universität wien

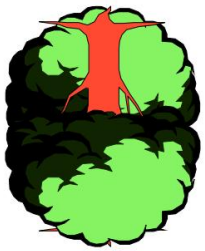
Chapter 5

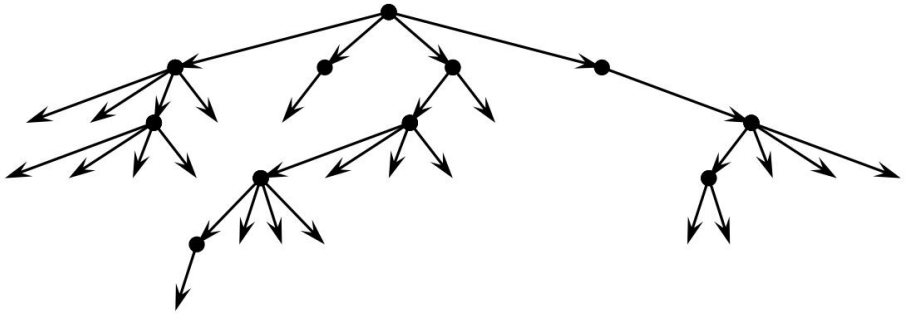
trees

5.1 Definition of trees

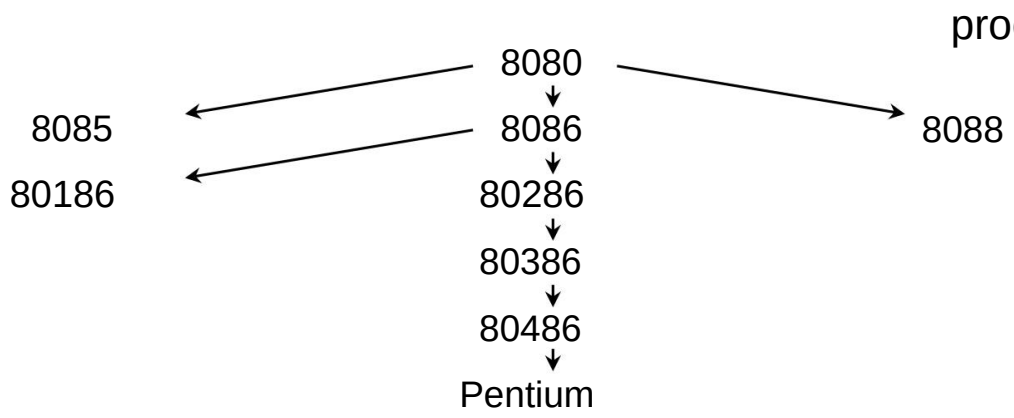
A **tree** is a special graph that defines a hierarchical structure over a set of objects

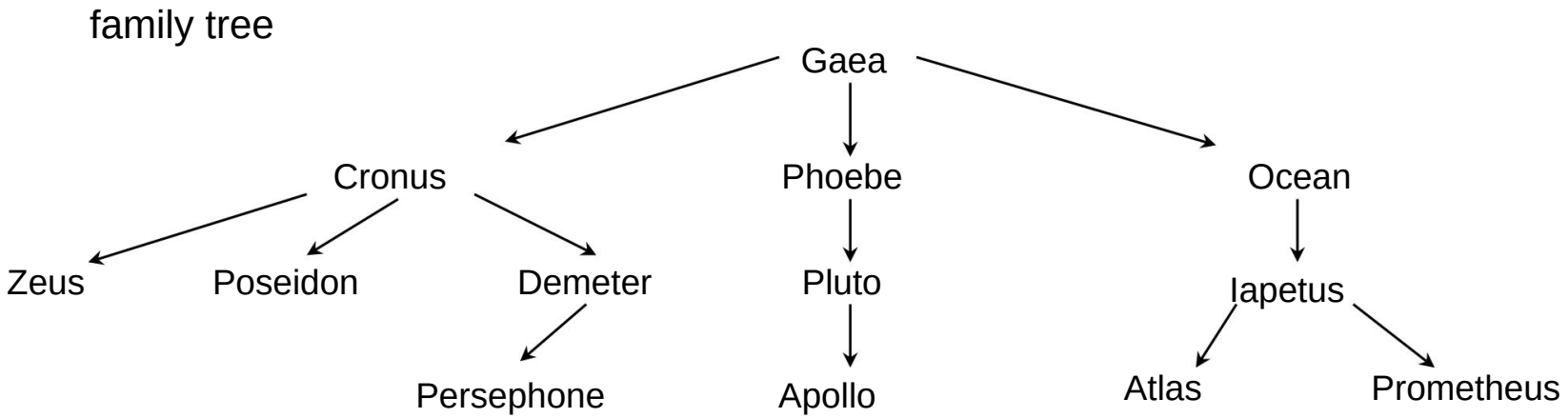
- Intuitive conceptual model - Non-linear data structure
- Applications
  - Represent knowledge, representation of structures, mapping of access paths, analysis of hierarchical relationships, ...





Example trees





Tree, nodes, edges, way, path

A **tree** consists of a set of **nodes** connected by **directed edges**

A **path** is a list of distinct nodes, with consecutive nodes connected by an **edge**

Defining property of a tree • There is exactly one path that connects 2 nodes

- Each node has only one direct predecessor
- all predecessors of a node are different from itself

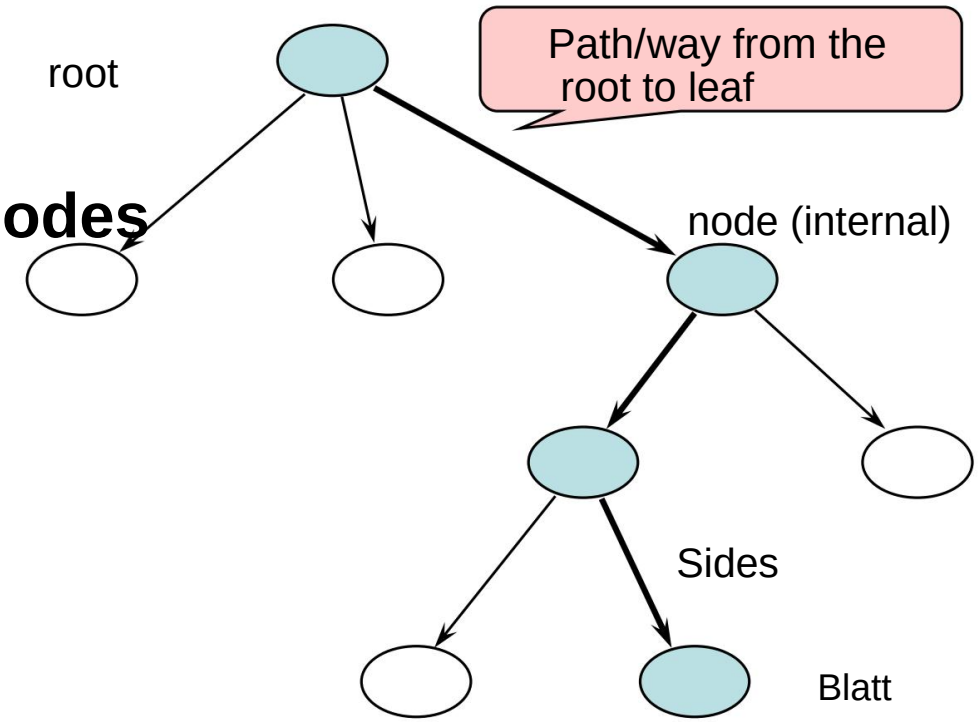
A tree therefore does not contain **any circles,** ie a path where the Start node is equal to the end node

Root, leaf, internal nodes

The **root** is **the** only node with only leading ones
Laces

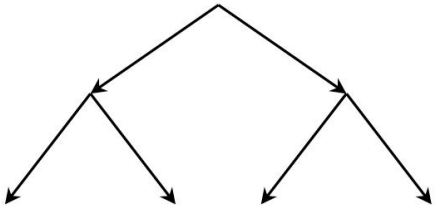
A **leaf** is **a** node with no edges leading away from it

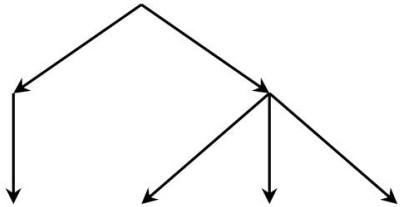
Nodes into which an
edge leads and
from which edges lead
away are called **internal nodes**



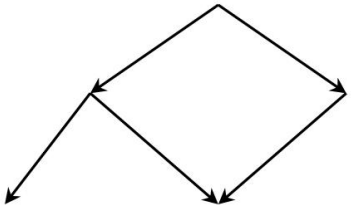
Valid and invalid trees

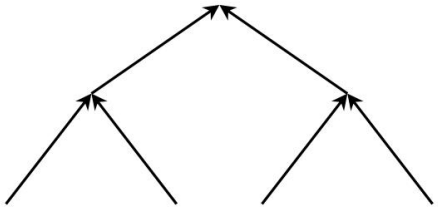
Valid trees

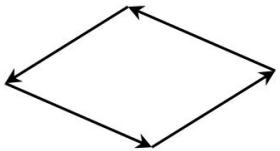




Invalid trees







- There is exactly one path that connects 2 nodes
- Each node has only one direct predecessor
- all predecessors of a node are different from itself

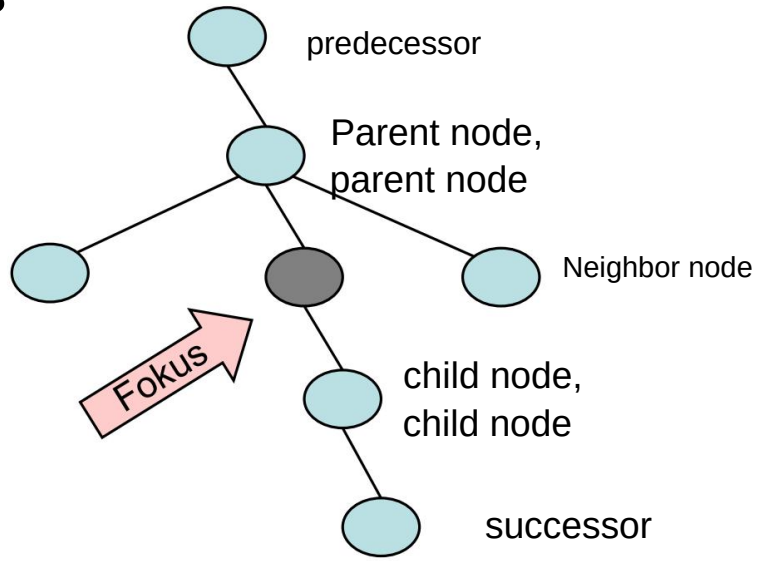
Special knots

Every node (except the root) has exactly one
Node above itself, which is called **parent node** or **parent node**

The nodes directly below a node are called **child nodes**
or **child nodes**

Transitive parents are referred to as **predecessors** and transitive children are referred to as **successors**

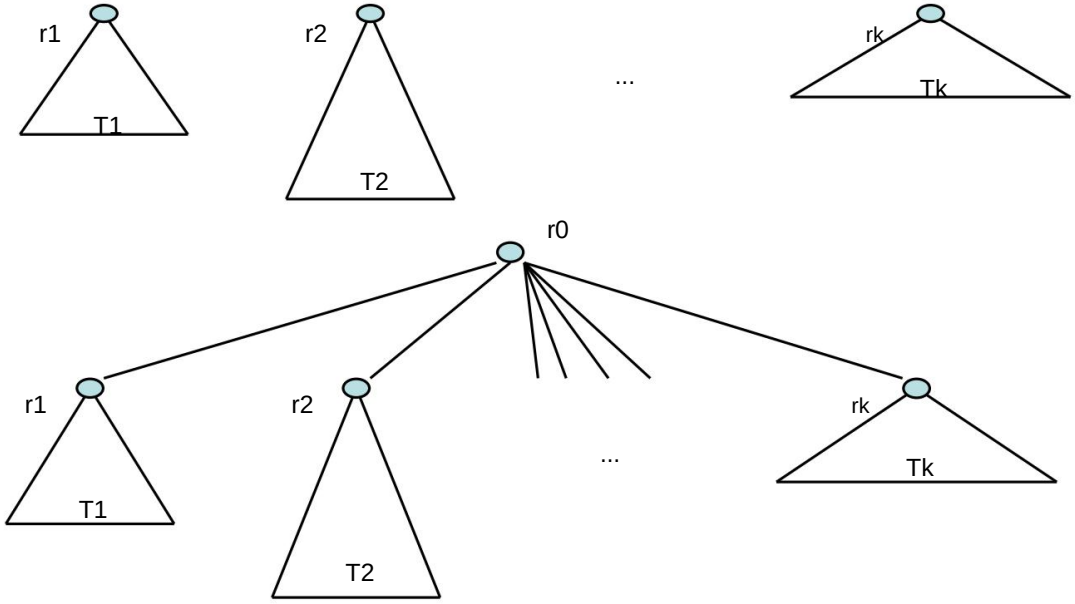
Spatially adjacent nodes at the same depth are called **Neighbors**



Recursive tree definition

A single node without edges is a tree

Let T1 , …, Tk (k > 0) be trees without common nodes. Let the roots of these trees be r1 , …, rk . A tree T0 with the root r0 consists of the nodes and edges of the trees T1 , ..., Tk and new edges from r0 to the nodes r1 , ..., rk The node r0 is then the new root and T1 , ..., Tk are **subtrees** of T0



ri … root of tree i Ti… subtree i

Length, height, depth

The **length of a path** between

2 nodes corresponds to the number of edges on the path between the two nodes

The **height of a node** is

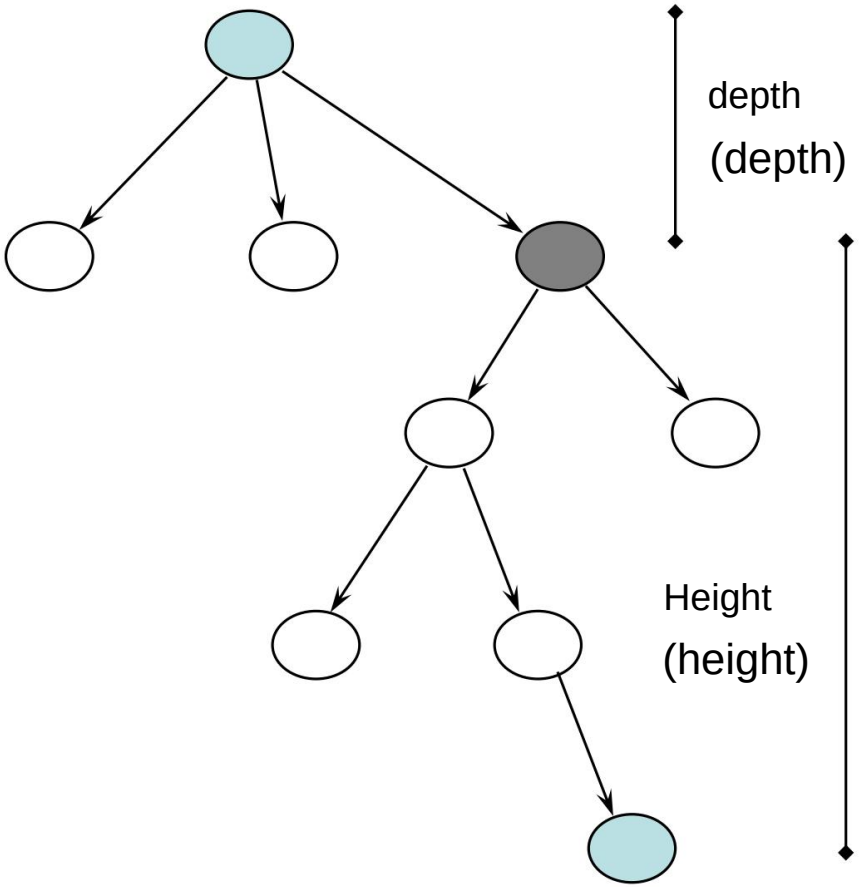
Length of the longest path from this node to the reachable leaves

The **depth of a node** is

Length of the path to the root

The **height of a tree**

corresponds to the height of the root



Example Parse Tree

Parse trees analyze statements or programs of a programming language with respect to a given grammar

A grammar defines rules about how and from which elements a programming language is constructed

<stmt>	Nonterminal Symbole (are dissolved)
if, ident	Terminalsymbole (no longer dissolved)
::=, |	Grammar symbolism

Example: C++ (excerpt)

<stmt> ::= *<select-stmt>* | *<expr>*

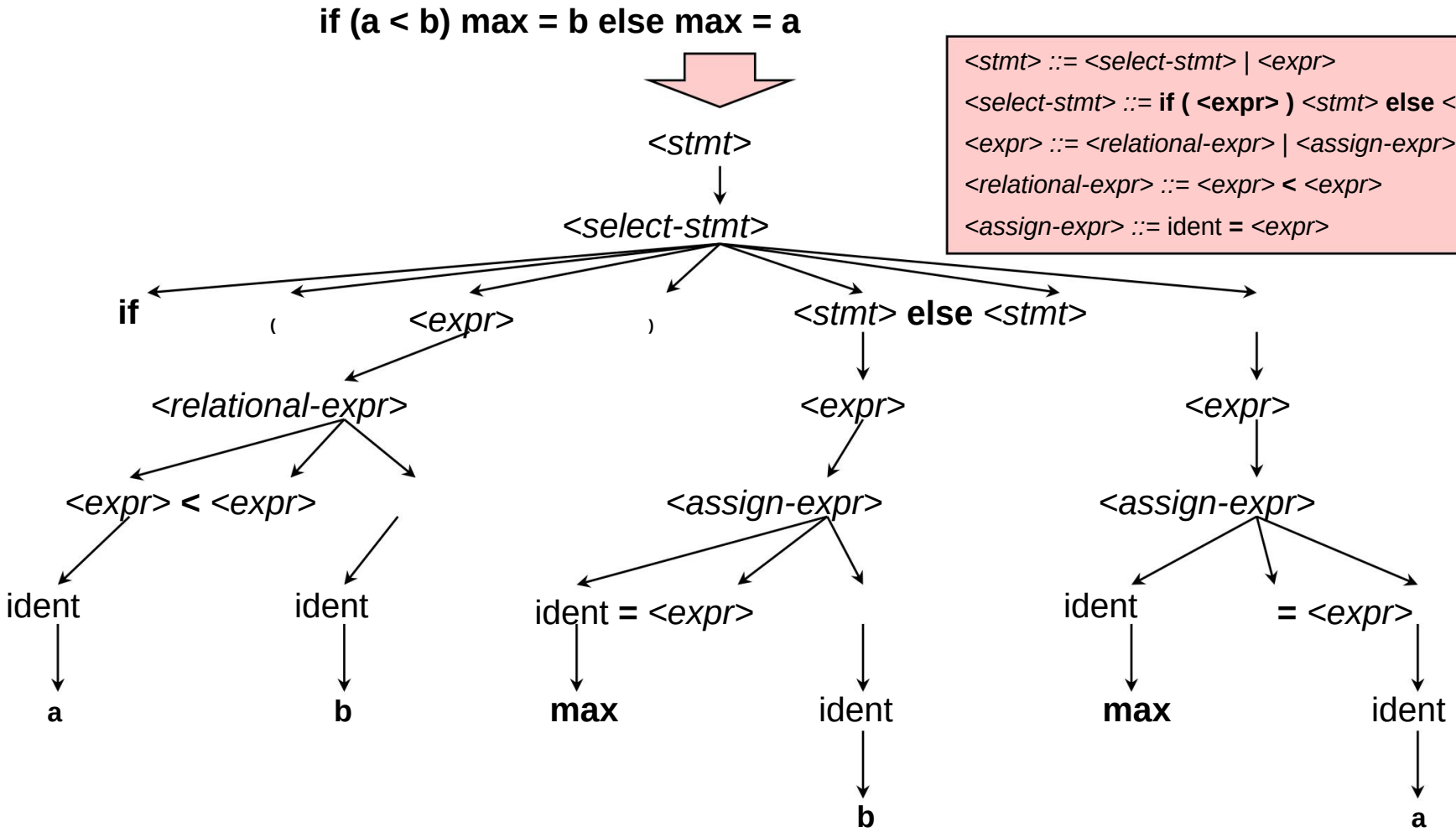
<select-stmt> ::= **if (<expr>)** *<stmt>* **else** *<stmt>*

<expr> ::= *<relational-expr>* | *<assign-expr>* | ident

<relational-expr> ::= *<expr>* **<** *<expr>*

<assign-expr> ::= ident **=** *<expr>*

Example Parse Tree (2)

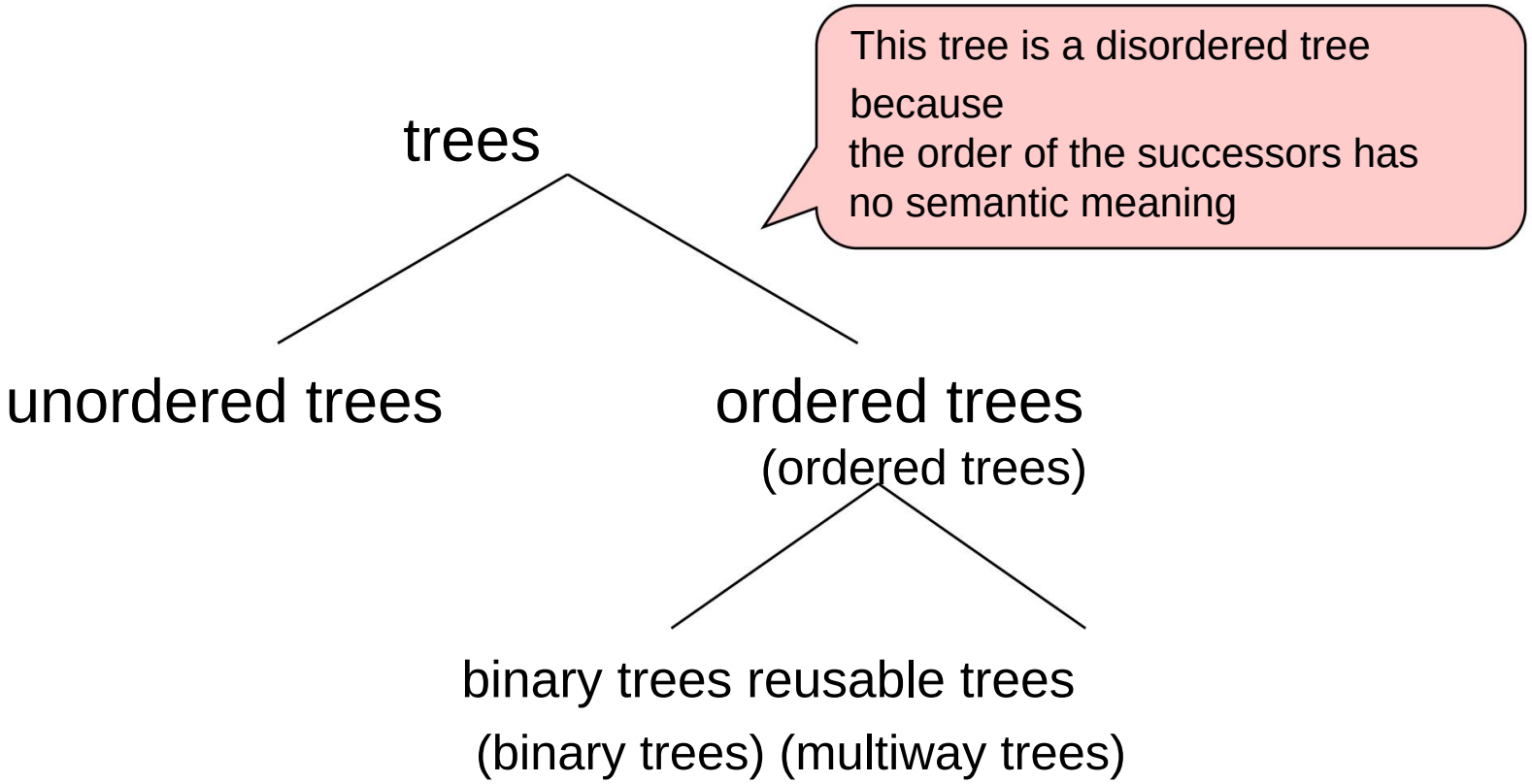


<stmt> ::= <select-stmt> | <expr>
<select-stmt> ::= **if (<expr>)** <stmt> **else** <stmt>
<expr> ::= <relational-expr> | <assign-expr> | ident
<relational-expr> ::= <expr> **<** <expr>
<assign-expr> ::= ident **=** <expr>

Special trees

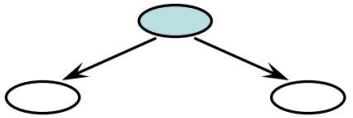
The **order** determines the position of the successors of a node in the graphical representation of the tree (node A to the left of node B)

This means that an order relation is defined for the children of each node

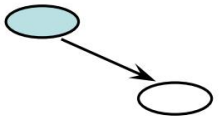


Binary trees are trees in which each node has a maximum of 2 has children and the order of the nodes is defined as left and right

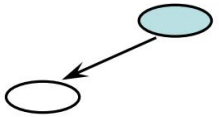
Binary tree with 2 children



Binary tree with right child



Binary tree with left child



Binary tree without child (leaf)

Binary Trees - Properties

Frequently used in algorithms

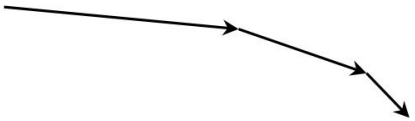
Efficient manipulation algorithms

Relationship between number of elements and height

Height 0	Height 1	Height 2
1 knot	3 knots	7 knots
1 Blatt	2 sheets	4 sheets

In each new level, in the optimal case, the number of sheets doubled

Degeneracy possible: Every node has exactly one child ÿ corresponds to linear list



Forms of binary trees

Empty binary tree

Binary tree without nodes

Full binary tree

Each node has zero or 2 children

Perfect binary tree

A full binary tree in which all leaves have the same depth

Complete binary tree

A perfect binary tree except that the leaf level is not
is completely filled, but from left to right

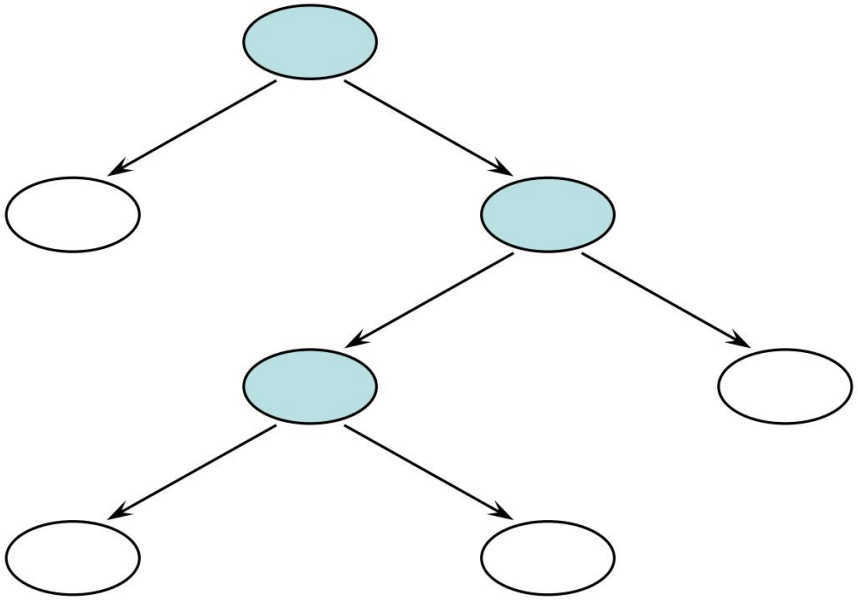
Height-balanced binary tree

For each node is the difference in the height of the left and right
child maximum 1

Full binary tree

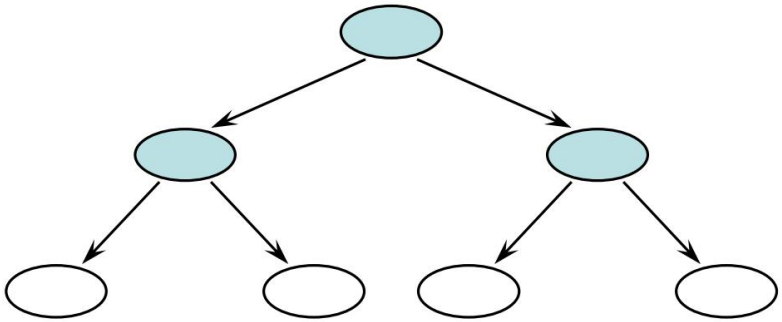
Each node in the tree has no children or exactly 2 children.

In other words, no node has only 1 child



Perfect binary tree

A full binary tree (all nodes have no or exactly 2 children) in which all leaves have the same depth.

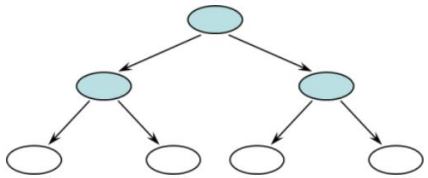


Properties of the perfect binary Baumes

Characteristics

Question: what height h must a perfect binary tree have to reach n

To have leaves



Doubling in every level, i.e. $2^h = n$

$\log_2 n = \log n$

$h = \log_2 n = \mathrm{ld}\ n$

A perfect binary tree of height h has $2^{h+1}-1$ nodes

2^h of which are leaves

Proof with complete induction

Relationship between nodes/leaves and height:

O(n) nodes/leaves : *O(log n)* height

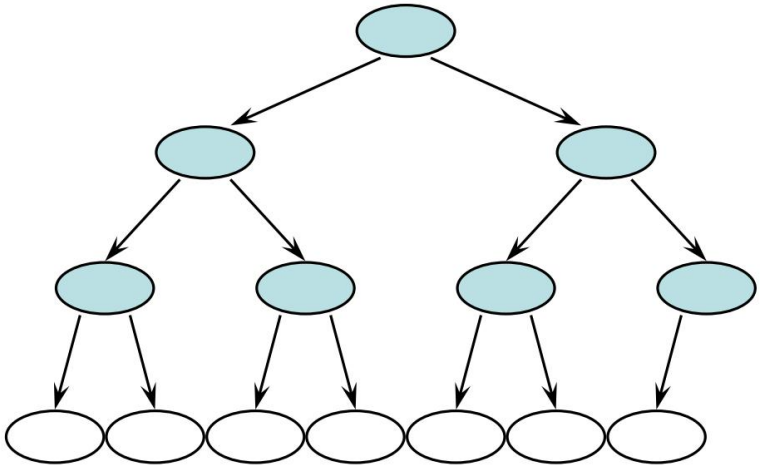
most important Characteristic of trees

Complete binary tree

Complete binary tree

A complete binary tree is a perfect binary tree with the

Exception that the sheet layer is not completely filled, but from left to right



Characteristic

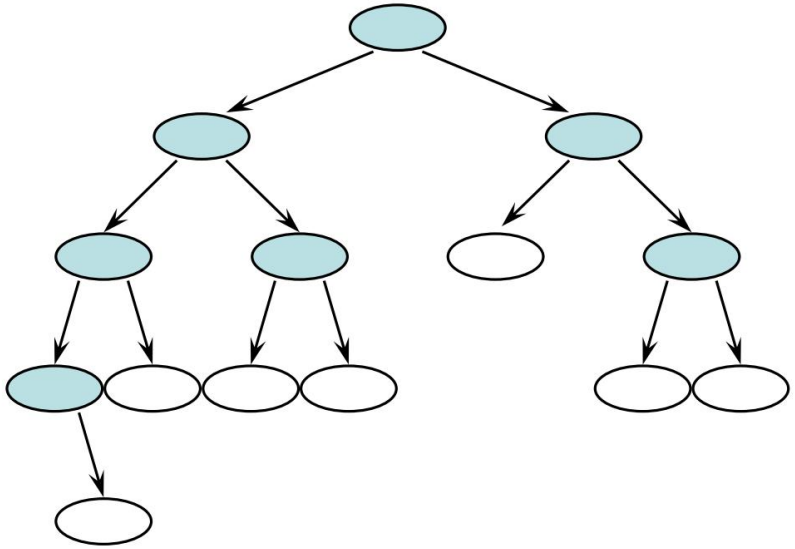
A complete binary tree with n nodes has a height of

maximal h = ÿlog2 nÿ

Height-balanced binary tree

For each node is the difference in the height of the left and right child maximum 1

This guarantees that locally for each child the balancing property is fulfilled relatively well, but globally the overall tree differences can be larger ÿ simpler algorithms



Tree traversal

Traversing a tree means systematic

Visiting all of its nodes

Different methods differ in the order of visited nodes

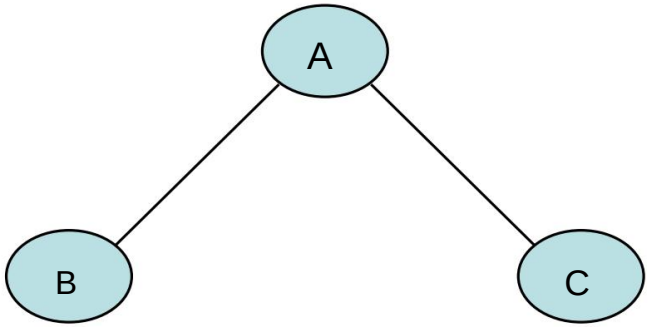
Possible orders

A, B, C

B, A, C

B, C, A

...

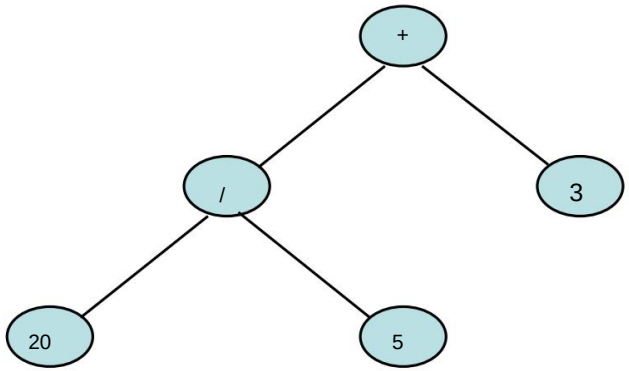


Expression Tree

Systematic evaluation of a mathematical expression

A mathematical expression can be in the form of an Expression Trees are specified

(20 / 5) + 3



Leaves represent operands (numerical values), internal nodes Operators

Evaluation of the expression runs from the leaves to the root

Tree representation saves bracket notation

Traversal algorithm

Traversal algorithms basically consist of 3 different steps

- Editing a node (process node)
- Recursively visit and edit the left child (visit left child)
- Recursively visit and edit the right child (visit right child)

By arranging the 3 steps differently, different orders

3 processing sequences interesting

- Preorder traversal
- Postal order Traversierung
- Inorder traversal

Preorder traversal

algorithm

```
preorder(node) {
    if(node != 0) {
        process(node)
        preorder(left child)
        preorder(right child)
    }
}
```

Visits the nodes in the tree in *prefix*

Notation order

(Operator, Operand1 , Operand2)

Rule of thumb

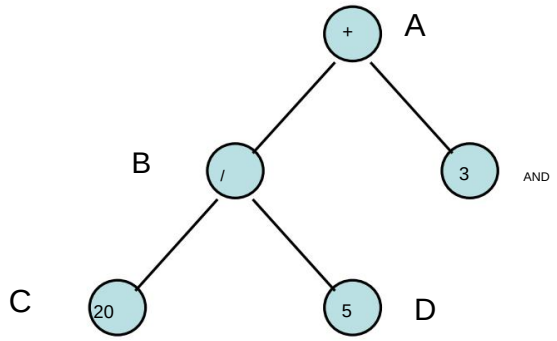
Edit nodes on 1st visit

Application

LISP, Assembler

Example

20/5+3



Process order: ABCDE

Notation order: + / 20 5 3

Postal order Traversierung

algorithm

```
postorder(node) {
    if(node != 0) {
        mail order(left child)
        mail order(right child)
        process(node)
    }
}
```

Postfix notation order

(Operand1, Operand2, Operator)

Rule of thumb

Edit nodes at the last one

Visit

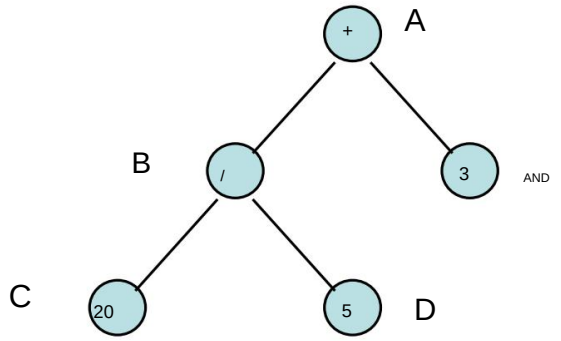
Application

Invers Polish Notation, HP

Calculator, FORTH

Example

20/5+3



Process order: CDBEA

Notation order: 20 5 / 3 +

Inorder traversal

algorithm

```
inorder(node) {
    if(node != 0) {
        inorder(left child)
        process(node)
        inorder(right child)
    }
}
```

Infix notation order

(Operand1 , Operator, Operand2)

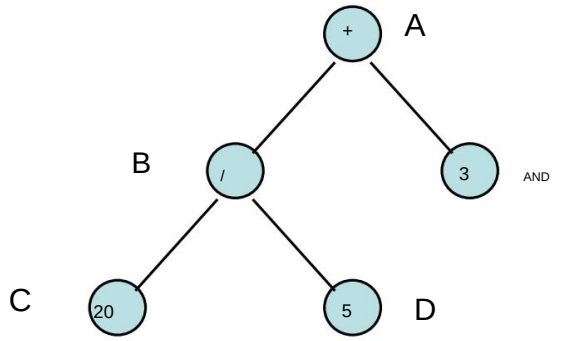
Rule of thumb

Edit nodes on 2nd or last visit

Using simple algebraic calculators (without brackets)

Example

20/5+3



Process order: CBDAE

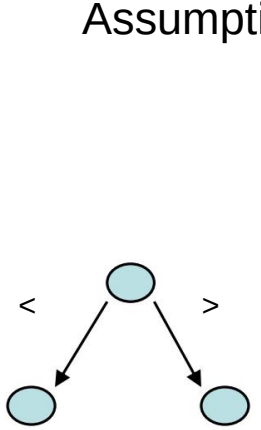
Notation order: 20 / 5 + 3

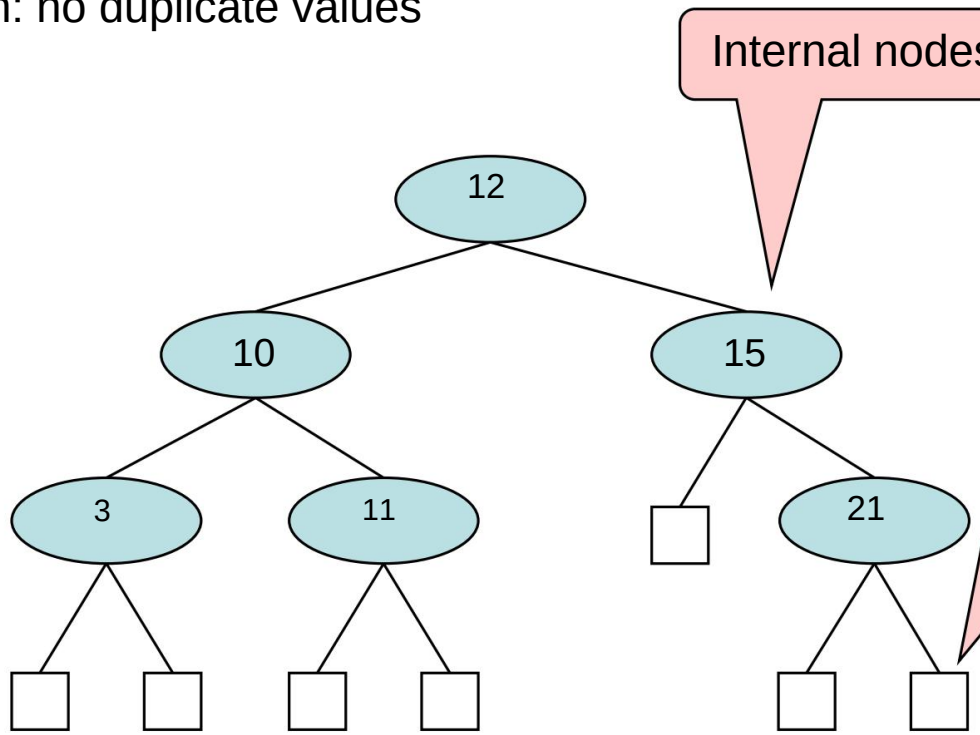
- In a binary tree, each (internal) node has a left and a right connection that points to a binary tree or an external node (full binary tree)
  - Connections to external nodes are called zero connections, external Nodes have no further connections
- A binary search tree (BST) is a Binary tree in which each internal node has a key
  - external nodes do not have keys
    - External nodes contain the managed information, i.e. data storage (will not discussed further here)
  - A linear order “<“ is defined on the keys
    - If x and y are different, then x<y or y<x; if x<y and y<z, then x<z also applies

Key property in BST

For each internal node, all values of the successors in the left subtree are less than the node value and the values of the successors in the right subtree are greater than that node value are

Assumption: no duplicate values





External nodes

They basically represent the data storage (contain the managed information, no pointers to successors), e.g Pages on the plate

Usually negligible for main memory structures

are often only represented graphically by a line

Properties of binary search trees

- Management of any size of data
  - dynamic structure
- Efficient administration
  - Effort proportional to the height of the tree (expected value!) and not to the number of elements
  - Insertion, access and deletion average O(log n)
  - Significant improvement compared to the linear effort O(n). List
  - Access elements in sorted order through inorder traversal

The operation

Create

- Creating an empty search tree

Insert

- Inserting an element into the tree taking the
  - Order property

Search

- Test for inclusion

Delete

- Removing an element from the tree while preserving the
  - Order property

output

- Output all elements in sorted order

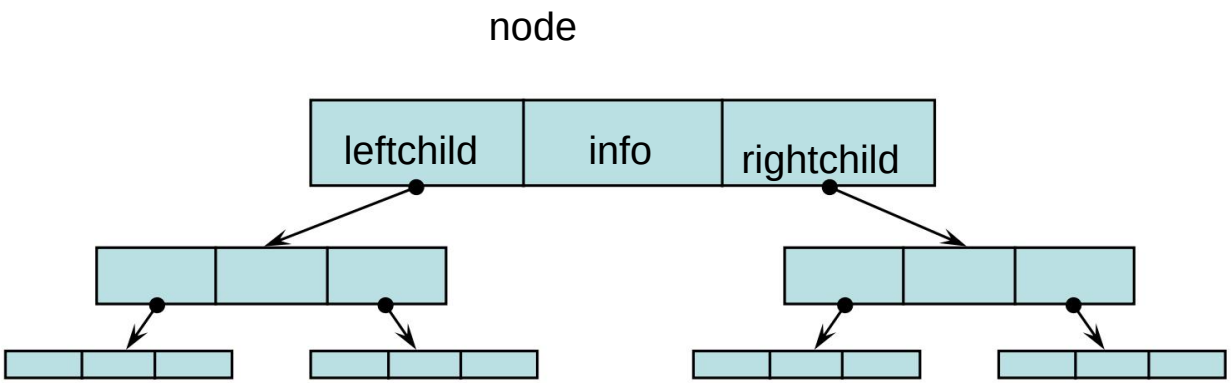
...

Class definition

```
typedef int ItemType; class
SearchTree { class node
    { public:

        ItemType info;
        node * leftchild, * rightchild; node(ItemType
        x, node * l, node * r) {info=x; leftchild=l; rightchild=r;} }; typedef node * link; link root; void

    AddI(ItemType); void
    AddR(link&,
    ItemType); bool
    MemberI(ItemType); bool MemberR(link,
    ItemType); void PrintR(link, int);
    public:

    SearchTree(){root = 0;} void
    Add(ItemType); int
    Delete(ItemType); bool
    Member(ItemType); void
    Print(); };
```

For the sake of simplicity,

everything is **public** in the **node**



Finding a key involves tracing a path from the root downwards

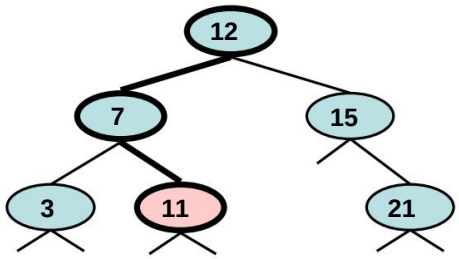
For each internal node, the key x is combined with the search key s compared

If x = s there is a hit, if s < x search in the left subtree, otherwise in the right

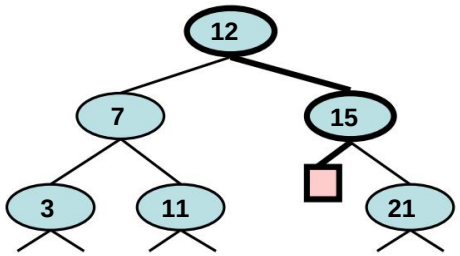
Reaching an external node was the key search

"unsuccessful"

Successful search (e.g. 11)



Unsuccessful search (e.g. 13)

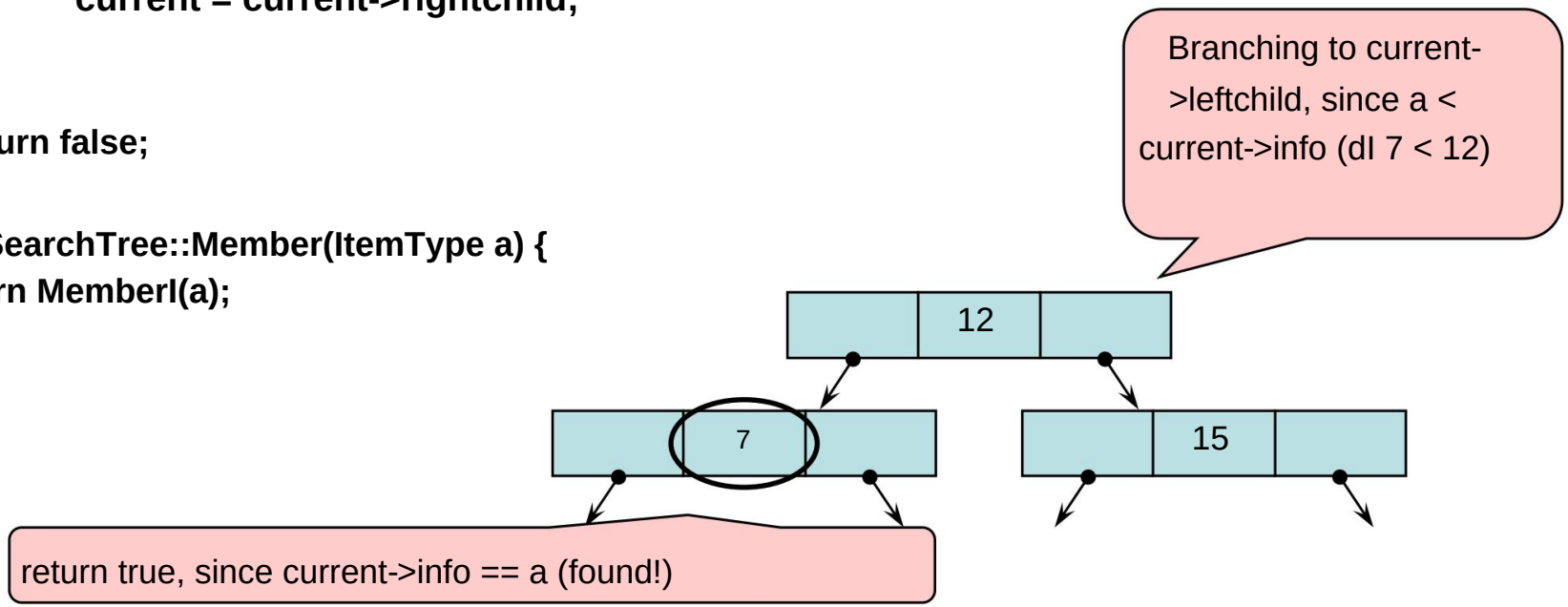


Search (iterative)

```
bool SearchTree::MemberI(ItemType a) { if(root) { link
    current = root;
        while(current) {

            if(current->info == a) return true; if(a < current->info)

                current = current->leftchild;
            else
                current = current->rightchild;
        }

    } return false;

} bool SearchTree::Member(ItemType a) {
    return MemberI(a);
}
```

Call:

```
SearchTree t;
...
t.Member(7);
```



Search (recursive)

```
bool SearchTree::MemberR(link h, ItemType a) { if(!h) return false;
    else { if(a == h->info) return
    true; if(a
        < h->info)

            return MemberR(h->leftchild, a); else return

        MemberR(h->rightchild, a);
    }

} bool SearchTree::Member(ItemType a) {
    return MemberR(root, a);
}
```

Call:

```
SearchTree t;
...
t.Member(7);
```

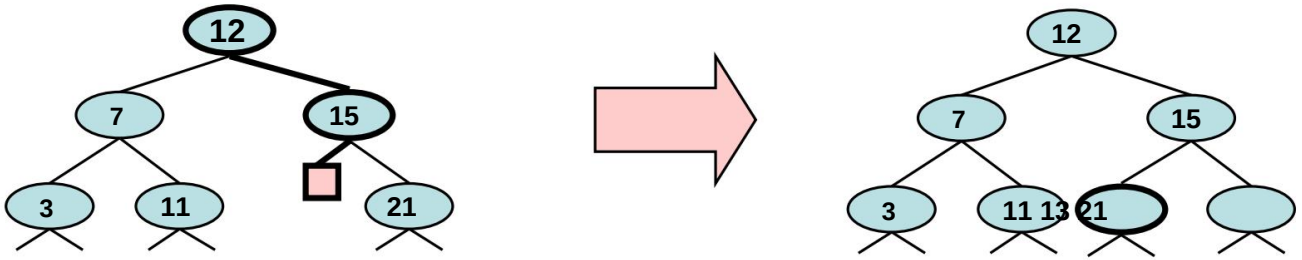
Inserting into a binary search tree

Inserting corresponds to an unsuccessful search and that
Adding a new node to the zero connection where the
Search ends (instead of the external node)

Insert 13

Search

Add nodes



Insert (iterative)

Similar to inserting into a list (2 auxiliary pointers)

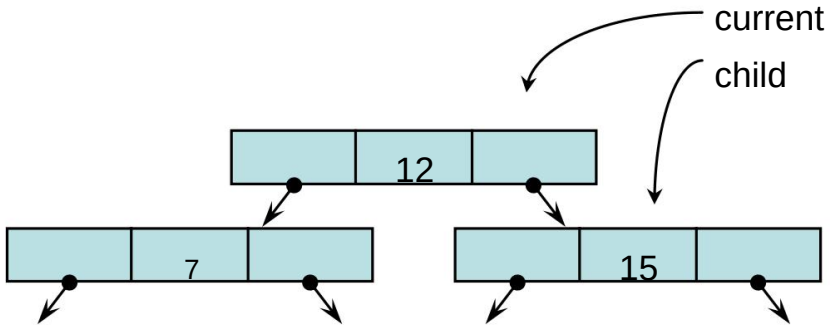
```
void SearchTree::AddI(ItemType a) { if(root) { link
    current = root;
        link child; while(1) { if(a ==
        current->info)
        return; if(a <
            current->info) { child = current->leftchild; if(!
            child) {current->leftchild = new
                node(a, 0, 0); return;} } else { child =
                current->rightchild; if(!child) {current->rightchild = new node(a, 0, 0); return;}
```

Hilfszeiger: **current, child**

Call:

```
SearchTree t;
...
t.Add(7);
```

```
            } current = child;

    } } else
        { root = new node(a, 0, 0); return;

    }

} void SearchTree::Add(ItemType a) {
    ...
    AddI(a);
}
```



Insert (recursive)

```
void SearchTree::AddR(link& h, ItemType a) {
    if(!h) {h = new node(a, 0, 0); return;}
    if(a == h->info) return;
    if(a < h->info)
        AddR(h->leftchild, a);
    else
        AddR(h->rightchild, a);
}
void SearchTree::Add(ItemType a) {
    AddR(root, a);
}
```

call

```
SearchTree t;
...
t.Add(7);
```

Note the use of a reference parameter **(link&),** which saves the 2nd one

Auxiliary pointer

traverse

```
void SearchTree::PrintR(link h, int n) {
    if(!h) return; PrintR(h-
    >rightchild, n+2); for(int i = 0; i < n; i++)
    cout << cout << h->info << endl; PrintR(h->leftchild,    " ";
    n+2);



} void SearchTree::Print() {
    PrintR(root, 0);
}
```

Inorder traversal

call

```
SearchTree t;
...
t.Print();
```

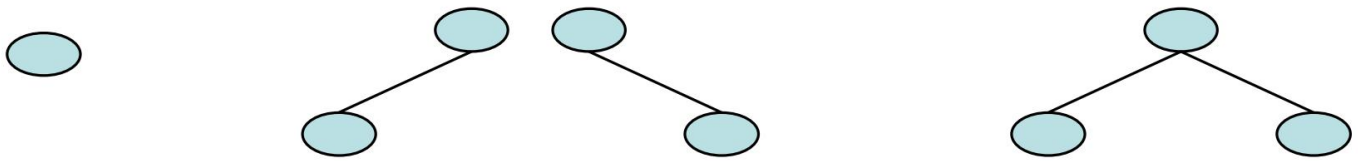
Gives tree 90 degrees against the twisted clockwise

Deletion in a binary tree

The deletion process distinguishes between 3 cases

Delete internal nodes with

(1) none (2) an internal node as child(ren) (3) two



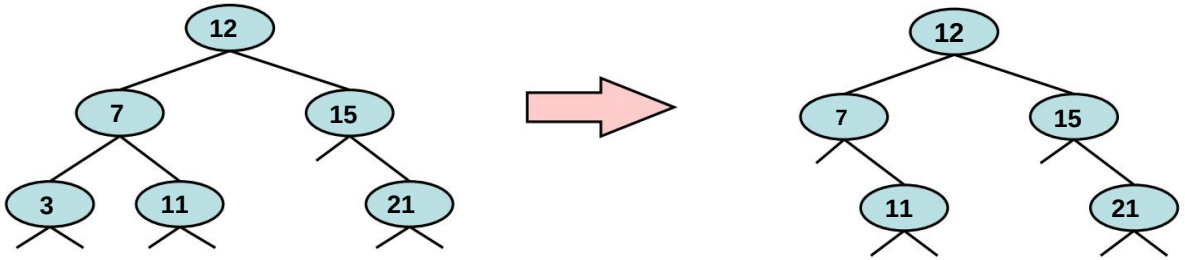
Cases 1 and 2 are easily solved by appending the remaining subtree to the parent of the node to be deleted.

For case 3, a suitable replacement node must be found. For this Either the smallest in the right subtree (inorder successor) or the largest in the left subtree (inorder predecessor) are suitable.

Delete (2)

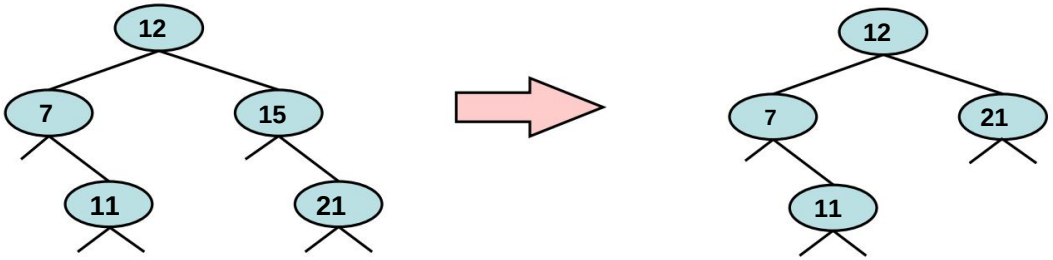
Fall 1

Delete node 3 by replacing the left child pointer of node 7 through the external node (here zero connection)



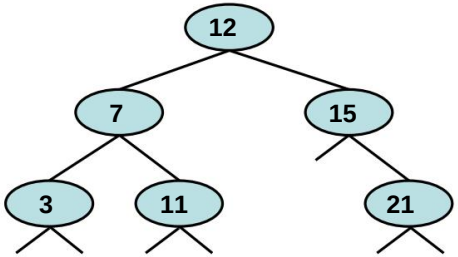
Fall 2

Delete node 15 by mounting node 21 as the right one Child of 12

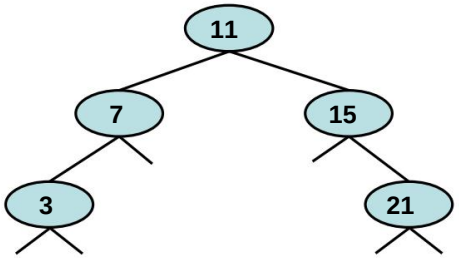


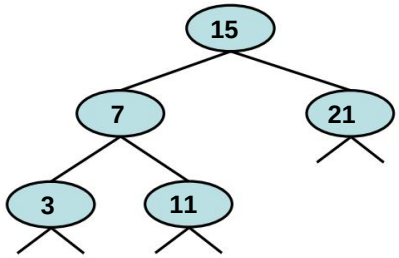
Delete (3)

Fall 3



In the case of deleting 12, suitable replacements are the nodes with the values 11 (largest in the left subtree) or 15 (smallest in the right subtree).
This results in 2 possible trees:





Thus, the deletion process for case 3 must be broken down into two parts:

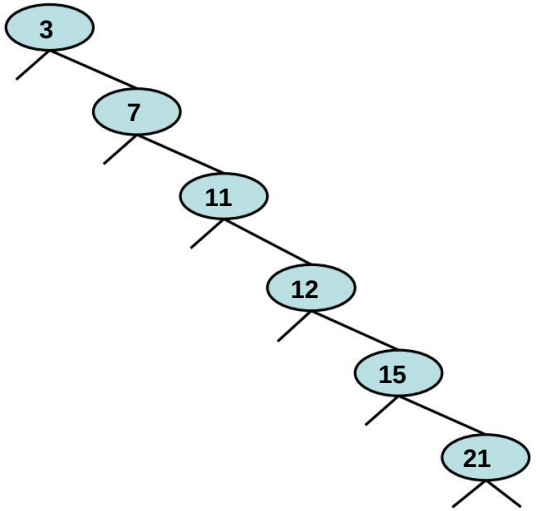
1. Find a suitable replacement node
2. Replacing the node to be deleted (which in turn consists of removing of the replacement node from its original position (corresponds to case 1 or 2) and hanging it in the new location)

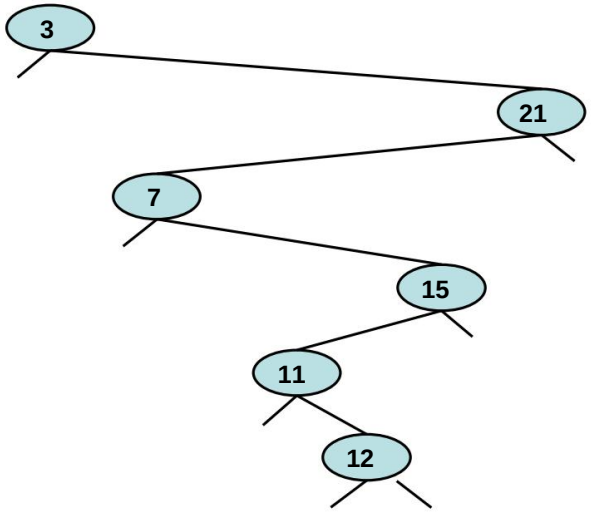
Duration

Expected value of the insert and search operations with n random key values is approximately 1.39 ld n

In the worst case, the effort can be approximately n Operations “degenerate”

Examples of unfavorable search trees





Binary search tree analysis

Data management
- supports insert and delete

Amount of data
- unlimited

Models
- Main memory oriented
- Support complex operations
  - Range queries, sort order

Duration

Storage space	O(n)
Constructor	O(1)
access	O(log n)
Insert	O(log n)
Delete	O(log n)
Sorting order	O(n)

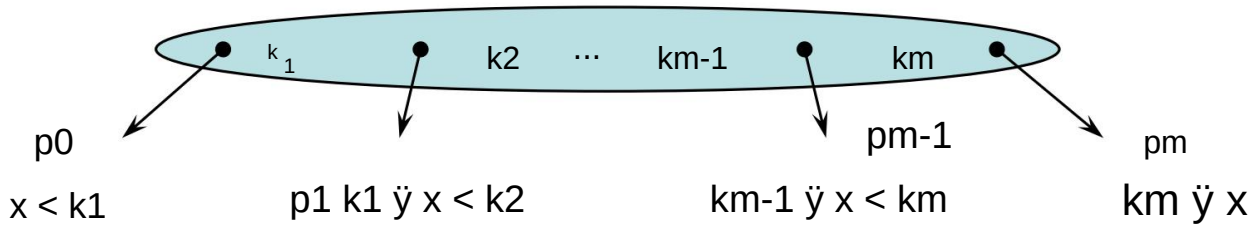
Please note: considerable constant effort is necessary!

Expected value!

5.4 Reusable trees

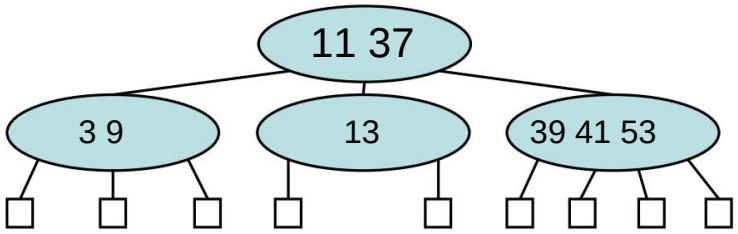
Reusable trees are trees with nodes that can have more than 2 children

Interval-based search data structures



Node consists of a set of **m key values** k1 , k2 , …, km and m **+1 references** (edges) p0 , p2 , …, pm, such that for all keys xj in the subtree referenced by pi , ki ÿ xj < ki+1 (keys lie in the interval [ki , ki+1[).

Example



Internal nodes

External nodes

Search in the reusable tree

Similar to searching in the binary search tree

For every internal node with keys k1 , k2 , ..., km and connections p0 , p1 , ..., pm search key

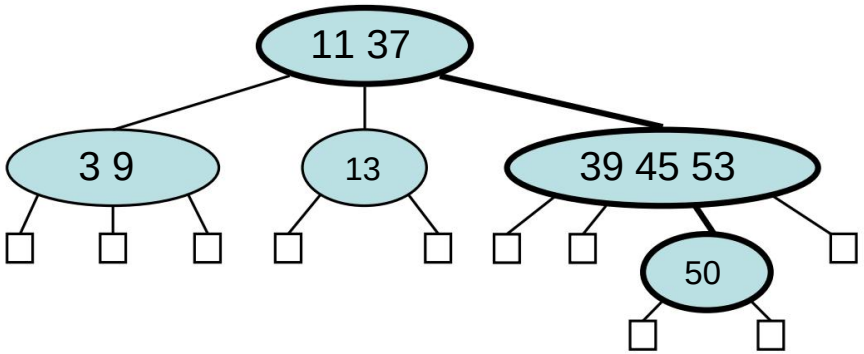
s = ki (i = 1, ..., m): search successful s <

k1 : continue in subtree p0 ki ÿ s < ki+1: continue in the subtree pi s

ÿ km: continue in the subtree pm If you reach

an external node: key search unsuccessful

Example: Search 50



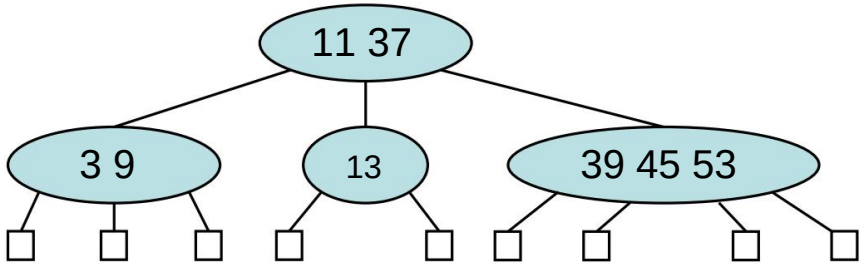
2-3-4 trees

A **2-3-4 tree** is a multi-way tree with the following Characteristics

Size property: each internal node has a minimum of 2 and a maximum 4 children

Depth property: all external nodes have the same depth

Depending on the number of children, an internal node is called 2-node, 3-knot or 4-knot



Height of a 2-3-4 tree

Theorem: A 2-3-4 tree that stores n internal nodes has (always) a height of O(log n)

Proof

Let the height of the 2-3-4 tree with n internal nodes be h

Since there are at least 2i internal nodes at depths i = 0, ... , h-1 and no internal nodes at depth h, + 2h-1 = 2h – 1

n ÿ 1 + 2 + 4 + ...

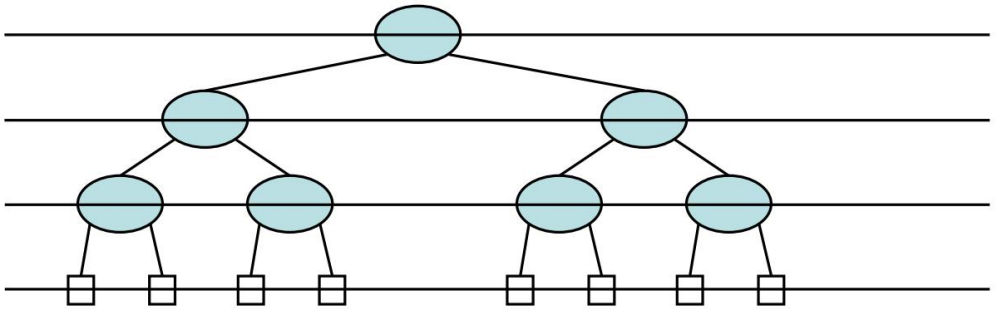
Therefore h ÿ log2 (n + 1)

n ÿ 2h –

1 n + 1 ÿ 2h

log (n + 1) ÿ h log 2

h ÿ log (n +1) / log 2 = log2 (n + 1)



node	Depth
1	0
2	1
2 h-1	h-1
0	h

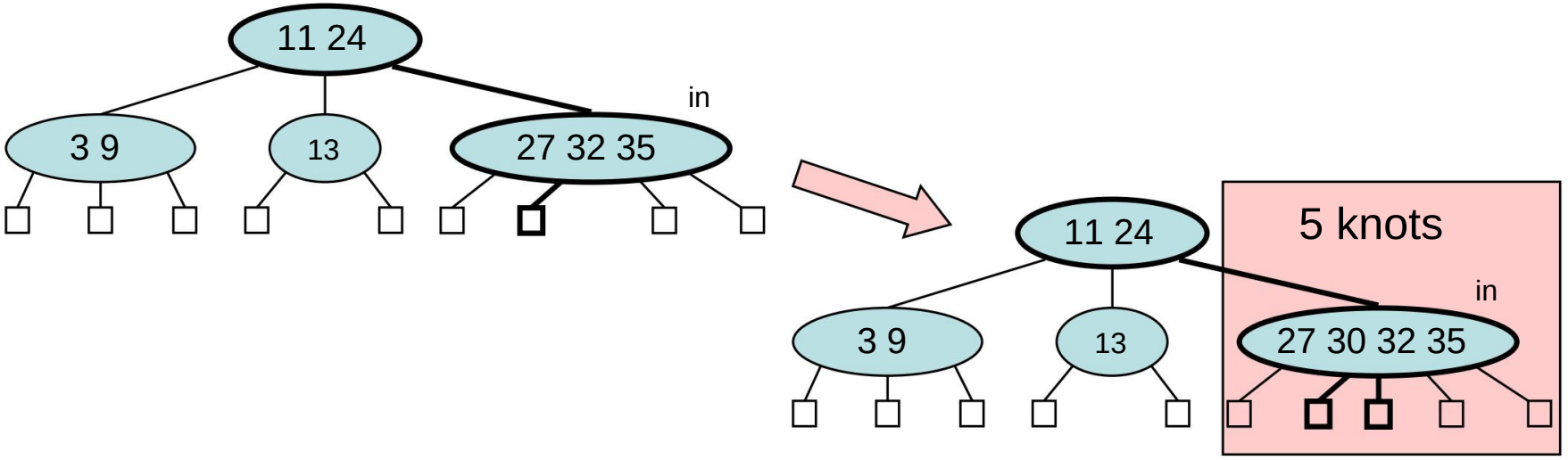
Insert into a 2-3-4 tree

A new key s is created in the parent node v of the external
Inserted node that was reached when searching for s

The depth property of the tree is preserved, but the size property is potentially violated

A (node) overflow is possible (a 5-node is created)

Example: Inserting 30 creates overflow



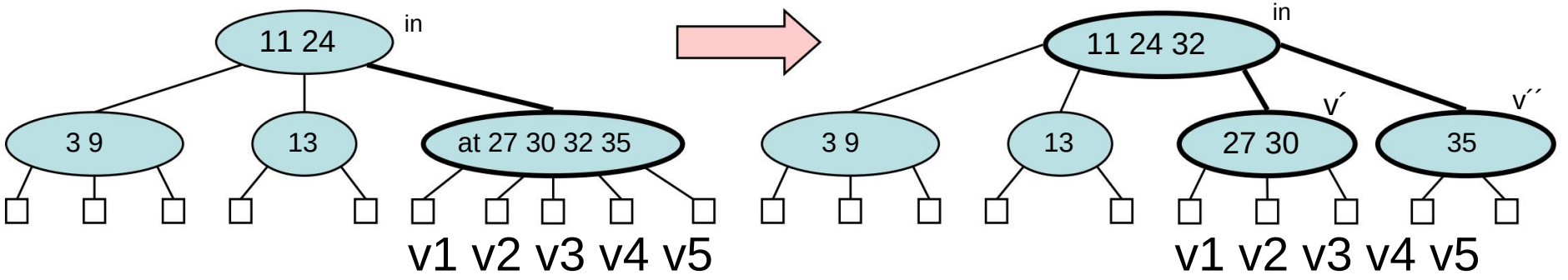
Overflow and split

An overflow at a 5-node v is resolved by a split operation. The children of v are v1 , … , v5 and the keys of v are k1 , … , k4 Node v is represented by the nodes v´ and v´´ replaced, whereby

v´ is a 3-node with keys k1 and k2 and children v1 , v2 and v3 , v´´ is a 2-node with key k4 and children v4 and v5

Key k3 is inserted into the parent node u of v (this can create a new root)

An overflow can be propagated to the predecessors of u



Delete

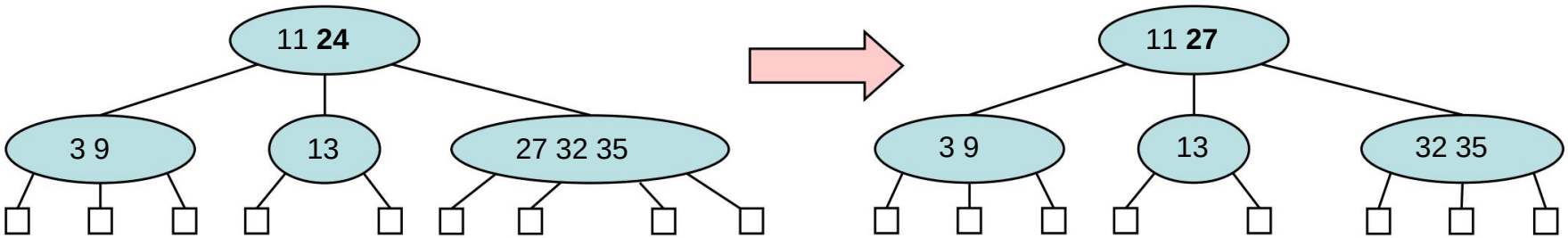
The deletion process is reduced to the case where the key value to be deleted lies in an internal node with external node children

Otherwise the key value will be with its inorder

Successor (or inorder predecessor) replaced

Analogous to case 3 binary search trees

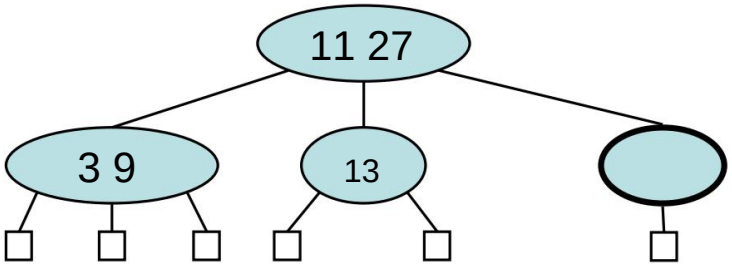
Example Delete Key 24



Underflow

Deleting a key in a node v can cause an underflow

Node v degenerates into a 1-node with one child and none Key



in

Two cases can be distinguished in the case of an underflow

Case 1: Merge

Adjacent nodes are merged into one allowed node

Case 2: Move

Key values and corresponding children are passed between nodes delay

Merge in the 2-3-4 tree

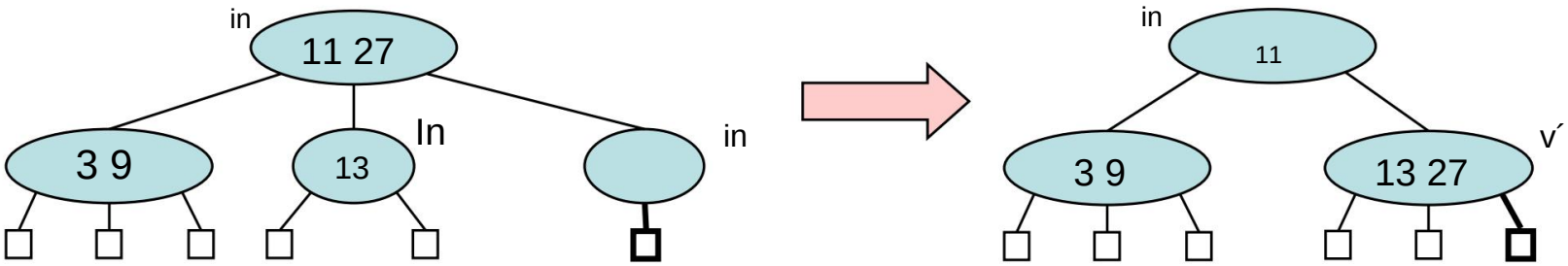
Case 1: Merging nodes

Condition: All neighboring nodes at the same depth to the underlying node v are 2-nodes

You merge v with a neighbor w and don't move it

more required key from the parent node u to the merged node v´

Merging can propagate the underflow to the parent node



Case 2: Moving keys

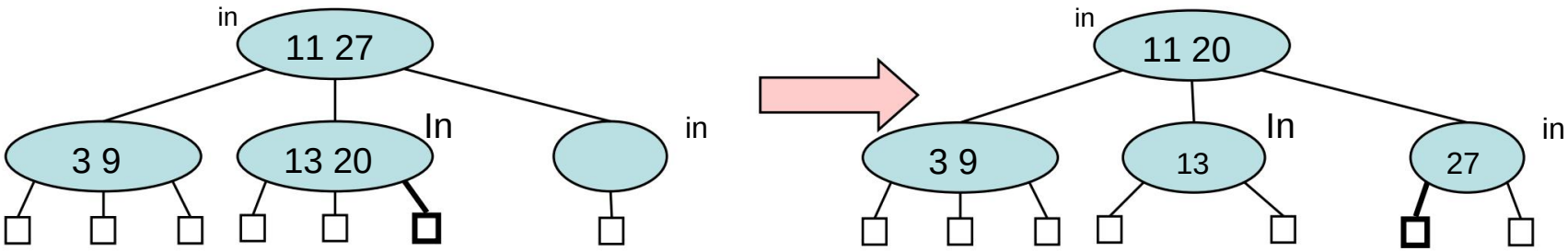
Condition: An adjacent node at the same depth w to the underlying node v is a 3-node or 4-node

You move a child from w to v

You move a key from u to v

You move a key from w to u

After moving, the underflow is fixed



Analysis 2-3-4 tree

Data management
- Insert and delete supported

Amount of data
- unlimited

Models
- Main memory oriented
- Support complex operations
  - Range queries, sort order

Duration

Storage space	O(n)
Merge, Move O(1)	
access	O(log n)
Insert	O(log n)
Delete	O(log n)
Sorting order	O(n)

} Guaranteed!

Comparison of tree variants

Baumvariant	*Tree height + effort of The operation*	*Balancing form*	*Method*
More general Binary Search tree	On average log(n)	Depends on the input	Coincidence, law of large numbers
2-3-4 Baum	log(n)	perfectly balanced	Split, merge, Move

5.5 External storage management

Goal

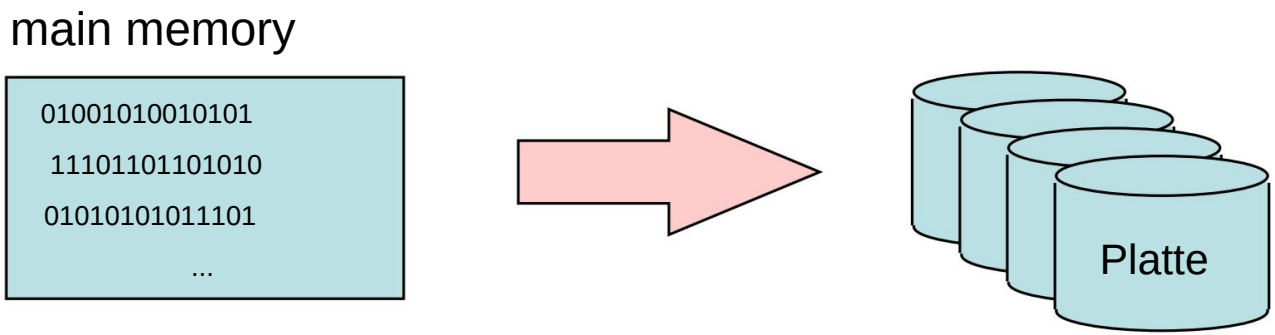
Creating a key tree for a large number of elements

Problem

Main memory too small

Solution

Storage of the nodes on the external memory (disk)



Disk accesses

Approach

- Referencing a node corresponds to accessing the disk
  - When building a binary key tree for a data set of, for example, one million elements, the average tree height is log2106 ÿ 20, ie 20 search steps ÿ 20 disk accesses

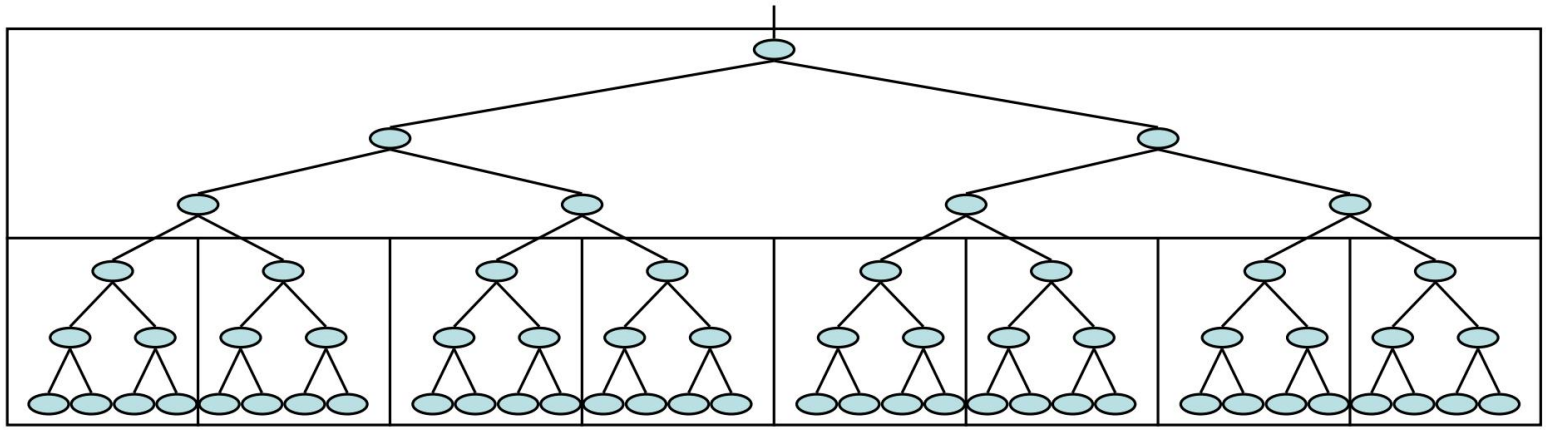
Dilemma

- Difference in effort between main memory and disk access is a factor of 10,000 (SSD) to 100,000 (HD)
  - milli- to nano-seconds (e.g. 60 ns to 0.25-9ms)
- Each search step requires disk access - a lot of effort
  - Calculation time

Page management

Solution

Decomposition of the binary tree into subtrees, which are saved in *so* -called *pages*



ÿ Multiway tree

Interval-based search data structure

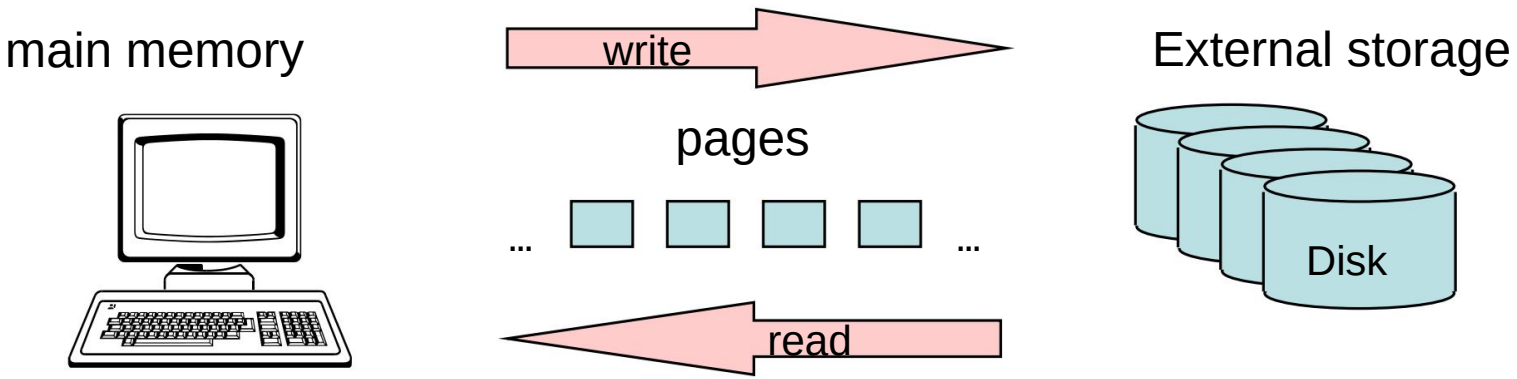
With 100 nodes per page for 1 million elements, only log100106 = 3 Pageviews

In the worst case (linear degeneracy) but still 104 accesses (106 / 100)

Page management (2)

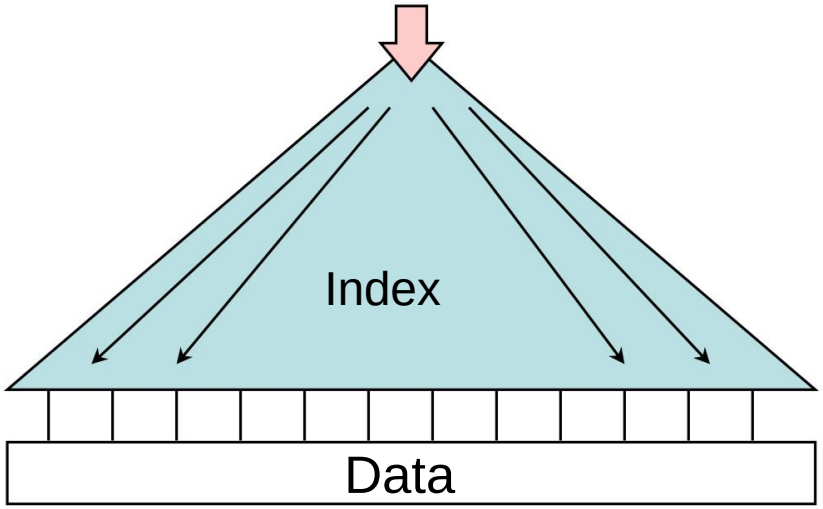
Nodes correspond to pages (blocks)

- have a size, page size (number of entries contained or Memory size in bytes)
- represent the transfer unit between main and external memory
- To increase the efficiency of the transfer, the size of the pages is adjusted Block size of the memory transfer units of the operating system adjusted (Unit of Paging/Swapping)



Index structure consists of key part (index) and data part

Graphic structure



Index

enables efficient access to the data

Data

contain the stored information (compare external nodes)

Types of knots

2 types of knots

Index notes

Allow efficient access to external nodes

Define the interval ranges of the elements in the associated subtree are stored.

realized by internal nodes

External nodes

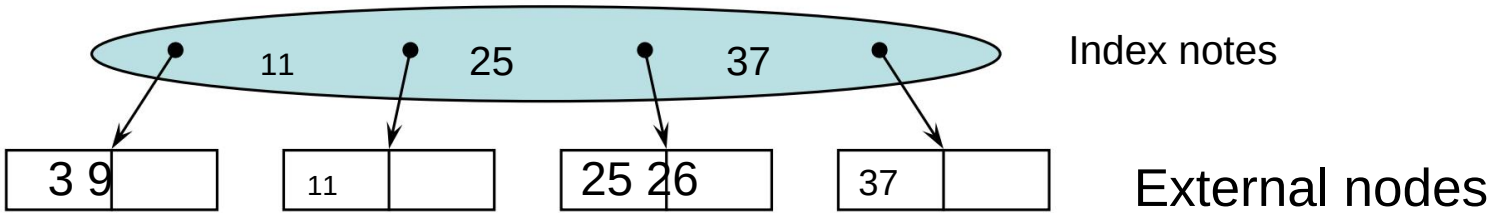
Contain the data set identified by key **(key, info)**

compare with **Dictionary**

realized by leaf nodes

In the example we only use key value, but represents entire data set

Example



universität wien

Height-balanced (perfectly balanced) reusable tree

Characteristics

- External storage data structure
  - One of the most common data structures in database systems
- Dynamic data structure (insertion and deletion)
- Insertion and deletion algorithms maintain the balancing characteristic
- Guarantees a limited (worst-case) effort for access, insertion and delete
- The effort for the operations access, insertion and deletion is due to the tree structure of the order log n (O(log n))
- Consists of index and data
- The path to all data is the same length

Definition of B+ tree of order k

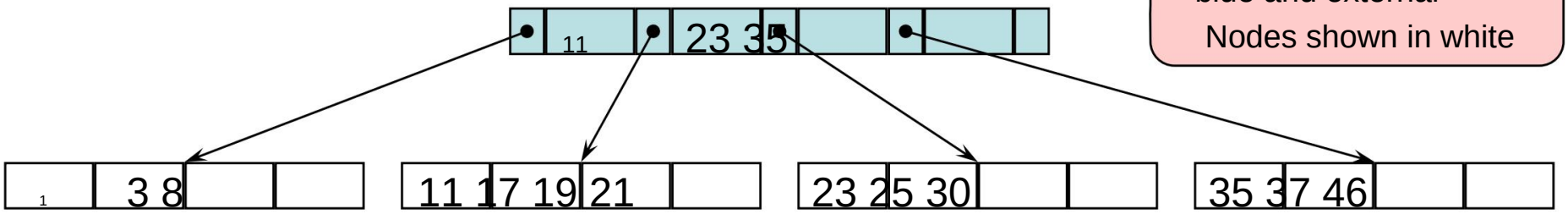
All leaf nodes have the same depth (same path length to the root)
The root is either a leaf or has at least 2 and max. 2k+1 children

Each (internal) node has at least k and a maximum of 2k key values, therefore at least k+1 and a maximum of 2k+1 children

For each index entry, the left subtree only stores values that are smaller than the index entry and the right subtree only stores values that are greater than or equal to the index entry.

Interval-based search data structure

Example: B+ tree of order 2



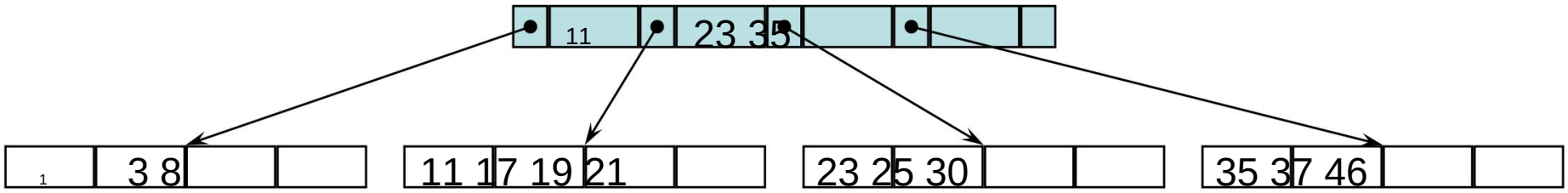
Note: Index nodes turn blue and external
Nodes shown in white

Each index node has a minimum of 2 and a maximum of 4 key values and therefore a minimum of 3 and a maximum of 5 children

The exception is the root, which can only contain 1 key value with 2 children

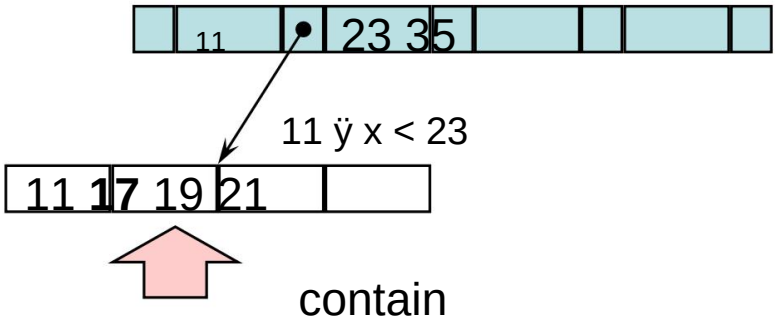
The size of the external nodes is actually not defined by the order but usually equated in the literature

Search



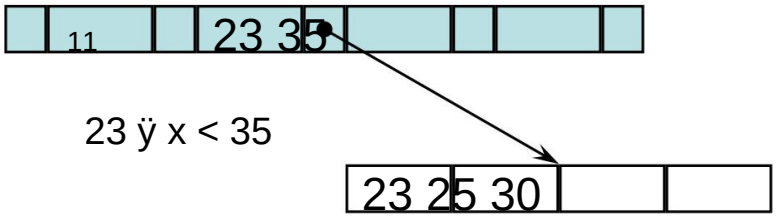
Find a record with given key

Data record with key 17



contain

Data record with key 27



not included ?

Effort of the search

Height of a B+ tree of order k with *data block size* b (at least b, max 2b elements) is maximum logk+1(n/b).

Area query possible

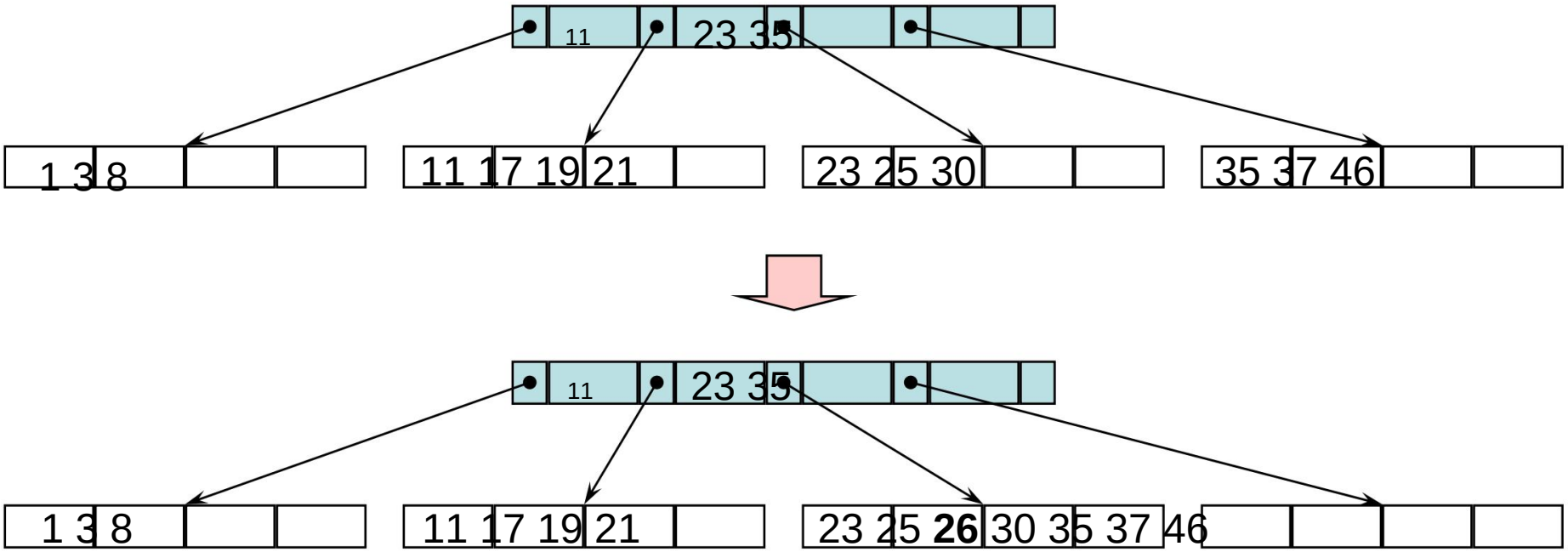
Find all elements in the range [lower limit, upper limit]

B+ tree insertion (1)

Inserting here means that a data record with identifying Key is inserted into an external node, which may require index entries to be adjusted

Case 1: Insert record with key 26

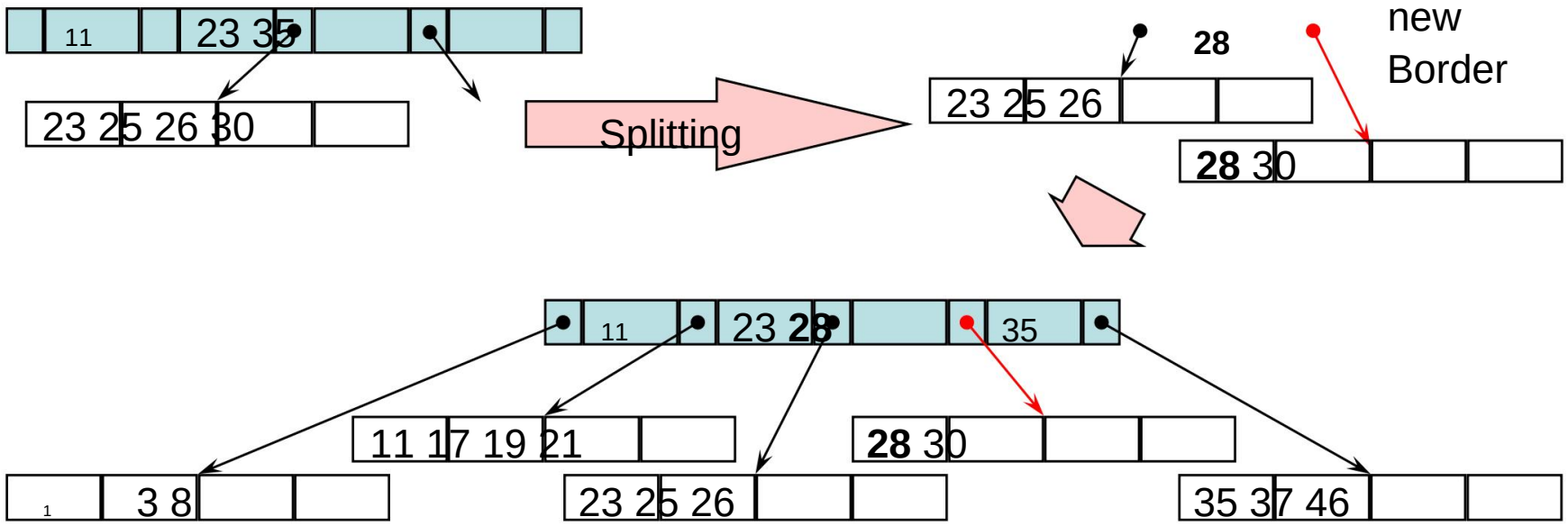
Space available in the external node, easy insertion



B+ tree insertion (2)

Case 2: Insert record with key 28

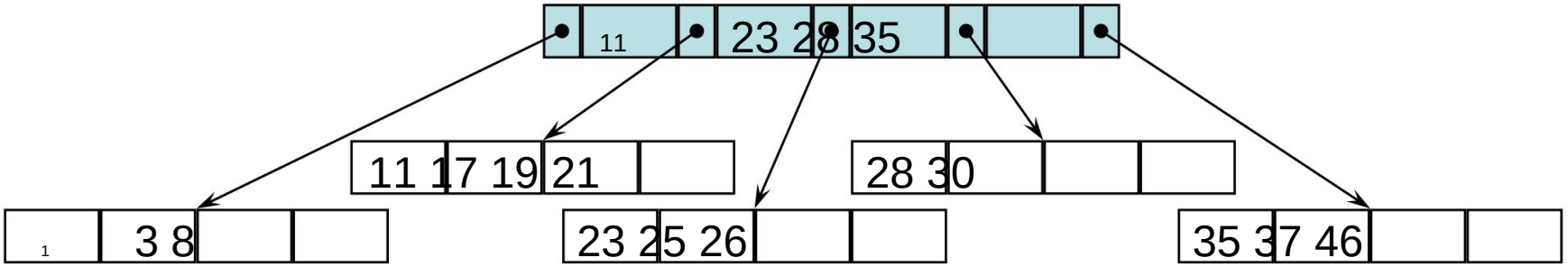
No more space in the external node **(overflow, overflow),** external node must be split ÿ **Split**



B+ tree insertion (3)

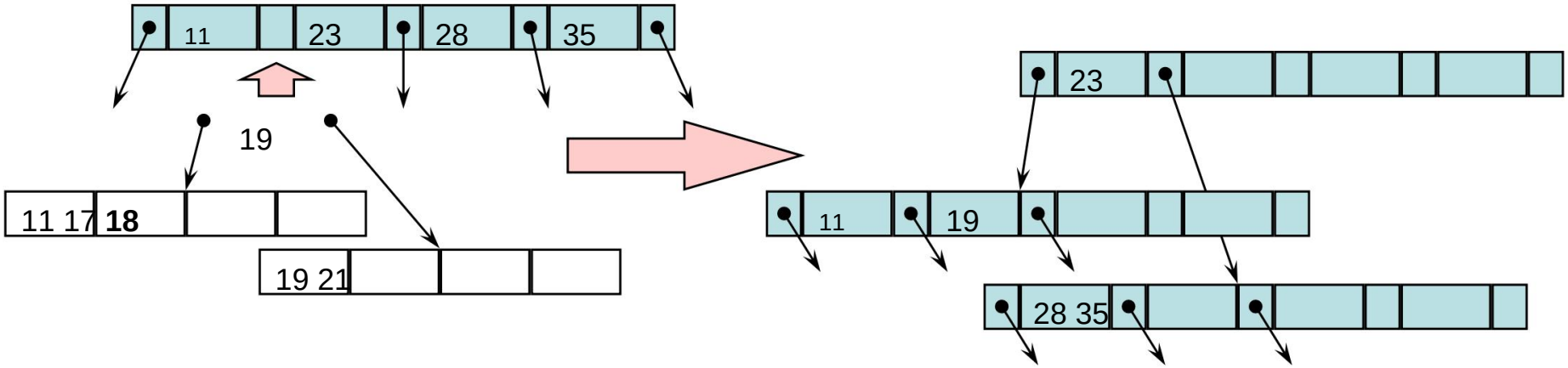
Case 3: Insert record with key 18

No more room in the knot and no more room in the index node above, index node must be split



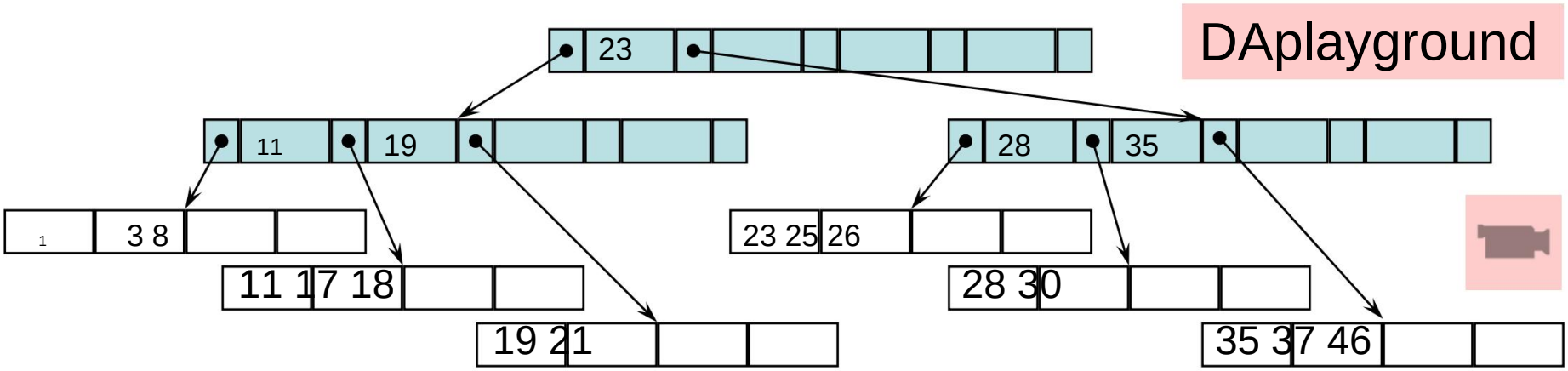
B+ tree insertion (4)

When splitting an index node, the middle index element is entered in the index node above it. If it does not exist *(root split)*, a new root is created



B+ tree insertion (5)

The resulting tree has the following appearance



Differentiation of the split for external and internal nodes

External nodes: 2k+1 elements are divided between 2 neighboring external nodes

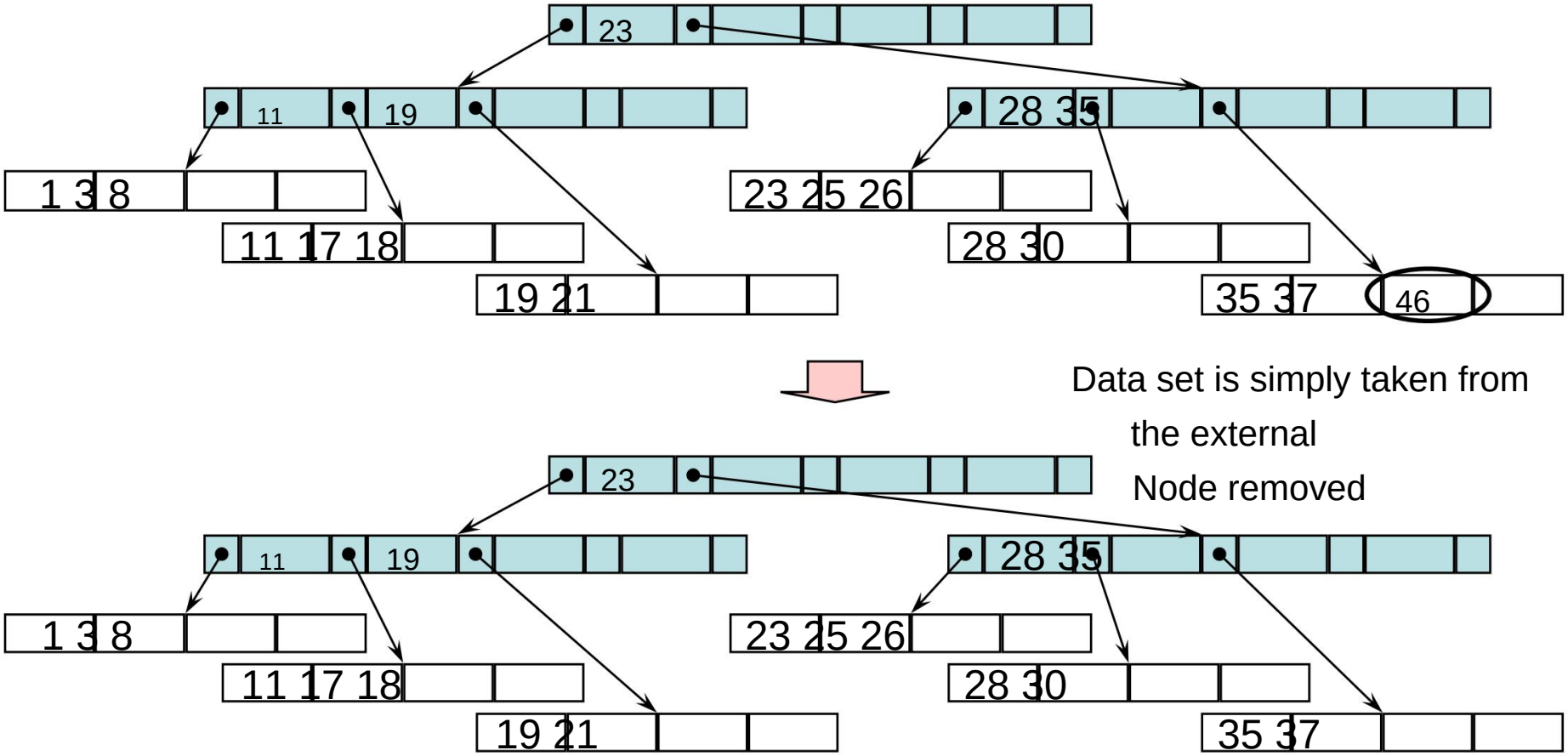
Internal nodes: 2k+1 index elements (key values) are divided into 2 neighboring index nodes with k elements each, the middle index element is entered into the index level above.

If no level above exists, a new root is created, the tree grows by one level (“tree grows from the leaves to the root”), the access path to the data increases by 1.

B+ Tree Delete (1)

Case 1: Remove record with key 46

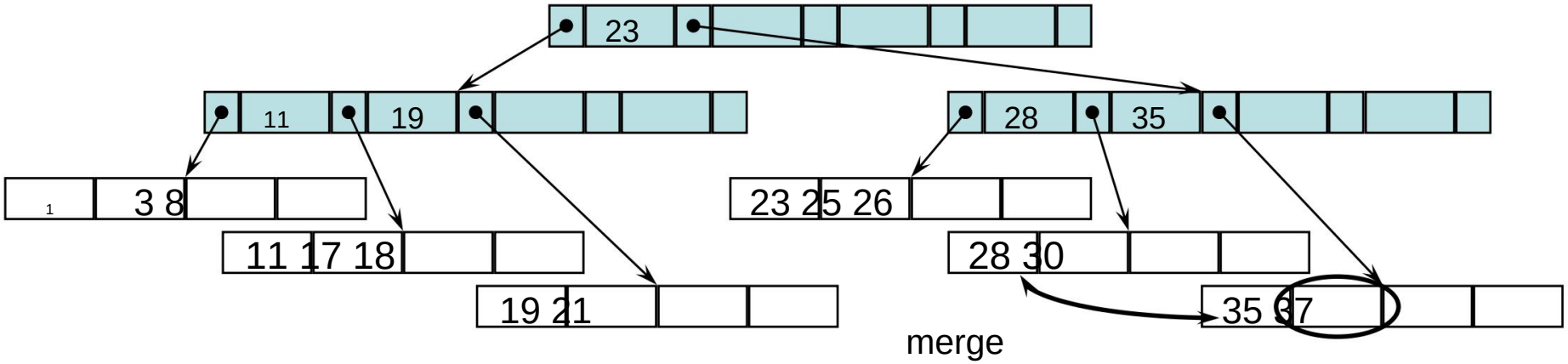
External node not understaffed after deletion



B+ Tree Delete (2)

Case 2: Remove record with key 37

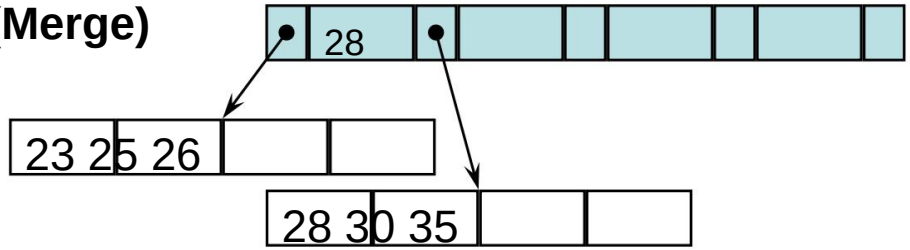
External node understaffed after deletion



Element is removed, node is underflow , number of elements < k

Reverse process to split ÿ neighboring nodes are **merged**

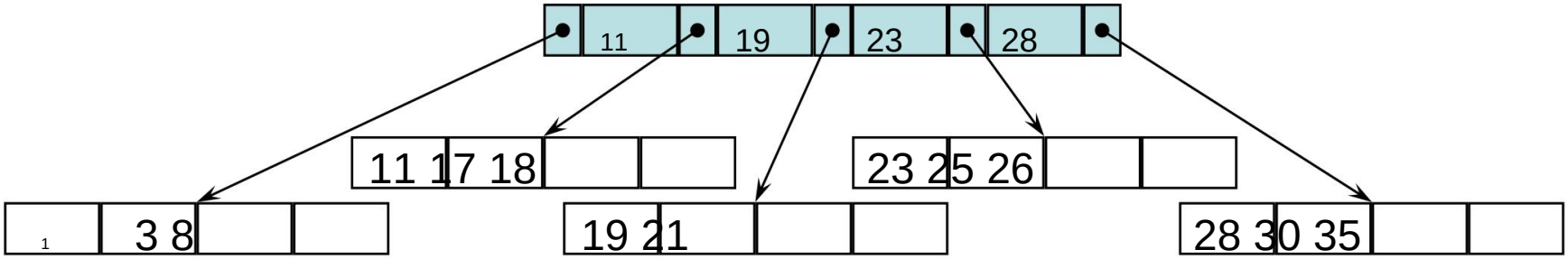
(Merge)



Internal node understaffed, also needs to be merged with neighbors.

B+ Tree Delete (3)

Merging can lead to tree height reduction, 2 nodes are merged to form new root, old root is removed.

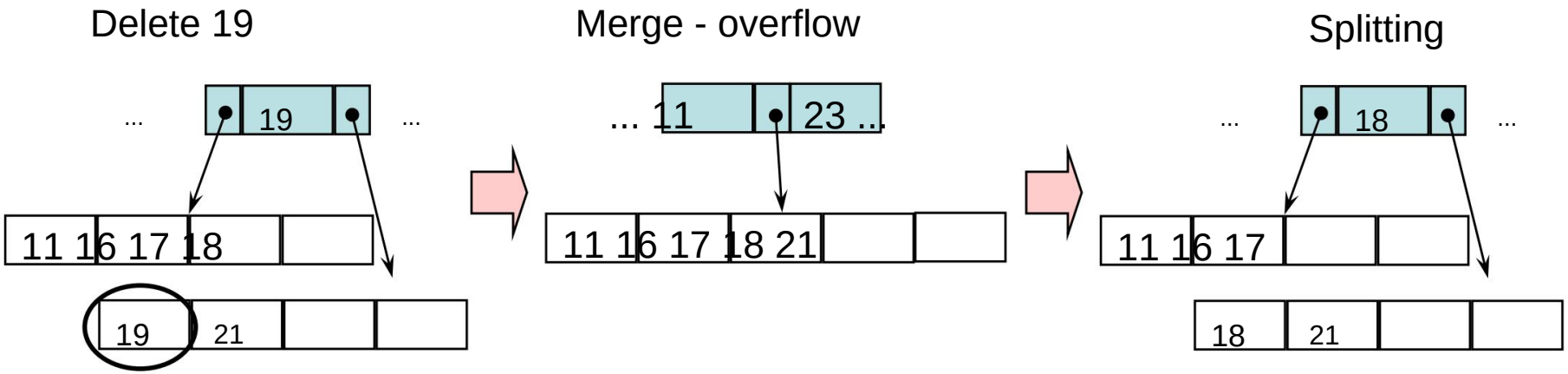


B+ Tree Delete (4)

annotation

Merging two nodes can lead to an overflow (analog Insertion) of the new node, which makes a subsequent split necessary.

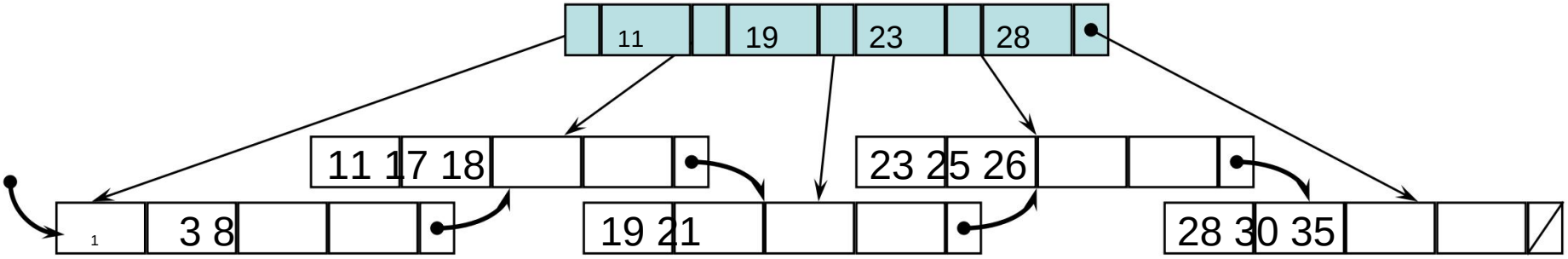
Example: (tree section)



This (merge-split) sequence produces a better division of records between neighboring nodes. This is also referred to here (see 2-3-4 tree, actually incorrectly) as a data record shift.

Data block chaining

In practice, the external nodes (data blocks) linearly chained to allow efficient, sequential access in the sort order of stored elements



Analysis B+ - Tree

- Data management
  - Insert and delete supported
- Unlimited
  - data volume
    - depending on the size of the available storage space
- Models
  - **External storage oriented**
  - Support complex operations
    - Range queries, sort order

Analysis B+ - Tree (2)

Storage space	O(n)
Split, merge	O(1)
access	O(log n)
Insert	O(log n)
Delete	O(log n)
Sorting order	O(n)

Guaranteed!

Please note: Order of the tree is a constant factor!

Because of these properties and the focus on external storage media, the B+ tree is very good for

Suitable for **use in database systems**

A trie is a digital search tree or prefix tree

Used to store and search character strings

Management of dictionaries, telephone books, spell checkers, etc.

The name is derived from “retrieval” but is pronounced like “try”.

For k letters, a “k+1-ary tree”, where each node is separated by a

Table is represented with k+1 edges on children

A path starting from the root to a leaf in the shoot represents a character string

Can also be interpreted in the broadest sense as an interval-based search data structure , since intervals [A,B[, [B,C[, ...

Special shapes

Patricia Tree, from the Briandais Tree

Example Trie

Trie

TEN

THE

THEN

THIN

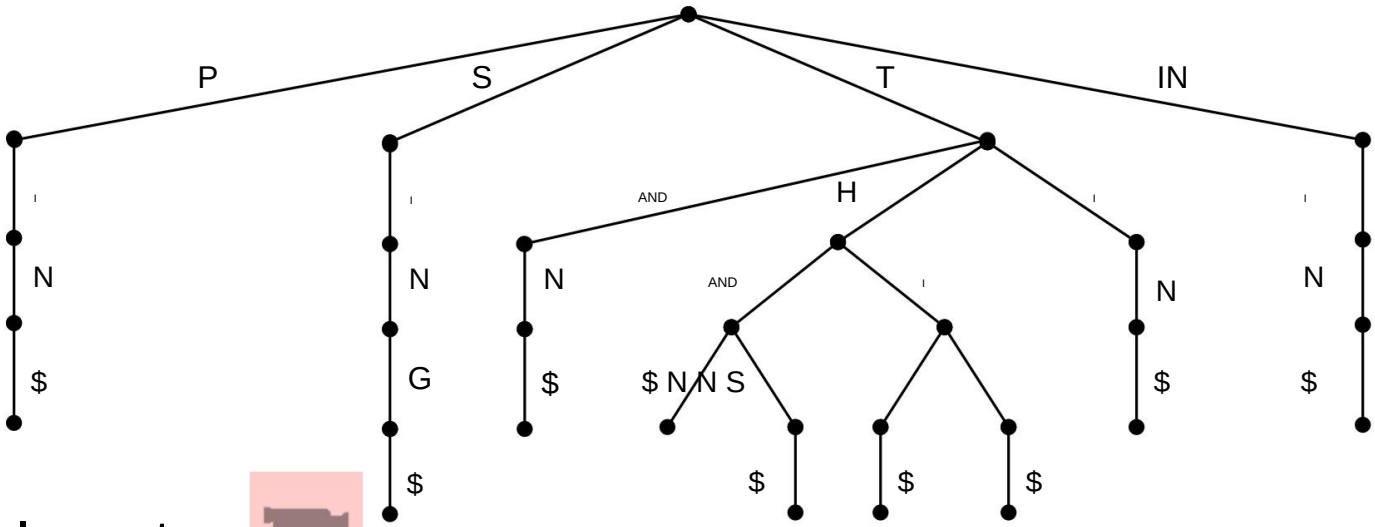
THIS

BELIEVE

SING

PIN

WIN



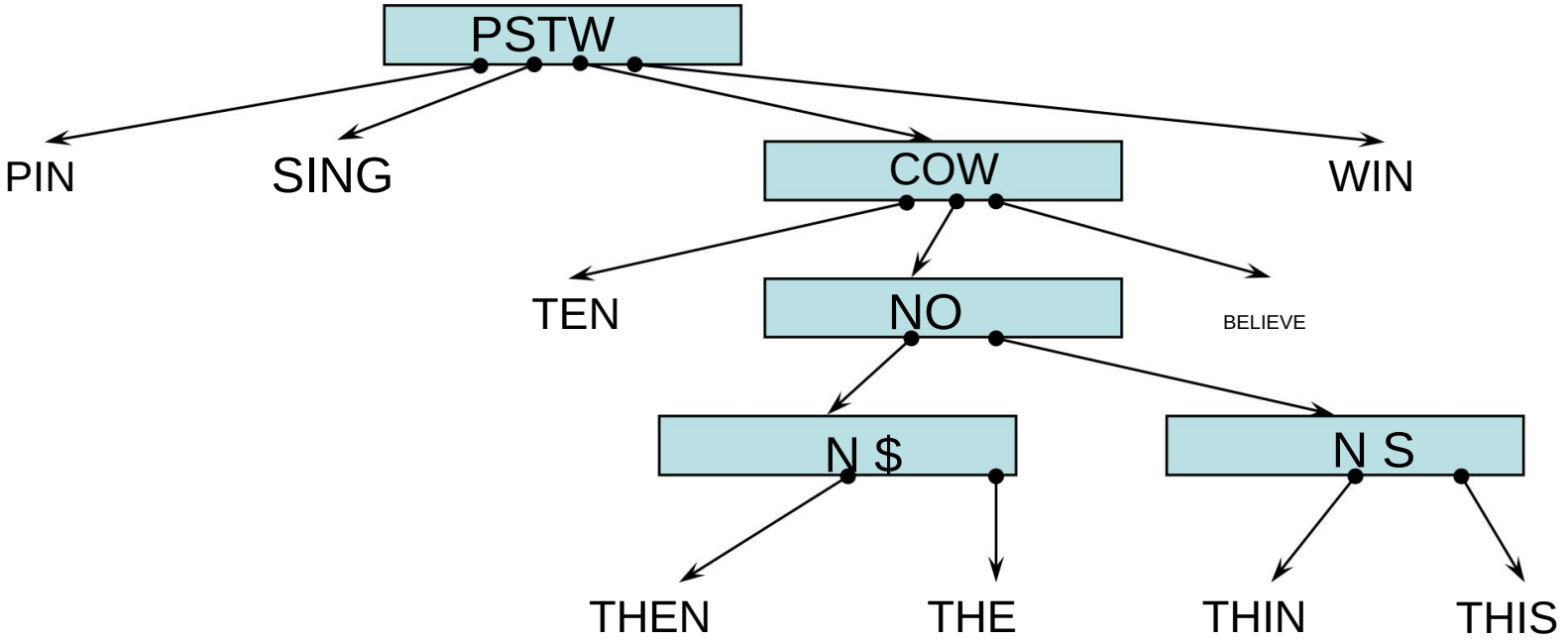
Insert

Seek

universität wien

Practical Algorithm to Retrieve Information Coded in Alphanumeric

The aim is to compress the trie

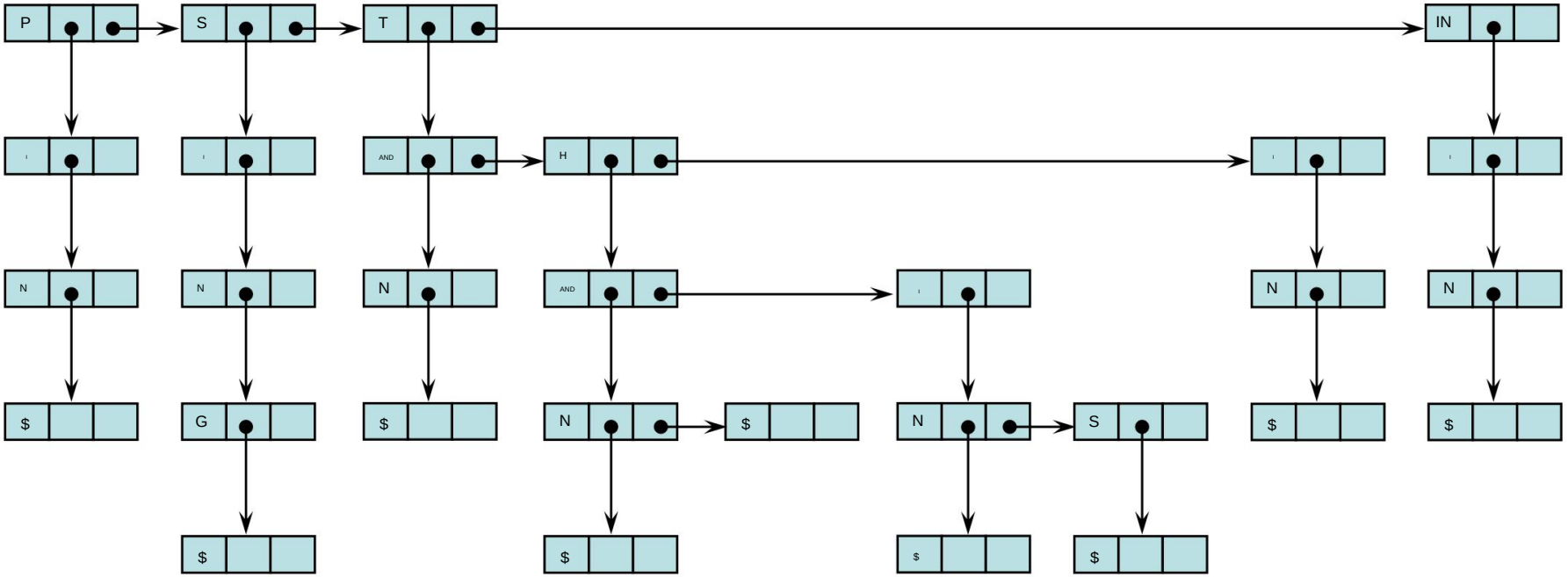


List representation of a trie

instead of the tables in the linear list node

Node consists of 2 child components

Next value - next level



Sort Analysis

Data management
- Insert and delete supported
- Structure of the trie independent of the insertion order

Amount of data
- unlimited

Models
- Main memory oriented
- Support complex operations
  - Range queries, sort order

Duration

Storage space	O(n)
access	O(log n)
Insert	O(log n)
Delete	O(log n)
Sorting order	O(n)

For the exact calculation, the base of the logarithm depends. on the cardinality of the character set, e.g. 26 (upper case letters), 2 (bit strings)

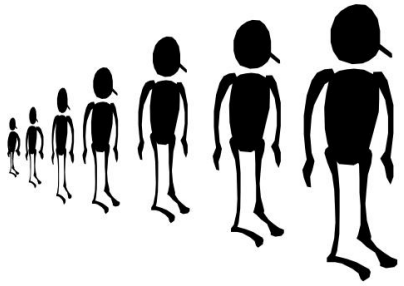
5.8 Priority Queues

Data structure for repeating

Find and remove the element

with the smallest (or largest)

Key value from a lot of applications



- Simulation systems (key values represent execution times of events)
- Process management (priorities of the processes)
- Numerical calculations (key values correspond to the calculation errors, with the big ones being eliminated first)
- Basis for a number of complex algorithms (graph theory, Filekompression, etc.)
  - Note: In the following we only consider priority queues that have the Support access to the smallest elements

The operation

Construct

Creating an empty priority queue

IsEmpty

Query for empty priority queue

Insert

Inserting an element

FindMinimum

Returning the smallest element

DeleteMinimum

Delete the smallest element

Implementation approaches

Unordered list

- Elements are entered into the list arbitrarily (cost O(1)).
- When accessing or deleting the 'smallest' element, the list must effort ÿ O(n).

Ordered list

- Elements are entered in the order of their size (effort O(n)).
- Access or deletion constant effort ÿ O(1)

Problem

- Effort O(n) difficult to avoid.

Priority queue as a tree structure

Balanced key trees

- Insertion in the balanced key tree requires O(log n) effort
- Realize access by following the leftmost path in the tree (to the smallest element) ÿ effort O(log n)

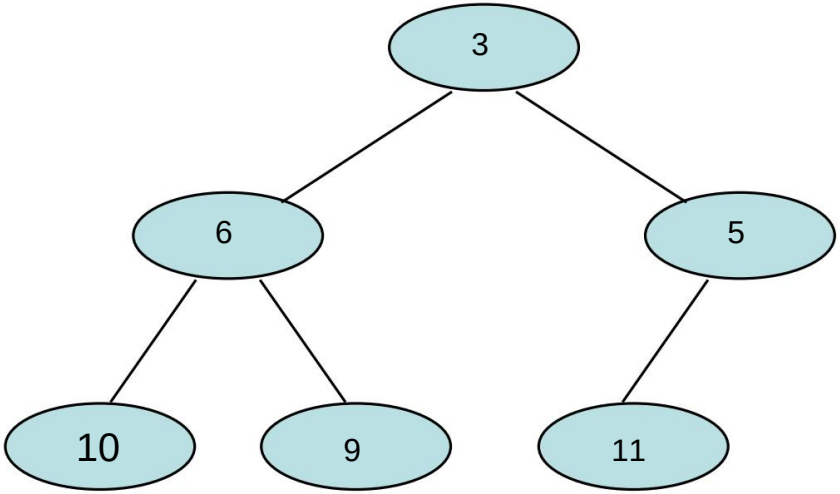
Cheapest implementation via

- unordered, complete key trees
- with the property that for all nodes the key values of their Successors are larger (or smaller).

5.8.1 Heap

set of values

3 6 5 10 9 11



Heap

Unordered, binary, complete key tree

Value of each node is less than (larger) or equal to the values of its successor nodes

Root contains smallest (largest) element

Arrangement of the subtrees in relation to their root undefined (unordered Baum)

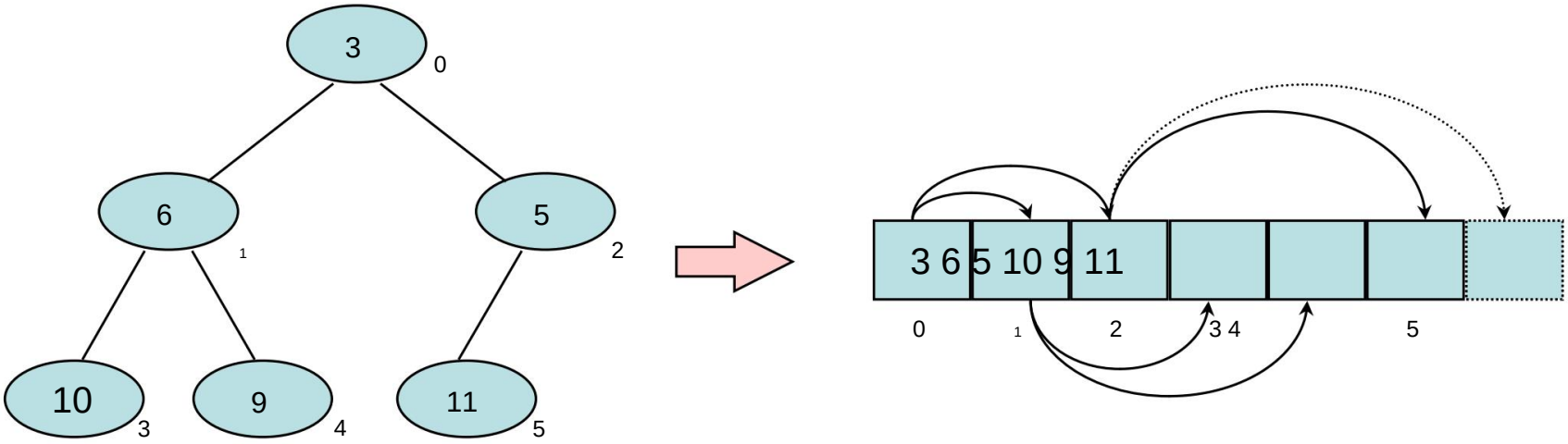
Heap data structure

Realization of a heap efficiently through a field

The n key values of the heap can be interpreted as a sequence of elements x0 , x1 , x2 , ... xn-1 , where the position of each node value in the field is determined by the following rule:

Root in position 0, the children of node i are stored at position 2i+1 and 2i+2.

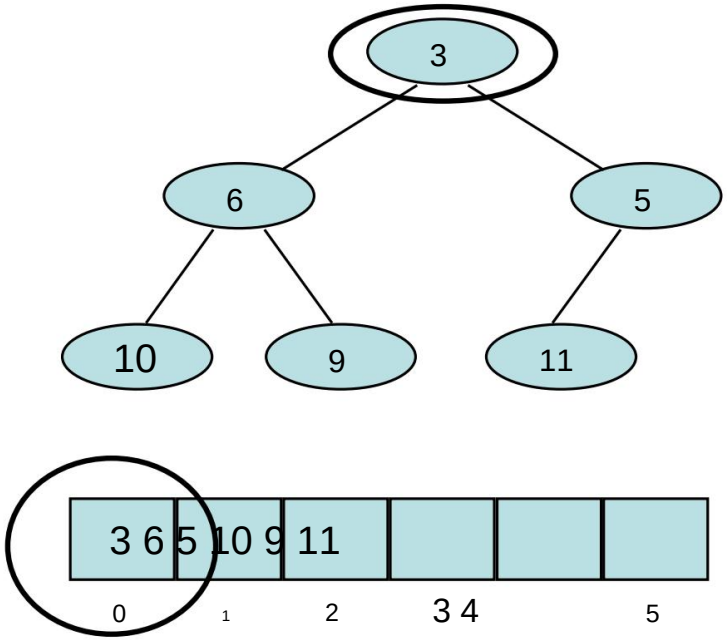
Example



access

Accessing the smallest element with constant effort: O(1)

ÿ Root = first element in the field.



Class Priority Queue

```
class PQ {
    int *a, N;
public:
    PQ(int max) { a = new int[max]; N = -1; }
    ~PQ() { delete[] a; }
    int FindMinimum(){ return a[0]; }
    int IsEmpty() { return (N == -1); } void Insert(int); int
    DeleteMinimum();

    ...
};
```

Insert into a heap

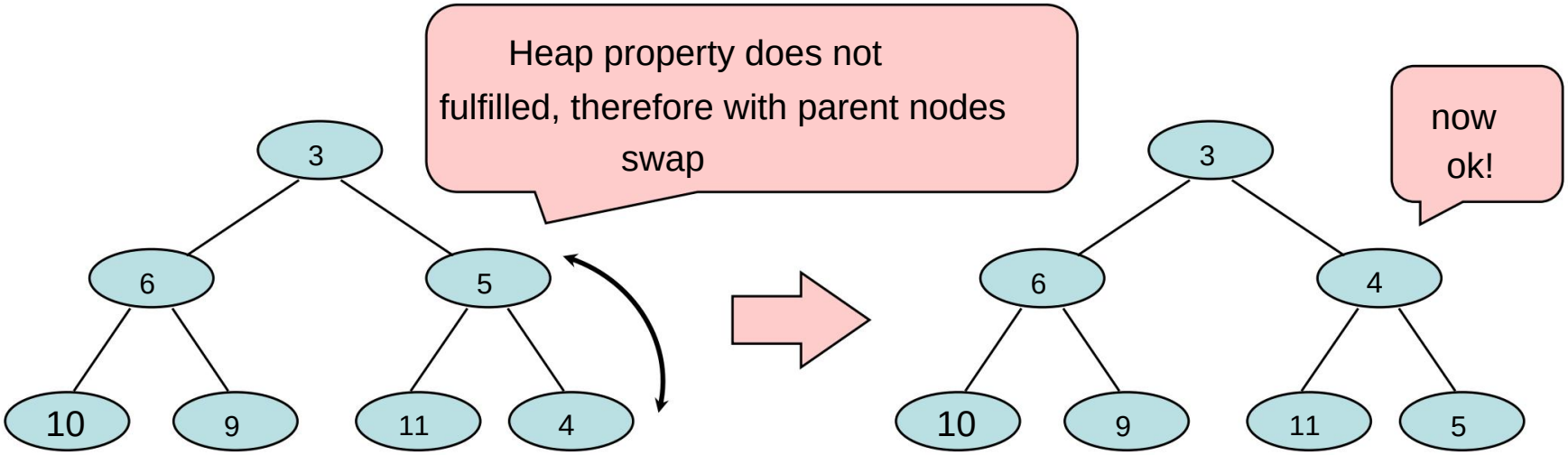
Enter new element at the last position in the field

Check whether the heap property is fulfilled

If not, swap with parent node and repeat until fulfilled

Enter 4 (at the last position in the field)

Effort: O(log n)



Method Insert

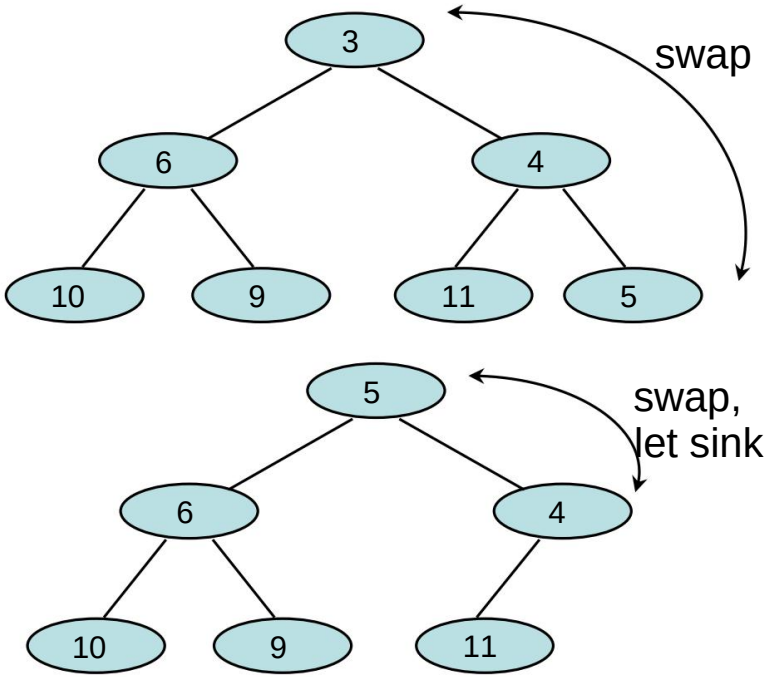
```
void PQ::Insert(int v) {
    int child, parent; a[++N] =
    v;
    child = N;
    parent = (child - 1)/2; while(child !
    = parent) {
        if(a[parent] > a[child]) {
            swap(a[parent], a[child]); child =
            parent; parent =
            (parent - 1)/2; } else break; // to stop
        the loop
    }
}
```

Delete from a heap

Swap root with rightmost leaf node, this one

Delete leaf node, let root sink into the tree (swap with smaller child node) until Heap property applies.

Effort: O(log n)



Method DeleteMinimum

```
int PQ::DeleteMinimum() {
    int parent = 0, child = 1; int v = a[0];
    a[0] = a[N];

    N--;
    while(child <= N) { if(a[child]
        > a[child+1]) child++; if(a[child] < a[parent]) {

            swap(a[parent], a[child]); parent =
            child; child = 2*child
            + 1; } else break; // to stop
        the loop

    } return v;
}
```

Analyse Heap

Data management

- Insert and delete supported

Amount of data

- unlimited

Models

- Main memory oriented
- Just simple operations

Duration

Storage space	O(n)
Access (minimum/maximum)	O(1)
Insert	O(log n)
Delete	O(log n)

The heap provides an efficient Basic data structure for many others based on it data structures

Heaps und Heaps

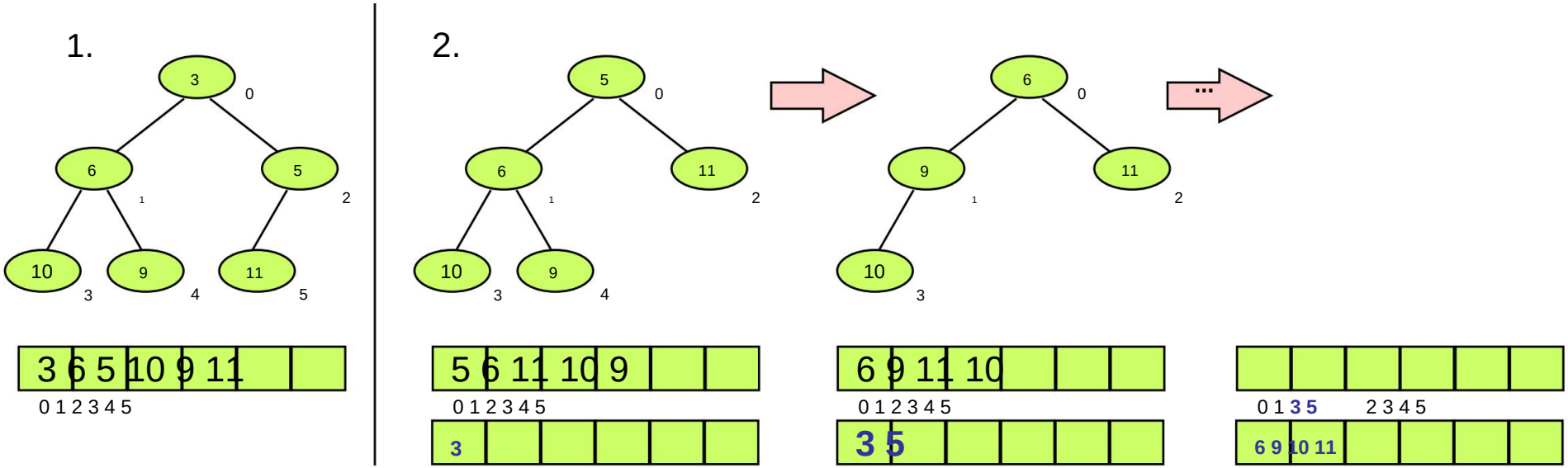
Please differentiate! The term heap is used in computer science to describe various concepts

- Heap as a data structure for storing priority queues
- Heap as a memory area for managing dynamically allocated memory areas (managed by the user).
- Heap as a form of storage in databases

5.8.2 Heapsort

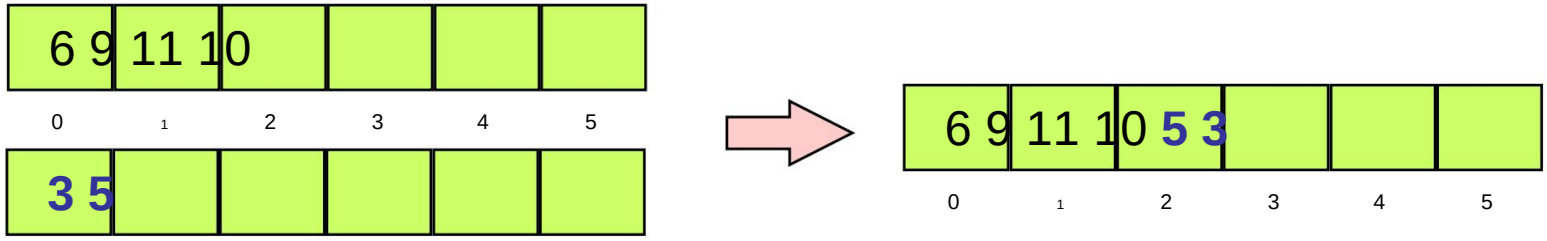
A heap (priority queue) can form the basis for sorting (Williams 1964):

1. Insert one element at a time into a heap
2. the smallest or largest element is successively removed
   the elements are delivered in ascending or descending order

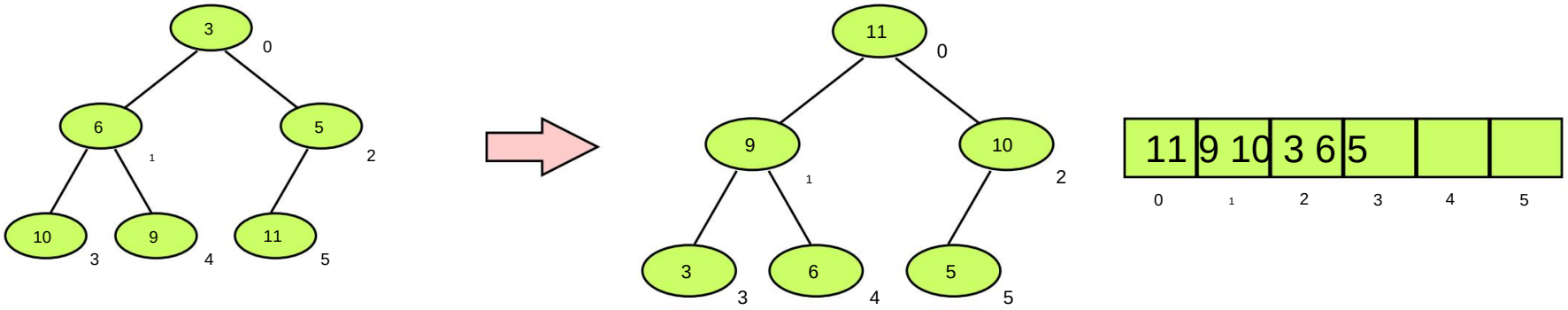


Heapsort (modified approach)

Combining the two arrays, avoiding duplicate Storage space



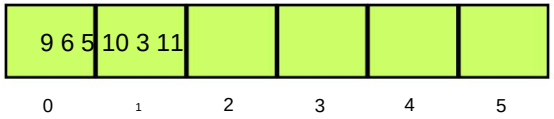
Flip heap property: Value of each node is greater than or equal to the values of its children nodes

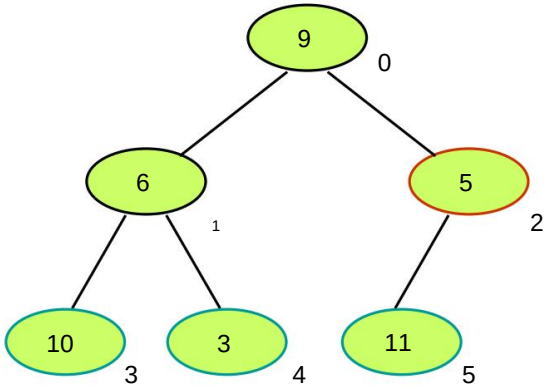


Example: buildheap

set of values

9 6 5 10 3 11





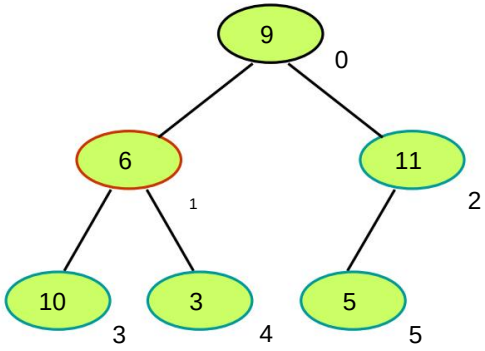
Idea: Heap property generate; for leaf nodes

trivially fulfilled

No. 2 first to be examined

Knot, 11 does not fit, in let tree sink, ie

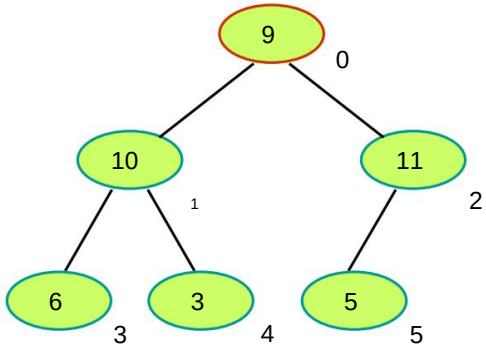
Swap 5 for 11



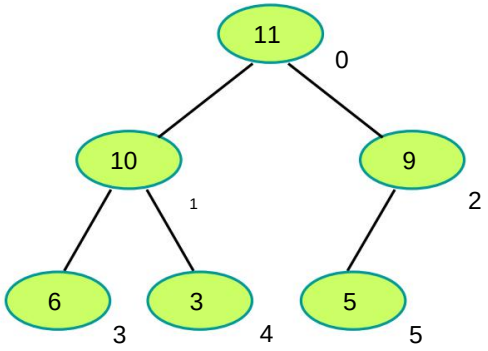
Value 6 (No. 1) does not fit, drop into subtree, ie

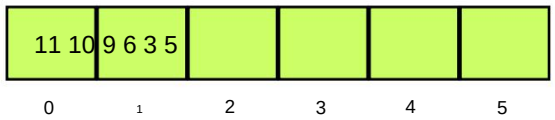
with larger the successor

swap, i.e. 6 with 10

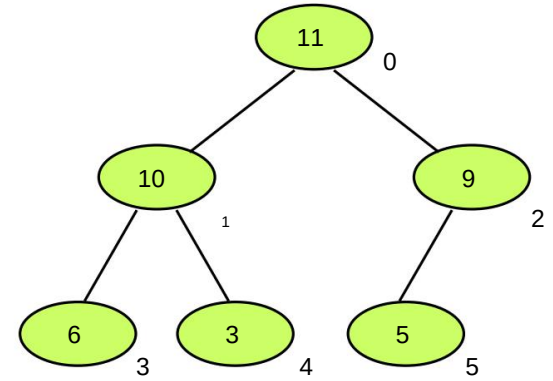


No. 0 (9) sink into tree. Heap property fulfilled!

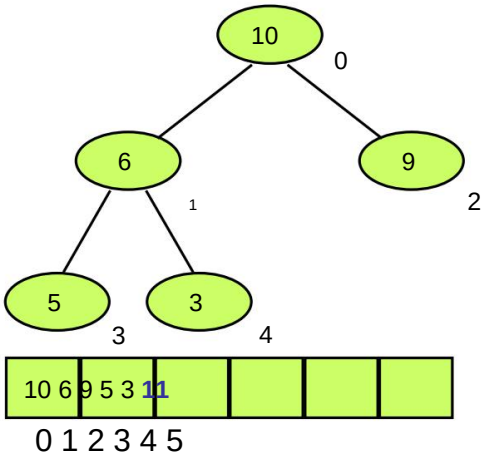


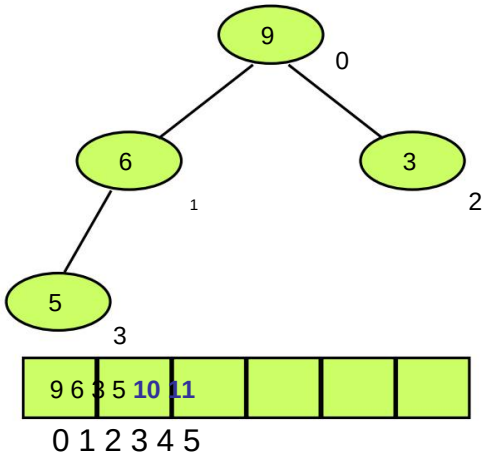


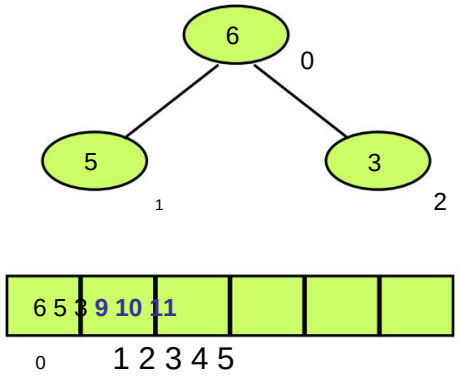
Example: heapsort

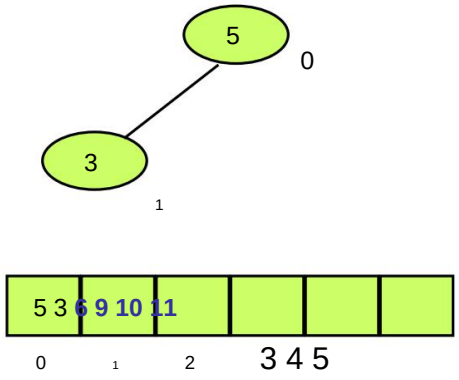


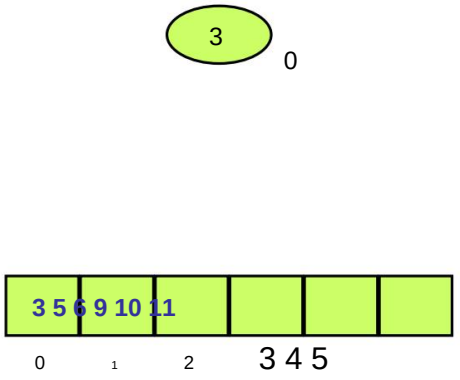
Idea: “Delete” the maximum and save it in the space that becomes free











Heapsort algorithm (1)

```
void Vector::Heapsort() {
    int heapsize = Length();
    BuildMaxheap();
    for(int i = Length()-1; i >= 1; i--) { swap(a[0], a[i]);
        heapsize--; heapify(0,
        heapsize);


    }
}

void Vector::BuildMaxheap() { for(int i =
    Length()/2 - 1; i >= 0; i--)
        heapify(i, Length());
}
```

Heapsort algorithm (2)

```
void Vector::heapify(int i, int heapsize) {
    int left = 2*i + 1; int right =
    2*i + 2; int largest; if (left <
    heapsize &&
    a[left] > a[i]) largest = left;

    else
        largest = i; if
    (right < heapsize && a[right] > a[largest])
        largest = right; if
    (largest != i) { swap(a[i],
        a[largest]); heapify(largest,
        heapsize);
    }
}
```

Quicksort

- Degeneracy to O(n2) possible
  - Choice of pivot element!

Merge sort

- always runs time O(n *
- log(n)) twice the storage space required

Heapsort

- none of the above
- disadvantages but higher constant effort

What do we take with us?

Tree structures

- Notation
- spec. Properties, from O(n) to O(log n)

Search trees

- Binary search trees
- AVL trees
- 2-3-4 trees
- B+ tree
- Balancing

Trie

Priority Queues

- Heap