Ассоциативные контейнеры

Введение в ассоциативные контейнеры

Для реализации последовательных контейнеров (массивов, векторов, двусторонних очередей и списков) используются массивы и списки. Кроме этого применяются *сбалансированные деревья,* предназначенные для их эффективного хранения и извлечения.

Сбалансированные деревья **составляют основу для другой группы контейнеров, определенной в STL, так называемых *(сортированных) ассоциативных контейнеров.***

Сбалансированные деревья

Бинарное **дерево называется** ***сбалансированным*** **или** ***АВЛ-деревом*****, если для любой вершины дерева, высоты левого и правого поддеревьев отличаются не более чем на единицу. Показатель сбалансированности бинарного дерева, равный +1, 0, -1,означает соответственно: правое поддерево выше, они равной высоты, левое поддерево выше.**

М. Адельсон-Вельский и Е.М. Ландис доказали, что при таком определении можно написать программы добавления/удаления, имеющие ***логарифмическую сложность*** **и сохраняющие дерево сбалансированным.**

Типы ассоциативных контейнеров

Всего существует 5 типов этих контейнеров:

- **множества *(sets),***
- **множества с дубликатами *(multisets),***
- **словари *(maps),***
- **словари с дубликатами *(multimaps),***
- **битовые множества *(bitset).***

Множества – sets

Каждый элемент *множества* является собственным ключом, и эти ключи уникальны. Поэтому два различных элемента множества не могут совпадать. Например, множество может состоять из следующих элементов:

123 124 800 950

Множества и словари

Множества с дубликатами – multisets

Множество с дубликатами **отличается от просто множества только тем, что способно содержать несколько совпадающих элементов.**

123 123 800 950

Словари – maps

Каждый элемент *словаря* имеет несколько членов, один из которых является ключом. В словаре не может быть двух одинаковых ключей.

123	**John**
124	**Mary**
800	**Alexander**
950	**Jim**

Множества и словари

Словари с дубликатами – multimaps

Словарь с дубликатами **отличается от просто словаря тем, что в нем разрешены повторяющиеся ключи.**

123 John

123 Mary

800 Alexander

950 Jim

В отличие от последовательных контейнеров ассоциативные контейнеры хранят свои элементы *отсортированными,* вне зависимости от того, каким образом они были добавлены.

Примеры

```
// set.cpp: Два идентичных множества,
// созданных разными способами.
#include <iostream>
#include <set>
using namespace  std;
int set1()
{    set<int, less<int> > S, T;
S.insert(10);  S.insert(20);  S.insert(30); S.insert (10);
T.insert (20);  T.insert (30);  T.insert (10);
if (S == T) cout << "Equal sets, containing:\n";
for (set<int,  less<int> >::iterator i = T.begin();
      i != T.end(); i++)
      cout << *i << " ";
cout << endl;
return 0;
}
```

```
// Результат:
Equal sets, containing:
10 20 30
```

Замечания

Порядок чисел 20, 30, 10, в котором были добавлены элементы *T*, несущественен; равным образом множество *S* не изменяет добавление элемента 10 во второй раз.

Ключи уникальны во множествах, но могут повторяться во множествах с дубликатами.

Определение *S* и *T:*

```
set<int, less<int> > S, T;  // > > разделены пробелом
```

Предикат less<int> требуется для определения упорядочения значения выражения $k_1 < k_2$,

где k_1 и k_2 являются ключами.

Хотя множества и не являются последовательностями, мы можем применять к ним итераторы и функции *begin()* и *end().* Данные итераторы являются *двунаправленными.*

Таблица операций, применимых к итераторам

x - переменная того же типа, что и элементы рассматриваемого контейнера, а n – int.

Категория итератора	*Операции (дополнительно к* i == j, i !=j, i =j)	Какие контейнеры предоставляют	Каким алгоритмом используется
входной (input)	x = *i, ++i, i++	все четыре	*find*
выходной (output)	*i =x, ++i, i++	все четыре	*copy* (приемник)
прямой (forward)	как у входного и выходного сразу	все четыре	*replace*
двунаправленный (bidirectional)	как у прямого и --i, i--	все четыре	*reverse*
произвольного доступа (random access)	как у двунаправленного и i + n, i - n, i +=n, i -= n, i<j, i>j, i<=j, i>= j	массив, *vector*, *deque* (*no* не *list*)	*sort*

```
// multiset.cpp:  Два множества  с дубликатами.
#include <iostream>
#include <set>
using namespace  std;
int multiset1()
{   multiset<int, less<int> > S, T;
S.insert(10); S.insert(20); S.insert(30); S.insert(10);
T.insert(20); T.insert(30); T.insert(10);
if (S == T) cout << "Equal multisets: \n";
else cout << "Unequal multisets:\n";
cout << "S:  ";
copy (S.begin() , S.end(),
   ostream_iterator<int>(cout, " "));
cout << endl; cout << "T:  ";
copy (T.begin() , T.end(),
   ostream_iterator<int>(cout, " "));
cout << endl;
return 0;
}
```

```
// Вывод:
Unequal multisets:
S: 10 10 20 30
T: 10 20 30
```

Примеры работы со словарями

Происхождение термина «ассоциативный контейнер» становится ясным, когда начинаем рассматривать словари. Например, телефонный справочник связывает (*ассоциирует*) имена с номерами. Имея заданное имя или *ключ*, нужно узнать соответствующий номер. Т.е., телефонная книга является *отображением имен на числа.*

Если имя *Johnson, J.* соответствует номеру *12345*, STL позволяет определить словарь *D,* -> D["Johnson, J."] = 12345;

Это означает:

***"Johnson, J."* -> 12345**

```
// mapl.cpp:  Первая программа со словарями.
#include <iostream>
#include <string>
#include <map>
using namespace  std;
// Создадим функциональный объект:
class  compare2
{
public:
bool operator()(const char *s, const char *t) const
{   return strcmp (s, t) <  0;
}
};
```

```
int map1()
{   map<char*, long, compare2> D;
D["Johnson, J."]   =  12345;
D["Smith, P."]     =  54321;
D["Shaw, A."]      =  99999;
D["Atherton, K."]  =  11111;
char GivenName [30] ;
cout  <<  "Enter a name:  ";
cin.get(GivenName,  30);
if  (D.find (GivenName) != D.end())
     cout << "The number is " << D[GivenName];
else cout  <<  "Not  found.";
cout  <<  endl;
return  0;
}
```

Замечания

Программа *map1.cpp* содержит определенный нами функциональный объект *compare2.*

Определение *map<char*, long, compare2> D;*

справочника D содержит следующие параметры шаблона:

- **тип ключа char*;**
- **тип сопутствующих данных long;**
- **класс функционального объекта compare2.**

Функция-член *operator()* класса *compare2* определяет отношение меньше для ключей.

Примеры: словари с дубликатами

```
// multimap1.cpp: Множество с дубликатами,
// содержащее одинаковые ключи.
#include <iostream>
#include <string>
#include <map>
using namespace std;
class compare3
{
public:
bool operator() (const char *s, const char *t) const
{  return strcmp(s, t) < 0;
}
};
```

```
typedef multimap<char*, long, compare3> mmtype;
int multimap1()
{   mmtype D;
D.insert(mmtype::value_type("Johnson, J.",  12345));
D.insert(mmtype::value_type("Smith, P.",  54321));
D.insert(mmtype::value_type("Johnson, J.",  10000));
cout << "There are " << D.size() << " elements. \n";
return  0;
}
```

Программа выведет:

```
There are 3 elements.
```

Замечания

Оператор доступа по индексу [ ] *не определен для множеств с дубликатами*, поэтому нельзя добавить элемент, написав, к примеру:

D["Johnson, J."] = 12345;

Вместо этого напишем:

D.*insert* (mmtype::value_type ("Johnson, J.", 12345));

где *mmtype* на самом деле означает:

multimap<char*, long, compare3>

Так как идентификатор *value_type* определен внутри шаблонного класса *multimap*, перед *value_type* здесь требуется написать префикс *mmtype::*. Определение идентификатора *value_type* основано на шаблоне *pair*.

Алгоритмы работы с ассоциативными контейнерами

includes **– выполняет проверку *включения* одной последовательности в другую. Результат равен *true* в том случае, когда каждый элемент первой последовательности содержится во второй последовательности.**

set_intersection **– создаёт отсортированное *пересечение множеств*, то есть множество, содержащее только те элементы, которые одновременно входят и в первое, и во второе множество.**

set_difference **– создание отсортированной последовательности элементов, входящих только в первую из двух последовательностей.**

Алгоритмы работы с ассоциативными контейнерами

set_union **– создает отсортированное *объединение множеств*, то есть множество, содержащее элементы первого и второго множества без повторяющихся элементов.**

Методы

begin() **– указывает на первый элемент,**

end() **– указывает на последний элемент,**

insert() **– для вставки элементов,**

erase() **– для удаления элементов,**

size() **– возвращает число элементов,**

empty() **– возвращает true, если контейнер пуст и др.**

```
int set_algorithm()
{
const int N = 5;
string s1[]= {"Bill", "Jessica", "Ben", "Mary",  Monica"};
string s2[N] = {"Sju","Monica","John","Bill","Sju"};
    typedef set<string> SetS;
    SetS A(s1,  s1 + N);
    SetS B(s2,  s2 + N);
    print(A); print(B);
    SetS prod, sum; // множества для результата
set_intersection (A.begin(), A.end(), B.begin(), B.end(),
                  inserter(prod, prod.begin()));
    print(prod);
```

```
// Продолжение
set_union (A.begin(), A.end(), B.begin(),B.end(),
     inserter (sum, sum.begin()));
print(sum);
if (includes (A.begin(), A.end(), prod.begin(),
prod.end()))
     cout << "Yes" << endl;
else cout <<"No" << endl;
return 0;
}
// Результат:
     Ben  Bill  Jessica  Mary  Monica // Множество A
     Bill  John  Monica  Sju              // Множество B
     // Пересечение set_intersection -> prod
     Bill  Monica
     // Объединение set_union  -> sum
     Ben  Bill  Jessica John  Mary  Monica  Sju          //
Включение includes можества prod в множество A
     Yes
```

Программа «Формирование частотного словаря»

Программа формирует частотный словарь появления отдельных слов в некотором тексте. Исходный текст читается из файла prose.txt, результат – частотный словарь – записывается в файл freq_map.txt.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <map>
#include <set>
#include <string>   …
int freq_map()
{char punct[6] = {'.', ',', '?', '!', ':', ';'};
set <char> punctuation(punct, punct + 6);
ifstream in("prose.txt");
if (!in)  { cerr << "File not found\n"; exit(1);}
```

```
map<string, int> wordCount;
string s;
while (in >> s)
{  int n = s.size();
   if (punctuation.count(s[n - 1]))
         s.erase(n - 1, n);
   ++wordCount[s];
}
ofstream out("freq_map.txt");
map<string, int>::const_iterator it =
wordCount.begin();
for (it; it  != wordCount.end(); ++it)
         out << setw(20) << left << it->first
         << setw(4) << right << it->second << endl;
cout <<"Rezalt in file freq_map.txt" << endl;
return 0;
```

Результат

Файл *prose.txt*:

«Рассмотрим, как работает эта программа. Программа подсчитывает сколько раз встречается каждое слово. Эта важная программа для изучения повторяемости различных слов.

Хорошая программа! Хорошая погода!»

Файл *freq_map.txt:*

Программа	1
Рассмотрим	1
Хорошая	2
Эта	1
важная	1
встречается	1
для	1
изучения	1
каждое	1
как	1
повторяемости	1
погода	1
подсчитывает	1
программа	3
работает	1
раз	1
различных	1
сколько	1
слов	1
слово	1
эта	1

Битовые множества *(bitset)*

Битовое множество – это шаблон для представления и обработки длинной последовательности битов. *bitset* – битовый массив, для которого определены операции произвольного доступа, изменения отдельных битов и всего массива. Биты нумеруются с 0.

Шаблон битового множества определён в заголовочном файле *<bitset>*.

Примеры создания битовых множеств:

```
bitset <100> b1;                          // сто нулей
bitset <16> b2 (0xf0f);                   // 0000111100001111
bitset <16> b3 (“0000111100001111”);
bitset <5> b4 (“00110011”, 3);            //10011
bitset <3> b5 (“00110101”, 1, 3);         //011
```

Первый параметр – строка из “0” и “1”. Второй параметр – позиция начала, третий – количество символов.

Constructors	создает новое битовое множество
Operators	сравнивают и устанавливают битовые множества
any	истина, если хотя бы один бит установлен
count	возвращает число установленных бит
flip	разворачивает битовое множество
none	истина, если ни один из битов не установлен
reset	устанавливает один или все биты в ноль
set	устанавливает один или все биты
size	количество битов, которое битовое множество может содержать
test	возвращает значение данного бита
to_string	строковое представление битового множества
to_ulong	возвращает целочисленное представление битового множества

Сортированные и хешированные ассоциативные контейнеры

К сортированным ассоциативным контейнерам относятся: *set, multiset, map, multimap.*

К хешированным: *hash_set, hash_multiset, hash_map, hash_multimap.*

Сортированные контейнеры соблюдают отношение порядка (ordering relation) для своих ключей. Сортированные контейнеры гарантируют *логарифмическую* эффективность большинства своих операций.

Это гораздо более сильная гарантия, чем та, которую предоставляют хешированные ассоциативные контейнеры. Последние гарантируют постоянную эффективность только в среднем, а в худшем случае – линейную.

Хешированные ассоциативные контейнеры

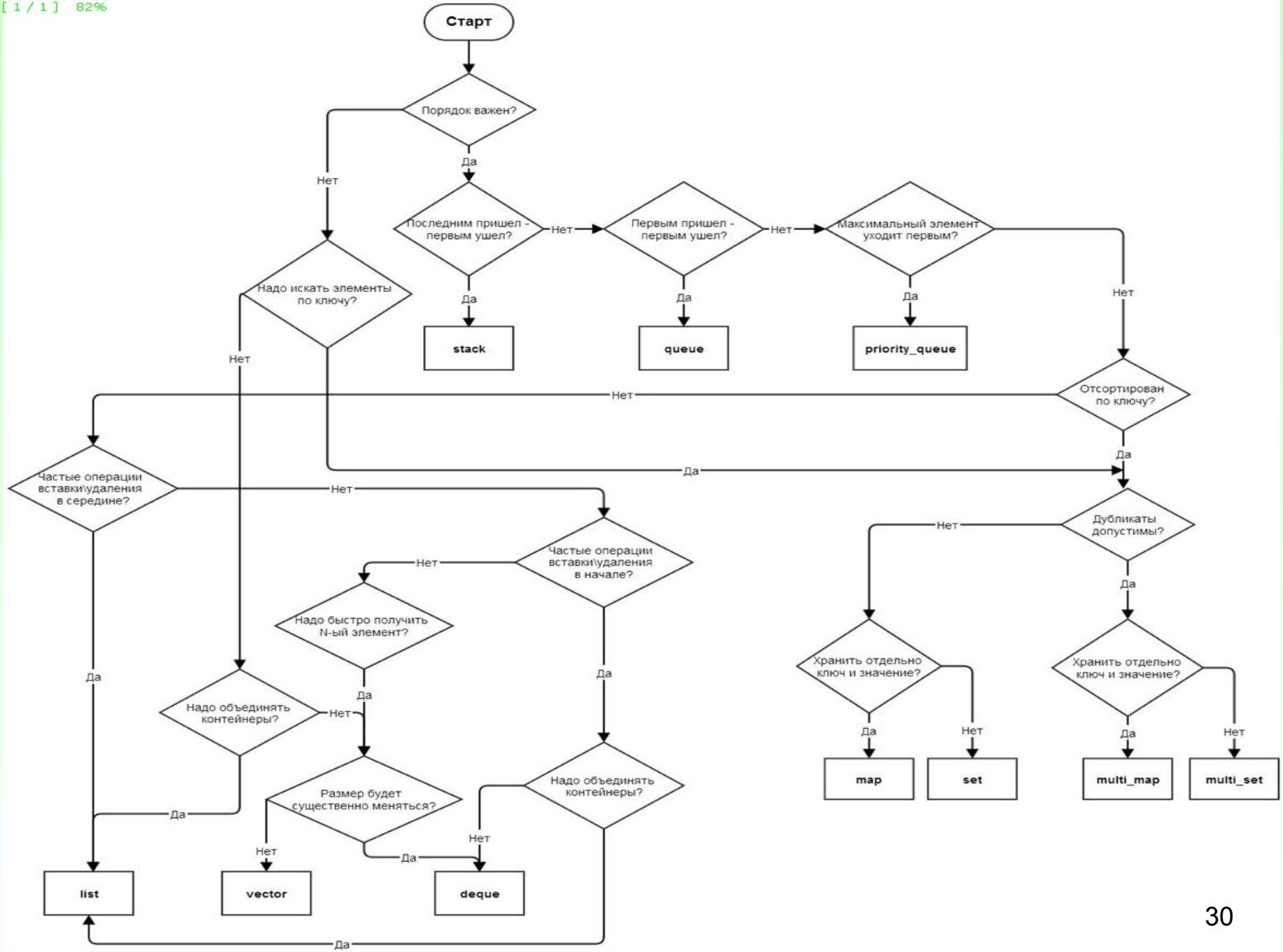
Хешированные ассоциативные контейнеры основаны на той или иной реализации *хэш-таблиц.*

Элементы в таком контейнере не упорядочены, хотя их можно добывать последовательно. Если вы вставите или удалите элемент, то последовательность оставшихся элементов может измениться.

Преимуществом хешированных ассоциативных контейнеров является то, что в среднем они значительно быстрее сортированных ассоциативных контейнеров.

Хешированные ассоциативные контейнеры

Удачно подобранная *функция хеширования* позволяет выполнять вставки, удаления и поиск за постоянное, не зависящее от *n*, время. Кроме того, она обеспечивает равномерное распределение хешированных значений и минимизирует количество коллизий.


Старт
Порядок важен?
Да
Нет
Последним пришел - первым ушел?
Нет
Первым пришел - первым ушел?
Нет
Максимальный элемент уходит первым?
Да
Нет
Да
Да
Да
stack
queue
priority_queue
Надо искать элементы по ключу?
Нет
Нет
Отсортирован по ключу?
Да
Да
Частые операции вставки\удаления в середине?
Нет
Дубликаты допустимы?
Нет
Да
Частые операции вставки\удаления в начале?
Нет
Надо быстро получить N-ый элемент?
Да
Да
Надо объединять контейнеры?
Нет
Да
Хранить отдельно ключ и значение?
Да
Нет
Хранить отдельно ключ и значение?
Да
Нет
Да
Размер будет существенно меняться?
Надо объединять контейнеры?
Нет
Да
Нет
Да
map
set
multi_map
multi_set
list
vector
deque
Да