
O
G
D
O
O
O
D
O

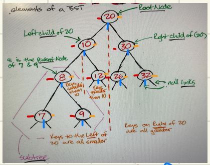
Linked Lists

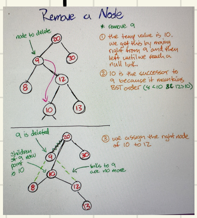
cons -
use more memory
must traverse through nodes

insertion + deletion O(1)
traverse O(n)

BST

- left node < Parent
- right node > parent
- Balanced if every level filled except last which must be filling left to right





hight of balanced search tree with n nodes = log(n)

Time complexity balanced

Insertion
Removal } O(log(n))
Deletion

Search max element
balanced : O(log n)
not balanced : O(n)

RBT

- Self balancing binary search tree

Time Complexity

	best case	worst case
search	O(log n)	O(log n)
insert	O(1)	O(log n)
delete	O(1)	O(log n)

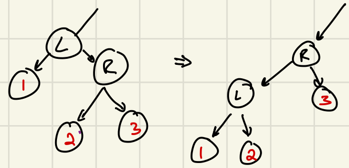
black hight of tree is # nodes from root → leaf
correct

Nodes rotated clockwise or CCW around a pivot node

properties
- every node either black or red
- all NULL nodes are black
- red nodes do not have red child
- Root is black

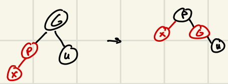
Operations

rotate :

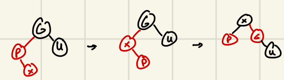


insert : ① insert node make color red
② if (node is Root): change color to black
else : Check parent is black
if (black) : done
else :

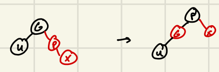
Case 1 : (LL rotation)



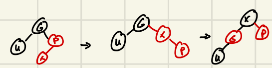
case 2 (RL rotation)



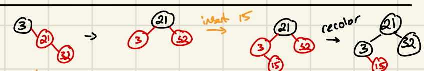
Case 3 : (RR rotation)



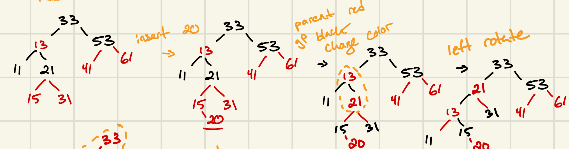
Case 4 : (RL rotation)

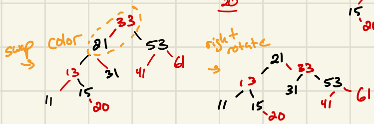


Ex :



Ex :





\# Rotate when tree not balanced

DFT vs BFT

inorder
Left → root → right
7, 8, 9, 10, 12, 20, 26, 30, 32

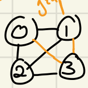
pre order
root → left → right
20, 10, 8, 7, 9, 12, 30, 26, 32

post order
Left → right → root
7, 9, 8, ...

hits each level at a time
left → right

Graphs

add orange make complete graph



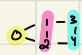
adj list : 0 → 1 → 2
1 → 0 → 2
2 → 0 → 1 → 3
3 → 2

Time Complexity
Add edge O(1)
Remove edge O(n)
Initializing O(n)

BFS → find shortest path

- select any vertex
- initialize visited queue
- add all of selected vertex neighbors to quee
- deque vertex
- add vertex neighbors neighbors to que
- deque vertex neighbors
- continue untill que is empty

Ex :



visited
0 1 2 3 4
queue
~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~

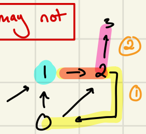
Time Complexity for DFS or BFS
O(V + E)
V: # vertexes
E: # edges

BFS always finds shortest path DFS may not

DFS

- pick a root node
- traverse as far as possible on each branch before going to next branch

Ex :



root

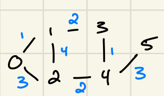
Visited
1 2 0 3

Dijkstra's

time complexity
$O(v^2)$ → $O(E \log(v))$
with use of heap and priority queue

- each edge is weighted
- choose path that adds to lowest #

Ex :



Shortest from 0 → 5

0 → 1 → 3 → 4 → 5
weight = 7

Hashing

index = key % size of hash table

Ex: 42 29 44 52 25 66 32

table size = 7

0	1	2	3	4	5	6
42	29	44	52 → 66	25 → 33		

chaining to deal with collisions

worst case access time $O(n)$

↳ collision is linked list search = $O(n)$

best case access time $O(1)$

Open Addressing → second way to deal with Collisions

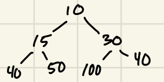
↳ find next empty spot in hash table and insert

note: Clustering can be an issue

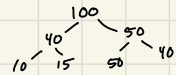
many values having same index after hashing

Heaps

Min heap



Max heap



- complete binary tree
- heapify → create heap from array : Time complexity $O(n)$
- insertion and deletion : Time complexity $O(\log n)$ ← hard rule

Time complexity access : $O(1)$

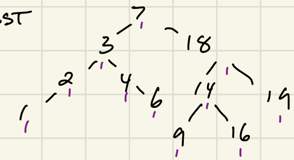
Heap - Binary tree that satisfies heap property

Heap used to impliment priority queue

Binary Heap – each node has at most 2 children

$\{7, 3, 18, 2, 4, 14, 19, 1, 6, 9, 16\}$

to BST ⇒



print inorder → 1 2 3 4 6 7 9 14 16 18 19

Priority Queues

first element in queue either greatest or least of all elements in queue

Ex: max priority queue

10 → 8 → 5 → 3 → 1

min priority queue

1 → 3 → 5 → 8 → 10

operation	complexity
empty	$O(1)$
size	$O(1)$
top	$O(1)$
push	$O(\log n)$
pop	$O(\log n)$
swap	$O(1)$

Question 2

Say we have a directed, unweighted graph C++ implementation. You are given the following pair of functions.

```
bool detectCycleHelper(Vertex* v, int startK) {
    if (!v->visited) {
        v->visited = true;
        for (unsigned int i=0; i<v->adjList.size(); i++) {
            if (v->adjList[i].v->key == startK)
                return true;
            if (detectCycleHelper(v->adjList[i].v, startK) == true)
                return true;
        }
    }
    return false;
}

bool Graph::detectCycle(int k) {
    Vertex* startV = search(k);
    return detectCycleHelper(startV, k);
}
```

Now assume an object of the Graph class is constructed with the following list of vertices and their corresponding edges.

```
Graph g;

g.insertVertex(10);
g.insertVertex(12);
```

Page 3 of 8

```
Node* BST::magicA(Node* currNode, int* counter, int k)
{
    if ( currNode == NULL) {
        return NULL;
    }

    Node* right = magicA(currNode->rightChild, counter, k);
    if (right != NULL) {
        return right;
    }

    ++(*counter);

    if (*counter == k) {
        return currNode;
    }

    return magicA(currNode->leftChild, counter, k);
}
```

practice MCQ's

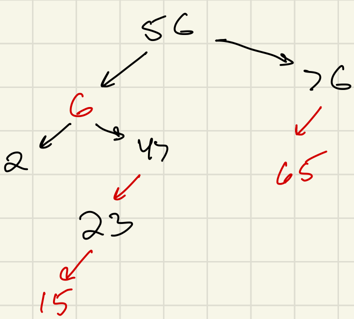
1. c
2. c
3. b
4. b
5. ~~d~~ a
6. b
7. a
8. d
9. ~~b~~ c
10. ~~a~~ b
11. b
12. b
13. a
14. d
15. b

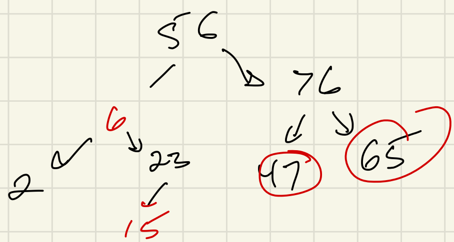
1. b
2. b
3. b
4. b
5. ~~d~~ b
6. b, c, ~~d~~ a

1. ~~d (c)?~~ a
2. ? ~~b~~ d
3. c
4. b
5. a
6. ~~c~~ a
7. a

1. c
2. a
3. c
4. b
5. b
6. b
7. d
8. c
9. c
10. a
11. c
12. a
13. d
14. b
15. b
16. b

1. ~~a~~ c ?
2. b
3. b
4. ~~a~~ b
5. a
6. a
7. ~~d~~ a
8. b
9. ~~a~~ b
10. ~~b~~ a
11. a
12. b





R R B

926

203