Stacks（1）

College of Computer Science, CQU

Outline

- Stack ADT
- Array-Based Stack
- Linked Stack
- Comparison of Array-Based and Linked Stacks
- Applications

Stacks

- The stack is a list-like structure in which elements may be inserted or removed from only one end, called the top of the stack.
- All access is restricted to the most recently inserted elements
- Basic operations are push, pop

Stacks

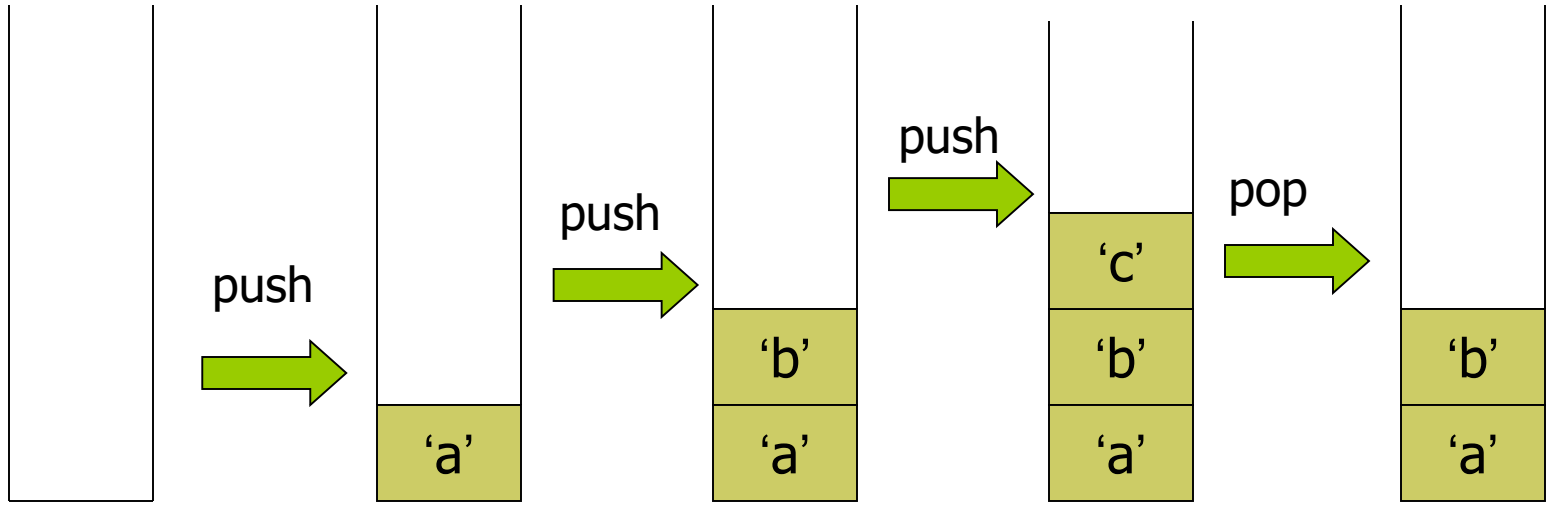
LIFO: Last In, First Out.

Restricted form of list: Insert and remove only at front of list.

Notation:

- Insert: PUSH
- Remove: POP
- The accessible element is called **top element**.

Example: Stack of Char



Stack ADT

```
// Stack abtract class
template <typename E> class Stack {
private:
  void operator =(const Stack&) {} // Protect assignment
  Stack(const Stack&) {} // Protect copy constructor
public:
  Stack() {} // Default constructor
  virtual ~Stack() {} // Base destructor
  // Reinitialize the stack. The user is responsible for
  // reclaiming the storage used by the stack elements.
  virtual void clear() = 0;
```

Stack ADT

```
  // Push an element onto the top of the stack.
  // it: The element being pushed onto the stack.
  virtual void push(const E& it) = 0;
  // Remove the element at the top of the stack.
  // Return: The element at the top of the stack.
  virtual E pop() = 0;
  // Return: A copy of the top element.
  virtual const E& topValue() const = 0;
  // Return: The number of elements in the stack.
  virtual int length() const = 0;
}
```

Array-Based Stack

- The array-based stack implementation is essentially a simplified version of the array-based list.
- The only important design decision to be made is which end of the array should represent the top of the stack.
  - One choice is to make the top be at **position 0** in the array. This implementation is inefficient.（Why？）
  - The other choice is have the top element be at **position n-1** when there are n elements in the stack.
- For the implementation of Figure 4.18, **top** is defined to be the array index of the **first free position** in the stack. Thus, an empty stack has top set to 0, the first available free position in the array.

Array-Based Stack

```
// Array-based stack implementation
Template <typename E> class AStack:public Stack<E>{
private:
  int MaxSize;      // Maximum size of stack
  int top;       // Index for top element
  E   *listArray; // Array holding elements
public:
  AStack(int size =defaultSize) // Constructor
    { maxSize = size; top = 0; listArray = new E[size]; }
  ~AStack() { delete [] listArray; } // Destructor
  void clear() { top = 0; } // Reinitialize
  void push(const E& it) { // Put "it" on stack
    Assert(top != maxSize, "Stack is full");
    listArray[top++] = it;
 }
```

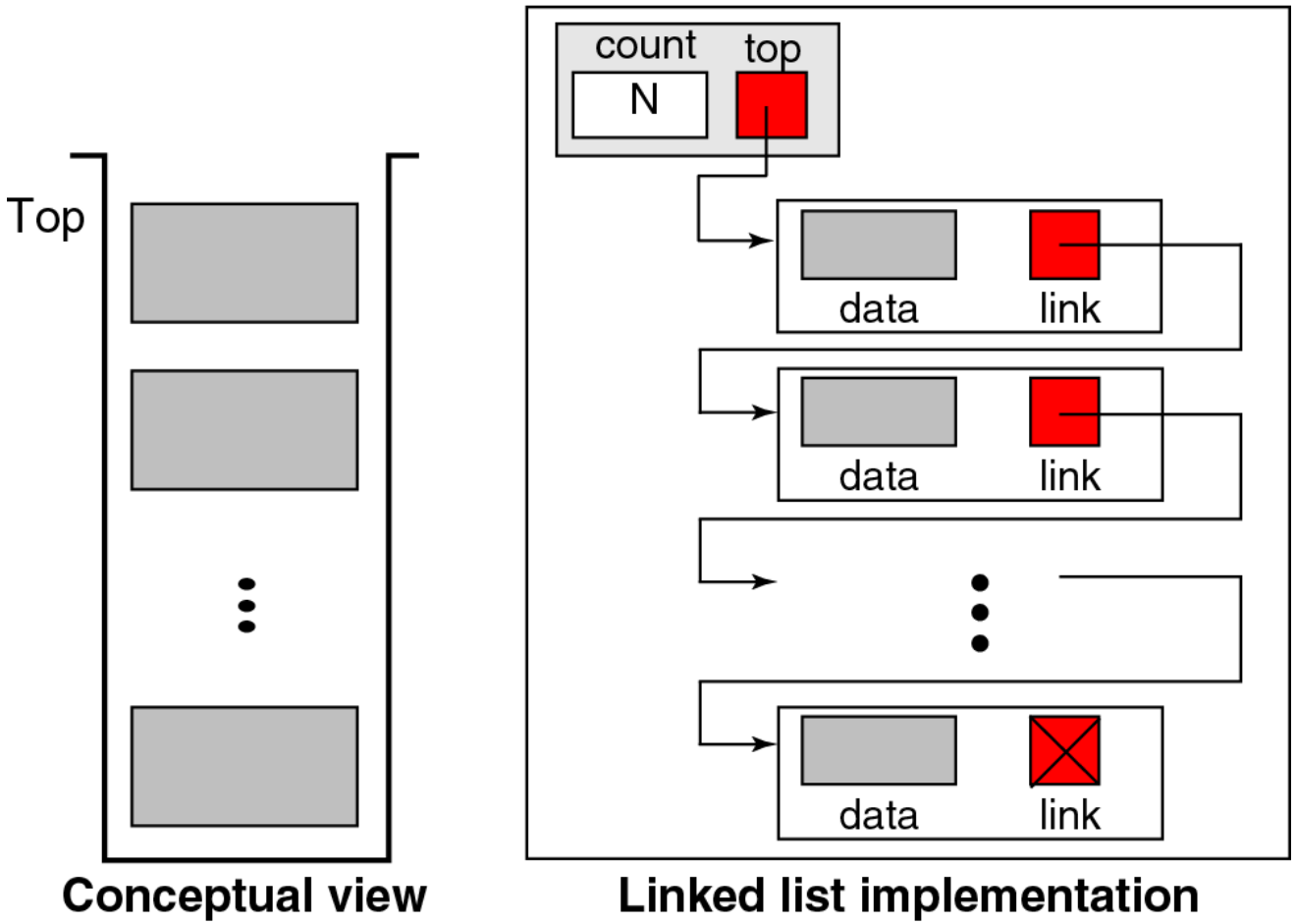
Array-Based Stack

```
  E pop() { // Pop top element
    Assert(top != 0, "Stack is empty");
    return listArray[--top];
  }
  const E& topValue() const { // Return top element
    Assert(top != 0, "Stack is empty");
    return listArray[top-1];
  }
  int length() const { return top; } // Return length
};
```

Issues:

- Which end is the top?
- What is the cost of the operations?

Linked Stack



Linked Stack

- The linked stack implementation is quite simple. The freelist of Section 4.1.2 is an example of a linked stack.
- Elements are inserted and removed only from the head of the list. A header node is not used because no special-case code is required for lists of zero or one elements.
- The only data member is **top**, a pointer to the first (top) link node of the stack.

Linked Stack

```
// Linked stack implementation
Template <typename E> class LStack: public Stack<E>{
private:
  Link<E>* top;     // Pointer to first element
  int size;         // Count number of elements
public:
  LStack(int sz =defaultSize) // Constructor
  { top = NULL; size = 0; }
  ~LStack() { clear(); } // Destructor
```

Linked Stack

```
void clear() { // Reinitialize
  while (top != NULL) { // Delete link nodes
  Link<E>* temp = top;
  top = top->next;
  delete temp;
  }
   size = 0;
}

void push(const E& it) { // Put "it" on stack
  top = new Link<E>(it, top);
  size++;
}
```

Linked Stack

```
E pop() { // Remove "it" from stack
  Assert(top != NULL, "Stack is empty");
  E it = top->element;
  Link<E>* ltemp = top->next;
  delete top;
  top = ltemp;
  size--;
  return it;
}
```

Linked Stack

```
  const E& topValue() const { // Return top value
    Assert(top != 0, "Stack is empty");
    return top->element;
  }
  int length() const { return size; } // Return length
};
```

What is the cost of the operations?

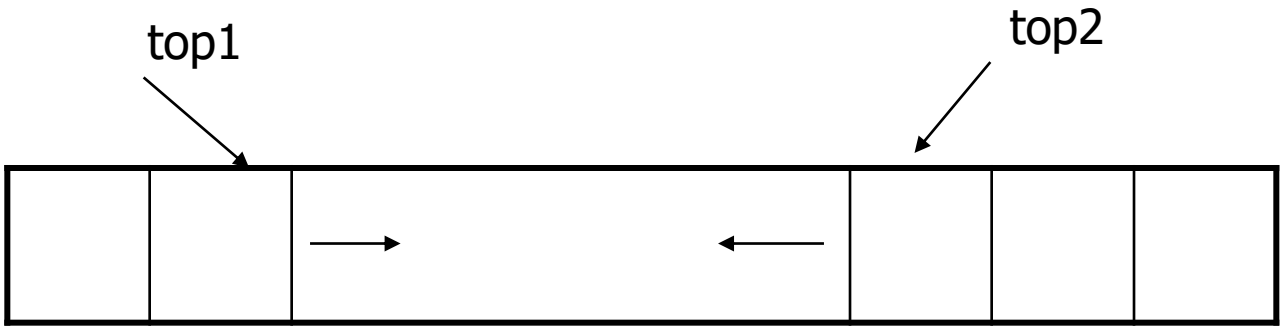
How do space requirements compare to the array-based stack implementation?

Comparison of Array-Based and Linked Stacks

- All operations for the array-based and linked stack implementations take constant time
- Total space required：the analysis is similar to that done for list implementations
- When multiple stacks are to be implemented, it is possible to take advantage of the one-way growth of the array-based stack. This can be done by using a single array to store two stacks，as illustrated by Figure 4.20

Problem

- Modify the code of Figure 4.18 to implement two stacks sharing the same array, as shown in Figure 4.20.



Solution

- template <class E> class AStack2:public Stack<E> {
- private:
- int size; // Maximum size of stack
- int top1, top2; // Index for top element (two)
- E *listArray; // Array holding stack elements
- public:
- AStack2(int sz =defaultListSize) // Constructor
- { size = sz; **top1 = 0;top2 = size - 1;** listArray = new E[sz]; }
- ~AStack2() { delete [] listArray; } // Destructor

Solution

```
void clear(int st) {
  if (st == 1) top1 = 0;
  else top2 = size - 1;
}
bool push(int st, const Elem& item) {
  if (top1+1 >= top2) return false; // Stack is full
  if (st == 1) listarray[top1++] = item;
  else listarraay[top2--] = item;
  return true;
}
```

Solution

```
bool pop(int st, Elem& it) { // Pop top element
  if ((st == 1) && (top1 == 0)) return false;
  if ((st == 2) && (top2 == (size-1))) return false;
  if (st == 1) it = listArray[--top1];
  else it = listArray[++top2];
  return true;
}
```

Solution

```
bool topValue(int st, Elem& it) const { // Return top
  if ((st == 1) && (top1 == 0)) return false;
  if ((st == 2) && (top2 == (size-1))) return false;
  if (st == 1) it = listArray[top1-1];
  else it = listArray[top2+1];
  return true;
}
int length(int st) const {
  if (st == 1) return top1;
  else return size - top2 - 1;
  }
};
```

Implementing Recursion

- Perhaps the most common computer application that uses stacks is the implementation of subroutine calls in most programming language runtime environments.
- A subroutine call is normally implemented by placing necessary information about the subroutine (including **the return address**, **parameters**, **and local variables**) onto a stack.
- Further subroutine calls add to the stack
- Each return from a subroutine pops the top **activation record** off the stack

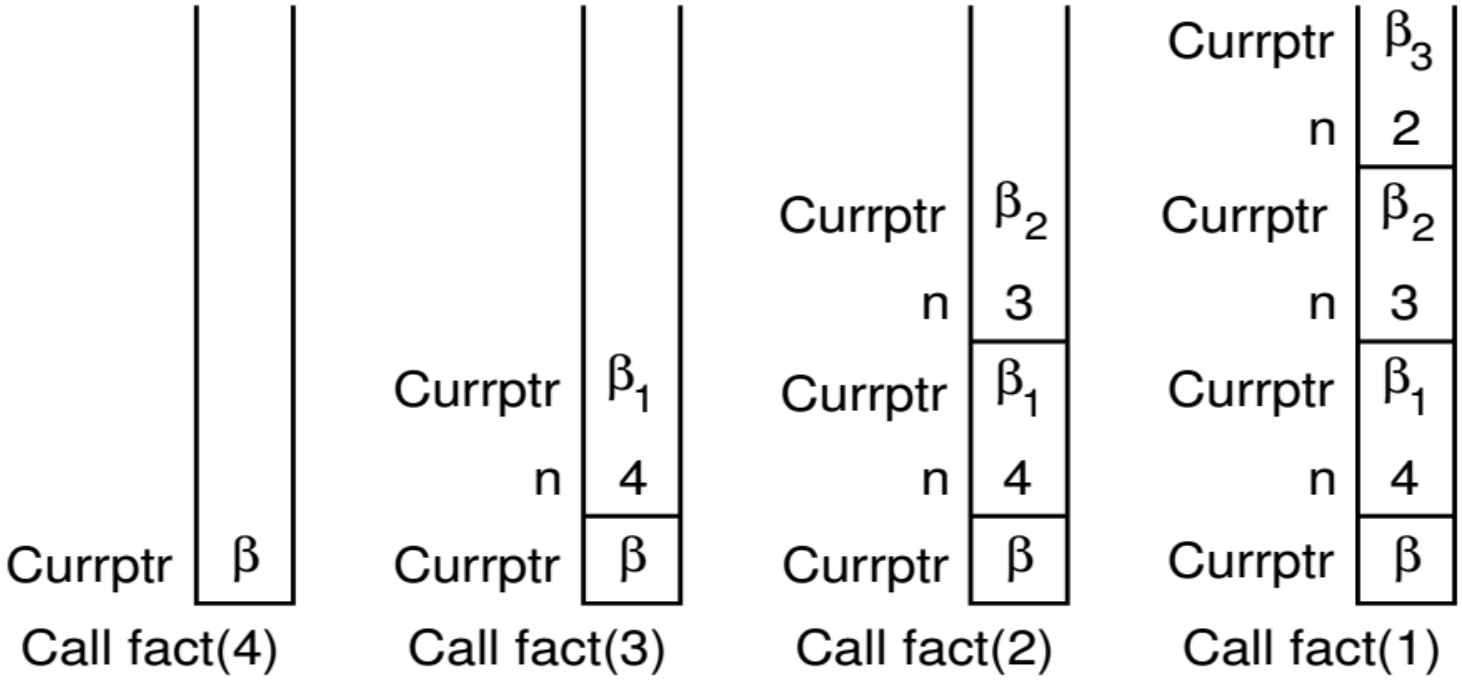
Implementing Recursion

- An algorithm is recursive if it calls itself to do part of its work.
- In general, a recursive algorithm must have two parts:
  - **The base case**, which handles a simple input that can be solved without resorting to a recursive call
  - **The recursive part** which contains one or more recursive calls to the algorithm where the parameters are in some sense "closer" to the base case than those of the original call.

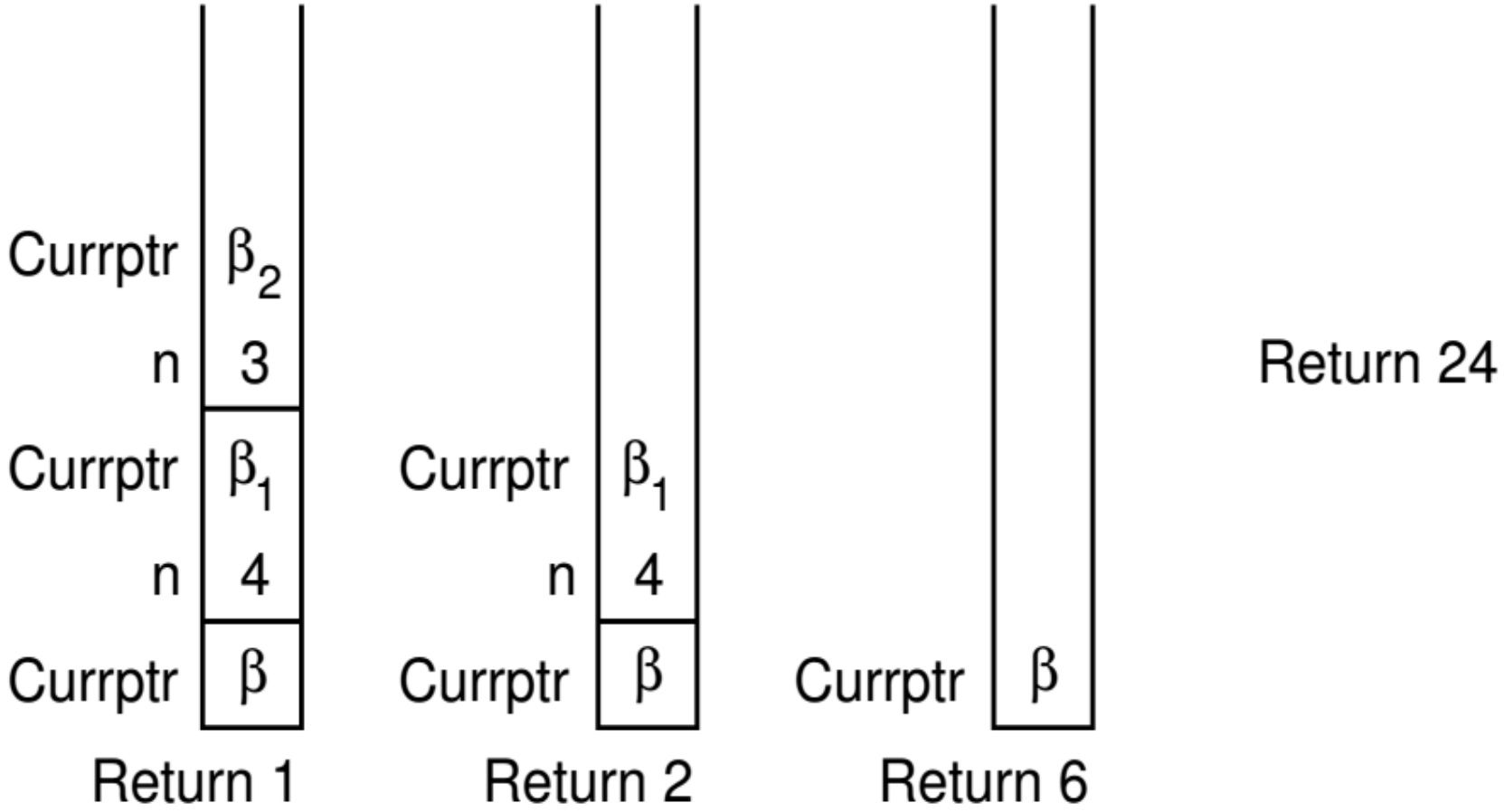
```
long fact(int n) {        // Compute n! recursively
  // To fit n! into a long variable, we require n <= 12
  Assert((n >= 0) && (n <= 12), "Input out of range");
  if (n <= 1)  return 1; // Base case: return base solution
  return n * fact(n-1);  // Recursive call for n > 1
}
```

Implementing Recursion

Consider what happens when we call fact with the value 4:



Implementing Recursion



Stack Applications

Application 1
Data conversion

- A simple algorithm uses the following law:

  N=(N/ d) × d+ N% d

  eg: Convert decimal to binary

  $(100)_{10} = (1100100)_2$

  - divide decimal number by 2 continuously
  - record the remainders until the quotient becomes 0
  - print the remainders in backward order.

- Stack is an ideal structure to implement this process.
- How to implement it?

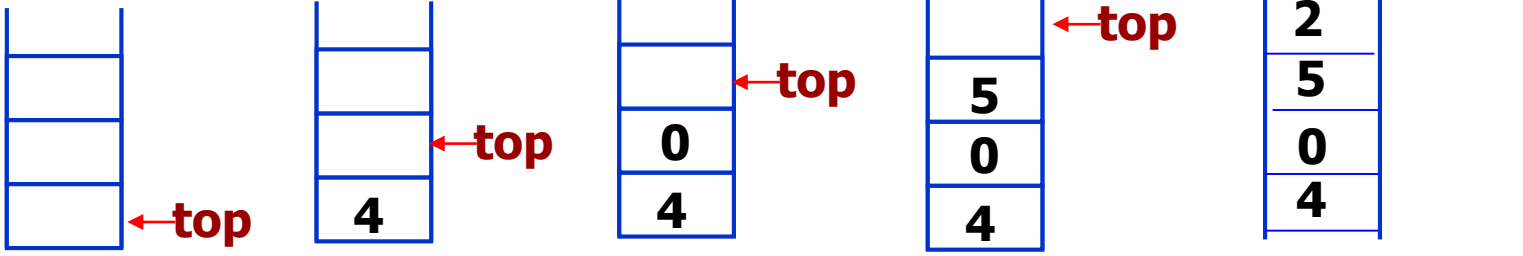
2	100	
2	50	**0**
2	25	0
2	12	1
2	6	0
2	3	0
2	1	1
	0	1

例如：$(1348)_{10}=(2504)_{8}$，其运算过程如下：

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

计算顺序

输出顺序



Applications 2
Balancing Symbols

- How do we know the following parentheses are nested correctly ?

  7-((X*((X+Y)/(J-3))+Y)/(4-2.5))

  1. There are an equal number of right and left parentheses.
  2. Every right parenthesis is preceded by a matching left parenthesis.

- ((A+B) or A+B(violate 1
-)A+B(-C or (A+B))-(C+D violate 2

Applications 2
Balancing Symbols

- Stack can be used to check a program for balanced symbols (such as {}, (), []).
- Example: {()} is legal, {(}) is not (so simply counting symbols does not work).
- When a closing symbol is seen, it matches the most recently seen unclosed opening symbol. Therefore, a stack will be appropriate.

Applications 2
Balancing Symbols

- Make an empty stack. Read characters until end of file.
- If the character is an opening symbol, push it onto stack.
- If it is a closing symbol, then if the stack is empty, report an error. Otherwise, pop the stack.
- If the symbol popped is not the corresponding opening symbol, then report an error.
- At the end of file, if the stack is not empty, report an error.

Reference

- **Data Structures and Algorithm Analysis in C++ .Third Edition.Clifford A. Shaffer**(P.120-127)
- 《数据结构（C 语言版）》，严蔚敏，吴伟民编著，清华大学出版社，1997年第1版,P48-49

-END-