STU FIIT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Základy procedurálneho programovania 2

FIIT STU, Mlynská dolina

Aula Minor, pondelok 9:00

letný semester

2016/2017

Ideme podľa plánu

dátum	prednáška	8:00	9:00	cvičenie	obsah
20.3.	6		Čítanie kódu, Hľadanie chýb v kóde	6	Projekt 1: odovzdanie
27.3.	7	Test 3	Riešenie testu 3, Spájané zoznamy	7	
3.4.	8		Tezeus, Bitové operácie	8	Projekt 2 **Tezeus a Minotaurus**
10.4.	9	Test 4	Riešenie testu 4, Rekurzia, Minotaurus	9	
17.4.	Veľká noc			X	
24.4.	10		Ďalšie prvky jazyka C	10	Projekt 2: odovzdanie, konzultovanie
1.5.	Sviatok			11	
8.5.	Sviatok			12	
9.5.	11		Opakovanie	X	
15.5.	12	Predtermín?		X	

Tretí priebežný test ... A

- A (max. 1b): Daný je názov súboru a pole reťazcov, napíš hlavičku funkcie **write**, ktorá do súboru zapíše toto pole reťazcov, každý reťazec do samostatného riadku. Nepoužívajte globálne premenné.
- **Riešenie:**

  void write(char *nazov, char **pole, int pocet);

Tretí priebežný test ... B (riešenie)

B (max. 2b): Daný je zdrojový kód (vľavo) a zistite, ktoré upozornenia a chyby obsahuje a do tabuľky (vpravo) uveďte riadky zdrojového kódu, v ktorých sa nachádzajú. Otáznik ??? zakrýva konkrétny názov.

```
 1  #include <stdio.h>
 2
 3  int strdelete(char *str, int n, int offset)
 4  {
 5    // sem napis svoje riesenie
 6    int i, len = strlen(str);
 7    if (len-offset < n)
 8       return;
 9    for (i = offset; i = len-n; i++)
10       str[i] = str[i+n]
11    str[i] = 0;
12    return 0;
13  }
14
15  int main()
16  {
17    char buf[100];
18
19   sprintf(buf, 'totojedruhyretazec');
20   if (str_delete(buf, x, y))
21     printf("Nepodarilo sa vymazat.\n");
22    else
23     printf('%s', buf);
24   return;
25  }
```

Chyby alebo upozornenia	Riadky
error: '???' undeclared (first use in this function)	20
error: expected ';' before '???'	11
warning: passing argument 1 of '???' makes pointer from integer without a cast	19, 23
warning: 'return' with no value, in function returning non-void	8, 24
warning: suggest parentheses around assignment used as truth value	9
warning: multi-character character constant	19, 23
warning: implicit declaration of function '???'	6, 20

STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Základy procedurálneho programovania 2

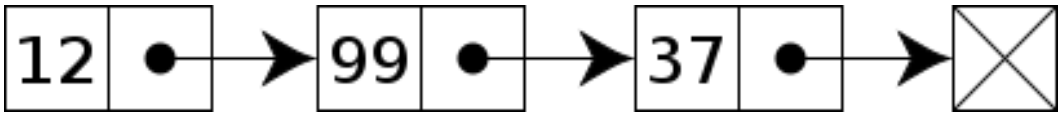
Prepojené dátové štruktúry

27.3.2017

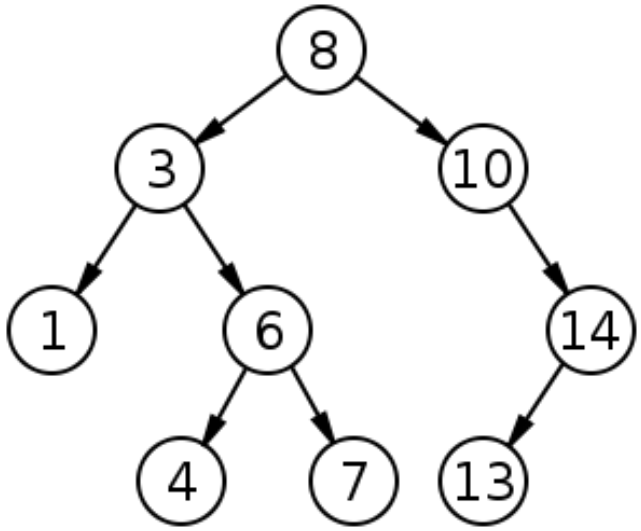
letný semester 2016/2017

Prepojené dátové štruktúry

- Angl. linked data structures
- Údaje prepojené (medzi sebou) linkami
- Linky sú ako samostatné údaje, ktoré možno prechádzať, porovnávať, upravovať
- Linky sú zvyčajne implementované
  - **smerníkmi,**
  - alternatívne ako indexy do poľa
- Príklady:
  - Spájaný zoznam (angl. linked list)

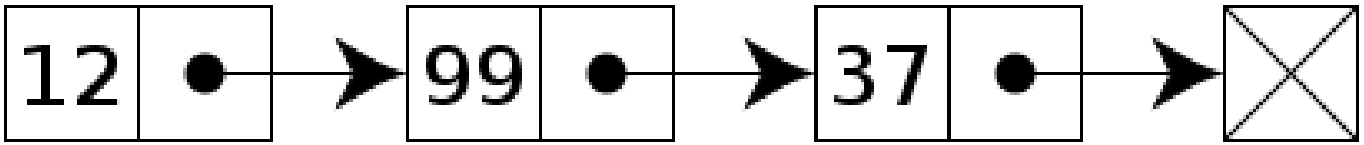


  - Binárny vyhľadávací strom (angl. binary search tree)



Spájaný zoznam (linked list)

- Jednosmerne zreťazený spájaný zoznam



- Reprezentácia v programovacom jazyku C
  - Dve štruktúry: zoznam, prvok zoznamu

```
struct Zoznam
{
    struct Prvok *prvy;
};

struct Prvok
{
    struct Prvok *dalsi;
    int data;
};
```

Vytvoriť spájaný zoznam

- Operácia vytvorenia nového spájaného zoznamu
- Vstupy (vstupné parametre)?
  - Žiadne
- Stav pred vykonaním:
  - Nič neexistuje
- Stav po vykonaní:
  - V pamäti existuje štruktúra Zoznam bez prvkov.
- Výstupy (návratová hodnota)?
  - Smerník na novovytvorenú štruktúru Zoznam

```
struct Zoznam *zoznam_vytvor()
```

Vytvorenie spájaného zoznamu (impl.)

- Operácia vytvorenia nového spájaného zoznamu

```
struct Zoznam *zoznam_vytvor()
{
    struct Zoznam *z = (struct Zoznam *)malloc(sizeof(struct Zoznam));
    z->prvy = NULL;
    return z;
}
```

  - Alternatívne:

```
struct Zoznam *zoznam_vytvor()
{
    return (struct Zoznam *)calloc(1, sizeof(struct Zoznam));
}
```

- Zoznam, ktorý neobsahuje prvky je platným zoznamom
- Použitie v programe:

```
struct Zoznam *moj1 = zoznam_vytvor();
struct Zoznam *moj2 = zoznam_vytvor();
```

Vypísať obsah spájaného zoznamu

- Operácia vypísania spájaného zoznamu
- Vstupy (vstupné parametre)?
  - Spájaný zoznam, ktorý chceme vypísať
- Stav pred vykonaním:
  - Spájaný zoznam na vstupe musí byť korektný spájaný zoznam, obsahujúci správne spojené prvky.
- Stav po vykonaní:
  - Na obrazovku vypísané hodnoty prvkov v spájanom zozname v poradí v akom sú v spájanom zozname.
- Výstupy (návratová hodnota)?
  - Žiadne, resp. počet vypísaných prvkov, podobne ako printf

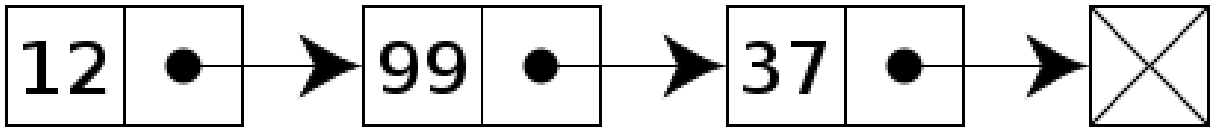
```
int zoznam_vypis(struct Zoznam *z)
```

Vypísať obsah spájaného zoznamu (impl.)

- Operácia vypísania spájaného zoznamu

```
int zoznam_vypis(struct Zoznam *z)
{
    struct Prvok *p = z->prvy;
    while (p != NULL)
    {
        printf("%d\n", p->data);
        p = p->dalsi;
    }
    return 0;
}
```

- Čo vypíše pre zoznam?



Vložiť do spájaného zoznamu

- Operácia vloženia údaju do spájaného zoznamu
- Vstupy (vstupné parametre)?
  - Spájaný zoznam, do ktorého chceme vložiť
  - Údaj, ktorý chceme vložiť
- Stav pred vykonaním:
  - Spájaný zoznam na vstupe musí byť korektný spájaný zoznam, obsahujúci správne spojené prvky.
- Stav po vykonaní:
  - Korektný spájaný zoznam zo vstupu, obsahujúci o jeden prvok (obsahujúci údaj, ktorý sme chceli vložiť) navyše.
- Výstupy (návratová hodnota)?
  - Žiadne.

```
void zoznam_vloz(struct Zoznam *z, int cislo)
```

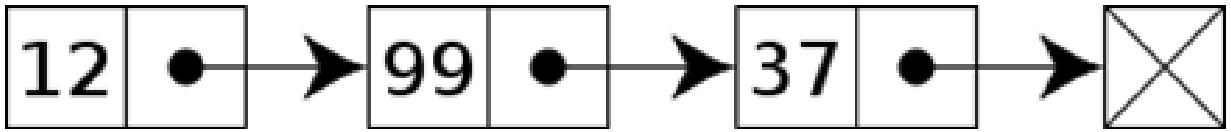
Vložiť do spájaného zoznamu (impl.)

- Operácia vloženia údaju do spájaného zoznamu

```
struct Prvok *zoznam_vloz(struct Zoznam *z, int cislo)
{
    z->prvy = prvok_vytvor(cislo, z->prvy);
}
```

- Vkladáme 12 do zoznamu

  - Vytvoríme nový prvok, ktorý bude: obsahovať údaj 12, a linka na ďalší bude ukazovať na začiatok zoznamu
  - Nový začiatok nastavíme na tento nový prvok



Odstrániť zo spájaného zoznamu

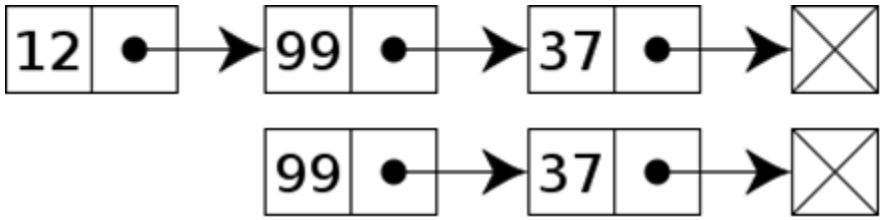
- Operácia odstránenie údaju zo spájaného zoznamu
- Vstupy (vstupné parametre)?
  - Spájaný zoznam, z ktorého chceme odstrániť
  - Údaj, ktorý chceme odstrániť
- Stav pred vykonaním:
  - Spájaný zoznam na vstupe musí byť korektný spájaný zoznam, obsahujúci správne spojené prvky.
- Stav po vykonaní:
  - Korektný spájaný zoznam zo vstupu, možno obsahujúci o jeden prvok (obsahujúci údaj, ktorý sme chceli odstrániť) menej, ak sa v pôvodnom zozname taký prvok nachádza.
- Výstupy (návratová hodnota)?
  - Žiadne.

```
void zoznam_odstran(struct Zoznam *z, int cislo)
```

Odstrániť zo spájaného zoznamu (impl.)

- Operácia odstránenie údaju zo spájaného zoznamu

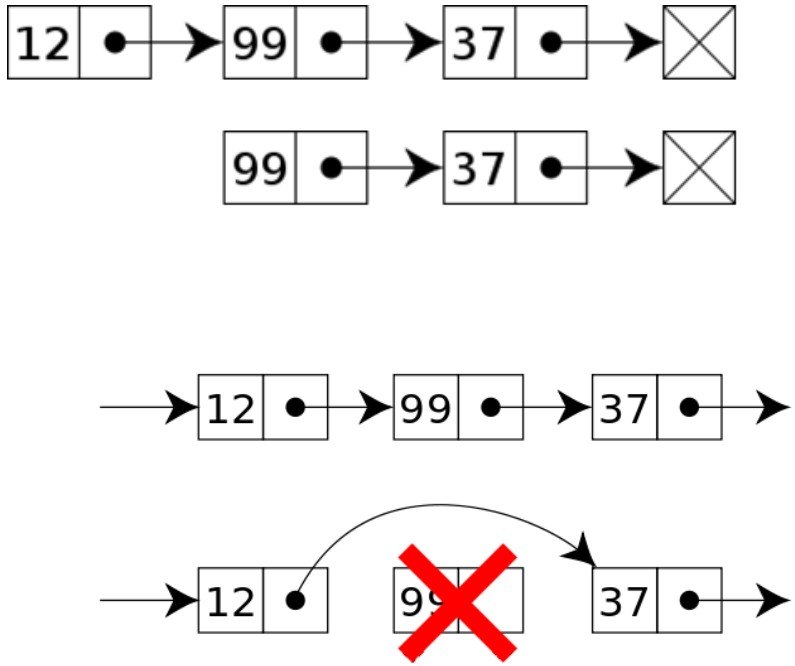
```
void zoznam_odstran(struct Zoznam *z, int cislo)
{
    struct Prvok *p = z->prvy;
    if (p && p->data == cislo)
    {
        z->prvy = p->dalsi;
        free(p); return;
    }

    // ...
}
```



Odstrániť zo spájaného zoznamu (impl.)

- Operácia odstránenie údaju zo spájaného zoznamu

```
void zoznam_odstran(struct Zoznam *z, int cislo)
{
    struct Prvok *p = z->prvy;
    if (p && p->data == cislo)
    {
        z->prvy = p->dalsi;
        free(p); return;
    }
    while (p != NULL)
    {
        struct Prvok *q = p->dalsi;
        if (q && q->data == cislo)
        {
            p->dalsi = q->dalsi;
            free(q); break;
        }
        p = q;
    }
}
```



Použitie v programe

- Takto naprogramované operácie môžeme ľahko použiť v programe
- Príklad použitia:

```
zoznam_vloz(moj, 47);
zoznam_odstran(moj, 47);

// Otazka: Bude zoznam moj opat v povodnom stave?
```

- Rozšírenie implementácie:
  - Operácie vlož, resp. odstráň by mohli vracať prvok, ktorý sa vložil, resp. odstránil

```
struct Prvok *zoznam_vloz(struct Zoznam *z, int cislo)
```

Použitie v programe

- Operácia odstránenia podľa údaju v sebe obsahuje (nevhodne) spojené dve operácie:
  - **vyhľadaj údaj (resp. hľadaj prvok)**, a
  - **odstráň prvok**.
- Lepšie, keď odstraňujeme nie údaj ale prvok:

```
void zoznam_odstran(struct Zoznam *z, struct Prvok *p)
```

- Príklad použitia:

```
struct Prvok *p = zoznam_vloz(moj, 47);
zoznam_odstran(moj, p);

// Otazka: Bude zoznam moj opat v povodnom stave?
```

Použitie v programe

- Ak chceme odstrániť údaj, tak prvok najskôr vyhľadáme operáciou **hľadaj**:

```
struct Prvok *zoznam_hladaj(struct Zoznam *z, int cislo)
```

- Operácia odstránenia podľa údaju:

```
void zoznam_odstran(struct Zoznam *z, int cislo)
{
    struct Prvok *p = zoznam_hladaj(z, cislo);
    if (p != NULL)
        zoznam_odstran(z, p);
}
```

Hľadať prvok v spájanom zozname

- Operácia vyhľadania prvku podľa údaju
- Vstupy (vstupné parametre)?
  - Spájaný zoznam, do ktorého chceme vložiť
  - Údaj, ktorý chceme vyhľadať
- Stav pred vykonaním:
  - Spájaný zoznam na vstupe musí byť korektný spájaný zoznam, obsahujúci správne spojené prvky.
- Stav po vykonaní:
  - Ten istý korektný spájaný zoznam zo vstupu bez zmeny
- Výstupy (návratová hodnota)?
  - Prvok, ktorý obsahuje hľadaný údaj, alebo
  - NULL ak taký prvok neexistuje.

```
struct Prvok *zoznam_hladaj(struct Zoznam *z, int cislo)
```

Hľadať prvok v spájanom zozname (imlp.)

- Operácia vyhľadania prvku podľa údaju

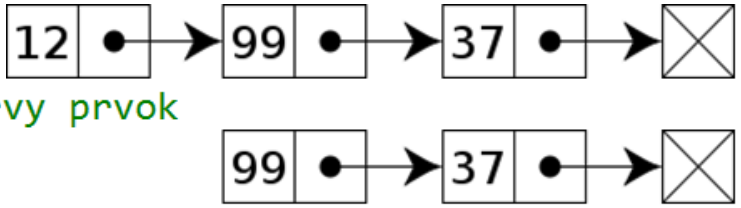
```
struct Prvok *zoznam_hladaj(struct Zoznam *z, int cislo)
{
    struct Prvok *p = z->prvy;
    while (p != NULL)
    {
        if (p->data == cislo)
            return p;
        p = p->dalsi;
    }
    return NULL;
}
```

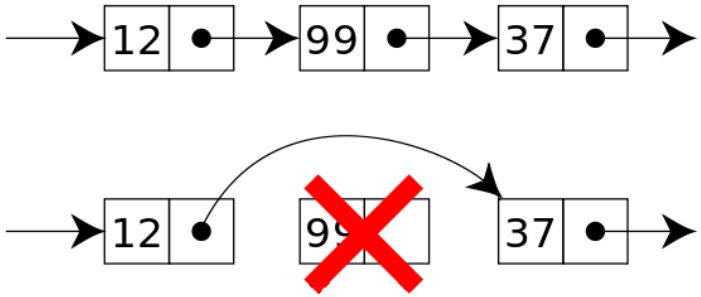
- Implementácia využitím prechádzania celého zoznamu, filtrovania prvkov podľa podmienky

Odstrániť prvok zo spájaného zoznamu (impl.)

- Operácia odstránenie prvku zo spájaného zoznamu

```
void zoznam_odstran(struct Zoznam *z, struct Prvok *prvok)
{
    struct Prvok *p = z->prvy;
    if (p == prvok)
    {
        z->prvy = p->dalsi; // odstranme prvy prvok
        free(p); return;
    }
    while (p != NULL)
    {
        struct Prvok *q = p->dalsi;
        if (q == prvok)
        {
            p->dalsi = q->dalsi; // odstranme q
            free(q); break;
        }
        p = q;
    }
}
```





Efektivita operácií so spájaním zoznamom

- Uvažujme spájaný zoznam, ktorý obsahuje N prvkov, napr. pre N = 50000
- Koľko inštrukcií programu musíme vykonať, aby sme zistili, či v ňom existuje prvok s údajom 47?
  - Efektivita operácie vyhľadania prvku s údajom

```
struct Prvok *p = zoznam_hladaj(moj, 47);
```

  - V najhoršom prípade (ak tam prvok nie je, alebo je na poslednom mieste) musíme prejsť všetkých N prvkov

```
while (p != NULL)
{
    if (p->data == cislo)
        return p;
    p = p->dalsi;
}
```

Efektivita operácií so spájaním zoznamom

- Uvažujme spájaný zoznam, ktorý obsahuje N prvkov, napr. pre N = 50000
- Koľko inštrukcií musíme vykonať, aby sme vložili ďalší prvok:
  - na začiatok?

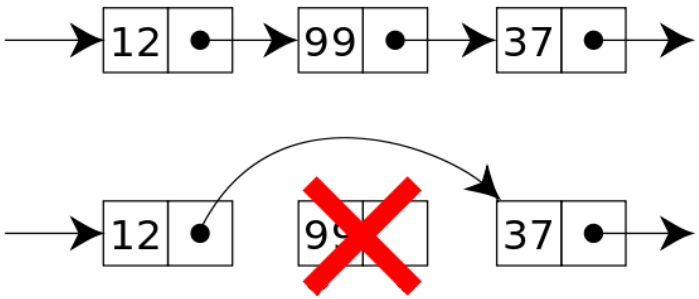
```
z->prvy = prvok_vytvor(cislo, z->prvy);
```

  - na koniec?
  - za nejaký prvok?

- Vedeli by sme vylepšiť štruktúru Zoznam-u, aby to šlo rýchlejšie?
  - Obojsmerný zoznam, pričom si navyše pamätáme smerník na posledný prvok

Efektivita operácií so spájaním zoznamom

- Uvažujme spájaný zoznam, ktorý obsahuje N prvkov, napr. pre N = 50000
- Koľko inštrukcií musíme vykonať, aby sme odstránili nejaký prvok, ktorý už máme vyhľadaný (určený)?



- Je potrebné prejsť celý zoznam, keď už poznáme príslušný Prvok, ktorý chceme odstrániť?

Rôzne implementácie spájaného zoznamu

```
typedef struct prvok
{
    struct prvok *dalsi;
    int data;
} PRVOK;

PRVOK *vloz(PRVOK *zoznam, int cislo)
{
    if (zoznam == NULL)
    {
        zoznam = (PRVOK*)malloc(sizeof(PRVOK));
        zoznam->dalsi = NULL;
        zoznam->data = cislo;
        return zoznam;
    }
    PRVOK *p = (PRVOK*)malloc(sizeof(PRVOK));
    p->dalsi = zoznam->dalsi;
    p->data = cislo;
    return p;
}
```

Prázdny spájaný zoznam je NULL (neexistuje)

Zoznam sa vytvorí až keď doňho vložím prvý prvok

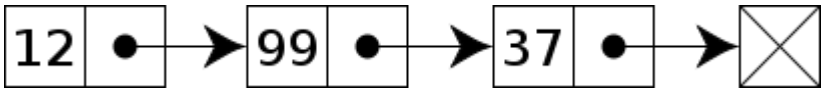
Vraciam nový zoznam: Pri každom vložení vlastne vznikne iný zoznam

```
PRVOK *zoznam = NULL;
zoznam = vloz(zoznam, 47);
```

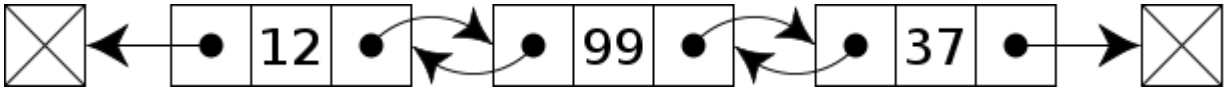
Pri vkladaní priraďujem do starej premennej zoznamu (existujúce odkazy sa porušia!)

Rôzne typy spájaného zoznamu

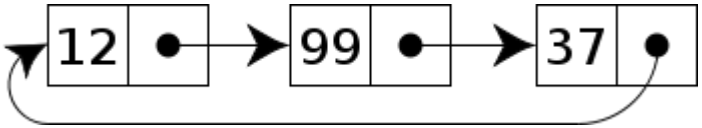
- Jednosmerne zreťazený



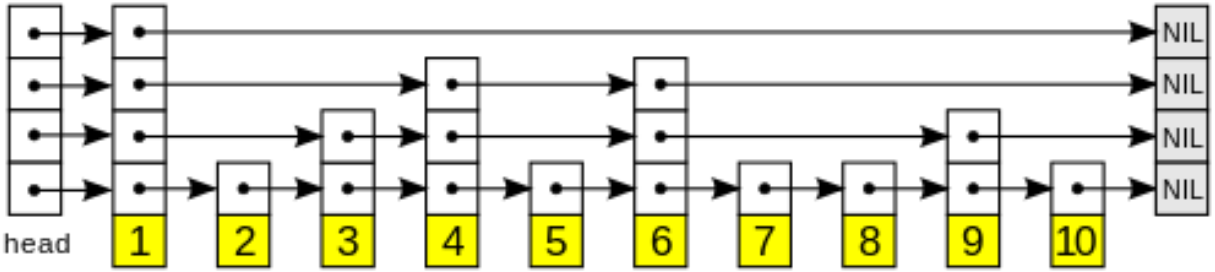
- Obojsmerne zreťazený



- Cyklický



- Preskakovací zoznam (angl. skip list)



Polia vs. Prepojené dátové štruktúry

- **Porovnanie vzhľadom na počiatočný rozsah**
- Polia musia vopred špecifikovať rozsah (veľkosť) a môžu teda spôsobovať mrhanie pamäťou
- Prepojené dátové štruktúry sú flexibilnejšie, umožňujú postupne pridávať alebo odoberať prvky, resp. údaje

Polia vs. Prepojené dátové štruktúry

- **Porovnanie vzhľadom na vnútorné usporiadanie**

- Prvky poľa sú v pamäti za sebou
  - napr. Prvok 3 nasleduje bezprostredne za Prvkom 2
  - Nie je možné prvky ich presúvať bez porušenia vzájomného usporiadania

- Prvky v prepojených dátových štruktúrach sú v pamäti nezávisle od seba a sú prepojené smerníkmi
  - Prvky je možné presúvať bez porušenia vzájomného usporiadania (ktoré je zaručené smerníkmi)

Polia vs. Prepojené dátové štruktúry

- **Porovnanie vzhľadom na rýchlosť prístupu**
- Uvažujeme, že v poli resp. prepojenej dátovej štruktúre je uložených N prvkov.
- K údajom ľubovoľného prvku v poli možno pristúpiť vykonaním konštantného počtu inštrukcií
- Pristúpiť k údajom v prvku v prepojenej dátovej štruktúre vyžaduje najskôr vyhľadanie príslušného prvku.
  - Efektívne dátové štruktúry to umožňujú vykonaním najviac log N inštrukcií (napr. vyvážené binárne vyhľadávacie stromy)
  - Najhorší prípad môže vyžadovať v menej efektívnych prepojených štruktúrach až N inštrukcií (napr. spájané zoznamy)

Polia vs. Prepojené dátové štruktúry

- **Porovnanie vzhľadom na využitie pamäte**
- Uvažujeme, že v poli resp. prepojenej dátovej štruktúre je uložených N prvkov.
- Údaje v poli sú za sebou a teda v pamäti ako jeden blok s malým „overheadom“ – réžiou operačného systému (OS)
- Údaje v poli je teda aj možné zvyčajne načítať využitím malého počtu prístupov do fyzickej pamäte.
- Údaje v prepojenej dátovej štruktúre môžu byť „roztrúsené“ v pamäti, každý údaj vo svojom bloku pamäti, čo spôsobuje veľký overhead (réžiu OS), a tiež zvyčajne je potrebné vykonať väčší počet prístupov do fyzickej pamäte.

Načo sú prepojené dátové štruktúry?

- Umožňujú nám:
  - efektívne spracúvať vopred neznáme rozsahy údajov,
  - flexibilne s údajmi pracovať (pridávať, odoberať),
  - za cenu určitej pamäťovej „roztrieštenosti“ (fragmentácie)

Abstraktná dátová štruktúra

- Spájaný zoznam ako operácie, ktoré sú jasne definované
  - Vytvoriť prázdny zoznam
  - Vložiť údaj/prvok
  - Odstrániť údaj/prvok
  - Vyhľadať prvok
  - Zmazať/uvoľniť zoznam

- V programe ho používame v zmysle týchto operácií, bez ohľadu na jeho vnútornú reprezentáciu

Spájaný zoznam implementovaný poľom

- Obmedzená veľkosť
  - Kapacita – celková maximálna veľkosť poľa
  - Počet – aktuálny počet prvkov (Počet <= Kapacita)
- Vloženie
  - Ak Počet < Kapacita, pridáme nakoniec a hotovo.
  - Ak Počet = Kapacita, musíme zväčšiť (najlepšie zväčšovať po väčších kusoch, napr. o 100, alebo zdvojnásobiť)
- Vyhľadanie prvku podľa údaju (tzv. kľúča)
  - Prejdeme všetky prvky poľa
- Odstránenie prvku
  - Prvok odstránime tak, že všetky údaje ktoré nasledujú za týmto prvkom posunieme o jedno miesto doľava.

Ďalšie prepojené dátové štruktúry

- Zásobník (angl. stack)
  - Polož prvok na zásobník (angl. push)
  - Vyber prvok zo zásobníka (angl. pop)
  - Obmedzený prístup k údajom (len na vrchný prvok)
- Implementácia?
  - **Využitím spájaného zoznamu** (ak nevieme ohraničiť množstvo, ktoré môže byť počas práce programu v jednom okamihu v zásobníku)
  - **Využitím poľa** (ak vieme ohraničiť množstvo, ktoré môže byť najviac v jednom okamihu v zásobníku)

Zásobnik (stack) implementovaný poľom

```c
#define MAX 1000

struct Zasobnik
{
    int pole[MAX+1];
    int head;
};

void push(struct Zasobnik *z, int cislo)
{
    z->pole[z->head++] = cislo;
}

int pop(struct Zasobnik *z)
{
    return z->pole[--z->head];
}

int is_empty(struct Zasobnik *z)
{
    return z->head == 0;
}
```

Ďalšie prepojené dátové štruktúry

- Rad, fronta (angl. queue)
  - Vlož prvok do radu na začiatok (angl. enqueue)
  - Vyber prvok z konca radu (angl. dequeue)
  - Obmedzený prístup (len k najskôr vloženému)

- Prístup k spracovaniu:
  - FIFO (first-in, first-out) – rad
  - LIFO (last-in, first-out) – zásobník

Rad (queue) implementovaný poľom

- Obmedzená veľkosť
  - implementácia cyklickým poľom je celkom efektívna

```
#define MAX 1000

struct Rad
{
    int pole[MAX+1];
    int head, tail;
};

void enqueue(struct Rad *r, int cislo)
{
    r->pole[r->tail++] = cislo;
    if (r->tail >= MAX)
        r->tail = 0;
}

int dequeue(struct Rad *r)
{
    int t = r->pole[r->head++];
    if (r->head >= MAX)
        r->head = 0;
}
```

STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Základy procedurálneho programovania 2

Projekt 2: Tezeus a Minotaurus

27.3.2017

letný semester 2016/2017

Projekt 2: Tezeus a Minotaurus

- Vstup:
  - Mapa bludiska s určenými význačnými bodmi
- Tezeus:
  - Nájsť susedné význačné body (nájsť medzi nimi cestu)
  - Kreslenie obrázku (mapy a prechodu)
- Minotaurus:
  - Analýza možností prechodu bludiska
  - Identifikovať kde môže Minotaurus prekvapiť Tezeusa pri prechode do stredu bludiska
  - Vyznačiť smery ako môže Minotaurus zaútočiť ...

Nabudúce...

dátum	prednáška	8:00	9:00	cvičenie	obsah
20.3.	6		Čítanie kódu, Hľadanie chýb v kóde	6	Projekt 1: odovzdanie
27.3.	7	Test 3	Riešenie testu 3, Spájané zoznamy	7	
3.4.	8		Tezeus, Bitové operácie	8	Projekt 2 **Tezeus a Minotaurus**
10.4.	9	Test 4	Riešenie testu 4, Rekurzia, Minotaurus	9	
17.4.	Veľká noc			X	
24.4.	10		Ďalšie prvky jazyka C	10	Projekt 2: odovzdanie, konzultovanie
1.5.	Sviatok			11	
8.5.	Sviatok			12	
9.5.	11		Opakovanie	X	
15.5.	12	Predtermín?		X	