CSC10004: Data Structure and Algorithms

Lecture 6: Stack and Queue

Lecturer: Bùi Văn Thạch
TA: Ngô Đình Hy/Lê Thị Thu Hiền
{bvthach,ndhy}@fit.hcmus.edu.vn, lthien@hcmus.edu.vn

Course topics

0. Introduction
1. Algorithm complexity analysis
2. Recurrences
3. Search
4. Sorting
5. Linked list
6. Stack, and Queue, Priority queue
7. Tree
    1. Binary search tree (BST)
    2. AVL tree
8. Graph
    1. Graph representation
    2. Graph search
9. Hashing
10. Algorithm designs
    1. Greedy algorithm
    2. Divide-and-Conquer
    3. Dynamic programming

Goals

1. Able to define a Stack, and to implement it with array and linked list.
2. Able to define a (priority) Queue, and to implement it with array and linked list.
3. Able to use stack and queue in applications.

Summary

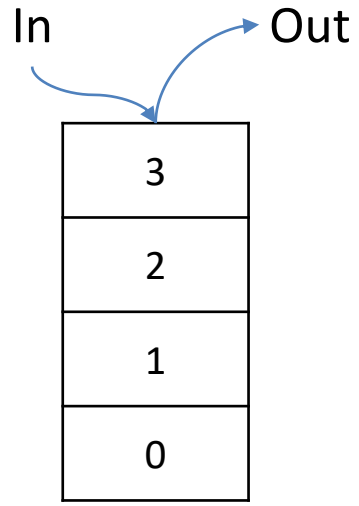
		Design	Run time				Space	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	Det.	$O(n^2)$	$O(\log n)$	$O(1)$ if position known with list $O(n)$ for array		Yes	No
	Insertion sort	Det.	$O(n^2)$				Yes	Yes
	Heap sort	Det.	$O(n \log n)$				Yes	No
	Merge sort	Det.	$\Theta(n \log n)$				No	Yes
	Quick sort	Rnd.	$O(n \log n)$				Yes	No
Non-comparison-based	Radix sort	Det.	$O(d(n+k))$	$O(\log n)$			No	Yes

d is the number of digits and k is the range of digits.

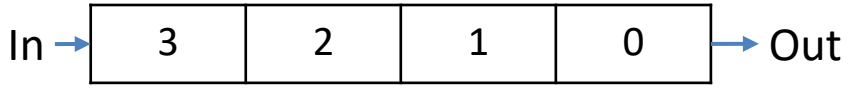
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

Last-In-First-Out (LIFO)



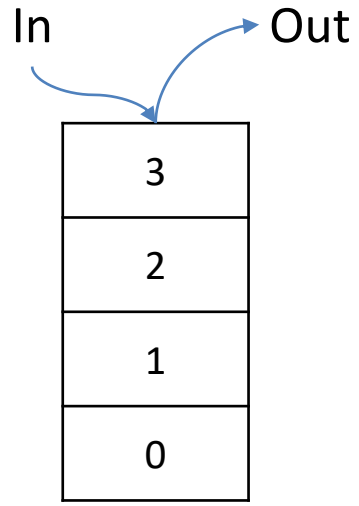
First-In-First-Out (FIFO)



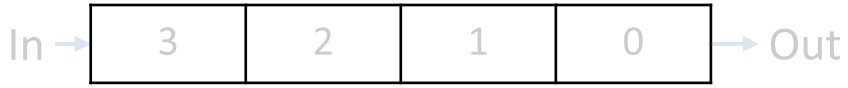
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

Last-In-First-Out (LIFO)

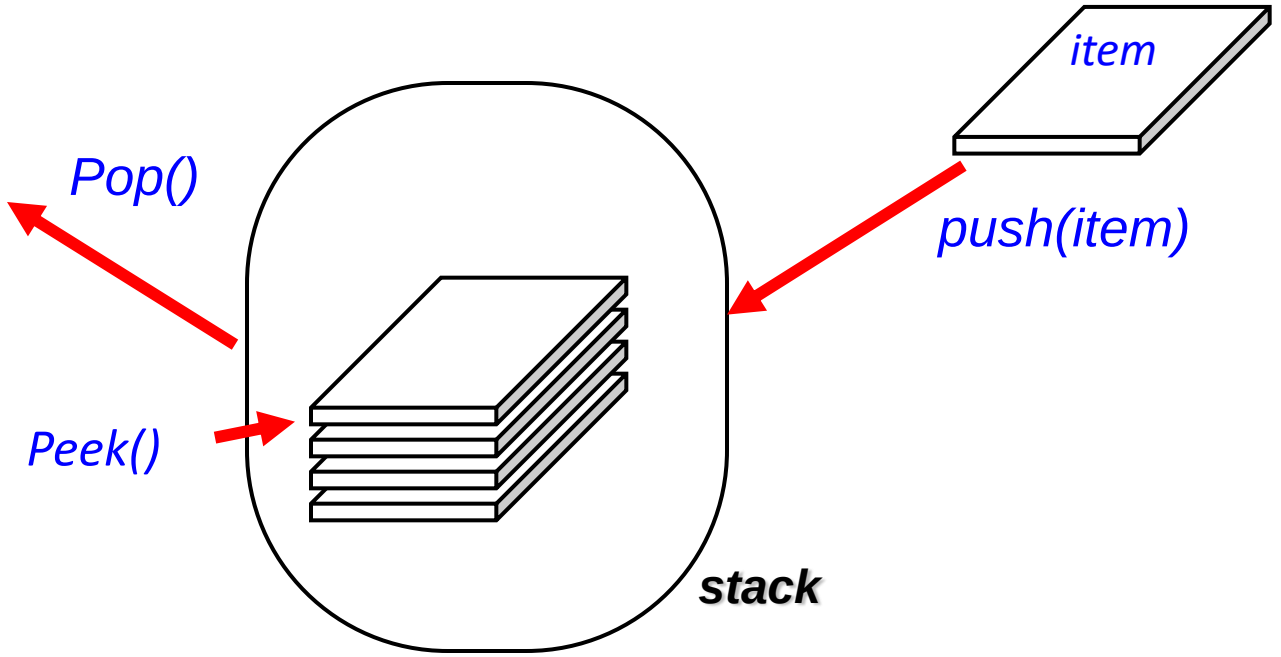


First-In-First-Out (FIFO)



Operations

- A Stack is a collection of data that is accessed in a Last-In-First-Out (LIFO) manner
- Major operations: “push”, “pop”, and “peek”.



Uses

- Calling a function
  - Before the call, the state of computation is saved on the stack so that we will know where to resume
- Recursion
- Matching parentheses
- Evaluating arithmetic expressions (e.g. a + b – c) :
  - postfix calculation
  - Infix to postfix conversion
- Traversing a maze

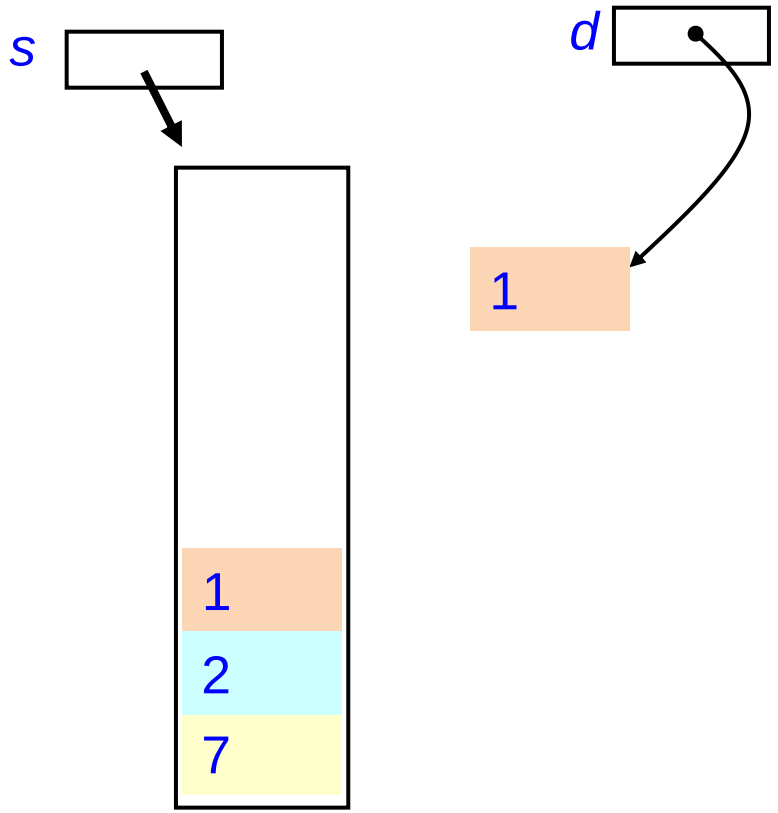
Interface

```
#define MAX 10  // Maximum size of the stack

class Stack { // ARRAY
private:
    int top;
    int arr[MAX]; // can replace int by an abstract item



public:
    bool isEmpty() const;
    bool isFull() const;

    void push(int item);
    void pop();

    int peek() const;
};
```

```
class Stack { // SINGLY LINKED LIST
private:
    struct Node {
        int data;
        Node* next;
        Node(int value) : data(value), next(nullptr) {}
    };
    Node* top;

public:
    bool isEmpty() const;
    bool isFull() const;

    void push(int item);
    void pop();

    int peek() const;
};
```

Usage

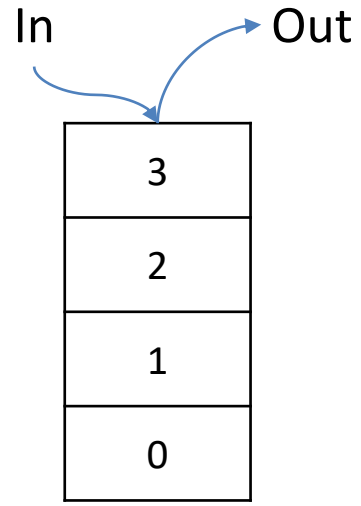
```
Stack s = new Stack();
s.push(7);
s.push(2);
s.push(1);
d = s.peek();
s.pop();
s.push(5);
s.pop();
```



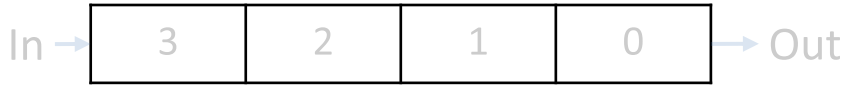
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

Last-In-First-Out (LIFO)

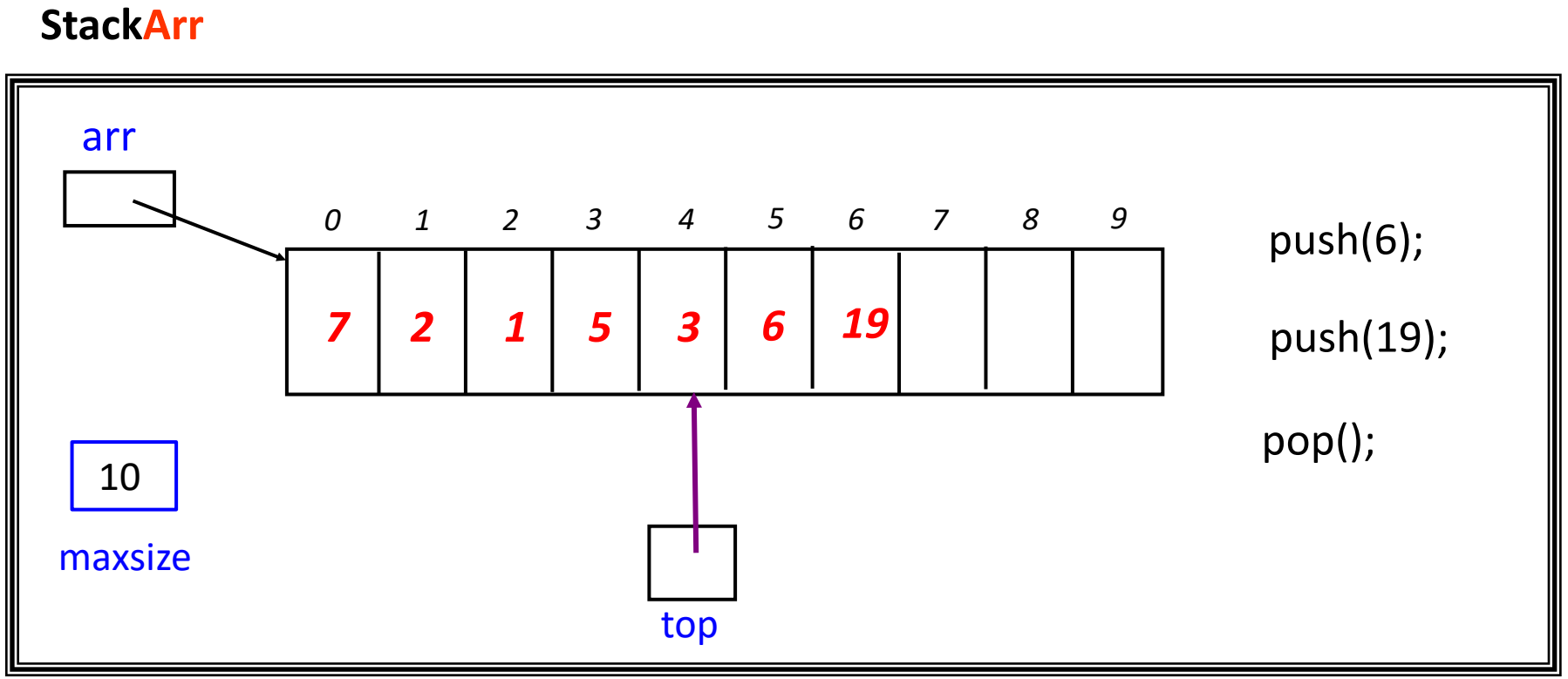


First-In-First-Out (FIFO)



Example

- Use an Array with a top index pointer



C++ source code (1/3)

```
#define MAX 100  // Maximum size of the stack

class Stack {
private:
    int top;
    int arr[MAX];

public:
    Stack() {
        top = -1;
    }

    bool isEmpty() const {
        return top == -1;
    }

    bool isFull() const {
        return top == MAX - 1;
    }

void push(int value) {
    if (isFull()) {
        std::cout << "Stack Overflow\n";
        return;
    }
    arr[++top] = value;
}
```

C++ source code (2/3)

```
void pop() {
    if (isEmpty()) {
        std::cout << "Stack Underflow\n";
        return;
    }
    --top;
}

int peek() const {
    if (isEmpty()) {
        std::cout << "Stack is empty\n";
        return -1;
    }
    return arr[top];
}

void display() const {
    if (isEmpty()) {
        std::cout << "Stack is empty\n";
        return;
    }
    std::cout << "Stack elements: ";
    for (int i = top; i >= 0; i--) {
        std::cout << arr[i] << " ";
    }
    std::cout << "\n";
}
};
```

C++ source code (3/3): running

```
int main() {
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    s.display();

    s.pop();
    s.display();

    cout << "Top element: " << s.peek() <<endl;

    return 0;
}
```

Stack elements: 30 20 10

Stack elements: 20 10

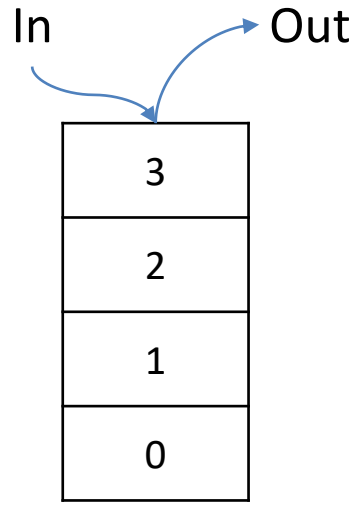
Top element: 20

1. `s.push(10), s.push(20), s.push(30)` → stack becomes [10, 20, 30] (with 30 at the top)
2. First `display()` → prints top to bottom → 30 20 10
3. `s.pop()` → removes 30 → stack becomes [10, 20]
4. Second `display()` → prints 20 10
5. `peek()` shows top → 20

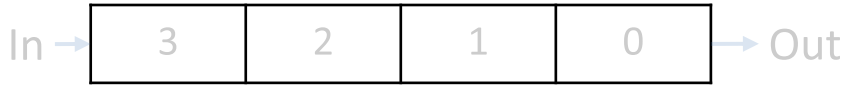
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

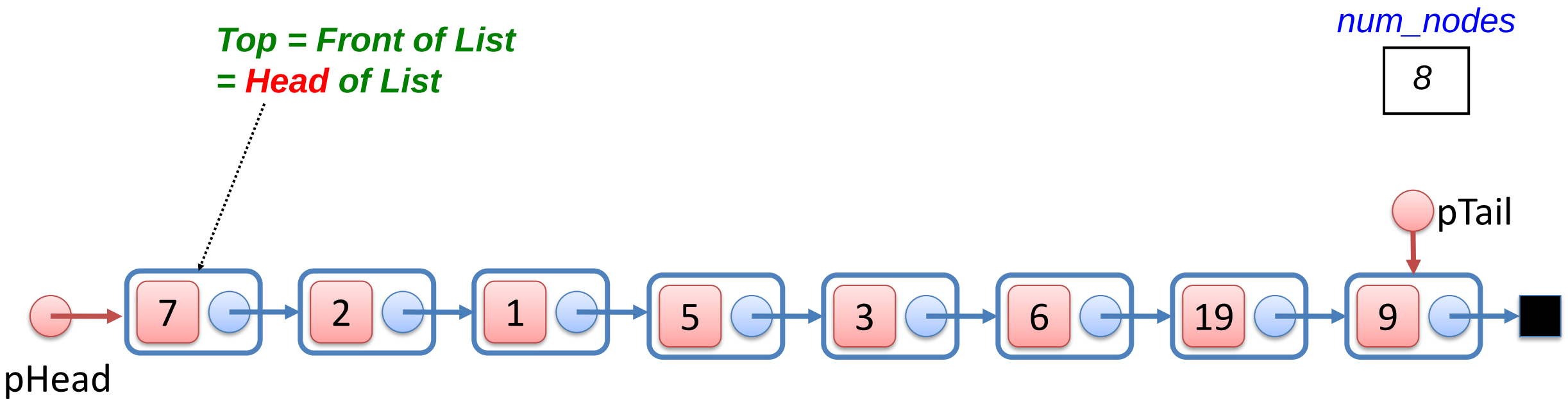
Last-In-First-Out (LIFO)



First-In-First-Out (FIFO)



Example: StackLL



C++ source code (1/3)

```
1.  class Stack {
2.  private:
3.      struct Node {
4.          int data;  // can replace int by an abstract item
5.          Node* next;

6.          Node(int value):data(value), next(nullptr) {}
7.      };
8.      Node* top;

9.  public:
10.     Stack() : top(nullptr) {}

11.     ~Stack() {
12.         while (!isEmpty()) {
13.             pop();
14.         }
15.     }

16.     bool isEmpty() const {
17.         return top == nullptr;
18.     }
19.     void push(int value) {
20.         Node* newNode = new Node(value);
21.         newNode->next = top;
22.         top = newNode;
23.     }
24.     void pop() {
25.         if (isEmpty()) {
26.             cout << "Stack Underflow\n";
27.             return;
28.         }
29.         Node* temp = top;
30.         top = top->next;
31.         delete temp;
32.     }
33.     int peek() const {
34.         if (isEmpty()) {
35.             cout << "Stack is empty\n";
36.             return -1;
37.         }
38.         return top->data;
39.     }
```

C++ source code (2/3)

```
40.     void display() const {
41.         if (isEmpty()) {
42.             cout << "Stack is empty\n";
43.             return;
44.         }
45.         Node* current = top;
46.         cout << "Stack elements: ";
47.         while (current != nullptr) {
48.             cout << current->data <<" ";
49.             current = current->next;
50.         }
51.         cout << "\n";
52.     }
53. };
```

C++ source code (3/3): running

```
int main() {
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    s.display();

    s.pop();
    s.display();

    cout << "Top element: " << s.peek() <<endl;

    return 0;
}
```

Stack elements: 30 20 10

Stack elements: 20 10

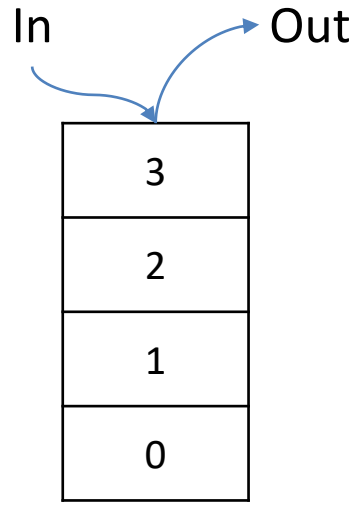
Top element: 20

1. `push(10)` → 10 is added
2. `push(20)` → 20 is added on top → 20 -> 10
3. `push(30)` → 30 is added on top → 30 -> 20 -> 10
4. `display()` shows top to bottom → 30 20 10
5. `pop()` removes top → 30 is removed
6. `display()` → now it's 20 10
7. `peek()` → shows top element → 20

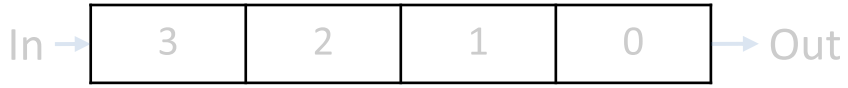
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

Last-In-First-Out (LIFO)



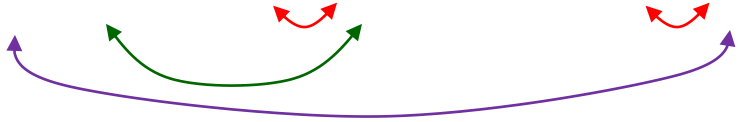
First-In-First-Out (FIFO)



Bracket matching (1/2)

Ensures that pairs of brackets are properly matched

An example: $\{a, (b + c[1]) * 2, d + e[3]\}$

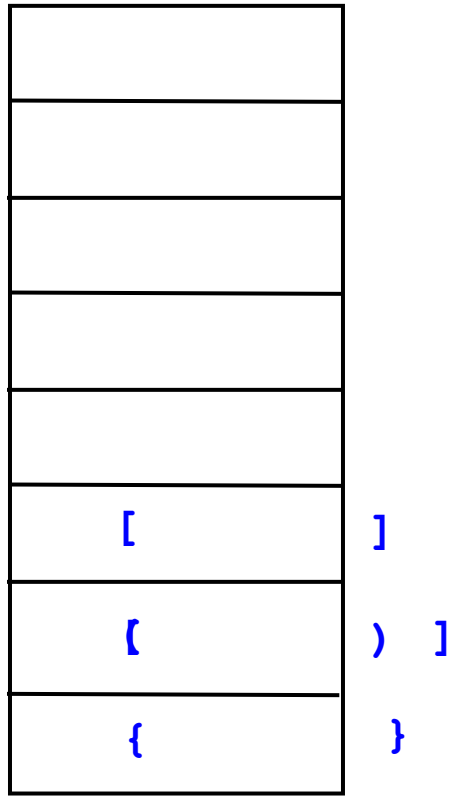


Incorrect examples:

```
( . . ) . . )        // too many close brackets

( . . ( . . )        // too many open brackets

[ . . ( . . ] . . )  // mismatched brackets
```

Bracket matching (1/2)

```
create empty stack
for every char read
{
  if open bracket then
      push onto stack
  if close bracket, then
      pop from the stack
      if doesn't match or underflow then
   flag error
}
if stack is not empty then flag error
```



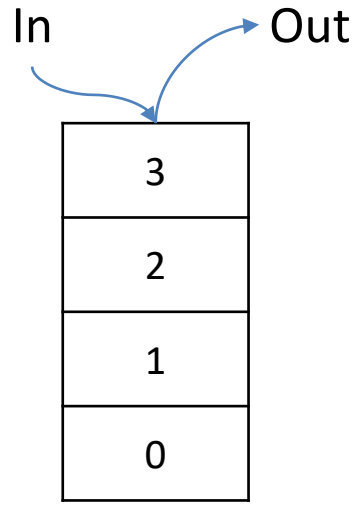
Example

{ a +(b + c [1]) * 2 * d + e [2] }

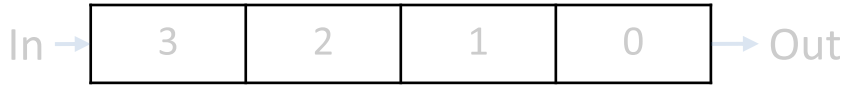
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

Last-In-First-Out (LIFO)



First-In-First-Out (FIFO)



Arithmetic Expression (1/7)

- Terms:
  - Expression: a = b + c * d
  - Operands: a, b, c, d
  - Operators: =, +, –, *, /, %
- Precedence Rules:
  - Operators follow specific priority levels, which determine the order in which they are evaluated.
  - For example, the ^ (power), ∗ (multiplication) and / (division) operators have higher precedence than + (addition) and – (subtraction).
  - When multiple operators share the same precedence level (like ∗ and /), they are evaluated from left to right.

Arithmetic Expression (2/7)

Infix : operand1 operator operand2

Prefix : operator operand1 operand2

Postfix : operand1 operand2 operator

Ambiguous, need () or precedence rules

Unique interpretation

postfix

infix

$1 + 2/3$

$(1 + 2)/3 \longrightarrow 1\ 2\ +\ \ 3\ /$

$1 + (2/3) \longrightarrow 1\ \ 2\ \ 3\ \ /\ +$

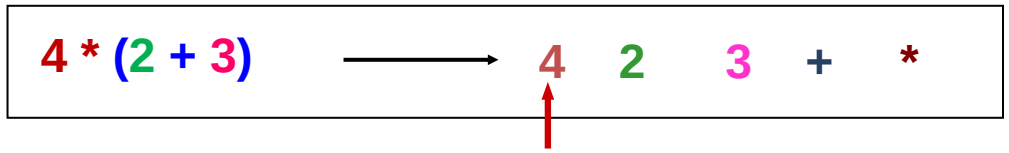
Arithmetic Expression (3/7)

Algorithm: Calculating Postfix expression with stack

```
create empty stack
for every char read
{
  if open bracket then
      push onto stack
  if close bracket, then
      pop from the stack
      if doesn't match or underflow then
   flag error
}
if stack is not empty then flag error
```

Infix: 4 * (2 + 3)

postfix: 4 2 3 + *

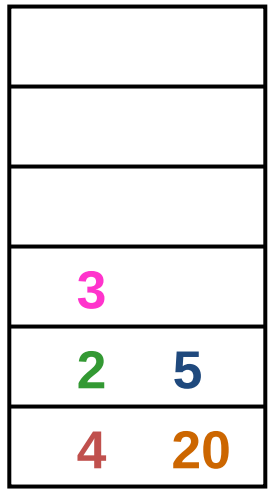


- 4 s.push(4)
- 2 s.push(2)
- 3 s.push(3)
- \+ arg2 = s.pop ()
  arg1 = s.pop ()
  s.push (arg1 + arg2)
- * arg2 = s.pop ()
  arg1 = s.pop ()
  s.push (arg1 * arg2)

arg1: 2 4

arg2: 3 5

Stack



Arithmetic Expression (4/7): algorithm

Infix to Postfix Conversion Steps (e.g., “A + B * C” to “A B C * +”):

1. Initialize an empty stack for operators and an empty list for output.
2. Scan the infix expression from left to right.
3. Operands (A, B, etc.): Add directly to the output.
4. Left Parenthesis “(“: push onto the stack.
5. Right Parenthesis “)”: pop and output from the stack until a left parenthesis is encountered. Discard the pair of parentheses.
6. Operators (+, –, *, /, etc.):
   - While the top of the stack has equal or higher precedence, and it’s not “(“, pop from the stack to output.
   - Then push the current operator onto the stack.
7. After scanning the expression: pop and append all remaining operators from the stack to the output.

Arithmetic Expression (5/7): algorithm in pseudocode

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b
c	+ (-	a b c

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b
c	+ (-	a b c
/	+ (- /	a b c

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b
c	+ (-	a b c
/	+ (- /	a b c
d	+ (- /	a b c d

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b
c	+ (-	a b c
/	+ (- /	a b c
d	+ (- /	a b c d
)	+ (-	a b c d /
	+ (	a b c d / -

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b
c	+ (-	a b c
/	+ (- /	a b c
d	+ (- /	a b c d
)	+ (-	a b c d /
	+ (	a b c d / -
	+	a b c d / -

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b
c	+ (-	a b c
/	+ (- /	a b c
d	+ (- /	a b c d
)	+ (-	a b c d /
	+ (	a b c d / -
	+	a b c d / -
*	+ *	a b c d / -

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b
c	+ (-	a b c
/	+ (- /	a b c
d	+ (- /	a b c d
)	+ (-	a b c d /
	+ (	a b c d / -
	+	a b c d / -
*	+ *	a b c d / -
e	+ *	a b c d / - e

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b
c	+ (-	a b c
/	+ (- /	a b c
d	+ (- /	a b c d
)	+ (-	a b c d /
	+ (	a b c d / -
	+	a b c d / -
*	+ *	a b c d / -
e	+ *	a b c d / - e
		a b c d / - e * +

Arithmetic Expression (7/7): algorithm by step-by-step

```
1.   Algorithm InfixToPostfix(expression)
2.       Create an empty stack for operators
3.       Create an empty output string (postfix)

4.       For each character ch in expression:
5.           If ch is an operand:
6.               Add ch to postfix
7.           Else if ch is '(':
8.               push ch onto the stack
9.           Else if ch is ')':
10.              While top of stack is not '(':
11.                  pop from stack and add to postfix
12.              pop '(' from the stack
13.          Else if ch is an operator:
14.              While stack is not empty AND
                 precedence(top of stack) >=
                 precedence(ch) AND top is not '(':
15.                  pop from stack and add to postfix
16.              push ch onto the stack

17.      While stack is not empty:
18.          pop from stack and add to postfix

19.      Return postfix
20.  End Algorithm
```

Example: a + (b - c / d) * e

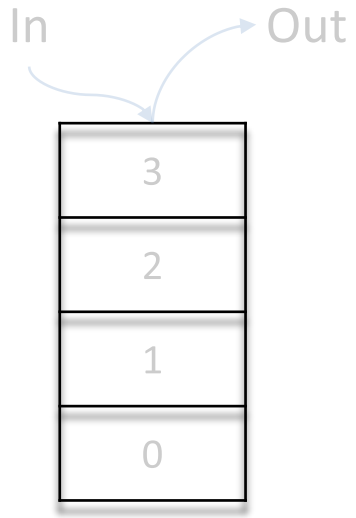
ch	Stack (bottom to top)	postfixExp
a		a
+	+	a
(	+ (	a
b	+ (	a b
-	+ (-	a b
c	+ (-	a b c
/	+ (- /	a b c
d	+ (- /	a b c d
)	+ (-	a b c d /
	+ (	a b c d / -
	+	a b c d / -
*	+ /	a b c d / -
e	+ /	a b c d / - e
		a b c d / - e * +

ch	Stack	Postfix Expression
a		a
+	+	a
(	(	a
b	+	ab
-	-	ab
c	(	abc
	+	
/	/	abc
d	-	abcd
	(	
	+	
)	-	abcd/
	(	
	+	
	(	abcd/-
	+	
	+	abcd/-
*	*	abcd/-
e	+	abcd/-e
		abcd/-e*+

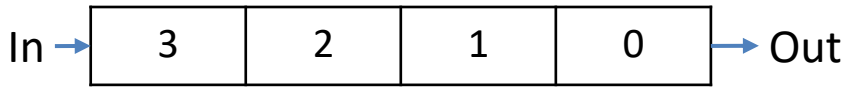
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

Last-In-First-Out (LIFO)

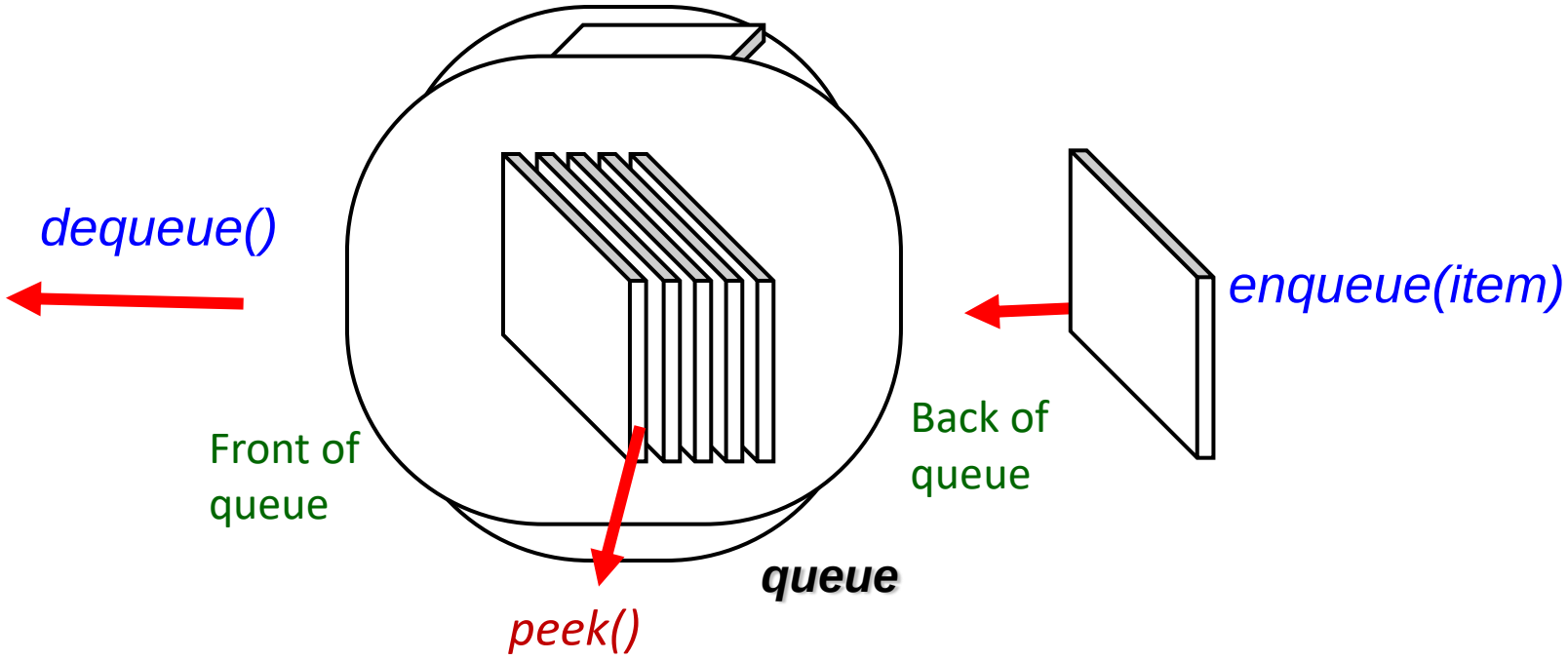


First-In-First-Out (FIFO)



Operations

- A Queue is a collection of data that is accessed in a First-In-First-Out (FIFO) manner.
- Major operations: "dequeue" (poll), "enqueue" (offer), and "peek".



Uses

- Print queue
- Simulations
- Breadth-first traversal of trees
- Checking palindromes - for illustration only as it is not a real application of queue.

Interface

```
#define MAX 10  // Maximum size of the stack

class Stack { // ARRAY
private:
    int top;
    int arr[MAX]; // can replace int by an abstract item



public:
    bool isEmpty() const;
    bool isFull() const;

    void push(int item);
    void pop();

    int peek() const;
};
```

```
class Stack { // SINGLY LINKED LIST
private:
    struct Node {
        int data;
        Node* next;
        Node(int value) : data(value), next(nullptr) {}
    };
    Node* top;

public:
    bool isEmpty() const;
    bool isFull() const;

    void push(int item);
    void pop();

    int peek() const;
};
```

Interface

```
#define MAX 100  // Maximum size of the stack

class Queue { // ARRAY
private:
    int arr[MAX];
    int front;
    int rear;

public:
    Queue();     // Constructor
    ~Queue();    // Destructor
    bool isEmpty() const; // Check if queue is empty
    bool isFull() const; // Check if queue is full

    void enqueue(int value); // Enqueue operation
    void dequeue(); // Dequeue operation

    int peek() const; // Get the front element
};
```

```
class Queue { // SINGLY LINKED LIST
private:
    struct Node {
        int data;
        Node* next;

        Node(int value) : data(value), next(nullptr) {}
    };
    Node *front, *rear;
public:
    Queue() : front(nullptr), rear(nullptr) {}
    ~Queue();    // Destructor
    bool isEmpty() const;

    void enqueue(int item);
    void dequeue();
    int peek() const;
};
```

Usage

Queue q = new Queue ();

⇨ q.enqueue("a");

⇨ q.enqueue("b");

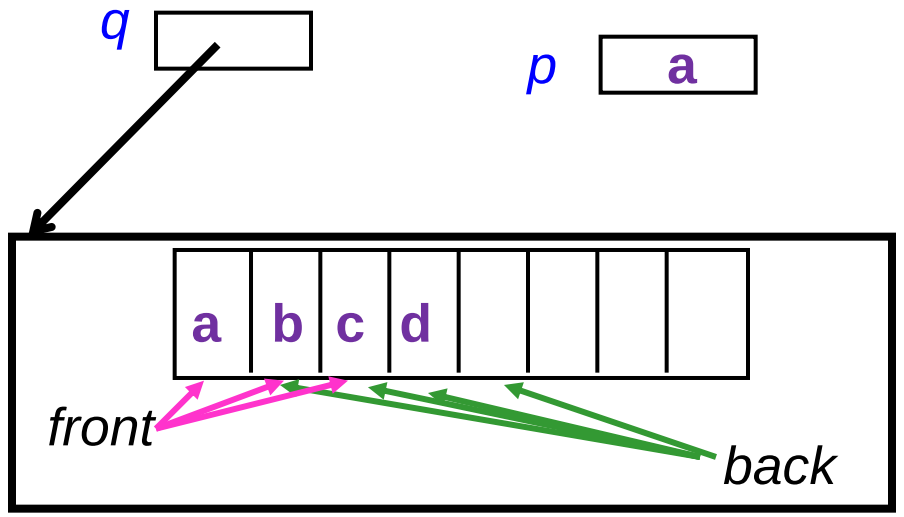
⇨ q.enqueue("c");

⇨ p = q.peek ();

⇨ q.dequeue();

⇨ q.enqueue("d");

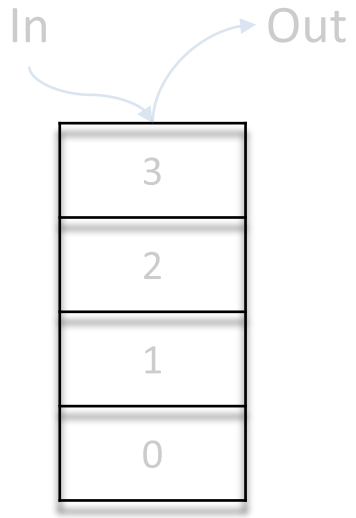
⇨ q.dequeue();



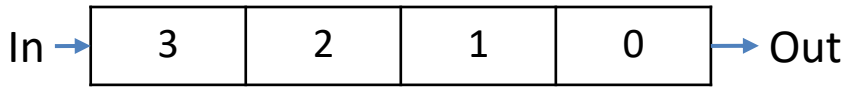
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

Last-In-First-Out (LIFO)

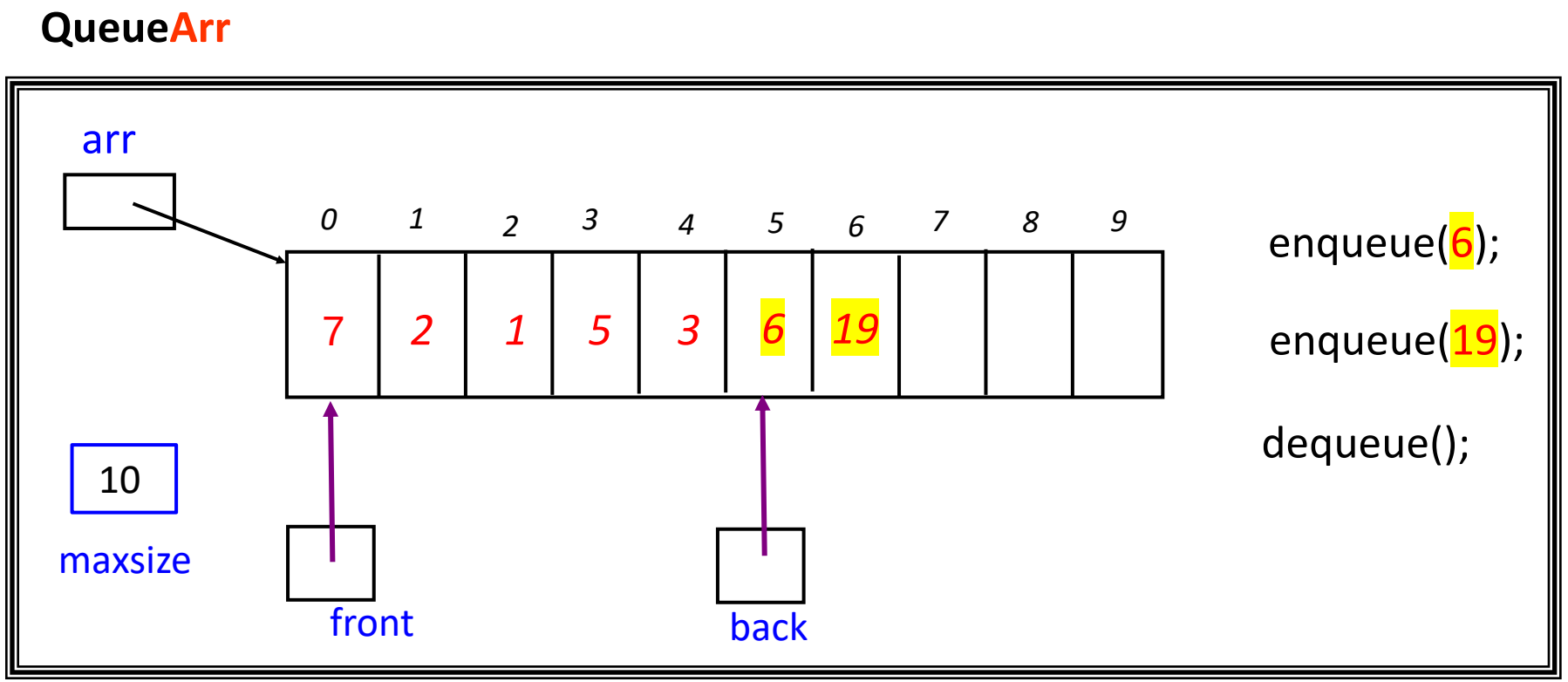


First-In-First-Out (FIFO)



Example

- Use an Array with front and back pointers



C++ source code (1/3)

```
#include <iostream>
#define MAX 100  // Maximum size of the queue

class Queue {
private:
    int arr[MAX];
    int front;
    int rear;

public:
    Queue() { // Constructor
        front = -1;
        rear = -1;
    }

    bool isEmpty()const{//Check if queue is empty
        return front == -1 || front > rear;
    }
```

```
// Check if queue is full
bool isFull() const {
    return rear == MAX - 1;
}

void enqueue(int value){ // Enqueue operation
    if (isFull()) {
        cout << "Queue Overflow\n";
        return;
    }

    if (isEmpty()) front = 0;
    arr[++rear] = value;
}
```

C++ source code (2/3)

```
void dequeue() { // Dequeue operation
    if (isEmpty()) {
        cout << "Queue Underflow\n";
        return;
    }

    front++;
}

int peek() const { // Get the front element
    if (isEmpty()) {
        cout << "Queue is empty\n";
        return -1;
    }

    return arr[front];
}
```

```
void display() const { // Display the queue
    if (isEmpty()) {
        cout << "Queue is empty\n";
        return;
    }

    cout << "Queue elements: ";
    for (int i = front; i <= rear; ++i) {
        cout << arr[i] << " ";
    }
    cout << "\n";
}
};
```

C++ source code (3/3): running

```
int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    q.display();  // Output: 10 20 30
    q.dequeue();
    q.display();  // Output: 20 30

    std::cout << "Front element: " << q.peek()
<< "\n";  // Output: 20
    return 0;
}
```

Queue elements: 10 20 30

Queue elements: 20 30

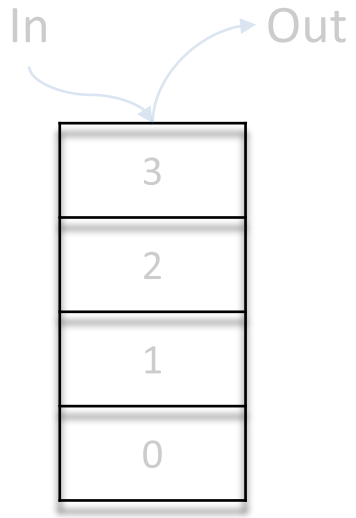
Front element: 20

1. `q.enqueue(10), q.enqueue(20), q.enqueue(30)` → queue becomes [10, 20, 30] (with 10 at the front)
2. First `display()` → prints front to rear → 10 20 30
3. `q.dequeue()` → removes 10 → queue becomes [20, 30]
4. Second `display()` → prints 20 30
5. `peek()` shows front → 20

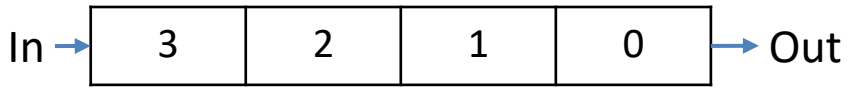
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

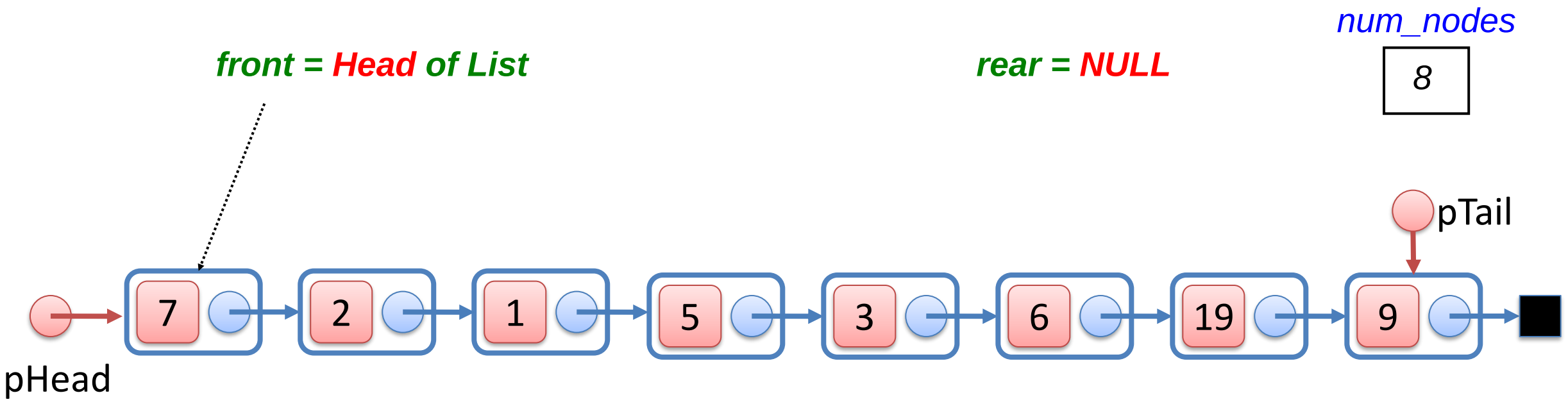
Last-In-First-Out (LIFO)



First-In-First-Out (FIFO)



Example: QueueLL



C++ source code

```
1.  class Queue {
2.      struct Node {
3.          int data;
4.          Node* next;
5.          Node(int val) : data(val), next(nullptr) {}
6.      };

7.      Node* front = nullptr;
8.      Node* rear = nullptr;

9.  public:
10.     ~Queue() { while (front) dequeue(); }

11.     bool isEmpty() const { return front == nullptr; }

12.     void enqueue(int val) {
13.         Node* node = new Node(val);
14.         if (rear) rear->next = node;
15.         else front = node;
16.         rear = node;
17.     }
18.     void dequeue() {
19.         if (isEmpty()) { cout << "Underflow\n";
    return; }
20.         Node* temp = front;
21.         front = front->next;
22.         if (!front)
                    rear = nullptr;
23.         delete temp;
24.     }

25.     int peek() const {
26.         if (isEmpty()) { cout << "Empty queue\n";
    return -1; }
27.         return front->data;
28.     }

29.     void display() const {
30.         for (Node* cur = front; cur; cur = cur->next)
31.             cout << cur->data << " ";
32.         cout << "\n";
33.     }
34. };
```

C++ source code (3/3): running

```
int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    q.display();  // Output: 10 20 30
    q.dequeue();
    q.display();  // Output: 20 30

    std::cout << "Front element: " << q.peek()
<< "\n";  // Output: 20
    return 0;
}
```

Queue elements: 10 20 30

Queue elements: 20 30

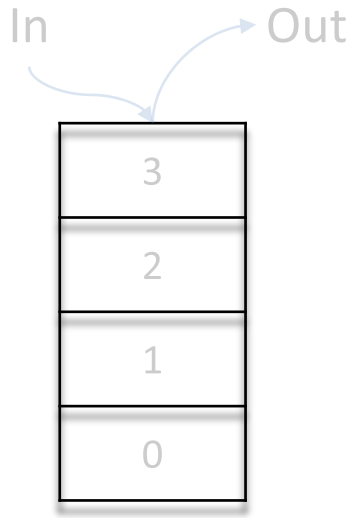
Front element: 20

1. `q.enqueue(10), q.enqueue(20), q.enqueue(30)` → queue becomes [10, 20, 30] (with 10 at the front)
2. First `display()` → prints front to rear → 10 20 30
3. `q.dequeue()` → removes 10 → queue becomes [20, 30]
4. Second `display()` → prints 20 30
5. `peek()` shows front → 20

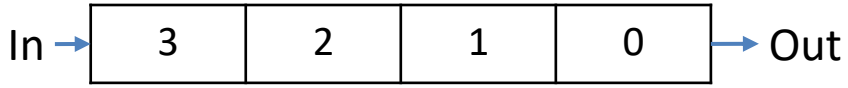
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

Last-In-First-Out (LIFO)



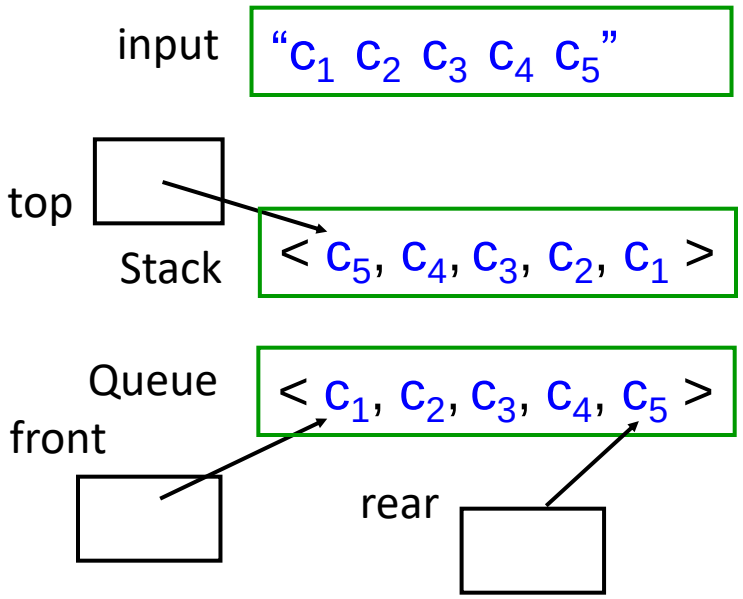
First-In-First-Out (FIFO)



- A string which reads the same either left to right, or right to left is known as a palindrome
  - Palindromes: “radar”, “deed”, “aibohphobia”
  - Non-palindromes: “data”, “little”

Algorithm
Given a string, use:
a Stack to *reverse* its order
a Queue to *preserve* its order

Check if the sequences are the same



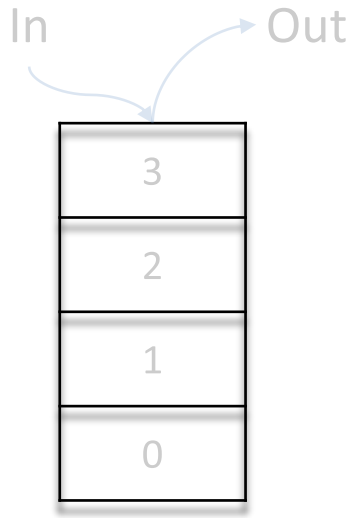
C++ source code

```
1.  #include <iostream>
2.  #include <stack>
3.  #include <queue>
4.  #include <cctype>
5.  using namespace std;

6.  bool isPalindrome(const string& str) {
7.      stack<char> s;
8.      queue<char> q;

9.      // Process characters: ignore non-alphanumeric
    and use lowercase
10.     for (char ch : str) {
11.         if (isalnum(ch)) {
12.             char lower = tolower(ch);
13.             s.push(lower);
14.             q.push(lower);
15.         }
16.     }
17.     // Compare stack and queue characters
18.     while (!s.empty()) {
19.         if (s.top() != q.front())
20.             return false;
21.         s.pop();
22.         q.pop();
23.     }
24.     return true;
25. }

26. int main() {
27.     string input;
28.     cout << "Enter a string: ";
29.     getline(cin, input);

30.     if (isPalindrome(input))
31.         cout << "The string is a palindrome.\n";
32.     else
33.         cout << "The string is not a palindrome.\n";

34.     return 0;
35. }
```

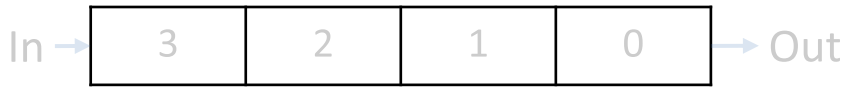
Outline

1. Stack
2. Stack Implementation via Array
3. Stack Implementation via Singly Linked List
4. Stack Applications
   - Bracket matching
   - Postfix calculation
5. Queue
6. Queue Implementation via Array
7. Queue Implementation via Singly Linked List
8. Application: Palindromes
9. Priority queue

Last-In-First-Out (LIFO)



First-In-First-Out (FIFO)



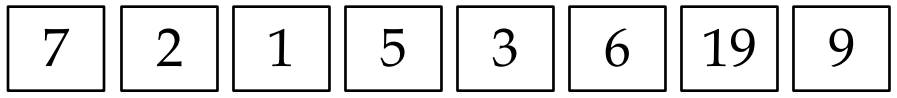
- Element in queue has a priority
  - Generally, value of element is considered for priority
  - The first element is the greatest or smallest

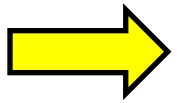
Important Basic Operations:

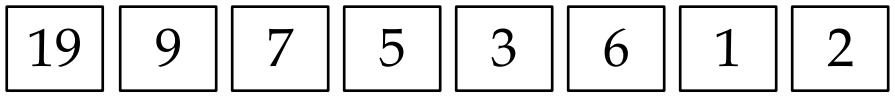
- Enqueue(x)
  - Put a new item x in the priority queue PQ (in some order)
- y ← Dequeue()
  - Return an item y that has the **highest priority** (key) in the PQ
  - If there are more than one item with highest priority, return the one that is inserted first (FIFO)

Example 1: Max-Heap (Default Priority Queue)

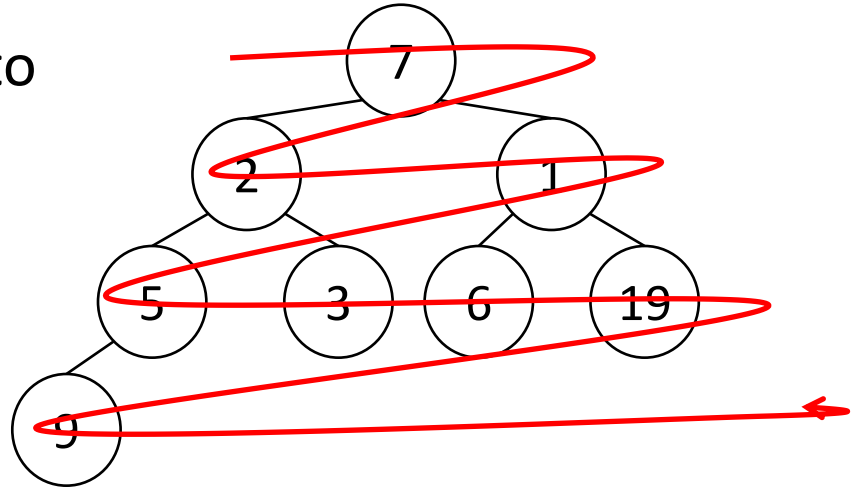
- What it does: Stores numbers in a priority queue.
- Behavior: Largest number has the highest priority.
- Use case: Useful when you always want to access the biggest number quickly (e.g., leaderboard scores, max-heap in heapsort).



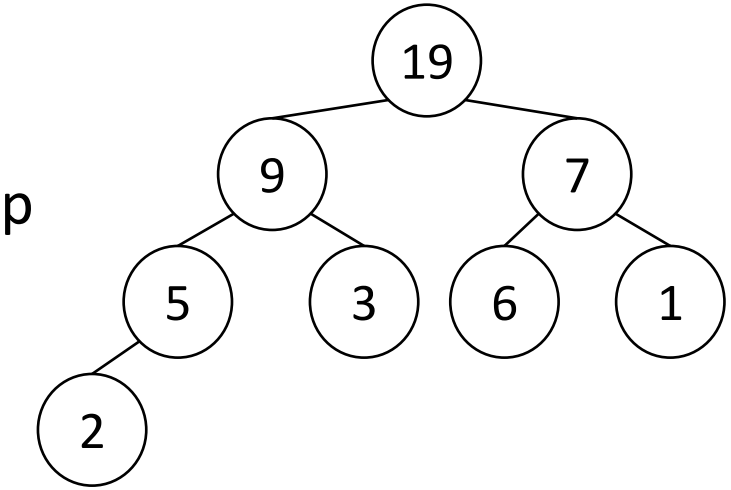




Convert to a heap but the result is NOT a heap

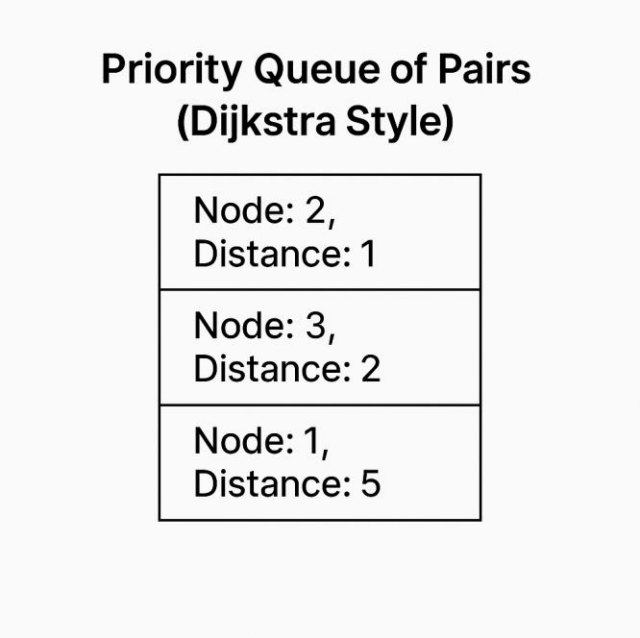


Max-heap



Example 2: Priority Queue of Pairs (Dijkstra Style)

- What it does: Stores pairs like (distance, node_id).
- Behavior: Prioritizes based on the first value (distance).
- Use case: Perfect for algorithms where you need to process elements in order of cost or time.



Example 3: Custom Comparator for Tasks

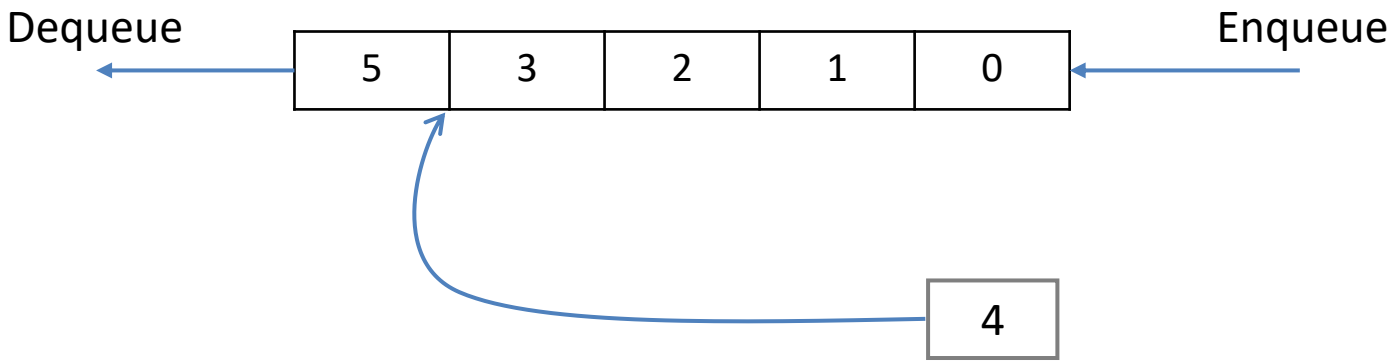
- What it does: Stores custom Task objects with a name and priority.
- Behavior: Higher priority value means higher processing order.
- Use case: Great for task scheduling or process management systems (like CPU job queues or to-do apps).

Example”

- Task: Code, Priority: 3
- Task: Meeting, Priority: 2
- Task: Email, Priority: 1

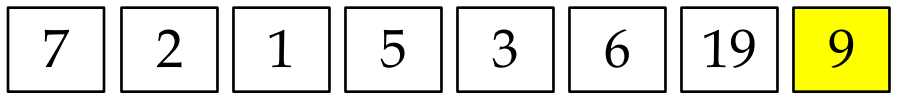
Insertion (1/2)

- An element is inserted to a correct position based on its priority value.

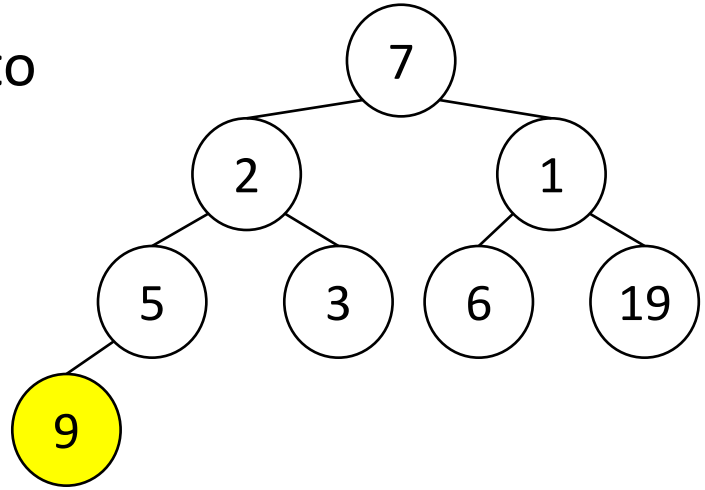


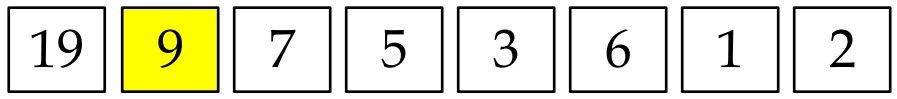
Insertion (2/2): example

- Given a priority queue max-heap $\{7,2,1,5,3,6,19\}$. Insert new item 9 to the queue.
- Requirement: create a max-heap after insertion.

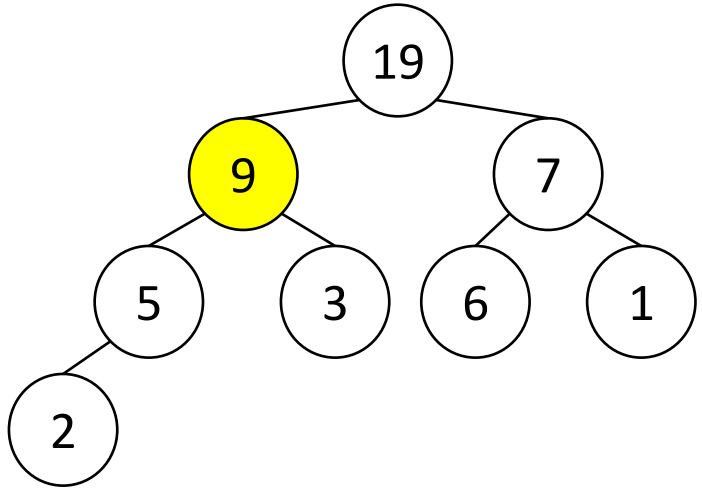


Convert to a heap but the result is NOT a heap



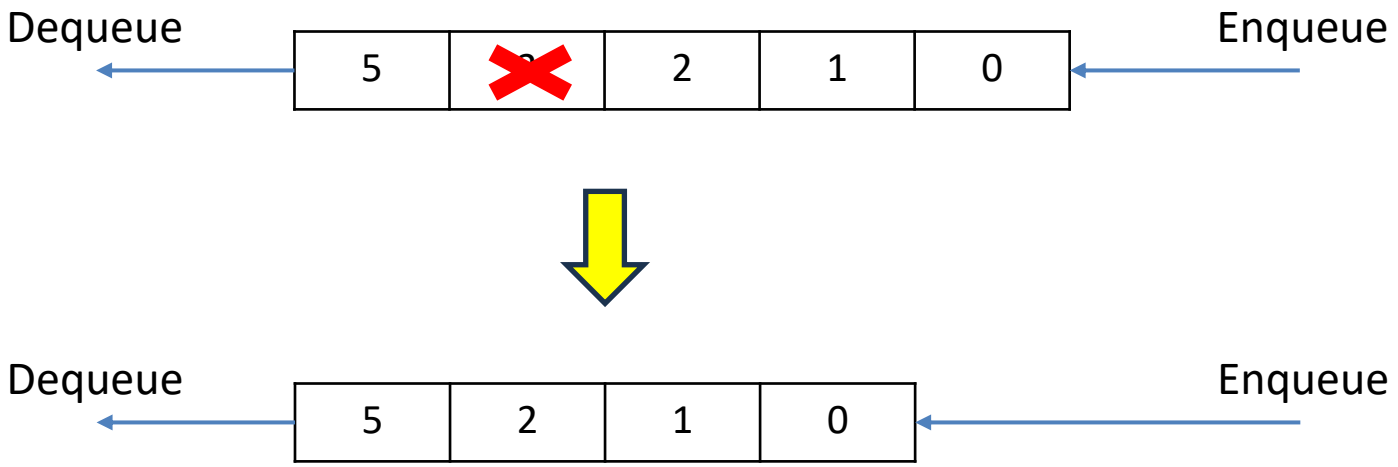


heapify()



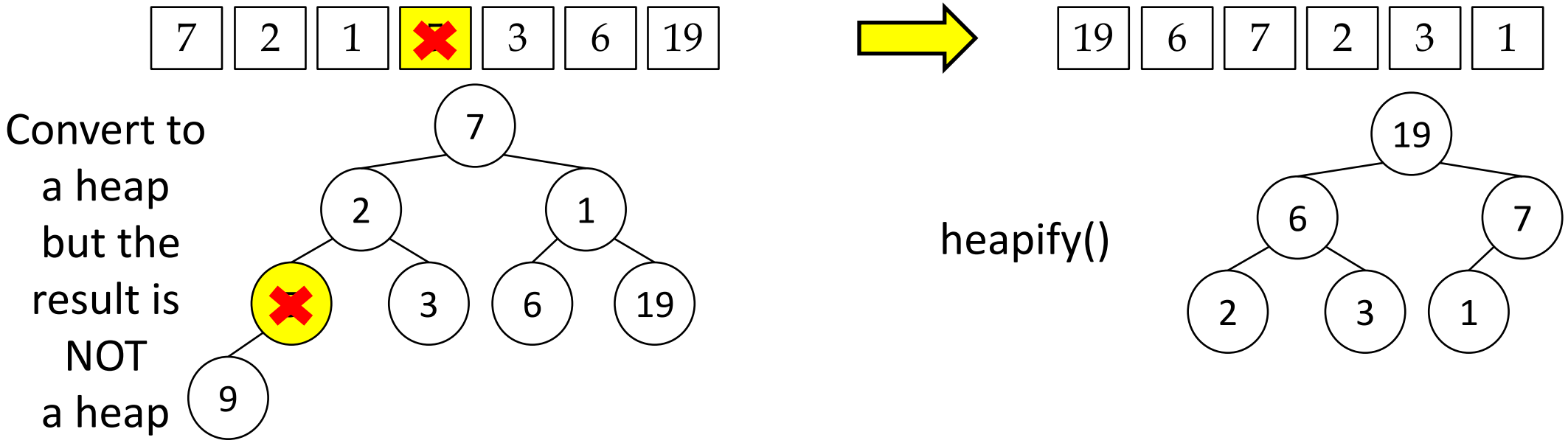
Deletion (1/2)

- An element is deleted and the updated priority queue remains a priority queue.



Deletion (2/2): example

- Given a priority queue max-heap $\{7,2,1,5,3,6,19\}$. Delete the item 5.
- Requirement: create a max-heap after deletion.



Q & A