Balanced Search Trees

Binary Search Tree review
B-Trees
2-3-4 (or 2-4) Trees
Applications

Dynamic Set Data Structures

<table>
<thead>
<tr>
<th>D.S.</th>
<th>Insert</th>
<th>Delete</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Table</td>
<td>$O(1)$ exp</td>
<td>$O(1)$ exp</td>
<td>$O(1)$ exp</td>
</tr>
</tbody>
</table>

Why do we need anything else?

Fast search + find-min?
  - Run heap & hash table in parallel
  - Pointers between corresponding elements in two structures (e.g., to support delete-min)

In exact Searches
  - Library search: Intro to Algorithms
  - Intro to Alkaloids
  - Nearest lexical match
  - Hashing insufficient, sorted array too slow for updates
Binary Search Trees (BSTs)

BST property: All nodes in left subtree of node \( x \) have keys \( \leq \) key at \( x \)
\[ ... \text{ right } ... \geq \text{ key at } x \]

In order traversal produces sorted array (go left till you can't, backtrack, go right)

Search \( (T, k) \)
- Start at root of tree \( T \)
- If \( k \leq \text{ key [current node]} \)
  - then go to left child
- else go to right child
- If hit node with key = \( k \), stop & return node
- If hit nil, stop & return nil.

Running time \( O(h) \): \( h \) is height of tree.
Use similar strategy for insert & delete \( O(h) \)

Problem: \( h \) can be \( O(n) \)! linked list!
Balanced BSTs

Guarantee $O(\log n)$ height by modifying Insert & Delete

Red-Black Trees, AVL trees, treaps use rotations

Right-Rotate

Left-Rotate

Depth (A) reduced
Depth (C) increased

<< tricky, messy, will avoid rotations >>

2-3-4 Trees also called 2-4 Trees

- Relax binary constraint: allow up to 4 children
- Force leaves to be at the same level
  \[ \Rightarrow O(\log n) \text{ height} \]
- Nodes with $c \leq 4$ children, store
  $c-1 \leq 3$ keys for search
- Leaves store up to 3 keys
Example

B-Tree Definition

B-tree with parameter \( t \geq 2 \) \((t=2 \Rightarrow 2-3-4\) tree\) must have these properties:

- Every non-leaf node has between \( t \) and \( 2t \) children, except root which has between \( 2 \) and \( 2t \) children.
- Each non-leaf node stores one key in between every adjacent pair of children.
  \[ \Rightarrow \text{ # keys } = \text{ # children } - 1 \]
  \[ \Rightarrow \text{ # keys } \text{ is in between } t-1 \text{ and } 2t-1 \]
- Require key bound of leaves as well.
- Search tree property
  all keys in subtree \((\text{left})\) of a key are \( (\leq) \) that key.
  \[ \text{e.g. } \begin{array}{ccc}
    & x & y \\
  A & B & C
  \end{array} \]
  keys \( A \leq x \leq \text{keys} \leq B \leq y \leq \text{keys} \leq C \)
**Lemma:** \[ \text{height of } b\text{-tree } = O(\log_t n) \]
\[ = O(\lg n) \]

**Proof:**
- \# leaves \( \leq n \)
- branching factor \( \geq t \) except at root
- \( \Rightarrow \) height \( \leq \log_t n + 1 \)
- Subtree of root

Search \((T, k)\)
- visit nodes in root-to-leaf path
- at each node:
  - compare all keys to \( k \)
  - if \( k \) found, return
  - else follow pointer between predecessor and successor

**Time:** \( O(t)/\text{node} \)
\[ O(\log_t n) \text{ height} \]
\[ \Rightarrow O(t \log_t n) = O(\lg n) \text{ for } t = O(1) \]
B-Tree Insert Example

Insert(16)

Insert(18)

Unchanged on right side

Illegal

Promote median
Insert Example (contd.)

Insert(2)

DONE!
B-tree Insert

- Find leaf where new key belongs (via search)
- Insert new key into leaf: sort ($\Theta(t)$ time)
- If leaf is now overflowing (2t keys)
  - split node into left half, median, right half

\[
\begin{array}{c}
\text{2t} \\
\rightarrow \\
\text{t} \\
\text{t-1}
\end{array}
\]
- promote median up to parent node
  - if parent now overflowing, recursively split
  - if root splits, create new root with 2 children (incrementing total height).

Time: again $\Theta(t \log_t n)$
at worst $h$ splits
B-Tree Delete

- If key not in leaf, replace it with its successor (which is in a leaf)
- Now just have to delete from leaf.

Delete from leaf:

- remove key from leaf
- if leaf not underflowing (still \( \geq t-1 \) keys), done
- else \( \leq 2 \) tricks instead of one:
  1. try to steal from siblings
     - if an adjacent sibling has \( \geq t-1 \) keys then shift through parent
  2. if adjacent siblings have only \( t-1 \) keys (close to underflow), then merge
     with one of them, and parent key

If parent underflows, recurse

Time: some modification per level \( \Rightarrow \Theta(t \log_t n) \).
Applications

B-trees used extensively in databases.
With very large $t$ (in the thousands or more!)

A node fills a disk-block (secondary storage). Disk-access is SLOW!
Operations require reading $O(\log t)$ blocks from disk, which is small for large $t$.

Operations on the block/node in main memory are fast ($O(t)$) but reading from main memory and checking for keys is very fast in modern processors.)