#### Complexity Analysis of

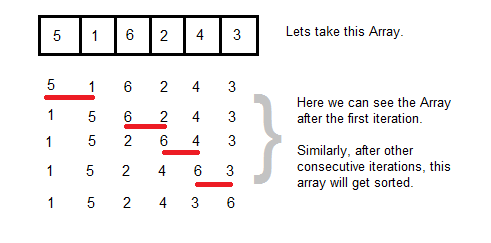
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Worst Case Time Complexity** | **Best Case Time Complexity** | **Average Time Complexity** | **Space Complexity** |
| Insertion Sorting | O(n2) | O(n) | O(n2) | O(1) |
| Bubble Sort | O(n2) | O(n) | O(n^2) | O(1) |
| Selection Sorting | O(n2) | O(n2) | O(n2) | O(1) |
| Quick Sort | O(n2) | O(n log n) | O(n log n) | O(n log n) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) |

## Bubble Sorting

**Bubble Sort** is an algorithm which is used to sort **N** elements that are given in a memory for eg: an Array with **N** number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the largest element in the list bubbles up towards the last place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.



#### Sorting using Bubble Sort Algorithm

Let's consider an array with values {5, 1, 6, 2, 4, 3}

int a[6] = {5, 1, 6, 2, 4, 3};

int i, j, temp;

for(i=0; i<6; i++)

{

for(j=0; j<6-i-1; j++)

{

if( a[j] > a[j+1])

{

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

//now you can print the sorted array after this

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm is not efficient because as per the above logic, the for-loop will keep executing for six iterations even if the array gets sorted after the second iteration.

Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not in the following iteration. If no swapping is taking place, it means the array is sorted and we can jump out of the for loop, instead executing all the iterations.

int a[6] = {5, 1, 6, 2, 4, 3};

int i, j, temp;

for(i=0; i<6; i++)

{

int **flag** = 0; *//taking a flag variable*

for(j=0; j<6-i-1; j++)

{

if( a[j] > a[j+1])

{

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

**flag** = 1; *//setting flag as 1, if swapping occurs*

}

}

if(!**flag**) *//breaking out of for loop if no swapping takes place*

{

break;

}

}

In the above code, if in a complete single cycle of j iteration(inner for loop), if no swapping takes place, then flag will remain 0 and then we will break out of the for loops, because the array has already been sorted.

#### Complexity Analysis of Bubble Sorting

In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be

(n-1)+(n-2)+(n-3)+.....+3+2+1

Sum = n(n-1)/2

i.e O(n2)

Hence the complexity of Bubble Sort is **O(n2)**.

The main advantage of Bubble Sort is the simplicity of the algorithm. Space complexity for Bubble Sort is **O(1)**, because only single additional memory space is required i.e. for **temp** variable

**Best-case** Time Complexity will be **O(n)**, it is when the list is already sorted.

#### Complexity Analysis of Bubble Sort

**Worst Case Time Complexity :** O(n^2)

**Best Case Time Complexity :** O(n)

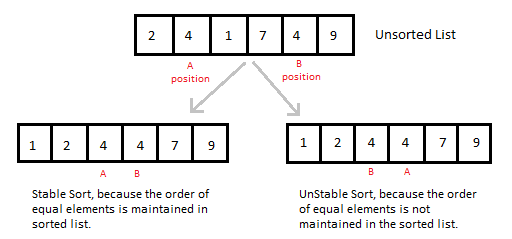
**Average Time Complexity :** O(n^2)

**Space Complexity :** O(1)

## Insertion Sorting

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less. Like Bubble Sorting, insertion sort also requires a single additional memory space.
6. It is a **Stable** sorting, as it does not change the relative order of elements with equal keys



#### How Insertion Sorting Works



#### Sorting using Insertion Sort Algorithm

int a[6] = {5, 1, 6, 2, 4, 3};

int i, j, key;

for(i=1; i<6; i++)

{

key = a[i];

j = i-1;

while(j>=0 && key < a[j])

{

a[j+1] = a[j];

j--;

}

a[j+1] = key;

}

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable **key**, in which we put each element of the array, in each pass, starting from the second element, that is **a[1]**.

Then using the while loop, we iterate, until **j** becomes equal to zero or we find an element which is greater than **key**, and then we insert the key at that position.

In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

#### Insertion Sorting in C++

#include <stdlib.h>

#include <iostream.h>

using namespace std;

*//member functions declaration*

void **insertionSort**(int arr[], int length);

void **printArray**(int array[],int size);

int **main**() {

int array[5]= {5,4,3,2,1};

**insertionSort**(*array*,5);

return 0;

}

void **insertionSort**(int arr[], int length) {

int i, j ,tmp;

for (i = 1; i < length; i++) {

j = i;

while (j > 0 && arr[j - 1] > arr[j]) {

tmp = arr[j];

arr[j] = arr[j - 1];

arr[j - 1] = tmp;

j--;

}

printArray(arr,5);

}

}

void **printArray**(int array[], int size){

cout<< "Sorting tha array using Insertion sort... ";

int j;

for (j=0; j < size;j++)

for (j=0; j < size;j++)

cout <<" "<< array[j];

cout << endl;

}

#### Complexity Analysis of Insertion Sorting

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n)

**Average Time Complexity :** O(n2)

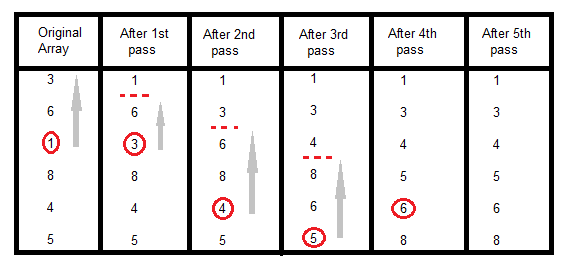
**Space Complexity :** O(1)

## Selection Sorting

Selection sorting is conceptually the most simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

**Note:**Selection sort is an unstable sort i.e it might change the occurrence of two similar elements in the list while sorting. But it can also be a stable sort when it is implemented using linked list data structures.

#### How Selection Sorting Works



In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving the first element, next smallest element is searched from the rest of the elements. We get 3 as the smallest, so it is then placed at the second position. Then leaving 1 and 3, we search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

#### Sorting using Selection Sort Algorithm

void selectionSort(int a[], int size)

{

int i, j, min, temp;

for(i=0; i < size-1; i++ )

{

min = i; //setting min as i

for(j=i+1; j < size; j++)

{

if(a[j] < a[min]) //if element at j is less than element at min position

{

min = j; //then set min as j

}

}

temp = a[i];

a[i] = a[min];

a[min] = temp;

}

}

#### Complexity Analysis of Selection Sorting

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n2)

**Average Time Complexity :** O(n2)

**Space Complexity :** O(1)

## Quick Sort Algorithm

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not a stable search, but it is very fast and requires very less additional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :

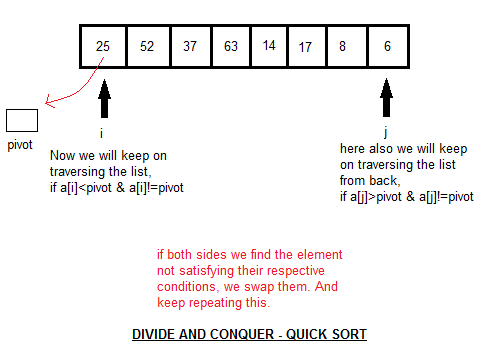
1. Elements less than the Pivot element
2. Pivot element(Central element)
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

#### How Quick Sorting Works



#### Sorting using Quick Sort Algorithm

/\* a[] is the array, p is starting index, that is 0,

and r is the last index of array. \*/

void **quicksort**(int a[], int p, int r)

{

if(p < r)

{

int q;

q = partition(a, p, r);

quicksort(a, p, q);

quicksort(a, q+1, r);

}

}

int **partition**(int a[], int p, int r)

{

int i, j, pivot, temp;

pivot = a[p];

i = p;

j = r;

while(1)

{

while(a[i] < pivot && a[i] != pivot)

i++;

while(a[j] > pivot && a[j] != pivot)

j--;

if(i < j)

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

else

{

return j;

}

}

}

#### Complexity Analysis of Quick Sort

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :** O(n log n)

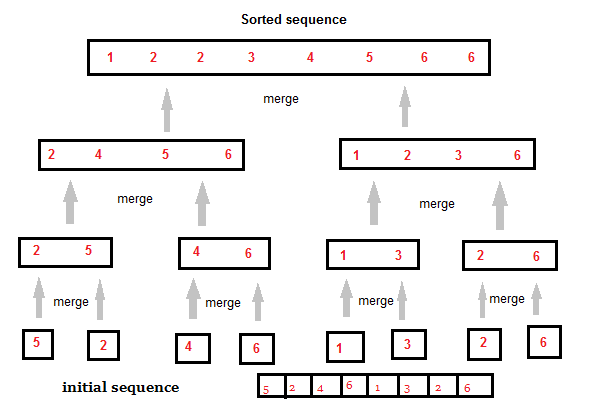
* Space required by quick sort is very less, only O(n log n) additional space is required.
* Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

## Merge Sort Algorithm

Merge Sort follows the rule of **Divide and Conquer**. In merge sort the unsorted list is divided into N sublists, each having one element, because a list consisting of one element is always sorted. Then, it repeatedly merges these sublists, to produce new sorted sublists, and in the end, only one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of **O(n log n)**. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

#### How Merge Sort Works



Like we can see in the above example, merge sort first breaks the unsorted list into sorted sublists, each having one element, because a list of one element is considered sorted and then it keeps merging these sublists, to finally get the complete sorted list.

#### Sorting using Merge Sort Algorithm

/\* a[] is the array, p is starting index, that is 0,

and r is the last index of array. \*/

//Lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

void **mergesort**(int a[], int p, int r)

{

int q;

if(p < r)

{

q = floor( (p+r) / 2);

mergesort(a, p, q);

mergesort(a, q+1, r);

merge(a, p, q, r);

}

}

void **merge**(int a[], int p, int q, int r)

{

int b[5]; //same size of a[]

int i, j, k;

k = 0;

i = p;

j = q+1;

while(i <= q && j <= r)

{

if(a[i] < a[j])

{

b[k++] = a[i++]; // same as b[k]=a[i]; k++; i++;

}

else

{

b[k++] = a[j++];

}

}

while(i <= q)

{

b[k++] = a[i++];

}

while(j <= r)

{

b[k++] = a[j++];

}

for(i=r; i >= p; i--)

{

a[i] = b[--k]; // copying back the sorted list to a[]

}

}

#### Complexity Analysis of Merge Sort

**Worst Case Time Complexity :** O(n log n)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :** O(n)

* Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
* It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.
* It is the best Sorting technique used for sorting **Linked Lists**.

## Heap Sort Algorithm

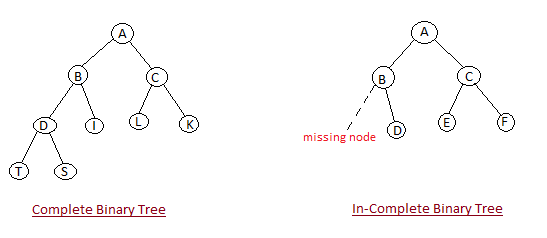
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts :

* Creating a Heap of the unsorted list.
* Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

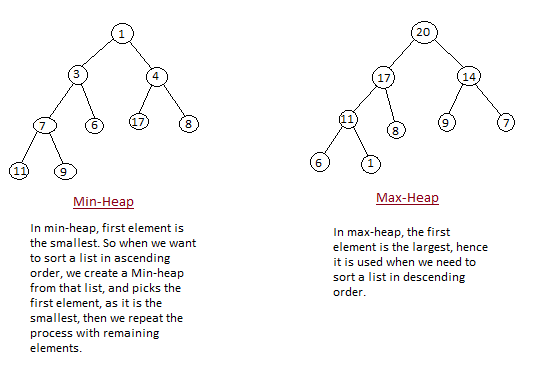
#### What is a Heap ?

Heap is a special tree-based data structure, that satisfies the following special heap properties :

1. **Shape Property :** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.



1. **Heap Property :** All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.



#### How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

In the below algorithm, initially **heapsort()** function is called, which calls **buildheap()** to build heap, which inturn uses **satisfyheap()** to build the heap.

#### Sorting using Heap Sort Algorithm

/\* Below program is written in C++ language \*/

void heapsort(int[], int);

void buildheap(int [], int);

void satisfyheap(int [], int, int);

void main()

{

int a[10], i, size;

cout << "Enter size of list"; // less than 10, because max size of array is 10

cin >> size;

cout << "Enter" << size << "elements";

for( i=0; i < size; i++)

{

cin >> a[i];

}

heapsort(a, size);

getch();

}

void **heapsort**(int a[], int length)

{

buildheap(a, length);

int heapsize, i, temp;

heapsize = length - 1;

for( i=heapsize; i >= 0; i--)

{

temp = a[0];

a[0] = a[heapsize];

a[heapsize] = temp;

heapsize--;

satisfyheap(a, 0, heapsize);

}

for( i=0; i < length; i++)

{

cout << "\t" << a[i];

}

}

void **buildheap**(int a[], int length)

{

int i, heapsize;

heapsize = length - 1;

for( i=(length/2); i >= 0; i--)

{

satisfyheap(a, i, heapsize);

}

}

void **satisfyheap**(int a[], int i, int heapsize)

{

int l, r, largest, temp;

l = 2\*i;

r = 2\*i + 1;

if(l <= heapsize && a[l] > a[i])

{

largest = l;

}

else

{

largest = i;

}

if( r <= heapsize && a[r] > a[largest])

{

largest = r;

}

if(largest != i)

{

temp = a[i];

a[i] = a[largest];

a[largest] = temp;

satisfyheap(a, largest, heapsize);

}

}

#### Complexity Analysis of Heap Sort

**Worst Case Time Complexity :** O(n log n)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :** O(1)

* Heap sort is not a Stable sort, and requires a constant space for sorting a list.
* Heap Sort is very fast and is widely used for sorting.

**HashMap Custom implementation in java**

**Save**

[**Tweet**](https://dzone.com/articles/hashmap-custom-implementation)

**12.41k Views**

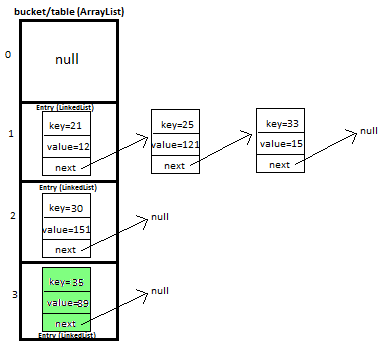
Join the DZone community and get the full member experience.

[**JOIN FOR FREE**](https://dzone.com/articles/hashmap-custom-implementation)

**Contents of page :**

* Custom HashMap >
* Entry<K,V>
* Putting 5 key-value pairs in HashMap  (step-by-step)>
* Methods used in custom HashMap >
* What will happen if map already contains mapping for key?
* Complexity calculation of put and get methods in HashMap >
  + put method - worst Case complexity >
  + put method - best Case complexity >
  + get method - worst Case complexity >
  + get method - best Case complexity >
* Summary of complexity of methods in HashMap >

**Custom HashMap >**

This is very **important**and **trending**topic. In this post i will be explaining HashMap custom implementation in lots of detail with diagrams which will help you in visualizing the HashMap implementation.

**I will be explaining how we will put and get key-value pair in HashMap by overriding-**

> **equals**method - helps in checking equality of entry objects.

> **hashCode**method - helps in finding bucket’s index on which data will be stored.

We will maintain bucket ( [ArrayList](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)) which will store Entry ( [LinkedList](http://javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html)).

Entry<K,V>

We store key-value pair by usingEntry<K,V>

Entry contains

* K key,
* V value and
* Entry<K,V>next  (i.e. next entry on that location of bucket).

static class Entry<K, V> {

K key;

V value;

Entry<K,V> next;

public Entry(K key, V value, Entry<K,V> next){

this.key = key;

this.value = value;

this.next = next;

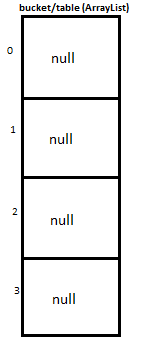
}

}

**Putting 5 key-value pairs in custom HashMap (step-by-step)>**

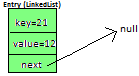
I will explain you the whole concept of HashMap by putting 5 key-value pairs in HashMap.

**Initially**, we have bucket of capacity=4. (all indexes of bucket i.e. 0,1,2,3 are pointing to null)



**Let’s put first key-value pair in HashMap-**

Key=21, value=12

newEntry Object will be formed like this > 

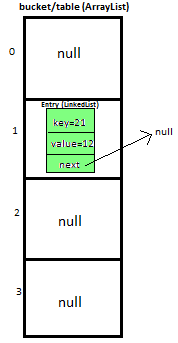
We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 21%4= 1.

So, 1 will be the index of bucket on which newEntry object will be stored.

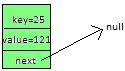
We will go to 1stindex as it is pointing to null we will put our newEntry object there.

**At completion of this step, our HashMap will look like this-**



**Let’s put second key-value pair in HashMap-**

Key=25, value=121

newEntry Object will be formed like this > 

We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 25%4= 1.

So, 1 will be the index of bucket on which newEntry object will be stored.

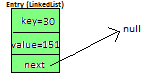
We will go to 1st index, it contains entry with key=21, we will compare two keys(i.e. compare 21 with 25 by using equals method), as two keys are different we check whether entry with key=21’s next is null or not, if next is null we will put our newEntry objecton next.

**At completion of this step our HashMap will look like this-**



**Let’s put third key-value pair in HashMap-**

Key=30, value=151

newEntry Object will be formed like this > 

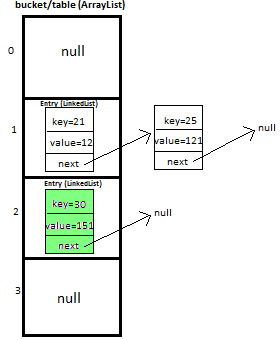
We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 30%4= 2.

So, 2 will be the index of bucket on which newEntry object will be stored.

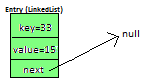
We will go to 2nd index as it is pointing to null we will put our newEntry object there.

**At completion of this step, our HashMap will look like this-**



**Let’s put fourth key-value pair in HashMap-**

Key=33, value=15

Entry Object will be formed like this > 

We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 33%4= 1,

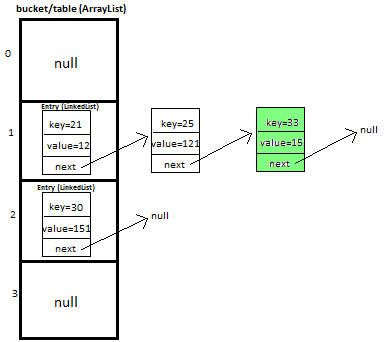
So, 1 will be the index of bucket on whichnewEntry object will be stored.

We will go to 1st index -

>it contains entry with key=21, we will compare two keys (i.e. compare 21 with 33 by using equals method, as two keys are different,  proceed to next  of entry with key=21 (proceed only if next is not null).

>now, next contains entry with key=25, we will compare two keys (i.e. compare 25 with 33 by using equals method, as two keys are different,  now next of entry with key=25 is pointing to null so we won’t proceed further, we will put our newEntry object on next.

**At completion of this step our HashMap will look like this-**

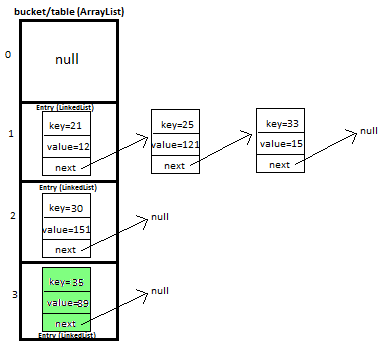


**Let’s put fifth key-value pair in HashMap-**

Key=35, value=89

Repeat above mentioned steps.

**At completion of this step our HashMap will look like this-**



Must read: [LinkedHashMap Custom implementation](http://javamadesoeasy.com/2015/02/linkedhashmap-custom-implementation.html)

Methods used in custom HashMap >

|  |  |
| --- | --- |
| public void put(K newKey, V data) | -Method allows you put key-value pair in HashMap  -If the map already contains a mapping for the key, the old value is replaced.  -provide complete functionality how to override equals method.  -provide complete functionality how to override hashCode method. |
| public V get(K key) | Method returns value corresponding to key. |
| public boolean remove(K deleteKey) | Method removes key-value pair from HashMapCustom. |
| public void display() | -Method displays all key-value pairs present in HashMapCustom.,  -insertion order is not guaranteed, for maintaining insertion order refer [LinkedHashMapCustom](http://javamadesoeasy.com/2015/02/linkedhashmap-custom-implementation.html). |
| private int hash(K key) | -Method implements hashing functionality, which helps in finding the appropriate bucket location to store our data.  -This is very important method, as performance of HashMapCustom is very much dependent on  this method's implementation. |

**What will happen if map already contains mapping for key?**

If the map already contains a mapping for the key, the old value is replaced.

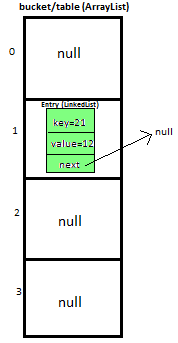
**Complexity calculation of put and get methods in HashMap >**

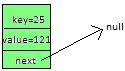
**put method - worst Case complexity >**

O(n).

**But how complexity is O(n)?**

Initially, let's say map is like this -



And we have to insert newEntry Object with Key=25, value=121 

We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 25%4= 1.

So, 1 will be the index of bucket on which newEntry object will be stored.

We will go to 1st index, it contains entry with key=21, we will compare two keys(i.e. compare 21 with 25 by using equals method), as two keys are different we check whether entry with key=21’s next is null or not, if next is null we will put our newEntry objecton next.

At completion of this step our HashMap will look like this-



Now let’s do complexity calculation -

Earlier there was 1 element in HashMap and for putting newEntry Object we iterated on it. Hence complexity was O(n).

Note: We may calculate complexity by adding more elements in HashMap as well, but to keep explanation simple i kept less elements in HashMap.

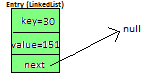
**put method - best Case complexity >**

O(1).

But how complexity is O(n)?

Let's say map is like this -



And we have to insert newEntry Object with Key=30, value=151 

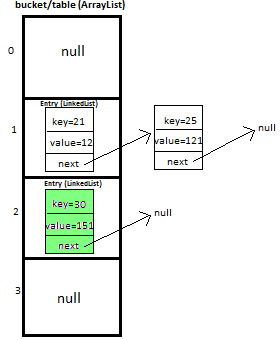
We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 30%4= 2.

So, 2 will be the index of bucket on which newEntry object will be stored.

We will go to 2nd index as it is pointing to null we will put our newEntry object there.

At completion of this step our HashMap will look like this-



Now let’s do complexity calculation -

Earlier there 2 elements in HashMap but we were able to put newEntry Object in first go. Hence complexity was O(1).

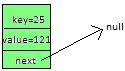
**get method - worst Case complexity >**

O(n).

But how complexity is O(n)?

Initially, let's say map is like this -



And we have to get Entry Object with Key=25, value=121 

We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 25%4= 1.

So, 1 will be the index of bucket on which Entry object is stored.

We will go to 1st index, it contains entry with key=21, we will compare two keys(i.e. compare 21 with 25 by using equals method), as two keys are different we check whether entry with key=21’s next is null or not,  next is not null so we will repeat same process and ultimately will be able to get Entry object.

Now let’s do complexity calculation -

There were 2 elements in HashMap and for getting Entry Object we iterated on both of them. Hence complexity was O(n).

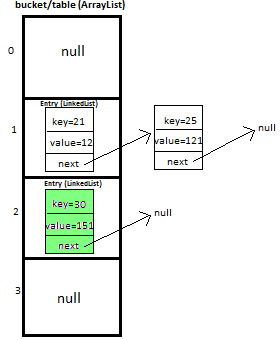
Note: We may calculate complexity by using HashMap of larger size, but to keep explanation simple i kept less elements in HashMap.

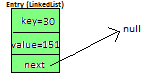
**get method - best Case complexity >**

O(1).

But how complexity is O(n)?

Initially, let's say map is like this -



And we have to get Entry Object with Key=30, value=151 

We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 30%4= 2.

So, 2 will be the index of bucket on which Entry object is stored.

We will go to 2nd index and get Entry object.

Now let’s do complexity calculation -

There were 3 elements in HashMap but we were able to get Entry Object in first go.

Hence complexity was O(1).

**Summary of complexity of methods in HashMap >**

|  |  |  |
| --- | --- | --- |
| Operation/ method | Worst case | Best case |
| put(K key, V value) | O(n) | O(1) |
| get(Object key) | O(n) | O(1) |