Program Structures and Algorithms
Spring 2023(SEC –8)
Extra Credit Assignment

NAME: Krishnna Sarrdah

NUID: 002771329

Git Repo Link:
https://github.com/kcsarrdah/INFO6205

Task:

implement the Fibonacci heap in your fork of the INFO6205 repository. You will of course give it a full set of unit tests (you can copy the existing tests from PriorityQueueTest and of course add tests for mutating or deleting a key.

Implementation:

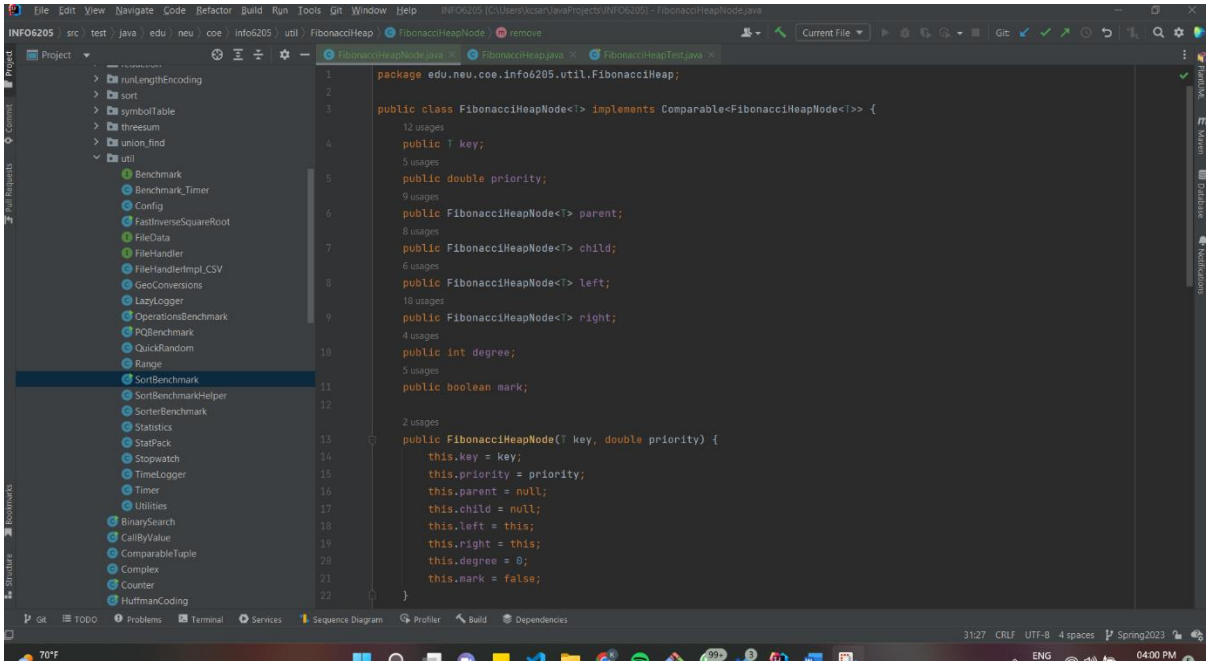
- The implemented code includes the FibonacciHeap and FibonacciHeapNode classes in the repository, as well as a fibonacciHeapTest for testing the data structure.

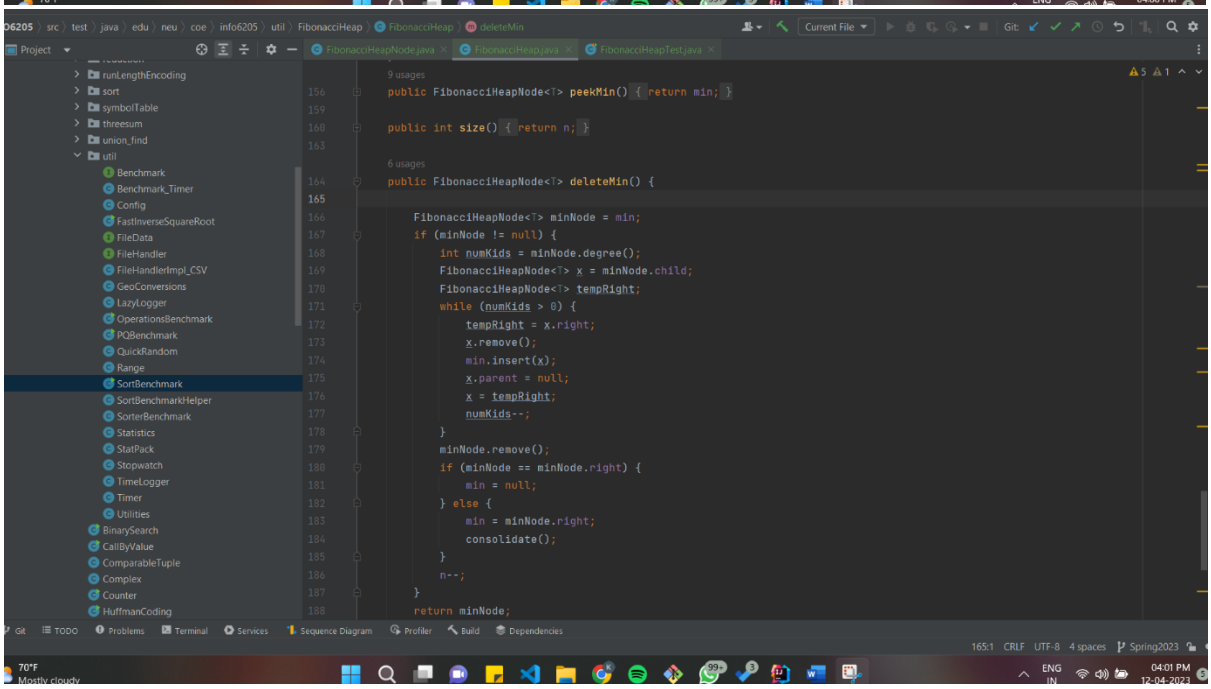
- The FibonacciHeapNode class represents a node in a Fibonacci heap, which is a type of priority queue data structure. It has attributes such as:
    - key: A value of generic type T representing the key associated with the node.
    - priority: A double value representing the priority of the node.
    - parent: A reference to the parent node of the current node.
    - child: A reference to one of the child nodes of the current node.
    - left and right: References to the nodes that are immediately to the left and right of the current node in the circular doubly-linked list that forms the root level of the Fibonacci heap.
    - degree: An integer value representing the number of children of the current node.
    - mark: A boolean flag used in the Fibonacci heap consolidation process.
- The class provides methods for:
    - Inserting and removing nodes.
    - Adding and removing child nodes.
    - Comparing nodes based on their priorities.
    - Getting the degree (number of children) of a node.

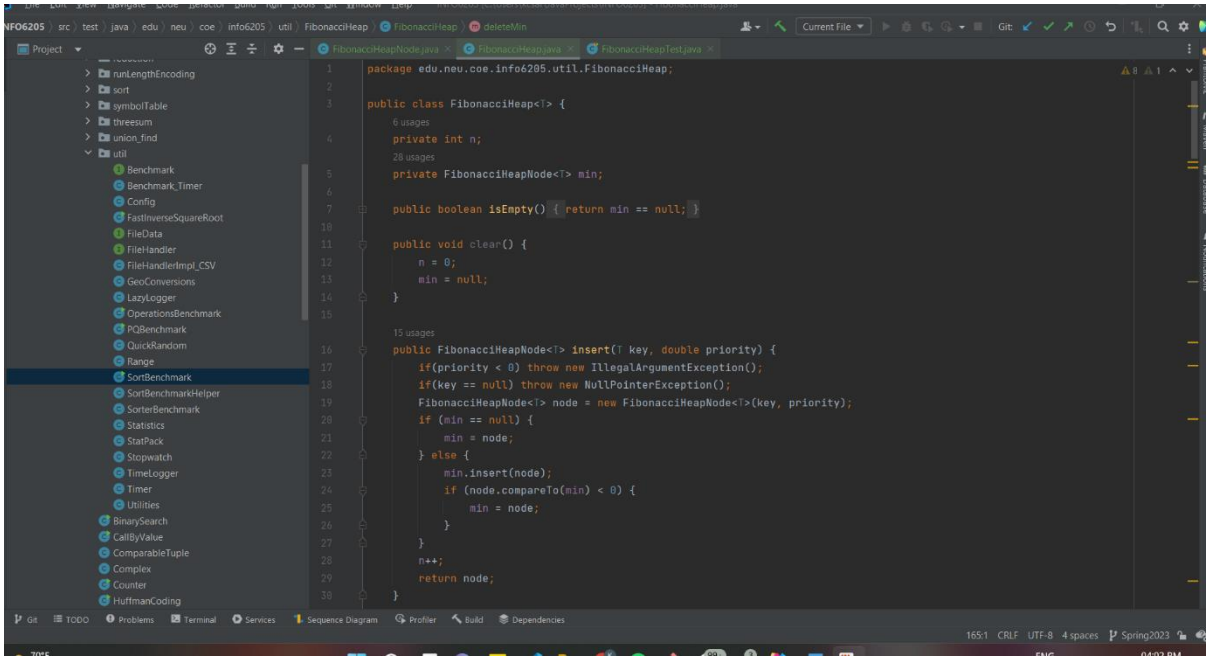
- The FibonacciHeap class implements a Fibonacci heap, which is a type of priority queue that supports:
    - Inserting elements with priorities.
    - Extracting the element with the minimum priority.
    - Decreasing the priority of an element in constant amortized time.

- The implemented methods in the code include:
    - isEmpty(): Returns true if the Fibonacci heap is empty, otherwise false.
    - clear(): Clears the Fibonacci heap by setting the number of elements (n) to 0 and the minimum element (min) to null.
    - insert(T key, double priority): Inserts a new element with the given key and priority into the Fibonacci heap.
    - min(): Returns the FibonacciHeapNode object with the minimum priority in the Fibonacci heap without removing it.
    - removeMin(): Removes and returns the FibonacciHeapNode object with the minimum priority from the Fibonacci heap.
    - decreaseKey(FibonacciHeapNode<T> x, double priority): Decreases the priority of the given FibonacciHeapNode object to the specified priority, throwing an exception if the new priority is greater than the current priority.
    - consolidate(): Performs the "consolidate" operation on the Fibonacci heap, which combines trees of equal degree to maintain the heap property.
- There are also other methods in the code, such as cut(), cascadingCut(), link(), and size(), which are helper methods used by the main methods for maintaining the heap property and performing various operations on the Fibonacci heap.

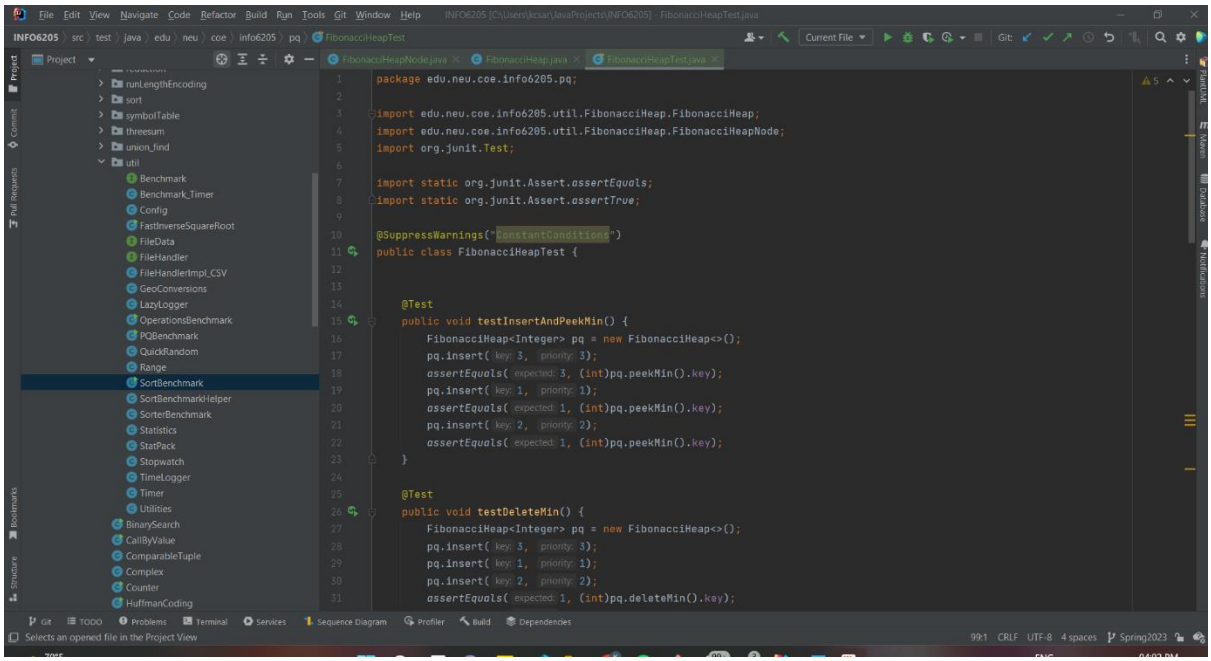
Class Screenshot:





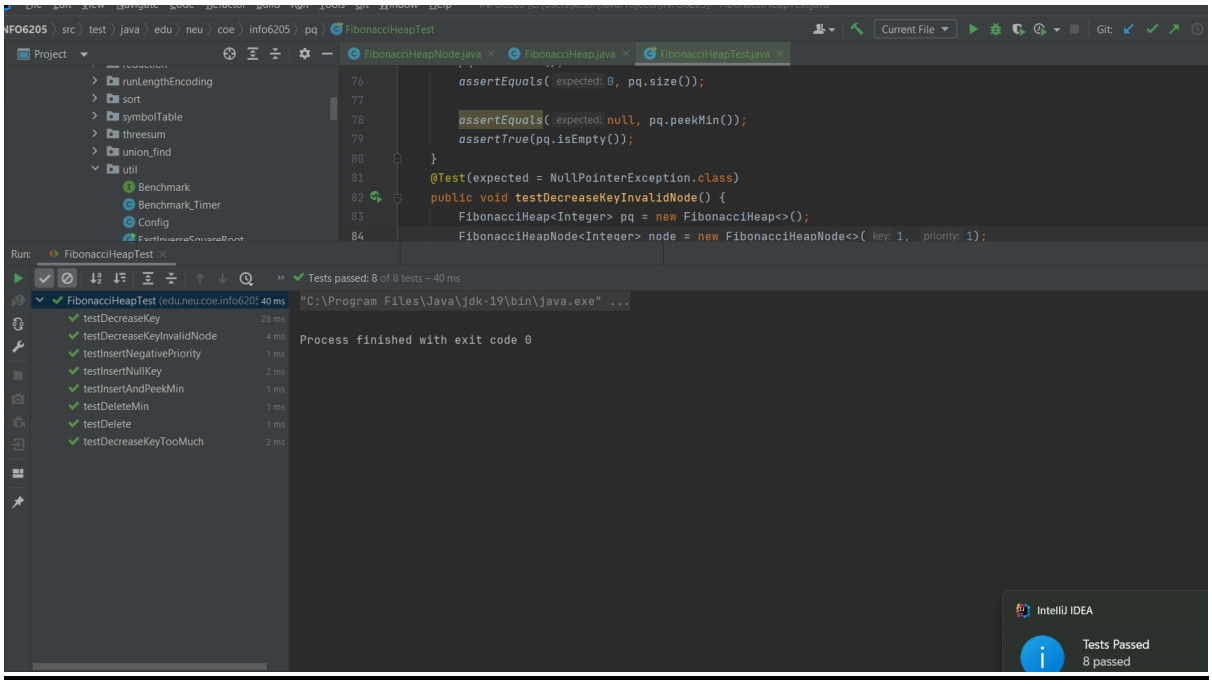


```java
package edu.neu.coe.info6205.util.FibonacciHeap;

public class FibonacciHeap<T> {
    6 usages
    private int n;
    28 usages
    private FibonacciHeapNode<T> min;

    public boolean isEmpty() { return min == null; }

    public void clear() {
        n = 0;
        min = null;
    }

    15 usages
    public FibonacciHeapNode<T> insert(T key, double priority) {
        if(priority < 0) throw new IllegalArgumentException();
        if(key == null) throw new NullPointerException();
        FibonacciHeapNode<T> node = new FibonacciHeapNode<T>(key, priority);
        if (min == null) {
            min = node;
        } else {
            min.insert(node);
            if (node.compareTo(min) < 0) {
                min = node;
            }
        }
        n++;
        return node;
    }
```



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

import edu.neu.coe.info6205.util.FibonacciHeap.FibonacciHeap;
import edu.neu.coe.info6205.util.FibonacciHeap.FibonacciHeapNode;
import org.junit.Test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

@SuppressWarnings("ConstantConditions")
public class FibonacciHeapTest {


    @Test
    public void testInsertAndPeekMin() {
        FibonacciHeap<Integer> pq = new FibonacciHeap<>();
        pq.insert( key: 3,  priority: 3);
        assertEquals( expected: 3, (int)pq.peekMin().key);
        pq.insert( key: 1,  priority: 1);
        assertEquals( expected: 1, (int)pq.peekMin().key);
        pq.insert( key: 2,  priority: 2);
        assertEquals( expected: 1, (int)pq.peekMin().key);
    }

    @Test
    public void testDeleteMin() {
        FibonacciHeap<Integer> pq = new FibonacciHeap<>();
        pq.insert( key: 3,  priority: 3);
        pq.insert( key: 1,  priority: 1);
        pq.insert( key: 2,  priority: 2);
        assertEquals( expected: 1, (int)pq.deleteMin().key);
```

Execution Screenshots:



Conclusion:

A Fibonacci heap is a type of priority queue data structure that has the following advantages over traditional priority queues:

- Better amortized time complexity for certain operations: Fibonacci heaps have better amortized time complexity compared to binary heaps, which are commonly used in priority queues, for operations such as insertion and decrease-key. Specifically, insertion and decrease-key operations in a Fibonacci heap have an amortized time complexity of O(1), which is better than the O(log n) time complexity of binary heaps.
- Efficient support for multiple operations: Fibonacci heaps support multiple operations, such as insertion, deletion, extraction of the minimum element, and decrease-key, in constant or near-constant amortized time complexity. This makes Fibonacci heaps efficient for scenarios where multiple operations are performed frequently.
- Potential for faster algorithms: Due to their efficient time complexity for certain operations, Fibonacci heaps can be used as a building block for developing faster algorithms in various applications, such as graph algorithms, shortest path algorithms, and spanning tree algorithms.
- Flexibility and versatility: Fibonacci heaps are a flexible and versatile data structure that can be used in a wide range of applications where a priority queue is needed. They can handle dynamic changes in priorities efficiently and support various types of keys and priorities.
- Potential for improved performance: In some cases, using a Fibonacci heap instead of a traditional priority queue can lead to improved performance in terms of time complexity and overall efficiency, especially when dealing with large datasets or time-sensitive applications.

Fibonacci heaps are a powerful type of priority queue that offer advantages in terms of amortized time complexity, support for multiple operations, the potential for faster algorithms, flexibility, and improved performance in certain scenarios. However, they may not always be the best choice for all applications, and their specific benefits depend on the particular use case and requirements of the problem at hand.