Queue

The Abstract Data Type Queue

☐Operation Contract for the ADT *Queue*

isEmpty():boolean {query} 是否為空

enqueue(in newItem:QueueItemType) 新增
    throw QueueException

dequeue() throw QueueException 移除

getFront(out queueFront:QueueItemType) 擷取
    {query} throw QueueException

dequeue(out queueFront:QueueItemType)
    throw QueueException 擷取後移除

☐ADT *queue* operations

- Create an empty queue 建構
- Destroy a queue 解構
- Determine whether a queue is empty 是否為空
- Add a new item to the queue 新增
- Remove the item that was added earliest 移除
- Retrieve the item that was added earliest 擷取

Recognizing Palindrome 回文

☐A palindrome (e.g.,, dad, noon, civic, level, …) 迴文

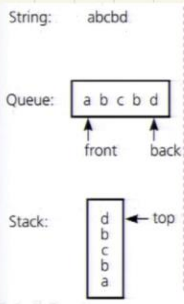
- A string of characters that reads *the same* from left to right as its does from right to left

☐To recognize a palindrome, you can use a queue in conjunction with a stack 堆疊

- A stack reverses the order of occurrences 顛倒次序
- A queue preserves the order of occurrences 佇列 保持次序

利用堆疊和次序相反的性質

比較 front of queue & top of stack



isPal(in str: string): boolean
aQueue.createQueue()
aStack.createStack()
for (the next character ch in str)
{ *store ch into aQueue & aStack*
} // end for
while (aQueue is not empty)
{ *compare front & top*
} // end while

辨識迴文


```
isPal(in str: string): boolean
aQueue.createQueue()
aStack.createStack()
for (the next character ch in str)
{ aQueue.enqueue(ch)
  aStack.push(ch)
} // end for
charEqual = TRUE
```

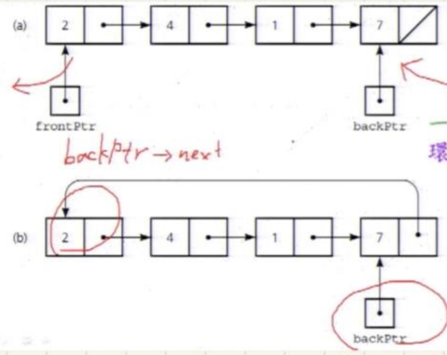
```
while (!aQueue.isEmpty()
  && charEqual)
{ aQueue.dequeue(front)
  aStack.pop(top)
  if (front != top)
     charEqual = FALSE
} // end while
```

辨識迴文

Implementation of the ADT Queue

- **An *array*-based implementation (later)**
- **Possible implementations of a *pointer*-based queue**
  - A *linear* linked list with two external references
    - **A reference to the front** 前端
    - **A reference to the back** 後端
  - A *circular* linked list with one external reference
    - **Only a reference to the back** 環狀：只有後端

P. 17



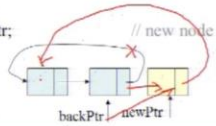
一般鏈結串列
環狀鏈結串列

P. 18

1. A Pointer-based Implementation

環狀佇列 新增

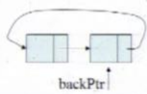
```
void Queue::enqueue(const QueueItemType& newItem)
{  QueueNode *newPtr = new QueueNode;
   newPtr->item = newItem;
   if (isEmpty())                          // 0 → 1 node
       newPtr->next = newPtr;              // point to itself
   else                                    // k → k+1 nodes, k > 0
   {   newPtr->next = backPtr->next;       // point to the front
       backPtr->next = newPtr;             // put behind the back
   }   // end else
   backPtr = newPtr;                       // new node at the back
}  // end enqueue
```



P. 28

環狀佇列 移除

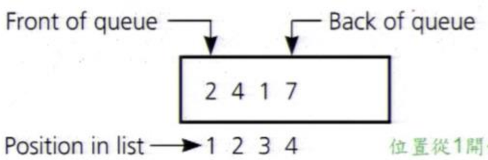
```
void Queue::dequeue() throw(QueueException)
{  if (isEmpty())
      throw ...;
   else
   {   QueueNode *tempPtr = backPtr->next;      // the front
       if (backPtr == backPtr->next)
           backPtr = NULL;                      // one node → empty
       else   backPtr->next = tempPtr->next;    // the next front
       tempPtr->next = NULL;                    // defensive strategy
       delete tempPtr;                          // release space
   }   // end else
}  // end dequeue
```



P. 30

2. An Implementation That Uses the ADT List

The front of the queue is at position 1 of the list;
The back of the queue is at the end of the list



位置從1開始

P. 42

- **enqueue()** 新增
  ```
  aList.insert(aList.getLength()+1,
               newItem)
  ```
- **dequeue()** 移除
  ```
  aList.remove(1)
  ```
- **getFront(queueFront)** 擷取
  ```
  aList.retrieve(1, queueFront)
  ```

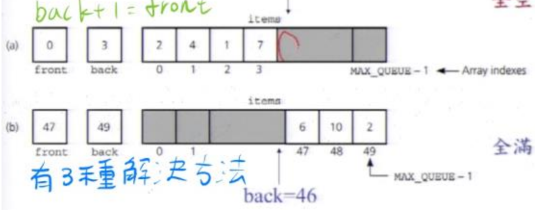
P. 43

3. An Array-based Implementation

全滿和全空
條件相同
back+1 = front

front = 4

環狀陣列
back+1 = front?



有3種解決方法

back=46

P. 32

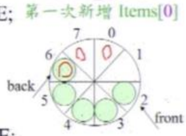
☐Inserting into a queue 1. 設 count

back初始值 MAX_QUEUE-1
第一次新增 Items[0]

back = (back+1) % MAX_QUEUE;
items[back] = newItem;
++count;

☐Deleting from a queue

front = (front+1) % MAX_QUEUE;
--count;



front初始值0
第一次移除
front 0→1

全空的條件？ count == 0
全滿的條件？ count == MAX_QUEUE

P. 36

2. 多宣告一個空間

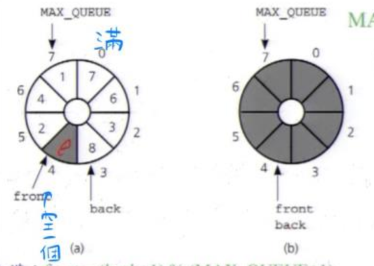
☐ Variations of the array-based implementation

2. **Declare MAX_QUEUE + 1 locations for the array items, but use only MAX_QUEUE of them for queue items**
3. **Use a flag *isFull* to distinguish between the full and empty conditions**

設定旗標：是否全滿

P. 37

MAX_QUEUE = 7



全滿：front = (back+1) % (MAX_QUEUE+1)

P. 38

```
Queue::Queue():back(MAX_QUEUE-1), front(0), isFull(FALSE)
{ }      // end default constructor

bool Queue::isEmpty() const
{   return (!isFull) && (front == (back+1) % MAX_QUEUE);
}   // end isEmpty

void Queue::enqueue(const QueueItemType& newItem) throw(QueueException)
{  if (isFull == TRUE)
        throw ...;
   else                                    // queue is not full
   {    back = (back+1) % MAX_QUEUE;
        items[back] = newItem;
        if (front == (back+1) % MAX_QUEUE)
             isFull = TRUE;                // queue is full now
   }    // end else
}  // end enqueue
```

3. isFull Flag

P. 40

```
void Queue::dequeue() throw(QueueException)
{  if (isEmpty())                          // queue is empty
        throw ...;
   else                                    // queue is not empty
   {    front = (front+1) % MAX_QUEUE;
        if (isFull == TRUE)                // queue is full
             isFull = FALSE;               // queue is not full
   now
   }    // end else
}  // end dequeue
```

P. 41

Comparing Implementations

☐Fixed size versus dynamic size 固定大小

- A *statically* allocated array-based implementation
  - **Fixed-size queue that can get full**
  - **Prevents the enqueue operation from adding an item to the queue, if the array is full**
- A *dynamically* allocated array-based implementation or a *pointer*-based implementation 動態配置
  - **No size restriction on the queue**

P. 44

☐A *pointer*-based implementation vs. one that uses a pointer-based implementation of the ADT *list*

- Pointer-based implementation is more efficient
- ADT *list* approach reuses an already implemented class 程式執行效率
  - **Much simpler to write**
  - **Saves programming time** 程式撰寫效率

P. 45

Application: Simulation

比較常用

An event-driven simulation 事件驅動
- Simulated time advances to time of next event
- Events are generated by using a mathematical model based on statistics and probability

Events (input file)

Arrival	duration	Departure	waiting
20	5	25	0
22	4	29	3
23	2	?	?
30	3	?	?

P. 53

A time-driven simulation 時間驅動
- Simulated time advances by one time unit
- The duration of each event is randomly determined and compared with the simulated time

time

20 21 22 23 24 25

~~D 25~~ ~~A 22~~ D 29 A 23

Events (input file)

Arrival	duration	Departure	waiting
20	5	25	0
22	4	29	3
23	2	?	?
30	3	?	?

P. 52

Bank simulation is event-driven and uses an event list 事件清單
- Keeps track of arrival and departure events that will occur but have not occurred yet
- Contains at most one arrival event and one departure event

P. 55

事件驅動

```
Simulate()
  Create an empty bankQueue;      // represent the bank line
  Create an empty eventList;      // keep the future events
  Get the earliest arrival event X from input file;
  Put X into eventList;
  while (eventList is not empty)
  {   newEvent = the earliest event in eventList;
      if (newEvent is an arrival event)
            processArrival();
      else  processDeparture();
  }   // end while
```

P. 56

Time	Action	bankQueue (front to back)	anEventList (beginning to end)
0	Read file, place event in anEventList	(empty)	A 20 5
20	Update anEventList and bankQueue: Customer 1 enters bank	20 5	(empty)
	Customer 1 begins transaction, create departure event	20 5	D 25
	Read file, place event in anEventList	20 5	A 22 4, D 25
22	Update anEventList and bankQueue: Customer 2 enters bank	20 5, 22 4	D 25
	Read file, place event in anEventList	20 5, 22 4	A 23 2, D 25
23	Update anEventList and bankQueue: Customer 3 enters bank	20 5, 22 4, 23 2	D 25
	Read file, place event in anEventList	20 5, 22 4, 23 2	D 25, A 30 3
25	Update anEventList and bankQueue: Customer 1 departs	22 4, 23 2	A 30 3
	Customer 2 begins transaction, create departure event	22 4, 23 2	D 29, A 30 3

按先後排隊

按時間排序

20 5
22 4
23 2
30 3

wait 3 (=25-22)

P. 57

Simulation by Queue

☐Use the following event list to simulate a single bank queue and calculate the *average waiting time*.

Events (input file)

Arrival	transaction	Departure	waiting
5	9	14	0
7	5	19 (14+5)	7 (14-7)
14	5	24	5
30	5	35	0
32	5	40	3
34	5	45	6
38	3		

單一佇列

P. 59

Multi-queue Simulation

☐Use the following event list to simulate two bank queues with bankQueue 1 first selection strategy and calculate the *average waiting time*.

Events (input file)

Arrival	transaction	Departure	waiting
5	9	14	0
7	5	___	___
14	5	___	___
30	5	___	___
32	5	___	___
34	5	___	___
38	3	___	___

P. 61

☐Use the following event list to simulate two bank queues with bankQueue 1 first selection strategy and calculate the *average waiting time*.

Events (input file)

AWT = 1/7

兩個佇列

Arrival	transaction
5	9
7	5
14	5
30	5
32	5
34	5
38	3

anEventList: D 40 | D 41

bankQueue 1: 34 5

bankQueue 2: 38 3

P. 62

Measuring the Efficency of Algorithms

☐How many time units does the nested loop take?

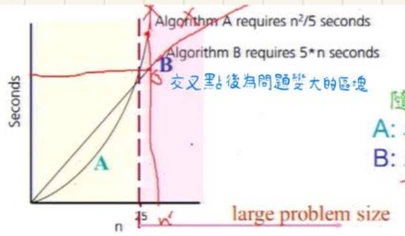
– n = 10
– n = 100

a=1, b=1, 5×1
a=2, b=1,2, 5×2
3, 1,2,3, 5×3

```
for (a = 1; a <= n; a++)
  for (b = 1; b <= a; b++)
    for (c = 1; c <= 5; c++)
      cout << a << b << c << endl;    //t ms
```

n

Σ(t*5*a) for a = 1 to n ➔ 5 * t * n * (n+1) / 2
➔ t * (2.5n² + 2.5n) ➔ n = 10: 275t, n = 100: 25250t



隨著n增加
A: 指數成長
B: 線性成長

Algorithm A requires time proportional to n^2
Algorithm B requires time proportional to n

☐Algorithm A is order $f(n)$ – denoted O ($f(n)$)

if constants k and n_0 exist such that A requires no more than $k * f(n)$ time units to solve a problem of size $n \geq n_0$

迴 遞 大小位階

☐Examples

Practice 7-1 $2.5n^2 - 2.5*n$ is O(?), k=? n_0=?

$\forall n \geq n_0$, $(2.5n^2 - 2.5*n) \leq k * f(n)$

$\forall n \geq 10$, $(2.5n^2 - 2.5*n) \leq 1 * n^{10}$ D – O (n^{10})

$\forall n \geq n_0$, $(2.5n^2 - 2.5*n) \leq k * n^2$ A

$\forall n \geq 0$, $(2.5n^2 - 2.5*n) \leq 3 * n^2$ O(n^2)

☐Algorithm A is order $f(n)$ – denoted O ($f(n)$)

if constants k and n_0 exist such that A requires no more than $k * f(n)$ time units to solve a problem of size $n \geq n_0$

☐Examples 鏈結串列 假設c, a, w皆為常數

Example 1. $(n+1)*(c+a) + n*w$ is O(?), k=? n_0=?

$\forall n \geq n_0$, $(n+1)*(c+a) + n*w \leq k * f(n)$ O(n)

$\forall n \geq 1$, $(n+1) \leq 2n$ ➔

$(n+1)*(c+a) + n*w \leq n * (2*(c+a) + w) \leq k * n$ A+

☐Worst-case analysis 最多

1 2 3
3 2 1

– A determination of the maximum amount of time that an algorithm requires to solve problems of size n

☐Average-case analysis 平均

2 1 3

– A determination of the average amount of time that an algorithm requires to solve problems of size n

☐Best-case analysis 最少

– A determination of the minimum amount of time that an algorithm requires to solve problems of size n

☐Sequential search 循序搜尋

– Strategy
  - ■Look at each item in the data collection in turn
  - ■Stop when the desired item is *found*, or the *end* of the data is reached
– Efficiency
  - ■Worst case: O(n)
  - ■Average case: O(n)
  - ■Best case: O(1)

2
8
1
9
5
7

Efficiency of Sorting Algorithms

□Binary search of a sorted array 二元搜尋

- Strategy
  - ■Repeatedly divide the array in half
  - ■Determine which half could contain the item, and discard the other half
- Efficiency
  - ■Worst case: $O(\log_2 n)$

1 2 5 7 8 9

$\lceil 6/2 \rceil = 3$ $\quad 2^{k-1} < n < 2^k$

$\lceil 3/2 \rceil = 2$ $\quad 2^2 < 6 < 2^3$

$\lceil 2/2 \rceil = 1$ $\quad 2 < \log_2 6 < 3$

$n = 2^k$

e.g., $16 = 2^4$

$\log_2 16 = 4$

P. 28

□Binary search of a sorted array

- Efficiency
  - ■Worst case: $O(\log_2 n)$
  - ■For large arrays, the binary search has an enormous advantage over a sequential search
    - At most 20 comparisons to search one million items
    - $\log_2 10^6 = 19.9$

一百萬筆資料只需要做二十次比較！

P.

□ Consider a sequential searching of *n* data items

2. What is the order of the sequential search algorithm when the desired item is not in the data collection?
   - **Sorted vs. unsorted**
   - **Worse vs. average vs. best**

0 → 1 3 5 7 9

不同狀況下的位階？

	sorted	unsorted	*found*
Worse case	O(n)	O(n)	O(n)
Average case	O(n)	O(n)	O(n)
Best case	O(1)	O(n)	O(1)

P. 31

Categories of sorting algorithms

□Categories of sorting algorithms

- An internal sort 在記憶體內部做排序 内部排序
  - ■Requires that the collection of data fit entirely in the computer's main memory
- An external sort 在記憶體外做排序 外部排序
  因內容太大！下學期會學到
  - ■The collection of data will not fit in the computer's main memory all at once, but must reside in secondary storage 速度慢

P. 36

Stable Sort

Stable Sort vs. Unstable sort

Stable Sort	Unstable sort
1. bubble	1. quick
2. insertion	2. heap
3. merge	
4. radix	

相同值能夠/不能維持不變的排序

Ex.

8 28 14 5 8 26 2 6 29 5

⇓

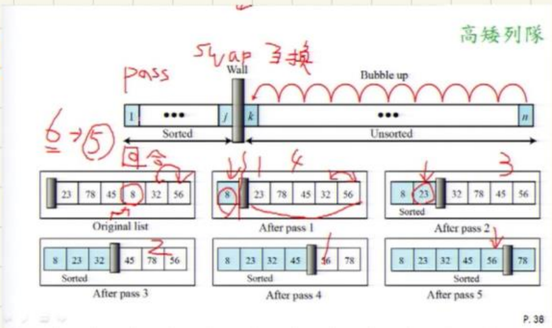
2 5 5 6 8 8 14 26 28 29

Sort Algorithms comparisons

Bubble sort

bast $O(n^2)$
or $O(n)$

worst $O(n^2)$

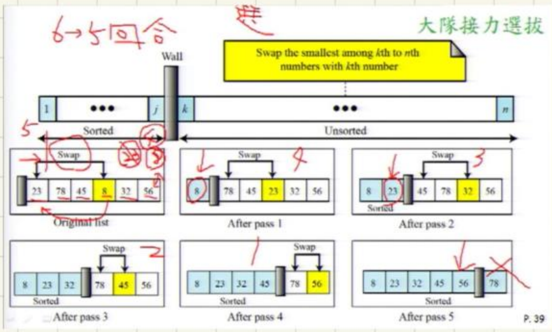


swap 次數多

selection sort

bast $O(n^2)$
or $O(n)$

worst $O(n^2)$



swap 次數少

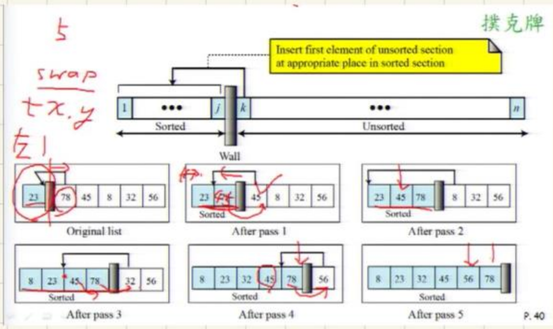
最差的 sort

只適合應用在
單筆資料太大的情況

insertion sort

bast $O(n)$

worst $O(n^2)$



無 swap
僅將元素
往右移後
插入

- Sort the following data by using each algorithm:

8_a 28 14 5_a 8_b 26 2 6 29 5_b

1. bubble sort
2. selection sort unstable !
3. insertion sort

stable

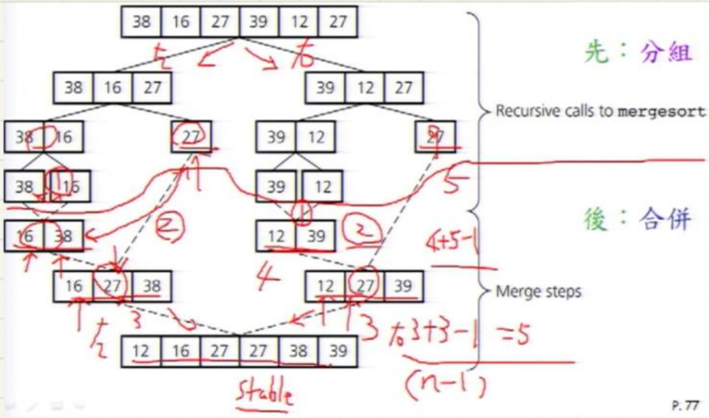
BS:	2	5_a	5_b	6	8_a	8_b	14	26	28	29
SS:	2	5_a	5_b	6	8_b	8_a	14	26	28	29
IS:	2	5_a	5_b	6	8_a	8_b	14	26	28	29

shellSort

```
void shellSort(int A[], int n)
{  for (int h = n/2; h > 0; h = h/2)
   for (int unsorted = h; unsorted < n; ++unsorted)
   {   int loc = unsorted;
       int nextItem = A[unsorted];
       for (; (loc >= h)&&(A[loc-h] > nextItem); loc=loc-h)
              A[loc] = A[loc-h];
       A[loc] = nextItem;
   }   // end for
}  // end shellSort
```

Merge Sort

先分組，再排序，後合併




```
void mergeSort(DataType theArray[], int first, int last)
{  if (first < last)
   {  int mid = (first + last)/2;              // middle point
      mergeSort(theArray, first, mid);         // sort the left half
      mergeSort(theArray, mid+1, last);        // sort the right half
      merge(theArray, first, mid, last);       // merge the two halves
   }  // end if
}  // end mergeSort
```

先：分組（遞迴呼叫）
後：合併

```
void merge(DataType theArray[], int first, int mid, int last)
{  DataType tempArray[MAX_SIZE];            // temporary array
   int first1 = first, last1 = mid;         // the left half [first…mid]
   int first2 = mid+1, last2 = last;        // the right half [mid+1...last]
   int index = first;                       // next available location
   for (; (first1 <= last1) && (first2 <= last2); ++index)
      if (theArray[first1] < theArray[first2])
      {     tempArray[index] = theArray[first1];
            ++first1;
      } else
      {     tempArray[index] = theArray[first2];
            ++first2;
      }     // end if-else
```

10	33	60	
21	43	57	72

兩組中最小的優先

$3+4-1$

n

Worst case: **n-1** comparisons

P. 79

```
   …
   for (; first1 <= last1; ++first1, ++index)    // finish the left half
      tempArray[index] = theArray[first1];
   for (; first2 <= last2; ++first2, ++index)    // finish the right half
      tempArray[index] = theArray[first2];

   for (index = first; index <= last; ++index) // copy the result back
      theArray[index] = tempArray[index];
}  // end merge
```

寫回資料！

theArray <==> tempArray: **2n** moves
∴ (n-1) + 2n = **3 * n – 1** major operations ➔ **O(n)**

P. 80

Analysis
- Worst case: O(n * $\log_2 n$)
- Average case: O(n * $\log_2 n$)

Advantage
- Mergesort is an extremely fast algorithm

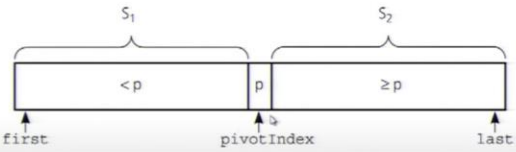
Disadvantage
- Mergesort requires a second array as large as the original array

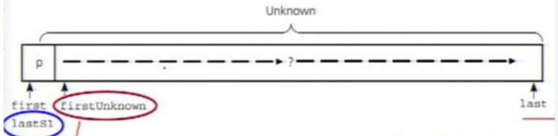
Quick Sort

- **Another** divide-and-conquer **algorithm**
- **Strategy** 找pivot分左右組後排序
  - Choose a pivot 樞紐、軸
  - *Partition* the array about the pivot 先：分組（軸的位置）
    - **items < pivot**
    - **items >= pivot**
    - **Pivot is now in correct sorted position**
  - *Sort* the left section 後：遞迴呼叫
  - *Sort* the right section

P. 86

- **If pivot is placed at the correct sorted position…**

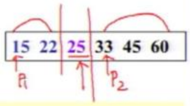





```
int lastS1 = first;                    S1的最後一個位置
int firstUnknown = first+1;    // index of the first item in Unknown
while (firstUnknown <= last)
{       if (A[firstUnknown] < p)
                move A[firstUnknown] into S1
        else    move A[firstUnknown] into S2
        ++firstUnknown;
}
```

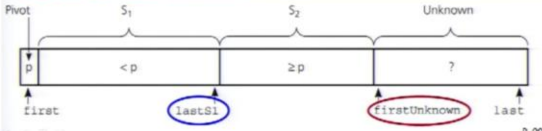
P. 89

```
while (firstUnknown <= last)
{       if (A[firstUnknown] < p)
                move A[firstUnknown] into S1
        else    move A[firstUnknown] into S2
        ++firstUnknown;
}
```




```
swap(A[first], A[lastS1]);
pivotIndex = lastS1;
```

```
++lastS1;
swap(A[lastS1], A[firstUnknown]);
```



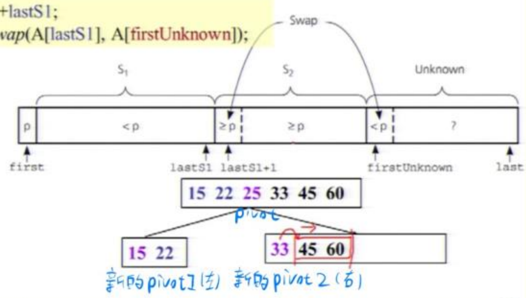
P. 90

```
void quickSort(DataType theArray[], int first, int last)
{  int pivotIndex;
   if (first < last)      // create the partition: S1, pivot, S2
   {  partition(theArray, first, last, pivotIndex);
      quickSort(theArray, first, pivotIndex-1);
      quickSort(theArray, pivotIndex+1, last);
   }  // end if
}  // end quickSort
```

先：依軸分組
後：遞迴呼叫

P. 92

```
++lastS1;
swap(A[lastS1], A[firstUnknown]);
```



P. 91

average O(n logn) unstable!

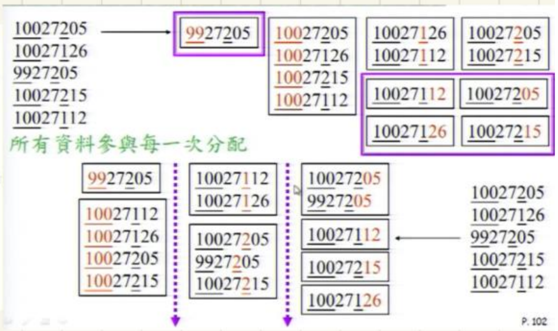
Worst O(n²)

Radix Sort

取一個索引作為分組的radix

左邊補零

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)



P. 102

☐LSD (*Least Significant Digit*) 依照最右側數字分組、串接

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150
- [0] (1560, 2150)
- [1] (1061)
- [2] (0222)
- [3] (0123, 0283)
- [4] (2154, 0004)

1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004
- [0] (0004)
- [2] (0222, 0123)
- [5] (2150, 2154)
- [6] (1560, 1061)
- [8] (0283)

P. 104

0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283
- [0] (0004, 1061)
- [1] (0123, 2150, 2154)
- [2] (0222, 0283)
- [5] (1560)

0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560
- [0] (0004, 0123, 0222, 0283)
- [1] (1061, 1560)
- [2] (2150, 2154)

0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154

P. 105

☐MSD (*Most Significant Digit*) 改從最左側數字開始？

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150
- [0] (0123, 0222, 0004, 0283)
- [1] (1560, 1061)
- [2] (2154, 2150)

0123, 0222, 0004, 0283, 1560, 1061, 2154, 2150
- [0] (0004, 1061)
- [1] (0123, 2154, 2150)
- [2] (0222, 0283)
- [5] (1560)

P. 106

☐MSD指排序時最重要的數(Most Significant Digit)
- 例：十進位的兩個數字X1X2X3和Y1Y2Y3（Xi和Yi都是0-9的數字，如356<412），X1和Y1是決定大小最重要的數字，故最左邊的數稱為MSD
- 字串排序以最左邊為MSD（如"john"<"mary"）

☐以356和412為例，觀察每個回合的分配和串接：

從MSD開始做：最後一次會按照LSD分配後並依序串接

串接的次序是錯的！

P. 107

Radix Implementation

```
void radixSort(int A[], int first, int last)
{ // the maximum in A is a d-digit integer
   for (j = d down to 1)  從最右側開始！
   {  Initialize 10 groups with counters reset;
      for (i = first to last)
      {    k = jth  digit of A[i];  第j位數字決定分組
           increase the counter of group k by 1;
           Append A[i] to group k;
      }    // end for            依序串接分組
      replace A with the sequence of group 1,… group 10
   }  // end for
} // end radixSort
```

先：分組
後：串接

P. 108

```
void radixSort(int A[], int first, int last)
{ int temp[MAX_SIZE], maxData;
  int bucket[10], i;
  for (maxData=A[first], i=first+1; i <= last; i++)
     if (maxData < A[i])
          maxData=A[i];           // d-digit integer
  for (int base=1; (maxData / base) > 0; base*=10)
  {  for (i=first; i <= last; i++)       // counting
          bucket[ (A[i] / base) % 10 + 1]++;
     …
  }  // end for
} // end radixSort
```

先：分組
後：串接

P. 109

```
void radixSort(int A[], int first, int last)
{ …
  for (int base=1; (maxData / base) > 0; base*=10)
  {  … bucket[0] = 0;
     for (i=1; i < 10; i++)       // the start of each group
          bucket[i] += bucket[i-1];
     for (i=first; i <= last; i++)     依序串接分組
          temp[ bucket[ (A[i] / base) % 10 ]++ ] = A[i];
  }  // end for
} // end radixSort
```

[0] (0004, 0123, 0222, 0283) — 0
[1] (1061, 1560) — 4
[2] (2150, 2154) — 6
[3] 0 — 8

P. 110

Implementation II

```
void radixSort(int A[], int first, int last)
{ int temp[MAX_D][MAX_SIZE], maxData;
  int counter[10]={0}, i, j;
  for (maxData=A[first], i=first+1; i <= last; i++)
     if (maxData < A[i])     maxData=A[i];
  for (int base=1; (maxData / base) > 0; base*=10)
  {  for (i=first; i <= last; i++)          // counting
     {    int LSD = (A[i] / base) % 10;
          temp[LSD][ counter[LSD] ] = A[i];
          counter[LSD]++;
     }    // end for                   LSD即代表分組
  …
```

[0]2 (1560, 2150)
[1]1 (1061)
[2]1 (0222)
[3]2 (0123, 0283)
[4]2 (2154, 0004)

P. 111

```
  for (base=1; (maxData / base) > 0; base*=10)
  {  …
     int k=0;                                   依序串接分組
     for (i=0; i < 10; i++)       // concatenate the groups
          if (counter[i] > 0)
          {    for (int j=0; j < counter[i]; j++, k++)
                    A[k]= temp[i][j];
               counter[i] = 0;
          }    // end if
  }  // end for
```

[0]2 (1560, 2150)
[1]1 (1061)
[2]1 (0222)
[3]2 (0123, 0283)
[4]2 (2154, 0004)

O(2*n*d) → **O(n)**

P. 112

Tree

Data-Management Operations

位置 導向
position oriented ADTs:

- Insert data into the i^{th} position
- Delete data from the i^{th} position
- Ask about the data in the i^{th} position

Ex. list, stack, queue, binary tree

需先知道要操作的位置!

內容 導向
Value oriented ADTs:

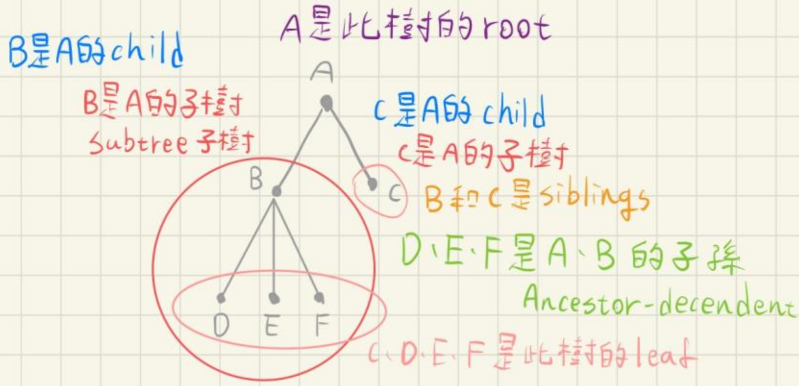
- Insert data according to its value.
- Delete data knowing only its value
- Ask about the data knowing only its value

Ex. sorted list, binary search tree.

不需先知道要操作的位置!

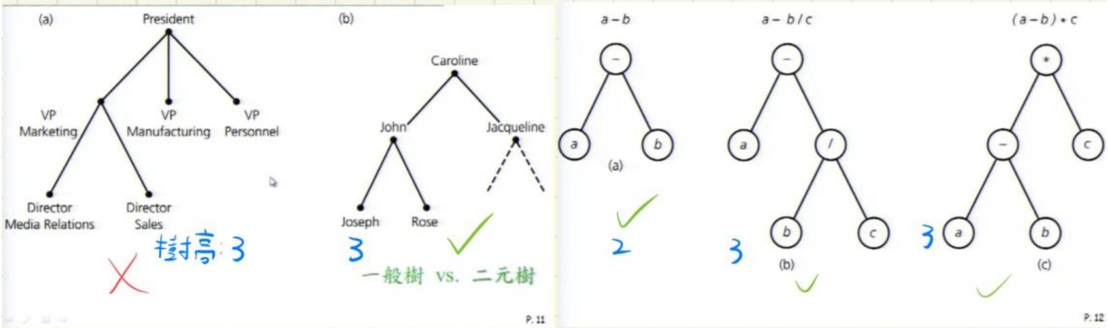
較能節省管理負擔!

Terminology



Binary Tree

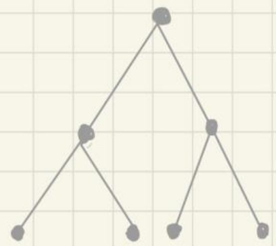
二元樹可以是零節點、一節點或二節點



樹高為能傳遞最遠的節點深度

樹越高，耗時越久！零節點樹高為0！

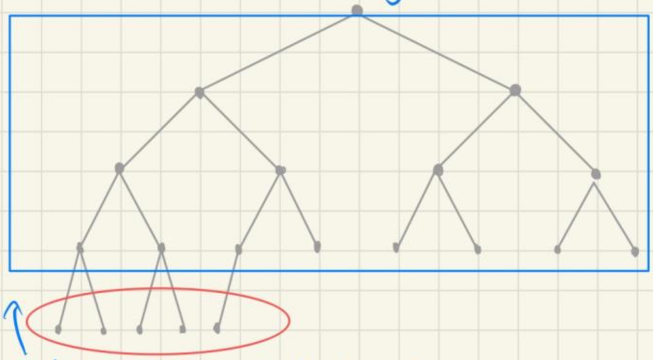
Full Binary tree



所有<h樹高的節點都有2個子節點

零節點也算完全樹!

Complete Binary tree

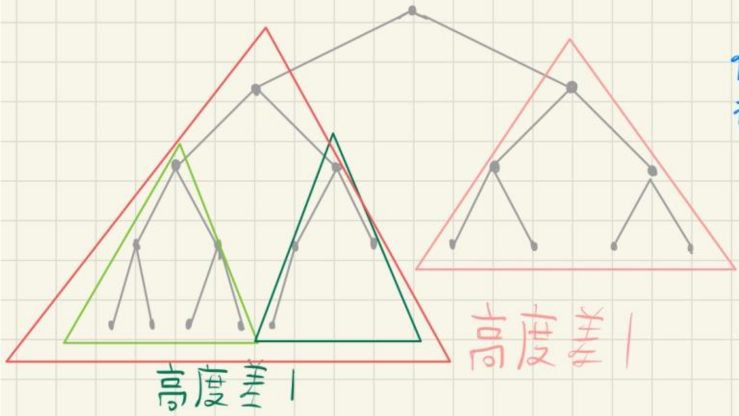


在level h-1為完全樹

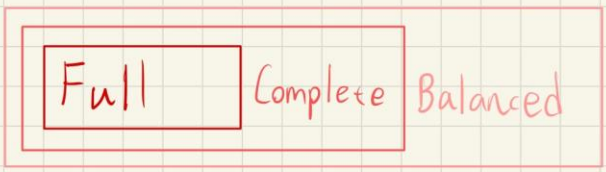
level h由左至右補滿!

此稱為完整樹

Balanced Binary tree



任一個節點的子樹樹高相差不大於1!



The ADT Binary Tree

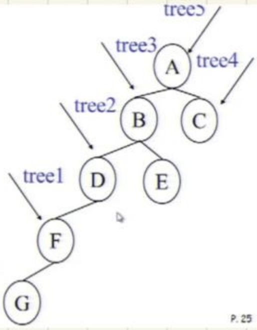
Binary tree
root left subtree right subtree
createBinaryTree() destroyBinaryTree() isEmpty() getRootData() setRootData() attachLeft() attachRight() attachLeftSubtree() attachRightSubtree() detachLeftSubtree() detachRightSubtree() getLeftSubtree() getRightSubtree() preorderTraverse() inorderTraverse() postorderTraverse()

建構
是否為空
存取樹根
新增
移除
(節點/子樹)
擷取
巡行

P. 24

☐ Building the ADT binary tree

tree1.setRootData('F')
tree1.attachLeft('G')
tree2.setRootData('D')
tree2.attachLeftSubtree(tree1)
tree3.setRootData('B')
tree3.attachLeftSubtree(tree2)
tree3.attachRight('E')
tree4.setRootData('C')
tree5.createBinaryTree('A',tree3, tree4)



P. 25

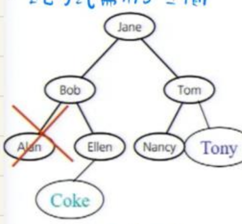
Array-based ADT Binary Tree

```
const int MAX_NODES = 100;          // maximum number of nodes

class TreeNode                      // a node in the tree            樹節點
{  private:
      TreeItemType item;            // data portion                  資料部分
      int leftChild;                // index to left child           左子節點
      int  rightChild;              // index to right child          右子節點
}; // end TreeNode

TreeNode tree[MAX_NODES];           // array of tree nodes
int root;                           // index of root                 樹根
int free;                           // index of free list            閒置串列
```

P. 27

效率較差!
↓

(a) 此方式需用3^n空間



attachRight(2,"Tony")
detachLeftSubtree(1)
attachLeft(4,"Coke")

(b) tree

	item	leftChild	rightChild
0	Jane	1	2
1	Bob	-1	4
2	Tom	5	6
3	Coke	–1	–1
4	Ellen	3	–1
5	Nancy	–1	–1
6	Tony	–1	-1
7	?	–1	8
8	?	–1	9
⋮	⋮	⋮	⋮

root: 0
free: 7

陣列表示法
-1表示null
待處理
Free list

P. 28

效率較佳!
↓

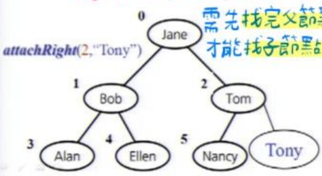
☐ If a binary tree remains *complete* 保持完整二元樹

– A *memory-efficient* array-based implementation

- **leftChild = 2*parent + 1**
- **rightChild = 2*parent + 2**
- **parent = (child-1) / 2**

attachRight(2,"Tony")

需先找完父節點
才能找子節點!



0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy
6	Tony
7	

P. 29

Properties

full binary tree

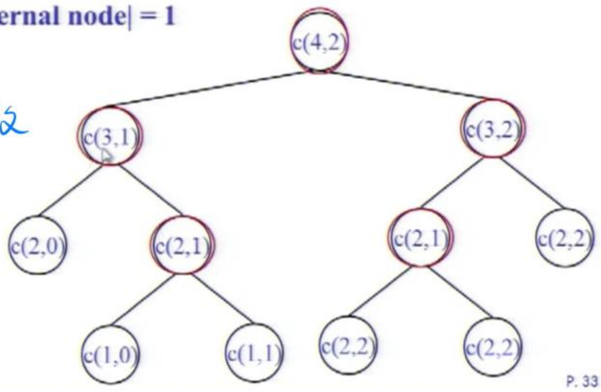
目前level的節點數 2^{h-1}

前level的節點數 $2^{h}-1$

	最大樹高	最小樹高
Complete binary tree	$\lceil \log_2(n+1) \rceil$	$\lfloor \log_2(n) \rfloor + 1$

- **Leaf nodes (N_0):** recursive calls to base cases
- **Internal nodes (N_2):** recursive calls to non-base cases
- **|leaf nodes| - |internal node| = 1**
  - $N_0 = N_2 + 1$

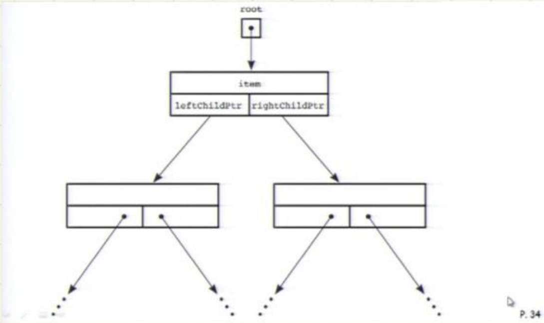
總遞迴次數



Pointer-based ADT Binary Tree

```
const int MAX_NODES = 100;          // maximum number of nodes

class TreeNode                      // a node in the tree          樹節點
{  private:
      TreeItemType item;            // data portion                資料部分
      TreeNode *leftChildPtr;       // pointer to left child       左子節點
      TreeNode *rightChildPtr;      // pointer to right child      右子節點
}; // end TreeNode

TreeNode *root;                     // pointer to the root
```

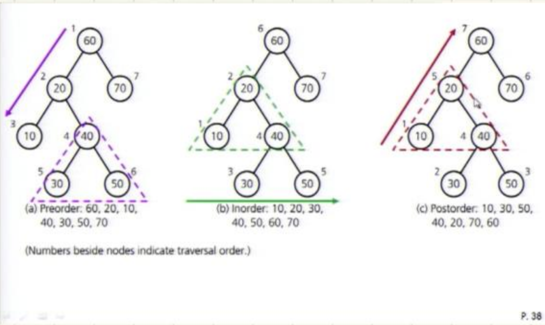
P. 35

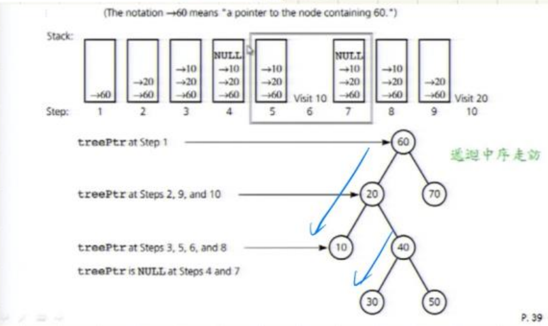


- **A traversal *visits* each node in a tree**
  - You do something with or to the node during a visit
  - For example, display the data in the node
- **General form of a *recursive* traversal algorithm**

traverse (in binTree:BinaryTree)　　先左　後右
　if (binTree is not empty)　　先印　前序*preorder*
　{
　　traverse(Left subtree of binTree's root)　　中間印　中序*inorder*
　　traverse(Right subtree of binTree's root)
　}　　後序*postorder*　後印

P. 36





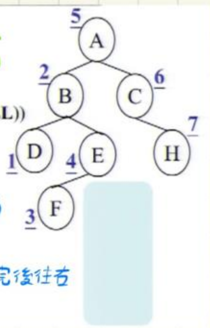
Non-recursive Traversal

中序

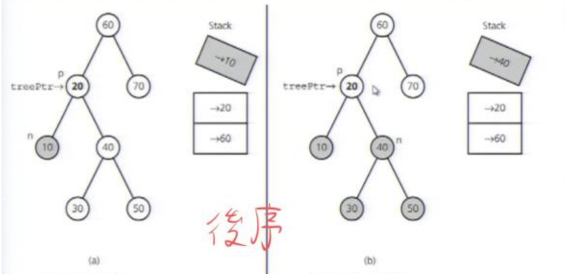
```
inorderTraversal(binaryTree root)
{ binaryTree treePtr = root;
  nodeStack aStack;

  while (!aStack.empty() || (treePtr != NULL))
  {  while (treePtr != NULL)
     {     aStack.push(treePtr);
           treePtr = treePtr->leftChild
     } // end while on treePtr
     aStack.pop(treePtr);
     cout << treePtr->data << endl;
     treePtr = treePtr->rightChild ;
  } // end while
}
```

不斷往下找

找完後往右



後序



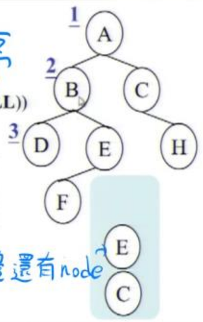
(a) Left subtree of 20 has been traversed. Pop reference to 10 from stack, visit 20.

(b) Right subtree of 20 has been traversed. Pop reference to 40 from stack.

前序

```
preorderTraversal(binaryTree root)
{ binaryTree treePtr = root;
  nodeStack aStack;

  while (!aStack.empty() || (treePtr != NULL))
  {   while (treePtr != NULL)
      {
      cout << treePtr->data << endl;
      aStack.push(treePtr->rightChild);
      treePtr = treePtr->leftChild;
      } // end while on treePtr
      aStack.pop(treePtr);
  } // end while
}
```

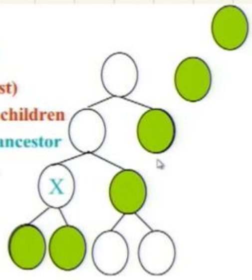
右邊還有node



Recursive Traversal

Preorder *successor*

- Possible positions?
  1. **Child (left child first)**
  2. **Sibling if X has no children**
  3. **Right child of one ancestor**
     - along the path…

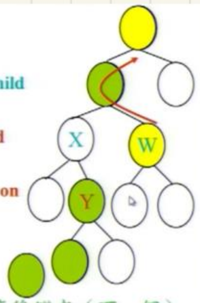


1. 小孩
2. 兄弟
3. 祖先的右小孩

前序後繼者（下一個）

Inorder *successor*

- Possible positions?
  1. **Leftmost descendant of right child**
     - along the path…
  2. **Right child if Y has no left child**
  3. **Parent if X has no right child**
  4. ***First-turn-right* ancestor if X is on the right**



1. 右小孩的最左邊子孫
2. 右小孩
3. 父親
4. 往上追溯第一個右轉的祖先

中序後繼者（下一個）

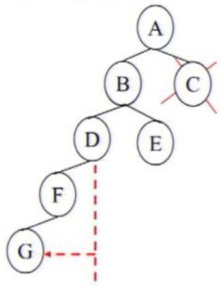
Try again!

Preorder: A B D F G E C

Preorder: … XY…
1. Child (left child first)
2. Sibling if X has no children
3. Right child of one ancestor

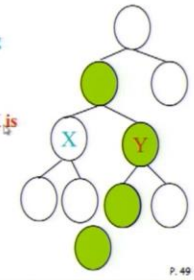
Inorder: G F D B E A C

Inorder: … XY…
1. Leftmost descendant of right child
2. Right child if Y has no left child
3. Parent if X has no right child
4. *First-turn-right* ancestor if X is on the right



Postorder *successor*

- Possible positions?
  1. **Leftmost descendant of sibling**
     - along the path…
  2. **Sibling if Y has no children**
  3. **Parent if X has no sibling or X is on the right**



1. 兄弟的最左邊子孫
2. 兄弟
3. 父親

後序後繼者（下一個）