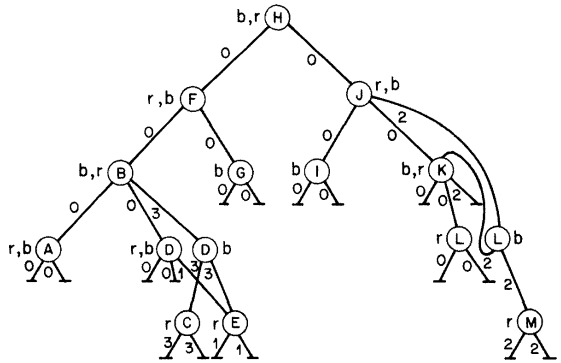
Persistent Search Trees



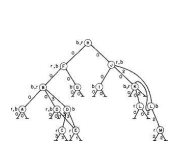
Presentation from the article
Planar Point Location Using Persistent Search Trees
of Neil Sarnak and Robert E. Tarjan

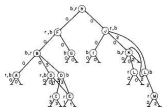
Olivier Pirson

INFO-F413 *Data structures and algorithms*

ULB

December 8, 2016
(Some corrections November 26, 2017)
Last version:
`https://bitbucket.org/OPiMedia/persistent-search-trees/`





Binary search trees

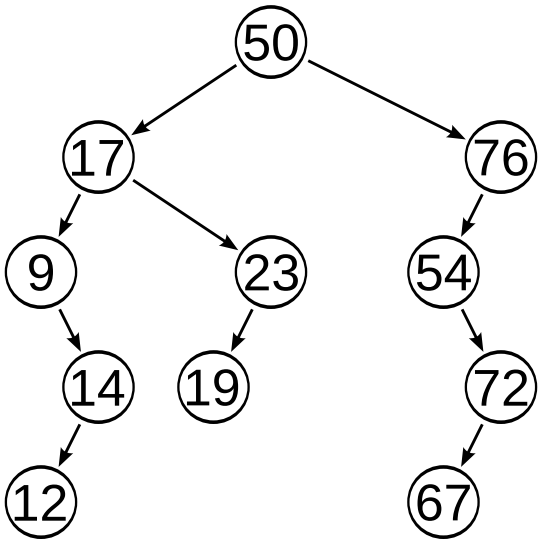
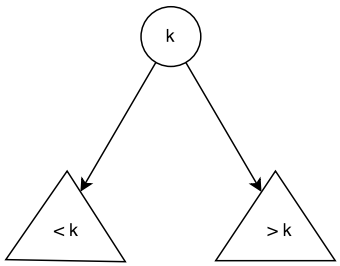
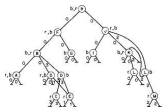


Figure: Intgr, Wikipedia

- Each node contains a **key** (a value, and in general an associated data).
- All keys in the left subtree are **less than** the key's root.
- All keys in the right subtree are **greater than** the key's root.
- And recursively.





Binary search trees

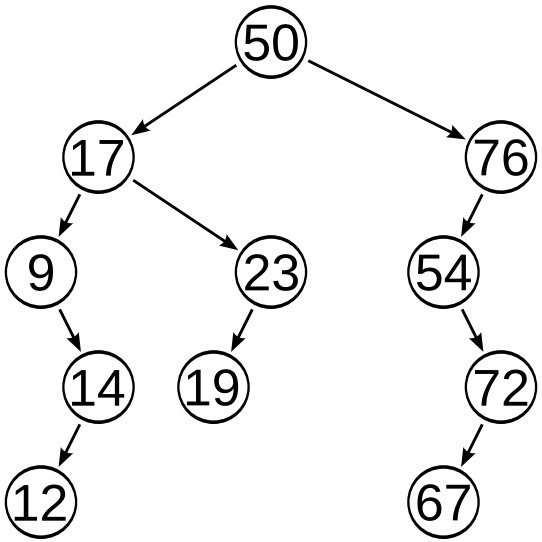
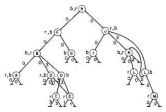


Figure: Intgr, Wikipedia

A binary search tree constructs **a set** and provides these operations:

- `access(x)`: find and return the item with the greatest key less than or equal to x (or a NIL value if doesn't exist). So if x is in the tree, then return the item with x.
- `insert(x)`
- `delete(x)`



Binary search trees

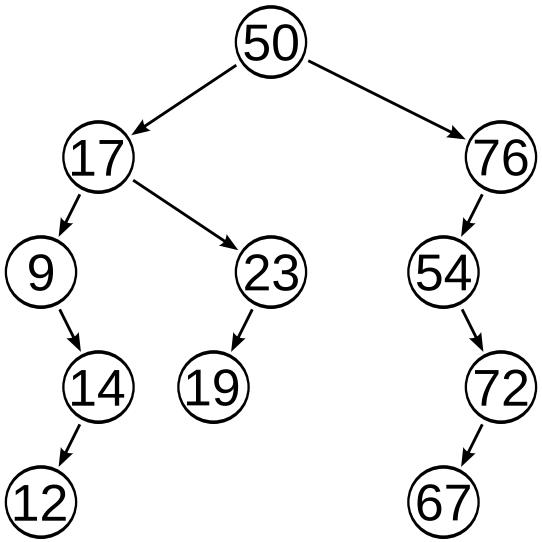
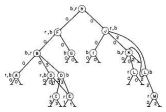


Figure: Intgr, Wikipedia

The problem with this tree...



Balanced binary search trees

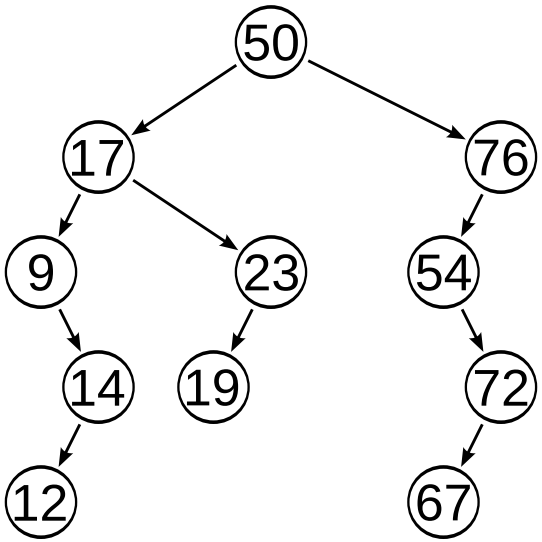


Figure: Intgr, Wikipedia

height of the tree $\in O(n)$, so

- `access(x)`: $O(n)$ in time
- `insert(x)`: $O(n)$
- `delete(x)`: $O(n)$

in worst case
(n = size of the tree = number of nodes)

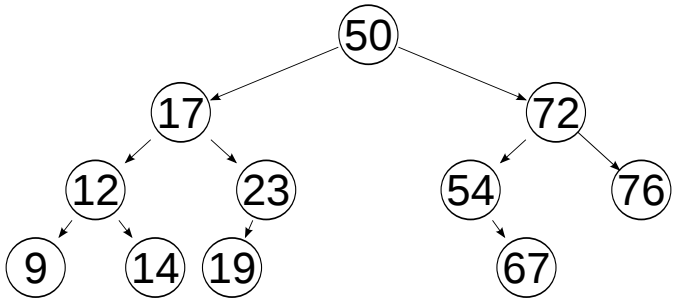


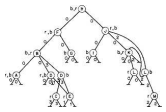
Figure: Mikm, Wikipedia

With a **balanced** binary search tree:
height of the tree $\in O(\log n)$, so

- `access(x)`: $O(\log n)$
- `insert(x)`: $O(\log n)$
- `delete(x)`: $O(\log n)$

And $\Theta(n)$ for space.

(Of course, all these complexities depend on the implementation, but it is possible.)



Red–black trees

One way to ensure a **good balancing** and have **good complexities**:

- add extra-information in each node
- rearrange after each modification (with some specific local rotations)

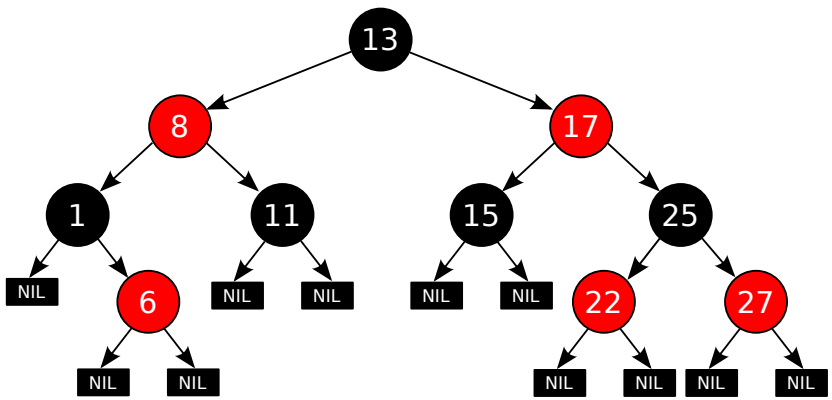


Figure: Cburnett, Wikipedia

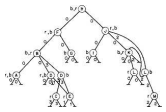
(All NIL can are an unique sentinel.)

Red–black trees:

(The type of binary search trees used in the article.)

- A color red or black for each node (in fact 1 bit of information).
- Add (pseudo)-leaves NIL.
- Some constraints on colors:
  - every leaf (NIL) is black
  - children of red node are black
  - all descending path contain same number of black nodes

These constraints ensure a **height** in $O(\log n)$, with some rotations and recoloring when we insert or delete.



Red–black trees

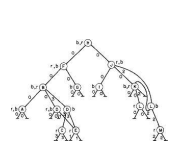
Insertions and deletions require

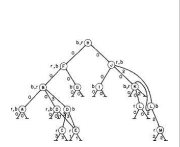
- only $O(1)$ rotations
- and $O(\log n)$ recoloring
  (in worst case, and only $O(1)$ in amortized case).

In summary,

with some requirements, we have a **balanced binary search tree** with:

- Operations in $\mathsf{O}(\log \mathsf{n})$
- and space in $\Theta(\mathsf{n})$.



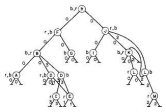


Volatile data structures

If we modify these kind of data structures,
we **lost the previous versions**.

Those are **volatile** data structures.

In general, it is exactly what we want.
But not always.



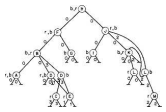
Persistent data structures

A **persistent** data structure, it is a data structure that **preserve** all old versions after any modification.

It is also an **immutable** data structure.
That is the old structures are never modified.
(From an external point of view. Maybe the internal data are modified, but is not visible.)
Instead the structure is modified in place; a new updated structure is build.

These two notions are close.

- *Persistence* is about all the new updated structure,
- and *immutability* is about the old not modified structure.



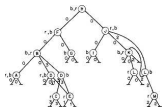
And now… a digression!

Immutable data structures are a foundation of **functional paradigm** languages (like Lisp, ML, Haskell, Scala… and progressively more and more other languages add functional aspects).

It was my motivation to choose this subject. I would like more understand immutable data structures. (Maybe soon, I will understand how deal with immutable graphs!)

I think it is an **important paradigm**, and it will more important in the **future**.

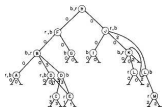
- First, because it have a mathematical elegance. It is important. ☺
- But mostly because our computers today, and more after, must be use multiple cores and for that programs must become parallelized programs.



Trivial and stupid way

Go back to the persistence.

How build a persistent data structure?



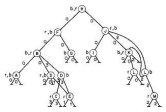
Trivial and stupid way

Go back to the persistence.

How build a persistent data structure?

Copy all the current version, and apply the modification on the copy.
It works.

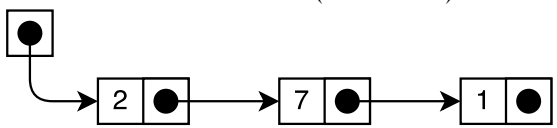
But it is inefficient! Waste time and space.
So, it does *not* works.



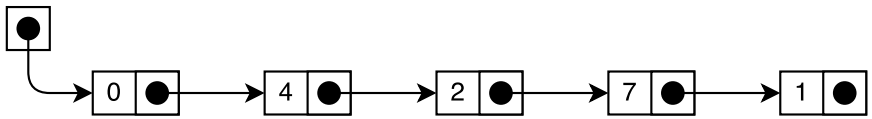
Linked-list example

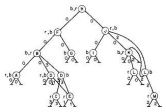
I will show you on a linked-list a better idea
and after that we will do the same with binary search tree.

Start with a list $(2, 7, 1)$



And push front 4, and next 0. We obtain a new list, $(0, 4, 2, 7, 1)$

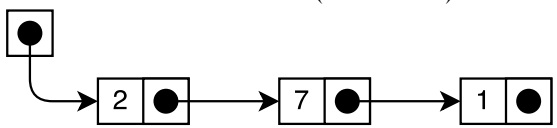




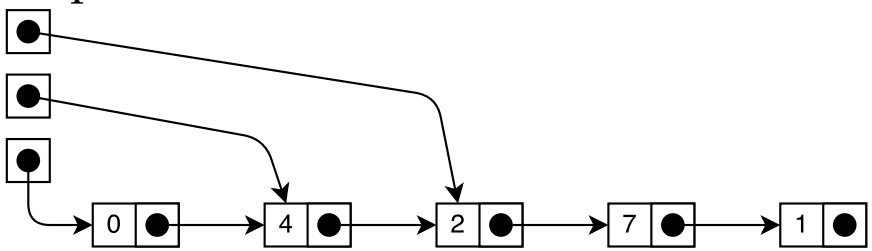
Linked-list example

I will show you on a linked-list a better idea
and after that we will do the same with binary search tree.

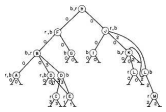
Start with a list $(2, 7, 1)$



And push front 4, and next 0. We obtain a new list, $(0, 4, 2, 7, 1)$

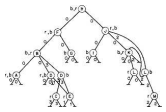


If we preserve links to previous versions,
we have a persistent data structure.



Very simple

And now… let's do that on a binary search tree…



Persistent search tree with path copying

Persistent red–black tree with **path copying**.

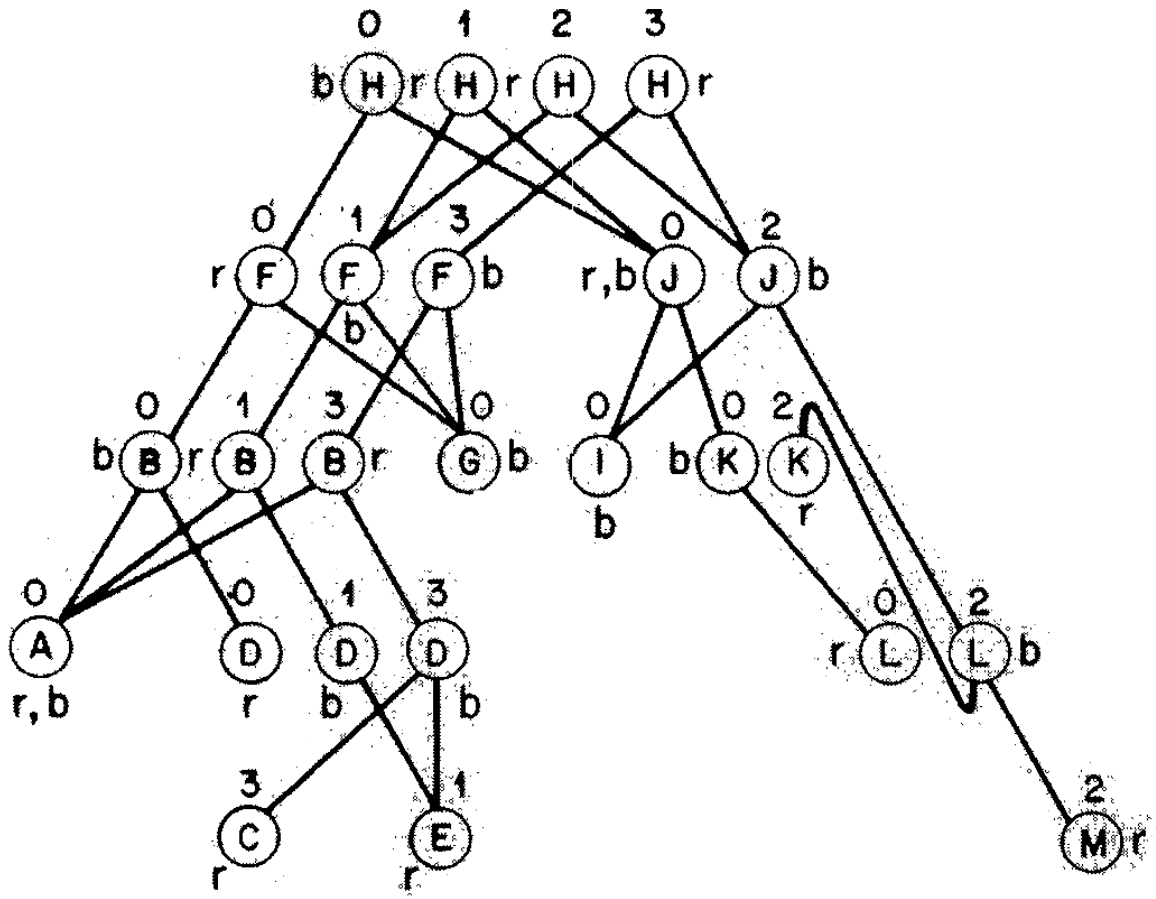
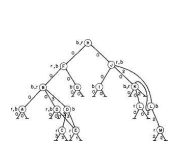


Figure: Figure 6 of Neil Sarnak, Robert E. Tarjan (Ref. 28)



Persistent search tree with path copying

We have now a notion of **time**. We can **access** to current tree, but also to all **past trees**.

- `access(x, t)`
- `insert(x)`
- `delete(x)`

Only the current tree is modifiable.

And each modification implies a **path copying**.

Persistent red–black tree with **path copying**.

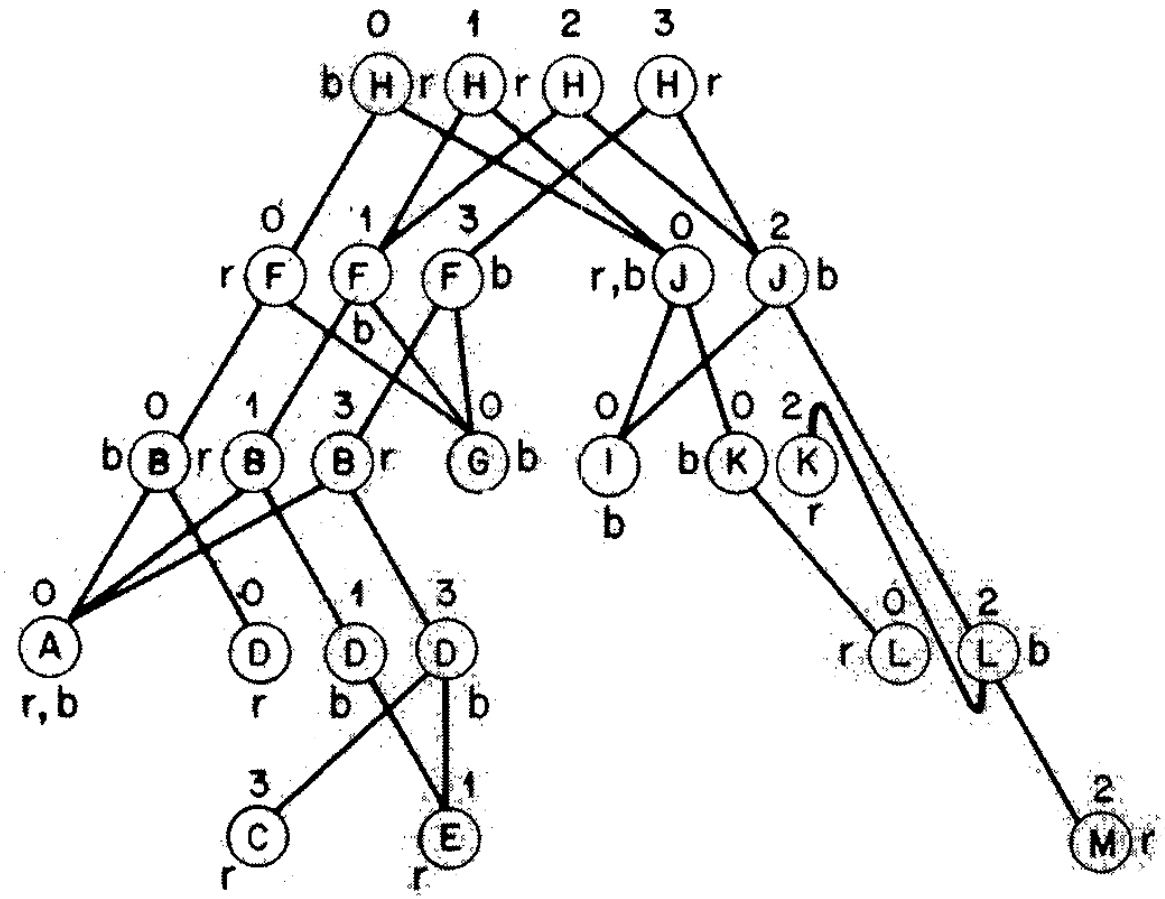
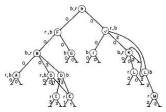


Figure: Figure 6 of Neil Sarnak, Robert E. Tarjan (Ref. 28)



Persistent search tree with path copying

Persistent red–black tree with **path copying**.

- Restart from time = 0, with A, B, D, F, G, H, I, J, K and L in the tree.

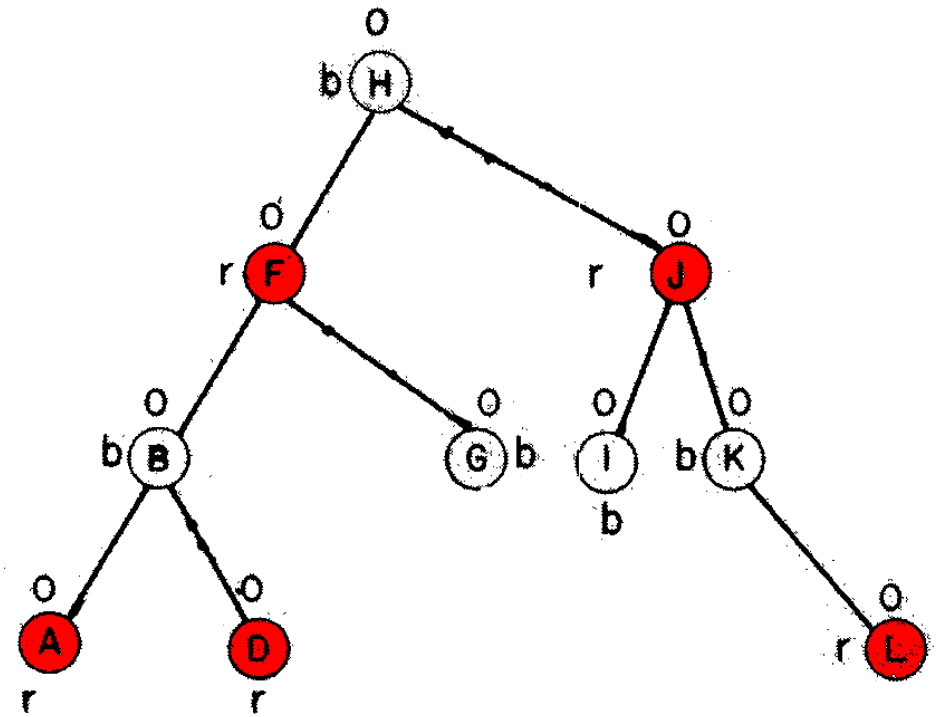
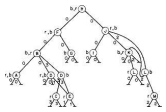


Figure: Partial figure 6 of Neil Sarnak, Robert E. Tarjan (Ref. 28)



Persistent search tree with path copying

Persistent red–black tree with **path copying**.

- Restart from time = 0, with A, B, D, F, G, H, I, J, K and L in the tree.
- Add E, in the time 1.

  Note that J was changed of color. (Colors are only used for update, so they useless for past version.)

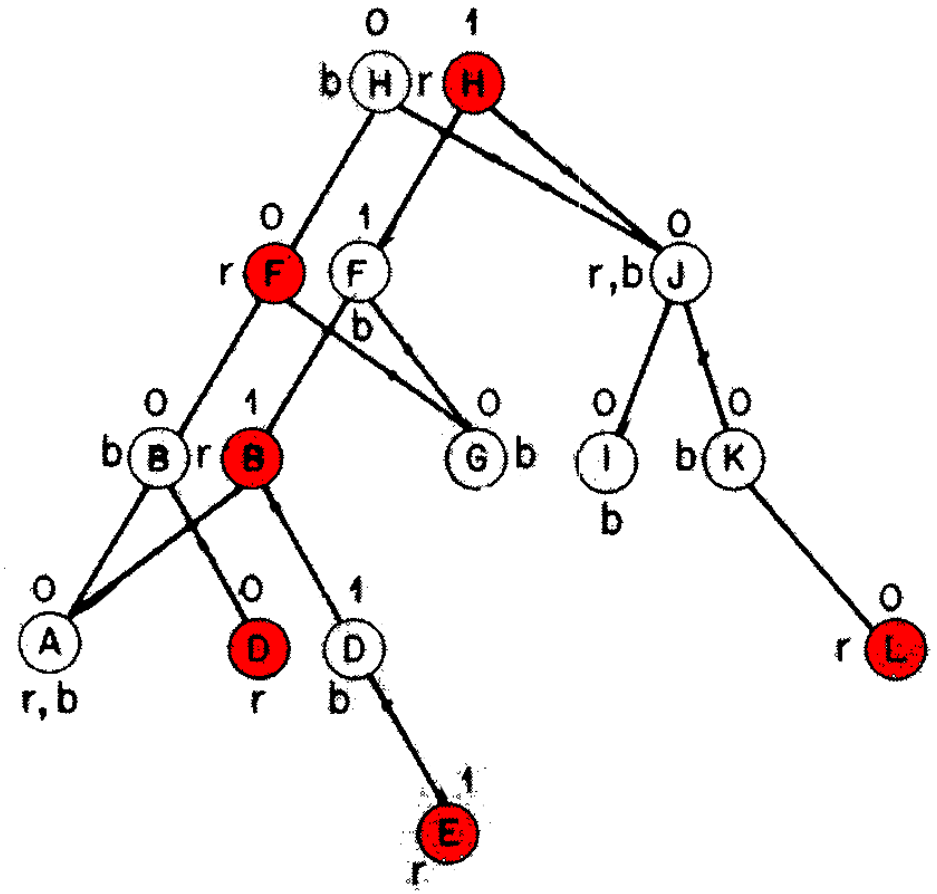
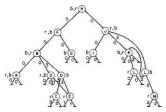


Figure: Partial figure 6 of Neil Sarnak, Robert E. Tarjan (Ref. 28)



Persistent search tree with path copying

- Restart from time = 0, with A, B, D, F, G, H, I, J, K and L in the tree.
- Add E, in the time 1.

  Note that J was changed of color. (Colors are only used for update, so they useless for past version.)
- Add M, in the time 2.

Persistent red–black tree with **path copying**.

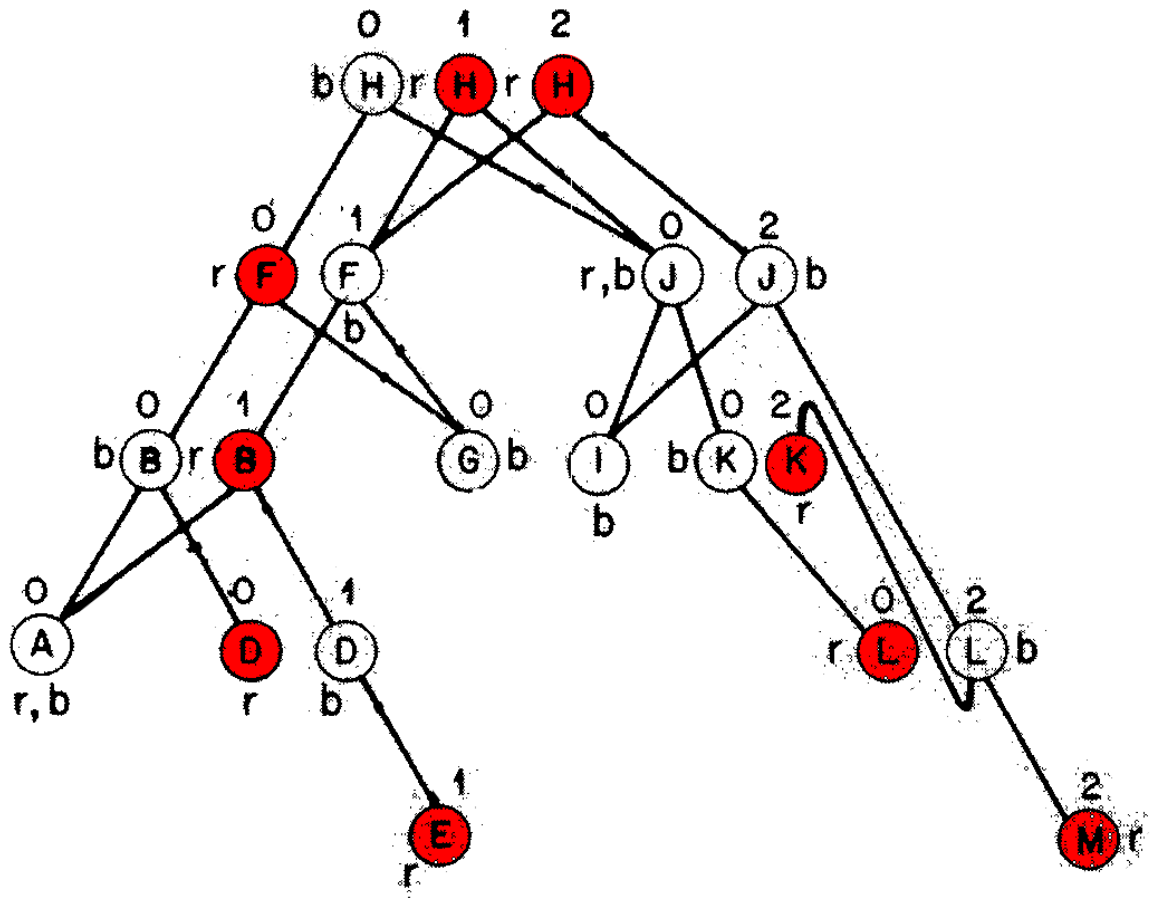
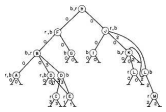


Figure: Partial figure 6 of Neil Sarnak, Robert E. Tarjan (Ref. 28)



Persistent search tree with path copying

- Restart from time = 0, with A, B, D, F, G, H, I, J, K and L in the tree.
- Add E, in the time 1.

  Note that J was changed of color. (Colors are only used for update, so they useless for past version.)
- Add M, in the time 2.
- Add C, in the time 3.

We have preserved the **O**(log n) **complexity of operations**.
Maybe $O(\log n + t)$ for the access operation
(it depends on implementation).

But we copy a lot of paths.

Persistent red–black tree with **path copying**.

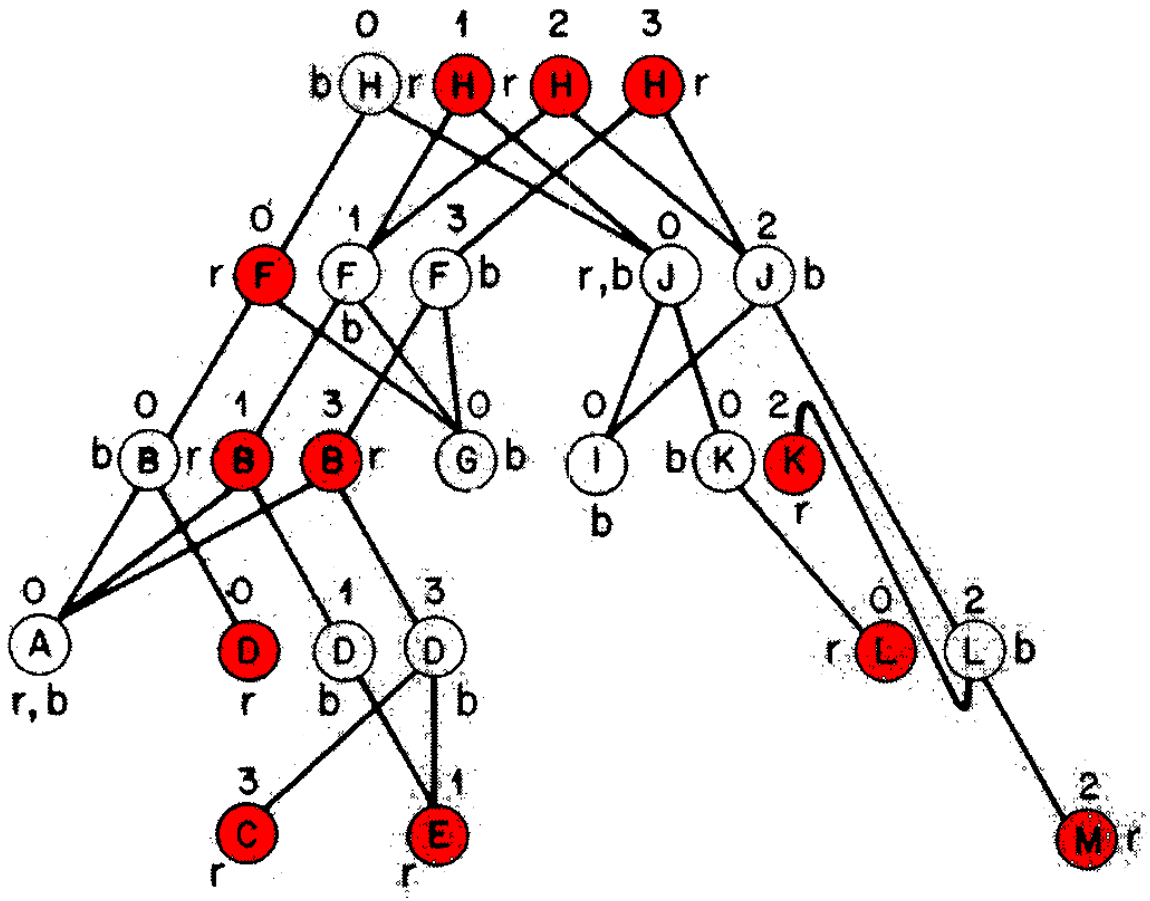
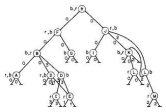


Figure: Figure 6 of Neil Sarnak, Robert E. Tarjan (Ref. 28)



Persistent search tree with no node copying

We can do better,
with **no node copying**.

Instead copying path, we will **add links in nodes**.
Each insertion or deletion cost $O(1)$ space.

But we have a time penalty.
Access become $O(\log n \log m)$ (with m maximum number of links in nodes).

Persistent red–black tree with **no node copying**.

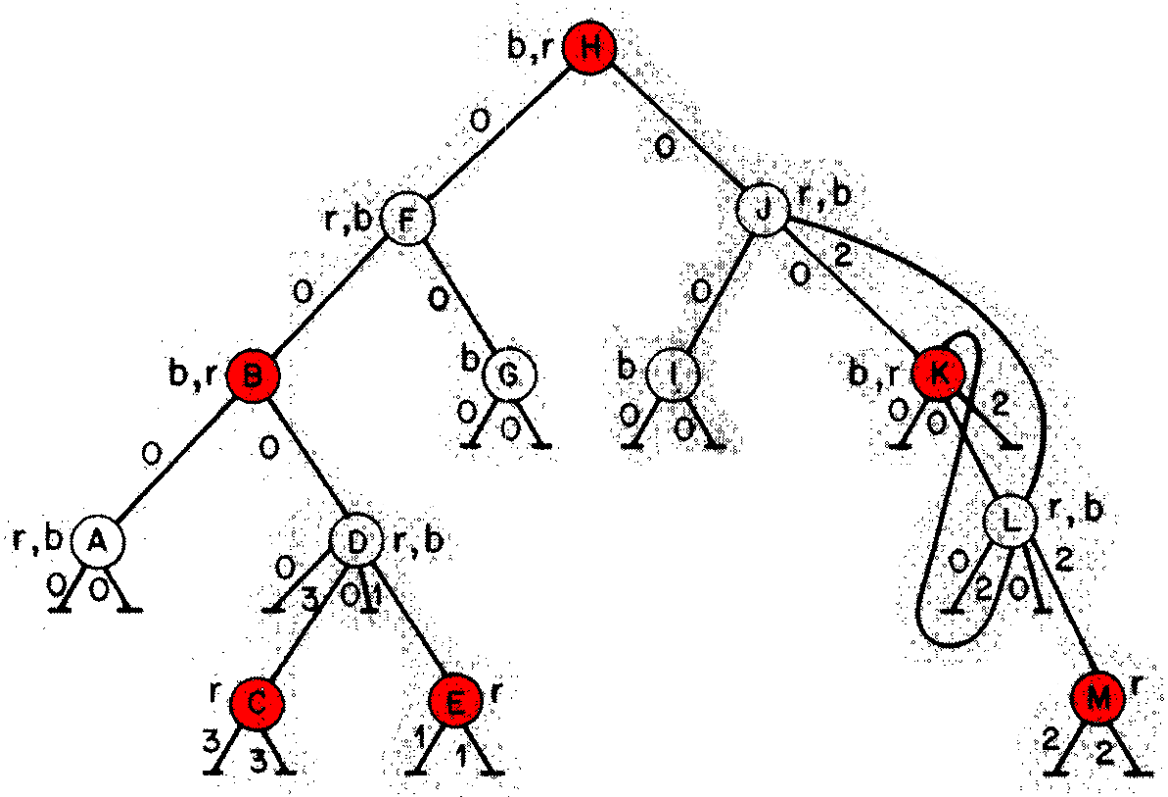
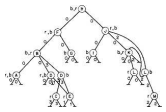


Figure: Figure 7 of Neil Sarnak, Robert E. Tarjan (Ref. 28)



Persistent search tree with limited node copying

We mix the two ways.
In each node we allow
k extra links.

And if no empty link is available then we copy the node.

The article of Sarnak and Tarjan study the **amortized space cost** and conclude that is **linear**: $O(n)$.

The good choice of k depend of what we want (speed or space economy).
$k = 1$ is a good choice by default.

Previous methods path copying and no node copying are specific cases of the limited node copying method (corresponding to $k = 0$ and $k = \infty$).

Persistent red–black tree **limited** node copying with only **one extra link** ($k = 1$).

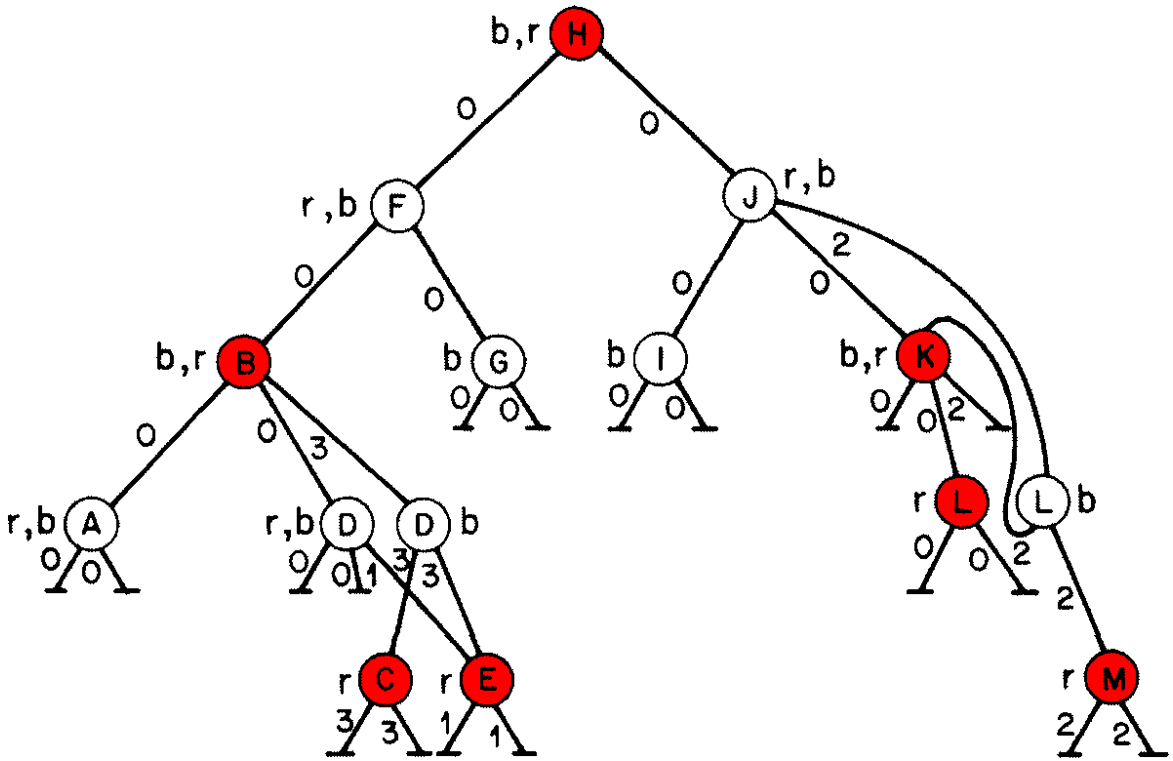
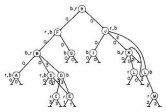


Figure: Figure 8 of Neil Sarnak, Robert E. Tarjan (Ref. 28)



Persistent search tree

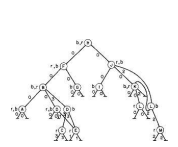
In summary,

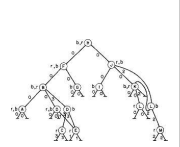
with a red–black tree we have built
a **persistent binary search tree** with **good complexities**:

- **Operations** in $O(\log n)$ in **worst case**
- and **space** in $O(n)$ in **amortized space cost**.

Applications (of this persistent data structure, or similar):

- In computational geometry (planar point location problem)
- Functional languages
- Incremental backup system
- Versioning system (like Git, Mercurial, SVN…)
- …





References

Thank you!

References:

- Neil Sarnak, Robert E. Tarjan (1986).
  Planar Point Location Using Persistent Search Trees.
  Communications of the ACM. 29 (7) pp.669–679
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
  Introduction to Algorithms.
  MIT Press, 3rd 2009
  - draw.io
  - LaTeX with beamer class

Questions time...