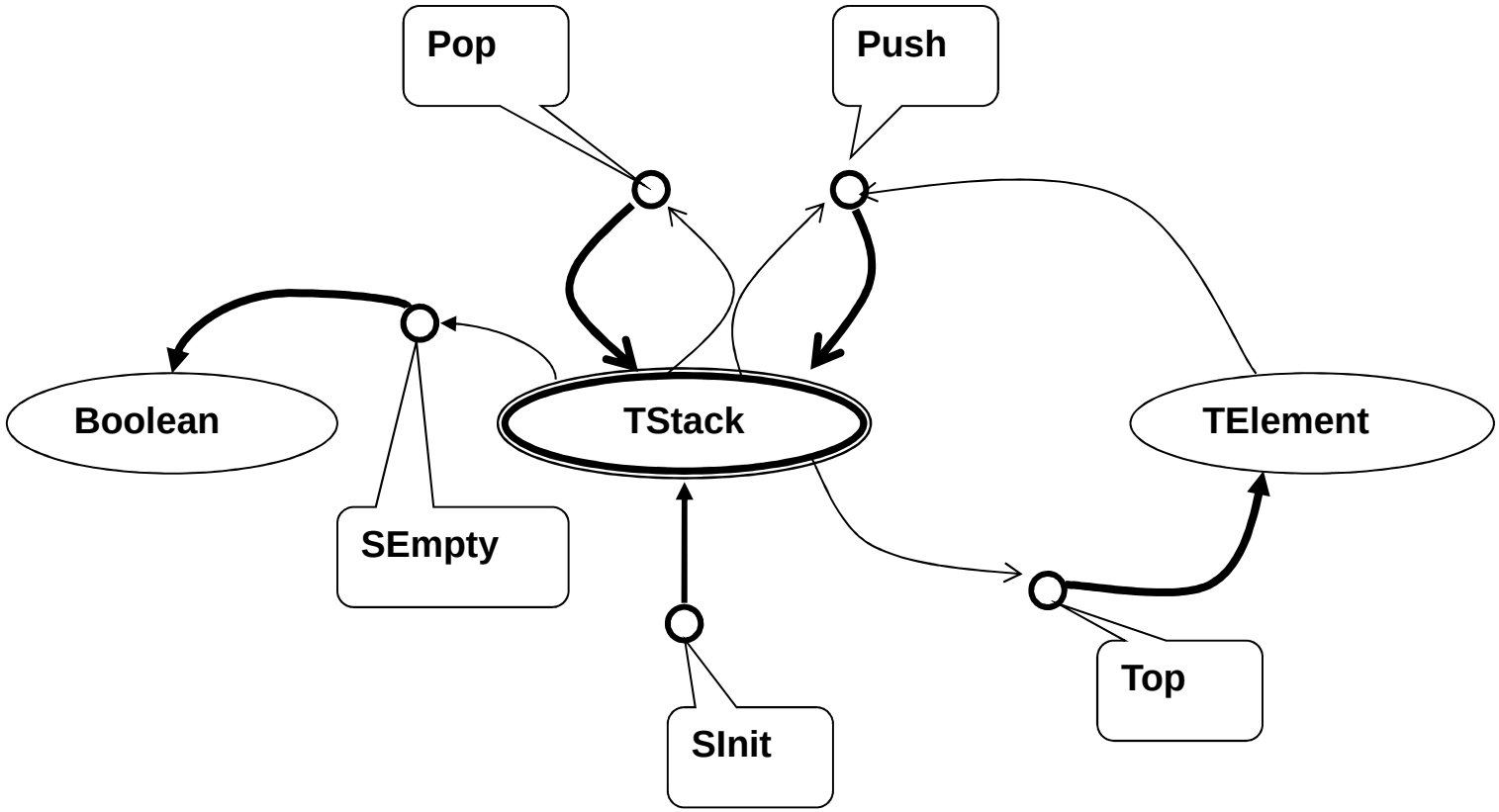
5 Přednáška IAL

- ADT zásobník. Užití zásobníku: algoritmy s návratem, rekurze, bloková struktura a dynamické přidělování paměti, převod z infixové do postfixové notace se zásobníkem, vyčíslování aritmetických výrazů v postfixové notaci se zásobníkem.
- ADT fronta a její užití. Implementace zásobníku a fronty.
- ADT pole, jedno a vícedimenzionální, statické a dynamické, přístupové metody k prvku pole (mapovací funkce, informační vektor/záznam). Prostorově úsporné metody implementace některých polí - trojúhelníková matice, pole s nestejně dlouhými řádky, řídké pole.
- ADT množina.

Zásobník - Stack

- Homogenní, dynamický, lineární datový typ
- LIFO – Last In – First Out
- Použití:
  - Reverzuje pořadí
  - Převod infixové notace na postfixovou
  - Konstrukce rekurzívních podprogramů
  - Konstrukce blokové struktury

Diagram signatury ADT Stack



SInit(S) (nebo StackInit) Nechť vznikne prázdný zásobník

Push(S,El) Vložení prvku El na vrchol zásobníku

SEmpty(S) Predikát, který vrací hodnotu
true je-li zásobník prázdný a false v jiném případě

Top(S,El) Čtení hodnoty prvku na vrcholu zásobníku.
Obsah zásobníku beze změny. Pro prázdný zásobník dojde k chybě. Operace Top je vždy ošetřena testem na neprázdnost zásobníku:

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

Pop(S) Zrušení hodnoty prvku na vrcholu zásobníku.
V případě, že je zásobník je prázdný, bez účinku.

Axiomy sémantiky ADT Stack

1. Pop(StackInit(S)) = StackInit(S)
2. Pop(Push(S,El)) = S
3. Top(StackInit(S)) = error
4. Top(Push(S,El)) = El
5. SEmpty(StackInit(S)) = true
6. SEmpty(Push(S,El)) = false

Infixová, prefixová (polská), postfixová (obrácená polská) notace (Jan Lukasiewić)

- Infixová ⇨ a + b
- prefixová ⇨ + a b

prefixová notace připomíná zápis funkce..?

function ADD(a,b:integer):integer

- postfixová ⇨ a b +
- x + y= ⇨ x y + =
- (a+b)*(c-d)/(e+f)*(g-h)=⇨ab+cd-*ef+/gh-*=

Výhody postfixové notace

- Neobsahuje závorky
- Snadné vyčíslení – operace se provádějí v pořadí operátorů v řetězci

Výraz: a b + c d - * e f + / g h - * =

se zpracuje jako by měl tvar:

(((a b +) (c d -) *) (e f +) /) (g h -) * =

Algoritmus vyčíslení postfixového výrazu

- Zpracovávej řetězec zleva doprava
- Je-li zpracovávaným prvkem operand, vlož ho do zásobníku
- Je-li zpracovávaným prvkem operátor, vyjmi ze zásobníků tolik operandů, kolika-adický je operátor (pro dyadické operátory dva operandy), proveď danou operaci a výsledek uloží na vrchol zásobníku
- Je-li zpracovávaným prvkem omezovač ‘=‘, je výsledek na vrcholu zásobníku

Úloha k domácímu procvičení

Předpokládejte, že je definován ADT TStack čísel integer a deklarována globální proměnná S tohoto typu, nad kterou smíte aplikovat všechny operace nad TStack. Je dán řetězec znaků, obsahující číslice, operátory ‘+’, ‘-’, ‘*’ a ‘/’, které představují operace nad typem integer a ukončovací znak (omezovač) ‘=’, kterým je řetězec zakončen. Číslice představují jednomístná celá čísla. Předpokládejte, že řetězec představuje syntakticky správný aritmetický výraz.

Napište proceduru (funkci), která má na vstupu řetězec s postfixovým výrazem a která vrátí hodnotu vyčísleného výrazu.

Převod infixové notace na postfixovou s použitím zásobníku

1. Zpracovávej vstupní řetězec položku po položce zleva doprava a vytvářej postupně výstupní řetězec.
2. Je-li zpracovávanou položkou operand, přidej ho na konec vznikajícího výstupního řetězce.
3. Je-li zpracovávanou položkou levá závorka, vlož ji na vrchol zásobníku.

4. Je-li zpracovávanou položkou operátor, pak ho na vrchol zásobníku vlož v případě, že:
   - » **zásobník je prázdný**
   - » **na vrcholu zásobníku je levá závorka**
   - » **na vrcholu zásobníku je operátor s nižší prioritou**

- Je-li na vrcholu zásobníku operátor s vyšší nebo shodnou prioritou, odstraň ho, vlož ho na konec výstupního řetězce a opakuj krok 4, až se ti podaří operátor vložit na vrchol

5. Je-li zpracovávanou položkou pravá závorka, odebírej z vrcholu položky a dávej je na konec výstupního řetězce až narazíš na levou závorku. Tím je pár závorek zpracován.
6. Je-li zpracovávanou položkou omezovač ‘=’, pak postupně odstraňuj prvky z vrcholu zásobníku a přidávej je na konec řetězce, až zásobník zcela vyprázdníš a na konec přidej rovnítko.

Implementace ADT Stack

Většina ADT bude implementována buď souvislé paměti po sobě jdoucích prvků (pole) nebo v paměti zřetězených položek (seznam). V prvním případě je datová struktura většinou “pseudodynamická” – tedy dynamická tak dlouho, dokud nevyčerpá přidělený prostor (deklarované pole).

Pro implementaci zásobníku polem tedy účelné dodefinovat predikát, která bude signalizovat naplnění vyhraženého prostoru a tedy zákaz vkládání dalšího prvku do zásobníku. Nechť se tato operace jmenuje SFull(S), a v případě, že je zásobník plný, bude tato funkce vracet hodnotu true, jinak bude vracet hodnotu false.

Nechť jsou pro účely implementace definovány následující typy:

```
const SMax=1000;   (* Maximální kapacita zásobníku *)
type
  TStack=record
      SPole=array[1..SMax] of TElem; (* pole pro zásobník *)
      STop: 0..SMax     (* index ukazatele na vrchol *)
  end;
```

Pak operace inicializace bude mít podobu:

```
procedure SInit(var S:TStack);
begin
  S.STop:=0
end;
```

```
procedure Push(var S:TStack; El:TElem);
begin
  with S do begin
      STop:=STop + 1
     SPole[STop]:= El;
   end (* with *)
end;
```

Pozn: Operace nevrací hodnotu

```
procedure Pop (var S:TStack);
begin
  with S do begin
     if STop > 0
     then STop:=STop - 1
  end (* with *)
end;
```

```
procedure Top (S:TStack; var El:TElem);

(* Procedura ukončí program zásadní chybou,
je-li při vyvolání zásobník prázdný, protože index Stop
je nulový a tudíž mimo definovaný rozsah pole.
Podmínku však do procedury nevkládáme. Podmínka
ošetřující neprázdnost zásobníku je povinnou součástí
použití operace. Ponechme na programátorovi, co udělá
v případě pokusu o čtení z prázdného zásobníku.
Proceduru lze zapsat jako funkci v případě,
že to dovoluje typ TElem *)

begin
   El:= S.SPole[S.STop]
end;
```

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

```
function SFull(S:TStack):Boolean;
begin
   SFull:= S.STop=SMax
end;
```

Implementace jednosměrným seznamem

```
type
   TUk = ^TPrvek;

   TPrvek = record
       Data:Telem;
       UkDalsi:TUk
   end; (* TPrvek *)


   TStack=record
      TopUk:TUk;
      ...
    end; (* TStack*)
```

```
procedeure SInit(var  S:TStack);
begin
   S.TopUk:=nil
   ...
end;

procedure Push(var S:TStack; El:TElem);
var
   PomUk:TUk;
begin
   new(PomUk);
   PomUk^.Data:=El;
   PomUk^.UkDalsi:=S.TopUk;
   S.TopUk:=PomUk
end;
```

```
procedure Pop(var S:TStack);
var
   PomUk:TUk;
begin
    if S.TopUk<>nil
    then begin
       PomUk:=S.TopUk;
       S.TopUk:=S.TopUk^.UkDalsi;
       dispose(PomUk)
   end (* if *)
end;
```

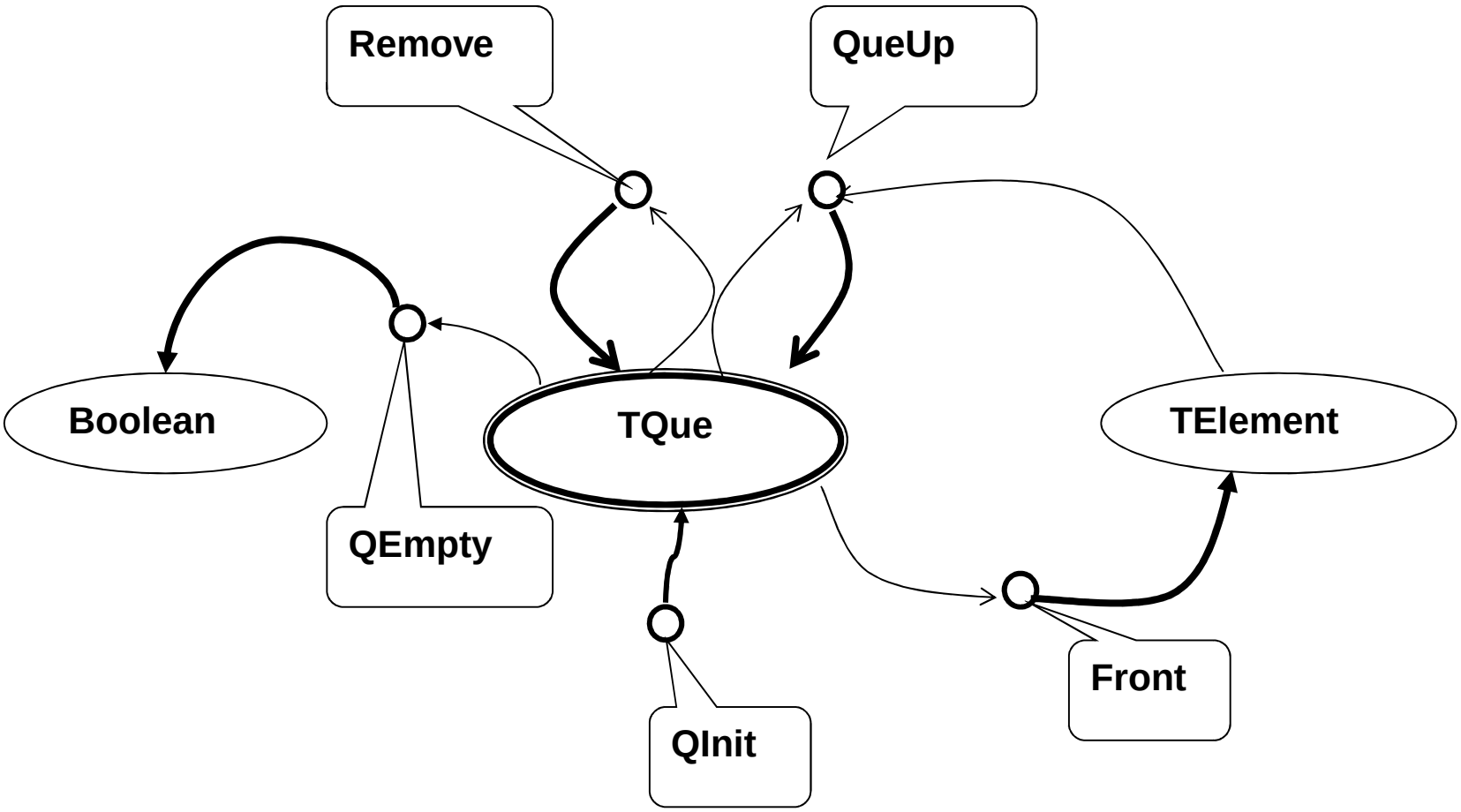
```
procedure Top(S.TStack; var El:TElem);
```
(* V případě prázdného zásobníku je ukazatel nilový a v při jeho referenci dojde k chybě. Podmínku prázdnosti však ponecháme vně procedury *)
```
begin
   El:=S.TopUk^.Data
end;
```

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

ADT Fronta - Queue

- Dynamická, homogenní a lineární struktura.
- FIFO : First In First Out
- Na jednom konci přidává a na druhém (říká se mu začátek !!!) čte a odebírá – obsluhuje.
- Teorie front (teorie hromadné obsluhy) – historie s vodovodními kohoutky

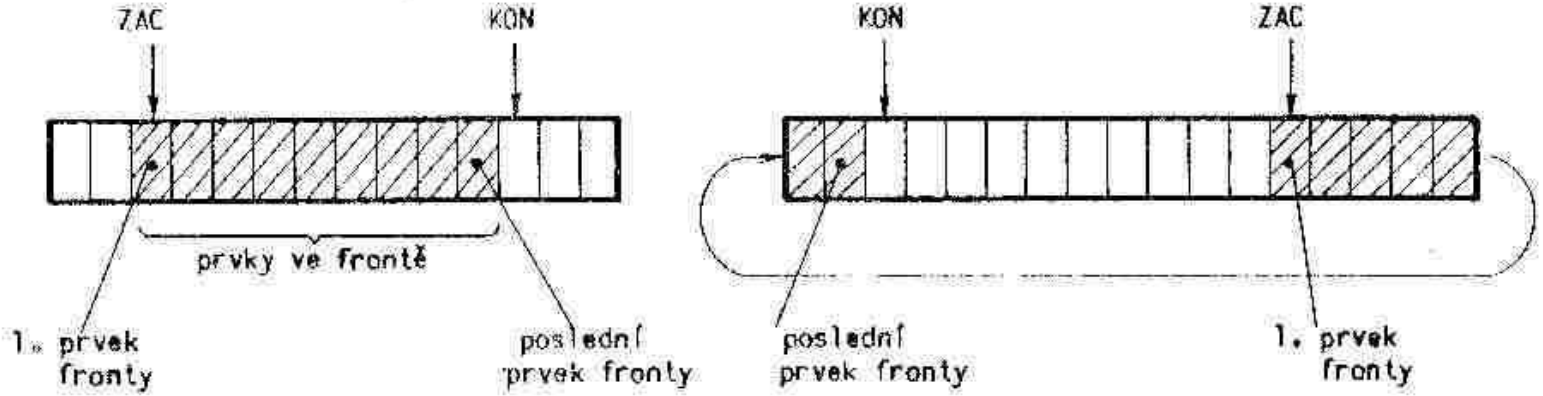
Diagram signatury ADT Fronta - TQUEUE



Sémantiku fronty lze vyjádřit
následujícími 8 axiomy:

1. QEmpty(QInit(Q)) = true
2. QEmpty(QueUp(Q,El)) = false
3. Front(QueInit(Q)) = error
4. Front (QueUp(QInit(Q),El) = El
5. Front (Queup(Queup(QInit(Q),ElA),ElB)) = Front (QueUp(QInit(Q),ElA)
6. Remove(QInit(Q)) = QInit(Q)
7. Remove(QueUp(QInit(Q), El)) = QInit(Q)
8. Remove(QueUp(QueUp(QInit(Q),ElB), ElA)) = QueUp(Remove(QueUp(QInit(Q), ElB)), ElA)

Stavy fronty



Obr. 5.33. Běžné stavy fronty

Obr. 5.34. Stavy ukazatelů při plné frontě

Implementace fronty polem

```
const
  QMax =1000; (* Fronta má kapacitu QMax - 1 prvků !! *)
type
   TQueue : record
       QPole: array [1..QMax] of TEl;
       QZac,QKon:  integer;
   end;

procedure QInit(var Q:TQueue);
begin
  with Q do begin
    QZac:=1;
    QKon:=1
  end (* with *)
end;
```

```
procedure QueUp (var Q:TQeue; El:TElem);
begin
   Q.QPole[Q.QKon] := El;
   Q.QKon:=Q.QKon + 1;
    if Q.QKon > QMax
    then Q.QKon := 1;(* Ošetření kruhovosti seznamu *)
end;

procedure Remove (var Q:TQueue);
begin
   if Q.QZac<>Q.QKon
   then begin
      Q.QZac:=Q.Qzac + 1;
       if Q.QZac > QMax
       then Q.QZac:=1; (* Ošetření kruhovosti pole*)
   end (* if *)
end;
```

```
procedure Front (Q.TQueue; var El:TElem);
```
(* Procedura způsobí chybu v případě čtení z prázdné fronty *)
```
begin
   El:=   Q.QPole[Q.QZac]
end;
function QEmpty(Q:TQueue): Boolean;
begin
   QEmpty:= Q.Zac=Q.Kon
end;
```

Soubor lze doplnit o predikát QFull takto:
```
function QFull(Q: TQueue): Boolean;
begin
   QFull:= (Q.Zac=1) and (Q.Kon=QMax) or
((Q.Zac - 1) = Q.Kon)
end;
```

Pozn. Ošetření kruhovosti pole lze zajistit také operací mod QMax. Druhým argumentem operace mod je vždy počet Poc prvků pole. Operace mod dává výsledky v intervalu 0..(Poc-1)

S ohledem na skutečnost, že pole začíná indexem 1, je nutné tuto jedničku přičíst. Pole má QMax prvků. Lze tedy inkrementaci ukazatele zajistit příkazem

Q.Zac:= Q.Zac mod QMax +1;

t zn. když je QMax = 100 má pole 100 prvků, pak
když staré Q.Zac = 99 pak po inkrementaci je:
Q.Zac= to 99 mod 100 +1 = 100
když staré Q.Zac = 100 pak po inkrementaci je:
Q.Zac= 100 mod 100 +1 = 1

Když je pole definováno intervalem 0..QMax, má (QMax+1) prvků a inkrementaci s kruhovým převodem je možno zapsat :

$$Q.Zac := (Q.Zac + 1) \bmod (QMax+1)$$

t.zn. když je QMax = 100 a pole má 101 prvků, pak
když staré Q.Zac = 99 pak po inkrementaci je

$$Q.Zac = (99 + 1) \bmod (100+1) = 100$$

když staré Q.Zac = 100 pak po inkrementaci je

$$Q.Zac = (100+1) \bmod (100 +1) = 0$$

Implementace fronty zřetězeným seznamem

```
type
   TUk = ^TPrvek;
   TPrvek = record
      Data: TElem;
      UkDalsi: TUk
   end;


   TQueue = record
      QZacUk,QKonUk:TUk;
   end;
```

```
procedure QInit(var Q:TQueue);
begin
   Q.QZacUk:=nil;
   Q.QKonUk:=nil
end;
```

```
procedure QueUp(var Q;TQueue; El:TElem);
var  PomUk:TUk;
begin
  new (PomUk);
  PomUk^.Data:= El;(*naplnění nového prvku*)
  PomUk^.UkDalsi:=nil;(*ukončení nového prvku*)
  if Q.ZacUk = nil
  then(* fronta je prázdná, vlož nový jako první a jediný*)
    Q.QZacUk:=PomUk
  else (* obsahuje nejméně jeden prvek, přidej na konec*)
    Q.QKonUk^.UkDalsi:=Pomuk;
  Q.QKonUk:=PomUk   (* korekce konce fronty *)
end;
```

```
procedure Front (Q;TQueue; var El: TElem);
begin
   El:=Q.QZac^.Data
end;

procedure Remove (Q:TQueue);
var
   PomUk: TUk;
begin
   if Q.QZac <> nil
   then  begin (* Fronta je neprázdná *)
      PomUk := Q.QZac;
      if Q.QZac = Q.QKon
      then Q.QKon:=nil; (* Zrušil se poslední a jediný *)
      Q.QZac:=Q.QZac^.UkDalsi;
      dispose (PomUk);
end;
```

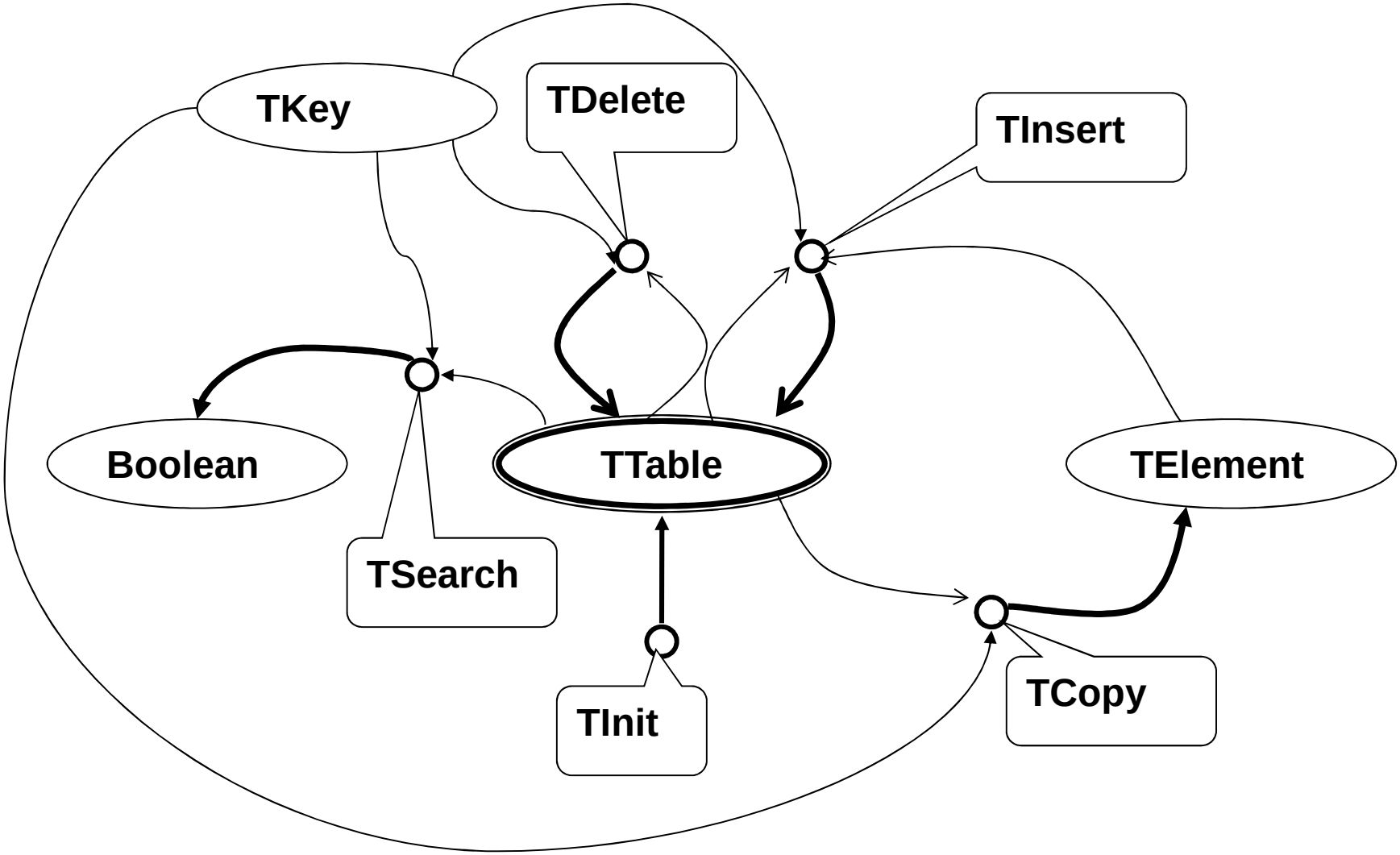
Oboustranně ukončená fronta DEQUE
– Double Ended Queue

- domino, kolej na seřazovacím nádraží aj.

Vyhledávací tabulka –
(Search table, Look-up table)

- Homogenní, obecně dynamická struktura
- Každá položka má zvláštní složku – klíč
- V tabulce s (ostrým) vyhledáváním je hodnota klíče jedinečná (neexistují dvě či více položek se stejnou hodnotou klíče)
- Tabulka je jako „kartotéka“, je to základ databází

Diagram signatury ADT „Vyhledávací tabulka“



Sémantika operací:

TInit (T) operace, která inicializuje (vytváří) prázdnou tabulku položek se složkami: klíč K typu Tklic a daty Data typu Tdata.

TInsert(T,K,Data) vložení položky se složkami K a Data do tabulky T. Pokud tabulka T již obsahuje položku s klíčem K dojde k přepisu datové složky Data novou hodnotou. Tato vlastnost se podobá činnosti v kartotéce, kdy při existenci staré karty se shodným klíčem se stará karta zahodí a vloží se nová (aktualizační sémantika operace TInsert).

TSearch(T,K) predikát, který vrací hodnotu true v případě, že v tabulce T existuje položka s klíčem K a hodnotu false v opačném případě

TDelete(T,K) operace rušení prvku s klíčem K v tabulce T. V případě, že prvek neexistuje má operace sémantiku prázdné operace

TCopy(T,K,El) operace, která vrací (čte) ve výstupním parametru El hodnotu datové složky položky s klíčem K. V případě, že položka s klíčem K v tabulce T neexistuje, dochází k zásadní chybě. Proto je povinností programátora ošetřit každý výskyt operace TCopy predikátem TSearch:

```
if TSearch(T,K)
then begin
     TCopy(T,K,El);
end;
```

Celá kapitola o vyhledávání bude o různých metodách implementace ADT vyhledávací tabulka

Pole - Array

Pole je homogenní struktura ortogonálního (pravoúhlého) typu. V Pascalu je pole statické.

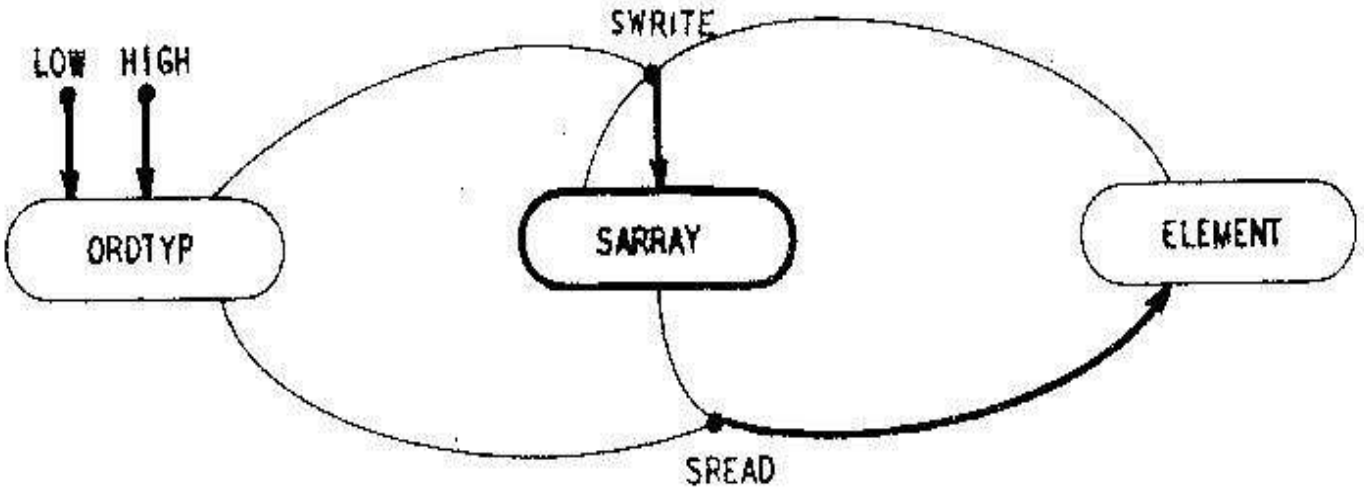
Pascal definuje obecně pole jako jednorozměrné. Pak n-rozměrné pole je jednorozměrné pole položek, jimiž jsou (n-1) rozměrná pole.

Hlavními operacemi nad polem jsou:

- „zápis hodnoty do prvku pole daného indexem“ - „aktivní přístup“
- „čtení“ hodnoty hodnoty prvku zadaného indexem – „pasivní přístup“

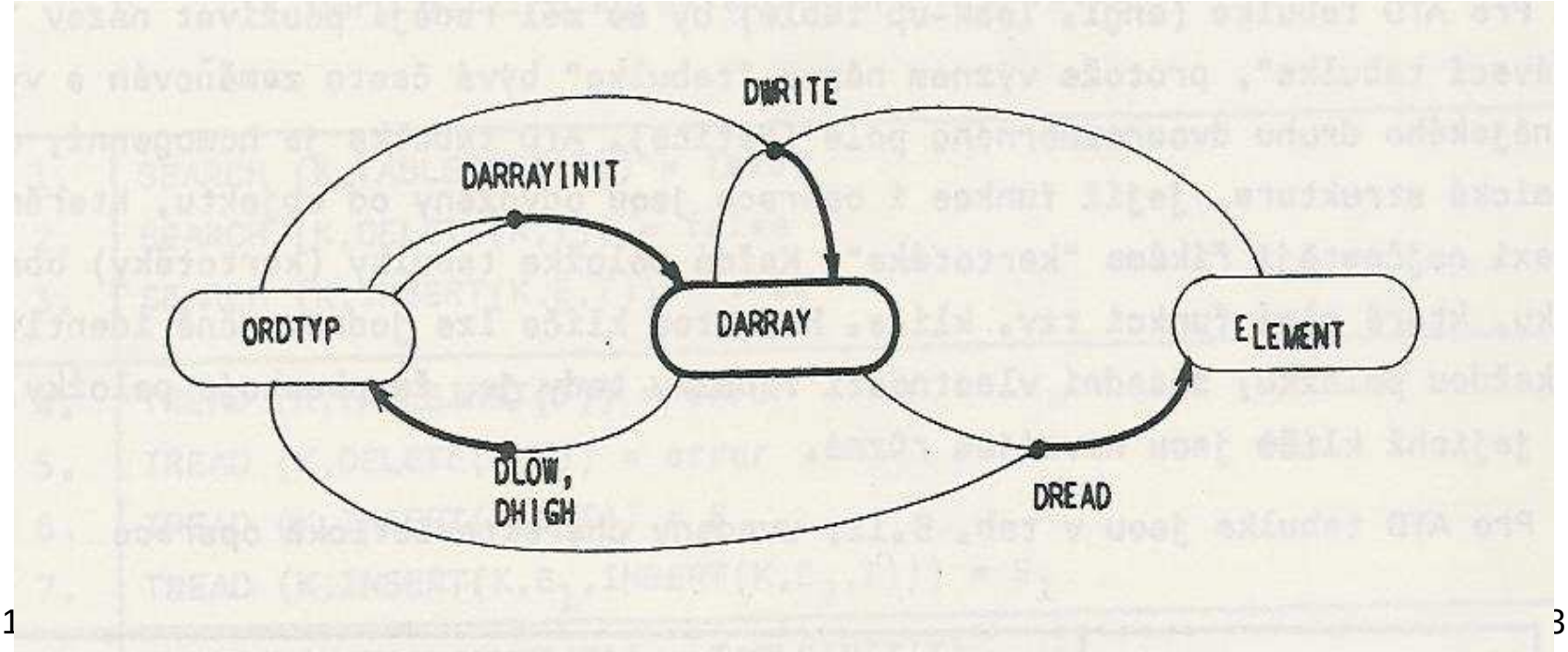
Statické pole

Diagram signatury statického pole SARRAY



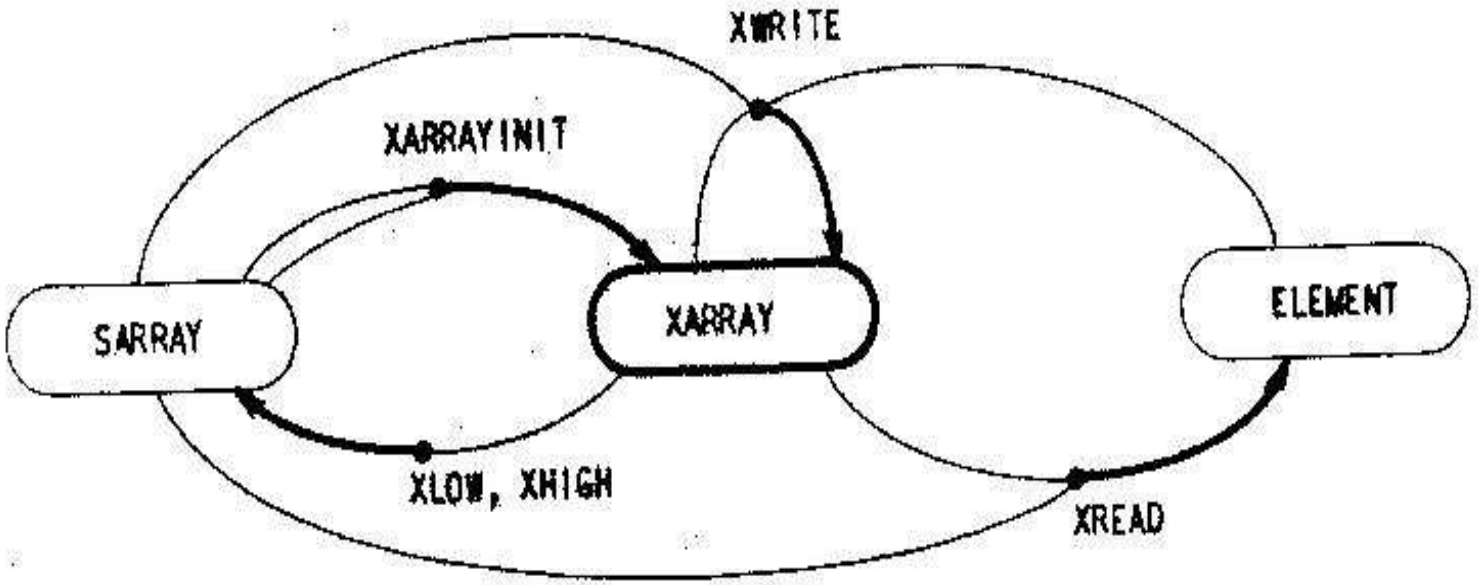
Dynamické pole

• Velikost dynamického pole je inicializovaná v průběhu chodu programu operací DARRAYINIT
• Aktuální velikost dimenze lze zjistit operacemi DLOW a DHIGH



Vícerozměrné statické pole

Indexy vytvářejí jednorozměrné pole SARRAY polí o dvou prvcích, (každá dvojice vymezuje rozsah jedné dimenze)



Mapovací funkce

Mapovací funkce zajišťuje zpřístupnění prvku pole.
Realizuje funkci selektoru.

Nechť je dáno k-rozměrné pole

B: array [low1 .. high1, low2 .. high2, … , lowk .. highk]
of TElement;

Pak prvek tohoto pole, jehož zápis má tvar
B[j1 ,j2 , … , jk]
bude zobrazen (mapován) do jednorozměrného pole A s indexem, jehož hodnotu definuje výraz mapovací funkce:

$$1 + \sum_{m=1}^{m=k} \left(j_m - low_m \right) * D_m$$

Pak prvek B[j_1, j_2..., j_k] bude v paměti umístěn (bude mapován) do vektoru A podle následujícího vztahu

$$B[j_1, j_2, \ldots, j_k] \rightarrow A\left[1 + \sum_{m=1}^{m=k} (j_m - low_m) * D_m\right]$$

kde

$D_1 = 1$

$D_m = (high_m - low_m + 1) * D_{m-1}$

Hodnota mapovací funkce se musí vyčíslovat v průběhu výpočtu při každé referenci (odkazu) na indexovanou proměnnou. Jak je z tvaru výrazu vidět, může být vyčíslení, zejména u vícedimenzionálních polí, časově náročnější.

Trojúhelníková matice

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

Lze zaujmout prostor pro pouze *(N/2)* * *(N+1)* položek místo *N*N* položek.
Použije se mapovací funkce

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

Matice s nestejně dlouhými řádky

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}								

Přístupový vektor PV má hodnoty: 0 2 4 7 16 17
Mapovací funkce: a[i,j] -> v[PV[i]+j]

v:

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}

Řídké pole

- Pole, v němž má převážná většina prvků stejnou (dominantní) hodnotu (např. 0).
- Řídké pole lze implementovat s cílem ušetřit paměťové nároky za cenu přístupového času (access time) k prvkům pole
- Implementace s použitím vyhledávací tabulky, v níž prvky pole tvoří hodnotu položek tabulky a index(y) tvoří klíč položky tabulky

- InitArr(Arr) -> TInit(T)
- ReadArr(Arr,Ind,El) -> if Search(T,Ind)
  then El:=TCopy(T,Ind)
  else El:= dominant value;
- WriteArr(Arr,Ind,El) -> if El=dominant value
  then TDelete(T,Ind)
  else TInsert(T,Ind,El)

Pro dvou(více)rozměrná pole lze použít mapovací funkce, která mapuje dvojici (n-tici) indexů do jednoho indexu

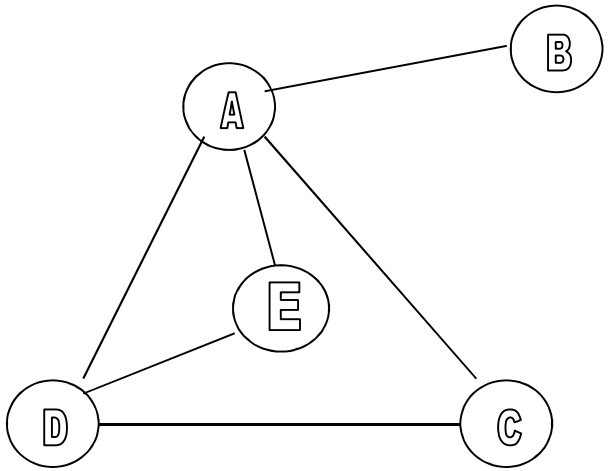
Graf a jeho implementace

- Graf je definován trojicí G=(N,E,I)

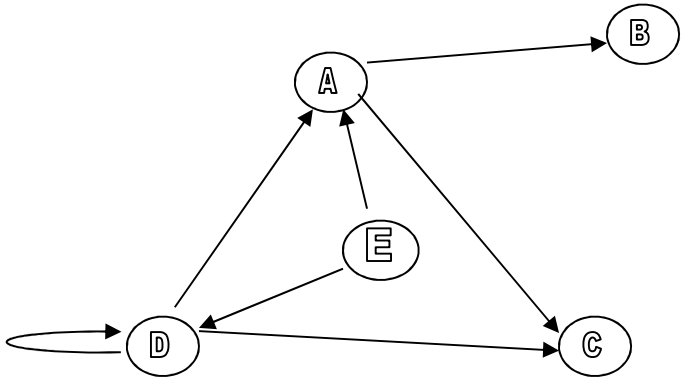
kde

- N je množina uzlů, jimž lze přiřadit hodnotu
- E je množina hran, kterým lze přiřadit hodnotu. Každá hrana spojuje dva uzly a může být orientovaná. Je-li hrana orientovaná, pak jejímu grafu se říká orientovaný graf.
- I je množina spojení, která jednoznačně určuje spojení dvojic uzlů daného grafu.

Příklad neorientovaného grafu



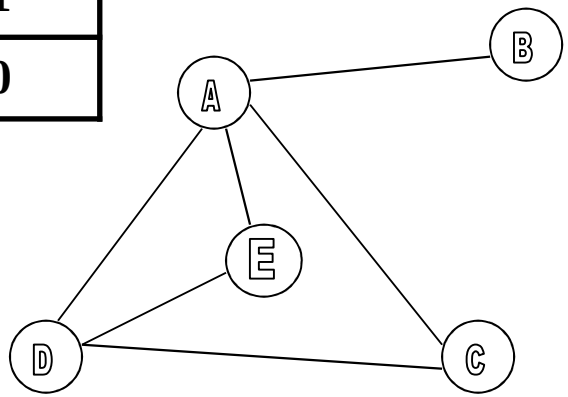
Příklad orientovaného grafu



- Průchodem se nazývá posloupnost všech uzlů grafu.
- Průchod je operace nad grafem, která provádí transformaci nelineární struktury na lineární.
- Cesta z uzlu A do uzlu B je posloupnost hran, po nichž se dostaneme z uzlu A do uzlu B, aniž bychom šli po některé hraně dvakrát.
- Cyklický graf je graf, který obsahuje „cyklus“. Cyklus je neprázdná cesta, která končí v témže uzlu, v němž začíná.

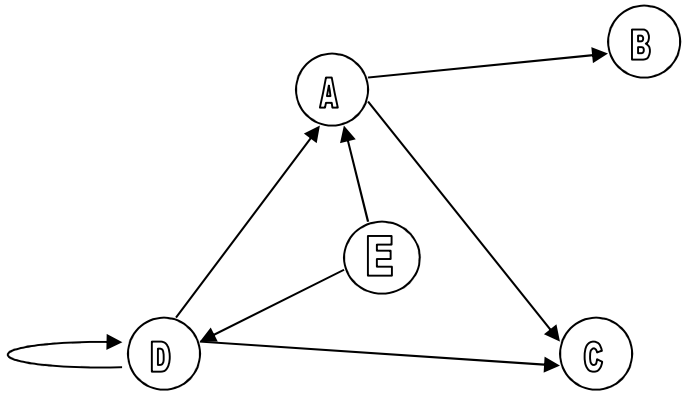
Statická implementace neorientovaného grafu maticí spojení

	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**



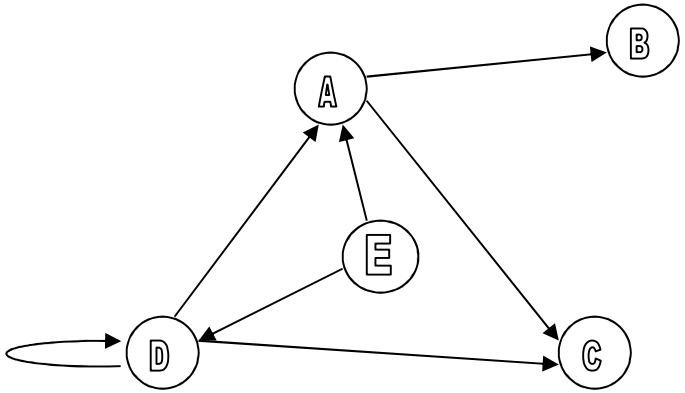
Implementace orientovaného grafu maticí spojení

	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**

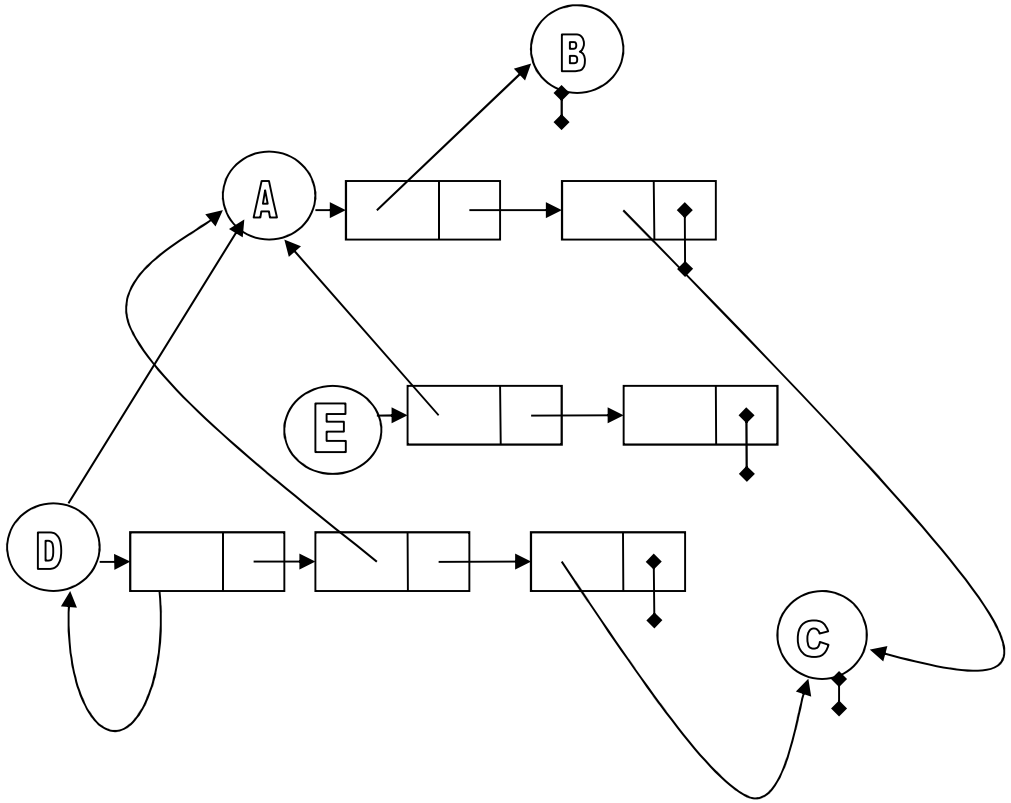


Implementace orientovaného grafu maticí sousednosti

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



Dynamická implementace grafu



Kořenový strom

- Kořenový strom je acyklický graf, který má jeden zvláštní uzel, který se nazývá kořen (root).
- Kořen je takový uzel, že platí, že z každého uzlu stromu vede jen jedna cesta do kořene.
- Z každého uzlu vede jen jedna hrana směrem ke kořeni do uzlu, kterému se říká „otcovský“ uzel a libovolný počet hran k uzlům, kterým se říká „synovské“.
- Výška prázdného stromu je 0, výška stromu s jediným uzlem – kořenem je 1. Výška jiného stromu je počet hran od kořene k nejvzdálenějšímu uzlu + 1.

Binární strom (BS)

- Rekurzívní definice binárního stromu:

  Binární strom je buď prázdný, nebo sestává z jednoho uzlu zvaného kořen a dvou binárních podstromů – levého a pravého. (Oba podstromy mají vlastnosti stromu.)

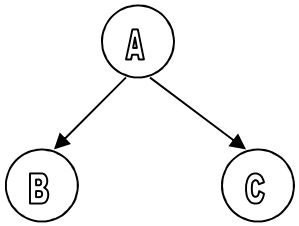
• Binární strom sestává z kořene, neterminálních uzlů, které mají ukazatel na jednoho nebo dva uzly synovské a terminálních uzlů, které nemají žádné „potomky“.

• Každý uzel, který není kořenem stromu je kořenem svého “podstromu”.

• Binární strom je váhově vyvážený, když pro všechny jeho uzly platí, že počty uzlů jejich levého podstromu a pravého podstromu se rovnají, nebo se liší právě o 1.

• Binární strom je výškově vyvážený, když pro jeho všechny uzly platí, že výška levého podstromu se rovná výšce pravého podstromu, nebo se liší právě o 1.

Mějme kořen BS se třemi uzly A,B, a C ve tvar



Pak průchod PreOrder má tvar A,B,C
průchod InOrder má tvar B,A,C
průchod Postorder má tvar B,C,A

Inverzní průchody mají obrácené pořadí synovských uzlů:
InvPreOrder má tvar A,C,B
InvInOrder má tvar C,A,B
InvPostOrder má tvar C,B,A

PostOrder je invertovaný InvPreOrder

```
type
  TPtr=^TNode;  (* ukazatel na typ TUzel *)
  TNode=record  (* type of node *)
          Data:Tdata;
          LPtr,RPtr:TPtr;
        end;

 procedure PreOrder(var L:TList;RootPtr:TPtr);
    (* Seznam L byl inicializován před voláním *)
    begin
      if RootPtr  <> nil
      then begin
        DInsertLast(L,RootPtr^.Data);
        PreOrder(L,RootPtr^.LPtr);
        PreOrder(L,RootPtr^.RPtr)
      end
    end; (* procedure *)
```

Záměnou pořadí rekurzívního volání v podmíněném příkazu if získáme průchody InOrder a PostOrder

```
(* 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).
```