
# 数据结构与算法

## 第 2 讲：栈

韩文弢

清华大学

2024—2025 学年度春季学期

# 本讲内容

# 2.1 问题引入

# 2.1.1 操作受限的线性表

在实际生活中，有的线性结构操作是有限制的。

例如，对于**叠起来的盘子**，一般只能从顶部拿走已有的盘子或者放上新的盘子。

思考：生活中还有哪些例子是操作受限的线性结构?

在计算机科学中，这样的结构可以用**操作受限的线性表**来表示。

# 2.1.2 常见的操作受限的线性表

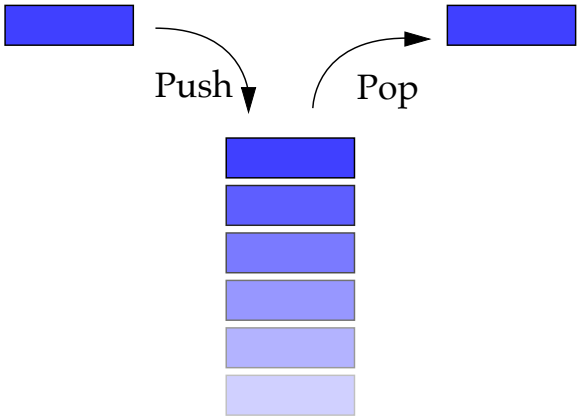



图 2.1 栈

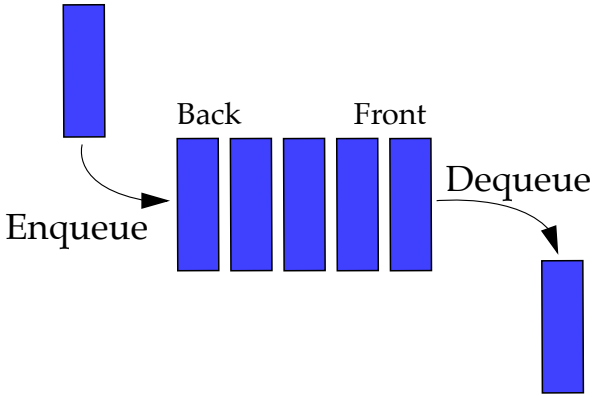



图 2.2 队列

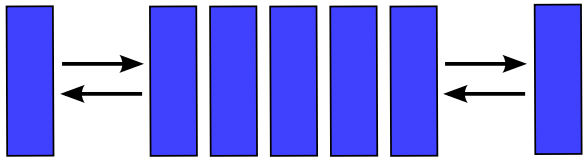


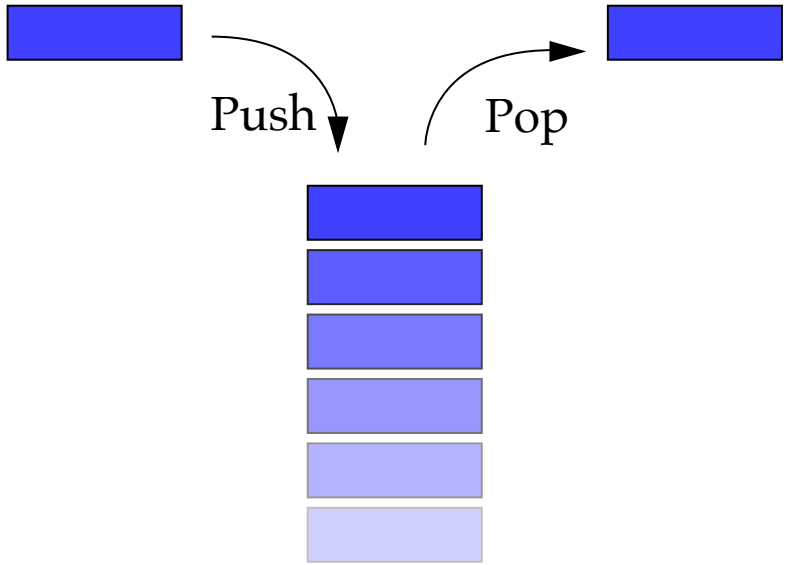
图 2.3 双端队列

# 2.2 栈的概念

# 2.2.1 栈的定义

**定义 2.1.1. 栈**是一种操作受限的线性表，插入和删除元素操作只能在线性表的某一端进行。允许进行操作的一端被称为**栈顶**，另一端被称为**栈底**。

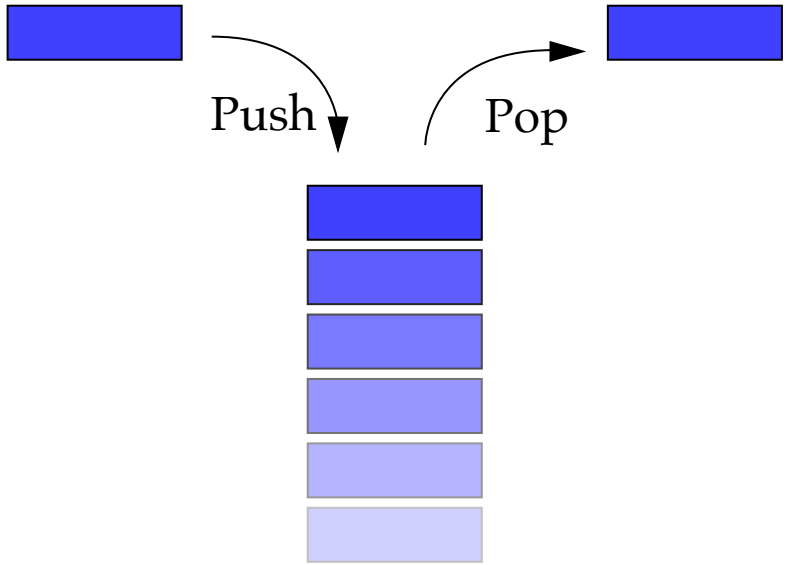
栈具有**后进先出**的特点（last in first out，LIFO），也被称为 LIFO 表。



# 2.2.2 栈的操作

栈的两种基本操作：

- **入栈**（push）：也叫进栈、压栈，向栈顶插入新的元素。
- **出栈**（pop）：也叫退栈、弹栈，从栈顶删除已有元素。



# 2.2.3 栈的抽象数据类型

**伪码 2.1 :** 栈的 ADT

**元素**：$D = \{a_i \mid a_i \in T, i = 0, 1, 2, ..., n-1\}$，$T$ 是元素全集，$n$ 是元素个数

**关系**：$R = \{(a_i, a_{i+1}) \mid a_i, a_{i+1} \in T, i = 0, 1, 2, ..., n-2\}$

**基本操作**：

> Initialize($S$): 初始化一个空的栈 $S$
>
> Destroy($S$): 销毁栈 $S$
>
> Clear($S$): 清空栈 $S$
>
> IsEmpty($S$): 返回栈 $S$ 是否为空
>
> IsFull($S$): 返回栈 $S$ 是否已满
>
> Length($S$): 返回栈 $S$ 中的元素个数
>
> Top($S$): 获得栈 $S$ 的栈顶元素，若栈为空则返回 $\varnothing$
>
> Push($S$, $x$): 将元素 $x$ 压入栈 $S$
>
> Pop($S$): 从栈 $S$ 中弹出栈顶元素并返回，若栈为空则返回 $\varnothing$

# 2.3 栈的实现

# 2.3.1 栈的存储方式

- **顺序栈**：栈的顺序存储实现
- **链式栈**：栈的链式存储实现

# 2.3.2 栈的顺序存储实现

栈可以采用顺序存储方式，称为**顺序栈**。

**伪码 2.2 :** 栈的顺序存储实现

SequentialStack:

> data: 指向数据的指针，数据连续存储
>
> size: 当前元素的个数
>
> capacity: 栈的容量

**伪码 2.3：** 初始化顺序栈

```
Initialize(S):
1   S.data ← Allocate(栈的容量)
2   S.size ← 0
3   S.capacity ← 栈的容量
```

**伪码 2.4：**销毁顺序栈

```
Destroy(S):
1   Free(S.data)
2   S.data ← ∅
3   S.size ← 0
4   S.capacity ← 0
```

# 2.3.5 顺序栈：清空

**伪码 2.5 :** 清空顺序栈

Clear($S$):

1 | $S$.size $\leftarrow 0$

**伪码 2.6 :** 判断顺序栈是否为空

IsEmpty($S$):

1 **return** $S$.size $= 0$

**伪码 2.7 :** 判断顺序栈是否为满

```
IsFull(S):
1   return S.size = S.capacity
```

# 2.3.8 顺序栈：元素个数

**伪码 2.8 :** 获得顺序栈的元素个数

Length($S$):

1 | **return** $S$.size

# 2.3.9 顺序栈：栈顶元素

**伪码 2.9：** 获得栈顶元素

```
Top(S):
1   if S.size > 0
2       return S.data[S.size − 1]
3   else
4       return ∅
```

# 2.3.10 顺序栈：压栈

**伪码 2.10 :** 将一个元素压入顺序栈

```
Push(S, x):
1   if S.size = S.capacity
2       error 栈已满
3   S.data[S.size] ← x
4   S.size ← S.size + 1
```

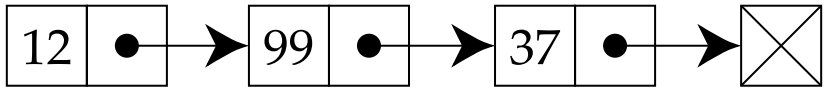
# 2.3.11 顺序栈：弹栈

**伪码 2.11 :** 将栈顶元素弹出顺序栈

```
Pop(S):
1   if S.size > 0
2       S.size ← S.size − 1
3       return S.data[S.size]
4   else
5       return ∅
```

# 2.3.12 栈的链式存储实现

栈也可以采用链式存储方式，称为**链式栈**。一般可以使用单向链表来实现栈。




与顺序栈相比，链式栈

- 优点：一般不需要考虑栈满的问题；
- 缺点：需要更多的额外存储空间。

## 2.3.13 C++ 中的栈 stack

C++ 语言的标准库提供了 stack 类表示栈，注意 stack 类并不是一种容器，而是一种**容器适配器**（container adaptor）。也就是说，它本身不提供实际的存储空间，而是将一种容器包装出需要的接口。

# 2.3.14 stack 的接口

C++

```cpp
#include <stack>

template<class T, class Container = deque<T>>
class stack {
public:
  using value_type = typename Container::value_type;
  // 以及 reference、const_reference 和 size_type
  using container_type  = Container;

  stack();
  explicit stack(const Container&);
```

# 2.3.14 stack 的接口

```
12    template<class InputIter>
13    stack(InputIter first, InputIter last);
14
15    bool empty() const { return c.empty(); }
16    size_type size() const { return c.size(); }
17    reference top() { return c.back(); }
18    const_reference top() const { return c.back(); }
19    void push(const value_type& x) { c.push_back(x); }
20    void pop() { c.pop_back(); }
21  };
```

# 2.3.15 stack 使用示例

C++

```cpp
#include <iostream>
#include <stack>

void ReportStackSize(const std::stack<int>& s) {
    std::cout << s.size() << " elements on stack\n";
}

void ReportStackTop(const std::stack<int>& s) {
    // 元素在访问后仍然留在栈上
    std::cout << "Top element: " << s.top() << '\n';
}
```

# 2.3.15 stack 使用示例

```
12
13  int main() {
14      std::stack<int> s;
15      s.push(2);
16      s.push(6);
17      s.push(51);
18
19      ReportStackSize(s);
20      ReportStackTop(s);
21
22      ReportStackSize(s);
```

# 2.3.15 stack 使用示例

```
23      s.pop();
24
25      ReportStackSize(s);
26      ReportStackTop(s);
27  }
```

# 2.4 栈的应用

# 2.4.1 表达式求值

问题. 给出一个由数字、运算符和小括号组成的表达式，其中数字都是 0 到 9 之间的一位数；运算符有加减乘除（除法结果取整），先乘除后加减；小括号可以嵌套，且是合法的。计算该表达式的值。

例如：

- `1+1` $\longrightarrow$ `2`
- `(1+5)/2` $\longrightarrow$ `3`
- `(8+0)/(9-(2*3+1))` $\longrightarrow$ `4`

# 2.4.2 表达式求值：解题思路

思路一：找到整个表达式优先级最低的运算符，将问题转化为对运算符两边的子表达式求值，然后计算最终结果。子表达式的求值如法炮制（**递归**）。

例如，对于表达式 `(1+5)/2`，其中优先级最低的运算符是除号，左右分别为 `(1+5)` 和 `2`，递归计算这两部分的结果（递归到只剩一个数，这个数就是表达式的值），然后计算整个表达式的值。

# 2.4.3 表达式求值：解题思路

思路二：从左往右扫描表达式，遇到目前还无法决定如何处理的字符时先暂存；一旦确认前面暂存的运算符可以进行时，将运算符和对应参与运算的数取出并进行计算。

例如，对于表达式 `(1+5)/2`，有如下过程（简便起见，在最外面再加一对小括号）：

# 2.4.4 表达式求值：解题思路

对于表达式 `((1+5)/2)`

1. `(`：暂存 `(`；
2. `(`：暂存 `((`；
3. `1`：暂存 `((1`；
4. `+`：暂存 `((1+`；
5. `5`：暂存 `((1+5`；
6. `)`：计算暂存的内容（直到最近的左小括号），得到结果 `6`，将其暂存 `(6`；
7. `/`：暂存 `(6/`；
8. `2`：暂存 `(6/2`；
9. `)`：计算暂存的内容（直到最近的左小括号），得到结果 `3`，将其暂存 `3`。

此时表达式扫描结束，暂存的 `3` 就是最终结果。

# 2.4.5 表达式求值：解题思路

再考虑下面几个表达式在扫描到第二个运算符时的情况：

1. `1*2+3`：第二个运算符优先级较低，此时要计算前面部分。
2. `1+2+3`：同优先级从左往右计算，同上一种情况。
3. `1+2*3`：第二个运算符优先级更高，此时还不能计算。

# 2.4.6 表达式求值：解题思路

启发：

1. 只有遇到优先级更低的运算符时，才去计算之前暂存的式子（推论：暂存的式子中的运算符优先级是递增的）；
2. 计算暂存的式子顺序是由近及远（也就是后进先出），因此使用栈来辅助计算过程；
3. 同一种运算符，暂存之后的优先级要高于正在扫描的优先级。

思考：如何给运算符（包括小括号）制定优先级？

# 2.4.7 表达式求值：制定优先级

| 操作符 | 扫描态优先级 $P$ | 暂存态优先级 $Q$ |
|---|---|---|
| + - | 1 | 2 |
| * / | 3 | 4 |
| ( | 5 | 0 |
| ) | 0 | / |

# 2.4.8 表达式求值：算法

**伪码 2.12 :** 表达式求值

```
Evaluate(E):
E ← '(' + E + ')'; Initialize(S); Initialize(T)
for c in E
    if c ∈ ['0', '9']
        Push(S, c 对应的数值)
    else
        while ¬ IsEmpty(T) ∧ Q[Top(T)] > P[c]
            b ← Pop(S); a ← Pop(S); o ← Pop(T)
            r ← Calculate(a, o, b)
            Push(S, r)
        if ¬ IsEmpty(T) ∧ Q[Top(T)] = P[c]
            Pop(T)  # 弹出左小括号
        else
            Push(T, c)  # 暂存运算符
return Top(S)
```

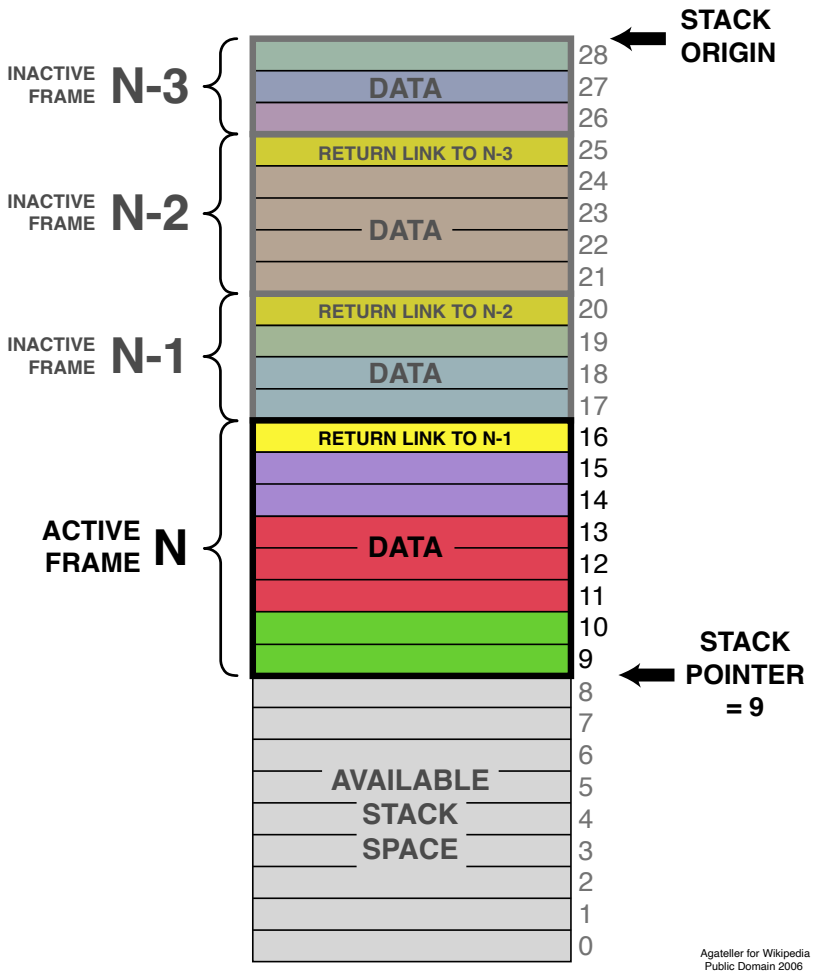
# 2.4.9 函数调用

在多数编程语言中，函数调用依靠栈来实现。

原因：后调用的函数先返回 → 后进先出

每次函数调用会在系统提供的**调用栈**（call stack）上创建一个**调用帧**（call frame），用于存储该次函数调用的参数、返回地址和局部变量。

函数调用层数过大会导致**栈溢出**（stack overflow），常见情况是递归调用终止条件不正确。



# 2.4.10 递归函数

**定义 4.10.1.** 直接或间接调用自己的函数被称为**递归函数**。

```
Factorial(n)
1 | if n ≤ 0
2 |   | return 1
3 | else
4 |   | return n × Factorial(n − 1)
```

尾递归

```
Fibonacci(n)
1 | if n ≤ 1
2 |   | return 1
3 | else
4 |   | return Fibonacci(n − 1) + Fibonacci(n − 2)
```

非尾递归

尾递归可以简单地变换成非递归（迭代）方式来计算。

# 2.5 小结

# 2.5.1 本讲小结

- 栈的概念
  - 特殊位置：栈顶、栈底
  - 基本操作：压栈、弹栈
- 栈的实现：顺序存储方式、链式存储方式
- 栈的应用：表达式求值、函数调用