GIT Department of Computer Engineering CSE 222/505 - Spring 2022
Homework # Report

Sefa Çiçek
1801042657

1. **System Requirements**

- jdk and jre are requested from operating system to execute this java program.

-User has to run makefiles for folder

2. Problem Solution Approach

Q1

Part1:

I have implemented hash table using BST . I’ve wrote BST Iterator class for traversing BST. On Iterator class, I used stack for traversing. On size and isEmpty methods, I checked number of keys. On put and remove methods, I used friend method(findKey) to find key index using hashcode. I wrote preOrderTraverse function for traversing. On rehash function, I traversed indices of table with iterator to transfer key-value pairs to the new hash table.

Part2:

- Coalesced hashing is very efficient algorithm if the chains of indices are short. It uses the principle of open addressing to find empty place. Disadvantages of this algorithm is deletion is hard because many items are linked. We must observe this linking.

- Double hashing is reducing the clustering with using a formula to find empty space for key-value pair. It makes the job easier by using prime numbers. Disadvantage of double hashing is the performance decreases when table fills up.

I implemented KWHashMap methods for Hybrid Hashing. I have a table which holds Entries. On Entry class, I added ‘next’ property for holding next entry to chain items. I wrote findKey and searchKey methods for remove and get operations. On put function, I called h1 and h2 functions to calculate position of key value. Also, to find largest prime number, I wrote a iterative method.

3. Test Cases and Running Command and Results

```
isEmpty : true
CAPACITY: 10
LOAD_THRESHOLD: 0.75

Insert 3:
----------
(3, 1)
    null
    null


Insert 12:
----------
(12, 2)
    null
    null
(3, 1)
    null
    null


Insert 13:
----------
(12, 2)
    null
    null
(3, 1)
    null
    (13, 3)
        null
        null
```

```
Insert 25:
----------
(12, 2)
    null
    null
(3, 1)
    null
    (13, 3)
        null
        null
(25, 4)
    null
    null


Insert 23:
----------
(12, 2)
    null
    null
(3, 1)
    null
    (13, 3)
        null
        (23, 5)
            null
            null
(25, 4)
    null
    null
```

```
Insert 51:
----------
(51, 6)
    null
    null
(12, 2)
    null
    null
(3, 1)
    null
    (13, 3)
        null
        (23, 5)
            null
            null
(25, 4)
    null
    null


Delete 25:
----------
(51, 6)
    null
    null
(12, 2)
    null
    null
(3, 1)
    null
    (13, 3)
        null
        (23, 5)
            null
            null

isEmpty : false
```

```
*****************************************
           Q1-P2 HYBRID HASHING
*****************************************

isEmpty : true

Insert 3:
----------
Hash Value     Key    Next
    0                 null
    1                 null
    2                 null
    3                 null
    4                 null
    5                 null
    6                 null
    7           3    null
    8                 null
    9                 null


Insert 12:
----------
Hash Value     Key    Next
    0                 null
    1                 null
    2                 null
    3                 null
    4           12    null
    5                 null
    6                 null
    7           3    null
    8                 null
    9                 null
```

```
Insert 13:
----------
Hash Value     Key    Next
    0                 null
    1                 null
    2                 null
    3                 null
    4           12      5
    5           13    null
    6                 null
    7           3    null
    8                 null
    9                 null


Insert 25:
----------
Hash Value     Key    Next
    0                 null
    1                 null
    2                 null
    3                 null
    4           12      5
    5           13    null
    6                 null
    7           3    null
    8           25    null
    9                 null
```

```
Insert 23:
----------
Hash Value     Key    Next
    0                 null
    1                 null
    2                 null
    3           23    null
    4           12      5
    5           13    null
    6                 null
    7           3    null
    8           25      3
    9                 null


Insert 51:
----------
Hash Value     Key    Next
    0                 null
    1                 null
    2                 null
    3           23    null
    4           12      5
    5           13    null
    6           51    null
    7           3    null
    8           25      3
    9                 null
```

```
Delete 25:
----------
Hash Value     Key    Next
    0                 null
    1                 null
    2                 null
    3                 null
    4           12      5
    5           13    null
    6           51    null
    7           3    null
    8           23    null
    9                 null
```

Q1

PART3:

```
--------------
100 element
--------------
Is HashTableChain table empty: true
Is HybridHashing table empty: true

Removing NOT exist elements in HashTableChain: 0.03745093000000001
Removing NOT exist elements in HybridHashing: 0.0013250499999999986

Getting NOT exist elements in HashTableChain: 0.0010539700000000024
Getting NOT exist elements in HybridHashing: 3.5400000000000085E-4

Adding operation in HashTableChain:: 0.007337010000000004
Adding operation in HybridHashing: 0.014483060000000002

Getting elements in HashTableChain: 0.0023501300000000006
Getting elements in HybridHashing: 0.001060980000000003

Removing elements in HashTableChain: 0.003512019999999997
Removing elements in HybridHashing: 0.003978990000000026


--------------
1000 element
--------------
Removing NOT exist elements in HashTableChain: 5.371909999999995E-4
Removing NOT exist elements in HybridHashing: 0.0012607080000000032

Getting NOT exist elements in HashTableChain: 3.274389999999773E-4
Getting NOT exist elements in HybridHashing: 2.127049999999969E-4

Adding operation in HashTableChain:: 0.0027650929999999984
Adding operation in HybridHashing: 0.08905908999999995

Getting elements in HashTableChain: 3.4440299999999465E-4
Getting elements in HybridHashing: 9.940600000000114E-5

Removing elements in HashTableChain: 6.896169999999911E-4
Removing elements in HybridHashing: 0.002586301000000043
```

```
--------------
10000 element
--------------
Removing NOT exist elements in HashTableChain: 1.108301999999934E-4
Removing NOT exist elements in HybridHashing: 0.00163166369999999

Getting NOT exist elements in HashTableChain: 1.0033139999999526E-4
Getting NOT exist elements in HybridHashing: 1.2294849999999182E-4

Adding operation in HashTableChain:: 9.261270000000129E-4
Adding operation in HybridHashing: 0.171146850599999

Getting elements in HashTableChain: 8.937829999999683E-5
Getting elements in HybridHashing: 4.869139999999946E-5

Removing elements in HashTableChain: 1.5211289999999378E-4
Removing elements in HybridHashing: 0.017505738200000102
```

Conclusion:

On smaller sizes, hashTableChain is efficient. Getting elements in hybrid hashing is better than hashTableChain. Putting and removing is efficient on hashTableChain if collision is less.

Q2

```
SMALL SIZE: 100
----------------
Average running time of Merge Sort: 0.025660287000000001
Average running time of Quick Sort: 0.018552902000000232
Average running time of New Sort: 0.041760183000000076

MEDIUM SIZE: 1000
----------------
Average running time of Merge Sort: 0.0942213159999999
Average running time of Quick Sort: 0.5296206719999984
Average running time of New Sort: 1.4340973920000017

LARGE SIZE: 10000
----------------
Average running time of Merge Sort: 0.7903475290000004
Average running time of Quick Sort: 51.78818396100002
Average running time of New Sort: 136.44382580000004
```

```java
*/
private static <T extends Comparable<T>> MinMax min_max_finder(T[] arr, int low, int high) {

    MinMax minmax = new MinMax();
    MinMax minMaxLeft = new MinMax();
    MinMax minMaxRight = new MinMax();
    int mid;

    // There is only one element
    if (low == high) {
        minmax.max = low;
        minmax.min = low;
        return minmax;
    }

    // There are two elements
    if (high - low == 1) {
        if (arr[low].compareTo(arr[high]) == 1) {
            minmax.max = low;
            minmax.min = high;
        } else {
            minmax.max = high;
            minmax.min = low;
        }
        return minmax;
    }

    // There are more than 2 elements
    mid = (low + high) / 2;
    minMaxLeft = min_max_finder(arr, low, mid);
    minMaxRight = min_max_finder(arr, mid + 1, high);

    if (arr[minMaxLeft.min].compareTo(arr[minMaxRight.min]) == -1)
        minmax.min = minMaxLeft.min;
    else
        minmax.min = minMaxRight.min;

    if (arr[minMaxLeft.max].compareTo(arr[minMaxRight.max]) == 1)
        minmax.max = minMaxLeft.max;
    else
        minmax.max = minMaxRight.max;

    return minmax;
}
```

$\Theta(1)$

$\Theta(1)$

$\Theta(1)$

$T(n/2)$

$T(n/2)$

$2T(n/2)$

$\Theta(1)$

$$T(n) = 2T(n/2) + 1 \rightarrow 2^k(T(\overbrace{n/2^k}^{k=\log(n)})) + \overbrace{2^{k-1} + \dots + 2^0}^{2^k - 1} = \Theta(n)$$

$T(n) \overset{?}{\leq} 2^n$

$T(n) \leq 2T(n-1) + cn$

$\leq 2^{n-1} + 2^{n-1} + cn$

$= 2^n + cn$

```java
public static <T extends Comparable<T>> T[] new_sort(T[] array, int head, int tail) {

    if (head > tail)
        return array;
    else {
        MinMax obj = min_max_finder(array, head, tail);
        swap(array, head, obj.min);
        swap(array, tail, obj.max);
        return new_sort(array, head + 1, tail - 1);
    }
}
```

$\Theta(1)$ $\Theta(n)$ $\Theta(1)$ $\rightarrow T(n-2)$

$T(n) = 2T(n-2) + n = \Theta(2^n)$

```java
private static <T extends Comparable<T>> void swap(T[] array, int first, int second) {
    T temp = array[first];
    array[first] = array[second];
    array[second] = temp;
}
```

$\Theta(1)$ $\rightarrow \Theta(1)$

MergeSort time comp:

$$T(n) = 2T(n/2) + \theta(n)$$

$T(n/2) = 2T(n/4) + n/2$

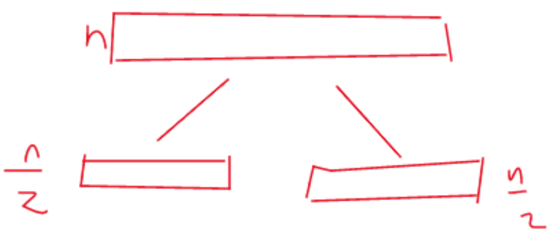
$T(n/4) = 2T(n/8) + n/4$

$\vdots$

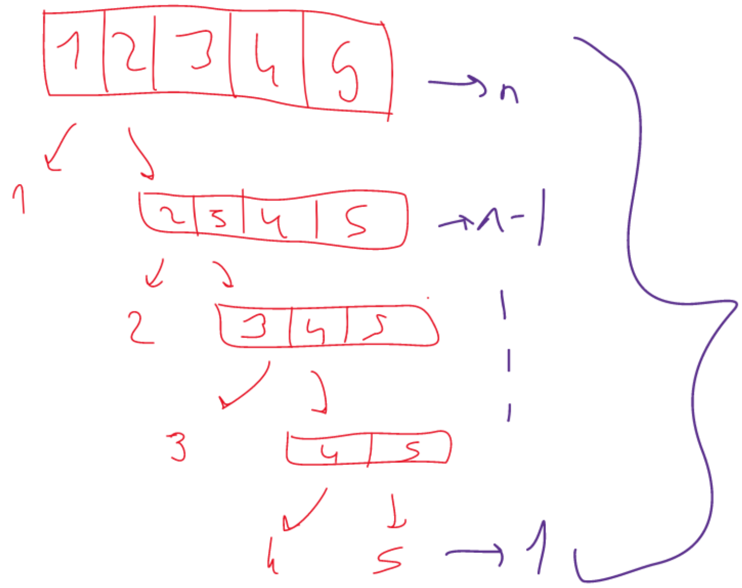
$T(n) = 2^k T(\overbrace{n/2^k}^{n}) + kn = 2^k + kn = \Theta(n\log(n))$

$k = \log_2 n$

$n = 2^k$



QuickSort time comp:



$$n+(n-1)+\dots 1=\frac{n(n-1)}{2}$$

$$\Downarrow$$

$$O(n^2)$$

CONCLUSİON:

- Merge sort is more efficient than quick sort on larger array sizes.
  Quick sort is more efficient merge sort on smaller array sizes.
- Merge sort requires additional memory space to store arrays.
  Quick sort doesn’t require any additional storage.
- Merge sort is efficient than quicksort for sorting linked lists.(partitition is easy)
  Quick sort is efficient than quicksort for sorting arrays.