Search Trees

Tobias Lieber

April 14, 2008

In this paper I want to introduce the basics on graph theory with respect to trees. But the main part shows up four approaches for organizing data in trees for searching. I will show advantages and disadvantages of these approaches.

1 Graphs and Trees

Definition 1.1 *An (undirected) graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E.*

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

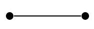
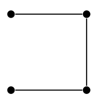
A directed graph $G = (V, E)$ is given by a set of nodes and a set of directed edges:

$$E \subseteq V \times V$$

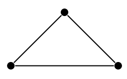
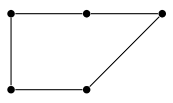
Definition 1.2 *The neigborhood of node x is given by:*

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

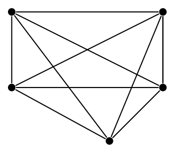
I want to introduce 3 types of graphs which have been given special names due to their significant role in graph theory:
Path:

P_2:  P_4: 

Circle:

C_3:  C_5: 

Complete graph/ Clique:



K_5:

Definition 1.3 *A graph $G = (V, E)$ is called connected, if there is a path from each node x to each other node y.*

Definition 1.4 *A graph $H = (W, F)$ is called subgraph of $G = (V, E)$ if*

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

Definition 1.5 *An acylic graph $G = (V, E)$ does not contain any circle as a subgraph.*

Definition 1.6 *A graph $G = (V, E)$ is called a tree if it is connected and acyclic.*

Definition 1.7 *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 1.8 *The height of a tree $G = (V, E)$ with root $r \in V$ is defined as*

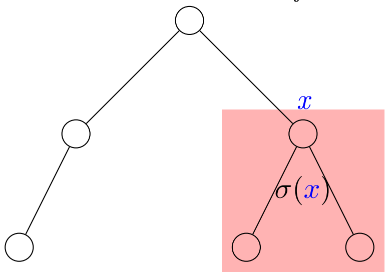
$$h = \max_{x \in V}\{\text{distance from } r \text{ to } x\}$$

Theorem 1 *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 1.9 *A tree $H = (W, F)$ is called a spanning tree of a graph $G = (V, E)$ if $W = V$ and $F \subseteq E$.*

Definition 1.10 *The function $\sigma(x)$ returns the subtree, which is rooted in x:*



2 Search Trees

The central problem we want to consider is given by a set of items $x_1, \ldots, x_n$. Each item contains of a key and a value which can be accessed by

```
x.key and
x.value.
```

All keys x arise from an ordered set S. The goal is to minimize the total access time on an arbitrary sequence of operations. The operations can perfom a test for the existence of a key in the datastructure, insertion or deletion of an element. These operations are generally called IsElement, Insert and Delete.
There are two different concepts of storing items in a tree. An internal search tree stores all keys in internal nodes. It leaves contain no further information. Accordingly there is no need to store them and they can be represented by NIL-pointers.
On the other hand in an external search tree, all keys are stored at the leaves. The internal nodes only contain information for managing the data structure.

3 Binary search trees

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:

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

In a (binary) tree in each node (vertice) we have to store some additional information, pointers to the left and the right child and for search trees with more than two children the number of children. This information can be addressed in the same manner as keys and values. The advantage of binary search trees are their quite simple algorithms.

```
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
}
```

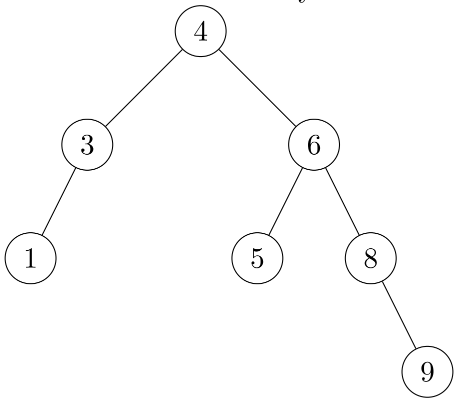
Unfortunately there are sequences of operations such that every operation requires $\Theta(n)$ steps. This leads us to the same worst case complexity like linear lists.

4 AVL-Trees

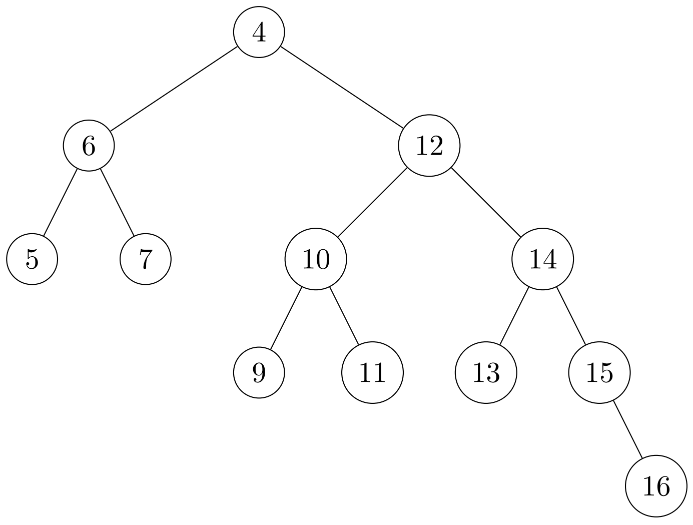
For reducing this worst case complexity AVL-trees have been invented in 1962 by Georgy Adelson-Velsky and Yevgeniy Landis [AVL62]. AVL-trees are represented as internal search trees. The main idea of AVL-trees is to keep the tree height balanced. This means

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

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



is an AVL-tree. But



is not an AVL-tree because the balancing requirement is not fulfilled in node 4.

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

Proof

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

□

Theorem 3 *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 $h \Rightarrow height(\sigma(\text{r.leftChild})) = h-1, height(\sigma(\text{r.rightChild})) = h-2$
$\Rightarrow n(T_h) = 1 + n(T_{h-1}) + n(T_{h-2})$

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

□

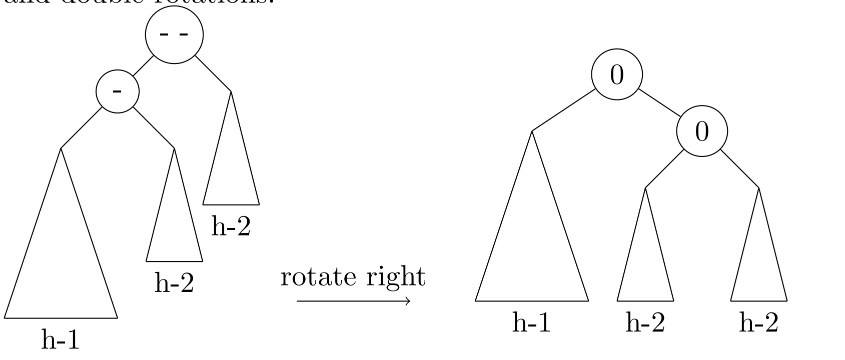
So we know that the number of nodes n of an AVL-tree with height h is limited by:

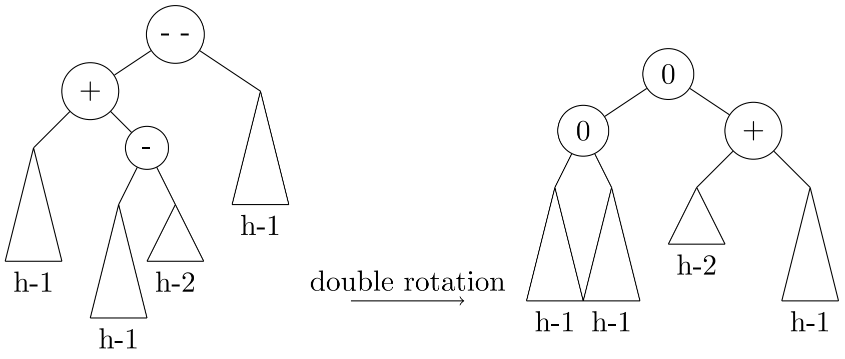
$$n \geq \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2} \tag{1}$$

By taking logarithm we receive:

$$h \leq \frac{\log n}{\log\left(\frac{1+\sqrt{5}}{2}\right)} - \log\left(\frac{1}{\sqrt{5}}\right) - 2 \approx 1.44 \log n + 1.1 \tag{2}$$

So the height of a tree is in $O(\log n)$.
As the structure of an AVL-tree is equalivalent to the structure of a binary search tree we can reuse the main parts of the algorithms we developed in the last section. The IsElement operation is obviously the same as in the last section. For insertion of an item we just have to add a rebalance operation to the end of the Insert operation for binary search trees. The rebalance operation has to check if the tree is unbalanced. For this we can store the balance status of each node with three symbols: $-$ means unbalanced by 1 to the left side, 0 means balanced and $+$ means unbalanced by 1 to the right side. Thus, except symmetry, we get the following two cases, which can be solved by single and double rotations:





The Delete operation on a binary search tree can be fixed in a similar way.
Nevertheless all operations depend on the height of the AVL-tree. As the height is in $O(\log n)$ all the three operations will finish in $O(\log n)$ steps.

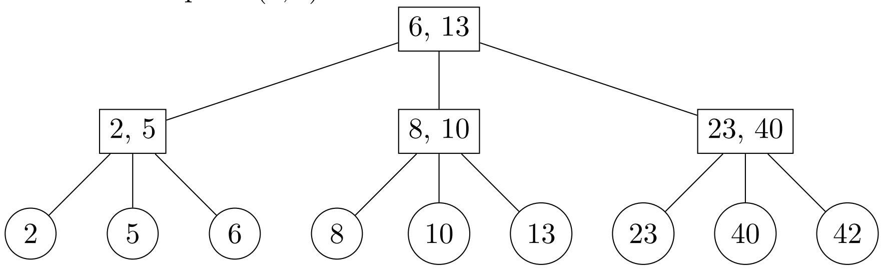
5 (a,b)-Trees

Although the height of AVL-trees is limited by $O(\log n)$ the access times can show large differences, if we assume costs for accessing each level. This assumption is realistic if we consider trees which are stored in the memory of computers. Therefore we would like to have a datastructure in which every node has the same height. One approach are (a,b)-trees which are defined as follows [OW02]:

Definition 5.1 *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.*
- $b \geq 2a - 1$

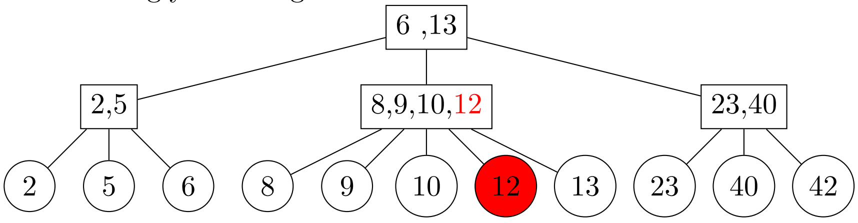
As an example a $(2, 4)$-tree could look like this:



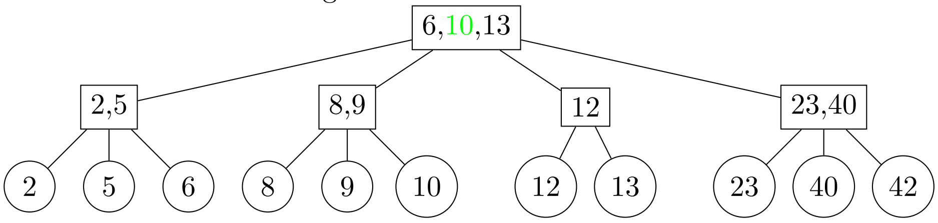
As in the case of binary search trees the algorithms for this datastructure do not seem to be to complicated.

```
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 ($\lceil \frac{m}{2} \rceil$ ... $m$)
  w.add_node($K_{\frac{m}{2}}$,r)
  if(w.children>b)
    rebalance(w)
}
```

Accordingly inserting 12 in the tree



will result in the following tree:



```
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)
  }
}
```

The runtime of the algorthms above is still interesting due to the fact that the Insert, Delete and IsElement operations depend on the height h of an (a,b)-tree with n nodes.

Theorem 4 *Every (a,b)-Tree with height h has*

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

leaves.

Proof

1. In an (a,b)-tree whose 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.

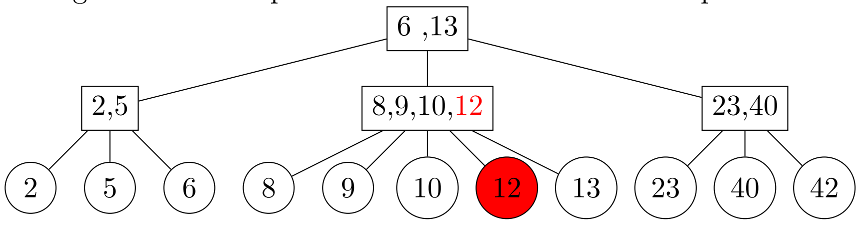
□

Thus we get for the height h:

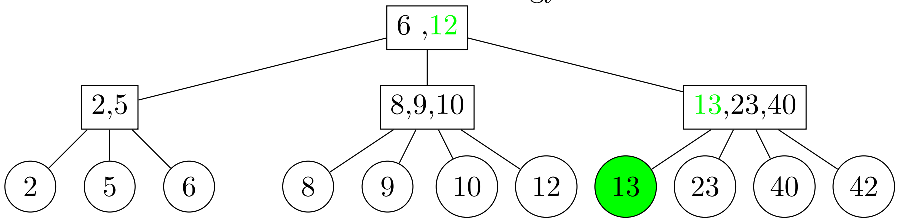
$$log_b n \leq h \leq \log_a \frac{n}{2} + 1$$

This means that all the algorithms above have runtime $O(\log n)$.

Another strategy for rebalancing nodes in an (a,b-tree is the overfull strategy. By this strategy, we try to reduce splitting nodes by moving nodes from overfull nodes to siblings. As an example we can consider our last example:



This would result with the overfull strategy to:



Using this strategy for inserting nodes we receive with small modifications B*-trees.

Definition 5.2 *A B*-tree with order b is defined as follows:*

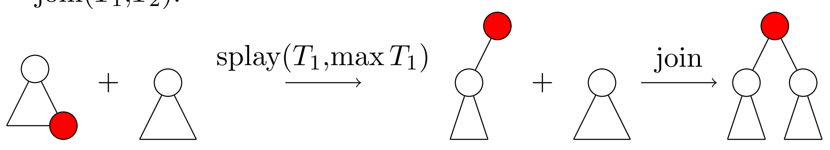
- *All leaves appear on the same level*
- *Every node except when the root has at most b children*
- *Every node except when the root has at least $(2b-1)/3$ children*
- *The root has at least two and at most $2\lfloor(2m-2)/3\rfloor+1$*
- *Every internal node with k children contains $k-1$ keys*

For being able to use the new instertion strategy we have to modify the case if two siblings are full. In this case we split these two siblings into three nodes with at least $(2b-1)/3$ children.
Comparing this strategy to our old algorthims for (a,b)-trees it turns out, that B*-trees have a filling factor of 87% instead of 67% [Knu98]. This means that the IsElement operation in total is faster, since the height of a B*-tree and an (a,b)-tree containing the same nodes is smaller. But as the Insert and Delete operation are slower since it is more difficult finding siblings and splitting up two nodes into three nodes. But in total, in applications like databases, we receive a faster access, since there are more IsElement queries than Insert and Delete queries.

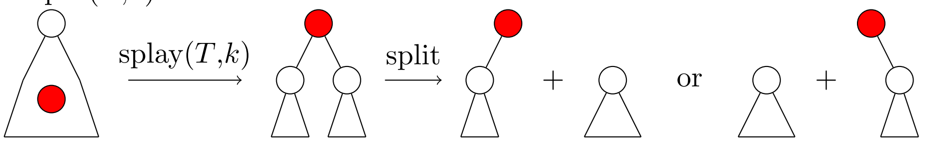
6 Splay-Tree

Since our algorithms get more complex, it is appropriate to think on easier algorithms. To this approach we achieve self organizing datastructures whose additional target is minimizing the costs. A good example in which self organizing datastructures succeeded are linear lists, where the move to the front rule is proveably just twice as bad in an amortized analysis as an optimal algorithm.
Let us consider splay trees. Splay trees are internal binary search trees. Similar to the move to the front rule, Tarjan introduced a move to the root rule, which moves a node x to the root of a tree T with respect to the properties of binary search trees [ST85]. This mechanism we can apply to a tree with a function splay(T,x). For the moment we do not care how this works, we first want to look what we can use this splay function for.
First let there be two help functions join and split which can join two splay-trees or alternativley split a tree into two trees.

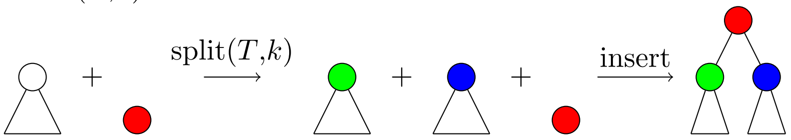
join(T_1,T_2):



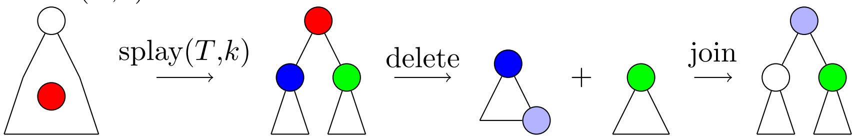
split(T,k):



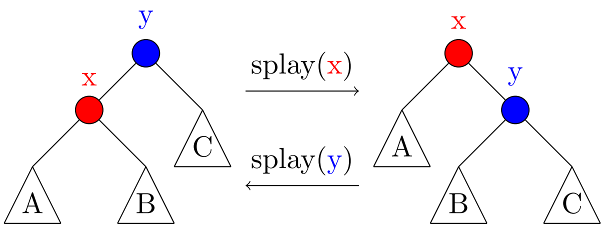
With these two help functions we can realize easily an Insert and Delete operation:
insert(T,k):



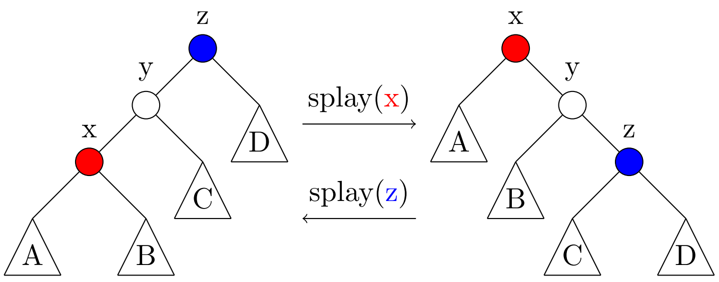
delete(T,k):



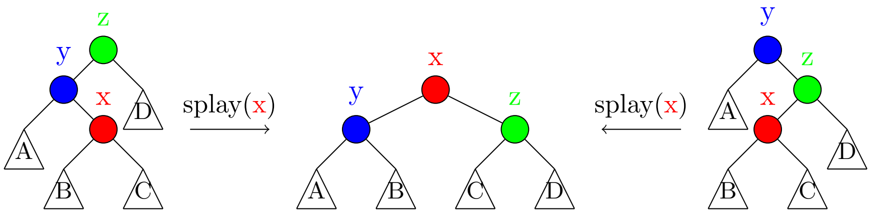
But how does splay really work? Splay(T,x) uses single and double rotations for transporting node x to the root of a splay tree T.
Three different cases may occur, when moving a node to the root:
Case 1:



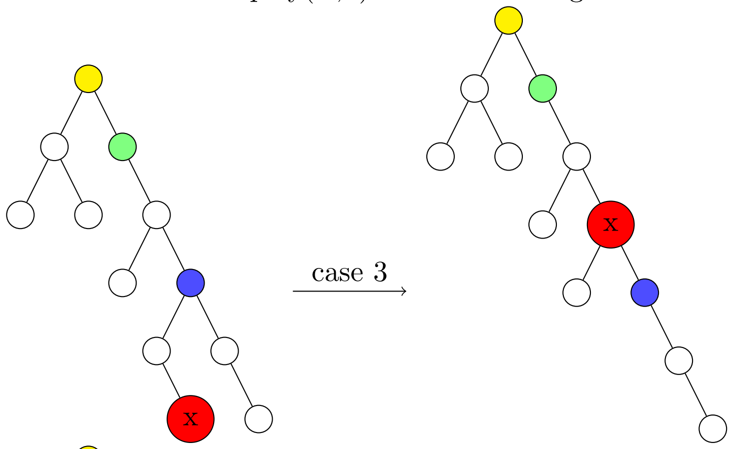
Case 2:

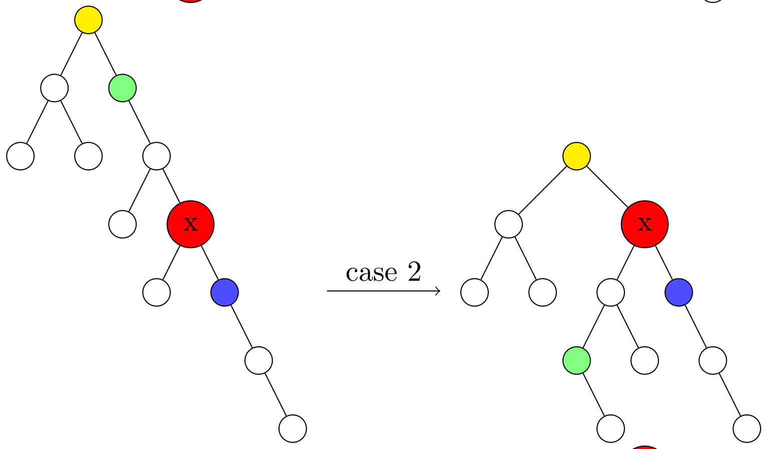


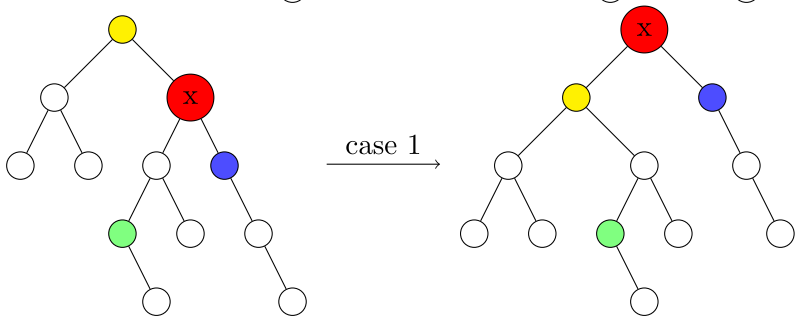
case 3:



Let us consider splay(T,x) on the following tree:







For analysing the costs of splay, we use amortized analysis. In amortized analysis of algorithms we investigate the costs of m operations.

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

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

a represents the amortized costs, t the actual costs and Φ is a potential function which we can choose. 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: $\Phi = \sum_{i \in T} r(i)$

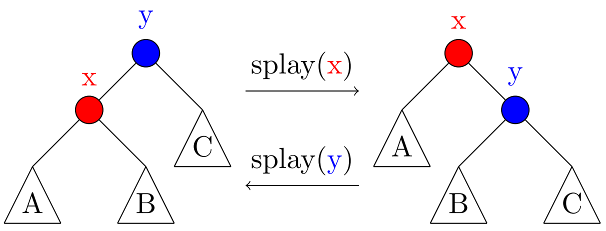
Theorem 5 *Splay(T,x) needs at most*

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

amortized time, where v is the root of T.

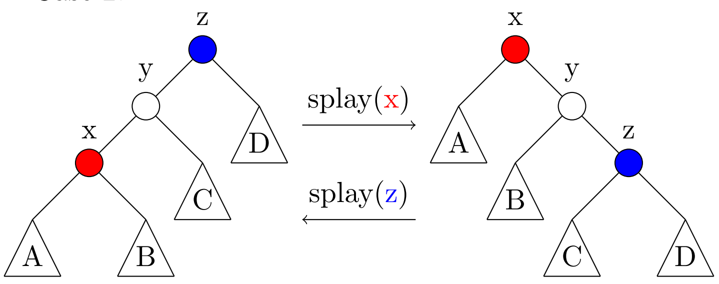
Proof We can divide the splay operation in the rotations which are the influential operations in splay. Thus we consider the number of the rotations.
Just one more notation:
Let $r(x)$ be the rank of x before the rotation and $R(x)$ the rank after the rotation. Let $s(x)$ be the size of x before the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

Case 1:



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

Case 2:



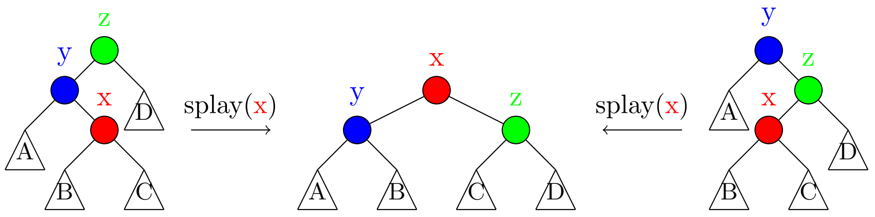
$$\begin{array}{rccc} & 2 + R(x) + R(y) + R(z) - r(x) - r(y) - r(z) & \\ = & 2 + R(y) + R(z) - r(x) - r(y) & \text{since } R(X) = r(z) \\ \leq & 2 + R(x) + R(z) - 2r(x) & \text{since } R(y) \leq R(x) \\ & & \text{and } r(x) \leq r(y) \end{array}$$

Claim:

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

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

Since the log function is convex we know that the claim holds.
Case 3:



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

This holds with the claim we used in case 2. By adding all rotations we used for splay(T, x) we receive a telescope sum, which yields us the amortized time $\leq 3(R(x) - r(x)) + 1 = 3(r(t) - r(x)) + 1$. □

Thus we can perform a splay operation in $O(\log n)$ time in amortized sense. This means we can perform the Insertion and Deletion of nodes in $O(\log n)$ amortized time, too.

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 6 *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 $t = a - \Phi_m = m(1 + 3\log n) + n\log n$ □

7 Summary

After an short introduction to graph theory four approaches for organizing data in trees were introduced. Binary search trees and splay trees have simple algorithms while AVL-trees and (a,b)-trees are more complicated to implement. As AVL-trees, splay trees and (a,b)-trees have logarithmic runtime for the operations IsElement, Insert and Delete in at least amortized sense, for binary search trees there exist sequences of operations which run in $\Theta(n)$ steps.

References

[AVL62] G.M. Adel'son-Vel'skii and E.M. Landis. An algorithm for the organization of information. *Sov. Math., Dokl.*, 3:1259–1262, 1962.

[Knu98] Donald E. Knuth. *The Art of Computer Programming Volumes 1-3 Boxed Set.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[OW02] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen. 4. Auflage.* Spektrum Akademischer Verlag GmbH, Heidelberg-Berlin, 2002.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.