Ch 5

- rule: last in first out (LIFO)

- Applications of Stacks:
  (1) Check for balanced braces 檢查成對括號
  (2) Recognize strings in a language 辨識語言
  (3) Algebraic expression evaluation 代數運算式求解
  (4) Search a path 搜尋一條路徑

- Operation for ADT stack:
  - isEmpty( ) 是否為空
  - push (in newItem : StackItemType) 新增一筆
  - pop ( ) 移除最近一筆
  - getTop (out stackTop : StackItemType) 擷取最近一筆
  - pop (out stackTop : StackItemType) 擷取後移除最近一筆

  } throw StackException

- Examples of checking for balanced braces

```
while (not end of string) {
    if (ch == "[")
        aStack.push(ch)
    else if (ch == "]")
        aStack.pop( )
}
```

```
while (not end of string) {
    if (ch == "[")
        aStack.push(ch)
    else if (ch == "]") {
        if (!aStack.IsEmpty())
            aStack.pop()
        else return false
    }
}
```

```
while (not end of string && count >= 0) {
    get next ch;
    switch (ch) {
        case "[": count ++
        case "]": count --
    }
}
```

Ch 6

rule. first in first out (FIFO)

- Applications of Queue
  (1) Reading a string of characters 讀入字串
  (2) Recognize a palindrome 辨識迴文
  (3) Simulation 模擬

- A queue:
  - New items enter at the back, or rear, of the queue.
  - Items leave from the front of the queue.
  - First in first out (FIFO) property

- Queues:
  - Are appropriate for many real-world situations
  - Have applications in computer science
  - Simulation
    - A study to show how to reduced the wait involved in an application (自然界或人類的行為)

- Operations for ADT queue:
  - isEmpty() 是否為空
  - enqueue(in newItem: QueueItemType) 新增
  - dequeue() 移除
  - getFront(out queueFront: QueueItemType) 擷取
  - dequeue(out queueFront: QueueItemType) 擷取後移除

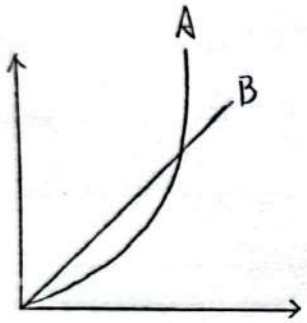
  } throw QueueException

- Simulation
  - A technique for modding the behavior of both natural and human-made systems (行為模型)
  - Goal:
    - Generate statistics that summarize the performance of an existing system
    - Predict the performance of a proposed system

Ch. 7

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

- 3 difficulties with comparing programs of algorithms:
  (1) specific implementation 實作
  (2) computer 電腦設備
  (3) data 資料



Algorithm A requires time proportional to n^2 growth-rate functions

Algorithm B requires time proportional to n

Algorithm A : $O(n^2)$, Algorithm B : $O(n)$ — Big O notation

- Growth-rate function $f(n)$
  - A mathematical function used to specify an algorithm's order in terms of the size of the problem
- Definition of the order of an algorithm
  - Algorithm A is order $f(n)$ - denoted $O(f(n))$

$O(1)$ $O(\lg n)$ $O(n)$ $O(n \lg n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

優 ←——————————→ 劣

$O(f(n)) + O(g(n)) = O(f(n)+g(n))$

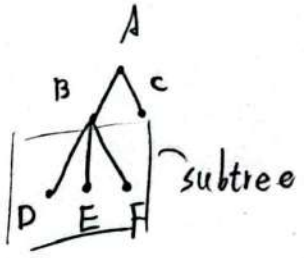
- Worst case : maximum amount of time an algorithm requires to solve problems of size n.
- Average case : average amount of time an algorithm requires to solve problems of size n.
- Best case : minimum amount of time an algorithm requires to solve problems of size n.

· Stable sort vs Unstable sort

bubble	quick	if a==b，排序完後仍保持
insertion	selection	a，b的順序 → stable
merge	heap	
radix		

	worst	best
bubble	$O(n^2)$	$O(n^2)$
insertion	$O(n^2)$	$O(n)$
merge	$O(n\log n)$	$O(n\log n)$
radix	$O(d(n+r))$	$O(d(n+r))$
quick	$O(n^2)$	$O(n\log n)$
selection	$O(n^2)$	$O(n)$

Ch 8



· Trees are composed of nodes and edges

· Trees are hierarchical
  - Parent-child relationship between 2 nodes
  - Ancestor-descendent relationships among nodes

· Subtree :
  - Any node and its descendents

· Leaf :
  - A node with no children

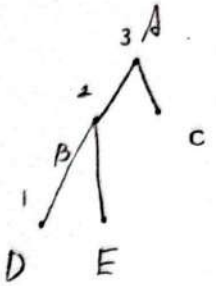
· Siblings :
  - Nodes with a common parent

· Ancestor of node B :
  - A node on the path from root to B

· Descendent of node B :
  - A node on the path from B to leaf

· Root
  - The only node in the tree with no parent

- 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) 2 possibly empty sets that are binary trees, called the left subtree of r and the right subtree of r

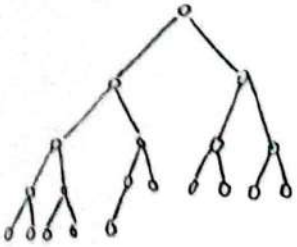


- Height of a tree:
  - Number of nodes along the longest path from the root to a leaf

- Level of a node in a tree T:
  - If n is the root of T, it's at level 1
  - If n isn't the root, its level is 1 greater than the level of its parent

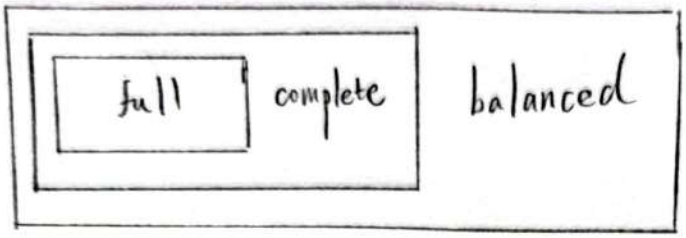
- Height of a tree T defined in terms of the levels of its nodes
  - If T is empty, its height is 0
  - If T isn't empty, its height is equal to the maximum level of its nodes

- Full Binary Tree
  - A binary tree of height h is full if nodes at levels < h have 2 children nodes

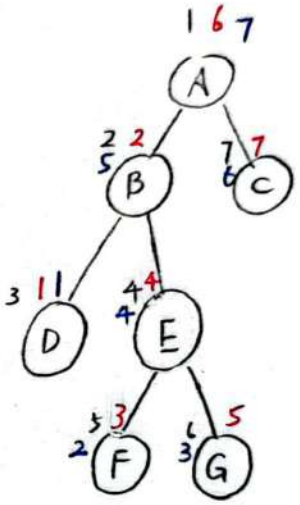


- Complete Binary Tree
  - A binary tree of height h is complete if
    (1) All nodes at levels <= h-2 have 2 children
    (2) When a node at level h-1 has children, all nodes to its left at the same level have 2 children
    (3) When a node at level h-1 has 1 child, it's a left child

- Balanced binary tree
  - A binary tree is balanced if the height of any node's 2 subtrees differ by no more than 1



- Traversals of a binary tree
  - Preorder Traversal
    - Visit root before visiting its subtrees
    - Before the recursive calls
  - Inorder Traversal
    - Visit root between visiting its subtrees
    - Between the recursive calls
  - Postorder Traversal
    - Visit root after visiting its subtrees
    - After the recursive calls



Preorder :
Inorder :
Postorder :