实现自己的IQueue

1. 实现queue普通队列
2. 实现priority_queue优先队列
3. 实现【优先队列】自定义优先规则。

- **当定义了一个局部功能的宏的时候，出了函数范围一定记得把他undef掉。**

```
#include<iostream>
#include<functional>
using namespace std;
namespace haizei{


//队列的接口类
class IQueue {
public:
    virtual void push(int) = 0;
    virtual void pop() = 0;
    virtual bool empty() const = 0;
    virtual int top() const = 0;
    virtual int front() const = 0;
    virtual int size() const = 0;
    virtual ~IQueue() {};
};

class queue : public IQueue {
private:
    friend ostream &operator<<(ostream &, const queue &);
    int *data;
    int head, tail, count, __size;
    void expand() {
        queue q(2 * __size);
        while (!empty()) {
            q.push(front());
            pop();
        }
        this->swap(q);
        return ;
    }
    int top() const override { return data[head]; }
public:
    queue(int n = 10) : data(new int[n]), head(0), tail(0), count(0), __size(n) {}
    void push(int val) override {
        if (count == __size) {
            expand();
        }
        data[tail++] = val;
        tail %= __size;
        count += 1;
        return ;
    }
    void pop() override {
        if (empty()) return ;
        head += 1;
        head %= __size;
        count -= 1;
        return ;
    }
    bool empty() const override { return count == 0; }
    int front()  const override { return data[head];  }
    int size()   const override { return count; }
    void swap(queue &q) {
        std::swap(this->data, q.data);
        std::swap(this->head, q.head);
        std::swap(this->tail, q.tail);
        std::swap(this->count, q.count);
        std::swap(this->__size, q.__size);
        return ;
    }
    ~queue() {
        if (data) delete[] data;
        return ;
    }
};
//优先队列：弹出最大值
class priority_queue : public IQueue {
private:
    friend ostream &operator<<(ostream &, const priority_queue &);
    typedef function<bool(int, int)> CMP_T;
    int *raw_data, *data;
    int count, __size;
    CMP_T cmp;
    void expand() {
        priority_queue q(cmp, 2 * __size);
        while (!empty()) {
            q.push(front());
            pop();
        }
        this->swap(q);
        return ;
    }
    void up_maintain(int ind) {
        if (ind == 1) return ;
        if (cmp(data[ind >> 1], data[ind])) {
            std::swap(data[ind], data[ind >> 1]);
            up_maintain(ind >> 1);
        }
        return ;
    }
    void down_maintain(int ind) {
        #define LIND(i) (i << 1)
        #define RIND(i) (i << 1 | 1)
        if (LIND(ind) > count) return ;
        int temp = ind;
        if (cmp(data[temp], data[LIND(ind)])) {
            temp = LIND(ind);
        }
        if (RIND(ind) <= count && cmp(data[temp], data[RIND(ind)])) {
            temp = RIND(ind);
        }
        #undef LIND
        #undef RIND
        if (temp == ind) return ;
        std::swap(data[ind], data[temp]);
        down_maintain(temp);
        return ;
    }
    int front()  const override { return data[1];  }
public:
    priority_queue(CMP_T cmp = less<int>(), int n = 10) : raw_data(new int[n]), data(raw_data - 1)
    void push(int val) override {
        if (count == __size) {
            expand();
        }
        count += 1;
        data[count] = val;
        up_maintain(count);
        return ;
    }
    void pop() override {
        if (empty()) return ;
        data[1] = data[count];
        count -= 1;
        down_maintain(1);
        return ;
    }
    bool empty() const override { return count == 0; }
    int top() const override { return data[1]; }
    int size()   const override { return count; }
    void swap(priority_queue &q) {
        std::swap(this->raw_data, q.raw_data);
        std::swap(this->data, q.data);
        std::swap(this->count, q.count);
        std::swap(this->__size, q.__size);
        return ;
    }
    ~priority_queue() {
        if (raw_data) delete[] raw_data;
        return ;
    }
};
ostream &operator<<(ostream &out, const queue &q) {
    out << "queue : ";
    for (int i = 0, j = q.head; i < q.count; i += 1, j += 1) {
        j %= q.__size;
        out << q.data[j] << " ";
    }
    return out;
}
ostream &operator<<(ostream &out, const priority_queue &q) {
    out << "priority_queue : ";
    for (int i = 0; i < q.count; i += 1) {
        out << q.raw_data[i] << " ";
    }
    return out;
}

}
bool cmp(int a, int b) {
    return a > b;
}
int main() {
    int op, val;
    haizei::queue q1;
    haizei::priority_queue q2;
    haizei::priority_queue q3(cmp);
    while (cin >> op) {
        switch(op) {
            case 0: {
                cin >> val;
                q1.push(val);
                q2.push(val);
                q3.push(val);
            } break;
            case 1: {
                cout << "queue front : " << q1.front() << endl;
                cout << "priority(less) top : " << q2.top() << endl;
                cout << "priority(greater) top : " << q3.top() << endl;
                q1.pop();
                q2.pop();
                q3.pop();
            } break;
        }
        cout << q1 << endl;
        cout << q2 << endl;
        cout << q3 << endl;
    }
    return 0;
}
```

实现哈希表

> 课后练习
>
> 题目：实现一个可以自定义哈希函数的哈希表类
>
> 哈希函数包括：函数、函数对象、lambda表达式
>
> 要求：定义和声明的分离。

底层实现为哈希表的数组对象。

有两大类可以快速地索引数据结构，一类就是哈希表，一类是红黑树，平衡二叉排序树。
平衡二叉排序树可以维护数据之间地有序性。
而哈希表完全没有必要照顾到数据地有序性。换句话说，他俩就这点区别。

我们也可以实现一个底层实现为平衡二叉排序树的数组对象。

```
#include<iostream>
#include<functional>
#include<vector>
using namespace std;

//链表节点类
class Node {
public:
    Node() = default;
    Node(string, int, Node *);
    string key();
    int value;
    Node *next();
    void set_next(Node *);
    void insert(Node *);
    void erase_next();
private:
    string __key;
    Node *__next;
};

Node::Node(string key, int value, Node *next = nullptr) : __key(key), value(value), __next(next) {

string Node::key() { return __key; }

Node *Node::next() { return __next; }

void Node::set_next(Node *next) {
    __next = next;
    return ;
}

void Node::insert(Node *node) {
    node->set_next(this->next());
    this->set_next(node);
    return ;
}

void Node::erase_next() {
    Node *p = this->next();
    if (p == nullptr) return ;
    this->set_next(this->next()->next());
    delete p;
    return ;
}
//完成对Node的封装
```

```
class HashTable {
public:
    typedef function<int(string)> HASH_FUNC_T;
    HashTable(HASH_FUNC_T hash_func, int size);
    bool insert(string, int);
    bool erase(string);
    bool find(string);
    int capacity();
    int &operator[](string);
    void swap(HashTable &h);
    ~HashTable();
private:
    Node *__insert(string, int);
    Node *__find(string);
    void __expand();
    int __size, data_cnt;
    vector<Node> data;
    HASH_FUNC_T hash_func;
};

HashTable::HashTable(HASH_FUNC_T hash_func, int size = 10) : hash_func(hash_func), data(size), __s

bool HashTable::insert(string key, int value = 0) {
    Node *p = __insert(key, value);
    if (data_cnt > __size * 2) __expand();
    return p != nullptr;
}
Node *HashTable::__insert(string key, int value) {
    if (find(key)) return nullptr;
    int ind = hash_func(key) % __size;
    Node *p;
    data[ind].insert((p = new Node(key, value)));
    data_cnt += 1;
    return p;
}

bool HashTable::erase(string key) {
    int ind = hash_func(key) % __size;
    Node *p = &data[ind];
    while (p->next() && p->next()->key() != key) p = p->next();
    if (p->next() == nullptr) return false;
    p->erase_next();
    data_cnt -= 1;
    return true;
}
```

```
bool HashTable::find(string key) {
    return __find(key) != nullptr;
}

Node *HashTable::__find(string key) {
    int ind = hash_func(key) % __size;
    Node *p = data[ind].next();
    while (p && p->key() != key) p = p->next();
    return p;
}

void HashTable::swap(HashTable &h) {
    std::swap(__size, h.__size);
    std::swap(data_cnt, h.data_cnt);
    std::swap(data, h.data);
    std::swap(hash_func, h.hash_func);
    return ;
}
HashTable::~HashTable() {
    for (int i = 0; i < __size; i++) {
        while (data[i].next()) data[i].erase_next();
    }
    return ;
}
void HashTable::__expand() {
    HashTable h(hash_func, 2 * __size);
    for (int i = 0; i < __size; i++) {
        Node *p = data[i].next();
        while (p) {
            h.insert(p->key(), p->value);
            p = p->next();
        }
    }
    this->swap(h);
    return ;
}

int HashTable::capacity() { return data_cnt; }
int &HashTable::operator[](string key) {
    Node *p = __find(key);
    if (p) return p->value;
    insert(key, 0);
    return __find(key)->value;
}
```

```cpp
int BKDRHash(string s) {
    int seed = 31;
    int h = 0;
    for (int i = 0; s[i]; i++) {
        h = h * seed + s[i];
    }
    return h & 0x7fffffff;
}
class APHash_Class {
public:
    int operator()(string s) {
        int h = 0;
        for (int i = 0; s[i]; i++) {
            if (i % 2) {
                h = (h << 3) ^ s[i] ^ (h >> 5);
            } else {
                h = ~((h << 7) ^ s[i] ^ (h >> 11));
            }
        }
        return h & 0x7fffffff;
    }
};

int main() {
    APHash_Class APHash;
    HashTable h1(BKDRHash);
    HashTable h2(APHash);
    int op;
    string s;
    cout << h1.capacity() << endl;
    cout << h2.capacity() << endl;
    h1["hello"] = 123;
    h1["world"] = 456;
    h1["haizei"] = 789;
    cout << h1.capacity() << endl;
    cout << h2.capacity() << endl;
    cout << h1["hello"] << " " << h1["world"] << " " << h1["hahaha"] << endl;
    while (cin >> op >> s) {
        switch(op) {
            case 0: {
                cout << "insert" << s << "to hash table 1 = ";
                cout << h1.insert(s) << endl;
                cout << "insert" << s << "to hash table 2 =";
                cout << h2.insert(s) << endl;
            } break;
            case 1: {
                cout << "erase" << s << "from hash table 1 = ";
                cout << h1.erase(s) << endl;
                cout << "erase" << s << "from hash table 2 = ";
                cout << h2.erase(s) << endl;
            } break;
            case 2: {
                cout << "find" << s << "at hash table 1 = ";
                cout << h1.find(s) << endl;
                cout << "find" << s << "at hash table 2 = ";
                cout << h2.find(s) << endl;
            } break;
        }
    }
    return 0;
}
```