Anotace

- Třídění,
- Pointery, dynamické proměnné

Radix- alias bucketsort

- Třídíme-li celá (přirozená) čísla, jejichž velikost je omezena, můžeme využít toho, že v desítkovém zápisu je na každé pozici jen jedna číslice z deseti.
- Můžeme tedy čísla rozhodit na hromádky podle těchto číslic.
- Naivní algoritmus by zřejmě začal od začátku, vytřídil "nejmenší" a jel dále.
- To by sice fungovalo, ale bylo by to ošklivé.
- My začneme od poslední číslice.

Bucketsort

Pozor, kreativní pseudokód!

```
procedure bucketsort(pole);
begin
      for i:=0 to delka do
      begin
            rozhod cisla z pole do pole0 az pole9;
                  podle cislice na i-tem miste;
            pole:=pole0+pole1+pole2+...+pole9;
      end;
end;
```

Bucketsort – analýza

- Složitost: `delka` je konstanta, tedy konstantně-krát prolezeme pole délky n, tudíž složitost je $\Theta(n)$ (dolní odhad opět triviálně n, na každý prvek musíme "kouknout").
- Korektnost:
  - Pokud se dvě čísla liší, liší se na nějaké pozici.
  - Uvažme nejvyšší řád, na kterém se liší. Na tomto řádu má menší číslo menší číslici a tudíž:
  - při rozhazování podle tohoto řádu se menší číslo dostane před větší.
  - Dále jdou čísla do stejné přihrádky a tudíž se jejich pořadí nemění.
- Co je s naším dolním odhadem $\Omega(n \log n)$?
- Nemá splněné předpoklady, bucketsort čísla vůbec neporovnává.

Paměti

- Počítače mají několik typů pamětí:
- Operační (zpravidla RAM),
- persistentní (disky, diskety, magnetofonové pásky, děrné štítky).
- Prvním občas říkáme vnitřní, druhým vnější.
- Vnitřní paměť je rychlá, kdežto vnější paměť je velká.
- Ve vnitřní paměti můžeme "dovádět", ve vnější paměti hledání trvá déle (je vhodné načíst údaje za sebou, ne hledat po celé paměti).
- Naše třídicí algoritmy jsou určitě vhodné pro vnitřní paměť (obzvlášť heapsort, který nepotřebuje paměť navíc, ale s haldou nevybíravě otřásá).
- Pro vnější paměti je vhodný mergesort.

Vnější třídění

- Načti co největší blok do operační paměti, sestav haldu a při extract-min místo zakořenění posledního zkus načíst další prvek.
- Je-li tento prvek aspoň tak velký jako poslední prvek, co jsme poslali na výstup, zabublej ho.
- Je-li tento prvek menší než poslední prvek, co jsme poslali na výstup, nepřidávej ho do haldy, dokonči nad haldou heapsort a založ novou haldu.
- Takto získáme v mezích možností dlouhé setříděné bloky, na které můžeme pustit `mergesort`.

Vnější třídění s více páskami

- Dnes pro praktické použití už asi passé, ale:
- Stává se, že máme k dispozici pásek více (dnes třeba disků).
- Pak je výhodou mergovat ze všech ostatních na jednu.
- Na každé pásce ovšem máme několik bloků.
- Dojde-li obsah jedné pásky, doběhneme současné mergované "bloky" a začneme mergovat na uvolněnou pásku.
- Jak zorganizovat počty bloků na jednotlivých páskách?
- Zobecněnými Fibonacciho čísly.

Organizace počtu bloků pro mergesort

- Předpokládejme, že máme tři pásky.
- Chceme, aby slévání skončilo jedním dlouhým blokem na jedné pásce":
  (0,0,1)
- Tudíž předtím na ostatních dvou páskách muselo být po jednom bloku:
  (1,1,0)
- Bezprostředně předtím jsme mergovali ze třetí pásky (protože se vyprázdnila) a museli jsme mergovat někam (BÚNO na druhou). Tedy předtím vypadalo rozmístění bloků takto:
  (2,0,1)

- Předtím tudíž výpočet vypadal takto:
  (0,2,3)
- Předtím zase takto:
  (3,5,0)...
- Tedy vždy jde o dvě po sobě jdoucí Fibonacciho čísla (a třetí je nula).
- Vychází totiž rekurence $a_n = a_{n-1} + a_{n-2}$, obecně $a_n = a_{n-1} + ... + a_{n-k}$.

K čemu jsou dynamické proměnné?

- K mnoha algoritmům bychom potřebovali pole proměnlivé délky
- nebo aspoň jinou datovou strukturu proměnlivé délky.
- Jak implementovat frontu a zásobník?
- Použijeme pointery.

Pointery alias ukazatele

- Paměť je organizována lineárně v podobě jednotlivých adres.
- Na těchto adresách jsou ukládány hodnoty (kód, data).
- Paměť zpravidla adresujeme přirozenými čísly, která zapisujeme v šestnáctkové soustavě.
- Na data v paměti si tedy můžeme ukazovat.
- Pod těmito ukazateli můžeme mít údaje libovolných typů, z čehož vyplývá jisté nebezpečí.
- Z toho vzplývá jisté nebezpečí a nutnost být obezřetný.

Pointery – syntax a sémantika

- S pointery pracujeme zpravidla tak, že definujeme datový typ *ukazatel na něco.*
- *Ukazatel na* řekneme pomocí operátoru stříšky:
- `type pint=^integer; {ukazatel na integer}`
- Následně definujeme proměnnou příslušného typu:
  `var ukaz:pint;`
- Pod pointer "koukneme" opět pomocí operátoru stříšky:
  `writeln(ukaz^);`
- Jenže ono to zdaleka není tak snadné!

Organizace paměti s ohledem na program

- Program sestává z kódu, tzv. statických dat, prostoru zásobníku a oblasti haldy.
- Kam ukazuje pointer, který si lehkomyslně vyrobíme?
- Pokud s ním budeme zacházet rozumně, bude ukazovat někam do oblasti haldy, k tomu ale máme ještě daleko.
- Kam tedy pointer ukazuje?
- V lepším případě na adresu 0, v horším na náhodně vybraný kus paměti.
- Pořádek v paměti hlídá allokátor, který ví, které části paměti jsou použité a které ne a může nám přidělit prostor.

Allokátor a jeho použití

- Abychom mohli pod pointer kouknout, musíme ho někam nasměrovat. A to buďto na už existující pointer (`var a,b:pint;... naalokuj b;... a:=b;`),
- anebo "urafnutím" nové paměti: `new(a);`
- Funkci `new` předáváme jako parametr proměnnou typu ukazatel.
- Funkce `new` najde v paměti vhodné místo a nasměruje na něj příslušný pointer.
- Cvičení zimního učiva: Bere `new` parametr hodnotou nebo referencí?
- Od chvíle, kdy máme naalokováno, můžeme pod pointer koukat i zapisovat:
- `new(a); a^:=5; writeln(a^);` Ale pozor na přesměrování ukazatele!

Pointery a práce s nimi

- `var a,b:pint;`
- `new(a);` – naallokuje místo pro proměnnou typu integer
- `a^:=5;` – zapíše pod pointer hodnotu 5.
- `b^:=a^;` – okopíruje pod pointer `b` hodnotu, na kterou ukazuje pointer `a`.
- `b:=a;` – okopíruje pointer (tedy `a` a `b` ukazují na stejné místo).
- Co v kontextu uvedeného kódu udělá `b^:=10;` `writeln(a^);`?
- Pozor, více pointery si můžeme ukazovat na to samé místo!

Deallokace

- Nepotřebujeme-li už příslušné místo, musíme to říci allokátoru voláním funkce `dispose`:
- `dispose(a);`
- Tím prohlásíme, že pod dotyčný pointer už nepotřebujeme.
- Pozor pokud si na totéž místo ukazujeme víckrát, nesmíme provést vícenásobné odallokování!
- Pokud pointer přesměrujeme bez odallokování (ničím si na dotyčné místo neukazujeme), vzniká tzv. garbage, o které má allokátor za to, že je používaná, my však už nemáme, jak se k dotyčnému kusu paměti dostat (tak ho nemůžeme ani odallokovat).
- Některé jazyky (Java, C#) takto odallokovávají (přestaneme si na kus paměti ukazovat). Říká se tomu garbage collector, je to nevýchovné a v Pascalu není implementován.

Typičtější použití pointerů

- V zimě jsme se učili o strukturách (records).
- Struktury nám umožňovaly udržovat údaje různých typů spolu souvisejících. Například dvě souřadnice bodu v rovině, údaje o lidech, údaje o knihách...
  Do recordu se přistupuje operátorem tečky.
- Typicky se dělají pointery na struktury:
- Příklad:

```
type tbod=record
x,y:integer;
end;
pbod=^tbod;
```

- Stále ještě neřešíme původní problém, co když máme bodů víc a nevíme předem kolik?

- "Přivřeme" do struktury ukazatel na další prvek:
- ```
  type pbod=^tbod;
  tbod=record
          x,y:integer;
          next:pbod;
  end;
  ```
- Tomu říkáme spojový seznam.
- Jak poznáme, který ukazatel někam ukazuje?
- Tak, že nepoužitý pointer okamžitě nasměrujeme na nulu, tedy `nil`.
  ```

Spojový seznam čísel – načti a vypiš

Příklad – datové struktury

```
type pint=^sint;
      sint=record
             cislo:integer;
             next:pint;
      end;
var spojak,pom:pint;
```

Spojový seznam čísel – načti a vypiš

Příklad – tělo kódu

```
begin spojak:=nil; pom:=nil;
      while not EOF do
      begin new(pom);
            readln(pom^.cislo);
            pom^.next:=spojak;
            spojak:=pom;
      end;
      while spojak<>nil do
      begin writeln(spojak^.cislo);
            pom=spojak;
            spojak=spojak^.next;
            dispose(pom);
      end;
end.
```