Chapter 8 Data Structures

8.4 The Queue

8.4.1 The defining notion of a queue

FIFO, that is the first thing you stored in the queue is the first thing you remove from it.

8.4.2 Two typical operations on stack

- **Insert**: to store a value into the queue, it will be at the rear of the stack

- **Remove**: to get and remove the value of the front of the queue.

8.4.3 Implementation in Memory

We use $R3$ as our **FRONT** pointer and $R4$ as our **REAR** pointer.

- **FRONT**: point to the location **just before** the first element of the queue.
- **REAR**: point to the location containing the most recent element that was added to the queue.

8.4.4 Wrap-Around

Target: To make the location that got free for removing an element available for storing new element.

Strategy: Allow the available storage locations to ***wrap around*** by having our removal and insertion algorithms test the contents of FRONT and REAR for the value of the bottom of queue.

Remove using Wrap-Around

Suppose the bottom of the queue is x8005.

```
        LD      R2, LAST
        ADD     R2, R3, R2
        BRnp    SKIP_1
        LD      R3, FIRST
        BR      SKIP_2
SKIP_1  ADD     R3, R3, #1
SKIP_2  LDR     R0, R3, #0   ; R0 gets the front of the queue
        RET
LAST    .FILL   x7FF8        ; Last contains the negative of x8005
FIRST   .FILL   x8000        ; First contains the value of top
```

Insert using Wrap-Around

Suppose the bottom of the queue is x8005.

```
        LD      R2, LAST
        ADD     R2, R4, R2
        BRnp    SKIP_1
        LD      R4, FIRST
        BR      SKIP_2
SKIP_1  ADD     R4, R4, #1
SKIP_2  LDR     R0, R4, #0   ; R0 gets the front of the queue
        RET
LAST    .FILL   x7FF8        ; Last contains the negative of x8005
FIRST   .FILL   x8000        ; First contains the value of top
```

By using Wrap-Around, we can store $n-1$ elements into a queue which occupies for n locations.

8.4.5 Test for Underflow & Overflow

Underflow -- Empty -- Remove

Condition: **FIRST == REAR**

Overflow -- Full -- Insert

Condition: **FIRST == REAR + 1**

8.4.6 The Complete Story

Chapter 8.4.5 in the 3rd textbook is recommend reading. The **undo** operation for overflow should be paid more attention to.

8.5 Character String

1. The string should end with x0000.
2. Each character in the string only use low 8-bits to store its ASCII value, which means its high 8-bits is x00.
3. Try to detect the difference between TRAP x22 & Trap x24. *(You can read A.3)*

8.6 List

There are two ways to organize a list.

If a list obey a order based on some variable (like non-decreasing order on a number variable or alphabetic order on a string variable), we call it a ordered list.

Two typical operations on List:

- **Access**: it can also be named as Find. To get the location of some node.
- **Update**: it contains two operations actually: insert & delete.

8.6.1 Sequentially Storage

We hold a period **sequential** locations to store the pointers pointing to the start address of nodes and make the pointers in order of the order that the list obeys.

> *Actually it is a pointer array.*

- **Good for Access Operation**: we can do randomly access on the pointers, and we can use **binary search** to find the node we need.

  > *If there are N elements, we need at most $log_2 N$ turns to find the element by using binary search.*

- **Bad for Update Operation**: we must **move all the elements after the element** that we insert into the list or delete from the list.

8.6.2 Linked List

We add a pointer pointing to **Next Node** to each node, the last node points to x0000 so that we do not need hold all the pointers in order.

- **Good for Update Operation**: we can easily insert or delete a node by just considering the nodes before and after the node. We do not need move all the nodes after it.
- **Bad for Access Operation**: we cannot access the nodes randomly. We can only access the node one by one from the head & we cannot access the node before current node directly.

NOTE: The start address of each node is arbitrary **both in** Sequentially Storage & Linked List.

8.7 Array

8.7.1 1-dimension Array

Just like character string & sequentially storage list, we can get A[n] by accessing $A + n$

8.7.2 2-dimension Array

For 2-dimension Array, we have two storage strategy: Row Major & Column Major. In LC-3, we use Row Major.

So, for $A[M, N]$ (it means that at most M+1 rows & N+1 columns), we can access $A[i, j]$ by accessing $A + i * M + j$.

8.7.3 3-dimension Array

Similar to 2-dimension Array, for $A[M, N, P]$, we can access $A[i, j, k]$ by accessing $A + i * (N * p) + j * P + k$

NOTE: in the formulas stated above, we suppose each element just occupies for 1 location , that is, 16 bits. If each element in array needs more than 1 location, the number of locations it need should be taken into consideration.