Lecture 13

Today:
- 2-3 Trees
- Search, Insert, Delete
- Other Extensions

Review: Definition of 2-3 Tree (Modified)
1. Internal nodes have 2 or 3 children.
2. Internal nodes have no keys, but store max of each subtree.
3. Leaves have 1 key.
4. (Balance): All leaves at same level.
5. (Search Property):
   \[ \max \text{ (Subtree }_{i-1} \text{ ) } < \min \text{ (Subtree }_i \text{ ) } \]
Review (contd.):

**Fact**: Depth of 2-3 tree $= \Theta(\log n)$

where $n = \#$ keys stored.

Implementation of “node”:

linked list of (child, max) pairs
SEARCH (x, node);

Let \( k_1, k_2 \ldots \) be max of \( \text{subtree}_1, \text{subtree}_2 \ldots \)
if \( k_{i-1} < x < k_i \)

return SEARCH (x, child_i);

Time complexity = \( \Theta(\text{depth}) = \Theta(\log n) \).

PREDECESSOR, SUCCESSOR similar

- But can we really maintain 2-3 tree properties?
- Do 2-3 trees with \( n \) keys exist for every \( n \)?
Key Idea:

- Allow (temporarily) one node to have 4 children. Call this a flaw.
- Initially node 1 level above leaf flawed.
- Repair node $\Rightarrow$ moves flaw up a level in $O(1)$ time
- if root flawed $\Rightarrow$ Repair creates new unflawed root at higher level.
Example Sequence Of Inserts

- Initial tree empty: \( \emptyset \)
- \textbf{INSERT (20)};

\[ \begin{array}{c}
20
\end{array} \]

- \textbf{INSERT (30)};

\[ \begin{array}{c}
20, 30
\end{array} \]

\begin{array}{c}
20 \\
30
\end{array}

- \textbf{INSERT (10)};

\[ \begin{array}{c}
10, 20, 30
\end{array} \]

\[ \begin{array}{c}
10 \\
20 \\
30
\end{array} \]
- INSERT (25);

```
10, 20, 25, 30
```

```
10  20  25  30
```

```
20, 30
```

```
10, 20
```

```
25, 30
```

- INSERT (45);

```
20, 45
```

```
10  20
```

```
25, 30, 45
```

```
10  20
```

```
25  30  45
```

- INSERT (8);

```
8, 10, 20
```

```
8  10  20
```

```
20, 45
```

```
25, 30, 45
```

```
8
```

```
10
```

```
20
```

```
25
```

```
30
```

```
45
```
. INSERT (15);
**Pseudo Code**

- **INSERT** \((x, \text{root})\);  
  - **SEARCH** \((x, \text{root})\) to find parent, of leaf
  - where \(x\) must be inserted;
  - go up from node to root
  - **REPAIR** \((\text{node})\);
  - **RECOMPUTE** \((\text{max})\);

- **REPAIR** \((\text{node})\);
  - if node has 4 children;
    - Split into node₁, node₂ with 2 children each;
    - Make both children of parent(\(\text{node}\));
DELETE

Key Idea: Build subroutine to delete entire subtree at interior node.

Case 1: Node has 3 children ⇒ No problem
Case 2: Node’s parent has $\geq 5$ grandchildren;

(example)
Case 26: parent has only 4 grandchildren ⇒ move problem up one level.

```
parent
  /  
-node
     / 
   -A   B
     /   /
   C     D

new
  parent
  /  
new node
  /  
A   C   D

Dummy
```

Delete this subtree
Example

DELETE (15);  Case 2a

20, 45

8, 10, 20

8, 10, 20

8, 10
Suppose orig. tree binary
Conclusions

- Can maintain balanced trees with INSERT / DELETE, with $\Theta(\log n)$ depth.
- 2-3 Trees: Pros / Cons:

  Pro's: Conceptually simple;
  No need to memorize rotation sequence.

  Con's: Not binary;
  Many real cases;
  Coding hard

  => Exist binary trees as well...