Computational Geometry

3. Line Segment Intersection

Content

3.1 Motivation

3.2 Naïve approach

3.3 Scan-line approach

- **3.3.1** Iso-oriented segments
- **3.3.2** Arbitrary segments

3.4 Literature

Appendix: AVL trees

Appendix: Heaps

3.1 Motivation

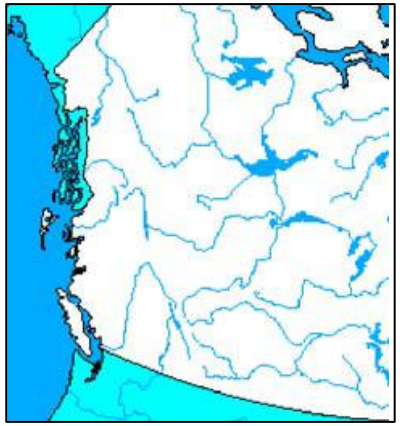
- In a geographical information system (GIS) information is stored in layers of different maps (e.g. streets, rivers, villages,…).
- To mark bridges these layers are overlaid and all intersections of streets with rivers must be computed.



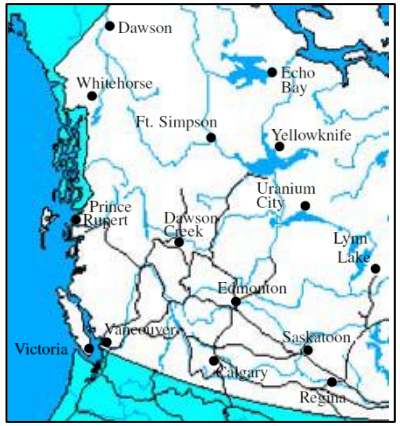
Cities of western Canada



Railroads of western Canada



Rivers of western Canada



Overlay of these maps

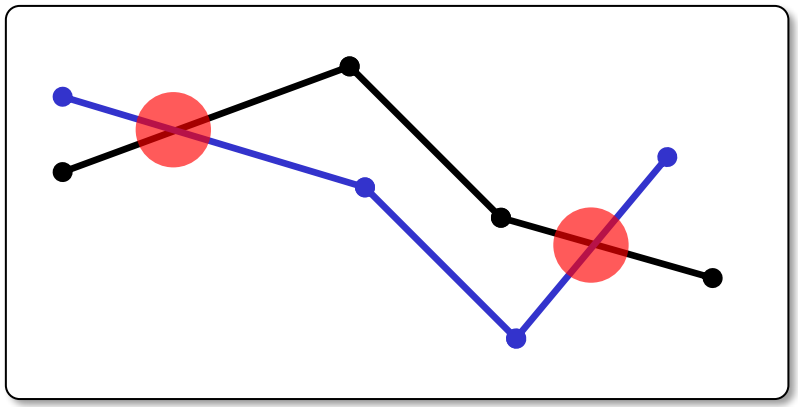
Source: de Berg et al.

3.1 Motivation

- If streets and rivers are approximated by polygon chains, intersections of line segments must be computed.
- This may exceed the capabilities of a GIS, if the intersections of every street with every river is computed.

3.1 Motivation

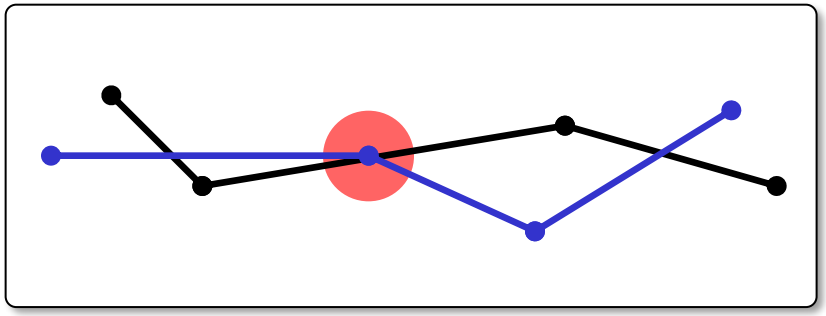
- Example:
  - A polygon chain of layer one,
  - a polygon chain of layer two, etc.



- For simplicity do not consider polygon chains but line segments only.

3.1 Motivation

- Principal approach
  - Add all line segments of all polygon chains to a single set and compute all intersections.
  - Then search for the intersections of individual polygon chains.
  - **Attention:** It might happen that a „real intersection“ coalesces with the start and end point of two adjacent line segments.



Content

3.1 Motivation

3.2 Naïve approach

3.3 Scan-line approach

- **3.3.1** Iso-oriented segments
- **3.3.2** Arbitrary segments

3.4 Literature

Appendix: AVL trees (yet in German)

Appendix: Heaps (yet in German)

3.2 Naïve approach

- **Input:** Set $S = \{s_1, \ldots, s_n\}$ of n line segments in the plane.
- **Goal:** All intersections of line segments from S.
- **Naïve Solution:**
  1. Test all pairs of segments.
  2. If they intersect, return the intersection point.
- **Run time:** $\Theta(n^2)$
- Optimal, if all pairs of line segments intersect, because then there are $\Theta(n^2)$ intersection points!
- Usually, the number of intersections is much smaller.

Content

3.1 Motivation

3.2 Naïve approach

3.3 Scan-line approach

- **3.3.1** Iso-oriented segments
- **3.3.2** Arbitrary segments

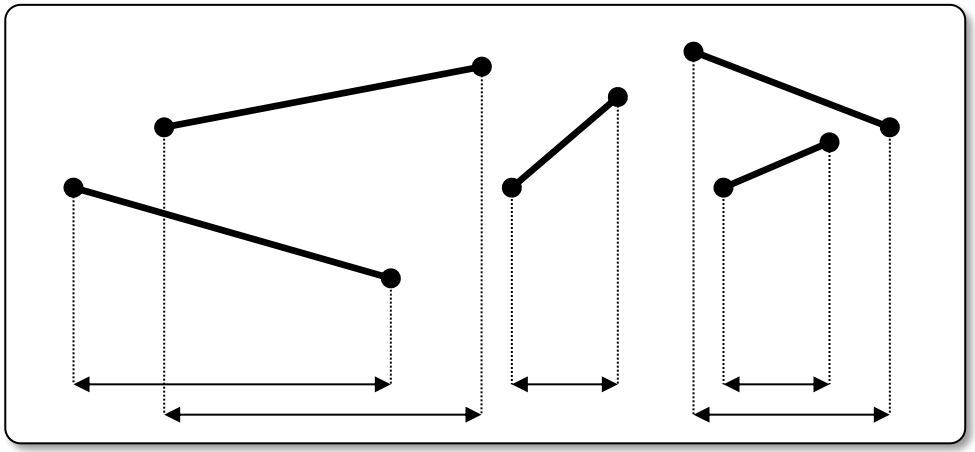
3.4 Literature

Appendix: AVL trees (yet in German)

Appendix: Heaps (yet in German)

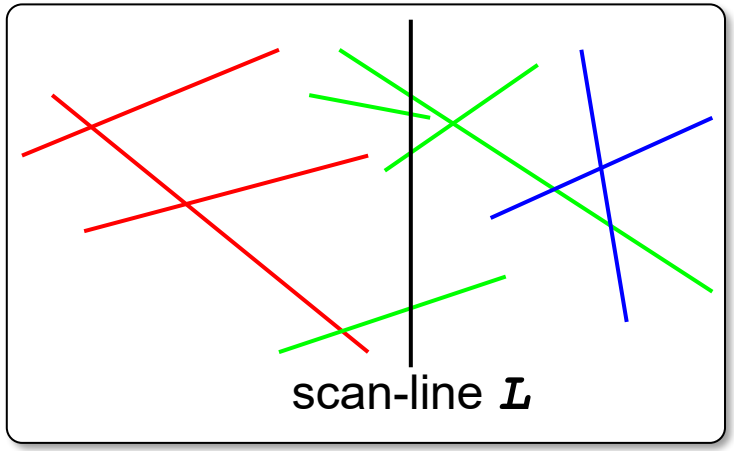
3.3 Scan-Line-Approach

- **Input:** Set $S = \{s_1, \ldots, s_n\}$ of n line segments in the plane.
- **Idea:** Test only segments that are close to each other, because then it is likely that they intersect.
- Scan-line principle
  - Scan-line algorithms are more efficient for a relatively small number of intersections (*output-size sensitive*).



3.3 Scan-Line-Approach

- **Approach**
  - Move a vertical line L (scan-line or sweep-line) from left to right over the set of line segments.
- The scan-line L clusters the segments in three classes:
  - ***dead objects***: lie left of L,
  - ***active objects***: intersect L,
  - ***inactive objects***: lie right of L.
- The state of L is the set of active objects.



- The state of L is only changed at start or end points of line segments.
  - These points a called *event points*.

Content

3.1 Motivation

3.2 Naïve approach

3.3 Scan-line approach

3.3.1 Iso-oriented segments

3.3.2 Arbitrary segments

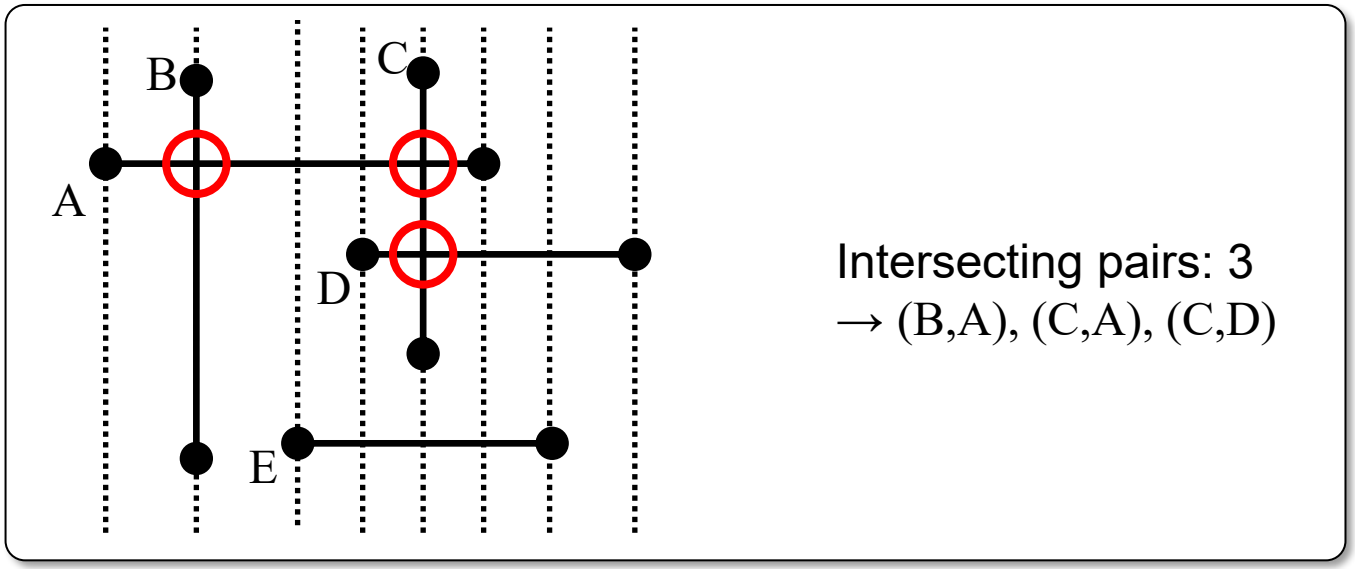
3.4 Literature

Appendix: AVL trees (yet in German)

Appendix: Heaps (yet in German)

3.3.1 Iso-oriented segments

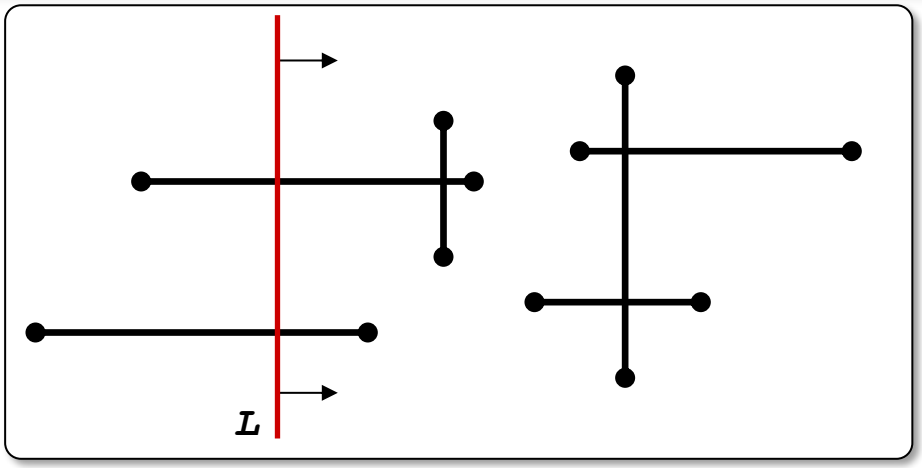
- For simplicity first the simpler case of ***iso-oriented line segments***.
- **Input:** Set $S = \{s_1, \ldots, s_n\}$ of n vertical and horizontal line segments in the plane.
- **Goal:** All pairs of intersecting line segments in S.



3.3.1 Iso-oriented segments

- What is the maximal number of intersections of n iso-oriented line segments?
  - 50% of the segments are vertical and 50% are horizontal.
  - All possible intersections: $\left(\frac{n}{2}\right)^2$
- Simplifying assumption:
  - All start and end points of horizontal segments und all vertical segments have mutually distinct x-coordinates.
  - This eliminates some special cases in the algorithm.
  - For our example these assumptions are satisfied.

3.3.1 Iso-oriented segments



- **Observation 1**
  - It is not necessary to move the scan line L continuously, since only the event points are important.
  - Thus, there are three kinds of event points with respect to the x-coordinates:
    - start of a horizontal segment,
    - end of a horizontal segment,
    - vertical segment.
  - If L hits a vertical segment, this must be an active vertical segment.

3.3.1 Iso-oriented segments

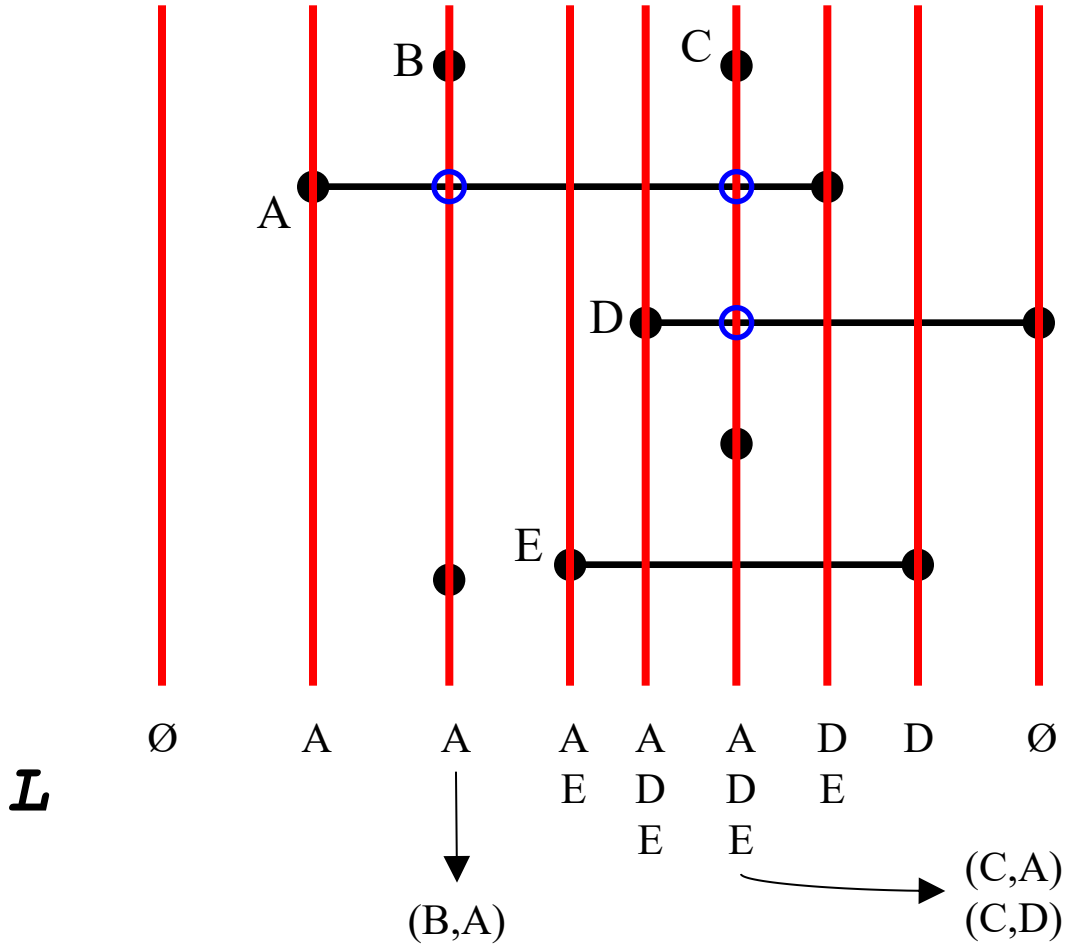
Algorithm 1: Intersect iso-oriented line segments

Input: n iso-oriented line segments.
Output: All intersections.

```
 1: Initialize set Q with all events sorted by x-coordinate;
 2: L = ∅; // List of active segments, sorted by y-coordinate
 3: while (Q is not empty) {
 4:   Take next event p from Q and remove it from Q;
 5:   if (p is left end point of horizontal segment s) then {
 6:     Add s to L;
 7:   } else if (p is right end point of segment s) then {
 8:     Remove s from L;
 9:   } else { // p.key is the x-coordinate of a vertical
                // segment s with lower end point (p.key,y_l)
                // and  upper end point (p.key,y_u)
10:     Determine all horizontal segments t in L, whose
        y-coordinate t.y is in [y_l,y_u] and report the
        intersecting pair (t,s);
11:   }
12: }
```

3.3.1 Iso-oriented segments

- **Example**



3.3.1 Iso-oriented segments

- Requirements for the data structure for $\boldsymbol{L}$ (*event structure*)
  - Horizontal elements are stored in $\boldsymbol{L}$ with their y-coordinate as key.
  - Efficient operations for
    - **`insertion`** of a new element,
    - **`deletion`** of an element,
    - output of all elements in $\boldsymbol{L}$, whose key lies within a certain range (**`range_search`**).
  - Possible solution: balanced binary search trees, e.g. AVL-trees ([1,2]), red-black-trees, etc.

3.3.1 Iso-oriented segments

- Operations for the data structure for $\boldsymbol{L}$
  - `insertion` and `deletion` have run time $O(\log|\boldsymbol{L}|)$.
  - Line 10: `range_search` for horizontal line segments p with y-coordinate in $[y_l, y_u]$
    - Find element p with smallest key $\geq y_l$.
    - Return p, if key $\leq y_u$.
    - Traverse tree from p in in-order-sequence, i.e.
      - left sub-tree – root – right sub-tree
    - and return all visited knots, as long as key $\leq y_u$.
      - This is assumed to be known and will not be detailed here.
    - Auxiliary function `successor(knot q)`: Successors of knot $\boldsymbol{q}$ in in-order-sequence.

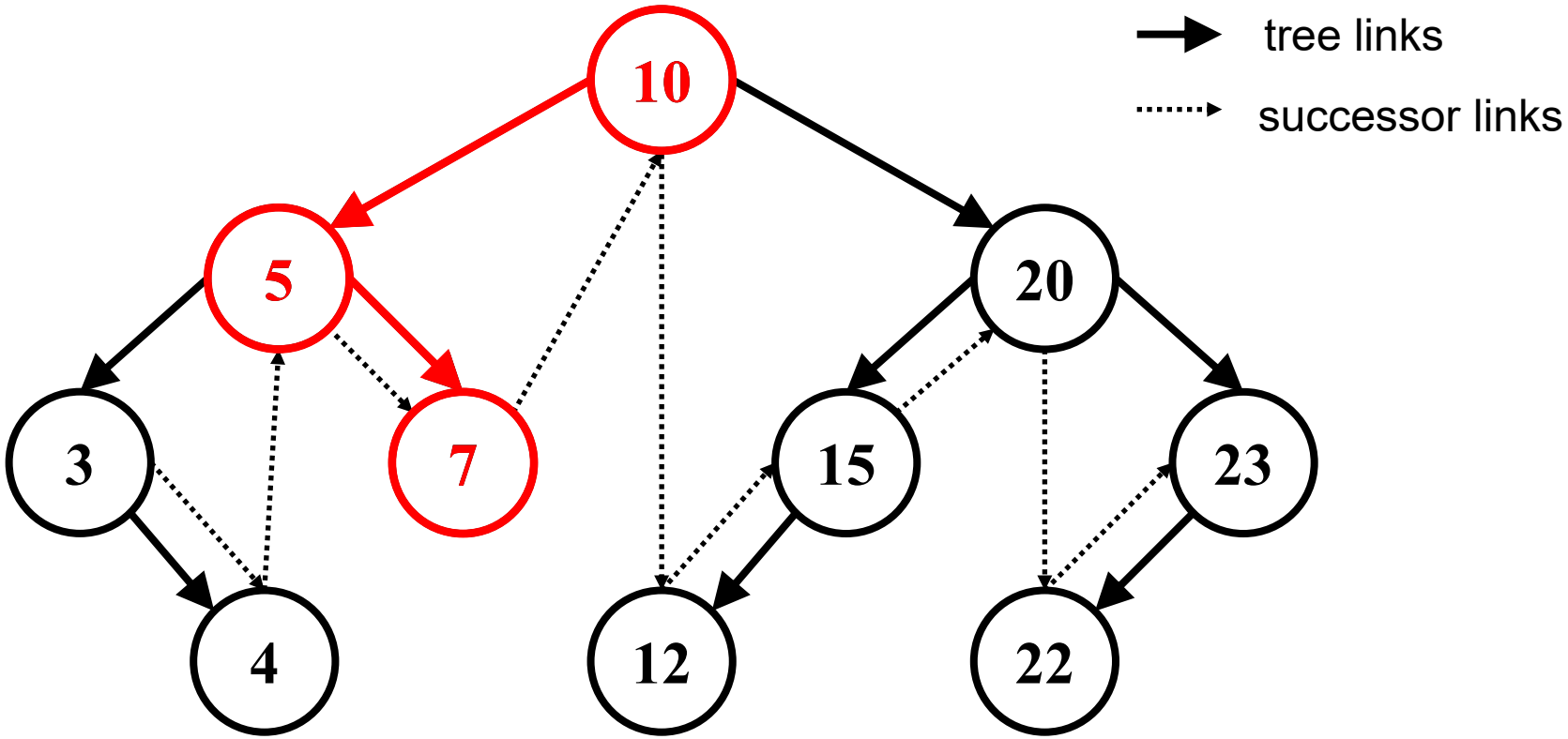
3.3.1 Iso-oriented segments

Algorithm 2: Find knot *p* with smallest key ≥ *y*

```
 1:   p = root; q = NULL;
 2:   while ((p != NULL) && (p.key != y)) {
 3:      q = p;
 4:      if (y < p.key) then  p = p.leftchild;
 5:      else                 p = p.rightchild;
 6:   }
 7:   if (p    != NULL) then return p;     // key==y found
 8:   if (root == NULL) then return NULL; // empty tree
 9:   if (q    == root) then     // tree has only one knot
         return NULL or root depending on root.key;
10:   if (y < q.key) then  return q;
11:   else                 return successor(q);
```

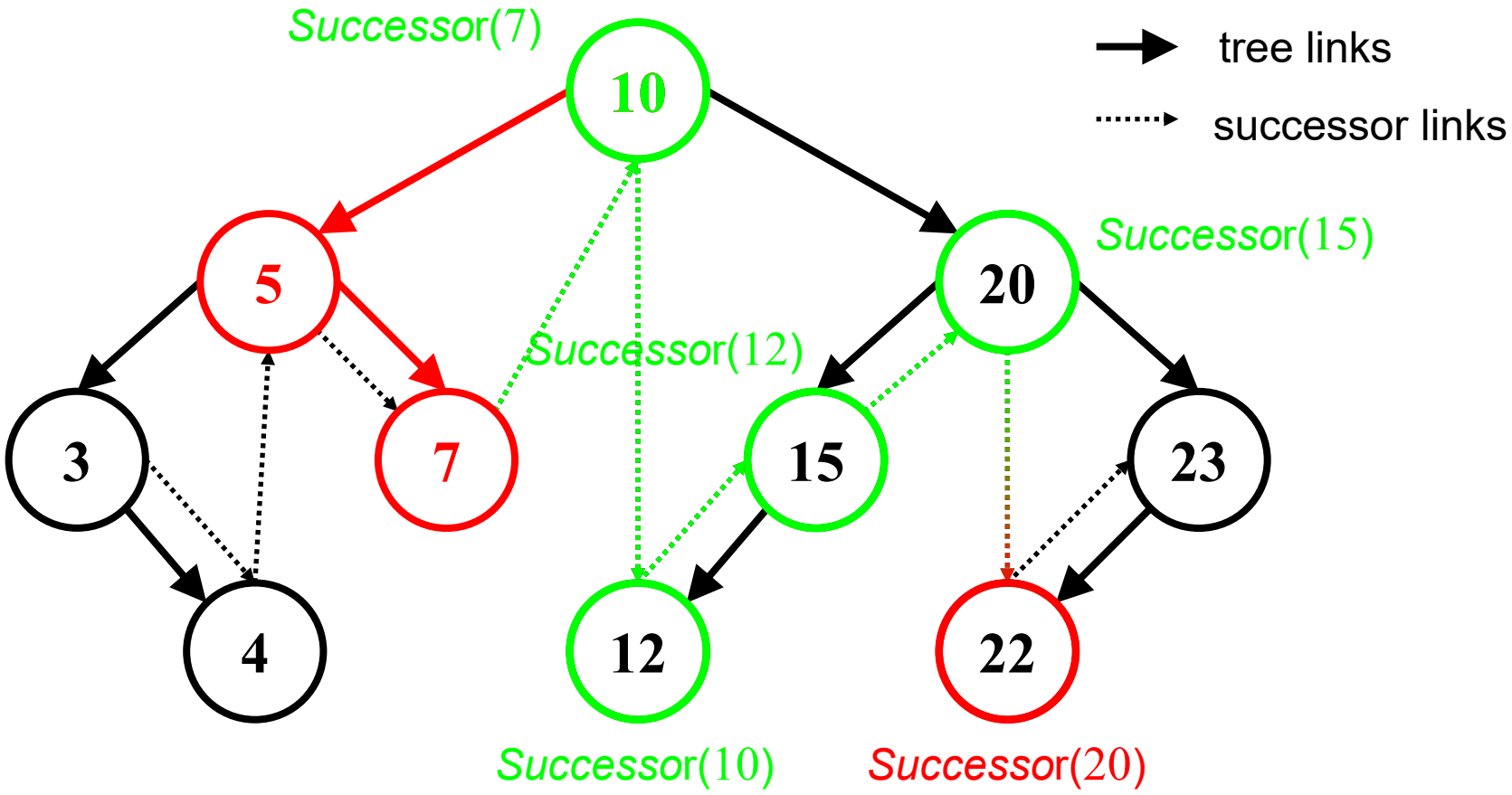
3.3.1 Iso-oriented segments

- **Example:** Search knots with keys in [8,21]



3.3.1 Iso-oriented segments

- **Example:** Search knots with keys in [8,21]



3.3.1 Iso-oriented segments

- **Run time**
  - Range search
    - Let r be the number of elements in $[y_l, y_u]$
    - **Search** for y_l: $O(\log n)$
    - **Traverse** the tree to knot $> y_u$: $O(r)$
    - **Total:** $O(r + \log n)$
  - Total scan-line run time
    - **Sorting** of the event points takes $O(n \log n)$.
    - **Scanning:** For every event point p_i we need $O(\log n + r_i)$, where r_i is the number of intersections at p_i.
    - **Total:** $O(n \log n + R)$, where $R = \sum r_i$ is the total number of intersections.

3.3.1 Iso-oriented segments

Remarks

- If R grows sub-quadratically, the scan-line algorithm is superior to the naïve approach!
- One can show, that in the worst case $\Omega(n \log n + R)$ steps are necessary to return all intersections.
- Scan-line-algorithms for iso-oriented line segments are *worst-case-optimal*.

Content

3.1 Motivation

3.2 Naïve approach

3.3 Scan-line approach

- **3.3.1** Iso-oriented segments
- **3.3.2** Arbitrary segments

3.4 Literature

Appendix: AVL trees (yet in German)

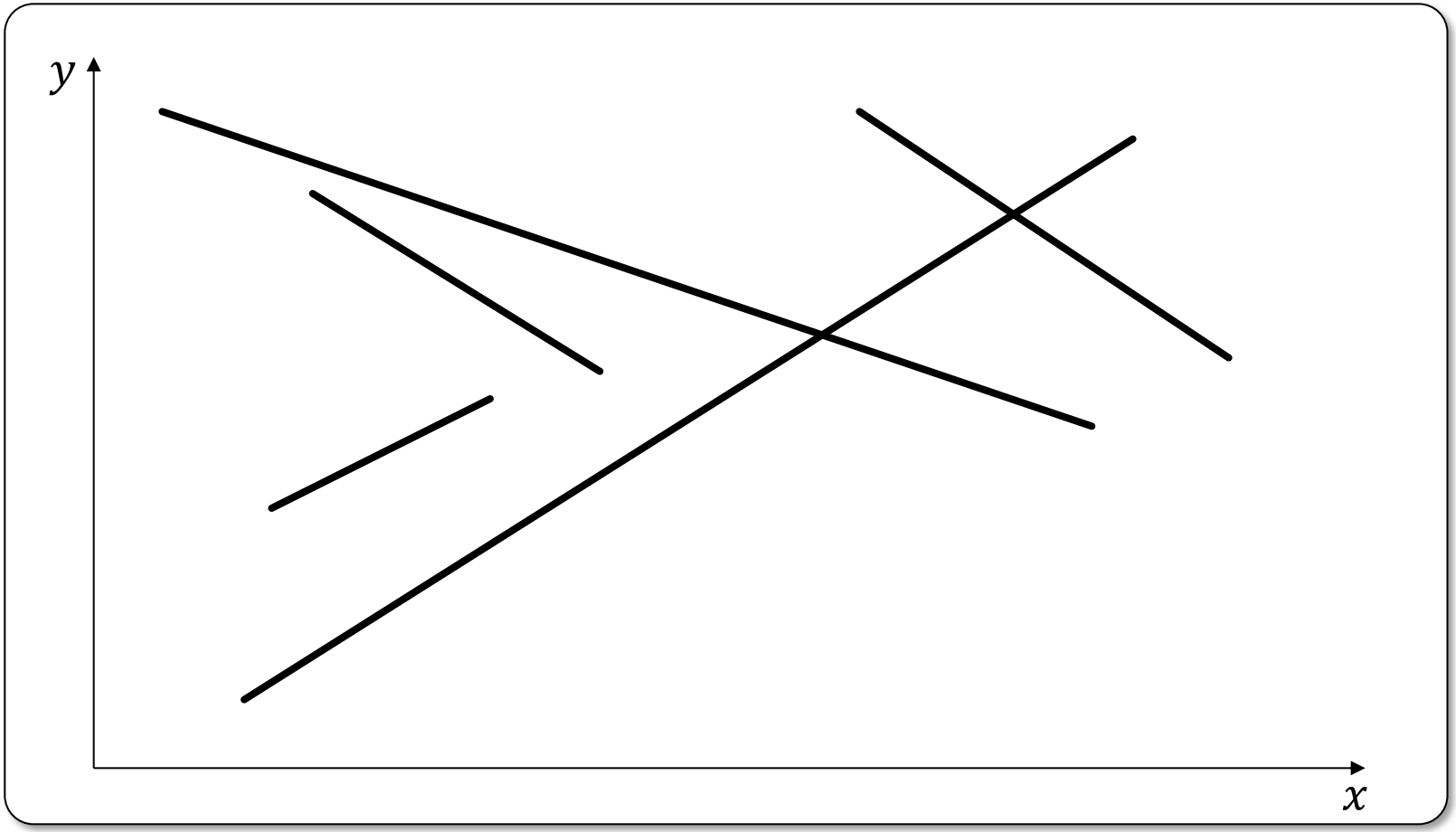
Appendix: Heaps (yet in German)

3.3.2 Arbitrary segments

- Now, arbitrary line segments in general position.
- **Input:** Set $S = \{s_1, \ldots, s_n\}$ of n line segments in 2D.
- **Goal:** Set of all *true intersections*, i.e. intersections, that are not end points.
- Assumption: *General position of the segments*
  - No segment is vertical.
  - The intersection of two segments is either empty or exactly one point.
  - In one point no more than two segments intersect.
  - All end- and intersection-points have mutually distinct x-coordinates.

3.3.2 Arbitrary segments

- **Example**



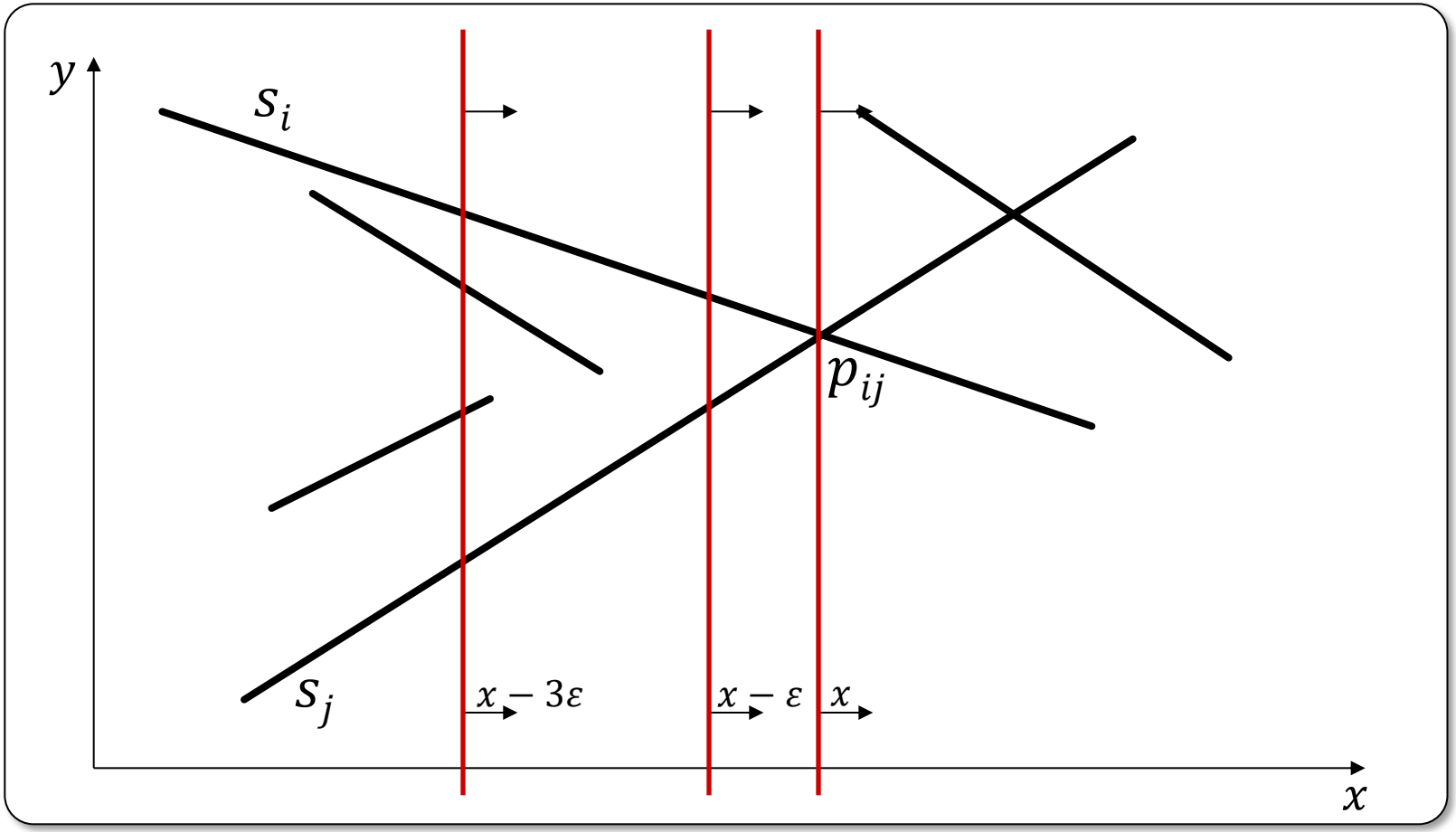
3.3.2 Arbitrary segments

- **Approach**
  - The principle is the same as for iso-oriented line segments.
  - Scan-line from left to right.

- **Observation 2**
  - Let s_i and s_j be two line segments, intersecting in a point $p_{ij} = (x, y)$.
  - Then s_i and s_j are adjacent on the scan-line, if the scan-line is at $x - \varepsilon$ for a sufficiently small and positive ε.

3.3.2 Arbitrary segments

- **Example**



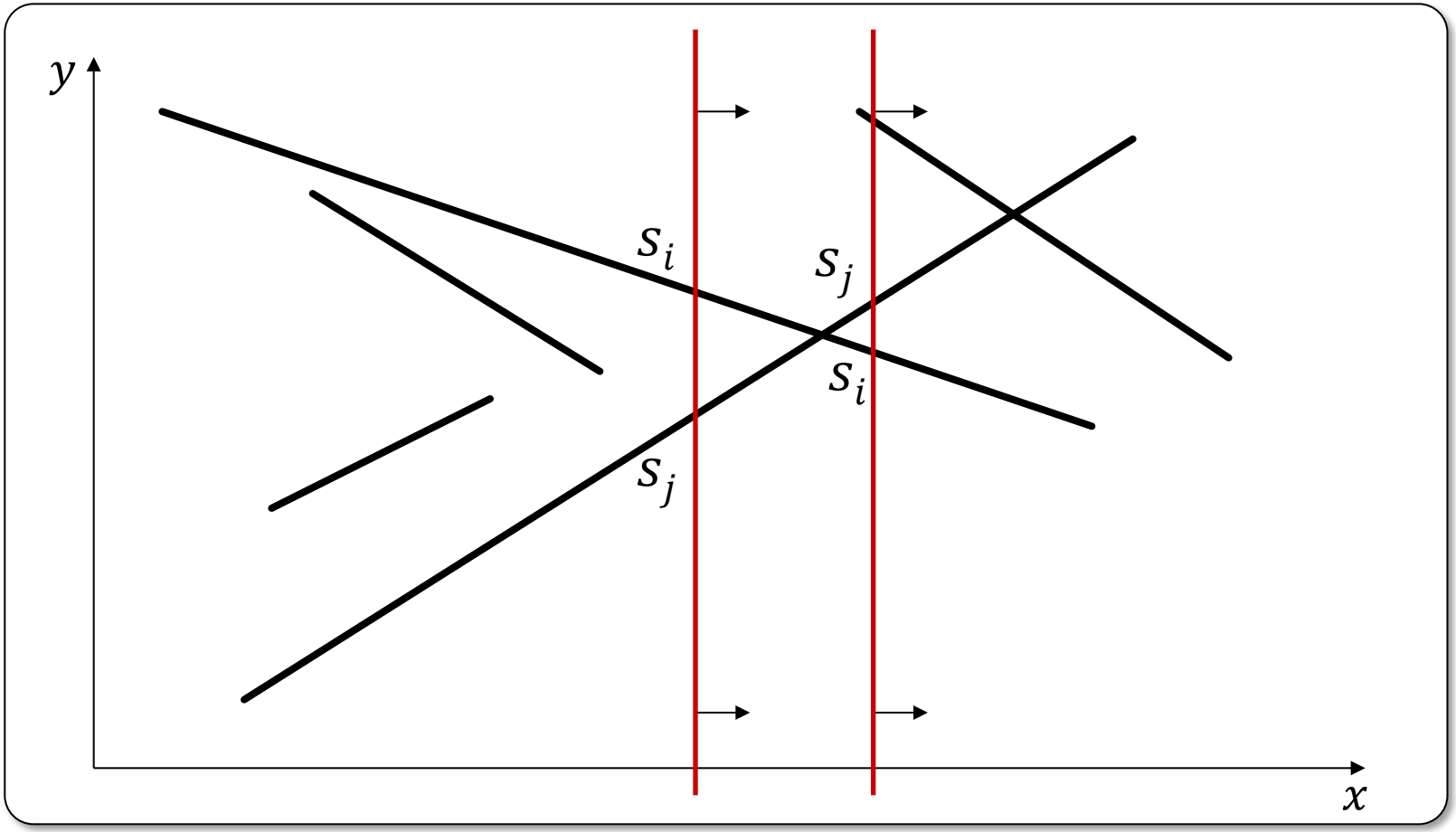
3.3.2 Arbitrary segments

- **Observation 3**
  - At an intersection of two line segments, the order of these two segments is swapped on the scan-line.

- **Consequence**
  - When an intersection is reached, the event structure needs to be updated to adjust the order of segments on the scan-line.
    - Also intersections are event points.
    - Intersections must be added to the event structure on runtime.

3.3.2 Arbitrary segments

- **Example**



3.3.2 Arbitrary segments

- There are three types of events:
  1. Start point of a segment is known a priori,
  2. End point of a segment is known a priori,
  3. Intersection point of two segments is computed on runtime.
- The data structure **`Event`** ***E*** represents one event and contains:
  - its x-coordinate,
  - identifiers of the corresponding segments:
    - one segment ID for start and end points,
    - two segment IDs for intersection points.

3.3.2 Arbitrary segments

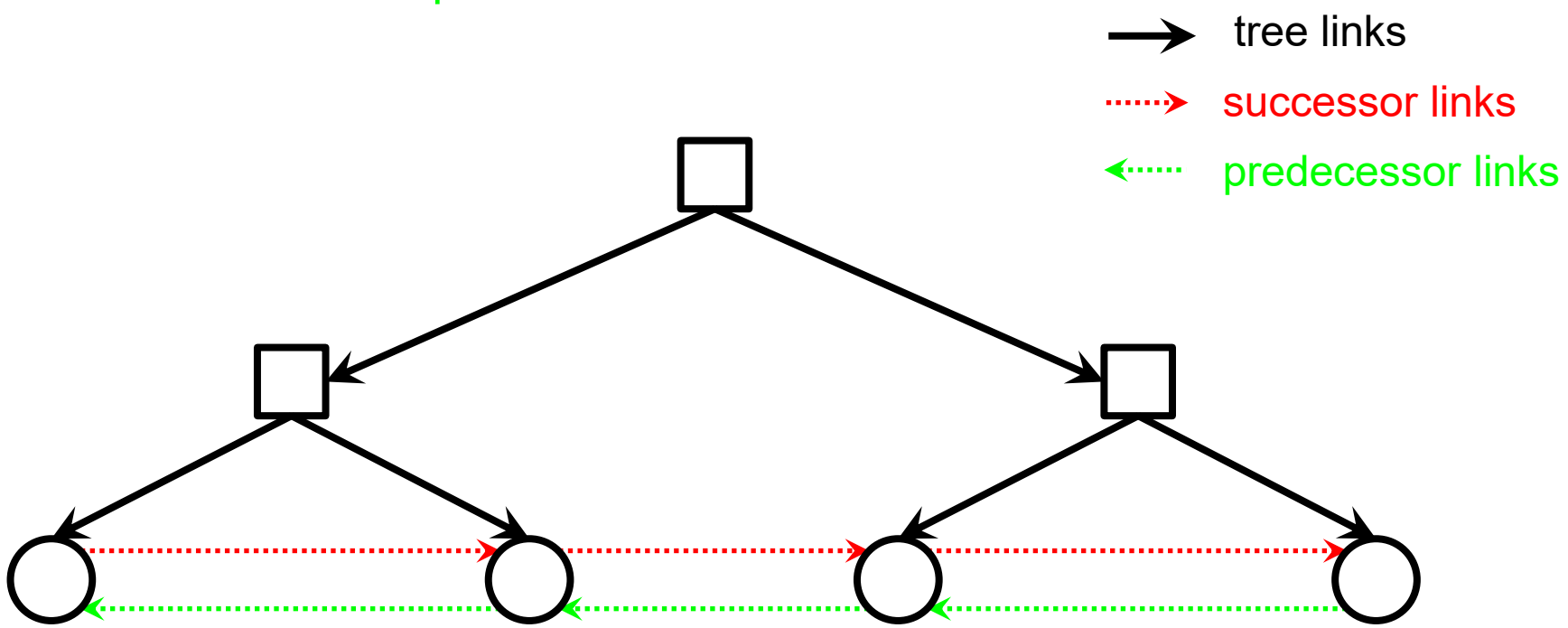
- Operations for the event data structure ***ES***, which maintains the events:
  - **`NextEvent():`** Returns event with smallest x-coordinate from ***ES*** and removes it.
  - **`AddEvent(Event E):`** Inserts a new event to ***ES*** according to its x-coordinate.
- Implementation of ***ES***:
  - Heap or balanced tree.
  - Run time of both operations: $O(\log|\boldsymbol{ES}|)$.

3.3.2 Arbitrary segments

- *Scan-Line-State-Structure* ***SSS***
  - Stores segments ordered by y-coordinate of the intersection of the active segments with the scan-line ***L***.
  - Operations
    - `AddSegment(Segment s)`: Adds segment ***s*** to ***SSS***.
    - `RemoveSegment(Segment s)`: Removes segment ***s*** from ***SSS***.
    - `Pred(Segment s)`: Returns predecessor of ***s*** on ***L***.
    - `Succ(Segment s)`: Returns successor of ***s*** on ***L***.
    - `Swap(Segment` s_1`,Segment` s_2`)`: Swaps two segments s_1 and s_2 on ***L***.

3.3.2 Arbitrary segments

- Implementation of ***sss***
  - Leaf-oriented, balanced, binary search tree with additional links for successors and predecessors.



3.3.2 Arbitrary segments

- Auxiliary function for intersection-events

  `TestIntersectionGenerateEvent(`s_1`,`s_2`):`

  - Takes two adjacent segments s_1 and s_2 from L.
  - Tests, if they intersect.
    - If yes, a new event is generated with x-coordinate of the intersection and ID's of the intersecting segments.
    - Otherwise, the empty event is returned.

3.3.2 Arbitrary segments

Algorithm 3: Intersect arbitrary line segments (Part 1)

Input: *n* line segments in general position.
Output: All intersections.

```
 1:  Initialize ES and SSS;
 2:  Sort the 2n end point by x-coordinate;
 3:  Store the resulting events in ES;
 4:  while (ES is not empty) {
 5:     E = ES.NextEvent();
 6:     if (E is start of a segment) then {
 7:        SSS.AddSegment(E.Segment);
 8:        PredS = SSS.Pred(E.Segment);
 9:        E'    = TestIntersectionGenerateEvent(E,PredS);
10:        if (E' ≠ ∅) then ES.AddEvent(E');
11:        SuccS = SSS.Succ(E.Segment);
12:        E'    = TestIntersectionGenerateEvent(E,SuccS);
13:        if (E' ≠ ∅) then ES.AddEvent(E');
14:     }
 ⋮
32:  }
```

3.3.2 Arbitrary segments

Algorithm 3: Intersect arbitrary line segments (Part 2)

Input: *n* line segments in general position.
Output: All intersections.

```
 ⋮
 ⋮
 4:  while (ES is not empty) {
 5:     E = ES.NextEvent();
 ⋮
15:     if (E is end of a segment) then {
16:        PredS = SSS.Pred(E.Segment);
17:        SuccS = SSS.Succ(E.Segment);
18:        SSS.Remove(E.Segment);
19:        E' = TestIntersectionGenerateEvent(PredS,SuccS);
20:        if (E' ≠ ∅) then ES.AddEvent(E');
21:     }
 ⋮
 ⋮
 ⋮
32:  }
```

3.3.2 Arbitrary segments

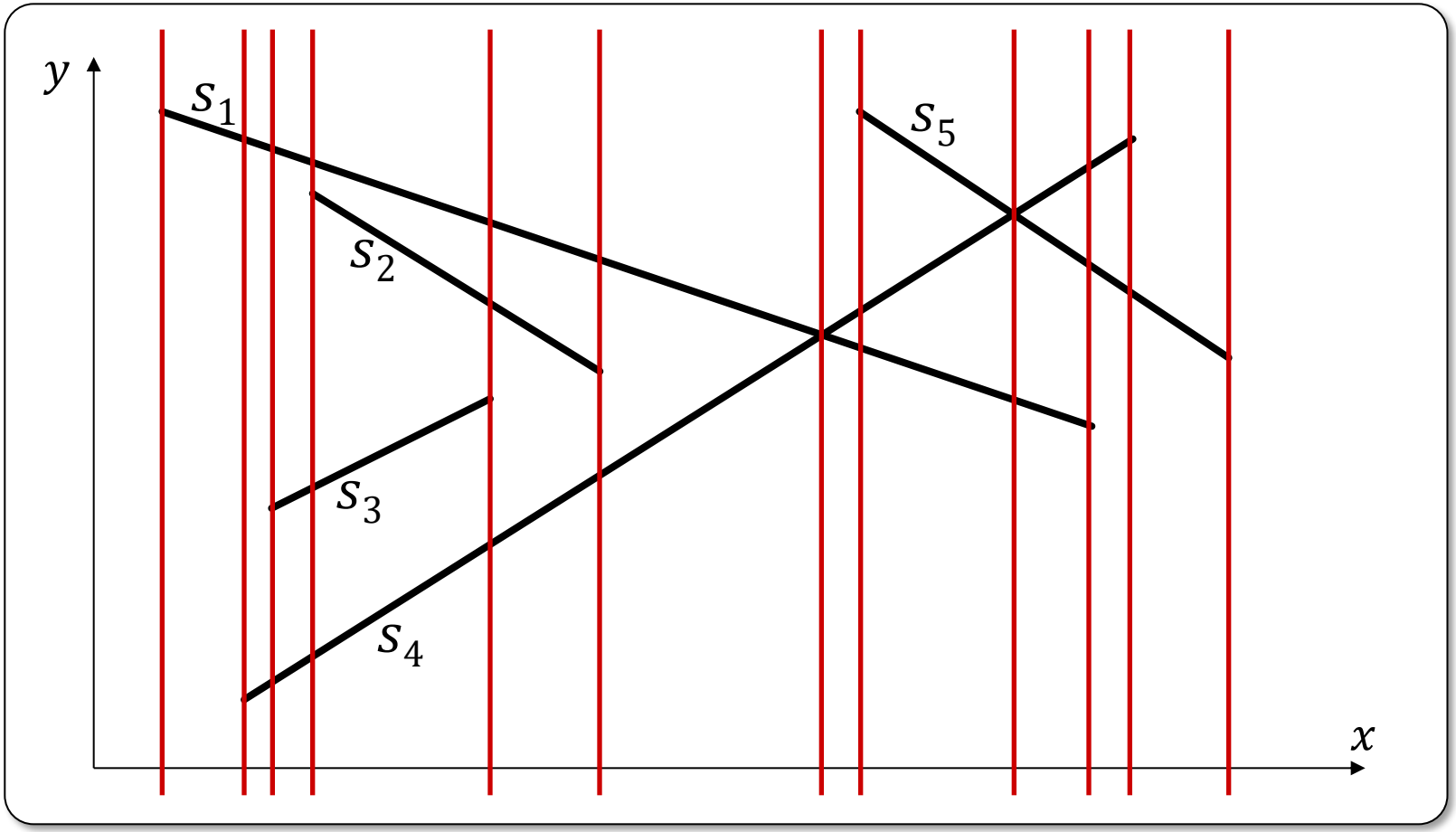
Algorithm 3: Intersect arbitrary line segments (Part 3)

Input: *n* line segments in general position.
Output: All intersections.

```
 ⋮
 ⋮
 4:  while (ES is not empty) {
 5:     E = ES.NextEvent();
 ⋮
22:     if (E is intersection) then {
23:        Report intersection of E.SegUp and E.SegLow;
24:        SSS.Swap(E.SegUp,E.SegLow);
25:        PredS = SSS.Pred(E.SegUp);
26:        E' = TestIntersectionGenerateEvent(E.SegUp,PredS);
27:        if (E' ≠ ∅) then ES.AddEvent(E');
28:        SuccS = SSS.Succ(E.SegLow);
29:        E' = TestIntersectionGenerateEvent(E.SegLow,SuccS);
30:        if (E' ≠ ∅) then ES.AddEvent(E');
31:     }
32:  }
```

3.3.2 Arbitrary segments

- **Example**



3.3.2 Arbitrary segments

- **Run time**
  - Sorting of the $2n$ end points by x-coordinate: $O(n \log n)$.
  - Total number of events: $2n + k$ (k is the number of intersections).
    - The outer loop is repeated at most $O(2n + k)$ times.
    - Per iteration there are at most six operations on ***ES*** and ***SSS***.
  - **Runtime of *ES*-Operations:** There are at most $O(n^2)$ events at the same time in ***ES***, because there are at most $O(n^2)$ intersections.
    - Every operation on ***ES*** takes $O\big(\log(n^2)\big) = O(\log n)$.
  - **Runtime of *SSS*-Operations:** There are at most n elements in ***SSS***.
    - Every operation on ***SSS*** takes at most $O(\log n)$.

3.3.2 Arbitrary segments

- **Total run time**
  - $O\big((n+k)\log n\big)$
- **Memory**
  - $O\big(n^2\big)$, because there are at most $O\big(n^2\big)$ intersection in ***ES***.

3.4 Literature

[1] Aho, Hopcroft, Ullman: *Data structures and algorithms*, Addison-Wesley, 1982.

[2] Ottmann, Widmayer: *Algorithmen und Datenstrukturen*, Spektrum akademischer Verlag, 2. Auflage, 1996.

[3] Bentley, Ottmann: *Algorithms for reporting and counting geometric intersections*. IEEE Trans. Comput., C-28: 643-647, 1979.

Appendix: AVL trees

Definition

A search tree is a **balanced search tree**, if it has a maximal height of $O(\log n)$.

- All dictionary-operations can be executed in $O(\log n)$.

Definition

A binary search tree is called **AVL tree**, if for every knot k the heights of the two sub-trees differs by at most 1.

- An AVL tree has maximal height of ca. $1.5 \log_2 n$. [Ottmann/Widmayer].
- The acronym goes back to **A**delson-**V**elskij and **L**andis.

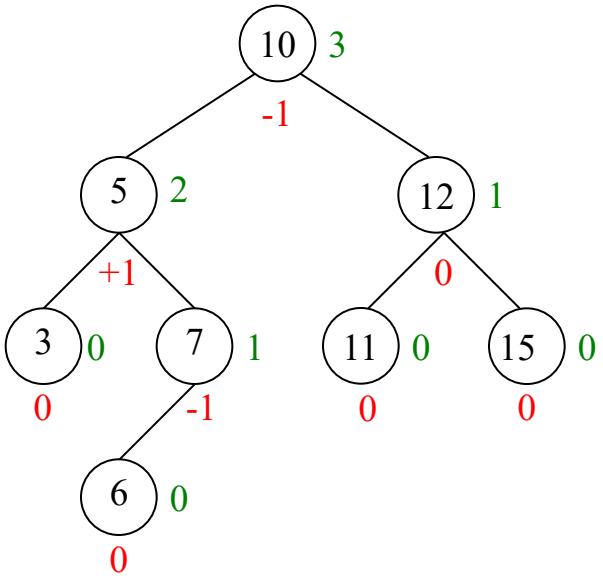
Appendix: AVL trees

Examples:

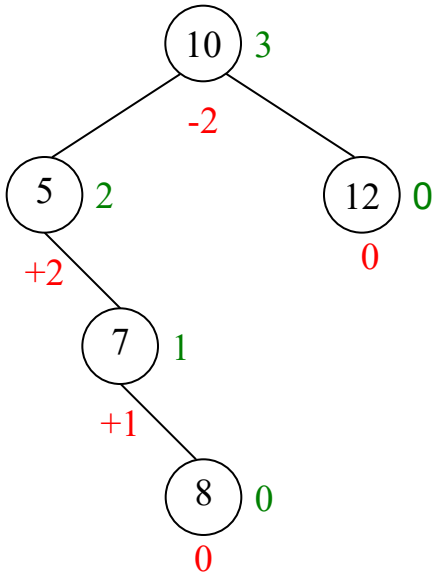
height of a sub-tree

height difference = height of right sub-tree – height of left sub-tree

height of an empty sub-tree is −1.



AVL tree



Not AVL tree

Appendix: AVL trees

Properties of AVL trees:

An unbalanced tree, with

- height difference of
  - −2 (left heavy) or
  - +2 (right heavy) and
- whose other sub-trees are balanced,

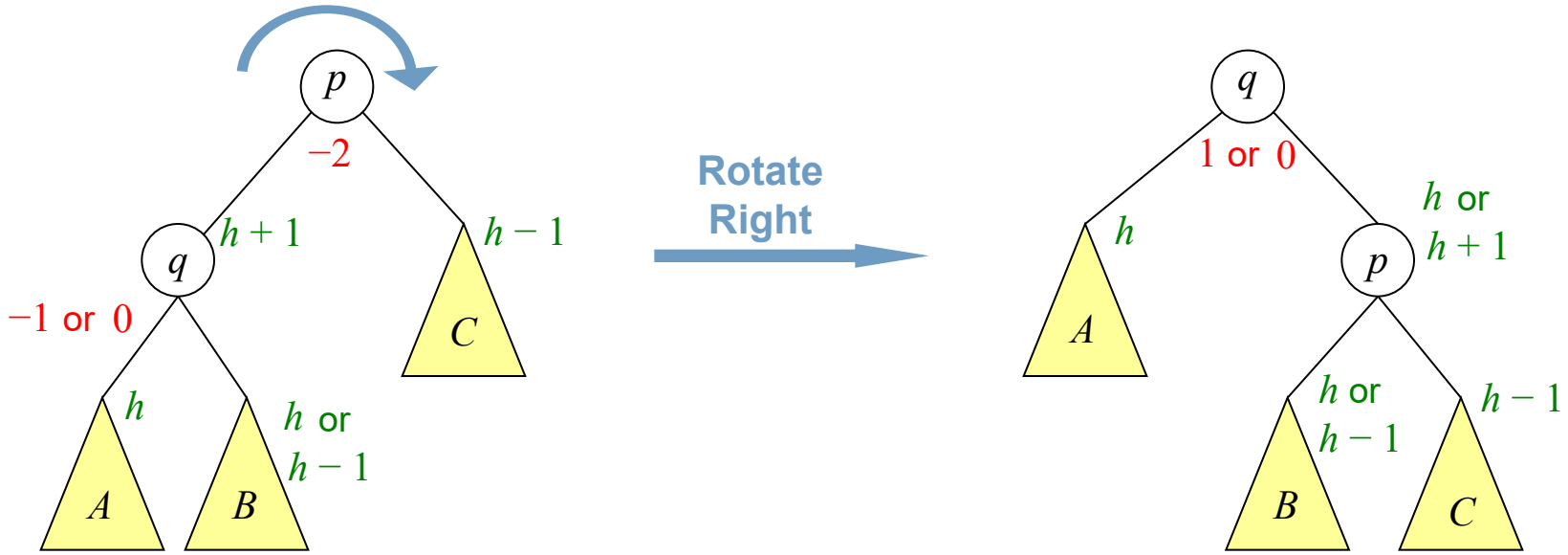
can be re-balanced by so-called **rotation operations**.

- There are four different cases…

Appendix: AVL trees

Case A: The tree is left heavy, i.e., the height difference is −3.

Case A1: The left sub-tree has height difference −1 or 0.

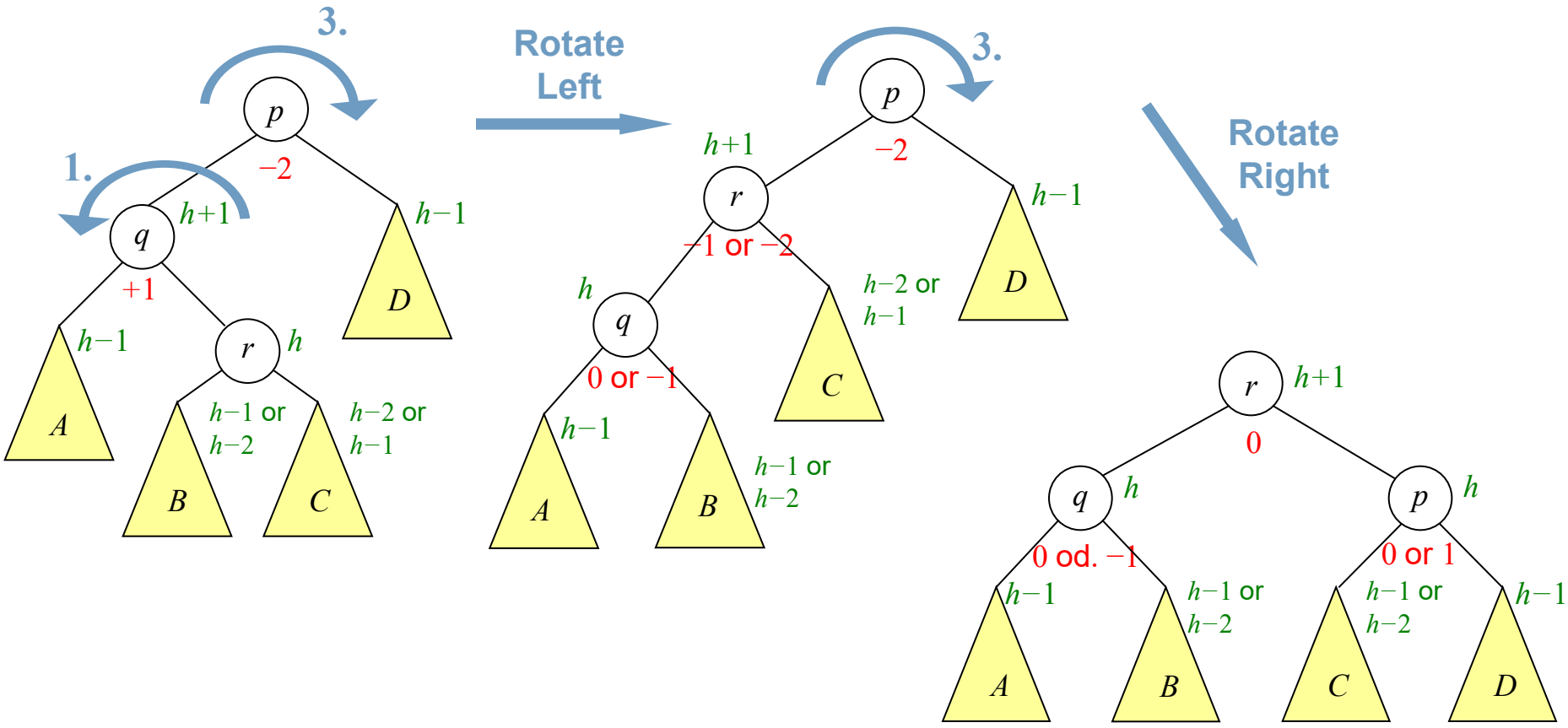


Sub-trees A, B and C are balanced and might also be empty.

Appendix: AVL trees

Case A: The tree is left heavy, i.e., the height difference is −3.

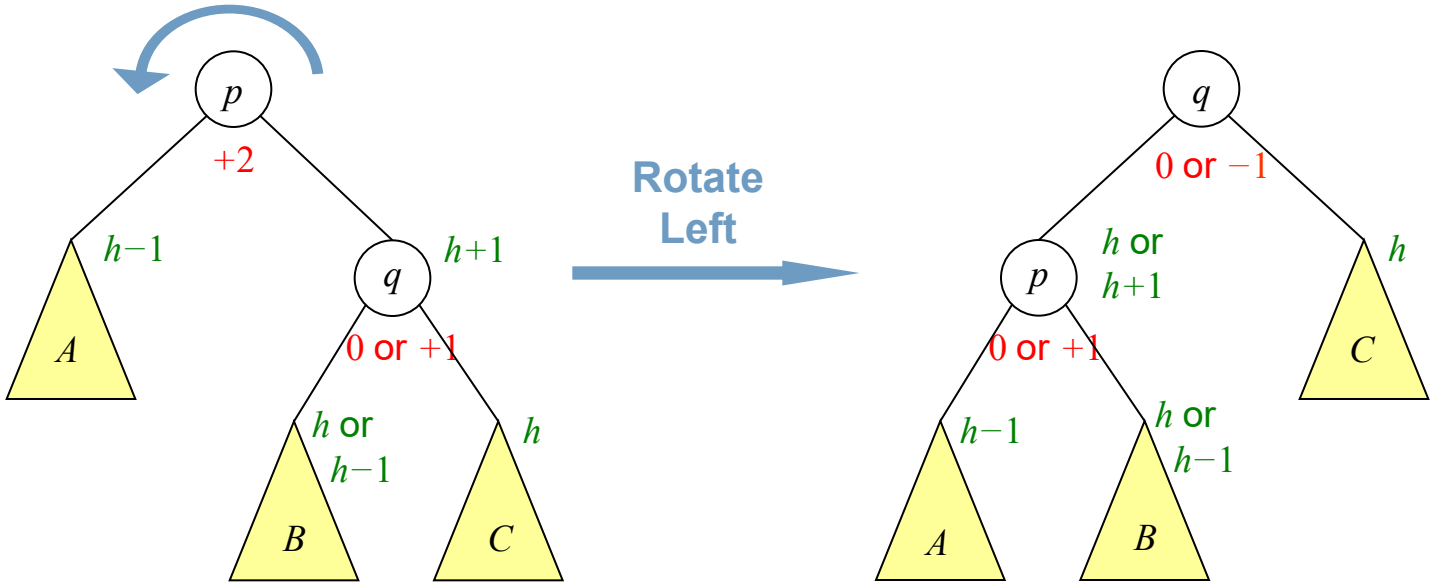
Case A2: The left sub-tree has height difference +1.



Appendix: AVL trees

Case B: The tree is right heavy, i.e., the height difference is +3.

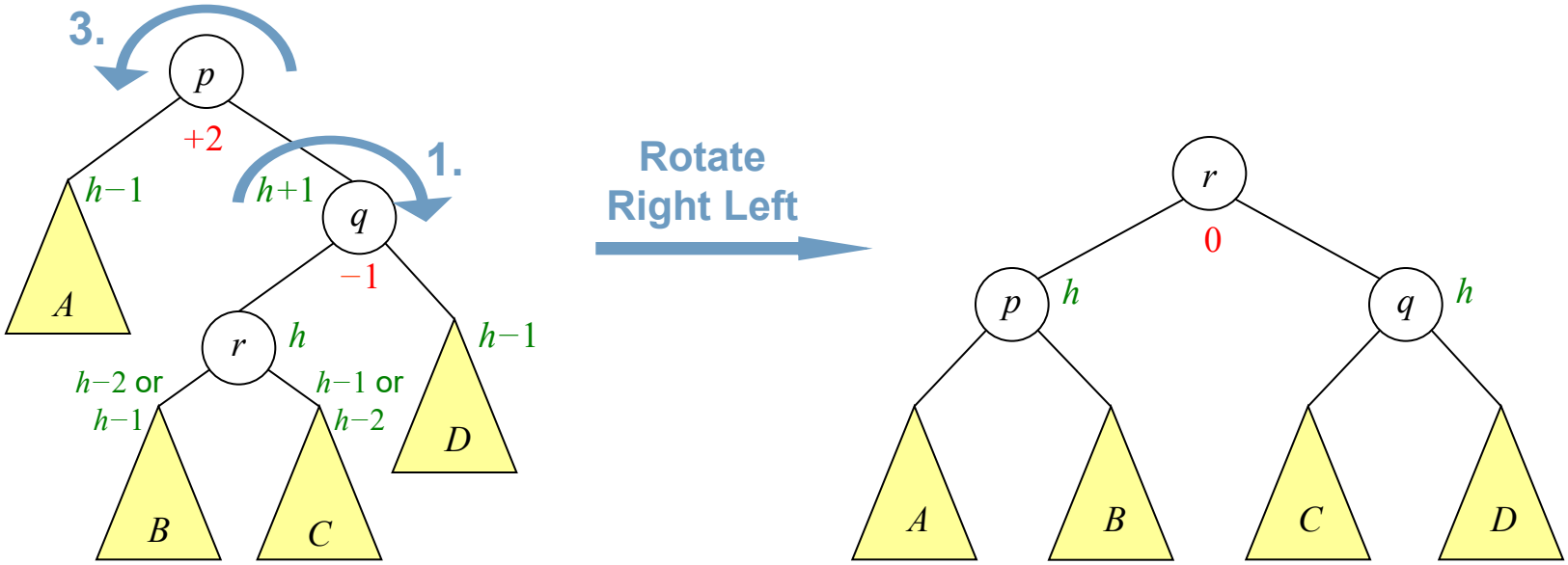
Case B1: The right sub-tree has height difference 0 or +1.



Appendix: AVL trees

Case B: The tree is right heavy, i.e., the height difference is +2

Case B2: The right sub-tree has height difference −1.



Appendix: AVL trees

Principle

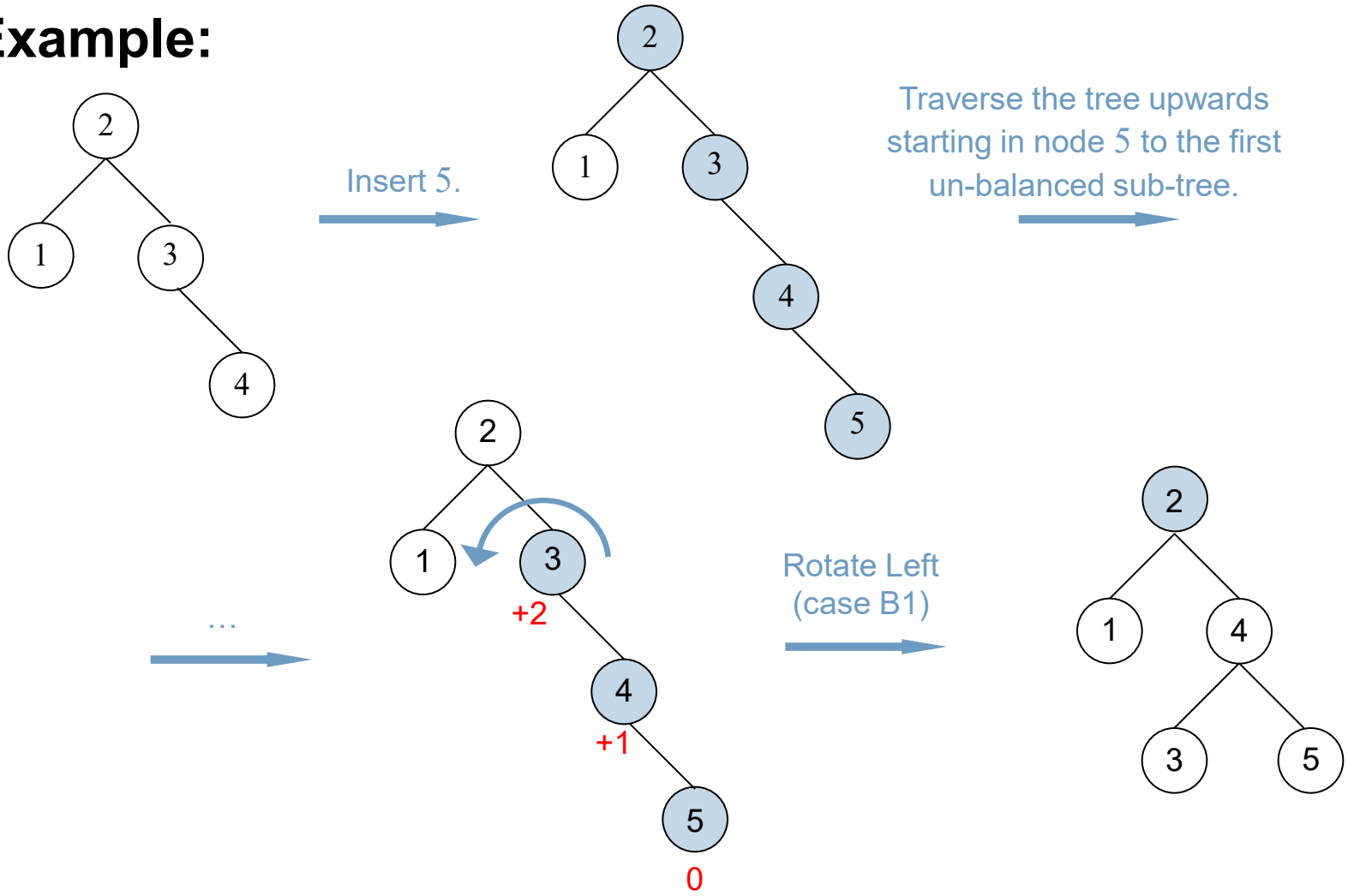
1. Insert and remove elements from the tree as for binary search trees.
2. Traverse the tree from the inserted or removed node upwards to the root and re-balance the visited sub-trees in necessary using the respective rotation operations.

Remark:

- Because the tree was balanced before the insertion or removal operations, all sub-trees had a height difference of -1, 0 or $+1$.
  - Thus, after step 1. only the afore discussed cases A1, A2, B1 or B2 can happen.
  - For an insertion, there is at most one rotation necessary in step 3.
  - For a removal, multiple rotations might be necessary in step 3. along the path in the tree upwards to the root.

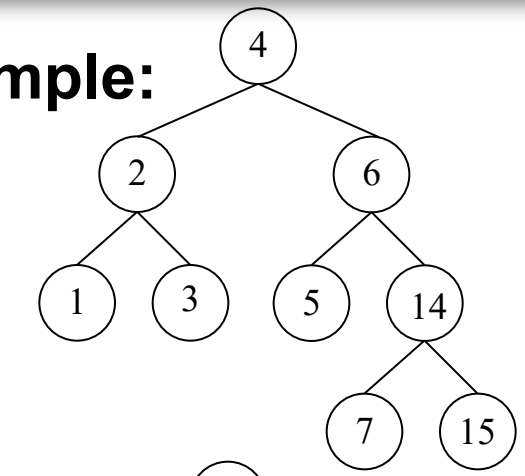
Appendix: AVL trees

Example:

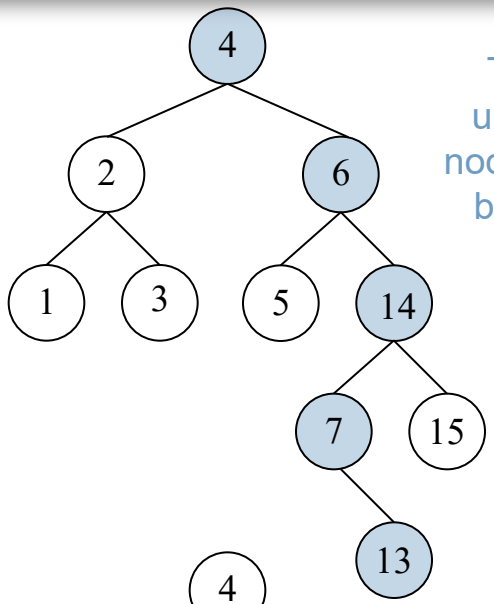


Appendix: AVL trees

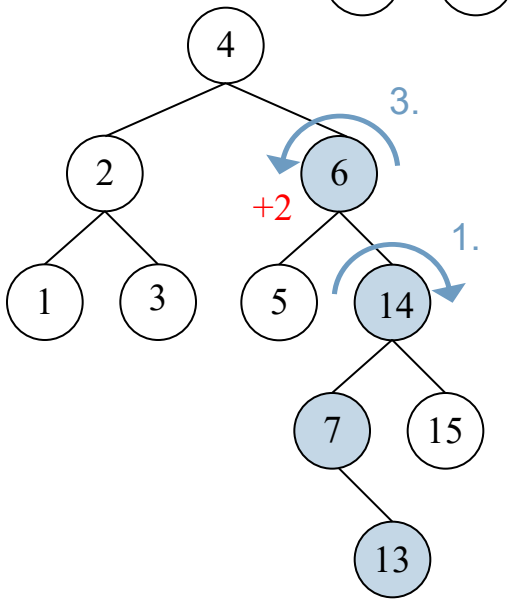
Example:



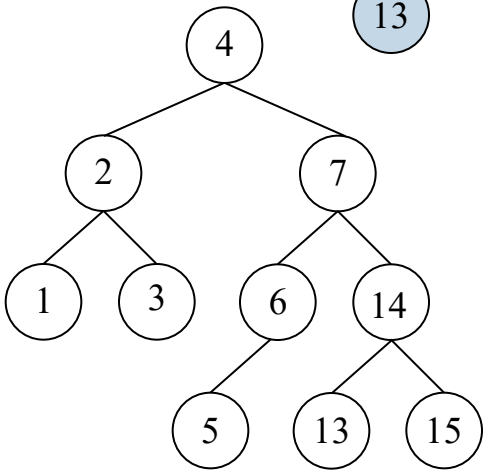
Insert 13.



Traverse the tree upwards starting in node 13 to the first un-balanced sub-tree.



Rotate Right Left (case B2)



Appendix: AVL trees

Data structure

- The same as for binary search-trees.
- Additionally, every node stores the height of its sub-tree.

```
struct Node
{
   int        height;
   KeyType    key;
   ValueType  val;
   Node*      left;
   Node*      right;
};
```

Algorithms

- Extend the **`insert`**-operation for binary search-trees to re-balance the tree, see next page.
- Extend the **`remove`**-operation for binary search-trees analogously.
- The **`search`**-operation for binary search-trees does not need to be extended.

Appendix: AVL trees

```
bool insert(KeyType key, ValueType& v, Node*& p)
{
   bool result;
   if (p == 0) {// key does not exist in the tree
      p           = new Node;
      p->key      = key;
      p->val      = v;
      p->height   = 0;
      result      = true;
   }
   else if ( key < p->key ) result = insert(key,v,p->left);
   else if ( key > p->key ) result = insert(key,v,p->right);
   else                     result = false; // key exists
   if (result)              balance(p);
   return result;
}
```

A leaf has height 0.

Knot has been inserted: Update the height ans re-balance if necessary.

Appendix: AVL trees

```
void balance(Node*& p)
{
  if (p == 0) return;

  // Update the height.
  p->height = max(getHeight(p->left),getHeight(p->right))+1;

  if    ( getBalance(p)==-2 )          // Tree is left heavy.
    if ( getBalance(p->left) <=0 ) rotateRight(p);     // A1
    else                           rotateLeftRight(p);// A2
  else if ( getBalance(p)==+2 )      // Tree is right heavy.
    if ( getBalance(p->right)>=0 ) rotateLeft(p);      // B1
    else                           rotateRightLeft(p);// B2
}
```

Appendix: AVL trees

Auxiliary frunctions:

```
int getHeight(const Node* p)
{
   if (p == 0) return -1;// Empty tree has height -1
   else        return p->height;
}
```

```
int getBalance(const Node* p)
{
   if (p == 0) return 0;// Empty tree has height difference 0
   else        return getHeight(p->right)-getHeight(p->left);
}
```

Run-time: $O(1)$.

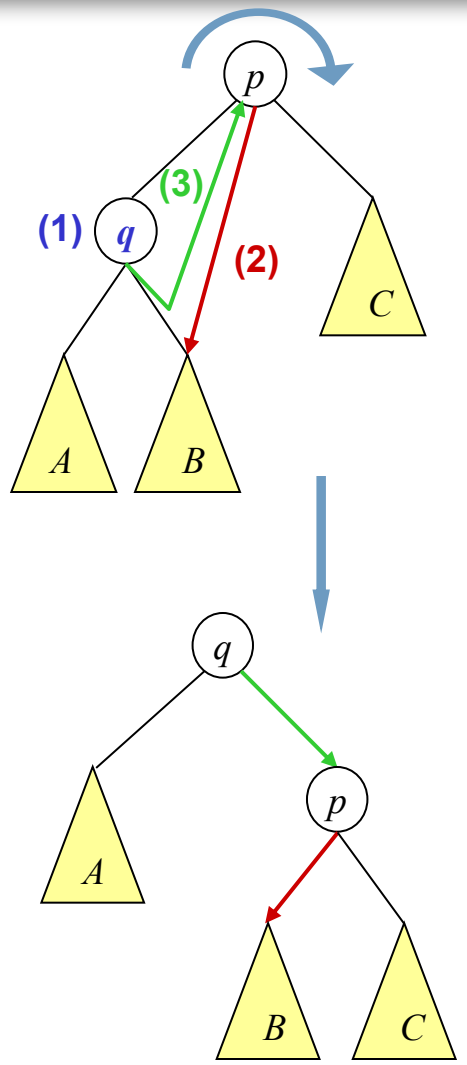
Appendix: AVL trees

Rotations:

```
void rotateRight(Node*& p)
{
   // Vor.: p->left!=0, sonst wäre p nicht
   //       linkslastig unbalanciert.

   Node* q   = p->left;     // (1)
   p->left   = q->right;    // (2)
   q->right  = p;           // (3)

   p->height = max(getHeight(p->left),
                   getHeight(p->right))+1;
   q->height = max(getHeight(q->left),
                   getHeight(q->right))+1;
   p         = q;
}
```



Appendix: AVL trees

Rotations:

```
void rotateLeft(Node*& p)
{  // Pre-condition: p->right!=0, ...
   // analog to rotateRight
}
```

```
void rotateLeftRight(Node*& p)
{  // Pre-condition: p->left!=0, ...
   rotateLeft (p->left);
   rotateRight(p);
}
```

```
void rotateRightLeft(Node*& p)
{  // Pre-condition: p->right!=0, ...
   rotateRight(p->right);
   rotateLeft (p);
}
```

Appendix: AVL trees

Run-time

- **`search`**: $O(\log n)$,
  because an AVL tree is balanced.

- **`rotateRight`**, **`rotateLeft`**,
  `rotateLeftRight`, **`rotateRigthLeft`**: $O(1)$.
- **`balance`**: $O(1)$.
- **`insert`**: $O(\log n)$.
- **`remove`**: $O(\log n)$.

Appendix: Heaps

- In a binary search-tree every node **k** satisfies the relation

  k.left.key < k.key < k.right.key .

  This property facilitates fast search queries for arbitrary keys.

- For the search for the largest key, the following relation in every node **k** is better

  k.key > k.left.key, k.right.key .

  This property is called **(max)-heap-property**.

- The max-heap-property implies that the node with the largest key is the root of the tree.
- Analog, min-heap-property.

Appendix: Heaps

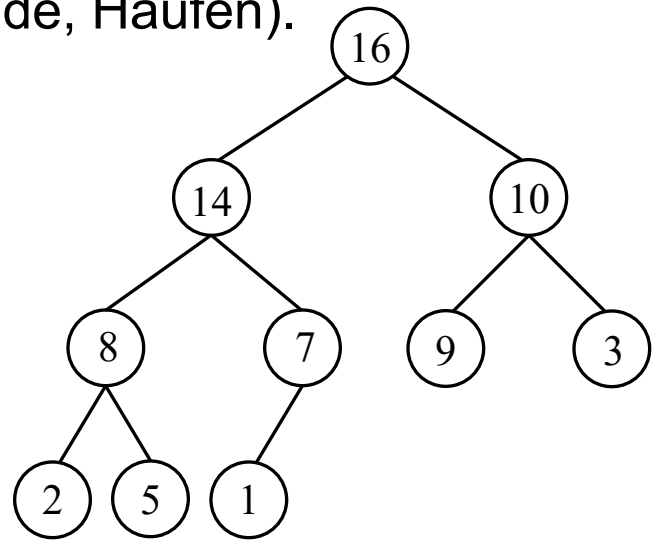
Definition

A binary tree,

1. which satisfies for every node the max-heap-property,
2. for which all levels are filled completely (except the last level) and
3. for which the last level is left-aligned,

is called a **heap** (dt.: Halde, Haufen).

Example



Appendix: Heaps

Properties

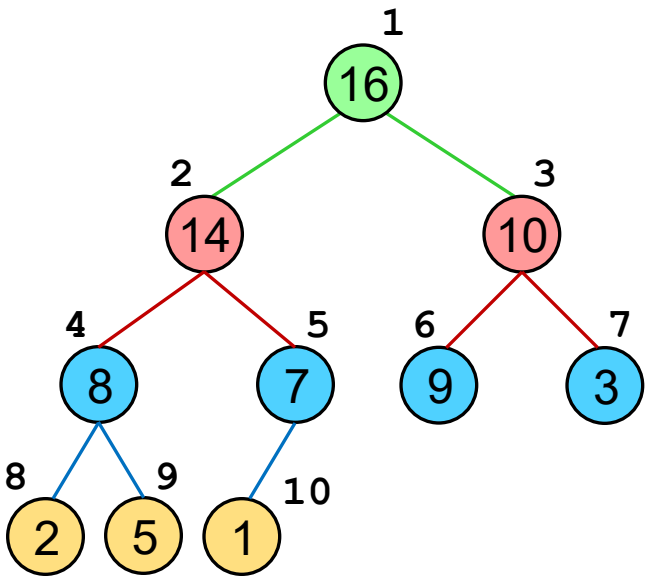
- The root of a max-heap is the node with the largest key.
- The root of a min-heap is the node with the smallest key.
- Because a heap is a complete binary tree, the height of a heap with n Elements is $\Theta(\log n)$.

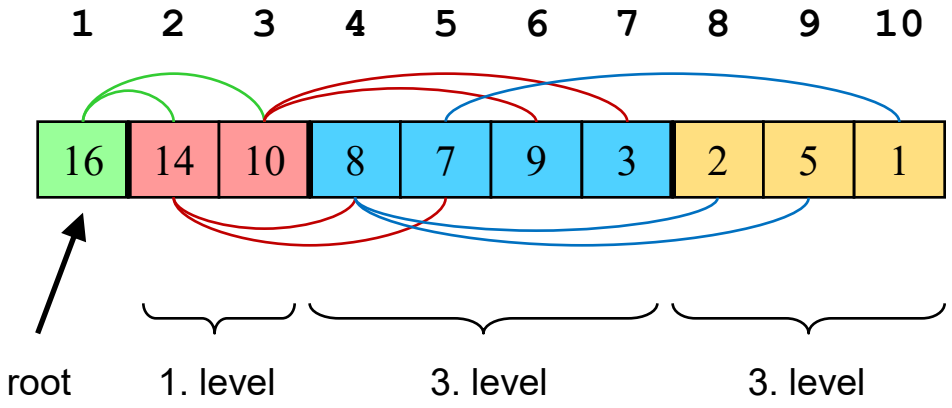
Appendix: Heaps

Remark

- A heap can be realized in-situ with an array **`a[1..n]`**.
- Then, the heap-property has the form

```
a[i].key > a[2i].key, a[2i+1].key
```





Appendix: Heaps

Construction of a heaps

- Store the n elements in an arbitrary order in an array **`a[1..n]`**.
  - This corresponds to an arbitrary order of elements in a binary tree, that satisfies the properties 3. and 3.
- Subsequently, the heap-property must be established for all nodes.
  - Every leaf is already a one-element heap.
  - The sub-array **`a[(⌊n/2⌋+1)..n]`** contains only leaves.
  - The heap-property will be established from the leaves up to the root (aka bottom-up) using the auxiliary function **`heapify`**.

```
void build_heap(Node[] a)
{
   for (int i=floor(length(a)/2); i>0; i--)
      heapify(a,i);
}
```

Appendix: Heaps

- The auxiliary function **`heapify`** is applied in the tree from the leaves to the root.
  - Thus, in a current node the two child-nodes satisfy already the heap-property.
  - If the current node's key is larger than the key of both child-nodes, it also satisfies the heap-property.
  - Otherwise, the current node
    a. is swapped with the child-node, that has the larger key, and
    b. sinks in the respective sub-tree until both child-nodes have smaller keys.

Appendix: Heaps

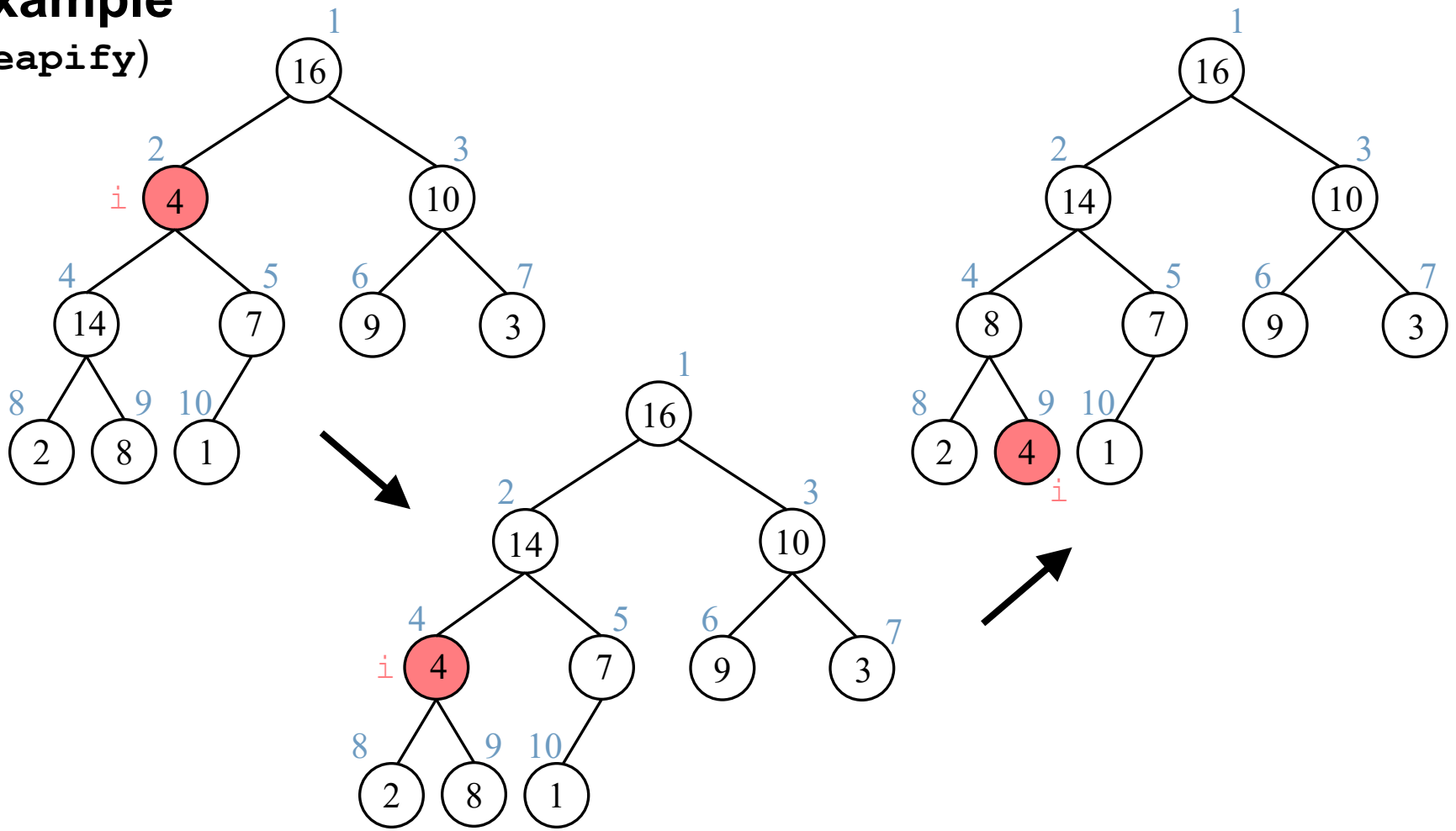
heapify

```
void heapify(Node[] a, int i)
{
   int max = i;
   int li  = 2*i;
   int re  = li+1;
   if (a[li].key > a[max].key) max = li;
   if (a[re].key > a[max].key) max = re;
   if (i != max) {
      swap(a[i],a[max]);
      heapify(a,max);
   }
}
```

```
// Attention: Here one
   should test, if li
   and re are valid
   indices!
```

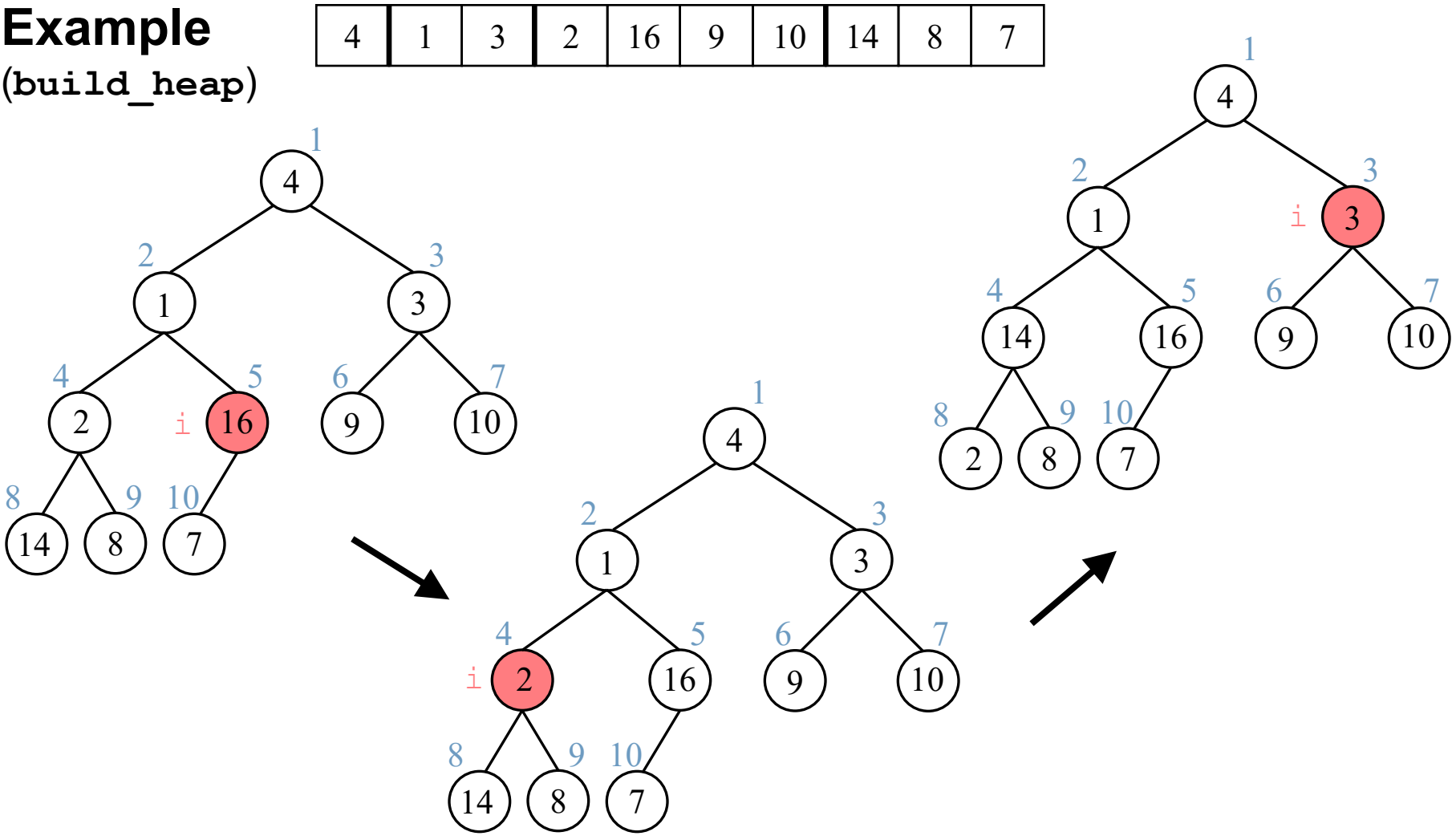
Appendix: Heaps

Example
(`heapify`)



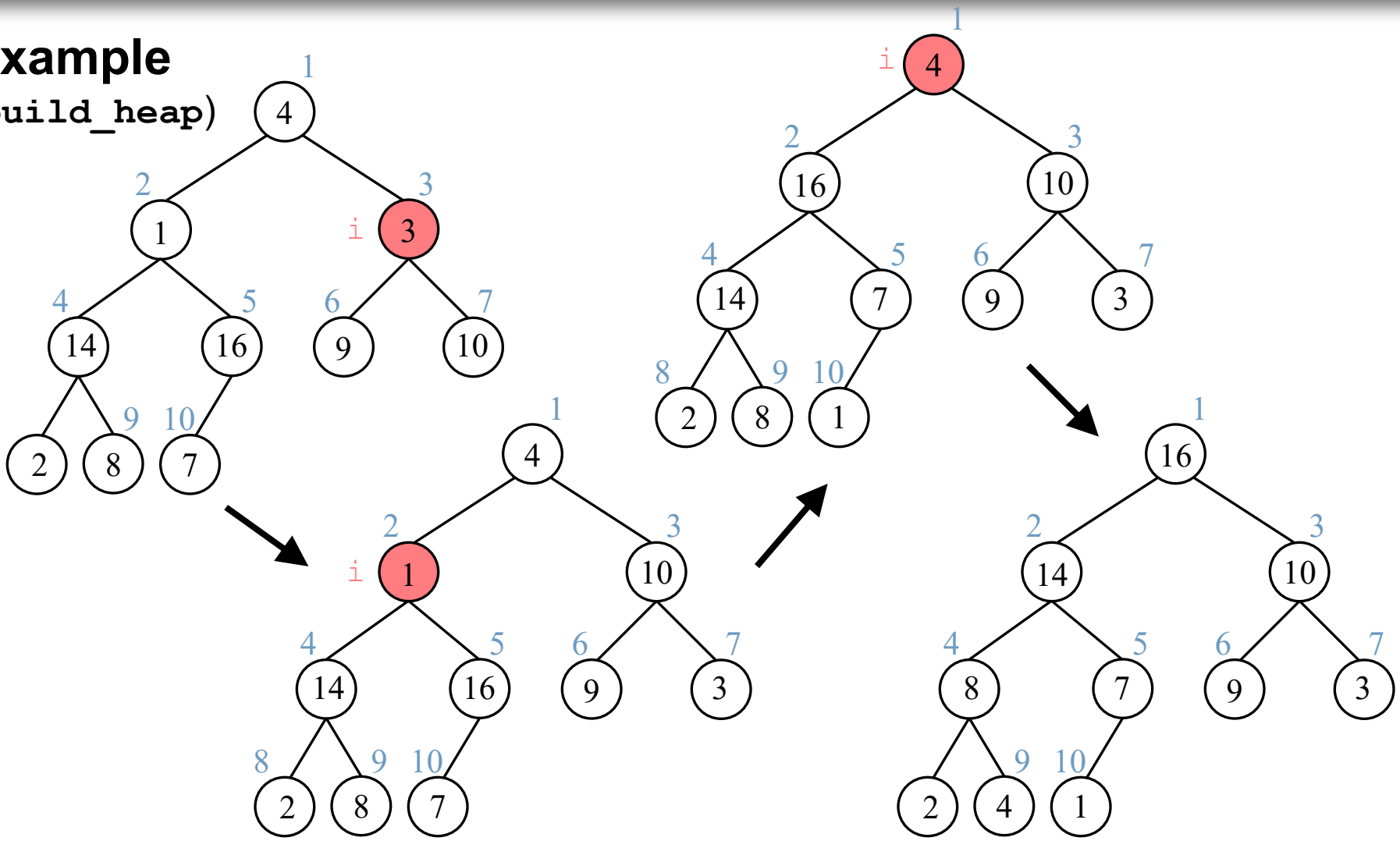
Appendix: Heaps

Example
(`build_heap`)



Appendix: Heaps

Example
(`build_heap`)



Appendix: Heaps

Runtime of `heapify`

- Runtime of **`heapify`** for a sub-tree of size n.
  - Determine the relation of **`a[i], a[le], a[re]`** costs $\Theta(1)$.
  - Application of **`heapify`** to one sub-tree.
    - The left sub-tree has maximal size of $2n/3$, if the last level is exactly half-full.
  - Runtime of **`heapify`**: $T(n) \leq T(2n/3) + \Theta(1)$ (use Master-Thm)
    - ➡ $T(n) = O(\log n)$.
  - Alternatively: Runtime of **`heapify`** for a node of height h is $O(h)$.

Appendix: Heaps

Runtime of `build_heap`

- Runtime of **`build_heap`** for an array of size n.
  - Upper bound: $O(n)$ calls to **`heapify`**
    - ➡ $O(n \log n)$.
  - Tight upper bound: The runtime of **`heapify`** varies with the height of the sub-tees and is mostly small.
    - A n-element heap has height $\lfloor \log n \rfloor$.
    - There are at most $\lceil n/2^{h+1} \rceil$ nodes of height h.
    - Total runtime:

$$O(h) \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}} \right) = O\left(n \underbrace{\sum_{h=0}^{\infty} \frac{h}{2^{h+1}}}_{=2} \right) = O(n).$$