Abstraktní datové typy

- Pokud nabízíme omezené rozhraní datového typu pomocí specifikované sady operací, nazýváme takový typ abstraktní datový typ (ADT). Konkrétní struktura nového datového typu je totiž pro jeho uživatele (=programátora) překryta vrstvou funkcí/procedur, které s tímto typem pracují. Uživatel pak získává na tento typ mnohem abstraktnější pohled.
- Úplná abstrakce (ukrytí jednotlivých datových položek nového typu před uživatelem) je možná až v objektově orientovaných jazycích – v neobjektovém Pascalu nebo ANSI C není možnost, jak toto udělat. V neobjektových jazycích sice uživatel vidí „dovnitř“ daného ADT, ale je v jeho vlastním zájmu, aby k operacím s tímto typem používal poskytovaného rozhraní procedur a funkcí – v opačném případě se vystavuje riziku, že jeho zásah do datových položek způsobí chybu.
- Další výhodou použití abstraktních datových typu, a tedy především oddělení rozhraní datového typu od jeho implementace, je to, ze můžeme vytvořit různé implementace (např. nové verze) téhož datového typu, aniž se tato změna projeví při jeho používání.

Základní ADT a časové složitosti jejich operací

	Vkládání	Mazání	Vyhledávání	Řazení
Dynamické pole	Pokud je buňka volná: O(1) Jinak: O(N)	O(1)	O(N)	$O(N.\log_2 N)$
Lineární seznam	O(1)	O(1)	O(N)	$O(N.\log_2 N)$
Binární strom	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$	Vždy seřazen
Hash tabulka	O(1)	O(1) až O(K)	O(1) až O(K)	Lze jen v kombinaci s jiným ADS

ADT Dynamické pole

- Základní funkcí tohoto ADT je realokace paměti při potřebě zvětšit kapacitu pole.
- Realokace je drahá operace, protože v případě, kdy je blok za realokovaným blokem již obsazen:
  - musí se obsah kopírovat do nově alokovaného bloku.
  - Zvětšuje se fragmentace dynamické paměti, ve výsledku může proces vyčerpat veškerou volnou paměť, přestože jeho heap obsahuje spoustu volných bloků
- Řešení:
  - alokace dostatečně velkých bloků paměti
  - Nebo řetězení alokovaných bloků do seznamu
- V STL C++ je dynamické pole hotové v std::vector nebo wxArray, v Javě ve třídě Vector a ArrayList, ...

Implementace dynamického pole

```
typedef struct _Vector {
  tData * data;     /**< interní pole */
  int capacity;     /**< velikost pole jako max. počet prvků  */
  int max_index;    /**< největší zatím obsazený index pole */
} Vector;
```

- **void vector_new**(Vector * v, **int** capacity);
  Alokuje paměť pro strukturu Vector, nastaví jí správně capacity a size, alokuje paměť pro interní pole o velikosti "capacity"
- **void vector_put**(Vector * v, **int** index, tData d);
  Uloží do Vektoru novou položku na daný index.
  Pokud je index >= capacity, realokuje paměť pro vnitřní pole tak,aby nová větší velikost byla celým násobkem minimální velikosti bloku.
  Pokud je index > size, nastaví size = index.
- tData **vector_get**(Vector * v, **int** index);
  Získá z vektoru data na daném indexu.
  Pokud je zadaný index > size, vypíše chybu a končí (exit).
- **inline** size_t **vector_max_index**(Vector * v);

Příklad implementace metody

```
const size_t Vector_Min_Alloc=1024; /**< Min. počet bajtů alok. bloku */

/*Vypočte kapacitu tak, aby byla násobkem min. velikosti alok. bloku.*/
inline int vector_adjust_capacity(int capacity) {
  int blocks;
  if (capacity < Vector_Min_Alloc)
    return Vector_Min_Alloc;
  else {
    blocks=capacity/Vector_Min_Alloc;
    return (blocks+1)*Vector_Min_Alloc;
  }
}

Vector * vector_new(int capacity) {
  Vector * v = malloc(sizeof(Vector));
  capacity = v->capacity = vector_adjust_capacity(capacity);
  v->max_index = 0;
  v->data = calloc(capacity, sizeof(tData));
  return v;
}
```

ADT Lineární seznam

- **Lineární seznam** (anglicky list) je datová struktura, která představuje posloupnost položek.
- Seznam se nejčastěji implementuje tak, že každá položka obsahuje odkaz nebo ukazatel na následující prvek = jednosměrný seznam
- Pokud položka obsahuje i ukazatel na předchozí prvek, jde o obousměrný seznam
- Pokud poslední položka ukazuje na první, jedná se o kruhový seznam (v případě obousměrného seznamu i první položka ukazuje na poslední).
- Samotný ADT lineární seznam je nejčastěji deklarován jako struktura se dvěma položkami – ukazatel na první prvek seznamu a ukazatel na aktivní prvek seznamu. Někdy bývá deklarován i ukazatel na poslední prvek (např. u obousměrného seznamu)
- Seznam může být i prázdný - nemusí obsahovat žádný prvek.
- Seznamy obvykle používáme jako dynamické datové struktury a jednotlivé prvky seznamu podle potřeby dynamicky alokujeme nebo rušíme.

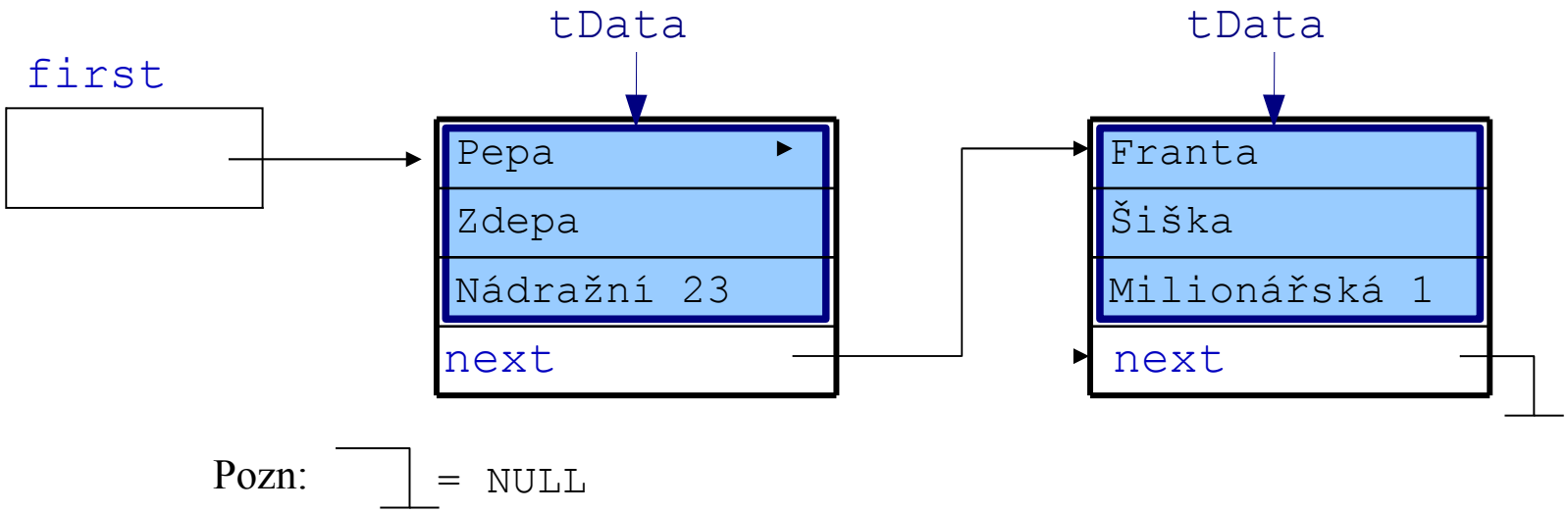
ADT Lineární seznam

- Deklarace typu Jednosměrný lineární seznam:

```
typedef struct _ListNode
{    tData data;
     struct _ListNode * next;
} ListNode;

typedef struct {
  ListNode * first;
  ListNode * active;
} List;
```

- Po vložení několika prvků do seznamu bude seznam vypadat v paměti takto:



ADT Lineární seznam

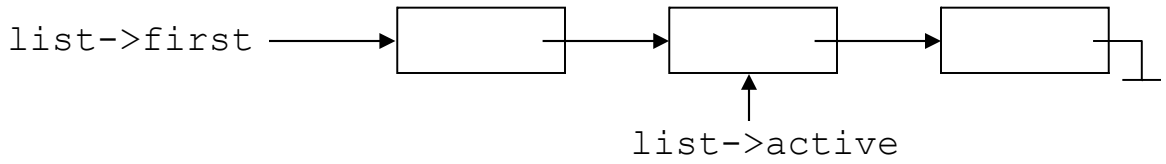
Příklad sady metod pro práci s lineárním seznamem:

```
void list_init(List * list);
```
Provede inicializaci seznamu před jeho prvním použitím.
```
void list_insert_first(List * list, tData data);
```
Vloží prvek na začátek seznamu.
```
void list_seek_first(List * list);
```
Nastaví aktivitu seznamu na první prvek.
```
tData list_get_first_data(List * list);
```
Vrátí hodnotu prvního prvku. Pokud je seznam prázdný, volá proceduru Error.
```
void list_delete_first(List * list);
```
Maže 1. prvek. Pokud byl aktivní, aktivita se ztrácí. Pokud byl seznam prázdný, nedělá nic.
```
void list_delete_next(List * list);
```
Maže první prvek seznamu za aktivním prvkem. Pokud nebyl seznam aktivní, nic se neděje.
```
void list_insert_next(List * list, tData data);
```
Vloží prvek za aktivní. Pokud nebyl seznam aktivní, nic se neděje.
```
tData list_get_next_data(List * list);
```
Vrátí hodnotu aktivního prvku. Pokud seznam není aktivní, volá se procedura Error.
```
void list_set_active_data(List* list, tData data);
```
přepíše obsah aktivní položky. Pokud není žádná položka aktivní, nedělá nic.
```
void list_next(List * list);
```
posune aktivitu na následující prvek seznamu.
```
bool list_is_active(List * list);
```
Je-li seznam aktivní, vrací True. V opačném případě vrací false.

Příklad implementace jedné metody

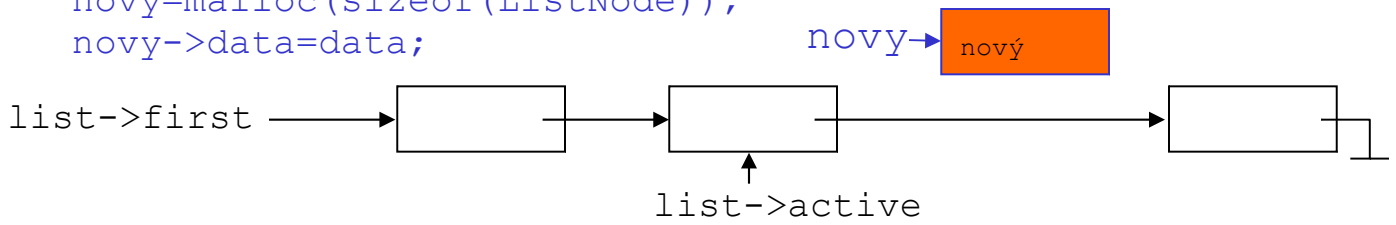
Metoda **`list_insert_next(List * list, tData data);`**

Příklad stavu seznamu před zavoláním:

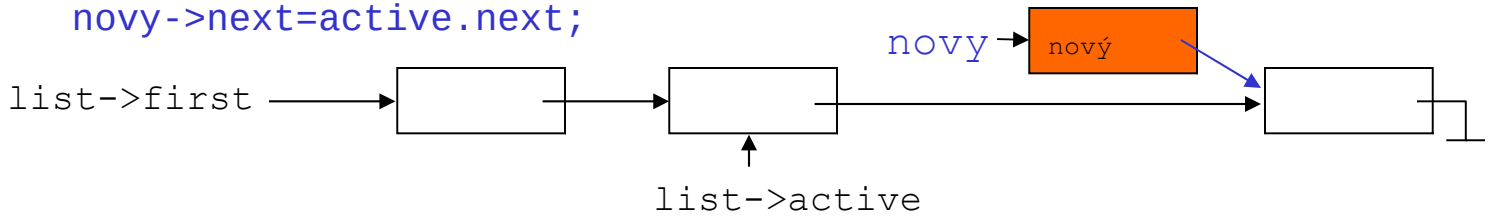


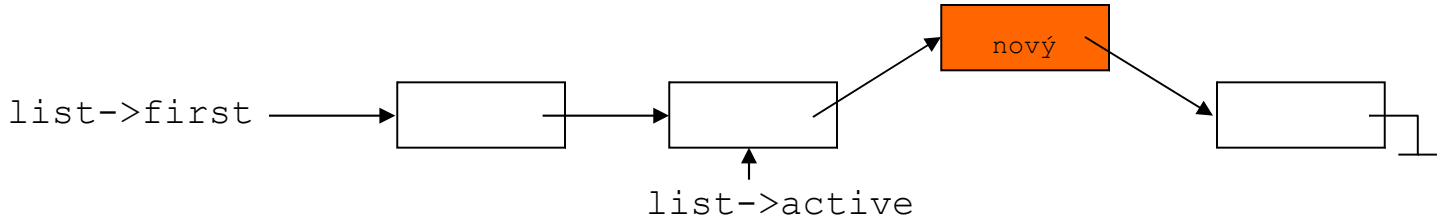
- Abychom vložili prvek za aktivní prvek, je potřeba udělat tyto kroky:

1. Na začátku procedury je potřeba ošetřit možnost, že aktivní prvek není nebo za ním již žádný není.
   `if (list->active!=NULL and list->active->next!=NULL) {`
2. Pak si s použitím pomocného ukazatele alokujeme místo pro nový prvek a naplníme jeho datovou část:
   `novy=malloc(sizeof(ListNode));`
   `novy->data=data;`



3. nasměrovat ukazatele "next" v novém prvku na prvek za aktivním prvkem
   `novy->next=active.next;`

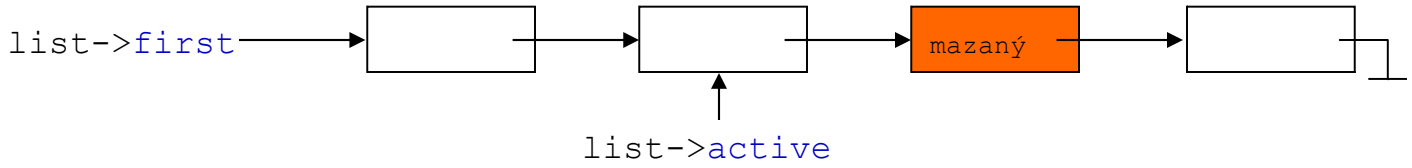




Příklad implementace jedné metody

Metoda **`void list_delete_next`**`(List* list);`

Příklad stavu seznamu před zavoláním metody:



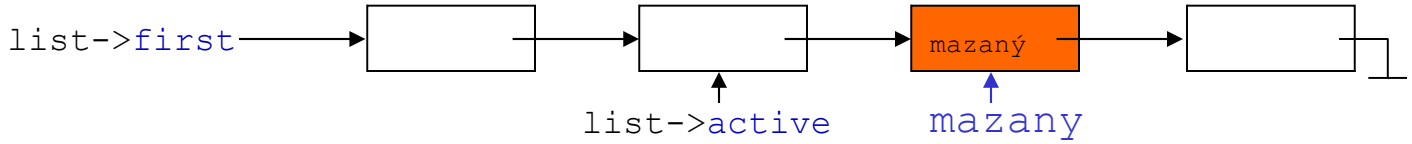
Abychom smazali prvek za aktivním prvkem, je potřeba udělat tyto kroky:

1. Na začátku procedury je potřeba ošetřit možnost, že ukazatele mohou být NULL:

```
if( list->active != NULL && list->active->next != NULL ) {
```

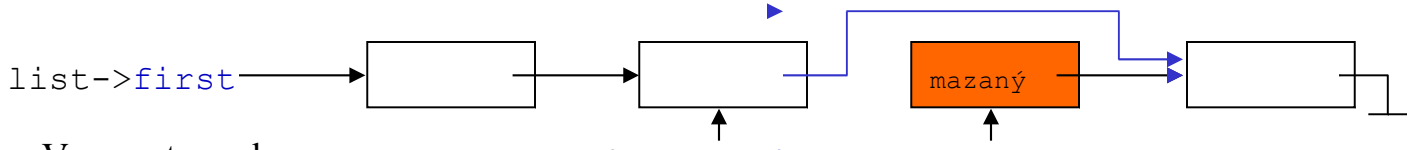
2. Pak si uložit ukazatel na mazaný prvek do pomocného ukazatele

```
mazany=list->active->next;
```



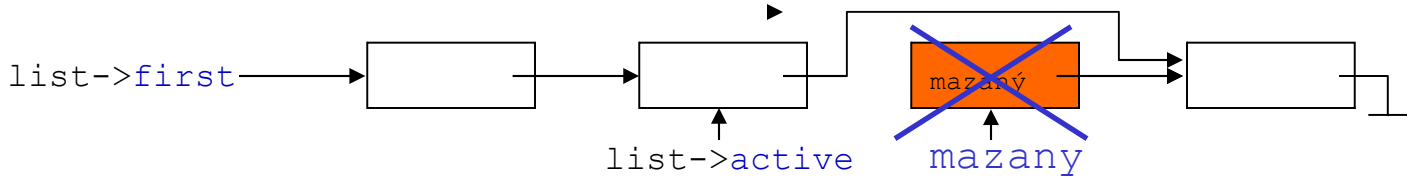
3. přesměrovat ukDalsi v aktivním prvku na prvek za mazaným prvkem

```
list->active->next = mazany->next;
```



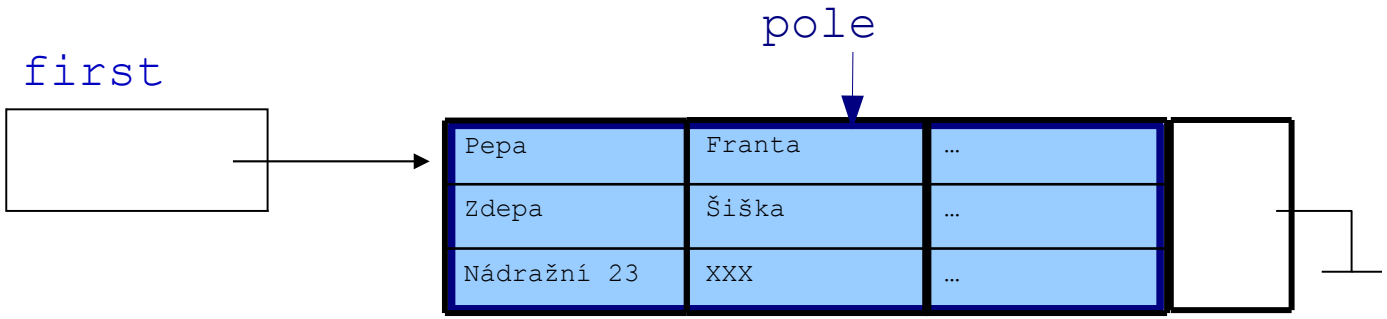
4. Vymazat prvek:

```
if (mazany->data!=NULL) myFree(mazany->data);
myFree(mazany);
```



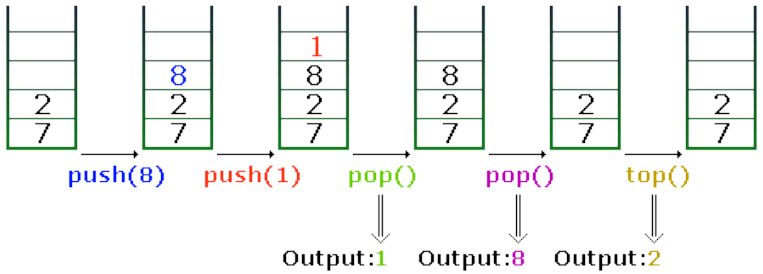
Další možnosti implementace

- V závislosti na velikosti datových prvků je dobré zamyslet se nad paměťovou režií zvolené implementace – např. paměťová režie pro lineární seznam implementovaný jako seznam jednotlivých prvků provázaných ukazateli je = sizeof(tukazatel)/sizeof(tPrvek). Pokud bude velikost ukazatele=4 bajty a velikost prvku 12 bajtů (včetně ukazatele), je výsledná režie=30%!!!
- V takovém případě je lepší seznam implementovat jako seznam polí:



ADT zásobník

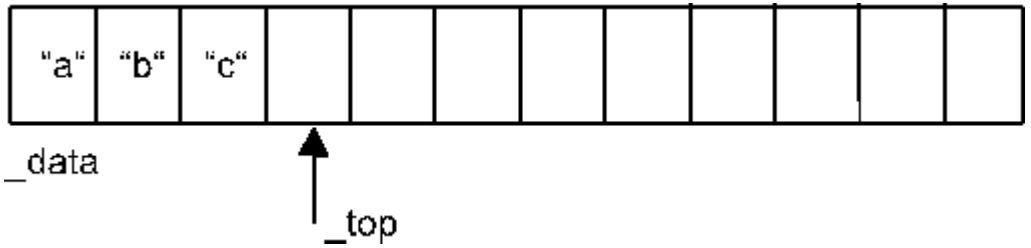
- Zásobník (stack) je dynamická datová struktura, která umožňuje postupné vkládání prvků a jejich odstraňování, ovšem v takovém poradí, že naposledy vložený prvek můžeme odstranit jako první (last in - first out, LIFO). Nad zásobníkem můžeme provádět následující operace:
  - push: vložení hodnoty na vrchol zásobníku,
  - pop: odstranění hodnoty z vrcholu zásobníku,
  - top: čtení hodnoty z vrcholu zásobníku a
  - empty: testování, zda je zásobník prázdný.



- Příklady použití zásobníku:
  - Tlačítko Undo a Redo v textových editorech
  - Tlačítka Zpět a Vpřed v HTML prohlížeči
  - Vyhodnocování aritmetických výrazů pomocí převodu jejich zápisu z infixového do postfixového tvaru
- Zásobník lze implementovat jako:
  - Statické pole
  - Dynamické pole
  - Lineární seznam

Implementace zásobníku polem

- Pokud víme, že pro svůj zásobník potřebujeme pouze předem známý maximální počet položek, můžeme jej implementovat pomocí statického pole:



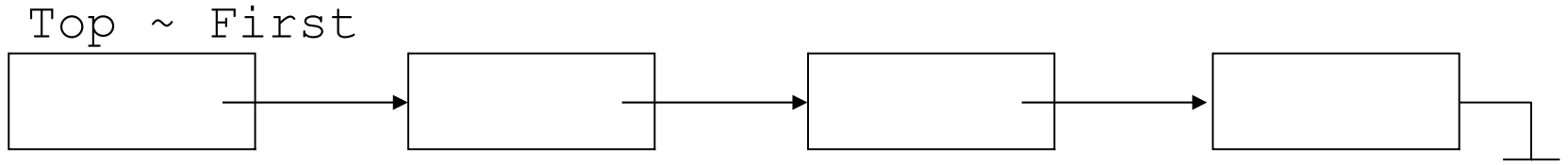
- V takovém případě bude ADT zásobník v Pascalu deklarován jako:

```
Type tStack=record
   pole: array[1..100] of tPrvek;
   top: integer;
End;
```

Implementace jednotlivých metod je triviální...

Implementace zásobníku seznamem

- Pokud neznáme maximální počet položek, můžeme zásobník implementovat pomocí lineárního seznamu:



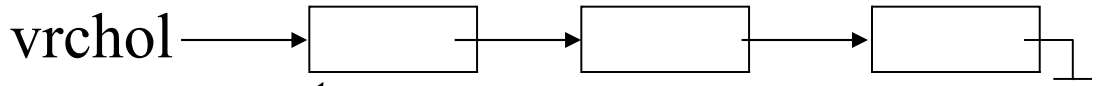
- V takovém případě bude ADT zásobník v Pascalu deklarován jako:

```
type tZasobnik =  tUkPrvek;
var zasobnik1: tStack;
```

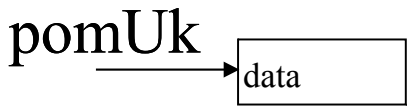
- metoda Push bude implementována zavoláním metody `InsertFirst(var L : tList; d: tData)` lineárního seznamu
- metoda Top bude implementována zavoláním metody `CopyFirst` lineárního seznamu
- metoda Pop bude implementována zavoláním metod CopyFirst a DeleteFirst
- Metoda Empty bude testem zda First=nil

Příklad implementace jedné metody

- Metoda **push** `(var vrchol : tZasobnik, d:tData);`
- příklad stavu seznamu před zavoláním push:

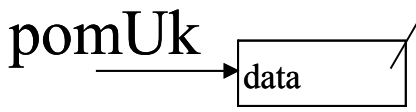


1. a 2. krok – vytvoření uzlu



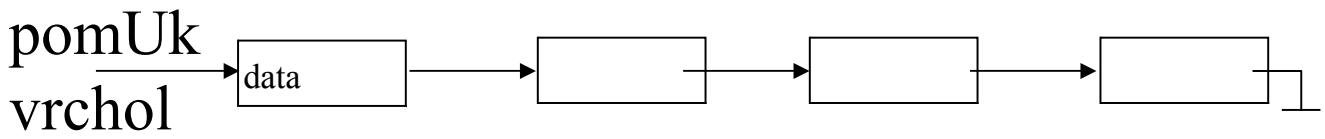

```
new(pomUk);
pomUk^.data:=d;
```

3. krok – napojení nového uzlu na stávající seznam




```
pomUk^.ukDalsi:=vrchol;
```

4. krok – nastavení vrcholu na nový 1. uzel

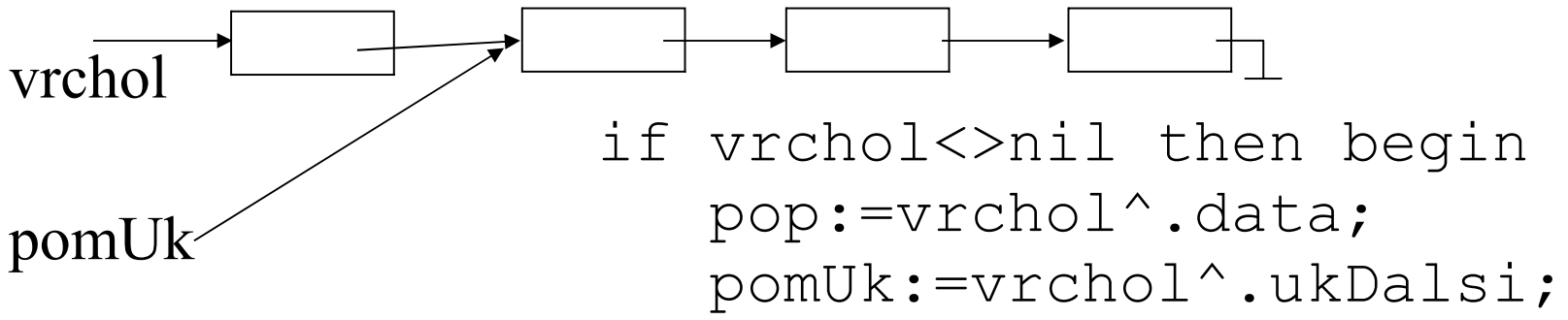



```
vrchol:=pomUk;
```

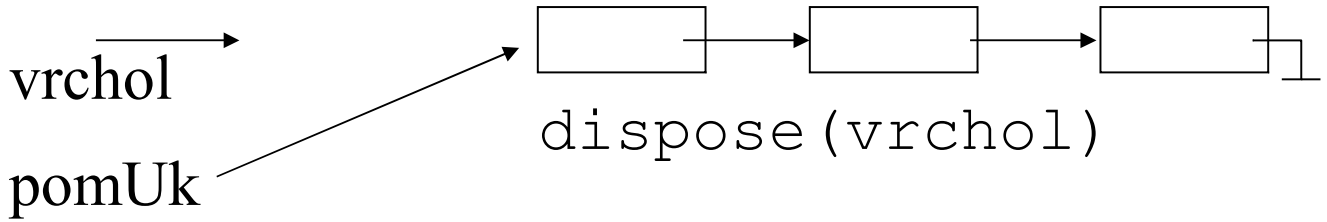
Příklad implementace jedné metody

- Metoda **pop** `(var vrchol : tZasobnik);`
- příklad stavu seznamu před zavoláním pop:

1. krok – před zrušením 1. prvku si zapamatujeme jeho data a ukazatel na 2. prvek



2. krok



3. krok