Today: Balanced Search Trees
- 2-3 trees
- Search, Insert, Delete
- (a,b)-trees
- B-trees
- red-black trees

Recall: Balanced Binary Search trees
(e.g. AVL trees)
- guarantee $O(lg n)$ height
  by rebalancing during Insert/Delete
- typically using rotations:
2-3 trees: balanced (nonbinary) search tree
- Force leaves to be all at the same depth & all nonleaves to branch ($\geq 2$ children)
$\Rightarrow$ height $\leq \lg n$ (denser than perfect BST)
- Achieve by relaxing binary constraint
- Allow nonleaf $x$ to have $c \in \{2, 3\}$ children: $x$.child[1], ..., $x$.child[c]
- $x$ also stores $c-1 \in \{1, 2\}$ keys: $x$.key[i], one key between every 2 consecutive children
- Leaves store 1 or 2 keys too

Example:

Search tree property:
- Keys in node are in sorted order
- Keys in subtree left of key $k$ are $\leq k$
- Keys in subtree right of key $k$ are $\geq k$

keys in $x$.child[i] $\leq x$.key[i] $\leq$ keys in $x$.child[i+1]
**Search** $(x, k)$:
1. Initially, $T.root$
2. Search for key $k$ in $x.keys$
3. If $x.key[i] = k$:
   - Return $x.key[i]$
4. Else if $x.key[i] < k < x.key[i+1]$
   - Return $\text{Search}(x.child[i+1], k)$
   - If nil then absent

**Time:** $O(\log n)$

**Example:**

![Binary Search Tree Example](image)

- Predecessor, Successor, Inorder Traversal similar to BSTs
**Insert** \( (T, k) \):
- Search \( (T, \text{root}, k) \) for leaf \( x \) where \( k \) fits
- insert key \( k \) into \( x\.keys \), keeping sorted
- while \( x \) is overflowing: \( (\text{has 3 keys}) \)
  - split node into left half, median, right half
- if \( x\.parent \neq \text{nul} \): \( (x \text{ isn't root}) \)
  - promote median up to \( x\.parent \),
    inserting into \( x\.parent\.keys \)
else:
  - create new node \( T\.root \)
    with one key: the median
    & two children: left half & right half
- \( x = x\.parent \)

Time: \( \mathcal{O}(\log n) \)
Example:

1. Insert(60)
   - 30
   - 10 17
     - 3 7 14 20 24
     - 33 42
   - 37 45
     - 50
   - 60

2. Insert(70)
   - 30
   - 10 17
     - 3 7 14 20 24
     - 33 42
   - 37 45
     - 50
   - 60
   - 70
   - OVERFULL!

3. OVERFULL!

4. viola!
Delete \((T,k)\):
- Search \(T,root, k\) \(\rightarrow x.key[i]\)
- if \(x\) is not a leaf:
  - Successor \((x,k)\) \(\rightarrow y.key[j]\)
  - \(x.key[i] = y.key[j]\)
  - \(i \times x = j \times y\)
- Delete \(x.key[i]\), keeping sorted
- While \(x\) is underflowing: (has no keys)
  - Try to steal from siblings:
    - If an adjacent sibling has extra keys: \(\geq 1\)
      - Shift a key through \(x\).parent
      - Steal a child subtree from \(y\)

... continued on next page...
Delete \((T, k)\):

- else steal from parent:
  - if adjacent sibling(s) have no extra keys:
    - merge with a sibling & intervening parent key

\[
\begin{array}{c}
\text{AT MIN.} \\
A \\
\end{array}
\quad
\begin{array}{c}
\text{AT MIN.} \\
\rightarrow
\end{array}
\quad
\begin{array}{c}
\text{potential underflow} \\
\text{AT MIN.} \\
A
\end{array}
\]

- special case: if \(x, \text{parent} = \text{nil} \) (root)
  then just delete node \(x\) & set root of tree to \(x\)’s only child

\[
\begin{array}{c}
x \\
A \\
\end{array}
\quad
\begin{array}{c}
\rightarrow
\end{array}
\quad
\begin{array}{c}
A
\end{array}
\]

- \(x = x, \text{parent}\)

[end while loop]

Time: \(O(\log n)\)
\((a, b)\)-trees: \(2 \leq a \leq b\) (soon: \(b = 2a - 1\))

- Internal nodes have \(c \in [a, a+1, \ldots, b] \) children
  
  \((\& c - 1\ \text{keys})\)

- Search tree property & Search identical

- Insert: overflowing = \(> b - 1 \) keys

  \[ \left\lfloor \frac{b - 1}{2} \right\rfloor \] left keys, \(1\) median, \(\left\lceil \frac{b - 1}{2} \right\rceil\) right

  \(< b \Rightarrow \text{no overflow} \iff < b\)

- Need \(\left\lfloor \frac{b - 1}{2} \right\rfloor \geq a - 1\) so no underflow

  \(\iff b \geq 2a - 1\)

- Delete: underflowing = \(< a - 1 \) keys

  \(\text{EXCEPT root can have down to 1 key}\)

  - If \(x\) has \(a - 2\) keys, sibling has \(a - 1\)

    Then merge has \(2a - 2 \leq b - 1\)

    \(\Rightarrow \text{no overflow}\)

- Search costs \(O(\log_a n \cdot \log b)\)

  \[ = O\left(\frac{\log n}{\log a} \cdot \log b\right) = O(\log n) \text{ if } a \geq b^\varepsilon\]

- Insert/Delete costs \(O(\log_a n \cdot b)\)

  \(= O(\log n) \text{ if } b = O(1)\)

  Or use balanced search tree for a node

  \(\Rightarrow O(\log n)\)
B-trees = (⌈\frac{B}{2}\⌉, B) - trees
\implies one parameter instead of two
- Search \(O(\log n)\)
- Insert/Delete \(O(\log n)\) if \(B = O(1)\) or BST nodes

Motivation: caches read whole blocks of data
- want entire block to be useful
\implies set \(B = \text{block size}\)
\implies O(\log_B n) \text{ block reads/Search/Insert, ... (optimal in comparison model)}

Typical cache size:
- 1KB for RAM
- 1MB for disk

10x
100x

Used by most databases & most file systems

- Sleepycat/Berkeley DB
- MySQL
- SQLite

- MacOS HFS/HFS+
- ReiserFS
- Linux ext3, shmfs
- Windows NTFS
2-3-4 trees \(=\) (2,4) trees

Red-black trees: represent 2-3-4 trees as BST with red & black nodes: [Bayer 1972]

- root always black
- never two red nodes in a row
- black depth of every leaf is the same
- # black ancestors
- see CLRS [Ch. 13] for Insert & Delete in this form (implemented with rotations)