Annotation

- Directive `forward`
- Standard units,
- Pointers.

Directive `forward`

- It is typical that one function calles another but

Directive `forward`

- It is typical that one function calles another but
- sometimes the latter function calls the former, too.

Directive `forward`

- It is typical that one function calles another but
- sometimes the latter function calls the former, too.
- Problem: In Pascal we have to define first (then we may use).

Directive `forward`

- It is typical that one function calles another but
- sometimes the latter function calls the former, too.
- Problem: In Pascal we have to define first (then we may use).
- Cyclic dependence seems unsolvable...

Directive `forward`

- It is typical that one function calles another but
- sometimes the latter function calls the former, too.
- Problem: In Pascal we have to define first (then we may use).
- Cyclic dependence seems unsolvable...
- until we find the `forward` directive!

Directive `forward`

- It is typical that one function calles another but
- sometimes the latter function calls the former, too.
- Problem: In Pascal we have to define first (then we may use).
- Cyclic dependence seems unsolvable...
- until we find the `forward` directive!
- This directive is placed after the function prototype:

Directive `forward`

- It is typical that one function calles another but
- sometimes the latter function calls the former, too.
- Problem: In Pascal we have to define first (then we may use).
- Cyclic dependence seems unsolvable...
- until we find the `forward` directive!
- This directive is placed after the function prototype:
- `procedure two(a:integer);forward;`

Forward example:

```
program qq;
procedure two(a:integer);forward;
procedure one(a:integer);
begin
      two(a);
end;
procedure two(a:integer);
begin
      one(a);
end;
begin
      one(1);
      {Let us ignore that this program does
not make a good sense!}
end
```

Linear list typology

- circular (instead of `nil` point at the first)

Linear list typology

- circular (instead of `nil` point at the first)
- with a head (first element is not a member)

Linear list typology

- circular (instead of `nil` point at the first)
- with a head (first element is not a member)
- with a tail (last element is not a member)

Linear list typology

- circular (instead of `nil` point at the first)
- with a head (first element is not a member)
- with a tail (last element is not a member)
- without head/tail

Linear list typology

- circular (instead of `nil` point at the first)
- with a head (first element is not a member)
- with a tail (last element is not a member)
- without head/tail
- bidirectional (pointers `next` and `prev`).

A Queue and a Buffer

- Queue is a data structure organizing the elements in a FIFO-way,

A Queue and a Buffer

- Queue is a data structure organizing the elements in a FIFO-way,
- it is equipped with functions `enqueue` and `dequeue`.

A Queue and a Buffer

- Queue is a data structure organizing the elements in a FIFO-way,
- it is equipped with functions `enqueue` and `dequeue`.
- Buffer is a data structure organizing the elements in a LIFO-way,

A Queue and a Buffer

- Queue is a data structure organizing the elements in a FIFO-way,
- it is equipped with functions `enqueue` and `dequeue`.
- Buffer is a data structure organizing the elements in a LIFO-way,
- it is equipped with functions `push` and `pop` (or `pull`).

A Queue and a Buffer

- Queue is a data structure organizing the elements in a FIFO-way,
- it is equipped with functions `enqueue` and `dequeue`.
- Buffer is a data structure organizing the elements in a LIFO-way,
- it is equipped with functions `push` and `pop` (or `pull`).
- It is possible to implement them using array,...

A Queue and a Buffer

- Queue is a data structure organizing the elements in a FIFO-way,
- it is equipped with functions `enqueue` and `dequeue`.
- Buffer is a data structure organizing the elements in a LIFO-way,
- it is equipped with functions `push` and `pop` (or `pull`).
- It is possible to implement them using array,...
- but it is much better to use linear lists!

Buffer

Implementation I/III

```
type pbuf=^buf;
buf=record
     val:integer;
     next:pbuf;
end;
var head:pbuf;
procedure init;
begin head:=nil;
end;
```

Buffer

Implementation II/III

```
type pbuf=^buf;
buf=record
     val:integer;
     next:pbuf;
end;
var head:pbuf;
procedure push(what:integer);
var tmp:pbuf;
begin
     new(tmp);
     tmp^.val:=what;
     tmp^.next:=head;
     head:=tmp;
end;
```

Buffer

Implementation III

```
function pop:integer;
var tmp:pbuf;
begin
     tmp:=head;
     if head<>nil then
     begin pop:=head^.val;
          head:=tmp^.next;
          dispose(pom);
     end else
     begin writeln('Error!');
          pop:=-1;
     end;
```

Queue

Implementation

```
type=pq=^queue;
queue=record
     val:integer;
     next:pq;
end;
var head,tail:pq;
procedure init;
begin
     head:=nil;      tail:=nil; end;
```

```
procedure enqueue(what:integer);
var tmp:pq;
begin if head=nil then
     begin new(head);
          tail:=head;
          head^.next:=nil;
          head^.val:=what;
     end else
     begin new(tmp);
          tmp^.next:=nil;
          tmp^.val:=what;
          head^.next:=tmp;
          head:=tmp;
     end;
end;
```

```
function dequeue:integer;
var tmp:pq;
begin if head=nil then
      begin dequeue:=-1;
      end else
      begin if head=tail then
            begin dequeue:=tail^.val;
                  dispose(tail);
                  head:=nil; tail:=nil;
            end else
            begin dequeue:=tail^.val;
                  tmp:=tail;
                  tail:=tail^.next;
                  dispose(tmp);
            end;
      end;
end;
```

Switch two neighboring elements

Switch an element in a linear list with its neighbor

```
procedure swap(var head:ll;what:ll);
var tmp:ll;
begin tmp:=head;
      if head=what then
      begin head:=head^.next;
            tmp^.next:=head^.next;
            head^.next:=tmp;
      end else
      begin while(tmp^.next<>what) do
                  tmp:=tmp^.next;
            tmp^.next:=what^.next;
            what^.next:=tmp^.next^.next;
            tmp^.next^.next:=what;
end; end;
```

Dynamic data structures

- The examples sometimes omit singularities (empty list, an element not in the list, one-element-list...). All this would be implemented by several tests for `nil`.

Dynamic data structures

- The examples sometimes omit singularities (empty list, an element not in the list, one-element-list...). All this would be implemented by several tests for `nil`.
- Good exercise: Bubblesort over linear list.

Dynamic data structures

- The examples sometimes omit singularities (empty list, an element not in the list, one-element-list...). All this would be implemented by several tests for `nil`.
- Good exercise: Bubblesort over linear list.
- Organizing (an ordered) linear list (functions insert, delete and member that are working with the ordered linear list).

Ordered list

- A linear list may be ordered (with respect to the values of the elements, w.l.o.g. in a non-decreasing order).

Ordered list

- A linear list may be ordered (with respect to the values of the elements, w.l.o.g. in a non-decreasing order).
- For such lists we usually implement functions:

Ordered list

- A linear list may be ordered (with respect to the values of the elements, w.l.o.g. in a non-decreasing order).
- For such lists we usually implement functions:
    - `member` – says whether an element with an appropriate key is in the list,

Ordered list

- A linear list may be ordered (with respect to the values of the elements, w.l.o.g. in a non-decreasing order).
- For such lists we usually implement functions:
    - `member` – says whether an element with an appropriate key is in the list,
    - `insert` – inserts an element into a list,

Ordered list

- A linear list may be ordered (with respect to the values of the elements, w.l.o.g. in a non-decreasing order).
- For such lists we usually implement functions:
  - `member` – says whether an element with an appropriate key is in the list,
  - `insert` – inserts an element into a list,
  - `delete` – deletes an element from a list.

Ordered list

- A linear list may be ordered (with respect to the values of the elements, w.l.o.g. in a non-decreasing order).
- For such lists we usually implement functions:
  - `member` – says whether an element with an appropriate key is in the list,
  - `insert` – inserts an element into a list,
  - `delete` – deletes an element from a list.
- Example – see webpage (or we are going to write it here).

Further data structures

- Self-organizing lists – lists that get modified by accessing them.

Further data structures

- Self-organizing lists – lists that get modified by accessing them.
- Move-front rule, transposition rule:

Further data structures

- Self-organizing lists – lists that get modified by accessing them.
- Move-front rule, transposition rule:
- When accessing a member, we move it to the beginning or change with its (immediate) predecessor, respectively.

Further data structures

- Self-organizing lists – lists that get modified by accessing them.
- Move-front rule, transposition rule:
- When accessing a member, we move it to the beginning or change with its (immediate) predecessor, respectively.
- Idea: Usually we are accessing the same element repeatedly (in a short time) but our interests are changing.

Trees

- In a linear list, it is a problem to search an element.

Trees

- In a linear list, it is a problem to search an element.
- It takes a linear time, we want something better.

Trees

- In a linear list, it is a problem to search an element.
- It takes a linear time, we want something better.
- We want to implement a data structure where binary search is possible.

Trees

- In a linear list, it is a problem to search an element.
- It takes a linear time, we want something better.
- We want to implement a data structure where binary search is possible.
- Natural idea is to create a binary search tree (smaller values in the left subtree, larger in the right one).

Trees

- In a linear list, it is a problem to search an element.
- It takes a linear time, we want something better.
- We want to implement a data structure where binary search is possible.
- Natural idea is to create a binary search tree (smaller values in the left subtree, larger in the right one).
- How to implement it?

Trees

- In a linear list, it is a problem to search an element.
- It takes a linear time, we want something better.
- We want to implement a data structure where binary search is possible.
- Natural idea is to create a binary search tree (smaller values in the left subtree, larger in the right one).
- How to implement it?
- Each element gets more than one ancestor (left, right).

Tree representation

in Pascal

```
type tree=^vertex;
     vertex=record
          val:longint;
          left:tree;
          right:tree;
          ...
     end;
```

Binary search trees

- Binary tree is such a tree where each element has at most two ancestor.

Binary search trees

- Binary tree is such a tree where each element has at most two ancestor.
- Binary search tree is a binary tree which for each element with a key K contains in the left subtree values with key smaller than K and in the right subtree values with key larger than K.

Binary search trees

- Binary tree is such a tree where each element has at most two ancestor.
- Binary search tree is a binary tree which for each element with a key K contains in the left subtree values with key smaller than K and in the right subtree values with key larger than K.
- Thus it is possible to search efficiently in such a tree. Advantages/disadvantages?

Binary search trees

- Binary tree is such a tree where each element has at most two ancestor.
- Binary search tree is a binary tree which for each element with a key K contains in the left subtree values with key smaller than K and in the right subtree values with key larger than K.
- Thus it is possible to search efficiently in such a tree. Advantages/disadvantages?
- If we build it well, it becomes more efficient than a linear-list.

Binary search trees

- Binary tree is such a tree where each element has at most two ancestor.
- Binary search tree is a binary tree which for each element with a key K contains in the left subtree values with key smaller than K and in the right subtree values with key larger than K.
- Thus it is possible to search efficiently in such a tree. Advantages/disadvantages?
- If we build it well, it becomes more efficient than a linear-list.
- If we build it badly, it collapses into a linear-list.

Binary search trees

- Binary tree is such a tree where each element has at most two ancestor.
- Binary search tree is a binary tree which for each element with a key K contains in the left subtree values with key smaller than K and in the right subtree values with key larger than K.
- Thus it is possible to search efficiently in such a tree. Advantages/disadvantages?
- If we build it well, it becomes more efficient than a linear-list.
- If we build it badly, it collapses into a linear-list.
- How to build a balanced binary search tree (and how to keep the tree balanced)?

Binary search trees

- Binary tree is such a tree where each element has at most two ancestor.
- Binary search tree is a binary tree which for each element with a key K contains in the left subtree values with key smaller than K and in the right subtree values with key larger than K.
- Thus it is possible to search efficiently in such a tree. Advantages/disadvantages?
- If we build it well, it becomes more efficient than a linear-list.
- If we build it badly, it collapses into a linear-list.
- How to build a balanced binary search tree (and how to keep the tree balanced)?
- Balanced BST is a tree where for each element # elements in the left subtree (of this element) and # elements in the right subtree differ at most by 1.

Building a balanced BST

- Find a median and root it.

Building a balanced BST

- Find a median and root it.
- Build a balanced BST on smaller elements (recursively),

Building a balanced BST

- Find a median and root it.
- Build a balanced BST on smaller elements (recursively),
- build a balanced BST on larger elements (recursively),

Building a balanced BST

- Find a median and root it.
- Build a balanced BST on smaller elements (recursively),
- build a balanced BST on larger elements (recursively),
- set these trees to be sibings of the root.

BST – data structures

- We are going to build from an array (uninteresting [obvious])
- Dynamic data structure representing nodes [vertices] of the tree:

```
type pbst:^bst;
     bst=record
          val:longint;
          left:pbst;
          right:pbst;
```

Building a balanced BST

(pseudocode)

```
function build(array):pbst;
begin
     if empty(array) then build:=nil; else begin
          med:=median(array);
          small:=smaller(med,array);
          large:=larger(med,array);
          new(root);
          root^.hod:=med;
          root^.left:=build(small);
          root^.right:=build(large);
          build:=root;
     end;
end;
```

Further operations on balanced BST

`member`, `insert`, `delete`

- Operation `member` is simple:

```
function member(what:longint,where:pbst):pbst;
begin if where=nil then member:=nil
      else  if where^.val=what then member:=where
            else  if where^.val>what then
                       member:=member(where^.left)
                  else  member:=member(where^.right);
end;
```

Further operations on balanced BST

`member`, `insert`, `delete`

- Operation `member` is simple:

```
function member(what:longint,where:pbst):pbst;
begin if where=nil then member:=nil
      else  if where^.val=what then member:=where
            else  if where^.val>what then
                        member:=member(where^.left)
                  else  member:=member(where^.right);
end;
```

- Beware of the algorithm's side-effect using trichotomy (i.e., the third branch ensures that where^.val<what)

Further operations on balanced BST

`member`, `insert`, `delete`

- Operation `member` is simple:

```
function member(what:longint,where:pbst):pbst;
begin if where=nil then member:=nil
      else  if where^.val=what then member:=where
            else  if where^.val>what then
                        member:=member(where^.left)
                  else  member:=member(where^.right);
end;
```

- Beware of the algorithm's side-effect using trichotomy (i.e., the third branch ensures that where^.val<what)
- Function `insert` and `delete` are almost unimplementable (it would be necessary to destruct the whole tree).

Binary search tree

far from being balanced!

```
procedure insert(what,where);
begin {Marginal cases!}
      while((( what<where^.val) and
(where^.left<>nil)) or
            ((what>where^.val)and
(where^.right<>nil)))
            if(what<where^.val) then
where:=where^.left
            else  where:=where^.right;
      if(what=where^.val) then error("Already
there!");
      if(what<where^.val) then
      begin new(where^.left);
            kam:=where^.left;
```

BST – delete – bad version

- Find an element,

BST – delete – bad version

- Find an element,
- if it has out-degree at most 1, delete it (or bypass it).

BST – delete – bad version

- Find an element,
- if it has out-degree at most 1, delete it (or bypass it).
- With an out-degree 2,
  add its left son as the left son of the left-most element in the right subtree,
  now the erased element behaves as with out-degree 1.

BST – delete – bad version

- Find an element,
- if it has out-degree at most 1, delete it (or bypass it).
- With an out-degree 2,
  add its left son as the left son of the left-most element in the right subtree,
  now the erased element behaves as with out-degree 1.
- What's wrong?

BST – delete – bad version

- Find an element,
- if it has out-degree at most 1, delete it (or bypass it).
- With an out-degree 2,
  add its left son as the left son of the left-most element in the right subtree,
  now the erased element behaves as with out-degree 1.
- What's wrong?
- In a short time the tree looks like a linear list.

BST – delete – correct version

- Find an element,

BST – delete – correct version

- Find an element,
- With an out-degree at most 1, delete it (or bypass it).

BST – delete – correct version

- Find an element,
- With an out-degree at most 1, delete it (or bypass it).
- Otherwise find the left-most element in the right subtree and switch these elements.

BST – delete – correct version

- Find an element,
- With an out-degree at most 1, delete it (or bypass it).
- Otherwise find the left-most element in the right subtree and switch these elements.
- We violate the property of a BST for a while!

BST – delete – correct version

- Find an element,
- With an out-degree at most 1, delete it (or bypass it).
- Otherwise find the left-most element in the right subtree and switch these elements.
- We violate the property of a BST for a while!
- Now, the deleted vertex (on the incorrect location) has an out-degree at most 1 ⇒

BST – delete – correct version

- Find an element,
- With an out-degree at most 1, delete it (or bypass it).
- Otherwise find the left-most element in the right subtree and switch these elements.
- We violate the property of a BST for a while!
- Now, the deleted vertex (on the incorrect location) has an out-degree at most 1 $\Rightarrow$
- delete it (bypass).

BST – delete – correct version

- Find an element,
- With an out-degree at most 1, delete it (or bypass it).
- Otherwise find the left-most element in the right subtree and switch these elements.
- We violate the property of a BST for a while!
- Now, the deleted vertex (on the incorrect location) has an out-degree at most 1 ⇒
- delete it (bypass).
- Instead of the left-most element in the right subtree we may use the right-most element in the left subtree (as it has the closest value to the erased element). Thus both keep the pivoting properties of the erased element.

Balancedness

- Generally, it is an unpleasant problem.

Balancedness

- Generally, it is an unpleasant problem.
- Thus AVL-trees got introduced with a bit relaxed notion of balancedness.

Balancedness

- Generally, it is an unpleasant problem.
- Thus AVL-trees got introduced with a bit relaxed notion of balancedness.
- AVL-tree is a BST where for each element the depth of the left subtree differs at most by 1 from the depth of the right subtree.

Balancedness

- Generally, it is an unpleasant problem.
- Thus AVL-trees got introduced with a bit relaxed notion of balancedness.
- AVL-tree is a BST where for each element the depth of the left subtree differs at most by 1 from the depth of the right subtree.
- AVL – Adelson-Velskij and Landis.

Balancedness

- Generally, it is an unpleasant problem.
- Thus AVL-trees got introduced with a bit relaxed notion of balancedness.
- AVL-tree is a BST where for each element the depth of the left subtree differs at most by 1 from the depth of the right subtree.
- AVL – Adelson-Velskij and Landis.
- Operations `member`, `insert` and `delete` are the same as for BST, just

Balancedness

- Generally, it is an unpleasant problem.
- Thus AVL-trees got introduced with a bit relaxed notion of balancedness.
- AVL-tree is a BST where for each element the depth of the left subtree differs at most by 1 from the depth of the right subtree.
- AVL – Adelson-Velskij and Landis.
- Operations `member`, `insert` and `delete` are the same as for BST, just
- after `insert` and `delete` we perform the balance-renewing operations.

Balancedness

- Generally, it is an unpleasant problem.
- Thus AVL-trees got introduced with a bit relaxed notion of balancedness.
- AVL-tree is a BST where for each element the depth of the left subtree differs at most by 1 from the depth of the right subtree.
- AVL – Adelson-Velskij and Landis.
- Operations `member`, `insert` and `delete` are the same as for BST, just
- after `insert` and `delete` we perform the balance-renewing operations.
- For each vertex we define a value `balance` saying `depth_right - depth_left`, permitted values are `-1`, `0` and `1`.

Balance-renewing operations

- Problem appears with balance WLOG 2.

Balance-renewing operations

- Problem appears with balance WLOG 2.
- We start solving on the bottom-most level with this balance.

Balance-renewing operations

- Problem appears with balance WLOG 2.
- We start solving on the bottom-most level with this balance.
- We explore two possibilities, the remaining 2 are symmetric.

Balance-renewing operations

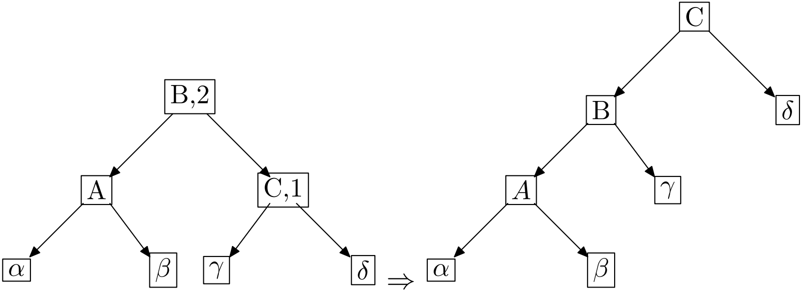
- Problem appears with balance WLOG 2.
- We start solving on the bottom-most level with this balance.
- We explore two possibilities, the remaining 2 are symmetric.
- The tree may be falling "to the side" or "to the interior".

Balance-renewing operations

- Problem appears with balance WLOG 2.
- We start solving on the bottom-most level with this balance.
- We explore two possibilities, the remaining 2 are symmetric.
- The tree may be falling "to the side" or "to the interior".
- In the former case we use a rotation, in the latter a double-rotation.

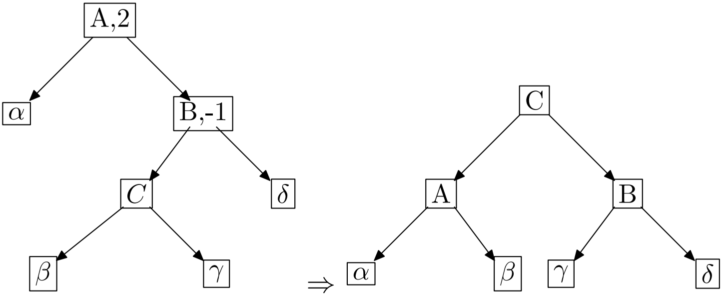
Rotation

Tree is falling "to the side".



Double-rotation

Tree is falling "to the interior".



Analysis and remarks

rotation, double-rotation, depths

- While inserting, one rotation (or double-rotation) suffices.

Analysis and remarks

rotation, double-rotation, depths

- While inserting, one rotation (or double-rotation) suffices.
- Delete may start a cascade of rotations (the distortion is travelling towards the root).

Analysis and remarks

rotation, double-rotation, depths

- While inserting, one rotation (or double-rotation) suffices.
- Delete may start a cascade of rotations (the distortion is travelling towards the root).
- Number of elements in an AVL-tree with depth n:

Analysis and remarks

rotation, double-rotation, depths

- While inserting, one rotation (or double-rotation) suffices.
- Delete may start a cascade of rotations (the distortion is travelling towards the root).
- Number of elements in an AVL-tree with depth n:
- Depth of the sons differs at most by one, thus: $T(n) \geq T(n-1) + T(n-2)$,

Analysis and remarks

rotation, double-rotation, depths

- While inserting, one rotation (or double-rotation) suffices.
- Delete may start a cascade of rotations (the distortion is travelling towards the root).
- Number of elements in an AVL-tree with depth n:
- Depth of the sons differs at most by one, thus: $T(n) \geq T(n-1) + T(n-2)$,
- Thus the number of elements is at least the nth Fibonacci number,

Analysis and remarks

rotation, double-rotation, depths

- While inserting, one rotation (or double-rotation) suffices.
- Delete may start a cascade of rotations (the distortion is travelling towards the root).
- Number of elements in an AVL-tree with depth n:
- Depth of the sons differs at most by one, thus: $T(n) \geq T(n-1) + T(n-2)$,
- Thus the number of elements is at least the nth Fibonacci number,
- thus the depth is logarithmic w.r.t. number of elements.

Red-black trees

- Another method how to keep the tree sufficiently spreaded.

Red-black trees

- Another method how to keep the tree sufficiently spreaded.
- Each vertex is colored with red or black color.

Red-black trees

- Another method how to keep the tree sufficiently spreaded.
- Each vertex is colored with red or black color.
- Red vertices must not appear one after another,

Red-black trees

- Another method how to keep the tree sufficiently spreaded.
- Each vertex is colored with red or black color.
- Red vertices must not appear one after another,
- number of black vertices is the same for any path from the root to all the leaves.

Red-black trees

- Another method how to keep the tree sufficiently spreaded.
- Each vertex is colored with red or black color.
- Red vertices must not appear one after another,
- number of black vertices is the same for any path from the root to all the leaves.
- Thus one subtree has depth at most twice larger than the other.

Red-black trees

- Another method how to keep the tree sufficiently spreaded.
- Each vertex is colored with red or black color.
- Red vertices must not appear one after another,
- number of black vertices is the same for any path from the root to all the leaves.
- Thus one subtree has depth at most twice larger than the other.
- The tree is administrated using rotations, double-rotations and recoloring.

Red-black trees

- Another method how to keep the tree sufficiently spreaded.
- Each vertex is colored with red or black color.
- Red vertices must not appear one after another,
- number of black vertices is the same for any path from the root to all the leaves.
- Thus one subtree has depth at most twice larger than the other.
- The tree is administrated using rotations, double-rotations and recoloring.
- Exact rules get lectured on Algorithms.

Red-black trees

- Another method how to keep the tree sufficiently spreaded.
- Each vertex is colored with red or black color.
- Red vertices must not appear one after another,
- number of black vertices is the same for any path from the root to all the leaves.
- Thus one subtree has depth at most twice larger than the other.
- The tree is administrated using rotations, double-rotations and recoloring.
- Exact rules get lectured on Algorithms.
- The depth is also logarithmic w.r.t. number of elements.

FIXME!!!

A-B-trees, *k*-ary tree canonical representation.
Passing a function as an argument.
A queue and a buffer,
graph-searching algorithms (including graph representation).
Odstrasujici priklady (slidy10.tex for mathematicians).