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

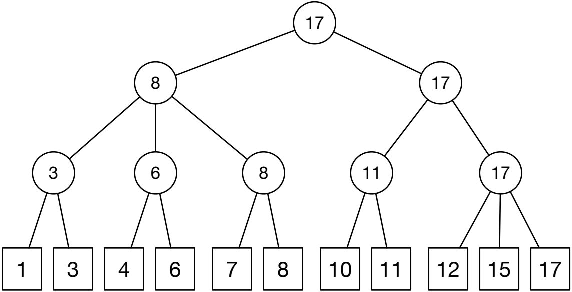
Time per dictionary operation is $O(\log n)$

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 depth (i.e., distance from root)
- looking at the leaves from left to right, keys appear in sorted order
- each internal node:
  - has either 2 or 3 children
  - has a “guide”: the maximum key in its subtree

Example



Let h := height of tree *(Recall: height = max depth of any node)*

Let n := # of leaves

Claim: $n \geq 2^h$

- Proof by induction on h
- Base case: $h = 0$, $n = 1$ ✓
- 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 ith subtree has n_i leaves, then $n_i \geq 2^{h-1}$ [here, $i = 1 \mathinner{..} 2$ (or 3)]
  - $\therefore \quad n = \sum_i n_i \geq \sum_i 2^{h-1} \geq 2 \cdot 2^{h-1} = 2^h$ ✓

Corollary: $h \leq \log_2 n$

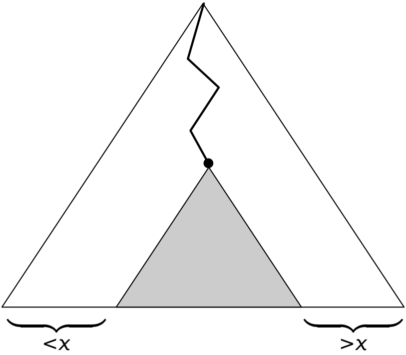
Example Data Layout (Java syntax)

```
class Node {
    KeyType guide;
    // guide points to max key in subtree rooted at node
}

class InternalNode extends Node {
    Node child0, child1, child2;
    // child0 and child1 are always non-null
    // child2 is null iff node has only 2 children
}

class LeafNode extends Node {
    // guide points to the key
    ValueType value;
}
```

```
Search(x):   // use guides to search for the key x
p ← root of tree
h ← height of tree
repeat h times
    // p points to an internal node
    if  x ≤ p.child0.guide  then
        p ← p.child0
    else if  p.child2 = null or x ≤ p.child1.guide  then
        p ← p.child1
    else
        p ← p.child2
// p now points to a leaf node
if  x = p.guide  then
    return p.value
else
    return null   // or some default value
```



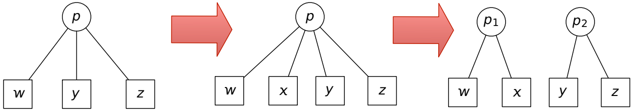
Search Invariant

$\underline{Insert(x)}$: 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 nodes, p_1 and p_2, each with two children



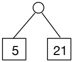
- process p's parent in the same way
- Special case: no parent — create new root, increasing height of tree by 1

Also need to update "guides" — easy

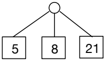
Time = $O(\text{height}) = O(\log n)$

Insert 5

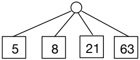
Insert 21

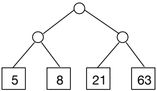


Insert 8

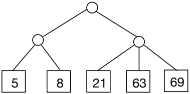


Insert 63

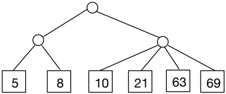


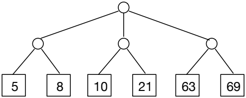


Insert 69

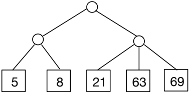


Insert 10

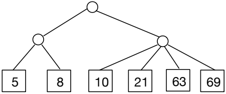


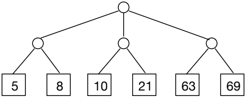


Insert 69

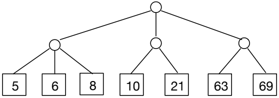


Insert 10

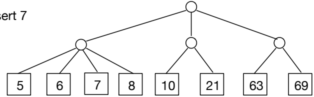


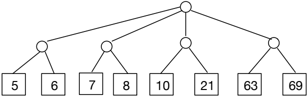


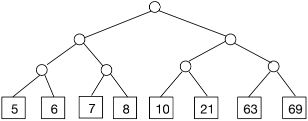
Insert 6



Insert 7





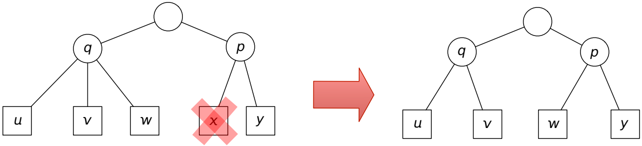


$\underline{Delete(x)}$: Search for x, and if found under p:

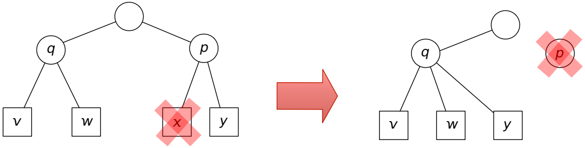
remove x

if p now only has one child:

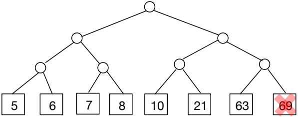
- if p is the root: delete p (height decreases by 1)
- if one of p's adjacent siblings has 3 children: p adopts one



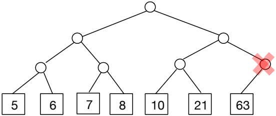
- if none of p's adjacent siblings has 3 children:
  - one sibling q must have 2 children
  - give p's only child to q
  - delete p
  - process p's parent



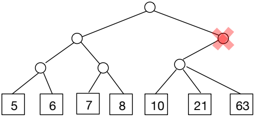
Delete 69



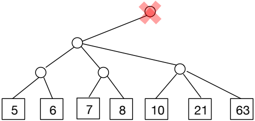
(give)

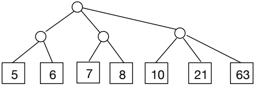


(give)

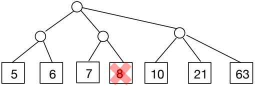


(delete root)

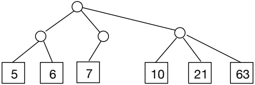


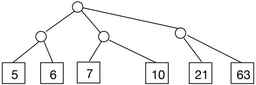


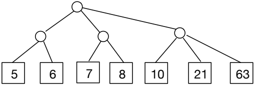
Delete 8



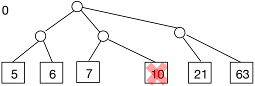
(adopt)



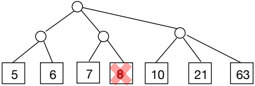




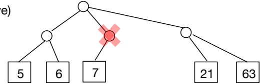
Delete 10



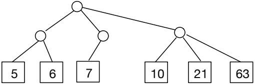
Delete 8

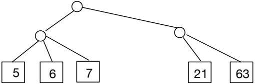


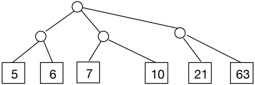
(give)



(adopt)







2-3 trees: summary

Assume n keys 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, “traditional” 2-3 trees usually store the guides *in the parent node*

- every node contains one or two guides

“Traditional” 2-3 Trees

Idea: move the guides into the parent node

```
class Node { }

class InternalNode extends Node {
        KeyType guide0, guide1;
        Node child0, child1, child2;
}

class LeafNode extends Node {
        KeyType key;
        ValueType value;
}
```

<u>*Search*(x)</u>:

$p \leftarrow$ root of tree, $h \leftarrow$ height of tree

repeat h times
        // p points to an internal node
        if $x \le p.$guide0 then $p \leftarrow p.$child0

        else if $p.$child2 = null or $x \le p.$guide1 then $p \leftarrow p.$child1

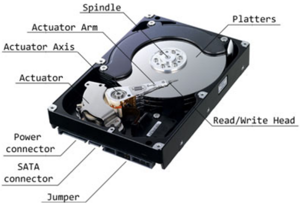
        else $p \leftarrow p.$child2

// p now points to a leaf node

if $x = p.$key then
        return $p.$value
else
        return null

A generalization: B-trees

- allow between B and $2B$ guides in each internal node
- branching factor is at least $B + 1$
- height of the tree is at most $\log_{B+1}(n)$
- example: $B = 2^{10}$ and $n = 2^{30}$, then height is just 3, instead of 30
- useful for high-latency memory (like hard drives)



- many file systems use B-trees to organize their metadata

Augmenting 2-3 trees

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

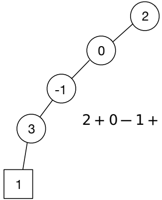
Example: store # of keys in subtree at each internal node

Queries:

- What is the kth smallest key?
- How many keys are $\leq x$?

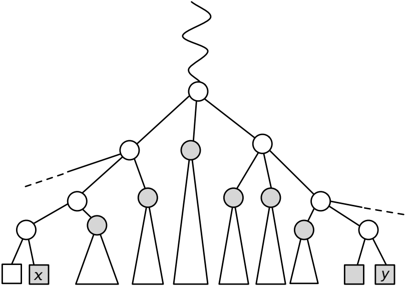
Fast range operations

- Suppose values associated with keys are numbers
- Operation $AddRange(x, y, \Delta)$ adds the same value Δ to the values associated with keys in the range $[x, y]$
- We can do this in time $O(\log n)$ using a "lazy" update technique
  - Store a value field at *every* node: internal nodes and leaves
  - "effective" value associated with a key is the sum all value fields on path from root to its leaf



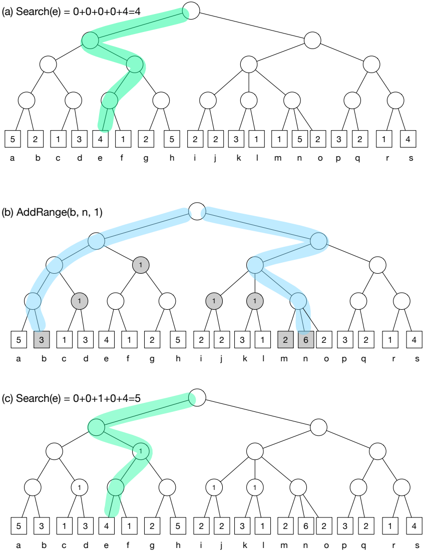
$2 + 0 - 1 + 3 + 1 = 5$

- To perform $AddRange(x, y, \Delta)$:
  - trace paths e, f to x, y
  - add Δ to x, y, and to all roots of "internal" subtrees



- Insert and Delete operations also need to be slightly adjusted

Detailed Example:



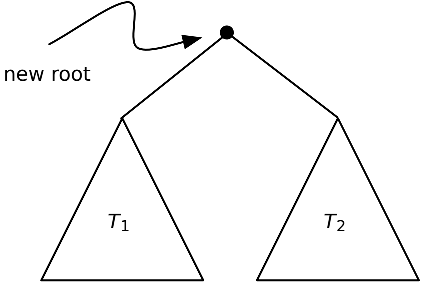
2-3 Trees: Join and Split

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

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

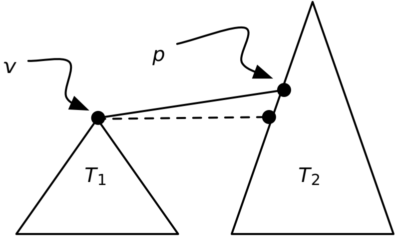
Assume T_i has height h_i for $i = 1, 2$

Case 1: $h_1 = h_2$



Time: $O(1)$

Case 2: $h_1 < h_2$

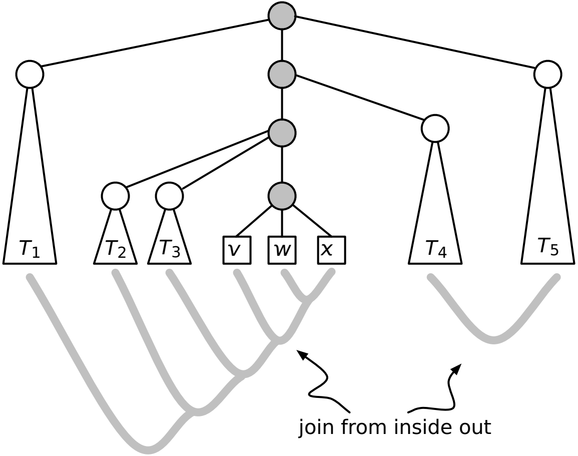


- Attach v as the left-most child of p
- If p now has 4 children, we split p, and proceed up the tree as in *Insert*

Time: $O(h_2 - h_1 + 1) = O(\log n)$

Case 3: $h_1 > h_2$ — similar

$Split(T, x) \implies (T_1\ [\leq x], T_2\ [> x])$



We want to merge 2-3 trees $X_1, \ldots, X_k$ of heights $h_1, \ldots, h_k$:

$$\begin{aligned} Y_1 &:= X_1 \\ Y_2 &:= Join(Y_1, X_2) \\ Y_3 &:= Join(Y_2, X_3) \\ &\vdots \\ Y_k &:= Join(Y_{k-1}, X_k) \end{aligned}$$

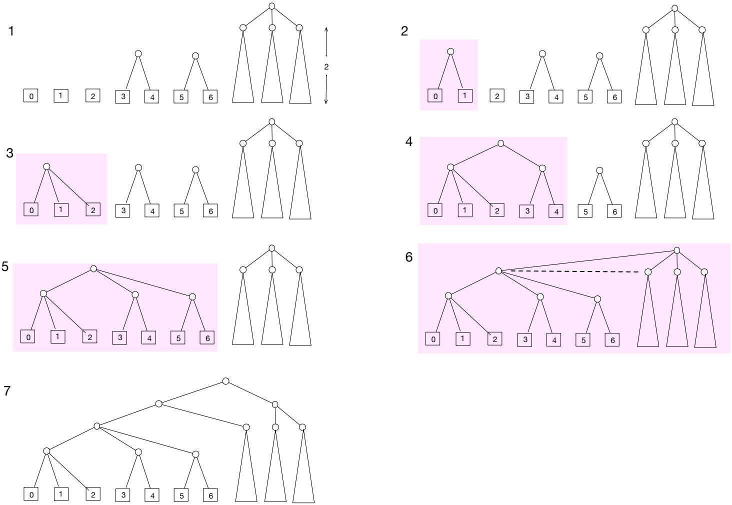
Assumption:

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

Claim: Y_i has height h_i or $h_i + 1$ for $i = 2, \ldots, k$

- *Proof:* See 2-3 Tree Handout

Example:


1
0 1 2 3 4 5 6
2
2
0 1 2 3 4 5 6
3
0 1 2 3 4 5 6
4
0 1 2 3 4 5 6
5
0 1 2 3 4 5 6
6
0 1 2 3 4 5 6
7
0 1 2 3 4 5 6

Claim $\implies$ Time needed to compute

$Y_{i+1} = Join(Y_i, X_{i+1})$ is $O(h_{i+1} - h_i + 1)$

$\therefore$ the total cost is $O(t)$, where

$$\begin{aligned} t \le\ & (h_2 - h_1 + 1) + \\ & (h_3 - h_2 + 1) + \\ & (h_4 - h_3 + 1) + \\ & \qquad \vdots \\ & (h_{k-1} - h_{k-2} + 1) + \\ & (h_k - h_{k-1} + 1) \end{aligned}$$

$$= h_k - h_1 + k - 1 = O(h)$$

and where h is the height of the original tree

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