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　　任课老师：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:	24325155	姓名:	梁桂铭
电话:	15817681625	Email:	lianggm8@mail2.sysu.edu.cn
开始日期:	2025年10月24日	完成日期:	2025年10月24日

第一题

1. 实验题目

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
}
```

---

2. 实验目的

1. 掌握**堆**的定义与性质。
2. 理解并实现小根堆的**插入**与**删除堆顶**操作。
3. 理解堆的数组存储结构与**上滤、下滤**（sift-up, sift-down）算法。
4. 提高对**优先队列**及其在算法设计中应用的理解。

---

3. 算法设计

（1）堆的结构

堆使用数组存储，其中下标从1开始：

- 对于结点i：
  - 左子节点下标为$2i$
  - 右子节点下标为$2i+1$
  - 父节点下标为$\lfloor i/2 \rfloor$

（2）插入操作 `push(int x)`

1. 将新元素x插入堆尾（即 `h[++n] = x`）。

2. 进行**上滤**（sift-up）操作：若父节点值大于当前节点值，则交换两者。
3. 不断重复，直到堆序性质满足或到达根节点。

（3）删除操作 pop()

1. 将堆顶元素（最小值）删除，并用堆尾元素覆盖 `h[1] = h[n--]`。
2. 从堆顶开始执行**下滤**（sift-down）：
    - 比较当前节点与左右子节点，选择其中较小者进行交换。
    - 直至堆序性恢复。

在执行上滤操作时，只和当前分支有关，不会对其他分支的堆序性产生影响，而下滤操作需要同时考虑两个分支

（4）C++代码实现

```
#include "heap.h"
#include <iostream>
using namespace std;

void heap::push(int x) {
    h[++n] = x;
    int i = n;
    while (i > 1 && h[i/2] > h[i]) {
        int tmp = h[i];
        h[i] = h[i/2];
        h[i/2] = tmp;
        i = i / 2;
    }
}

void heap::pop() {
    if (n == 0) return;
    h[1] = h[n--];
    int i = 1;
    while (1) {
        int l = i * 2;
        int r = i * 2 + 1;
        int min = i;
        if (l <= n && h[l] < h[min]) min = l;
        if (r <= n && h[r] < h[min]) min = r;
        if (min != i) {
            int tmp = h[min];
```

```
            h[min] = h[i];
            h[i] = tmp;
            i = min;
        } else break;
    }
}
```

5）复杂度分析

- **插入操作**：每次上滤至多经过$O(\log n)$层。
- **删除操作**：下滤同样至多经过$O(\log n)$层。
- 因此整体时间复杂度为$O(\log n)$，空间复杂度为$O(1)$。

---

4. 程序运行与测试

测试代码

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

运行结果

```
1
2
```

---

5. 实验总结与心得

通过本实验，我深入理解了堆这种**完全二叉树**结构的逻辑实现，掌握了其数组下标映射规则与**上滤、下滤算法**的核心思想。
在实际编码中，最关键的部分是：

- 保证堆序性（父节点始终小于子节点）；

- 注意下标从1开始；
- 在 pop() 时同时更新堆大小 n 。

在编写代码时我注意到，无论是上滤还是下滤操作都需要着重考虑边界处理条件，否则在遍历到最后会出错。

第二题

1. 实验题目

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。

---

2. 实验目的

1. 理解**堆**（Heap）结构的定义与性质。
2. 掌握**极小堆**与**极大堆**的判断条件：
    - 极小堆（min heap）：任意节点值 ≤ 其左右子节点值。
    - 极大堆（max heap）：任意节点值 ≥ 其左右子节点值。
3. 熟悉堆的**顺序存储**结构（数组）及其父子节点索引关系。
4. 能够编写程序判断任意整数序列是否为堆。

---

3. 算法设计

（1）堆结构性质

堆通常用数组表示。若下标从0开始，则有：

- 左子节点下标：$2i+1$
- 右子节点下标：$2i+2$
- 父节点下标：$\lfloor (i-1)/2 \rfloor$

（2）判断极大堆

对所有有子节点的节点i，若存在子节点j满足：

$$nums[i] < nums[j]$$

则该序列不是极大堆。

（3）判断极小堆

同理，对所有有子节点的节点i，若存在子节点j满足：

$$nums[i] > nums[j]$$

则该序列不是极小堆。

（4）实现思路

1. 读取输入的n个整数。
2. 分别设置两个标志 `flag1` 和 `flag2` 表示是否为极大堆或极小堆。
3. 分两次遍历：
    - 第一次判断是否满足极大堆条件；
    - 第二次判断是否满足极小堆条件。
4. 根据两个标志输出：
    - 仅 `flag1==1` → `"max heap"`
    - 仅 `flag2==1` → `"min heap"`
    - 二者都为真 → `"both"`
    - 否则 → `"no"`

（5）复杂度分析

- 时间复杂度：遍历数组每个下标并检查其最多两个子节点，操作为常数次。因此总体时间复杂度为 $O(n)$。

- 空间复杂度：只使用常数个额外标志变量，除输入存储外额外空间为 $O(1)$。

---

4. 程序运行与测试

C++代码实现

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

int N;
vector<int> nums;

int main () {
    int tmp;
    cin >> N;
    for (int i = 0; i < N; i++) {
        cin >> tmp;
        nums.push_back(tmp);
    }

    int flag1 = 1, flag2 = 1;

    // 判断是否为极大堆
    for (int i = 0; i < N; i++) {
        if (i * 2 + 1 < N && nums[i] < nums[i * 2 + 1]) flag1 = 0;
        if (i * 2 + 2 < N && nums[i] < nums[i * 2 + 2]) flag1 = 0;
    }

    // 判断是否为极小堆
    for (int i = 0; i < N; i++) {
        if (i * 2 + 1 < N && nums[i] > nums[i * 2 + 1]) flag2 = 0;
        if (i * 2 + 2 < N && nums[i] > nums[i * 2 + 2]) flag2 = 0;
    }

    if (flag1 && !flag2) cout << "max heap";
    else if (!flag1 && flag2) cout << "min heap";
    else if (flag1 && flag2) cout << "both";
    else cout << "no";
    cout << endl;
    return 0;
}
```

测试样例

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


no
```

---

5. 实验总结与心得

本实验让我深入理解了堆的两种形式及其存储特征。
通过数组索引关系的计算，可以在线性时间内判断一个序列是否为堆结构。实验要点包括：

1. 堆的性质是**局部有序**而非全局有序；
2. 只需检查每个父节点与其子节点的大小关系；
3. 边界情况（叶节点、单元素序列）需要特别注意；
4. 算法时间复杂度为 $O(n)$，额外空间为 $O(1)$，适合大规模输入的在线判断。

第三题

---

1. 实验题目

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。

---

2. 实验目的

1. 掌握结构体（ `struct` ）的定义与比较函数的设计方法；
2. 理解**优先队列）**在自动排序场景中的应用；
3. 能够根据多级排序条件实现**稳定的综合排序逻辑**；

---

3. 算法设计

（1）题目要求回顾

对每位学生的三门成绩（语文、数学、英语）计算总分，然后依据以下规则进行排序：

1. **总分降序**；
2. 若总分相同，**语文成绩降序**；
3. 若总分与语文成绩均相同，**学号升序**。

排序完成后，输出前5名学生的学号及总分。

（2）数据结构定义

使用结构体 score 存储每个学生的信息：

```
struct score {
    int num; // 学号
    int sum; // 总分
    int c;   // 语文成绩
    int m;   // 数学成绩
    int e;   // 英语成绩
};
```

（3）比较规则实现

通过**重载小于号运算符** operator< 实现自定义排序逻辑。

```
bool operator<(const score& other) const {
    if (sum < other.sum) return true;                // 总分较低者优先级低
    else if (sum == other.sum) {
        if (c < other.c) return true;                // 语文分低者优先级低
        else if (c == other.c) return num > other.num; // 学号大者优先级低
    }
    return false;
}
```

注意这里的逻辑是**反向定义**，因为 priority_queue 默认是大顶堆：

- 若 a < b 为真，则 b 的优先级更高；
- 因此，为了让“高分在前”，我们需要在比较时返回与排序直觉相反的布尔值。

(4)复杂度分析

- **时间复杂度：**
  - 每次插入 priority_queue 的时间为$O(\log n)$；
  - 共插入n个学生，时间复杂度为$O(n \log n)$；
  - 输出前5个元素只需常数时间；
  - **总体复杂度：**$O(n \log n)$。

- **空间复杂度：**
  - 所有学生数据及堆结构存储需$O(n)$空间；
  - 额外变量开销为常数；
  - **空间复杂度：$O(n)$。**

4.程序运行与测试

代码实现

```
#include <iostream>
#include <queue>
using namespace std;

struct score {
    int num;
    int sum;
    int c;
    int m;
    int e;
    bool operator<(const score& other) const {
        if (sum < other.sum) return true;
        else if (sum == other.sum) {
            if (c < other.c) return true;
            else if (c == other.c) return num > other.num;
        }
        return false;
    }
};

priority_queue<score> scores;

int main () {
    int N, c, m, e, s;
    while (cin >> N) {
        for (int i = 1; i <= N; i++) {
            cin >> c >> m >> e;
            s = c + m + e;
            scores.push(score{i, s, c, m, e});
        }

        int count = 0;
        while (!scores.empty() && count < 5) {
```

```
            score tmp = scores.top();
            cout << tmp.num << ' ' << tmp.sum << endl;
            scores.pop();
            count++;
        }

        // 清空队列并输出空行以分隔测试数据
        while (!scores.empty()) scores.pop();
        cout << endl;
    }
    return 0;
}
```

测试样例

```
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
```

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

8 265
2 264
6 264
1 258
```

```
5 258
```

---

6. 实验总结与心得

通过本实验，我掌握了如何使用**优先队列**(priority_queue)处理复杂的排序任务。
相比于手动排序（如 `sort`），`priority_queue` 在需要动态选取前若干元素的场景中更高效。

本实验中难点在于：

1. 理解 `priority_queue` 默认比较方向与逻辑的反转；
2. 正确实现多关键字排序规则；
3. 注意多组输入之间输出格式的空行要求。

附录，提交文件清单

第一题

```cpp
#include "heap.h"
#include <iostream>
using namespace std;

void heap::push(int x) {
    h[++n] = x;
    int i = n;
    while (i > 1 && h[i/2] > h[i]) {
        int tmp = h[i];
        h[i] = h[i/2];
        h[i/2] = tmp;
        i = i / 2;
    }
}

void heap::pop() {
    if (n == 0) return;
    h[1] = h[n--];
    int i = 1;
    while (1) {
```

```
            int l = i * 2;
            int r = i * 2 + 1;
            int min = i;
            if (l <= n && h[l] < h[min]) min = l;
            if (r <= n && h[r] < h[min]) min = r;
            if (min != i) {
                int tmp = h[min];
                h[min] = h[i];
                h[i] = tmp;
                i = min;
            } else break;
        }
    }
```

第二题

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

int N;
vector<int> nums;

int main () {
    int tmp;
    cin >> N;
    for (int i = 0; i < N; i++) {
        cin >> tmp;
        nums.push_back(tmp);
    }

    int flag1 = 1, flag2 = 1;

    // 判断是否为极大堆
    for (int i = 0; i < N; i++) {
        if (i * 2 + 1 < N && nums[i] < nums[i * 2 + 1]) flag1 = 0;
        if (i * 2 + 2 < N && nums[i] < nums[i * 2 + 2]) flag1 = 0;
    }

    // 判断是否为极小堆
    for (int i = 0; i < N; i++) {
        if (i * 2 + 1 < N && nums[i] > nums[i * 2 + 1]) flag2 = 0;
        if (i * 2 + 2 < N && nums[i] > nums[i * 2 + 2]) flag2 = 0;
    }
```

```
    if (flag1 && !flag2) cout << "max heap";
    else if (!flag1 && flag2) cout << "min heap";
    else if (flag1 && flag2) cout << "both";
    else cout << "no";
    cout << endl;
    return 0;
}
```

第三題

```
#include <iostream>
#include <queue>
using namespace std;

struct score {
    int num;
    int sum;
    int c;
    int m;
    int e;
    bool operator<(const score& other) const {
        if (sum < other.sum) return true;
        else if (sum == other.sum) {
            if (c < other.c) return true;
            else if (c == other.c) return num > other.num;
        }
        return false;
    }
};

priority_queue<score> scores;

int main () {
    int N, c, m, e, s;
    while (cin >> N) {
        for (int i = 1; i <= N; i++) {
            cin >> c >> m >> e;
            s = c + m + e;
            scores.push(score{i, s, c, m, e});
        }

        int count = 0;
        while (!scores.empty() && count < 5) {
```

```
                score tmp = scores.top();
                cout << tmp.num << ' ' << tmp.sum << endl;
                scores.pop();
                count++;
            }

            // 清空队列并输出空行以分隔测试数据
            while (!scores.empty()) scores.pop();
            cout << endl;
        }
        return 0;
    }
```