Search Trees

Tobias Lieber

April 14, 2008

Definition

An (undirected) graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E.

$$E \subseteq \binom{V}{2} := \{X : X \subseteq V, |X| = 2\}$$

A directed graph $G = (V, E)$ is given by a set of nodes and a set of directed edges:

$$E \subseteq V \times V$$

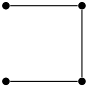
Definition

The neigborhood of node x is given by:

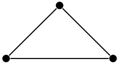
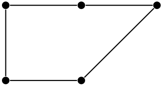
$$N(x) = \{y : x \in V,\ \{x, y\} \in E\}$$

Special Graphs

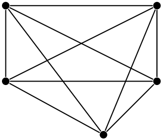
Path:

P_2:  P_4: 

Circle:

C_3:  C_5: 

Complete graph/ Clique:

K_5: 

Definition

A graph $G = (V, E)$ is called connected, if there is a path from each node x to each other node y.

Definition

A graph $H = (W, F)$ is called subgraph of $G = (V, E)$ if

$$W \subseteq V \text{ and } F \subseteq E.$$

Definition

An acylic graph $G = (V, E)$ does not contain any circle as a subgraph.

Definition

A graph $G = (V, E)$ is called a tree if it is connected and acyclic.

Definition

A rooted binary tree $G = (V, E)$ is a tree with one root node r.

$$\begin{aligned} |N(r)| &< 3 \quad r \in V \\ 1 \leq |N(x)| &\leq 3 \quad \forall x \in V \setminus \{r\} \end{aligned}$$

Definition

The height of a tree $G = (V, E)$ with root $r \in V$ is defined as

$$h = \max_{x \in V}\{\text{distance from } r \text{ to } x\}$$

Theorem

The following definitions of a tree $G = (V, E)$ are equivalent

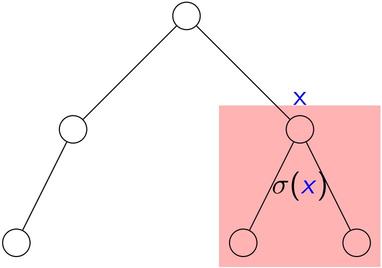
- *G is connected and acyclic.*
- *G is connected and $|V| = |E| + 1$.*
- *G is acyclic and $|V| = |E| + 1$.*
- *When adding a new edge to G the resulting graph will contain a circle.*
- *When removing an edge from G the resulting graph is not connected anymore.*
- *For all two nodes $x, y \in V$ and $x \neq y$ there is exactly one path from x to y.*

Definition

A tree $H = (W, F)$ is called a spanning tree of a graph $G = (V, E)$ if $W = V$ and $F \subseteq E$.

Definition

The function $\sigma(x)$ returns the subtree, which is rooted in x:



Problem:
For a set of items $x_1, \ldots, x_n$ where each dataset consists of a key and a value, we want to minimize the total access time on an arbitrary sequence of operations.
One operation can perform

- a test if a key is stored in the data structure (IsElement),
- the insertion of an item in the data structure (Insert)
- or a deletion of a key in the data structure (Delete).

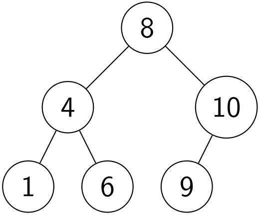
- An internal search tree stores all keys in internal nodes. The leaves contain no further information. Accordingly there is no need to store them and they can be represented by NIL-pointers.
- In an external search tree, all keys are stored at the leaves. The internal nodes only contain information for managing the data structure.

A binary search tree is a binary tree, whose internal nodes contain the keys $k = x.key \ \forall x \in S$. For each node x the following equation must hold if node y is in the left subtree of x and node z is in the right subtree of node x:

$$y.key < x.key < z.key$$

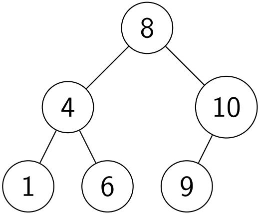
A binary search tree is a binary tree, whose internal nodes contain the keys $k = x.key \ \forall x \in S$. For each node x the following equation must hold if node y is in the left subtree of x and node z is in the right subtree of node x:

$$y.key < x.key < z.key$$



For making algorithms more understandable, here are more definitions.
A node v of a search tree stores several values:

- key – key of the stored item
- leftChild, rightChild which are pointers to left/right child (only if it is a binary tree)
- children, the number of children



The items are accessible in pseudocode as follows:

```
k=v.key // stores 8 in k if v is the root
```

```
IsElement(T,k)
{
  v:=T.root
  while(v!=NIL)
  {
    if(v.key==k)
      return v
    else if(v.key>k)
      v=v.leftChild
    else
      v=v.rightChild
  }
  return v
}
```

```
Insert(T,k)
{
  v=IsElement(T,k)
  if(v==NIL)
  {
    // Inserts a node, updates pointers
    add a node w with w.key=k
    v=w
  }
}
```

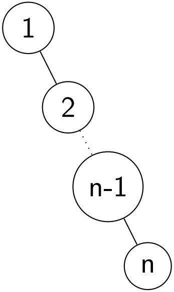
```
Delete(T,k)
{
  v=isElement(T,k)
  if(v==NIL)
    return
  else
    replace v by a InOrder-predecessor/successor
}
```

There are sequences of operations, such that each operation requires $\Theta(n)$ operations, if n is the number of nodes in the tree.
Thus the worst-case complexity of a binary search tree is

$$\Theta(n)$$

There are sequences of operations, such that each operation requires $\Theta(n)$ operations, if n is the number of nodes in the tree.
Thus the worst-case complexity of a binary search tree is

$$\Theta(n)$$

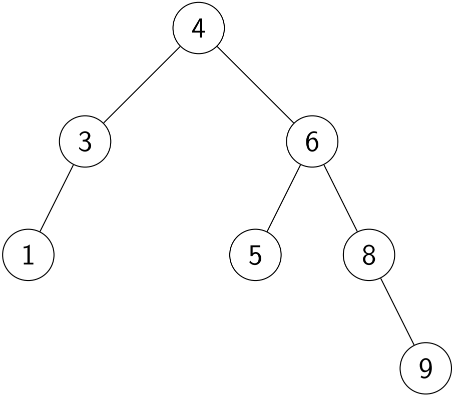


AVL-trees have been invented in 1962 and are internal binary search trees. They are named after their inventors: Georgy Adelson-Velsky and Yevgeniy Landis.

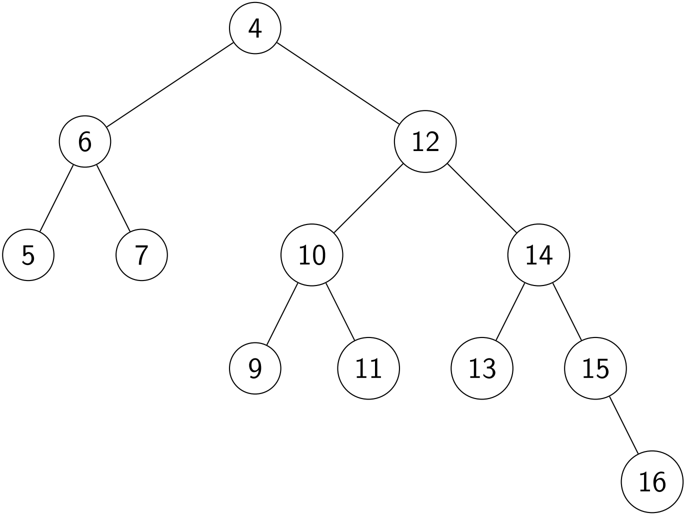
The main idea of AVL-trees is to keep the tree height balanced. This means

$$|\text{height}(\sigma(\text{v.leftchild})) - \text{height}(\sigma(\text{v.rightChild}))| \leq 1$$

has to be valid for every node v in an AVL-tree.



... is an AVL tree.



… is not an AVL tree.

Theorem

An internal binary search tree with height h contains at most $2^h - 1$ nodes.

Proof.

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

$\square$

Theorem

An AVL-tree with height h consists at least of $F_{h+2} - 1$ internal nodes.

Proof.

How could an AVL-tree T_h with height h and a minimal number of nodes be constructed?

AVL-condition: $height(\sigma(\text{r.leftchild})) - height(\sigma(\text{r.rightchild})) = 1$

Height should be $h \Rightarrow$

$height(\sigma(\text{r.leftChild})) = h - 1, height(\sigma(\text{r.rightChild})) = h - 2$

$\Rightarrow n(T_h) = 1 + n(T_{h-1}) + n(T_{h-2})$

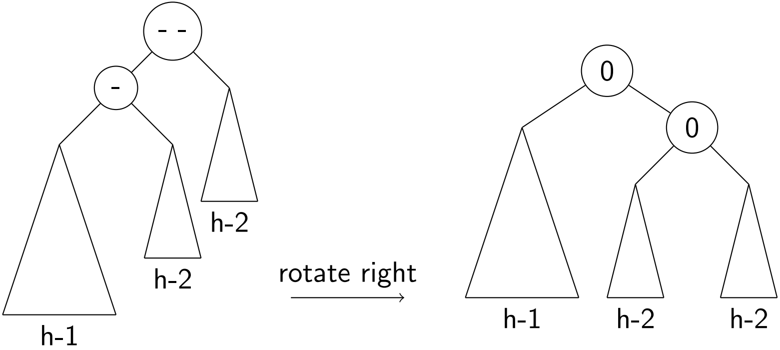
$$
\begin{aligned}
n(T_1) &= 1 && = 2 - 1 && = F_3 - 1 \\
n(T_2) &= 2 && = 3 - 1 && = F_4 - 1 \\
n(T_3) &= 4 && = 5 - 1 && = F_5 - 1 \\
n(T_h) &= 1 + n(T_{h-1}) + n(T_{h-2}) && = 1 + F_{h+1} - 1 + F_h - 1 && = F_{h+2} - 1
\end{aligned}
$$

We know:

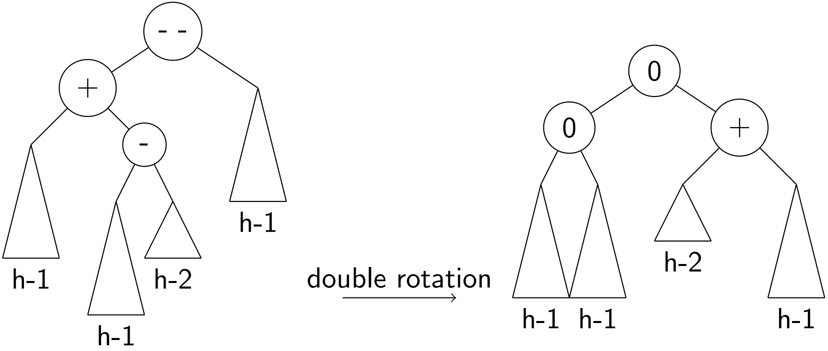
$$n \geq \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2}$$

$$\begin{aligned} h \leq &\quad \frac{\log n}{\log\left(\frac{1+\sqrt{5}}{2}\right)} - \log\left(\frac{1}{\sqrt{5}}\right) - 2 \\ \approx &\quad 1.44 \log n + 1.1 \end{aligned}$$

Single rotation:



Double rotation:

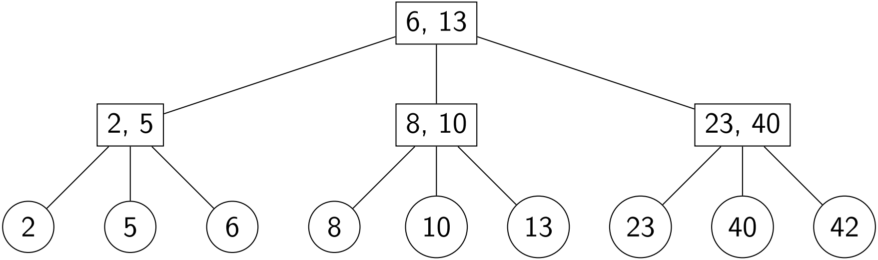


Definition

An external search tree is an (a, b)-tree if it applies to the following conditions:

- All leaves appear on the same level.
- Every node, except of the root, has $\geq a$ children.
- The root has at least two children.
- Every node has at most b children.
- Every node with k children contains $k - 1$ keys.
- $b \geq 2a - 1$

A $(2, 4)$-tree:



Theorem

Every (a, b)-Tree with height h has

$$2a^{h-1} \leq n \leq b^h$$

leaves.

Proof.

1. In an (a, b)-tree which branching factor is as small as possible, the root has two children and every other node has a children.
2. If we choose the branching factor as high as possible, every node has b children.

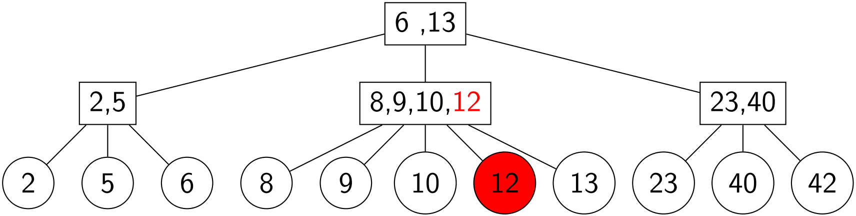
□

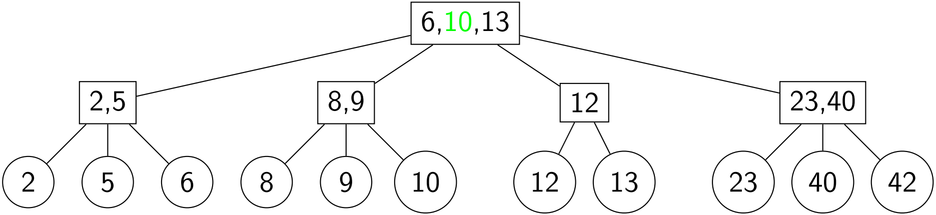
$$log_b n \leq h \leq \log_a \frac{n}{2} + 1$$

```
IsElement(T,k)
{
  v=T.root
  while(not v.leaf)
  {
    i=min{s; 1 ≤ s ≤ v.children+1 and k ≤ key no. s}
    // define key no. v.children+1 = ∞
    v=child no. i
  }
  return v
}
```

```
Insert(T,k)
{
  w=IsElement(T,k)
  v=parent(w)
  if(w.key!=k)
  {
    if(k< max_key(v) )
      insert k left of w
    else
      insert k right of w
    if( v.children > b )
      rebalance(v)
  }
}
```

```
rebalance(T,l)
{
  w=parent_n(l) // returns an new root, if w==T.root
   r=new node with nodes (⌈m/2⌉ ... m)
  w.add_node(K_{m/2},r)
   if(w.children>b)
     rebalance(w)
}
```

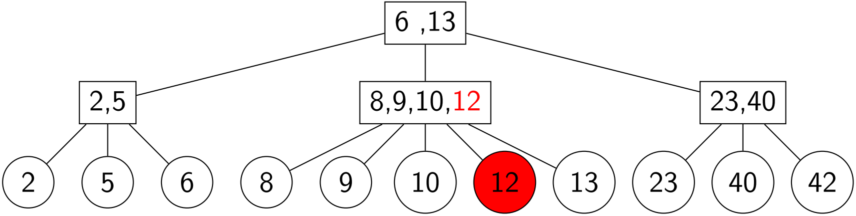

6 ,13
2,5
8,9,10,12
23,40
2
5
6
8
9
10
12
13
23
40
42

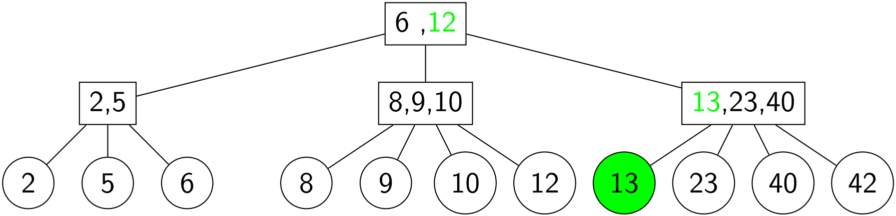

6,10,13
2,5
8,9
12
23,40
2
5
6
8
9
10
12
13
23
40
42

```
Delete(T,k)
{
  w=IsElement(T,k)
  v=parent(w)
  if(k=w.key)
    remove(w)
  if( v.children < a)
    rebalance_delete(T,v)
}
```

```
rebalance_delete(T,v)
{
  w=previous/next_sibling(v)
  r=join(v,w)
  if(r.children >b)
  {
    rebalance_delete(r)
  }
}
```

An alternative way for rebalancing is the idea of overflow.
Test if a sibling can adopt a child of an overfull node.




6 ,12
2,5
8,9,10
13,23,40
2
5
6
8
9
10
12
13
23
40
42

Definition

A B*-tree with order b is defined as follows:

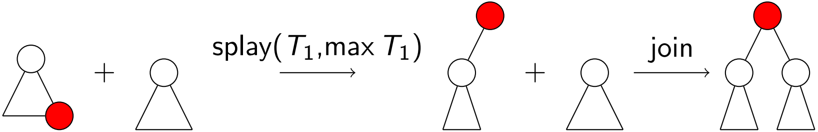
- All leaves appear on the same level
- Every node except when the root has at most b children
- Every node except when the root has at least $(2b-1)/3$ children
- The root has at least two and at most $2\lfloor(2m-2)/3\rfloor+1$
- Every internal node with k children contains $k-1$ keys

Splay trees are self-organizing internal binary search trees.
Basic idea: Self-adjusting linear list with the move to front rule.

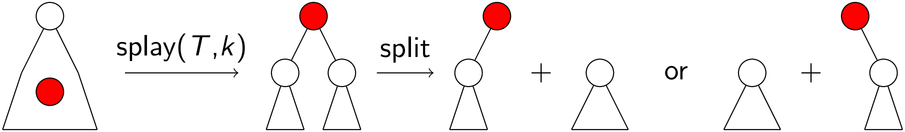
- Simple algorithm
- Good run time in an amortized sense

The splay operation moves a node x with respect to the properties of a search tree to the root of a binary tree T.

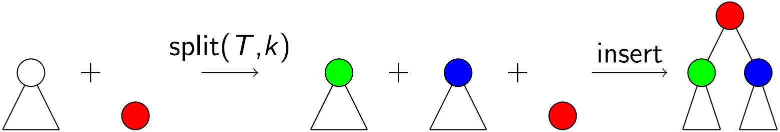
join(T_1, T_2):



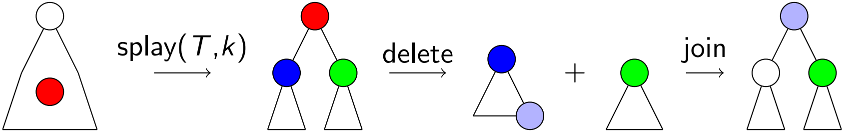
split(T,k):



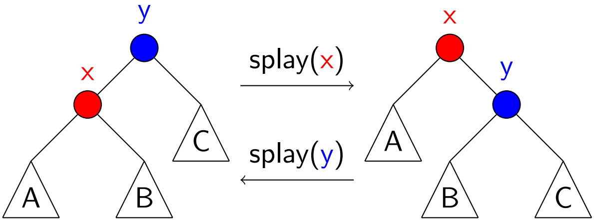
insert(T,k):

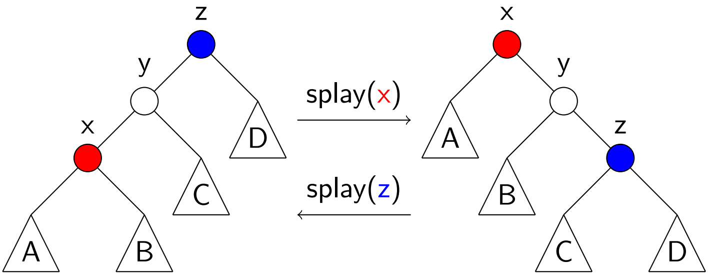


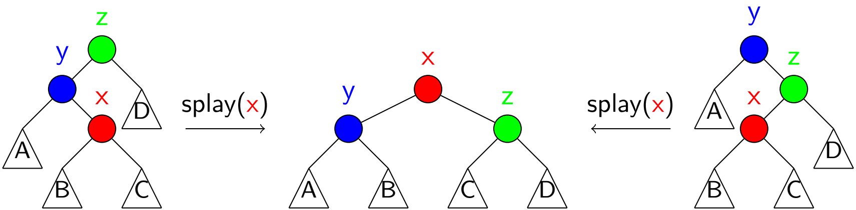
delete(T,k):



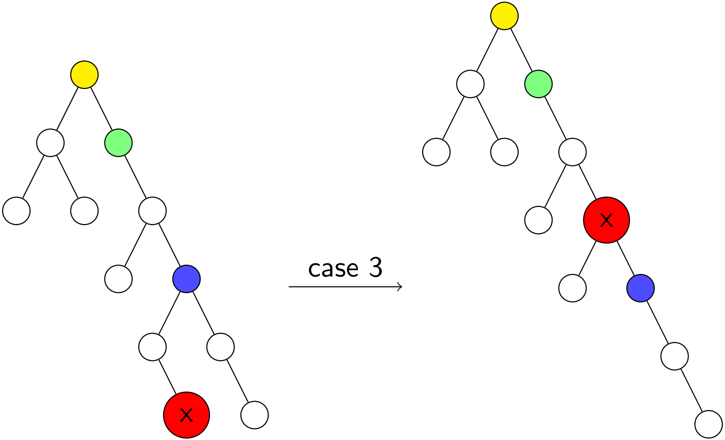
Splay(T,x) uses single and double rotations for transporting node x to the root of a splay tree T.



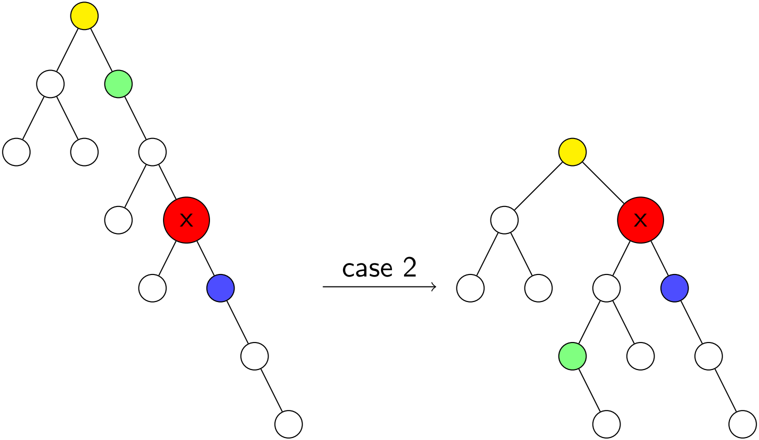

z
y
x
D
C
A
B
splay(x)
splay(z)
x
y
A
z
B
C
D


z
y
x
D
A
B
C
splay(x)
x
y
z
A
B
C
D
splay(x)
y
z
x
A
D
B
C

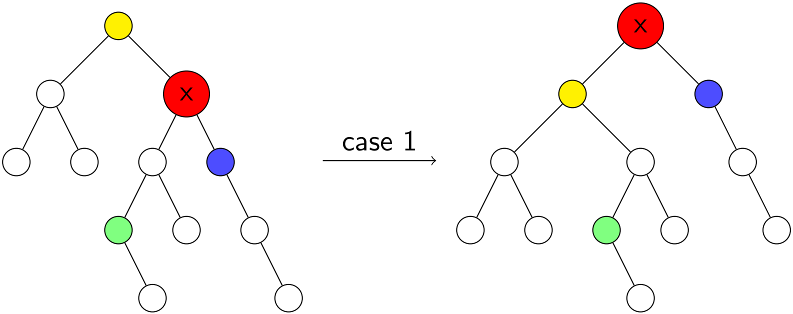
splay(T,x):



splay(T,x):



splay(T,x):



In amortized analysis of algorithms we investigate the costs of m operations.

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i + \Phi_{i-1} - \Phi_i) = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m$$

For the following analysis, we define:

- A weight $w(i)$ for each node i
- The size of node x: $s(x) = \sum_{i \in \sigma(x)} w(i)$
- The rank of node x: $r(x) = \log s(x)$
- The potential of a tree T: $\Phi = \sum_{i \in T} r(i)$

Theorem

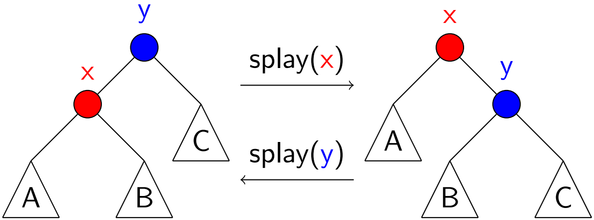
Splay(T,x) needs at most

$$3(r(v) - r(x)) + 1 = O(\log\left(\frac{s(v)}{s(x)}\right))$$

amortized time, where v is the root of T.

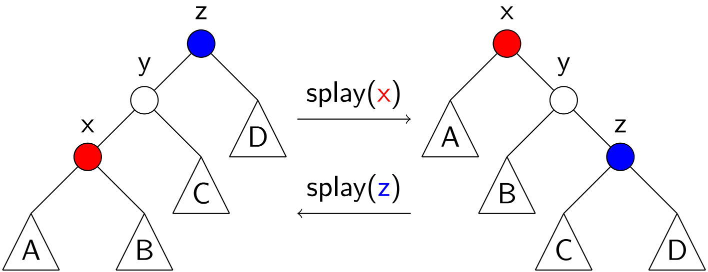
We can divide the splay operation in the rotations which are the influential operations in splay. Thus we consider the number of the rotations.
Just one more notation:
Let $r(x)$ be the rank of x before the rotation and $R(x)$ the rank after the rotation. Let $s(x)$ be the size of x before the rotation and $S(x)$ the size after the rotation.

Case 1:



$$
\begin{aligned}
& 1 + R(x) + R(y) - r(x) - r(y) & \\
\leq\ & 1 + R(x) - r(x) & \text{since } R(y) \leq r(y) \\
\leq\ & 1 + 3(R(x) - r(x)) & \text{since } r(x) \leq R(x)
\end{aligned}
$$

Case 2:



$$
\begin{aligned}
& 2 + R(x) + R(y) + R(z) - r(x) - r(y) - r(z) & \\
= \; & 2 + R(y) + R(z) - r(x) - r(y) & \text{since } R(X) = r(z) \\
\leq \; & 2 + R(x) + R(z) - 2r(x) & \text{since } R(y) \leq R(x) \\
& & \text{and } r(x) \leq r(y)
\end{aligned}
$$

Claim:

$$\begin{aligned} 2 + R(x) + R(z) - 2r(x) &\leq 3(R(x) - r(x)) \\ 2 &\leq 2R(x) - r(x) - R(z) \\ -2 &\geq \log(\frac{s(x)}{S(x)}) + \log(\frac{S(z)}{S(x)}) \end{aligned}$$

$$\begin{aligned} s(x) + S(z) &\leq S(x) \\ \frac{s(x)}{S(x)} + \frac{S(z)}{S(x)} &\leq 1 \end{aligned}$$

The log-function is strictly increasing. Thus the maximum of $f(x, y) = \log x + \log y$ is given by x, y with $y = 1 - x$.
For maximization we receive the function $g(x) = \log_a x + \log_a(1 - x)$.

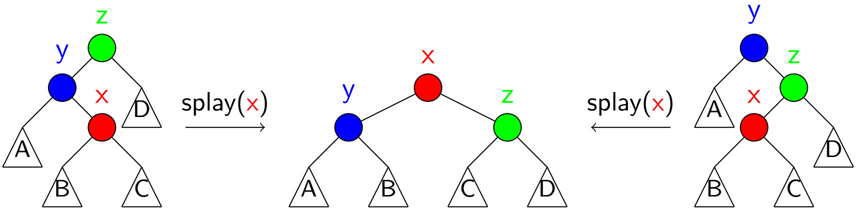
$$
\begin{aligned}
g'(x) = &\quad \tfrac{1}{\ln a}\left(\tfrac{1}{x} - \tfrac{1}{1-x}\right) \\
g''(x) = &\quad \tfrac{1}{\ln a}\left(\tfrac{1}{x^2} + \tfrac{1}{(1-x)^2}\right)
\end{aligned}
$$

This leads us to $x = \frac{1}{2}$. Since $g''(\frac{1}{2})$ is negative we can be sure that $x = \frac{1}{2}$ is a local maximum. Because $g(\frac{1}{2}) = -2$ equation

$$-2 \geq \log\left(\frac{s(x)}{S(x)}\right) + \log\left(\frac{S(z)}{S(x)}\right)$$

holds.

Case 3:



$$
\begin{aligned}
& 2 + R(x) + R(y) + R(z) - r(x) - r(y) - r(z) & \\
= \quad & 2 + R(y) + R(z) - r(x) - r(y) & \text{since } R(x) = r(z) \\
\leq \quad & 2 + R(y) + R(z) - 2r(x) & \text{since } r(x) - r(y)
\end{aligned}
$$

Proof.

By adding all rotations used for *splay*(T, x) we receive a telescope sum, which yields us the amortized time
$\leq 3(R(x) - r(x)) + 1 = 3(r(t) - r(x)) + 1.$

$\square$

If the weights $w(i)$ are constant, $-\Phi_m(x)$ for a sequence of m splay has the upper bound:

$$\sum_{i=1}^{n} \log W - \log w(i) = \sum_{i=1}^{n} \frac{W}{w(i)}$$

with

$$W = \sum_{i=1}^{n} w(i)$$

Theorem

The costs of m access operations in a splay tree are

$$O((m+n)\log n + m)$$

Proof.

Choose $w(i) = \frac{1}{n}$.
Because $W = 1$ it follows, $a_i \leq 1 + 3\log n$.
$-\Phi_m = \sum_{i=1}^{n} \log \frac{W}{w(i)} = \sum_{i=1}^{n} \log n = n \log n$
Thus $t = a - \Phi_m = m(1 + 3\log n) + n\log n$ $\square$

Summary

- Graph theory
- Binary search trees
- AVL-trees
- (a, b)-trees
- Splay trees

End

Thank you for your attention