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：202000130198 | 姓名： 隋春雨 | | 班级： 20.4 |
| 实验题目：堆及其应用 | | | |
| 实验学时：2 | | 实验日期： 2021-12-2 | |
| 实验目的：  1、掌握堆结构的定义、描述方法、操作定义及实现。  2、掌握堆结构的应用。 | | | |
| 软件开发环境：  CLION2020 | | | |
| 1. 实验内容   ①题目描述：  创建最小堆类，使用数组作为存储结构，提供操作：插入、删除、初始化、排序。保证第一个操作是建堆操作，接下来是对堆的插入和删除操作，删除和插入都在建好的堆上进行操作。  输入输出格式：  输入：  第一行一个数n（n<=5000)，代表堆的大小；第二行n个数，代表堆的各个元素；第三行一个数m (m<=1000)，代表接下来共m个操作。接下来m行，分别代表各个操作。下面是各个操作的格式：  插入操作：1 num；  删除操作：2；  排序操作：第一行两个数3和n，3代表是排序操作，n代表待排序的数的数目，接下来一行n个数是待排序数。  保证排序操作只出现一次且一定是最后一个操作。  输出：  第一行建堆操作输出建好堆后的堆顶的元素。接下来m个操作，若是插入和删除操作。每行输出执行操作后堆顶的元素的值；若是排序操作，输出一行按升序排序好的结果，每个元素间用空格分隔。  ②题目描述：哈夫曼编码。  输入输出格式：  输入：  一串小写字母组成的字符串（不超过1000000)。  输出：  输出这个字符串通过Huffman编码后的长度。   1. 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） 2. A题 3. 数据结构：选择堆。堆其实就是优先队列，大根树（小根树）是每个节点的值都大于（小于）或等于其子节点（如果有的话）值的树。大根树或 小根树节点的子节点个数可以大于 2。 如果既是大根树（小根树）又是完全二叉树，那么我们称为大根堆（小根堆）。      1. 算法： 2. 插入：类似与我们实现中的二分法，我们每一步都进行这个元素和heap中的元素的大小比较，如果是小于，那么我们就继续向下找，同时注意这里需要防止短路问题，要防止数组下标越界。      1. 排序：使用堆排序，首先进行数组的初始化。进行一个时间复杂度O（n）的初始化，初始化的步骤是，将每一个子树都调整成为一个堆，这样我们就需要一个down的操作，实现如下：     我们需要每一步都进行一个取出堆顶的操作，然后保存到数组中，实现如下：     1. 删除操作：首先我们需要明确如何进行删除操作，我们将数组最后的一个元素放置到堆顶，然后调整这个堆的位置。同时我们知道，需要特判堆元素的个数，如果堆此时只有一个元素，那么我们就不能用最后一个元素进行替换，此时会RE，实现代码如下：     ②霍夫曼编码：  数据结构：霍夫曼编码使用扩充二叉树（外部节点对应于字符串中被编码的字符）进行编码。对于每一个次的合并，我们都需要找出来这个数组中的最小元素，故使用堆。我们只需要重载一下比较运算符就可以了。    算法：每一次都进行pop操作合并，然后对于合并出来的结点的元素值都是两个弹出来结点的权重之和。合并之后push进去。等只有一棵树的时候，弹出来。此时再进行一次遍历操作即可。Main函数中实现代码如下：     1. 测试结果（测试输入，测试输出） 2. a题   输入：    输出：    提交OJ的结果：    (2)b题  输入：    输出：    提交OJ最后的结果：     1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径） 2. 堆排序为什么要把 a 和 heap 给分开?也就是解耦合？   我们知道大根堆退出的时候会调用析构函数，将堆中数组 heap 删除，执行 heap.initialize(a,n)时，把数组 heap 初始化为数组 a。 为避免调用堆的析构函数时将数组 a 删除，在 maxHeap 类中增加 deactiveArray()函数   1. 对于指针的使用，一定要谨慎！比如以下这个错误，查了好几个小时。我在输入测试用例的时候，发现输出不对，在经过了二分等debug技巧后确定了bug所在的位置。     我发现这行代码在return array[1]之后，它结点的值自动的改变了！按理来说它是绝对不应该改变的，后来就逐行查找，最终发现      在insert的时候，对于重载的赋值操作，他是直接进行了浅层赋值。就是说，犯错的array[3]中的左孩子指向的是top1,而top1是一个局部变量！它在本轮循环结束后会自动析构，而array[3]中的左孩子指针的值却不会更新，在下一次执行的时候，对于top1的新的值，地址却没有变，array[3]依然指向了它，这就导致了错误。  解决：使用深层copy,或者使用动态分配     1. 在提交oj的时候要把自己的测试删除，比如多输出了一个换行可能就会导致全部的判错。 2. 对于BOOL数组的初始化，绝对不能想当然，每一次的定义都要对其进行初始化。在本次的实验中，因为bool数组没有初始化导致debug了很久。      1. 对于下标从1开始的数组，要多动态分配一块内存。因为索引为0的地方我们是没有访问的。      1. 对于一个只有两个私有成员的struct，我们可以直接使用pair来存储。简化了我们的代码量。同时对于返回多个值的函数，我们只能使用引用来返回。这也是为什么兴起了Go语言的一个原因。 2. 对于height等操作的计算，一定要特判是否合法。因为对于root结点来说，它的父节点是自身，不能直接增加。      1. 我们在写循环条件判断的时候，对于短路情况的判断一定要慎重，我们对于指针的使用一定要先判断是否为空，在进行取值操作，如下：      1. 对于维护私有变量的时候，要特殊情况特殊判：比如说删除结点的时候，要考虑这个是不是头结点，如果是，那么更新私有变量。如下：     (10)要注意私有成员的更新，public函数知道自己所处的状态都是靠着私有成员才知道的，如果没有及时更新，那数据就成了垃圾数据，没有任何意义。我在写实验的时候，也经历过没更新导致的Bug，最终debug查出来，就是下面这个：     1. 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） 2. **A题** 3. #include<iostream> 4. using namespace std; 5. template<class T> 6. class minHeap //小根堆类 7. { 8. public: 9. minHeap() :heapSize(0), arrayLength(10) { heap = new T[10]; } *//构造函数* 10. minHeap(T\* theData, int theSize):heap(NULL){ initialize(theData, theSize); } *//构造并初始化* 11. ~minHeap() { delete[] heap; } *//析构函数* 12. int size() { return heapSize; } *//堆元素个数* 13. bool empty() const {return heapSize == 0;} *//是否为空* 14. void initialize(T\* theHeap, int theSize); *//初始化* 15. void push(const T& theElement); *//插入元素* 16. void pop(); *//弹出元素* 17. T& top() { return heap[1]; }*//首元素* 18. void deactivateArray() { heap = NULL; } 19. private: 20. T\* heap;*//元素数组，一个类型为T的一维数组* 21. int arrayLength;*//数组的容量* 22. int heapSize;*//堆的元素个数* 24. }; 25. template<class T> 26. void minHeap<T>::initialize(T\* theHeap, int theSize) *//初始化一个非空小根堆* 27. {*//在数组theHeap[1:theSize]中建小根堆* 28. delete[] heap; 29. heap = theHeap; 30. heapSize = theSize; 31. arrayLength = theSize + 1; 33. *//堆化* 34. for (int root = heapSize / 2; root >= 1; root--) 35. { 36. *//为元素rootElement寻找位置* 37. T rootElement = heap[root];*//子树的根* 38. int child = root \* 2;*//child的父节点是rootElement的位置* 39. while (child <= heapSize) 40. { 41. *//heap[child]应是兄弟中较小者* 42. if (child < heapSize && heap[child] > heap[child + 1]) 43. child++; 44. *//能把rootElement放入heap[child/2]吗* 45. if (rootElement <= heap[child])*//小于两个儿子* 46. break;*//可以* 47. *//不可以* 48. heap[child/2] = heap[child];*//把孩子上移* 49. child \*= 2;*//下移一层* 50. } 51. heap[child / 2] = rootElement; 52. } 53. } 54. template<class T> 55. void minHeap<T>::push(const T& theElement) 56. { 57. *//必要时增加数组长度* 58. if (heapSize == arrayLength-1) *//没有足够空间* 59. {*//数组长度加倍，以下为changeLength1D 内容* 60. arrayLength = 2 \* heapSize + 1;*//数组长度倍增* 61. T\* temp = new T[arrayLength]; 62. copy(heap+1, heap+heapSize+1, temp+1); 63. delete [] heap; 64. heap = temp; 65. } 66. heapSize++;*//heapSize+1 （新叶子）* 67. *//为元素theElement寻找插入位置* 68. int currentNode = heapSize;*//currentNode从新叶子向上移动* 69. while (currentNode > 1 && heap[currentNode/ 2] > theElement) 70. {*//当插入元素小于父节点，不能把theElement插在heap[currentNode]* 71. heap[currentNode] = heap[currentNode / 2]; *//元素向下移动* 72. currentNode /= 2; *//移向父节点* 73. } 74. heap[currentNode] = theElement;*//插入* 75. } 76. template<class T> 77. void minHeap<T>::pop() 78. { 79. *//删除最小元素* 80. heap[1].~T(); 81. *//删除最后一个元素，然后重新建堆* 82. T lastElement = heap[heapSize--]; 83. *//从根开始，为最后一个元素寻找位置* 84. int currentNode = 1;*//最后一个元素当前在根* 85. int child = 2;*//尾节点的孩子节点* 86. while (child <= heapSize) 87. { 88. *//heap[child]应该是currentNode的更小孩子* 89. if (heap[child + 1] < heap[child]) 90. child++; 91. *//判断可以把theElement放在heap[currentNode]吗* 92. if (lastElement <= heap[child])*//lastElement小于儿子* 93. break;*//可以，插入* 94. *//不可以* 95. heap[currentNode] = heap[child]; *//孩子上移* 96. currentNode = child;  *//向下一层寻找位置* 97. child \*= 2; 98. } 99. heap[currentNode] = lastElement; 100. } 101. template<class T> 102. void heapSort(T a[], int n) 103. {*//利用堆排序方法给数组a[1:n]排序* 104. *//在数组上建立小根堆* 105. minHeap<T> heap(a, n); 107. *//逐个从小根堆中提取元素* 108. for (int i = n-1; i >= 1; i--) 109. { 110. T x = heap.top(); 111. heap.pop(); 112. a[i + 1] = x; 113. } 114. *//从堆的析构函数中保存数组a* 115. heap.deactivateArray();*//把a和heap给分开！原来他俩绑定的（解绑）* 116. } 117. int main() 118. { 119. minHeap<int> heap; 120. int n;*//n代表堆的大小。第二行n个数，代表堆的各个元素* 121. int m;*//m为行数* 122. int \* a; 123. cin >> n; 124. a = new int[n+1]; 125. for (int i = 1; i <= n; i++) 126. cin >> a[i];*//输入n个数* 127. heap.initialize(a, n); 128. cout << heap.top() << endl;*//建堆操作后输出栈顶元素* 129. cin >> m;*//行数* 130. for(int j= 1; j<=m; j++) 131. { 132. int opt; 133. int num; 134. cin >> opt; 135. switch(opt) 136. { 137. case 1: 138. cin >> num; 139. heap.push(num); 140. cout << heap.top()<<endl; 141. break; 142. case 2: 143. heap.pop(); 144. cout << heap.top()<<endl; 145. break; 146. case 3: 147. cin >> num; 148. int \*  a=new int[num+1]; 149. for (int i = 1; i <= num; i++) 150. cin >> a[i]; 151. heapSort(a, num); 152. for (int i = n; i >= 1; i--) 153. cout << a[i]<<" "; 154. } 156. } 157. }   **(2)B题**   1. #include<iostream> 2. using namespace std; 3. template<class T> 4. class minHeap 5. { 6. public: 8. minHeap() :heapSize(0), arrayLength(10) { heap = new T[10]; } *//构造函数* 9. minHeap(T\* theData, int theSize):heap(NULL){ initialize(theData, theSize); } *//构造并初始化* 10. minHeap(int theSize)*//构造已知元素个数的堆* 11. { 12. arrayLength = theSize + 1; 13. heap = new T[arrayLength]; 14. heapSize = 0; 15. } 16. ~minHeap() { delete[] heap; } *//析构函数* 17. int size() { return heapSize; } *//堆元素个数* 18. bool empty() const {return heapSize == 0;} *//是否为空* 19. void initialize(T\* theHeap, int theSize); *//初始化* 20. void push(const T& theElement); *//插入元素* 21. void pop(); *//弹出元素* 22. T& top() { return heap[1]; }*//首元素* 23. void deactivateArray() { heap = NULL; } 24. private: 25. T\* heap;*//元素数组，一个类型为T的一维数组* 26. int arrayLength;*//数组的容量* 27. int heapSize;*//堆的元素个数* 28. }; 29. template<class T> 30. void minHeap<T>::initialize(T\* theHeap, int theSize) *//初始化一个非空小根堆* 31. {*//在数组theHeap[1:theSize]中建小根堆* 32. delete[] heap; 33. heap = theHeap; 34. heapSize = theSize; 35. arrayLength = theSize + 1; 37. *//堆化* 38. for (int root = heapSize / 2; root >= 1; root--) 39. { 40. *//为元素rootElement寻找位置* 41. T rootElement = heap[root];*//子树的根* 42. int child = root \* 2;*//child的父节点是rootElement的位置* 43. while (child <= heapSize) 44. { 45. *//heap[child]应是兄弟中较小者* 46. if (child < heapSize && heap[child] > heap[child + 1]) 47. child++; 48. *//能把rootElement放入heap[child/2]吗* 49. if (rootElement <= heap[child])*//小于两个儿子* 50. break;*//可以* 51. *//不可以* 52. heap[child/2] = heap[child];*//把孩子上移* 53. child \*= 2;*//下移一层* 54. } 55. heap[child / 2] = rootElement; 56. } 57. } 58. template<class T> 59. void minHeap<T>::push(const T& theElement) 60. { 61. *//必要时增加数组长度* 62. if (heapSize == arrayLength-1) *//没有足够空间* 63. {*//数组长度加倍，以下为changeLength1D 内容* 64. arrayLength = 2 \* heapSize + 1;*//数组长度倍增* 65. T\* temp = new T[arrayLength]; 66. copy(heap+1, heap+heapSize+1, temp+1); 67. delete [] heap; 68. heap = temp; 69. } 70. heapSize++;*//heapSize+1 （新叶子）* 71. *//为元素theElement寻找插入位置* 72. int currentNode = heapSize;*//currentNode从新叶子向上移动* 73. while (currentNode > 1 && heap[currentNode/ 2] > theElement) 74. {*//当插入元素大于父节点，不能把theElement插在heap[currentNode]* 75. heap[currentNode] = heap[currentNode / 2]; *//元素行下移动* 76. currentNode /= 2; *//移向父节点* 77. } 78. heap[currentNode] = theElement;*//插入* 79. } 80. template<class T> 81. void minHeap<T>::pop() 82. { 83. *//删除最小元素* 84. heap[1].~T(); 85. *//删除最后一个元素，然后重新建堆* 86. T lastElement = heap[heapSize--]; 87. *//从根开始，为最后一个元素寻找位置* 88. int currentNode = 1;*//最后一个元素当前在根* 89. int child = 2;*//尾节点的孩子节点* 90. while (child <= heapSize) 91. { 92. *//heap[child]应该是currentNode的更小孩子* 93. if (heap[child + 1] < heap[child]) 94. child++; 95. *//判断可以把theElement放在heap[currentNode]吗* 96. if (lastElement <= heap[child])*//lastElement小于儿子* 97. break;*//可以，插入* 98. *//不可以* 99. heap[currentNode] = heap[child]; *//孩子上移* 100. currentNode = child;  *//向下一层寻找位置* 101. child \*= 2; 102. } 103. heap[currentNode] = lastElement; 104. } 105. int main(){ 106. string theString;*//要输入的字符串* 107. cin >> theString; 108. int occurrenceOfLetters[26] = {0}; 109. int stringLength = theString.length();*//获取字符串长度* 110. minHeap<int> huffmanTreeElements(stringLength); *//建立大小为字符串长度的小根堆* 111. for(int i = 0; i < stringLength; i ++) *//统计26个字幕出现的次数，并存入数组occurrenceOfLetters* 112. { 113. int j = theString[i] - 'a'; 114. occurrenceOfLetters[j] ++; 115. } 116. for(int i = 0; i < 26; i ++) *//如果字母出现次数不是0，把它的出现次数放入小根堆* 117. { 118. if(occurrenceOfLetters[i] != 0) 119. { 120. huffmanTreeElements.push(occurrenceOfLetters[i]); 121. } 122. } 123. *//“建树”并进行计算* 124. int answer = 0; *//要输出的答案* 125. while(huffmanTreeElements.size() != 1) *//除非只剩一个，否则还处于“建树”过程中* 126. { 127. *//每次弹出两个最小的合并* 128. int leftChild = huffmanTreeElements.top(); 129. huffmanTreeElements.pop(); 130. int rightChild = huffmanTreeElements.top(); 131. huffmanTreeElements.pop(); 132. answer += leftChild; 133. answer += rightChild; 134. *//为何直接加：每合并了一次就相当于增加了从外部节点到跟的一条线，也就增加了一次字符串长度* 135. *//生成的新节点再去合并* 136. huffmanTreeElements.push(leftChild + rightChild); 137. } 138. cout << answer << endl;*//输出答案* 140. return 0; 141. } | | | |
|  | | | |