CSE221

Stacks and Queues

2021 Fall

Young-ri Choi

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided in former lectures at UNIST.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

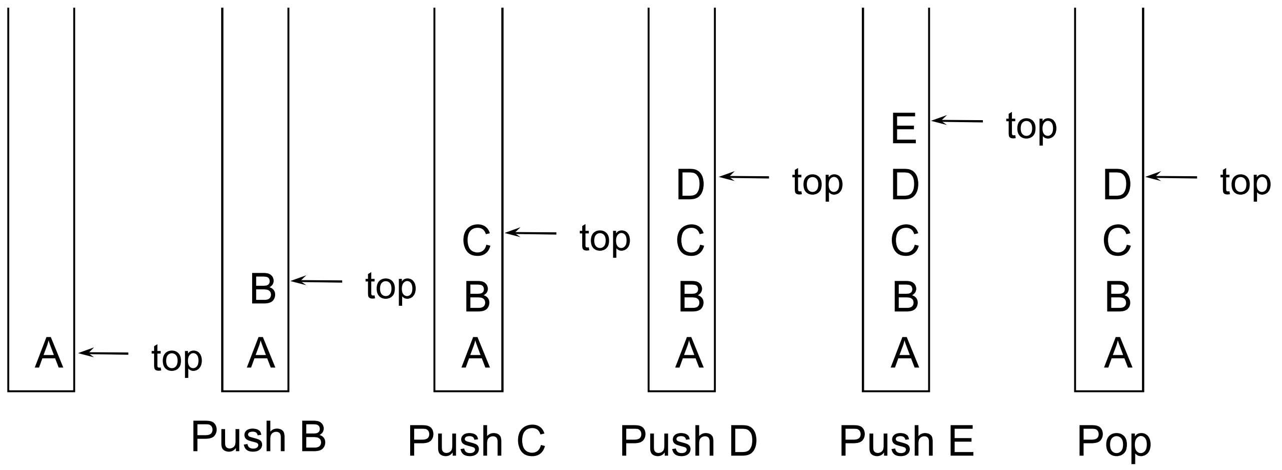
- Stacks & Queues
  - Stack ADT
  - Linear queue
  - Circular queue
- Examples
  - Queue using Stacks
  - Evaluation of expression

Stack

- Special case of ordered (linear) list
- Data insertion (push) and deletion (pop) happen at the top
- Last In First Out (LIFO)

Push & Pop

- Push A,B,C,D,E and pop one element



Stack ADT

```
template <class KeyType>
class Stack
{
// A finite ordered list with zero or more elements
public:
   Stack (int MaxStackSize = DefaultSize);
   ~Stack();

   Boolean IsFull();

   Boolean IsEmpty();

   void Push(const KeyType& item);
   // Insert item into the top of the stack

   KeyType& Top() const;
   // Return top element of stack (but not delete)

   void Pop();
   // Delete top element
};
```

Stack Implementation

Implementation based on array

- Push

```
template <class KeyType>
void Stack<KeyType>::Push(const KeyType& x)
{
   if (IsFull()) ChangeSize();
   stack[++top] = x;
}
```

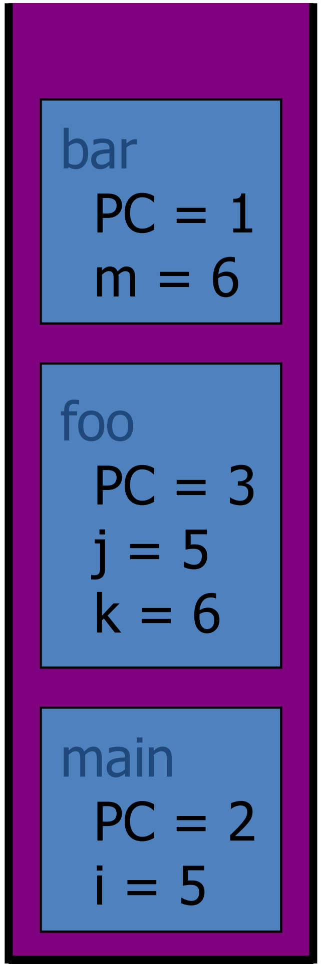
- Pop

```
template <class KeyType>
void Stack<KeyType>::Pop()
{
   if (IsEmpty()) return;
   stack[top--].~KeyType(); // destructor
}
```

C++ Run-Time Stack

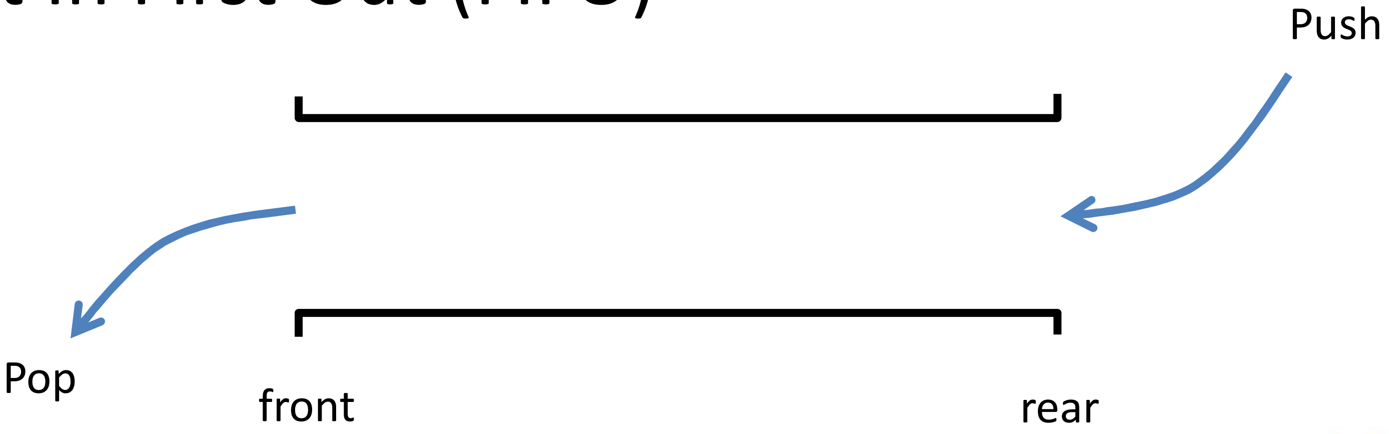
- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When the function ends, its frame is popped from the stack and control is passed to the function on top of the stack
- Allows for recursion

```
main() {
   int i = 5;
   foo(i);
   }
foo(int j) {
   int k;
   k = j+1;
   bar(k);
   }
bar(int m) {
   …
   }
```

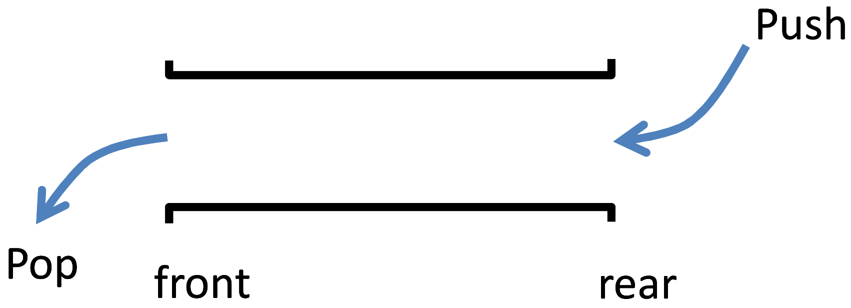


Queue

- Special case of ordered (linear) list
- Data insertion (push) takes place at *rear*
- Data deletion (pop) takes place at *front*
- First In First Out (FIFO)



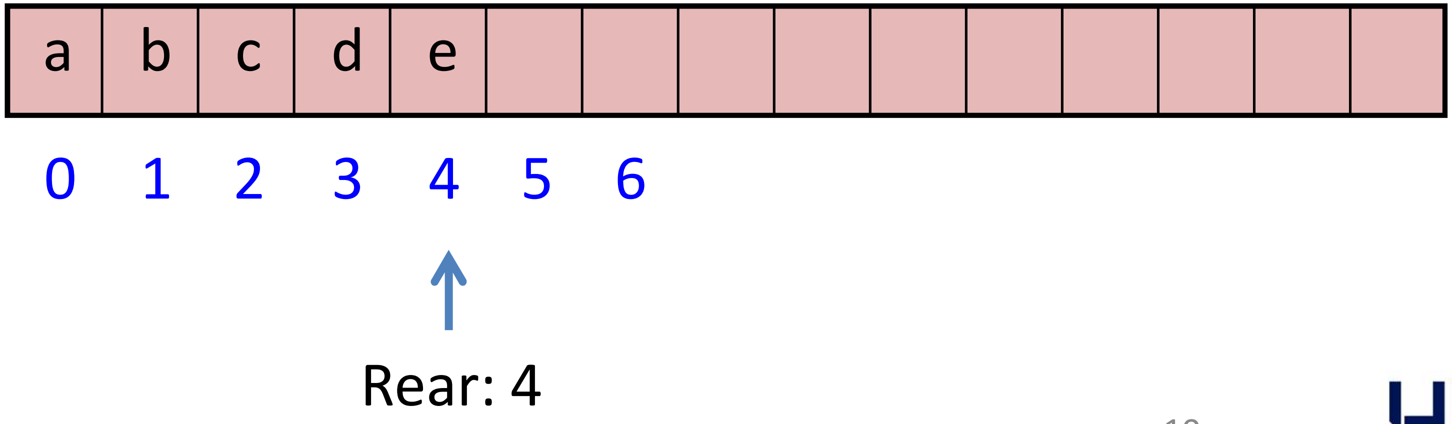
Queue



- Container of (data) objects
  - that shows First In First Out (FIFO) behavior
- Data manipulation happens at “rear”/”front”
  - insertion (push/enqueue) at *rear*
  - deletion (pop/dequeue) at *front*
- Special case of ordered (linear) list

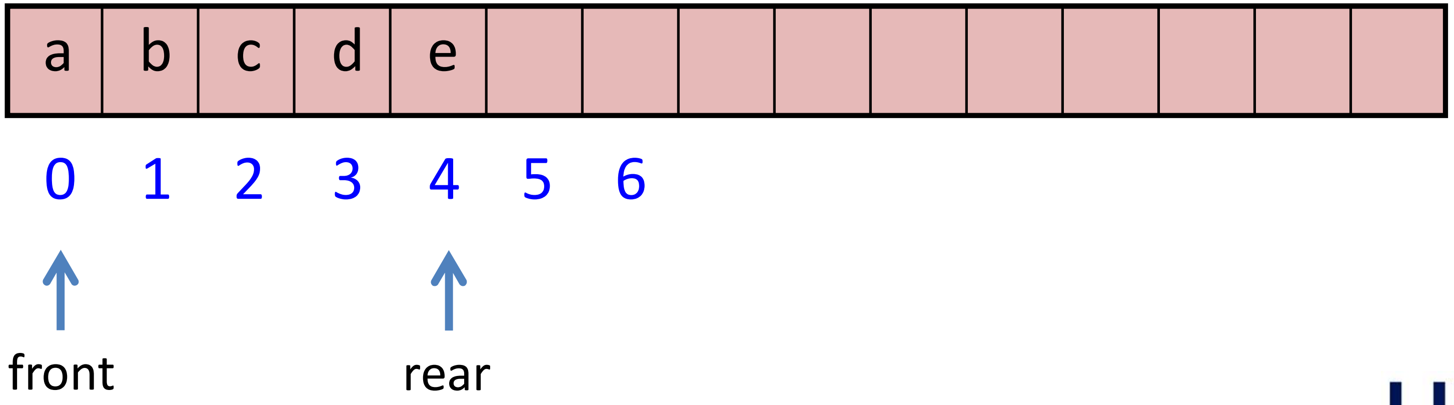
Simple Queue using Array

- Keep *rear* index only, first element must be at queue[0]
- Pop: delete queue[0] and shift elements to left
  - Θ(queue size) time
- Push: Θ(1) time



Improved Queue

- Keep both *front* and *rear* index
- When pop, *front* index increases
- When push, *rear* index increases
- When *rear* reaches to the rightmost location, all elements have to be shifted to the left

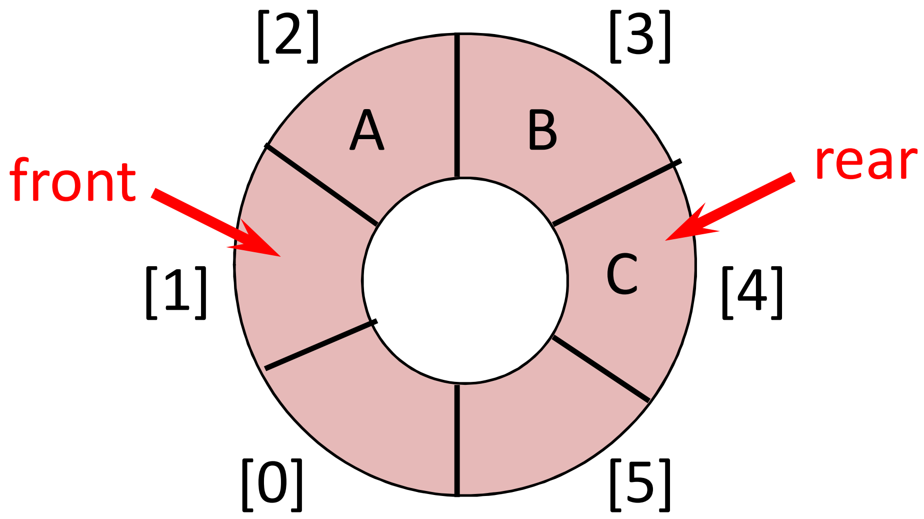


Improved Queue

- Empty
  - *front* == *rear*
- Full
  - *front* == 0 AND *rear* == *capacity-1*
- Resize queue when queue is full and you want to push a new element
  - Or wait until an element is popped

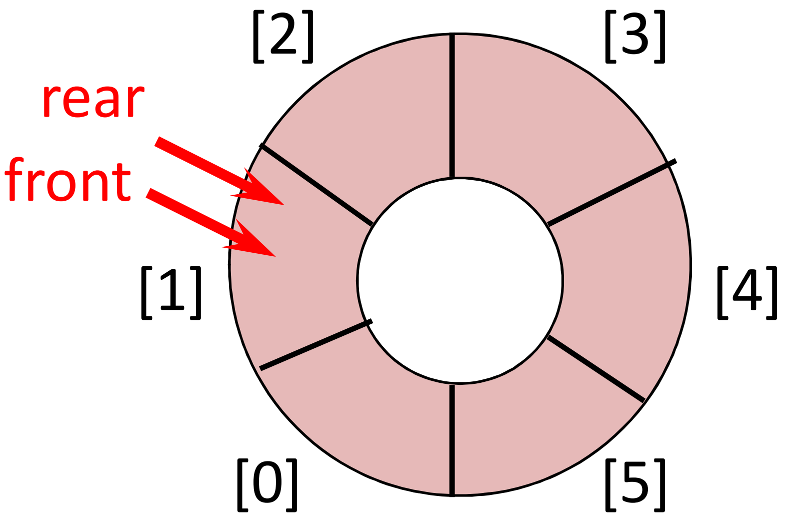
Circular Queue

- Keep both *front* and *rear* index
- *front* is one position counterclockwise from the first element
- *rear* is the position of the last element



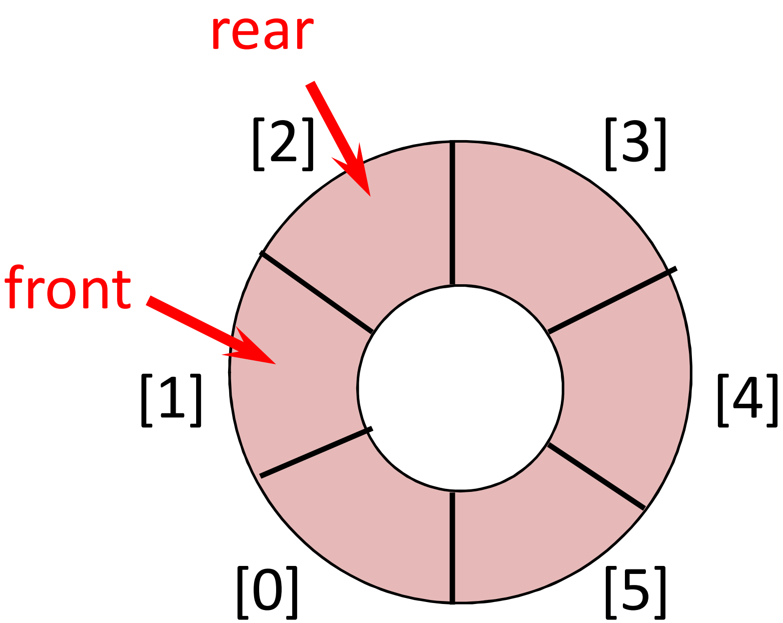
Circular Queue

- Empty: *front* == *rear*



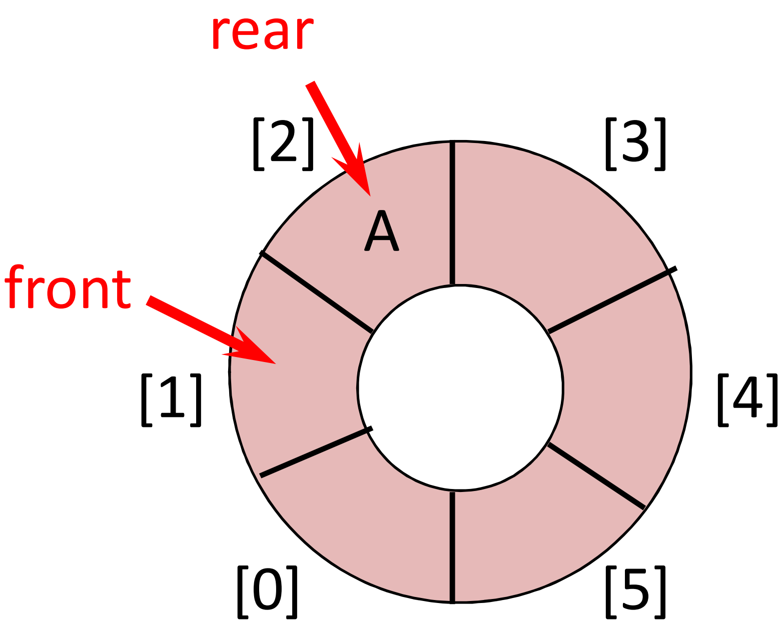
Circular Queue

- Add element (push)
  - –Move *rear* one clockwise



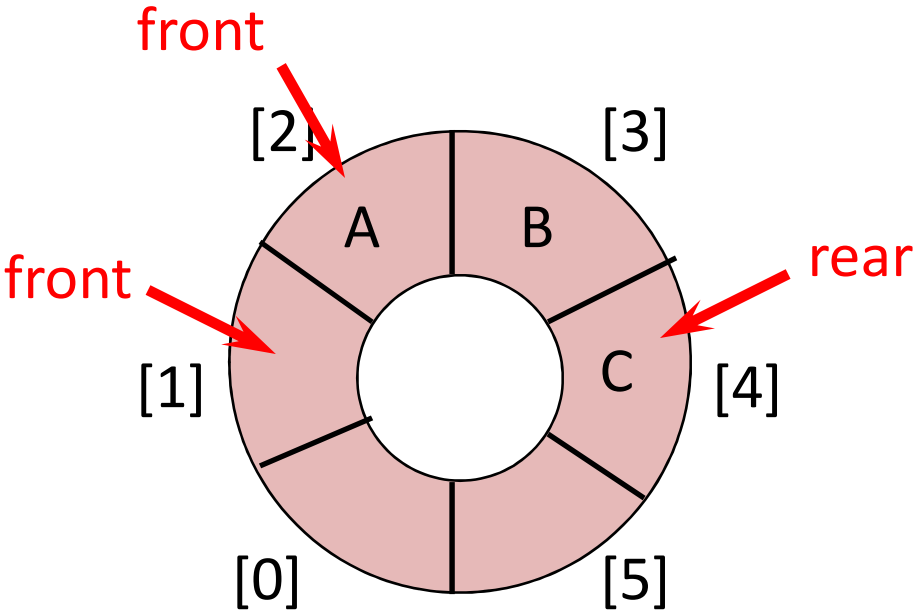
Circular Queue

- Add element (push)
  - –Move *rear* one clockwise
  - –Put into queue[rear]



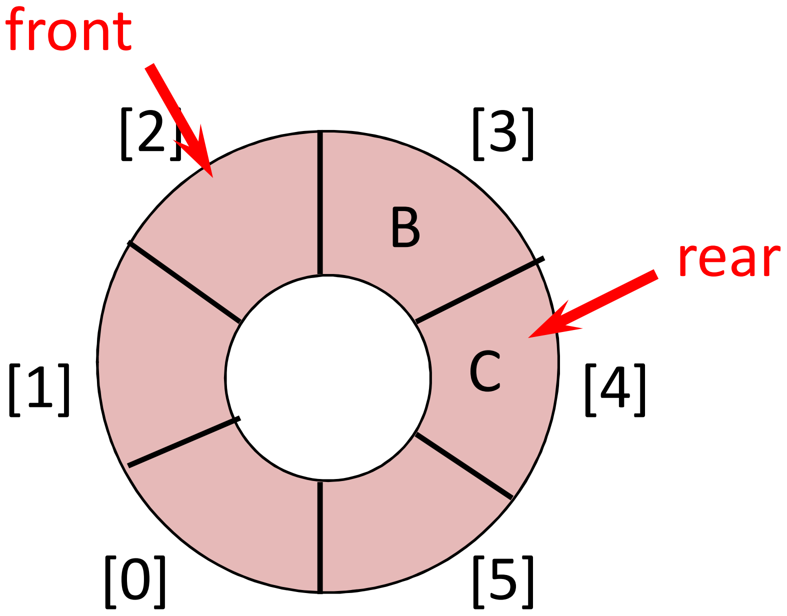
Circular Queue

- Delete element (pop)
  - –Move *front* one clockwise



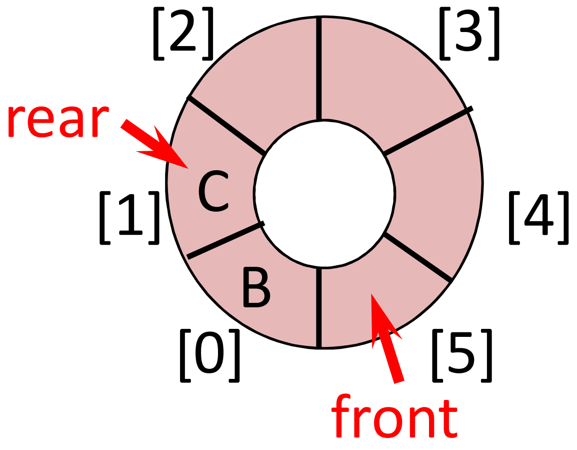
Circular Queue

- Delete element (pop)
  - –Move *front* one clockwise
  - –Remove queue[front]



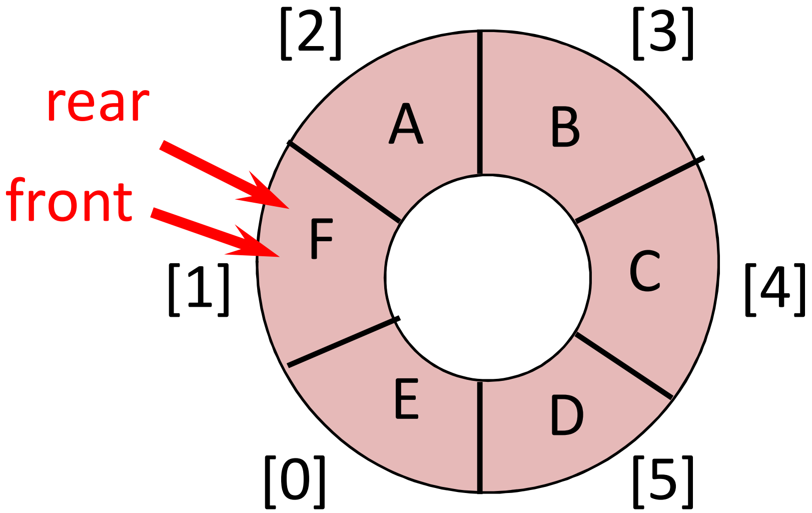
Circular Queue

- Push / pop can be done in Θ(1)
  - –No shifting elements
- Move *front* and *rear* clockwise
  - –front = (front +1) % capacity
  - –rear = (rear+1) % capacity
- Access *front* element
  - –queue[(front+1)%capacity]



Circular Queue

- Full queue
  - *front* == *rear,* same as empty
  - How do we distinguish?



Circular Queue

- Full queue
  - *front* == *rear,* same as empty
  - How do we distinguish?
    - Pop makes front==rear then empty
    - Push makes front==rear then full
    - or keeping track of queue size
      - size++ when push
      - size— when pop
      - if size == capacity then full
      - if size == 0 then empty

Outline

- Stacks & Queues
  - –Stack ADT
  - –Linear queue
  - –Circular queue

- Examples
  - –Queue using Stacks
  - –Evaluation of expression

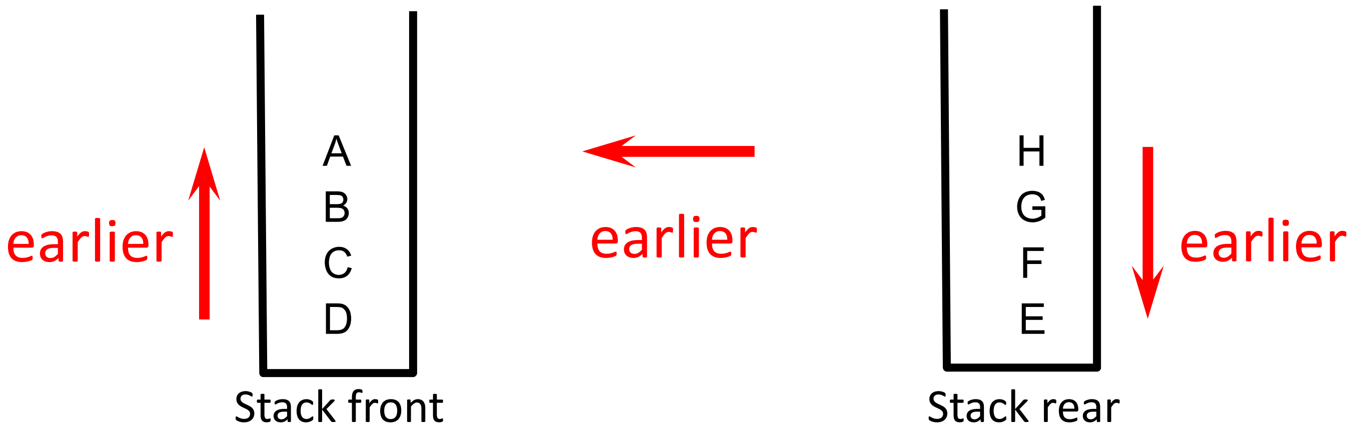
Queue using Stacks

- Assume you only have stack class
- Can you implement a queue using stacks?

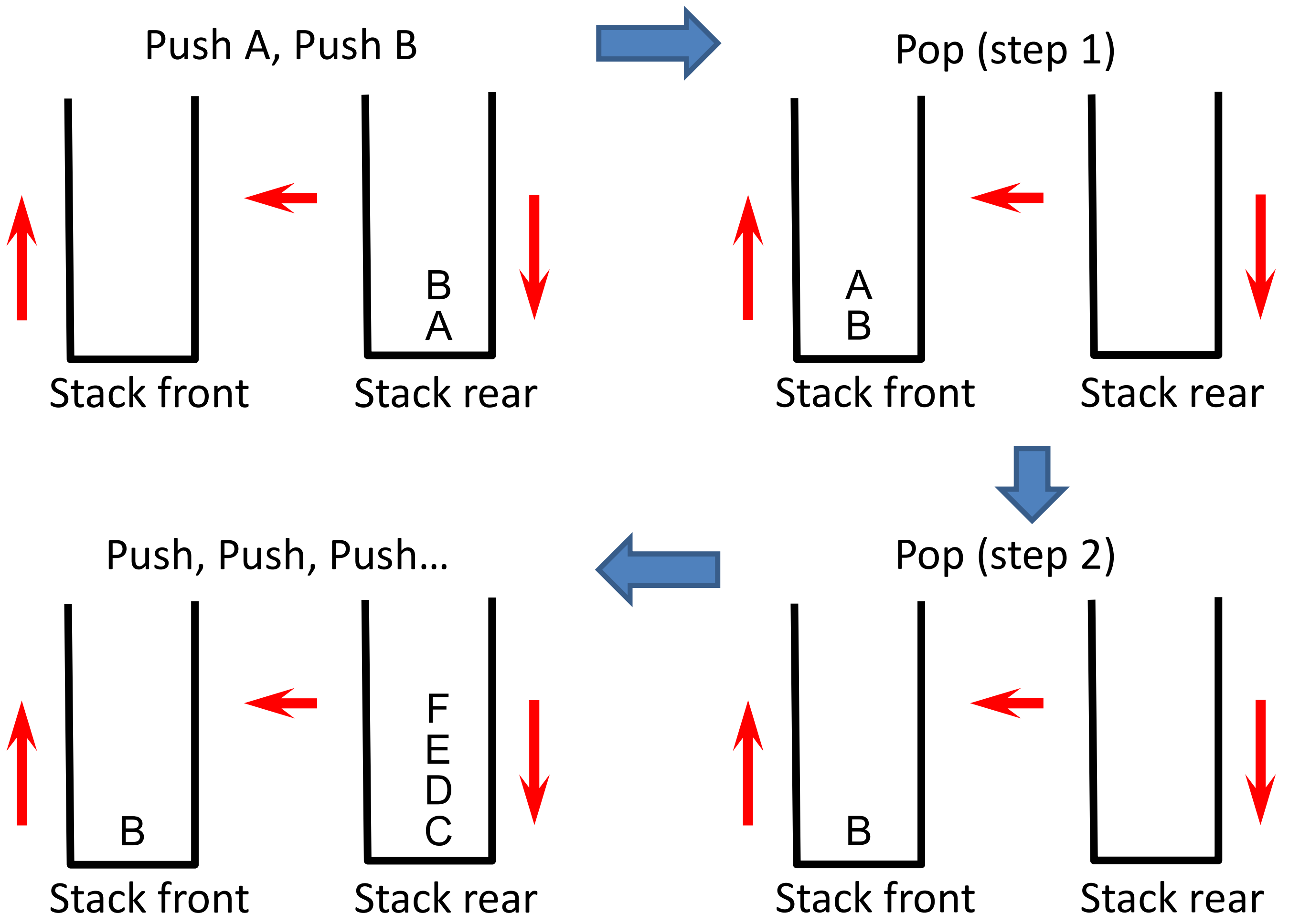
Stack front
Stack rear

Intuitions

- Elements in stack front came earlier
- Stack front: last-in at the bottom (handle pop)
- Stack rear: first-in at the bottom (handle push)



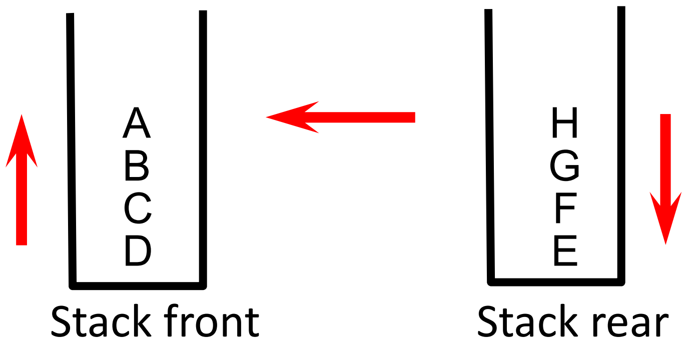
Example


Push A, Push B
B
A
Stack front
Stack rear
Pop (step 1)
A
B
Stack front
Stack rear
Pop (step 2)
B
Stack front
Stack rear
Push, Push, Push…
B
F
E
D
C
Stack front
Stack rear

Queue using Stacks

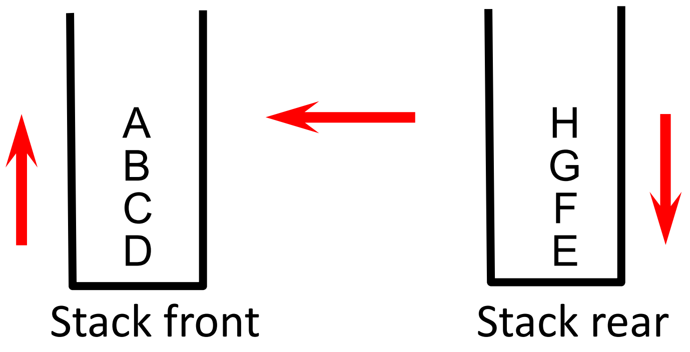
```
class stack {
public:
    Node& Top();
    void Pop();
    void Push(Node& n);
    bool IsEmpty();
};

void
Queue::Push(Node &n)
{
    rear.Push(n);
}
```



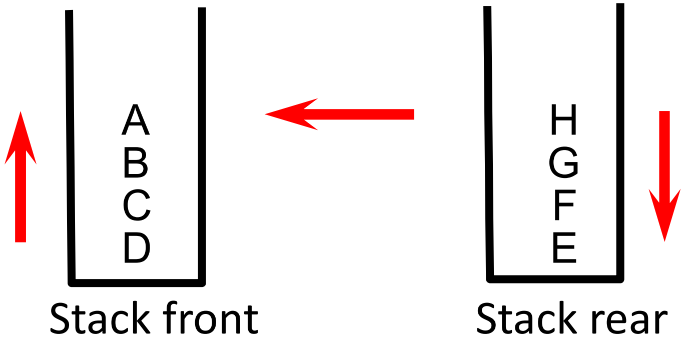
Queue using Stacks

```
void
Queue::Pop()
{
    Front();
    front.pop();
}
```



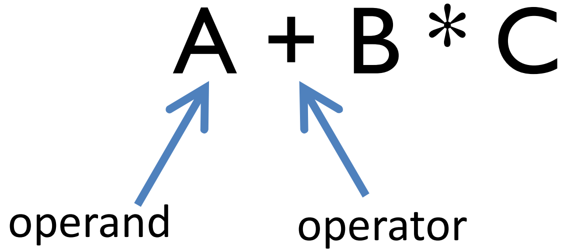
Queue using Stacks

```
Node&
Queue::Front()
{
     if(front.IsEmpty())
     {
          while(!rear.IsEmpty())
          {
               front.push(rear.Top());
               rear.Pop();
          }
     }
     return front.Top();
}
```



Evaluation of Expression

- Expression



A + B * C

operand

operator

- Infix notation
  - Operator is placed between two operands
    - e.g., A + B, C + D * E
  - 48/2(9+3) is not a complete infix expression as there is missing * between 2 and (9+3)

Evaluation of Expression

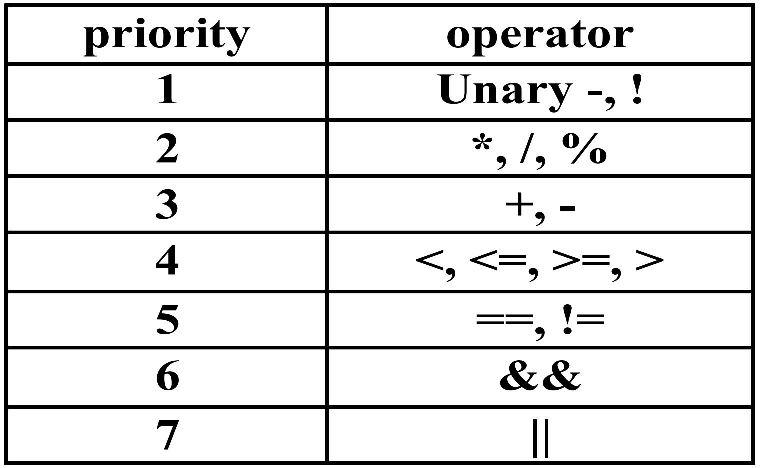
- How do we evaluate expression?

  X = (A+B)*C-D/E

- Rules
  - Parenthesis has the highest priority
  - Follow operator's priority
  - If operators have same priority, the left one has higher priority than the right one

Evaluation of Expressions

- Priority of operators



priority	operator
1	Unary -, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	==, !=
6	&&
7	||

Various Notations

- Infix: more human readable
  - A*B/C
- Postfix: more friendly to computer
  - AB*C/
- Prefix
  - /*ABC

Postfix Notation

- Benefits
  - –No parenthesis
  - –No operator priority
  - –Simple to evaluate (left to right scan)
- We can convert between infix and postfix
  - –Infix: A/B-C+D*E-A*C
  - –Postfix: AB/C-DE*+AC*-

Evaluate Postfix Notation

AB/C-DE*+AC*-

1. Push operands to stack until operator is reached
2. Once operator is reached, pop two operands from stack and apply the operator
3. Push the result back to stack

Evaluate Postfix Notation

Postfix : AB/C-DE*+AC*-

Operation	Postfix
$T_1 = A / B$	T_1 C – D E * + A C * -
$T_2 = T_1 – C$	T_2 D E * + A C * -
$T_3 = D * E$	T_2 T_3 + A C * -
$T_4 = T_2 + T_3$	T_4 A C * -
$T_5 = A * C$	T_4 T_5 –
$T_6 = T_4 – T_5$	T_6

Infix : A/B-C+D*E-A*C

Evaluation Algorithm

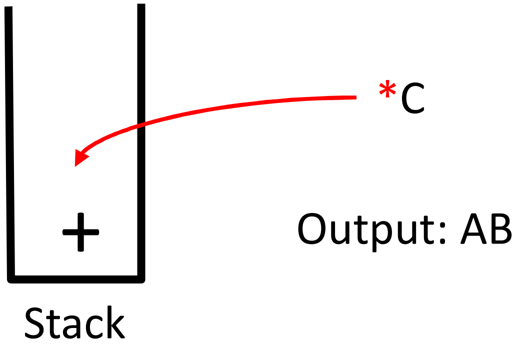
```
void eval(expression e)
// Last token is '#’
{
   Stack<token> stack;
   token x;
   for(x = NextToken(e); x != '#'; x = NextToken(e))
      if (x is an operand) stack.Push(x) // push
      else { // operator
         Pop two operands from stack;
         Push the result back to stack;
      }
}
```

Infix to Postfix Conversion

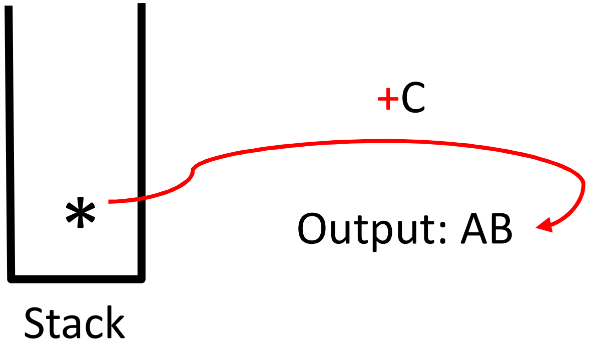
- Algorithm
  - Left to right, output operands & stack operators
  - The priority of incoming operator is compared to the priority of top operator in stack
    - If higher, stack the higher-priority operator
    - If not, pop the top operator and output

Two Examples

- A+B*C ⟶ ABC*+

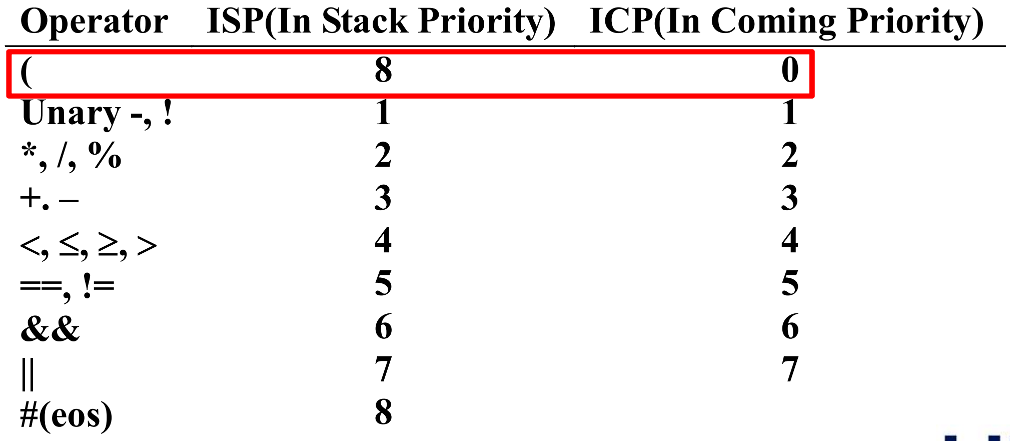


- A*B+C ⟶ AB*C+



Infix to Postfix Conversion

- Algorithm
  - Left to right, output operands & stack operators
  - The priority of incoming operator is compared to the priority of top operator in stack
  - Right parenthesis pops all operators above left parenthesis



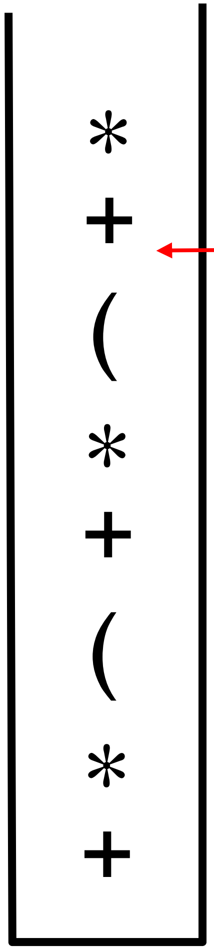
Operator	ISP(In Stack Priority)	ICP(In Coming Priority)
(	8	0
Unary -, !	1	1
*, /, %	2	2
+. –	3	3
<, ≤, ≥, >	4	4
==, !=	5	5
&&	6	6
||	7	7
#(eos)	8	

small number = higher priority

Infix to Postfix Conversion

small number = higher priority

Operator	ISP(In Stack Priority)	ICP(In Coming Priority)
(	8	0
Unary -, !	1	1
*, /, %	2	2
+. –	3	3
<, ≤, ≥, >	4	4
==, !=	5	5
&&	6	6
||	7	7
#(eos)	8	



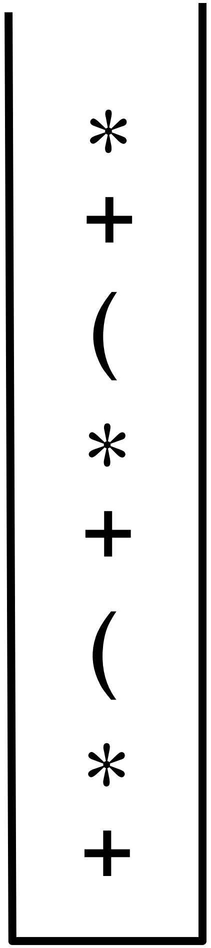
Infix :A+B*(C+D*(E+F*G))

left-to-right scan

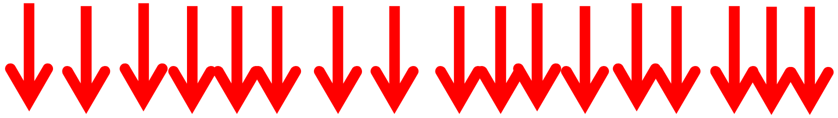
Postfix: A B C D E F G * + * + * +

Infix to Postfix Conversion

Operator	ISP(In Stack Priority)	ICP(In Coming Priority)
(	8	0
Unary -, !	1	1
*, /, %	2	2
+. –	3	3
<, ≤, ≥, >	4	4
==, !=	5	5
&&	6	6
||	7	7
#(eos)	8	



*
+
(
*
+
(
*
+



Infix :A+B*(C+D*(E+F*G))

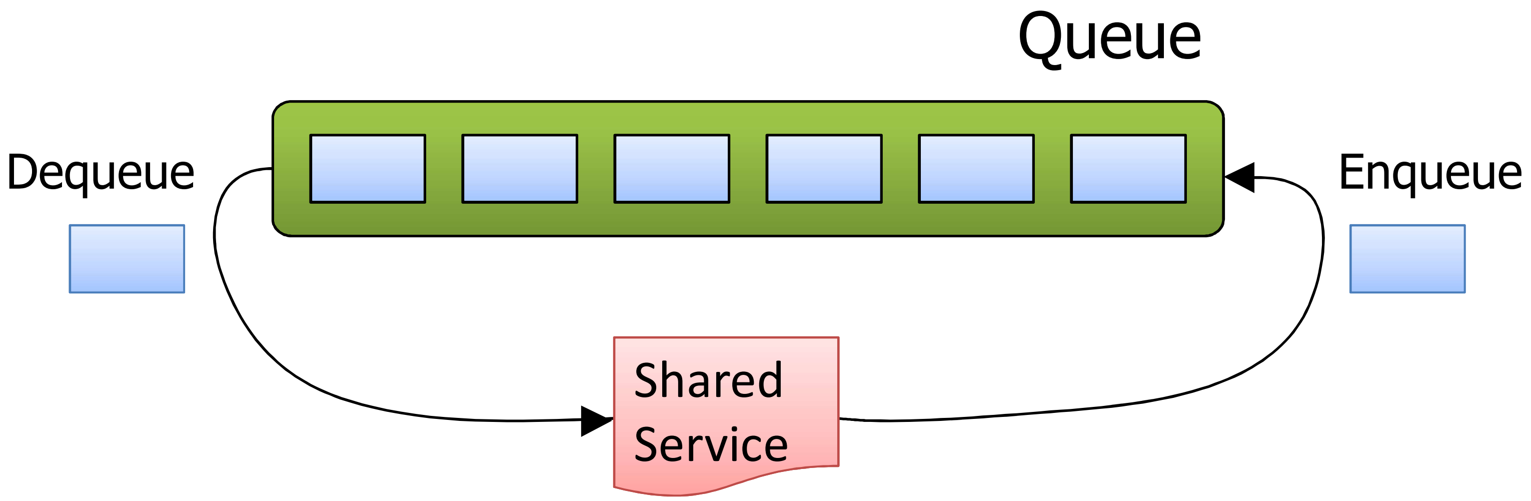
Postfix: A B C D E F G * + * + * +

```
void postfix(expression e)
{
   // Assume that the last token in e is # and
   // # is used at the bottom of the stack
   Stack<token> stack;
   stack.Push('#');
   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(); // unstack ‘(‘
      }
      else { // x is operator
         for (;isp(stack.Top())<=icp(x);stack.Pop())
             cout << stack.Top();
         stack.Push(x);
      }
   }

   // empty stack
   while(!stack.IsEmpty()) cout << stack.Top(), stack.Pop();
}
```

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.front(); Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Questions?