ch5 stack 堆疊

1、Last in first out (LIFO)

2、operation contract for the ADT stack

isEmpty() , push, pop, getTop

3、迴文 (abc$cba)

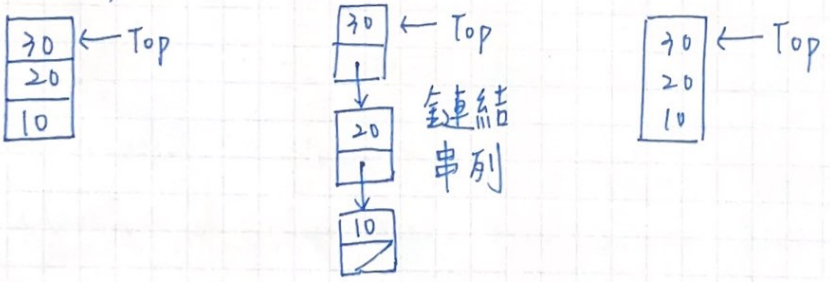
solution using a stack :

Traverse the first half of the string, pushing each character onto a stack.

Once you reach the $, for each character in the second half of the string, match a popped character off the stack.

4、實作 stack ADT

① An array 陣列 ② Linked list ③ ADT List



ch6 Queue 佇列

1、First in first out (FIFO)

2、ADT queue operations

① create a empty queue 建構

② destroy a queue 解構

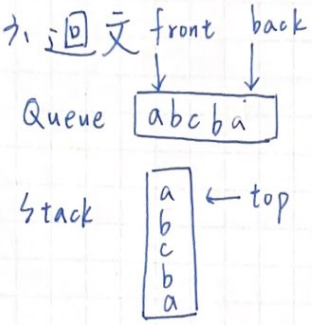
③ isempty

④ enqueue 新增

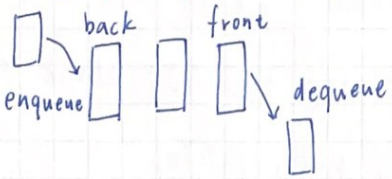
⑤ dequeue 移除

⑥ getfront 取第一個

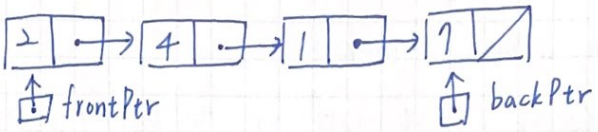
3. 迴文



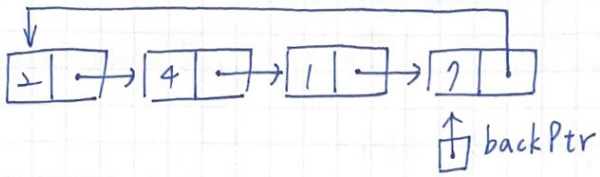
4. 以指標實作佇列



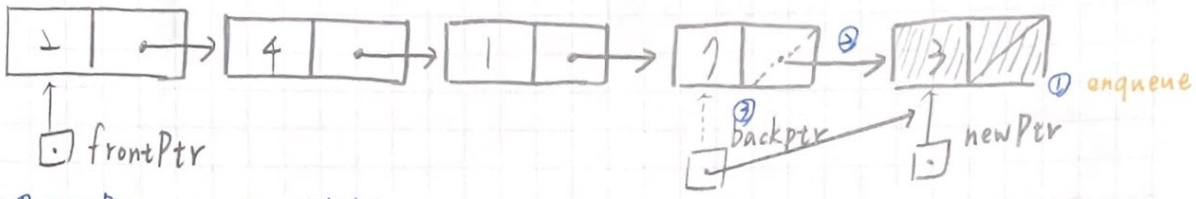
• 一般鏈結串列



• 環狀鏈結串列




```
class Queue {
  public : Queue();
           Queue (const Queue &Q);
          ~Queue ();
           bool isempty() const;
           void enqueue (const QueueItemType & newItem);
           void dequeue ();
           void dequeue (const QueueItemType & queueFront);
           void getFront (const QueueItemType & queueFront);
  private: struct QueueNode{
               QueueItemType item;
               QueueNode *next;
           }
           QueueNode *backPtr;
           QueueNode *frontPtr;
} // Queue()
```

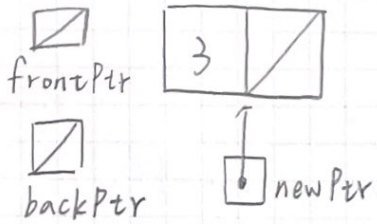


① newPtr → next = NULL;

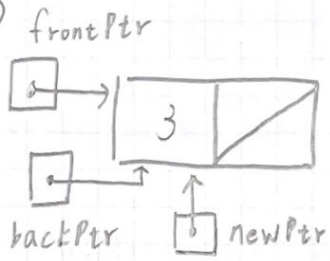
② backPtr → next = newPtr;

③ backPtr = newPtr;

(a)



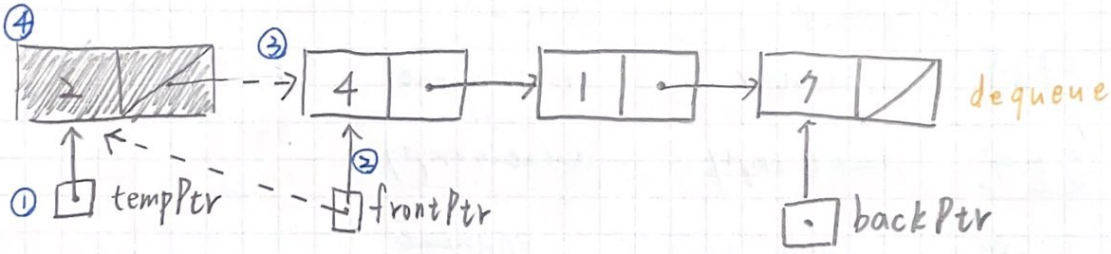
(b)



更新後端指標

frontPtr = newPtr;

backPtr = newPtr;



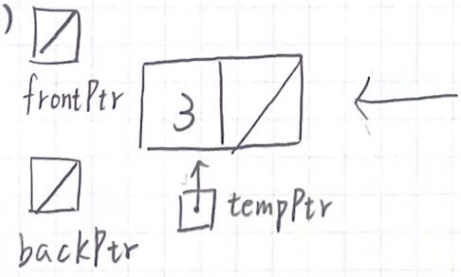
① tempPtr = frontPtr;

② frontPtr = frontPtr → next;

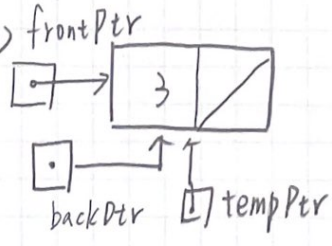
③ tempPtr → next = NULL;

④ delete tempPtr;

(a)



(b)



① tempPtr = frontPtr;

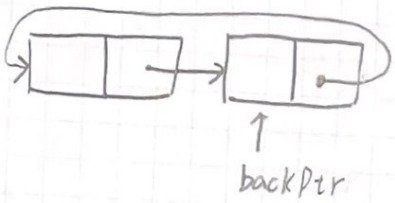
② frontPtr = NULL;

③ backPtr = NULL;

④ tempPtr → next = NULL;

⑤ delete tempPtr;

Circular Queue 環狀佇列



A summary of Position-Oriented ADT (位置導向)

Position-oriented

→ List — all position can be accessed

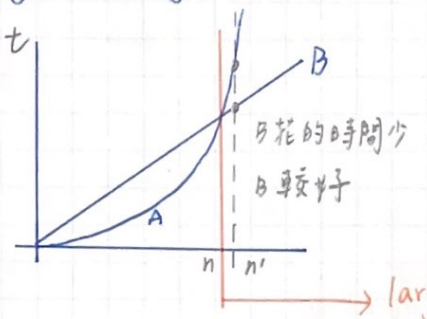
→ Stack
→ Queue
} only the end positions can be accessed

	Stack	Queue
建構	createStack	create Queue
是否空	Stack is empty	Queue is empty
新增	push	enqueue
移除	pop	dequeue
擷取	getTop	get Front

Ch 7 演算法效率

Algorithm efficient is typically a concern of large problems only

Algorithm growth-Rate



A: $\frac{n^2}{5}$ 指數成長

B: 5n 線性成長

Big O notation 成長函數 (計算時會忽略快的，只在意慢的)

- O(1) 常數
- $O(\log_2 n)$ 對數
- O(n) 線性
- $O(n\log_2 n)$
- $O(n^2)$ 平方
- $O(n^3)$ 立方
- $O(2^n)$ 指數

- Sequential search 循序搜尋

  Efficient → ① worst O(n)

  ② Average O(n)

  ③ best O(1)

- Binary search of sorted array

  Efficient → worst $O(\log_2 n)$

排序演算法效率

stable	unstable
bubble	selection
insertion	quick
merge	heap
radix	

bubble sort

```
void bubbleSort (int A[], int n) {          第幾回合
    for (int pass = 1; pass < n; pass++) {
        for (int index = 0; index < n-pass; index++) {      第幾筆資料
            if (A[index] > A[index+1])
                swap (A[index], A[index+1]);
        } // for()
    } // for()
} // bubbleSort()
```

worst case $O(n^2)$

best case $O(n)$

Selection Sort

```
void selectionSort(int A[], int n) {
    for (int last = n-1; last > 0; last--) {        n-1 回合
        int largest = indexofLargest(A, last+1);    找出最大值位置
        swap (A[largest], A[last]);                 把最大值移到末端
    } // for()
} // selectionSort()
```

best case $O(n^2)$

worst case $O(n^2)$

```
Void InsertSort ( int A[], int n) {
    for( int unsort = 1 ;  unsort < n ; unsort ++) {
        int loc = unsort ;
        nextItem = A[unsort];
        for (  ; (loc>0) && (A[loc-1] > nextItem); loc--)
            A[loc] = A[loc-1] ;
        A[loc]= nextItem ;
    } // for
} // insertion sort ( )
```

worst case $n(n-1)/2 \rightarrow O(n^2)$

best case $O(n)$

Shell sort 希爾排序 not stable

- Merge Sort 合併　先分組 → 各自排序 → 合併

worst case $O(n\log_2 n)$ { 優：fast
Average case $O(n\log_2 n)$ { 缺：need second array

- Quick sort　先找 pivot → 依 pivot 分組 → 遞回呼叫

Average case $O(n\log_2 n)$

worst case $O(n^2)$

not stable

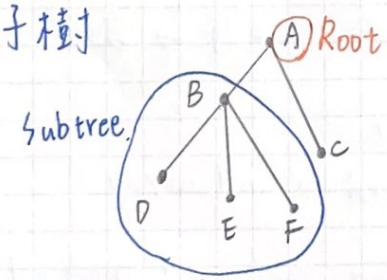
- Radix Sort (基數排序) 分解取部分值 → 分配置對應容器

Sort	worst	best
Selection	n^2	n^2
Bubble	n^2	n^2
Insertion	n^2	n^2
Merge	$n \log n$	$n \log n$
Quick	n^2	$n \log n$
Radix	n	n
heap	$n \log n$	$n \log n$

ch8 Tree

· Parent-child 親子 | Ancestor-decendant 祖孫

Subtree 子樹



A is parent of node B、C

B is parent of node DEF

B is child of node A

· Left → no children 葉節點

· Siblings → Node of common parent

· Ancestor of B ⇒ root to B

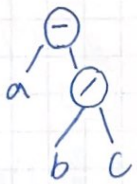
· Decendant of B ⇒ B to a left

Binary Tree

a-b

height = 2

a-b/c



height = 3

(a-b)*c

height = 3

心得：看完老師的影片，真的學到很多，寫作業時也比較理解該怎麼做比較好，比較不會花很多時間在想要怎麼辦。