GTU Department of Computer Engineering

CSE 222 / 505 – Spring 2022

Homework 6

Ahmet USLUOGLU

1801042602

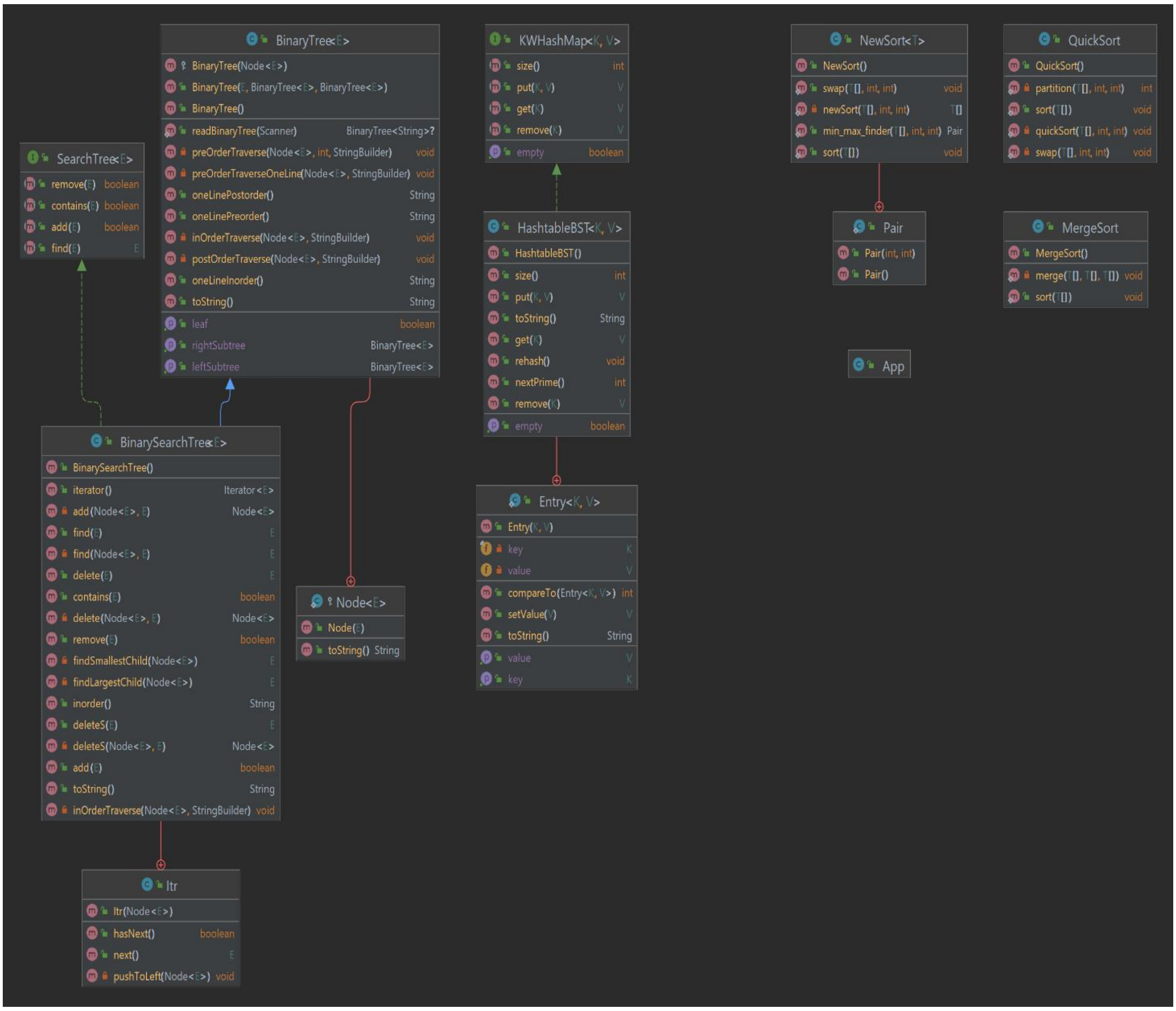
1 – System Requirement

Operating System must have JDK (Java Development Kit) 11 and JRE (Java RuntimeEnvironment) 11 or higher.

There should be enough space for storing data’s.

2 – Class Diagrams

*Higher Resolution Version of the Class Diagram is in the files.



3. Problem - Solution Approach

Problem:

1 -) Implement the chaining technique for hashing. U binary search trees to chain items as underlying structure mapped on the same table slot.

2-) Compare the sorting algorithms Quick Sort, Merge Sort and New Sort both empirically and theoretically.

Solution:

1-) When implementing a hash table using chaining technique if the items stack over one index more than the others, searching for an item gets longer and it approaches to linear O(n) complexity. To prevent this instead of using linked list as underlying structure , using Binary search trees makes searching time an average of O(nlogn). Thus, gives us faster average but more complex space structure.

2-) Tested 3 sort algorithms by 100 time each with 100, 1000 and 10000 elements.

4 – Test Cases

a-) Testing the hash table for part1

1 – Adding 100 elements to the hash table and measuring time.

2 – Check the isEmpty with empty hash table.

3 – Check the isEmpty with filled hash table.

4 – Remove element from hashtable.

5 – Try removing unexisting element.

6 – Add an element with same key value to update it.

b-) Testing the sort algorithms for part2

1 -) Compare 3 algorithms 100 times with 100 sized arrays.

2 -) Compare 3 algorithms 100 times with 1000 sized arrays.

3 -) Compare 3 algorithms 100 times with 10000 sized arrays.

5 – Running Program and Results

a) Testing the hash table for part1

1 – Adding 100 elements to the hash table and measuring time

```
Added 100 elements to hashtable = 12 ms
```

2 – Check the isEmpty with empty hash table.

```
Creating new empty hashtable
This hashtable is empty ==> true
```

3 – Check the isEmpty with filled hash table.

```
Added one to hashtable
one
This hashtable is empty ==> false
```

4 – Remove element from hashtable.

```
Removed one from hashtable
This hashtable is empty ==> true
```

5 – Try removing unexisting element.

```
Trying to remove unexisting element from table
```

6 – Add an element with same key value to update it.

```
Added one to six to hashtable
one two three four five six

Added eight and nine to hashtable as 5 and 6 keys to update five and six
one two three four eight nine
```

b-) Testing the sort algorithms for part2

1 -) Compare 3 algorithms 100 times with 100 sized arrays.

```
100 x 100
Merge Sort =5 ms
Quick Sort =7 ms
New Sort   =15 ms
```

2 -) Compare 3 algorithms 100 times with 1000 sized arrays.

```
100 x 1000
Merge Sort =49 ms
Quick Sort =14 ms
New Sort   =174 ms
```

3 -) Compare 3 algorithms 100 times with 10000 sized arrays.

```
100 x 10000
Merge Sort =169 ms
Quick Sort =102 ms
New Sort   =16390 ms
```

6 – Calculate Time complexity

--Example Hash Table functions implementations

```java
/**
     * If the key is in the table, remove it and return its value. Otherwise, return null
     *
     * @param key The key of the entry to remove.
     * @return The value of the key that was removed.
     */
    public V remove(K key)
    {
        int index = Math.abs(key.hashCode()) % CAPACITY;
        if (table[index] == null) {return null;}

        var entry = table[index].find(new Entry<>(key, null));

        if (entry == null)  {return null;}

        V oldValue = entry.getValue();
        table[index].remove(entry);
        numKeys--;
        return oldValue;
    }
```

```java
public void rehash()
    {
        BinarySearchTree<Entry<K, V>>[] prevTable = table;

        int NEWCAPACITY = nextPrime();
        numKeys = 0;
        table = new BinarySearchTree[NEWCAPACITY];

        for (var bst : prevTable)
        {
            if(prevTable != null)
            {
                for (var element : bst)
                {
                    put(element.key,element.value);
                }
            }
        }
    }
```

```java
@Override
    public V put(K key, V value)
    {
        int index = key.hashCode() % table.length;

        if (index < 0)
            index += table.length;

        if (table[index] == null)
        {
            // Create a new linked list at table[index].
            table[index] = new BinarySearchTree<Entry<K,V>>();
        }

        // Search the list at table[index] to find the key.
        for (Entry<K, V> nextItem : table[index])
        {
            // If the search is successful, replace the old value.
            if (nextItem.getKey().equals(key))
            {
                // Replace value for this key.
                V oldVal = nextItem.getValue();
                nextItem.setValue(value);
                return oldVal;
            }
        }

        // assert: key is not in the table, add new item.
        table[index].add(new Entry<>(key, value));
        numKeys++;

        if (numKeys > (LOAD_THRESHOLD * table.length))
            rehash();

        return null;
    }
```

-- Hash table searching complexity average = O(logn).

-- Hash table searching complexity worst case = O(n).

-- Hash table searching complexity best case = O(1).

--Example Merge Sort functions implementations

```java
public static <T extends Comparable<T>> void sort(T[] table)
    {
        // A table with one element is sorted already.
        if (table.length > 1)
        {
            // Split table into halves.
            int halfSize = table.length / 2;
            T[] leftTable = (T[]) new Comparable[halfSize];
            T[] rightTable = (T[]) new Comparable[table.length - halfSize];
            System.arraycopy(table, 0, leftTable, 0, halfSize);
            System.arraycopy(table, halfSize, rightTable, 0, table.length - halfSize);
            // Sort the halves.
            sort(leftTable);
            sort(rightTable);
            // Merge the halves.
            merge(table, leftTable, rightTable);
        }
    }
```

-- Merge Sort Average Complexity = O(nlogn)
-- Merge Sort Worst Case Complexity = O(nlogn)
-- Merge Sort Best Case Complexity = O(nlogn)

```
100 x 100
Merge Sort =5 ms
Quick Sort =7 ms
New Sort   =15 ms

100 x 1000
Merge Sort =49 ms
Quick Sort =14 ms
New Sort   =174 ms

100 x 10000
Merge Sort =165 ms
Quick Sort =107 ms
New Sort   =16405 ms
```

--Example Quick Sort functions implementations

```java
private static <T extends Comparable<T>> void quickSort(T[] table, int first, int last) {
        if (first < last) { // There is data to be sorted.
            // Partition the table.
            int pivIndex = partition(table, first, last);
            // Sort the left half.
            quickSort(table, first, pivIndex - 1);
            // Sort the right half.
            quickSort(table, pivIndex + 1, last);
        }
    }
```

-- Quick Sort Average Complexity = O(nlogn)
-- Quick Sort Worst Case Complexity = O(n^2)
-- Quick Sort Best Case Complexity = O(nlogn)

```
100 x 100
Merge Sort =5 ms
Quick Sort =7 ms
New Sort   =15 ms

100 x 1000
Merge Sort =49 ms
Quick Sort =14 ms
New Sort   =174 ms

100 x 10000
Merge Sort =165 ms
Quick Sort =107 ms
New Sort   =16405 ms
```

-- Example Quick Sort functions implementations

```java
private static <T extends Comparable<T>> T[] newSort(T[] table, int head, int tail)
    {
        if(head > tail)
            return table;
        else
        {
            Pair min_max = min_max_finder(table,head, tail);

            swap(table, tail, min_max.max);
            min_max = min_max_finder(table,head, tail);
            swap(table, head, min_max.min);
            return newSort(table, head+1,tail-1);
        }
    }
```

-- New Sort Average Complexity = O(n^2)

-- New Sort Worst Case Complexity = O(n^2)

-- New Sort Best Case Complexity = O(n^2)

```
100 x 100
Merge Sort =5 ms
Quick Sort =7 ms
New Sort   =15 ms

100 x 1000
Merge Sort =49 ms
Quick Sort =14 ms
New Sort   =174 ms

100 x 10000
Merge Sort =165 ms
Quick Sort =107 ms
New Sort   =16405 ms
```