IntervalHeap

算法简介

初始化

直接用插入操作初始化。

插入

找到完全二叉树最后一个节点，如果还没满就插入，插入后对该节点调整区间左右端点。

最后一个节点已满，则新建一个节点插入。

全过程中保证完全二叉树性质。

插入后，自底向上调整区间左右端点。

删除

最大和最小的删除方式类似。这里以最大为例。

弹出堆顶的最大值，从左右儿子拉一个最大值到堆顶，接着是左右儿子的删除操作，递归进行。

递归到最后一个点时，需要从完全二叉树中找到最后一个节点，拉一个值过来补位，接着再自底向上调整区间左右端点。

查询

直接查询堆顶即可。

实现

核心思想声明

- 使用 `vector<array<int, 2>>` 来动态保存完全二叉树，`rt` 号节点的左右儿子分别为 `rt * 2` 和 `rt * 2 + 1`，`rt` 号节点的数据存储在 `a[rt-1]` 中。
- private 函数内部采用 0-index, 除了 `insert_element` 外，public 函数采用 1-index，但是 private 函数参数均采用 1-index。

核心函数解释

```
bool swap_node(int son, int fa); // 对于 fa-son 父子链，合理交换数据保证 IntervalHeap 的区间包含性质
void access(int rk); // 从 rk 号节点开始到根自底向上调整区间保证 IntervalHeap 的区间包含性质
void validate_node(int rk); // 对于 rk 号节点，检查区间左右端点大小关系，如果不满足，交换左右端点。
int pop_back(); // 弹出二叉树节点上的最后一个元素用于补位，返回这个元素的值。
```

源代码

```
#include <bits/stdc++.h>
using namespace std;

class IntervalHeap {
private:
    vector<array<int, 2>> a;
    int cnt;
    bool swap_node(int son, int fa){
        // do swap if intersection
        son--, fa--; // real index
        if(a[son][0] < a[fa][0])
            return swap(a[son][0], a[fa][0]), true;

        // max value cross are two cases
        if(is_full_node(son + 1)){
            if(a[son][1] > a[fa][1])
                return swap(a[son][1], a[fa][1]), true;
        }
        else{
            if(a[son][0] > a[fa][1])
                return swap(a[son][0], a[fa][1]), true;
        }
        return false;
    }
    bool exist(int x){
        return x <= (cnt + 1) / 2;
    }
    bool is_last_node(int rk){
        return rk == (cnt + 1) / 2;
    }
    bool is_full_node(int rk){
        return !is_last_node(rk) || cnt % 2 == 0;
    }
    void access(int rk){
        // call for any node
        // access to root to make the path validated
        while(rk != 1){
            swap_node(rk, rk >> 1);
            rk >>= 1;
        }
    }
    int get_min(int rk){
        return a[rk - 1][0];
    }
    int get_max(int rk){
        return is_full_node(rk) ? a[rk - 1][1] : a[rk - 1][0];
    }
    void set_min(int rk, int val){
        a[rk - 1][0] = val;
```

```
    }
    void set_max(int rk, int val){
        a[rk - 1][1] = val;
    }
    void validate_node(int rk){
        // DO NOT CALL THIS FOR THE LAST NODE
        // when replace the value from the last node, we need to validate it
        rk--;
        if(a[rk][0] > a[rk][1])
            swap(a[rk][0], a[rk][1]);
    }
    int pop_back(){
        int rk = (cnt + 1) / 2;
        // perserve val before do delete because get_min depends on cnt
        if(cnt % 2){
            int val = get_min(rk);
            return cnt--, a.pop_back(), val;
        }
        else{
            int val = get_max(rk);
            return cnt--, val;
        }
    }
public:
    IntervalHeap(){
        cnt = 0;
    }
    void Init(const vector<int> &other){
        a.clear(); cnt=0;
        for(auto x:other)
            insert_element(x);
    }
    void remove_min(){
        int rk=1;
        while(exist(rk << 1)){
            if(exist(rk << 1 | 1) && get_min(rk << 1 | 1) < get_min(rk << 1)){
                rk = rk << 1 | 1;
            }else{
                rk <<= 1;
            }
            set_min(rk >> 1, get_min(rk));
        }
        if(is_last_node(rk)){
            int val = pop_back();
            if(cnt % 2) // if it's interval, max_value will be set as the min_value
                set_min(rk, val);
        }
        else{
            set_min(rk, pop_back());
            validate_node(rk);
            access(rk);
        }
```

```
    }
    void remove_max(){
        int rk=1;
        while(exist(rk << 1)){
            if(exist(rk << 1 | 1) && get_max(rk << 1 | 1) > get_max(rk << 1)){
                rk = rk << 1 | 1;
            }else{
                rk <<= 1;
            }
            set_max(rk >> 1, get_max(rk));
        }
        if(is_last_node(rk)){
            pop_back(); // max_value is moved(pop_back it safely either it's
interval or single value)
        }
        else{
            // normal case
            set_max(rk, pop_back());
            validate_node(rk); // validate interval
            access(rk);
        }
    }
    void insert_element(int x){
        if(++cnt % 2)a.push_back({0, 0}); // expand
        int rk = (cnt + 1) / 2 - 1; // get real index
        if(cnt % 2){
            a[rk][0] = x;
        }
        else{
            a[rk][1] = max(a[rk][0], x);
            a[rk][0] = min(a[rk][0], x);
        }
        access(rk + 1);
    }
    int get_min(){
        return get_min(1);
    }
    int get_max(){
        return get_max(1);
    }
    void validate(){
        // only for debug
        for(int i=1; exist(i); i++){
            if(exist(i << 1)){
                if(get_min(i) > get_min(i << 1) || get_max(i) < get_max(i << 1))
                    cout << "error" << endl;
            }
            if(exist(i << 1 | 1)){
                if(get_max(i) < get_max(i << 1 | 1) || get_min(i) > get_min(i << 1 |
1))
                    cout << "error" << endl;
```

```
            }
        }
    }
};

int main(){
    int n;
    IntervalHeap heap;
    cin >> n;
    while(n--){
        int op, x;
        cin >> op;
        if(op == 1){
            cin >> x;
            heap.insert_element(x);
        }
        if(op == 2){
            cout << heap.get_min() << endl;
        }
        if(op == 3){
            heap.remove_min();
        }
        if(op == 4){
            cout << heap.get_max() << endl;
        }
        if(op == 5){
            heap.remove_max();
        }
    }
    return 0;
}
```

细节

- `remove_min` 和 `remove_max` 的后处理稍有不同。但后处理部分，都需要分最后一个节点是否为二叉树最后一个节点来讨论。
- `pop_back` 需要注意 `get_min` 和 `get_max` 时
- `get_max` 需要注意在最后一个节点只有一个值时需要返回 `a[rk][1]`。
- `swap_node` 需要考虑特殊处理不满的节点。

测试

一共五个功能，分别是插入、删除最大最小、查找最大最小，操作编号为 $1-5$，使用 `multiset` 作为检验器，编写对拍程序进行正确性检验。

测试方式写于 validator 的注释部分。

在 Windows11 操作系统下进行如下测试，并解决了初始版本的一些问题，得到了最终版本代码。

数据生成器

```
#include <bits/stdc++.h>

using namespace std;

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());


int rand(int l, int r){
    return rng() % (r - l + 1) + l;
}

int sz = 0;
multiset<int> s;

void remove_max(){
    s.erase(s.find(*s.rbegin()));
    sz--;
}

void remove_min(){
    s.erase(s.begin());
    sz--;
}

int get_max(){
    return *s.rbegin();
}

int get_min(){
    return *s.begin();
}

void insert_element(int x){
    s.insert(x);
    sz++;
}

int main(){
    ofstream fout("data.in");
    ofstream fans("data.ans");
    int k = 1000;
    int level = rand(1, 10);
    fout << k << endl;
    while(k--){
        int op = rand(1, 5), val = rand(1, 1 << level);
        if(op == 1 || sz == 0){
            fout << 1 << " " << val << endl;
            insert_element(val);
            continue;
```

```
        }
        if(op == 2){
            fout << op << endl;
            fans << get_min() << endl;
        }
        if(op == 3){
            fout << op << endl;
            remove_min();
        }
        if(op == 4){
            fout << op << endl;
            fans << get_max() << endl;
        }
        if(op == 5){
            fout << op << endl;
            remove_max();
        }
    }
}
```

检验器

```
/*
使用:

作业名为 <homework>, 工作目录下有 <homework>/<homework>.exe, <homework>/gen.exe

<homework>/<homework>.exe: 标准输入输出读写
<homework>/gen.exe: 输出测试数据到 data.in, 标准答案到 data.ans

每次运行时需要重新编译 <homework>/<homework>.cpp 和 <homework>/gen.cpp 生成对应可执行文件.

运行 validator.exe <homework> 生成测试数据并检查答案.
*/
#include <bits/stdc++.h>

using namespace std;


int main(int argc, char *argv[])
{
    string generator = string(argv[1]) + "\\gen"s;
    string myprog = string(argv[1]) + "\\"s + string(argv[1]);
    string exec = myprog + " < data.in > data.out";
    cout << "checking:" << exec << '\n';
    for(int test=1; test<=100; test++){
        cout << "test " << test << '\n';
        system(generator.c_str());
        system(exec.c_str());
        if(system("fc data.out data.ans")){
```

```
            cout << "wrong answer\n";
            return 0;
        }
        cout << "ok\n";
    }
    return 0;
}
```

讨论

关于查找第 k 大元素的效率。

由于 IntervalHeap 的实现，查找第 k 大/小元素的时间复杂度均为 $O(k \log n)$。理论上，当 $k > \frac{size}{2}$ 时，相比于普通堆，IntervalHeap 的查找效率更高，只需要 $n - k$ 次操作，而 $n - k < k$，因此理论效率更高。

实际上 IntervalHeap 的常数因子较大，所以对于 k 的分布比较平均的情况，查找效率不一定比普通堆高。但是如果 k 集中在 $[1, \epsilon] \bigcup [n - \epsilon, n]$ 上，那么 IntervalHeap 的效率会更高。

关于本数据结构

本数据结构在实际中有更加优秀的替代，无论在实现难度，功能性，效率上，IntervalHeap 都比不上朴素的平衡树，所以不具备练习之外的任何意义。

真的认真看到这里了嘛……

如果看到这里了给我 Repo 点个 star 好嘛，谢谢！