Лекция 2. Структуры данных. Словари. Множества

Стандартная библиотека шаблонов STL. Ассоциативные контейнеры: множества, словари. Пары. Методы работы с множествами и словарями. Представление о бинарном дереве поиска. Примеры задач, решаемых с использованием структур set и multiset.

Общее представление

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Такие контейнеры построены на основе сбалансированных деревьев. К ассоциативным контейнерам относятся: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset). Для использования указанных контейнеров необходимо подключить заголовочные файлы: map, set, bitset.

Множество set

Множество состоит из элементов. Запись $x \in A$ означает, что x является элементом множества A. Примерами множеств являются множество натуральных чисел $N = \{1, 2, 3, \dots\}$, множество простых чисел $P = \{2, 3, 5, 7, \dots\}$. С математической точки зрения относительно каждого объекта можно лишь утверждать – принадлежит ли он множеству или не принадлежит, другие условия на элементы множества не накладываются. Элементы в множестве могут быть представлены в неупорядоченном виде, а также в множестве могут быть несколько экземпляров одного элемента. Элементами множества могут являться сложные объекты, например пары. Примером множества, состоящего из пар целых чисел, произведение которых равно 15, является множество {(5,3), (1,15), (-1, -15), …}. Множество может состоять и из геометрических объектов, например, множество окружностей на плоскости, множество прямых – касательных к заданной окружности. Особо выделяют пустое множество $\emptyset$ – множество, не содержащее ни одного элемента.

В языке программирования C++ контейнер **set** является множеством, в котором элементы упорядочены, и каждый элемент представлен в единственном экземпляре (нет повторяющихся элементов). Операции добавления, удаления и поиска в множестве выполняются за $O(\log n)$, где n – текущий размер множества. При объявлении множества указывают его имя и тип элементов, из которых множество будет состоять.

```
set<int> x; // Объявляется множество x целых чисел
```

Для работы с множеством используются следующие методы.

Метод	**Описание метода**
insert(value)	Добавляет элемент в множество
erase(value)	Удаляет элемент из множества
erase(it)	Удаляет элемент, соответствующий итератору it
find (it1, it2, value)	Возвращает итератор на элемент value в множестве. Если такого элемента нет, то возвращает end()

lower_bound(it1, it2, value)	Возвращает итератор на первый элемент в упорядоченном диапазоне, который имеет значение большее или равное val. Если такого элемента нет, то метод возвращает итератор it2.
	Пример.
	Вектор `v[8] = ( -1, -1, 1, 2, 2, 3, 3, 4 ).`
	`auto result = lower_bound(v.begin(), v.end(), 3);`
	`//*result=3`
	`auto result = lower_bound(v.begin(), v.end(), 5);`
	`//result=v.end()`

upper_bound(it1, it2, value)	Возвращает итератор на первый элемент в упорядоченном диапазоне, который имеет значение строго больше val. Если такого элемента нет, то метод возвращает итератор it2.
	Пример.
	Вектор `v[8] = ( -1, -1, 1, 2, 2, 3, 3, 4 ).`
	`auto result = upper_bound(v.begin(), v.end(), 3);`
	`//*result=4`
	`auto result = upper_bound(v.begin(), v.end(), 4);`
	`//result=v.end()`

Рассмотрим работу с множеством на примере ряда несложных задач.

Количество различных элементов последовательности

На координатную прямую по одной добавляют N точек. Нужно узнать, сколько среди них различных, после каждого добавления новой точки. Так как множество хранит уникальные элементы, то каждый раз при добавлении элемента, уже имеющегося в множестве, добавления на самом деле не происходит. Таким образом, количество элементов в множестве после добавления очередной точки будет являться ответом к задаче.

```
#include<iostream>
#include<set>
using namespace std;
int main()
{
	int n,x;
	set<int> a;
	cin >>n;
	for (int i=0;i<n;++i)
	{
		cin >>x;
		a.insert(x);
		cout<<a.size()<<endl;
	}
	return 0;
}
```

Заметим также, что на каждом шаге алгоритма мы получаем отсортированное множество точек.

Поиск элементов в множестве. Вывод элементов множества

Рассмотрим добавление точек на плоскость. Будем добавлять точки в множество точек **p**. Для оперирования точками удобно использовать структуру пар **pair**, позволяющую обрабатывать два объекта как один объект. При помощи множества реализуется добавление точек только в одном экземпляре. Заметим, что множество пар будет отсортировано. Пары сортируются сначала по первой координате, затем по второй. После заполнения множества точек поступает запрос на наличие точки с заданными координатами во множестве. Для ответа на вопрос используется метод **find**.

```
int main()
{
	int n,x,y;
	pair<int,int> c_p; //пара состоит из двух координат точки
	set<pair<int,int>> p; //множество точек
	cin>>n;
	for (int i=0;i<n;++i)
	{
		cin >>x>>y;		//вводятся координаты точек
		c_p=make_pair(x,y); //конструируется пара
		p.insert(c_p);	// точка добавляется в множество.
	}
	cin>>x>>y; //координаты точки,
	c_p=make_pair(x,y);
	if (p.find(c_p)!=p.end()) cout<<"YES"; else cout<<"NO";
	return 0;
}
```

Элементы множества можно вывести на экран. Это делается при помощи итератора, указывающего на объект множества. Поскольку итератор только указывает на элемент, то для печати на экран самого значения элемента необходимо разыменовать итератор при помощи операции «звездочка» *, которая записывается перед итератором.

```
	for (auto j=p.begin();j!=p.end();++j)
	{
		c_p=(*j);
		cout<<c_p.first<<" "<<c_p.second<<endl;
	}
	return 0;
```

Заметим, что элементы множества вывелись на экран в отсортированном (по возрастанию порядке). Для того, чтобы вывести элементы множества в отсортированном по убыванию порядке можно использовать методы доступа rbegin(), rend().

```
	for (auto j=p.rbegin();j!=p.rend();++j)
	{
		c_p=(*j);
		cout<<c_p.first<<" "<<c_p.second<<endl;
	}
```

Рейтинг объектов

При помощи множества удобно ранжировать объекты по какому либо признаку. Например, если в множестве хранить объекты пар: (средняя оценка учащегося, номер учащегося), то мы получим рейтинг учащихся. То есть, объекты будут отсортированы по средней оценке. Этот факт удобно использовать в ряде задач.

Будем добавлять в множество пары, состоящие из среднего балла ученика и его номера в журнале. Выведем на экран элементы множества – получим рейтинг учеников по среднему баллу.

```
#include<iostream>
#include <set>
using namespace std;
int main()
{
	int n;
	double x; // Средняя оценка ученика
```

```
    pair<double,int> c_p; //Пара – ср.оценка, номер по журналу
    set<pair<double,int>> p; //объявление множества
    cin>>n;
    for (int i=1;i<=n;++i) //Цикл ввода средних баллов
    {
        cin >>x;
        c_p=make_pair(x,i);
        p.insert(c_p);
    }

    for (auto j=p.rbegin();j!=p.rend();++j) //Вывод элементов
    {
        c_p=(*j);
        cout<<c_p.second<<" "<<c_p.first<<endl;
    }
    return 0;
}
```

В set все элементы последовательности записывались в одном экземпляре в отсортированном виде. Если использовать структуру множество для сортировки элементов, то необходимо, чтобы в множество можно было добавить все элементы, а не только уникальные. Такую логику работы с данными обеспечивает множество с дубликатами **multiset.**

Множество с дубликатами multiset

В языке программирования C++ контейнер **multiset** является множеством, в котором элементы упорядочены и могут храниться повторяющиеся элементы. Операции добавления, удаления и поиска, также как и в обычном множестве, в множестве с дубликатами выполняются за $O(\log n)$, где n – текущий размер multiset. Объявление множества с дубликатами осуществляется по тем же правилам, что и для обычного множества.

```
    multiset<int> x; //Объявляется мультимножество x целых чисел
```

Для работы с множеством используются такие же методы, что и для set.

С помощью множества с дубликатами удобно сортировать объекты – одинаковые объекты останутся в нужном количестве экземпляров в множестве в отсортированном виде. Используя структуру multiset и стандартные методы работы с ней, удобно отвечать на запрос – какое количество одинаковых элементов в множестве, равных заданному значению.

Количество одинаковых элементов, равных заданному значению

```
    int n,x,val_search,count=0;
    cin>>n;
    multiset<int> a;
    for (int i=0;i<n;++i){
        cin>>x;
        a.insert(x);   //заполнение множества с дубликатами
    }
    cin>>val_search;
for(auto i=a.lower_bound(val_search);i!=a.upper_bound(val_search); i++){
            count++;
    }
    cout<<count;
    return 0;
```

Рассмотрим работу программы на примере множества с дубликатами {1,1,2,2,2,3,3}. Вводится *val_search=2*. Программа определяет количество элементов множества, равных 2. На рисунке показано, на какой элемент указывает итератор a.lower_bound(2) – на первый элемент, больший или равный 2, и на какой элемент указывает итератор a.upper_bound(2) – на первый строго больший 2. Таким образом, программы выведет значение count=3.

1		2	2	2	3	3
		↑a.lower_bound(2)			↑a.upper_bound(2)	

Если в множестве нет элементов, равных 2, то указатели будут находится на первом элементе, большим 2.

1	1	3	3	3	3	4
					↑a.lower_bound(2)↑a.upper_bound(2)	

Заметим, что можно было получить значение количества элементов, равных val_search при помощи стандартного метода count().

```
cout<<a.count(val_searsch);
```

Использование предиката greater и собственных предикатов

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения и логических операций, определенных в языке C++. Они возвращают значения типа bool – их называют предикатами. По умолчанию элементы в множествах и словарях в STL упорядочены в порядке возрастания. Это обеспечивает по умолчанию предикат less. В явном виде можно использовать другой предикат, обеспечивающий сортировку элементов по убыванию – greater.

```
multiset<int,greater<int>> a;
```

Для использования предиката greater при создании множества (словаря) необходимо указать его после типа элементов множества. Для предиката указывается тип сравниваемых элементов.

Чтобы применить свой критерий сортировки, надо представить бинарный предикат в форме класса или структуры, реализующий оператор operator(). Ниже в примере приведен текст программы, которая заполняет множество ***a*** точками таким образом, чтобы они сортировались по убыванию расстояний от них до начала координат. Бинарный предикат представлен в форме класса Sort_set.

```
struct point{
          int x;
          int y;
};
class Sort_set{
     public:
          bool operator() (const point& p1,const point& p2){
               return (p1.x*p1.y>p2.x*p2.y);
          }
};
int main()
{
     set <point,Sort_set> a;
     int n;
     cin>>n;
     for(int   i=0;i<n;++i){
          int x_c,y_c;
          point p_c;
```

```
            cin>>x_c>>y_c;
            p_c.x=x_c; p_c.y=y_c;
            a.insert(p_c);
        }
        return 0;
}
```

Словари map

Словарь построен на основе пар значений. Первое значение пары – ключ для идентификации элементов, второе - собственно элемент. Например, в телефонном справочнике номеру телефона соответствует фамилия абонента. *В словарях элементы хранятся в отсортированном по ключу виде.* Поэтому для ключей должно быть определено отношение «меньше». В словаре map, в отличие от словаря с дубликатом (multimap) все ключи являются уникальными.

Рассмотрим задачу определения слова – синонима. Вначале работы программы заполняется словарь, в котором ключ будет являться первым словом, а синоним – вторым словом. Далее используется функция поиска элементов в словаре по ключу. Если поиск дал результат, то итератор будет указывать на найденный элемент словаря, если же поиск не дал результатов, то итератор будет указывать на end() словаря.

```
#include<iostream>
#include<map>
#include<string>
using namespace std;
int main()
{
    string word_first, word_second,word_find;
    map<string, string> voc;
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i){
        cin >> word_first >> word_second;
        voc[word_first] = word_second; }// Создание элемента map
    cin >> word_find;
    if (voc.find(word_find) != voc.end())//Поиск по ключу
        cout << voc[word_find];
    else
        for (auto c : voc){ //Проход по элементам map
            if (c.second == word_find) cout << c.first;}
    return 0;
}
```

Методы работы с map	
lower_bound(key)	Указывает на первый элемент, ключ которого равен или больше указанному ключу key или на end(), если такой ключ не найден
upper_bound(key)	Указывает на элемент, ключ которого больше указанного ключа key или end(), если такого нет.
find(key)	Возвращает итератор на найденный элемент
count(key)	Возвращает количество элементов, ключ которых равен key

Словарь map хранит элементы в упорядоченном по возрастанию ключа порядке. Для реализации другого порядка упорядочивания элементов необходимо использовать собственные предикаты или предикат greater.

Словари с дубликатами multimap допускают хранение элементов с одинаковыми ключами. Элементы с одинаковыми ключами хранятся в словаре в порядке их занесения в словарь. При удалении элементов удаляются все экземпляры элементов, соответствующие ключу.

Бинарное дерево поиска

Реализация set/map в STL основана на красно-черных деревьях, представляющих собой один из видов сбалансированных деревьев. Представление о деревьях и их использовании для реализации ассоциативных контейнеров дает бинарное дерево поиска (двоичное дерево поиска). Рассмотрим идею построения бинарного дерева поиска и работу с ним. Предварительно приведем основные понятия, необходимые для изучения бинарного дерева поиска.

Ориентированным деревом называется ориентированный граф без циклов, в котором входящие степени всех вершин, кроме одной, равны 1.

Единственная вершина, входящая степень которой равна 0 (в нее не входит ни одного ребра), называется *корнем дерева*. Вершины двоичного дерева, исходящая степень которых равна 0 (из них не выходит ни одного ребра) называются *листьями дерева*.

Ориентированное двоичное дерево – это ориентированное дерево, в котором исходящие степени вершин равны 0, 1 или 2.

Структура данных «двоичное дерево» - это множество записей вида (key, l, r), где key – ключ, l, r – указатели на две другие записи. Записи называются узлами дерева (tree nodes). Узлы , на которые указывают l и r называют левым и правым ребенком узла n=(key, l, r), причем ключ в любом узле больше или равен ключам во всех узлах левого поддерева этого узла и меньше или равен ключам во всех узлах правого поддерева этого узла. То есть, ключ родителя находится между ключами своего левого и правого ребенка: $n \to l \to key < n \to key < n \to r \to key$.

Указатели l и r могут быть пустыми, то есть равными специальной константе NULL. Совокупность узлов структуры данных «двоичное дерево» образует граф, в котором вершинами являются узлы, а ребрам соответствуют указатели l и r. Этот граф является ориентированным двоичным деревом.

Для обозначения бинарного дерева поиска используют аббревиатуру BST – binary search tree. Ниже на рисунке изображено двоичное дерево поиска, каждая вершина которого хранит ключ в виде целого числа. Также приведен массив чисел в порядке их добавления в бинарное дерево поиска.

Операции, поддерживаемые для структуры данных «двоичное дерево поиска»: добавление элементов, поиск элементов, удаление элементов. Эти операции довольно просты, и обычно реализуются рекурсивно.

1. **Операция добавить. insert** (value). Если дерево пусто, в h (корень дерева) заносится ссылка на новый узел, содержащий данный элемент. Если ключ добавляемого элемента меньше ключа в корне, то вызывается функция вставки элемента в левое поддерево, иначе вызывается функция вставки элемента в правое поддерево. Функция принимает дерево в качестве первого параметра и ключ вставляемого элемента в качестве второго. В том случае, если левое (правое) поддерево не содержит детей, то создаем ребенка: помещаем в него данные, и устанавливаем на него указатель.

2. Операция поиска. **find** (value). Если дерево пусто, то, очевидно, поиск неудачен. Если ключ поиска равен ключу в корне, то поиск успешен. Иначе выполняется рекурсивно поиск в соответствующем поддереве. Функция принимает дерево в качестве первого параметра и ключ в

качестве второго. Поиск завершает свою работу, либо когда текущее поддерево станет пустым (неудачный поиск), либо когда будет найден элемент с нужным ключом (успешный поиск).

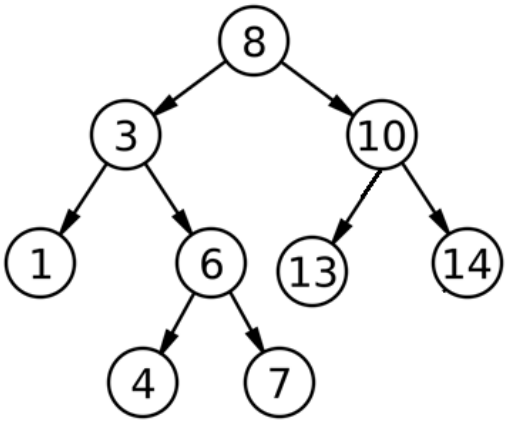

Рис.1. Бинарное дерево поиска

8	3	10	6	14	1	4	7	13

Приведем текст программы, в которой описывается структура node для построения двоичного дерева поиска. Реализуется функция добавления элементов в дерево insert(value), функция поиска элементов find(value).

```
#include<iostream>
using namespace std;

struct node{
     int key; //ключ узла
     node *l, *r; //ссылки на левое и правое поддеревья
     };
node *tree=NULL; //инициализация BST дерева tree
void insert(node*& t, int x){
          if (t==NULL){ //если дерево пустое
               t=new node; //выделили память
               t->key=x;  // добавили элемент x
               t->l=NULL; t->r=NULL;
               return;
          }
          if (x<t->key) // вставляемый элемент меньше ключа
               insert(t->l,x);//переход в левое поддерево
          else
               insert(t->r,x); //в противном случае в правое
     }
int find(node *t, int x){
          if (t==NULL) return 0; // дерево пустое
          if (x==t->key) return 1; //ключ совпал с искомым эл.
          if (x<t->key) //искомый элемент меньше ключа
               return find(t->l,x); //переход в левое поддерево
          else
               return find(t->r,x); //иначе в правое
     }
```

```
int main()
{
     int n;
     cin>>n; //количество элементов в дереве
     for (int i=0;i<n;++i){// цикл добавления элементов
          int a;
          cin>>a;
          insert(tree, a); ..
     }
     cout<<find(tree,100);// поиск элемента
return 0;
}
```

Чтобы разобраться в логике работы программы рекомендуем заполнить множество элементами массива, приведенного на рисунке 1, и изучить структуру полученного дерева при помощи окна «Контрольные значения» Visual Studio.

Оценка сложности операций, производимых на бинарном дереве поиска

Глубиной узла дерева n называется величина d(n), равная количеству ребер в пути от корня до этого узла. *Высотой дерева* H называется глубина самого глубокого листа. Рекурсивные процедуры добавления и поиска для двоичного дерева работают за время, ограниченное сверху высотой дерева. В худшем случае число шагов равно глубине самого глубокого узла, то есть высоте дерева, а в среднем – средней глубине узлов. Полное двоичное дерево – это дерево, у которого путь от корня до любого листа содержит H ребер. Если в таком дереве n узлов, то высота дерева оценивается как log n. Приведем оценку сложности операций на полном двоичном дереве поиска. Заметим, что в общем случае сложность операций зависит от высоты дерева.

Операция	Сложность. BST	Сложность. Отсортированный массив
Вставка элемента	O(log n)	O(n)
Поиск элемента по ключу	O(log n)	O(log n)
Другие операции с BST: получение отсортированного массива из BST за O(n), поиск порядковой статистики (k-я порядковая статистика – k-е по величине число в массиве) за O(log n), определение порядкового номера элемента за O(log n) (в отсортированном массиве, соответствующему элементам дерева).		

Реализация функции удаления элементов из бинарного дерева поиска зависит от типа удаляемого элемента в дереве:

- при удалении листьев очищается указатель из предка на данный лист;
- при удалении узла с одним потомком переписывается ссылка из предка удаляемого узла на потомка удаляемого узла (раньше она указывала на удаляемый узел);
- при удалении узла, у которого два потомка действуем по алгоритму: выбираем самый правый узел в левом поддереве удаляемого узла (максимальный элемент из элементов, меньших удаляемого узла), записываем его значение в удаляемый узел, удаляем найденный элемент.

Проиллюстрируем функцию удаления элемента 8 в бинарном дереве поиска на рисунке 2. Сначала находим самый правый элемент в левом поддереве элемента с ключом 8, это элемент с ключом 6. Заменяем ключ 8 в «удаляемом» элемента на ключ 6. Удаляем элемент с ключом 6 из дерева. Найденный элемент будет либо листом, либо будем иметь одного левого ребенка.

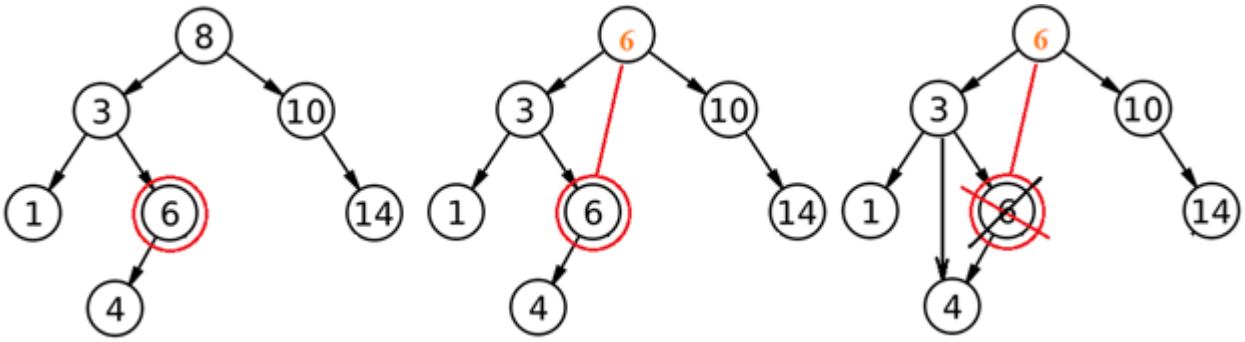

Рис.2. Удаление элемента key=8 в бинарном дереве поиска

Удаляя элементы, которые имеют двух детей, можно искать не самый правый элемент в левом поддереве, а самый левый элемент в правом поддереве.

Задание

1. Реализуйте функцию удаления элемента из бинарного дерева поиска erase(value).
2. Изобразите двоичное дерево поиска из элементов (перечислены в порядке их добавления в дерево): 10, 20, 30, 40, 50, 9, 8, 7, 6, 11, 12, 13. Вы получите пример «слабо ветвящегося» бинарного дерева поиска. Оцените его высоту.
3. Случайное двоичное дерево поиска размера n – это дерево, которое получается из пустого двоичного дерева, после добавления в него n записей с различными ключами в случайном порядке. (все n! возможных последовательностей добавления равновероятны). Получите рекуррентную последовательность средней высоты узлов дерева.
4. Напишите программу, реализующую телефонный справочник для абонентов при помощи структуры map. Предусмотрите функцию добавления новых абонентов в телефонный справочник, удаления абонентов, поиска фамилии абонента по его номеру.
5. Словарь map с ключами типа string имеет по умолчанию критерий сортировки на основании оператора <, определенного в классе string и применяемого заданным по умолчанию предикатом сортировки less. Предикат чувствителен к регистру. Одним из способов решения этой проблемы является создание предиката сортировки, возвращающего значение true или false в зависимости от результата сортировки без учета регистра: map<ключ, значение элемента, предикат>. Модифицируйте программу задания 4 таким образом, чтобы функции добавления и поиска не были чувствительны к регистру.
6. Проведите лексикографический анализ текста первой главы романа «Евгений Онегин». При помощи структуры map <string, int> получите данные о том, какие слова встречают в романе в каком количестве. Выведите эти слова в порядке убывания частотности их встречаемости в тексте.

Литература

А.Шень. Программирование: теоремы и задачи. – М.: МЦНМО, 2004. Второе издание, исправленное и дополненное.

Ворожцов А.В., Винокуров Н.А. Лекции «Алгоритмы: построение, анализ и реализация на языке программирования Си». – М.: Издательство МФТИ, 2007.

Курс сайта www.stepic.org «Введение в теоретическую информатику». Александр Шень.

Курс «Олимпиадные задачи по программированию». М.С. Густокашин. Электронный ресурс. http://prog.mathbaby.ru/dokuwiki/lib/exe/fetch.php?media=2015-9m:9m-binsearch-lection.pdf

Седжвик Р. Алгоритмы на C++. – М.: ООО «И.Д. Вильямс», 2014 .