我笃定一生砥砺奋进的担当，
就算不谙世事，也有光芒。

开讲啦!

The Third Course

Stack & Queue

Stacks

- LIFO: Last In, First Out
- Restricted form of list
  - Insert and remove only at front of list
- A stack is a data structure in which all insertions and deletions of entries are made at one end, called the top of the stack.

Stacks

- The last entry which is inserted is the first one that will be removed.

- Notation:
  - Insert: PUSH
  - Remove: POP
  - The accessible element is called TOP

DEFINITION

- DEFINITION: A stack of elements of type T is a finite sequence of elements of T , together with the following operations:
  - ***Create*** the stack, leaving it empty.
  - Test whether the stack is ***Empty***.
  - ***Push*** a new entry onto the top of the stack, provided the stack is not full.
  - ***Pop*** the entry off the top of the stack, provided the stack is not empty.
  - Retrieve the ***Top*** entry from the stack, provided the stack is not empty.

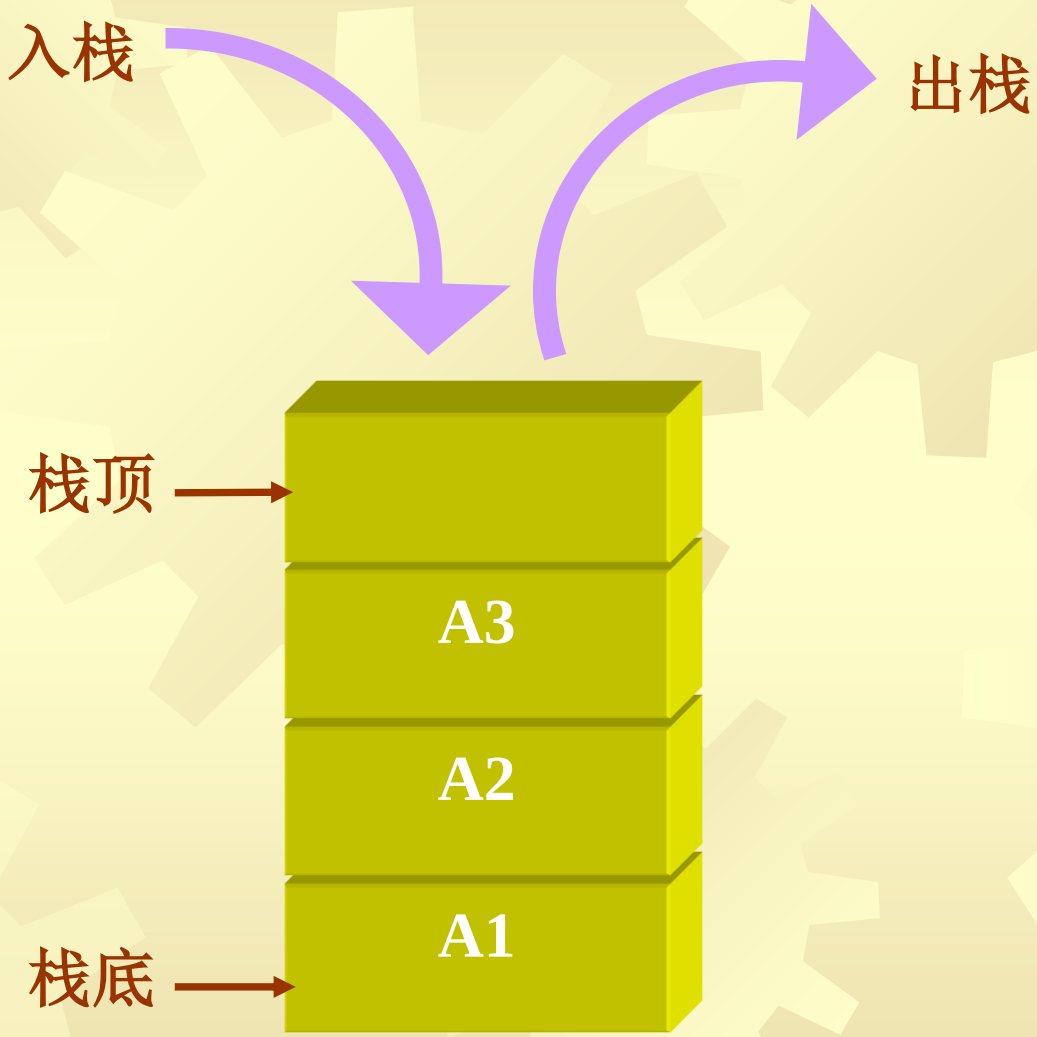
Specification for Methods

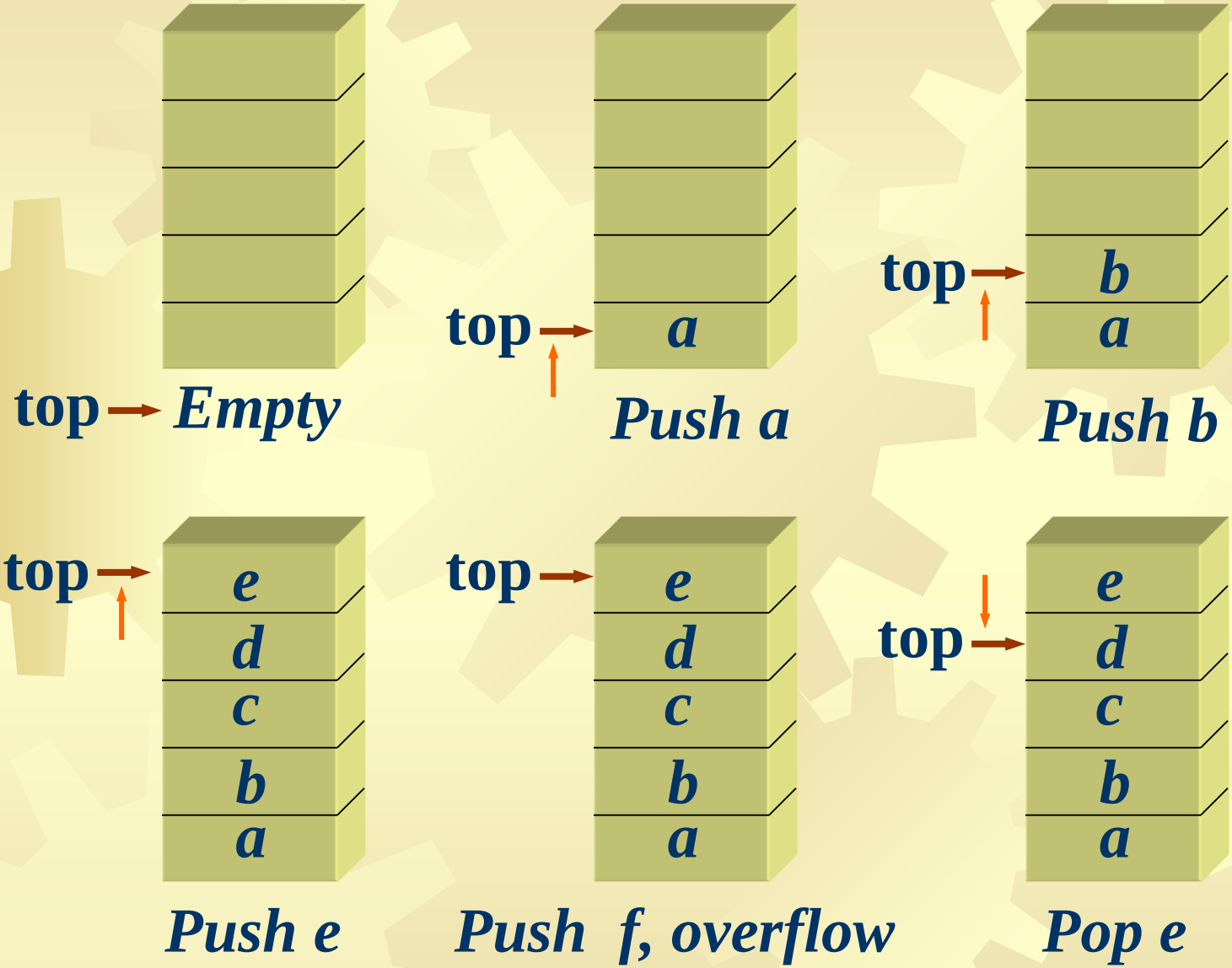
- Error_code Stack ::pop( );
  - Pre: None.
  - Post: If the Stack is not empty, the top of the Stack is removed. If the Stack is empty, an Error_code of underflow is returned and the Stack is left unchanged.
- Error_code Stack ::push(const Stack entry &item);
  - Pre: None.
  - Post: If the Stack is not full, item is added to the top of the Stack. If the Stack is full, an Error_code of overflow is returned and the Stack is left unchanged.

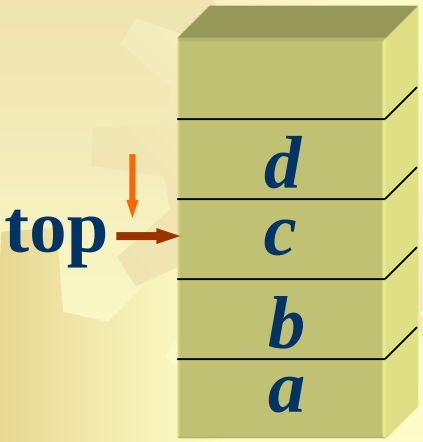
Specification for Methods

- Error_code Stack ::top(Stack entry &item) const;
  - Pre: None.
  - Post: The top of a nonempty Stack is copied to item. A code of fail is returned if the Stack is empty.

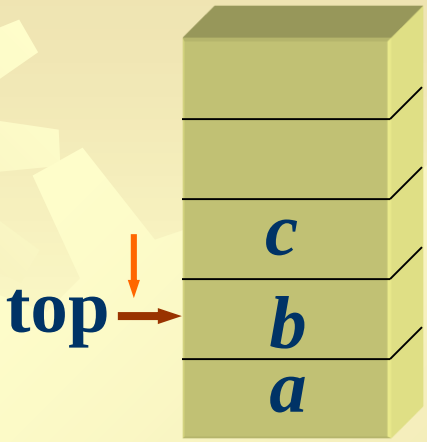
- bool Stack ::empty( ) const;
  - Pre: None.
  - Post: A result of true is returned if the Stack is empty, otherwise false is returned.


入栈
出栈
栈顶
A3
A2
A1
栈底

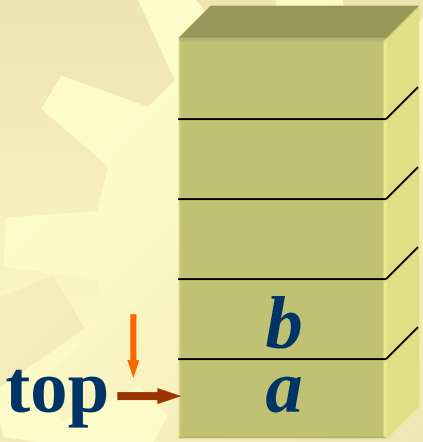

top
Empty
top
a
Push a
top
b
a
Push b
top
e
d
c
b
a
Push e
top
e
d
c
b
a
Push f, overflow
top
e
d
c
b
a
Pop e



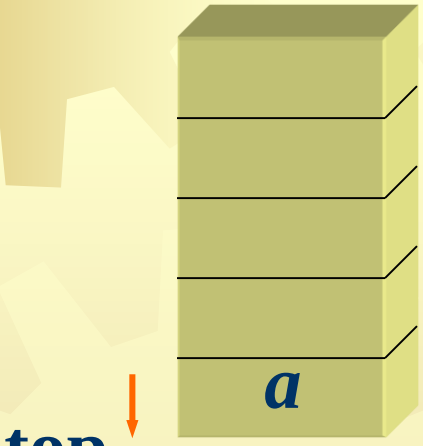
Pop d



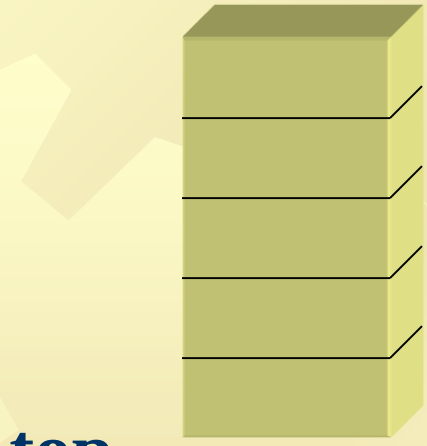
Pop c



Pop b



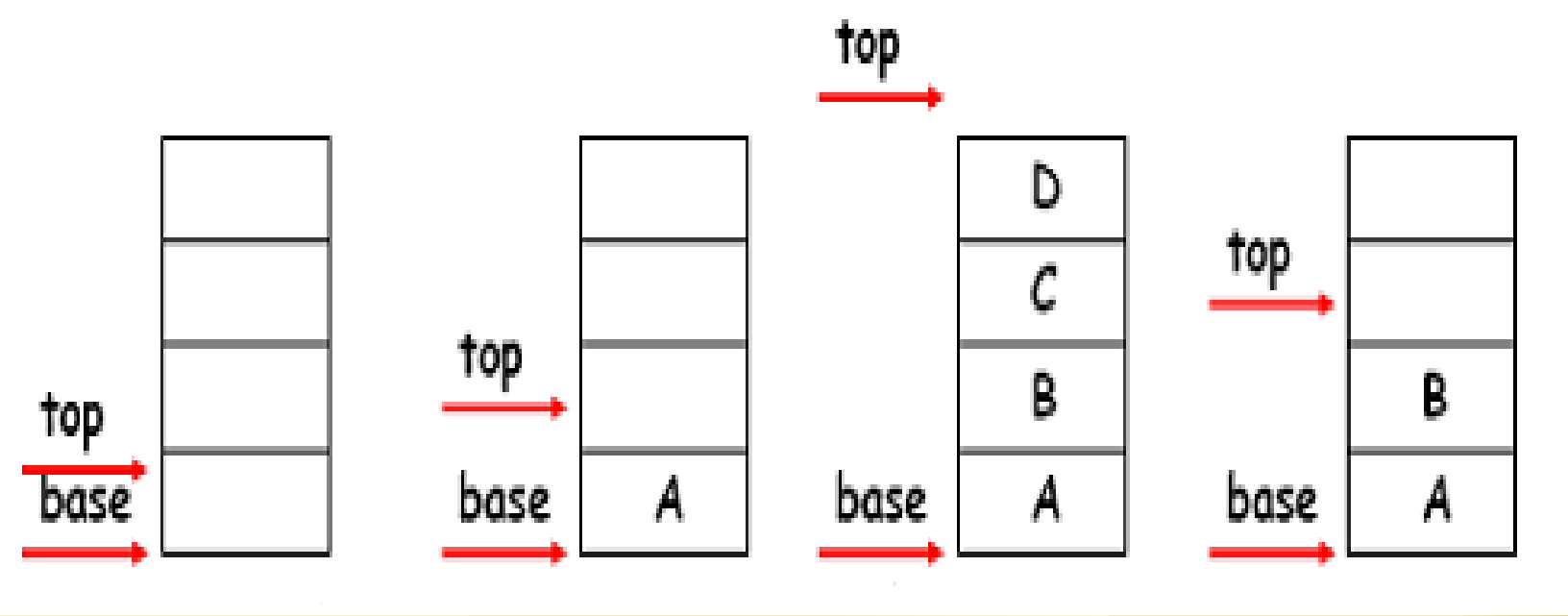
Pop a



Empty

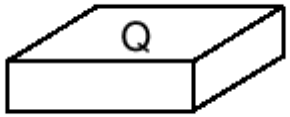
栈顶指针指示
实际栈顶位置
即最后加入新
元素的位置

Top vs. Element in stack

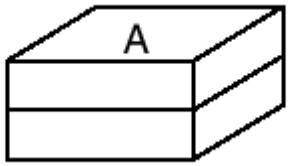


Example

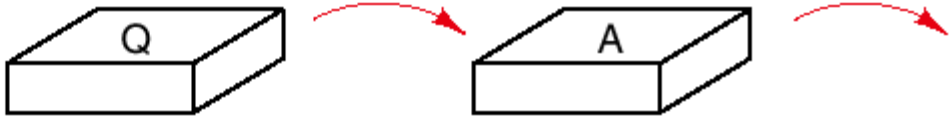
Push box Q onto empty stack:



Push box A onto stack:

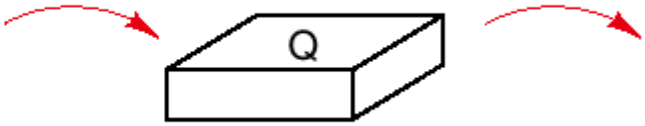


Pop a box from stack:



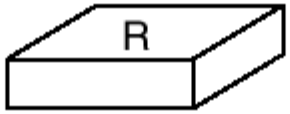
Pop a box from stack:

(empty)

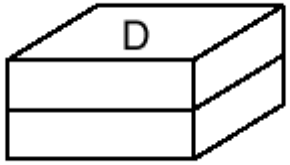


Example

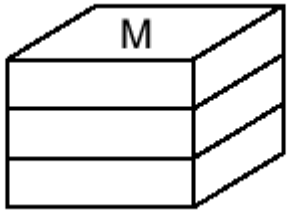
Push box R onto stack:



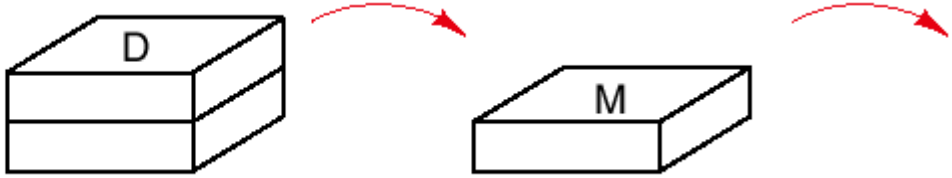
Push box D onto stack:



Push box M onto stack:

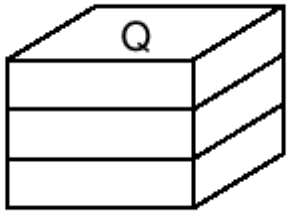


Pop a box from stack:

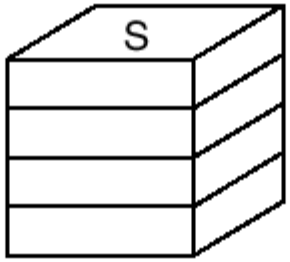


Example

Push box Q onto stack:



Push box S onto stack:



Class Specification, Contiguous Stack

- We set up an array to hold the entries in the stack and a counter to indicate how many entries there are.

Class Specification, Contiguous Stack

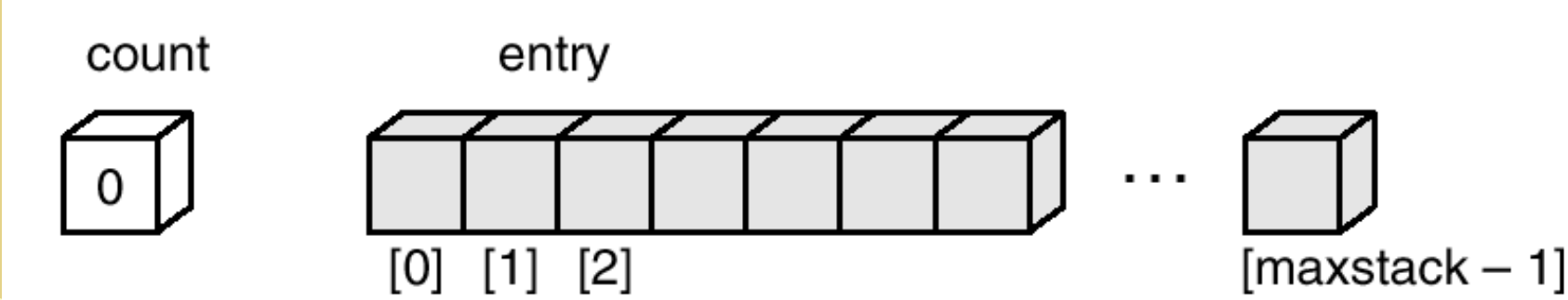
- const int maxstack = 10; //small value for testing

```
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;
    Stack entry entry[maxstack];
};
```

Notes

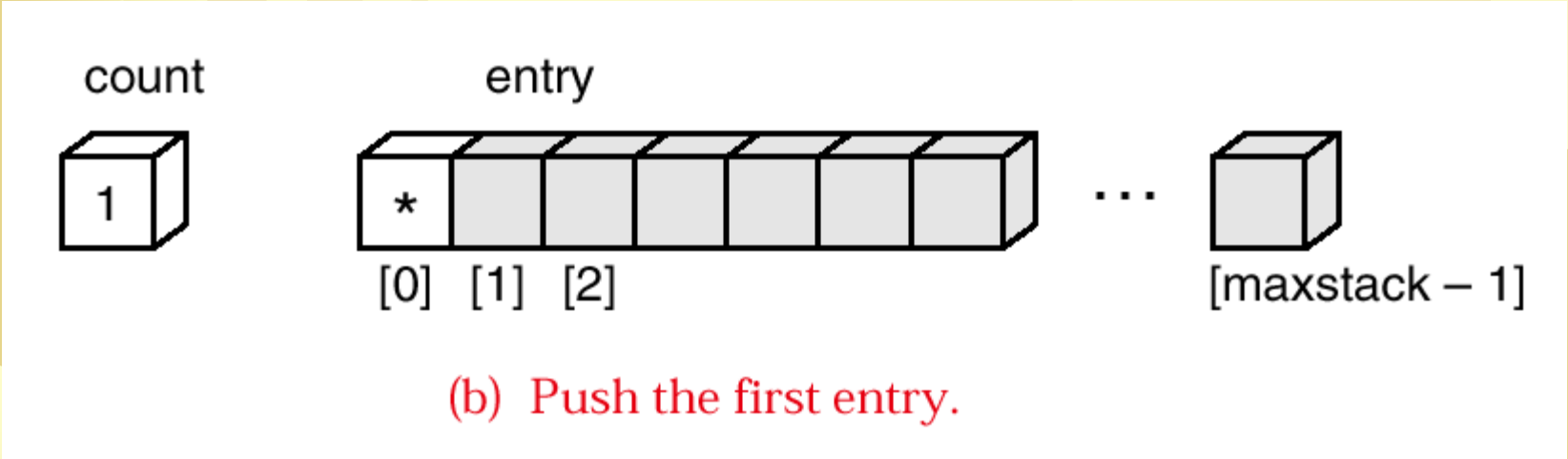
- The declaration of private visibility for the data makes it is impossible for a client to access the data stored in a Stack except by using the methods push( ), pop( ), and top( ).
- Important result: A Stack can never contain illegal or corrupted data. In general, data is said to be encapsulated if it can only be accessed by a controlled set of functions.

Representation of Data in a Contiguous Stack



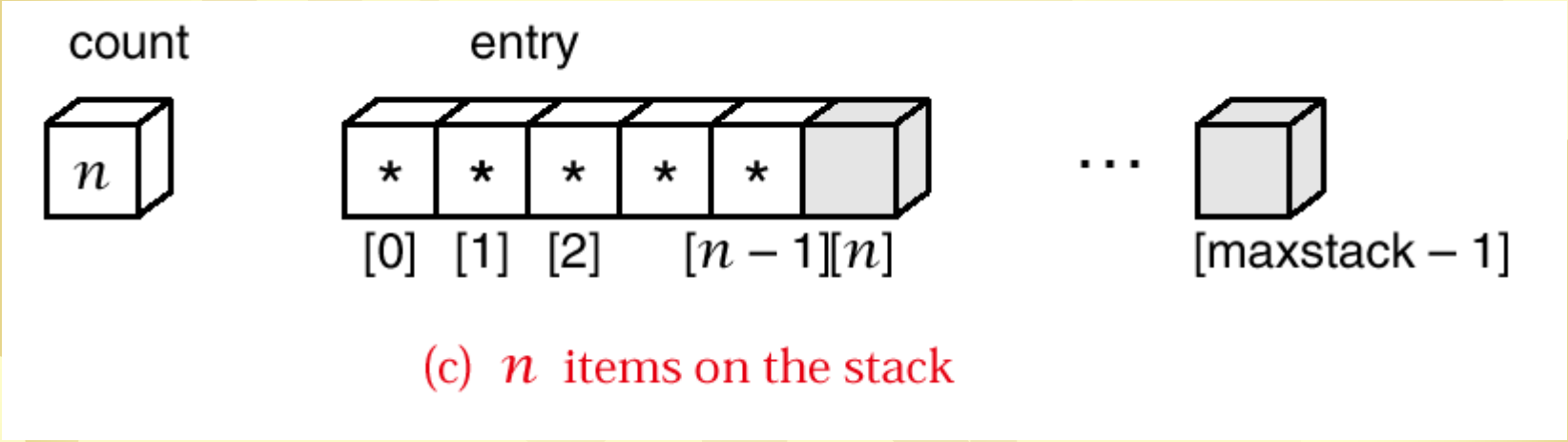
(a) Stack is empty.

Representation of Data in a Contiguous Stack



(b) Push the first entry.

Representation of Data in a Contiguous Stack



(c) n items on the stack

Stack Methods

- Error_code Stack ::push(const Stack entry &item)

```
/* Pre: None.
Post: If the Stack is not full, item is added to the top of
the Stack . If the Stack is full, an Error_code of overflow
is returned and the Stack is left unchanged. */
{
    Error_code outcome = success;
    if (count >= maxstack)
        outcome = overflow;
    else
        entry[count ++] = item;
    return outcome;
}
```

Stack Methods

- Error_code Stack ::pop( )

```
/* Pre: None.
Post: If the Stack is not empty, the top of the Stack is
removed. If the Stack is empty, an Error_code of
underflow is returned. */
{
    Error_code outcome = success;
    if (count == 0)
        outcome = underflow;
    else  -- count;
    return outcome;
}
```

Further Stack Methods

- Error_code Stack ::top(Stack entry &item) const

```
/* Pre: None.
Post: If the Stack is not empty, the top of the Stack is
returned in item . If the Stack is empty an Error_code of
underflow is returned. */
{
    Error_code outcome = success;
    if (count == 0)
        outcome = underflow;
    else
        item = entry[count - 1];
    return outcome;
}
```

Further Stack Methods

- bool Stack ::empty( ) const

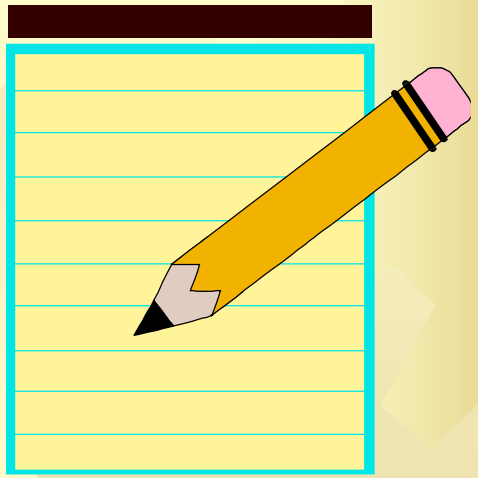
```
/* Pre: None.
Post: If the Stack is empty, true is returned. Otherwise
false is returned. */
{
    bool outcome = true;
    if (count > 0) outcome = false;
    return outcome;
}
```

Further Stack Methods

- Stack ::Stack( )

```
/* Pre: None.
Post: The stack is initialized to be empty. */
{
    count = 0;
}
```

Questions?



Application1: Reversing a List

```
#include <stack>

int main( )
/* Pre: The user supplies an integer n and n decimal
        numbers.
   Post: The numbers are printed in reverse order.
   Uses: The STL class stack and its methods */
{
   int n;
   double item;
   stack<double> numbers;
   //declares and initializes a stack of numbers
```

Application1: Reversing a List

```
cout << " Type in an integer n followed by n
             decimal numbers."
     << endl
     << " The numbers will be printed in reverse
             order."
     << endl;
cin >> n;
for (int i = 0; i < n; i ++ ) {
     cin >> item;
     numbers.push(item);
}
```

Application1: Reversing a List

```
    cout << endl << endl;
    while (!numbers.empty( )) {
        cout << numbers.top( ) << " " ;
        numbers.pop( );
    }
    cout << endl;
}
```

Application2: Reverse Polish Calculator

- Infix notation
  - A / B – C + D * E – A * C
  - (A / B) – (C + D) * (E – A) * C
- Postfix notation
  - A B / C – D E * + A C * -
  - A B / CD + E A - * C * -
- Priority in C/C++

priority	1	2	3	4	5	6	7
oper	单目 –, !	*, /, %	+, –	<, <=, >, >=	==, !=	&&	||

- isp 叫做栈内 (in stack priority) 优先数
- icp 叫做栈外 (in coming priority) 优先数。
- 操作符优先数相等的情况只出现在括号配对或栈底的“#”号与输入流最后的“#”号配对时。
- 在把中缀表达式转换为后缀表达式的过程中，需要检查算术运算符的优先级，以实现运算规则。

中缀表达式转换为后缀表达式的算法

- 操作符栈初始化，将结束符 ‘#’ 进栈。然后读入中缀表达式字符流的首字符ch。
- 重复执行以下步骤，直到 ch = ‘#’，同时栈顶的操作符也是‘#’，停止循环。
  a) 若ch是操作数，直接输出，读入下一个字符ch。
  b) 若 ch 是操作符，判断 ch 的优先级 icp 和位于栈顶的操作符op的优先级 isp：
     - 若 icp (ch) > isp (op)，令ch进栈，读入下一个字符ch。（看后面是否有更高的）

- 若 icp (ch) < isp (op)，退栈并输出。
  （执行先前保存在栈内的优先级高的操作符）
- 若 icp (ch) == isp (op)，退栈但不输出，若退出的是“(”号读入下一个字符ch。（销括号）

- 算法结束，输出序列即为所需的后缀表达式。
- 举例，将中缀表达式

  a + b * (c – d) – e / f #

  转换为后缀表达式

  a b c d – * + e f / –

步	输入	栈内容	语义	输出	动作
1		#			栈初始化
2	a	#		a	操作数 a 输出，读字符
3	+	#	+>#		操作符+进栈，读字符
4	b	#+		b	操作数 b 输出，读字符
5	*	#+	*>+		操作符*进栈，读字符
6	(	#+*	(>*		操作符(进栈，读字符
7	c	#+*(		c	操作数 c 输出，读字符
8	–	#+*(	–>(		操作符–进栈，读字符
9	d	#+*(–		d	操作数 d 输出，读字符
10	)	#+*(–	)<–	–	操作符–退栈输出
11		#+*(	)=(		(退栈，消括号，读字符

步	输入	栈内容	语义	输出	动作
12	–	#+*	–<*	*	操作符*退栈输出
13		#+	–<+	+	操作符+退栈输出
14		#	–>#		操作符–栈，读字符
15	e	#–		e	操作数 e 输出，读字符
16	/	#–	/>–		操作符/进栈，读字符
17	f	#–/		f	操作数 f 输出，读字符
18	#	#–/	#</	/	操作符/退栈输出
19		#–	#<–	–	操作符–退栈输出
20		#	#=#		#配对，转换结束

Application2: Reverse Polish Calculator

- In a Reverse Polish calculator, the operands (numbers, usually) are entered before an operation (like addition, subtraction, multiplication, or division) is specified. The operands are pushed onto a stack. When an operation is performed, it pops its operands from the stack and pushes its result back onto the stack.

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

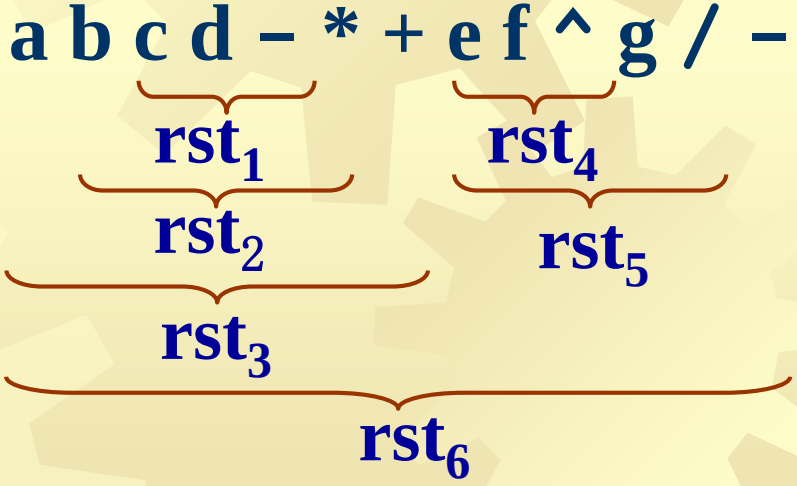
Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

A * (B + C) * D

Step	Symbol	Stack	Output
0	NULL	NULL	NULL
1	A	NULL	A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10	NULL	NULL	ABC+*D*

应用3：后缀表示计算表达式的值

- 从左向右顺序地扫描表达式，并用一个栈暂存扫描到的操作数或计算结果。
- 在后缀表达式的计算顺序中已隐含了加括号的优先次序，括号在后缀表达式中不出现。
- 计算例 a b c d – * + e f ^ g / –



通过后缀表示计算表达式值的过程

- 顺序扫描表达式的每一项，根据它的类型做如下相应操作：
  a) 若该项是操作数，则将其压栈；
  b) 若该项是操作符<op>，则连续从栈中退出两个操作数Y和X，形成运算指令X<op>Y，并将计算结果重新压栈。
- 当表达式的所有项都扫描并处理完后，栈顶存放的就是最后的计算结果。

计算a b c d – * + e f ^ g / –

步	输入	类 型	动 作	栈内容
1			置空栈	空
2	a	操作数	进栈	a
3	b	操作数	进栈	a b
4	c	操作数	进栈	a b c
5	d	操作数	进栈	a b c d
6	–	操作符	d、c 退栈，计算 c–d，结果 r1 进栈	a b r1
7	*	操作符	r1、b 退栈，计算 b*r1，结果 r2 进栈	a r2
8	+	操作符	r2、a 退栈，计算 a+r2，结果 r3 进栈	r3

步	输入	类 型	动 作	栈内容
9	e	操作数	进栈	r3 e
10	f	操作数	进栈	r3 e f
11	^	操作符	f、e 退栈，计算 e^f，结果 r4 进栈	r3 r4
12	g	操作数	进栈	r3 r4 g
13	/	操作符	g、r4 退栈，计算 r4/g，结果 r5 进栈	r3 r5
14	−	操作符	r5、r3 退栈，计算 r3−r5，结果 r6 进栈	r6

Application4: Bracket Matching

- We develop a program to check that brackets are correctly matched in an input text file.
- We limit our attention to the brackets {, }, (,), [, and].
- We read a single line of characters and ignore all input other than bracket characters.

Algorithm

- Read the file character by character. Each opening bracket (, [, or { that is encountered is considered as unmatched and is stored until a matching bracket can be found.

- Any closing bracket),], or } must correspond, in bracket style, to the last unmatched opening bracket, which should now be retrieved and removed from storage. Finally, at the end of the program, we must check that no unmatched opening brackets are left over.

Algorithm

- A program to test the matching of brackets needs to process its input file character by character, and, as it works its way through the input, it needs some way to remember any currently unmatched brackets.

- These brackets must be retrieved in the exact reverse of their input order, and therefore a Stack provides an attractive option for their storage.

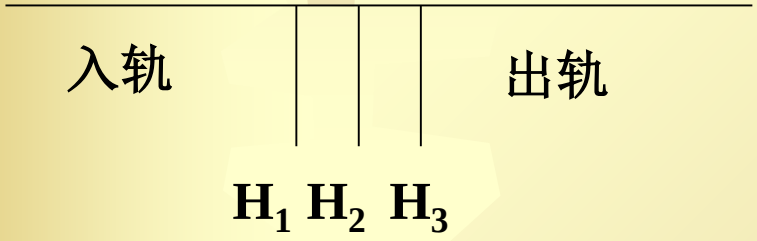
Algorithm

- Our program need only loop over the input characters, until either a bracketing error is detected or the input file ends. Whenever a bracket is found, an appropriate Stack operation is applied.

栈的应用5

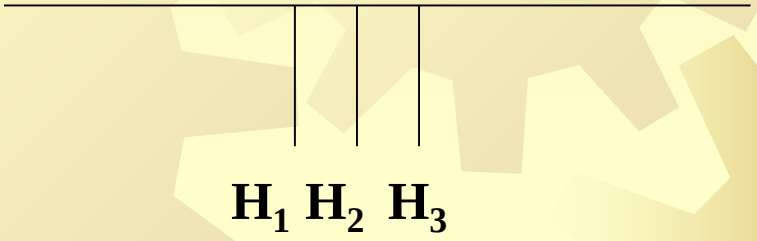
- 例：一列货运列车共有n节车厢，每节车厢将从列车上卸下来停放在不同的车站。假定n个车站的编号分别为1，…，n。货运列车按照第n站至第1站的次序经过这些车站。车厢的编号与它们的目的地相同。为了便于从列车上卸掉相应的车厢，必须事先排好这些车厢，使各车厢从前至后按编号1到n的次序排列。我们在一个转轨站里完成车厢的排列工作。在转轨站中有一个入轨，一个出轨和k个缓冲铁轨（位于入轨和出轨之间）。

[581742963]



(a)

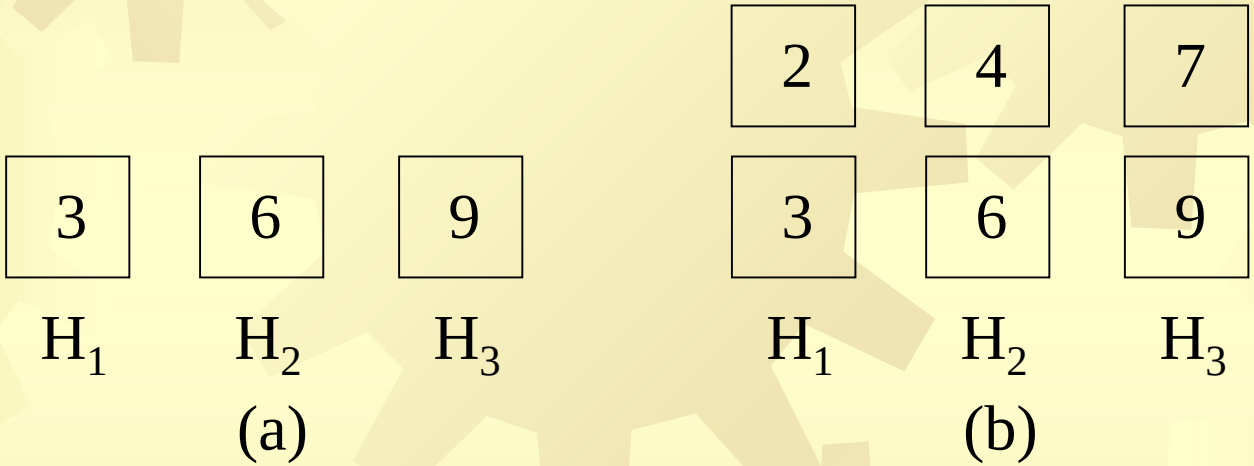
[987654321]



(b)

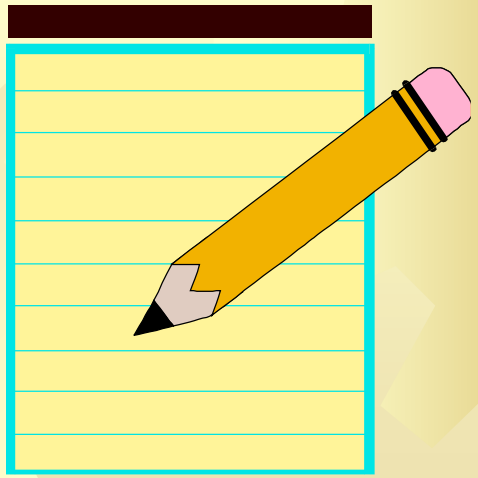
具有三个缓冲铁轨的转轨站

- 为了重排车厢，需从前至后依次检查入轨处的所有车厢，如果正在检查的车厢就是下一个满足排列要求的车厢，可以直接把它放到出轨上去；如果不是，则把它移动到缓冲铁轨上去，直到按输出次序要求轮到它时才将它放到出轨上。缓冲铁轨是按照后进先出（LIFO）的方式使用的，因为车厢的进和出都是在缓冲铁轨的顶部进行的。
- 在重排车厢的过程中，仅允许以下移动：
  - 车厢可以从入轨的前部（即右端）移动到一个缓冲铁轨的顶部或出轨的左端；
  - 车厢可以从一个缓冲铁轨的顶部移动到出轨的左端。



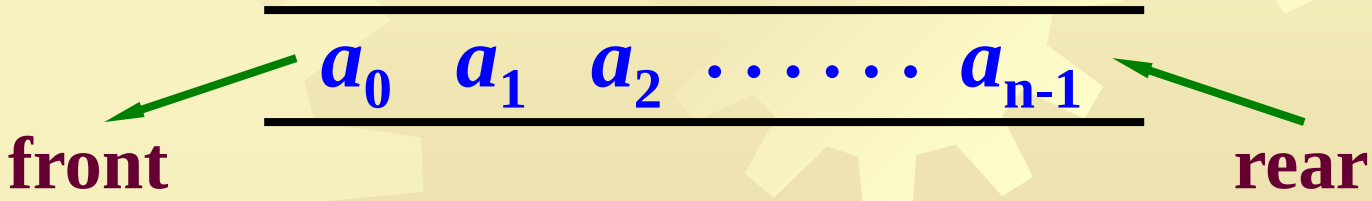
缓冲铁轨中间状态

Questions?



Queue

- FIFO: First In, First Out
- Restricted form of list:
  - Insert at one end, remove from other.
- Notation:
  - Insert: Enqueue
  - Delete: Dequeue
  - First element: FRONT
  - Last element: REAR



Queue

- DEFINITION: A queue of elements of type T is a finite sequence of elements of T together with the following operations:
  - *Create* the queue, leaving it empty.
  - Test whether the queue is *Empty*.
  - *Append* a new entry onto the rear of the queue, provided the queue is not full.
  - *Serve* (and remove) the entry from the front of the queue, provided the queue is not empty.
  - *Retrieve* the front entry off the queue, provided the queue is not empty.

Queue

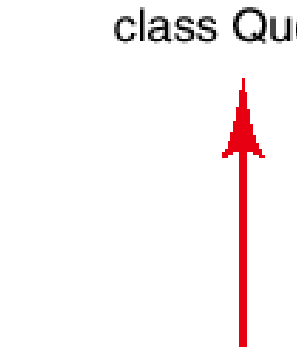
- Queue ::Queue( );
  - Post: The Queue has been created and is initialized to be empty.
- Error_code Queue ::append(const Queue entry &x);
  - Post: If there is space, x is added to the Queue as its rear. Otherwise an Error_code of overflow is returned.
- Error_code Queue ::serve( );
  - Post: If the Queue is not empty, the front of the Queue has been removed. Otherwise an Error_code of underflow is returned.

Queue

- Error_code Queue ::retrieve(Queue entry &x) const;
  - Post: If the Queue is not empty, the front of the Queue has been recorded as x. Otherwise an Error_code of underflow is returned.

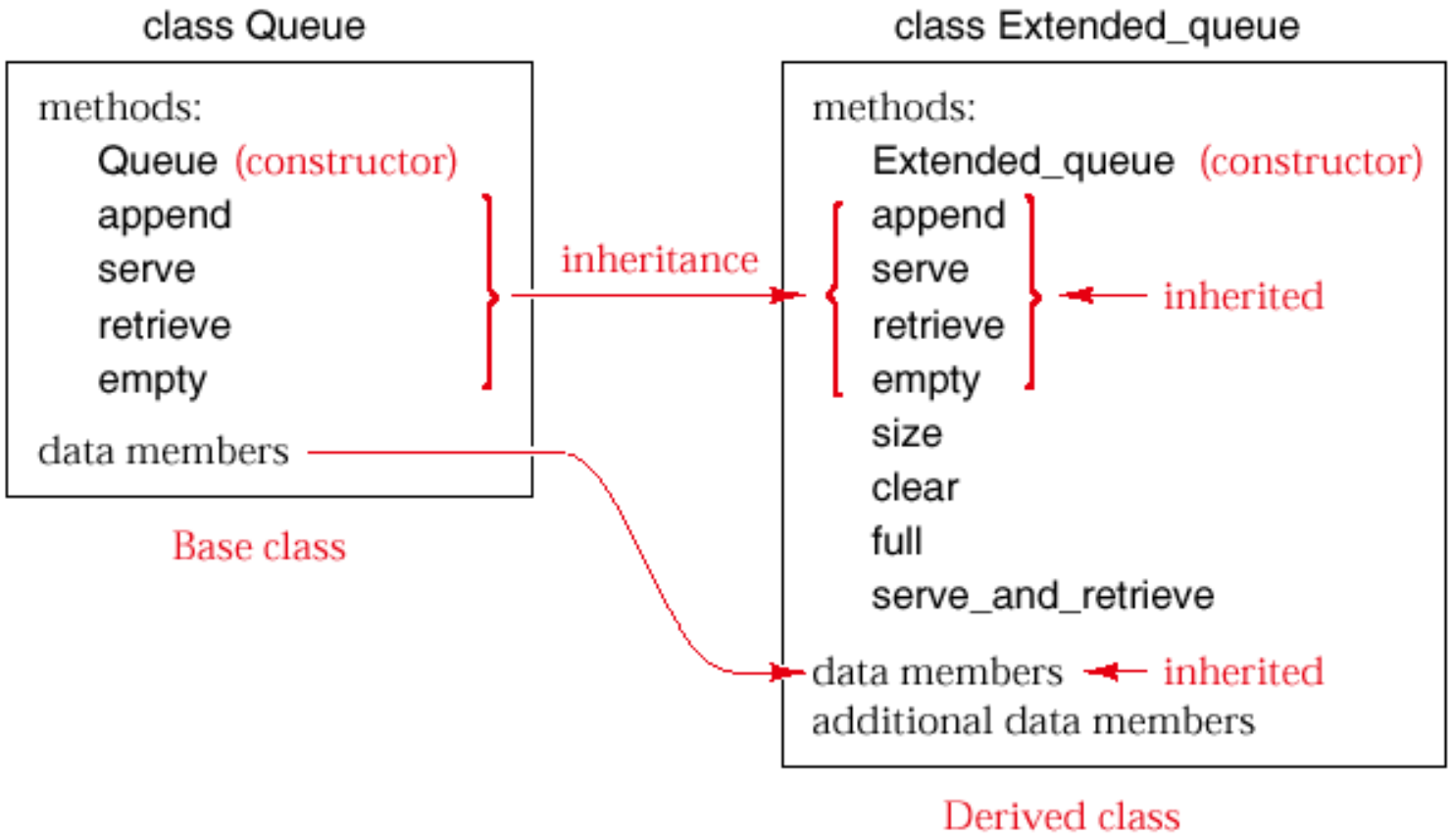
- bool Queue ::empty( ) const;
  - Post: Return true if the Queue is empty, otherwise return false.

Hierarchy diagram



(a) Hierarchy diagram

Inheritance



(b) Derived class **Extended_queue** from base class **Queue**

Extended_queue

- DEFINITION: An extended_queue of elements of type T is a queue of elements of T together with the following additional operations:
  - Determine whether the queue is *full* or not.
  - Find the *size* of the queue.
  - *Serve and retrieve* the front entry in the queue, provided the queue is not empty.
  - *Clear* the queue to make it empty.

Extended_queue

- bool Extended_queue ::full( ) const;
  - Post: Return true if the Extended_queue is full; return false otherwise.
- void Extended_queue ::clear( );
  - Post: All entries in the Extended_queue have been removed; it is now empty.

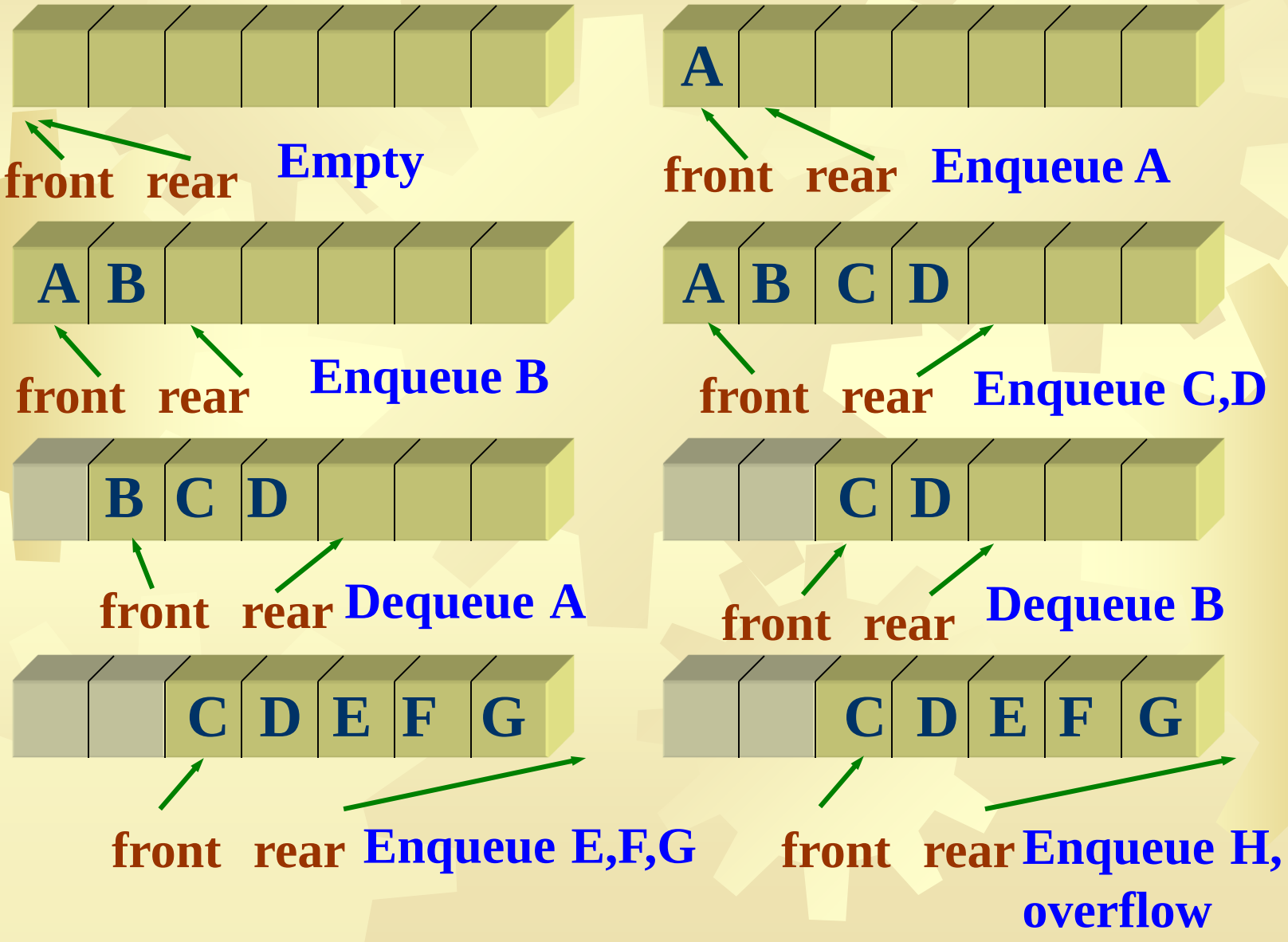
Extended_queue

- int Extended_queue ::size( ) const;
  - Post: Return the number of entries in the Extended_queue.
- Error_code Extended_queue :: serve_and_retrieve(Queue entry &item);
  - Post: Return underflow if the Extended_queue is empty. Otherwise remove and copy the item at the front of the Extended_queue to item and return success.

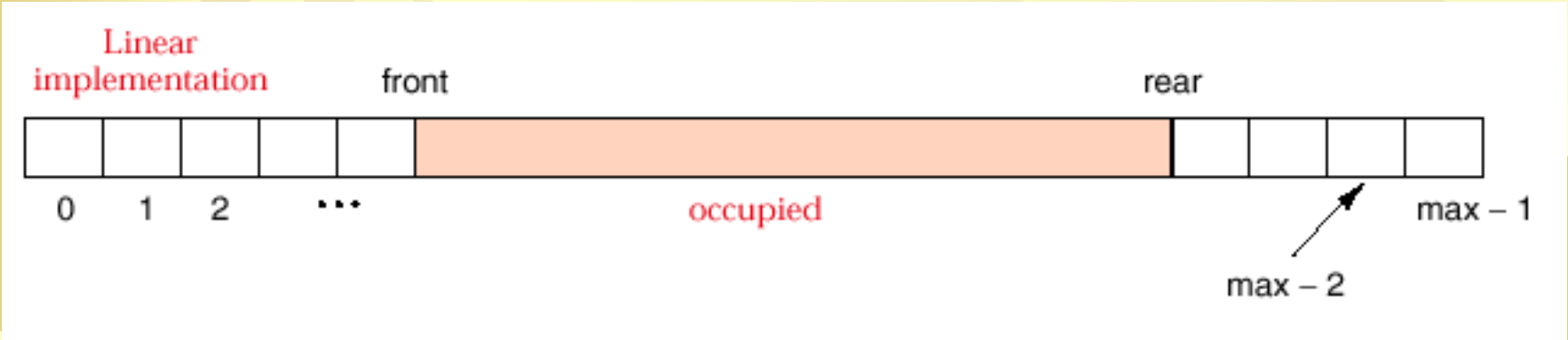
Implementations of Queues

- The physical model: a linear array with the front always in the first position and all entries moved up the array whenever the front is deleted.
- A linear array with two indices always increasing (front and rear).

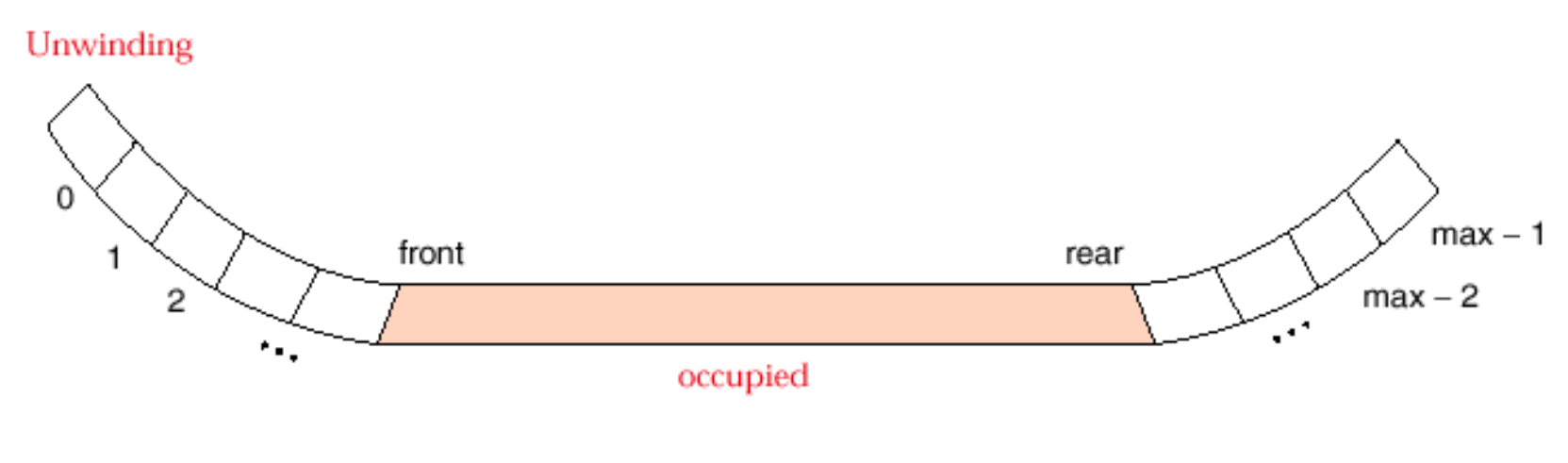
Enqueue/Dequeue



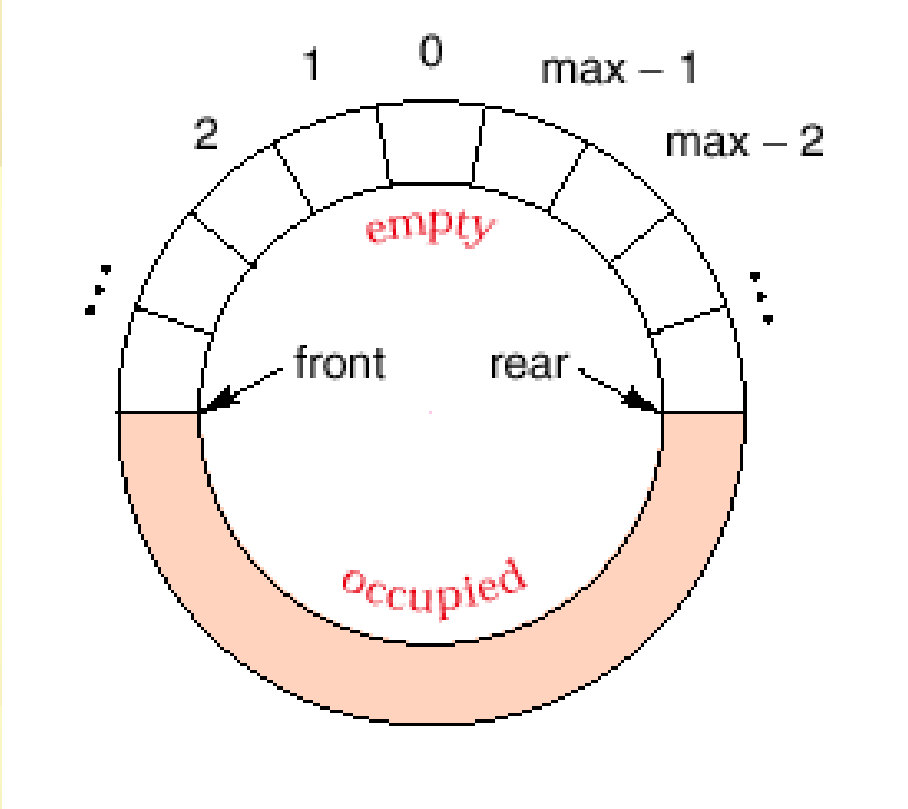
Linear Implementation of Queues

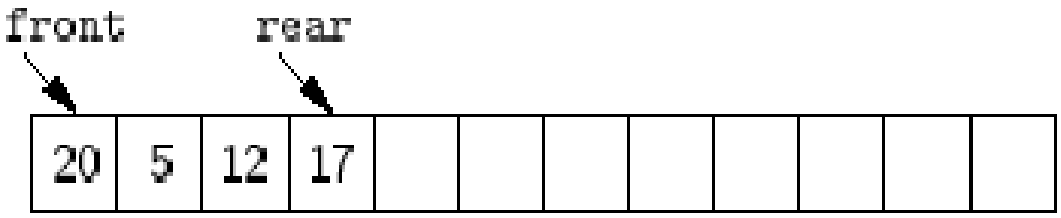


Winding

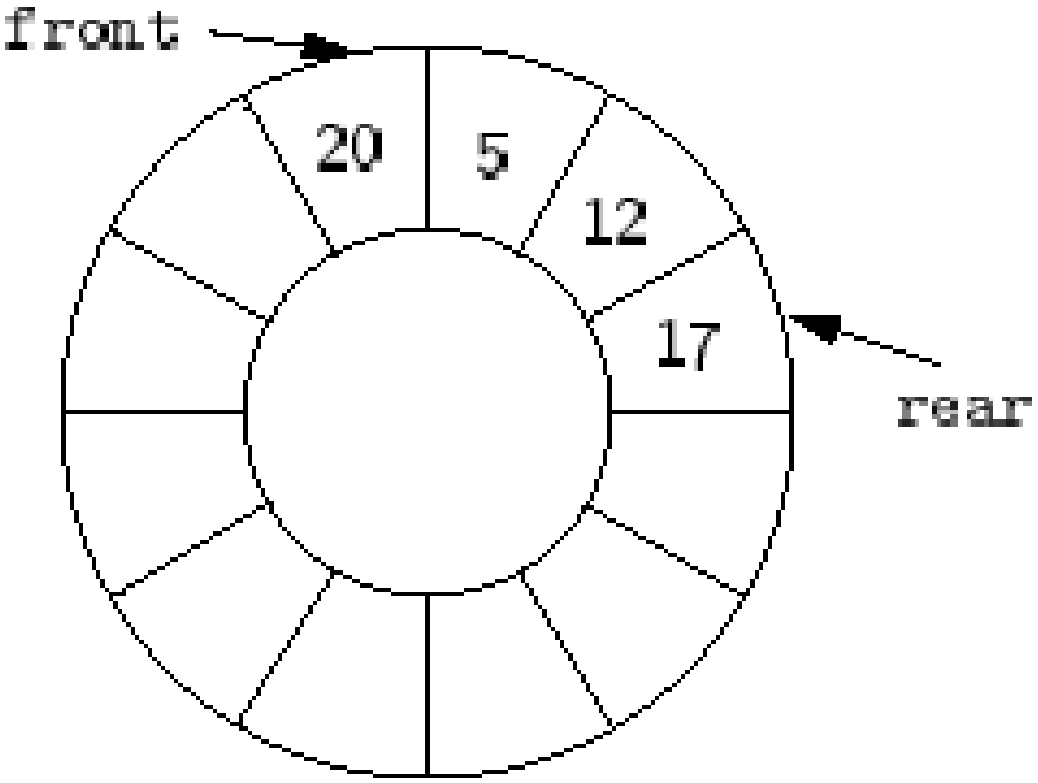


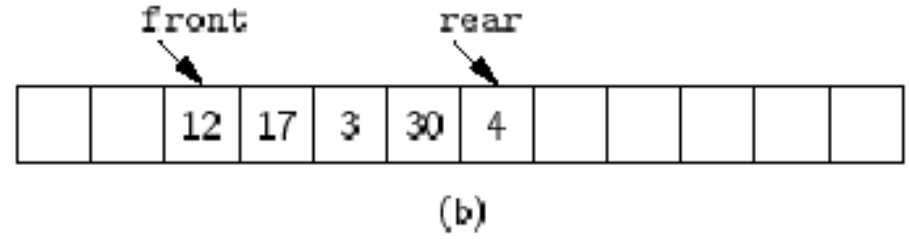
Circular Implementation of Queues



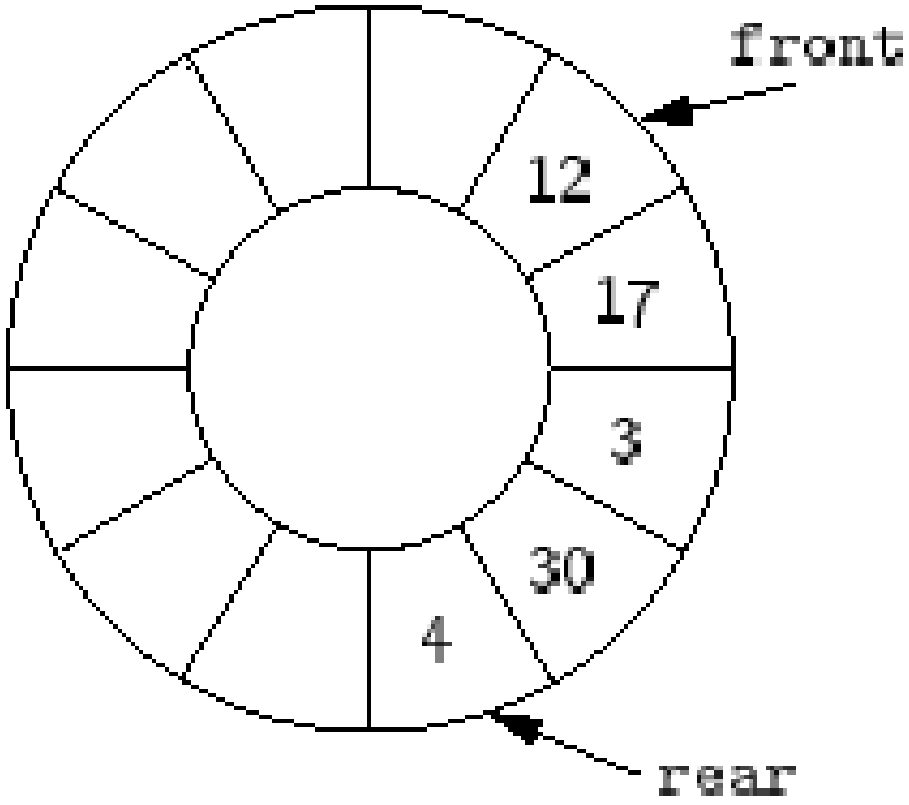


(a)





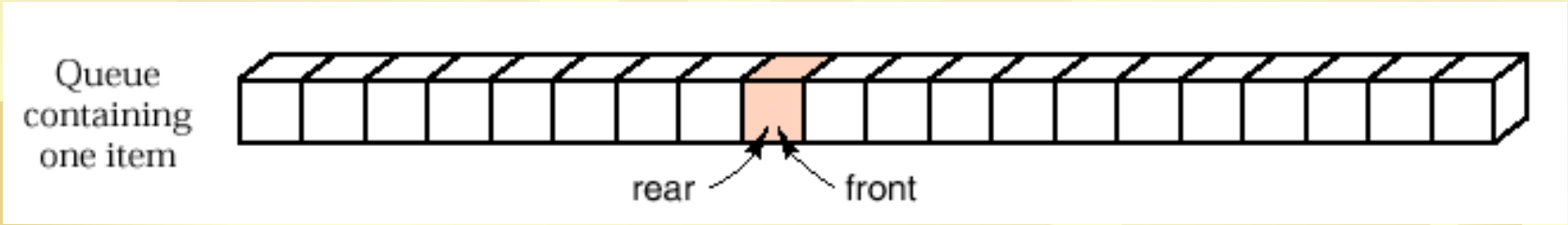
(b)



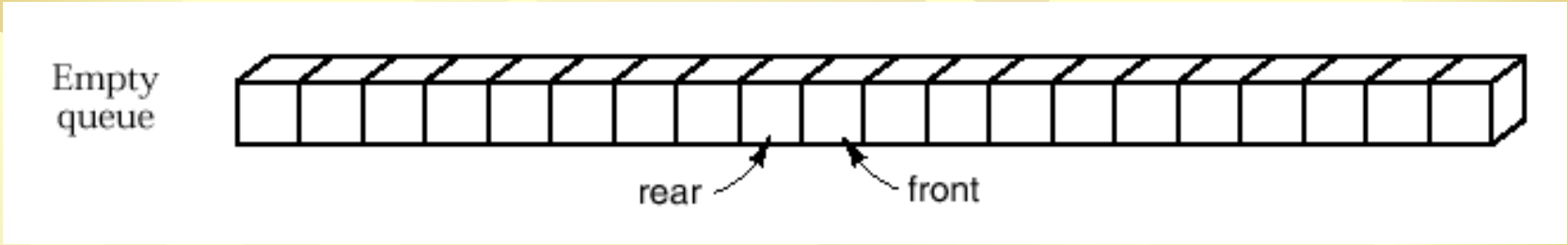
Circular arrays in C++

- Equivalent methods to increment an index i in a circular array:
  - i = ((i + 1) == max) ? 0 : (i + 1);
  - if ((i + 1) == max) i = 0; else i = i + 1;
  - i = (i + 1) % max;

Boundary Conditions

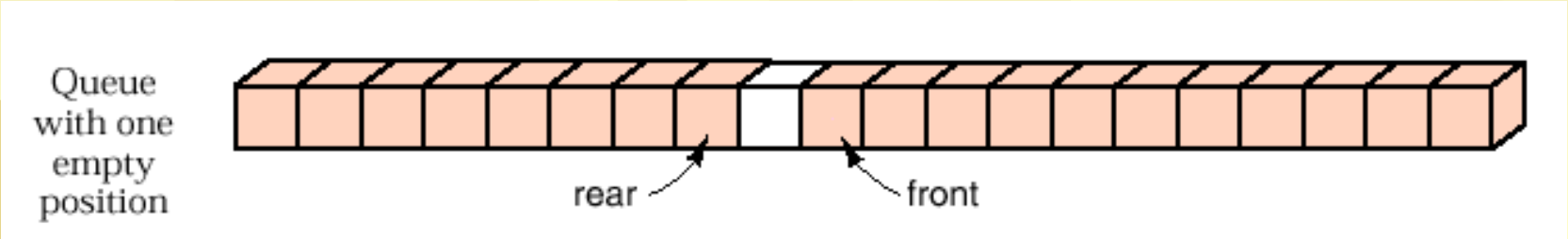


Remove the item

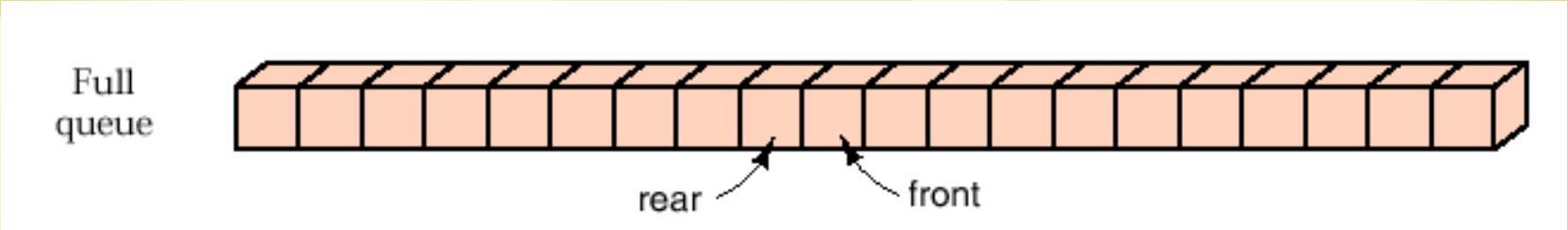


rear + 1 = front (emptiness)

Boundary Conditions



Insert an item



rear + 1 = front (fullness)

Circular Implementation of Queues

- A circular array with front and rear indices and one position left vacant.
  - A circular array with front and rear indices and a Boolean flag to indicate fullness (or emptiness).
  - A circular array with front and rear indices and an integer counter of entries.
  - A circular array with front and rear indices taking special values to indicate emptiness.
    - Leave one slot empty, or use a separate flag

Class definition

- const int maxqueue = 10; //small value for testing

```
class Queue {
public:
        Queue( );
        bool empty( ) const;
        Error_code serve( );
        Error_code append(const Queue entry &item);
        Error_code retrieve(Queue entry &item) const;
protected:
        int count;
        int front, rear;
        Queue entry entry[maxqueue];
};
```

Initialization:

- Queue ::Queue( )

```
/* Post: The Queue is initialized to be empty. */
{
        count = 0;
        rear = maxqueue - 1;
        front = 0;
}
```

- bool Queue ::empty( ) const

```
/* Post: Return true if the Queue is empty, otherwise return
false . */
{
        return count == 0;
}
```

Basic Queue Methods

- Error_code Queue ::append(const Queue entry &item)
  /* Post: item is added to the rear of the Queue . If the Queue is full return an Error_code of overflow and leave the Queue unchanged. */
  {
  ```
  if (count >= maxqueue) return overflow;
  count ++ ;
  rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
  entry[rear] = item;
  return success;
  ```
  }

Basic Queue Methods

- Error_code Queue ::serve( )
  /* Post: The front of the Queue is removed. If the Queue is empty return an Error_code of underflow. */

```
{
    if (count <= 0) return underflow;
    count -- ;
    front = ((front + 1) == maxqueue) ? 0 : (front
                                                + 1);
    return success;
}
```

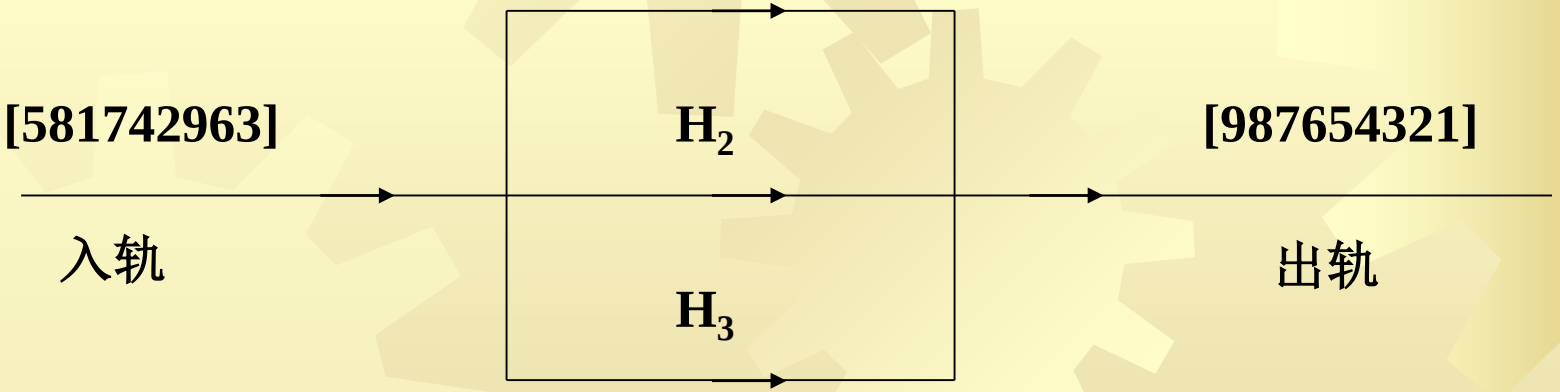
Basic Queue Methods

- Error_code Queue ::retrieve(Queue entry &item) const

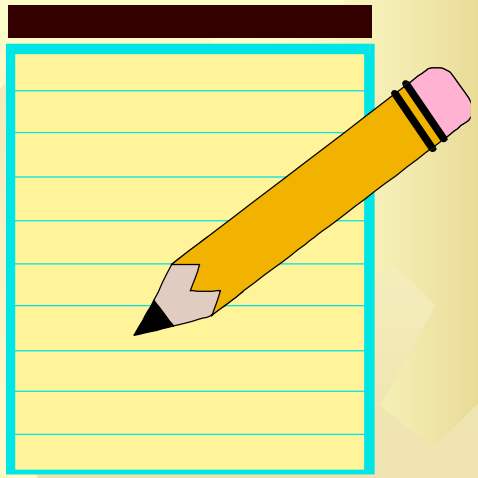
```
/* Post: The front of the Queue retrieved to the output parameter item . If the Queue is empty return an Error_code of underflow. */
{
    if (count <= 0) return underflow;
    item = entry[front];
    return success;
}
```

队列的应用

- 例：重新考虑火车车厢重排问题，不过这次假设缓冲铁轨位于入轨和出轨之间，由于这时缓冲铁轨均按先进先出（FIFO）的方式工作，因此可将它们视为队列。我们可以采用链式队列描述车厢重排算法。使用队列来重排车厢的实现。

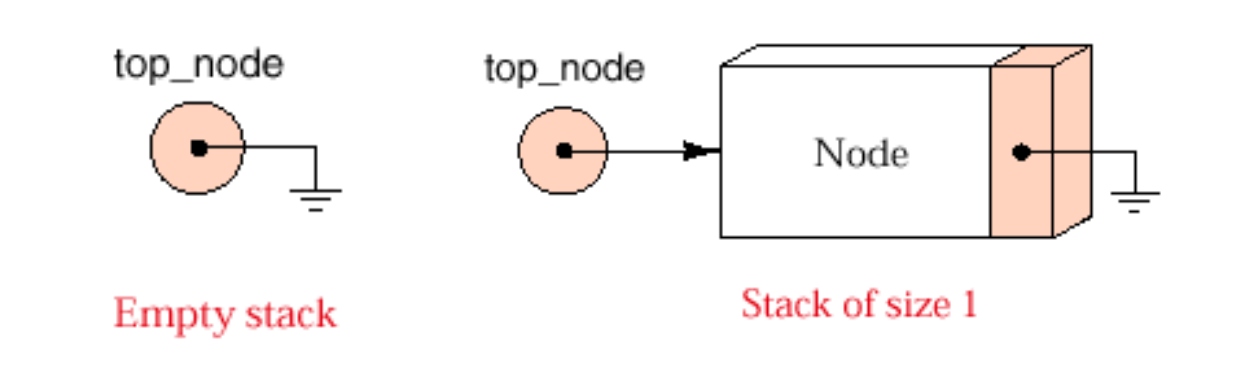


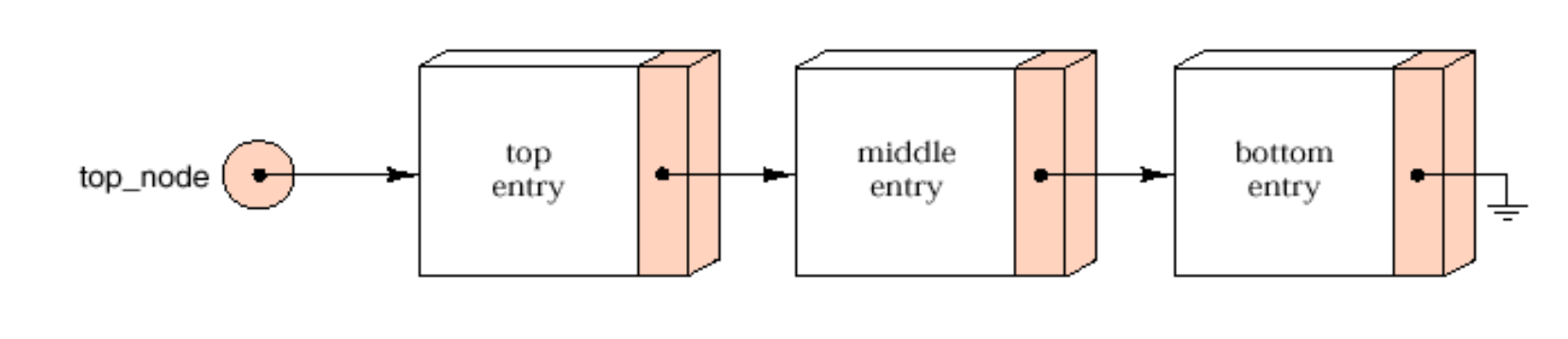
Questions?



Linked Stacks

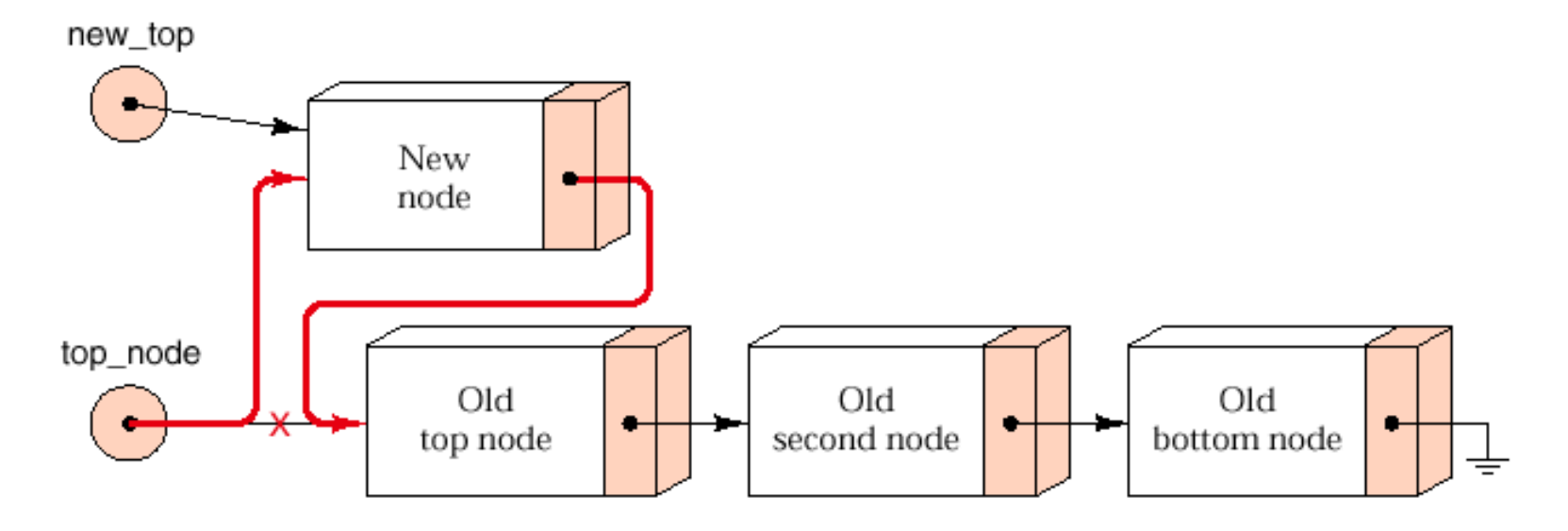
- Linked Stack





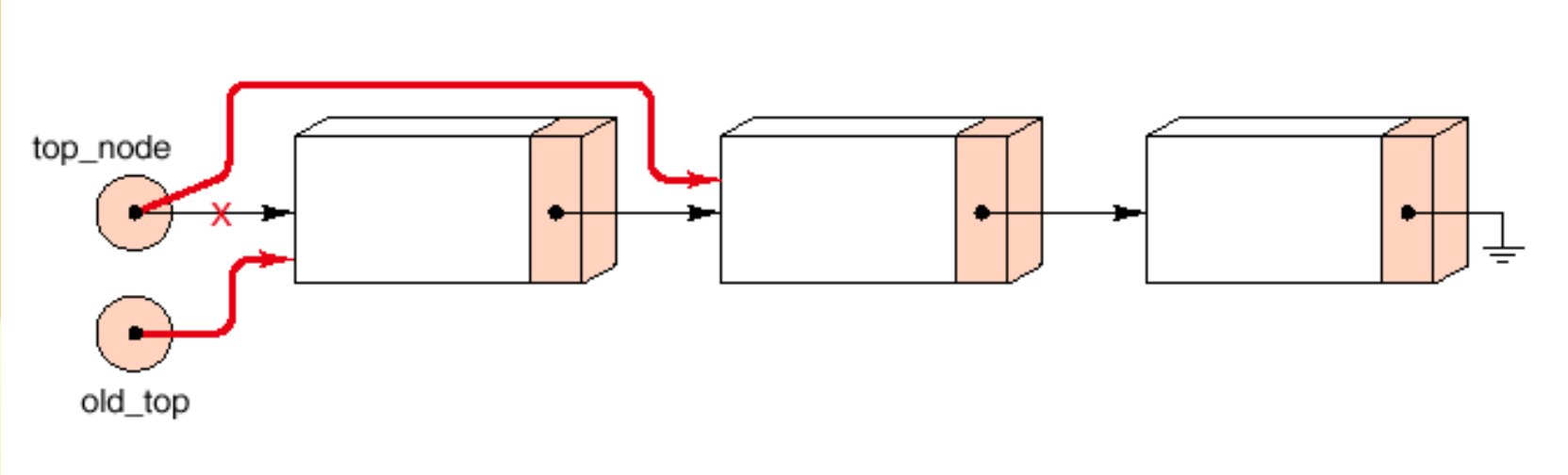
Linked Stacks

- Insert



Linked Stacks

- Remove



Class Declaration for Linked Stack

```
class Stack {
public:
    Stack( );
    bool empty( ) const;
    Error_code push(const Stack entry &item);
    Error_code pop( );
    Error_code top(Stack entry &item) const;
protected:
    Node *top_node;
};
```

Methods

- Error_code Stack ::push(const Stack entry &item)

```
/* Post: Stack entry item is added to the top of the Stack ;
returns success or returns a code of overflow if dynamic
memory is exhausted. */
{
    Node *new_top = new Node(item, top_node);
    if (new_top == NULL) return overflow;
    top_node = new_top;
    return success;
}
```

Methods

- Error_code Stack ::pop( )

```
/* Post: The top of the Stack is removed. If the Stack is
empty the method returns underflow; otherwise it returns
success . */
{
    Node *old_top = top_node;
    if (top_node == NULL) return underflow;
    top_node = old_top->next;
    delete old_top;
    return success;
}
```

Linked Stacks with Safeguards

- Code:
  - ```
    for (int i = 0; i < 1000000; i ++ ) {
      Stack small;
      small.push(some_data);
    }
    ```

- Suppose that the linked Stack implementation is used. As soon as the object small goes out of scope, the data stored in small becomes garbage.
    ```

Linked Stacks with Safeguards

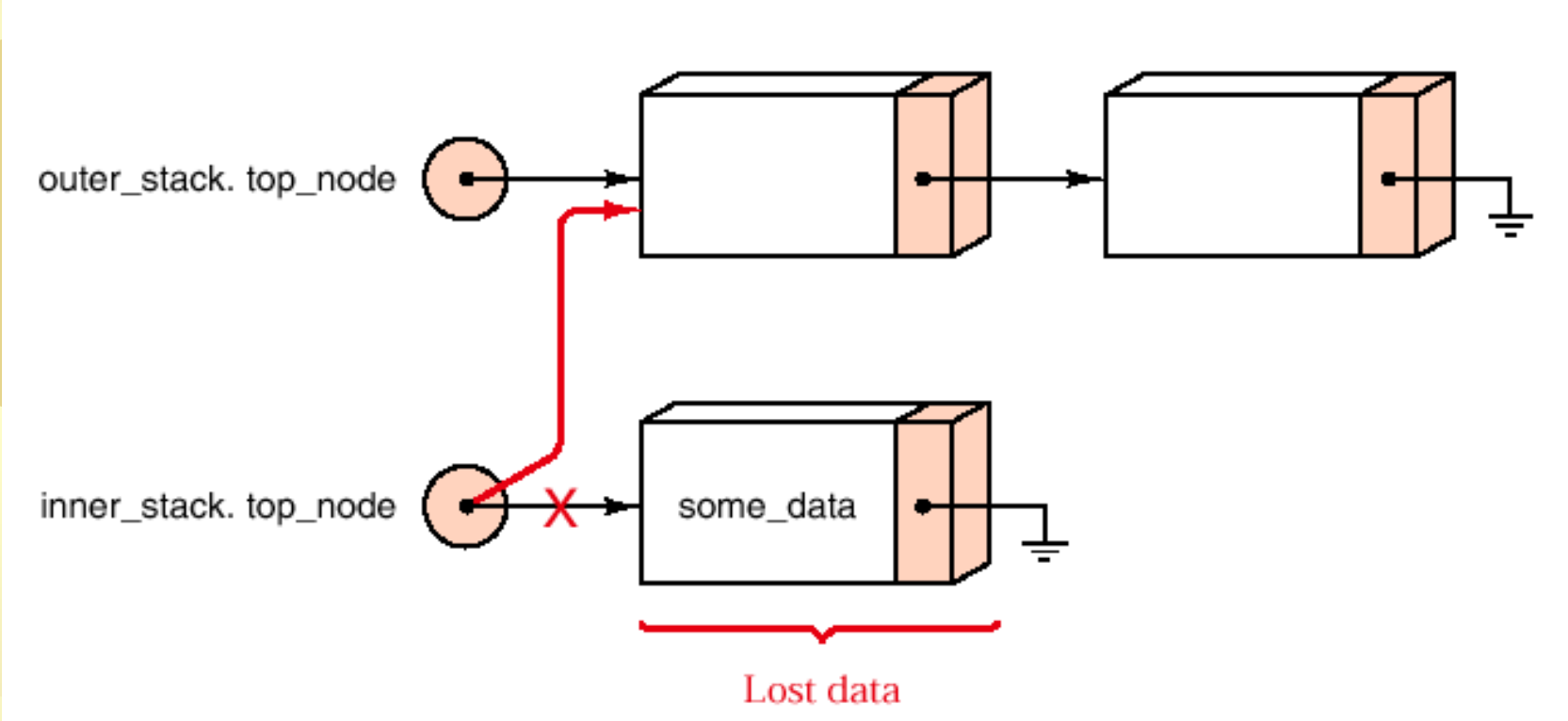
- Destructors are often used to delete dynamically allocated objects that would otherwise become garbage.

- Every linked structure should be equipped with a destructor to clear its objects before they go out of scope.

Linked Stacks with Safeguards

- Dangers in Assignment
  - Stack outer_stack;
    for (int i = 0; i < 1000000; i ++) {
      Stack inner_stack;
      inner_stack.push(some data);
      inner_stack = outer_stack;
    }

Dangers in Assignment



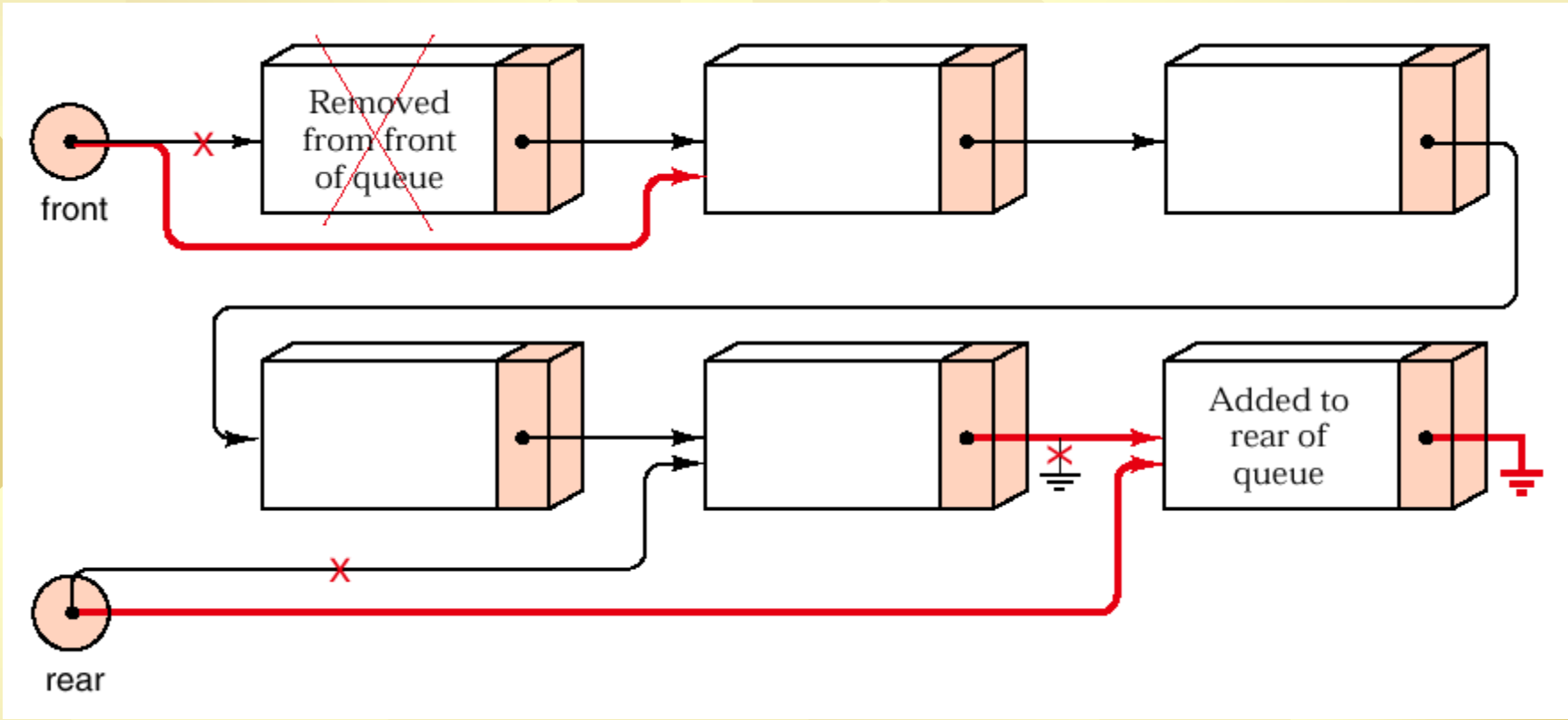
Misbehaviors:

- Lost data space.
- Two stacks have shared nodes.
- The destructor on inner stack deletes outer stack.
- Such a deletion leaves the pointer outer stack.top node addressing what a random memory location.

Class declaration for Linked Queues

```
class Queue {
public:       // standard Queue methods
        Queue( );
        bool empty( ) const;
        Error_code append(const Queue entry &item);
        Error_code serve( );
        Error_code retrieve(Queue entry &item) const;
    // safety features for linked structures
        ~Queue( );
        Queue(const Queue &original);
        void operator = (const Queue &original);
protected:
        Node *front, *rear;
};
```

Insert and Remove



Methods

- Error_code Queue ::append(const Queue entry &item)
  /* Post: Add item to the rear of the Queue and return a code of success or return a code of overflow if dynamic memory is exhausted. */

```
{
        Node *new_rear = new Node(item);
        if (new_rear == NULL) return overflow;
        if (rear == NULL) front = rear = new_rear;
        else {
                rear->next = new_rear;
                rear = new_rear;
        }
        return success;
}
```

Methods

- Error_code Queue ::serve( )

```
/* Post: The front of the Queue is removed. If the Queue
is empty, return an Error_code of underflow. */
{
    if (front == NULL) return underflow;
    Node *old_front = front;
    front = old_front->next;
    if (front == NULL) rear = NULL;
    delete old_front;
    return success;
}
```

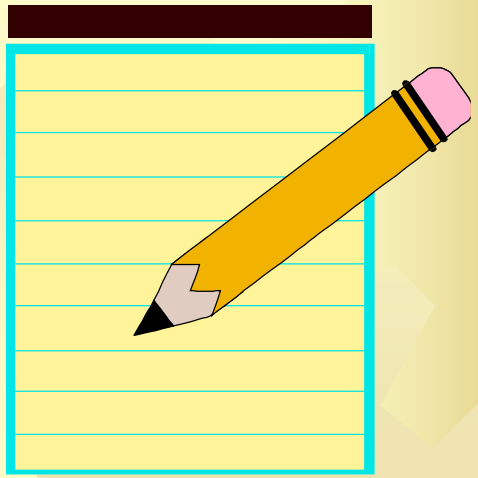
Extended_queue

- class Extended_queue: public Queue {
  public:
    bool full( ) const;
    int size( ) const;
    void clear( );
    Error_code serve and retrieve(Queue
                entry &item);
  };

Extended_queue

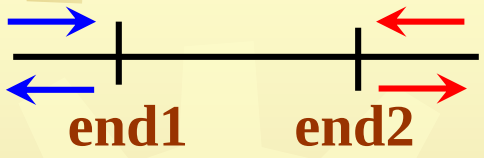
- int Extended_queue ::size( ) const
  /* Post: Return the number of entries in the Extended_queue . */
  {
  Node *window = front;
  int count = 0;
  while (window != NULL) {
  window = window->next;
  count ++ ;
  }
  return count;
  }

Questions?



双端队列

- 双端队列（Deque）是对队列的扩展，它允许在队列的两端进行插入和删除。双端队列英文的全称是Double-ended queue。
- 我们可以把双端队列视为底靠底的双栈，但它们相通，成为双向队列。两端都可能是队头和队尾。



双端队列

输入受限的双端队列

输出受限的双端队列

- 一般地，若有 n 个元素，进入双端队列的顺序是1, 2, …, n（不管是何端进/出），可用数学归纳法证明：全进全出后可能的出队顺序有 $n!$ 种。
- 而普通的先进先出队列的可能的出队顺序仅有 1种。

输入受限的双端队列

- 如果限定只能在双端队列的一端输入，可以在两端输出，那么对于一个确定的输入序列，输出只能有 3 种可能：在同一端输出，相当于栈；或者在另一端输出，相当于队列；或者混合进出。
- 如果1, 2, 3, 4 顺序入队，都在同一端出队（相当于栈）有 14 种出队顺序，除此之外还有 4! – 14 = 10 种可能的出队序列。
- 实际上，不合理的出队序列基本上都是以 4 打头的。当 4 先出队时，1, 2, 3 依次排在队列里，2 夹在中间，它不可能在 1 和 3 之前出队，所以不可能的出队序列只有 4 2 3 1和4 2 1 3。

输出受限的双端队列

- 这种双端队列限定只能在队列的一端输出，但可以在两端输入，对于一个确定的输入序列，输出顺序也有 3 种可能：在同一端输入和输出，相当于栈；或者在一端输入一端输出，相当于队列；或者混合进出。
- 当输入整数是 1, 2, 3, 4 时，同样先排除允许在同一端输入 / 输出的14种情况，不可能的输出序列还是要在以 4 开头的排列中查找。

- 在以 4 开头的排列中，有问题的是 4, 1, 2, 3 / 4, 1, 3, 2 / 4, 3, 1, 2 / 4, 2, 1, 3 / 4, 2, 3, 1。
- 对于4, 1, 2, 3，先从右端输入1, 2, 3，再从左端输入4，即可从左端输出4, 1, 2, 3。
- 对于4, 1, 3, 2，必须最后从左端输入 4，在此之前在队列中需得到1, 3, 2的排列，这是不可能的。
- 对于4, 3, 1, 2，先从右端输入1, 2，再从左端输入3, 4，即可从左端输出4, 3, 1, 2。
- 对于4, 2, 1, 3， 先从左端输入1, 2，再从右端输入3，最后从左端输入4，即可从左端输出4, 2, 1, 3。

- 对于4, 2, 3, 1，同样在 4 输入前，在队列中得不到 2, 3, 1 这种排列。3 不可能夹在 1 和 2 中间，所以这是不可能的输出序列。
- 最后可知，4, 1, 3, 2 和 4, 2, 3, 1 是不可能的出队序列。问题出在 3 不可能在1, 2之间进队。