Basic Data Structure – List Map Stack Queue

Contents

[**Tree Stucture in Java** 2](#_Toc457162222)

[**ArrayList** 4](#_Toc457162223)

[**LinkedList** 6](#_Toc457162224)

[**A simple LinkedList** 12](#_Toc457162225)

[**Detect Loop Inside a LinkedList** 14](#_Toc457162226)

[**Array Based Stack** 15](#_Toc457162227)

[**Array Based Queue** 16](#_Toc457162228)

[**Node based Queue** 17](#_Toc457162229)

[**Node based Stack** 18](#_Toc457162230)

[**Write your own iterator** 21](#_Toc457162231)

[**Difference between Iterable and Iterator** 22](#_Toc457162232)

[**System.arraycopy and Arrays.copyOf** 24](#_Toc457162233)

[**Boyer-Moore Algorithm** 26](#_Toc457162234)

[**Another variation of Boyer-Moore Algorithm** 27](#_Toc457162235)

[**Brute Force Algorithm** 28](#_Toc457162236)

# **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());

}

}

}

}

# **ArrayList**

**package** com.ddlab.rnd.algol;  
**import** java.util.Arrays;  
  
**public class** MyArrayList {  
  
 **private int size**;  
 **private int defaultCapacity** = 10;  
 **private** Object[] **elements**;  
  
 **public** MyArrayList() {  
 **elements** = **new** Object[**defaultCapacity**];  
 }  
  
 **public void** add(Object x) {  
 **if** (**elements**.**length** == **size**)  
 ensureCapacity();  
 **elements**[**size**++] = x;  
 }  
  
 **private void** ensureCapacity() {  
 **int** oldCapacity = **elements**.**length**;  
 **int** newCapacity = oldCapacity + (oldCapacity >> 1);  
 **elements** = Arrays.*copyOf*(**elements**, newCapacity);  
 *//or you can write  
 //elements = Arrays.copyOf(elements, elements.length+(elements.length \* 2));* }  
  
 **public** Object get(**int** index) {  
 **return elements**[index];  
 }  
  
 **public void** delete(Object obj) {  
 **for** (**int** i = 0; i < **elements**.**length**; i++) {  
 **if** (**elements**[i].equals(obj)) {  
 **elements**[i] = **null**;  
 **size**--;  
 condenseArray(i);  
 **break**;  
 }  
 }  
 }  
  
 **public int** size() {  
 **return size**;  
 }  
  
 **private void** condenseArray(**int** start) {  
 **for** (**int** i = start; i < **size**; i++)  
 **elements**[i] = **elements**[i + 1];  
 }  
  
 **public void** trimArray() {  
 **elements** = Arrays.*copyOf*(**elements**, **size**);  
 }

**public static void** main(String[] args) {  
 MyArrayList list = **new** MyArrayList();  
  
 **for** (**int** i = 0; i < 13; i++) {  
 list.add(**new** Integer(i));  
 }  
  
 System.***out***.println(**"Size ::: "** + list.size());  
  
 list.delete(**new** Integer(5));  
 System.***out***.println(**"Size ::: "** + list.size());  
 list.delete(**new** Integer(7));  
 System.***out***.println(**"Size ::: "** + list.size());  
 **for** (**int** i = 0; i < list.size(); i++) {  
 System.***out***.println(list.get(i));  
 }  
 }  
}

# **LinkedList**

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

**public** **class** LinkedList1

**Operation**

1. **Add**
2. **Add First**
3. **Print**
4. **Print Reverse**
5. **get Byi ndex**
6. **Delete**
7. **Get Size**
8. **delete by Index**
9. **delete first**
10. **delete last**
11. **Find middle**
12. **Reverse**
13. **Detect Loop**
14. **Create a Loop**

{

**private** Node front;

**private** Node back;

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

{

**private** Object element;

**private** Node next;

**public** Node( Object data )

{

element = data;

next = **null**;

}

}

**private** **boolean** isEmpty()

{

**return** front == **null**;

}

**public** **void** add(Object x)

{

**if**( isEmpty() )

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

**else**

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

}

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

Node f = front;

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

front.next = f;

}

**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;

}

//Print all the elements

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

{

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

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

}

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

**printRecursive(front);**

}

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

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

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

**printRecursive(p.next)**

**}**

**}**

// Print in reverse

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

{

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()

{

**return** length(front);

}

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

{

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

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

**return** 0;

**else**

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

}

//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) ) {

// special case, have to delete first node?

front = front.next;

}

**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 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++;

            }

        }

    }

**public** Object deleteFirst() //POLL operation

{

**if**(isEmpty()) **throw** **new** RuntimeException("No data");

Object data = front.element;

front = front.next;

**return** data;

}

**public** Object deleteLast( ) //POP() operation

{

**if**( isEmpty( ) )

**throw** **new** RuntimeException( "No data" );

Object returnValue = back.element;

**if**( front == back )

front = back = **null**;

**else**

{

Node current = front;

**while** ( current.next != back ) // not last node

current = current.next; // move to next node

back = current;

current.next = **null**;

}

**return** returnValue;

}

//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 pop()

{

**return** deleteLast();

}

**public** Object poll()

{

**return** deleteFirst();

}

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

{

front = deleteAt(p, front);

}

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

{

**if**(node == **null**)

**throw** **new** java.util.NoSuchElementException("cannot delete.");

**else**

**if**(p == 0)

**return** node.next;

**else**

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

**return** node;

}

//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);

}

//Reverse the LinkedList

**public** **void** reverseList() {

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 ) //**Best way to do it**

{

//check for empty list

**if**(currentNode == **null**)

**return**;

/\* 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();

}

}

# **A simple LinkedList**

**public** **class** MyLinkedList

{

// reference to the head node.

**private** Node head;

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

**private** **class** Node

{

// reference to the next node in the chain,

**private** Node next;

// data carried by this node.

**private** Object data;

**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);

}

}

# **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());

}

}

# **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() { // **deleteLast**  
 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() { // **Delete First**  
 **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();  
 }  
}

# **Write your own iterator**

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());  
 }  
  
 }  
 }

*//public interface Collection<E> extends Iterable<E>  
 //public interface List<E> extends Collection<E>*

# **Difference between Iterable and Iterator**

An implementation of Iterable is one that provides an Iterator of itself:

public interface Iterable<T>

{

Iterator<T> iterator();

}

An iterator is a simple way of allowing some to loop through a collection of data without assignment privileges (though with ability to remove).

public interface Iterator<E>

{

boolean hasNext();

E next();

void remove();

}

If a collection is iterable, then it can be iterated using an iterator (and consequently can be used in a for each loop.) The iterator is the actual object that will iterate through the collection. Iterable is an interface which provides Iterator.

**Iterable :** A class that can be iterated over. That is, one that has a notion of "get me the first thing, now the next thing, and so on, until we run out."

**Iterator :** A class that manages iteration over an iterable. That is, it keeps track of where we are in the current iteration, and knows what the next element is and how to get it.

To make an object iterable it needs to emit an Iterator object. To enforce this contract, Iterator interface is to be used. It contains a method named iterator() and it returns Iterator. Hence, any class that implements Iterable will return an Iterator.

public interface Collection<E> extends Iterable<E> {}

# **System.arraycopy and Arrays.copyOf**

**System.arraycopy**

**The method signature is given below.**

**public static native void** arraycopy(Object src,

**int** srcPos,  
 Object dest,

**int** destPos,  
 **int** length);

**Parameters**

1. **src** -- This is the source array.
2. **srcPos** -- This is the starting position in the source array.
3. **dest** -- This is the destination array.
4. **destPos** -- This is the starting position in the destination data.
5. **length** -- This is the number of array elements to be copied.

Points to be noted

1. Destination array should not be null, ie **int[] b = null**;
2. It is a native method.

Example

**int**[] a = {1,2,3,4,5,6,7,8,9,10};  
**int**[] b = **new int**[a.**length**];

*System.arraycopy(a,0,b,0,a.length);//copy all elements, 1 2 3 4 5 6 7 8 9 10*

System.*arraycopy*(a,0,b,0,2);*//copy 2 elements , 1 2 0 0 0 0 0 0 0 0*

**Arrays.copyOf**

**The method structure is given below.**

**public static int**[] copyOf(**int**[] original, **int** newLength)

**Parameters**

**Original – pass the actual array.**

**newLength – pass the number of elements to copy**

The source code is given below.

**public static int**[] copyOf(**int**[] original, **int** newLength) {  
 **int**[] copy = **new int**[newLength];  
 System.*arraycopy*(original, 0, copy, 0,  
 Math.*min*(original.**length**, newLength));  
 **return** copy;  
}

Arrays.copyOf always creates a new array and internally uses System.arrayCopy.

An example is given below.

**int**[] a = {1,2,3,4,5,6,7,8,9,10};

**int**[] b = **null**;

b = Arrays.*copyOf*(a, a.**length**); //1 2 3 4 5 6 7 8 9 10

b = Arrays.*copyOf*(a, 3); // 1 2 3 , here the size of the array is 3, not the 10.

**Why Math.min() is used in case of Arrays.copyOf() & Arrays.copyOfRange() ?**

**Ans : To avoid IndexOutOfBoundsException**

**Arrays.copyOfRange()**

**public static int[] copyOfRange(int[] original, int from, int to)**

The source code is given below.

**public static int**[] copyOfRange(**int**[] original, **int** from, **int** to) {  
 **int** newLength = to - from;  
 **if** (newLength < 0)  
 **throw new** IllegalArgumentException(from + **" > "** + to);  
 **int**[] copy = **new int**[newLength];  
 System.*arraycopy*(original, from, copy, 0,  
 Math.*min*(original.**length** - from, newLength));  
 **return** copy;  
}

parameters

**original – Actual array**

**from – starting position , initial index of the array, including**

**to – ending position, final index of the array, excluding**

**Example is given below.**

**int**[] a = {1,2,3,4,5,6,7,8,9,10};

**int**[] b = **null**;

b = Arrays.*copyOfRange*(a, 2,5); // 3 4 5

So here

Starting position - a[2] ie 3

Ending position – a[5] ie 6, but it will copy upto a[4}, as mentioned in the document as excluding.

# **Boyer-Moore Algorithm**

**public class** BoyerMooreAlgorithm {  
  
 **private final int BASE**;  
 **private int**[] **occurance**;  
 **private** String **pattern**;  
  
 **public** BoyerMooreAlgorithm(String pattern) {  
 **this**.**BASE** = 256;  
 **this**.**pattern** = pattern;  
 **occurance** = **new int**[**BASE**];  
 **for** (**int** c = 0; c < **BASE**; c++)  
 **occurance**[c] = -1;  
 **for** (**int** j = 0; j < pattern.length(); j++)  
 **occurance**[pattern.charAt(j)] = j;  
 }  
  
 **public int** search(String text) {  
 **int** n = text.length();  
 **int** m = **pattern**.length();  
 **int** skip;  
  
 **for** (**int** i = 0; i <= n - m; i += skip) {  
  
 skip = 0;  
 **for** (**int** j = m - 1; j >= 0; j--) {  
 **if** (**pattern**.charAt(j) != text.charAt(i + j)) {  
 skip = Math.*max*(1, j - **occurance**[text.charAt(i + j)]);  
 **break**;  
 }  
 }  
 **if** (skip == 0) **return** i;  
  
 }  
 **return** n;  
 }  
  
 **public static void** main(String[] args) {  
 String text = **"Lorem ipsum dolor sit amet"**;  
 String pattern = **"ipsum"**;  
 BoyerMooreAlgorithm bm = **new** BoyerMooreAlgorithm(pattern);  
 **int** firstPosition = bm.search(text);  
 System.***out***.println(**"Position :::"**+firstPosition);  
 }  
}

# **Another variation of Boyer-Moore Algorithm**

**import** java.util.Arrays;  
  
**public class** BoyerMooreAlgorithm {  
  
 **public static int** search(String text, String pat) {  
 **int** val = -1;  
 **int**[] dummy = **new int**[256];  
 Arrays.*fill*(dummy, -1);  
 **for** (**int** j = 0; j < pat.length(); j++) {  
 dummy[pat.charAt(j)] = j;  
 }  
  
 **int** m = text.length();  
 **int** n = pat.length();  
  
 **int** skip;  
  
 **for** (**int** i = 0; i <= m - n; i += skip) {  
 skip = 0;  
 **for** (**int** j = n - 1; j >= 0; j--) {  
  
 **if** (pat.charAt(j) != text.charAt(i + j)) {  
 skip = Math.*max*(1, j - dummy[text.charAt(i + j)]);  
 **break**;  
 }  
 }  
 **if** (skip == 0) **return** i;  
 }  
 **return** val;  
 }  
  
 **public static void** main(String[] args) {  
 String text = **"Lorem ipsum dolor sit amet"**;  
 String pattern = **"ipsum"**;  
 BoyerMooreAlgorithm bm = **new** BoyerMooreAlgorithm();  
 **int** firstPosition = bm.*search*(text,pattern);  
 System.***out***.println(**"Position :::"** + firstPosition);  
 }  
}

# **Brute Force Algorithm**

The brute force algorithm consists in checking, at all positions in the text between 0 and *n*-*m*, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right. The brute force algorithm requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. During the searching phase the text character comparisons can be done in any order. The time complexity of this searching phase is ***O***(*mn*)

**public class** BruteForceAlgorithm {  
  
 **public static int** search( String txt , String pat ) {  
  
 **int** M = pat.length();  
 **int** N = txt.length();  
  
 **for** (**int** i = 0; i <= N - M ; i++) {  
 **int** j;  
 **for** (j = 0; j < M; j++) {  
 **if**( txt.charAt(i+j) != pat.charAt(j))  
 **break**;  
 }  
 **if**( j == M) **return** i;  
 }  
 **return** -1;  
 }  
  
 **public static void** main(String[] args) {  
  
 **int** result = *search*(**"Hello world"** ,**"o wo"**);  
 System.***out***.println(result);  
 }  
}