,4,5, and 6 until we find the exact match or until the search element is compared with the last key value in the leaf node
- If the last key value in the leaf node is also not matched then display “element not found” and terminate the function
- Insertion
 - In a b-tree, a new element must be added only at the leaf node
 - That means, the new key value is always attached to the leaf node only
- Algorithm
 - Check whether tree is empty
 - If tree is empty, then create a new node with new key value and insert it into the tree as a root node
 - If tree is not empty, then find the suitable leaf node to which the new key value is added using BST logic
 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node
 - If that leaf node is already full, split that leaf node by sending the middle value to its parent node. Repeat the same until the sending value is fixed into a node
 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one
- 
- 
- 
- 
- 
- 
- 
- 
- Deletion

**