# 真·超浓缩DSA整合攻略（期末篇）

---

## Time complexity

`Comparison-based Sorting Algorithm`

| Algorithm | Time Complexity | memo |
|---|---|---|
| Stack push and pop | $\Theta(1)$ | |
| Queue enqueue and dequeue | $\Theta(1)$ | |
| LinkedList Search | $\Theta(n)$ | |
| LinkedList Insert | $\Theta(1)$ | |
| LinkedList Delete | $\Theta(1)$ | If we need to search for the key then $\Theta(n)$ |
| Hash table | worst case: $\Theta(n)$ | normally quicker, search can be $O(1)$ |
| direct-address table:insert,delete,search | $O(1)$ | drawback: memory, keys must be integers |
| hash table chain: insert&delete | $O(1)$ | |
| hash table chain search | $O(n)$ | worst case: every key hashes the same slot; average case: $\Theta(1+\alpha)$ [$\alpha = n/m$] |

## Elementary Data Structures

- Data structure is a systematic way of storing and organizing data in a computer so that it can be used efficiently.
- We use different data structures for different applications

**Abstract data type** `ADT`
: Specifies what are the operations and what are the errors/ exceptions

**Dynamic sets**
: The element removed from the set by the `DELETE` operation is prespecified.

---

# Stack

- A linear data structure with last-in-first-out (**LIFO**) behaviour : the element deleted from the set is the one most recently inserted.

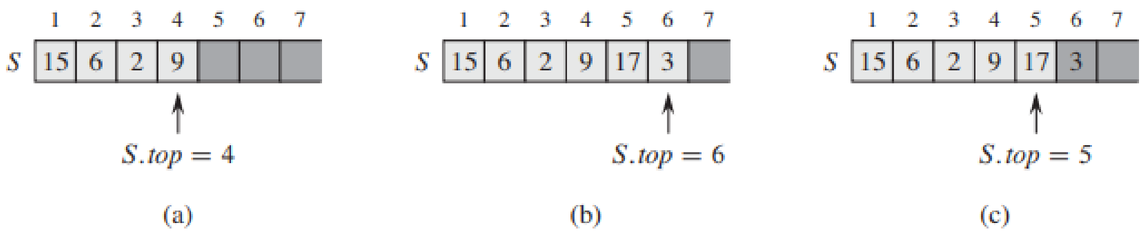


**Figure 10.1** An array implementation of a stack $S$. Stack elements appear only in the lightly shaded positions. **(a)** Stack $S$ has 4 elements. The top element is 9. **(b)** Stack $S$ after the calls PUSH($S$, 17) and PUSH($S$, 3). **(c)** Stack $S$ after the call POP($S$) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

- We can implement a stack of at most `n` elements with an array `S[1..n]`.
- Elements are added from left to right in the array.
- The array has an attribute `S.top` that indexes the most recently inserted element.
- The stack consistes of elements `S[1..S.top]`, where `S[1]` is the element at the bottom of the stack and `S[S.top]` is the element at the top.
- When `S.top = 0`, the stack contains no elements and is `empty`.
- If we attempt to `pop` an `empty` stack, we say the stack **underflows**
- If `S.top = S.length`,the stack is `full`. If `S.top` exceeds `n`, the stack **overflows**

## Stack Empty

```
STACK-EMPTY(S)
  if S.top == 0
    return TRUE
  else return FALSE
```

## PUSH

```
Algorithm push(S,x):
  if S.top = N then //N = S.length
    error FullStackException //overflow
  else
    S.top := S.top + 1
    S[S.top] := x
```

## POP

```
Algorithm pop(S):
  if isEmpty(S) then
    error EmptyStackException //underflow
  else
    x := S[S.top]
    S.top := S.top - 1
    return x
```

- Time complexity for push and pop: $\Theta(1)$

---

## Queue

- A linear data structure with first-in-first-out(**FIFO**) behaviour : the element deleted is always the one that has been in the set for the longest time. (add and remove at different sides)

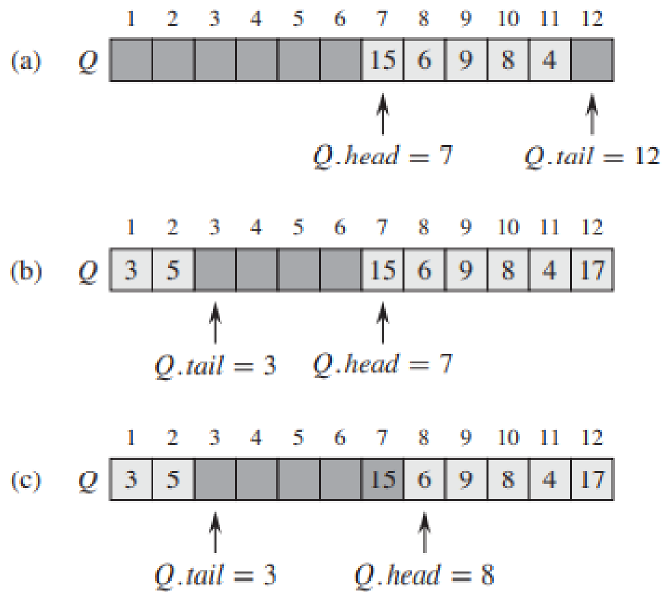


**Figure 10.2** A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE($Q$, 17), ENQUEUE($Q$, 3), and ENQUEUE($Q$, 5). **(c)** The configuration of the queue after the call DEQUEUE($Q$) returns the key value 15 formerly at the head of the queue. The new head has key 6.

- We can implement a queue of at most `n-1` elements using an array `Q[1..n]`.
- We call the `INSERT` operation on a queue `ENQUEUE` and we call the `DELETE` operation `DEQUEUE`.
- The element dequeued is always the one at the head of the queue.
- `Q.head` is index of oldest element (also front); initially `Q.head = 1`.
- `Q.tail` is first index **available** following index of youngest element; initially `Q.tail = 1`.
- The elements in the queue reside in locations `Q.head`, `Q.head+1`, ...,`Q.tail-1`, where we "wrap around" in the sense that location `1` immediately follows location `n` in a **circular order.**
- When `Q.head = Q.tail`, the queue is `empty`.
- When `Q.head = Q.tail + 1`/`Q.tail = Q.head - 1` or `Q.head = 1` and `Q.tail = Q.length`, the queue is `full`.
- If we attempt to `dequeue` an element from an empty queue, the queue **underflows**.
- If we attempt to `enqueue` an element when the queue is full, the queue **overflows**

## ENQUEUE

```
//Size of the array fixed
//Initially Q.head = Q.tail = 1
//Elements are at indices Q.head, Q.head + 1, ..., Q.tail - 1
//Check on overflow omitted

Algorithm enqueue(Q,x):
  Q[Q.tail] := x
  if Q.tail = Q.length then
    Q.tail := 1 //wrap around | |a|b|c|<-Q.tail = Q.length   =>   Q.tail->|
|a|b|c|
  else
    Q.tail : = Q.tail + 1
```

## ENQUEUE(Adapt Version)

```
//Check overflow

Algorithm enqueue(Q,x):
  Q[Q.tail] := x
  if (Q.head = Q.tail + 1) or (Q.head = 1 and Q.tail = Q.length) then
    error FullQueueException
  if Q.tail = Q.length then
    Q.tail := 1
  else
    Q.tail : = Q.tail + 1
```

## DEQUEUE

```
//Size of the array  N = Q.length fixed
//check on empty queue omitted

Algorithm dequeue(Q):
  x := Q[Q.head]
  if Q.head = Q.length then    //|b|c| |a|<-Q.head = Q.length   =>   Q.head-
>|b|c| | |
    Q.head := 1
  else
    Q.head := Q.head + 1      // Q.head ->|a|b|c| |  => | |b|c| |   Q.head ->
|b|
```

## DEQUEUE(Adapt Version)

```
//Check underflow

Algorithm dequeue(Q):
  if (Q.head ==Q.tail) then
    error EmptyQueueException
  x := Q[Q.head]
  if Q.head = Q.length then
    Q.head := 1
  else
    Q.head := Q.head + 1
```

## size(Q)

```
//queue implemented as circular array
//n = Q.length to be the size of the array

Algorithm size(Q):
    if Q.head = Q.tail then
        return 0
    if Q.head < Q.tail then
        return Q.tail - Q.head
    if Q.tail < Q.head then
        return n - Q.head + Q.tail  // n - (Q.head + Q.tail)
```

## isEmpty

```
//queue implemented as circular array
//n = Q.length to be the size of the array

Algorithm isEmpty(Q):
    return Q.head = Q.tail
```

## head(Q)

```
//queue implemented as circular array
//n = Q.length to be the size of the array

Algorithm head(Q):
    return Q[Q.head]
```

- Time complexity for ENQUEUE and DEQUEUE is $\Theta(1)$
- 用一个`stack`无法实现`queue`，同样，用一个`queue`无法实现`stack`。如果要用`stack`来implement `queue`，需要至少两个`stack`(详见Exercise6，Q6)，而两个`queue`也可以implement `stack`(详见Exercise6，Q7)。
- 一个`singly linked list`可以implement`stack`，也可以implement`queue`。（详见Exercise6，Q10&Q11）

---

# Linked List

- Linked list is a linear data structure for a dynamic set in which the object are arranged in a linear order.
- The order in a linked list is determined by a pointer in each object.
- We have access to elements and access to an element gives us access to the next element.
- In case of doubly linked list, we also have access to the previous element.
- The list has an attribute giving us access to the head.

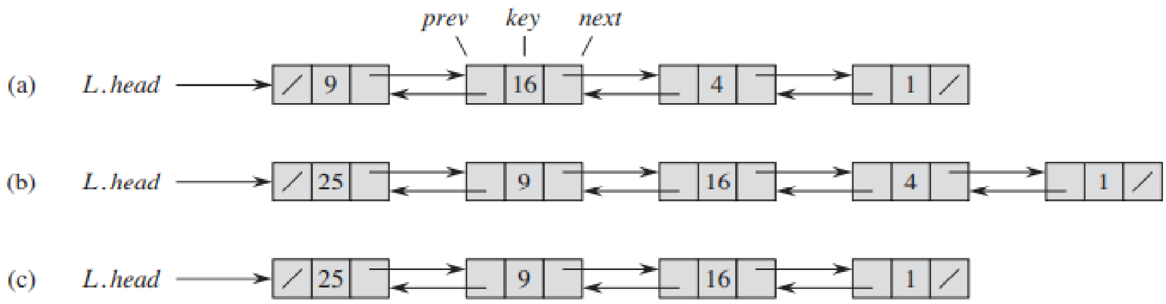


**Figure 10.3** **(a)** A doubly linked list $L$ representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-INSERT$(L, x)$, where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE$(L, x)$, where $x$ points to the object with key 4.

- We have elements usually written `x`, with `x.key` which gives the key, the elements may also contain other satellite data.
- If `x.next = NIL`, the element `x` has no successor and is therefore the last element, or `tail` of the list.
- For a **doubly linked list,** if `x.prev = NIL`, the element `x` has no predecessor and is therefore the first element, or `head` of the list.
- A list `L` contains zero , one or more elements. We have `L.head` pointing to the first node. If list is empty, then `L.head = NIL`
- If **a list is singly linked**, we omit the `prev` pointer in each element.
- If **a list is sorted**, the linear order of the list corresponds to the linear order of keys stored in elements of the list; the minimum element is then the `head` of the list, and the maximum element is the `tail`.
- If the **list is unsorted**, the elements can appear in any order.
- In **a circular list**, the `prev` pointer of the `head` of the list points to the `tail`, and the `next` pointer of the `tail` of the list points to the `head`.

## SEARCH

- Worst case: listSearch is in $\Theta(n)$ with $n$ the number of elements in the list

```
//input: a list and a key
//output: pointer to the first element containing the key or nil

Algorithm listSearch(L,k):
    x := L.head
    while x != nil and x.key != k do
```

```
        x := x.next
    return x // if x = nil, then x.key doesn't exist.

/*
x -> -> -> ->....-> -> nil
|
L.head
*/
```

### INSERT

- Worst case: insert takes $O(1)$ so constant time

```
//input: list, an element with a key;
//update: node added in front

Algorithm listInsert(L,x):
    x.next := L.head
    if L.head != nil then
        L.head.prev := x
    L.head := x
    x.prev := nil
```

### DELETE

- Worst case: delation is in $O(1)$ so constant time
- If we wish to delete an element with a given key, we must first call listSearch to retrieve a pointer to the element: $\Theta(n)$

```
//input: a list and an element x to be deleted
//x前面一位的 element.next 跳过x 指向x后面一位
//x后面一位的 element.prev 跳过x 指向x前面一位

Algorithm listDelete(L,x):
    if x.prev != nil then    //check if x is first element(x.prev == nil), if
so, update L.head
        x.prev.next := x.next
    else
        L.head := x.next
    if x.next != nil then   //check if x is the last element
        x.next.prev := x.prev
```

### Sentinels

- A sentinel is a dummy element (so, no `key`) that simplifies dealing with boundaries.

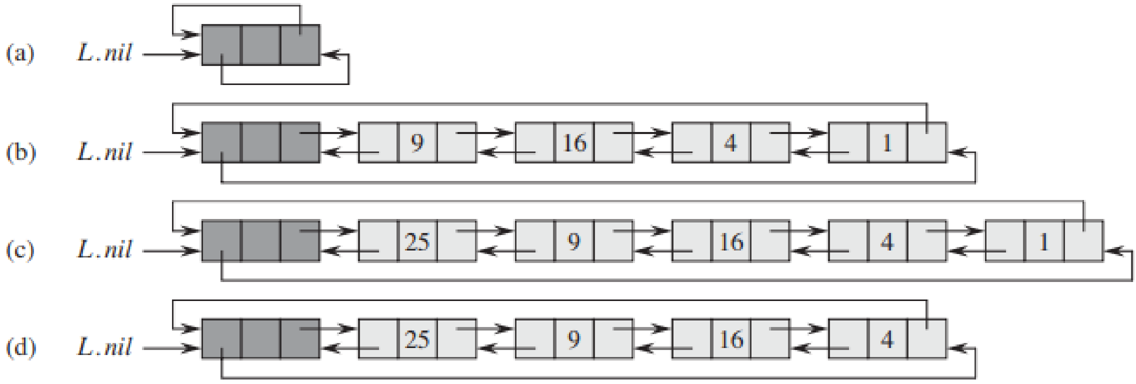


**Figure 10.4** A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

- `L.nil` represents `nil` and is as the start.
- `L.nil.next` points to the head of the list.
- `L.nil.prev` points to the tail of the list (**doubly linked**)
- next element of last element and previous of first point to `L.nil`
- empty list is just the sentinal `L.nil`
- `L.head` no longer necessary

### SEARCH (with sentinels)

```
Algorithm listSearchSen(L,k):
    x := L.nil.next
    while x != L.nil and x.key != k do
        x := x.next
    return x // if x = L.nil, then x.key doesn't exist.
```

### INSEART (with sentinels)

```
Algorithm listInsertSen(L,x):
    x.next := L.nil.next //without sentinels, x.next := L.head
    L.nil.next.prev := x // L.head.prev = x
    L.nil.next := x
    x.prev := L.nil
```

### DELETE (with sentinels)

```
Algorithm listDeleteSen(L,x):
    x.prev.next := x.next
    x.next.prev := x.prev
```

### Intermezz: dancing links

- consider a node `x` in a doubly linked list
- we remove `x` as follows:

```
(x.prev).next := x.next
(x.next).prev := x.prev
```

- if we do not do garbage collection, we can put `x` back in as follows:

```
(x.prev).next := x
(x.next).prev := x
```

`see the paper Dancing Links by Knuth`

---

# Trees

## Recap definitions of binary tree

**binary tree**
: every node has at most 2 successors(empty tree is also a binary tree)

**depth of a node** `x`
: length(number of edges) of a path from the root to `x`

**height of a node** `x`
: length of a maximal path from `x` to a leaf

**height of a tree**
: height of its root
number of levels is `height + 1`

## Binary tree: linked implementation

linked data structure with nodes containing

- `x.key` from a totally **ordered set**
- `x.left` points to left child of node `x`
- `x.right` points to right child of node `x`
- `x.p` points to parent of node `x`
- if `x.p = nil` then `x` is the root
- `T.root` points to the root of the tree (`nil` if empty tree)

Alternative implementation of to linked list is **heap**, where binary trees can be represented as arrays using the level numbering.