CSE 100 Tutor Review Section

Anish, Daniel, Galen, Raymond
THIS MAY NOT COVER EVERYTHING...

... Such as C++ code implementation, and everything else in the lecture slides that’s not in here.

- Go over lecture slides
- Try the practice final
- Look over your PAs
TREES
BST (Binary Search Tree)

- Basic properties
- Successor
- Runtime analysis
- Average total depth
BST - Basic Properties

1. Given node X, all nodes left to X have smaller keys.

2. Given node X, all nodes right to X have larger keys.

3. No duplicates.
BST - Basic Properties (Example)
BST (Binary Search Tree)

- Basic properties
- Successor
- Runtime analysis
- Average total depth
**BST - Successor**

How to find the successor of node X:

1. If X has right child, go right once, and then go all the way left. Where you end up is the successor node.

2. If X doesn’t have a right child, keep going up the parent until:
   a. find a node where it is the left child of *its* parent. -> that parent
   b. There’s no more parent -> None.
BST (Binary Search Tree)

---

- Basic properties
- Successor
- Runtime analysis
- Average total depth
BST - Runtime Analysis

---

Worst Case:

Imagine a straight-line BST:

- Insert: 0(n)
- Find: 0(n)
- Remove: 0(n)
Binary Search Tree - Runtime Analysis

---

**Average Case:**

Imagine a full BST:

- Insert: $O(\log n)$
- Find: $O(\log n)$
- Remove: $O(\log n)$
BST (Binary Search Tree)

---

- Basic properties
- Successor
- Runtime analysis
- Average total depth
BST - Average Total Depth

---

Probabilistic Assumptions:

1. All keys in the tree are equally likely to be searched for.

2. Any order of insertion (i.e. any permutation) of the keys $k_1, \ldots, k_n$ is equally likely to be used to build the BST.

3. $ND(N) = (N+1)D(N-1)+2N-1$
You are given a BST with keys 4, 8, 10, 12. Assume that 8 is always inserted first. What is the average total depth of this BST?

```
    8
   / \
  4   (10, 12)
```

- \( D(1) = 1 \)
- \( D(2) = 3 \)
- \( D(\text{Left-subtree}) = D(1) + 1 = 2 \)
- \( D(\text{Right-subtree}) = D(2) + 2 = 5 \)
- \( d(\text{root}) = 1 \)

\[ D(\text{Total}) = d(\text{root}) + D(\text{Left-subtree}) + D(\text{Right-subtree}) = 8 \]
AVL (Georgy Adelson-Velsky and Evgenii Landis’) Tree

---

- Basic properties
- Insert & rotate examples
- Runtime analysis
AVL Tree - Basic Properties

1. Basically the same as BST except it’s always balanced.

2. A node is out of balance when the heights of its left and right children differ by more than 1.

3. To balance, it does rotations (single/double).
AVL (Georgy Adelson-Velsky and Evgenii Landis’) Tree

---

- Basic properties
- Insert & rotate examples
- Runtime analysis

h = 3
d = 2
AVL Tree - Insert & Rotate (Single rotation)

---

Insert: C

```
A
  h = 2
d = 1

B
  h = 1
d = 0

C
  h = 1
d = 0
```
AVL Tree - Insert & Rotate (Single rotation)

---

Insert: C

Update heights

Node A is out of balance!
AVL Tree - Insert & Rotate (Single rotation)

---

Insert: C

A
  B
    C
  h = 3  
d = 2

B
  C
  h = 2  
d = 1

A
  B
    C
  h = 1  
d = 0

Left

B
  A
    C
  h = 2  
d = 0

A
  B
    C
  h = 1  
d = 0
AVL Tree - Insert & Rotate (Double rotation)

---

Insert: E
AVL Tree - Insert & Rotate (Double rotation)

Insert: E
AVL Tree - Insert & Rotate (Double rotation)

---

Insert: D

```
A
h = 1
d = 0

B
h = 3
d = 1

C
h = 2
d = 1

E
h = 1
d = 0
```
AVL Tree - Insert & Rotate (Double rotation)

Insert: D
AVL Tree - Insert & Rotate (Double rotation)

---

Insert: D

Update heights

Node C is out of balance!
AVL Tree - Insert & Rotate (Double rotation)
AVL Tree - Insert & Rotate (Double rotation)

---

Before Insert:

- B
  - A
    - C
    - D
    - E
    - h = 3
    - d = 1
  - h = 2
  - d = 1

- C
  - h = 1
  - d = 0

- D
  - h = 1
  - d = 0

- E
  - h = 1
  - d = 0

After Insert & Rotate (Double rotation):

- B
  - A
    - C
    - D
    - E
    - h = 1
    - d = 0
  - h = 3
  - d = 1

- C
  - h = 1
  - d = 0

- D
  - h = 1
  - d = 0

- E
  - h = 1
  - d = 0

Left rotation applied.
AVL Tree - **Insert & Rotate (Summary)**

---

Single rotations are shown to the right:

Sometimes, you need to rotate twice (watch out for jagged path!)

For more practice/demo: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
AVL (Georgy Adelson-Velsky and Evgenii Landis’) Tree

---

- Basic properties
- Insert & rotate examples
- Runtime analysis
AVL Tree - Runtime Analysis

---

Remember, AVL Tree is simply balanced BST.

Therefore, there is not really the “worst case”.

Average Case / Worst Case:

- Insert: $O(\log n)$
- Find: $O(\log n)$
- Remove: $O(\log n)$
Binary Heap

---

- Basic properties
- Runtime analysis
Binary Heap - Basic Properties (Part 1: Order)

- Max heap: Given a node X, all of its children have smaller or equal key
- Min heap: Given a node X, all of its children have larger or equal key
- Duplicates allowed!
Binary Heap - Basic Properties (Part 1: Order)

Min heap

Max heap
Binary Heap - Basic Properties (Part 2: Shape)

---

- All of the leaves at depth d-1 are to the right of the leaves at depth d.

- There is at most 1 node with just 1 child.

- That child is the left child of its parent, and it is the rightmost leaf at depth d.

**TL;DR:** Heap is always filled in from top to bottom, and left to right.
Binary Heap - Basic Properties (Part 2: Shape)

bad:

good:
Binary Heap

- Basic properties
- Runtime analysis
Notice that heap looks like a full BST.
Therefore, insert/remove = $O(\log n)$.

However, the most important characteristic of heap is that:

- $\text{findMin} = O(1)$ for a min heap, and
- $\text{findMax} = O(1)$ for a max heap.

That is because the root always has the min/max key.
Treap

---

- Basic properties
- Insert & Rotate
- Remove
- Runtime analysis
Treap - **Basic Properties**

---

Treap is basically BST + heap. Every node is a pair of (key, priority).

And **It must satisfy both of these conditions:**

- Given a node X, all of its children have smaller or equal priority than X.
- All nodes in its left subtree have smaller key than X, and all nodes in its right subtree have larger key than X.
Treap

---

- Basic properties
- Insert & Rotate
- Remove
- Runtime analysis
How to insert node $X(\text{key, priority})$:

1. Insert like you do in BST by comparing key with other nodes’ keys.

2. Look at the priorities and do some AVL rotations around $X$ until the priorities ordering is correct.
Treap - Insert & Rotate

Insert: (B,99)
Treap - Insert & Rotate

---

Insert: (B, 99)
Treap - Insert & Rotate

---

Insert: (B,99)
Treap - Insert & Rotate

---

Insert: (B,99)

50  

35  

40  

5

99 > 50

99  

B

A

C

F

L
Treap - Insert & Rotate

---

Insert: (B, 99)

Done!
Treap

---

- Basic properties
- Insert & Rotate
- Remove
- Runtime analysis
**Treap - Remove**

---

1. Find the node you want to remove.

2. Rotate it down (use the child with higher priority) until it’s a leaf node.

3. Delete it!
Remove: B
Treap - Remove

Remove: B
Remove: B
Treap - Remove

Remove: B
Treap - Remove

Remove: B

Done! Safely remove B.
Treap

---

- Basic properties
- Insert & Rotate
- Remove
- Runtime analysis
Treap - Runtime Analysis

Worst Case:

Imagine a straight-line treap:

- Insert: $O(n)$
- Find: $O(n)$
- Remove: $O(n)$
Treap - Runtime Analysis

---

Average Case:

Imagine a balanced treap:

- Insert: $O(\log n)$
- Find: $O(\log n)$
- Remove: $O(\log n)$
RST (Randomized Search Tree)

---

Is a treap with randomly generated priorities.

THAT’S IT.
Red-Black Tree

---

- Basic properties
- Insert & Rotate
- Runtime analysis
Red-Black Tree - Basic Properties

1. Every node is either red or black.

2. Root is always black.

3. Red node can’t have red children.

4. Every path from a node to null references (empty children) must have same number of black nodes.
Red-Black Tree - Basic Properties

1. Every node is either red or black.
2. Root is always black.
3. Red node can’t have red children.
4. Every path from a node to null references (empty children) must have same number of black nodes.
Red-Black Tree

---

- Basic properties
- Insert & Rotate
- Runtime analysis
Red-Black Tree - Insert & Rotate

---

Case 1 - X is left child of red parent

Single rotate
Red-Black Tree - Insert & Rotate

Inserted: 3
Red-Black Tree - Insert & Rotate

---

Inserted: 3

Fixed
Red-Black Tree - Insert & Rotate

---

Case 2 - X is right child of red parent.

Double Rotate!
Red-Black Tree - **Insert & Rotate**

---

Inserted: 7
Red-Black Tree - Insert & Rotate

Inserted: 7

Fixed
Red-Black Tree - Insert & Rotate

---

While you insert:

Step 1 - As you go down the tree, whenever you see a node with two red children, make the node red and its children black.

Step 2 - If step 1 breaks the RB tree properties, do some rotations there.

Step 3 - Continue inserting
Red-Black Tree - Insert & Rotate

Insert: 64
Red-Black Tree - Insert & Rotate

---

Insert: 64
Red-Black Tree - Insert & Rotate

Insert: 64
Red-Black Tree - Insert & Rotate

Insert: 64
Red-Black Tree - Insert & Rotate

Insert: 64

Done!
Red-Black Tree

---

- Basic properties
- Insert & Rotate
- Runtime analysis
Red-Black Tree - **Runtime Analysis**

---

Red-Black tree is like an average case BST.

**Worst/Average case:**

- **Insert:** $O(\log n)$
- **Find:** $O(\log n)$
- **Remove:** $O(\log n)$
Depth First Search (DFS)

- Recursion based
- $O(|V|+|E|)$
- High level algorithm:
  - function DFS(n):
    - 1) Mark n as visited
    - 2) for every adjacent vertex v:
      - if v has not been visited
        - run DFS(v)
  - Begin with DFS(s), where s is the starting node

Courtesy of Wikimedia Commons
Breadth First Search (BFS)

- Queue based
- $O(|V|+|E|)$
- High level algorithm:
  - 1) Start at some node $s$
  - 2) Queue $s$ into a queue
  - 3) While the queue is not empty:
    - Pop the first vertex $v$ in the queue
    - Mark $v$ as visited
    - For every adjacent node $n$ of $v$:
      - if $n$ has not been visited
        - enqueue $n$
Dijkstra’s Shortest Path

- Uses a priority queue!
- $O(|E|\log|V|)$
- High level algorithm:
  - Enqueue tuple $(s, c)$ starting vertex $s$; cost $c = 0$
  - While the priority queue is not empty:
    - Pop next tuple $(v, c)$
    - If $v$ is marked done, ignore and continue
    - Else set $v$’s done to true
    - For each of $v$’s adjacent nodes, $w$, that aren’t done:
      - $C = \text{cost}(v,w) + \text{dist}(v)$
      - if $C < \text{dist}(w)$,
        - $\text{dist}(w) = C$, $\text{prev} = v$, enqueue tuple $(w, C)$
Dijkstra’s cont.

- Don’t try to find the shortest path by first look
  - feel free to double check your work with this
- When running Dijkstra's, create a table like so:

<table>
<thead>
<tr>
<th>VERTEX</th>
<th>DISTANCE</th>
<th>PREV</th>
<th>DONE</th>
<th>ADJACENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>∞</td>
<td>-1</td>
<td>F</td>
<td>(v1, 1), (v2, 4)</td>
</tr>
<tr>
<td>v1</td>
<td>∞</td>
<td>-1</td>
<td>F</td>
<td>(v3, 1)</td>
</tr>
<tr>
<td>v2</td>
<td>∞</td>
<td>-1</td>
<td>F</td>
<td>(v1, 100)</td>
</tr>
<tr>
<td>v3</td>
<td>∞</td>
<td>-1</td>
<td>F</td>
<td>(v1, 3)</td>
</tr>
</tbody>
</table>
Minimum Spanning Tree (MST)

- Remember:
  - A graph can have many spanning trees
  - A tree cannot have any cycles
- MST = the least-cost version of a graph that is still connected
- Algorithms to use?
  - Prim’s
  - Kruskal’s
Prims vs Kruskal's

- "Add the cheapest local edge in your growing tree"
- $O(|E|\log|V|)$
- Priority Queue

- "Add the next cheapest edge in the whole graph"
- $O(|E|\log|V|)$
  - faster than Prims in practice
- Union find for faster cycle checking
Union Find w/ Up Trees

- 1) Find(4) -> 1
- 2) Find(3) -> 0
- Union(1, 0)
- 3) Find(4) -> 0
- 4) Find(3) -> 0

- Array implementation on slides
Smarter Unions

- With default union operations, we might have a larger tree
- Improve with deterministic approach:
  - Union by size:
    - root with larger size becomes parent
  - Union by height:
    - root with larger height becomes parent
Path Compression w/ Find

- Taking advantage of an initial find’s traversal to optimize a later find
- Can be done with tail recursion
Hashing
Hashing

- 0(1) search, insert, delete
- However, in order traversals on hash tables are nigh impossible
- Design table size, hash function, and collision resolution accordingly
- For good performance, make hash function distribute keys uniformly
Collision Resolution Strategies

- Linear Probing
- Double Hashing
- Random Hashing
- Separate Chaining
Linear Probing

- Walk the array with step size 1 until you reach an empty spot
- However this creates clumps of elements
Collision Resolution Strategies

- Linear Probing
- Double Hashing
- Random Hashing
- Separate Chaining
Double Hashing

- Hash function finds starting spot
- Walk down the array with step size equal to a hash function
- Reduces clumping
- This requires a prime table size. Why?

Double Hashing Example

<table>
<thead>
<tr>
<th>Insert</th>
<th>Hash Value</th>
<th>Insert</th>
<th>Hash Value</th>
<th>Insert</th>
<th>Hash Value</th>
<th>Insert</th>
<th>Hash Value</th>
<th>Insert</th>
<th>Hash Value</th>
<th>Insert</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>76%7 = 6</td>
<td>93</td>
<td>93%7 = 2</td>
<td>40</td>
<td>40%7 = 5</td>
<td>47</td>
<td>47%7 = 5</td>
<td>10</td>
<td>10%7 = 3</td>
<td>55</td>
<td>55%7 = 6</td>
</tr>
<tr>
<td></td>
<td>5 - (47%5) = 3</td>
<td></td>
<td>5 - (55%5) = 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Probes: 1 1 1 2 1 2
Collision Resolution Strategies

- Linear Probing
- Double Hashing
- Random Hashing
- Separate Chaining
Random Hashing

- Hash function finds starting spot
- Walk down the array with step size equal to some random number generator
- Reduces clumping. Similar idea and performance to Double Hashing
Collision Resolution Strategies

- Linear Probing
- Double Hashing
- Random Hashing
- Separate Chaining
Separate Chaining

- Hashing function finds starting point
- Value inserted at the end of the linked list
- Make sure to search linked list for duplicates
- Advantage: Table never gets “full”
- Disadvantage: Dynamic memory usage yields poor performance
TRIES and Skip List
Radix Search

---

Key: the value in the nodes (e.g. 010101001010101001010101)

Segment/digit: smaller part of a key (e.g. 0,1)

Radix/Base: The size of possible digit values. (e.g. 2)

If the length of a digit is 4 bits, for example, (1111, then the radix is 16)

So radix search is comparisons that involve only one digit of the keys at a time.
Binary Tries

1. Binary Tries are binary trees, but not BST.
2. Difference between Binary Tries and BST:
   a. in BST, both internal nodes and leaves hold values. in Binary Tries, only leaves hold values.
So basically, to search value in BST, you compare values between nodes and nodes (parents and children). To search value/key in Binary Tries, you need go by digit/segment (e.g. 100101 or sth like that).

And insertion in Binary Tries is the same as found. If the node is not found by the given index/sequence (In case some wield words are given in final...), then insert at the place.
Binary Tries (worst case example)
Why do we need a smaller height?

---

Less Time Complexity in worst case.
Binary Tries (Continued)

3. Similarity between BST and Binary Tries:
   a. Left Small. Right Biggggg. (You know what I mean bro)
   b. Traversal order

4. Limitation on Binary Tries:
   a. When keys are of different length, and one key is the prefix of another. A oh....... :(
Solution to Prefix in Binary Tries
Multiway Tries

Multiway Tries are just more complex tries (Are u sure u use ‘just’?)

A binary trie use radix 2 (which means 0 and 1). A multiway Tries use radix > 2 (which means 0, 1, 2, ... ,radix − 1)

\[ \text{radix} = 2^n \]

n is number of bits of the value of keys.
Multiway Tries (with end)
Disadvantage of Multiway Tries

---

Waste a lot of space:

Think about alphabet multiway tries. Each node has 26 children......

Solution is?
Ternary Tries

---

A KEY digit (the c, a, l, and etc.)

3 Children (left, right, mid)

An end bit
Ternary Tries (Example)
Ternary Tries (Runtime Analysis)

---

Although Ternary Tries are more space efficient, it is less time efficient.

But in practice they are good. $O(\log N)$ for in average case.

So lol...
Huffman Encoding Tree

Characters with least frequencies go first.
Skip List

---

Big-0 for search, delete, insert of linkedlist? O(N)

Big-0 for Skip List?

O(logN) in average case.

O(N) in worst case (very very rare...)
Skip List
Skip List (Useful Links)

https://www.youtube.com/watch?v=Dx7Hk8-8Kdw
B Trees and B+ Trees
B and B+ Trees

- Useful for organizing data on disk
- $O(\log n)$ Search, Insert, Delete
Typical memory hierarchy: a picture

AMOUNT OF STORAGE (approx!) -> CPU -> TIME TO ACCESS (approx!)

- hundreds of bytes -> CPU registers -> 1 nanosecond
- hundreds of kilobytes -> cache -> 10 nanoseconds
- hundreds of megabytes -> main memory -> 100 nanoseconds
- hundreds of gigabytes -> disk -> 10 milliseconds
Basic Idea

- All leaves are at the same depth.
- Nodes are at least half full.
- Therefore tree is balanced.
- Overflowing a node rebalances by splitting the node, pushing the middle value to parent, and creating two children.
B-Trees (Section C only)

- Since Prof. Sahoo already went over this on Friday I’m hoping you guys are OK...?
- B-Trees store keys and data at all intermediary nodes and leaves.
- As a result, inorder traversals take much longer
- Sections A&B are not responsible for knowing this in detail.
B+ Tree (Sections A, B, & C)

- Two parameters: $M$ (# children), $L$ (# of keys/data for leaves)
- Internal nodes have between $\lceil M/2 \rceil$ and $M$ children
- Leaves have between $\lceil L/2 \rceil$ and $L$ keys/data
- Internal nodes have 1 more key than they have children
- **Data is only stored at leaf nodes**
B and B+ Trees are challenging...

---

Good visualization tool:

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
2-3 B+ Tree

- We like 2-3 B & B+ Trees because they’re easy to draw (hint-hint, nudge-nudge, wink-wink)
- Definition: $M=L=3$.
- Each internal node has between 2 and 3 children. Each leaf has 2-3 data nodes
Insert 15 into this B+ Tree (easy)
15 successfully inserted into this B+ Tree
Now insert 10 into this B+ Tree (fun)
Inserting 10 into this B+ Tree

- - -

- First, try to insert 10 into the 9,13,15 node
  - It doesn’t fit!!
- In order to keep all invariants consistent
  - Split the node into two nodes
  - Connect both new nodes to the parent
  - And add a key to the parent
10 successfully inserted into this B+ Tree
## Data Structure Comparison

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Avg Insert</th>
<th>Worst Insert</th>
<th>Avg Find</th>
<th>Worst Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted Array</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Queue</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Skip List</td>
<td>O(log N)</td>
<td>O(N)</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>BST</td>
<td>O(log N)</td>
<td>O(N)</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>AVL/RBT</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
</tr>
<tr>
<td>Min-heap</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Hash table</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>B Trees</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
</tr>
</tbody>
</table>

Courtesy of Professor Sahoo
Now homeworks are done, a joyous elate
but finals are come, and you can't see straight
So go and relax, enjoy life for a bit
For you are more stressed, than you would admit

And you'll hear the bells, see candles aglow
So put on Adele, and go say Hello
And though nights be dark, go stroll in moonlight
It may light your heart, this December night.
It’s been a pleasure being your tutors for this quarter!
Q + A