Weight-Balanced Trees I

Weight Balance:

- Given a set of keys $X = \{x_0, \dots, x_{n-1}\}$
- and values $V = \{v_0, \dots, v_{n-1}\}$
- and weights $W = \{w_0, \dots, w_{n-1}\}$
- Assume:
  - $x_0 < x_1 < \dots < x_{n-1}$ sorted
  - $w_i > 0$ positivity

Pseudo-Probability:

- Let: $\overline{W} = \sum_{i=0}^{n-1} w_i$ total weight
- Let: $p_i = w_i / \overline{W}$ pseudo-prob
- Obs: $\left.\begin{array}{l} 0 < p_i \leq 1 \\ \sum_i p_i = 1 \end{array}\right\} \Rightarrow$ discrete prob. distribution

Shannon's Theorem:

If p_i is the prob. of accessing x_i, any BST has expected search at least $\sum_i p_i \lg 1/p_i$ ← (called the entropy of distrib)

Overview:

- Splay trees - Static Optimality
- More frequently accessed keys closer to root

$\Rightarrow$ Weight-balanced trees

How to (Nearly) Achieve Shannon's bound

→ Weight-balanced tree

→ For each node p:

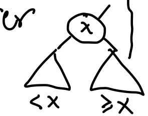
wt(p) = total weight of keys in p's subtree

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

Implementation: (as extended BST)

Internal node: Stores:

- Key key → splitter
- float wt → total weight of entries in subtree
- left, right



External Node:

- Key key, ← x_i
- Value value
- float wt ← w_i

Given $1/2 \leq \alpha \leq 1$, a BST is α-balanced if for all internal nodes p, $\text{balance}(p) \leq \alpha$

- $\alpha = 1/2$: Perfectly balanced
- $= 1$: Arbitrarily bad
- $\alpha = 2/3$: A reasonable compromise

Weight-Balanced Trees II

How to maintain balance?

Options:

- **Rotations**: Similar to AVL trees (single + double) ⋯→ **BB[α] trees**
- **Rebuild subtrees**: Similar to scapegoat

Balance by Rebuilding:

Given an array $A[0..k-1]$ of external nodes:

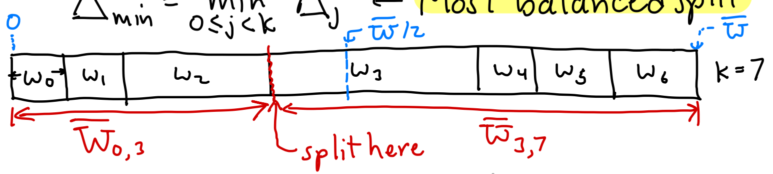
$A[i].key$, $A[i].value$, $A[i].wt$

- Assume keys are sorted
- Assume weights > 0

Weight-based median:

- Select splitter to minimize left-right weight difference
- Let $\overline{W} = \sum_{i=0}^{k-1} A[i].wt$ ← **Total wt**
- Let $\overline{W}_{i,j} = \sum_{m=i}^{j-1} A[i].wt$ ← Total wt of $A[i..j-1]$ (note!)
- Let $\Delta_j = |\overline{W}_{0,j} - \overline{W}_{j,k}|$ ← Absolute diff if we split $A[0..j-1]$ | $A[j..k-1]$
- **Goal**: Split at $0 \le j < k$ that minimizes:

$$\Delta_{min} = \min_{0 \le j < k} \Delta_j \quad \leftarrow \textbf{Most balanced split}$$



Example:

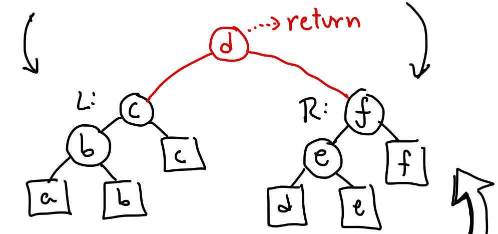
k=6, ideal

	0	1	2	3	4	5
key:	a	b	c	d	e	f
wt:	3	2	4	1	2	4

$\overline{W} = 16$

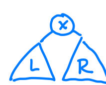
$\Delta_{min} = |(1+2+4)-(3+2+4)| = 2$

$L = buildTree(A[0..2])$ $R = buildTree(A[3..5])$



buildTree (A[0..k-1])

```
if (k == 1) return A[0]            /* base case */
W̄ = Σ_{i=0}^{k-1} A[i].wt           /* total weight */
Init: b = 0; Lwt = 0; Rwt = W̄; Δmin = W̄
for (i = 0 ... k-1)
    Lwt += A[i].wt ; Rwt -= A[i].wt
    Δ = | Rwt - Lwt |              /* weight difference */
    if (Δ < Δmin) { b = i+1;  Δmin = Δ }
L = buildTree(A[0..b-1])
R = buildTree(A[b..k-1])
return new IntNode(A[b].key, L, R)
```



Weight-Balanced Trees III

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 $\leq \log_{3/2} n$, 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)?

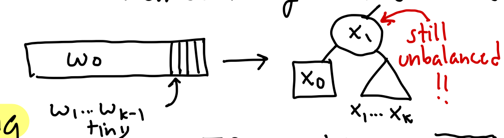
- No. ☹ The optimal BST can be computed by **dynamic programming** (CMSC 451)

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.
  $$\text{balance}(p) > \alpha$$
  Recall def earlier
  Given by designer e.g. $\alpha = 2/3$

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.

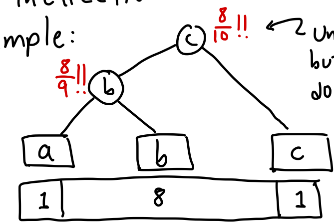
If no such p found – Great! Tree is balanced
Else: **Jackhammer!**

- Traverse p's subtree inorder, store extern nodes in array A[0..k-1]
- Replace p's subtree with

  buildTree(A)

Very heavy entries:

- If an entry's weight is too high, rebuilding is ineffective
- Example:



unbalanced but rebuilding doesn't help!

- This tree is best possible!

- **Exemption**: Don't rebuild if a key's weight is **very high**

For node p: **max(p)** = max weight in p's subtree

max-ratio(p) = $\dfrac{\max(p)}{\text{weight}(p)}$

Given parameter $0 < \beta < 1$, a node is **β-exempt** if **max-ratio(p) > β**

Dictionary Operations:

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

Weight-Balanced Trees IV

(α, β)-balance: Every internal node p is **either** **α-balanced** or **β-exempt**

Lemma: For any set of weighted entries, ∃ an (α, β)-balanced BSTree if $\frac{1}{2} < \alpha < 1$ and $\beta < 2\alpha - 1$

E.g. $\alpha = 2/3$, $\beta = 1/4$

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:

balance(p) > α

-and-

max-ratio(p) ≤ β

then:

- **Rebuild p**:
  - Traverse p's subtree inorder
  - Collect external nodes in array A[0..k-1]
  - replace p with buildTree(A)