EECS 204002
Data Structures 資料結構
Prof. REN-SONG TSAY 蔡仁松 教授
NTHU

CH. 3
STACKS AND QUEUES

3.2

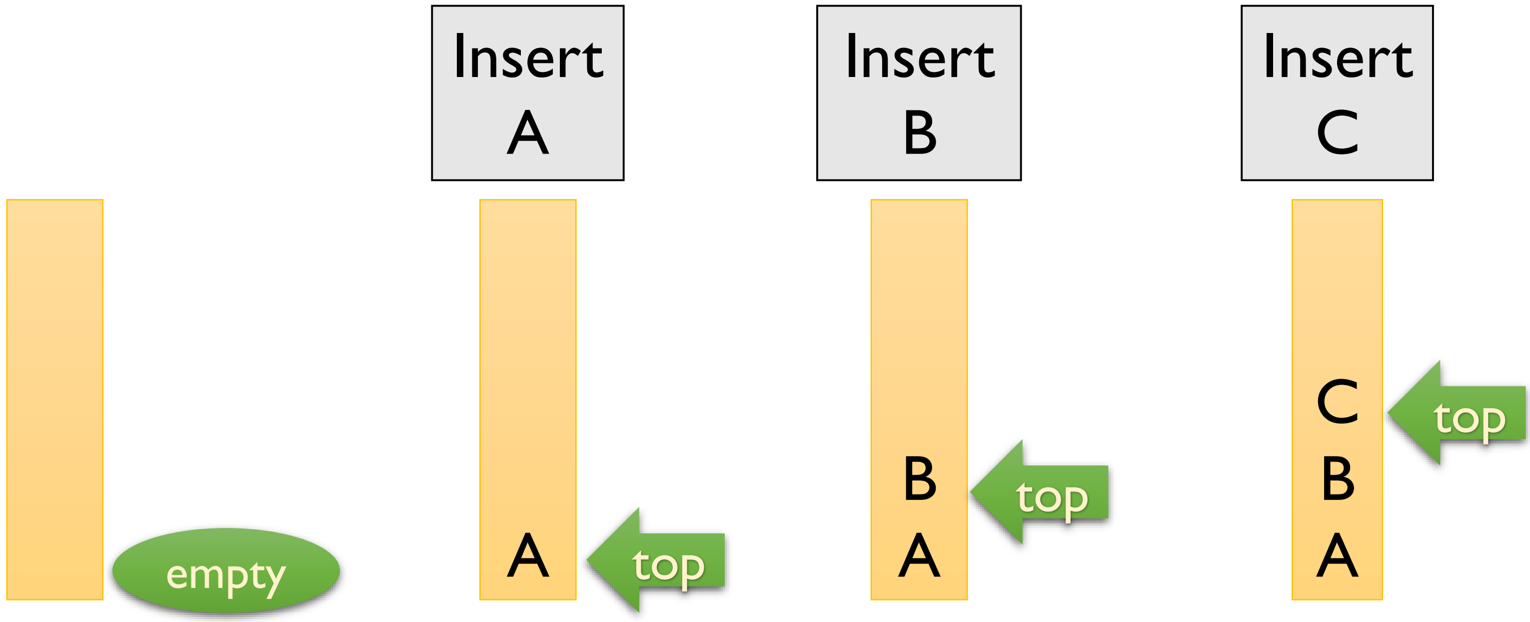
The Stack Abstract Data Type

Stack

- A ***stack*** is an ***ordered list*** in which ***insertions*** (or called ***additions*** or ***pushes***) and ***deletions*** (or called ***removals*** or ***pops***) are made at ***one end*** called the ***top***.
- Operate in ***Last-In-First-Out (LIFO)*** order

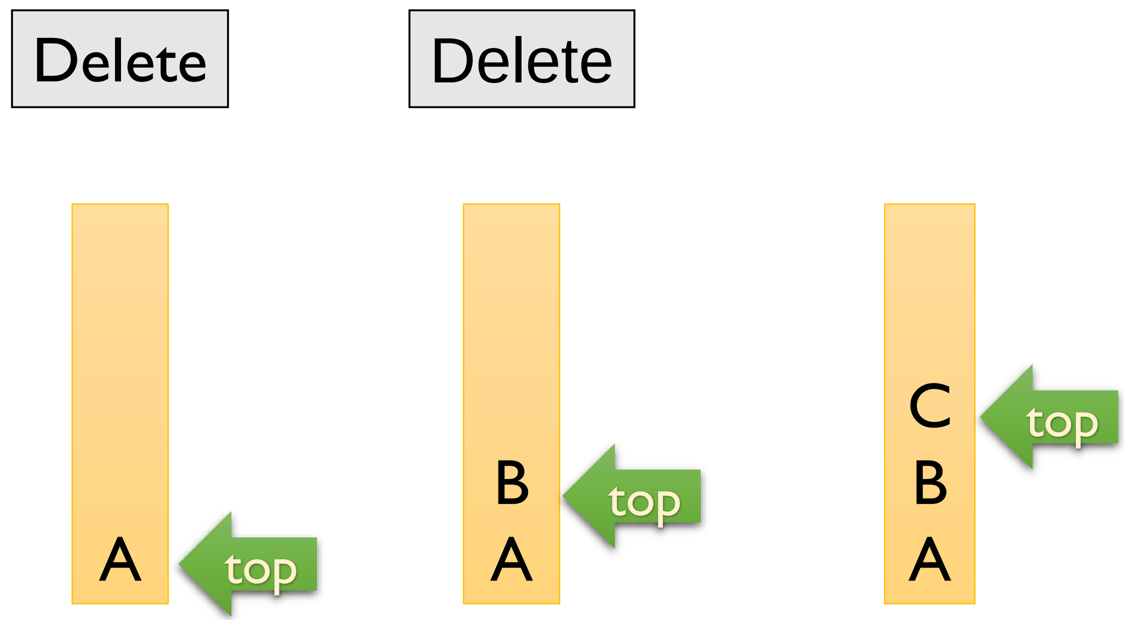
Stack Operations

- Insert a new element into stack



Stack Operations

- Delete an element from stack



Stack: ADT

```
template < class T >
class Stack // A finite ordered list
{
public:
        // Constructor
        Stack (int stackCapacity = 10);

        // Check if the stack is empty
        bool IsEmpty ( ) const;

        // Return the top element
        T& Top ( ) const;

        // Insert a new element at top
        void Push (const T& item);

        // Delete one element from top
        void Pop ( );
private:
        T* stack;
        int top;    // init. value = -1
        int capacity;
};
```

Stack Operations: Push & Pop

```
template < class T >
void Stack < T >::Push (const T& x)
{    // Add x to stack
     if(top == capacity - 1)
     {
        ChangeSize1D(stack, capacity, 2*capacity);
        capacity *= 2;
     }
     stack [ ++top ] = x;
}
```

```
template < class T >
void Stack < T >::Pop ( )
{    // Delete top element from stack
     if(IsEmpty()) throw “Stack is empty. Cannot delete.”;
     stack [ top-- ].~T();  // Delete the element
}
```

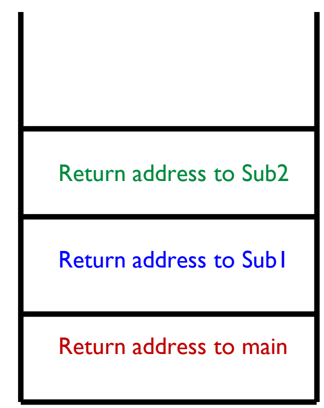
Stack Application

- Function recursion
- System stack
  - Used in the run time to process **recursive function calls**
  - Store the **return addresses** of previous outer procedures

```
main PROC
     .
     .
     call Sub1
     exit
main ENDP

Sub1 PROC
     .
     .
     call Sub2
     ret
Sub1 ENDP

Sub2 PROC
     .
     .
     call Sub3
     ret
Sub2 ENDP

Sub3 PROC
     .
     .
     ret
Sub3 ENDP
```

By the time Sub3 is called, the stack contains all three return addresses:

System Stack

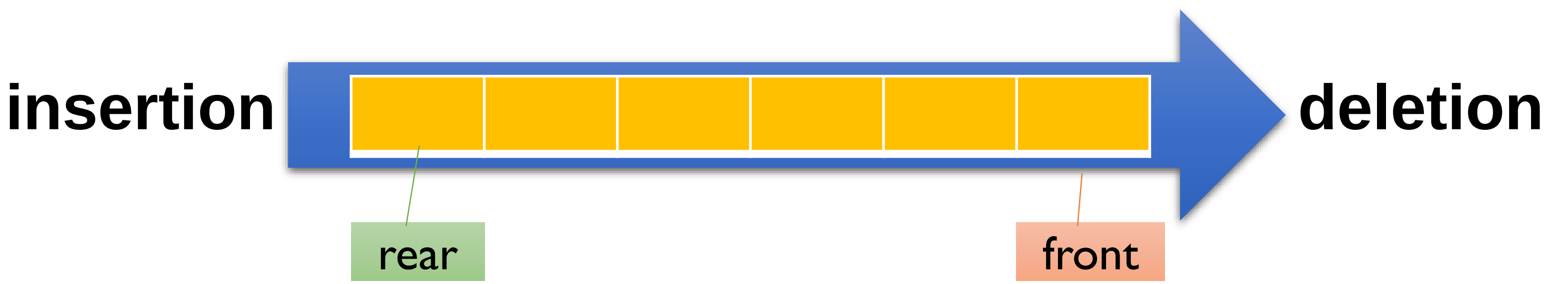


3.3

The Queue Abstract Type

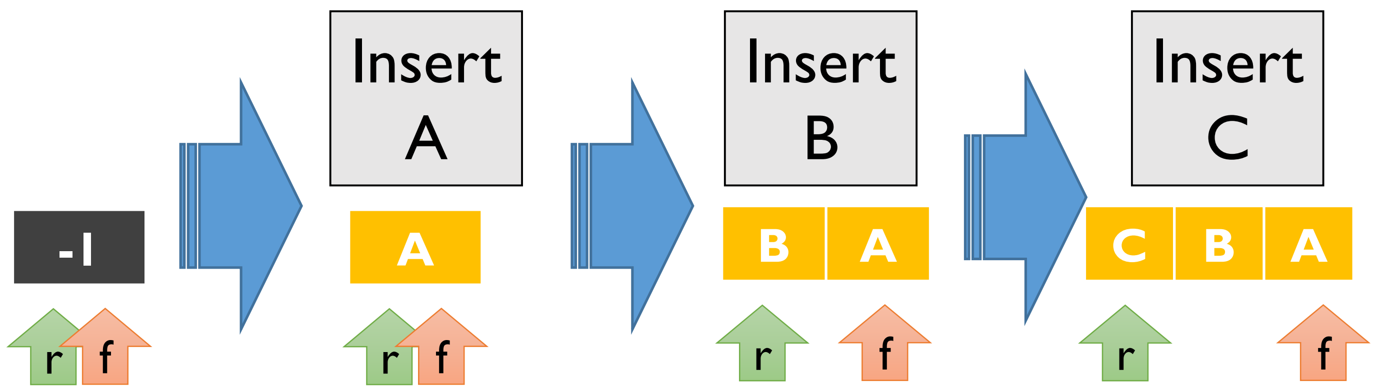
Queue

- A ***queue*** is an ***ordered list*** in which ***insertions*** (or called ***additions*** or ***pushes***) and ***deletions*** (or called ***removals*** or ***pops***) are made at ***different ends***.
- New elements are inserted at ***rear*** end.
- Old elements are deleted at ***front*** end.



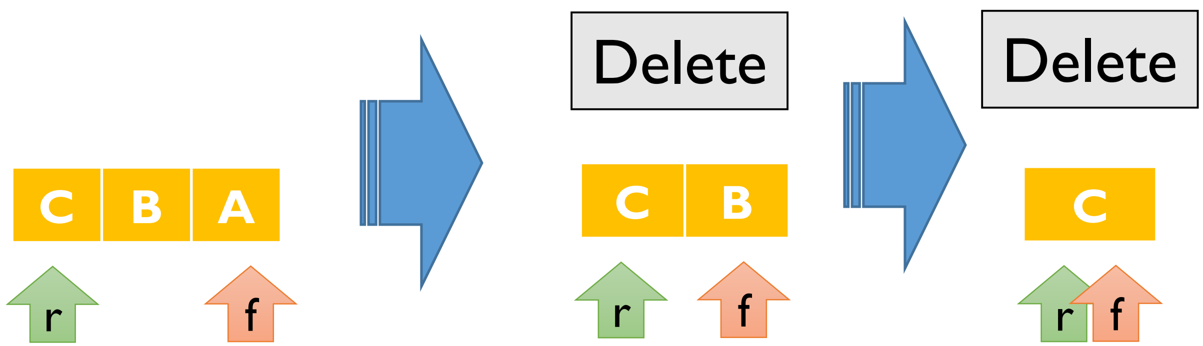
Queue Operations

- Insert a new element into queue
  - f: front position
  - r: rear position



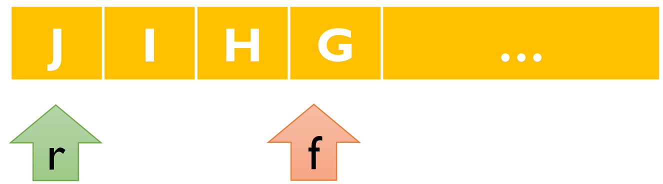
Queue Operations

- Delete an old element from queue
  - f: front position
  - r: rear position

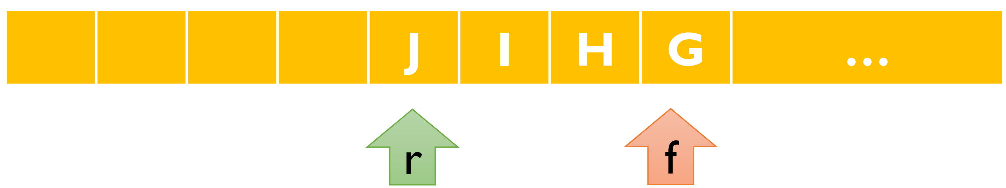


Problems

- What happen if rear == capacity-1 ?

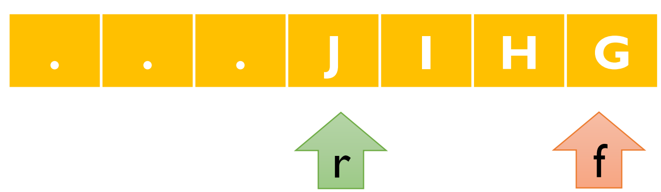


- Add more space ? wasted

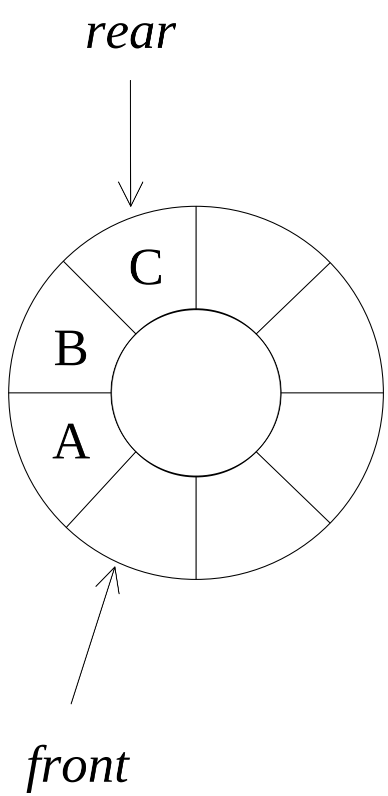


- Shift right?

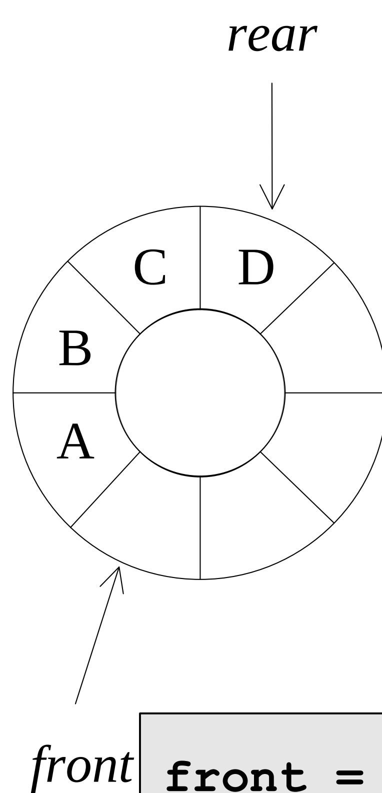
Codes are complicated…



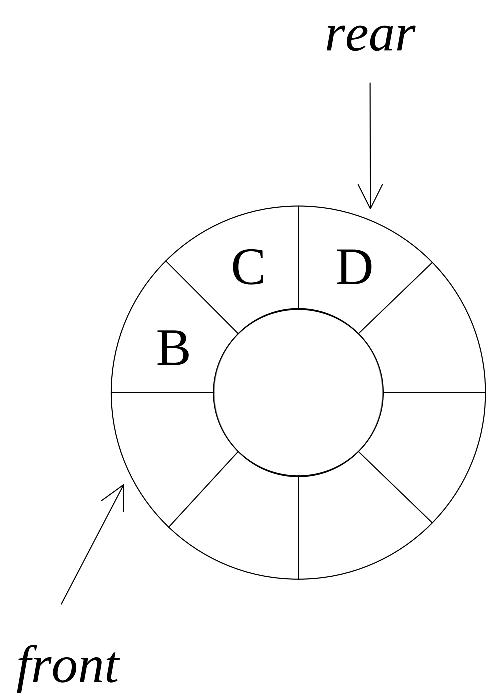
Circular Queue



Initial



Insertion



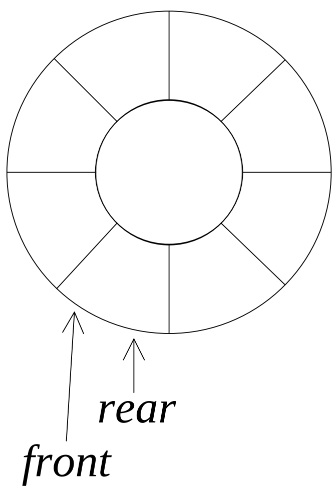

```
front = (front+1) % capacity;
```

Deletion

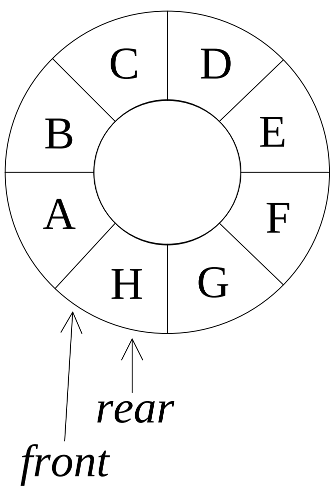
```
rear = (rear+1) % capacity;
```

When is the Queue Empty?

- rear == front ? NO!



Queue is empty



Queue is full

Allocate extra space before the queue is full

Queue: ADT

```
template < class T >
class Queue // A finite ordered list
{
public:
         // Constructor
         Queue (int queueCapacity = 10);

         // Check if the stack is empty
         bool IsEmpty ( ) const;

         // Return the front element
         T& Front ( ) const;

         // Return the rear element
         T& Rear ( ) const;

        // Insert a new element at rear
         void Push (const T& item);

         // Delete one element from front
         void Pop ( );
private:
        T* queue;
        int front, rear; // init. value = -1
        int capacity;
};
```

Queue Operations

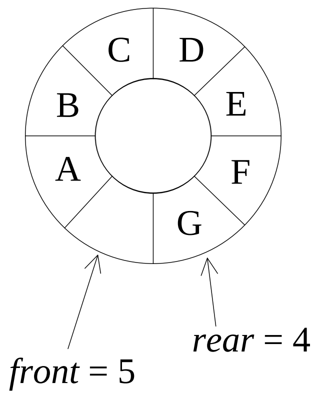
```
template < class T >
void Queue < T >::IsEmpty() const { return front==rear; }

template < class T >
T& Queue < T >::Front() const {
    if(IsEmpty()) throw "Queue is empty!";
    return queue[(front+1)%capacity];
}

template < class T >
T& Queue < T >::Rear() const {
    if(IsEmpty()) throw "Queue is empty!";
    return queue[rear];
}
```

Queue Operations: Push & Pop

```
template < class T >
void Queue< T >::Push (const T& x)
{   // Add x at rear of queue
    if((rear+1)%capacity == front)
    {
        // queue is going to full, double the capacity!
    }
    rear = (rear+1)%capacity;
    queue [rear] = x;
}
```

```
template < class T >
void Queue < T >::Pop ( )
{   // Delete front element from queue
    if(IsEmpty()) throw "Queue is empty. Cannot delete.";
    front = (front+1)%capacity;
    queue[front].~T(); // Delete the element
}
```

Doubling Queue Capacity



Full circular queue

queue	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	C	D	E	F	G		A	B

front = 5, rear = 4

Expanded full circular queue

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	D	E	F	G		A	B								

front = 5, rear = 4

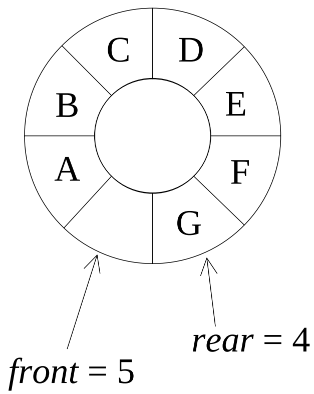
Doubling the array

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	D	E	F	G										A	B

front = 13, rear = 4

Scenario 1: After shifting right segment

Doubling Queue Capacity



Full circular queue

queue	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	C	D	E	F	G		A	B

front = 5, rear = 4

Expanded full circular queue

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	D	E	F	G		A	B								

front = 5, rear = 4

Doubling the array

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C	D	G	F	G									

front = 15, rear = 6

Scenario 2: Alternative configuration

3.4

Generic Bag Container

Bag V.S. Stack

```
class Stack
{
public:
    Stack(int stackCapacity = 10);
    ~Stack();

    bool IsEmpty() const;

    int Top() const;

    void Push(const int);
    void Pop();
};
```

```
class Bag
{
public:
    Bag(int bagCapacity = 10);
   ~Bag();

    int Size() const;
    bool IsEmpty() const;
    int Element() const;

    void Push(Push(const int);
    void Pop()
};
```

Bag V.S. Queue

```
class Bag
{
public:
    Bag(int bagCapacity = 10)
   ~Bag();

    int Size() const;
    bool IsEmpty() const;
    int Element() const;

    void Push(Push(const int);
    void Pop()
};
```

```
class Queue
{
public:
    Queue(int queueCapacity = 10);
    ~Queue();

    bool IsEmpty() const;
    int Rear() const;
    int Front() const;

    void Push(const int);
    void Pop();
};
```

Generic Bag ADT

```
Class Bag
{
public:
  Bag(int bagCapacity=10);
  virtual ~Bag();
  virtual int Size() const;
  virtual bool IsEmpty() const;
  virtual int Element() const;
  virtual void Push(const int);
  virtual void Pop();
protected:
  int *array;
  int capacity;
  int top;
};
```

Implement operations not exist in the Bag class

```
class Stack: public Bag
{
public:
  Stack(int stackCapacity=10);
  ~Stack();
  int Top()const;
  void Pop();
};
```

$$A/B - C + D * E - A * C = ?$$

3.6

Evaluation of Expressions

3.6.1 Regular Expression

$$X = \boxed{A}\boxed{/}\boxed{B}\ \boxed{-}\ \boxed{C}\ \boxed{+}\ \boxed{D}\ \boxed{*}\ \boxed{E}\ \boxed{-}\boxed{A}\ \boxed{*}\boxed{C}$$

- Operators
  - +,-,*,/,…,etc
- Operands
  - A,B,C,D,E,F

Expression Evaluation

- For $X = A/B - C + D * E - A * C$
- If A = 4, B=C=2, D=E=3

- For $X = ((A/B) - C) + (D * E) - (A * C)$
- X = ((4/2)-2)+(3*3)-(4*2)=1

- For $X = (A/(B - C + D)) * (E - A) * C$
- X = (4/(2-2+3))*(3-4)*2 = -2.6666666

Evaluation Rules

- Operators have **priority**
- Operator with **higher priority** is evaluated first
- Operators of **equal priority** are evaluated from **left to right**
- **Unary** operators are evaluated from **right to left**

Priority of Operators in CPP

Priority	Operators
1	Minus, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	==, !=
6	&&
7	||

3.6.2

Infix and Postfix Notation

- **Infix** notation (中序式)
  - Operator comes in–between the operands
  - Ex. A+B*C
  - Hard to evaluate using code…
- **Postfix** notation (後序式)
  - Each operator appears after its operands
  - Ex. ABC*+

Advantages of Postfix Notation

- You don’t need **parentheses**
- Priority of operators is no longer relevant!
- Expression can be efficiently evaluated by
  - Making a left to right scan
  - **Stacking operands**
  - **Evaluating operators**
  - **Push** the **result** into stack

Example 1

- Infix: A+B – C => Postfix: A B + C –
- Suppose A = 4, B = 3, C = 2

A B + C –

Operation
See operand A, put it into stack

4

Operand Stack

Example 1

- Infix : A+B – C => Postfix : A B + C –
- Suppose A = 4, B = 3, C = 2

A $\boxed{B}$ + C –

Operation
See operand B, put it into stack

3
4

Operand Stack

Example 1

- Infix : A+B – C => Postfix : A B + C –
- Suppose A = 4, B = 3, C = 2

A B $\boxed{+}$ C –

3

4

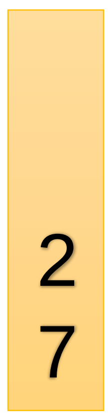
Operand Stack

Operation
See operator '+' (binary operator) 1. Pop two elements from stack 2. Perform evaluation (3+4) 3. Push result into stack (7)

Example 1

- Infix : A+B – C => Postfix : A B + C –
- Suppose A = 4, B = 3, C = 2

A B + $\boxed{\text{C}}$ –



Operand Stack

Operation
See operand C, put it into stack

Example 1

- Infix : A+B – C => Postfix : A B + C –
- Suppose A = 4, B = 3, C = 2

A B + C $\boxed{–}$

2

5 7

Operand Stack

Operation
See operator '-' (binary operator) 1. Pop two elements from stack 2. Perform evaluation (7-2) 3. Push result into stack (5)

Example 2

- Infix: $X = A/B - C + D * E - A * C$
- Postfix: $X = \boxed{A}B/C - DE * +AC * -$

Operand Stack

Operation
See operand A, put it into stack

Example 2

- Infix: $X = A/B - C + D * E - A * C$
- Postfix: $X = A\boxed{B}/C - DE * + AC * -$

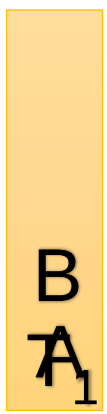
B
A

Operand Stack

Operation
See operand B, put it into stack

Example 2

- Infix: $X = A/B - C + D * E - A * C$
- Postfix: $X = AB\boxed{/}C - DE * +AC * -$



Operand Stack

Operation
See operator '/' 1. Pop two elements from stack 2. Perform evaluation (T_1=A/B) 3. Push result into stack (T_1)

Example 2

- Infix: $X = A/B - C + D * E - A * C$
- Postfix: $X = AB/\boxed{C} - DE * +AC * -$

C

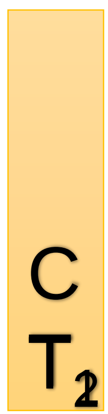
T_1

Operand Stack

Operation
See operand C, put it into stack

Example 2

- Infix: $X = A/B - C + D * E - A * C$
- Postfix: $X = AB/C \boxed{-} DE * +AC * -$



Operand Stack

Operation
See operator '-' 1. Pop two elements from stack 2. Perform evaluation (T_2=T_1-C) 3. Push result into stack (T_2)

Example 2

- Infix: $X = A/B - C + D * E - A * C$
- Postfix: $X = AB/C - \boxed{D}E * +AC * -$

D

T_2

Operand Stack

Operation
See operand D, put it into stack

Example 2

- Infix: $X = A/B - C + D * E - A * C$
- Postfix: $X = AB/C - D\boxed{E} * +AC * -$

E

D

T_2

Operand Stack

Operation
See operand E, put it into stack

Example 2

- Infix: $X = A/B - C + D * E - A * C$
- Postfix: $X = AB/C - DE\boxed{*}+AC * -$

E

D T_3

T_2

Operand Stack

Operation
See operator '*' 1. Pop two elements from stack 2. Perform evaluation (T_3=D*E) 3. Push result into stack (T_3)

Example 2

- Infix: $X = A/B - C + D * E - A * C$
- Postfix: $X = AB/C - DE * \boxed{+} AC * -$

T_3

T_2

Operand Stack

Operation
See operator ‘+’ 1. Pop two elements from stack 2. Perform evaluation (T_4=T_2+T_3) 3. Push result into stack (T_4)

Try the rest of steps yourself!

Evaluation Pseudo Code

```
void Eval(Expression e)
{  // Assume the last token of e is '#'
   // A function NextToken is used to get next token in e
   Stack<Token> stack; // initialize stack
   for (Token x = NextToken(e); x != '#'; x = NextToken(e)){
     if(x is an operand) stack.Push(x);
     else{
          // Remove the correct number of operands from stack
          // Perform the evaluation
          // Push the result back to stack
          // ***Try to fill up the code ***
     }
   }
};
```

Infix to Postfix

- Fully parenthesize algorithm:
  - Fully parenthesize the expression
  - Move all operators so the they replace the corresponding right parentheses
  - Delete all parentheses

((((A / B) – C) + (D * E)) - (A * C))

A B / C – D E * + A C * -

Smarter Infix to Postfix Algorithm

- Utilize **stack**
- Scan the expression only once
- The order of operands does not change between infix and postfix
  - Output every visiting operand directly
- Use stack to store visited operators and pop them out at the proper sequence
  - When the ***priority*** of the operator on top of stack is ***higher or equal to*** that of the incoming operator (left-to-right associativity)

Example 1

- Infix: A + B * C

Next token	Stack	Output
None	Empty	None
A	Empty	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
	+	ABC*
	Empty	ABC*+

Example 2

- Infix: A * (B + C) * D

Next token	Stack	Output
None	Empty	None
A	Empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
*	*	ABC+*
D	*	ABC+*D
	Empty	ABC+*D*

Notes: Expression with ( )

- ‘(‘ has the highest priority, always push to stack.
- Once pushed, ‘(’ get the lowest priority.
- ‘)’ has the lowest priority, therefore pop the operators in the stack until you see the matched ‘(’, then eliminate both.

Postfix Pseudo Code

```
void Postfix(Expression e)
{  // Assume the last token of e is '#'
   // A function NextToken is used to get next token in e
   Stack<Token> stack; // initialize stack
   for (Token x = NextToken(e); x != '#'; x = NextToken(e)){
     if(x is an operand) cout << x;
     else if (x == ')'){ // pop until '('
       for(; stack.Top()!='('; stack.Pop()) cout<<stack.Top();
       stack.Top(); // pop '('
     }
     else{ // x is an operator
       for(;icp(stack.Top()) <= icp(x);stack.Pop())
            cout<<stack.Top();
       stack.Push(x);
     }
   }
   // end of expression; empty the stack
   for(;!stack.IsEmpty(); cout << stack.Top(), stack.Pop());
};
```