Pracovní list 11: Vyhledávání

Co už máme znát

- princip sekvenčního hledání;
- princip hledání s logaritmickou časovou složitostí (BVS, půlení intervalu);
- princip hledání s konstantní časovou složitostí;
- indexsekvenční hledání – hešovací tabulka;
- tvorba programových modulů;
- vnější operace programového modulu.

Kontrolní otázky

11.1 Co zahrnuje abstraktní datový typ „Vyhledávací tabulka“?

11.2 Kdy použijeme sekvenční vyhledávání?

11.3 Jak se liší vyhledávání se zarážkou od běžného sekvenčního vyhledávání?

11.4 Jakou časovou složitost má vyhledávání v BVS za běžných okolností?

11.5 Kdy lze aplikovat metodu hledání půlením intervalu?

11.6 Jakou časovou složitost má vyhledávání půlením intervalu a jak se složitost liší od vyhledávání v BVS?

11.7 Jak je organizována metoda vyhledávání pomocí indexace?

11.8 Jaká je nevýhoda metody vyhledávání pomocí indexace?

11.9 Jak je organizována hashovací (hešovací) tabulka?

11.10 Které faktory ovlivňují efektivnost hešovací tabulky?

Příprava na cvičení

Ve cvičení budeme potřebovat překladač jazyka C++, editor pro přípravu zdrojových textů a vybavení příkazového řádku. Pro jednotlivé úlohy jsou k dispozici soubory s daty, případně výsledné soubory v adresáři `/home/rybicka/vyuka/progt/cecko/cviceni/cv11` na serveru `akela`. Konkrétní jména těchto souborů jsou uvedena u jednotlivých úloh.

Řešené příklady

Příklad 11.1 Implementujte modul s abstraktním typem „Vyhledávací tabulka“ a s jeho třemi základními operacemi. Implementaci proveďte polem.

Řešení: Implementujeme abstraktní datový typ s operacemi inicializace, vložení údaje, vyhledání údaje. Protože jde o strukturu s obecnými daty, bude abstraktní typ reprezentován záznamem se třemi složkami: základní pole obecných ukazatelů, počet aktuálně obsazených prvků a ukazatel na funkci, která je schopna porovnat konkrétní datové složky – ta je nutná k vyhledávání, kde potřebujeme porovnat data a zjistit, zda jsou shodná. Modul pracovně nazveme `vyhltab1` – vyhledávací tabulka, 1. verze. Nejprve tedy vytvoříme hlavičkový soubor `vyhltab1.h` následujícího obsahu:

```
#ifndef VYHLTAB1_H
#define VYHLTAB1_H

const int Kapacita = 100000; //max. počet údajů ve struktuře
typedef void* ZaklPole[Kapacita];
typedef bool (*TypPorovnej)(void*, void*);
struct VyhledTab {
   ZaklPole ZP;       //základní pole
   int Obsazeno;      //počet obsazených prvků pole
   TypPorovnej Rovno; //porovnávací funkce, true -> data shodná
};

void VTInit(VyhledTab &VT, TypPorovnej X);
void VTVloz(VyhledTab &VT, void *Data);
bool VTNajdi(VyhledTab VT, void *Data);

#endif
```

Implementační část modulu v souboru `vyhltab1.cpp` realizuje naznačené operace. Vzhledem k jednoduchosti tohoto přístupu není nezbytný další podrobný komentář:

```
#include ”vyhltab1.h”

void VTInit(VyhledTab &VT, TypPorovnej X){
  VT.Obsazeno=0;
  VT.Rovno=X;  //přiřazení uživatelské funkce
}

void VTVloz(VyhledTab &VT, void *Data){
  VT.ZP[VT.Obsazeno]=Data;  //nová data na konec pole
  VT.Obsazeno++;
}

```

```
677 bool VTNajdi(VyhledTab VT, void *Data){
678   int Index=0;   //obyčejné sekvenční hledání
679   while (Index<VT.Obsazeno and not VT.Rovno(VT.ZP[Index],Data))
680      Index++;
681   return Index<VT.Obsazeno;
682 }
```

Příklad 11.2 Implementujte hešovací tabulku, do níž budou vkládány celočíselné hodnoty v intervalu $\langle -9999; 9999 \rangle$.

Řešení: Implementace předpokládá základní pole o určitém počtu indexů (s tím lze ještě manipulovat), složkami pole budou ukazatele na lineární seznamy synonym. Prvkem seznamu je záznam s celočíselnou datovou složkou a ukazatelem na další prvek. Hešovací funkce je vzhledem k celočíselným datům poměrně jednoduchá: posune interval více do kladných hodnot a vytvoří požadovaný index jako zbytek po dělení počtem indexů základního pole. Celou implementaci uvedeme s případnými komentáři ve zdrojovém textu. Implementovanou tabulku vyzkoušíme na následující úloze: V souboru `celacisla.txt` se nachází milion čísel. Vložte je do tabulky. Pak čtěte soubor ještě jednou, každé číslo vydělte dvěma a vyhledejte je v tabulce. Kolik takových čísel se v tabulce nachází?

```
683 #include <iostream>
684 #include <fstream>
685 using namespace std;
686
687 const int MinData = -9999; //vymezení intervalu dat
688 const int MaxData = 9999;
689 const int Max = 10000;     //velikost základního pole
690 typedef int TypData;
691 struct Clen {
692    TypData Data;
693    Clen *Dalsi;
694 };
695 typedef Clen *UkClen;
696 typedef UkClen ZaklPole[Max];
697 typedef int (*HashFce)(TypData);
698 struct HashTab {
699   ZaklPole ZP;
700   HashFce H;      //hešovací funkce je součástí tabulky
701 };
702
703 int Hesuj(TypData D){
704   return (D - MinData) % Max;
705 }
706
707 void HTInit(HashTab &HT, HashFce X){
708   for (int i=0; i<Max; i++) HT.ZP[i]=NULL;
```

```
709   HT.H = X;  //uživatelská hešovací funkce se vloží do struktury
710 }
711
712 void HTVloz(HashTab &HT, TypData D){
713   UkClen Pom = new Clen;
714   int I = HT.H(D);        //hešování -- kam přijdou data?
715   Pom->Data = D;          //naplnění nového záznamu
716   Pom->Dalsi = HT.ZP[I];  //vložení na začátek seznamu
717   HT.ZP[I] = Pom;
718 }
719
720 bool HTNajdi(HashTab HT, TypData D){
721   UkClen Pom = HT.ZP[HT.H(D)]; //hešování a indexace
722   while (Pom!=NULL and Pom->Data!=D) //sekvenční hledání
723      Pom = Pom->Dalsi;
724   return Pom != NULL;
725 }
726
727 int main(){
728   ifstream Cisla (”celacisla.txt”);
729   if (not Cisla.is_open()) {
730     cerr << ”Vstupní data nelze číst.” << endl;
731     return 4;
732   }
733   HashTab T;   //hešovací tabulka
734   int Cis;     //proměnná pro čtená data
735   int pocet=0;
736   HTInit(T, Hesuj);
737   while (Cisla>>Cis) HTVloz(T, Cis);
738   Cisla.close();
739   Cisla.open(”celacisla.txt”);
740   while (Cisla>>Cis) {
741     Cis /= 2;
742     if (HTNajdi(T, Cis)) pocet++;
743   }
744   cout << ”Nalezeno ” << pocet << ” vzorků.”<<endl;
745   return 0;
746 }
```

Na tomto příkladu lze taktéž sledovat závislost rychlosti hledání na velikosti základního pole. Čísla ve zdroji dat jsou generována generátorem náhodných čísel, takže jejich rozdělení je přibližně rovnoměrné (je tam provedeno několik ručních zásahů, není to tedy přesně pouhý výstup generátoru rovnoměrného rozložení). Je pravděpodobné, že počty synonym budou všude přibližně stejné (můžete ověřit analytickou operací, která vypíše průměrný počet, maximální počet a minimální počet synonym). Zvětšíme-li pak základní pole například 10×, poklesne počet synonym na desetinu,

zrychlí se i vyhledávání na přibližně desetinu, protože majoritní čas je věnován sekvenční části hledání.

Příklady

Příklad 11.3 Využijte modul z příkladu 11.1 pro tuto úlohu: Vložte do vyhledávací tabulky všechna slova ze souboru `data.txt` (tj. čtěte řetězce metodou `proud>>R`). Pak soubor `data.txt` čtěte ještě jednou, čtené řetězce delší než 2 znaky obraťte a vyhledejte je ve vyhledávací tabulce. Na výstupu dostanete seznam řetězců delších než 2 znaky, které dávají smysl čtené zleva i zprava.

Řešení: Ze souboru se získá 61 984 řetězců. Počet nalezených řetězců delších než 2 znaky je 62, ale řada z nich se opakuje; počet unikátních řetězců je 18.

Příklad 11.4 Vytvořte modul `vyhledtab2` implementující vyhledávací strukturu pomocí binárního vyhledávacího stromu, rozhraní bude stejné jako u příkladu 11.1.

Příklad 11.5 Použijte modul z příkladu 11.4 pro řešení stejné úlohy, jako je v příkladu 11.3.

Příklad 11.6 Implementujte modul `vyhledtab3` vycházející z příkladu 11.1, ale data budou ukládána do pole uspořádaně a vyhledávání bude probíhat metodou půlení intervalu. Porovnávací funkci navrhněte tak, že jejím výsledkem bude celočíselná hodnota 0 při rovnosti, hodnota -1 nebo 1 při nerovnosti (první je menší, druhý je menší).

Příklad 11.7 Modul `vyhledtab3` opět použijte na řešení úlohy z příkladu 11.3.

Příklad 11.8 Implementujte speciální vyhledávací strukturu pro ukládání celočíselných hodnot v intervalu $\langle -9999; 9999 \rangle$ formou multimnožiny.

Příklad 11.9 Vyhledávací strukturu z příkladu 11.8 použijte pro následující úlohu: Čtěte čísla ze souboru `celacisla.txt` a uložte je do struktury. Následně čtěte tento soubor ještě jednou, každé číslo vydělte celočíselně dvěma a zjistěte, zda se toto číslo vyskytuje ve vyhledávací struktuře.

Řešení: Stejná úloha je použita v řešeném příkladu, na těchto dvou implementacích můžete porovnat rychlost vyhledávání. Celkem by mělo být nalezeno 994 093 vzorků.

Příklad 11.10 Vytvořte modul `vyhledtab4` implementující ukládání obecných dat formou hešovací tabulky. Hešovací funkci navrhněte jako vnější operaci.

Příklad 11.11 Využijte modul z příkladu 11.10 opět pro stejnou úlohu jako v příkladu 11.3. Vyzkoušejte různé možnosti hešování řetězců, jejichž účinek si pak můžete ověřit příkladu 11.12.

Příklad 11.12 Přidejte do modulu `vyhledtab4` operaci vypisující 10 indexů hešovací tabulky s největším počtem synonym. Na základě této analytické operace si ověřte, jaký vliv na kvalitu rozptylování má počet indexů základního pole a konstrukce hešovací funkce. Najděte takovou kombinaci uvedených faktorů, která umožní snížit maximální počet synonym na méně než 20. Pozor – hešovací funkce úzce souvisí s rozsahem indexů, musí být navržena tak, aby využila celého prostoru základního pole.

Co máme po cvičení umět

- možnosti vyhledávání v sekvenční struktuře,
- možnosti vyhledávání metodou půlení intervalu,
- strukturu pro vyhledávání indexací,
- strukturu pro indexsekvenční hledání (hešovací tabulka),
- porovnání jednotlivých způsobů vyhledávání.

Kontrolní otázky

11.11 Jaká je základní výhoda sekvenčního vyhledávání?

11.12 Jaká je nevýhoda vyhledávání metodou půlení intervalu?

11.13 Jaké nebezpečí hrozí při použití binárního vyhledávacího stromu jako vyhledávací struktury?

11.14 Jaké jsou výhody a nevýhody vyhledávání indexací?

11.15 Jaké jsou možnosti rozptylování v případě řetězcových klíčů v hešovací tabulce?

11.16 Jak lze zjistit účinnost rozptylování v hešovací tabulce?