Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 21.02. 2002
Url: http://www.builder.cz/art/cpp/cpphalda.html

Halda v C++

Než se začneme věnovat algoritmům pro práci s haldou, které jsou v STL, vysvětlíme si, co vlastně halda je. Nejprve pár matematických pojmů.

Graf

Graf je uspořádaná dvojce (V,H), kde V je množina vrcholů (uzlů) a H je množina hran. Hrana je dvojce vrcholů. Za touto definicí se nesmí hledat žádná magie. Pro názornost uvedu příklad. Představme si 4 počítače. Označme si je A, B, C, D. Počítače jsou propojeny v síti. Počítač A je propojen s B a s C. Počítač B je tedy spojen s A a dále s D. Počítač C je propojen s počítači A, D. Počítač D je propojen s B a C. Je dobré si tuto síť nakreslit. Jednotlivé počítače budeme považovat za vrcholy. Spoje mezi počítači jsou "dráty" se dvěma konci. Každý "drát" lze přesně určit pomocí počítačů na jeho dvou koncích. Jedná se o hrany. Naše počítačová síť je graf, tedy uspořádaná dvojce ({A,B,C,D} , { (A,B) , (A,C) , (B,D) , (C,D) }). Stejně tak graf je například i železnice. Vrcholy jsou nádraží a hrany jsou koleje spojující nádraží. Graf je v informatice poměrně častý pojem. V grafu je definováno mnoho algoritmů, například nalezení nejlevnější cesty, atd...

Cesta v grafu

Cesta v grafu z počátečního vrcholu do koncového je posloupnost vrcholů, které jsou spojeny hranou. Tato posloupnost začíná počátečním vrcholem, a končí koncovým. Definice by se dala uvést exaktněji, ale pro pochopení myslím stačí. Například v naší počítačové síti je cesta z A do D třeba posloupnost (A,B,D), nebo (A,C,D), ale také třeba (A,B,D,C,A,C,D).

Souvislý graf

Souvislý graf je graf, ve kterém existuje alespoň jedna cesta mezi libovolnou dvojicí vrcholů. Naše počítačová síť je souvislá. Kdyby jsme ale přidali další počítač (nazvěme ho E), který by nebyl propojen s žádným jiným počítačem, graf by už nebyl souvislý. Protože např z A do E nevede žádná cesta. Když přidáme počítač F, který bude propojen s E, stále bude graf nesouvislý. Pořád nevede cesta z A do E.

Strom

Každý, kdo programuje už asi o stromu něco slyšel, zvláště o binárním stromu. Strom je specielní případ grafu. Strom je souvislý graf, ve kterém mezi

libovolnými vrcholy vede právě jedna cesta. Strom je souvislý graf, protože mezi každými vrcholy existuje jedna cesta. Náš příklad počítačové sítě není strom. Jak jsem uvedl u definice cesty, například z A do D vede více než jedna cesta. Zrušíte-li v našem grafu (poč. síti) jednu libovolnou hranu, vznikne strom. Můžete si to nakreslit a přesvědčit se o tom.

Binární strom

Binární strom je strom, kde každý vrchol má maximálně 2 následovníky. Hodně populární formou lze říci, že jednou hranou jsem do vrcholu přišel, a dále mám maximálně dvě možné hrany, po kterých jít dál.

Halda

Halda je binární strom, který má ohodnocené vrcholy. Ohodnocení vrcholů znamená, že ke každému vrcholu je přiřazena nějaká jeho hodnota. Například číslo. Aby byl ohodnocený binární strom haldou, musí platit, že každý vrchol NEMÁ menší hodnotu než kterýkoliv z jeho následovníků. Často se halda definuje tak, že ohodnocení vrcholu není větší, než ohodnocení jeho následovníků. Standardní halda v C++ to má ale naopak. Není to příliš důležité. Důležitý je spíše fakt, že existuje nějaká relace, která dokáže prvky seřadit. I v C++ budeme mít možnost tuto relaci zadat. Pro haldu ale platí ještě jedna podmínka ohledně uspořádání vrcholů a hran. Vysvětlím to opět velmi zjednodušeně. Mám graf o jednom vrcholu - což je halda. Mám tím pádem první řadu vrcholů. Přidám nový vrchol do druhé řady zleva. Další vrchol přidám zase zleva vedle již existujících vrcholů. Až dokončím druhou řadu, začnu vytvářet novou řadu opět zleva. Novou řadu dokončím tehdy, když každý vrchol předchozí řady má dva následníky. V haldě má tedy každý vrchol, který není v poslední a předposlední řadě dva následníky. V poslední řadě jsou vrcholy bez následovníků. V předposlední řadě mohou mít vrcholy 2, 1, nebo žádného následníka. V předposlední řadě jsou z leva vrcholy s dvěma následníky, jestliže existují, potom je jeden, nebo žádný vrchol s jedním následovníkem a následují vrcholy bez následovníků, pokud existují.

Implementace haldy

Haldu lze velmi efektivně uchovávat v poli. Kořen haldy by byl první prvek v poli. Jeho dva následovníci by byli druhý a třetí prvek v poli. Následovníci druhého prvku v poli by byli 4. a 5. prvek v poli atd... Tedy i-tý prvek v poli by měl následovníky na pozicích 2 * i, 2 * i + 1. Indexy 2 * i, nebo 2 * i + 1 mohou být mimo rozsah pole. V takovém případě následníci neexistují. Protože v C i v C++ se indexuje pole od nuly, nikoliv od jedné musíme si naše dva vzorce pro polohu následovníků upravit takto: i-tý prvek má za své následovníky prvky s indexem 2 * i + 1, 2 * i + 2, nebo prvky nemá, jeli výsledný index mimo rozsah pole. To bylo jen tak pro zajímavost, pro práci se standardní haldou v C++ nic takového vědět nemusíme.

Halda v C++

V C++ neexistuje pro haldu nějaký kontejner, nebo adaptér kontejneru. Haldu můžeme vytvořit v libovolném kontejneru mezi zadanými iterátory. Pro práci s haldou existují standardní algoritmy `make_heap`, `pop_heap`, `push_heap`, `sort_heap`.

- `template <class RandomAccessIterator> void make_heap(RandomAccessIterator zacatek, RandomAccessIterator konec);` - vytvoří v úseku daném iterátory začátek a konec haldu. Každý prvek haldy není menší, než jeho následovníci. Halda bude v posloupnosti uložena tak, jak jsem uvedl v odstavci o implementaci haldy. Haldu lze vytvořit v kontejneru, kterým lze procházet pomocí iterátorů s náhodným přístupem. Já vám doporučuji používat `vector`, ale není to podmínka.
- `template <class RandomAccessIterator, class TRelace> void make_heap(RandomAccessIterator zacatek, RandomAccessIterator konec, TRelace relace);` - obdobně jako minulá funkce. Jen je nový parametr udávající relaci, podle které bude halda organizována. Může se opět jednat o funkční objert, nebo o ukazatel na funkci.
- `template <class RandomAccessIterator> void push_heap(RandomAccessIterator zacatek, RandomAccessIterator konec);` - slouží k vložení prvku na haldu. Předpokládá se, že v rozmezí daném iterátory začátek a konec - 2 prvky je halda. Poslední prvek (konec - 1 prvek ; prvek hned před prvkem na který ukazuje iterátor konec) je nový prvek, který do haldy vkládám. Vkládání (stejně jako odebírání) prvku na haldu je trochu zvláštní. Uvedl jsem jej v příkladu.
- `template <class RandomAccessIterator, class TRelace> void push_heap(RandomAccessIterator zacatek, RandomAccessIterator konec, TRealce realce);` - obdobně jako minulá funkce. Jen je nový parametr udávající relaci, podle které bude halda organizována. Může se opět jednat o funkční objert, nebo o ukazatel na funkci.
- `template <class RandomAccessIterator> void pop_heap(RandomAccessIterator zacatek, RandomAccessIterator konec);` - slouží k odebrání prvku z haldy.
- `template <class RandomAccessIterator, class TRelace> void pop_heap(RandomAccessIterator zacatek, RandomAccessIterator konec, TRealce realce);` - obdobně jako minulá funkce. Jen je nový parametr udávající relaci, podle které bude halda organizována. Může se opět jednat o funkční objert, nebo o ukazatel na funkci.
- `template <class RandomAccessIterator> void sort_heap(RandomAccessIterator zacatek, RandomAccessIterator konec);` - převede haldu v oblasti dané iterátory začátek a konec na seřazenou posloupnost.
- `template <class RandomAccessIterator, class TRelace> void sort_heap(RandomAccessIterator zacatek, RandomAccessIterator konec, TRelace relace);` - obdobně jako minulá funkce. Jen je nový parametr udávající relaci, podle které bude posloupnost seřazena. Může se opět jednat o funkční objert, nebo o ukazatel na funkci.

Metody `pop_heap`, `push_heap` vlastně nevloží, případně neodeberou prvek s haldy. Slouží pouze k přeorganizování haldy. Chci-li vložit nový prvek, vložím nový prvek do kontejneru a poté zavolám `push_heap`. Chci-li odebrat prvek s haldy (odebírá se vždy kořen), přečtu si první prvek na haldě, zavolám `pop_heap` a odeberu poslední prvek z kontejneru. Funkce `pop_heap` vlastně přehodí první prvek s posledním a přeorganizuje haldu bez posledního prvku. Příklad:

```
#include<algorithm>
#include<iostream>
#include<vector>

using namespace std;

int main()
```

```
{
    int pole[10] = {7, 3, 2, 1, 1, 0, 8, 9, 4, 5};
    vector<int> v(pole,&pole[10]);

    // Vytvoří haldu
    make_heap(v.begin(), v.end());
    cout << "Halda:" << endl;
    copy(v.begin(),v.end(),ostream_iterator<int>(cout,"\t"));
    cout << endl;

    // Vložím prvek
    v.push_back(5);
    cout << "Halda pred operaci push_heap:" << endl;
    copy(v.begin(),v.end(),ostream_iterator(cout,"\t"));
    cout << endl;
    push_heap(v.begin(),v.end());
    cout << "Halda po operaci push_heap:" << endl;
    copy(v.begin(),v.end(),ostream_iterator<int>(cout,"\t"));
    cout << endl;

    // Vyberu prvek:
    pop_heap(v.begin(),v.end());
    cout << "Halda po operaci pop_heap:" << endl;
    copy(v.begin(),v.end(),ostream_iterator<int>(cout,"\t"));
    cout << endl;
    cout << "Odebral jsem: " << v.back() << endl;
    v.pop_back();

    // Setřídím haldu
    sort_heap(v.begin(),v.end());
    copy(v.begin(),v.end(),ostream_iterator<int>(cout,"\t"));
    cout << endl;

    /* Použiji jinou relaci: */
    make_heap(v.begin(),v.end(),greater<int>());
    cout << "Pouziji jinou relaci:" << endl;
    copy(v.begin(),v.end(),ostream_iterator<int>(cout,"\t"));
    cout << endl;
    v.push_back(2);
    push_heap(v.begin(),v.end(),greater<int>());
    copy(v.begin(),v.end(),ostream_iterator<int>(cout,"\t"));
    cout << endl;
    return 0;
}
```

Halda má mnoho využití. Na haldě je založeno řazení heap-sort. Pomocí haldy lze implementovat prioritní frontu atd...

V úvodu článku jsem některé matematické pojmy definoval velmi zjednodušenou, populární formou. Snad se skuteční matematici nebudou zlobit. Příště si shrneme vše, co jsme si o standardních algoritmech řekli. Tím téma standardních algoritmů ukončíme.

------ http://www.builder.cz ------