27. 1. 2025

17. Vyhledávací algoritmy

Vyhledávání údajů

- jedna z nejčastějších operací na počítači
- hledá se v obrovských objemech dat
  ↳ databáze, weby, počítačové sítě atd...
- potřeba vyhledávat rychle a efektivně
  ↳ seznam s uložením dat, třídění (ne vždy jde)
- Co je výsledkem vyhledávání?
  a) ověřujeme, že zadaný údaj existuje v prohledávaných datech MNOŽINA
  b) data vázaná na klíč - když je chceme pouze číst
  c) poloha klíče - když chceme nalezená data i měnit
    - index v poli
    - ukazatel na záznam
- např. telefonní seznam - hledáme jméno (jeho polohu), abychom zjistili telefonní číslo (v místě, kde se toto jméno vyskytuje)

Data pro vyhledávání

- obvykle strukturované údaje
  - pole struktur
  - lineární seznam
  - strom - každý prvek je struktura

```
typedef struct {
    char * jmeno;
    char * mesto;
    long tel_cislo;
} Tpolozka;
```

- Klíč = údaj podle kterého vyhledáváme
  - jedna ze složek struktury => JEDNODUCHÝ KLÍČ (např. jméno)
  - více složek struktury => SLOŽENÝ KLÍČ (např. jméno + místo)

```
typedef struct {
    Tdata data;
    Tklic klic;
} TobecnaPolozka;
```

```
TobecnaPolozka pole[N];

// pole[i].klic - klíč, který hledáme, index i hledáme
// pole[i].data - data asociovaná s klíčem, která chceme získat
```

Operace pro vyhledávání
- Porovnávání - isEqual, isLess, strcmp, ==, !=, <, >, <=, >=

např. město

vracíme data nebo index

Klasifikace algoritmů vyhledávání

- Podle místa uložení dat:
  - interní (vnitřní)
  - externí (vnější)
  - kombinované
- Podle principu:
  - **Sekvenční**: - data se prochází sekvenčně
    - data nemusí být seřazena
  - **Nesekvenční**: - data musí být v poli seřazena
    - data lze vyhledávat výrazně rychleji než sekvenčním přístupem
  - **Indexsekvenční**: - přímým přístupem se vyhledá blok dat, ve kterém se pak údaj dohledává sekvenčně
  - **Stromové**: - data jsou uložena ve stromové struktuře přizpůsobené pro rychlé vyhledávání

Vyhledávání v poli

- V neseřazeném poli
  - sekvenční vyhledávání + se zarážkou
    - hledání „hrubou silou"
    - „fulltextové" vyhledávání
- V seřazeném poli
  - sekvenční vyhledávání + se zarážkou
    - v průměru o 1/2 rychlejší než v neseřazeném poli
- Binární vyhledávání
  ↳ nesekvenční vyhledávání, $O(\log_2 N)$
- Hašovací tabulka
  ↳ ukládání i hledání pomocí hašovací funkce s $O(1)$ (!)

Sekvenční vyhledávání

- princip alg.: – sekvenční prochází prvky datové struktury dokud nenajde shodu nebo nedosáhne konce
- datové struktury pro ukládání a sekvenční vyhledávání dat: – Tabulka = jednoduchá reprezentace pomocí pole
  - Seznam = lineární spojový seznam

OBECNĚ

```
int algorithm seqSearch(in: pole, N, hledanyKlic){
  i ← 0
  while (i < N AND
         NOT isEqual(pole[i].klic, hledanyKlic))
  {
    i ← i + 1
  }

  if (i < N) return i          // konec v poli ⇒ našel
  else return -1               // za polem ⇒ nenašel
}
```

- Výhody: – nejobecnější alg.
  - hledá i v neseřazeném poli – některé úlohy nebo souborech (kniha, neseřazená data, časové posloupnosti, ...)
- Nevýhody: – lineární časová složitost O(N) ⇒ pro velká data/pole (> 1M prvků) nepoužitelné
  - zbytečně neefektivní vyhledávání ⇒ hledání hrubou silou

NEJHORŠÍ PŘÍPAD: NENAŠEL

ZJEDNODUŠENĚ

```
int algorithm seqSearch(in: pole, N, hledanyKlic){
  i ← 0
  while (i < N AND pole[i] ≠ hledanyKlic))
  {
    i ← i + 1
  }

  if (i < N) return i          // konec v poli ⇒ našel
  else return -1               // za polem ⇒ nenašel
}
```

Sekvenční vyhledávání se zarážkou

- princip alg: – pole delší o 1 prvek ⇒ pro zarážku
  - do zarážky zkopíruj hledaný klíč
  - sekvenčně hledej klíč
  - našel zarážku? (koncový index)
    - ANO – hledaný klíč se v poli nenachází!
    - NE – nalezený prvek (index) je hledaným prvkem
- Výhody: – funguje i v neseřazeném poli   i < N
  - vždy něco najde ⇒ není potřeba hlídat konec pole ⇒ rychlé (kompaktní)
- Nevýhody: – potřeba upravit pole o 1 prvek delší
  - časová složitost průměr O(N)

```
int algorithm seqStopSearch(in: pole, N, hledanyKlic){
  pole[N-1] ← hledanyKlic      // posledni prvek je zarážka
  i ← 0
//while (NOT isEqual(pole[i].klic, hledanyKlic))
  while (pole[i] ≠ hledanyKlic)
  {
    i ← i + 1
  }
  if (i < N) return i           // konec v poli ⇒ našel
  else return -1                // na zarážce ⇒ nenašel
}
```

Sekvenční vyhledávání v seřazeném poli

- princip: – sekvenčně hledej klíč
  - dokud jej nenajdeš nebo
  - dokud nenajdeš větší klíč nebo
  - dokud nedojdeš na konec pole
  - neporovnání se shoda klíčů (==, !=), ale jejich velikost (<, >)
- pokud našel klíč – jeho index je výsledkem
- pokud našel větší klíč – hledaný prvek v poli není

Sekvenční vyhledávání v seřazeném poli se zarážkou

- analogické použití zarážky
- srovnávej velikosti klíčů, ne jejich shodu
- ohlídej, zda byl prvek nalezen nebo ne

  if(pole[i] == hledanyKlic) return i;
  else return -1;   ! nutné porov.

```
int algorithm seqSortSearch(in: pole, N, hledanyKlic){
  i ← 0
  while (i < N AND pole[i] < hledanyKlic)
  {
    i ← i + 1
  }
  if (i < N AND pole[i] = hledanyKlic)
! //if (i < N)            // TAKTO NE! Proč?
    return i              // konec v poli ⇒ našel
  else return -1          // za polem ⇒ nenašel
}
```

narazil jsem na větší klíč ⇒ NENAŠEL JSEM

- Výhody: – v průměru o 1/2 rychlejší než neseřazené pole
  - se zarážkou – méně testů v cyklu a na konci algoritmu
- Nevýhody: – stále jde o průměrnou složitost O(N)

Binární vyhledávání ⇒ ROZDĚLOVÁNÍ NA POLOVINY

- vyžaduje seřazené pole !
- rekurzivní princip ⇒ zabírá paměť na zásobníku a je pomalejší
  - ↳ jde rozepsat nerekurzivně
- intuitivní princip „hledání ve slovníku"
- má-li úsek pole délku 1 a není v něm hledaný klíč, pak hledaný klíč neexistuje

intuitivní princip:

- hledá se v úseku daném intervalem $\langle L ; P \rangle$
- podívej se na klíč prostředního prvku (index S) daného úseku

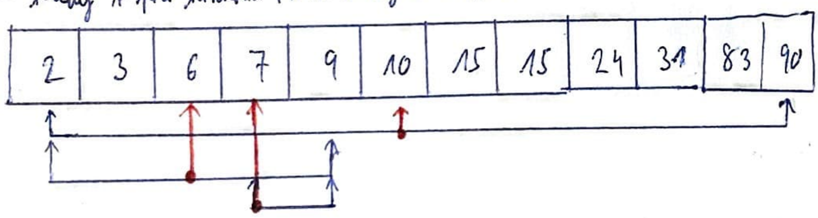
NEREKURZIVNĚ

```
int algorithm binSearchNR(in: pole, l, p, hledanyKlic){
  while (l ≤ p) {
    stred ← (l+p)/2
    if (pole[stred] = hledanyKlic)
      return stred          // našel
    if (pole[stred] < hledanyKlic)
      l ← stred+1           // bude hledat v pravé části
    else
      p ← stred-1           // bude hledat v levé části
  }
  return -1 // nenašel
}
```

! POZOR NA DÉLKU POLE !

binSearch(pole, 0, 10, 7)

- hledej v poli hodnotu 7 mezi indexy 0-10 včetně



NEJHORŠÍ PŘÍPAD: PRVEK V POLI NENÍ

REKURZIVNĚ - moc se nepoužívá

```
// l je index levého kraje, p je index pravého kraje pole
int algorithm binSearchR(in: pole, l, p, hledanyKlic){
  if (l > p) return -1 // koncová podmínka - nenašel
  stred ← (l+p)/2
  if (pole[stred] = hledanyKlic)
    return stred          // koncová podmínka - našel
  if (pole[stred] < hledanyKlic)
    return binSearchR(pole, stred+1, p, hledanyKlic)
  else
    return binSearchR(pole, l, stred-1, hledanyKlic)
    // proč je tady return?
}
```

- Výhody: - nejrychlejší algoritmus nad seřazeným polem $O(\log_2 N)$ (základ 2)
  - v poli o 1 000 000 prvků ⇒ max 20 kroků
  - 1 000 000 000 prvků ⇒ max 30 kroků
- Nevýhody: - nefunguje na neseřazeném poli
  - ! - duplicitní klíče ⇒ nemusí najít vždy první, musí se dohledat sekvenčně

5

Hašovací tabulka ⇒ Hash ⇔ hašé, sekaná, rozemletí

Hašovací funkce (hash function)

nikdy MODULO (prvočíslo) → nějaký hezký nahoře

- Matematická funkce : hash (obrovské číslo) ⇒ malé číslo
- někdy nazýváme mapovací funkce
- nemá inverzní funkci (z výsledku nelze zjistit vstupní hodnotu)
- výsledku se říká hash nebo otisk
- využití v kryptografii, pro kontrolní součty (např. MD5)
- kvalitní hashovací funkce ⇒ kvalitní tabulky

CHCEME VELIKOU NESTABILITU

Hashovací tabulka

- Tabulka s rozptýlenými položkami
- Ideální struktura!
  - Vkládání prvku se složitostí $O(1)$ — konstantní
  - Vyhledání prvku se složitostí $O(1)$
  - ale toto dosáhnout jen pro speciální případy omezující vstupní data. Obecně jsou tyto ideální vlastnosti nedosažitelné!

Realizace

- musíme dopředu znát množinu všech klíčů $K = \{k_1, k_2, \dots, k_m\}$
- musíme mít k dispozici mapovací funkci :
  - $f(k_i) = i$, $i = \{0; 1; 2; \dots, m-1\}$
  - každý klíč přepočítá na index v poli
  - v ideálním případě je jednoznačná, tj. pro různé různé klíče neexistuje stejný index ⇒ jinak are KOLIZE

↳ DEGENEROVANÁ TABULKA

- Tabulka je pole

```
typedef struct {
  T data data;
  Tklic klic;
  bool obsazeno;
} Tpolozka;
```

Tpolozka tabulka[N];

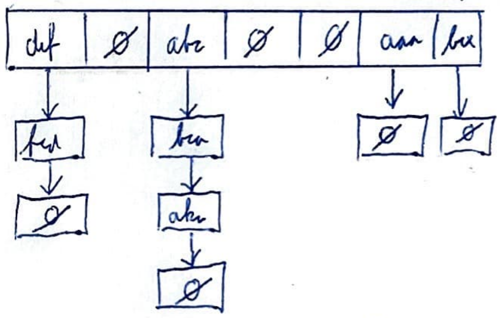
NEJHORŠÍ PŘÍPAD: EXTRÉMNĚ MOC KOLIZÍ

- Operace : init, vlož, vyjmi, najdi

Řešení v praxi

- spokojíme se s funkcí, která produkuje v rozumné míře
- Hashovací tabulka ⇒ indexsekvenční struktura
  - prvky tabulky — malý lineární seznam prvků s kolizními klíči
  - hledání — mapovací funkce najde prvek v tabulce + sekvenční vyhledání v lineárním seznamu

kolidující klíče



Realizace v. 2

```
typedef struct tpol {
  T data data;
  Tklic klic;
  struct tpol * dalsi;
} Tpolozka;
```

Tpolozka * tabulka[N]

Obsazeno/Volno ⇒ není / je NULL

BVS — Binární vyhledávací strom

✱ TÉMA 9 — PREORDER ASI