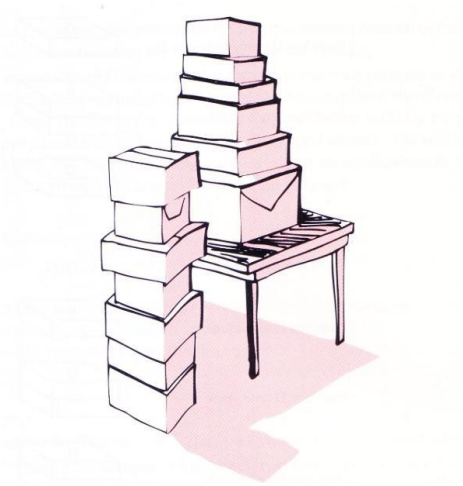
Stack

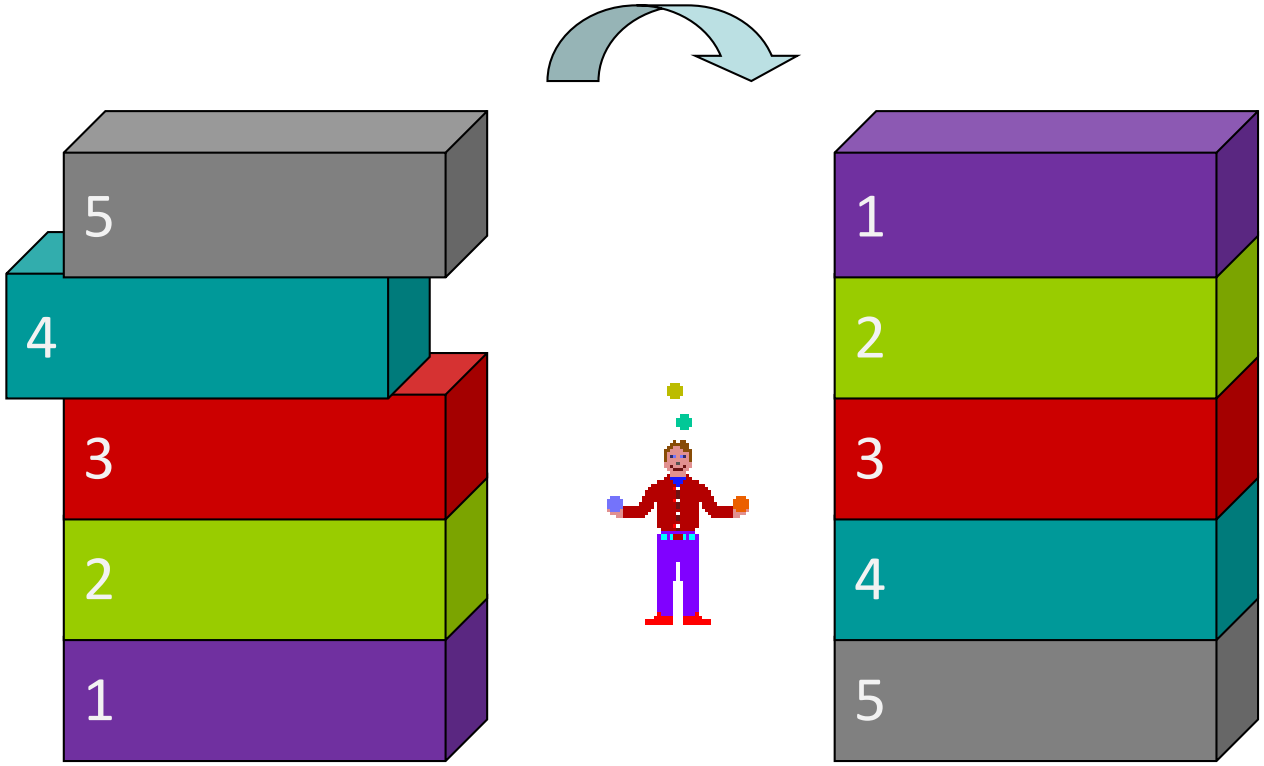
Teacher: Dr. Zhuo SU (苏卓)

E-mail: suzhuo3@mail.sysu.edu.cn

School of Data and Computer Science

Stack





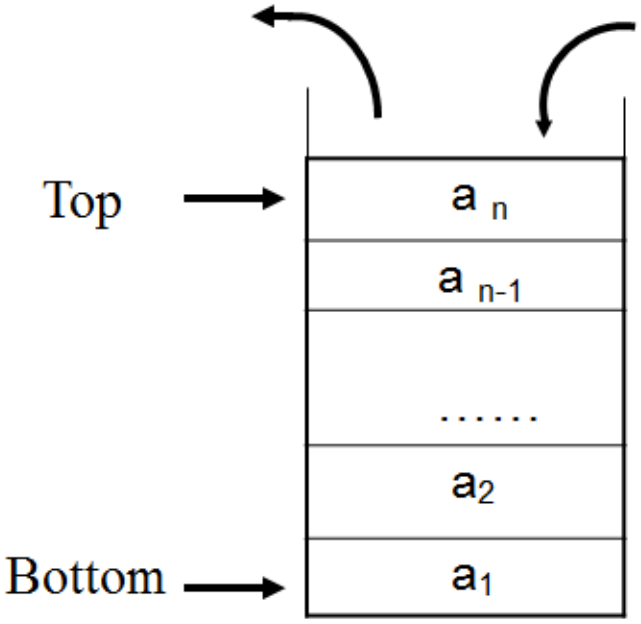
Main contents

- Definition and operations
- Implementation
- Applications

Definition

- Stack（栈）

- 限定仅在表尾进行插入和删除操作的线性表。



Stacks are sometimes known as:

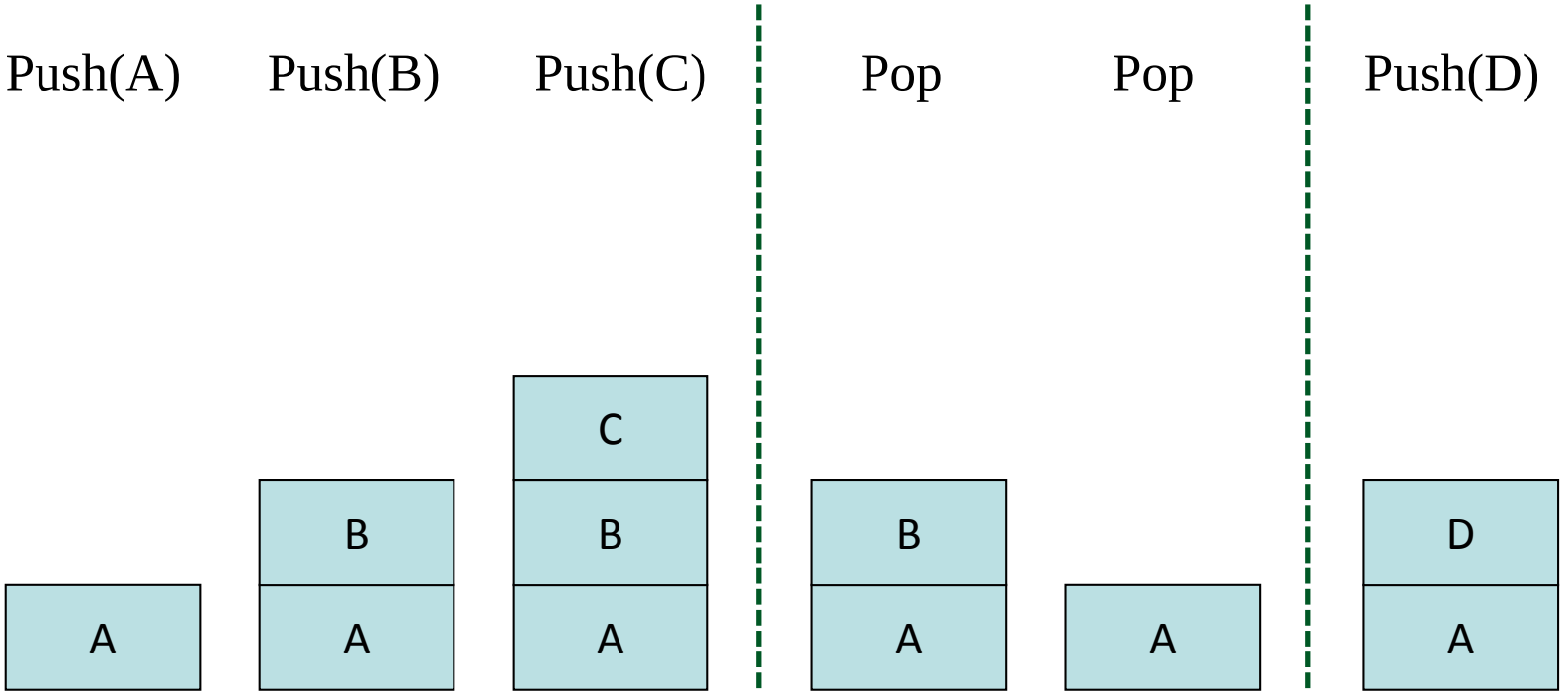
Last In, First Out（LIFO）

Operations

- The fundamental operations on a stack are **push**, which is equivalent to an insert, and **pop**, which deletes the most recently inserted element.

  - **置空栈**：Inistack
  - **判断栈是否为空**：Empty
  - **入栈**：Push
  - **出栈**：Pop
  - **得到栈顶元素**：GetTop

Status of Stack with operations



ADT for stack

ADT Sqlist

Data

$D=\{a_i, a_i \in \mathbf{ElementType}, i=1,2,\ldots,n, n\geq 0\}$

$R=\{<a_{i-1}, a_i> | a_{i-1}, a_i \in D, i=1,2,\ldots,n\}$

Operation

初始化，构造一个空的栈

InitList

Post-condition： None.

ADT for stack

判栈空

EmptyStack(b)

Output：b: Boolean.

Post-condition：Return True if stack is empty; else return False.

入栈

Push(,e)

Input：e: ElementType.

Post-condition：Add element e to top of stack.

ADT for stack

出栈

Pop(e)

Output：Top element e of stack.

Pre-condition：Stack is not empty.

Post-condition：Remove top element from stack.

读栈顶元素

Top(e)

Output：Top element e of stack .

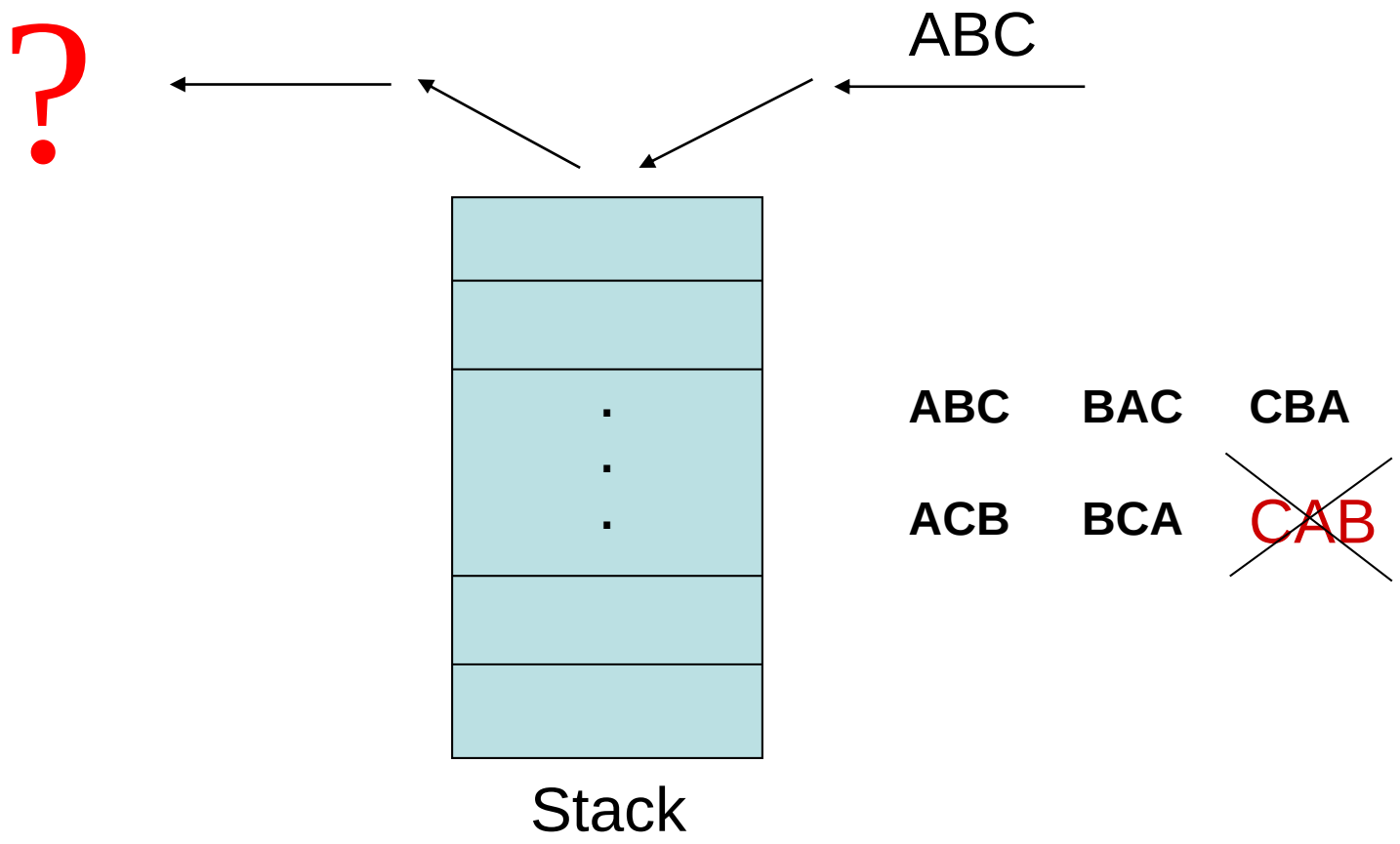
Pre-condition：Stack is not empty .

Post-condition：Return top element from stack.

End ADT

A case

- Which sequences are impossible ?



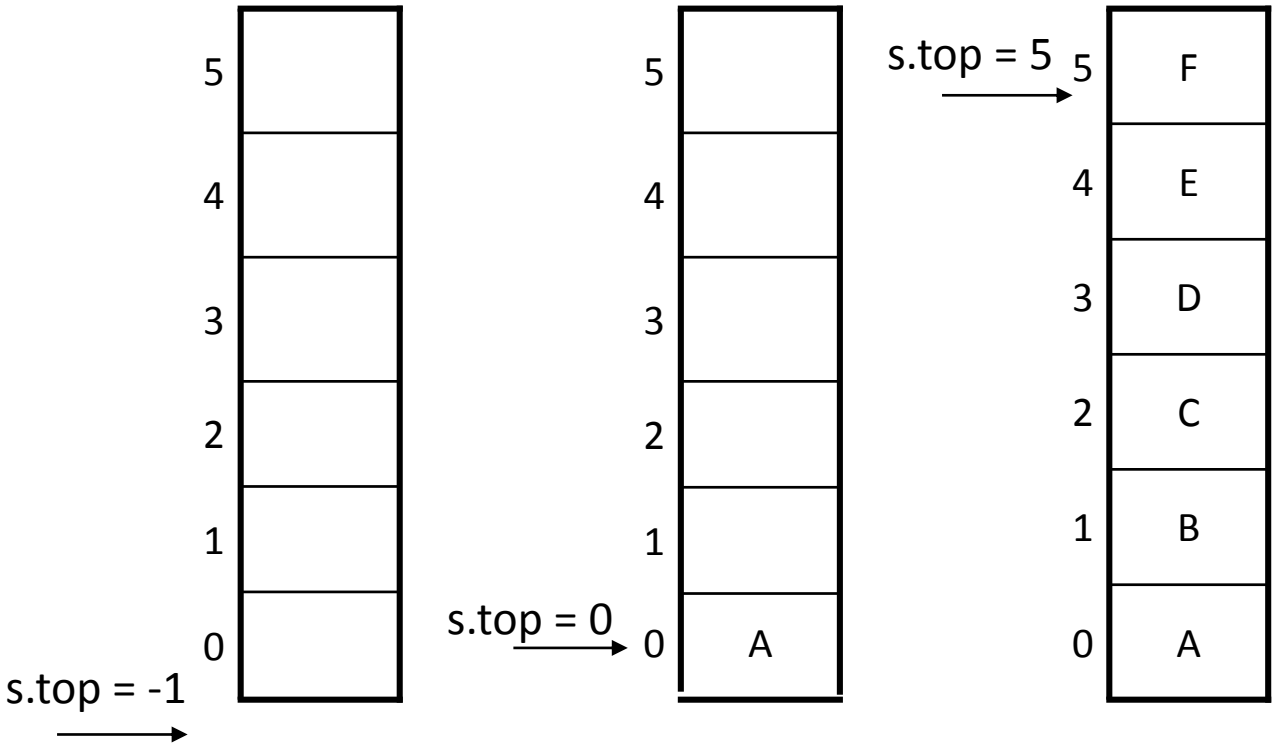
Stack Implementation: Sequence LSist

- In sequence list

```
Const int maxstack = 10;
Class Stack {
  Public:
    Stack();
    bool empty() const;
    Error_code pop();
    Error_code top(Stack_entry &item) const;
    Error_code push(const Stack_entry &item);

   private:
    int count;
    Stackentry entry[maxstack];
}
```

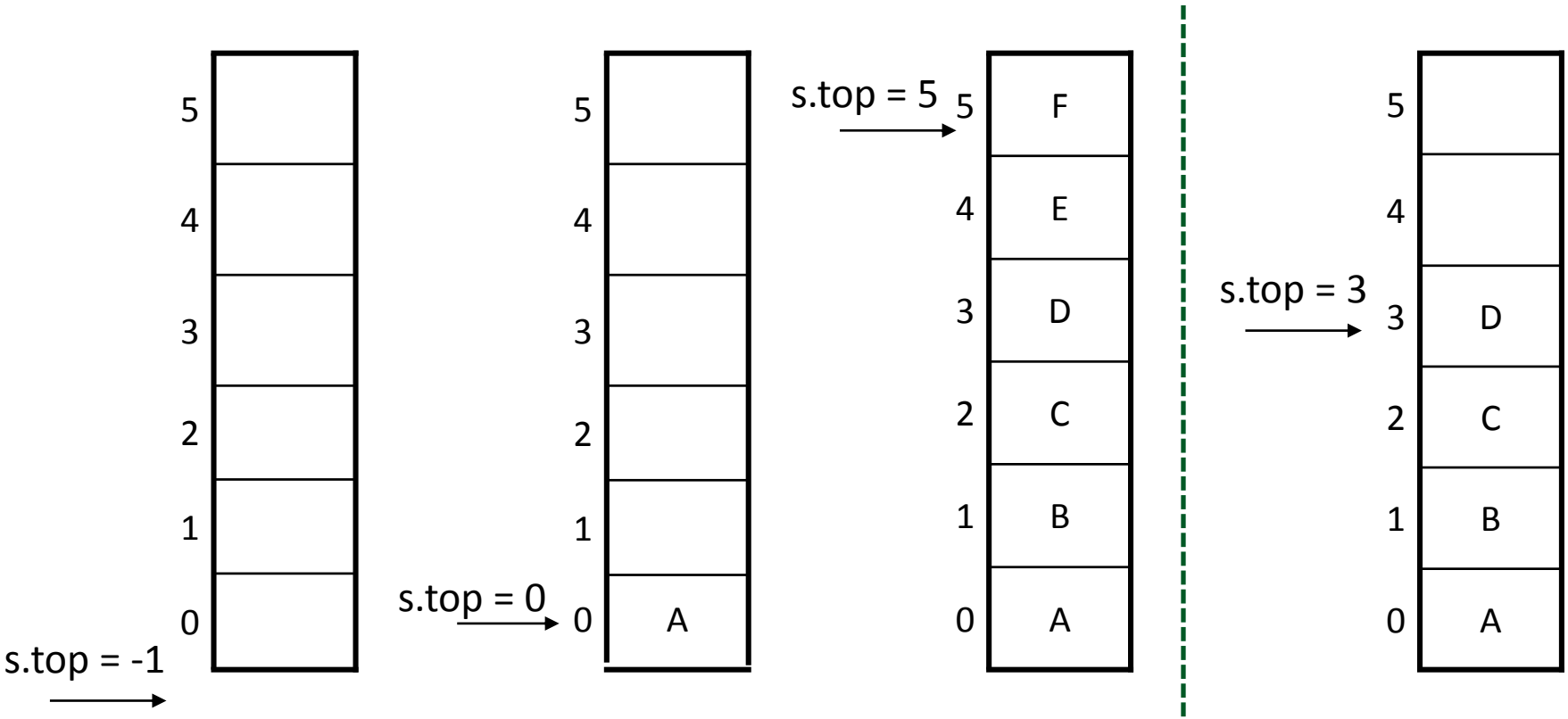
Sequenced stack



压栈操作 push

```
bool arrStack<T>::push(const T item) {
   if (top == mSize-1) {                    // 栈已满
            cout << "栈满溢出" << endl;
            return false;
   }
   else {                                   // 新元素入栈并修改栈顶指针
            st[++top] = item;
            return true;
   }
}
```

Sequenced stack



出栈操作 pop

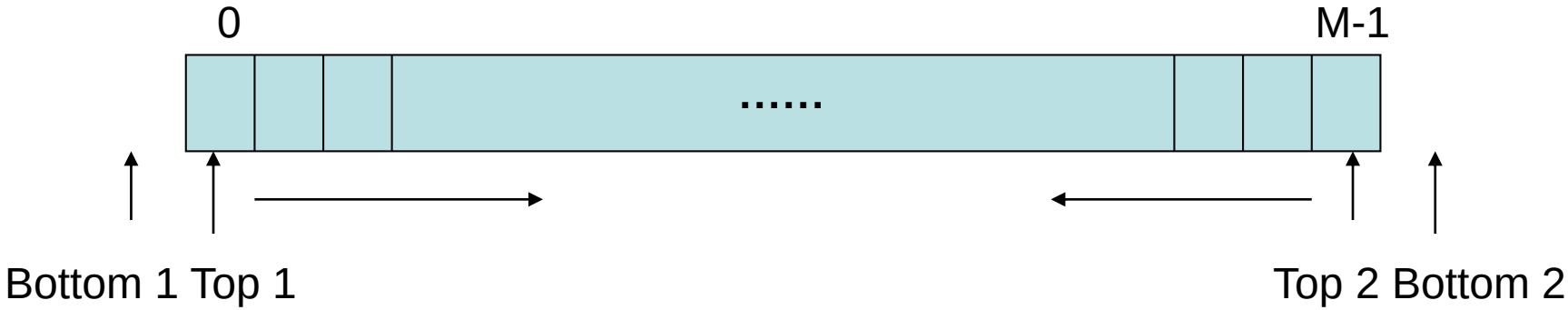
```
bool arrStack<T>::pop(T & item) {  // 出栈的顺序实现
   if (top == -1) {                        // 栈为空
          cout << "栈为空，不能执行出栈操作"<< endl;
          return false;
   }
   else {
          item = st[top--];                // 返回栈顶元素并修改栈顶指针
          return true;
   }
}
```

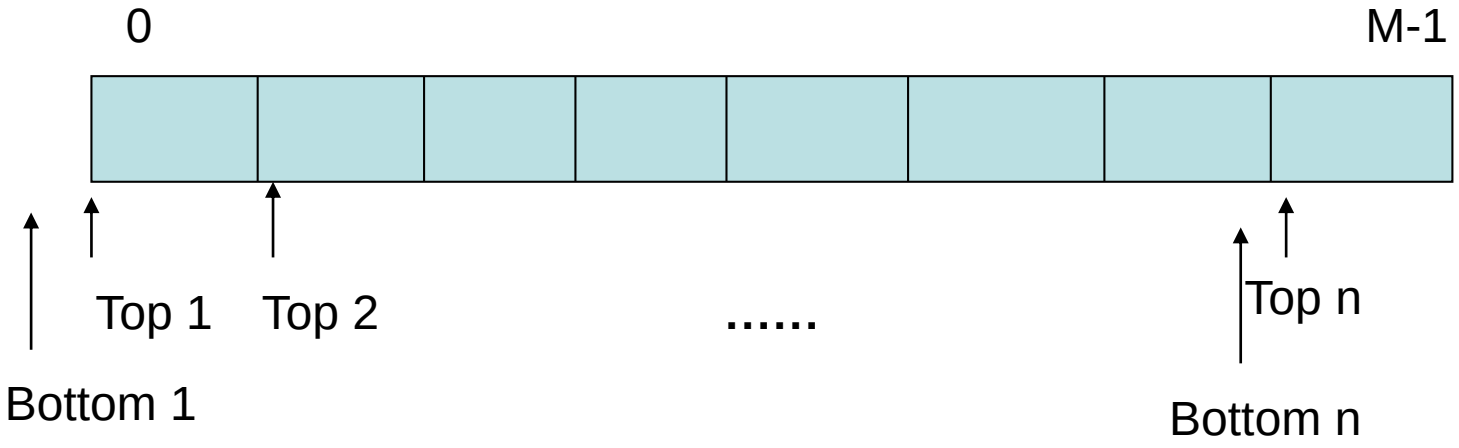
读栈操作 top

```
bool arrStack<T>:: top(T & item) {
                                    // 返回栈顶内容，但不弹出
   if (top == -1) {                 // 栈空
            cout << " 栈为空，不能读取栈顶元素"<< endl;
            return false;
   }
   else {
            item = st[top];
            return true;
   }
}
```

Variants of the stacks

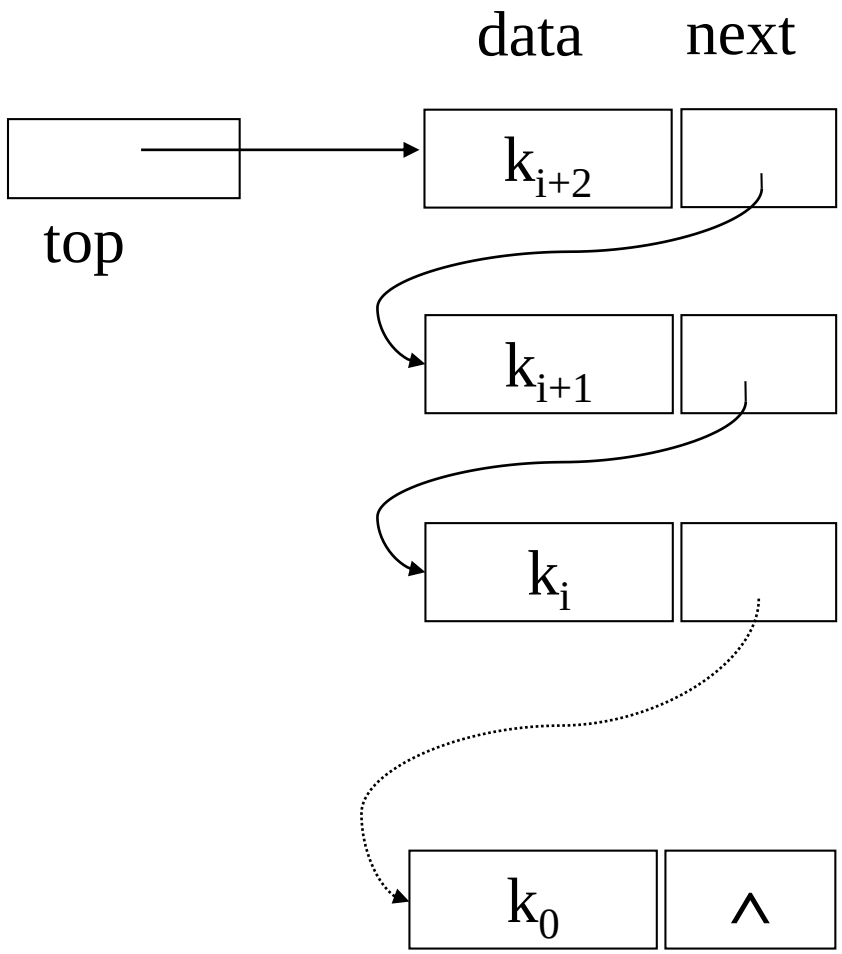
- 共享栈空间





Stack Implementation: Linked List

- 链式栈：用单链表方式存储
- 指针的方向从栈顶向下链接




```
template <class T>
class lnkStack : public Stack <T> {
private:                        // 栈的链式存储
    Link<T>*top;                // 指向栈顶的指针
    int         size;           // 存放元素的个数
public:                         // 栈运算的链式实现
    lnkStack(int defSize) {                 // 构造函数
       top = NULL;
       size = 0;
    }
    ~lnkStack() {               // 析构函数
        clear();
    }
}
```

Operations

压栈操作

```
// 入栈操作的链式实现
bool lnksStack<T>:: push(const T item) {
  Link<T>* tmp = new Link<T>(item, top);
   top = tmp;
   size++;
   return true;
}
```

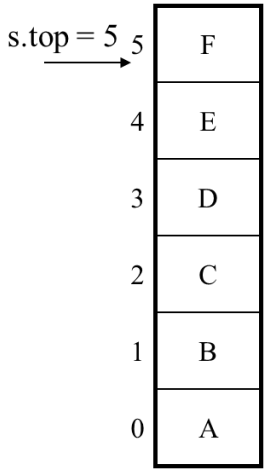
出栈操作

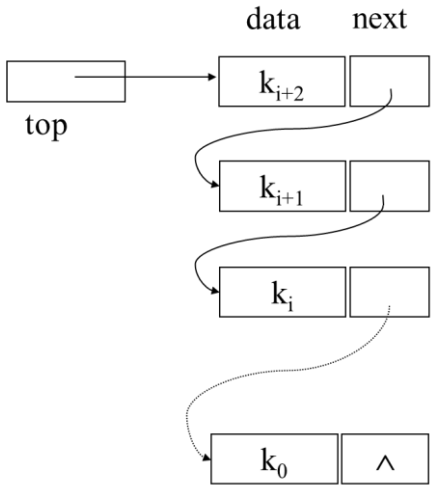
```
// 出栈操作的链式实现
bool lnkStack<T>:: pop(T& item) {

    Link <T> *tmp;
    if (size == 0) {
       cout << "栈为空，不能执行出栈操作"<< endl;
       return false;
    }
    item = top->data;
    tmp = top->next;
    delete top;
    top = tmp;
    size--;
    return true;
}
```

顺序栈和链式栈的比较

- 时间效率
  - 所有操作都只需常数时间
  - 顺序栈和链式栈在时间效率上难分伯仲
- 空间效率
  - 顺序栈须说明一个固定的长度
  - 链式栈的长度可变，但增加结构性开销



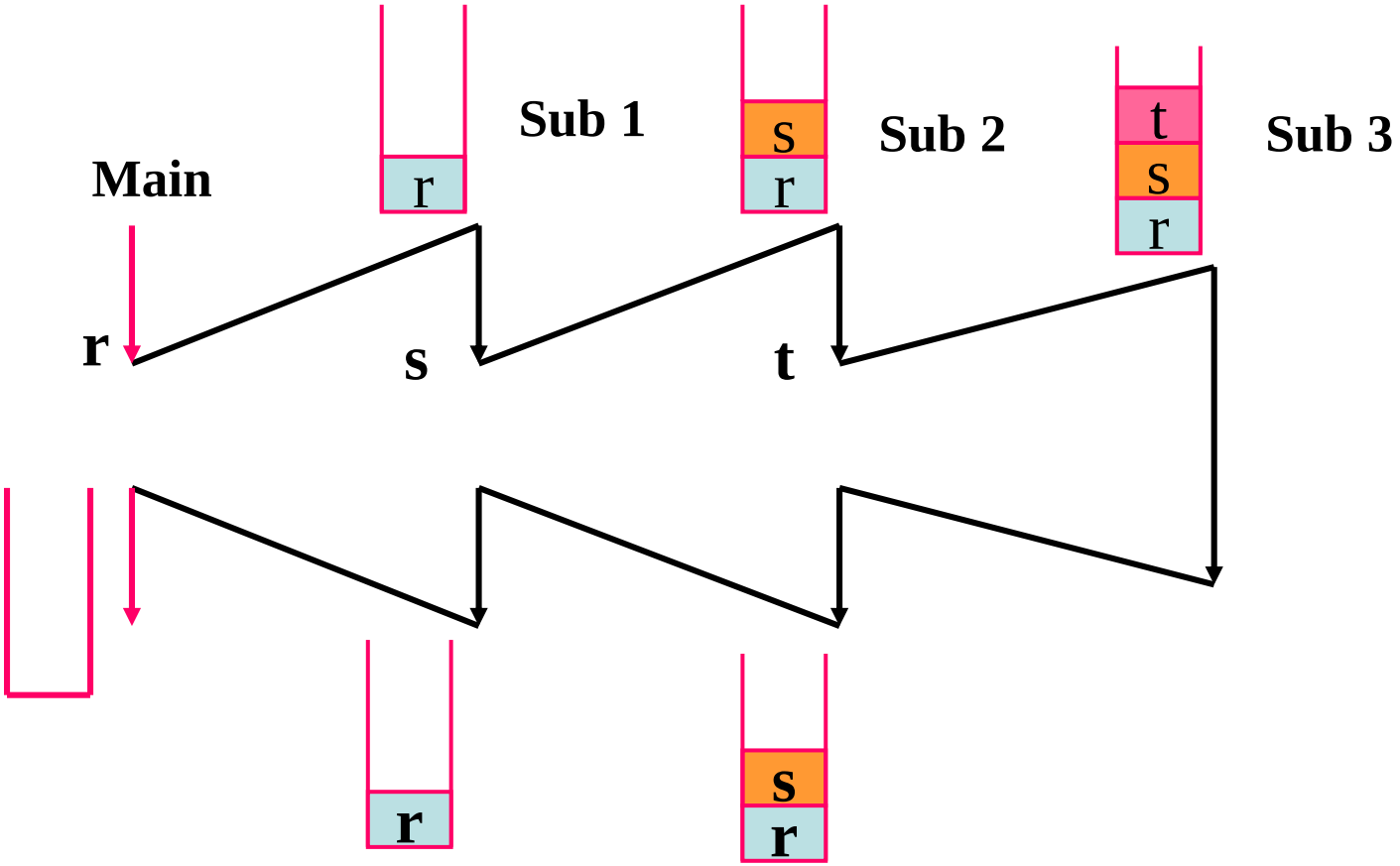


顺序栈和链式栈的比较

- 实际应用中，顺序栈比链式栈用得更广泛些
  - 存储开销低
  - 顺序栈容易根据栈顶位置，进行相对位移，快速定位并读取栈的内部元素
    - 顺序栈读取内部元素的时间为**O(1)**，而链式栈则需要沿着指针链游走，显然慢些，读取第**k**个元素需要时间为**O(k)**
    - 一般来说，栈不允许“读取内部元素”，只能在栈顶操作

Applications

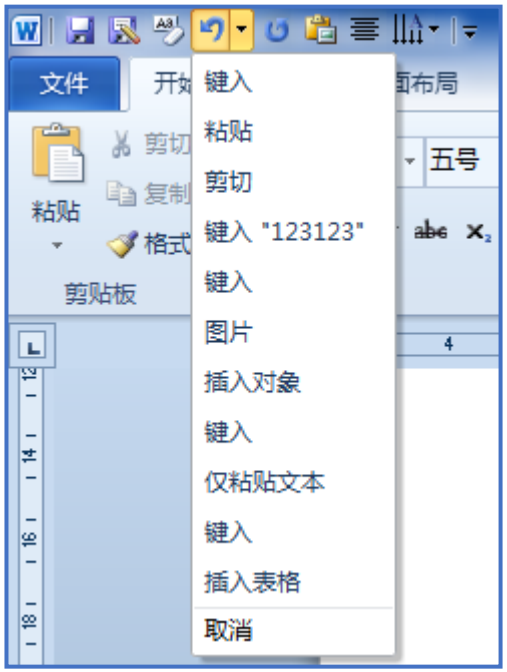
- Function call



Applications

- Historical record in the software

Office Word

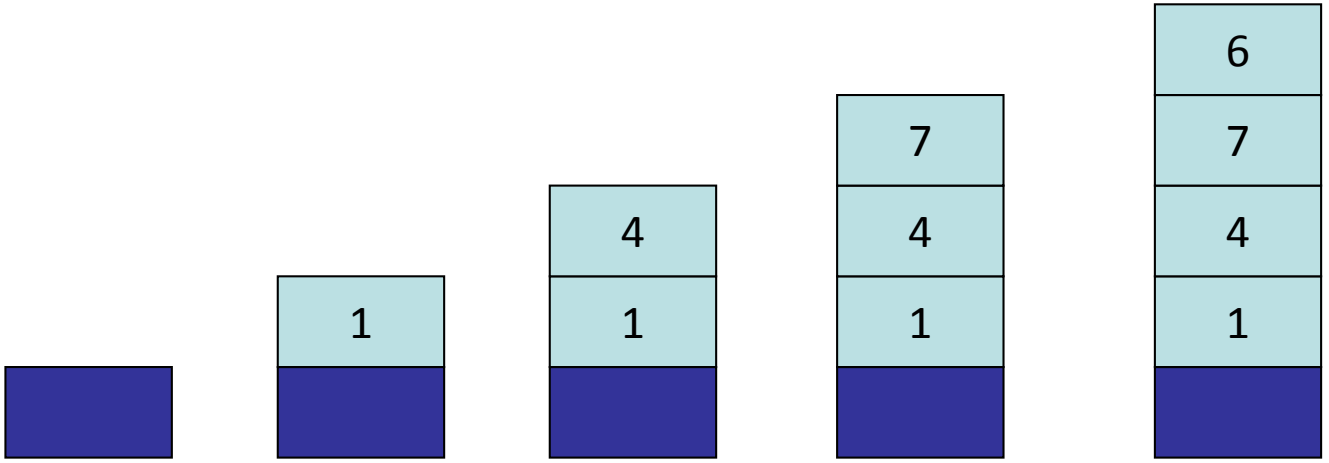


Photoshop

Applications

- 数制转换
  - 非负十进制数转换成其它进制的数的一种简单方法

例：十进制转换成八进制 $(3553)_{10} = (6741)_8$



Empty	n%8 = 1	n%8 = 4	n%8 = 7	n%8 = 6
n = 3553	n/8 = 444	n/8 = 55	n/8 = 6	n/8 = 0
	n = 444	n = 55	n = 6	n = 0

Applications

- 表达式求值

4 + 2 x 3 - 10 / 5

51 x (24 - 15 / 3) + 6

四则运算规则：

先乘除，后加减，先括号内，后括号外。

同一运算级别，从左到右。

- 中缀表达式

23+(34*45)/(5+6+7)

- 后缀表达式

23 34 45 * 5 6 + 7 + / +

表达式的语法定义

- **中缀**表达式的语法公式

```
<表达式> ::= <项> + <项>
           | <项> — <项>
           | <项>
<项>     ::= <因子> * <因子>
           | <因子> / <因子>
           | <因子>
<因子>   ::= <常数>
           | ( <表达式> )
<常数>   ::= <数字>
           | <数字><常数>
<数字>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

后缀表达式的语法公式

```
<表达式> ::= <项> <项> +
           | <项> <项> –
           | <项>
<项>     ::= <因子> <因子> *
           | <因子> <因子> /
           | <因子>
<因子>   ::= <常数>
<常数>   ::= <数字>
           | <数字><常数>
<数字>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Applications

- Symbol Priority

Compare θ_1 and θ_2

θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

Applications

- 中缀表达式求值

- 1：运算数栈置空，操作符栈压入算符“#”
- 2：依次读入表达式的每个单词
- 3：如果是运算数，压入运算数栈
- 4：如果是操作符，将操作符栈顶元素 θ_1 与读入的操作符 θ_2 进行优先级比较
  - 4.1 如果栈顶元素优先级低，将 θ_2 压入操作符栈
  - 4.2 如果相等，弹出操作符栈
  - 4.3 如果栈顶元素优先级高，弹出两个运算数，一个操作符，进行计算，并将计算结果压入运算数栈，重复第4步的判断
- 5:直至整个表达式处理完毕

Applications

- 3*(7-2)#

步骤	操作符栈	运算数栈	输入字符	操作
1	#		3*(7-2)#	压入“3”
2	#	3	*(7-2)#	压入“*”
3	#*	3	(7-2)#	压入“(”
4	#*(	3	7-2)#	压入“7”
5	#*(	37	-2)#	压入“-”
6	#*(-	37	2)#	压入“2”
7	#*(-	372	)#	弹出“-”压入7-2
8	#*(	35	)#	弹出“(”
9	#*	35	#	计算3*5
10	#	15	#	操作符栈空，结束

Applications

- 中缀→后缀表达式的转换

中缀	后缀
4+3*5	4,3,5 *+
2*(5+9*4/2)+6*5	2,5,9,4*2/+*6,5*+

中缀表达式中，运算符的出现次序与计算顺序不一致；

后缀表达式中，运算符的出现次序就是计算次序。

Applications

- 中缀转后缀表达式

A+B*C-D#　　ABC*+D-

读到的符号	运算符栈	输出序列
A	#	A
+	#+	A
B	#+	AB
*	#+*	AB
C	#+*	ABC
-	#-	ABC*+
D	#-	ABC*+D
#	#	ABC*+D-

	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

Applications

- Conversion algorithm

- **1**：操作符栈压入算符“#”
- **2**：依次读入表达式的每个单词
- **3**：如果是运算数，则输出
- **4**：如果是操作符，将操作符栈顶元素 θ_1 与读入的操作符 θ_2 进行优先级比较
  - 4.1 如果栈顶元素优先级低，将 θ_2 压入操作符栈
  - 4.2 如果相等，弹操作符栈
  - 4.3 如果栈顶元素优先级高，弹出栈定元素并输出，重复第4步的判断
- **5**：直至整个表达式处理完毕

Applications

- 后缀表达式求值

4+3*5	**4,3,5 *+**
2*(5+9*4/2)+6*5	**2,5,9,4*2/+*6,5*+**

求解算法：

- 设定一个**运算数栈**OPND;
- 从左向右依次读入，当读到的是**运算数**，将其加入到**运算数栈**中；
- 若读入的是**操作符**，从运算数栈取出两个元素，与读入的**操作符**进行运算，将运算结果加入到**运算数栈**。
- 直到表达式的最后一个**操作符**处理完毕。

Recursion

- **递归的定义**
  - 子程序（或函数）直接**调用自己**或通过一系列调用语句间接调用自己，是一种描述问题和解决问题的基本方法。
- **递归的基本思想**
  - 问题分解：把一个**不能或不好解决**的大问题转化为一个或几个小问题，再把这些小问题进一步分解成更小的小问题，直至每个小问题都可以直接解决。
- **递归的要素**
  - 递归**边界条件**：确定递归到何时终止，也称为递归出口；
  - 递归模式：大问题是如何分解为小问题的，也称为递归体。

计算阶乘 n!

- 阶乘 n! 的递归定义如下：

$$f(n)=\begin{cases}1 & n=0\\ n\times f(n-1) & n>0\end{cases}$$

```
long factorial (long n)
{
    int temp;
    if (n==0)
        return 1;
    else
    {
        temp=n*factorial(n-1);
        return temp;
     }
}
```