Weight Balance:
- Given a set of keys
  \( X = \{x_0, \ldots, x_{n-1}\} \)
- and values
  \( V = \{v_0, \ldots, v_{n-1}\} \)
- and weights
  \( W = \{w_0, \ldots, w_{n-1}\} \)
- Assume:
  \( x_0 < x_1 < \ldots < x_{n-1} \) sorted

Overview:
- Splay trees - Static
  - Optimal
  - More frequently accessed
  - keys closer to root

\( \Rightarrow \) Weight-balanced
  - trees

Implementation: (as extended BST)

Internal node:
- Stores:
  - Key key
  - \( \rightarrow \) splitter
  - float wt \( \rightarrow \) total
  - weight of
  - entries in subtree

Left, right

External Node:
- Key key,
- \( \leftarrow x_i \)
- Value value
- float wt \( \leftarrow w_i \)

How to (Nearly) Achieve Shannon's bound
- Weight-balanced
  - tree

\( \rightarrow \) For each node \( p \):
  - \( wt(p) = \text{total weight} \)
  - of keys in \( p \) subtree

balance(\( p \)) = \( \max( wt(p.\text{left}), wt(p.\text{right})) \)

\( \frac{wt(p)}{wt(p)} \)

Given \( \frac{1}{2} \leq \alpha \leq 1 \), a BST

is \( \alpha \)-balanced if

for all internal nodes \( p \),

\( \text{balance}(p) \leq \alpha \)

\( \alpha = \frac{1}{2} \): Perfectly balanced

\( \alpha = 1 \): Arbitrarily bad

\( \alpha = \frac{2}{3} \): A reasonable

compromise

Pseudo-Probability:
- Let: \( W = \sum_{i=0}^{n-1} w_i \) total weight
- Let: \( p_i = \frac{w_i}{W} \)
  - pseudo-prob
- Obs: \( 0 < p_i \leq 1 \)\( \Rightarrow \) discrete
  - \( \sum_i p_i = 1 \)
  - prob. distribution

Shannon's Theorem: If \( p_i \) is
the prob. of accessing \( x_i \),
any BST has expected search
at least \( \sum_i p_i \log p_i \) \( \leftarrow \) (called
the entropy of distrib)
Balance by Rebuilding:
Given an array $A[0..k-1]$ of external nodes:
- Assume $A[i].key$, $A[i].value$,
- Assume $A[i].wt$
- Assume keys are sorted
- Assume weights $> 0$

How to maintain balance?
Options:
- Rotations: Similar to AVL trees (single / double)
- Rebuild subtrees: Similar to scapegoat

Example:
- Given an array $A[0..k-1]$
- Let $\Delta_{min} = |(12+4)-(3+2+4)| = 2$
- $\overline{W} = 16$

Weight-balanced trees:
- Select splitter to minimize weight difference
- Let $\overline{W} = \sum_{i=0}^{k-1} A[i].wt$ (Total weight)
- Let $\overline{W}_{i,j} = \sum_{m=i}^{j-1} A[i].wt$ (Partial weight)
- Let $\Delta_j = |\overline{W}_{0,j} - \overline{W}_{j,k}|$ (Absolute weight difference)
- Goal: Split at $0 \leq j < k$ that minimizes $\Delta_{min} = \min_j \Delta_j$

$buildTree(A[0..k-1])$
- If ($k = 1$) return $A[0]$/base case*/
- $\overline{W} = \sum_{i=0}^{k-1} A[i].wt$/*total weight*/
- Init: $b = 0$; $Lut = 0$; $Rut = \overline{W}$; $\Delta_{min} = \overline{W}$
- For ($i = 0..k-1$)
  - $\Delta = |Rut - Lut|$/weight difference*/
  - If ($\Delta < \Delta_{min}$)
    - $b = i+1$; $\Delta_{min} = \Delta$
    - $L = buildTree(A[0..b-1])$; $R = buildTree(A[b..k-1])$
  - Return new Int Node($A[b].key$, $L, R$)
But it is pretty close! 😊

**Theorem:** (Mehlhorn '77)
The above balanced split algorithm produces a tree whose exp. search time is
\[ \leq H + 3 \]
where \( H \) = entropy bound.

**Dictionary Operations:**
→ Balance by destroying + rebuilding - **Jackhammer Trees**

**Find:** Same as usual. Tree height \( \leq \log_2 n \), so \( O(\log n) \) time guaranteed.

**Insert/Delete:** Start same as standard BST
→ After operation completes check + rebuild

**Analysis:**
Does this algorithm produce the optimal tree (w.r.t. expected case search time)?

- No. 😞 The optimal BST can be computed by dynamic programming

**Weight-Balanced Trees III**

**Check & Rebuild:**
→ When returning from recursive calls, update each node's weight
\[ p.wt \leftarrow p.left.wt + p.right.wt \]
→ Starting at root, walk down search path. Stop at first node \( p \) s.t.
\[ \text{balance}(p) > \alpha \]

Given by designer e.g. \( \alpha = 2/3 \)

→ If no such \( p \) found - Great! Tree is balanced
Else: **Jackhammer**!
→ Traverse \( p \)'s subtree in order, store external nodes in array \( A[0..k-1] \)
→ Replace \( p \)'s subtree with \( \text{buildTree}(A) \)

**Bad weight distributions?**
- If a weight is very large relative to neighbors, rebalance may be ineffective

**Lemma:** If weights are "nice" (not too much variation), insert + delete run in \( O(\log n) \) amortized time.
**Very heavy entries:**
- If an entry’s weight is too high, rebuilding is ineffective.
- Example:

```
    5
   / \
   a   b
  /     \
 /       8
1       c
1       1
```
- This tree is best possible!

**Exemption:** Don’t rebuild if a key’s weight is very high.

**For node p:**
\[
\text{max}(p) = \text{max weight in p's subtree}
\]
\[
\text{max-ratio}(p) = \frac{\text{max}(p)}{\text{weight}(p)}
\]

**Given parameter** \(0 < \beta < 1\), a node is \(\beta\)-exempt if \(\text{max-ratio}(p) > \beta\).

**Dictionary Operations:**
- **find:** as usual
- **insert:** insert as usual but rebuild if needed
- **delete:** delete as usual but rebuild if needed

**When to rebuild?**
- When “backing out” from insert/delete, update node weights.
- Walk down search path from root [opposite from scapegoat!]
- If any node \(p\) is out of balance:
  \[
  \text{balance}(p) > \alpha \\
  \text{and} \\
  \text{max-ratio}(p) \leq \beta
  \]
  then:
  - Rebuild \(p\)
  - Traverse \(p\)’s subtree inorder
  - Collect external nodes in array \(A[0..k-1]\)
  - replace \(p\) with \(\text{buildTree}(A)\)

**Lemma:**
For any set of weighted entries, \(\exists\) an \((\alpha, \beta)\)-balanced \(B\)Tree if
\[
\frac{1}{2} < \alpha < 1 \quad \text{and} \quad \beta < 2\alpha - 1
\]

**Weight-Balanced Trees IV**