14. Vyšší programovací jazyky (Java, C, C# apod.). Struktura programu, implementace příkazů a datových typů. Práce se soubory, operace vstupu a výstupu. Algoritmizace základních úloh, třídění, vyhledávání, porovnání algoritmů.

14.1. Vyšší programovací jazyky (Java, C, C# apod.)

Vyšší programovací jazyk (též vysokoúrovňový jazyk, problémově orientovaný jazyk) je v informatice označení pro programovací jazyk s větší mírou abstrakce. Vyšší abstrakcí je míněno přiblížení zápisu zdrojového kódu programu v daném programovacím jazyce k tomu, jak problémy zpracovává svým myšlením člověk. Nižší programovací jazyk se naopak svým zápisem přibližuje tomu, jak po technické stránce pracuje počítač (resp. jeho procesor).

Nevýhodou vyšších programovacích jazyků je fakt, že počítače umí přímo zpracovávat kód zapsaný v nejnižších programovacích jazycích (tzv. Jazyk symbolických adres). Proto musejí být programy zapsané ve vyšších programovacích jazycích překládány překladačem (kompilátorem) do nižších jazyků.

Rozdělují se na:

- Procedurální (Přesný algoritmus, jak úlohu řešit)
    - Strukturované (Funkce které se spojují v jeden celek)
    - Objektově orientované (Přiblížení reálnému světu, objekty, abstrakce)
- Neprocedurální (Programování definic co se má dělat, ne jak)
    - Funkcionální (Vyhodnocování matematických funkcí)
    - Logický (Matematická logika, důkazy)

14.2. Struktura programu

Řídicí struktury jsou používány ve vyšších programovacích a skriptovacích jazycích. Rozhodují o dalším provádění programu: větví jeho běh, vytváří cykly nebo jinak mění běh programu.

Typy řídících struktur:

- posloupnost příkazů – všechny příkazy se postupně provedou jeden po druhém
- větvení (či podmíněný příkaz) – v závislosti na splnění podmínky se určitý příkaz buď provede nebo neprovede
- cyklus – v závislosti na splnění podmínky se část programu vykoná vícekrát

14.2.1. Větvení

Pascal:

```
if a > 0 then begin
 writeln("ano")
end else begin
 writeln("ne")
end
```

C:

```
if (a > 0) {
 printf("ano");
} else {
 printf("ne");
}
```

Shell skript:

```
if [ $a -gt 0 ]
then
 echo "ano"
else
 echo "ne"
fi
```

Python:

```
if a > 0:
 print "ano"
else:
 print "ne"
```

PHP:

```
if ( $a > 0 ){
 echo "ano";
}
else{
 echo "ne";
}
```

Pascal:

```
case nejakyZnak of
 'a': akceNaA;
 'x': akceNaX;
 'y','z':akceNaYneboZ;
 else akceVPripadeNeshody;
end;
```

C:

```
switch (nejakyZnak) {
 case 'a': akceNaA; break;
 case 'x': akceNaX; break;
 case 'y':
 case 'z': akceNaYNeboZ; break;
 default: akceVPripadeNeshody;
}
```

Shell skript:

```
case $nejakyZnak in
 a) akceNaA ;;
 x) akceNaX ;;
 [yz]) akceNaYNeboZ ;;
 *) akceVPripadeNeshody ;;
esac
```

Python:

```
if nejakyZnak == "a":
    akceNaA
elif nejakyZnak == "x":
    akceNaX
elif nejakyZnak in ["y", "z"]:
    akceNaYneboZ
else:
    akceVPripadeNeshody
```

14.2.2. Cyklus

Cyklus se skládá z posloupnosti příkazů a podmíněného skoku, pomocí kterého se cyklus ukončuje při splnění podmínky.

```
while (podmínka) {
příkaz1; příkaz2; ... příkazN;
}

do {
příkaz1; příkaz2; ... příkazN;
} while (podmínka);

for (inicializátor; podmínka; inkrement) {
příkaz1; příkaz2; ... příkazN;
}
```

14.3. Implementace příkazů a datových typů

/

14.4. Práce se soubory

Se soubory pracujeme pomocí existujících knihoven nebo vnořených funkcí, například v jazyce C:

```c
int main(void)
{
    FILE *soubor;
    char text[255];

    soubor = fopen(NAZEV, "a+"); /* soubor se otevre pro
aktualizaci,
                                    neexistujici soubor se
vytvori */
    do {
        fputs("Zadejte slovo, ktere chcete zapsat do
souboru\n"
              "a stisknete ENTER, nebo \"q\" pro ukonceni:
", stdout);
        scanf("%254s", text);
        if (!strcmp(text, "q"))
            break;
        printf("Zapisuji >> %s <<\n", text);
        fprintf(soubor, ">> %s <<\n", text);
    } while (1);

    fclose(soubor);

    return 0;
}
```

Nebo Java:

```java
try (BufferedReader br = new BufferedReader(new
FileReader("soubor.txt")))
{
        String s;
        while ((s = br.readLine()) != null)
        {
                System.out.println(s);
        }
}
catch (Exception e)
```

```
{
        System.err.println("Chyba při četení ze souboru.");
}
```

14.5. Operace vstupu a výstupu

Čtení a zápis

14.6. Algoritmizace základních úloh

/

14.7. Třídění

Řadící algoritmy slouží k setřízení jednotlivých prvků vstupního souboru (obvykle seznamu) dle jejich velikosti. Při volbě vhodného řadícího algoritmu je třeba dbát na několik kritérií - výkon algoritmu (jeho časová složitost), implementační složitost, vhodnost pro danou datovou strukturu a v neposlední řadě stabilita algoritmu.

Možné třídící algoritmy:

- Bubble sort
- Selection sort
- Quick sort
- Merge sort
- Heapsort
- Shaker sort
- Bogosort
- …

14.7.1. Bubble sort

Pokud si představíme řazená čísla jako bublinky, tak ty s menší hodnotou jsou lehčí než ty s vyšší hodnotou a stoupají proto ve vodě rychleji.

Obdobně postupuje také bubble sort. Porovnává dva sousední prvky, a pokud je nižší číslo nalevo od vyššího, tak je prohodí (nižší číslo je lehčí a rychleji stoupá ke konci pole) a se stejnou logikou pokračuje na dalším indexu. Pokud jsou čísla ve správném pořadí, tak je neprohodí – pouze postoupí dále (algoritmus tím našel lehčí bublinku). Na konci iterace se tímto způsobem na konec pole vždy dostane ta nejlehčí bublinka (nejnižší číslo). Nyní algoritmus můžeme pustit znovu na redukovaný problém (na poslední pozici pole je již to správné číslo).

Po n-1 průchodech (poslední bublinka je seřazena triviálně) je pole seřazeno.

Ukázka v pseudokodu:

```
function bubbleSort(array a)
   for i in 1 -> a.length - 1 do
      for j in 1 -> a.length - i - 1 do
         if a[j] < a[j+1]
            prohoď(a[j], a[j+1]);
```

14.8. Vyhledávání

14.8.1. Vyhledávání v poli

Lineární (sekvenční) vyhledávání je nejjednodušším způsobem, jak zjistit, jestli se v poli (nebo jiné datové struktuře) nachází námi hledaný prvek. Princip je zcela triviální, procházíme jeden prvek po druhém a zjišťujeme, jestli to není právě ten, který hledáme. Z tohoto vyplývá složitost tohoto postupu $O(n)$.

Vyhledávání v poli dělíme na:

- Lineární vyhledávání
- Binární vyhledávání
- Prořezávej a hledej
- Interpolační vyhledávání

Lineární vyhledávání

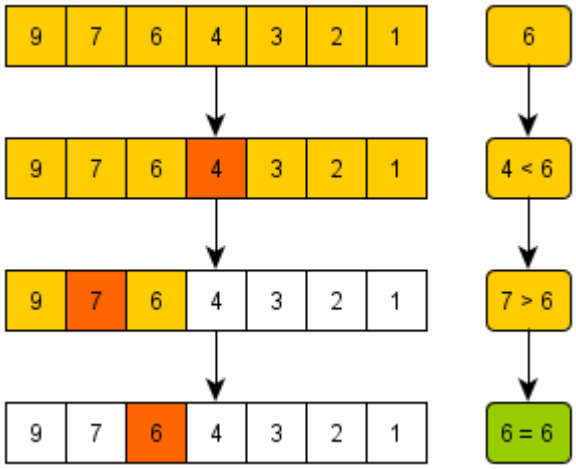
Lineární vyhledávání použijeme tehdy, pokud nemáme žádné informace o uspořádání prvků struktury nebo pokud nám datová struktura (například spojový seznam) neumožňuje efektivnější způsob vyhledávání.

Příklad Java:

```java
/**
  * Linearni vyhledavani
  * @param array pole, ve kterem hledame
  * @param value hodnota, kterou hledame
  * @return index hledaneho prvku, -1 v pripade nenalezeni
  */
 public static int linearSearch(int[] array, int value){
     for(int i = 0; i < array.length; i++){
         if(array[i] == value) return i;
     }
     return -1;
 }
```

Binární vyhledávání

Binární vyhledávání (půlení intervalu) je vyhledávací technika, která zjišťuje pozici zadaného prvku v seřazeném poli. Na rozdíl od sekvenčního vyhledávání, které vyžaduje až $O(n)$ operací, binární vyhledávání pracuje s asymptotickou složitostí $O(\log_2 n)$



Mějme sestupně seřazené pole a hledejme v něm prvek h. Binární vyhledávání v každém svém kroku zvolí prostřední prvek pole (p) a porovná jej s prvkem h. Pokud jsou tyto prvky totožné, tak vrátí index prvku p. Pokud má hledaný prvek vyšší hodnotu než p, tak je zřejmé, že h se musí nacházet v levé části pole. V opačném případě (p > h) se hledaný prvek musí nacházet v pravé části pole. Binární vyhledávání proto zavolá sebe sama na příslušnou perspektivní polovinu pole.

Protože dochází v každém kroku k půlení prohledávaného intervalu (a druhá polovina se nezpracovává), tak musí dojít k nalezení (nebo vyvrácení přítomnosti) hledaného prvku nejpozději v $\log_2 n$ krocích.

Příklad Java:

```
    /**
     * Binarni vyhledavani
     * @param array prohledavane pole (setridene od nejvyssiho)
     * @param leftIndex prvni index, na ktery smime sahnout
     * @param rightIndex posledni index, na ktery smime sahnout
     * @param value hodnota k nalezeni
     * @return index hodnoty, -1 v pripade nenalezeni
     */
    public static int binarySearch(int[]  array, int leftIndex, int rightIndex, int value){
        if(leftIndex == rightIndex && array[leftIndex] != value) return -1;

        int middleIndex = leftIndex + (rightIndex - leftIndex)/2;
        if(array[middleIndex] == value) return middleIndex;
        else if(array[middleIndex] > value)
            return binarySearch(array, middleIndex + 1, rightIndex, value);
        else return binarySearch(array, leftIndex, Math.max(leftIndex, middleIndex - 1), value);
}
```

Prořezávej a hledej

Prořezávej a hledej (Prune and search) je typ algoritmu založený na vyřazování neperspektivních dat – redukci velikosti problému. Toto paradigma je velmi podobné algoritmům typu rozděl a panuj (divide and conquer), zásadní rozdíl je ovšem v tom, že při prořezávání neprocházíme všechny větve, ale pouze ty, které pro nás dávají smysl.

Pokud hledáme n-té nejvyšší číslo v neseřazeném poli, tak by řešením zajisté bylo pole seřadit a podívat se na zadaný index. Toto řešení ovšem není příliš efektivní. Lepším řešením je upravit například Quicksort tak, aby se po rozdělení pole dle pivota prohledávala pouze ta část, která obsahuje řešení - Quicksort v každém svém kroku umístí pivota na korektní místo v seřazeném poli, není proto problém rozhodnout, ve které části se nachází ono hledané číslo.

Zatímco asymptotická složitost Quicksortu je $O(n^2)$ a očekávaná $O(n \cdot \log(n))$, tak díky eliminaci větví, které nemohou obsahovat řešení, je očekávaná složitost prune and search algoritmu $O(c \cdot n)$, kde c je malá konstanta.

Interpolační vyhledávání

Interpolační vyhledávání je vylepšením binárního vyhledávání pro případ, kdy víme, že jsou čísla v poli nejen seřazená, ale také rovnoměrně rozložená.

Interpolační vyhledávání vychází z úvahy, že pokud máme v poli například čísla od 0 do 100 a hledáme číslo 2, tak je přinejmenším nerozumné pole binárně dělit (napřed se podívat na index 50, pak 25, pak 12...), ale je daleko rozumnější se podívat někam kolem indexu 2, kde by se číslo mělo nacházet (vzhledem k rovnoměrnému rozložení). Složitost tohoto přístupu je $O(\log(\log n))$.

14.8.2. Vyhledávání v textu

Algoritmy pro vyhledávání v textu jsou důležitou třídou algoritmů pro práci s textovými řetězci. Slouží ke hledání místa, kde se jeden či více řetězců (vzorků) shoduje s částí většího textu.

Dělíme je na:

- Algoritmy používající jeden vzorek
- Algoritmy používající konečnou množinu vzorků
- Algoritmy používající nekonečně mnoho vzorků

Dále příklady pouze pro algoritmy používající jeden vzorek:

Naivní algoritmus:

Naivní algoritmus vyhledávání v textu slouží, jak již název napovídá, k vyhledání výskytu daného vzoru v textu. Algoritmus prochází jak text, tak vzor zepředu a porovnává, jestli jsou totožné (na m místech vzoru). Pokud ano, tak byl výskyt nalezen, pokud ne, tak se vzorem posune o jedno místo doprava a postup se opakuje (dokud algoritmus nenarazí na konec textu). Složitost tohoto postupu je $O(m \cdot n)$, kde m je délka vzoru a n je délka textu.

Dále existují například **Hammingova vzdálenost** a **Levenshteinova vzdálenost**.

14.9. Porovnání algoritmů

Řadící algoritmy slouží k setřízení jednotlivých prvků vstupního souboru (obvykle seznamu) dle jejich velikosti. Při volbě vhodného řadícího algoritmu je třeba dbát na několik kritérií - výkon algoritmu (jeho časová složitost), implementační složitost, vhodnost pro danou datovou strukturu a v neposlední řadě stabilita algoritmu.

14.9.1. Časová složitost

Z hlediska časové složitosti jsou nejvýkonnějšími algoritmy ty, které neporovnávají jednotlivé hodnoty prvků, ale fungují na jiném pricipu (složitost $O(n)$). Příkladem je například counting sort (který počítá výskyty jednotlivých hodnot) nebo radix sort (řadí řetězce fixní délky dle jednotlivých znaků). Tyto algoritmy ale nejsou vhodné pro všeobecné použití.

Z tohoto důvodu volíme algoritmy třídy $\mathrm{O}(n \cdot \log(n))$ – heapsort, quicksort nebo merge sort. Tyto algoritmy jsou optimální, jelikož bylo dokázáno, že algoritmus založený na bázi porovnávání hodnot musí mít složitost $\Theta(n \cdot \log(n))$.

Za předpokladu malé velikosti dat jsou vhodným řešením algoritmy se složitostí $\mathrm{O}(n^2)$ - bubble sort, insertion sort, selection sort, protože je jejich implementace velmi jednoduchá. Z kvadratických algoritmů je nejvýkonnější Shell sort - řazení se snižujícím se přírůstkem.

14.9.2. Stabilita řazení

Říkáme, že je řazení stabilní, pokud nedojde v jeho průběhu k prohození prvků se stejnou hodnotou. Tato vlastnost je užitečná, jestliže dochází k postupnému řazení podle dvou (a více) parametrů. Řadíme-li například žáky v třídní knize napřed dle křestního jména a poté dle příjmení, pak výsledek stabilního algoritmu odpovídá očekávání (první je Karel Novák, následuje Václav Novák). Pokud by algoritmus nebyl stabilní, tak tento postup nebude fungovat, protože druhé řazení by mohlo zpřeházet výsledek prvního (Václav Novák by mohl být před Karlem Novákem).

14.9.3. Přirozenost algoritmu

Další vlastností řadících algoritmů je jejich přirozenost. Přirozený algoritmus rychleji zpracuje již částečně seřazenou posloupnost, zatímco u algoritmu, který přirozený není, tento fakt nehraje žádnou roli.

14.9.4. Porovnání

Název	Poznámka	Přirozený	Stabilní	Složitost
Bubble sort		ano	ano	$O(n^2)$
Shakersort		ano	ano	$\mathrm{O}(n^2)$
Selection sort		ne	ne	$\Theta(n^2)$
Insertion sort		ano	ano	$\mathrm{O}(n^2)$
Shell sort	Nejvýkonnější kvadratický algoritmus	ano	ne	$\mathrm{O}(n^2)$
Heapsort		ne	ne	$\Theta(n \cdot \log(n))$
Merge sort		ano	ano	$\Theta(n \cdot \log(n))$
Quicksort	Očekávaná složitost - při špatné volbě pivota až $\mathrm{O}(n^2)$	ne	ne	$O(n \cdot \log(n))$
Radix sort	Konstantní délka řazených řetězců	ne	ano	$\Theta(n)$
Counting sort	Má smysl počítat výskyty hodnot	ne	ano	$\Theta(n)$