CSE221

Lecture 4:
Stacks and Queues

Hyungon Moon

Slide credits: The textbook authors, Won-ki Jeong,
Tsz-Chiu Au, and Myeongjae Jeon

UNIST
ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Outline

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

Outline

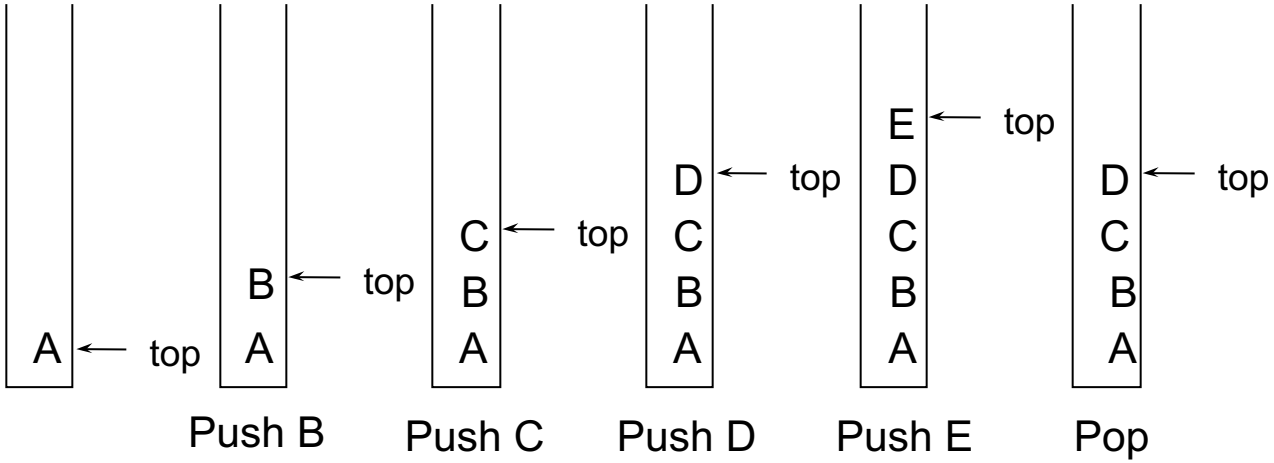
- Stacks & Queues
  - Stack ADT
  - Linear queue
  - Circular queue
- Examples
  - Queue using Stacks
  - Maze problem
  - Evaluation of expression

Stack

- Special case of ordered (linear) list
- Data insertion (push) and deletion (pop) are happened 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

- Push

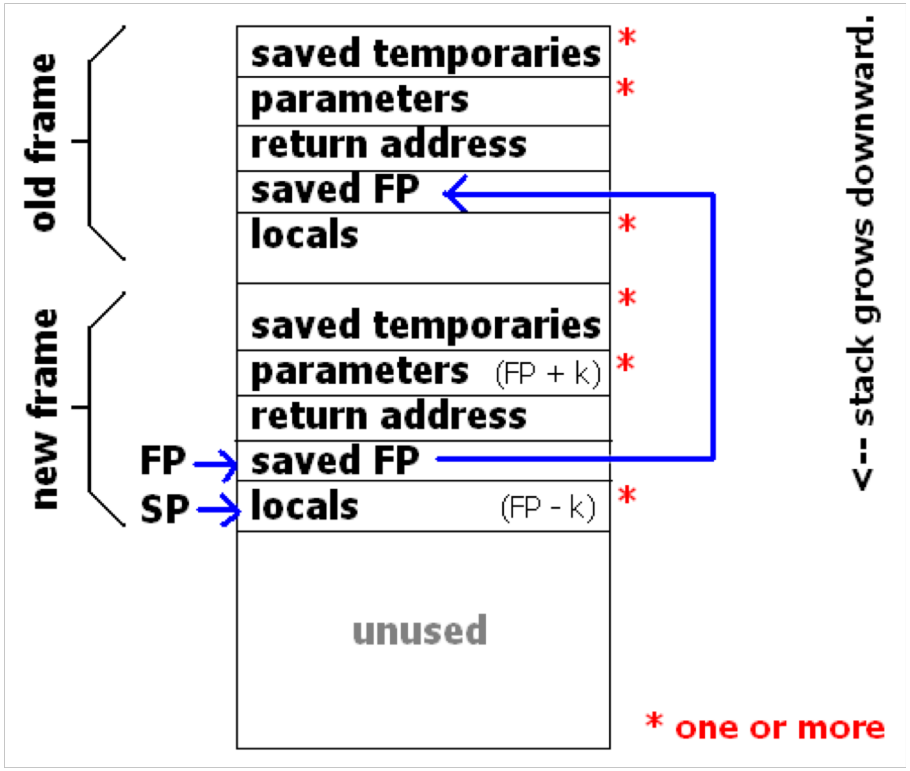
```
template <class KeyType>
void Stack<KeyType>::Push(const KeyType& x)
{
   if (IsFull()) ChangeSize();
   stack[++top] = x;
}
```

- Pop

```
template <class KeyType>
void Stack<KeyType>::Pop()
{
   if (IsEmpty()) return 0;
   stack[top--].~KeyType(); // destructor
}
```

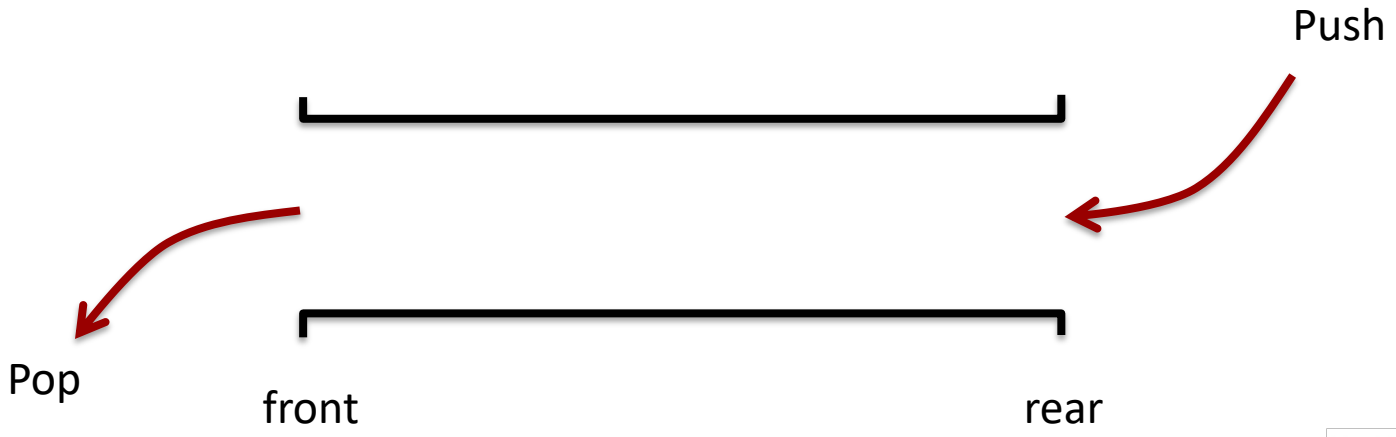
Example: System Stack

- Locals, temporaries, parameters, return address, frame pointer



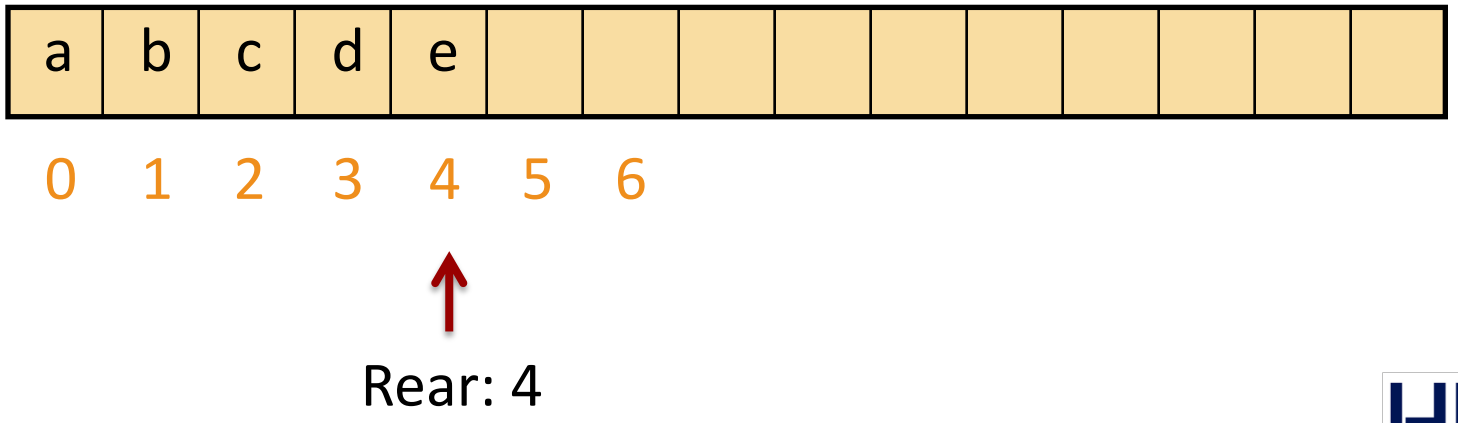
Queue

- 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)



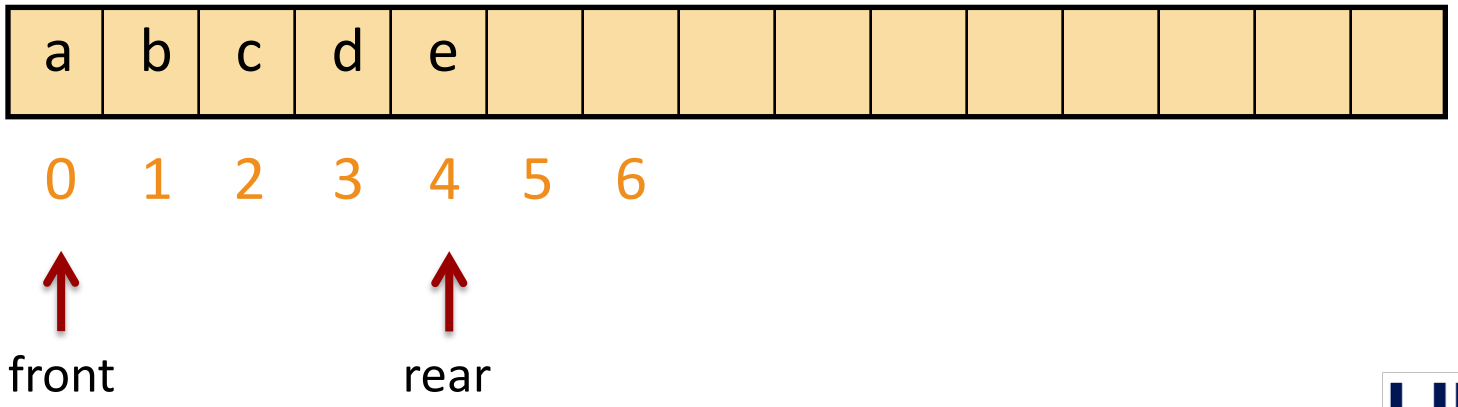
Simple Queue using Array

- Keep *rear* index only, first element must be at queue[0]
- Pop: delete queue[0] and shift elements to left
  - queue size times of data movements.
- Push: one data copy (or move).



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

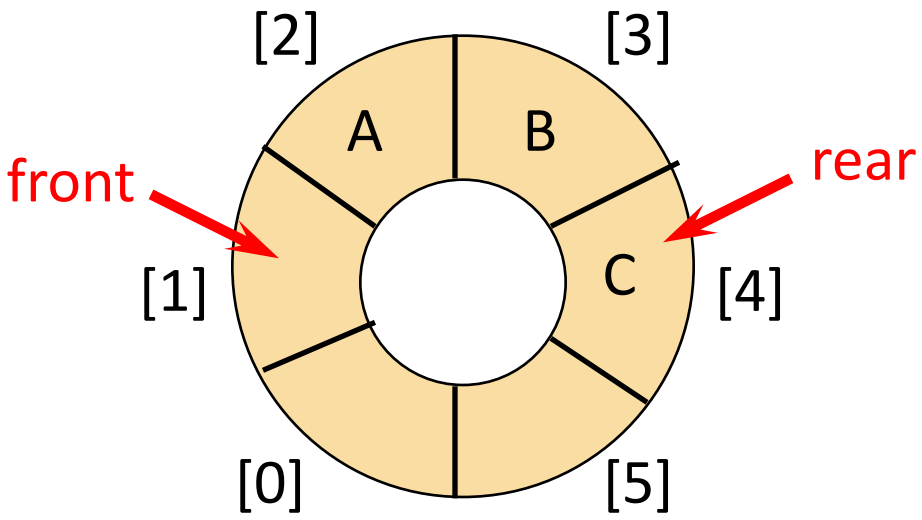


Improved Queue

- Empty
  - *front* == *rear*
- Full
  - *front* == 0, *rear* == *capacity-1*
- Doubling queue when queue is full and push a new element

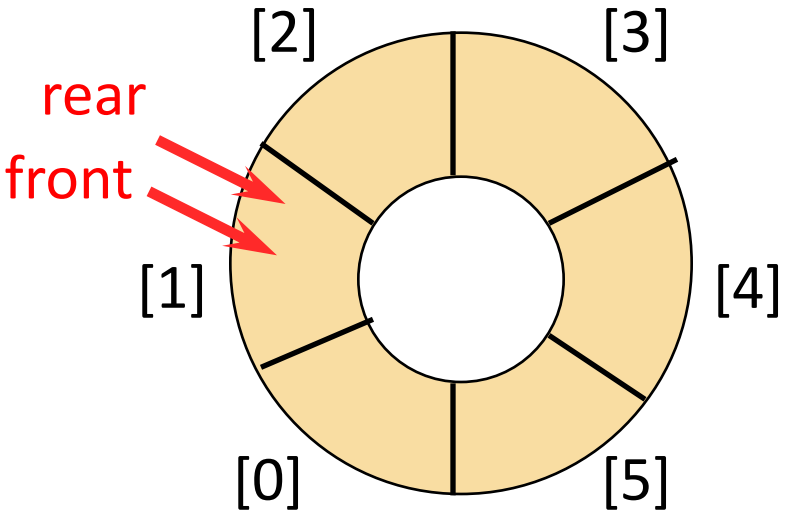
Circular Queue

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



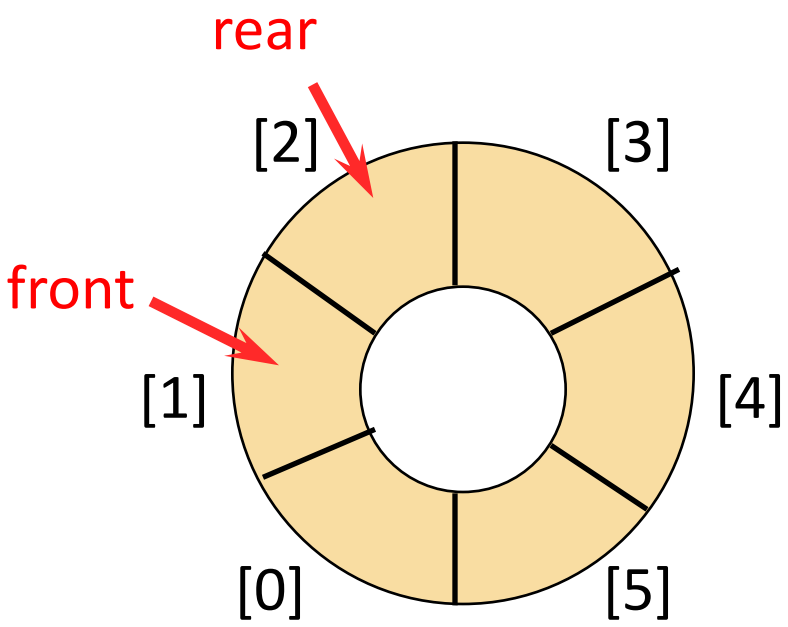
Circular Queue

- Empty: *front* == *rear*



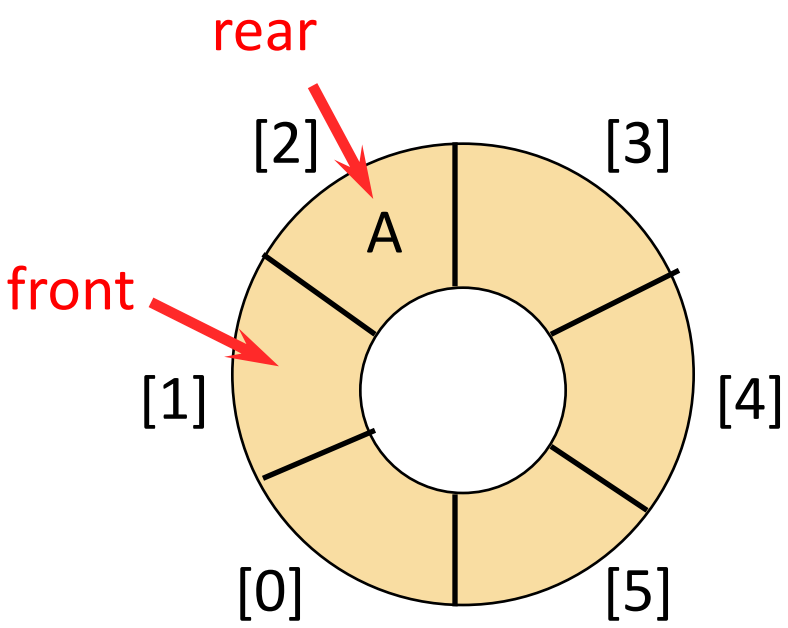
Circular Queue

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



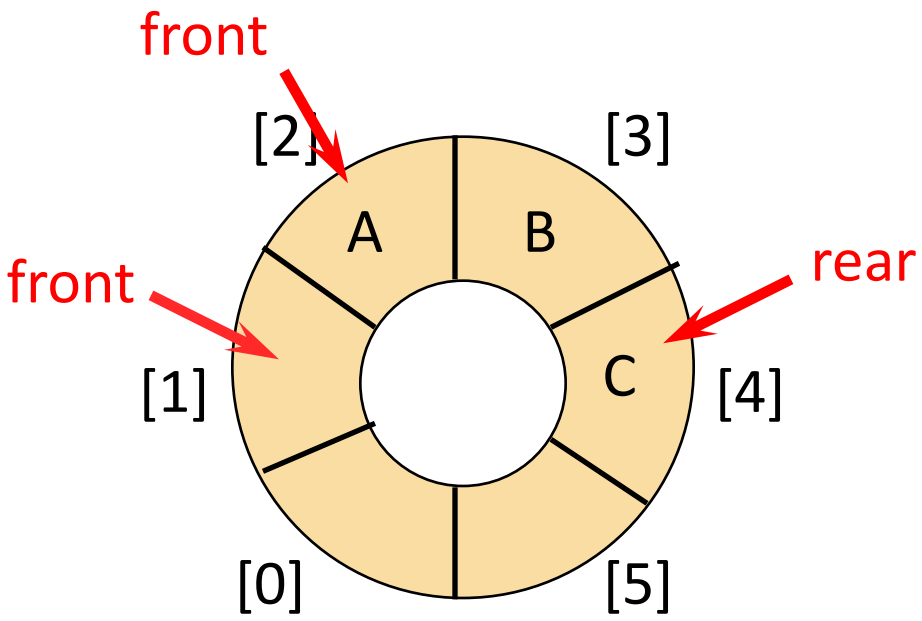
Circular Queue

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



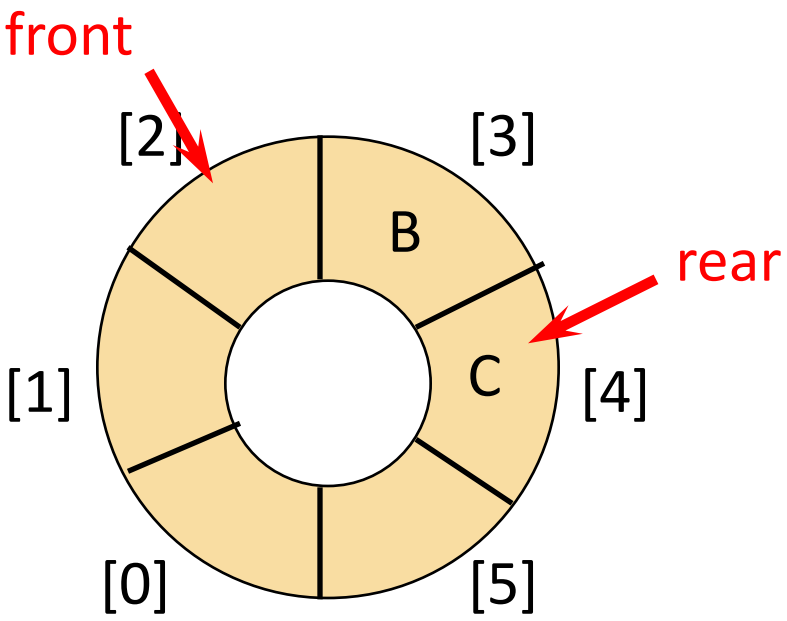
Circular Queue

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



Circular Queue

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

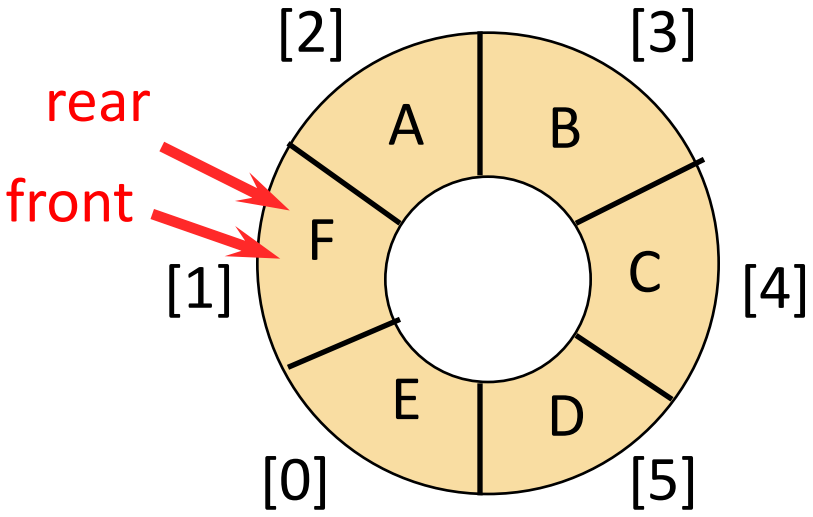


Circular Queue

- Push / pop can be done with one data copy.
  - No shifting elements
- Access *front* element
  - queue[(front+1)%capacity]
- Move *front* and *rear* clockwise
  - front = (front +1) % capacity
  - rear = (rear+1) % capacity

Circular Queue

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

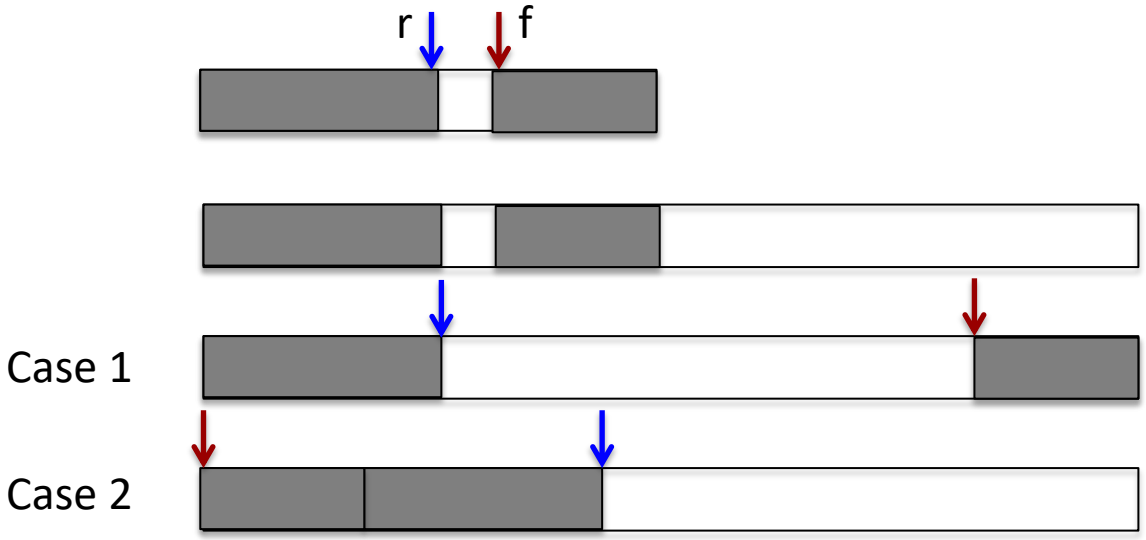


Circular Queue

- 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

Doubling Circular Queue

- When queue is full
- Need to shifting element
  - Implementation of circular queue is using linear array
- Two configurations

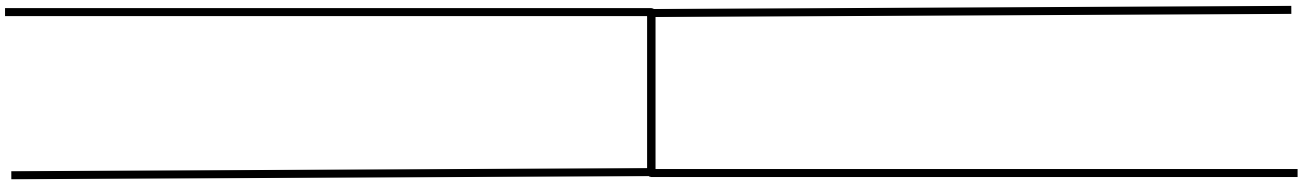


Outline

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

Queue using Stacks

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



Queue using Stacks

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

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

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();
    (_____________________________________);
}
```

Queue using Stacks

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

Queue using Stacks

```
Node&
Queue::Front()
{
     if(_________________________)
     {
          while(_____________________)
          {
               (_____________________________________);
               (_____________________________________);
          }
     }
     return (_____________________________________);
}
```

Queue using Stacks

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

Maze Problem

Maze Problem

- 2D array to model the map
  - 0: opened (can move), 1: blocked (cannot move)
  - int maze[i][j]

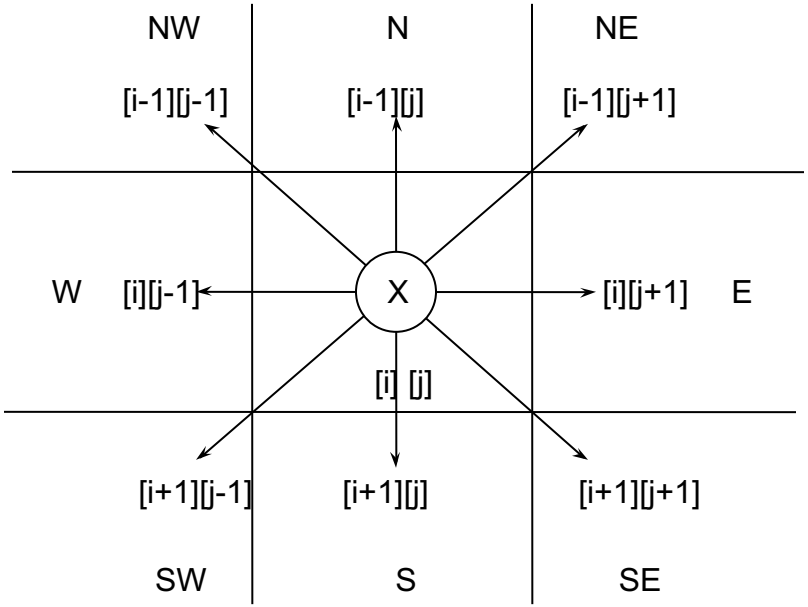
Enter

```
0 1 0 0 0 1 1 0 0 0 1 1 1 1 1
1 0 0 0 1 1 0 1 1 1 0 0 1 1 1
0 1 1 0 0 0 0 1 1 1 1 0 0 1 1
1 1 0 1 1 1 1 0 1 1 0 1 1 0 0
1 1 0 1 0 0 1 0 1 1 1 1 1 1 1
0 0 1 1 0 1 1 1 0 1 0 0 1 0 1
0 0 1 1 0 1 1 1 0 1 0 0 1 0 1
0 1 1 1 1 0 0 1 1 1 1 1 1 1 1
1 1 0 0 0 1 1 0 1 1 0 0 0 0 0
0 0 1 1 1 1 1 0 0 0 1 1 1 1 0
0 1 0 0 1 1 1 1 1 0 1 1 1 1 0
```

Exit

Maze Problem

- Allowable moves




```
struct offsets
{
   int a, b;
};

enum directions {N, NE, E, SE,
                 S, SW, W, NW};
offsets move[8];
```

q	move[q].a	move[q].b
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

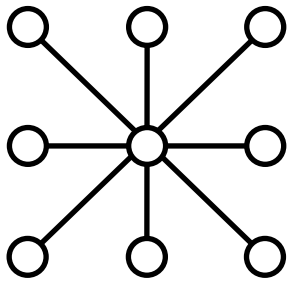
Table of moves

Move from x[I][j] to SW (x[g][h])
g = I + move[sw].a;
h = j + move[sw].b;

Maze Problem

- Items
  - (x,y,dir) : at (x,y), try to move dir
- Strategy
  - Try each direction unless it is not yet visited
  - When moving forward, use stack to store the current location and the next search direction
  - If all directions are blocked, pop the previous location and next search direction to continue

```
void path(int m, int p) // exit at (m,p)
// Output a path if exists
{
   mark[1][1]=1; // start at (1, 1)
   stack<items> stack(m*p);
   items temp;
   temp.x = 1; temp.y = 1; temp.dir = E;
   stack.Add(temp);
   while (!stack.IsEmpty()) // stack is not empty
   {
      temp = *stack.Top();
      stack.Pop(); // delete from stack
      while (d < 8) // move forward
      {
         int g = i + move[d].a; int h = j + move[d].b;
         if ((g == m) && (h == p)) { // reach exit
            cout << stack; // print path
            cout << i << " " << j << endl; // last two squares in the path
            cout << m << " " << p << endl;
            return;
         }
         if ((!maze[g][h]) && (!mark[g][h])) { // not a wall & not explored
            mark[g][h] = 1;
            temp.x = i; temp.y = j; temp.dir = d+1;
            stack.Push(temp); // store current location and next direction
            i = g; j = h; d = N; // move to (g, h)
         }
         else d++; // try next direction
      }
   }
   cout << "no path in maze" << endl;
}
```



reset d to N (= 0, enum North)

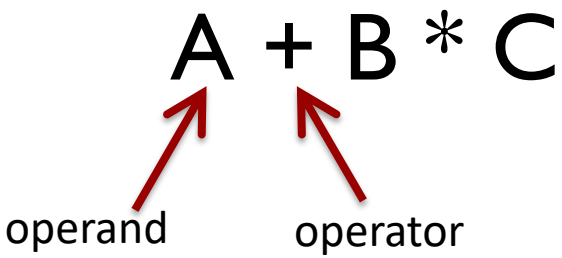
O(?)

Maze Algorithm

- For each element in the stack
  - While loop visits its 8 neighborhood
  - If neighbor is already visited, skip
  - If neighbor is not visited, store next node to stack and move to that node
    - When returns, visit next node by popping stack
- Move as far as possible before checking neighbors

Evaluation of Expression

- Expression



- 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 due to 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	||

- Example

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

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

Notation

- Infix
  - A*B/C
- Postfix
  - AB*C/
- Prefix
  - /*ABC

Postfix Notation

- Benefits
  - No parenthesis
  - No operator priority
  - Simple to evaluate (left to right scan)
- Example
  - Infix: A/B-C+D*E-A*C
  - Postfix: AB/C-DE*+AC*-

Evaluate Postfix Notation

- Push to stack until operator is reached
- Pop two operands from stack
- Push the result back to stack

```
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;
      }
}
```

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 to Postfix Conversion

- Steps
  - Fully parenthesize the expression
  - Move all operators to the right parenthesis to replace them
  - Delete all parenthesis

A/B-C+D*E-A*C
= ((((A/B)-C)+(D*E))-(A*C))
= AB/C-DE*+AC*-

Infix to Postfix Conversion

- Algorithm
  - Left to right, output operands & stack operators
  - Incoming operator must have higher priority than 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

```
void postfix(expression e)
{
   Stack<token>stack;
   token x, y;
   stack.Add('#');
   for (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();
      }
      else { // x is operator
         while(isp(stack.Top())<=icp(x)){
              cout << stack.Top();
              stack.Pop();
          }
         stack.Push(x);
      }
   }

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

O(?)

Infix to Postfix Conversion

- A+B*C → ABC*+

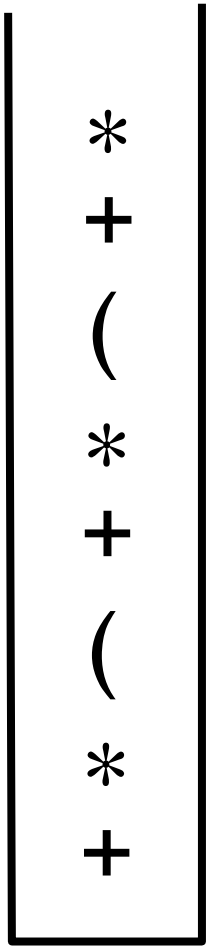
- A*B+C → AB*C+

Infix to Postfix Conversion

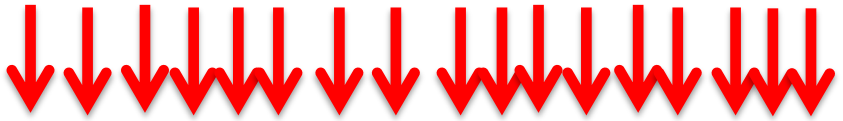
- A*(B+C)*D

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

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 * + * + * +

Questions?