Una petita introducció als diccionaris

29 de març de 2019

- Aquest document conté exercicis que cal resoldre en el Jutge (a la lista corresponent al curs actual) i que aquí estan assenyalats amb la paraula *Jutge*.

1 Introducció

Tot seguit veurem dos casos que ens serviran per motivar l'ús d'una nova estructura de dades: els diccionaris.

1.1 Cas 1

Suposem que volem poder consultar l'edat o el pes d'una persona donat el seu DNI. Suposem que els DNIs poden prendre valors entre 0 i `DNI_max`. La informació per a la consulta la podríem emmagatzemar en un vector de mida `DNI_max`+1 de la forma següent

```
#include <vector>

using namespace std;

const int DNI_max = 1000;

struct info_persona {
   double edat;
   double pes;
};

vector<info_persona> v(DNI_max + 1);
```

Obtenir l'edat o el pes d'una persona amb DNI x seria immediat mitjançant `v[x].edat` o `v[x].pes`. No obstant, el problema és que en realitat `DNI_max` és

un nombre de l'ordre de $4 \cdot 10^7$. Si en la nostra aplicació només hem d'emmagatzemar informació per als 300 alumnes de l'assignatura, cal gastar un vector de mida $4 \cdot 10^7$ aprox?

1.2 Cas 2

Suposem que volem poder consultar l'edat o el pes d'una persona però no mitjançant el seu DNI sinó el seu nom.

La informació per la consulta la podríem emmagatzemar en un vector de mida `NPERSONES` de la forma següent

```
#include <vector>
#include <string>

using namespace std;

const int NPERSONES = 300;

struct info_persona {
   string nom;
   double edat;
   double pes;
};

vector<info_persona> v(NPERSONES + 1);
```

Per a obtenir el pes d'una persona amb nom x podríem fer una cerca lineal (si no tenim el vector ordenat per nom) o una cerca dicotòmica (si tenim el tenim ordenat per nom). En tots dos casos ens veiem obligats a fer feina addicional de programació: adaptar un algorisme de cerca, i en el cas de tenir el vector ordenat per nom, mantenir-lo ordenat cada cop que afegim o esborrem una persona. A més, la implementació en un vector ens obliga a definir una capacitat màxima del vector (o a recórrer al mètode `push_back` dels vectors). S'ho val?

2 Diccionaris

La solució a totes els problemes vistos fins ara són els diccionaris, una estructura de dades bàsica en llenguatges de programació com ara Python i que en C++ es troben a la STL. C++ ofereix diferents opcions per a associar informació (p.e. edat, pes) a una clau (p.e. nom o DNI). Aquí triarem la de la classe genèrica `map` (en C++11 comptem també amb la classe `unordered_map`. Vegem-ho en els exemples següents.

2.1 Cas 1 amb `map`

La informació per a la consulta la podríem emmagatzemar en un `map` de la forma següent:

```
#include <map>

using namespace std;

struct info_persona {
   double edat;
   double pes;
};

map<int, info_persona> d;
```

Fixeu-vos que a diferència de les classes genèriques que hem vist fins ara (`stack`, `queue`, `list`, `Arbre`,...), que tenen uns sol paràmetre de tipus, la classe `map` té dos paràmetres: el primer tipus de la clau i el tipus del valor associat. En un `map<K,V>`, `K` és el tipus de la clau i `V` és el tipus del valor associat.

Simplement amb `d[x].edat` o `d[x].pes` s'obté l'edat o el pes d'una persona amb DNI x. Fixeu-vos que accedim a la informació com si es tractés d'un vector. La despesa de memòria del map es proporcional al seu nombre d'elements, així emmagatzemar-hi la informació dels 300 alumnes de PRO2 gasta una quantitat de memòria proporcional a 300. Amb el `map` de STL paguem el preu d'una certa ineficiència: mentre si usésim un vector el cost de les operacions `d[x].edat` o `d[x].pes` seria constant, ara el cost es logarítmic respecte el nombre d'elements emmagatzemats.

Un restricció important dels `maps` és que a una clau només se li pot associar un valor.

2.2 Cas 2 amb `map`

La informació per la consulta la podríem emmagatzemar en un `map` de la forma següent

```
#include <map>
#include <string>

using namespace std;

struct info_persona {
   double edat;
   double pes;
};

map<string, info_persona> d;
```

L'edat o el pes d'una persona amb nom y és com en el cas anterior: `d[x].edat` o `d[x].pes`. No cal fer res mes. El cost temporal d'aquestes operacions és logarítmic com si en el cas 2 original ens haguéssim preocupat de mantenir el vector ordenat i fer-hi cerques dicotòmiques per consultar la informació associada a un nom.

2.3 Recorregut de diccionaris

Un diccionari pot ser recorregut com si fos una llista mitjançant iteradors. Els elements d'un `map<K,V>` són de tipus `pair<K,V>`[1]. Per tant, els iteradors d'un `map<K,V>` referencien elements de tipus `pair<K,V>`. Per exemple per escriure la informació del map del cas 2 creixentment per nom podem fer

```
for (map<string, info_persona>::const_iterator it = d.begin(); it != d.end(); ++it) {
   cout << it->first << " " << it->second.edat << " " << it->second.pes << endl;
}
```

El cost temporal d'aquest recorregut és lineal respecte la mida de `d` perquè en un `map` qualsevol, el cost de `begin()`, `end()` i el passar d'un element al següent (o l'anterior) són constants.

Noteu que podem modificar el valor de l'element referenciat per un iterador amb l'operador de desreferenciació, *però no la clau.* A més l'iterator no pot ser `const`

```
it->second.edat += 10; // correcte
it->first = "Maria"; // incorrecte
```

2.4 Com saber la mida d'un `map`

Tal com hem vist en amb piles, cues i llistes de STL, `d.size()` ens permet saber la mida del `map d` (el nombre d'elements emmagatzemats). `d.empty()` ens permet saber si és buit.

2.5 Com consultar la informació

Hi ha dues maneres de consultar la informació. Una és mitjançant l'operador `[...]` a l'estil dels vectors. Una altra es mitjançant el mètode `find` que retorna un iterator a l'element buscat, si s'hi troba, i l'`end()` del map si no. Vegeu-ho amb l'exemple següent:

```
map<string, info_persona>::const_iterator it = d.find("Albert");
if (it == d.end()) cout << "L'Albert no hi és" << endl;
else cout << "L'Albert pesa " << it->second.pes << " kilos" << endl;
```

El mètode `find` és la forma més potent de consultar perquè permet controlar el cas que la informació no hi sigui i, per exemple, no fer res en aquest cas. Per contra, si `Albert` no es troba a `d`, `d["Albert"].pes/edat`, crea una entrada al diccionari amb clau "Albert"que queda inicialitzada amb una valor per defecte de 0 de pes i edat, que potser ens interessa o potser no. Ho podeu veure executant el següent programa:

```
#include <string>
#include <map>
```

[1] Vegeu al final d'aquest document un repàs de l'ús de `pair`

```
#include <iostream>

using namespace std;

struct info_persona {
   double edat;
   double pes;
};

int main() {
   map<string, info_persona> d;
   cout << "El diccionari té mida " << d.size() << endl;
   cout << "L'Albert pesa " << d["Albert"].pes << " kilos" << endl;
   cout << "El diccionari té mida " << d.size() << endl;
}
```

Fixeu-vos com la mida de `d` passa de 0 a 1.

2.6 Com afegir elements a un `map`

`d["Albert"].edat = 18` afegeix a `d` una entrada amb `Albert` si `Albert` no hi era deixant el camp `pes` amb valor per defecte (si hi era, simplement n'actualitza `edat`).

Alternativament es pot afegir un element amb el mètode Recordant que els elements d'un `map<K,V>` són de tipus `pair<K,V>`, s'entén clarament l'ús de `insert` següent:

```
info_persona info;
info.edat = 18;
info.pes = 64;
d.insert(make_pair("Manel", info));
// equivalent a d.insert(pair<string, info_persona>("Manel", info));
// però més curt
```

Noteu que si fem `insert` una altra vegada amb `Manel`, el map no canvia. Això ho podem saber amb la informació que retorna la crida:

```
info.edat = 30;
info.pes = 78;
pair<map<string, info_persona>::iterator, bool> p;
p = d.insert(make_pair("Manel", info));
// en aquest punt p.second es fals; p.first apunta a l'element de d
// amb clau "Manel"; la info es 18 i 64, no 30 i 78
```

2.7 Com esborrar elements d'un `map`

Els elements es poden esborrar de diferents formes. Una amb el mètode `erase` que rep com a paràmetre una clau. Per exemple, podem esborrar `Albert` del

`map` amb un codi com el següent:

```
if (d.erase("Albert") == 0) cout << "Albert no hi era" << endl;
else cout << "Hem esborrat Albert" << endl;
```

Hi ha una altra versió del mètode `erase`, que rep com a paràmetre un iterador a l'element que volem esborrar. Per exemple, podem esborrar `Albert` del `map` amb un codi com el següent:

```
it = d.find("Albert");
if (it != d.end()) d.erase(it);
```

Si sabem que `Albert` pertany a `d` podem fer simplement

```
d.erase(d.find("Albert")); // equivalent a d.erase("Albert");
```

En `C++11` aquesta darrera versió torna un iterador al següent element a l'esborrat, o a l'`end()` en cas que l'element esborrat sigui el darrer; així s'assoleix un comportament més semblant al cas de llistes.

3 Qüestions avançades en l'ús dels diccionaris

3.1 Pas de paràmetres

Un operació que usi el mètode [...] d'un `map` passat com a paràmetre necessita una referència modificable del `map`. Per exemple,

```
void consulta(const map<string, info_persona> &d) {
   cout << d["Albert"].edat << " " << d["Albert"].pes << endl;
}
```

donarà un error de compilació tot i que en cap moment es modifica el contingut de d si "Albert"ja es troba a `d`. En canvi,

```
void consulta(map<string, info_persona> &d) {
   cout << d["Albert"].edat << " " << d["Albert"].pes << endl;
}
```

no dóna cap error de compilació.

3.2 Paràmetres addicionals de la classe

En un recorregut amb iteradors a partir del `begin()` d'un `map` l'ordre per defecte és creixent pel valor de la clau. Es pot canviar el criteri d'ordenació amb un paràmetre addicional de la classe. Per exemple

```
map<string, info_persona, greater<string> > d;
```

produeix una ordenació decreixent. Una ordenació creixent es pot indicar amb

```
map<string, info_persona, less<string> > d;
```

o simplement

```
map<string, info_persona> d;
```

`greater<string>` i `less<string>` ens donen funcions de comparació predefinides. Si la clau és un tipus predefinit `T` (`int`, `double`, `string`,..., C++ ens dona `less<T>` i `greater<T>` per poder determinar l'ordenació adient en un `map`.

Suposem que hem definit un tipus `Data` de la forma següent:

```
struct Data {
  int dia;
  int mes;
  int any;
};
```

que ens servira per associar enters a dates per saber el nombre de visitants d'un museu en una data determinada mitjançant un `map`. Per tal que el map estigui ordenat correctament per data cal sobrebrecarregar l'operador $<$, per tal que pugui comparar objectes de tipus data. Això s'aconsegueix afegint un mètode a `Data` de nom `operator<`.

Vegem-ho en un exemple tot junt:

```
#include <map>
#include <iostream>

using namespace std;

struct Data {
  int dia;
  int mes;
  int any;
  bool operator<(const Data &d) const;
};

bool Data::operator<(const Data &d) const {
  if (any != d.any) return any < d.any;
  else if (mes != d.mes) return mes < d.mes;
  else return dia < d.dia;
}

int main() {
  map<Data, int> visitants;
  Data data1, data2, data3;
  data1.dia = 1;
  data1.mes = 1;
```

```
  data1.any = 2015;

  data2.dia = 2;
  data2.mes = 1;
  data2.any = 2015;

  data3.dia = 8;
  data3.mes = 1;
  data3.any = 2015;

  visitants[data2] = 142;
  visitants[data3] = 21;
  visitants[data1] = 0; // es un dia festiu

  for (map<Data, int>::const_iterator i = visitants.begin();
    i != visitants.end(); ++i) {
    cout << i->first.dia << "/" << i->first.mes << "/" << i->first.any;
    cout << ": " << i->second << endl;
  }
}
```

3.3 Fites inferiors i superiors

Hem vist que el mètode `find` ens permet obtenir una referència a un parell clau-valor. Si la clau cercada no es troba al map, `find` ens retorna l'`end()`. Ens pot interessar obtenir un referència al primer element que sigui igual o més gran que el cercat. Això és el que ens permet obtenir el mètode `lower_bound`. Per exemple, per tal de saber el nombre de visitants del dia de reis en endavant, podem substituir el `for` de la Secció 3.2 de l'exemple anterior per

```
Data reis;
reis.dia = 6;
reis.mes = 1;
reis.any = 2015;
for (map<Data, int>::const_iterator i = visitants.lower_bound(reis);
     i != visitants.end(); ++i) {
  cout << i->first.dia << "/" << i->first.mes << "/" << i->first.any;
  cout << ": " << i->second << endl;
}
```

També existeix un mètode `upper_bound`, que dóna una referència al primer element més gran que la clau subministrada.

3.4 Exercici: Freqüència de paraules (X34352) de la Llista *Sessió 7 (Jutge)*

4 Més informació sobre diccionaris

Una especificació de la classe `map` amb costos associats a cada operació:

`http://www.cplusplus.com/reference/map/map/`

Cal que pareu especial atenció a l'especificació de les operacions: `find`, `insert` i `erase`.

5 Annex: Ús de `pair`

Els `pair` permeten formar parells amb un valor de tipus `T1` i un valor de tipus `T2`. `T1` i `T2` poden ser el mateix tipus.

Noteu que `pair<T1,T2>` és equivalent a

```
struct pair {
  T1 first;
  T2 second;
};
```

o sigui, els dos camps d'un `pair` sempre es diuen `first` i `second`.

Els `pair` són útils perquè algunes funcions de la `STL` necessiten retornar 2 valors, i en aquest cas s'utilitzen `pair` per fer-ho. Per exemple acabem de veure que un element d'un diccionari és un `pair`.

A més, també van bé perquè tenen alguns operadors ja definits:

- operador d'igualtat == (component a component)
- operadors de comparació (lexicogràficament)

Es podem omplir els camps d'un `pair` a la declaració, per exemple:

```
pair<double, char> a(2.5, 'A');
```

anb un resultat equivalent a

```
pair<double, char> b;
  b.first = 3.5;
  b.second = 'B';
```

i també es poden modificar *simultàneament* els camps d'un `pair` ja existent:

```
a = make_pair(2.25, 'A');
```

Per exemple, amb els valors actuals

```
cout << (a == make_pair(2.25, 'A')) << endl;
```

hauria d'escriure 1, donat que els valors del `pair a` i del `pair` anònim retornat per `make_pair` són iguals, i

```
cout << (a < b) << endl;
```

també hauria d'escriure 1 perquè el camp `first` d'`a` és més petit que el camp `first` del `pair` anònim retornat per `make_pair`.