Binary Search Trees

- Store $n$ keys so that SEARCH, INSERT, DELETE can be performed fast.

**Example**

```
          5
         / \  \\
        3   7
       / \ /  \\
      2  4 8 \\
```

- A binary tree with key at each node

```
    x
   / \  \\
  y   z
```

- If $h$ is height of BST, then

  INSERT, SEARCH, DELETE can be done in $O(h)$ time.

  Straight forward more intricate, exercise.
- The tree could be very imbalanced and height may be too large.

- E.g. if $n$ keys are inserted
  
  \[ a_1 < a_2 < a_3 \ldots < a_n \]

  then the B.S.T. would look like

  ![Diagram of a B.S.T.]

  How to make sure $h = O(\log n)$?

  - Red-black trees
  - 2-3 trees

  We won't do in this course.

  Both these enable us to store $n$ keys so that search, insert, delete can be performed in $O(\log n)$ time.
2-3 Trees

Example

- Each internal node has 2 or 3 children.
- Keys stored only at leaves. Sorted in left to right order.
- All leaves at the same level (depth).
- Each internal node contains
  
  "range = [min : max]," 
  the min & max values in its subtree.
- Height = $O(\log n)$ if $n = \#leaves$. 

SEARCH
INSERT
DELETE

All in $O(\log n)$ time
**SEARCH**. Use the range information at internal nodes.

**INSERT**

- Insert value \( x \) as a child of appropriate internal node \( p \).
- If \( p \) still has \( \leq 3 \) children, done.
- Else \( p \) now has 4 children.
  Split \( p \) into two nodes, each with two children.

\[
\begin{array}{c}
\text{q} \\
\text{p} \\
\text{a b x c}
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
\text{q} \\
\text{p}_1 \quad \text{p}_2 \\
\text{a b x c}
\end{array}
\]

- Now the parent of \( p \), say \( q \), has one more child. Repeat the same process upwards.

- Finally, if the root has 4 children, split it into two and create new root. Height of the tree increases by 1.
DELETE - Delete x from parent p.
- If p still has 2 children, done.
- Else p only has 1 child now.

4 If sibling of p has 3 children then p can borrow a child from its sibling.
4 Else p gives away its child to its sibling. Now p can be deleted.
However, parent of p now has one less child, and the process is repeated upwards.

```
(0)
  ┌───┐
  │r  │
  └───┘
  ┌───┬───┐
  │ 2 │ p  │
  └───┴───┘
    │   │
    a  b  c  x  y

->
(0)
  ┌───┐
  │r  │
  └───┘
  ┌───┬───┐
  │ 2 │ p  │
  └───┴───┘
    │   │
    a  b  c  y

(2)
  ┌───┐
  │r  │
  └───┘
  ┌───┬───┐
  │ 2 │ p  │
  └───┴───┘
    │   │
    a  b  x  y

->
(2)
  ┌───┐
  │r  │
  └───┘
  ┌───┐
  │ 2 │
  └───┘
    │
    a  b  x  y

repeat upwards
```