# 数据结构 第一次作业报告

2022013014 黄泽文 未央-软件21

## 1. 面试

### 分析

为了存储环状的数据，需要对链表的实现代码进行修改，有两种实现方法：在插入第一个结点时，将其next指向自身，后续插入结点则会自动形成一个环；也可以在遍历结点时进行特殊判断，当next指向的不是一个结点时，直接回到head结点。

针对“走过m人”的要求，对add函数进行修改。维护一个指针last存储上次插入的结点的位置，每次从last出发，向下走m-1个结点即可。

需要注意的是，输出结果的方向与链表的方向是相反的，需要暂存结果并逆序输出。

### 代码

本题代码只需要在链表基础上对add与print进行修改，两个函数的代码如下。

```
// ...
class LinkedList
{
private:
      Node *head;
      Node *last;
// ...
      void addNode(int val, int m)
      {
            Node *newNode = new Node;
            newNode->data = val;

            if (head == NULL)
            {
                  newNode->next = newNode;
                  head = newNode;
                  last = newNode;
            }
            else
            {
                  Node *temp = last;
                  for (int i = 1; i < m; i++)
                  {
                        temp = temp->next;
                  }
                  newNode->next = temp->next;
                  temp->next = newNode;
                  last = newNode;
            }
      }
      void printList(int n)
      {
            Node *temp = last;
```

```
            int ans[n], cnt = 0;
            ans[cnt++] = temp->data;
            temp = temp->next;
            while (temp != last)
            {
                  ans[cnt++] = temp->data;
                  temp = temp->next;
            }
            cout << ans[0] << " ";
            for (int i = cnt - 1; i; i--)
            {
                  cout << ans[i] << " ";
            }
            cout << endl;
      }
};
// ...
```

## 2. 火车调度

### 分析

本题的中转盲端S是一个标准的栈。对于出口序列的每个数字：

1. 如果它在入口序列中，则将在其之前的所有数字推入栈中，取出这个数字。这也是唯一取出的方法。如果在将其之前的数字推入栈的过程中，栈达到了上限，则本题无解。
2. 如果它在栈顶，可以直接将其推出；如果它在栈中但不在栈顶，则本题无解。

模拟这个过程即可。

### 优化

1. 为了避免空间溢出，本题不必直接将栈的空间开到capacity的大小，根据需要动态调整即可。将变量声明为全局变量，避免re。
2. 避免使用标准输入输出，会导致tle。
3. 在将一长串数字推入栈前，可以直接判断这个操作是否会操作capacity，避免tle。

### 代码

本题核心部分的代码如下。

```
// ...
class Stack
{
public:
 // ...
      void push(int value)
      {
            if (size_ == capacity_)
            {
                  int *newData = new int[capacity_ * 2];
                  for (int i = 0; i < capacity_; i++)
                        newData[i] = data_[i];
                  delete[] data_;
                  data_ = newData;
                  capacity_ *= 2;
            }
            data_[size_++] = value;
```

```
        }
    // ...
    };

    Stack stack;
    int ope[1600000 * 2 + 1], cnt = 0, oriNow = 1;
    int main()
    {
          int n, cap;
          cin >> n >> cap;
          for (int i = 0; i < n; i++)
          {
                int v;
                scanf("%d", &v);
                if (!stack.empty() && stack.top() == v)
                {
                      stack.pop();
                      ope[cnt++] = 1;
                }
                else
                {
                      if (oriNow > v)
                      {
                            printf("No\n");
                            return 0;
                      }
                      while (stack.size() < cap && oriNow <= v)
                      {
                            stack.push(oriNow++);
                            ope[cnt++] = 0;
                      }
                      if (stack.top() != v)
                      {
                            printf("No\n");
                            return 0;
                      }
                      stack.pop();
                      ope[cnt++] = 1;
                }
          }
    // ...
    }
```

## 3. 灯塔

### 分析

将所有点按横坐标升序排列，考察其纵坐标。满足$i < j, y_i < y_j$的点，是可以互相看到的。对所有纵坐标取相反数，则逆序对的对数等于能看到的灯塔对数。

可以利用归并排序的过程统计逆序对——具体地说，在合并两个小块的过程中，每在右半块取出一个数字，其与左半块还没放入的数字就形成了mid-i+1组逆序对。通过递归过程统计每次合并产生的所有逆序对即可。

## 代码

以下是归并排序部分的代码，其中count即为逆序对数。可以通过重载运算符使其同时适用于横纵坐标。

```
long long merge(int left, int mid, int right)
{
      point temp[right - left + 1];
      int i = left, j = mid + 1, k = 0;
      long long count = 0;
      while (i <= mid && j <= right)
      {
            if (arr[i] <= arr[j])
            {
                  temp[k++] = arr[i++];
            }
            else
            {
                  temp[k++] = arr[j++];
                  count += mid - i + 1;
            }
      }
      while (i <= mid)
      {
            temp[k++] = arr[i++];
      }
      while (j <= right)
      {
            temp[k++] = arr[j++];
      }
      for (int p = 0; p < k; p++)
      {
            arr[left + p] = temp[p];
      }
      return count;
}

long long mergeSort(int left, int right)
{
      if (left >= right)
      {
            return 0;
      }
      int mid = (left + right) / 2;
      long long count = 0;
      count += mergeSort(left, mid);
      count += mergeSort(mid + 1, right);
      count += merge(left, mid, right);
      return count;
}
```