同济大学

TONGJI UNIVERSITY

数据结构课程设计

项目名称	银行业务
学　　院	计算机科学与技术学院
专　　业	软件工程
学生姓名	杨瑞晨
学　　号	2351050
指导教师	张颖
日　　期	2024 年 12 月 4 日

目 录

1 项目分析

1.1 项目背景分析

队列管理系统是一个模拟现实世界中排队处理业务的软件系统。在许多服务场景中，如银行、医院等，顾客需要按照到达的顺序排队等待服务。本系统旨在通过程序模拟这一过程，同时考虑到不同队列的处理速度和优先级，以提高服务效率。

1.2 项目功能分析

1.2.1 功能要求

设某银行有 A，B 两个业务窗口，且处理业务的速度不一样，其中 A 窗口处理速度是 B 窗口的 2 倍——即当 A 窗口每处理完 2 个顾客是，B 窗口处理完 1 个顾客。给定到达银行的顾客序列，请按照业务完成的顺序输出顾客序列。假定不考虑顾客信后到达的时间间隔，并且当不同窗口同时处理完 2 个顾客时，A 窗口的顾客优先输出。

1.2.2 输入要求

输入为一行正整数，其中第一数字 N（N ≤1000）为顾客总数，后面跟着 N 位顾客的编号。编号为奇数的顾客需要到 A 窗口办理业务，为偶数的顾客则去 B 窗口。数字间以空格分隔。

1.2.3 输出要求

按照业务处理完成的顺序输出顾客的编号。数字键以空格分隔，但是最后一个编号不能有多余的空格。

1.2.4 项目实例

序号	输入	输出	说明
1	8 2 1 3 9 4 11 13 15	1 3 2 9 11 4 13 15	正常测试，A 窗口人多
2	8 2 1 3 9 4 11 12 16	1 3 2 9 11 4 12 16	正常测试，B 窗口人多
3	1 6	6	最小 N

2 项目设计

2.1 数据结构设计

队列管理系统使用两个队列（Queue）来存储顾客信息。每个队列都是一个动态数组，支持动态扩容，使用模板支持任意数据类型。队列是一种先进先出（FIFO, First-In-First-Out）的数据结构，它允许在一端（队尾）插入元素，而在另一端（队头）移除元素，支持入队、出队、查看队头元素等操作。

队列具有以下一系列优点：

（1）有序性：队列中的元素按照它们被添加的顺序排列，先进入的元素先被移除。

（2）动态性：大多数队列实现都是动态的，可以根据需要自动调整大小。

（3）限制性访问：队列只允许在两端进行操作，队头用于移除元素，队尾用于添加元素。

2.2 类设计

Queue 类 Queue 类提供一系列操作：

- 构造函数：创建一个队列对象，可以指定队列的初始容量。
- 析构函数：销毁队列对象，释放内存。
- enQueue：入队操作，将一个元素添加到队列的末尾。
- deQueue：出队操作，移除队列开头的元素。
- front：获取队头元素，返回队列开头元素的引用。
- isEmpty：判断队列是否为空。
- isFull：判断队列是否已满。
- size：获取队列大小。
- clear：清空队列。

```
const int DEFAULT_CAPACITY = 10;

template <typename T>
class Queue
{
private:
    T *data;        // 队列元素数组
    int frontIndex; // 队头索引
    int rearIndex;  // 队尾索引
    int capacity;   // 队列容量

    void resize(int newCapacity)
    {
        // 重新分配更大的数组
        T *newData = new T[newCapacity];
        for (int i = 0; i < size(); ++i)
        {
            newData[i] = data[(frontIndex + i) % capacity];
```

```
        }
        delete[] data;
        data = newData;
        frontIndex = 0;
        rearIndex = size();
        capacity = newCapacity;
    }

public:
    // 构造函数
    Queue(int initialCapacity = DEFAULT_CAPACITY) : frontIndex(0), rearIndex(0),
    ↪  capacity(initialCapacity)
    {
        data = new T[capacity];
    }
    // 析构函数
    ~Queue()
    {
        delete[] data;
    }
    void enQueue(const T &value) // 入队
    {
        // 如果队满，扩容
        if (size() == capacity - 1)
        {
            resize(2 * capacity);
        }
        data[rearIndex] = value; // 入队
        rearIndex = (rearIndex + 1) % capacity;
    }
    T deQueue() // 出队
    {
        // 如果队空，抛出异常
        if (isEmpty())
        {
            throw std::out_of_range("Queue underflow");
        }
        T value = data[frontIndex]; // 出队
        frontIndex = (frontIndex + 1) % capacity;
        return value;
    }
    T &front() // 获取队头元素
    {
        // 如果队空，抛出异常
        if (isEmpty())
        {
            throw std::out_of_range("Queue is empty");
        }
        return data[frontIndex];
    }
    bool isEmpty() const // 判断队是否为空
    {
        return frontIndex == rearIndex;
    }
    bool isFull() const // 判断队是否为满
    {
        return (rearIndex + 1) % capacity == frontIndex;
    }
    int size() const // 获取队大小
```

```
77      {
78          return (rearIndex - frontIndex + capacity) % capacity;
79      }
80      void clear() // 清空队
81      {
82          frontIndex = rearIndex = 0;
83      }
84  };
```

3 项目实现

3.1 基本功能的实现

3.1.1 输入队列

getInput 函数负责从用户那里接收输入，并根据输入的顾客编号将顾客分配到两个不同的队列中。这个函数首先读取顾客总数，然后逐个读取顾客编号，并根据编号的奇偶性将顾客分配到队列 A 或队列 B 中。

```
void handleQueue(Queue<int> &A, Queue<int> &B)
{
    cout << " 业务处理顺序为: ";
    while (!A.isEmpty() || !B.isEmpty())
    {
        // A 处理的速度是 B 的两倍，且优先输出 A
        if (!A.isEmpty())
        {
            cout << A.deQueue() << (A.size() == 0 && B.size() == 0 ? "" : " "); // 保证队尾不输出空格
        }
        if (!A.isEmpty())
        {
            cout << A.deQueue() << (A.size() == 0 && B.size() == 0 ? "" : " ");
        }
        if (!B.isEmpty())
        {
            cout << B.deQueue() << (A.size() == 0 && B.size() == 0 ? "" : " ");
        }
    }
    cout << endl;
}
```

3.2 业务处理

handleQueue 函数模拟了业务处理过程。它首先检查队列 A 是否为空，如果不为空，则处理队列 A 中的顾客。由于队列 A 的处理速度是队列 B 的两倍，因此在处理完一个顾客后，会再次检查队列 A 是否还有顾客需要处理。然后，它检查队列 B 是否为空，如果不为空，则处理队列 B 中的顾客。这个过程一直持续到两个队列都为空。

```
void handleQueue(Queue<int> &A, Queue<int> &B)
{
    cout << " 业务处理顺序为: ";
    while (!A.isEmpty() || !B.isEmpty())
    {
        // A 处理的速度是 B 的两倍，且优先输出 A
        if (!A.isEmpty())
        {
            cout << A.deQueue() << (A.size() == 0 && B.size() == 0 ? "" : " "); // 保证队尾不输出空格
```

```
        }
        if (!A.isEmpty())
        {
            cout << A.deQueue() << (A.size() == 0 && B.size() == 0 ? "" : " ");
        }
        if (!B.isEmpty())
        {
            cout << B.deQueue() << (A.size() == 0 && B.size() == 0 ? "" : " ");
        }
    }
    cout << endl;
}
```

4 项目测试

4.1 输入有效数据

4.1.1 正常测试，A 窗口人多

测试用例:

c

预期结果: 1 3 2 9 11 4 13 15

测试结果:

```
请输入队列，其中第一个数为顾客总数 n，接下来 n 个数为顾客编号(数字间以空格分隔)：
8

2 1 3 9 4 11 13 15
业务处理顺序为：1 3 2 9 11 4 13 15
```

4.1.2 正常测试，B 窗口人多

测试用例:

8

2 1 3 9 4 11 12 16

预期结果: 1 3 2 9 11 4 12 16

测试结果:

```
请输入队列，其中第一个数为顾客总数 n，接下来 n 个数为顾客编号(数字间以空格分隔)：
8

2 1 3 9 4 11 12 16
业务处理顺序为：1 3 2 9 11 4 12 16
```

4.1.3 最小 N

测试用例:

1

6

预期结果: 6

测试结果:

```
请输入队列，其中第一个数为顾客总数 n，接下来 n 个数为顾客编号(数字间以空格分隔)：
1 6
业务处理顺序为：6
```

4.2 健壮性测试

输入非法字符或负数，程序抛出异常，终止运行。

```
请输入队列，其中第一个数为顾客总数 n，接下来 n 个数为顾客编号(数字间以空格分隔)：
9 -2 3 q
terminate called after throwing an instance of 'std::invalid_argument'
  what():  Invalid input
```

5 项目心得与体会

通过本项目的深入开发与实践，我全面掌握了以下几项关键技能：

（1）模板类和队列结合的动态数据结构设计：学会了如何巧妙地将模板类的灵活性与队列的动态扩展性相结合，设计出既能适应不同类型数据存储需求，又能高效进行元素插入、删除和遍历的动态数据结构。

（2）数据验证与用户输入交互的细节优化：在项目实践中，深刻认识到用户输入的正确性对于程序稳定运行的重要性，并专注于优化数据验证逻辑，确保所有输入都符合预期格式和范围。

（3）算法效率的理解和实际操作能力：项目中涉及的队列处理问题，不仅考验了我对队列操作和数据结构算法的深入理解，还为解决更复杂的数据处理问题奠定了坚实的基础。在解决这个问题的过程中，我不断尝试并优化不同的算法策略，如双指针法、哈希表法等，从而深刻体会到了算法选择对效率提升的关键作用。

（4）实际应用与理论结合：通过将理论知识应用到实际项目中，我更加深刻地理解了队列数据结构的实际应用价值。这种从理论到实践的转换不仅加深了我对数据结构的理解，也提高了我解决实际问题的能力。

简而言之，通过本项目的实践，我不仅提高了对数据结构和算法的理解，还锻炼了自己的编程能力和解决问题的能力，为今后的学习和工作打下了坚实的基础。