山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：202300130183	姓名： 宋浩宇	班级：23 级人工智能班
实验题目：2024 级数据结构--数据智能 实验 10 堆		
实验学时：2	实验日期：2024/11/20	

实验目的：

1. 掌握堆结构的定义与实现；
2. 掌握堆结构的使用。

软件开发工具：

1. visual studio code 2022（使用 C/C++、C/C++ Extension Pack、C/C++ Themes 插件）
2. mingw64 工具包

1. 实验内容

完成 2024 级数据结构--数据智能 实验 10 堆 的 A 堆的操作 B 哈夫曼编码

2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

本题主要使用堆数据结构，但第二问可以不使用堆数据结构就解决甚至可以计算的更快。

我们先来描述 A 题堆的数据结构和算法实现。

首先 A 题让我们使用最小堆来解决，根据定义，最小堆就是满足每个节点都小于它的子节点的堆，其中根节点为最小的节点。我们需要实现的主要是插入和删除两个操作，根据定义，堆的插入操作是在堆的最底层最靠左的位置插入节点，且还需要让堆满足原本的大小关系。堆的删除操作则是删除堆的根节点。这个删除和插入的实现和我们利用堆来排序有较大关系。首先是插入，我们每次插入的位置都是已知的，不过为了让这个堆可以构成一个树，我们需要确认插入的节点的父节点以及确认这个节点是它父节点的左子节点还是右子节点，幸运的是，堆是除了最后一层都是满的二叉树，因此我们可以很简单的使用下标减一除以二的方式来得到它的父节点的下标，只需要开一个数组存下来所有节点的指针即可轻易实现这一点。在成功将数据导入堆之后，还需要进行数据上浮来保证堆的有序性。具体的方式是，将插入的数据和它父节点的数据进行比较，如果插入的数据小于它的父节点的数据，则将这个数据和它父节点的数据进行交换，重复以上过程直到它没有父节点（为根节点）或它的父节点的数据小于它的数据。至于删除操作，于插入相对，会有一个数据下沉的过程，来保证整个堆的有序性，首先删除根节点，然后将序号最靠后的子节点的数据转移到原本根节点的位置，然后进行数据下沉，与数据上浮相对应，数据下沉就是让这个数据和他的子节点作比较，和子节点中的较小者进行交换，直到这个数据没有子节点（为叶节点）或它的子节点都比它大的时候停止。然后是堆排序算法，堆排序算法使用堆数据结构来完成，首先我们可以知道，在最小堆中根节点是最小的，因此我们可以将一个序列以堆的方式来储存，然后依次输出堆的根节点并删除这个根节点直到把堆清空,这样我们就获得了一个时间复杂度为 O(nlogn) 的排序算法。B 题，首先我们要先了解哈夫曼编码的核心思想，哈夫曼编码的核心思想就是用较短的编码来描述出现频率较高的数据，用较长的编码来描述出现频率较低的数据。而通过哈夫曼树构成的哈夫曼编码就是平均编码数最短的哈夫曼编码，哈夫曼编码是无损压缩。OK 我们接下来解释哈夫曼编码的计算过程。首先我们将每一个字符都视为一个节点（再将这个节点视为一棵树的根节点），然后我们可以统计每一个字符出现的次数，再将这些节点根据字符出现的次数进行排序（将这些树根据根节点记录的出现次数），然后我们取出其中的出现次数最少的两个字符，新建一个节点作为他们的根节点构成一棵子树，这个新的根节点记录的出现次数为两个子节点的记录的出现次数的和。重复以上过程直到我们将这些节点组合成一棵树。之后在分别计算每一个叶节点到根节点的最短距离（每条边的权都视为 1），然

后我们就可以获得每个叶节点对应的字符需要的编码的长度（就是这些最短距离的值）。之后我们根据它们单个字符的编码长度再乘各个字母出现的频数就可以获得最终的哈夫曼编码的长度。

3. 测试结果（测试输入，测试输出）

A 题测试输入：

```
10
-225580 113195 -257251 384948 -83524 331745 179545 293165 125998 376875
10
1 -232502
1 -359833
1 95123
2
2
2
1 223971
1 -118735
1 -278843
3 10
-96567 37188 -142422 166589 -169599 245575 -369710 423015 -243107 -108789
```

输出：

```
-257251
-257251
-359833
-359833
-257251
-232502
-225580
-225580
-225580
-278843
-369710 -243107 -169599 -142422 -108789 -96567 37188 166589 245575 423015
```

B 题测试输入：

```
Abcdabcaba
```

输出：

```
19
```

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

从测试结果来看，我们的算法成功解决了这个问题。存在的问题主要是，我们的堆在初始化的时候并没有直接开辟这块内存，而是随着数据的插入逐步开辟内存，这可能会使我们插入数据的效率比较低，并且我们的堆因为使用的动态申请的内存来进行储存，实际的内存清理和回收过程会比较麻烦。解决方法的话，我们在初始化一个堆的时候，就提前把这些节点都申请好并且将节点之间的父子关系都链接好就可以解决插入较慢的问题，但同时也会提高内存的占用。清理内存的话，只要在代码中合理的管理内存即可。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
1.  /*2024 级数据结构--数据智能 实验 10 堆 A 堆的操作.cpp*/
2.  #include <iostream>
3.
```

```
4.  using namespace std;
5.
6.  template<class T>
7.  struct Node
8.  {
9.      T data;
10.     Node<T>* left, * right, * parent;
11.     size_t depth;
12.     size_t sons;
13.     bool touched;
14.     Node(Node<T>* parent, const T& data) : data(data), left(nullptr
    ), right(nullptr), parent(parent), depth(0), sons(0), touched(false)
    {}
15.     Node(): left(nullptr), right(nullptr), parent(nullptr), depth(0
    ), sons(0), touched(false) {}
16. };
17.
18.
19. template <typename T>
20. class Heap
21. {
22. protected:
23.     Node<T>** data;
24.     size_t count;
25.     size_t capacity;
26. public:
27.     Heap(size_t capacity);
28.     ~Heap();
29.     virtual void push(const T& value) = 0;
30.     virtual T pop() = 0;
31.     T& top();
32.     bool empty() const;
33.     size_t size() const;
34.     void print() const;
35. };
36.
37. template <typename T>
38. Heap<T>::Heap(size_t capacity)
39. {
40.     this->capacity = capacity;
41.     this->count = 0;
42.     this->data = new Node<T>*[capacity];
43. }
44.
45. template <typename T>
46. Heap<T>::~Heap()
```

```
47.  {
48.      delete[] data;
49.  }
50.
51.  template <typename T>
52.  T& Heap<T>::top()
53.  {
54.      if (empty())
55.      {
56.          return *(new T(0));
57.      }
58.      else
59.      {
60.          return (*data[0]).data;
61.      }
62.  }
63.
64.  template <typename T>
65.  bool Heap<T>::empty() const
66.  {
67.      return count == 0;
68.  }
69.
70.
71.  template <typename T>
72.  size_t Heap<T>::size() const
73.  {
74.      return count;
75.  }
76.
77.  template <typename T>
78.  void Heap<T>::print() const
79.  {
80.      for (size_t i = 0; i < count; i++)
81.      {
82.          cout << (*data[i]).data << " ";
83.      }
84.      cout << endl;
85.  }
86.
87.
88.  template <typename T>
89.  class minHeap : public Heap<T>
90.  {
91.  public:
92.      minHeap(size_t capacity) : Heap<T>(capacity) {}
```

```
93.       virtual void push(const T& value) final override;
94.       virtual T pop() final override;
95.   };
96.
97.   template <typename T>
98.   void minHeap<T>::push(const T& value)
99.   {
100.      if (this->data == nullptr)
101.      {
102.          this->data = new Node<T>*[this->capacity];
103.      }
104.      if (this->count == this->capacity)
105.      {
106.          Node<T>** temp = new Node<T>*[this->capacity * 2];
107.          for (size_t i = 0; i < this->capacity; i++)
108.          {
109.              temp[i] = this->data[i];
110.          }
111.          delete[] this->data;
112.          this->data = temp;
113.          this->capacity *= 2;
114.      }
115.      if (this->count == 0)
116.      {
117.          this->data[0] = new Node<T>(nullptr, value);
118.          this->count++;
119.      }
120.      else
121.      {
122.          Node<T>* temp = new Node<T>(this->data[int(this->count - 1)
    / 2], value);
123.          this->data[this->count] = temp;
124.          this->count++;
125.          if (this->count % 2 == 0)
126.          {
127.              temp->parent->left = temp;
128.          }
129.          else
130.          {
131.              temp->parent->right = temp;
132.          }
133.          while (temp != nullptr && temp->parent != nullptr && temp->
   data < temp->parent->data)
134.          {
135.              swap(temp->data, temp->parent->data);
136.              temp = temp->parent;
```

```
137.         }
138.     }
139. }
140. 
141. template <typename T>
142. T minHeap<T>::pop()
143. {
144.     if (this->count == 0)
145.     {
146.         return *(new T());
147.     }
148.     T value = this->data[0]->data;
149.     this->count--;
150.     if (this->count == 0)
151.     {
152.         return value;
153.     }
154.     else
155.     {
156.         swap(this->data[0]->data, this->data[this->count]->data);
157.         if (this->count % 2 == 0)
158.         {
159.             this->data[this->count] -> parent -> right = nullptr;
160.         }
161.         else
162.         {
163.             this->data[this->count] -> parent -> left = nullptr;
164.         }
165.         delete this->data[this->count];
166.         this->data[this->count] = nullptr;
167.         Node<T>* temp = this->data[0];
168. 
169.         while (1)
170.         {
171.             if (temp->left != nullptr && temp->right != nullptr &&
    temp->left->data <= temp->right->data && temp->left->data < temp->dat
    a)
172.             {
173.                 swap(temp->data, temp->left->data);
174. 
175.                 temp = temp->left;
176.             }
177.             else if (temp->right != nullptr && temp -> right != nul
    lptr && temp->right->data < temp->left->data && temp->right->data < t
    emp->data)
178.             {
```

```
                    swap(temp->data, temp->right->data);

                    temp = temp->right;
                }
                else if (temp->left != nullptr && temp->right == nullpt
   r && temp->left->data < temp->data)
                {
                    swap(temp->data, temp->left->data);

                    temp = temp->left;
                }
                else if (temp->right != nullptr && temp->left == nullpt
   r && temp->right->data < temp->data)
                {
                    swap(temp->data, temp->right->data);

                    temp = temp->right;
                }
                else
                {
                    break;
                }
            }
        }
        // this->print();
        return value;
    }


    class Solution
    {
    public:
        void solve();
    };

    void Solution::solve()
    {
        size_t n;
        cin >> n;
        minHeap<int> heap(n);
        for (size_t i = 0; i < n; i++)
        {
            size_t x;
            cin >> x;
            heap.push(x);
        }
```

```
223.      cout << heap.top() << endl;
224.      size_t m;
225.      cin >> m;
226.      for (size_t i = 0; i < m; i++)
227.      {
228.          size_t operation;
229.          cin >> operation;
230.          if (operation == 1)
231.          {
232.              size_t x;
233.              cin >> x;
234.              heap.push(x);
235.              cout << heap.top() << endl;
236.          }
237.
238.          else if (operation == 2)
239.          {
240.              heap.pop();
241.              cout << heap.top() << endl;
242.          }
243.          else if (operation == 3)
244.          {
245.              size_t cnt;
246.              cin >> cnt;
247.              minHeap<int> sort_heap(cnt);
248.              for (size_t j = 0; j < cnt; j++)
249.              {
250.                  size_t x;
251.                  cin >> x;
252.                  sort_heap.push(x);
253.              }
254.              for (size_t j = 0; j < cnt; j++)
255.              {
256.                  cout << sort_heap.pop() << " ";
257.              }
258.          }
259.      }
260.  }
261.
262.  int main()
263.  {
264.      Solution solution;
265.      solution.solve();
266.      return 0;
267.  }
268.  /*2024 级数据结构--数据智能 实验 10 堆 B 霍夫曼编码.cpp*/
```

```
269.
270. #include<iostream>
271.
272. using namespace std;
273.
274. size_t count[26];
275. size_t sum = 0;
276.
277. template<class T>
278. struct Node
279. {
280.     T data;
281.     Node<T>* left, * right, * parent;
282.     size_t depth;
283.     size_t sons;
284.     bool touched;
285.     Node(Node<T>* parent, const T& data) : data(data), left(nullptr
    ), right(nullptr), parent(parent), depth(0), sons(0), touched(false)
    {}
286.     Node() : left(nullptr), right(nullptr), parent(nullptr), depth(
    0), sons(0), touched(false),data(0) {}
287. };
288.
289. template<class T>
290. class huffmanTree
291. {
292. public:
293.     huffmanTree() : root(nullptr) {}
294.     huffmanTree(huffmanTree<T>& tree_1, huffmanTree<T>& tree_2);
295.     huffmanTree(T data,size_t frequency);
296.     void ini_code_length(Node<T>* node);
297.     size_t getFrequency() const { return frequency; }
298.     Node<T>* getRoot() const { return root; }
299. private:
300.     Node<T>* root;
301.     size_t frequency;
302.     size_t code_length;
303. };
304.
305. template<class T>
306. huffmanTree<T>::huffmanTree(T data, size_t frequency)
307. {
308.     this->root = new Node<T>(nullptr, data);
309.     this->frequency = frequency;
310. }
311.
```

```cpp
template<class T>
huffmanTree<T>::huffmanTree(huffmanTree<T>& tree_1, huffmanTree<T>& tree_2)
{
    Node<T>* node_1 = tree_1.root;
    Node<T>* node_2 = tree_2.root;
    Node<T>* node_3 = new Node<T>();
    this->root = node_3;
    node_3->left = node_1;
    node_3->right = node_2;
    node_1->parent = node_3;
    node_2->parent = node_3;
    this->frequency = tree_1.frequency + tree_2.frequency;
}

template<class T>
void huffmanTree<T>::ini_code_length(Node<T>* node)
{
    if (node == nullptr)
    {
        return;
    }
    if (node == this->root)
    {
        node->depth = 0;
        ini_code_length(node->left);
        ini_code_length(node->right);
    }
    else
    {
        node->depth = node->parent->depth + 1;
        if (node->left == nullptr && node->right == nullptr)
        {
            // cout << node->data << " " << node->depth << endl;
            sum += node->depth * count[node->data - 'a'];
        }
        ini_code_length(node->left);
        ini_code_length(node->right);
    }
}

class Solution
{
public:
    void solute();
};
```

```
357.
358. size_t strlen(const char* str) {
359.     size_t len = 0;
360.     while (str[len]) { len++; }
361.     return len;
362. }
363.
364. void Solution::solute()
365. {
366.
367.     char string[1000000];
368.     size_t cnt = 0;
369.     huffmanTree<char>* tree[26];
370.     cin >> string;
371.     size_t length = strlen(string);
372.     for (size_t i = 0; i < 26; i++)
373.     {
374.         count[i] = 0;
375.     }
376.     for (size_t i = 0; i < length; i++)
377.     {
378.         count[string[i] - 'a']++;
379.     }
380.     for (size_t i = 0; i < 26; i++)
381.     {
382.         if (count[i] != 0)
383.         {
384.             tree[cnt] = new huffmanTree<char>(char(i + 'a'), count[
    i]);
385.             cnt++;
386.         }
387.     }
388.     while (cnt > 1)
389.     {
390.         for (size_t i = 0; i < cnt; i++)
391.         {
392.             for (size_t j = 0; j < cnt - i - 1; j++)
393.             {
394.                 if (tree[j]->getFrequency() < tree[j + 1]->getFrequ
    ency())
395.                 {
396.                     swap(tree[j], tree[j + 1]);
397.                 }
398.             }
399.         }
400.         // cout << endl;
```

```
401.             // for (size_t i = 0; i < cnt; i++)
402.             // {
403.             //     cout << tree[i]->getFrequency() << endl;
404.             // }
405.             // cout << endl;
406.             huffmanTree<char>* tree_1 = tree[cnt - 1];
407.             huffmanTree<char>* tree_2 = tree[cnt - 2];
408.             // delete tree_1;
409.             // delete tree_2;
410.             huffmanTree<char>* tree_3 = new huffmanTree<char>(*tree_1,
    *tree_2);
411.             tree[cnt - 2] = tree_3;
412.             cnt--;
413.         }
414.         tree[0]->ini_code_length(tree[0]->getRoot());
415.         cout << sum << endl;
416.     }
417.
418.     int main()
419.     {
420.         Solution s;
421.         s.solute();
422.         return 0;
423.     }
```