DataStructures and Its usages – 2021-2022

# **Create your own ArrayList**

**public** **void** addFirst(Object x) {

**if** (data.length == size)

ensureCapacity();

size++;

**data = *addByIndex*(data, x, 0);**

}

**import** java.util.Arrays;

**public** **class** MyArrayList {

**private** **int** size;

**private** **int** defaultCapacity = 10;

**private** Object[] data;

**public** **void** addLast(Object x) {

**if** (data.length == size)

ensureCapacity();

**data[size++] = x;**

}

**public** MyArrayList() {

data = **new** Object[defaultCapacity];

}

**public** **void** ensureCapacity() {

data = Arrays.*copyOf*(data, data.length \* 2);

**public** **void** deleteFirst() {

**deleteByIndex(0);**

}

}

**public** **void** condenseArray(**int** start) {

**for** (**int** i = start; i < size; i++)

data[i] = data[i + 1];

**public** **void** deleteLast() {

**int index = data.length - 1;**

**deleteByIndex(index);**

}

}

**public** **void** trimArray() {

data = Arrays.*copyOf*(data, size);

}

**private** **static** Object[] addByIndex(Object[] a, Object key, **int** index) {

**public** **int** size() {

**return** size;

}

Object[] b = **new** Object[a.length + 1];

**for (int i = 0; i < index; i++) { // Copy 0 to index**

**b[i] = a[i];**

**}**

**b[index] = key; // put at index**

**for** (**int** i = index + 1; i < b.length; i++) { // copy from index+1 to last

b[i] = a[i - 1];

}

**return** b;

}

**public** **static** Object[] copyOf(Object[] a, **int** newLen) {

Object[] b = **new** Object[newLen];

**int** minLen = Math.*min*(a.length, newLen);

**for** (**int** i = 0; i < minLen; i++) {

b[i] = a[i];

}

**return** b;

}

**public** **static** Object[] copyOfRange(Object[] a, **int** from, **int** to) {

**int** actualTo = Math.*min*(to, a.length);

**int** len = actualTo - from;

Object[] b = **new** Object[len];

**for** (**int** i = from, j = 0; i < actualTo; i++, j++) {

b[j] = a[i];

}

**return** b;

}

**public** **void** deleteObject(Object x) {

**for** (**int** i = 0; i < data.length; i++) {

**if** (data[i].equals(x)) {

size--;

data[i] = **null**;

condenseArray(i);

**break**;

}

}

}

**public** **void** deleteByIndex(**int** index) {

**for** (**int** i = 0; i < data.length; i++) {

**if** (i == index) {

size--;

data[i] = **null**;

condenseArray(i);

**break**;

}

}

}

**LinkedList in Java**

//Reference http://www.cs.bu.edu/fac/snyder/cs112/CourseMaterials/LinkedListNotes.html

**public** **class** LinkedList1 {

**private** Node front;

**private** Node back;

**public** **void** addFirst(Object x) {

Node f = front;

front = **new** Node(x);

front.next = f;

}

**private** **static** **class** Node {

**private** Object element;

**private** Node next;

**public** Node( Object data ) {

**public** **void** addLast(Object x) {

**if** (front == **null**)

back = front = **new** Node(x);

**else**

back = back.next = **new** Node(x);

}

element = data;

next = **null**;

}

}

**public** **void** deleteFirst() {

**if** (front == **null**) **return**;

**public** **void** deleteLast() {

**if** (front == **null**) **return**;

Object backData = back.data;

System.***out***.println(backData);

**if** (front == back)

front = back = **null**;

**else** {

Node prev = front;

**while** (prev.next != back)

prev = prev.next;

back = prev;

prev.next = **null**;

}

}

Object firstData = front.data;

front = front.next;

}

**public** **void** deleteLast2() { // Follow this

**if** (front == **null**) **return**;

Object data = back.data; // you can return

System.***out***.println("Back Data: " + data);

**if** (front == back)

front = back = **null**;

**else** {

Node temp = front;

**for** (Node p = front; p != back; p = p.next) {

temp = p;

}

back = temp;

temp.next = **null**;

**public** Object poll() {

**return** deleteFirst();

}

}

}

//Print all the elements

**public** **void** print() {

**for**( Node p = front ; p != **null** ; p = p.next )

**public** Object pop() {

**return** deleteLast();

}

System.*out*.print(p.element+" ");

}

**public** **void** reverse() { // Better way

Node prev = **null**;

Node nextN = **null**;

Node current = front;

**while** (current != **null**) {

nextN = current.next;

current.next = prev;

prev = current;

current = nextN;

}

front = prev;

}

//Another way of writing delete last

**public** Object deleteLast1( ) { //POP() operation

**if**( front == **null** ) **throw** **new** RuntimeException( "No data" );

Object returnValue = back.element;

**if**( front == back ) front = back = **null**;

**else** {

Node p = front;

**for**( ; p.next != back ; p = p.next ) {

//Nothing to do

}

back = p;

p.next = **null**;

}

**return** returnValue;

}

**public** Object get(**int** index ) {

**int** counter = 0;

Object data = **null**;

**for**(Node p = front ; p != **null** ; p = p.next ) {

**if**(counter == index ) {

data = p.element;

**break**;

}

counter++;

}

**return** data;

}

**public void printRecursive(Node p) {**

**if( p != null) {**

System.*out*.print(p.element+" ");

**printRecursive(p.next)**

**}**

**}**

**public** **void** print() {

**printRecursive(front);**

}

**public** **void** printReverseList() {

// Print in reverse

printReverseList(front);

}

**private** **void** printReverseList( Node p ) {

**if** (p != **null**) {

printReverseList( p.next );

System.*out*.println(p.element);

}

}

//~~~~~~~~~~~~~ size() or length() operations ~~~~~~~~~~~~

**public** **int** size() {

**int** counter = 0;

**for**(Node p = front ; p != **null**; p = p.next )

counter++;

**return** counter;

}

**public** **int** length( Node p ) {

// Example call: int len = length(head.next);

**if** (p == **null**)

**public** **int** length() {

**return** length(front);

}

**return** 0;

**else**

**return** 1 + length(p.next );

}

//Find the middle element

**public** **void** findMiddle() {

**int** length = 0;

Node middle = front;

**for**( Node p = front ; p != **null** ; p = p.next ) {

length++;

**if**( length % 2 == 0 )

middle = middle.next;

}

System.*out*.println("Middle Element : "+middle.element);

}

//for( Node p = front.next,q = front; p != null ; q = p , p = p.next )

// During first iteration, p points to first element and q to the header node;

// thereafter, p points to a node and q points to the previous node

**public** **void** deleteObject(Object x) {

**if** ( front != **null** && front.element.equals(x) )

front = front.next; // special case, have to delete first node?

**else {**

**for**( Node p = front,q = front; p != **null** ; q = p , p = p.next ) {

**if**( p.element.equals(x) ) {

q.next = p.next;

**break**;

}

}

}

}

**public** **void** deleteAt(**int** p) {

front = deleteAt(p, front);

}

**private** Node deleteAt(**int** p, Node node) {

**if**(node == **null**) **throw** **new** NoSuchElementException("cannot delete.");

**else**

**if**(p == 0)

**return** node.next;

**else**

node.next = deleteAt(p-1, node.next);

**return** node;

}

**public** **void** deleteByPosition(**int** index) {

**int** counter = 0;

**if** (front == **null**) **return**;

**if**(index == 0) front = front.next;

**for** (Node p = front, q = front; p != **null**; q = p, p = p.next) {

**if**(counter == index) {

q.next = p.next;

**break**;

}

counter++;

}

}

public void deleteByIndex(int index) {

        int counter = 0;

        if(front != null && index == 0) {

            front = front.next;

        } else {

            for(Node p = front, q = front; p != null; q = p, p = p.next) {

                if(counter == index) {

                    q.next = p.next;

                    break;

                }

                counter++;

            }

        }

    }

//Floyd’s Cycle Detection Algorithm Tortoise and Hare

**public** **boolean** isCyclic() {

Node fast = front;

Node slow = front;

**while** (fast != **null** && fast.next != **null**) {

fast = fast.next.next;

slow = slow.next;

//if fast and slow pointers are meeting then LinkedList is cyclic

**if** (fast == slow) {

**return** **true**;

}

}

**return** **false**;

}

**public** **void** reverseList() { //Reverse the LinkedList

recursiveReverse(front);

}

**public** **void** reverse(Node h) {

Node p = h.next, q = h, r;

**while** ( p != **null** ) {

r = q; // r follows q

q = p; // q follows p

p = p.next; // p moves to next node

q.next = r; // link q to preceding node

}

front.next.next = **null**;

front.next = q;

}

**public** **void** recursiveReverse(Node currentNode ) {

**if**(currentNode == **null**) **return**; //check for empty list

/\* if we are at the TAIL node: recursive base case: \*/

**if**(currentNode.next == **null**) {

//set HEAD to current TAIL since we are reversing list

front = currentNode;

**return**; //since this is the base case

}

recursiveReverse(currentNode.next);

currentNode.next.next = currentNode;

currentNode.next = **null**; //set "old" next pointer to NULL

}

**public** **static** **void** main(String[] args) {

LinkedList1 list = **new** LinkedList1();

//Add elements

**for**( **int** i = 0 ; i < 17 ; i++ )

list.add(**new** Integer(i));

//Display elements

list.print();

System.*out*.println("\n");

//Display First Time

System.*out*.println("-------------Display First Time----------");

**for**( **int** i = 0 ; i < list.size() ; i++ )

System.*out*.print(list.get(i)+" ");

//Display Second Time

System.*out*.println("\n-------------Display Second Time----------");

**for**( **int** i = 0 ; i < list.size() ; i++ )

System.*out*.print(list.get(i)+" ");

list.deleteFirst();//First object

list.deleteLast();//Last Object

System.*out*.println("\n");

list.print();

System.*out*.println("\n");

//POP() operation

System.*out*.println("pop() : "+list.pop());

System.*out*.println("pop() : "+list.pop());

System.*out*.println("\n");

list.print();

//POLL() operation

System.*out*.println("poll() : "+list.poll());

System.*out*.println("poll() : "+list.poll());

list.print();

//Delete from the index

System.*out*.println("\nDelete from the middle");

list.deleteAt(8);

list.print();

System.*out*.println("\nDelete an object");

list.deleteObject(**new** Integer(13));

list.print();

System.*out*.println("\n");

list.findMiddle();

System.*out*.println("All elements .....");

list.print();

list.reverseList();

System.*out*.println("\n");

list.print();

}

}

**Detect Loop Inside a LinkedList**

public class LoopInLinkedList {

private Node front;

private Node back;

private static class Node {

private Node next;

private Object element;

public Node(Object data) {

this.element = data;

this.next = null;

}

}

public void addLast(Object x) {

if (front == null)

back = front = new Node(x);

else

back = back.next = new Node(x);

}

**//Floyd’s Cycle Detection Algorithm Tortoise and Hare**

**public boolean isCyclic() {**

**Node fast = front;**

**Node slow = front;**

**while (fast != null && fast.next != null) {**

**fast = fast.next.next;**

**slow = slow.next;**

**//if fast and slow pointers are meeting then LinkedList is cyclic**

**if (fast == slow) {**

**return true;**

**}**

**}**

**return false;**

**}**

public void print() {

for (Node p = front; p != null; p = p.next) {

System.out.print(p.element + "\t");

}

}

**//Method to create a Loop inside a LinkedList**

**public void createALoop() {**

**front.next.next.next.next.next.next = front.next.next;**

**}**

public static void main(String[] args) {

LoopInLinkedList sl = new LoopInLinkedList();

for (int i = 0; i < 10; i++) sl.addLast(new Integer(i));

sl.print();

System.out.println("\nIs SL Is cyclic :::" + sl.isCyclic());

sl.createALoop();

System.out.println("\nIs SL Is cyclic :::" + sl.isCyclic());

}

}

**A simple LinkedList**

**public** **class** MyLinkedList {

// reference to the head node.

**private** Node head;

**private** **int** listCount;

**private** **class** Node {

**private** Node next; // reference to the next node in the chain

**private** Object data; // data carried by this node.

**public** Node(Object data) {

**this**.next = **null**;

**this**.data = data;

}

}

**public** MyLinkedList() {

// this is an empty list, so the reference to the head node

// is set to a new node with no data

head = **new** Node(**null**);

listCount = 0;

}

**public** **void** add(Object data) {

// post: appends the specified element to the end of this list.

Node temp = **new** Node(data);

Node currentNode = head;

// starting at the head node, crawl to the end of the list

**while**(currentNode.next != **null**) {

currentNode = currentNode.next;

}

// the last node's "next" reference set to our new node

currentNode.next = temp;

listCount++;// increment the number of elements variable

}

**public** **void** add1( Object x ) {

**if**( head == **null** ) head = **new** Node(x);

**else {**

Node current = head;

**for**( Node p = current ; p != **null** ; p = p.next ) {

current = p;

}

current.next = **new** Node(x);

}

listCount++;// increment the number of elements variable

}

**public** **int** size() {

**return** listCount;

}

**public** Object get(**int** index) {

// post: returns the element at the specified position in this list.

// index must be 1 or higher

**if**(index < 0) **return** **null**;

Node current = head.next;

**for**(**int** i = 0; i < index; i++) {

**if**(current.next == **null**)

**return** **null**;

current = current.next;

}

**return** current.data;

}

**public** **boolean** remove(**int** index) {

// post: removes the element at the specified position in this list.

// if the index is out of range, exit

**if**(index < 0 || index > size())

**return** **false**;

Node currentNode = head;

**for**(**int** i = 0; i < index; i++) {

**if**(currentNode.next == **null**)

**return** **false**;

currentNode = currentNode.next;

}

currentNode.next = currentNode.next.next;

listCount--; // decrement the number of elements variable

**return** **true**;

}

**public** String toString() {

Node current = head.next;

String output = "";

**while**(current != **null**) {

output += "[" + current.data.toString() + "]";

current = current.next;

}

**return** output;

}

**public** **static** **void** main(String[] args) {

MyLinkedList list = **new** MyLinkedList();

list.add1("abcd"); list.add1("bcd"); list.add1("pqrs");

System.*out*.println(list);

**for**( **int** i = 0 ; i < list.size() ; i++ )

System.*out*.print(list.get(i)+" ");

list.remove(1);

System.*out*.println("\n"+list);

}

}

**Write your own iterator for an array**

**import** java.util.ArrayList;  
**import** java.util.Iterator;  
  
**public class** ArrayIterator **implements** Iterable {  
  
 **private** Object[] **elements**;  
 **private int size**;  
 **private int counter** = 0;  
  
 **public** ArrayIterator() {  
  
 **elements** = **new** Object[20];  
 }  
  
 **public void** add( Object x ) {  
 **elements**[**size**++] = x;  
 }  
  
 **@Override  
 public Iterator iterator() {  
 *//reset the counter  
 //If you do not reset the counter, you will not be iterate once again* counter = 0;  
 return new MyIterator();  
 }**  
 **private class MyIterator implements Iterator {  
 @Override  
 public boolean hasNext() {  
 return counter < elements.length && elements[counter] != null ;  
 }  
  
 @Override  
 public Object next() {  
 return elements[counter++] ;  
 }  
  
 @Override  
 public void remove() {  
 System.*out*.println("Don't want to delete item");  
 }  
 }**

**public static void** main(String[] args) {  
  
 ArrayIterator arr = **new** ArrayIterator();  
 **for**( **int** i = 0 ; i < 10 ; i++ ) {  
 arr.add( **new** Integer(i));  
 }  
  
 Iterator itr = arr.iterator();  
 **while**( itr.hasNext() ) {  
 System.***out***.print(**"\t"**+itr.next()); *//0 1 2 3 4 5 6 7 8 9* }  
 System.***out***.println(**"\n\n"**);  
 itr = arr.iterator();  
 **while**( itr.hasNext() ) {  
 System.***out***.print(**"\t"** + itr.next()); *//0 1 2 3 4 5 6 7 8 9* }  
  
 *//In case of arraylist also, everytime, you get an iterator,  
 //the index is et to 0 so that it can be iterated.* ArrayList al = **new** ArrayList();  
 **for**( **int** i = 0 ; i < 10 ; i++ ) {  
 al.add( **new** Integer(i));  
 }  
 System.***out***.println(**"\n\n"**);  
 itr = arr.iterator();  
 **while**( itr.hasNext() ) {  
  
 System.***out***.print(**"\t"**+itr.next());  
 }  
  
 System.***out***.println(**"\n\n"**);  
 itr = arr.iterator();  
 **while**( itr.hasNext() ) {  
 System.***out***.print(**"\t"**+itr.next());  
 }  
 }  
}

**Tree Stucture in Java**

In case of tree structure we can get the child information and its parent. Sometimes interviewer ask questions like how will design a structure so that for a particular employee, we can get its Manager and its children. Interviewer will ask the question that how will you design your object model to accommodate company hierarchy. Example, Managing Director or President of the Company has CEO,CTO,COO, each CEO has Delivery Head, each Delivery Head has List of Project managers etc. At any point of time you can get the Parent and the Children.

**import** java.util.List;

**import** java.util.ArrayList;

**public** **class** TreeNode {

**private** TreeNode parent = **null**;

**private** List children = **null**;

**private** Object reference;

**public** TreeNode(Object obj) {

**this**.parent = **null**;

**this**.reference = obj;

**this**.children = **new** ArrayList();

}

**public** **void** addChildNode(TreeNode child) {

**child.parent = this;**

**if (!children.contains(child))**

**children.add(child);**

}

**public** List getChildren() {

**return** children;

}

**public** TreeNode getParent() {

**return** parent;

}

**public** String toString() {

**return** reference.toString();

}

**public** **static** **void** main(String[] args) {

TreeNode root = **new** TreeNode( "Root" ); TreeNode child1 = **new** TreeNode("Child1");

root.addChildNode(child1);

TreeNode child2 = **new** TreeNode("Child2");

root.addChildNode(child2);

TreeNode grandChild1 = **new** TreeNode("GrandChild1");

child1.addChildNode(grandChild1);

TreeNode grandChild2 = **new** TreeNode("GrandChild2");

child2.addChildNode(grandChild2);

List list = root.getChildren();

**for**( **int** i = 0 ; i < list.size() ; i++ ) {

TreeNode tt = (TreeNode) list.get(i);

System.*out*.println(tt+"----"+tt.getParent());

List list1 = tt.getChildren();

**for**( **int** j = 0 ; j < list1.size() ; j++ ) {

TreeNode grand = (TreeNode) list1.get(j);

System.*out*.println(grand+"----"+grand.getParent());

}

}

}

}

**Another example of similar kind is given below**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** EmpNode

{

**private** EmpNode parentEmp;

**private** List childrenEmps;

**private** String name;

**private** String id;

**public** EmpNode( String id , String name ) {

**this**.parentEmp = **null**;

**this**.name = name;

**this**.id = id;

**this**.childrenEmps = **new** ArrayList();

}

**public** EmpNode getParent() {

**return** parentEmp;

}

**public** List getChildren() {

**return** childrenEmps;

}

**public** **void** addEmp( EmpNode childEmp ) {

**childEmp.parentEmp = this;**

**if( !childrenEmps.contains(childEmp) )**

**childrenEmps.add(childEmp);**

}

**public** String toString() {

**return** id+"---"+name;

}

**public** **static** **void** main(String[] args) {

EmpNode root = **new** EmpNode( "1","Root" ); EmpNode child1 = **new** EmpNode("2","Child1");

root.addEmp(child1);

EmpNode child2 = **new** EmpNode("3","Child2");

root.addEmp(child2);

EmpNode grandChild1 = **new** EmpNode("4","GrandChild1");

child1.addEmp(grandChild1);

EmpNode grandChild2 = **new** EmpNode("5","GrandChild2");

child2.addEmp(grandChild2);

List list = root.getChildren();

**for**( **int** i = 0 ; i < list.size() ; i++ ) {

EmpNode tt = (EmpNode) list.get(i);

System.*out*.println(tt+"----"+tt.getParent());

List list1 = tt.getChildren();

**for**( **int** j = 0 ; j < list1.size() ; j++ ) {

EmpNode grand = (EmpNode) list1.get(j);

System.*out*.println(grand+"----"+grand.getParent());

}

}

}

}

**Array Based Stack**  
  
**import** java.lang.reflect.Array;  
**import** java.util.Arrays;  
**public class** ArrayBasedStack<E> { *//LIFO*  
 **private int defaultCapacity** = 10;  
 **private int size**;  
 **private** Object[] **elements**;  
  
 **public** ArrayBasedStack() {  
 **elements** = **new** Object[**defaultCapacity**] ;  
 }  
  
 **private void** ensureCapacity() {  
 **elements** = Arrays.*copyOf*( **elements**, **size**\*2 );  
 }  
  
 **public void** push( E e ) {  
 **if**( **size** == **elements**.**length**) ensureCapacity();  
 **elements**[**size**++] = e;  
 }  
  
 **public int** getSize() {  
 **return size**;  
 }  
  
 **public** E pop() {  
 E e = (E) **elements**[ --**size** ];  
 **elements**[**size**] = **null**;  
 **return** e;  
 }  
  
 **public** E getElement( **int** index ) {  
 **return** (E) **elements**[index];  
 }  
  
 **public static void** main(String[] args) {  
 ArrayBasedStack stack = **new** ArrayBasedStack();  
 stack.push(**"1"**); stack.push(**"4"**);  
 System.***out***.println(stack.pop());  
 System.***out***.println(stack.pop());  
 **for** (**int** i = 5; i < 25; i++) {  
 stack.push( String.*valueOf*(i) );  
 }  
 **for** (**int** i = 0; i < stack.getSize(); i++) {  
 System.***out***.println(**"Element --->"**+stack.getElement(i));  
 }  
 }  
}

**Array Based Queue**

**import** java.util.Arrays;  
**public class** ArrayBasedQueue<E> { *// FIFO* **private int defaultCapacity** = 10;  
 **private int size**;  
 **private** Object[] **elements**;  
  
 **public** ArrayBasedQueue() {  
 **elements** = **new** Object[**defaultCapacity**];  
 }  
  
 **private void** ensureCapacity() {  
 **elements** = Arrays.*copyOf*(**elements**, **size** \* 2);  
 }  
  
 **public void** offer(E e) {  
 **if** (**size** == **elements**.**length**)  
 ensureCapacity();  
 **elements**[**size**++] = e;  
 }  
  
 **public int** getSize() {  
 **return size**;  
 }  
  
 **private int front** = 0;  
 **public** E poll() {  
 **front**++;  
 E e = (E) **elements**[**front** - 1];  
 **size**--;  
 **return** e;  
 }  
  
 **public** E getElement(**int** index) {  
 **return** (E) **elements**[index];  
 }  
  
 **public static void** main(String[] args) {  
 ArrayBasedQueue queue = **new** ArrayBasedQueue();  
 queue.offer(**"1"**);  
 queue.offer(**"4"**);  
 System.***out***.println(queue.poll());  
 System.***out***.println(queue.poll());  
 **for** (**int** i = 5; i < 25; i++) {  
 queue.offer(String.*valueOf*(i) );  
 }  
 **for** (**int** i = 0; i < queue.getSize(); i++)

System.out.println("Element --->" + queue.getElement(i));  
 }  
}

**Node based Queue**

**public class** NodeBasedQueue {  
  
 **private** Node **front**;  
 **private** Node **back**;  
  
 **private static class** Node {  
 **public** Object **element**;  
 **public** Node **next**;  
  
 **public** Node(Object element) {  
 **this**.**element** = element;  
 **next** = **null**;  
 }  
 }  
  
 **public void** offer(Object x) { *//enqueue()* **if**( **front** == **null** ) **back** = **front** = **new** Node(x);  
 **else  
 back** = **back**.**next** = **new** Node(x);  
 }  
  
 **public** Object poll() {  
 **if**( **front** == **null** ) **throw new** NullPointerException(**"No element found"**);  
 Object returnValue = **front**.**element**;  
 **front** = **front**.**next**;  
 **return** returnValue;  
 }  
  
 **public** Object getFront() {  
 **if**( **front** == **null** ) **throw new** NullPointerException(**"No element found"**);  
 **return front**.**element**;  
 }  
  
 **public void** makeQueueEmpty() {  
 **front** = **null**;  
 **back** = **null**;  
 }  
  
 **public void** print() {  
 **for**( Node p = **front** ; p != **null** ; p = p.**next**)  
 System.***out***.println(**"Element --->"**+p.**element**);  
 }  
  
 **public static void** main(String[] args) {  
 NodeBasedQueue queue = **new** NodeBasedQueue();  
 queue.offer(**"1"**); queue.offer(**"2"**); queue.offer(**"3"**); queue.offer(**"4"**);  
 System.***out***.println(queue.poll());  
 System.***out***.println(queue.poll());  
 queue.print();  
 }  
}

**Node based Stack**

**public class** NodeBasedStack {  
 **private** Node **front**;  
 **private** Node **back**;  
  
 **private static class** Node {  
 **public** Object **element**;  
 **public** Node **next**;  
  
 **public** Node(Object element) {  
 **this**.**element** = element;  
 **next** = **null**;  
 }  
 }  
  
 **public void** push(Object x) { *//enqueue()* **if**( **front** == **null** ) **back** = **front** = **new** Node(x);  
 **else  
 back** = **back**.**next** = **new** Node(x);  
 }  
  
 **public** Object pop() {  
 **if**( **front** == **null** ) **throw new** NullPointerException(**"No element found"**);  
  
 Object returnValue = **back**.**element**;  
 **if**( **front**.equals(**back**))  
 **front** = **back** = **null**;  
 **else** {  
 Node current = **front**;  
 **while**( current.**next** != **back** )  
 current = current.**next**;  
 **back** = current;  
 current.**next** = **null**;  
 }  
 **return** returnValue;  
 }  
  
 **public** Object getFront() {  
 **if**( **front** == **null** ) **throw new** NullPointerException(**"No element found"**);  
 **return front**.**element**;  
 }  
  
 **public void** makeQueueEmpty() {  
 **front** = **null**;  
 **back** = **null**;  
 }  
  
 **public void** print() {  
 **for**( Node p = **front** ; p != **null** ; p = p.**next**)  
 System.***out***.println(**"Element --->"**+p.**element**);  
 }  
  
 **public static void** main(String[] args) {  
  
 NodeBasedStack stack = **new** NodeBasedStack();  
 stack.push(**"1"**);  
 stack.push(**"2"**);  
 stack.push(**"3"**);  
 stack.push(**"4"**);  
  
 System.***out***.println(stack.pop());  
 System.***out***.println(stack.pop());  
  
 **for** (**int** i = 5; i < 25; i++) {  
 String val = String.*valueOf*(i);  
 stack.push(val);  
 }  
  
 stack.print();  
 }  
}

**HashCode and Equals in an Object**

How to write hashcode for an object

**public** **enum** EmpType {

***PERMANENT***,***TEMPORARY***

}

**import** java.util.Arrays;

**public** **class** Employee {

**private** String stringVal;

**private** **byte** byteVal;

**private** **short** shortVal;

**private** **int** intVal;

**private** **float** floatVal;

**private** **double** doubleVal;

**private** **long** longVal;

**private** EmpType enumType; // Permanent or Temporary

**private** **char** ch;

**private** **byte**[] byteArrVal;

**private** String[] strArrVal;

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + Arrays.*hashCode*(byteArrVal);

result = prime \* result + byteVal;

result = prime \* result + ch;

**long** temp;

temp = Double.*doubleToLongBits*(doubleVal);

result = prime \* result + (**int**) (temp ^ (temp >>> 32));

result = prime \* result + ((enumType == **null**) ? 0 : enumType.hashCode());

result = prime \* result + Float.*floatToIntBits*(floatVal);

result = prime \* result + intVal;

result = prime \* result + (**int**) (longVal ^ (longVal >>> 32));

result = prime \* result + shortVal;

result = prime \* result + Arrays.*hashCode*(strArrVal);

result = prime \* result + ((stringVal == **null**) ? 0 : stringVal.hashCode());

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Employee other = (Employee) obj;

**if** (!Arrays.*equals*(byteArrVal, other.byteArrVal))

**return** **false**;

**if** (byteVal != other.byteVal)

**return** **false**;

**if** (ch != other.ch)

**return** **false**;

**if** (Double.*doubleToLongBits*(doubleVal) != Double.*doubleToLongBits*(other.doubleVal))

**return** **false**;

**if** (enumType != other.enumType)

**return** **false**;

**if** (Float.*floatToIntBits*(floatVal) != Float.*floatToIntBits*(other.floatVal))

**return** **false**;

**if** (intVal != other.intVal)

**return** **false**;

**if** (longVal != other.longVal)

**return** **false**;

**if** (shortVal != other.shortVal)

**return** **false**;

**if** (!Arrays.*equals*(strArrVal, other.strArrVal))

**return** **false**;

**if** (stringVal == **null**) {

**if** (other.stringVal != **null**)

**return** **false**;

} **else** **if** (!stringVal.equals(other.stringVal))

**return** **false**;

**return** **true**;

}

}

# **Key points for Hashcode**

**final int prime = 31;**

**int result = 1;**

**Arrays.*hashCode*(byteArrVal); //For byte[]**

**prime \* result + ch; // For char type**

//For double type value

**long temp = Double.*doubleToLongBits*(doubleVal);**

**result = prime \* result + (int) (temp ^ (temp >>> 32));**

//For enum

**result = prime \* result + ((enumType == null) ? 0 : enumType.hashCode());**

**Always remember an enum gives a random value for hashcode.**

//For float type value

**result = prime \* result + Float.*floatToIntBits*(floatVal);**

//For long type value

**result = prime \* result + (int) (longVal ^ (longVal >>> 32));**

//For short type value

**result = prime \* result + shortVal;**

//For String[]

**Arrays.*hashCode*(strArrVal);**

//For String

**result = prime \* result + ((stringVal == null) ? 0 : stringVal.hashCode());**

# **Key Points for equals()**

**public** **boolean** equals(Object obj) {

**Always do the first operation**

1. **if (this == obj) return true;**
2. **if (obj == null) return false;**
3. **if (getClass() != obj.getClass()) return false;**

**Second try to compare each field**

1. Employee other = (Employee) obj;
2. For byte[]

**if (!Arrays.equals(byteArrVal, other.byteArrVal)) return false;**

1. For byte value

**if (byteVal != other.byteVal) return false;**

1. For char type value

**if (ch != other.ch) return false;**

1. For double type value

**if(Double.doubleToLongBits(doubleVal) != Double.doubleToLongBits(other.doubleVal))**

**return false;**

1. For float type value

**if(Float.floatToIntBits(floatVal) != Float.floatToIntBits(other.floatVal))**

**return false;**

1. For enum type

**if (enumType != other.enumType) return false;**

1. For int type value

**if (intVal != other.intVal) return false;**

1. For long type value

**if (longVal != other.longVal) return false;**

1. For short type value

**if (shortVal != other.shortVal) return false;**

1. For String[] type

**if (!Arrays.equals(strArrVal, other.strArrVal)) return false;**

1. For normal String value

**if (stringVal == null) {**

**if (other.stringVal != null)**

**return false;**

**} else if (!stringVal.equals(other.stringVal))**

**return false;**

**Points to Remember**

For Any other Array, use the below

**if (!Arrays.equals(ArrVal, other.ArrVal)) return false;**

For Double type value

**if (Double.doubleToLongBits(doubleVal) != Double.doubleToLongBits(other.doubleVal))**

**return false;**

For float type value

**if(Float.floatToIntBits(floatVal) != Float.floatToIntBits(other.floatVal))**

**return false;**

For char type value

**if (ch != other.ch) return false;**

For enum type value

**if (enumType != other.enumType) return false;**

Reference: The hashcode and equals generated by Intellij Idea is given below for the above class.

**public int** hashCode() {  
 **int** result;  
 **long** temp;  
 result = **stringVal** != **null** ? **stringVal**.hashCode() : 0;  
 result = 31 \* result + (**int**) **byteVal**;  
 result = 31 \* result + (**int**) **shortVal**;  
 result = 31 \* result + **intVal**;  
 result = 31 \* result + (**floatVal** != +0.0f ? Float.*floatToIntBits*(**floatVal**) : 0);  
 temp = Double.*doubleToLongBits*(**doubleVal**);  
 result = 31 \* result + (**int**) (temp ^ (temp >>> 32));  
 result = 31 \* result + (**int**) (**longVal** ^ (**longVal** >>> 32));  
 result = 31 \* result + (**enumType** != **null** ? **enumType**.hashCode() : 0);  
 result = 31 \* result + (**int**) **ch**;  
 result = 31 \* result + (**byteArrVal** != **null** ? Arrays.*hashCode*(**byteArrVal**) : 0);  
 result = 31 \* result + (**strArrVal** != **null** ? Arrays.*hashCode*(**strArrVal**) : 0);  
 **return** result;  
}

**public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
  
 Employee employee = (Employee) o;  
  
 **if** (**byteVal** != employee.**byteVal**) **return false**;  
 **if** (**shortVal** != employee.**shortVal**) **return false**;  
 **if** (**intVal** != employee.**intVal**) **return false**;  
 **if** (Float.*compare*(employee.**floatVal**, **floatVal**) != 0) **return false**;  
 **if** (Double.*compare*(employee.**doubleVal**, **doubleVal**) != 0) **return false**;  
 **if** (**longVal** != employee.**longVal**) **return false**;  
 **if** (**ch** != employee.**ch**) **return false**;  
 **if** (**stringVal** != **null** ? !**stringVal**.equals(employee.**stringVal**) : employee.**stringVal** != **null**) **return false**;  
 **if** (**enumType** != employee.**enumType**) **return false**;  
 **if** (!Arrays.*equals*(**byteArrVal**, employee.**byteArrVal**)) **return false**;  
 *// Probably incorrect - comparing Object[] arrays with Arrays.equals* **return** Arrays.*equals*(**strArrVal**, employee.**strArrVal**);  
  
}

**Example on Negative HashCode**

**class Emp {  
  
 private int sal = Integer.*MAX\_VALUE*;*//It can be MIN\_VALUE* private String name;  
  
 public Emp( String name ) {  
 this.name = name;  
 }**

**}**

**It will give negative hashcode as shown below.**

Emp emp = **new** Emp(**"a"**);  
System.***out***.print(**"Emp HashCode :::"**+emp.hashCode());

**Hashing function in Map in Java**

//**As of JDK 6**  
static int hash(int h) {

// This function ensures that hashCodes that differ only by  
// constant multiples at each bit position have a bounded  
// number of collisions (approximately 8 at default load factor).

**h ^= (h >>> 20) ^ (h >>> 12);  
 return h ^ (h >>> 7) ^ (h >>> 4);  
}**

//**As of JDK 7**  
final int hash(Object k) {

int h = hashSeed;  
**if (0 != h && k instanceof String) {  
 return sun.misc.Hashing.stringHash32((String) k);  
}**  
**h ^= k.hashCode();**// This function ensures that hashCodes that differ only by  
// constant multiples at each bit position have a bounded  
// number of collisions (approximately 8 at default load factor).  
  
**h ^= (h >>> 20) ^ (h >>> 12);  
 return h ^ (h >>> 7) ^ (h >>> 4);  
}**

### How does Java 6 hashing work?

java.lang.String#hashCode() is calculated by iterating over each character in the string, and executing this function:

**h = 31\*h + val[i]**

Now let’s examine how the hash code is used by a popular data structure: HashMap. Looking at java.util.HashMap#put(K,V) runs HashMap#hash(int) on the key’s hashCode. That function is a supplemental hash that provides some protection against bad hash function.

**h ^= (h >>> 20) ^ (h >>> 12);**  
**return h ^ (h >>> 7) ^ (h >>> 4);**

This provides no randomization or protection from [**collision attacks**](http://en.wikipedia.org/wiki/Collision_attack). In Java 6, every JVM running everywhere follows these 2 mathematical functions. Some might think, Java hash code should always be the same whenever you call hashCode on the same string, but that’s only true within a single invocation of the JVM. There is no requirement that if 2 different JVM instances running should resolve to the same hashCode for the same string.

### Java 7 finally fixed hashing.

java.lang.String has add a static initializer for the single purpose of creating a**HASHING\_SEED**. This is brand new for Java 7. The “seed materials” used to randomize hash codes so that different JVMs don’t have predictable hash codes are: System.currentTimeMillis and System.nanoTime and others. But wait, if you look at java.lang.String#hashCode() in Java 7 source ***it looks identical to Java 6’s implementation!***So let’s checkout java.util.HashMap in Java 7, and we find that the hash() function there has changed. It takes an Object instead of an int, and if the Object is an instanceof String, then through **sun.misc.Hashing.stringHash32()**it actually called a new method insidejava.lang.String#hash32(). The hash32() function passes the character data plus the HASHING\_SEED computed above to **sun.misc.Hashing.murmur3\_32(HASHING\_SEED, value, 0, value.length)**

The [**Murmur3 hashing function**](http://en.wikipedia.org/wiki/MurmurHash) with the hashing seed data which is based on the time that java.lang.String’s static initializer is executed provides excellent protection against collision attacks.

public class Hash {

public static void main(String[] args) {

String str = "Java Programming Language";

System.out.println(str + " hashCode(): "

+ str.hashCode()

+ " new Java 7 hashCode used by collections: "

+ sun.misc.Hashing.stringHash32(str));

}

}

The above provide two different hashcode, str.hashCode() always gives constant value where as sun.misc.Hashing.stringHash32(str) gives random value.

Internals of Java Data Structures

**About ArrayList**

**How to get the capacity of ArrayList**

//How to get the capcity of ArrayList using reflection

**static** **int** getCapacity(ArrayList<?> list) **throws** Exception {

Field dataField = ArrayList.**class**.getDeclaredField("elementData");

dataField.setAccessible(**true**);

**return** ((Object[]) dataField.get(list)).length;

}

**How will you get deep copy and shallow of ArrayList**

Shallow copy can be obtained using list.clone() method.

Deep copy can be achieved using serializing and deserializing

ByteArrayOutputStream bos = **new** ByteArrayOutputStream();

ObjectOutputStream o = **new** ObjectOutputStream(bos);

o.writeObject(list);

**byte**[] yourBytes = bos.toByteArray();

InputStream in = **new** ByteArrayInputStream(yourBytes);

ObjectInputStream oin = **new** ObjectInputStream(in);

ArrayList<Employee> deepCopyList = (ArrayList<Employee>) oin.readObject();

**How arrayList grows internally**

**As of JDK 6 or prior to JDK 7**

**int oldCapacity = elementData.length;**

**int newCapacity = (oldCapacity \* 3)/2 + 1;**

**elementData = Arrays.copyOf(elementData, newCapacity);**

public void ensureCapacity(int minCapacity) {

modCount++;

int oldCapacity = elementData.length;

if (minCapacity > oldCapacity) {

Object oldData[] = elementData;

int newCapacity = (oldCapacity \* 3)/2 + 1;

if (newCapacity < minCapacity)

newCapacity = minCapacity;

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

}

**As per JDK 7**

**int oldCapacity = elementData.length;**

**int newCapacity = oldCapacity + (oldCapacity >> 1);**

**elementData = Arrays.copyOf(elementData, newCapacity);**

private void grow(int minCapacity) {

// overflow-conscious code

int oldCapacity = elementData.length;

int newCapacity = oldCapacity + (oldCapacity >> 1);

if (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

newCapacity = hugeCapacity(minCapacity);

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

private static int hugeCapacity(int minCapacity) {

if (minCapacity < 0) // overflow

throw new OutOfMemoryError();

return (minCapacity > MAX\_ARRAY\_SIZE) ? Integer.MAX\_VALUE : MAX\_ARRAY\_SIZE;

}

**How hashcode is calculated in case of ArrayList ?**

The code is given below. It is the part of AbtractList.

**public** **int** hashCode() {

**int** hashCode = 1;

**for** (E e : **this**)

hashCode = 31\*hashCode + (e==**null** ? 0 : e.hashCode());

**return** hashCode;

}

Basically it calculates the hashcode of each object present inside the List.

**How equals method works for ArrayList**

The code is given below. It is the part of AbtractList.

**public** **boolean** equals(Object o) {

**if** (o == **this**)

**return** **true**;

**if** (!(o **instanceof** List))

**return** **false**;

ListIterator<E> e1 = listIterator();

ListIterator e2 = ((List) o).listIterator();

**while** (e1.hasNext() && e2.hasNext()) {

E o1 = e1.next();

Object o2 = e2.next();

**if** (!(o1==**null** ? o2==**null** : o1.equals(o2)))

**return** **false**;

}

**return** !(e1.hasNext() || e2.hasNext());

}

**What is difference between System.arrayCopy and Arrays.copyOf ?**

The difference is that Arrays.copyOf does not only copy elements, it also creates a new array. System.arrayCopy copies into an existing array. Arrays.copyOf also internally uses System.arrayCopy.

**Internals of HashSet**

HashSet internally uses HashMap as the code is given below.

**private** **transient** HashMap<E,Object> map;

**private** **static** **final** Object ***PRESENT*** = **new** Object();

The internal HashMap used “PRESENT” as seen above as a dummy object.

In case of LinkedHashSet, the constructor is given below.

**HashSet(int initialCapacity, float loadFactor, boolean dummy) {**

**map = new LinkedHashMap<>(initialCapacity, loadFactor);**

**}**

**The question may be how LinkedHashSet works internally. The answer is LinkedHashSet internally uses LinkedHashMap.**

**The following methods are given about how HashSet internally works.**

**To get size :: map.size();**

**To check hashSet is empty : map.isEmpty();**

**To check whether hashset contains an object : map.containsKey(o);**

**To add object to hashset : map.put(e, *PRESENT*)==null;**

**How Serialization and Deserialization works in HashSet ?**

**The code is given below.**

**private** **void** writeObject(java.io.ObjectOutputStream s)

**throws** java.io.IOException {

// Write out any hidden serialization magic

s.defaultWriteObject();

// Write out HashMap capacity and load factor

s.writeInt(map.capacity());

s.writeFloat(map.loadFactor());

// Write out size

s.writeInt(map.size());

// Write out all elements in the proper order.

**for** (E e : map.keySet())

s.writeObject(e);

}

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException {

// Read in any hidden serialization magic

s.defaultReadObject();

// Read in HashMap capacity and load factor and create backing HashMap

**int** capacity = s.readInt();

**float** loadFactor = s.readFloat();

map = (((HashSet)**this**) **instanceof** LinkedHashSet ?

**new** LinkedHashMap<E,Object>(capacity, loadFactor) :

**new** HashMap<E,Object>(capacity, loadFactor));

// Read in size

**int** size = s.readInt();

// Read in all elements in the proper order.

**for** (**int** i=0; i<size; i++) {

E e = (E) s.readObject();

map.put(e, ***PRESENT***);

}

}

**How clone() method works in HashSet ?**

It returns the shallow copy of the HashSet.

**public** Object clone() {

**try** {

HashSet<E> newSet = (HashSet<E>) **super**.clone();

newSet.map = (HashMap<E, Object>) map.clone();

**return** newSet;

} **catch** (CloneNotSupportedException e) {

**throw** **new** InternalError();

}

}

**How equals() and hashcode() method works in HashSet ?**

The equals method is given below. These methods are part of AbstractSet

**public** **boolean** equals(Object o) {

**if** (o == **this**)

**return** **true**;

**if** (!(o **instanceof** Set))

**return** **false**;

Collection c = (Collection) o;

**if** (c.size() != size())

**return** **false**;

**try** {

**return** containsAll(c);

} **catch** (ClassCastException unused) {

**return** **false**;

} **catch** (NullPointerException unused) {

**return** **false**;

}

}

The hashcode method is given below.

**public** **int** hashCode() {

**int** h = 0;

Iterator<E> i = iterator();

**while** (i.hasNext()) {

E obj = i.next();

**if** (obj != **null**)

h += obj.hashCode();

}

**return** h;

}

How TreeSet works internally

The code is given below.

**private** **transient** NavigableMap<E,Object> m;

**private** **static** **final** Object ***PRESENT*** = **new** Object();

**public** TreeSet() {

**this**(**new** TreeMap<E,Object>());

}

**public** TreeSet(Comparator<? **super** E> comparator) {

**this**(**new** TreeMap<>(comparator));

}

From the above code it is obvious that TreeSet internally uses TreeMap.

\*\* The equals(), clone(), hashcode() methods work like HashSet.

**How LinkedHashSet internally works ?**

LinkedHashSet internally uses HashSet, methods and constructors are inherited.

\*\* TreeSet internally uses TreeMap which has been developed on the base line of Red and Black Tree algorithm.

**How HashMap put() method works internally ?**

Always remember that, the initial capacity and threashold in case of HashMap is 16, 2 << 4.

**The load factor is 0.75f.**

The constructor for HashMap is given below.

public HashMap() {

this(DEFAULT\_INITIAL\_CAPACITY, DEFAULT\_LOAD\_FACTOR);

}

The source code for put method given below.

**public** V put(K key, V value) {

**if** (table == ***EMPTY\_TABLE***) {

**inflateTable(threshold);**

}

**if** (key == **null**)

**return** putForNullKey(value);

**int hash = hash(key);**

**int i = *indexFor*(hash, table.length);**

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {

Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

**return** **null**;

}

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

**if** ((size >= threshold) && (**null** != table[bucketIndex])) {

**resize(2 \* table.length);**

hash = (**null** != key) ? hash(key) : 0;

bucketIndex = *indexFor*(hash, table.length);

}

createEntry(hash, key, value, bucketIndex);

}

**void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {

Entry<K,V> e = table[bucketIndex];

table[bucketIndex] = **new** Entry<>(hash, key, value, e);

size++;

}

Steps of Put Method

1. Get the hashcode of the Key
2. Calculate the hash value of the hashcode
3. Find the index of the bucket using the following function

**static** **int** indexFor(**int** h, **int** length) {

**return** h & (length-1);

}

1. Iterate the entire bucket and check whether key is already present or not, if present, simply modify the value or update the value.
2. If the key is not present, check the bucket has been filled up or not.
3. If the bucket has been filled up, increase the size of the Entry object array as double, the code is given below.

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

**if** ((size >= threshold) && (**null** != table[bucketIndex])) {

**resize(2 \* table.length);**

hash = (**null** != key) ? hash(key) : 0;

bucketIndex = *indexFor*(hash, table.length);

}

createEntry(hash, key, value, bucketIndex);

}

**void resize(int newCapacity)** {

**Entry[] oldTable = table;**

**int oldCapacity = oldTable.length;**

**if (oldCapacity == *MAXIMUM\_CAPACITY*) {**

**threshold = Integer.*MAX\_VALUE*;**

**return;**

**}**

**Entry[] newTable = new Entry[newCapacity];**

**transfer(newTable, initHashSeedAsNeeded(newCapacity));**

table = newTable;

**threshold = (int)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

}

Let see how it works

* Suppose initial capacity is 16.
* Initial capacity and threshold are same ie 16. Refer to the constructor of hashmap , it

**threshold = initialCapacity;**

**this.loadFactor = loadFactor; // ie 0.75**

* If the size of threshold is greater than or equal to 16 and bucket index has a value, increase the size to double. As it is obvious from the below code.

**if ((size >= threshold) && (null != table[bucketIndex])) {**

**resize(2 \* table.length);**

**In the above case, resize( 2 \* 16 ), so size = 16**

* Now calculate the old capacity which is nothing but the length of the bucket ie

**int oldCapacity = oldTable.length;**

* Check if the old capacity is greater than Maximum Size ie

**static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 30; //**1073741824**

* A new array of size 32 will be created.
* Now new threshold value will be calculated as

**threshold = (int)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

**threshold = (int)Math.*min*(32 \* 0.75, 1073741824 + 1);**

**threshold = (int)Math.*min*(32 \* 0.75, 1073741824 + 1);**

**threshold = (int) Math.min(24, 1073741824 + 1);**

**threshold = 24; //Finally**

How hashmap reacts when you pass the your capacity in hashmap ? How capacity is calculated ?

Let us take an example below.

**HashMap hm = new HashMap(7);**

Now in this case, the initial capacity will be 7, load factor will be 0.75. The code is given below.

**public** HashMap(**int** initialCapacity) {

**this**(initialCapacity, ***DEFAULT\_LOAD\_FACTOR***);

}

**public** HashMap(**int** initialCapacity, **float** loadFactor) {

**if** (initialCapacity < 0)

**throw** **new** IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

**if** (initialCapacity > ***MAXIMUM\_CAPACITY***)

initialCapacity = ***MAXIMUM\_CAPACITY***;

**if** (loadFactor <= 0 || Float.*isNaN*(loadFactor))

**throw** **new** IllegalArgumentException("Illegal load factor: " +

loadFactor);

**this**.loadFactor = loadFactor;

threshold = initialCapacity;

init();

}

Now in this case the growth of HashMap is bit different. Let us see below. Now again see the put method.

**public** V put(K key, V value) {

**if (table == *EMPTY\_TABLE*) {**

**inflateTable(threshold);**

**}**

**if** (key == **null**)

**return** putForNullKey(value);

**int** hash = hash(key);

**int** i = *indexFor*(hash, table.length);

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {

Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

**return** **null**;

}

Let us see the inflateTable() method.

**private** **void** inflateTable(**int** toSize) {

// Find a power of 2 >= toSize

**int capacity = *roundUpToPowerOf2*(toSize);**

**threshold = (int) Math.*min*(capacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

**table = new Entry[capacity];**

initHashSeedAsNeeded(capacity);

}

The above method is used to create initial array of Entry object. Let us see how threshold is calculated.

**int capacity = *roundUpToPowerOf2*(toSize); The method is given below.**

**private** **static** **int** roundUpToPowerOf2(**int** number) {

// assert number >= 0 : "number must be non-negative";

**return** number >= ***MAXIMUM\_CAPACITY***

? ***MAXIMUM\_CAPACITY***

: (number > 1) ? Integer.*highestOneBit*((number - 1) << 1) : 1;

}

What is Integer.*highestOneBit*((number - 1) << 1) : 1; The code is given below.

**public static int** highestOneBit(**int** i) {  
 *// HD, Figure 3-1***i |= (i >> 1);  
 i |= (i >> 2);  
 i |= (i >> 4);  
 i |= (i >> 8);  
 i |= (i >> 16);  
 return i - (i >>> 1);**}

We need to understand about how the above method ***roundUpToPowerOf2*(toSize)** works. Let us consider a small java program and see how numbers are getting generated.

**static final int *MAXIMUM\_CAPACITY*** = 1 << 30;  
**public static int** roundUpToPowerOf2(**int** number) {  
 *// assert number >= 0 : "number must be non-negative";* **return** number >= ***MAXIMUM\_CAPACITY*** ? ***MAXIMUM\_CAPACITY*** : (number > 1) ? Integer.*highestOneBit*((number - 1) << 1) : 1;  
}

System.***out***.println(*roundUpToPowerOf2*(0)); *//1*System.***out***.println(*roundUpToPowerOf2*(1)); *//1*System.***out***.println(*roundUpToPowerOf2*(2)); *//2*System.***out***.println(*roundUpToPowerOf2*(3)); *//4*System.***out***.println(*roundUpToPowerOf2*(4)); *//4*System.***out***.println(*roundUpToPowerOf2*(5)); *//8*System.***out***.println(*roundUpToPowerOf2*(7)); *//8*System.***out***.println(*roundUpToPowerOf2*(9)); *//16*System.***out***.println(*roundUpToPowerOf2*(11)); *//16*System.***out***.println(*roundUpToPowerOf2*(15)); *//16*System.***out***.println(*roundUpToPowerOf2*(16)); *//16*System.***out***.println(*roundUpToPowerOf2*(19)); *//32*System.***out***.println(*roundUpToPowerOf2*(27)); *//32*System.***out***.println(*roundUpToPowerOf2*(32)); *//32*System.***out***.println(*roundUpToPowerOf2*(33)); *//64*System.***out***.println(*roundUpToPowerOf2*(48)); *//64*System.***out***.println(*roundUpToPowerOf2*(65)); *//128*

The mathematical trick is given below.

Choose any number and see how it falls in the below segment.

**… 512 256 128 64 32 16 8 4 2 1**

Let us number 65, it falls in between 64 and 128, so capacity will be 128.

If number is 16, capacity will 16

No threashold calculation

**threshold = (int) Math.*min*(capacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);**

Let us consider, initial capacity is 5, it falls in between 4 and 8, so capacity will be 8.

initialCapacity = 5;

capacity = 8;

Threshold = (int) Math.min(8 \* 0.75 , ***MAXIMUM\_CAPACITY* + 1)**

Threshold = (int) Math.min(6 , ***MAXIMUM\_CAPACITY* + 1)**

Threshold = 6;

It means after 6 entries, the Entry object will grow and all the values will be copied.

**\*\* When the Entry array object is resized, again the hash value is calculated.**

**In case of LinkedHashMap, the following init() is overridden, but method is blank in case of HashMap. The purpose is to create a LinkedList in case of LinkedHashMap. LinkedHashMap extends HashMap.**

**private** **transient** Entry<K,V> header;

**void** init() {

header = **new** Entry<>(-1, **null**, **null**, **null**);

header.before = header.after = header;

}

**Besides the following methods have been overridden in LinkedHashMap.**

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

**super**.addEntry(hash, key, value, bucketIndex);

// Remove eldest entry if instructed

Entry<K,V> eldest = header.after;

**if** (removeEldestEntry(eldest)) {

removeEntryForKey(eldest.key);

}

}

**void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {

HashMap.Entry<K,V> old = table[bucketIndex];

Entry<K,V> e = **new** Entry<>(hash, key, value, old);

table[bucketIndex] = e;

e.addBefore(header);

size++;

}

**void** transfer(HashMap.Entry[] newTable, **boolean** rehash) {

**int** newCapacity = newTable.length;

**for** (Entry<K,V> e = header.after; e != header; e = e.after) {

**if** (rehash)

e.hash = (e.key == **null**) ? 0 : hash(e.key);

**int** index = *indexFor*(e.hash, newCapacity);

e.next = newTable[index];

newTable[index] = e;

}

}

TreeMap does not extend HashMap, it extends AbstractHashMap only.