**Data Structure Implementation: Array Lists**

|  |
| --- |
| **Part 1 – Array List Attributes and Constructors:**  **public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List,RandomAccess,Cloneable,Serializable {  **private** **static** **final** **int** *MAX\_ARRAY\_SIZE* = Integer.*MAX\_VALUE* - 8;  **private** **transient** Object[] elementData;  **private** **int** size;    **public** ArrayList(**int** initialCapacity) {  **super**();  **if** (initialCapacity < 0){  **throw** **new** IllegalArgumentException();  }  **this**.elementData = **new** Object[initialCapacity];  }  **public** ArrayList() {  **this**(10);  }  **public** ArrayList(Collection<? **extends** E> collection) {  elementData = collection.toArray();  size = elementData.length;  **if** (elementData.getClass() != Object[].**class**){  elementData = Arrays.*copyOf*(elementData, size, Object[].**class**);  }  }  }   * Some JVMs reserve some header words in an array. So you must deduct 8 from the maximum for the header words to guarantee no out of space exceptions. * It only has two class attributes, and Object Array and an integer for the size. The array size doubles whenever more space is required during additions. * When you initialize an ArrayList, the default length of the array is 10. This length will increase if more space is required. * When you initialize an ArrayList with a list in the parameter (or a collection in general), it will set the array to the lists “to array” return. If this return is not of type object, it will copy this element into a new object array and set the attribute array to the new object array. |

|  |
| --- |
| **Part 2 – Generic Methods:**    **public** **int** size() {  **return** size;  }    **public** **boolean** isEmpty() {  **return** size == 0;  }    **public** **boolean** contains(Object o) {  **return** indexOf(o) >= 0;  }    **public** **int** indexOf(Object o) {  **if** (o == **null**) {  **for** (**int** i = 0; i < size; i++)  **if** (elementData[i]==**null**)  **return** i;  } **else** {  **for** (**int** i = 0; i < size; i++)  **if** (o.equals(elementData[i]))  **return** i;  }  **return** -1;  }  **private** **void** rangeCheck(**int** index) {  **if** (index >= size)  **throw** **new** IndexOutOfBoundsException();  }  **public** E get(**int** index) {  rangeCheck(index);  **return** (E) elementData[index];  } |

|  |
| --- |
| **Part 3– Adding elements to the Array List:**  **public** **void** ensureCapacity(){  //Increases the size by 50% if necessary.  //New size is equal to old size \* 1.5  //It had a tedious implementation that I avoided to show.  }    **private** **void** rangeCheckForAdd(**int** index) {  **if** (index > size || index < 0)  **throw** **new** IndexOutOfBoundsException();  }  **public** **boolean** add(E e) {  ensureCapacity();  elementData[size++] = e;  **return** **true**;  }  **public** **void** add(**int** index, E element) {  rangeCheckForAdd(index);  ensureCapacity();  System.*arraycopy*(elementData, index, elementData, index + 1,size - index);  elementData[index] = element;  size++;  }  **public** **boolean** addAll(Collection<? **extends** E> c) {  Object[] a = c.toArray();  **int** numNew = a.length;  ensureCapacity();  System.*arraycopy*(a, 0, elementData, size, numNew);  size += numNew;  **return** numNew != 0;  } |

|  |
| --- |
| **Part 4 – Removing things from the list:**  **private** **void** rangeCheck(**int** index) {  **if** (index >= size)  **throw** **new** IndexOutOfBoundsException();  }  **public** E remove(**int** index) {  rangeCheck(index);  E oldValue = (E) elementData[index];  **int** numMoved = size - index - 1;  **if** (numMoved > 0){  System.*arraycopy*(elementData, index+1, elementData, index, numMoved);  }  elementData[--size] = **null**;  **return** oldValue;  }  }  **public** **boolean** remove(Object o) {  **if** (o == **null**) {  **for** (**int** index = 0; index < size; index++)  **if** (elementData[index] == **null**) {  remove(index);  **return** **true**;  }  } **else** {  **for** (**int** index = 0; index < size; index++)  **if** (o.equals(elementData[index])) {  remove(index);  **return** **true**;  }  }  **return** **false**;  } |

**Part 5 – Time and Space Complexities:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Average Time Complexity** | **Worst Time Complexity** | **Space Complexity** |
| System.*arraycopy()* | O(n) | O(n) | O(n) |
| contains(Object o) | O(n) | O(n) |
| indexOf(Object o) | O(n) | O(n) |
| get(int index) | O(1) | O(1) |
| add(E e) | O(1) | O(1) |
| add(int index, E element) | O(n) | O(n) |
| addAll(Collection c) | O(n) | O(n) |
| remove(int index) | O(n) | O(n) |
| remove(Object o) | O(n) | O(n) |

* The System.*arraycopy()* method is O(n) in average and worst case. This method takes in four parameters; the source array, the source index, the destination array, the destination index, and the length. What it will do is it will copy (length) indexes from source array, source position into destination array, destination position. This method is primarily used to create space for an element at a particular index (when you call add at an index) or it will remove space at a particular index (when you call any remove). So it moves a portion of the array one spot backwards or one spot forwards to create or remove space.
* Since add(int index, E element), remove(int index), remove(Object o), addAll(Collection c) all call System.*arraycopy()* they all have linear time complexities. Other than one method call, the rest of their code has a constant number of steps (Note all these methods need to either create space in the middle of the array or remove space in the moddile of the array). Note that the add all function is different that it appends an entire array at the end of this array. So this linear time with respect to the number of elements in the collection that is being added (n is the number of elements in collection c). So it is a bit different.
* contains(Object o) and indexOf(Object o), (contains calls indexOf) both have to do a linear search in the array for the element they are searching for. This causes the linear time complexity.
* The remove(Object o) has to also search linearly for the item it is removing. This aloing with the System.*arraycopy()* makes its time complexity technically O(2n) in the worst case.
* No method uses any extra space. The arraylist itself is O(n) to store n elements. Note that O(n) is ideal to store n elements in that it is impossible to store n elements in O(1). The actual operations perform in place (they don’t use any extra space except).