Key Based Search/Retrieval

2 Big ideas

- Hashing

- keys not thought of as ordered

But memory (or disk) address numbers can be computed from a key

Binary search in an ordered set of keys

(> more than binary)

(> like in B-trees)

(> also Multi-dim. Search (advanced) topic!)
**Binary Search Tree**

**General Search Tree**

\[ n = x \cdot n \]

- Node \( x \)
  - \( K_1, K_2, K_i, K_{i+1}, K_n \)
  - \( L_{C_1}, L_{C_2}, L_{C_i}, L_{C_{i+1}} \)
  - \( all < K_1 \)
  - \( K_i < all < K_{i+1} \)
  - \( all > K_n \)

Every key in left subtree \(< K\)
Every key in right subtree \(> K\)

Pay attention to relationship between \(K_i, C_i, C_{i+1}!\)

Different nodes \( x \) can have different \( n = x \cdot n \)

---

**Balance**: a different concept

- **Height** \( H = \max \) of steps from the root to a leaf

\[ H = O(\log N) \]

\( \rightarrow \) tree (family) is Balanced

---

**Balanced Search Tree**

**Search Problem**:
- Input: Balanced search tree \( T \) and key \( k \)
- Output: If \( k \) is in \( T \), output is a node with key \( k \). Otherwise, nil.
An sorted array used for old-fashioned binary search can be thought of as a tree!

```
if ($k < A[5]$)
```

etc
B-Tree-Search(x, k)

1  i = 1
2  while i <= x.n and k > x.key[i]  
   {  Find spot where key might be.
   loop 1-3 ended become k <= x.key[i]  
3  i = i + 1
4  if i <= x.n and k == x.key[i]  
5  return (x, i)  
   found the key!
6  elsif x.leaf  
7  return nil  
8  else Disk-Read(x.c[i])
9  return B-Tree-Search(x.c[i], k)

This algorithm solves the above problem in O(log N) time, where N = # nodes in T.
So computer people invented dozens (?) of ways to insert more wisely, so the tree remains balanced.

Idea 1:

\[
\min N(H) = \begin{cases} 
1 & \text{if } H = 0 \\
1 + \min N(H-1) + \min N(H-2) & \text{else}
\end{cases}
\]

\[
H_R = H_L \pm 1 \\
H_L \text{ or } H_R
\]

"AVL" tree

Do the Math!

\[
\min N(H) \times C \text{ Fibonacci } \#(H)
\]

\[
= \Theta \left( \left( \frac{1 + \sqrt{5}}{2} \right)^H \right)
\]

\[
\frac{\min N(H) + 1}{= (\min N(H-1) + 1) + (\min N(H-2) + 1)}
\]

Name this \( F(H) \)

\[
F(H) = F(H-1) + F(H-2)
\]

\[
\begin{align*}
F(0) &= \min N(0) + 1 = 2 \\
F(1) &= \min N(1) + 1 = 3
\end{align*}
\]

\[
\min N(H) = \text{ Fibonacci } \# - 1
\]
Idea 2: Force the leaves to have uniform depth. Force each non-leaf node to have 2 or more kids. (except root)

But the # of children of each node vary.

Earliest follow-through of this idea: 2-3 tree

\( k_1, k_2 \)

\( k_1, k_2, k_3 \)

\( k, k_1 \)

\( k_1, k, k_2 \)

B-tree

Given \( t \)

Between \( t-1 \) and \( 2t-1 \) keys in each node

\( t, \ 2t \)

\# children of non-leaves is between \( t \) and \( 2t \) (except for the root)
B-trees:

Lesson from this case study:

- A small amount of extra work done now pays off BIG in the future! (To maintain property 5 of B-trees for example.)

Example: Dynamic array resizing by doubling (by multiplying by \(1 + c\)) (amortized)

N things at the end of the day only \(O(N)\) time was used.
B-tree Definition

1. \( x \): node with \( n \) \( \geq 4 \) keys
   - \( x \).key_1, \ldots, \( x \).key_\( n \)
   - \( x \).leaf

2. \( x.c_1, x.c_2, \ldots, x.c_{(n+1)} \)
   - Undefnied if \( x.leaf = \) false

Programmers might work hard to implement two versions of a node: one with the child pointers and another, for leaves, without child pointer space.

3. \( k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \)
   \( k_i \leq x.key_i \leq \ldots \leq k_n \)
   - for all keys \( k_i \) in subtree rooted at \( x.c_i \)

B-tree HW:

18.2-1 18.2-3 18.2-6
18.3-1 - subtle
Due Thursday as usual
4. All leaves have the same height, between \((t-1)\) and \((2t-1)\).

5. (except root might be tiny...)

4.5 Guarantee the tree is balanced:

\[
\text{Height} = \Theta(\log \# \text{keys})
\]

Why except the root from the \(t-1\) minimum?

1. So the first few keys can be inserted.

2. When a new root is created, which happens when the old root is split, the new root can have only 1 key.
Simple search tree insertion algorithm idea:

Given $k$ and $T$, do the search algorithm. If $k$ is not found, insert $k$ in the empty spot where the search ended.

FSQKCLHTVWMRNPAUBYDZE

Cleverness: When a node is full (max max # keys = $2t - 1$) the # keys is odd ($2t - 1$ is always odd)
Cleverness: When a node is full ($2t \times \text{max \#keys} = 2t - 1$), the #keys is odd ($2t - 1$ is always odd).

$2^t - 1 = 7$
$2^t = 8$
$t = 3$

$PQRSTUV$

To make more room in $SPLIT$ it

$SPLIT$ it

$2t - 1$ keys
$2t - 2$ w/o $S$
$(2t - 2)/2 = t - 1$

$N\cdot SW$

$PQR$

$TUUV$

(Suppose the key to insert is $V^*$)

$V < V^* < W$
Bug: What if the parent is full?
HA!

1. Don't insert \( \sqrt{n} \) into node \( x \) unless \( x \) is not full.
   \[ \text{INSERT}(x, k) \quad \text{(Don't call unless the parent of} \ x \text{ is not full)} \]

2. If insert into root, and root is full:
   \[ \text{root = new node} \quad \text{(Just one key...)} \]
Clever application of today's theme:

Only delete if \( x \) or its descendants if

\[ \begin{align*}
\delta(x.n) & \geq t \quad \Rightarrow \quad \text{not}\ t - 1!
\end{align*} \]

symbol for this clever idea.

1. \( k \) is in node \( x \) and \( x \) is a leaf

(because clever inductive assumption 😊)

\( x \) has \( t \) or more keys,

delete \( k + 1 \).

2. \( k \) is \( \leq x \) and \( x \) is not a leaf

\( \delta(x.n) \geq t \)

- Delete \( k \), largest

\( \delta(x.n) \geq t \) key in or under \( x \), \( \delta(x.n) \geq t \)

- Replace \( k \) by \( k' \)

in \( x \).

2b) \( y.n = t - 1 \) and \( z.n \geq t \)

handle like 2a)
2. \( k \) is an element of \( x \) or \( x \) is not a leaf.

If \( \mu \geq t \), then \( k \) is in \( x \).

Case 3) \( k \) is not in \( x \).

Find \( i \) so subtree rooted at \( x, ci \) would contain \( k \).

- If \( x, ci \) has \( t \) or more keys, recurse on \( x, ci \).
- If \( x, ci \) has only \( t-1 \) keys, do complicated steps 3a) or 3b.

Combine

\( \mu = 1 \), \( (t-1) + (t-1) = 2t - 2 \leq 2 \mu - 1 \)

OK!
3b) an immediate sibling of \( x, c_i \) has to move keys only t-1 keys.

Move a key from \( x \) down to \( x, c_i \) and a key from a sibling of \( x, c_i \) up to \( x, c_i \).

\[
3(t-1) + (t-1) + 1 = 2t - 1
\]