> To make it easier for your implementation, you can just add code to class defined in `iterator.hpp`

题目描述

要访问顺序容器和关联容器中的元素，需要通过"迭代器（iterator）"进行。迭代器是一个变量，相当于容器和操纵容器的算法之间的中介。迭代器可以指向容器中的某个元素，通过迭代器就可以读写它指向的元素。所以，你可以将迭代器理解成遍历容器中的元素的一种抽象。从这一点上看，迭代器和指针类似。

如果你理解了实验内容的话，**这次实验的主体部分应该比较容易。**

Why we use iterators?——迭代器的介绍

如果觉得中文不够清晰，可以阅读附录的英文描述。当然内容都差不多。

之前的迭代器遍历：

```
for(std::vector<T>::iterator it = vec.begin(); it != vec.end(); ++it) {
    std::cout << *it << '\n';
}
```

C++11 中新的遍历方式：

```
for(const auto & elem : vec) {
    std::cout << elem << '\n';
}
```

可以减少出错的机会：不用重新输入一遍容器的类型；以及当容器类型发生改变时，不需要修改上述代码。

另外迭代器遍历与容器无关的特性，简化了函数的实现，比如排序`sort`。

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );
```

`sort`可以作用于任何`random access iterators`（随机访问迭代器:

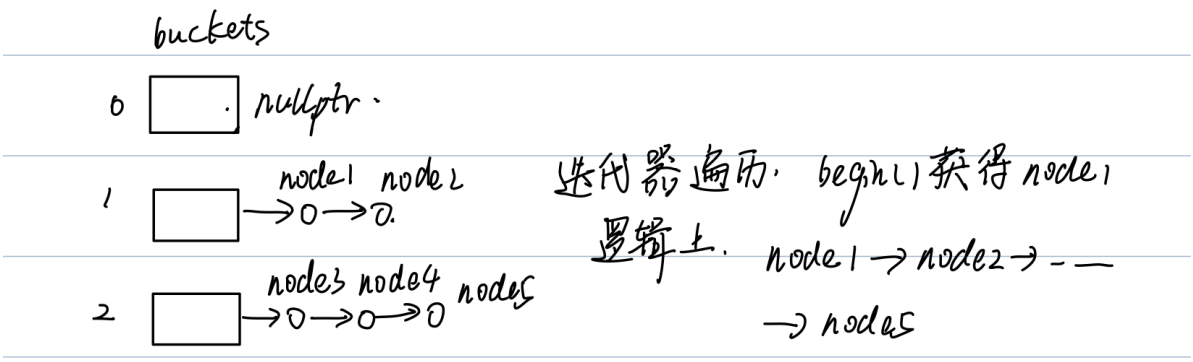
- `C arrays`
- `std::array`
- `std::vector`
- `std::deque`
- `std::string (which can be treated as a container of char)`
- 流行第三方库：`boost::ptr_vector`, `boost::ptr_deque`, etc.
- 甚至你自己的容器

所以说，本次实验希望大家了解一下迭代器的实现，增加一下对它的了解。

上机内容介绍

迭代器的逻辑可以视作与下面这段代码类似，上次上机我们实现的`hashmap`：`key value`对经过`hash`，散列到`buckets`中，每个`bucket`都是一个链表。

那这次我们想要实现的迭代器，就是将`hashmap`底层的一段一段链表在遍历的时候看做是**逻辑上的一整条链表**：`begin()`获得`hashmap`中的第一个`key value`对，`iterator++`获得下一个`key value`对，直到`end()`。



这里的获得并不是说迭代器指向`node1`，而是指迭代器中的`cur_node`指向`node1`。

私有成员变量	简要说明
const HashMap*hashMap;	迭代器对应的hashmap
bool is_end = true;	指示当前迭代器对应是否为end()
int index = 0;	hashmap中第几个bucket
node*curr_node= nullptr;	当前迭代器对应的node

```
for(int i=0;i<hashMap->bucket_count();++i){
    auto curr_node = hashMap->_buckets_array[i];
    //对应于上图buckets[0]=nullptr
    while(curr_node != nullptr){
        ...
        curr_node = curr_node -> next;
    }
}
```

接口	说明
`iterator(const HashMap*mp,bool end)`	constructor, you need to initlize private menbers in this function
`iterator(const iterator&it)`	constructor, useful for you implementation in `iterator operator++(int)`
`iterator&operator++()`	let curr_node point to the next node and return curr_node
`iterator operator++(int)`	return the curr_node and let curr_node point to the next node
`iterator&operator=(iterator&other)`	I think this does the same as constructor
`iterator&operator=(const iterator&other)`	above
`==,!=`	judge whether two iterators are the same. If they both point to the end position or the same node,they are the same.
`value_type& operator*()`	return value stored in curr_node
`value_type*operator->()`	return address of value stored in curr_node
`bool key_equal(const K&_key)`	judge whether key of curr_node is _key

Tips:

- 构造函数的`end`的作用：

```
iterator begin(){
    return iterator(this,false);
}
iterator end(){
    return iterator(this,true);
}
```

  构造函数需要保证`cur_node`指向它应该指向的位置，而`cur_node`保存的其实是迭代器应该指向的`node`。

  注意需要考虑如果`hashmap`所有`buckets`全空的情况，这个时候`is_end`和`cur_node`应该怎么变化?

提交要求

- 提交源码文件：`iterator.hpp`,直接打包成zip格式的压缩包。**不要添加其他任何目录**
- 文件的编码格式只支持utf-8。
- 请严格按照给定的接口进行编码,否则无法调用测试用例。
- **提交的源码文件中不需要包含main函数,否则无法通过编译。**

附录：

Why we use iterators?

Well, first off, if you're reading some kind of book or tutorial that suggests doing something like:

```
for(std::vector<T>::iterator it = vec.begin(); it != vec.end(); ++it) {
  std::cout << *it << '\n';
}
```

...you should probably get a newer book because as of C++11 the preferred way is with a range-based for loop:

```
for(const auto & elem : vec) {
  std::cout << elem << '\n';
}
```

Not only is it less to type, there's fewer opportunities for mistakes. For example, you don't have to repeat the type of the vector, so if it changes there's nothing to update.

But that aside, the reason for using iterators is that they abstract away the differences between containers. Take a look at any of the algorithms of the standard library. These all deal in iterators, which makes them container-agnostic. Take sorting for instance:

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );
```

This one function can work with any sequence container that has random access iterators, which includes:

* `C arrays`

* `std::array`

* `std::vector`

* `std::deque`

* `std::string (which can be treated as a container of char)`

* popular third party libraries, e.g. `boost::stable_vector`, `boost::ptr_vector`, `boost::ptr_deque`, etc.

* your own containers

That's a pretty remarkable feat. Imagine if you had to have a different version of std::sort() for every different kind of container. It would be a mess, and it wouldn't be expandable because the implementation that comes with your standard library wouldn't be able to work with custom types. Iterators are what make this possible. (And that example is one of the milder cases, since std::sort requires random access. There are numerous algorithms that only require forward iterators and work with many more containers like std::list or std::map, and so on.)

And by the way, all of the above applies equally to the range-based for loop, which is really just syntactic sugar on top of the traditional method of iterating over a container using iterators.

So, I think it's of great importance to get a glimpse of how to implement the iterator and I hope this lab will be helpful for your future study.

about this lab

Because I believe what I cannot create I don't understand, so I want you to know how to add a iterator for your own container. I will explaine them to you below.

you can just think that we can iterate a HashMap by the order defined below:

```
for(int i=0;i<hashMap->bucket_count();++i){
  auto curr_node = hashMap->_buckets_array[i];
  while(curr_node != nullptr){
    ...
    curr_node = curr_node -> next;
  }
}
```