Lecture 5: 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 if no other landings are scheduled within k minutes either way. Assume that k can vary.
  - else error, don't schedule

Example

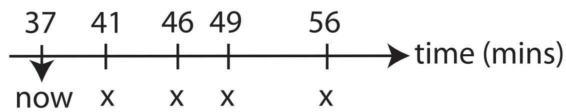


Figure 1: Runway Reservation System Example

Let R denote the reserved landing times: $R = (41, 46, 49, 56)$ and $k = 3$

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

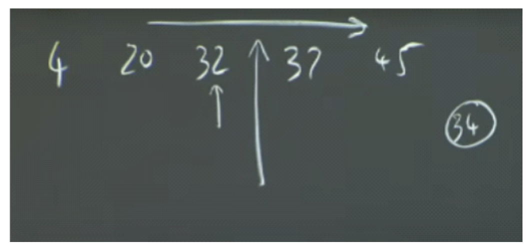
Goal: Run this system efficiently in $O(\lg n)$ time

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]) < k: return "error"
R.append(t)
R = sorted(R)
land: t = R[0]
if (t != now) return error
R = R[1: ]              (drop R[0] from R)
```

1. find the next larger element.
2. find the next smaller element.



Can we do better?

- **Sorted list**: Appending and sorting takes $\Theta(n \lg n)$ time. However, it is possible to insert new time/plane rather than append and sort but insertion takes $\Theta(n)$ time. A k minute check can be done in $O(1)$ once the insertion point is found.

- **Sorted array**: It is possible to do binary search to find place to insert in $O(\lg n)$ time. Using binary search, we find the smallest i such that $R[i] \geq t$, i.e., the next larger element. We then compare $R[i]$ and $R[i-1]$ against t. Actual insertion however requires shifting elements which requires $\Theta(n)$ time.

- **Unsorted list/array**: k minute check takes $O(n)$ time.

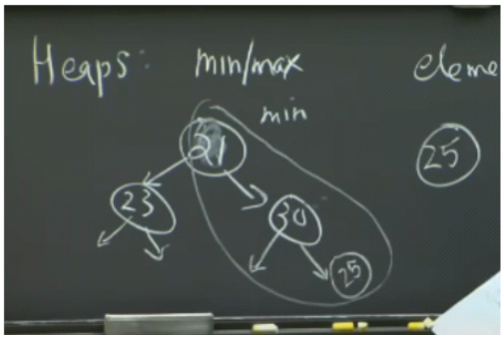
- **Min-Heap**: It is possible to insert in $O(\lg n)$ time. However, the k minute check will require $O(n)$ time.

- **Dictionary or Python Set**: Insertion is $O(1)$ time. k 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.



Binary Search Trees (BST)

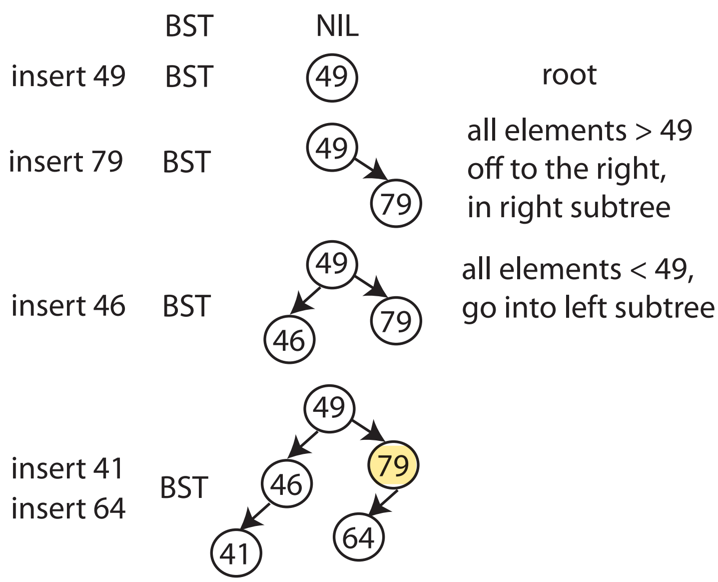


Figure 2: Binary Search Tree

Properties

Each node x in the binary tree has a key $key(x)$. Nodes other than the root have a parent $p(x)$. Nodes may have a left child $left(x)$ and/or a right child $right(x)$. These are pointers unlike in a heap.

The invariant is: for any node x, for all nodes y in the left subtree of x, $key(y) \leq key(x)$. For all nodes y in the right subtree of x $key(y) \geq key(x)$.

Insertion: insert(val)

Follow left and right pointers till you find the position (or see the value), as illustrated in Figure 2. We can do the "within $k = 3$" check for runway reservation during insertion. If you see on the path from the root an element that is within $k = 3$ of what you are inserting, then you interrupt the procedure, and do not insert.

Finding a value in the BST if it exists: find(val)

Follow left and right pointers until you find it or hit NIL.

Finding the minimum element in a BST: findmin()

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

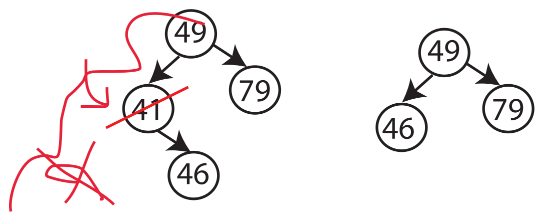


Figure 3: Delete-Min: finds minimum and eliminates it

Complexity = Not O(lgn) if the tree is unbalanced.

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

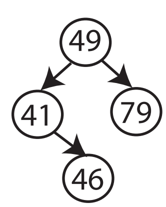
Finding the next larger element: next-larger(x)

1. For all nodes y in the right subtree of key(y) $\geq$ key(x).
2. minimum of the right subtree = the next larger element of x.

Note that x is a node in the BST, not a value.
next-larger(x)

```
if right child not NIL, return minimum(right)   1.
else y = parent(x)
while y not NIL and x = right(y)
2.  x = y; y = parent(y)
return(y);
```

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



1. For all nodes y in the left subtree of key(y) $\leq$ key(x)
2. maximum of the left subtree = the next smaller element(=y0) of x.
3. x is the next larger element of y0.

Figure 4: next-larger(x)

New Requirement

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

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

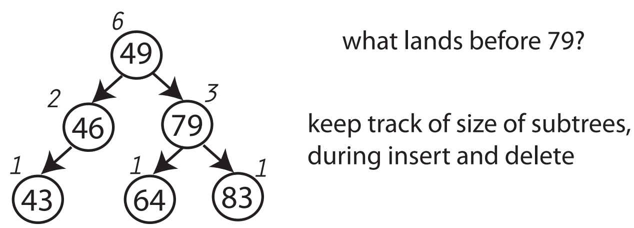
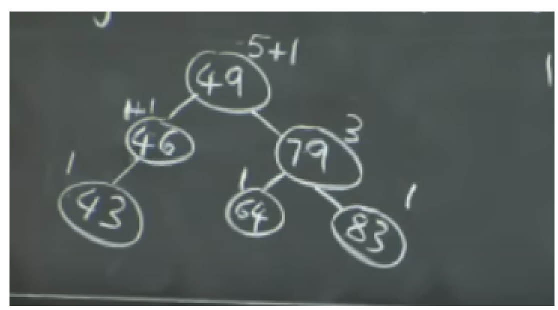


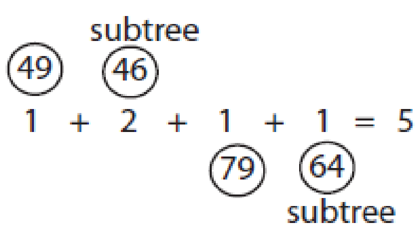
Figure 5: Augmenting the BST Structure

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.





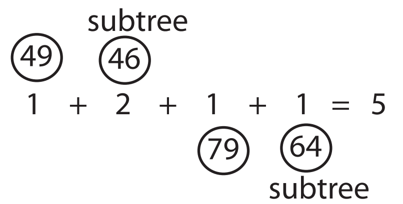


Figure 6: Augmentation Algorithm Example

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 $O(\lg n)$.

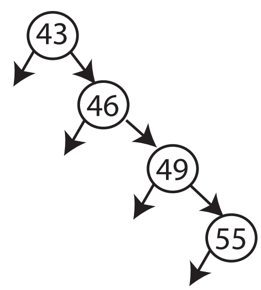


Figure 7: Insert into BST in sorted order

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

Balanced BSTs to the rescue in the next lecture!