同济大学

TONGJI UNIVERSITY

数据结构课程设计

项目名称	修理牧场
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

本项目实现了一个基于优先队列的霍夫曼编码树构建程序，并结合了一个木头锯切成本计算的实用工具。程序的主要功能是构建霍夫曼树，并计算给定长度的木头锯切成指定块数后的总成本。

1.2 项目功能分析

1.2.1 功能要求

农夫要修理牧场的一段栅栏，他测量了栅栏，发现需要 N 块木头，每块木头长度为整数 L_i 个长度单位，于是他购买了一个很长的，能锯成 N 块的木头，即该木头的长度是 L_i 的总和。但是农夫自己没有锯子，请人锯木的酬金跟这段木头的长度成正比。为简单起见，不妨就设酬金等于所锯木头的长度。例如，要将长度为 20 的木头锯成长度为 8，7 和 5 的三段，第一次锯木头将木头锯成 12 和 8，花费 20；第二次锯木头将长度为 12 的木头锯成 7 和 5 花费 12，总花费 32 元。如果第一次将木头锯成 15 和 5，则第二次将木头锯成 7 和 8，那么总的花费是 35（大于 32）.

1.2.2 输入要求

输入第一行给出正整数 N（$N \le 10^4$），表示要将木头锯成 N 块。第二行给出 N 个正整数，表示每块木头的长度。

1.2.3 输出要求

输出一个整数，即将木头锯成 N 块的最小花费。

1.2.4 项目实例

2 项目设计

2.1 数据结构设计

在霍夫曼编码项目中，数据结构的选择对于算法的效率至关重要。我们需要一个能够快速比较节点频率并构建二叉树的数据结构。以下是选择和设计数据结构的分析过程：

- 节点频率存储：需要一个数据结构来存储节点的频率，以便后续构建霍夫曼树。
- 节点频率比较需求：霍夫曼编码的核心在于根据节点频率构建树，因此需要一个能够快速比较频率大小的数据结构。
- 自动排序：在添加新节点后，数据结构应能自动维持元素的有序状态，以便快速找到最小元素。

基于以上分析，我们确定了以下数据结构：

2.1.1 优先队列 PriorityQueue

优先队列是一个能够自动排序的队列，它能够根据元素的优先级自动调整元素的顺序。在霍夫曼树的构建过程中，我们需要不断地取出频率最小的两个节点进行合并，这要求我们的优先队列能够快速地提供最小元素，并在合并后快速地插入新节点。

为了满足这些需求，我们选择了最大堆结构来实现优先队列。在最大堆中，父节点的值总是大于或等于子节点的值，这使得堆顶元素（即优先队列的最小元素）总是最小的节点。我们使用数组来存储堆中的元素，并通过上滤（heapifyUp）和下滤（heapifyDown）操作来维护堆的性质。以下是优先队列的一些关键特性：

（1）元素优先级：每个元素都有一个优先级，通常使用数值表示，数值越小优先级越高。

（2）出队顺序：元素出队时，总是优先级最高的元素先出队。

（3）入队操作：元素可以按照任意顺序入队。

（4）动态性：优先队列可以在运行时动态地添加和移除元素。

（5）无序性：优先队列内部元素的存储通常是无序的，即元素的存储顺序并不反映它们的优

2.1.2 霍夫曼树节点（HuffmanNode）

霍夫曼树是由多个节点构成的树结构，每个节点代表一个字符及其频率。在构建霍夫曼树的过程中，我们需要不断地合并频率最小的两个节点，创建新的节点，直到构建出完整的树结构。

为了表示霍夫曼树的节点，我们设计了 HuffmanNode 结构体，它包含节点的频率和指向左右子节点的指针。这样的设计允许我们灵活地构建和遍历霍夫曼树。

2.1.3 CompareHuffmanNode 比较器

CompareHuffmanNode 是一个自定义比较器，用于比较两个霍夫曼树节点的频率，确保小频率的节点优先处理。

2.2 类设计

2.2.1 PriorityQueue 类

PriorityQueue 类是优先队列的实现，它需要提供一系列操作来管理堆中的元素。这些操作包括插入、删除、获取最小元素等。我们使用模板类来实现，以支持不同类型的元素存储。提供以下操作：

- 构造函数：创建一个优先队列对象，可以指定队列的初始容量。
- 析构函数：销毁优先队列对象，释放内存。
- push：入队操作，将一个元素添加到队列中，并执行上滤操作。
- pop：出队操作，移除队列中优先级最高的元素，并执行下滤操作。
- top：获取栈顶元素即队列中优先级最高的元素。
- empty：判断队列是否为空。
- size：获取队列大小。

```
const int maxPQSize = 50; // 优先队列的最大容量

// 定义一个模板类 PriorityQueue，用于实现优先队列
template <typename T, typename Compare = less<T>>
class PriorityQueue
{
private:
    T *heap;       // 存储堆元素的数组
    int capacity; // 堆的容量
    int count;    // 当前堆中元素的数量
    Compare comp; // 比较器对象，用于定义堆的优先级规则

    // 上滤操作，维护堆的性质
    void heapifyUp(int index)
    {
        while (index > 0) // 如果当前节点不是根节点
        {
            int parent = (index - 1) / 2; // 计算父节点的索引
            if (!comp(heap[parent], heap[index]))
                break;                              // 如果当前节点不优于父节点，停止上滤
            std::swap(heap[index], heap[parent]); // 交换当前节点和父节点的位置
            index = parent;                         // 更新当前节点索引为父节点索引
        }
    }

    // 下滤操作，维护堆的性质
    void heapifyDown(int index)
    {
```

```
29          while (index < count) // 确保当前节点有子节点
30          {
31              int left = 2 * index + 1;  // 左子节点索引
32              int right = 2 * index + 2; // 右子节点索引
33              int largest = index;       // 假设当前节点是最大的
34
35              if (left < count && comp(heap[largest], heap[left]))
36              {
37                  largest = left; // 如果左子节点更优，更新 largest
38              }
39              if (right < count && comp(heap[largest], heap[right]))
40              {
41                  largest = right; // 如果右子节点更优，更新 largest
42              }
43              if (largest == index)
44                  break; // 如果当前节点已经是最优的，停止下滤
45
46              std::swap(heap[index], heap[largest]); // 交换当前节点和更优子节点
47              index = largest;                       // 更新当前节点索引为更优子节点的索引
48          }
49      }
50
51      // 扩容操作，增加堆的容量
52      void resize()
53      {
54          capacity *= 2;                // 容量加倍
55          T *newHeap = new T[capacity]; // 创建一个更大容量的数组
56          for (int i = 0; i < count; ++i)
57          {
58              newHeap[i] = heap[i]; // 复制旧堆元素到新堆中
59          }
60          delete[] heap;  // 释放旧堆内存
61          heap = newHeap; // 更新堆指针为新堆
62      }
63
64  public:
65      // 构造函数，初始化堆的容量和元素数量
66      PriorityQueue(int cap = maxPQSize, Compare comparator = Compare())
67          : capacity(cap), count(0), comp(comparator)
68      {
69          heap = new T[capacity]; // 分配容量为 cap 的数组
70      }
71
72      // 析构函数，释放堆数组的内存
73      ~PriorityQueue()
74      {
75          delete[] heap;
76      }
77
78      // 插入元素到堆中
79      void push(const T &value)
80      {
81          if (count == capacity) // 如果堆满了
82          {
83              resize(); // 扩容
84          }
85          heap[count] = value; // 将新元素放到堆的末尾
86          heapifyUp(count);    // 上滤操作，维护堆的性质
87          ++count;             // 增加堆中元素数量
```

```
    }

    // 删除堆顶元素
    void pop()
    {
        if (count == 0) // 如果堆为空
        {
            throw std::out_of_range("PriorityQueue is empty"); // 抛出异常
        }
        heap[0] = heap[--count]; // 将最后一个元素移到堆顶
        heapifyDown(0);          // 下滤操作，维护堆的性质
    }

    // 获取堆顶元素
    T top() const
    {
        if (count == 0) // 如果堆为空
        {
            throw std::out_of_range("PriorityQueue is empty"); // 抛出异常
        }
        return heap[0]; // 返回堆顶元素
    }

    // 判断堆是否为空
    bool empty() const
    {
        return count == 0;
    }

    // 获取堆中元素的数量
    size_t size() const
    {
        return count;
    }
};
```

2.2.2 HuffmanNode 类

HuffmanNode 类是霍夫曼树节点的实现，它包含节点的频率和指向左右子节点的指针。我们使用模板类来实现，以支持不同类型的元素存储。

```
// 霍夫曼树节点结构
struct HuffmanNode
{
    int freq;           // 频率，表示权值
    HuffmanNode *left;  // 左子树指针
    HuffmanNode *right; // 右子树指针

    HuffmanNode(int freq) : freq(freq), left(nullptr), right(nullptr) {}
};
```

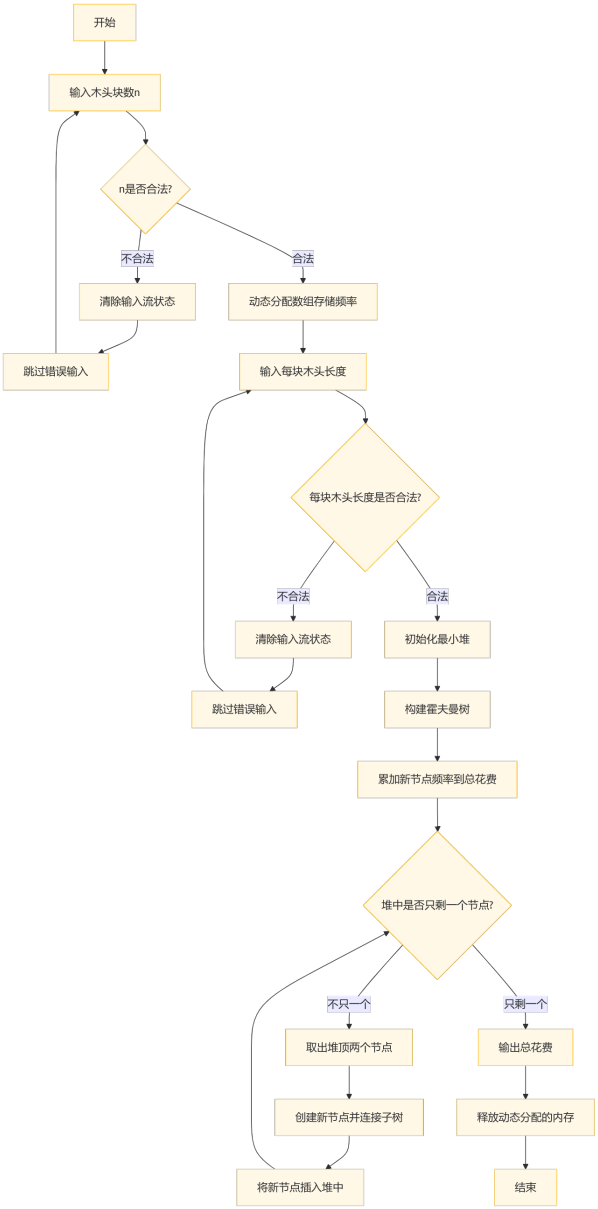
2.2.3 CompareHuffmanNode 比较器

CompareHuffmanNode 是一个自定义比较器，用于比较两个霍夫曼树节点的频率，确保小频率的节点优先处理。

```
// 自定义比较器，用于霍夫曼节点的比较
struct CompareHuffmanNode
{
    bool operator()(HuffmanNode *a, HuffmanNode *b) {return a->freq > b->freq;} // 小频率优先
};
```

3 项目实现

3.1 流程表示



3.2 功能实现

3.2.1 初始化霍夫曼树节点队列（initializeTree）

initializeTree 函数是霍夫曼树构建过程的起点，它负责将输入的频率数组转换为霍夫曼树节点，并初始化一个优先队列（最小堆）。

```cpp
// 初始化霍夫曼树节点队列
PriorityQueue<HuffmanNode *, CompareHuffmanNode> initializeTree(int freq[], int size)
{
    PriorityQueue<HuffmanNode *, CompareHuffmanNode> minHeap(maxPQSize); // 最小堆
    for (int i = 0; i < size; ++i)
    {
        minHeap.push(new HuffmanNode(freq[i])); // 将频率数组的元素转换为霍夫曼节点并插入堆中
    }
    return minHeap;
}
```

3.2.2 构建霍夫曼树（buildHuffmanTree）

buildHuffmanTree 函数负责根据优先队列中的节点构建霍夫曼树。在 buildHuffmanTree 函数中，我们使用一个循环来不断地从优先队列中取出两个最小频率的节点。每次循环，我们都会创建一个新的 HuffmanNode 实例，其频率等于两个取出节点的频率之和，并将这两个节点作为新节点的左右子节点。然后，我们将新节点推送回优先队列中。这个过程一直持续到优先队列中只剩下一个节点，这个节点就是霍夫曼树的根节点。在整个过程中，我们还维护一个全局变量 total_cost 来记录构建霍夫曼树的总成本，即所有节点频率的总和。

```cpp
// 构建霍夫曼树
HuffmanNode *buildHuffmanTree(PriorityQueue<HuffmanNode *, CompareHuffmanNode> &minHeap)
{
    HuffmanNode *left, *right, *top;

    while (minHeap.size() != 1) // 当堆中剩余元素不止一个
    {
        left = minHeap.top(); // 取出堆顶的最小频率节点作为左子节点
        minHeap.pop();

        right = minHeap.top(); // 取出堆顶的次小频率节点作为右子节点
        minHeap.pop();

        top = new HuffmanNode(left->freq + right->freq); // 创建新节点，其频率为左右子节点频率之和
        top->left = left;                                // 左子树连接
        top->right = right;                              // 右子树连接

        total_cost += top->freq; // 累加新节点的频率到总花费

        minHeap.push(top); // 将新节点插入堆中
    }

    return minHeap.top(); // 返回最终的霍夫曼树根节点
}
```

3.2.3 main 函数

在 main 函数中，我们首先提示用户输入木头要锯成的块数，并进行输入验证。如果输入不合法（非正整数），我们清除输入流状态并提示用户重新输入。接下来，我们动态分配一个数组 freq 来存储每块木头的长度，并再次进行输入验证。如果输入的长度不合法（非正数），我们同样清除输入流状态并提示用户重新输入指定块的长度。

输入验证通过后，我们调用 initializeTree 函数来初始化霍夫曼树节点队列，并调用 buildHuffmanTree 函数来构建霍夫曼树。最后，我们输出总花费，并释放动态分配的内存。

```
int main()
{
    int n;
    cout << " 请输入木头要锯成的块数 (正整数): ";
    while (true)
    {
        cin >> n;
        if (cin.fail() || n <= 0)
        {
            cin.clear();                                          // 清除输入流状态
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // 跳过错误输入
            cout << " 输入错误，请重新输入: ";
            continue;
        }
        else
        {
            break;
        }
    }

    int *freq = new int[n]; // 动态分配数组存储频率

    cout << " 请依次输入每块木头的长度: ";
    for (int i = 0; i < n; ++i)
    {
        cin >> freq[i];
        if (cin.fail() || freq[i] <= 0)
        {
            cerr << " 输入错误，请重新输入第 " << i + 1 << " 块及以后的木头的长度: ";
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            --i; // 回退索引以重新输入
        }
    }

    auto minHeap = initializeTree(freq, n);         // 初始化最小堆
    HuffmanNode *root = buildHuffmanTree(minHeap); // 构建霍夫曼树
    cout << " 总花费: " << total_cost << endl;      // 输出总花费

    delete[] freq; // 释放动态分配的内存

    return 0;
}
```

4 项目测试

4.1 输入有效数据

4.1.1 正常测试

测试用例:

8

4 5 1 2 1 3 1 1

预期结果: 49

测试结果:

```
请输入木头要锯成的块数(正整数): 8
请依次输入每块木头的长度: 4 5 1 2 1 3 1 1
总花费: 49
```

4.1.2 最小 N

木头块数为 1 时，不需要锯木头，花费为 0.

测试用例:

1

6

预期结果: 0

测试结果:

```
请输入木头要锯成的块数(正整数): 1
请依次输入每块木头的长度: 6
总花费: 0
```

4.2 健壮性测试

输入非法字符或负数，程序提示输入错误，重新输入。

```
请输入木头要锯成的块数(正整数): -2
输入错误，请重新输入: qw
输入错误，请重新输入: 3
请依次输入每块木头的长度: 1 q 3
输入错误，请重新输入第 2 块及以后的木头的长度: 2 siuhyfgv
输入错误，请重新输入第 3 块及以后的木头的长度: 5
总花费: 11
```

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：项目中涉及的最小堆处理问题，不仅考验了我对优先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。

（4）实际应用与理论结合：我学会了如何将抽象的数据结构和算法知识应用到具体的编程问题中，这不仅增强了我的理论知识，也提升了我的实践技能。通过构建霍夫曼树和计算锯切木头的成本，我学会了如何将复杂的理论问题转化为具体的编程任务，并找到有效的解决方案。

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。