單元六： Queue

- A queue
  - New items enter at the back, or rear, of the queue —— 後端
  - Items leave from the front of the queue —— 前端
  - First - in, last - out (FIFO) property
    - The first item inserted into a queue is the first item to leave (先進先出)

□ ADT queue operations
- Create an empty queue
- Destroy a queue
- Determine whether a queue is empty
- Add a new item to the queue
- Remove the item that was added earliest
- Retrieve the item that was added earliest

□ Operation Contact 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)
  {query} throw QueueException —— 擷取
dequeue(out queueFront : QueueItemType)
  throw QueueException —— 擷取後移除

- convert a sequence of digits into the decimal value

```
aQueue.createQueue()
while (not end of line) {
  Read a new character ch —建構
  aQueue.enqueue(ch)      —新增
}// end while

do {
  aQueue.dequeue(ch)
} // while (ch is blank)

n = 0
done = False
```

```
while (! done and ch is digit) {
  n = n * 10 + integer represented by ch
  if (aQueue.isEmpty())
    done = True
  else
    aQueue.dequeue(ch)
} // end while

if (!done and ch == '-') {
  aQueue.dequeue(ch)
  p = 0
  while (! done and ch is digit) {
    n = n * 10 + integer of ch
    p++
    if (aQueue.isEmpty())
      done = True
    else aQueue.dequeue()
  } // while
  n = n * (a.1)^p
} // if()
```

- Recognize Palindromes
  - To recognize a palindrome, you can use a queue in conjunction with a stack
    - A stack reverses the order of occurrences
    - A queue preserves the order of occurrences 保留原始順序
  - A nonrecursive recognition algorithm for palindromes 非遞迴
    - As you traverse the character string from left to right, insert each character into both a queue and a stack
    - Compare the characters at the front of the queue and the top of the stack

入列的前端 V.S. 堆疊的頂端

string : a b c b d

Queue : a b c b d (front ↑ back ↑)

Stack : d (top) / b / c / b / a

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

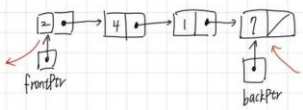
```
isPal(in str: string): boolean
aQueue.createQueue()
aStack.createStack()
for (the next character ch in str) {
  aQueue.enqueue(ch)
  aStack.push(ch)
} // for()
charEqual = True

while (!aQueue.isEmpty() && charEqual) {
  aQueue.getFront(front)
  aStack.getTop(top)
  if (front == top) {
    aQueue.dequeue()
    aStack.pop()
  } // if()
  else charEqual = false
} // while
```

— A Linked list with two external references

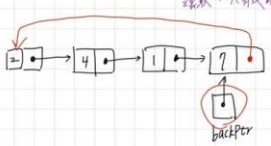
- A reference to the front 前端
- A reference to the back 後端

一般連結串列



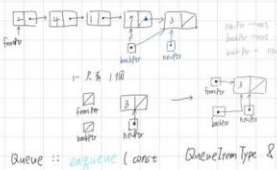
— A circular linked list with one external reference
- Only a reference to the back 環狀，只有後端

環狀連結串列



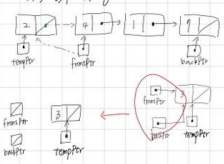
```
class Queue {
public:
    Queue();
    Queue( const Queue & Q );
    ~Queue();
    bool isEmpty() const;
    void enqueue( const QueueItemType & newItem );
    void dequeue();
    void dequeue( QueueItemType & queueItem );
    void getFront( QueueItemType & queueItem ) const;
private:
    struct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

— 插入 enqueue




```
void Queue :: enqueue ( const QueueItemType & newItem ) {
    QueueNode * newPtr = new QueueNode;
    newPtr -> item = newItem;
    newPtr -> next = NULL;
    if ( isEmpty() )
        frontPtr = newPtr;
    else    backPtr -> next = newPtr
    backPtr = newPtr;
} // Queue()
```

— 移除 dequeue




```
tempPtr = frontPtr;
frontPtr = frontPtr -> next;
tempPtr -> next = NULL;
delete tempPtr
```

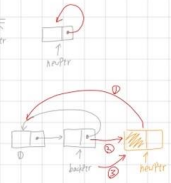


```
tempPtr = frontPtr;
frontPtr = NULL;
backPtr = NULL;
tempPtr -> next = NULL;
delete tempPtr
```

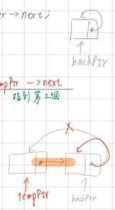
Practice 6-1 : Circular Queue

— Modify the enqueue and dequeue operations in the pointer-based implementation such that a circular linked list is used. (backPtr only)

```
void Queue :: enqueue ( const QueueItemType & newItem ) {
    QueueNode * newPtr = new QueueNode;
    newPtr -> item = newItem;
    if ( isEmpty() )
        newPtr -> next = newPtr ;
    else {
        newPtr -> next = backPtr -> next ;   ①
        backPtr -> next = newPtr;  第2個     ②
    } // else
    backPtr = newPtr ;  ③
} // enqueue()
```

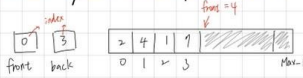



```
void Queue :: dequeue() throw ( QueueException ) {
    if ( isEmpty() )
        throw ... ;
    else {
        QueueNode *tempPtr = backPtr -> next;   指到第1個
        if ( backPtr == backPtr->next )
            backPtr == NULL;
        else backPtr -> next = tempPtr -> next   指到第2個
        tempPtr -> next = NULL;
        delete tempPtr;
    } // else
} // dequeue()
```



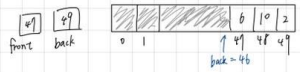
— An Array-Based Implementation

(a)

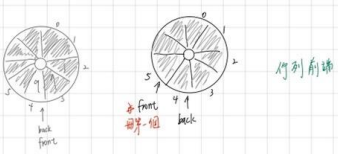


環狀陣列 back+1 = front ? 全空

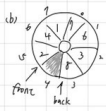
(b)

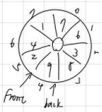


(c)



(b)





☐ Inserting into a queue

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

☐ Deleting from a queue

```
front = (front+1) % Max_QUEUE;
-- count;
```

全空的時候 count = 0
全滿的時候 count = Max_QUEUE

Practice 6-2 : Solution

```
Queue :: Queue(): back(MAX_QUEUE), front(0), isFull(False) { ... }
bool Queue :: isEmpty() const {
    return ( !isFull ) && ( front == ( back+1 ) % MAX_QUEUE );
} // isEmpty()
void Queue :: enqueue ( const QueueItemType & newItem ) throws ( QueueException ) {
    if ( isFull == TRUE )
        throw ... ;
    else {
        back = ( back+1 ) % MAX_QUEUE;
        items[back] = newItem;
        if ( front == ( back+1 ) % MAX_QUEUE )
            isFull = TRUE;
    } // else
} // enqueue()
```

```
void Queue :: dequeue() throws(QueueException) {
    if (isEmpty())
        throw ... ;
    else {
        front = (front+1) % MAX_QUEUE;
        if (isFull == TRUE)
            isFull = False;
    }//else
}// dequeue()
```

An Implementation That Uses the ADT List

- front : position 1
- back : the end

□ enqueue()

aList.insert(aList.getLength()+1, newItem)

□ dequeue()

aList.remove(1)

□ getFront(queueFront)

aList.retrieve(1, queueFront)

- Application : Simulation

□ Simulation

- modeling behavior
  - statistics 統計
  - predict 預測

□ 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

□ 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

等候時間
排隊長度

Arrival	duration	Departure	waiting
20	5	25	0
22	4	29	3
23	2	31	6
30	3	34	1

AWT = 10/4 = 2.5 (sum/4)
MWT = 6
MQL = 2
AQL = (1+2+1)/4 = 1

□ The bank simulation is concerned with

- Arrival events 輸入決定時間
- Departure events 模擬決定時間

Simulate()　事件驅動

```
Create an empty bankQueue;
Create an empty eventList;
Get the earlist 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
```

Practice 6-3 : Simulation by Queue

△ Single bank queue

Arrival	transaction	Departure	waiting
5	9	14	0
7	5	19	14-7=7
14	5	24	19-14=5
30	5	35	0
32	5	40	35-32=3
34	5	45	40-34=6
38	5	48	45-38=7

單一佇列

三種系統架構

□ Single teller / single queue
□ Multiple tellers / single queue
□ Multiple tellers / multiple queues

Practice 6-4 : Multi-queue Simulation

1個主排隊 | 2個queue

- Use the following event list to simulation two bank queues with bankQueue | first selection strategy and calculate the average waiting time.

單元1，演算法的基本觀念

Measuring the Efficiency of Algorithms

- Analysis of algorithms
  - Time efficiency
  - space efficiency
- A comparison of algorithms
  - Should focus on significant differences in efficiency 顯著差異
  - Should not consider reductions in computing costs due to clever coding tricks

1. specific implementation
2. computer
3. data

Example 1.

- Traverse a linked list of n nodes
  - n+1 comparisons, n+1 assignments, n writes

```
Node * cur = head;              // 1 assignment
while ( cur != NULL ) {         // n+1 comparisons
    cout << cur->item << endl;  // n writes
    cur = cur -> next;          // n assignments
}// while()
```

time : (n+1) * (c+a) + n * w

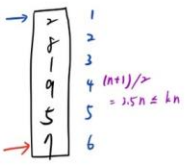
看電腦

- Definition of the order of an algorithm ★小到大
  - Algorithm A is order f(n) - denoted O(f(n))
    存在兩個常數 k 和 n_0，使演算法A能夠在不超過 k*f(n) 時間內解決大小不少於 n_0 的問題，稱 A 為 order (f(n))
- Properties of growth-rate functions
  - $O(n^3+3n)$ is $O(n^3)$: ignore low-order terms 忽略低階項
  - $O(5f(n)) = O(f(n))$: ignore multiplicative constant in the high-order term 忽略常數
  - $O(f(n)) + O(g(n)) = O(f(n)+g(n))$

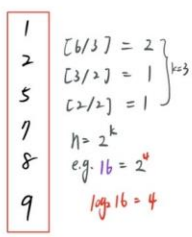
$O(1) < O(\log_2 n) < O(n) < O(n*\log_2 n) < O(n^2) < O(n^3) < O(2^n)$

- Worst-case analysis 最多
  - A determination of the maximum amount of time that an algorithm requires to solve problems of size n
- Average-case analysis 平均
  - 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)



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



$n = 2^k$

e.g. $16 = 2^4$

$\log 16 = 4$

ex: 10^6 筆資料 $\Rightarrow \log_2 10^6 = 19.9$

一百萬筆資料只需要做二十次比較！

- Sequential search on Sorted Data
  - Sorted v.s. unsorted 循序搜尋
  - Worst v.s. average v.s. best

	sorted	unsorted	found
Worst case	O(n)	O(n)	O(n)
Average case	O(n)	O(n)	O(n)
Best case	O(1)	O(n)	O(1)

Efficiency of Sorting Algorithms

- Sorting
  - A process that organizes a collection of data into either ascending or descending order
  - The sort key 排序鍵
    - The part of a data item that we consider when sorting a data collection 搜尋鍵

Student ID | Name | Birthday | Phone | Address | Grade

- Categories of sorting algorithms
  - An internal sort 內部排序
    - Require that the collection of data fit entirely in the computer's main memory

— An external sort 外部排序

▣ The collection of data will not fit in the computer's main memory all at once, but must reside in secondary storage

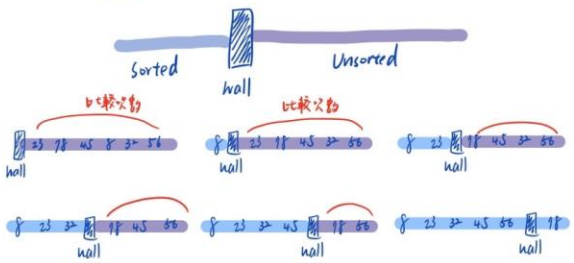
Stable

相同值維持不變的排序

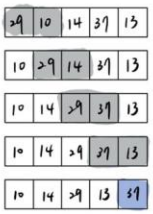
□ Stable Sort v.s. Unstable sort

bubble	selection
insertion	quick
merge	heap
radix	

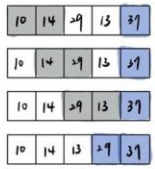
• Bubble sort



(a) Pass 1



(b) Pass 2



```
void bubbleSort ( int A[], int n ) {
  for ( pass = 1 ; pass < n ; ++pass ) {
    for ( int index = 0 ; index < n-pass; index++ ) {
      if ( A[index] >= A[index+1] )
        Swap ( A[index], A[index+1] )
    } // for
  } // for
} // end bubbleSort
```

all $1+(n-1)$ assignments; n comparisons

In each pass $1+(n-pass)$ assignments; $n-pass$ comparisons

>= unstable; stable

For each array item: 1 comparison

1 data exchange = 3 assignments (moves)

Comparisons:

$$n + \sum_{pass=1 \cdots n-1}(n-pass+1) + \sum_{pass=1 \cdots n-1}(n-pass)$$
$$= n + 2[n*(n-1) - n*(n-1)/2] + (n-1) = n^2/2 - \cdots \Rightarrow O(n^2)$$

What is the worst case ?

Ans, 最差的情況 ⇒ 終止, $O(n^2)$ comparisons

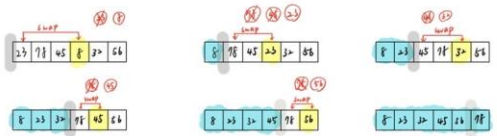
What is the best case ?

Ans, 最好的情況 ⇒ 已排序

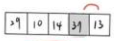
still comparison → $O(n^2)$ ⟹ 如何判斷已排序(in 1)

Best case, $O(n)$ comparisons

• Selection Sort 大隊接力選拔

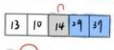


Initial array:

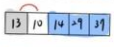


After 1st swap:

After 2nd swap:



After 3rd swap:



After 4th swap:

```
void selectionSort ( int A[], int n ) {
  for ( last = n-1 ; last > 0 ; --last ) {   n-1個回合
    int largest = indexOfLargest ( A, last+1 );  找出最大值的位置
    Swap ( A[largest], A[last] );  將最大值移到最末端
  } // end for
} // end selectionSort

int indexOfLargest ( int A[], int size ) {
  int indexSoFar = 0;
  for ( index = 1; index < size; ++index ) {
    if ( A[indexSoFar] < A[index] )
      indexSoFar = index;
  } // for()
  return indexSoFar;
} // end indexOfLargest()
```

Data exchanges: $n-1$ swaps ⇒ $O(n)$

Comparisons: $\sum_{size=n \cdots 2}(size-1) = (n-1)+(n-2)+\cdots+1 = n(n-1)/2$

⇒ $O(n^2)$

What is the worst case ?

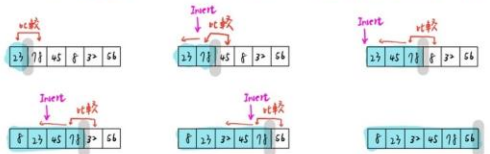
Ans, 最差的情況 ⇒ 終止, $O(n^2)$ comparison; $O(n)$ swaps

What is the best case ?

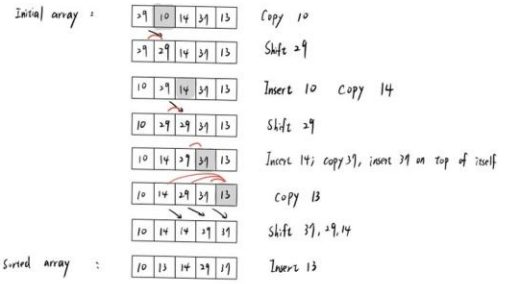
Ans, 最好的情況 ⇒ 已排序 $O(n^2)$ comparison; $O(n)$ swaps → $O(1)$ 相當於做交換 (no data transfers)

• Insertion Sort 撲克牌

插入時搬動, 比swap少



將未排序區的第一筆正確插入已排序區中




```
void insertionSort ( int A[], int n ) {                    n-1 個回合
    for ( unsorted = 1; unsorted < n; ++ unsorted ) {
        int loc = unsorted, nextItem = A [unsorted];
        for (; (loc > 0) && (A[loc-1] > nextItem); -- loc ) {
            A [loc] = A [loc - 1];
        A[loc] = nextItem;
    }// for()
}// insertionSort()
```

逐一往後挪動
直到 nextItem 的位置

>= not stable

Outer for loop : n-1 times

Inner for loop : at most unsorted times, unsorted = 1 ... n-1

Worst case : 1 + 2 + ... + (n-1) = n* (n-1)/2 comparisons / moves

What is the worst case ?

Ans. 最差的情況 ⇒ 對逆序, $O(n^2)$ comparisons

What is the best case ?

Ans. 最好的情況 ⇒ 已排序 O(n) comparisons
O(n) moves

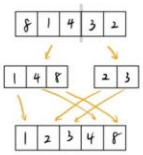
• Shell Sort 插入排序的變形 (Not Stable)

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

- Worst case = $O(n^2)$
- average case = $O(n(\log_2 n)^2)$

• Merge Sort (Stable)

- recursive
- Divide-and-conquer
- 先：分組各自排序
- 後：迴合併迴排序
- 效率穩定



先：分組各自排序

排序

後：合併+排序

① 內部排序

② 外部排序

```
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
    }// if()                                   3*n-2*
}// mergeSort()

void merge ( DataType theArray[], int first, int mid, int last ) {
    DataType tempArray [MAX_SIZE];
    int first1 = first, last1 = mid;
    int first2 = mid+1, last2 = last;
    int index = first;
    for (; (first1 <= last1) && (first2 <= last2); ++ index ) {
        if ( theArray [first1] < theArray [first2] ) {
            tempArray [index] = theArray [first1];
            ++ first1;
        }// if()
        else {
            tempArray [index] = theArray [first2];
            ++ first2;
        }// else
    }// for()      ; ⟶ 把資料放進去
```

<= stable

兩組中最小的優先

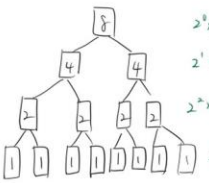
合併

∴ (n-1) + 2n ⇒ O(n)

theArray <==> tempArray : 2n moves

∴ (n-1) + 2n = 3*n-1 major operations ⇒ O(n)

O ($\log_2 n$)



$2^3 \times (3\times 8-1)$ O(n)

$2^2 \times (3\times 4-1)$ O(n)

$2^1 \times (3\times 2-1)$ O(n)

$2^0 \times (3\times 1-1)$ O(n)

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

優：速度快

缺：需有第二個陣列 比 original array 大

mergeSort(0,9) → MS(0,4) → MS(0,2) → MS(0,1)

→ MS(0,0), MS(1,1), merge(0,0,1) → MS(2,2), merge(0,1,2)

→ MS(3,4) → MS(3,3), MS(4,4), merge(3,3,4) → merge(0,2,4)

→ MS(5,9) → MS(5,7) → ... → MS(8,9) → ... → merge(5,7,9)

→ merge(0,4,9)

ms (0, 9)

ms (0, 4)

ms (0, 2)

ms (0, 1) → ms(0,0), ms(1,1), merge(0,0,1)

ms (2, 2)

merge (0,1, 2)

ms(3,4) → ms(3,3), ms(4,4), merge(3,3,4)

merge (0,2, 4)

ms (5, 9)

ms(5,7)

ms(5,6) → ms(5,5), ms(6,6), merge (5,5,6)

ms (7,7)

merge (5,6,7)

ms(8,9) → ms(8,8), ms(9,9), merge (8,8,9)

merge (5,7,9)

merge (0,4,9)

● Quick Sort 快速排序

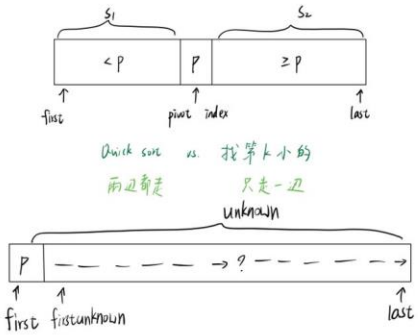
— divide-and-conquer

— Choose pivot 先分組 (軸的位置)

↳ is now in correct sorted position

▣ Item < pivot — Sort the left section

▣ item >= pivot — Sort the right section




```
void partition() {
    int lastS1 = first;
    int firstUnknown = first+1;
    while (firstUnknown <= last) {
        if (A[firstUnknown] < p)   O(n)
            move A[firstUnknown] into S1
        else move A[firstUnknown] into S2
        ++firstUnknown;
    } // while()
} // partition
```

25 33 60 22 45 15

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

```
void quickSort(DataType theArray[], int first, int last) {
    int pivotIndex;
    if (first < last) {
        partition(theArray, first, last, pivotIndex);
        quickSort(theArray, first, pivotIndex-1);
        quickSort(theArray, pivotIndex+1, last);
    } // if()
} // quickSort()
```

先: 依軸分類 partition → O(n)

後: 遞迴呼叫 recursive call → O(log n)

Original array:

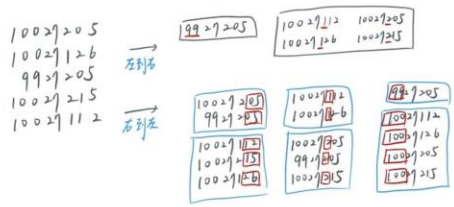
— Average case: $O(n * \log_2 n)$

— Worst case: $O(n^2)$ (sorted, smallest pivot)

☆ ● Radix Sort 基數排序 (速度最快)

— 10 進位系統

— 分解取部份值, 分配至對應容器



— LSD (Least significant Digit)
依照最右側數字分組, 串接

— MSD (Most significant Digit)

```
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];
    } // for()

    for ( int base = 1; (maxData/base) > 0; base *= 10 ) {
        for ( i = first; i <= last; i++ )
            bucket[ (A[i]/base) % 10 + 1 ]++;
        bucket[0] = 0;
        for (i=1; i<10; i++)
            bucket[i] += bucket[i-1];
        for (i= first; i <= last; i++)
            temp[ bucket[(A[i]/base) % 10]++ ] = A[i];
        for (i= first; i <= last; i++)
            A[i] = temp[i];
    }//for()
} // radixSort()
```

① 找出最大值 (做多少次)

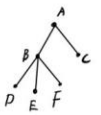
Counting (第?位數字檢分組)

— $O(2*n*d) \Rightarrow O(n)$

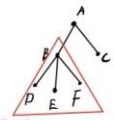
單元 8, Trees

- Binary tree 位置導向
- Binary Search Tree 內容導向

- Trees are composed of nodes and edges
- Trees are hierachical
  - Parent - child relationship between two nodes 親子關係
  - Ancestor - descendent relationships among nodes 祖孫關係
- Subtree of a tree : Any node and its descendants 子樹



- General tree
  - A general tree T is a set of one or more nodes such that T is partitioned into disjoint subsets :
    1. A single node r, the root
    2. Sets that are general trees, called subtrees of r
  - Parent of node B 父節點
    - The node directly above node B in the tree
  - Child of node B 子節點
    - A node directly below node B in the tree
  - Root
    - The only node in the tree with no parent
  - Subtree of node B
    - A tree that consists of a child (if any) of node B and the child's descendants



- Leaf 葉節點
  - A node with no child
- Siblings 兄弟節點
  - Nodes with a common parent
- Ancestor of node B 祖先節點
  - A node on the path from root to B
- Descendant of node B 子孫節點
  - A node on the path B to a leaf

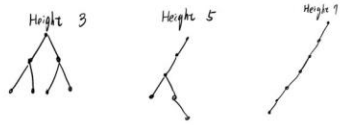
Binary Tree

- A binary tree is a set T of nodes such that either
  - T is empty, or
  - T is partitioned into three disjoint subsets :
    1. A single node r, the root
    2. Two possibly empty sets that are binary trees, called the left subtree of r and the right subtree of r

ex: 中序式適合做 二元樹

Height of a tree (階級)

— Number of nodes along the longest path from the root to a leaf



- Level of node n in a tree T
  - If n is the root of T, it is at level 1
  - If n is not the root of T, its level is 1 greater than the level of its parent
- Height of a tree T defined in terms of the levels of its nodes
  - If T is empty, its height is 0
  - If T is not empty, its height is equal to the maximum level of its nodes

最大階層 == 樹高

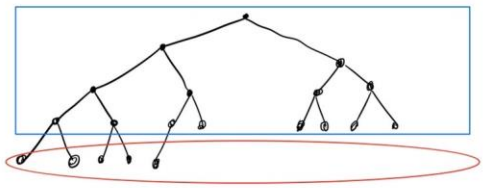
• A recursive definition of height 樹高的遞迴定義
- If T is empty, its height is 0
- If T is not empty,

height(T) = 1 + max{height(T_L), height(T_R)}

□ A binary tree of height h is full if 完全樹
- Node at levels < h have two children each

□ Recursive definition
- If T is empty, T is a full binary tree of height 0
- If T is not empty, and has height h > 0, T is a full binary tree if its root's subtree are both full binary trees of height h-1

Complete Binary Trees 完整樹
- It is full to level h-1, and
- Level h is filled from left to right

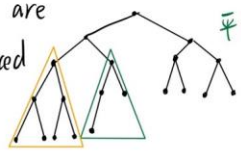


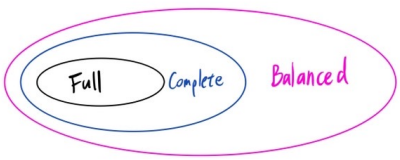
Balanced Binary Trees

□ A binary tree is balanced if the heights of any node's two subtrees differ by no more than 1

□ Complete binary trees are balanced

☑ Full binary trees are complete and balanced 平衡樹





Representations of Binary Tree

□ An array-based representation 陣列表示法
- Use an array of tree nodes 閒置串列
- Requires the creation of a free list that keeps track of available nodes

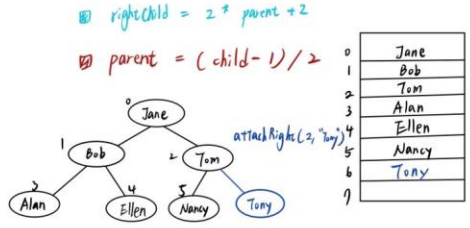
□ A pointer-based representation 指標表示法
- Nodes have two pointers that link the nodes in the tree

Array-based ADT Binary Tree

```
const int MAX_NODES = 100;
class TreeNode {
  private:
    TreeItemType item;    // 資料部分
    int leftChild;        // 左子節點
    int rightChild;       // 右子節點
};  // end TreeNode

TreeNode tree[MAX_NODES];
int root;   // 樹根
int free;   // 閒置串列
```

□ If a binary tree remains complete 保持完整=元樹
- A memory-efficient array-based implementation
  - leftChild = 2 * parent + 1
  - rightChild = 2 * parent + 2
  - parent = (child - 1) / 2



Height (h) v.s. Number of Nodes (n)

1. Full binary tree

Level (層數)	Number of nodes at this level	Number of nodes at this and previous levels
h	2^{h-1}	$2^h - 1$

2. Minimum / Maximum Tree Height 的公式推導

- Minimum height for n nodes
  - Complete binary tree
    - $h \le 2^{h-1} \rightarrow \log_2(n+1) \le \log_2(2^h) \rightarrow$
    - $h \ge \log_2(n+1) \rightarrow h = \lceil \log_2(n+1) \rceil$
- Maximum height for n nodes
  - skewed binary tree : h
  - Complete binary tree
    - $(2^{h+1} - 1) + 1 \le h \rightarrow \log_2(2^{h+1}) \le \log_2(n) \rightarrow$
    - $h \le \log_2(n) + 1 \rightarrow h = \lceil \log_2(n) \rceil + 1$

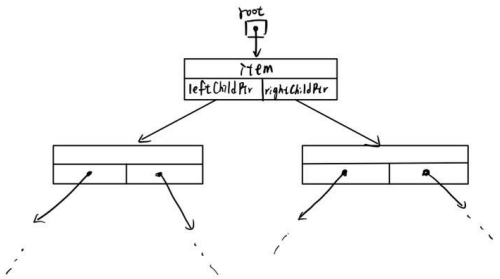
Properties

- Notes
  - N_2 = 3 nodes with two children
  - N_0 = 4 leaves
  - $N_0 = N_2 + 1$?
  - B: # Branches (edges)
    - $B = |E| = 2 * N_2 + 1 * N_1$

Flash back : Property of Binary Trees

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

Pointer-based ADT Binary Tree

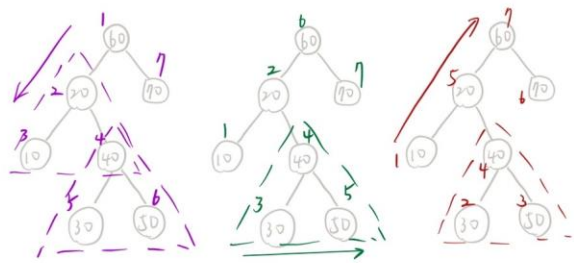


- A traversal visits each nodes in a tree
  - You do something with or to the node during a visit
  - For example, display the data in the node
- General form of a recursive traversal algorithm

```
Traverse ( in binTree : BinaryTree )
    if ( binTree is not empty ) {
        traverse ( Left subtree of binTree's root )
        traverse ( Right subtree of binTree's root )
    } // if ()
```

前序 preOrder
中序 inorder
後序 postorder

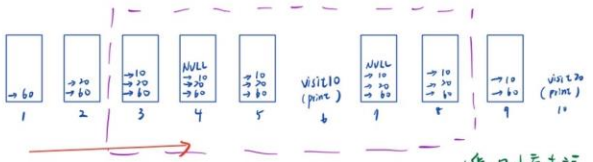
- PreOrder traversal 在 ... 之前
  - Visit root before visiting its subtrees
    - Before the recursive calls
- InOrder traversal 在 ... 中間
  - Visit root between visiting its subtrees
    - Before the recursive calls
- PostOrder traversal 在 ... 之後
  - Visit root after visiting its subtrees
    - Before the recursive calls

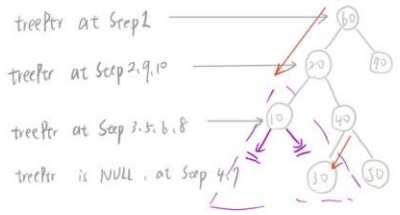


preorder :
60, 20, 10, 40, 30, 50, 70

inorder :
10, 20, 30, 40, 50, 60, 70

postorder :
10, 30, 50, 40, 20, 70, 60





Non-recursive In-order Traversal

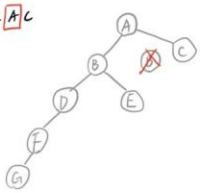
```
inorderTraversal (binaryTree root) {
    binaryTree treePtr = root;
    NodeStack aStack;
    while ( !aStack.empty() || (treePtr != NULL)) {
        while ( treePtr != NULL ) {
            aStack.push (treePtr);
            treePtr = treePtr -> leftChild;
        } // while
        aStack.pop (treePtr);
        cout << treePtr -> data << endl;
        treePtr = treePtr -> rightChild;
    } // while()
} // inorder Tree()
```

Non-recursive Pre-order Traversal

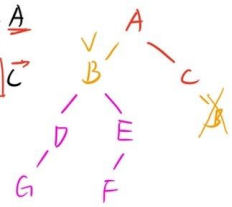
```
preorderTraversal (binaryTree root) {
    binaryTree treePtr = root;
    NodeStack aStack;
    while ( !aStack.empty() || (treePtr != NULL)) {
        while ( treePtr != NULL ) {
            cout << treePtr -> data << endl;
            aStack.push (treePtr->rightChild);
            treePtr = treePtr -> leftChild;
        } // while
        aStack.pop (treePtr);
    } // while()
} // preorder Tree()
```

Reconstruct the binary tree uniquely

ex: Preorder : A B D F G E C
Inorder : G F D B E A C

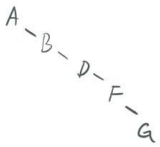


ex: Postorder : G D F E B C A
Inorder : G D B F E A C

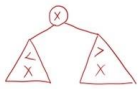


Preorder, Postorder 不能還原二元樹

ex: Preorder : A B D F G
Inorder : A B D F G



Binary Search Tree



- insert newItem into the binary search tree to which treePtr points

```
insertItem ( in treePtr : TreeNodePtr, in newItem : TreeItemType )
    if ( Search stops at X's left subtree )
        Make X's leftChildPtr point to newItem
    else
        Make X's rightChildPtr point to newItem
```

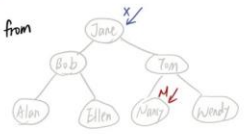
Average case = O(log n)
Worst case = O(n)

ADT Binary Search Tree : Deletion

- Three possible cases for deleting node X
  - X is a leaf
    - Set the pointer in X's parent to NULL
  - X has only one child
    - Let X's parent adopt X's only child

    Average case = O(log n)
    Worst case = O(n)
  - X has two children
    - Locate another node M that is easier to deleted
      - M is the leftmost node in X's right subtree
      - M will have no more than one child (0 or 1) : easier
      - M's key is called the inorder successor of X's key

        ↳ 可能有多達6好小孩
    - Remove the node M from the tree



Average case = O(log n)

Worst case : O(n)

① 無子孫 ⇒ 直接刪

② 有一子孫 ⇒ 提上去、刪除

③ 有二子孫 ⇒ 找到右邊的最左節點 copy 後刪除
如有子節點 再提上去

□ 最高比較次數 = 樹高

□ 加入和移除的次序會影響樹高

□ 隨機次序加入可逼近最小樹高

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

Sorting by Binary Tree

□ Tree sort

— Build a binary search tree by n Insertion

Average case : O(n * log n)

Worst case : O(n * n)

— Inorder traversal : O(n)

visit the nodes in sorted order

— Overall efficiency

Average case : O(n log n)

Worst case : O(n²)

Saving a Binary Search Tree in a File

□ Two Algorithms for saving a binary search tree

1. Perform preorder traversal while saving a binary search tree and then restore it to its original tree
2. Perform inorder traversal while saving a binary search tree and then restore it to a balanced tree 從中點開始找又（遞迴）