Контейнеры

Обзор основных контейнеров STL

Полубенцева М.И.

Контейнеры

Контейнер – это объект, который **хранит** другие объекты и **контролирует их размещение** в памяти посредством конструкторов, деструкторов и методов вставки/удаления

Коллекция

- это объединение 0 или более элементов, предоставляющее интерфейс, посредством которого можно получать доступ к элементам в модифицирующем или не модифицирующем стиле

Состав STL

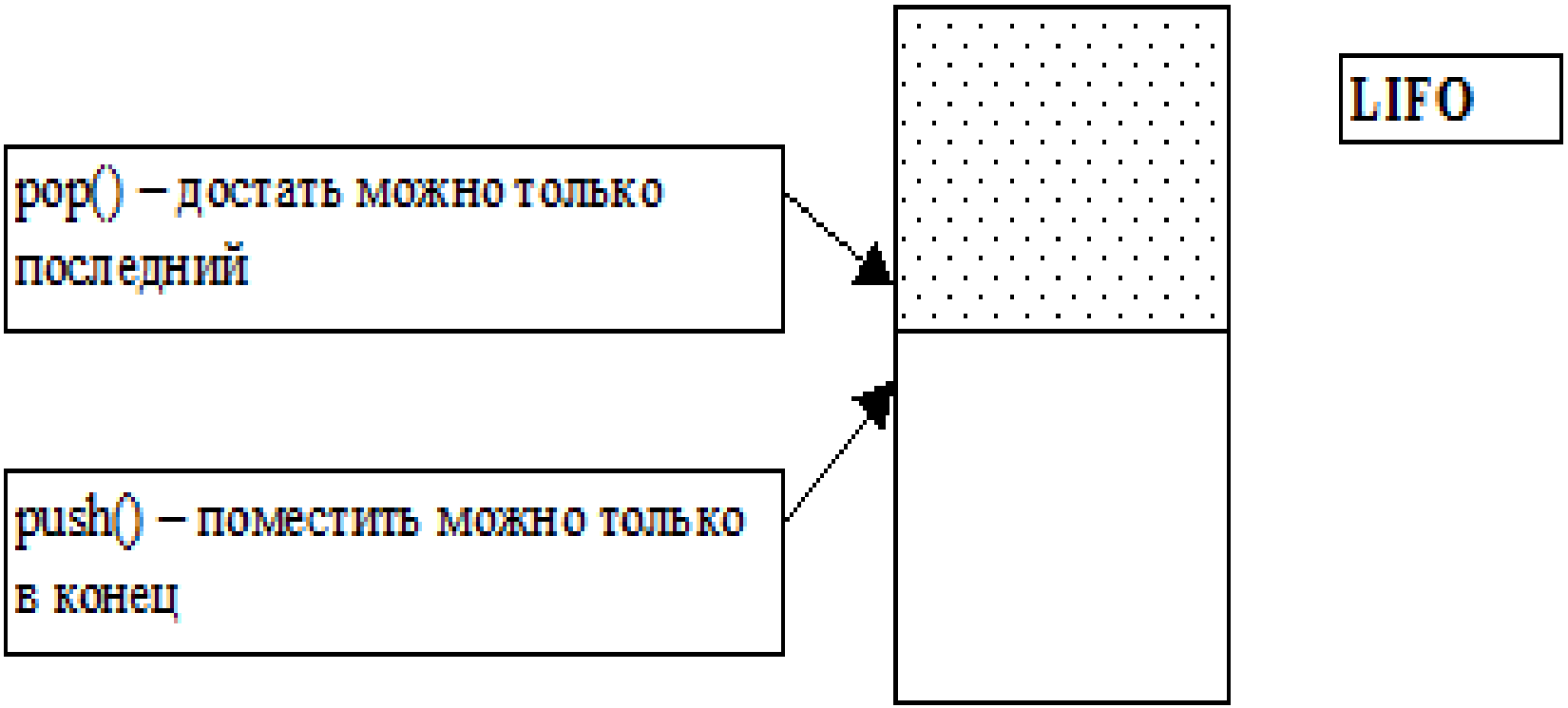
- Контейнеры
- Итераторы
- Аллокаторы
- Обобщенные алгоритмы
- Адаптеры
- Предикаты

Контейнеры

- **Последовательные:**
  - Базовые (vector, list, deque)
  - Адаптеры базовых (stack, queue, priority_queue)
- **Ассоциативные:**
  - set, multiset, map, multimap
  - **stdext::**hash_map, **stdext::**hash_multimap, **stdext::** hash_set, **stdext::** hash_multiset
  - **C++11** – unordered_set, unordered_multiset, unordered_map, unordered_multimap

АДАПТЕРЫ БАЗОВЫХ КОНТЕЙНЕРОВ

Стек. Назначение



Стек – интерфейс для любого базового контейнера.

Header - <stack>, namespace - std

template<class T, **class C = deque<T>** >

class stack{…};

1. Запретить все не стековые операции
2. Переопределить общепринятые для стека посредством базовой последовательности

Стек. Адаптер

pop_back() – уничтожает только последний	**pop()** – ничего не возвращает!!! =>
back() – значение последнего	для получения значения последнего элемента – **top()**
push_back() – добавляет только в конец	**`push()`**

stack

```
template<class T, class C = deque<T> > class std::stack{
protected: C c;
public:
        typedef typename C::value_type value_type;
        typedef C container_type;

        explicit stack(const C& a=C() ):c(a){};
        bool empty() const {return c.empty();}
        size_type size() const {return c.size();)

        value_type& top(){return c.back();}
        const value_type& top()const {return c.back();}

        void push(const value_type& x) {c.push_back(x);}
        void push(value_type&& x) {c.push_back(std::move(x));}

        void pop(){c.pop_back();}
};
```

stack

Извне **недоступны**:

- Методы базового контейнера
- Итераторы базового контейнера
- Псевдонимы базового контейнера

Пример

```
stack<int> s; //???
s.push(1);  //size==1
s.push(2);  //size==2
s.pop();      // size==1
int tmp = s.top();          //tmp==1, size==1
if(s.top() == 1) s.pop();
```

Пример

1.

vector<int> v(10,1);

//создать стек таким образом, чтобы его элементы стали копиями элементов вектора

2. Создать пустой стек на базе list

Пример

1. vector<int> v(10,1);
   stack<int, vector<int> > s2(v);

2. stack<int, list<int> > s3;

Пример

```
stack<int> s; //???
s.push(1);  //size==1
s.push(2);  //size==2
…
```

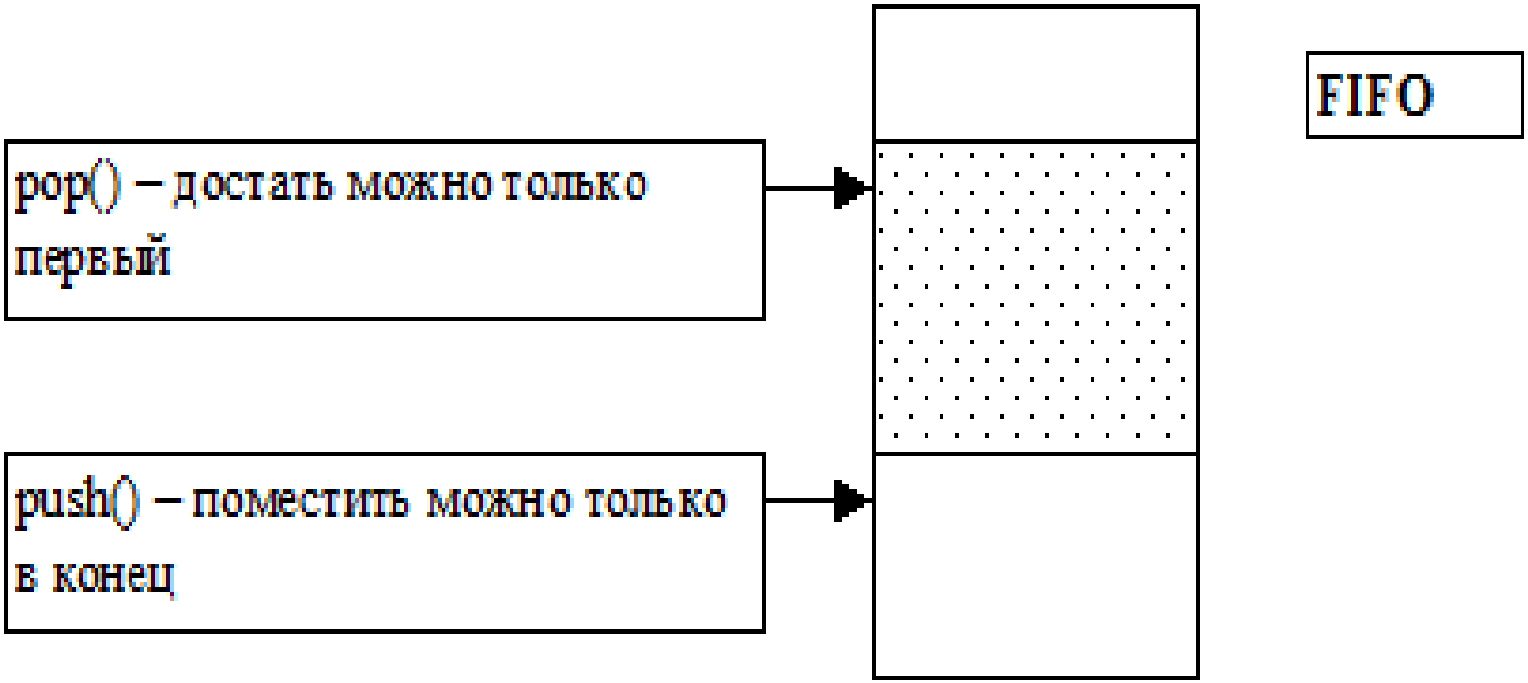
//Распечатать значения элементов стека:

Пример

```
stack<int> s;
s.push(1);
s.push(2);
…
while(s.size())  //или while(!s.empty())
{
        std::cout<<s.top();
        s.pop();
}
std::cout<<s.size(); //???
```

QUEUE

queue


pop() – достать можно только первый
push() – поместить можно только в конец
FIFO

queue

Можно использовать любой базовый контейнер???

queue

```
template<class T, class C = deque<T> > class std::queue{
protected:
        C c;
public:
        explicit queue (const C& a=C()):c(a){};
        value_type& front(){return c.front();}
        value_type& back(){return c.back();}
        bool empty() const {return c.empty();}
        size_type size() const {return c.size();)

        void push(const value_type& x) {c.push_back(x);}
        void pop(){c.pop_front();}
};
```

Пример

```
struct Message{…};

void Message_Loop(queue<Message>& q)
{
        while(!q.empty())
        {
                Message& msg = q.front();
                msg.service();
                q.pop();
        }
}
```

Пример

```
queue<string> q;
q.push("aaa");
q.push("b");
q.push("www");
q.push("dd");

//Требуется удалить заданную строку

string toDelete("b");
```

Пример

(шаблон)

```
queue<string> q;
//сформировали значения
string toDelete = …;
size_t n = q.size();
for(size_t i=0; i<n; i++)
{
        string & s = q.front();
         if(s!=toDelete)   {   q.push(std::move(s));  }
        q.pop();
}
```

PRIORITY_QUEUE

Специфика priority_queue

- на верхушке всегда максимальный элемент
- вставка – согласно частичному упорядочению
- удаляется всегда верхний элемент
- в качестве базового контейнера – контейнер с **произвольным** доступом

priority_queue – очереди с приоритетами

```
template<typename T,
        typename C = vector<T>,
        typename Cmp = less<T> >
class std::priority_queue{
protected:
        C c;
        Cmp cmp; //объект – для формирования критерия вставки
 …
};
```

Специфика

Добавляются конструкторы:

- **priority_queue**(const Traits& __Comp,
  const container_type& _Cont);
- **priority_queue**(const priority_queue& _Right);
- template<class InputIterator> **priority_queue** (InputIterator _First, InputIterator _Last);
- template<class InputIterator> **priority_queue** (InputIterator _First, InputIterator _Last, const Traits& __Comp);
- template<class InputIterator> **priority_queue** (InputIterator _First, InputIterator _Last, const Traits& __Comp, const container_type& _Cont);

Специфика

- Вставка и удаление элементов реализованы посредством std::make_heap(), std::push_heap() и std::pop_heap()
- Внедренный объект компаратора используется: if(cmp(<элемент очереди>, <вставляемое значение>))...
- Для критерия сравнения по умолчанию используется шаблон структуры std::less

Шаблон структуры less

Реализация???

less
#include <functional>

```
template<typename T> struct less{
 bool operator()(const T& x, const T& y)const
                { return x < y;}
};
```

Heap - сортировка

- реализуется посредством последовательной коллекции (массива)
- Представляет собой линеаризованное бинарное дерево

Heap - сортировка

Специфика:

- первый элемент всегда является максимальным => идеальная основа для реализации **очереди с приоритетами**;
- добавление и удаление элементов в общем случае производится с логарифмической сложностью

Алгоритмы STL для работы с кучами

- **make_heap()** преобразует интервал элементов в кучу;
- **push_heap()** добавляет новый элемент в кучу;
- **pop_heap()** удаляет элемент из кучи;
- **sort_heap()** преобразует кучу в упорядоченную коллекцию (которая после этого **перестает** быть кучей)

make_heap()

Преобразует последовательность [beg, end) в кучу (сложность линейная - не более **3*n** сравнений):

void make_heap (RandomAccessIterator beg,
RandomAccessIterator end);

void make_heap (RandomAccessIterator beg,
RandomAccessIterator end, BinaryPredicate op);

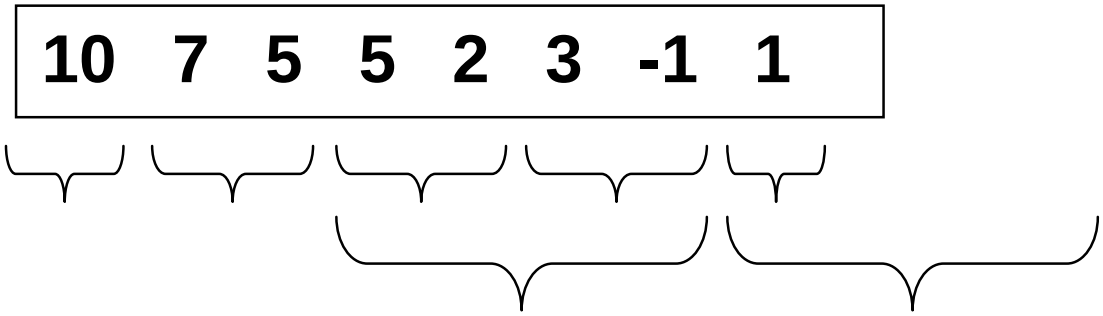
make_heap()

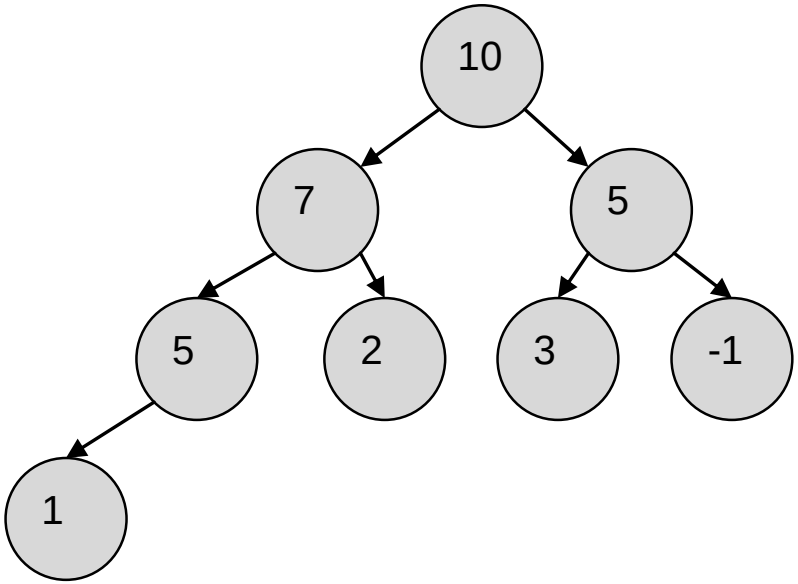
```
#include <algorithm>
{
        int ar[] = {3,5,-1,7,2,5,10,1}; //исходный массив
        size_t n = sizeof(ar)/sizeof(int);
        std::make_heap(ar, ar+n);
                        //10 7 5 5 2 3 -1 1  - куча
}
```

make_heap()

Значение каждого узла бинарного дерева меньше или равно значению родительского узла

make_heap()





push_heap

Добавляет последний элемент (находящийся перед end) в существующую кучу (в результате [begin, end) становится кучей.

Сложность логарифмическая (не более **log(numberOfElements)** сравнений)

- void **push_heap** (RandomAccessIterator beg, RandomAccessIterator end)
- void **push_heap** (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

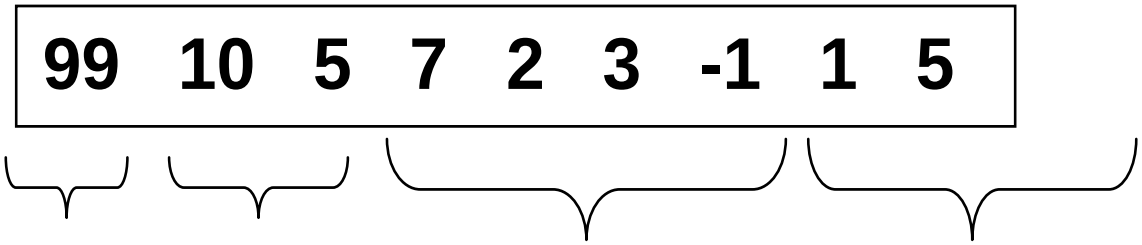
push_heap

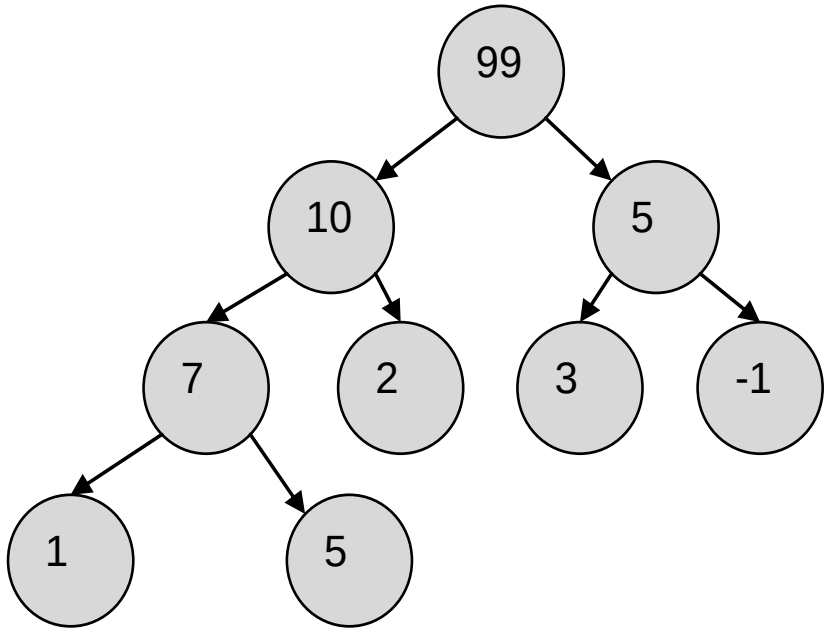
Алгоритм **push_heap()** переставляет элементы таким образом, что инвариант структуры бинарного дерева (**любой узел не больше своего родительского узла**) остается неизменным!

push_heap

```
int ar[] = {3, 5, -1, 7, 2, 5, 10, 1, 99};
int n = sizeof(ar)/sizeof(int);
std::make_heap(ar,ar+n-1);
                    //10 7 5 5 2 3 -1 1 99  - куча из n-1 элементов
std::push_heap(ar, ar+n);
                    //99 10 5 7 2 3 -1 1 5 - после push
```

push_heap





pop_heap

Перемещает максимальный (то есть первый) элемент кучи [beg,end) в конец и создают новую кучу из оставшихся элементов в интервале [beg,end-1). Сложность логарифмическая (не более 2*log(numberOfElements) сравнений)

- void **pop_heap** (RandomAccessIterator beg, RandomAccessIterator end)
- void **pop_heap** (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

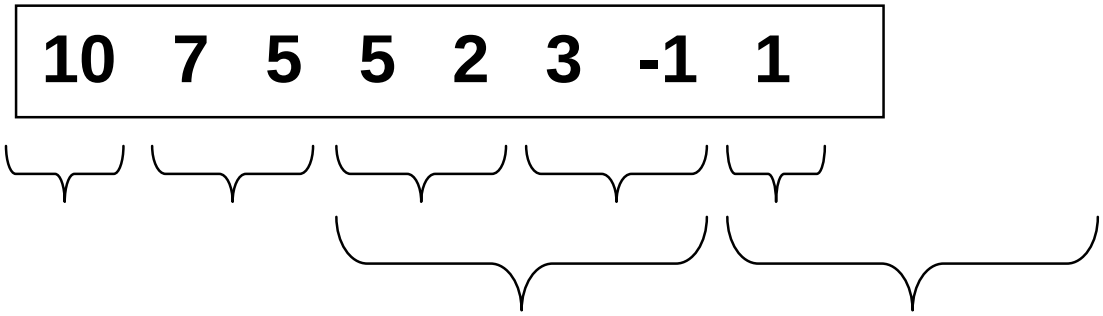
pop_heap

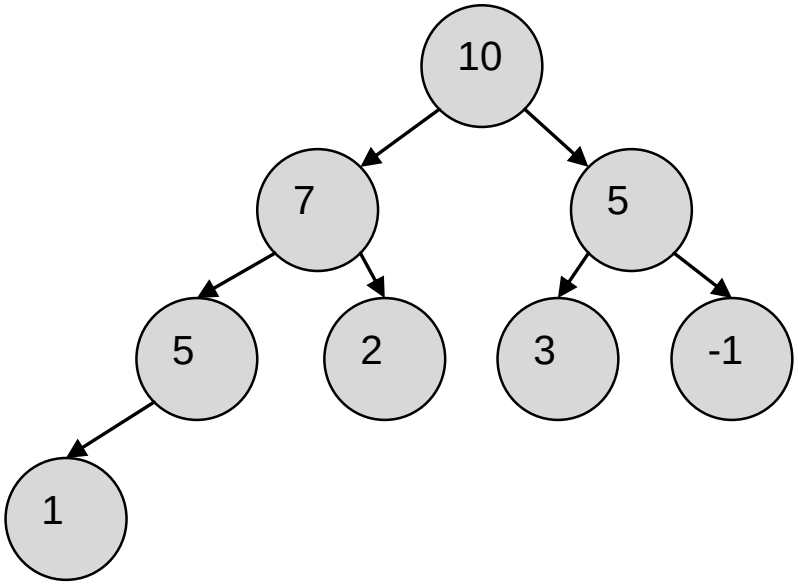
Алгоритм **pop_heap()** переставляет элементы таким образом, что инвариант структуры бинарного дерева (любой узел не больше своего родительского узла) остается неизменным!

pop_heap

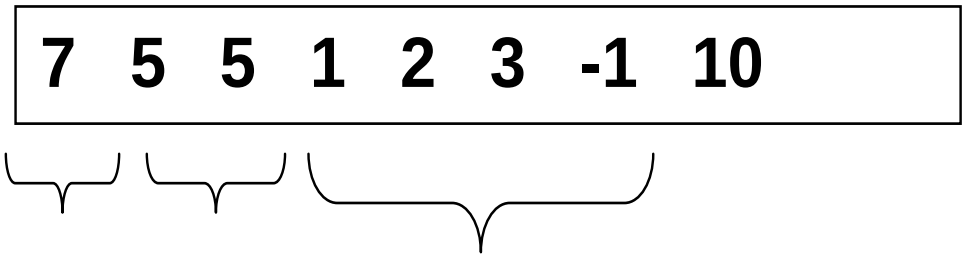
```
int ar[] = {3,5,-1,7,2, 5,10,1};
size_t n = sizeof(ar)/sizeof(int);
make_heap(ar,ar+n);

pop_heap(ar,ar+n); //7 5 5 1 2 3 -1 10
pop_heap(ar,ar+n-1); //5 5 3 1 2 -1 7 10
```

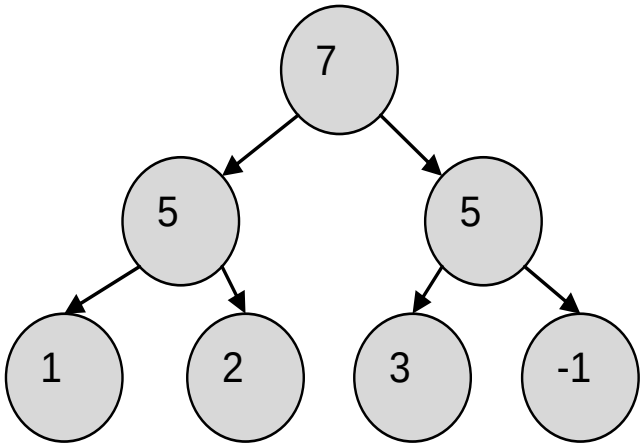
make_heap()



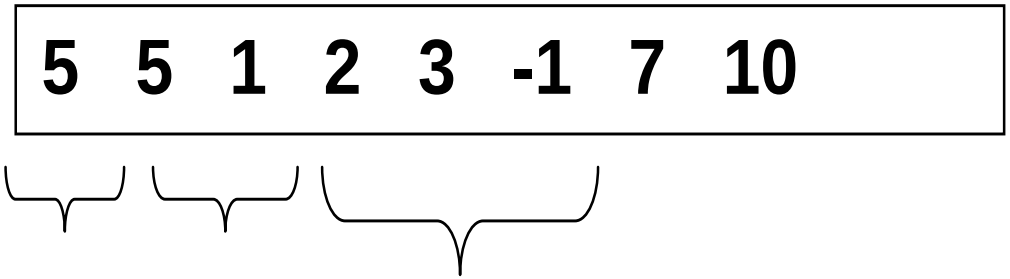


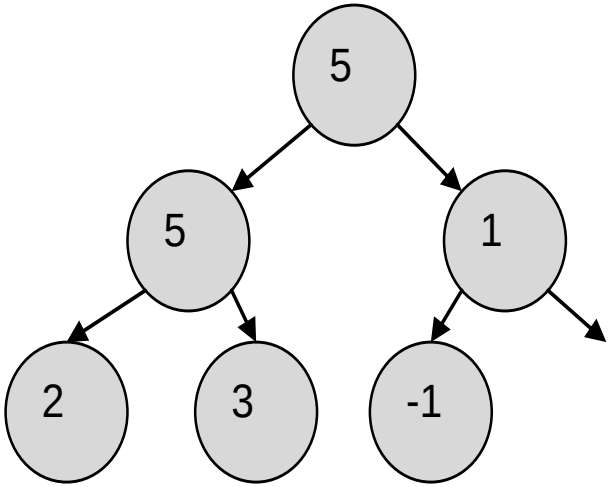
pop_heap





pop_heap





sort_heap

преобразует кучу [beg,end) в упорядоченный интервал. После вызова интервал перестает быть кучей. Сложность не более numberOfElements*log(numberOfElements) сравнений

- void **sort_heap** (RandomAccessIterator beg, RandomAccessIterator end)
- void **sort_heap** (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)

sort_heap

```
int ar[] = {3,5,-1,7,2, 5,10,1};
int n = sizeof(ar)/sizeof(int);
make_heap(ar,ar+n);
…
sort_heap(ar,ar+n); //-1,1,2,3,5,5,7,10
```

priority_queue – push()

```
void push(const T& t)
{
        c.push_back(t);
        push_heap(c.begin(), c.end(), comp);
}
```

priority_queue – pop()

```
void pop()
{
        pop_heap(c.begin(), c.end(), comp);
        c.pop_back();
}
```

Пример priority_queue
(использование умолчаний)

```
priority_queue<char> q; //???
q.push('B'); //B
q.push(‘A'); //BA
q.push(‘F'); //FBA
q.push('W'); //WFBA

//Распечатать значения элементов
        …//WFBA
```

Пример (использование умолчаний)

```
priority_queue<char> q;
q.push('B');
q.push('F');
q.push('A');
q.push('W');

//Распечатать значения элементов
while(!q.empty())
{
    cout<<q.top()<<" ";
    q.pop();
}
```

Пример (явное задание)

greater - #include <functional>

```
priority_queue<char,deque<char>, greater<char>> q;
    q.push('B');//B
    q.push('F');//BF
    q.push('A');//AFB
    q.push('W');//AFBW
    while(!q.empty())
    {
            cout<<q.top()<<" ";          //ABFW
                q.pop();
    }
```

Пример

«Упорядочение» без учета регистра???

Пример (пользовательский критерий)

```
class Nocase{
public:
        bool operator()  (char left, char right)
        { … }
};
int main()
{
        priority_queue<char,vector<char>, Nocase > q;
        q.push('b');
        q.push('F');
        q.push('a');
        q.push('W');
//abFW
}
```

АССОЦИАТИВНЫЕ КОНТЕЙНЕРЫ

set (множество)

namespace std

#include <set>

template<class _Key,

class _Pr = less<_Key>,

class _Alloc = allocator<_Key> >

class set : **public _Tree**<_Tset_traits<_Kty, _Pr, _Alloc, false> >

{// ordered red-black tree of key values, unique keys

С. Мейерс «Эффективное использование STL».

“Даже если гарантированное логарифмическое время поиска вас устраивает, стандартные ассоциативные контейнеры не всегда являются лучшим выбором. Как ни странно, стандартные ассоциативные контейнеры по быстродействию нередко уступают банальному контейнеру vector”

Красно-черное дерево

Баланс достигается за счет поддержания раскраски вершин в два цвета (красный и черный). Подчиняется следующим правилам:

- Красная вершина не может быть сыном красной вершины
- Черная глубина любого листа одинакова (черной глубиной называют количество черных вершин на пути из корня)
- Корень дерева черный

Специфика set

1. Данные **всегда** хранятся в **упорядоченном** виде! =>

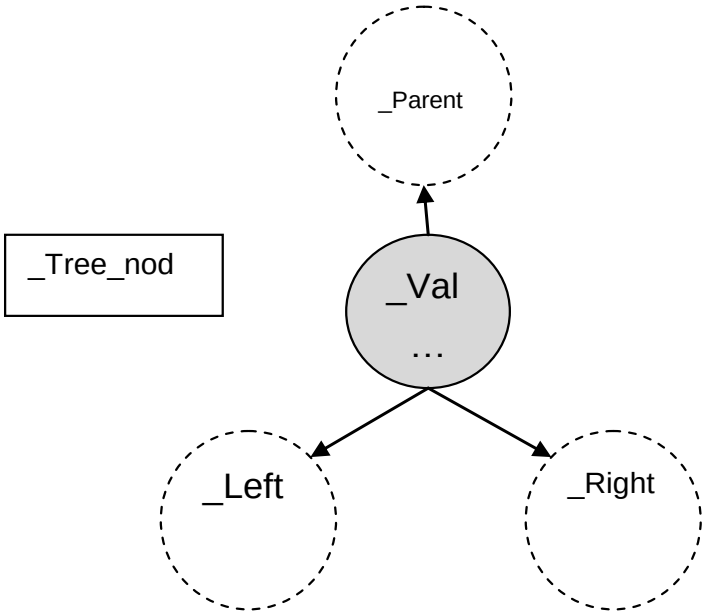
поиск происходит очень **быстро**!

Специфика set

2. Самый простой из ассоциативных контейнеров, так как ключ совпадает со значением

Специфика set

3. Базовым классом является двоичное сбалансированное дерево. Дерево состоит из узлов:



Замечание: тип конкретного используемого дерева зависит от реализации

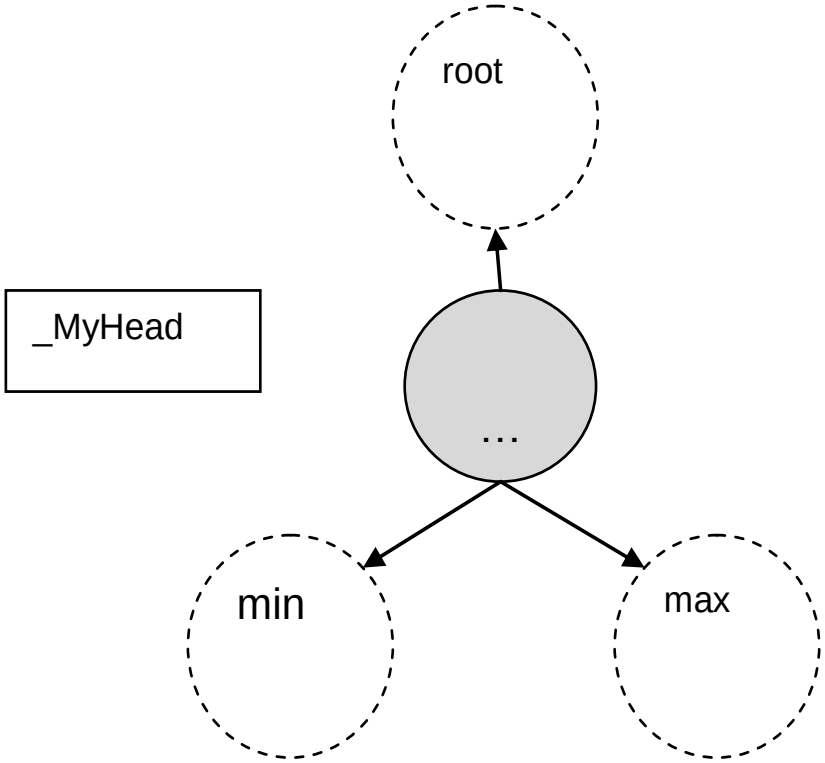
Специфика set

4. Значения уникальны => дубли игнорируются!

Специфика set

5. Данные: фиктивный элемент-**страж -**

_MyHead + _Mysize



Специфика set

6. Нет операций push_back(), так как заполняется в соответствии с критерием упорядоченности => вставка происходит только методом **insert()**

pair<iterator,bool> **insert** (const value_type& val);
pair<iterator,bool> **insert** (value_type&& val);

Метод insert() возвращает объект типа std::pair<Iterator, bool> - пару, в которой содержится

- итератор, указывающий на добавленный/уже существующий объект
- и признак: true – значение было добавлено, false – такое значение уже присутствует в контейнере

Пример

```
set <int> s; //???
s.insert(10);
s.insert(2);
s.insert(30);
s.insert(20);
s.insert(1);
s.insert(11);
s.insert(10); //???
//Распечатать значения элементов
…
```

Продолжение примера

```
for(set<int>::iterator it=s.begin();
                    it!=s.end(); ++it)
{
        cout<<*it<<"  ";
}
```

std::set::insert()

- iterator insert (const_iterator position, const value_type& val);
- iterator insert (const_iterator position, value_type&& val);
- template <class InputIterator> void insert (InputIterator first, InputIterator last);
- void insert (initializer_list<value_type> il);

Примеры insert()

```
std::set<int> myset = { 5, 1, -10, 33 };
std::pair<std::set<int>::iterator, bool> ret= myset.insert(5);
ret= myset.insert(20); //???
std::set<int>::iterator it = ret.first;
myset.insert(it, 25);


myset.insert({ 4, 7 });   //???

int ar[] = { 5, 10, 15 };
myset.insert(ar, ar + 3);   // ???
```

Специфика set

7. Сложный итератор, который ++ (--) умеет перемещаться на узел с бОльшим значением, выбирая наименьшее из поддеревьев

Специфика set

8. Итератор предназначен только для чтения. Почему???

```
set <int> s;
s.insert(10);
…
set<int>::iterator it=s.begin();
*it = 33; //???
```

Специфика set

9. Реверсивный итератор

Распечатать set в обратном порядке

Специфика set

10. Если дерево «вырождается» в **не** сбалансированное, оно **перестраивается**! (то есть изменяется root)

```
set<int> s;
s.insert(10);
s.insert(9);
s.insert(8); //дерево «перестраивается»
s.insert(7);
```

Пример

1.

```
int ar1[5] = {1,2,3,4,5};
set<int> s1(ar1, ar1+5);
//печать элементов
int ar2[5] = {5,4,3,2,1};
set<int> s2(ar2, ar2+5);
 //печать элементов ???
```

Пример

2.

```
vector<int> v;
//формирование значений v
set<int> s1(v.begin(), v.end());
//печать элементов s1
set<int> s2(v.rbegin(), v.rend());
 //печать элементов s2
```

Пример

3.

```
int ar1[10] = {1,2,3,4,5,1,3,5};
set<int> s(ar1, ar1+10);  //???
//печать элементов
```

Пример

4. Явное задание критерия упорядочения

```
int ar1[10] = {1,2,3,4,5};
set<int, greater<int>> s(ar1, ar1+10);
//печать элементов
 set<int, greater<int>> ::iterator it =
                                  s.begin();
…
```

Пример

5. Требуется упорядочить значения по модулю???

Пример

5. Пользовательский критерий упорядочения

```
struct Abs{
bool operator()(int x, int y){return abs(x)<abs(y);}
};

        int ar1[10] = {5,-1,-3,2,10};
        set<int, Abs> s(ar1, ar1+10); //печать элементов
        set<int, Abs> ::iterator it = s.begin();
        …
```

Пример

6. Упорядоченное чтение строк из файла

```
{
        ifstream f(“my.txt”);
        string word;
        set<string> words;
        while(f>>word, !f.eof())
        {
                words.insert(word); //???
        }
        f.close();
}
```

Пример

7.

```
vector<int> v;
//формирование значений
set<int>  s(v.begin(), v.end());
vector<int> v2(s.begin(), s.end());
```

v==v2 **???**

Специфические методы set

iterator **lower_bound**(const Key& _Key);
//возвращает итератор на элемент, значение которого **>=**_Key

iterator **upper_bound**(const Key& _Key);
//возвращает итератор на элемент, значение которого **>**_Key

pair <iterator, iterator> **equal_range** (const Key& _Key); //возвращает пару итераторов

erase

- iterator erase(iterator _Where);
- iterator erase(iterator _First, iterator _Last);
- size_type erase(const key_type& _Key);

Пример

```
set<int> s;
//10, 20, 25, 30
typedef set<int>::iterator IT;
IT it1 = s.lower_bound(15); //???
IT it2 = s.lower_bound(30); //???
IT it3 = s.upper_bound(15); //???
IT it4 = s. upper_bound(30); //???
//Распечатать от it1 до it2
//Стереть от it3 до it4
```

multiset (множество с повторениями)

namespace std

#include <set>

template<class _Key,

class _Pr = less<_Key>,

class _Alloc = allocator<_Key> >

class **multiset**

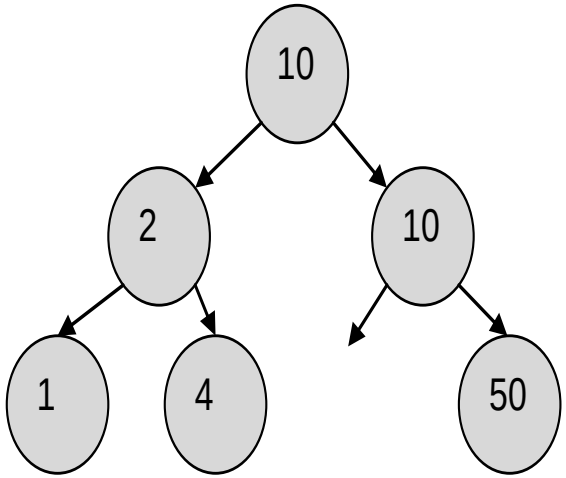
: public _Tree<_Tset_traits<_Key, _Pr, _Alloc, true> >

{// ordered red-black tree of key values, non-unique keys

Специфика multiset

Допускает **дублирование** значений

```
multiset <int > s;
s.insert( 10 );
s.insert( 2 );
s.insert( 10 );
s.insert( 4 );
s.insert( 1);
s.insert( 50);
```



Специфика multiset

- Значения могут повторяться
- Метод find возвращает итератор на первый элемент с «искомым» значением

map (словарь)

namespace std

#include <map>

template<class **_Key**,

class **_Val**,

class _Pr = less<_Key>,

class _Alloc = allocator<pair<**const _Key**, _Val> > >

class **map**

: public _Tree<_Tmap_traits<_Key, _Val, _Pr, _Alloc, false> >

{// ordered red-black tree of {key, mapped} values, unique keys

Специфика map

1. Настоящий ассоциативный контейнер (значение не совпадает с ключом)

Специфика map

2. **Ключи уникальны** => для пары с существующим ключом модифицируется значение

Специфика map

3. map отсортирован по ключам => критерий сравнения задается для ключей

Специфика map

4. Для хранения пары ключ/значение используется шаблон структуры pair:

```
template<typename T1, typename T2> struct pair{
        T1 first;
        T2 second;
        pair(const T1& t1, const T2& t2):first(t1), second(t2){}
};

#include <utility>
int main()
{
        pair<char,double> p('A',1.1); //ключ/значение
        pair<char,double> p1=make_pair('B', 2.2);
        p.first = 'C'; //изменяю ключ
        p.second = 3.3;// изменяю значение
}
```

Специфика map

5. Псевдоним value_type сопоставляется чему???

При разыменовании итератора получаем что???

Можно ли использовать итератор для записи???

Использование pair в map:

typedef pair<**const** _Kty, _Ty> value_type;

Специфика map

6. operator[]

Специфика map

7. метод insert()

- принимает пару ключ/значение
- Возвращает пару итератор/bool (true, если пара занесена)

Специфика map

8. Методы find() и erase() осуществляют поиск по ключу

Пример

```
map<string, int> book;
book[“Иванов”] = 1111111;
pair< map<string, int>::iterator, bool> r=
      book.insert(pair<string, int>(“Петров”,
                                    2222222));
//Печать записной книжки
???
```

Пример

```
map<string, int> book;
book[string(“Иванов”)] = 1111111;
book[string(“Петров”)] = 2222222;
…
map<string, int>::iterator it = book.begin();
while(it != book.end())
{
  cout<<(*it).first<<“:”<<(*it).second<<endl;
  ++it;
}
```

???

```
map<string, int>::iterator it=
                book.find(“Петров”);
//проверка – если существует ???
        …
(*it).second = 3333333; //???
(*it).first = “Сидоров”; //???
```

Как изменить значение ключа?

???

at()

```
Type& at( const Key& _Key ) throw(out_of_range);
const Type& at( const Key& _Key ) const
    throw(out_of_range);
```

Пример at()

```
typedef std::map<char, int> Mymap;
int main()
   {
   Mymap m;
    m.insert(Mymap::value_type('a', 1));
   m.insert(Mymap::value_type('b', 2));

try{
   std::cout << "a -> " << m.at('a') << std::endl;  //???
   std::cout << "c -> " << m.at('c') << std::endl;  //???
   } catch( std::out_of_range& x){std::cout <<x.what();}
}
```

multimap (словарь с с повторениями)

namespace std

#include <map>

template<class _Key,

class _Val,

class _Pr = less<_Key>,

class _Alloc = allocator<pair<const _Key, _Val> > >

class **multimap**

: public _Tree<_Tmap_traits<_Key, _Val, _Pr, _Alloc, true> >

{// ordered red-black tree of {key, mapped} values, non-unique keys

Специфика multimap

- позволяет дублировать значения ключей, элементы с одинаковыми ключами в контейнере хранятся в порядке занесения
- => не определен operator[]
- добавление элементов с помощью только insert()
- при удалении по ключу (erase()) удаляются все элементы, удовлетворяющие условию
- count() – возвращает количество пар, в которых ключ совпадает с указанным значением