

Údajová abstrakcia

- sústredenie sa na operácie nad údajmi, nie na spôsob, ako ich implementovať v počítači
- príklad: čísla sú abstrakcie
 - definovať množinu čísiel
 - definovať, aké operácie nad nimi
- čísla v počítači
 - špecifikovať, aký interval
 - implementovať operácie

1

Údajový typ

- čísla, znaky, reťazce atď. sa reprezentujú (t.j. zapisujú) ako reťazce bitov
- údajový typ je metóda, ako interpretovať také bitové reťazce
- údajový typ `real` **nie je** množina všetkých reálnych čísiel

2

Abstraktný údajový typ (abstract data type ADT)

- údajový typ ako abstraktný pojem definovaný pomocou množiny vlastností
- určia sa prípustné operácie nad týmto typom
- ADT sa môže implementovať
 - hardvérovo
 - softvérovo

3

Špecifikácia ADT prirodzené číslo

```
structure NATNO
  declare   ZERO() → natno
           ISZERO(natno) → boolean
           SUCC(natno) → natno
           ADD(natno,natno) → natno
           EQ(natno,natno) → boolean
```

Continued ➡

4

Špecifikácia ADT prirodzené číslo

```
for all x,y ∈ 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
```

5

Zásobník (STACK)

6

príklad: vnorené volania

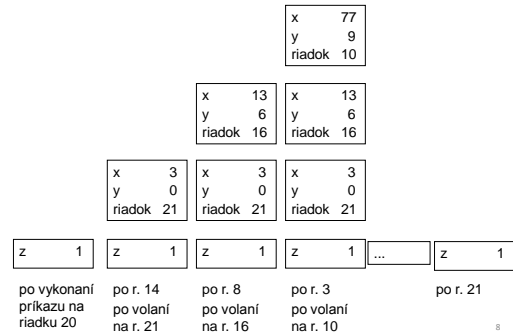
```

1: void nested2(int x)
2: {
3:     int y = 9;
4:     printf("hello2= %d,%d\n",
5:         x, y);
6: }
7: void nested1(int x)
8: {
9:     int y = 6;
10:    printf("hello1= %d, %d\n",
11:        x, y);
12:    nested2(77);
13: }
14: void nested0(int x)
15: {
16:     int y = 0;
17:     printf("hello0= %d, %d\n",
18:         x, y);
19:     nested1(13);
20: }
21: int main(void)
22: {
23:     int z = 1;
24:     nested0(3);
25:     fflush(stdin);
26:     getchar();
27:     return 0;
28: }

```

7

stopovanie vnorených volaní funkcií



8

príklad: rekurzívny algoritmus na určenie dĺžky reťazca

if reťazec je prázdny (nemá žiadne znaky)
dĺžka je 0

else

dĺžka je 1 plus dĺžka reťazca bez prvého znaku

9

príklad: rekurzívny algoritmus na určenie dĺžky reťazca

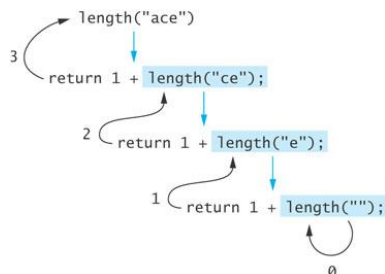
```

/** Recursive method length
 * @param str The string
 * @return The length of the string
 */
public static int length(String str) {
    if (str == null || str.equals(""))
        return 0;
    else
        return 1 + length(str.substring(1));
}

```

10

stopovanie rekurzívneho algoritmu



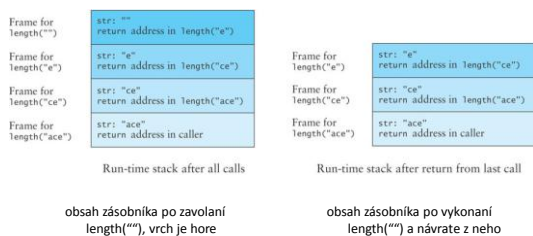
11

aktivačný rámec

- aby vykonanie volaní funkcií, v špeciálnom prípade rekurzívnych volaní fungovalo, treba si uchovať informácie:
 - argumenty funkcie (skutočné parametre)
 - lokálne premenné (hodnoty)
 - adresa, kam sa má vrátiť vykonávanie programu (návrátová adresa pre return)
- vytvorí sa aktivačný rámec a vloží sa do zásobníka (pre každé volanie nový)

12

aktivačné rámce v zásobníku



13

Zásobník

- Pracuje na princípe LIFO (Last In, First Out)
 - Údaje vložené ako posledné budú vyberané ako prvé
- Možné implementácie
 - dynamickou pamäťou
 - poľom

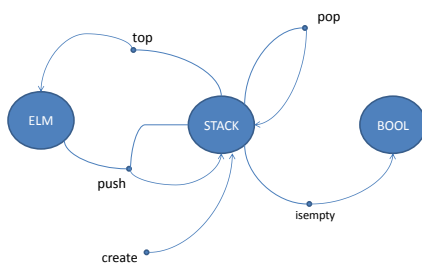
14

Zásobník – formálna špecifikácia

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

15

Zásobník



16

Zásobník – formálna špecifikácia

Pre všetky $S \in \text{stack}$, $i \in \text{elm}$ platí

ISEMPY(CREATE) = true

ISEMPY(PUSH(S,i)) = false

POP(CREATE) = error

POP(PUSH(S,i)) = S

TOP(CREATE) = error

TOP(PUSH(S,i)) = i

17

Implementácia zásobníka pomocou poľa

```

STACK S
CREATE(S)
  top(S) ← 0

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

POP(S)
  if ISEMPY(S)
    then error "underflow"
  else top(S) ← top(S) - 1
  return S
  
```

18

príklad: zápis aritmetických výrazov

Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD

prepis z infixu do postfixu

prezeraj postupne aritmetický výraz zapísaný v infixe zľava doprava

- ak aktuálny znak je operand
 - zapíš znak do výstupného postfixového zápisu výrazu
- ak aktuálny znak je ľavá alebo pravá zátvorka
 - ak znak je "("
 - vlož znak do zásobníka
 - ak znak je ")"
 - opakuj výber znaku zo zásobníka a jeho zapísanie do výstupu pokiaľ sa vyberie "(" zo zásobníka.
- ak aktuálny znak je operátor
 - **krok 1:** zisti, aký znak je na vrchu zásobníka.
 - **krok 2:** ak je zásobník prázdny alebo je na vrchu "(" alebo je na vrchu operátor s menšou precedenciou než je aktuálny znak tak vlož znak do zásobníka.
 - **krok 3:** ak je na vrchu zásobníka operátor s väčšou alebo rovnakou precedenciou ako je aktuálny znak tak vyber operátor zo zásobníka a zapíš ho do výstupu.

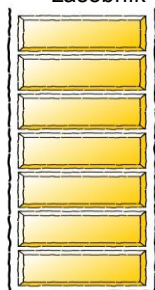
Po prezretí všetkých znakov vstupu vyberaj znak zo zásobníka a zapisuj do výstupu pokiaľ nebude zásobník prázdny.

19

20

prepis z infixu do postfixu

zásobník



infixový výraz

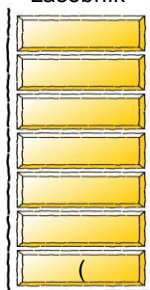
$(a + b - c) * d - (e + f)$

postfixový výraz

21

prepis z infixu do postfixu

zásobník



Infix

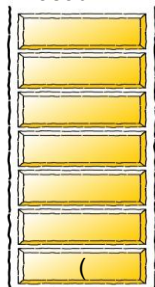
$a + b - c) * d - (e + f)$

Postfix

22

prepis z infixu do postfixu

zásobník



Infix

$+ b - c) * d - (e + f)$

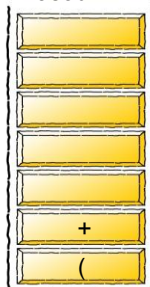
Postfix

a

23

prepis z infixu do postfixu

zásobník



Infix

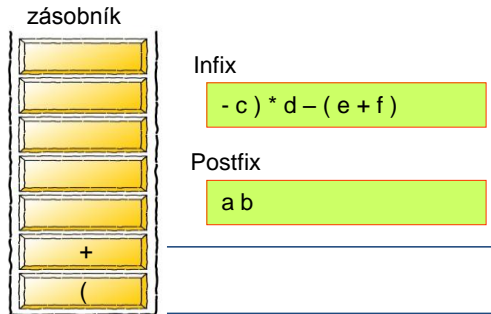
$b - c) * d - (e + f)$

Postfix

a

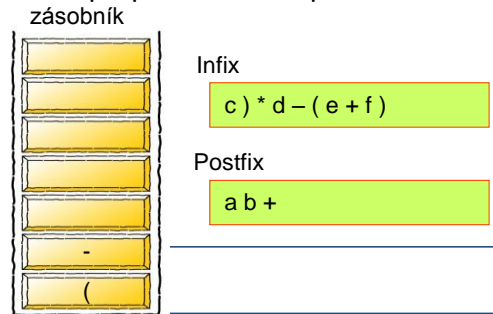
24

prepis z infixu do postfixu



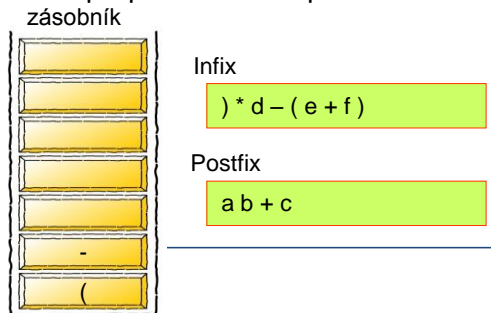
25

prepis z infixu do postfixu



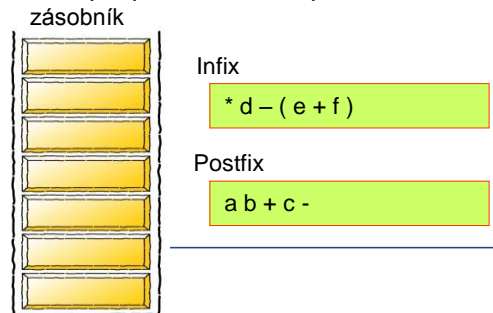
26

prepis z infixu do postfixu



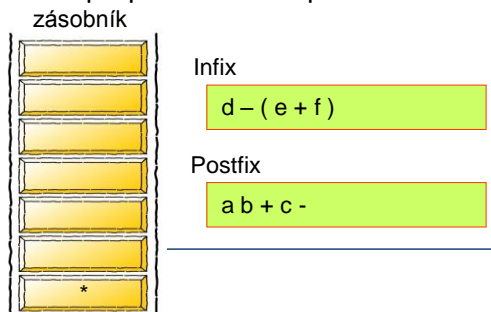
27

prepis z infixu do postfixu



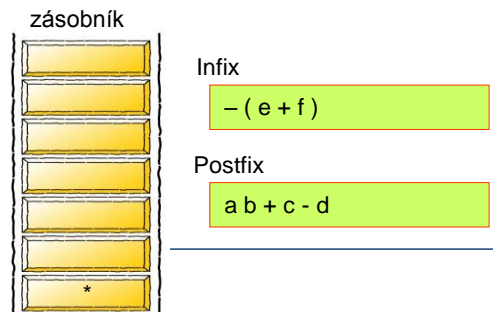
28

prepis z infixu do postfixu



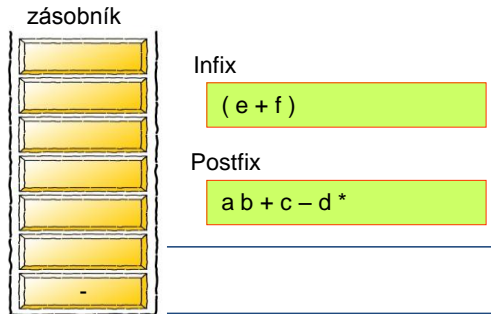
29

prepis z infixu do postfixu



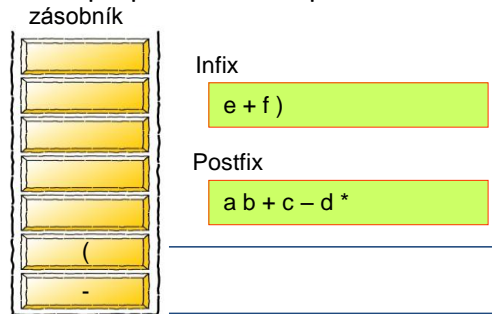
30

prepis z infixu do postfixu



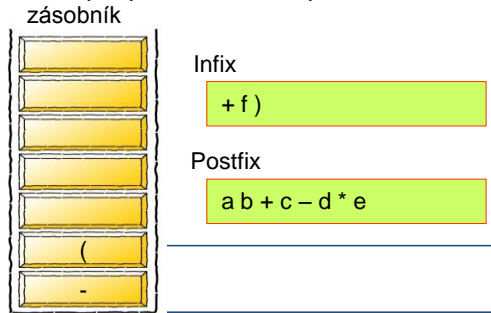
31

prepis z infixu do postfixu



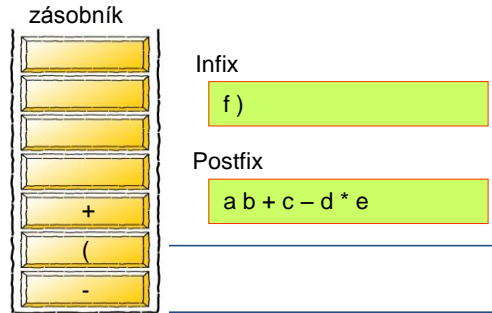
32

prepis z infixu do postfixu



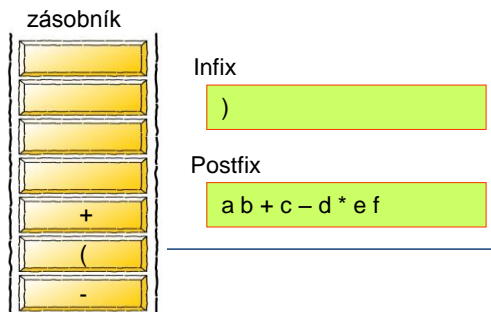
33

prepis z infixu do postfixu



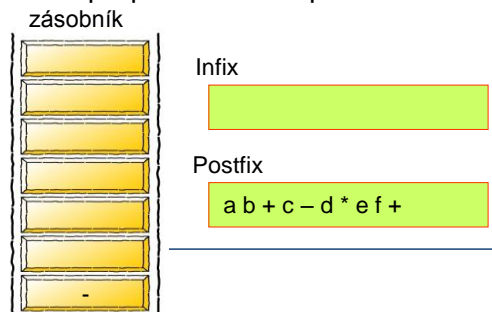
34

prepis z infixu do postfixu



35

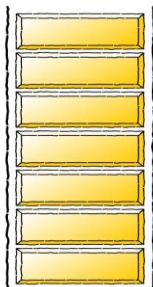
prepis z infixu do postfixu



36

prepis z infixu do postfixu

zásobník



Infix

Postfix

$a b + c - d * e f + -$

príklad: výpočet hodnoty aritmetického výrazu
zapísaného v postfixe

prezeraj postupne aritmetický výraz zapísaný v postfixe na
vstupe zľava doprava

- ak aktuálny znak je čísla
 - vlož ho do zásobníka
- ak aktuálny znak je operátor (+, -, *, /)
 - vyber dve položky zo zásobníka
 - vykonaj operáciu určenú znakom operátora
 - vlož výsledok operácie do zásobníka

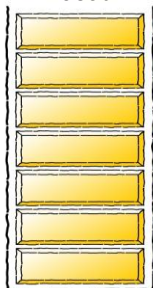
Po prezretí všetkých znakov vstupu vyberaj znak zo zásobníka a zapisuj do
výstupu pokiaľ nebude zásobník prázdny.

37

38

príklad: vyhodnotenie postfixového výrazu

zásobník



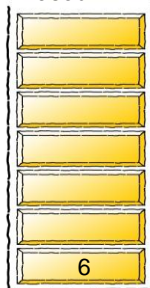
Postfix

$6 5 2 3 + 8 * + 3 + *$

$\square \square \square = \square$

príklad: vyhodnotenie postfixového výrazu

zásobník



Postfix Expression

$5 2 3 + 8 * + 3 + *$

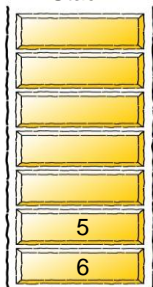
$\square \square \square = \square$

39

40

príklad: vyhodnotenie postfixového výrazu

Stack

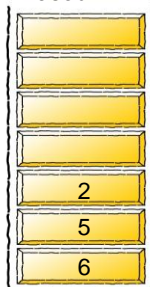


Postfix

$2 3 + 8 * + 3 + *$

príklad: vyhodnotenie postfixového výrazu

zásobník



Postfix Expression

$3 + 8 * + 3 + *$

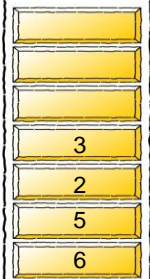
$\square \square \square = \square$

41

42

príklad: vyhodnotenie postfixového výrazu

Stack



Postfix

$$+ 8 * + 3 + *$$

$$\square \square \square = \square$$

43

príklad: vyhodnotenie postfixového výrazu
zásobník

Postfix

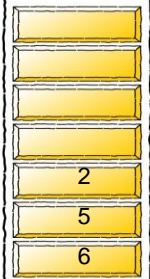
$$8 * + 3 + *$$

$$\square + \square = \square$$

44

príklad: vyhodnotenie postfixového výrazu

Stack



Postfix

$$8 * + 3 + *$$

$$\square + 3 = \square$$

45

príklad: vyhodnotenie postfixového výrazu

zásobník

Postfix

$$8 * + 3 + *$$

$$2 + 3 = \square$$

46

príklad: vyhodnotenie postfixového výrazu

zásobník

Postfix

$$8 * + 3 + *$$

$$2 + 3 = 5$$

47

príklad: vyhodnotenie postfixového výrazu

zásobník

Postfix

$$8 * + 3 + *$$

$$\square \square \square = \square$$

48

príklad: vyhodnotenie postfixového výrazu
zásobník

8
5
5
6

Postfix

$* + 3 + *$

$\square \square \square = \square$

49

príklad: vyhodnotenie postfixového výrazu
zásobník

8
5
5
6

Postfix

$+ 3 + *$

$\square * \square = \square$

50

príklad: vyhodnotenie postfixového výrazu
zásobník

5
5
6

Postfix

$+ 3 + *$

$\square * 8 = \square$

51

príklad: vyhodnotenie postfixového výrazu
zásobník

5
6

Postfix

$+ 3 + *$

$5 * 8 = \square$

52

príklad: vyhodnotenie postfixového výrazu
zásobník

5
6

Postfix

$+ 3 + *$

$5 * 8 = 40$

53

príklad: vyhodnotenie postfixového výrazu
zásobník

40
5
6

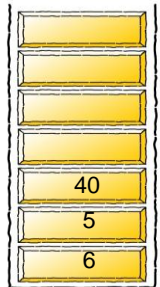
Postfix

$+ 3 + *$

$\square \square \square = \square$

54

príklad: vyhodnotenie postfixového výrazu
zásobník



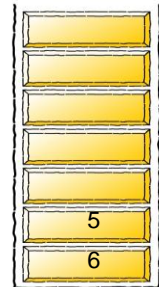
Postfix

$3 + *$

$\square + \square = \square$

55

príklad: vyhodnotenie postfixového výrazu
zásobník



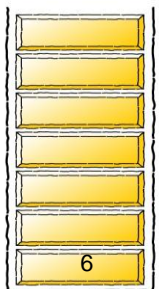
Postfix

$3 + *$

$\square + 40 = \square$

56

príklad: vyhodnotenie postfixového výrazu
zásobník



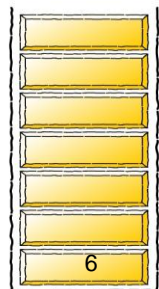
Postfix

$3 + *$

$5 + 40 = \square$

57

príklad: vyhodnotenie postfixového výrazu
zásobník



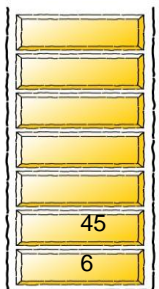
Postfix

$3 + *$

$5 + 40 = 45$

58

príklad: vyhodnotenie postfixového výrazu
zásobník



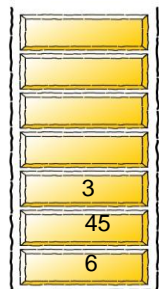
Postfix

$3 + *$

$\square \square = \square$

59

príklad: vyhodnotenie postfixového výrazu
zásobník



Postfix

$+ *$

$\square \square \square = \square$

60

príklad: vyhodnotenie postfixového výrazu
zásobník

Postfix

*

□ + □ = □

61

príklad: vyhodnotenie postfixového výrazu
zásobník

Postfix

*

□ + 3 = □

62

príklad: vyhodnotenie postfixového výrazu
zásobník

Postfix

*

45 + 3 = □

63

príklad: vyhodnotenie postfixového výrazu
zásobník

Postfix

*

45 + 3 = 48

64

príklad: vyhodnotenie postfixového výrazu
zásobník

Postfix

*

□ □ □ = □

65

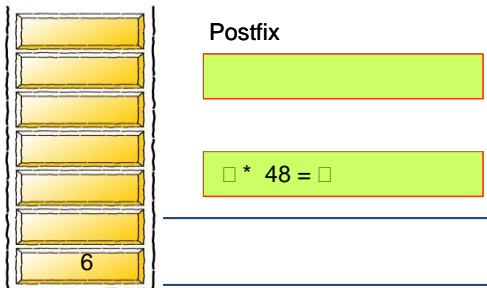
príklad: vyhodnotenie postfixového výrazu
zásobník

Postfix

□ * □ = □

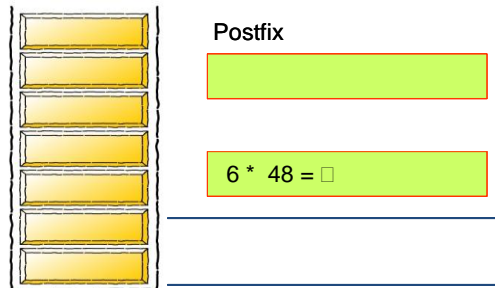
66

príklad: vyhodnotenie postfixového výrazu
zásobník



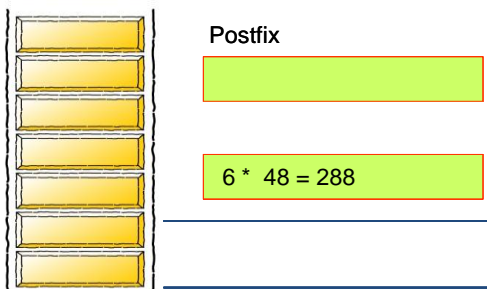
67

príklad: vyhodnotenie postfixového výrazu
zásobník



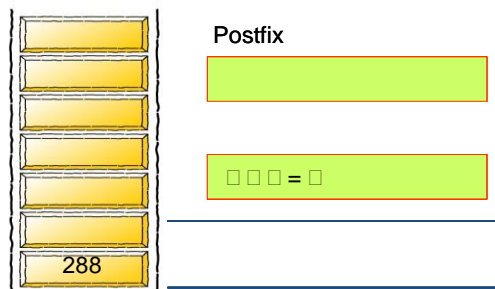
68

príklad: vyhodnotenie postfixového výrazu
zásobník



69

príklad: vyhodnotenie postfixového výrazu
zásobník



70

Implementácia zásobníka pomocou poľa (dokončenie)

```
TOP(S)
  if ISEMPY(S)
    then error "underflow"
    else return S[top(S)]
```

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

71

jednosmerne zreťazený zoznam (Singly Linked List SLL) začiatok

72

jednosmerne zreťazený zoznam (Singly Linked List SLL)

- Najjednoduchšia reprezentácia lineárneho spájaného zoznamu
- Každý prvok obsahuje údajovú časť a ukazovateľ na ďalší prvok
- Ukazovateľ na ďalší prvok posledného prvku ukazuje na NULL

73

jednosmerne zreťazený zoznam

- Základné operácie:
 - CREATE: vytvorenie prázdneho SLL
 - ISEMPY: test na prázdnosť
 - INSERT: vloženie prvku
 - DELETE: vymazanie prvku
 - FIND: nájdenie prvku
- Ďalšie operácie:
 - DELETE_ALL, NUM_ELEMENTS, ...

74

SLL - Reprezentácia

```

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



75

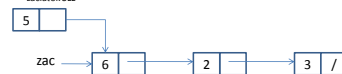
SLL insert

Insert 5:

1. Krok : Vytvorenie nového prvku



2. Krok : Priradenie smerníka a hodnoty novovytvorenému prvku
Nový prvok bude ukazovať na to isté miesto v pamäti (na ten istý prvok) ako ukazuje začiatok SLL



3. Krok : Nastavenie nového začiatku SLL

Začiatok SLL bude ukazovať na nový prvok

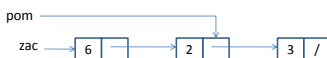


76

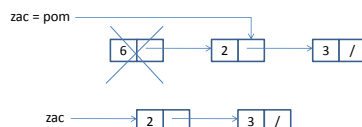
SLL delete

Delete:

1. Krok: Pomocnej premennej sa priradí ukazovateľ na ďalší prvok prvého prvku



2. Krok : Vymazanie prvého prvku a nastavenie začiatku SLL
Začiatku SLL sa priradí ukazovateľ uložený v pomocnej premennej



77

jednosmerne zreťazený zoznam (Singly Linked List SLL) *prerušenie*

78

Implementácia zásobníka pomocou SLL

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

STACK zasobnik;

STACK CREATE()
{
    SLL_create( zasobnik );
}

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

79

Implementácia zásobníka pomocou SLL

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

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

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

80

FRONT (QUEUE)

FRONT

- Pracuje na princípe FIFO (First In, First Out)
 - Údaje vložené ako prvé budú vyberané ako prvé
- Možné implementácie
 - dynamickou pamäťou
 - poľom (vektorm)

81

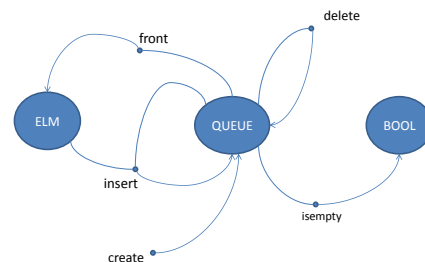
82

Front – formálna špecifikácia

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

83

Front



84

Front – formálna špecifikácia

pre všetky $Q \in \text{queue}$, $i \in \text{elm}$ platí
 $\text{ISEMPTY}(\text{CREATE}) = \text{true}$
 $\text{ISEMPTY}(\text{INSERT}(Q,i)) = \text{false}$
 $\text{DELETE}(\text{CREATE}) = \text{error}$
 $\text{DELETE}(\text{INSERT}(Q,i)) =$
 if $\text{ISEMPTY}(Q)$ then CREATE
 else $\text{INSERT}(\text{DELETE}(Q),i)$
 $\text{FRONT}(\text{CREATE}) = \text{error}$
 $\text{FRONT}(\text{INSERT}(Q,i)) =$
 if $\text{ISEMPTY}(Q)$ then i
 else $\text{FRONT}(Q)$

85

Front

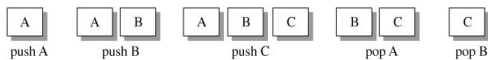
- Front je zoznam prvkov, v ktorom je možné pristupovať iba k prvému a poslednému prvku. Nový prvok sa vkladá na koniec. Pri výbere sa vyberá zo začiatku.



86

Front - operácie

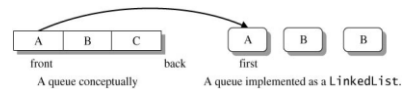
$\text{insert}(\text{item})$ – vloženie prvku na koniec, na obrázku označené push
 $\text{pop}()$ – výber prvku zo začiatku
 $\text{front}()$ – vráti hodnotu prvého prvku



87

Front – implementácia v java

- Front sa môže byť implementovať pomocou existujúcej triedy `LinkedList`



88

Front – implementácia v java

```
public class LinkedListQueue<T> implements Queue<T>
{
    private LinkedList<T> qlist = null;
    public LinkedListQueue ()
    {
        qlist = new LinkedList<T>();
    }
    . . .
}
```

89

Front – implementácia v java metóda `pop()`

```
public T pop()
{
    // ak je front prázdny - chyba
    if (isEmpty())
        throw new NoSuchElementException(
            "LinkedListQueue pop(): queue empty");

    // vráti prvý element
    return qlist.removeFirst();
}
```

90

Implementácia frontu pomocou SLL

```
typedef struct uzo1 *SLL_UZOL;
struct uzo1
{
    SLL_TYP   prvok;      //prvok typu SLL_TYP
    SLL_UZOL  n;          //nasledovnik
}
typedef struct hlavicka
{
    SLL_UZOL  zac, kon;    // smerniky začiatku a konca
} F_HLAV;

F_HLAV h; // smernik na front

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

91

Implementácia frontu pomocou SLL

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

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

92

Implementácia frontu pomocou SLL

```
F_HLAV INSERT(F_HLAV h, SLL_TYP hodnota)
{
    SLL_UZOL  pom;
    pom = (SLL_UZOL) malloc( sizeof(uzo1));
    pom->prvok = hodnota;
    pom->n = NULL;
    if( ISEMPY(h)) { h.kon = h.zac = pom; }
    else // Pridaj na koniec
    {
        h.kon->n = pom;
        h.kon = pom;
    }
    return h;
}
```

93

Implementácia frontu pomocou SLL

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

94

Implementácia frontu pomocou poľa

QUEUE Q

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

95

Implementácia frontu pomocou poľa

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)]

ISEMPY(Q)

return head(Q) = tail(Q)

96

Ohraničený front

- Front, ktorý pozostáva najviac z daného počtu elementov. Nový prvok sa môže vložiť, len keď front nie je plný.
- poznámka: **toto je zmena špecifikácie!**
- Funkcia `bool full()` určuje, či je front plný
- Implementácia je možná napríklad pomocou poľa (vektora)

97

BQueue príklad

```
BQueue<Integer> q = new BQueue<Integer>(15);
int i;

// naplnenie frontu
for (i=1; !q.full(); i++)
    q.push(i);

System.out.println(q.peek() + " " + q.size());

try
{
    q.push(40); // exception
}

catch (IndexOutOfBoundsException iobe)
{ System.out.println(iobe); }
```

98

BQueue príklad

```
Output:
1 15
java.lang.IndexOutOfBoundsException: BQueue push(): queue full
```

99

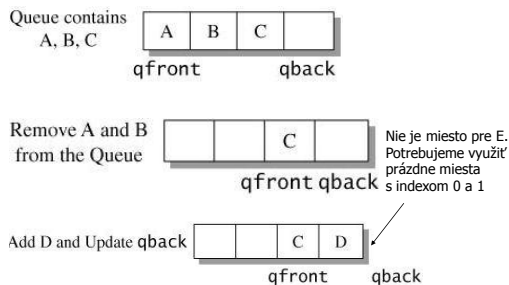
BQueue implementácia

```
public class BQueue<T> implements Queue<T>
{
    private T[] queueArray;
    private int qfront, qback;
    private int qcapacity, qcount;

    public BQueue(int size)
    {
        qcapacity = size;
        queueArray = (T[])new Object[qcapacity];
        qfront = 0;
        qback = 0;
        qcount = 0;
    }
}
```

100

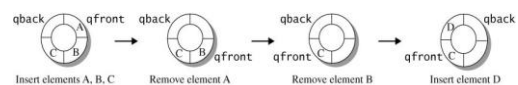
BQueue implementácia



101

BQueue implementácia

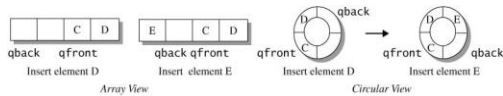
- Jedným z riešení ako sa vyhnúť problému s ohraničením je vytvorenie kruhového vektora.
- Prvky sa vkladajú v smere hodinových ručičiek



102

BQueue implementácia

- Vytvorenie kruhového vektora vyžaduje pravidelné aktualizovanie začiatkovej a koncovkej pozície frontu pri každej zmene (vložení, výbere prvku).



Move qback forward: $qback = (qback + 1) \% qcapacity$;
 Move qfront forward: $qfront = (qfront + 1) \% qcapacity$;

103

BQueue implementácia metóda full()

```
public boolean full()
{
    return qcount == qcapacity;
}
```

104

BQueue implementácia metóda push(item)

```
public void push(T item)
{
    if (qcount == qcapacity)
        throw new IndexOutOfBoundsException(
            "BQueue push(): queue full");

    queueArray[qback] = item;
    qback = (qback+1) % qcapacity;

    qcount++;
}
```

105

BQueue implementácia metóda pop()

```
public T pop()
{
    if (count == 0)
        throw new NoSuchElementException(
            "BQueue pop(): empty queue");

    T queueFront = queueArray[qfront];

    qfront = (qfront+1) % qcapacity;

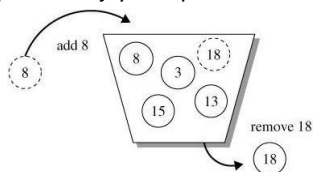
    qcount--;

    return queueFront;
}
```

106

Prioritný front

- Prioritný front je zoznam prvkov, ktorým je pridelená priorita – je ich možné porovnávať. Prvky je možné vkladať v akomkoľvek poradí s rôznou prioritou avšak pri výbere sa vyberá vždy prvok s najvyššou prioritou.



107

jednosmerne zreťazený zoznam (Singly Linked List SLL)

pokračovanie

108

jednosmerne zreťazený zoznam

- Základné operácie:
 - CREATE: vytvorenie prázdneho SLL
 - ISEMPY: test na prázdnosť
 - INSERT: vloženie prvku
 - DELETE: vymazanie prvku
 - FIND: nájdenie prvku
- Ďalšie operácie:
 - DELETE_ALL, NUM_ELEMENTS, ...

109

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;
}

```

110

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;
}

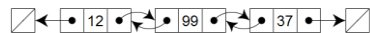
void SLL_all_elements( SLL_UZOL smernik )
{
    for( ; !SLL_isempty(smernik); smernik = smernik->n )
        printf( "%d \n", smernik->prvok );
}

```

111

Iné druhy spájaného zoznamu

- Obojsmerne spájaný zoznam
 - Každý prvok obsahuje ukazovateľ na ďalší prvok a aj na predchádzajúci prvok
- Cyklický spájaný zoznam
 - Posledný prvok zoznamu ukazuje na prvý prvok



112

Zreťazená voľná pamäť aka dynamická pamäť

113

Zreťazená voľná pamäť

- Predstavuje základný abstraktný typ, ktorý sa využíva pri implementácii 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 (ak je voľný), alebo dátovú časť (ak nie je voľný)
- ZVP obsahuje okrem poľa prvkov ešte aj informáciu o prvom voľnom prvku

114

ZVP – Formálna špecifikácia

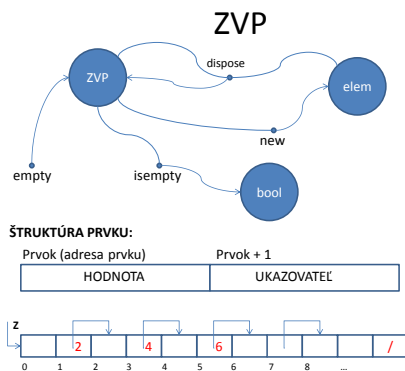
- CREATE() -> zvp
- NEW(zvp) -> item //a.k.a. malloc
- DISPOSE(zvp, item) -> zvp //a.k. free
- ISEMPY(zvp) -> bool

Pre všetky $Z \in \text{zvp}$, $i \in \text{item}$ platí

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

115

116



117

ZVP – príklad implementácie

```

VAR ZVP: IND;

Function EMPTY(VAR ZVP: IND) //vyhodenie zvp, vykoná sa požadované premenenie volajúcich prvkov
VAR
  UK: IND;
BEGIN
  UK := ADDRMIN;
  WHILE UK < ADDRMAX-2 DO
    BEGIN
      PARAM[UK+1] := UK+2;
      UK := UK+2;
    END;
  PARAM[UK+1] := NIL;
  ZVP := ADDRMIN;
END;

Function NEW(Z: VAR ZVP: IND; VAR PRVOK: IND) //operácia poskytnutia prvku zo ZVP
BEGIN
  IF EMPTY(ZVP)
    THEN WRITELN("Chyba: vyčerpanie pamäte");
  ELSE
    BEGIN
      PRVOK := ZVP;
      ZVP := PARAM[ZVP+1];
    END;
  END;
END;

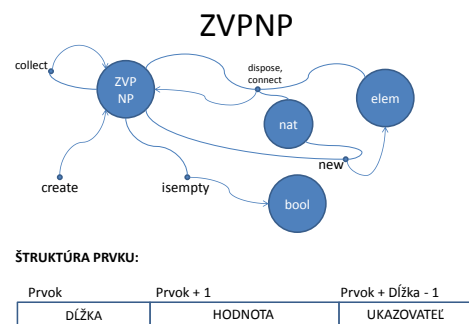
```

118

ZVP nerovnakých prvkov

- Druhy: ZVPNP, ELM, NAT, BOOL
- Operácie:
 - CREATE() -> ZVPNP
 - NEW(ZVPNP, NAT) -> ELM //a.k.a. malloc
 - DISPOSE(ZVPNP, NAT, ELEM) -> ZVPNP //a.k. free
 - CONNECT(ZVPNP, NAT, ELM) -> ZVPNP (spojenie 2 prvkov)
 - COLLECT(ZVPNP) -> ZVPNP (spojenie do súvislej oblasti)
 - ISEMPY (ZVPNP) -> BOOL

119



120

príklad pridelenia zreťazenej voľnej pamäte pre 3 zásobníky

```

STACK z1, z2, z3;
...
PUSH(z1, 3);
...
PUSH(z3, 4);
...
PUSH(z2, 2);
...
PUSH(z1, 1);
...
PUSH(z3, 5);

```

121

