Project Five: List, Stack, and Queue

Out: July 15th, 2022; Due: 23:59 July 29th, 2022

Background

This project will give you experience in implementing a templated double-ended doubly-linked list (or `Dlist`) and building a stack machine with the double-ended doubly-linked list you just implement.

***Reminder: You need to finish Task 1 to proceed to Task 2.**

Task 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.

Note that while this list is templated with the contained type, `T`, it is a container of **pointers-to-T**, not a container of instances of T. Insertion takes a **pointer-to-T** as an argument and puts **that pointer** into the node to be inserted. Removal removes a node and returns the pointer-to-T kept in that node. The implementation of `Dlist` abides by the following principle: 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 `Dlist.h`, which is available in `Project-5-Related-Files.zip` on Canvas.

The definition of `struct node`, the private type for the nodes 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 five operational methods, there are the usual four maintenance methods: the default constructor, the copy constructor, the assignment operator, and the destructor. Make sure that your copy constructor and assignment operator do **full** deep copies, **including making copies of `T`'s owned by the list**. Also make sure that when you implement the destructor, you need to free all allocated memory, **including the T pointer stored in each node**.

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 `DlistImpl.h`. We will test your `Dlist` implementation separately from the other components of this project, so it must work independently of the application 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`.

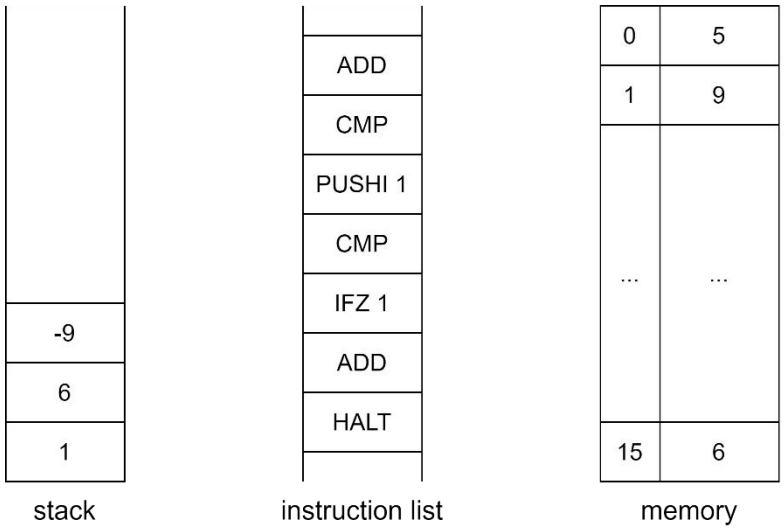
Task 2: Stack Machines

Problem

Most computers are built according to the von Neumann architecture, which consists of a central processing unit (CPU), a memory unit, and I/O devices. This architecture is fast and robust in general. Therefore, many instruction set architectures (ISAs) like ARM, MIPS, and so on, are designed based on it. However, another type of architecture is popular for virtual machines: stack machine. One of its main components is a stack to store operands. All operations can be executed just using this stack. Therefore, this architecture is simple to implement. Java virtual machine (JVM) and WebAssembly (Wasm), which are widely used nowadays, are both based on abstract stack machines. In this part, we plan to use our own `Dlist` class to implement a simple but powerful stack machine by ourselves.

General Computational Model

Our computational model is based on a simplified stack machine. It contains an operand stack, an instruction queue (or list), and a memory array. The structure is shown below.



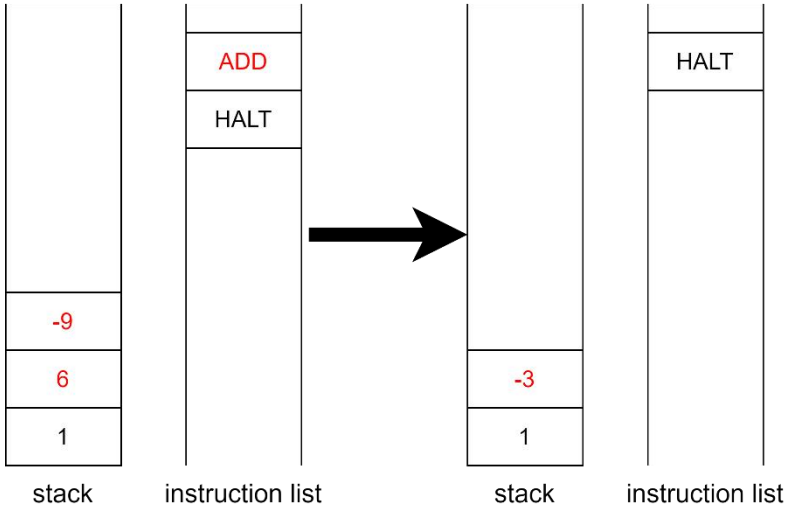
To simplify our model, we only have 32-bit integers in the stack and memory. The memory is only 64 bytes (recall a byte = 8 bits), hence, we have 16 addresses where each address corresponds to a 32-bit integer (16 addresses = 64 bytes * 8 bits per byte / 32 bits per address). To compute on this machine, we always pop the first instruction from the queue and execute this instruction accordingly. We now introduce those instructions.

Instruction Set

This machine has an instruction set that contains many operations like arithmetic operations, control operations, memory operations, and so on.

Arithmetic Operations: ADD,NOR

This kind of operations is designed to deal with the operand stack. For example, `ADD` means popping up two elements from the stack and then pushing the sum of these two elements into the stack. The detail is shown below:



At first, the stack is `[n1, n2, ...]`. And after executing `ADD`, the stack is `[n1+n2, ...]`.

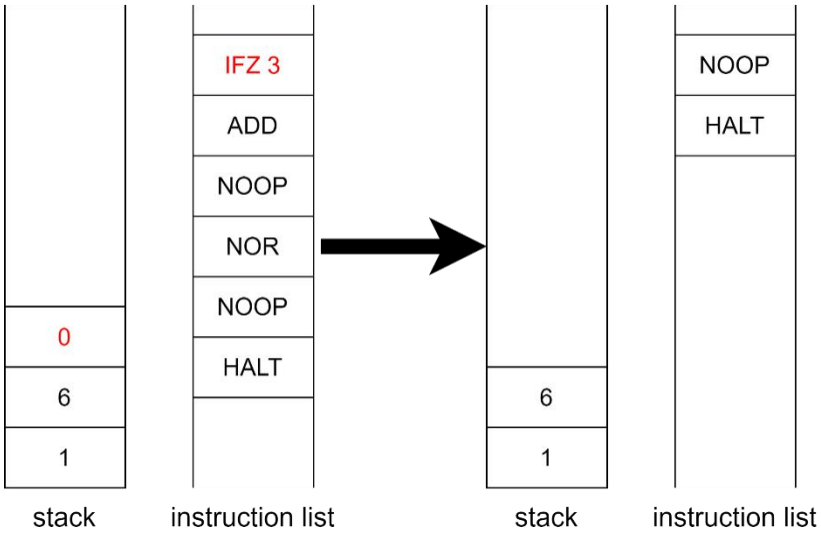
Another operation is the bitwise Boolean operator `NOR`. Its truth table is shown below:

n1	n2	n1 NOR n2
0	0	1
0	1	0
1	0	0
1	1	0

For example, `5 NOR 7 = -8` because `5` is represented in binary by `00000101`, `7` by `00000111`, and `00000101 NOR 00000111 = 11111000`, which is `-8` in the 2's complement representation. In C++, you may use `int result = ~(a|b);` to perform the `NOR` operation. This instruction runs similar to `ADD`, hence, `[n1, n2, ...] -> [n1 NOR n2, ...]`.

Control Operations: IFZ,HALT

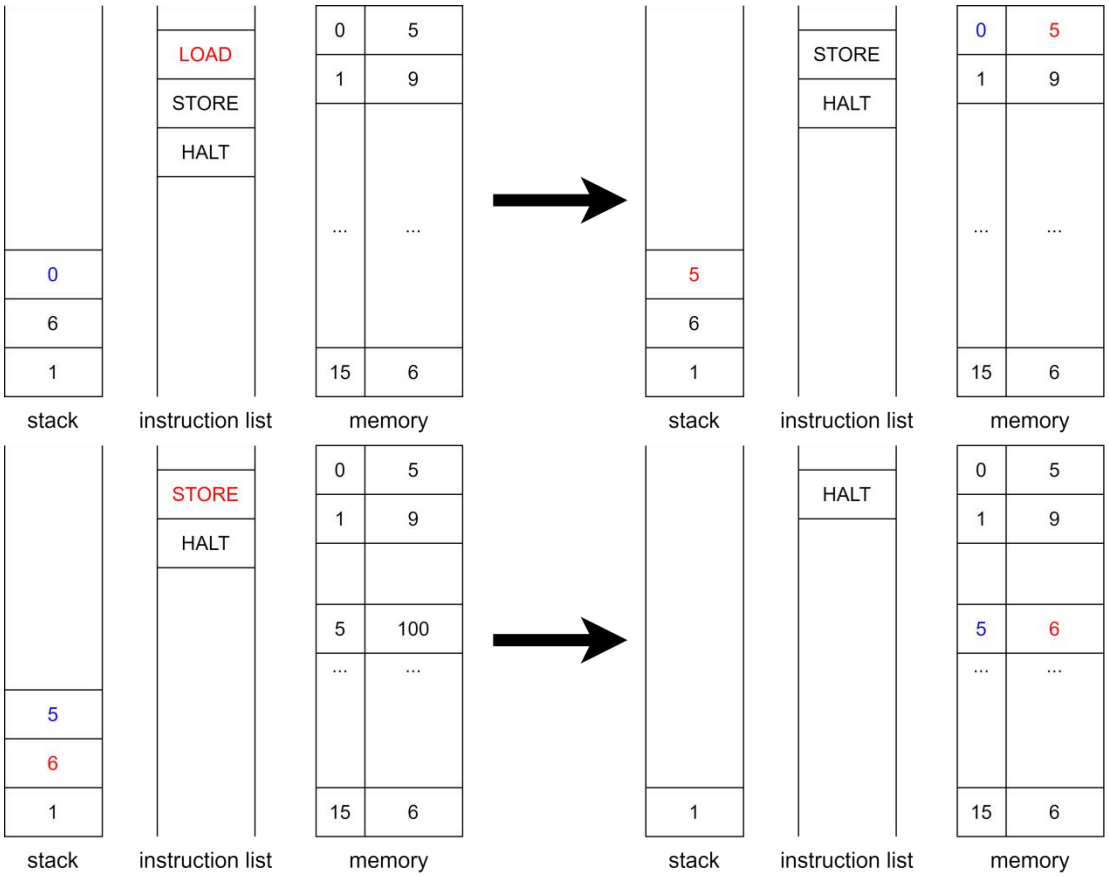
A control operation is used to control the instruction queue. For example, `HALT` means to halt the whole machine, hence, stopping the program. Another non-trivial instruction is `IFZ`. It also has an argument `n`. When the front of the queue is `IFZ n`, we first pop up the first element in the stack. If this element is zero, we pop up the following `n` instructions in the queue. If this element is not zero, we continue executing the next instruction. An example is shown below.



Memory Operations: LOAD, STORE

A memory operation is designed for operating with the memory. For example, we want to get a value stored at an address or store a value at an address.

`LOAD` is to get the value stored in the memory. When executing it, we first pop up an element from the stack. This first element is the `address`. Then, we find the value stored at this address and push it into the stack. `STORE` means to store a value in the memory. When executing it, we first pop up **two elements**. The first element is the `address` and the second is the `value`. If we declare the memory as `int mem[16];`. This means `mem[address] = value`. An example is shown below.



Stack Operations: POP, PUSHI

These two operations can control the elements in the stack directly. `POP` means to pop up an element directly and `PUSHI n` means to push a constant `n` into the stack.

Other Operation: NOOP

`NOOP` does nothing. Just read the operation and skip it.

Setup

As an ideal model, we assume that the stack and queue have an unlimited capacity. In the beginning, the instruction queue, operand stack, and memory are initialized. Then, the stack machine starts by executing the instructions from the queue. Further details are provided in the I/O Format section below.

I/O Format

Input

In the first line, there are two numbers `n`, `m` where `n` is the number of elements in the stack, and `m` is the number of elements in the queue. In the second line, there are `n` numbers divided by a space. You may push them in sequence in the stack. **Hence, `1 2 3 4` means that `4` is on the top and `1` is at bottom of the stack.** In the next

following `m` lines, there are `m` instructions. In the header file `Instr.h`, we provide a data structure `Instr` to hold values representing instructions. You can use the functions provided in it directly, e.g.,

```
Instr ins;
std::cin >> ins;
```

In the last line, there are 16 numbers. They are shown as `mem[0] mem[1] ... mem[15]`. Here is a sample input:

```
5 14
0 1 2 3 4
PUSHI 0
LOAD
PUSHI 1
LOAD
PUSHI 1
LOAD
NOR
PUSHI 1
ADD
ADD
PUSHI 2
STORE
NOOP
HALT
999 888 1 2 0 0 0 0 0 0 0 0 0 0 0 1
```

You do not need to check for errors. Whenever there is an instruction like `ADD`, the number of elements in the stack would be enough. Also, when the instruction is `LOAD` or `STORE`, the top of the stack is between 0 and 15. The input will ensure the machine will halt.

Output

When there is no argument provided, **after** executing **each** instruction, you should print the machine status. It consists of four lines: instruction that is executed, stack, queue, and memory. A sample is provided below:

```
PUSHI 1
1 2 3 4 999 1
LOAD PUSHI 1 LOAD NOR PUSHI 1 ADD ADD PUSHI 2 STORE NOOP HALT
999 888 1 2 0 0 0 0 0 0 0 0 0 0 0 1
```

Similarly, the output for `Instr` is provided, you can use it by:

```
Instr ins;
// initialize ins
std::cout << ins;
```

You may also use the operator `<<` overloaded for `Dlist<T>` to output:

```
Dlist<int> intList;
std::cout << intList;
```

Argument

When the program gets an argument `-s`, it doesn't need to output anything after executing each instruction. You only need to print the machine status when the machine halts. The output consists of **two** lines: stack and memory. Here is an example of a sample input:

```
$ ./sam -s < ./sample.in
1 2 3 4
999 888 111 2 0 0 0 0 0 0 0 0 0 0 1
```

General Steps

- You may first implement two classes `Stack` and `Queue` by using your own `DList<T>` class. They may contain methods like `push(T x)` to operate on elements.
- Initialize three data structures (i.e., stack, queue, array) by reading from the inputs. You may use `push` for queue and stack to add elements. The format is shown in the I/O Format section.
- Implement the stack machine. After executing each instruction, you should print the machine status.

Implementation Requirements and Restrictions

- **You must use your `Dlist` container to implement both your stack and queue.** 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 run the program under testing.

  Another way to check memory leaks is to compile with the `-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=addres
  s -o sam sam.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 stack machine may not exercise all of a `Dlist`'s functionality, but this is fine.
- You may `#include <iostream>`, `<string>`, `<sstream>`, `<cstdlib>`, `<algorithm>` 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`.

Source Code Files and Compiling

There are two header files `Dlist.h`, `Instr.h` located in `Project-5-Related-Files.zip` from our Canvas Resources. Please do not modify them. We will use your `DlistImpl.h` and `sam.cpp` to compile. So, no other files will be compiled.

Submitting and Due Date

You should submit two source code files: `DlistImpl.h` and `sam.cpp`. These files should be submitted as a tar or zip file via the online judgment system. The due date is 23:59 on July 29th.

Testing

You should write your own testcases to cover all functions.

There are three files `sample.in` , `sample.out`, and `sample_s.out` in the starter file folder. They are sample cases for the stack machine part, which can help to make sure your output format is correct.

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, and 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 any memory leaks. 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.