

# Lists

Rupesh Nasre.  
*rupesh@cse.iitm.ac.in*

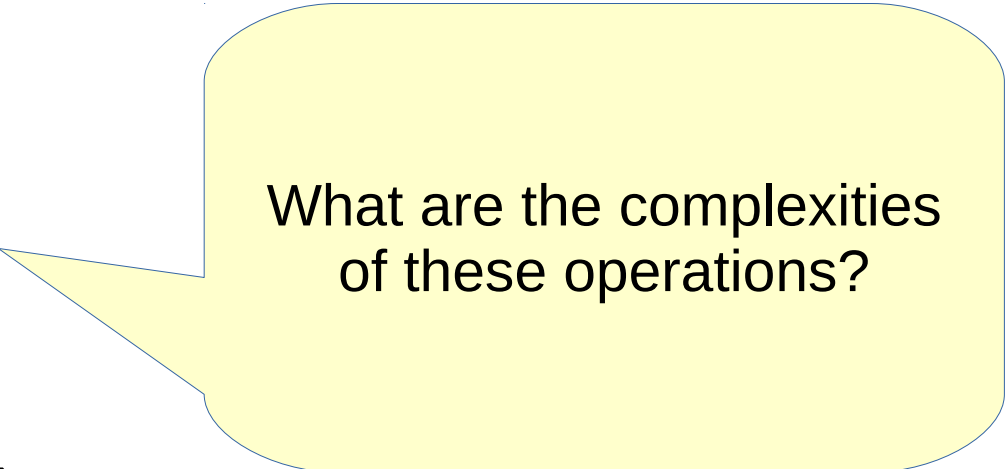
August 2021

# ADT

- Abstract Data Type
- Defines the **interface** of the functionality provided by the data structure.
- Hides implementation details.
  - Defines *what* and hides *how*.
- Makes software modular.
- Allows easy change of implementation.

# List as an ADT

```
class List {  
public:  
    List();  
    void insert(Element e);  
    bool find(Element e);  
    void remove(Element e);  
    void print();  
    int size();  
};
```



What are the complexities of these operations?

# Other ADTs

- Fan regulator
  - IncSpeed, decSpeed, getSpeed, getCompanyName
- Integer
  - size, isSigned, getValue, setValue, add, sub
- Student
  - getRollNo, getHostel, getFavGame, setHostel, getSlots, setCGPA

# List using Array

```
class List {
```

```
public:
```

```
    List();
```

```
    void insert(Element e);
```

```
    bool find(Element e);
```

```
    void remove(Element e);
```

```
    void print();
```

```
    int size();
```

```
};
```

4	2	7	2	9			
---	---	---	---	---	--	--	--

## Design decisions

- Size of the array?
- Maintain size separately or use a sentinel?
- On overflow: error or realloc?
- On underflow: error message or exit or silent?
- Printing order?
- Duplicates allowed?
- For duplicates, what does remove do?
- ...

# List using Array

```
class List {
```

```
public:
```

```
    List();
```

```
    void insert(Element e);
```

```
    bool find(Element e);
```

```
    void remove(Element e);
```

```
    void print();
```

```
    int size();
```

```
};
```

4	2	7	2	9			
---	---	---	---	---	--	--	--

**With certain design decisions:**

**$O(1)$**

**$O(N)$**

**$O(N)$**

**$O(N)$**

**$O(1)$**

# List using Linked List

```
class List {
```

```
public:
```

```
    List();
```

```
    void insert(Element e);     $O(N)$  without tail pointer, else  $O(1)$ 
```

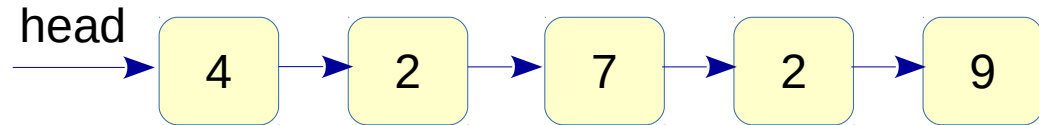
```
    bool find(Element e);     $O(N)$ 
```

```
    void remove(Element e);  $O(N)$ 
```

```
    void print();     $O(N)$ 
```

```
    int size();     $O(1)$ 
```

```
};
```



If the complexities of array-based versus linked-list-based implementations are the same, **why use linked lists?**

# Arrays versus Linked Lists

- Need to copy the existing array on reallocation.
- Removal of  $i^{\text{th}}$  element needs element-shifting from  $i+1$  to end.
- Same with insertion.
- Array concatenation is linear time.
- Only a link needs to be established ( $O(1)$ ).
- Removal of an element using pointers can be done in  $O(1)$ .
- Same with insertion.
- List concatenation is  $O(1)$ .

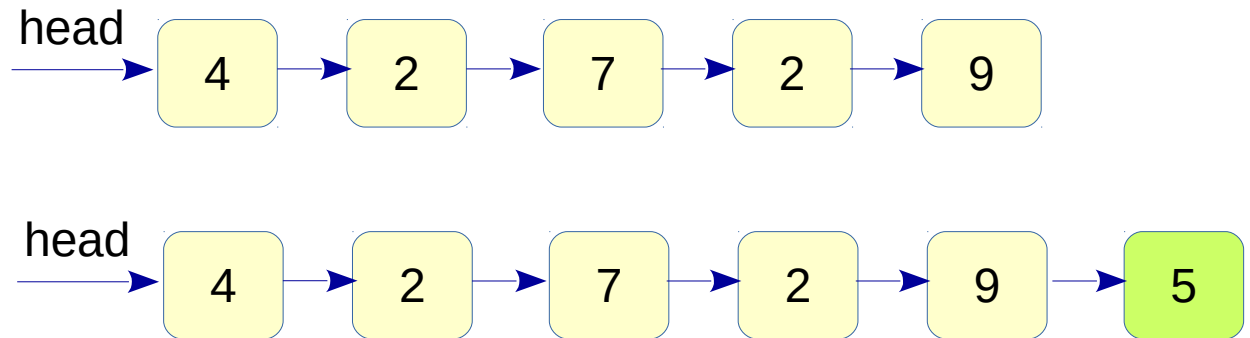


# Linked List Implementation

- Source: **sll.cpp**

# List insert

- insert(5)



## Setup node:

```
Node *newptr = new Node();  
newptr->val = 5;  
newptr->next = NULL;
```

## End case:

```
if (head == NULL) head = newptr;
```

## Regular case:

```
for (Node *ptr = head; ptr->next; ptr = ptr->next)  
    ;  
ptr->next = newptr;
```

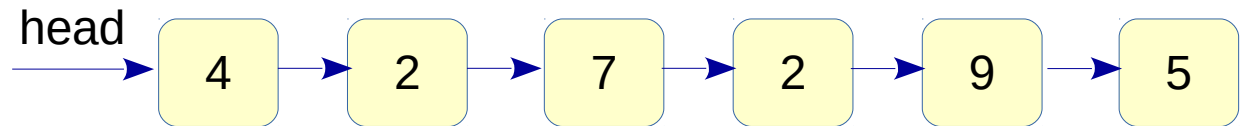
# Surprise Quiz

Login to moodle.

Solve **quiz** (10 minutes).

# List print

- print()



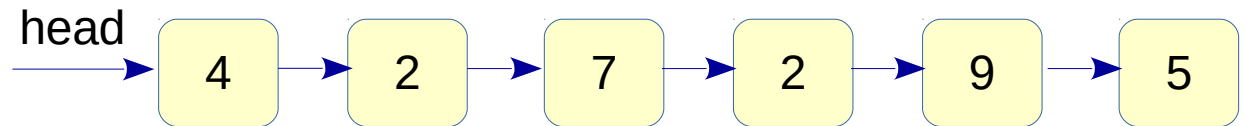
**Output:** 4 2 7 2 9 5

For each element in the list  
Print the element

```
for (Node *ptr = head; ptr; ptr = ptr->next)
    printf("%c ", ptr->val);
printf("\n");
```

# List find

- find(9)



For each element in the list

    If the element is same as that to be searched

        Found the element

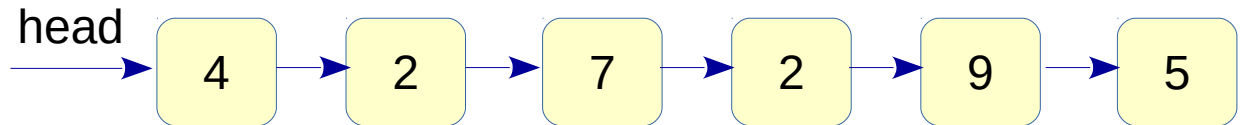
Element not present

```
for (Node *ptr = head; ptr; ptr = ptr->next)
    if (ptr->val == val) return true;
return false;
```

# List remove

- remove(2)
- remove(5)
- remove(4)

We want to remove all occurrences of the value.



## Special case:

```
if (head == NULL) return false;
```

## General case:

```
Node *previous = NULL;
for (Node *ptr = head; ptr;) {
    if (ptr->val == val) {
        Node *toberemoved = ptr;
        if (previous) {
            previous->next = ptr->next;
        } else head = ptr->next;
        ptr = ptr->next;
        delete toberemoved;
        removed = true;
    } else {
        previous = ptr;
        ptr = ptr->next;
    }
}
```

# Pitfalls

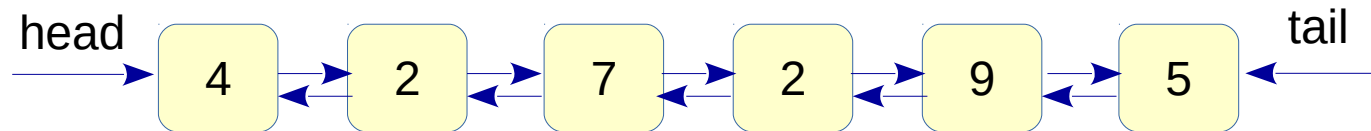
- `ptr = head->next;` // segfault. Check if head is NULL.
- `Node *ptr = &node1; return;` // local variable node1.
- `ptr = malloc(sizeof(Node*));` // insufficient memory.

Wrong **deleteList** program

```
for (ptr = head; ptr; ptr = ptr->next)
    free(ptr);
```

// invalid memory on free.  
// may work but wrong.

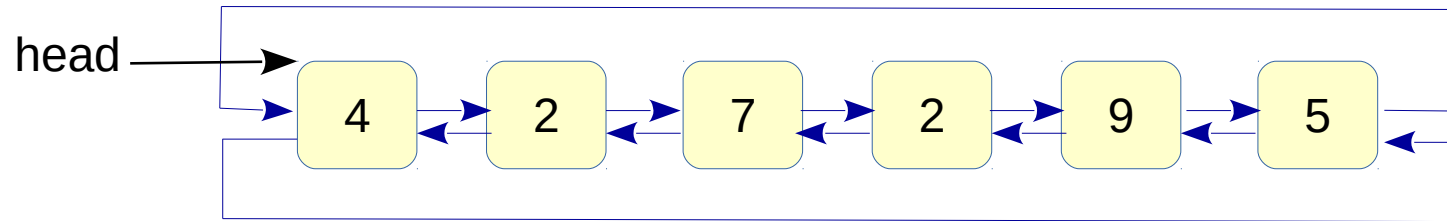
# Doubly Linked List



- Links in both the directions.
- Node structure contains two pointers: next and previous.
- Deletion now becomes simpler.
- Two pointers: head and tail maintain list ends.
- **Classwork:** Write a function to remove a node.



# Circular Doubly Linked List



- Last element points to the first, and first element's previous is the last node.
- Node structure continues to contain two pointers: next and previous.
- Tail pointer is **not** required.
- A singly linked list can also be circular.
- **Classwork:** Write a function to print all the node values in a CDLL.

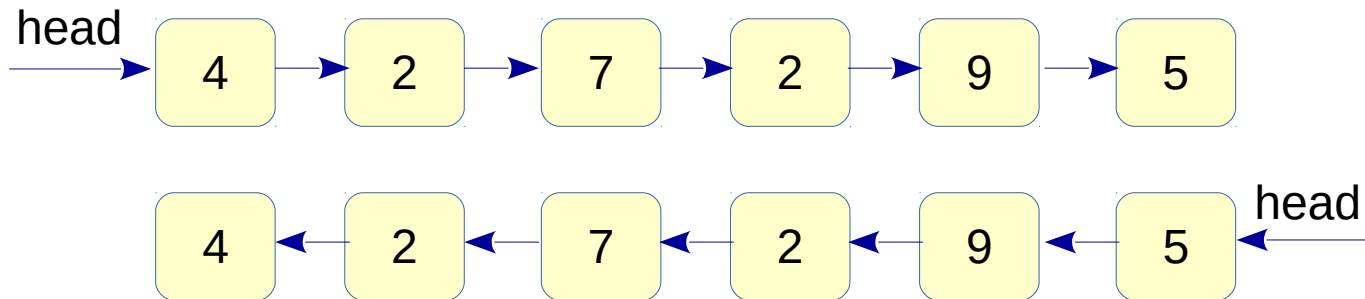
# Polynomial ADT

- $F(X) = \sum_{i=0}^N A_i X^i$
- Example:  $x^4 - 4x^3 + 7x - 6$
- **Member functions**
  - Initialize
  - Set a coefficient (for a power)
  - Add polynomials
  - Multiply polynomials
  - ...
- **Implementation**
  - Could be using arrays
  - Could be using linked lists
- **Classwork:** Create a struct / class to implement polynomials.
- Are there disadvantages of using arrays?
  - $2x^{1000} - x$
  - What are the design decisions for using lists?

# Polynomial ADT

```
class Polynomial {  
    int coeff[MaxDegree + 1];  
};  
  
void Polynomial::initialize(int coeff[ ]) {  
    // Classwork: implement this.  
}  
  
void Polynomial::add(Polynomial p2, Polynomial psum) {  
    // Classwork: implement this.  
}
```

# List Reversal



- Given a list (SLL, DLL, CSLL, CDLL), reverse it.
- The traversal from head should result in the opposite order.
- Typically need three pointers: previous, current and next.
- **Classwork:** Write a list reversal for SLL (*sll.cpp*).
- **Classwork:** Write a recursive list reversal.

# Recursive Methods

- Sometimes natural to model.
- Sometimes inefficient to implement.
- **Classwork**: find an element recursively.
- **Classwork**: print a list recursively.
  - How to print in reverse?
  - **sl.cpp**

# Stack ADT

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



# List versus Stack

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

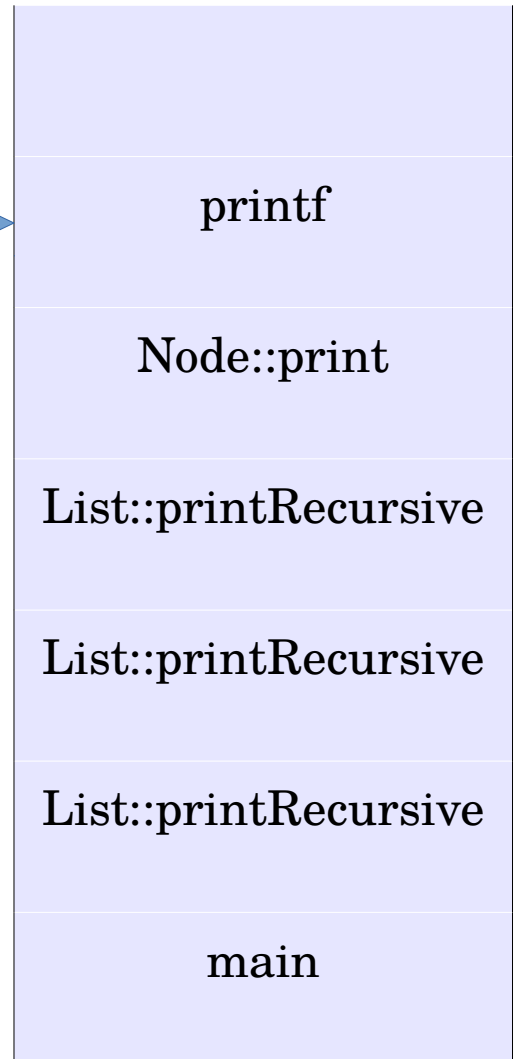
```
class Stack {  
    void push(Element);  
    Element pop(Element);  
bool search(Element);  
int size(); bool isEmpty();  
void print();  
    ...  
};
```

# 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?

Stack top



Source: [stack.cpp](#)



# Balanced Parentheses

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

**Classwork:** Use stack to design an algorithm to check for balanced parentheses.

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

# Balanced 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

Find a string to match  
this error.

}

**Source:** `parentheses.cpp`

if (stack is empty)

// all good.

else error

Find a string to match  
this error.

# Stack Implementation

- `stackimpl.c`

# Expressions

- $1 + 2 * 3 - 4$

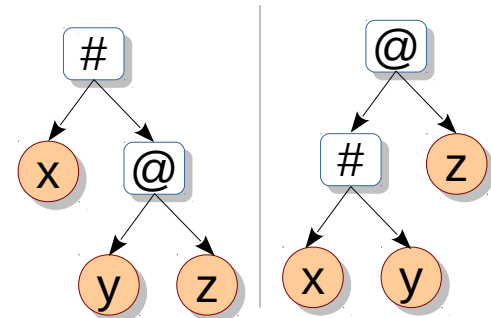
- Binary operators appear **between** the operands
- Ambiguous without extra knowledge

$(1 + 2) * (3 - 4)$  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?

# 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?

# Prefix, Postfix and Non-ambiguity

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

# 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.

# 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



# 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.

# Infix to Postfix

- Given an infix expression (with parentheses), convert it to a postfix form (without parentheses).

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

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

# Recursive Formulations

- **Infix**

- Base: Each operand is an infix expression.
- Inductive: *infix op infix* is an infix expression.

- **Postfix**

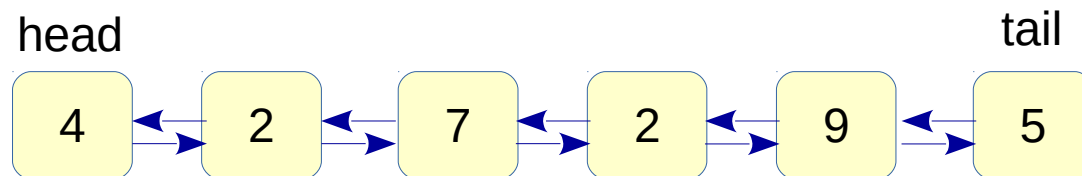
- Base: Each operand is a postfix expression.
- Inductive: *postfix postfix op*

- **Prefix**

- Base: Each operand is a prefix expression.
- Inductive: *op prefix prefix*

# 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



# 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();  
    Element back();  
    bool isEmpty();  
    ...  
};
```

**Source:** q.cpp

# 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?

# Call Center: Data Structures

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



# 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.

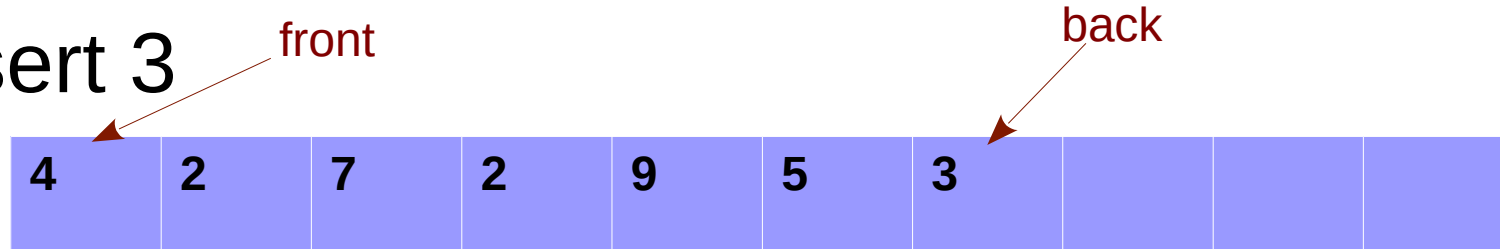
# Queue Implementation

Recall  
circular list

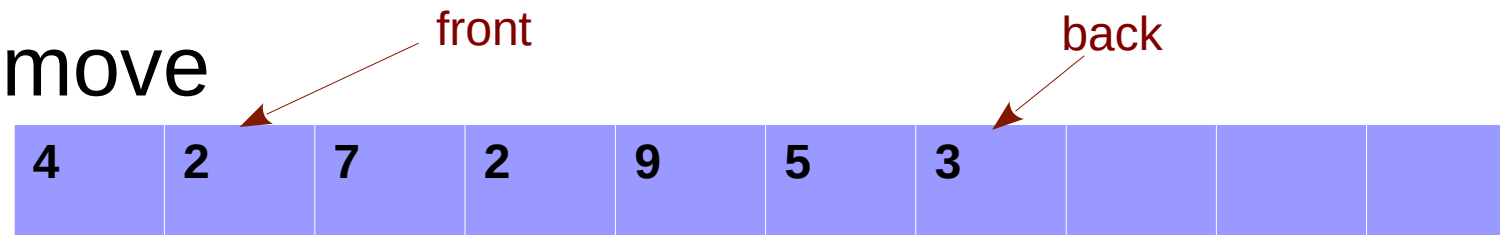
- This time, we will use arrays.



- Insert 3

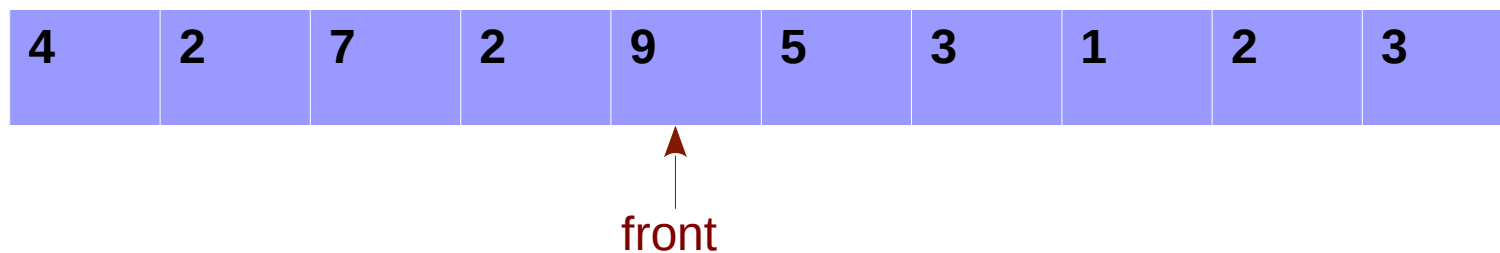


- Remove

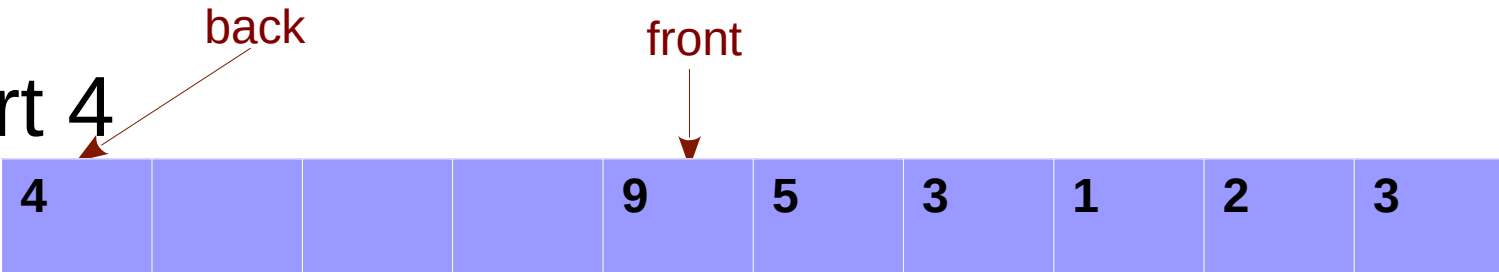


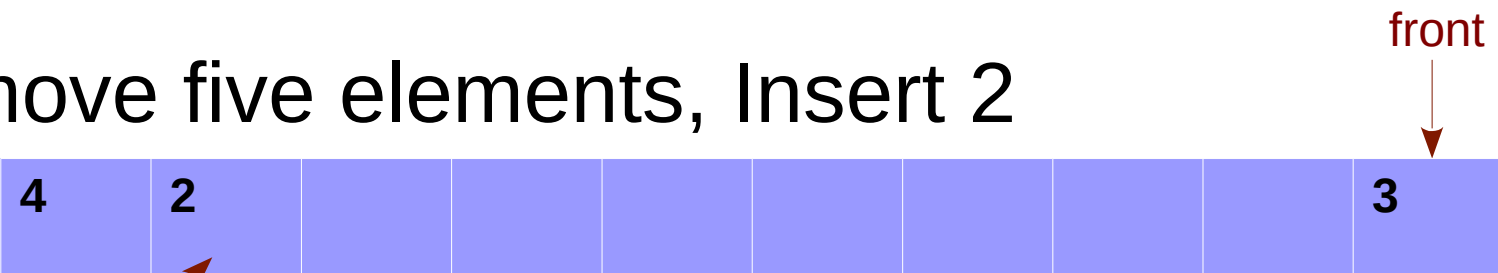
qimpl.c

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

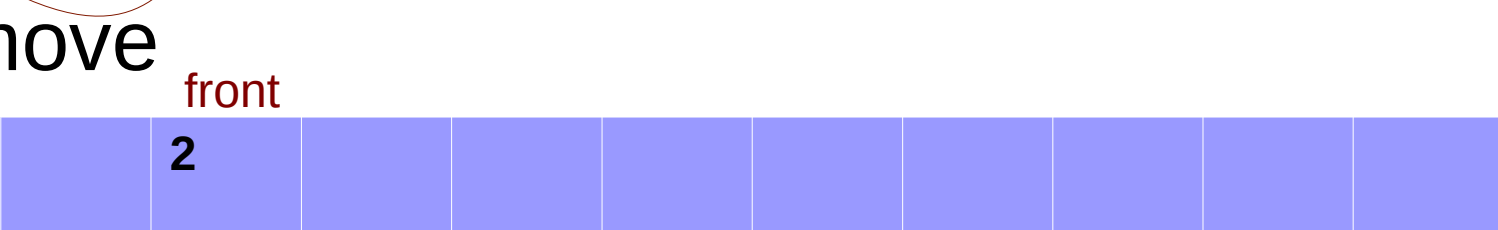


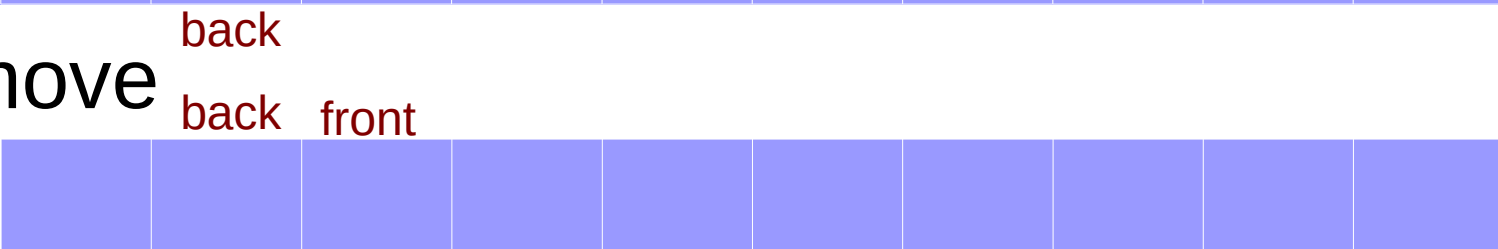
# Wrap-around

- Insert 4  


- Remove five elements, Insert 2  


- Remove  
 qimpl2.c

- Remove  


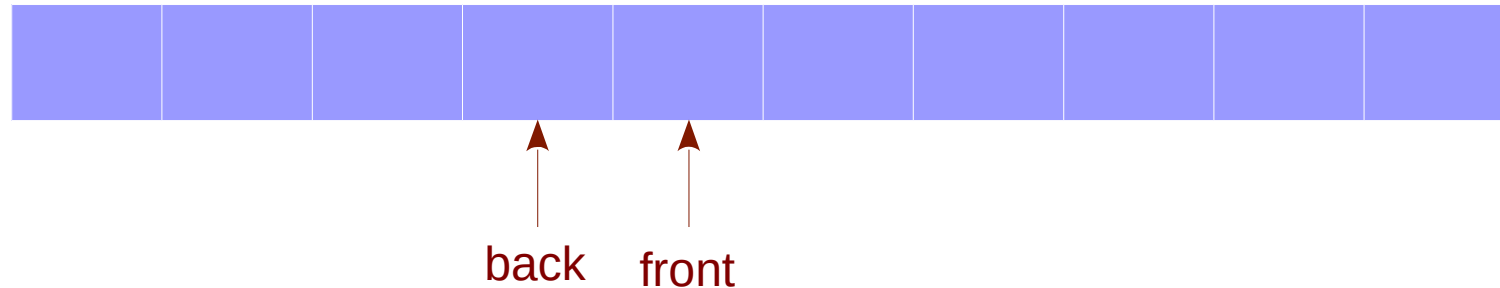
- Remove  


# Queue Conditions

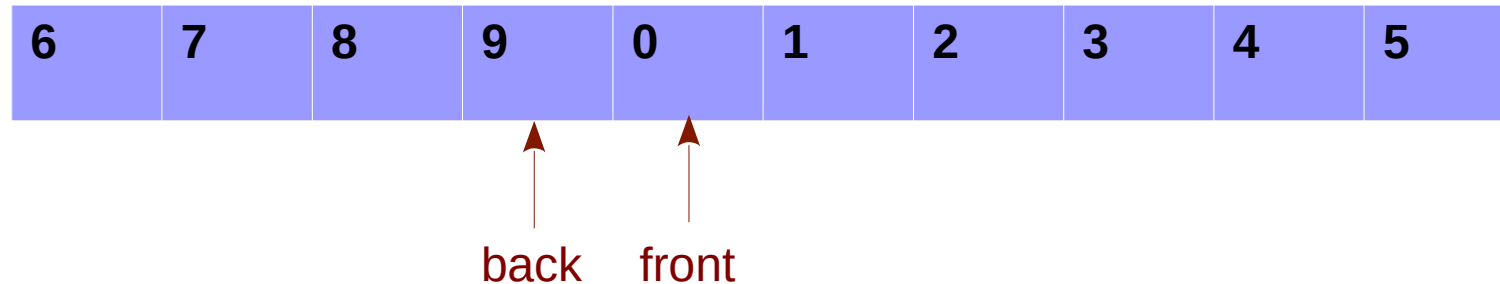
- Queue is empty:
  - when  $\text{front} > \text{back}$  (in previous slide)
  - That is also initialization:  $\text{front} = 0, \text{back} = -1$
  - Our implementation **qimpl.c** uses  $\text{front} = 0, \text{back} = 0$
- Whichever you use, follow invariants:
  - **qimpl.c**:  $\text{front}$  points to the first element in the queue.  
 $\text{back}$  points to the place where next element should be inserted.
  - Previous slide:  $\text{front}$  points to the first element in the queue.  
 $\text{back}$  points to the last element in the queue.
- **Classwork**: Write conditions for when queue is full.

# Empty versus Full

- Empty queue



- Full queue



- Possible solutions

- Leave one space unused ( $N-1$  elements).
- Track size separately (used in `qimpl.c`).

# Practice problems

- Implement a stack using two queues.
  - push/pop should be implemented using enqueue / dequeue.
- Implement a queue using two stacks.
- Implement three stacks using an array (without space wastage).
- Solve problems at the end of Chapter 3.

# Learning Outcomes

- Use List, Stack, Queue ADTs in applications.
- Implement these ADTs using C/C++ with pointers or arrays.
- Study various applications using these data structures.