Abstraktní datové typy

Ing. Josef Vogel, CSc

Katedra softwarového inženýrství

Katedra teoretické informatiky,
Fakulta informačních technologii, ČVUT v Praze

Programování a algoritmizace 2, BI-PA2, 2011, Přednáška 10

Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Abstraktní datové typy

- Připomeňme:
  - Abstraktní datový typ specifikuje množinu hodnot a množinu operací nezávisle na implementaci
  - ADT lze specifikovat formálně pomocí signatury operací (syntaxe) a axiomů, které udávají sémantiku operací (viz 7. přednáška)
  - ADT lze méně formálně specifikovat pomocí signatury a slovního popisu sémantiky operací
  - V C++ ADT realizujeme pomocí třídy, větší použitelnost má realizace pomocí šablony třídy

Kontejnery

- Kontejner (kolekce) je ADT na organizované skladování prvků podle určitých pravidel

- Sekvenční kontejnery – různě organizované sekvence
  - Zásobník (Stack)
  - Fronta (Queue)
  - Seznam (List)
  - Pole (Array, Vector, Matrix, ...) umožňuje náhodný přístup k prvkům
  - Množina (Set)

- Asociativní kontejnery – pro vkládání, odebírání a vyhledávání je důležitý klíč prvku
  - Tabulka (Table, Map)

- V C++ realizujeme ADT kontejneru buď mocí třídy s daným typem prvků, nebo pomocí šablony třídy, kde typ prvků je parametrem šablony

ADT Zásobník

- Připomenutí: zásobník je kontejner, kde vkládání a odebírání prvků probíhá na stejném konci zvaném vrchol zásobníku (LIFO)
- Šablona:

```
template <class T>
class Stack {
public:
  Stack();
  bool empty() const;
  void push(const T& x);
  T top() const;
  void pop();
private:
  ...
};
```

- Poznámka:
  - operace *top* a *pop* bývají často spojeny do jediné:

```
T pop();
```

ADT Zásobník

- Implementace zásobníku:
  - pomocí pole, jehož počet prvků je dán konstantou (počet prvků zásobníku je staticky omezen)
  - pomocí dynamicky alokovaného pole, jehož počet prvků je dán parametrem konstruktoru (počet prvků zásobníku je dynamicky omezen)
  - pomocí dynamicky alokovaného a rozšiřitelného pole
  - pomocí jednosměrného spojového seznamu

- Složitost všech operací je konstantní, výjimkou je vkládání do zásobníku realizovaného pomocí rozšiřitelného pole; pokud se pole rozšiřuje vždy na dvojnásobnou délku, je složitost vkládání „téměř“ konstantní

- Podrobnosti viz příklady k přednášce 7

ADT Fronta

- Připomenutí: fronta je kontejner, kde se vkládá na konec a odebírá z čela (FIFO)
- Šablona:

```
template <class T>
class Queue {
public:
  Queue();
  bool empty() const;
  void add(const T& x);
  T front() const;
  void remove();
private:
  ...
};
```

- Poznámka:
  - operace *front* a *remove* bývají často spojeny do jediné:

```
T remove();
```

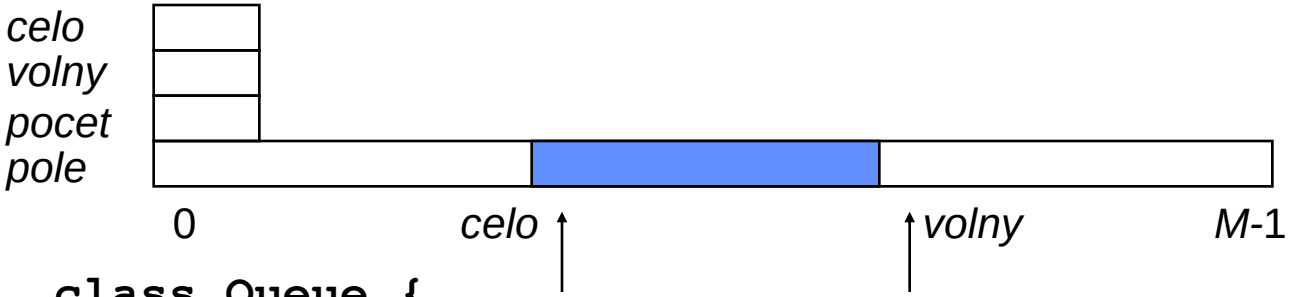
ADT Fronta

- Implementace fronty:
  - pomocí pole, jehož počet prvků je dán konstantou (počet prvků fronty je staticky omezen)
  - pomocí dynamicky alokovaného pole, jehož počet prvků je dán parametrem konstruktoru (počet prvků fronty je dynamicky omezen)
  - pomocí dynamicky alokovaného a rozšiřitelného pole
  - pomocí jednosměrného spojového seznamu

- Od implementace se očekává, že složitost všech operací bude konstantní (s výjimkou vkládání do rozšiřitelného pole, které se řeší podobně jako v případě zásobníku)

Implementace fronty

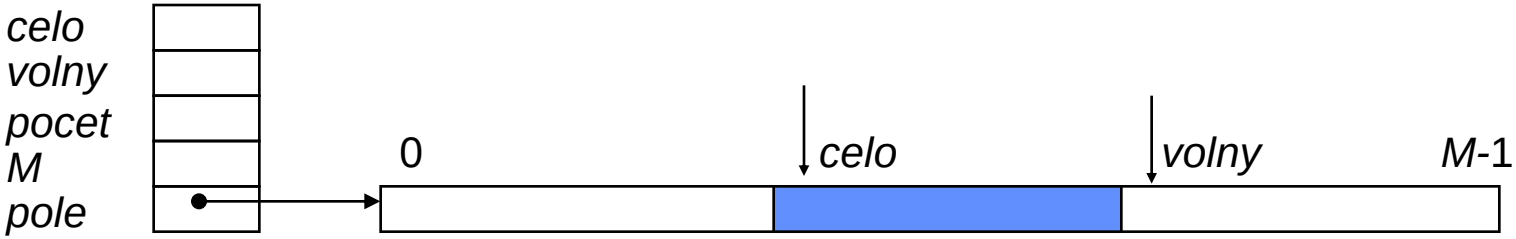
- Pomocí pole se staticky daným počtem prvků




```
class Queue {
public:
//enum {M = 40};
  static const int M=40; // tak to jde take
  Queue();
  void add(const T&);
  T front() const;
  void remove();
  bool empty() const;
private:
  int celo, volny, pocet;
  T pole[M];
};
```

Implementace fronty

- Pomocí pole s dynamicky daným počtem prvků




```
class Queue {
 public:
  Queue(int = 40);
  ~Queue();
  void add(const T&);
  T front() const;
  void remove();
  bool empty() const;
 private:
  int celo, volny, pocet;
  int M;
  T* pole;
  Queue(const Queue&);
  Queue& operator=(const Queue&);
};
```

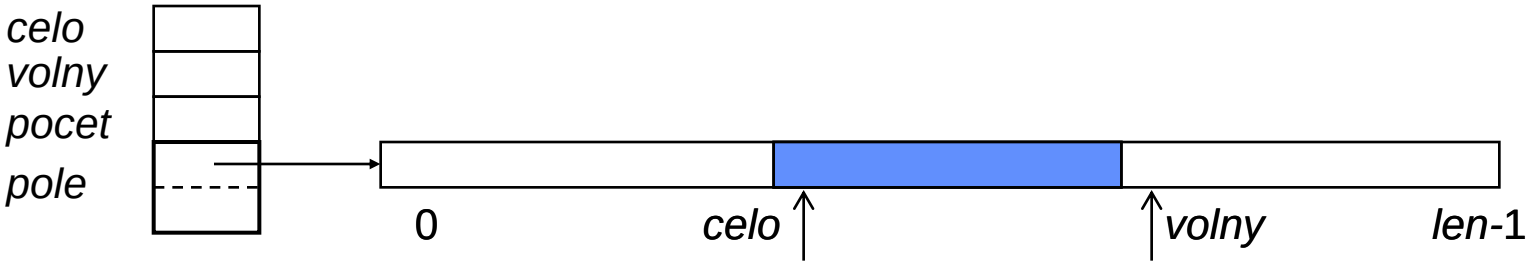
Implementace fronty

- Pomocí rozšiřitelného pole
- Připomeňme šablonu třídy *Array* pro rozšiřitelné pole

```
template <class Elem>
class Array {
public:
  Array(int=10);
  Array(const Array&);
  ~Array();
  int length() const {return len;}
  Elem& operator[](int);
  const Elem& operator[](int) const;
  Array& operator=(const Array&);
  void extend(int);
private:
  Elem* array;
  int len;
  void copy(const Array&);
};
```

Implementace fronty

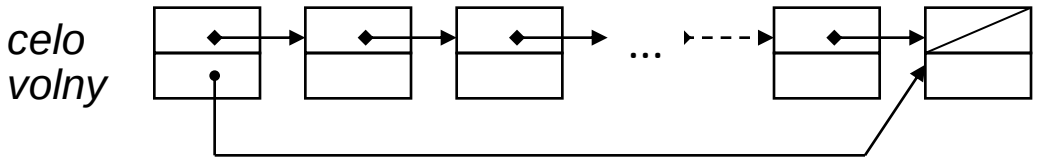
- Pomocí rozšiřitelného pole (instance šablony třídy *Array*)




```
class Queue {
public:
     Queue();
     bool empty() const {return pocet==0;}
     void add(const T&);
     T front() const;
     void remove();
private:
     int celo;
     int volny;
     int pocet;
     Array<T> pole;
};
```

Implementace fronty

- Pomocí spojového seznamu




```
class Queue {
public:
  Queue();
  ~Queue();
  void add(const T&);
  T front() const;
  void remove();
  bool empty() const;
private:
  struct Item {
    T val;
    Item *next;
    Item() {next = NULL;}
  };
  Item *celo, *volny;
  Queue(const Queue&);
  Queue& operator=(const Queue&);
};
```

ADT Seznam

- Seznam je sekvenční kontejner, pro který operace vložení, odebrání a čtení se provádějí na označené pozici, kterou lze měnit
- Příkladem seznamu je text zobrazený v okně Poznámkové bloku:
  - pozice je dána umístěním kurzoru
  - stiskem klávesy vložíme znak před kurzor
  - klávesou *Del* vyjmeme znak označný kurzorem
  - klávesou *Backspace* vyjmeme znak před kurzorem
  - klávesou ← resp. → přesuneme kurzor na předchozí resp. následující znak
  - klávesou *Home* resp. *End* přesunem kurzor na začátek resp. konec textu
- Vyzkoušejme ...

ADT Seznam

- Signatura operací nad seznamem (*T* je typ hodnot v seznamu):

`init: -> List`	prázdný seznam
`insert(_,_): List,T -> List`	vlož před kurzor
`remove(_): List -> List`	odeber
`removePrev(_): List -> List`	odeber před kurzorem
`toNext(_): List -> List`	kurzor na další
`toPrev(_): List -> List`	kurzor na předchozí
`toBegin(_): List -> List`	kurzor na začátek
`toEnd(_): List -> List`	kurzor na konec
`empty(_): List -> bool`	je seznam prázdný?
`atBegin(_): List -> bool`	je kurzor na začátku?
`atEnd(_): List -> bool`	je kurzor na konci?
`read(_): List -> T`	hodnota na kurzoru

Třída List

```
class List {
public:
  List();
  ~List();
  void insert(T);     // vloz pred kurzor
  void remove();      // odstran prvek na kurzoru
  void removePrev(); // odstran prvek pred kurzorem
  void toNext();      // kurzor na dalsi prvek
  void toPrev();      // kurzor na predchozi prvek
  void toBegin();     // kurzor na zacatek
  void toEnd();       // kurzor na konec
  bool empty();       // je seznam prazdny?
  bool atBegin();     // je kurzor na zacatku?
  bool atEnd();       // je kurzor na konci
  bool read(T&);      // cteni z pozice kurzoru
  friend ostream& operator<<(ostream&, const List&);
private:
  ...
};
```

Třída List

- Sémantiku operací si upřesníme na příkladech
- Příklad výpisu seznamu: seznam obsahující hodnoty *x*, *y* a *z*, kde na pozici kurzoru je *y*, vypíšeme ve tvaru
  [*x* |*y* *z*]
- Budeme uvažovat seznam celých čísel

```
List szn;                    [|]
szn.insert(10);              [10 |]
szn.insert(20);              [10 20 |]
szn.toPrev();                [10 |20]
szn.insert(30);              [10 30 |20]
szn.toNext();                [10 30 20 |]
szn.toBegin();               [|10 30 20]
szn.toEnd();                 [10 30 20 |]
szn.remove();                [10 30 20 |]
szn.removePrev();            [10 30 |]
szn.toPrev();                [10 |30]
szn.remove()                 [10 |]
...
```

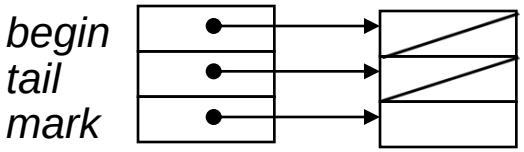
Jakou datovou strukturu použít pro implementaci?

- Chceme, aby všechny operace měly konstantní složitost
- Tento požadavek nesplňuje použití pole, kde při vkládání jinam než na konec by bylo třeba posouvat prvky pole (lineární složitost), podobně při odebírání
- Použijeme proto spojový seznam
- Objekt bude obsahovat ukazatel na první prvek a dále ukazatel na prvek označený kurzorem
- Aby operace „kurzor na předchozí prvek“ měla konstantní složitost, použijeme dvojsměrný spojový seznam
- Aby operace „kurzor na konec“ měla konstantní složitost, bude objekt obsahovat též ukazatel na konec seznamu
- Aby kurzor ukazující na konec seznamu bylo možné posunout na předchozí prvek, bude koncem seznamu poslední prvek spojového seznamu, ve kterém nebude žádná hodnota
- Grafické znázornění vnitřní reprezentace na dalším slajdu

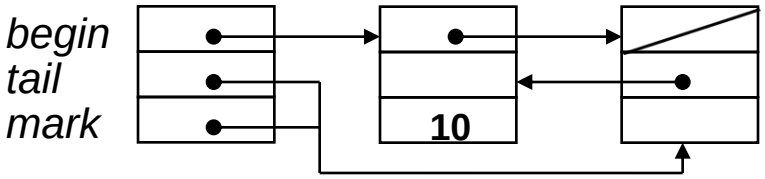
Implementace dvojsměrným spojovým seznamem

- Příklady seznamů a jejich vnitřní reprezentace

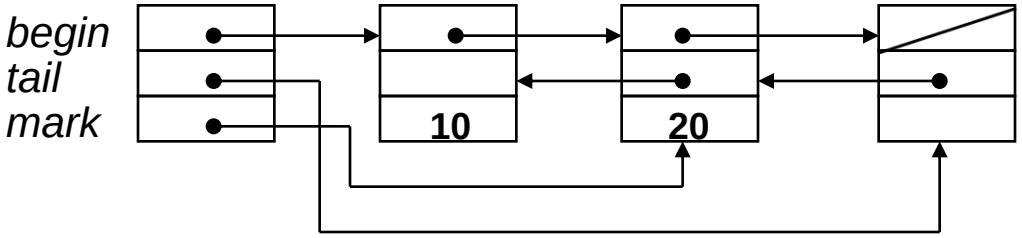
`[ | ]`



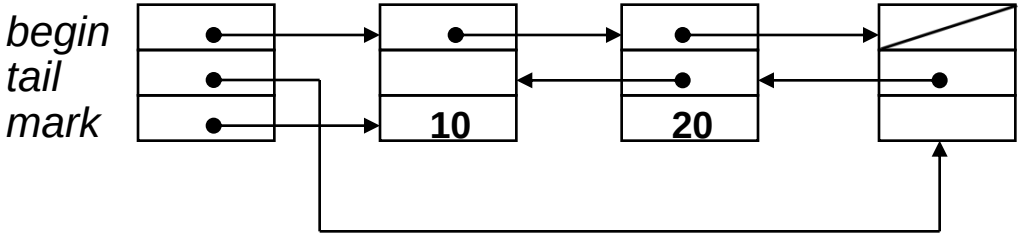
`[10 | ]`



`[10 |20]`



`[|10 20]`



Privátní část třídy List

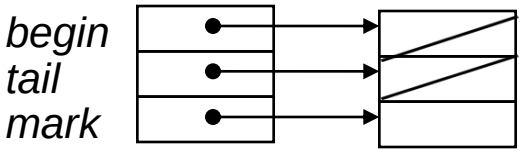
```
private:
  struct Node {
    T val;
    Node *next, *prev;
    Node(T x, Node *n) {
      // n nesmi byt NULL
      val = x; next = n; prev = n->prev;
    }
    Node() {next = NULL; prev = NULL;}
  };
  Node *begin;
  Node *tail;
  Node *mark;
  List(const List &);
  List& operator=(const List&);
};
```

- Konstruktor *Node*() bude použit pro inicializaci uzlu *tail*
- Konstruktor *Node*(*x*, *n*) bude použit pro inicializaci uzlu vkládaného před uzel, na který ukazuje *n* (*n* bude vždy ukazatel *mark*, tj, ukazatel na kurzorem označený uzel)

Konstruktor a destruktor třídy List

```
List::List() {
  begin = tail = mark = new Node;
}
```

prázdný seznam:




```
List::~List() {
  Node *p;
  while (begin) {
    p  = begin;
    begin = begin->next;
    delete p;
  }
}
```

Testy na prázdnost a pozici kurzoru

```
bool List::empty() {
  return begin == tail;
}

bool List::atBegin() {
  return mark==begin;
}

bool List::atEnd() {
  return mark==tail;
}
```

Přesuny kurzoru

```
void List::toBegin() {
  mark = begin;
}

void List::toEnd() {
  mark = tail;
}

void List::toNext() {
  if (atEnd()) return;
  mark = mark->next;
}

void List::toPrev() {
  if (atBegin()) return;
  mark = mark->prev;
}
```

Vložení prvku před kurzor

```
void List::insert(T x) {
  Node *p = new Node(x, mark);
  mark->prev = p;
  if (atBegin())
    begin = p;
  else
    p->prev->next = p;
}
```

- Poznámky:
  - položku *prev* v novém prvku nastaví konstruktor (zkopíruje *mark->prev*)
  - vkládá-li se před první prvek, je třeba *p* uložit do položky *begin*, jinak do položky *next* přechozího prvku

Odebrání prvku označeného kurzorem

```
void List::remove() {
  if (atEnd()) return;
  Node *p = mark;
  mark->next->prev = mark->prev;
  if (atBegin())
    begin = mark->next;
  else
    mark->prev->next = mark->next;
  mark = mark->next;
  delete p;
}
```

- Poznámky:
  - je-li kurzor na konci, neodebere se nic
  - ukazatel na další prvek (*mark->next*) se uloží do *begin*, pokus se odebírá první prvek, jinak do položky *next* předchozího prvku
  - kurzor se posune na prvek za odebraným prvkem

Odebrání prvku před kurzorem

```
void List::removePrev() {
  if (atBegin()) return;
  toPrev();
  remove();
}
```

- Poznámky:
  - je-li kurzor na začátku, neodebere se nic
  - jinak se kurzor přesune na předchozí prvek a odebere se tento prvek metodou *remove*

Čtení hodnoty prvku a výpis seznamu

```
bool List::read(T &x) {
  if (atEnd()) return false;
  x = mark->val;
  return true;
}
```

- Poznámka: je-li kurzor na konci, do proměnné se nic neuloží

```
ostream& operator<<(ostream& s, const List& lst) {
  s << '[';
  List::Node *p = lst.begin;
  while (p) {
    if (p==lst.mark) s << '|';
    if (p!=lst.tail) s << p->val << ' ';
    p = p->next;
  }
  s << ']';
  return s;
}
```

Třída List

- Testovací program je v souboru p10\list\main.cpp
- Šablona třídy List je v souboru p10/list-sablona/list.h, testovací program pro šablonu je p10\list-sablona\main.cpp
- Vyzkoušejte!