数据结构 第四次作业报告

2022013014 黄泽文 未央-软件21

1. 任务调度

分析

本题需要实现一个优先队列，本质上就是一个小根堆。堆的两个基本操作是向上更新与向下更新。插入一个元素，将其放在堆底部（即数组末尾），向上更新即可；弹出顶部元素，将数组的第一个元素删去（为了去除空位，将最后一个元素移动到第一个位置覆盖），向下更新即可。

本题需要实现两个关键字的排序，定义一个结构体及比较函数即可。

题目要求优先级以 2^{32} 为上界，为了避免溢出导致判断错误，可以在乘 2 前直接与 2^{31} 比大小。

代码

优先队列：

```
template<typename T>
class priority_queue {
private:
      T heap[MaxSize];
      size_t size = 0;

      void heapifyUp() {
            size_t currentIndex = size - 1;
            while (currentIndex > 0) {
                  size_t parentIndex = (currentIndex - 1) / 2;
                  if (heap[currentIndex] < heap[parentIndex]) {
                        std::swap(heap[currentIndex], heap[parentIndex]);
                        currentIndex = parentIndex;
                  } else {
                        break;
                  }
            }
      }

      void heapifyDown() {
            size_t currentIndex = 0;
            while (true) {
                  size_t leftChild = 2 * currentIndex + 1;
                  size_t rightChild = 2 * currentIndex + 2;
                  size_t smallest = currentIndex;
                  if (leftChild < size && heap[leftChild] < heap[smallest]) {
                        smallest = leftChild;
                  }
                  if (rightChild < size && heap[rightChild] < heap[smallest]) {
                        smallest = rightChild;
                  }
                  if (smallest != currentIndex) {
                        std::swap(heap[currentIndex], heap[smallest]);
                        currentIndex = smallest;
```

```
                } else {
                        break;
                }
            }
        }

public:
        void push(const T &value) {
            if (size >= MaxSize) {
                throw;
            }
            heap[size] = value;
            size++;
            heapifyUp();
        }

        void pop() {
            if (size == 0) {
                throw;
            }
            std::swap(heap[0], heap[size - 1]);
            size--;
            heapifyDown();
        }

        T top() const {
            if (size == 0) {
                throw;
            }
            return heap[0];
        }

        bool empty() const {return size == 0;}
        size_t getSize() const {return size;}
};
```

主体代码:

```
struct process {
        unsigned long priority;
        char name[9];

        bool operator<(const process &x) const {
            if (priority == x.priority) {
                for (int i = 0; i <= 8; i++) {
                    if (name[i] < x.name[i])return true;
                    if (name[i] > x.name[i])return false;
                }
                return false;
            } else {
                return priority < x.priority;
            }
        }
};

priority_queue<process> pq;
```

```
int main() {
      int n, m;
      cin >> n >> m;
      for (int i = 0; i < n; i++) {
            process tmp;
            scanf("%lu %s", &tmp.priority, tmp.name);
            pq.push(tmp);
      }
      while (m--) {
            if (pq.empty())break;
            process t = pq.top();
            pq.pop();
            printf("%s\n", t.name);
            if (t.priority < ((unsigned long) 1 << 31)) {
                  t.priority *= 2;
                  pq.push(t);
            }
      }
}
```

2. 重名剔除

分析

计算哈希。本题可以直接将单词的每个字母 ASCII 码减去 'a' 后求和，结果作为哈希。这样的哈希容易重复，一种解决方法是将一个哈希对应的不同单词维护成一个链表。这样一来恰好满足了题目的需要，第一次读入单词时将其插入链表，第二次读入时在链表内发现重复并输出。

代码

```
struct hashnode {
      char val[41] = "";
      hashnode *next = nullptr;
      bool vis = false;
};

int main() {
      int n;
      cin >> n;
      hashnode hashhead[1000];
      for (int i = 0; i < n; i++) {
            char val[41];
            cin >> val;
            int hash = 0;
            for (int j = 0; val[j]; j++) {
                  hash += val[j] - 'a';
            }
            hashnode *h;
            bool find = false;
            for (h = &hashhead[hash]; h; h = h->next) {
                  if (strcmp(h->val, val) == 0) {
                        find = true;
                        if (!h->vis) {
                              cout << val << endl;
```

```
                                h->vis = true;
                            }
                             break;
                        }
                         if (h->next == nullptr)break;
                    }
                     if (find) continue;
                     h->next = new hashnode;
                     for (int j = 0; val[j]; j++) {
                           h->next->val[j] = val[j];
                    }
               }
          }
```

3. 玩具

分析

状态是一个排列，为了便于储存为节点，需要将其转换为一个数字。可以利用

$$Index = \sum_{i=1}^{8} R_i(n-i)!$$

生成。其中 R_i 是第 i 个数字与后面的数字组成的逆序对个数。

本题的规模不算很大，可以直接预处理计算出所有状态的结果。

第一步是搜索建图。从初始状态开始向外搜索，通过枚举三种操作方式更新状态并建边。需要注意第二、三种操作不是可逆的，注意建边的方向要与更新状态的方向相反（因为题目所需要的是从给定状态到初始状态的最短距离，而下面计算的是从初始状态到给定状态的最短路）。为了避免递归层数过多导致段错误，这里使用了广度优先搜索。

第二步是搜寻最短路。直接使用 Dijkstra 算法计算初始状态到各个状态的最短距离即可。

代码

状态编解码：

```
typedef struct {int q[8];} state;
const int factorial[8] = {1, 1, 2, 6, 24, 120, 720, 5040};

inline int encode(int q[8]) {
      int ans = 0;
      for (int i = 0; i < 8; i++) {
            int cnt = 0;
            for (int j = i + 1; j < 8; j++) {
                  if (q[j] < q[i])cnt++;
            }
            ans += cnt * factorial[7 - i];
      }
      return ans;
}

inline state decode(int id) {
      state ans;
      int list[8] = {1, 2, 3, 4, 5, 6, 7, 8};
      for (int i = 0; i < 8; i++) {
```

```
            int cnt = id / factorial[7 - i];
            id %= factorial[7 - i];
            for (int j = 0; j < 8; j++) {
                  if (list[j] != 0) {
                        if (cnt == 0) {
                              ans.q[i] = list[j];
                              list[j] = 0;
                              break;
                        } else {
                              cnt--;
                        }
                  }
            }
      }
       return ans;
}
```

搜索建图：

```
int searchtable[50000], searchcnt = 0;
void bfs() {
      searchtable[searchcnt++] = encode(ini);
      while(searchcnt) {
            int p = searchtable[--searchcnt];
            if (vis[p])continue;
            vis[p] = true;
            state ps = decode(p);
            int q[8];
            // 1. upside down
            for (int i = 0; i < 8; i++) {
                  q[i] = ps.q[7 - i];
            }
            int code1 = encode(q);
            addEdge(p, code1);
            if(!vis[code1])searchtable[searchcnt++] = code1;
            // 2. right shift
            q[0] = ps.q[1]; q[1] = ps.q[2]; q[2] = ps.q[3]; q[3] = ps.q[0];
            q[4] = ps.q[7]; q[5] = ps.q[4]; q[6] = ps.q[5]; q[7] = ps.q[6];
            int code2 = encode(q);
            addEdge(p, code2);
            if(!vis[code2]) searchtable[searchcnt++] = code2;
            // 3. central rotate
            q[0] = ps.q[0]; q[1] = ps.q[2]; q[2] = ps.q[5]; q[3] = ps.q[3];
            q[4] = ps.q[4]; q[5] = ps.q[6]; q[6] = ps.q[1]; q[7] = ps.q[7];
            int code3 = encode(q);
            addEdge(p, code3);
            if(!vis[code3]) searchtable[searchcnt++] = code3;
      }
}
```

最短路计算与查询：

```
      priority_queue<node> q;
      for (int i = 0; i < MAXN; i++) {
            vis[i] = false;
            dis[i] = 1e9;
```

```
    }
    dis[encode(ini)] = 0;
    // dijkstra calculate distance from ini to all other states
    q.push({encode(ini), 0});
    while (!q.empty()) {
        node p = q.top();
        q.pop();
        if (vis[p.id])continue;
        vis[p.id] = true;
        for (int i = head[p.id]; i != -1; i = map[i].next) {
            int v = map[i].v;
            if (dis[v] > dis[p.id] + 1) {
                dis[v] = dis[p.id] + 1;
                q.push({v, dis[v]});
            }
        }
    }

    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int qq[8],rr[8];
        for (int j = 0; j < 8; j++) {
            scanf("%d", &qq[j]);
        }
        int id = encode(qq);
        if (dis[id] == 1e9)printf("-1\n");
        else printf("%d\n", dis[id]);
    }
```