Binary Search Trees

Binary Tree: A rooted tree, each node having a left and a right child, either or both missing.

Binary Search Tree: Each node contains an item. Items are totally ordered and arranged in the tree in symmetric order: all items in left subtree are less, all items in right subtree are greater.

Binary search trees support access, insert, delete in $O(\text{depth})$ time.
A binary search tree

Diagram:

```
  in
 /  
as  that
|    /
and I  of
|    /
|    to
a  for  is
|  /
|  /
for it
|  /
|  /
|  /
  go
```
Classical answer: Maintain a (local) balance condition.

Two properties:

(i) Implies $O(\log n)$ depth of an $n$-node tree.

(ii) Easily restoreable after an update: $O(\log n)$ time by rebalancing along access path.

Since ~1962 many kinds of such balanced search trees have been discovered.
A Rotation

Changes depths of some nodes

Takes $O(1)$ time (3 pointer changes)

Preserves symmetric order
A Binary Search Tree

Red/Black:
1) All blacks on paths,
2) Red nodes have black parents

Items in internal nodes, in symmetric order:
   items in left subtree smaller,
   items in right subtree larger.

Allows binary search for items
   search time = 1+ depth.
Red-black tree updates

- black
- red

Insert: root → .

recolored

possibly nonterminating

-
Persistent Search Trees

How do we preserve old versions of tree, allowing queries in the past, updates in the present (and possibly in the past)?

Objective: Avoid copying the entire tree. (takes O(n) space and time per update)

Applications

Text editing*

Applicative programming languages*

Computational Geometry
Obtaining (Partial) Persistence in Search Trees

Easy Solution: Copy the entire access path and all changed nodes during each update.

Reps, Krijnen and Meertens, Verhulst... Myers, Swart
initial tree: A, C, E, G, I, K, M, O
iJ, dM
Two Insertions
time is logarithmic per access step
but
space is old per update step

Version stamps:

Uses binary search on the
structure

Navigation through the structure:

a field name and a version stamp:

Each new value stored in a node gets

Allow nodes to become arbitrarily "fat"

(parallel persistence)

Ideas in our Construction
Improving the Time Bound

Allow only a fixed amount of extra space in a node.

When a node becomes full, create a new copy, with newest pointers.

Node copying can proliferate, but amortized time and space is $O(1)$ per update step.

Amortized bound: copying a node consumes a full node, creation of full nodes is $O(1)$ per update step.
Where is amortization used?

To prove space bound of \(O(1)\) per update.

Needs only an amortized \(O(1)\) bound on the structural update time in the ephemeral (non-persistent) data structure.

Red-Black Trees give \(O(1)\) space bound per insert/delete.

\(\Phi = \#\text{filled extra slots in live nodes}\)
Computational Geometry

2-D point location

The post office problem: Given n points in the plane, answer queries of the form, given a new point, to which old point is it closest?
Given point A and your plane, the shortest distance can be
plane, contain a plane containing such line, the

The point given: Given n points in the

2-3 point located

A application to computational geometry
Planar Point Location

Plane Sweep: dimension reduction - one space dimension becomes a time dimension

... in data structures - persistent search trees
Orthogonal Line Intersection

Data lines

Query line

Search in space

Search in time