2020/06/03_数据结构_第4课_迭代器bug、栈、队列

笔记本: 数据结构
创建时间: 2020/6/3 星期三 15:37
作者: ileemi
标签: 迭代器删除元素后继元素无法访问bug, 队列, 栈

迭代器bug

当通过迭代器删除结点操作，再遍历该结点后面的结点时，后面的结点无法访问，程序调用断言。

示例：

```
int main()
{
    list<int> lst;
    for (int i = 0; i < 20; i++)
    {
        lst.push_back(i);
    }

    // 将该链表内的数据为2倍数进行删除
    for (auto itr = lst.begin(); itr != lst.end(); ++itr)
    {
```

```
        if (*itr % 2 == 0)
        {
            lst.erase(itr); // 此处出发断言
        }
    }

    return 0;
```

virtual：1，尝试先存储该结点
代码示例：

```
for (auto itr = lst.begin(); itr != lst.end(); ++itr)
{
    if (*itr % 2 == 0)
    {
        auto itrOld = itr++;
        lst.erase(itrOld);
        // 尝试这样做修改，同样出发断言，这里使用自己的Clist类可以解决

        //lst.erase(itr);        // 此处出发断言
        // itr 结点被删除后，后面的结点无法访问，失效的迭代器
        itr--;
    }
}
```

使用自己实现的CList，进行上述的操作，程序运行输出成功

但是对于标准库的问题处理，需要使用while循环进行下面的操作：
代码示例：

```
int main()
{
    list<int> lst;
    for (int i = 0; i < 20; i++)
    {
        lst.push_back(i);
    }

    // 将该链表内的数据为2倍数进行删除
    // 使用while循环解决 迭代器存在的删除结点后，后继结点无法访问的bug
问题
    auto itr = lst.begin();
    while (itr != lst.end())
    {
        //凡是2的倍数的值都删除
        auto itrOld = itr++;
        if (*itrOld % 2 == 0)
```

```
            {
                lst.erase(itrOld);
            }
        }

        for (auto val : lst)
        {
            cout << val << " ";
        }

        cout << endl;

        // 反向迭代器
        for (auto itr = lst.rbegin(); itr != lst.rend(); ++itr)
        {
            cout << *itr << " ";
        }
        cout << endl;
        return 0;
    }
```

运行效果：

反向迭代器

在上面的程序中使用STL提供的反向迭代器，头文件#include <list>
代码示例：

```
    // 反向迭代器
    for (auto itr = lst.rbegin(); itr != lst.rend(); ++itr)
    {
        cout << *itr << " ";
    }
```

代码运行效果图：

C++反向迭代器相关文档

正向、反向迭代器

不适用auto关键字时，需要添加下面的声明

正向迭代器：
`list<int>::iterator`
示例：

```
list<int>::iterator itr = lst.begin();
```

反向迭代器：
`list<int>::reverse_iterator`
示例：

```
for (list<int>::reverse_iterator itr = lst.rbegin(); itr != lst.rend();
++itr)
{
    cout << *itr << " ";
}
```

为什么数据结构都要提供迭代器

主要目的：给所有的数据结构（容器）提供一种统一的遍历方式

代码示例：

```
// 迭代器的目的：为所有的数据结构（容器）提供一种统一的遍历方式
set<int> st;
for (int i = 0; i < 20; i++)
{
    st.insert(i);   // 插入元素数值
}
for (auto itr1 = st.begin(); itr1 != st.end(); ++itr1)
{
    cout << *itr1 << " ";
}
cout << endl;


map<int, int> mp;
for (int i = 0; i < 20; i++)
{
    //mp.insert(i, i);
    mp.insert({ i, i });
}
```

```
for (auto itr = mp.begin(); itr != mp.end(); ++itr)
{
    cout << itr->first << itr->second << " ";
}
```

示例代码运行结点：

完整代码：

```
#include <iostream>
#include <list>
#include <set>
#include <map>
using namespace std;

int main()
{
    list<int> lst;
    for (int i = 0; i < 20; i++)
    {
        lst.push_back(i);
    }

    // 将该链表内的数据为2倍数进行删除
    // 使用while循环解决 迭代器存在的删除结点后，后继结点无法访问的bug
    auto itr = lst.begin();
    while (itr != lst.end())
    {
        //凡是2的倍数的值都删除
        auto itrOld = itr++;
        if (*itrOld % 2 == 0)
        {
            lst.erase(itrOld);
        }
    }
```

```
    for (auto val:lst)
    {
        cout << val << " ";
    }

    cout << endl;

    // 反向迭代器
    for (auto itr = lst.rbegin(); itr != lst.rend(); ++itr)
    {
        cout << *itr << " ";
    }
    cout << endl;


    // 迭代器的目的：为所有的数据结构（容器）提供一种统一的遍历方式

    set<int> st;
    for (int i = 0; i < 20; i++)
    {
        st.insert(i);   // 插入元素数值
    }
    for (auto itr1 = st.begin(); itr1 != st.end(); ++itr1)
    {
        cout << *itr1 << " ";
    }
    cout << endl;


    map<int, int> mp;
    for (int i = 0; i < 20; i++)
    {
        //mp.insert(i, i);
        mp.insert({ i, i });
    }

    for (auto itr = mp.begin(); itr != mp.end(); ++itr)
    {
        cout << itr->first << itr->second << " ";
    }
    return 0;
}
```

链表各个操作的时间复杂度

插入：O(1)
删除：O(1)
查询：有循环，平均O(n)，最好时间复杂度：O(1)，最坏O(n)

随机访问（数组的下标索引）：数组 -->O(1)，链表 -->O(n);
占用的内存：链表的占用内存比数组多，但是插入和删除的效率高（数组的插入O(n)，数组的删除O(n)），
这就是**空间换取时间**思想。

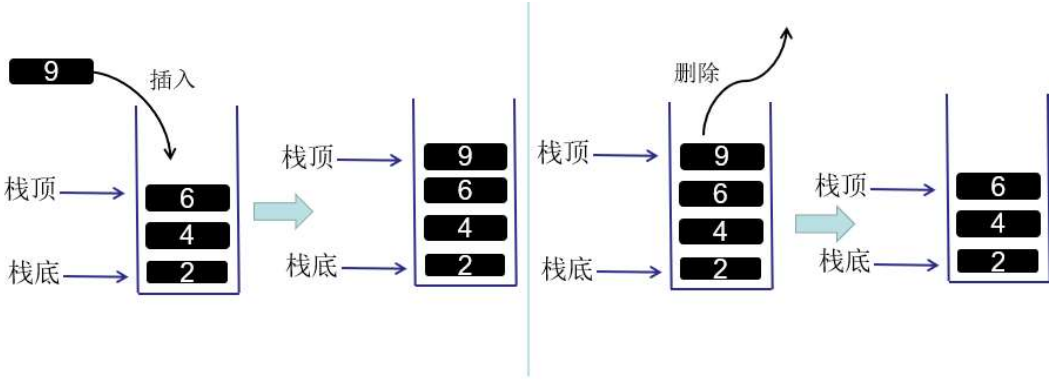
数组和链表的使用时机：
数组：适合频繁查询的操作
链表：适合频繁的插入和删除操作

栈

栈的定义

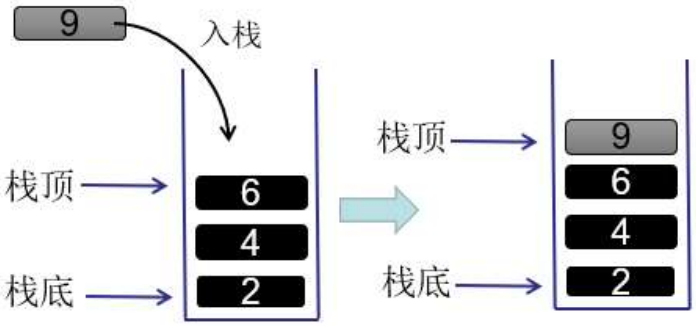
栈是仅限定义在线性表的一端进行插入和删除的线性结构。允许进行插入和删除的一段称之为栈顶，另一端称之为栈底。

先进后出（后进先出）：因为最先入栈的数据，最后出栈，所以栈被称作先进后出结构（FILO -- first in last out），或者后进先出（LIFO -- last in first out）。



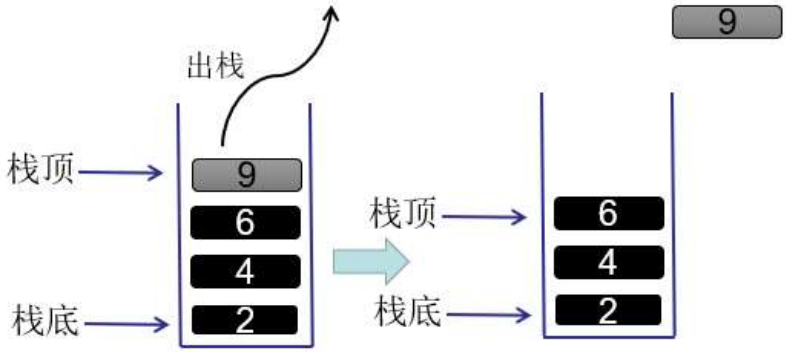
进栈

在栈顶插入数据叫做进栈，也叫做压栈，入栈。



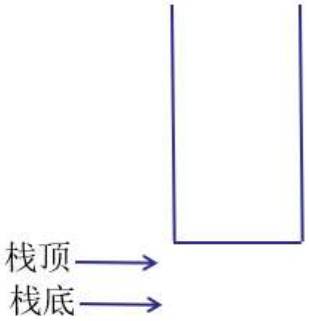
出栈

从栈顶删除数据叫做出栈，也叫做弹栈。



空栈

栈中没有数据元素，则称之为空栈。



栈的操作

- 入栈（push）
- 出栈（pop）
- 访问栈顶元素（top）
- 获取元素个数（size）
- 清空栈（clear）
- 是否为空（empty）

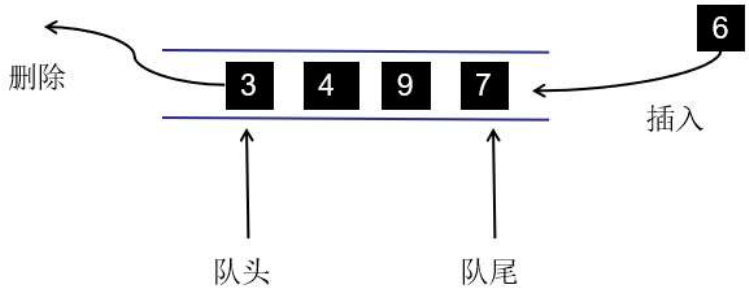
栈的应用

1. 递归
   递归本身的效率比较低，递归层次太深，存在将栈"撑爆"的可能，为此，大多数情况都是通过栈模拟函数的调用，将函数的递归通过栈转换成循环。
2. 后缀表达式（逆波兰表达式）
   运算符的位置：
   中缀：2 + 3 + 5 + 6
   后缀：2 3 + 5 + 8 + 10，（2、3入栈，然后检查到运算符，将2、3弹出做运算(运算符可入可不入栈)，之后将运算后的结果入栈）
3. 匹配检查
   检查某个文件中的括号是不是一一匹配，例如.cpp文件中的 () {} < > 等。空栈匹配，栈残留不匹配

队列（queue）

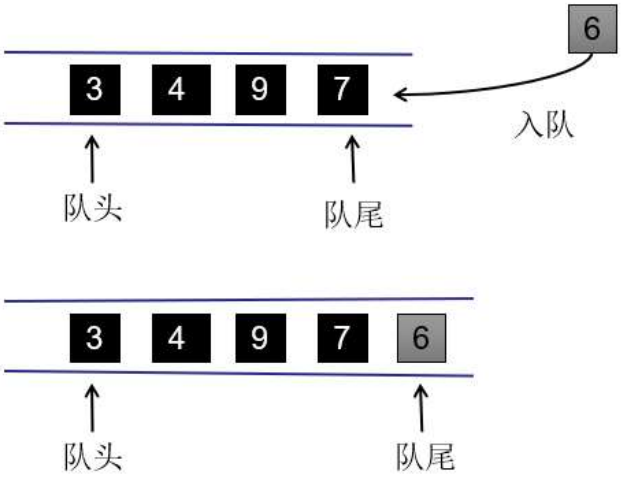
队列的定义

队列是限定于仅在一端进行插入，在另一端进行删除的线性表。
允许插入的一段称之为队尾，允许删除的一端称之为对头。



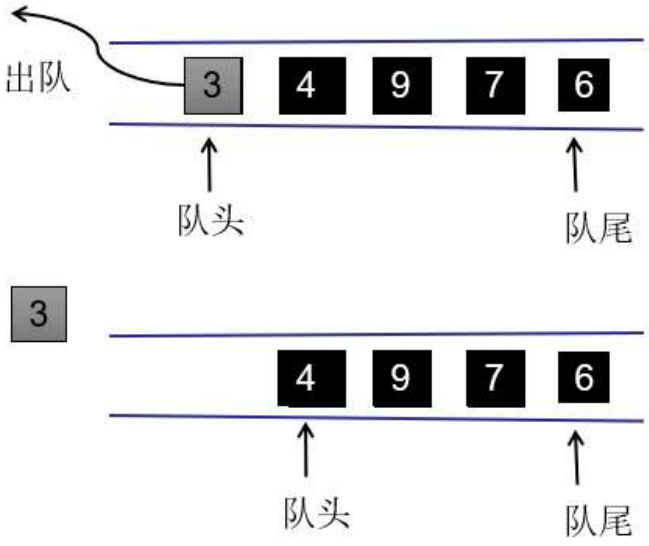
入队

队列的插入操作叫做入队。



出队

队列的删除操作叫做出队



先进先出（后进后出）：最先入队的元素最先出队，所以队列被称之为先进先出（FIFO-first in first out）结构，或者叫做后进后出（LILO- last int last out）。

队列的操作

- 入队（push、enter）
- 出队（pop、leave）
- 访问队首元素（front）
- 获取队列中的元素个数（size）
- 清空队列（clear）
- 是否为空

栈和队列一般不实现迭代器，没有意义，每次访问只访问栈顶，队首。

双栈：
浏览器的前进、后退

Ctrl + Z --> 撤销
Ctrl + Y --> 重做

栈和队列访问的时间复杂度为O(1), 只访问栈顶和队首