Lecture 3: Scheduling and Binary Search Trees

Lecture Overview

- Runway reservation system
  - Definition
  - How to solve with lists
- Binary Search Trees
  - Operations

Readings

CLRS Chapter 10, 12. 1-3

Runway Reservation System

- Airport with single (very busy) runway (Boston 6 $\rightarrow$ 1)
- "Reservations" for future landings
- When plane lands, it is removed from set of pending events
- Reserve req specify "requested landing time" t
- Add t to the set of no other landings are scheduled within $<$ 3 minutes either way.
  - else error, don't schedule

Example

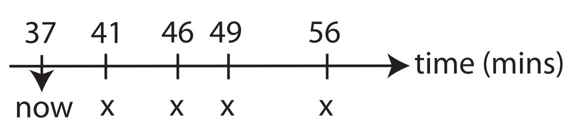


Figure 1: Runway Reservation System Example

Let R denote the reserved landing times: $R = (41, 46, 49, 56)$

Request for time: 44 not allowed ($46 \epsilon R$)
53 OK
20 not allowed (already past)
$| R |= n$

Goal: Run this system efficiently in $O(\lg n)$ time

Algorithm

Keep R as a sorted list.

```
init: R = [ ]
req(t): if t < now: return "error"
for i in range (len(R)):
    if abs(t-R[i]) <3: return "error" %\Theta (n)
R.append(t)
R = sorted(R)
land: t = R[0]
if (t != now) return error
R = R[1: ] (drop R[0] from R)
```

Can we do better?

- **Sorted list**: A 3 minute check can be done in $O(1)$. It is possible to insert new time/plane rather than append and sort but insertion takes $\Theta(n)$ time.
- **Sorted array**: It is possible to do binary search to find place to insert in $O(\lg n)$ time. Actual insertion however requires shifting elements which requires $\Theta(n)$ time.
- **Unsorted list/array**: Search takes $O(n)$ time
- **Dictionary or Python Set**: Insertion is $O(1)$ time. 3 minute check takes $\Omega(n)$ time

What if times are in whole minutes?
Large array indexed by time does the trick. This will not work for arbitrary precision time or verifying width slots for landing.
Key Lesson: Need fast insertion into sorted list.

New Requirement

Rank(t): How many planes are scheduled to land at times $\leq t$? The new requirement necessitates a design amendment.

Binary Search Trees (BST)

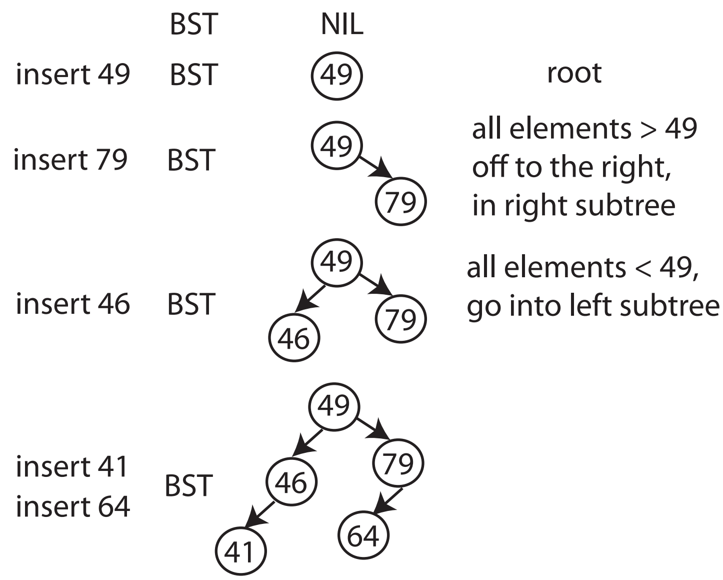


Figure 2: Binary Search Tree

Finding the minimum element in a BST

Key is to just go left till you cannot go left anymore.

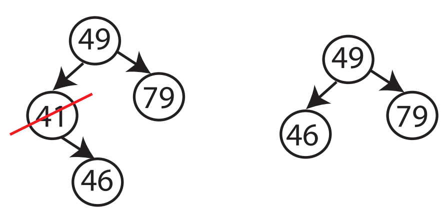


Figure 3: Delete-Min: finds minimum and eliminates it

All operations are $O(h)$ where h is height of the BST.

Finding the next larger element

next-larger(x)

```
if right child not NIL, return minimum(right)
else y = parent(x)
while y not NIL and x = right(y)
    x = y; y = parent(y)
return(y);
```

See Fig. 4 for an example. What would next-larger(46) return?

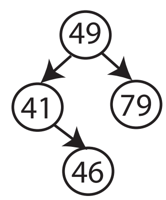


Figure 4: next-larger(x)

What about rank(t)?

Cannot solve it efficiently with what we have but can augment the BST structure.

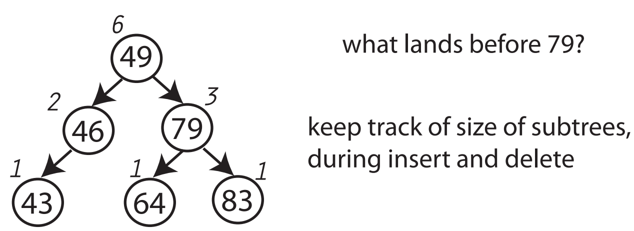


Figure 5: Augmenting the BST Structure

Summarizing from Fig. 5, the algorithm for augmentation is as follows:

1. Walk down tree to find desired time
2. Add in nodes that are smaller
3. Add in subtree sizes to the left

In total, this takes $O(h)$ time.

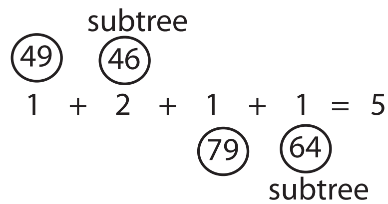


Figure 6: Augmentation Algorithm Example

All the Python code for the Binary Search Trees discussed here are available at this link

Have we accomplished anything?

Height h of the tree should be $O(log(n)$.

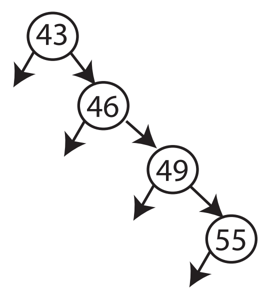
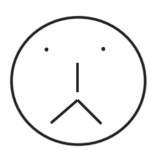


Figure 7: Insert into BST in sorted order

The tree in Fig. 7 looks like a linked list. We have achieved $O(n)$ not $O(log(n)$!!



Balanced BSTs to the rescue...more on that in the next lecture!