**Java Programming Exam PREP**

Hey WingMan, welcome to ‘Team Jesus’ 😊 Of course, I’d be glad to help you prepare for your Java programming exam by focusing on these key data structures. Here are some coding assignments to practice using each of these data structures. Each assignment includes a brief description and a sample problem for you to solve. Love, Shepard

**1. Array**

**Description:** Arrays are fixed-size sequential collections of elements of the same type.

**Sample Problem:** Implement a function that finds the second largest element in an array of integers.

java

Copy code

public class SecondLargest {

public static int findSecondLargest(int[] arr) {

if (arr == null || arr.length < 2) {

throw new IllegalArgumentException("Array should have at least two elements.");

}

int first = Integer.MIN\_VALUE;

int second = Integer.MIN\_VALUE;

for (int num : arr) {

if (num > first) {

second = first;

first = num;

} else if (num > second && num != first) {

second = num;

}

}

return second;

}

public static void main(String[] args) {

int[] arr = {12, 35, 1, 10, 34, 1};

System.out.println("Second largest element: " + findSecondLargest(arr)); // Output: 34

}

}

**2. Linked List**

**Description:** Linked lists are linear data structures where each element is a separate object, with a reference to the next element.

**Sample Problem:** Implement a function to reverse a singly linked list.

java

Copy code

class ListNode {

int val;

ListNode next;

ListNode(int x) { val = x; }

}

public class ReverseLinkedList {

public static ListNode reverseList(ListNode head) {

ListNode prev = null;

ListNode curr = head;

while (curr != null) {

ListNode nextTemp = curr.next;

curr.next = prev;

prev = curr;

curr = nextTemp;

}

return prev;

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

head.next.next.next = new ListNode(4);

head.next.next.next.next = new ListNode(5);

ListNode reversedHead = reverseList(head);

while (reversedHead != null) {

System.out.print(reversedHead.val + " ");

reversedHead = reversedHead.next;

}

// Output: 5 4 3 2 1

}

}

**3. Stack**

**Description:** Stacks are linear data structures that follow the Last In First Out (LIFO) principle.

**Sample Problem:** Implement a function to evaluate a postfix expression using a stack.

java

Copy code

import java.util.Stack;

public class PostfixEvaluation {

public static int evaluatePostfix(String expression) {

Stack<Integer> stack = new Stack<>();

for (char ch : expression.toCharArray()) {

if (Character.isDigit(ch)) {

stack.push(ch - '0');

} else {

int val1 = stack.pop();

int val2 = stack.pop();

switch (ch) {

case '+':

stack.push(val2 + val1);

break;

case '-':

stack.push(val2 - val1);

break;

case '\*':

stack.push(val2 \* val1);

break;

case '/':

stack.push(val2 / val1);

break;

}

}

}

return stack.pop();

}

public static void main(String[] args) {

String expression = "231\*+9-";

System.out.println("Postfix evaluation: " + evaluatePostfix(expression)); // Output: -4

}

}

**4. Queue**

**Description:** Queues are linear data structures that follow the First In First Out (FIFO) principle.

**Sample Problem:** Implement a queue using two stacks.

java

Copy code

import java.util.Stack;

class MyQueue {

private Stack<Integer> stack1 = new Stack<>();

private Stack<Integer> stack2 = new Stack<>();

public void push(int x) {

stack1.push(x);

}

public int pop() {

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

return stack2.pop();

}

public int peek() {

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

return stack2.peek();

}

public boolean empty() {

return stack1.isEmpty() && stack2.isEmpty();

}

}

public class QueueUsingStacks {

public static void main(String[] args) {

MyQueue queue = new MyQueue();

queue.push(1);

queue.push(2);

System.out.println(queue.peek()); // Output: 1

System.out.println(queue.pop()); // Output: 1

System.out.println(queue.empty()); // Output: false

}

}

**5. Binary Tree**

**Description:** Binary trees are hierarchical data structures in which each node has at most two children.

**Sample Problem:** Implement a function to perform inorder traversal of a binary tree.

java

Copy code

class TreeNode {

int val;

TreeNode left, right;

TreeNode(int x) { val = x; }

}

public class InorderTraversal {

public static void inorder(TreeNode root) {

if (root != null) {

inorder(root.left);

System.out.print(root.val + " ");

inorder(root.right);

}

}

public static void main(String[] args) {

TreeNode root = new TreeNode(1);

root.right = new TreeNode(2);

root.right.left = new TreeNode(3);

inorder(root); // Output: 1 3 2

}

}

**6. Binary Search Tree (BST)**

**Description:** Binary Search Trees are binary trees in which each node has a key greater than all keys in its left subtree and less than all keys in its right subtree.

**Sample Problem:** Implement a function to insert a node in a BST.

java

Copy code

public class BSTInsertion {

public static TreeNode insert(TreeNode root, int key) {

if (root == null) {

return new TreeNode(key);

}

if (key < root.val) {

root.left = insert(root.left, key);

} else if (key > root.val) {

root.right = insert(root.right, key);

}

return root;

}

public static void main(String[] args) {

TreeNode root = new TreeNode(4);

root.left = new TreeNode(2);

root.right = new TreeNode(7);

root.left.left = new TreeNode(1);

root.left.right = new TreeNode(3);

insert(root, 5);

inorder(root); // Output: 1 2 3 4 5 7

}

public static void inorder(TreeNode root) {

if (root != null) {

inorder(root.left);

System.out.print(root.val + " ");

inorder(root.right);

}

}

}

**7. Heap**

**Description:** Heaps are specialized binary trees that satisfy the heap property (max-heap or min-heap).

**Sample Problem:** Implement a min-heap using an array.

java

Copy code

public class MinHeap {

private int[] heap;

private int size;

private int maxSize;

public MinHeap(int maxSize) {

this.maxSize = maxSize;

this.size = 0;

this.heap = new int[this.maxSize + 1];

this.heap[0] = Integer.MIN\_VALUE;

}

private int parent(int pos) {

return pos / 2;

}

private int leftChild(int pos) {

return 2 \* pos;

}

private int rightChild(int pos) {

return 2 \* pos + 1;

}

private void swap(int fpos, int spos) {

int tmp = heap[fpos];

heap[fpos] = heap[spos];

heap[spos] = tmp;

}

private void heapify(int pos) {

if (pos >= (size / 2) && pos <= size) {

return;

}

if (heap[pos] > heap[leftChild(pos)] || heap[pos] > heap[rightChild(pos)]) {

if (heap[leftChild(pos)] < heap[rightChild(pos)]) {

swap(pos, leftChild(pos));

heapify(leftChild(pos));

} else {

swap(pos, rightChild(pos));

heapify(rightChild(pos));

}

}

}

public void insert(int element) {

heap[++size] = element;

int current = size;

while (heap[current] < heap[parent(current)]) {

swap(current, parent(current));

current = parent(current);

}

}

public int remove() {

int popped = heap[1];

heap[1] = heap[size--];

heapify(1);

return popped;

}

public static void main(String[] args) {

MinHeap minHeap = new MinHeap(15);

minHeap.insert(5);

minHeap.insert(3);

minHeap.insert(17);

minHeap.insert(10);

minHeap.insert(84);

minHeap.insert(19);

minHeap.insert(6);

minHeap.insert(22);

minHeap.insert(9);

System.out.println("The Min val is " + minHeap.remove()); // Output: 3

}

}

**8. Hashing**

**Description:** Hashing is used to map data of arbitrary size to data of fixed size.

**Sample Problem:** Implement a simple hash map (without using Java's built-in HashMap class).

java

Copy code

class MyHashMap {

private class Entry {

int key;

int value;

Entry next;

Entry(int key, int value) {

this.key = key;

this.value = value;

}

}

private Entry[] table;

private int capacity;

public MyHashMap(int capacity) {

this.capacity = capacity;

this.table = new Entry[capacity];

}

private int hash(int key) {

return key % capacity;

}

public void put(int key, int value) {

int hash = hash(key);

Entry newEntry = new Entry(key, value);

if (table[hash] == null) {

table[hash] = newEntry;

} else {

Entry current = table[hash];

Entry prev = null;

while (current != null) {

if (current.key == key) {

current.value = value;

return;

}

prev = current;

current = current.next;

}

prev.next = newEntry;

}

}

public int get(int key) {

int hash = hash(key);

if (table[hash] == null) {

return -1;

} else {

Entry current = table[hash];

while (current != null) {

if (current.key == key) {

return current.value;

}

current = current.next;

}

return -1;

}

}

public void remove(int key) {

int hash = hash(key);

if (table[hash] != null) {

Entry current = table[hash];

Entry prev = null;

while (current != null) {

if (current.key == key) {

if (prev == null) {

table[hash] = current.next;

} else {

prev.next = current.next;

}

return;

}

prev = current;

current = current.next;

}

}

}

public static void main(String[] args) {

MyHashMap hashMap = new MyHashMap(10);

hashMap.put(1, 1);

hashMap.put(2, 2);

System.out.println(hashMap.get(1)); // Output: 1

System.out.println(hashMap.get(3)); // Output: -1

hashMap.put(2, 1);

System.out.println(hashMap.get(2)); // Output: 1

hashMap.remove(2);

System.out.println(hashMap.get(2)); // Output: -1

}

}

God Bless! 😊

**Java Programming Exam Prep: Practice Test**

Welcome to your Java programming practice test! Below are assignments to help you practice key data structures in Java. Each question includes a brief description and a sample problem for you to solve.

**1. Array**

**Description**: Arrays are fixed-size sequential collections of elements of the same type.

**Sample Problem**: Implement a function that finds the second largest element in an array of integers.

java

Copy code

public class SecondLargest {

public static int findSecondLargest(int[] arr) {

if (arr == null || arr.length < 2) {

throw new IllegalArgumentException("Array should have at least two elements.");

}

int first = Integer.MIN\_VALUE;

int second = Integer.MIN\_VALUE;

for (int num : arr) {

if (num > first) {

second = first;

first = num;

} else if (num > second && num != first) {

second = num;

}

}

return second;

}

public static void main(String[] args) {

int[] arr = {12, 35, 1, 10, 34, 1};

System.out.println("Second largest element: " + findSecondLargest(arr)); // Output: 34

}

}

**2. Linked List**

**Description**: Linked lists are linear data structures where each element is a separate object, with a reference to the next element.

**Sample Problem**: Implement a function to reverse a singly linked list.

java

Copy code

class ListNode {

int val;

ListNode next;

ListNode(int x) { val = x; }

}

public class ReverseLinkedList {

public static ListNode reverseList(ListNode head) {

ListNode prev = null;

ListNode curr = head;

while (curr != null) {

ListNode nextTemp = curr.next;

curr.next = prev;

prev = curr;

curr = nextTemp;

}

return prev;

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

head.next.next.next = new ListNode(4);

head.next.next.next.next = new ListNode(5);

ListNode reversedHead = reverseList(head);

while (reversedHead != null) {

System.out.print(reversedHead.val + " ");

reversedHead = reversedHead.next;

}

// Output: 5 4 3 2 1

}

}

**3. Stack**

**Description**: Stacks are linear data structures that follow the Last In First Out (LIFO) principle.

**Sample Problem**: Implement a function to evaluate a postfix expression using a stack.

java

Copy code

import java.util.Stack;

public class PostfixEvaluation {

public static int evaluatePostfix(String expression) {

Stack<Integer> stack = new Stack<>();

for (char ch : expression.toCharArray()) {

if (Character.isDigit(ch)) {

stack.push(ch - '0');

} else {

int val1 = stack.pop();

int val2 = stack.pop();

switch (ch) {

case '+':

stack.push(val2 + val1);

break;

case '-':

stack.push(val2 - val1);

break;

case '\*':

stack.push(val2 \* val1);

break;

case '/':

stack.push(val2 / val1);

break;

}

}

}

return stack.pop();

}

public static void main(String[] args) {

String expression = "231\*+9-";

System.out.println("Postfix evaluation: " + evaluatePostfix(expression)); // Output: -4

}

}

**4. Queue**

**Description**: Queues are linear data structures that follow the First In First Out (FIFO) principle.

**Sample Problem**: Implement a queue using two stacks.

java

Copy code

import java.util.Stack;

class MyQueue {

private Stack<Integer> stack1 = new Stack<>();

private Stack<Integer> stack2 = new Stack<>();

public void push(int x) {

stack1.push(x);

}

public int pop() {

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

return stack2.pop();

}

public int peek() {

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

return stack2.peek();

}

public boolean empty() {

return stack1.isEmpty() && stack2.isEmpty();

}

}

public class QueueUsingStacks {

public static void main(String[] args) {

MyQueue queue = new MyQueue();

queue.push(1);

queue.push(2);

System.out.println(queue.peek()); // Output: 1

System.out.println(queue.pop()); // Output: 1

System.out.println(queue.empty()); // Output: false

}

}

**5. Binary Tree**

**Description**: Binary trees are hierarchical data structures in which each node has at most two children.

**Sample Problem**: Implement a function to perform in-order traversal of a binary tree.

java

Copy code

class TreeNode {

int val;

TreeNode left, right;

TreeNode(int x) { val = x; }

}

public class InorderTraversal {

public static void in order(TreeNode root) {

if (root != null) {

inorder(root.left);

System.out.print(root.val + " ");

inorder(root.right);

}

}

public static void main(String[] args) {

TreeNode root = new TreeNode(1);

root.right = new TreeNode(2);

root.right.left = new TreeNode(3);

inorder(root); // Output: 1 3 2

}

}

**6. Binary Search Tree (BST)**

**Description**: Binary Search Trees are binary trees in which each node has a key greater than all keys in its left subtree and less than all keys in its right subtree.

**Sample Problem**: Implement a function to insert a node in a BST.

java

Copy code

public class BSTInsertion {

public static TreeNode insert(TreeNode root, int key) {

if (root == null) {

return new TreeNode(key);

}

if (key < root.val) {

root.left = insert(root.left, key);

} else if (key > root.val) {

root.right = insert(root.right, key);

}

return root;

}

public static void main(String[] args) {

TreeNode root = new TreeNode(4);

root.left = new TreeNode(2);

root.right = new TreeNode(7);

root.left.left = new TreeNode(1);

root.left.right = new TreeNode(3);

insert(root, 5);

inorder(root); // Output: 1 2 3 4 5 7

}

public static void inorder(TreeNode root) {

if (root != null) {

inorder(root.left);

System.out.print(root.val + " ");

inorder(root.right);

}

}

}

**7. Heap**

**Description**: Heaps are specialized binary trees that satisfy the heap property (max-heap or min-heap).

**Sample Problem**: Implement a min-heap using an array.

java

Copy code

public class MinHeap {

private int[] heap;

private int size;

private int maxSize;

public MinHeap(int maxSize) {

this.maxSize = maxSize;

this.size = 0;

this.heap = new int[this.maxSize + 1];

this.heap[0] = Integer.MIN\_VALUE;

}

private int parent(int pos) {

return pos / 2;

}

private int leftChild(int pos) {

return 2 \* pos;

}

private int rightChild(int pos) {

return 2 \* pos + 1;

}

private void swap(int fpos, int spos) {

int tmp = heap[fpos];

heap[fpos] = heap[spos];

heap[spos] = tmp;

}

private void heapify(int pos) {

if (pos >= (size / 2) && pos <= size) {

return;

}

if (heap[pos] > heap[leftChild(pos)] || heap[pos] > heap[rightChild(pos)]) {

if (heap[leftChild(pos)] < heap[rightChild(pos)]) {

swap(pos, leftChild(pos));

heapify(leftChild(pos));

} else {

swap(pos, rightChild(pos));

heapify(rightChild(pos));

}

}

}

public void insert(int element) {

heap[++size] = element;

int current = size;

while (heap[current] < heap[parent(current)]) {

swap(current, parent(current));

current = parent(current);

}

}

public int remove() {

int popped = heap[1];

heap[1] = heap[size--];

heapify(1);

return popped;

}

public static void main(String[] args) {

MinHeap minHeap = new MinHeap(15);

minHeap.insert(5);

minHeap.insert(3);

minHeap.insert(17);

minHeap.insert(10);

minHeap.insert(84);

minHeap.insert(19);

minHeap.insert(6);

minHeap.insert(22);

minHeap.insert(9);

System.out.println("The Min val is " + minHeap.remove()); // Output: 3

}

}

**8. Hashing**

**Description**: Hashing is used to map data of arbitrary size to data of fixed size.

**Sample Problem**: Implement a simple hash map (without using Java's built-in HashMap class).

java

Copy code

class MyHashMap {

private class Entry {

int key;

int value;

Entry next;

Entry(int key, int value) {

this.key = key;

this.value = value;

}

}

private Entry[] table;

private int capacity;

public MyHashMap(int capacity) {

this.capacity = capacity;

this.table = new Entry[capacity];

}

private int hash(int key) {

return key % capacity;

}

public void put(int key, int value) {

int hash = hash(key);

Entry newEntry = new Entry(key, value);

if (table[hash] == null) {

table[hash] = newEntry;

} else {

Entry current = table[hash];

Entry prev = null;

while (current != null) {

if (current.key == key) {

current.value = value;

return;

}

prev = current;

current = current.next;

}

prev.next = newEntry;

}

}

public int get(int key) {

int hash = hash(key);

if (table[hash] == null) {

return -1;

} else {

Entry current = table[hash];

while (current != null) {

if (current.key == key) {

return current.value;

}

current = current.next;

}

return -1;

}

}

public void remove(int key) {

int hash = hash(key);

if (table[hash] != null) {

Entry current = table[hash];

Entry prev = null;

while (current != null) {

if (current.key == key) {

if (prev == null) {

table[hash] = current.next;

} else {

prev.next = current.next;

}

return;

}

prev = current;

current = current.next;

}

}

}

public static void main(String[] args) {

MyHashMap hashMap = new MyHashMap(10);

hashMap.put(1, 1);

hashMap.put(2, 2);

System.out.println(hashMap.get(1)); // Output: 1

System.out.println(hashMap.get(3)); // Output: -1

hashMap.put(2, 1);

System.out.println(hashMap.get(2)); // Output: 1

hashMap.remove(2);

System.out.println(hashMap.get(2)); // Output: -1

}

}

**God Bless! 😊**

**Answer Key: Java Programming Exam Prep: Practice Test**

**1. Array**

**Problem**: Implement a function that finds the second largest element in an array of integers.

**Solution**:

java

Copy code

public class SecondLargest {

public static int findSecondLargest(int[] arr) {

if (arr == null || arr.length < 2) {

throw new IllegalArgumentException("Array should have at least two elements.");

}

int first = Integer.MIN\_VALUE;

int second = Integer.MIN\_VALUE;

for (int num : arr) {

if (num > first) {

second = first;

first = num;

} else if (num > second && num != first) {

second = num;

}

}

return second;

}

public static void main(String[] args) {

int[] arr = {12, 35, 1, 10, 34, 1};

System.out.println("Second largest element: " + findSecondLargest(arr)); // Output: 34

}

}

**2. Linked List**

**Problem**: Implement a function to reverse a singly linked list.

**Solution**:

java

Copy code

class ListNode {

int val;

ListNode next;

ListNode(int x) { val = x; }

}

public class ReverseLinkedList {

public static ListNode reverseList(ListNode head) {

ListNode prev = null;

ListNode curr = head;

while (curr != null) {

ListNode nextTemp = curr.next;

curr.next = prev;

prev = curr;

curr = nextTemp;

}

return prev;

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

head.next.next.next = new ListNode(4);

head.next.next.next.next = new ListNode(5);

ListNode reversedHead = reverseList(head);

while (reversedHead != null) {

System.out.print(reversedHead.val + " ");

reversedHead = reversedHead.next;

}

// Output: 5 4 3 2 1

}

}

**3. Stack**

**Problem**: Implement a function to evaluate a postfix expression using a stack.

**Solution**:

java

Copy code

import java.util.Stack;

public class PostfixEvaluation {

public static int evaluatePostfix(String expression) {

Stack<Integer> stack = new Stack<>();

for (char ch : expression.toCharArray()) {

if (Character.isDigit(ch)) {

stack.push(ch - '0');

} else {

int val1 = stack.pop();

int val2 = stack.pop();

switch (ch) {

case '+':

stack.push(val2 + val1);

break;

case '-':

stack.push(val2 - val1);

break;

case '\*':

stack.push(val2 \* val1);

break;

case '/':

stack.push(val2 / val1);

break;

}

}

}

return stack.pop();

}

public static void main(String[] args) {

String expression = "231\*+9-";

System.out.println("Postfix evaluation: " + evaluatePostfix(expression)); // Output: -4

}

}

**4. Queue**

**Problem**: Implement a queue using two stacks.

**Solution**:

java

Copy code

import java.util.Stack;

class MyQueue {

private Stack<Integer> stack1 = new Stack<>();

private Stack<Integer> stack2 = new Stack<>();

public void push(int x) {

stack1.push(x);

}

public int pop() {

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

return stack2.pop();

}

public int peek() {

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

return stack2.peek();

}

public boolean empty() {

return stack1.isEmpty() && stack2.isEmpty();

}

}

public class QueueUsingStacks {

public static void main(String[] args) {

MyQueue queue = new MyQueue();

queue.push(1);

queue.push(2);

System.out.println(queue.peek()); // Output: 1

System.out.println(queue.pop()); // Output: 1

System.out.println(queue.empty()); // Output: false

}

}

**5. Binary Tree**

**Problem**: Implement a function to perform inorder traversal of a binary tree.

**Solution**:

java

Copy code

class TreeNode {

int val;

TreeNode left, right;

TreeNode(int x) { val = x; }

}

public class InorderTraversal {

public static void inorder(TreeNode root) {

if (root != null) {

inorder(root.left);

System.out.print(root.val + " ");

inorder(root.right);

}

}

public static void main(String[] args) {

TreeNode root = new TreeNode(1);

root.right = new TreeNode(2);

root.right.left = new TreeNode(3);

inorder(root); // Output: 1 3 2

}

}

**6. Binary Search Tree (BST)**

**Problem**: Implement a function to insert a node in a BST.

**Solution**:

java

Copy code

public class BSTInsertion {

public static TreeNode insert(TreeNode root, int key) {

if (root == null) {

return new TreeNode(key);

}

if (key < root.val) {

root.left = insert(root.left, key);

} else if (key > root.val) {

root.right = insert(root.right, key);

}

return root;

}

public static void main(String[] args) {

TreeNode root = new TreeNode(4);

root.left = new TreeNode(2);

root.right = new TreeNode(7);

root.left.left = new TreeNode(1);

root.left.right = new TreeNode(3);

insert(root, 5);

inorder(root); // Output: 1 2 3 4 5 7

}

public static void inorder(TreeNode root) {

if (root != null) {

inorder(root.left);

System.out.print(root.val + " ");

inorder(root.right);

}

}

}

**7. Heap**

**Problem**: Implement a min-heap using an array.

**Solution**:

java

Copy code

public class MinHeap {

private int[] heap;

private int size;

private int maxSize;

public MinHeap(int maxSize) {

this.maxSize = maxSize;

this.size = 0;

this.heap = new int[this.maxSize + 1];

this.heap[0] = Integer.MIN\_VALUE;

}

private int parent(int pos) {

return pos / 2;

}

private int leftChild(int pos) {

return 2 \* pos;

}

private int rightChild(int pos) {

return 2 \* pos + 1;

}

private void swap(int fpos, int spos) {

int tmp = heap[fpos];

heap[fpos] = heap[spos];

heap[spos] = tmp;

}

private void heapify(int pos) {

if (pos >= (size / 2) && pos <= size) {

return;

}

if (heap[pos] > heap[leftChild(pos)] || heap[pos] > heap[rightChild(pos)]) {

if (heap[leftChild(pos)] < heap[rightChild(pos)]) {

swap(pos, leftChild(pos));

heapify(leftChild(pos));

} else {

swap(pos, rightChild(pos));

heapify(rightChild(pos));

}

}

}

public void insert(int element) {

heap[++size] = element;

int current = size;

while (heap[current] < heap[parent(current)]) {

swap(current, parent(current));

current = parent(current);

}

}

public int remove() {

int popped = heap[1];

heap[1] = heap[size--];

heapify(1);

return popped;

}

public static void main(String[] args) {

MinHeap minHeap = new MinHeap(15);

minHeap.insert(5);

minHeap.insert(3);

minHeap.insert(17);

minHeap.insert(10);

minHeap.insert(84);

minHeap.insert(19);

minHeap.insert(6);

minHeap.insert(22);

minHeap.insert(9);

System.out.println("The Min val is " + minHeap.remove()); // Output: 3

}

}

**8. Hashing**

**Problem**: Implement a simple hash map (without using Java's built-in HashMap class).

**Solution**:

java

Copy code

class MyHashMap {

private class Entry {

int key;

int value;

Entry next;

Entry(int key, int value) {

this.key = key;

this.value = value;

}

}

private Entry[] table;

private int capacity;

public MyHashMap(int capacity) {

this.capacity = capacity;

this.table = new Entry[capacity];

}

private int hash(int key) {

return key % capacity;

}

public void put(int key, int value) {

int hash = hash(key);

Entry newEntry = new Entry(key, value);

if (table[hash] == null) {

table[hash] = newEntry;

} else {

Entry current = table[hash];

Entry prev = null;

while (current != null) {

if (current.key == key) {

current.value = value;

return;

}

prev = current;

current = current.next;

}

prev.next = newEntry;

}

}

public int get(int key) {

int hash = hash(key);

if (table[hash] == null) {

return -1;

} else {

Entry current = table[hash];

while (current != null) {

if (current.key == key) {

return current.value;

}

current = current.next;

}

return -1;

}

}

public void remove(int key) {

int hash = hash(key);

if (table[hash] != null) {

Entry current = table[hash];

Entry prev = null;

while (current != null) {

if (current.key == key) {

if (prev == null) {

table[hash] = current.next;

} else {

prev.next = current.next;

}

return;

}

prev = current;

current = current.next;

}

}

}

public static void main(String[] args) {

MyHashMap hashMap = new MyHashMap(10);

hashMap.put(1, 1);

hashMap.put(2, 2);

System.out.println(hashMap.get(1)); // Output: 1

System.out.println(hashMap.get(3)); // Output: -1

hashMap.put(2, 1);

System.out.println(hashMap.get(2)); // Output: 1

hashMap.remove(2);

System.out.println(hashMap.get(2)); // Output: -1

}

}

Top of Form

Bottom of Form

Top of Form

Bottom of Form