Balanced Search Trees
Binary Search Trees
A binary tree is a *binary search tree* if

- each element in the left subtree is smaller than the root,
- each element in the right subtree is larger than the root, and
- the left and the right subtree are binary search trees.

![Binary Search Tree Diagram](image-url)
Implementation

42
(key, value)

parent

left / right subtree
## Dictionary

A *dictionary* is an abstract data type which stores key-value pairs hand has the following operations:

- **Insert**($k$, $v$)
  - Inserts a key-value pair ($k$, $v$) into the dictionary.

- **Find**($k$)
  - Returns a value with the key $k$.

- **Delete**($k$)
  - Deletes a key-value pair with the key $k$. 

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert($k$, $v$)</td>
<td>Inserts a key-value pair ($k$, $v$) into the dictionary.</td>
</tr>
<tr>
<td>Find($k$)</td>
<td>Returns a value with the key $k$.</td>
</tr>
<tr>
<td>Delete($k$)</td>
<td>Deletes a key-value pair with the key $k$.</td>
</tr>
</tbody>
</table>
BST – Insert($k$, $v$)

Idea
- Find a a free spot in the tree and add a node which stores ($k$, $v$).

Strategy
- Start at root $r$.
- If $k < \text{key}(r)$, continue in left subtree.
- If $k > \text{key}(r)$, continue in right subtree.

What if $k = \text{key}(r)$?

Runtime
- $\mathcal{O}(h)$
  ($h$ is the height of the tree.)
Insert the numbers 22, 80, 18, 9, 90, 24.
Insert the numbers 22, 80, 18, 9, 90, 24.
Insert the numbers 22, 80, 18, 9, 90, 24.
Insert the numbers 22, 80, 18, 9, 90, 24.
Insert the numbers 22, 80, 18, 9, 90, 24.
Insert the numbers 22, 80, 18, 9, 90, 24.
Insert the numbers 22, 80, 18, 9, 90, 24.
Find the node with key $k$.

**Strategy**

- Start at root $r$.
- If $k = \text{key}(r)$, return $r$.
- If $k < \text{key}(r)$, continue in left subtree.
- If $k > \text{key}(r)$, continue in right subtree.

**Runtime**

- $O(h)$  
  
  ($h$ is the height of the tree.)
Find the number 22.
Delete the node with key $k$.

**Strategy**

1. $n := \text{Find}(k)$
2. Let $m$ be the node in the left subtree with the largest key or the node in the right subtree with the smallest key.
3. Replace $n$ with $m$.

**Runtime**

$\mathcal{O}(h)$

($h$ is the height of the tree.)
Delete the number 24.
Delete the number 24.
Runtime of all operations is $O(h)$.

- What is $h$ in the worst case?

Consider inserting the sequence $1, 2, \ldots, n - 1, n$

Thus, worst case height $h \in O(n)$.

- How do we keep the tree balanced?
Rotation

How do we use this to keep a tree balanced?
Red-Black Trees
A **red-black tree** is a binary search tree with the following properties:

0. The root is black.
1. A node is either red or black.
2. All null-pointers are black.
3. If a node is red, then both its children are black.
4. Every path from a given node \( n \) to any of its descendant null-pointers contains the same number of black nodes. This number is called black-height of \( n \).
The tree on the right validates property (0), (1), and (2).

(We will ignore Null-pointers from here.)
The tree on the right validates property (3).
Validation of property (3).
Theorem

A red-black tree with $n$ nodes has a height of at most $O(\log n)$. 
$T'$ is full. Thus, $h' \leq \log n$.

Because $h \leq 2h'$, $h \leq 2 \log n \in \mathcal{O}(\log n)$
Basic Strategy

- Use \( \text{Insert}(k, v) \) and \( \text{Delete}(k) \) as defined for BSTs.
- New added nodes are red.
- Problem: The resulting tree may violate some properties of a red-black tree.

Restoring Red-Black Property

- Done by rotation and recolouring.
- There are five cases for insertion and six for removal. We will not discuss them here.
- General idea: Restore properties for the current layer, move the "incorrectness" to an upper layer, and repeat this on the upper layer.

Runtime

- \( \mathcal{O}(\log n) \) for both operations
Given this red-black tree. We want to insert 4.
Given this red-black tree. We want to insert 4.
Given this red-black tree. We want to insert 4.
Given this red-black tree. We want to insert 4.
Given this red-black tree. We want to insert 4.
AVL Trees
A binary tree is an *AVL tree* if, for each node, the height of the left and right subtree differ by at most one.
AVL Tree – Example
AVL Tree – Example
Theorem

An AVL tree with $n$ nodes has a height of at most $O(\log n)$.

Proof. Let $N_h$ be the min. number of nodes in an AVL tree of height $h$.

\[
N_h = 1 + N_{h-1} + N_{h-2} \geq 2 \cdot N_{h-2} \geq 2^{h/2}
\]

Thus, $h \leq 2 \log_2 N_h$, i.e., $h \in O(\log n)$. □
Basic Strategy (similar to red-black trees)
- Use Insert($k, v$) and Delete($k$) as defined for BSTs.
- Problem: The resulting tree may violate some properties of an AVL tree.

Restoring AVL Property
- Done by rotation.
- General idea: Restore properties for the current layer and repeat this on the upper layer.
- We will not discuss the details here.

Runtime
- $O(\log n)$ for both operations
AVL Tree – Insertion Example

Insert(55)
AVL Tree – Insertion Example

Insert(55)
AVL Tree – Insertion Example

Insert(55)
Insert(55)
Insertion Example

Insert(55)
B-Trees
A B-Tree is a search tree such that, for some constant $t \geq 2$,

1. each node $n$ stores $|n|$ sorted keys ($t - 1 \leq |n| \leq 2t - 1$),
2. each node which is not a leaf has $|n| + 1$ subtrees, and
3. all leaves are on the same layer.

The root $r$ is excluded from property (1). Instead, $1 \leq |r| \leq 2t - 1$. 

```
  2  11
 /  \
/    /
1 5  7  8
   /  \
  14 15
```
Full nodes (with $2t - 1$ keys) can be slitted.

- Remove middle key.
- Include it into parent node.

Neighbouring nodes with $t - 1$ keys can be merged.

- Remove separating key from parent node.
- Add it in middle of new node.
Keys can be shifted to decrease the size of a node and increase the size of its neighbour.
B-Tree – Insertion

Idea

- Similar to BSTs, find leaf which would contain the key and add it.

Problem

- What if leaf is full (stores $2t - 1$ keys)?
- What if leaf cannot be split because parent is full too?

Solution

- When searching for leaf, split every full node on the path.

Runtime: $\mathcal{O}(t \cdot \log_t n)$

- $\mathcal{O}(t)$ for splitting nodes.
- $\mathcal{O}(\log_t n)$ for the path from root to leaf.
B-Tree – Deletion

Strategy
- Search key in tree.
- For every node on path, ensure at least $t$ keys are in the node (using merging and shifting).

Case 1: Key is in leaf.
- Simply delete key.

Case 2: Key is not in leaf.
- Replace key by $k'$, the largest key in left child or smallest key in right child.
- Recursively delete $k'$.

Runtime: $O(t \cdot \log_t n)$
- $O(t)$ for merging nodes.
- $O(\log_t n)$ for the path from root to leaf.