Struktury s obecnými daty. Moduly

Programovací techniky

doc. Ing. Jiří Rybička, Dr.
Ústav informatiky
PEF MENDELU v Brně
rybicka@mendelu.cz

Struktury a jejich data

Struktury a jejich data

- Seznamy, stromy apod. mají operace dvojího druhu:

Struktury a jejich data

- Seznamy, stromy apod. mají operace dvojího druhu:
  1. manipulace týkající se struktury;

Struktury a jejich data

- Seznamy, stromy apod. mají operace dvojího druhu:
  1. manipulace týkající se struktury;
  2. manipulace vyžadující znalost obsahu.

Struktury a jejich data

- Seznamy, stromy apod. mají operace dvojího druhu:
  1. manipulace týkající se struktury;
  2. manipulace vyžadující znalost obsahu.
- Snahou je vytvářet obecné struktury (nikoliv třeba „zásobník celých čísel“)

Struktury a jejich data

- Seznamy, stromy apod. mají operace dvojího druhu:
  1. manipulace týkající se struktury;
  2. manipulace vyžadující znalost obsahu.
- Snahou je vytvářet obecné struktury (nikoliv třeba „zásobník celých čísel“)
- Operace týkající se struktury jsou stále identické.

Struktury a jejich data

- Seznamy, stromy apod. mají operace dvojího druhu:
  1. manipulace týkající se struktury;
  2. manipulace vyžadující znalost obsahu.
- Snahou je vytvářet obecné struktury (nikoliv třeba „zásobník celých čísel“)
- Operace týkající se struktury jsou stále identické.
- Operace týkající se obsahu je potřebné nějak řešit.

Datový obsah

Datový obsah

- Chceme-li se vyhnout konkrétní datové složce nějaké struktury, je potřebné využít některého mechanismu obecných dat, např.:

Datový obsah

- Chceme-li se vyhnout konkrétní datové složce nějaké struktury, je potřebné využít některého mechanismu obecných dat, např.:
  - obecný ukazatel,

Datový obsah

- Chceme-li se vyhnout konkrétní datové složce nějaké struktury, je potřebné využít některého mechanismu obecných dat, např.:
  - obecný ukazatel,
  - struktura typu union,

Datový obsah

- Chceme-li se vyhnout konkrétní datové složce nějaké struktury, je potřebné využít některého mechanismu obecných dat, např.:
  - obecný ukazatel,
  - struktura typu union,
  - šablonování,

Datový obsah

- Chceme-li se vyhnout konkrétní datové složce nějaké struktury, je potřebné využít některého mechanismu obecných dat, např.:
  - obecný ukazatel,
  - struktura typu union,
  - šablonování,
  - dědičnost objektů.

Datový obsah

- Chceme-li se vyhnout konkrétní datové složce nějaké struktury, je potřebné využít některého mechanismu obecných dat, např.:
  - obecný ukazatel,
  - struktura typu union,
  - šablonování,
  - dědičnost objektů.
- Nejprůzračnější princip: obecný ukazatel

Datový obsah

- Chceme-li se vyhnout konkrétní datové složce nějaké struktury, je potřebné využít některého mechanismu obecných dat, např.:
  - obecný ukazatel,
  - struktura typu union,
  - šablonování,
  - dědičnost objektů.
- Nejprůzračnější princip: obecný ukazatel
- Možnost uložit do jakéhokoliv prvku jakýkoliv obsah, maximální jednoduchost a pružnost

Datový obsah

- Chceme-li se vyhnout konkrétní datové složce nějaké struktury, je potřebné využít některého mechanismu obecných dat, např.:
  - obecný ukazatel,
  - struktura typu union,
  - šablonování,
  - dědičnost objektů.
- Nejprůzračnější princip: obecný ukazatel
- Možnost uložit do jakéhokoliv prvku jakýkoliv obsah, maximální jednoduchost a pružnost
- Veškerá odpovědnost za správnost manipulace s obsahem je na uživateli struktury

Datový obsah

- Chceme-li se vyhnout konkrétní datové složce nějaké struktury, je potřebné využít některého mechanismu obecných dat, např.:
  - obecný ukazatel,
  - struktura typu union,
  - šablonování,
  - dědičnost objektů.
- Nejprůzračnější princip: obecný ukazatel
- Možnost uložit do jakéhokoliv prvku jakýkoliv obsah, maximální jednoduchost a pružnost
- Veškerá odpovědnost za správnost manipulace s obsahem je na uživateli struktury
- Technické řešení: datový typ **`void`***

Zásobník s obecnými daty

Zásobník s obecnými daty

- Upravíme strukturu jednoho prvku zásobníku:

```
typedef struct Prvek {
   void *Data; //obecná data
   Prvek *Dalsi;
} Prvek;
typedef Prvek *UkPrvek;  //ukazatel na prvek
typedef UkPrvek Zasobnik;  //datový typ zásobník
```

Zásobník s obecnými daty

- Upravíme strukturu jednoho prvku zásobníku:

```
typedef struct Prvek {
   void *Data; //obecná data
   Prvek *Dalsi;
} Prvek;
typedef Prvek *UkPrvek;  //ukazatel na prvek
typedef UkPrvek Zasobnik;  //datový typ zásobník
```

- Operace přidání (push) a odebrání (pop) jsou prakticky beze změny (jde o strukturní manipulaci).

Zásobník s obecnými daty

- Upravíme strukturu jednoho prvku zásobníku:

```
typedef struct Prvek {
   void *Data; //obecná data
   Prvek *Dalsi;
} Prvek;
typedef Prvek *UkPrvek;  //ukazatel na prvek
typedef UkPrvek Zasobnik;  //datový typ zásobník
```

- Operace přidání (push) a odebrání (pop) jsou prakticky beze změny (jde o strukturní manipulaci).
- O datech nyní nepotřebujeme nic vědět...

Zásobník s obecnými daty

- Operace zapíšeme jako podprogramy:

```
void push(Zasobnik &Z, void *E){
   UkPrvek Pom = new Prvek;
   Pom->Data = E; //Vložení dat
   Pom->Dalsi=Z; Z=Pom; //strukturní manipulace
}
void *pop(Zasobnik &Z){
   UkPrvek Pom = Z; void *vysled; //pro výsledek
   if (Pom != NULL) {
      Z = Pom->Dalsi;  //nový vrchol
      vysled = Pom->Data;
      delete Pom; //odstranění z paměti
   } else vysled = NULL;
   return vysled;
}
```

Obecný zásobník – použití

- Vstupem jsou celá čísla, chceme obrátit jejich pořadí.

```
int vstup, *U;
Zasobnik S=NULL;
while (cin>>vstup){ //uložení do zásobníku
   U = new int; //NOVÝ! ukazatel na data
   //POZOR! U = &vstup; projde, ale je chybný!!
   *U = vstup;  //uložení čtené hodnoty
   push(S, U);  //U je kompatibilní s void*
}
while (S!=NULL) { //výpis na výstup
   U = (int *)pop(S);
   cout << *U << " ";
}
```

Moduly

- Účel: rozdělit programové dílo na logické celky

Moduly

- Účel: rozdělit programové dílo na logické celky
- Jsou samostatně kompilovatelné a použitelné v různých souvislostech

Moduly

- Účel: rozdělit programové dílo na logické celky
- Jsou samostatně kompilovatelné a použitelné v různých souvislostech
- Vhodná rekvizita pro implementaci abstraktního typu dat

Moduly

- Účel: rozdělit programové dílo na logické celky
- Jsou samostatně kompilovatelné a použitelné v různých souvislostech
- Vhodná rekvizita pro implementaci abstraktního typu dat
- Snaha vytvářet obecné struktury – „rozumná“ obecnost!

Moduly

- Účel: rozdělit programové dílo na logické celky
- Jsou samostatně kompilovatelné a použitelné v různých souvislostech
- Vhodná rekvizita pro implementaci abstraktního typu dat
- Snaha vytvářet obecné struktury – „rozumná“ obecnost!
- Provedení: deklarační část (veřejně dostupné prvky) + implementační část

Moduly

- Účel: rozdělit programové dílo na logické celky
- Jsou samostatně kompilovatelné a použitelné v různých souvislostech
- Vhodná rekvizita pro implementaci abstraktního typu dat
- Snaha vytvářet obecné struktury – „rozumná“ obecnost!
- Provedení: deklarační část (veřejně dostupné prvky) + implementační část
- Kompilace všech zdrojů + sestavení (linking) zajistí, že se do výsledného programu dostanou všechny odkazované součásti

```
g++ -o vysled *.cpp
```

Realizace modulů

- Veřejná část: soubor `neco.h`, implementační část: soubor `neco.cpp`

Realizace modulů

- Veřejná část: soubor `neco.h`, implementační část: soubor `neco.cpp`
- Kompilační chaos: nebezpečí vícenásobného zahrnutí stejného zdroje do kompilace se řeší vhodnými direktivami překladače

Realizace modulů

- Veřejná část: soubor `neco.h`, implementační část: soubor `neco.cpp`
- Kompilační chaos: nebezpečí vícenásobného zahrnutí stejného zdroje do kompilace se řeší vhodnými direktivami překladače
- Systém:

Realizace modulů

- Veřejná část: soubor `neco.h`, implementační část: soubor `neco.cpp`
- Kompilační chaos: nebezpečí vícenásobného zahrnutí stejného zdroje do kompilace se řeší vhodnými direktivami překladače
- Systém:
  - Direktiva `#ifndef MAKRO` – podmíněná kompilace při neexistenci makra

Realizace modulů

- Veřejná část: soubor `neco.h`, implementační část: soubor `neco.cpp`
- Kompilační chaos: nebezpečí vícenásobného zahrnutí stejného zdroje do kompilace se řeší vhodnými direktivami překladače
- Systém:
  - Direktiva `#ifndef MAKRO` – podmíněná kompilace při neexistenci makra
  - Direktiva `#endif` – konec podmíněné kompilace

Realizace modulů

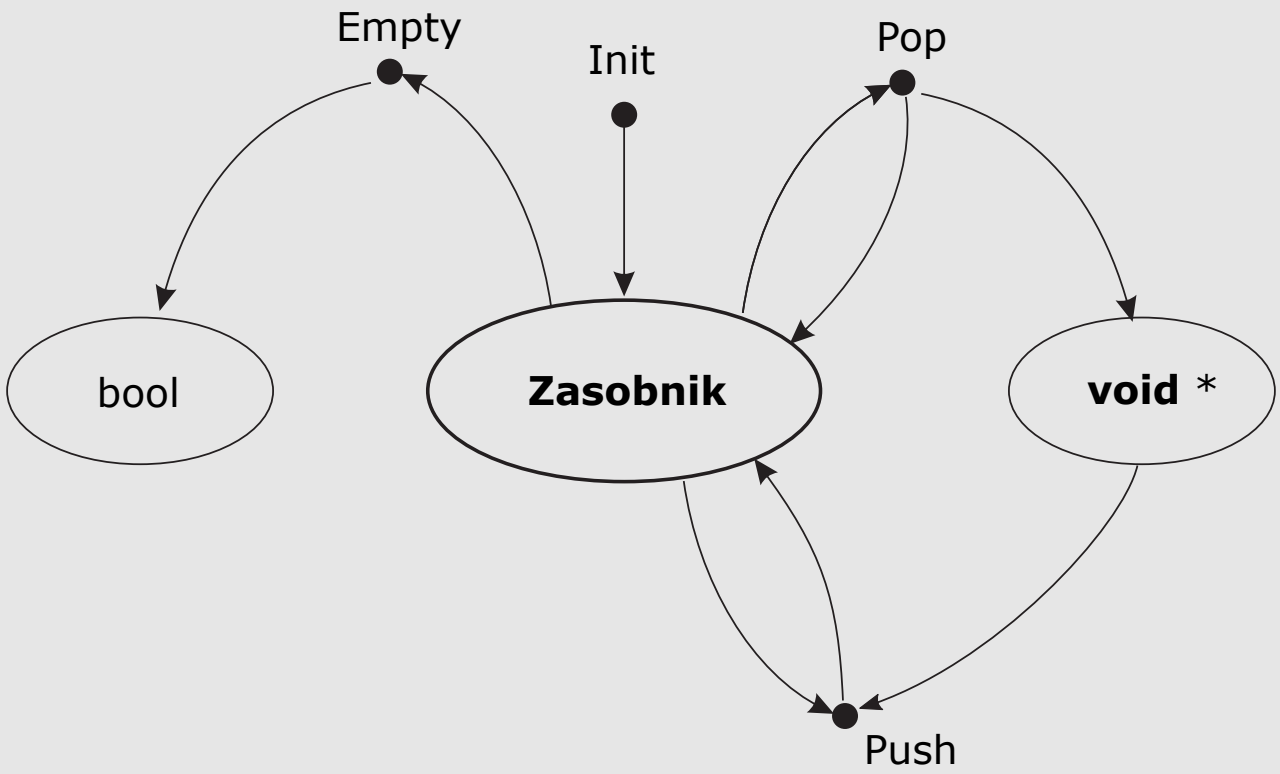
- Veřejná část: soubor `neco.h`, implementační část: soubor `neco.cpp`
- Kompilační chaos: nebezpečí vícenásobného zahrnutí stejného zdroje do kompilace se řeší vhodnými direktivami překladače
- Systém:
  - Direktiva `#ifndef MAKRO` – podmíněná kompilace při neexistenci makra
  - Direktiva `#endif` – konec podmíněné kompilace
  - Direktiva `#define MAKRO` – definice makra

Realizace modulů

- Veřejná část: soubor `neco.h`, implementační část: soubor `neco.cpp`
- Kompilační chaos: nebezpečí vícenásobného zahrnutí stejného zdroje do kompilace se řeší vhodnými direktivami překladače
- Systém:
  - Direktiva `#ifndef MAKRO` – podmíněná kompilace při neexistenci makra
  - Direktiva `#endif` – konec podmíněné kompilace
  - Direktiva `#define MAKRO` – definice makra
- Užitečná praxe: Makro je odvozeno od jména hlavičkového souboru (jednoznačnost, snadná orientace)

Implementace zásobníku modulem

- Veřejná část `zasob.h` je obrazem diagramu signatury:



Implementace zásobníku modulem


```
#ifndef ZASOB_H
#define ZASOB_H
//Potřebné datové typy z diagramu signatury:
typedef struct Prvek {
   void *Data; //obecná data
   Prvek *Dalsi;
} Prvek;
typedef Prvek *UkPrvek;   //ukazatel na prvek
typedef UkPrvek Zasobnik; //datový typ zásobník
//Operace:
void Init(Zasobnik &Z);
void Push(Zasobnik &Z, void *E);
void *Pop(Zasobnik &Z);
bool Empty(Zasobnik Z);
#endif
```

Implementace zásobníku modulem

- Sémantické definice (zejména operace Pop):

Implementace zásobníku modulem

- Sémantické definice (zejména operace Pop):
- Operace Init:

$$\Rightarrow \langle]$$

Implementace zásobníku modulem

- Sémantické definice (zejména operace Pop):
- Operace Init:

$$\Rightarrow \langle]$$

- Operace Push:

$$x, \langle a_1, \ldots, a_n] \Rightarrow \langle x, a_1, \ldots, a_n]$$

Implementace zásobníku modulem

- Sémantické definice (zejména operace Pop):
- Operace Init:

$$\Rightarrow \langle]$$

- Operace Push:

$$x, \langle a_1, \ldots, a_n] \Rightarrow \langle x, a_1, \ldots, a_n]$$

- Operace Pop:

$$\begin{aligned} \langle a_1, a_2, \ldots, a_n] &\Rightarrow a_1, \langle a_2, \ldots, a_n] \\ \langle] &\Rightarrow NULL, \langle] \end{aligned}$$

Implementace zásobníku modulem

- Sémantické definice (zejména operace Pop):
- Operace Init:

$$\Rightarrow \langle]$$

- Operace Push:

$$x, \langle a_1, \ldots, a_n] \Rightarrow \langle x, a_1, \ldots, a_n]$$

- Operace Pop:

$$\begin{aligned} \langle a_1, a_2, \ldots, a_n] &\Rightarrow a_1, \langle a_2, \ldots, a_n] \\ \langle] &\Rightarrow NULL, \langle] \end{aligned}$$

- Operace Empty:

$$\begin{aligned} \langle a_1, \ldots, a_n] &\Rightarrow \text{false} \\ \langle] &\Rightarrow \text{true} \end{aligned}$$

Implementace zásobníku modulem

```
#include <iostream>
#include ”zasob.h”

void Init(Zasobnik &Z){
   Z = NULL;
}
void Push(Zasobnik &Z, void *E){
  ... //tělo procedury
}
void *Pop(Zasobnik &Z){
  ... //tělo funkce
}
bool Empty(Zasobnik Z){
   return Z==NULL;
}
```

Užití modulu

```
#include <iostream>
#include ”zasob.h”
using namespace std;
int main(){
  int vstup, *U; Zasobnik S;  Init(S);
  while (cin>>vstup){ //uložení do zásobníku
     U = new int; //NOVÝ! ukazatel na data
     *U = vstup;  //uložení čtené hodnoty
     Push(S, U);  //U je kompatibilní s void*
  }
  while (not Empty(S)) { //výpis na výstup
     U = (int*)Pop(S); cout << *U << ” ”;
  }
  return 0;
}
```

Ukazatel na podprogram

- Podprogram leží v operační paměti stejně jako jakákoliv data

Ukazatel na podprogram

- Podprogram leží v operační paměti stejně jako jakákoliv data
- Ukazatel na podprogram umožňuje považovat podprogram za data

Ukazatel na podprogram

- Podprogram leží v operační paměti stejně jako jakákoliv data
- Ukazatel na podprogram umožňuje považovat podprogram za data
- Nástroj implementace metod objektů a umožňuje polymorfismus

Ukazatel na podprogram

- Podprogram leží v operační paměti stejně jako jakákoliv data
- Ukazatel na podprogram umožňuje považovat podprogram za data
- Nástroj implementace metod objektů a umožňuje polymorfismus
- Proměnná typu podprogram je použitelná stejně jako podprogram, který je do ní přiřazen

Ukazatel na podprogram

- Podprogram leží v operační paměti stejně jako jakákoliv data
- Ukazatel na podprogram umožňuje považovat podprogram za data
- Nástroj implementace metod objektů a umožňuje polymorfismus
- Proměnná typu podprogram je použitelná stejně jako podprogram, který je do ní přiřazen
- Syntaktický koncept: `proc()` = volání podprogramu; `proc` = adresa podprogramu

Ukazatel na podprogram

- Podprogram leží v operační paměti stejně jako jakákoliv data
- Ukazatel na podprogram umožňuje považovat podprogram za data
- Nástroj implementace metod objektů a umožňuje polymorfismus
- Proměnná typu podprogram je použitelná stejně jako podprogram, který je do ní přiřazen
- Syntaktický koncept: `proc()` = volání podprogramu; `proc` = adresa podprogramu
- Užití: funkcionální (procedurální) parametr; například integrál reálné funkce:

```
double Integral(RealFce F, double A, double B){
   ... //numerická integrace funkce F
}
```

Definice typu ukazatele na podprogram

- Ukazatel na podprogram – syntax se odlišuje od výstupní hodnoty funkce typu ukazatel, například:

```
typedef double (*RealFce)(double X);
  //závorky ohraničují ukazatel na funkci
double *RealFce(double X);
  //definice funkce typu ukazatel na double

double plocha(double A){
   return 2*M_PI*A*A;
}
RealFce fce;
fce=plocha;
double vstup;
cin>>vstup;
cout << fce(vstup) << endl;
```

Příklad

```
typedef double (*RealFce)(double X);
double mojefce(double A){
   return (cos(A)-1)/(sin(A)+3);
}
double Integral(RealFce F, double A, double B){
 //numerická integrace funkce F v intervalu <A,B>
 double suma=0, Krok=(B-A)/1000000;
  for (double X=A+Krok; X<B; X+=Krok) suma+=F(X);
  return (F(A)+F(B))/2+suma*Krok;
}
//použití:
double mezA, mezB;
cin>>mezA >> mezB;
cout << Integral(mojefce, mezA, mezB) << endl;
cout << Integral(sin, mezA, mezB) << endl;
```

Operace s obecnými daty

- V modulu není žádná informace o konkrétních datových typech uložených dat

Operace s obecnými daty

- V modulu není žádná informace o konkrétních datových typech uložených dat
- Některé manipulace vyžadují znalost dat, například porovnání, zobrazení

Operace s obecnými daty

- V modulu není žádná informace o konkrétních datových typech uložených dat
- Některé manipulace vyžadují znalost dat, například porovnání, zobrazení
- Informaci o uložených datech zná pouze uživatel modulu

Operace s obecnými daty

- V modulu není žádná informace o konkrétních datových typech uložených dat
- Některé manipulace vyžadují znalost dat, například porovnání, zobrazení
- Informaci o uložených datech zná pouze uživatel modulu
- Uvnitř modulu jsou pouze hlavičky podprogramů

Operace s obecnými daty

- V modulu není žádná informace o konkrétních datových typech uložených dat
- Některé manipulace vyžadují znalost dat, například porovnání, zobrazení
- Informaci o uložených datech zná pouze uživatel modulu
- Uvnitř modulu jsou pouze hlavičky podprogramů
- Těla podprogramů definuje uživatel modulu

Operace s obecnými daty

- V modulu není žádná informace o konkrétních datových typech uložených dat
- Některé manipulace vyžadují znalost dat, například porovnání, zobrazení
- Informaci o uložených datech zná pouze uživatel modulu
- Uvnitř modulu jsou pouze hlavičky podprogramů
- Těla podprogramů definuje uživatel modulu
- Formální parametry jsou vždy obecné ukazatele, skutečné parametry je nutné přetypovat (konvertovat)

Operace s obecnými daty

- V modulu není žádná informace o konkrétních datových typech uložených dat
- Některé manipulace vyžadují znalost dat, například porovnání, zobrazení
- Informaci o uložených datech zná pouze uživatel modulu
- Uvnitř modulu jsou pouze hlavičky podprogramů
- Těla podprogramů definuje uživatel modulu
- Formální parametry jsou vždy obecné ukazatele, skutečné parametry je nutné přetypovat (konvertovat)
- Přetypování: `(novy_typ)(vyraz)`

Operace s obecnými daty

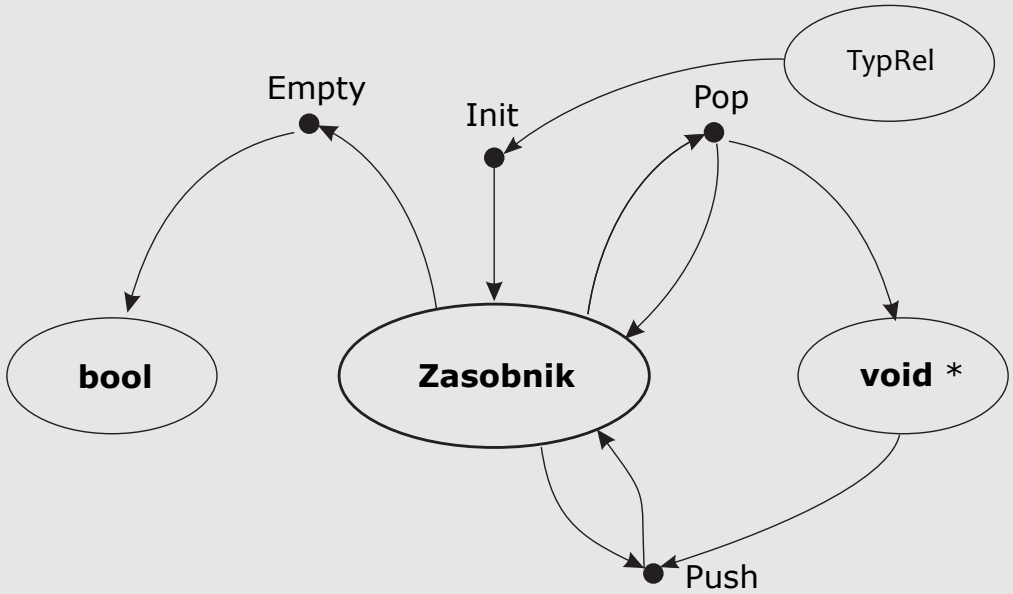
- V modulu není žádná informace o konkrétních datových typech uložených dat
- Některé manipulace vyžadují znalost dat, například porovnání, zobrazení
- Informaci o uložených datech zná pouze uživatel modulu
- Uvnitř modulu jsou pouze hlavičky podprogramů
- Těla podprogramů definuje uživatel modulu
- Formální parametry jsou vždy obecné ukazatele, skutečné parametry je nutné přetypovat (konvertovat)
- Přetypování: `(novy_typ)(vyraz)`
- V některých případech lze využít kompatibility obecného ukazatele s libovolným jiným ukazatelem.

Příklad

- Uvažujme opět zásobník, nyní ovšem se speciální vlastností: neobsahuje duplicitní hodnoty. Při vkládání se testuje, zda je nová hodnota unikátní, opakované hodnoty se nevloží.

Příklad

- Uvažujme opět zásobník, nyní ovšem se speciální vlastností: neobsahuje duplicitní hodnoty. Při vkládání se testuje, zda je nová hodnota unikátní, opakované hodnoty se nevloží.
- Změna diagramu signatury: Operace `Init` definuje způsob porovnání dat, jejím vstupem je typ logické funkce, která je schopna porovnat uživatelská data (rovnost = **`true`**)



Příklad

- Modifikace hodnot typu Zasobnik: $(\langle a_1, a_2, \ldots, a_n]; R)$

Příklad

- Modifikace hodnot typu Zasobnik: $(\langle a_1, a_2, \ldots, a_n]; R)$
- Modifikace sémantiky operací Init a Push:

Příklad

- Modifikace hodnot typu Zasobnik: $(\langle a_1, a_2, \ldots, a_n]; R)$
- Modifikace sémantiky operací Init a Push:
- Operace Init:

$$R \Rightarrow (\langle]; R)$$

Příklad

- Modifikace hodnot typu Zasobnik: $(\langle a_1, a_2, \ldots, a_n]; R)$
- Modifikace sémantiky operací Init a Push:
- Operace Init:

$$R \Rightarrow (\langle]; R)$$

- Operace Push:

$$x, (\langle a_1, \ldots, a_n]; R), \quad \exists i \in \{1, \ldots, n\}, R(x, a_i) = true \Rightarrow (\langle a_1, \ldots, a_n]; R)$$

$$x, (\langle a_1, \ldots, a_n]; R), \quad \forall i \in \{1, \ldots, n\}, R(x, a_i) = false \Rightarrow (\langle x, a_1, \ldots, a_n]; R)$$

Příklad

- Modifikace hodnot typu Zasobnik: $(\langle a_1, a_2, \ldots, a_n]; R)$
- Modifikace sémantiky operací Init a Push:
- Operace Init:

$$R \Rightarrow (\langle]; R)$$

- Operace Push:

$$x, (\langle a_1, \ldots, a_n]; R), \quad \exists i \in \{1, \ldots, n\}, R(x, a_i) = true \Rightarrow (\langle a_1, \ldots, a_n]; R)$$

$$x, (\langle a_1, \ldots, a_n]; R), \quad \forall i \in \{1, \ldots, n\}, R(x, a_i) = false \Rightarrow (\langle x, a_1, \ldots, a_n]; R)$$

- Operace Pop a Empty pracují s modifikovanými hodnotami stejně jako s původními

Implementace

- Datový typ Zasobnik bude nyní obsahovat dvě složky:

```
typedef bool (*TypSrov)(void *A, void *B);
  //datový typ relační funkce
typedef struct {  //datový typ zásobník
   UkPrvek Vrchol;
   TypSrov Rovno;
} Zasobnik;
```

Implementace

- Datový typ Zasobnik bude nyní obsahovat dvě složky:

```
typedef bool (*TypSrov)(void *A, void *B);
  //datový typ relační funkce
typedef struct {  //datový typ zásobník
   UkPrvek Vrchol;
   TypSrov Rovno;
} Zasobnik;
```

- Operace Init:

```
void Init(Zasobnik &Z, TypSrov R){
   Z.Vrchol = NULL;
   Z.Rovno = R;
}
```

Implementace

- Operace Push:

```
bool Pritomen(Zasobnik Z, void *E){
  UkPrvek Pom = Z.Vrchol;   //sekvenční hledání
  while (Pom!=NULL and not Z.Rovno(Pom->Data, E))
     Pom=Pom->Dalsi;
  return Pom!=NULL;
}
void Push(Zasobnik &Z, void *E){
   if (not Pritomen(Z, E)){//nenalezen, vkládá se
     UkPrvek Pom = new Prvek;
     Pom->Data = E; //Vložení dat
     Pom->Dalsi = Z.Vrchol;
     Z.Vrchol = Pom;
   }
}
```

Implementace

- Užití modulu:

Implementace

- Užití modulu:
- Pouze zde je soustředěna informace o datovém typu vkládaných dat

```
bool MojeSrov(void *A, void *B){
  int *X = (int*)A, *Y = (int*)B; //konverze
  return *X==*Y;
}

int main(){
  int vstup, *U;
  Zasobnik S;
  Init(S, MojeSrov); //vložení relační funkce
  ... //zbytek je stejný
```