Time Complexities

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Access by Index** | **Search** | **Add** | | | **Deletion** | | |
|  |  |  | Before first Node | After given Node | After Last Node | The First Node | A given Node | The last Node |
| **Array** | O (1) | O (n) | O(n) | O(n) | O(1) | O(n) | O(n) | O(1) |
| **Singly LinkedList** | O(N) | O(n) | O(1) | O(1) | O(1) | O(1) | O(n) | O(n) |
| **Doubly LinkedList** | O(N) | O(N) | O(1) | O(1) | O(1) | O(1) | O(1) | O(1) |
|  |  |  |  |  |  |  |  |  |

* If you need to add or delete a node frequently, a linked list could be a good choice.
* If you need to access an element by index often, an array might be a better choice than a linked list.

**Arrays Methods**

1. Print Array as String – Arrays.toString(arr);
2. Create an ArrayList from Array

Integer [] arr = new int[7];

List<Integer> lst = new ArrayList<>(Arrays.asList(arr));

1. Convert Arraylist to Array

List<Integer> lst = new ArrayList<>();

Integer[] arr = new Integer[lst.size()];

lst.toArray(arr);

1. Convert Array to a Set

String[] arr = new String[];

Set<String> s = new HashSet<>(Arrays.asList(arr));

1. Reverse elements of an Array

String[] arr = new String[];

ArrayUtils.reverse(arr);

1. Iterating through array

For(int item:nums){

System.out.println(item);

}

For(int i=0;i<nums.length;i++){

System.out.println(nums[i]);

}

1. Concatenate 2 arrays

Int[] arr1 = new int[7];

Int[] arr2= new int[8];

Int[] arr3 = ArrayUtils.addAll(arr1,arr2);

1. Java.util.Arrays methods

Arrays.sort(object[] a) 🡪 Sorts Array in ascending order

Arrays.fill(arr,-1) 🡪 Fills all elements of array with specific value

Arrays.binarySearch(int[] arr,int key) 🡪 returns index of the element If its present else returns –(insertion point +1)

Arrays.equals(int[] arr1,int[] arr2) 🡪 Returns true if both arr1 and arr2 have same elements and same length

1. Arrays Comparator , comparing 2D Arrays

**import** java.util.Arrays;

**import** java.util.Comparator;

Int[][] arr = new int[][]{};

Arrays.sort(arr,new Comparator<int[]>(){

public int compare(int[] a, int[] b){

return a[0] – b[0];

}

});

Return value :

<0 indicates accending order

0 indicates same value

>0 indicates descending value

**Primitive Types**

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Description** |
| int | 4 bytes | Stores value between -2^32 to +2^31-1  **Default value 0** |
| char | 2 bytes | Stores single character ASCII Character/Letter  0 to 65535(Unsigned)  A – Z is 65 to 90  a – z is 97 to 122  **Default value ‘u0000’** |
| byte | 1 byte | -128 to 127  **Default value 0** |
| short | 2 bytes | -2^16 to +2^16-1(-32768 to + 32767)  **Default value 0** |
| long | 8 bytes | Stores whole numbers -2^64 to +2^64-1  **Default value 0L** |
| float | 4 bytes | Stores fractional numbers . Decimals sufficient for storing 6 to 7 decimal digits  **Default value 0.0f** |
| double | 8 bytes | Stores fractional numbers. Decimals sufficient for storing upto 15 digits  **Default value 0.0d** |
| boolean | 1 byte | Stores true or false values  **Default Value false** |

Declaring primate types

char ch =’a’;

int n = 1;

long lg = 1500000000L;

float myNum = 1.234f

double myNum = 19.999d

**Conversion Table**

|  |  |  |
| --- | --- | --- |
| **From** | **To** |  |
| Integer | String | Integer.valueOf(str) |
| Integer | Character | Character.forDigit(n);  //Different behavior for int to char  Int n = 1;char ch = (char)(n+’0’); //prints 1  Int n = 1;char ch = (char)n; //it will store ASCII character of given number which is start of heading which is not printable  Int n =’1’;char ch = (char)n; //prints 1 |
| Character | Integer | Character.getNumericValue(ch)  Char ch = ‘6’;  int n = ch – ‘0’; |
| Character | String | String.valueOf(ch);  Character.toString(ch); |
| String | Integer | Integer.valueOf(str)  Integer.parseInt(str); |
| String | Character | s.charAt(i);  str.toCharArray() |
| Display Integer in Binary Format |  | Integer.toBinaryString(n) |

**Important in-built Methods**

|  |  |
| --- | --- |
| **Function** | **Description** |
| Math.sqrt(dbl) | Sqrt(x)   * public static double sqrt(double a)   Returns the correctly rounded positive square root of a double value. Special cases:   * + If the argument is NaN or less than zero, then the result is NaN.   + If the argument is positive infinity, then the result is positive infinity.   + If the argument is positive zero or negative zero, then the result is the same as the argument.   Otherwise, the result is the double value closest to the true mathematical square root of the argument value. |
| Math.pow(x,y) | X^y   * public static double pow(double a,double b)   Returns the value of the first argument raised to the power of the second argument. Special cases:   * + If the second argument is positive or negative zero, then the result is 1.0.   + If the second argument is 1.0, then the result is the same as the first argument.   + If the second argument is NaN, then the result is NaN. |
| Math.floor(dbl) | * public static double floor(double a)   Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer. |
| Math.ceil(dbl) | * public static double ceil(double a)   Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer. |
| Random | Import java.util.Random  Random rand = new Random();  rand.nextInt() 🡪 returns a number between 0 and 2^32 inclusive  rand.nextInt(16) 🡪 returns a number between 0(included) and 16 (excluded) |
| Math.sin(x) | sinxsin⁡x |
| Math.cos(x) | cosxcos⁡x |
| Math.tan(x) | tanxtan⁡x |
| Math.asin(x) | sin−1x |
| Math.acos(x) | cos−1x |
| Math.atan(x) | tan−1x |
| Math.exp(x) | ex |
| Math.log(x) | lnx |
| Math.log10(x) | log10x |
| Math.round(x) | the closest integer to x, as athe closest integer to x, as a long |
| Math.abs(x) | |x| |
| Math.max(x,y) | The maximum of the two values |
| Math.min(x,y) | The minimum of the two values |
| Math.PI | Double  3.14159265358979323846 |
| Math.E | Double  2.7182818284590452354 |

**Bit Manipulation**

|  |  |
| --- | --- |
| **Operation to Check** | **How to Check** |
| Check whether lowest significant bit is 1 or 0 | X & 1 |
| Reset Lowest bit set to 0 | X & (X -1) // drops lowest set bit of x  Example – 7 & 6 makes  111 & 110 🡪 110 which causes lowest bit to 0 |
| Retain Only Lowest bit and reset all other bits | X & ~(X -1) |
|  |  |

Iterator<E>

Iterable<E>

Java.util.Collection<E>

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered.

List<E>

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Queue<E>

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out).

Deque<E>

A linear collection that supports element insertion and removal at both ends. The name *deque* is short for "double ended queue" and is usually pronounced "deck". Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

Set<E>

A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

SortedSet<E>

A [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) that further provides a *total ordering* on its elements. The elements are ordered using their [natural ordering](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html), or by a [Comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order. Several additional operations are provided to take advantage of the ordering. (This interface is the set analogue of [SortedMap](https://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html).)

NavigableSet<E>

A [SortedSet](https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html) extended with navigation methods reporting closest matches for given search targets. Methods lower, floor, ceiling, and higher return elements respectively less than, less than or equal, greater than or equal, and greater than a given element, returning null if there is no such element. A NavigableSet may be accessed and traversed in either ascending or descending order.

Java.util.Collection<E>

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered.

List<E>

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Queue<E>

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out).

AbstractCollection<E>

AbstractList<E>

ArrayList<E>

(non-synchronized) Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.

Vector<E> (Synchronized)

Stack<E>

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

AbstractSequentialList

LinkedList

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Queue<E>

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out).

Dequeue<E>

ArrayDeque<E>

Resizable-array implementation of the [Deque](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html) interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads. Null elements are prohibited. This class is likely to be faster than [Stack](https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html) when used as a stack, and faster than [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) when used as a queue.

PriorityQueue<E>

An unbounded priority [queue](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html) based on a priority heap. The elements of the priority queue are ordered according to their [natural ordering](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html), or by a [Comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.