修理牧场

1651573 刘客

功能介绍

- 输入格式：输入第一行给出正整数N，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。
- 输出格式：输出一个整数，即将木头锯成N块的最小花费。

编译说明

- 在windows平台下的.exe文件
- 在Linux平台下的out文件

一 . 设计

1. 数据结构设计

> 分析题干,可得本题所要求的数据结构的本质是huffman树,故通过一个**树**的数据结构以及一个**最小堆**来解题。

2. 类的设计

- ## HuffmanNode 节点
  - 成员变量

成员名称	属性	类型	描述
data	private	int	存储的数值
leftChild	private	HuffmanNode*	指向左孩子的指针
rightChild	private	HuffmanNode*	指向右孩子的指针

  - 成员函数

函数名称	返回值类型	描述
HuffmanNode	无	构造函数
getData	int	获得当前节点的数值
setData	void	设置当前节点存储的数值
setLeft	void	设置左孩子
setRight	void	设置右孩子

|getLeft|HuffmanNode|*获得指向左孩子的指针* |*getRight*|*HuffmanNode*|获得指向右孩子的指针

- **MyHeap 节点**
  - 成员变量

成员名称	属性	类型	描述
heapContainer	private	HuffmanNode**	存储HuffmanNode* 的数组
currentSize	private	int	当前数组的容量
maxSize	private	int	数组的最大容量

  - 成员函数

函数名称	返回值类型	描述
Myheap	无	构造函数
~MyHeap	无	析构函数
Insert	bool	插入节点函数
RemoveMin	HuffmanNode*	将堆顶元素移除的函数
isFull	bool	判断数组是否已满
isEmptey	bool	判断数组是否为空

|shiftUp|void|向上转移函数 |shiftDown|void|向下转移函数

- **HuffmanTree 节点**
  - 成员变量

成员名称	属性	类型	描述
root	private	HuffmanNode*	根节点
huffmanheap	private	MyHeap*	指向最小堆的指针

- 成员函数

函数名称	返回值类型	描述
HuffmanTree	无	构造函数
~HuffmanTree	无	析构函数
mergeTree	void	将子树合并
getRoot	HuffmanNode*	获得树的根节点
insertNode	void	插入节点
isEmptey	bool	判断堆中数组是否为空
remove	HuffmanNode*	移除堆顶元素

|countWeight|int|计算权重

二. 类的具体实现

1. myHeap类
   - 构造函数

> 为Huffman指针节点容器分配内存

```
MyHeap::MyHeap(int n) {
    //为Huffman指针节点容器分配内存
    heapContainer = new HuffmanNode*[n+1];
    maxSize = n;
    currentSize = 0;
}
```

   - 向下转移函数

> 因为存储的是HuffmanNode*,需要从中取出data进行比较

```
void MyHeap::shiftDown(int start, int m) {
    int i = start;
    int j = 2 * start + 1;
    HuffmanNode* tempNode = heapContainer[i];
    while (j <= m) {
        if (j < m && heapContainer[j]->getData() >= heapContainer[j + 1]-
>getData()) {
            j++;
        }
        if (tempNode->getData() <= heapContainer[j]->getData()) {
            break;
        }
        else {
            heapContainer[i] = heapContainer[j];
            i = j;
```

```
            j = 2 * j + 1;
        }
    }
    heapContainer[i] = tempNode;
}
```

- 向上转移函数

```
void MyHeap::shiftUp(int start) {
    int i = start;
    int j = (i - 1) / 2;
    HuffmanNode* tempNode = heapContainer[i];
    //当i == 0时循环结束
    while (i > 0) {
        if (heapContainer[j]->getData() <= tempNode->getData()) {
            break;
        }
        else {
            heapContainer[i] = heapContainer[j];
            i = j;
            j = (j - 1) / 2;
        }
    }
    heapContainer[i] = tempNode;
}
```

- 插入函数

```
bool MyHeap::Insert(HuffmanNode* node) {
    //当已满时,给出错误信息并退出
    if (isFull()) {
        cerr << "最大堆已满,插入失败!!" << endl;
        exit(1);
    }
    else {
        //否则,在容器中加入节点,并使用向上转移函数重新排序
        heapContainer[currentSize] = node;
        shiftUp(currentSize);
        currentSize++;
        return true;
    }
}
```

- 移除堆顶元素函数

```
HuffmanNode* MyHeap::RemoveMin() {
    //当堆中无元素时,给出错误信息
    if (currentSize == 0) {
        cerr << "堆中无元素,请确认您的操作是否正确\n";
        return NULL;
    }
```

```
    else {
        //将堆顶节点返回,并将堆尾的节点提到堆顶,使用向下转移算法重新排序
        HuffmanNode* tempNode = heapContainer[0];
        heapContainer[0] = heapContainer[currentSize - 1];
        currentSize--;
        shiftDown(0, currentSize - 1);
        return tempNode;
    }
}
```

2. HuffmanTree类
    - 构造函数

```
HuffmanTree::HuffmanTree(int num) {
    huffmanheap = new MyHeap(num);
    //一开始还没有将树合并,所以祖宗节点为NULL
    root = NULL;
}
```

    - 合并树函数

> 对两个左右孩子节点建立一个父亲节点,并将它塞入堆中,如果堆为空,则表明该父亲节点为最后的祖宗节点

```
void HuffmanTree::mergeTree(HuffmanNode* left, HuffmanNode* right) {
    HuffmanNode* parent = new HuffmanNode(left->getData() + right->getData(),
left, right);
    //如果堆为空的话,说明所有节点已经处理完毕,此时就把根节点设为当前合并所得到的节点
    if (huffmanheap->isEmptey()) {
        setRoot(parent);
    }
    huffmanheap->Insert(parent);
}
```

    - 计算权重函数

```
//递归计算
int HuffmanTree::countWeight(HuffmanNode* node) {
    //不计算root节点的data,其他节点的data全部计入
    if (node == root) {
        return countWeight(node->getLeft()) + countWeight(node->getRight());
    }
    else if (node->getLeft() != NULL && node->getRight() != NULL) {
        return countWeight(node->getLeft()) + countWeight(node->getRight()) +
node->getData();
    }
    else {
        return node->getData();
    }
}
```

- 删除树操作

> 递归删除,将空间全部释放掉

```
void HuffmanTree::deleteTree(HuffmanNode* node) {
    if (node->getLeft() != NULL && node->getRight() != NULL) {
        deleteTree(node->getLeft());
        deleteTree(node->getRight());
    }
    delete node;
}
```

三. 主程序实现

- 对堆的逻辑操作

> 使用循环,当堆不为空时,持续进行循环 每次推出两个节点,进行merge操作,再将生成的节点推入堆中

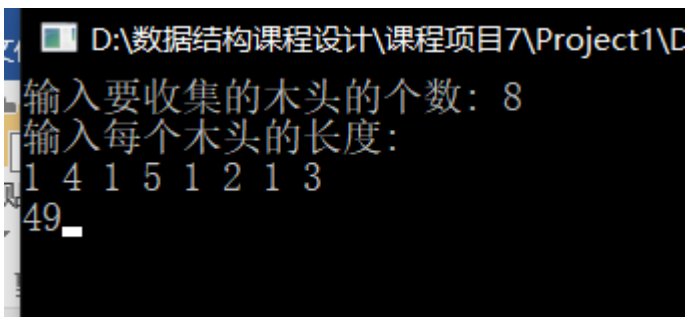
```
  while (!huffTree->isEmptey()) {
          bool empty = false;
          left = huffTree->remove();
          right = huffTree->remove();
          if (huffTree->isEmptey()) {
              empty = true;
          }
          huffTree->mergeTree(left, right);
          if (empty) {
              break;
          }
      }
```

- 计算权重

> 直接调用huffmanTree中的递归计算权重函数

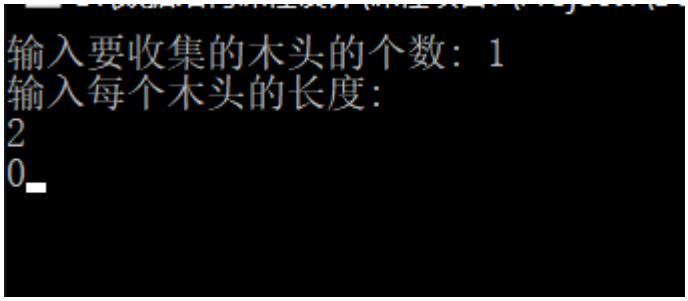
```
  cost = huffTree->countWeight(huffTree->getRoot());
```

四. 程序运行效果



五. 边界测试

1. 输入的n为1时


输入要收集的木头的个数：1
输入每个木头的长度：
2
0