University of Central Punjab

(Incorporated by Ordinance No. XXIV of 2002 promulgated by Government of the Punjab)

FACULTY OF INFORMATION TECHNOLOGY

Data Structures and Algorithms - Lab

Lab 3

CLO NO	**CLO STATEMENT**	**Blooms Taxonomy Level**	**PLO**
1	Solve real-world problems skillfully with precision using programming constructs learned in theory with the course toolkit.	P3	5

Task 1: Binary Search Algorithm

Binary search is searching algorithm used to search a specific value (or index of value) from the *sorted array*. In binary search, we first compare the *value to be searched* with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater than the item sought must lie in the upper half of the array. So, we repeat the procedure on the lower (or upper) half of the array. The pseudocode of the task is given below:

```
Input: A Sorted array, A of N elements and value to be searched
Output: Index of searched element or -1 if not found

low = 0, high = N-1;
While (low <= high)
   mid = (low + high)/2;
   If (A[mid] == Value )
     Return mid;
   Else-if (A[mid] < Value )
     low = mid + 1;
   Else
     high = mid - 1;
   End-if
End-While
return -1;
```

Implement the code for binary search using templates. Test your code using the main function below.

```
int main () {
    // Test with an integer array (sorted) of size 5
    int intArray[5] = {11, 12, 22, 25, 64};
    int intKey = 22;
    int intIndex = binarySearch(intArray, intKey);
    printSearchResult(intIndex, intKey);

    // Test with a float array (sorted) of size 4
    float floatArray[4] = {0.57, 1.62, 2.71, 3.14};
    float floatKey = 2.71;
    int floatIndex = binarySearch(floatArray, floatKey);
    printSearchResult(floatIndex, floatKey);

    // Test with a string array (sorted) of size 4
    string stringArray[4] = {"apple", "banana", "grape", "orange"};
    string stringKey = "grape";
    int stringIndex = binarySearch(stringArray, stringKey);
    printSearchResult(stringIndex, stringKey);

    return 0;
}
```

Task 2: Template-Based Array Implementation

You are required to implement a generic **Array** class using C++ templates to handle elements of any data type. The class should include the following:

Attributes:

```
Type* data – A pointer to an array of template type
elements.
int size – Capacity of the array.
int n – Number of elements currently in the array.
```

Functions:

1. **Constructor**: Parameterized constructor with a default size, initializing the dynamic array.
2. **add (Type element)**: Adds an element at the end of the array. If the array is full, call the grow () function.
3. **remove ()**: Removes the last element in the array. If the number of elements becomes less than half the size, call the shrink () function.
4. **search (Type element)**: Searches for an element and returns the index if found; otherwise, return -1.
5. **grow ()**: Doubles the size of the array and reallocates memory.
6. **shrink ()**: Shrinks the size of the array to half if the number of elements is less than half of the current size.
7. **sort ()**: Sorts the array in ascending order using any sorting algorithm (e.g., bubble sort or selection sort).
8. **print ()**: Prints all elements in the array.

Instructions:

- Write a **parameterized constructor with default arguments** for the above class.
- Write a **copy constructor** for the above class.
- Write **destructor** for the above class.

Task 3: Application-Based Question

In the above task you have created a generic **Array<Type>** class that implements a dynamic array using templates. Now create a **student** class with the following attributes:

```
int id
string name
float CGPA
```

The input/output operators (>>, <<) and comparison operators (<, ==) have been overloaded for **student.**

Write a C++ program using your **Array<Student>** class to perform the following:

1. **Initialize** an Array<Student> object with a capacity of 3.
2. **Input and add** 4 student records into the array (one more than the initial size to trigger grow ()).
3. **Print** all student records using the print () function.
4. **Search** for a specific student (by ID or any attribute) using the search () function.
5. **Remove** the last student using the remove () function.
6. **Shrink** the array (if conditions are met after removal).
7. **Sort** the array based on CGPA.
8. **Print** the final sorted list of students.

Sample Output:

```
Initial array created with size 3.
Adding 4 students to array...

Student added: Ahmad (ID: 1, CGPA: 3.8)
Student added: Abdul Rehman (ID: 2, CGPA: 3.2)
Student added: Noraize (ID: 3, CGPA: 3.9)

Array full. Growing array...
Array size doubled to 6
Student added: Qasim (ID: 4, CGPA: 2.9)

--- Current Student Records ---
1. ID: 1 | Name: Ahmad | CGPA: 3.8
2. ID: 2 | Name: Abdul Rehman | CGPA: 3.2
```

```
3. ID: 3 | Name: Noraize | CGPA: 3.9
4. ID: 4 | Name: Qasim | CGPA: 2.9
Searching for student: Qasim
Student found at index: 3

Removing last student (Qasim)
Student removed.
Shrinking array...
Array size reduced to 3

Sorting students by CGPA
Sorting completed.

--- Sorted Student Records ---
1. ID: 2 | Name: Abdul Rehman | CGPA: 3.2
2. ID: 1 | Name: Ahmad   | CGPA: 3.8
3. ID: 3 | Name: Noraize | CGPA: 3.9
```