Overview

- Standard units,
- Pointers.

Standard units

Turbo Pascal is equipped with several standard units:

- crt,
- dos,
- graph,
- printer,
- ...

Units may differ for individual compilers!

Unit `crt`

- Unit for working with a keyboard and a display (colors, sounds)
- Variables: `LastMode` (says what textmode was the last one used before switching graphics on),
- `TextAttr` (current attributes for displaying (text). Gets operated by `TextBackground` and `TextColor`),
- Procedure `TextBackground` sets the background color, proc. `TextColor` sets the color of foreground.
- function `keypressed` (returns `boolean` saying whether any key was pressed, `clrscr` (erases the display).

Units `dos`, `graph` a `printer`

- Unit `dos` works with files, directories, disks...
- Unit `graph` enables graphic mode (`InitGraph`, `CloseGraph`, `GraphResult`, `SetColor`, `GetColor...`).
- Unit `Printer` serves for printing.
- All these units consist of many functions, procedures and variables. If you want to know more, you find information in Help.

Strange example:

Probably you have already seen this several times:

```
program nothing;
uses crt;
...
begin
...  repeat until keypressed;
end.
```

What is this?

Use of unit `crt`, namely its function `keypressed`.

Pointers – motivation

- Sometimes we would need an unbounded amount of memory.
- In Pascal (so far) it is impossible...
- if we do not know pointers.
- Memory is linearly organized (individual addresses are indexed by natural numbers usually in hexadecimal system),
- on these addresses, data (and also code) can be stored.

Pointers – ideas

- A pointer is something that points at something (in this case, at an address).
- It is useful to be able to directly access a particular address (larger data-storage),
- but this functionality has to be used responsibily – we are not the only ones to use the memory
- in particular we are sharing the memory with the code we are executing.
- So one has to pay attention – using pointers incorrectly can crash your program (or even your system)!!!

Pointers – syntax and semantic

- Technically we establish a data-type pointer.
- To do so, we use the operator `^`:
  `type pint=^integer; {pointer at integer}`
- Further we define an appropriate variable:
  `var a:pint;`
- The operator can be dereferenced by (unary postfix) operator `^`:
  `writeln(a^);`
- But in practice it is not so simple!

Memory organization

- Memory contains code, static data, buffer and a heap.
- Where does a pointer point to?
- A correct pointer should point into the heap, but an incorrectly created pointer can point anywhere!

The @-operator

- We still did not solve the problem how to initialize a pointer-type variable:
- For a variable of a given data-type, we can create the pointer via the @-operator:
  `p:pint; a:integer; p:=@a;`
- The pointer now points at the memory address of the variable!
- What would this code do, then?
  `p^:=5; writeln(a);`
- Also it may happen that several pointers are pointing at the same point (pointer-aliasing).

Dynamic variables

- Our motivation was to use the memory dynamically, i.e., during the computation we decide how much we do need.
- For this we use the function `new`.
- As an argument we provide a pointer-typed variable,
- function `new` allocates a new space for this variable:
- Example: `new(p); p^:=10;`
- After we finish using the variable, we have to deallocate the space:
  `dispose(p);`
- Otherwise we create a memory leak!

Example

- `var a,b:pint;`
- `new(a);` – allocate a space for an integer-typed variable,
- `a^:=5;` – fill the value 5 under pointer `a`.
- `new(b);` – allocate space for `b`,
- `b^:=a^;` – copy the value stored under `a` under `b`.
- `b:=a;` – copy the pointer – `a` and `b` are pointing at the same location (memory leak! why?).
- `b^:=10;` – write under the pointer `b`,
- `writeln(a^);` – what does this do?

Deallocation

- When we do not need an allocated space any more, we have to deallocate it.
- A deallocator is provided by the function `dispose`.
- Example: `dispose(a);`
- Now we must not use `a^...` until we newly allocate `a`!
- We must *not* deallocate the same pointer more than once!
- If we redirect the (last) pointer to a particular address, we can never access this memory again (before the program ends)!
- Some languages use a garbage-collector (Java, C#), i.e., no explicit deallocation is necessary, garbage-collector takes effect at unexpected time (convenient but not as efficient as explicit deallocation).
- Pascal does not have a garbage-collector.

One application of pointers

- Linked list – a data structure where each element points at its successor.
- We define a structure (record) containing a value (or values) and a pointer to the next element.
- Note that it is possible in Pascal to make a pointer at a data-type that is so far undefined!
- Applications: Library, phone-book,...
- Individual elements are pointing at their ancestors.
- How do we recognize the end?
- By a special constant `nil` (representing address 0).

Example

```
type ll=^packet;
     packet=record
          data:integer;
          next:ll;
     end;
var list,tmp:ll;
```

Linked list of numbers – read and write

```
begin list:=nil; tmp:=nil;
      while not EOF do
      begin new(tmp);
            readln(tmp^.data);
            tmp^.next:=list;
            list:=tmp;
      end;
      while list<>nil do
      begin writeln(list^.data);
            tmp:=list;
            list:=list^.next;
            dispose(tmp);
      end;
end.
```

Linked 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,
- 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 linked 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;
          headˆ.next:=nil;
          headˆ.val:=what;
     end else
     begin new(tmp);
          tmpˆ.next:=nil;
          tmpˆ.val:=what;
          headˆ.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 linked 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`.
- Good exercise: Bubblesort over linked list.
- Organizing (an ordered) linked list (functions insert, delete and member that are working with the ordered linked list).

Ordered list

- A linked 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.
- 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 linked list, it is inefficient to search for a given 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 does one implement this?
- Each element gets more than one sibling (left, right).

Tree representation

in Pascal

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

Binary search trees

- A binary tree is such a tree where each element has at most two siblings.
- A 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 linked-list.
- If we build it badly, it collapses into a linked-list.
- How to build a balanced binary search tree (and how to keep the tree balanced)?
- A 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.
- 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

- The data is given in an array and we convert it to a tree (we omit the details of array handling).
- The following dynamic data structure represents the vertices of a 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;
```

- Beware of the algorithm's implicit assumption 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,
- 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 linked list.

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

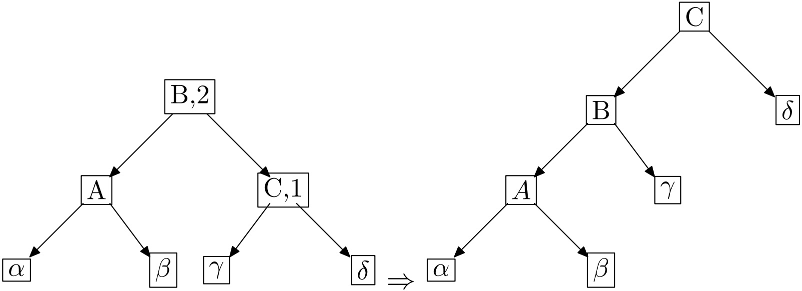
- Generally, it is an unpleasant problem.
- Thus AVL-trees got introduced with a bit relaxed notion of balancedness.
- AVL-tree is a BST where for each element the depth of the left subtree differs at most by 1 from the depth of the right subtree.
- AVL – Adelson-Velskij and Landis.
- Operations `member`,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

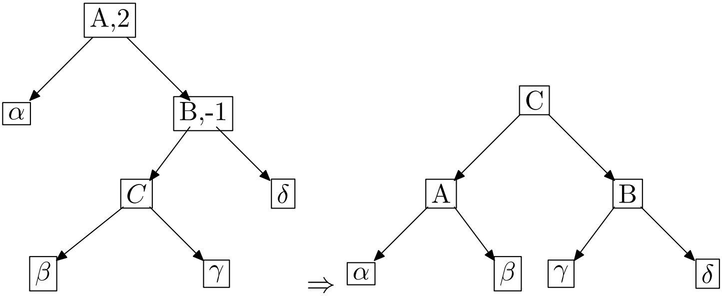
Rotation

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.
- 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 spread out.
- 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!!!

binary search trees,
AVL-trees,
red-black-trees.

FIXME

Passing a function as an argument.
A queue and a buffer, graph-searching algorithms (including graph representation). Odstrasujici priklady (slidy10.tex for mathematicians).