中山大学计算机学院本科生实验报告

(2025 学年第 1 学期)

课程名称：数据结构与算法　　实验任课教师：张子臻

年级	2024 级	专业（方向）	计算机科学与技术（人工智能与大数据）
学号	242325157	姓名	梁玮麟
电话	18620062182	Email	3371676041@qq.com
开始日期	2025.10.22	结束日期	2025.10.25

第一题

1、实验题目

z7-检查一个序列是否构成堆

问题描述　给定一个整数序列，请检查该序列是否构成堆。

问题补充

- 输入: 第一行是一个整数 $n(0 < n < 10000)$，第二行是 n 个整数。
- 输出: 如果序列是极小堆，即堆顶是最小元，则输出" min heap"；如果是极大堆，即堆顶是最大元，则输出 "max heap"；否则不是堆，则输出 "no"。如果既是极大堆，又是极小堆，则输出 "both"。
- 输出包括一行，最后有换行。

输入输出格式样例

```
## input
5
3 4 2 1 1
```

样例输出

```
no
```

2、实验目的

掌握**堆的结构性质**和判定原理，能够根据父子节点关系判断一个序列是否为最大堆、最小堆或非堆。

3、算法设计

设计思路：

- 我首先明确了堆的定义：最大堆要求**父节点值不小于子节点值**，最小堆则**相反**。为了同时判断两种情况，我使用两个标志 `isMax` 和 `isMin` 来记录。
- 算法的核心在于从根节点开始遍历所有非叶子节点，检查它与左右子节点的大小关系。若出现违反条件的情况，就将相应标志置为 `false`。

流程图：

```
Start
  ↓
读入数组 → 初始化 isMax, isMin
  ↓
遍历所有非叶节点
  - 若左子存在 → 比较并更新标志
```

- 若右子存在 → 比较并更新标志
- 若两标志皆为 `false` → 结束

↓

输出结果(both / max / min / no)

↓

End

复杂度分析：

时间复杂度：O(n)
空间复杂度：O(1)

细节注意:

- 循环只到 n/2-1，因为**后半部分是叶子节点。**
- 判断孩子存在时要避免越界访问。
- 若所有父子节点都相等，则同时满足最大堆和最小堆。

具体实现：

```
//z7-检查一个序列是否构成堆
#include<iostream>
#include<vector>
using namespace std;
int main(){
    int n;
    cin>>n;
    vector<int> num(n);
    for (int i=0;i<n;i++){
        cin>>num[i];
    }
    int max=1,min=1;
    for(int i=0;i<=n/2-1;i++){
        int left=2*i+1;
        int right = 2*i+2;
        if(left<n&&num[left]>num[i]) max=0;
        if(right<n&&num[right]>num[i]) max=0;
        if(left<n&&num[left]<num[i]) min=0;
        if(right<n&&num[right]<num[i]) min=0;
    }
    if(max&&min) cout<<"both"<<endl;
    else if(max) cout<<"max heap"<<endl;
    else if(min) cout<<"min heap"<<endl;
    else cout<<"no"<<endl;
    return 0;
}
```

4、程序运行与测试

运行结果： (仅挑选个别复杂测试样例作为展示)

测试样例一

- 标准输入：

```
1931
0 0 0 0 0 1 0 1 0 4 0 1 1 13 0 2 1 0 4 5 4 8 0 2 1 1 8 13 23 0 0 2 7 1 5 7 0 4 13 5 15 8 10 10 8 0 4 2
```

- 实际输出：

```
min heap
```

- 期望输出：

```
min heap
```

测试样例二

- 标准输入：

```
6588
499 499 499 499 499 499 499 499 499 499 497 499 499 498 499 499 498 499 499 499 498 497 497 497 499 497
```

- 实际输出：

```
max heap
```

- 期望输出：

```
max heap
```

5、实验总结与心得

- 通过这题，我理解了堆的本质是完全二叉树结构，并能在一次遍历中用逻辑判断出四种结果。相比直接分类讨论，这种方式更简洁高效。

第二题

1、实验题目

z7-奖学金

题目描述 某小学最近得到了一笔赞助，打算拿出其中一部分为学习成绩优秀的前 5 名学生发奖学金。期末，每个学生都有 3 门课的成绩: 语文、数学、英语。先按总分从高到低排序，如果两个同学总分相同，再按语文成绩从高到低排序，如果两个同学总分和语文成绩都相同，那么规定学号小的同学排在前面，这样，每个学生的排序是唯一确定的。任务: 先根据输入的 3 门课的成绩计算总分，然后按上述规则排序，最后按排名顺序输出前 5 名学生的学号和总分。注意，在前 5 名同学中，每个人的奖学金都不相同，因此，你必须严格按上述规则排序。例如，在某个正确答案中，如果前两行的输出数据 (每行输出两个数: 学号、总分) 是: 7 279 5 279 这两行数据的含义是: 总分最高的两个同学的学号依次是 7 号、5 号。这两名同学的总分都是 279(总分等于输入的语文、数学、英语三科成绩之和)，但学号为 7 的学生语文成绩更高一些。如果你的前两名的输出数据是: 5 279 7 279 则按输出错误处理，不能得分。

输入描述 输入包含多组测试数据，每个测试数据有 n+1 行。第 1 行为一个正整数 n，表示该校参加评选的学生人数。第 2 到 n+1 行，每行有 3 个用空格隔开的数字，每个数字都在 0 到 100 之间。第 j 行的 3 个数字依次表示学号为 j-1 的学生的语文、数学、英语的成绩。每个学生的学号按照输入顺序编号为 1~n(恰好是输入数据的行号减 1)。所给的数据都是正确的，不必检验。

输出描述 对于每个测试数据输出 5 行，每行是两个用空格隔开的正整数, 依次表示前 5 名学生的学号和总分。两个相邻测试数据间用一个空行隔开。

样例输入

```
6
90 67 80
87 66 91
78 89 91
88 99 77
```

```
67 89 64
78 89 98
8
80 89 89
88 98 78
90 67 80
87 66 91
78 89 91
88 99 77
67 89 64
78 89 98
```

样例输出

```
6 265
4 264
3 258
2 244
1 237
8 265
2 264
6 264
1 258
5 258
```

2、实验目的

掌握**利用堆结构实现 Top-K 选拔**的方法，理解多种实现方案在时间和空间复杂度上的差异，并能灵活选择合适算法。

3、算法设计

设计思路：

- 我对比了四种方法：**直接排序、矩阵 + 堆、手搓堆、优先队列。**在实验过程中，我先实现了手搓堆，因为当时还不熟悉 `priority_queue`；之后学习 STL 后又实现了优先队列版本，用来验证效率与正确性。

方法一：直接排序

- 将所有学生信息读入数组后，计算总分并根据题意排序，最后输出前 5 名。
- 优点是简单，缺点是当学生数量较多时，排序浪费性能。

方法二：矩阵存储 + 小根堆

- 将每个学生信息按 `id, chinese, math, english, sum` 存在**二维数组**中，用小根堆维护前 5 名的 `id`。
- 若堆未满直接插入，若堆已满则比较当前学生与堆顶，若更优则替换。
- 时间复杂度近似线性，为 `O(n log 5)`。

方法三：手搓堆

- 自己实现 `siftup` 与 `siftdown`。每插入一名学生都执行**上浮**操作，当堆大小超过 5 时弹出堆顶。
- 下沉操作中，我采用“三节点择劣”的设计思路：先判断**左右子中哪个更差，再决定是否交换，这样可以避免重复比较。**

```
// 手搓堆维护前 5 名
void insertStudent(Student s) {
    heap.push_back(s);
```

```
    siftUp(heap.size() - 1);
    if (heap.size() == 6) {
        popTop();
    }
}
```

方法四：优先队列

- 学习 priority_queue 后，使用内置容器替代手写堆。
- 每插入一个元素后判断堆是否超过 5 个，如果超过就 pop 一次。该方法**代码简洁且稳定。**

```
// 使用优先队列维护前 5 名
priority_queue<Student, vector<Student>, worseFirst> pq;
for (auto s : students) {
    pq.push(s);
    if (pq.size() > 5) pq.pop();
}
```

复杂度比较：

方法	时间复杂度	空间复杂度	特点
直接排序	O(n log n)	O(n)	实现简单
矩阵 + 堆	O(n log 5)	O(n)	保留全表信息
手搓堆	O(n log 5)	O(5)	控制力强
优先队列	O(n log 5)	O(5)	代码简洁

流程图：

```
Start
  ↓
读入学生信息
  ↓
维护小根堆(<=5)
  - 若堆未满 → 插入
  - 若堆已满且更优 → 弹出堆顶插入
  |- 否则跳过
  ↓
弹出堆中前5个元素
  ↓
输出结果
  ↓
End
```

细节注意：

- 注意边界情况，索引计算！
- 优先队列中，**a 和 b 作比较，如果是 Ture，说明 a 要在 b 后面**！

具体实现：

```
//z7-奖学金
//手搓 heap 版本
#include <iostream>
#include <vector>
using namespace std;
```

```
class student {
  private:
    int id = 0;
    int sum = 0;
    int chinese = 0;
    int math = 0;
    int english = 0;

  public:
    student() {}
    student(int id, int chinese, int math, int english)
        : id(id), chinese(chinese), math(math), english(english) {
        sum = chinese + math + english;
    }
    student (const student& other)=default;
    student &operator=(const student &other) =default;
    bool operator>(const student &other) { //> 表示更优
        if (this->sum != other.sum)
            return this->sum > other.sum;
        if (this->chinese != other.chinese)
            return this->chinese > other.chinese;
        return this->id < other.id;
    }
    friend ostream &operator<<(ostream &os, const student a) {
        os << a.id << " " << a.sum;
        return os;
    }
};
void studentswap(student &a, student &b) {
    student temp = a;
    a = b;
    b = temp;
}
// 0 1 2 3 4 5     6
void siftup(vector<student> &heap) { // size=idx+1
    int size=heap.size();
    int idx = size - 1;
    while (idx > 0) {
        int father = (idx - 1) / 2; // father(idx)
        if (heap[father] > heap[idx]) {
            studentswap(heap[father], heap[idx]);
            idx = father;
        }else break;
    }
}
void siftdown(vector<student> &heap) {
    int size=heap.size();
    int idx=0;
    while(idx<=size/2-1){
        int left=idx*2+1;
        int right=left+1;
        int worse = right<size&&heap[left]>heap[right]? right:left;
        if(heap[idx]>heap[worse]){
            studentswap(heap[idx],heap[worse]);
```

```
            idx=worse;
        }
        else break;
    }
}
student poptop(vector<student> &heap) {
    int size=heap.size();
    student temp = heap[0];
    heap[0] = heap[size - 1];
    heap.pop_back();
    siftdown(heap);
    return temp;
}
void insert(vector<student> &heap, const student &a) {
    // 维护小根堆
    heap.push_back(a);
    siftup(heap);
    if (heap.size() == 6) {
        // pop
        poptop(heap);
    }
}

int main() {
    int n;
    while (cin >> n) {
        vector<student> heap;
        for (int i = 0; i < n; i++) {
            int a, b, c;
            cin >> a >> b >> c;
            insert(heap, student(i + 1, a, b, c));
        }
        vector<student> result(5);
        for (int i = 4; i > -1; i--) {
            result[i] = poptop(heap);
        }
        for (int i = 0; i < 5; i++) {
            cout << result[i] << endl;
        }
        cout<<endl;
    }
}
//z7-奖学金
//优先队列版本
#include <iostream>
#include <queue>
#include<vector>
using namespace std;
class student {
  private:
    int id = 0;
    int sum = 0;
    int chinese = 0;
```

```
    int math = 0;
    int english = 0;

  public:
    student() {}
    student(int id, int chinese, int math, int english)
        : id(id), chinese(chinese), math(math), english(english) {
        sum = chinese + math + english;
    }
    bool operator>(const student &other) const{ //> 表示更优
        if (this->sum != other.sum)
            return this->sum > other.sum;
        if (this->chinese != other.chinese)
            return this->chinese > other.chinese;
        return this->id < other.id;
    }
    friend ostream &operator<<(ostream &os, const student a) {
        os << a.id << " " << a.sum;
        return os;
    }
};
class worsefirst{
    public:
    bool operator()(const student& a, const student& b)const{
        return a>b;
    }
};
int main() {
    int n;
    while (cin >> n) {
        priority_queue<student,vector<student>,worsefirst> heap;
        for (int i = 0; i < n; i++) {
            int a, b, c;
            cin >> a >> b >> c;
            heap.push(student(i+1,a,b,c));
            if(heap.size()>5) heap.pop();
        }
        vector<student> result(5);
        for (int i = 4; i > -1; i--) {
            result[i] = heap.top();
            heap.pop();
        }
        for (int i = 0; i < 5; i++) {
            cout << result[i] << endl;
        }
        cout<<endl;
    }
}
```

4、程序运行与测试

运行结果： (仅挑选个别复杂测试样例作为展示)

测试样例一

- 标准输入：

```
30
78 51 71
41 59 72
34 62 31
56 74 37
81 96 92
58 37 80
94 58 59
98 84 84
80 44 34
44 35 97
95 96 94
76 41 86
88 42 57
84 89 84
75 43 98
31 48 61
95 96 93
72 46 47
61 85 49
81 69 42
71 56 88
97 77 93
87 92 92
38 85 55
56 53 46
52 89 33
48 38 82
69 60 31
93 96 92
85 36 99
```

- 实际输出：

```
11 285
17 284
29 281
23 271
5 269
```

- 期望输出：

```
11 285
17 284
29 281
23 271
5 269
```

测试样例二

- 标准输入：

```
10
78 44 40
91 91 83
98 51 54
61 88 33
77 46 83
76 93 67
66 68 61
76 35 69
68 79 36
90 90 84
```

- 实际输出：

```
2 265
10 264
6 236
5 206
3 203
```

- 期望输出：

```
2 265
10 264
6 236
5 206
3 203
```

5、实验总结与心得

- 这题让我理解了“**优先队列的本质是堆**”。手写堆让我熟悉了堆操作的实现，而 STL 优先队列展示了标准化写法。Top-K 问题在实践中很常见，堆结构是最有效的解决方案之一。

第三题

1、实验题目

z7-heap

题目描述 实现一个小根堆用以做优先队列。给定如下数据类型的定义：

```
class array {
 private:
 int elem[MAXN];
 public:
 int &operator[](int i) { return elem[i]; }
};
class heap {
 private:
 int n;
 array h;
 public:
 void clear() { n = 0; }
 int top() { return h[1]; }
 int size() { return n; }
 void push(int);
```

```
 void pop();
};
```

要求实现：

```
void heap::push(int x) {
 // your code
}
void heap::pop() {
 // your code
}
```

测试样例 test code:

```
heap h;
h.push(3);
h.push(1);
h.push(2);
printf("%d\n", h.top());
h.pop();
printf("%d\n", h.top());
```

test output:

```
1
2
```

提示 只提交 heap::push 和 heap::pop，注意堆顶元素的下标是 1 而不是 0。

2、实验目的

掌握**堆的插入与删除**原理，理解上浮、下沉操作的过程与逻辑，并能在代码中准确实现。

3、算法设计

设计思路：

- 采用 **1 下标**存储方式（h[1] 为堆顶）。push 操作通过**上浮**维持堆序，pop 操作通过**尾替顶 + 下沉**恢复堆序。

流程图：

Push：
插入元素到尾部 → 上浮维持堆序 → 结束

Pop：
尾替顶 → 下沉维持堆序 → 结束

复杂度分析： push 与 pop 均为 O(log n)，空间为 O(n)。

细节注意：

- 索引从 **1** 开始，而不是 0！计算得到最**后一个非叶子节点**的时候很容易出错！

具体实现：

```
//z7-heap
#include <iostream>
#include"heap.h"
```

```
using namespace std;
void heap::push(int a){
    n++;
    h[n]=a;//放入最后的节点
    int i=n;
    while(i>1){
        int father=i/2;
        if(h[father]>h[i]){
            int temp=h[father];
            h[father]=h[i];
            h[i]=temp;
        }
        i=father;

    }
}
void heap::pop(){
    if(n>=1){
        h[1]=h[n];
        h[n--]=0;
        int i=1;
        while(true){
            int left=2*i;
            int right=2*i+1;
            if(left<=n){
                if(right<=n){//有两个节点
                    if(h[i]>h[left]&&h[i]>h[right]){
                        if(h[left]>h[right]){
                            int temp=h[right];
                            h[right]=h[i];
                            h[i]=temp;
                            i=right;
                        }
                        else{
                            int temp=h[left];
                            h[left]=h[i];
                            h[i]=temp;
                            i=left;
                        }
                    }
                    else if(h[i]>h[left]){
                            int temp=h[left];
                            h[left]=h[i];
                            h[i]=temp;
                            i=left;
                    }
                    else if(h[i]>h[right]){
                            int temp=h[right];
                            h[right]=h[i];
                            h[i]=temp;
                            i=right;
                    }
                    else break;
                }
```

```
                else{
                    if(h[left]<h[i]){
                        int temp=h[i];
                        h[i]=h[left];
                        h[left]=temp;
                    }
                    break;
                }
            }
            else break;
        }
    }
}
```

4、程序运行与测试

运行结果： (仅挑选个别复杂测试样例作为展示)

测试样例一

- 标准输入：

```
51007
861014941 893482403 220308889 -506052296 -658696671 -891058487 -808104526 -778424549 416391927 900682410
```

- 实际输出：

```
-999988834 -999977234 -999950505 -999929867 -999903630 -999899449 -999867776 -999856528 -999823636 -9998
```

- 期望输出：

```
-999988834 -999977234 -999950505 -999929867 -999903630 -999899449 -999867776 -999856528 -999823636 -9998
```

测试样例二

- 标准输入：

```
85397
-655301987 119301786 -680264625 768873105 910883774 -327780111 512210548 802042028 694643116 -768406799
```

- 实际输出：

```
-999980821 -999950650 -999947690 -999928697 -999902420 -999875017 -999830372 -999794899 -999757712 -9997
```

- 期望输出：

```
-999980821 -999950650 -999947690 -999928697 -999902420 -999875017 -999830372 -999794899 -999757712 -9997
```

5、实验总结与心得

- 本题让我掌握了堆的基本操作实现方式。对比 STL 优先队列，我更能理解它的底层原理。通过这题，我体会到复用堆结构在后续 Top-K 问题中的优势。