IAL – 2. přednáška

Lineární abstraktní datové typy

24. a 25. září 2024

Obsah přednášky

- Úvod – lineární datové struktury
  - Pole
  - Spojový seznam (angl. *linked list*)
- Abstraktní datový typ
  - Syntaktická a sémantická specifikace
  - Diagramy signatury
- ADT seznam
  - Jednosměrný, s hlavičkou
  - Dvousměrný
  - Kruhový – základní operace
  - Náměty k procvičení

Abstraktní datový typ (ADT)

- Matematický model pro datové typy, které jsou definované svým chováním (množinou hodnot a operací).
- Implementačně nezávislá specifikace datového typu s operacemi povolenými nad tímto typem.
- Cíl:
  - Zjednodušit a zpřehlednit program.
  - Odstínit uživatele od implementačních detailů.
- ADT x datová struktura (konkrétní způsob uložení/reprezentace dat).
- *Příklad:* zásobník – úložiště pro elementy pracující v režimu LIFO
- Implementace – pomocí modulů přístupných přes rozhraní nebo pomocí tříd.
- Data ale musí být nějak uložena – jaké máme možnosti?

Lineární datové struktury

- **Pole**
  - Statická struktura
  - Přímý přístup
  - Některé aplikace vyžadují přesuny segmentů pole – pomalé!
- **Spojový seznam**
  - Dynamická struktura
  - Sekvenční přístup
  - Snadné vkládání prvků na konkrétní pozici

- Lineární abstraktní datové typy (specifické):
  - Zásobník, fronta
  - Mohou být implementovány polem nebo lineárním seznamem
- *Pozn.:* množina – záleží na konkrétní implementaci, často implementována jako vyhledávací tabulka

Pole

- Z principu se jedná o **statickou strukturu**.
- Výhoda:
  - přímý přístup k jednotlivým prvkům
- Nevýhody:
  - nezbytné přesuny segmentů pole, pokud potřebujeme vložit/odstranit prvek na/z konkrétní pozice
  - předem určený počet prvků
- Časová složitost operací:
  - Přístup k prvku: ***O(1)***
  - Vyhledání prvku: ***O(n)***, v seřazeném poli ***O(log n)***
  - Vkládání prvku: na konec pole ***O(1)***, na konkrétní pozici ***O(n)***
  - Mazání prvku: při prohození s posledním prvkem ***O(1)***, bez prohození ***O(n)***

Nafukovací pole

- Chová se jako **dynamické** pole:
  - Začneme s polem konstantní velikosti (často jednoprvkovým).
  - Kdykoliv dojde místo, velikost pole zdvojnásobíme.
- **Pozor:** při každém zvětšování je třeba alokovat nový paměťový prostor a staré pole přesunout do tohoto prostoru!
- Časová složitost operací:
  - Přístup k prvku: ***O(1).***
  - Vyhledání prvku: ***O(n)***, v seřazeném poli ***O(log n).***
  - Vkládání prvku: **?**
  - Mazání prvku: **?**
- ***Pozn.:*** Pole lze také dynamicky zmenšovat (typicky při poklesu zaplnění pod čtvrtinu).

Nafukovací pole

- Přesun prvků pole do nového prostoru je časově drahá operace, ale provede se jen *jednou za čas.*
- Vkládání/mazání prvku může mít konstantní časovou složitost, ale někdy má lineární časovou složitost, protože se mění velikost pole – jak to vyjádřit? → amortizovaná časová složitost.
- *Amortizovaná časová složitost:*
  - Rozloží čas potřebný pro jednu drahou operaci do času potřebného pro všechny operace.
  - Udává odhad času potřebného pro provedení nějaké posloupnosti operací.
  - Můžeme např. vyjádřit, jak dlouho trvá vložení *n* prvků?

Nafukovací pole

- Amortizovaná časová složitost pro vložení n prvků:
  - Čas potřebný pro vložení *n* prvků bez zvětšování: $\Theta(n)$.
  - Čas potřebný pro zvětšení pole odpovídá aktuálnímu počtu prvků: $\Theta(i)$.
  - Zvětšujeme právě tehdy, když je aktuální počet prvků roven mocnině 2.
  - Čas potřebný pro všechna zvětšování: $\Theta(2^0+2^1+\ldots+2^k)$, kde 2^k je nejvyšší mocnina dvojky menší než *n* – odpovídá geometrické řadě se součtem $2^{k+1} - 1 < 2n$.
  - Celkově časová složitost pro vložení *n* prvků je $\Theta(n)$.

- Každý prvek má 2k+1 mincí, jedna se použije na vložení, 2k přijde do banku.
- Vždy, když se pole velikosti n zaplní, bude v něm k*n mincí. Pro přesun prvku se použije k mincí, po přesunutí všech mincí je bank prázdný a nové pole z poloviny plné. Nově vkládané prvky zase přinesou n/2*2k mincí.
- Protože jeden prvek přinese 2k mincí, k je konstanta, má vložení jednoho prvku konstantní amortizovanou časovou složitost.

Nafukovací pole

- Časová složitost operací:
  - Přístup k prvku: ***O(1)***
  - Vyhledání prvku: ***O(n)***, v seřazeném poli ***O(log n)***
  - Vkládání prvku: **amortizovaně konstantní *O(1)*** (na konec pole)
  - Mazání prvku: **amortizovaně konstantní *O(1)*** (s prohozením prvků)
- **Pozor:** Reálně zvětšení pole probíhá v jednom okamžiku, a proto vložení prvku ***občas*** **trvá dlouho** – nelze použít v real-time systémech, kde je potřeba garantovat určitou odezvu.
- Vkládání prvku **na určitou pozici** a mazání bez prohození prvků má stále složitost: ***O(n)***

Spojový seznam

- **Dynamická datová struktura**, prvky provázány pomocí ukazatelů
- Výhody:
  - Není třeba definovat počet prvků.
  - Prvky lze snadno a libovolně vkládat/mazat.
- Nevýhody:
  - Sekvenční přístup k jednotlivým prvkům
  - Nepohodlná práce s ukazateli – využít vhodný ADT
- Časová složitost operací:
  - Přístup k prvku: ***O(n)***
  - Vyhledání prvku: ***O(n)***, v seřazeném seznamu ***O(n)***
  - Vkládání prvku: ***O(1)*** (na aktuální/první pozici)
  - Mazání prvku: ***O(1)*** (na aktuální/první pozici)

Obsah přednášky

- Úvod – lineární datové struktury
  - Pole
  - Spojový seznam (angl. *linked list*)
- Abstraktní datový typ
  - Syntaktická a sémantická specifikace
  - Diagramy signatury
- ADT seznam
  - Jednosměrný, s hlavičkou
  - Dvousměrný
  - Kruhový – základní operace
  - Náměty k procvičení

Abstraktní datový typ

- **Abstraktní datový typ (ADT)** je definován množinou hodnot, kterých smí nabýt každý prvek tohoto typu, a množinou operací nad tímto typem.

- Smysl ADT:
  - Zvýšit datovou abstrakci
  - Snížit algoritmickou složitost programu (algoritmu)
- ADT:
  - zdůrazňuje *„co dělá“*
  - potlačuje *„jak to dělá“*
  - připomíná *„černou skříňku“*
- Příklad jednoduchého ADT: textový řetězec (string)

Abstraktní datový typ

- Lze definovat vlastní ADT – na míru pro konkrétní aplikaci
- Jak vytvořit vlastní ADT:
  - Řekneme, jak budou vypadat prvky tohoto ADT (typicky budou nějakého dříve definovaného typu).
  - Stanovíme množinu operací, pro každou operaci definujeme syntaxi a sémantiku.
  - Naimplementujeme.
- *Pozn.:* Nejčastěji ale využijeme nějaký ADT dostupný v knihovnách našeho oblíbeném programovacího jazyka. (Pozor na to, jak je vnitřně implementován!)

Abstraktní datový typ

- Návrh operací pro ADT obvykle zohledňuje:
  - Potřebu s daty (prvky) manipulovat
  - Vnitřní strukturu datového typu
  - Případně potřeby cílové aplikace
- Typické operace pro ADT:
  - Inicializace – připraví datovou strukturu
  - Operace pro:
    - vkládání prvků
    - zpřístupnění (získání) hodnoty prvku
    - nastavení (změnu) hodnoty prvku
    - pro odstranění prvku
    - vyhledání prvku nebo pro „pohyb“ po datové struktuře
  - Predikáty pro zjištění stavu struktury

Syntaxe a sémantika

- **Syntaxe** vyjadřuje pravidla korektního zápisu jazykové konstrukce.
- **Sémantika** vyjadřuje (popisuje) účinek (význam) zápisu jazykové konstrukce.

Syntaxe a sémantika u ADT

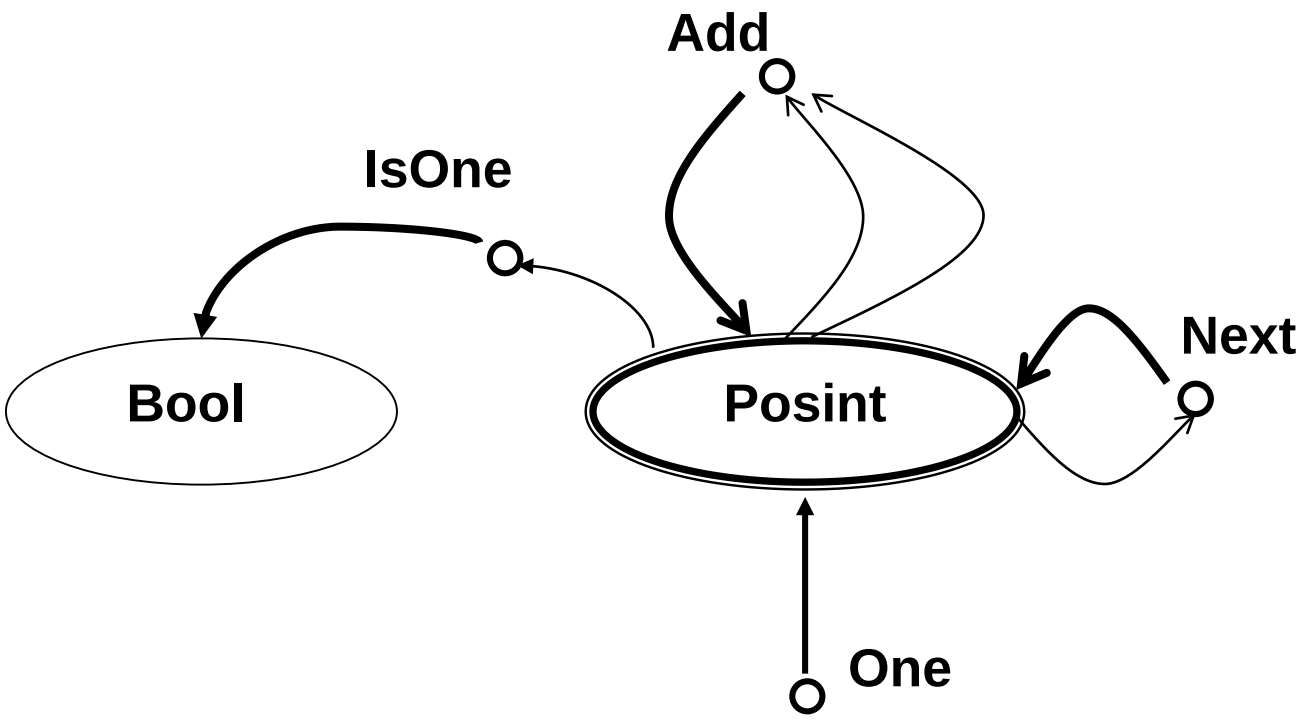
- Specifikace syntaxe pro ADT:
  - algebraická signatura
  - diagram signatury
- Specifikace sémantiky pro ADT:
  - slovní popis
  - axiomatická specifikace
  - operační (funkční popis)
    - popis prostřednictvím funkcí/procedur
    - nevýhoda – podkládá způsob implementace
  - další možnosti (specifikace úplným výčtem účinků)

Příklad: Algebraická signatura

- Příklad algebraické signatury pro ADT kladný integer (Posint)

  One: → *Posint* (generátor, inicializace)
  Next: Posint → *Posint*
  Add: Posint x Posint → *Posint*
  IsOne: Posint → *Bool*

Příklad: Diagram signatury



Příklad: Slovní popis sémantiky

- Slovní popis sémantiky operací typu Posint:
  - Operace **One** ustaví hodnotu typu Posint rovnu jedné. Tato operace je inicializace typu (generátor).
  - Operace **Next** vrátí další (následující) hodnotu (plus jedna).
  - Operace **Add** vrátí aritmetický součet dvou prvků typu Posint.
  - Operace (predikát) **IsOne** nabude hodnoty *true*, pokud je argument hodnota rovna jedné. V jiných případech má operace hodnotu *false*.

Příklad: Axiomatická specifikace sémantiky

- Axiomatická specifikace sémantiky operací typu Posint:
  1. Add(X,Y) = Add(Y,X)
  2. Add(One,X) = Next(X)
  3. Add(Next(X),Y) = Next(Add(X,Y))
  4. IsOne(One) = true
  5. IsOne(Next(X)) = false.

Obsah přednášky

- Úvod – lineární datové struktury
  - Pole
  - Spojový seznam (angl. *linked list*)
- Abstraktní datový typ
  - Syntaktická a sémantická specifikace
  - Diagramy signatury
- ADT seznam
  - Jednosměrný, s hlavičkou
  - Dvousměrný
  - Kruhový – základní operace
  - Náměty k procvičení

ADT seznam (list)

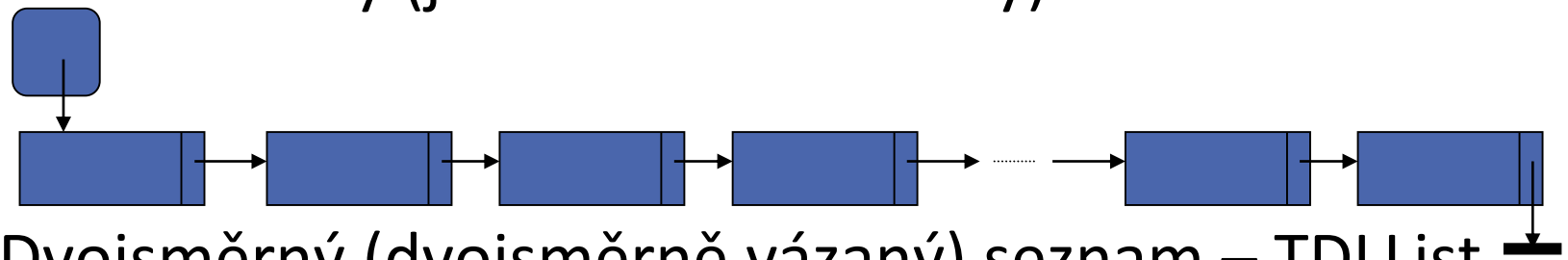
- **Seznam je lineární, homogenní, dynamická datová struktura.**

- **Prvkem** seznamu může být libovolný jiný dříve definovaný datový typ (i strukturovaný – např. seznam seznamů)

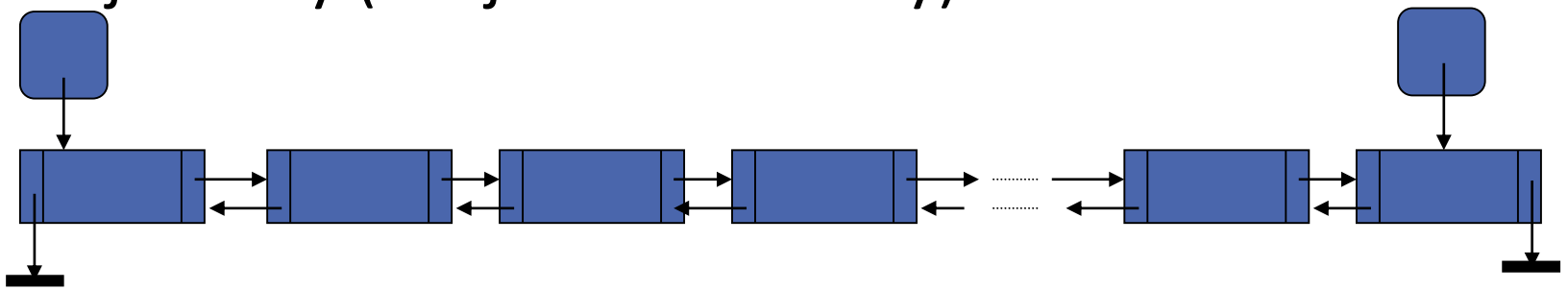
- *Pozn.:* Lineárnost znamená, že každý prvek struktury má právě jednoho předchůdce a jednoho následníka. Výjimku tvoří první a poslední prvek.
- *Pozn.:* Seznam může být prázdný.

Typy seznamů

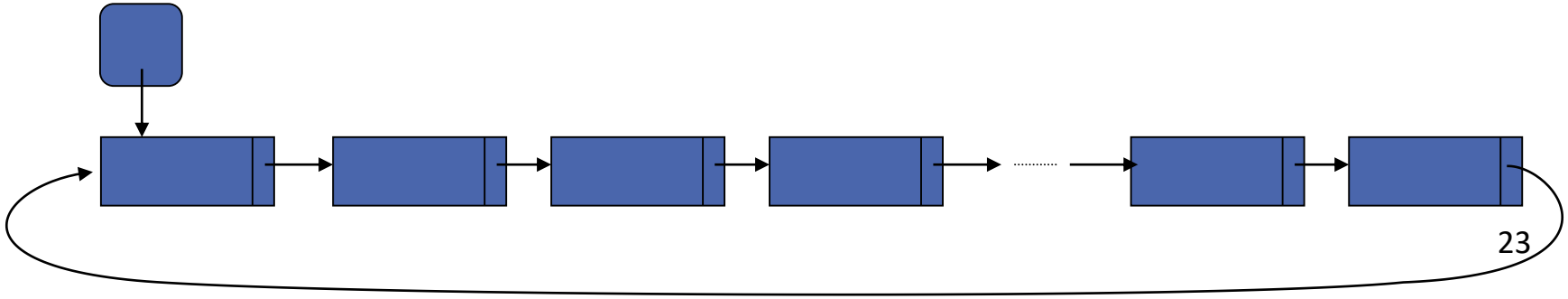
- Jednosměrný (jednosměrně vázaný) seznam – TList



- Dvojsměrný (dvojsměrně vázaný) seznam – TDLList



- Kruhový jednosměrný seznam

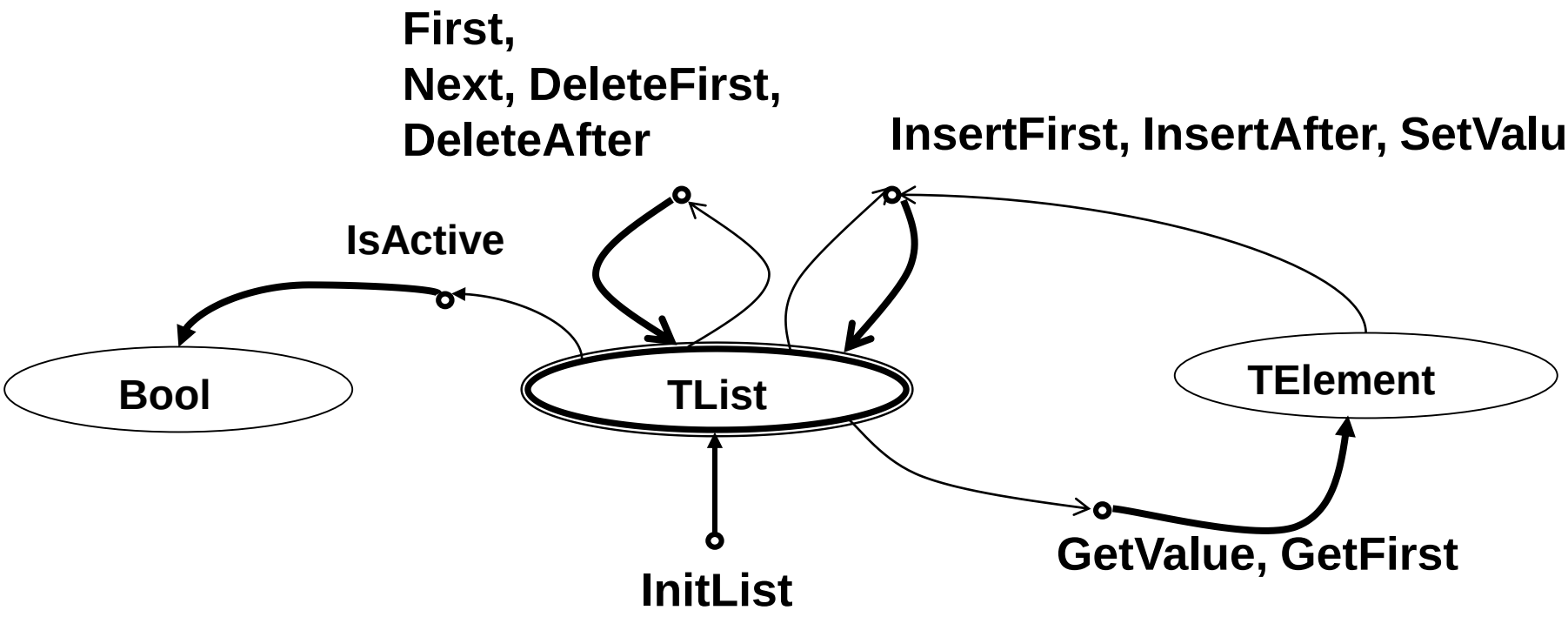


ADT TList – návrh operací

- **Inicializace**: InitList
- **Vložení** prvku: InsertFirst, InsertAfter
- **Zpřístupnění** hodnoty prvku: GetFirst, GetValue
- **Nastavení** hodnoty prvku: SetValue
- **Mazání** (rušení) prvku: DeleteFirst, DeleteAfter
- Operace pro **pohyb** po datové struktuře: First, Next
- **Predikát**: IsActive

- *Pozn.:* Pro ADT seznam je jistě možné definovat i jinou množinu operací, ale toto je definice, kterou budeme používat v rámci předmětu IAL a budeme ji také vyžadovat. Sada operací je navržena tak, aby umožňovala provedení nejčastějších aktivit nad lineárním seznamem.

ADT TList – diagram signatury



ADT TList – sémantika operací

- **InitList(L)** – vytvoří prázdný seznam prvků daného typu.
- **InsertFirst(L, El)** – vloží element **El** do seznamu **L** jako první prvek (jediná operace, která může vložit prvek do prázdného seznamu).
- **GetFirst(L)** – funkce, která vrátí hodnotu prvního prvku.
  V případě prázdného seznamu chyba!
- **GetValue(L)** – funkce, která vrátí hodnotu aktivního prvku.
  V případě neaktivního seznamu chyba!
- **DeleteFirst(L)** – zruší první prvek seznamu (pro prázdný seznam bez účinku).
- **First(L)** – první prvek se stane aktivním (pro prázdný seznam bez účinku).

ADT TList – sémantika operací

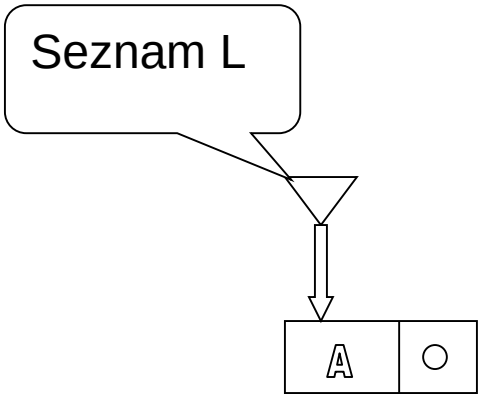
- **Next(L)** – aktivita se přenese na následující prvek (pro prázdný seznam bez účinku; byl-li aktivní prvek posledním, aktivita se ztratí – seznam přestane být aktivním).
- **SetValue(L, El)** – hodnota aktivního prvku je přepsána hodnotou **El** (v případě neaktivního seznamu bez účinku).
- **InsertAfter(L, El) –** vloží prvek **El** jako nový za aktivní prvek (v případě neaktivního seznamu bez účinku).
- **DeleteAfter(L) –** zruší prvek za aktivním prvkem (v případě neaktivního seznamu nebo neexistence prvku za aktivním – bez účinku).
- **IsActive(L) –** predikát – vrací hodnotu *true* v případě, že seznam je aktivní; v jiném případě vrací hodnotu *false.*

ADT TList – ukázka účinků operací

`InitList(L)`

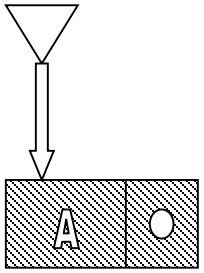
```
// Operace vytvoří prázdný seznam.
```

`InsertFirst(L,A)`

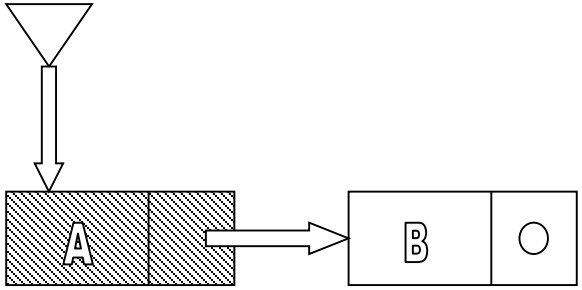


ADT TList – ukázka účinků operací

```
// Operace
First(L)
// První je aktivní
```

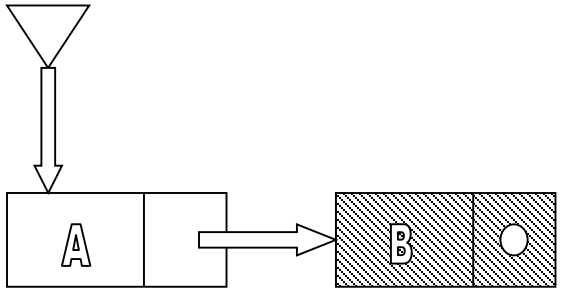



```
// Operace
InsertAfter(L,B)
// Prvek B se vloží za aktivní
```

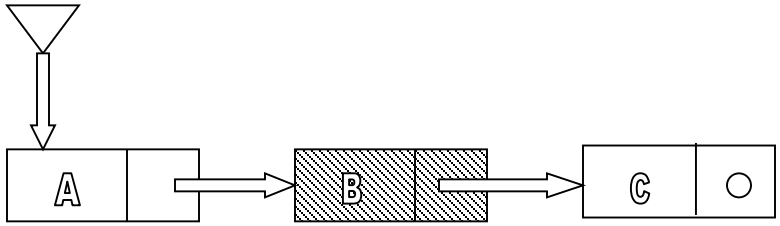


ADT TList – ukázka účinků operací

```
// Operace
Next(L)
// posune aktivitu na další prvek
```

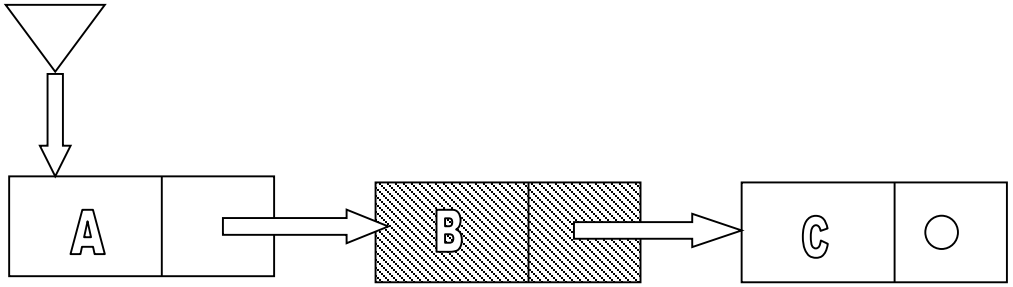



```
// Operace
InsertAfter(L,C)
// vloží nový prvek C za aktivní
```

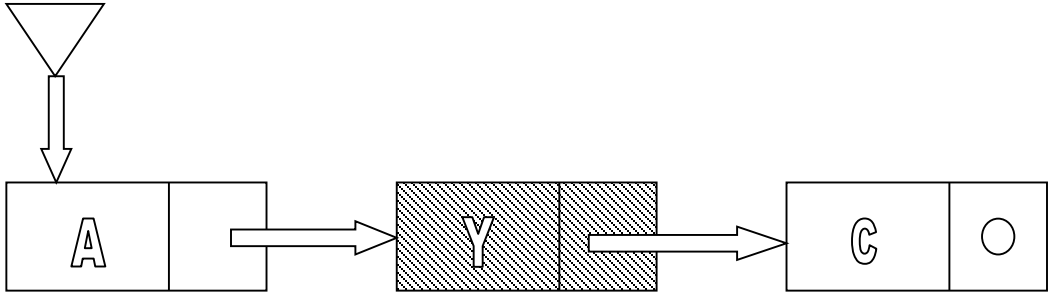


ADT TList – ukázka účinků operací

```
// Operace
Val ← GetValue(L)
// stejný efekt jako Val ← B
// V případě neaktivního seznamu by došlo k chybě!
```

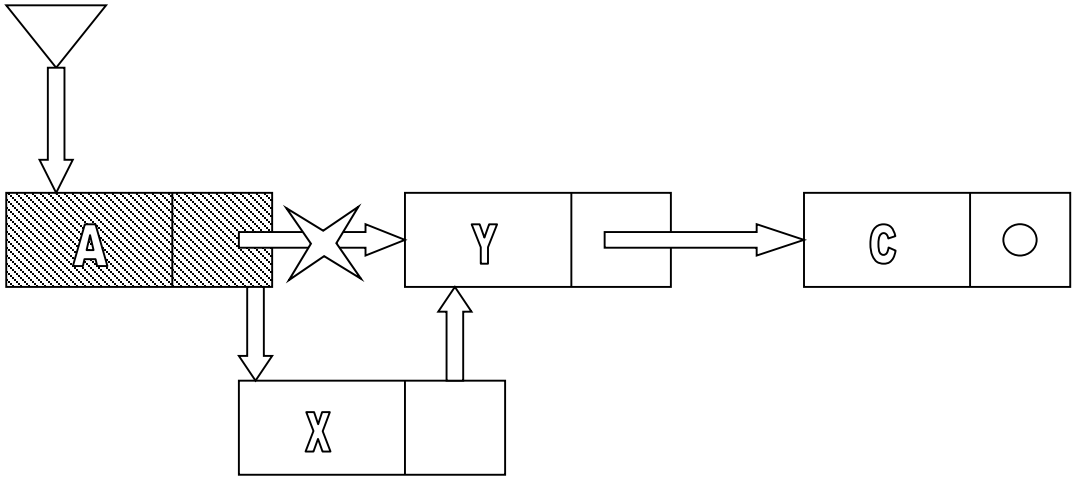



```
// Operace
SetValue(L,Y)
// přepíše hodnotu aktivního parametrem
```



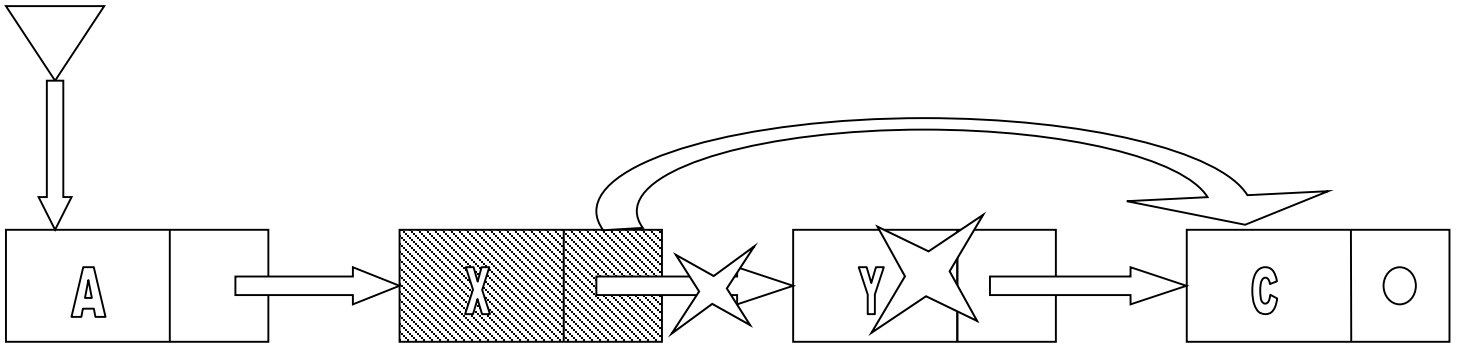
ADT TList – ukázka účinků operací

```
// Dvojice operací
First(L)              // Učiní první aktivním
InsertAfter(L,X)      // Vloží X za aktivního
```



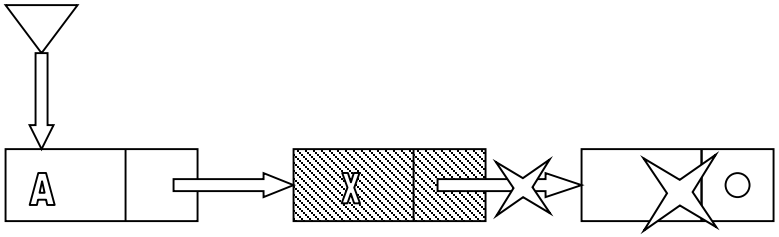
ADT TList – ukázka účinků operací

```
// Dvojice operací
Next(L)              // posune aktivitu na další prvek
DeleteAfter(L)       // zruší prvek za aktivním
```



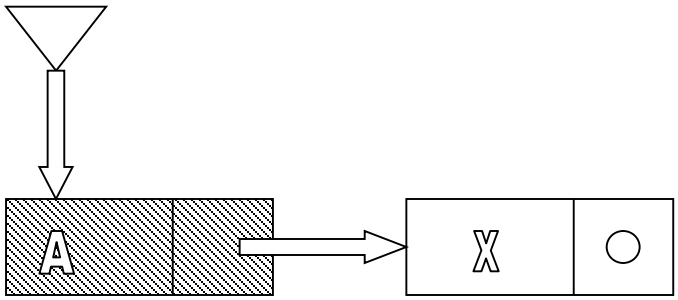
ADT TList – ukázka účinků operací

```
// Operace
DeleteAfter(L)
// zruší prvek za aktivním
```

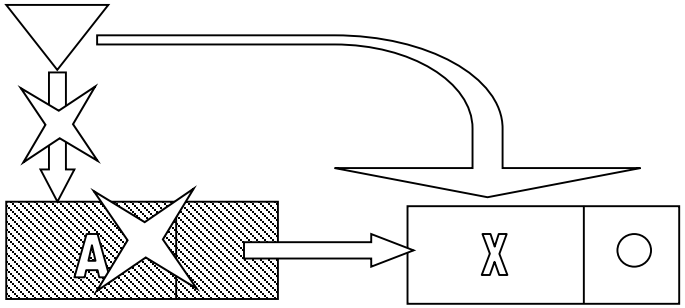


ADT TList – ukázka účinků operací

```
// Operace
First(L)     // první je aktivní
```

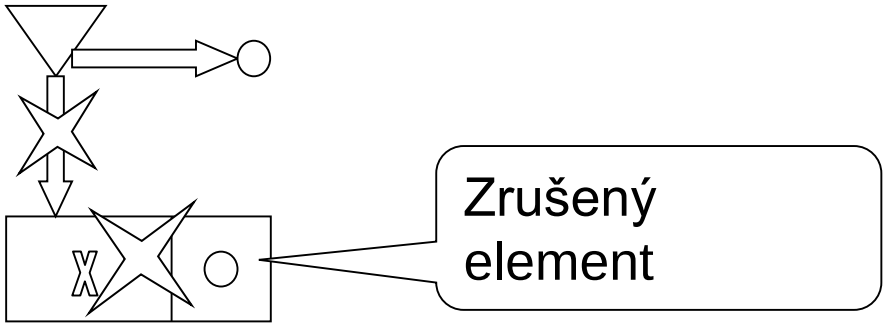


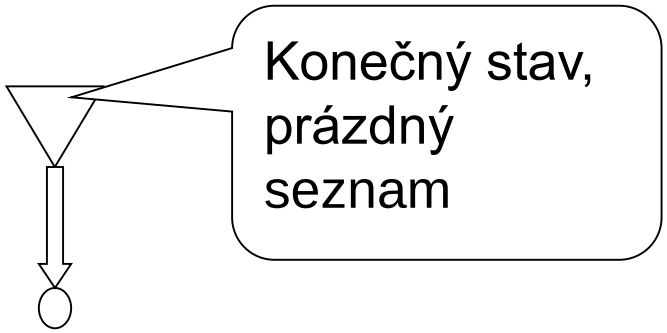

```
// Operace
DeleteFirst(L)     // zruší první prvek
```



ADT TList – ukázka účinků operací

```
// Operace
DeleteFirst(L)   // Zruší první, v našem případě jediný
                 // prvek. Vznikne prázdný seznam.
```





ADT TList – implementace operací

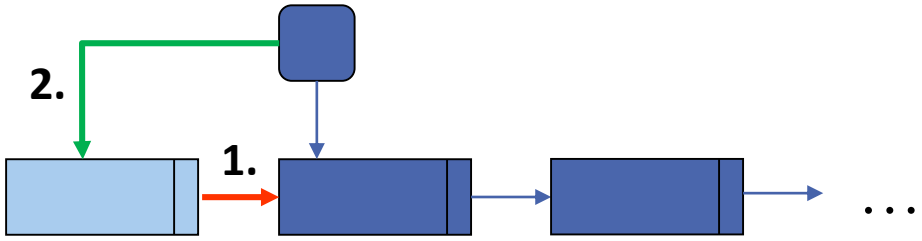
```
typedef struct telem                //typ prvku seznamu
{
      TData data;                   //datové složky elementu
      struct telem *nextPtr;        //ukazatel na následníka
}TElem;


typedef struct tlist  //ADT seznam
{
      TElem *first;   //ukazatel na první prvek seznamu
      TElem *act;     //ukazatel na aktivní prvek sez.
}TList;
```

ADT TList – implementace operací

```
void InitList (TList *l)
{
  l->act = NULL;
  l->first = NULL;
} /*InitList*/
```

ADT TList – implementace operací

```
void InsertFirst(TList *l,TData d)
{
  TElem *newElemPtr = (TElem *) malloc(sizeof(TElem));
                                        //vytvoření nového prvku
  if (newElemPtr == NULL) {
        printf("Chyba pri alokaci pameti \n");
        exit(1);
    }
  newElemPtr->data = d;           //nastavení datové složky
  newElemPtr->nextPtr = l->first;  //uk tam, kam začátek
  l->first = newElemPtr;     //zač. ukazuje na nový prvek
} /*InsertFirst*/
```



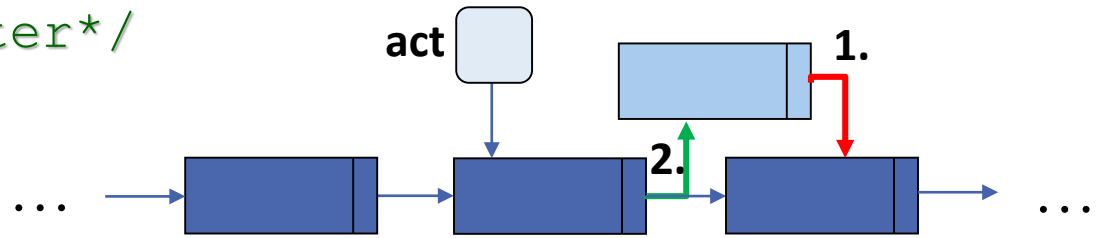
ADT TList – implementace operací

```
void First (TList *l)
{
  l->act = l->first       /* první se stane aktivním;
                          v prázdném seznamu beze změny*/
}  /*First*/
```

```
bool IsActive (TList *l)
{
  return l->act != NULL
}  /*IsActive*/
```

ADT TList – implementace operací

```
void InsertAfter (TList *l, TData d)
{
  if (l->act != NULL){
              //operace se provede jen pro aktivní seznam
    TElem *newElemPtr = (TElem *) malloc(sizeof(TElem));
    if (newElemPtr == NULL) {
        printf("Chyba pri alokaci pameti \n");
        exit(1);
    }
    newElemPtr->data = d;
    newElemPtr->nextPtr = l->act->nextPtr;
                          //nový ukazuje tam, kam aktivní
    l->act->nextPtr = newElemPtr;
                          //aktivní ukazuje na nového
  } //if aktivní
}  /*InsertAfter*/
```

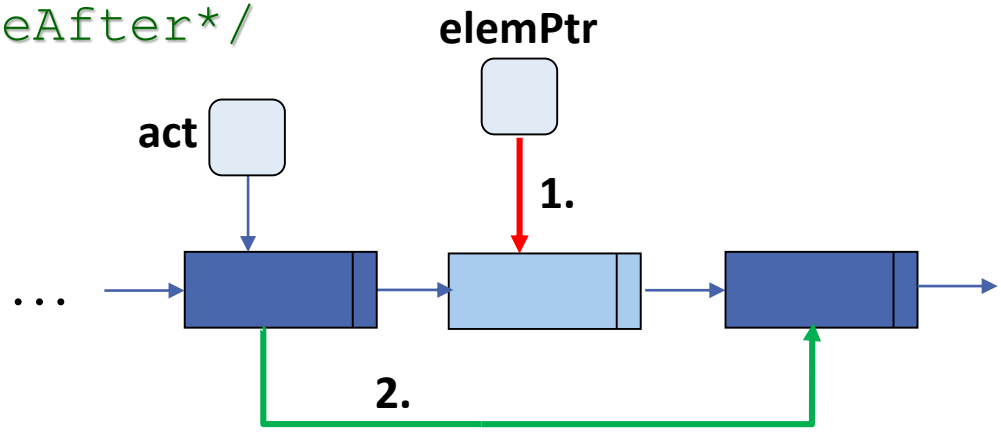


ADT TList – implementace operací

```
TData GetValue (TList *l)
/* Operace předpokládá ošetření aktivity seznamu
ve tvaru: if IsActive(L){...GetValue(&l)...}
GetValue v neaktivním seznamu způsobí chybu! */
{
  return (l->act->data);
}  /*GetValue*/


void SetValue (TList *l, TData d)
{
  if (l->act != NULL) {
    l->act->data = d;
  }
}   /*SetValue*/
```

ADT TList – implementace operací

```
void DeleteAfter (TList *l)
{
  TElem *elemPtr;
  if (l->act != NULL){
    if (l->act->nextPtr != NULL) {          //je co rušit
      elemPtr = l->act->nextPtr; //ukazatel na rušeného
      l->act->nextPtr = elemPtr->nextPtr; //překlenutí
      free(elemPtr);
    } //existuje rušený
  }  //aktivní seznam
}   /*DeleteAfter*/
```



Práce s ADT

- Po definici ADT už s danou strukturou pracujeme **pouze** s využitím definovaných operací!
- **Nezasahujeme do vnitřní struktury** jiným způsobem, nemanipulujeme s ukazateli atd.

Typické algoritmy nad ADT seznam

1. Délka seznamu
2. Kopie (duplikát) seznamu
3. Zrušení seznamu
4. Ekvivalence dvou seznamů
5. Relace (lexikografická) dvou seznamů
6. Vkládání nových prvků do seznamu (na začátek, na pozici danou ukazatelem, pořadím, aktivitou, na konec)
7. Vkládání a rušení podseznamu

Typické algoritmy nad ADT seznam

8. Vyhledávání prvku v seznamu
9. Rušení prvku seznamu (prvního, posledního, prvku na a za pozicí dané ukazatelem, pořadím, aktivitou)
10. Seřazení prvků seznamu podle velikosti (klíče)
11. Konkatenace (zřetězení) dvou a více seznamů (podseznamů) do jednoho seznamu (např. podseznamů obsahujících textová „*slova*“ do „*věty*“)
12. Dekatenace (rozčlenění) jednoho seznamu na podseznamy (např. rozčlenění textové „*věty*“ na „*slova*“)

Příklad: délka seznamu

```
int function Length(Tlist L)
    Count ← 0
    First(L)
    while IsActive(L):
      Count ← Count+1
      Next(L)
    return (Count)
```

Rekurzivní definice

- **Délka seznamu:**
  Je-li seznam prázdný, má délku nula. V jiném případě je jeho délka 1 plus délka zbytku seznamu.

- **Ekvivalence dvou seznamů:**
  Dva seznamy jsou ekvivalentní, když jsou oba prázdné nebo když se rovnají jejich první prvky a současně jejich zbytky.

- *Pozn.:* Rekurzivní implementaci operací, které zjistí délku resp. ekvivalenci seznamů, lze snadno provést v ukazatelové doméně. Uvedené definice nelze snadno přepsat na rekurzivní funkce čistě s využitím zavedených operací ADT. V případě potřeby lze zvážit rozšíření repertoáru operací.

Seznam s hlavičkou

- Abychom se vyhnuli „extra“ ošetřování prvního prvku, můžeme použít tzv. hlavičku.
- Hlavička je první prvek seznamu, který má pomocnou funkci a není skutečným prvkem seznamu.
- Prázdný seznam obsahuje pouze hlavičku.
- Seznam lze dočasně opatřit hlavičkou, která se v závěru odstraní.
- Hlavička odstraňuje zbytečný prolog pro první prvek před cyklem opakovaných operací nad ostatními prvky seznamu.

Příklad – kopie seznamu bez hlavičky

```
procedure CopyList (TList LOrig, TList LDupl)
// S využitím ADT TList nad prvky integer a jeho operací
   InitList(LDupl)
   First(LOrig)
   if IsActive(LOrig):
      // vytvoření prvního prvku se provede před cyklem
      El ← GetValue(LOrig)
      InsertFirst(LDupl,El)
      Next(LOrig)
      First(LDupl)
      // vytvoření zbytku seznamu se provede v cyklu
      while IsActive(LOrig):
         El ← GetValue(LOrig)
         InsertAfter(LDupl,El)
         Next(LOrig)
         Next(LDupl)
```

Příklad – kopie seznamu s hlavičkou

```
procedure CopyList (TList LOrig, TList LDupl)
// S využitím ADT TList s hlavičkou
// nad prvky integer a jeho operací

    InitList(LDupl)          //prázdná kopie
    InsertFirst(LDupl,0)  //vložení hlavičky s hodnotou 0
    First(LOrig)          //první originálu nastav na aktivní
    First(LDupl)          // nastav hlavičku na aktivní
           // vytvoření celého seznamu se provede v cyklu
    while IsActive(LOrig):
        El ← GetValue(LOrig)
        InsertAfter(LDupl,El)
        Next(LOrig)
        Next(LDupl)
    DeleteFirst(LDupl)              //odstranění hlavičky
```

Příklad – reverzní kopie seznamu

Pozn.: bez použití aktivity v duplikátním seznamu lze vytvořit kopii, v níž jsou prvky v obráceném pořadí:

```
while IsActive(LOrig):
    El ← GetValue(LOrig)
    InsertFirst(LDupl,El)
    Next(LOrig)
```

ADT Dvojsměrný seznam (TDLList)

- **Návrh operací:**
  - Využijeme operace jednosměrného seznamu (pro rozlišení operací doplníme předponu DLL – doubly linked list):
    - DLL_InitList, DLL_GetValue, DLL_SetValue, DLL_IsActive, DLL_First, DLL_InsertFirst, DLL_DeleteFirst, DLL_GetFirst, DLL_Next, DLL_InsertAfter, DLL_DeleteAfter
  - Doplníme symetrické operace:
    - DLL_Last, DLL_InsertLast, DLL_DeleteLast, DLL_GetLast, DLL_Previous, DLL_InsertBefore, DLL_DeleteBefore
- Musíme udržovat více ukazatelů, implementace operací je komplikovanější, ale struktura je uživatelsky přívětivější.

ADT TDLList – sémantika operací

- **DLL_InsertLast(L, El)** – vloží prvek **El** jako poslední (do prázdného seznamu současně i první; spolu s DLL_InsertFirst je to jediná operace, která může vložit prvek do prázdného seznamu).
- **DLL_GetLast(L)** – funkce, která vrátí hodnotu posledního prvku. V případě prázdného seznamu chyba!
- **DLL_Previous(L)** – posune aktivitu o prvek zpět.
- **DLL_Last(L)** – nastaví aktivitu na poslední prvek. V případě prázdného seznamu bez účinku.

ADT TDLList – sémantika operací

- **DLL_DeleteLast(L)** – zruší poslední prvek. (V případě, že poslední prvek je současně jediný a první, vznikne prázdný seznam. Je s operací DLL_DeleteFirst jediná operace, která může odstranit jediný – poslední prvek. Pro prázdný seznam bez účinku.)
- **DLL_InsertBefore(L, El)** – vloží hodnotu prvku El před aktivní prvek. Je-li seznam neaktivní, je operace bez účinku.
- **DLL_DeleteBefore(L)** – operace zruší prvek před aktivním prvkem. Je-li seznam neaktivní nebo před aktivním prvkem (nalevo od něj) není žádný prvek, je operace bez účinku.

ADT TDLList – implementace operací

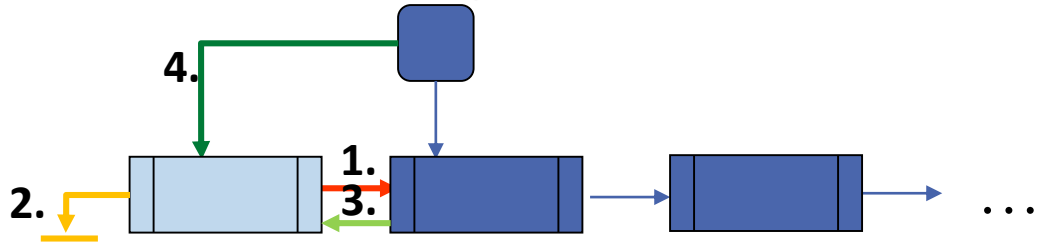
```
typedef struct tdllelem          //typ prvku seznamu
{
      TData data;                //datové složky elementu
      struct tdllelem *next;     //ukazatel na následníka
      struct tdllelem *prev;     //ukazatel na předchůdce
}TDLLElem;

typedef struct tdllist  //ADT seznam
{
      TDLLElem *first;  //ukazatel na první prvek seznamu
      TDLLElem *last    //ukazatel na poslední prvek
      TDLLElem *act;    //ukazatel na aktivní prvek sez.
  /*  int numberOfEl;   - počítadlo prvků seznamu */
}TDLList;
```

ADT TDLList – implementace operací

```
void DLL_InitList (TDLList *l)
{
   l->act = NULL;
   l->last = NULL;
   l->first = NULL;
   /*l->numberOfEl = 0;*/
} /*DLL_InitList*/
```

ADT TDLList – implementace operací

```
void DLL_InsertFirst (TDLList *l, TData d)
{
  TDLLElem *newElemPtr = (TDLLElem *)
     malloc(sizeof(TDLLElem));
                        //zkontrolovat úspěšnost operace malloc!
  newElemPtr->data = d;
  newElemPtr->next = l->first; //pravý nového na prvního
  newElemPtr->prev = NULL;  //levý nového ukazuje na NULL
  if (l->first != NULL) {   //seznam už měl prvního
    l->first->prev = newElemPtr;  //první bude doleva
                                  //ukazovat na nový prvek
  }
  else{                           //vložení do prázdného seznamu
    l->last = newElemPtr;
  }
  l->first = newElemPtr;    //korekce ukazatele začátku
}
```

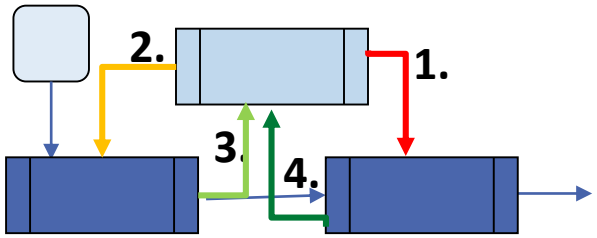


ADT TDLList – implementace operací

```
void DLL_DeleteFirst (TDLList *l)
{ //nutno sledovat, zda neruší aktivní prvek resp. jediný prvek
  TDLLElem *elemPtr;
  if (l->first != NULL){
    elemPtr = l->first;
    if (l->act == l->first){   //první byl aktivní
       l->act = NULL;                //ruší se aktivita
    }
    if (l->first == l->last){//seznam měl jediný prvek-zruší se
       l->first = NULL;
       l->last = NULL;
    }
    else {
      l->first = l->first->next; //aktualizace začátku seznamu
      l->first->prev = NULL;  //ukazatel prvního doleva na NULL
    }
    free(elemPtr);
  }
}
```

ADT TDLList – implementace operací

```
void DLL_InsertAfter (TDLList *l, TData d)
{  //nutno sledovat, zda nevkládá za poslední prvek
  if (l->act != NULL){          //je kam vkládat
    TDLLElem *newElemPtr =(TDLLElem *)
       malloc(sizeof(TDLLElem));
                                 //zkontrolovat úspěšnost malloc!
    newElemPtr->data = d;
    newElemPtr->next = l->act->next;
    newElemPtr->prev = l->act;
    l->act->next = newElemPtr;
                                 //navázání levého souseda na nový
    if (l->act == l->last){   //vkládá za posledního
      l->last = newElemPtr;   //korekce ukazatele na konec
    }
    else{              //navázání pravého souseda na vložený prvek
      newElemPtr->next->prev = newElemPtr;
    }
  } //aktivní
}
```



ADT TDLList – implementace operací

```
void DLL_DeleteAfter (TDLList *l)
{    //nutno sledovat, zda neruší poslední prvek
  if (l->act != NULL) {
    if (l->act->next != NULL) {    //Je co rušit?
      TDLLElem *elemPtr;
      elemPtr = l->act->next;      //ukazatel na rušený
      l->act->next = elemPtr->next;//překlenutí rušeného
      if (elemPtr == l->last){     //rušený poslední
        l->last = l->act;        //posledním bude aktivní
      }
      else{    //prvek za zrušeným ukazuje doleva na Act
        elemPtr->next->prev = l->act;
      }
      free(elemPtr);
    } //je co rušit
  } //aktivní
}
```

ADT TDLList – implementace operací

```
void DLL_DeleteBefore (TDLList *l)
{     //nutno sledovat, zda neruší první prvek
  if (l->act != NULL) {
    if (l->act->prev != NULL) {      //Je co rušit?
      TDLLElem *elemPtr;
      elemPtr = l->act->prev;        //ukazatel na rušený
      l->act->prev = elemPtr->prev; //překlenutí rušeného
      if (elemPtr == l->first){      //rušený první
        l->first = l->act;           //prvním bude aktivní
      }
      else{  //prvek před zrušeným ukazuje doprava na Act
        elemPtr->prev->next = l->act;
      }
      free(elemPtr);
    } //je co rušit
  } //aktivní
}
```

Příklad ADT TDLList: kopie

```
procedure CopyDoublyLinkedList(TDLList LOrig,
                               TDLList LDupl)
    DLL_InitList(DLLDupl)
    DLL_First(DLLOrig)
    while DLL_IsActive(DLLOrig):
        El ← DLL_GetValue(DLLOrig)
        DLL_Next(DLLOrig)
        DLL_InsertLast(DLLDupl,El)
        // díky vkládání na konec je kopírování jednoduché
```

- Podobně snadné je vkládání nebo rušení nalezeného prvku nebo před a za nalezeným prvkem.

Kruhový seznam

- **Kruhový (cyklický) seznam** lze vytvořit z lineárního seznamu tak, že se poslední prvek připojí na první prvek.
- Může být jednosměrně i dvojsměrně vázaný.
- Sémantický pohled: kruhový seznam nemá začátek ani konec.
- Praktický pohled: potřebujeme přístup alespoň k jednomu prvku, který bude mít pozici pracovního začátku.

ADT Kruhový seznam

- **Návrh operací:**
  - Lze odvodit ze souboru operací nad ADT seznam.
  - Operace odvozená od operace *First* ustaví aktivitu na „přístupový“ prvek seznamu.
  - Je vhodné doplnit možnost testu na průchod celým seznamem:
    - Zavedení počitadla prvků – umožní pohyb seznamem s využitím počítaného cyklu.
    - Zavedení predikátu, který vrátí hodnotu *true*, když se dostaneme na poslední (znovu první) prvek.

K procvičení

- Do seznamu celých čísel seřazených podle velikosti vložte nový prvek tak, aby seřazení seznamu zůstalo zachováno. V případě, že seznam obsahuje prvek rovný vkládanému, vložte vkládaný za shodný (poslední ze shodných).
- Ze seznamu celých čísel vyřaďte prvek zadaný parametrem. Příklad: Vyřazení hodnoty 5 ze seznamu <1,3,6,8> nemá žádný účinek. Zrušení hodnoty 5 ze seznamu <1,3,5,7> má za výsledek seznam <1,3,7>.
  V případě, že seznam obsahuje více shodných prvků rovných zadanému, zrušte poslední z nich.

K procvičení

- V seznamu celých čísel nalezněte nejdelší neklesající posloupnost. Výsledkem bude počáteční index a délka nalezené posloupnosti.
  Příklad: Výsledky pro seznam: <4,3,2,1> je začátek = 1, délka = 1.
  Výsledek pro seznam: <4,1,3,5,2,4,1,8,9> je začátek = 2, délka = 3;
  Nápověda: Algoritmus při hledání maximální délky uchovává index a délku každé kandidátské posloupnosti.
- V jednom průchodu seznamem najděte průměrnou hodnotu a rozptyl (dispersi) délek všech neklesajících posloupností daného seznamu celých čísel.
  Nápověda: Rozptyl D daného souboru hodnot je průměrná hodnota kvadratických odchylek všech hodnot souboru od průměrné hodnoty.

$$D = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2$$

K procvičení

- Jsou dány dva seřazené seznamy celých čísel. Sloučením z nich vytvořte jeden nový seznam seřazených celých čísel.
  Příklad: Je dán seznam L1=<1,3',5,7,9> a L2=<2,3'',4,6,8>.
  Výsledkem bude L3=<1,2,3',3'',4,5,6,7,8,9>.
  Poznámka: *Při práci s ukazateli k vytvoření nového seznamu použijte prvky zdrojových seznamů, které tím zaniknou. V případě práce s abstraktními operacemi nad ADT List zachovejte seznamy L1 a L2 a vytvořte nový seznam L3.*

K procvičení

- Navrhněte takový soubor operací nad ADT kruhový seznam, kterým lze provádět všechny nejznámější operace nad kruhovým seznamem jako:
  - vytvoření kruhového seznamu z lineárního seznamu
  - průchod kruhovým seznamem
  - vytvoření kopie kruhového seznamu
  - zrušení kruhového seznamu
  - ekvivalence dvou kruhových seznamů (Pozor! Poloha pracovního „prvního“ prvku není pro ekvivalenci významná!)
- Vyřešte uvedené úkoly s využitím abstraktních operací nad ADT kruhový jedno a/nebo dvojsměrně vázaný seznam i s využitím ukazatelů.