Údajová abstrakcia

- 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

Údajový typ

- 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

Abstraktný údajový typ

- 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

Špecifikácia ADT prirodzené číslo

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

Continued ➔

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

Zásobník (STACK)

Zásobník

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

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 ∈ stack, i ∈ elm platí

ISEMPTY(CREATE) = true

ISEMPTY(PUSH(S,i)) = false

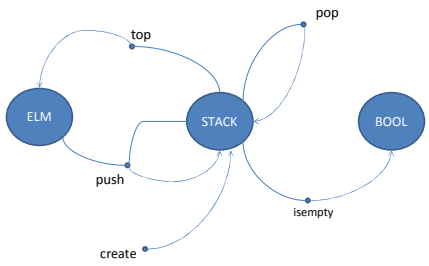
POP(CREATE) = error

POP(PUSH(S,i)) = S

TOP(CREATE) = error

TOP(PUSH(S,i)) = i

Zásobník



Implementácia zásobníka pomocou poľa

```
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
```

Implementácia zásobníka pomocou poľa

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

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

SLL

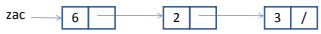
- Najjednoduchšia reprezentácia lineárneho spájaného zoznamu
- Každý prvok obsahuje dátovú časť a ukazovateľ na ďalší prvok
- Ukazovateľ posledného prvku ukazuje na NULL

Singly Linked List

- Základné operácie:
  - CREATE: vytvorenie prázdneho SLL
  - ISEMPTY: test na prázdnosť
  - 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
```



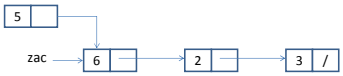
SLL insert

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 isty prvok) ako ukazuje začiatok SLL



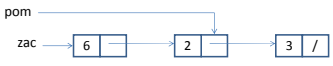
3. Krok : Nastavenie nového začiatku SLL

Začiatok SLL bude ukazovať na nový prvok

SLL delete

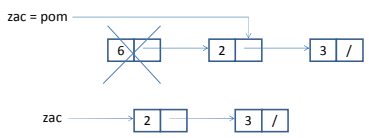
Delete:

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 );
}
```

Zásobník

FRONT (QUEUE)

FRONT

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

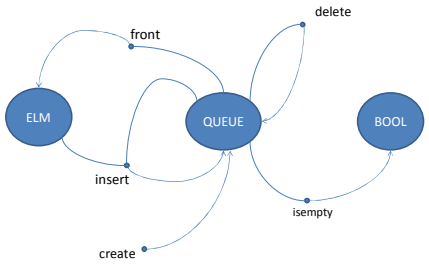
Front – formálna špecifikácia

- 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
  - ISEMPTY(QUEUE) -> BOOL //test na prázdnosť

Front – formálna špecifikácia

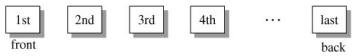
pre všetky Q ∈ queue, i ∈ elm platí
ISEMPTY(CREATE) = true
ISEMPTY(INSERT(Q,i)) = false
DELETE(CREATE) = error
DELETE(INSERT(Q,i)) =
if ISEMPTY(Q) then CREATE
else INSERT(DELETE(Q),i)
FRONT(CREATE) = error
FRONT(INSERT(Q,i)) =
if ISEMPTY(Q) then i
else FRONT(Q)

Front



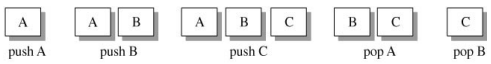
Front

- 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.



Front - operácie

push(item) – vloženie prvku na koniec
pop() – výber prvku zo začiatku
front() – vráti hodnotu prvého prvku

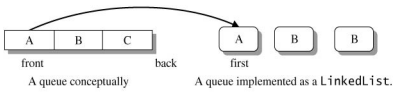


Front

- Front používa FIFO (first-in-first-out) disciplínu (prvý vložený prvok sa prvý vyberie z frontu)

Front – implementácia v jave

- Front sa môže byť implementovať pomocou existujúcej triedy LinkedList



Front – implementácia v jave

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

Front – implementácia v jave metóda pop()

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

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

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;
}
```

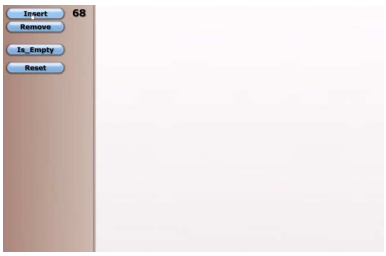
Implementácia frontu pomocou poľa

```
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
```

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)]
ISEMPTY(Q)
   return head(Q) = tail(Q)
```

FRONT



Ohraničený front

- Front, ktorý pozostáva najviac z daného počtu elementov. Nový prvok sa môže vložiť, len keď front nie je plný.
- Funkcia bool full() určuje, či je front plný
- Implementácia je možná napríklad pomocou poľa

BQueue príklad

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

// naplnenie fronty
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); }
```

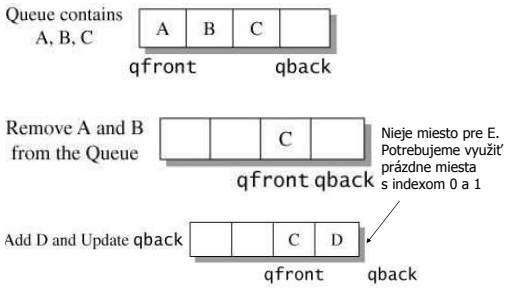
BQueue príklad

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

BQueue implementácia

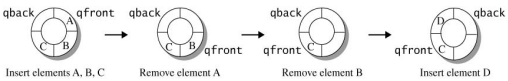
```
public class BQueue<T> implements Queue<T>
{
   private T[] queueArray;
   private int qfront, qback;
   private int qcapacity, qcount;

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

BQueue implementácia



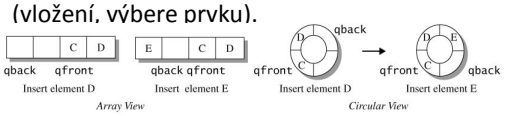
BQueue implementácia

- Jedným s riešení ako sa vyhnúť problému s ohraničením je vytvorenie cyklickej sekvencie.
- Prvky sa vkladajú v smere hodinových ručičiek



BQueue implementácia

- Vytvorenie cyklickej sekvencie vyžaduje pravidelné aktualizovanie začiatočnej a koncovej pozície frontu pri každej zmene (vložení, výbere prvku).




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

BQueue implementácia metóda full()

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

BQueue implementácia metóda push(item)

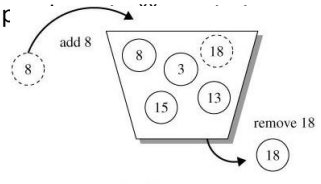
```
public void push(T item)
{
   if (qcount == qcapacity)
      throw new IndexOutOfBoundsException(
            "BQueue push(): queue full");

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

   qcount++;
}
```

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;
}
```

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 p



Singly Linked List (SLL)

SLL

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

Singly Linked List

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

SLL implementácia

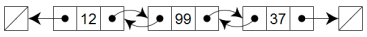
```
SLL_UZOL SLL_create ( SLL_UZOL zac)
{
        zac = NULL;
        return zac;
}

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

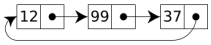
```
SLL_UZOL SLL_delete( SLL_UZOL smernik )
{
        SLL_UZOL    pom;
        if(! SLL_isempty ( smernik ) )
        {
                pom = smernik->n;
                free( smernik);
                smernik = pom;
        }
        return smernik;
}

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

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

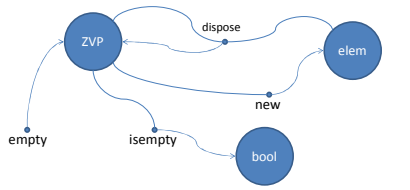
ZVP – Formálna špecifikácia

- sll = single link list
- CREATE() -> sll
- NEW(sll) -> item
- DISPOSE(sll, item) -> sll
- ISEMPTY(sll) -> bool

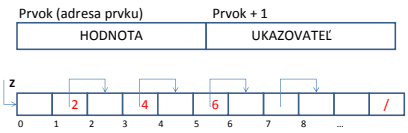
Pre všetky Z ∈ sll, i ∈ item platí

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

ZVP



ŠTRUKTÚRA PRVKU:



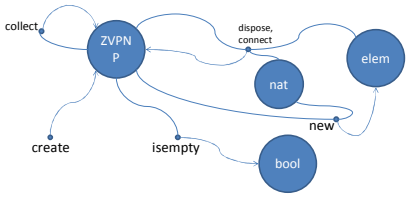
ZVP – príklad implementácie

```
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)