问题 1

问题描述

- 给出栈的实现策略，使得入栈、出栈与求最小值都能在 $\mathcal{O}(1)$ 内完成，并进行相应的算法分析。

算法思路

- 由于栈是仅在表尾进行插入和删除操作的线性表，考虑除存储元素的数据栈 $data$ 外，另外增加一个辅助栈 aux，辅助栈中存储在数据栈入栈过程中出现的所有可能的最小值。
- 具体来说：
  - 当数据栈入栈元素为 val 时，若辅助栈此时为空或者 val 小于等于辅助栈栈顶元素，则将 val 一并压入辅助栈。
  - 当数据栈出栈元素为 val 时，若辅助栈栈顶元素恰为 val，则将 val 从辅助栈中一并弹出。
  - 当需要询问数据栈中最小值时，直接输出辅助栈栈顶元素的值即可。

算法设计与代码

- 该代码基于以下输入输出格式：
  - 第一行一个正整数 $qcnt$ 表示操作个数。
  - 接下来 $qcnt$ 行每行表示一个操作，三种操作的格式如下：
    - `1 x`，x 为整数，表示入栈元素为 x。
    - `2` 表示出栈，并输出出栈元素，栈为空时输出 `ERROR`。
    - `3` 表示询问栈中元素最小值，栈为空时输出 `ERROR`。

```cpp
#include <bits/stdc++.h>

using std::min;
using std::stack;

inline void Push(int val, stack<int> &data, stack<int> &aux)
{
    if (aux.empty() || val <= aux.top())
        aux.push(val);
    data.push(val);
    return ;
}

inline void Pop(stack<int> &data, stack<int> &aux)
{
    if (data.empty())
    {
        puts("ERROR");
        return ;
    }
    int val = data.top();
    if (val == aux.top())
        aux.pop();
    data.pop();
```

```
    printf("%d\n", val);
    return ;
}

inline void queryMin(stack<int> &aux)
{
    if (aux.empty())
    {
        puts("ERROR");
        return ;
    }
    printf("%d\n", aux.top());
    return ;
}

int main()
{
    stack<int> data, aux;
    int qcnt, opt, val;

    scanf("%d", &qcnt);
    while (qcnt--)
    {
        scanf("%d", &opt);
        if (opt == 1)
        {
            scanf("%d", &val);
            Push(val, data, aux);
        }
        else if (opt == 2)
            Pop(data, aux);
        else
            queryMin(aux);
    }
    return 0;
}
```

算法演示

- 依次进行以下输入，输出结果已做加粗标识，以便与输入区分。

```
14
2
ERROR
3
ERROR
1 3
1 2
1 2
1 5
1 6
3
2
2
6
```

```
2
5
2
2
3
2
2
2
3
3
```

- 可见程序正确输出了结果。

总结与讨论

- 最坏情况下，该算法每次数据栈元素入栈都伴随着辅助栈元素入栈，设栈中元素个数最多为 n，则额外的空间开销为 $\mathcal{O}(n)$，因为该算法只涉及到入栈和出栈，三种操作的时间复杂度均为 $\mathcal{O}(1)$。

问题 2

问题描述

- 最多使用一个辅助栈，实现关键字序列的排序（默认正序），并进行相应的算法分析。

算法 1

算法思路

- 考虑用数据栈 $data$ 存储当前序列，辅助栈 aux 存储序列的排序结果。
- 采用类似插入排序的思路，每次从数据栈中取出栈顶元素 val，将辅助栈中小于 val 的元素从辅助栈弹出并压入数据栈，记录弹出元素的个数 cnt，同时将 val 压入辅助栈，根据 cnt 将刚刚压入数据栈的元素再弹出并压回辅助栈。
- 最终从辅助栈中不断弹出栈顶元素并输出，我们便得到了排序结果。
- 最坏情况下，每次均要弹出辅助栈中的所有元素，总的出入栈次数（出栈和入栈合起来算作一次）为 $2n + n(n-1) = n(n+1)$，时间复杂度 $\mathcal{O}(n^2)$。
- 最好情况下，每次均不需要弹出辅助栈中的任何元素，总的出入栈次数为 $2n$，时间复杂度 $\mathcal{O}(n)$。

算法设计与代码

- 输入输出格式如下：
  - 第一行一个正整数 n，表示序列的长度。
  - 接下来一行 n 个整数，表示序列中每个元素。

```cpp
#include <bits/stdc++.h>

using std::stack;

inline void insertSort(int n)
{
    stack<int> data, aux;
    int x;
```

```
    while (n--)
    {
        scanf("%d", &x);
        data.push(x);
    }
    while (!data.empty())
    {
        int m = 0;
        int val = data.top();
        data.pop();
        while (!aux.empty() && aux.top() < v)
        {
            data.push(aux.top());
            aux.pop();
            ++m;
        }
        aux.push(v);
        while (m--)
        {
            aux.push(data.top());
            data.pop();
        }
    }
    while (!aux.empty())
    {
        printf("%d ", aux.top());
        aux.pop();
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    insertSort(n);
    return 0;
}
```

算法演示

- 依次进行以下输入，输出结果已做加粗标识，以便与输入区分。

> 12
> 1 8 2 4 7 6 3 5 2 9 10 9
> **1 2 2 3 4 5 6 7 8 9 9 10**

- 可见程序正确输出了结果。

算法 2

算法思路

- 观察算法 1 的实现，不难发现存在一些冗余的出入栈操作，考虑做进一步优化。
- 在该算法中，设当前欲入栈元素为 val，我们维护两个栈 $mins, maxs$，分别存储小于等于 val 的元素和大于等于 val 的元素，且元素在栈中从栈顶到栈底分别遵循单调不增和单调不降的大小关系。

- 每次新加入一个元素 val，我们需要维护 $mins, maxs$ 的性质，依次做如下操作：
  - 不断将 $mins$ 中大于 val 的栈顶元素弹出并压入 $maxs$。
  - 不断将 $maxs$ 中小于 val 的栈顶元素弹出并压入 $mins$。
  - 将 val 压入 $maxs$。
- 最终将 $mins$ 中所有元素依次弹出并压入 $maxs$，再依次弹出 $maxs$ 的栈顶元素，即得到了排序结果。
- 显然每次调整完，$mins, maxs$ 能满足我们要求的性质，并且每次最多只有一个栈的栈顶元素会被弹出。
- 最坏情况下，每次都恰好将一个栈的所有元素弹出并压入另一个栈，总出入栈次数为 $\frac{n(n-1)}{2} - \lfloor \frac{n-1}{2} \rfloor + n$，时间复杂度 $\mathcal{O}(n^2)$。
- 最好情况下，每次都不需要调整元素，总出入栈次数为 n，时间复杂度 $\mathcal{O}(n)$。

算法设计与代码

- 输入输出格式如下：
  - 第一行一个正整数 n，表示序列的长度。
  - 接下来一行 n 个整数，表示序列中每个元素。

```cpp
#include <bits/stdc++.h>

using std::stack;

inline void stackSort(int n)
{
    stack<int> mins, maxs;
    int x;
    while (n--)
    {
        scanf("%d", &x);
        while (!maxs.empty() && maxs.top() < x)
        {
            mins.push(maxs.top());
            maxs.pop();
        }
        while (!mins.empty() && mins.top() > x)
        {
            maxs.push(mins.top());
            mins.pop();
        }
        maxs.push(x);
    }
    while (!mins.empty())
    {
        maxs.push(mins.top());
        mins.pop();
    }
    while (!maxs.empty())
    {
        printf("%d ", maxs.top());
        maxs.pop();
    }
}

int main()
```

```
{
    int n;
    scanf("%d", &n);
    stackSort(n);
    return 0;
}
```

算法演示

- 依次进行以下输入，输出结果已做加粗标识，以便与输入区分。

```
12
1 8 2 4 7 6 3 5 2 9 10 9
1 2 2 3 4 5 6 7 8 9 9 10
```

- 可见程序正确输出了结果。

总结与讨论

- 算法 2 相对于算法 1 省去了一部分冗余的出入栈操作，且实现更为简单，但两者最坏的时间复杂度均为 $\mathcal{O}(n^2)$。
- 若允许使用的栈的个数超过 2，则一些经典的排序如基数排序、归并排序等也能依赖于栈实现，最终便能得到一个较好的时间复杂度，但相应的空间消耗也会因此增大。
- 如下是用 3 个栈实现的二进制基数排序，已在相关 OJ 做过测试，其中 $D = \lceil \log_2 \max_{i=1}^{n} \{a_i\} \rceil$，总时间复杂度为 $\mathcal{O}(nD)$。

```
inline void radixSort(int n, int D)
{
    stack<int> bit[2], data;
    for (int i = 1, val; i <= n; ++i)
    {
        scanf("%d", &val);
        data.push(val);
    }
    for (int i = 0; i < D; ++i)
    {
        while (!data.empty())
        {
            int val = data.top();
            bit[val >> i & 1].push(val);
            data.pop();
        }
        for (int j = 1; j >= 0; --j)
            while (!bit[j].empty())
            {
                data.push(bit[j].top());
                bit[j].pop();
            }
    }
    n = 0;
    while (!data.empty())
    {
        printf("%d ", data.top());
        data.pop();
    }
}
```