[CSED233-01] Data Structure

List, Stack, and Queue

Jaesik Park

POSTECH

Attendance Check

- If you are having trouble with the electronic attendance check
  - Check with TA
  - TA will be at the entrance gate
    - Before ~9:30. No problem
    - 9:30~9:45. Late
    - 9:45~ Absent
    - More than 3 absents without proper reason: F
  - We will not accept the attendance issue after the class finishes.

Academic Dishonesty

- Assignments, exams, …
- This is just a class. Do **not try to risk your entire school life**!
- Reported cases will be judged by POSTECH's Committee
- We won't accept any execuse

(Linear) Lists

- List $L = <a_1, a_2, \ldots, a_n>$
  - a finite, *ordered* collection of elements
  - n: length (size) of the list
    - empty list <> : $n = 0$ (no elements)

- Position of a_i is i
  - head (front) ↔ tail (rear)
  - "*current*" position

    <20, 23, | 12, 15> : separated by *fence*

    <20, 23, | 10, 12, 15> after insertion of 10 (at "current" position)

head ↓ $<a_1, a_2, \ldots, a_n>$ ↓ tail

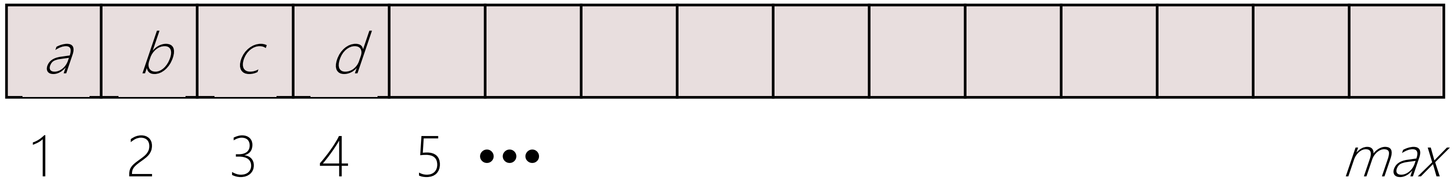
- Sorted list ↔ Unsorted list
- Don't be confused, ordered and sorted mean different things

List Operations

- $L = <a_1, a_2, \ldots, a_{p-1}, a_p, a_{p+1}, \ldots, a_n>$
  - *Insert(x, p, L).* : insert x at position p in list L
    - $<a_1, a_2, \ldots, a_{p-1}, x, a_p, \ldots, a_n>$
  - *Delete(p, L)* : delete element at position p in L
    - $<a_1, a_2, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n>$
  - *Next(p, L)* : returns the position or pointer immediately following position p
  - *Previous(p, L)* : returns the position or pointer previous to p
  - *Locate(x, L)* : returns the position or pointer of x on L
  - *Retrieve(p, L)* : returns element at position p on L
  - *MakeNull(L)* : causes L to become an empty list and returns position END(L)
  - *First(L)* : the first position on *L*
- This is just one example! There is no rule about the list's function names, arguments, their behavior, and return types.
- Check out STL's vector ☺

Array-Based List Implementation (1)

- *L* = <*a, b, c, d*>
  - char L[max]



  - Integer *size* = 4 or
    - Integer *last* = the position of the last element

- *Insert (x, 2, L)* ?

Array-Based List Implementation (2)

- *L* = <*a*, *b*, *c*, *d*>
  - char L[max]

a	*b*	*c*	*d*											
1	2	3	4	5	•••									*max*

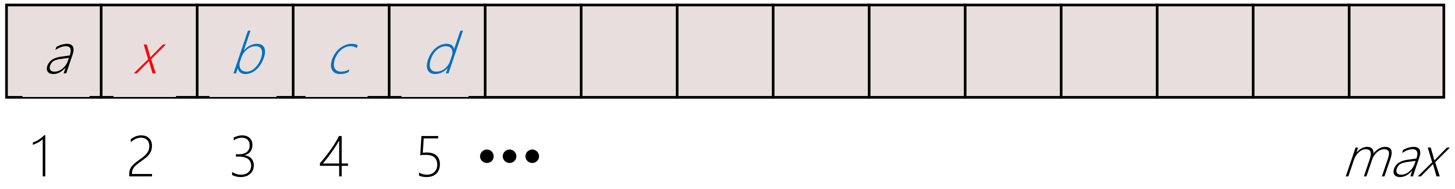
  - Integer *size* = 4 or
    - Integer *last* = the position of the last element

- *Insert (x, 2, L)* ?

a		*b*	*c*	*d*										
1	2	3	4	5	•••									*max*

Array-Based List Implementation (3)

- *Insert (x, 2, L)*
  - *void insert(char x, int i, char* L)*



  - Before insertion, we need to *shift right* all following elements by one position
  - *size* = 5
  - Running time? O(*n*) – Linear time

- *Delete(p, L), Locate(x, L)*
  - Running time? O(*n*) – Linear time
- What does this mean?

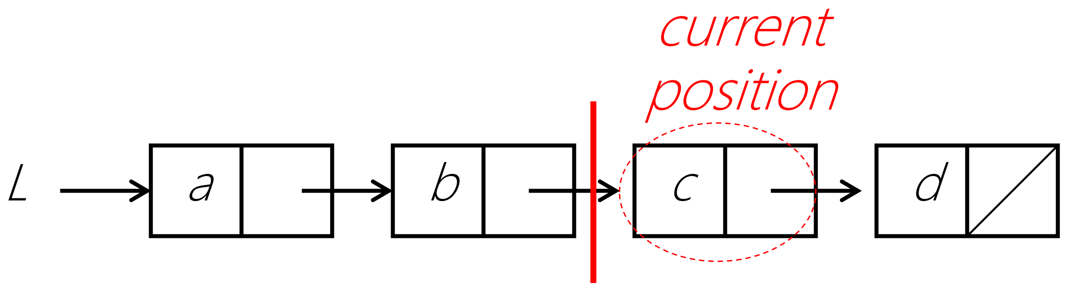
Pointer-Based List Implementation (1)

- Singly-linked list (One-way list)

Of course, we also have doubly-linked list ☺

$L = <a, b, | c, d>$

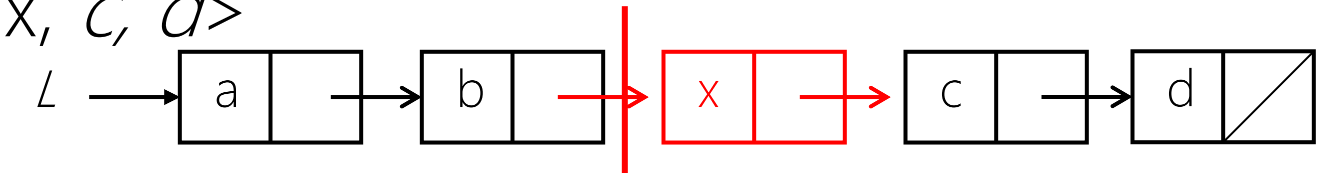
```
Struct item{
   char info;
   item* link;
};
```



  - Node (or cell) = info + link

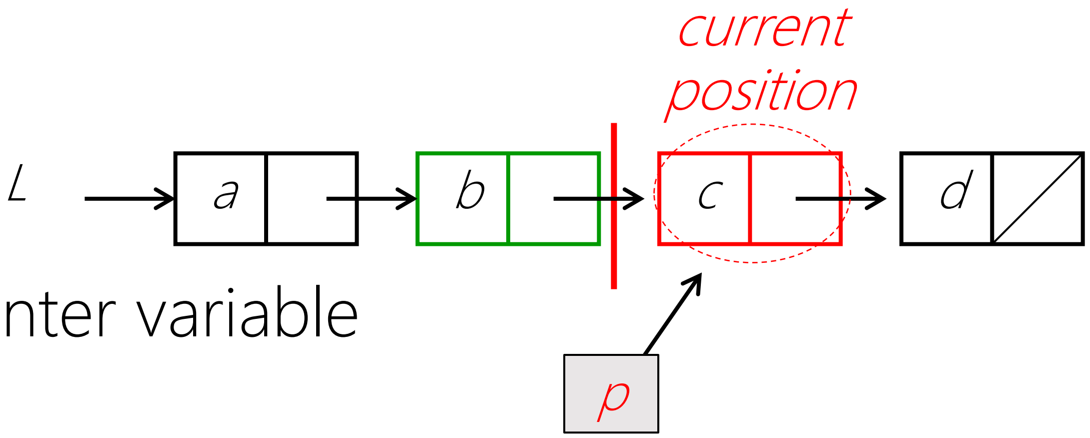
- *Insert (x, p, L) when p = current*

$L = <a, b, | x, c, d>$



Pointer-Based List Implementation (2)

- How to represent the logical fence?

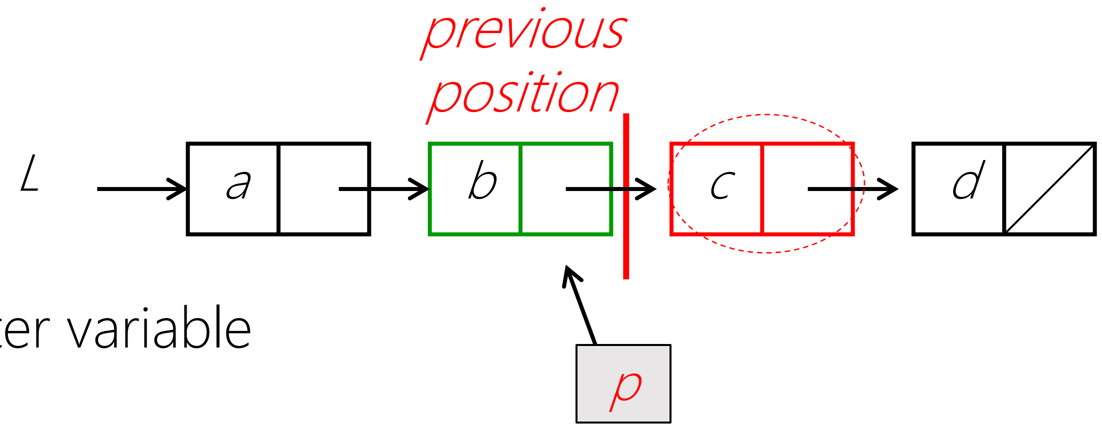


  - We need a pointer variable

- Option-1:
  - *P* directly points to the current element
  - What difficulty for *Insert (x, p, L)* ?
    - Inconvenient access to the preceding node of the current one
    - We have to change the link of the preceding node

Pointer-Based List Implementation (3)

- How to represent the logical fence?

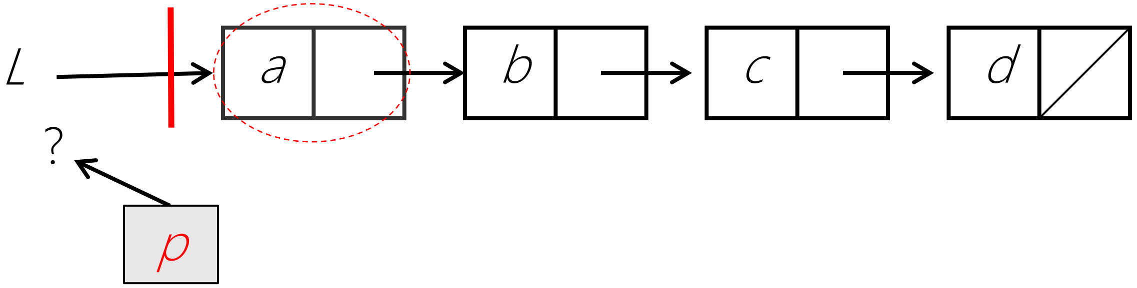


- We need a pointer variable

- Option-2 (*One-step ahead* convention):
  - *P* points to the previous element of the current position
  - Different definition of position
    - *P* is the position of the element "*b*" (not "*c*")

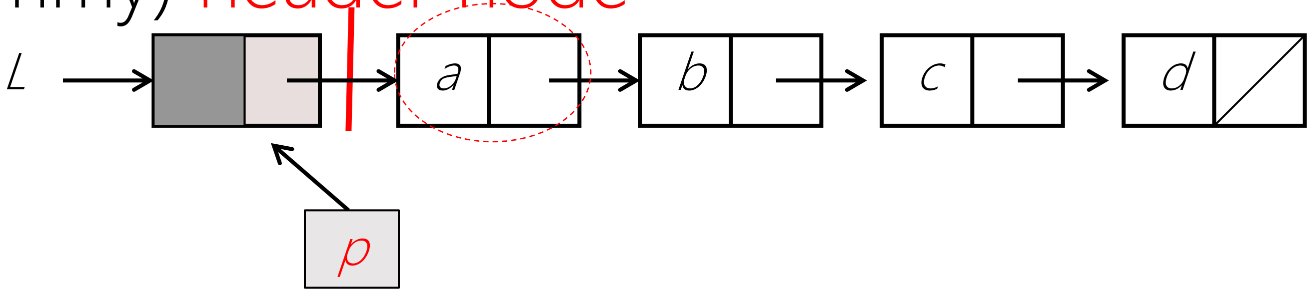
Pointer-Based List Implementation (4)

- *Insert (x, p, L)*
  - When the list is empty or the left partition is empty, What difficulty ?



    - Need to implement a number of special cases

- Solution: (dummy) header node



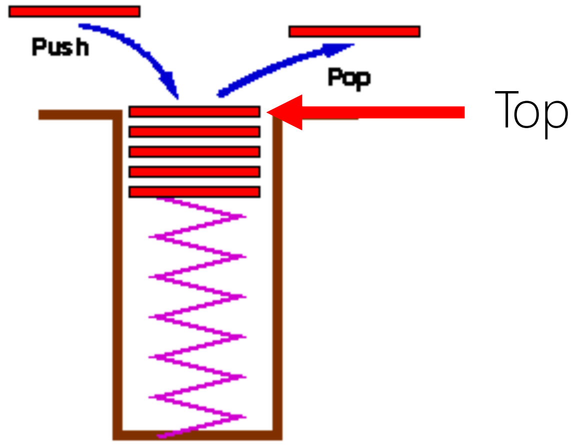
Cursor-Based List Implementation

- Cursor (Simulated Pointer)
  - Integer index indicating positions in array to simulate pointers
  - We maintain the heads of the lists
  - Zero value of the link means the tail of the list
- *Example*
  - $L_1 = \langle a, b, c \rangle$ $\quad$ $L_2 = \langle x, y \rangle$
- Time complexity of insert and delete
  - O(1)
- Time complexity of retrieve
  - O(n)

		info	link
	1	c	0
Head of L_1 →	2	a	4
	3	y	0
	4	b	1
	5		
Head of L_2 →	6	x	3
	7	...	

Stacks

- All insertions & deletions take place at one end (called *Top*)
- Special type of list
  - LIFO (Last-In, First-Out)
    - Pushdown list
- You can implement stacks using any type of list implementation (pointer, array, cursor, ...)



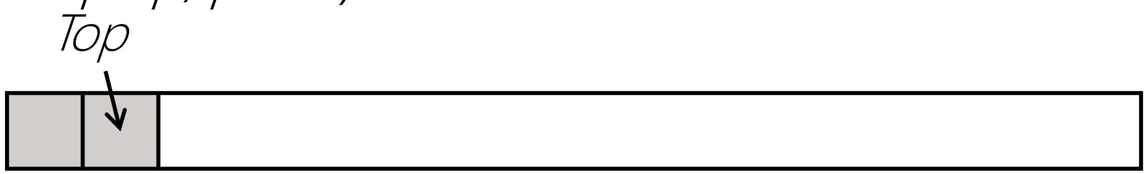
- $S = <a_1, a_2, \ldots, a_n>$

  $\uparrow$ *Top*

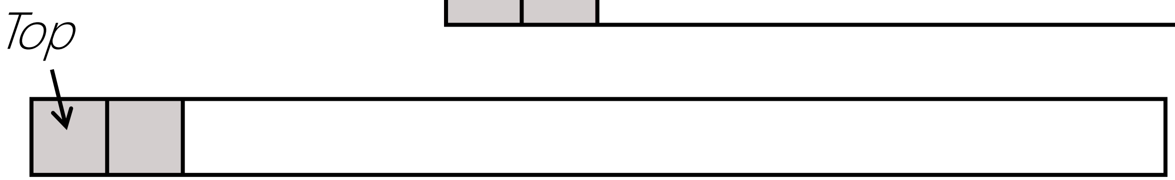
  - *Push(x, S)* — $S = <x, a_1, a_2, \ldots, a_n>$
  - *Pop(S)* — $S = <a_2, \ldots, a_n>$
  - *Top(S)* — a_1
  - *MakeNull(S)* — $S = <>$
  - *IsEmpty(S)* — true if $S = <>$

Stack Implementations

- Array-based stacks
  - *How to implement TOP*? (in terms of cost of *pop/push*)
    - Position *k* (when *k* elements in stack): O(1)



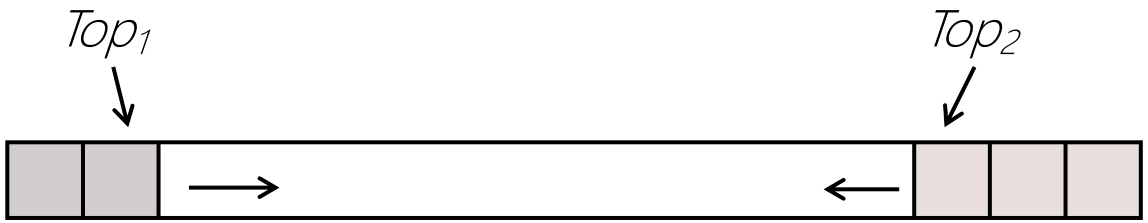
    - (Fixed) Position 1: O(n)



  - We can even store multiple stacks in a single array

- Linked stacks
  - Very much similar to the pointer-based list implementation

- Example: call stack



Queues

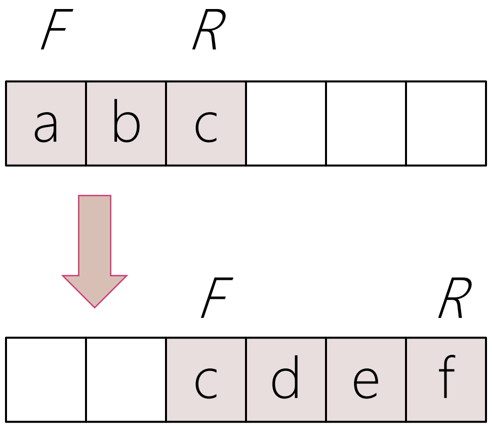
- FIFO (First-In First-Out) list
- Similar to top in the stack, here we have front and rear
- $Q = \langle a_1, a_2, \ldots, a_n \rangle$

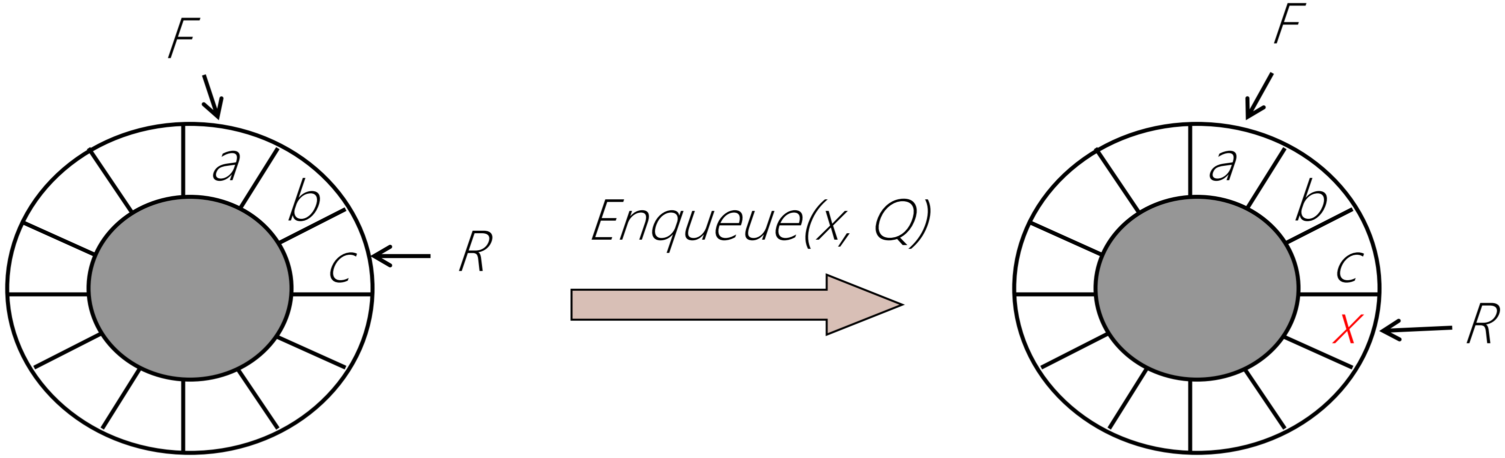
$$\underset{\textit{Front}}{\underset{\uparrow}{a_1}}, a_2, \ldots, \underset{\textit{Rear}}{\underset{\uparrow}{a_n}}$$

• *Enqueue(x, Q)*	$Q = \langle a_1, a_2, \ldots, a_n, x \rangle$
• *Dequeue(Q)*	$Q = \langle a_2, a_3, \ldots, a_n \rangle$
• *Front(Q)*	a_1
• *MakeNull(Q)*	$Q = \langle \rangle$
• *IsEmpty(Q)*	true if $Q = \langle \rangle$

Array-Based Queues (1)

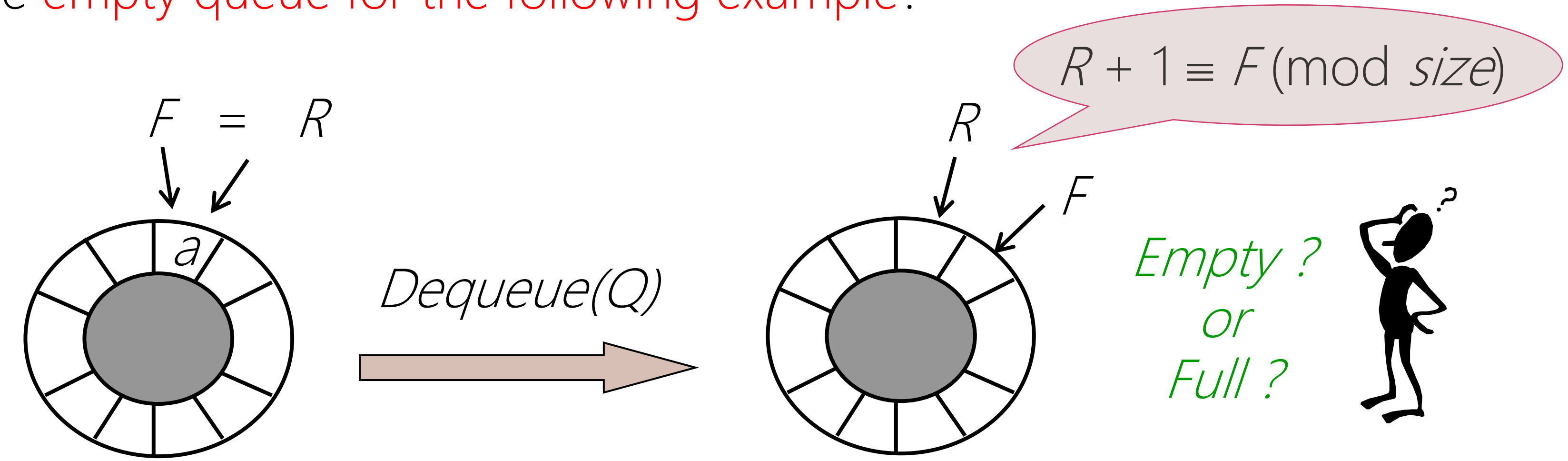
- Simple array implementation
  - "*drifting queue*" problem
  - We can solve this in an Inefficient way…
    - *Enqueue & Dequeue*: O(1) & O(n) or vice versa
- Circular array
  - Modulus/modulo operator: $n + 1 \equiv 1 \pmod{n}$
  - Mathematically connect the last element to the first element via modulo operator





Array-Based Queues (2)

- For the circular array, we have a problem…
  - How to recognize empty queue for the following example?

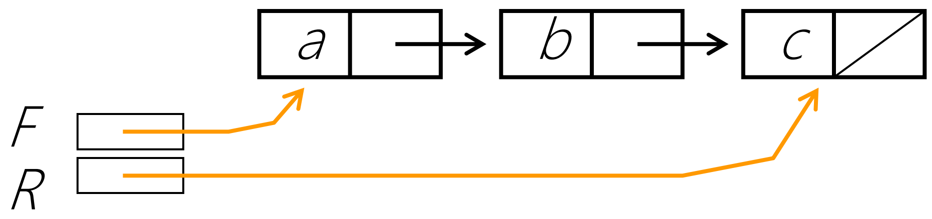


  - Problem - indistinguishable from full queue

- Several solutions
  - Explicit count variable (# of elements in queue)
    - For the enqueue: cnt ++;
    - For the dequeue: cnt --;
  - Boolean variable (to indicating empty queue)
    - bool isempty;

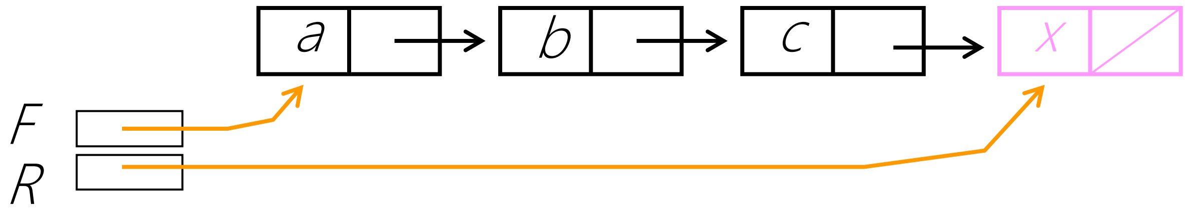
Linked Queues (1)

- Two pointers (*F*, *R*) (without a dummy header)



- *Empty(Q)*
  - true if ($F = R \rightarrow$ null)

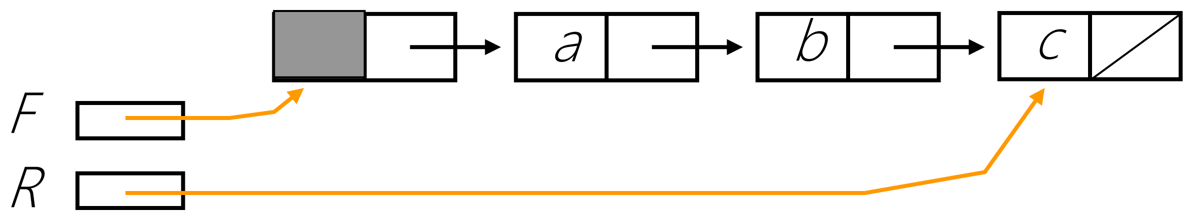
- *Enqueue(x, Q)*



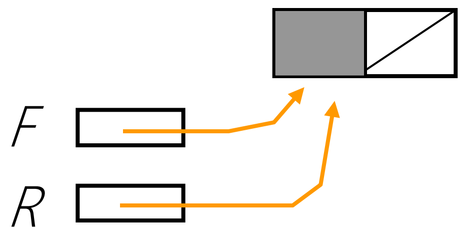
  - Problem: Can we use the same code when inserting into an empty queue?

Linked Queues (2)

- With a dummy header



- *Empty(Q)*
  - true if ($F = R \rightarrow$ header)



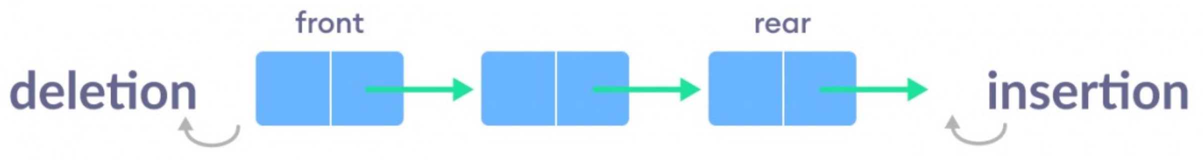
- *Enqueue(x, Q)*

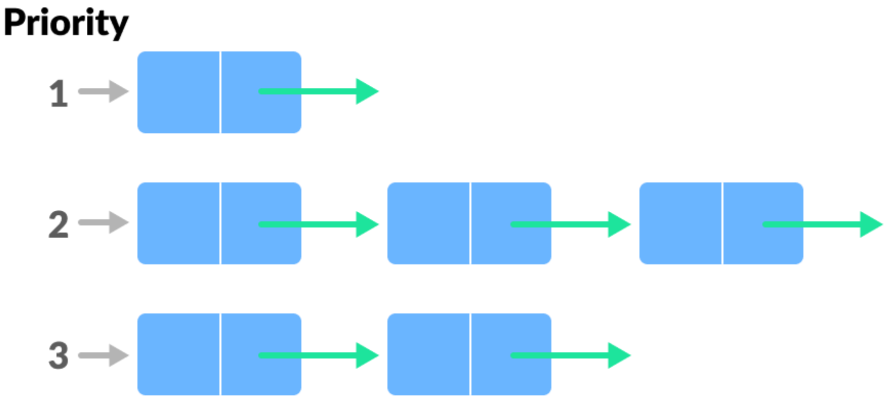
Linked Queues (3)

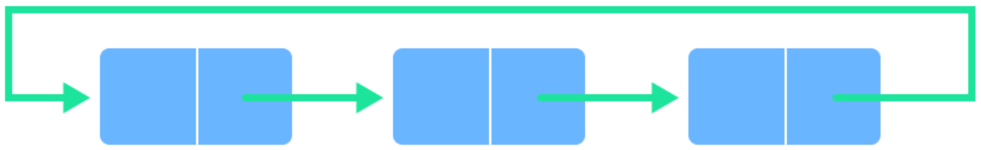
- Comparison (with & without header)
    - Speed
    - Space utilization
    - Code conciseness - the same codes for
        - Insertion into an empty queue
        - Deletion when queue has only one element

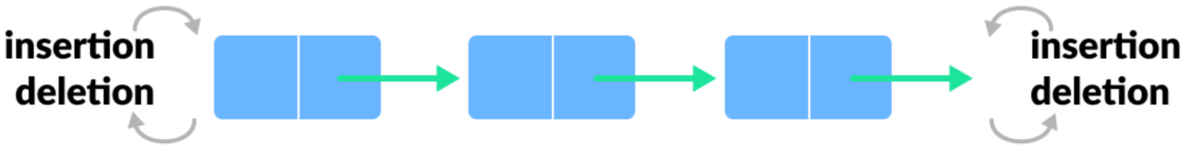
Type of Queues

- Queue, circular queue, priority queue, double-ended queue









References

- Further reading list and references
  - https://www.geeksforgeeks.org/binary-tree-data-structure/?ref=shm

- Slide credit
  - Jaesik Park
  - Seung-Hwan Baek
  - Jong-Hyeok Lee