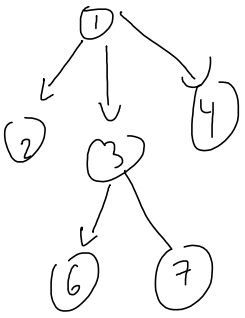
9. Trees

https://visualgo.net/en/bst

Monday, May 16, 2022 7:01 AM

Hierarchical Data Structure



root is topmost node
- 1

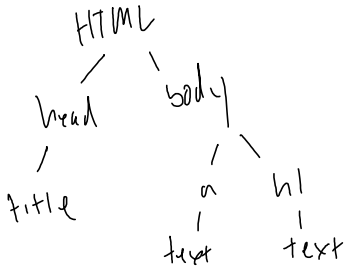
Parents are Elements directly above elements
- 1, 3

Children are Elements directly below elements
- 2 3, 4 and 6, 7

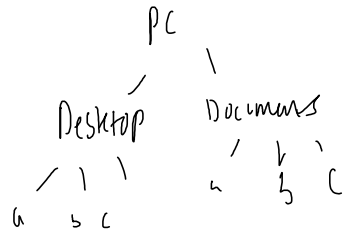
leaves are Elements without children
- 2, 4, 6, 7

Siblings are elements that share the same parent
- 2, 3, 4 and 6, 7

Examples: HTML DOM



PC File directory



Linked Lists are linear trees

Binary Trees

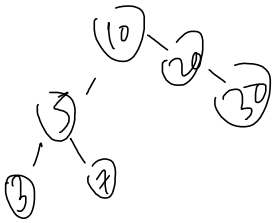
1) Each node can only have 0, 1, or 2 child node
2) Each child has only one parent

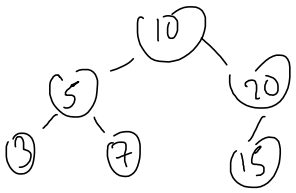
Types:

incomplete binary tree vs

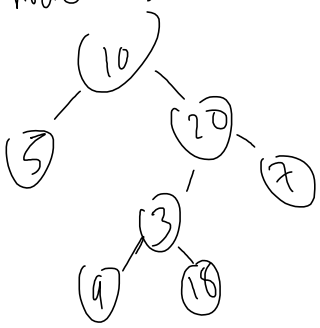
Full Binary Tree

- every level is fully filled, except for perhaps the last level. the last level is filled left to right.

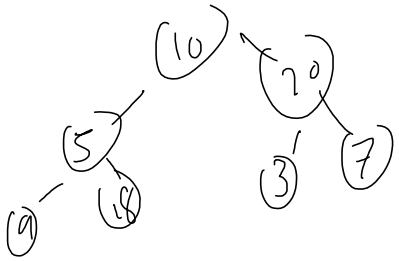




Full Binary Tree
- every node has either 0 or 2 children. No node has 1 child



Perfect Binary Tree ($2^k - 1$ nodes, half of nodes at bottom)
- Both full and complete. all leaf nodes are at the same level and this level has the max amount of nodes

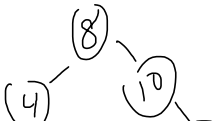


* Binary Search Tree

https://visualgo.net/en/bst

	Best	Worst
. lookup	O(log N)	O(n)
. insert	O(log N)	O(n)
. Delete	O(log N)	O(n)

- BST is a binary tree in which every node is: all left descendants <= n < all right descendants
  - This applies to all descendants and not just immediate children



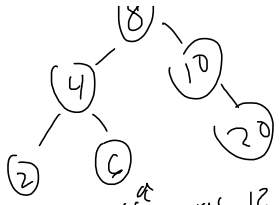
children

level 0 : $2^0 = 1$

1 : $2^1 = 2$

2 : $2^2 = 4$

3 : $2^3 = 8$



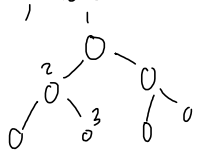
if 6 was 12, then invalid b/c 12 is left of 8

# of total nodes $= 2^h - 1$, where h = height

log nodes = steps

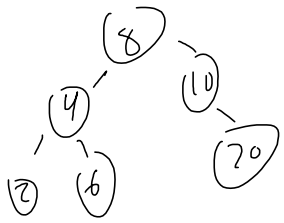
log 100 = 2 because $10^2 = 100$

log N, based on height, the max # of decisions is log N

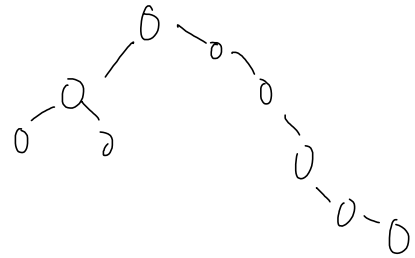


- max of 3 steps instead of 8
  - b/c of divide & conquer
- analogous to looking through a phonebook
  - you go by names, sections, not entire book

- good for searching, eg. finding 6
  1. 6 < 8, so go left
  2. 6 > 4, so go right



- bad for insert & delete b/c if there are a lot of child nodes then you have to shift a lot of nodes
  - or trees can be really unbalanced, which is O(n)



BST

Pros

- n log than O(n)

Cons

- no O(1) operations
- will need to iterate

Pros

- Better than O(n)
- Ordered (sorted data)
- Flexible size

Cons

- No O(1) operations
  - b/c you need to iterate

BST Implementation

Node

- data
- left pointer
- right pointer

BST

- lookup
- insert
- delete

Main

- Test

Insert(value) & lookup(value) pseudocode

- Declare temp node, currentNode, for traversal
- while loop or recursive

```
if (value < currNode.data) {
    if (currNode.left == null) {      // insert if empty
        currNode.left = newNode;
    }
    currNode = currNode.left          // keep traversing
}
```

Delete

- temp node for traversal
- temp node to keep reference of parent node that you want to delete
- while / recursive

...to it's parent

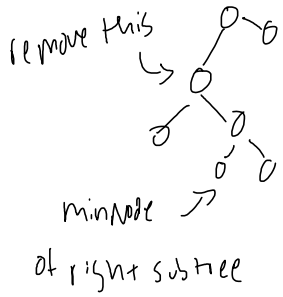
1) traverse while tracking currNode & ...

2) Match

3 cases:

1 child [1) no left child: overwrite w/ right child
2) no right child: overwrite w/ left child

3) 2 child: find min node in right subtree,
append currNode.left (node to be deleted) to minNode.left
so it saves deleted node's left subtree




```
public class binarySearchTree {
   Node root = null;

   // For print2D function
   static final int COUNT = 10;


   public Node insert(int value){
      // Initialize new node to be inserted
      Node newNode = new Node(value);

      // Set new node as root if tree is empty
      if ( root == null) {
         root = newNode;
         return newNode;
      }

      // set temp currentNode for traversal
      Node currentNode = root;

      while(true) {
         // left
         if (value < currentNode.data) {
            // Insert if left of traversed node is empty
            if (currentNode.left == null) {
               currentNode.left = newNode;
               return newNode;
            }
            // if not empty, keep traversing
            currentNode = currentNode.left;
         }
         // right
         else if (value > currentNode.data) {
            // Insert if right of traversed node is empty
            if (currentNode.right == null) {
               currentNode.right = newNode;
               return newNode;
            }
```

```
   public class Node {
      int data;
      Node left;
      Node right;

      Node(int d) {
         data = d;
      }
   }



public class Main {

   public static void main(String[] args) {
      binarySearchTree tree = new binarySearchTree();
      tree.insert(8);
      tree.insert(4);
      tree.insert(2);
      tree.insert(6);
      tree.insert(10);
      tree.insert(20);
      System.out.println(tree.lookup(4));
      System.out.println(tree.lookup(9));

      tree.print2D();
      tree.remove(4);
      System.out.println("------------------------------------");
      tree.print2D();
   }
}
```

```
Key 4 exists:
true
Key 9 exists:
false
```

```
            if (currentNode.right == null) {
               currentNode.right = newNode;
               return newNode;
            }
            // If not empty, keep traversing
            currentNode = currentNode.right;
         }
      }


   }

   // Check if the node/value ur existing for exists

   public boolean lookup(int value) {
      System.out.println("Key " + value + " exists: ");
      if (root == null) {
         return false;

      }
      Node currentNode = root;
      // set temp currentNode for traversal, if currentNode finishes
traversing and there's nothing left then exit
      while (currentNode != null) {
         // left
         if (value < currentNode.data ) {
            currentNode = currentNode.left;
         }
         // right
         else if (value > currentNode.data ) {
            currentNode = currentNode.right;
         } else if (currentNode.data == value) {
            return true;
         }
      }
      return false;
   }

   public Node remove(int value){
      if (root == null) {
         System.out.println("Tree is empty, nothing to delete");
         return null;
      }

      Node currentNode = root; // For traversal to node you want to
delete
      Node parentNode = null; // Reference to parent node of node you
want to delete

      // traverse
      while (currentNode != null && currentNode.data != value) {
         // save parent ref
         parentNode = currentNode;
         // left
         if (value < currentNode.data) {
            currentNode = currentNode.left;
         }
         // right
         else if (value > currentNode.data) {
            currentNode = currentNode.right;
         }
      }
      // Match, value == currentNode.data
      // Node to be deleted is the root node
      if (parentNode == null) {
         return removeThisNode(currentNode);
      }
      // Node to be deleted is left of parent
      if (parentNode.left == currentNode) {
         parentNode.left = removeThisNode(currentNode);
      }
      // Node to be deleted is right of parent
      else if (parentNode.right == currentNode) {
         parentNode.right = removeThisNode(currentNode);
      }
          return root;
   }

   private Node removeThisNode(Node curr) {

      // 1 child case
```

```
Key 4 exists:
true
Key 9 exists:
false

                20

        10

8

                6

        4

                2
-----------------------------------

                20

        10

8

        6

                2
```

IRS, del, lookup

time: O(log N) for balanced trees,
basically the tree's height
O(N) worst case

space: O(1) because we're iterative
not recursive

```
        // No left child, return right child, so it can be overwritten
        if (curr.left == null) {
            return curr.right;
        }
        // No right child, return left child, so it can be overwritten
        if (curr.right == null) {
            return curr.left;
        }

        // 2 child case

        // Find min node in right child subtree
        // Append curr.left to minNode.left, so it saves deleted node's left
subtree
        // Return right subtree, so it can be overwritten

        Node minNode = findMin(curr.right);
        minNode.left = curr.left;
        return curr.right;
    }


    private Node findMin(Node curr) {
        while (curr.left != null) {
            curr = curr.left;
        }
        return curr;
    }

    // GFG helper function to print tree
    public static void print2DUtil(Node root, int space)
    {
        // Base case
        if (root == null)
            return;

        // Increase distance between levels
        space += COUNT;

        // Process right child first
        print2DUtil(root.right, space);

        // Print current node after space
        // count
        System.out.print("\n");
        for (int i = COUNT; i < space; i++)
            System.out.print(" ");
        System.out.print(root.data + "\n");

        // Process left child
        print2DUtil(root.left, space);
    }

    // Wrapper over print2DUtil()
    public void print2D()
    {
        // Pass initial space count as 0
        Node printNode = root;
        print2DUtil(printNode, 0);
    }

}
```

https://visualgo.net/en/bst?slide=1

Balancing Trees

AVL Tree | Red-Black Tree

AVL Tab 4

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

balances itself

. does switch and/or rotation to auto balance

https://visualgo.net/en/heap?slide=1

Binary Heap Tree

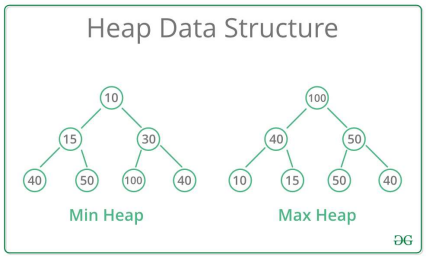
lookup: O(n)
insert: O(log n)
delete: O(log n)

max heap: root node is highest

min heap: root node is lowest

left & right can be value, as long as it's less than the top value

- good for comparative operations
  - I want all values under 31 → [



- good for priority queues, data storage, sorting algorithms

insertion is left to right (w/ node switch)
- always complete
- never unbalanced
- very efficient b/c balanced

Memory heap != heap data structure (free storage)

Priority Queues (why heaps are important)

- each element has a priority
- elements with higher priority are served first
  - analogous to airplanes (pilot boards then stewardess then passengers)
  - nodes switch places to correct priority order (even if inserted in the wrong order)

heaps
pros: priority
cons: slow lookup

pros: priority slow lookup

Flexible size

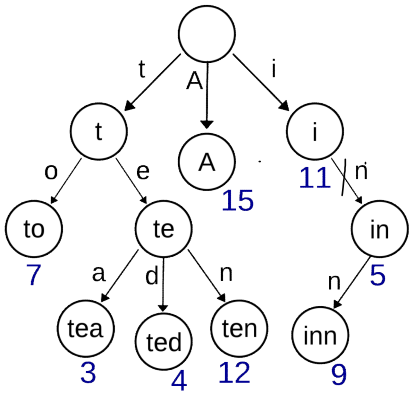
Fast insert

good for find max or find min , O(1)

https://www.cs.usfca.edu/~galles/visualization/Trie.html

Trie Tree

- aka prefix tree
- good for searching words for a dictionary / providing auto suggesting / IP routing
- specialized tree for searching
  - most often with text
- finds if a word / part of word exists in a body of text
- usually has an empty root node as the start



- Big O O(length of word)