Algorithms

*Balanced Trees*

2 - 3 Tree

2 – 3 – 4 Tree
2-3 Trees

Features

- each internal node has either 2 or 3 children
- all leaves are at the same level
2-3 Trees with Ordered Nodes

- Leaf node can be either a 2-node or a 3-node
Why 2-3 tree

● Faster searching?
  ■ Actually, no. 2-3 tree is about as fast as an “equally balanced” binary tree, because you sometimes have to make 2 comparisons to get past a 3-node

● Easier to keep balanced?
  ■ Yes, definitely.
  ■ Insertion can split 3-nodes into 2-nodes, or promote 2-nodes to 3-nodes to keep tree approximately balanced!
Why is this better?

- Intuitively, you unbalance a binary tree when you add height to one path significantly more than other possible paths.
- With the 2-3 insert algorithm, you can only add height to the tree when you create a new root, and this adds one unit of height to all paths simultaneously.
- Hence, the average path length of the tree stays close to log N.
Example of 2-3 Tree
What did we gain?

What is the time efficiency of searching for an item?
Gain: Ease of Keeping the Tree Balanced

**Binary Search Tree**

- 60
- 30
- 10
- 20
- 39
- 38
- 37
- 36
- 35
- 34
- 33
- 32

**2-3 Tree**

- 60
- 90
- 80
- 100
- 70
- 40
- 80
- 100

Both trees after inserting items 39, 38, ... 32
Inserting Items

Insert 39

```
  50
 /  \
30   70
|    |  \/   \
10   39|  40   90
|    |      \   \
|    |       60
|    |         \   \
|    |          100
```

Inserting Items

Insert 38

insert in leaf

divide leaf
and move middle
value up to parent

result
Inserting Items

Insert 37

```
               50
              /   
           30    39
          /  
        10    37  38    40
          /  
       60   80  100

```
Inserting Items

Insert 36

insert in leaf

(a)  

(b)  

divide leaf  
and move middle  
value up to parent

overcrowded  
node
Inserting Items

... still inserting 36

divide overcrowded node,
move middle value up to parent,
attach children to smallest and largest

result
Inserting Items

After Insertion of 35, 34, 33
Inserting so far

(a)  
```
P   
  S M L  
```

\[ \rightarrow \]
```
P   
S   \quad \quad L  
\quad n_1 \quad \quad \quad \quad n_2  
```

(b)  
```
P   
   
  S M L  
```

\[ \rightarrow \]
```
P   
S   \quad \quad L  
\quad n_1 \quad \quad \quad \quad n_2  
```
Inserting so far

(a)

(b)
How do we insert 32?
Inserting Items

→ creating a new root if necessary
→ tree grows at the root
Inserting Items

Final Result

(b)

37

33

30

32

34

36

35

39

38

40

50

70

90

10

20

60

80

100
Deleting Items

Delete 70

Swap with inorder successor
Deleting Items

Deleting 70: swap 70 with inorder successor (80)

Swap with inorder successor
Deleting Items

Deleting 70: ... get rid of 70

(b) Delete value from leaf

(c) Merge nodes by deleting empty leaf and moving 80 down
Deleting Items

Result

(e)

```
30
/  \
10  20  40

50
/        /
/  30  40  60
   /  \
   /    /
   /      /
 10      20  40  60

90
/  \
60  80

100
```
Deleting Items

Delete 100

(e)
Deleting Items

Deleting 100

(a) Delete value from leaf
(b) Doesn’t work
(c) Redistribute
Deleting Items

Result

(d)
Deleting Items

*Delete 80*

(d)
Deleting Items

Deleting 80 ...

(a)

Swap with inorder successor
Deleting Items

Deleting 80 ...

(b) 90
   60
   —

Delete value from leaf

(c)
   30
   10 20
   40
   50
   —
   60 90

Merge by moving 90 down and removing empty leaf

Node becomes empty
Deleting Items

Deleting 80 ...

(d) Root becomes empty

Merge: move 50 down, adopt empty leaf’s child, remove empty node

(e) Remove empty root
Deleting Items

Final Result

comparison with binary search tree
Deletion Algorithm I

Deleting item l:

1. Locate node n, which contains item l
2. If node n is not a leaf \rightarrow swap l with inorder successor
   \rightarrow deletion always begins at a leaf
3. If leaf node n contains another item, just delete item l
   else
   try to redistribute nodes from siblings (see next slide)
   if not possible, merge node (see next slide)
Deletion Algorithm II

Redistribution

A sibling has 2 items:
→ redistribute item between siblings and parent

Merging

No sibling has 2 items:
→ merge node
→ move item from parent to sibling
Deletion Algorithm III

Redistribution

Internal node $n$ has no item left
→ redistribute

Merging

Redistribution not possible:
→ merge node
→ move item from parent to sibling
→ adopt child of $n$

If $n$'s parent ends up without item, apply process recursively
Deletion Algorithm IV

If merging process reaches the root and root is without item
→ delete root
Operations of 2-3 Trees

all operations have time complexity of $\log n$