Chapter 3
Lists, Stacks and Queues

South China University of Technology

College of Software Engineering

Huang Min

- 数据结构
  - 逻辑结构
    - 线性结构（线性表、栈 、队、串、数组）
    - 非线性结构
      - 树结构
      - 图结构
  - 物理（存储）结构
    - 顺序结构
    - 链式结构
    - 索引结构
    - 散列结构
  - 数据运算
    - 插入运算
    - 删除运算
    - 修改运算
    - 查找运算
    - 排序运算

逻辑结构唯一

存储结构不唯一

运算的实现依赖于存储结构

List

What is List

- A list is a finite, ordered sequence of data items called elements.
  - Notation: $<a_0, a_1, \ldots, a_{n-1}>$
  - Each element has a position in the list.
  - Each element may be of arbitrary type, but all are of the same type
  - The length of a list is the number of elements currently stored
    - An empty list contains no elements
  - The beginning and the end of the list are, respectively, called the head and the tail
  - Common List operations are:
    - insert, append, delete/remove, find, isEmpty, prev, next, currPos, moveToPos, moveToStart, length, etc

List Implementation

- Two standard list implementations
  - Array-based lists
  - Pointer-based lists (Linked lists)

List: Array Implementation

- Basic Idea:
  - › Pre-allocate a big array of size MAX_SIZE
  - › Keep track of current size using a variable **`count`**
  - › Shift elements when you have to insert or remove

0	1	2	3	…	count-1		MAX_SIZE
A_1	A_2	A_3	A_4	…	A_N		

List: Array Implementation

- Insert

insert Z in kth position

0	1	2	3	4	5			MAX_SIZE
A	B	C	D	E	F			

0	1	2	3	4	5	6		MAX_SIZE
A	B	Z	C	D	E	F		

Running time for N elements?

On average, must move half the elements to make room – assuming insertions at positions are equally likely

Worst case is insert at position 0. Must move all N items one position before the insert

This is O(N) running time.

$\Theta(1)$ for best case

List: Array Implementation

- remove the element at position curr
  - Shift left n-i-1 elements toward the head

Time cost – $\Theta(1)$ for best case; $\Theta(n)$ for worst- and average cases

- Other operations

bool moveToPos(int pos)
void moveToStart()
void moveToEnd()
void prev()
void next()
int Length() const
int currPos() const

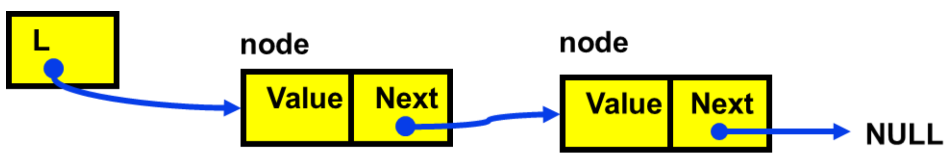
Time cost – $\Theta(1)$ for best, worst- and average cases

- Search for a value K in the list

Time cost – $\Theta(n)$ for worst- and average cases

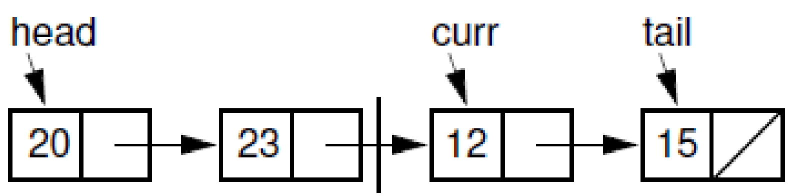
List: Pointer Implementation

- Linked list
  - › Use dynamic memory allocation which allocates memory for new list elements as needed
  - › Elements are called nodes, which are linked using pointers.
    - Keep track of list by linking the nodes together
  - › Change links when you want to insert or delete



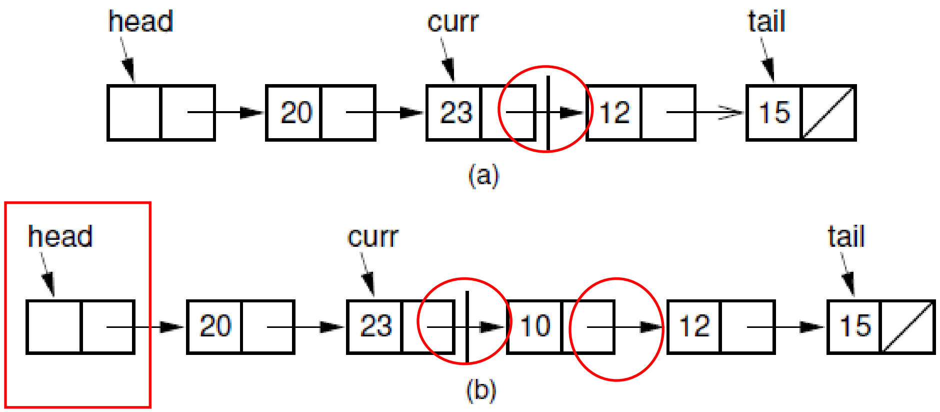
List: Pointer Implementation

- A linked list with 4 elements
  - Head pointer – for scanning the whole list
  - Tail pointer – to speed up “append” operation
  - Curr pointer – pointing to the current element
  - Value cnt – store the length of the list

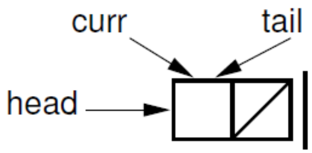


List:
Pointer Implementation

- Insertion
  - › With *curr* points to the node preceding the current position

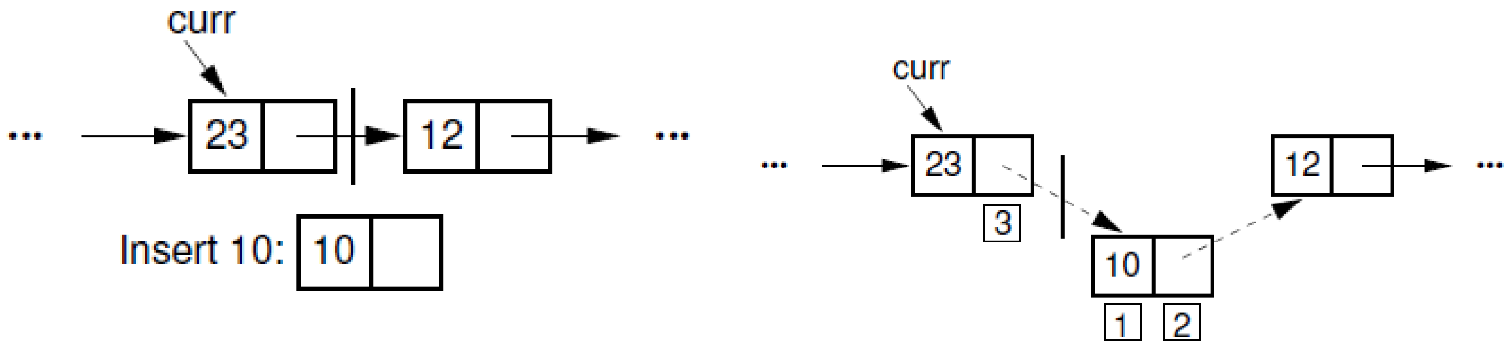


(a) (b)



List:
Pointer Implementation

- Linked List – Insertion
  - › Three-step insertion process
    - Create a new list node, store the new element
    - Set the next field of the new node
    - set the next field of the node pointed by <u>*curr*</u>



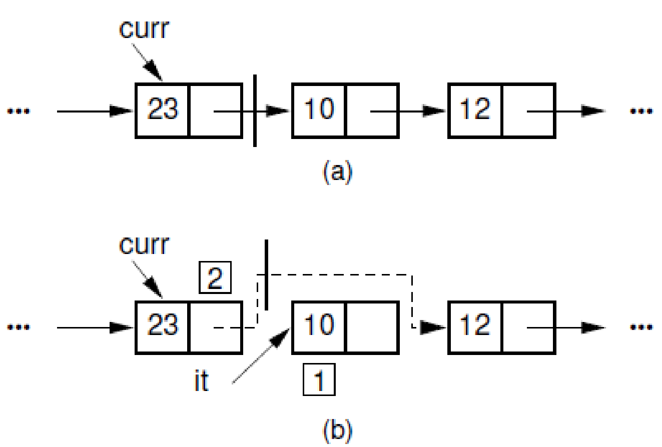
List: Pointer Implementation

```
//Insert a node to current position
  public:
    void insert(const E& it) {
        curr->next = new Link<E>(it, curr->next);
        if (tail == curr) tail = curr->next; //new tail
        cnt++;
    }
  //Append a node at the tail of the list
    void append(const E& it) {
        tail = tail->next = new Link<E>(it, NULL);
        cnt++;
    }
```

Time cost – $\Theta(1)$

List: Pointer Implementation

- Linked List – Removal
  - › Removing a node only requires to redirect some pointers around the node to be deleted.
  - › Remember to reclaim the space occupied by the deleted node



Time cost – $\Theta(1)$

List:
Pointer Implementation

- Linked List – Position Ops

```
//Next – move curr one pos toward the tail
   void next() {  }
//Prev – move curr one pos toward the head
   void prev() {  }
```

Time cost: $\Theta(1)$ for next;

$\Theta(n)$ for prev in the average and worst cases.

Comparison of List Implementations

Array-Based List		Linked List	
Predetermine the size before allocation.		Space is allocated on demand; No limit to the element number.	√
No waste space for an individual element.	√	Require to add an extra pointer to every list node.	
Random access and Prev takes $\Theta(1)$ time	√	Random access and Prev takes $\Theta(n)$ time	
Insertion and deletion takes $\Theta(n)$ time.		Insertion and deletion takes $\Theta(1)$ time.	√

Comparison of List Implementations

- linked lists are more space efficient when implementing lists whose number of elements varies widely or is unknown.

- Array-based lists are generally more space efficient when the user knows in advance approximately how large the list will become.

Comparison of List Implementations

- Comparison formula
  - The number of element currently in the list – n;
  - The size of a pointer – P
  - The size of a data element – E
  - The maximum number of elements in the array – D

- The array-based list requires space DE

- The linked list requires space n(P+E)

When n> DE/(P+E), the array-based list is more space efficient!

Exercise

- Determine the break-even point for a linked list being more efficient than an array-based list
  - The data field is 2 bytes, a pointer is 4 bytes, the array has 30 elements
    - $n < DE/(P+E) = 2*30/(2+4) = 10$
  - The data field is 8 bytes, a pointer is 4 bytes, the array has 30 elements
    - $n < DE/(P+E) = 8*30/(8+4) = 20$
  - The data field is 32 bytes, a pointer is 4 bytes, the array has 40 elements
    - $n < DE/(P+E) = 32*40/(32+4) = 35.555$

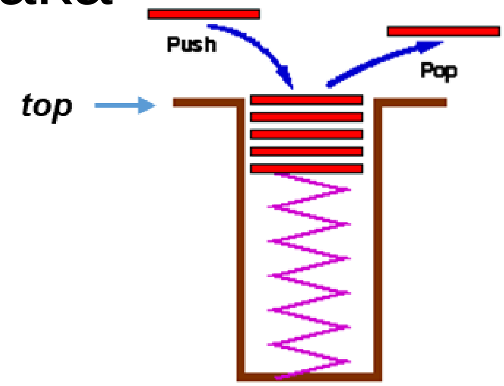
Stack

图片来自互联网

What is Stack

- A list for which Insert and Delete are allowed only at one end of the list (the top)
  - the implementation defines which end is the "top"
  - LIFO – Last in, First out

- Push: Insert element at top
- Pop: Remove and return top element (aka TopAndPop)
- IsEmpty: test for emptyness



a tray stack

Two Basic Implementations of Stacks

- Array-based
  - The k items in the stack are the first k items in the array.
- Linked List

Array-Based Stacks (II)

- Make the tail of the array be the top of the stack
  - Pushing an element onto the stack by appending it to the tail of the list
  - The cost for each **push** and **pop** operation is simply $\Theta(1)$.

- Setting of **top**
  - The array index of the first free position in the stack
    - An empty stack has top set to 0.

  - Push: first insert the element, then increment top
  - Pop: first decrement top, then removes the top element;
    - Pay attention to the order of two operations

Array-Based Stacks (III)

```
void clear() { top=0; }      //Reinitialize

void push(const E& it) {  // put "it" on stack
  Assert(top != maxSize, "Stack is full");
  listArray[top++] = it; }

E pop() {//pop top element
  Assert(top != 0, "Stack is empty");
  return listArray[--top]; }

const E& topValue() const {        //return top element
  Assert(top != 0, "Stack is empty");
  return listArray[top-1]; }
```

Comparison of Array-Based and Linked Stacks

	Array-Based Stack	**Linked Stack**
Implementation	Take the end of array as the top of stack	Take the head of linked list as the top of stack
Time cost	Constant time for push, pop, topValue; Constant time for clear	Constant time for push, pop and topValue; Linear time for clear
Space cost	Waste some space when the stack is not full - Overflow possible	Require the overhead of a link field for every element

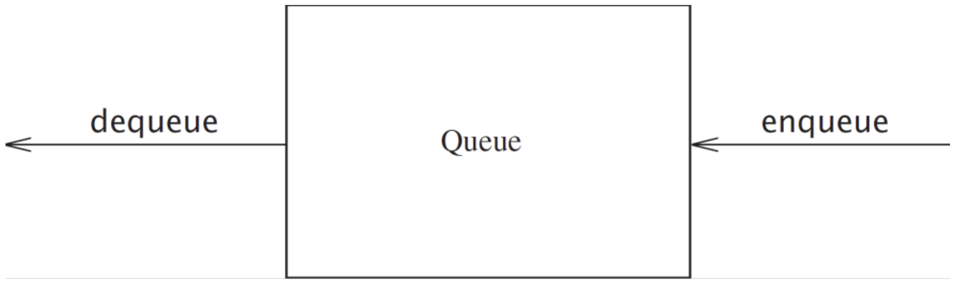
Q: How to implement two stacks using a single array?

Queue

图片来自互联网

What is Queue

- In a queue, elements may only be inserted from one end (back) of the list and removed from the other end (front) of the list
  - First-In, First-Out
  - Enqueue: insert an element at the back
  - Dequeue: remove an element from the front



Two Implementations of Queue

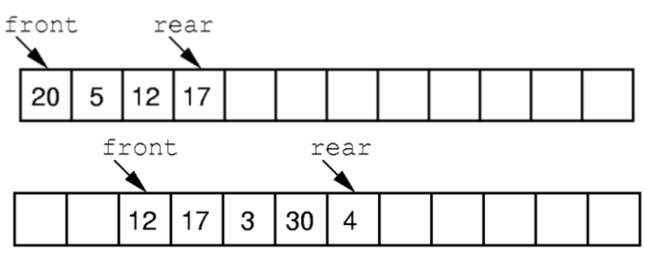
- The array-based queue
- The linked queue

Array-Based Queues

- An efficient and tricky implementation
  - The queue is still required to be stored in contiguous array positions
  - The queue position can **drift** within the array

Array-Based Queues

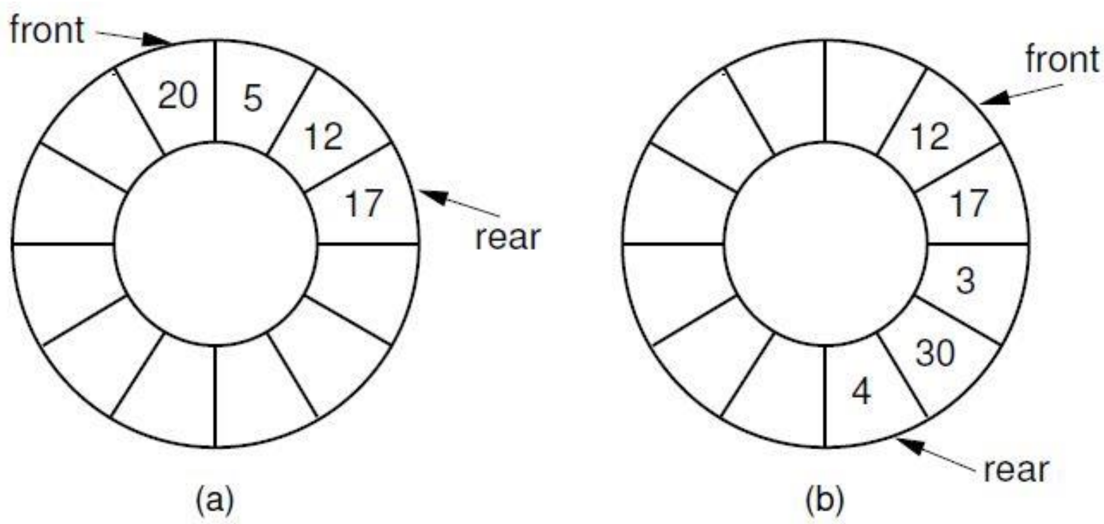
- Drifting queue
  - The **front** of the queue is initially at **position 0** of the array
  - The elements are added to successively higher-numbered positions
  - When elements are removed, the front index increases
  - **Both** enqueuer and dequeuer cost **Θ(1)** time



Problem?

Array-Based Queues

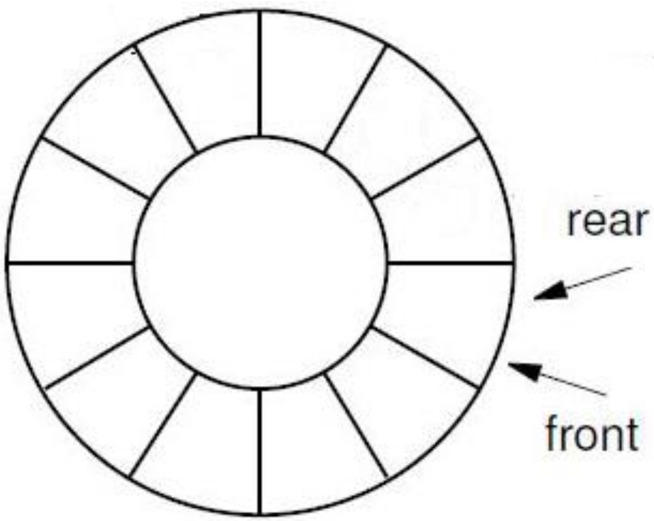
- Circular queue

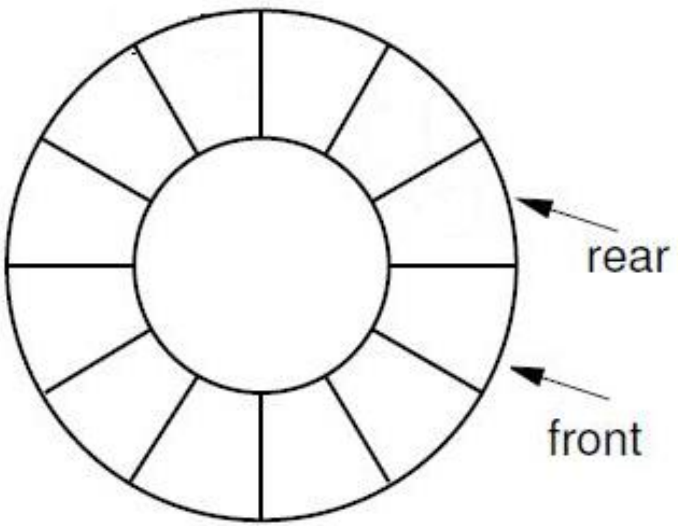


- Easily implemented using the modulus operator
  - Position maxSize-1 immediately precede position 0

Array-Based Queues

- Circular queue
  - How to recognize whether the queue is empty or full?





Array-Based Queues

- When **front = rear**, there has one element in the queue
- When **front** is one larger than **rear**, the queue is empty or full?
  - › Solution 1: explicitly keep a count of the number of elements in the queue
  - › Solution 2: make the array be of size n+1 and only allow n elements to be stored.
    - **front** = **rear**+1, the queue is empty
    - **front = rear**+2, the queue is full.

Linked Queues

- A straightforward adaptation of the linked list

- Structures
  - Use a header node
  - The **front** pointer points always points to the header node
  - The **rear** pointer points to the last link node in the queue

- Operations
  - **Enqueue**: places the new element in a link node at the end of the linked list, advances **rear** to point to the newly-inserted node
  - **Dequeue**: removes and returns the first element of the list

Comparison of Array-Based and Linked Queues

- Time cost
  - All member functions for both implementations require constant time **$\Theta(1)$**
- Space cost
  - For array-based queues, there are some space waste if the queue is not full.
  - For linked queues, there are overhead of link field in each element.