CS 225 – Data Structures

ZJUI – Spring 2022

Lecture 7: Search Trees

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

7 Tree Data Structures

We now proceed to non-linear data structures, mainly emphasising trees and graphs

The emphasis is first on search, for which we will first explore simple **binary search trees**, then proceed to more sophisticated balanced trees such as **AVL trees** and **splay trees**

As much as time allows we will also look into some advanced concepts such as (a, b)**-trees**, **red-black trees** and **K-d trees**

We will also encounter the simple TRIE data structure supporting search for keys in a large dictionary

7.1 Binary Search Trees

For search trees we will label vertices by elements of a set—to support efficient search we assume again that the set of labels is ordered

A ***binary search tree*** (**BST**) over a totally ordered set T is a binary tree, in which vertices v are labelled by elements $\ell(v) \in T$ such that for any vertex v we have:

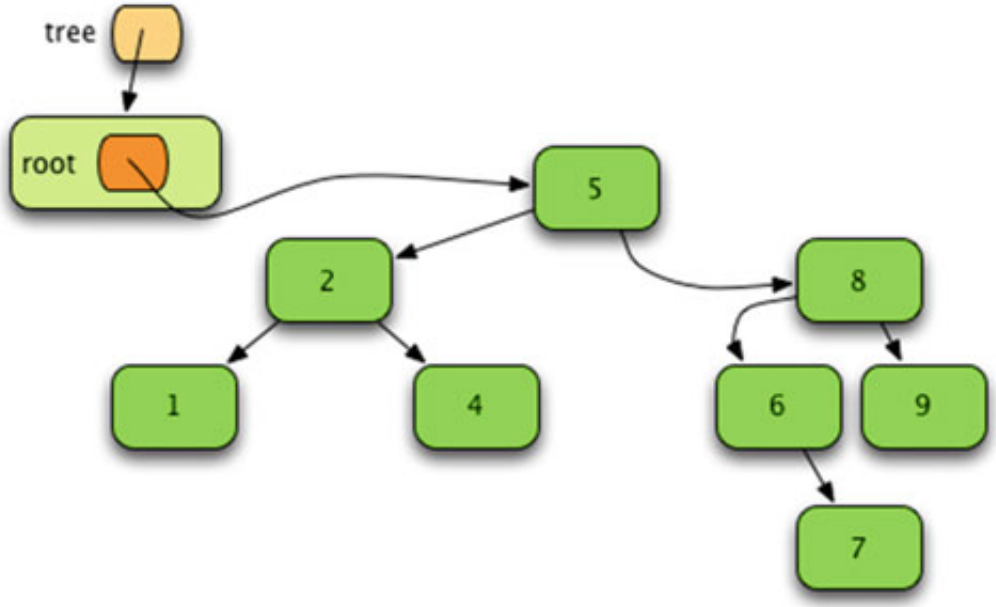
- If v' a vertex in the left successor tree of v, then $\ell(v') < \ell(v)$
- If v' a vertex in the right successor tree of v, then $\ell(v') \geq \ell(v)$

The definition can be generalised to partially ordered sets of labels

The empty tree is considered also to be a BST, though only for technical reasons

Example: BST

The tree below is a BST (with elements in $\mathbb{N}$) with eight vertices clearly satisfying the required conditions



The additional dummy node is used to enable the representation of an empty BST

We will see how to construct such BSTs, search in them and delete nodes in them

Insertion into a BST

Insertion of a new value $e \in T$ can be done recursively starting from the root of the tree

If the tree is empty, then create a new vertex without successors and let the tree with only this vertex be the new BST

If the tree is not empty, then consider the label $e' \in T$ of the root

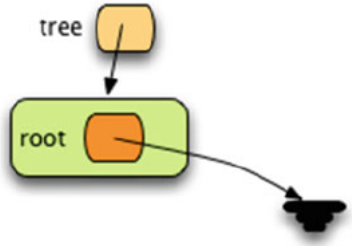
- If $e' < e$ holds, then insert e into the left sucessor tree (which may be empty)
- If $e' \geq e$ holds, then insert e into the right sucessor tree (which may be empty)

Insertion is used to create a BST in a step-by-step way

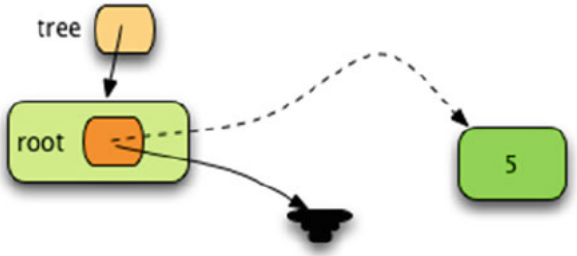
The following slides illustrate step-by-step how the insertion works—we create a BST with the elements $5, 8, 2, 1, 4, 9, 6, 7$

Example / Illustration

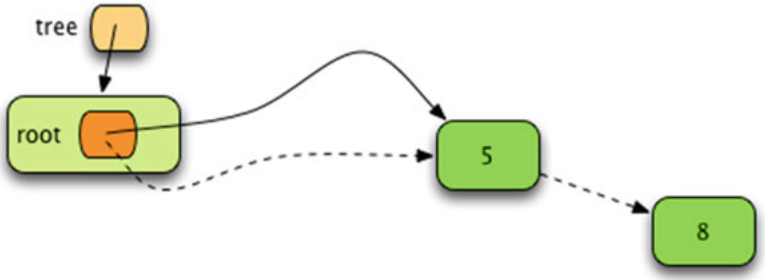
Initially empty BST



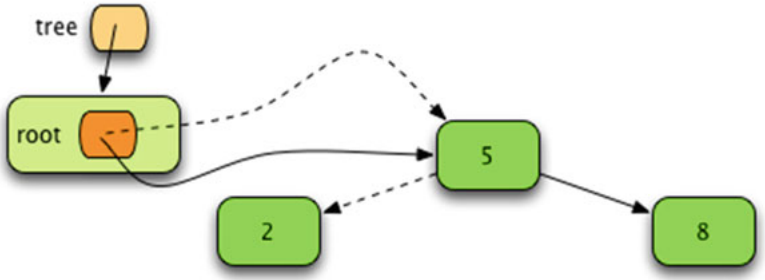
BST after insertion of 5



BST after insertion of 8



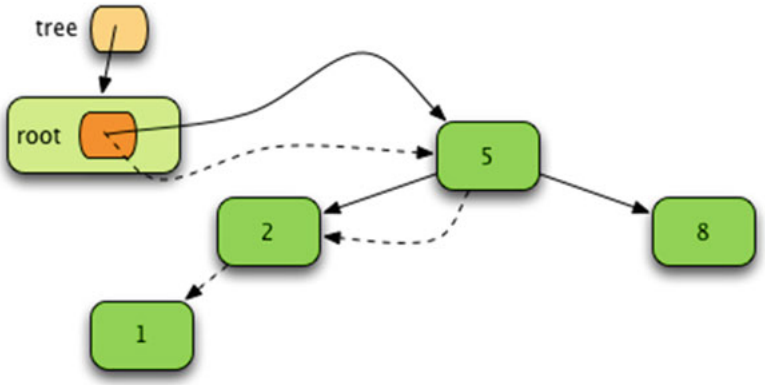
BST after insertion of 2



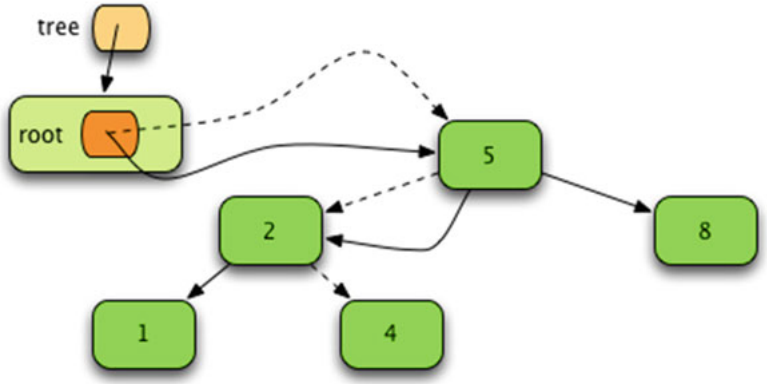
Dashed arrows mark newly introduced edges (mostly those that already exist)

Illustration / cont.

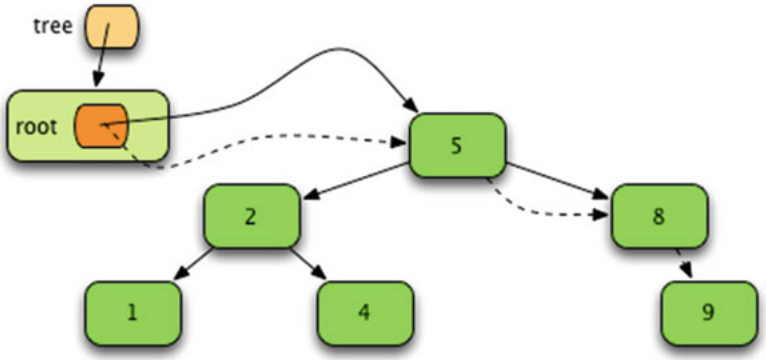
BST after insertion of 1



BST after insertion of 4



BST after insertion of 9



BST after insertion of 6

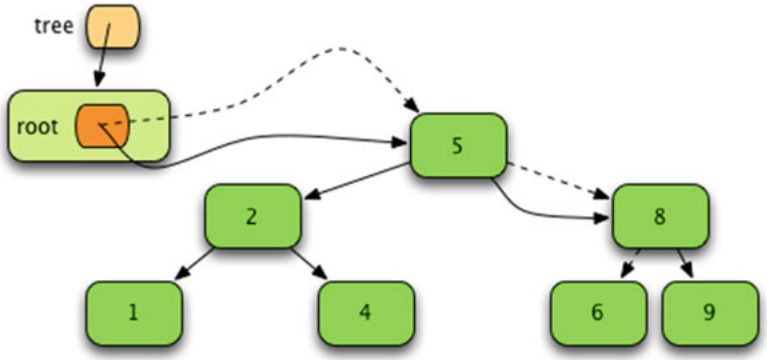
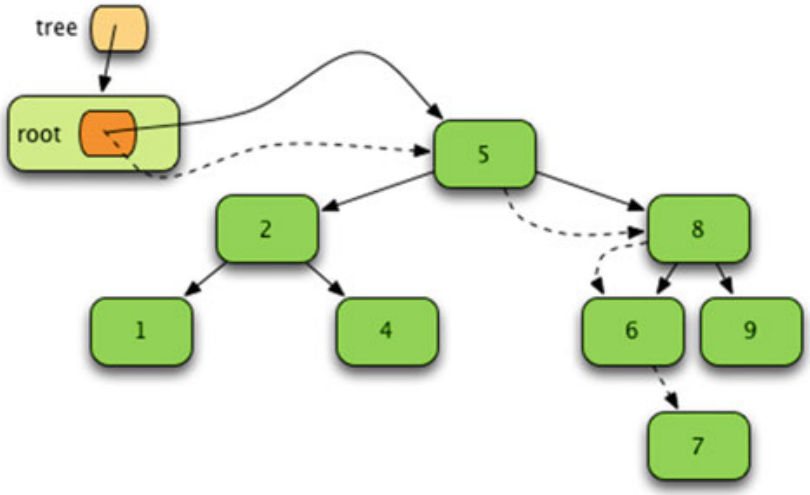
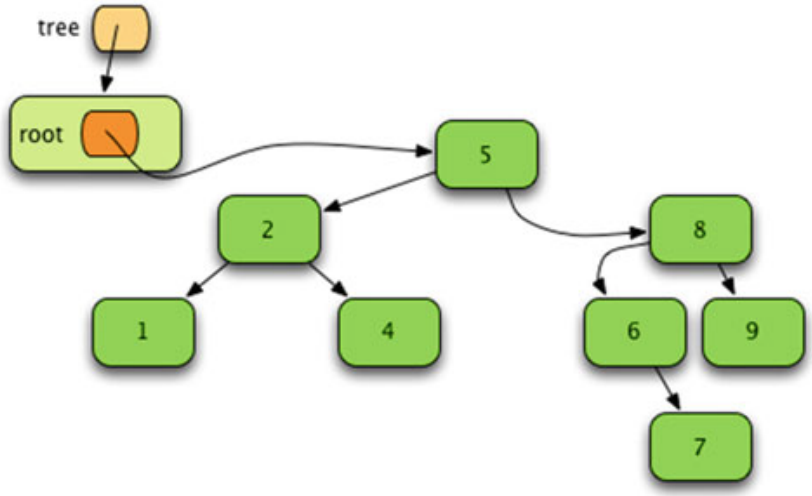


Illustration / cont.

BST after insertion of 7



Final BST



Retrieval Operation on a BST

In order to find an element e in a BST proceed analogously

If the BST is empty, return FALSE

If the BST is non-empty, compare the label e' of the root with e

- If $e = e'$ holds, return TRUE
- If $e < e'$ holds, proceed recursively searching for e in the left successor subtree of the root
- If $e > e'$ holds, proceed recursively searching for e in the right successor subtree of the root

Worst Case Complexity

Consider the time complexity of the *insert*, *find* and *delete* operations on a BST

In the worst case a BST degenerates to a (singly) linked list, e.g. if the elements to be inserted are sorted, this will result

In this case finding an element e in the BST requires comparisons with all elements, so we get a complexity bound in $O(n)$, where n is the number of elements

Likewise, for insertion we may have to follow the path from the root to a leaf, and if the BST is degenerated, then this requires time in $O(n)$

This also applies to the *delete* operation: if we find the element to be deleted associated with a non-leaf vertex, then we have to find a leaf in the right successor tree to insert the left successor tree in the *__merge* operation, so again we obtain a worst case complexity bound in $O(n)$

Therefore, binary search trees are merely an "appetiser" for further considerations on tree data structures—they are practically irrelevant

Average Case Complexity

Our considerations actually show that the complexity of *insert*, *find* and *delete* is determined by the length of the path from the root to an element e in the BST

Let I_n denote the sum of these path lengths in a random BST with n elements, call this the **internal path length** of the BST

Then $I_n = I_{n_1} + I_{n_2} + n - 1$, where n_1 and n_2 are the number of vertices in the left and right successor trees of the root

The summand $n - 1$ comes from the fact that each path length of the $n - 1$ nodes in both subtrees is extended by 1

From this we get $I_n = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} I_i$

Average Case Complexity / cont.

This gives $n \cdot I_n = n(n-1) + 2\sum_{i=0}^{n-1} I_i$

Replacing n by $n-1$ gives us $(n-1) \cdot I_{n-1} = (n-1)(n-2) + 2\sum_{i=0}^{n-2} I_i$

The difference of the two equations gives $n \cdot I_n - (n-1)I_{n-1} = 2I_{n-1} + 2(n-1)$

and further $n \cdot I_n = (n+1)I_{n-1} + 2(n-1)$

Average Case Complexity / cont.

We rewrite this equation in the form $\frac{I_n}{n+1} = \frac{I_{n-1}}{n} + \frac{2(n-1)}{n(n+1)}$

Using induction this gives us $I_n = 2 \cdot (n+1) \sum_{i=1}^{n} \frac{i-1}{i(i+1)}$

So we have $I_n \approx 2 \cdot n \int_1^n \frac{1}{x} dx \approx 2 \cdot n \log n$

That is, on average the path length in a BST is in $O(\log n)$, which implies that the average case complexity of *insert*, *find* and *delete* on a BST is in $O(\log n)$

7.2 AVL Trees

The term AVL stands for Adelson-Velskii & Landis, two Russian computer scientists who introduced this type of trees

We have seen that on average *insert*, *find* and *delete* on a BST have a sublinear complexity, but this is no guarantee for efficiency: the worst case complexity is in $O(n)$

That is, if a BST is degenerated—it is (almost) a linked list—a BST offers no advantages over lists that can be sorted efficiently and then searched efficiently (and with reduced space)

We have also seen that the length of paths from the root is decisive for the complexity—this depends on the height of the tree

Clearly, we can construct a BST with n vertices that has a height $\lfloor \log_2 n \rfloor$

We call a tree with n vertices and height $\lfloor \log_2 n \rfloor$ (perfectly) ***balanced***

Almost Balanced Trees

If a BST is perfectly balanced, then also the maximal path length from the root to a vertex is $\log n$, and the worst case complexity for *insert*, *find* and *delete* will be in $O(\log n)$

However, perfectly balanced trees are hard to build and even harder to maintain—almost each insertion or deletion requires a reorganisation of the tree

Taking the complexity of such a reorganisation into account, the amortised complexity of *insert* and *delete* will no longer be in $O(\log n)$

An ***AVL tree*** is a binary search tree such that for every vertex v the height of the left and right successor trees differ at most by 1

That is, AVL trees are "almost balanced trees", and we will show that this characteristic property of AVL trees can be efficiently maintained

Implementation Alternatives

We define the **balance** of a vertex v in a binary tree as the difference $bal(v) = height(t_{right}) - height(t_{left})$, i.e. the difference of the heights of the right and left successor trees

In an AVL tree we have $bal(v) \in \{-1, 0, 1\}$ for all vertices v; we have $bal(v) = -1$ iff the left successor tree has a greater height than the right one, and $bal(v) = +1$ iff the right successor tree has a greater height than the left one

We then have the following implementation alternatives:

Variant 1. In an implementation we may store the balance of a node in the node itself and use this for reorganisation, when necessary

Variant 2. In an implementation we may store the height of a node in the node itself and use this for reorganisation, when necessary

We will explore variant 1

Insertion into an AVL Tree

We will now look into an iterative algorithm for inserting an element into an AVL tree—this algorithm is also used to create an AVL tree

We will keep the description rather informal—details are handled in the **C++** implementation in the labs

First we proceed in the same way as for insertion into a BST, i.e. we follow the existing nodes in the tree comparing the value to be inserted with the values stored in the nodes

If smaller, continue with the left successor tree; if larger, continue with the right successor tree

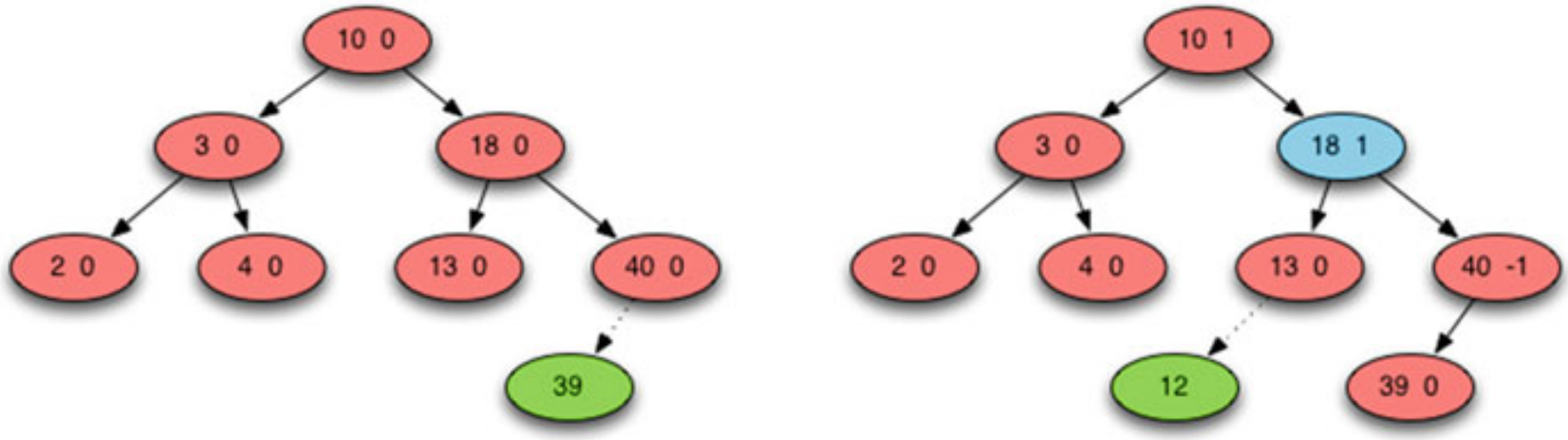
When we do not detect the value already in the tree, we create a new leaf node with the new value and balance 0

We keep track of all nodes on the path from the root to the new leaf storing them in a stack

Update of Balance Values

Next we update the balance values on the nodes in the stack

Depending on whether we had a left or right sucessor, we have to decrement or increment the balance value



If the new balance becomes 0, we can stop—there is no need to work through all elements in the stack

Update of Balance Values / Single Rotation Case

However, we may reach a node, where the update of the balance would give a value -2 or 2 violating the characteristic property of an AVL tree

If this is the case, the node is the last on the path from the root to the new leaf that had a balance -1 or 1; all following nodes have a balance 0

We vcall the successor of this node the **bad child** and the successor of the bad child the **bad grandchild**—where the successor is taken according to the path from the root to the new leaf

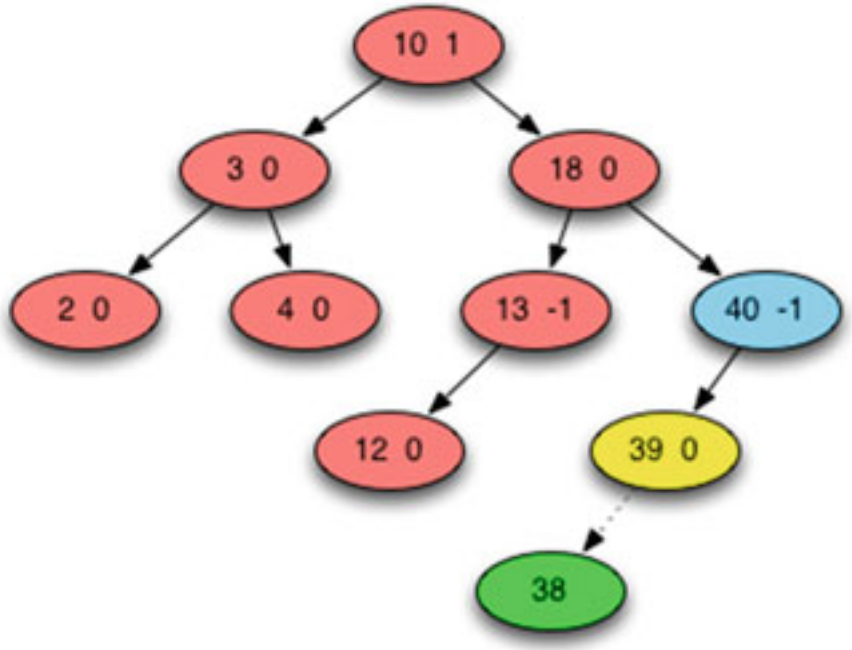
We have a **single rotation case** if the new element is inserted in the same direction as the imbalance (both left or both right)

Otherwise, we have a **double rotation case**

Update of Balance Values / Single Rotation Case

The following example tree shows the single rotation case: the insertion of 38 violates the AVL tree property for node labelled by 40

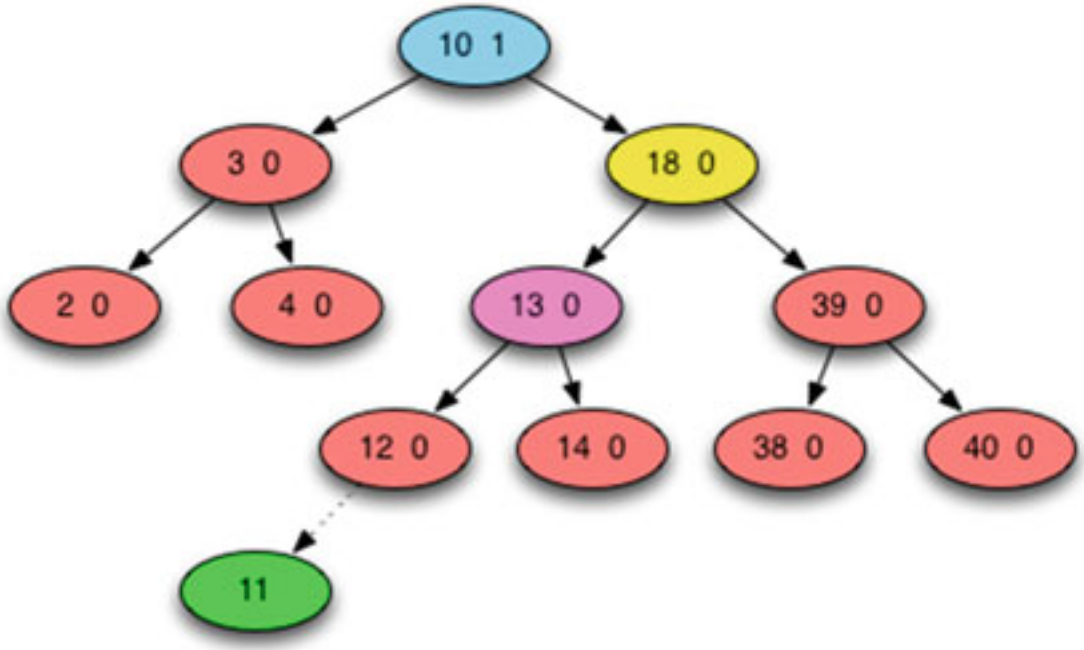
In this case the successor node labelled by 39 is the *bad child* and its successor (on the path to the new leaf) labelled by 38 is the *bad grandchild*



Update of Balance Values / Double Rotation Case

The following example tree shows the single rotation case: the insertion of 11 violates the AVL tree property for node labelled by 10

In this case the successor node labelled by 18 is the *bad child* and its successor (on the path to the new leaf) labelled by 13 is the *bad grandchild*



Single and Double Rotation

In a single rotation case we have to swap the positions of the bad child and its predecessor, and adjust the subtrees and balances accordingly

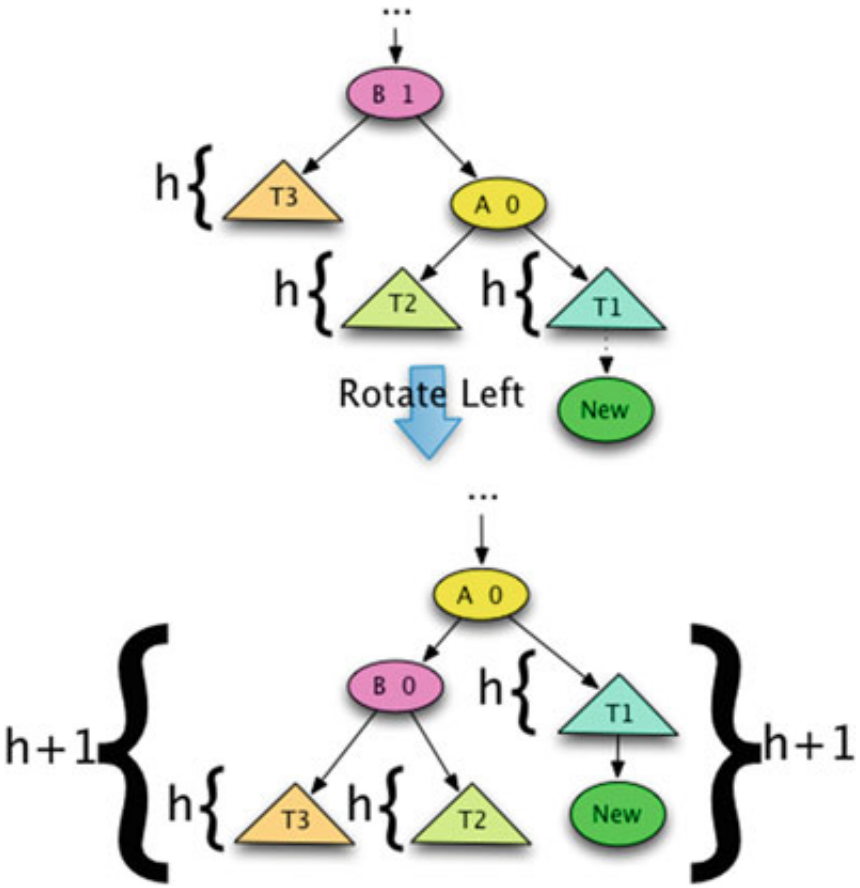
For this we use a **left** or **right rotation operation**

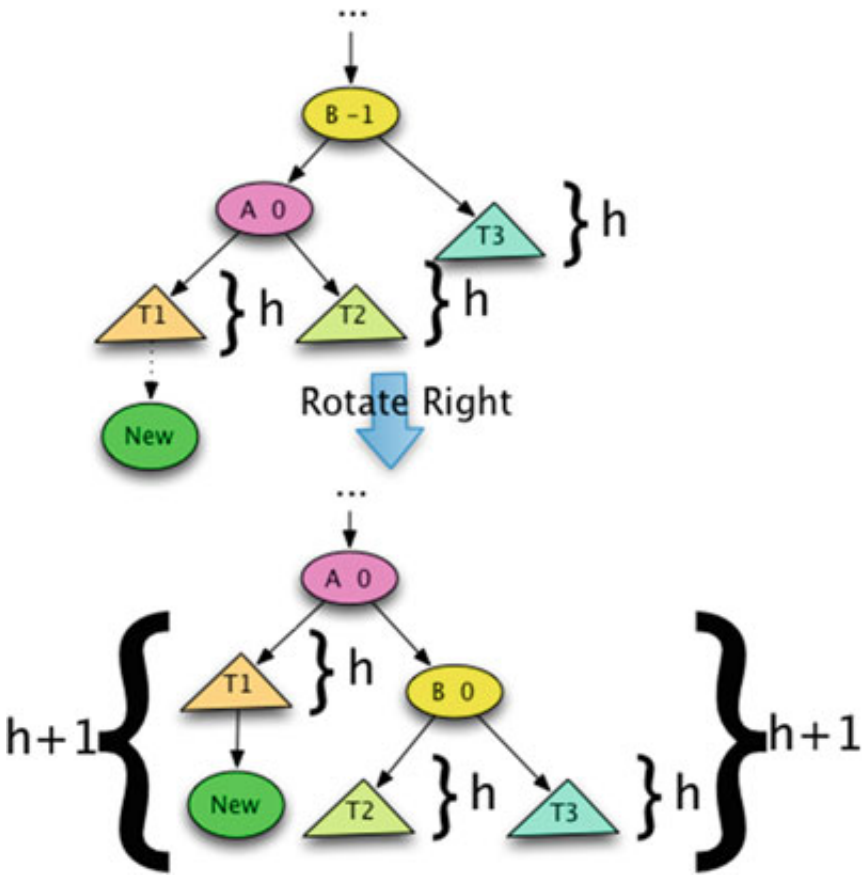
In a double rotation case we first swap the positions of the bad child and the bad grandchild and adjust the subtrees and balances accordingly

Then we swap the positions of the bad grandchild and its predecessor and adjust the subtrees and balances accordingly

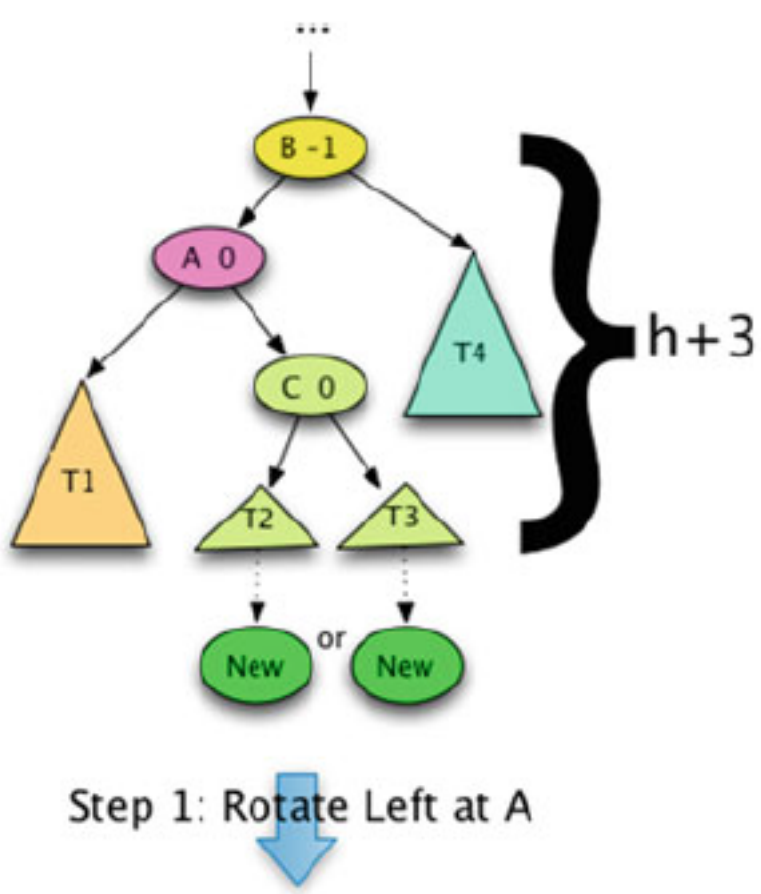
That is, in a double rotation case we apply a left rotation operation followed by a right rotation operation or the other way round

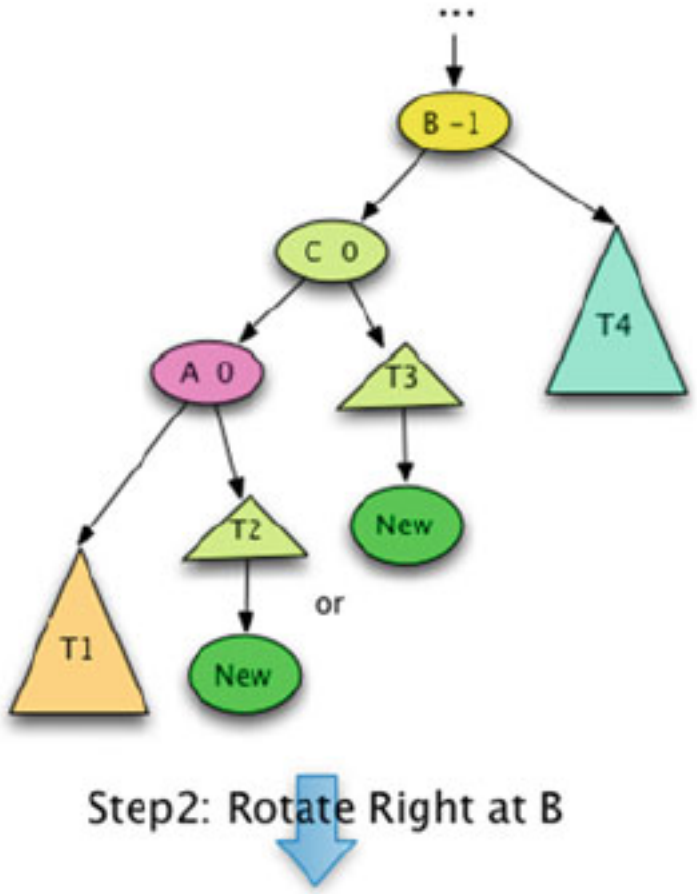
Illustration of Rotation Operations



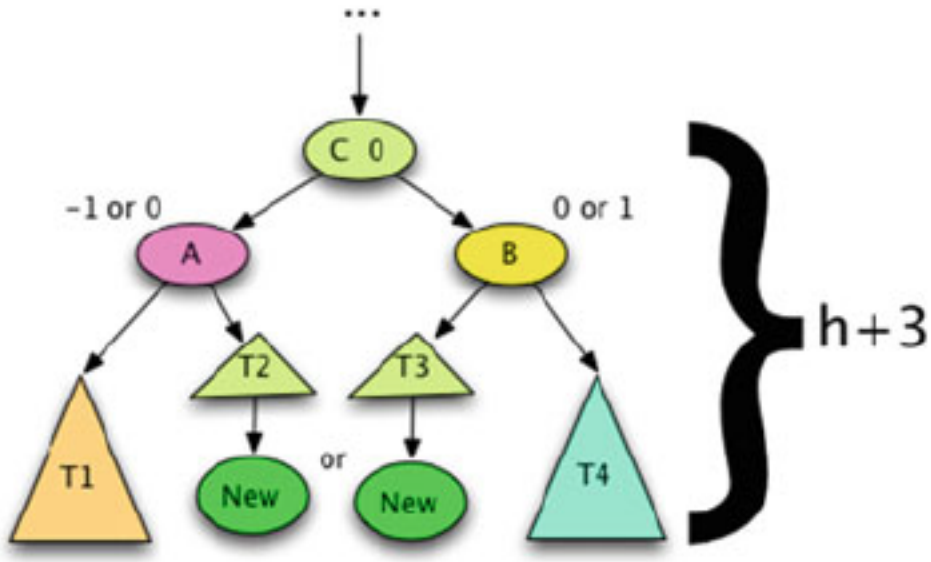


Balance Adjustment in Double Rotation Case





Balance Adjustment in Double Rotation Case / cont.



We see that the previous bad grandchild C always has a balance 0, while for the node B and the bad child A the new balances are either 0 and -1 or 1 and 0

The case of a right rotation followed by a left rotation is dual

The *find* operation is again completely analogous to the case of binary search trees:

A Note on Complexity

We can benefit from the thorough investigation of complexity for BSTs, however in an AVL tree t with n nodes we have $height(t) \in O(\log n)$

In order to find an element in an AVL tree t we simply follow the path from the root to the element, if it exists—so the complexity is in $O(height(t)) = O(\log n)$

In order to insert an element into an AVL tree we first follow a path from the root to a leaf and insert a new leaf—this requires time in $O(height(t)) = O(\log n)$, as all the additional operations such as insertion into a stack and labelling a node as bad child take constant time

The second part of an insertion works one-by-one on the nodes collected in the stack, which again requires time in $O(height(t)) = O(\log n)$

Adjusting balances and performing, if necessary, one or two rotations requires constant time

In summary, the time complexity for *insert* and *find* in an AVL tree with n nodes is in $O(\log n)$

AVL Tree Delete Operation

In order to delete an element from an AVL tree, we first search for it—however, we keep track of the nodes visited on the path to the one to be deleted including the potential information how the balance might change

Once the node is found, it is replaced by None, if it is a leaf

Otherwise, either the minimum in the right successor tree or the maximum in the left successor tree is swapped with the element to be deleted, and the corresponding leaf is deleted

When searching in the left or right successor tree, the stack is extended by the nodes visited and the information how the balance might change

Finally, the created stack is used to work back to the root, readjust balances, and apply single or double rotations when necessary to restore the AVL tree property

Using the same arguments as for *insert* we see that the worst case and average case time complexity of *delete* on AVL trees is in $O(\log n)$, where n is the number of nodes in the tree

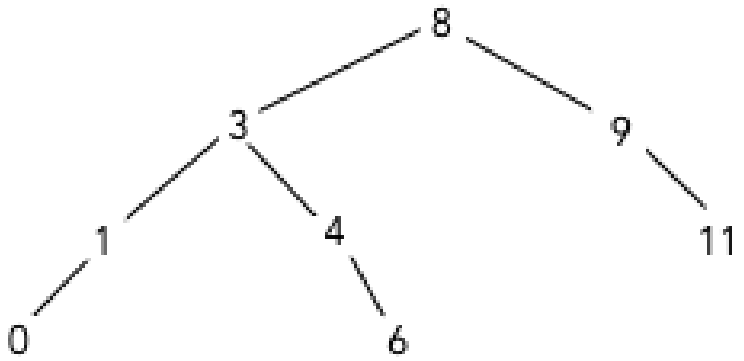
Example (Insertion)

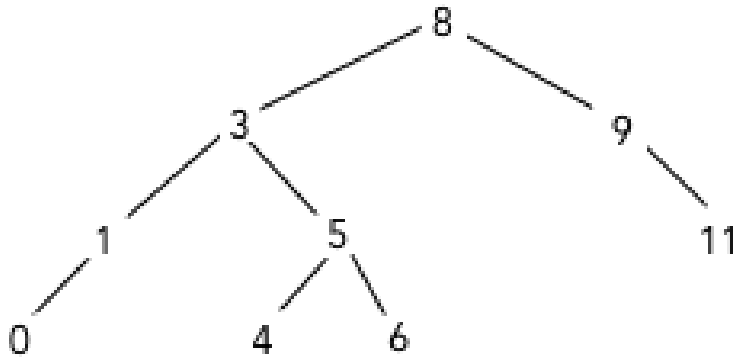
Let us insert successively the elements $8, 4, 9, 3, 1, 11, 0, 6, 5, 2, 12, 13, 10$

Inserting 8, 4, 9 and 3 is straightforward, then the insertion of 1 requires a single right rotation

Example / cont.

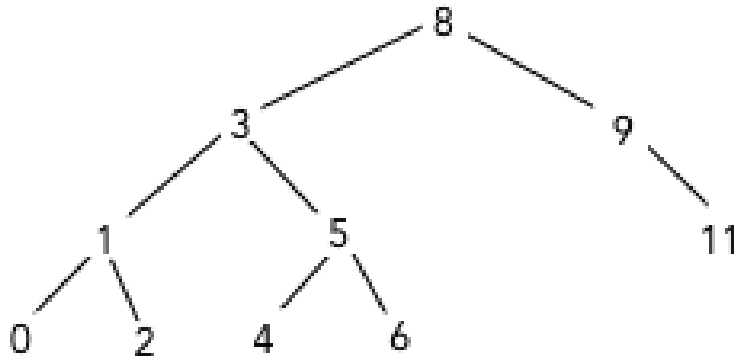
The following insertions of 11, 0 and 6 are again straightforward, but the insertion of 5 requires a single left rotation

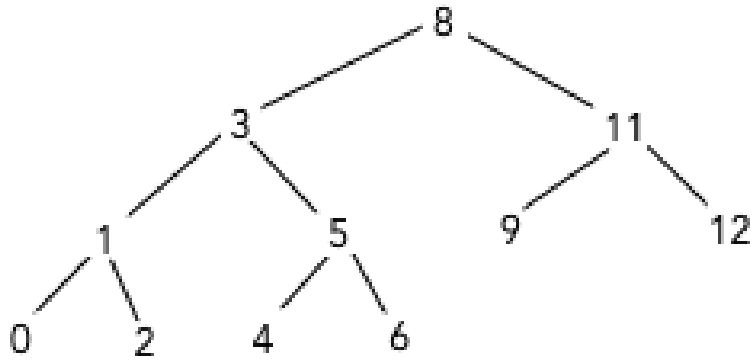




Example / cont.

Again, the following insertion of 2 is straightforward, but the insertion of 12 requires a single left rotation

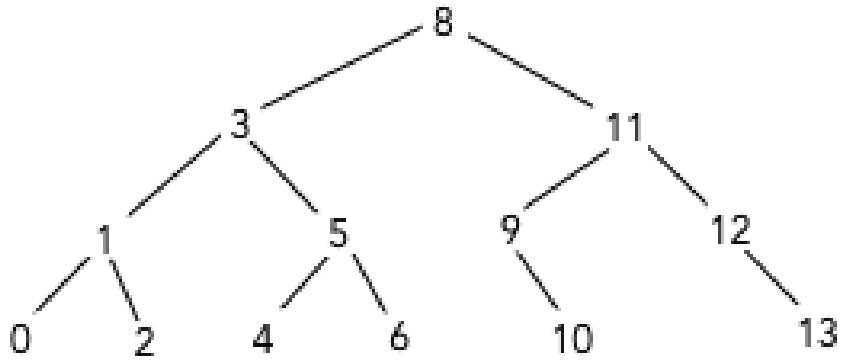




The final insertions of 13 and 10 are again straightforward

Example (Deletion)

The resulting AVL tree is



Let us now delete 3

For this we first find the 3 in the tree, then we search for the minimum in the right successor tree, so we find 4

We swap the 3 and the 4, delete the leaf (with the 3), and only have to adjust the balance of the node labelled 5

7.3 Splaying

As worst case and average case complexity of AVL trees is in $O(\log n)$, they are optimal for search using binary trees

However, they require space for the storage of balance values (or heights), and the (constant) time required for rotation operations is not negligible

Splaying adopts these rotation operations without caring about the balances of the nodes in the binary search tree

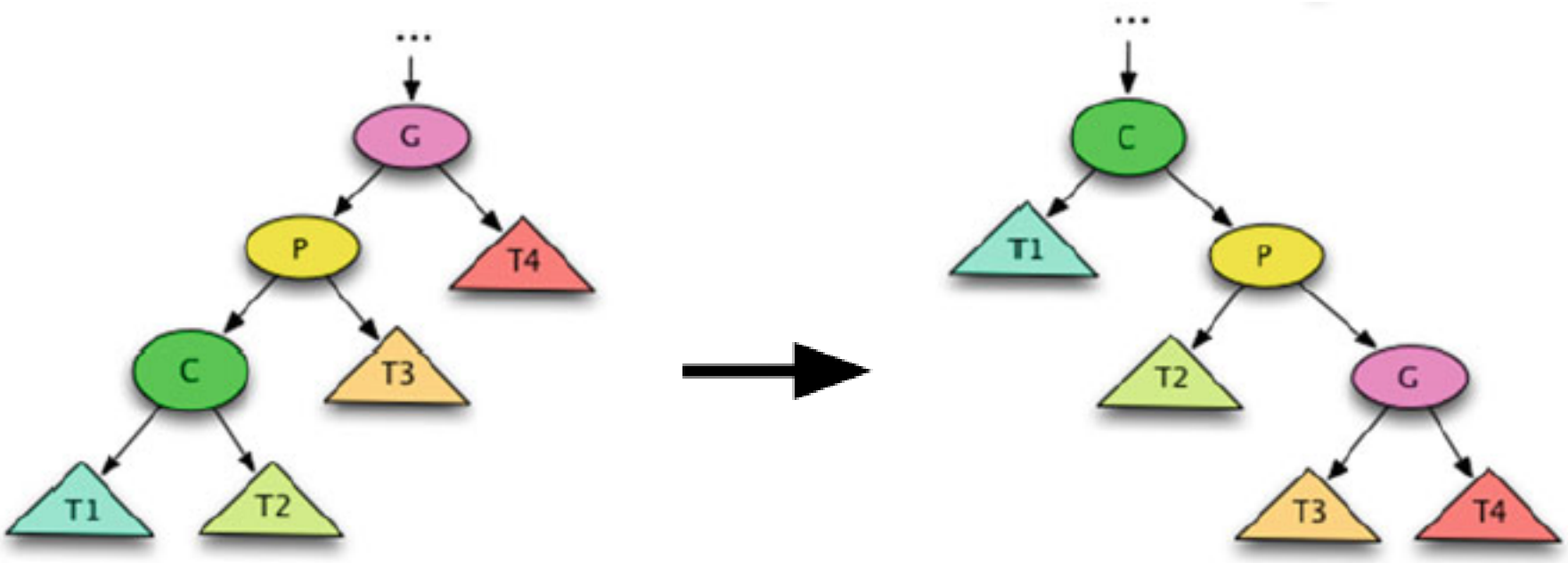
The principle is that whenever an element e is inserted or e has been searched for, it is moved by a sequence of rotations to the root of the tree—deletions can be handled as for BSTs

We illustrate these **splaying (rotation)** operations

An implementation of splaying operations is similar to AVL trees (yet simpler) and therefore left as an exercise

Right-Right Splaying Operation

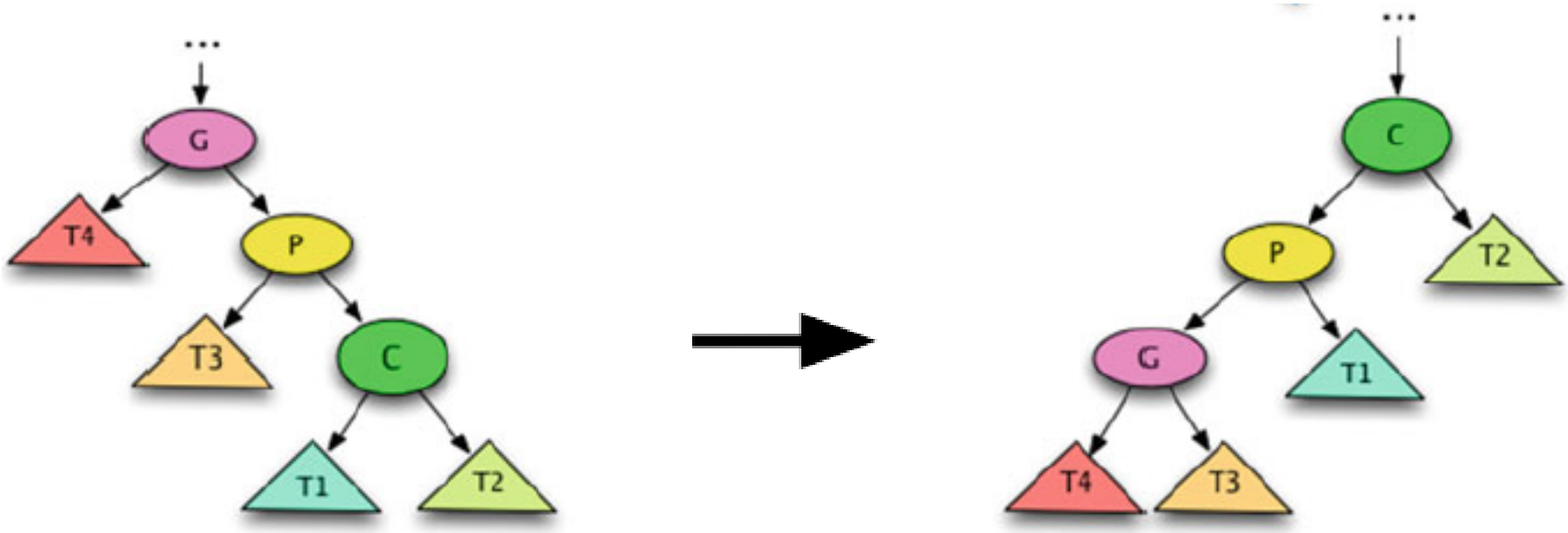
The ***right-right splaying*** operation cyclically swaps a node C, its parent P, and its grandparent G



Successor trees are shifted accordingly

Left-Left Splaying Operation

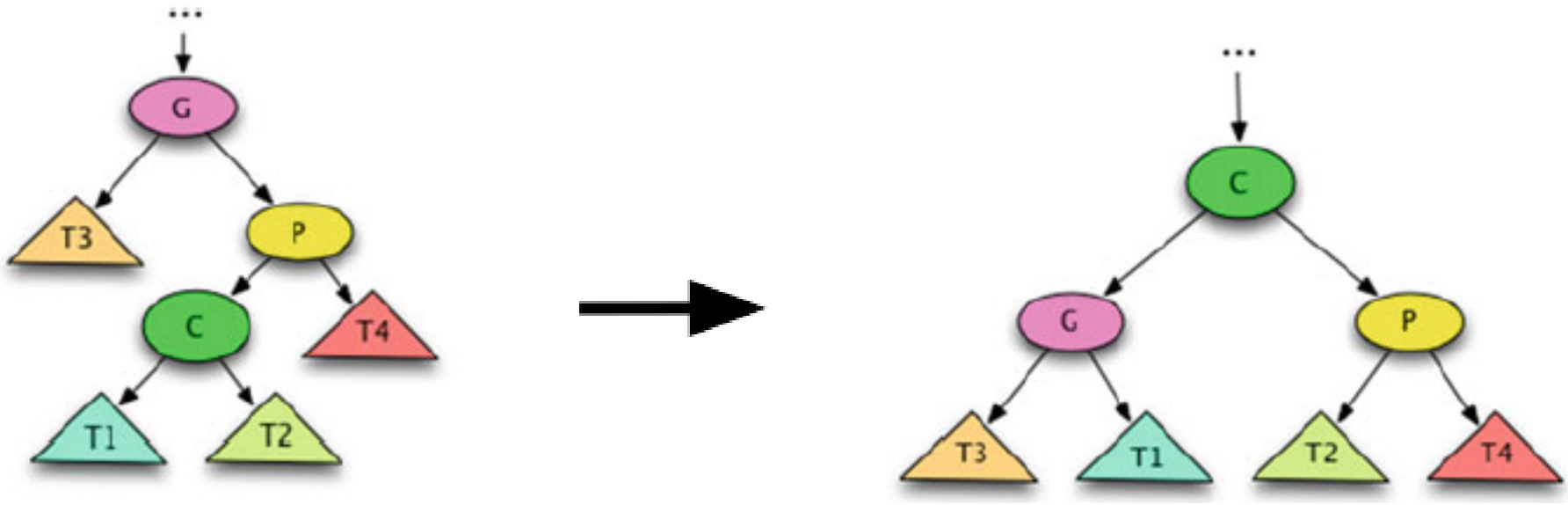
The ***left-left splaying*** operation is the inverse of the right-right splaying operation



Successor trees are shifted accordingly

Right-Left Splaying Operation

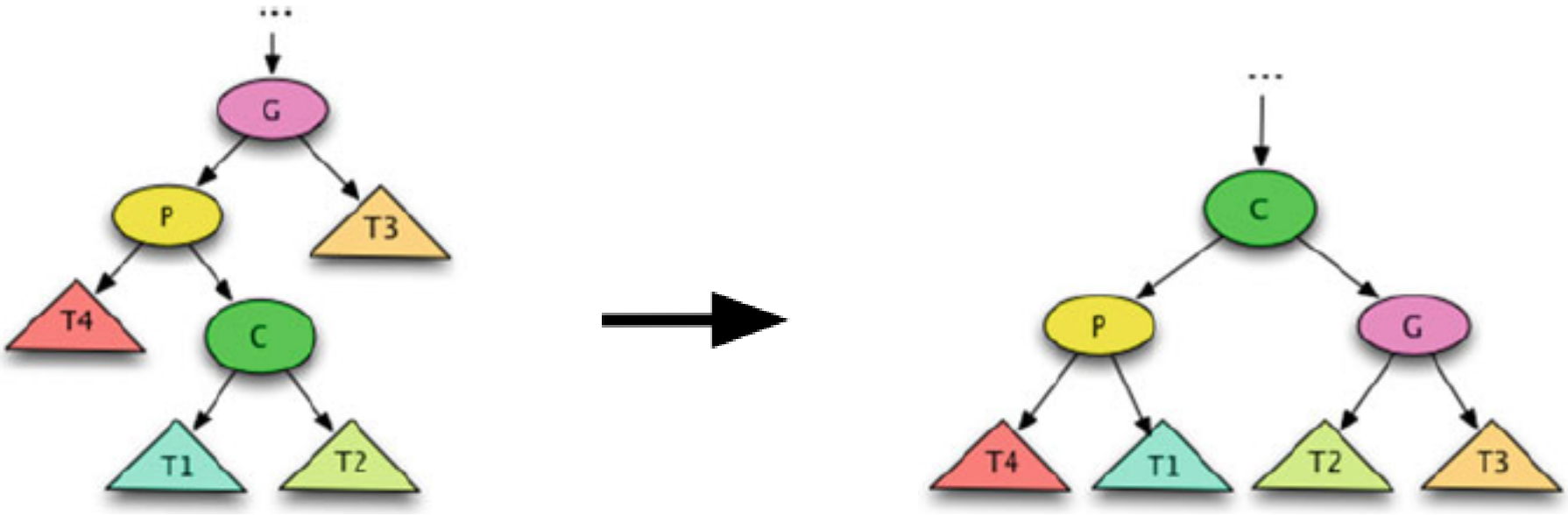
The ***right-left splaying*** operation swaps a node C to the top and make its grandparent G and its parent P its left and right successors, respectively



Other successor trees are shifted accordingly

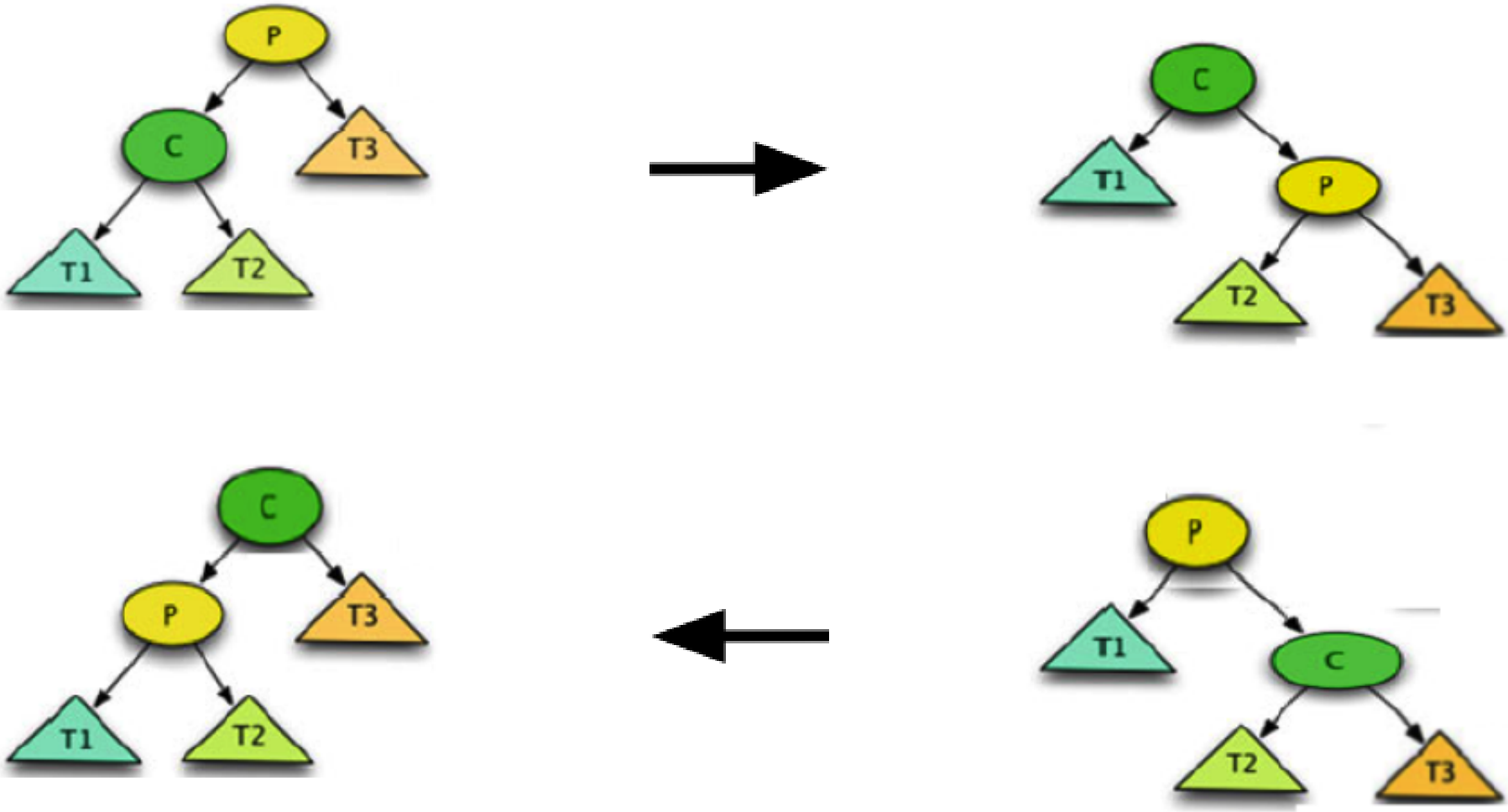
Left-Right Splaying Operation

The ***left-right splaying*** operation swaps a node C to the top and make its parent P and its grandparent G its left and right successors, respectively

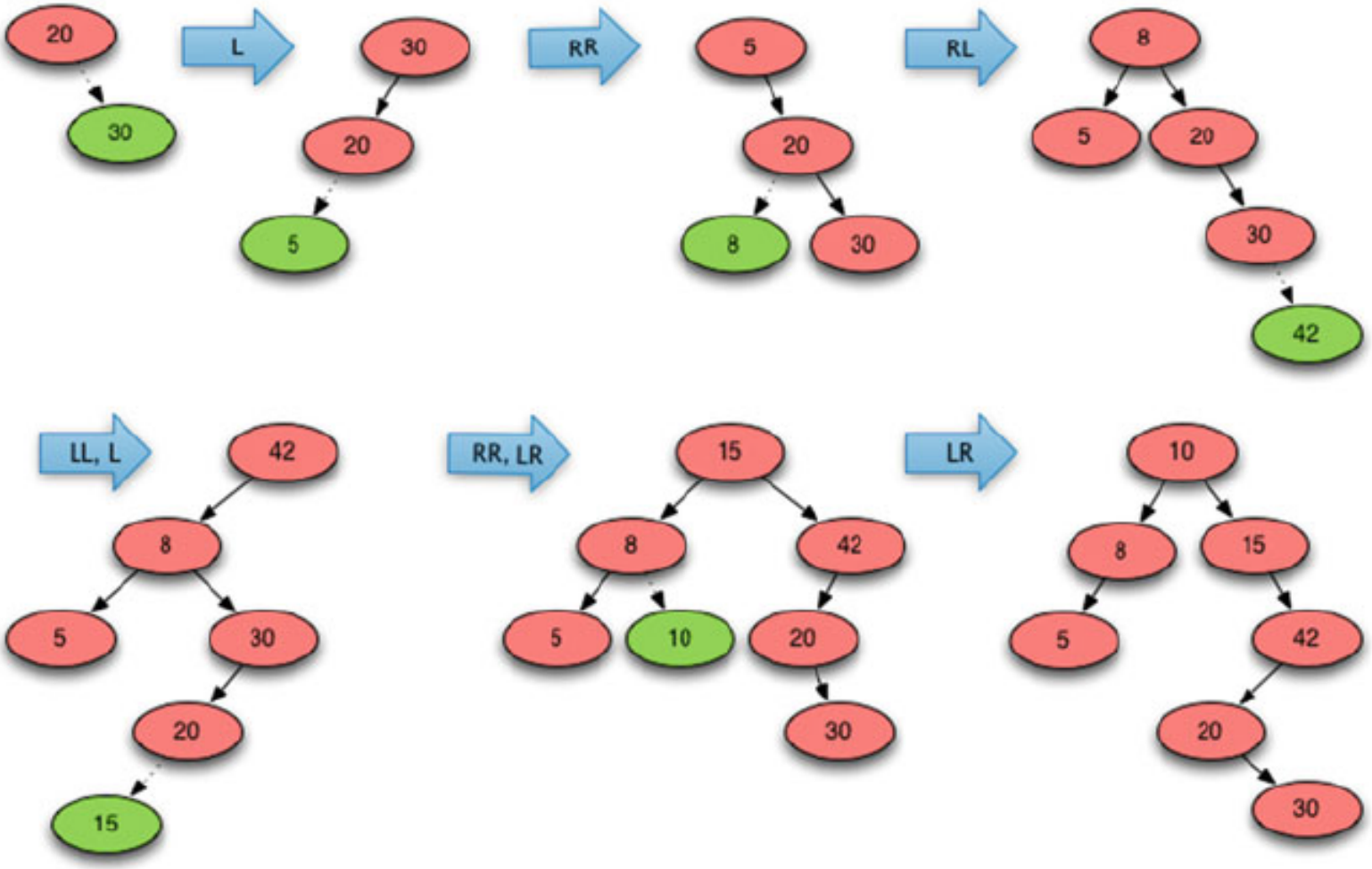


Other successor trees are shifted accordingly

Single Left and Right Splaying Rotation



Example


20
30
L
30
20
5
RR
5
20
8
30
RL
8
5
20
30
42
LL, L
42
8
5
30
20
15
RR, LR
15
8
42
5
10
20
30
LR
10
8
15
5
42
20
30

Spatial Locality

The example shows that **splay trees**, i.e. trees resulting from splaying operations that move a particular element to the root, are by no means balanced, so the worst case complexity for *insert*, *delete* and *find* will be in $O(n)$

However, splay trees are binary search trees, and so the average case complexity for *insert*, *delete* and *find* remains in $O(\log n)$

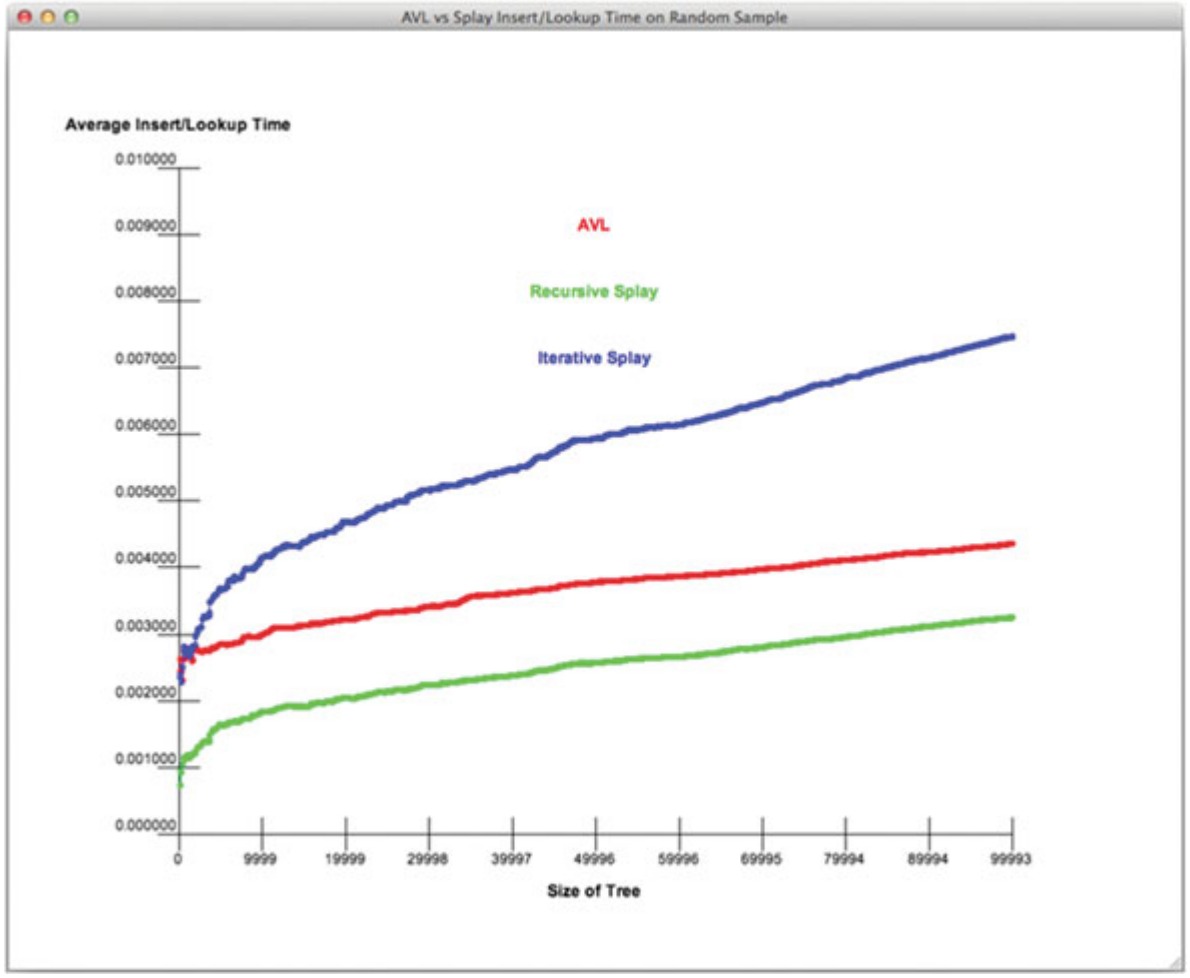
One can easily construct splay trees that are degenerate (same as for BSTs)

The main motivation for splaying is **spatial locality**: *Over a short period of time only a relatively small subset of data is accessed*

Informally, this implies that it is unlikely that very expensive operations occur often

Splay Trees in Practice

However, even for random sequences of operations (ignoring spatial locality) splay trees perform really well



Amortised Complexity of Splay Trees

How can the surprisingly good performance of splay trees be explained

This explanation is possible by using general **amortisation analysis**

Whenever an operation *op* transforms a search tree T into a new search tree T', the **amortised cost** is a sum

$$t + \Phi(T') - \Phi(T) \ ,$$

where t is the time required for the operation and $\Phi(T') - \Phi(T)$ is a contribution to other operations

In general amortisation analysis we make this contribution dependent on a function Φ

Amortisation Analysis

Now consider an arbitrary sequence of operations $op_1, \ldots, op_m$, where each op_i transforms a tree T_{i-1} into a tree T_i

So the total amortised cost of this sequence of operations is

$$\sum_{i=1}^{m} t_i + \sum_{i=1}^{m} (\Phi(T_i) - \Phi(T_{i-1})) = \sum_{i=1}^{m} t_i + \Phi(T_m) - \Phi(T_0)$$

If we assume to start with a trivial tree (empty or with just one node), we may assume $\Phi(T_0) = 0$

For a node v let $s(v)$ be the number of nodes in the subtree rooted at v

Furthermore, define $rank(v) = \log_2 s(v)$ and $\Phi(T) = \sum_{v \in T} rank(v)$

If we have reached a tree with n nodes, we clearly have $\Phi(T_m) \leq n \log_2 n$

Amortisation Analysis / 1

Now consider the three types of splay rotation operations

I. Simple Rotation. It suffices to consider a simple right rotation—left rotations are handled analogously

Let T and T' denote the trees on the left and right, respectively

Let r and r' denote the *rank* in T and T', respectively

We have

$$\Phi(T') = r'(C) + r'(P) + \sum_{z \in T_i} r'(z) \text{ and}$$

$$\Phi(T) = r(C) + r(P) + \sum_{z \in T_i} r(z)$$

Amortisation Analysis / 2

For nodes z in the subtrees T_1, T_2 and T_3 the size $s(z)$ does not change, neither does the *rank*

Therefore, we get

$$\Phi(T') - \Phi(T) = (r'(C) - r(C)) + (r'(P) - r(P)) \leq r'(C) - r(C) \ ,$$

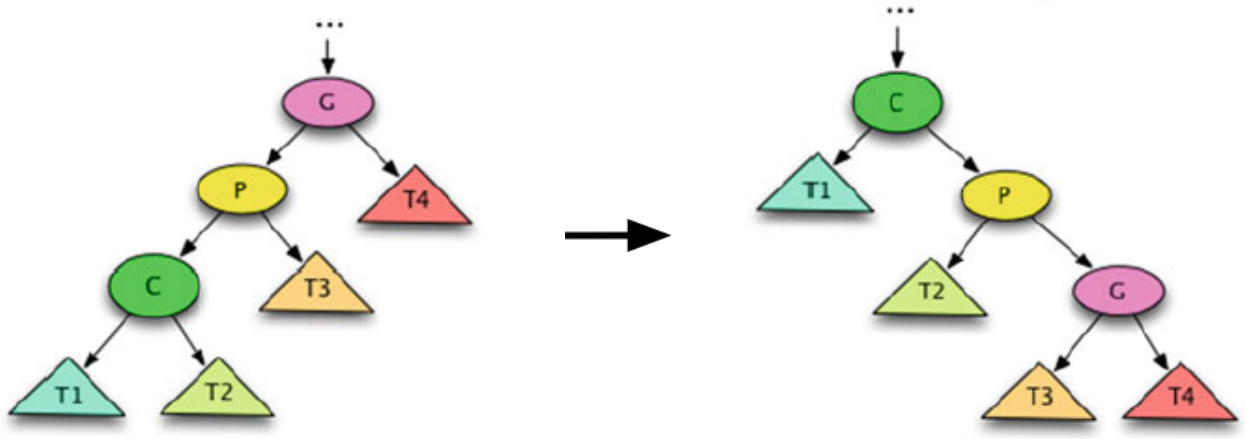
because $r'(P) \leq r(P)$

In addition, we can count 1 for the cost of swapping the nodes C and P, so in total the amortised cost of the operation is

$$1 + \Phi(T') - \Phi(T) \leq r'(C) - r(C) + 1 \leq 3(r'(C) - r(C)) + 1$$

Amortisation Analysis / 3

II. Left-Left or Right-Right Rotation. These are handled analogously



Here we get

$$\begin{aligned}\Phi(T') - \Phi(T) &= (r'(C) - r(C)) + (r'(P) - r(P)) + (r'(G) - r(G)) \\ &= (r'(P) - r(P)) + (r'(G) - r(C)) \leq r'(C) - 2r(C) + r'(G)\ ,\end{aligned}$$

because $r'(C) = r(G)$, $r'(P) \leq r'(C)$ and $r(C) \leq r(P)$

Amortisation Analysis / 4

Now consider

$$\begin{aligned} 2r'(C) - r(C) - r'(G) &= (r'(C) - r(C)) + (r'(C) - r'(G)) \\ &= \log_2 \frac{s'(C)}{s(C)} + \log_2 \frac{s'(C)}{s'(G)} \geq 2 \log_2 2 = 2 \ , \end{aligned}$$

because the sum of two logarithms is minimal, when they are equal, i.e. when

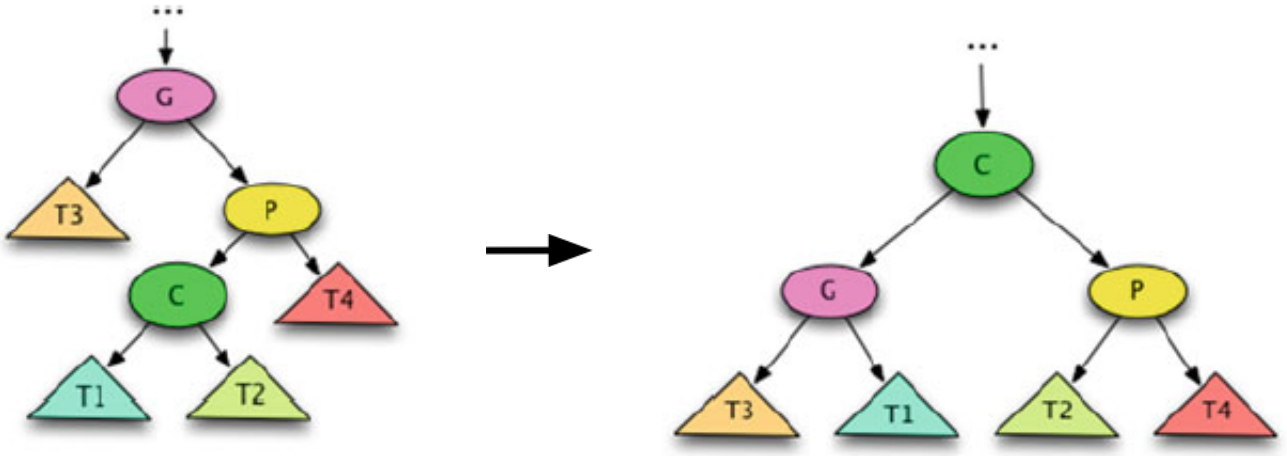
$$s(C) = s'(G) = \frac{s(C) + s'(G)}{2} \leq \frac{s'(C)}{2}$$

In this case count 2 for the cost of swapping the three nodes C, P and G, so in total the amortised cost of the operation is

$$\begin{aligned} 2 + \Phi(T') - \Phi(T) &\leq (r'(C) - 2r(C) + r'(G)) + (2r'(C) - r(C) - r'(G)) \\ &= 3(r'(C) - r(C)) \end{aligned}$$

Amortisation Analysis / 5

II. Left-Right or Right-Left Rotation. These are also handled analogously



Here we get

$$\begin{aligned}\Phi(T') - \Phi(T) &= (r'(C) - r(C)) + (r'(P) - r(P)) + (r'(G) - r(G)) \\ &= (r'(P) - r(P)) + (r'(G) - r(C)) \leq r'(P) - 2r(C) + r'(G)\ ,\end{aligned}$$

because $r'(C) = r(G)$ and $r(C) \leq r(P)$

Amortisation Analysis / 6

Analogous to the previous case consider

$$\begin{aligned} 2r'(C) - r'(P) - r'(G) &= (r'(C) - r'(P)) + (r'(C) - r'(G)) \\ &= \log_2 \frac{s'(C)}{s'(P)} + \log_2 \frac{s'(C)}{s'(G)} \geq 2 \log_2 2 = 2 \ , \end{aligned}$$

because the sum of two logarithms is minimal, when they are equal, i.e. when

$$s'(P) = s'(G) = \frac{s'(P) + s'(G)}{2} \leq \frac{s'(C)}{2}$$

Count again 2 for the cost of swapping the three nodes C, P and G, so in total the amortised cost of the operation is

$$\begin{aligned} 2 + \Phi(T') - \Phi(T) &\leq (r'(P) - 2r(C) + r'(G)) + (2r'(C) - r'(P) - r'(G)) \\ &= 2(r'(C) - r(C)) \leq 3(r'(C) - r(C)) \end{aligned}$$

Amortisation Analysis / 7

Bringing an element to the root requires a sequence of splaying rotation operations, at most one of which is a simple left or right rotation

Ther amortised cost of these operations is bounded by $3(r'(C) - r(C)) + 1$ for simple rotations and by $3(r'(C) - r(C))$ for the other rotations

Adding up these values is $\leq 3 \log_2 n + 1$, as only the rank of root is not cancelled and only once at $+1$ appears in the sum

Altogether we get $\sum_{i=1}^{n} t_i + \Phi(T_n) \in O(n \log n)$, so the amortised complexity of each single operation is in $O(\log n)$

This shows why the observed performance of splay trees is even better than the performance of AVL trees—the improvement comes from the simpler implementation

7.4 (a, b)-Trees and Red-Black Trees

Binary search trees are de facto data structures for ordered sequences

Different to sequence data structures that can be sorted in $O(n \log n)$ time, the AVL and splay trees maintain the property of being sorted when they are updated, and the complexity remains in $O(n \log n)$

Unbalanced BSTs are only of academic interest, as they cannot guarantee the $O(\log n)$ worst case complexity for *insert*, *find* and *delete*, and perfectly balanced search trees are not feasible

AVL trees solve this problem using "almost" balanced trees, and splay trees exploit similar reorganisation procedures to ensure an amortised complexity in $O(\log n)$

(a, b)-trees provide an alternative approach not controlling the balance of nodes, but to permit n-ary trees instead

Red-black trees are equivalent to $(2, 3)$-trees

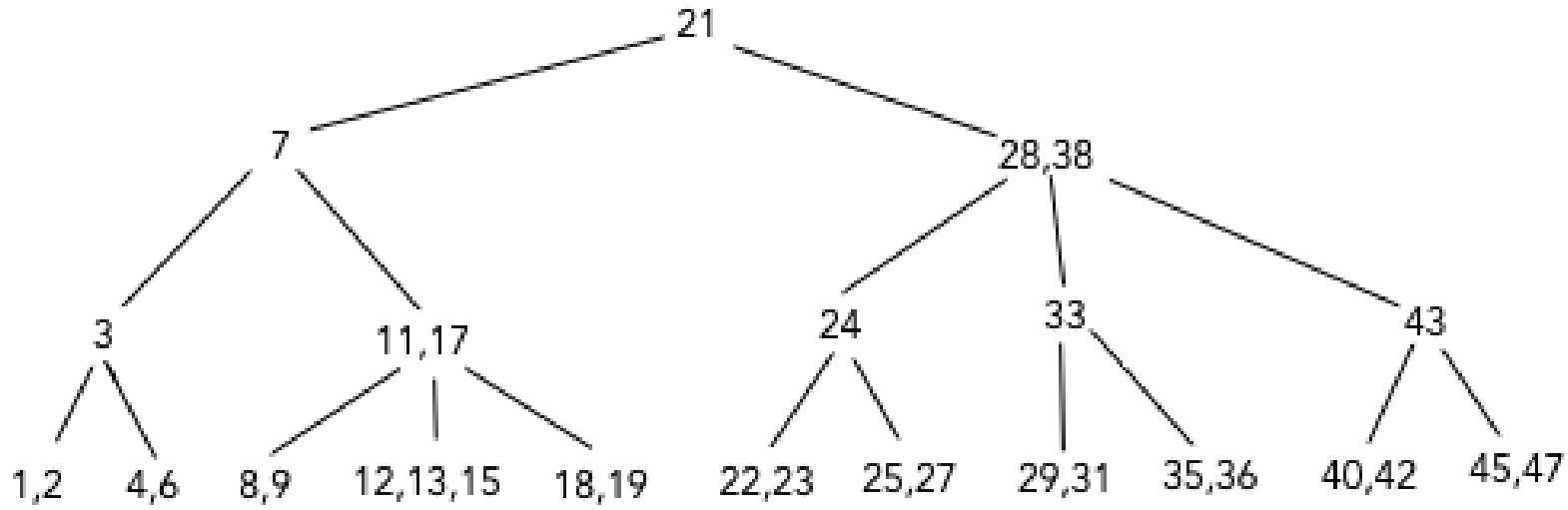
(a,b)-Trees

An **(a,b)-tree** with $a, b \in \mathbb{N}$, $0 < a \leq b$ is a tree with nodes labelled by several elements $e \in T$, where T is a totally ordered set satisfying the following conditions:

- every non-leaf node except the root has between a and b children
- if the number of nodes is > 2, the root has between 2 and b children
- each non-leaf node v with k children is labelled by $e_1 < \cdots < e_{k-1} \in T$ such that
  - all nodes in the i'th successor tree of v $(2 \leq i \leq k-1)$ contain only elements $e \in T$ with $e_{i-1} < e < e_i$
  - all nodes in the first successor tree of v contain only elements $e \in T$ with $e < e_1$
  - all nodes in the last successor tree of v contain only elements $e \in T$ with $e_k < e$
- each leaf node v is labelled by $e_1 < \cdots < e_k \in T$ with $a \leq k \leq b$

Example

The following tree is a $(2, 3)$-tree:



We see that it is perfectly balanced—search in the tree will be very efficient

Complexity of (a,b)-Trees

The key result on (a, b)-trees is the following:

Theorem. For $a \geq 2$ and $b \geq 2a - 1$ the operations *insert*, *find* and *delete* on (a, b)-trees with n elements can be done in time $O(\log n)$.

We will not conduct the detailed proof here, but only give a rough idea, why this result holds and how the operations can be implemented

For the proof as well as for the implementation it is decisive that for (a, b)-trees with $2 \leq a$ and $2a - 1 \leq b$ it is always to maintain the perfect balance of the tree—as we saw in our simple example

Then the height of the tree is bounded by $\log n$, which immediately implies the complexity result for *find*, which is implemented by a search from the root of the tree following the links to the appropriate successor tree (similar to BSTs and AVL trees)

Insertion into an (a,b)-Tree

The idea for insertion is also rather straightforward

First follow the links down to a leaf, where the new element is to be inserted

If the number of elements stored in a leaf exceeds b, so after the insertion we have at least $2a$ elements, then the node is split into two, and an appropriate value for distinguishing the elements in the new nodes is propagated to the parent node

It is also possible to consider sibling nodes, merge the nodes and split them in a more favourable way

This process of merging and splitting nodes is continued until the root is reached—in some cases it will become necessary to create a new root with two successor trees

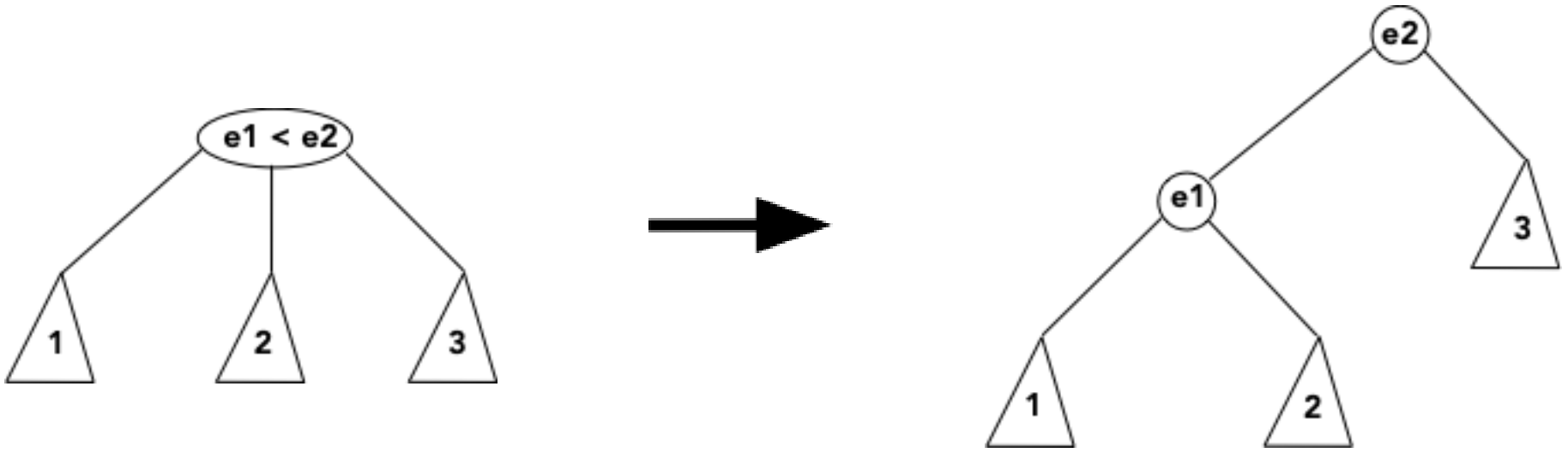
For the *delete* operation the process is analogous merging nodes, when the number of children falls below the lower bound a, and merge the root with its successors, if necessary

(2,3)-Trees and Red-Black Trees

Clearly it suffices to consider just $a = 2$ and $b = 3$, which define the smallest possible (a, b)-trees, those that are "almost binary"

For balanced $(2, 3)$-trees we find a different and more efficient way to implement them

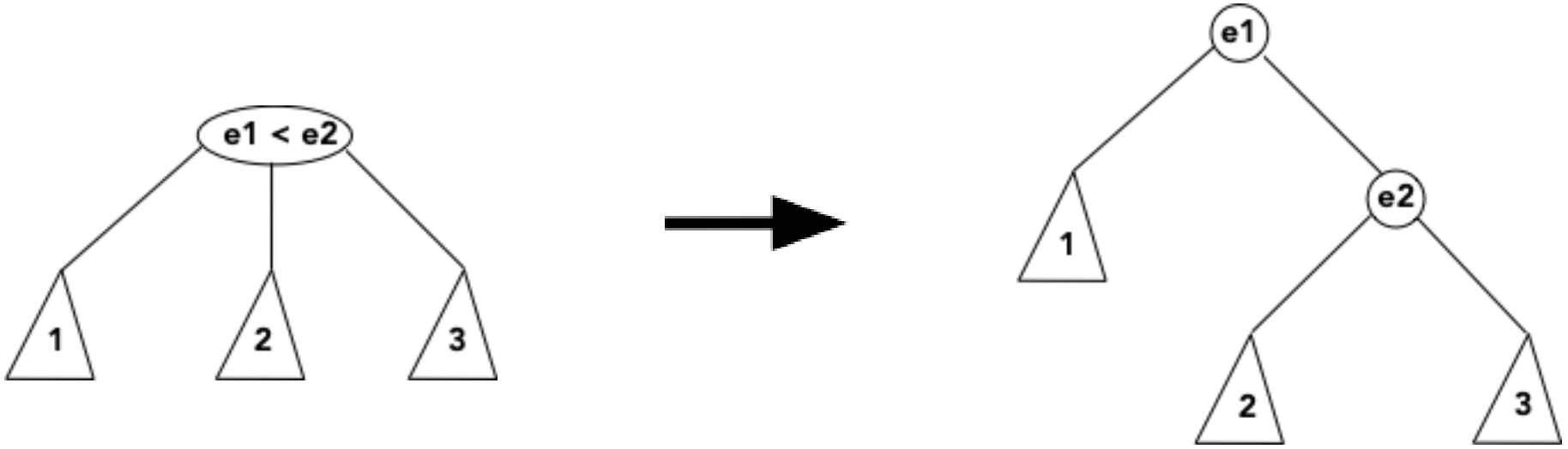
Whenever we have a node v labelled with $e_1 < e_2$, i.e. the number of children is 3, we can replace this node by two nodes, each with 2 children



The two nodes are labelled by e_1 and e_2, respectively

(2,3)-Trees and Red-Black Trees

It does not matter, which of the nodes labelled by e_1 and e_2 is the new parent; it only determines, where the successor trees are placed



Do the same for leafs with 2 or 3 elements—a leaf with 3 elements $e_1 < e_2 < e_3$ gives rise to a new non-leaf node labelled by e_2 with new leaves labelled by e_1 and e_3, respectively, as children

Whenever we make such a transformation, colour the new children as **red nodes**, while all other nodes remain **black**

Red-Black Trees

An ***red-black tree*** is a binary search tree with nodes coloured either *red* or *black* satisfying the following conditions:

- The root is black
- If a node is red, both its children are black
- In all subtrees rooted at some node v all simple paths from v to a leaf have the same number of black nodes

In the literature there are different definitions of red-black trees, where not the nodes, but the edges are coloured

If instead of colouring a node v red we colour the edge between the parent of v and v red, we obtain the edge-coloured version, and vice versa

Clearly, our transformation of $(2, 3)$-trees leads to red-black trees according to the definition above

Operations on Red-Black Trees

The *find* operation on red-black trees works in the same way as for BSTs or AVL trees

Using the last property in the definition of red-black trees we can easily prove that the height of a red-black tree with n elements is at most $2\log_2(n+1)$

Consequently, the complexity of *find* on red-black trees is again in $O(\log n)$

For *insert* and *delete* we employ again rotations, but in this case simple rotations suffice

These are defined in the same way as simple left or right splaying rotations

Insertion into a Red-Black Tree

Also with insertions we proceed analogously to insertions into AVL trees: we first locate a new leaf, where the new element has to be inserted, then propagate backards to the root

Colour the new leaf red—if the parent of the new leaf is black, nothing else needs to be done

Otherwise, as long we have two red nodes (child and parent) we distinguish three cases:

- If the grandparent node is black and has only red children, turn the grandparent red and its children black

- If the grandparent node is black and its other child (the uncle) is also black and both child and parent are in the same (left or right) relationship to their parent, perform a rotation with the grandparent and the parent and swap their colours

Operations on Red-Black Trees / cont.

- If the grandparent node is black and its other child (the uncle) is also black and child and parent are in the different (left or right) relationship to their parent, perform a rotation of child and parent followed by a rotation of child and grandparent together with a swap of colour

In each of these cases the last property of red-black trees is preserved, and with a sequence of such rotations we restore the properties of red-black trees

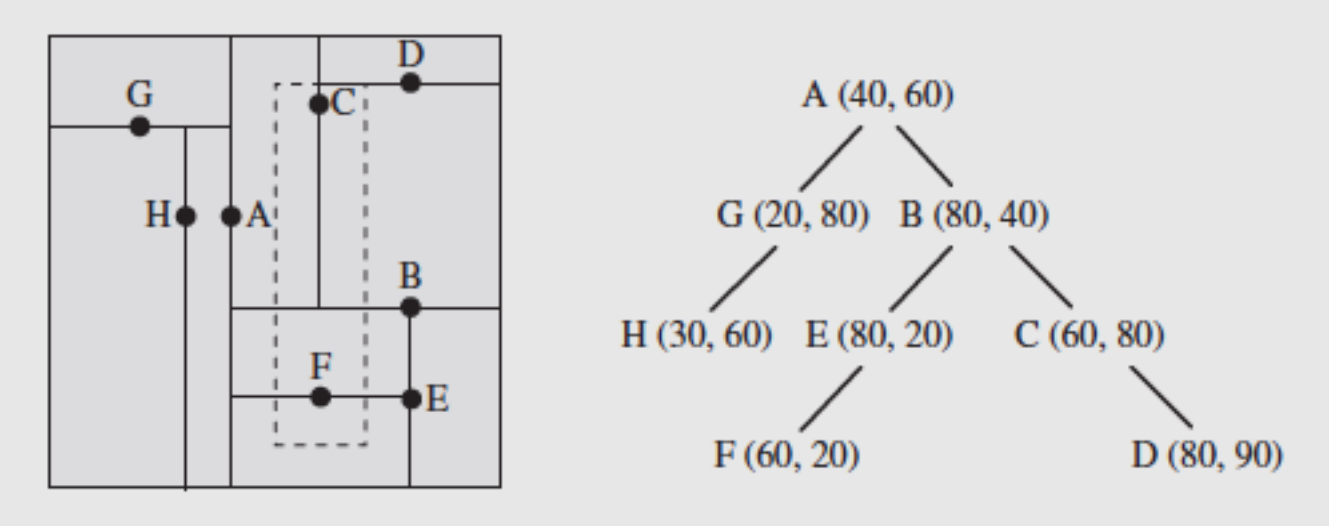
The fact that rotations require constant time implies that *insert* on red-black trees requires time in $O(\log n)$

For deletions we proceed analogously (as for splaying and AVL trees), which shows that also *delete* on red-black trees requires time in $O(\log n)$

7.5 Multi-Key Search with K-d Trees

With binary search trees (and all the optimisations: AVL trees, splay trees, red-black trees) we use a single key for navigation

However, it may be required to search in a multi-dimensional space, e.g. having points $(x, y) \in \mathbb{R}$ in the plane



The Idea of K-d Trees

In the previous example we use two-dimensional keys (x, y) in the binary search structure:

- On odd level $(1, 3, \dots)$ we use the x-component to navigate to the successor tree, i.e.
  - in the left successor tree all first components are smaller or equal than in the key value stored in the node
  - in the right successor tree all first components are greater or equal than in the key value stored in the node

The Idea of K-d Trees

In the previous example we use two-dimensional keys (x, y) in the binary search structure:

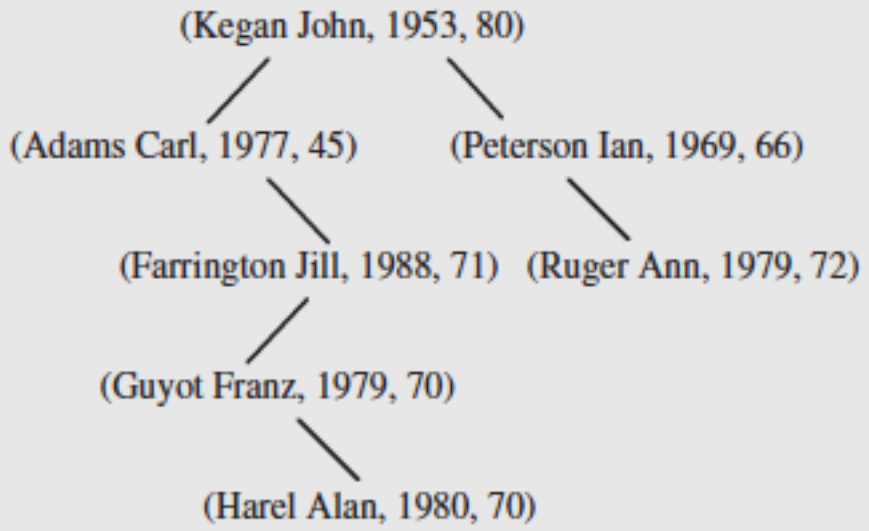
- On even level $(2, 4, \dots)$ we use the y-component to navigate to the successor tree, i.e.
  - in the left successor tree all second components are smaller or equal than in the key value stored in the node
  - in the right successor tree all second components are greater or equal than in the key value stored in the node

A k-dimensional tree (short: **k-d tree** generalises this idea for arbitrary k, not just 2

Example

The following tree on the right shows a 3-d tree representing the relation on the left

Name	YOB	Salary (K)
Kegan, John	1953	80
Adams, Carl	1977	45
Peterson, Ian	1969	66
Farrington, Jill	1988	71
Ruger, Ann	1979	72
Guyot, Franz	1979	70
Harel, Alan	1980	70



On level i we use the j'th component for navigation, where $j \equiv i \bmod k$

Operations on K-d Trees

Thus, operations *find* and *insert* on k-d trees are straightforward: just navigate from the root using first the first component, then the second, etc.

In case of *find* we proceed until we find the node, where the sought k- tuple is stored

In case of *insert* we proceed until we find a node, where the sought k- tuple is to be stored in the empty left or right successor tree, so we create a new leaf

Clearly, the time complexity for *find* and *insert* is in $O(n)$ as for BSTs

Optimisations similar to AVL trees, splay trees, (a, b)-trees or red-black trees are necessary for k-d trees as well to improve the complexity bounds

We dispense with further details

Deletion in K-d Trees

Nonetheless, let us look into the *delete* operation on k-d trees

Clearly we first have to find the k-tuple to be deleted, which is done analogously to the *find* operation

In a BST tree we could continue searching for the smallest successor in the (orphaned) right subtree or the smallest predecessor in the (orphaned) left successor tree and move this element to the node, where the delted element was stored

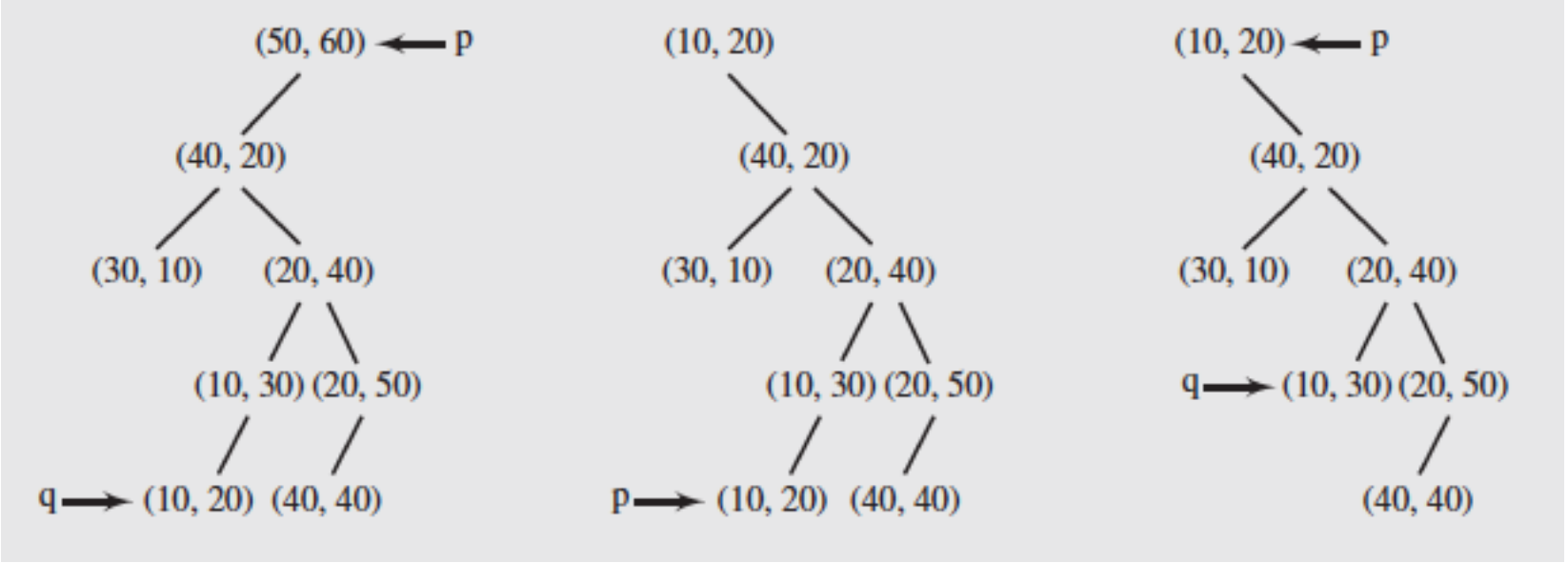
However, in a k-d tree such an approach cannot work

Nonetheless, we have to search either for direct predecessor or a direct successor, for which k levels of the tree have to be searched

We will only illustrate this procedure by an example

Example

In order to delete the root $(50, 60)$ find the direct predecessor in the left successor tree, so both the left successor trees of $(40, 20)$ and $(20, 40)$ have to be explored

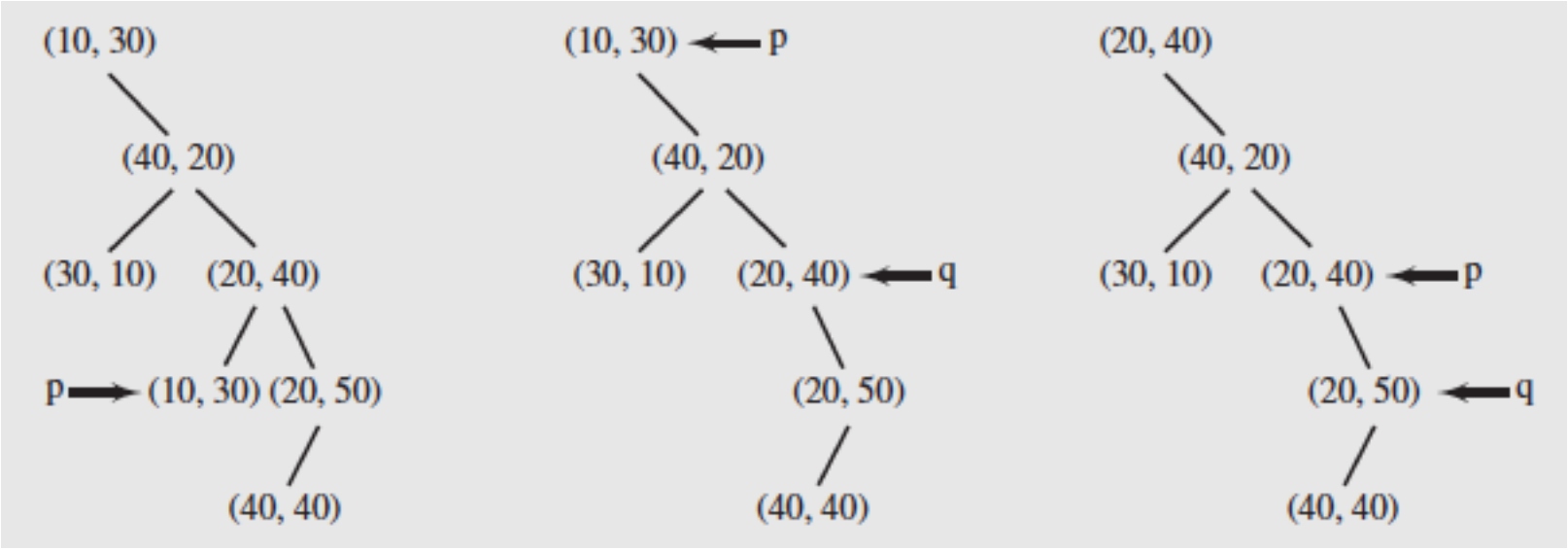


Moving the found predecessor to the root and deleting the corresponding leaf further requires $(40, 20)$ to become a right instead of a left successor

Deletion in K-d Trees / cont.

The next deletion looks for a direct successor of the $(10, 20)$ to be deleted, which is $(10, 30)$

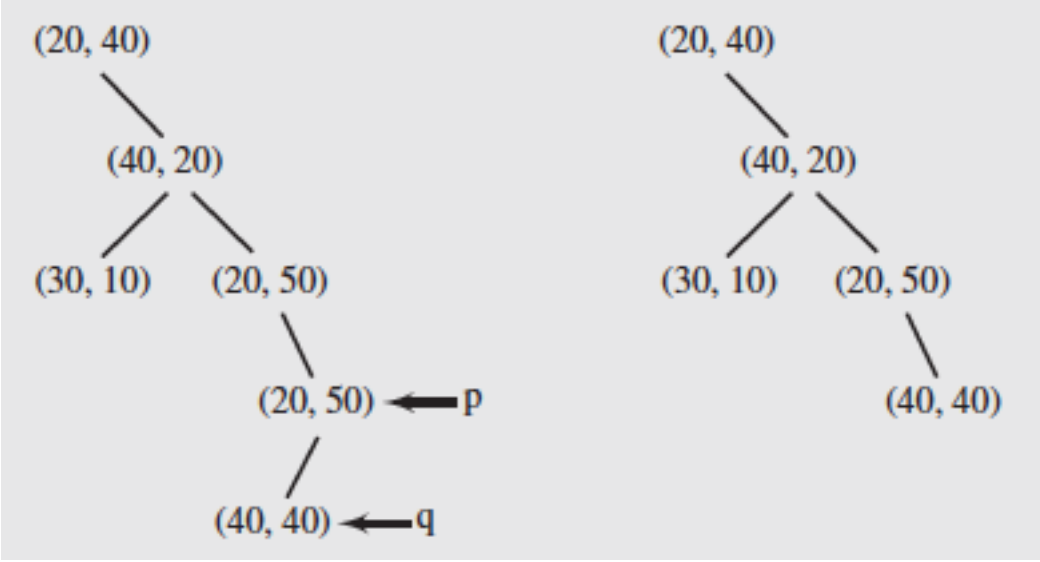
Again, it is moved to the root and the corresponding leaf is deleted



Deletion in K-d Trees / cont.

Finally, the deletion of $(10, 30)$ requires its direct successor $(20, 40)$ to become the new root

This triggers a cascade of shifts towards the root



7.6 Tries: Retrieval from a Set

The hashset data structure for sets based we looked at emphasised *insertion*, *deletion* and *retrieval*

However, we sometimes encounter a situation, where we have to deal with fixed sets, for which only the retrieval operation is important

Insertion is only needed to create the set, and deletion is not used at all

A common application example is a large dictionary used e.g. by a spell checker

For the representation of such sets it is more convenient not to rely on hashing, but to exploit other data structures

We will now look into such a data structure called **Trie** (derived from *re**trie**val*), which operates on similar ideas as (singly) linked lists

Preliminaries

An ***alphabet*** is a finite set, the elements of which are called ***symbols***

Let A be any alphabet and let A^* denote the set of all finite sequences of symbols in A (including the empty sequence)

A subset $L \subseteq A^*$ is called a ***language***

Given a finite set S we assume a one-to-one relationship between S and a language K, we will call the elements of K the **keys** of S

Tries are data structures storing the keys in K in a way that common prefixes are exploited

For this fix an alphabet A with $|A| = n$, let $K \subseteq A^*$ be a fixed language of keys, and let $\#$ be another symbol, $\# \notin A$

Full Tries

A ***full trie-node*** is a record $(\# : N_0, a_1 : N_1, \ldots, a_n : N_n)$, where $\{a_1, \ldots, a_n\} = A$, and each N_i is either a pointer to another full trie-node, a null pointer or a pointer to an element of K

A ***full trie*** is a $(n+1)$-ary tree, in which all non-leaf vertices are full trie-nodes, leaves are either null pointers or elements of K, and edges are defined by the pointers in the full trie-nodes

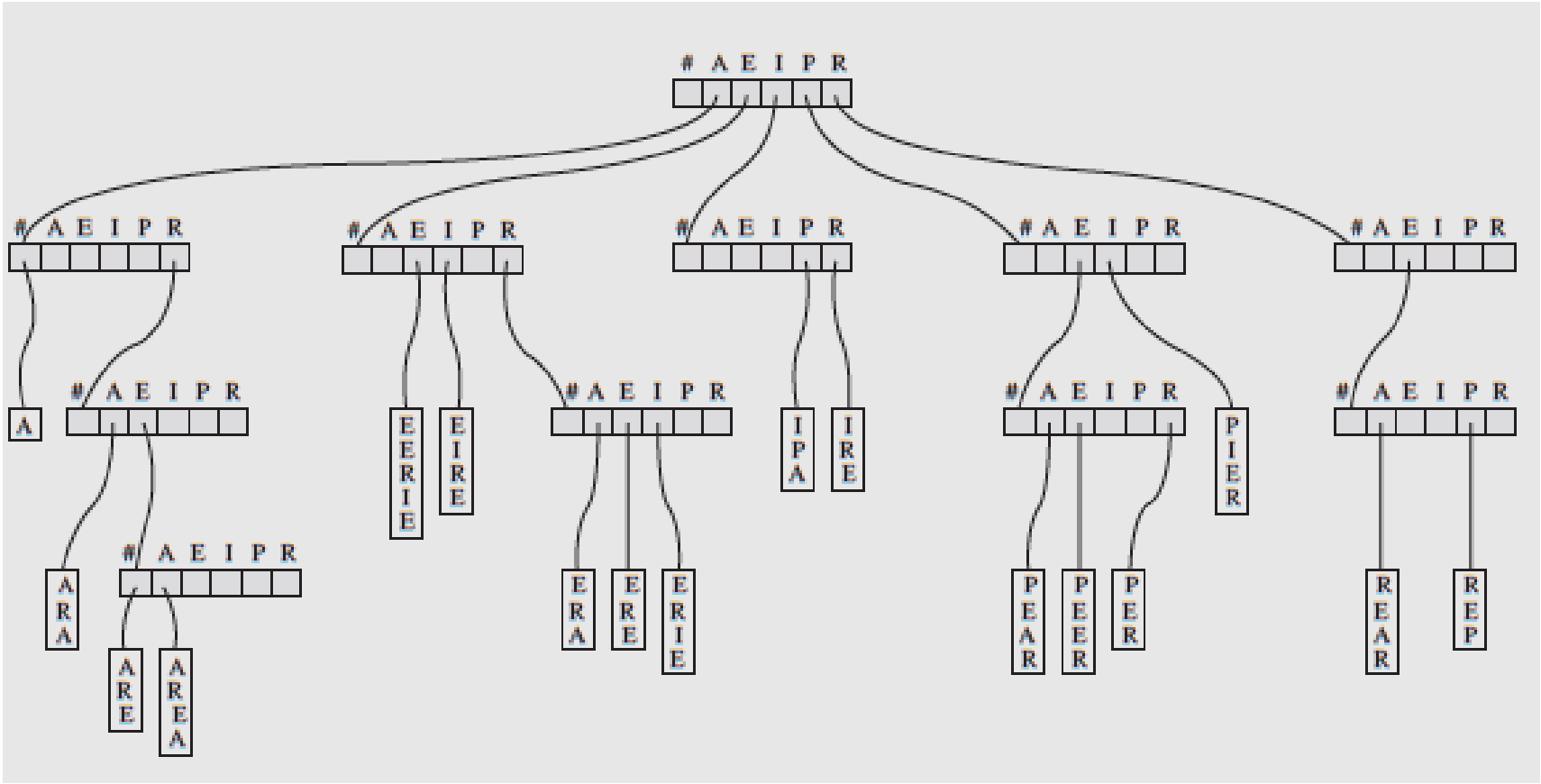
A path $v_0, \ldots, v_d$ from the root v_0 to a leaf $v_d \in K$ defines a sequence $a_{i_0}, \ldots, a_{i_{d-1}}$ of symbols in $(A \cup \{\#\})^*$, if the edge from v_j to v_{j+1} is defined by $a_{i_j} : N_{i_j}$ and N_{i_j} is not a null pointer

We say that a full trie ***represents*** the set K of keys iff

- for each sequence $w \in A^*$ defined by a path $v_0, \ldots, v_d$ from the root v_0 to a leaf $v_d \in K$ either w ends with $\#$, say $w = w'\#$, and $v_d = w'$ or w is a prefix of v_d
- each $w \in K$ appears exactly once as a leaf

Example: Full Trie

Let $K = \{A, ARA, ARE, AREA, EERIE, EIRE, ERA, ERE, ERIE, IPA, IRE, PEAR, PEER, PER, PIER, REAR, REP\}$



Reduced Tries

Clearly, the null pointers do not contribute to the representation of the key set K and thus can be omitted—all definitions are analogous leading to reduced tries representing K

A ***reduced trie-node*** is a record $(\ell_0 : N_0, \ldots, \ell_m : N_m)$, where $\{\ell_0, \ldots, \ell_m\} \subseteq A \cup \{\#\}$, and each N_i is either a pointer to another reduced trie-node or a pointer to an element of K

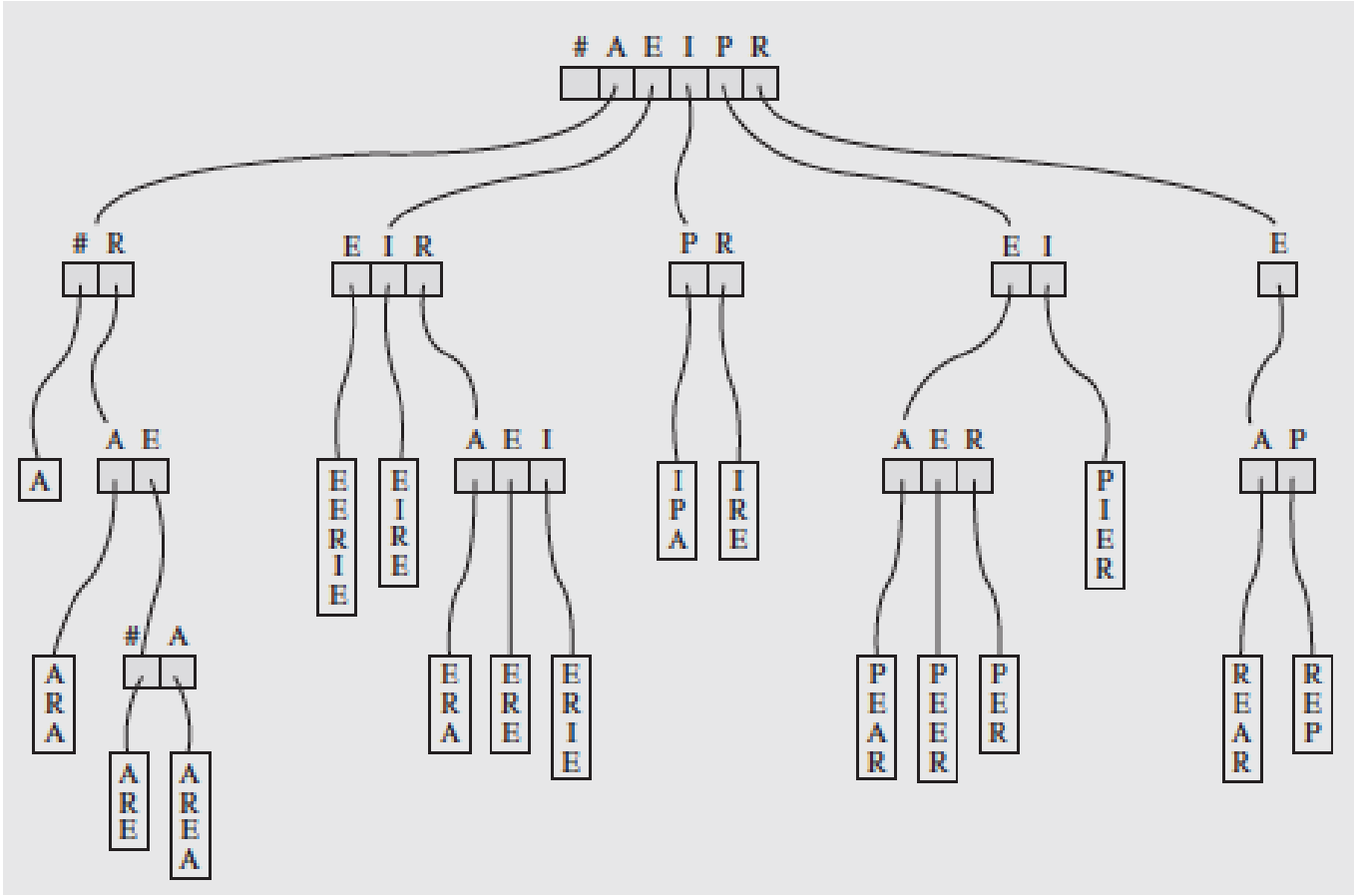
A ***reduced trie*** is a tree, in which all non-leaf vertices are reduced trie-nodes, leaves are elements of K, and edges are defined by the pointers in the reduced trie-nodes

A path $v_0, \ldots, v_d$ from the root v_0 to a leaf v_d defines a sequence $a_{i_0}, \ldots, a_{i_{d-1}}$ of symbols in $(A \cup \{\#\})^*$, if the edge from v_j to v_{j+1} is defined by $a_{i_j} : N_{i_j}$

The conditions for a reduced trie to ***represent*** the set K of keys are the same as for full tries

Example: Reduced Trie

Let $K = \{A, ARA, ARE, AREA, EERIE, EIRE, ERA, ERE, ERIE, IPA, IRE, PEAR, PEER, PER, PIER, REAR, REP\}$



Binary Tries

Instead of using records of varying length we can use nodes with a single symbol in $A \cup \{\#\}$ together with a *next* pointer to another node with an alternative symbol and a *follows* pointer to the next symbol in a sequence (as before)—this leads to a binary tree

A ***binary trie-node*** is a triple $(a, next, follows)$, where $a \in A \cup \{\#\}$, $next$ is a pointer to another binary trie-node or a null pointer, and $follows$ is either a pointer to another binary trie-node or a pointer to an element of K

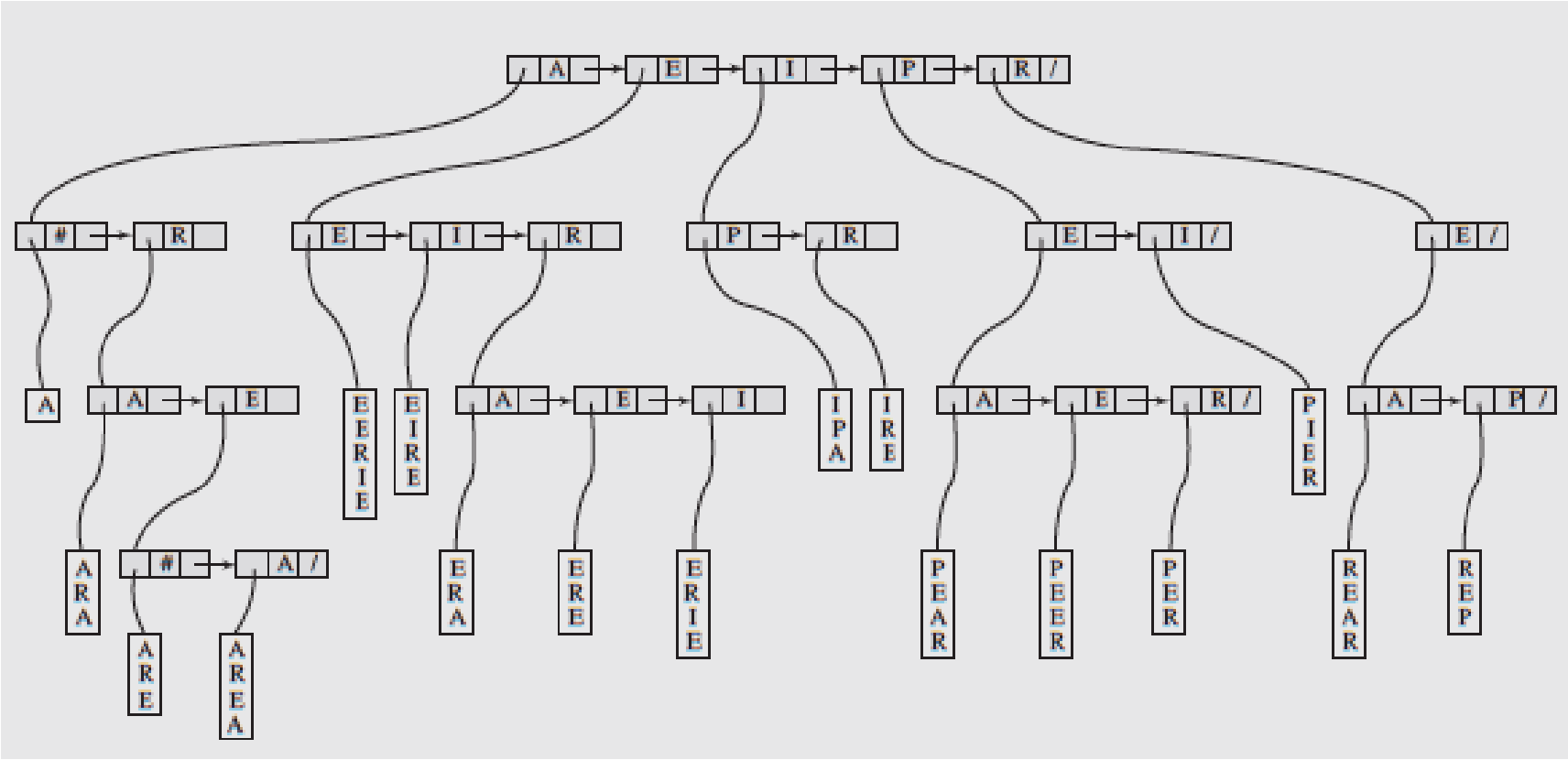
A ***binary trie*** is a binary tree, in which all non-leaf vertices are binary trie-nodes, leaves are elements of K or null pointers, and edges are defined by the pointers in the reduced trie-nodes

A path $v_0, \ldots, v_d$ from the root v_0 to a leaf v_d defines a sequence $s_0 \ldots s_{d-1}$, where s_j is the element $a_{i_j} \in A \cup \{\#\}$, if the edge from v_j to v_{j+1} is defined by $follows$, and s_j is the empty sequence otherwise

The conditions for a binary trie to ***represent*** the set K of keys are the same as for full tries

Example: Binary Trie

Let $K = \{A, ARA, ARE, AREA, EERIE, EIRE, ERA, ERE, ERIE, IPA, IRE, PEAR, PEER, PER, PIER, REAR, REP\}$



Trie Implementation

We concentrate on binary tries

Instead of leaves in K we only permit the special symbol #

Then # never appears in non-leaf nodes

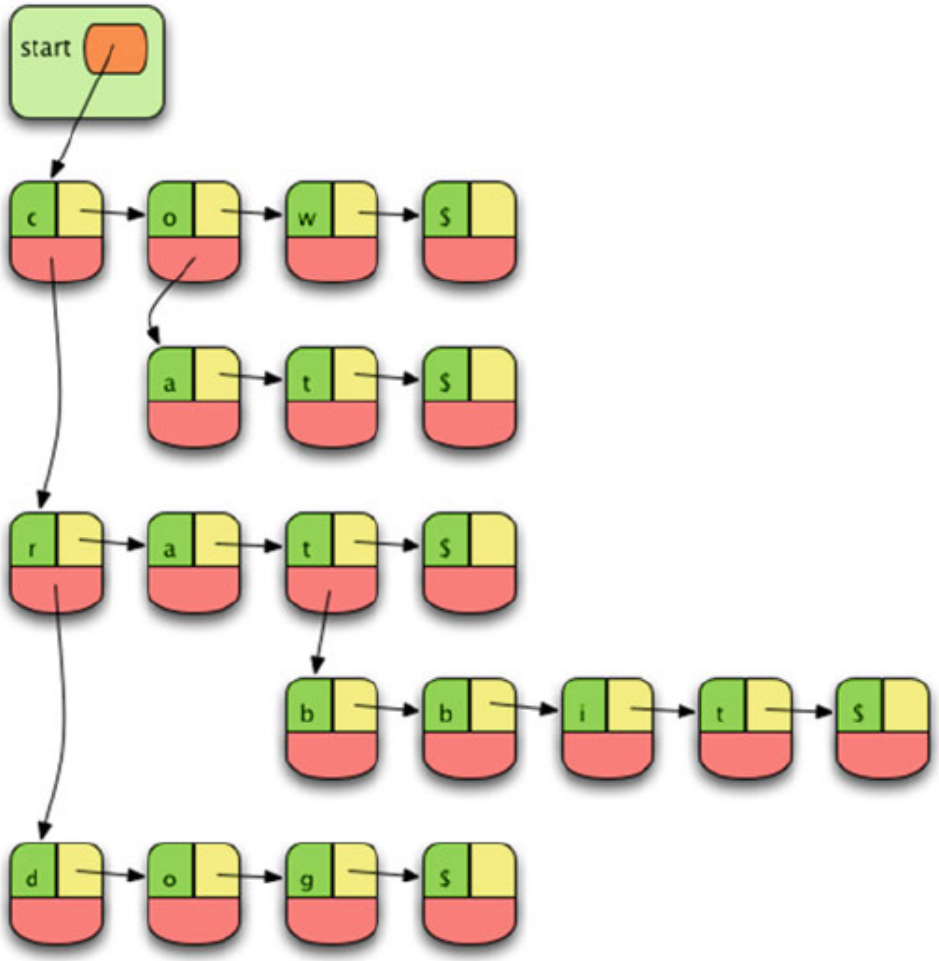
The structure is analogous to singly linked lists

Alternatively, mark final nodes and replace # by a null pointer

We will leave the concrete **C++** implementation for the discussion in the labs

Trie Implementation – Illustration

$K = \{cow, cat, rat, rabbit, dog\}$



Trie Implementation – Insert & Find

To *insert* a new sequence ℓ into a trie, follow the *follows* and *next* pointers to find the longest prefix of ℓ that already appears in the trie

Example. When inserting the sequence *carrot* of characters into the previous trie, we find the prefix *ca* in the trie—the *follows* pointer gives a *t*, and the *next point* is null

If ℓ is a prefix of an entry in the trie, simply create a new # node and let the *follows* pointer of the last node point to it—equivalently, mark this last node

If not, the missing postfix gives rise to new nodes link to each other by *follows* pointers, while a *next* pointer points to the first node of this postfix

Example. We create new nodes with values *r*, *r*, *o* and *t*, respectively, linked by *follows* pointers; the *a* node has a *next* pointer to the first *r* node, the final *t* node points to a # node (or is marked)

For the *find* operation we proceed analogously