数据结构课程设计文档

题目：修理牧场

姓名：赵卓冰

学号：2252750

专业：软件工程

年级：2023 级

指导教师：张颖

2024 年 11 月 20 日

1. 运行环境与开发工具

本项目支持在以下开发环境和编译运行环境中运行：

- **Windows 操作系统**：
  - 版本：Windows 10 x64
  - IDE：Visual Studio 2022 (Debug 模式)
  - 编译器：MSVC 14.39.33519
- **Linux 操作系统**：
  - 版本：Ubuntu 20.04.6 LTS
  - IDE：VS Code
  - 编译器：gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)

2. 项目要求

本项目旨在实现一个高效的锯木头最小费用计算系统，通过模拟木头分段锯开的过程，计算完成锯木所需的最小花费。农夫需要将一整块木头锯成若干段，每次锯木头的费用等于所锯木头的长度，为了降低总花费，需要选择最佳的锯木策略。

2.1. 功能要求

1. **输入说明**：
   - 输入第一行：正整数 (N) ((N < 10^4))，表示将木头锯成的段数。
   - 输入第二行：(N) 个正整数，表示每段木头的长度。
2. **输出说明**：
   - 输出一个整数，表示将木头锯成 (N) 段的最小花费。

3. 项目简介

本项目实现了一个基于最小堆的算法，用于计算将一段木头锯成 (N) 段所需的最小费用。锯木的费用与锯开的木头长度成正比，因此通过贪心策略，每次选择当前最短的两段木头合并，最终得出总的最小花费。

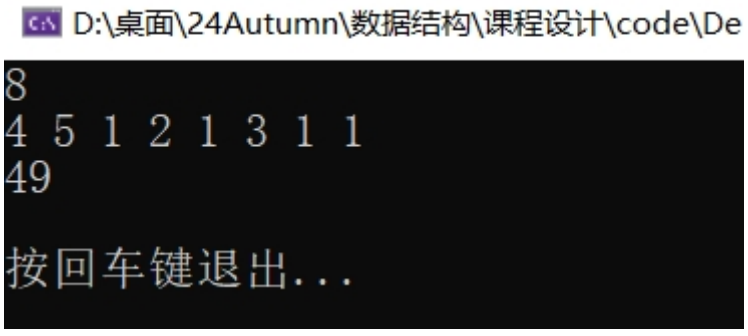
项目的主要功能包括：

- **木段长度输入**：用户输入木段数量和每段长度。
- **锯木花费计算**：基于最小堆的贪心策略，逐步合并长度最小的两段木头，记录总花费。
- **输出结果**：程序计算并输出最小花费。

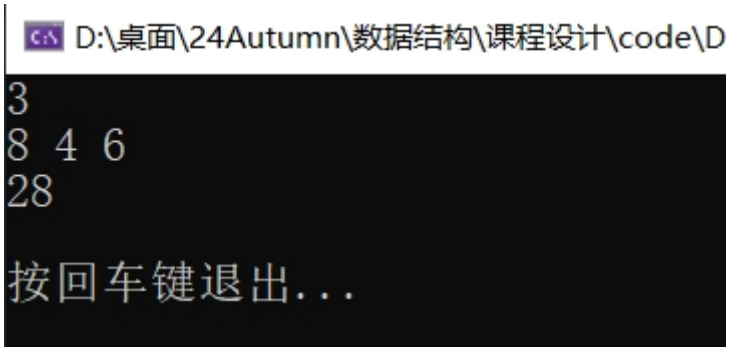
4. 测试界面展示

4.1. Windows 界面

- **测试用例1**：



- **测试用例2**：



4.2. Linux 界面

- **测试用例1**:

```
bing@bing-virtual-machine:~/ds$ g++ -o framRepair framRepair.cpp
bing@bing-virtual-machine:~/ds$ ./framRepair
8
4 5 1 2 1 3 1 1
49

按回车键退出...
```

- **测试用例2**:

```
bing@bing-virtual-machine:~/ds$ ./framRepair
3
8 4 6
28

按回车键退出...
```

5. 代码架构

项目主要包含两个核心部分：`MinHeap` 类和主函数。`MinHeap` 类用于实现最小堆操作，而主函数负责控制台交互及贪心策略的实现。

5.1. MinHeap 类

`MinHeap` 类是一个最小堆，用于动态存储木段长度并支持快速获取最小值。

- **核心成员函数**:
  - `Push()`：向堆中插入一个元素，并调整堆结构。
  - `Pop()`：移除堆顶元素（最小值），并调整堆结构。
  - `Top()`：获取堆顶元素（最小值）。
  - `HeapifyUp()`：插入元素时向上调整堆。
  - `HeapifyDown()`：移除堆顶元素后向下调整堆。
- **代码实现**

```cpp
// 最小堆类
class MinHeap {
private:
  int* data;
  size_t size;
  size_t capacity;

  void Resize() {
    size_t new_capacity = (capacity == 0) ? 1 : capacity * 2;
    int* new_data = new int[new_capacity];
    for (size_t i = 0; i < size; ++i) {
      new_data[i] = data[i];
    }
    delete[] data;
```

```
    data = new_data;
    capacity = new_capacity;
  }

  // 将index位置的数向上调整
  void HeapifyUp(size_t index) {
    while (index > 0) {
      size_t parent = (index - 1) / 2;
      if (data[index] < data[parent]) {
        int temp = data[index];
        data[index] = data[parent];
        data[parent] = temp;
        index = parent;
      }
      else {
        break;
      }
    }
  }

  // 将index位置的数向下调整
  void HeapifyDown(size_t index) {
    while (index * 2 + 1 < size) {
      size_t left = index * 2 + 1;
      size_t right = index * 2 + 2;
      size_t smallest = left;
      if (right < size && data[right] < data[left]) {
        smallest = right;
      }
      if (data[index] > data[smallest]) {
        int temp = data[index];
        data[index] = data[smallest];
        data[smallest] = temp;
        index = smallest;
      }
      else {
        break;
      }
    }
  }

public:
  MinHeap() {
    data = nullptr;
    size = 0;
    capacity = 0;
  }

  ~MinHeap() {
    delete[] data;
  }

```

```
  size_t Size() {
    return size;
  }

  void Push(const int num) {
    if (size + 1 >= capacity) {
      Resize();
    }
    data[size] = num;
    HeapifyUp(size);
    ++size;
  }

  void Pop() {
    if (0 == size) {
      throw out_of_range("Heap is empty!");
    }
    data[0] = data[size - 1];
    --size;
    HeapifyDown(0);
  }

  int Top() {
    if (0 == size) {
      throw out_of_range("Heap is empty!");
    }
    return data[0];
  }

  bool Empty() {
    return (0 == size);
  }

  void Disp() {
    for (int i = 0; i < size; ++i) {
      cout << data[i] << ' ';
    }
    cout << endl;
  }
};
```

5.2. 主函数

主函数通过以下步骤实现锯木花费的计算：

1. 将所有木段长度插入最小堆。
2. 依次取出堆中最小的两段长度，合并后将新长度重新插入堆。
3. 每次合并的长度累加到总花费中。
4. 输出最终花费。

```cpp
int main() {
  int N;
  cin >> N;
  MinHeap min_heap;
  // 将木块的长度push进最小堆
  for (int i = 0; i < N; ++i) {
    int length;
    cin >> length;
    min_heap.Push(length);
  }
  long long total_cost = 0;
  // 当最小堆中木头个数多于一个
  while (min_heap.Size() > 1) {
    // 取出第一个最小的木板
    int len1 = min_heap.Top();
    min_heap.Pop();
    // 取出第二个最小的木板
    int len2 = min_heap.Top();
    min_heap.Pop();
    // 新的木板的长度
    int new_len = len1 + len2;
    // 合并木板花费的费用
    total_cost += new_len;
    // 将木板push进最小堆
    min_heap.Push(new_len);
  }
  cout << total_cost << endl;
  cout << endl << "按回车键退出..." << endl;
#ifdef _WIN32
  while (_getch() != '\r')
    continue;
#endif
  return 0;
}
```

6. 功能模块介绍

6.1. 最小堆实现

`MinHeap` 类通过数组存储数据，支持动态扩容，维护堆结构使得堆顶始终是最小值。主要功能包括插入、删除和获取堆顶元素，时间复杂度均为 (O(\log n))。

6.2. 贪心策略的花费计算

主函数采用贪心算法，每次选择当前最短的两段木头进行合并，确保每次锯木的长度最短，从而降低总花费。

6.3. 输入与输出处理

通过标准输入读取木段数量 (N) 和长度列表，并通过标准输出打印最小花费。

7. 用户交互

用户在命令行中输入木段数量和长度，程序将完成以下操作：

1. 将所有木段长度插入最小堆。
2. 按贪心策略依次取出最小的两段木头合并，记录合并长度和费用。
3. 最终输出锯木的最小总花费。

7.1. 示例交互

- **输入**：

```
8
4 5 1 2 1 3 1 1
```

- **输出**：

```
49
```

- **输入**：

```
3
8 4 6
```

- **输出**：

```
28
```

8. 性能考虑

- 最小堆的插入和删除操作时间复杂度为 `O(logn)`，初始化堆需要 `O(n)`。整个算法需要 `N-1` 次合并操作，因此总时间复杂度为 `O(nlogn)`。
- 空间复杂度为 `O(n)`，用于存储最小堆。

该算法高效且适用于木段数量较多的情况。

9. 错误处理和输入验证

为了保证输入合法性，本项目对输入进行了严格的检查：

- 验证 (N) 是否为正整数，且 (N < 10^4)。
- 验证每段木头长度是否为正整数。

若输入不合法，程序会报错并终止运行。

10. 未来改进

1. **支持多种费用计算模式**：扩展支持不同的锯木费用计算规则。

2. **优化输出展示**：为用户提供更详细的锯木过程，如每次合并的木段长度。
3. **可视化展示**：通过图形界面动态展示木段合并的过程和费用变化。

11. 心得体会

通过本项目的开发，我深入理解了堆数据结构及其在贪心算法中的应用，尤其是在动态维护最小值场景下的高效性。该项目体现了算法设计中的局部最优策略（贪心）的价值，锯木费用的最小化问题是数据结构与算法结合的一个典型案例。