**Gebze Technical University**

**Computer Engineering**

**CSE 222 - 2018 Spring**

**HOMEWORK 5 REPORT**

**CENGİZ TOPRAK**

**161044087**

Course Assistant:

**FATMA NUR ESİRCİ**

# Double Hashing Map

This part about Question1 in HW5

## Pseudocode and Explanation

Code of get method:

public Object get(Object key) {

int index = key.hashCode();

int temp;

if (index < 0) index \*= -1;

index %= table.length;

temp = index;

while (true) {

if (table[index] != null && !table[index].getState()

&& table[index].getKey().equals(key))

return table[index].getValue();

index = (index + 1)%table.length;

if (index == temp) return null;

}

}

Pseudocode for get method:

Object get(Object key):

1-) Set index to key.hashCode()

2-) if index < 0

multiply index to mines 1

3-)set temp to index

4-) if table[index] != null and table[index] != deleted and key == table[index].getKey() do

return table[index].getValue()

5-) Set index to (index + 1) % length of table

6-) if index == temp do

return null

7-) repeat 4-6 steps

Pseudocode for get mehod:

Object put(Object key, Object value):

1-) Decleare variables index,newIndex,hash2,i and initialize them.

2-) Set index to key.hashCode.

3-) if index < 0 do

multiply index to mines 1

4-) Set index to (index + 1)% size of table

5-) call IsRequiredRehash() function check return value.

6-) if return value == true do

call rehash()

7-) if table[index].getKey() == null do

create a new entry that has key and value

set table[index] to this created entry

increase numbers of items in the table by one and

return table[index].getValue()

8-) else if table[index].getKey() == key do

update value by setting new value and return old value.

9-) call doubleHash(index) and assign return value of it to hash2 variable.

10-) Set newIndex to(index + i\*hash2)% size of table

11-) if table[newIndex].getKey() == null do

create a new entry that has key and value

set table[newindex] to this created entry

increase numbers of items in the table by one and

get out loop using break keyword.

12-) increase i by one

13-) repeat 10-12 steps

14-) return value of table[newIndex].getValue()

IsRequiredRehash():---> Check wheather is required rehash or not

rehash():--->increase size of table to proper size and insert all elements into table properly.

doubleHash(index):--->decrease collisions by second hashing.

Code of put method:

public Object put(Object key, Object value) {

int index = key.hashCode();

int hash2,i = 0;

int newIndex = 0;

if (index < 0) index \*= -1;

index = index % table.length;

System.*out*.printf("Index : %d Key : %s\n",index,key.toString());

if (IsRequiredRehash()) {

rehash();

}

if (table[index] == null) {

table[index]=new Entry(key,value,false);

++numbersOfItems;

return table[index].getValue();

}

else if (table[index].getKey().equals(key)){

Object oldValue = table[index].getValue();

table[index].setValue(value);

return oldValue;

}

hash2 = doubleHash(index);

while (true) {

newIndex = (index + i\*hash2)%table.length;

if (table[newIndex] == null) {

table[newIndex] = new Entry(key,value,false);

++numbersOfItems;

break;

}

++i;

}

return table[newIndex].getValue();

}

Pseudocode for remove method:

Object remove(Object key):

1-) Decleare variable index,temp and initialize them.

2-) Set index to key.hashCode()

3-) if index is negative do

multiply index to mines one

4-) Set index to index by modula length of table.

5-) Set temp to index.

6-) if key == table[index].getKey() do

decleare variable value

set value to table[index].getValue()

set table[index] to deleted

increase number of deleteted items by one

decrease number of undeleted items by one

return taken value

7-) Set index to (index plus one) by modula length of table

8-) if index == temp do (All table iterate and item is not found)

return null

9-) if table[index] != null and table[index] != deleted and

key == table[index].getKey() do

decleare variable value

set value to table[index].getValue()

set table[index] to deleted

increase number of deleteted items by one

decrease number of undeleted items by one

return taken value

10-) repeat 7-9 steps

## Code for remove method:

public Object remove(Object key) {

int index = key.hashCode();

int temp;

if (index < 0) index \*= -1;

index %= table.length;

temp = index;

if (table[index].getKey().equals(key)) {

Object value = table[index].getValue();

table[index].setState(true);

++numberOfDeletedItems;

--numbersOfItems;

return value;

}

while (true) {

if ((index = (index + 1) % table.length) == temp) return null;

if (table[index] != null && !table[index].getState()

&& table[index].getKey().equals(key)) {

Object value = table[index].getValue();

table[index].setState(true);

++numberOfDeletedItems;

--numbersOfItems;

return value;

}

}

}

In implementation of double hashing map used java map interface and enrty class from course textbook.It is implemented just remove,add,get,size,Isempty and containsKey methods from map interface

## 

|  |  |  |
| --- | --- | --- |
| 0 | K:Jennefer V: | collusion occurs(normal index=8) hash2 = doubleHash(index) = 11  doubleHash(index)=prime – (index%prime) where prime=19  formula for each collusion index=(index + i\*hash2)%19  where i initialize zero and increase value of it by one when each collusion occurs in a while loop  index=(8 +0\*11)%19 = 8 collusion occurs , increase i by one  index=(8 +1\*11)%19 = 0 no collusion add key to table[0] break loop. |
| 1 | K:Victory V: | key:Victory 1=index=key.hashCode()%19(jack added 1.inedexed cell) |
| 2 | K:Baron V: | key:Baron 2=index=key.hashCode()%19(jack added 2.inedexed cell) |
| 3 |  |  |
| 4 |  |  |
| 5 | K:Richard V: | collusion occurs(normal index 14) perform operation like 0.indexed table cell. |
| 6 |  |  |
| 7 |  |  |
| 8 | K:Jack V: | key:Jack 8=index = key.hashCode()%19(jack added 8.inedexed cell) |
| 9 | K:Elisabet V: | collusion occurs(normal index 10) perform operation like 0.indexed table cell. |
| 10 | K:Teresa V: | key:Teresa 10=index=key.hashCode()%19(Teresa added 10.inedexed cell) |
| 11 |  |  |
| 12 | K:George V: | key:George 12=index=key.hashCode()%19(George added 12.inedexed cell) |
| 13 |  |  |
| 14 | K:Hector V: | key:George 14=index=key.hashCode()%19(George added 14.inedexed cell) |
| 15 | K:Pillow V: | collusion occurs(normal index 14) perform operation like 0.indexed table cell. |
| 16 | K:David V: | key:David 16=index = key.hashCode()%19(David added 16.inedexed cell) |
| 17 | K:Edward V: | collusion occurs(normal index 1) perform operation like 0.indexed table cell. |
| 18 | K:Frank V: | collusion occurs(normal index 1) perform operation like 0.indexed table cell. |

Check whetaher rehash is required or not.

LOAD FACTOR:0.65

table size : 19

number of items in the table :13

13/19 = 0.68

since it exceeds load factor it is needed to rehash table.

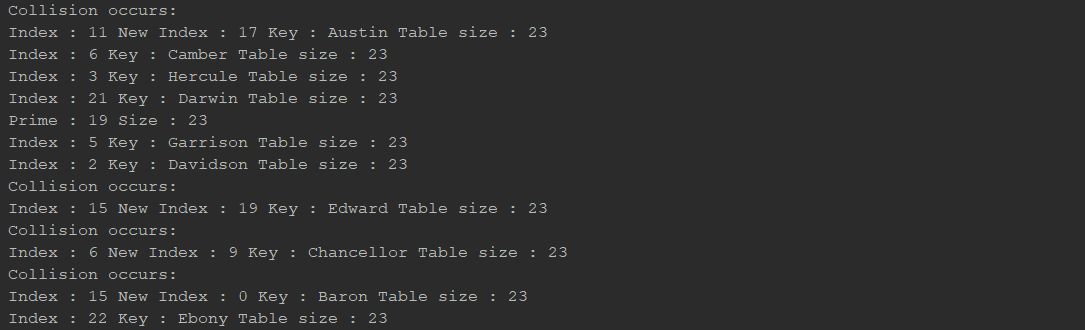
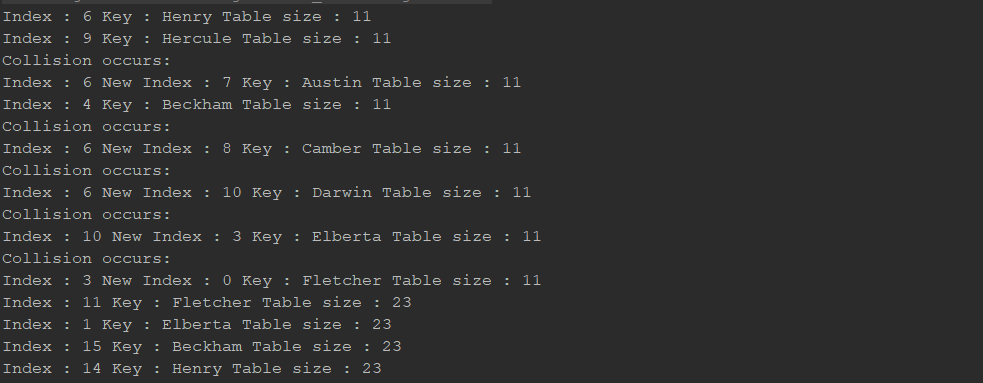
It is grown table size to a number which this number is smaller prime number that is bigger than 2\* sizeoftable + 1.

new size >= 2\*19 + 1>=39

new size = 41

After that all elements in the old table is added to new table properly.

# **second table**



|  |  |  |
| --- | --- | --- |
| 0 | K:Fletcher V: | collusion occurs(normal index 3) perform operation like 7.indexed table cell |
| 1 |  |  |
| 2 |  |  |
| 3 | K:Elberta V: | collusion occurs(normal index 10) perform operation like 7.indexed table cell |
| 4 | K:Beckham V: | key: Beckham 4 = index = key.hashCode()%11( Beckham added 4.inedexed cell) |
| 5 |  |  |
| 6 | K:Henry V: | key:Henry 6 = index = key.hashCode()%11(Henry added 6.inedexed cell) |
| 7 | K:Austin V: | collusion occurs(normal index = 6) hash2 = doubleHash(index) = 1  doubleHash(index)=prime – (index%prime) where prime=7  formula for each collusion index=(index + i\*hash2)%11  where i initialize zero and increase value of it by one when each collusion occurs in a while loop  index=(6 +0\*1)%11 = 6 collusion occurs , increase i by one  index=(6 +1\*1)%11 = 7 no collusion add key to table[0] break loop. |
| 8 | K:Camber V: | collusion occurs(normal index 6) perform operation like 7.indexed table cell |
| 9 | K:Hercule V: | key:Hercule 9 = index = key.hashCode()%11(Hercule added 9.inedexed cell) |
| 10 | K:Darwin V: | collusion occurs(normal index 6) perform operation like 7.indexed table cell |

Check whetaher rehash is required or not.

LOAD FACTOR:0.65

table size : 11

number of items in the table :8

8/11 = 0.72

since it exceeds load factor it is needed to rehash table.

It is grown table size to a number which this number is smaller prime number that is bigger than 2\* sizeoftable + 1.

new size >= 2\*11 + 1>=23

new size = 23

After that all elements in the old table is added to new table properly.

# Recursive Hashing Set

## Pseudocode and Explanation

Code for contains method:

public boolean contains(Object o) {

E obj = (E)o;

int hashcode = obj.hashCode();

if (hashcode < 0) hashcode \*= -1;

return containsHelper(table,hashcode,obj);

}

## Pseudocode for contains method:

boolean contains(Object o):

1-) cast object to type E and declare variable ojb and set obj to casted value

2-) decleare hashcode and set to obj.hashCode()

3-) if hashcode < 0 do

make hashcode positive

4-) call containsHelper parameters with table,hashcode,obj and

return value of calling function.

## Code for contains method:

private boolean containsHelper(Node<E>[] array,int hashcode,E target) {

if (array == null) return false;

int index = hashcode % array.length;

if (array[index] != null && array[index].getElement().equals(target))

return true;

return containsHelper(array[index].getTable(),hashcode,target);

}

Pseudocode for containsHelper :

boolean containsHelper(Node<E>[] array,int hashcode,E target) {

1-) if array == null do

return false

2-) decleare variable index and set index to hashcode % size of array

3-) if array[index] != null and array[index].getElement() == target do

return true

4-) call containsHelper parameters with table.getTable(),hashcode,obj and

return value of calling function.

## Pseudocode for add method:

public boolean add(Object o) {

1-) cast object to type E and declare variable ojb and set obj to casted value

2-) decleare hashcode and set to obj.hashCode()

3-) if hashcode < 0 do

make hashcode positive

4-) decleare variable index and set index to hashcode % length of table;

5-) call addHelper parameters with table,table.length,index,hashcode,obj and

return value of calling function.

## Code for add method:

boolean add(Object o) {

E obj = (E)o;

int hashcode = obj.hashCode();

if (hashcode < 0) hashcode \*= -1;

int index = hashcode%table.length;

return addHelper(table,table.length,index,hashcode,obj);

}

## Code for addHelper method:

private boolean addHelper(Node<E>[] array,int length,int originIndex,int hashcode,E target) {

int index;

index = hashcode % array.length;

if (array[index] != null && array[index].getElement().equals(target))

return false;

else if (array[index] == null) {

array[index] = new Node<>(target);

++numberOfItems;

return true;

}

else if (array[index] != null && array[index].getTable() != null) {

return addHelper(array[index].getTable(),array.length,originIndex,hashcode,target);

}

else if (array[index].getTable() == null) {

if (length < 1) array[index].createTable(1);

else array[index].createTable(findPrime(length - 1,0));

int newIndex = target.hashCode()%array[index].getTable().length;

if ( newIndex < 0) newIndex \*= -1;

array[index].getTable()[newIndex] = new Node<>(target);

++numberOfItems;

return true;

}

return false;

}

PseudoCode for addHelper method:

private boolean addHelper(Node<E>[] array,int length,int originIndex,int hashcode,E target) {

1-) Decleare variable index;

2-) Set index to hashcode % size of array

3-) if array[index] != null and array[index].getElement() == target do

return false;

4-) else if array[index] == null do

array[index] <-- new Node<>(target);

increase numberOfItems by one

return true;

5-) else if array[index] != null and array[index].getTable() != null do

call addHelper parameters with array[index].getTable(),array.length originIndex,hashcode,target and return value of called function.

6-) else if array[index].getTable() == null do

if length < 1 do

array[index].createTable(1);

else do

array[index].createTable(findPrime(length – 1,0));

decleare variable newIndex

newIndex <-- target.hashCode()%array[index].getTable().length;

if newIndex < 0 do

newIndex <-- newIndex \* -1;

array[index].getTable()[newIndex] <-- new Node<>(target);

increase numberOfItems by one

return true;

7-) return false;

## Code for remove method:

public boolean remove(Object o) {

E obj = (E)o;

int hashcode = obj.hashCode();

if (hashcode < 0) hashcode \*= -1;

return removeHelper(table,hashcode,obj);

}

Pseudocode for remove method:

boolean remove(Object o):

1-) cast object to type E and declare variable ojb and set obj to casted value

2-) decleare hashcode and set to obj.hashCode()

3-) if hashcode < 0 do

make hashcode positive

5-) call removeHelper parameters with table,hashcode,obj and

return value of calling function.

## Code for removeHelper method:

private boolean removeHelper(Node<E>[] array,int hashcode,E target) {

if (array == null) return false;

int index = hashcode % array.length;

if (array[index] != null && array[index].getElement().equals(target)) {

array[index] = null;

--numberOfItems;

return true;

}

if (array[index] != null)

return removeHelper(array[index].getTable(),hashcode,target);

return false;

}

Pseudocode for removeHelper:

private boolean removeHelper(Node<E>[] array,int hashcode,E target):

1-) if array == null do

return false;

2-) decleare variable index

3-) index <-- hashcode % array.length;

4-) if array[index] != null and array[index].getElement() == target do

array[index] <-- null

decrease numberOfItems by one

return true

5-) if array[index] != null do

call removeHelper parameters with table.getTable(),hashcode,target and

return value of calling function.

6-) return false

In implementation of recursive hashing set is used java set interface and generic class that has generic item and an array of same type of Node class course.It is implemented just remove,add,size,Isempty,contains methods from set interface and some useful helper methods named removeHelper,addHelper and contiansHelper

## Test Cases

Original index : 2 original table size : 11 index : 2 item : Jack table size : 11

Original index : 6 original table size : 11 index : 6 item : George table size : 11

Original index : 7 original table size : 11 index : 7 item : Victory table size : 11

Original index : 1 original table size : 11 index : 1 item : Jennifer table size : 11

Original index : 6 original table size : 11 index : 4 item : Hector table size : 7

Original index : 10 original table size : 11 index : 10 item : Richard table size : 11

Original index : 1 original table size : 11 index : 1 item : Teresa table size : 7

Original index : 0 original table size : 11 index : 0 item : Pillow table size : 11

Original index : 10 original table size : 11 index : 6 item : Frank table size : 7

Original index : 3 original table size : 11 index : 3 item : David table size : 11

Original index : 9 original table size : 11 index : 9 item : Edward table size : 11

Original index : 5 original table size : 11 index : 5 item : Elisabet table size : 11

Original index : 8 original table size : 11 index : 8 item : Baron table size : 11

Original index : 6 original table size : 11 index : 3 item : Alexis table size : 7

Original index : 8 original table size : 11 index : 3 item : Caroline table size : 7

Original index : 7 original table size : 11 index : 6 item : Jacky table size : 7

Original index : 1 original table size : 11 index : 6 item : Banner table size : 7

Original index : 7 original table size : 11 index : 4 item : Culver table size : 7

Original index : 0 original table size : 11 index : 6 item : Denver table size : 7

Original index : 4 original table size : 11 index : 4 item : Eleanor table size : 11

Original index : 2 original table size : 11 index : 2 item : Forrest table size : 7

Original index : 8 original table size : 11 index : 2 item : Grayson table size : 7

Original index : 8 original table size : 11 index : 4 item : Harley table size : 7

Original index : 9 original table size : 11 index : 2 item : Hawthorne table size : 7

Original index : 7 original table size : 11 index : 0 item : Jefferson table size : 7

Original index : 2 original table size : 11 index : 5 item : Keaton table size : 7

Original index : 10 original table size : 11 index : 6 item : Kimberley table size : 7

Original index : 7 original table size : 11 index : 0 item : Perry table size : 7

As seen above when each collusion occurs table size decrease.

# Sorting Algortihms

## MergeSort with DoubleLinkedList

This part about Question3 in HW5

### Pseudocode and Explanation

Code of sort method:

public static void sort(LinkedList<Integer> arr, int left, int right) {

if (left < right)

{

// Find the middle point

int middlePoint = (left+right)/2;

// Sort first and second halves

*sort*(arr, left, middlePoint);

*sort*(arr , middlePoint+1, right);

// Merge the sorted halves

*merge*(arr, left, middlePoint, right);

}

}

Pseudocode for sort method:

sort(LinkedList<Integer> arr, int left, int right):

1-) if left < right do

decleare variable middlepoint

Set middlepoint to left plus right by over two

call the itself with parameters ,in turn,arr,left,middlepoint

call the itself with parameters ,in turn,arr,middlepoint + 1,right

call *merge function with parameters,in turn,arr,left,middlepoint,right*

*2-) else do nothing*

### Code for merge method:

public static void merge (LinkedList<Integer> arr, int left, int middlePoint, int right)

{

int n1 = middlePoint - left + 1;

int n2 = right - middlePoint;

LinkedList<Integer> Left = new LinkedList<>();

LinkedList<Integer> Right = new LinkedList<>();

for (int i=0; i<n1; ++i)

Left.add(arr.get(left + i));

for (int j=0; j<n2; ++j)

Right.add(arr.get(middlePoint + 1 + j));

int i = 0, j = 0;

int k = left;

while (i < n1 && j < n2)

{

if (Left.get(i).compareTo(Right.get(j)) <= 0) {

arr.set(k,Left.get(i));

i++;

}

else {

arr.set(k,Right.get(j));

j++;

}

k++;

}

while (i < n1)

{

arr.set(k,Left.get(i));

i++;

k++;

}

while (j < n2)

{

arr.set(k,Right.get(j));

j++;

k++;

}

}

Pseudocode for merge method:

merge (LinkedList<Integer> arr, int left, int middlePoint, int right) :

1-) decleare n1 and n2 variables

2-) Set n1 to middlePoint - left + 1 and n2 to right – middlePoint

3-) decleare two LinkedList of Integer named left and right.

4-) allocate memory for both of them.

5-) for (int i=0; i<n1; ++i) do

add arr.get(left + i) item to Left linkedList

6-) for (int j=0; j<n2; ++j) do

add arr.get(middlePoint + 1 + j) item to Right LinkedList

7-) decleare two variables named i,j,k and i and j initialize to zero,

and set k to left.

8-) while(i < n1 and j < n2) do

if Left.get(i) < Right.get(j) do

set k. indexed item in arr to Left.get(i) item

increase i by one

else do

set k. indexed item in arr to Right.get(j) item

increase j by one

increase k by one

9-) while (i < n1) do

set k.indexed item in arr to Left.get(i)

increase i by one

increase k by one

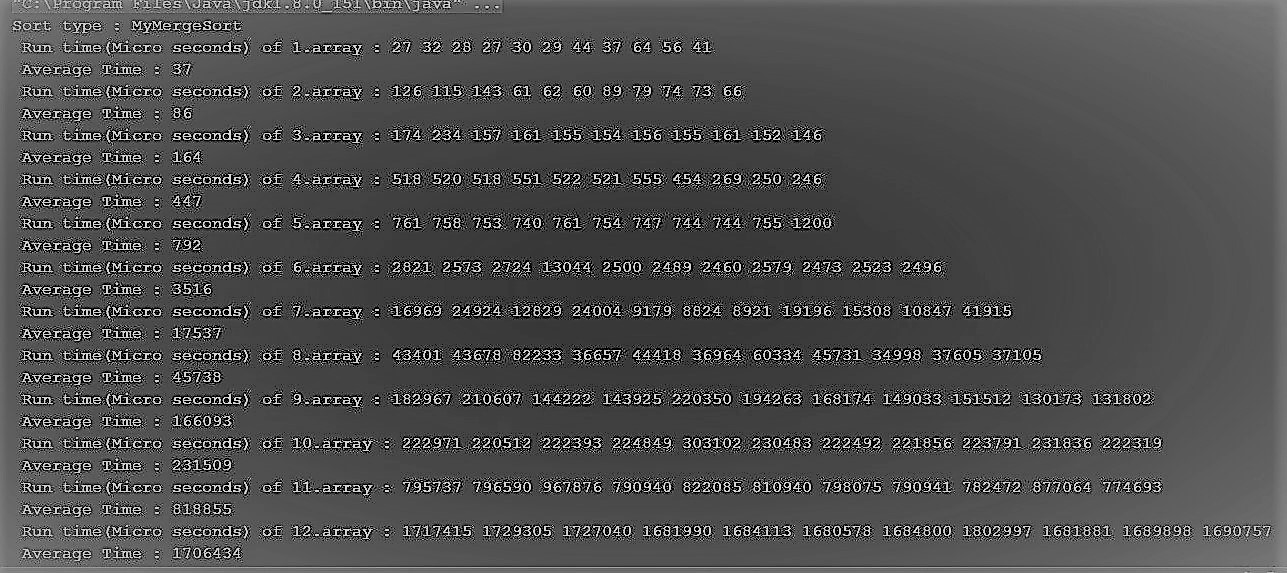
10-) while (j < n2) do

set k.indexed item in arr to Right.get(j)

increase j by one

increase k by one

### Average Run Time Analysis



Since the above pictures is not enough clear ,I need to explain it.

Input sizes are ,in turn,20,40,100,200,400,800,1600,3200,4000,7000 and 10000.

Run time of merge sort with dll is reasonable up to 800 input size.In input size that is bigger than 1000, Run time of merge sort with dll is increasing much more and more ,and effiency of it is getting bad.Probably, time complexity of merge with dll is a bit bigger than nlogn.

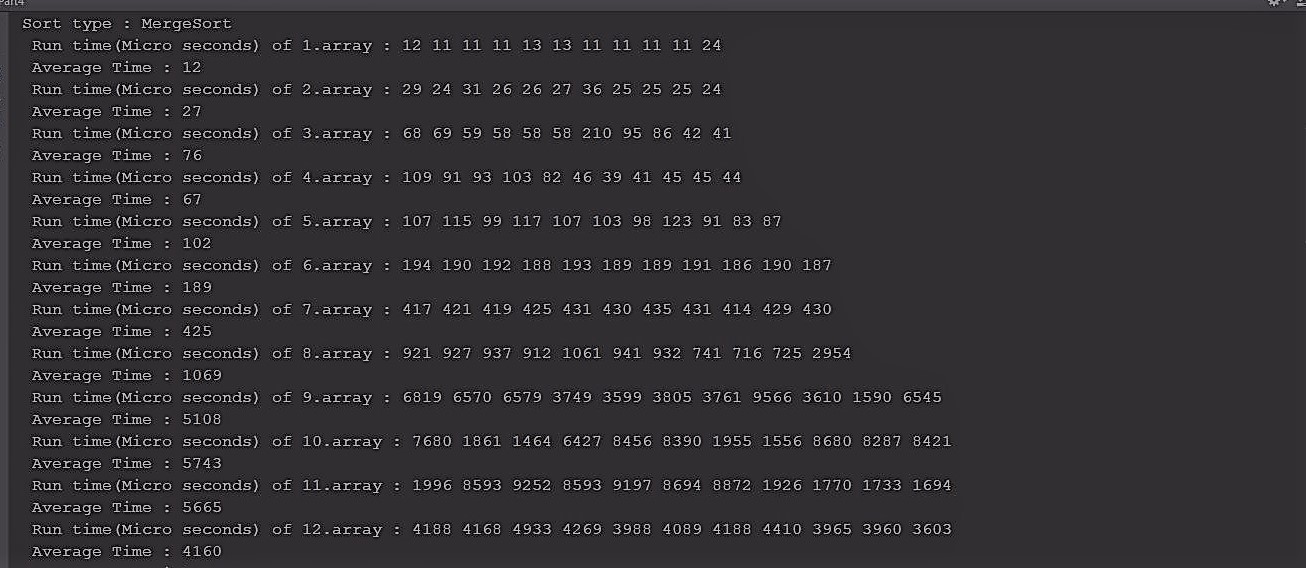
### Wort-case Performance Analysis



As seen above,run time of the worst case of merge sort with dll is increasing when input size is increase.Predictive of time complexity of it is bigger than nlogn.

## MergeSort

### Average Run Time Analysis



Input sizes are ,in turn,20,40,100,200,400,800,1600,3200,4000,7000 and 10000.

As seen above ,input size is getting increase ,avarage run time of merge sort increases with respect to nlog complexity ,generally.But since orderless of arrays is not same level , run time of array that has bigger input size maybe smaller than one that has smaller input size.

### Wort-case Performance Analysis

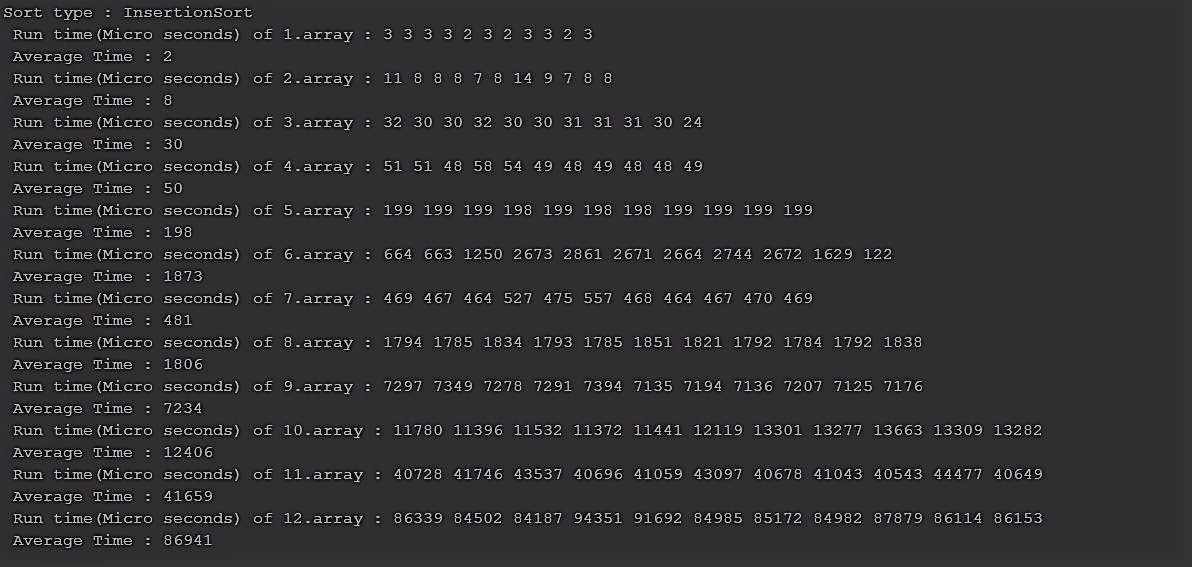


Input sizes are ,in turn,100,1000,2000,5000 and 10000

Time complexity of this merge sort algorithm at the best case and worst case is O(nlogn).

## Insertion Sort

### Average Run Time Analysis



Input sizes are ,in turn,20,40,100,200,400,800,1600,3200,4000,7000 and 10000.

As seen above ,input size is getting increase ,avarage run time of insertion sort increases with respect to n\*n complexity.But since orderless of arrays is not same level , run time of array that has bigger input size maybe smaller than one that has smaller input size.(like input size 400 and 800).

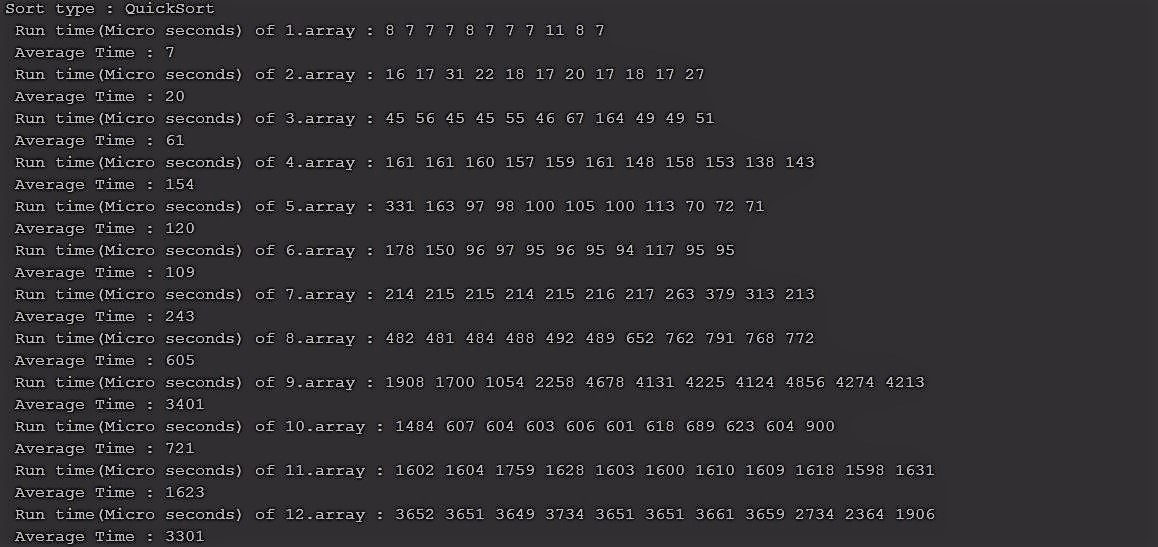
### Wort-case Performance Analysis

Input sizes are ,in turn,100,1000,2000,5000 and 10000

Time complexity of this insertion sort algorithm at worst case is O(n\*n).

## Quick Sort

### Average Run Time Analysis



Input sizes are ,in turn,20,40,100,200,400,800,1600,3200,4000,7000 and 10000.

As seen above ,input size is getting increase ,avarage run time of insertion sort increases with respect to n\*logn complexity.But since orderless of arrays is not same level , run time of array that has bigger input size maybe smaller than one that has smaller input size.(like input size 4000 and 7000).

### Wort-case Performance Analysis

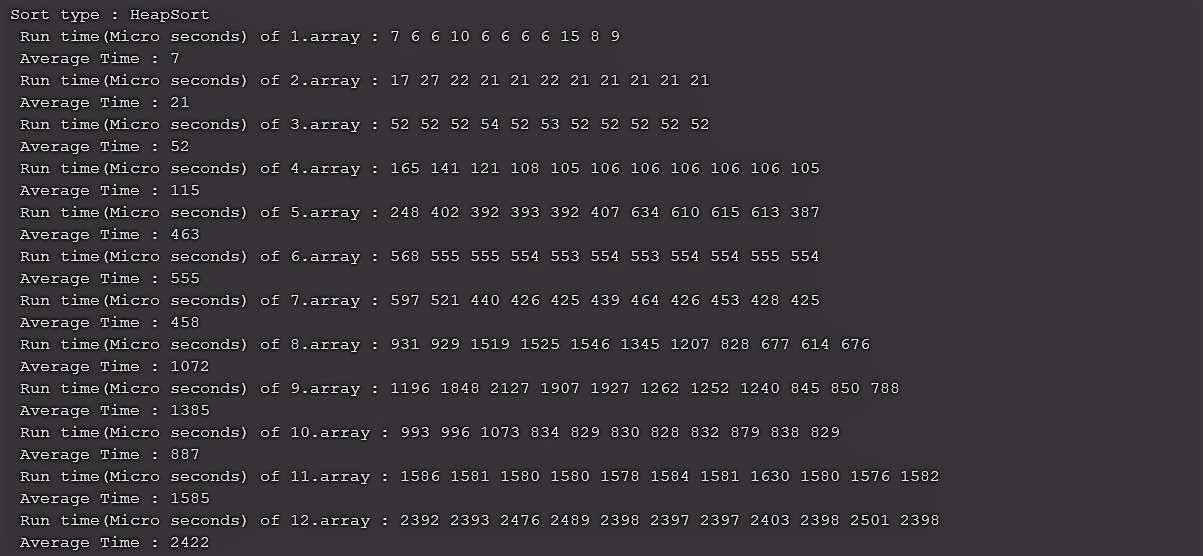


Input sizes are ,in turn,100,1000,2000,5000 and 10000

Time complexity of this Quick sort algorithm at worst case is O(n\*n).

## Heap Sort

### Average Run Time Analysis



Input sizes are ,in turn,20,40,100,200,400,800,1600,3200,4000,7000 and 10000.

As seen above ,input size is getting increase ,avarage run time of heap sort increases with respect to n\*logn complexity.But since orderless of arrays is not same level , run time of array that has bigger input size maybe smaller than one that has smaller input size.(like input size 3200 and 4000).

### Wort-case Performance Analysis



Input sizes are ,in turn,100,1000,2000,5000 and 10000

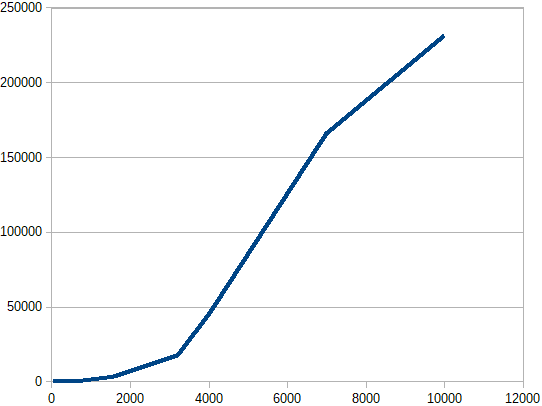
Time complexity of this heap sort algorithm at worst case is O(n\*logn).

# Average Run Time Analysis Tables And Graph

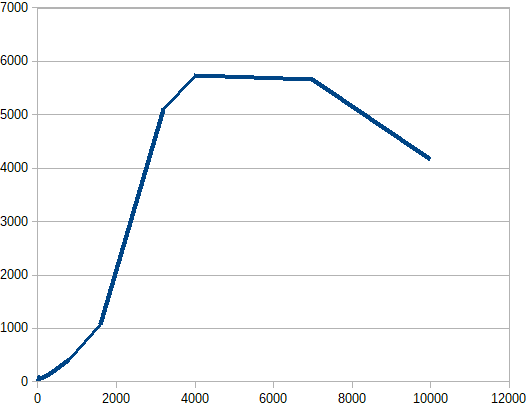


Run times with respect to input sizes in terms of micro seconds.

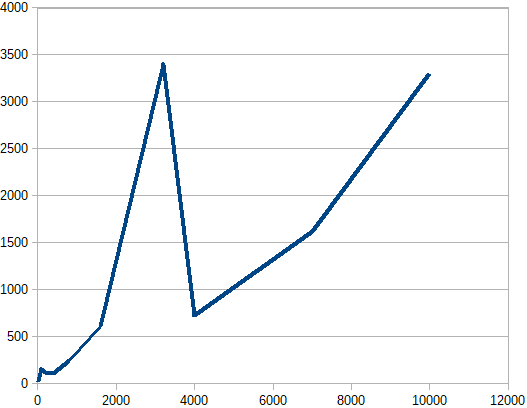
**a) MyMergeSort Run Time versus Input Size**



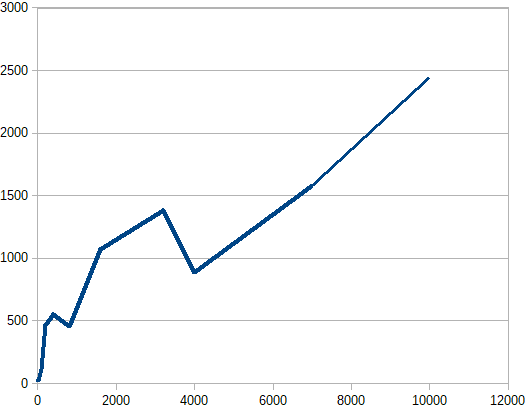
**b) MergeSort Run Time versus Input Size**

****

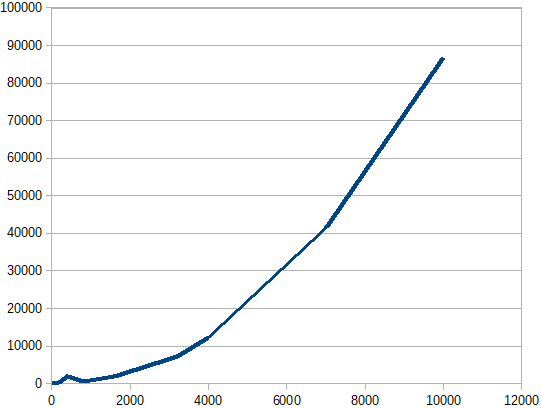
**c) QuicSort Run Time versus Input Size**



**d) HeapSort Run Time versus Input Size**

****

**e) InsertionSort Run Time versus Input Size**



# **Worst Case Run Time Analysis Tables And Graph**