Una petita introducció als conjunts

26 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

Un cop vistos els diccionaris o `maps` de `C++` veurem una estructura similar però una mica més senzilla: els conjunts o `sets`. Resumidament, els `conjunts` són diccionaris on els elements no són parelles *clau* i *valor*, sinó simplement *claus*.

Un conjunt és útil quan volem emmagatzemar plegats una sèrie d'elements no repetits i amb una certa volatilitat, és a dir, que es facin moltes insercions, esborrats i cerques. La major eficiència dels conjunts en aquestes tasques compensa l'esforç extra que ha de fer el `C++` per mantenir l'estructura del conjunt.

Si fem servir un diccionari en un programa i ens adonem que el *valor* de cada *clau* és irrellevant, llavors és convenient substituir-ho per un conjunt.

1.1 Com fer servir `sets`

Un exemple senzill per crear un `set`.

```
#include <iostream>
#include <set>
#include <string>

using namespace std;

int main()
{
  set<string> colors;
```

```
  colors.insert("verd");
  colors.insert("groc");
  colors.insert("taronja");
  colors.insert("blau");
  colors.insert("blanc");
  ...
}
```

Es pot veure que per insertar elements es fa servir l'operació `insert`. No cal indicar res, donat que com el conjunt està ordenat, igual que els diccionaris, l'operació `insert` posarà l'element on correspongui. Si s'intenta insertar un element ja existent, el conjunt no canvia.

Per recórrer un `set` es fan servir iteradors.

```
  set<string>::iterator it = colors.begin();
  while(it != colors.end() ){
    cout << *it << endl;
    ++it;
  }
```

Aquí tenim al versió amb `for`. Si el `set` és `const`, l'iterador ha de ser també `const`.

```
for (set<string>::const_iterator it = colors.begin(); it != colors.end(); ++it) {
  cout << *it << endl;
}
```

L'`insert` por tornar un parell amb un iterador i un booleà. El booleà ens diu si l'element s'ha afegit o bé ja hi era. L'iterador apunta a l'element del `set` sigui que s'acaba d'afegir o que ja estava abans. Continuant amb l'exemple, podem declarar:

```
pair<set<string>::iterator,bool> res;
res = colors.insert("groc");
```

llavors `res.second` serà fals perqué `groc` ja hi era i `res.first` apuntarà a l'element `groc` al `set`.

Hi ha una altra versió de l'`insert` que a més té un iterador. Pot ser útil si l'iterador assenyala un element a prop d'on s'hauria d'insertar l'element nou, perque fa la inserció més ràpida, però ha de quedar clar que el punt d'inserció és únic perque el `set` està ordenat.

Altres mètodes útils són `size()` que ens permet saber la mida del `set` (el nombre d'elements emmagatzemats), `empty()` que ens permet saber si és buit i `clear()` que buida el `set`.

Per veure si un element està al `set` o no, es pot fer servir `find` que torna un iterador a l'element buscat si hi és o a l'element fictici `end()` si no hi és. Per exemple:

```
if (colors.find("fucsia") != colors.end()) {
  cout << "El color fucsia es al conjunt" << endl;
}
else {
  cout << "El color fucsia no hi es" << endl;
}
```

Per esborrar elements del `set` es fa servir `erase`. Es pot dir quin element es vol esborrar i l'`erase` ens torna un valor enter que diu quants elements ha esborrat. En el cas de `sets` el valor només pot ser 1 o 0 naturalment. Una altra possibilitat és esborrar l'element assenyalat per un iterador; en `C++11` aquesta 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.

```
  set<string>::iterator it;

  it = colors.begin();
  ++it;
  colors.erase (it);
  it=colors.erase (it); // només c++11
  colors.erase ("blanc");
  int i = colors.erase("groc");
```

Hi ha més maneres de fer servir `erase`, però amb aquestes dues en tindrem prou.

Cal remarcar els elements d'un `set` no es poden modificar amb l'operador de desreferenciació aplicat a un iterador, si de cas, s'ha d'esborrar l'element i insertar un altre.

1.2 Comparació d'elements

Els elements d'un `set` s'inserten en un determinat ordre, que si no s'indica altra cosa és l'ordre creixent, però si el tipus dels elements no és estàndard o volem un altra ordenació ho hem d'indicar.

Podem declarar una funció booleana de comparació que serveixi per ordenar els elements del set, que ha de complir unes certes restriccions: ha de ser irreflexiva (el resultat serà fals si es comparen dos elements iguals), ha de ser antisimètrica i ha de ser transitiva.

Es pot canviar el criteri d'ordenació amb un paràmetre addicional de la classe. Per exemple

```
set<string, greater<string> > colors;
```

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

```
set<string, less<string> > colors;
```

o simplement

```
set<string> colors;
```

`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 `set`.

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

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

que ens servirà per fer un conjunt de dates. Per poder ordenar el conjunt correctament cal sobrebrecarregar l'operador $<$, per tal que pugui comparar objectes de tipus Data. Això s'aconsegueix amb un mètode a `Data` de nom `operator<`, la capçalera del qual s'ha d'afegir a la definició de l'`struct`. Es pot posar el codi dintre de l'`struct` o fora tal i com fem aquí:

```
bool Data::operator<(const Data &d) const
// el resultat és cert si el p.i, és menor que d;
// fals en cas contrari
{
  if (any != d.any) return any < d.any;
  else if (mes != d.mes) return mes < d.mes;
  else return dia < d.dia;
}
```

És important remarcar que s'ha d'implementar $<$, és a dir, *estrictament menor*. Si s'implementa *menor o igual* els resultats de l'ordenació poden no ser correctes perqué $\leq$ no és irreflexiva ni antisimètrica.

Per últim, si tenim una classe, com ara `BinTree`, sense un ordre definit per defecte i no podem dotar-li de l'`operator<` de la manera anterior, podem definir-lo de forma que no pertany a cap clase:

```
bool operator<(const BinTree<int> &a1, const BinTree<int> &a2)
// el resultat és cert si a1, és menor que a2;
// fals en cas contrari
```

Podeu veure això en acció al fitxer `orden_set_bintree.cc` de la carpeta de la sessió.

1.3 Fites inferiors i superiors

Hem vist que el mètode `find` ens permet obtenir un iterador a un element. Si aquest no es troba al `set`, `find` ens retorna una referència l'element fictici `end()`. Ens pot interessar obtenir un iterador al primer element que sigui igual o al primer vagi darrera en l'ordre. Això ho podem fer amb el mètode `lower_bound`. Noteu que en el cas de l'ordenació habitual (<), el primer element que va darrera és el primer element estrictament més gran.

També existeix un mètode `upper_bound`, que dóna un iterador al primer element que vagi darrera d'un determinat element.

1.4 Exercici: Activitats esportives (X83904) de la Llista *Sessió 7 (Jutge)*

2 Més informació

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

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

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

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