INF280 : Projet de programmation : Problèmes pratiques et concours

Conteneurs de la bibliothèque standard C++

Pierre Senellart, Antoine Amarilli

21 février 2017

Collections

- Regroupent un nombre arbitraire d'éléments de même type
- La complexité des opérations varie suivant la collection :
  - accéder au n-ième élément
  - rechercher un élément
  - ajouter un élément à une position arbitraire
  - supprimer un élément
  - parcourir les éléments dans l'ordre
- Cas particulier, tableaux associatifs : peuvent être vus comme collections de paires (clé, valeur)
- Choisir la collection adaptée pour chaque problème

Collections en C++

- Pas d'héritage, C++ utilise le mécanisme de la programmation générique (templates) pour permettre à des fonctions de manipuler des collections diverses
- Toutes les collections sont dans l'espace de nom `std` (ou `std::tr1` pour `unordered_set`, `unordered_map`, etc., sur les compilateurs pré-C++ 2011)
- Toujours écrire `using namespace std;`
- Certaines collections (`stack`, `queue`, `priority_queue`) sont des adapteurs qui peuvent utiliser plusieurs classes (`list`, `vector`...) comme implémentation

Itérer sur une collection (C++ 2011)

```
for(auto t : collection) {
  /* Faire quelque chose avec t */
}
```

Utiliser `auto &` pour modifier la collection. Par exemple :

```
vector<int> v = {1, 2, 3};

for(auto& t : v) {
  t++;
}

// va afficher 2, 3, et 4
for(auto t : v) {
  printf("%d\n", t);
}
```

Itérer sur une collection (C++ 1998)

```
for(collection_type::const_iterator
      it=collection.begin(),itend=collection.end();
      it!=itend;
      ++it) {
  value_type v=*it;
  /* Faire quelque chose avec v */
}
```

Remplacer `const_iterator` par `iterator` pour modifier.

Algorithmes de la bibliothèque standard

- Algorithmes polymorphes dans la bibliothèque standard : s'appliquent à des collections quel que soit leur type
- Écrire `#include <algorithm>`
- Ex : le tri prend en argument un itérateur vers le début et la fin :

```
std::sort(collection.begin(),collection.end());
```

- Beaucoup d'algorithmes disponibles : tri, recherche, recherche dichotomique, copie, inversion, remplissage, énumération des permutations d'un ensemble…

Tableaux de taille fixe

- `value_type[size]`
- Taille fixée à la compilation
- Il existe aussi une classe `array<value_type,size>` en C++ 2011, plus homogène avec les autres collections, et rendant certaines interactions avec la bibliothèque standard plus pratiques
- Occupation mémoire minimale (on ne peut pas faire mieux)
- Accès à un élément arbitraire : $O(1)$
- Pas de suppression, pas d'ajout d'élément
- Si l'efficacité est critique, initialiser avec memset :
  `memset(t, 0, sizeof(value_type) * size)`
  (plus efficace qu'une boucle)

Tableaux de taille variable

- `vector<value_type>`
- Occupe jusqu'à 2 fois plus d'espace qu'un tableau de taille fixe
- Accès à un élément arbitraire : $O(1)$
- Ajout/suppression à la fin du tableau : $O(1)$ amorti
- Ajout/suppression à un autre endroit : $O(n)$
- Pratique pour indexer des éléments par des entiers quand il n'y a pas (ou peu) de « trous » dans les entiers indexant
- Aussi `deque` pour ajout/suppression en temps constant à la fois au début et à la fin du tableau (mais non-contigu en mémoire)

Liste doublement chaînée

- `list<value_type>`
- Accès au premier ou dernier élement : $O(1)$
- Accès à l'élement suivant ou précédent : $O(1)$
- Accès à un élément arbitraire : $O(n)$
- Ajout/suppression à un endroit arbitraire : $O(1)$
- Version simplement chaînée : `forward_list`

Paires

- `pair<type1, type2>`
- `make_pair(a, b)` pour construire une paire
- `p.first` et `p.second` comme accesseurs
- Ordre lexicographique
- Pour le changer, définir un opérateur :

```
using namespace std;
typedef pair<int, int> pii;
bool operator< (pii a, pii b) {
  return a.first * b.second - b.first * a.second;
}
```

Arbre de recherche équilibré (ensemble)

- `set<value_type>`
- Les éléments sont parcourus dans l'ordre :
  naturel, ou défini par un comparateur (cf slide précédente)
- Accès à un élément : $O(\log(n))$
- Ajout/suppression d'un élément arbitraire : $O(\log(n))$
- Aussi `multiset` : permet d'avoir plusieurs fois le même élément

Table de hachage (ensemble)

- `unordered_set<value_type>` en C++ 2011
- Dans l'espace de noms `std::tr1` en C++ pré-2011
- Les éléments sont parcourus dans un ordre arbitraire (celui de la fonction de hachage)
- Accès à un élément : $O(1)$ amorti
- Ajout/suppression d'un élément arbitraire : $O(1)$ amorti
- Aussi `unordered_multiset`

Pile (Last In First Out)

- `stack<value_type>`
- Accès au premier élement : $O(1)$
- Ajout/suppression au début : $O(1)$
- Accès à un autre élément ou ajout/suppression à un autre endroit impossibles

File (First In First Out)

- `queue<value_type>`
- Accès au premier élémént : $O(1)$
- Suppression du premier élément : $O(1)$
- Ajout à la fin : $O(1)$
- Accès à un autre élément ou ajout/suppression à un autre endroit impossibles

File de priorité

- `priority_queue<value_type>`
- Les éléments sont insérés et gardés triés dans la file de priorité, par leur ordre naturel ou par un ordre de priorité défini par un comparateur ; deux éléments peuvent avoir la même priorité
- Accès à l'élément de plus haute priorité : $O(1)$
- Suppression de l'élément de plus haute priorité : $O(\log(n))$
- Ajout d'un élément : $O(\log(n))$
- Accès à un autre élément ou suppression à un autre endroit impossibles
- Modification dynamique des priorités impossible

Arbre de recherche équilibré (tableau associatif)

- Pour stocker des paires (clé, valeur)
- `map<key_type, value_type>`
- Les éléments peuvent être parcourus dans l'ordre de leur clé (ordre naturel, ou ordre défini par un comparateur)
- Accès à un élément par sa clé : $O(\log(n))$
- Ajout/suppression d'un élément arbitraire par sa clé : $O(\log(n))$
- Aussi `multimap` : permet d'avoir plusieurs fois le même élément

Table de hachage (tableau associatif)

- `unordered_map<key_type, value_type>` en C++ 2011
- Dans l'espace de noms `std::tr1` en C++ pré-2011
- Les éléments sont parcourus dans un ordre arbitraire (celui de la fonction de hachage)
- Accès à un élément par sa clé : $O(1)$ amorti
- Ajout/suppression d'un élément arbitraire par sa clé : $O(1)$ amorti
- Aussi `unordered_multimap`