SULEYMAN SULEYMANZADE

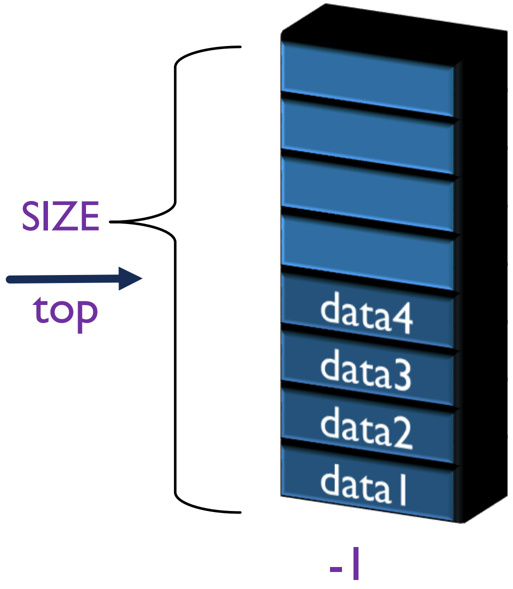
# LINEAR DATA STRUCTURES: STACK, QUEUE, DEQUE

DATA STRUCTURES AND ALGORITHMS

# LINEAR DATA STRUCTURE: STACK QUEUE DEQUEUE

## Content

- Stack data structure
  - Stack by using array
  - Stack by using linked list
  - Problems to solve with stack
- Queue
  - Implementation of queue data structure
- Dequeue
  - Implementation of dequeue data structure



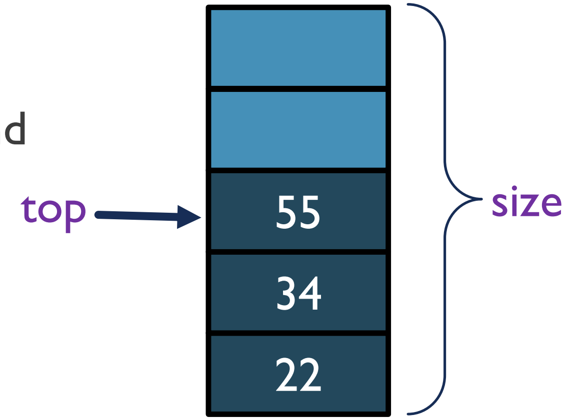

# STACK

## LINEAR DATA STRUCTURES: STACK, QUEUE, DEQUEUE

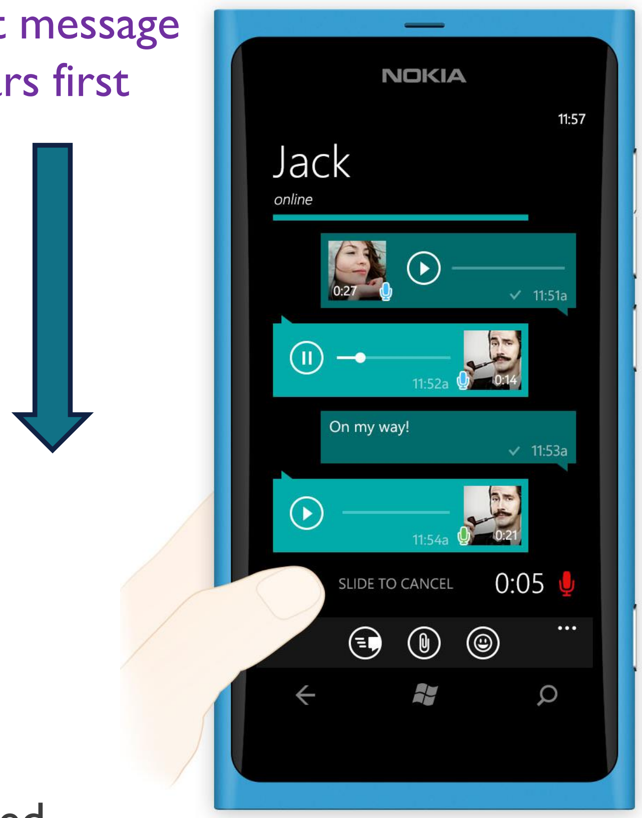
# STACK

## Definition

- A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data
- uses LIFO strategy
  - Last In – First out




The last message appears first

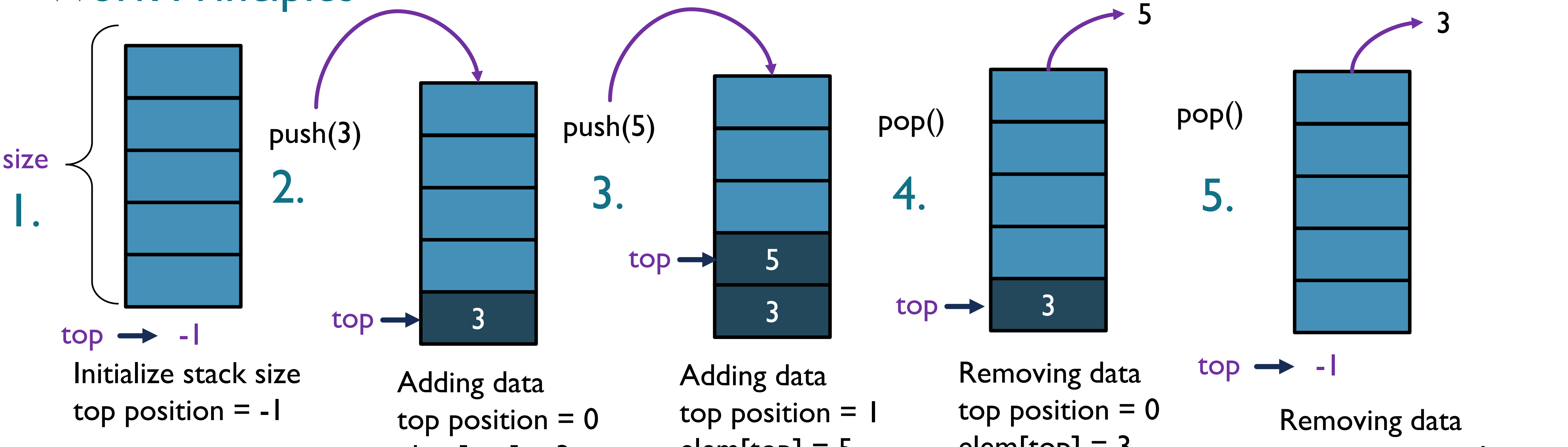



- Applications
  - Depth First Search (DFS)
  - Messengers
  - Tracking systems
  - Undo Redo Operations

- LIFO strategy
- push( ) - adds an element to the stack
- top( ) – always holds the pointer to the last added element
- pop() – removes the last element from the stack.

# STACK: STRUCTURE

## Work Principles

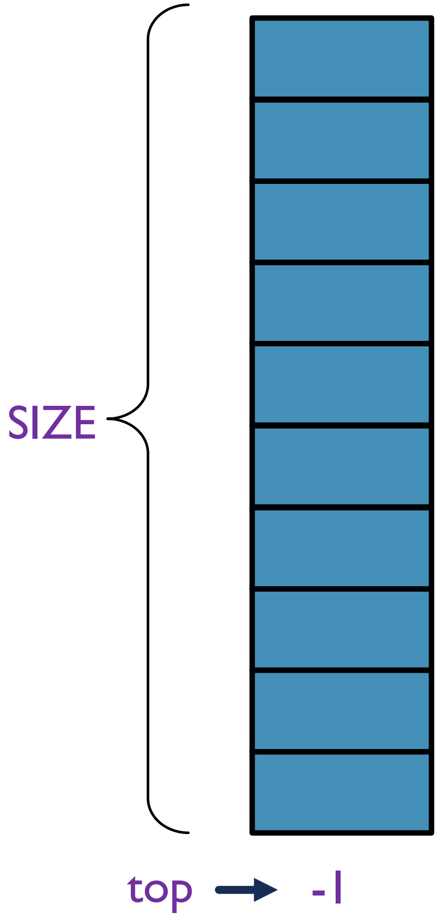



- LIFO strategy
- Push( ) - adds an element to the stack
- Top( ) – always holds the pointer to the last added element
- Pop() – removes the last element from the stack.

# STACK: ARRAY IMPLEMENTATION

```
#define SIZE 10

class stack{

    int *arr;
    int top;
    int capacity;
public:
    stack(int size = SIZE);
    void push(int);
    int pop();
    int peek();
    int size();
    bool isEmpty();
    bool isFull();
};
```

```
stack::stack(int size){
    arr = new int[size];
    capacity = size;
    top = -1;
}

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

bool stack::isFull(){
    return top == capacity - 1;
}

int stack::size(){
    return top + 1;
}
```



# STACK: ARRAY IMPLEMENTATION

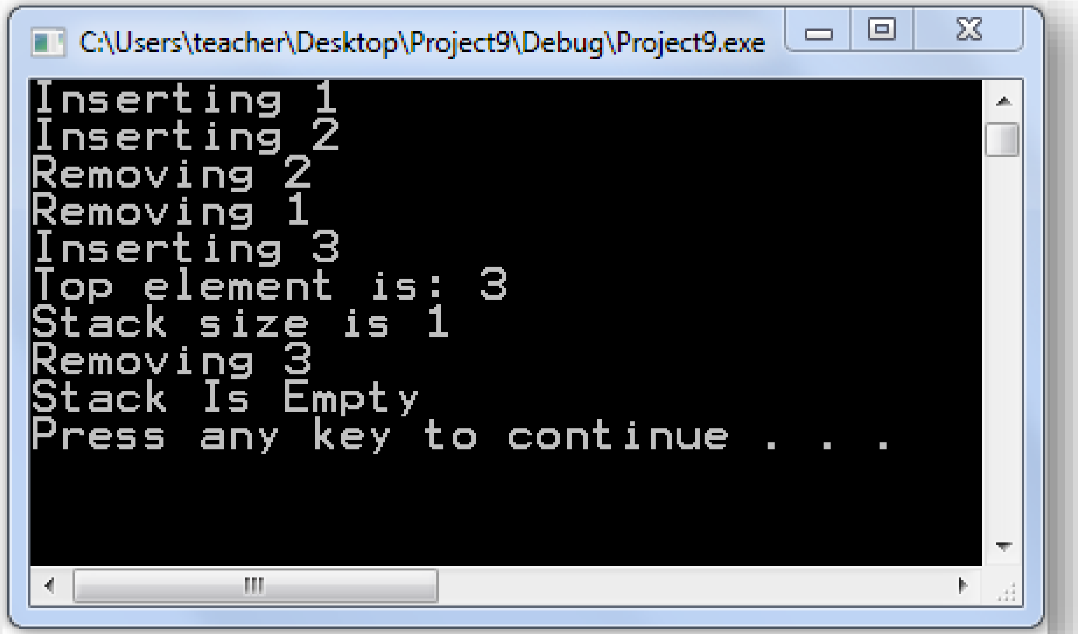
```
void stack::push(int x){
    if (isFull()){
        cout << "OverFlow\n";
        exit(EXIT_FAILURE);
    }
    cout <<"Inserting "<< x <<endl;
    arr[++top] = x;
}
```

```
int stack::pop(){
    if (isEmpty()){
        cout << "UnderFlow\nProgram Terminated\n";
    exit(EXIT_FAILURE);
    }
    cout << "Removing " << peek() << endl;
    return arr[top--];
}
```

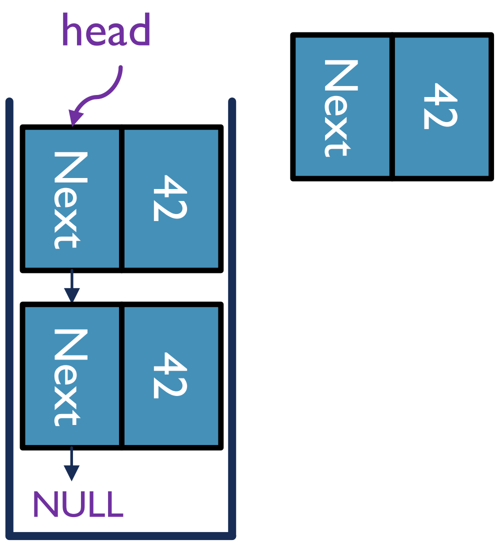
```
int stack::peek(){
    if (!isEmpty())
        return arr[top];
    else
        exit(EXIT_FAILURE);
}
```

# STACK: ARRAY IMPLEMENTATION

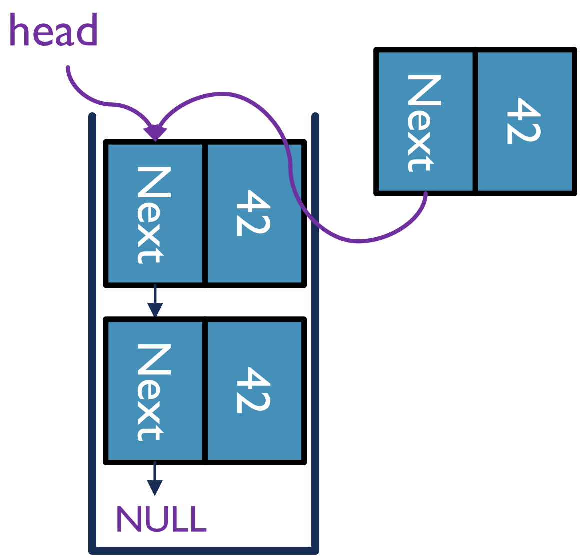
```
int main()
{
    stack pt(3);

    pt.push(1);
    pt.push(2);

    pt.pop();
    pt.pop();

    pt.push(3);
    cout << "Top element is: " << pt.peek() << endl;
    cout << "Stack size is " << pt.size() << endl;
    pt.pop();
    if (pt.isEmpty())
        cout << "Stack Is Empty\n";
    else
    cout << "Stack Is Not Empty\n";
    return 0;
}
```



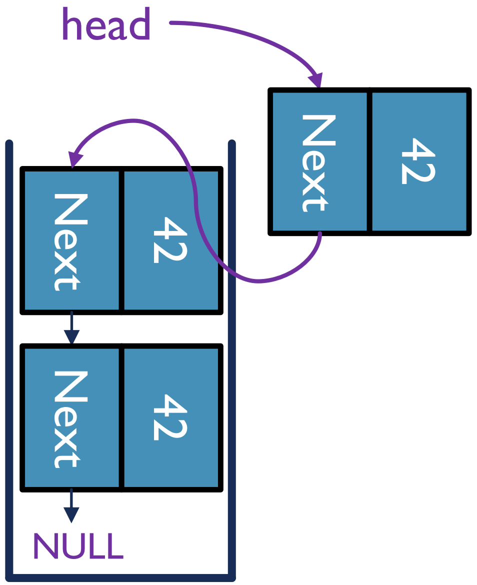
# STACK LINKED LIST: ALGORITHM




1.




2.

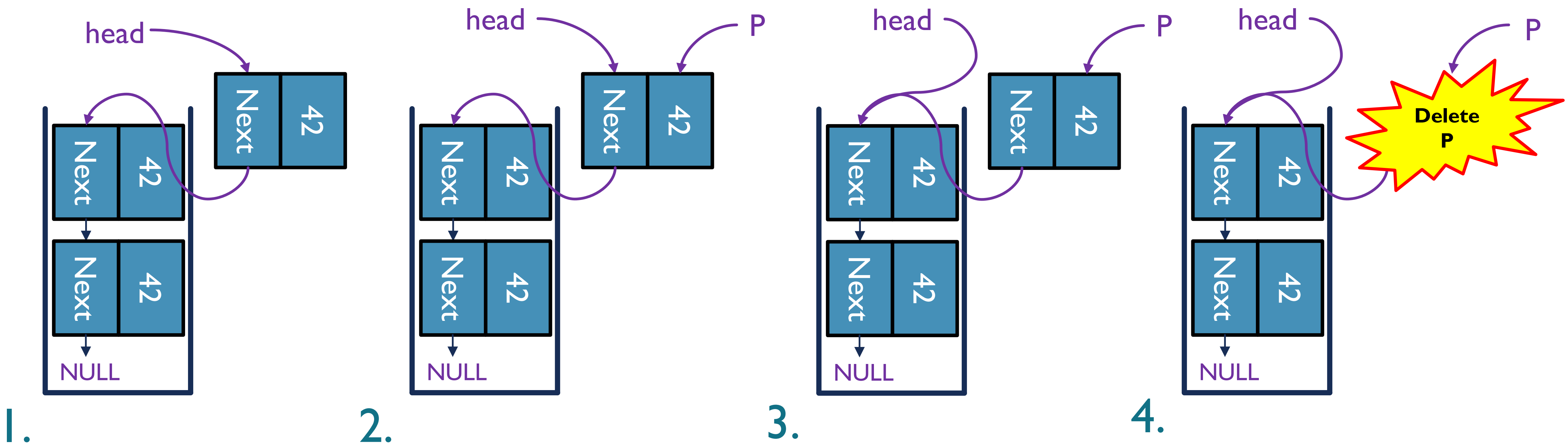



3.

- Create linked list object
- Attach the next pointer of created object to the head object
- Update the head pointer ( refer to the new object )

Push algorithm

# STACK LINKED LIST: ALGORITHM




- Create temp pointer to the head object
- Change the head pointer to the next item
- Delete temp pointer object. (free from the heap emmory)

Pop algorithm

# STACK: LINKED LIST IMPLEMENTATION

```
#include<iostream>
using namespace std;

struct Stack{
  int data;
  Stack* next;
};

Stack* create_node(int data){
  Stack* s = new Stack;
  s->data = data;
  s->next = NULL;
  return s;
}
```

```
void push(Stack** shead, int n){

  Stack* nn = create_node(n);
  nn->next = *shead;
  *shead = nn;
}

bool isempty(Stack* shead){

  return !shead;
}

void pop(Stack** shead){
  if(isempty(*shead))
    return;
  Stack* temp = *shead;
  *shead = (*shead)->next;
  delete temp;
}
```

```
int top(Stack* shead){
  if(isempty(shead))
    return -1e9;
  return shead->data;
}
int main(){

  Stack* s = NULL;
  push(&s, 12);
  push(&s, 55);
  push(&s, 123);

  while( !isempty(s)){

    cout<<top(s)<<endl;
    pop(&s);
  }
  system("pause");
}
```

# RIGHT BRACKETS

## Application 1

In C++ programs, we have the following delimiters: parentheses "(" and ")", square brackets "[" and "]", curly brackets "{" and "}", and comment delimiters "/*" and " */"

- Examples of C++ statements that use delimiters properly:
  - a = b + ( c - d ) * ( e – f )
  - g[10] = h[ i [ 9 ] ] + ( j + k ) * l
  - while (m < (n[8] + o)) { p = 7; /*initialize p */ r = 6; }

- These examples are statements in which mismatching occurs
  - a = b + (c - d) * (e - f)); while (m < (n[8] + o)) { p = 7; /*initialize p */ r = 6; }

< < > >( [ ( < > ) ] )[( )

( [ ] ) [ < > ] ( )

RIGH ORDER BRACKETS

( < ) >

( [ ] ) (

WRONG ORDER BRACKETS

# BRACKET

## Algorithm

| Stack | Nonblank Character Read | Input Left |
|---|---|---|
| empty | | s = t[5] + u / (v * (w + y)); |
| empty | s | = t[5] + u / (v * (w + y)); |
| empty | = | t[5] + u / (v * (w + y)); |
| empty | t | [5] + u / (v * (w + y)); |
| [ | [ | 5] + u / (v * (w + y)); |
| [ | 5 | ] + u / (v * (w + y)); |
| empty | ] | + u / (v * (w + y)); |
| empty | + | u / (v * (w + y)); |
| empty | u | / (v * (w + y)); |
| empty | / | (v * (w + y)); |
| ( | ( | v * (w + y)); |
| ( | v | * (w + y)); |
| ( | * | (w + y)); |
| ( ( | ( | w + y)); |
| ( ( | w | +y)); |
| ( ( | + | y)); |
| ( ( | y | )); |
| ( | ) | ); |
| empty | ) | ; |
| empty | ; | |

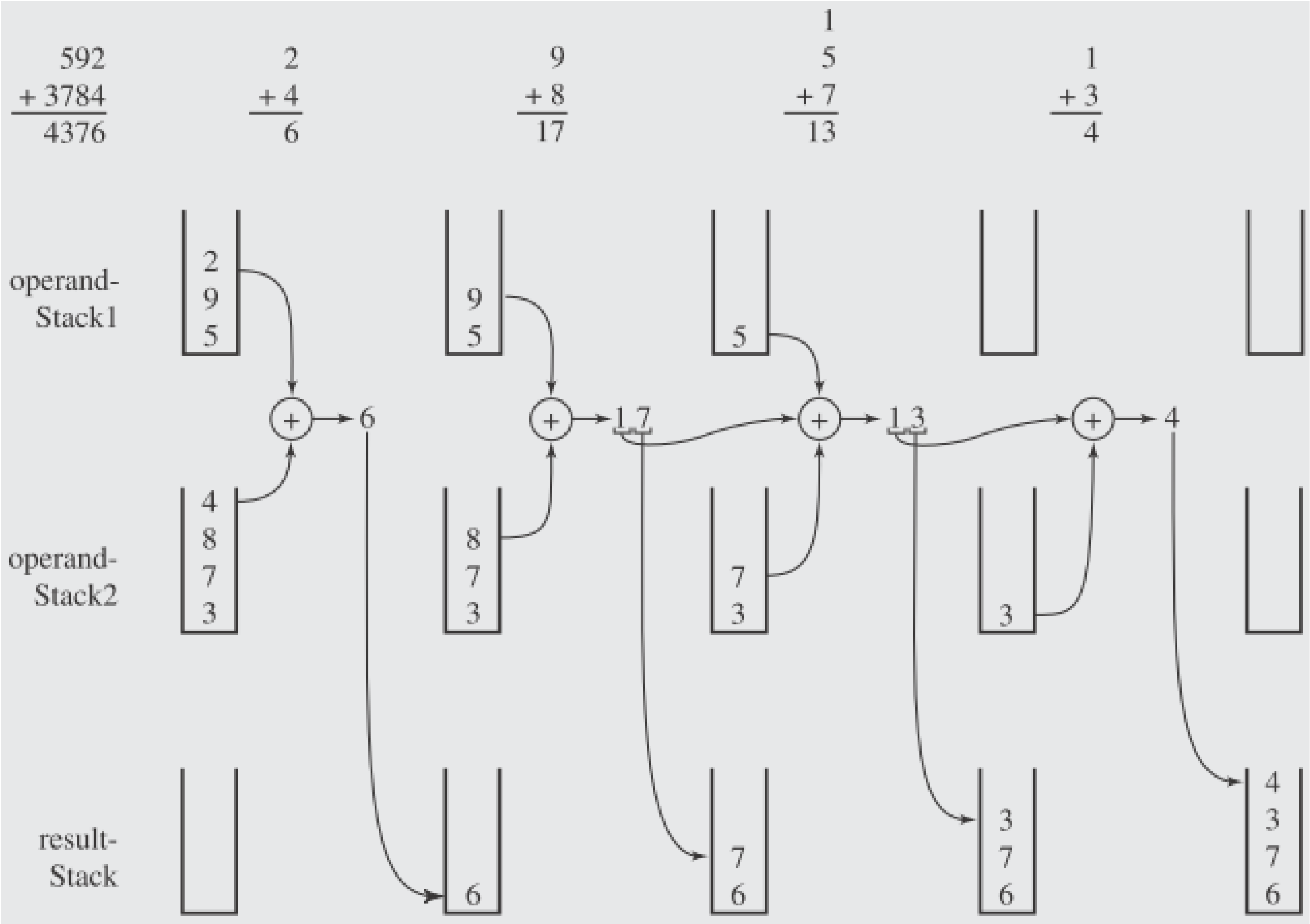
```
delimiterMatching(file)
  read character ch from file;
    while not end of file
        if ch is '(', '[', or '{'
                push(ch);
        else if ch is ')', ']', or '}'
             if ch   and popped off delimiter do not match
                          failure;
             else if  ch  is '/'
                  read the next character;
                        if this character is '*' skip all characters until "*/" is found and report an error
   if the end of file is reached before "*/" is encountered;
              else    ch    =    the character read in;
                      continue;    //    go to the beginning of the loop;
              else    ignore other characters;
                      read next character   ch   from   file;
              if      stack is empty
                          success;
              else    failure;
```

# STACK USAGE: ADDING A LARGE NUMBERS

## Algorithm

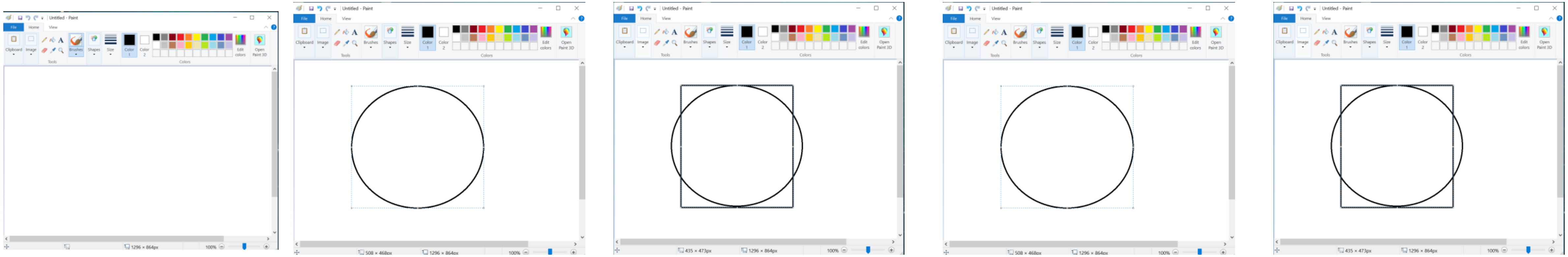
*addingLargeNumbers()*

*read the numerals of the first number*
*and store the numbers corresponding to*
*them on one stack;*
*read the numerals of the second number and*
*store the numbers corresponding to them on another stack;*
*carry = 0;*
*while at least one stack is not empty*
*pop a number from each nonempty stack*
*and add them to carry;*
*push the unit part on the result stack;*
*store carry in carry;*
*push carry on the result stack if it is not zero;*
*pop numbers from the result stack and display them;*




*source: Adam Drozdek: Data structures and Algorithms in C++*

# EXAMPLE OF STACK USAGE IN SOFTWARE: UNDO-REDO OPERATIONS



1. Draw circle

2. Draw Rectangle

3. Undo

4. Redo

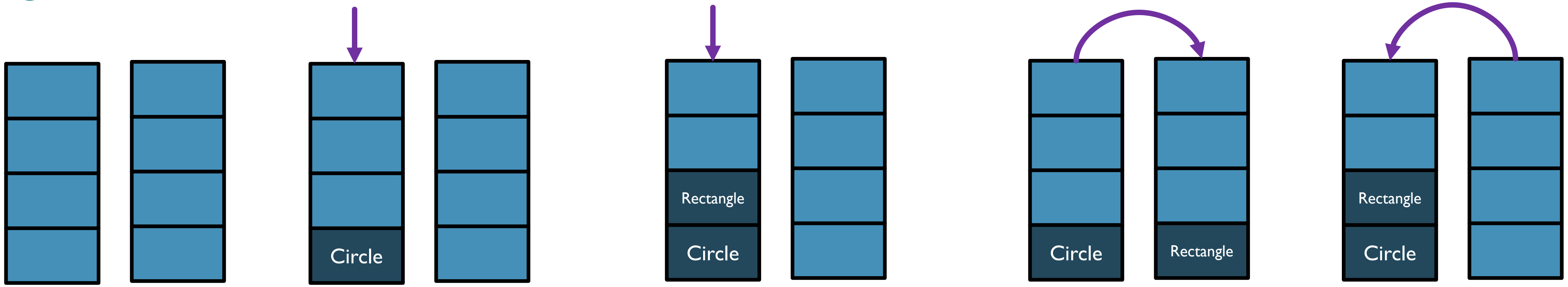
## Undo-Redo Algorithm

S1.push(Circle)

S1.push(Rectangle)

S1.pop(Rectangle)
S2.push(Rectangle)

S2.pop(Rectangle)
S1.push(Rectangle)




Undo stack | Redo stack | Undo stack | Redo stack | Undo stack | Redo stack | Undo stack | Redo stack | Undo stack | Redo stack

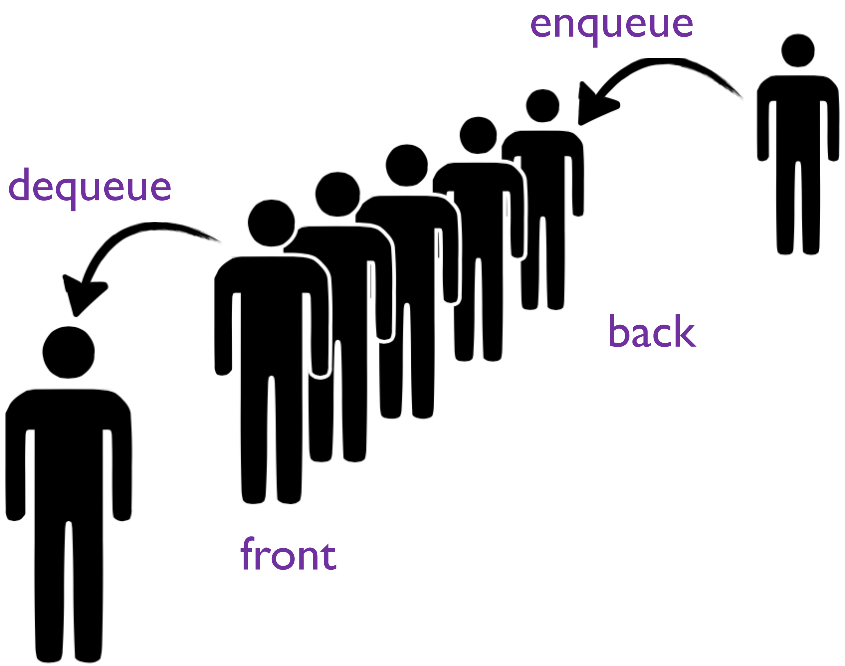
# QUEUE

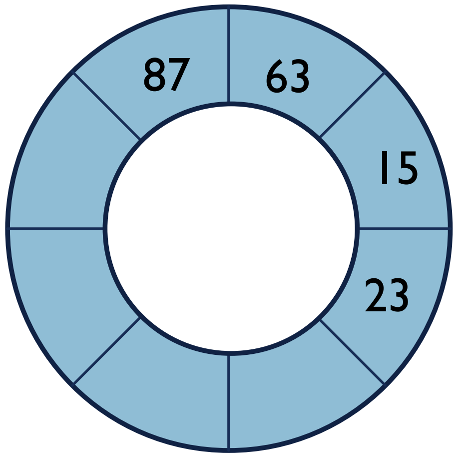
## LINEAR DATA STRUCTURES: STACK, QUEUE, DEQUEUE

# QUEUE DATA STRUCTURE

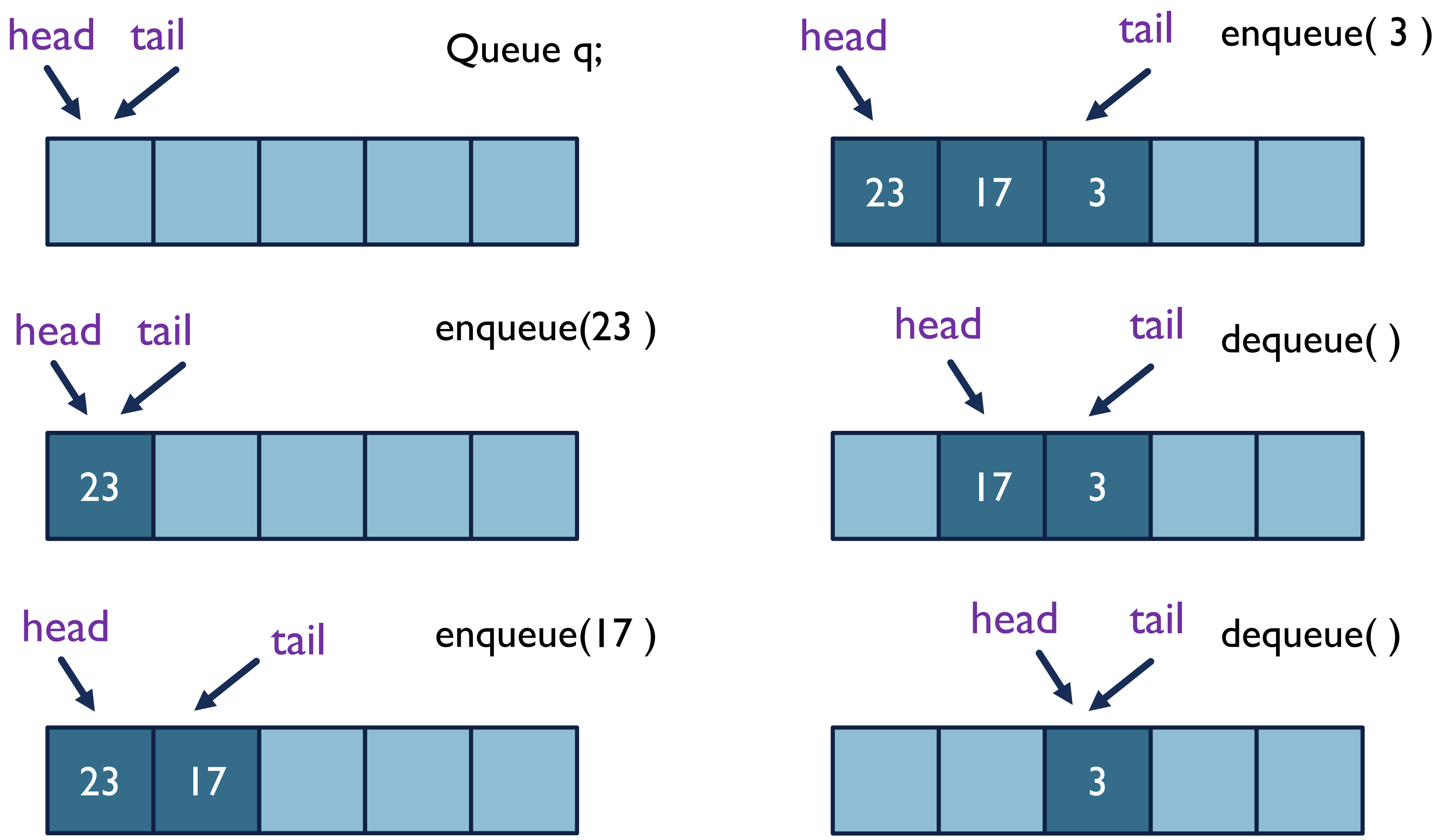
## Definition

- A queue is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front. Unlike a stack, a queue is a structure in which both ends are used: one for adding new elements and one for removing them uses LIFO strategy.
- LIFO – Last in First Out
- Types
  - Linear
  - Cyclic
- Applications
  - Process synchronization
  - Loaders
  - Scheduling's




# QUEUE LIFO PRINCIPLES




## Algorithm

- Operations
  - enqueue – adds element to the structure
  - dequeue – removes and returns element from the structure. While implementation it is non necessary to delete element from the queue. But only change the pointer of an element
- Problems with linear Queue
  - enqueue and dequeue – moves elements' interval to the right.
  - Queue size must be much bigger than bussible using of an elements
  - Solusion – Cyclic queue.

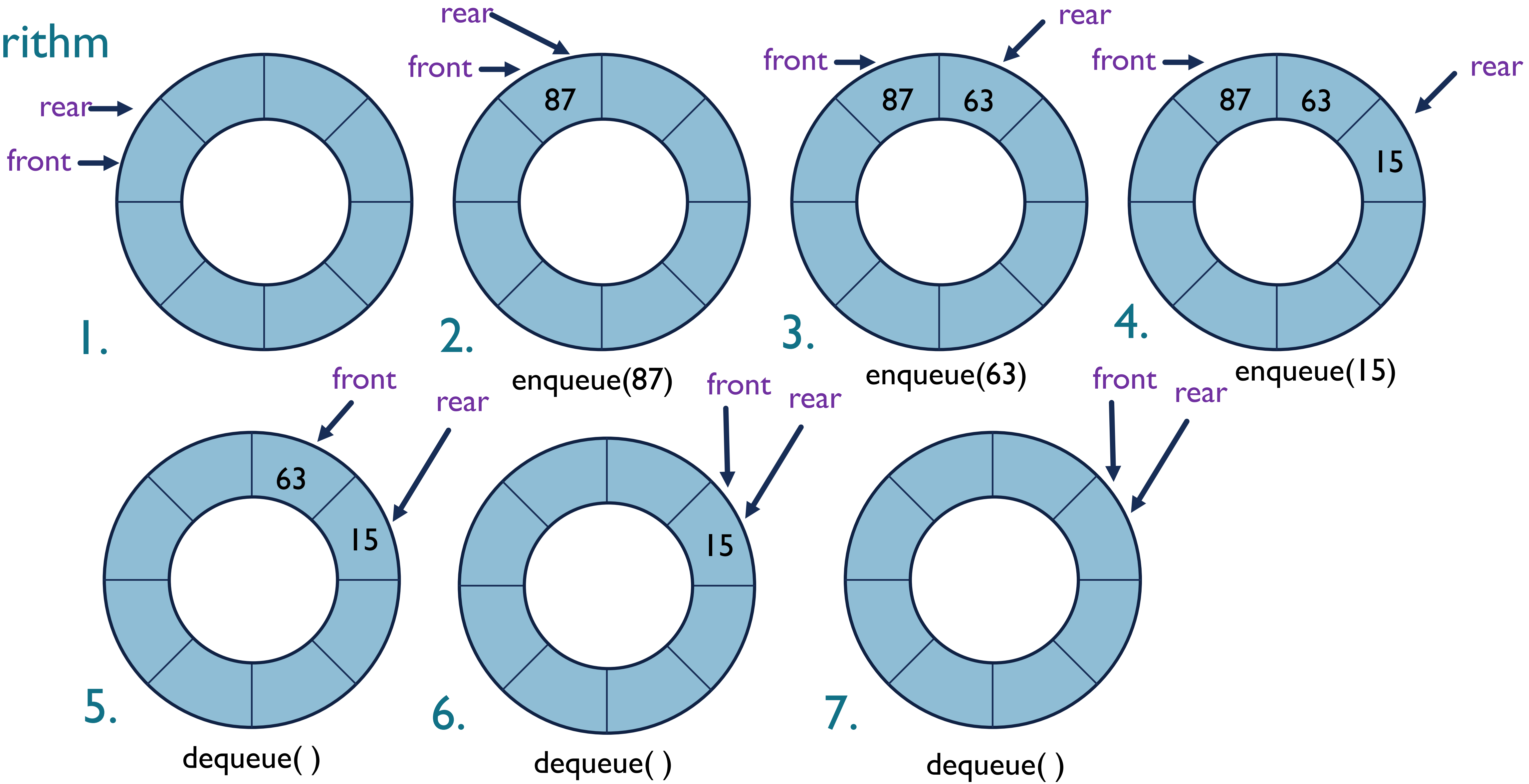
# SIMPLE QUEUE IMPLEMENTATION

```
class Queue {
private:
    int *a;
    int head;
    int tail;
public:
    Queue(int n):head(0),tail(0){
        a = new int[n];
    }
    void enqueue(int x);
    int  dequeue();
    bool is_empty();
    ~Queue(){
        delete[] a;
    }
};
```

```
void Queue::enqueue(int x) {
    a[tail] = x;
    tail++;
}
int Queue:: dequeue() {
    if (head != tail) {
        head++;
        return a[head - 1];
    }
    else{
        cout<<"Error: the queue is empty";
    }
}
bool Queue::is_empty() {
    return head == tail;
}
```

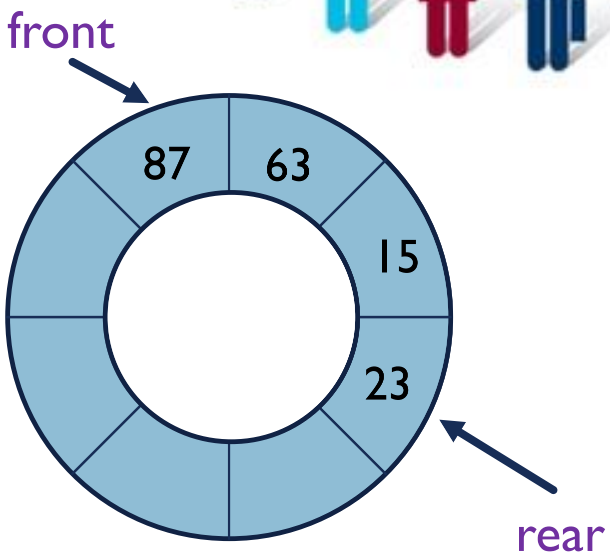
# CIRCULAR QUEUE

## Algorithm



# CYCLIC QUEUE

```
#include<iostream>
#define SIZE 5
using namespace std;

class cQueue {
   int arr[SIZE];
   int front, rear;
public :
   cQueue() {
      front = rear = -1;
   }
   void enqueue(int); // insert an element
                      //into queue

   void dequeue();   // Remove the front
                     //element from queue

   void display(); // display the queue
                   // elements
};
```



# CYCLIC QUEUE: IMPLEMENTATION

```
void cQueue :: enqueue(int data) {

   if (rear == -1) { // queue is empty
      front = rear = 0;
      arr[front] = data;
   }
   else {

      int pos = (rear + 1) % SIZE;
      if (pos == front) { // queue is full
         cout << "No space in queue ..." << endl;
         return;
      }
      else {

         rear = pos; // update rear
         arr[pos] = data; // insert the data in queue
      }
   }
}
```

```
void cQueue :: dequeue() {

   if (front == -1) { // queue is empty
      cout << "Queue is empty ... " << endl;
      return;
   }
   else {
      if (front == rear) { // only one element in queue
         front = rear = -1;
      }
      else {
         front = (front + 1) % SIZE; // shift front
                                     // by 1 position
      }
   }
}
```

# CYCLIC QUEUE: IMPLEMENTATION

```
void cQueue :: display() {
   int i;
   cout <<"front : " << front << "   rear : " << rear << endl;
   cout <<"Circular Queue Elements ( front to rear ) :"<< endl;
   if (front == -1) {
      cout << "Queue is empty ... " << endl;
      return;
   }
   else {
      i = front;
      do {
         cout << arr[i] << " ";
         i = (i + 1) % SIZE;
      } while(i != rear);
      cout << arr[rear];

   }
   cout << endl;
}
```

```
int main() {
   cQueue cq;
   cq.enqueue(7);
   cq.enqueue(11);
   cq.enqueue(8);
   cq.enqueue(2);
   cq.enqueue(6); // queue becomes full
   cq.display();
   cq.dequeue(); // 7 is dequeued from index 0
   cq.dequeue(); // 11 is dequeued from index 1
   cq.display();
   cq.enqueue(5); // 5 is inserted at index 0
   cq.enqueue(3); // 3 is inserted at index 1
   cq.display();
   return 0;
}
```

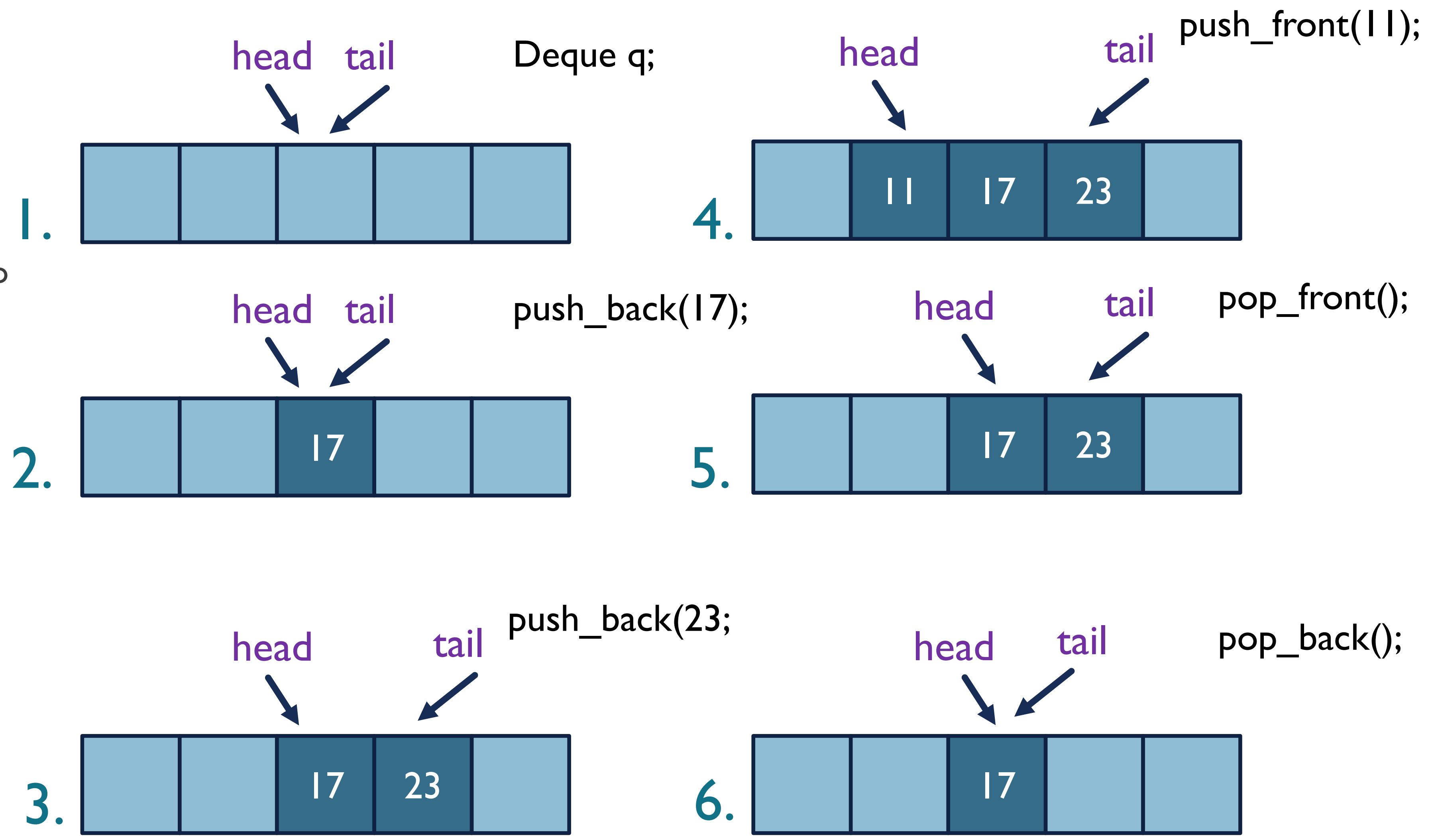
# DEQUEUE

## LINEAR DATA STRUCTURES: STACK, QUEUE, DEQUEUE

# DEQUEUE DATA STRUCTURE

## Work Principles

- Operations
  - push_back– adds element to the tail
  - push_front – adds element to the head
  - pop_back – returns element from the tail updates tail position
  - pop_front – returns element from the head and updates head position
- Advantages
  - Can be used as Stack and Queue



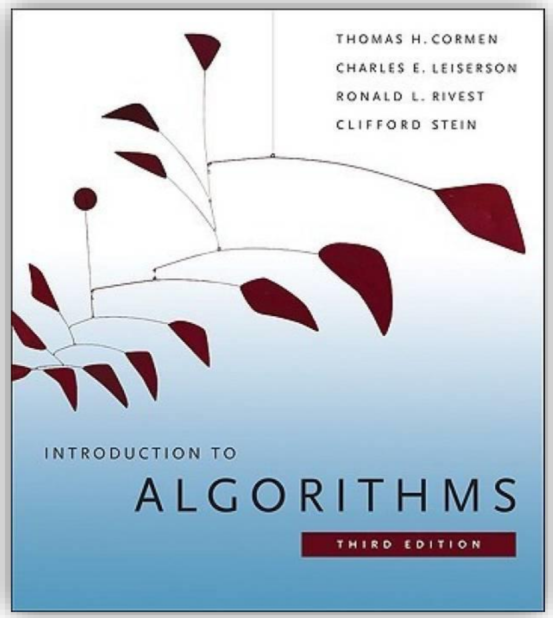
# DEQUEUE

```
class Dequeue{
    int *a;
    int head;
    int tail;
public:
    Dequeue(int n){
        a = new int[n];
        head = n/2;
        tail = n/2;
}
~Dequeue(){
    delete[] a;
}
void push_front(int x);
void push_back(int x);
int pop_front();
int pop_back();
bool is_empty();
};
```

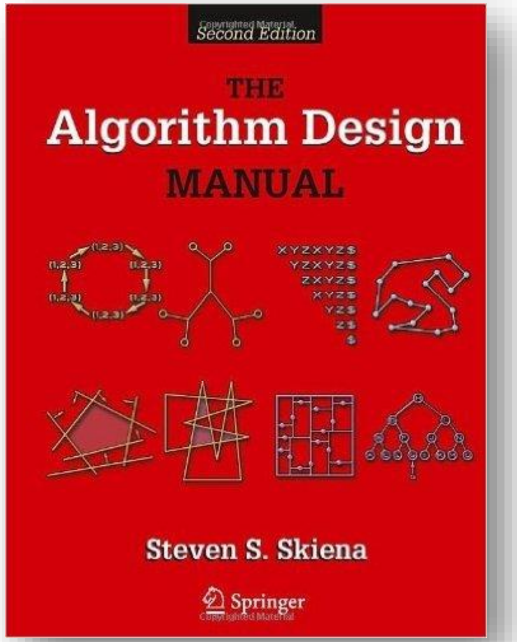
```
void Dequeue::push_front(int x){
    head--;
    a[head] = x;
}

void Dequeue::push_back(int x){
    a[tail] = x;
    tail++;
}

int Dequeue::pop_front(){
    if (head != tail) {
        head++;
    return a[head - 1];
    } else {
        cout<<"Error: pop from empty dequeue";
    }
}
```

```
int Dequeue::pop_back(){

  if (head != tail){
        tail--;
  return a[tail];
  } else {
   cout<<"Error:empty dequeue";
  }
}

bool Dequeue::is_empty(){
    return head == tail;
}
```

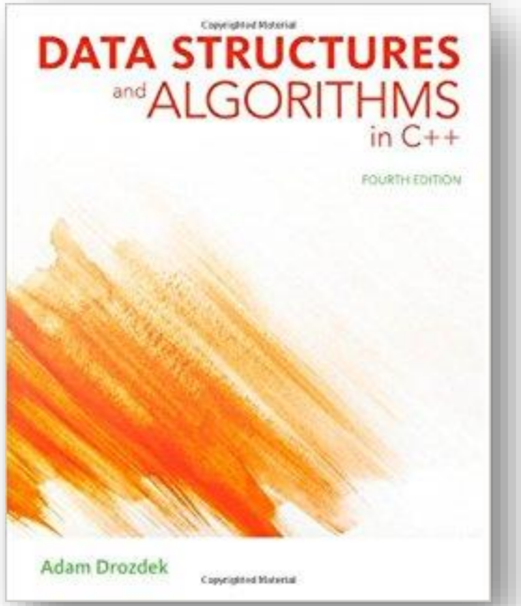
# LITERATURE




Thomas H. Cormen
Introduction to Algorithms
Chapter III: Data structures
Page 232 (Stack and queues)




Stieven Skienna
Algorithms design manual
3.2 Stack and Qeueues
Page 71




Adam Drozdek
Data structures and Algorithms in C++
Chapter 4: Stack and Queues
Page 131