Prof. Jan M Honzik

Algorithms and Data Structures – Syllabi of lectures

Recommended literature

Wirth,N.: Algorithms + Data Structures = Programs.
Prentice Hall, Inc., 1975.

Knuth,D.: The Art of Computer Programming.
Vol.3. Sorting and Searching.
Addison - Wesley, 1973.

Baase,S.: Computer Algorithms, Introduction to Design and Analysis.
Addison - Wesley, 1989.

CD Issued by Dr. Dobb's Essential Books on Algorithms and Data Structures CD-ROM - Release 2

1. Horowitz, Sahni: Fundamentals of Data Structures
2. Korsh, Garrett: Data Structures,Algorithms and Program Style Using C
3. Mark Allen Weiss:Data Structures and Algorithm Analysis in C
4. Wayne Amsbury:Data Structures: From Arrays to Priority Queues
5. William B. Frakes and Ricardo Baeza-Yates:Information Retrieval: Data Structures & Algorithms
6. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest: Introduction to Algorithms
7. Flamig: Practical Data Structures in C++
8. Thomas Plum:Reliable Data Structures in C
9. Alfred V. Aho, John E. Hopcroft,Jeffrey D. Ullman:Data Structures and Algorithms
10. Dr.Dobbs Jornal: Algorithms and Data Structure Articles

Evaluation with ECTS grades:

less then 50 - "fail/F"
50-59 points pass as "good/E"
60-69 points pass as "good/D"
70-79 points pass as "very good/C"
80-89 points pass as "very good/B"
90-100 points pass as "excellent/A"

Summary of the course

Abstract Data Structures. Dynamic allocation of storage. User implemented allocation. Abstract Data Types. Lists. Single-linked and double-linked lists. Circular Lists. Their specification and implementation. Stack - specification and implementation. Examples of usage. Evaluation of arithmetic expression. Queue - specification and implementation. Examples of usage. Trees. Binary trees. Operations upon the binary trees: comparing,

copying, destroying, transversals. Recursive and non-recursive implementation of tree operations. Arrays, their specification and implementation. Dynamic array. Dense array.

Searching. Search tables, specification of operations. Searching methods: Sequential searching in file, list and array. Sentinel. Searching in ordered sequence. Binary searching. Dijkstra and normal binary searching. Searching in binary search trees. Searching, insertion, deletion. Recursive and non-recursive versions of implementation. Hashing tables.

Sorting (Ordering). Basic concepts. Stability. Ordering with multiple keys. Ordering without movement of items. Classification of ordering methods. Time complexity of algorithms. Big 0, Omega and Theta. Ordering by comparison. Insert sort, Bubble sort, Heap sort. Ordering by insertion. Insert sort, Bubble insert sort. Ordering by partition. Quick sort, recursive and non-recursive implementation of algorithm. Ordering by merging. Merge sort. List-merge sort. Shell sort. Radix sort. Sequential ordering of files or sequences. Straight and natural merging. Balanced multi-way merging. Poly-phase merging.

Text processing. Searching samples in strings. Knuth-Morris-Pratt's algorithm. Boyle-Moore's algorithm.

WARNING AND APPEAL

Course-text is in the draft form. Algorithms are not debugged in the form involved in the text.
Reader is encouraged to send all errors, corrections and recommendations to the author mail address:
honzik@fit.vutbr.cz
Students will be awarded by the points according the seriousness and value of their message.

Opening lesson

1 Repetition of prerequisite fundamental knowledge

1.1 **One Hundred words**

A hundred of keywords expresses pre-requisite concepts and knowledge to the course IAL - Algorithms .

1.1.1 Concepts from the course of Programming and Usage of Computers.

Structure of the computer, memory, access time to the memory, address, data. instruction, sequential processing of the instructions, external memory, communication of the computer with its environment, input/output, interactive computing, batch computing.

Basic knowledge and ability to use programming language C or ANSI Pascal, and especially:

1.1.2 syntax graphs of the programming language, Backus Normal Form (BNF) of syntax

- Basics types, ordinal types, enumeration type and its usage, type array, string, type record, compatibility of types, type pointer and its usage, equivalence of types, compatibility by assignment, type checking.
- Notation and enumeration of expressions, rounding and truncating of integer expressions, conversion of types (integer <--> real), short-evaluation of Boolean expression.
- Looping, choice of type of loop statement, semantics of for, while and repeat cycle, case statement, with statement.
- Procedures and functions - choice and usage, passing of parameters, side effects, features of block structure, recursion.

1.1.3 Representation of numbers.

- Binary numbers, straight and complement binary code, numbers with floating and fixed representation, mantissa, exponent, normalized number.

1.1.4 Fundamentals of algorithms

- Structured programming, choice of the type of cycle, choice of the structure of data, algorithms with sequential access to data, algorithms with random (straight) access to data, searching for extremes (minimal, maximal value, key value, subsets), exchange (swapping) of the two values, ordering versus sorting, fundamental numerical algorithms (evaluation of the sequences, mass summation, mass multiplication), fundamental algorithms on texts, non-decreasing (-increasing) sub-sequence and its searching in the sequence.

1.1.5 Basic concepts of mathematics in algorithms

- Linear, logarithmic, quadratic, cubic, polynomial, binomial and exponential functions.
- Fundamental logic operation, namely implication.
- Operation upon the sets, set quantifiers (ForAll, Exist).
- Average value of the set of numbers, weighted average, dispersion, standard deviation - square root of dispersion.

1.1.6 Extended Backus-Naur Formalism

Formal language is defined by a set of sequences of symbols. Elements of the set are called sentences. In the case of programming languages sentence is the program.

The symbols are elements of a finite set, called dictionary. The set of programs (which is finite) is defined by rules on composition. Sentences created according these rules are called to be syntactically correct. Set of the rules of composition is called syntax of the given language. Program (sentence) of formal language consists of the parts called "syntactic entities" i.e. declaration, statements or expressions.

Syntactic factors:

If the construction A consists from B followed by C, i.e. concatenation of BC, then B and C are called syntactic factors, and A is described by the formula:

A = BC.

Syntactic terms.

If A consists alternatively from B or from C, we call B and C syntactic terms and A is described:

A = B | C

(Remark: sometimes symbol "::=" is used instead of "=" i.e. A ::= B | C)

Besides "concatenation" and "selection" it is convenient to work with "option" and "repetition". If A may be either B or nothing, we express the construction as

A = [B]

If A consists from concatenation of any number of B, including the zero number, we express the construction as

A = {B}

The brackets may be used for association of factors and terms. While A,B and C denote syntactic entities, symbols as |, [,], {, }, (,), are called meta-symbols. The definition using the above mentioned rules and methodology is called Extended Backus-Naur Formalism (EBNF).

Example of definition of Pascal identifier.

identifier = letter { letter | digit }.
letter = A | B | C | ... | Z | a | b | c | ... | z.
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

1.2 Complexity of algorithms and data structure

Mutual evaluation of the two difference algorithms needs objective criteria. The "time-space" approach is often used. This approach takes into account the time necessary to finish the

algorithm until the correct result is reached and the memory space needed for it. These features are expressed by "time complexity" and by "space complexity" of an algorithm. Both complexities are expressed by the functions where the size of data is an argument of them. Sizes are expressed mostly by the number of items of problem oriented homogeneous data structure (e.g.file, list, array), processed by the algorithm.

1.2.1 Asymptotic expression of time complexity

The criterion used frequently for the evaluation of the speed of the algorithm is its asymptotic time complexity. It results from the behavior of the function for n increasing to infinite.

Complexity expressed by notation Omicron (Big O, Ό) expresses the upper limit of the time behavior of the algorithm.

Omicron (g(n)) denotes the set of functions f(n), for which is true:

{f(n) : ∃ (c>0, n0>0) such, that ∀ n>=n0 is valid
[0<=f(n)<=c*g(n)]}
where c and n0 are adequate positive constants.

Then the notation f(n)=Omicron(g(n)) or Ό(g(n)) denotes that function f(n) is growing maximally as fast as the function g(n). Function g(n) is the upper limit of the set of all functions, determined by the notation Omicron(g(n)) or (Ό(g(n))).

Complexity expressed by notation Omega (Ω) denotes the lower the limit of time behavior of the algorithm.

Omega(g(n)) (Ω(g(n))) denotes the set of all functions f(n) for which is true:
{f(n) : ∃ (c>0, n0>0) such, that ∀ n>=n0 is valid
[0<=c*g(n)<=f(n)]}
where c and n0 are adequate positive constants.

Then the notation f(n)=Omega(g(n)) (Ω(g(n))) denotes that function f(n) is growing minimally as fast as the function g(n). Function g(n) is the lower limit of the set of all functions, determined by the notation Omega(g(n)) or (Ω(g(n))).

Theta(g(n)) (Θ(g(n))) denotes the set of all functions f(n) for which is valid::
{f(n) : Exist∃ (c1>0, C2>0, n0>0) such, that
∀ (n>=n0) is valid [0<=c1*g(n)<=f(n)<=c2*g(n)]}
where c1,C2 and n0 are positive constants.

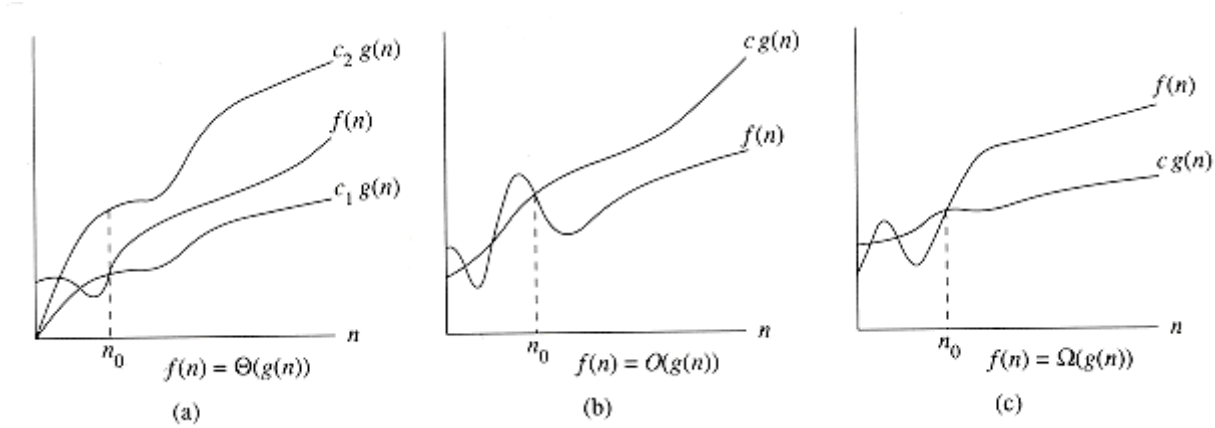
Complexity expressed by the notation Theta (Θ) denotes the time behavior same as the given function.

Then the notation f(n)=Theta(g(n)) or Θ(g(n)) denotes, that function f(n) is growing as fast as the function g(n). Function g(n) is expression of the order of upper and as well the lower limit of the set of functions, determined by the notation Theta(g(n)).

Note:

∃ stands for "Such that exist…"
∀ stands for "For all that …"

Omicron, Omega and Theta is expressed on the following figures.



1.3 Types of complexity

- Θ(1) denotes the constant time complexity.
- Θ(log(n) denotes the algorithms with the logarithmic time complexity. The value of the base (root) of logarithm is not important, as various bases differ only by the constant of mutual transformation. Such a complexity has some quick searching algorithms.
- Θ(n) is denotation of the algorithms with the linear time complexity. This complexity is typical for sequential algorithms.
- Θ(n*log(n)) is called "linearithmic" complexity. This complexity is typical for some fast sorting algorithms based on comparing model.
- Θ(n*n) or Θ(n^2) is quadratic complexity. It is typical for many algorithms based on the double cycle model- one inner cycle and one outer cycle. Many ordinary sorting algorithms belong to them.
- Θ(n*n*n) or Θ(n^3) is denotation of the algorithms with cubic complexity. Algorithms with that complexity are usable only for limited number of problems. Any doubling of the n results in eight times enlarging of the time.
- Θ k exp n) (where k is real positive number, mostly integer) is denotation of algorithms with exponential complexity (for k=2 binomial complexity). There exist several practically usable algorithms of that class. They are called "algorithms

working with the brute force". Whenever their n is doubled, resulting time is powered by two.

1.4 Space complexity

Space complexity is expressed by the amount of memory space needed for the code and data of algorithm. With the exception of extremes, the code is negligible to data. Most frequently the memory complexity may be expressed as constant, logarithmic or linear order. An algorithm, which doesn't need more memory space than the own processed data is called to be working "in situ" (on it's own place).

_

1.5 Home assignments:

1. Send email message to: honzik@fit.vutbr.cz containing.:

Your Surname, Given name, your other names, your date of birth, your address in Brno, your faculty email address, your private email address.

2. Using the random function or the other way, create an array of 25 random integer positive numbers. Write the procedure which finds the maximal number. Print on the screen (or to the output text file) all input data (created numbers) and the result. (or to the output text file)

3. Using the random function or other way, create an array of 25 integer positive numbers. Write the procedures which finds the minimal number. Print on the screen (or to the output text file) all input data (created numbers) and the result.

4. Using the random function or the other way, create an array of 25 integer numbers greater than 0. Write the procedures which evaluate the weighted average value. The weight of every integer is its square root truncated to (nearest highest) integer value. Print on the screen (or to the output text file) all input data (created numbers) and the result.

5. Using the random function or the other way, create an array of 25 integer numbers greater than 0. Write the procedure, which finds the longest non-decreasing sub-sequence of numbers. Write on the screen (or to the output text file) the first index of the sub-sequence, length and all numbers from the found sub-sequence.

- Each source program should begin with the comment introducing:
- The Name of program
- The assignment (text of the task, purpose of the piece of program)
- The name of programmer and his email address

- The date the program was created
- Other necessary comments helping to run or to use the piece of program.

1.6 Recommendations:

- Avoid using the "go-to" statements in the programs.

- It is advisable to start identifiers of types with "T" as prefix, e.g.

```
type
  TIntArray=array[1..25]of integer; (* Array of random numbers *)
  ...
var
  IntArray:TIntArray;
```

In declaration use only type identifier after colon ":" and not explicit declaration. It is not an error but bad programming style, leading easily to errors.

Example of bad programming style with explicit specification of declared variable:

```
var
  IntArray:array[1..10] of integer;
...
```

Good style :

```
type
    TIntArray[1..10] of integer;
…
var
    IntArray : TIntArray;
…
```

2. LECTURE ON ABSTRACT DATA STRUCTURES AND LISTS. Lesson two.

2.1 Abstract data type - definition:

ADT is defined by the set of all values which the element of the given type may have, and by the set of operations belonging to the given type.

- ADT is defined (specified) by the definition of its syntax and semantics.
- Syntax may be defined by EBNF or by signature or by syntax graphs.
- Semantics may be defined: verbally, functionally or axiomatically.

Example of definition by signature (syntax graph) and axiomatic semantics:

Let us define the ADT "positive integer" - Posint.

2.1.1 Algebraic Signature:

```
One  : → Posint    (This operation is called "generator" or
                    "initialization". It should be used before all
                    other operations)
ADD  : Posint x Posint → Posint
SUCC : Posint → Posint
IsOne : Posint → Boolean
```

(Syntax graph drawn on the whiteboard).

2.1.2 Axiomatic specification

Minimal set of operations may be defined as:

1. ADD(X,Y) = ADD(Y,X)
2. ADD(One,X)=SUCC(X)
3. ADD(SUCC(X),Y)=SUCC(ADD(X,Y))
4. IsOne(One)=true
5. IsOne(SUCC(X))=false.

Algebraic signature and axiomatic specification of semantics are precious but not very efficient for everyday work. We will use syntax graph widely and verbal specification of semantics.

Functional specification of semantics is specification by means of procedures/functions of a real programming language. This method is understandable, but it implies the implementation of ADT.

2.2 The significance of ADT.

It is better to create a simpler (shorter) program and to use more complex data. Such a program is easier to prove for correctness, more legible and understandable. This rule leads to exploitation of abstract data types.

ADT is like black-box which ADT hides (encapsulates) its inner structure and construction. If we know, what there is on the input and which operation is used, we may determine what there will be on the output, but we may not know (we don't need to know) what happens inside the black-box and how black-box is designed inside.

Example:

Real number in Pascal is a kind of ADT. Its inner structure is rather complex. Perhaps it consists of the mantissa in certain form and exponent in certain form. Addition of two real numbers is represented by rather complex algorithm. We don't need to know what happen inside the addition. We get the result. The inner structure is encapsulated and we don't need to know it. If we find it and use somehow the specific features of the inner structure, our program will be "implementation dependent".

Abstract data structure - ADS is one case (one entity, one appearance, one item) of given ADT. We may declare many items (cases, appearances) of given ADT.

2.3. Characteristics (features) of ADT.

Static versus dynamic

Static structure has a fixed number of components/items and fixed form of their arrangement. The memory allocated to static structure is allocated before the run of the program.

Dynamic structure may be created and destroyed (be born and die) while the program is running. It may change the number of its components/items and/or of their arrangement.

Homogeneous versus heterogeneous

Homogeneous structure consists of "items" of the same types. Heterogeneous structure consists of the "components" possibly of different types. Items imply the homogeneity of the structure.

Sequential versus random access.

Random access allows the access to any item of the structure at any time. For example: chess player has access to any piece of the chessboard.

Sequential access to the item of the structure means that the access to the item is possible only after the access to its predecessor. Access is done one after the other, starting from the first

and following to the next one. The example is the tape player. To play a song, you should pass/play all tunes before.

Special operations with the data structure:

Constructor is an operation upon the ADT for creating/assembling the structure from enumerated individual components. The input consists of enumeration of components (elements) and the result is the structure itself.

Example: 'HALLO' is a string constructor for the structure of five characters (letters) string. ['A','E','I','O','U','Y'] is set constructor for the subset of the character set, representing the vowels.
Those are the only two constructors in the Pascal.

Selector is an operation upon the ADT which realizes the access to one of the components of the structure. The input of the operation consists of the name of structure and from the designator of the element. The result is the access to specified element of the structure.

Example :

A[4] := 20; is active access to 4th element of array A – value 29 is assigned to 4^{th} element of A
I := A[4]; is passive access to 4th element of array A. The value of 4^{th} element is obtained.
Person.Name:= 'Mary'; is active access to name of record Person. Component Name of the record Person is assigned to value "Mary".

Linear and non-linear structure

Data structure is "linear" if there is only one successor for every item (with exception of the last item) and if there is only one predecessor for every item (with exception of the first item). Linear structure has the first and the last item. In other cases the structure is non-linear.

Example: The string is a linear structure. The tree is a non-linear structure. The circular structure is a special case; every its item has only one predecessor and one successor but the circle has no first and no last item.

2.4 The ADT "List".

List is dynamic, linear, sequential and homogeneous abstract data type.
The most common different forms of list are:

single linked list (one way list)
double linked list (bidirectional list)
circular list (single or double linked).

Not empty single linked list (SLL) is the chain of elements which has a starting (first) element and ending element which has no successor. SSL can be accessed in the sequential manner, one after another, from the first to last – in only one direction from the predecessor to successor. This is achieved by operations defined upon the single linked list. Initial access to the SLL is via its starting (first) element.

Not empty double linked list (DLL) is the chain of elements which has a starting (first) element and ending (last) element which as no successor. DLL can be accessed in sequential manner, one after another (from left to right or from right to left), either starting from the first or from the last element in both directions. This is achieved by operations defined upon the double linked list. Initial access to DLL is either via its left end (first) element or via its right end (last) element. The first element has no predecessor and the last element has no successor.

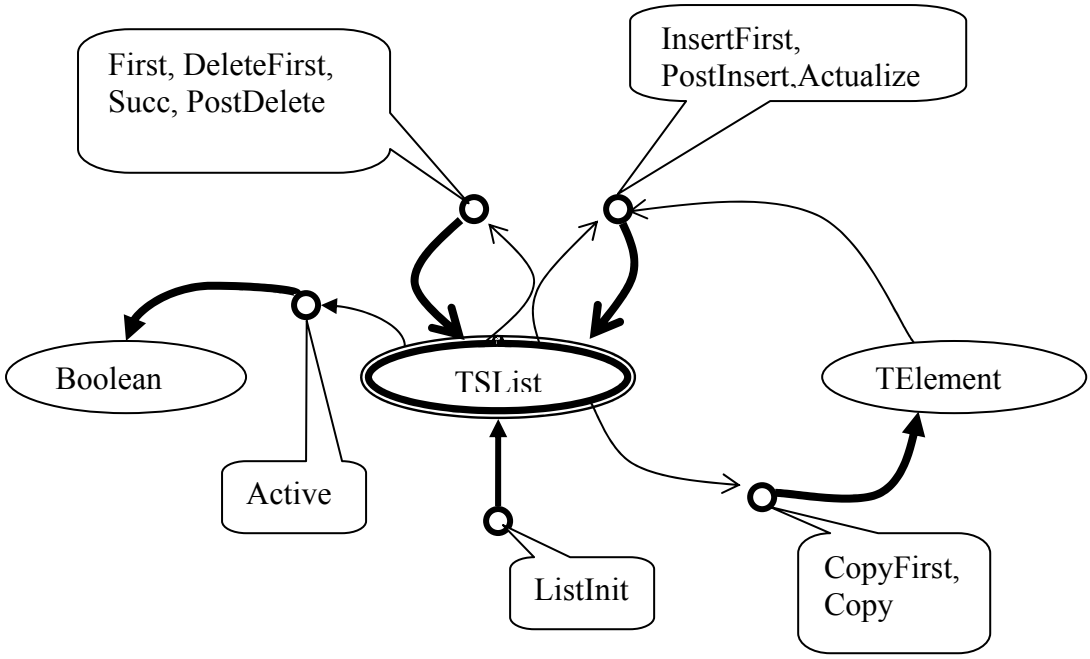
The set of operations upon the single and double linked list:

Single linked list additional operations for Double linked list.

(Note: L means list as one entity of the ADT TList, El means Element of the type TEl, what is type of list item.)

ListInit(L),	
InsertFirst(L,El)	InsertLast(L)
DeleteFist(L)	DeleteLast(L)
CopyFirst(L,El)	CopyLast(L,El)
First(L)	Last(L)
SuccList(L),	PredList(L)
PostInsert(L,El)	PreInsert(L,El)
PostDelete(L)	PreDelete(L,El)
Copy(L)	
Actualise(L,El)	
Active(L)	

2.4.1 Diagram signature of ADT TSLIT



Where:

- Oval represent the type. Oval with double and bold line expresses just a specified type. Other ovals represents pre-defined types.
- Small circle represents a specified type of operation
- Thin arrow represents the input parameter of the operation arrow is directed to.
- Bold arrow represents the output operation of the operation arrow outgoes from.
- Descriptive label contains the names of operations belonging to the type of operation represented by the circle.

2.4.2 Description of the operations.

Empty list is initialized by the operation ListInit.

First item should be inserted only by operation InsertFirst (InsertLast).

Single linked list has one item which is the first, double linked list has the last item, too.

There is one and only one item which may be "Active". Activation of the first (last) item is done by operation First (Last). Activity can be moved along the list forwards/backwards by operations SuccList/PredList. List which has one single active item is called active list. Otherwise the list is not active. If the activity is moved forwards from the last item, the list becomes inactive (activity is pushed beyond the list). Symmetric situation is on the beginning of the list with operation PredList applied on the first item.

Operation Active(L) returns "true", if the list is active, otherwise it returns "false". We call such operation "predicate". Predicates return the boolean results.

Insertion is possible only behind the active item - PostInsert(L,El) or in front of the active item - PreInsert(L,El).

Deletion is possible only behind the active item - PostDelete(L) or in front of the active item - PreDelete(L). Deleted item is completely lost (thrown away - its value is not accessible after deletion).

To make the list empty - to remove the last and single item from the list - is done only by operations "DeleteFirst(L)" or "DeleteLast(L)".

The operation "Copy(L,El)" returns the value of active item to output parameter El. The operation is not defined if the list is not active! In such situation an error state occurs and program can't continue normally! All other operations related somehow to activity of the items result as empty operation, if the list is not active!

Operations CopyFirst(L,El) and CopyLast(L,El) are defined only for not-empty list. They return the value of the first/last item of the list to output parameter El. Both operations used for empty list resulted in error state! The program can't continue normally!

The operation "Actualise(L,El)" sets the active item to the value of input parameter El. If the list is not active, the operation is performed as the empty operation.

The emptiness of the list may be tested by the sequence:

First(L);
Active(L,Not-Empty); (* if the result is true, the list is not empty *)

Activity is auxiliary feature of the list. We don't expect to start work with the given list expecting certain state of activity and we leave the list without certain state of activity.

Example of the procedure counting the number of items in the list L.

```
procedure Length(L:TList; var Length:integer);
(* Procedure returns the number of items in the output parameter Length
*)
begin
  Length:=0;       (* initialization of the counter *)
  First(L);        (* making the first active *)
  while Active(L) do begin
    Length:=Length+1;       (* increasing the counter *)
    SuccList(L);            (* pushing the activity forwards *)
  end; (* while *)
end; (* procedure *)
```

2.4.3. Overview of typical higher (more abstract) operations on the list:

- Length of the list
- Copying the list into duplicate list
- Destroying the list
- Equivalence of the two lists
- Lexicographic relation on the two lists
- Insertion and deletion of the items (on the margins, on the searched place, behind the searched place, in front of the searched place)
- Looking for the longest non-decreasing sequence in the list (return the order and length)
- Insertion and deletion of the sub-sequence (sub-list) into the list.
- Searching for the certain sub-list (sample) in the list (return the order)
- Concatenation and decatenation of the lists.
- Making list ordered

2.4.4. Circular list

A circular list may be single or double linked. A circular list has no end. Moving along the circular list we need to recognize passing the starting point (running the second round through the list). The recommended method is to keep the number of the items in the circular list and to count the steps of movement along the list.

Typical operations upon the circular list:

Equivalence of two circular lists.
Duplication of a circular list.
Destruction of the circular list.

Try to design a good set of efficient operations working with the circular list.

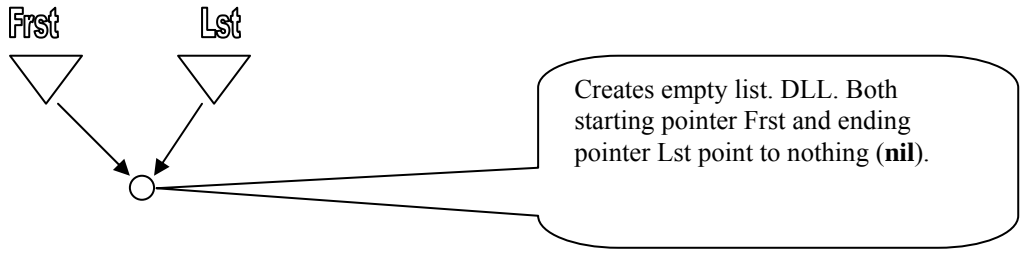
NOTE ON CIRCULAR LISTS!

Note the necessity of recognizing one fixed point in circular list. It may be implemented by operation counting the number of items in circular list like Count(CL,No) or by predicate IsFirst(CL), returning the true, if activity of item is on the first item and returning the false, if circular list is not active or empty.

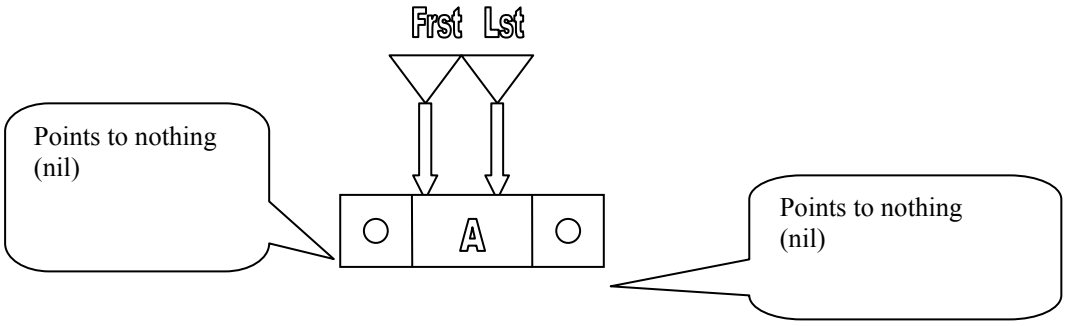
END OF NOTE

Demonstration of effect of simple operations upon the Double Linked List (DLL) with the graphic representation.

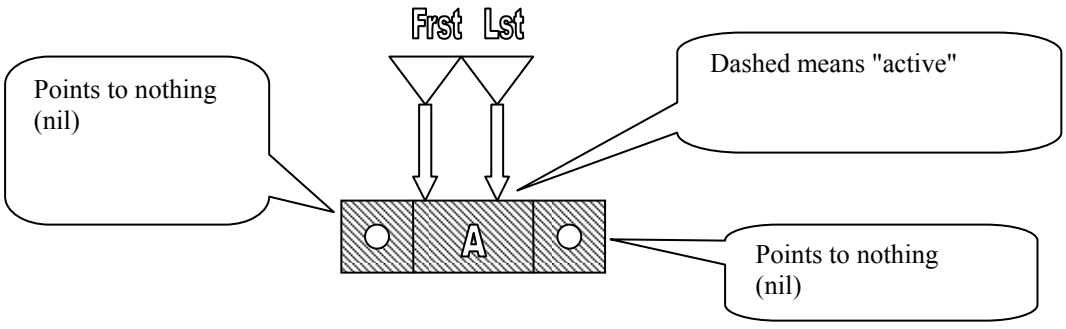
DInitList(DLL); - The result is the empty list. This operation is obligatory prior to the starting the activity with the new SLL.



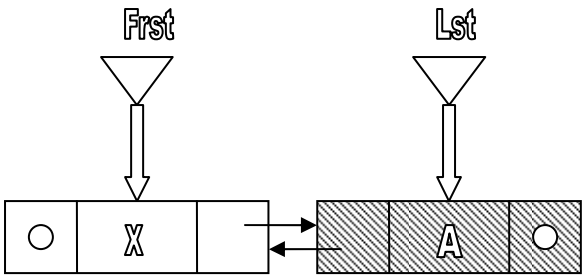
DInsertFirst (DLL,'A'); Inserts value 'A' as the first item to the empty list



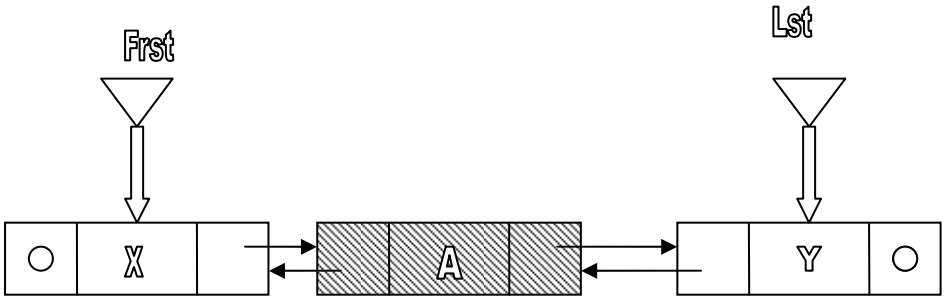
DFirst(DLL); Makes the first item active (dashed filling means "active item").



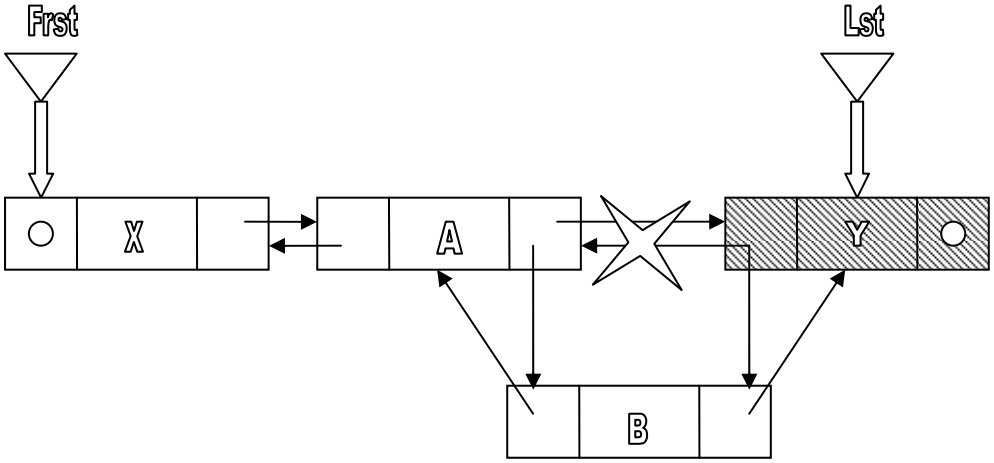
DInsertFirst(DLL,'X'); Operation inserts item with the value 'A' as the first item



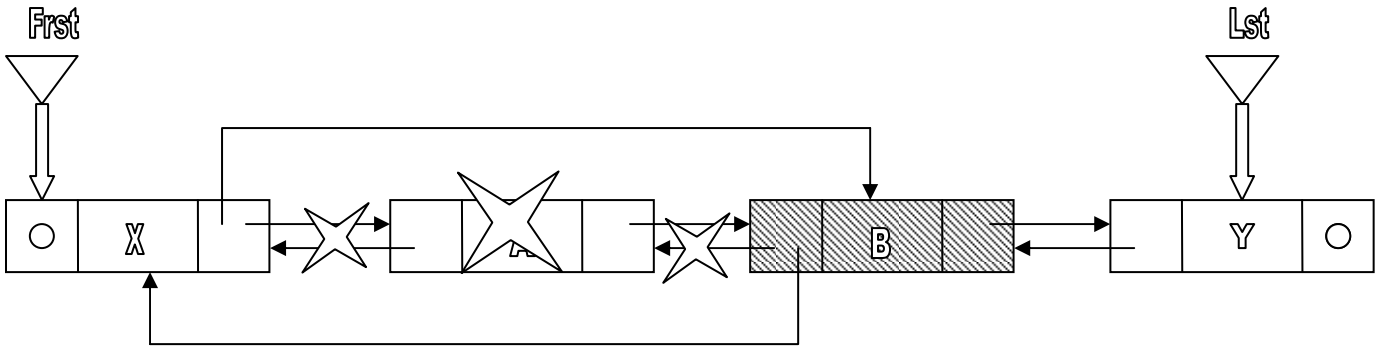
DPostInsert(DLL,'Y'); Operation inserts item 'Y' after the active item.



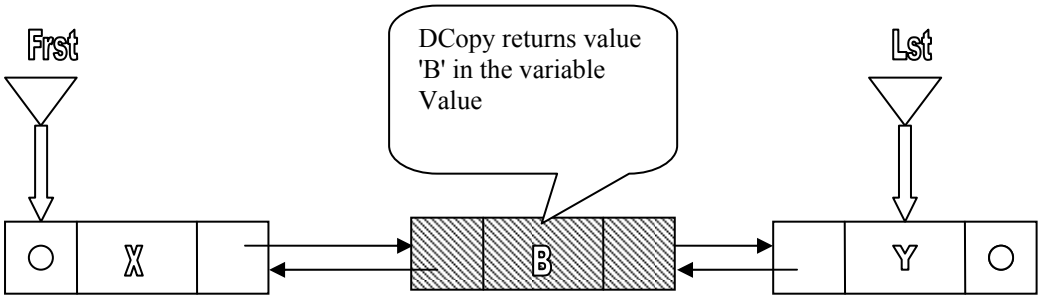
DSucc(DLL); DSucc moves activity to the next item 'Y'.
DPreInsert(DLL, 'B'); PreInsert inserts item 'B' before the active item.



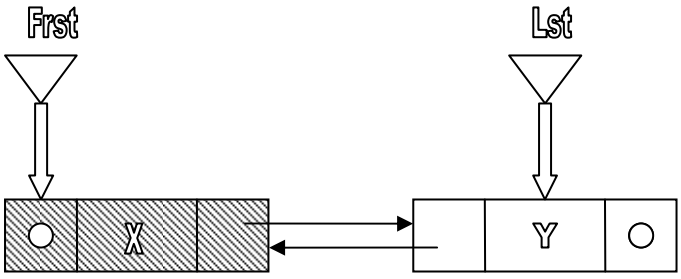
DPred(DLL); DPred returns activity to the preceding item
DPreDelete((DLL); DPredelete deletes item preceding the active item



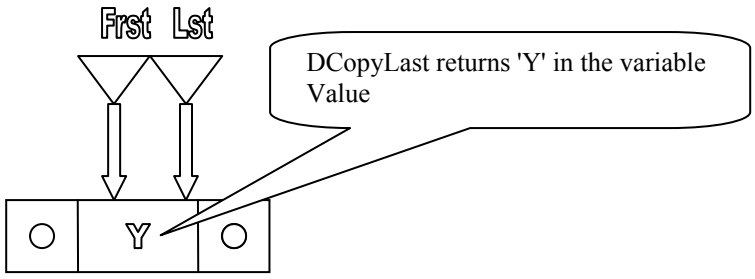
DCopy(DLL, Value); DCopy returns value of the active item. An attempt to read the value from not active list results in error!



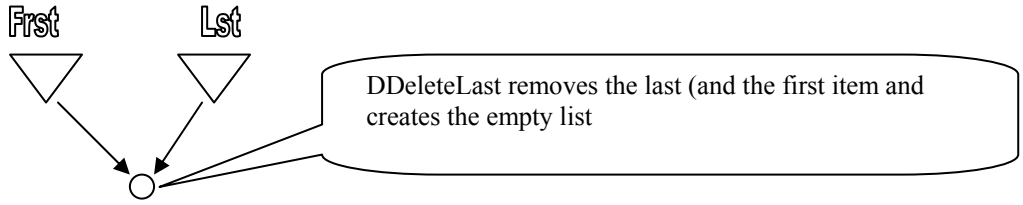
DPred(DLL); Move of activity to the left. Attempt to move activity to the left from the first active item leads to loss of activity (list becomes not active)
DPostDelete(DLL); Deleting the item after active item. Attempt to use of PostDelete operation in the not active list or attempt to delete item after the last active item has no effect (no operation - no error).



DDeleteFirst(DLL); Deletes the first and active item. Results in not-active list.
DCopyLast(DLL, Value); Returns the value of item 'Y'. In this situation the DCopyLast has the same effect like DCopyFirst as list contains only one element which is the first and the last at the same time. An attempt to use the operation DDeleteFirst or DDeleteLast upon the empty list results in the error!



DDeleteLast(DLL); In this situation the DDeleteLast has the same effect as DDeleteFirst as the list contains only one element which is the first and the last at the same time.



Algorithms and Data Structures
Lesson Three.

3. Examples of algorithms upon the ADT list exploiting the abstract operations.

1. Given the type TSList (single linked SL), type TElement and source SL list called SL, create the duplicate SL list called DL,

```
procedure CopySList (LS:TSList; var LD:TSList);
var
  AuxEl:TElement; (* Auxiliary variable *)
begin
  SListInit(LD);       (* initialization of the duplicate list *)
  SFirst(LS);          (* set of activity of the Source list *)
  if SActive(LS)       (* if source list not empty..*)
  then begin
    SCopyFirst(LS,AuxEl);          (* Copy first *)
    SInsertFirst(LD,AuxEl);        (* to duplicate *)
    SFirst(LD);                    (* make first of duplicate active *)
    SSucc(LS);                     (* push activity of source list *)
    while SActive(LS) do begin     (* cycle of duplicating the second
                                   item and others *)
      Copy(LS,AuxEl);
      PostInsert(LD,AuxEl);
      SSucc(LS);
      SSucc(LD)
    end (* while *);
  end (* if *)
end; (* procedure *)
```

Solution with the head item (heading).

```
procedure CopySListWithHead (LS:TSList; var LD:TSList);
var
  AuxEl:TElement; (* Auxiliary variable *)
begin
  SListInit(LD);       (* initialization of the duplicate list *)
  SInsertFirst(LD,AnyValueOfTElement); (* Creating of the auxiliary
                                   head item *)
  SFirst(LD);
  SFirst(LS);          (* set of activity of the Source list *)
  while SActive(LS) do begin     (* cycle of duplicating the second
                                   item and others *)
      SCopy(LS,AuxEl);
      SPostInsert(LD,AuxEl);
      SSucc(LS);
      SSucc(LD)
  end (* while *);
  SDeleteFirst(LD);     (* erasing of the auxiliary heading *)
```

```
end; (* procedure *)
```

2. Given the type TDList (double linked - DL), given the type TElement and the source DL called SL, create the duplicate DL called DL.

```
procedure CopyDList (SL:TDList; var DL:TDList);
var
  AuxEl:TElement; (* Auxiliary variable *)
begin
  DListInit(DL);       (* initialization of the duplicate list *)
  DFirst(SL);          (* set of activity of the Source list *)
  while DActive(SL) do begin
    DCopy(SL,AuxEl);
    DInsertLast(DD,AuxEl);
    DSucc(SL)
  end (* while *)
end (* procedure *);
```

3. Given the SL list called LS of the typ TSList, erase all its items.

```
procedure Destruct (var LS:TSList);
begin
  SFirst(LS);
  while SActive(LS) do begin
    SSucc(LS)l
    Deletefirst(LS)
  end (* while *)
end (* procedure *)
```

4. Compare two lists for equality

Definition of the equality of two lists: Two lists are equal if both are empty or if their first items are equal and at the same time the rests of both those lists are equal.

```
function EquList(L1,L2:TSlist):Boolean;
var
 Equ:Boolean;
 El1,El2:TElement;
begin
  SFirst(L1);    (* Initialization *)
  SFirst(L2);
  Equ:=true;
  while SActive(L1) and SActive(L2) and Equ do begin
    SCopy(L1,El1);
    SCopy(L2,El2);
    Equ:=El1=El2;
    SSucc(L1);
    SSucc(L2);
  end (* while *);
```

```
  EquList:= Equ and not SActive(L1) and not SActive(L2);
end;
```

5. Write the procedure for relation of two lists (the relation upon the TElement is defined).

```
procedure FirstListGreater(L1,L2:TSlist);
var
  Equ:Boolean;
 El1,El2:TElement;
begin
  SFirst(L1);    (* Initialization *)
  SFirst(L2);
  Equ:=true;
  while SActive(L1) and SActive(L2) and Equ do begin
    SCopy(L1,El1);
    SCopy(L2,El2);
    Equ:= El1=El2
    if Equ
    then
      Succ(L1);
      Succ(L2);
    end (* if *);
  end (* while *);

  if Active(L1) and Active(L2)
  then begin
    FirstListGreater:=El1>El2      (* Cycle was finished by Equ=false, ending
                              elements decide *)
  end else begin
   if Active(L1) and not Active(L2)


   then FirstListGreater:= true     (* L1 is longer but it is equal to
                                    the L2 in all L2 elements
                                    -> L1 is greater *)
   else FirstListGreater:= false;   (* L1 is shorter and it is equal to
                                first part of the L2 list
                                -> L1 is less or equal
                                -> L2 is not  greater *)
  end (* if *)
end;
```

Exercises:

6. Insert the new element and keep (preserve) the ordering of the list. The relation on the TElement is defined. While inserting the value contained in the list, insert the new as the last of the sequence of the same items. (insertion of the 8 to sequence "1,3,4,5',5",7,9" will result in: " 1,3,4,5',5",7,8,9"; insertion of the 5 to this sequence will result in: 1,3,5',5",5"',7,9, where 5"' is the inserted value.).

7. Delete the found value if present in the list. (example: deleting the value 5 from the list "1,3,6,8" has no effect. deleting the 5 from the list "1,3,5,7" will result in "1,3,7"). Delete the first item from the subsequence of the same values.

8. Find the longest non-decreasing sub-sequence in the list. Return the order and length of the first from the same longest sub-sequences. (example: The result for the list: "4,3,2,1" is order=1, length=1. The result for the list: 4,1,3,5,2,4,1,8,9" is order=2, length=3;)

Hint: Algorithm is looking for the maximal value of the length. The algorithm should save the order and length of more prospective sub-sequence, to find the parameters of the longest one.

9. Find the average value and dispersion of the lengths of all non-decreasing sequences. (It is necessary to find all lengths of sub-sequences and store their sum and sum of their squares...).

10. There are two lists ordered by the values. Create the new ordered list by merging the two source lists.
(Example: Given lists L1="1,3',5,7,9" and L2="2,3",4,6,8" the resulting list L3="1,2,3',3",4,5,6,7,8,9").

11. Given the list, write the procedure which finds the starting point and the length of the greatest non-decreasing sequence.

```
procedure FindMaxNDS (L:TList; var Nmax,OrdMax:integer);
var
  End, EndOfNDS : Boolean;
  Right, Left:char ;
  n, StartNDS: integer;
begin
(* Initial settings *)
  First(L);

  if not active(L)
  then begin
    Nmax:=0;
    OrdMax:=0
  end else begin
     Left:=CopyFirst(L);
     succ(L);
     if not Active(L)
     then begin
       NMax:=1;
       OrdMax:=1
    end else begin
      ord:=1;
      End:=false;
      Right:=Copy(L);

      while not End do begin (* cycle for all NDS's  - outer cycle *)
         n:=1;
```

```
        startNDS:=ord;
        EndOfNDS:=False;
        while not End and Not EndOfNDS do begin  (* cycle for one NDS - inner cycle *)
          if (Left<=Right)
          then begin  (* NDS is continuing *)
            n:=n+1;
            ord:=ord+1;
            Left:=Rigt;
            succ(L);
            if not Active(L)
            then
              End:=true
            else
              Right:=Copy(L)
          else begin  (* this is the enf of NDS *)
            EndOfNDS:+true
            if n > NMax
            then begin
              NMax:=n;
              OrdMax:=startNDS;
            end;
          end;
        end; (* inner while *)
        Left:=Right;
        succ(L);
        if not Active(l)
        then End:=true
       else Right:=Copy(L)
     end; (* outer while *)
   end;
  end;
end;
```

Algorithms and Data Structures
Lesson Four

4. Principles of dynamic allocation of memory.

Static allocation is done during the processing of the declaration part of the program in compilation. Memory items are accessible by the name of declared variable.

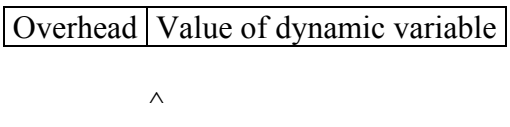
Dynamic allocation is done during the run of the program. Memory is allocated by operation system, compiler, or other system on the basis of request originated during the running program. Memory space is accessible exclusively indirectly, by means of the pointer, which is the part of the static or dynamic structure.

Memory is for allocated items taken from the reserved space in the main memory until it is exhausted. If the reserved memory is exhausted, it causes the fatal error. Program is finished and the corrections of the size of reserved memory should be done.

Principles of dynamic allocation of memory (DAM) may be divided according to the behaviour of the "dispose" operation into the systems with non-regeneration and with regeneration of the disposed space. The reserved space in the main memory is called "heap".

4.1 System without regeneration.

DAM without regeneration allocates adjacent segments of memory in the reserved space sequentially until the space is exhausted. The allocated segments consist obviously of two parts: overhead and used parts.



^

Pointer (address) returned by the new operation

While overhead contains the Boolean indicator free, and/or other control information and it is not accessible to user, the other part represents the space for value of dynamic variable. Pointer returned is address of the "used" part. Mechanism DAM within regeneration uses the pointer
"FreePtr" pointing to the free part of the heap. Statement new, returns the value of FreePtr and increases its value by the length(overhead) + length(used) and sets the indicator "Free" (in the overhead) to "false".

```
new(Ptr) =>   Ptr:=FreePtr;
          FreePtr:=FreePtr + length(allocated item)
```

Operation Dispose results in no regeneration. It changes indicator "Used" to value false only. The main and the only reason of this is to provide the diagnostic for not legal attempt to access not used (disposed) dynamic variable. The lack of such mechanism causes the non-predictable errors and it is difficult to find errors caused by incorrectly directed pointers. It is

the good moral of the programmer to dispose all not used dynamic variables even if they don't regenerate the heap!

Mechanism without regeneration mostly enables "mass" regeneration of the contiguous part of the memory space by means of the operations "mark" and "release". Operation "Mark" saves the value of the FreePtr and operation "Release" reassigns its value back. It is used mainly in the procedures
working with the dynamic structures.

```
procedure DynStructProcess;
...
begin
  mark;  (* saving of the current state of the FreePtr *)
     ....
     (* creation and processing of the dynamic structure by means of
        the new operation *)

     (* at the end of the procedure, created dynamic structure is not
used any more; it may be destroyed and its space returned to the heap by
the means of the "release" operation *)

  release;  (* making free of the all space, allocated in the procedure *)
end;
```

Mechanism DAM with the regeneration may be simply simulated my means of the array with the base type of the single dynamically allocated type and by means of the indexes. Index serves as the pointer. We don't use the "overhead" part of the allocated item.

DAM is taken as the data abstract type with the defined operations:

- InitDAM,
- NewDAM, DisposeDAM,
- MarkDAM
- ReleaseDAM.

Implementation of DAM operations

```
type
  const HeapLength=100;
  TDAM = record
     FreePtr, MarkPtr: integer;
     ArrItem: array [1..HeapLenght] of TItem;
  end; (* record *)

var
  DAM:TDAM; (* global variable representing data part of DAM*)

procedure InitDAM;
(* Procedure sets the initiate values of the pointers. It should
```

```
be called prior the other DAM operations *)

begin
  with DP do begin
    FreeAMPtr:=1;
  end
end;

procedure NewDAM(var Ptr:integer);
(* Procedure returns the index of the first free item of the array,
which will be the dynamically allocated space *)
begin
  with DAM do begin
    Ptr:=FreePtr;       (* Returned pointer - index to thr ArrItem *)
    FreePtr:=FreePtr+1;  (* Increasing of the pointer by the "length"
                        of allocated space *)
  end;
end;


procedure DisposeDPP(Ptr:integer);
(* This procedure is empty as we don't use the overhead part and the
   indicator Free *)

begin
end;
```

```
procedure MarkDPP;
(* Procedure saves the current value of the FreePtr to MarkPtr *)

begin
  with DPP do begin
    MarkPtr:=FreePtr;
  end;
end;

procedure ReleaseDPP;
(* Procedure reset the saved valure of the FreePtr *)
begin
  with DPP do begin
    FreePtr:=MarkPtr;
  end
end;
```

This mechanism is very simple and reliable. New operation is fast. Mechanism doesn't allow the individual regeneration of the heap by the dispose operation. It enables the economy usage of the space by means of the mark and release operations. Similar mechanisms are used in many not sophisticated applications and the above-mentioned simulated system may be used in the environments without DAM (Basic, Cobol, Fortran etc). While the simplified simulation system of DAM allocates the single type only (the items of the same length), the real system allows to allocate the memory items of different sizes.

4.2 Dynamic allocation of the memory with regeneration.

The dispose which returns (regenerating) individually the space of the items with different sizes causes the problem of fragmentation and de-fragmentation of the heap memory space. Problem is caused by the fact that returned (freed) space, with adjacent not-free neighbours may have the size not sufficient for the request of following new operation. It is necessary to move all free spaces and create one contiguous space. This process is called "defragmentation".

4.2.1. Principles of DAM in contiguous addressed space

Initiating operation divides the space to overhead part, which contains the indicator Free and value of the Heap length and to working space with the length (length(Heap)-length(overhead)).

Mechanism New searches for the first segment which is free and in size equal or greater to requested size. In searching, the mechanism goes sequentially through the whole contiguous address space (including not free - dynamically allocated items). By means of the value of the length, the move to next item is determined. If the item with the greater size is detected, it is split to two parts. Pointer to one is returned to the new operation and in both parts the value of the length is corrected. The indicator Free in returned part is set to false.

Mechanism Dispose, changes the free indicator to true. If one of the neighbour items is free too, the de-fragmentation process joints them into one contiguous space by means of the length information and addresses. The information on length of joined items should be actualised.

The above mentioned principle makes the time complexity of the algorithms less efficient.

4.2.2 Principle of DAM in address linked space.

Initializing operation creates the list of free segments each of which contains single item of the size of all reserved space. List may be double linked for easier insertion and deletion of items.

Operation new moves along the list and looks for the first item which is in size equal or greater than requested size. If the greater is found, it is divided to two parts: one part of requested size and the rest. The "overhead" information is actualized and the requested size is deleted from the list and the pointer to it is returned to "new" operation. The operation is faster than previous "new" operation, as it searches only in the list of free items.

Operation "dispose" inserts the disposed segment into the list of free items. In the de-fragmentation process it goes through the whole list looking for the free "neighbours". It uses the information on the address and length to determine the free neighbour. If free neighbour is found, the both segments are joined, and "overhead" information is updated. The simulated and simplified system uses single type of allocated items. The problem of fragmentation and de-fragmentation is thus postponed.

This mechanism may be simulated in the simplified way by the list of the items of the same length (type). The heap is represented by the array of items. There is the second array, items of whose serve as links of the list.

```
type
  const
     LenhthOfHeap=100;
     NilDAM=0;  (* Zero value will serve as index for nil pointer *)
  TDAM = record
    StartPtr : integer;
    ArrOfItems: array [1..LenghtOfHea] of TItem;
    ArrOfLinks:array[1..LengthOfHeap] of integer; (* array of links *)
  end; (* record *)

var
  DAM:TDAM;

procedure InitDPP;
(* Procedure links all items to one list. StartPtr points to the first
item. Link of the last item points to NilDAM (0). *)
var
  i:integer;
begin
  with DAM do begin
    for i:=1 to LengthOfHeap-1 do begin
      ArrOfLinks[i]:=i+1;          (* contiguous linking of the list items *)
      ArrOfLinks[LengthOfHeap]:=NilDAM;   (* setting the end of list *)
      StartPtr:=1;              (* setting of the beginning of te list *)
  end; (* for *)
end;

procedure NewDAM(var Ptr:integer);
begin
  with DAM do begin
    Ptr:=StartPtr;       (* returns te pointer to the first *)
    StartPtr:=ArrLinks[StartPtr]  (* Pointer to the first is pushed to
                                the next *)
  end
end;
```

```
procedure DisposeDAM(Ptr:integer);
begin
  with DAM do begin
    ArrOfLinks[Ptr]:=StartPtr;  (* Pointer to the disposed is set to
                          the actual first item *)
    StartPtr:=Ptr;        (* actualisation of the first *)
  end
end;
```

This mechanism is simple and very fast. It enables individual regeneration by means of the dispose operation. It is usable in simple applications in the environment not offering the real DAM system like basic, Fortran, Cobol etc.

4.3 Using the system simulating DAM.

We should use following transformation index notation for true pointer system:

(* Suppose we are inside of the body of "with DAM do begin... end;" *)

Real DAM		Simulated or "user defined DAM" - UDAM
Ptr is pointer		notation with pointers with arrow PtrI is index	index notation
nil	=>	NilDAM (* const NilDAM=0 *)
Ptr	=>	PtrI
Ptr^	=>	ArrItem[PtrI]
Ptr^.RPtr	=>	ArrItem[PtrI].RPtr
Ptr^.Rtr^.LPtr	=>	ArrItem[ArrItem[PtrI].RPtr].LPtr

Examples of using of both notations (DAM and UDAM) in the operation of deleting the item from double linked list.

```
type
  const
   NilUDAM=0;
  TListPtr=^TList;        (* type pointer to list item *)

  TItem=record          (* type item *)
    data:char;
    LPtr,RPtr:TListPtr      (* left and right pointer  *)
  end;

  TList=record          (* type list *)
    Frst,Lst:TListPtr     (* pointers to the first and last item of the
                        list *)
  end;
```

```
procedure deleteDAM(var L:TList; Ptr:TListPtr);
begin
  if Ptr<>nil then begin  (* is pointer to deleted not nil? ? *)
     if (Ptr=L.Frst) and (Ptr=L.Lst)
     then begin          (* deleted is the single item of the list *)
        L.Frst:=nil;  (* deletion oft single item *)
        L.Lst:=nil;
     end else begin
       if Ptr^.LPtr<>nil         (* is deleted the first? *)
       then Ptr^.RPtr^.LPtr:=Ptr^.LPtr   (* setting the left pointer of
                                    the item right to deleted item
                                    to the item to the left from
                                    the deleted item *)
       else begin              (* deleted is the first *)
         Ptr^.RPtr^.LPtr:=nil;      (* setting the pointer of the
                                right neighbour to nil *)
         L.Frst:=Ptr^.RPtr    (* correction of the pointer pointing to
                             the first *)
       end;
       if Ptr^.RPtr<>nil         (* is deleted the last one? ? *)
       then Ptr^.LPtr^.RPtr:=Ptr^lRPtr    (* setting the right pointer
of the item left to deleted to the right item to the deleted *)
       else begin              (* deleted is the last one *)
         Ptr^.LPtr^.RPtr:=nil;     (* setting the pointer of the left
                               item to nil *)
         L.Lst:=Ptr^.LPtr    (* correction of the pointer pointing to the
                             last item  *)
       end;
       dispose(Ptr)       (* releasing of the deleted item *)
    end;
  end;
end;
```

UDAM with indexes and with ADT DAM

Example is comment-less. Comments would be the same like in the previous example.

"Field" means ArrayItem.

```
const
  NilUDAM=0;
type
  TListPtr=integer;

  TItem=recod
    data:char;
    LPtr,RPtr:TListPtr
  end;
```

```
  TList=record
    Frst,Lst:TListPtr
  end;


procedure deleteUDAM(var L:TList; Ptr:TListPtr);
begin
  with UDAM do begin
    if (Ptr=L.Frst) and (Ptr=L.Lst)
    then begin
       L.Frst:=NilUDAM;
       L.Lst:=NilUDAM;
    end else begin
      if Ptr<>NilUDAM then begin
        if Field.[Ptr].RPtr<>NilUDAM
        then Field[Field[Ptr].RPtr].LPtr:=Field[Ptr].LPtr
        else begin
          Field[Field[Ptr].RPtr].LPtr:=NilUDAM;
          L.Frst:=Field[Ptr].RPtr
        end;
        if Field[Ptr].LPtr<>NilUDAM
        then Field[Field[Ptr].LPtr].RPtr:=Field[Ptr].RPtr
        else begin
          Field[Field[Ptr].LPtr].RPtr:=NilUDAM;
          L.Lst:=Field[Ptr].LPtr
        end;
      end; (* if *)
    end; (* if *)
  end; (* with *)
end
```

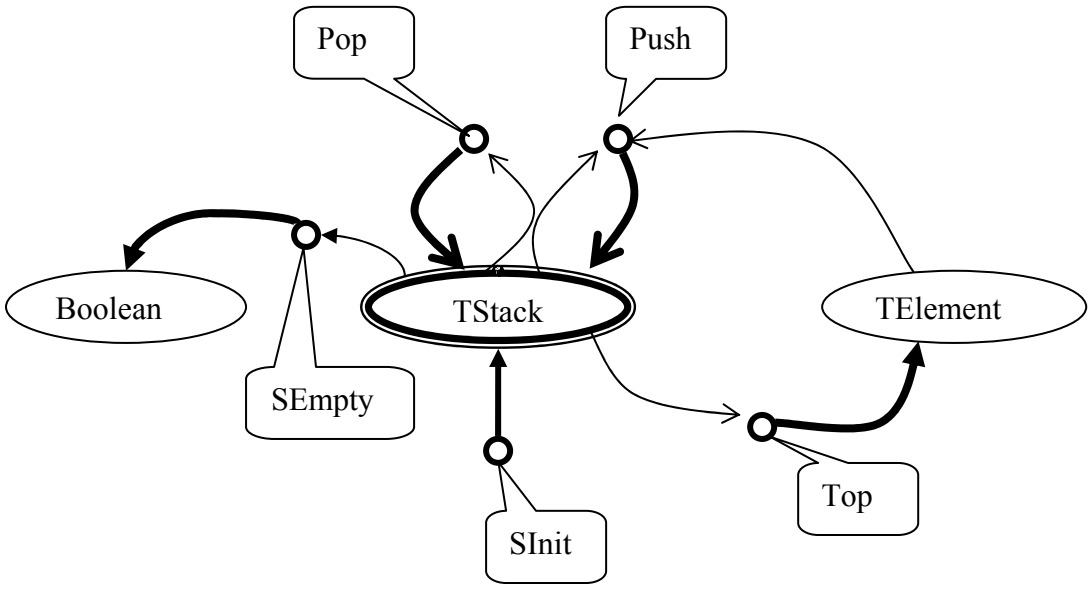
See to the files slist.pas, dlist.pas and misclist.pas at the course web site.

Lesson Five

5 Abstract Data Types

5.1 STACK

Stack is a linear, homogeneous, dynamic structure. Typical feature of stack is the access exclusively to one side of the linear structure. An example from life is the set of plates, located one on top of the other. You take a plate from the top of the column of plates and put the plate back to the top of the column. You never take/put one from/to the middle of the column. Similarly the items of a stack are accessible only from one side of the stack, called the TOP. Set of operations defined upon the stack is expressed in the Diagram of signature.



- **SInit(S) or StackInit(S) -** Initialization of the stack. The result of the operation is generation of the empty stack.
- **SEmpty(S) or StackEmpty** - The operation is predicate, which is true, if the stack is empty and false if the stack is not-empty.
- **Push(S,El)** - Inserting an item to the top of the stack. The operation is similar to InsertFirst in list.
- **Pop(S) -** Deleting the item at the top of the stack (in hereby defined semantics, the operation returns no value). The operation is similar to DeleteFirst in list.
- **Top(S,El)** The operation reads (returns) the value of the item onto the top of the stack. No change is made upon the stack. Operation has often the form of function, returning the value of top item. Operation causes a fatal error, if the stack is empty. A good programming style uses this operation after SEmpty operation:

```
if not SEmpty(s)
then begin
  Top(S,El);
  ...
end; (* if *)
```

The semantics of the ADT Stack is defined by a set of axioms

1) SEmpty(Sinit(S))=true;
2) SEmpty(Push(S,El))=false;
3) Top(Push(S,El))=El; (* Top is used as function *)
4) Top(S)=Top(Push(S,El);Pop(S));

5.1.1 The usage of stack.

Stack is one of the most important data structures. Its characteristic feature is the reversing of the order of items of linear structure. Stack is called LIFO structure (Last-In-First-Out). If items of linear structure (list) are sequentially pushed to the stack, and consequently by the couple of operation Top(S), Pop(S) taken from the stack and inserted to the list (in Insertlast manner) the results is the linear structure (list) with items reversed in their order. Backtracking algorithms are based on this concept. Another usage of this principle are in allocation of memory of called procedures especially in recursive calling, passing the graphs and trees with the return, enumeration of arithmetic (and other) expression expressed in postfix notation, conversion of expression expressed in infix notation to the postfix notation.

5.1.2 In-fix, pre-fix and post-fix notation.

Operation with two operands O1 and O2 and one operation P is notated as:

(O1 P O2) (5 + 13)

As operand is between operands, it is called infix notation. The greatest disadvantage of this commonly used notation is need for brackets, and difficulty in evaluation by computer algorithms.

Notation of the form of (P O1 02) is called pre-fix notation. We use this form e.g. in functions: add(O1,O2). Symmetrical notation is called post-fix or reverse Polish notation (called according the Polish mathematician, who introduced these notations.

If infix notation has the form:

(A + B) * (C + D) / ((E - F) / (G + H)) =

and postfix notation has the form:

A B + C D - * E F - G H + / / =

5.1.3 Evaluation of arithmetic expression in postfix by means of the stack

Let the arithmetic expression be noted in postfix notation and ended by delimiter '=', for example: 3 7 + 9 5 - * 7 1 - 6 2 / / =,

Then the evaluation is done according the rules:

Process sequentially the string of items in expression.

1) If an item is operand, push it to the stack.

2) If an item is operator, get out of the stack (Top and Pop) as many operands as is the n-arity of operator (two operands for dyadic or binary operator); produce the result of the operation with the two operands and push the result to the stack.

3) If you encounter the delimiter '=', the result is on the top of the stack.

With the above mentioned expression when '*' is processed, the situation at the stack is:

stack * 7 1 - 6 2 / / =

4←Top
10

when the first '/' is expressed, the situation is

stack / / =

3←Top
6
40

when the second '/' is expressed, the situation is

stack / =

2←Top
40

The resulting state is:

stack =

20←Top

5.1.4 The conversion of infix notation

The conversion of infix notation from the input string of items to postfix notation formed by output string of items with the use of the stack is done according the following rules.

Move sequentially along the input string of items and:

1) If an item is operand, add it to the end of generated output string.

2) If an item is left bracket, push it to the stack.

3) If an item is right bracket remove sequentially from the top of the stack all operators and add them to the end of output string until you encounter the left bracket. Remove the left bracket from the stack. This is the way how the pair of brackets is processed.

4) **If** *an item is operator*
then
if *stack is empty OR the left bracket is on the top of stack OR the operator with the lower priority is on the top of stack*
then *push the operator to the stack*
else (* on the top is operand with the same or greater priority *)
remove the operand from the top of stack and add it to the end of output string; repeat the algorithm from the point 4) as long as the insertion is successful (condition for insertion is true).

5) If the item is delimiter '=', remove sequentially all items from stack and add them one by one to the end of output string. When stack is empty, add the delimiter '=' to the end of output string.

5.1.5 Stack and allocation of memory in block structure.

Block structure is the inner part of the main program or the body of procedures. All local variables of given procedure are allocated by means of stack like organized memory space. All variables declared in the procedure are allocated before the first statement of the procedure body is executed. When the procedure ends, the top of stack-like memory is moved back to the position prior to the procedure call. This ensures the de-allocation of memory allocated to all local variables.

5.1.6 Implementation of the stack

Stack may be implemented by an array or by linked list.

5.1.6.1 Implementation by the array

```
const
  MaxStack=500;
type
  TSArray=array[1..MaxStack] of TData;
  TStack=record  (* stack implemented by an aray *)
    SArray:TSArray;
    TopInd:0..MaxStack
  end;

  procedure SInit(S:TStack);
  begin
    S.TopInd:=0;
  end;

  function SEmpty(S:TStack):Boolean;
  begin
   SEmpty:=S.TopInd=0;
  end;

  procedure Push(var S:TStack; El:TData);
  begin
   with S do begin
     TopInd:=TopInd + 1;
     SArray[TopInd]:=El;
   end; (* with *)
  end;

  procedure Pop(var S:TStack);
  begin
   if S.TopInd>0
   then
     S.TopInd:=S.TopInd - 1;
  end;

  procedure Top(S:TStack; var El:TData); (* sometimes implemented as
                                    the function *)
```

```
  begin
    El (* Top *) := S.SArray[S.TopInd];
  end;

  function  SFull(S:TStack):Boolean;
  begin
     SFull:=S.TopInd=MaxStack;
  end;
```

This function is implemented only for implementation using an array. Operation Push should be guarded by SFull condition:

```
    if not SFull(S)
    then begin
      Push(S,El);
      ...
    end;
```

5.1.6.2 Implementation by the list:

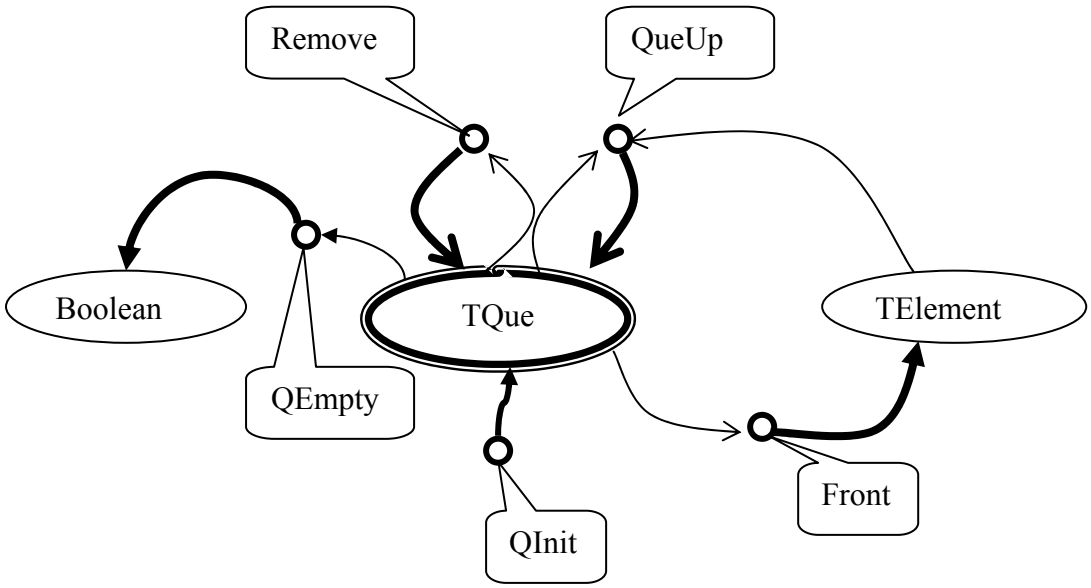
```
type
  TPtr=^TItem;
  TItem=record
    Data:TData;
    NextPtr:TPtr;
  end;

  TStack=record  (* Stack implemented by the list *)
    Top:TPtr;
  end;

  procedure SInit(var S:TStack);
  begin
    S.Top:=nil;
  end;

  function SEmpty(S:TStack):Boolean;
  begin
    SEmpty:=S.Top=nil;
  end;

  procedure Push(var S:TStack, El:TData);
  var
    TmpPtr:TPtr;
  begin
    new(TmpPtr);
    TmpPtr^.Data:=El;
    TmpPtr^.NextPtr:=S.Top;
    S.Top:=TmpPtr;
  end;
```

```
procedure Pop(var S:TStack);
var
  TmpPtr:TPtr;
begin
  TmpPtr:=S.Top;
  S.Top:=S.Top^.NextPtr;
  dispose(TmpPtr);
end;

procedure Top(S:TStack; var El:TData);
begin
  El:=S.Top^.Data
end;
```

5.2 QUEUE

Queue is a linear, homogeneous, dynamic structure. Queue is called FIFO structure : (First-In-First-On). Item is inserted to one side of the linear structure called the end of queue and item is removed or read from the other side called the begining of queue.

There is set of operations defined upon the queue in the Diagram of signature:



- **QInit(Q)** ... Initialization of the queue. The result of the operation is the generation of the empty queue.

- **QEmpty(Q)** ... The operation is predicate which is true, if queue is empty and false, if queue is not-empty.

- **QueUp(Q,El)** ... The operation inserts the element into the queue as the edge item to the "end of the queue".

- **Remove(Q)** ... The operation remove (destroys) the edge item at the "beginning of the queue"

- **Front(Q,El)** ... The operation returns the value of the edge element at the begin of the queue. Operation causes the fatal error if the queue is empty. Operation is regularly conditioned by the not empty predicate:

```
if not QEmpty(Q)
then begin
  Front(Q,El)
  ...
end;
```

Implementation of queue by array uses predicate QFull, which indicates the full queue and prevents the Queup operation.

```
if not  Full(Q)
then begin
  Queup(Q,El)
  ...
end;
```

The queue is the essential concept of the "Queue theory" and of the discrete event simulation systems solving the problems of optimal service in the service systems.

5.2.1 Implementation of the queue.

Implementation with the use of an array,

```
const
  MaxQue=500;  (* Capactity of the Queue is 499 *)
type
  TQArr:array[1..MaxQue] Tdata;
  TQueue=record   (* type Queue implemented by ab array *)
    Frst,Lst:1..MaxQue;
    QArr:TQArr;
  end;
```

```
procedure QInit(vat Q:TQueue);
begin
```

```
  Q.Frst:=1;
  Q.Lst:=1;
end;
```

```
function QEmpty(Q:TQue):Boolean;
begin
  QEmpty:=S.Lst=S.Frst
end;
```

```
procedure QueUp(var Q:TQueue; El:TData);
begin
  with Q do begin
    QArr[Lst]:=El;
    Lst:=Lst+1;
    if Lst > MaxQue  (* back linking of the circular list *)
    then Lst:=1;
  end
end;
```

```
procedure Remove(var Q:TQue);
begin
  with Q do begin
    if Frst<>Lst
    then begin
      Frst:=Frst+1;
      if Frst>MaxQue
      then Frst:=1
    end; (* if *)
  end (* with *)
end;
```

```
procedure Front(Q:TQue; var El:TData);
begin
  El:=Q.Qarr[Q.Frst];
end;
```

```
function QFull(Q:TQue):Boolean; (* This operation is used for
                    implementation of the queue by the array *)
begin
  with Q do begin
    QFull := (Frst=1) and (Lst=MaxQue) or ((Frst-1)=Lst)
  end;
end;
```

5.2.2 Implementation of the queue by the linked list.

```
type
  TPtr=^TItem;
  TItem=record
    Data:TData;
    NextPtr:TPtr;
  end;

  TQueue=record  (* Queue implemented by the list *)
    Frst,Lst:TPtr;
  end;
```

```
  procedure QInit(var Q:TQue);
  begin
    Q.Frst:=nil;
    Q.Lst:=nil;
  end;
```

```
  function QEmpty(Q:TQue):Boolean;
  begin
    SEmpty:=S.Frst=nil;
  end;
```

```
  procedure Queup(var Q:TQue, El:TData);
  var
    TmpPtr:TPtr;
  begin
    new(TmpPtr);
    TmpPtr^.Data:=El;
    TmpPtr^.NextPtr:=nil;
    if Q.Frst=nil
    then Q.Frst:=TmpPtr; (* inserted is the first and single item *)
    Q.Lst^.NextPtr:=nil;
    Q.Lst:=TmpPtr;
  end
```

```
  procedure Remove(var Q:TQueue);
  var
    TmpPtr:TPtr;
  begin
    TmpPtr:=Q.Frst;
    if Q.Frst=Q.Lst
    then begin
      Q.Lst:=nil
    end;
      Q.Frst:=Q.Frst^.NextPtr;
    end;
    dispose(TmpPtr);
```

```
  end;
```

```
procedure Front(Q:TQue; var El:TData);
begin
  El:=Q.Frst^.Data
end;
```

5.3 Exercises:

1. Convert infix notation to postfix and evaluate the postfix result:

   procedure Evaluate(S;string; var Result:real);

2. Draw the diagram of signature of TStack

3. Create the Map Function for the three dimensional array

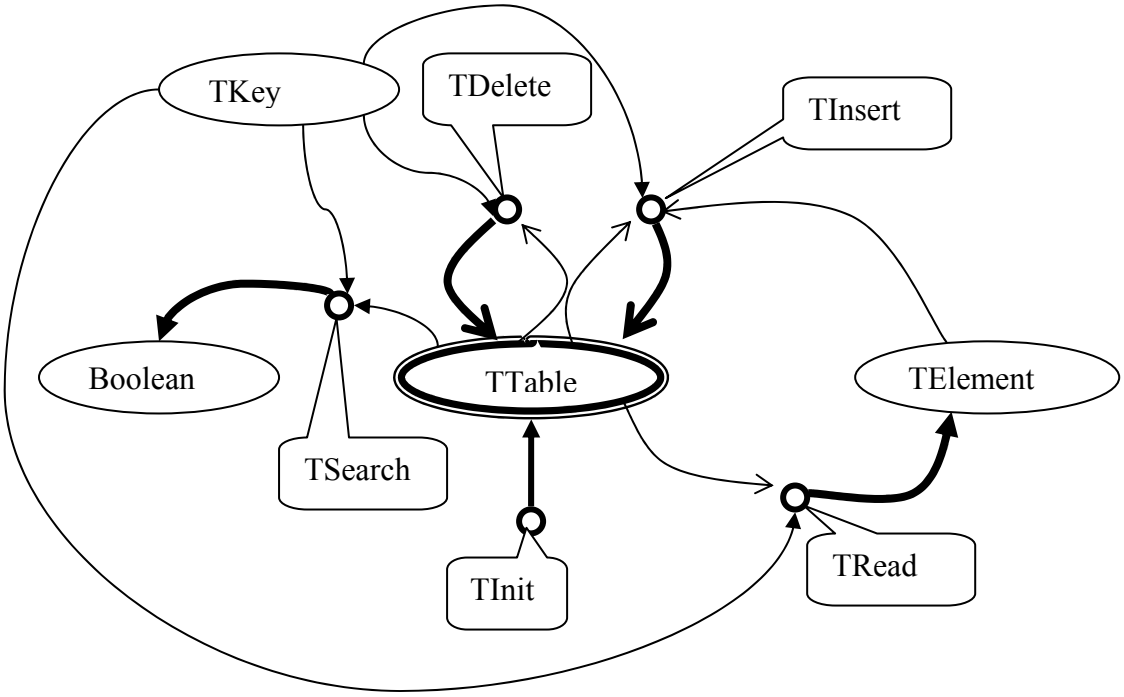
   [1..10,1..20.1..5]

6. ALGORITHMS AND DATA STRUCTURES

Lesson Six:

6.1 Look-up tables

The Look-up-table (search table or simply "table") is a homogeneous, dynamic structure. Element of the table has a component called "key", which is unique for every element in the table. There doesn't exist a single one couple of elements with the same key. The key serves for unique identification of every element. Another component of an item is data. The set of operation
defined upon the table consists of:



Let T be the name of table, k is the identifier of the key and El the identifier of data.

- TInit(T) - (Initialization of the table; the result is an empty table)
- TInsert(T,K,El) (The item with the key K and data El is inserted to the table. If table contains the item with the key K, the old data component is replaced by the value El. This is called "the actualization semantics of TInsert operation").
- TDelete(T,K) (If the table contains an item with the key K, it is deleted from the table. Ifthere is not such item, nothing happens)
- TSearch(T,K) Predicate Search (function Search) returns value true, if an element with the key K exists in the table; otherwise it returns false.

- TRead(T,K,El) (If the table contains the item with the key K, operation returns the value of data in the output parameter El. An error occurs, if operation TRead tries to read the item, which doesn't exist!!!
  Any TRead operation should be conditioned by the Search test operation:

  ```
  if Search(T,K)
  then TRead(T,K,El);
  ```

The table is one of the most important data structures. It is something the database is based on. A large part of syllabus will be dedicated to various methods of implementation of tables and their operations.

6.2 Array

Array is an orthogonal, homogeneous, mostly static structure. If an array is n-dimensional, it is taken as one dimensional array, whose items are (n-1) dimensional arrays. Item of n-dimensional array is accessed by means of name (identifier) of array and n indexes. Every dimension is defined by an interval upon any ordinal type. Let us define:

TArr=array[1..10,1..20] of integer.

Let us declare the structured variable Arr:TArr. The notation Arr[5,6] means the 6th item in 5th row of matrix with 10 rows and 20 columns. The notation Arr[6] denotes the vector of 20 items,
which is the 6th row of the matrix. It is possible to write:

Arr[1]:=Arr[10] which means that the first row will be rewritten by the 10th row.

There are three most important operations defined upon the type array:

1) InitArray(ArrName, range1, range2,...,rangeN, TElem);
   where rangeI is interval of Ith dimension.

The result is allocation of the memory space for array for the base type of TElem. All items have undefined values after initialization.In Pascal the initialization is made by declaration of an array.

2) WriteArr(ArrName, ind1,ind2,...,indN, Val).

The result of an operation is setting the value of specified item to value ov Val.

In Pascal the same effect has the statement:
ArrName[ind1,ind2,...,indN]:=Val
Constructor is the special operation upon any structured type, which builds the structured value of the structure. Let us give the type TArr=array[1..2,1..3] of char and let us declare an Arr:TArr, then the result of operation like: Arr:=(Arr, 'A','B','C','D','E','F') is an array:

'A'	'B'	'C'
'D'	'E'	'F'

ANSI Pascal have no constructor for arrays, with exception of string, which has the features of one dimensional array of char, where the range of dimension starts with the value 1.

Therefore for s:string is allowed to write:
s:='ABCDEF'
which has the resulting effect of

s

'A'	'B'	'C'	'D'	'E'

3) ReadArr(ArrName, ind1,ind2,...,indN, Val).
The result of an operation is setting the value of Val to value of specified item of array. This operation has the features of "selector".Selector makes conversion from structure to single element.

In Pascal the same effect has the statement:
Val:=ArrName[ind1,ind2,...,indN];

An effort of reading the value of not-defined item results in error. (Some systems return any value and may not refer an error state).

6.2.1 Mapping function

A multidimensional array is an abstraction. Items of multidimensional array in memory are stored on adjacent positions, like vector. There is the function, transforming the n-indexes of item of multidimensional array to one index of vector.

Let there be an 2-dimensional array TArr[1..n,1..m] where n=2 is dimension of the rows and m=3 is dimension of columns. The array is represented by matrix

Arr 1 m=3

'A'	'B'	'C'
'D'	'E'	'F'

n=2

The storing of items of this array has the form of vector Vec

Vec
1 k=m*n

'A'	'B'	'C'	'D'	'E'	'F'

where the dimension of vector is k=m*n.

The mapping function Map of item Arr[i,j]=Vec[Map[i,j] has the form:
k=Map[i,j]= m * (i-1) + j
It means i-1 of full rows plus column index of item.

Let B be the L-dimensional array:

B:array[low1..high1, low2..high2,...,lowL..highL] of TElem;

The item B[j1,j2,...,jL] will be mapped into one-dimensional array A with the index:

$$1+\sum_{m=1}^{L}(j_m - low_m)*d_m \qquad (1)$$

a so

$$B[j_1, j_2,..., j_L] \rightarrow A[\ 1+\sum_{m=1}^{L}(j_m - low_m)*d_m\ ]$$

where:

$d_1 = 1$
$d_M = (high_M - low_{M+1}) * d_{M-1}$

This mapping function expresses the storing of multidimensional function "by the rows" or "with the decreasing speed of change of the indexes from the right side". (passing the matrix, the column index is increased faster then the row index ").

6.2.2 Problem for solving:

Write the mapping function for FORTRAN-like array, which is stored not by rows but by columns!

6.2.3 Dope vector (information vector, dope record, information record).

From the above mentioned paragraph it is clear, that the values ($low_m * d_m$) are independent of the value of the subscript. They may be counted before the usage of an array.

Formula (1) may be reformulated to the form:

$$1+\sum_{m=1}^{L}(j_m * d_m) - \sum_{m=1}^{L}(low_m * d_m) \qquad (2)$$

It is possible to preprocess the values of d_m and $\sum_{m=1}^{L}(low_m * d_m)$

for m=1,2,...,L. This evaluation is done once in the time of initiation. In this way the access time to element of multidimensional array may be shorter. Typical dope vector (information vector) contains:

1) Number of dimensions
2) Lower and upper limit for every dimension (L couples)
3) Total number of elements and size of one element
4) d_m for m=1,2,...,L
5) $\sum_{m=1}^{L}(low_m * d_m)$
6) Address of the first element (address of the beginning of array)

6.2.4 Triangle matrix

Triangle matrix, matrix with the different length of rows and thin (low dense) array are the methods saving the space in memory. The access time to an item of the structure will be longer as the consequence of saving the space.

Let the square matrix of shape be:

a_{11}
a_{21} a_{22}
a_{31} a_{32} a_{33}
...
a_{N1} a_{N2} a_{N3} .... a_{NN}

To spare the space, we may use only *(N div 2) * (N+1)* items instead of *N*N* items . The mapping function for this method is:

```
a[j,k] ->  b[ j*(j-1) div 2 + k];
```

It may happen that there are necessary two triangle matrixes of the same type. The array

```
c[1..N, 1..(N+1)] of TElem
```

may be used with the two different mapping functions:

a[j,k] -> c[j,k]
b[j,k] -> c[k,j+1]

6.2.5 The matrix with different length of rows (rag tables, jagged tables).

Let's have the following matrix:

a_{11}	a_{12}							
a_{21}	a_{22}							
a_{31}	a_{32}	a_{33}						
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{51}								
a_{61}								

where empty field is the not defined (not used) item of matrix. To spare the memory space we may use the vector as storing strategy:

vect

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a_{11}	a_{12}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}	a_{51}	a_{61}

We may form the mapping function in the form:

a[i,j] -> vect[AV[i]+j]

where AV is access vector, which for the introduced example would have the values

AV

1	2	3	4	5	6
0	2	4	7	8	17

AV[i] is sum of number of all defined items in all i-1 rows.

For example: a[5,4] = vect[AV[5]+4}=vect[8+4] =vect[12];

6.2.6 Thin array (Low dense array)

Let there exist an array, which contains a great majority of items of the same value. Such value is called a dominant value of an array. To spare the space in memory, it is possible to use the strategy of "thin (low dense) array". In the new table-like structure, only items with not dominant value will be stored. Items with dominant value will not be contained in the

structure, which causes the sparing of the memory space. The structure acts with the index(es) like with the key in the table.

If the Read-like access to item of the array on given index is to be done, index is searched in the table. If the search is successful, operation returns the result of TRead operation, otherwise it returns the dominant value!

If the Write-like access to the item of array on given index is to be done, then, if the written item is of not dominant value, it is written by the TInsert operation, where index serves as the key and item value as data. Otherwise the operation TDelete of item with the index=K is processed.

Thin (low dense) array is implemented by the ADT Table this way:

- InitArr(Arr) -> TInit(T)

- ReadArr(Arr,Ind,El) -> if Search(T,Ind)
  then El:=TRead(T,Ind)
  else El:= dominant value;

- WriteArr(Arr,Ind,El) -> if El=dominant value
  then TDelete(T,Ind)
  else TInsert(T,Ind,El)

6.2.7 Dynamic array

Dynamic array may be created and destroyed during the run of program. Its dimensions are defined during the initialization operation. An operation, returning the size(s) of initialized (created) array should be added to the set of operations upon the array. Any dynamic structure can be created only with the use of essential dynamic new-like operations.

Statement CreateDynArr(DArr,1..5) may be implemented by creating the list of five items. The index is defined by order of item in list. To access the item, the passing the list is necessary. More advanced techniques are based on address arithmetics and size flexible dynamic allocation of memory space.

6.3 Graph and its implementation

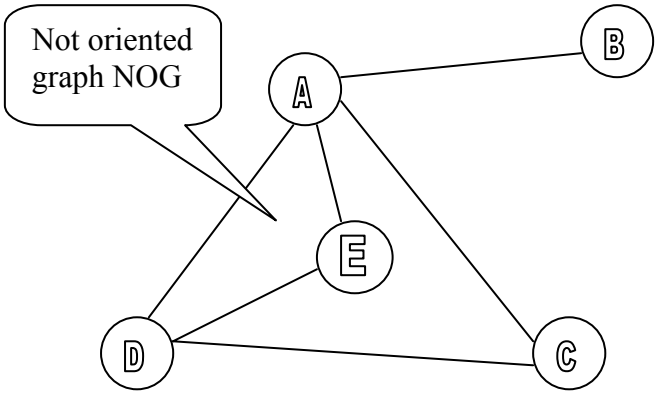
Graph may be defined by three-tuple G(N,E,I),

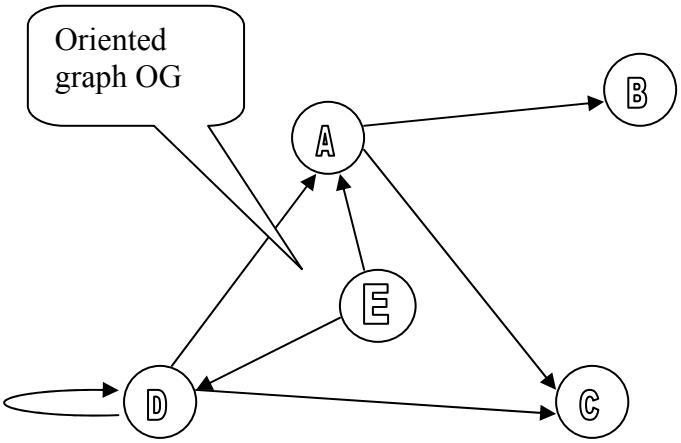
where

- N is the set of nodes, which may have assigned the value
- E is the set of edges, which may have assigned the value. Every edge connects two nodes which may not be necessarily different (edge may start and end in the same node). If the edge is oriented (directed from one node to another or the same node, the graph is called oriented graph).

- I is the set of interconnections, which defines completely the interconnections of related nodes by edges. In oriented graph the interconnection expresses orientation of edge.

Example of the graph.





Pass through all nodes of the graph and/or of the tree which visits every node only once – is an operation transforming a non-linear structure of the graph and/or tree to the linear one.

Graph may be implemented mostly by three possible ways:

6.3.1 Static implementation of graph.

Static implementation uses matrixes.

There are two methods for graph implementation by matrixes:

Interconnection matrix is a square matrix, where the size of both dimensions is determined by the number of nodes. Its items have two values (0,1 or true, false) for not oriented graphs and three values (-1, 0, 1) for oriented graphs. The value 0 on the index i, j expresses the fact that the nodes i and j are not interconnected. Another value expresses the interconnection of relevant couple of nodes i and j. We may state, that positive value 1 of item on indexes i, j expresses orientation of the edge between nodes i and j, from i to j, and negative value -1 expresses the orientation of the edge from j to i. The non-zero value on the diagonal of matrix for oriented graph has the same
significance. (The edge starts and is directed to the same node).

Interconnection matrix implementing not oriented graph NOG

	A	B	C	D	E
A	0	1	1	1	1
B	1	0	0	0	0
C	1	0	0	1	0
D	1	0	1	0	1
E	1	0	0	1	0

Interconnection matrix implementing oriented graph OG

	A	B	C	D	E
A	0	1	1	-1	-1
B	-1	0	0	0	0
C	-1	0	0	-1	0
D	-1	0	1	1	1
E	1	0	0	1	0

Neighbour matrix expresses the edges to neighbour node for every node. Each node has one row. This square matrix has often the form of "matrix with different size of rows". The item of matrix has the value denoting the node (content A means the node A).

The oriented graph OG expressed by the matrix of neighbour has the form:

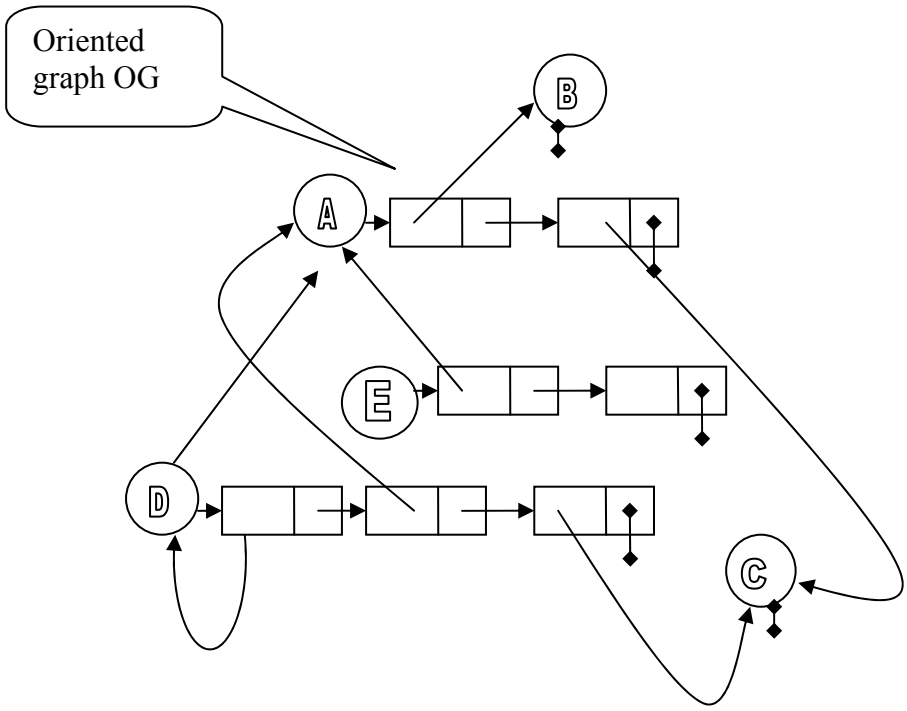
A	B	C	-	-	-
B	-	-	-	-	-
C	-	-	-	-	-
D	A	C	D	-	-
E	A	D	-	-	-

This means:

- The node A has edges directed to nodes B and C.
- The node B has no edges directed to neighbours.
- The node C has no edges directed to neighbours.
- The node D has edges directed to nodes A,C and to D itself.
- The node E has edges directed to nodes A and D.

Dynamic implementation of graph consists of the nodes represented by a list like structure. Any node list consists from the denotation (identification) of the node in the headings item, and pointers to neighbour nodes in items of the rest of list. It is in fact the dynamic representation of the matrix of neighbours.

For example the A of the above mentioned matrix has the form:



Acyclic graph is the graph with no cycle. Cycle is the path through the graph, which starts and ends in the same node.

6.4 Rooted tree and its implementation

Rooted tree is oriented acyclic graph, which has one node called root. There exists the only one path from the root to any other node. The consequence of the rule is, that for every node there exist only one edge directed into the node, and any number of nodes directed out for the node. The node is called "parent" or "father" node and edges directed out of the node point to "children" nodes. Trees may be implemented by the same methods as graphs.

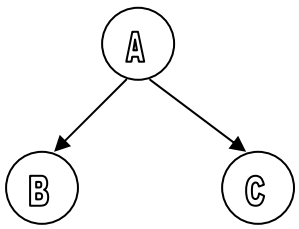
6.5 Binary tree (BT)

Recursive definition:

Binary tree is either empty or it consists of the node called root and of two sub-trees, left and right, both are binary trees.

Binary tree consists of the root, current nodes, which points at one or two children sub-trees (nodes) and terminal nodes, which have no "children". Distance of the node from the root defines the level of the node. The root is in level 1. The height of the tree is defined by the maximal distance of terminal node. Binary tree with a single node-root has the height of the value 1.

Let there be the binary tree with three nodes A,B,C, where A is the root, B is the left sub-node and C is the right sub-node:



There are three most important passes through the tree:

- Pre-order : A,B,C
- In-order : B,A,C
- Post-order: B,C,A

and there are three symmetric or inverted passes:

- Inverted pre-order : A,C,B
- Inverted In-order : C,A,B
- Inverted Post-order: C,B,A

Notice that the pre-order and inverted post-order are symmetric. Similarly the post-order and inverted pre-order are symmetric, while the in-order is symmetric with its inverted version.

Let there be types necessary to implement the binary tree (the similar that these for double-linked lists).

```
type
  TPtr=^TNode;  (* pointer to node *)
  TNode=record  (* type of node *)
         Data:Tdata;
         LPtr,RPtr:TPtr;
        end;
```

The recursive procedure which creates a list containing the pre-order sequence of all nodes of binary tree defined by the pointer to its root has the form:

```
    procedure Preoder(RootPtr:TPtr; var L:TList);
    (* suppose the InitList(L) was done before the call of this
       procedure *)
    begin
      if RootPtr is <> nil
      then begin
        DInsertLast(L,RootPtr^.Data);
        Preorder(L,RootPtr^.LPtr);
        Preorder(L,RootPtr^.RPtr)
      end;
    end; (* procedure *)
```

By changing the order of three statements in the if statement other passes are created:

```
    (* Inorder *)
    Inorder(L,RootPtr^.LPtr);
    DInsertlast(L,RootPtr^.Data);
    Inorder(L,RootPtr^.RPtr);

    (* Postorder *)
    Postorder(L,RootPtr^.LPtr);
    Postorder(L,RootPtr^.RPtr);
    DInsertlast(L,RootPtr^.Data).
```

ALGORITHMS AND DATA STRUCTURES
Lesson seven

7 Exercise: Implementation of 13 tree-etudes and some of their variants.

Declaration of the necessary types

```
type
(* Double linked list DList *)
  TDItemPtr=^TDItem;
  TDItem = record
           Data:integer;
           LItPtr,RItPtr:TItemPtr;
  end;

  TDlist=record
         Frst,Lst,Act:TDItPtr;
        end;

  TPtr=TDItemPtr; (* Type for tree item is equivalent to  type
                     for double list item *)

(* Single linked list SList *)

  TSItemPtr=^TSItem;
  TSItem = record
           Data:integer;
           RItPtr:TSItemPtr;
  end;

  TSlist=record
         Frst,Act:TSItPtr;
        end;


 TStackPtr=   (* type stack of Pointers to tree (TPtr) *)
 TStackBool = (* type stack of boolean values *)
```

7.1 Non-recursive Pre-Order. The result is inserted into ADT list by means of abstract operations.

```
procedure PreOrder(RootPtr:TPtr; var DL:TDList);
procedure LeftMost(RootPtr:TPtr);
(* procedure passes along the main left diagonal, pushing the node pointers to the stack *)
begin
   while RootPtr<>nil do begin
     PushPtr(S,RootPtr);
     DInsertLast(DL,RootPtr^.Data);   (* Output to DL *)
     RootPtr:=RootPtr^.LPtr
  end;
end;

var
  TmpPtr:TPtr;
  DL:TDlist;
  S:TStackPtr;
begin
   DInitList(L);  (* Initiation of the list *)
   SInit(S);     (* Initiation of the stack *)
   LeftMost(RootPtr);  (* Initial call of LeftMost *)

   while not SEmpty(S) do begin (* passing cycle *)
     Top(S,TmpPtr); Pop(S);
     LeftMost(TmpPtr^.RPtr)
   end;
end; (* procedure *)
```

7.2 Non-recursive In-Order. The result inserted into ADT single linked list by means of abstract operations (with the use of list-heading).

```
procedure InOrder(RootPtr:TPtr; var SL:TSList);

procedure LeftMost(RootPtr:TPtr;var DL:TDList);
(* procedure passes along the main left diagonal, pushing the node
pointers to the stack *)
begin
   while RootPtr<>nil do begin
     PushPtr(S,RootPtr);
     RootPtr:=RootPtr^.LPtr
  end;
end;

var
  TmpPtr:TPtr;
  S:TStackPtr;
begin
   InitList(SL);  (* Initilization of the list *)
   SInsertFirst(SL,0); (* Creation of headings with the Data=0*)
   First(SL);
   SInit(S);     (* Initilization of the stack *)
   LeftMost(RootPtr,DL);  (* Initial call of LeftMost *)

   while not SEmpty(S) do begin (* passing cycle *)
     Top(S,TmpPtr); Pop(S);
     SPostInsert(SL,TmpPtr^.Data); (* Output to SL *)
     SSucc(SL);
     LeftMost(TmpPtr^.RPtr,DL)
   end;
   SDeleteFirst(SL); (* Erase heading *)
end; (* procedure *)
```

7.2 a Variant of Pre-Order with one cycle (Double linked list.)

```
procedure  Preorder(RootPtr:TPtr;var DL:TDlist);
var
  S:TStackPtr;
begin
   SInit(S);
   PushPtr(S,RootPtr);
   DInitList(DL);
   repeat              (* passing cycle *)
     Top(S,RootPtr); Pop(S);
     if RootPtr<>nil
     then begin
        DInsertLast(DL, RootPtr^.Data); (* output to DList *)
        PushPtr(S,RootPtr^.RPtr);
        PushPtr(S,RootPtr^.LPtr);
     end;
  until SEmpty(S);
end;
```

7.3 Non-recursive Post-Order with double linked output list .

```
procedure PostOrder(RootPtr:TPtr; var DL:TDList);

  procedure LeftMostPost(RootPtr:TPtr);
  (* Auxiliary procedure passing along the left diagonal and storing  in stacks node pointers
and true values *)
  begin
    while RootPtr<>nil do begin
      PushPtr(S,RootPtr); (* Stack with pointers *)
      PushBool(SB,true);   (* Stack with Boolean *)
      RootPtr:=RootPtr^.LPtr
    end; (* while *)
  end; (* LeftmostPost *)

var
  FromLeft:Boolean;  (* Aux. Bool. variable *)
  TmpPtr:TPtr;
  S:TStackPtr;  (* stack for pointers *
  SB:TStackBool;  (* stack for Booleans *)
begin
  DListInit(DL);
  SInitPtr(S);
  SInitBool(SB);
  LeftMostPost(RootPtr);




  while not SEmptyPtr(S) do begin
    TopBool(SB,Fromleft); PopBool(SB);
    TopPtr(S,TmpPtr);
    if FromLeft
    then begin
      PushBool(SB,false);
      LeftMostPost(TmpPtr^.RPtr);
    end else begin
      DInsertLast(DL,TmpPtr^.Data);
      PopPtr(S)
    end; (* if *)
  end (* while *)
end; (* procedure PostInsert *);
```

7.3 a Variant of Post-Order with use of two pointer stack and inversed Pre-Order, Output to DList;

```
procedure PostOrder2(RootPtr:TPtr; var DL:TDlist);
var
  S1,S2:TStackPtr;
  TmpPtr:TPtr;
  Val:integer; (* TData *)

  procedure RightMost(RootPtr:TPtr);
  begin
    while RootPtr<>nil do begin
      PushPtr(S1,RootPtr);  (* first stack for backtracking *)
      PushPtr(S2,RootPtr);  (* Second stack for temp. output *)
      RootPtr:=RootPtr^.RPtr;
    end (* while )
  end; (* procedure RightMost *)

begin
   SInitPtr(S1);
   SInitPtr(S2);
   RightMost(RootPtr);
   while not SEmpty(S1) do begin
     TopPtr(S1,TmpPtr);
     RighMost(TmpPtr^.LPtr)
   end; (* while *)

(*   Reversing the result from Stack 2 *)

   while not SEmptyPtr(S2) do begin
     TopPtr(S2, Val); PopPtr(S2);
     InsertLast(DL,Val);
   end (* while *)
end; (* procedure PostOrder *)
```

7. 4 General procedure for all three passes. Output to DL.

```
type
  TPass=(PreO,InO,PostO);
```

```
procedure GeneralPass(RootPtr:TPtr; Pass=TPass; var DL:TDList);

  procedure LeftMostPost(RootPtr:TPtr);
  (* Auxiliary procedure passing along the left diagonala and storing
     in stacks node pointers and true values *)
  begin
    while RootPtr<>nil do begin
      PushPtr(S,RootPtr); (* Stack with pointers *)
      PushBool(SB,true);   (* Stack with Boolean *)
      if Pass=PreO
      then DInsertLast(DL,RootPtr^.Data); (* output for PreOrder *)
      RootPtr:=RootPtr^.LPtr
    end; (* while *)
  end; (* LeftmostPost *)
var
  FromLeft:Boolean;  (* Aux. bool. variable *)
  TmpPtr:TPtr;
  S:TStackPtr;  (* stack for pointers *
  SB:TStackBool;  (* stack for booleans *)
begin
  DListInit(DL);
  SInitPtr(S);
  SInitBool(SB);
  LeftMostPost(RootPtr);
  while not SEmptyPtr(S) do begin
    TopBool(SB,Fromleft);PopBool;
    TopPtr(S,TmpPtr);
    if FromLeft
    then begin
      if Pass=InOrder
      then DInsertLast(DL,TmpPtr^.Data); (* Output for InOrder *)
      PushBool(SB,false);
      LeftMostPost(TmpPtr^.RPtr);
    end else begin
      if Pass=PostO
      then DInsertLast(DL,TmpPtr^.Data);  (* Output for PostOrder *)
      PopPtr
    end; (* if *)
  end (* while *)
end; (* procedure PostInsert *);
```

7.5 Equivalence of the structure of two trees. Recursive version.

```
  (* in Comment brackets extension for equivalence of two trees - data
     are involved *)

function EQTS(RootPtr1,RootPtr2:TPtr):Boolean;
begin
 if (RootPtr1=nil) or (RootPtr2=nil)
 then
   EQTS:=(RooTPtr1=nil) and (RootPtr2=nil)
 else
   EQTS:=EQTS(RootPtr1^.LPtr,RootPtr2^.LPtr) and
         EQTS(RootPtr1^.RPtr,RootPtr2^.RPtr)
         (* and (RootPtr1^.Data=RootPtr2^.Data);  - for equality of
          the two trees *);
end;
```

7. 5 a Variant with encapsulated procedure.

```
function EQUTS(RootPtr1,RootPtr2:TPtr):Boolean;
  procedure AuxEQ(RootPtr1,RootPtr2:TPtr; var EQU:Boolean);
  (* As EQU should be set to true before AuxEqu is called, this
     procedure is "wrapped" by outer procedure EQUTS *)

  begin
    if ((RootPtr1=nil) and (RootPtr2=nil) or
       (RootPtr1<>nil) and (RootPtr2<>nil) and EQU
    then begin
      if RootPtr1<>nil
      then begin
        AuxEQ((RootPtr1^.LPtr,RootPtr2^.LPtr,EQU);
        AuxEQ((RootPtr1^.RPtr,RootPtr2^.RPtr,EQU);
      end else begin
        EQU:=false
      end (* if *);
   end; (* EuxEQ *)
var
  EQU:Boolean;
begin
  EQU:=true;    (* Initialization of EQU *)
  AuxEQ(RootPtr1,RootPtr2,EQU);   (* call of main procedure *)
  EQUTS:=EQU;                     (* forming output *)
end; (* EQUTS *)
```

7.6 Equivalence of the structure of two trees. Non-recursive version.

```
function EquTreeStrNonRec(RootPtr1,RotPtr2:TPtr):Boolean;
  procedure LeftMostEQUS(RootPtr1,RootPtr2:TPtr; var EQU:Boolean);
  begin
    while (RootPtr1<>nil) and (RootPtr2<>nil) do begin
    (* cyclus ends with the end of any diagonala *)
      PushPtr(S1,RootPtr1);
      PushPtr(S2,RootPtr2);
      RootPtr1:=RootPtr1^.LPtr; (* move along the left diagonala 1 *)
      RootPtr2:=RootPtr2^.LPtr; (* move along the left diagonala 2 *)
    end; (* while *)
    EQU:= (RootPtr1=nil) and ( RootPtr2=nil); (* if both diagonalas end
              simultanously, then structures are still equivalent *)
  end; (* LeftmostEQUS *)

var
   EQU:Boolean;

begin (* body of main function *)
  SInitPtr(S1);  (* Stacks initialization *)
  SInitPtr(S2);

  LeftMostEQUS(RootPtr1,RootPtr2,EQU);
  while not SEmptyPtr(S1) and EQU do begin
    TopPtr(S1,RootPtr1);
    TopPtr(S2,RootPtr2);
    LeftMostEQUS(RootPtr1^.RPtr,RootPtr2^.RPtr,EQU);
  end; (* cycle while finishes with the end of pass or with EQU=false*)
  EquTreeStrNonRec:=EQU
end;
```

7.7 Equivalence of the two trees. Non-recursive version.

The main procedure is the same as the 6). Different is the Leftmost:

```
procedure LeftMostEQUTree(RootPtr1,RootPtr2:TPtr; var EQU:Boolean);
begin
  EQUL=true;
  while (RootPtr1<>nil) and (RootPtr2<>nil) and EQU do begin
  (* cyclys ends with the end of any diagonala *)
    PushPtr(S1,RootPtr1);
    PushPtr(S2,RootPtr2);
    RootPtr1:=RootPtr1^.LPtr; (* move along the left diagonala 1 *)
    RootPtr2:=RootPtr2^.LPtr; (* move along the left diagonala 2 *)
    EQU:=RootPtr1^.Data= RootPtr2^.Data;(* Equivalence of node data *)
  end; (* while *)
  EQU:= (RootPtr1-nil) and ( RootPtr2=nil); (* of both diagonalas end
            simultanously, then structures are still equivalent *)
end; (* LeftmostEQUTree *)
```

7. 8 Procedure creating the copy of the tree. Recursive version.

```
procedure CopyTree(RootPtrI:TPtr; var RootPtrO:TPtr);
begin
  if RootPtrI<>nil
  then begin
    new(RootPtrO);
    RootPtrO^.Data:=RootPtrI^.Data;
    CopyTree(RootPtrI^.LPtr,RootPtrO^.LPtr);  (* Left subtree *)
    CopyTree(RootPtrI^.RPtr,RootPtrO^.RPtr);  (* Right subtree *)
  end else begin
    RootPtrO:=nil
end; (* procedyre *)
```

7.9 Procedure creating the copy of the tree. Non-recursive version.

```
procedure CopyTreeNonR(RootPtrI:TPtr; var RootPtrO:TPtr);

 procedure LeftMost(Ptr1:TPtr; var Ptr2:TPtr);
 var
  TmpPtr:TPtr;   (* substitutes output parameter Ptr2, which should
                    be saed as it creates root and the head of
                    the right-son diagonala *)
 begin
  if PTr1<>nil
  then begin
   new(Ptr2);
   Ptr2^.Data:=Ptr1^.Data;  (* copying data *)
   PushPtr(S1,Ptr1);       (* original to stack *)
   Ptr1:=Ptr1^.LPtr;  (* move along diagonala  of original *)
   TmpPtr:=Ptr2;  (* Substitute of output parameter, which should be
                     saved *)

   while Ptr1<>nil do begin
    PushPtr(S2,TmpPtr);
    new(TmpPtr^.LPtr);
    TmpPtr:=TmpPtr^.LPtr;   (* move along diagonala of copy *)
    TmpPtr^.Data:=Ptr1^.Data;  (* copying data *)
    Ptr1:=Ptr1^.LPtr; (* move along the original diagonala *)
    PushPtr(S1,Ptr1);   (* push original poiter to stack *)
   end;
  end else begin
   Ptr2:=nil    (* creation of of nil Root,
                   or nil right pointer of nonexist right son
                   or left nil pointer from the leftmost node *)
  end; (* if *)
 end; (* procedure LeftMost *)
var
 S1,S2:TStackPtr;
 AuxPtrI,AuxPtrO:TPtr;  (* Auxiliary pointer *)
begin  (* body of outer procedure *)
 SInitPtr(S1);  (* Initializatin of stacks *)
 SInitPtr(S2);

 LeftMost(RootPtrI,RootPtrO);
 while not SEmptyPtr(S1) do begin
  TopPtr(S1,AuxPtrI);
  PopPtr(S1);
  TopPtr(S2,AuxPtrO);
  PopPtr(S2);
  LeftMost(AuxtPtrI^.RPtr,AuxPtrO^.RPtr); (* Call of LeftMost for
                                   right sons *)
 end; (* while *)
```

```
end; (* procedure *)
```

7.10 Creating the well weight balanced btree from the ordered array.

```
type
  TArr=array[1..Max] of integer;
```

```
procedure TreeFromArr(var RootPtr:TPtr; Right,Left:integer; Arr:TArr);
var
  Middle:integer;
begin
  if Left <= Right
  then begin
    Middle:=(Left+Right)div 2;
    new(RootPtr);
    RootPtr^.Data:=Arr[Middle];
    TreeFromArr(RootPtr^.LPtr,Left,Middle-1,Arr);
    TreeFromArr(RootPtr^.RPtr,Middle+1,Right,Arr)
  end else begin
    RootPtr:=nil
  end; (* if *)
end; (* procedure *)
```

7.11 Height of the tree. Recursive version with procedure.

```
procedure Height(RootPtr:TPtr; var Max:integer);
var
  Tmp1,Tmp2:integer;
begin
  if RootPtr<>nil
  then begin
    Height(RootPtr^.LPtr,Tmp1);  (* left son *)
    Height(RootPtr^.RPtr,Tmp2);  (* right son *
    if Tmp1>Tmp2
    then Max:=Tmp1+1
    else Max:=Tmp2+1
  end else begin
    Max:=0
  end; (* if *)
end (* procedure *)
```

7.11 a Height of the tree. Recursive version with function.

```
function HeightF(RootPtr:Tptr):integer;
  function Max(N1,N2:integer):integer;
```

```
  begin
    if N1>N1
    then Max:=N1
    else Max:=N2
  end; (* Max *)

begin (* body of outer function *)
  if RootPtr=nil
  then HeightF:=0
  else HeightF:=Max(HeighF(RootPtr^.LPtr),Heigh(RootPtr^.RPtr))+1
end; (* function *)
```

another notation of the same version:

```
function Height(RootPtr:TPtr):integer;
begin
  if RootPtr=nil
  then Height:=0
  else
    if Height(RootPtr^.LPtr) > Hight(RootPtr^.RPtr)
    then Height:= Height(RootPtr^.LPtr) + 1
    else Height:= Height(RootPtr^.RPtr) + 1
end; (* function *)
```

```
procedure TestBal(RootPtr:TPtr; var Balanced:Boolean, var
                  count:integer);

var
  LeftBal,RightBal:Boolean;    (* vyv §en¡ vlevo a vpravo *)
  LeftCount,RightCount:integer;   (* poŸet vlevo a vpravo *)

begin
  if RootPtr<>nil
  then begin
    TestBal(RootPtr^.LPtr,LeftBal,LeftCount);
    TestBal(RootPtr^.RPtr,RightBal,RightCount);
    Count:=LeftCount+RightCount+1; (* Entire count of nodes *)
    Balanced:=LeftBal and RightBal and (abs(LeftCount-RightCount)<=1);
  end else begin
    Count:=0;
    Balanced:=true
  end (* if *)
end; (* procedure *)
```

7.13 Count of the terminal nodes. Recursive function version .

```
function Count(RootPtr:TPtr):integer;
begin
  if RootPtr=nil
  then
    Count:=0
  else
    if (RootPtr^.LPtr=nil) and (RootPtr^.RPtr=nil)
    then Count:=1
    else Count:=Count(RootPtr^.LPtr) + Count(RootPtr^.RPtr);
end; (* Function *)
```

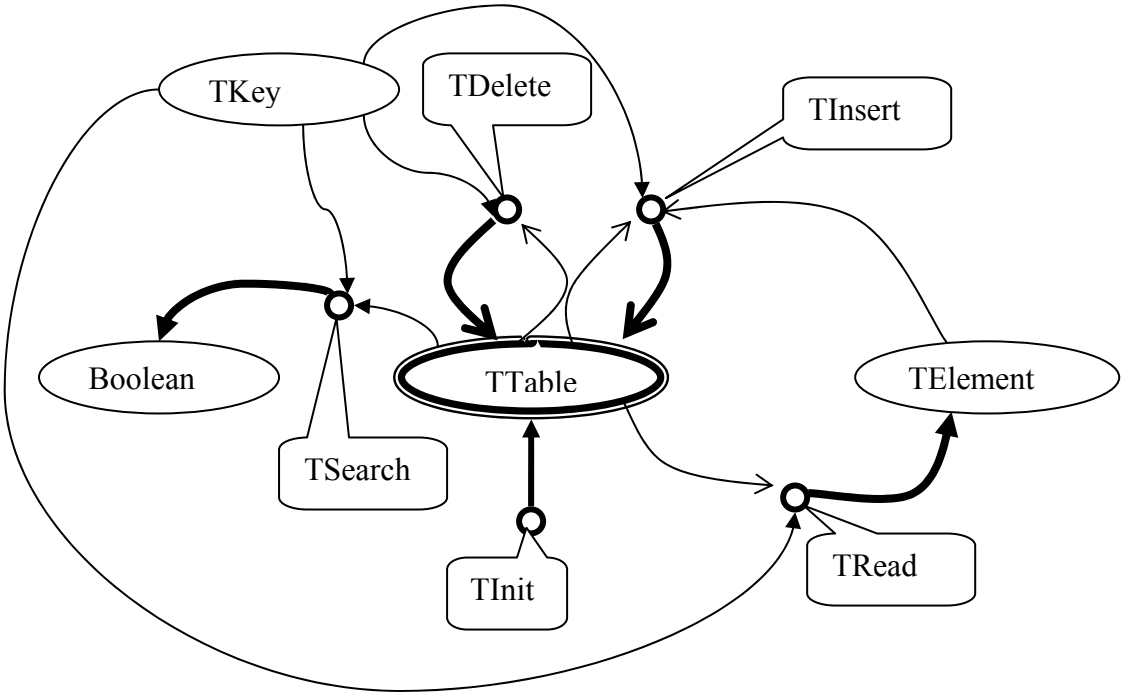
–

6. ALGORITHMS AND DATA STRUCTURES

Lesson Six:

6.1 Look-up tables

The Look-up-table (search table or simply "table") is a homogeneous, dynamic structure. Element of the table has a component called "key", which is unique for every element in the table. There doesn't exist a single one couple of elements with the same key. The key serves for unique identification of every element. Another component of an item is data. The set of operation
defined upon the table consists of:



Let T be the name of table, k is the identifier of the key and El the identifier of data.

- TInit(T) - (Initialization of the table; the result is an empty table)
- TInsert(T,K,El) (The item with the key K and data El is inserted to the table. If table contains the item with the key K, the old data component is replaced by the value El. This is called "the actualization semantics of TInsert operation").
- TDelete(T,K) (If the table contains an item with the key K, it is deleted from the table. Ifthere is not such item, nothing happens)
- TSearch(T,K) Predicate Search (function Search) returns value true, if an element with the key K exists in the table; otherwise it returns false.

- TRead(T,K,El) (If the table contains the item with the key K, operation returns the value of data in the output parameter El. An error occurs, if operation TRead tries to read the item, which doesn't exist!!!
  Any TRead operation should be conditioned by the Search test operation:

  ```
  if Search(T,K)
  then TRead(T,K,El);
  ```

The table is one of the most important data structures. It is something the database is based on. A large part of syllabus will be dedicated to various methods of implementation of tables and their operations.

6.2 Array

Array is an orthogonal, homogeneous, mostly static structure. If an array is n-dimensional, it is taken as one dimensional array, whose items are (n-1) dimensional arrays. Item of n-dimensional array is accessed by means of name (identifier) of array and n indexes. Every dimension is defined by an interval upon any ordinal type. Let us define:

TArr=array[1..10,1..20] of integer.

Let us declare the structured variable Arr:TArr. The notation Arr[5,6] means the 6th item in 5th row of matrix with 10 rows and 20 columns. The notation Arr[6] denotes the vector of 20 items,
which is the 6th row of the matrix. It is possible to write:

Arr[1]:=Arr[10] which means that the first row will be rewritten by the 10th row.

There are three most important operations defined upon the type array:

1) InitArray(ArrName, range1, range2,...,rangeN, TElem);
   where rangeI is interval of Ith dimension.

The result is allocation of the memory space for array for the base type of TElem. All items have undefined values after initialization.In Pascal the initialization is made by declaration of an array.

2) WriteArr(ArrName, ind1,ind2,...,indN, Val).

The result of an operation is setting the value of specified item to value ov Val.

In Pascal the same effect has the statement:
ArrName[ind1,ind2,...,indN]:=Val
Constructor is the special operation upon any structured type, which builds the structured value of the structure. Let us give the type TArr=array[1..2,1..3] of char and let us declare an Arr:TArr, then the result of operation like: Arr:=(Arr, 'A','B','C','D','E','F') is an array:

'A'	'B'	'C'
'D'	'E'	'F'

ANSI Pascal have no constructor for arrays, with exception of string, which has the features of one dimensional array of char, where the range of dimension starts with the value 1.

Therefore for s:string is allowed to write:
s:='ABCDEF'
which has the resulting effect of

s

'A'	'B'	'C'	'D'	'E'

3) ReadArr(ArrName, ind1,ind2,...,indN, Val).
The result of an operation is setting the value of Val to value of specified item of array. This operation has the features of "selector".Selector makes conversion from structure to single element.

In Pascal the same effect has the statement:
Val:=ArrName[ind1,ind2,...,indN];

An effort of reading the value of not-defined item results in error. (Some systems return any value and may not refer an error state).

6.2.1 Mapping function

A multidimensional array is an abstraction. Items of multidimensional array in memory are stored on adjacent positions, like vector. There is the function, transforming the n-indexes of item of multidimensional array to one index of vector.

Let there be an 2-dimensional array TArr[1..n,1..m] where n=2 is dimension of the rows and m=3 is dimension of columns. The array is represented by matrix

Arr 1 m=3

'A'	'B'	'C'
'D'	'E'	'F'

n=2

The storing of items of this array has the form of vector Vec

Vec
1 k=m*n

'A'	'B'	'C'	'D'	'E'	'F'

where the dimension of vector is k=m*n.

The mapping function Map of item Arr[i,j]=Vec[Map[i,j] has the form:
k=Map[i,j]= m * (i-1) + j
It means i-1 of full rows plus column index of item.

Let B be the L-dimensional array:

B:array[low1..high1, low2..high2,...,lowL..highL] of TElem;

The item B[j1,j2,...,jL] will be mapped into one-dimensional array A with the index:

$$1+\sum_{m=1}^{L}(j_m - low_m)*d_m \qquad (1)$$

a so

$$B[j_1, j_2,..., j_L] \rightarrow A[\ 1+\sum_{m=1}^{L}(j_m - low_m)*d_m\ ]$$

where:

$d_1 = 1$
$d_M = (high_M - low_{M+1}) * d_{M-1}$

This mapping function expresses the storing of multidimensional function "by the rows" or "with the decreasing speed of change of the indexes from the right side". (passing the matrix, the column index is increased faster then the row index ").

6.2.2 Problem for solving:

Write the mapping function for FORTRAN-like array, which is stored not by rows but by columns!

6.2.3 Dope vector (information vector, dope record, information record).

From the above mentioned paragraph it is clear, that the values ($low_m * d_m$) are independent of the value of the subscript. They may be counted before the usage of an array.

Formula (1) may be reformulated to the form:

$$1+\sum_{m=1}^{L}(j_m *d_m) - \sum_{m=1}^{L}(low_m*d_m) \qquad (2)$$

It is possible to preprocess the values of d_m and $\sum_{m=1}^{L}(low_m * d_m)$

for m=1,2,...,L. This evaluation is done once in the time of initiation. In this way the access time to element of multidimensional array may be shorter. Typical dope vector (information vector) contains:

1) Number of dimensions
2) Lower and upper limit for every dimension (L couples)
3) Total number of elements and size of one element
4) d_m for m=1,2,...,L
5) $\sum_{m=1}^{L}(low_m * d_m)$
6) Address of the first element (address of the beginning of array)

6.2.4 Triangle matrix

Triangle matrix, matrix with the different length of rows and thin (low dense) array are the methods saving the space in memory. The access time to an item of the structure will be longer as the consequence of saving the space.

Let the square matrix of shape be:

a_{11}
a_{21} a_{22}
a_{31} a_{32} a_{33}
...
a_{N1} a_{N2} a_{N3} .... a_{NN}

To spare the space, we may use only *(N div 2) * (N+1)* items instead of *N*N* items . The mapping function for this method is:

```
a[j,k] ->  b[ j*(j-1) div 2 + k];
```

It may happen that there are necessary two triangle matrixes of the same type. The array

```
c[1..N, 1..(N+1)] of TElem
```

may be used with the two different mapping functions:

a[j,k] -> c[j,k]
b[j,k] -> c[k,j+1]

6.2.5 The matrix with different length of rows (rag tables, jagged tables).

Let's have the following matrix:

a_{11}	a_{12}							
a_{21}	a_{22}							
a_{31}	a_{32}	a_{33}						
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{51}								
a_{61}								

where empty field is the not defined (not used) item of matrix. To spare the memory space we may use the vector as storing strategy:

vect

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a_{11}	a_{12}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}	a_{51}	a_{61}

We may form the mapping function in the form:

a[i,j] -> vect[AV[i]+j]

where AV is access vector, which for the introduced example would have the values

AV

1	2	3	4	5	6
0	2	4	7	8	17

AV[i] is sum of number of all defined items in all i-1 rows.

For example: a[5,4] = vect[AV[5]+4}=vect[8+4] =vect[12];

6.2.6 Thin array (Low dense array)

Let there exist an array, which contains a great majority of items of the same value. Such value is called a dominant value of an array. To spare the space in memory, it is possible to use the strategy of "thin (low dense) array". In the new table-like structure, only items with not dominant value will be stored. Items with dominant value will not be contained in the

structure, which causes the sparing of the memory space. The structure acts with the index(es) like with the key in the table.

If the Read-like access to item of the array on given index is to be done, index is searched in the table. If the search is successful, operation returns the result of TRead operation, otherwise it returns the dominant value!

If the Write-like access to the item of array on given index is to be done, then, if the written item is of not dominant value, it is written by the TInsert operation, where index serves as the key and item value as data. Otherwise the operation TDelete of item with the index=K is processed.

Thin (low dense) array is implemented by the ADT Table this way:

- InitArr(Arr) -> TInit(T)

- ReadArr(Arr,Ind,El) -> if Search(T,Ind)
  then El:=TRead(T,Ind)
  else El:= dominant value;

- WriteArr(Arr,Ind,El) -> if El=dominant value
  then TDelete(T,Ind)
  else TInsert(T,Ind,El)

6.2.7 Dynamic array

Dynamic array may be created and destroyed during the run of program. Its dimensions are defined during the initialization operation. An operation, returning the size(s) of initialized (created) array should be added to the set of operations upon the array. Any dynamic structure can be created only with the use of essential dynamic new-like operations.

Statement CreateDynArr(DArr,1..5) may be implemented by creating the list of five items. The index is defined by order of item in list. To access the item, the passing the list is necessary. More advanced techniques are based on address arithmetics and size flexible dynamic allocation of memory space.

6.3 Graph and its implementation

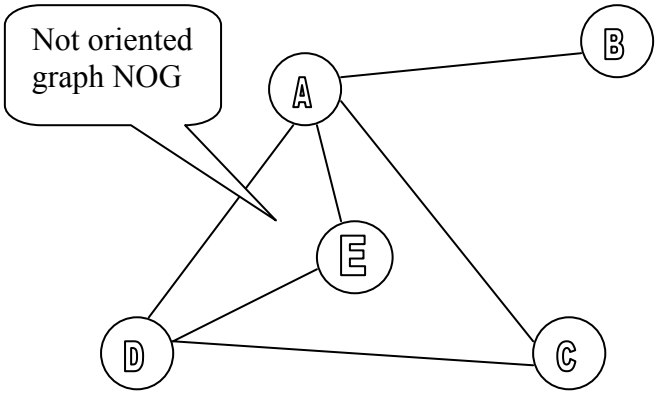
Graph may be defined by three-tuple G(N,E,I),

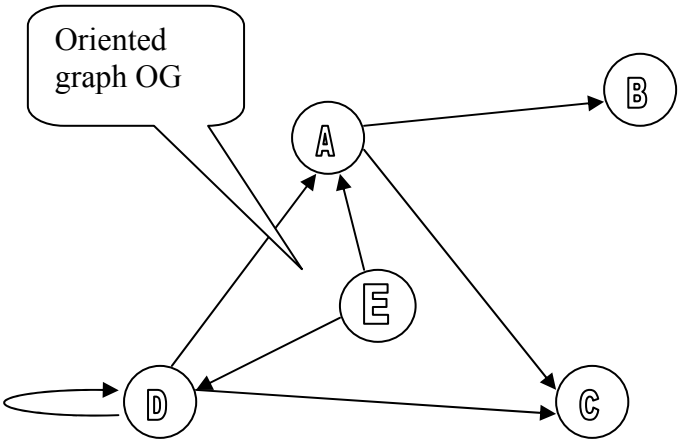
where

- N is the set of nodes, which may have assigned the value
- E is the set of edges, which may have assigned the value. Every edge connects two nodes which may not be necessarily different (edge may start and end in the same node). If the edge is oriented (directed from one node to another or the same node, the graph is called oriented graph).

- I is the set of interconnections, which defines completely the interconnections of related nodes by edges. In oriented graph the interconnection expresses orientation of edge.

Example of the graph.





Pass through all nodes of the graph and/or of the tree which visits every node only once – is an operation transforming a non-linear structure of the graph and/or tree to the linear one.

Graph may be implemented mostly by three possible ways:

6.3.1 Static implementation of graph.

Static implementation uses matrixes.

There are two methods for graph implementation by matrixes:

Interconnection matrix is a square matrix, where the size of both dimensions is determined by the number of nodes. Its items have two values (0,1 or true, false) for not oriented graphs and three values (-1, 0, 1) for oriented graphs. The value 0 on the index i, j expresses the fact that the nodes i and j are not interconnected. Another value expresses the interconnection of relevant couple of nodes i and j. We may state, that positive value 1 of item on indexes i, j expresses orientation of the edge between nodes i and j, from i to j, and negative value -1 expresses the orientation of the edge from j to i. The non-zero value on the diagonal of matrix for oriented graph has the same
significance. (The edge starts and is directed to the same node).

Interconnection matrix implementing not oriented graph NOG

	A	B	C	D	E
A	0	1	1	1	1
B	1	0	0	0	0
C	1	0	0	1	0
D	1	0	1	0	1
E	1	0	0	1	0

Interconnection matrix implementing oriented graph OG

	A	B	C	D	E
A	0	1	1	-1	-1
B	-1	0	0	0	0
C	-1	0	0	-1	0
D	-1	0	1	1	1
E	1	0	0	1	0

Neighbour matrix expresses the edges to neighbour node for every node. Each node has one row. This square matrix has often the form of "matrix with different size of rows". The item of matrix has the value denoting the node (content A means the node A).

The oriented graph OG expressed by the matrix of neighbour has the form:

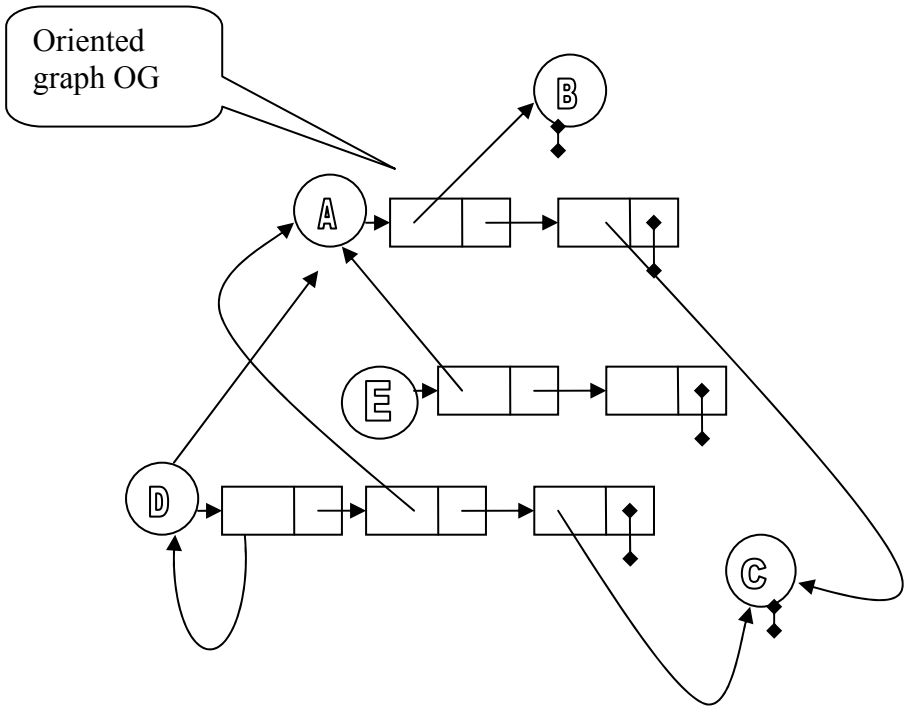
A	B	C	-	-	-
B	-	-	-	-	-
C	-	-	-	-	-
D	A	C	D	-	-
E	A	D	-	-	-

This means:

- The node A has edges directed to nodes B and C.
- The node B has no edges directed to neighbours.
- The node C has no edges directed to neighbours.
- The node D has edges directed to nodes A,C and to D itself.
- The node E has edges directed to nodes A and D.

Dynamic implementation of graph consists of the nodes represented by a list like structure. Any node list consists from the denotation (identification) of the node in the headings item, and pointers to neighbour nodes in items of the rest of list. It is in fact the dynamic representation of the matrix of neighbours.

For example the A of the above mentioned matrix has the form:



Acyclic graph is the graph with no cycle. Cycle is the path through the graph, which starts and ends in the same node.

6.4 Rooted tree and its implementation

Rooted tree is oriented acyclic graph, which has one node called root. There exists the only one path from the root to any other node. The consequence of the rule is, that for every node there exist only one edge directed into the node, and any number of nodes directed out for the node. The node is called "parent" or "father" node and edges directed out of the node point to "children" nodes. Trees may be implemented by the same methods as graphs.

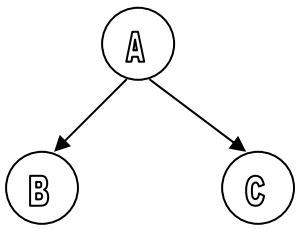
6.5 Binary tree (BT)

Recursive definition:

Binary tree is either empty or it consists of the node called root and of two sub-trees, left and right, both are binary trees.

Binary tree consists of the root, current nodes, which points at one or two children sub-trees (nodes) and terminal nodes, which have no "children". Distance of the node from the root defines the level of the node. The root is in level 1. The height of the tree is defined by the maximal distance of terminal node. Binary tree with a single node-root has the height of the value 1.

Let there be the binary tree with three nodes A,B,C, where A is the root, B is the left sub-node and C is the right sub-node:



There are three most important passes through the tree:

- Pre-order : A,B,C
- In-order : B,A,C
- Post-order: B,C,A

and there are three symmetric or inverted passes:

- Inverted pre-order : A,C,B
- Inverted In-order : C,A,B
- Inverted Post-order: C,B,A

Notice that the pre-order and inverted post-order are symmetric. Similarly the post-order and inverted pre-order are symmetric, while the in-order is symmetric with its inverted version.

Let there be types necessary to implement the binary tree (the similar that these for double-linked lists).

```
type
  TPtr=^TNode;  (* pointer to node *)
  TNode=record  (* type of node *)
         Data:Tdata;
         LPtr,RPtr:TPtr;
        end;
```

The recursive procedure which creates a list containing the pre-order sequence of all nodes of binary tree defined by the pointer to its root has the form:

```
procedure Preoder(RootPtr:TPtr; var L:TList);
(* suppose the InitList(L) was done before the call of this
   procedure *)
begin
  if RootPtr is <> nil
  then begin
    DInsertLast(L,RootPtr^.Data);
    Preorder(L,RootPtr^.LPtr);
    Preorder(L,RootPtr^.RPtr)
  end;
end; (* procedure *)
```

By changing the order of three statements in the if statement other passes are created:

```
(* Inorder *)
Inorder(L,RootPtr^.LPtr);
DInsertlast(L,RootPtr^.Data);
Inorder(L,RootPtr^.RPtr);

(* Postorder *)
Postorder(L,RootPtr^.LPtr);
Postorder(L,RootPtr^.RPtr);
DInsertlast(L,RootPtr^.Data).
```

ALGORITHMS AND DATA STRUCTURES
Lesson seven

7 Exercise: Implementation of 13 tree-etudes and some of their variants.

Declaration of the necessary types

```
type
(* Double linked list DList *)
  TDItemPtr=^TDItem;
  TDItem = record
           Data:integer;
           LItPtr,RItPtr:TItemPtr;
  end;

  TDlist=record
         Frst,Lst,Act:TDItPtr;
        end;

  TPtr=TDItemPtr; (* Type for tree item is equivalent to  type
                     for double list item *)

(* Single linked list SList *)

  TSItemPtr=^TSItem;
  TSItem = record
           Data:integer;
           RItPtr:TSItemPtr;
  end;

  TSlist=record
         Frst,Act:TSItPtr;
        end;


 TStackPtr=   (* type stack of Pointers to tree (TPtr) *)
 TStackBool = (* type stack of boolean values *)
```

7.1 Non-recursive Pre-Order. The result is inserted into ADT list by means of abstract operations.

```
procedure PreOrder(RootPtr:TPtr; var DL:TDList);
procedure LeftMost(RootPtr:TPtr);
(* procedure passes along the main left diagonal, pushing the node pointers to the stack *)
begin
   while RootPtr<>nil do begin
     PushPtr(S,RootPtr);
     DInsertLast(DL,RootPtr^.Data);   (* Output to DL *)
     RootPtr:=RootPtr^.LPtr
  end;
end;

var
  TmpPtr:TPtr;
  DL:TDlist;
  S:TStackPtr;
begin
   DInitList(L);  (* Initiation of the list *)
   SInit(S);     (* Initiation of the stack *)
   LeftMost(RootPtr);  (* Initial call of LeftMost *)

   while not SEmpty(S) do begin (* passing cycle *)
     Top(S,TmpPtr); Pop(S);
     LeftMost(TmpPtr^.RPtr)
   end;
end; (* procedure *)
```

7.2 Non-recursive In-Order. The result inserted into ADT single linked list by means of abstract operations (with the use of list-heading).

```
procedure InOrder(RootPtr:TPtr; var SL:TSList);

procedure LeftMost(RootPtr:TPtr;var DL:TDList);
(* procedure passes along the main left diagonal, pushing the node
pointers to the stack *)
begin
  while RootPtr<>nil do begin
    PushPtr(S,RootPtr);
    RootPtr:=RootPtr^.LPtr
 end;
end;

var
 TmpPtr:TPtr;
 S:TStackPtr;
begin
  InitList(SL);  (* Initilization of the list *)
  SInsertFirst(SL,0); (* Creation of headings with the Data=0*)
  First(SL);
  SInit(S);     (* Initilization of the stack *)
  LeftMost(RootPtr,DL);  (* Initial call of LeftMost *)

  while not SEmpty(S) do begin (* passing cycle *)
    Top(S,TmpPtr); Pop(S);
    SPostInsert(SL,TmpPtr^.Data); (* Output to SL *)
    SSucc(SL);
    LeftMost(TmpPtr^.RPtr,DL)
  end;
  SDeleteFirst(SL); (* Erase heading *)
end; (* procedure *)
```

7.2 a Variant of Pre-Order with one cycle (Double linked list.)

```
procedure  Preorder(RootPtr:TPtr;var DL:TDlist);
var
  S:TStackPtr;
begin
   SInit(S);
   PushPtr(S,RootPtr);
   DInitList(DL);
   repeat              (* passing cycle *)
     Top(S,RootPtr); Pop(S);
     if RootPtr<>nil
     then begin
       DInsertLast(DL, RootPtr^.Data); (* output to DList *)
       PushPtr(S,RootPtr^.RPtr);
       PushPtr(S,RootPtr^.LPtr);
     end;
  until SEmpty(S);
end;
```

7.3 Non-recursive Post-Order with double linked output list .

```
procedure PostOrder(RootPtr:TPtr; var DL:TDList);

  procedure LeftMostPost(RootPtr:TPtr);
  (* Auxiliary procedure passing along the left diagonal and storing  in stacks node pointers
and true values *)
  begin
    while RootPtr<>nil do begin
      PushPtr(S,RootPtr); (* Stack with pointers *)
      PushBool(SB,true);   (* Stack with Boolean *)
      RootPtr:=RootPtr^.LPtr
    end; (* while *)
  end; (* LeftmostPost *)

var
  FromLeft:Boolean;  (* Aux. Bool. variable *)
  TmpPtr:TPtr;
  S:TStackPtr;  (* stack for pointers *
  SB:TStackBool;  (* stack for Booleans *)
begin
  DListInit(DL);
  SInitPtr(S);
  SInitBool(SB);
  LeftMostPost(RootPtr);




  while not SEmptyPtr(S) do begin
    TopBool(SB,Fromleft); PopBool(SB);
    TopPtr(S,TmpPtr);
    if FromLeft
    then begin
      PushBool(SB,false);
      LeftMostPost(TmpPtr^.RPtr);
    end else begin
      DInsertLast(DL,TmpPtr^.Data);
      PopPtr(S)
    end; (* if *)
  end (* while *)
end; (* procedure PostInsert *);
```

7.3 a Variant of Post-Order with use of two pointer stack and inversed Pre-Order, Output to DList;

```
procedure PostOrder2(RootPtr:TPtr; var DL:TDlist);
var
  S1,S2:TStackPtr;
  TmpPtr:TPtr;
  Val:integer; (* TData *)

  procedure RightMost(RootPtr:TPtr);
  begin
    while RootPtr<>nil do begin
      PushPtr(S1,RootPtr);  (* first stack for backtracking *)
      PushPtr(S2,RootPtr);  (* Second stack for temp. output *)
      RootPtr:=RootPtr^.RPtr;
    end (* while )
  end; (* procedure RightMost *)

begin
   SInitPtr(S1);
   SInitPtr(S2);
   RightMost(RootPtr);
   while not SEmpty(S1) do begin
     TopPtr(S1,TmpPtr);
     RighMost(TmpPtr^.LPtr)
   end; (* while *)

(*   Reversing the result from Stack 2 *)

   while not SEmptyPtr(S2) do begin
     TopPtr(S2, Val); PopPtr(S2);
     InsertLast(DL,Val);
   end (* while *)
end; (* procedure PostOrder *)
```

7. 4 General procedure for all three passes. Output to DL.

```
type
  TPass=(PreO,InO,PostO);
```

```
procedure GeneralPass(RootPtr:TPtr; Pass=TPass; var DL:TDList);

  procedure LeftMostPost(RootPtr:TPtr);
  (* Auxiliary procedure passing along the left diagonala and storing
     in stacks node pointers and true values *)
  begin
    while RootPtr<>nil do begin
      PushPtr(S,RootPtr); (* Stack with pointers *)
      PushBool(SB,true);   (* Stack with Boolean *)
      if Pass=PreO
      then DInsertLast(DL,RootPtr^.Data); (* output for PreOrder *)
      RootPtr:=RootPtr^.LPtr
    end; (* while *)
  end; (* LeftmostPost *)
var
  FromLeft:Boolean;  (* Aux. bool. variable *)
  TmpPtr:TPtr;
  S:TStackPtr;  (* stack for pointers *
  SB:TStackBool;  (* stack for booleans *)
begin
  DListInit(DL);
  SInitPtr(S);
  SInitBool(SB);
  LeftMostPost(RootPtr);
  while not SEmptyPtr(S) do begin
    TopBool(SB,Fromleft);PopBool;
    TopPtr(S,TmpPtr);
    if FromLeft
    then begin
      if Pass=InOrder
      then DInsertLast(DL,TmpPtr^.Data); (* Output for InOrder *)
      PushBool(SB,false);
      LeftMostPost(TmpPtr^.RPtr);
    end else begin
      if Pass=PostO
      then DInsertLast(DL,TmpPtr^.Data);  (* Output for PostOrder *)
      PopPtr
    end; (* if *)
  end (* while *)
end; (* procedure PostInsert *);
```

7.5 Equivalence of the structure of two trees. Recursive version.

```
  (* in Comment brackets extension for equivalence of two trees - data
     are involved *)

function EQTS(RootPtr1,RootPtr2:TPtr):Boolean;
begin
 if (RootPtr1=nil) or (RootPtr2=nil)
 then
   EQTS:=(RooTPtr1=nil) and (RootPtr2=nil)
 else
   EQTS:=EQTS(RootPtr1^.LPtr,RootPtr2^.LPtr) and
         EQTS(RootPtr1^.RPtr,RootPtr2^.RPtr)
         (* and (RootPtr1^.Data=RootPtr2^.Data);  - for equality of
          the two trees *);
end;
```

7. 5 a Variant with encapsulated procedure.

```
function EQUTS(RootPtr1,RootPtr2:TPtr):Boolean;
  procedure AuxEQ(RootPtr1,RootPtr2:TPtr; var EQU:Boolean);
  (* As EQU should be set to true before AuxEqu is called, this
     procedure is "wrapped" by outer procedure EQUTS *)

  begin
    if ((RootPtr1=nil) and (RootPtr2=nil) or
       (RootPtr1<>nil) and (RootPtr2<>nil) and EQU
    then begin
      if RootPtr1<>nil
      then begin
        AuxEQ((RootPtr1^.LPtr,RootPtr2^.LPtr,EQU);
        AuxEQ((RootPtr1^.RPtr,RootPtr2^.RPtr,EQU);
      end else begin
        EQU:=false
      end (* if *);
   end; (* EuxEQ *)
var
  EQU:Boolean;
begin
  EQU:=true;    (* Initialization of EQU *)
  AuxEQ(RootPtr1,RootPtr2,EQU);   (* call of main procedure *)
  EQUTS:=EQU;                     (* forming output *)
end; (* EQUTS *)
```

7.6 Equivalence of the structure of two trees. Non-recursive version.

```
function EquTreeStrNonRec(RootPtr1,RotPtr2:TPtr):Boolean;
  procedure LeftMostEQUS(RootPtr1,RootPtr2:TPtr; var EQU:Boolean);
  begin
    while (RootPtr1<>nil) and (RootPtr2<>nil) do begin
    (* cyclus ends with the end of any diagonala *)
      PushPtr(S1,RootPtr1);
      PushPtr(S2,RootPtr2);
      RootPtr1:=RootPtr1^.LPtr; (* move along the left diagonala 1 *)
      RootPtr2:=RootPtr2^.LPtr; (* move along the left diagonala 2 *)
    end; (* while *)
    EQU:= (RootPtr1=nil) and ( RootPtr2=nil); (* if both diagonalas end
              simultanously, then structures are still equivalent *)
  end; (* LeftmostEQUS *)

var
   EQU:Boolean;

begin (* body of main function *)
  SInitPtr(S1);  (* Stacks initialization *)
  SInitPtr(S2);

  LeftMostEQUS(RootPtr1,RootPtr2,EQU);
  while not SEmptyPtr(S1) and EQU do begin
    TopPtr(S1,RootPtr1);
    TopPtr(S2,RootPtr2);
    LeftMostEQUS(RootPtr1^.RPtr,RootPtr2^.RPtr,EQU);
  end; (* cycle while finishes with the end of pass or with EQU=false*)
  EquTreeStrNonRec:=EQU
end;
```

7.7 Equivalence of the two trees. Non-recursive version.

The main procedure is the same as the 6). Different is the Leftmost:

```
procedure LeftMostEQUTree(RootPtr1,RootPtr2:TPtr; var EQU:Boolean);
begin
  EQUL=true;
  while (RootPtr1<>nil) and (RootPtr2<>nil) and EQU do begin
  (* cyclys ends with the end of any diagonala *)
    PushPtr(S1,RootPtr1);
    PushPtr(S2,RootPtr2);
    RootPtr1:=RootPtr1^.LPtr; (* move along the left diagonala 1 *)
    RootPtr2:=RootPtr2^.LPtr; (* move along the left diagonala 2 *)
    EQU:=RootPtr1^.Data= RootPtr2^.Data;(* Equivalence of node data *)
  end; (* while *)
  EQU:= (RootPtr1-nil) and ( RootPtr2=nil); (* of both diagonalas end
            simultanously, then structures are still equivalent *)
end; (* LeftmostEQUTree *)
```

7. 8 Procedure creating the copy of the tree. Recursive version.

```
procedure CopyTree(RootPtrI:TPtr; var RootPtrO:TPtr);
begin
  if RootPtrI<>nil
  then begin
    new(RootPtrO);
    RootPtrO^.Data:=RootPtrI^.Data;
    CopyTree(RootPtrI^.LPtr,RootPtrO^.LPtr);  (* Left subtree *)
    CopyTree(RootPtrI^.RPtr,RootPtrO^.RPtr);  (* Right subtree *)
  end else begin
    RootPtrO:=nil
end; (* procedyre *)
```

7.9 Procedure creating the copy of the tree. Non-recursive version.

```
procedure CopyTreeNonR(RootPtrI:TPtr; var RootPtrO:TPtr);

 procedure LeftMost(Ptr1:TPtr; var Ptr2:TPtr);
 var
  TmpPtr:TPtr;   (* substitutes output parameter Ptr2, which should
                    be saed as it creates root and the head of
                    the right-son diagonala *)
 begin
  if PTr1<>nil
  then begin
   new(Ptr2);
   Ptr2^.Data:=Ptr1^.Data;  (* copying data *)
   PushPtr(S1,Ptr1);       (* original to stack *)
   Ptr1:=Ptr1^.LPtr;  (* move along diagonala  of original *)
   TmpPtr:=Ptr2;  (* Substitute of output parameter, which should be
                     saved *)

   while Ptr1<>nil do begin
    PushPtr(S2,TmpPtr);
    new(TmpPtr^.LPtr);
    TmpPtr:=TmpPtr^.LPtr;   (* move along diagonala of copy *)
    TmpPtr^.Data:=Ptr1^.Data;  (* copying data *)
    Ptr1:=Ptr1^.LPtr; (* move along the original diagonala *)
    PushPtr(S1,Ptr1);   (* push original poiter to stack *)
   end;
  end else begin
   Ptr2:=nil    (* creation of of nil Root,
                   or nil right pointer of nonexist right son
                   or left nil pointer from the leftmost node *)
  end; (* if *)
 end; (* procedure LeftMost *)
var
 S1,S2:TStackPtr;
 AuxPtrI,AuxPtrO:TPtr;  (* Auxiliary pointer *)
begin  (* body of outer procedure *)
 SInitPtr(S1);  (* Initializatin of stacks *)
 SInitPtr(S2);

 LeftMost(RootPtrI,RootPtrO);
 while not SEmptyPtr(S1) do begin
  TopPtr(S1,AuxPtrI);
  PopPtr(S1);
  TopPtr(S2,AuxPtrO);
  PopPtr(S2);
  LeftMost(AuxtPtrI^.RPtr,AuxPtrO^.RPtr); (* Call of LeftMost for
                                   right sons *)
 end; (* while *)
```

```
end; (* procedure *)
```

7.10 Creating the well weight balanced btree from the ordered array.

```
type
  TArr=array[1..Max] of integer;
```

```
procedure TreeFromArr(var RootPtr:TPtr; Right,Left:integer; Arr:TArr);
var
  Middle:integer;
begin
  if Left <= Right
  then begin
    Middle:=(Left+Right)div 2;
    new(RootPtr);
    RootPtr^.Data:=Arr[Middle];
    TreeFromArr(RootPtr^.LPtr,Left,Middle-1,Arr);
    TreeFromArr(RootPtr^.RPtr,Middle+1,Right,Arr)
  end else begin
    RootPtr:=nil
  end; (* if *)
end; (* procedure *)
```

7.11 Height of the tree. Recursive version with procedure.

```
procedure Height(RootPtr:TPtr; var Max:integer);
var
  Tmp1,Tmp2:integer;
begin
  if RootPtr<>nil
  then begin
    Height(RootPtr^.LPtr,Tmp1);  (* left son *)
    Height(RootPtr^.RPtr,Tmp2);  (* right son *
    if Tmp1>Tmp2
    then Max:=Tmp1+1
    else Max:=Tmp2+1
  end else begin
    Max:=0
  end; (* if *)
end (* procedure *)
```

7.11 a Height of the tree. Recursive version with function.

```
function HeightF(RootPtr:Tptr):integer;
  function Max(N1,N2:integer):integer;
```

```
  begin
    if N1>N1
    then Max:=N1
    else Max:=N2
  end; (* Max *)

begin (* body of outer function *)
  if RootPtr=nil
  then HeightF:=0
  else HeightF:=Max(HeighF(RootPtr^.LPtr),Heigh(RootPtr^.RPtr))+1
end; (* function *)
```

another notation of the same version:

```
function Height(RootPtr:TPtr):integer;
begin
  if RootPtr=nil
  then Height:=0
  else
    if Height(RootPtr^.LPtr) > Hight(RootPtr^.RPtr)
    then Height:= Height(RootPtr^.LPtr) + 1
    else Height:= Height(RootPtr^.RPtr) + 1
end; (* function *)
```

```
procedure TestBal(RootPtr:TPtr; var Balanced:Boolean, var
                  count:integer);

var
  LeftBal,RightBal:Boolean;    (* vyv §en¡ vlevo a vpravo *)
  LeftCount,RightCount:integer;   (* poŸet vlevo a vpravo *)

begin
  if RootPtr<>nil
  then begin
    TestBal(RootPtr^.LPtr,LeftBal,LeftCount);
    TestBal(RootPtr^.RPtr,RightBal,RightCount);
    Count:=LeftCount+RightCount+1; (* Entire count of nodes *)
    Balanced:=LeftBal and RightBal and (abs(LeftCount-RightCount)<=1);
  end else begin
    Count:=0;
    Balanced:=true
  end (* if *)
end; (* procedure *)
```

7.13 Count of the terminal nodes. Recursive function version .

```
function Count(RootPtr:TPtr):integer;
begin
  if RootPtr=nil
  then
    Count:=0
  else
    if (RootPtr^.LPtr=nil) and (RootPtr^.RPtr=nil)
    then Count:=1
    else Count:=Count(RootPtr^.LPtr) + Count(RootPtr^.RPtr);
end; (* Function *)
```

–

ALGORITHMS AND DATA STRUCTURES

8 Search Table I

8.1 Access time.

Access time is the time that has passed from the request for item with the searched key to the time when the item is usable or when the predicate "search" is set to true or false. There are several access times:

minimal search time
maximal search time (the worst case time)
average search time (the sum of search times of all keys divided by their number)

The access time for successful and not successful search may differ.

8.2 Basic structure of the search algorithm.

Search algorithm, which searches the item with given key in the set of items has the form of cycle:

```
found:=false;
while <not found> and <set of items is not exhausted> do
  <examine the next item and set found to true if key found >
Search:=found;
```

8.3 Classification of search algorithms

According to the access to individual items of the set of items the algorithms can be divided to:

1. sequential algorithms - the next item may be examined only after its predecessor
2. algorithms with the random access - all items are accessible immediately

According to processing of the set of items (and according to the type of the structure implementing the set of items) algorithms can be classified as:

1 Sequential searching in not ordered sequential structure
2.Sequential searching in array with the sentinel (guard)
3.Sequential searching in ordered sequential structure (Sequential structures may be:array, list, file)
4.Sequential searching in ordered array with the sentinel (guard)
5.Sequential searching in structure ordered according to the probability of being searched. (Searching in probability adaptive array)
6.Binary search in the ordered array. (Normal binary search, Dijkstra binary search for multiple keys)
7. Fibonacci search
8. Binary search trees (BST) (BST with the back pointers, AVL trees)
9. Hash-tables

..

8.4 Sequential searching in not ordered sequential structure

Let us take the type

```
type
  TPtr=^TItem; (* for table implemented by the list *)
  TItem=record
        Key:TKey;  (* operation equal is defined upon the TKey *)
        Data:TData;
        Next:TPtr;  (* only if item is a member of list *)
       end;
   TArray=array[1..Max] of TItem;
   TFile=file of TItem;
   TList=TPtr;
```

Search cycle for the array:

```
  function Search(Table:TArray; K:TKey):Boolean;
(* algorithm returns true/false in the name of the function *)
  var
   i:integer;
   Found:Boolean;
  begin
   i:=1;
   Found:=false;
   while not Found and (i<=Max) do begin
     Found:=Table[i].Key=K;
     i:=i+1;
   end;
   Search:=Found;
  end;
```

Note: the cycle

```
while (Table[i].Key<>K and (i<=Max)  do...
```

is not correct.

If i>Max the Table[i] is out of range!!!

The cycle:

```
while (i<=Max) and (Table[i].Key<>K do...
```

is correct even for "short-cut evaluation of Boolean expression". We avoid this style unless we have some special reason for it!

Search cycle for the file has the form:

```
function Search(var F:TFile; K:TKey):Boolean;
var
  Found:Boolean;
  Tmp:TItem;
begin
  Found:=false;
  reset(F);
  while not Found and not eof(F) do begin
    Read(F,Tmp);
    Found:=Tmp.Key=K
  end;
  Search:=Found
end;
```

Search cycle for the list has the form:

```
function Search(T:Tptr; K:TKey):Boolean;
var
  Found:Boolean;
begin
  Found:=false;
  while not Found and (T<>nil) do begin
    Found:=T^.Key=K;
    T:=T^.Next
  end;
  Search:=Found;
end;
```

Similarly the cycle:

```
while (T^.Key <> K) and (T<>nil) do T:=T^.Next;
```

is not correct as for T=nil the T^.Key is not defined;

Reversed Boolean expression is possible only with short-cut evaluation of Boolean expression:

```
while (T<>nil) and (T^.Key <> K) do T:=T^.Next;
```

We avoid this style unless we have some special reason for it!

==========

8.4.1 Exercise:

Write the search cycle for search table implemented by an array or by a list, so that the output parameter "where" gives an access (index or pointer) to the searched item. If the search is false the value of Where is not defined.

```
procedure Search(T:TArray; Key:TKey; var Found:Boolean,
            var Where:integer); { assuming the type of data is integer }
(* procedure heading for table implemented by an array; Value of Where
will not be defined for Found=false *)

procedure Search(T:TPtr; Key:TKey; var Found:Boolean; var Where:TPtr);
(* procedure heading for table  implemented by a list. Value of Where will not be defined for
Found=false  *)
```

variants:

```
procedure Search(T:TArray; Key:TKey; var Where:integer);
(* if Where=0 then Search = false
            else index Where points to the found item *)

procedure Search(T:TPtr; Key:TKey; var Where:TPtr);
(*  if Where = nil then Search=false
              else pointer Where points to the found item *)
```

==========

8.4.2 Table implemented by not ordered array

There is N items in the array (N<=Max)

Search is mentioned above;

Operation Insert

```
procedure Insert(var T:TArr; var N:Integer; K:TKey; El:TData; var Full:Boolean);
(* N is the number of items contained on the table. N<=Max *)
var
  Found:Boolean;
  Where:integer;
begin
  Full:=false;
  Search(T,N,K,Found,Where);
  (* N is the number of items in the array *)
  if Found
  then T[Where].Data:=El
  else begin
    if N=Max
    then Full:=true
    else begin
```

```
      N:=N+1;
      T[N].Key:=K;
      T[N].Data:=El
    end
  end;
end;
```

Operation Delete

```
procedure Delete(var T:TArr; var N:integer; K:Tkey);
var
   Found:Boolean;
   Where:integer;
begin
  Search(T,N,K,Found,Where);
  if Found
  then begin
    T[Where] := T[N];
    N:=N-1
  end
end;
```

==========

8.4.3 Search with the sentinel (guard) or "quick linear search"

```
procedure Search(T:TArr;N:integer;K:TKey;var Found:Boolean;var
                Where:integer);
var
begin
  T[N+1].Key:=K;   (* setting the sentinel (or guard) *)
  Where:=1;
  while T[Where].Key<>K do Where:=Where+1;
  Found:=Where<>(N+1)
end;
```

Note: The Key is stored at the position N+1. This position is called "a sentinel" or "a guard". The cycle finished with success in search, but the real Found is true only if the cycle ends on the index different from N+1! The algorithm is faster than previous as it doesn't evaluate two-part Boolean expression every pass.

==========

8.4.4 Sequential searching in ordered sequential structure
(Sequential structures may be: array,list, file)

The sequence of items in the sequential structure may be ordered by the value of the key. It is possible if there is relation operation defined upon the type TKey!

The only advantage of the ordered sequence is faster not successful Search !!!

```
procedure Search (T:TArr;N:integer;K:TKey;var Found:Boolean; var Where:integer);
var
 i:integer;
 Stop:Boolean; (* Auxiliaryn variable *)
begin
 Where:=1;
 Stop:=false;
 while not Stop and (Where<=N) do begin
  Stop:=K<=T[Where].Key; (* stop if processed item has the key equal
                        or less the searched key !!! *)
 end;
 Found:=T[Where].Key=K;
 (* if Found=False, then Where points to the place to which the new
   item may  be placed! *)

  (* for K=7, N=6,
  T =  1,3,5,8,10,12
              ^
         Where
     Found=false,  Where=4,  as the item 8 has index 4*)
end;
```

==========

Exercise:

1. Write the similar procedure for the list implemented table.
2. Write the similar procedure for the array implemented table with the sentinel (guard).

8.4.5 Operation Insert in ordered array

Insertion of new item should preserve ordering of all items. The free space for the new item should be made by shift of the massive of items to the right.

```
procedure Insert (var T:TArr;var N:integer;K:TKey;El:TData;
              var  Full:Boolean);
var
 Found:Boolean;
 Where,i:integer;
```

```
begin
  Full:=false;
  Search(T,N,K,Found,Where);
  if Found
  then begin
    T[Where].Data:=El;  (* Actualization of Data *)
  else begin
    if N=Max
    then Full:=true
    else begin
      for i:=N-1 downto Where do T[i+1]:=T[i];  (* Shift of the massive of
                                        items - making free space
                                        for new item *)
      T[Where].Key:=K;       (* Insertion of the Key *)
      T[Where].Data:=El;     (* Insertion of the Data *)
      N:=N+1
  end;
end;
```

```
procedure Delete(var T:TArr; var N:integer; K:TKey);
var
  Found:Boolean;
  Where,i:integer;
begin
  Search(T,N,K,Found,Where);
  if Found
  then begin
    for i:=Where to N-1 do begin T[i]:=T[i+1]
    (* The cycle shifts the massive of items one possiton to the left
       and thus rewrites the deleted item *)
    N:=N-1;
  end;
end;
```

==========

8.4.5 Sequential searching in structure ordered according to the probability of being searched. (Serching in probability adaptive array)

If the probability of searching all the keys is not equal, the table may be ordered according to the probability of being searched.

```
procedure Search(var T:TArr; N:integer; K:TKey; var Found:Boolean;
              var Where:integer);
begin
  Where:=1;
  Found:=false;
```

```
  while not Found and Where<=N do begin
    If T[Where].Key=K
    then Found:=true
    else Where:=Where+1;
  end;
  if Found and (Where<>1)
  then T[Where]:=:T[Where-1]  (* operation  ":=:" is our shorthand
                         notation for swapping of the two elements,
                         i.e. :   A:=:B is the same like
                         P:=A; A:=B; B:=P   *)

end;
```

Note: If Search finds the item, algorithm exchange (swap) it with its predecessor. This results in accumulation of often searched items in the left part of an array and moving the rarely searched items in the right part of an array. The algorithm, in the self-adaptive manner rebuilds the table according to the probability of keys being searched.

==========

=

ALGORITHMS AND DATA STRUCTURES
Lesson Nine

9 Search Tables II

9.1 Binary search in ordered array.

Binary search in an ordered array is a non-sequential (non-linear) search process.

Let the array implementing the table consists of items for which is valid:

A[1].Key < A[2].Key< ... < A[n].Key

and let for the searched key K be valid:

A[1].Key <= K <= A[n].Key

then, the principle of searching can be described in the following words:

Searched key K is compared with the item in the middle of an array A, limited by left and right index. Index of the middle item is defined by index = (left+right) div 2. If key fits to the middle item, search finishes successfully. If not, the key K is compared with the middle item and if the searched key K is less than the middle item, the next searching area is defined by left index and (middle-1) as the right boundary. Symmetrically, if the searched key is greater than the middle
item, the next searching area is defined by (middle+1) as the left boundary and right index. The search is finished as not-successful, if the index of the right boundary is greater than index of the left boundary. The algorithm has the form:

```
left:=1;  (* left boundary *)
right:=n;  (* right boundary *)
repeat
  middle:=(left+right) div 2;
  if K < A[middle].Key
  then right:=middle - 1    (* searched item is on the left half *)
  else left:= middle +1;    (* searched item is on the right half *)
until (K=A[middle].Key) or (right < left);

Search:= K=A[middle].Key;
```

(* if Search then item A[Middle] is searched item and i is access index to it *)

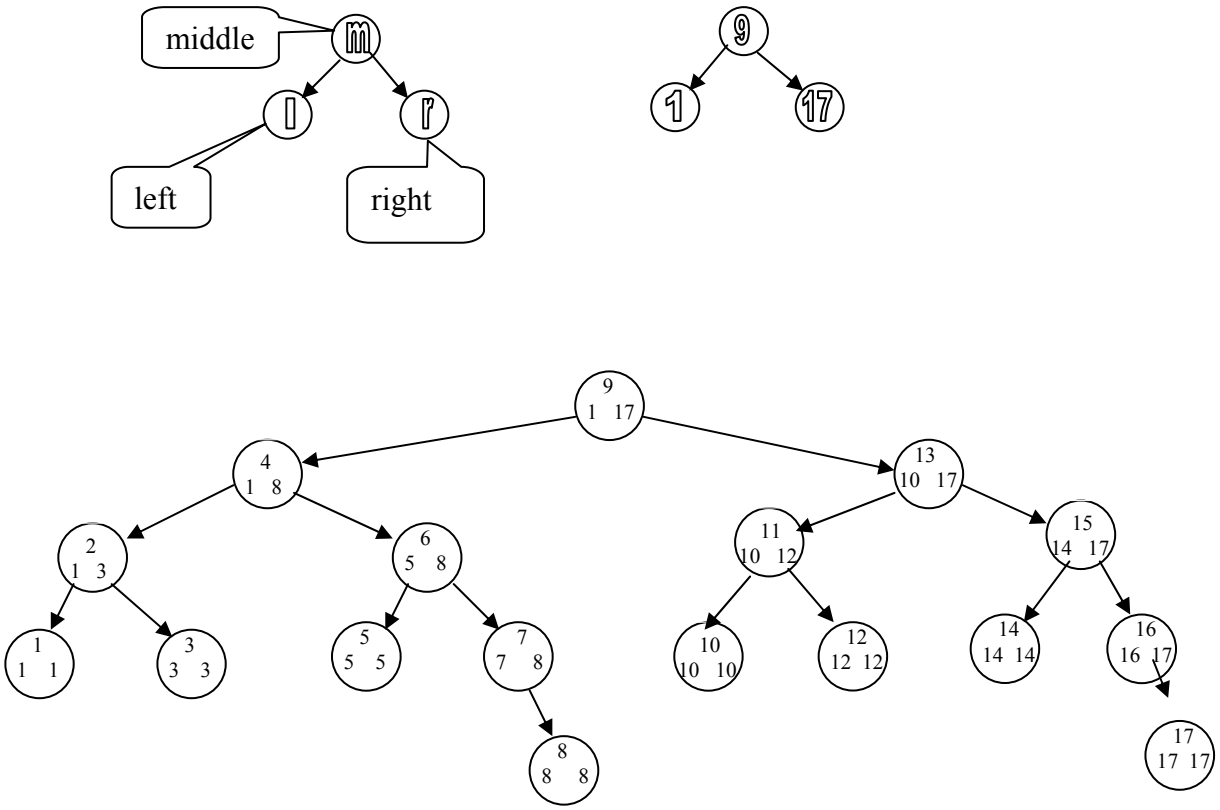
9.1.1 Analysis of the time complexity of the algorithm:

Successful search is determined by the position of the item. The worst case is the lg(n) steps, where lg(n) is logarithm with the base (2) of n.

Unsuccessful search has the same value as the worst case of successful search - lg(n). This time-complexity is called logarithmic and it is much more efficient than linear search complexity of sequential algorithms.

Decision tree representation of the binary search for array [1..17]
Decision tree node: middle=(left+right)div2.

The termination test is based on equality of left and right.



9.2 Dijkstra's variant of binary search.

Dijkstra's variant is a variant for the special case of searching, where multiple equal items are allowed. If the key is searched, and there are more items with the same value of the key, the search algorithm finds the left (or right) most item.

Example expresses the variant for searching the rightmost item.

Let is: A[1].Key <= A[2].Key <= ... <= A[n-1].key < A[n].Key

(Notice, that all items are "les or equal", only the leftmost couple has the relation "less" (without equal).

For searched key is valid:
A[1].Key <= K < A[n].Key.

(Notice the non symmetrical relation of the key to boundary items).

Algorithm of Dijkstra's variant, searching the rightmost from the series of equal keys has the form:

```
left:=1;
right:=n;
while right <> (left+1) do begin
  middle:=(left+right) div 2;
  if A[middle].Key <= K
  then left:=middle
  else right:=middle
end;

Search:= K=A[middle].Key;
```

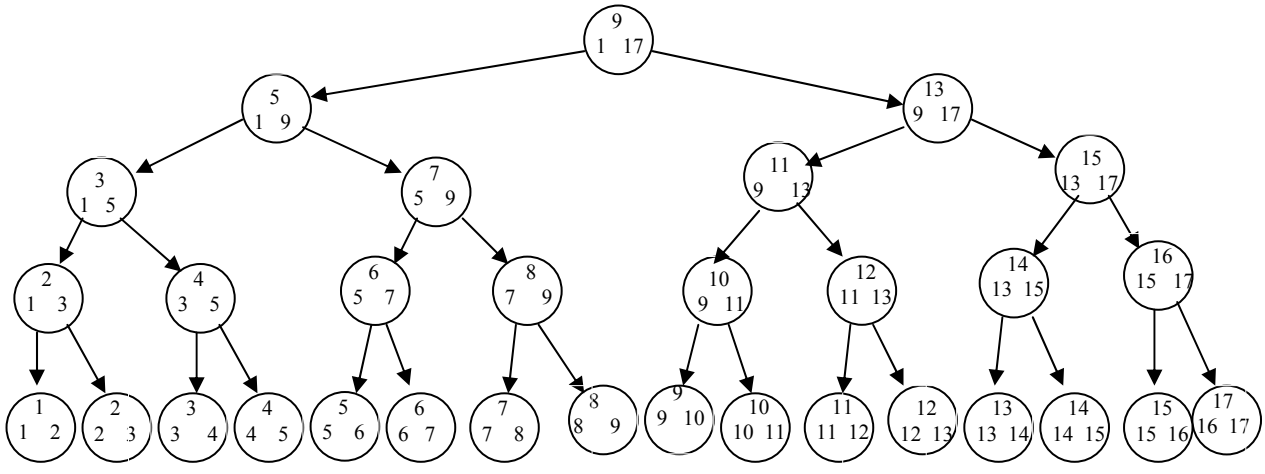
(* if Search then item A[middle] is the rightmost item from the series of items with the same value of keys. *)

Example:

In the array: 1,2,3,4,5,5,6,6,6,8,9,13 the algorithm finds the key K=6 on the 8th position :
^

In the array: 1,1,1,1,1,1,1,1,1,1,2 the algorithm finds the K=1 on the10th position.

Decision tree representation of Dijkstra's variant for an array [1..17].



9.2.1 Analysis of the time complexity of the algorithm:

Both successful and unsuccessful search of Dijkstra's variant has the same complexity given by formula lg(n).

9.3 Fibonacci Search

Fibonacci sequence is given by recursive formula:

A[0]=0;
A[1]=1;
for i>1 A[i]= A[i-1]+A[i-2]

for example:

i	0	1	2	3	4	5	6	7	8	9	10	11
F(i)	0	1	1	2	3	5	8	13	21	34	55	89

Fibonacci sequence is similar in course features to binomial sequence.

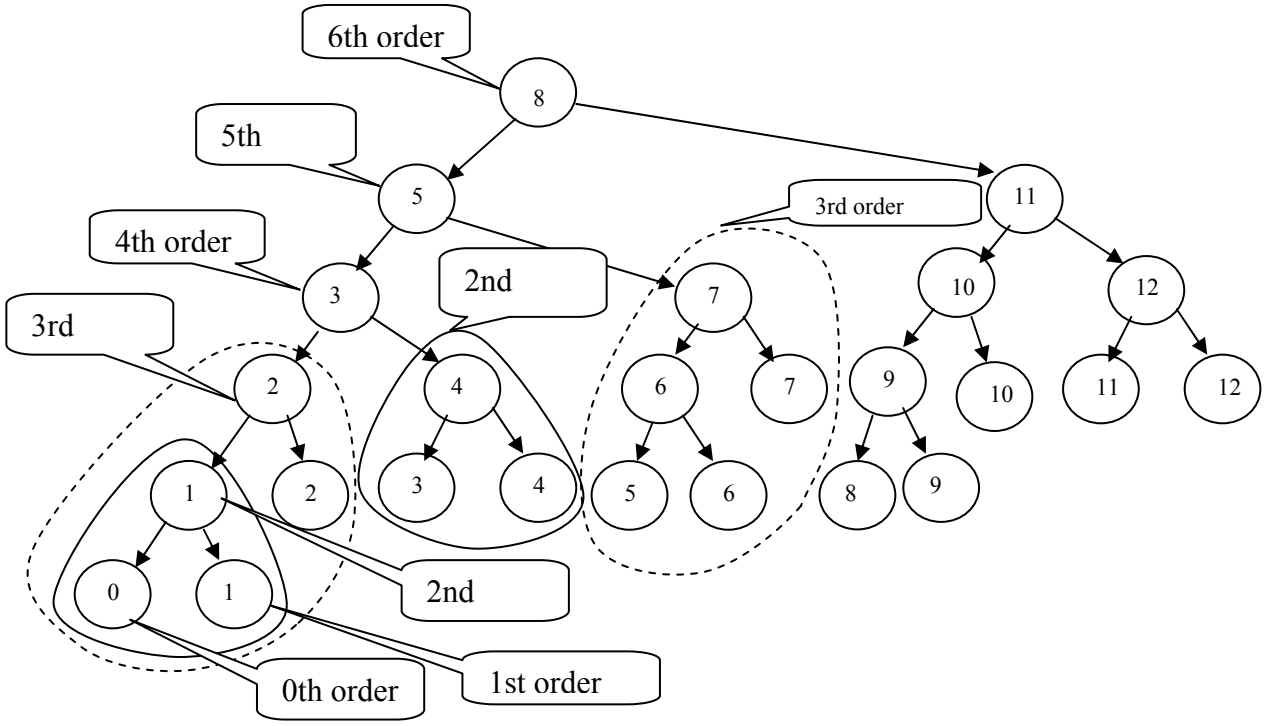
Fibonacci tree (F-tree) is defined by the rules:

1) F-tree of lst order consists from F(l-1) non-terminal nodes and from F(l+1) terminal nodes.

2) if l=0 or l=1 the tree is represented only by the root and simultaneously terminal node [0].

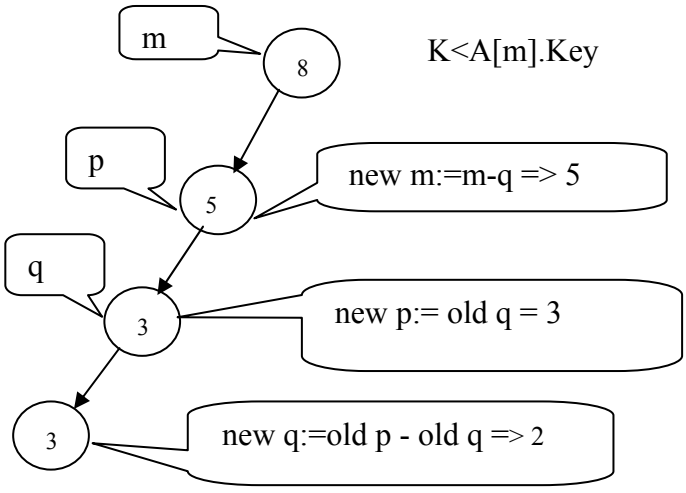
3) if l>=2 then the root of the tree is represented by the root F(l) and its left sub-tree is the F-tree of (l-1) order and its right sub-tree is the F-tree of (l-2) order, the nodes of which are incremented by the value F(l).

F-tree for the array [0..12].

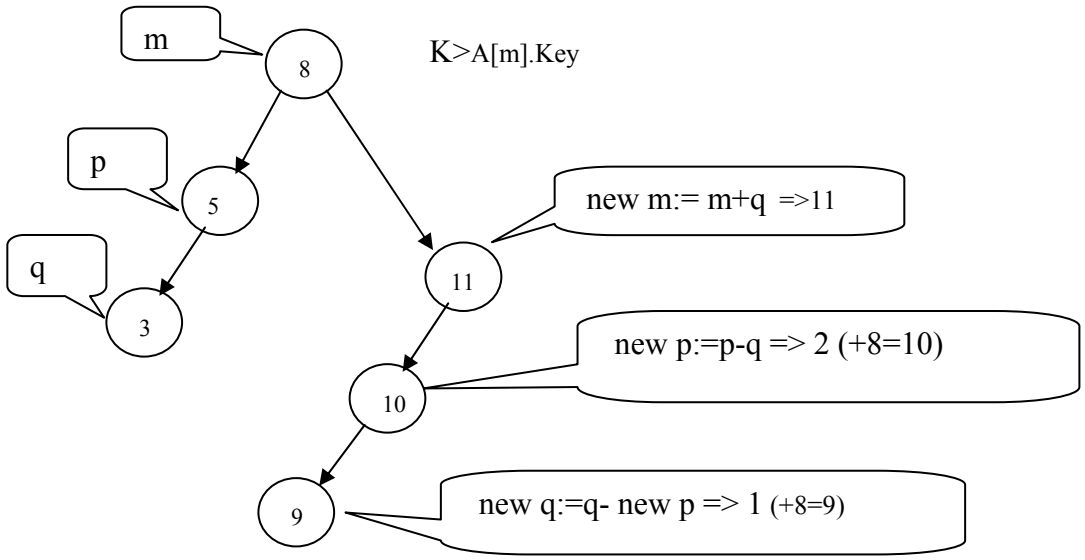


The principle, which is in binary search derived from binary tree is in Fibonacci search similar, but derived from F-tree.

Moving to the left part of an array (tree):



Moving to the right part of an array (tree)



Search algorithm has the form:

```
m:=F(l);
p:=F(l-1);
q:=F(l-2);

TERM:=false;
while (K<>A[m].Key) and not TERM do begin
 if K<A[m].Key
 then (* search continues in the left sub-tree *)
   if q=0
   then TERN:=true (* search ends on the left zero-like terminal of
                      zero order*)
   else begin  (* moving left son along the diagonal *)
     m:=m-q;
     p1:=q;
     q1:=p-q;
     p:=p1;
     q:=q1;
  else (* search continues in the right sub-tree *)
   if p=1
   then TERM:=true (* search ends on the right 1-like terminal of
                      first order *)
   else begin (* setting of new values of m, p and q in the right
                 sub-tree *)
     m:=m+q;
     p:=p-q;
     q:=q-p
```

```
   end; (* if
 end; (* while *)
 Search:= not TERM;
```

The most important feature (and advantage) of Fibonacci search is avoiding multiplicative operations - here dividing by two. There are some cases of non-binary arithmetic systems (Binary-coded-decimal), where multiplication and division are very time-consuming operations. Fibonacci search makes dividing of the interval to two sub-intervals (nearly like halving) only by additive operations!

9.3.1 Insert and delete operations

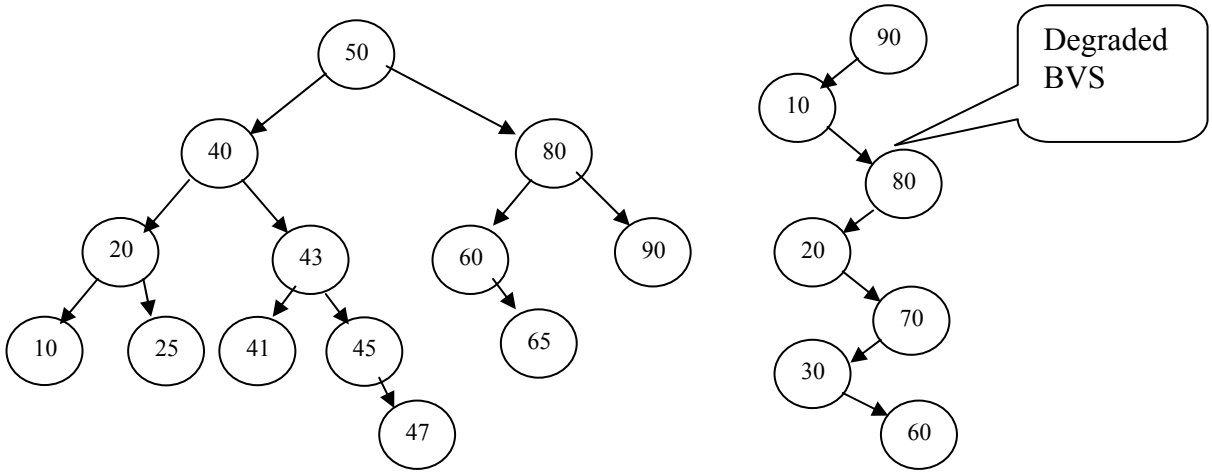
As the base structure of a search table is an ordered array, the insert operation is the same as in sequential (linear) search in ordered array. The massive of items should be moved by one index to the left and new item is inserted to freed space and the counter of items is increased by one.

The delete operation is done by move of massive of items by one index to left and decreasing of the counter of items.

9.4 Binary search trees

Binary search tree (BST) is "ordered binary tree" for all nodes of which there is the rule saying that keys of nodes of left sub-tree are less than the key in current node and all keys in the right sub-tree are greater than the key in current node.

Example of BSTs.



==========
The "Inorder pass" through the BST gives the sequence ordered by the key value!!!

The "inorder" pass through the left above shown BST gives the sequence: "10, 20, 25, 40, 41, 43, 45, 47, 50, 60, 65, 80, 90".

==========

Let are the types:

```
TPtr=^TNode;
TNode=record
       Key:TKey; (*  often integer or char or string *)
       Data:TData;
       LPtr,RPtr:TPtr
      end;
```

9.4.1 Recursive search as function:

```
function Search(RootPtr:TPtr; K:TKey):Boolean;
(* RootPtr is root pointer of the BST; Search is true if the Key part of the node equals the K
*)
begin
  if RootPtr<>nil
  then
    if RootPtr^.Key=K
    then
      Search:=true
    else
      if RootPtr^.Key>K
      then (* search continues in the left subtree *)
        Search:=Search(RootPtr^.LPtr,K)
      else (* search continues in the right subtree *)
         Search:=Search(RootPtr^.RPtr,K)
  else (* path finishes in the leaf - not successful search *)
    Search:=false
end;
```

Variant of the search as procedure:

```
procedure SearchTree(var RootPtr:TPtr; K:TKey);
(* Proceure returns the pointer to found node in the var parameter RootPtr or nil if not found
*)
begin
 if RootPtr<>nil
 then
```

```
   if RootPtr^.Key <> k

   then
       if RootPtr^.Key > K
       then begin
           SearchTree(RootPtr^.LPtr,K)
       else SearchTree(RootPtr^.RPtr,K)

end; (* procedure *)
```

9.4.2 Nonrecursive search as function:

```
function Search(RootPtr:TPtr; K:TKey):Boolean;
var Fin:Boolean (* Cycle control variable *)
begin
  Search:=false;
  Fin:=RootPtr=nil;
  while not Fin do begin
    if RootPtr^.Key=K
    then begin
      Fin:=true;
      Search:=true;
    end else begin
      if RootPtr^.Key > K
      then RootPtr:=RootPtr^.LPtr  (* Continue in the left subtree *)
    else
      RootPtr:=RootPtr^.RPtr; (* Continue in the right subtree *)
      if RootPtr=nil
      then Fin:=true
 end; (* while *)
end; (* function *)
```

Exercise:

Create non-recursive procedure which :

a) returns the Boolean Search parameter and pointer WherePtr of found node (not defined for Search=false)

```
  procedure InsertSearch1(RootPtr:TPtr; K:TKey; var Search:Boolean;
                    var WherePtr:TPtr);
```

b) returns the Where pointer to found node or nil for not found.

```
  procedure InsertSearch2(RootPtr:TPtr; K:TKey; var WherePtr:TPtr);
```

9.4.3 Recursive insert operation in BST.

New item is inserted as leaf (terminal node). The insert operation is very fast. The BST is very efficient because of its dynamism.

```
procedure Insert (var RootPtr:TPtr; K:TKye; El:TData);
begin
  if RootPtr=nil
  then begin (* create te new node *)
     new(RootPtr);
     with RootPtr^ do begin
       Key:=K;
       LPtr:=nil;
       RPtr:=nil;
       Data:=El
     end
  end else begin
     if K<RootPtr^.Key
     then Insert(RootPtr^.LPtr,K,El) (* continue in the left subtree *)
     else
        if K>RootPtr^.Key
        then Insert(RootPtr^.RPtr,K,El) (* continue in the right  subtree *)
        else RootPtr^.Data:=El  (* rewrite the old data by new ones *)
end; (* procedure *)
```

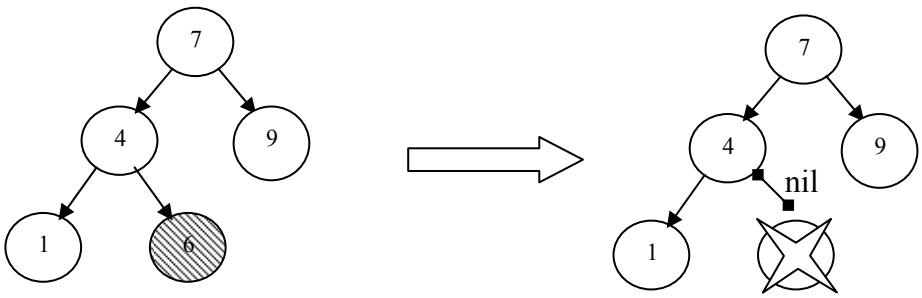
9.4.4 Non-recursive Insert operation

```
procedure Insert (var RootPtr:TPtr; T:TKey; El:TData);
var
  TmpPtr, WherePtr:TPtr;
  Found:Boolean;
  procedure InsertSearch1(RootPtr:TPtr;k:TKey;var Found:Boolean; var where:TPtr);
  var Ptr:TPtr;
  begin
     if RootPtr=nil
     then begin
       found:=false;
       where:=nil;
     end else
       repeat
          where:=RootPtr;
          if RootPtr^.key > k
          then
             RootPtr:=RootPtr^.LPtr
          else
             if RootPtr^.key < key
             then
               RootPtr:=RootPtr^.RPtr
             else
                found:=true
```

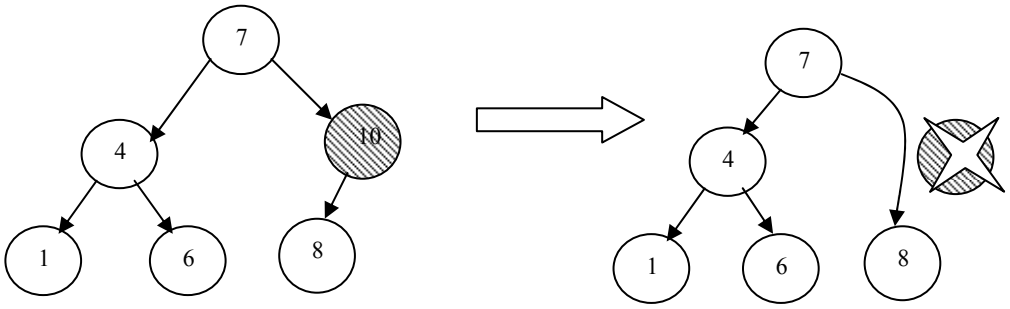
```
    until found or (RootPtr=nil)
  end; (* proceure *)

begin
  InsertSearch1(RootPtr,K,Found,WherePtr);
  (* procedure was created as the above mentioned exercise *)
  if Found
  then
    WherePtr^.Data:=El; (* rewriting the old data by new one *)
  else begin
    new(TmpPtr); (* creation the new terminal node *)
    with TmpPtr^ do begin
      Data:=El;
      Key:=K;
      LPtr:=nil;
      RPtr:=nil;
    end; (* with *)
    if WherePtr=nil
    then RootPtr:=TmpPtr; (* Tree was empty; new node becomes the root *)
    else (* tree is not empty, new node is linked to node *)
      if WherePtr^.Key>K
      then (* node is linked to the left *)
        WherePtr^.LPtr:=TmpPtr
      else (* node is linked to the right *)
        WherePtr^.Ptr:=TmpPtr
  end (* if Found *)
end; (* procedure *)
```

9.4.5 Deleting the node in BST.

Deleting the node in the BST is more difficult than inserting.
a) It is not difficult to delete the terminal node:
To delete the terminal node f the pointer from b to f is set to nil.



b) It is not difficult to delete the node with only one sub-tree:



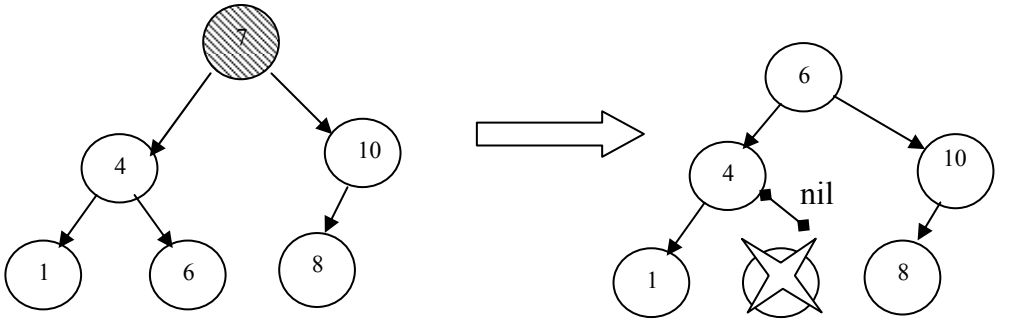
Subtree linked to the deleted node is linked to the father of the deleted node.

c) To delete the node with two sub-trees is done according to this technique:
Find the rightmost node of the left sub-tree right diagonal. The "rightmost" node has only one sub-tree. Rewrite the data of node which is to be deleted by the data of found rightmost node and delete the rightmost node (what is to be done according to the rule a) or b).

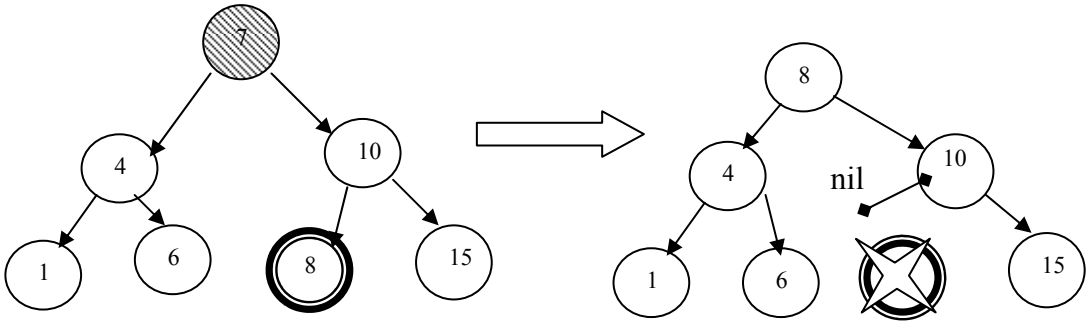
Symmetric technique finds the leftmost node at left diagonal in the right sub-tree. Rewrites the node to be deleted by its data and delete the leftmost node.

Example:

To delete node (7), find the rightmost node on the right diagonal of left sub-tree. It is node (6). Rewrite the (7) by (6). The modified tree fulfils the rules for BST.



Symmetric solution finds the leftmost node on the left diagonal of the right sub-tree. It is (8). Rewrite the (7) by (8). The modified tree fulfils the rules for BST.



The recursive notation of procedure Insert is not simple. It uses nested recursive procedure:

```
procedure Delete(var RootPtr:TPtr; K:TKey);

(* If procedure finds the node with the key K, t rewrites the data of the found node. In other
cases it follows the above mentioned rules *)

var
  TmpPtr:TPtr;
procedure Del(var Ptr:TPtr);
(* Auxiliary procedure moves along the rightmost branch (diagonal) of the left sub-tree of
deleted node TmpPtr and look for the rightmost node Ptr. When it finds the node, the node
TmpPtr is rewritten by the data of node Ptr  and Ptr is made free for later disposing *)

begin
  if Ptr^.RPtr <>nil
  then
     Del(Ptr^.RPtr) (* continue in the right subtree *)
  else begin (* rightmost node found; rewriting and releasing the node *)
     TmpPtr^.Data:=Ptr^.Data;
     TmpPtr^.Key:=Ptr^.Key;
     TmpPtr:=Ptr;
     Ptr:=Ptr^.LPtr (* releasing the Ptr node ! Careful! Ptr is in the procedure Del the pointer of
node superior to node Ptr^ *)
  end
end; (* of auxiliary procedure Del *)

begin (* start of main procedure Delete *)
  if RootPtr <> nil
  then (* search not finished; searched node still may be in the BST *)
    if K < RootPtr^.Key
    then
      Delete (RootPtr^.LPtr,K) (* continue in the left sub-tree *)
    else
```

```
    if K > RootPtr
    then Delete(RootPtr^.RPtr,K)  (* continue in the right sub-tree *)
    else begin (* node RootPtr is to be deleted *)
      TmpPtr:=RootPtr;
      if TmpPtr^.RPtr=nil
      then (* node has no right subtree; rule b) is followed *)
        RootPtr:=TmpPtr^.LPtr
      else (* node has right subtree; it will be rewritten by the
              rightmost node in the left sub-tree by the procedure Del *)
        if TmpPtr^.LPtr=nil
        then
          RootPtr:=TmpPtr^.RPtr (* attaching the right sub-tree *)
        else
          Del(TmpPtr^.LPtr);
      dispose (TmpPtr); (* releasing the node *)
    end
  else (* here may be message or reaction on the fact that the key was  not found; normally no
action takes place *)
end (* delete *).
```

9.4.6 Nonrecursive operation delete

Non-recursive procedure for deleting is perhaps more transparent but much more complex. Here is its short design:

```
procedure Delete (var RootPtr:TPtr; K:TKey);
begin
  DeleteSearch(RootPtr,K.Found,FatherLeft,GrandFatherPtr,FatherPtr);
  (* Procedure searches K in the BST RootPtr. If RootPtr = nil, then       Found=false and
Father and GrandFather are not defined; If found  node is the Root, then GrandFather=nil,
father=Root, Found=true and FatherLeft is not defined. If the found node is inner node of  the
tree, so Found=true, GrandFather is node superior to Father,  FatherLeft=true if Father is left
son of the Grandfather. If the  key is not found, Found=false and other parameters are not
defined *);

  if Found
  then begin
    if FatherPtr^.RPtr=nil
    then begin (* deleted node has no right sub-tree *)
      if GrandFatherPtr=nil
      then begin
        RootPre:=FatherPtr^.LPtr (* deleted node is the root ! *)
      end else begin
        (* block = make link from grandfather to left son *)
      end
    end else begin
      if FatherPtr^.LPtr=nil
      then begin (* deleted node has no left sub-tree *)
        if GrandFatherPtr=nil
```

```
        then begin
          RootPtr:=FatherPtr^.RPtr (* deleted node is the root ! *)
        end else begin
          (* block = make link from grandfather to right son *)
        end (* if *)
       end else begin
        RightMost (FatherPtr);
         (* Procedure finds the rightmost node of the left sub-tree,  rewrites the data of father
by found node, release the rightmost node and returns it in parameter for later disposing *)
        dispose (FatherPtr);
    end
   end
end; (* procedure *)
```

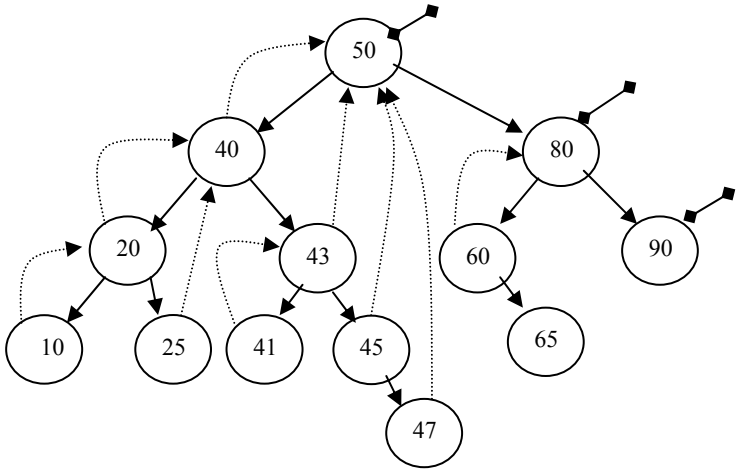
We need to write detailed procedures: Delete, DeleteSearch and RighMost.

```
procedure Delete(var RootPtr:TPtr; K:TKey);
var
  FatherPtr,GradndFatherPtr:TPtr;
  procedure DeleteSearch(RootPtr:TPtr;K:TKey;
                    var Found,FatherLeft:Boolean;
                    var FatherPtr,GrandFatherPtr:TPtr);
  (* the body of procedure will be introduced later *)

  procedure RightMost(var FatherPtr:TPtr);
  (* the body of procedure will be introduced later *)
begin
  DeleteSearch(RootPtr,K,Found,FatherLeft,FatherPtr,GradnFatherPtr);
  if Found
  then begin
    if FatherPtr^.RPtr=nil
    then
      if GrandFatherPtr=nil
      then RootPtr :=FatherPtr^.LPtr    (* Deleted node is root with the only left son;  left son
becomes root  *)
      else
       if Fatherleft (* attach left son to gradfather *)
       then GrandFatherPtr^.LPtr:=FatherPtr^.LPtr
       else GrandFatherPtr^.RPtr:=FatherPtr^.LPtr
    else
      if FatherPtr^.LPtr=nil
      then
        if GrandFather=nil
        then RootPtr:=FatherPtr^.RPtr
        else
          if FatherLeft (* attach right son to grandfather *)
          then GrandFatherPtr^.LPtr:=FatherPtr^.RPtr
          else GrandFatherPtr^.RPtr:=FatherPtr^.RPtr
      else Rightmost(Father);
    dispose(FatherPtr)
```

```
  end (* if Found *)
end; (* procedure *)


procedure DeleteSearch(RootPtr:TPtr;K:TKey; var
                    Found,Fatherleft:Boolean;
                    vat FatherPtr,GrandFatherPtr:TPtr);
var
  Fin:Boolean (* cycle control variable *)
begin
  Found:=true;
  FatherPtr:=RootPtr;
  if RootPtr<>nil
  then
    if RootPtr^.Key=K
    then begin
      Found:=true; (* found is the root *)
      GrandFatherPtr=nil;
    end else begin
      Fin:=false;
      while not Fin do begin
        if FatherPtr^.Key=K
        then begin
          Found=true;
          Fin:=true;
        end else begin
          GrandFatherPtr:=FatherPtr;
          if Father^.Key > K
          then begin
            FatherLeft:=true;
            FatherPtr:=FatherPtr^.LPtr
          end else begin
            FatherLeft:=false;
            FatherPtr:=FatherPtr^.RPtr
          end;
        end; (*  if *)
        if FatherPtr=nil
        then Fin:=true
      end (* while *)
    end; (* if *)
end (* procedure *)

procedure RightMost(var FatherPtr:TPtr);
var
  RMost:FatherRMost:TPtr;
begin
  RMost:=FatherPtr^.LPtr;
  if RMost^.RPtr<>nil
  then begin (* search for rightmost node *)
    repeat
```

```
    FatherRMost:=RMost;
    RMost:=RMost^.RPtr
  until RMost^.RPtr=nil;
  FatherRMost^.RPtr:=RMost^.LPtr
 end else begin (* RMost is rightmost itself *)
  FatherPtr^.LPtr:=RMost^.LPtr;
 end;

 FatherPtr^.Key:=RMost^.Key; (* rewritting the key *)
 FatherPtr^.Data:=RMost^.Data; (* rewritting the data *)
 FatherPtr:=RMost (* Father, in fact te RMost  will be disposed *)
end;
```

9.4.7. BST with the back-pointers (Monkey puzzle tree)



BST with the back-pointer is used if there is good reason not to use recursion or stack. In other words, back-pointer is used to enable implementation of in-order pass without using stack or recursion..

Every node has the back pointer (the dashed arrow in the above mentioned figure). The rules for back-pointer are as follows:
a) The back-pointer of the root is nil.
b) The back-pointer of the left son points to its father.
c) The back-pointer of the right son points to the same place as father (right son inherits the pointer of his father).
Note: The consequence of the a) and b) is that back-pointers of the nodes at right diagonal from the starting with the root are all nil.

The inorder pass has the form:

```
type
 TPtr=TPtrItem;
 TItem=record
```

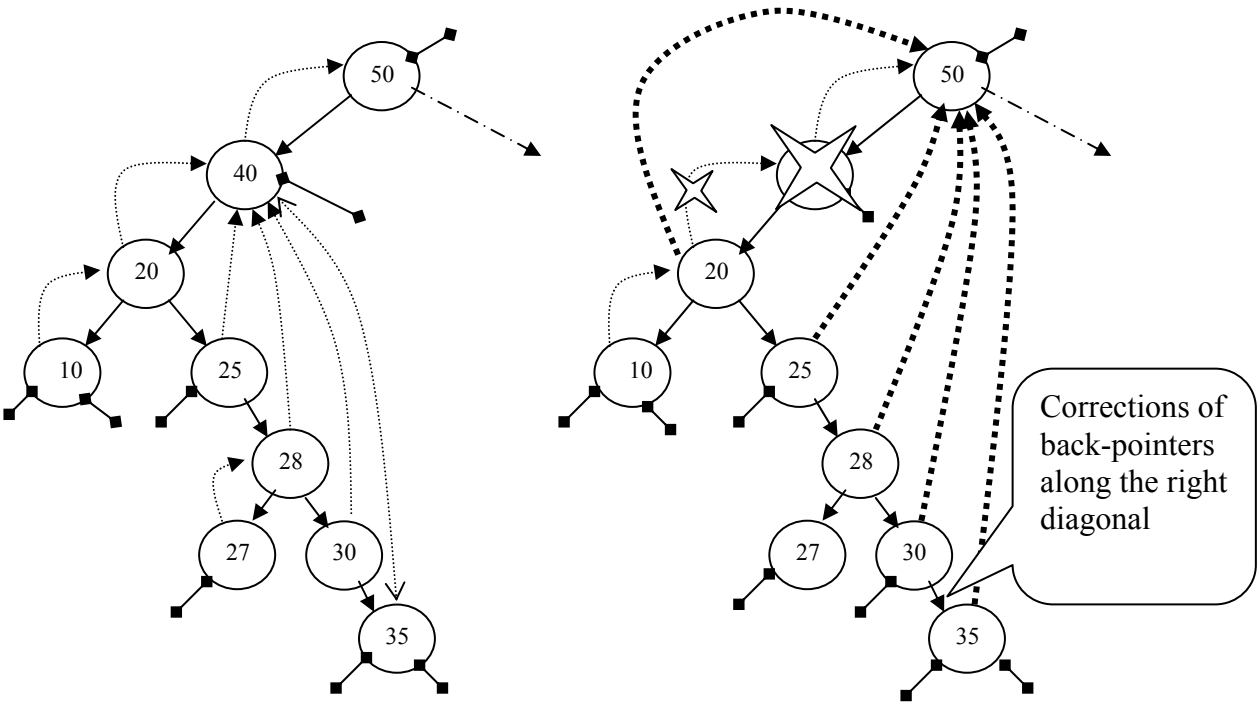
```
    Key:TKey;
    Data:TData;
    LPtr,RPtr,BackPtr:TPtr
  end;

procedure LeftMost(RootPtr:TPtr; var LeftMostPtr:TPtr);
(* procedure returns the pointer of the leftmost node of RootPtr in the parameter LeftMostPtr
*)
begin
  LeftMostPtr:=RootPtr;
  while RootPtr <> nil do begin
    LeftMostPtr:=RootPtr;
    RootPtr:=RootPtr^.LPtr
  end;

procedure Inorder(RootPtr:TPtr; var DL:TDlist);
(* procedure inserts processed nodes to output double linked list DL in "inorder" sequence
using operation "DInsertLast"*)
var
  Fin:Boolean;
  LeftMostPtr:TPtr;
begin
  InitDList(DL); (* Initiation of the list *)
  LeftMost(RootPtr,LeftMostPtr);
  Fin:=LeftMostPtr<>nil;    (* Control variable of the cycle finalizing the cycle *)

  while not Fin do begin
    DInsertLast(DL, LeftMostPtr^.Data); (* Inserting the processed node into the output list *)
    if LeftMostPtr^.RPtr<>nil
    then LeftMost(LeftMostPtr^.RPtr, LeftMostPtr)
    else
      if LeftMostPtr^.BackPtr=nil
      then Fin:=true
      else LeftMostPtr:=LeftMostPtr^.BackPtr  (* moving backward *)
  end;
end; (* procedure *)
```

When inserting the new node, the back-pointer is set:

- to point to the father node
- to point to the same node as its father (the back-pointer of father is copied to the back-pointer or the back-pointer is inherited from the father).

Deletion of the node needs some additional, if deleted node is ***the left son of its father and it has only one sub-tree***. The necessary correction concerns to all nodes of the right diagonal of deleted node.

Exercise: Draw a similar situation where deleted node is *the right son of its father having only one sub-tree* and prove that there is no corrective action needed.


50
40
20
10
25
28
27
30
35
50
20
10
25
28
27
30
35
Corrections of back-pointers along the right diagonal

ALGORITHMS AND DATA STRUCTURES
Lesson Ten

Implementation of ADT "Searching tables" III.

10 HASHING TABLES

10.1 Tables with the straight access.

Let us say there exists the set of keys K, used for the searching table and the set of adjacent addresses A, implementing the search table so, that cardinality of set K is less than cardinality of set A: card(K)<card(A).

If there can be found the mapping function H(K->A) so, that any key from the set K is mapped to unique address A, we say that there exists the straight implementation of the search table.

Let us declare the array Arr implementing the set of address A. Each key from the set K gives the index to array Arr - with use of mapping function H. All items of array contain the component called "busy", which indicates if the element is empty or not. This indicator is set to "not-busy" at the initialization.

Any search is implemented by the statement, where H(K) is function which maps the Key into appropriate index :

```
Search:= Arr[H(K)].busy.
```

which returns "true" if the item with the key K is present in the table and it returns "false" if not.

Such kind of searching is called ***indexed-search***, as there is the function which determines in the straight way the index to array from the given key. Access to searched item is done by the function.

Generally it is difficult to find the function, which has the necessary properties.

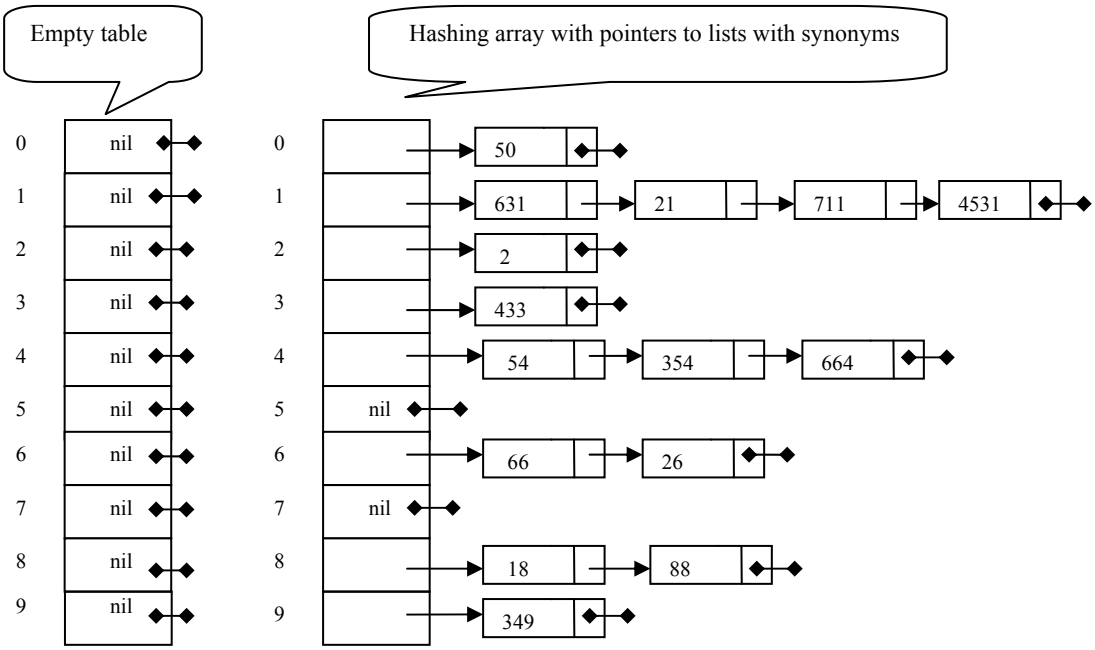
10.2 Hashing tables

Hashing tables works with a mechanism which combines the principles of index (straight) and sequential way of searching to "index-sequential". While in true straight search method one unique address exist for any unique key and one unique key exists for any unique address, in hashing tables the unique property is valid only in one direction, which means that one address exists for any unique key, but it may happen that several different keys are mapped to the same address! The subset of keys, mapped by the given mapping function to the same address is called the subset of synonyms. The mapping function is called the "hashing" function. Synonyms make collisions (conflict) in getting the same address.

The principle of search in hashing tables is composed from the following two phases: In the first phase there is the "index" access to the starting point of the sequence of synonyms, in the second phase the "sequential" search in the sequence of synonyms is done.

Let there be the hashing array Arr contained from 10 items indexed from 0 to 9 and let the key K be any integer greater than 0. Let the hashing function be defined as H= (K mod 10). In such way the index (address) of key K=354 is (354 mod 10) = 4. Thus key 354 is mapped into the index 4. Address 4 contains pointer which is a starting point of all synonyms (all keys with the same digit at the lowest position).

Let the hashing array consist of 10 pointers to lists. Each list contains items with the synonymous keys (for instance keys 54, 3674, 694, 4, 124 are synonymous).



The items of an array are set to nil at the initialization.

The table is implemented by the operations:

```
const
  max=10;
  max_minus_one=9;
type
  TPtr=^TItem;
  TKey_integer;
  TItem=record
```

```
    key:TKey;
    Data:TData;
    Next:TPtr
  end;
  TArr=array[0..max_minus_one] of TPtr;

TTable=TArr;
```

```
procedure TInit(var T:TTable);
(* procedure sets all items of hashing array to nil *)
var
  i:integer;
begin
  for i:=0 to max_minus_one do begin
    T[i]:=nil
  end;
end;
```

```
function Search(T:TTable;K:TKey):Boolean;
var
  index:integer;
  Ptr:TPtr;
  Found:Boolean;
begin
  index:=K mod max; (* index-phase)
  Found:=false;
  Ptr:=T[index];
  while (Ptr<>nil) and not Found do begin (* sequential search in the list of synonyms *)
    if Ptr^.Key=K
    then Found:=true  (* successful search *)
    else Ptr:=Ptr^.Next  (* moving ahead *)
  end;
  Search:= Found
end;
```

```
procedure TInsert(var T:TTable; K:TKey; El:TData);
var
  index:integer;
  Found:Boolean;
  Where:TPtr;
begin
  Search(T,K,Found,Where);
  (* This procedure is similar to the function TSearch, but it returns  in "where" the pointer of
found item or nil if not found *)
  if Found
  then Where^.Data:=El (* rewriting the old data *)
  else begin
    new(Ptr);           (* filling the new item *)
    Ptr^.Data:=El;
```

```
   Ptr^.Key:=K;
   Ptr^.Next:=Arr[K mod max]; (* insert the new item as the first into the list of synonyms *)
   Arr[K mod max]:=Ptr;
  end; (* if *)
end;
```

```
procedure TDelete(var T:TTable;K:TKey);
(* If procedure Search finds the item, the item is deleted from the list of synonyms, otherwise
no action takes place *)

begin
  (* create solution as the home assignment *)
end;
```

10.3 Hashing functions

To map the key to address needs typically two phases. In the first phase the key is transformed to positive integer. In the second phase the function modulo (mod) is used to transform any positive integer to the range of array used as the table.

The good hashing function should be sufficiently fast and should create low number of synonyms. There is not a general rule to find efficient hashing function. Looking for good hashing functions, the properties of the set of keys should be analyzed.

10.3.1 Hashing tables with implicit linking of synonyms

The previous method was based on dynamic allocation of items, which were linked explicitly by pointers. The following method uses "implicit linking" of synonyms, which are allocated in the hashing array, working as circular list.

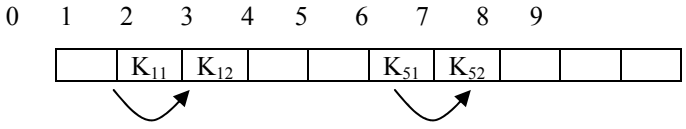
Every item has indicator "occupied", which is set to false at the initialization. All synonyms are inserted into the list, which is linked implicitly. Implicit linking means that the address of the next item is determined as the function of the address of current item. As an example, the

$$Addr(i+1)=Addr(i) + step$$

where "step" is some positive integer, like 1.

In this method the space of hashing table (an array into which the index access is directed) is overlapped by the space in which the synonyms are allocated. That means, that the same place, to which the hashing function is directed may be occupied by the item which is not synonymous with the given key.

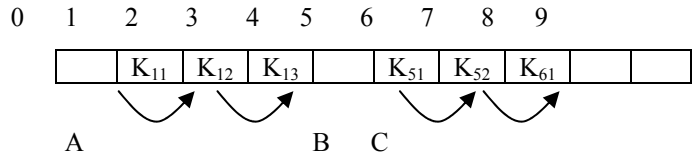
Let the key K_{11} be hashed into index A, and its synonym K_{12} is hashed into the same address. Let the key K_{51} and its synonym K_{52} be hashed into index B. The situation after insertion of the keys K1 and K1' is expressed in the following
figure:



A B

If the key K_{13} is to be inserted: the hashing function $H(K_{13})=A$. The conflict happens. The item $K_{13\ is}$ synonymous to K_{11} and K_{12} . The index A is occupied. The next item in the list at the index A+1 is occupied by K_{12} too. If the next item is free, it means that this is the end of the list of synonyms. This position is free for insertion of the key K_{13}.

Similar situation occurs with insertion of K_{61} . The hashing function gives index C. It is occupied. The insertion mechanism looks for the nearest free item, which indicates the end of synonyms (or associated items) and inserts the K_{61} into it. The situation is illustrated in the following figure:



The most important features of the above mentioned example are:

1. Hashing function may have the form: H = int(K) mod Max + 1; where int is the function transforming the key to positive integer.

2. Moving along the array, when reaching the index=max, the index 0 is to be set as the next. (Circular behaviour of the array).

3. The delete operation is not possible. By setting the deleted item to "not occupied", the link of the list would be destroyed. The only possibility how to delete the item is to "blind it". Blinding the item means to set its key to the value, which will never be searched. By blinding, the capacity of the table is decreased by every "blind" item.

```
const
  max=100;
type
  TItem=record
    Key:TKey;
    Data:TData;
    Occup:Boolean;
  end;
  TArr=array[1..Max] of TItem;
  TTable=record
    Arr:TArr;
    Count:integer
  end;
```

```
procedure TInit(var T:TTable);
```

```
var
  i:integer;
begin
  for i:=1 to max do begin
    T.Arr[i].Occup:=false;
    T.Count:=0;
end;

procedure TSearch(T:TTable; K:TKey; var Found;Boolean; var Where:integer);
var
  index,step:integer;
  begin
  index:= int(K) mod Max + 1; (* index access *)
  step:=1;
  Found:=false;
  while not Found and not T.Arr[index].Occup do begin
    if T.Arr[index].Key<>K
    then begin
      index:=index + step;   (* increasing the index *)
      if index > Max then index:=index-Max; (* circularity of the array *)
    end else begin
      Found:=true;
    end; (* if *)
  end; (* while *)
    Where:=index  (* index of found or of free position
                  for following insertion *)
  end; (*procedure *)
```

```
procedure TInsert(var T:TTable; K:TKey; El:TData; var OverFlow:Boolean);
var
  Found:Boolean;
  Where:integer;
begin
  Overflow:=false;
  Search(T,K,Found,Where);
  if not Found
  then begin
    if T.Count<Max
    then begin  (* insertion of the new item *)
      T[Where].Key:=K;
      T[Where].Occup:=true
      T[Where].Data:=El;
      T.Count:=T.Count+1;
    end else begin
      Overflow:=true  (* table overflow; no action *)
  end else begin
    T[Where].Data:=El; (* rewriting of the data of found item *)
end;
```

10.3.2. Hashing table with two hashing functions

The step of implicit link to the next item may be defined by another hashing function. The value of the step may be any integer from the range 1..Max-1 (step=Max would give the same address at circular array as for example (1 + max) mod max + 1 = 1.)). The second hashing function determining the step may be:
H2 = int(K) mod (Max-1) + 1

To prevent the visiting of the same item by the stepping process, the capacity of the table should be the prime number! (If the capacity is 10 and step is 2, the stepping process would visit only even indexes. If the capacity and step have the greatest common divisor not greater than 1, there will not be duplicated visits).

The above mentioned procedure TSearch may be modified for "two hashing tables" by changing the lines:

```
const
  max= (* suitable prime number *)
...
  step:= int(K) mod (max-1) + 1; (* determining the step by the "second
                                    hashing function" *)
...
```

10.4 Conclusion of searching

1) Sequential search in not ordered sequence is not efficient for a large number of items. The guarded search increases the speed of search.

2) Ordering the sequence according to the value of the key increases the efficiency of the unsuccessful search only. The keeping of ordering the array after the insert or delete operation take some time.

3. Binary search in the array decreases the time complexity (efficiency) to logarithmic value. Troubles with the keeping the array ordered are the same as in 2).

3. Binary tree allows very efficient insertion and deletion. The in-order pass creates ordered sequence. If the binary tree is weight-balanced, the access time is maximally logarithmic. To keep the tree balanced is not easy.

4. AVL trees are height-balanced binary search trees. The broken balance as the result of insertion or deletion can be re-established by limited rearrangements of the nodes in the near environment of the "Critical node". The height of the AVL tree (named according the Russian mathematicians Adelson-Velski and Landis) are maximally 1.7 times higher then relevant weight-balanced tress. AVL trees are not part of the syllabi of this course.

5. Hashing tables are based on index-sequential principle. H.T. with explicit linkage of items allow the delete operation while H.T. with implicit linkage of items in circular array doesn't

allow delete operation. Delete might be implemented by blinding the item - it means by rewriting the key by never searched k value. It decreases the effective capacity of the table.

ALGORITHMS AND DATA STRUCTURES
Lesson eleven

Lecture: Sorting - Ordering I.

11 ORDERING

11.1 Definitions

"Sorting" is distributing the elements of the set to groups of elements according to a defined feature attribute. (Example: Cars may be sorted to groups according to colors, fruits can be sorted according to kinds).

"Ordering" is a linear arrangement of the elements of a set, according to the relation based on the value of the key. The key is the necessary part of every element. Ordering is ascending (non-decreasing): (El[i] <= El[i+1]) or descending (non-increasing) : (El[i] >= El[i+1]).

"Merging" is creating one resulting ordered file from two or more ordered source ordered files. The type of source and resulting ordering is the same.

The term "sorting" is widely used instead of "ordering". We will use the term "sorting" with the meaning of the "ordering".

"Sequential sorting" is sorting process which processes the elements in the order, in which they are located in the source sequence.

"Stability" of the ordering is the feature of sorting process, which keeps the relative order of the keys with the same value.

Example: source sequence has three keys "5":

6 3 2 5' 1 5" 4 7 5'"

stable ordering will results in:

1 2 3 4 5' 5" 5'" 6 7
2

while the non-stable ordering doesn't keep the order of keys "5".).

"Naturalism" of the sorting is a feature, which expresses that the time necessary for sorting of randomly arranged sequence is greater than time necessary for sorting of the ordered sequence and that time is less than time necessary for ordering of the sequence ordered in inverse order.

Note:

Swapping notation: To make the code more understandable, the operator of swapping "**:=:**" will be used. To exchange the values of the two variables a,b, the statement a:=:b will be used instead of the sequence:

```
Temp:=a;
a:=b;
B:=Temp;
```

The swap operation is illegal in Pascal!!!

11.2 Ordering with the multiple keys

Let is the set of records representing the persons of the type:

```
const
  Size:1000;
type
  TMonth:(January,February,March,April,May,June,July,August,September,
        October,November,December);
  TPerson=record
            Name: string;
            Year:1800..2100;
            Month:TMonth;
            Day:1..31
          end;
  TArr=array[1..Size]of TPerson;
```

The task is to create two lists:

a) List ordered according to the age of each person (the oldest person is the first).

b) List according to the date of birthday in the current year. If two people have the birthday at the same date, the older precedes the younger.

The priority of the keys for the first list is: Year,Month,Day.
The priority of the keys for the second list is: Month,Day,Year.

There are two different approaches to solve the problem: using of the multiple key relation or using the multiple sorting run.

11.2 1 Multiple key relation

Multiple key relation "older" is Boolean function, which compares the data of the two persons and returns the "true", if the first person is older and "false" if the first person is younger or of the same age!

```
function Older(P1,P2:TPerson):Boolean;
(* Function "Older" serves as function "Less" in the process of sorting
the array of person to create the list of persons ordered according the
```

```
age *)
begin
  if P1.Year<>P2.Year
  then Older:=P1.Year<P2.Year (* Different years resolve the relation *)
  else (* The same value of years *)
    if P1.Month<>P2.Month
    then Older:=P1.Month<P2.Month (* Different months resolve the relation *)
    else Older:=P1.Day<P2.Day  (* Same months -> days resolves the
                                          relation *)
end;
```

Home assignment: Create the similar function "LessForBirthday", for sorting of the array of person to create the list of persons ordered according to their birthday.

11.2.2 Multiple sorting run

Let the array be:

```
var A:TArray;
```

If the array is ordered several times, every time according to a single key, starting with the Day, Month and ending with the Year, the resulting array will be ordered according to the age of the people. There are three procedures, each of them ordering the array according to a different key:

```
procedure SortDay(var A:TArray; Size:integer);
begin
  ...
end;
procedure SortMonth(var A:TArray; Size:integer);
begin
  ...
end;
procedure SortYear(var A:TArray; Size:integer);
begin
  ...
end;
```

The following sample of program will create the list of persons sorted according to the age.

```
begin
 ...
  SortDay(A,Size);  (* sorting array according day *)
  SortMonth(A,Size); (* sorting array according month *)
  SortYear(A,Size);  (* sorting array according year *)

end;
```

Sorting with multiple keys by successive sorting runs with increasing priority of keys should use STABLE sorting method. (Note: further explaining the sorting methods, the stability of every method will be mentioned).

11.3. Ordering without the movement of items of sorted array

Most important operations of all sorting algorithms are "comparing of the keys of the two items" and "move of item (swapping of two items) of sorted array". Number of these operations determines the time complexity of the sorting method. If the moved item is too long, the moves of such items increase the time necessary for sorting. The ordering without movement of items solves the problem and reduces the sorting time.

The ordering without movement of (long) items uses auxiliary array (IndArr) with indexes.

Initial state of sorted array A and its column of indexes:

A

Index	Data	Key	IndArr
1		Paul	1
2		Adam	2
3		Denis	3
4		Martha	4
5		Eva	5

The sorting without movement of items moves only values of the IndArr column (which are short) and keeps the data and key of items at their place. The state of IndArr expresses the result of sorting by ordering the indexes according to the values of their items in the array A.

A

Index	Data	Key	IndArr
1		Paul	5
2		Adam	1
3		Denis	2
4		Martha	4
5		Eva	3

The pass through the ordered array and processing the items is done by the cycle:

```
for i:=1 to Size do begin
  Process(A[IndArr[i]]).
```

Really sorted array B from the array A sorted without movement the cycle is created by:

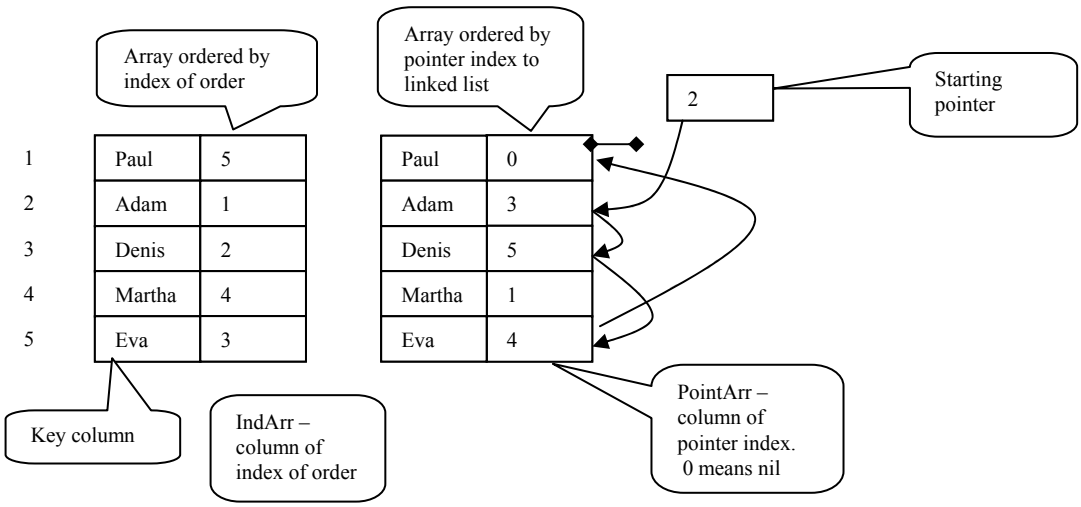
```
for i:=1 to Size do B[i]:=A[IndArr[i]];
```

To modify method normal sorting method (with the movement of items) two types of statement are to be changed:

with movement	without movement
A[i]:=:A[j]	IndArr[i]:=:IndArr[j]
A[i].K<A[j].K	A[IndArr[i]].K < A[IndArr[j]].K

While moves are done only in the IndArr, the comparison is done in the source array via index in the IndArr.

The array sorted by IndArr may be converted to ordered linked list by means of array of index-pointers PointArr.



The PointArr is created by the sequence of statements:

```
First:= IndArr[1];  (* pointer to the first item of the list *)
for i:=1 to Size-1 do PointArr[IndArr[i]]:=IndArr[i+1];
PointArr[IndArr[N]]:=0; (* nil *)
```

McLarren's algorithm rearranges the items in the array A without the need of temporary array. The algorithm works "in situ". It is difficult

to understand it at the first glance:

```
(* McLarren algorithm *)
i:=1;
Temp:=First;
while i<Size do begin
 (* looking for the successor moved to the position greater then i *)
  while Temp<i do Temp:=PointArr[Temp];
  A[i]:=:A[Temp];  (* exchange of the items *)
  PointArr[i]:=:Temp; (* exchange of pointers *)
  i:=i+1  (* the first i items are located at their place *)
end;
```

11.4 Classification of algorithms

a) According to the type of memory the sorted structure is located in the methods are divided to:

- methods of internal sorting (sorting of arrays, sorting with random access)
- methods of external sorting (sorting of files, sorting with sequential access)

b) According to the type of processor running the sorting process sorting algorithms are divided into:

- serial algorithms - are processed on the processor allowing only one operation at a time.
- parallel algorithms - are processed on the parallel processor allowing execution of the multiple operation at the same time.

Note: This course doesn't expose the parallel algorithms.

c) According to the essential principle of the sorting the methods are classified to two complementary couples:

Algorithms based on **selection**
Algorithms based on **insertion** (* selection is complementary to insertion *)

Algorithms based on **partition**
Algorithms based on **merging** (* partition is complementary to merging *)

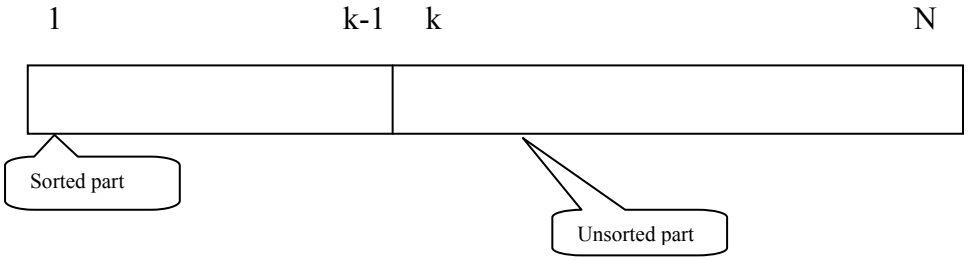
Note: For the following examples we assume, that the type TArr is defined with const

```
  N=100; (* size of sorted array *)
type
  TArr=array[1..N] of integer;

  (* the only component of array item is integer key *)
```

11.5 Sorting methods based on selection

11.5.1 Straight selection-sort

Let the sorted array with N items be at a moment split into two parts: 1..k-1, k..N. The left part contains all elements which are less than any element of the right part. The sorting step is represented by searching for minimal item of right hand part and swapping it with the first (k-th) item of the right hand part. It is done by the following sequence of statements:




```
   Min:=A[k];  (* initial setting  *)
   MinInd:=k;
   for i:=k+1 to N do begin
     if Min > A[i]
(* for sorting with the multiple keys the comparison function
  "less(Temp,A[i])" is used instead of "Temp<A[i]" *)

     then begin
       Min:=A[i];  (* storing the better minimum *)
       MinInd:=i;  (* storing the index of better minimum
     end;
   A[k]:=:A[MinInd];  (* swapping of floating first item of the right hand
                         part with the found  minimum *)
```

If we start the sorting with the empty right hand part and full left hand part of an array and repeat the above mentioned sequence N times, the array is sorted. The sorting procedure has the form:

```
procedure SelectSort(var A:TArr);
var
  Min,MinInd,i,j,k:integer;
begin
  for j:=1 to N-1 do begin
    k:=j;
    Min:=A[k];
    MinInd:=k;
    for i:=k+1 to N do begin
      if A[i] < Min
```

```
(* for sorting with the multiple keys the comparison function
  "less(Temp,A[i])" is used instead of "Temp<A[i]" *)

      then begin
        Min:=A[i];
        MinInd:=i;
      end; (* for i:= *)
    A[k]:=:A[MinInd];
  end; (* for j:= *)
end; (* procedure *)
```

Notice the modification of the algorithm for the sorting with multiple keys.

Analysis:

a) The method is not stable. After the swapping k-th item with minimum it may happen, that the k-th item might be positioned behind the item with the same value, and that way is violated the relative ordering of the keys with the same value.

b) The time complexity of the method is quadratic:

- average time complexity: TA(n) ≈ 4.5 * sqr(n)
- maximal time complexity: TM(n) ≈ 6 * sqr(n)

c) Experimental values of time units for an array with n items ordered initially randomly (RAO) and ordered initially inversely (INV) are in the following table: (numbers in brackets expresse the size of array, the others the time units).

N	(128)	(256)	(512)
INV	64	244	968
RAO	50	212	744

The behaviour of the ordering method is "natural".

11.5.2 Bubble sort

Bubble sort is well known to programming amateurs and very popular. Its principle is the same, but the way how to find and swap the minimum is different and less efficient.

Bubble sort compares all pairs of neighbouring items and swaps the couple and checks if it is ordered inversely. If yes, it swaps items in the pair:

```
if A[j-1]>A[j]
then A[j-1]:=:A[j];
```

This is done for each item from left to right (or from right to left). After one pass, the maximal item (going from right to left) is moved by swapping to the leftmost position. This is the selectkion process combined with swapping process. If the pass of comparing is finished with no swapping, the sorting process is finished.

The procedure of bubble sort has the form:

```
procedure BubbleSelectSort(var A:TArr);
(* Array A is array of 100 items *)
var
  i,j,N:integer;

  Fin:Boolean;  (* Boolean variable indicating the end of sorting *)
begin
   N:=100;
  i:=2;
  repeat
    Fin:=true;
    for j:=N downto i do begin
      if A[j-1]>A[j]
      then begin
        A[j-1]:=:A[j];
        Fin:=false;  (* swapping occurs, sorting not finished yet *)
      end (* if *)
    end; (* for j:=N *)
    i:=i+1;  (* first i items are sorted already *)
  until Fin or (i>N);
end; (* procedure *)
```

Note the modification of the algorithm for sorting with no movement of the items of sorted array.

Analysis:

a) The Bubble-select-sort algorithm is stable! It may be used for the sorting with multiple keys using the method of multiple sorting runs according to the keys with increasing priority.

b) The algorithm behaves naturally.

c) The time complexity is quadratic:
maximal time complexity: TM(n) ~= 11.5 * sqr(n)

Experimental values of time units for an array with n items ordered initially randomly (RAO) and ordered initially inversely (INV) can be seen in following table: (numbers in brackets expresses the size of array, other the time units).

N	(256	(512

	)	)
INV	338	1562
RAO	558	2224

11.5.3 Heap sort

Heap sort is the most important sorting method based on the selection. "Heap" is a binary tree-like structure, here implemented by the array. For all nodes of the tree stands:

"father-node" is greater than both "child-nodes".

If it is true, then maximal value of all nodes is located in the root position.

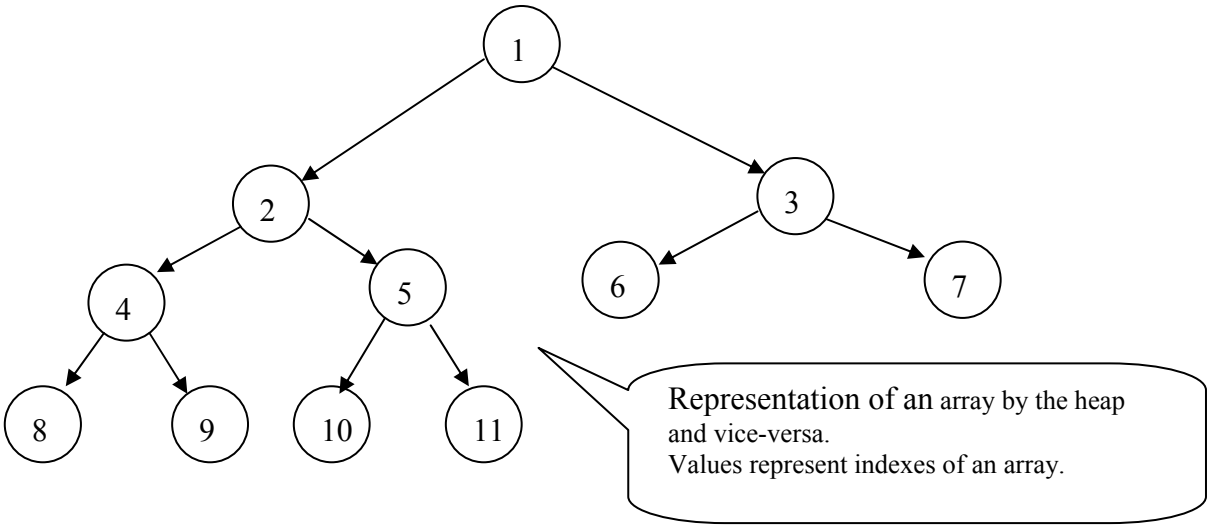
Suppose the root is removed and its value is substituted by some other value. The problem is to re-establish the rules of the heap. It is done by the procedure "SiftDown".

Procedure SiftDown compares sequentially all "father-nodes" with this node of the couple of "child-nodes", which is greater. If this child-node is greater than the father it both are swapped and the comparison is repeated downstairs until there is no need for swapping or the terminal node of the branch has been encountered.

What is important is the implementation of the tree by an array. There is an implicit linking applied: if ***i*** is the index of "father-node", then left hand child-node has index=2*i and right hand child-node has index=2*i+1. This way an array with indexes:

1	2	3	4	5	6	7	8	9	10	11

will represent the tree:



If N is the size of an array (and the number of nodes of the heap-tree too), then for the terminal node the condition "(2*i) > N" is true.This is the way how to recognize the terminal node passing down along the branch.

The procedure SiftDown which re-established the heap-rules has the form:

```
procedure SiftDown(var A:TArr;Left,Right:integer);
(* Left is the "father" index of sifted item; Right is the size of an
   array *)
var
  i,j:integer;
  Cont:Boolean;  (* Cycle control variable "Continue" *)
  Temp:integer; (* Auxiliary variable of the same type like item *)
begin
  i:=Left;
  j:=2*i;  (* Index of the left child-node *)
  Temp:=A[i];
  Cont:=j<=Right;
  while Cont do begin
    if j<Right
    then (* Node i has both child-nodes *)
      if A[j]< A[j+1]
      then (* The right son is greater *)
        j:=j+1; (* set the j to the greater son of the pair *)

    if Temp >= A[j]
    then (* Element Temp has been sifted to its position;
            cycle finishes *)
      Cont:=false
    else begin (* Temp is sifted to lower position and A[j] is emerging
                  one level higher *)
      A[i]:=A[j];  (* emerging *)
      i:=j;        (* son becomes the next "father",
                      preparation for the next pass *)
      j:=2*i;      (* next "left son" *)
      Cont:=j<=Right; (* condition : "j is not terminal node" *)
    end (* if *)
  end; (* while *)
  A[i]:=Temp; (* final positioning of the sifted root *)
end; (* procedure *)
```

To establish the heap from the randomly organized array, the SiftDown procedure is used. Starting with the lowest and the rightmost "father-node" the sub-heaps are established for all other "father-nodes" up to the root. The index of the lowest and the rightmost "father-node" is (N div 2);

(Note, that in the above placed figure of the heap-tree with the 11 nodes, the lowest and the rightmost "father-node" has index 11 div 2 = 5).

The following statement makes the heap:

```
for i:=Right div 2 downto 1 do SiftDown(A,i,Right);
```

The Heap-Sort works in the following way:

1. Create heap. The maximal value is in the root on the index=1.
2. Swap the root with the currently last index. Its result is violating the rule of heap in the root. The right part of an array - from currently last index to N is ordered.
3. Currently last index is thereby decreased by 1.
4. Calling the SiftDown the heap is re-established.
5. The cycle is repeated until the right part contains entire array.

The resulting procedure has the form:

```
procedure HeapSort(var A:TArr);
var
  i,Left,Right:integer;
begin
  Left:= N div 2; (* index of the rightmost, lowest father-node *)
  Right:=N;
  for i:= Left downto 1 do SiftDown(A,Left,Right); (* Creation of the heap *)

  for Right:=N downto 2 do begin  (* Cycle of the own Heap-sorting *)
    A[1]:=:A[Right];  (* swapping of the root with the currently last
                         item *)
    SiftDown(A,1,Right-1)  (* Re-establishing of heap *)
  end; (* for *)
end; (* procedure *)
```

Analysis:

a) Heap-sort is not stable! Heap-sort does not behave naturally! Method works "in situ" (no need for additional array space!).

b) Time complexity has linearithmic order (n * lg(n)).

- maximal time complexity: TM(n) ~= 31.5 * n * lg(n-1)

Experimental values of time units for an array with n items ordered initially randomly (RAO), ordered initially inversely (INV) and initially properly ordered (IPO) are in the following table: (numbers in the table expresses time units).

n	(256)	(512)
IPO	42	210
RAO	38	186
INV	40	196

The home assignments:

1) Use the Select Sort to create the list of students ordered according to the multiple keys.

The necessary types are:
TBranch = (Informatics, Electronics, Cybernetics, PowerEng);
TStudent = record
Name:string;
YearOfStudy:integer;
Branch:TBranch;
GPA:real;
IdentNo:integer
end;
TArray=array[1..200] of TStudent;
Suppose the a
The created list should be ordered according to:

Branch, Year in the Branch, GPA in the year, Name in the GPA.

2) Create the procedure which sorts the source array A full of students without movement of items, creates the inner array of indexes and ordered linked list (by using the inner array of pointers) and place the resulting array into output array B. As a sorting key use IdentNo.

procedure Sort(A:TArr; var B:TArr);

3) Modify the SelectSort for sorting without movement of the items. Sort the array A of students without the movement of items according the name. Create the array B:TArr, which will be finally sorted, by moving the items from A to B.

4) Modify the SeletSort for sorting without movement of the items. Sort the array A of students by name without the movement of items. Create the linked list of items by means of PointArr. Sort the array A "in situ" by means of McLarren algorithm.

5) Modify the Bubble-Select-Sort for sorting with multiple key. From the input array AI, create two output arrays AAge and ABirth, containing the list of persons sorted by the age (AAge) or a list of persons sorted according to the birthday.

The type of item is:
type
TPerson = record
Name: string;

```
    Year:integer;
     Month:integer;
     Day:integer
  end;
```

6) Modify the Heapsort the same way as the first and third previous exercises (for sorting without movement and for sorting with multiple keys). Use source array A and destination array B of students. Sort them without movement according the GPA and the result place into the array B.

Lecture: Sorting - Ordering II.

12 Sorting by insertion

12.1 Insert sort

Principle of insertion is very similar to the method, which would be used in the common life. Let the set of ordered elements be divided to two parts, where the first is ordered and the second is not ordered. One step of ordering by insertion consists of:

- taking one element from the not ordered part and
- searching the proper place (index) in the first part, which, if the element is inserted into, it keeps the previous ordering.

If the array is initially divided into two parts:

A[1] ordered part
A[2]..A[N] not ordered part

then the general form of the insert-like algorithm is:

```
for i:=2 to N do begin
  "find the index K (1<=K<=i-1) , into which the A[i] may be
   inserted";
  "Shift the segment of array from index K to i by one and make place
    for inserted element"
  "Rewrite the A[K] by  A[i]"
end;
```

12.2 Bubble-Insert-Sort

The simplest insert sort is Bubble-Insert-Sort. The method make searching and shifting in one cycle by comparing and swapping all adjacent pairs of elements. The following algorithm uses the guard
element A[0], by which is the array extended and which is not the regular item of sorted array.

```
procedure BubbleInsertSort(var A:TArray);
var
  i,j:integer;
  Tmp:integer;
begin
  for i:=2 to N do begin
    Tmp:=A[i];
    A[0]:=Tmp; (* establishing the guard *)
    j:=j-1;
    while Tmp<A[j] do begin (* search and shift element *)
      A[j+1]:=A[j];
      j:=j-1
    end; (* while *)
    A[j+1]:=Tmp; (* insertion *)
```

```
  end (* for *)
end; (* procedure *)
```

Analysis:

a) The Bubble-Insert - Sort is stable method and it behaves naturally.

b) The time complexity is quadratic:
average time complexity TA(n) ≈ 3.75 * sqr(n)
maximal time complexity TM(n) ≈7.5 * sqr(n)

c)Experimental values of time units for an array with n items ordered initially randomly (RAO), ordered initially inversely (INV) and initially properly ordered (IPO) are in following table: (numbers in the table expresses time units).

n	(256)	(512)	(1024)
IPO	4	6	14
RAO	156	614	2330
INV	312	1262	5008

12.3. Insert sort with the binary search

This variant of insert sort searches for inserted place by using the binary search. Exploiting of Dijkstra's binary search ensures the stability of the sorting mechanism, as if there is multiple
occurrence of the elements with the same value of the key, the position behind the last of the same is determined. The procedure has the following form:

```
procedure BinaryInsertSort(var A:TArr);
var
  i,j,m,Left,Right,Insert:integer;
  Tmp:integer;
begin
  for i:=2 to N do begin
    Tmp:=A[i];
    Left:=1; Right:=i-1; (* setting the left and right boundary of
                         searched segment *)
      while Left <=Right do begin (* standard binary search *)
        m:=(Left+Right)div 2;
        if Tmp<A[m]
        then Right:=m-1
        else Left:= m+1;
      end;
      for j:=i-1 downto Left do
        A[j+1]:=A[j]; (* shift of the array segment *)
      A[Left]:=Temp;
  end; (* for *)
end; (* procedure *)
```

Analysis:

a) The method is stable and it behaves naturally.

b) The time complexity is quadratic:
average time complexity TA(n)~=2.25 * sqr(n)
maximal time complexity TM(n)~=4.5 * sqr(n)

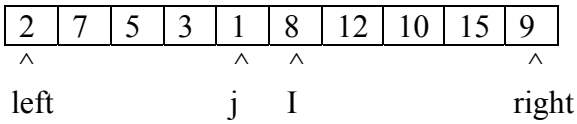
n	(256)	(512)	(1024)
RAO	134	502	1930
INV	248	956	3736

12.4 Sorting by partition - Quick-Sort

Quick-Sort is the fastest and widely used method. It is based on the principle of partition. Suppose there exist the algorithm, which re-arranges the given part of an array to two parts: in the left hand part are all items less than the special value X and in the right hand part are all items greater then value X. If X is median of all items of an array, then the array is halved to two equally sized parts separated by the median. Let the procedure implementing the mechanism has the headings of
form:

procedure Partition(var A:TArr; Left,Right:integer; var i,j:integer);

Procedure Partition divides the array A[Left..Right] to two sub-arrays
A[Left..j] and A[i..Right]. See the following figure:



If we have such mechanism, we can write the sort algorithm which is recursively partitioning both generated parts while there exist a part of array, which can be partitioned. The condition for further
partition of the originated left part is (Left<j) and for further partition of the right part is (i<Right). The whole procedure has the form:

```
procedure QuickSort(var A:TArr, left, right:integer);
(* Left is variable set to the 1 and Right set to the N before
the first call of QuickSort *)

var i,j:integer;
begin
  Partition(A,Left,Right,i,j);
  if Left<j then QuickSort(A, Left, j);  (* Recursive call for the left part *)
```

```
  if i<Right then QuickSort(A, i, Right);  (* Recursive call for the right part *)
end;
```

The secret of the speed of Quick-Sort is in the efficiency of partition mechanism, invented by the famous programmer C.A.R. Hoare. Instead of median, algorithm takes the PM value from the middle of array. The value is called pseudo-median. In the first cycle, the index i, initially set to
1, is increased while the A[i] which is greater (or equal) to pseudo-median is found. In the second cycle, the index j, initially set to N, is decreased while the A[j] which is less (or equal) to
pseudo-median is found. Both found elements are swapped. The process of finding the pair of elements and mutual swapping is repeated until the indexes i and j are crossed by one to another. The algorithm is implemented by the following procedure:

```
procedure Partition(var A:TArr; Left,Right:integer; var i,j:integer);
var
  PM:integer;
begin
  i:=Left;    (* initial setting of i *)
  j:=Right;   (* initial setting of j *)
  PM:=A[(i+j) div 2]; (* Establishing the value of pseudo-median *)
  repeat
    while A[i] < PM do i:=i+1; (* Searching from the left to find the
                        nearest i for which is valid A[i]>=PM  *)
    while A[j] > PM do j:=j-1; (* Searching from the right to find the
                        nearest j for which is valid A[j]<=PM  *)
    if i<=j
    then begin
      A[i]:=:A[j]; (* swapping of the found elements *)
      i:=i+1;
      j:=j-1
    end
  until i>j; (* cycle finishes when the indexes cross one another *)
end; (* procedure *)
```

12.4.1. Non-recursive version of Quick-Sort

Nonrecursive version of Quick-Sort uses stack for saving one of the two intervals generated by procedure partition. The other is taken for the next partitioning. The stack should contain in the worst case the n-1 pairs of interval indexes. Intervals are compared just after their origin, the pair of indexes of larger interval is saved on the stack and pair of indexes of smaller is submitted to following partition.

```
procedure NonRecQuicksort (left,right:integer);
var
  i,j:integer;
  S:TStack;   (* declaration of ADT stack *)
```

```
begin
  SInit(S); (* initialization of the stack *)
  Push(S,left);     (* initial interval
   Push(S,right);
  while not Sempty do begin (* outer cycle, while stack is not empty *)
    Top(S,right);Pop(S); (* Reading from the stack is done in inverse order *)
    Top(S,left); Pop(S);
    while left<right fo begin (* inner cycle, while there is
                                 something to partition *)
      Partition(A,left,right,i,j);
      Push(S,i);  (* saving interval of the right part to stack *)
      Push(S,right);
      right:=j;   (* preparing of the right index for the next cycle *)
    end; (* while *)
  end (* while *)
end; (* procedure *)
```

Analysis

a) Quick sort is not a stable method and it doesn't work naturally. Due to recursion it doesn't work "in situ". Its speed is given by the fact that items bring themselves near to their proper place by a larger step than 1. This is done by swapping the distant items in the procedure partition.

b) The time complexity is linearithmic:

- average time complexity TA(n)~=17 * n * lg(n)
- time complexity for initially properly ordered array TIPO(n)≈9 * n * lg(n)
- time complexity for inversely ordered array TINV(n)~=9 * n * lg(n)

Quick-sort is one of the most efficient sorting methods.

c) Experimental values of time units for an array with n items ordered initially randomly (RAO), ordered initially inversely (INV) and initially properly ordered (IPO) are in following table: (numbers in the table expresse time units).

n	(256	(512)	(1024)
IPO	10	24	50
RAO	22	48	50
INV	12	26	56

12.5 Sorting by merging

Principle of merging is based on unification of two or more ordered sequences into one resulting list. If n1 is the size of the first sequence 1 and n2 is the size of the second sequence, then the time complexity of the merging of these two lists into one resulting list is (n1+n2). The most frequently the sequences are implemented by files of linked lists.

The following examples expose the principle of merging of files and lists.

12.5.1 Merging procedure for two files with the use of access variable

```
procedure MergeFiles1(var F1,F2,F3:TFile);
(* Procedure uses access variable to file, according to ASCII Pascal    rules. Ordered files F1
and F2 are merged into file F3. Not possible in    TURBO and BORLAND Pascal !!! *)

var Temp:TypElemOfFile; (* Base type of files *)
begin
  reset(F1); reset(F2); rewrite(F3);

  while not eof(F1) and not eof(F2) do
    begin
      if F1^ < F2^   (* F1^ and F2^ are access variables to files F1
                     resp F2 *)
      then begin
             read(F1,Temp);
             write(F3,Temp)
           end
       else begin
             read(F2,Temp);
             write(F3,Temp)
           end;  (* The cycle finishes by exhausting of one of the two
                    files *)
   (* end of while *)

  while not eof(F1) do  (* Adding of the rest of not-empty file to the
                        result file *)
  begin
    read(F1,Temp);
    write(F3,Temp)
  end;

  while not eof(F2) do  (* Adding of the rest of not-empty file to the
                        result file *)
  begin
    read(F2,Temp);
    write(F3,Temp)
  end
end.
```

12.5.2 Procedure for merging of two files without use of access variable

```
procedure MergeFiles2(var F1,F2,F3:TFile);
(* For processing of files procedure uses predicate eof and  statements read and write *)
```

```
var A,B: TElemOfFile  (* Temporary variable of the base type *)
begin
  reset(F1); reset(F2); rewrite(F3);
  if not eof(F1) and not eof(F2)
  then                    (* both files are not empty *)
    begin
      read(F1,A); read(F2,B);
      repeat
        if A<B
        then begin write(F3,A);
                   if not eof(F1) then read(F1,A)
             end
        else begin write(F3,B);
                   if not eof(F2) then read(F2,B)
             end;
      until eof(F1) or eof(F2);
        (* One file is empty, one element of the second file is not
           written *)

      if A < B
      then write(F3,B)   (* Not written was B *)
      else write(F3,A);  (* Not written was A *)
    end;   (* One and/or both files are empty, all read is written *)



   while not eof(F1) do  (* writing of not empty file F1 *)
   begin
     read(F1,A);
     write(F3,A)
   end;

   while not eof(F2) do (* writing of not empty file F2 *)
   begin
     read(F2,B);
     write(F3,B)
   end;

       end
end (* procedure *)
```

12.5.3 Procedure for merging of the two lists

```
procedure MergeLists(L1,L2:TList;var L3:TList);
var TempPoint  (* Temporary pointer moving along the list *)
             :TList;
begin
```

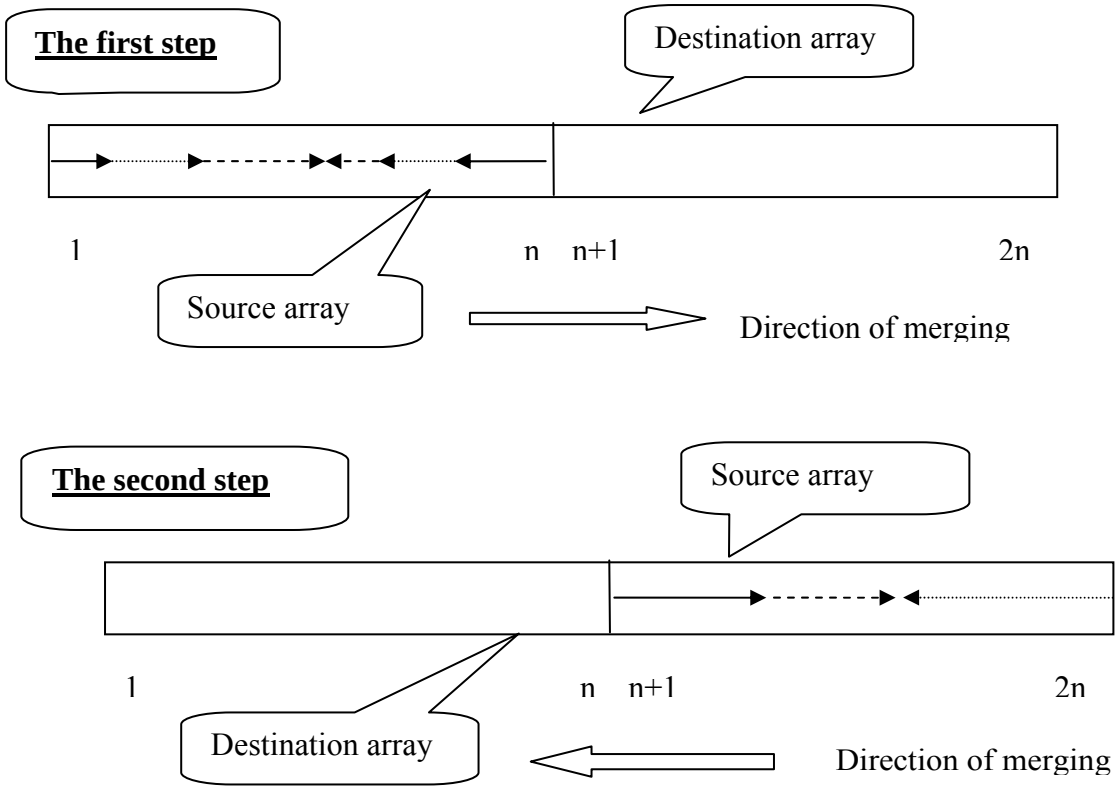
```
  new(L3);     (* creating the headings *)
  TempPoint:=L3;
  while (L1<>nil) and (L2<>nil) do     (* cycle of merging *)
  if L1^.Data < L2^.Data
  then begin
        TempPoint^.RPtr:=L1;
        TempPoint:=L1;
        L1:=L1^RPtr
       end
  else begin
        TempPoint^.RPtr:=L2;
        TempPoint:=L2;
        L2:=L2^RPtr
       end
(* Joining of the not-empty list *)

 if L1=nil
 then TempPoint^.RPtr:=L2
 else TempPoint^.RPtr:=L1
 (* Destroying the headings *)
 TempPoint:=L3;
 L3:=L3^.RPtr;
 dispose(TempPoint)
end;
```

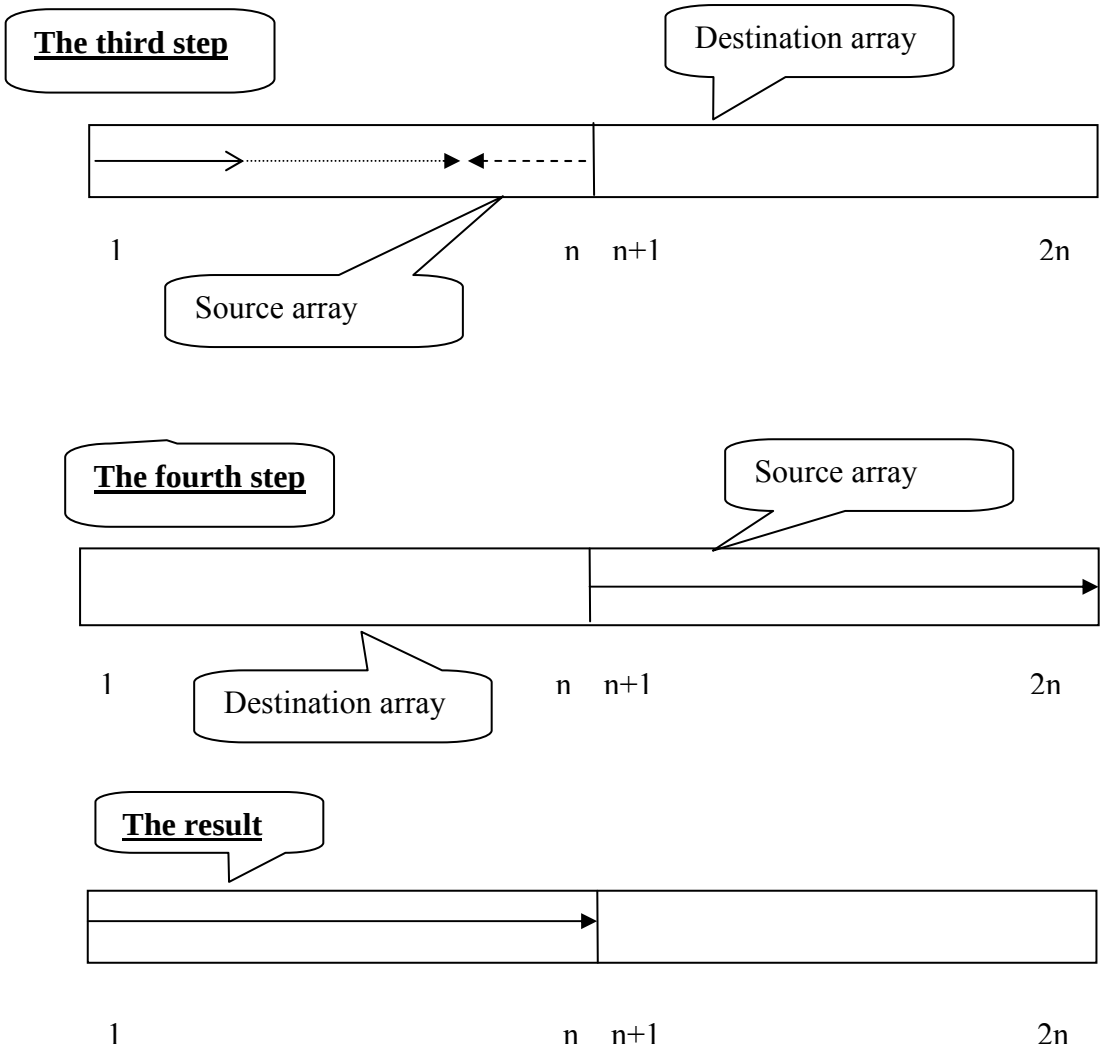
12.5.4 Merge-sort

Merge sort is a sorting method for sorting of arrays. The principle is described as follows:

The ordered set is located in the left half of the double sized array. This part is called "the source array" while the other is called the "destination part". The situation is expressed in following figure:

1st step


The first step
Destination array
1
n
n+1
2n
Source array
Direction of merging
The second step
Source array
1
n
n+1
2n
Destination array
Direction of merging



The algorithm takes items from the left and from the right of the source array while they constitute the non-decreasing sequence from the left and non-decreasing sequence form the right and merge them into one destination sequence, which is located alternatively from the left and from the right to the destination array. This way the total number of 6 no-decreasing sequence is reduced to one half. See the figure with the 1st step.

In the second step, the role of source and destination arrays is swapped. The process of merging of pairs of counter-directed sequence continues, producing another halving of the number of non-decreasing sequences. See the figure with the second step.

If the merging process results in one non-decreasing sequence in the destination array, that part of array is finally sorted. If the result is located in the right part and it is to be located in the left part,
additional cycle moves the sorted array from the right half to the left half of the double sized array. See the figure of the 3rd step.

General form of the algorithm is expressed as follows:

```
procedure MergeSort(var A:TArr);
(* TArr is double n sized array *)
var
  FromLeft:Boolean;
  i,j,k,p: integer;
  Count,Step:integer;

  FromLeft:=true; (* variable controlling the direction from source to
                     destination part of array *)
  repeat
    if FromLeft
    then begin
      i:=1;  (* left index of source array *)
      j:=n;  (* right index of source array *)
      k:=n+1;  (* left index of destination array *)
      p:=2*n;  (* right index of destination array *)
    end else begin
      k:=1;  (* left index of source array *)
      p:=n;  (* right index of source array *)
      i:=n+1;  (* left index of destination array *)
      j:=2*n;  (* right index of destination array *)
    end;

    Count:=0; (* counter of non-decreasing sequences in destination
                 array *)


    Step:=1; (* step has the value 1 or -1, according the direction of
                success in the processed sequence *)
    repeat

      "Merge one pair od non-decreasing sequences from the source array
       from i to left  and from j to right and locate the resulting
       sequence form the k with the step=Step ";

      k:=:p (* change of the stating index of next destination sequence
               *)
      Count:=Count+1; (* actualization of the number of destination sequences *)

      Step:= -Step;  (* inversion of the polarity of the Step *)
```

```
   until i=j; (* indexes had met within the source array *)

   FromLeft:= not FromLeft; (* inversion of the direction *)

  until Count=1; (* Sorting cycle finishes if the number of destination
                non-decreasing sequences is one *)

  if not FromLeft
  then "Copy the right half of array to left half";
(* Final copy of the  result to the proper - left part of an array *)
```

Note: Abstract statements, described in quotation marks should be implemented in detail.

Analysis:

a) method is not stable, does not behave naturally and does not work "in situ" as it needs double sized memory space for sorted data.

b) Maximal time complexity TM(n)~= (28 * n + 22) lg(n)

c) Experimental values of time units for an array with n items ordered initially randomly (RAO), ordered initially inversely (INV) and initially properly ordered (IPO) are in the following table: (numbers in the table express time units).

n	(256)	(512)
IPO	8	13
RAO	6	12
INV	32	72

12.6 List-merge sort

List-Merge-Sort uses merging principle applied on lists, which are created with the use of auxiliary pointer array for non-decreasing sequences of elements of sorted array. Let us look at the following figure. There are five non-decreasing sequences (one of them - 6 - has single item). The Auxiliary array of pointer linking the items into the list is called APtr. Index in APtr is the pointer of next item. Zero means "nil". Non-decreasing sequences are illustrated by arrows.

Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	2	5	9	3	7	10	16	5	8	6	3	11	18	20
APtr	2	3	0	5	6	7	0	9	0	0	12	13	14	0

The initializing procedure makes linking of all consequent non-decreasing sequences and potting their starting indexes into the Queue of indexes.

```
procedure InitListMergeSort(A:TArr; var APtr:TAPtr; var
                      QueForIndex:TQue);
(* proceure link all non-decreasing sequences by means of APtr and stores
all their starting indexes into the Queue "QueForIndex" *)
var
  i:integer;

begin
  QInit(QueForIndex);
  QueUp(QueForIndex,1);  (* storing the first starting index in the
                      queue *)
  for i:=1 to n-1 do begin
   if A[i+1]>=A[i]
   then begin (* non-decreasing sequence continues *)
     APtr[i]:=i+1  (* the index of the next item is set to APtr *)
   end else begin (* end of non-decreasing sequence *)
     APtr[i]:=0     (* "nil" is set to APtr *)
     QueUp(QueForIndex, i+1); (* storing the starting index of
              following non-decreasing sequence into the Queue *)
  end; (* for *)
  APtr[n]:=0; (* final "nil" is set *)
end; (* procedure *)
```

Procedure MergeLists merges two source non-decreasing list into one destination non-decreasing list:

```
procedure MergeLists(A:TArr;var AP:TArrPtr; si1,si2:integer; var
                  di:integer);

(* A ... Sorted array; AP ... Array of pointers for linked lists;
   indexes of si1,si2... sources lists 1 and 2; di is index of destination
   list; indexes serve as pointers *)

begin
  if A[si1] <= A[si2]  (* establishing of the begin of
                          resulting list *)
  then begin
    di:=si1;
    si1:=AP[si1]
  end else begin
    di:=si2;
    si2:=AP[si2];
  end;
```

```
  TmpInd:=di;  (* Temporary index is set to the first element of
                  resulting list *)
  while (si1<>0) and (si2<>0) do begin
    if A[si1] <= A[si2]
    then begin      (* First item of the si1 list is less and added to
                       resulting list *)
      AP[TmpInd]:=si1;
      si1:=AP[si1];
    end else begin    (* First item of the si2 list is less and added to
                       resulting list *)
      AP[TmpInd]:=si2;
      si2:=AP[si2]
    end; (* if *)
    TmpInd:=AP[TmpInd];   (* shifting the pointer along resulting list
                             by one item *)
  end; (* while *)

 (* Joining of remaining non-empty list *)
  if si1=0
  then AP[TmpInd]:=si2
  else AP[TmpInd]:=si1;
end;
```

The sorting algorithm takes two starting indexes of non-decreasing sequences from the queue, and merges them into one resulting list. Index of resulting list is inserted into the queue. This cycle is repeated until the single item - "starting index" is contained in the queue.

```
procedure ListMergeSort(var A:TArr);
(* resulted array is sorted *)
var
  QueForIndex:TQue;
  APtr:TAptr;
  Fin:Boolean;
  s1,s2,sd :integer; (* starting indexes of two source and one  destination list *)
begin
  InitListMergeSort(A, APtr, QueForIndex);
  Fin:=false;

  while not Fin do begin
    if not QEmpty
    then begin
      Front(QueForIndex,s1);  (* taking the first starting index *)
      Remove(QueForIndex);
      if not QEmpty
      then begin
        Front(QueForIndex,s2);  (* taking the second starting index *)
        Remove(QueForIndex);
        MergeLists(A, APtr, s1,s2,sd);  (* merging of the two lists *)
        QueUp(QueForIndex,sd)  (* storing the result into the queue *)
      end else begin
```

```
      Fin:=true (* the last, single and resulting list starts at s1 *)
  end; (* while *)

(*  "Here  the "McLarren's algorithm can be applied, which rearranges the list-
    like ordered array into the ordered array "; *)

end; (* procedure *)
```

12.7 Shell sort

Shell sort is very efficient and simple sort. It is very popular. The principle is based on natural behavior of used sorting method. The items in sort array are subdivided to multiple sequences with the certain step. If there is an array of items on indexes:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

The last three stages of the bubble-sort are demonstrated as:

Array is taken as four subsequences with the step 4:

1st : 1 5 9 13 17
2nd : 2 6 10 14 18
3rd : 3 7 11 15 19
4th : 4 8 12 16 20

These four arrays are processed by Bubble-pass with the step 4.

The following action is 2-Sort. The original array is subdivided into 2 parallel sub-arrays, with the step 2.

1st : 1 3 5 7 9 11 13 15 17 19
2nd : 2 4 6 8 10 12 14 16 18 20

Both arrays are processed by Bubble-pass with the step 2.

The last step is 1-sort. Here the original array is rearranged by the "Buble-pass" with the step 1. In the sorting processes with the greater step, the items get nearer to their proper place with the greater step. Consequent processings make only "corrections" in sorted sequence.

The starting step of the sorting method is one half of the size of sorted array and the first pass (n div 2) - processes (n div 2) parallel arrays (all consisting from two items only). Following pass
process halves the step to (n div 4) etc. Last pass is the Bubble-pass with step=1.

```
(* Shell sort - sorting with decreasing increment *)

(*   Kerningham variant *)
```

```
procedure ShellSort(var A:TArr, N: integer);
var
  step,I,J:integer;
begin
  step:=N div 2; (*  The first step = half of size of array *)

  while step > 0 do begin  (* step is halved gradually *)

    for I:=step to N-1 do begin  (* cycle for all parallel sequences *)
    J:=I-step+1;

    while (J>=1) and (A[J]>A[J+step] do begin  (* cycle of bubble exchanges
                                        in the sequence *)
       A[J]:=:A[J+step];
       J:=J-step;  (* decrease of index by step *)
      end; (* while *)

    end; (* for *)

    step:=step div 2;  (* half of step *)
  end; (* while *)
end; (* procedure *)
```

Analysis:

a) Shell sort is not a stable method. It works "in situ". It is simpler than Heapsort and contrary to Quick-sort it doesn't need recursion or usage of the stack.

There is no formula and experimental time values available for this implementation of method.

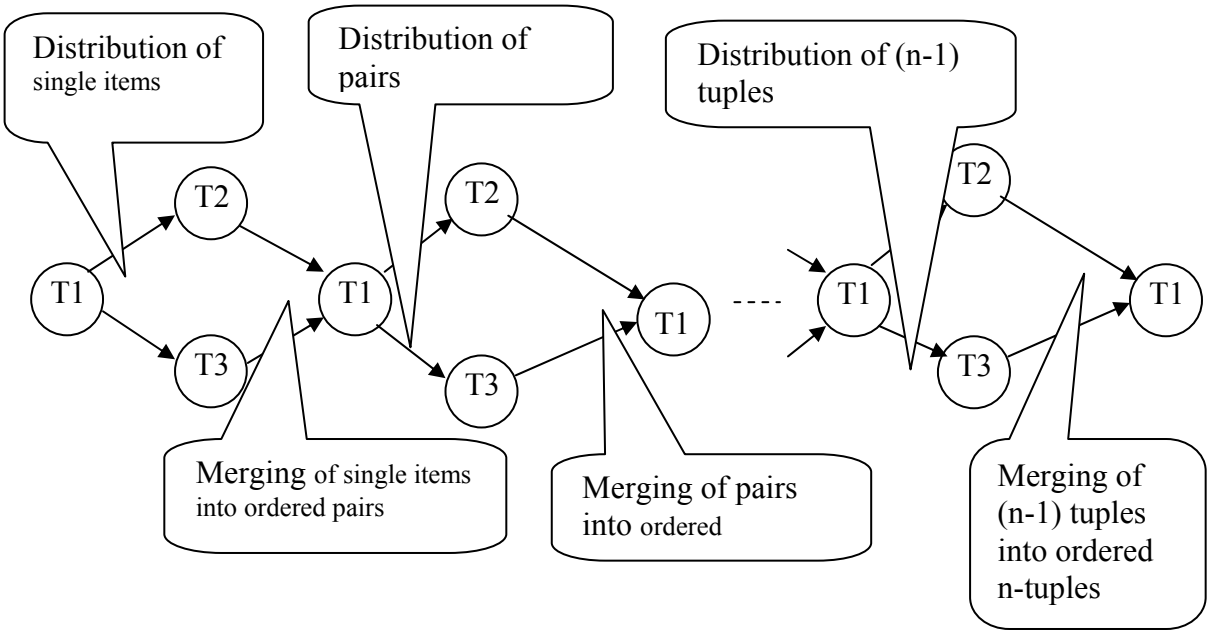
_

Lesson Thirteen

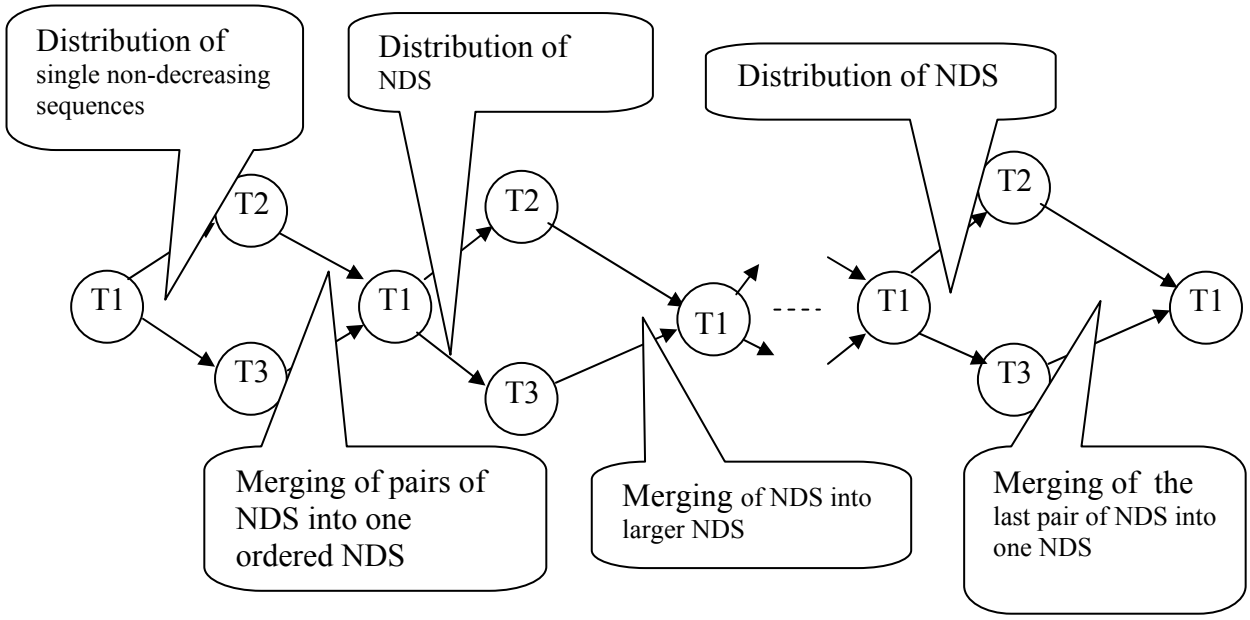
Lecture: Sorting - Ordering III.

13 Merging - Sorting of files.

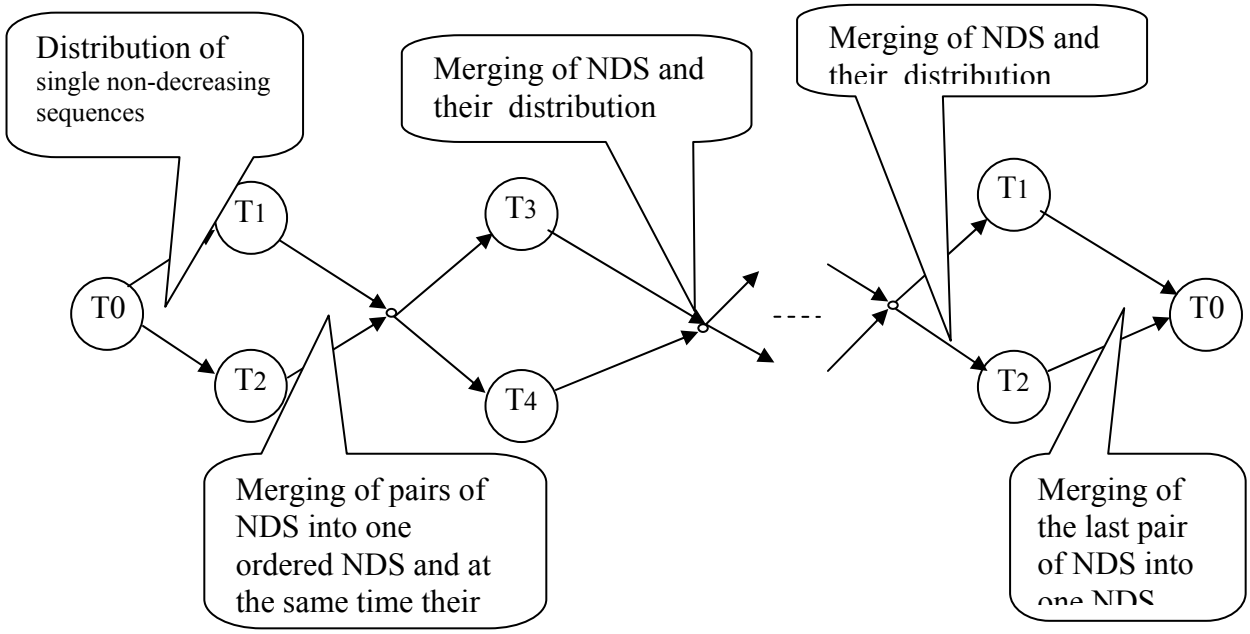
13.1 Straight merging (three tapes method)



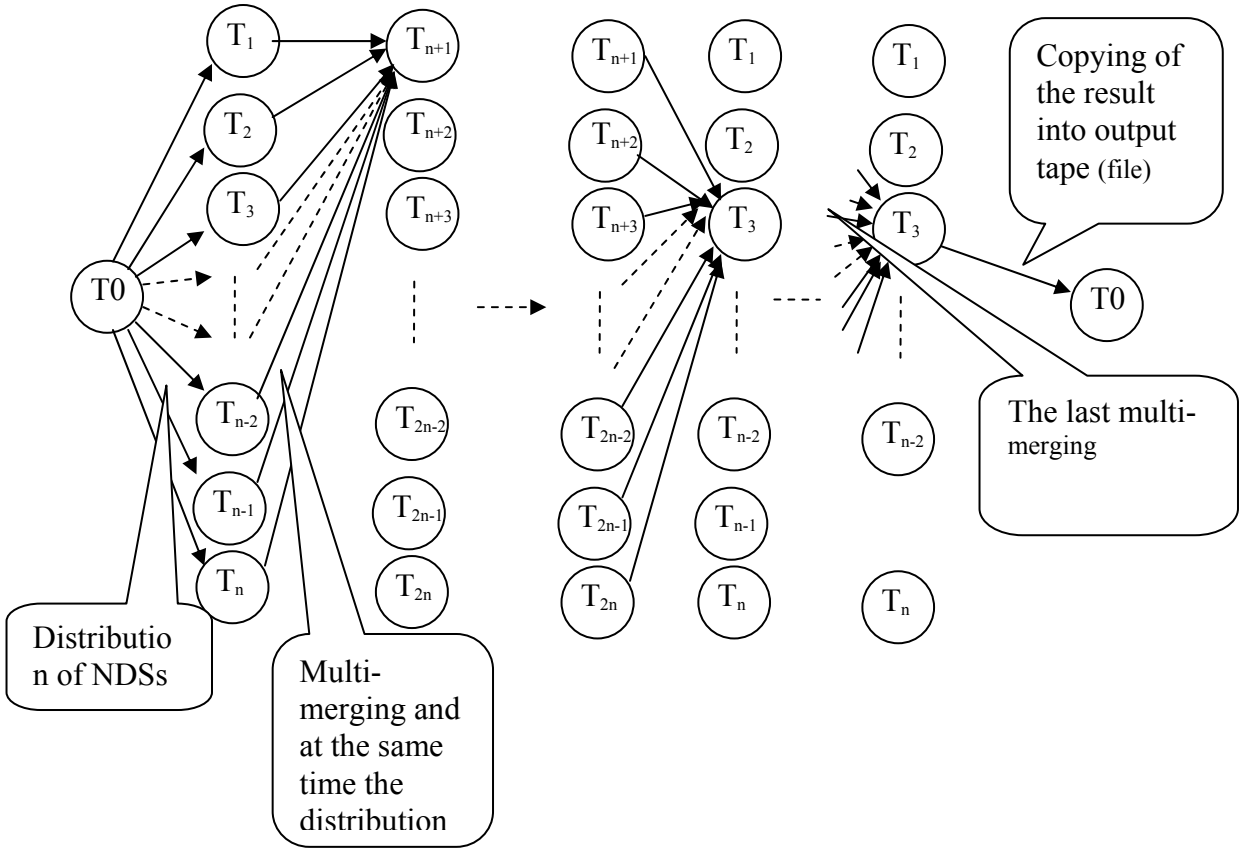
3.2 Natural merging (three tapes method)



13.3 Natural merging (four tapes method)



13.4 Multiway balanced merging.



13.5 Examples of basic principles of merging:

13.5.1 Procedures for merging of two sequential structures in the Pascal language

13.5.1.1. Merging procedure for two files with the use of access variable

```
procedure MergeFiles1(var F1,F2,F3:TFile);
(* Procedure uses access variable to file, according to ASCII Pascal
   rules. Ordered files F1 and F2 are merged to file F3. Not possible in
   TURBO and BORLADN Pascal !!!
*)

var Temp:TypElemOfFile; (* Base type of files *)
begin
  reset(F1); reset(F2); rewrite(F3);

  while not eof(F1) and not eof(F2) do
    begin
      if F1^ < F2^   (* F1^ and F2^ are access variables to files F1
                        resp F2 *)
      then begin
             read(F1,Temp);
             write(F3,Temp)
           end
       else begin
              read(F2,Temp);
              write(F3,Temp)
            end;  (* The cycle finishes by exhausting of one of the two
                     files *)
   (* end of while *)

  while not eof(F1) do  (* Adding of the rest of not-empty file to the
                           result file *)
  begin
    read(F1,Temp);
    write(F3,Temp)
  end;

  while not eof(F2) do  (* Adding of the rest of not-empty file to the
                           result file *)
  begin
    read(F2,Temp);
    write(F3,Temp)
  end
end.
```

13.5.1.2.Procedure for merging of two files without use of access variable

```
procedure MergeFiles2(var F1,F2,F3:TFile);
(* Procedure uses only predicate  eof and statements read and write *)

var A,B: TElemOfFile  (* Temp variable of the base type *)
begin
  reset(F1); reset(F2); rewrite(F3);
  if not eof(F1) and not eof(F2)
  then                    (* both files are not empty *)
    begin
      read(F1,A); read(F2,B);
      repeat
        if A<B
        then begin write(F3,A);
                  if not eof(F1) then read(F1,A)
             end
        else begin write(F3,B);
                  if not eof(F2) then read(F2,B)
             end;
      until eof(F1) or eof(F2);
        (* One file is empty, one element of the econd file is not
           writed *)

      if eof(F1)
      then write(F3,B)   (* Not writed was B *)
      else write(F3,A);  (* Not writed was A *)
    end;   (* One and/or both files are empty, all read is written *)


  while not eof(F1) do  (* writing of not empty file F1 *)
  begin
    read(F1,A);
    write(F3,A)
  end;

  while not eof(F2) do (* writing of not empty file F2 *)
  begin
    read(F2,B);
    write(F3,B)
  end;

       end
end (* procedure *)
```

13.5.1.3 Procerure for merging of two lists

```
procedure MergeLists(L1,L2:TList;var L3:TList);
var TempPoint  (* Temporary pointer moving along the list *)
             :TList;
begin
  new(L3);     (* creating the headings *)
  TempPoint:=L3;
  while (L1<>nil) and (L2<>nil) do begin    (* cycle of merging *)
  if L1^.Data < L2^.Data
  then begin
        TempPoint^.RPtr:=L1;
        TempPoint:=L1;
        L1:=L1^.RPtr
       end
  else begin
        TempPoint^.RPtr:=L2;
        TempPoint:=L2;
        L2:=L2^.RPtr
       end
  end; (* while *)
(* Joining of the not-empty list *)

 if L1=nil
 then TempPoint^.RPtr:=L2
 else TempPoint^.RPtr:=L1
 (* Destroying the heading *)
 TempPoint:=L3;
 L3:=L3^.RPtr;
 dispose(TempPoint)
end;
```

13.5.1.4 Procedure for merging of two lists used for List-Merge-sort

```
type
  TArr=array[1..Size] of integer;
  TArrPtr=array[1..Size] of intger;

procedure MergeLists(A:TArr;var AP TArrPtr; si1,si2:integer; var
                  di:integer);

(* A ... Sorted array; AP ... Array of pointers for linked lists;
   indexes of si1,si2... sources lists 1 and 2; index of destination
   list; indexes serve as pointers *)

var

begin
  if A[AP1[si1]] <= A[AP2[si2]]  (* establishing of the begin of
```

```
                              resulting list *)
  then begin
    di:=si1;
    si1:=AP[si1]
  end else begin
    di:=si2;
    si2:=AP[si2];
  TmpInd:=di;  (* Temporary index is set to the first element of
                  resulting list *)
  while (si1<>0) and (si2<>0) do begin
    if A[AP1[si1]] <= A[AP2[si2]]
    then begin      (* First item of the si1 list is less and added to
                       resulting list *)
      AP[TmpInd]:=si1;
      si1:=AP[si1];
    end else begin    (* First item of the si2 list is less and added to
                       resulting list *)
      AP[TmpInd:=si2;
      si2:=AP[si2]
    end; (* if *)
    TmpInd:=AP[TmpInd];   (* shifting the pointer along resulting list
                             by one item *)
  end; (* while *)

 (* Joining of remaining non-empty list *)
  if si1=0
  then AP[TmpInd]:=si2
  else AP[TmpInd]:=si1;
end;
```

13.5.2 Multi-merging

Comment: Algorithms are not debugged. They expose the main principles. In the text, the procedures SiftDown and MakeHeap, which are explained at the Heap-sort section are not introduced

13.5.2.1 Algorithms for multi-merging of files with the use of heap.

The problem:
In the array, there are N files containing one non-decreasing sequence of integer values. Merge them to one output non-decreasing sequence.

Principle:

N input files are contained in the array of N items.

Create the temporary array TmpArr. Every line of the array contains two components: Index (Number of the file - NoOfFile), which points to the array of input files and the key (InKey), into which the first element of the non-decreasing sequence - the pilot element - of the file is read.
The index - NoOfFile is necessary as the type file is illegal to assign, and consequently it is not possible to move it or change (swap) it. It is possible to move and swap the index-pointers.

The TepmArr is initialized by the ordinal numbers of files from 1 to N, and the first value from the corresponding file is in the key-item..

Procedure MakeHeap creates the heap from the array according the value of keys.

The variable MaxSeqAct traces the number of active sequences; it is initialized to N (number of files).

In the cycle, which is executed while the number of active sequences is non-zero, the simple action is done:

The key from the first line is written to the output file. While the corresponding file is not empty, a new element from the corresponding file is read to the key. Correspondency is given by the NoOfFile at the first line. If the corresponding file is empty, the first line is rewritten by the line MaxSeqAct, and MaxSeqAct is decreased by one. If the decreased value is equal to zero, the cycle ends and in the output file is one non-decreasing sequence merged from N non-decreasing sequences.

```
type
  TFile=file of TKey; (* type of merged file;
                      premise: every file consists from
                      one non-decreasing sequence only!  *)

  TFileArr=array[1..N] of TFile; (* array of files which are to be merged*)



procedure HeapMerge(var FileArr:TFileArr; var DestFile:TFile);
(* Procedure merges N non-decreasing sequences from N files into
   one output file *)

type
  TItem=record
          NoOfFile:integer; (* ord. number of the file *)
          InKey:TKey;     (* Input element for read key *)
        end;
```

```
  TTempArr=array[1..N] of TItem;

var
     TmpArr:TTempArr;  (* Temporary array of number of files and keys *)
     MaxSeqAct:integer; (* The number of active sequences *)
     MaxFilAct:integer; (* The number of active files *)
     i,j: integer;  (* counters *)


begin

  MaxSeqAct:=N;
  MaxFilAct:=N;

  for i:=1 to N do begin    (* inicialization of temp. array *)
   with TmpArr[i] do begin
    Read(FileArr[i],InKey); (* reading of the first element of sequence
                         from i-th file *)
    NoOfFile:=i    (* setting of the No of the file *)
  end;

  MakeHeap(MaxFilAct);

  while MaxSeqAct<>0 do begin
   Write(DestFile; TmpArr[1].Inkey);  (* write the top of heap to
                                output file *)
   if not eof(TmpArr[1].NoOfFile
   then Read(FileArr[TmpArr[1].NoOfFile]);   (* Read the new element
                               from the file to te top of heap *)
   else begin
    TmpArr[1]:=TmpArr[MaxSeqAct];  (* the last sequence to the top of
                                heap *)
    MaxSeqAct:=MaxSeqAct-1;      (* decreasing the No of sequences *)
   end;
   Sift(MaxSeqAct);  (* re-establishing the Heap *)
  end; (* while *)
end; (* procedure *)
```

13.5.2.2 Repeated multi-merging

The task: There are N files containing many non-decreasing sequences. Merge N first non-decreasing sequences form N files into one output sequence and write it to the output file.

Principle:

Principle of solution is continuing of the previous example.

The end of non-decreasing sequence is given by two neighbouring items, where the predecessor is greater than successor. As the file contains more non-decreasing sequences, another indicator is end of file mark (eof).

Example:

in the file: | ...5 8 6 4 3|

item 6 is the last item of NDS because the following item (4) is less than (6). Item 3 is the last item of the following NDS as it is the latest item of the file.

Comments on the following program:

- N output files are contained in the input array of files. Resulting non-decreasing sequences are placed to the output files in cyclic manner.
- For the end-test of non-decreasing sequence the auxiliary variable is used (EndSeq). It keeps the last value written to output file. If this value is greater than the first pilot value, it indicates the end of non-decreasing sequence and pilot value serves as the first value for the next merging. It should be swapped with the last active sequence, and the total number of active sequences is decreased by one.
- If the file is empty, it indicates the end of non-decreasing sequence too, but simultaneously the number of active files (MaxFileAct) is decreased by one. In this case the last active item of the array is swapped with the first line of array (top of heap). The MaxSeqAct and MaxFileAct are decreased by one.
- The inner cycle merges one N-tuple of non-decreasing sequences. The cycle ends if MaxSeqAct=0. Before the inner cycle the heap is established and the value MaxSeqAvct is set to MaxFileAct. After finishing the cycle, there is the actualization of the cycle index pointing to output array.
- Outer cycle ends if the MaxFileAct=0.

```
procedure HeapMerge(var SourceFileArr,DestFileArr:TFileArr);
(* Procedure merges non-decreasing sequences from N files, and from every N-tuple of non-
decreasing sequences writes one resulting non-decreasing sequence to one output file in
cyclic manner *)

type
  TItem=record
          NoOfFile:integer; (* Ordinal number of the file *)
          InKey:TKey;     (* Input variable for the read key *)
        end;
  TTempArr=array[1..N] of TItem;

var
```

```
    TmpArr:TTempArr;  (* Aux. array of No. of files and keys *)
    MaxSeqAct:integer; (* No. of active non-dec. sequences *)
    MaxFilAct:integer; (* No. of active files *)
    i,j: integer;  (* counters *)
    EndSeq:integer; (* Aux. variable for the test of non-decreasing
                sequence *)

begin


 for i:=1 to N do begin    (* inicialization of TmpArr *)
  with TmpArr[i] do begin
   Read(SourceFileArr[i],InKey; (* reading of the first element of non-
                        dec. seq.  from the i-th file *)
   NoOfFile:=i    (* setting the No. of file *)
 end;

 MaxFilAct:=N;  (* Inicialization of No. of active files *)

 j:=1; (* Inicialization of the index to output array *)

 while MaxFilAct <> 0 do begin  (* outer cycle, given by the largest
              number of non-decreasing sequences in some file *)

  MaxSeqAct:=MaxFileAct;  (* inicialization  of MaxSeqAct *)

  MakeHeap(MaxFilAct); (* creating the heap from lines of all active
                    lines of all active files *)

  while MaxSeqAct<>0 do begin (* inner cycle for creating one output
non-decreasing sequence from MaxFileAct of non-decreasing sequences *)

    Write(DestFileArr[j]; TmpArr[1].Inkey);  (* writing the top of
                                heap to output file *)

    EndSeq:=TmpArr[1].Inkey;  (* setting the test variable *)

    if not eof(SourceFileArr[TmpArr[1].NoOfFile]
    then begin Read(SourceFileArr[TmpArr[1].NoOfFile],TmpArr[1].InKey);
        (* reading the new pilot to the top of heap *)
      if EndSeq > TmpArr[1].InKey
      then begin (* this is the end ofnon-decreasing sequence, as
              pilot is less then its predecessor *)

        TmpArr[1]:=:TmpArr[MaxSeqAct];  (* exchange the pointers of
                                top sequence and the pointer
                                of the last active sequence *)
        MaxSeqAct:=MaxSeqAct-1;       (* decreasing the No. of active
                                 sequences *)
```

```
        Sift(MaxSeqAct);  (* re-establishing the heap *)
       end
     else begin (* the end of file and the end of non-dec. sequence too *)

       TmpArr[1]:=TmpArr[MaxFilAct];
       TmpArr[1]:=:TmpArr[MaxSecAct];
       (* the statements rewrite the first line by  MaxFilAct line,
        and exchange first line with  MaxFilAct line  *)


       MaxSeqAct:=MaxSeqAct-1;  (* Decreasing No. of active sequences *)
       MaxFilAct:=MaxFilAct-1;  (* Decreasing No. of active files *)
     end;
   end (*  End of inner cycle while *)
   j:=j+1;  (* cyclic actualization of the index to output array *)
   if j<>N then j:=1;
 end; (* End of outer cycle while *)

end; (* end of procedure *)
```

–