B-trees and 2-3-4-trees

Last Time
- Binary Search Tree
  Insert, Delete, Search, Min, Max, Successor, Predecessor \( \rightarrow \Theta (n) \)
- \( n \) random inserts build BST with height \( \Theta (\sqrt{n}) \) on avg
  Adversary can still prepare non-random data that leads to unbalancing trees and thus long execution times approaching \( \Theta (n) \)

Idea this time:
Rebalancing to guarantee height \( \Theta (\log n) \) deterministically using local rebalancing transformation

One useful operation: Rotations

\[
\begin{array}{c}
\text{right-rotate} \\
\text{left-rotate}
\end{array}
\]

Time:
\( \Theta (1) \)
because and involves mostly
  changing pointers locally
  (Not that all parent-child relationships change)

\[
\begin{array}{c}
\text{left-rotate} \\
\text{left-rotate}
\end{array}
\]

\[
\begin{array}{c}
\text{not very helpful.}
\end{array}
\]
Questions:

1. What sequence of rotations causes?

2. What is the running time of such an algorithm on a linear tree of length $n$?

3. What is the worst-case running time of an algorithm that, after every insertion or deletion, converts BST into heapsorted array/linear tree and applies rotations as above to create balanced tree?

Attempt at Global Solution
Another solution

Red-Black Trees (CPS chpt 13)

- AVL trees
- k-neighbors trees
- 2-3 trees
- B-trees
- 2-3-4 trees
- Splittree

Enforce reasonably well-balanced tree (difference in longest & shortest path from root to leaf is 1-fold for red-black tree). Often, if subtree insert/delete, rebalance using rotation or other operations

\[ \Theta(\lg n) \text{ operations because } h = \Theta(\lg n) \]

2-3-4 trees use a different idea:

- Relaxed balance constraint
  - Allow nodes to have 2, 3, or 4 children
  - Force all leaves to be at same depth
  - Nodes with \( c \) children \( c(\leq 4) \) store \( c-1(\leq 3) \) keys to facilitate search
  - Leaves can store up to 3 keys also

Example:

```
Tree property:
- Each node is a successor
- Each child node has keys intermediate in value between pair of elements in parent node.
```

```
Example of 2-3-4 tree:

- Diagram showing tree structure with keys at each node.
```
Search for 12
- like binary search, but multi-way

Insert 5+8

Insert 5+28

Insert 5+38

Cannot have 4 elements in a node.
Most "split" this node, which includes 1 insertion into parent.

Insert 0
- cause double split
Ideas

- Every node branches until reach leaves all of same level
  - prevents very uneven (unbalanced) trees
- Height is \( O(\log n) \)
- Variability in branching factors leads to flexibility
  for fast upgrades
  - worst case involves split at every level
  of tree and insertion of new
  root above current root \( \mathcal{O}(k) = \mathcal{O}(\log n) \)

More formal and general class of threes to which B-trees belong:

B-trees, with parameter \( t \geq 2 \) (Case of \( t=2 \) \( \Rightarrow 2-3-4-1 \) keys)

- Every non-leaf node has \( \geq t \) and \( \leq 2t \) children
  (except root, which has \( \geq 2 \) children)
  (leaf has 0 children)
- Each non-leaf node stores one key inbetween
  every adjacent pair of children
- \# keys = \# children - 1
  \( \geq t-1 \) and \( \leq 2t-1 \)
  - This key bound is enforced on leaves, as well

\[ \text{e.g. } \begin{array}{c}
  x \\
  A \\
  y \\
  B \\
  C \\
\end{array} \Rightarrow \text{all keys of } A \leq x < y \leq \text{all keys of } B \leq y \leq \text{all keys of } C \]

Lemma: Height of B-tree = \( \mathcal{O}(\log t n) = \mathcal{O}(\log n) \)

Proof:
- # leaves \( \leq n \)
- branching factor \( \geq t \)
- height \( \leq \log n + 1 \)
Search

- Visit nodes in root-to-leaf path
  - At each node
    - examine all keys
    - if desired key found, done
    - else, find where desired key would fit among
      the ordered keys and follow that pointer

\[ \text{Example: } \text{Search}(A) \]
\[ k_1 < k_2 < k_3 \]
\[ \text{Time: } \Theta(t) \text{ to visit a node } \quad \text{binary search case} \]
\[ \text{height } \Theta(\log_2 n) \quad \text{number of levels} \]
\[ = \Theta(t \log_2 n) \quad \rightarrow \quad \Theta(\log n) \quad \text{for } t=O(1) \]

Insert (This is where things start to get interesting)

- find leaf where new key belongs (using search)
- if leaf has fewer than \(2t+1\) keys, then reuse them and add
  new key to leaf, keeping keys in sorted
  order \[ \rightarrow \Theta(t) \text{ time } \rightarrow \text{ may need to shift data} \]
- else \[ \text{leaf is full} \]
  - insert new key into left or right half \[ \rightarrow \text{overflow} \]
  - split node into left, median, right

\[ \begin{array}{c}
2t-1 \\
\hline
2t \\
\hline
2t+1 \\
\hline
2t+2
\end{array} \quad \rightarrow \quad \begin{array}{c}
2t \\
+1 \\
1 \\
\hline
2t+1
\end{array} \]
Time for insert: \( O(\log n) \), same as search

Analysis: as events in splits

Delete: Worst case \( O(\log n) \)
- if key is not in a leaf, replace it with successor (which is in leaf)
- now just details few key
- remaining key found
- now just details few leaf
- if leaf still has \( \geq t-1 \) keys, then done
- else [underflow]
- \( \geq 2 \) tricks here

1) Try to steal from siblings
- if an adjacent sister has \( > t-1 \) keys, then shift through parent

Maintain balance property

2) If adjacent siblings have only \( t-1 \) keys (close to underflow), then merge with one of them and parent key.
- Essentially reversing a split

- This can lead to underflow in parent and require propagation upward to removal of root node.
External Memory

Between read to exploit cache & disk, in practice

Model: Can read/write B elements stored together in one block transfer

- [Diagram showing block transfer]

Goal: minimize # block transfer

Let $t = B$ in B-tree

Search/Insert/Delete are $O(\log_B n)$ block transfer

(If this is optimal for this problem)

CLRS has more to say about this.