GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING

CSE 222
HOMEWORK 6 REPORT

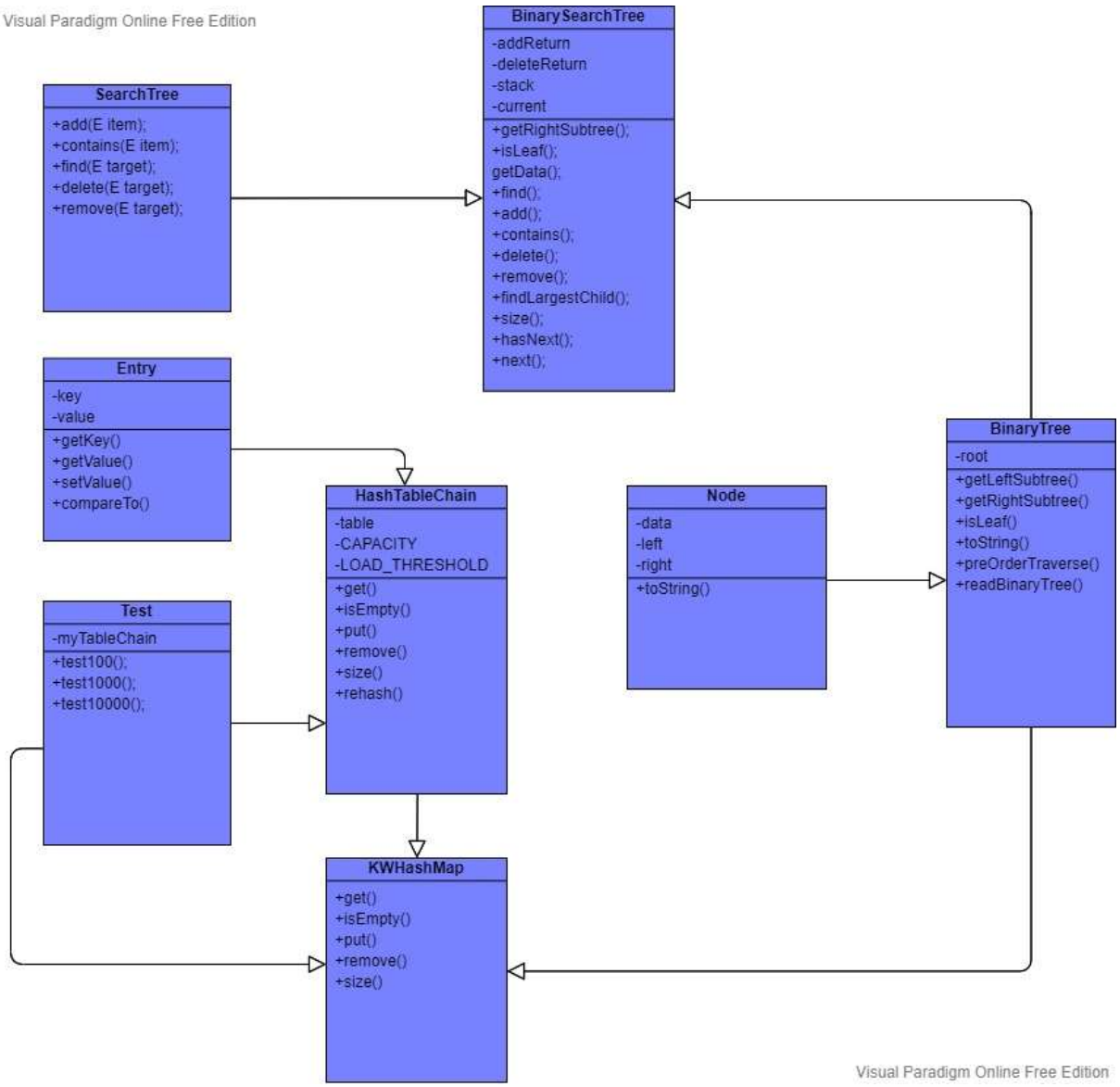
ERAY ALÇİN
1801042687

1. SYSTEM REQUIREMENTS

As a system, we should know the hashing technique and the chaining technique. At the same time, we need to know because we will use binary search tree while using these techniques. We should know and research coalesced hashing and double hashing methods. Advantages and disadvantages We should show these studies in our report.

For the second question, we need to know the mergesort algorithm and the quicksort algorithm and we are expected to implement them.

2. CLASS DIAGRAM



3. PROBLEM SOLUTION APPROACH

As a result, we implemented hashmap. We have seen the advantages and disadvantages of chaining and hashing techniques. We saw the working hours. We implemented these techniques using the binary search tree. We remembered the binary search tree and saw the use of binary search tree in hashmap.

Double Hashing

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

- **Advantages of Double hashing**

The advantage of Double hashing is that it is one of the best form of probing, producing a uniform distribution of records throughout a hash table.

- ✓ This technique does not yield any clusters.
- ✓ It is one of effective method for resolving collisions.

Coalesced Hashing

Coalesced Hashing is approach takes advantage of two different collision resolution techniques to handle collision in a hash table, an open addressing and chaining. It uses similar insertion procedure as open addressing to insert an element in the hash table using h(k) mod n. When a collision occurs at i position in the hash table, coalesced hashing resolve it similar to separate chaining by inserting the key that causes the collision in the first empty slot from the bottom of the hash table i.e. for (i ¡ n, i ¡=0, i–) where i is an index and n is the size of the hash table. It then chains the colliding key original hash value to hash value the colliding key is inserted using a pointer instead of creating a linked list like separated chaining. It minimizes space usage but constant time lookup is not achievable at a high load factor like an open addressing and separate chaining. Any open addressing method can be used to identify a position to insert a key that collides in coalesced hashing.

- **Advantages:**

1) Simple to implement.

2) Hash table never fills up, we can always add more elements to the chain.

3) Less sensitive to the hash function or load factors.

4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

- **Disadvantages:**

1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.

2) Wastage of Space (Some Parts of hash table are never used)

3) If the chain becomes long, then search time can become O(n) in the worst case.

4) Uses extra space for links.

- **Performance of Chaining:**

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

4. TEST, RUNNING COMMAND AND RESULTS

Q1

Small

```
private void test100(){

        myTableChain= new HashTableChain<> ();
        System.out.println ("\t***TEST With Map Size = 100***");
        for (int i = 0 ; i<60;i++) System.out.print("-");
        for (int j = 0; j < 1; j++) {
            //execute each table and  print the times.
            //-------------------------------------------------------------------------

            for (int i = 0 ; i<60;i++) System.out.print("-");
            System.out.println ("\n\t***HashTableChain with 100 element***");
            startTime = System.currentTimeMillis();
            for (int i = 0; i < 1000; i++)
            {
                //put key=ith element,value=random number
                //values are random number
                myTableChain.put (i,random.nextInt(1000));
            }
            System.out.println ("85th key of open table: "+myTableChain.get (85));
            System.out.println ("14th key of open table: "+myTableChain.get (14));
            endTime = System.currentTimeMillis();
            System.out.println("HashTableChain put 100 element and get 2 element time: "
                    + (endTime - startTime) + " ms");

            System.out.println ();

        }
    }
```

Medium

```
private void test1000(){

        myTableChain= new HashTableChain<> ();
        System.out.println ("\t***TEST With Map Size = 1000***");
        for (int i = 0 ; i<60;i++) System.out.print("-");


        for (int j = 0; j < 1; j++) {
            //execute each table and  print the times.
            for (int i = 0 ; i<60;i++) System.out.print("-");
            System.out.println ("\n\t***HashTableChain with 1000 element***");
            startTime = System.currentTimeMillis();
            for (int i = 0; i < 1000; i++)
            {
                //put key=ith element,value=random number
                //values are random number
                myTableChain.put (i,random.nextInt(1000));
            }
            System.out.println ("650th key of open table: "+myTableChain.get (650));
            System.out.println ("900th key of open table: "+myTableChain.get (900));
            endTime = System.currentTimeMillis();
            System.out.println("HashTableChain put 1000 element and get 2 element time: "
                    + (endTime - startTime) + " ms");
            System.out.println ();
        }
}
```

Large

```
private void test10000(){

        myTableChain= new HashTableChain<> ();
        System.out.println ("\t***TEST With Map Size = 10000***");
        for (int i = 0 ; i<60;i++) System.out.print("-");

        for (int j = 0; j < 1; j++) {
            //execute each table and  print the times.
            for (int i = 0 ; i<60;i++) System.out.print("-");
            System.out.println ("\n\t***HashTableChain with 100000 element***");
            startTime = System.currentTimeMillis();
            for (int i = 0; i < 1000; i++)
            {
                //put key=ith element,value=random number
                //values are random number
                myTableChain.put (i,random.nextInt(1000));
            }
            System.out.println ("650th key of open table: "+myTableChain.get (650));
            System.out.println ("900th key of open table: "+myTableChain.get (900));
            endTime = System.currentTimeMillis();
            System.out.println("HashTableChain put 10000 element and get 2 element time: "
                    + (endTime - startTime) + " ms");

            System.out.println ();

        }
}
```

TESTING

```
        ***TEST With Map Size = 100***
-----------------------------------------------------------------------------------------------------------------------------------
        ***HashTableChain with 100 element***
85th key of open table: 698
14th key of open table: 67
HashTableChain put 100 element and get 2 element time: 5 ms

        ***TEST With Map Size = 1000***
-----------------------------------------------------------------------------------------------------------------------------------
        ***HashTableChain with 1000 element***
650th key of open table: 973
900th key of open table: 652
HashTableChain put 1000 element and get 2 element time: 3 ms

        ***TEST With Map Size = 10000***
-----------------------------------------------------------------------------------------------------------------------------------
        ***HashTableChain with 100000 element***
650th key of open table: 475
900th key of open table: 499
HashTableChain put 10000 element and get 2 element time: 1 ms
```

Q2

Small

```
/**
 * Test each algorithm with arrays with small size(100).
 */
private void test100(){

        System.out.println ("\t***ARRAY/LIST SIZE = 100***");
        for (int i = 0 ; i<60;i++) System.out.print("-");

        for (int j = 0; j < 1000; j++) {

                getRandomArray (100);//fill array ,arrayCopy
        }
            for (int i = 0 ; i<60;i++) System.out.print("-");
            System.out.println ("\n\t***Merge Sort***");
            startTime = System.currentTimeMillis();
            MergeSort.sort (array);
            endTime = System.currentTimeMillis();
            System.out.println(" Book Merge sort time is " + (endTime-startTime) + " ms");
            System.out.println(" Book Merge sort is successful (true/false): "+ verifyArray (array));
            reloadArray (array,arrayCopy);
            //-------------------------------------------------------------------------

            for (int i = 0 ; i<60;i++) System.out.print("-");
            System.out.println ("\n\t***Quick Sort***");
            startTime = System.currentTimeMillis();
            QuickSort.sort (array);
            endTime = System.currentTimeMillis();
            System.out.println(" Book Quick sort time is " + (endTime-startTime) + " ms");
            System.out.println(" Book Quick sort is successful (true/false): "+ verifyArray (array));
            reloadArray (array,arrayCopy);

}
```

Medium

```
/**
 * Test each algorithm with arrays with medium size(1000).
 */
private void test1000(){

    System.out.println ("\t***ARRAY/LIST SIZE = 1000***");
    for (int i = 0 ; i<60;i++) System.out.print("-");

    for (int j = 0; j < 1000; j++) {

            getRandomArray (1000);//fill array ,arrayCopy, list and listCopy as random.

    }
        for (int i = 0 ; i<60;i++) System.out.print("-");
        System.out.println ("\n\t***Merge Sort***");
        startTime = System.currentTimeMillis();
        MergeSort.sort (array);
        endTime = System.currentTimeMillis();
        System.out.println(" Book Merge sort time is " + (endTime-startTime) + " ms");
        System.out.println(" Book Merge sort is successful (true/false): "+ verifyArray (array));
        reloadArray (array,arrayCopy);
        //-------------------------------------------------------------------------

        for (int i = 0 ; i<60;i++) System.out.print("-");
        System.out.println ("\n\t***Quick Sort***");
        startTime = System.currentTimeMillis();
        QuickSort.sort (array);
        endTime = System.currentTimeMillis();
        System.out.println(" Book Quick sort time is " + (endTime-startTime) + " ms");
        System.out.println(" Book Quick sort is successful (true/false): "+ verifyArray (array));
        reloadArray (array,arrayCopy);
    }
```

Large

```
/**
 * Test each algorithm with arrays with small size(10000).
 */
private void test10000(){

    System.out.println ("\t***ARRAY/LIST SIZE = 10000***");
    for (int i = 0 ; i<60;i++) System.out.print("-");

    for (int j = 0; j < 1000; j++) {

            getRandomArray (10000);//fill array ,arrayCopy, list and listCopy as random.

    }
        for (int i = 0 ; i<60;i++) System.out.print("-");
        System.out.println ("\n\t***Merge Sort***");
        startTime = System.currentTimeMillis();
        MergeSort.sort (array);
        endTime = System.currentTimeMillis();
        System.out.println(" Book Merge sort time is " + (endTime-startTime) + " ms");
        System.out.println(" Book Merge sort is successful (true/false): "+ verifyArray (array));
        reloadArray (array,arrayCopy);
        //-------------------------------------------------------------------------

        for (int i = 0 ; i<60;i++) System.out.print("-");
        System.out.println ("\n\t***Quick Sort***");
        startTime = System.currentTimeMillis();
        QuickSort.sort (array);
        endTime = System.currentTimeMillis();
        System.out.println(" Book Quick sort time is " + (endTime-startTime) + " ms");
        System.out.println(" Book Quick sort is successful (true/false): "+ verifyArray (array));
        reloadArray (array,arrayCopy);

    }
```

TESTING

```
       ***ARRAY/LIST SIZE = 100***
------------------------------------------------------------------------------------------------------------------------
       ***Merge Sort***
 Book Merge sort time is 0 ms
 Book Merge sort is successful (true/false): true
------------------------------------------------------------
       ***Quick Sort***
 Book Quick sort time is 5 ms
 Book Quick sort is successful (true/false): true
       ***ARRAY/LIST SIZE = 1000***
------------------------------------------------------------------------------------------------------------------------
       ***Merge Sort***
 Book Merge sort time is 1 ms
 Book Merge sort is successful (true/false): true
------------------------------------------------------------
       ***Quick Sort***
 Book Quick sort time is 1 ms
 Book Quick sort is successful (true/false): true
       ***ARRAY/LIST SIZE = 10000***
------------------------------------------------------------------------------------------------------------------------
       ***Merge Sort***
 Book Merge sort time is 1 ms
 Book Merge sort is successful (true/false): true
------------------------------------------------------------
       ***Quick Sort***
 Book Quick sort time is 2 ms
 Book Quick sort is successful (true/false): true
```