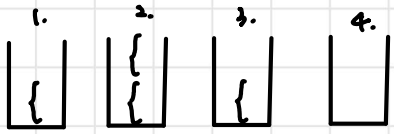
Stack

⇒ 處理對像加資料的最後一筆

ex. 括號(成對)

一右必有一左.

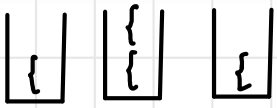
{a{b}c}



1. push
2. push
3. pop
4. pop

stack empty ⇒ balance

{a{bc}



1. push
2. push
3. pop

stack not empty ⇒ not balance

<way 2> 計數器

左括 +1 右括 -1

右括先遇 ⇒ counter < 0
結尾 counter > 0
} ⇒ 括號不成對

ex. 迴文 { push 前半 / $ / pop 後半

```
aStack.createStack
while (ch != $)
    aStack.push(ch)
isLanguage = true

while (string is not end && inLangeage)
    if (! aStack.isEmpty()) {
        aStack.pop(stackTop)
        ch = nextChar
        if (stackTop != ch)
            inLanguage = false
    }
    else   inLanguage = false   後半部比較長
    if (inLanguage && aStack.isEmpty()) 一樣長
        return true
    else   前半長 or 有錯
        return false
```

Stack :: push

```
newPtr = new StackNode;
newPtr -> Item = newItem
newPtr -> next = topPtr
topPtr = newPtr
```

新增刪除都會在第一個節點

Stack :: pop.

```
if ( ! isEmpty() ) {
    temp = topPtr
    topPtr = topPtr -> next
    temp -> next = NULL
    delete temp
}
```

Stack :: getTop (StackItemType & stackTop)

```
if (! isEmpty)
    stackTop = top -> Item
```

vector

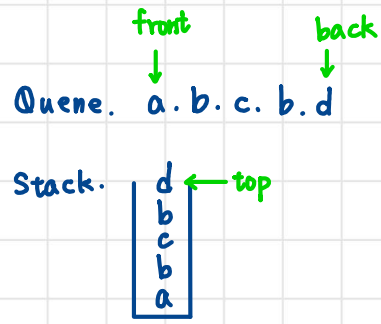
```
push (Item)
    insert (1, newItem)
pop
    remove (1)
getTop
    retrieve ( 1, stackTop)
```

Queue ⇒ FIFO (first-in, first-out)

佇列 ≡ 排隊

ex. 迴文：stack vs. quene.

```
isPal ( in str ) : boolean
    for ( the next ch in str) {
        aQuene.enquene (ch)
        aStack.push (ch)
    }
    charEqual = true
    while ( ! aQuene.Empty && charEqual ) {
        aQuene.dequene (front)
        aStack.pop (top)
        if (front != pop )
            charEqual = false
```

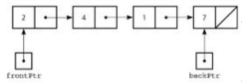


指標

```
class Queue
{   public:     Queue();                    // Constructors and destructor
                Queue(const Queue& Q);
                ~Queue();

                bool isEmpty() const;       // Queue operations
                void enqueue(const QueueItemType& newItem);
                void dequeue();
                void dequeue(QueueItemType& queueFront);
                void getFront(QueueItemType& queueFront) const;
    private:    struct QueueNode {
                        QueueItemType  item;
                        QueueNode     *next;
                        };          // end QueueNode
                QueueNode *backPtr;
                QueueNode *frontPtr;
};  // end Queue
```

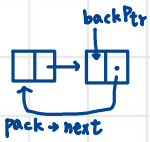
} 注意 specilal case

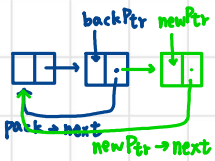


P. 18

*環狀

```
enquence
    newPtr → item = newItem
    if ( isEmpty ())
        newPtr → next = newPtr
    else {
        newPtr → next = backPtr → next
        backPtr → next = newPtr
    }
    back = newPtr
```





```
void Queue::dequeue() throw(QueueException)
{  if (isEmpty())
        throw QueueException("QueueException: …");
   else
   {    QueueNode *tempPtr = frontPtr;
        if (frontPtr == backPtr) {      // special case: one node only
                frontPtr = NULL;
                backPtr = NULL;
        }       // end if
        else    frontPtr = frontPtr->next;
        tempPtr->next = NULL;       // defensive strategy
        delete tempPtr;
   }    // end else
}  // end dequeue
```

Algorithm efficiency

(演算法)

- Time efficiency
- space efficiency

⇒ 影響
- 實作
- 電腦
- 資料

ex. Traverse a linked list of a node

```
for (Node *cur = head; cur != NULL; cur = cur -> next)
    cout << cur -> item << endl;
```

c: 比較 a: assighment w: write (受電腦影響)

$(n+1)(c+a) + n \cdot w$

⇒ Execution time is related to the number of operation

Big O Noation (位階)

存在2個常數 k 和 n_0

使演算法 A 能夠不超過 $k \cdot f(n)$ 時間內

解決大小不小於 n_0 的問題

⇒ 稱 A 為 order(n)

- 忽略低位階 ex. $O(n^3+3n)$ is $O(n^3)$
- 忽略常數 ex. $O(5f(n))$ is $O(5f(n))$
- $O(f(n)) + O(g(n)) = O(f(n)+g(n))$

ex.

$2.5^* n^2 - 2.5^* n$ is $O(?)$

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

$\forall n \geq 10 \ (2.5n^2 - 2.5n) \leq 1 \cdot n^{10}$

$\forall n \geq n_0 \ (2.5n^2 - 2.5n) \leq k \cdot n^2$

$\forall n \geq 0 \ (2.5n^2 - 2.5n) \leq 3 \cdot n^2$

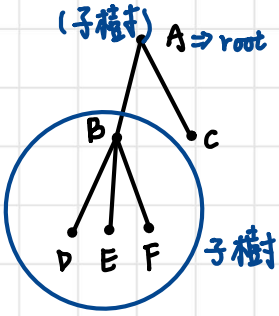
Tree

- Position-oriented ADTs 位置導向 ex. list. stack. quene. binary tree
- Value-oriented ADTs 內容導向 ex. sorted list. binary search tree

Paraent - child relationship between two nodes

Ancestor - descendant relationship among nodes

Subtree of A tree: Any node and its descendant
(子樹)



Leaf — A node with no children

Siblings — Nodes with a common parent ex. DEF

Binary tree

Full Binary tree 完全樹

⇒ Nodes at levels < h have two children

除了最尾巴其他都有2個小孩

沒有小孩

Complete Binary tree 完整樹

It is full to levels - h - 1,

and level h is fill from left to right

除了最尾巴其他都被填滿(有2個小孩)

從最左開始往右放

Balance Binary tree 平衡樹

Any node's two subtree differ by no more 1

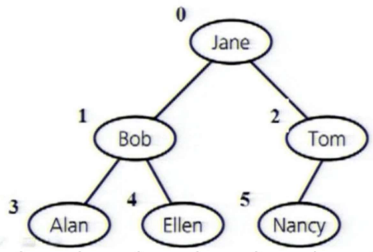
左右子樹差距不超過一

陣列

□ **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



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