M6. Stack and Queue (ADT/DS)

Instructor: Manikandan Narayanan

Weeks 6-7

CS2700 (PDS) Moodle: https://courses.iitm.ac.in/course/view.php?id=4892

Acknowledgment of Sources

- Slides based on content from related
  - Courses:
    - IITM – Profs. **Rupesh**/Krishna(S)/Prashanth/Kartik's PDS (Thy/Lab) offerings (slides, quizzes, notes, lab assignments, etc. for instance from Rupesh's Jul 2019 offering - www.cse.iitm.ac.in/~rupesh/teaching/pds/jul19/)
    - *Most slides are based on Rupesh Nasre's slides – we thank him and acknowledge by marking **[RN]** in the bottom right of these slides.*
    - *Stack ADT vs. DS view from brilliant.org:*

    https://brilliant.org/wiki/stacks/

  - Books:
    - **Main textbook:** *"Data Structures and Algorithm Analysis in C++"* by ***Weiss*** (content, figures, slides, exercises/questions, etc.). – cited as [WeissBook]
    - Additional/optional book: *"Practice of Programming"* by *Kernighan and Pike* (style of programming, programming exercises/questions, etc.) – cited as [KPBook]

Outline for Module M6

- M6 Stacks and Queues
  - **M6.1 Stack ADT and DS (incl. Applications)**
  - M6.2 Queue ADT and DS

Stack ADT

- Special List
- Operations restricted to one end.
- Insert --> Push
- Remove --> Pop
- LIFO (Last In First Out)
- Cannot access arbitrary element.
- **Important**: Since this is ADT, we do not care about the implementation yet.

Image source: Google images

[RN]

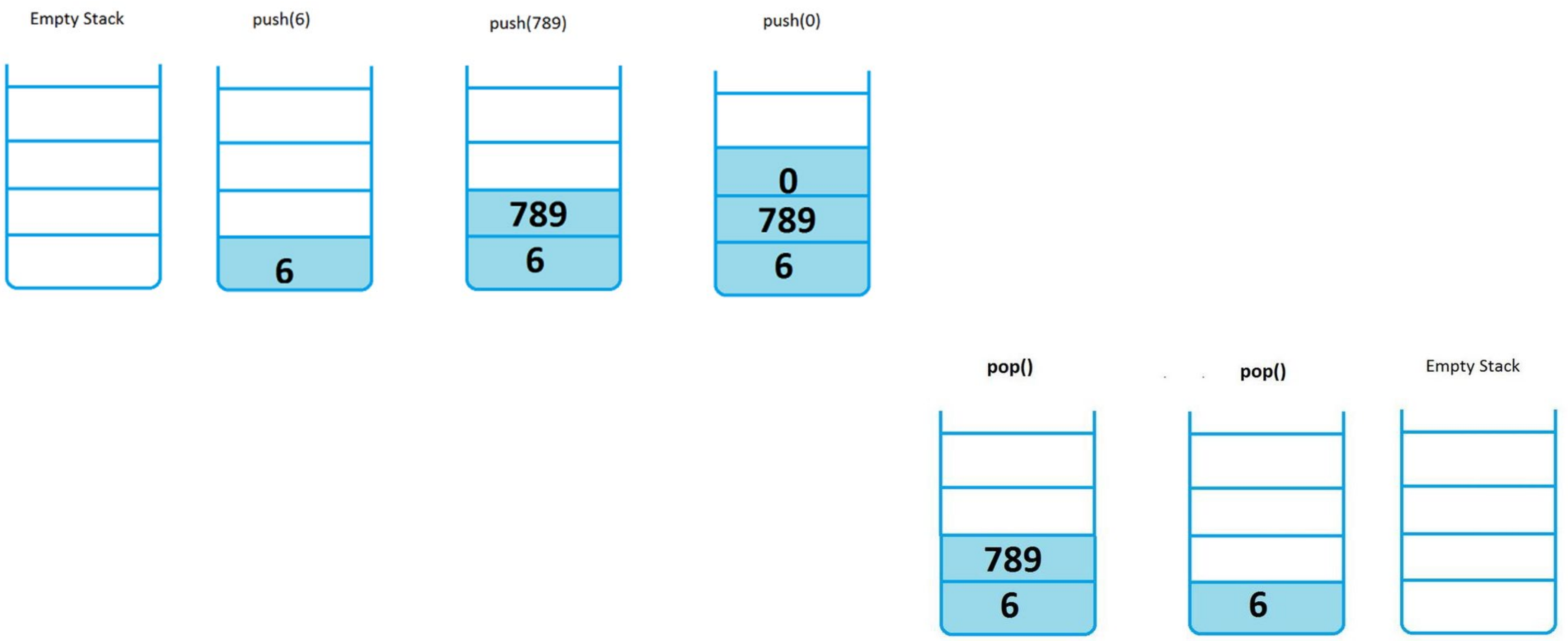
List versus Stack

class **List** {
  void insert(Element);
  void remove(Element);
  bool search(Element);
  int size();
  void print();
  ...
};

class **Stack** {
  void push(Element);
  Element pop(~~Element~~);
  ~~bool search(Element)~~;
  ~~int size()~~; bool isEmpty();
  ~~void print();~~
  ...
};

[RN]

Stack illustration (of push and pop)



[From https://brilliant.org/wiki/stacks/]

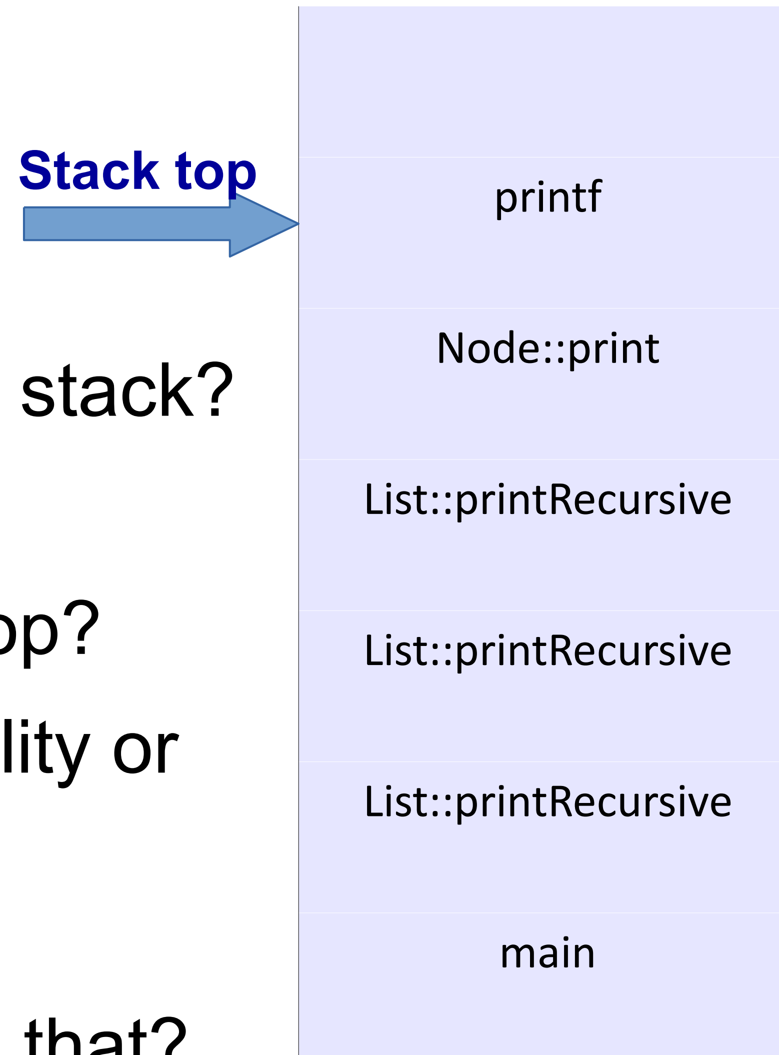
ADT (interface) vs. DS (impl.)

Abstract Data Type (ADT)	**Other Common Names**	**Commonly Implemented with (DS)**
Array	*Vector*	*Array (static/dynamic)*
List	Sequence	Array, Linked List
Queue		Array, Linked List
Double-ended Queue	Dequeue, Deque	Array, Doubly-linked List
Stack		**Array, Linked List**
Associative Array	Dictionary, Hash Map, Hash, Map**	Hash Table
Set		Red-black Tree, Hash Table
Priority Queue	Heap	Heap
Binary Tree…	…	…
Graph…	…	…

[From Abstract Data Types. *Brilliant.org*. Retrieved 06:42, August 13, 2024, from https://brilliant.org/wiki/abstract-data-types/]

Stack Implementation

- **Design decisions**
  - Array versus Linked List
  - Allow traversing through the stack?
  - Allow querying stack size?
  - Allow peeking at the stack top?
  - IsEmpty is user's responsibility or library implementation's?
  - Stack Top points to the last element, or the entry next to that?



Source: stack_impl_typedef.cpp – **Linked List Implementation**

[RN]

Application I: Well-formed (Balanced and Nested) Parentheses

- We want to check if parentheses are well-formed or not.
- Three types of parentheses: ( ), [ ] and { }
- Valid inputs:
  - ([ ] [{ }])
  - [ ] { } [ ] ( ) [[[ ]]]
- Invalid inputs:
  - ((( ))
  - ([ )] { }
  - } }) ({ {

Classwork: Use stack to design an algorithm to check for well-formed parentheses.

Question: Can we design an application of stack from its ADT without knowing its implementation?

Source: stack_usage_std.cpp

[RN]

An aside: Production rules of the grammar generating the language of well-formed parantheses!

S → SS
S → (S)
S → [S]
S → {S}
S → ()
S → []
S → {}

Recursive application of the above rules can generate any string of well-formed parantheses!

Well-formed Parentheses

```
for each input symbol c {
  if (c is an open parenthesis) stack.push(c)
  else if (c is a close parenthesis) {
     if stack.top contains the matching open parenthesis
         pop the element from stack
     else error
  }
}
if (stack is empty)
     // all good.
else error
```

Find a string to match this error (actually two types of errors possible here).

Find a string to match this error.

Homework: Code this up. [RN]

Other non-stack-based solutions to check well-formed parantheses?

Counter approach:

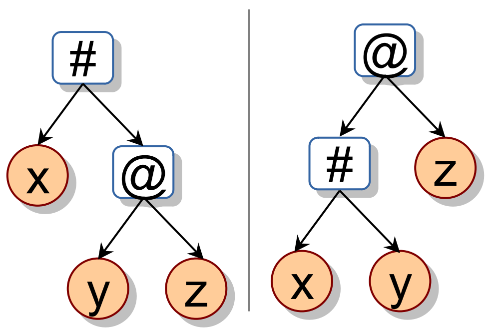
+1 on opening and -1 on closing paranthesis
if counter becomes –ve at any point or doesn’t become 0 in the end, then invalid string.
Will work for single type of parantheses, but not for different types of parantheses.

Repeatedly search for and remove occurrences of [], (), {} in the string.

Will work for different types of parantheses.
What is its running time though?

Application II: Analyzing Expressions

- 1 + 2 * 3 – 4
  - Binary operators appear between the operands
  - Ambiguous without extra knowledge
  - (1 + 2) * (3 – 4) OR
  - 1 + (2 * (3 – 4)) OR
  - (1 + (2 * 3)) – 4 OR
  - ((1 + 2) * 3) – 4 ?
  - Parentheses help disambiguate; domain knowledge helps disambiguate (operator precedence).
  - Won't it be nice if expressions can be written in unambiguous manner?



[RN]

Prefix and Postfix Forms

- 1 + 2 * 3 – 4
  - Binary operators appear between the operands.
  - Called as **infix** form.
- 1 2 3 * + 4 -
  - Binary operators appear after the operands.
  - Called as **postfix** form.
- - + 1 * 2 3 4
  - Binary operators appear before the operands.
  - Called as **prefix** form.

How do these forms help resolve ambiguity?

[RN]

Recursive Formulations (aka production rules of corresp. grammar)

- **Infix**
  - Base: Each operand is an infix expression.
  - Inductive: *infix op infix* is an infix expression.
- **Postfix**
  - Base: Each operand is a postfix expression.
  - Inductive: *postfix postfix op*
- **Prefix**
  - Base: Each operand is a prefix expression.
  - Inductive: *op prefix prefix*
  - Alternate way to express the above two rules:
    - S → op S S
    - S → operand
    - operand → 0|1|2|…|9
    - op → -|+|*|/

[RN]

Prefix, Postfix and Non-ambiguity

Infix	Prefix	Postfix
(1 + 2) * (3 – 4)		
1 + (2 * (3 – 4))		
(1 + (2 * 3)) – 4		
((1 + 2) * 3) – 4		
1 + ((2 * 3) - 4)		

[RN]

Prefix, Postfix and Non-ambiguity

Infix	Prefix	Postfix
(1 + 2) * (3 – 4)	* + 1 2 – 3 4	1 2 + 3 4 - *
1 + (2 * (3 – 4))	+ 1 * 2 – 3 4	1 2 3 4 - * +
(1 + (2 * 3)) – 4	- + 1 * 2 3 4	1 2 3 * + 4 -
((1 + 2) * 3) – 4	- * + 1 2 3 4	1 2 + 3 * 4 -
1 + ((2 * 3) - 4)	+ 1 - * 2 3 4	1 2 3 * 4 - +

- No parentheses in prefix and postfix forms.
- Infix is ambiguous; prefix and postfix are not.
- Unique prefix and postfix forms for different orders of operator evaluation.

[RN]

Postfix Evaluation

- Find the value of `5 1 2 3 * –  4 + 6 * –`.
- Write a program to evaluate a postfix expression.
  - Assume digits, +, –, *, /.

```
For each symbol in the expression
        If the symbol is an operand
                Push its value to a stack
        Else if the symbol is an operator
                Pop two nodes from the stack
                Apply the operator on them
                Push result to the stack
```

Source: infixfp2postfix_postfixeval_etc.cpp

[RN]

Prefix Evaluation

For each symbol in the expression right-to-left
  If the symbol is an **operand**
    Push its value to the stack
  Else if the symbol is an **operator**
    Pop two symbols from the stack
    Apply the operator on them
    Push result to the stack

Prefix
* + 1 2 – 3 4
+ 1 * 2 – 3 4
- + 1 * 2 3 4
- * + 1 2 3 4
+ 1 - * 2 3 4

Homework: Code this up.

[RN]

Infix to Posfix

- Given an infix expression (that is fully paranthesized), convert it to a postfix form (without parentheses).

Infix	Prefix	Postfix
((1 + 2) * (3 – 4))	* + 1 2 – 3 4	1 2 + 3 4 - *
(1 + (2 * (3 – 4)))	+ 1 * 2 – 3 4	1 2 3 4 - * +
((1 + (2 * 3)) – 4)	- + 1 * 2 3 4	1 2 3 * + 4 -
(((1 + 2) * 3) – 4)	- * + 1 2 3 4	1 2 + 3 * 4 -
(1 + ((2 * 3) - 4))	+ 1 - * 2 3 4	1 2 3 * 4 - +

Source: infixfp2postfix_postfixeval_etc.cpp

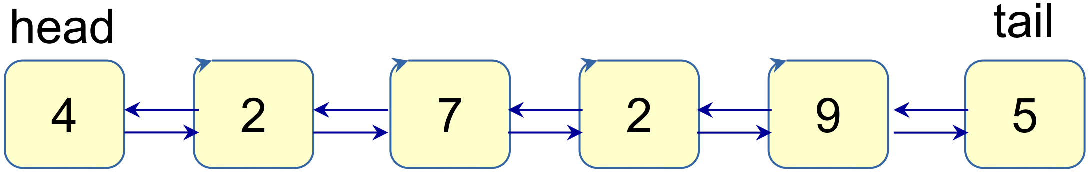
[RN]

Outline for Module M6

- M6 Stacks and Queues
  - M6.1 Stack ADT and DS (incl. Applications)
  - **M6.2 Queue ADT and DS**

Queue

- Special list
- Insertions at one end, deletions at the other
- Tracked using two pointers: head and tail
- FIFO (what is FCFS?)
- Cannot access arbitrary element
- Insert → push / enqueue
- remove → pop / dequeue



[RN]

Queue ADT

- **Classwork**: Write down the Queue ADT.

```
struct Queue {
        void push(Element); // enqueue
        Element pop();
        // dequeue
        bool isEmpty();
        ...
};

class Queue {
        void push(Element);
        void pop();
        Element front();

        bool isEmpty();
```

Source:
queue_usage_std.cpp

[RN]

Applications of Queues

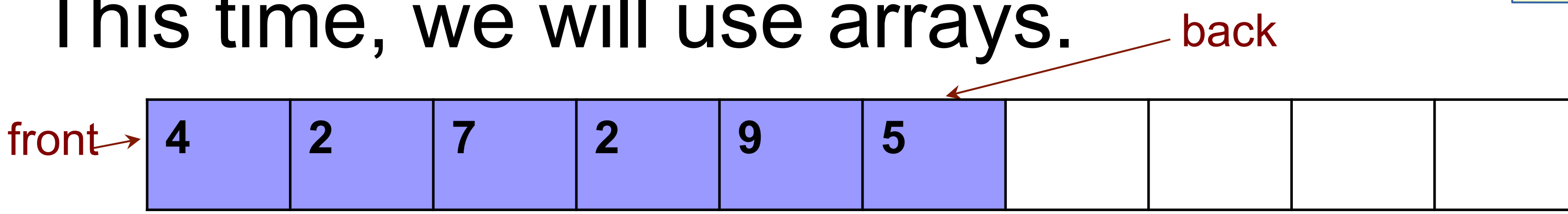
- Ticket counters, shops, canteens, ...
- Resource sharing in hardware
  - Printing
  - Writing to the same memory
- Process scheduling in OS (FCFS)
- ...

[RN]

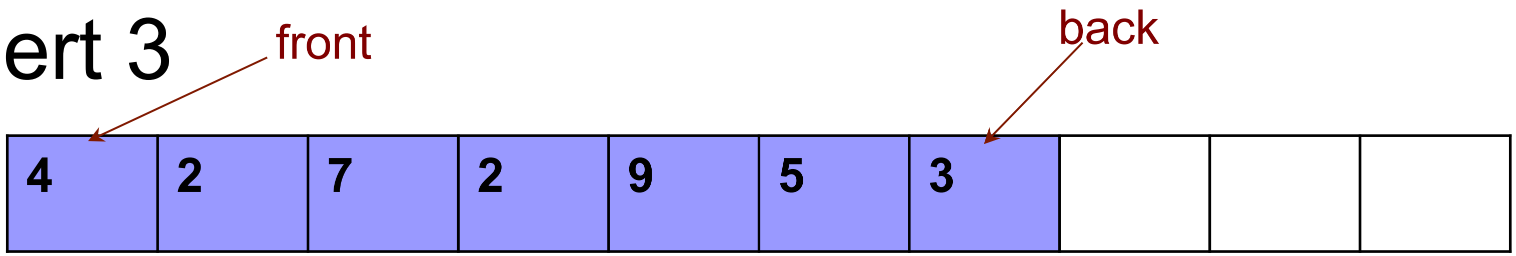
Queue DS (Implementation)

Recall circular list

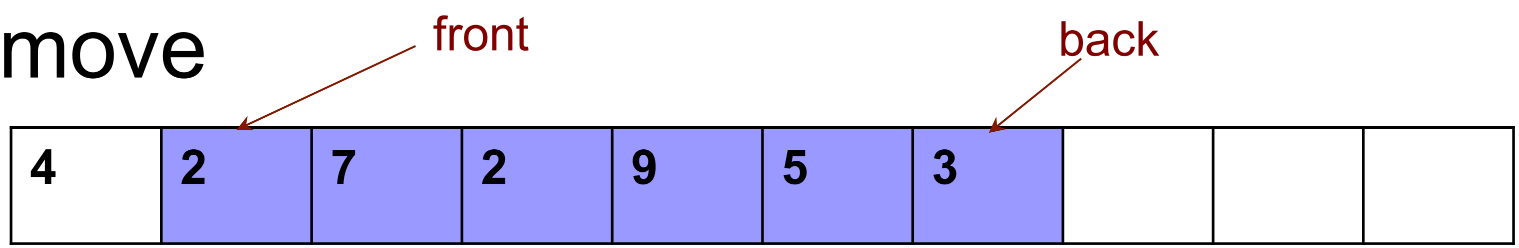
- This time, we will use arrays.



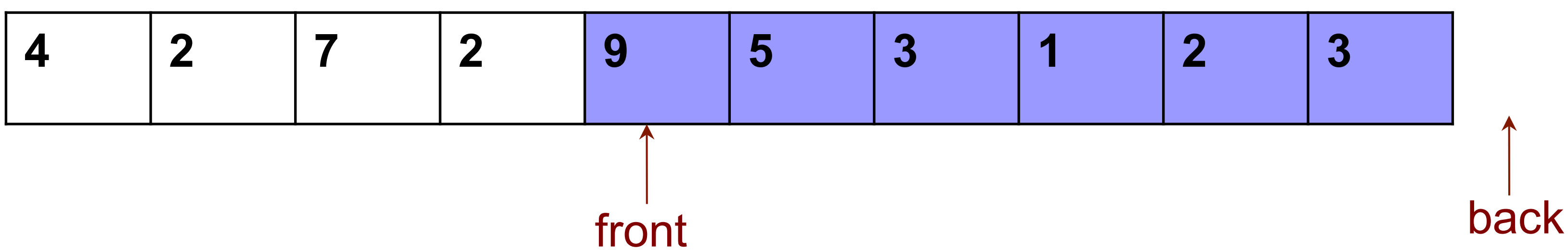
- Insert 3



- Remove



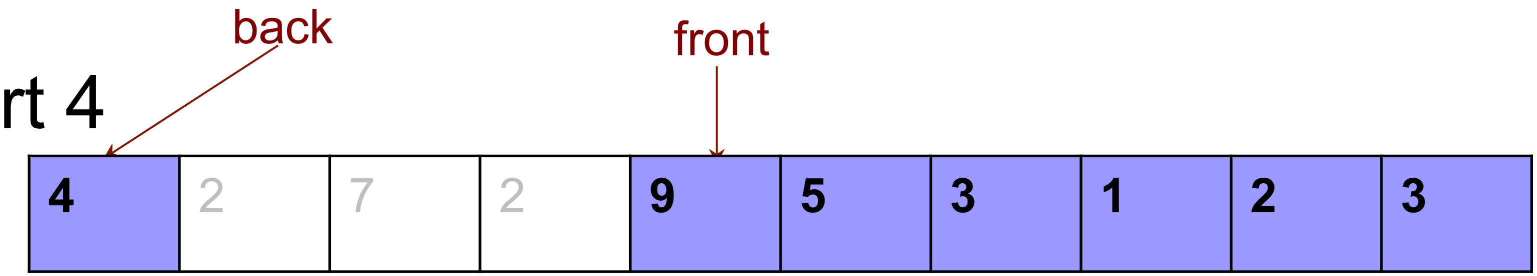
- Remove, Remove, Remove, Insert 1, 2, 3, 4



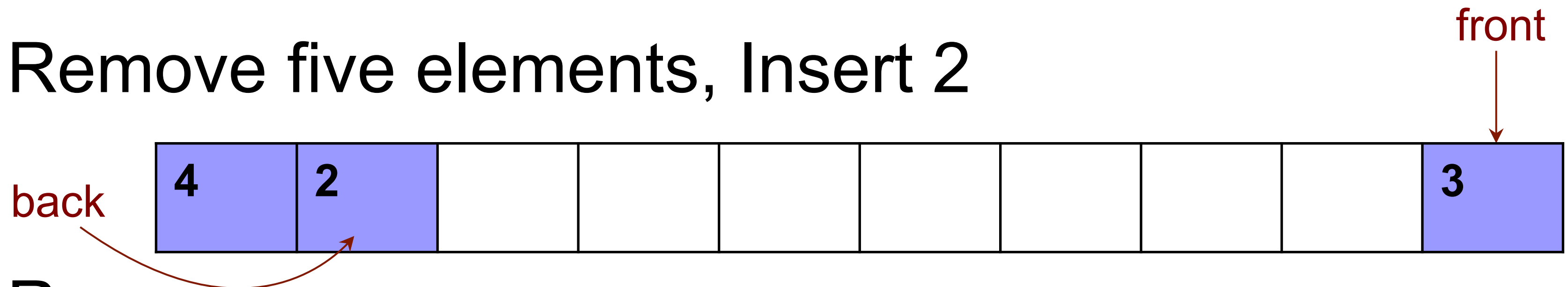
[RN]

Wrap-around

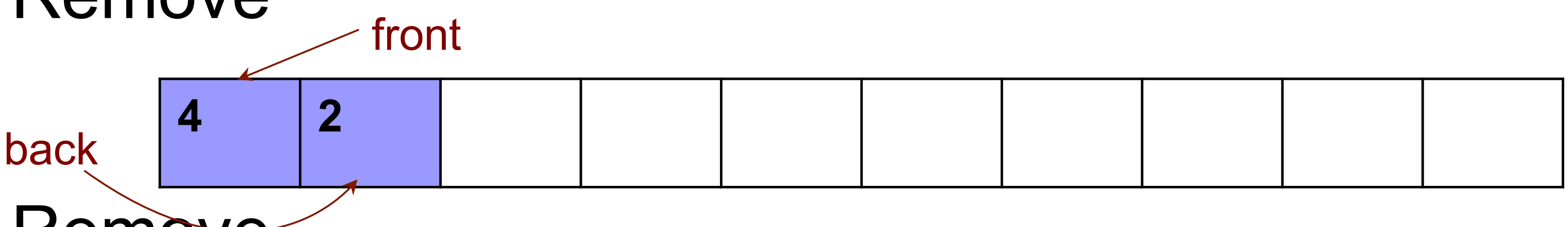
- Insert 4



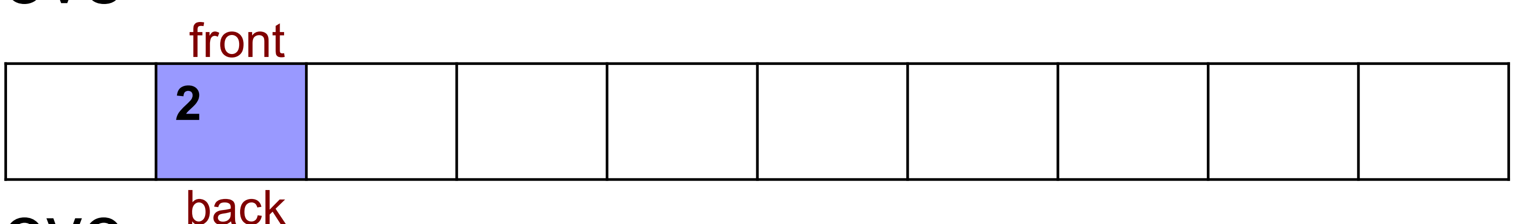
- Remove five elements, Insert 2



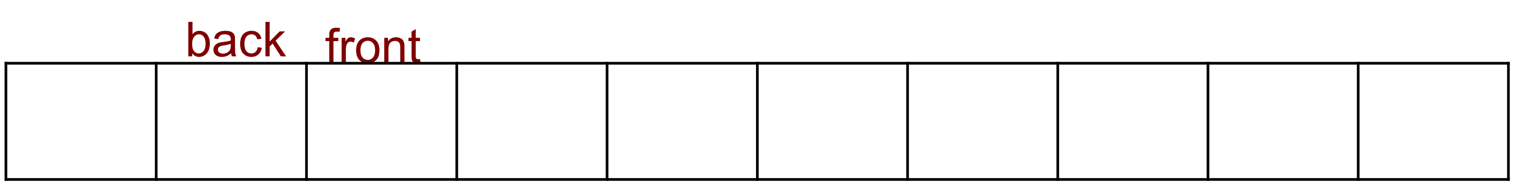
- Remove



- Remove



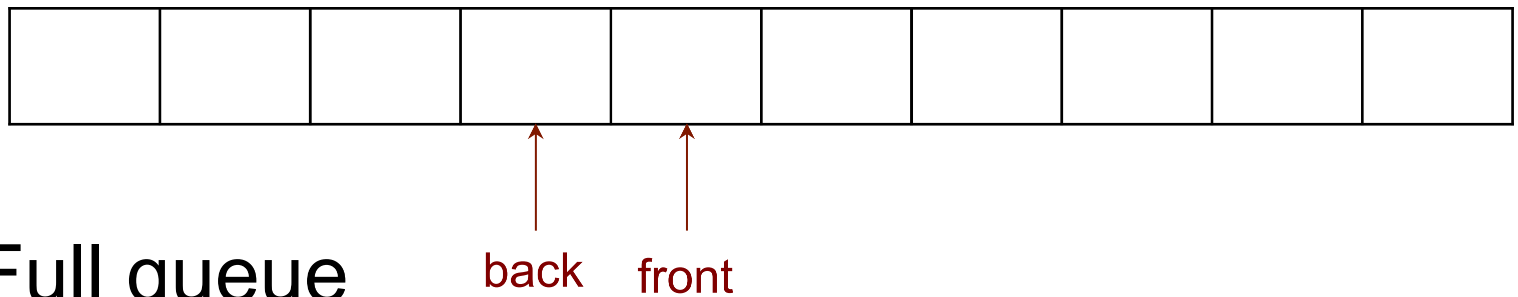
- Remove



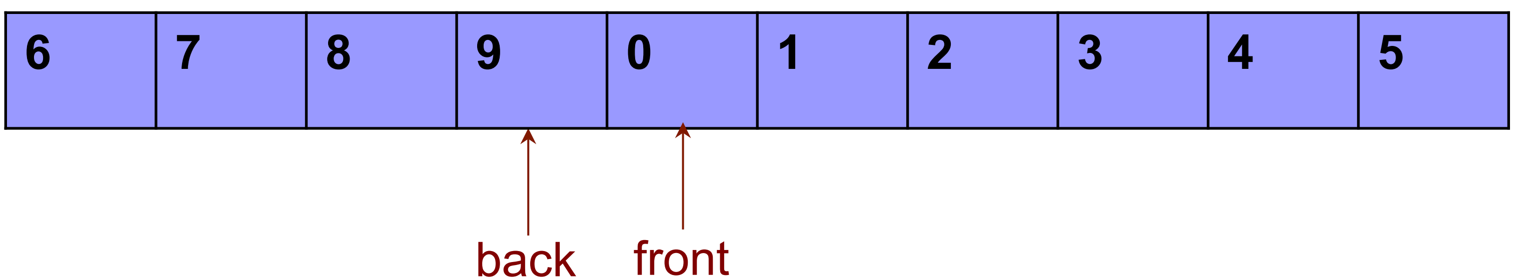
[RN]

Empty versus Full

- Empty queue



- Full queue



- Possible solutions
  - Leave one space unused (N-1 elements).
  - Track size separately (via **curSize** in **Source:** queue_impl_template.cpp).

[RN]

Practice problems

- Implement a stack using two queues.
  - push/pop should be implemented using enqueue / dequeue.
  - What is the complexity (O(.)) for these operations?

- Implement a queue using two stacks.

- Solve problems at the end of Chapter 3.

[RN]

Learning Outcomes

- Understand List, Stack, Queue ADTs.
- Implement these ADTs using C/C++ with pointers (linked list) or arrays.
- Study various applications using these data structures.

[RN]

Backup Slides

Task: Convert Infix (not necessarily fully-paranthesized) to Postfix Expn.
Solution: One solution approach below. Another solution approach using two stacks is in [WeissBook]. Read that up also.
Homework: Code this or [WeissBook] soln. up.

```
For each symbol in the expression
        If the symbol is an operand
                Print the symbol
        Else if the symbol is an opening parenthesis
                Push the symbol on stack
        Else if the symbol is a closing parenthesis
                Do {
                        Pop symbol from the stack
                        If symbol is not opening parenthesis
                                Print the symbol
                } while symbol is not opening parenthesis
        Else {
                        // symbol c is an operator
                        Peek symbol d from the stack
                        While symbol d has higher or equal priority than c
                                Print the symbol d
                                Pop symbol d from the stack
                        Push the symbol c on stack
                }
        }
        While stack is not empty {
                Pop symbol from the stack
                Print the symbol
        }
        Return postfix
```

[RN]

Queue Application: Call Center

- Multiple users call a call-center.
- Multiple operators answer the call.
- Each call takes an unknown amount of time.
- When all the operators are busy
  - Calling users need to wait.
- When an operator becomes available
  - Which waiting user is answered?
- Can we use Queue ADT to implement this?

[RN]

Call Center: Data Structures

- User (id, call time)
- Operator (id)
- Queue of waiting users
- List of busy operators
- Queue of free operators

[RN]

Call Center: Simulation

- Simulation is often based on time.
- At each time unit, various actions occur.
  - A new user arrives.
  - A free operator needs to be assigned to a user.
  - No operator is free, so the user needs to wait.
  - A busy operator becomes free.
  - Nothing happens, call time of engaged users reduces.
- Simulation ties these actions together logically.

[RN]