Dátové štruktúry a algoritmy

Pavol Návrat

2008/2009

Data Abstraction

- focus on operations on the data, not on how to implement them on a computer
- example: numbers are abstractions
  - define what set of numbers
  - define what operations on them
- numbers in a computer
  - specify what interval
  - implement operations

Data Type

- numbers, characters, strings etc. are represented as bit patterns
- data type is a method of interpreting such bit patterns
- data type `real` is not a set of all real numbers

Abstract Data Type

- data type as an abstract concept defined by a set of logical properties
- legal operations involving that type are specified
- ADT may be implemented
  - hardware implementation
  - software implementation

Specification of the ADT Natural Number

structure NATNO

declare ZERO() $\rightarrow$ natno

ISZERO(natno) $\rightarrow$ boolean

SUCC(natno) $\rightarrow$ natno

ADD(natno,natno) $\rightarrow$ natno

EQ(natno,natno) $\rightarrow$ boolean

Continued ➔

Specification of the ADT Natural Number (cont.)

for all x,y $\in$ natno let

ISZERO(ZERO) = true

ISZERO(SUCC(x)) = false

ADD(ZERO,y) = y

ADD(SUCC(x),y) = SUCC(ADD(x,y))

EQ(x,ZERO) = if ISZERO(x) then true else false

EQ(ZERO,SUCC(y)) = false

EQ(SUCC(x),SUCC(y)) = EQ(x,y)

end

end NATNO

Zásobník (STACK)

- Abstraktná dátová štruktúra
- Pracuje na princípe LIFO(Last In, First Out)
  - Údaje vložené ako posledné budú vyberané ako prve
- Možné implementácie
  - dynamicky
  - staticky

Zásobník – formálna špecifikácia

- Druhy: STACK, ELM, BOOL
- Operácie:
  - CREATE() -> STACK //vytvorenie zásobníka
  - PUSH(STACK, ELM) -> STACK //vloženie prvku
  - TOP(STACK) -> ELM //výber prvku
  - POP(STACK) -> STACK //zrušenie prvku
  - ISEMPTY(STACK) -> BOOL //test na prázdnosť

Zásobník – formálna špecifikácia

Pre všetky $S \in$ stack, $i \in$ elm platí

ISEMPTY(CREATE) = true

ISEMPTY(PUSH(S,i)) = false

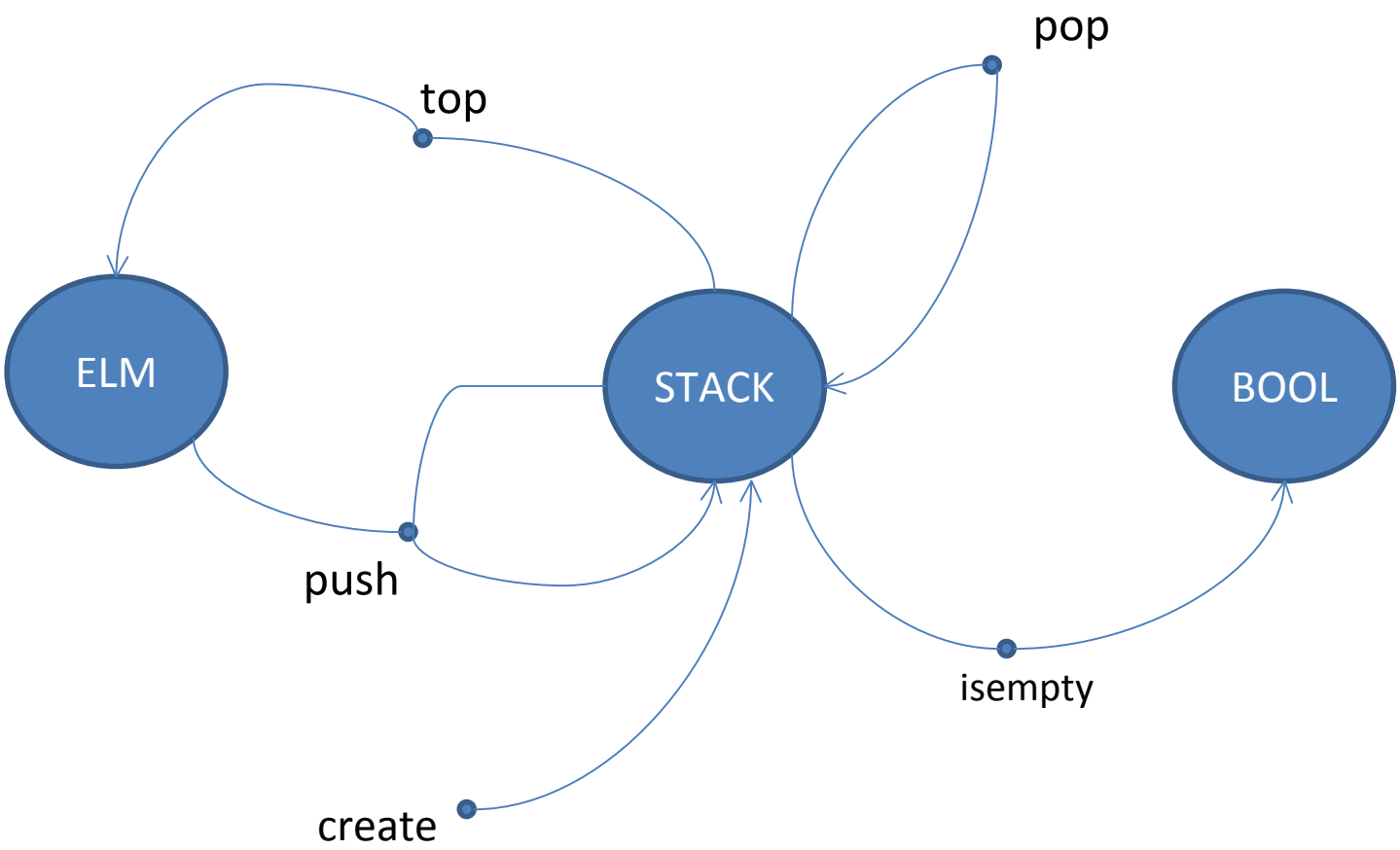
POP(CREATE) = error

POP(PUSH(S,i)) = S

TOP(CREATE) = error

TOP(PUSH(S,i)) = i

Zásobník



Reprezentácia zásobníka

- Staticky – poľom/vektorom
- Dynamicky –
  - Potrebujeme nejakú dynamickú štruktúru
  - (počet jej prvkov sa môže počas výpočtu meniť)
- Návrh:
  - Zoznam, presnejšie
  - Jednosmerne zreťazený zoznam

Jednosmerne zreťazený zoznam Singly Linked List (SLL)

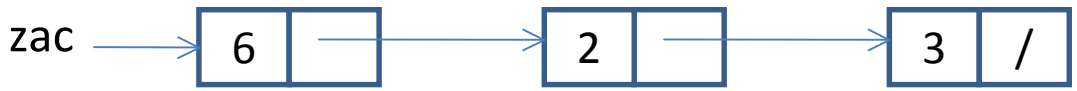
- Najjednoduchšie vyjadrenie lineárneho spájaného zoznamu
- Každý prvok obsahuje dátovú časť a ukazovateľ na ďalší prvok
- Ukazovateľ posledného prvku ukazuje na NULL

1ZZ - SLL

- Základné operácie:
  - CREATE: vytvorenie prázdneho SLL
  - ISEMPTY: test, či je zoznam prázdny
  - INSERT: vloženie prvku
  - DELETE: vymazanie prvku
  - FIND: nájdenie prvku
- Ďalšie operácie:
  - DELETE_ALL, NUM_ELEMENTS, ...

SLL - Reprezentácia

```
Typedef struct uzol *SLL_UZOL;
Struct uzol
{
   SLL_TYP        prvok;   //dátová časť
   SLL_UZOL       next;    //nasledovník
}
SLL_UZOL zac;    //smerník na začiatok
```



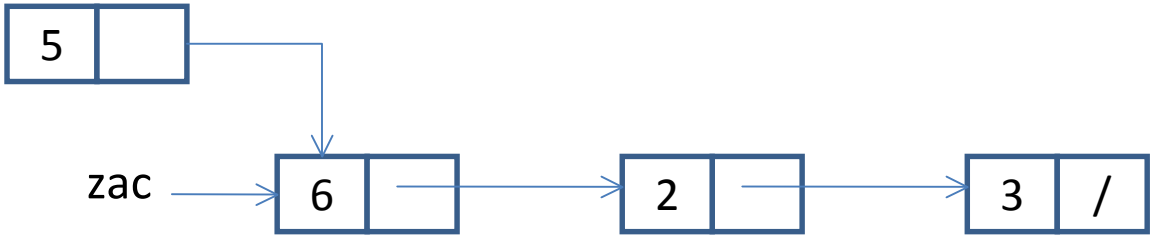
SLL insert

Insert 5:

1. Krok : Vytvorenie nového prvku

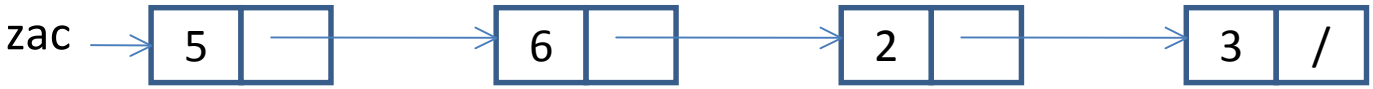
2. Krok : Priradenie smerníka a hodnoty novovytvorenému prvku

Nový prvok bude ukazovat na to iste miesto v pamati (na ten istý prvok) ako ukazuje začiatok SLL



3. Krok : Nastavenie nového začiatku SLL

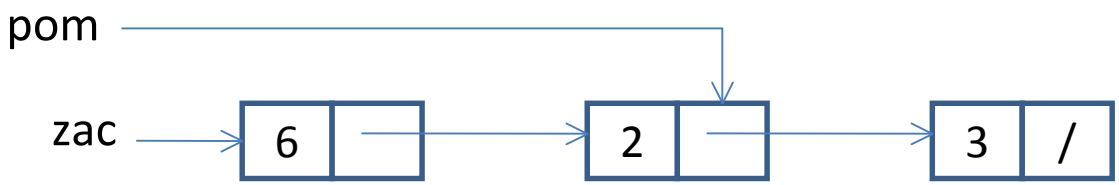
Začiatok SLL bude ukazovať na nový prvok



SLL delete

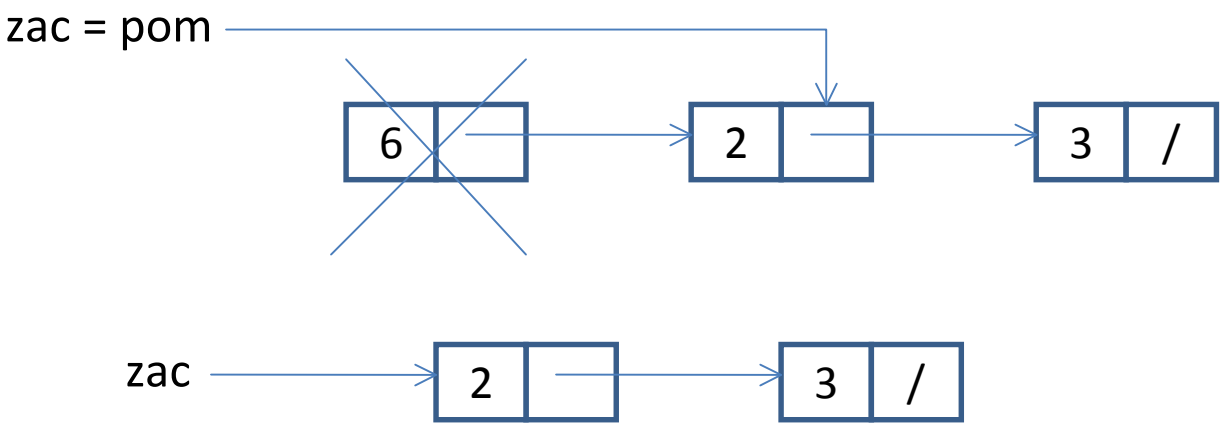
Delete:

1. Krok: Pomocnej premennej sa priradí ukazovateľ prvého prvku



2. Krok : Vymazanie prvého prvku a nastavenie zaciatku SLL

Začiatku SLL sa priradí ukazovateľ uložený v pomocnej premennej



Implementácia zásobníka pomocou SLL

```
typedef  SLL_UZOL  STACK;
typedef  SLL_TYP    ST_TYP;
typedef  int            BOOL;

STACK  zasobnik;

STACK  CREATE()
{
     SLL_create( zasobnik );
}

BOOL  ISEMPTY( STACK zasobnik )
{
     return  SLL_isempty( zasobnik );
}
```

Implementácia zásobníka pomocou SLL

```
STACK  POP( STACK zasobnik )
{
      if( ISEMPTY( zasobnik))
            return ERROR;
      else
            return SLL_delete( zasobnik );
}

ST_TYP TOP( STACK zasobnik )
{
      if( ISEMPTY( zasobnik))
            return ERROR;
      else
            return zasobnik->prvok;
}

STACK  PUSH( STACK zasobnik, ST_TYP hodnota )
{
      return SLL_insert( zasobnik, hodnota );
}
```

Implementácia zásobníka pomocou vektora

using a simple array to represent it

```
CREATE(S)
    top(S) ← 0

PUSH(S,x)
    top(S) ← top(S) + 1
    S[top(S)] ← x

POP(S)
    if ISEMPTY(S)
          then error”underflow”
          else top(S) ← top(S) - 1
               return  S[top(S) + 1]
```

Continued ➔

Implementation of the ADT Stack (cont.)

```
TOP(stack) → item
    return S[top(S)]

ISEMPTY(S)
    return top(S) = 0
```

FRONT (QUEUE)

- Abstraktná dátová štruktúra
- Pracuje na princípe FIFO(First In, First Out)
  - Údaje vložené ako prvé budú vyberané ako prvé
- Možné implementácie
  - ZVP
  - Poľom

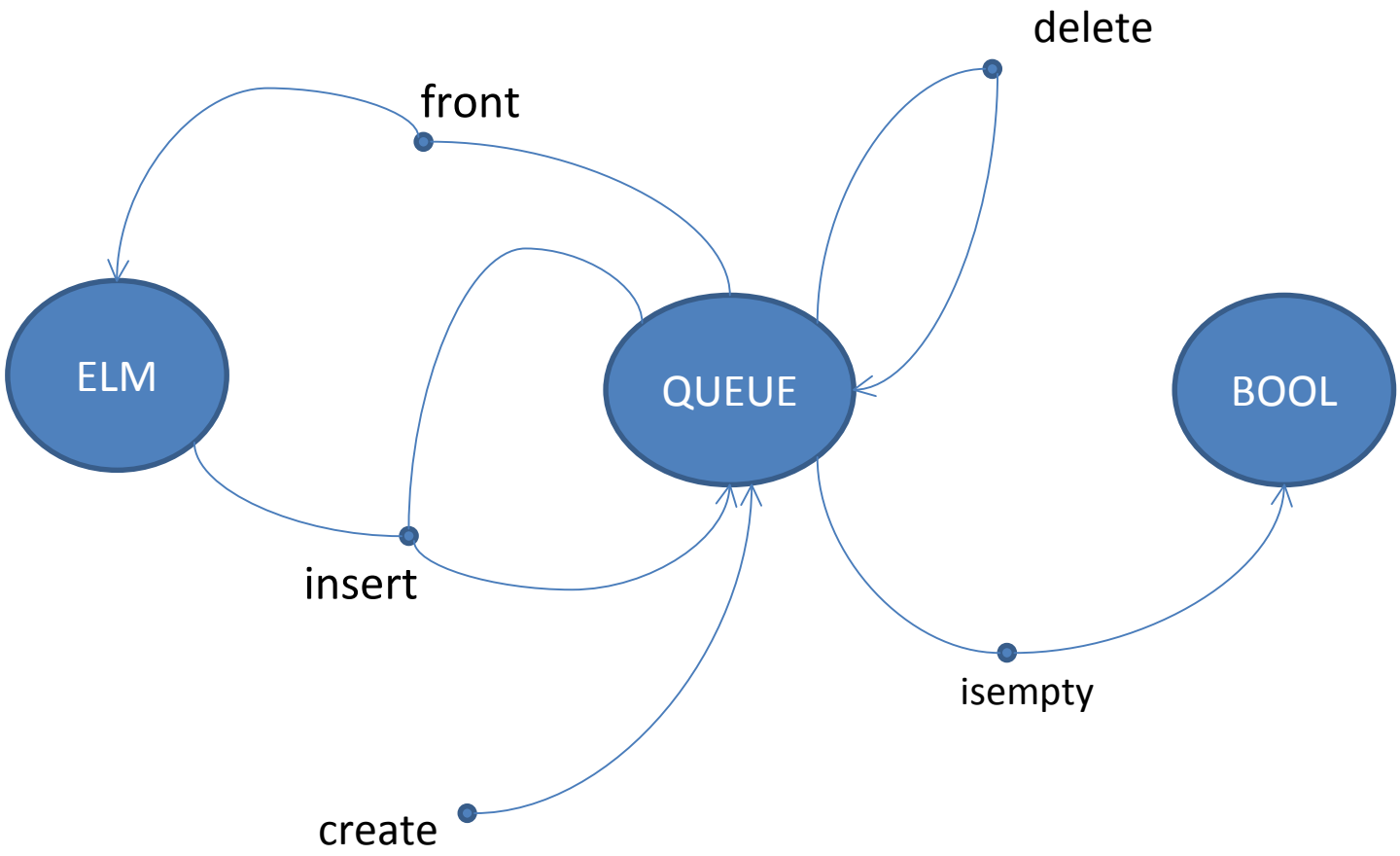
Front – formálna špecifikácia

- Druhy: QUEUE, ELM, BOOL
- Operácie:
  - CREATE() -> QUEUE //vytvorenie frontu
  - INSERT(QUEUE, ELM) -> QUEUE //vloženie prvku
  - FRONT(STACK) -> ELM //výber prvku
  - DELETE(QUEUE) -> QUEUE //zrušenie prvku
  - ISEMPTY(STACK) -> BOOL //test na prázdnosť

Front – Formálna špecifikácia

pre všetky Q ∈ queue, i ∈ elm platí
ISEMPTY(CREATE) = true
ISEMPTY(INSERT(Q,i)) = false
DELETE(CREATE) = error
DELETE(INSERT(Q,i)) =
if ISEMPTY(Q) then CREATE
else INSERT(DELETE(Q),i)
FRONT(CREATE) = error
FRONT(INSERT(Q,i)) =
if ISEMPTY(Q) then i
else FRONT(Q)

Front



Implementácia frontu pomocou SLL

```
typedef struct uzol *SLL_UZOL;
struct uzol
{
    SLL_TYP     prvok;          //prvok typu SLL_TYP
    SLL_UZOL    n;     //nasledovník
}
typedef struct hlavicka
{
    SLL_UZOL    zac, kon;       // smerníky začiatku a konca
} F_HLAV;

F _HLAV  h;  // smerník na front

F_HLAV   CREATE( F_HLAV h)
{
    h.zac = h.kon = NULL;
    return h;
}
```

Implementácia frontu pomocou SLL

```
bool   ISEMPTY( F_HLAV h)
{
      return (h.zac == NULL );
}

SLL_TYP  FRONT(F_HLAV h)
{
      if( ISEMTY(h)
            printf("CHYBA !");
      else
            return h.zac->prvok;
}
```

Implementácia frontu pomocou SLL

```
F_HLAV   DELETE( F_HLAV h )
{
        SLL_UZOL    pom;
        if( ISEMTY(h))   printf("CHYB A !");
        else
        {
                pom = h.zac->n;
                free( h.zac);
                h.zac = pom;
        }
        return h;
}

F_HLAV   INSERT(F_HLAV h, SLL_TYP hodnota)
{
        SLL_UZOL    pom;
        pom = (SLL_UZOL) malloc( sizeof(uzol));
        pom->prvok = hodnota;
        pom->n = NULL;
        if( ISEMPTY(h)) {     h.kon = h.zac = pom; }
        else   // Pridaj na koniec
        {
                h.kon->n = pom;
                h.kon = pom;
        }
        return h;
}
```

Implementácia frontu pomocou vektora

using a simple array to represent it.

structure QUEUE

CREATE(Q)

    tail(Q) ← 1

    head(Q) ← 1

INSERT(Q,x)

    Q[tail(Q)] ← x

    if tail(Q) = length(Q)

        then tail(Q) ← 1

        else tail(Q) ← tail(Q) + 1

Continued ➔

Implementation of the ADT Queue (cont.)

```
DELETE(Q, x)
    x ← Q[head(Q)]
    if head(Q) = length(Q)
          then head(Q) ← 1
          else head(Q) ← head(Q) + 1
    return x
FRONT(Q)
    return Q[head(Q)]
ISEMPTY(Q)
    return head(Q) = tail(Q)
```

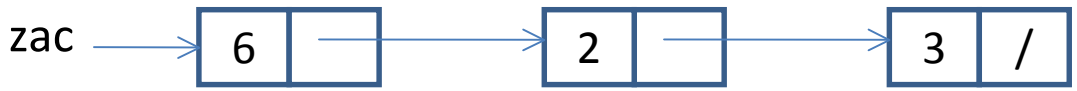
Obrázok štruktúry kruhového venktora

Hierarchia abstrakcií

- Pre zásobník:
  - Zásobník
  - Jednosmerne zreťazený zoznam
  - ? -> zreťazená voľná pamäť (dynamická pamäť)
- Alebo:
  - Zásobník
  - vektor
  - ? -> statická pamäť

SLL - Reprezentácia

```
Typedef struct uzol *SLL_UZOL;
Struct uzol
{
   SLL_TYP        prvok;   //dátová časť
   SLL_UZOL       next;    //nasledovník
}
SLL_UZOL zac;    //smerník na začiatok
```



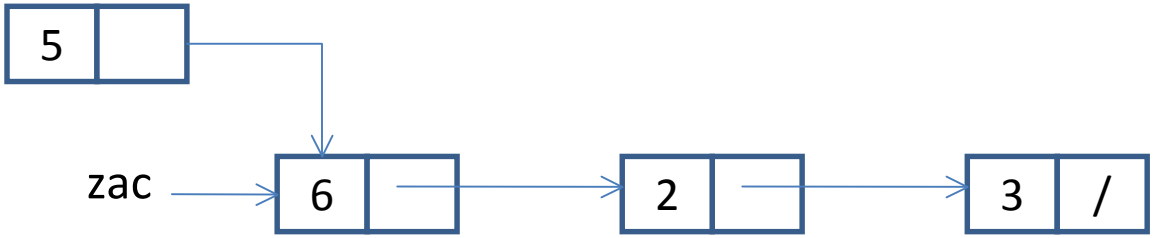
SLL insert

Insert 5:

1. Krok : Vytvorenie nového prvku

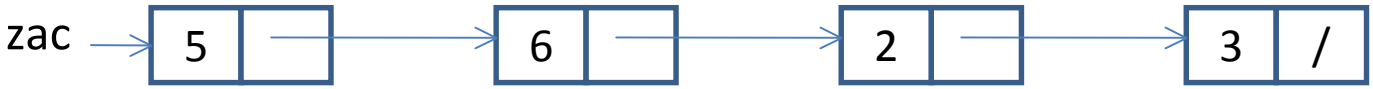
2. Krok : Priradenie smerníka a hodnoty novovytvorenému prvku

Nový prvok bude ukazovat na to iste miesto v pamati (na ten istý prvok) ako ukazuje začiatok SLL



3. Krok : Nastavenie nového začiatku SLL

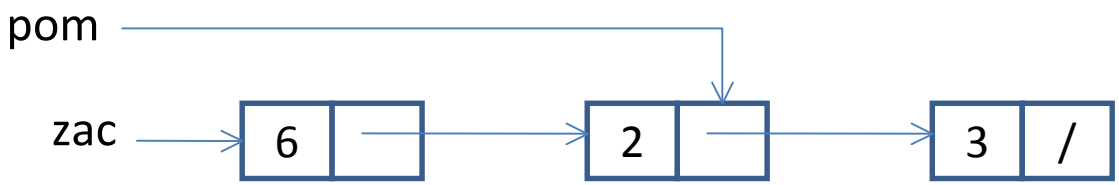
Začiatok SLL bude ukazovať na nový prvok



SLL delete

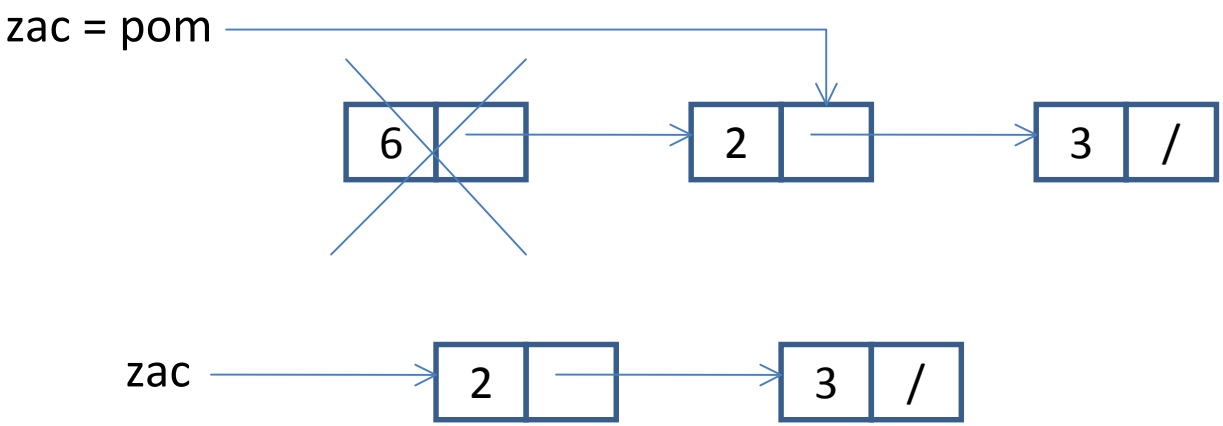
Delete:

1. Krok: Pomocnej premennej sa priradí ukazovateľ prvého prvku



2. Krok : Vymazanie prvého prvku a nastavenie zaciatku SLL

Začiatku SLL sa priradí ukazovateľ uložený v pomocnej premennej



SLL implementácia

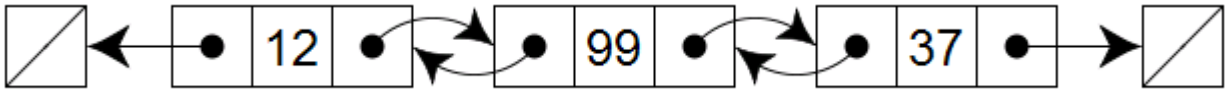
```
SLL_UZOL SLL_create ( SLL_UZOL zac)
{
        zac = NULL;
        return zac;
}

int   SLL_isempty ( SLL_UZOL smernik )
{
        return ( smernik= = NULL );
}

SLL_UZOL SLL_insert (    SLL_UZOL smernik, SLL_TYP hodnota   )
{
        SLL_UZOL    pom;
        pom = (SLL_UZOL) malloc( sizeof(uzol));
        pom->prvok = hodnota;
        pom->n = smernik;
        smernik = pom;
        Return smernik;
}
```

SLL implementácia

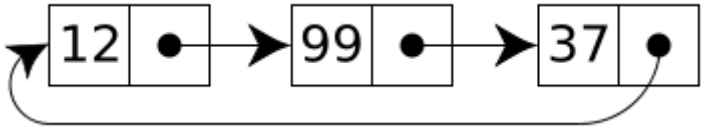
```
SLL_UZOL SLL_delete( SLL_UZOL smernik )
{
        SLL_UZOL    pom;
        if(! SLL_isempty ( smernik ) )
        {
                pom = smernik->n;
                free( smernik);
                smernik = pom;
        }
        return smernik;
}

SLL_UZOL SLL_find( SLL_UZOL smernik, SLL_TYP hodnota )
{
        for (; !SLL_isempty(smernik); smernik = smernik->n)
         if( hodnota = smernik->prvok)
                return smernik;
        return     NULL;
}

void SLL_all_elements( SLL_UZOL smernik)
{
        for(; !SLL_isempty(smernik); smernik = smernik->n)
                printf(" %d \n", smernik->prvok);
}
```

Iné reprezentácie spájaného zoznamu

- Obojsmerne spájaný zoznam
  - Každý prvok obsahuje ukazovateľ na ďalší prvok a aj na predchádzajúci prvok



- Cyklicky spájaný zoznam
  - Posledný prvok zoznamu ukazuje na prvý prvok



Zreťazená voľná pamäť

- Predstavuje základný abstraktný typ, ktorý sa využíva pri implementácií ostatných abstraktných údajových typov
- Prvok obsahuje príznak, či je voľný a potom v závislosti od toho či je voľný obsahuje buď informáciu o ďalšom voľnom (keď je voľný), alebo dátovú časť (keď nie je voľný)
- ZVP obsahuje okrem poľa prvkov ešte aj informáciu o prvom voľnom prvku

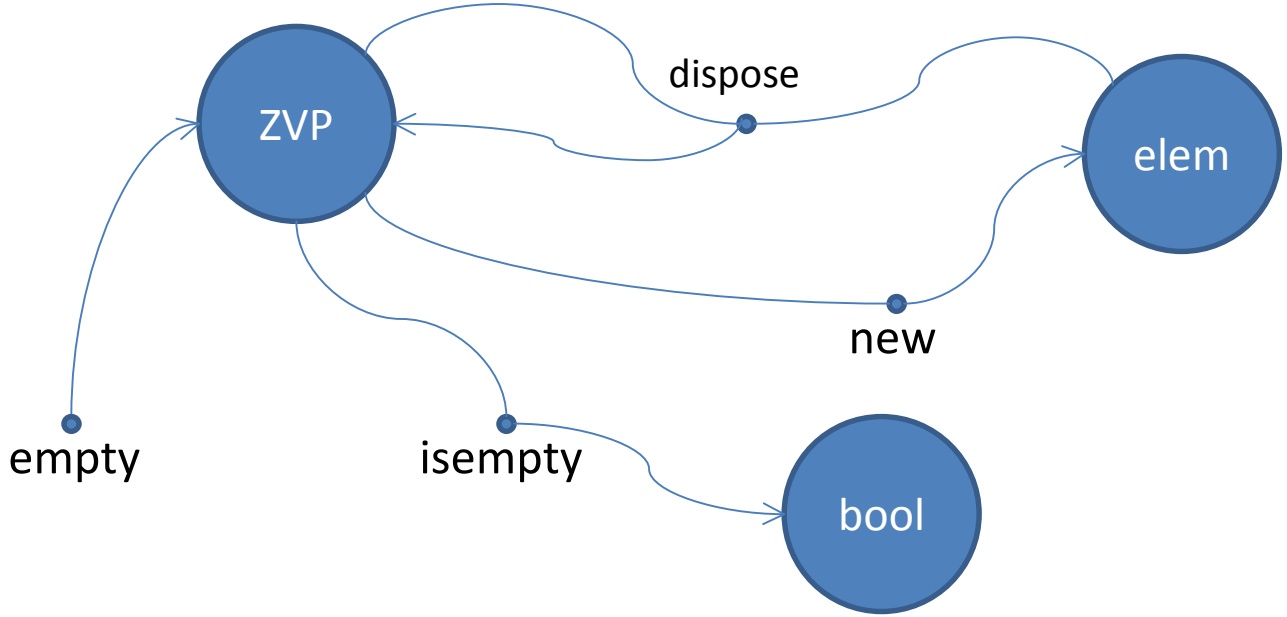
ZVP – Formálna špecifikácia

- CREATE() -> zvp
- NEW(zvp) -> item
- DISPOSE(zvp, item) -> zvp
- ISEMPTY(zvp) -> bool

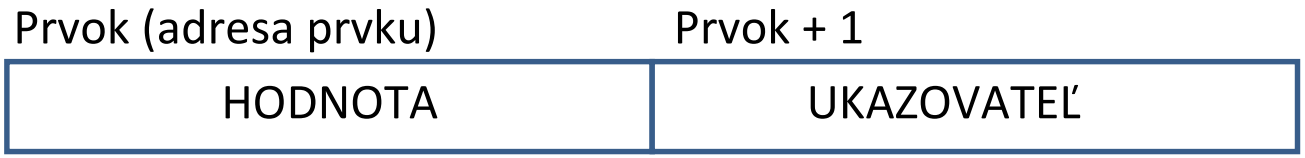
Pre všetky Z ∈ zvp, i ∈ item platí

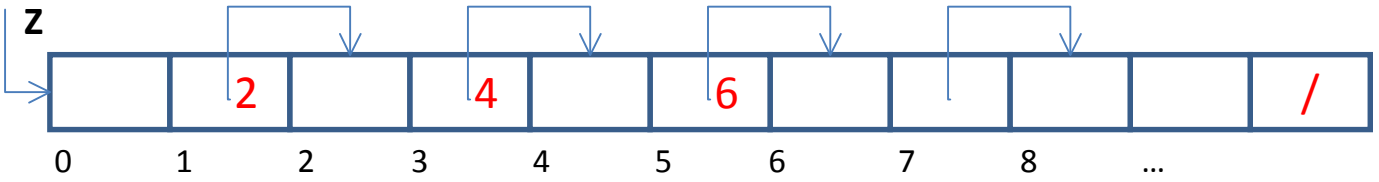
- ISEMPTY(DISPOSE(Z , i)) = true
- DISPOSE(CREATE , i) = ERROR
- DISPOSE(Z,NEW(Z)) = Z
- NEW(DISPOSE(Z , i)) = i

ZVP



ŠTRUKTÚRA PRVKU:





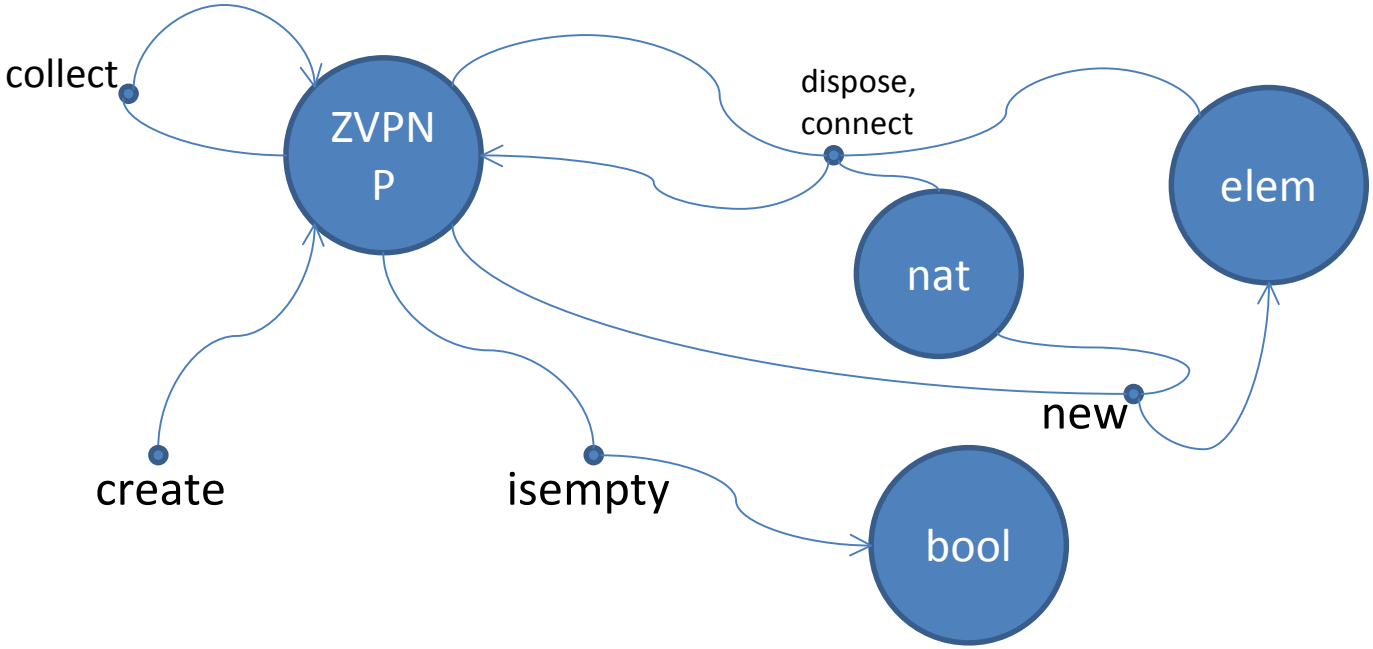
ZVP – príklad implementácie

```
VAR ZVP: IND;

Function EMPTY(VAR ZVP:IND)   //vytvorenie zvp, vykoná sa začiatočné zretazenie voľných prvkov
VAR
        UK : IND;
BEGIN
        UK := ADRMIN;
        WHILE  UK < ADRMAX -2  DO
                        BEGIN
                            PAMAT[UK +1] := UK +2;
                            UK := UK + 2;
                        END;
        PAMAT[ UK + 1] := NIL;
        ZVP := ADRMIN;
END;

Function NEWZ( VAR ZVP : IND; VAR PRVOK : IND) //operácia poskytnutia prvku zo ZVP
BEGIN
        IF ISEMPTY(ZVP)
                        THEN WRITELN(" Chyba: vyčerpanie pamate");
        ELSE
                        BEGIN
                            PRVOK := ZVP;
                           ZVP := PAMAT[ ZVP + 1];
                        END
END;
```

ZVP nerovnakých prvkov

- Druhy: ZVPNP, ELM, NAT, BOOL
- Operácie:
  - CREATE() -> ZVPNP
  - NEW(ZVPNP, NAT) -> ELM
  - DISPOSE(ZVPNP, NAT, ELEM) ->ZVPNP
  - CONNECT(ZVPNP, NAT, ELM) -> ZVPNP (spojenie 2 prvkov)
  - COLLECT(ZVPNP) -> ZVPNP (spojenie do súvislej oblasti)
  - ISEMPTY (ZVPNP) -> BOOL

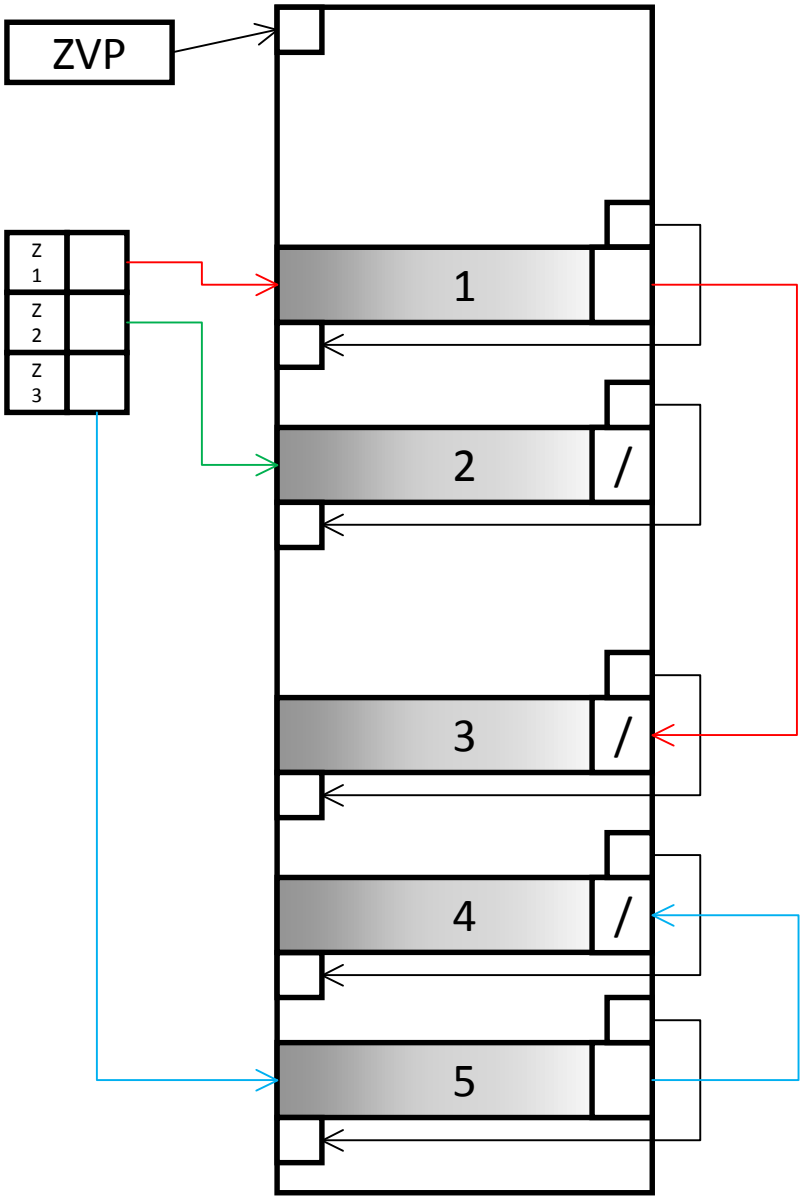
ZVPNP



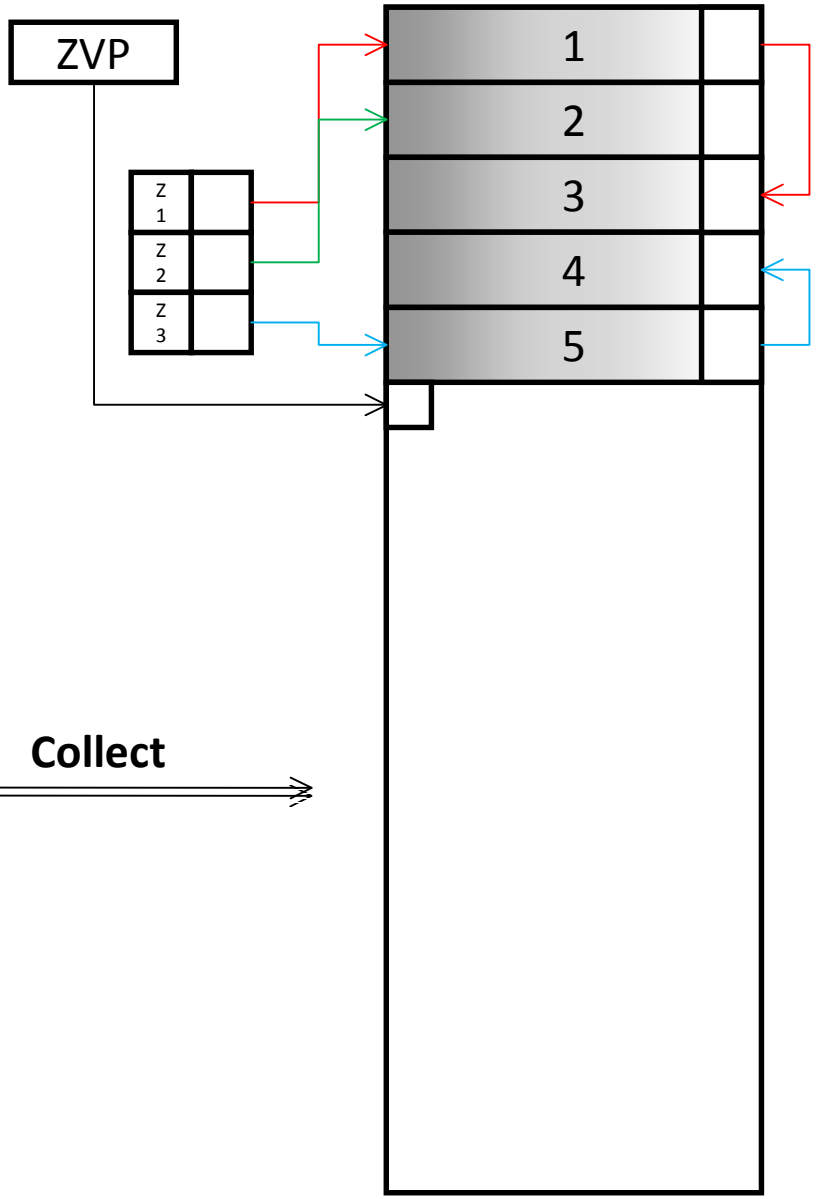
ŠTRUKTÚRA PRVKU:

Prvok	Prvok + 1	Prvok + Dĺžka - 1
DĹŽKA	HODNOTA	UKAZOVATEĽ

ZVP – COLLECT (implementácia zásobníkov v ZVP)



Collect



STROM

Strom – definícia

1. Jediný vrchol je strom – tento vrchol je zároveň koreň tohto stromu
2. Nech **V** je vrchol a **S1**,**S2**..**Sn** sú stromy s koreňmi **V1**,**V2**..**Vn**. Nový strom môžeme zostrojiť tak, že vrchol V urobíme **PREDCHODCOM** vrcholov **V1**,**V2**..**Vn**. V tomto novom strome je V koreň a **S1**,**S2**..**Sn** sú jeho podstromy. Vrcholy **V1**,**V2**..**Vn** sú **NASLEDOVNÍCI** vrcholu V

Binárny strom

- Strom, ktorý má najviac dvoch potomkov.
- Potomkovia sa označujú ako ĽAVÝ a PRAVÝ nasledovník
- Jedno s bežných využití binárneho stromu je binárny vyhľadávací strom

Strom – formálna špecifikácia

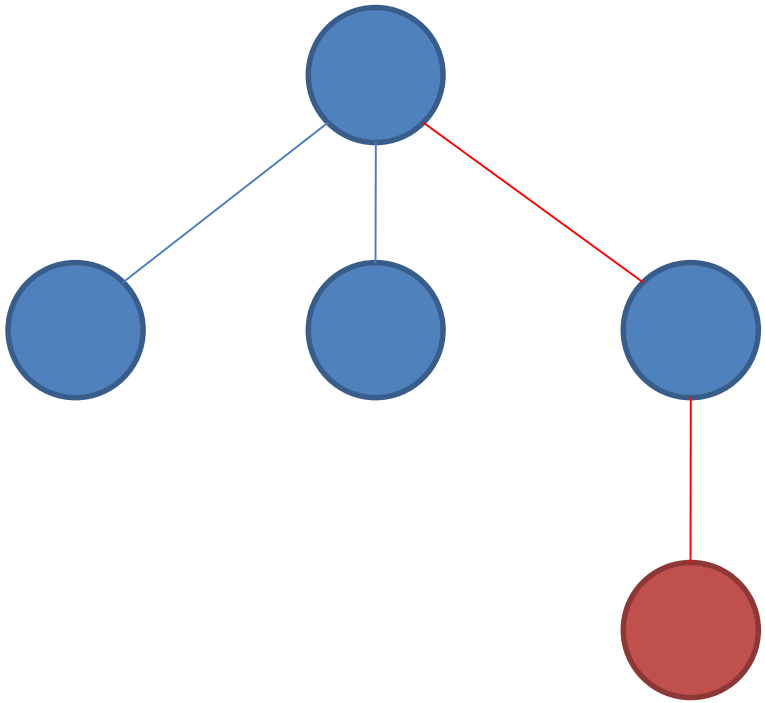
- Operácie:
  - EMPTY : vytvorenie prázdneho stromu
  - EMPTYn : nekonečná rodina operácií.
    - EMPTYi(a, S1,S2..Si) vytvorí nový vrchol V s hodnotou a, ktorý má i nasledovníkov – sú to korene stromov S1..Si
  - KOREŇ : Nájdenie koreňa stromu
  - PREDCHODCA : Nájdenie predchodcu daného vrcholu

Strom – formálna špecifikácia

- LNASLEDOVNIK : Nájdenie najľavejšieho nasledovníka
- PSUSED : Nájdenie vrcholu, ktorý má rovnakého predchodcu ale v usporiadaní stromu je vpravo za daným vrcholom.
- HOD : Získanie Ohodnotenia vrcholu

Strom – základné definície

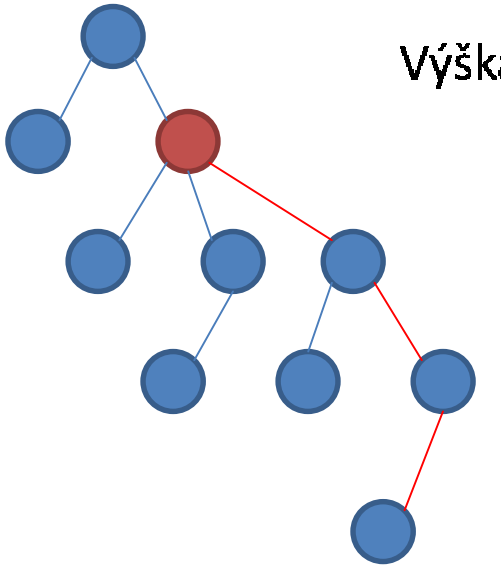
- Hĺbka vrcholu – počet hrán od koreňa stromu k danému vrcholu



hĺbka = 2

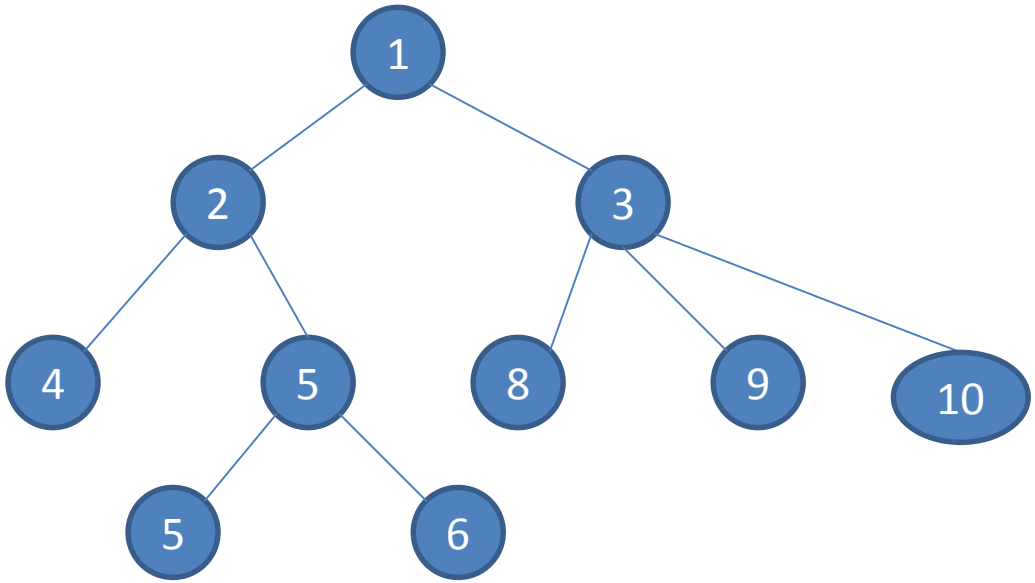
Strom - základné definície

- Výška vrcholu – najdlhšia cesta z vrcholu k ľubovoľnému koncovému vrcholu
- Výška stromu – výška jeho koreňa



Výška vrcholu = 3

Strom – reprezentácia poľom

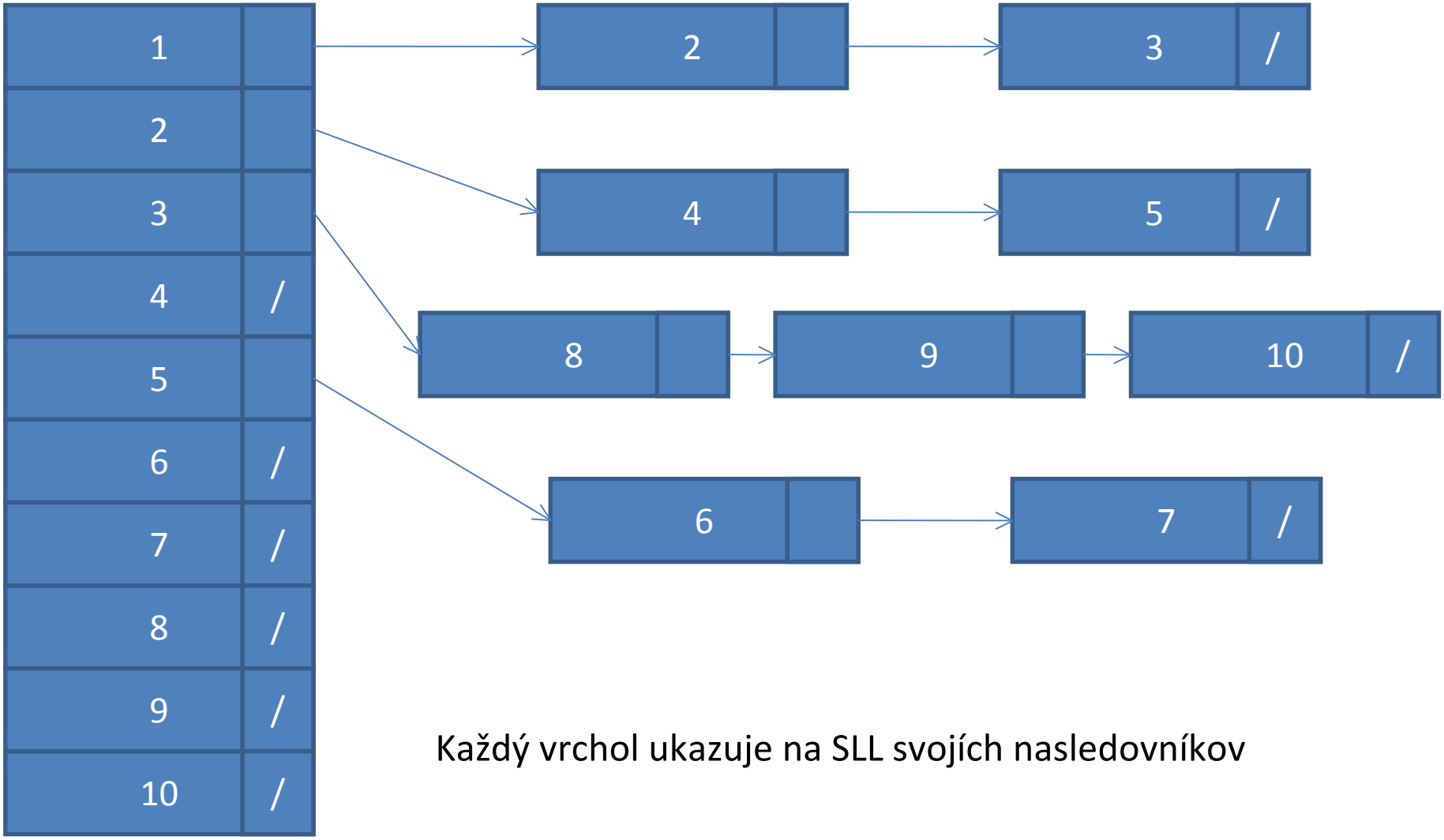


Pole[i]	0	1	1	2	2	5	5	3	3	3
index	1	2	3	4	5	6	7	8	9	10

Index poľa = hodnota vrcholu

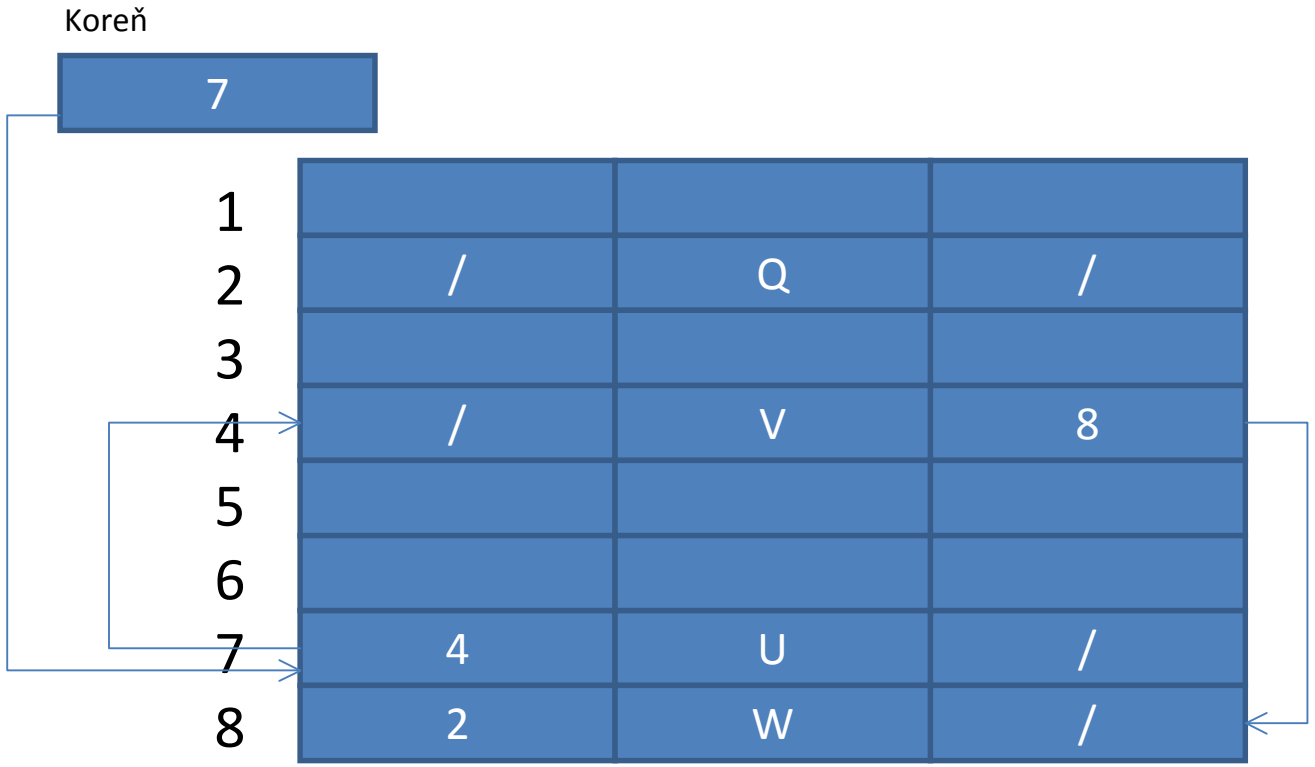
Hodnota prvku poľa = ukazovateľ na rodiča

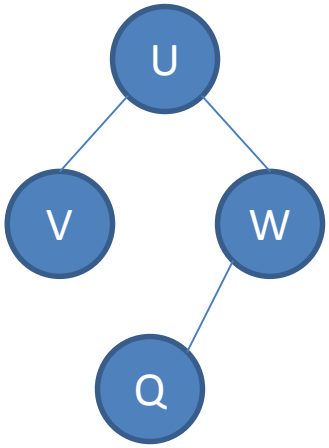
Strom – reprezentácia pomocou ZVP



Každý vrchol ukazuje na SLL svojích nasledovníkov

Strom – reprezentácia





Každý prvok obsahuje ukazovateľ na ľavého nasledovníka a pravého suseda

Binárny strom

- Pozostáva z vrcholov.
- Jediný vrchol je binárny strom a súčasne koreň.
- Ak *u* je vrchol a T1 a T_2 sú stromy s koreňmi v_1 a v_2, tak usporiadaná trojica (T_1, *u,* T_2) je binárny strom, ak v_1 je ľavý potomok koreňa *u* a v_2 je jeho pravý potomok.
- List - vrchol bez potomkov.
- Úplný binárny strom - binárny strom, v ktorom každý nelistový vrchol má práve dvoch potomkov.

Binárny strom

Operácie nad binárnym stromom:

- CREATE: vytvorenie prázdneho binárneho stromu
- MAKE: vytvorenie binárneho stromu z dvoch už existujúcich binárnych stromov a hodnoty
- LCHILD: vrátenie ľavého podstromu
- DATA: vrátenie hodnoty koreňa v danom binárnom strome
- RCHILD: vrátenie pravého podstromu
- ISEMPTY: test na prázdnosť

Binárny strom – formálna špecifikácia

CREATE() → btree

MAKE(item,btree,item) → btree

LCHILD(btree) → btree

DATA(btree) → item

RCHILD(btree) → btree

ISEMPTY(btree) → boolean

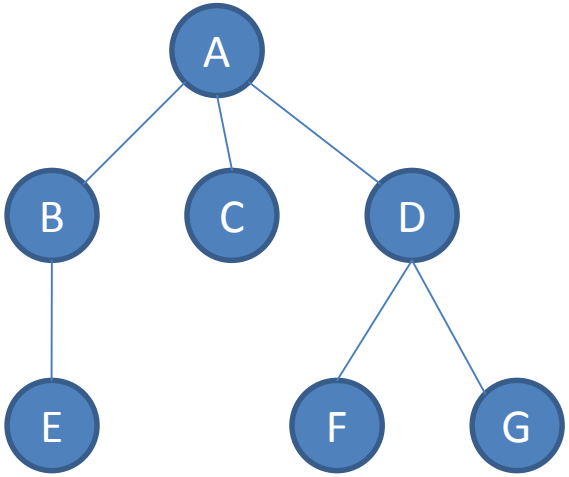
Binárny strom – formálna špecifikácia

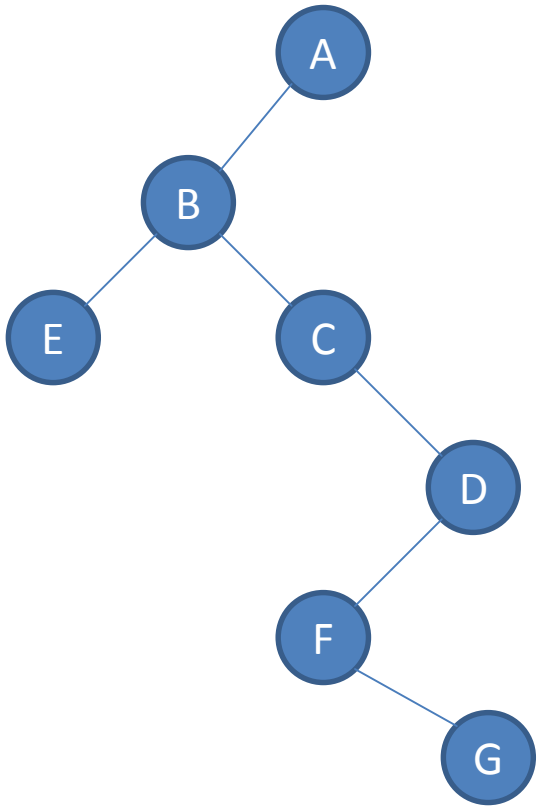
Pre všetky p,r $\in$ btree, i $\in$ item platí:

- ISEMPTY(CREATE) = true
- ISEMPTY(MAKE(p,i,r)) = false
- LCHILD(MAKE(p,i,r)) = p
- LCHILD(CREATE) = error
- DATA(MAKE(p,i,r) = i
- DATA(CREATE) = error
- RCHILD(MAKE(p,i,r)) = r
- RCHILD(CREATE) = error

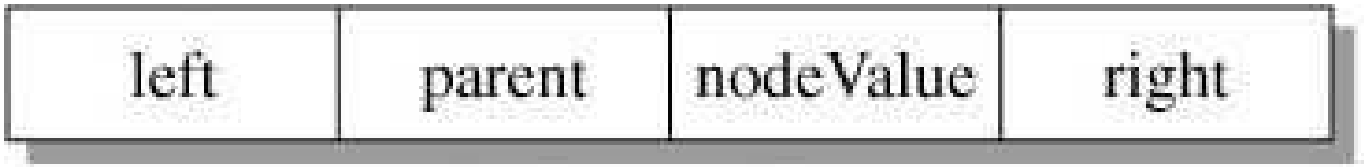
Reprezentácia stromu pomocou binárneho stromu

- LCHILD = ľavý nasledovník daného vrcholu
- RCHILD = pravý sused daného vrcholu

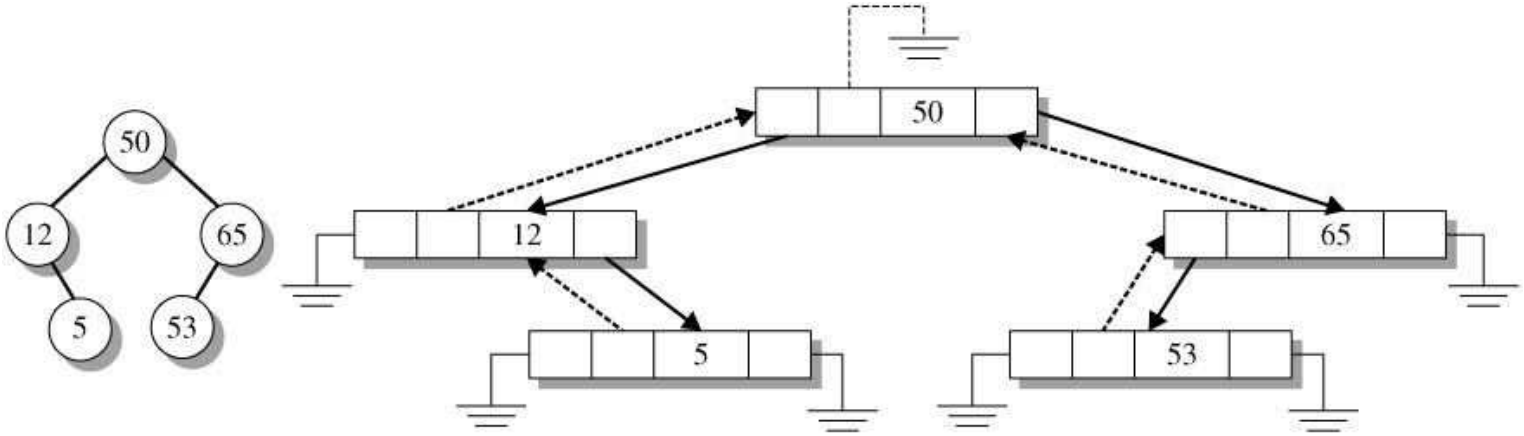




Binárny strom – implementácia



STNode object



Binary Search Tree

STnode Representation of Binary Search Tree using Parent Pointers

Ford, W., Topp, W.: *Data Structures with Java.* Pearson Prentice Hall, 2004. ISBN 0130477249, 9780130477248.

Prehľadávanie binárnych stromov

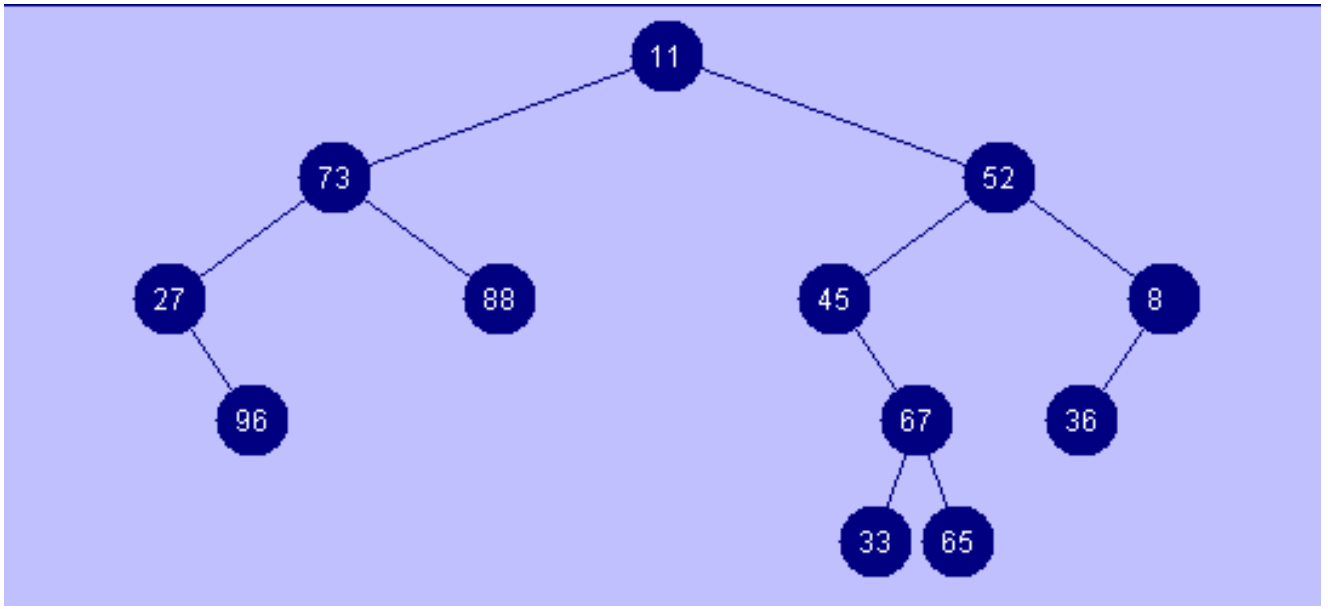
Tri základné algoritmy:

- ***preorder*** - poradie prehľadávania:
  koreň - ľavý podstrom - pravý podstrom
- ***inorder*** - poradie prahľadávania:
  ľavý podstrom - koreň - pravý podstrom
- ***postorder*** - poradie prahľadávania:
  ľavý podstrom - pravý podstrom - koreň.

Prehľadávanie binárnych stromov

```
PREORDER(T) if T <> nil then
    OUTPUT(DATA(T))
    PREORDER(LCHILD(T))
    PREORDER(RCHILD(T))

INORDER(T) if T <> nil then
    INORDER(LCHILD(T))
    OUTPUT(DATA(T))
    INORDER(RCHILD(T))

POSTORDER(T) if T <> nil then
    POSTORDER (LCHILD(T))
    POSTORDER (RCHILD(T))
    OUTPUT(DATA(T))
```

Prehľadávanie binárnych stromov



Preorder: 11,73,27,96,88,52,45,67,33,65,8,36

Inroder: 27,96,73,88,11,45,33,67,65,52,36,8

Postorder: 96,27,88,73,33,65,67,45,36,8,52,11

Binárne vyhľadávacie stromy (BVS)

- BVS je binárny strom.
- BVS môže byť prázdny.
- Ak BVS nie je prázdny, tak spĺňa nasledujúce podmienky:
  - každý prvok má kľúč a všetky kľúče sú rôzne,
  - všetky kľúče v ľavom podstrome sú menšie ako kľúč v koreni stromu
  - všetky kľúče v pravom podstrome sú väčšie ako kľúč v koreni stromu,
  - ľavý aj pravý podstrom sú tiež BVS.

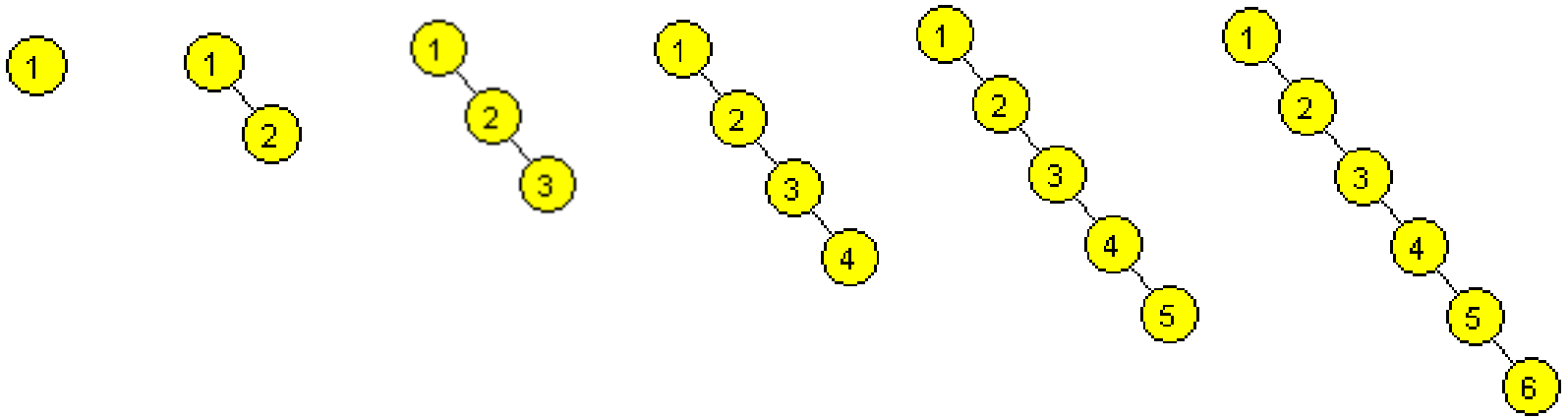
BVS – insert (implementácia)

```
TREE-INSERT(T,n)
    Y ← nil; X ← ROOT(T)
    while X <> nil
          do       Y←X
          if DATA(n) < DATA(X)
                   then X ← LCHILD(X) else X ← RCHILD(X)
    PARENT(n) ← Y
    If Y = nil
          then ROOT(T) ← n
    else if DATA(n) < DATA(Y)
          then LCHILD(Y) ← n else RCHILD(Y) ← n
```

BVS – insert (zložitosť)

- Musíme nájsť miesto, kde môžeme prvok vložiť – časová zložitosť závisí od hĺbky stromu
  - Najhorší prípad O(n)
    - Na vstupe je zoradená postupnosť – vytvárame nevyvážený strom –> rýchle zväčšovanie hĺbky stromu

      1,3,4,5,6

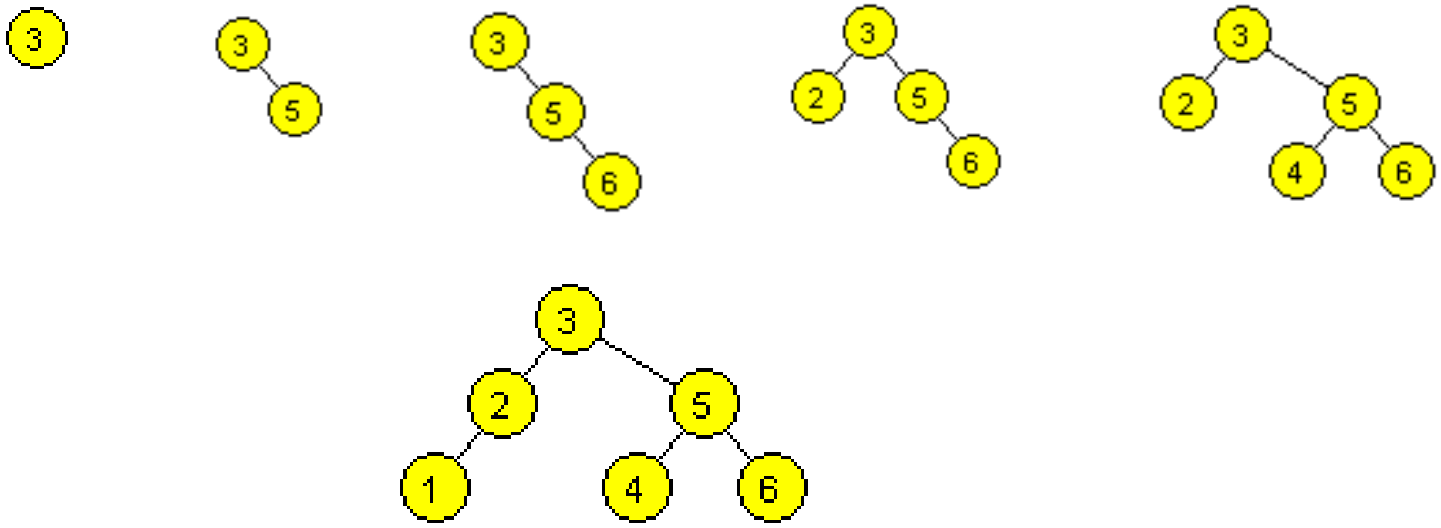


BVS – insert (zložitosť)

– Priemerný prípad O(log n)

- Na vstupe náhodná postupnosť – vytvárame väčšinou „dobre“ vyvážený strom –> pomalé zväčšovanie hĺbky stromu

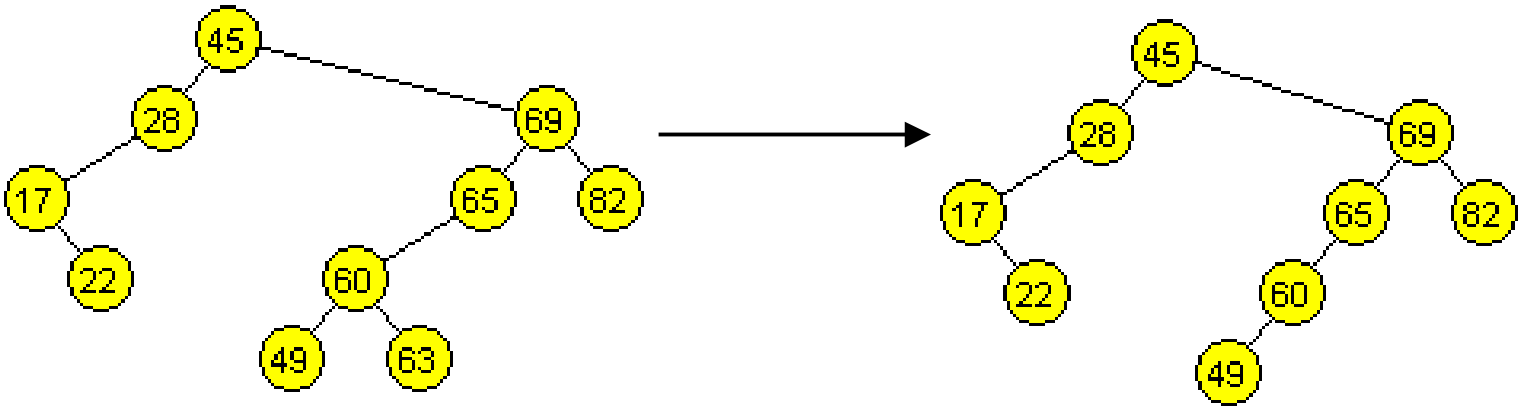
  3,5,6,2,4,1



BVS - DELETE

- Rozloženie algoritmu na tri prípady
  - uzol na odstránenie nemá žiadny podstrom
    - jednoduché odstránenie uzla

Odstránenie uzla 63

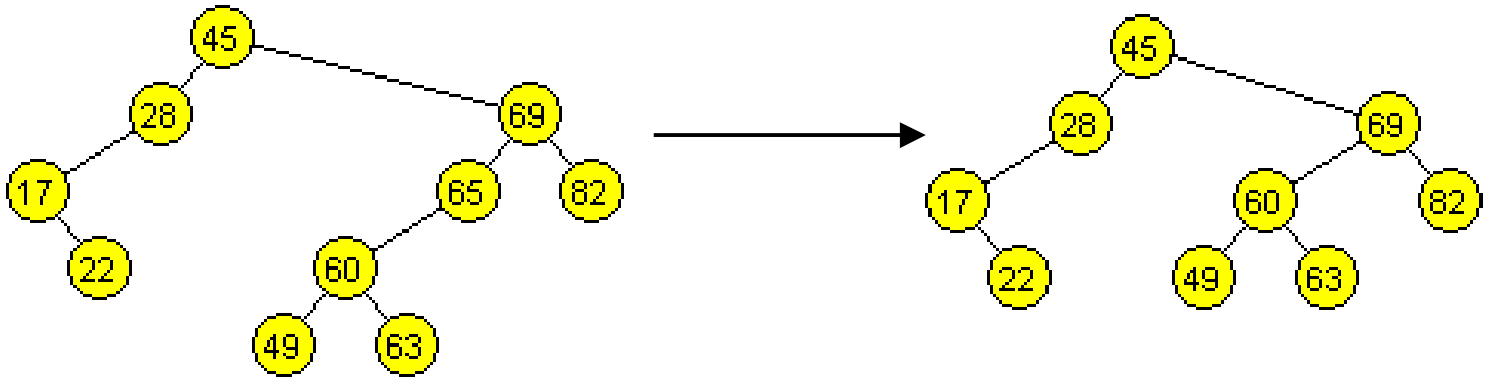


BVS - DELETE

– uzol na odstránenie má jeden podstrom

- odstránenie uzla, prepojenie koreňa jeho podstromu s jeho rodičom

Odstránenie uzla 65

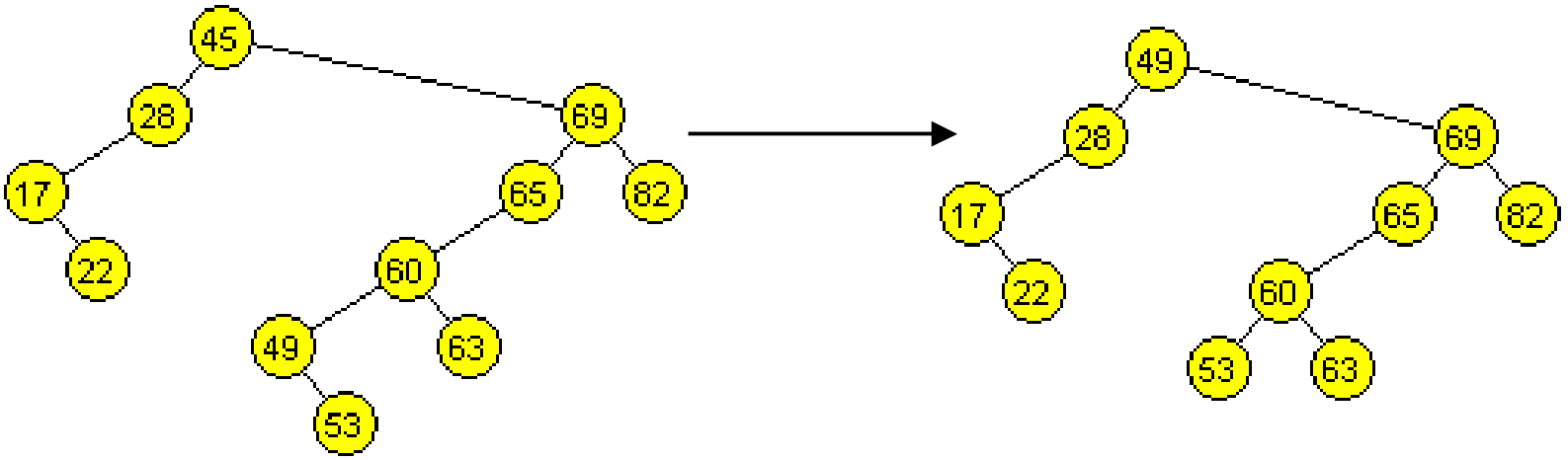


BVS - DELETE

- uzol na odstránenie má dva podstromy
  - nájsť za neho náhradu, skopírovať kľúč z náhr. uzla, odstrániť náhr. uzol, prepojiť koreň podstromu náhr. s rodičom náhrady
  - náhradou je jeho nasledovník, t.j. najmenší (najľavejší) prvok z jeho pravého podstromu –> násl. má pravý alebo žiadny podstrom

    (náhradou môže byť aj jeho predchodca, t.j. najväčší prvok z jeho ľavého podstromu)

Odstránenie uzla 45 – nahradenie 49



BVS – delete (implementácia)

```
btree TREE-DELETE(T,n)
    if LCHILD(n) = nil or RCHILD(n) = nil
            then Y ← n
            else Y ← TREE-SUCCESSOR(n)
    if LCHILD(Y) <> nil
            then X ← LCHILD(Y)
            else X ← RCHILD(Y)
    if X <> nil
            then PARENT(X) ← PARENT(Y)
    if PARENT(Y) = nil
            then ROOT(T) ← X
            else if Y = LCHILD(PARENT(Y))
                        then LCHILD(PARENT(Y)) ← X
                        else  RCHILD(PARENT(Y)) ←X
    if Y <> n
            then DATA(n) ← DATA(Y)
return Y
```

BVS - Nájdenie nasledovníka

```
btree TREE-SUCCESSOR(T)
    if RCHILD(T) <> nil
            then return TREE-MINIMUM(RCHILD(T))
    S ← PARENT(T)
    while S <> nil and T = RCHILD(S)
    do
            T←S
            S ← PARENT(T)
    return S
```

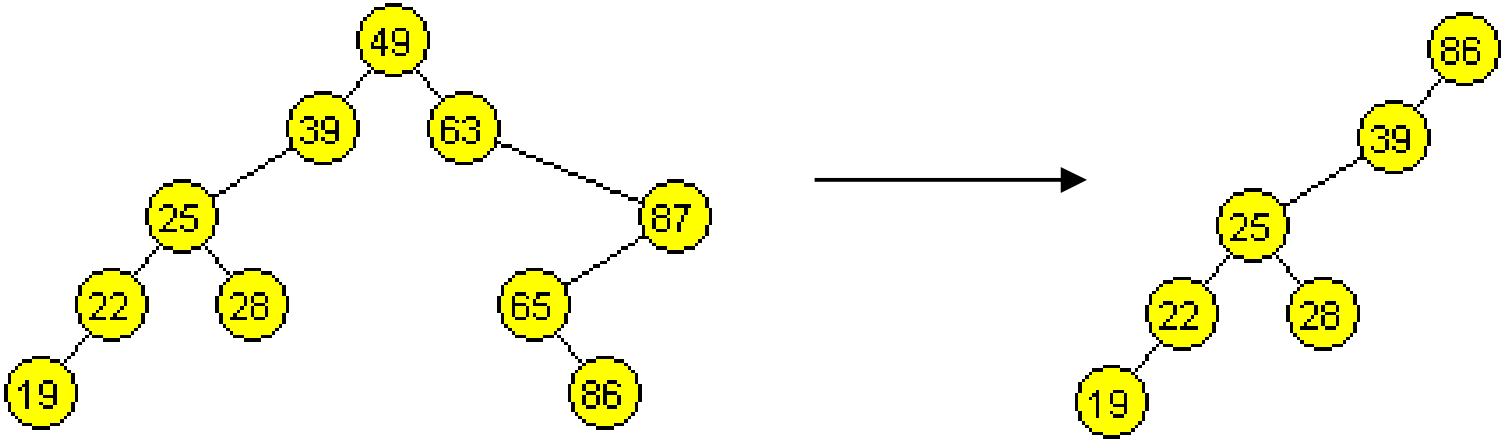
BVS - Nájdenie minima, resp. maxima

```
btree TREE-MINIMUM(T)
    while LCHILD(T) <> nil
          do
                    T ← LCHILD(T)
    return T

btree TREE-MAXIMUM(T)
    while RCHILD(T) <> nil
    do
                    T ← RCHILD(T)
    return T
```

BVS – delete (zložitosť)

- Musíme nájsť uzol, ktorý chceme odstrániť a uzol, ktorý sa stane náhradou – časová zložitosť závisí od hĺbky stromu
- Odstraňovanie uzlov spôsobuje nevyváženosť stromu, pretože vždy vyberáme ako náhradu nasledovníka –> pravý podstrom sa redukuje, ľavý ostáva
  - preto najhorší prípad má zložitosť O(n), ináč v priemere je to O(log n)

Po odstránení uzlov 49,63,65,87,65



BVS – search (implementácia)

Rekurzívna verzia:

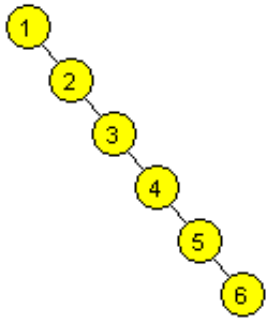
```
btree TREE-SEARCH(T,k)
if T=nil or k=DATA(T)
        then return x
if k<DATA(T)
        then return TREE-SEARCH(LCHILD(T),k)
        else return TREE-SEARCH(RCHILD(T),k)
```

Iteratívna verzia:

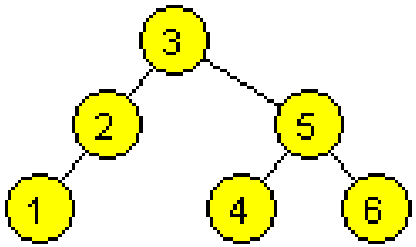
```
btree ITERATIVE-TREE-SEARCH(T,k)
while T <> nil and k<>DATA(T)
        do
                if k<DATA(T)
                        then T ← LCHILD(T)
                        else T ←RCHILD(T)
return T
```

BVS – search (zložitosť)

- Závisí od hĺbky, resp. úplnosti stromu
  - najhoršia zložitosť je O(n)
    - nájdenie uzla 6



  - priemerná a zároveň najlepšia zložitosť je O(log n)
    - nájdenie uzla



BVS – výpis obsahu

- Inorder – usporiadaný výpis obsahu BVS
- Časová zložitosť pre preorder, inorder, postorder je O(n)

BVS – zložitosť

- Operácie search, delete, insert majú najhoršiu časovú zložitosť O(n)
- Na získanie najlepšej zložitosti O(log n) musíme zabezpečiť, že strom po týchto operáciach zostane úplny (dokonale vyvážený) –> použitie samo vyvažovacích stromov ako sú AVL stromy alebo červeno-čierne stromy, ktoré automaticky menia svoje rozloženie tak, aby po týchto operáciách bol rozdiel hĺbok ľavého a pravého podstromu nanajvýš 1