Array Data Structure

Linear Search and the Binary Search Algorithms for arrays

- Searching:
  - Searching is a very common task in computer programming.
  - Many algorithms and data structures are invented to support fast searching.
- Searching arrays:
  - Arrays are often used to store a large amount of data.
  - Searching is the process of looking for a specific element in an array.
  - There are two search techniques for arrays: **linear search** and **binary search**.

> **The Search Problem for Arrays**:
>
> - For a given search value `key`, find the index of the first array element that contains the search value `key`.
> - Return `-1` when the `key` is not found in the array.

- Linear Search algorithm:
  - The linear search algorithm compares the search value `key` sequentially with each element in the array.
  - The linear search algorithm continues to do so until the `key` matches an element in the array or the array is exhausted without a match being found.
  - If a match is made, the linear search returns the index of the element in the array that matches the `key`.
  - If no match is found, the search returns `-1`.

```java
/**
 * The linear search algorithm to find key in the array list
 */
public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length, i++>) {
        if (list[i] == key) {
            return i;
        }
    }
    // key is not found in list[]
    return -1;
}
```

- Complexity Analysis:
  - Best case scenario: the first element in the array contains the search key. Running time = 1 step (iteration)
  - Worst case scenario: array does not contain the search key. We run through the whole array. Running time = N steps.
  - Average case scenario: on average, we will probe half of the array elements. Running time = $\frac{N}{2}$ steps.
- Binary search is a more efficient (faster) search algorithm for arrays.
  - For binary search to work, the elements in the array must already be ordered.
    - For the presentation of the binary search, we assume that the array is in ascending order.
  - The binary search compares the `key` with the element in the middle of the array.

```java
/**
 * The binary search algorithm for arrays
 */
public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (list[mid] == key) {
            return mid;
        } else if (list[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    // If key is not found in the list[]
    return -1;
}
```

- Complexity Analysis:
    - Best case scenario: the middle element in the array contains the search key. Running time = 1 step (iteration).
    - Worse case scenario: Suppose the binary search takes k iterations to complete, then $\frac{N}{2^k} = 1$ (after halving N elements for k times, we get 1). Solving the equation, we get $k = \log(N) + 1$. So, running time $\approx \log(N)$ steps.
    - Average case scenario: it is very hard to estimate, but we can use the worse case scenario as an upper bound for the running time in average.

Adding or Deleting Elements from an Array.

- Adding an element at the end of an array
    - The array size is fixed after its creation.
    - To add a new element to the end of an array:
        - Create a new array with the 1 more element
        - Copy the elements in the original array to the new array.
        - Copy the new value in the last element of the new array.
        - Change the array reference to point to the new array.

```java
public static int[] addElement(int[] x, int e) {
    int[] temp = new int[x.length + 1];
    for (int i = 0; i < x.length; i++) {// copying
        temp[i] = x[i];
    }
    temp[temp.length - 1] = e;
    return temp;
}
```

- The algorithm executes $\mathtt{x.length} + 1$ data copy statements per addition.
- Deleting the last element from an array.
  - The array size is fixed after its creation.
  - To delete the last element from an array:
    - Create a new array with the 1 less element.
    - Copy all except the last elements in the original array to the new array.
    - Change the array reference to point to the new array.

```java
public static int[] deleteElement(int[] x) {
    int[] temp = new int[x.lenght - 1];
    for (int i = 0; i < temp.length; i++) {
        temp[i] = x[i];
    }
    return temp;
}
```

- We can delete an element at a different location with a similar algorithm.
- A better way to add and delete elements in arrays: Dynamic arrays (aka. `ArrayList` in Java). It consists of
  - A (fixed size) array
  - A count of the actual number of elements stored in the array.
  - The array is increased only when the `add()` operation encounters a full array.
  - The array is reduced when the occupancy drops below a certain threshold.
  - Inserting a new value will increase the count. If the array is not full, we do not need to increase its size.
  - The array is increased only when the `add()` operation encounters a full array.
    - The `add()` method will increase the array size by approximately twice the original size. This will avoid frequent copy operations.
  - The array is reduced when the occupancy drops below a certain threshold.
- A commonly used algorithm to implement dynamic array is array doubling:

```
temp = new int[2 * x.length];
for (int i = 0; i < x.length; i++) {
  temp[i] = x[i];
}
x = temp;
```

- The `ArrayList` class in Java implements a dynamic (resizable) array
  - To use it, we `import java.util.ArrayList;`
  - Syntax to define an `ArrayList` (reference) variable:

    ```
    ArrayList<ObjectType> varName
    ```

  - Syntax to create an `ArrayList` object

    ```
    new ArrayList<Object Type> ();
    ```

  - The `ArrayList` object will start with an array of limited size (about 10).
  -
    - See `DynamicArray.java`
- Commonly used methods in the `ArrayList` class
  - `size()` : returns the actual number of elements in the `ArrayList`
  - `toString()` returns a `String` representation of all elements stored in the `ArrayList`
  - `add(E e)` : appends the element `e` to the end of the `ArrayList` (`E` is the declared data type of the `ArrayList` elements)
  - `add(int index, E elem)` : inserts the element `e` at index `index` and shifts and subsequent items to the right
  - `remove(int index)` : removes the element at index `index` and shifts all remaining items to the left.
  - `get(int index)` : returns the element stored at the index `index`
  - `set(int index, E elem)` : replaces the element at index `index` with the element `elem`
  - If the element at the `index` does not exist, `get()` and `set()` will throw `IndexOutOfBoundsException` .
- Iterating through an `ArrayList` :
  - Use a regular `for` -loop and `get(index)` :

    ```
    for (int i = 0; i < numbers.size(); i++) {
      System.out.println(numbers.get(i));
    }
    ```

  - Use a `foreach` loop:

```java
for (int item: numbers) {
  System.out.println(item);
}
```

- Note: a `foreach` loop cannot be used to update array elements
- Using an iterator object:

```java
Iterator<Integer> numItr = numbers.iterator();
while (numItr.hasNext()) {
  System.out.println(numItr.next());
}
```

- Java `Iterator` interface and `Iterable` interface
  - `Iterator` is an interface (class containing all virtual methods) in `java.util.Iterator` .
  - An object that implements the `Iterator` interface must provide at least the following methods:
    - `hasNext()` : returns true if the iteration has more elements
    - `next()` : return the next element in the iteration
  - An `Iterator` allows the user to iterate over the elements stored in an `Iterable` interface.
  - An object is `Iterable` if it implements the `java.util.Iterable` interface.
    - It must implement the `iterator()` method that returns a `Iterator` object.

	Array	**ArrayList**
Pros	Uses less memory; can store primitive types; can be multi-dimensional	Size is dynamic; easy to add/remove elements
Cons	Size cannot change;hard to add/remove elements	Uses more memory; cannot store primitive types; can only be one-dimensional