**Persistent Data Structures:**
- Preserves prior states of data structure
- Allows searches in history
  \( \text{Let } T_{z} \text{ be new subtree} \)
- Full/Partial Persistence:
  - Change can be made to any/current version

**Example:** Rebuild subtree \( T_{1} \) at time \( t \):

\[
\text{BJ Tree: } \quad \text{PBJ Tree: }
\]

\( T_{1} \text{ to be unlinked} \)

**Approach:** Whenever a modification made - save old contents & use temporal node to distinguish

**Change at \( t \):**
- Old
- New

**Case study:** PBJ Tree Persistent Weight-Balanced Jackhammer trees
- A partially persistent BJ tree
- Uses rebuilding to balance subtrees
- Allows weighted entries

**Copy-on-write:** Whenever you modify, make a copy
- Very inefficient!

**Change-log:** Make a list of recent updates
- Slow to process

**Fat nodes/Node copying:**
When changes occur, save just modified portions

**Temporal node:** New node type.

- Time
  - Pre
  - Post
- Subtree contents prior to time \( t \)
  - Less than subtree contents on or after time \( t \)

**Example:**

- \( 01 \) insert IAD
- \( 03 \) insert BWI
- \( 05 \) insert SFO
- \( 07 \) delete BWI

**Find IAD at \( 00 \):**
- Not found
  - Get-min at \( 04 \): BWI
  - Get-min at \( 08 \): IAD

**History:**
- Driscoll, Sarnak, Sleator, Tarjan (1986) - First serious theoretical analysis
- Applied to geometric search (time is coordinate)

**Approaches:**

- Copy-on-write: Whenever you modify, make a copy
  - Very inefficient!
- Change-log: Make a list of recent updates
  - Slow to process
- Fat nodes/Node copying:
  - When changes occur, save just modified portions
  - Slow to process
- Temporal node:
  - New node type.

**Types:**
- Pre
- Post

**Content:**
- Subtree contents prior to time \( t \)
  - Less than subtree contents on or after time \( t \)
PBJ Tree (Private) Data:
- `final float ALPHA, BETA`
- Node root
- `root → init: null`
- `int firstInsertTime` → `init: -1`
- `int lastInsertTime`

Insertion (without rebalancing): Node structure:
- `Node: weight + maxWt (float)`
- `TempNode: time (int) pre, post (Node)`
- `IntNode: key, left, right`
- `ExtNode: key, value`

- Before insert, check that `t > lastInsertTime`
  → else Error!
- Set `lastInsertTime ← t`
- Note: Partial persistence
- Rebalance → Next

Persistent Search Trees II

Example:
- `IntNode: wt ← left. wt + right. wt` maxWt ← `max(left. maxWt, right.)`
- `Temp Node: wt ← post. wt` only `post!`

Updating Weights?
- `IntNode: left ← left. insert(...)`
- `right ← right. insert(...)`
- `Temp: post ← post. insert(...)`
- Update node weight + maxWt (later)

Few More Things:
- For `insert`, check that `t > lastInsertTime`
- Set `lastInsertTime ← t`
- Note: Partial persistence
- Rebalance → Next
**Internal Node Rebalance:**
- Test α, β condition (same as BJ tree)
- If unbalanced, compile list A of external nodes for current tree
- \( T = \text{original tree} \)
- \( T' = \text{buildTree}(A) \)
- return:

```
  T
  \text{ \textbullet \rightarrow \textbullet }
\```

**Temporal Node Rebalance:**
- Recurse on post side:
  \( \text{post} = \text{post\_rebalance}(x, t) \)
- On return:
  - if (post child is temporal)
    - perform left rotation
    - update weights
  return root of subtree

Why? Don't like long post chains

---

**Rebalancing after insertion:**
- Starting at root, retrace search path
- Recursive helper:
  \( \text{Node rebalance}(x, t) \)
  - Internal: Apply to left/right based on \( x \to \text{key} \)
  - Temporal: Apply to post only.

**Persistent Search Trees**

---

**Example:**

---

**find(x, t), findUp(x, t), getMin(t):**
- same as before but for temporal nodes visit pre/post based on \( t \)
- check whether \( t < \text{first UpdateTime} \)
  → if so → null

**getPreorderList(t)/getFullPreorderList()**
- First gets preorder list at time \( t \)
  - no temporal nodes!
- Second gets full preorder list (all nodes)

**Delete/Clear:** Not implemented/
Not required! (A bit messy)

---

**Shallow or deep copy?**
- We did deep
  → Does it matter?