1 Introduction

After learning queue and stack in class, we are wondering if there is something more flexible. Maybe a list supporting pop and push on both sides sounds great. So we decide to write a program to implement this data structure, which supports 4 operations:

- +s : push s to the left side
- - : pop a char from the left side and print it
- [s : push s to the right side
-] : pop a char from the right side and print it

Here, the list is empty at first, and s can be letters either lowercase or uppercase. If the list pops when empty, just print a as the result.

2 Algorithm Specification

This data structure should implement four operator: Lpush, Rpush, Lpop, Rpop

Initially, let's ignore the detail implementation of the data structure. It's true that we can use this data structure with those four operator to implement the program. When the instruction inputted, we choose to store all the instructions into the memory. After confronting the Enter, the program stop reading. Then according to the instruction inputted to use Lpush, Rpush, Lpop, Rpop operators.

Here is the pseudocode:

```
1         i ← 0
2         c ← input
3         while(c!=\n)
4             memory[i++] ← c
5             c ← input
6         i ← 0
7         c ← memory[i++]
8         while(c!=0)
9             if c == '+' Lpush(memory[i++])
10            if c == '[' Rpush(memory[i++])
11            if c == '-' Lpop
12            if c == ']' Rpop
13
```

Therefore, what matters is the implementation of the data structure. In this program, we use a queue which can be enqueued and dequeued in both side. We use the front point and the back point to do operations to the queue. when front point is just behind the back point, the queue is empty.

Here is the pseudocode of the push and pop is following (take right side as an example)

```
1         push: R0 is the element , front_ptr is the point to the front of queue
2         *(--front_ptr) ← element
3
```

```
1         pop: front_ptr is the point to the front of queue,
2             back_ptr is the point to the back of queue
```

```
        if front_ptr == back_ptr + 1
            R0 ← '_'
            trap x21
        else
            R0 ← *(front_ptr++)
            trap x21

```

---

3 Q and A

- Q: what is the data structure you use?

  A: in the program, the data structure we use is much like a combination of queue and stack. because in each side, the operation is like stack while both side can operate push and pop.

  we use two points point to the front of the queue and the back of queue.when doing push and pop, the point will change. when the front point is just behind the back point, it means the queueu is empty.

4 essential parts of code

Fig 1 is the implement of check operation

```
                        ld r1 Input_InputBuff_Addr      ;r1 point to inputbuff

Exe_Loop                ldr r2,r1,#0                    ;read a character into r2
                        brz Exe_End                     ;no characters remain

                        ld r3 Left_Add                  ;check the character
                        add r0,r2,r3
                        brz Exe_Lpush                   ;do Lpush

                        ld r3 Left_Sub
                        add r0,r2,r3
                        brz Exe_Lpop                    ;do Lpop

                        ld r3 Right_Add
                        add r0,r2,r3
                        brz Exe_Rpush                   ;do Rpush

                        ld r3 Right_Sub
                        add r0,r2,r3
                        brz Exe_Rpop                    ;do Rpop
```

Figure 1: implement

Fig 2 is the Lpop function

Fig 3 is the Lpush function

```
Lpop                    st r7 Lpop_Temp
                        jsr RegSave
                        ld r7 Lpop_Temp

                        ldi r1 FrontPtr_Addr        ;r1 is front ptr
                        ldi r2 BackPtr_Addr         ;r2 is the back ptr
                        add r0,r2,#1                ;r0 = -(back + 1)
                        not r0,r0
                        add r0,r0,#1
                        add r0,r1,r0                ;r0 = front - (back + 1)
                        brz Lpop_Empty
                        ldr r0,r1,#0
                        trap x21
                        add r1,r1,#1
                        br Lpop_End
Lpop_Empty              ld r0 SubLine
                        trap x21
Lpop_End
                        sti r1 FrontPtr_Addr

                        st r7 Lpop_Temp
                        jsr RegCov
                        ld r7 Lpop_Temp
                        ret
Lpop_Temp               .BLKW 1
```

Figure 2: Lpop

```
Lpush                   st r7 Lpush_Temp
                        jsr RegSave
                        ld r7 Lpush_Temp

                        ldi r1 FrontPtr_Addr        ;r1 is front ptr
                        add r1,r1,#-1
                        str r0,r1,#0
                        sti r1 FrontPtr_Addr


                        st r7 Lpush_Temp
                        jsr RegCov
                        ld r7 Lpush_Temp
                        ret

Lpush_Temp              .BLKW 1
FrontPtr_Addr           .FILL frontPtr
```

Figure 3: Lpush