Technology Tarena 达内科技

模板和STL

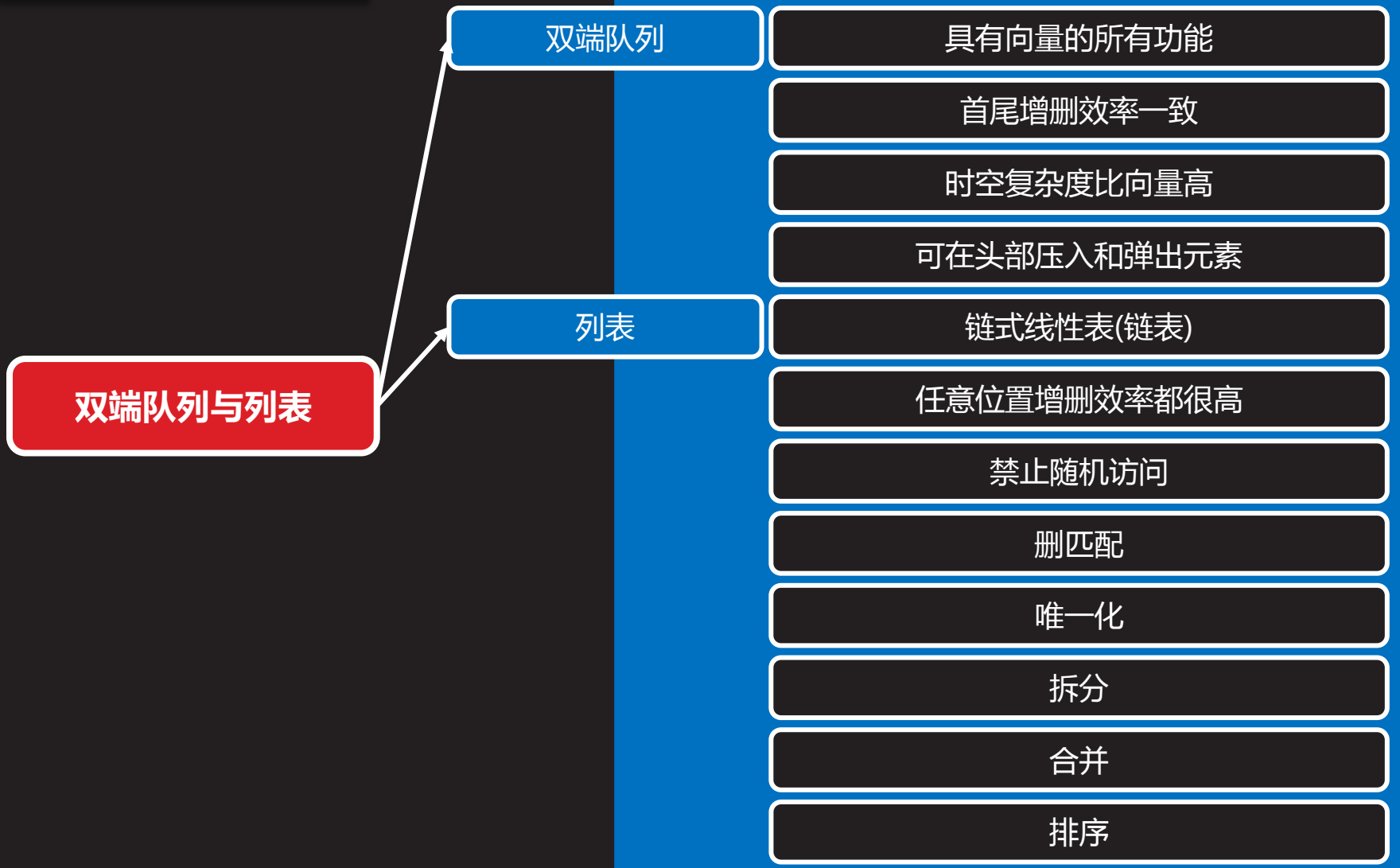
TEMPLATE & STL

DAY04

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	双端队列与列表
	10:30 ~ 11:20	
	11:30 ~ 12:20	堆栈、队列和优先队列
下午	14:00 ~ 14:50	
	15:00 ~ 15:50	映射与多重映射
	16:00 ~ 16:50	
	17:00 ~ 17:30	总结和答疑

双端队列与列表


双端队列与列表
双端队列
具有向量的所有功能
首尾增删效率一致
时空复杂度比向量高
可在头部压入和弹出元素
列表
链式线性表(链表)
任意位置增删效率都很高
禁止随机访问
删匹配
唯一化
拆分
合并
排序

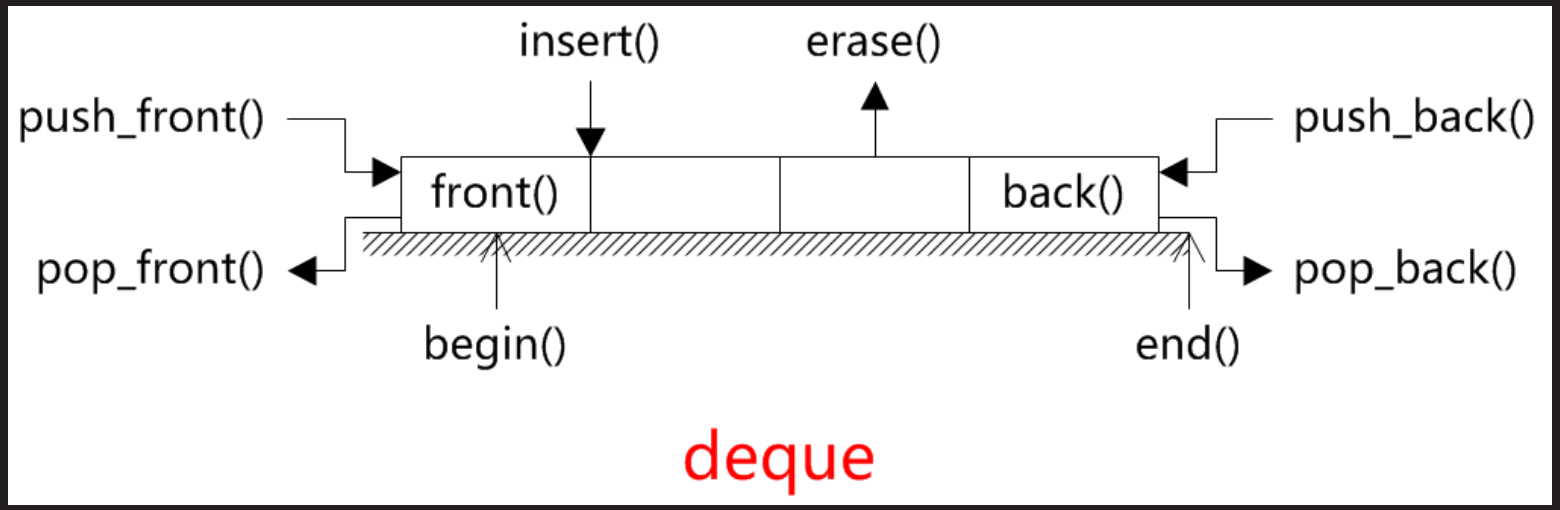
Technology Tarena 达内科技

双端队列

具有向量的所有功能

- 双端队列具有向量的所有功能，除了capacity()和reserve()

知识讲解



首尾增删效率一致

- 在双端队列的头部与在其尾部进行插入/删除(insert()/erase())的效率一样高
  - vector<int> vi (5);
    vi.insert (vi.begin () + 1, 10); // 移动4个元素
    vi.insert (vi.end () - 1, 10); // 移动1个元素
  - deque<int> di (5);
    di.insert (di.begin () + 1, 10); // 移动1个元素
    di.insert (di.end () - 1, 10); // 移动1个元素

知识讲解

时空复杂度比向量高

- 和向量相比，双端队列的内存开销要大一些，对元素下标访问的效率也要略低一些
- 为了保持连续存储空间首尾两端的开放性，双端对列的动态内存管理比向量要复杂一些
- 双端队列需要预留更多的内存，以应对首尾两端的动态增长，其空间复杂度较向量略高
- 双端队列虽然也提供了下标访问的功能，但其首端的开放性势必会增加计算起始地址的时间复杂度

知识讲解

可在头部压入和弹出元素

- 双端队列所提供的push_front()和pop_front()成员函数，可以与push_back()和pop_back()成员函数相类似的方式，在容器的头部压入和弹出元素，其效率也是相当的
- 双端队列的操作性更接近于列表，但它地址连续的存储结构又与向量别无二致。双端队列在随机访问、插入删除等操作的性能方面往往也介于列表和向量之间
- 相对于强调随机访问的向量和突出增删效率的列表，双端队列的表现更加趋于中庸，在需求尚不明确的情况下选择双端队列，即使不是最优的也至少算不上是最差的

知识讲解

双端队列

【参见：TTS COOKBOOK 】

课堂练习

- 双端队列

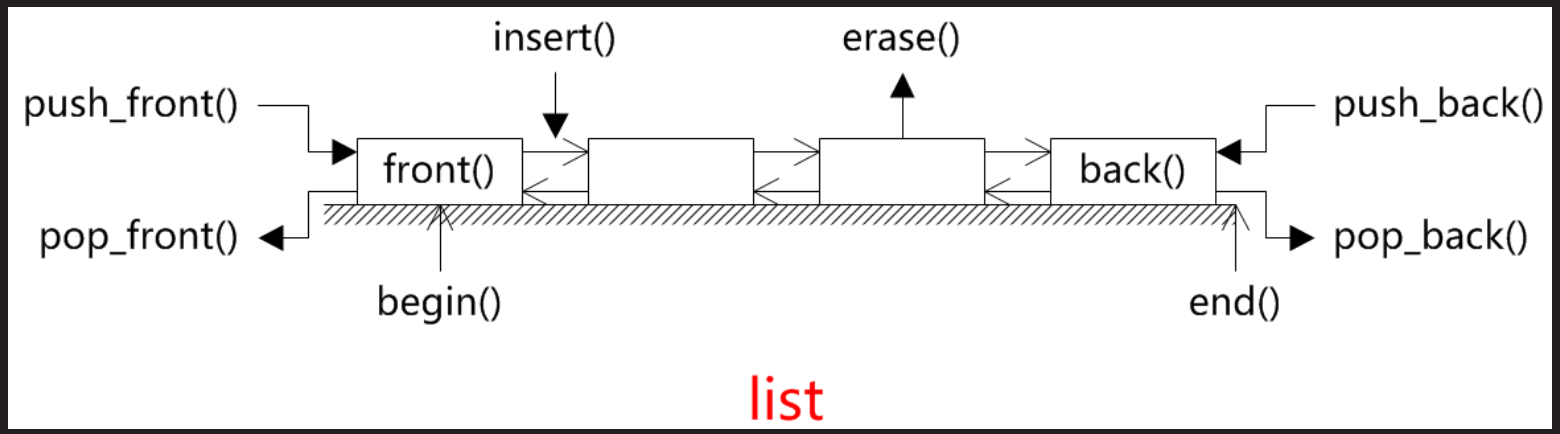
Technology Tarena 达内科技

列表

链式线性表(链表)

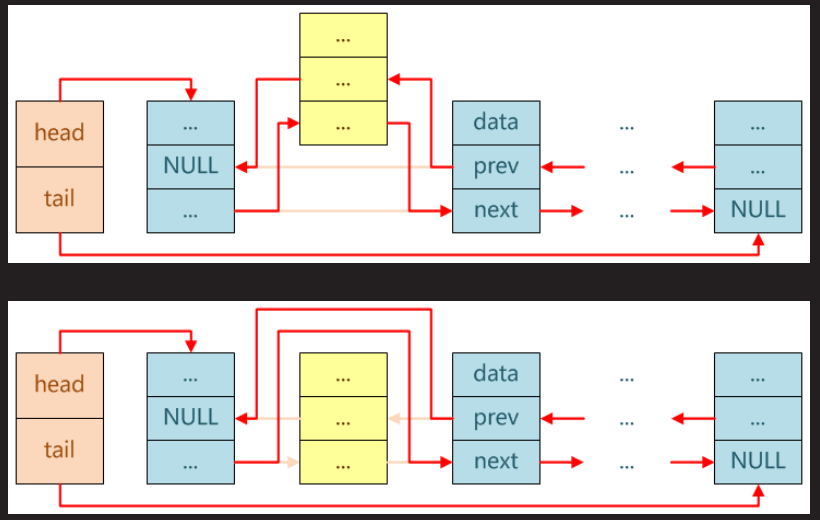
- 列表是按照链式线性表(链表)的形式进行存储的，数据元素存放在地址不连续的离散节点中，彼此通过前后指针彼此关联，符合数据结构中双向线性链表的结构特征

知识讲解



任意位置增删效率都很高

- 在列表的任何位置插入或删除元素的效率都很高。离散的链式存储结构决定了在列表中增删元素不需要移动除被增删节点以外的节点。所需要做的唯一工作仅仅是调整前后节点的指针，以维持链式结构的连续性，而完成这种工作所花费的时间与列表中的元素个数无关，与被增删元素的位置亦无关，即所谓常数时间O(1)，这也是选择列表容器的理由之一



知识讲解

禁止随机访问

- 无法通过下标或迭代器对列表中的元素做随机访问。因为对于列表来说，执行这样的操作至少需要线性时间O(N)
- STL的设计哲学是只在容器内部实现具有较好性能表现的操作，而把那些时间或者空间复杂度偏高的工作，留给用户根据实际应用的具体需求量力而为。这总比包揽一切却处处挨骂要好
- STL容器中只有向量和双端队列采用连续内存存放数据元素，只有这两个容器的随机访问(指针计算)可以在常数时间内完成，因此只有这两个容器的迭代器支持随机迭代，同时也只有这两个容器提供下标运算符。其它容器，包括列表，都只能做顺序迭代，而且也不支持下标运算符

知识讲解

删匹配

- 删除所有匹配元素
  - void remove (const value_type& val);
- 例如
  - list<int> li;
    li.push_back (13);
    li.push_back (27);
    li.push_back (13);
    li.push_back (39);
    li.push_back (13);
    li.remove (13);
    for (list<int>::iterator it = li.begin (); it != li.end (); ++it)
        cout << *it << ' '
    cout << endl;

知识讲解

唯一化

- 连续重复出现的元素只保留一个
  - void unique (void);
- 例如
  - int arr[] = {10, 10, 20, 20, 10, 20, 30, 20, 20, 10, 10};
    list<int> li (arr, arr + sizeof (arr) / sizeof (arr[0]));
    li.unique ();
    for (list<int>::iterator it = li.begin (); it != li.end (); ++it)
        cout << *it << ' ';
    cout << endl;
    // 10 20 10 20 30 20 10

知识讲解

拆分

- 将参数列表的部分或全部元素剪切到调用列表中
  - void splice (iterator pos, list& lst);
    将lst列表全部元素剪切到调用列表pos处
  - void splice (iterator pos, list& lst, iterator del);
    将lst列表del处元素剪切到调用列表pos处
  - void splice (iterator pos, list& lst, iterator begin,
    iterator end);
    将lst列表从begin到end之间的元素剪切到调用列表pos处
- 用两个迭代器表示一个容器中的元素范围，其中的上限迭代器，通常指向该范围中最后一个元素的下一个位置
- 列表拆分的时间复杂度为常数级：O(1)

合并

- 将有序参数列表中的全部元素，合并到有序的调用列表中，合并后的调用列表依然有序，参数列表为空
  - void merge (list& lst);
    利用元素类型的“<”运算符比大小
  - void merge (list& lst, less cmp);
    利用小于比较器比大小
- 列表合并的过程中并没有排序，因此其时间复杂度仅为线性级：O(N)

知识讲解

排序

- 列表自己的排序函数
  - void sort (void);
    利用元素类型的“<”运算符比大小
  - void sort (less cmp);
    利用小于比较器比大小
- 全局函数sort()只适用于拥有随机迭代器的容器，列表的迭代器是顺序迭代器，不能使用全局域的sort()排序函数
- 列表利用自身内存不连续的特点，可以更好的性能实现排序算法函数

知识讲解

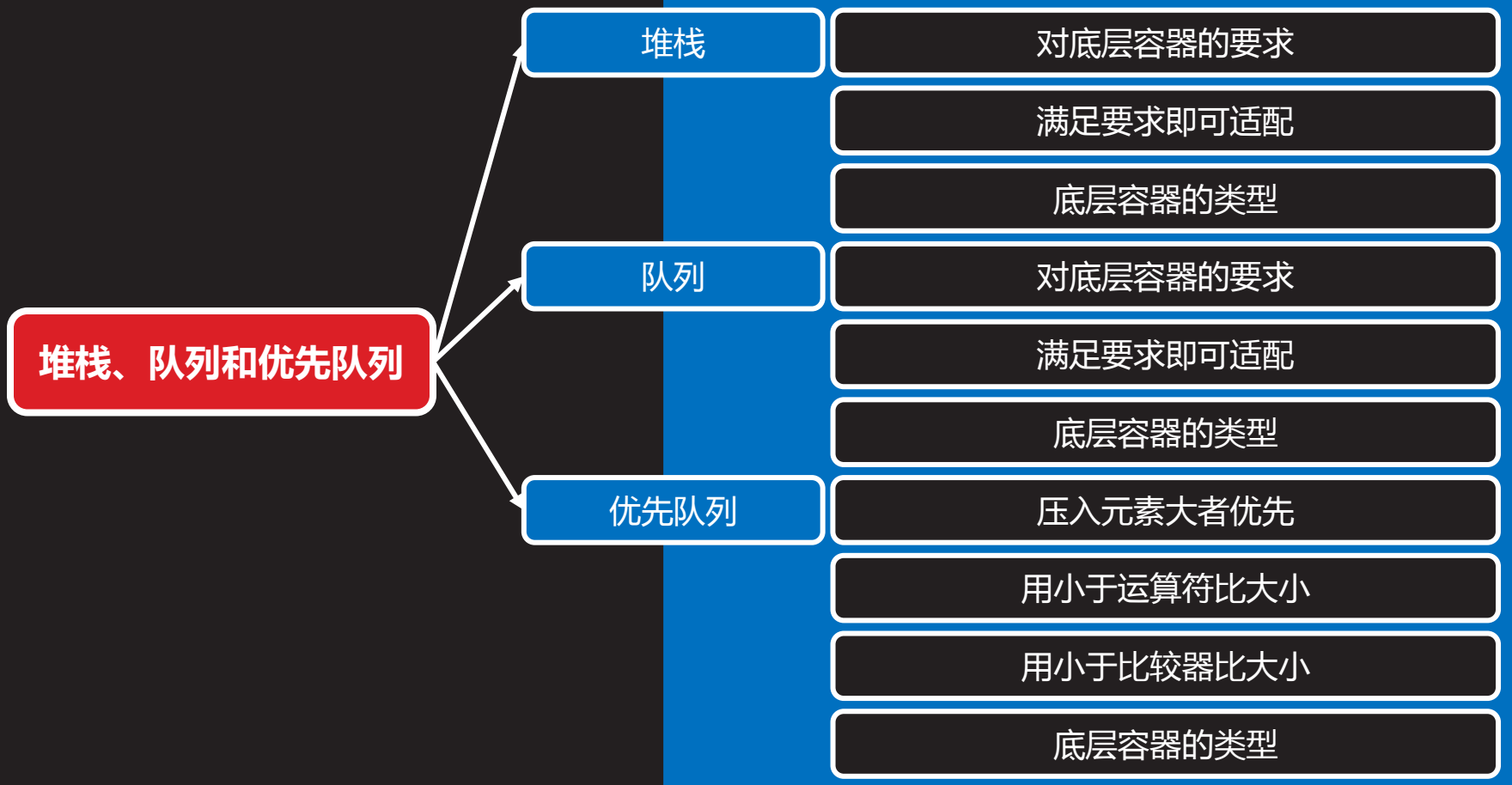
列表

【参见：TTS COOKBOOK 】

课堂练习

- 列表

堆栈、队列和优先队列

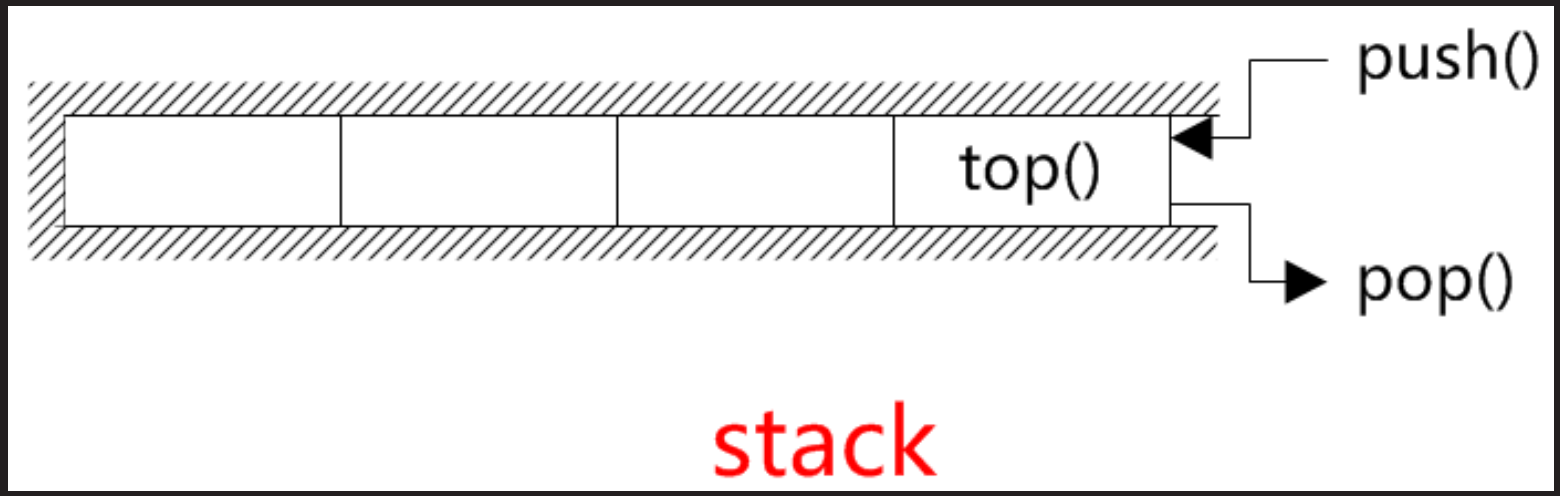

堆栈、队列和优先队列
堆栈
对底层容器的要求
满足要求即可适配
底层容器的类型
队列
对底层容器的要求
满足要求即可适配
底层容器的类型
优先队列
压入元素大者优先
用小于运算符比大小
用小于比较器比大小
底层容器的类型

Technology Tarena 达内科技

堆栈

对底层容器的要求

- 堆栈作为一种适配器容器，可以用任何支持push_back()、pop_back()、back()、size()和empty()等操作的底层容器进行适配



满足要求即可适配

- 除了STL的三种线性容器外，程序员自定义的容器，只要提供了正确的接口函数，也可用于适配堆栈
  - void push (value_type const& val);
    - void push_back (value_type const& val);
  - void pop (void);
    - void pop_back (void);
  - value_type& top (void);
    - value_type& back (void);
  - value_type const& top (void) const;
    - value_type const& back (void) const;
  - size_type size (void) const;
    - size_type size (void) const;
  - bool empty (void) const;
    - bool empty (void) const;

知识讲解

满足要求即可适配（续1）

- 适配器模式的本质就是接口翻译，即将一种形式的接口翻译成另一种形式，以下是堆栈容器的简化实现
  - template<typename value_type, typename container_type = deque<value_type> > class stack {
    public:
    void push (value_type const& val) {
    m_container.push_back (val); }
    void pop (void) { m_container.pop_back (); }
    value_type& top (void) { return m_container.back (); }
    value_type const& top (void) const {
    return const_cast<stack*> (this)->top (); }
    size_t size (void) const { return m_container.size (); }
    bool empty (void) const { return m_container.empty (); }
    private:
    container_type m_container; };

知识讲解

底层容器的类型

- 定义堆栈容器时可以指定底层容器的类型
  - stack<string, vector<string> > ss;
    注意两个“>”之间的至少要留一个空格，否则编译器会把“>>”理解为右移运算符，导致编译错误
- 三种线性容器(vector、deque、list)中的任何一种都可以作为堆栈的底层容器，其中双端队列是缺省底层容器
  - stack<int> si; // stack<int, deque<int> > si;

知识讲解

堆栈

【参见：TTS COOKBOOK 】

课堂练习

- 堆栈

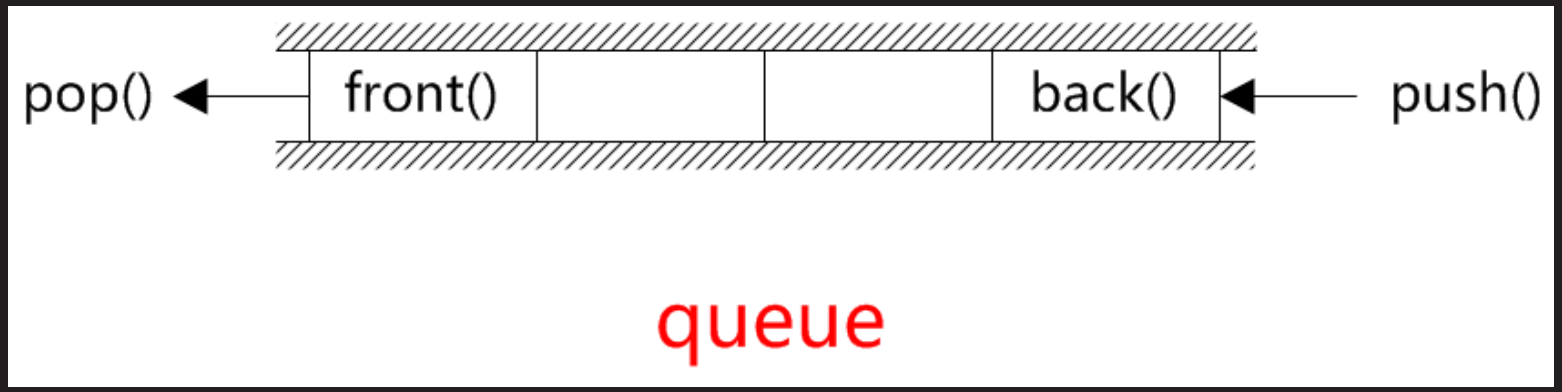
Technology Tarena 达内科技

队列

对底层容器的要求

- 队列作为一种适配器容器，可以用任何支持push_back()、pop_front()、 back()、front()、 size()和empty() 等操作的底层容器进行适配

知识讲解



满足要求即可适配

- 除了STL的三种线性容器外，程序员自定义的容器，只要提供了正确的接口函数，也可用于适配队列
  - void push (value_type const& val);
    - void push_back (value_type const& val);
  - void pop (void);
    - void pop_front (void);
  - value_type& back (void);
    - value_type& back (void);
  - value_type const& back (void) const;
    - value_type const& back (void) const;
  - value_type& front (void);
    - value_type& front (void);
  - value_type const& front (void) const;
    - value_type const& front (void) const;
  - size_type size (void) const;
    - size_type size (void) const;
  - bool empty (void) const;
    - bool empty (void) const;

知识讲解

底层容器的类型

- 定义队列容器时可以指定底层容器的类型
  - queue<string, list<string> > qs;
    注意两个“>”之间的至少要留一个空格，否则编译器会把“>>”理解为右移运算符，导致编译错误
- 三种线性容器中除向量以外的任何一种都可以作为队列的底层容器，其中双端队列是缺省底层容器
  - queue<int> qi; // queue<int, deque<int> > qi;
- 不能用向量适配队列的原因是，它缺少pop_front()和front()接口

知识讲解

队列

【参见：TTS COOKBOOK 】

课堂练习

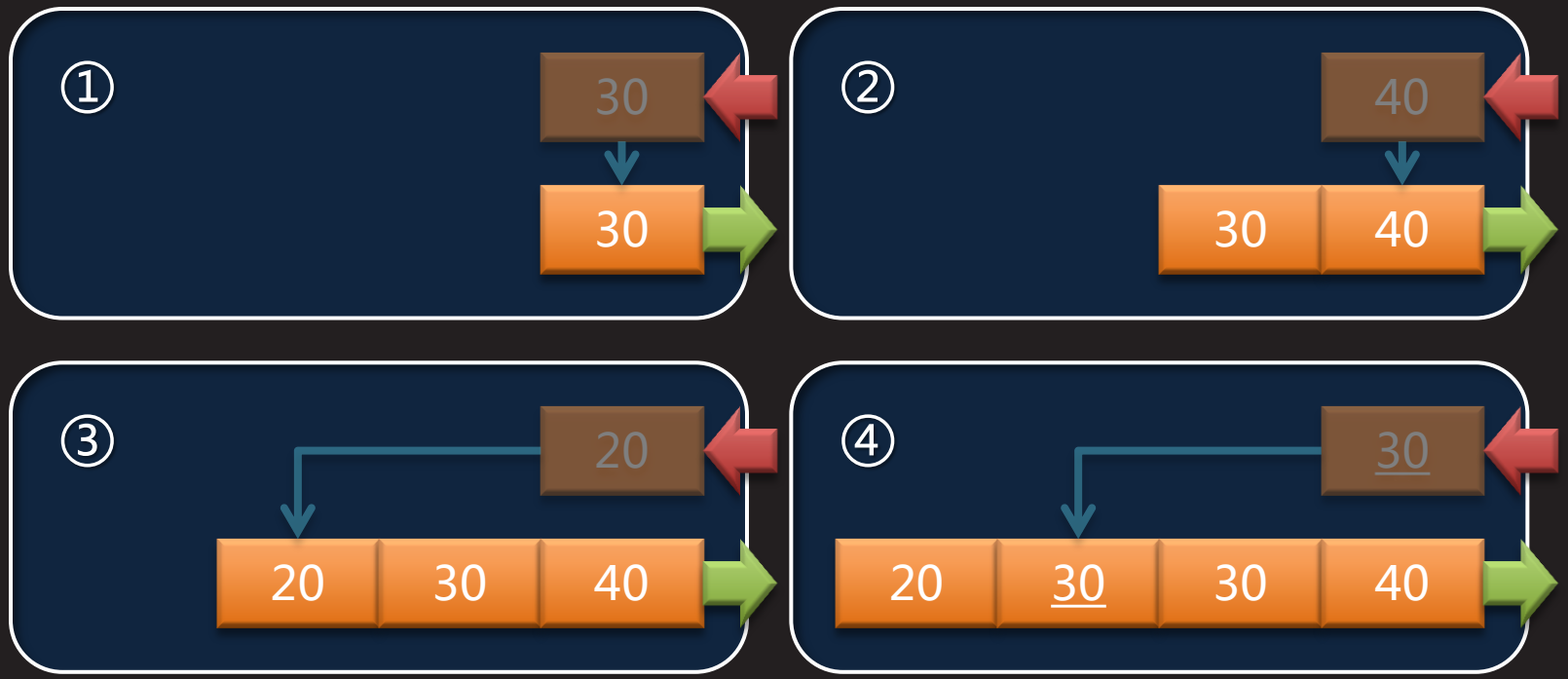
- 队列

Technology Tarena 达内科技

优先队列

压入元素大者优先

- 每当向优先队列中压入一个元素，队列中具有最高优先级的元素会被自动排至队首(类似插入排序)。因此从优先队列中弹出元素，既不是先进先出，也不是后进先出，而是优者先出，即谁的优先级高谁先出队。只有那些优先级相同的元素才遵循和队列一样的先进先出规则



知识讲解

用小于运算符比大小

- 缺省情况下，优先队列利用元素类型的“<”运算符比较大小，谁大谁的优先级高
  - class Student {
    public:
    Student (string const& name = "", int age = 0,
    int score = 0) : m_name (name), m_age (age),
    m_score (score) {}
    bool operator< (Student const& rhs) const {
    return m_age < rhs.m_age; } // 年长者优
    private:
    string m_name;
    int m_age, m_score;
    };
  - priority_queue<Student, vector<Student> > ps;

知识讲解

用小于比较器比大小

- 也可以通过小于比较器比较大小，确定优先级高低
  - class Student {
    public:
    Student (string const& name = "", int age = 0, int score = 0) :
    m_name (name), m_age (age), m_score (score) {}
    private:
    string m_name; int m_age, m_score;
    friend class Less; };
  - class PrioritizeByScore {
    public:
    bool operator() (Student const& sa,
    Student const& sb) const {
    return sa.m_score < sb.m_score; } }; // 分高者优
  - priority_queue<Student, vector<Student>,
    PrioritizeByScore> ps;

知识讲解

底层容器的类型

- 定义优先队列容器时可以指定底层容器的类型
  - priority_queue<string, vector<string> > ps;
    注意两个“>”之间的至少要留一个空格，否则编译器会把“>>”理解为右移运算符，导致编译错误
- 三种线性容器中除列表以外的任何一种都可以作为优先队列的底层容器，其中双端队列是缺省底层容器
  - priority_queue<int> pi;
- 不能用列表适配优先队列的原因是，它缺少按优先级调整元素顺序时所需要的随机迭代器

知识讲解

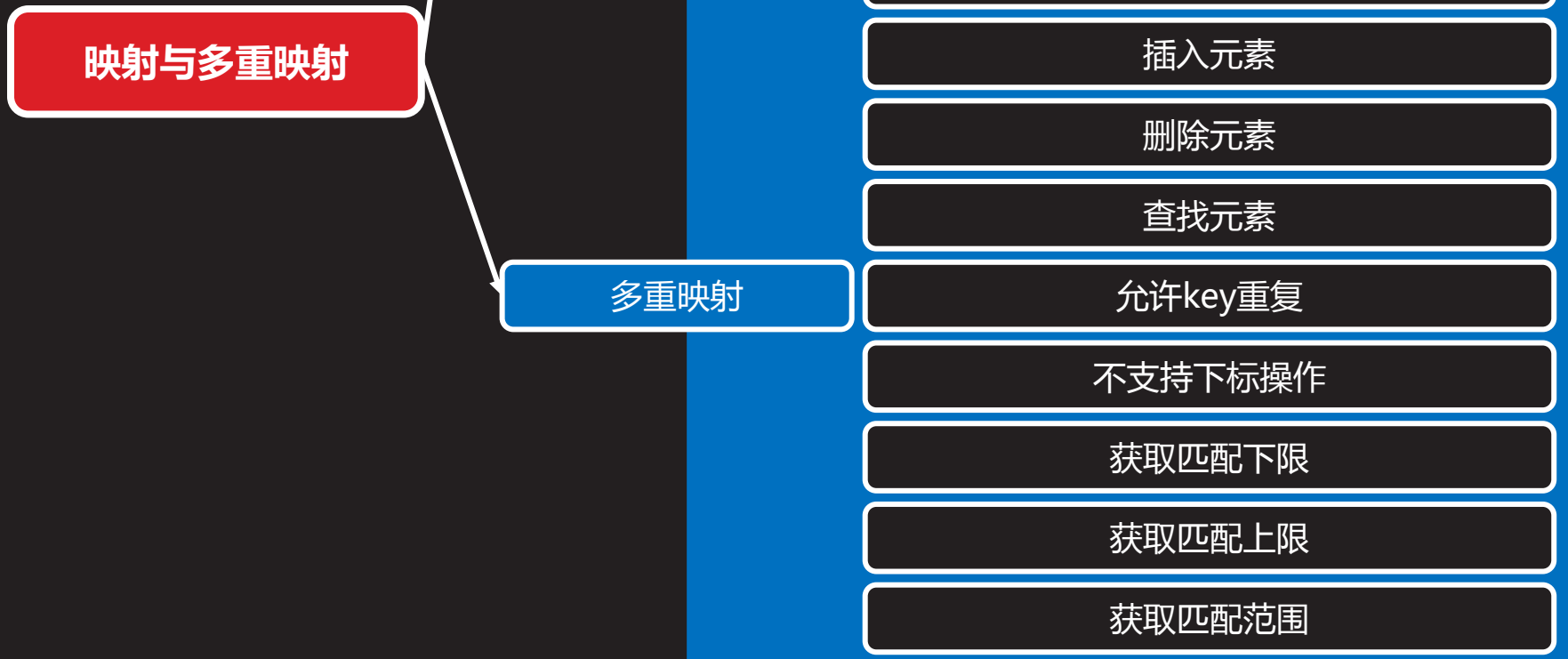
优先队列

【参见：TTS COOKBOOK 】

课堂练习

- 优先队列

映射与多重映射


映射与多重映射
映射
key-value对
红黑树
根据key找value
以key为索引的下标操作
key类型的小于运算符
key类型的小于比较器
基本访问单元是pair
插入元素
删除元素
查找元素
多重映射
允许key重复
不支持下标操作
获取匹配下限
获取匹配上限
获取匹配范围

Technology Tarena 达内科技

映射

key-value对

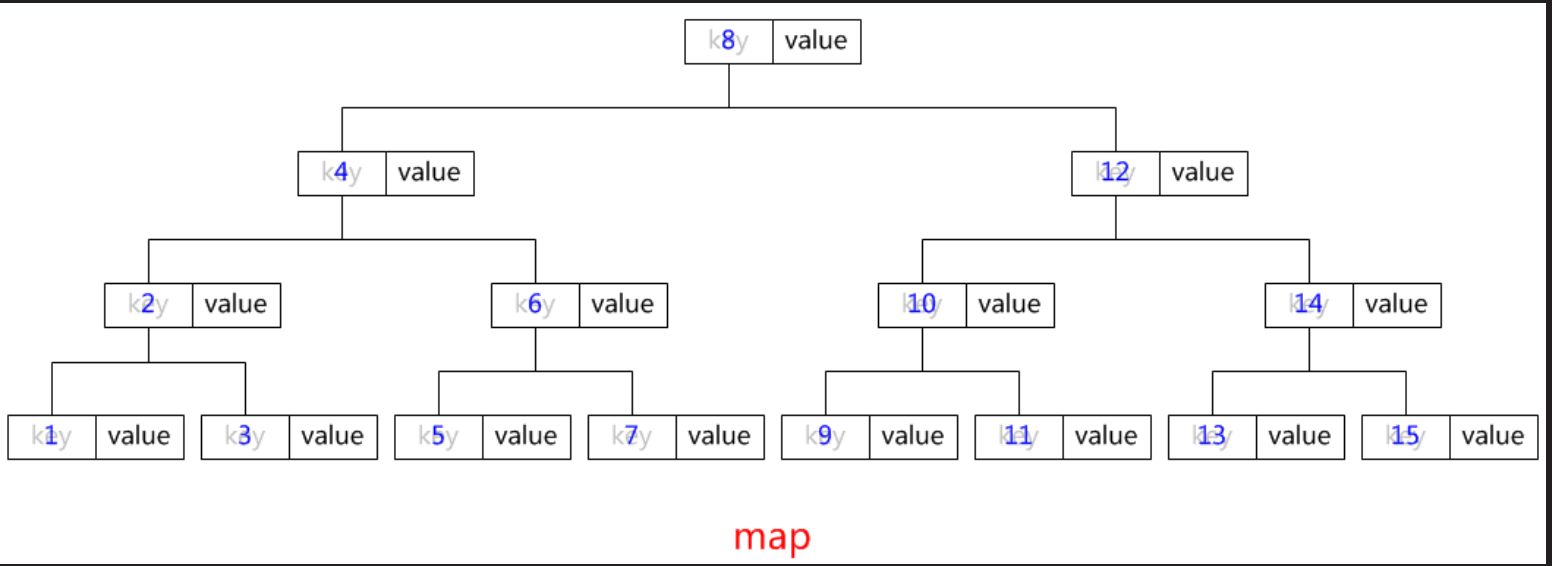
- 映射将现实世界中不同类型事物间的对应关系，抽象为由一系列key-value(键-值)对组成的集合，并提供通过key快速检索与之相对应的value的方法
  - 居民身份证号码 —> 个人信息
  - 学号 —> 考试成绩
  - 姓名 —> 个人简历
  - 条形码 —> 商品价格
  - 主键 —> 数据库表记录

知识讲解

红黑树

- 在映射内部，所有的key-value对被组织成以key为基准的红黑树(自平衡有序二叉树)的形式
- 映射中的key必须是唯一的，表达一一对应的逻辑关系

知识讲解



根据key找value

- 基于有序二叉树的结构特征，映射可以在对数时间(O(logN))内，根据key找到与之相对应的value。因此映射主要被应用在需要高速检索特性的场合
- 具有平衡有序二叉树特征的红黑树，相较于一般有序二叉树的特殊之处在于，其任何节点的左右子树大小之差不会超过1。这样做既有利于保证其搜索性能的均化，又可以有效避免单连枝树向链表退化的风险
- 当由于向红黑树中插入或删除节点而导致其平衡性遭到破坏时，必须根据具体情况对树进行一次或多次的单旋或双旋，直至其恢复平衡。因此映射的构建和修改通常较耗时，不适于存储频繁变化的key

知识讲解

以key为索引的下标操作

- 映射支持以key为索引的下标操作
  - value_type& operator[] (key_type const& key);
  - 返回与参数key相对应的value的引用，若key不存在，则新建一个key-value对，其中的value按缺省方式初始化
- 映射的下标运算符既可用于插入又可用于检索
  - map<string, int> msi;
    msi["张飞"] = 10000; // 插入
    msi["张飞"] = 20000; // 检索
    cout << msi["张飞"] << endl; // 20000

知识讲解

key类型的小于运算符

- 映射内部为了维护其二叉树结构的有序性，必然需要对key做大小比较。缺省情况下这项工作由key类型的“<”运算符来完成
  - map<string, int> msi;
    ++msi["tarena"];
    ++msi["TARENA"];
    ++msi["Tarena"];
    cout << msi["tarena"] << endl; // 1
  - 注意string类型的“<”运算符是大小写敏感的

知识讲解

key类型的小于比较器

- 也可以通过针对key类型的小于比较器自定义排序规则
  - class StrCaseCmp {
    public:
      bool operator() (string const& sa,
        string const& sb) const {
        return strcasecmp (sa.c_str (), sb.c_str ()) < 0;
      }
    };
  - map<string, int, StrCaseCmp> msi;
    ++msi["tarena"];
    ++msi["TARENA"];
    ++msi["Tarena"];
    cout << msi["tarena"] << endl; // 3

基本访问单元是pair

- 映射的基本访问单元和存储单元都是pair，pair只有两个成员变变量，first和second，分别表示key和value
  - template<typename first_type, typename second_type>
    class pair {
    public:
        pair (first_type const& first, second_type const& second) :
            first (first), second (second) {}
        first_type first;
        second_type second;
    };
  - map<string, int> msi;
    msi.insert (pair<string, int> ("张飞", 10000));
    msi.insert (make_pair ("赵云", 20000));

知识讲解

插入元素

- pair<iterator, bool> insert (
  pair<key_type, value_type> const& pair);
  - 插入位置由映射根据key的有序性确定
  - 插入成功，返回pair的second成员为true，first成员为指向新插入元素的迭代器
  - 插入失败，返回pair的second成员为false，first成员未定义
- 例如
  - map<string, int> msi;
    typedef map<string, int>::iterator IT;
    ...
    pair<IT, bool> res = msi.insert (pair<string, int> ("张飞", 10000));
    if (! res.second) cout << "插入失败！" << endl;
    else cout << res.first->first << "->" << res.first->second << endl;

知识讲解

删除元素

- 删除特定位置的元素
  - void erase (iterator pos);
- 删除特定范围的元素
  - void erase (iterator begin, iterator end);
- 删除特定key的元素
  - size_type erase (key_type const& key);

知识讲解

查找元素

- iterator find (key_type const& key);
  - 查找与key匹配的元素，对数时间复杂度O(logN)，成功返回指向匹配元素的迭代器，失败返回终止迭代器
  - 与全局线性查找函数find()不同，映射采用基于树的查找，并不需要key的类型支持“==”运算符，仅支持“<”运算符或小于比较器即可
    - if (目标key < 节点key)
      在该节点的左子树中继续查找;
      else
      if (节点key < 目标key)
      在该节点的右子树中继续查找;
      else
      该节点即为匹配节点，查找成功！

知识讲解

映射

【参见：TTS COOKBOOK 】

课堂练习

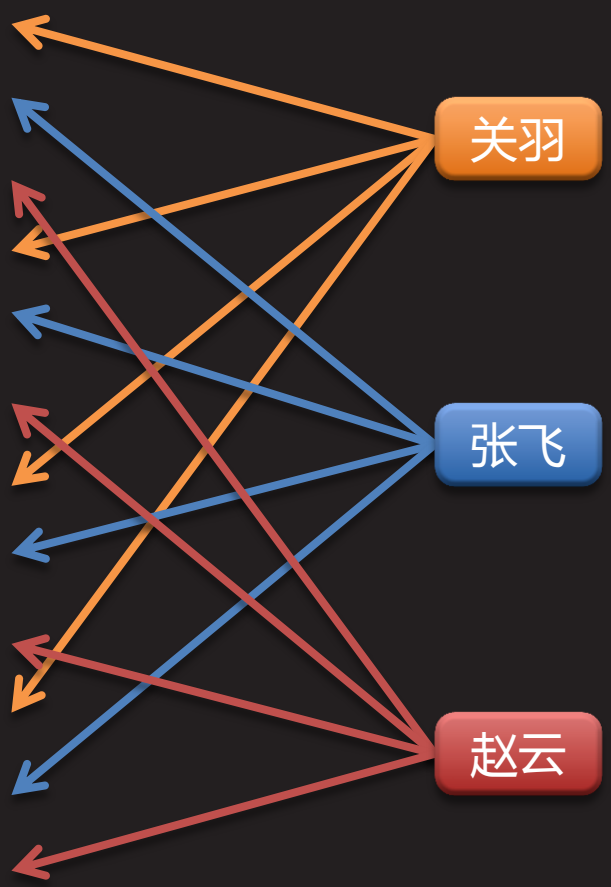
- 映射

Technology Tarena 达内科技

多重映射

允许key重复

- 允许key重复的映射即为多重映射，表示一对多的逻辑关系
  - 销售员—>季度销售额
    - 关羽 -> 10000
    - 张飞 -> 15000
    - 赵云 -> 20000
    - 关羽 -> 25000
    - 张飞 -> 30000
    - 赵云 -> 35000
    - 关羽 -> 40000
    - 张飞 -> 45000
    - 赵云 -> 50000
    - 关羽 -> 55000
    - 张飞 -> 60000
    - 赵云 -> 65000

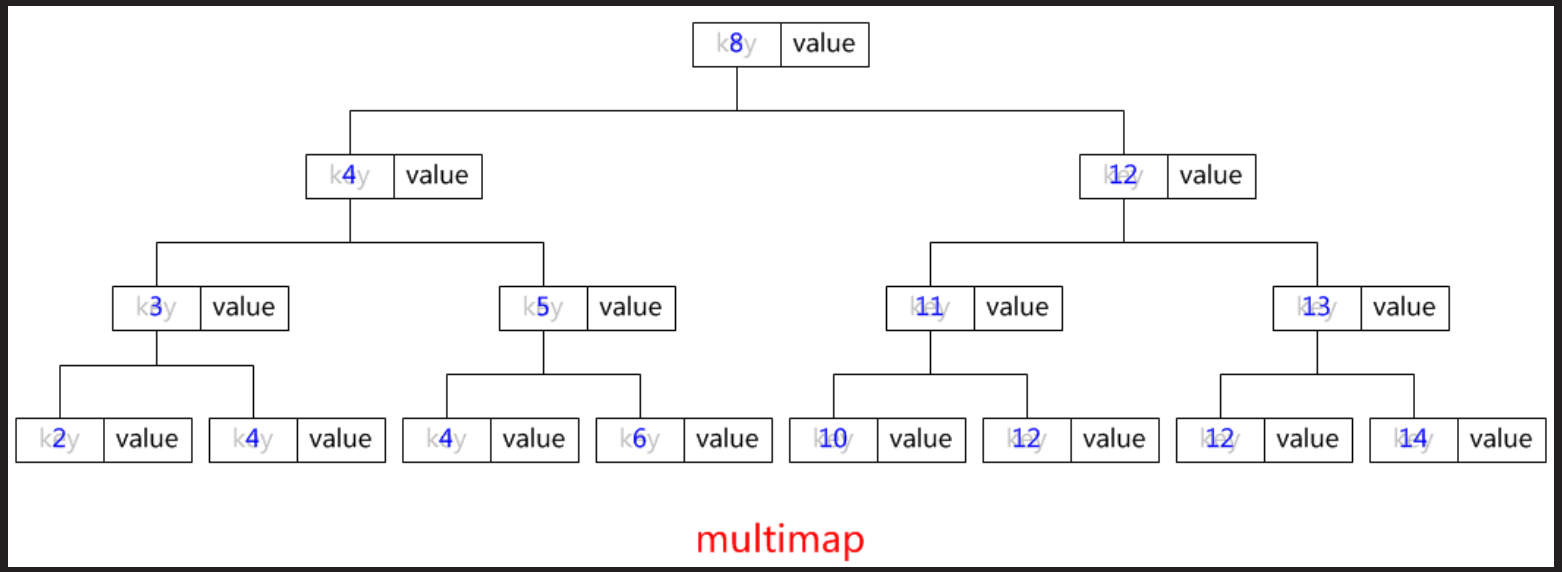


知识讲解

不支持下标操作

- 多重映射无法根据给定的key唯一确定与之对应的value，因此多重映射不支持以key为索引的下标操作

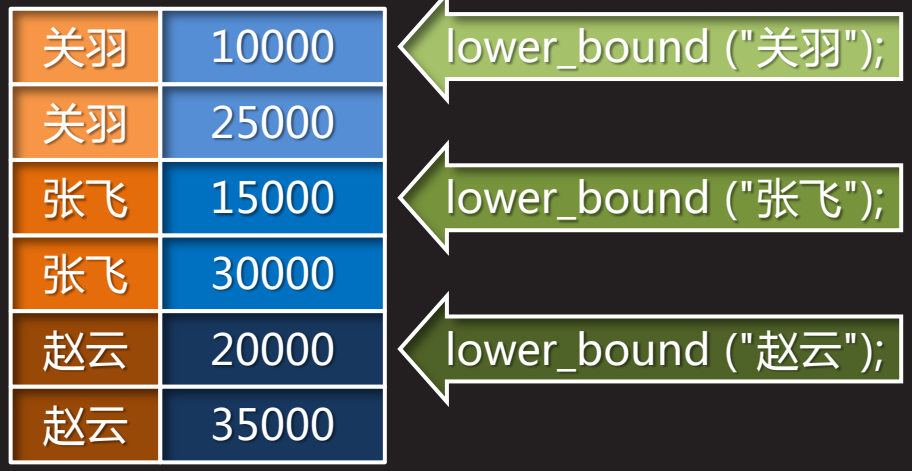
知识讲解



获取匹配下限

- iterator lower_bound (key_type const& key);
  - 返回指向中序遍历中第一个键大于或等于key的元素的迭代器
- 例如
  - multimap<string, int> msi;

```
msi.insert (make_pair (
    "关羽", 10000));
msi.insert (make_pair (
    "张飞", 15000));
msi.insert (make_pair (
    "赵云", 20000));
msi.insert (make_pair (
    "关羽", 25000));
msi.insert (make_pair ("张飞", 30000));
msi.insert (make_pair ("赵云", 35000));
```

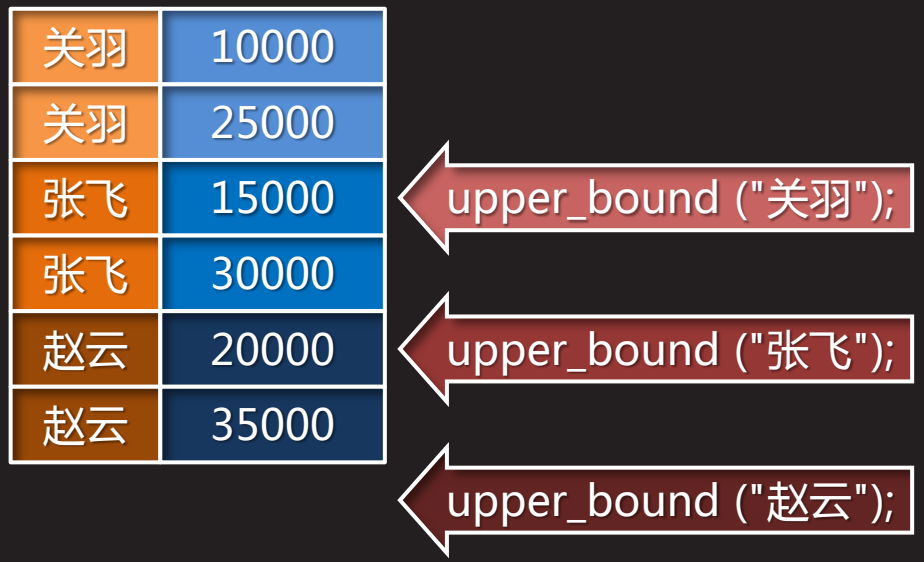


知识讲解

获取匹配上限

- iterator upper_bound (key_type const& key);
  - 返回指向中序遍历中第一个键大于key的元素的迭代器
- 例如
  - multimap<string, int> msi;

```
msi.insert (make_pair (
    "关羽", 10000));
msi.insert (make_pair (
    "张飞", 15000));
msi.insert (make_pair (
    "赵云", 20000));
msi.insert (make_pair (
    "关羽", 25000));
msi.insert (make_pair ("张飞", 30000));
msi.insert (make_pair ("赵云", 35000));
```



知识讲解

获取匹配范围

- pair<iterator, iterator> equal_range (
  key_type const& key);
  - 返回由匹配下限迭代器(first)和匹配上限迭代器(second)组成的pair对象
- 例如
  - multimap<string, int> msi;
    ...
    typedef multimap<string, int>::iterator IT;
    pair<IT, IT> er = msi.equal_range ("张飞");
    for (IT it = er.first; it != er.second; ++it)
    cout << it->first << "->" << it->second << endl;

知识讲解

多重映射

【参见：TTS COOKBOOK 】

课堂练习

- 多重映射

Technology Tarena 达内科技

总结和答疑