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：201900161140 | 姓名： 张文浩 | | 班级： 19级人工智能 |
| 实验题目：堆及其应用 | | | |
| 实验学时：2 | | 实验日期： 10.4 | |
| 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。 | | | |
| 软件开发工具：  Vscode | | | |
| 1. 实验内容   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。   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）   根据题目要求，定义一个小根堆的类minHeap，  其成员变量包括heap：一个类型为T的一维数组、arrayLength数组heap的容量、heapSIze堆的元素个数。  成员函数包括：  top()获取堆的根节点的数值、push(x)，把元素x插入到堆中合适的位置、pop()弹出堆的根节点的元素，并对堆进行重构操作  **Top()：**  因为堆实际上是在一维数组heap中存储的，所以heap[1]就是堆顶，直接返回heap[1]即可  **Push(x)：**  首先要判断当前数组容量够不够，如果不够了就要新开一个更大数组，并把原来数组的值复制到新的数组中，然后从最后的叶子结点开始寻找x插入的位置，currentNode从新建的叶子结点开始不断地向上移动，如果当前没找到合适的位置，就把双亲向下移动，给新来的元素腾个地方，currentNode也向上移动，直到找到合适的插入位置。  **Pop()：**  首先删除堆顶元素heap[1]，析构，然后去除最后一个元素，重新建堆，从根开始，为最后一个元素寻找位置，遍历每一层的时候，先找到currentNode的孩子中的最小的值，然后判断lastElement是否比这个最小的孩子小，如果是，则可以当孩子的根节点，如果不是，就要将child孩子向上移动，currentNode继续向下去寻找位置。   1. 测试结果（测试输入，测试输出）   **1.**  C:\WINDOWS\TEMP\WeChat Files\198b531e2a3194f8c3126782b42befc.jpg   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   通过本次实验，我掌握了堆结构的定义、描述方法、操作定义及实现。在第二题中，我学习了关系哈弗曼编码的相关知识，掌握了堆结构的应用。  在构造堆的时候，应当注意排序的操作是按照层数进行的。每一次删除或者是初始化二叉树的时候，应该对其进行排序，将所有的元素中最小的那个放置到堆顶。这样才可以保证排序后的输出能够按照输出堆顶元素、弹出堆顶元素的顺序循环进行完成。如果不确定哪些时候需要排序，可以在每次更改堆的结构时更新顺序，也就是一直维护这个最小堆的结构，从而保证无论何时都可以得到由小到大的元素排列。  在第二个题目中，要求计算出霍夫曼编码的长度，也就是各个叶节点权值与路径长度乘积之和，同时这个值等于非叶节点之和。根据这个特性，无需完整地建立霍夫曼树，只需要用数组存储每个字符出现的频次，同时将其按照频次由小到大排序，取出最小的两个元素便可获得所有非叶节点包含的权重。这个思路在解决该问题时尤为重要，它让我避免了许多无用的步骤。   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）   #include <iostream>  using namespace std;  template <class T>  class minHeap  {  public:      minHeap(int initialCapacity = 10);      ~minHeap() { delete[] heap; }      const T &top() { return heap[1]; }      void push(const T &);      void pop();      void initialize(T \*theHeap, int theSize);  private:      T \*heap;         //一个类型为T的一维数组      int arrayLength; //数组heap的容量      int heapSize;    //堆的元素个数  };  template <class T>  minHeap<T>::minHeap(int initialCapacity)  {      arrayLength = initialCapacity + 1;      heap = new T[arrayLength];      heapSize = 0;  }  template <class T>  void minHeap<T>::push(const T &theElement) //把元素theElement加入堆  {      //必要时增加数组长度      if (heapSize == arrayLength - 1) //数组长度加倍      {          T \*temp = new T[2 \* arrayLength];          copy(heap, heap + arrayLength, temp);          delete[] heap;          heap = temp;          arrayLength \*= 2;      }      //为元素theElement寻找插入位置      //currentNode从叶子节点向上移动      int currentNode = ++heapSize;      while (currentNode != 1 && heap[currentNode / 2] > theElement)      {          //不能把元素theElement插入在heap[currentNode]          heap[currentNode] = heap[currentNode / 2]; //把元素向下移动          currentNode /= 2;                          //currentNode移向双亲      }      heap[currentNode] = theElement;  }  template <class T>  void minHeap<T>::pop() //删除最大元素  {      //删除最大元素      heap[1].~T();      //删除最后一个元素，然后重新建堆      T lastElement = heap[heapSize--];      //从根开始，为最后一个元素寻找位置      int currentNode = 1, child = 2;      while (child <= heapSize)      {          //heap[child]应该是currentNode的更大的孩子          if (child < heapSize && heap[child] > heap[child + 1])              child++;          //可以把lastElement放在heap[currentNode]吗？          if (lastElement <= heap[child])              break; //可以          //不可以          heap[currentNode] = heap[child]; //把孩子child向上移动          currentNode = child;             //向下移动一层寻找位置          child \*= 2;      }      heap[currentNode] = lastElement;  }  template <class T>  void minHeap<T>::initialize(T \*theHeap, int theSize) //在数组theHeap[1:theSize]中建小根堆  {      delete[] heap;      heap = theHeap;      heapSize = theSize;      arrayLength = theSize + 1;      //堆化      for (int root = heapSize / 2; root >= 1; root--)      {          T rootElement = heap[root];          //为元素rootElement寻找位置          int child = 2 \* root; //孩子child的双亲是元素rootElement的位置          while (child <= heapSize)          {              //heap[child]应该是兄弟中的较小者              if (child < heapSize && heap[child] > heap[child + 1])                  child++;              //可以把元素rootElement放在heap[child/2]吗              if (rootElement <= heap[child])                  break; //可以              //不可以              heap[child / 2] = heap[child]; //把孩子向上移              child \*= 2;                    //移到下一层          }          heap[child / 2] = rootElement;      }  }  int main()  {      int n, m, t;      cin >> n;      minHeap<int> minheap(n);      for (int i = 1; i <= n; i++)      {          cin >> t;          minheap.push(t);      }      cout << minheap.top() << endl;      cin >> m;      while (m--)      {          int op, num;          cin >> op;          if (op == 1) //插入操作          {              cin >> num;              minheap.push(num);              cout << minheap.top() << endl;          }          else if (op == 2) //删除操作          {              minheap.pop();              cout << minheap.top() << endl;          }          else if (op == 3) //排序操作          {              cin >> n;              minHeap<int> minheap\_t(n);              for (int i = 1; i <= n; i++)              {                  cin >> t;                  minheap\_t.push(t);              }              while (n--)              {                  cout << minheap\_t.top() << " ";                  minheap\_t.pop();              }              cout << endl;          }      }      //system("pause");      return 0;  } | | | |