Informatics II

Tutorial Session 11

Wednesday, 11th of May 2022

Discussion of Exercises 9 and 10,
Red-Black Trees, Hashing

14.00 – 15.45

BIN 0.B.06

Agenda

- Review of Exercise 9
  - Red-Black Trees
- Review of Exercise 10
  - Hashing

Universität Zürich[UZH]

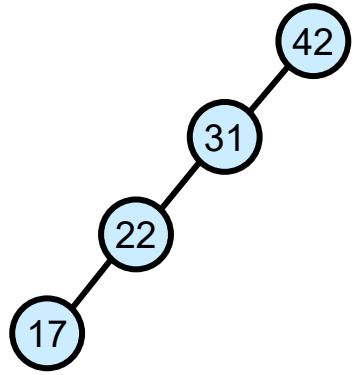
Institut für Informatik

Tree Balancing, Tree Rotations, Red-Black Trees

- Tree Balancing
- Tree Rotations
- Red-Black Trees

Balancing Mechanisms for Binary Search Trees: Motivation

- Binary search trees may degenerate, in particular if nodes are inserted in sorted order.
- In this case, the asymptotic complexity of operations is suboptimal.
- We need a balancing mechanism to prevent this from happening.
- Examples for trees with such a mechanism: AVL trees, red-black trees, 2-3 trees, 2-3-4 trees, splay trees, B-trees, B^+-trees, scapegoat trees, …



Red-Black Trees: General Remarks

- Red-black trees (RB trees) are a special kind of binary search tree.
- Every node is either red or black. (Typical implementation: Each node has an extra bit to save this.)
- All leaves are black and have no values attached to them; they are referred to as (black) «nil pointers» (this is also called the external node property).
- A balancing mechanism prevents the tree from degenerating.
  - Perfect balancing is neither aimed for nor (usually) achieved, though.
  - The longest path (root to deepest nil pointer) is no longer than twice the length of the shortest path (root to nearest nil pointer). The shortest path has all black nodes. The longest path exhibits an alternating sequence of red and black nodes.

Remarks:

- Searching in a red-black tree is exactly the same as in an elementary binary search tree (just ignore colour). Read-only operations are also identical (but they benefit from the fact that red-black trees are much better balanced).
- There is a 1:1 correspondence between red-black trees and 2-3 trees.

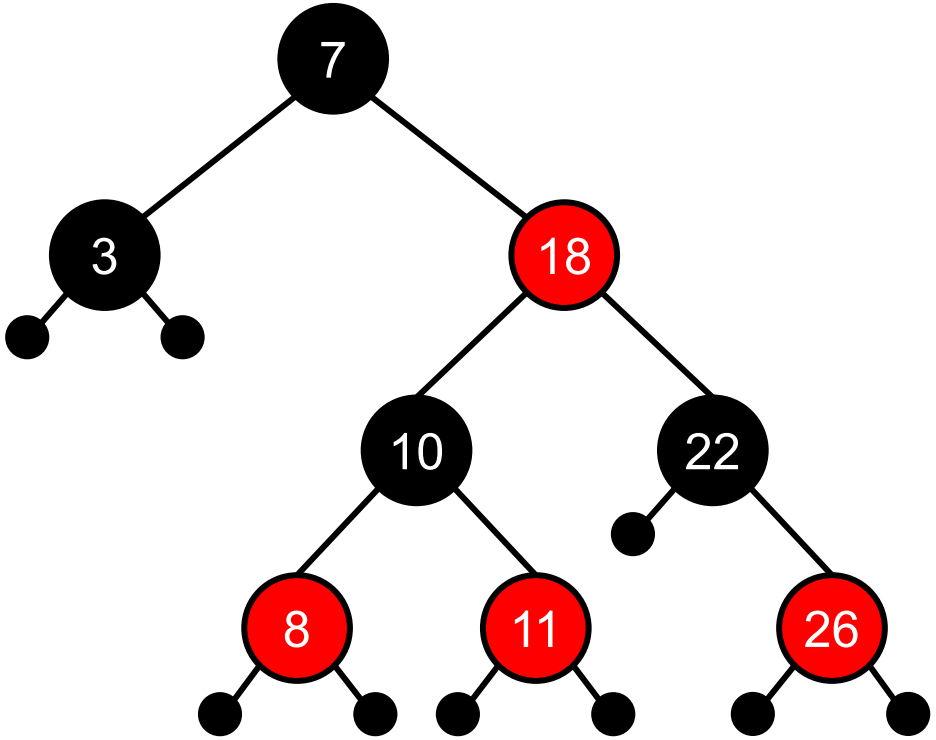
Red-Black Trees: Conditions

In a valid red-black tree, the following conditions must hold (additionally to BST, nodes red/black, external node property):

- **Root property**: The root node is always black.
- **Red property**: If a node is red, both its children are black. (Every red node has a black parent. On every path in the tree, there cannot be two red nodes consecutive.)
- **Depth property**: All leaves (black nil pointers) have the same black depth: For each node, all paths from the node to descendant leaves contain the same number of black nodes (count includes black nil pointers but not the root).

The latter two properties make sure that the tree remains balanced.

Red-Black Trees: Example



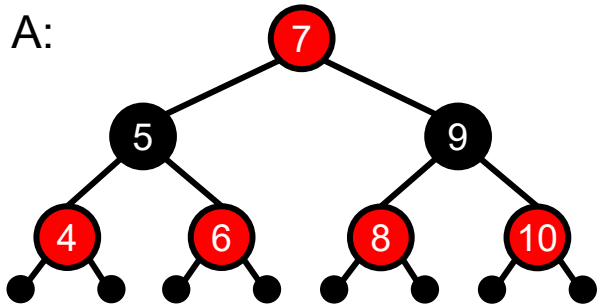
Check of RB conditions:

- It is a BST.
- All nodes are either red or black.
- All leaves are black and have no value (are black nil pointers).
- The root is black.
- The red node 18 has only black children, the same applies for 8, 11, 26.
- The black depth from 7 is always 2. The Black depth from 18 is always 2. The Black depth from all other nodes is always 1.

Red-Black Trees: Conditions Exercise

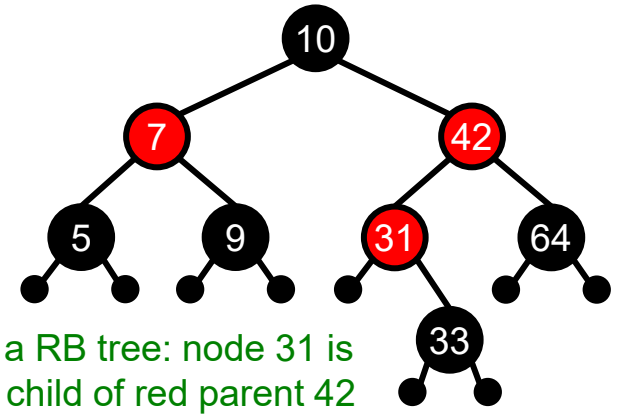
Are the following trees valid red-black trees?

A:



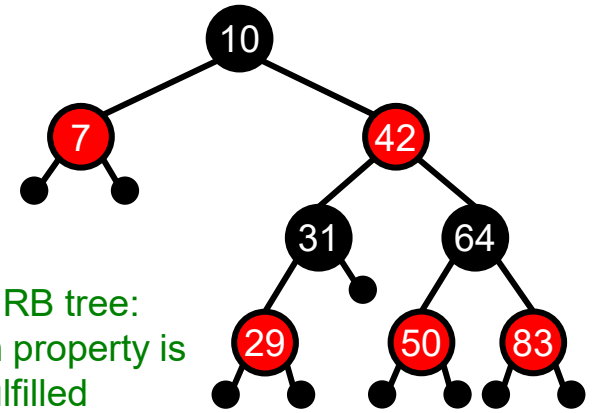
not a RB tree: root is red
(could be fixed by coloring root black)

B:



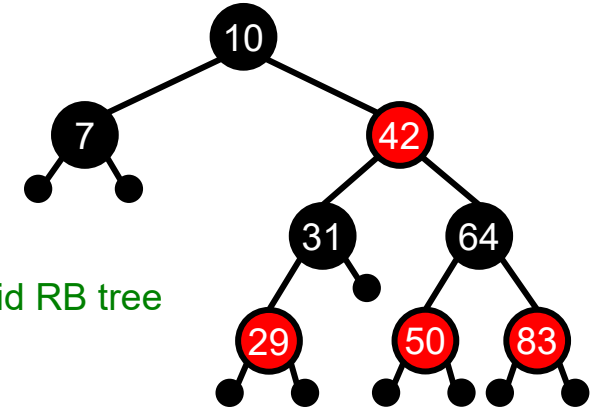
not a RB tree: node 31 is
red child of red parent 42

C:



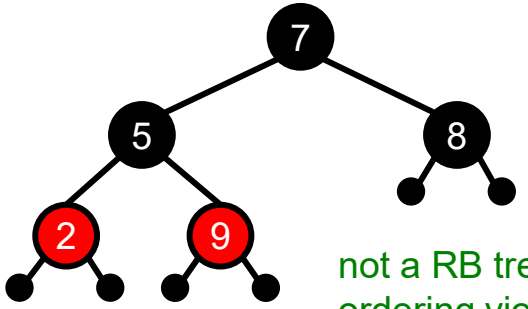
not a RB tree:
depth property is
not fulfilled

D:



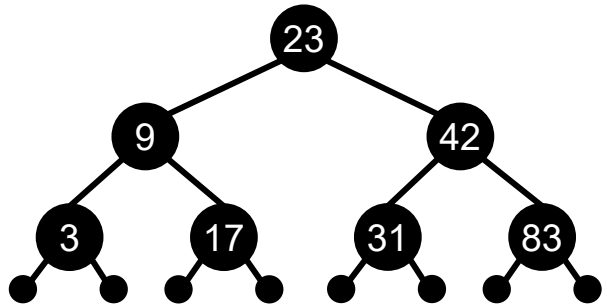
valid RB tree

E:



not a RB tree: BST
ordering violated by
node with value 9

F:



valid RB tree

Tree Rotations: Introduction

A tree rotation is a way of transforming an ordered tree (e.g. binary search tree) into another ordered tree, thus an operation preserving the sorted property. Thus, these operations will change the structure of a binary search tree without affecting the order of its nodes / its inorder sequence.

Rotations are used to rebalance an ordered tree (e.g. red-black trees, but not only them).

Tree Rotations: Introduction

There are two types of such rotations:

- left rotations (LR)
- right rotations (RR)

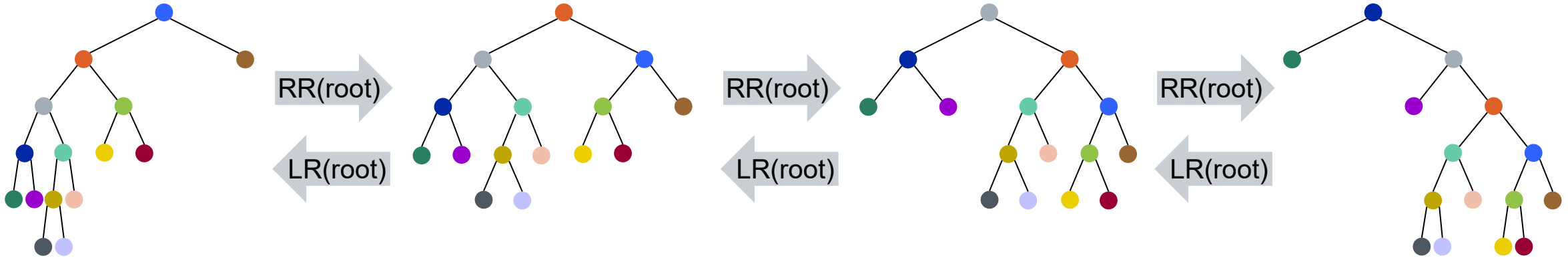
Rotations are applied on a certain node, i.e. LR(N). A left rotation on node N will shift nodes from the right subtree of N to the left subtree of N, thus.

Left rotations and right rotations will cancel out each other (they are inverse operations).

Tree Rotations: Effect Example

Example: Right rotations applied successively on the root of an initially left-heavy binary tree.

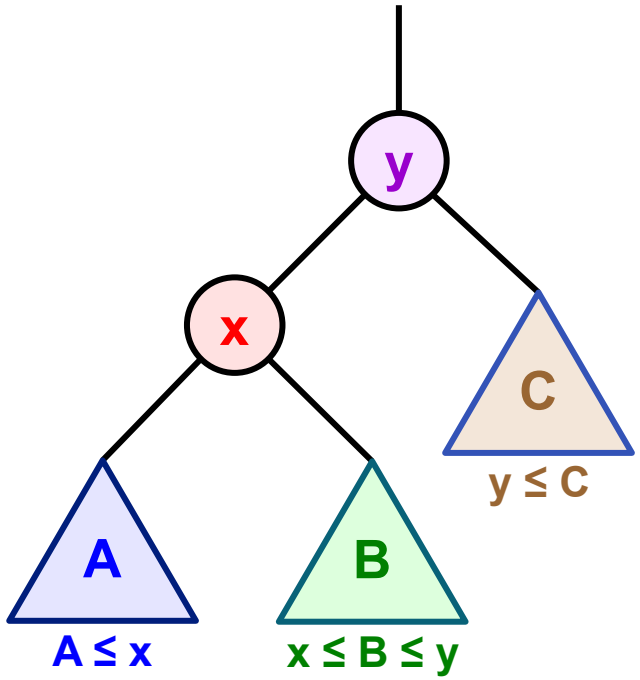
(Opposite direction: left rotations on the root of an initially right-heavy tree.)



Note that the root is changed by each tree rotation applied on the (current) root.

Tree Rotations: Preserving Sorted Property / Invariant

Consider the following binary search tree:



Note that the following conditions must hold (invariant of operation):

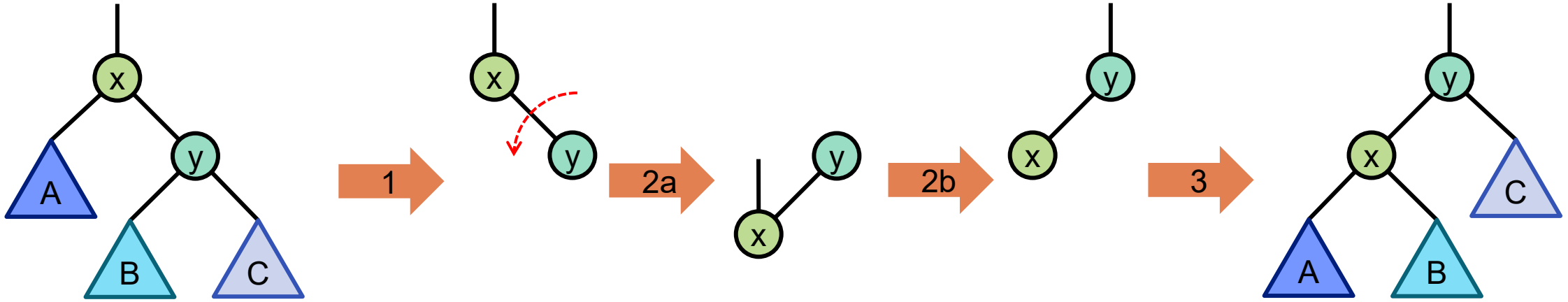
- all nodes in subtree A must be smaller than or equal to the value of x
- all nodes in subtree B must be bigger than or equal to x and smaller than or equal to y
- all nodes in subtree C must be bigger than y

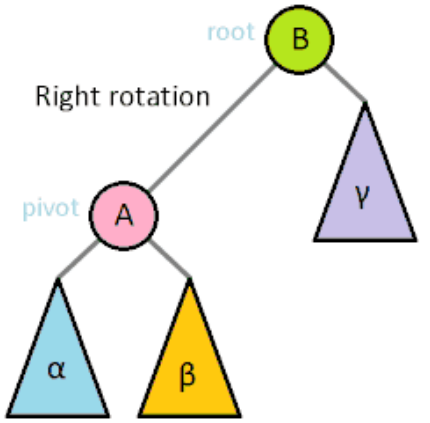
nodes in A ≤ x ≤ nodes in B ≤ y ≤ nodes in C

Tree Rotations: Left Rotation (LR)

1) From the node on which the rotation is performed (x), regard the *right* child (y). Cut off all subtrees (A, B, C) from the nodes involved (x, y).
2) a) Rotate the nodes x and y to the right around their common edge.
   b) Make the parent of x the parent of y. (This step can also be done in the end after step 3.)
3) Reattach the subtrees A, B, C according to the sorting condition as stated before:

   nodes in A ≤ x ≤ nodes in B ≤ y ≤ nodes in C

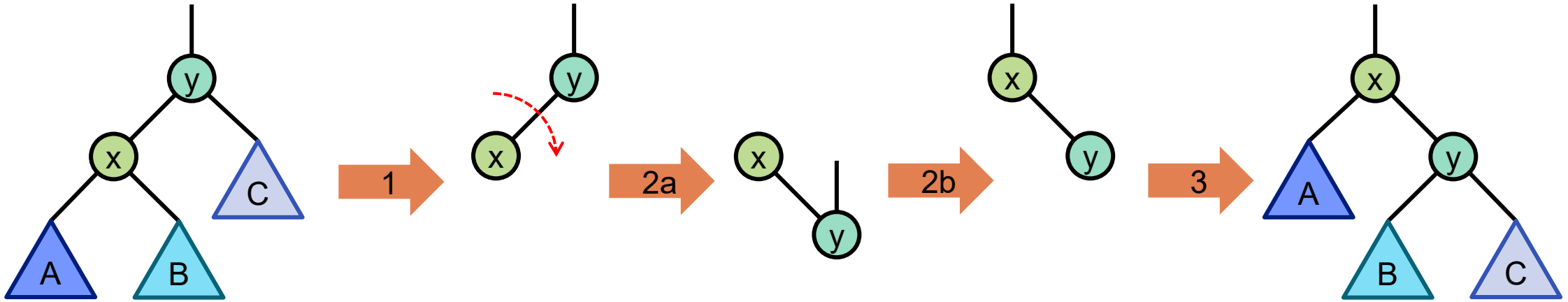




Tree Rotations: Right Rotation (RR)

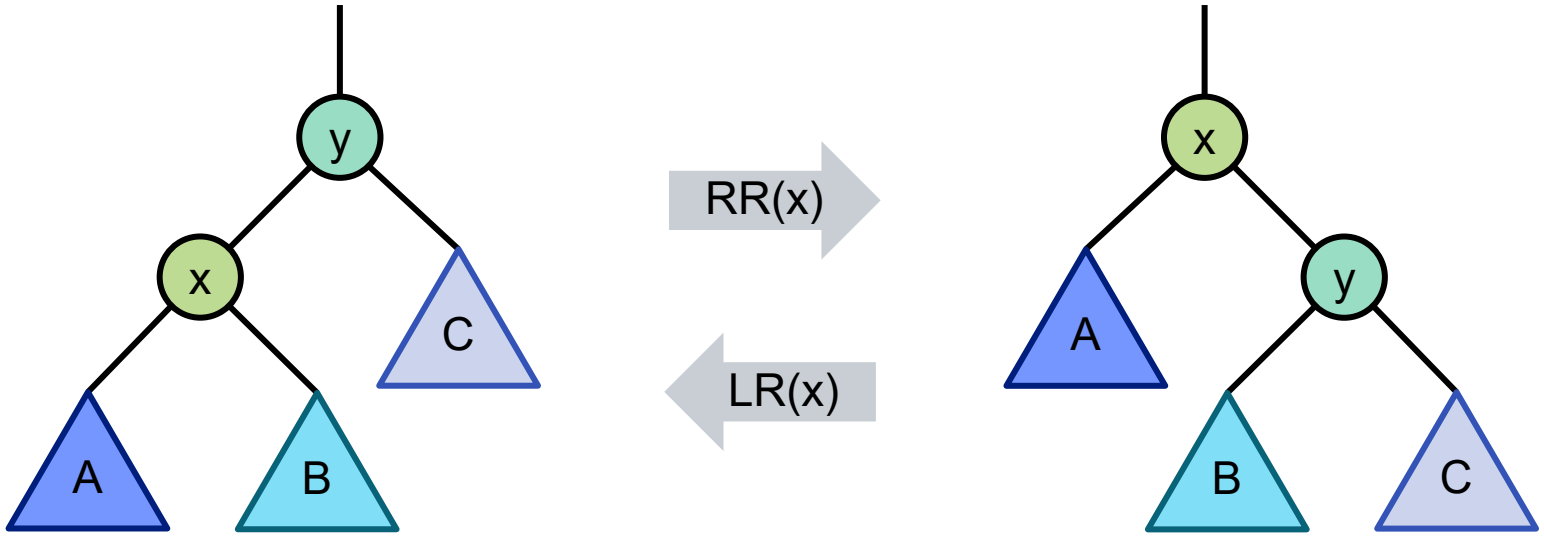
A right rotation is simply the opposite operation of a left rotation.

Note that in this case, the left child of the node on which the operation is performed is taken into consideration when determining the axis which is rotated.



Tree Rotations: Check Invariant

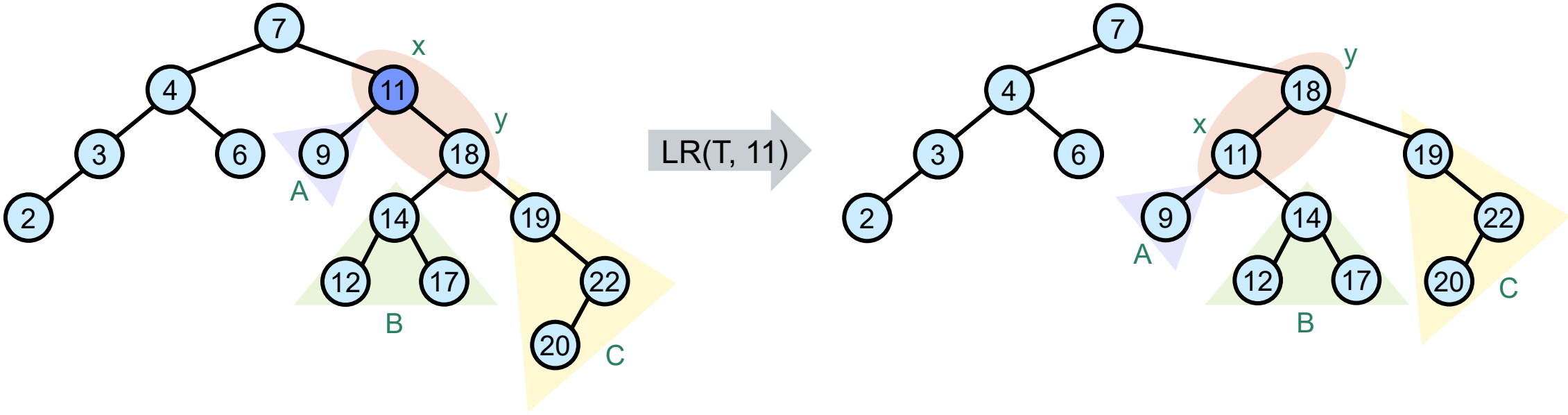
An easy way to see (and check) that tree rotations will preserve the sorted condition of a BST, is to look at the results of an inorder traversal of the tree before and after the rotation:



Inorder sequence:	A x B y C	A x B y C

Tree Rotations: Example / Exercise

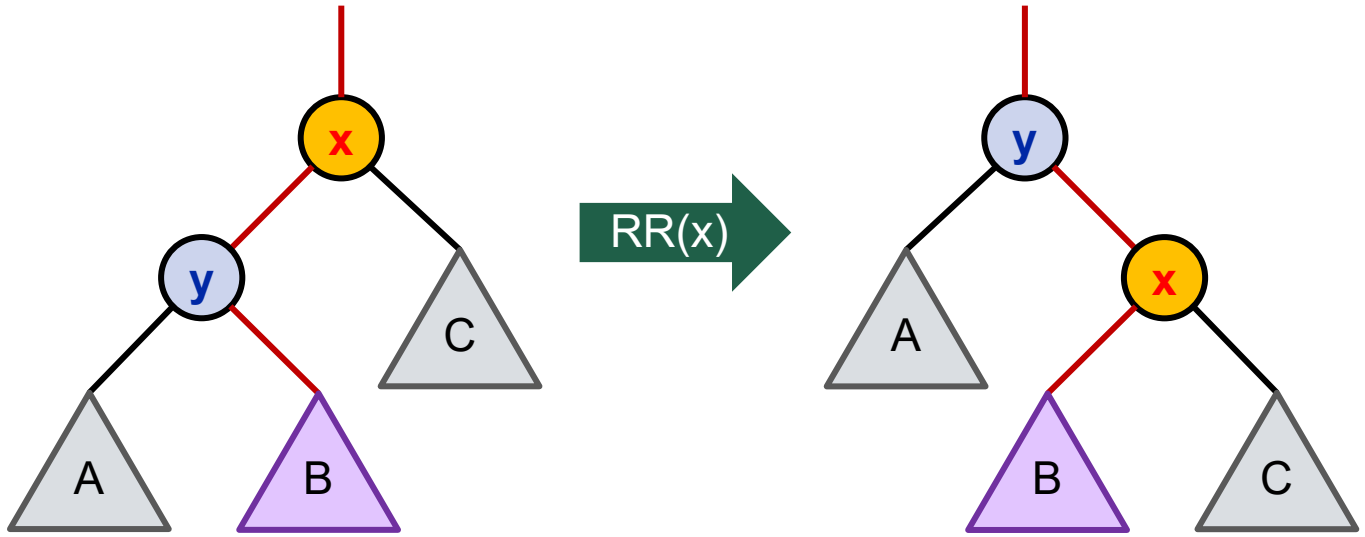
Consider the following binary search tree T. Perform a left rotation on the node with value 11.



Right Rotation Implementation – Without Parent Pointers

Pseudocode for right rotation:

```
rightRotate(x) {
        y = x->left;
        x->left = y->right;
        y->right = x;
}
```



Note that the subtrees A and C remain unchanged and attached to their previous parents.

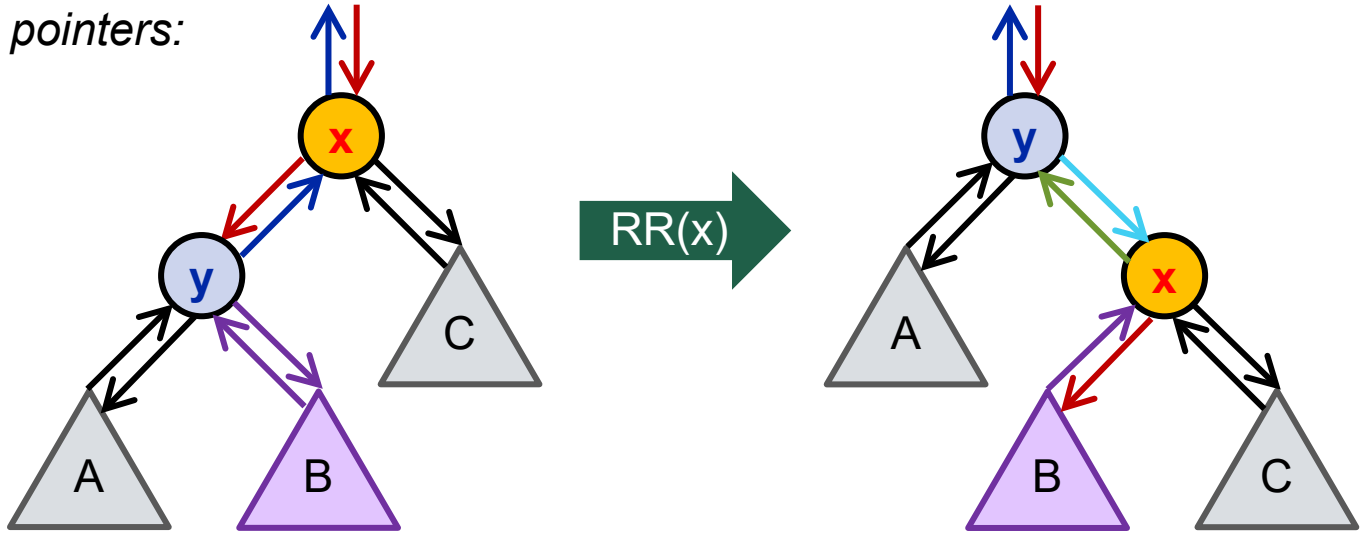
The above pseudocode only shows the basic idea and does not yet consider special cases. For example, if the node on which the rotation shall be performed (x) has no left child (i.e. y = x->left does not exist), the rotation has no effect / cannot be done.

Also note that the asymptotic time complexity of tree rotations is O(1).

Right Rotation Implementation – With Parent Pointers

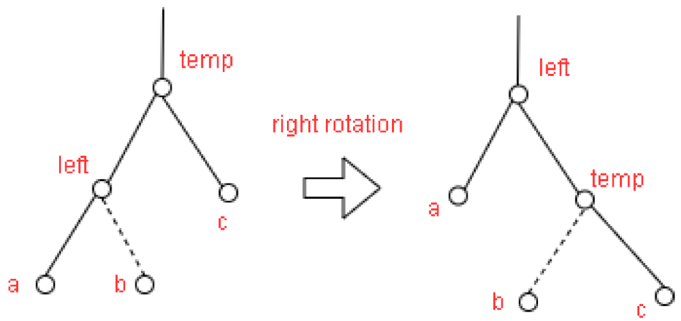
Pseudocode for right rotation with parent pointers:

```
rightRotate(T, x) {
        y = x->left;
        x->left = y->right;
        y->parent = x->parent;
        if (y->right != NIL) {
                y->right->parent = x;
        }
        if (x->parent == NIL) {
                T.root = y;
        } else {
                if (x == x->parent->right) {
                        x->parent->right = y;
                } else {
                        x->parent->left = y;
                }
        }
        y->right = x;
        x->parent = y;
}
```



Exercise 9, Task 4

1. Complete the code segment of `RightRotation(temp)` according to the diagram and instructions. Assume that the left child of node `temp` exists.




```
1  struct node {
2  struct node* p; // parent
3  struct node* r; // right child
4  struct node* l; // left child
5  };
6
7
8  void rightrotate(struct node* temp){
9      struct node * g = temp->p;
10     struct node* left = temp->l;
11     (a)_____ // 'b' becomes left child of 'temp'
12     if(left->r) {
13         (b)_____ // 'temp' becomes parent of 'b'
14     }
15     (c)_____ // 'left' become the child of g
16     if (temp == g->l){
17         g->l = left;
18     }
19     else {
20         g->r = left;
21     }
22     (d)_____ // 'temp' becomes right child of 'left'
23     (e)_____ // 'left' becomes parent of 'temp'
24 }
```

a. temp->l = left->r;
b. left->r->p = temp;
c. left->p = g;
d. left->r = temp;
e. temp->p = left;

2. State what lines 16-21 have done.

Since node `temp` is replaced by node `left`, these several lines make node `left` the right or left child as node `temp` was.

Red-Black Trees: Insertion

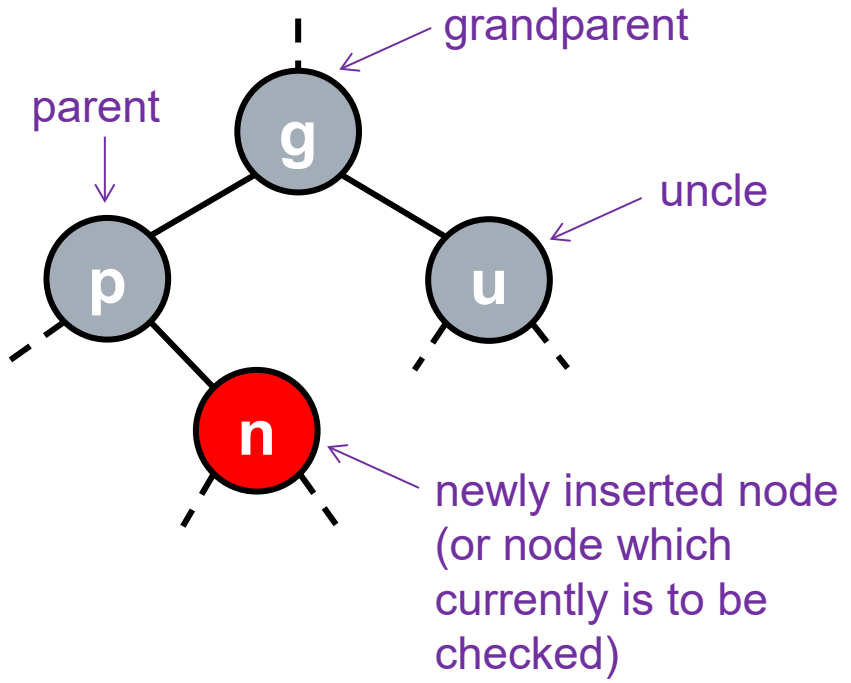
Insertion works the same way as inserting into a binary search tree at first (i.e. smaller values go to the left, bigger values go the right).

When a node is inserted, it is always colored red initially.

Next, the RB conditions are checked. If the tree which arose this way violates the RB conditions, one or both of the following operations is applied to restore the RB conditions:

- node recolorings
- tree rotations

Note that the new node can have children in general, because it is possible that changes propagate through the tree.
Also note that rotations do not change the black height of any node. (After a rotation, all subtrees are exactly the same relative to the root as they were before the rotation.)



Red-Black Trees: Insertion Cases

There are four different cases which can occur:

new node has black parent

- **Case 0:** Black parent: Parent of new node is black.

  → Conditions cannot be violated if the tree was a valid RB tree before; nothing to do.

new node has red parent

- **Case 1:** Red uncle: Red property violated by the new node and the uncle of the new node is red.

  → Recolor the parent and uncle in black, recolor the grandparent in red; propagate upwards through the tree (in the end change root to black if necessary)

- **Case 2:** Black uncle, triangle: Red property violated and the uncle of the new node is black; grandparent, parent and new node form a triangle.

  → Transform the tree to a case 3 configuration by rotating the parent and considering it as the new node.

- **Case 3:** Black uncle, line: Red property violated and the uncle of the new node is black; grandparent, parent and new node form a straight line.

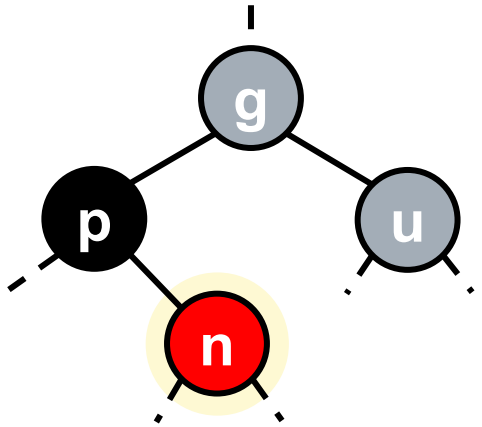
  → Recolor the parent black and the grandparent red, then rotate the grandparent.

Red-Black Trees: Insertion Cases: Remarks

- It could be argued that there is one additional case («case E»), namely the situation where the tree is empty before inserting the new node. In this case the new node can just be inserted as a black node as an exception (or it is insert as a red node as usual and a check is performed in the end, whether the root is black and recolor it if necessary since this is always allowed).
- Regarding cases 2 and 3, take in mind that the uncle can also be a nil pointer – which is a black node, too.
- The cases 2 and 3 each consist of two equivalent mirror cases.

Red-Black Trees: Insertion, Case 0 (Black Parent)

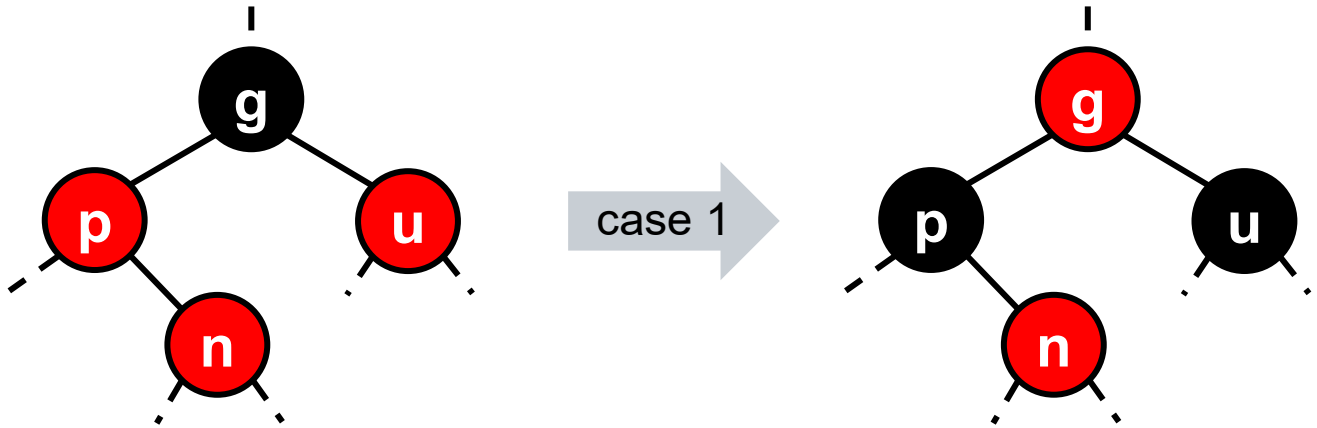
If the parent of the new node is black and the tree was a valid RB tree before insertion, no condition can be violated and there is nothing to do. (In this case, it doesn't matter what colors the grandparent and the uncle have and hence are shown in grey here.)



Red-Black Trees: Insertion, Case 1 (Red Uncle)

If the parent of the newly inserted node is red, there is a violation of the red property (two consecutive red nodes). If, furthermore, the uncle of the new node is red, the tree can be restored to a valid RB tree by a set of recolorings as follows:

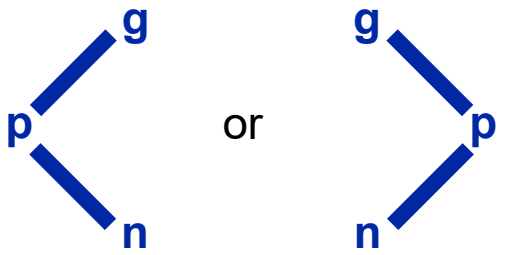
1) Recolor parent and uncle in black.
2) Recolor the grandparent in red.
3) If the grandparent's parent is red, there could be another violation of the red property. Therefore, this recoloring scheme could propagate upwards in the tree (while treating the grandparent as the newly inserted node, possibly until reaching the root).



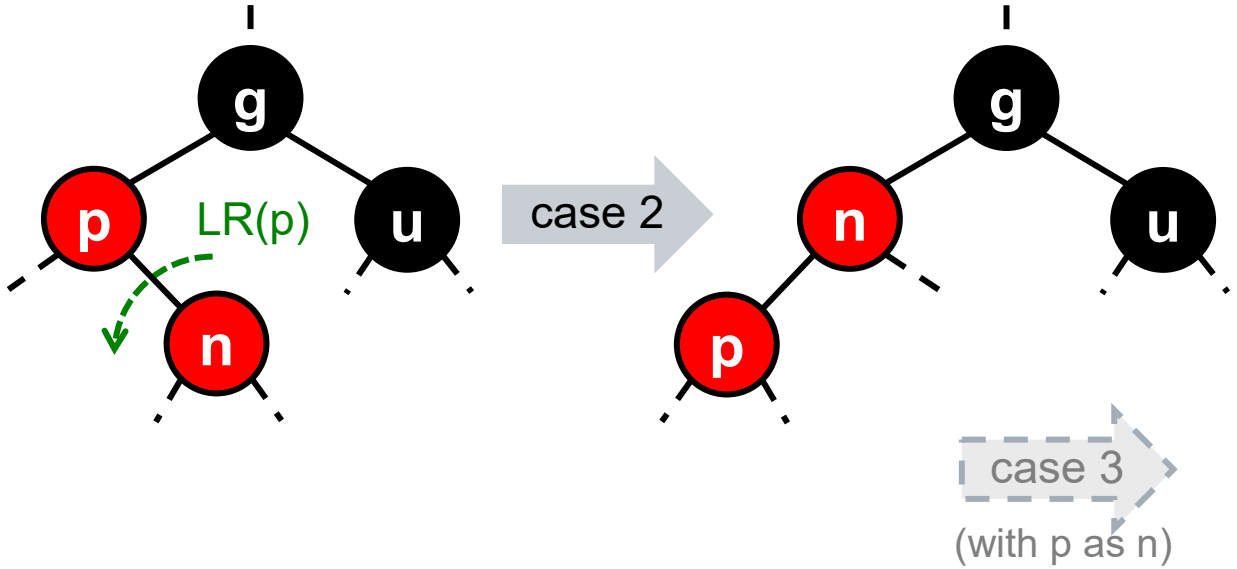
Red-Black Trees: Insertion, Case 2 (Black Uncle): Triangle Case

In this situation, the parent of the newly inserted node is red, but in contrast to case 1, the uncle is black.

Furthermore, the grandparent, the parent and the uncle form a triangle shape.

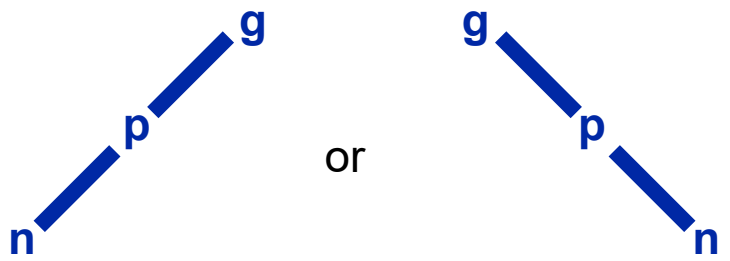


1) Apply a rotation on the parent of the new node. A left rotation is applied when the triangle points to the left and a right rotation is applied when the triangle points to the right (i.e. the rotation should get grandparent, parent and new node into a line configuration).
2) Consider the former parent as the newly inserted node. (Do *not* swap them, but just shift the status of "new" to the former parent.)
3) The tree has now been transformed into a case 3 configuration (line case), thus continue with the operations of case 3 from here.

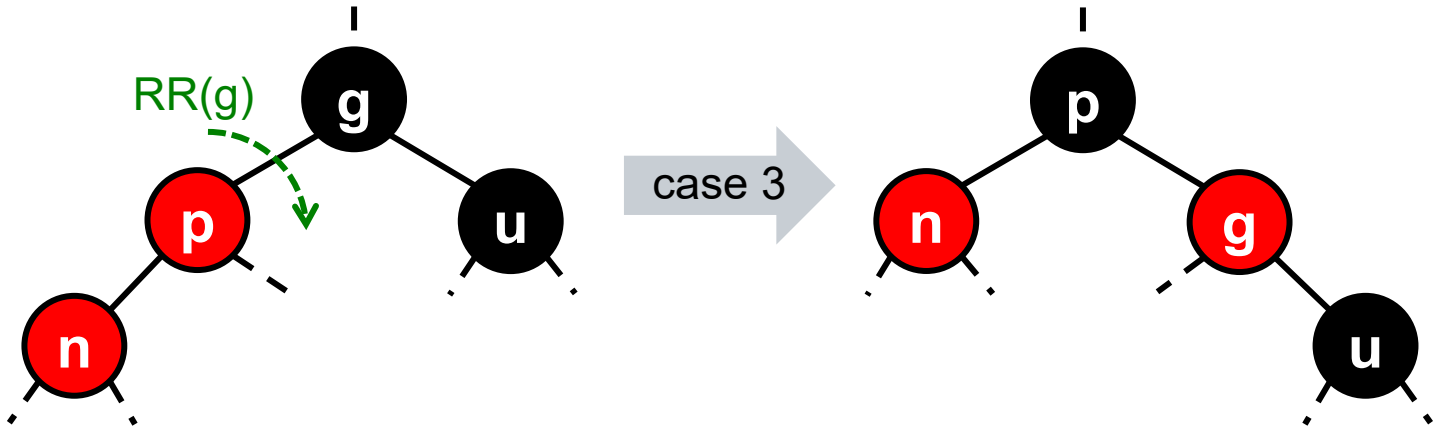


Red-Black Trees: Insertion, Case 3 (Black Uncle): Line Case

Again, the parent of the newly inserted node is red and its uncle is black. Though, in this case, the new node, the parent and the grandparent form a straight line. This configuration can be the result of a previous case 2 transformation.

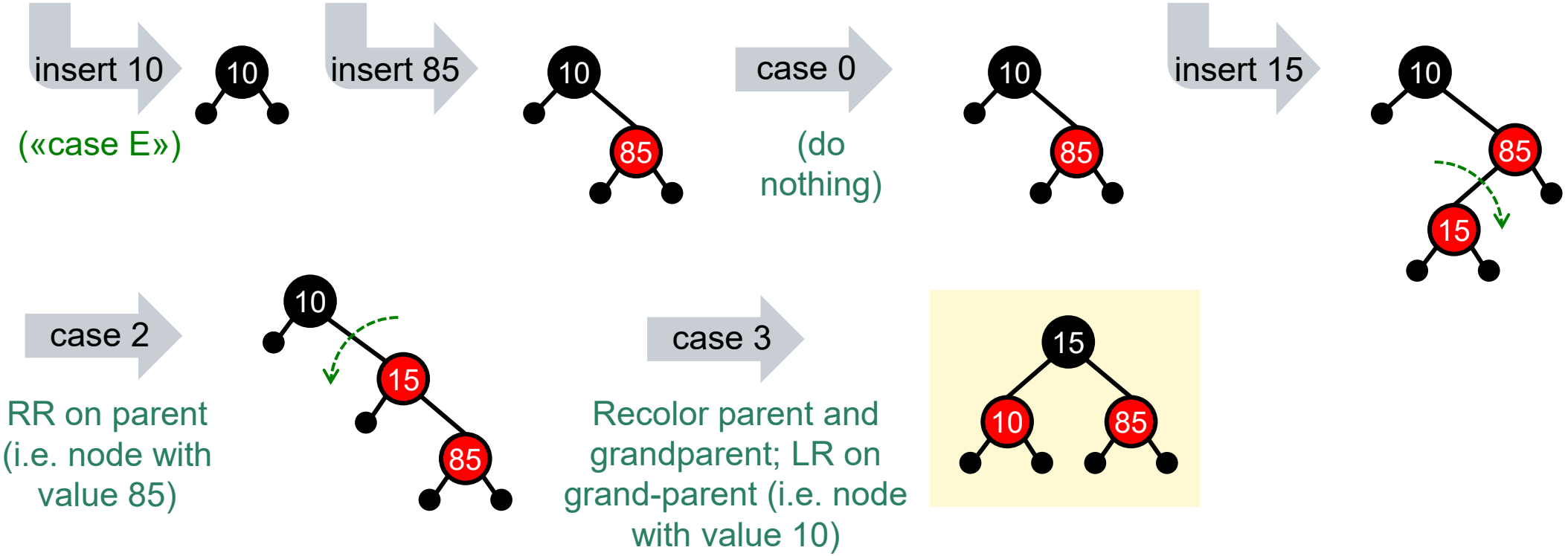


1) Recolor the parent black and recolor the grandparent red.
2) Apply a rotation on the grandparent of the new node towards the opposite site the tree is currently leaning. A right rotation is applied if the new node is a left child and a left rotation is applied when the new node is a right child of its parent.

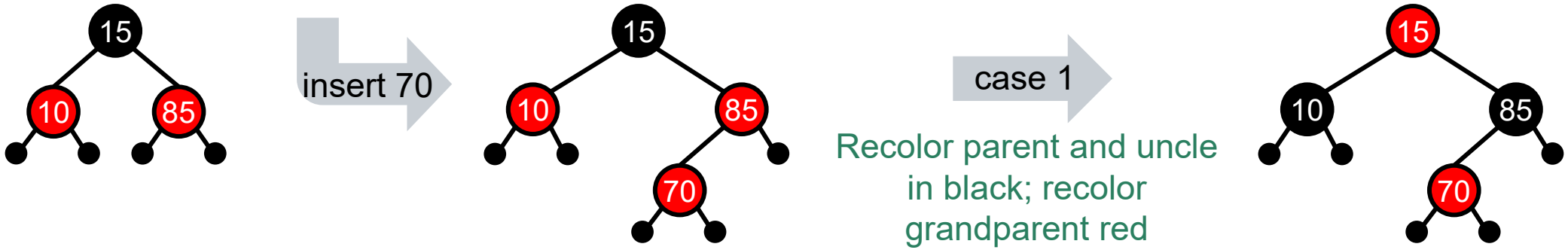


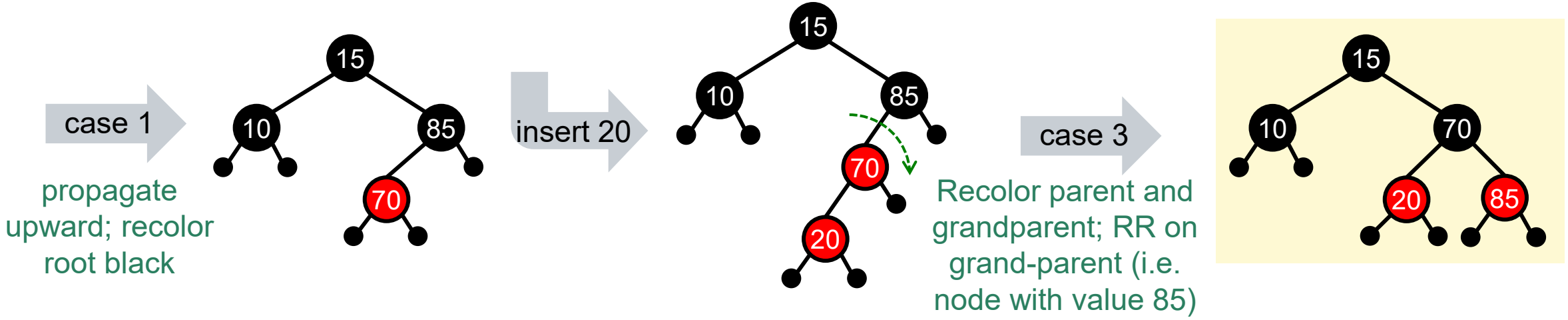
Red-Black Trees: Insertion Example / Exercise

The values **10 85 15 ; 70 20 ; 60 30 ; 50 ;** are inserted in the given order into an empty red-black tree. Draw the red-black tree at the positions marked by a semicolon. *(Task 2.3 of Midterm 2 from FS 2014)*

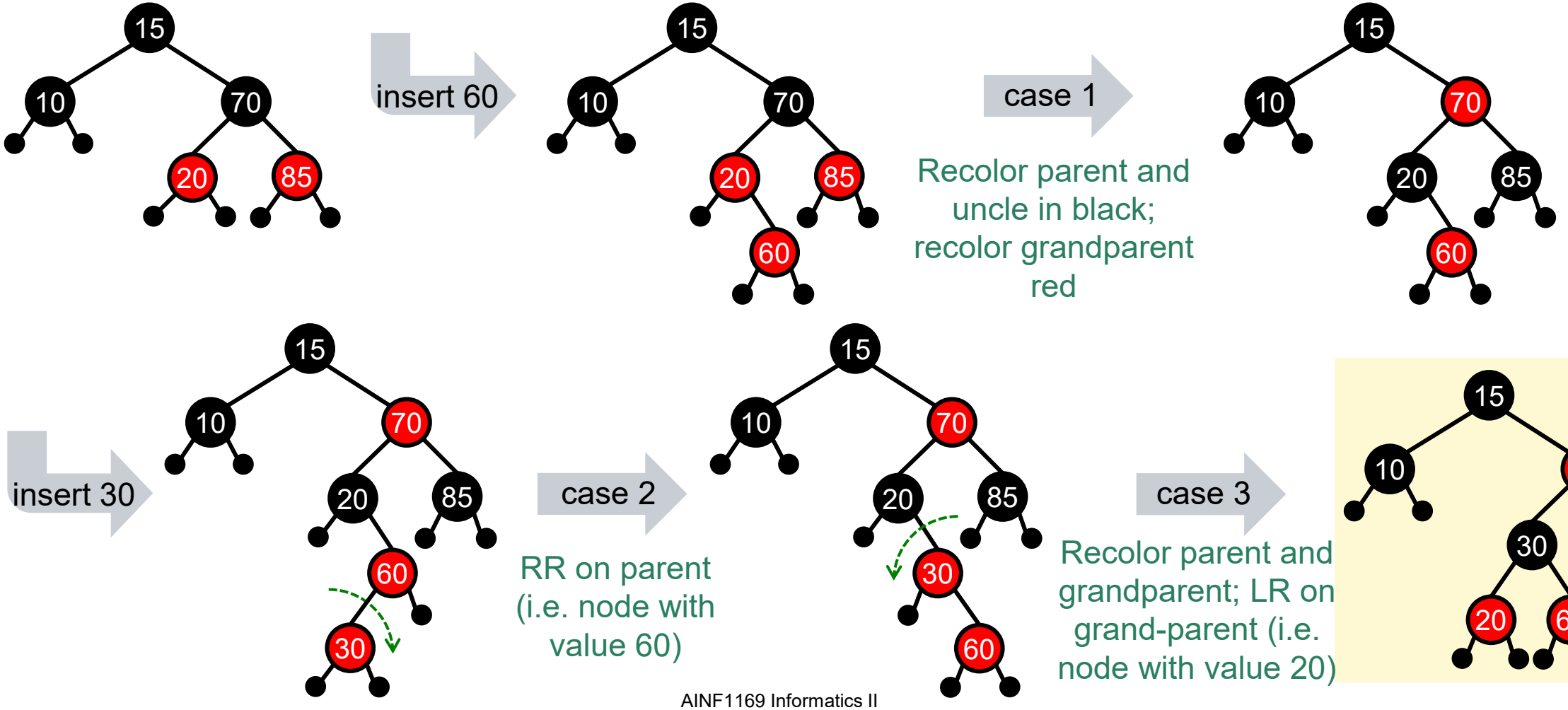


Red-Black Trees: Insertion Example / Exercise

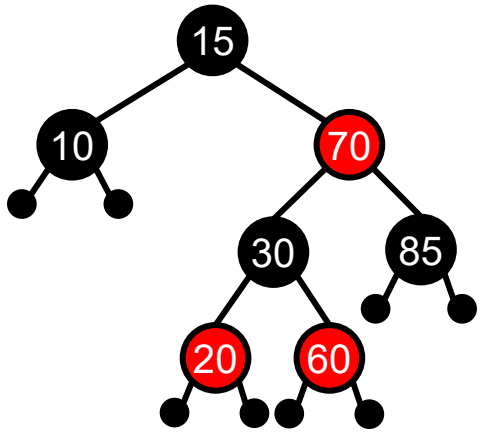




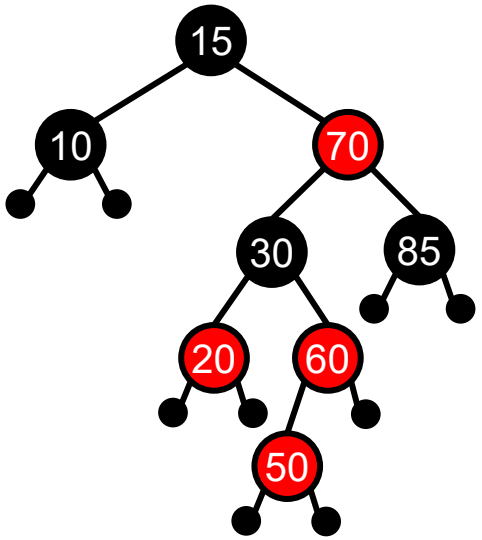
Red-Black Trees: Insertion Example / Exercise



Red-Black Trees: Insertion Example / Exercise

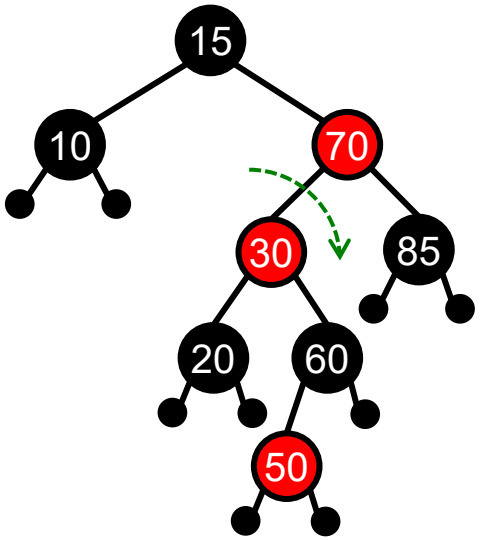


insert 50



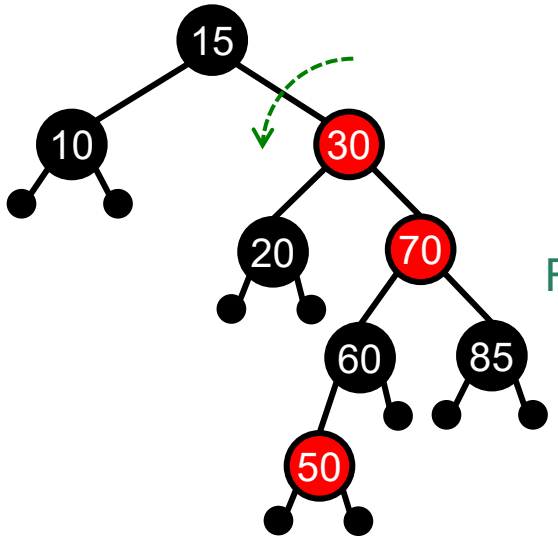
case 1

Recolor parent and uncle in black; recolor grandparent red; propagate with 30 as new node…



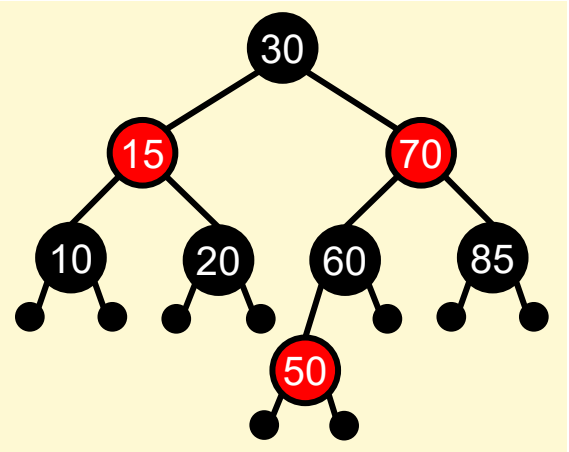
case 2

RR on parent (i.e. node with value 70)



case 3

Recolor parent and grandparent; LR on grand-parent (i.e. node with value 15); recolor the root black in the end



Red-Black Trees: Deletions

Deleting a node in a red-black tree is done in a two-stage operation:

① **Preparing the node to be removed**: If the node to be deleted has two (non-nil) children, it is overwritten with the value of the next smaller node in the tree without changing colors or structure in the tree and the node from where this copy has been taken is considered to be the node to be deleted.

② **Restoring the RB conditions**: The deletion may have broken the black-depth property (the red property cannot be broken because of the way step ① is performed). The depth property is restored by one of five rules which may involve recolorings and rotations and which might be applied recursively.

- While insertions lead to a violation of the red property (because inserting of the new node as red results in two consecutive red nodes) and has to be restored, the depth property is violated by deletions (because deleting of a black node results in a change of the black height in the respective subtree).
- While for insertions, the color of the uncle is checked to decide which case has to be applied, for deletions the color of the brother / sibling is checked to decide which case has to be applied.

Red-Black Trees: Deletions: Initial Preparatory Step

a) If the node which shall be deleted has no (non-nil) children or if it has only one (non-nil) child, i.e. if it has at least one nil child:

   → There is no preparatory work necessary and the process can be continued with step ②, i.e. restoring the RB conditions.

b) If the node which shall be deleted has two (non-nil) children, i.e. if it has no nil-children:

   → Copy: The value stored at the node which shall be deleted is overwritten with a copy of the largest value in its left subtree. Note that only the value is copied and the color and structure of the tree remain unchanged in this stage.

   → Reset the «flag of death»: The node from where this copy has been taken is considered to be the node which shall be deleted from now on and continue with step ②. This node is also called the replacement node.

Note that by applying the above technique, the node which is marked as the node to be deleted always will have either one non-nil child or no non-nil children. Also note that this initial preparatory step applies a similar approach as the removal of a node in a «normal» binary search tree with regard to the decision structure.

Red-Black Trees: Deletion Cases

- **Case 0:** Red node or red child: Either the node to be deleted is red or its single child is red.

  → nothing to do (if deleted node was red) or easy to fix by recoloring the child of the node to be deleted in black

- **Case 1:** Red brother: The depth property is violated and the node to be deleted has a red brother.

  → Recolor the brother in black and the parent in red; apply a rotation on the parent; continue with case 2, 3 or 4.

- **Case 2:** All black: The depth property is violated and the node to be deleted has a black brother which has two black children (nephews of node to be deleted).

  → Recolor the brother in red. Consider the parent to be the node to be deleted and apply the operation recursively, until the parent is not red.

- **Case 3:** Black brother, left nephew red: The node to be deleted has a black brother whose left child (the nephew of the node to be deleted) is red.

  → Recolor the left nephew in black. Recolor the brother in red. Apply a right rotation on the brother, then continue with case 4.

- **Case 4:** Black brother, right or both nephews red: The node to be deleted has a black brother which has at least the right child red (the color of the other child of the brother does not matter).

  → Recolor the brother in the color of the parent (may be black or red), recolor the parent and the right nephew in black, then rotate the parent.

either the node to be deleted or its replacement is red

both the node to be deleted and its replacement are black

Red-Black Trees: Deletion: Case 0: Red Node or Red Parent

If either the deleted node was red or it is black and has a single red child, there is either no violation of the depth property at all or the violation can easily be fixed.

There are two possible subcases:

- **red node** (case 0A): the node which shall be deleted is red

  → nothing to fix, since the depth property cannot be violated by removing a red tree (because we're only counting black nodes for this property)

- **red parent** (case 0B): the single child of the node which shall be deleted is red

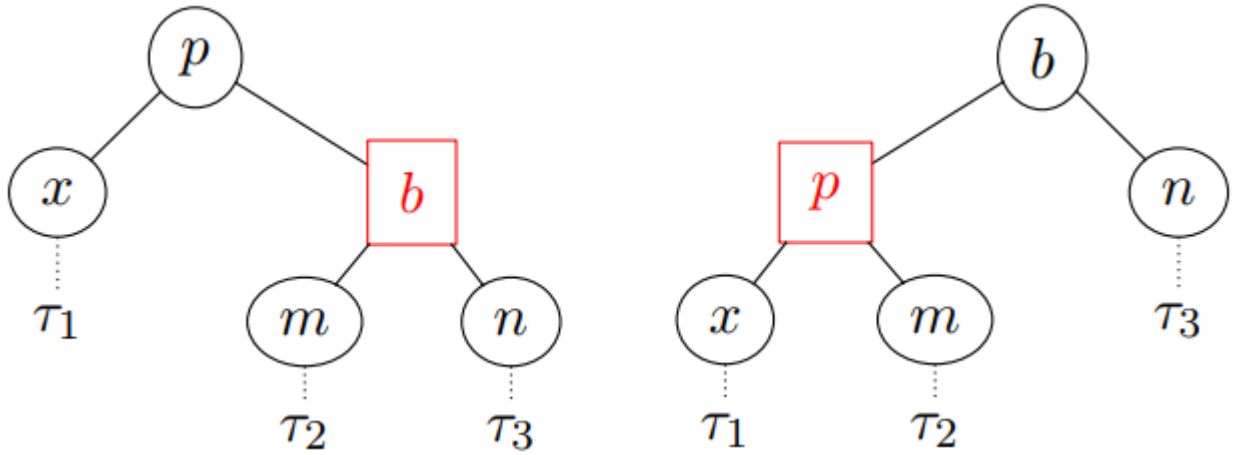
  → if applicable, set the color of the child of the node which shall be deleted to black; this makes up for the missing black node which would arise in black depth by deletion of a black node

Red-Black Trees: Deletion: Case 1: Red Brother

If the brother is red and the node to be deleted is black, then removing the node would lead to a decreased black depth in the subtree of the deleted node.

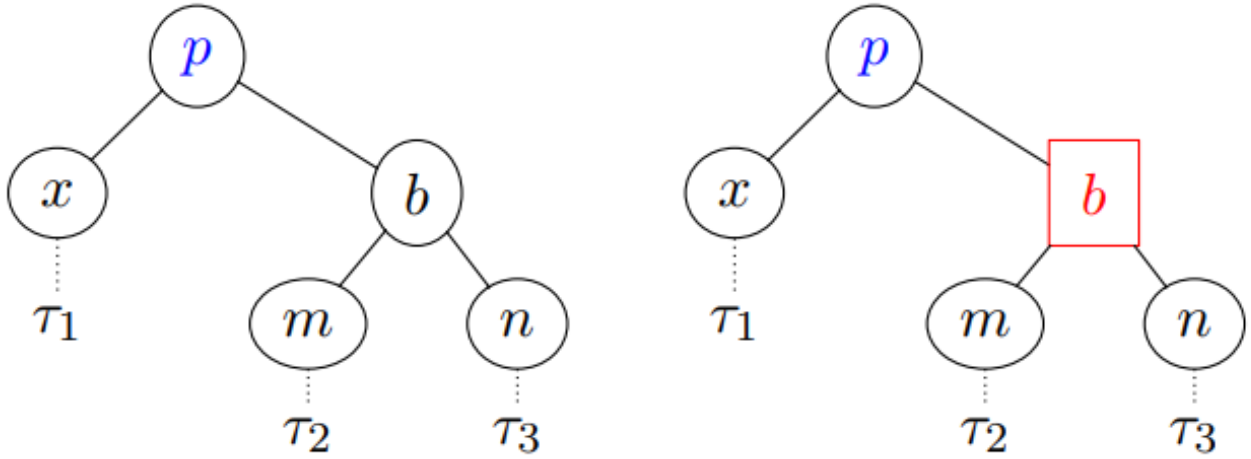
This can be fixed as follows:

1) Recolor the brother in black.
2) Recolor the parent in red.
3) Apply a left rotation on the parent of the node which shall be deleted. This converts the tree into a black brother case configuration.
4) Apply either case 2, 3 or 4 to fix the «double black» condition.



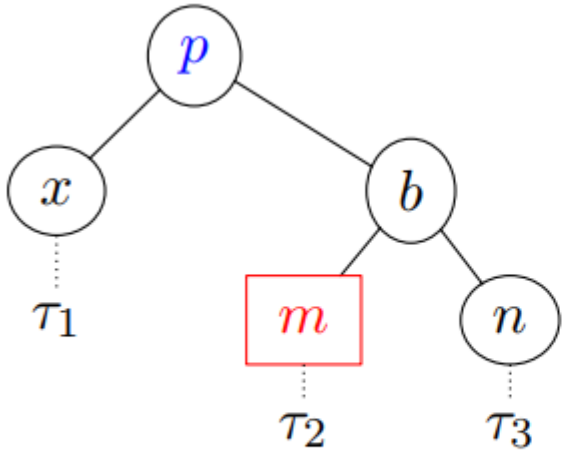
Red-Black Trees: Deletion: Case 2: All Black

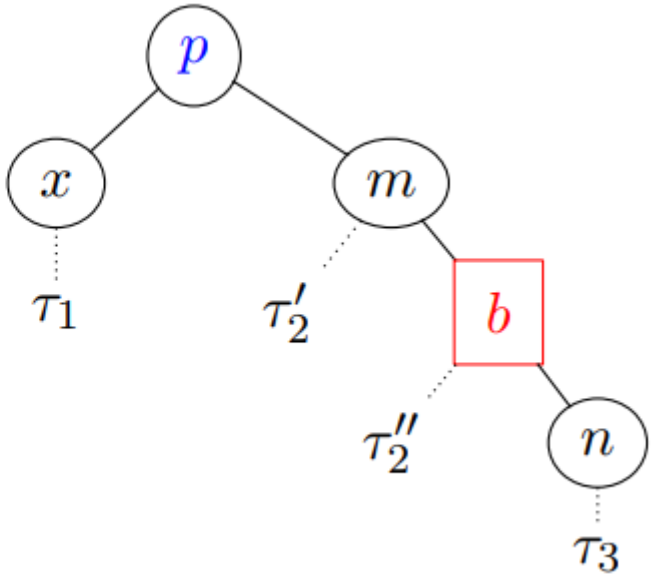
1) Recolor the brother in red.
2) Recursively call the delete procedure with the parent as the node which shall be deleted as long as the color of the parent is not red.
3) Set the color of the parent to black in the end.



Red-Black Trees: Deletion: Case 3: Black Brother, Left Nephew Red

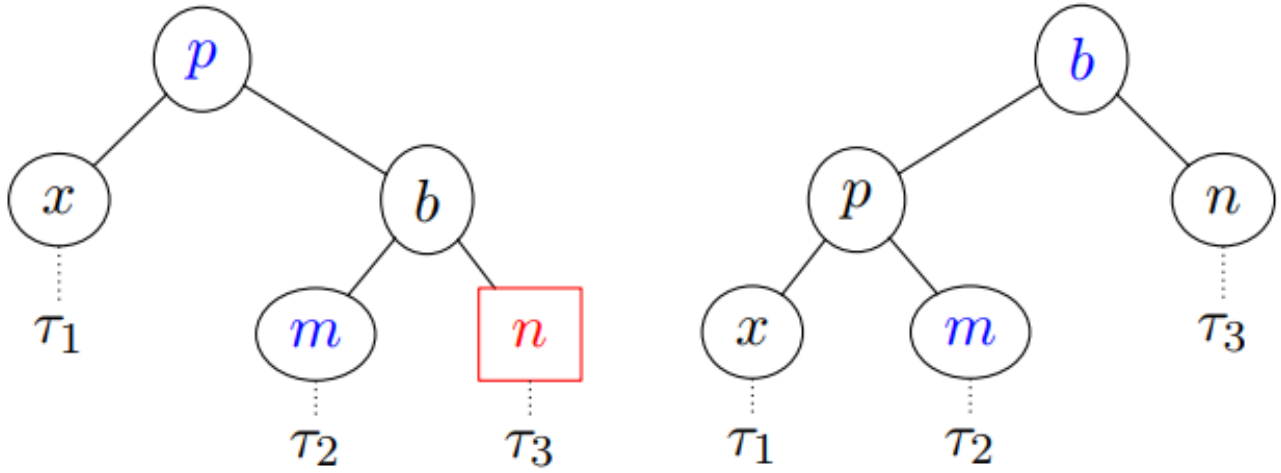
1) Recolor the left nephew in black.
2) Recolor the brother in red.
3) Apply a right rotation on the brother of the node which shall be deleted. This converts the tree into case 4 configuration.
4) Continue with case 4.





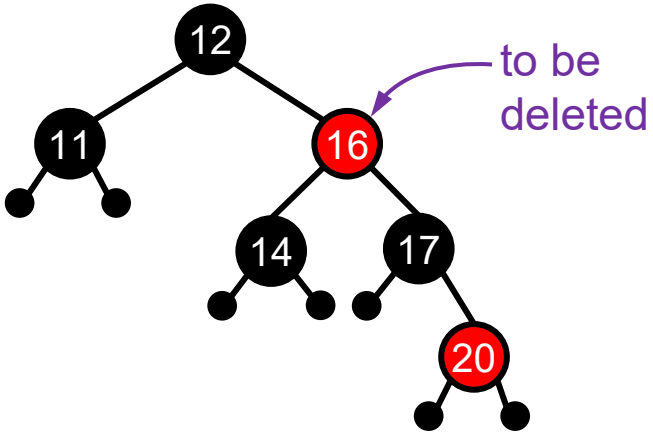
Red-Black Trees: Deletion: Case 4

1) Recolor the brother in the same color as the parent.
2) Recolor the parent and the nephew in black.
3) Apply a left rotation on the parent of the node which shall be deleted.



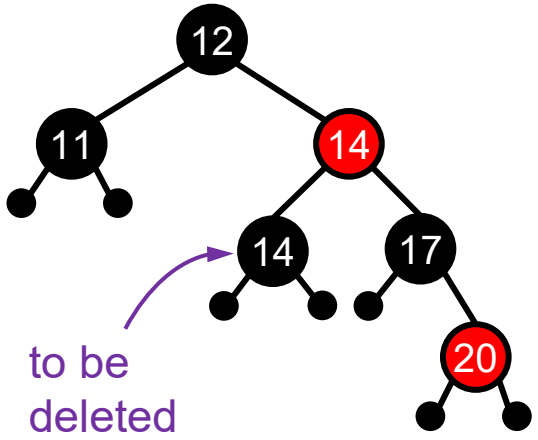
Red-Black Trees: Deletion Exercise

Delete the node with key 16 from the red-black tree given below. Show all intermediate steps. *(Task 3.1.1 of repetition exam from FS 2018)*



BST delete / replace

1) Get largest value from left subtree of the node which shall be deleted.
2) Replace the value of the node which shall be deleted with this value.
3) Mark the node from which the value was copied as the new node which shall be deleted.



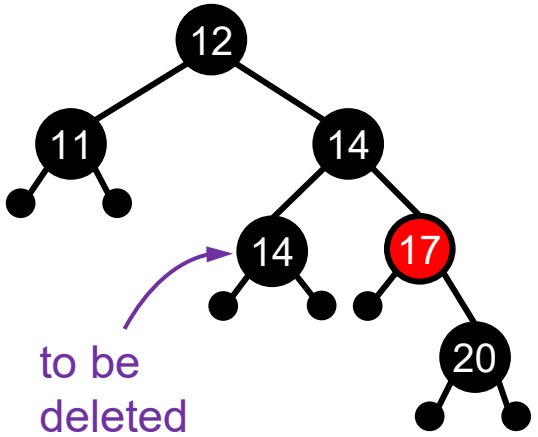
identify

Brother of node which shall be deleted after replacement step is black and has a red right child (= nephew).
→ Case 4

Red-Black Trees: Deletion Exercise

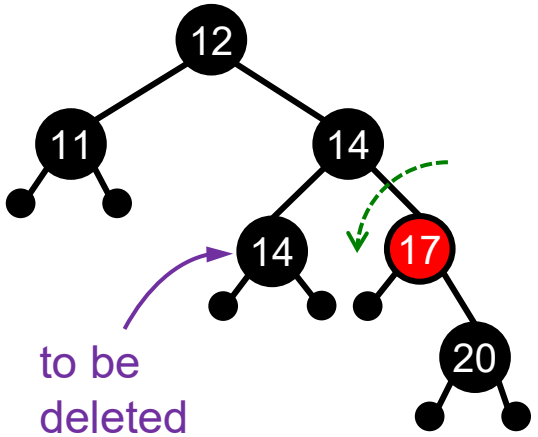
case 4 (part 1)

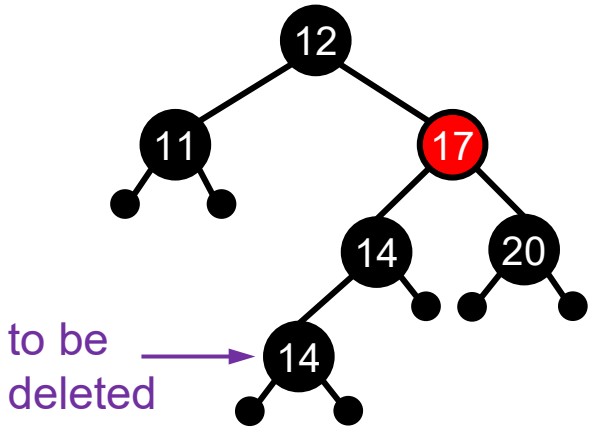
1) Recolor the brother in the color of the parent.
2) Recolor the parent and the nephew in black.



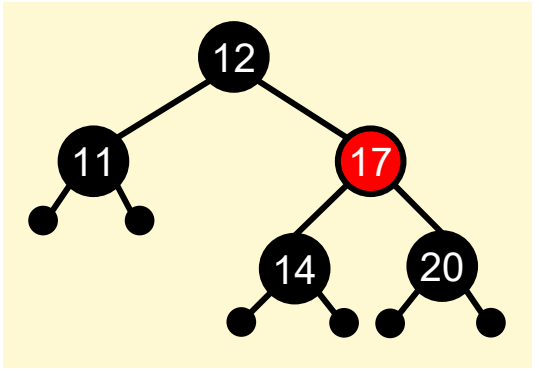
case 4 (part 2)

Left rotation on parent of node which shall be deleted.



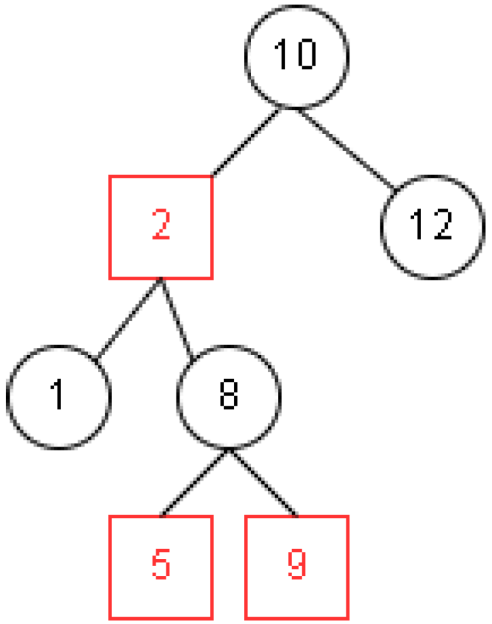


remove



Exercise 9, Task 2

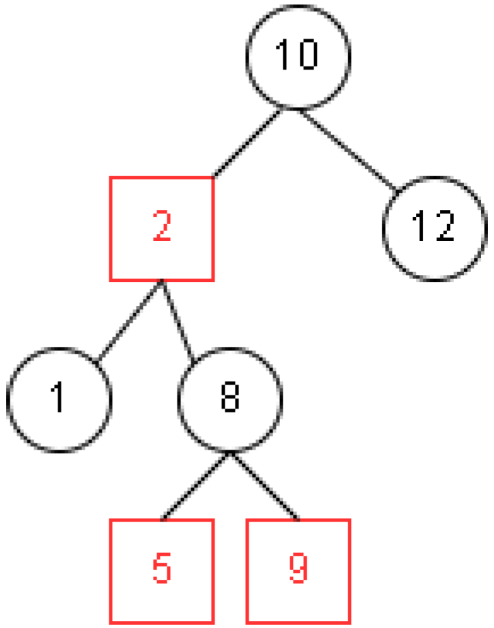
1. Consider the red-black tree shown below. State the operations that are required to insert 6 into the red-black tree.



2. Show the red-black tree that results after each of the integer keys 2, 3, 6, 7, and 1 insertion into an empty red black tree step by step.

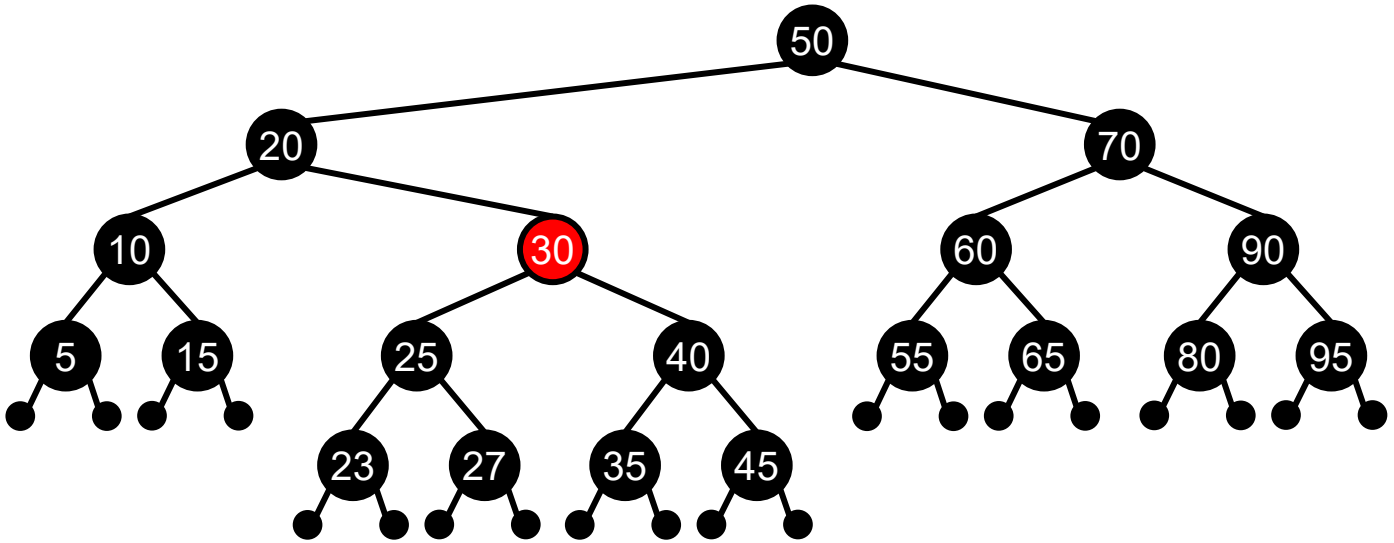
Exercise 9, Task 3

Delete 2 from the red black tree. Show state of tree after each step by drafts and explain the change.



Additional Exercise: Red-Black Tree Deletion

Delete the node with key 20 from the red-black tree given below. Show all intermediate steps.



Repetition on Red-Black Trees

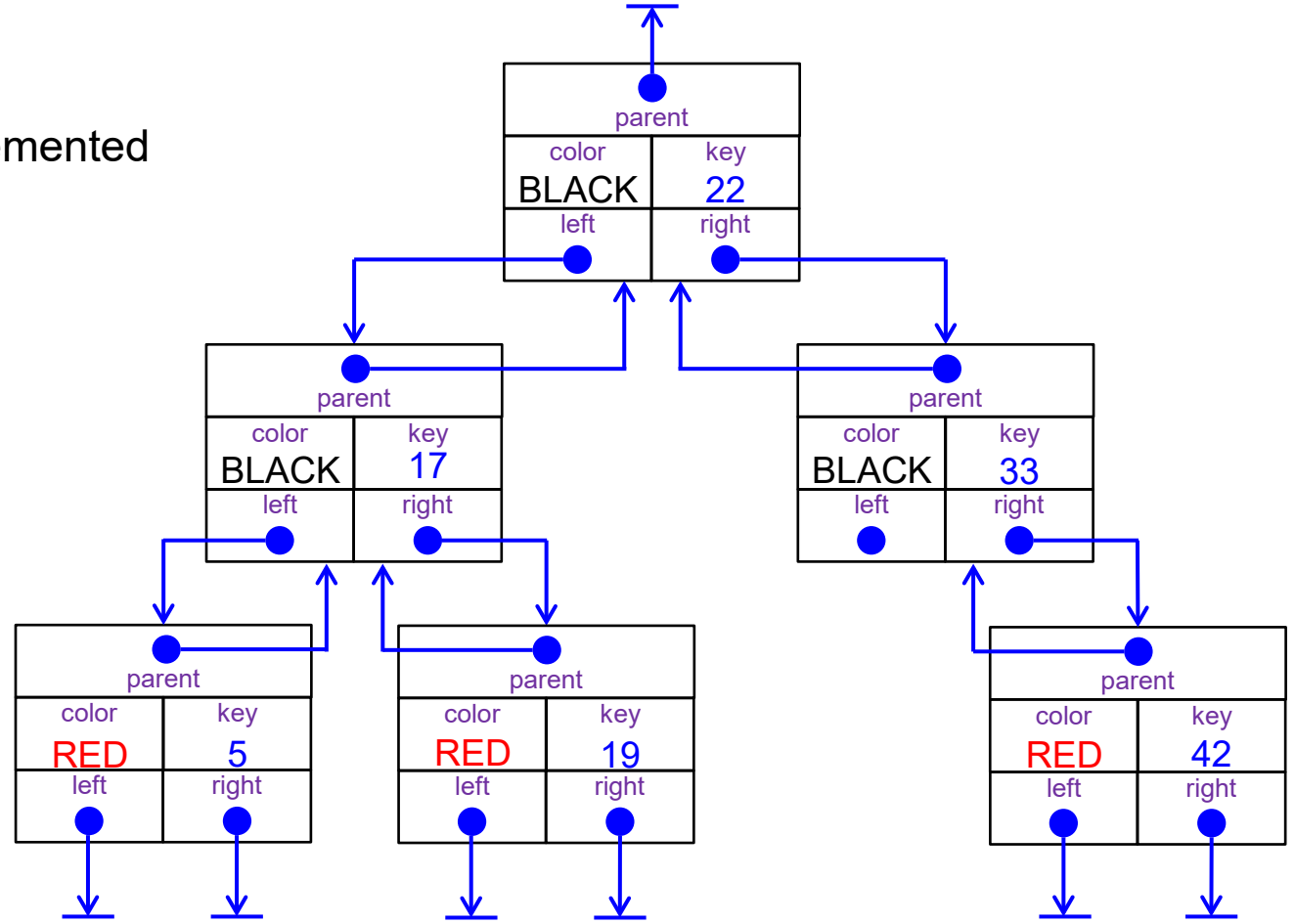
Potentially useful tools to understand and practice with red-black trees:

– Animation by David Galles: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
– Interactive animation by Tommi Kaikkonen: https://tommikaikkonen.github.io/rbtree/

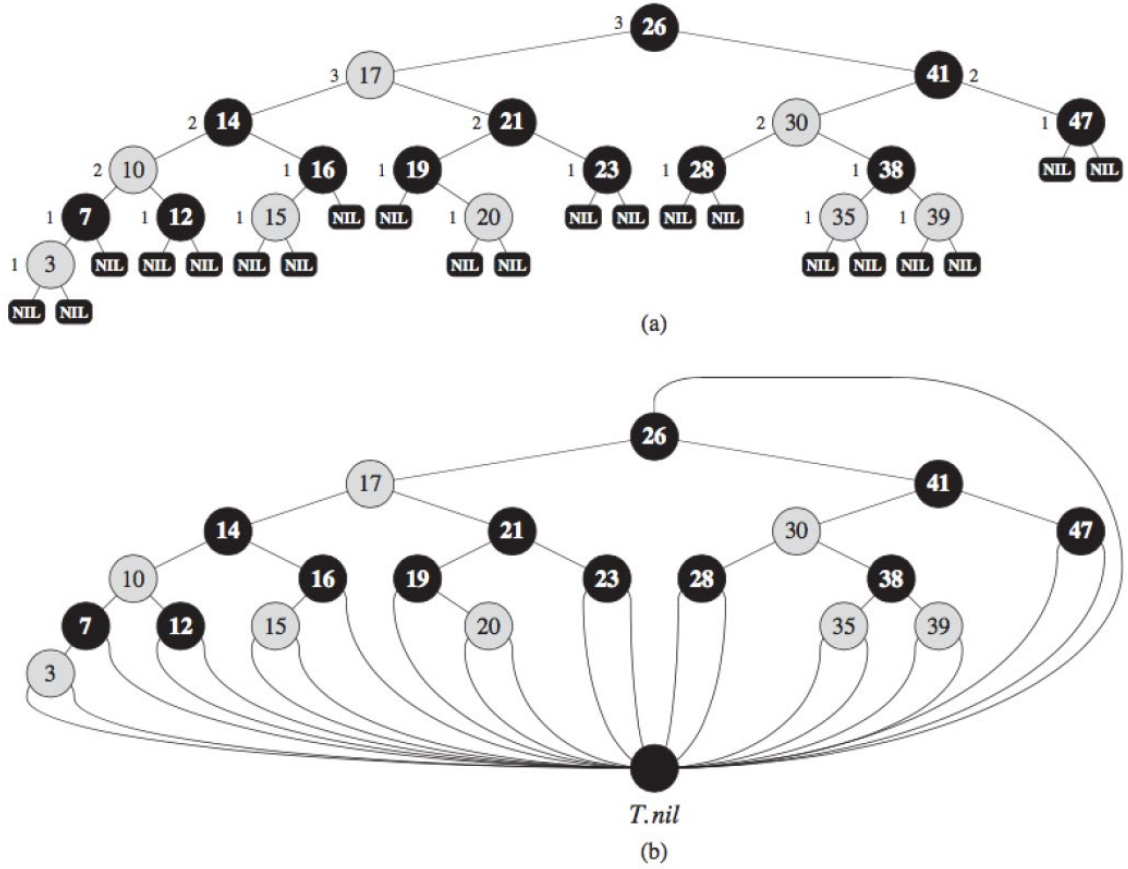
Red-Black Trees: Implementation

The nodes of a red-black tree can be implemented in C using a struct as follows:

```
struct rb_node {
        int key;
        int color;
        struct rb_node* left;
        struct rb_node* right;
        struct rb_node* parent;
};
```



Red-Black Trees: Implementation: Sentinel Nodes

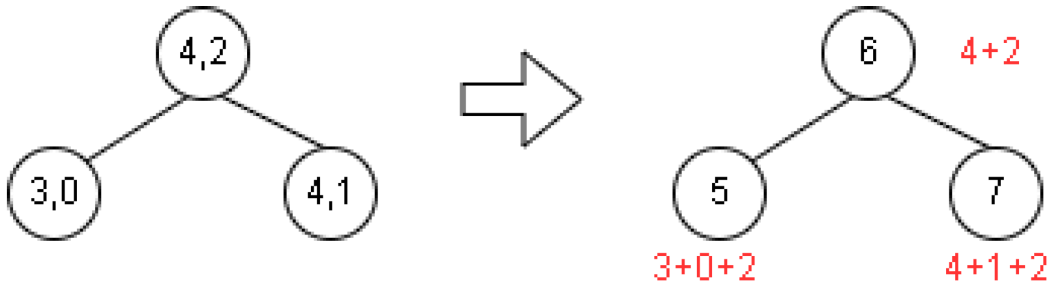


Red-Black Trees: Implementation: Comprehension Question

Why does the implementation of red-black trees contain parent pointers, while this is (usually) not the case for Binary Search Trees?

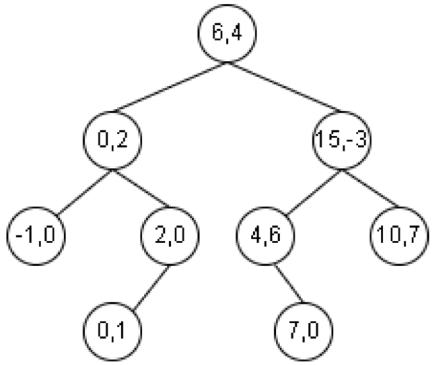
Exercise 9, Task 5

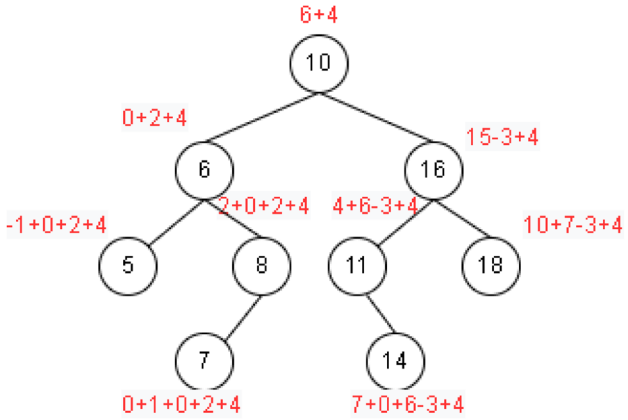
Consider an enhanced binary search tree in which each node v contains a key and an value called `addend`. The addend of node v is implicitly added to keys of all nodes in the subtree rooted at v (including v). Let (key, addend) denote the contents of a node. See the following example that transforms an enhanced binary search tree to a binary search tree.



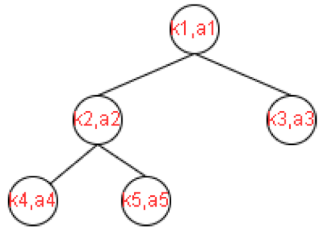
Exercise 9, Task 5

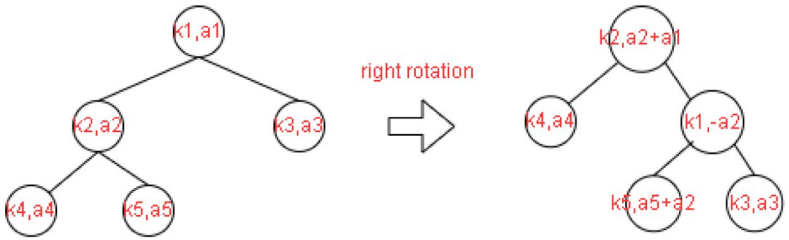
1. Show the binary search tree of the enhanced binary search tree below?





2. Consider the enhanced binary search tree below. Show the state of the enhanced binary search tree after the right rotation of node `k1`.



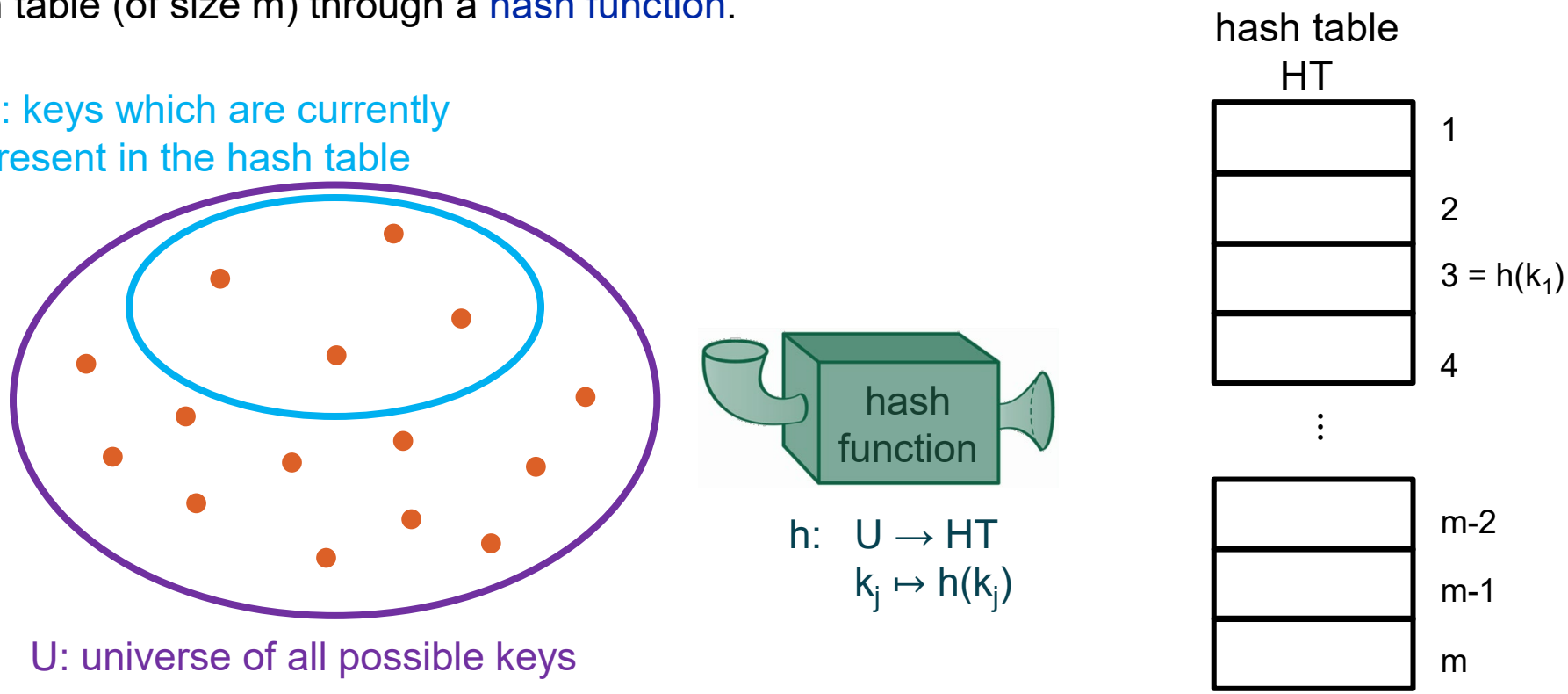


Hashing

- Introduction
- Hash Functions
- Collisions, Collision Resolution

Hashing: Basics

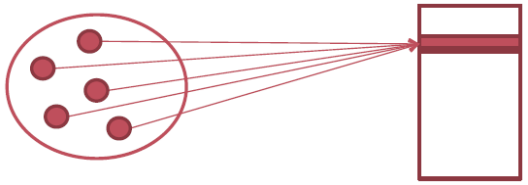
Each input key from a universe of possible keys is mapped to a certain, unambiguous index (value) in the hash table (of size m) through a hash function.



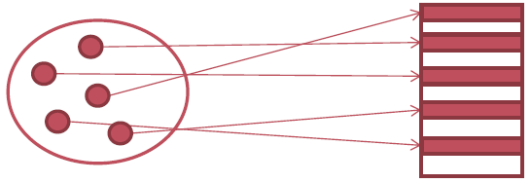
Desirable Characteristics of Hash Functions

A goof hash function should exhibit the following properties:

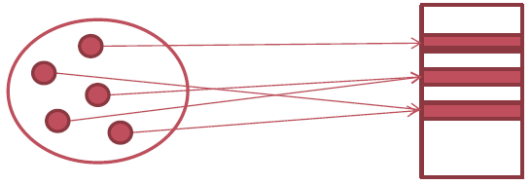
- The hash function should be fast to calculate (necessarily must be computed in O(1) time).
- The keys should be distributed uniformly to slots, i.e., not all keys should be mapped to the same slot but each bucket is assigned the same number of keys from the set of all possible values (this includes as a prerequisite that each available slot must be reachable by the hash function, i.e. there shouldn't be any slots which never get assigned a key).
- The distribution is random, i.e. patterns/regularities in the distribution of the keys (e.g. if all keys are even) should not affect uniform distribution of keys into slots



bad hash function



good hash function



maybe acceptable hash function

Designing / Choosing Hash Functions, Types of Hash Functions

It is not easy to find a good hash function and this does depend upon the characteristics of the input values (which usually are not known in advance). The following two general approaches to construct / types of hash functions are well-known and can produce good results under certain circumstances:

- **Division method**: $h(k) = (k \bmod m) + 1$ when indexing starts at 1, or
  $h(k) = k \bmod m$ when indexing starts at 0

- **Multiplication method**: $h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor + 1$ when indexing starts at 1, or
  $h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$ when indexing starts at 0
  (where $0 < A < 1$)

In all above cases, m will (should) denote the number of slots available in the hash table (otherwise, either not every index in the hash table might be reachable or invalid indices might result from the hash function).

Hashing: Collisions

If two input keys a and b have the same output values for a given hash function, i.e. if h(a) = h(b), this is called a collision.

We need a strategy to deal with such situations.

Collision resolution strategies (discussed in this lecture):

- Chaining

- Linear Probing
- Quadratic Probing
- Double Hashing

} open addressing

Open Addressing, Extended Hash Function

– While for the basic hash functions as discussed before, the hash value depends only on the key k which shall be inserted into the hash table, now we also consider an additional probing number i (which is equal to the number of collision that have occured before) as a second input:

$$h_{base}(k) \Rightarrow h_{ext}(k, i)$$

– The extended hash function is thus a modification of the basic hash function.

Collision Resolution Strategies Overview

- Chaining: add the key to the *front* of the linked list assigned to respective slot
- Linear probing: $h_{ext}(k, i) = (h_{base}(k) + c \cdot i) \bmod m$
- Quadratic probing: $h_{ext}(k, i) = (h_{base}(k) + c \cdot i + c \cdot i^2) \bmod m$
- Double hashing: $h_{ext}(k, i) = (h_{base}(k) + i \cdot h_{aux}(k)) \bmod m$

Hashing: Remark on the Modulo Function and Negative Numbers

In order to compute the modulus of a negative number, the following property of the modulo function can be used:

$$a \bmod m = (a + k \cdot m) \bmod m$$

Examples:

- $-3 \bmod 4 = (-3 + 4) \bmod 4 = 1 \bmod 4 = 1$
- $-17 \bmod 7 = (-17 + 3 \cdot 7) \bmod m = 4 \bmod 7 = 4$

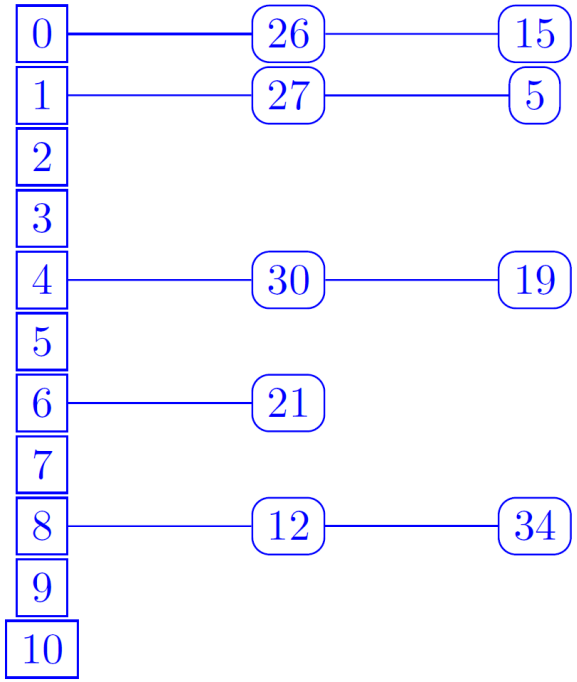
This can also be calculated as follows:

$$-a \bmod m = (a \cdot (m - 1)) \bmod m$$

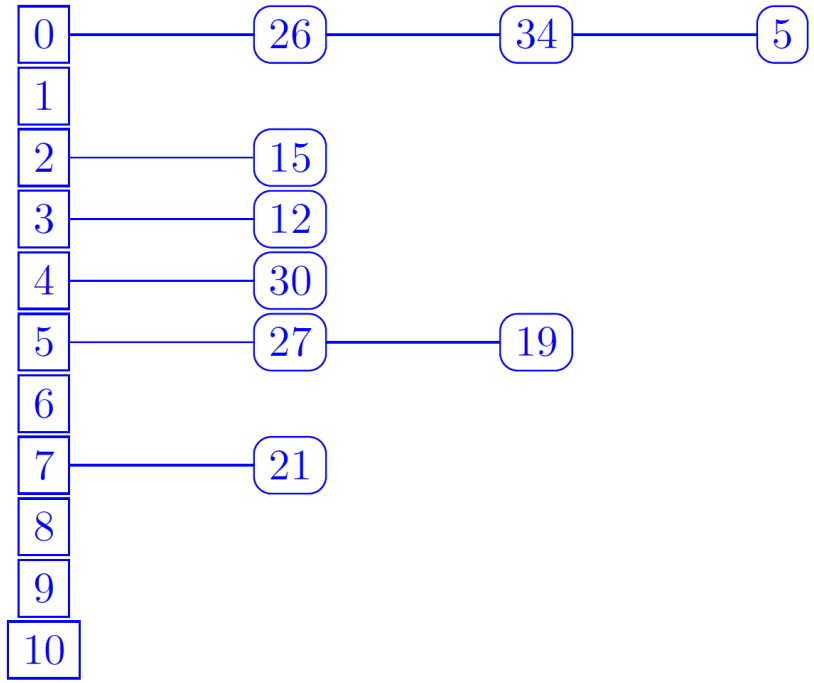
Exercise 10, Task 1.2: Division and Multiplication Method: Solution

Keys to insert: 5, 19, 27, 15, 30, 34, 26, 12, 21

a) $h_1(k) = (k + 7) \bmod 11$ (division method)



b) $h_2(k) = \lfloor 8 \cdot (k \cdot 0.618 \bmod 1) \rfloor$ (multiplication method)



Exercise 10 – Task 2: Linear Probing and Double Hashing

Keys to insert:

5, 19, 27, 15, 30, 34, 26, 12, 21

a) $h_1(k, i) = (k + i) \bmod 11$
(linear probing)

b) $h_2(k, i) = (k \bmod 11 + i(k \bmod 7)) \bmod 11$
(double hashing)

a)

Slot	Value
0	
1	34
2	12
3	
4	15
5	5
6	27
7	26
8	19
9	30
10	21

b)

Slot	Value
0	27
1	34
2	
3	
4	15
5	5
6	12
7	
8	19
9	26
10	30

Hashing: Comprehension Question

- A good hash function should have the property that it avoids hitting the same target slot again and again, i.e. we like to have a hash function with as few collisions as possible, i.e. it should scatter as evenly as possible throughout the target set / slots.
- In order to calculate the hash value for a given input key, certain properties of the input keys are taken into consideration and used in the calculation (h(k)).
- The difficulty of constructing a hash function is, to construct it in a way which makes sure it will scatter as good as possible as explained above.

- Question: Do we necessarily have to take some property of the input key in order to calculate the hash value? Why? Why can't we just take a random value for example when inserting a new key into the hash table? In this way, we would be guaranteed statistically to get an even scatter for large values automatically…

States of Slots, Searching and Deleting in an Hash Table

Each slot of a hash table is assigned one of three different states:

- **OCC**: the slot is occupied
- **EMP**: the slot is empty and always has been empty
- **DEL**: the slot is empty but was previously occupied (the content has been deleted)

These states have to be considered when inserting, searching and deleting in the hash table:

- Inserting: apply hash function; if resulting slot is occupied, apply next iteration of hash function; repeat until empty or deleted slot is reached; when slot found: insert and set state to occupied
- Searching: apply hash function; if resulting slot is occupied and value in the slot is the value we search for: return; if empty: return error; if occupied and not the key or if previously occupied, apply next iteration of hash function; repeat until key is found and has been returned or until empty slot is discovered which yields to returning «not found» message

States of Slots, Inserting and Searching in an Hash Table

Algorithms in pseudocode:

Algo: HTinsert(HT,k)

```
i = -1;
repeat
  | i++; probe = h(k,i);
until i≥m ∨ HT[probe].status≠OCC;
if i≥m then return -1;
HT[probe].status = 0;
HT[probe].key = k;
return probe
```

Algo: HTsearch(HT,k)

```
i = -1;
repeat
  | i++; probe = h(k,i);
until i≥m ∨
      HT[probe].status==OCC ∧ HT[probe].key==k ∨
      HT[probe].status==EMP;
if i≥m ∨ HT[probe].status==EMP then return -1;
return probe
```

States of Slots, Deleting in an Hash Table

Algorithm in pseudocode:

Algo: HTdelete(HT,k)

```
i = -1;
repeat
  | i++; probe = h(k,i);
until i≥m ∨
        HT[probe].status==EMP ∨
        HT[probe].status==OCC ∧ HT[probe].key==k;
if i≥m ∨ HT[probe].status==EMP then return -1;
HT[probe].status = DEL;
return probe
```

Implementation of a Hash Table in C

```c
#define OCC 0
#define EMP -1
#define DEL -2

struct HTElement {
     int value;
     int status;
};

#define m 7

struct HTElement hash_table[m];
```

Exercise 10, Task 2: Implementing a Hash Table in C

Implement a hash table with m slots, where m is a positive integer. Note that m = 0 indicates an empty hash table. All keys are positive integers. Assume that conflicts are resolved with double hashing defined as follows:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

$$h_1(k) = (k \bmod m) + 1$$

$$h_2(k) = m' - (k \bmod m')$$

$$m' = m - 1$$

Your program should include the following:

- The number m corresponding to the size of the hash table. The struct `HTElement` for elements of the hash table.
- The function `void init(struct HTElement A[])` fills all slots of the hash table A with the value 0.
- The hashing function `int hash(int k, int i)` that receives the key k and the probe number i and returns the hashed key.
- The function `void insert(struct HTElement A[], int key)` inserts the key into the hash table A.
- The function `void delete(struct HTElement A[], int key)` deletes the key from the hash table A.
- The function `int search(struct HTElement A[], int key)` that returns -1 if the key was not found in the hash table A. Otherwise, it should return the index of the key in the hash table.
- The function `void printHash(struct HTElement A[])` prints the table size and all non-empty slots of the hash table A accompanied with their index and the key.

Exercise 10, Task 2: Implementing a Hash Table in C: Solutions

```
4  void init(struct HTElement A[]) {
5       int i;
6       for (i = 0; i < m; i++) {
7            A[i].value = 0;
8            A[i].status = EMP;
9       }
10 }
11
12 int hash(int k, int i) {
13      int h1 = (k % m) + 1;
14      int h2 = (m-1) - (k % (m-1));
15      return (h1 + i * h2) % m;
16 }
17
```

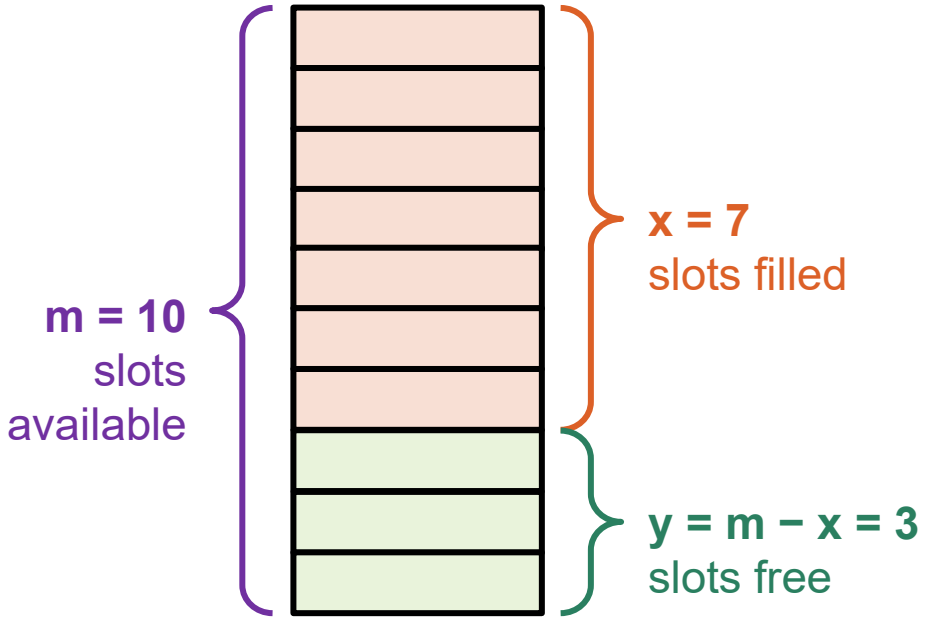
Hashing: Load Factor (α)

For analysis of hashing and comparing different techniques for collision resolution, an important number is the load factor α. It is defined as the number x of items currently stored in the hash table divided by the number m of slots available in the hash table:

$$\alpha = \frac{x}{m}$$

Since at most element can be in a slot, i.e. $x \leq m$, it must be true that $\alpha \leq 1$ for a table using open addressing.

(Remark: Obviously, the filled slots do not need to – and usually will not – occupy a continuous space in the hash table as depicted in the adjacent figure.)



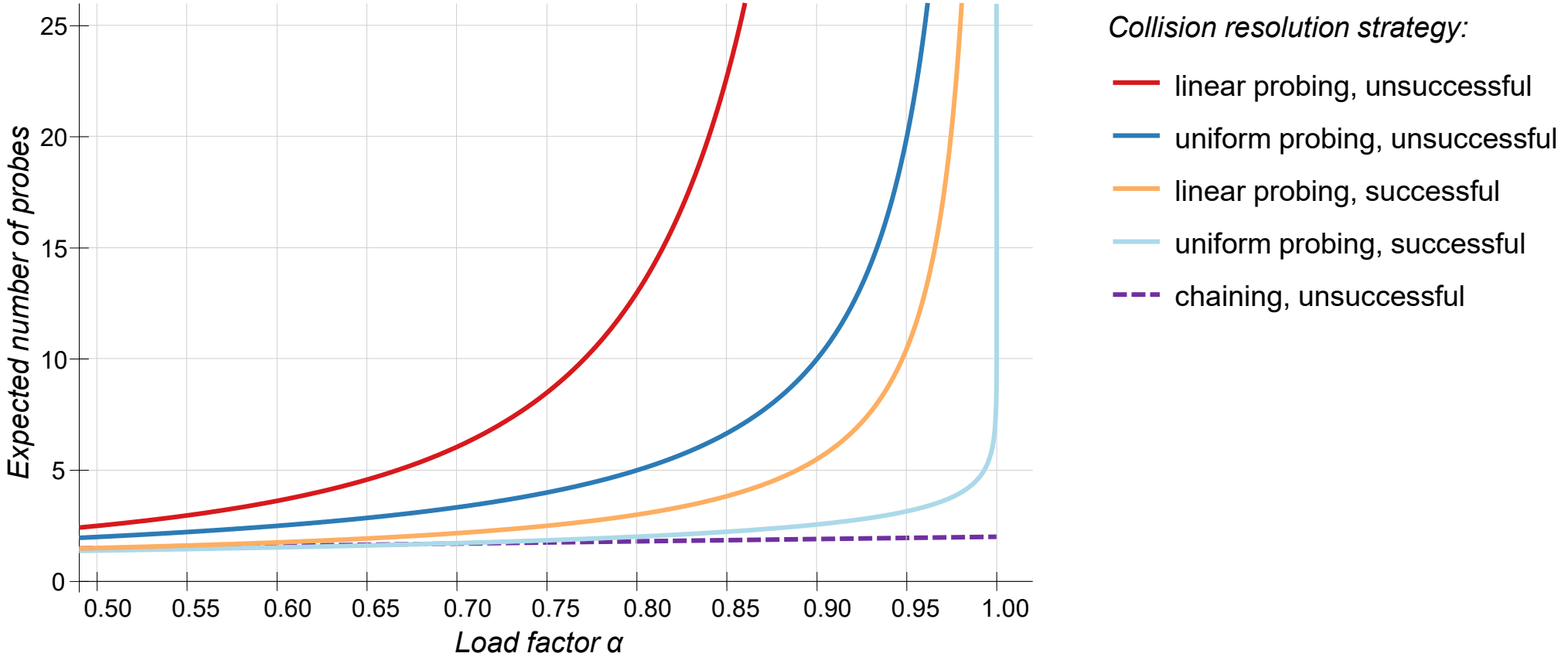
α = 7 / 10 = 0.7 = 70%

Overview: Cost of Search in a Hash Table

Expected number of probes for a successful / unsuccessful search in a hash table where m is the number of slots of the hash table, n is the number of slots currently occupied and $\alpha = n / m$ denotes the load factor. Note that insertion and deletion also need a search.

Collision resolution strategy	*Unsuccessful search*	*Successful search*
Uniform probing (≈ Double hashing)	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right)$
Linear probing	$\frac{1}{2} \cdot \left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$
Chaining	$\alpha + 1$	$1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$

Visualization of Expected Number of Probes for Different Collision Resolution Strategies



Exercise 10, Task 3: Birthday Paradox

Hashing: Efficiency

Hashing is very efficient / fast.

	unsorted singly linked list, worst-case	sorted doubly linked list, worst-case	min-heap, worst-case	hash table, worst-case	hash table, average-case
SEARCH(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
INSERT(L, x)	$O(1)$	$O(n)$	$O(lgn)$	$O(1)$ or $O(n)$	$O(1)$
DELETE(L, x)	$O(n)$	$O(1)$	$O(lgn)$	$O(1)$	$O(1)$

Comprehension question:

If hashing provides such fast means to access data, why do we even care about other data structures like binary trees for example?

Exercise 10, Task 1.1

Is the following statement true or false? Justify your answer.

"A hash table can be used directly for efficient range queries, i.e., queries that involve a range of keys. For example, find all keys that are smaller than a given key."

→ **false**

Philosophical / Epistemological Question

Are concepts like hash tables discovered or are they invented?

Wrap-Up

- Summary
- Feedback
- Outlook
- Questions

Wrap-Up

– Summary

Outlook on the Next Lab Session

Next tutorial: Wednesday, 18.05.2022, 14.00 h, BIN 0.B.06

Topics:

- Review of Exercise 11
- Dynamic Programming
- …
- … (your wishes)

Additional Training / Repetition Sessions

I will be here in BIN 0.B.06

- next Friday, 13th of May, from 14.15 h
- Next Wednesday, 18th of May, from 16.00 h

No new or additional topics will be covered.

Only repetition and training. Will rely on your input.

Questions?

Thank you for your attention.