Today

- Binary Search Tree (BST):
  - A data structure
  - Supports operations:
    - Insert
    - Delete
    - Search (Successor)
  - Works in comparison model
- Red-Black Tree
  - An efficient variant of BST

Binary Search Tree (BST)

- Each node $x$ has:
  - $\text{key}[x]
  - Pointers:
    - $\text{left}[x]
    - $\text{right}[x]
    - $p[x]$

- Property: for any node $x$:
  - For all nodes $y$ in the left subtree of $x$:
    \[ \text{key}[y] \leq \text{key}[x] \]
  - For all nodes $y$ in the right subtree of $x$:
    \[ \text{key}[y] \geq \text{key}[x] \]

- Given a set of keys, is BST for those keys unique?

No uniqueness

BST as a data structure

- Operations:
  - $\text{INSERT}(T, z)$: inserts node $z$ into the tree $T$
    (assumes that $\text{key}[z]$ not in $T$)
  - $\text{DELETE}(T, z)$: deletes node $z$ from the tree $T$
    (assumes $z$ exists in $T$)
  - $\text{SEARCH}(x, k)$: returns a node with key equal to $k$
    (if it exists), starting from a node $x$
### Search

**SEARCH (x,k)**

if \( x = \text{NIL} \) or \( \text{key}[x] = k \)
then return \( x \)
if \( k < \text{key}[x] \)
then return **SEARCH** (left[x],k)
else return **SEARCH** (right[x],k)

Examples:
- **SEARCH** (root,8)
- **SEARCH** (root,8.5)

### Modified Search: Search’

**SEARCH’ (x,k)**

\( y \leftarrow \text{NIL} \)
while \( x \neq \text{NIL} \) do
if \( k < \text{key}[x] \)
then \( y \leftarrow x \)
if \( k < \text{key}[x] \)
then \( x \leftarrow \text{left}[x] \)
else \( x \leftarrow \text{right}[x] \)
return \( y \)

Example: **SEARCH’** (root,8.5)

**SEARCH’** returns either:
- the predecessor of \( k \) (largest tree element smaller than \( k \)), or
- the successor of \( k \) (smallest tree element larger than \( k \))

This is more than just checking if \( k \) is in the tree or not!

### Insert

**INSERT** (T,z)

\( y \leftarrow \text{SEARCH’} (\text{root}[T],\text{key}[z]) \)

\( p[z] \leftarrow y \)
if \( y = \text{NIL} \)
then \( \text{root}[T] \leftarrow z \)
else if \( \text{key}[z] < \text{key}[y] \)
then \( \text{left}[y] \leftarrow z \)
else \( \text{right}[y] \leftarrow z \)

Example: **INSERT** (T,8.5)

### Analysis

- How much time does all of this take?
- Worst case: \( O(\text{height}) \)
- What is the worst possible height after \( n \) insertions?
  - Worst height = \( n-1 \)
  - Will need to **rebalance** the tree after insertions
    - Ensure the height is \( O(\log n) \)
    - **Balanced** search trees

### Search Tree Landscape

- AVL trees
- 2-3 trees
- **Red-black trees**
- 2-3-4 trees
- Treaps
- Leftist trees
- AA tree
- Skip lists

### Red-black trees

BSTs with an extra one-bit **color** field in each node.

**Red-black properties:**
1. Every node is either red or black.
2. The root and leaves (\( \text{NIL} \)'s) are black.
3. If a node is red, then its parent is black.
4. All paths from any node \( x \) to a descendant leaf have the same number of black nodes.
Example of a red-black tree

Use of red-black trees

- What properties would we like to prove about red-black trees?
  - They always have $O(\log n)$ height (proof later)
  - There is an $O(\log n)$-time insertion procedure which preserves the red-black properties

Inserting into RB tree

Suppose we want to insert 7.5 into the tree:

Rotations

Rotations maintain the inorder ordering of keys:
- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c$

A rotation can be performed in $O(1)$ time.

Rotations can reduce height

RB-Insert: Overview

- Perform the “usual” tree insertion
- Color the new node red
- Rebalancing:
  - Start from the inserted node $x$
  - Apply one of 3 cases
  - Possibly recurse on an ancestor node (new $x$)
- Invariant after each step:
  - All properties are satisfied in the $x$’s subtree
  - Only either Prop. 3 or Prop 2. or can be violated (but not both)
Case 1: “Uncle” node D is red

Case 2: “Uncle” node D is black and x is a right child

Case 3: “Uncle” node D is black and x is a left child

Red-black tree wrap-up

Case 1: “Uncle” node D is red

Case 2: “Uncle” node D is black and x is a right child

Case 3: “Uncle” node D is black and x is a left child

Height of a red-black tree

Theorem. A red-black tree with n keys has height 
\[ h \leq 2 \log(n + 1). \]

ARGUMENT:
- Merge red nodes into their black parents.

Height of a red-black tree

Theorem. A red-black tree with n keys has height 
\[ h \leq 2 \log(n + 1). \]

ARGUMENT:
- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth \( h' \) of leaves (by Property 4)
Height of 2-3 tree

- What is the maximum height $h'$ of a 2-3-4 tree with $n$ nodes?
- Alternatively, what is the minimum number of nodes in a 2-3-4 tree of height $h'$?
- It is $1 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$
- $n \geq 2^{h-1} - 1 \Rightarrow h' = O(\log n)$
- Full binary tree is the worst-case example!

Conclusions

- Can maintain BSTs:
  - $O(\log n)$ height
  - $O(\log n)$ operations per update