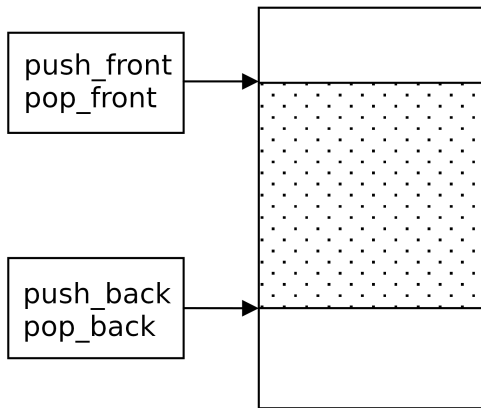


Технология ООП

Санкт-Петербургский государственный политехнический университет

11 октября 2011

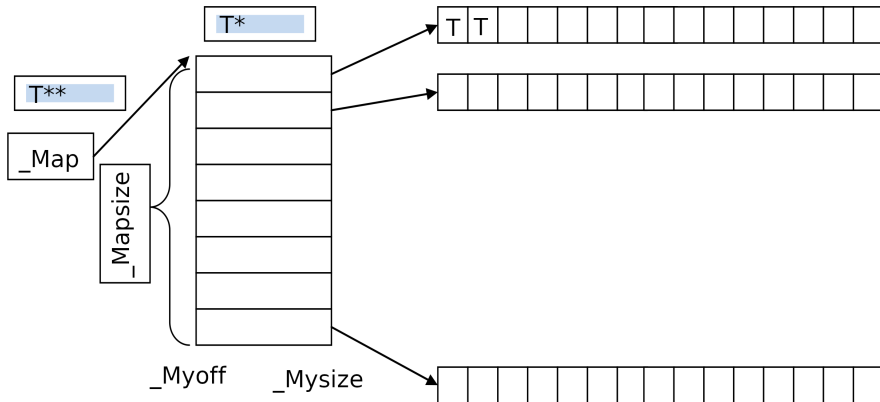
Назначение deque



- Обеспечивать произвольный доступ к элементам (как vector $\Rightarrow$ `at()`, `operator[]`)
- Эффективная работа с началом последовательности (как у list $\Rightarrow$ `push_front()`, `pop_front()`)

```
#include <deque>
template<class _Ty,
class _Ax = allocator<_Ty> >
class deque : public _Deque_val<_Ty, _Ax>
{ // circular queue of pointers to blocks
```

Внутреннее устройство deque



Реализация итератора deque

```
iterator begin() { return iterator(_MyOff, this); }  
iterator end() {  
    return iterator(_MyOff + _Mysize, this); }  
T& operator[](int i) { return *(begin() + i); }
```

```
class iterator {  
    deque* _MyCont;  
    size_type _Off;  
public:  
    iterator& operator++() { ++_Off; return *this; }  
    T& operator*() {  
        size_type n = _Off/16;  
        n = n % (_MyCont->_Mysize);  
        size_type m = _Off % 16;  
        return _MyCont->_Map[n][m];  
    }  
}
```

Пример создания deque

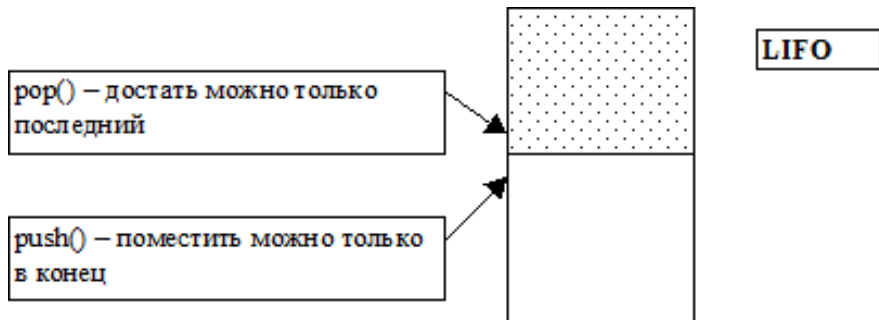
```
deque<char> d; //_Map=0, _Mapsize=0, _Myoff=0,
    _Mysize=0
d.push_front('A'); //_Map, _Mapsize=8, _Myoff=127,
    _Mysize=1
d.push_front('B'); //_Map, _Mapsize=8, _Myoff=126,
    _Mysize=2
d.push_front('C'); //_Map, _Mapsize=8, _Myoff=125,
    _Mysize=3
d.push_back('a'); //_Myoff=125, _Mysize=4
d.push_back('b'); //_Mysize=5
d.push_back('c'); //_Mysize=6
deque<char>::iterator it = d.begin();
while (it!=d.end()) {
    cout << *it << " "; ++it;
}
```

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

- Последовательные:
 - Базовые: `vector`, `list`, `deque`.
 - Адаптеры базовых: `stack`, `queue`, `priority_queue`.
- Ассоциативные: `set`, `multiset`, `map`, `multimap`, `hash_map`, `hash_set`.

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

Назначение стека



```
template<class T, class C = deque<T> > class stack
{
protected: C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_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 pop() { c.pop_back(); }
};
```

Запрещены все не стековые операции. Перегружены общепринятые операции для стека посредством базовой последовательности.

- Удаляет последний элемент: `pop_back()` $\Rightarrow$ `pop()`
- Возвращает значение последнего элемента: `back()` $\Rightarrow$ `top()`
- Добавляет в конец последовательности: `push_back()` $\Rightarrow$ `push()`

Из базового контейнера недоступны:

- методы;
- итераторы;
- псевдонимы.

Пример использования stack

```
stack<int> s; //size?  
s.push(1); //size == 1  
s.push(2); //size == 2  
s.pop(); // size == 1  
int tmp = s1.top(); //tmp == 1, size == 1  
if(s1.top() == 1)  
    s1.pop();
```

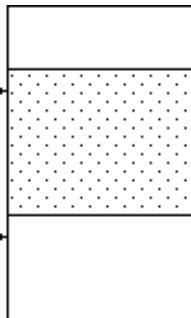
Пример создания stack

```
vector<int> v(10,1);  
//создать стек таким образом, чтобы его элементы  
    стали копиями элементов вектора  
  
//создать пустой стек на базе list  
  
//Распечатать значения элементов стека:
```

Назначение queue

`pop()` – достать можно только первый

`push()` – поместить можно только в конец



FIFO

```
template<class T, class C = deque<T> > class 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 использования очереди

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

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

Адаптер очереди с приоритетом

- Вставка и удаление элементов реализованы посредством: `std::make_heap()`, `std::push_heap()` и `std::pop_heap()`.
- Используется внедренный объект компаратора: `if(cmp(<элемент очереди>, <вставляемое значение>))`.
- Критерий сравнения по умолчанию использует предикат — шаблон структуры `std::less`.

Предикат сравнения less

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

- Реализуется посредством последовательного контейнера (массива).
- Представляет собой линейризованное бинарное дерево.
- Первый элемент всегда является максимальным.
- Добавление и удаление элементов в общем случае производится с логарифмической сложностью.

Алгоритмы STL для работы с кучами

Для использования требуется подключить `<algorithm>`, в котором реализованы функции:

- `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.

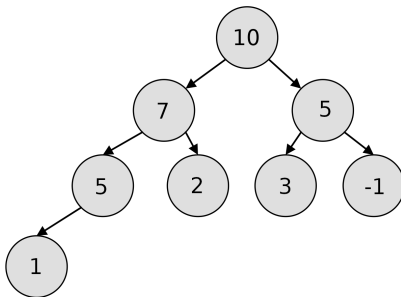
Прототипы всех функций унифицированы:

```
... _heap (RandomAccessIterator beg,  
          RandomAccessIterator end);  
... _heap (RandomAccessIterator beg,  
          RandomAccessIterator end, BinaryPredicate op);
```

make_heap()

Преобразует последовательность [beg, end) в кучу за линейное время — не более $3 \cdot n$ сравнений. Значение каждого узла бинарного дерева меньше или равно значению родительского узла.

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

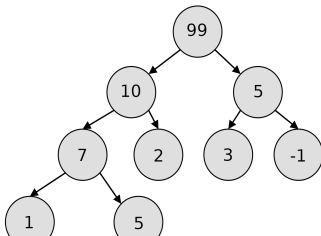


push_heap()

Добавляет последний элемент в существующую кучу.

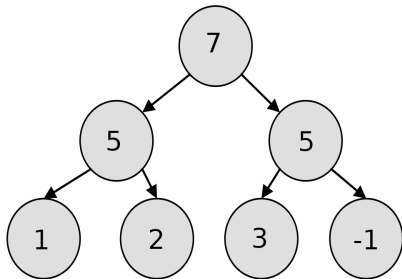
Сложность — не более $\log(n)$ сравнений. Алгоритм `push_heap()` переставляет элементы таким образом, что инвариант структуры бинарного дерева, т.е. любой узел не больше своего родительского узла, остается неизменным.

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

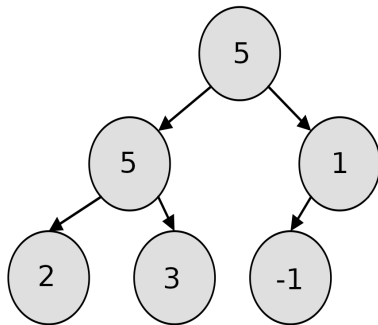


Перемещает максимальный элемент кучи [beg,end) в конец и создают новую кучу из оставшихся элементов в интервале [beg,end-1). Сложность — не более $2 \cdot \log(n)$ сравнений. Алгоритм pop_heap() переставляет элементы таким образом, что инвариант структуры бинарного дерева остается неизменным.

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



7 5 5 1 2 3 -1 10



5 5 3 1 2 -1 7 10

Преобразует кучу [beg,end) в упорядоченный интервал. После вызова интервал перестает быть кучей. Сложность логарифмическая (не более $n \cdot \log(n)$ сравнений).

```
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
```

Методы push и pop в очереди с приоритетом

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

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

Пример использования priority_queue

```
priority_queue<char> q; //???  
q.push('B'); //B  
q.push('F'); //FB  
q.push('A'); //FBA  
q.push('W'); //WFBA  
//Распечатать значения элементов
```

Явное задание предиката в очереди с приоритетом

```
priority_queue<char, deque<char>, greater<char>> q1;  
q1.push('B'); //B  
q1.push('F'); //BF  
q1.push('A'); //ABF  
q1.push('W'); //ABFW
```

```
class Nocache {  
public:  
    bool operator() (char left, char right) { }  
};
```

```
priority_queue<char, vector<char>, Nocache > q2;  
q2.push('b'); //b  
q2.push('F'); //bF  
q2.push('a'); //abF  
q2.push('W'); //abFW
```
