Runway Reservation System
- Maintains a set of future landing times
- When a plane lands, remove from the set
- For a request to land at time t:
  - reject request if there are reservations within 3 minutes of t,
  - otherwise add t to set

Example

\[
\begin{align*}
\text{now} & & 41 & & 46 & & 49 & & 56 \\
27 & & 41 & & 46 & & 49 & & 56
\end{align*}
\]

\( R = \{41, 46, 49, 56\}, \quad n = |R| = 4 \)

request for time:
\begin{align*}
44 & \text{ reject (too close to 46 } \in R) \\
53 & \text{ add reservation (just right)} \\
20 & \text{ already past, not allowed.}
\end{align*}

Goal: handle requests & landings in \( O(\log n) \) time.
- Keeping $R$ as an (arbitrarily ordered) list
  - Checking a request takes $O(n)$ time
  - Deleting upon landing takes $O(n)$ time

- Keeping $R$ as a sorted list

  \[
  \text{init: } R = []
  \]

  \[
  \text{req(t): if } t < \text{now return 'error'}
  \]

  \[
  \text{for } i \text{ in range (len(R)):
  \]

  \[
  \text{if abs(t-R[i])} < 3 \text{ : return 'error'}
  \]

  \[
  R.\text{append}(t)
  \]

  \[
  R.\text{sort}() \quad \leftarrow O(n \log n)
  \]

- land: \[ t = R[0] \]

  \[
  \text{if (t#now) return 'error'}
  \]

  \[
  R = R[1:] \quad \leftarrow O(n)
  \]

- Can we do better with a sorted list?

  \[
  \begin{array}{c|c|c|c|c}
  37 & 41 & 46 & 49 & 56 \\
  \end{array}
  \]

  can use binary search to determine conflicts in $O(\log n)$ but inserting $t$ into $R$ and deleting $R[0]$ still takes $O(n)$

  \[ \text{Faster (no sorting) but not enough} \]
- Indicator representations
- Represent a set $R$ such that

$$A[t] = True \iff t \in R$$

- Python dictionaries
- Python sets
- A Python list of all possible landing times (all possible elements of $R$)

- Fast insertion, deletion
- Checking a request is fast if landing times are whole minutes, but expensive for hi-res
- Not a flexible data structure:
  How many planes land on or before $t$?
- Binary Search Trees (BSTs)

1. insert (49)  
2. insert (79)  
3. insert (46)  

```
        49
       / \        
     46   79
    / \   /  
   41 49 79 
```

- this is the root to find elements > 49 go right
- elements < 49 are in the left subtree

A more compact representation:

```
       49
      /   
    46    79
   / \   /  
  41 49 79 
```

- Same representation in the program:
- every node has a value and left/right pointers

find (464): follow left/right pointers until done
find (48): how do you tell that 48 is not in the set?
findmax(): just go right until you can't go right
findmin(): similar, go left

delete min(): min & max nodes will have zero or one child => find, eliminate, & patch
next-larger(t): \( \nu = \text{find}(t) \)

\[
\begin{align*}
\text{if} & \quad \text{right}(\nu) \neq \text{None}: \quad \text{return} \quad \text{findmin}(\text{right}(\nu)) \\
\text{else} & \quad \nu = \text{parent}(\nu) \\
\quad \text{while} \quad \nu \neq \text{None} \text{ and } \nu = \text{right}(\nu): \\
\quad & \quad \nu = y \\
\quad & \quad y = \text{parent}(\nu) \\
\quad & \quad \text{return} \quad y
\end{align*}
\]

All of these operations on the BST take \( O(h) \) time, where \( h \) is the height of the tree.

Other operations that we can do in \( O(h) \) time:

* rank(t) how many planes land \( \leq t \)?
* must augment each node with size of subtree, update during insertions \& deletions.
* delet(t) more tricky if \( t \) has 2 children, but still \( O(h) \).
* (delete next-larger(t) than put it in the node that stored \( t \); see textbook).