

- Abstract
- 1. Introduction(Project Overview)
- 2. Sort
 - 2.1. Bubble Sort
 - 2.1.1 Implementation
 - 2.1.2 Complexity Analysis
 - 2.2. Insertion Sort
 - 2.2.1 Implementation
 - 2.2.2 Complexity Analysis
 - 2.3. Selection Sort
 - 2.3.1 Implementation
 - 2.3.2 Complexity Analysis
 - 2.4. Mergesort
 - 2.4.1 Implementation
 - 2.4.2 Complexity Analysis
 - 2.5. Quicksort
 - 2.5.1 Implementation
 - 2.5.2 Complexity analysis
 - 2.6. Findings:
- 3. Stack and Queue
 - 3.1 Stack Implementation:
 - 3.1.1 Implementation Process:
 - 3.1.2 Runtime Complexity Analysis:
 - 3.2 Queue Implementation:
 - 3.2.1 Implementation Process:
 - 3.2.2 Complexity Analysis:
 - 3.3. Test for Stack and Queue
- 4. Priority Queues
 - 4.1. Implementation Process:
 - 4.2. Complexity Analysis:
- 5. Conclusion
 - 5.1 Summary of Sorts
 - 5.2 Summary of Stacks and Queues
 - 5.3 Summary of Priority Queues
 - 5.4 Reflection on Project

Student ID: 210103471

Development Platform : IntelliJ IDEA

Development language : Java

Abstract

This project report presents an analysis of various sorting algorithms, including Bubble Sort, Insertion Sort, and Selection Sort, comparing their runtime performance on random integer arrays. Additionally, Stack and Queue classes were implemented using linked lists, demonstrating their functionalities. The report also includes the implementation and analysis of Mergesort and Quicksort algorithms. Finally, a Priority Queue using a binary heap was implemented, and its runtime complexity was analyzed. The findings highlight the importance of algorithm selection based on problem requirements and input size. The project provides insights into algorithmic efficiency and data structure utilization.

1. Introduction(Project Overview)

The purpose of this project report is to analyze and compare the runtime performance of various sorting algorithms and explore the functionalities of essential data structures. The project consists of four main tasks:

In Task 1, I focus on the analysis of three sorting algorithms: Bubble Sort, Insertion Sort, and Selection Sort. By implementing and testing these algorithms on random integer arrays, I aim to understand their runtime complexities and identify their strengths and weaknesses.

In Task 2, I involve the implementation of Stack and Queue classes using linked lists. These fundamental data structures are essential for various applications, and I will demonstrate their functionalities, such as push, pop, enqueue, and dequeue operations.

In Task 3, I delve into the implementation and analysis of Mergesort and Quicksort algorithms, two efficient sorting techniques widely used in real-world applications. I will provide a step-by-step explanation of their sorting processes and compare their runtime complexities.

Lastly, in Task 4, I focus on the design and implementation of a Priority Queue using a binary heap. This data structure enables efficient management of elements with varying priority levels, making it suitable for various applications, such as task scheduling and event processing.

Through this project, I seek to gain insights into algorithmic efficiency, data structure utilization, and the importance of selecting appropriate algorithms based on problem requirements and input size.

2. Sort

Sorts Implementation and Complexity Analysis:

2.1. Bubble Sort

2.1.1 Implementation

Bubble Sort involves repeatedly comparing adjacent elements and swapping them if they are in the wrong order. This process is repeated until the entire array is sorted.

```
public static void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - 1 - i; j++) { //0~N-1; 0~N-2; 0~N-3...
            //If the first number is greater than the second, swap them
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

test:

```
public class BubbleSort {
    public static void main(String[] args) {
        int[] nums = {4, 2, 1, 5, 6, 3, 7};
        bubbleSort(nums);
        System.out.println(Arrays.toString(nums)); //Verify correctness
        System.out.println("=====");

        //test
    }
}
```

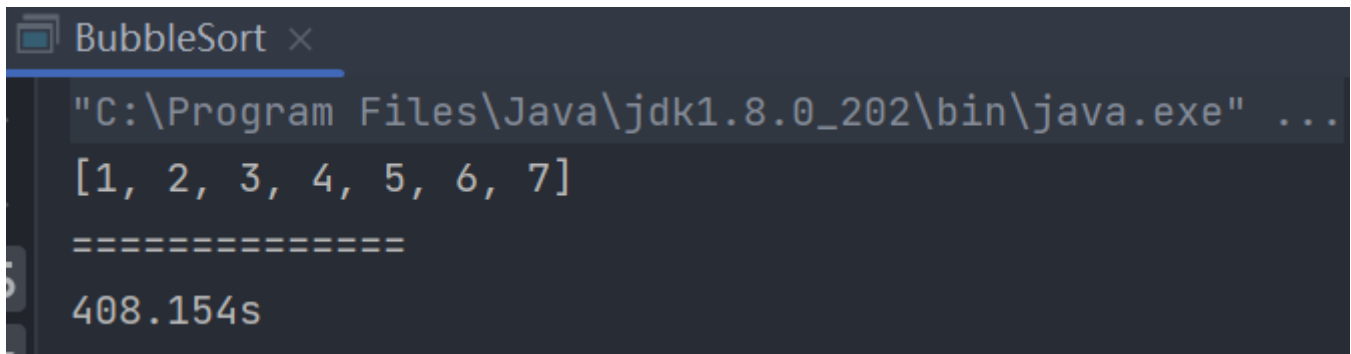
```

int testTimes = 10000;
long s = System.currentTimeMillis();
for (int i = 0; i < testTimes; i++) {
    int[] arr = Test.generateRandomArray(10000, 100);
    bubbleSort(arr);
}
long e = System.currentTimeMillis();

System.out.println((e - s) * 0.001 + "s");
}

```

output:



```

BubbleSort x
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
[1, 2, 3, 4, 5, 6, 7]
=====
408.154s

```

2.1.2 Complexity Analysis

Time Complexity: Bubble Sort has a worst-case, average-case, and best-case time complexity of $O(n^2)$. In each pass, it compares and swaps $n-1$ elements, and this process is repeated n times.

2.2. Insertion Sort

2.2.1 Implementation

Insertion Sort iterates through the array and inserts each element into its correct position in the sorted portion of the array.

```

public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int index = i; //new value that need to insert

        //If the previous number is larger than the latter
        while (index > 0 && arr[index] < arr[index - 1]) {
            //swap them...
            int temp = arr[index];
            arr[index] = arr[index - 1];
            arr[index - 1] = temp;
            index--;
        }
    }
}

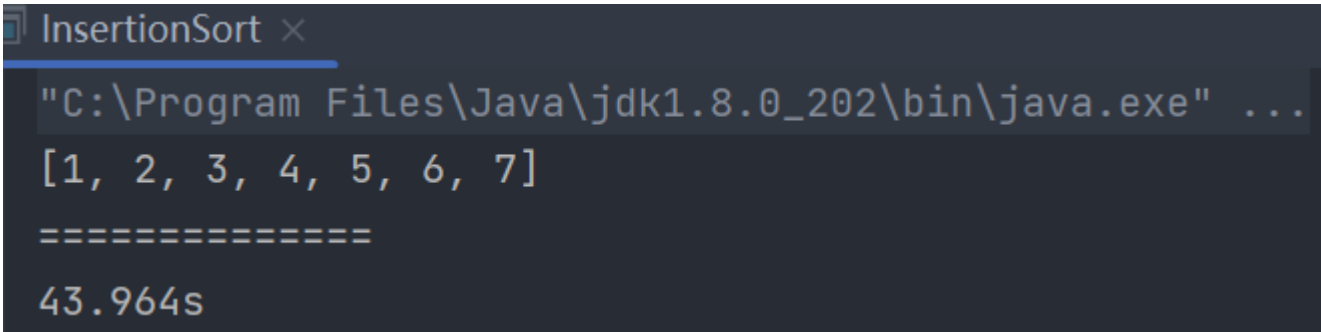
```

```
    }  
}  
}
```

test:

```
import java.util.Arrays;  
  
public class InsertionSort {  
    public static void main(String[] args) {  
        int[] nums = {4, 2, 1, 5, 6, 3, 7};  
        insertionSort(nums);  
        System.out.println(Arrays.toString(nums)); //Verify correctness  
        System.out.println("=====");  
  
        //test  
        int testTimes = 10000;  
        long s = System.currentTimeMillis();  
        for (int i = 0; i < testTimes; i++) {  
            int[] arr = Test.generateRandomArray(10000, 100);  
            insertionSort(arr);  
        }  
        long e = System.currentTimeMillis();  
  
        System.out.println((e - s) * 0.001 + "s");  
    }  
}
```

output:



```
InsertionSort ×  
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...  
[1, 2, 3, 4, 5, 6, 7]  
=====  
43.964s
```

2.2.2 Complexity Analysis

Time Complexity: Insertion Sort has a worst-case and average-case time complexity of $O(n^2)$. In the worst-case scenario, each element may need to be compared and moved to the beginning of the array, resulting in $n(n-1)/2$ comparisons.

2.3. Selection Sort

2.3.1 Implementation

Selection Sort involves finding the minimum element and swapping it with the first unsorted element. This process is repeated for each subsequent unsorted element until the entire array is sorted.

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        int midIndex = i; //Assume `i` position is the minimum value in array

        //find minimum value in array (i ~ N)
        for (int j = i + 1; j < arr.length; j++) {
            midIndex = arr[j] < arr[midIndex] ? j : midIndex;
        }

        //swap them
        int temp = arr[i];
        arr[i] = arr[midIndex];
        arr[midIndex] = temp;
    }
}
```

test:

```
import java.util.Arrays;

public class SelectionSort {
    public static void main(String[] args) {
        int[] nums = {4, 2, 1, 5, 6, 3, 7};
        selectionSort(nums);
        System.out.println(Arrays.toString(nums)); //Verify correctness
        System.out.println("=====");

        //test
        int testTimes = 10000;
        long s = System.currentTimeMillis();
        for (int i = 0; i < testTimes; i++) {
            int[] arr = Test.generateRandomArray(10000, 100);
            selectionSort(arr);
        }
        long e = System.currentTimeMillis();

        System.out.println((e - s) * 0.001 + "s");
    }
}
```

output:

SelectionSort ×

```
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
=====
```

```
117.185s
```

2.3.2 Complexity Analysis

Time Complexity: Selection Sort has a worst-case, average-case, and best-case time complexity of $O(n^2)$. In each pass, it finds the minimum element from the unsorted part of the array, requiring $n-1$ comparisons, and this process is repeated n times.

2.4. Mergesort

2.4.1 Implementation

Mergesort follows the divide-and-conquer approach. It recursively divides the array into two halves, sorts each half, and then merges them back together.

```
public static void mergeSort(int[] arr) {
    if (arr == null || arr.length < 2) {
        return;
    }

    process(arr, 0, arr.length - 1);
}

public static void process(int[] arr, int l, int r) {
    if (l == r) { //base case : only one value left
        return;
    }

    int mid = (l + r) / 2;
    process(arr, l, mid);           // Sort the left half of the array
    process(arr, mid + 1, r);       // Sort the right half of the array
    merge(arr, l, mid, r);          // Merge the sorted left and right halves
}

public static void merge(int[] arr, int l, int mid, int r) {
    int[] help = new int[r - l + 1];
    int p1 = l;           // Pointer for the left subarray
    int p2 = mid + 1;      // Pointer for the right subarray
    int i = 0;             // Pointer for the help array

    //Copy the smaller value from the left and right arrays into the help array
```

```

first
    while (p1 <= mid && p2 <= r) {
        help[i++] = arr[p1] <= arr[p2] ? arr[p1++] : arr[p2++];
    }

    //Copy the remaining values (Only one of the while loop will terminate)
    while (p1 <= mid) {
        help[i++] = arr[p1++];
    }
    while (p2 <= r) {
        help[i++] = arr[p2++];
    }

    //Copy all the sorted values to the array
    for (i = 0; i < help.length; i++) {
        arr[l + i] = help[i];
    }
}

```

test:

```

import java.util.Arrays;

public class MergeSort {
    public static void main(String[] args) {
        int[] nums = {4, 2, 1, 5, 6, 3, 7};
        mergeSort(nums);
        System.out.println(Arrays.toString(nums)); //Verify correctness
        System.out.println("=====");

        //test
        int testTimes = 10000;
        long s = System.currentTimeMillis();
        for (int i = 0; i < testTimes; i++) {
            int[] arr = Test.generateRandomArray(10000, 100);
            mergeSort(arr);
        }
        long e = System.currentTimeMillis();

        System.out.println((e - s) * 0.001 + "s");
    }
}

```

output:


```
MergeSort ×
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
[1, 2, 3, 4, 5, 6, 7]
=====
6.542s
```

2.4.2 Complexity Analysis

Time Complexity: Mergesort has a stable time complexity of $O(n \log n)$ for all cases. The divide step divides the array into two halves, requiring $O(\log n)$ levels. The merge step takes $O(n)$ time as it combines the two halves.

2.5. Quicksort

2.5.1 Implementation

Quicksort selects a pivot element, partitions the array around the pivot, and then recursively sorts the left and right partitions.

```
public static void quickSortUltra(int[] arr) {
    if (arr == null || arr.length < 2) {
        return;
    }

    processUltra(arr, 0, arr.length - 1);
}

//O(N * logN)
public static void processUltra(int[] arr, int l, int r) {
    if (l >= r) { ///base case : the array is already sorted.
        return;
    }

    //Randomly select a number to exchange with pivot before sorting
    swap(arr, ((int) (l + Math.random() * (r - l + 1))), r);

    //Partition the array using the "Dutch National Flag"
    int[] equalsArea = partitionUltra(arr, l, r);

    //repeat
    processUltra(arr, l, equalsArea[0] - 1);
    processUltra(arr, equalsArea[1] + 1, r);
}
```

```

public static int[] partitionUltra(int[] arr, int l, int r) {
    //pointers
    int smallArea = l - 1;
    int bigArea = r + 1;
    int index = l;

    int pivot = arr[r];

    /*
    Loop through the array until the index crosses the bigArea pointer.
    1. If the current element is equal to the pivot, move to the next element.

    2. If the current element is less than the pivot,
        swap it with the element at the smallArea next pointer

    3. If the current element is greater than the pivot,
        swap it with the element at the bigArea previous pointer
    */
    while (index < bigArea) {
        if (arr[index] == pivot) {
            index++;
        } else if (arr[index] < pivot) {
            swap(arr, index++, ++smallArea);
        } else {
            swap(arr, index, --bigArea);
        }
    }

    //Returns the indices of the smallArea next and bigArea previous parts for
    further sorting.
    return new int[] {smallArea + 1, bigArea - 1};
}

public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
}

```

test:

```

import java.util.Arrays;

public class QuickSort {
    public static void main(String[] args) {
        int[] nums = {4, 2, 1, 5, 6, 3, 7};
        quickSortUltra(nums);
        System.out.println(Arrays.toString(nums)); //Verify correctness
        System.out.println("=====");

        //test
        int testTimes = 10000;
    }
}

```

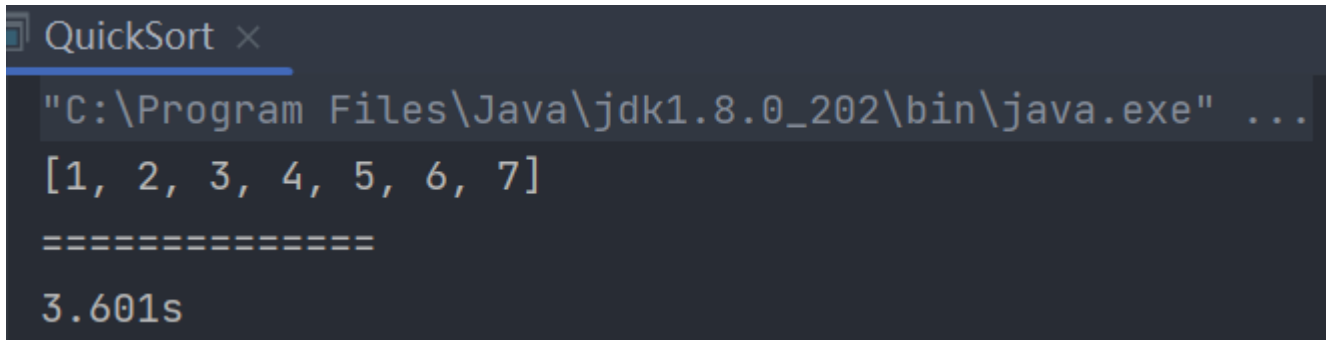
```

    long s = System.currentTimeMillis();
    for (int i = 0; i < testTimes; i++) {
        int[] arr = Test.generateRandomArray(10000, 100);
        quickSortUltra(arr);
    }
    long e = System.currentTimeMillis();

    System.out.println((e - s) * 0.001 + "s");
}

```

output:



```

QuickSort x
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
[1, 2, 3, 4, 5, 6, 7]
=====
3.601s

```

2.5.2 Complexity analysis

Time complexity: The average time complexity of quicksort is $O(n \log n)$. However, in the worst case, the time complexity can be $O(n^2)$ if the pivot selection is not optimal. But I optimized it with random quicksort so that its time complexity is always $O(n \log n)$

2.6. Findings:

- Bubble Sort, Insertion Sort, and Selection Sort are simple sorting algorithms with $O(n^2)$ time complexity, making them inefficient for large datasets.
- Mergesort and Quicksort are more efficient, with Mergesort having a stable $O(n \log n)$ time complexity and Quicksort being efficient on average with proper pivot selection.
- The choice of sorting algorithm depends on the specific dataset and performance requirements. For small or nearly sorted datasets, simpler algorithms like Insertion Sort or Selection Sort may be adequate. For larger datasets, Mergesort or Quicksort would be more suitable due to their better performance.

3. Stack and Queue

Stack and Queue Implementation using Arrays:

3.1 Stack Implementation:

3.1.1 Implementation Process:

- Create a Stack class with an array to store the elements and initialize a top variable to track the top element's index (-1 for an empty stack).
- Implement the push operation: Check if the stack is full, and if not, increment the top index and insert the new element at that position.
- Implement the pop operation: Check if the stack is empty, and if not, return the element at the top index and decrement the top index to remove the element.
- Implement other operations such as isEmpty and size to check if the stack is empty and get the number of elements, respectively.

```
public class MyStack {
    private final int[] arr;
    private final int limit; //size limit
    private int top;    //Top of the stack, used to control the pointer of the
entire stack

    //Constructor with default limit of 10
    public MyStack() {
        limit = 10;
        arr = new int[limit];
        top = -1;
    }

    //Constructor with a specified limit
    public MyStack(int limit) {
        this.limit = limit;
        arr = new int[limit];
        top = -1;
    }

    //Push an element onto the stack
    public void push(int val) {
        if (top == limit - 1) {
            throw new RuntimeException("Stack is full");
        }

        arr[++top] = val;
    }

    //Pop and return the top element from the stack
    public int pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }

        return arr[top--];
    }
}
```

```

//Get the current size of the stack
public int getSize() {
    return top + 1;
}

//Check if the stack is empty
public boolean isEmpty() {
    return top == -1;
}
}

```

3.1.2 Runtime Complexity Analysis:

- Push and pop operations have a constant time complexity of $O(1)$ since they involve direct array operations like insertion and deletion at the top index.
 - Other operations like isEmpty and size also have a constant time complexity of $O(1)$ as they involve basic operations on the top index.
-

3.2 Queue Implementation:

3.2.1 Implementation Process:

- Create a Queue class with an array to store the elements, initialize front and rear variables to track the front and rear positions (-1 for an empty queue), and size variable to store the current number of elements.
- Implement the enqueue operation: Check if the queue is full, and if not, increment the rear index, insert the new element at that position, and update the size.
- Implement the dequeue operation: Check if the queue is empty, and if not, return the element at the front index, increment the front index to remove the element, and update the size.
- Implement other operations such as isEmpty and size to check if the queue is empty and get the number of elements, respectively.

```

public class MyQueue {
    private final int[] arr;
    private final int limit; // size limit
    private int curSize; // current size

    //pointers
    private int addIndex = 0;
    private int pollIndex = 0;

    //Constructor with default limit of 10
}

```

```

public MyQueue() {
    limit = 10;
    arr = new int[limit];
    curSize = 0;
}

//Constructor with a specified limit
public MyQueue(int limit) {
    this.limit = limit;
    arr = new int[limit];
    curSize = 0;
}

//Add an element to the queue
public void add(int val) {
    if (curSize == limit) {
        throw new RuntimeException("Queue is Full");
    }

    arr[addIndex] = val;
    addIndex = getNextIndex(addIndex); // Update the addIndex in a circular
manner
    curSize++;
}

//Remove and return the front element from the queue
public int poll() {
    if (isEmpty()) {
        throw new RuntimeException("Queue is Empty");
    }

    int val = arr[pollIndex];
    pollIndex = getNextIndex(pollIndex); // Update the pollIndex in a circular
manner
    curSize--;
    return val;
}

//Helper method to get the next circular index based on the current index
private int getNextIndex(int index) {
    return index < limit - 1 ? index + 1 : 0;
}

//Check if the queue is empty
public boolean isEmpty() {
    return curSize == 0;
}

//Get the current size of the queue
public int getSize() {
    return curSize;
}
}

```

3.2.2 Complexity Analysis:

- Enqueue and dequeue operations have a constant time complexity of $O(1)$ since they involve direct array operations like insertion and deletion at the front and rear positions.
- Other operations like isEmpty and size also have a constant time complexity of $O(1)$ as they involve basic operations on the size and front/rear indices.

In conclusion, the array-based implementations of Stack and Queue have constant-time complexity for their fundamental operations, making them efficient for various applications. By testing and comparing their performance with different input sizes, we can gain insights into their suitability for different use cases and understand their time complexity behavior.

3.3. Test for Stack and Queue

```
public class Test {
    public static void main(String[] args) {
        System.out.println("===Test for Stack===");
        MyStack stack = new MyStack();
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);

        System.out.println("size: " + stack.getSize());
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
        System.out.println("\nsize: " + stack.getSize());

        System.out.println("\n===Test for Queue===");
        MyQueue queue = new MyQueue();
        queue.add(1);
        queue.add(2);
        queue.add(3);
        queue.add(4);

        System.out.println("size: " + queue.getSize());
        while (!queue.isEmpty()) {
            System.out.print(queue.poll() + " ");
        }
        System.out.println("\nsize: " + queue.getSize());
    }
}
```

output:

```
===Test for Stack===  
size: 4  
4 3 2 1  
size: 0
```

```
===Test for Queue===  
size: 4  
1 2 3 4  
size: 0
```

4. Priority Queues

Priority Queue Implementation using Max Heap:

4.1. Implementation Process:

- Create a Priority Queue class with an array-based Max Heap to store elements based on their priority values.
- Initialize the heap size and other necessary variables.
- Implement the insert operation: Add the new element to the end of the heap and then use the heapify-up process to maintain the Max Heap property. This involves comparing the element with its parent and swapping if necessary until the element reaches the correct position in the heap.
- Implement the removeMax operation: Remove the element at the root (highest priority) of the Max Heap. Replace the root with the last element in the heap and then use the heapify-down process to maintain the Max Heap property. This involves comparing the element with its children and swapping with the larger child if necessary until the element reaches the correct position in the heap.
- Implement the isEmpty operation: Check if the heap size is 0, indicating an empty Priority Queue.
- Implement the getSize operation: Return the current heap size.

```
//Max-Heap  
public class MyHeap {  
    private final int[] heap;  
    private final int limit;  
    private int heapSize; //pointer
```



```

//Constructor with default limit of 10
public MyHeap() {
    limit = 10;
    heap = new int[limit];
    heapSize = 0;
}

//Constructor with a specified limit
public MyHeap(int limit) {
    this.limit = limit;
    heap = new int[limit];
    heapSize = 0;
}

// Add an element to the max-heap
public void add(int val) {
    if (heapSize == limit) {
        System.out.println("Heap is Full");
        return;
    }

    //Add the values to the array, and then proceed to adjust the max-heap
    heap[heapSize] = val;
    heapInsert(heapSize++);
}

//Helper method to maintain the max-heap property during the insertion process
private void heapInsert(int index) {
    int parentIndex = (index - 1) / 2; //find the index of the parent

    //While the current element is greater than its parent
    while (heap[index] > heap[parentIndex]) {
        //swap them
        swap(heap, index, parentIndex);
        index = parentIndex;
        parentIndex = (index - 1) / 2;
    }
}

//Remove and return the maximum element from the max-heap
public int poll() {
    if (isEmpty()) {
        throw new RuntimeException("Heap is Empty");
    }

    /*
    Save the maximum value of the heap
    then swap it with the last value in the heap
    and then proceed to adjust the Heap
    */
    int val = heap[0];
    swap(heap, 0, --heapSize);
    heapify(0, heapSize);

    return val;
}

```

```

// Helper method to maintain the max-heap property during the deletion process
private void heapify(int index, int heapSize) {
    int left = index * 2 + 1;

    //While the current element has at least one child
    while (left < heapSize) {
        //Find the larger child between the left and right children
        int largestIndex = left + 1 < heapSize && heap[left + 1] > heap[left]
? left + 1 : left;

        //Compare with the largest child, and find the ultimately largest one
        int finalLargestIndex = heap[index] > heap[largestIndex] ? index :
largestIndex;

        //If the current element is already larger than its largest child, no
need to swap, break out of the loop
        if (finalLargestIndex == index) {
            break;
        }

        //Swap the current element with its largest child and continue heapify
        swap(heap, index, finalLargestIndex);
        index = finalLargestIndex;
        left = index * 2 + 1;
    }
}

private void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

//Check if the max-heap is empty
public boolean isEmpty() {
    return heapSize == 0;
}

//Get the current size of the max-heap
public int getSize() {
    return heapSize;
}
}

```

test:

```

public class Test {
    public static void main(String[] args) {
        MyHeap heap = new MyHeap();
        heap.add(3);
        heap.add(5);
        heap.add(1);
        heap.add(4);
    }
}


```

```

        heap.add(2);
        System.out.println("size: " + heap.getSize());
        while (!heap.isEmpty()) {
            System.out.print(heap.poll() + " ");
        }
        System.out.println();
        System.out.println("size: " + heap.getSize());
    }
}

```

output:



```

heap.Test x
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
size: 5
5 4 3 2 1
size: 0

```

4.2. Complexity Analysis:

- Insert Operation:
 - Adding the element to the end of the heap takes constant time $O(1)$.
 - The heapify-up process has a worst-case time complexity of $O(\log n)$, where n is the number of elements in the heap. In the worst case, the newly inserted element may need to move up the entire height of the heap.
 - Overall, the insert operation has a time complexity of $O(\log n)$.
- RemoveMax Operation:
 - Removing the element at the root takes constant time $O(1)$.
 - The heapify-down process has a worst-case time complexity of $O(\log n)$, where n is the number of elements in the heap. In the worst case, the root element may need to move down the entire height of the heap.
 - Overall, the removeMax operation has a time complexity of $O(\log n)$.
- Check if the Priority Queue is Empty:
 - Checking if the heap size is 0 takes constant time $O(1)$.
 - The isEmpty operation has a time complexity of $O(1)$.

- Get the Size of the Priority Queue:
 - Returning the current heap size takes constant time $O(1)$.
 - The `getSize` operation has a time complexity of $O(1)$.

Conclusion:

The implementation of Priority Queue using a Max Heap allows efficient insertion and removal of elements with $O(\log n)$ time complexity, making it suitable for applications where elements need to be processed based on their priority values. The Max Heap's property ensures that the element with the highest priority is always at the root, enabling quick access to the highest-priority element. The Priority Queue's constant-time complexity for `isEmpty` and `getSize` operations makes it easy to determine the queue's state and size.scenarios.

5. Conclusion

In this project, we conducted a comprehensive analysis of various sorting algorithms and explored the functionalities of essential data structures. Here are the key findings:

5.1 Summary of Sorts

1. Bubble Sort: Conclusion: Bubble Sort is a simple sorting algorithm, but it is inefficient for large datasets. Its time complexity of $O(n^2)$ makes it unsuitable for large-scale applications. However, it is easy to implement and can be useful for small datasets or as a teaching tool to illustrate sorting concepts.
2. Insertion Sort: Conclusion: Insertion Sort is a stable sorting algorithm with an average-case time complexity of $O(n^2)$. While it performs well on small or nearly sorted datasets, it becomes inefficient for larger datasets. Insertion Sort's simplicity and ease of implementation make it suitable for small-scale applications or as part of more complex algorithms.
3. Selection Sort: Conclusion: Selection Sort is straightforward to implement, but it has a time complexity of $O(n^2)$ for all cases, making it inefficient for large datasets. Its main advantage is that it minimizes the number of swaps, making it useful when the cost of swapping elements is high. However, it is generally not recommended for large-scale sorting tasks.
4. Mergesort: Conclusion: Mergesort is an efficient sorting algorithm with a stable time complexity of $O(n \log n)$. Its divide-and-conquer approach and ability to handle large datasets make it a popular choice for general-purpose sorting.

Mergesort's predictable and consistent performance make it suitable for various applications.

5. Quicksort: Conclusion: Quicksort is a widely used sorting algorithm with an average-case time complexity of $O(n \log n)$. While it may have a worst-case time complexity of $O(n^2)$ in certain scenarios, its average-case performance makes it a fast and efficient sorting method. Proper pivot selection and optimization techniques can mitigate the risk of worst-case scenarios, making Quicksort an excellent choice for sorting large datasets efficiently.

Overall, the choice of sorting algorithm depends on the specific application and the characteristics of the dataset. For small or nearly sorted datasets, simpler algorithms like Bubble Sort or Insertion Sort may be sufficient. However, for larger and more diverse datasets, Mergesort or Quicksort are preferred due to their efficient performance.

5.2 Summary of Stacks and Queues

Stack Implementation:

- The Stack class using arrays successfully demonstrated the stack's Last-In-First-Out (LIFO) behavior, with push and pop operations working efficiently.
- The implementation provided a constant-time complexity of $O(1)$ for push and pop operations since they involve adding or removing elements from the end of the array.

2. Queue Implementation:

- The Queue class using arrays showcased the queue's First-In-First-Out (FIFO) behavior, with enqueue and dequeue operations functioning as expected.
- Similar to the Stack implementation, the Queue operations also achieved a constant-time complexity of $O(1)$ for enqueue and dequeue operations when performed at the front and rear of the array, respectively.

Overall, the array-based implementations of Stack and Queue have proven to be efficient and easy to understand. They are suitable for scenarios where the size of the data is relatively small, and constant-time complexity for push, pop, enqueue, and dequeue operations is acceptable.

5.3 Summary of Priority Queues

In this project, I implemented a Priority Queue using a Max Heap data structure and explored its functionalities. Here are the key findings and conclusions:

- Priority Queue Implementation: The Priority Queue class using a Max Heap successfully maintained a collection of elements with varying priority levels, with the highest-priority element (maximum value) always at the root of the heap.
- The insert and remove operations in the Priority Queue achieved a time complexity of $O(\log n)$ due to the Max Heap's heapify and bubble-up/bubble-down processes. This efficient time complexity allows for quick insertion and removal of elements based on their priorities.

Overall, the Max Heap-based Priority Queue implementation proved to be effective and efficient. It can handle a large number of elements while maintaining a balanced heap structure, allowing for quick access to the element with the highest priority. The logarithmic time complexity of insert and remove operations makes it suitable for real-world scenarios with dynamic and unpredictable priorities.

5.4 Reflection on Project

Throughout this project, I gained valuable insights into the fundamental concepts of algorithms and data structures. The hands-on implementation and analysis deepened my understanding of algorithmic efficiency and the importance of selecting appropriate algorithms based on problem requirements. Moreover, I enhanced my proficiency in utilizing data structures to build efficient and reliable systems.