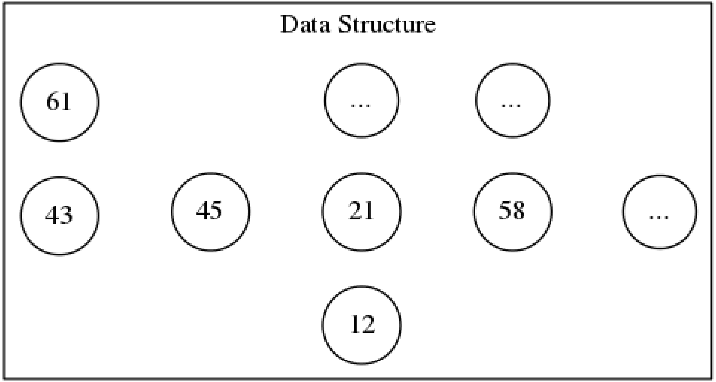
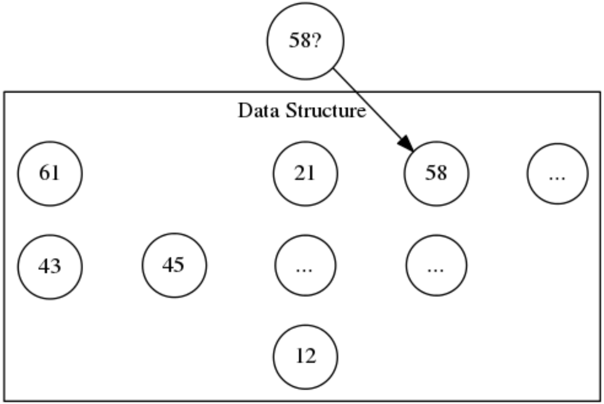
Self-Adjusting Binary Search Trees

Andrei Pârvu

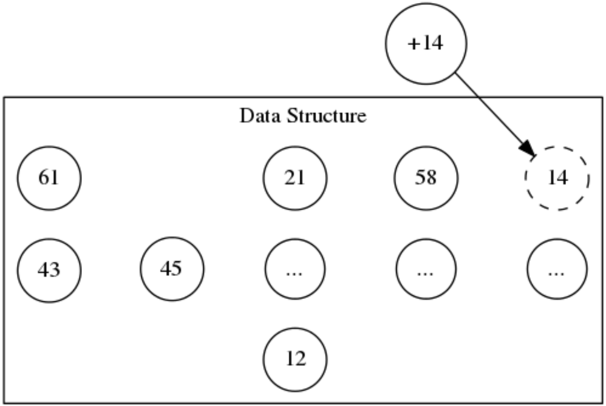
Motivation



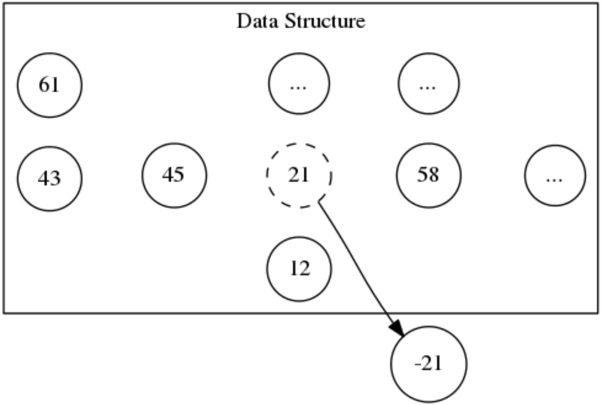
Motivation: Find



Motivation: Insert



Motivation: Delete



Goal: $O(M * logN)$ **time complexity (**N **elements,** M **operations)**

What is a Binary Search Tree?

- Binary tree :)

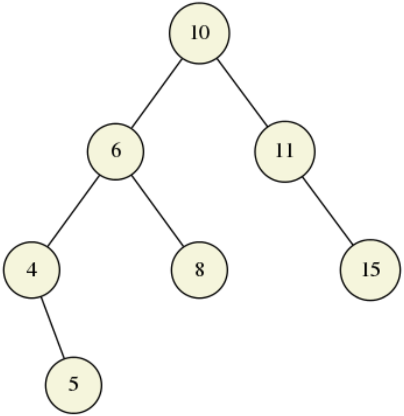
What is a Binary Search Tree?

- Binary tree :)
- Each node stores an element (key)

What is a Binary Search Tree?

- Binary tree :)
- Each node stores an element (key)
- Key of a node:
  - is bigger than the keys of the left subtree
  - is smaller than the keys of the right subtree

Example



Operations: find element

- walk recursively down the tree

Operations: find element

- walk recursively down the tree
- if element equals with node key, stop

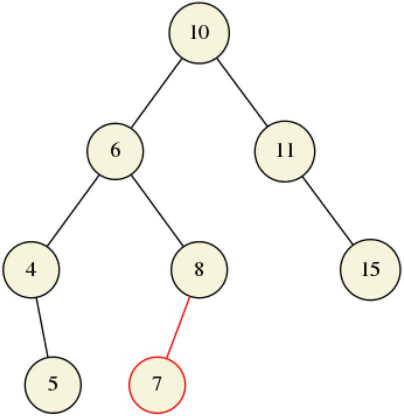
Operations: find element

- walk recursively down the tree
- if element equals with node key, stop
- else
  - go to left child if element < than node key
  - go to right child if element > than node key

Operations: insert element

- same algorithm as find
- add element as leaf

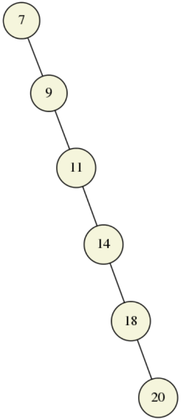
Example: insert element



Time complexity of operations

- if elements are chosen randomly, then $O(M * logN)$
- most of the time that is not the case :(

Example linear tree



How to make it faster?

Rotations

- rotate a node to the left or to the right

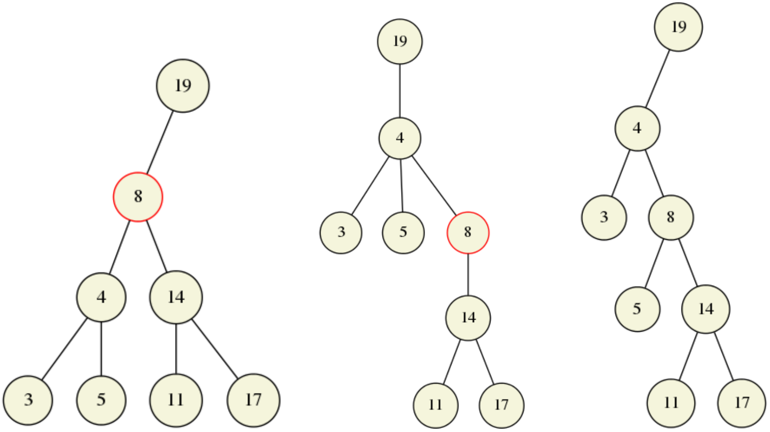
Rotations

- rotate a node to the left or to the right
- maintain the BST invariant

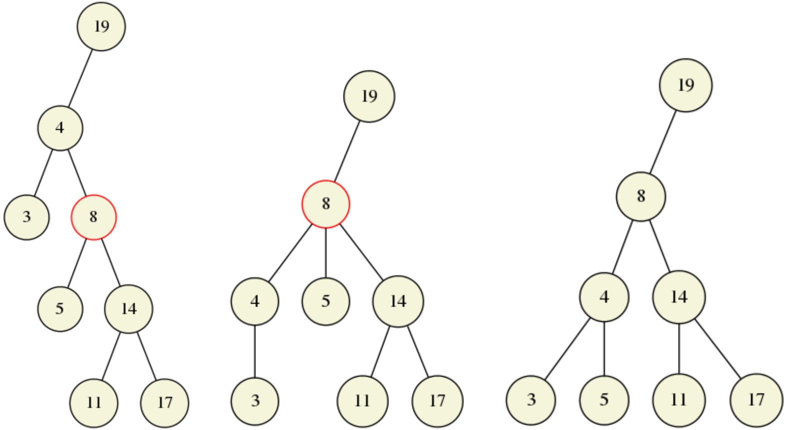
Rotations

- rotate a node to the left or to the right
- maintain the BST invariant
- use them to modify the tree structure and maintain it balanced

Example: rotation to the right



Operations: rotation to the left



How can we use rotations?

Move to root heuristic

- after accessing an item at node x, rotate edge from x to its parent until x becomes root.
- **Does this improve anything?**

Move to root heuristic

- after accessing an item at node x, rotate edge from x to its parent until x becomes root.
- **Does this improve anything?**
- **No, time of access can still be** $O(n)$

Splay tree

- BST with a restructuring heuristic, called splaying
- after inserting or finding an element, do pairs of rotations bottom-up

Splay tree

- BST with a restructuring heuristic, called splaying
- after inserting or finding an element, do pairs of rotations bottom-up
- rotations depend on the structure of the path
- each pair of rotations shall be named a splaying step

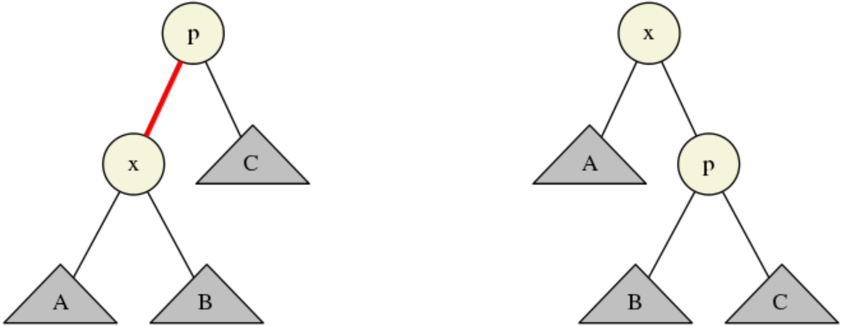
Splay tree

- BST with a restructuring heuristic, called splaying
- after inserting or finding an element, do pairs of rotations bottom-up
- rotations depend on the structure of the path
- each pair of rotations shall be named a splaying step
- repeat splaying step on x until it is root

Splaying step - case 1: zig

- if $p(x)$, parent of x, is root of tree, rotate edge joining x with $p(x)$
- terminal case

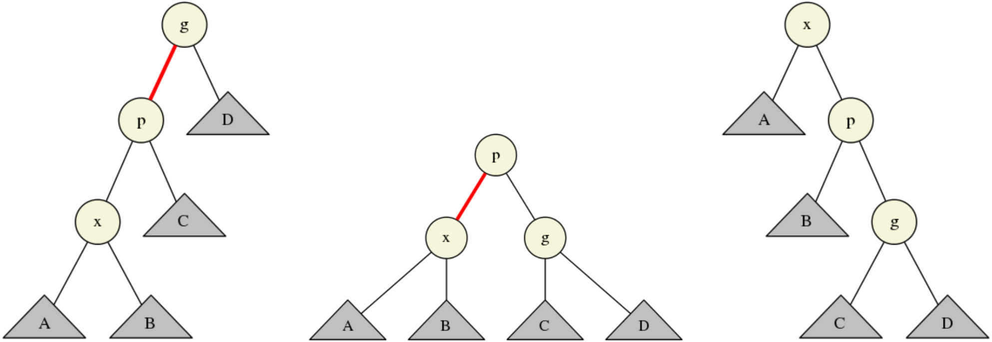
Example: zig



Splaying step - case 2: zig-zig

- $p(x)$ not the root
- $g(x)$ parent of $p(x)$
- x and $p(x)$ both right-children or both left-children
- rotate edge joining $p(x)$ with $g(x)$
- rotate edge joining $p(x)$ with x

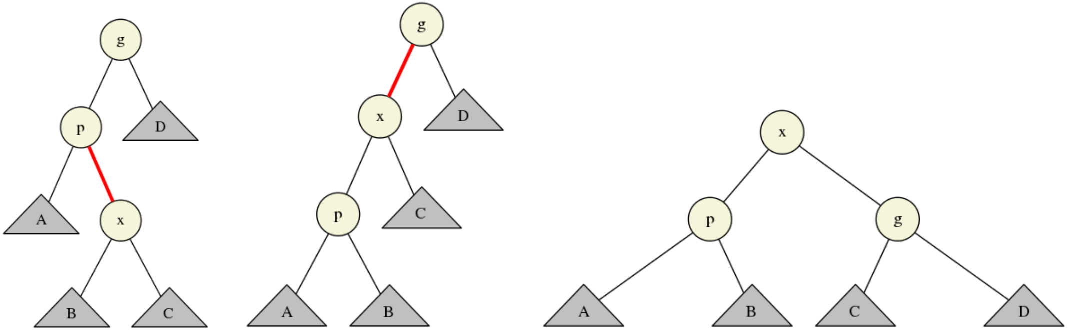
Example: zig-zig



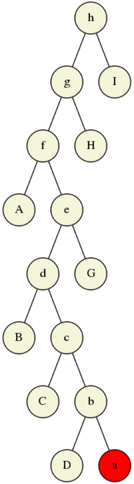
Splaying step - case 3: zig-zag

- $p(x)$ not the root
- $g(x)$ parent of $p(x)$
- x left child and $p(x)$ right child or vice-versa
- rotate edge joining x with $p(x)$
- rotate edge joining x with $g(x)$

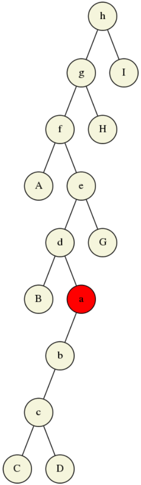
Example: zig-zag



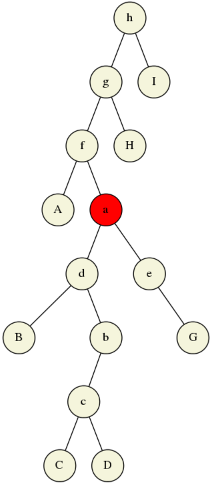
Example: splaying on a node



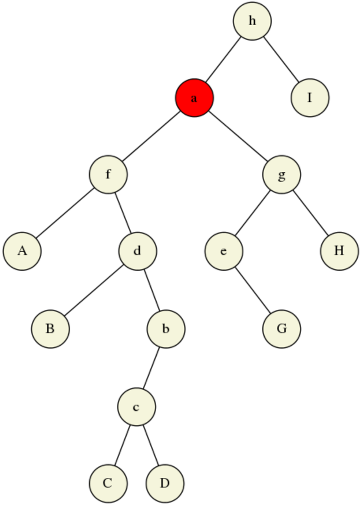
Example: splaying on a node (1)



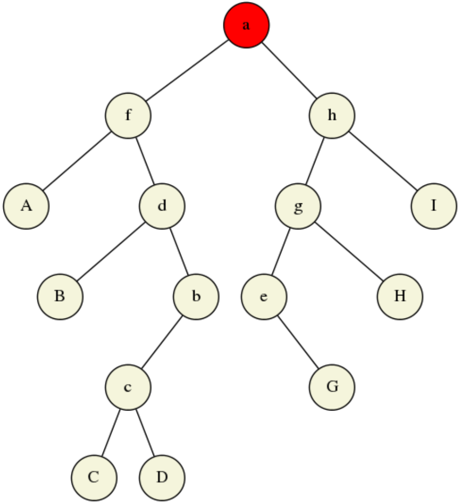
Example: splaying on a node (2)



Example: splaying on a node (3)



Example: splaying on a node (4)



Complexity & Analysis

- Why is splaying better than *move to root* heuristic?

Complexity & Analysis

- Why is splaying better than *move to root* heuristic?
- if a node is at depth d on the splaying path, it will be at about $d/2$ after the splay

Complexity & Analysis

- Why is splaying better than *move to root* heuristic?
- if a node is at depth d on the splaying path, it will be at about $d/2$ after the splay
  - except the root, its child and the splayed node

Complexity & Analysis II

- use the *potential method*
- $\Phi(T)$ = extra time that can be later consumed on tree T
- from T to T' amortized time = $actual\_time + \Phi(T') - \Phi(T)$

Complexity & Analysis II

- amortized time = $actual\_time + \Phi(T') - \Phi(T)$
- if *actual time* < *amortized time*, increase potential
- if *actual time* > *amortized time*, decrease potential

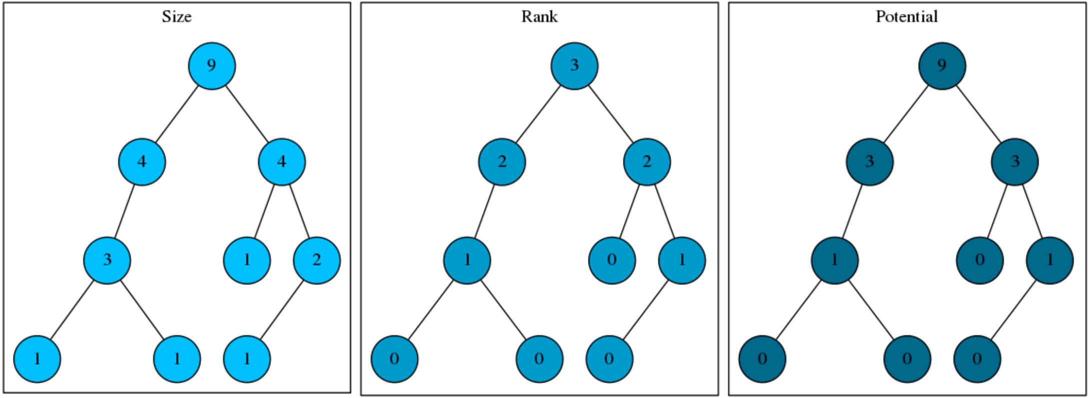
Analysis on M operations

$$t_1 + t_2 + ... + t_M + (\Phi(T_1) - \Phi(T_0)) + (\Phi(T_2) - \Phi(T_1)) + ... + (\Phi(T_M) - \Phi(T_{M-1})) =$$
$$t_1 + t_2 + ... + t_M + \Phi(T_M) - \Phi(T_0).$$

Potential function

- *size*(x) = number of nodes in the subtree rooted at x
- $rank(x) = log_2(size(x))$
- $\Phi(T)$ = sum of *ranks* of nodes in subtree T

Potential function



Potential splaying

- only x, $p(x)$ and $g(x)$ change rank
- $\Delta\Phi = rank_i(g) - rank_{i-1}(g) + rank_i(x) - rank_{i-1}(x) + rank_i(p) - rank_{i-1}(p)$
- $actual\_cost + \Delta\Phi \leq 3 * (rank_i(x) - rank_{i-1}(x)) + 1$

Complexity & Analysis III

- amortized time = $actual\_cost + \Delta\Phi \leq 3 * (rank_i(x) - rank_{i-1}(x)) + 1$
- total time $O(m * log(n))$

Analysis

Pros:

- no additional information stored in nodes
- not that hard to implement

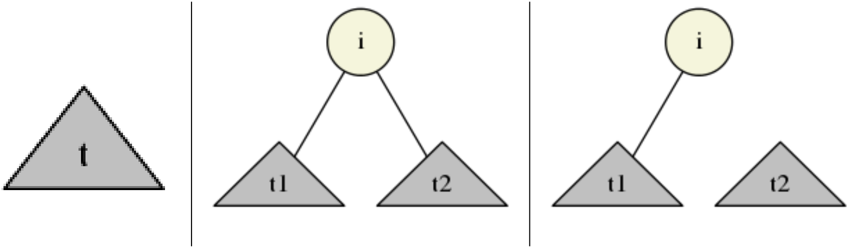
Cons:

- at one point an operation can have $O(n)$ time
- problems with multithreading

Splitting a splay tree

- split(i, t): construct and return t_1 and t_2
  - elements in t_1 smaller than i
  - elements in t_2 greater than i
- **Ideas?**

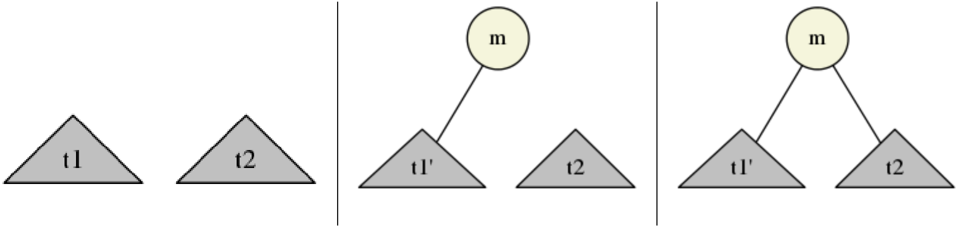
How to split?



Joining two splay trees

- join(t1, t2): combine t_1 and t_2 into single tree
  - elements in t_1 smaller than elements in t_2
- **Ideas?**

How to join?

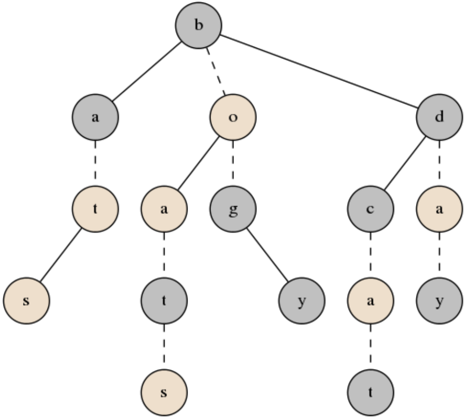


Applications: Lexicographic Search Tree

Lexicographic Search Tree

- store a set S of strings
- repeated access operations are efficient

Example - Lexicographic Tree



boy

bog

cat

as

day

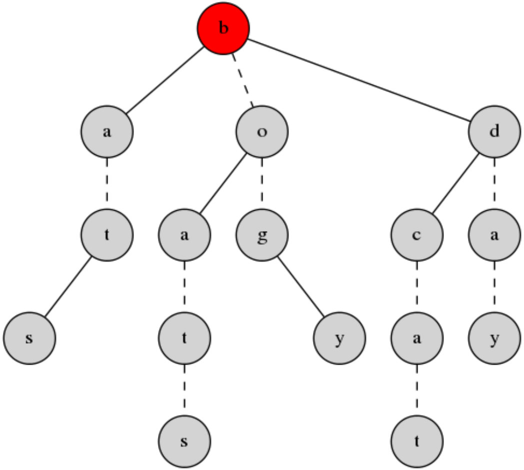
at

bats

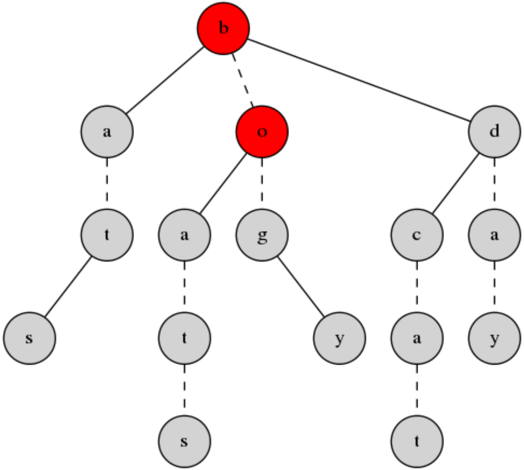
Lexicographic Search Tree II

- ternary tree
- symbols in each node
- two types of edges
  - middle (dashed)
  - left & right
- nodes in the tree correspond to prefixes of strings:
  - concatenate symbols from which we leave by a dashed edge
- nodes connected by continuous edges form a binary search tree

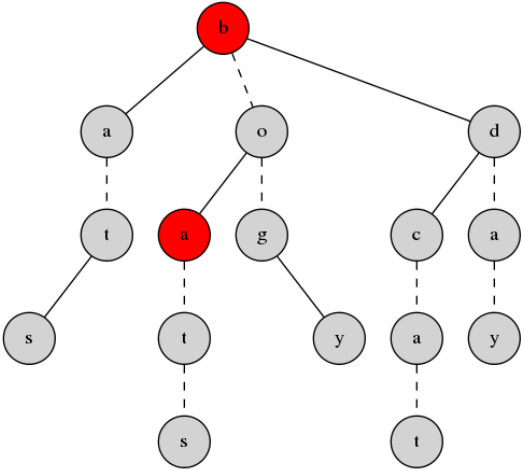
Search for ’bats’ (1)



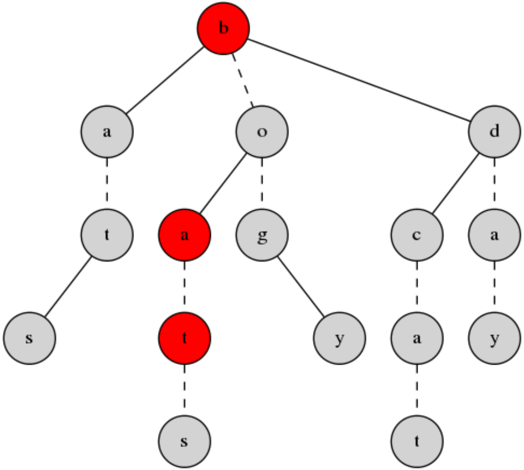
Search for 'bats' (2)



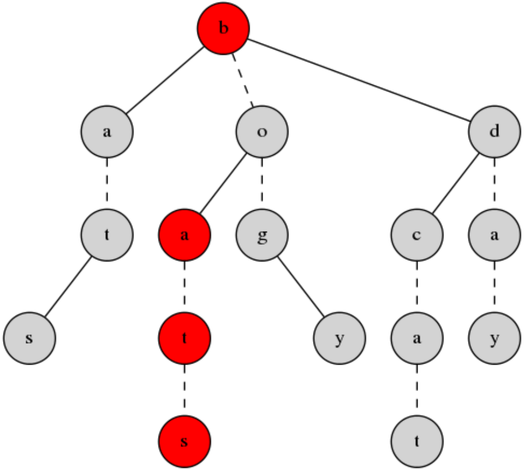
Search for ’bats’ (3)



Search for 'bats' (4)



Search for ’bats’ (5)



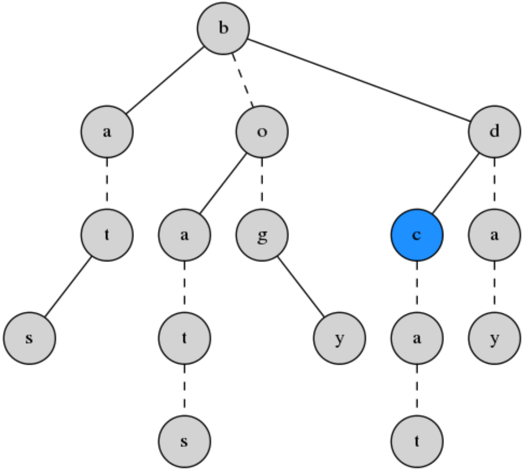
Using splaying

- rotation rearranges left and right child, but not the middle props
- splay at node x:
  - similar with normal splay tree
  - if node is middle child, continue with p(x)

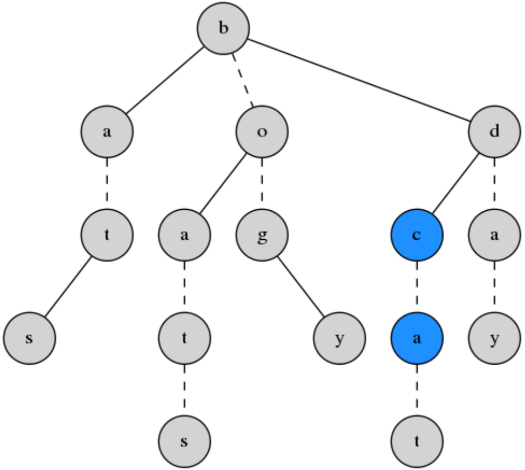
Using splaying

- rotation rearranges left and right child, but not the middle props
- splay at node x:
  - similar with normal splay tree
  - if node is middle child, continue with p(x)
- after splaying, path from root to x contains only dashed edges

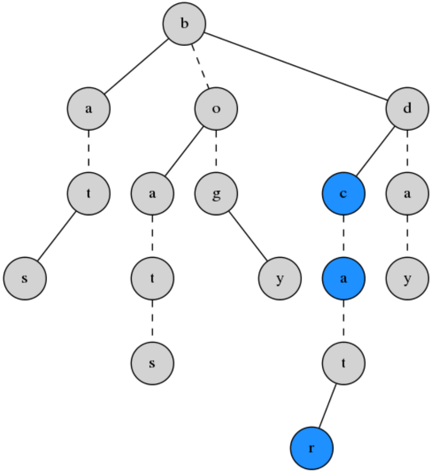
Insert ’car’



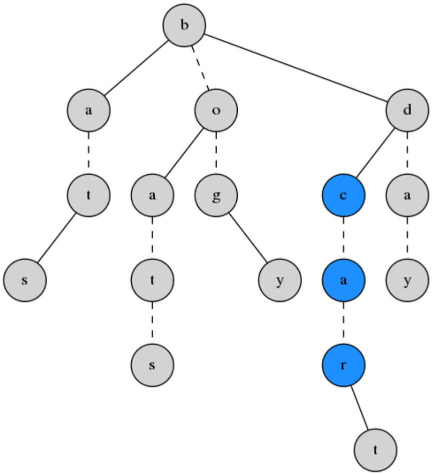
Insert ’car’ (2)



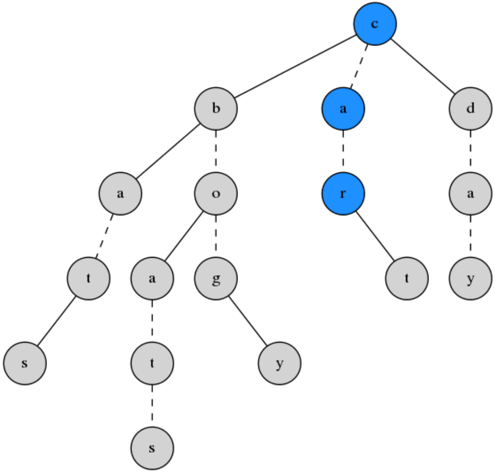
Insert ’car’ (3)



Lex Tree splay



Lex Tree splay



Lex Tree analysis

- time of access is bounded by $|\sigma|$ plus number of right and left edges traversed
- $O(|\sigma| + log_2(N))$

Application: Link-Cut Trees

Link-Cut Trees

- abstract data structure for maintaining a forest of rooted trees
- the following operations should be supported
  - find_root(v)
  - cut(v)
  - link(v, w)

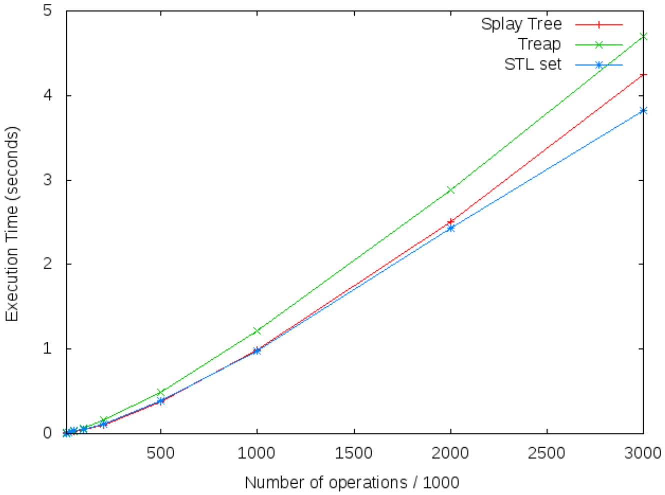
Where are self-adjusting BSTs used?

- Java (TreeMap, TreeSet) and C++ (set, map)

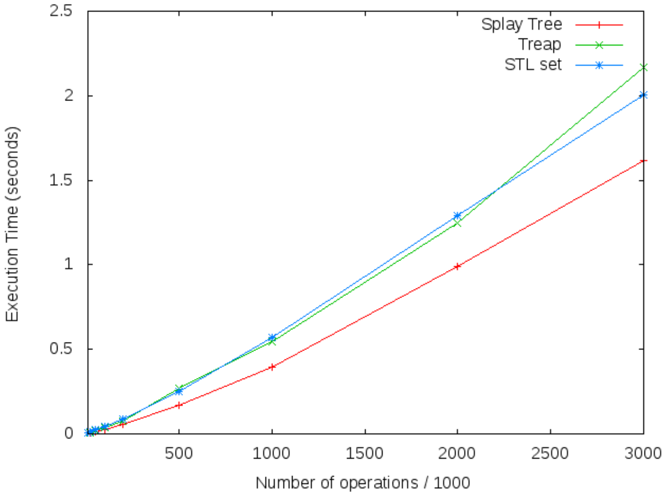
- Java (TreeMap, TreeSet) and C++ (set, map)
- Linux CFS scheduler, which decides which tasks are executed when

- Java (TreeMap, TreeSet) and C++ (set, map)
- Linux CFS scheduler: decides which tasks are executed when
- memory allocators

Experiment 1: normal queries

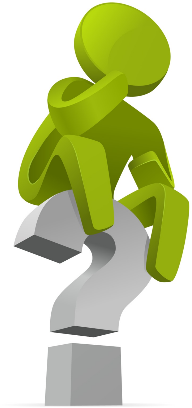


Experiment 2: reduced set of query elements



Take-home message

- you probably use self-adjusting binary search trees every day :)
- it is useful to know how they work and how to implement one
- C++ STL or java.util cannot save you all the time



Potential function zig-zag

- only x, $p(x)$ and $g(x)$ change rank
- $\Delta\Phi = rank'(g) - rank(g) + rank'(x) - rank(x) + rank'(p) - rank(p) = rank'(g) - rank(x) + rank'(p) - rank(p) \leq rank'(g) + rank'(p) - 2 * rank(x)$
- $rank'(g) + rank'(p) - 2 * rank(x) + 2 - 2 \leq [rank'(g) + rank'(p) - 2 * rank(x)] + 2 * rank'(x) - rank(p) - rank'(g) - 2 \leq 2 * (rank'(x) - rank(x)) - 2$