Stacks and Queues (chapter 3)

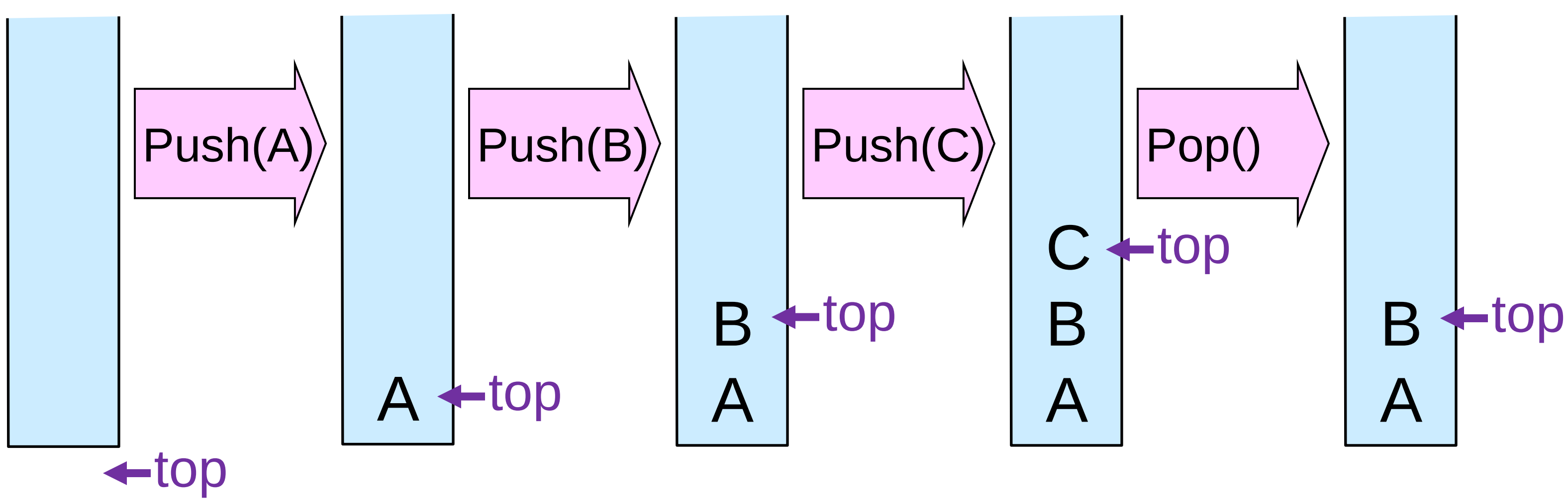
- Stacks: ADT, representation, and implementation
- Queues: ADT, representation, and implementation
- Representative problems (for stacks):
  - Maze
  - Expression evaluation

Note: Stacks and queues are just ADTs with specific operations. There are multiple possible representations. In this chapter, their representations are based on *arrays*. In chapter 4, we will represent them using *linked lists*.

Stacks

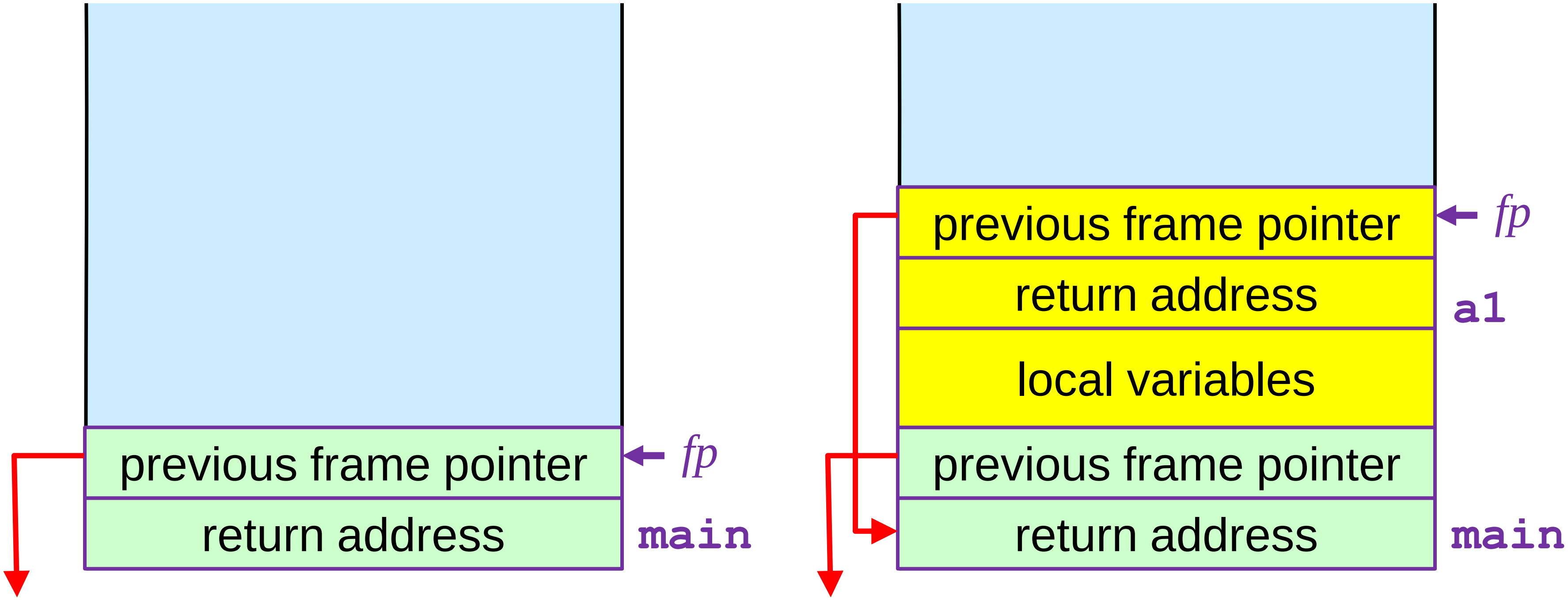
Stacks are <u>**First-In-Last-Out (FILO)**</u> lists.

- **Push**: Add an element to the top of the stack.
- **Pop**: Remove an element from the top of the stack.
- **Top**: To get the top element of the stack.
- A pointer "**top**" is used to remember the current top element of the stack.



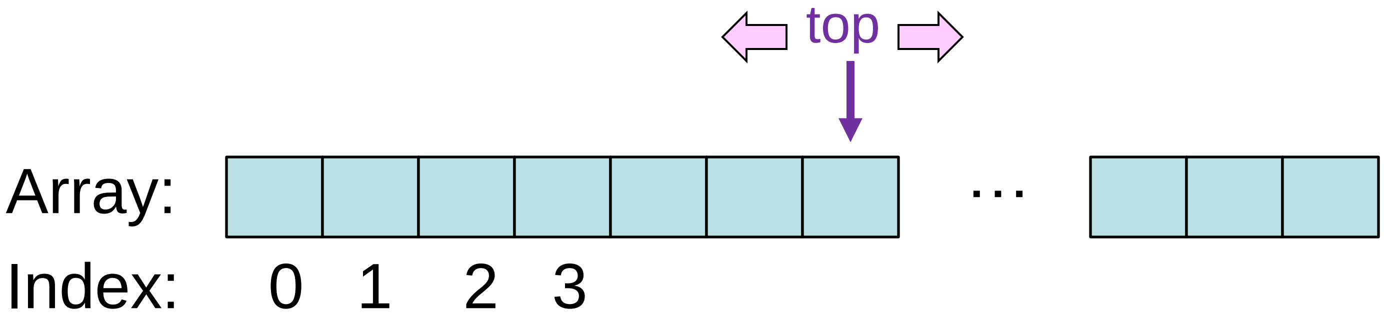
Example: Function Call Stack

The OS uses a stack to store local variables and function calls for each thread.



Stack Representation

The easiest way is to use a fixed-size array.



Stack ADT / Class Template

```
template <class T> class Stack
{ // a finite ordered list with zero or more elements
private:
  T* stack; // array for stack elements
  int top; // index of the top element
  int capacity; // allocated space
public:
  Stack(int initCapacity=10);
  // Initialization with the given capacity
  bool IsEmpty() const;
  // Return whether the stack is empty
  T& Top() const;
  // Return the top element
  void Push(const T& item);
  // Add 'item' to the stack
  void Pop();
  // Delete the top element
};
```

Note the use of references

Note the use of keyword **`const`**

Need to add in the implementation:

- destructor
- exception handling functions

Stack Implementation

```
template <class T>
Stack<T>::Stack(int initCapacity)
{
  stack = new T [initCapacity];
  capacity = initCapacity;
  top = -1;  // indicating an empty stack
}

template <class T>
bool Stack<T>::IsEmpty() const
{ return (top == -1); }

template <class T>
T& Stack<T>::Top() const
{
  if (IsEmpty()) { ... }; // exception handling
  return stack[top];
}
```

Stack Implementation

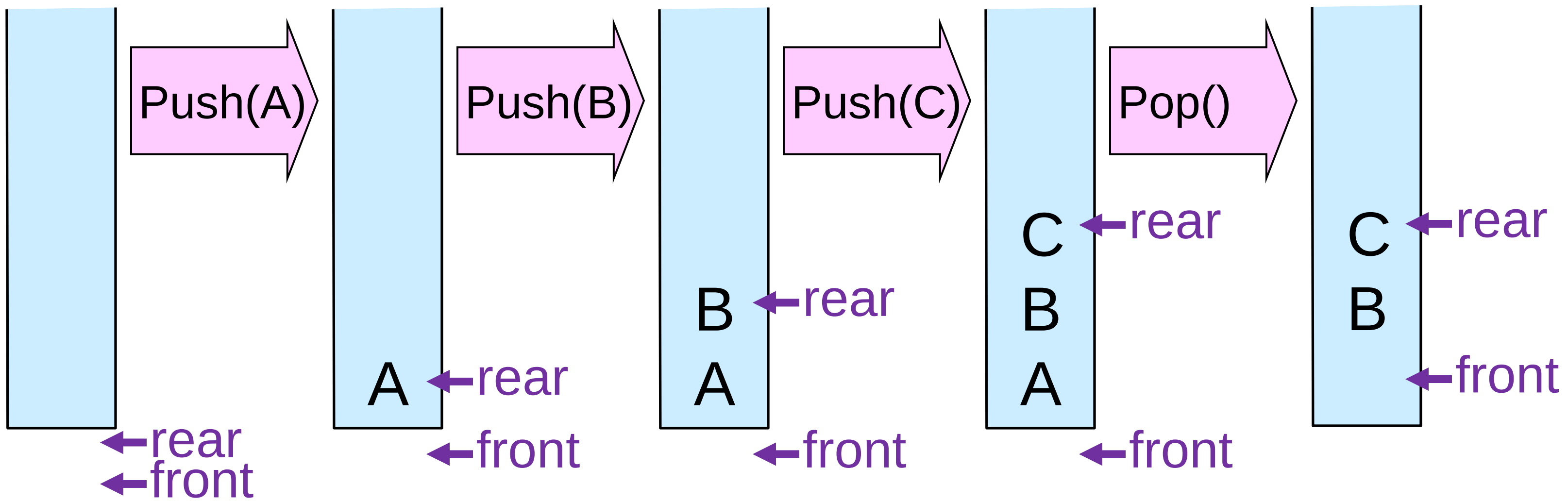
```
template <class T>
void Stack<T>::Push(const T& item)
{
  if (top == capacity-1) { // stack is full
    T* t = new T [capacity*2]; // double the capacity
    copy(stack, stack+top, t); // see textbook p.35
    delete [] stack;
    stack = t;
    capacity *= 2;
  }
  stack[++top] = item;
}

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

Queues

Queues are **First-In-First-Out (FIFO)** lists.

- **Push**: Add an element to the queue.
- **Pop**: Remove an element from the queue.
- **Front**: The element at the front end of the queue.
- **Rear**: The element at the rear end of the queue.
- Two pointers "**front**" and "**rear**" are used to remember the two ends of the queue.



Queue ADT / Class Template

```
template <class T>
class Queue
{ // a finite ordered list with zero or more elements
private:
  T* queue; // array for queue elements
  int front, rear;
  int capacity; // allocated space
public:
  Queue(int initCapacity=10);
  // Initialization with the given capacity
  bool IsEmpty() const;
  // Return whether the queue is empty
  T& Front() const;  // Return the front element
  T& Rear() const;  // Return the rear element
  void Push(const T& item); // Add 'item' to the queue
  void Pop();  // Delete the front element
};
```

Need to add in the implementation:
- destructor
- exception handling functions

Queue Implementation (Sequential)

```
template <class T>
Queue<T>::Queue(int initCapacity)
{
  queue = new T [initCapacity];
  capacity = initCapacity;
  front = rear = -1;  // indicating an empty queue
}
```

`front` is one position before the first element
`rear` is the position of the last element

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

Queue Implementation (Sequential)

```
template <class T>
T& Queue<T>::Front() const
{
  if (IsEmpty()) { ... }; // exception handling
  return queue[front+1];
}

template <class T>
T& Queue<T>::Rear() const
{
  if (IsEmpty()) { ... }; // exception handling
  return queue[rear];
}
```

Queue Implementation (Sequential)

```
template <class T>
void Queue<T>::Push(const T& item)
{
  if (rear == capacity-1) { // queue is full
    // code to double the capacity here
    // update front and rear
  }
  queue[++rear] = item;
}

template <class T>
void Queue<T>::Pop()
{
  if (IsEmpty()) { ... }; // exception handling
  front++;
}
```

Example Sequential Queue

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Queue is empty
-1	0	J1				J1 is added
-1	1	J1	J2			J2 is added
-1	2	J1	J2	J3		J3 is added
0	2		J2	J3		J1 is deleted
1	2			J3		J2 is deleted

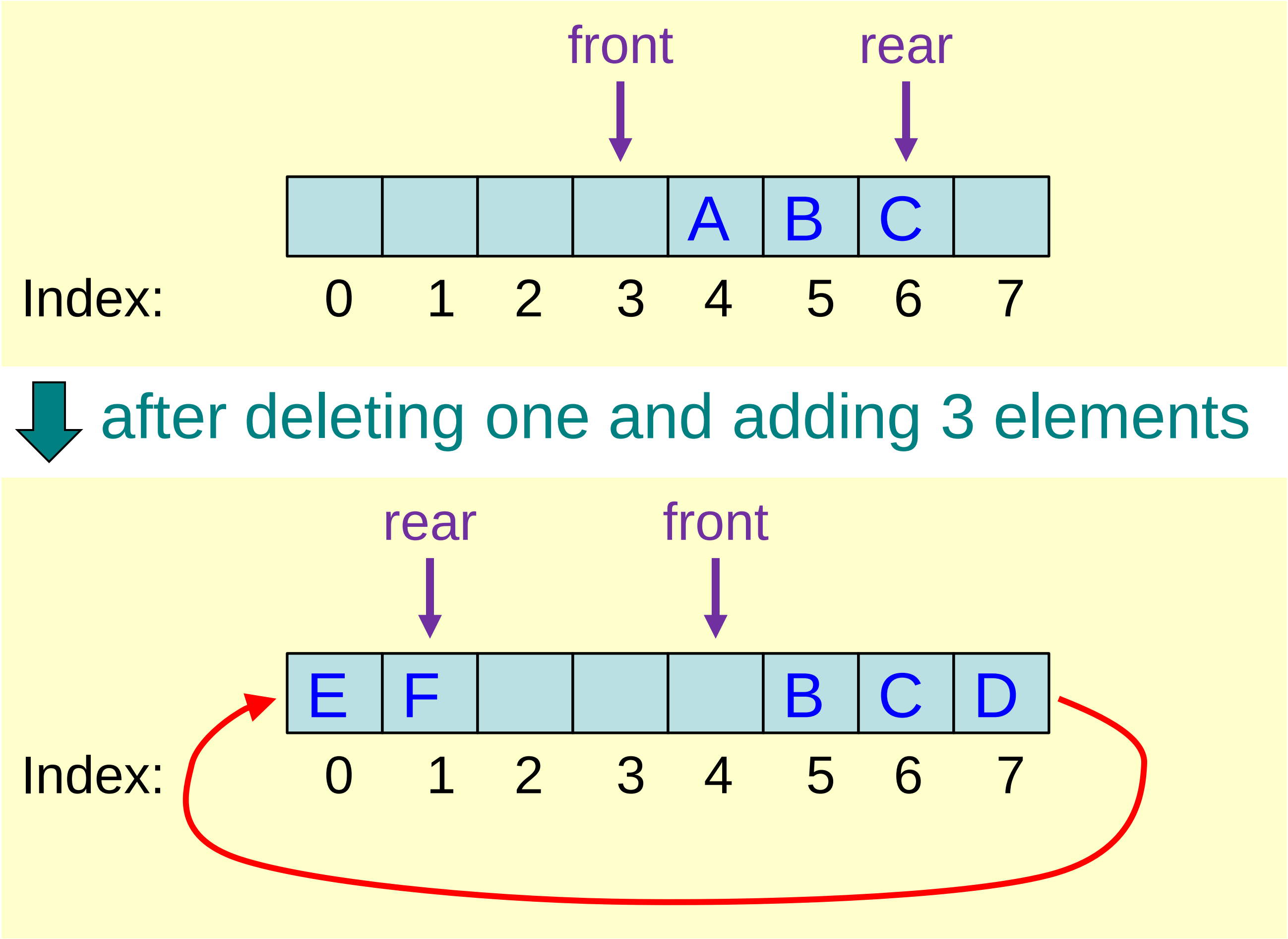
Issues:

- The spaces before `front` are wasted.
- Unnecessary array resizing when `rear` reaches the end of the allocated space.

Circular Queue

Idea: Allow the wraparound of available spaces.

Example with an 8-space queue:



Queue Implementation (Circular)

```
template <class T>
Queue<T>::Queue(int initCapacity)
{
  queue = new T [initCapacity];
  capacity = initCapacity;
  front = rear = 0;  // indicating an empty queue
}
```

Sequential queue: `front = rear = -1;`

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

Queue Implementation (Circular)

```
template <class T>
T& Queue<T>::Front() const
{
  if (IsEmpty()) { ... }; // exception handling
  return queue[(front+1) % capacity];
}
```

Sequential queue: `front+1`

```
template <class T>
T& Queue<T>::Rear() const
{
  if (IsEmpty()) { ... }; // exception handling
  return queue[rear];
}
```

Queue Implementation (Circular)

```
template <class T>
void Queue<T>::Push(const T& item)
{
  if ((rear+1) % capacity == front) {
    // queue is full
    // code to double the capacity here
    // update front and rear
  }
  rear = (rear+1) % capacity;
  queue[rear] = item;
}

template <class T>
void Queue<T>::Pop()
{
  if (IsEmpty()) { ... }; // exception handling
  front = (front+1) % capacity;
}
```

Sequential queue: `front == rear`

Sequential queue: `queue[++rear]=item;`

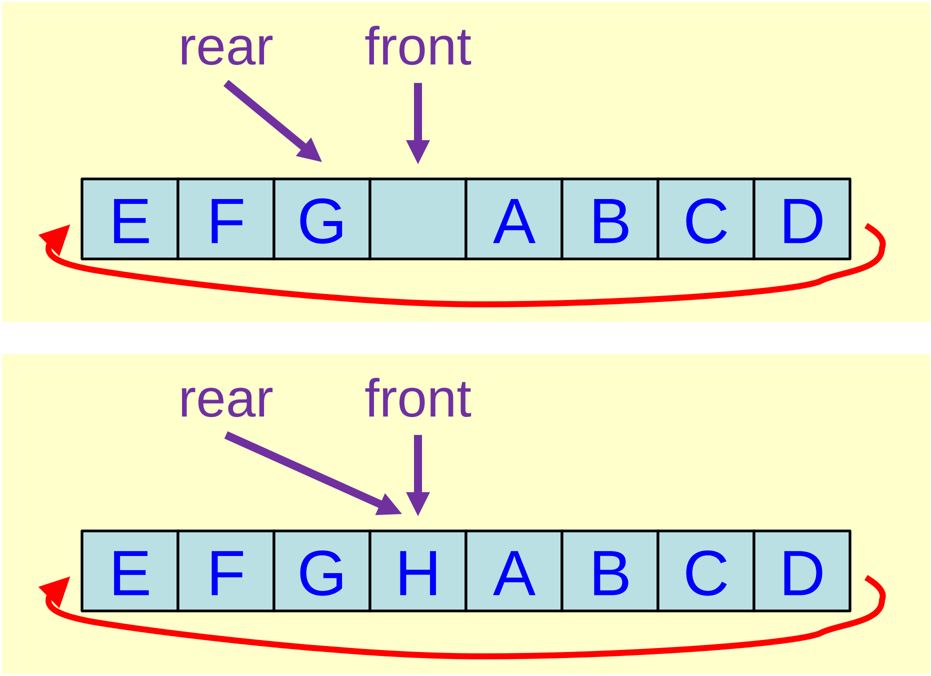
Sequential queue: `front++;`

Recognizing a Full Circular Queue

In circular queue, if all the spaces are occupied, we have `front==rear`.

Example:

One more element:

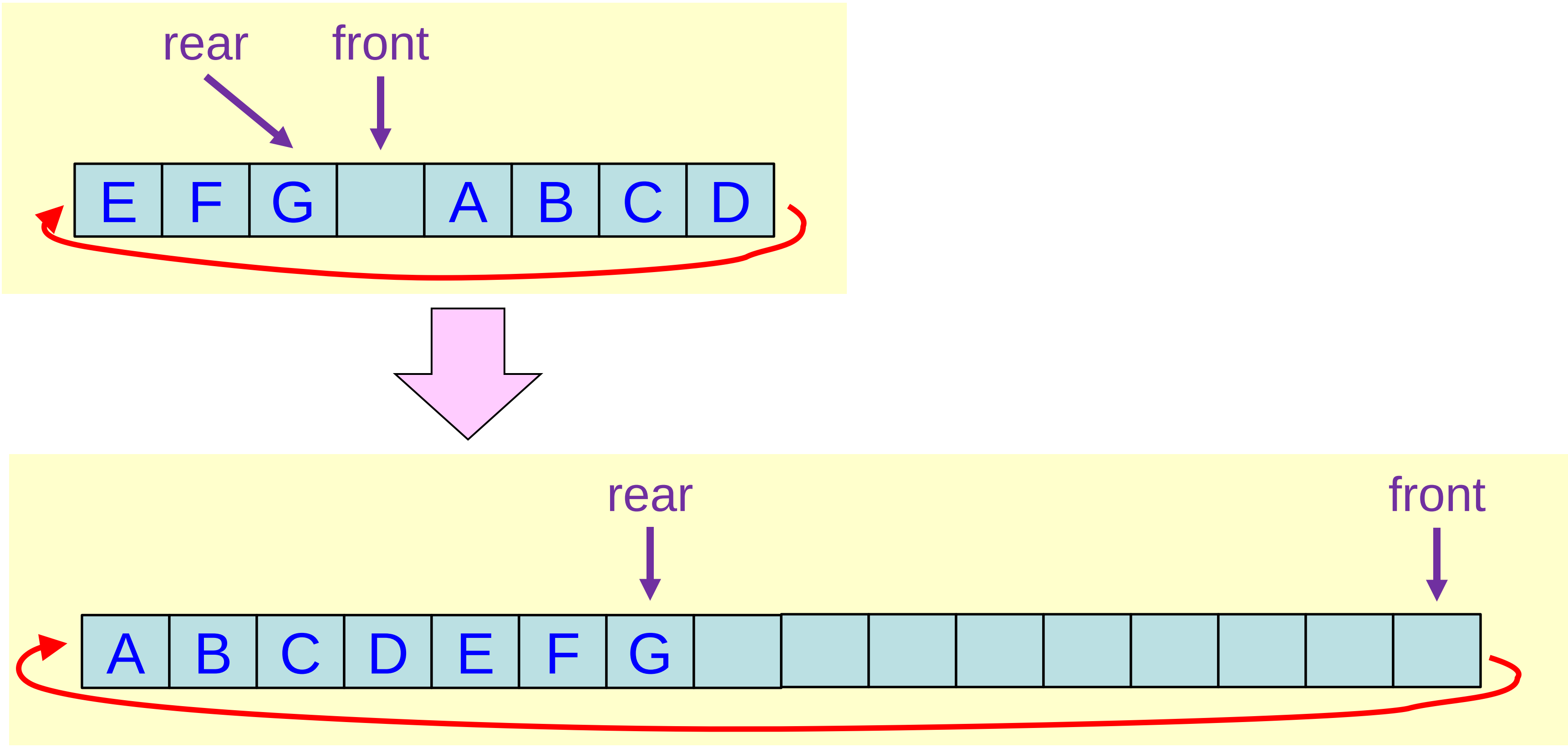


Now we can not distinguish this from an empty queue.

Solution: consider the queue full when `rear` is one position before `front`, i.e., we just waste one space.

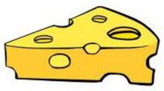
Resizing a Circular Queue

When doubling the capacity of a full circular queue:



A Maze Problem

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	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	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



1: blocked path; 0: through path

Valid movements: Any of the 8 neighboring spaces that are not blocked.

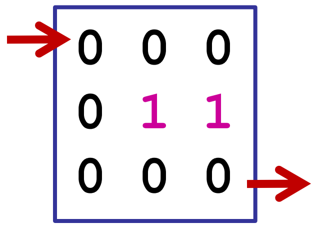
Finding a Path

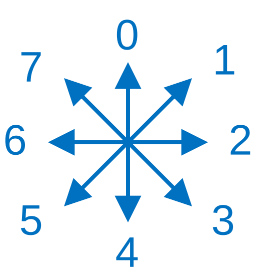
- How do we remember the current path (from the entrance to the current position)?
  - Need to back-trace ➔ Use a stack.
- Idea:
  - When moving to a new position, remember the **<u>old position</u>** and **<u>its next direction to try</u>** in the stack.
  - Whenever we get into a dead end, back-trace until we are at a position with a valid movement. Then continue by trying that movement.
  - Mark visited spaces. (Visited spaces are treated as blocked spaces.)
- Assume that the maze has a size of $m*p$. Use arrays of size $(m+2)*(p+2)$ to simplify the processing at borders.

Pseudo-Code for Finding a Path

```
First item in stack: position (1,1) and direction East
while (stack is not empty)
{
   (i, j, dir) = position and direction at Top of stack
  pop the stack
  while (dir is a valid direction)
  {
     (g, h) = position after movement
     if (g, h) is the goal position // success
       output the path and return
     else if ((!maze[g][h]) && (!mark[g][h])) // legal move
     {
       push (i, j, dir+1) to the stack
        (i, j, dir) = (g, h, 0); // move to (g, h)
       mark[g][h] = 1; // mark (g, h) as visited
     } else {
       ++dir; // try next direction from (i, j)
     }
  }
}
indicate failure and return
```

Finding a Path: Example





(i, j, dir)	(g,h)	ok	1,1,2
(1,1,2)	(1,2)	Y	1,1,3
(1,2,0)	(0,2)	N	
(1,2,1)	(0,3)	N	
(1,2,2)	(1,3)	Y	1,1,3 1,2,3
(1,3,0)	(0,3)	N	
…	…		
(1,3,7)	(0,2)	N	⟸ need to back-trace
(1,2,3)	(2,3)	N	1,1,3
(1,2,4)	(2,2)	N	
(1,2,5)	(2,1)	Y	1,1,3 1,2,6
(2,1,0)	(1,1)	N	
…	…		
(2,1,2)	(2,2)	N	
(2,1,3)	(3,2)	Y	1,1,3 1,2,6 2,1,4
(3,2,0)	(2,2)	N	
(3,2,1)	(2,3)	N	
(3,2,2)	(3,3)	Y	1,1,3 1,2,6 2,1,4 3,2,3
(3,3,0)			

More Thoughts on Maze Problems

- Under what conditions is the array **`mark`** necessary?
- Does the program works if there are multiple paths or loops?
- What does the program need to do to find a shortest path?
- If we don't need to back-trace (e.g., when finding a route on a map), we don't need to use a stack. Can we use a queue instead?
- Is it practical to implement the algorithm recursively?

Evaluation of Expressions

- Examples:
  - 3 + 2 * (3 – 1) ➔ 7
  - (a + b – c) * 2 with a=2, b=3, c=1 ➔ 8
- An expression consists of **operands** (variables, constants) and **operators**, both considered **tokens** of the expression.
- It is necessary to define the order (precedence) of the operators.

A subset of operator precedence in C:

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

Infix and Postfix Expressions

- Infix expressions:
  - (Binary) operators appear <u>between</u> their operands.
  - Standard mathematical expressions; also used in high-level programming languages.
  - Example: **3 + 2**
  - To evaluate, we need to process the operators according to their priorities. ➔ multiple passes
- Postfix expressions:
  - (Binary) operators appear <u>after</u> their operands.
  - The same arrangement of tokens as in machine languages.
  - Example: **3 2 +**
  - Single pass through the expression for evaluation.

Infix and Postfix Expressions

Infix	Postfix
2 + 3 * 4	2 3 4 * +
a * b + 5	a b * 5 +
(1 + 2) * 7	1 2 + 7 *
a * b / c	a b * c /
(a / (b – c + d)) * (e – a) * c	a b c – d + / e a – * c *
a / b – c + d * e – a * c	a b / c – d e * + a c * –

Evaluating Postfix Expressions

This is just pseudo-code

```
void Eval(Expression e)
{ /* A function NextToken(e) returns the next token in
     expression e. Token '#' is returned to indicate
     "end of expression". */
  Stack<Token> stack;
  for (Token x = NextToken(e); x != '#'; x=NextToken(e))
    if (x is an operand)
      stack.Push(x)
    else { // x is an operator
      remove the correct number of operands for x
               from the stack
      evaluate operator x
      push the result back to the stack
    }
}
```

Evaluating Postfix Expressions

Now let's try to evaluate 6 / 2 – 3 + 4 * 2.

Postfix expression: 6 2 / 3 – 4 2 * +

Infix to Postfix Conversion

- Orders of operands are the same in infix and postfix forms. Examples:
  - A + B * C ➔ A B C * +
  - A * B + C ➔ A B * C +
  - A * (B + C) ➔ A B C + *
- Operands sent directly to the output.
- Operators are stored in a stack and moved to the output when necessary.
  - The top operator in the stack should be the one to be computed first (since it will be popped first).
  - If an incoming operator has equal or lower priority than the current top operator, pop and move the top operator to the output.

Infix to Postfix Conversion

Now let's try to convert A + B * C – D

Incoming	Stack	Output
	empty	
A	empty	A
+	+	
B	+	B
*	+ *	
C	+ *	C
–	+	*
	empty	+
	–	
D	–	D
#		–

Infix to Postfix Conversion

- Complications caused by parentheses:
  - When the incoming token is '(', it does not cause any operator in the stack to be popped. ➔ '(' has high priority as an incoming token.
  - When the incoming token is ')', all the operators from **Top** to the first '(' are popped.
  - No incoming token other than ')' can cause '(' to be popped. ➔ '(' has low priority when in the stack.
- When treating '(' as an operator, assign it the highest incoming priority (***icp***) and the lowest in-stack priority (***isp***).
  - For a regular operator, these two priorities (***icp*** and ***isp***) are the same.

Infix to Postfix Conversion

Now let's try to convert A * (B + C) / D

Incoming	Stack	Output
	empty	
A	empty	A
*	*	
(	* (	
B	* (	B
+	* (+	
C	* (+	C
)	* (	+
	*	
/	empty	*
	/	
D	/	D
#		/

Infix to Postfix Conversion

```
void Postfix(Expression e)
{
  Stack<Token> stack;
  for (Token x = NextToken(e); x != '#'; x = NextToken(e))
    if (x is an operand) cout << x;
    else if (x == ')')
    { // output items from the stack until '('
      for (;  stack.Top() != '(';  stack.Pop())
        cout << stack.Top();
      stack.Pop(); // pop the '('
    }
    else { // x is an operator
      for (; isp(stack.Top()) <= icp(x); stack.Pop())
        cout << stack.Top();
      stack.Push(x);
    }
  // end of expression; output everything in the stack
  for (; !stack.IsEmpty(); cout<<stack.Top(), stack.Pop());
  cout << '#' << endl;
}
```

Extra Reading Assignments

- From the textbook: Sections 3.1.2. Focus on the concept of "container classes".