Basic Algorithms - lec - 1

Course Overview:

- Programming assignments. (C++)
  - Efficient implementations.
  - Optimization of code.
  - Memory management.

- Problem Sets.
  - Rigorous proofs.
  - Proof writing techniques.
  - Algorithm Design

- Exam
  - Open notes (only written notes).
  - Ofc no internet access.

- Quiz
  - 2-3 small problems. (open notes)

Basic Outline:
- Solve lots of problems to analyze techniques.
- Different areas of algorithms design.
- Intro to mathematical techniques & theories.
**What is an algorithm?**

Any structured procedure with well-defined steps, which has some input and output.

**What is it used for?**

- Human genome project
  - Aligning DNA, identifying mutations/SNP/indel
  - Translocation
- Internet routing / secure encryption
  - Sending packets (finding path to destination)
  - How to encrypt / decrypt them
- Signal processing
  - Making music into digital signals 128 Kbps
  - Discretizing from continuous back
  - Loss of compression (JPEG)
  - No lossy PNG
  - Different ones
Appointment Scheduling:

Problem
You are a secretary for a doctor, have to schedule appointments for the next month.
- maintain appointment times.
- after finishing remove from the pool.
- be able to add new appointments.
- if a new time is within 15 minutes of an appointment already booked, then it cannot be added.
- within 9:00 → 5:00 pm (last appointment tomorrow).

'ex.'

now  today

11:00 11:30 12:00 12:15 12:35

12:30 → not allowed.
12:50 → allowed.
11:45 → allowed.

Can implement using sorted / unsorted list:

insertion  O(n)   O(1)
deletion    O(n)   O(1)
lookup      O(log n) O(n)
**Binary Search Trees:**

**Search Trees → Terminology:**
- Node
- Children
- Descendants
- Ancestor
- Leaf
- Parent
- Different kinds: upper, lower
- Look-up
- Insertion
- Deletion
- Needs a searching property

**Binary:**
- Each node only has two children
  - Left, Right

**BST Property:**

∀ x ∈ BST
∀ y ∈ x. left subtree.

∀ y ∈ x. key ≤ y. key.

∀ y ∈ x. right subtree.

y. key ≥ x. key

---

```
11:30
11:00  12:15
  12:00  12:35
```
making a BST.

- 11:00
- 12:15
- 12:00
- 12:35

- 11:30

 gives the previous example.

- insert depends on height.

- how to search?  

- depends on height.

- Every operation depends on the height of the tree.

- need to keep the height small.

- (If we insert in increasing order.)
AVL Trees:

named after György Andelston-Velsky B.

Evgenii Landis, in 1962.

example of a self-balancing BST.

Keeps the height of the tree small.

- we need extra properties to ensure this.

\[ \text{height of left node} \leq 1 \]

\[ \text{height of right node} \leq 1 \]

\[
\begin{align*}
h - 2 & \quad | \quad h - 1 \quad | \quad h \\
\end{align*}
\]

\[ n_h \geq 1 + n_{h-1} + n_{h-2} \]

\[ \geq 1 + 2n_{h-2} \]

\[ n_h \geq 2n_{h-2} \]

\[ n_h \geq \frac{h}{2} \]

\[ h \leq 2 \log_2 n_h \]
need to keep this invariant true for every insertion / deletion.

Right Rotate($x$) →

Left Rotate($y$)

we assume $x$ is not balanced.

Case 1

$h + 1 < h - 1$

$h < h - 1 / h$

right rotate ($x$)

$y → h+1 / h+2$

$y → h/h+1$ balanced!
Case 2.

just right rotate doesn't work.

we need to go deeper!

R^{-1}/R^{-2}. (one of them is R^{-1}). could be both.
left rotate (Y).

Right rotate (X).

Conclusion:
we can maintain BST invariants.

⇒ BST has height \( \leq 2 \log_2 n \)

⇒ every operation can be done \( \leq 2 \log_2 n \) time.
Deletions
A tiny bit more complicated.

Steps.
1. Find the node to be deleted, X.

2. Swap X with minimum node of B or maximum node of A.
   (Edge case, don’t need to worry)
   (What happens if one of the children is empty?)

3. Now remove the leaf containing X.
   & balance up the tree up to the root.

(Tiny change, X might not be a leaf.)
In this case move the subtree up one level.
more operations.

1) closest upper closest.
2) closest lower.

- solving appointment scheduling
  - when appointment finishes remove from tree
  - find closest & check for overlap
    - if no overlap you can insert.

more variations
- 2-3 tree
- Red black tree
- a-b tree
- splay tree
- B-tree
other problems solving using BST.
  * Dictionary implementation. Also uses hash-maps.
  * C++ -> ordered map
    (uses red-black tree).

Augmentation

Store more information in each node:
  types of augmentation.
  → storing size of tree, (at each node)
  → storing min/max of tree, (at each node)
  → other info like booleans / AP/CAP, etc.

Size augmentation

At each node store number of nodes in the whole subtree.

\[
x.\text{size} = y.\text{size} + z.\text{size} + 1.
\]
Uses:
- find $k^{th}$ element in tree
- for any query, find number of elements less than (greater) query

no need to have $B & T \leq$ node. key
property. only do insertion, deletion by position. (this acts as a list interface).
with fast API.

---

min-max Range update

Problem:
insert $(k, v_{ai})$ - $O(\log n) \leq 2 \log n$
query $(k)$ - $O(\log n) \leq 2 \log n$
add $(a, b, v)$ - $\leq n$

naive is too slow.

range update can be made faster by augmenting nodes with a carry value.

add extra value node. carry.
$x \cdot \text{carry} = v$

\[ X \]

\[ A \quad B \]

$\Rightarrow$ every value in the tree has increased by $v$.

don't have to individually go to each node to increase the value. (how to query?).

how to update using this?

write update function for a node

update (node, a, b, v).

three cases.

1) if the range of the subtree is fully within $[a, b]$, then add $v$ to node carry.

2) if range is fully disjoint, don't need to do anything.

3) else there is a partial intersection. then do the update for left child, right child, & if necessary update node value.
Pseudocode

update (node, a, b, v) {
    if node is NULL
        return
    elif node range \subseteq [a, b]
        node.carry += v.
    elif node range \cap [a, b] = \emptyset
        do nothing.
    else
        update (node.left, a, b, v)
        update (node.right, a, b, v)
        if node.key \in [a, b]
            node.value += v.
}
Complexity:

![Tree Diagram]

- The recursive call will never go down the nodes which are shaded.

Hence we have two paths, potentially going down to some leaves, of height \( \leq 2 \log n \).

1) For each node on this path we have to update, at most one child. (Which is in the interior), & do one recursive call.

2) We might have to update points on the paths as well.

Hence a total of \( \leq 4 \log n \) nodes per path.

\[ \Rightarrow \text{Total number of updated nodes} \leq 8 \log n. \]
Points to note:
what if you do an insert on an range which had a value added to it?

- naive insertion results in extra value to new key.
- Path clearing.

push down the value of the carry to the children on the path to the inserted key.

```
push down (node).
    node. left. carry += node. carry
    node. value += node. carry
    node. carry = 0.
```
Ex. See that rotate (right/left) still preserves the cleared path to the new key.

Similar work for delete.
- when you go to find min/max & do a swap, keep doing a pushdown for each node on the path to min/max.