Project Five: List, Stack and Queue

Out: July 21, 2020; Due: August 4, 2020

I. Motivation

This project will give you experience in applying dynamic memory management, implementing a template container class (the double−ended, doubly−linked list, or `Dlist`), and using the at-most−once invariant and existence, ownership, and conservation rules to implement two simple applications using this structure.

II. Programming Assignment

You will first implement a templated double-ended, doubly-linked list, or `Dlist`. Then, you will use `Dlist` to build two applications: Reverse Polish Notation Calculator and a least-recently-used cache.

1. The Double-Ended, Doubly-Linked List

The double−ended, doubly-linked list, or `Dlist`, is a templated container. It supports the following operational methods:

```
isEmpty: a predicate that returns true if the list is empty,
false otherwise.

insertFront/insertBack: insert an object at the front/back of the
list, respectively.

removeFront/removeBack: remove an object from the front/back of a
non-empty list, respectively; throws an exception if the list is
empty.

remove: takes a function pointer and an object as arguments; uses
the input function pointer as comparison function and the input
object as reference; removes and returns the object in the list
that is equivalent to the reference object judging by the compar-
ison function (there is at most one such object existing in the
list); returns NULL pointer if no such object exists.
```

Note that while this list is templated across the contained type, `T`, it is a container of **pointers-to-T**, not container of instances of T. Insertion takes a **pointer-to−`T`** as an argument and put **that pointer** into the node to be inserted. Removal removes a node and returns the pointer-to-T kept in that node. This ensures that the `Dlist` implementation knows that: it owns inserted objects, it is responsible for copying them if the list is copied, and it must destroy them if the list is destroyed.

The complete interface of the `Dlist` class is provided in the `dlist.h`, which is available in the Project-5-Related-Files.zip on Canvas. The code is replicated here for your convenience.

```
#ifndef __DLIST_H__
#define __DLIST_H__

class emptyList {
    // OVERVIEW: an exception class
};

template <class T>
class Dlist {
    // OVERVIEW: contains a double-ended list of Objects

   public:
    // Operational methods

    bool isEmpty() const;
    // EFFECTS: returns true if list is empty, false otherwise

    void insertFront(T *op);
    // MODIFIES this
    // EFFECTS inserts op at the front of the list

    void insertBack(T *op);
    // MODIFIES this
    // EFFECTS inserts op at the back of the list

    T *removeFront();
    // MODIFIES this
    // EFFECTS removes and returns first object from non-empty list
    //           throws an instance of emptyList if empty

    T *removeBack();
    // MODIFIES this
    // EFFECTS removes and returns last object from non-empty list
    //           throws an instance of emptyList if empty

    T *remove(bool (*cmp)(const T*, const T*), T* ref);
    // MODIFIES this
    // REQUIRES there is only one or zero object in the list
    //          satisfying cmp(op, ref) == true
    // EFFECTS traverses through the whole list
    //           if `op` in a node satisfies cmp(op, ref) == true,
    //           removes and returns this object from the list
    //           returns NULL pointer if no such object exists

    // Maintenance methods
    Dlist();                              // constructor
    Dlist(const Dlist &l);                // copy constructor
    Dlist &operator=(const Dlist &l);     // assignment operator
    ~Dlist();                             // destructor

   private:
    // A private type
    struct node {
        node *next;
        node *prev;
        T *op;
    };
```

```
    node *first;  // The pointer to the first node (NULL if none)
    node *last;   // The pointer to the last node (NULL if none)

    // Utility methods

    void removeAll();
    // EFFECT: called by destructor/operator= to remove and destroy
    //           all list elements

    void copyAll(const Dlist &l);
    // EFFECT: called by copy constructor/operator= to copy elements
    //           from a source instance l to this instance
};

/*
Note: as we have shown in the lecture, for template, we also need
to include the method implementation in the .h file. For this
purpose, we include dlist_impl.h below. Please provide the method
implementation in this file.
*/

#include "dlist_impl.h"
#endif /* __DLIST_H__ */
```

The definition of "node", the private type for elements of the container list, is given in the private section of class `Dlist`. This is so to prevent the clients of the class from using that type.

In addition to the six operational methods, there are the usual four maintenance methods: the default constructor, the copy constructor, the assignment operator, and the destructor. Be sure that your copy constructor and assignment operator do **full** deep copies, **including making copies of `T`'s owned by the list.**

Finally, the class defines two private utility methods `removeAll` and `copyAll` that implement the behaviors common to two or more of the maintenance methods.

You must implement each `Dlist` method in a file called `dlist_impl.h`. We will test your `Dlist` implementation separately from the other components of this project, so it must work independently of the two applications described below.

To instantiate a `Dlist` of pointers-to-`int`, for example, you would declare the list:

```
Dlist<int> il;
```

This instructs the compiler to instantiate a version of `Dlist` that contains pointers-to-`int`, and the compiler compiles a version of the `Dlist` template that contains such pointers-to-`int`.

2. Reverse Polish Notation Calculator

2.1 Problem

Arithmetic expressions can be written in different forms. We are used to write them in **infix** notation, i.e., binary operators are written between their operands, such as 1+2. One issue with this notation is that parentheses are sometimes necessary to disambiguate the order of operations, e.g.,

(1+2)*3 vs 1+ (2*3). Arithmetic expressions can also be written in **postfix** notation, also called **Reverse Polish Notation (RPN)**, where operators are written after their operands. A nice property of the postfix notation is that parentheses are not needed anymore, since there cannot be any ambiguity on the order of operations. For instance, the last two expressions would be written as follows: 1 2 + 3 * vs 1 2 3 * +. Another nice property is that it is easy to evaluate such RPN expression with a stack.

In this part, we will write a single program rpn.cpp to

1. convert an infix expression to an RPN one (`getRPN`)

2. evaluate an RPN expression (`getVAL`)

An infix expression is composed of:

1. **Operands** are non-negative integers

2. **Operators** are +, -, *, / (**Precedence**: * / > + -; **Associativity**: all left associative)

3. **Parentheses** are (,)

A postfix expression may only contain operands or operators.

There are many different implementations for `getRPN` and `getVAL`, so we don't provide a starter file. You will probably need **stack**. However, **we require you not to use any container from C++ Standard Template Library (STL). This means you must implement the stack with your own DList.**

2.2 General Steps

Expressions are encoded as C++ strings composed of tokens separated by a space. **For an infix expression, a token may be an operand, an operator, or a parenthesis, while for a postfix expression, a token may only be an operand or an operator.**

You may want to follow the following three steps:

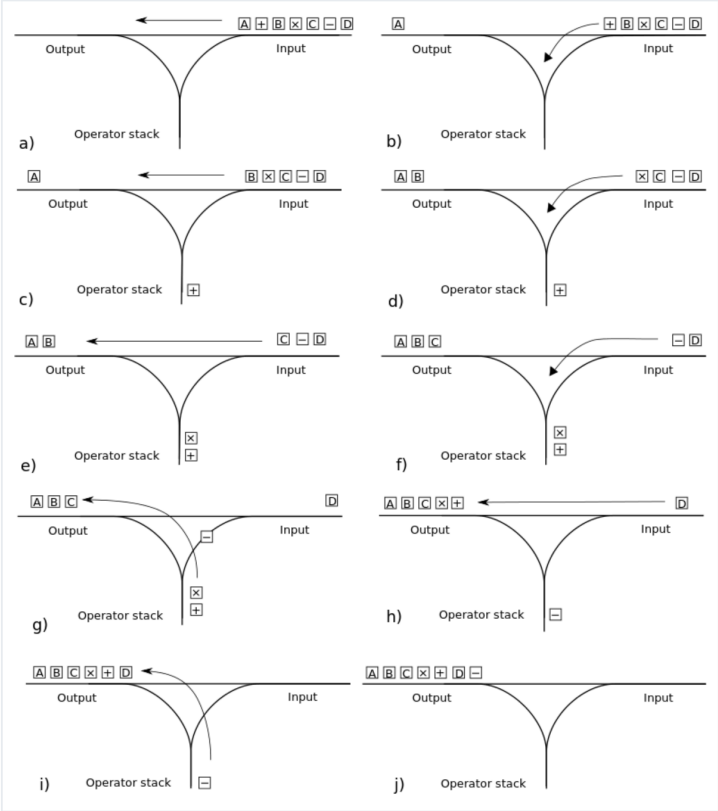
Step 1: Implementing your own template stack class. Use the **DList** you have implemented as a data member in **Stack**, and then offer **push(), pop(), top(), empty()** based on that data member. These operations may also throw exceptions just like DList operations. This should not be very hard because it is just calling some DList methods.

Step 2: Converting an infix expression to an RPN one. This can be achieved by **a char stack** and the **shunting-yard algorithm**. The algorithm scans the infix expression from left to right, and deals with each token. Here is the algorithm from Wikipedia, slightly modified to fit our set-up (compared to wiki version, we don't have functions in infix expression, we don't have variables as operand, and also we use a string as output instead of a queue).

```
while there are tokens to be read:
    read a token.
    if the token is a number, then:
        push it to the output string.
    else if the token is an operator then:
        while ((there is a operator at the top of the operator stack)
              and ((the operator at the top of the operator stack has
```

```
greater precedence)
               or (the operator at the top of the operator stack has equal
precedence and the token is left associative))
              and (the operator at the top of the operator stack is not a
left parenthesis)):
            pop operators from the operator stack onto the output string.
        push it onto the operator stack.
    else if the token is a left parenthesis (i.e. "("), then:
        push it onto the operator stack.
    else if the token is a right parenthesis (i.e. ")"), then:
        while the operator at the top of the operator stack is not a left
parenthesis:
            pop the operator from the operator stack onto the output
string.
        /* If the stack runs out without finding a left parenthesis, then
there are mismatched parentheses. */
        if there is a left parenthesis at the top of the operator stack,
then:
            pop the operator from the operator stack and discard it
/* After while loop, if operator stack not null, pop everything to output
string */
if there are no more tokens to read then:
    while there are still operator tokens on the stack:
        /* If the operator token on the top of the stack is a parenthesis,
then there are mismatched parentheses. */
        pop the operator from the operator stack onto the output string.
exit.
```

An example:



The algorithm and example are referenced from https://en.wikipedia.org/wiki/Shunting-yard_algorithm. You can check the link for more details.

Note one error can occur during the algorithm, “parenthesis mismatch”. More on this in I/O Format part.

Step 3: Compute value based on RPN. Given an RPN expression, it is very easy to evaluate it with **an int stack**. Scan the RPN from left to right. If an operand is found, push it to the stack. If an operator is found, pop the necessary operands from the stack, do the operation, and push back the result. When the scan finishes, what remains in the stack should be the final result.

Note three errors can occur during evaluation, “not enough operands”, “divide by zero”, and “too many operands”. More on this in I/O Format part.

2.3 I/O Format

Input format: You should read the infix expression from standard input. There will always be **a space between two tokens**. The operand will be a **non-negative integer and can be stored with int type (while intermediate value and result might be negative)**.

Output format: There are in total four kinds of errors. T**he first should be detected when converting infix to RPN. The second, the third and the fourth should be detected when computing VAL based on RPN**. Print the error messages to `stdout` rather than `stderr`.

1. When you find parentheses doesn’t match during the conversion from infix to RPN, do:
   cout << "ERROR: Parenthesis mismatch" << endl;

2. When you find there are not enough operands in the int stack for an operator, do:
   cout << "ERROR: Not enough operands" << endl;

3. When you find the operator is trying to divide by zero, do:
   cout << "ERROR: Divide by zero" << endl;

4. When you find the int stack used for evaluation contains more than one value at the end, do:
   cout << "ERROR: Too many operands" << endl;

Whenever an error occurs, you should print the error message and exit the program normally. Also You should pay special attention to memory leak when error occurs.

The output should contain **at most** two lines.

- For the first line. If there is no “**parenthesis mismatch” error**, it should print the RPN expression with a space between tokens, else print the previous corresponding error message.

- For the second line, (this line appears when there is no parenthesis mismatch error). If there is no “**not enough operands” or “division by zero” or “too many operands” error**, print the expression value, else print the corresponding error message.

When computing the expression value, you can assume the following:

- All intermediate values and result can be stored with int type

- The division is the c++ / between two ints

Sample inputs and outputs:

```
12 - 34 + 78 / 56 * 90      # input1
12 34 - 78 56 / 90 * +      # output1 line1
68                          # output1 line2

( 12 + 34 / 56              # input2
ERROR: Parenthesis mismatch # output2 line1
```

Implement your RPN Calculator in a file called `rpn.cpp`. It must work correctly with any valid implementation of `DList`.

2.4 References

Here are some readings on what is RPN, how to convert the infix expression to RPN, and how to get VAL based on RPN.

4. Understand RPN: https://en.wikipedia.org/wiki/Reverse_Polish_notation

5. Convert infix expression to RPN: https://en.wikipedia.org/wiki/Shunting-yard_algorithm

3. Least-Recently-Used Cache Simulation

The memory in a computer is used to store data and instructions and it is constructed as a hierarchy of layers. One of these layers is called main memory, and another layer is called cache, which is smaller but faster than the main memory. Cache is used to store copies of data that are frequently used in main memory so that this part of data could be access more quickly.

The cache stores the copies of data along with their addresses in the main memory. And there is at most one copy in the cache for each data in the main memory. Usually, a *block* is constructed to store continuous data in memory and their address, as well as other useful information. Here for simplicity, we assume a block stores only one datum, i.e., a unit of memory, and its address. Data are stored in the cache in the form of blocks. There are two concepts called *hit* and *miss*. A hit happens when the accessed data has a copy in the cache, i.e. there is a block in cache having the same address as the accessed data. A miss happens when the accessed data is absent in the cache; in that case, the accessed data needs to be copied from the main memory to the cache.

There are two operations on the data in memory. One is *READ* – given an address in the main memory, read the datum at the address. The other is *WRITE* – given an address in the main memory and a datum, write the datum to the address. To reduce access time, the *READ/WRITE* operations are performed only on the copy of the accessed data in the cache. If there is a miss, first copy the data from the main memory to the cache, and then perform *READ/WRITE* operations on the copy in the cache. When a block is removed from the cache because it has not been accessed for a long time, the datum in the block needs to be copied to the corresponding address in the main memory, since it may have been modified by *WRITE* when it is in the cache.

So how to choose the part of data in the main memory to be maintained in the cache? One of the

strategies is called **least recently used (LRU)**. It follows the principle that if a datum is used recently, it may be used again in the following time. To implement LRU, we can maintain the cache as a linked list of blocks, where the most recently used datum in the cache is stored at the front of the linked list and the least recently used datum is at the rear. Every time a datum is accessed, it should be put at the front of the linked list. When the cache is full (remember that the cache size is a lot smaller than the size of main memory) and a datum that is not stored in the cache is accessed, the datum at the rear of the linked list will be removed and this new datum will be inserted at the front.

In this part of project, you will be given a cache size, a main memory size and a series of instructions, and you are going to write a simulator to simulate the *READ/WRITE* processes on the cache and the main memory. The main memory could be represented as an array of `int`; the index in the array is the address of the corresponding data, ranging from 0 to (main_memory_size -1). At the beginning the linked list of the cache is empty, and the data in the main memory is initialized as **0**. As the data are accessed, the size of the linked list grows until it reaches the given cache size.

Besides *READ/WRITE* operations, you need to provide methods to print the contents of the cache and the memory. The format is shown in the description of instructions below.

Your simulator will obtain the cache size and main memory size from the **standard input stream** `cin`, following by a series of instructions. There are five types of instructions:

```
READ <address>
```

- Read the data from the `address` and print the data

```
WRITE <address> <data>
```

- Write the `data` to `address`

```
PRINTCACHE
```

- Print the content of the cache. Each line contains the `<address>` and the `<data>` of one block and blocks are ordered from the most recently used to the least recently used. For example, if the content of the cache is:

  where the left most block is the most recently used one and the right most block is the least recently used one, the printed message will be:

```
1 80
3 70
4 3
```

```
PRINTMEM
```

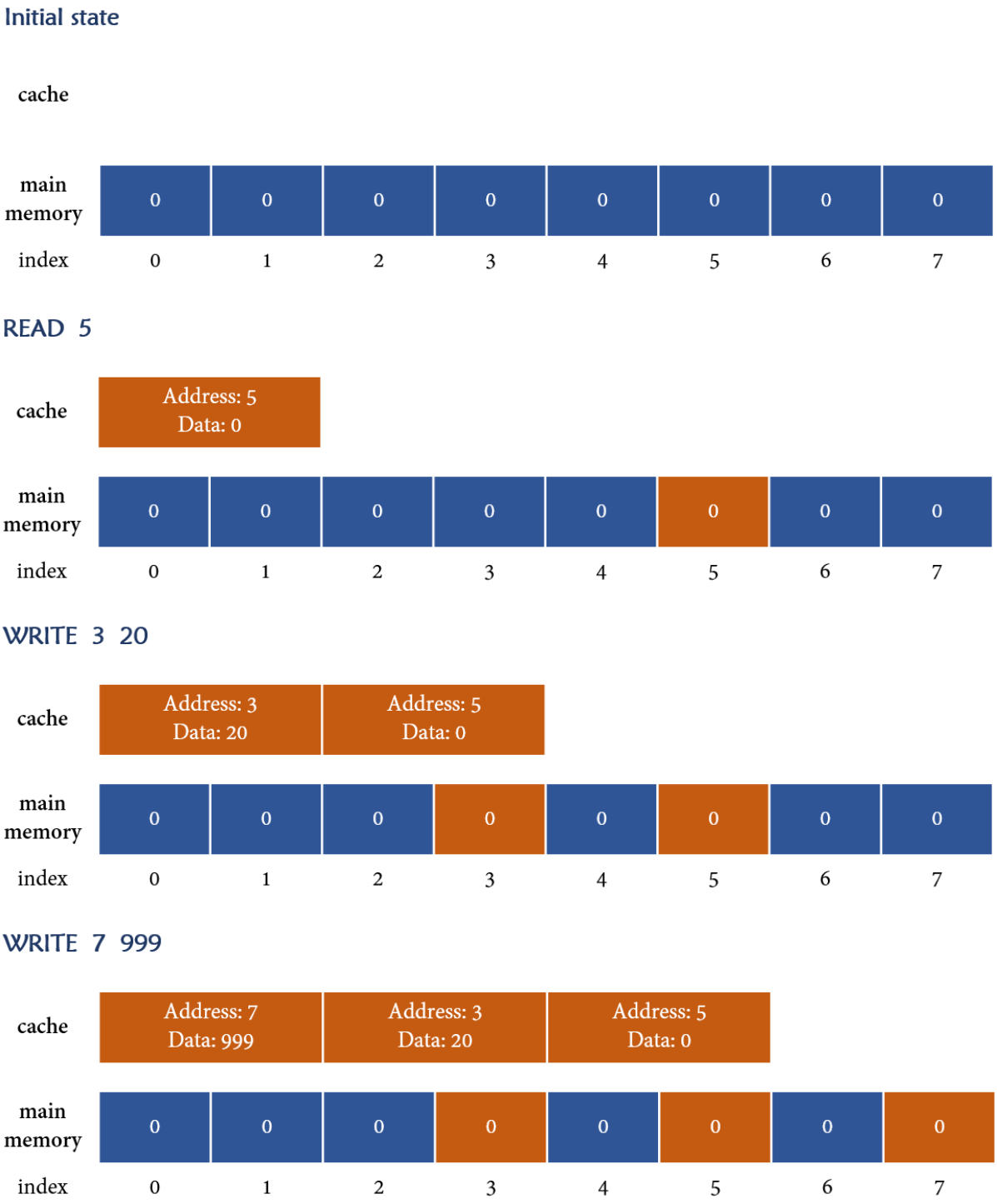
- Print the content of the memory in a line, starting from address 0 and separating adjacent data with a space. For example, if the content of the memory is {0, 0, 2, 0, 0, 4, 0, 0, 0, 0}, the output will look like:

```
0 0 2 0 0 4 0 0 0 0
```

```
EXIT
```

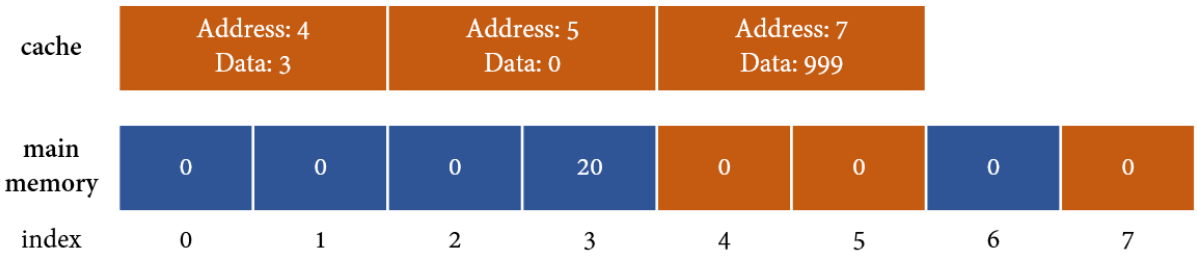
- Stop obtaining instructions from `cin` and exit the program.

A demo is shown below. The size of cache is 3 and the size of memory is 8:

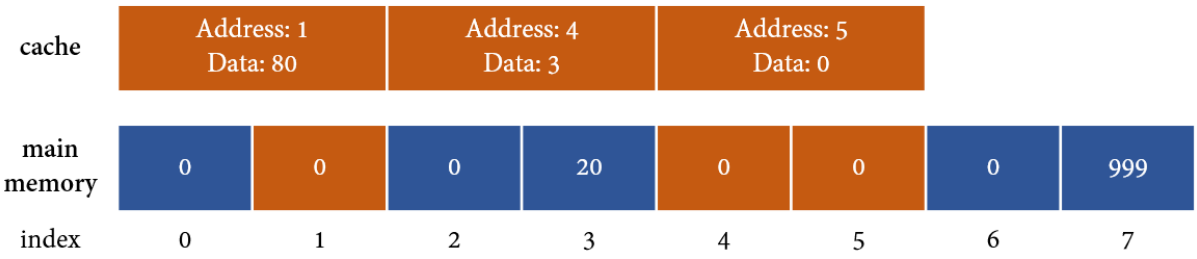


READ 5

WRITE 4 3



WRITE 1 80



READ 3

WRITE 3 70

READ 1

WRITE 0 8

There are three error messages to report:

1. If the input instruction is none of the above, e.g. "`WRITECACHE`", leave the cache and the main memory unchanged, advance to the next input, and use the following command to print the following message:

```
cout << "ERROR: Unknown instruction" << endl;
```

2. If the input instruction requires more operands than are present, e.g. "`WRITE 0`", leave the cache and the main memory unchanged, advance to the next input, and use the following command to print the following message:

```
cout << "ERROR: Not enough operands" << endl;
```

3. If the input instruction requires less operands than are present, e.g. "`READ 0 0`", leave the cache and the main memory unchanged, advance to the next input, and use the following command to print the following message:

```
cout << "ERROR: Too many operands" << endl;
```

4. If the given address in `READ/WRITE` instructions is out of bound, e.g. "`READ 10`" when the size of the memory ≤ 10, leave the cache and the main memory unchanged, advance to the next input, and use the following command to print the following message:

```
cout << "ERROR: Address out of bound" << endl;
```

Below is an example of inputs and outputs:

Inputs:

```
3 8
```

```
WRITE 5 1
PRINTCACHE
READ 3
READ 9
PRINTCACHE
PRINTMEM
WRITEC 5
READ
WRITE 0 1
WRITE 1 2
PRINTCACHE
WRITE 0 3
WRITE 5 3
PRINTCACHE
PRINTMEM
READ 0
READ 7
PRINTCACHE
PRINTMEM
EXIT
```

Outputs:

```
5 1
0
ERROR: Address out of bound
3 0
5 1
0 0 0 0 0 0 0 0
ERROR: Unknown instruction
ERROR: Not enough operands
1 2
0 1
3 0
5 3
0 3
1 2
0 0 0 0 0 1 0 0
3
0
7 0
0 3
5 3
0 2 0 0 0 1 0 0
```

You are welcome to construct the code structure as you want. If you have no idea about where to start, you could refer to the class below:

```
class LRUCache{
private:
    int mem_size; // size of memory
    int *memory;
```

```
    struct block{
        int address; // its address in memory
        int data;
    };
    Dlist<block> cache;
    int cur_cache_size;     // current length of `cache`
    int max_cache_size;     // maximum length of `cache`

    static bool compare(const block *a, const block *b);
    // EFFECTS: returns true if two blocks have the same address

public:
    LRUCache(int cache_size, int memory_size); // constructor
    // Initialize `cur_cache_size`, `max_cache_size`, `memory`
    // Initialize all elements in `memory` to 0

    ~LRUCache(); // destructor

    int read(int address);
    // EFFECTS: returns data corresponding to address,
    //          0 <= address < mem_size;
    //           if address is out of bound, throws an exception
    //
    // if hit,
    //  removes this block and insert it to the front;
    //  returns `data`;
    // if miss,
    //    if `cur_cache_size` equals to `max_cache_size`,
    //      removes the last (least recently used) block
    //    in the `cache`;
    //   writes data in the last block
    //   to the corresponding address in `memory`;
    //    if `cur_cache_size` < `max_cache_size`,
    //     increment `cur_cache_size` by 1;
    //    reads `data` of `address` from `memory`,
    //    inserts the block with `address` and `data`
    //   to the front of `cache`;
    //  returns `data`

    void write(int address, int data);
    // EFFECTS: writes data to address, 0 <= address < mem_size
    //            if address is out of bound, throws an exception
    //
    // if hit,
    //  removes this block from list,
    //   writes `data` to this block,
    //  and inserts this block to the front;
    // if miss,
    //    if `cur_cache_size` equals to `max_cache_size`,
    //      removes the last (least recently used) block
    //      in the `cache`;
    //   writes data in the last block
    //    to the corresponding address in `memory`;
```

```
    //    if `cur_cache_size` < `max_cache_size`,
    //     increment `cur_cache_size` by 1;
    //    inserts the block with `address` and `data`
    //    to the front of `cache`

    void printCache();
    // EFFECTS: prints the cache in given format

    void printMem();
    // EFFECTS: prints the memory in given format
};
```

Implement your simulator in a file called `cache.cpp`. It must work correctly with any valid implementation of `DList`.

III. Implementation Requirements and Restrictions

- You must use your `Dlist` container to implement both your stack in the RPN Calculator and your queue(s) in the LRU Cache Simulator. You may use any type you see fit as the type `T` in the template for each application. However, remember that you only insert and remove **pointers-to-objects**. You must use the at-most-once invariant plus the Existence, Ownership, and Conservation rules when using your `Dlist`. Therefore, you can only insert dynamic objects.

- You may not leak memory in any way. To help you see if you are leaking memory, you may wish to call `valgrind`, which can tell whether you have any memory leaks. The command to check memory leak is:

  ```
  valgrind --leak-check=full <PROGRAM_COMMAND>
  ```

  You should change `<PROGRAM_COMMAND>` to the actual command you use to issue the program under testing. For example, if you want to check whether running program

  ```
  ./cache < input-file
  ```

  causes memory leak, then `<PROGRAM_COMMAND>` should be `./call < input-file`.

  Thus, the command to check memory leak is

  ```
  valgrind --leak-check=full ./cache < input-file
  ```

  Another way to check memory leak is to compile with -fsanitize flag in g++. The program compiled in this way will report an error if it exits with memory leak. An example:

  ```
  g++ -Wall -Werror -O2 --std=c++17 -fsanitize=leak -fsanitize=address -o rpn rpn.cpp
  ```

  See http://gavinchou.github.io/experience/summary/syntax/gcc-address-sanitizer/ for details if you are interested.

- You must fully implement the `Dlist` ADT. Note that the implementations of the RPN Calculator and LRU Cache Simulator **may not exercise all of a `Dlist's` functionality**, but

this is fine.

- You may `#include <iostream>`, `<string>`, `<sstream>`, `<cstdlib>`, and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library.

- Input and output should only be done where it is specified.

- You may not use the `goto` command.

- You may not have any global variables that are not `const`.

IV. Source Code Files and Compiling

There is one header file `dlist.h` located in `Project-5-Related-Files.zip` from our Canvas Resources. You should copy `dlist.h` into your working directory. **DO NOT modify it!**

You need to write three C++ source files: `dlist_impl.h`, `rpn.cpp`, and `cache.cpp`. They are discussed above and summarized below:

1. `dlist_impl.h`: implementation of the `Dlist` methods.

2. `rpn.cpp`: implementation of the RPN Calculator.

3. `cache.cpp`: implementation of the LRU cache simulation.

In order to guarantee that JOJ compiles your program successfully, you should name your source code files exactly like how they are specified above. JOJ will test your implementation of

1. `Dlist` methods by building a program from our `dlist.h`, **your** `dlist_impl.h`, and **our** test file `test.cpp`.

2. RPN Calculator by building a program from our `dlist.h`, **our** `dlist_impl.h`, and **your** `rpn.cpp`.

3. LRU cache simulation by building a program from our `dlist.h`, **our** `dlist_impl.h`, and **your** `cache.cpp`.

Note that when testing your RPN Calculator and LRU cache simulation, we use the `dlist_impl.h` from us, not yours. This means that your implementation of the RPN Calculator and LRU cache simulation should be independent of the actual implementation of `Dlist`. It should work for any correct implementation of `Dlist`.

To compile a program that uses `Dlist`, you need to include `dlist.h`.

To compile the RPN Calculator program named `rpn`, type the following command:

```
g++ -Wall -Werror -O2 --std=c++17 -o rpn rpn.cpp
```

To compile the LRU cache simulation program named `cache`, type the following command:

```
g++ -Wall -Werror -O2 --std=c++17 -o cache cache.cpp
```

V. Testing

We provide you with a file called `test.cpp` in `Project-5-Related-Files.zip` to help you test a few very basic behaviors of the `Dlist` ADT. If `test.cpp` compiles successfully, run the program. After running the program, you can look at the return value of the program to see if your implementation of the `Dlist` ADT passes this test or not. If the return value is 0, it passes the test; otherwise it fails the test.

In Linux you can check the return value of a program by typing

```
echo $?
```

immediately after running the program.

We have also supplied an input file called `sample` for you to test your LRU cache simulation program named `cache`. To do this test, type the following into the Linux terminal once your program has been compiled:

```
./cache < sample > test.out

diff test.out sample.out
```

If the `diff` program reports any differences at all, you have a bug.

These are the minimal amounts of tests you should run to check your program. Programs that do not pass these tests are not likely to receive much credit. However, programs that pass these tests are not guaranteed to receive full credits (i.e., pass all the test cases) on JOJ. You should also write other different test cases yourself to test your program extensively.

You should also check whether there is any memory leak using `valgrind` or `g++ -fsanitize` as we discussed above. For those programs that behave correctly but have memory leaks, they only get half of the grade.

VI. Submitting and Due Date

You should submit three source code files: `dlist_impl.h`, `rpn.cpp`, and `cache.cpp`. These files should be submitted as a tar file via the online judgment system. See the announcement from the TAs for details about submission. The due date is 11:59 pm on August 4, 2020.

VII. Grading

Your program will be graded along three criteria:

1. Functional Correctness

2. Implementation Constraints

3. General Style

Functional Correctness is determined by running a variety of test cases against your program, checking against our reference solution. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leak. **For those programs that behave correctly but have memory leaks, they will only get half of the grade.** General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will lead to General Style deductions.