PRP

3. Imperativní programování, programovací jazyk C, abstraktní datové typy a spojové struktury

Imperativní programování

Imperativní programování je jedno z programovacích paradigmat, neboli způsobů, jak jsou v programovacím jazyku formulována řešení problémů. Imperativní programování popisuje výpočet pomocí posloupnosti příkazů a určuje přesný postup (algoritmus), jak danou úlohu řešit. Program je sadou proměnných, jež v závislosti na vyhodnocení podmínek mění pomocí příkazů svůj stav. Základní metodou imperativního programování je procedurální programování, tyto termíny bývají proto často zaměňovány.

Procedurální programování

Je to poddruh imperativního programování ale tyto názvy se často zaměňují. V procedurálním programování se program skládá z vícero procedur. Je to modernější přístup (vesměs veškeré současné programy jsou procedurální), protože umožňuje přehlednější a čitelnější kód. Druhým podtypem imperativního programování je objektově orientované programování.
Pro procedurální programování jsou typické `for`, `while` a `if` pro imperativní pak spíše `goto`.

Procedural	Object-oriented
procedure	method
record	object
module	class
procedure call	message

Programovací jazyk C

C je nízko úrovňový, kompilovaný. Je dostatečně mocný na většinu systémového programování (ovladače a jádro OS), přičemž zbytek lze dořešit tzv. inline assembly, tedy metodou zápisu assembly přímo do kódu. Zdrojový kód C je přitom mnohem čitelnější než assembly, je jednodušší ho zapsat a navíc je daleko snáze přenositelný na jiné procesory a počítačové architektury. Proto jsou často operační systémy, překladače, knihovny a interprety vysokoúrovňových jazyků implementovány právě v C.
Ukládání dat je v C řešeno třemi základními způsoby: statickou alokací paměti (při překladu), automatickou alokací paměti na zásobníku a dynamickou alokací na haldě (heap) pomocí knihovních funkcí. Jazyk disponuje jen minimální abstrakcí nad alokací: s pamětí se pracuje přes ukazatel, který drží odkaz na paměťový prostor daného typu proměnné, ale je na něm možné provádět aritmetické operace (tyto operace ale neoperují s ukazateli přímo na úrovni jednotlivých bajtů, nýbrž přihlíží k velikosti datového typu, na který ukazují – existují ale také ukazatele typu `void *`, které mohou odkazovat na jakýkoliv typ dat uložený v paměti.).

Řídící struktury

- for, while, if, posloupnost příkazů, switch

Výrazy

- Přepis ve kterém jsou operandy vzájemně propojeny pomocí operátorů a slouží k výpočtu hodnoty. Operandem může být proměnná, odkaz na funkci, prvek pole nebo konstanta

Funkce

- Slouží pro čitelnost a ucelenost kódu, aby se neopakoval
- Může mít vstupní parametry, může mít výstup
- Proměnné definované uvnitř funkce jsou lokální a po dokončení funkce jsou zničeny

Nedefinované chování

- Výsledek chování programu je dopředu neurčitelný
- Dělení 0, editování charu literárně definovaného stringu, null reference exception, porovnávání pointerů proměnných, funkce by měla vracet proměnnou ale nic nevrací

Co C nemá

- Automatickou správu paměti

    - Garbage collector
- Velikost proměnných nezávislou na platformě
- Způsob uložení proměnných nezávislý na OS (Little-endian, Big endian,...)

Co je v C navíc

- Preprocessor
    - Vkládání hlavičkových souborů (header file) do zdrojového kódu
    - Podmíněný překlad
    - Makra
    - #pragma – doplňující příkazy závislé na platformě
- Linker – spojování přeložených modulů a knihoven do spustitelného kódu
- **Ukazatele (pointer) jako prostředek nepřímého adresování proměnných**
- struct a bitová pole (strukturované proměnné z různých prvků)
- union – překrytí proměnných různého typu (sdílení společné paměti)
- typedef – zavedení nových typů pomocí již známých typů
- sizeof – určení velikosti proměnné (i strukturovaného typu)
- Jiné názvy i parametry funkcí ze standardních knihoven (práce se soubory, znaky, řetězci, matematické funkce, ….)
- příkaz goto navesti;

Program C obsahuje

- Příkazy preprocesoru (Preprocessor commands)
- Definice typů (type definitions)
- Prototypy funkcí (function prototypes) kde je uvedena deklarace:
    - Jména funkce
    - Vstupních parametrů
    - Návratové hodnoty funkce
- Proměnné (variables)
- Funkce (functions) (procedura v C je funkce bez návratové hodnoty nebo void)
- Komentáře: //, /* */, nesmí být vnořené

Uvnitř funkce nelze definovat lokální funkce (definice funkci nesmí být vnořené).

Coding Style

- čitelný, udržitelný, snadno pochopitelný
- konzistentní názvy, proměnné mají popisné názvy, pouze ASCII, 120 znaků maximum, sjednocené odsazení
- složitější části kódu mají komentáře
- makra pro netriviální numerické laterály
- rozdělen do krátkých a jednoduchých funkcí, nízké zanoření
- ošetření input a output funkcí, přehledné ukončení cyklů
- proměnné vytvořeny co nejblíže použití, životnost proměnné je co nejmenší
- alokovaná paměť a práce se soubory je kontrolována

C kompilace

1. **Preprocesor**:
    1. Čte zdrojový kód v C
    2. Odstraní komentáře
    3. Upraví zdrojový text podle direktiv preprocesoru (řádky začínající #)
        1. Vloží do textu obsah jiného souboru #include …
        2. Odebere text vymezený direktivami podmíněného překladu
        3. Expanduje makra
2. **Překladač C**:
    1. Čte výstup z preprocesoru
    2. Kontroluje syntaktickou správnost textu
    3. Hlásí chyby a varování
    4. Generuje text v assembly (když nejsou chyby)
3. **Assembler**:

    1. Čte výstup z překladače C
    2. Generuje relokovatelný object kód (kód s nevyřešenými odkazy mezi moduly)
    3. Přeloží případné moduly zapsané přímo v assembly (mix programovacích jazyků)
4. **Linker** (spojovací program):
    1. Čte object kód všech zúčastněných modulů programu
    2. Připojí knihovní object moduly (přeložené dříve nebo dodané)
    3. Vyřeší odkazy mezi moduly
    4. Generuje spustitelný kód (zjednodušené)

Celočíselné typy

- Rozsahy celočíselných typů v C nejsou dány normou, ale implementací (pro 16ti a 64 bitové prostředí jsou jiné než je uvedeno v následující tabulce - ta je pro 32 bitové prostředí)
- `limits.h`, `float.h`
- Norma pouze garantuje
    - short <= int <= long
    - unsigned short <= unsigned <= unsigned long
- Celočíselné literály (zápisy čísel, viz. APO):
    - dekadický 123 456789
    - hexadecimální 0x12 0xFFFF (začíná 0x nebo 0X)
    - oktalový 0123 0567 (začíná 0)
    - unsigned 123456U (přípona U nebo u)
    - long 123456L (přípona L nebo l)
    - unsigned long 123456UL (přípona UL nebu ul)
- Není-li uvedena přípona, jde o literál typu int
- C - racionální čísla (neceločíselné datové typy):
    - Velikost reálných čísel určená implementací
    - Většina překladačů se řídí standardem IEEE-754-1985 - 1 bit znaménko, 8 bitů exponent a 23 bitů mantisa
    - Je zaručeno, že velikost double = float * 2
- C - typ void: void značí prázdnou hodnotu nebo proměnnou bez typu (jen ukazatelé)
    - void funkce1 (….) - funkce bez návratové hodnoty (procedura)
    - int funkce2 (void) - funkce bez vstupních parametrů
    - void *ptr; - ukazatel bez určeného typu (viz dále)
- #define konstanta
    - #define CERVENA 0
    - je to makro bez parametrů, každé #define musí být na samostatné řádku)
    - Preprocesor provede textovou náhradu všech výskytů slova CERVENA znakem 0
    - může být i vnořená: #define MAX_2 MAX_1+30

Funkce

C je modulární jazyk = funkce je jeho základním stavebním blokem.

- Každý program v C obsahuje minimálně funkci main()

```
int main(int argc, char** argv) { ... }
```

- Běh programu začíná na začátku funkce `main()`
- Definice funkce obsahuje hlavicku funkce a její tělo
- C používá prototypu funkce k deklaraci informací nutných pro překladač, aby mohl správně přeložit volání funkcí i v případě, že definice funkce je umístěna dále v kódu modulu nebo je jiném modulu
- Deklarace se skládá pouze z hlavičky funkce, (odpovídá interface v Javě)
- Parametry se do funkce předávají hodnotou (call by value), parametrem může být i ukazatel (pointer). Ten pak dovolí předávat parametry i odkazem.
- C nepovoluje funkce vnořené do jiných funkcí (lokální funkce ve funkci)
- Jména funkcí jsou implicitně extern, a mohou se exportovat do ostatních modulů (samostatně překládaných souborů)
- Specifikátor **static** před jménem funkce omezí viditelnost jejího jména pouze na daný modul (lokální funkce modulu)
- Formální parametry funkce jsou lokální proměnné inicializované skutečnými parametry při volání funkce
- C dovoluje rekurzi, lokální proměnné jsou pro každé jednotlivé volání zakládány znovu (v zásobníku). Kód funkce v C reentrantní (reentrant = Reentrantní provádění bloků znamená, že je možné provádět několikanásobně volaný blok paralelně.).

- Funkce nemusí mít žádné vstupní parametry, zapisuje se funkceX(void) nebo funkceX()
- Funkce nemusí vracet žádnou funkční hodnotu, pak je návratový typ void
- Pokud v definici parametrů funkce je klíčové slovo **const** - tento parametr (předaný odkazem, např. pole) nelze uvnitř funkce měnit
  - př. `int fce (const char *src) { ... }` , z pole lze pouze číst, jeho prvky nelze uvnitř funkce měnit

Vstup a Výstup a interakce s OS

- `int main(int argc, char** argv) { ... }`
- program v C je proces, lze použít pipe, fork viz OSY
- argc - počet argumentů, argv - argumnety
- lze použít pro parametry programu

Stdin

Načtení hodnoty ze stdin: `scanf("%d", &x);`

- do funkce scanf vstupuje jako parametr adresa (resp. reference) paměťového místa, kam se má načtená hodnota uložit
- jde o předání parametru odkazem, jde tedy o parametr, který může být využit pro vstup i výstup hodnoty
- funkce v C může takto „vracet “ více hodnot

Vypsání hodnoty do stdout

- `printf("Delka %d %c", 5, 'cm')` - ukázka výstupu na stdout

Zapsání/čtení hodnoty do souboru

- `fopen(const char *filename, const char *mode)` - otevře nebo vytvoří soubor podle *filename* v daném módu *mode* a vrátí FILE descriptor
- `fgets(char *str, int n, FILE *stream)` - čtení *n* znaků do *str* ze FILE descriptoru *stream*
- `fscanf(FILE *stream, const char *format, ...)` - čte z FILE descriptoru *stream* znaky podle *format* do odkazů na proměnné v ...
- `fputs(const char *str, FILE *stream)` - zápis do FILE descriptoru *stream* textu v *str*
- `fprintf(FILE *stream, const char *format, ...)` - zápis do FILE descriptoru *stream* textu podle formátu *format*, který používá proměnné v ...
- `fseek(...)` - posune pointer v otevřeném souboru na danou polohu v něm
- fclose() !!!

Ošetření chybových stavů

- kontrola návratových hodnot čtecích funkcí fscanf(), fgets()...
- kontrola správnosti a použitelnosti daných argumentů ve funkcích

Bezpečné načtení double v C

```
#include <stdio.h>
#include <stdlib.h>
int nextDouble(double *cislo){
  // === Bezpecne pro libovolny zadany pocet znaku ===
  // Navratova hodnota:
  // TRUE - zadano realne cislo
  // FALSE - neplatny vstup
  enum boolean {FALSE,TRUE}; // v ANSI C99 uz existuje true a false, zde výčty
  const int BUF_SIZE = 80;
  char vstup[BUF_SIZE],smeti[BUF_SIZE];
  // fgets precte az sizeof(vstup)-1 znaků ze stdin (standardní vstup) a pushne je do vstup
  fgets(vstup,sizeof(vstup),stdin);
  // sscanf je jako scanf, ale cteno z bufferu
  if(sscanf(vstup,"%lf%[^\n]",cislo,smeti) != 1)
    return(FALSE); // Input error
  return(TRUE);
}
```

Dynamická správa paměti - Alokace paměti

Pole lze alokovat normálně staticky, musí se ale předem zadat jeho velikost.
Způsob dynamické alokace:

`(int*)malloc(count*sizeof(int)) //zde je typ proměnné int*, ne int !`
Komplexnější příklad použití:

```
int* ctiPole1 (int *delka, int max_delka){
  // Navratovou hodnotou funkce je ukazatel na pridelene pole
  int i, *p;
  printf(" Zadejte pocet cisel = ");
  // funkce nextInt je temer totozna, jako funkce nextDouble predstavena vyse
  if (!nextInt(delka)) {
    printf("\n Chyba - Zadany udaj neni cele cislo\n\n");
    exit(EXIT_FAILURE);
  }
  if(*delka < 1 || *delka > max_delka){
    printf("\n Chyba - pocet cisel = <1,%d> \n\n",MAX_DELKA);
    exit(EXIT_FAILURE);
  }
  // Alokace pameti (prideleni pameti z "heapu")
  if((p=(int*)malloc((*delka)*sizeof(int))) == NULL){
    printf("\n Chyba - neni dostatek volne pameti \n\n");
    exit(EXIT_FAILURE);
  }
  printf("\n Zadejte cela cisla (kazde ukoncit ENTER)\n\n");
  for (i = 0; i < *delka; i++) {
    if (!nextInt(p+i)) {
      printf("\n Chyba - Zadany udaj neni cele cislo\n\n");
      exit(EXIT_FAILURE);
    }
  }
  return(p); // p - ukazatel na pridelene a naplnene pole
}
```

Dealokace probíhá pomocí funkce `free`, předává se jí ukazatel na začátek alokované paměti:
`void free(void *ptr)`

Paměťové třída

- Každá proměnná má přiřazenou paměťovou třídu
  - Statická - globální nebo statické proměnné, alokováno při startu programu a není do konce programu uvolněn
  - Automatická - lokální proměnné ve funkcích, probíhá automaticky, alokováno na zásobník, uvolněno s koncem platnosti prom.
  - Dynamická alokace - v stdlib.h a malloc.h, malloc(), free(), alokováno na haldě
- Použít lze klíčová slova
  - auto - automatická třída
  - register - doporučuje překladači uložit proměnou do registrů procesoru, nemusí vyhovět
  - static - existuje v datovém bloku paměti, uvnitř bloku {...} proměnná pak existuje i po něm, vně bloku viditelnost omezena na modul
  - extern - rozšiřuje viditelnost static proměnných na celý program, datová oblast

Operátory

Výraz může být operandem, výraz má typ a hodnotu (x void hodnotu nemá). Priority operátorů:

- Ve výrazu: funkce1() + funkce2(), není definováno, která funkce se provede jako první.
- Zkrácené vyhodnocování logických operátorů
- Operandy musí být stejného aritmetického typu, nebo oba struct nebo union stejného typu, nebo oba pointery stejného typu (pravý může být NULL)

Ukazatelé - Pointery

- Motivace
  - Předávání parametru odkazem
  - Práce s poli, řízení průchodem polem
  - Pointer na funkci
  - Využívání pole funkcí

    - Vytváření seznamových struktur
    - Hashování
- Ukazatel v C je přímo implementován pamětí procesoru a je možné přímo adresovat, včetně požadavku na registry
    - Mocný nástroj pro implementaci strojově orientovaných aplikací
- Ukazatel (pointer) je proměnná jejíž hodnotou je „ukazatel“na jinou proměnnou (analogie nepřímé adresy ve strojovém kódu či v assembly)
- Ukazatel má též typ proměnné na kterou muže ukazovat
    - ukazatel na char, int, ...
    - „ukazatel na pole“
    - ukazatel na funkci
    - ukazatel na ukazatel
    - atd.
- Ukazatel muže být též bez typu (void), pak muže obsahovat adresu libovolné proměnné. Její velikost pak nelze z vlastností ukazatele určit
- Neplatná adresa, ale definovaná v ukazateli má hodnotu konstanty NULL (kterémukoliv pointeru lze přiřadit hodnotu NULL)
- C za běhu programu nekontroluje zda adresa v ukazateli je platná
- Specialitou C je pointer na funkci!
    - ukazatel umožní, aby funkce byla parametrem funkce
- Pomocí ukazatele lze předávat parametry funkci odkazem (call by reference) – základní využití
- Adresa proměnné se zjistí adresovým operátorem & (ampersand), tzv. referenční operátor
- K proměnné na kterou ukazatel ukazuje se přistoupí operátorem nepřímé adresy * (hvězdička)

Ukázka pro lepší pochopení

```
int x=30; // promenná typu int, &x - adresa promenné x
int *px; // *px promenná typu int, px promenná typu pointer na int
px=&x; // do promenné typu pointer na int se uloží adresa promenné x
printf(" %d " " %d \n", x, px); // 30 2280564
printf(" %d " " %d \n", &x, *px); // 2280564 30
printf(" %d " " %d \n", *(&x), &(*px)); // 30 2280564
```

Operace s pointery

- Povolené aritmetické operace s ukazateli:
    - pointer + integer
    - pointer - integer
    - pointer1 - pointer2 (musí být stejného typu)
- Povolené operandy relace:
    - dva ukazatele (pointers) shodného typu nebo jeden z nich NULL nebo typu void
- Jednoduché přiřazení - povolené operandy
    - dva operandy typu - pointer (stejného typu) nebo pravý operand=NULL nebo jeden pointer typu void
- Aritmetické operace jsou užitečné když ukazatel ukazuje na pole

Pole

- Pole je množina prvku (proměnných) stejného typu
- Index prvního prvku je vždy roven 0
- Pole se funkcím předává odkazem: `void funkcePole (int p[] )`
- Prvky pole mohou být proměnné libovolného typu (i strukturované)
- Definice pole určuje:
    - jméno pole
    - typ prvku pole
    - počet prvků pole
- Prvky pole je možné inicializovat
- Počet prvku statického pole musí být znám v době překladu
- Prvky pole zabírají v paměti souvislou oblast!
- Velikost pole (byte) = počet prvku pole * sizeof(prvek pole)
- VLA (variable length array) - délka je determinována za runtime, nelze pro statické pole

String

- C - nemá proměnnou typu String, nahrazuje se jednorozměrným polem z prvku typu char. Poslední prvek takového pole je vždy ‘\0’ (null char)
  - lze také zapsat `char \*c; c = “ja jsem string”; // c je ukazatel na prvni prvek pole`

Hlavičkové soubory

- Důvody:
  - Deklarace funkčního prototypy před použitím
  - Prostředek pro zpřehlednění struktury programu
  - Ukrytí definice funkce, možnost vytváření knihoven
    - Předání souboru .h + .obj
    - Vlastní definice funkce v souborech s relativním kódem .obj
- Obsahují
  - Hlavičky funkcí (funkční prototypy)
  - deklarace funkce
  - Deklarace globálních proměnných
  - Definice datových typů
  - Definice symbolických konstant
  - Definice make
- „obdoba interface“
- Příklad soubor xxx.h:

```
/* podmineny preklad proti opakovanemu vkládání „include“ */
#ifndef XXX // čti: if not defined = proti duplicitě = podmíněný překlad
#define XXX
/* definice symb. konstant vyuzivanych i v jinych modulech */
#define CHYBA -1.0
/* definice maker s parametry */
#define je_velke(c) ((c) >= ‘A’ && (c) <= ‘Z’)
/* definice globalnich typu */
typedef struct{
  int vyska;
  int vaha;
} MIRY;
/* deklarace globalnich promennych modulu xxx.c */
extern MIRY m; // v jiném modulu bude definice MIRY m;
/* uplne funkcni prototypy globalnich funkci modulu xxx.c */
extern double vstup_dat(void);
extern void vystup_dat(double obsah);
#endif
- Příklad soubor xxx.c (includuje xxx.h)
#include <stdio.h> /* standardni vklad*/
#include „xxx.h" /* natazeni konstant,prototypu funkci a globalnich typu vlastniho modulu */
/* deklarace globalnich promennych */
extern int z; /*ktere nebyly definovány v hlavičkovém soubor */
/* definice globalnich promennych */
int y;/* které nejsou definovány v hlavičkovém soubor */
/* lokalni definice symbolickych konstant a maker */
#define kontrola(x) ( ((x) >= 0.0) ? (x) : CHYBA_DAT )
/* lokalni definice novych typu */
typedef struct{} OSOBA;
/* definice statickych globalnich promennych */
static MIRY m;
/* uplne funkcni prototypy lokalnich funkci */
int nextDouble(double *cislo);
/* funkce main() */
int main(int argc, char** argv){}
/* definice globalnich funkci - to, ze je glob., bylo definovano v xxx.h */
double vstup_dat(void){ …return ();}
/*funkční prototypy v.h souboru*/
void vystup_dat(double obsah){ … }
```

```
/* definice lokalnich funkci */
int nextDouble(double *cislo){… }
```

Typedef

Umožňuje vytvářet nové datové typy:
typedef double *PF;
typedef int CELE;
PF x,y;
CELE i,j;

Struktury

- Struktura je konečná množina prvků (proměnných), které nemusí být stejného typu
  - Obdoba třídy bez metod v C++
- Skladba struktury je definovaná uživatelem jako nový typ sestavený z již definovaných typů
- K prvkům struktury se přistupuje tečkovou notací
- K prvkům struktury je možné přistupovat i pomocí ukazatele na strukturu operátorem ->
- Struktury mohou být vnořené
- Pro struktury stejného typu je definována operace přiřazení struct1=struct2
- Struktury (jako celek) nelze porovnávat relačním operátorem ==
- Struktura může být do funkce předávána hodnotou i odkazem
- Struktura může být návratovou hodnotou funkce

```
typedef struct { // <==== Pomoci Typedef
  char jmeno[20]; // Prvky struktury, pole
  char adresa[50]; // - “ - pole
  int telefon; // - “ -
} Tid, *Tidp;

Tid sk1, skAvt[20]; // struktura, pole struktur
Tidp pid; // ukazatel na strukturu

sk1.jmeno=“Jan Novak”; // teckova notace
skAvt[0].jmeno=”Jan Novak”; // prvek pole
pid=&sk1; // do pid adresa struktury

pid->jmeno=“Jan Novak”; // odkaz pomoci ->
(*pid).jmeno=“Jan Novak”; // odkaz pomoci *
```

Union

- Union je množina prvků (proměnných), které nemusí být stejného typu
- Prvky unionu sdílejí společně stejná paměťová místa (překrývají se)
- Velikost unionu je dána velikostí největšího z jeho prvků
- Skladba unionu je definovaná uživatelem jako nový typ sestavený z již definovaných typů
- K prvkům unionu se přistupuje tečkovou notací
- př:

```
union Tnum{ // <==== Tnum=jmeno sablony (tag)
  long n;
  double x;
};
union Tnum nx; // nx - promenna typu union
nx.n=123456789L; // do n hodnota long
nx.x=2.1456; // do x hodnota double (prekryva n)
```

Podmíněný překlad

```
#if VERSE_CITACE == 1
do {
...
} while(TRUE);
#endif
```

Definice vs. deklarace

1. Deklarace určuje interpretaci a vlastnosti identifikátoru(ů)
2. Definice je deklarace včetně přidělení paměti (memory allocation) proměnným, konstantám nebo funkcím

Standardní knihovny

Vlastní jazyk C neobsahuje žádné prostředky pro vstup a výstup dat, složitější matematické operace, práci s řetězci, třídění, blokové přesuny dat v paměti, práci s datem a časem, komunikaci s operačním systémem,správu paměti pro dynamické přidělování, vyhodnocení běhových chyb (run-time errors) apod..Tyto a další funkce jsou však obsaženy ve standardních knihovnách (ANSI C Library) dodávaných s překladači jazyka C. Uživatel dostává k dispozici přeložený kód knihoven (který se připojuje – linkuje k uživatelovu kódu) a hlavičkové soubory (headers) s prototypy funkcí, novými typy, makry a konstantami. Hlavičkové soubory (obdoba interface v Javě) se připojují k uživatelovu kódu direktivou preprocesoru #include <…>. Je zvykem, že hlavičkové soubory mají rozšíření *,h, např. stdio.h.
Příklad:

- Vstup a výstup (formátovaný i neformátovaný) - `stdin.h`
- Rozsahy čísel jednotlivých typů - `limits.h`
- Matematické funkce - `stdlib.h`, `math.h`
- Zpracování běhových chyb (run-time errors) - `errno.h`, `assert.h`
- Klasifikace znaků (typ char) - `ctype.h`
- Práce s řetězci (string handling) - `string.h`
- Internacionalizace (adaptace pro různé jazykové mutace) - `locale.h`
- Vyhledávání a třídění - `stdlib.h`
- Blokové přenosy dat v paměti - `string.h`
- Správa paměti (Dynamic Memory Management) - `stdlib.h`
- Datum a čas - `time.h`
- Komunikace s operačním systémem - `stdlib.h`, `signal.h`
- Nelokální skok (lokální je součástí jazyka, viz goto) - `setjump.h`

Abstraktní datové typy

Množina druhů dat (hodnot) a příslušných operací, které jsou přesně specifikovány a to nezávisle na konkrétní implementaci

Počet složek:

- neměnný = statický datový typ, počet položek je konstantní, pole, řetězec, třída
- proměnný = dynamický datový typ, počet složek je proměnný, mezi operace patří vložení, odebrání určitého prvku

Typ položek, dat:

- homogenní = všechny položky stejného typu
- nehomogenní = různého typu

Existence bezprostředního následníka

- lineární = existuje [např. pole, fronta, seznam,…]
- nelineární = neexistuje [strom, tabulka,…]

Můžeme definovat:
- Matematicky - axiomy a asignatura
- Rozhraní - konstruktor vracející odkaz, operace nad daty

Základní ADT jsou například:

- asociativní pole (dictionary, map)
- zásobník
- seznam

- fronta
- množina
- textový řetězec
- strom
- halda neboli prioritní fronta
- hashovací tabulka

Spojové seznamy

Lineární seznam (také lineární spojový seznam) je dynamická datová struktura, vzdáleně podobná poli (umožňuje uchovat velké množství hodnot ale jiným způsobem), obsahující jednu a více datových položek (struktur) stejného typu, které jsou navzájem lineárně provázány vzájemnými odkazy pomocí ukazatelů nebo referencí. Aby byl seznam lineární, nesmí existovat cykly ve vzájemných odkazech.
Lineární seznamy mohou existovat jednosměrné a obousměrné. V jednosměrném seznamu odkazuje každá položka na položku následující a v obousměrném seznamu odkazuje položka na následující i předcházející položky. Zavádí se také ukazatel nebo reference na aktuální (vybraný) prvek seznamu. Na konci (a začátku) seznamu musí být definována zarážka označující konec seznamu. Pokud vytvoříme cyklus tak, že konec seznamu navážeme na jeho počátek, jedná se o kruhový seznam.

Základním prvkem spojových struktur je dvojice:

- reference - odkaz(y) na další prvek spojové struktury
- hodnota – informace libovolného typu

Nejjednodušší typ spojové struktury – jednosměrné spojové seznamy.
Spojové struktury jsou mocným implementačním prostředkem pro ADT, viz další přednáška o ADT a předmět Algoritmizace:

1. – Fronty - FIFO, pomocí struktur, uchování ukazatele na začátek (možná konec pro rychlé vkládání), každý prvek zná následovníka
2. – Zásobníky - LIFO, pomocí struktur, uchování ukazatele na konec, každý prvek zná předchůdce, push, pop, isEmpty
3. – Stromy
4. – Grafy

Stromy

V informatice je strom široce využívanou datovou strukturou, která představuje stromovou strukturu s propojenými uzly.

Pojmy:

1. „Cesta“ k nějakému uzlu je definována jako posloupnost všech uzlů od kořene k uzlu.
2. „Délka cesty“ je rovna počtu hran, které cesta obsahuje, tedy počtu uzlů posloupnosti – 1 (mínus jedna).
3. „Hloubka uzlu“ je definována jako délka cesty od kořene k uzlu. Prvky se stejnou hloubkou jsou na „téže úrovni“.
4. „Výška stromu“ je rovna hodnotě maximální hloubky uzlu, se označuje též za „hloubku stromu“.
5. „Šířkou stromu“ je počet uzlů na stejné úrovni.
6. Strom má „nejmenší výšku“ právě tehdy, když na všech úrovních (s možnou výjimkou té poslední) má tato struktura plný počet uzlů. Úroveň všech listů je stejná nebo se liší maximálně o 1.

Uspořádanost
„Uspořádaný“ nebo také „seřazený strom“ je takový strom, ve kterém jsou všichni přímí potomci každého uzlu seřazeni. Tudíž, pokud uzel má n dětí, lze určit prvního přímého potomka, druhého přímého potomka, až n-tého přímého potomka. U „neuspořádaného stromu“ se jedná o strom v čistě strukturálním smyslu. To znamená, že pro daný uzel nejsou uspořádáni potomci.

Vyvážený strom
„Vyvážený strom“ je takový strom, který má uzly rovnoměrně rozložené, tedy má nejmenší výšku. Ideální situace je taková, kdy má strom v každé hladině, kromě poslední, maximální počet uzlů, a v poslední hladině má uzly co nejvíce vlevo.

Procházení stromu (podrobněji v ALG)

1. **Do šířky BFS** - Procházením „stromu do šířky“ (anglicky „level-order“) se rozumí procházení stromem po vrstvách úrovní (tzn. po hladinách).
2. **Do hloubky DFS** - Procházení začíná v kořeni stromu a postupuje se vždy na potomky daného vrcholu. Procházení končí, když v žádné větvi (tj. v žádném podstromu) již není následník. Podle pořadí, ve kterém se prochází uzly uspořádaného stromu, se rozlišují tři základní metody:
   a. PREORDER
   1. proveď akci
   2. projdi levý podstrom
   3. projdi pravý podstrom
   b. INORDER
   1. projdi levý podstrom
   2. proveď akci

3. projdi pravý podstrom
c. POSTORDER
1. projdi levý podstrom
2. projdi pravý podstrom
3. proveď akci

Binární stromy

Každý prvek (uzel) má nanejvýš dva následníky (node). Binární strom obsahuje uzly, které mají nejvíce dva potomky.

Důležité pojmy:

1. kořen stromu - nejvyšší prvek, nemá rodiče
2. levý podstrom - strom levých potomků
3. pravý podstrom - pravých
4. vnitřní uzel - prvek, který má potomky (často se sem nepočítá kořen, někdy ano)
5. list - prvek, který již nemá potomky

U binárního stromu rozlišujeme další pojmy:

- Plný binární strom - všechny jeho listy jsou ve stejné hloubce.
- Úplný binární strom - každý vnitřní uzel má dva potomky.
- Vyvážený binární strom - hloubka listů se od sebe liší maximálně o jedna.

Trochu matiky

- h – hloubka stromu,
- n – počet uzlů
- n0 – počet listů
- n2 – počet vnitřních uzlů

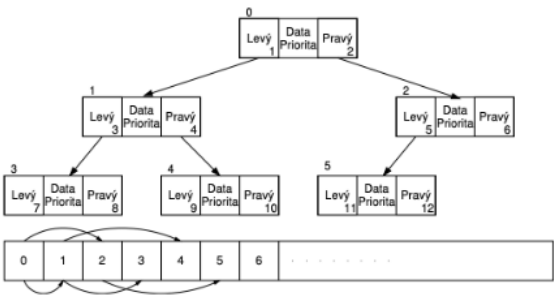
Úplný binární strom:
minimální počet uzlů: $n = 2h + 1$
maximální počet uzlů: $n = 2^{(h+1)} - 1$
počet listů: $\lceil n/2 \rceil$

Prioritní fronta

- spojový seznam, kde každý prvek má svoji prioritu a při vložení je podle toho seřazen
- pomocí pole - stejně jako haldu
- u hledání nejkratší cesty v grafu je priorita je určena cenou kroku ze startu hledání do aktuálního uzlu

Halda

- Halda je dynamická datová struktura, která má reprezentaci binárního stromu a uspořádání prioritní fronty
- Vlastnost haldy - zajišťují, že kořen je vždy prvek s nejmenší hodnotou (prioritou)
  - Hodnota každého prvku je menší než hodnota libovolného potomka
  - Binární plný strom
  - Odebírání jen přes kořen
- Pomocí pole - první prvek je kořen, levý následník 2i +1, pravý 2i + 2



Kruhový buffer (pole)

- Pole fixní délky, 2 ukazatele - první obsazený prvek, první volný prvek, idx modulo velikost