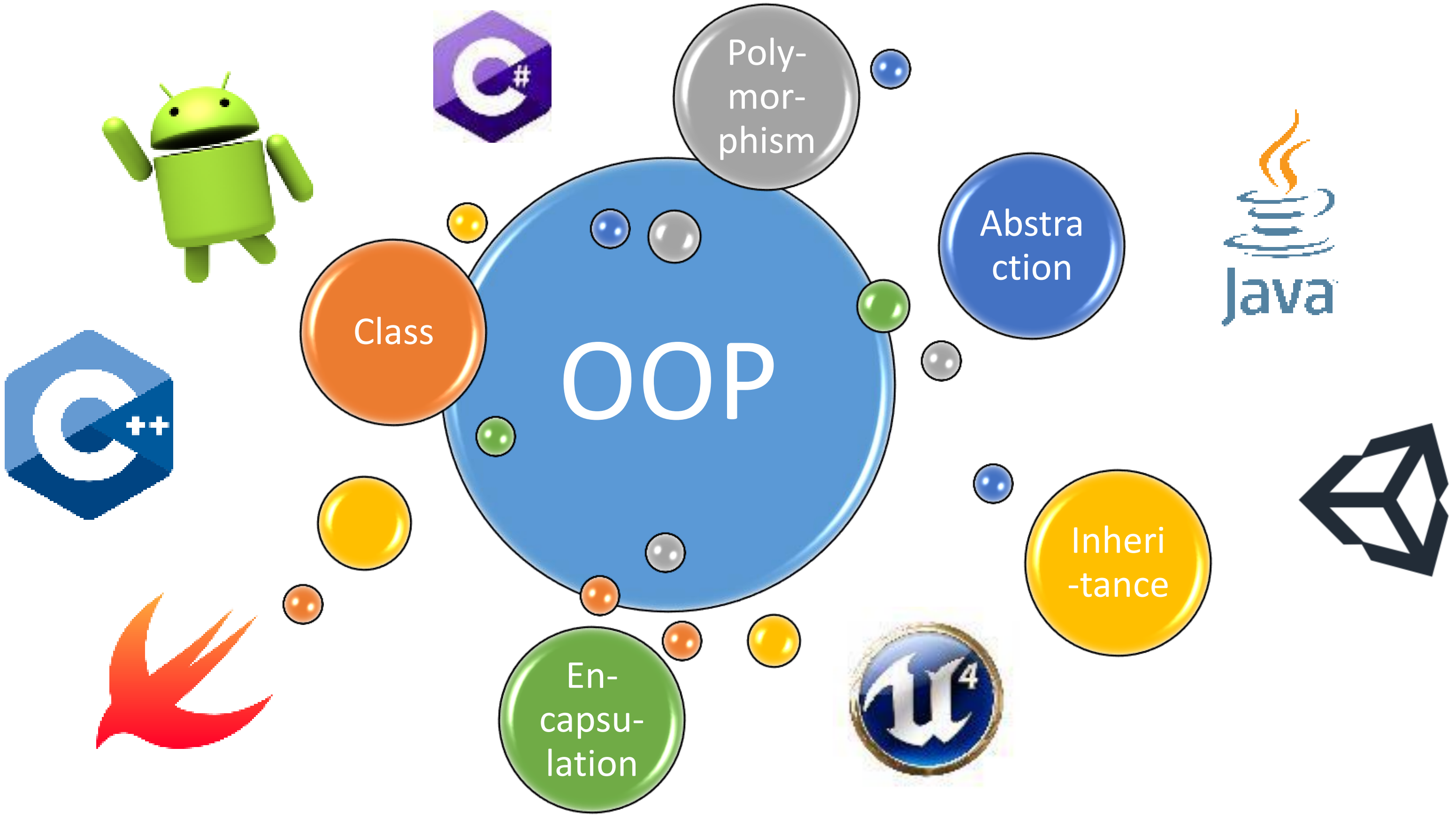


Basic Data Structures

Kuan-Ting Lai

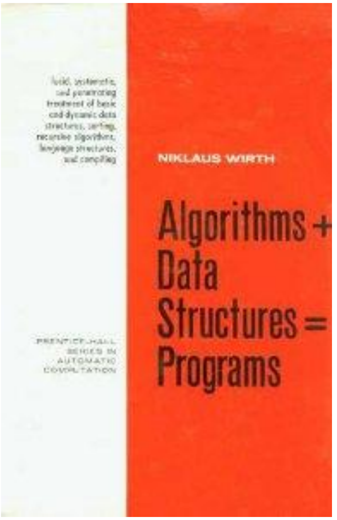
2021/5/31

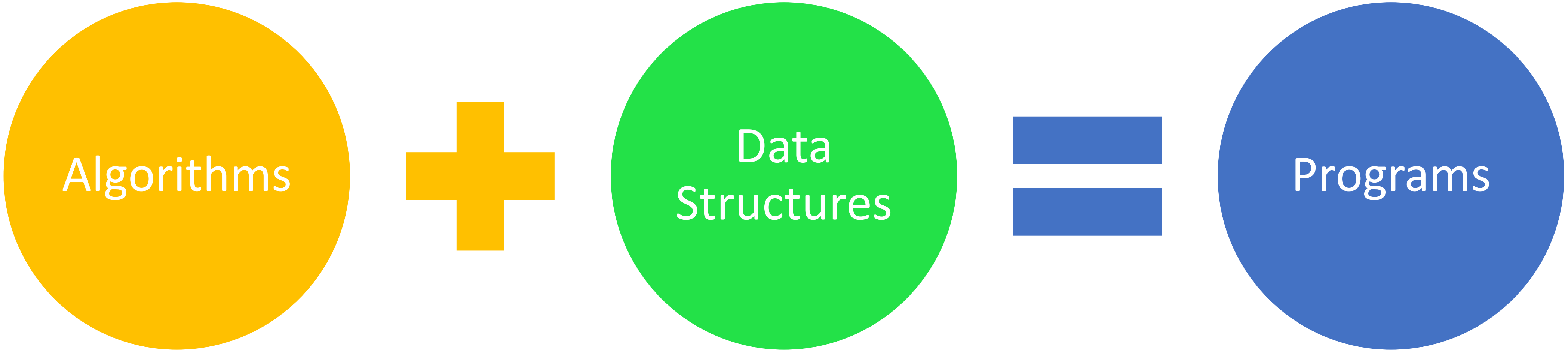
Organizing Data in Memory

Array	Linked List	Stack & Queue
Tree	Graph	Hash Table

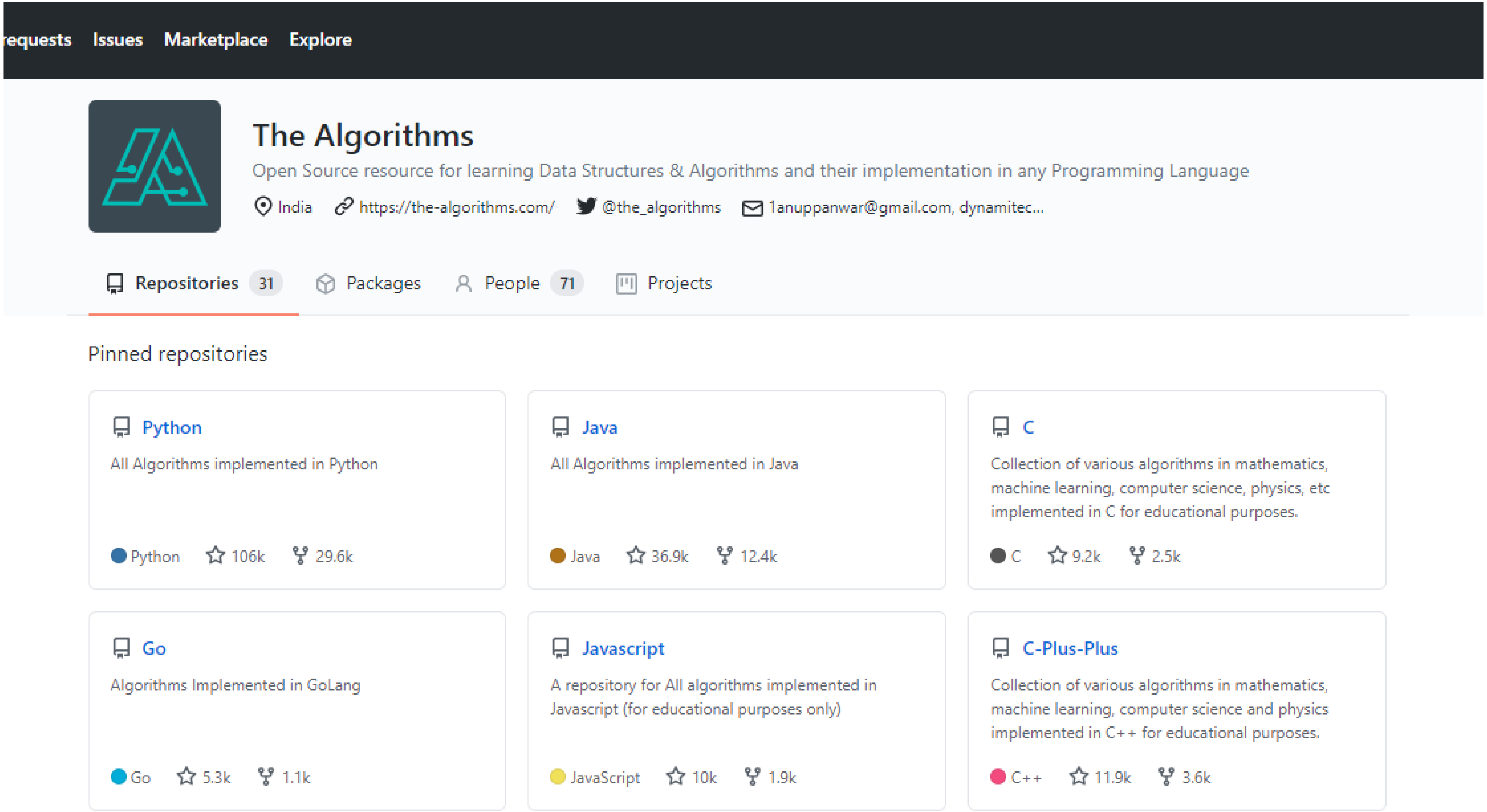
Data Structures and Algorithms

- Niklaus Wirth, 1976





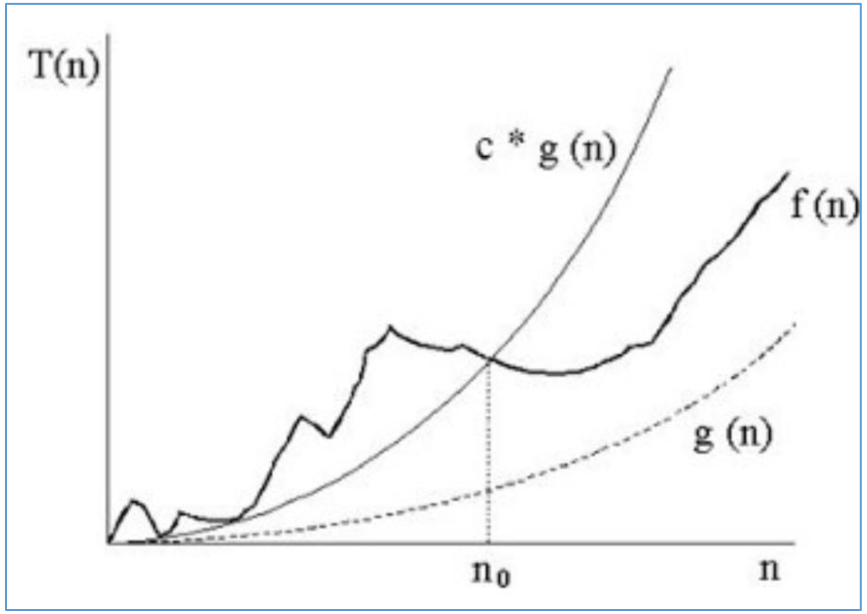
The Algorithms (github.com/TheAlgorithms)



Time Complexity (Big O Notation)

- Big-O is about finding an asymptotic upper bound
  - Ex: $O(n^2)$, n is the input size.

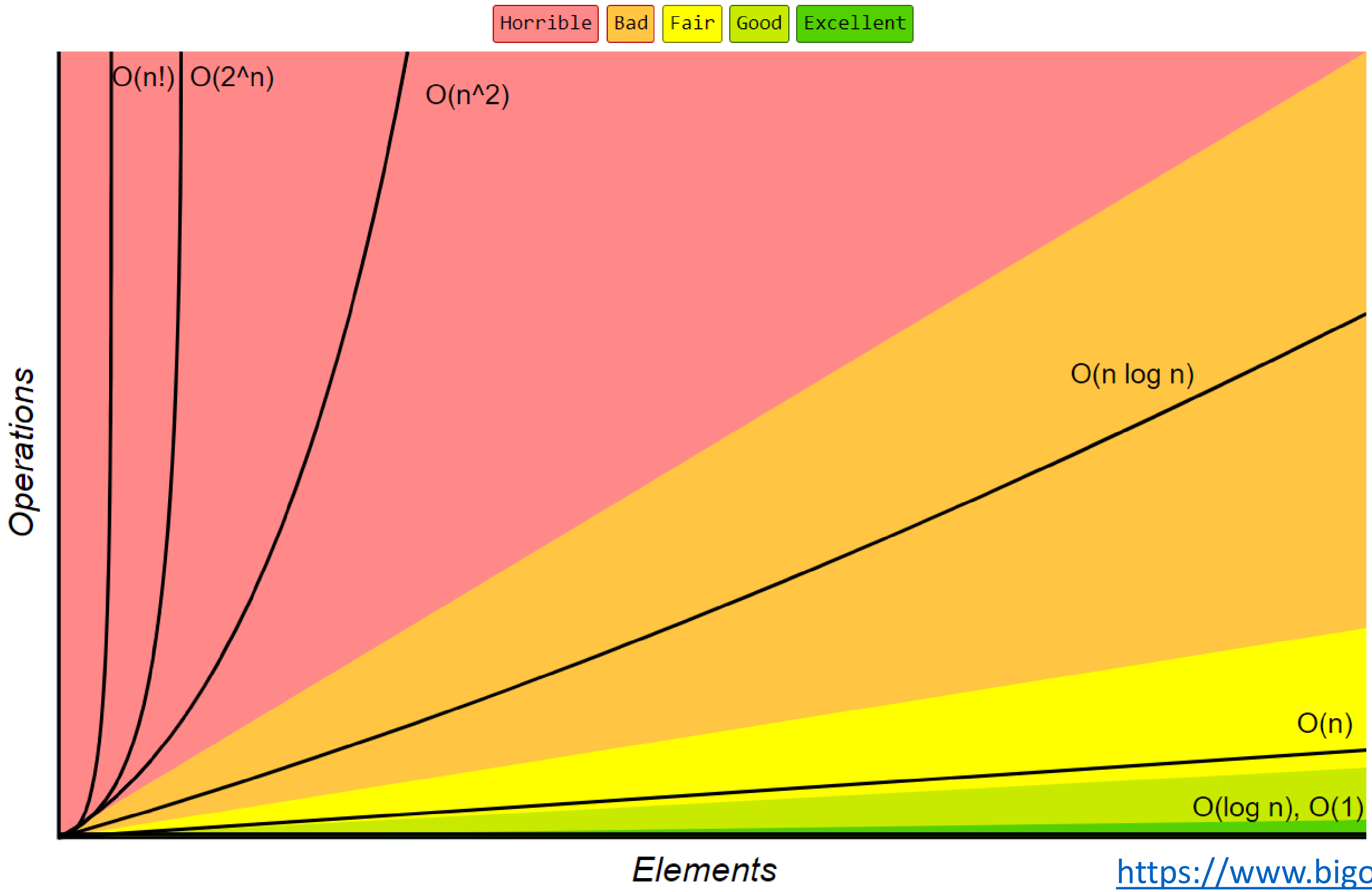
CSE 373 Slides from University of Washington



- Formal definition of Big-O:
  - $f(N) = O(g(N))$, is there exists positive constants c, N_0 such that
    $f(N) \leq c \cdot g(N)$ for all $N \geq N_0$
  - We are concerned with how f grows when N is large
    - Not concerned with small N or constant factors
  - "$f(N)$ grows no faster than $g(N)$"

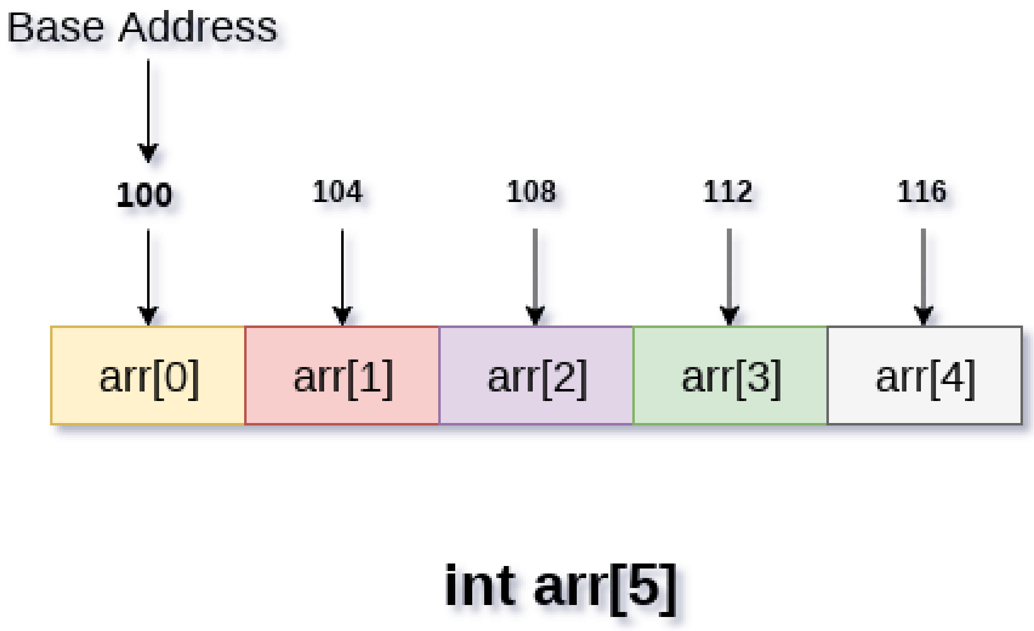
https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/

Big-O Complexity Chart



https://www.bigocheatsheet.com/

Java Array




```
public class ArrayDemo {
    public static void main(String []args) {
        ArrayDemo ad = new ArrayDemo();
        int arr[] = {1, 2, 3, 4, 5};
        int sum = ad.summation(arr);
        System.out.println(sum);
     }

    public int summation (int arr[])
    {
        int sum=0;
        for (int i = 0; i < 5; i++)
        {
            sum = sum + arr[i];
        }
        return sum;
    }
}
```

https://www.javatpoint.com/data-structure-array

String Array

- Loop through data in an array

```
String[] cars = {"Lexus", "BMW", "Benz", "Tesla"};

for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}

for (String car : cars) {
  System.out.println(car);
}
```

Multi-dimensional Array

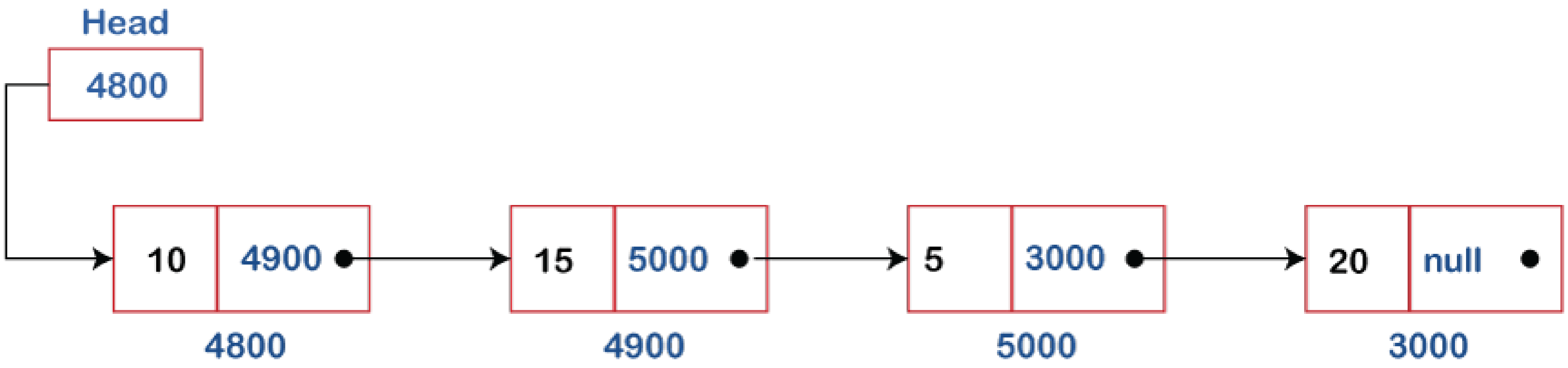
```
public class StrTest
{
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

    for (int i = 0; i < myNumbers.length; ++i) {
      for(int j = 0; j < myNumbers[i].length; ++j) {
        System.out.println(myNumbers[i][j]);
      }
    }
  }
}
```

Time Complexity of Array

Algorithm	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Linked List

- Collection of elements, but the elements are not stored continuously in memory



https://www.javatpoint.com/ds-linked-list

Linked List vs. Array

- Dynamic data structure
  - Create nodes when new data arrive
- Constant time of insertion and deletion
- Memory efficient?
  - Array requires pre-allocated memory, linked list's memory is dynamically assigned
  - Linked list needs additional node pointer for each datum, which lead to more total memory

Time Complexity of Linked List

Algorithm	Average Case	Worst Case
Access	O(n)	O(n)
Search	O(n)	O(n)
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)

SinglyLinkedList

- class **Node**

```
class Node {
  int value; // Data value
  Node next; /** Point to the next node */

  Node() {}

  Node(int value) {
    this(value, null);
  }

  Node(int value, Node next) {
    this.value = value;
    this.next = next;
  }
}
```

https://github.com/TheAlgorithms/Java/blob/master/DataStructures/Lists/SinglyLinkedList.java

SinglyLinkedList Functions

- insertHead(int x)
- Insert(int data)
- insertNth(int data, int position)
- deleteHead()
- delete()
- deleteNth(int position)

insertNth(…)

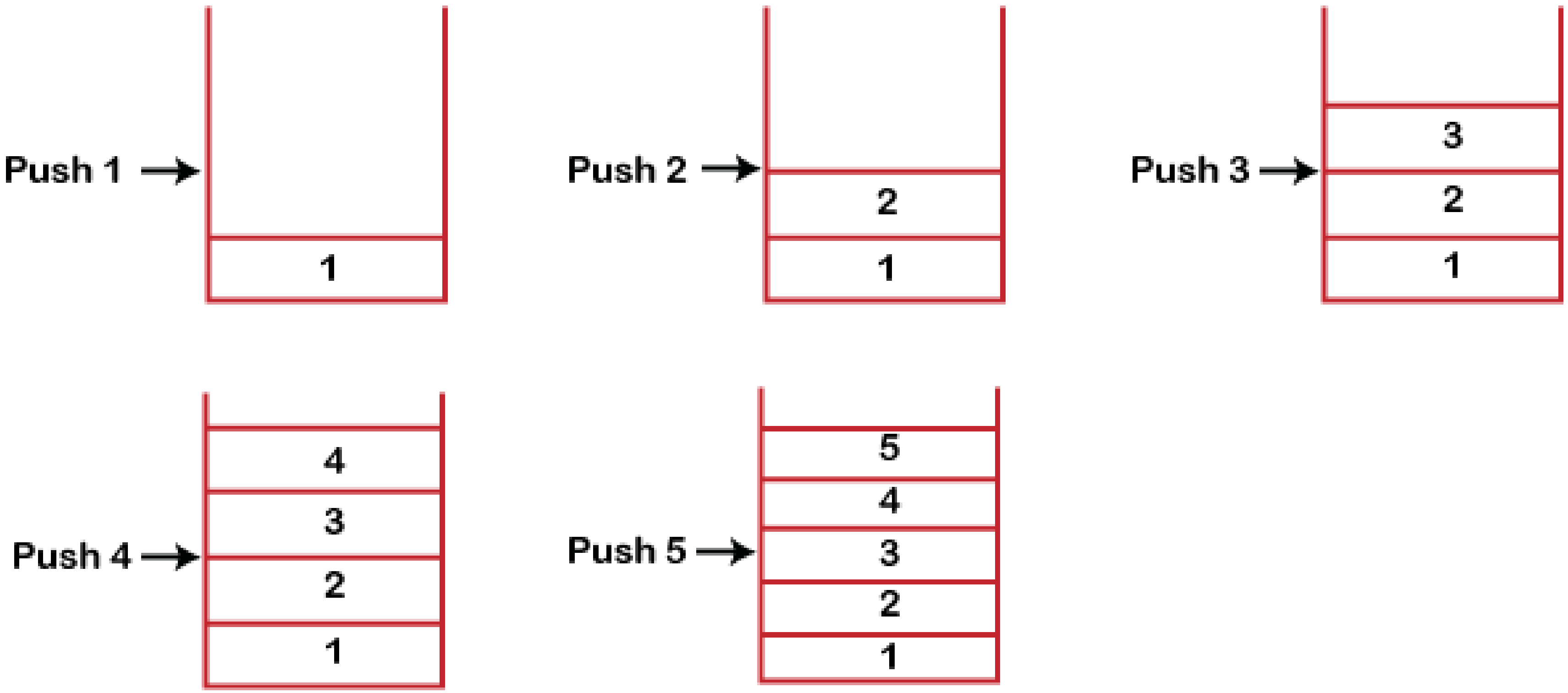
```
public void insertNth(int data, int position) {
    checkBounds(position, 0, size);
    Node newNode = new Node(data);
    if (head == null) {
      /* the list is empty */
      head = newNode;
      size++;
      return;
    } else if (position == 0) {
      /* insert at the head of the list */
      newNode.next = head;
      head = newNode;
      size++;
      return;
    }
    Node cur = head;
    for (int i = 0; i < position - 1; ++i) {
      cur = cur.next;
    }
    newNode.next = cur.next;
    cur.next = newNode;
    size++;
  }
```

deleteNth(…)

```java
public void deleteNth(int position) {
    checkBounds(position, 0, size - 1);
    if (position == 0) {
      Node destroy = head;
      head = head.next;
      destroy = null; /* clear to let GC do its work */
      size--;
      return;
    }
    Node cur = head;
    for (int i = 0; i < position - 1; ++i) {
      cur = cur.next;
    }

    Node destroy = cur.next;
    cur.next = cur.next.next;
    destroy = null; // clear to let GC do its work

    size--;
  }
```

Stack

- Last-In First-Out (LIFO)

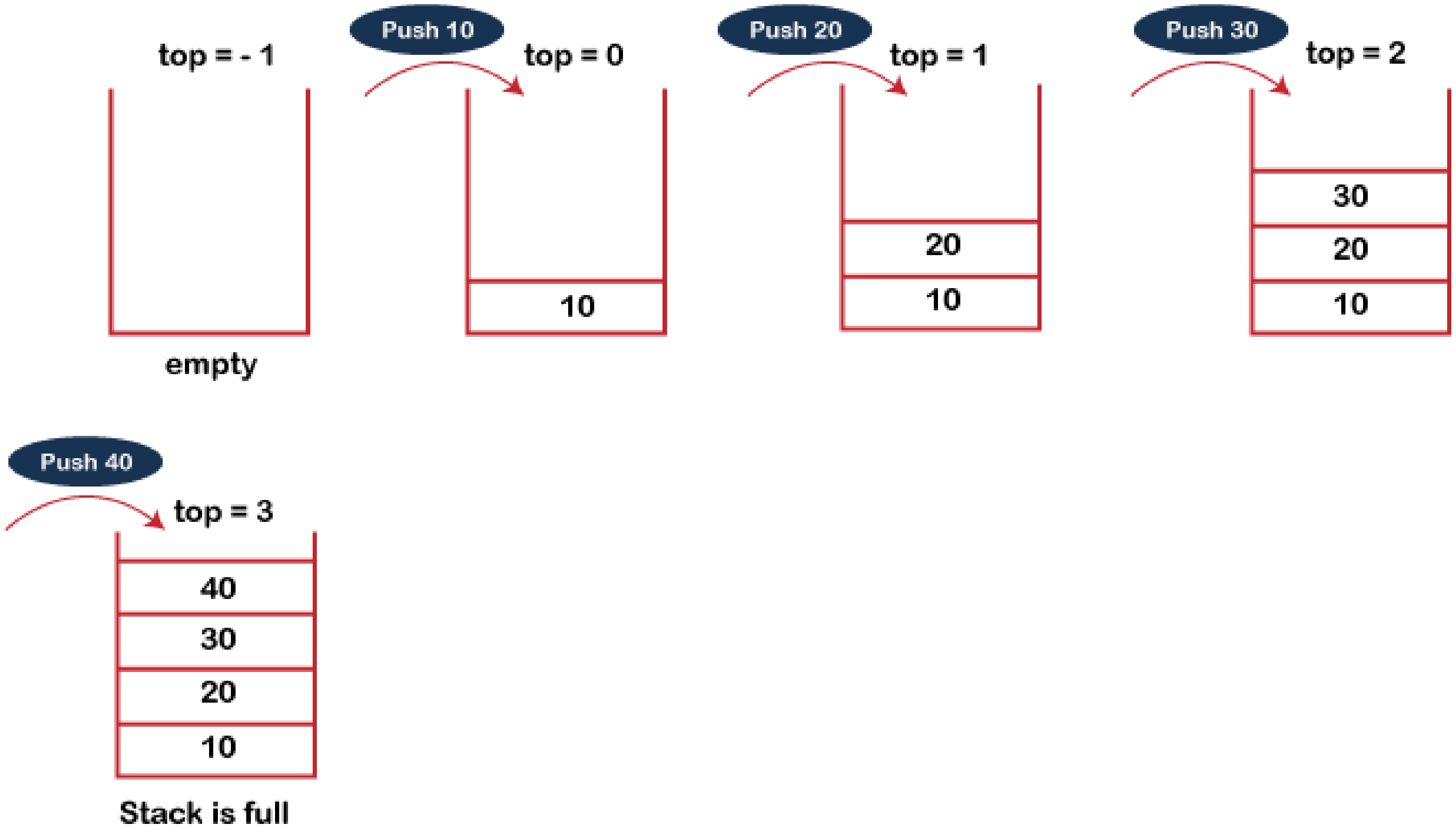


https://www.javatpoint.com/data-structure-stack

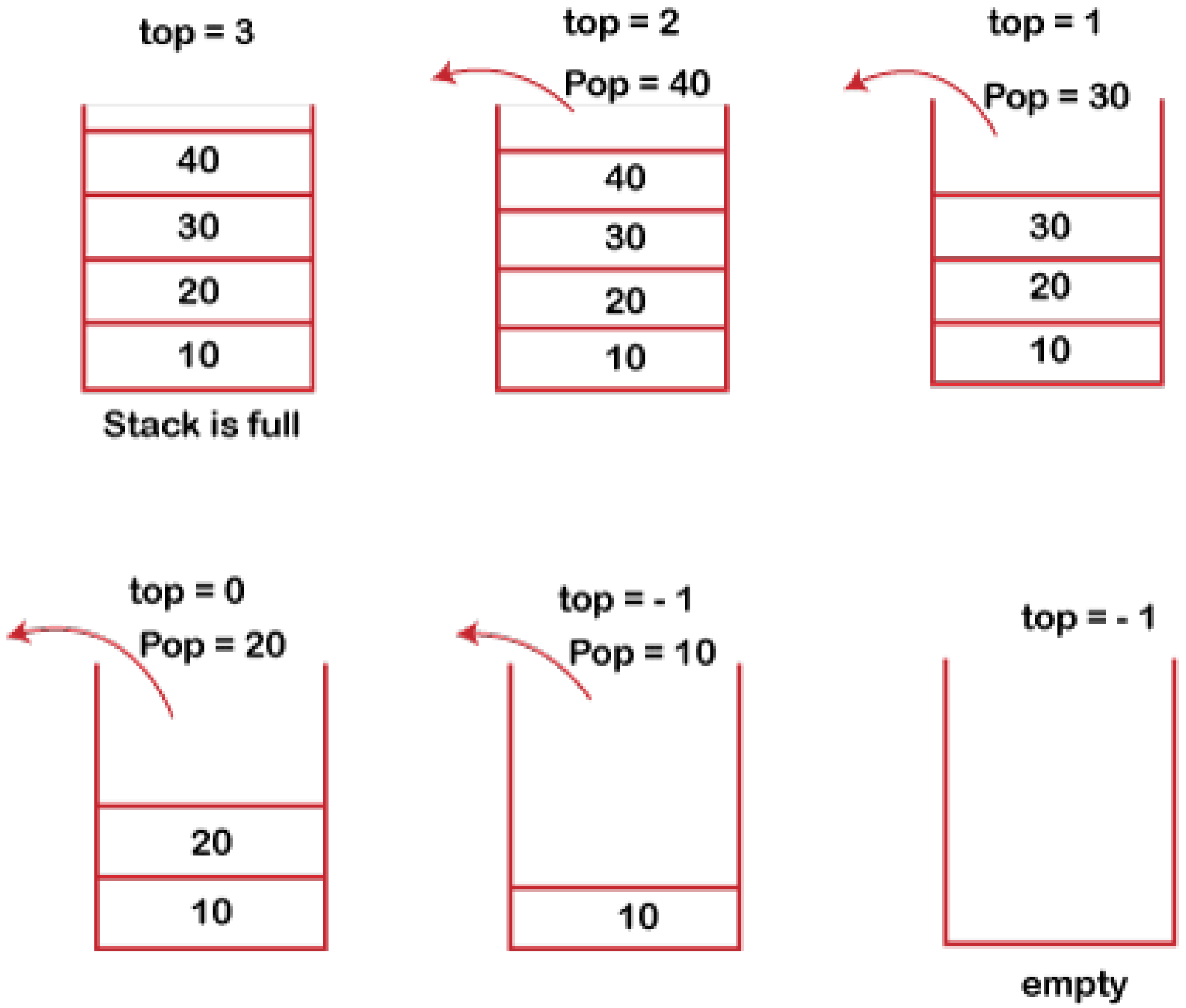
Standard Stack Operations

- **push():** Insert an element in a stack
- **pop():** Delete an element from the stack
- **isEmpty():** Check whether the stack is empty or not.
- **isFull():** Check whether the stack is full or not.
- **peek():** Return the element at the given position.
- **count():** Return the total number of elements in a stack.

Push Operation



Pop Operation



StackArray (3-1)

- Use array to save data
- Auto-resize if necessary

```
public class StackArray {
  private static final int DEFAULT_CAPACITY = 10;
  private int maxSize;
  private int[] stackArray;
  private int top;

  public StackArray() { this(DEFAULT_CAPACITY); }
  public StackArray(int size) {
    maxSize = size;
    stackArray = new int[maxSize];
    top = -1;
  }
  public void push(int value) {
    if (!isFull()) { // Check for a full stack
      top++;
      stackArray[top] = value;
    } else {
      resize(maxSize * 2);
      push(value); // don't forget to push after resizing
    }
  }
 ……
 ……
}
```

https://github.com/TheAlgorithms/Java/blob/master/DataStructures/Stacks/StackArray.java

StackArray (3-2)

- pop()
  - Pop a value, downsize the array if total data is less than maxSize / 4

```
public class StackArray {
  ……
  ……
  public int pop() {
    if (!isEmpty()) { // Check for an empty stack
      return stackArray[top--];
    }
  }

  public int isEmpty() {
      return (top == -1);
  }

  public int peek() {
    if (!isEmpty()) { // Check for an empty stack
      return stackArray[top];
    } else {
      System.out.println("The stack is empty, can't peek");
      return -1;
    }
  }
  ……
}
```

StackArray (3-3)

```
public class StackArray {
 ……
  public boolean isEmpty() {
    return (top == -1);
  }
  public boolean isFull() {
    return (top + 1 == maxSize);
  }
  public void makeEmpty() {
    top = -1;
  }
  public int size() {
    return top + 1;
  }
  private void resize(int newSize) {
    int[] transferArray = new int[newSize];

    for (int i = 0; i < stackArray.length; i++) {
      transferArray[i] = stackArray[i];
    }
    // This reference change might be nice in here
    stackArray = transferArray;
    maxSize = newSize;
  }
 ……
}
```

Test StackArray

- Create a stack with max size 4
- Push 4 data into the array

```
public class StackArray
{
    public static void main(String[] args) {
      // Declare a stack of maximum size 4
      StackArray myStackArray = new StackArray(4);

      assert myStackArray.isEmpty();
      assert !myStackArray.isFull();

      // Populate the stack
      myStackArray.push(5);
      myStackArray.push(8);
      myStackArray.push(2);
      myStackArray.push(9);

      assert !myStackArray.isEmpty();
      assert myStackArray.isFull();
      assert myStackArray.peek() == 9;
      assert myStackArray.pop() == 9;
      assert myStackArray.peek() == 2;
      assert myStackArray.size() == 3;
    }
    ...
}
```

PEMDAS (先乘除後加減)

- Parenthesis, Exponents, Multiplication, Division, Addition, Subtraction (PEMDAS)

Operators	Symbols
Parenthesis	( ), {}, [ ]
Exponents	^
Multiplication and Division	*, /
Addition and Subtraction	+ , -

https://www.javatpoint.com/convert-infix-to-postfix-notation

Infix and Postfix Expression

- Infix: 3*4 + 2*5
- Postfix: 3 4 * 2 5 * +
- Evaluate Postfix:

Input	Stack	
3 4 * 2 5 * +	empty	Push 3
4 * 2 5 * +	3	Push 4
*2 5 * +	4 3	Pop 3 and 4 from the stack and perform 3*4 = 12. Push 12 into the stack.
2 5 * +	12	Push 2
5 * +	2 12	Push 5
*+	5 2 12	Pop 5 and 2 from the stack and perform 5*2 = 10. Push 10 into the stack.
+	10 12	Pop 10 and 12 from the stack and perform 10+12 = 22. Push 22 into the stack.

Infix to Postfix using a Stack

- Infix expression
  - K + L - M*N + (O^P)

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^

Infix to Postfix

```java
public static String infix2PostFix(String infixExpression) throws Exception {
    if (!BalancedBrackets.isBalanced(infixExpression)) {throw new Exception("invalid expression");}
    StringBuilder output = new StringBuilder();
    Stack<Character> stack = new Stack<>();
    for (char element : infixExpression.toCharArray()) {
      if (Character.isLetterOrDigit(element)) {
        output.append(element);
      } else if (element == '(') {
        stack.push(element);
      } else if (element == ')') {
        while (!stack.isEmpty() && stack.peek() != '(') {
          output.append(stack.pop());
        }
        stack.pop();
      } else {
        while (!stack.isEmpty() && precedence(element) <= precedence(stack.peek())) {
          output.append(stack.pop());
        }
        stack.push(element);
      }
    }
    while (!stack.isEmpty()) {
      output.append(stack.pop());
    }
    return output.toString();
  }
```

https://github.com/TheAlgorithms/Java/blob/master/DataStructures/Stacks/InfixToPostfix.java

Test InfixToPosfix

```
import java.util.Stack;
public class InfixToPostfix {
  public static void main(String[] args) throws Exception {
    assert "32+".equals(infix2PostFix("3+2"));
    assert "123++".equals(infix2PostFix("1+(2+3)"));
    assert "34+5*6-".equals(infix2PostFix("(3+4)*5-6"));
  }
  public static String infix2PostFix(String infixExpression) throws Exception {......}
  private static int precedence(char operator) {
    switch (operator) {
      case '+':
      case '-':
        return 0;
      case '*':
      case '/':
        return 1;
      case '^':
        return 2;
      default:
        return -1;
    }
  }
}
```

Queue

- First In First Out list (FIFO)

Data Structure	Time Complexity								Space
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

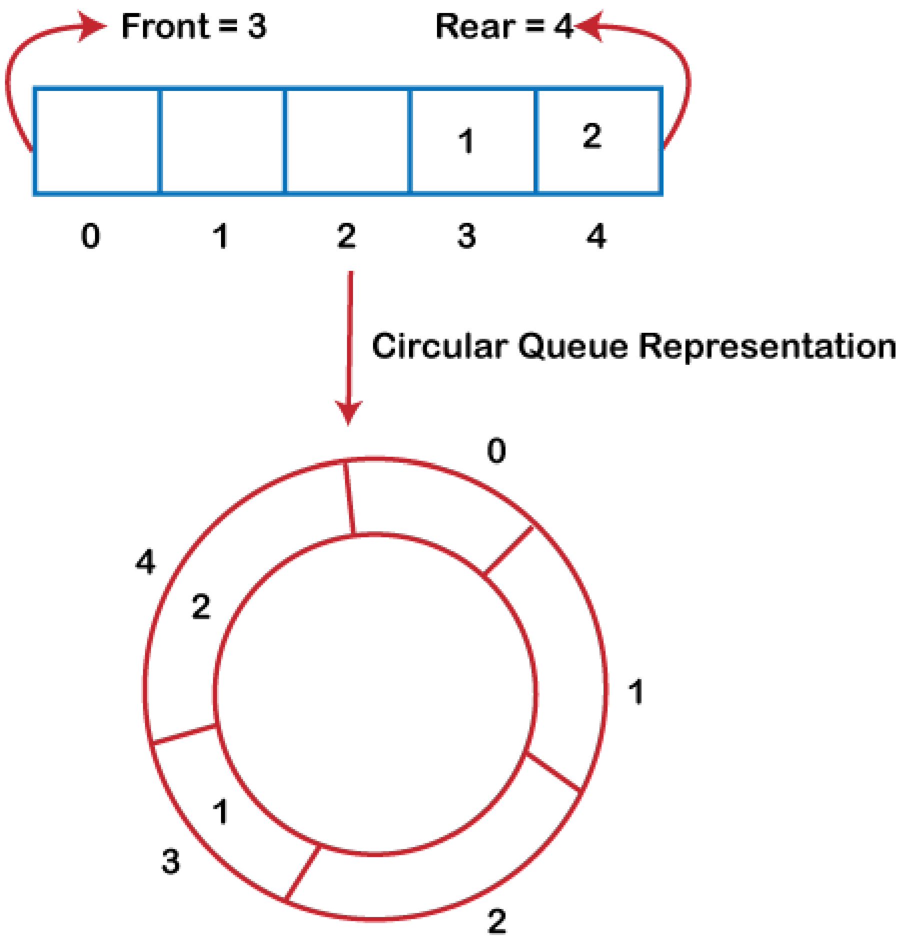
https://www.javatpoint.com/data-structure-queue

Standard Queue Operations

- **Enqueue:** Insert the element at the rear end of the queue.
- **Dequeue:** Delete and return data from the front-end
- **Peek:** Return the top element in the queue but does not delete it.
- **Queue overflow (isFull):** Check if the queue is full.
- **Queue underflow (isEmpty):** Check if the Queue is empty

https://www.javatpoint.com/ds-types-of-queues

Circular Queue



Queue (2-1)

- Implement circular queue

```
rear = (rear + 1) % maxSize
```

```
class Queue {
  private static final int DEFAULT_CAPACITY = 10;
  private int maxSize;
  private int[] queueArray;
  private int front;
  private int rear;
  private int nItems;
  public Queue() {
    this(DEFAULT_CAPACITY);
  }
  public Queue(int size) {
    maxSize = size;
    queueArray = new int[size];
    front = 0;
    rear = -1;
    nItems = 0;
  }
  public boolean insert(int x) {
    if (isFull()) return false;
    rear = (rear + 1) % maxSize;
    queueArray[rear] = x;
    nItems++;
    return true;
  }……}
```

Queue (2-2)

```
class Queue {
……
  public int remove() {
    if (isEmpty()) {
      return -1;
    }
    int temp = queueArray[front];
    front = (front + 1) % maxSize;
    nItems--;
    return temp;
  }
  public int peekFront() {
    return queueArray[front];
  }
  public int peekRear() {
    return queueArray[rear];
  }
  public boolean isEmpty() { return nItems == 0;}
  public boolean isFull() {return nItems == maxSize;}
  public int getSize() {
    return nItems;
  }}
```

Test Queue

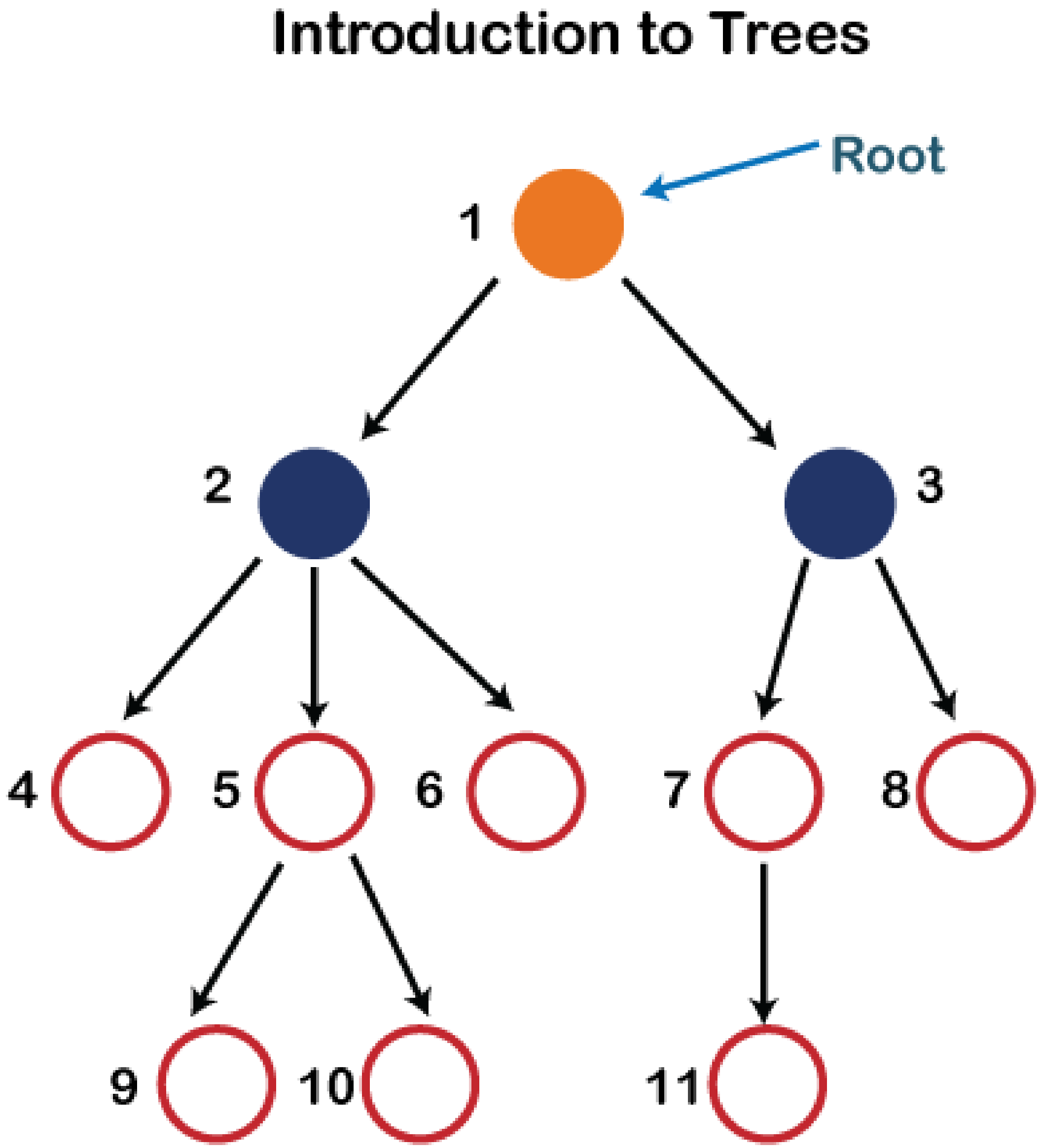
```java
public class Queues {
    public static void main(String[] args) {
    Queue myQueue = new Queue(4);
    myQueue.insert(10);
    myQueue.insert(2);
    myQueue.insert(5);
    myQueue.insert(3);
    // [10(front), 2, 5, 3(rear)]

    System.out.println(myQueue.isFull()); // Will print true

    myQueue.remove();
    // [10, 2(front), 5, 3(rear)]

    myQueue.insert(7);
    // [7(rear), 2(front), 5, 3]

    System.out.println(myQueue.peekFront()); // Will print 2
    System.out.println(myQueue.peekRear()); // Will print 7
    System.out.println(myQueue.toString()); // Will print [2, 5, 3, 7]
  }
}
```

Time Complexity of Stack and Queue

Algorithm	Average Case	Worst Case
Access	O(n)	O(n)
Search	O(n)	O(n)
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)

Tree

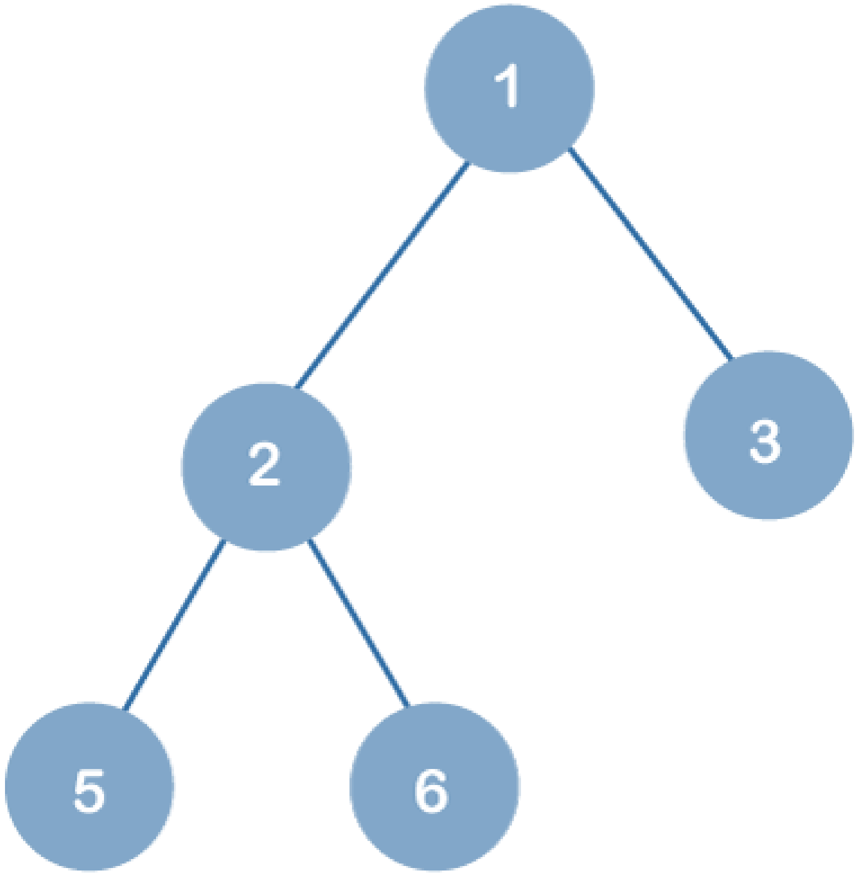
- **Root:** the topmost node.
- **Child node:** a descendant of any node
- **Parent:** the node contains any sub-node
- **Sibling:** nodes that have the same parent
- **Leaf Node:** the nodes don't have any child node, a.k.a. external nodes. (4, 9, 10, 6, 11, 8)
- **Internal nodes:** a node has at least one child node.
- **Ancestor nodes:-** any predecessor node on a path from the root to the given node.
  - nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** the immediate successors of the given node.
  - 10 is the descendant of node 5.



https://www.javatpoint.com/tree

Binary Tree

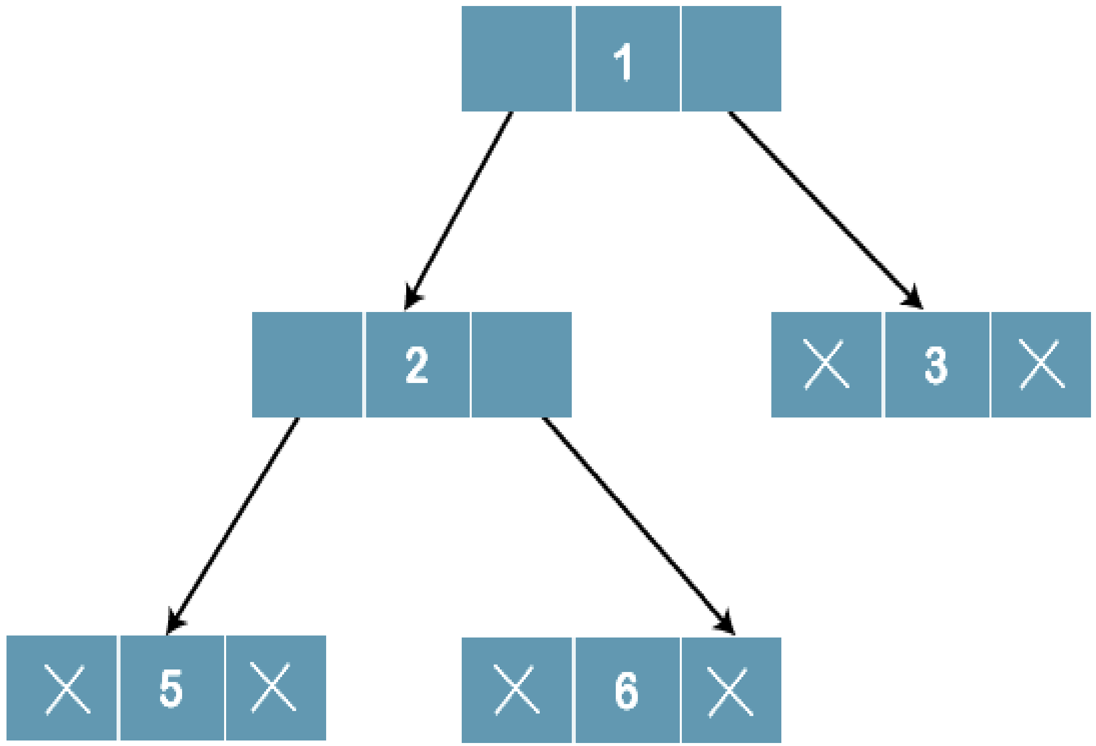
- At each level of i, the max number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3.



https://www.javatpoint.com/binary-tree

Binary Tree Implementation

```
class Node {
    public int data;
    public Node left;
    public Node right;
    public Node parent;

    public Node(int value) {
      data = value;
      left = null;
      right = null;
      parent = null;
    }
  }
```



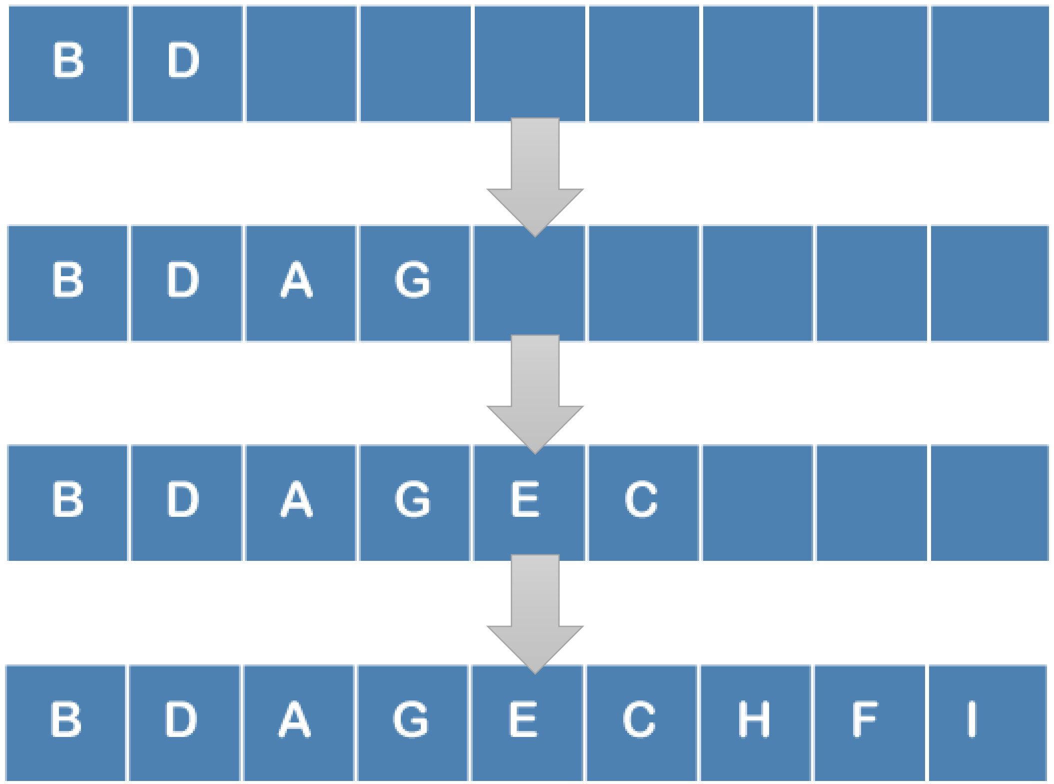
https://github.com/TheAlgorithms/Java/blob/master/DataStructures/Trees/BinaryTree.java

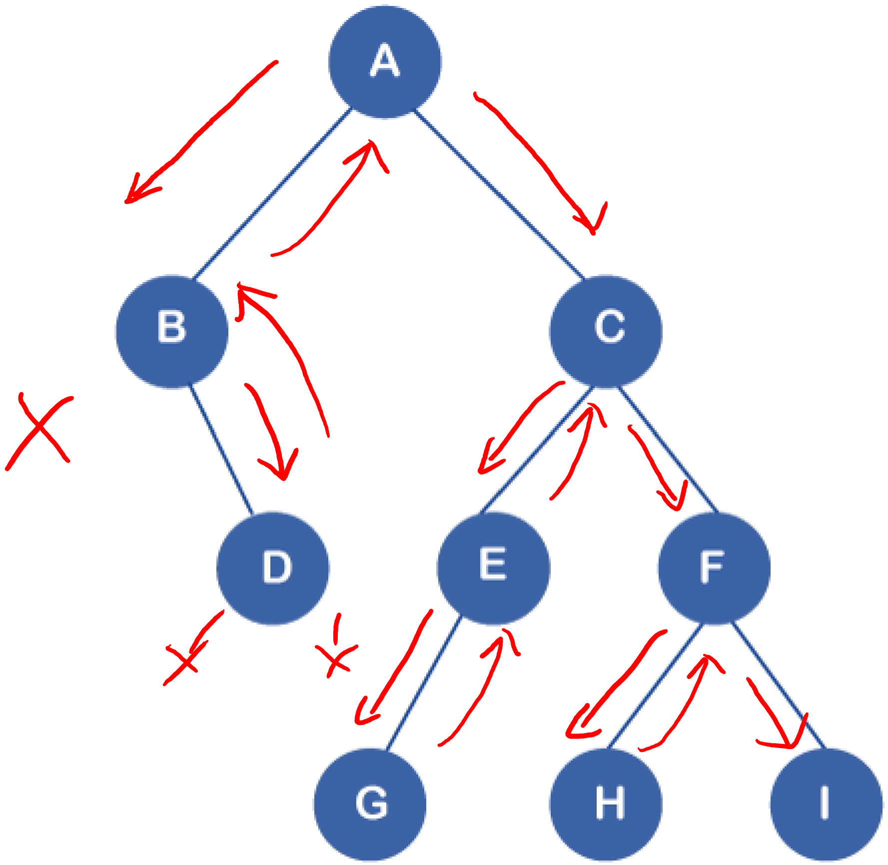
Tree Traversal

- In-order traversal
- Pre-order traversal
- Post-order traversal

In-order Traversal

- Left -> Root -> Right

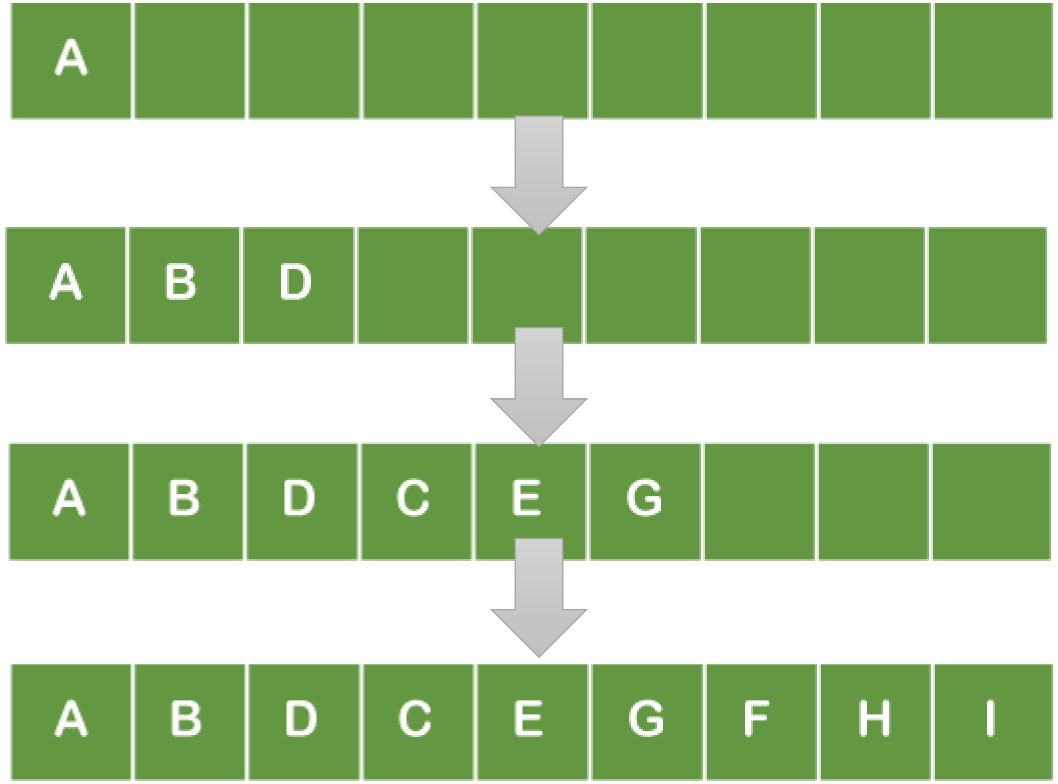


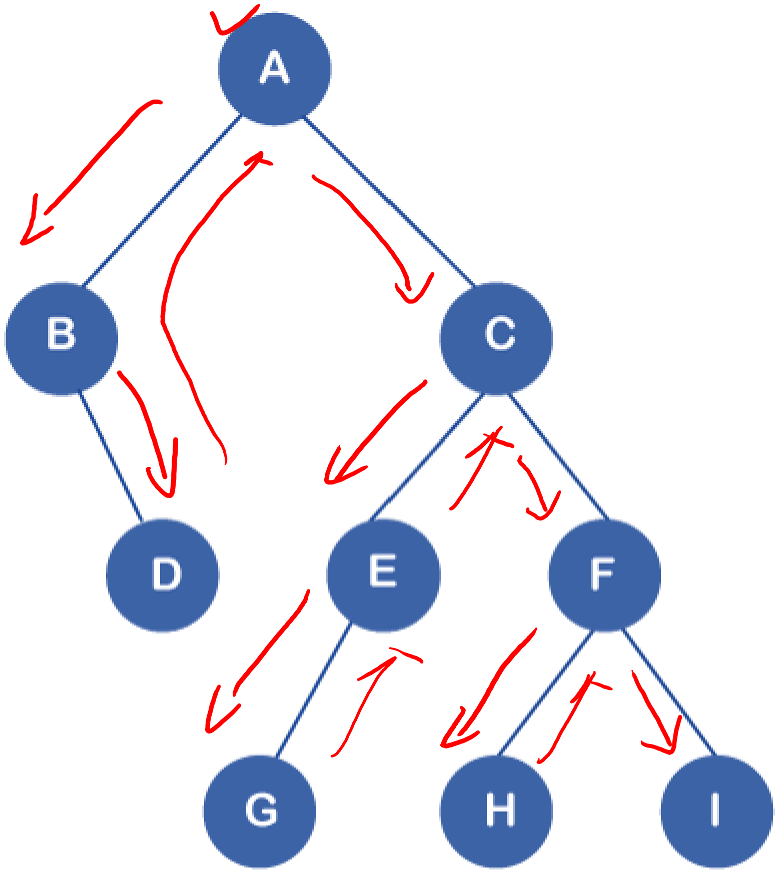


https://www.javatpoint.com/tree-traversal

Pre-order Traversal

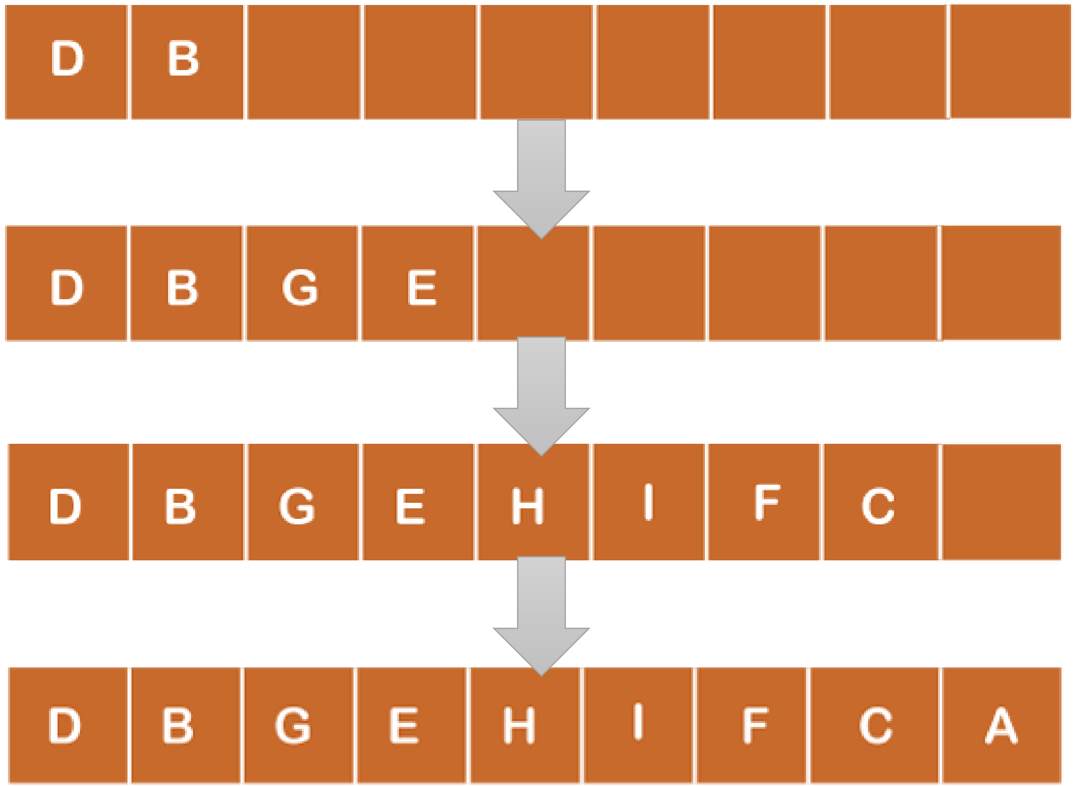
- Root -> Left -> Right

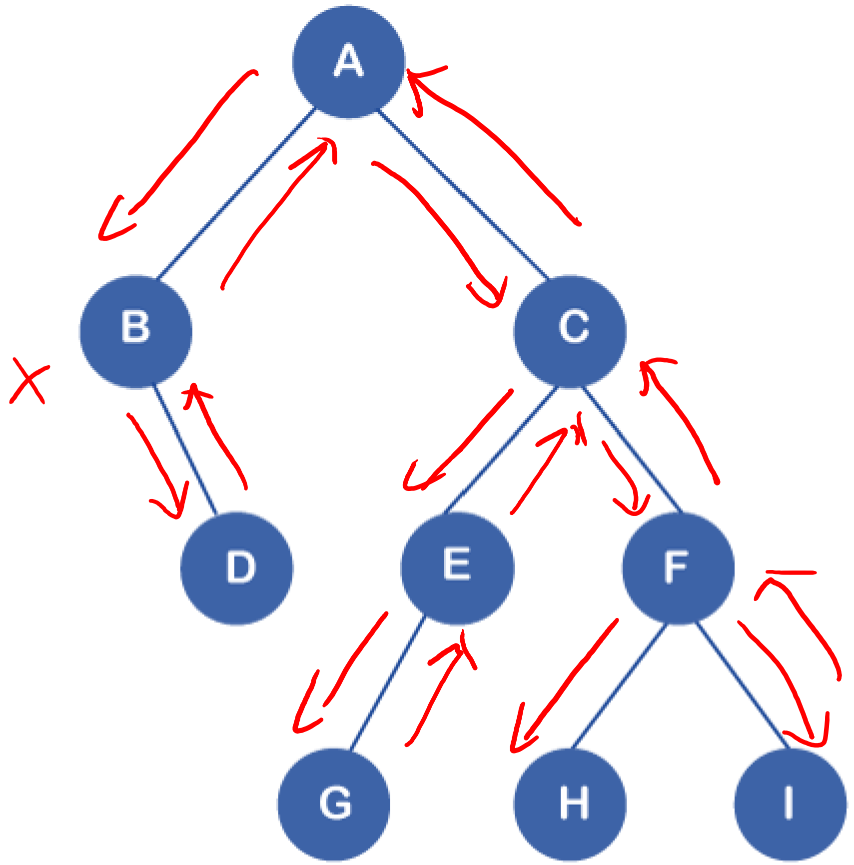




Post-order Traversal

- Left -> Right -> Root



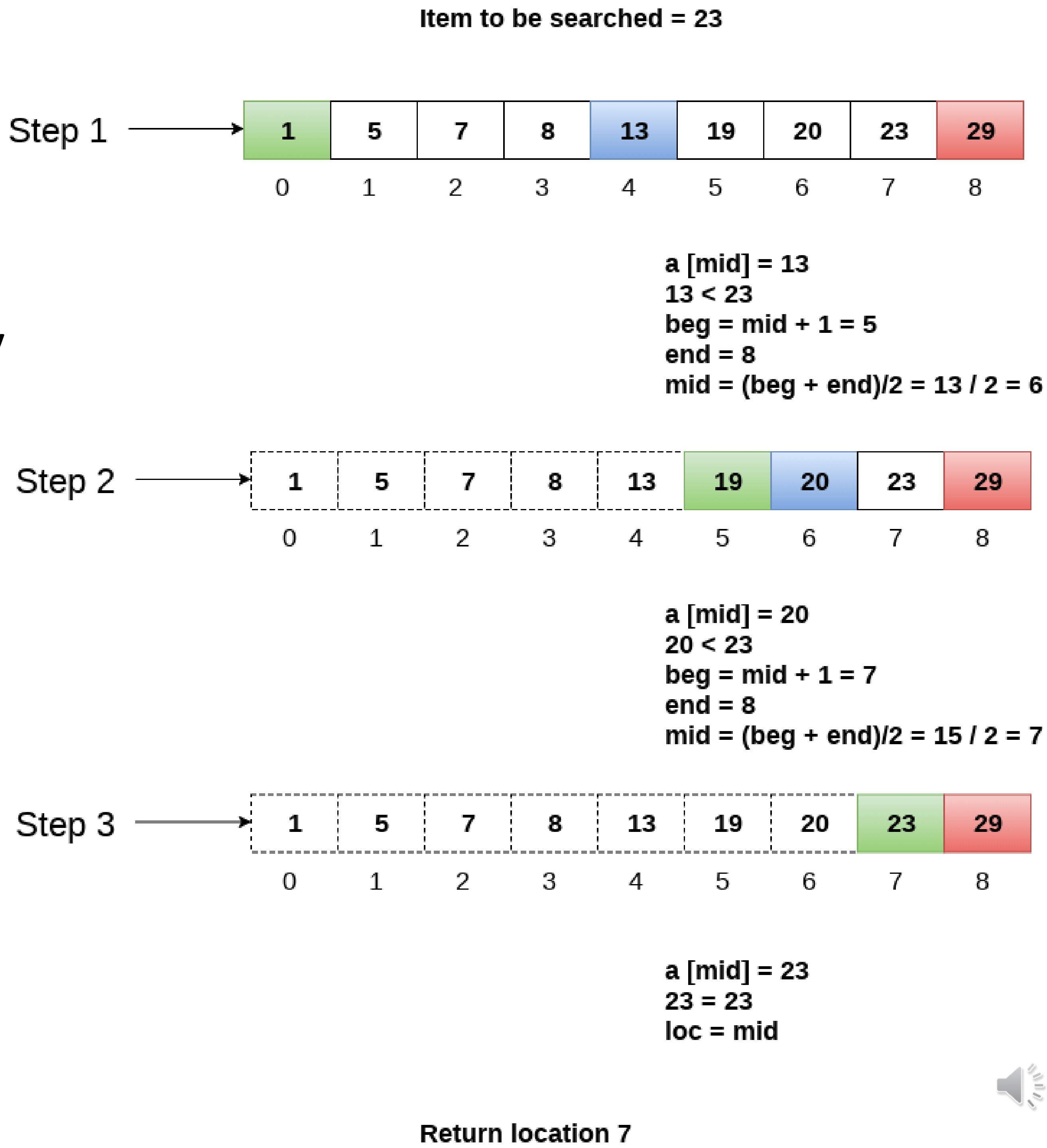


Recursive Traversal

```
public class BinaryTree {
  private Node root;
  public BinaryTree() { root = null;}
  public void inOrder(Node localRoot) {
    if (localRoot != null) {
      inOrder(localRoot.left);
      System.out.print(localRoot.data + " ");
      inOrder(localRoot.right);
    }
  }
  public void preOrder(Node localRoot) {
    if (localRoot != null) {
      System.out.print(localRoot.data + " ");
      preOrder(localRoot.left);
      preOrder(localRoot.right);
    }
  }
  public void postOrder(Node localRoot) {
    if (localRoot != null) {
      postOrder(localRoot.left);
      postOrder(localRoot.right);
      System.out.print(localRoot.data + " ");
    }
  }}
```

Binary Search

- Search an item in a **SORTED** array
- O(logN)



https://www.javatpoint.com/binary-search

Binary Search in Java

```
public static int binarySearch(int[] data, int beg, int end, int item) {
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        if(data[mid] == item)
        {
            return mid;
        }
        else if(data[mid] < item)
        {
            return binarySearch(data, mid+1, end, item);
        }
        else
        {
            return binarySearch(data, beg, mid-1, item);
        }
    }
    return -1;
}
```

https://www.javatpoint.com/binary-search

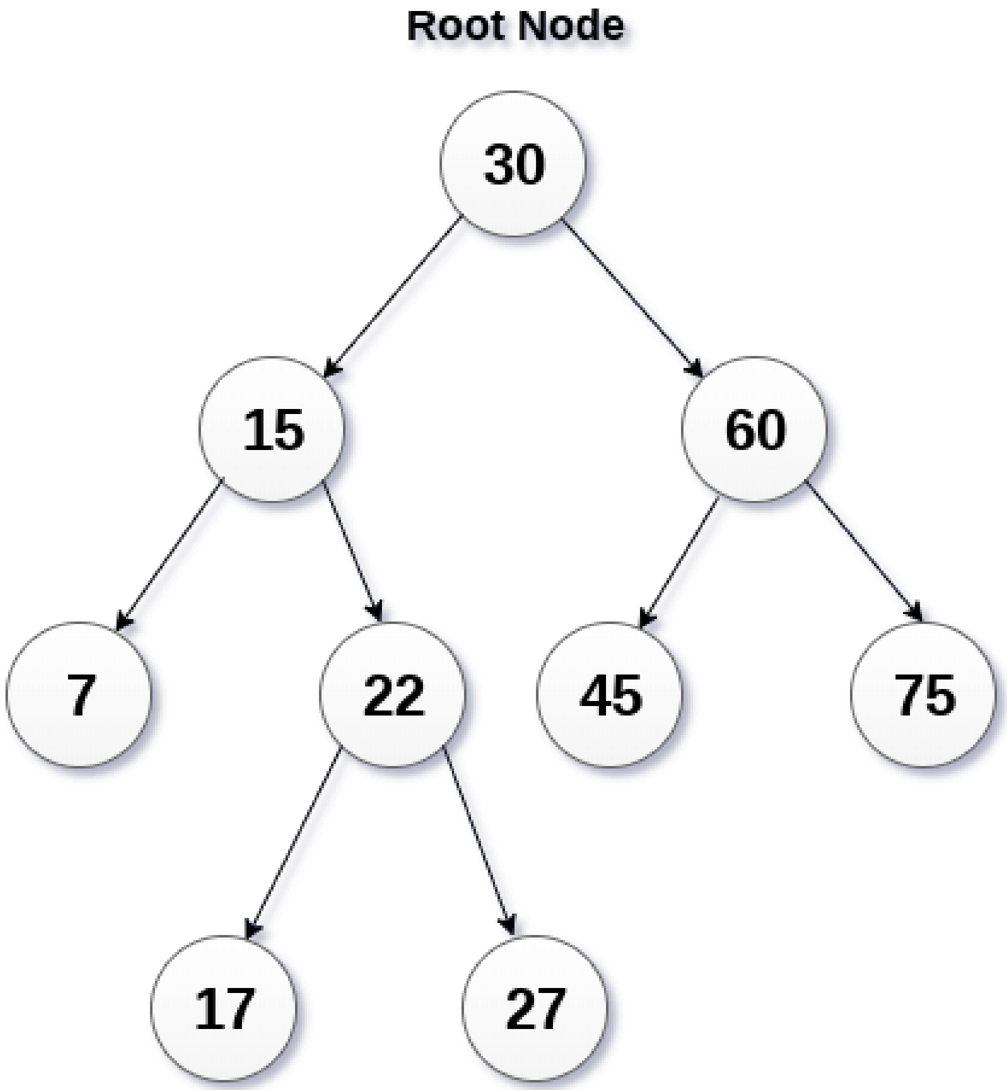
Binary Search Test

```
import java.util.*;
public class BinarySearch
{
  public static void main(String[] args) {
    int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
    int item, location = -1;
    System.out.println("Enter the item which you want to search");
    Scanner sc = new Scanner(System.in);
    item = sc.nextInt();
    location = binarySearch(arr, 0, 9, item);
    if(location != -1)
        System.out.println("the location of the item is "+location);
    else
        System.out.println("Item not found");
    }

  public static int binarySearch(int[] a, int beg, int end, int item) {…}
}
```

https://www.javatpoint.com/binary-search

Binary Search Tree (BST)

- The value of all the nodes in the left sub-tree is less than the value of the root
- Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root



Binary Search Tree

https://www.javatpoint.com/binary-search-tree

Insert a Node into BST

```
public class BSTRecursive {
  private Node root;
  BSTRecursive() {
    root = null;
  }
  private Node insert(Node node, int data) {
    if (node == null) {
      node = new Node(data);
    } else if (node.data > data) {
      node.left = insert(node.left, data);
    } else if (node.data < data) {
      node.right = insert(node.right, data);
    }
    return node;
  }
  public void add(int data) {
    this.root = insert(this.root, data);
  }
  …
  …
```

https://github.com/TheAlgorithms/Java/blob/master/DataStructures/Trees/BSTRecursive.java

Search a Node in BST

```
public class BSTRecursive {
  private boolean search(Node node, int data) {
    if (node == null) {
      return false;
    } else if (node.data == data) {
      return true;
    } else if (node.data > data) {
      return search(node.left, data);
    } else {
      return search(node.right, data);
    }
  }
  public boolean find(int data) {
    if (search(this.root, data)) {
      System.out.println(data + " is present in given BST.");
      return true;
    }
    System.out.println(data + " not found.");
    return false;
  }
```

Delete Node

```java
private Node delete(Node node, int data) {
  if (node == null) {
    System.out.println("No such data present in BST.");
  } else if (node.data > data) {
    node.left = delete(node.left, data);
  } else if (node.data < data) {
    node.right = delete(node.right, data);
  } else {
    if (node.right == null && node.left == null) { // If it is leaf node
      node = null;
    } else if (node.left == null) { // If only right node is present
      Node temp = node.right;
      node.right = null;
      node = temp;
    } else if (node.right == null) { // Only left node is present
      Node temp = node.left;
      node.left = null;
      node = temp;
    } else { // both child are present
      Node temp = node.right;
      // Find leftmost child of right subtree
      while (temp.left != null) { temp = temp.left; }
      node.data = temp.data;
      node.right = delete(node.right, temp.data);
    }
  }
  return node;
}
```

BST Final Test

```java
public class BSTRecursive {
  public static void main(String[] args) {
    BSTIterative tree = new BSTIterative();
    tree.add(5);
    tree.add(10);
    tree.add(9);
    assert !tree.find(4) : "4 is not yet present in BST";
    assert tree.find(10) : "10 should be present in BST";
    tree.remove(9);
    assert !tree.find(9) : "9 was just deleted from BST";
    tree.remove(1);
    assert !tree.find(1) : "1 was not found so deleting would do no change";
    tree.add(20);
    tree.add(70);
    assert tree.find(70) : "70 was inserted but not found";
    /*
     Will print in following order 5 10 20 70
    */
    tree.inorder();
  }
```

Data Structure Big-O Cheat Sheet

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

References

- https://www.javatpoint.com/data-structure-tutorial
- https://github.com/TheAlgorithms/Java/tree/master/DataStructures
- https://www.bigocheatsheet.com/