2-3 trees

Dictionary: an abstract data type

A container that maps keys to values

Dictionary operations

• Insert
• Search
• Delete

Several possible implementations

• Balanced search trees
• Hash tables
2-3 trees

A kind of balanced search tree
Assume keys are totally ordered (<, >, =)
Assume $n$ key/value pairs are stored in the dictionary
Time per dictionary operation is $O(\log n)$
Support of other useful operations as well
Basic structure: a tree

- Key/value pairs stored only at leaves (no duplicate keys)
- All leaves at the same level, with keys in sorted order
- Each internal node:
  - has either 2 or 3 children
  - has a “guide”: the maximum key in its subtree
Example
Let $h := \text{height of tree}$ \hspace{1em} (Recall: \text{height} = \text{length of longest path from root to leaf})

**Claim:** $n \geq 2^h$

- Proof by induction on $h$
- Base case: $h = 0$, $n = 1 \checkmark$
- Induction step: $h > 0$, assume claim holds for $h - 1$
  - Tree has a root node, which has either 2 or 3 children
  - Each of these children is the root of a subtree, which itself is a 2-3 tree of height $h - 1$
  - By induction hypothesis, if the $i$th subtree has $n_i$ leaves, then $n_i \geq 2^{h-1}$ [here, $i = 1 \ldots 2$ (or 3)]
  - $\therefore \quad n = \sum_i n_i \geq \sum_i 2^{h-1} \geq 2 \cdot 2^{h-1} = 2^h \checkmark$

**Corollary:** $h \leq \log_2 n$
Search(x): // use guides

\( p \leftarrow \text{root of tree} \)
\( h \leftarrow \text{height of tree} \)

repeat \( h \) times
  if \( x \leq p.\text{child0.guide} \) then
    \( p \leftarrow p.\text{child0} \)
  else if \( p.\text{child2} = \text{null} \) or \( x \leq p.\text{child1.guide} \) then
    \( p \leftarrow p.\text{child1} \)
  else
    \( p \leftarrow p.\text{child2} \)

// \( p \) now points to a leaf node

if \( x = p.\text{guide} \) then
  return \( p.\text{value} \)
else
  return null // or some default value
Search Invariant
**Insert(x):** Search for \( x \), and if it should belong under \( p \):

- add \( x \) as a child of \( p \) (if not already present)

If \( p \) now has 4 children:

- split \( p \) into two two nodes, \( p_1 \) and \( p_2 \), each with two children

- process \( p \)'s parent in the same way

- Special case: no parent — create new root, increasing height of tree by 1

Also need to update “guides” — easy

Time = \( O(\text{height}) = O(\log n) \)
Delete(x): Search for x, and if found under p:

remove x

if p now only has one child:

• if p is the root: delete p (height decreases by 1)

• if one of p’s adjacent siblings has 3 children: borrow one
• if none of $p$’s adjacent siblings has 3 children:
  ◦ one sibling $q$ must have 2 children
  ◦ give $p$’s only child to $q$
  ◦ delete $p$
  ◦ process $p$’s parent
Delete 69

(give)

(borrow)

(delete root)
2-3 trees: summary

Assume $n$ items in dictionary

Running time for lookup, insert, delete:
  $O(\log n)$ comparisons, plus $O(\log n)$ overhead

Space: $O(n)$ pointers

Note: in the literature, 2-3 trees usually store the guides in the parent node
  
  - every node contains two guides (the third guide is not needed)

A generalization: $B$-trees

  - allow many children (which makes the height smaller)
  - again, store guides in the parent node
  - useful for high-latency memory (like hard drives)
Augmenting 2-3 trees

Idea: augment nodes with additional information to support new types of queries

Example: store # of items in subtree at each internal node

Queries:
• What is the $k$th smallest item?
• How many items are $\leq x$?
Items may be marked with an attribute, say, “active”/“inactive”

Store a count of active items in subtree at each internal node

Queries:

• What is the $k$th smallest active item?
• How many active items are $\leq x$?
Attribute flipping

- Operation $\text{FlipRange}(x, y)$ flips all attribute bits of items in the range
- Assume attributes are bits
- Store an attribute bit at every node: internal nodes and leaves
  - “effective” value of the attribute is the XOR of all bits on path from root to leaf

\[
1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 1
\]
- To perform $\text{FlipRange}(x, y)$:
  - trace paths $e, f$ to $x, y$
  - flip bits at $x, y$, and all roots of "internal" subtrees
2-3 Trees: Join and Split

\( \text{Join}(T_1, T_2) \) joins two 2-3 trees in time \( O(\log n) \)

Assume \( \max(T_1) < \min(T_2) \)

Assume \( T_i \) has height \( h_i \) for \( i = 1, 2 \)

**Case 1:** \( h_1 = h_2 \)

\[ \text{Time: } O(1) \]
Case 2: $h_1 < h_2$

- Attach $v$ as the left-most child of $p$
- If $p$ now has 4 children, we split $p$, and proceed up the tree as in Insert

Time: $O(h_2 - h_1) = O(\log n)$

Case 3: $h_1 > h_2$ — similar
$\text{Split}(T, x) \iff (T_1 \leq x, T_2 > x)$

Join from inside out
We want to merge $T_1, T_2, T_3, \ldots$ of heights $h_1, h_2, h_3, \ldots$

**Invariant:** $h_i \leq h_{i+1}$ for $i = 1, 2, \ldots,$
and at most 2 trees of any given height — except the first 3 may be of the same height

**Case 1:** $h_1 \leq h_2 = h_3 [ < h_4 ]$

Then computing $T^* = \text{Join}(\text{Join}(T_1, T_2), T_3)$ takes time $O(h_2 - h_1 + 1)$, and $T^*$ has height $h_2 + 1$

Invariant holds for $T^*, T_4, T_5, \ldots$

**Case 2:** $h_1 \leq h_2 [ < h_3 ]$

Then computing $T^* = \text{Join}(T_1, T_2)$ takes time $O(h_2 - h_1 + 1)$, and $T^*$ has height $h_2$ or $h_2 + 1$

Invariant holds for $T^*, T_3, T_4, \ldots$
Example:

0 0 0 1 1 3 4 4 5 6 (case 1)
1 1 1 3 4 4 5 6 (case 1)
2 3 4 4 5 6 (case 2)
3 4 4 5 6 (case 1)
5 5 6 (case 2)
6 6 (case 2)
7

The total cost is $O(t)$, where

$$t \leq (h_2 - h_1 + 1) + (h_3 - h_2 + 1) + \cdots + (h_k - h_{k-1} + 1)$$

$$= h_k - h_1 + k - 1$$

$$= O(h),$$

where $h$ is the height of the original tree

**Conclusion:** total time for Split is $O(\log n)$