Многопоточное программирование на C++

Контейнеры

Контейнеры

Библиотека контейнеров является универсальной коллекцией шаблонов классов и алгоритмов, позволяющих программистам легко реализовывать общие структуры данных, такие как очереди, списки и стеки.

Контейнер управляет выделяемой для его элементов памятью и предоставляет функции-члены для доступа к ним, либо непосредственно, либо через итераторы (объекты, обладающие схожими с указателями свойствами).

Виды контейнеров

- Последовательные
- Ассоциативные
- Неупорядоченные ассоциативные
- Контейнеры-адаптеры

Контейнеры

- STL
- STL (C++11)
- Boost

Последовательные контейнеры

- std::list
- std::vector
- std::deque

C++11:

- array
- forward_list

Последовательные контейнеры

- std::vector хранит все элементы в куче
- std::array хранит все элементы в себе
- std::array не может изменить свой размер
- std::array должен знать свой размер на этапе компиляции
- std::array работает быстрее

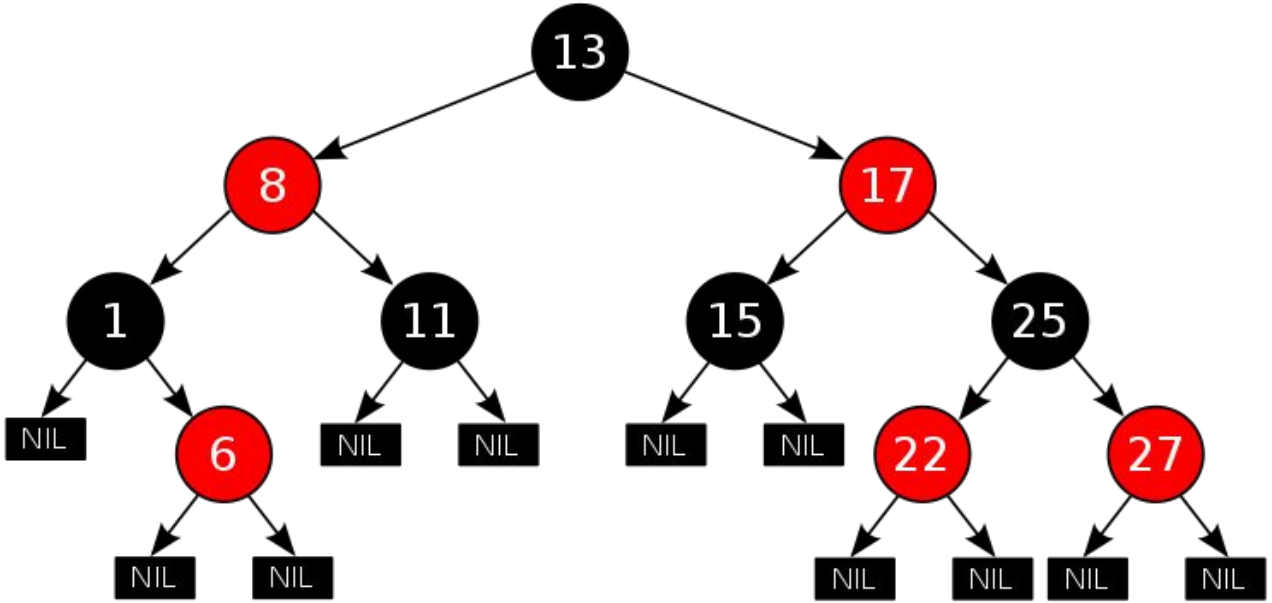
Ассоциативные контейнеры

- std::set
- std::map
- std::multiset
- std::multimap

std::forward_list

- Итератор может двигаться только в одном направлении.

Красно-чёрное дерево



Красно-чёрное дерево

- Узел либо красный, либо черный.
- Корень — чёрный. (В других определениях это правило иногда опускается. Это правило слабо влияет на анализ, так как корень всегда может быть изменен с красного на чёрный, но не обязательно наоборот).
- Все листья (NIL) — чёрные.
- Оба потомка каждого красного узла — чёрные.
- Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число чёрных узлов.

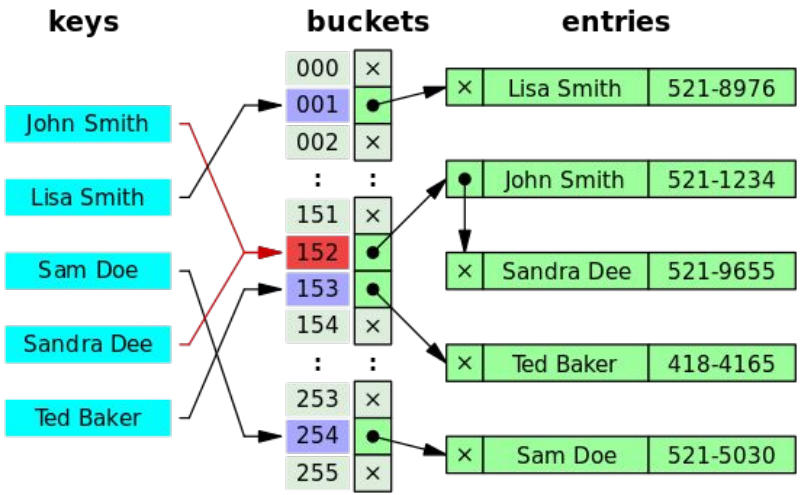
Неупорядоченные ассоциативные контейнеры

C++11:

- std::unordered_set
- std::unordered_map
- std::unordered_multiset
- std::unordered_multimap

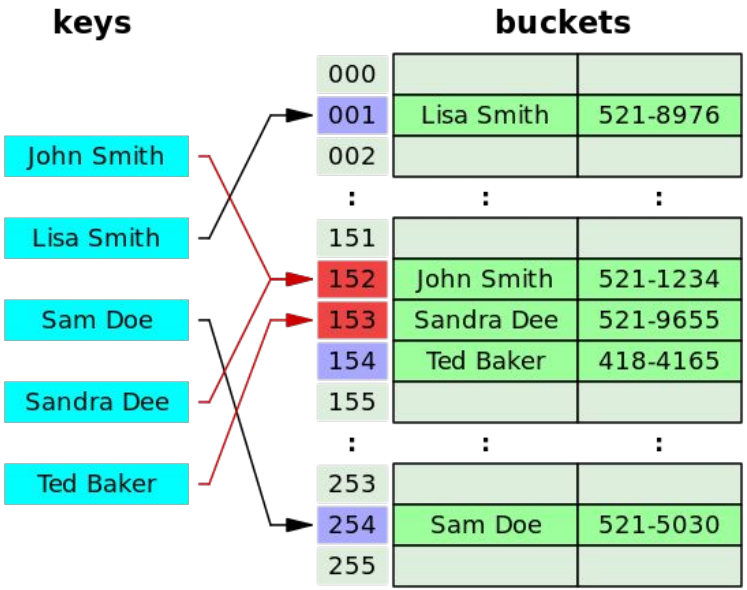
Хеш-таблицы

Разрешение коллизий с помощью цепочек.



Хеш-таблицы

Линейное разрешение коллизий



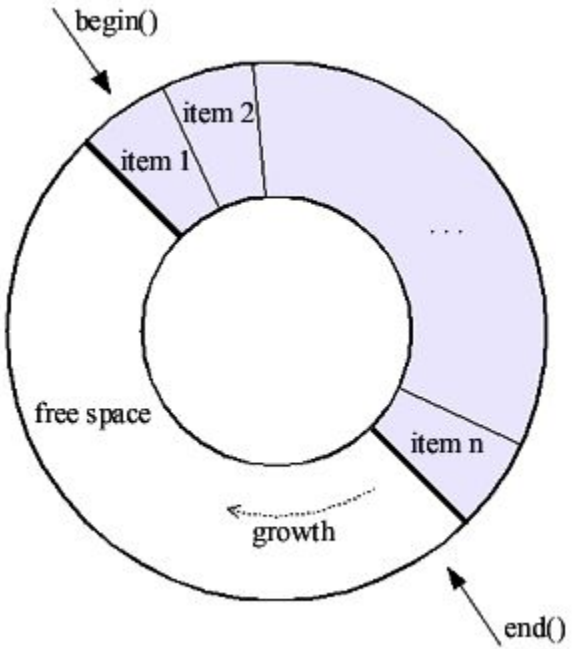
Контейнеры-адаптеры

- std::stack
- std::queue
- std::priority_queue

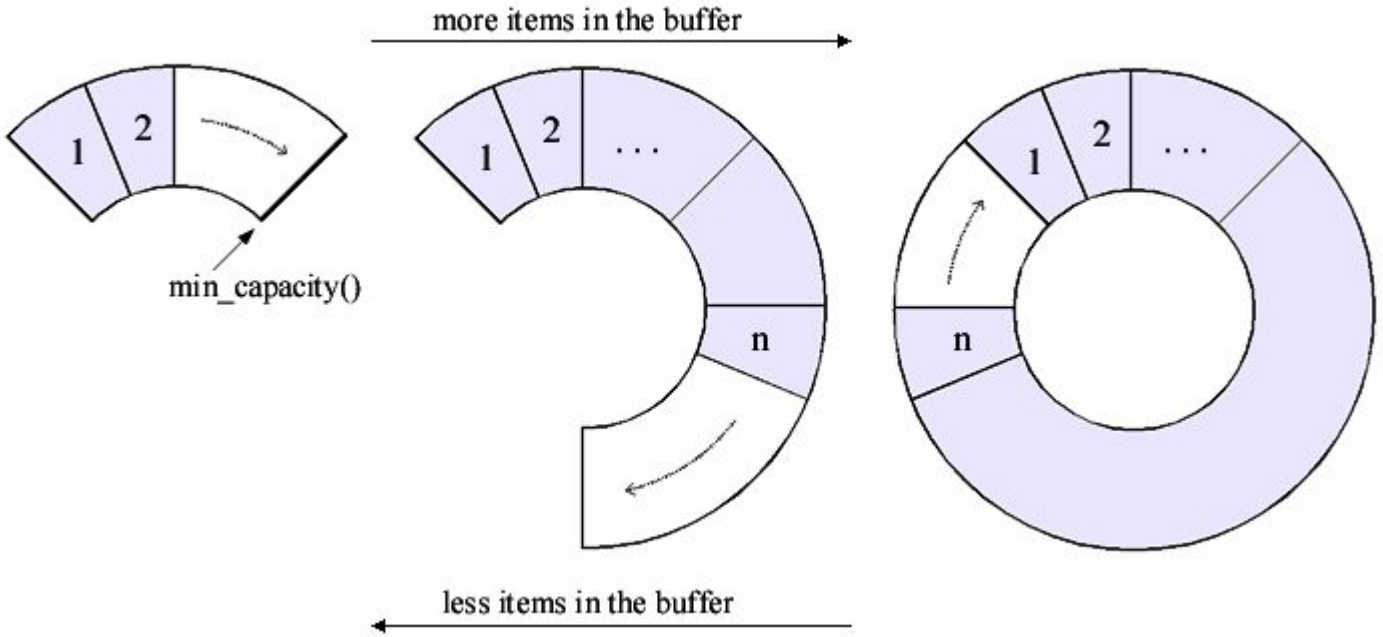
Псевдоконтейнеры STL

- std::bitset
- std::basic_string
- std::valarray

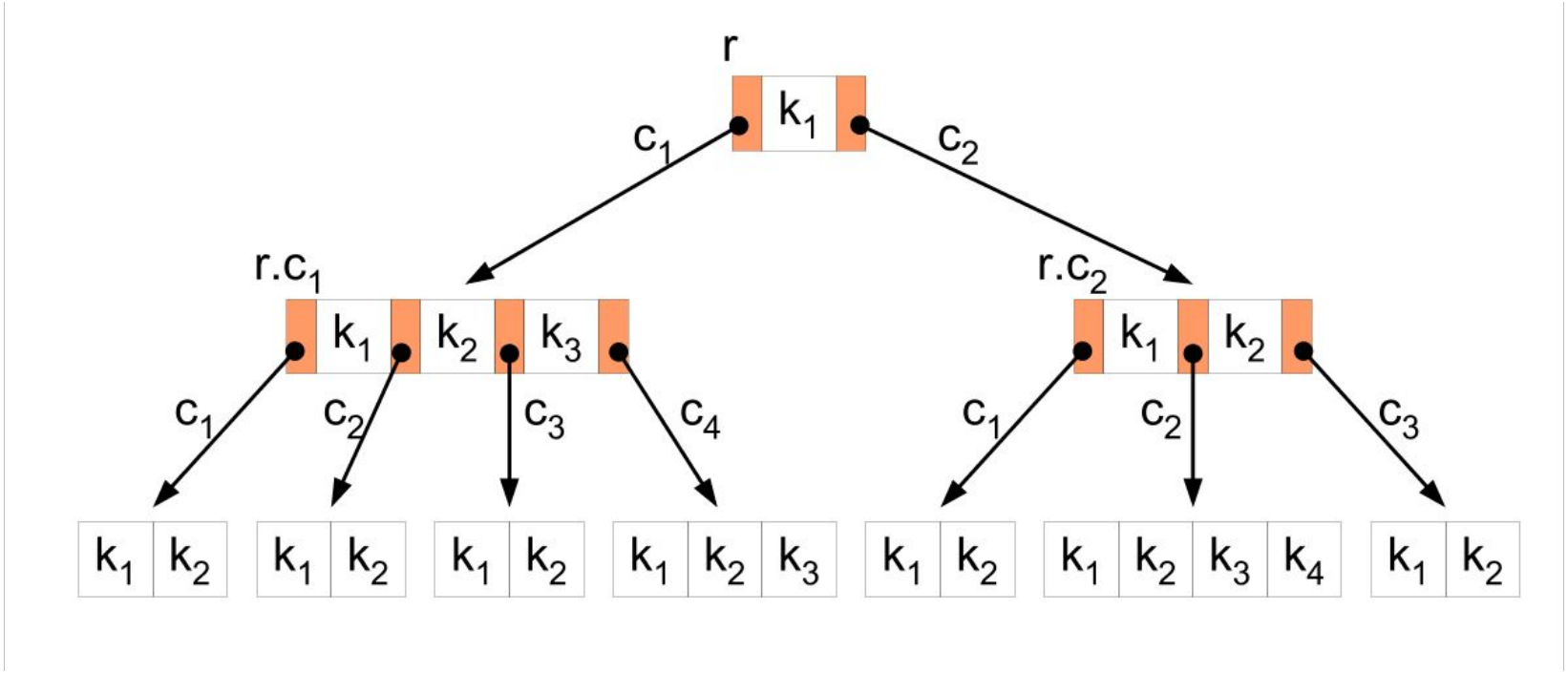
boost::circular_buffer



boost::circular_buffer_space_optimized



B-деревья


r
k1
c1
c2
r.c1
k1
k2
k3
c1
c2
c3
c4
k1
k2
k1
k2
k1
k2
k1
k2
k3
r.c2
k1
k2
c1
c2
c3
k1
k2
k1
k2
k3
k4
k1
k2

https://code.google.com/archive/p/cpp-btree/

- btree_set
- btree_map
- btree_multiset
- btree_multimap

Шаблоны

Traits

```
template< typename T >
struct is_pointer{
  static const bool value = false;
};

template< typename T >
struct is_pointer< T* >{
  static const bool value = true;
};
```

Traits

```
template <typename T,
          typename traits = elem_traits<T> > // свойство по умолчанию
class vector {
     // ...
    public:
     typedef T                          value_type;
     typedef typename traits::arg_type       arg_type;
     typedef typename traits::reference      reference;
     typedef typename traits::const_reference const_reference;

        void push_back(arg_type);

        // ...
};
```

Итераторы

Iterator is the base concept used by other iterator types:

- InputIterator
- OutputIterator
- ForwardIterator
- BidirectionalIterator
- RandomAccessIterator

Iterators can be thought of as an abstraction of pointers.

iterator_traits

Type trait class для свойств Iterator.

- ❏ difference_type
- ❏ value_type
- ❏ pointer
- ❏ reference
- ❏ iterator_category

Iterator tags

Defined in header <iterator>

```
struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag : public input_iterator_tag { };
struct bidirectional_iterator_tag : public forward_iterator_tag { };
struct random_access_iterator_tag : public bidirectional_iterator_tag { };
```

String

```
template <typename charT, typename traits = char_traits<charT> >
class basic_string;

typedef basic_string<char> string;

template <typename charT, typename traits = char_traits<charT> >
class basic_istream;

typedef basic_istream<char> istream;
```

String

```
traits::compare(this->data(), str.data(), rlen)
```

```
class ci_char_traits: public std::char_traits<char> {
public:
    static bool lt (char one, char two) {
        return std::tolower(one) < std::tolower(two);
    }

    static bool eq (char one, char two) {
        return std::tolower(one) == std::tolower(two);
    }

    static int compare (const char* one, const char* two, size_t length) {
        for (size_t i = 0; i < length; ++i) {
            if (lt(one[i], two[i])) return -1;
            if (lt(two[i], one[i])) return +1;
        }
        return 0;
    }
};
```

```
typedef std::basic_string<char, ci_char_traits> CaseInsensitiveString;
```