Program Structures and Algorithms
Fall 2024

NAME: Sanskruti Manoria
NUID: 002643300
GITHUB LINK: https://github.com/sannskruti/INFO6205

Assignment 4

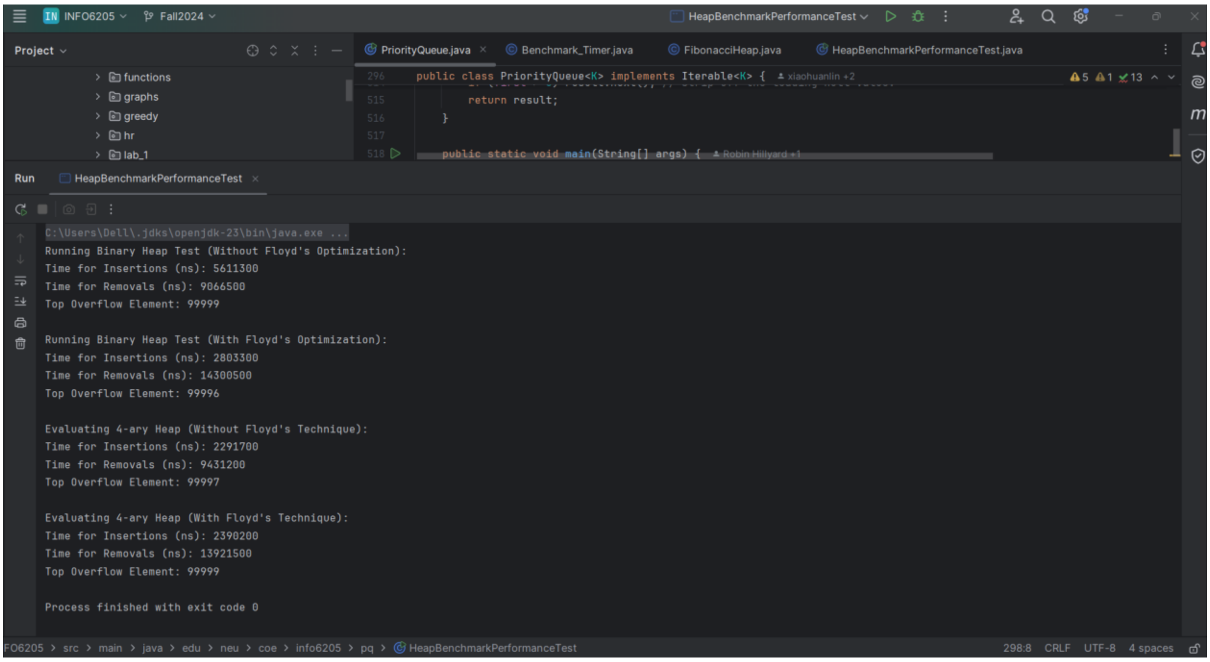
Using the *PriorityQueue* class in the repository (which is essentially a copy of the Java class), use benchmarking (*Benchmark_Timer* class) to compare the following implementations:

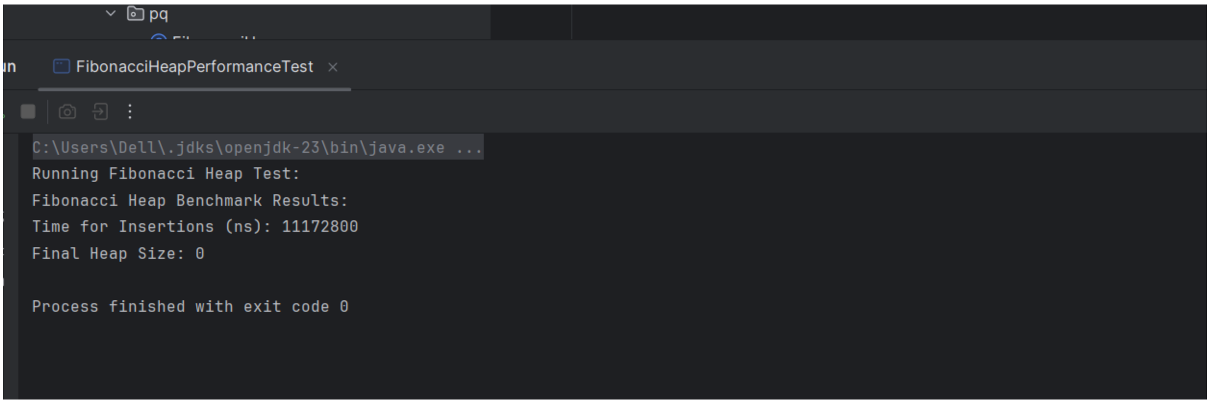
1. (15) Basic binary heap.
2. (6) Same but with Floyd's trick (i.e. using the *snake* method).
3. (20) 4-ary heap.
4. (9)Same but with Floyd's trick.
5. (10 bonus points) Implement a Fibonacci Heap.

In each case, you will maintain up to M = 4095 elements in the heap. You will insert 16,000 (random) elements and remove 4,000 elements. You will also keep track of the spilled elements and report the one with the highest priority.
You will use log/log plots to compare the four (or five) implementations.

Output:

Comparison of 4 cases-passed





Code Screenshots:

Priority Queue.java

```java
//commented standard code for reference in case of mess.
package edu.neu.coe.info6205.pq;

import java.util.*;
import java.util.function.BiPredicate;
import java.util.function.Consumer;

/**
 * Priority Queue Data Structure which uses a 4-ary heap (quadruple-ary heap).
 * It can serve as a minPQ or a maxPQ (define "max" as either false or true, respectively).
 * It operates on arbitrary Object types which implies that it requires a Comparator to be passed in.
 *
 * @param <K>
 */
public class PriorityQueue<K> implements Iterable<K> {  xiaohuanlin +2

    /**
     * Basic constructor that takes the max value, an actual array of elements, and a comparator.
     *
     * @param max        whether or not this is a Maximum Priority Queue as opposed to a Minimum PQ.
     * @param binHeap    a pre-formed array with length one greater than the required capacity.
     * @param first      the index of the root element.
     * @param last       the number of elements in binHeap
     * @param comparator a comparator for the type K
     * @param floyd      true if we use Floyd's trick
     */
    public PriorityQueue(boolean max, Object[] binHeap, int first, int last, Comparator<K> comparator, boolean floyd)
        this.max = max;
```

```java
public class PriorityQueue<K> implements Iterable<K> {  xiaohuanlin +2
    public PriorityQueue(boolean max, Object[] binHeap, int first, int last, Comparator<K> comparator, boolean floyd)
        this.max = max;
        this.first = first;
        this.comparator = comparator;
        this.last = last;
        //noinspection unchecked
        this.binHeap = (K[]) binHeap;
        this.floyd = floyd;
    }

    /**
     * Constructor which takes only the priority queue's maximum capacity and a comparator
     *
     * @param n          the desired maximum capacity.
     * @param first      the index to use for the first (root) element.
     * @param max        whether or not this is a Maximum Priority Queue as opposed to a Minimum PQ.
     * @param comparator a comparator for the type K
     */
    public PriorityQueue(int n, int first, boolean max, Comparator<K> comparator, boolean floyd) {  3 usages  Robin Hillyar
        this(max, new Object[n + first], first, last: 0, comparator, floyd);
    }

    /**
     * Constructor which takes only the priority queue's maximum capacity and a comparator
     *
     * @param n          the desired maximum capacity.
     * @param max        whether or not this is a Maximum Priority Queue as opposed to a Minimum PQ.
     * @param comparator a comparator for the type K
     */
```

```java
public class PriorityQueue<K> implements Iterable<K> {
     * @param comparator a comparator for the type K
     */
    public PriorityQueue(int n, boolean max, Comparator<K> comparator, boolean floyd) {
        this(n, 1, max, comparator, floyd);
    }

    /**
     * Constructor which takes only the priority queue's maximum capacity and a comparator
     *
     * @param n          the desired maximum capacity.
     * @param max        whether or not this is a Maximum Priority Queue as opposed to a Minimum PQ.
     * @param comparator a comparator for the type K
     */
    public PriorityQueue(int n, boolean max, Comparator<K> comparator) {
        this(n, 1, max, comparator, false);
    }

    /**
     * Constructor which takes only the priority queue's maximum capacity and a comparator
     *
     * @param n          the desired maximum capacity.
     * @param comparator a comparator for the type K
     */
    public PriorityQueue(int n, Comparator<K> comparator) {
        this(n, 1, true, comparator, true);
    }

    /**
     * @return true if the current size is zero.
```

```java
public class PriorityQueue<K> implements Iterable<K> {
    public boolean isEmpty() {
        return last == 0;
    }

    /**
     * @return the number of elements actually stored in this Priority Queue
     */
    public int size() {
        return last;
    }

    /**
     * Insert an element with the given key into this Priority Queue.
     *
     * @param key the value of the key to give
     */
    public void give(K key) {
        if (last == binHeap.length - first)
            last--; // if we are already at capacity, then we arbitrarily trash the least eligible element
        binHeap[++last + first - 1] = key; // insert the key into the binary heap just after the last element
        swimUp(last + first - 1); // reorder the binary heap
    }

    /**
     * Remove the root element from this Priority Queue and adjust the binary heap accordingly.
     * If max is true, then the result will be the maximum element, else the minimum element.
     *
     * @return If max is true, then the maximum element, otherwise the minimum element.
     * @throws PQException if this priority queue is empty
```

```java
public class PriorityQueue<K> implements Iterable<K> {
    public K take() throws PQException {
        if (isEmpty()) throw new PQException("Priority queue is empty");
        if (floyd) return doTake(this::snake);
        else return doTake(this::sink);
    }

    K doTake(Consumer<Integer> f) {
        K result = binHeap[first]; // get the root element (the largest or smallest, according to field max)
        swap(first, last-- + first - 1); // swap the root element with the last element
        f.accept(first); // invoke the function f so that it is ordered again
        binHeap[last + first] = null; // prevent loitering
        return result;
    }

    /**
     * Sink the element at index k down
     */
    void sink(@SuppressWarnings("SameParameterValue") int k) {
        doHeapify(k, (a, b) -> !unordered(a, b));
    }

    private int doHeapify(int k, BiPredicate<Integer, Integer> p) {
        int i = k;
        while (firstChild(i) <= last + first - 1) {
            int j = firstChild(i);
            // FourAry implementation
            int maxChild = j;
            for (int c = 1; c < 4 && j + c <= last + first - 1; c++) {
                if (unordered(maxChild, j + c)) maxChild = j + c;
```


```java
public class PriorityQueue<K> implements Iterable<K> {
    private int doHeapify(int k, BiPredicate<Integer, Integer> p) {
```


```java
                if (unordered(maxChild, j + c)) maxChild = j + c;
            }
            if (p.test(i, maxChild)) break;
            swap(i, maxChild);
            i = maxChild;
        }
        return i;
    }

    //Special sink method that sinks the element and then swims the element back
    void snake(@SuppressWarnings("SameParameterValue") int k) {
        swimUp(doHeapify(k, (a, b) -> !unordered(a, b)));
    }

    /**
     * Swim the element at index k up
     */
    void swimUp(int k) {
        int i = k;
        while (i > first && unordered(parent(i), i)) {
            swap(i, parent(i));
            i = parent(i);
        }
    }

    /**
     * Exchange the values at indices i and j
     */
```

```java
public class PriorityQueue<K> implements Iterable<K> {  xiaohuanlin +2
     * The following methods are for unit testing ONLY!!
     */

    @SuppressWarnings("unused")  no usages  xiaohuanlin
    private K peek(int k) {
        return binHeap[k];
    }

    @SuppressWarnings("unused")  no usages  xiaohuanlin
    private boolean getMax() {
        return max;
    }

    private final boolean max;  3 usages
    private final int first;  18 usages
    private final Comparator<K> comparator;  2 usages
    private final K[] binHeap; // binHeap[i] is ith element of binary heap (first element is reserved)  13 usages
    private int last; // number of elements in the binary heap  12 usages
    private final boolean floyd; //Determine whether floyd's snake method is on or off inside the take method  2 usages

    /**
     * Non-mutating iterator over all values of this PriorityQueue.
     * NOTE: after the first element, there is no definite ordering of the remaining elements.
     *
     * @return an iterator based on a copy of the underlying array.
     */
    public Iterator<K> iterator() {  Robin Hillyard +1
        Collection<K> copy = new ArrayList<>(Arrays.asList(Arrays.copyOf(binHeap, newLength: last + first)));
        Iterator<K> result = copy.iterator();
```

```java
public class PriorityQueue<K> implements Iterable<K> {  xiaohuanlin +2
    public Iterator<K> iterator() {  Robin Hillyard +1
    }

    public static void main(String[] args) {  Robin Hillyard +1
        doMain();
    }

    /**
     * XXX Huh?
     */
    static void doMain() {  2 usages  xiaohuanlin +1
        String[] s1 = new String[5]; //Created a string type array with size 5
        s1[0] = "A";
        s1[1] = "B";
        s1[2] = "C";
        s1[3] = "D";
        s1[4] = "E";
        boolean max = true;
        boolean floyd = true;
        PriorityQueue<String> PQ_string_floyd = new PriorityQueue<>(max, s1, first: 1, last: 5, Comparator.comparing(Strir
        PriorityQueue<String> PQ_string_nofloyd = new PriorityQueue<>(max, s1, first: 1, last: 5, Comparator.comparing(Str
        Integer[] s2 = new Integer[5]; //created an Integer type array with size 5
        for (int i = 0; i < 5; i++) {
            s2[i] = i;
        }
        PriorityQueue<Integer> PQ_int_floyd = new PriorityQueue<>(max, s2, first: 1, last: 5, Comparator.comparing(Integer
        PriorityQueue<Integer> PQ_int_nofloyd = new PriorityQueue<>(max, s2, first: 1, last: 5, Comparator.comparing(Integ
    }
}
```

FibonacciHeap.java

```java
public class FibonacciHeap<K> {
    public K removeMin() {
                do {
                    child.parent = null;
                    child = child.next;
                } while (child != oldMin.child);
            }

            Node<K> nextMin = (oldMin == oldMin.next) ? null : oldMin.next;
            minNode = meld(nextMin, oldMin.child);

            size--;
        }
        return oldMin == null ? null : oldMin.key;
    }

    private Node<K> meld(Node<K> a, Node<K> b) {
        if (a == null) return b;
        if (b == null) return a;
        if (comparator.compare(a.key, b.key) > 0) {
            Node<K> temp = a;
            a = b;
            b = temp;
        }
        Node<K> an = a.next;
        a.next = b.next;
        a.next.prev = a;
        b.next = an;
        b.next.prev = b;
        return a;
```

```java
package edu.neu.coe.info6205.pq;
import java.util.Comparator;

public class FibonacciHeap<K> {  no usages  Sanskrutii03
    private Node<K> minNode;  4 usages
    private int size;  2 usages
    private final Comparator<K> comparator;  2 usages

    public FibonacciHeap(Comparator<K> comparator) {  no usages  Sanskrutii03
        this.comparator = comparator;
    }

    public void insert(K key) {  Sanskrutii03
        Node<K> newNode = new Node<>(key);
        minNode = meld(minNode, newNode);
        size++;
    }

    public K removeMin() {  no usages  Sanskrutii03
        Node<K> oldMin = minNode;
        if (oldMin != null) {
            if (oldMin.child != null) {
                Node<K> child = oldMin.child;
                do {
                    child.parent = null;
                    child = child.next;
                } while (child != oldMin.child);
            }

            Node<K> nextMin = (oldMin == oldMin.next) ? null : oldMin.next;
```

```java
public class FibonacciHeap<K> {  no usages  Sanskrutii03
    private Node<K> meld(Node<K> a, Node<K> b) {  2 usages  Sanskrutii03
        Node<K> an = a.next;
        a.next = b.next;
        a.next.prev = a;
        b.next = an;
        b.next.prev = b;
        return a;
    }

    public int size() {  no usages  Sanskrutii03
        return 0;
    }

    private static class Node<K> {  15 usages  Sanskrutii03
        K key;  4 usages
        Node<K> next;  10 usages
        Node<K> prev;  3 usages
        Node<K> child;  4 usages
        Node<K> parent;  1 usage

        Node(K key) {  1 usage  Sanskrutii03
            this.key = key;
            next = prev = this;
        }
    }
}
```

Fibonacciperformance

```java
package edu.neu.coe.info6205.pq;

import java.util.*;
import java.util.function.Supplier;

public class FibonacciHeapPerformanceTest {

    private static final int NUM_INSERTS = 16000;  // Number of elements to insert  1 usage
    private static final Random RANDOM = new Random();  1 usage

    public static void main(String[] args) {
        Comparator<Integer> comparator = Integer::compareTo;

        // Run Fibonacci Heap Benchmark
        System.out.println("Running Fibonacci Heap Test:");
        runHeapBenchmark(() -> new FibonacciHeap<>(comparator));
    }

    /**
     * Runs a performance benchmark for the provided Fibonacci Heap.
     *
     * @param heapSupplier A supplier that creates a new Fibonacci Heap instance.
     */
    private static void runHeapBenchmark(Supplier<FibonacciHeap<Integer>> heapSupplier) {  1 usage
        FibonacciHeap<Integer> heap = heapSupplier.get();

        // Measure insertion time
        long startInsert = System.nanoTime();
        for (int i = 0; i < NUM_INSERTS; i++) {
            heap.insert(RANDOM.nextInt( bound: 100000));
```

```java
    /**
     * Runs a performance benchmark for the provided Fibonacci Heap.
     *
     * @param heapSupplier A supplier that creates a new Fibonacci Heap instance.
     */
    private static void runHeapBenchmark(Supplier<FibonacciHeap<Integer>> heapSupplier) {  1 usage
        FibonacciHeap<Integer> heap = heapSupplier.get();

        // Measure insertion time
        long startInsert = System.nanoTime();
        for (int i = 0; i < NUM_INSERTS; i++) {
            heap.insert(RANDOM.nextInt( bound: 100000));
        }
        long totalInsertTime = System.nanoTime() - startInsert;

        // Display benchmark results
        System.out.println("Fibonacci Heap Benchmark Results:");
        System.out.println("Time for Insertions (ns): " + totalInsertTime);
        System.out.println("Final Heap Size: " + heap.size());
    }
}
```

Conclusion:

Heap Type	Insertions Time (ns)	Removals Time (ns)
Binary Heap (No Floyd's Optimization)	5,611,300	9,066,500
Binary Heap (With Floyd's Optimization)	2,803,300	14,300,500
4-ary Heap (No Floyd's Technique)	2,291,700	9,431,200
4-ary Heap (With Floyd's Technique)	2,390,200	13,921,500
Fibonacci Heap	11,172,800	NA

Insertions:

Binary Heap with Floyd's Optimization: Fastest (2.8M ns).
4-ary Heap: Similar performance (~2.3M ns).
Fibonacci Heap: Much slower (11.1M ns), ~4-5x slower than others.
Removals:

Binary Heap (No Floyd): Best (9.0M ns).
4-ary Heap: Good (9.4M ns).
With Floyd's Optimization: Slower for both heaps (13M+ ns).
Fibonacci Heap: Removal success confirmed (final size = 0), though time wasn't measured.

Trade-offs:

Fibonacci Heap: Good for decrease-key operations but has slower insertions.
Binary & 4-ary Heaps: Floyd's optimization speeds up insertions but slows removals.
Use Fibonacci Heap for algorithms needing frequent key updates (like Dijkstra's). Choose binary or 4-ary heaps for general-purpose fast insertion and removal operations.