**Dynamic Finger Theorem:**

Keys: $x_1, \ldots, x_n$. We perform accesses $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$. Let $\Delta_j = i_j - i_{j-1}$: distance between consecutive items.

**Thm:** Total access time is $O(m \cdot n \log n + \sum_{j} (1 + \log \Delta_j))$.

---

**Static Optimality:**

- Suppose key $x_i$ is accessed with prob $p_i$. ($\sum_i p_i = 1$)
- Information Theory: Best possible binary search: tree answers queries in expected time $O(H)$ where $H = \sum_i p_i \log \frac{1}{p_i} = \text{Entropy}$

**Static Optimality Theorem:**

Given a seq. of $m$ ops. on splay tree with keys $x_1, \ldots, x_n$, where $x_i$ is accessed $q_i$ times. Let $p_i = q_i/m$. Then total time is $O(m \cdot \sum_i p_i \log p_i)$.

---

**Splay Trees are Amazingly Adaptive!**

---

**Balance Theorem:** Starting with an empty dictionary, any sequence of $m$ accesses takes total time $O(m \cdot \log n + m \cdot \log n)$ where $n = \max$ entries at any time.

---

**Splay Trees III**

---

**Analysis:**

- Amortized analysis
- Any one op might take $O(n)$
- Over a long sequence, average time is $O(\log n)$ each
- Amortized analysis is based on a sophisticated potential argument

---

**Potential:** A function of the tree's structure

- Balanced $\Rightarrow$ Low potential
- Unbalanced $\Rightarrow$ High potential
- Every operation tends to reduce the potential
What does entropy have to do with search times?

Keys: a, b, c, d, e, f, g

Access Probability: $\frac{1}{9} \quad \frac{1}{16} \quad \frac{1}{16} \quad \frac{1}{8} \quad \frac{1}{16} \quad \frac{1}{32} \quad \frac{1}{32} \Rightarrow \sum p_i = 1$

$= \frac{1}{2} \quad \frac{1}{2} \quad \ldots \quad \frac{1}{2^n}$

The higher the exponent the deeper the key should be.

Extended Binary Search Tree
(keys stored in leaves)

Depth:

0

1

-2

3

4

"Ordered" Huffman tree

Keys of prob $p_i = \frac{1}{2^i}$

Best depth

at depth $i=1$ $H = \sum p_i \log \frac{1}{p_i}$

$= \log \frac{1}{p_i} = \log \frac{1}{2^i} = i$

Entropy:

$H = \sum p_i \log \frac{1}{p_i}$
Multiway Search Trees:

- Most large data structures reside on disk storage
- Organized in blocks/pages
- Latency: High start-up time
- Want to minimize no. of blocks accessed

B-Tree:
- Perhaps the most widely used search tree
- 1970 - Bayer & McCreight
- Databases
- Numerous variants

B-Tree: of order \( m \) \( ( \geq 3 ) \)
- Root is leaf or has \( \geq 2 \) children
- Non-root nodes have \( \lceil \frac{m}{2} \rceil \) to \( m \) children [null for leaves]
- \( k \) children \( \Rightarrow k-1 \) key-values
- All leaves at same level

Example: \( m = 5 \)

\[ \log_2^n = \frac{1}{2} \cdot \frac{n}{\log_2^m} \]

Class BTreeNode:

\[
\begin{array}{c}
\text{int nChild} // no. of children \\
\text{BTreeNode child}[M] // children \\
\text{Key, key}[M-1] // keys \\
\text{Value, value}[M-1] // values
\end{array}
\]

Theorem: A B-tree of order \( m \) with \( n \) keys has height at most \( (\log_2 n)/\gamma \), where \( \gamma = \log(m/2) \)

(See full notes for proof)
**Key Rotation (Adoption)**
- A node has too few children \([m/2] - 1\)
- Does either immediate sibling have extra? \(\geq [m/2]+1\)
- Adopt child from sibling & rotate keys
- When applicable - preferred

**Node Splitting:**
- After insertion, a node has too many children \(m+1\)
- We split into two nodes of sizes \(m' = [m/2]\) and \(m'' = m+1-[m/2]\)

**Lemma:** For all \(m \geq 2\),
\([m/2] \leq m+1-[m/2] \leq m\)
\(\Rightarrow m' + m'' \) are valid node sizes

**B-Tree Restructuring:**
- Generalizes 2-3 restructure
- Key rotation (Adoption)
- Splitting (insertion)
- Merging (deletion)

**B-Trees II**

**Node Merging:**
- A node has too few children \(\lceil m/2 \rceil - 1\)
- Neither sibling has extra (both \(\lceil m/2 \rceil\))
- Merge with either sibling to produce node with \(\lceil m/2 \rceil - 1 + \lceil m/2 \rceil \) child

- \(j + 3 = 5\)
- \#keys: \(1 + 2 = 3\)
**Insertion:**
- Find insertion point (leaf level)
- Add key/value here
- If node overfull (m keys, m+1 children)
  - Can either sibling take a child (<m)?
  - Key rotation [done]
  - Else, split
    - Promotes key
    - If root splits, add new root

**Example:** \( m = 5 \)

**Deletion:**
- Find key to delete
- Find replacement/copy
- If underfull (\( \lceil m/2 \rceil - 1 \) child)
  - If sibling can give child
    - Key rotation
  - Else (sibling has \( \lceil m/2 \rceil \))
    - Merge with sibling
    - Propagates
      - If root has 1 child, collapse root

**Example:** \( m = 5 \)
\( \text{B}^+ \text{-Tree:} \) Widely used!

- Store all data (key+value) **only in leaves**
- Internal nodes store keys (but only for finding leaf)
  - Note: Keys in internal nodes are not necessarily in the dictionary
- Leaves are linked using next-leaf pointer

**Range Query: \((a, b)\)**

- Report all keys \(x\), where \(a \leq x \leq b\)
- Find leaf containing \(a\) (or its successor)
- Using next-leaf, list until reaching \(b\)

Ex. \((m = 3\), but each leaf can store 3 keys\)

\[
\begin{array}{ccccccc}
\end{array}
\]

Since internal nodes only store keys, greater fan-out \(\Rightarrow\) fewer disk accesses

28 is not in dictionary
It is just used to find actual data in leaves