# Assignment 1: Array List and Linked List

---

You need to implement a **MyList** class supporting the following functionalities as defined below in Table 1. You can use **C++/Java** as programming language.

| Fn | Function | Param. | Ret. | After Function Execution | Comment |
|---|---|---|---|---|---|
| | before execution of any function, consider current list is:<br><1, 2 \| 3, 4> | | | | The vertical bar indicates the current position. Here, the current position is 2 and corresponds to element 3. First position is 0. |
| 1 | size() | | 4 | <1, 2 \| 3, 4> | Returns the number of elements |
| 2 | push(item) | 5 | | <1, 2 \| 5, 3, 4> | Push an element at current position |
| 3 | pushBack(item) | 5 | | <1, 2 \| 3, 4, 5> | Push an element at the end of the list |
| 4 | erase() | | 3 | <1, 2 \| 4> | Erase and return the current element |
| 5 | setToBegin() | | | <\| 1, 2 ,3, 4> | Make beginning of the list as current position |
| 6 | setToEnd() | | | <1, 2, 3 \| 4> | Make end of the list as current position |
| 7 | prev() | | | <1 \| 2 , 3, 4> | Move the current position to one step left. Ignore if already at the beginning |
| 8 | next() | | | <1, 2, 3 \| 4> | Move the current position to one step right. Ignore if already at the end |
| 9 | currPos() | | 2 | <1, 2 \| 3, 4> | Return the position of current element |
| 10 | setToPos(pos) | 0 | | <\| 1, 2, 3, 4> | Set current position |
| 11 | getValue() | | 3 | <1, 2 \| 3, 4> | Return current element |

| Fn | Function | Param. | Ret. | After Function Execution | Comment |
|---|---|---|---|---|---|
| 12 | find(item) | 3 | 2 | <1, 2 \| 3, 4> | Return the position of item if it exists, otherwise return -1. |
| 13 | clear() | | | <> | Clear the list |

**The implementation must support the various types of 'elements'**. Use templates in C++ or something equivalent in the programming language you intend to use.

You need to provide two different implementations:

1. Array Based (Arr) and
2. Linked List Based (LL)

In case of **Arr** implementation, there must be a way to dynamically **grow and shrink** the list size as follows: The list should **allocate memory dynamically.** Initially the list should be able to hold X elements; as soon as the attempt is made to insert the X+1th element, memory should be allocated such that it can hold 2X elements, then 4X, 8X and so on. Similarly , if at any moment the current number of elements becomes less or equal half the current capacity, release the unused memory. **Static implementation would result in deduction of marks**.

Appropriate constructors and destructors must be implemented. Constructors should at least support initializing

- an empty list
- a list with Y elements where Y < X.

You may implement extra helper functions but those will not be available for programmers to use. So, while using the MyList implementations, one can only use the methods listed in Table 1. Please note that during evaluation, identical main function will be used to check the functionality of both Arr and LL implementations except for the object instantiation part.

### Input format (for checking the Arr and LL implementations):

First line will contain two integers K,X (K < X) describing the initial length of the list (K) and initial capacity X. (**Ignore X for LL implementation**).
Next line will contain K space separated integers denoting the initial list elements.
Next line will contain a single integer, Q denoting the number of operations to process.
Each of the next Q lines will contain two integers F (1<=F<=13), P where F denotes the function to execute, and P is its parameter. Ignore P in case F doesn't take any parameter.

Input will be given in a file named **list_input.txt** .

Example: list_input.txt

```
4 8
1 2 3 4
4
1 1
2 10
4 0
5 0
```

## Output format:

At first the system will output in one line the items of the list within angle bracket, space separated identifying the current position using "|" as shown in Table 1.
Then for each of the Q operations output two lines as follows:
The first line will output the items of the list as described above.
The second line will output the return value (if available) or -1 (if no return value is expected).

Output should be in a file named **list_output.txt** .

Example: list_output.txt

```
<1 2 | 3 4>
<1 2 | 3 4>
4
<1 2 | 3 4 10>
-1
<1 2 | 4 10>
3
<| 1 2  4 10>
-1
```

**Using the MyList Data Structures:**

Now that you have MyList ready, you are going to use it to implement the Least Recently Used (LRU) cache. Least Recently Used (LRU) is a cache replacement algorithm that replaces cache when the space is full. It allows us to access the values faster by removing the least recently used values.  It is an important algorithm for the operating system, it provides a page replacement method so that we get fewer page faults. The cache will be of fixed size and when it becomes full we will delete the value which is least recently used and insert a new value.

Now implement a **LRUCache** class with following functionalities:

| Function | Comment |
|---|---|
| LRUCache(capacity) | Initialize the LRU cache with given capacity |
| get(key) | Return the value of the key if the key exists, otherwise return -1 |
| put(key, value) | Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. **If the number of keys exceeds the capacity from this operation, evict the least recently used key** |

### Input format:

First line will contain a single positive integer, C denoting the capacity of the cache.
Next line will contain another positive integer Q, denoting the number of operations to perform.
There will be two types of operation:

1. get
2. put

Each of the next Q lines will describe the operations as follow:

I. 1 x (get the value associated with key x)
II. 2 x y (put value y against key x)

Input will be given in a file named **lru_input.txt** .

Example: lru_input.txt

```
2
7
2 1 1
2 2 2
2 3 3
1 1
1 2
2 4 4
1 3
```

## Output format:

For each type 1 operation, print the return value from get function.
Output should be in a file named **lru_output.txt** .

Example: lru_output.txt

-1
2
-1

## Submission Guidelines:

1. Create a directory with your 7-digit student id as its name.
2. You need to create separate files for the Arr implementation and LL implementation, putting the main function in a separate file demonstrating your implementations. Create a separate file for implementing the LRUCache. Your LRU implementation must use your own array/LL implementations. No other built in data structure can be used. You must write just one piece of code that will work on both Arr and LL implementations, the only difference will be the list object instantiations. So, you can only use the methods listed in Table 1.
3. Put all the source files only into the directory created in step 1. Also create a readme.txt file briefly explaining the main purpose of the source files.
4. Zip the directory (compress in .zip format. Any other format like .rar, .7z etc. is not acceptable).
5. Upload the .zip file on Moodle in the designated assignment submission link. For example, if your student id is xx05xxx, create a directory named xx05xxx. Put only your source files (.c, .cpp, .java, .h etc.) into xx05xxx. Compress the directory xx05xxx into xx05xxx.zip and upload the xx05xxx.zip on Moodle.

Failure to follow the above-mentioned submission guideline may result in up to 10% penalty.

**Submission Deadline**: 02/12/2022 11:59PM

**Plagiarism Policy:** You will be penalized with -100% if any sort of plagiarism found.