# Programming Assignment #1, Phase1

## The Program:

For this assignment, you will write two implementations of a *Priority Queue*. For this ADT, removal operations always return the object in the queue of highest priority that has been in the queue the longest. That is, no object of a given priority is ever removed as long as the queue contains one or more object of a higher priority. Within a given priority First-In-First-Out (FIFO) order must be preserved.

Your implementations will be:

- Ordered Array
- Unordered Array

Both implementations must have identical behavior and must implement the `PriorityQueue` interface (provided). The implementations must have two constructors, a default constructor with no arguments that uses the `DEFAULT_MAX_CAPACITY` constant from the `PriorityQueue` interface (implemented with a pure virtual class), and a constructor that takes a single integer parameter that represents the maximum capacity of the priority queue. The PriorityQueue interface follows in Phase 2.

This assignment will be implemented in two phases. Phase 1 consists of implementing the two Array classes: `UnorderedArrayList` and `OrderedArrayList`. Phase 2 consists of implementing a Priority Queue for each array class type.

---

## Phase 1 instructions

To begin, read *Separate Compilation*. If you skip this step, you may struggle to understand how the program files work together.

Class prototypes defined in `UnorderedArrayList.h` and `OrderedArrayList.h` are provided. Each of these classes are subclasses of the pure virtual `AbstractList` class defined in its header file. This file is also provided; you may not modify `AbstractList.h`

Your project Phase 1 will consist of the following files. You must use exactly these filenames.

| | |
|---|---|
| `AbstractList.h` | The ADT interface (provided). |
| `OrderedArrayList.h` | The ordered array definition (provided). |
| `UnorderedArrayList.h` | The unordered array definition (provided). |
| `OrderedArrayList.cpp` | The ordered array implementation. |
| `UnorderedArrayList.cpp` | The unordered array implementation. |

A makefile is provided.

Your task is to write *your own code* for the required public functions in `UnorderedArrayList.h` and `OrderedArrayList.h`. You may add private members as needed. Some public functions may be useful for debugging purposes, but these must not interfere with the required functions.

Testing your code is essential and may require more time than writing the classes. Be sure to test combinations of functions.

## Additional Details

Each method must be as efficient as possible. That is, a O(n) is unacceptable if the method could be written with O(log n) complexity. Accordingly, the ordered array implementation must use binary search where possible, such as `contains()`, `delete(int obj)`, and to identify the correct insertion point for new additions.

By convention, a lower number=higher priority. If there are five priorities for a given object, 1 .. 5, then 1 is the highest priority, and 5 the lowest priority.

Your project must consist of only the five files specified (including the provided interface), no additional source code files are permitted. (Do not hand in a copy of AbstractList.h, as it is provided to you).

You may not make any modifications to the AbstractList interface provided. I will grade your project with my copy of this file.

All source code files must have your name and class account number at the beginning of the file.

Your implementations may only include `"OrderedArrayList.h"`, `"UnorderedArrayList.h"`, `<stdexcept>`, and `<iostream>` (for debugging purposes and should be removed). If you feel that you need to include anything else, let me know. You are expected to write all the code yourself, and you may not use the C++ Standard Library API for any containers.

Your code must not print anything.

Your code should never crash but must handle any/all error conditions gracefully. i.e. if the user attempts to call the clear() method on an empty list, or remove an item from an empty list, the program should not crash. Be sure to follow the specifications for all methods.

You must write generic code according to the interface provided. You may not add any public methods to the implementations, but you may add private ones, if needed.

Your code may generate unchecked cast warnings when compiled, but it must compile and run correctly on Gradescope to receive any credit. Testing on edoras may prove to be a helpful exercise.

Minimal tester/driver programs will be provided to help you test your code. You should add more tests as you think of more scenarios.

Allowing sufficient time for testing <u>on the grading platform</u> is essential. If testing on edoras, use your ~/sandbox/p1/ with the following files:

```
p1tester.cpp
AbstractList.h
OrderedArrayList.h
UnorderedArrayList.h
OrderedArrayList.cpp
UnorderedArrayList.cpp
makefile
```

## Turning in your project

To submit your project, you must upload both C++ source code files to your `Gradescope` account through Canvas. The automated grading program will run. You're permitted unlimited submissions, but *don't use Gradescope as your testing platform.* If we see this is happening, we will enforce limited submissions.

```
OrderedArrayList.cpp
UnorderedArrayList.cpp
```

## Cheating Policy

There is a zero tolerance policy on cheating in this course. You are expected to complete all programming assignments on your own. Collaboration with other students in the course is not permitted. You may discuss ideas or solutions in general terms with other students, but you must not exchange code. During the grading process I will examine your code carefully. Anyone caught cheating on a programming assignment (or on an exam) will receive an "F" in the course, and a referral to Judicial Procedures.

AbstractList.h

```
#ifndef ABS_LIST_H
#define ABS_LIST_H
/**
 * Pure virtual abstract class
 * @author Student
 */
#include <stdexcept>

class AbstractList
{
public:

   /**
    * Appends the specified data to the end of this list
    * param  int  data element to insert
    * return bool 1 success
    */
    virtual bool add(int) = 0;

   /**
    * Appends the specified data to the list at the
    * specified index. Valid indexes are 0 to size.
    * @param int   index position to insert data
    * @param data  element to insert
    * @return bool success
    */
    virtual bool add(int index, int data) = 0;

   /**
    * Removes all of the Objects from this list leaving
    * capacity the same.
    */
    virtual void clear() = 0;

   /**
    * Returns true if this list contains the specified Object
    * @param data
    * @return bool 1 success
    */
    virtual bool contains(int data) = 0;

   /**
    * Returns the Object at the specified position in this list
    * @param index
    * @return Object
    * @throws invalid_argument if index out of range
    */
    virtual int get(int index) = 0;

   /**
    * Returns the Object at the specified position in this list and
    * deletes it from the list
    * @param index element to remove
    * @throws invalid_argument if index out of range
    */
```

```
    virtual int remove(int index) = 0;

    /**
     * Removes all occurences of the Object in this list
     * @param data element(s) to remove
     * @return 1 success
     */
    virtual bool removeAll(int data) = 0;

    /**
     * Returns the index of the first occurrence of the
     * specified Object in this list, or -1 if this list
     * does not contain the Object
     * @param data element to search for
     * @return int position of data if found, else -1
     */
    virtual int indexOf(int data) = 0;

    /**
     * Returns true if this list contains no Objects
     * @return bool
     */
    virtual bool isEmpty() = 0;

    /**
     * Returns the number of Objects in this list
     * @return int
     */
    virtual int size() = 0;

    /**
     * Trims the capacity of this instance to be the list's
     * current size.  An application can use this
     * operation to minimize the storage of an instance.
     */
    virtual void trimToSize() = 0;

protected:
    int currentSize;
};
#endif
```

OrderedArrayList.h

```
#ifndef ORD_ARRLIST_H
#define ORD_ARRLIST_H

#include "AbstractList.h"

class OrderedArrayList : public AbstractList {

private:
    int* array;
    int capacity;
    int currentPos;

public:
    OrderedArrayList();
    OrderedArrayList(int initialCapacity) ;

    virtual bool add(int data) ;
    virtual bool add(int index, int data) ;
    virtual void clear() ;
    virtual bool contains(int data) ;
    virtual int  get(int index) ;
    virtual int  indexOf(int data) ;
    virtual bool isEmpty() ;
    virtual int  remove(int index) ;
    virtual bool removeAll(int data) ;
    virtual int  size() ;
    virtual void trimToSize() ;

};
#endif
```

UnorderedArrayList.h

```
#ifndef UNORD_ARRLIST_H
#define UNORD_ARRLIST_H

#include "AbstractList.h"

class UnorderedArrayList : public AbstractList {

private:
    int* array;
    int capacity;
    int currentPos;

public:
    UnorderedArrayList();
    UnorderedArrayList(int initialCapacity) ;

    virtual bool add(int data) ;
    virtual bool add(int index, int data) ;
    virtual void clear() ;
    virtual bool contains(int data) ;
    virtual int  get(int index) ;
    virtual int  indexOf(int data) ;
    virtual bool isEmpty() ;
    virtual int  remove(int index) ;
    virtual bool removeAll(int data) ;
    virtual int  size() ;
    virtual void trimToSize() ;

};
#endif
```

**Minimal test file: p1tester.cpp**

```
#include "UnorderedArrayList.h"
#include "OrderedArrayList.h"
#include <iostream>

using namespace std;

int main()
{
  UnorderedArrayList list(2);

  cout << "capacity is: " << list.getCapacity() << endl;
  cout << "size is: " << list.size() << endl;
  cout << "isEmpty? " << list.isEmpty() << endl;
  cout << list.add(5) << endl;
  cout << "size is: " << list.size() << endl;
  cout << "isEmpty? " << list.isEmpty() << endl;
  cout << list.add(10) << endl;
  cout << list.add(15) << endl;
  cout << "size is: " << list.size() << endl;
  cout << list.add(1, 6);
  list.print();  // debug method
  cout << "size is: " << list.size() << endl;
  cout << list.add(20) << endl;
  cout << "size is: " << list.size() << endl;
  cout << "capacity is: " << list.getCapacity() << endl;
  cout << list.add(25) << endl;
  cout << "size is: " << list.size() << endl;
  cout << "capacity is: " << list.getCapacity() << endl;
  cout << "removing index 2" << endl;
  list.remove(2);
  list.print();
  cout << "size is: " << list.size() << endl;
  cout << "indexOf 20 " << list.indexOf(20) << endl;
  cout << "indexOf 21 " << list.indexOf(21) << endl;
  cout << "contains 20 " << list.contains(20) << endl;
  cout << "contains 21 " << list.contains(21) << endl;
  cout << "get 0 " << list.get(0) << endl;
  cout << "get 3 " << list.get(3) << endl;
  list.print();
  try {
      cout << "get 100 " << list.get(100) << endl;
  }
  catch (invalid_argument& e) {
      cout << e.what() << endl;
  }
  cout << "Still running...\n";

  OrderedArrayList ol;

  cout << ol.size() << endl;
  cout << ol.add(5) << endl;
  cout << ol.size() << endl;
  // etc.
  return 0;
}
```

**Makefile:**

```
# CS 210 Spring 2023
# p1 configuration
# Usage:
#   make UnorderedArrayList.o
#   make p1

CC=g++
CFLAGS=-g -Wall -std=c++11

p1: p1tester.cpp UnorderedArrayList.o OrderedArrayList.o
	$(CC) $(CFLAGS) -o p1 p1tester.cpp UnorderedArrayList.o OrderedArrayList.o

UnorderedArrayList.o: UnorderedArrayList.cpp UnorderedArrayList.h AbstractList.h
	$(CC) $(CFLAGS) -c UnorderedArrayList.cpp

OrderedArrayList.o: OrderedArrayList.cpp OrderedArrayList.h AbstractList.h
	$(CC) $(CFLAGS) -c OrderedArrayList.cpp

clean:
	rm *.o

cleanall:
	rm *.o p1
```