CS 235101

Data Structures

資料結構

Stacks and Queues

Department of Computer Science

National Tsing Hua University

ABSTRACTED CONTAINERS

Container Classes

- A data structure that contains or store a number of data objects (ordered list)
- Support various common operations
  - Is the container empty?
  - How many objects are in the container?
  - Add one object into the container
  - Delete one object from the container
  - Access one of the object in the container

Abstracted Bag Container

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

    int Size() const;           // Return the number of elements

    bool IsEmpty() const;       // Check if bag is empty

    int Element() const;        // Return an element in the bag

    void Push(const int);       // Insert an integer into the bag
    void Pop()                  // Delete an integer from the bag

private:
    int *array;                 // Integer array that stores the data
    int capacity;               // Capacity of array
    int top;                    // Position of top element
};
```

Bag Implementation

```
Bag::Bag( int bagCapacity):capacity( bagCapacity ) {
   if(capacity < 1) throw “Capacity must be > 0”;
   array = new int [ capacity ];
   top = -1;
}

Bag::~Bag(){ delete [] array; }

inline int Bag::Size() const { return top + 1; }

inline bool Bag::IsEmpty() const { return Size() == 0; }

inline int Bag::Element() const {
   if(IsEmpty()) throw “Bag is empty”;
   return array [0];  // Always return the first element
}

void Bag::Push(const int x) {
   if(capacity == top+1) ChangeSize1D(array,capacity,2* capacity);
   capacity *= 2;
   array[++top]=x;
}

void Bag::Pop( ) {
   if(IsEmpty()) throw “Bag is empty, cannot delete”;
   int deletePos = top / 2; // Always delete the middle element
   copy (array+deletePos+1, array+top+1, array+deletePos);
   top--;
}
```

How to Use Template?

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

    int Size() const;           // Return the number of elements
    bool IsEmpty() const;       // Check if bag is empty
    int Element() const;        // Return an element in the bag

    void Push(const int);       // Insert an integer into the bag
    void Pop()                  // Delete an integer from the bag

private:
    int *array;                 // Integer array that stores the data
    int capacity;               // Capacity of array
    int top;                    // Position of top element
};
```

Abstracted Bag Container

```
template<class T>
class Bag
{
public:
    Bag(int bagCapacity = 10);  // Constructor
   ~Bag();                      // Destructor

    int Size() const;           // Return the number of elements
    bool IsEmpty() const;       // Check if bag is empty
    T& Element() const;         // Return an element in the bag

    void Push(const T&);        // Insert an element into the bag
    void Pop()                  // Delete an element from the bag

private:
    T *array;                   // Data array
    int capacity;               // Capacity of array
    int top;                    // Position of top element
};
```

Template Bag Implementation

```
template<class T>
Bag<T>::Bag( int bagCapacity):capacity( bagCapacity ) {
   if(capacity < 1) throw “Capacity must be > 0”;
   array = new T [ capacity ];
   top = -1;
}

template<class T>
void Bag<T>::Push(const T& x)  {
   if(capacity == top+1) ChangeSize1D(array,capacity,2* capacity);
   capacity *= 2;
   array[++top]=x;
}

template<class T>
void Bag<T>::Pop()  {
   if(IsEmpty()) throw “Bag is empty, cannot delete”;
   int deletePos = top / 2; // Always delete the middle emelent
   copy (array+deletePos+1, array+top+1, array+deletePos);
   array[top--].~T();
}
```

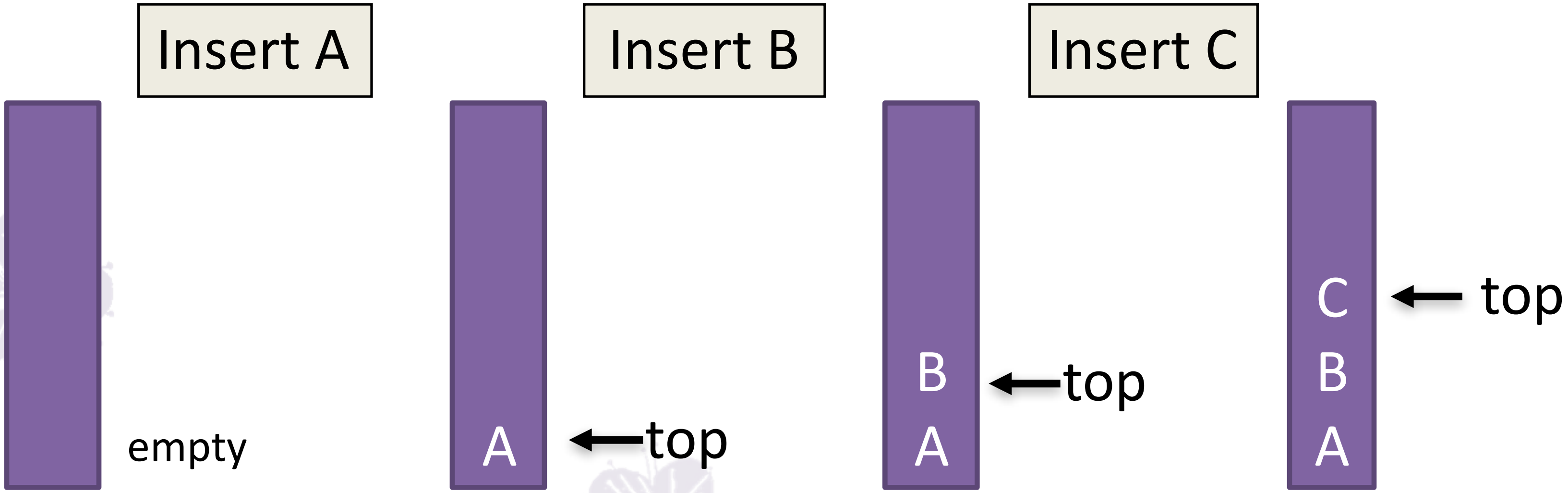
THE STACK

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.

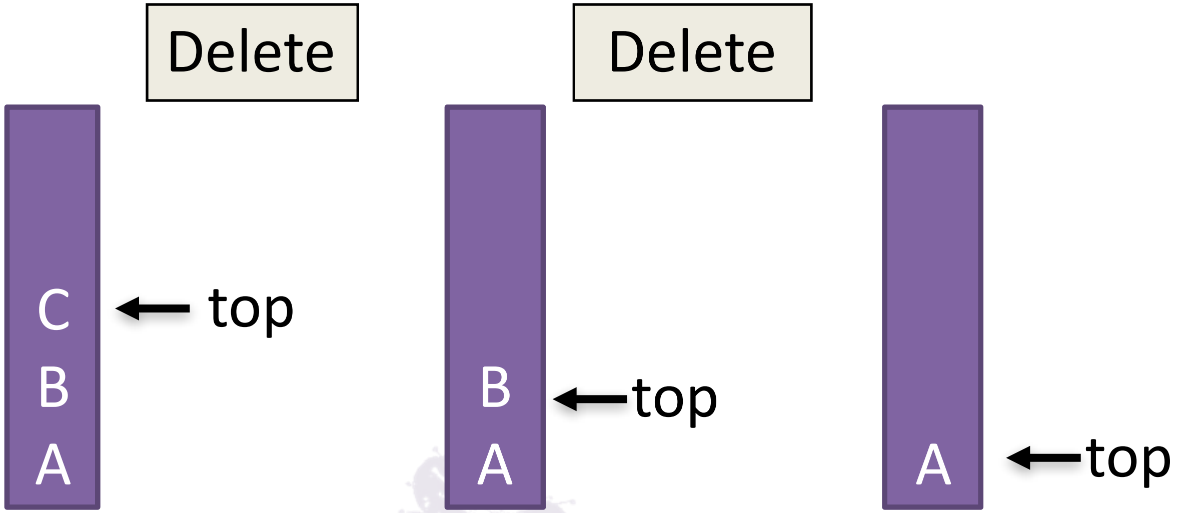
Stack Operations

- Insert a new element into stack



Stack Operations

- Delete an element from stack



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: 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;
};
```

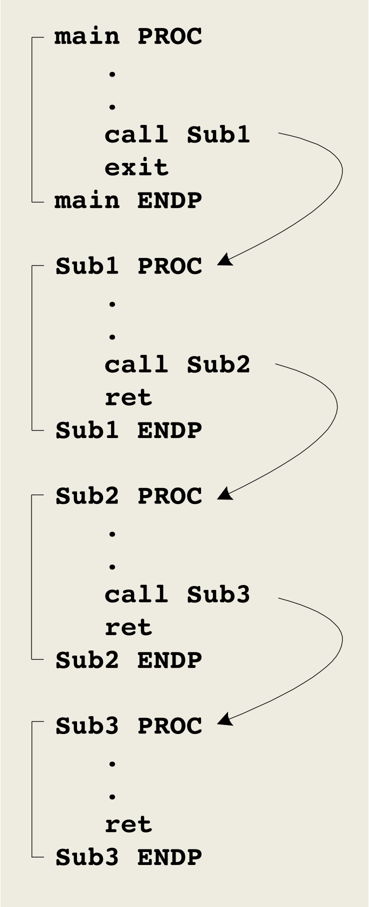
Stack Operations: Push & Pop

```
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
}
```

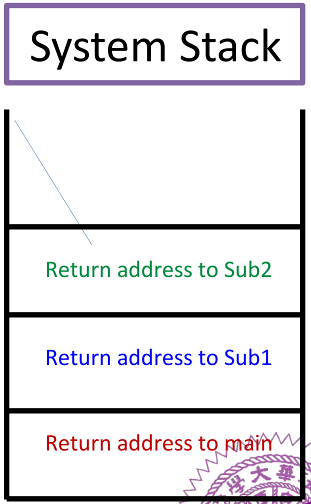
Stack Application

- 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:



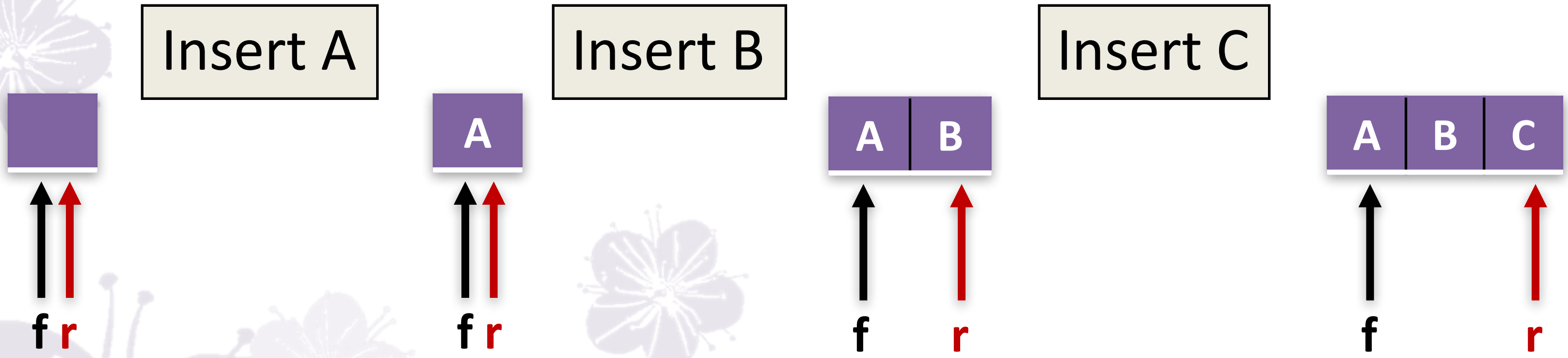
THE QUEUE

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.

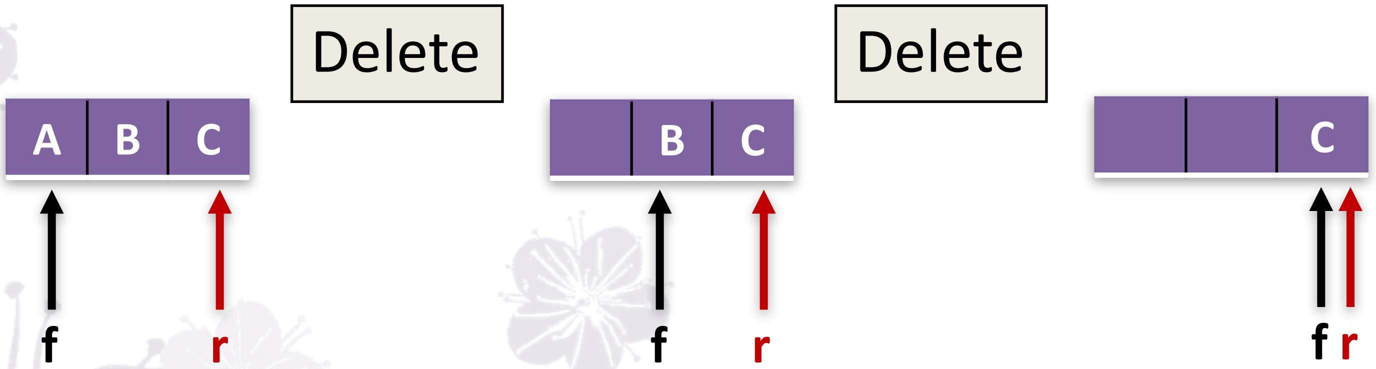
Queue Operations

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



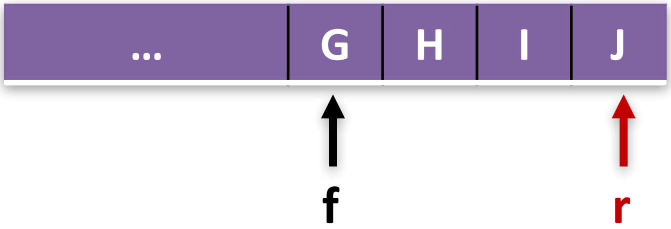
Queue Operations

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

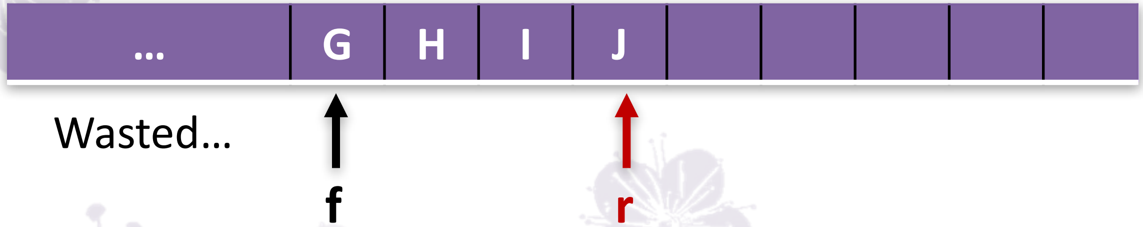


Problems

- What happen if rear == capacity-1 ?

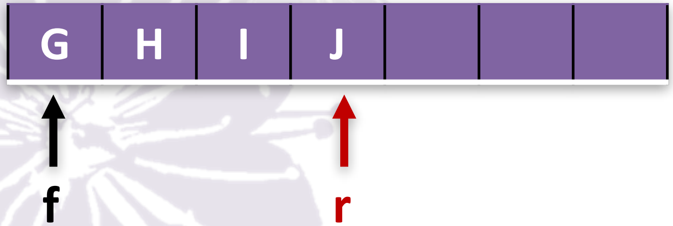


- Add more space ?



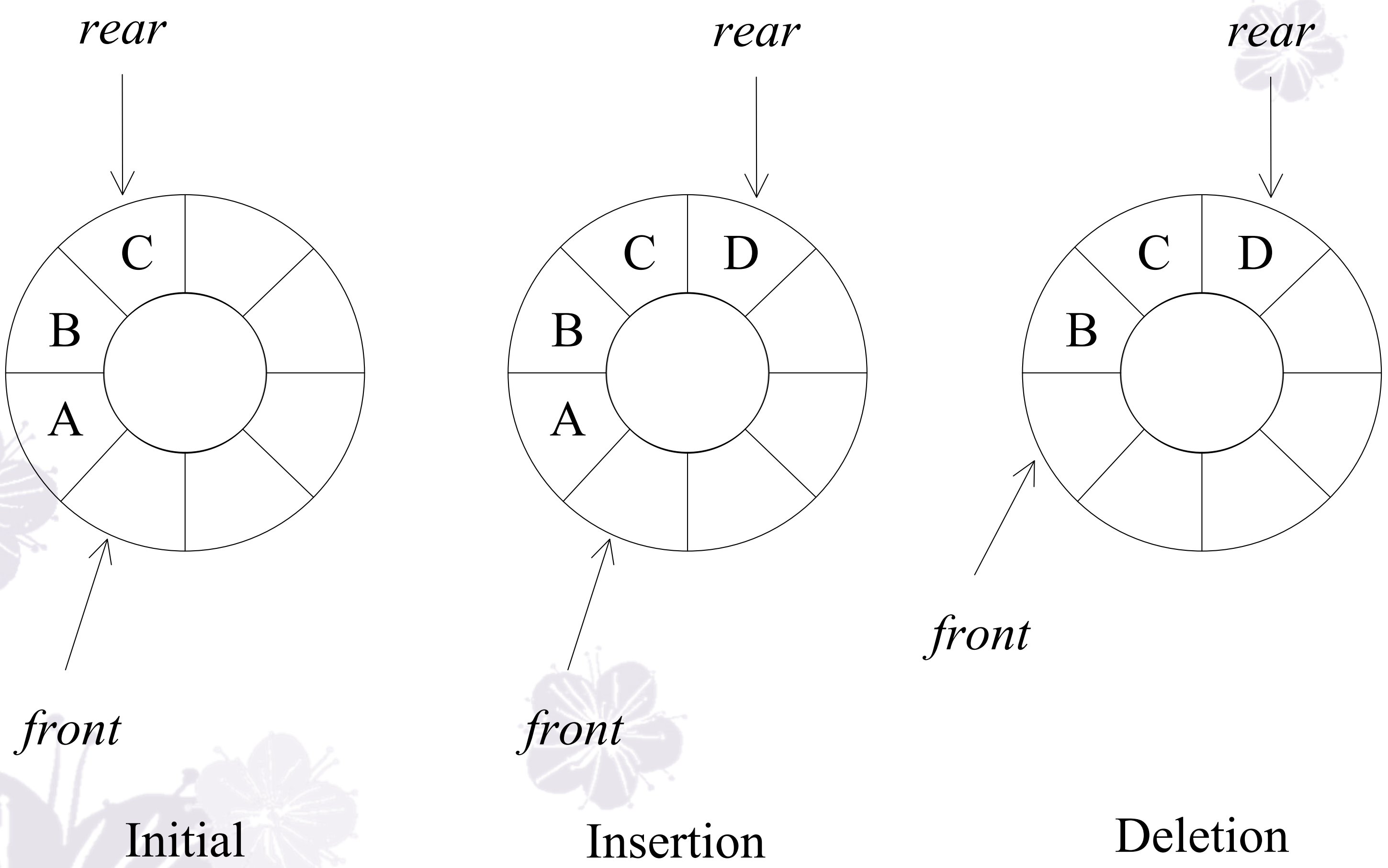
Wasted...

- Shift left ?



Codes are complicated...

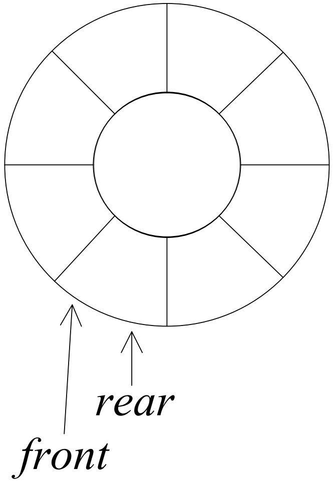
Circular Queue



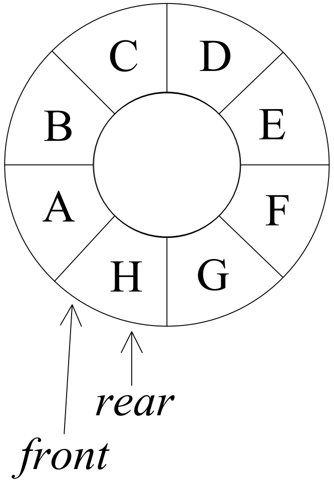
```
rear = (rear+1) % capacity;
```

Circular Queue

- When is the queue empty?
  - rear == front ? NO!



Queue is empty



Queue is full

Allocate addition 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

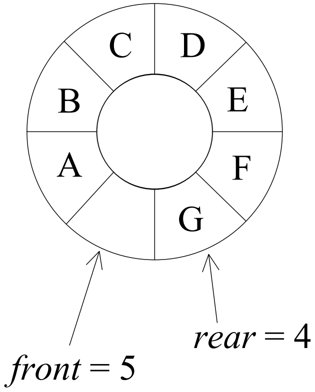
```
template < class T >
void Queue < T >::IsEmpty() const { return front==rear; }

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

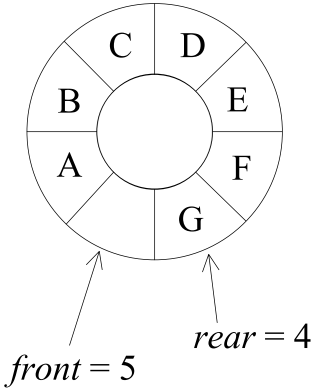
Doubling the array

[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

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
queue	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

GENERIC BAG CONTAINER!

Bag V.S. Stack

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

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

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

```
template < class T >
class Stack
{
public:
    Stack(int stackCapacity = 10);
    ~Stack();


    bool IsEmpty() const;

    T& Top() const;

    void Push(const T& item);
    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(const int);
    void Pop()
};
```

```
template < class T >
class Queue
{
public:
    Queue(int queueCapacity = 10);
    ~Queue();


    bool IsEmpty() const;
    T& Rear() const;
    T& Front() const;

    void Push(const T& item);
    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);
  virtual ~Stack();
  int Top()const;
  virtual void Pop();
};
```

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

EVALUATION OF EXPRESSIONS

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
- X = ((4/2)-2)+(3*3)+(4*2)=1

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

Infix and Postfix Notation

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

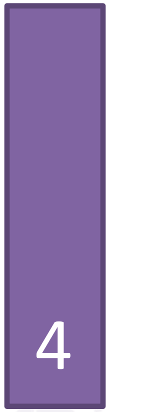
Advantages of Postfix Notation

- 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 –

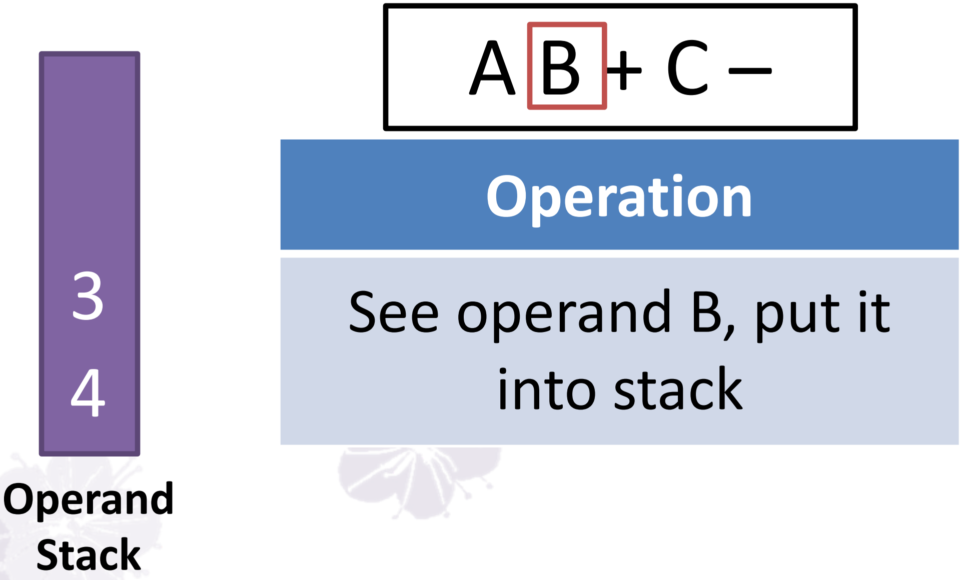


Operand Stack

Operation
See operand A, put it into stack

Example 1

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

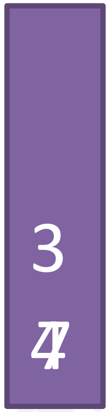


A B + C –

Operation
See operand B, put it into stack

Example 1

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



3

4

7

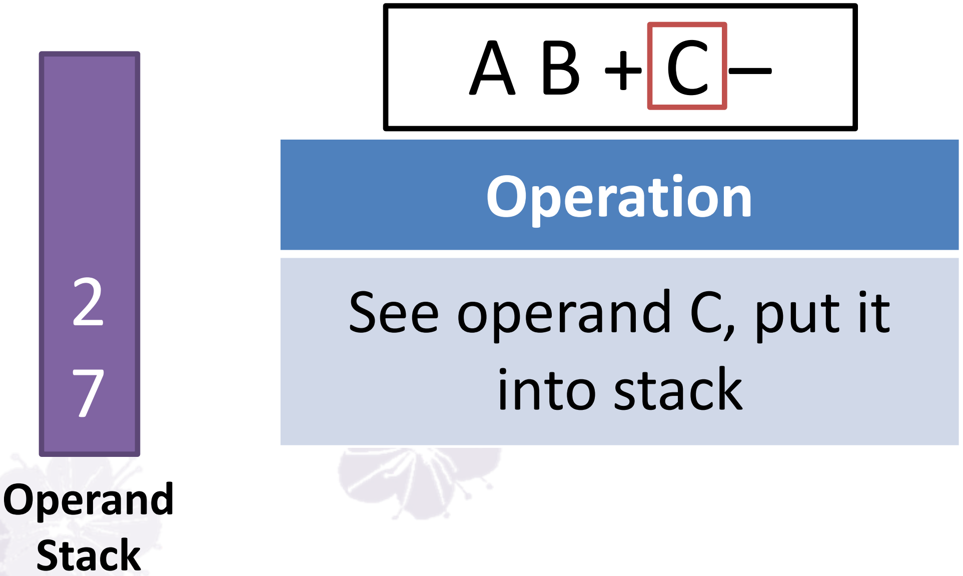
Operand Stack

A B $\boxed{+}$ C –

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

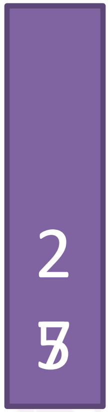


Operation
See operand C, put it into stack

Example 1

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

A B + C –

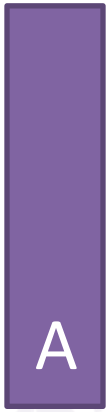


Operand Stack

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

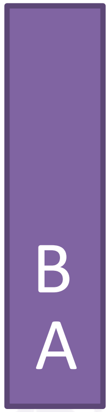


Operand Stack

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

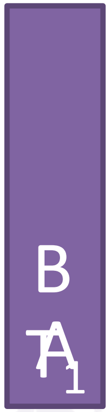
A

Operand Stack

Operation
See operand B, put it into stack

Example 2

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

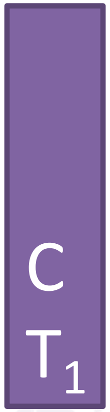


Operand Stack

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

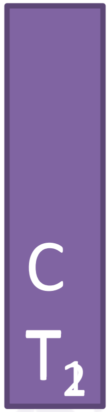
T_1

Operand Stack

Operation
See operand C, put it into stack

Example 2

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



C

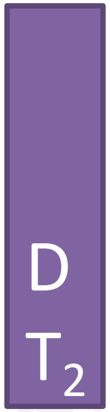
T_2

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

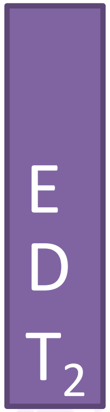


Operand Stack

Operation
See operand D, put it into stack

Example 2

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

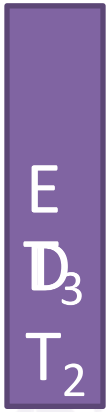


Operand Stack

Operation
See operand E, put it into stack

Example 2

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

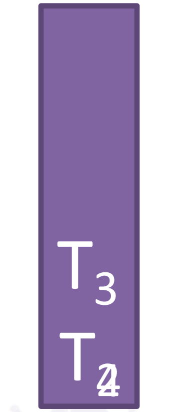


Operand Stack

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 : $X = AB/C - DE * \boxed{+} AC * -$



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)

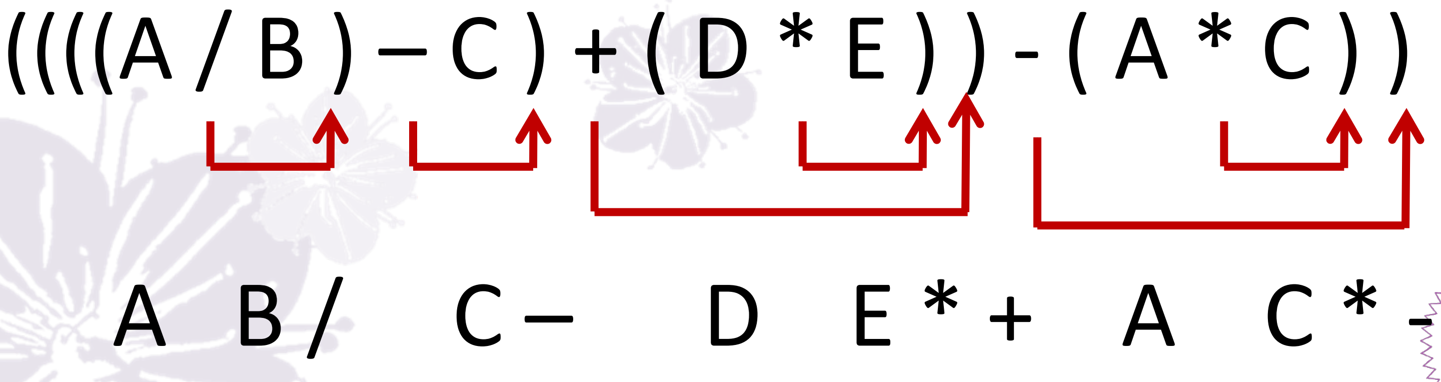
Try the rest of steps by your own!

Evaluation Pseudo Codes

```
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 codes by your own***
     }
   }
};
```

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 * -

Infix to Postfix

- Smarter algorithm
  - Scan the expression only once
  - Utilize **stack**
- The order of operands dose not change between infix and postfix
  - Output every visiting operand directly
- Use stack to store visited operators and pop them out at the right moment
  - When the ***priority*** of 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 lowest priority.
  - Pop the operators until you see the matched ‘)’

Pseudo Codes

```
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.Pop(); // 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());
};
```