資料結構

Data Structure

TZU-CHUN HSU[1]

[1]vm3y3rmp40719@gmail.com

[1]Department of Computer Science, Zhejiang University

2020 年 11 月 22 日

Version 2.0

Disclaimer

本文「資料結構」為台灣研究所考試入學的「資料結構」考科使用，內容主要參考 Introduction to Algorithms[1]，以及 wjungle 網友在 PTT 論壇上提供的資料結構筆記 [2]。本文作者為 TZU-CHUN HSU，本文及其 LaTeX 相關程式碼採用 **MIT 協議**，更多內容請訪問作者之 GitHub 分頁Oscarshu0719。

MIT License

Copyright (c) 2020 TZU-CHUN HSU

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1 Overview

1. 本文頁碼標記依照 TKB 筆記 [2] 的頁碼。

2. TKB 筆記 [2] 章節頁碼：

Chapter	Page No.	Importance
1	3	$\star\star\star$
2	259	$\star$
3	52	$\star\star\star$
4	259	$\star$
5	82	$\star\star\star\star\star$
6	228	$\star\star\star\star$
7	180	$\star\star\star\star$
8	221	$\star\star\star$
9	129	$\star\star\star\star$

Data structure	Page No.
Min-max heap	8
Deap	10
SMMH	10
AVL tree	11
m-way ST	11
Red-black tree	12
Splay tree	14
Leftist heap	15
Binomial heap	16
Fibonacci heap	17

3.
$$\begin{aligned}\log 2 &= 0.3010\\ \log 3 &= 0.4771\\ \log 5 &= 0.6990\\ \log 7 &= 0.8451\end{aligned} \tag{1}$$

4. OBST 在「演算法」中，不再贅述。

Trees				
Tree	Insert x	Delete x	Search x	Remark
BST	$O(\log n) \sim O(n)$			Create: $O(n \log n) \sim O(n^2)$
AVL tree				$F_{n+2} - 1 \le n \le 2^h - 1$
B tree	$O(\log_m n)$			$1 + 2\frac{\lceil \frac{m}{2} \rceil^{h-1} - 1}{\lceil \frac{m}{2} \rceil - 1} \le n \le 2\lceil \frac{m}{2} \rceil^{h-1} - 1$
RBT				$h \le 2\log(n+1)$
Splay tree				Worst: $O(n)$

Priority queues					
Operations	Max (Min)	Min-max & Deap & SMMH	Leftist	Binomial	Fibonacci
Insert x	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n), O(1)^*$	$O(1)^*$
Delete max	$O(\log n)$	$O(\log n)$			
Delete min	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
Delete x				$O(\log n)$	$O(\log n)^*$
Merge	$O(n)$		$O(\log n)$	$O(\log n)$	$O(1)^*$
Decrease key				$O(\log n)$	$O(1)^*$
Search x	$O(n)$				
Find max	$O(1)$	$O(1)$			
Find min		$O(1)$		$O(\log n)$	$O(1)$
Remark	Create: $O(n)$		Merge faster than Max (Min) heap.	Find min can be down to $O(1)$.	Decrease key is faster than binomial heap

Sorting algorithms					
Method	Time complexity			Space complexity	Stable
	Best	Worst	Average		
Insertion	$O(n)$	$O(n^2)$		$O(1)$	$\surd$
Selection	$O(n^2)$			$O(1)$	$\times$
Bubble	$O(n)$	$O(n^2)$		$O(1)$	$\surd$
Shell	$O(n^{1.5})$	$O(n^2)$		$O(1)$	$\times$
Quick	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n) \sim O(n)$	$\times$
Merge	$O(n \log n)$			$O(n)$	$\surd$
Heap	$O(n \log n)$			$O(1)$	$\times$
LSD Radix	$O(n \times k)$			$O(n + k)$	$\surd$
Bucket/MSD Radix	$O(n)$	$O(n^2)$	$O(n + k)$	$O(n \times k)$	$\surd$
Counting	$O(n + k)$				$\surd$

2 Summary

1. **Theorem (17)** Permutation：

```
function PERM(list, i, n)
    if i == n then
        PRINT(list)
    else
        for j := i to n do
            SWAP(list, i, j)
            PERM(list, i + 1, n)
            SWAP(list, i, j)
        end for
    end if
end function
```

2. **Theorem (58, 59)**

Priority	
Priority	Operator
1	(out of stack
2	$\uparrow$ out of stack
3	$\uparrow$ in stack
4	$*$, /
5	$+$, $-$
6	empty stack, (in stack

3. **Theorem (87)** 節點數：

$$n = (\sum_{i=1}^{\deg} i \times n_i) + 1 \tag{2}$$

$$n_0 = n_2 + 1 \text{（二叉樹）}$$

4. **Theorem (95, 97, 98)**

- 可以確定二叉樹，其他則否：
  - Preorder 和 Inorder。
  - Postorder 和 Inorder。
  - Level-order 和 Inorder。
  - Complete 和任意排序。

- Preoder = Inoder：Empty、Root、Right-skewed tree。
- Postoder = Inoder：Empty、Root、Left-skewed tree。
- Preoder = Postoder：Empty、Root。

5. **Theorem (101, 102, 103, 104)**

- 判斷二二叉樹是否相同：

```
function EQUAL(Tree s, t)
    res := False
    if s = NIL ∧ t = NIL then
        res := True
    else if s ≠ NIL ∧ t ≠ NIL then
        if s.data = t.data then
            if EQUAL(s.lchild, t.lchild) then
                res := EQUAL(s.rchild, t.rchild)
            end if
        end if
    end if
    return res
end function
```

- 計算節點個數：

```
function COUNT(Tree s)
    if s = NIL then
        return 0
    else
        return COUNT(s.lchild) + COUNT(s.rchild) + 1
    end if
end function
```

- 計算二叉樹高：

```
function HEIGHT(Tree s)
    if s = NIL then
        return 0
    else
        n_l := HEIGHT(s.lchild)
        n_r := HEIGHT(s.rchild)
        return MAX(n_l, n_r) + 1
    end if
end function
```

- 計算樹葉節點個數：

```
function LEAF(Tree s)
    if s = NIL then
        return 0
    else
        tmp := LEAF(s.lchild) + LEAF(s.rchild)
        if tmp > 0 then
            return tmp
        else
            return 1
        end if
    end if
end function
```

- 交換左右子樹：

```
function SWAPBT(Tree s)
    if s ≠ NIL then
        SWAPBT(s.lchild)
        SWAPBT(s.rchild)
        SWAP(s.lchild, s.rchild)
    end if
end function
```

6. **Theorem (116)**

```
function CreateMinHeap(Tree s, size n)
    for i := n/2 to 1 do                          ▷ Start from parent of the last node.
        tmp := s[i]
        j := 2 × i                                ▷ Left child of i.
        while j ≤ n do                            ▷ There is a child.
            if j < n then                         ▷ Right child exists.
                if s[j] > s[j + 1] then           ▷ Choose the smaller child.
                    j := j + 1
                end if
            end if
            if tmp ≤ s[j] then
                Break.
            else                                  ▷ Percolate one level.
                s[j/2] := s[j]
                j := j × 2
            end if
        end while
        s[j/2] := tmp
    end for
end function
```

7. **Theorem (122, 123, 124, 125))** Disjoint set:

Disjoint set		
Combination	Union	Find
Arbitrary Union & Simple find	$O(1)$	$O(h)$ Worst: $O(n)$
Union-by-height & Simple find	$O(1)$	$O(\log n)$
Union-by-height & Find with path compression	$O(1)$	$O(\alpha(m, n)) = O(\log^* n)$ close to $O(1)$

8. **Theorem (129, 130, 131)** Min-max heap:

- Complete。
- Root 為最小值。
- 最大值在第二層其中一個。
- 越下層 min-level 越大，越下層 max-level 越小。

```
function InsertMinMaxHeap(MinMaxHeap s, Element x)
    Put x at the last position n, which has parent p.
    if p is at min-level then
        if s[n].data < s[p].data then
            Swap(s[n], s[p])
            VerifyMin(s, p, x)
        else
            VerifyMax(s, n, x)
        end if
    else                                              ▷ p is at max-level.
        if s[n].data > s[p].data then
            Swap(s[n], s[p])
            VerifyMax(s, p, x)
        else
            VerifyMin(s, n, x)
        end if
    end if
end function
```

```
function DeleteMinMinMaxHeap(MinMaxHeap s)
    Remove the data of root. Copy the data of the last node as x and remove the last node.
    if Root has no children then                      ▷ Start.
        Set x as root and exit.
    else if Min is one of root's children k then
        if s[k].data < x.data then
            Swap(s[k], x)
        else
            Set x as root.
        end if
        Exit.
    else if Min grandchildren k and its parent p then
        if s[k].data < x.data then
            Swap(s[k], x)
            if x.data > s[p].data then
                Swap(x, s[p])
            end if
            Go to start.
        else
            Set x as root and exit.
        end if
    end if
end function
```

9. **Theorem (133, 134, 135)** Deap（Double-ended heap）：

- Complete。
- root 不存 data，root 左子樹是 min-heap，右子樹是 max-heap。
- root 左子樹中一節點必須 $<$ 右子樹中對應的節點。

```
function InsertDeap(Deap s, Element x)
    Put x at the last position n.
    if n is at min-heap then
        j is the corresponding position in the max-heap.
        if s[n].data > s[j].data then
            Swap(s[n], s[j])
            InsertMaxHeap(s, j, x)
        else
            InsertMinHeap(s, n, x)
        end if
    else                                                ▷ n is at max-heap.
        j is the corresponding position in the min-heap.
        if s[n].data < s[j].data then
            Swap(s[n], s[j])
            InsertMinHeap(s, j, x)
        else
            InsertMaxHeap(s, n, x)
        end if
    end if
end function
```

```
function DeleteMinDeap(Deap s)
    Replace the data of the left child of the root with the smaller of its children and
    recursively run the same process to its subtree, making an empty node i at the last level.
    Copy the data of the last node as x and remove the node.
    InsertDeap(x, i)
end function
```

10. **Theorem (136, 137)** SMMH（Symmetric min-max heap）：

- Complete。
- root 不存 data。
- 左兄弟節點 $\leq$ 右兄弟節點。
- 對一節點 x，祖父節點的左子節點 $\leq x$，祖父節點的右子節點 $\geq x$。

- 以一節點為 root，則該子樹最小值（不含 root）為左子節點，最大值（不含 root）為右子節點。

```
function InsertSMMH(SMMH s, Element x)
    Put x at the last position.
    Recursively swap those nodes which break the rules.
end function
```

```
function DeleteMinSMMH(SMMH s)
    Copy the data of the last node as x and remove the node.
    Replace the left child of the root with the smaller of the leftmost grandchild and the
  third grandchild of the root and replace the chosen the node with x.
    Recursively swap those nodes which break the rules.
end function
```

11. **Theorem (145, 151)** AVL tree：

- Height balanced BST.
- 平衡係數：左子樹高度減去右子樹高度。
- 左右子樹高度相差不超過 1，即平衡係數只能為 $-1, 0, 1$。
- 若不符合條件，根據父節點和祖父節點類型（LL, LR, RL, RR）調整樹。
- 高度為 h 的 AVL tree 且節點數為 n，則

$$F_{h+2} - 1 \leq n \leq 2^h - 1 \tag{3}$$

  其中 F 是費氏數列，最大值為 Full。

12. **Theorem (154, 155, 156, 158, 161)** B tree：

- Balanced m-way search tree。
- 若 order m、高度 h 且有 k 個 key，則

$$1 + 2\frac{\lceil \frac{m}{2} \rceil^{h-1} - 1}{\lceil \frac{m}{2} \rceil - 1} \leq k \leq 2\lceil \frac{m}{2} \rceil^{h-1} - 1 \tag{4}$$

- $$\begin{aligned} 2 \leq \deg(v) \leq m, v &\text{ is root} \\ \lceil \frac{m}{2} \rceil \leq \deg(v) \leq m, v &\text{ is NOT root or failure nodes} \end{aligned} \tag{5}$$

```
function InsertBTree(BTree s, Element x)
    Put x at proper position n.
    while n overflow do
        Choose the ⌈m/2⌉-th key of n (started from 1), move it to its parent, and split n.
        n := n.parent
    end while
end function
```

```
function DeleteBTree(BTree s, Element x)
    n := SearchBTree(s, x)
    if n is leaf then
        Delete n.
        while n underflow do
            if Can be rotated then
                Rotate.
                Break.
            else
                Combine.
                n := n.parent
            end if
        end while
    else                                                        ▷ Non-leaf
        Replace n with the max key of the left subtree, which is at position m.
        while m underflow do                                    ▷ Same as leaf deletion.
            if Can be rotated then
                Rotate.
                Break.
            else
                Combine.
                m := m.parent
            end if
        end while
    end if
end function
```

13. **Theorem (162)** Red-black tree：

- BST.
- root 和 NIL 皆黑色，紅色節點的兩個子節點必定是黑色。
- root 到不同樹葉節點路徑上皆有相同數量黑色節點。

- 若一 Red-black tree 高度為 h 且節點數為 n 的，則

$$h \leq 2\log(n+1) \tag{6}$$

```
function InsertRedBlackTree(RedBlackTree s, Element x)
    x.color := red.
    InsertBST(x).
    while x.parent.color = red do
        if x.parent is left child then
            if x.parent.sibling.color = red then
                x.parent.color := black
                x.parent.sibling.color := black
                x.parent.parent.color := red
                x := x.parent.parent
            else if x is right child then
                x := x.parent
                LeftRotate(s, x)
            else
                x.parent.color := black
                x.parent.parent.color := red
                RightRotate(s, x.parent.parent)
            end if
        else
            Similar to the process above, just change LeftRotate and RightRotate.
        end if
    end while
    s.root := black
end function
```

```
function DeleteRedBlackTree(RedBlackTree s, Element x)
    org − color := x.color
    if x is leaf then
        Set link from x.parent to x as NIL.
    else if x has 1 child then
        Replace x with its child.
    else                                                    ▷ x has 2 children
        Replace x with largest in left subtree or smallest in right subtree.
    end if
    if org − color = black then
        DeleteFixRedBlackTree(s, x)
    end if
end function
```

```
function DeleteFixRedBlackTree(RedBlackTree s, Element x)
    while x ≠ s.root ∧ x.color = black do
        if x is left child then
            w := x.sibling
            if w.color = red then
                w.color := black
                x.parent.color := red
                LeftRotate(s, x.parent)
                w := x.sibling
            else if w.lchild.color = black ∧ w.rchild.color = black then
                w.color := red
                x := x.parent
            else if w.rchild.color = black then
                w.lchild.color := black
                w.color := red
                RightRotate(s, w)
                w := x.sibling
            else
                w.color := x.parent.color
                x.parent.color := black
                w.rchild.color := black
                LeftRotate(s, x.parent)
                x = s.root
            end if
        else
            Similar to the process above, just change LeftRotate and RightRotate.
        end if
    end while
    x.color = black
end function
```

14. **Theorem (170, 171)** Splay Tree：

- BST.
- 每一次 splay 運算都將 splay 起點最終變為 root。

```
function InsertSplayTree(SplayTree s, Element x)
    n := InsertBST(x)
    Splay(n)
end function
```

function SearchSplayTree(SplayTree s, Element x)
    $n := \text{SearchBST}(x)$
    $\text{Splay}(n)$
end function

function DeleteSplayTree(SplayTree s, Element x)
    $n := \text{SearchBST}(x)$
    $\text{Splay}(n)$
    Remove n and get its left and right subtrees T_L and T_R.
    $max := \text{FindMaxBST}(T_L)$
    $\text{Splay}(max)$
    $max.rchild := T_R$
end function

AVL tree, B tree, and Splay tree		
	AVL tree and B tree	Splay tree
Worst	$O(\log n)$	$O(n)$
Amortized	$O(\log n)$	$O(\log n)$

15. **Theorem (172, 173, 174)** Leftist heap：

- $$shortest(x) = \begin{cases} 0 & ,x \text{ is external node} \\ 1 + \min\{shortest(x.lchild), shortest(x.rchild)\} & ,x \text{ is internal node} \end{cases} \quad (7)$$
- $\forall n \in$ leftist tree，$shortest(n.lchild) \geq shortest(n.rchild)$。
- Min(Max)-leftist heap：leftist tree and min(max)-tree.
- 一 n 個節點的 leftist tree，root 距離 $\leq \log(n+1) - 1$。

function MergeLeftistHeap(LeftistHeap s, t)
    if $s.data < t.data$ **then**
        $\text{MergeLeftistHeap}(s.rchild, t)$
    else
        $\text{MergeLeftistHeap}(t.rchild, s)$
    end if
    Check the *shortest* value of each node, if breaking the rule, swap the node and its sibling.
end function

```
function DeltetMinLeftistHeap(LeftistHeap s)
    Remove root and get its left and right subtrees T_L and T_R.
    MergeLeftistHeap(T_L, T_R)
end function
```

```
function InsertLeftistHeap(LeftistHeap s, Element x)
    Let x be a tree n.
    MergeLeftistHeap(s, n)
end function
```

<table>
<tr><th colspan="3">Heap and Leftist heap</th></tr>
<tr><th>Operation</th><th>Heap</th><th>Leftist heap</th></tr>
<tr><td>Insert x</td><td colspan="2" rowspan="2">$O(\log n)$</td></tr>
<tr><td>Delete min</td></tr>
<tr><td>Merge one or two heaps</td><td>$O(n)$</td><td>$O(\log n)$</td></tr>
</table>

16. **Theorem (174, 175, 176)** Binomial heap：

- root level 為 0。
- B_k 為高度為 k 的 binomial tree，由兩個高度 $k-1$ 的 B_{k-1} 組成，其中 B_0 只有 root 一個節點。
- B_k 第 i level 的節點數為 $\binom{k}{i}$，總共 2^k 個節點。
- Binomial heap：一組 binomial tree 且皆為 min-tree 組成的 forest。

```
function MergeBinomialHeap(BinomialHeap s, t)
    Merge all trees with same height recursively by choosing the smaller root as new root.
end function
```

```
function DeleteMinBinomialHeap(BinomialHeap s)
    Delete the smallest root from tree p and get new trees u, and the others are q.
    MergeBinomialHeap(q, u)
end function
```

```
function InsertBinomialHeap(BinomialHeap s, Element x)
    Let x be a tree n.
    MergeBinomialHeap(s, n)
end function
```

17. **Theorem (179)** Fibonacci heap：

- Binomial heap 的 superset，又稱 Extended binomial heap。
- 比 binomial heap 多 DeleteNode 和 DecreaseKey。
- 與 binomial heap 差異：
  - insert 與 delete 皆不合併。
  - 所有節點用一個 double-linked circular linked list 連結起來，同時紀錄左右兄弟、父節點。
  - DecreaseKey 若使該節點小於其父節點，則將該子樹獨立出來。

Binomial heap and Fibonacci heap		
Operation	Binomial heap	Fibonacci heap
Insert x	$O(\log n), O(1)^*$	$O(1)^*$
Delete x/min	$O(\log n)$	$O(\log n)^*$
Merge	$O(\log n)$	$O(1)^*$
Decrease key	$O(\log n)$	$O(1)^*$
Find min	$O(\log n)$	$O(1)$
Remark	Find min can be down to $O(1)$.	Decrease key is faster than binomial heap

18. **Theorem (195)** Quick sorting：

```
function QuickSort(Array A, index p, r)        ▷ Sorting from A[p] to A[r]
    if p < r then
        q := Partition(A, p, r)
        QuickSort(A, p, q − 1)
        QuickSort(A, q + 1, r)
    end if
end function
```

```
function Partition(Array A, index p, r)
    x := A[r]                                        ▷ Pivot.
    i := p − 1
    for j := p to r − 1 do
        if A[j] ≤ x then
            i := i + 1
            Swap(A[i], A[j])
        end if
    end for
    Swap(A[r], A[i + 1])
    return i + 1
end function
```

19. **Theorem (221, 223, 224, 225, 227)** Hashing：

- Uniform hashing function：使資料量 n 大致平均分布在所有 B 個 bucket，每個 bucket 內資料量大約 $\frac{n}{B} = \alpha$，則
  - 成功搜尋平均比較次數為 $\frac{1+2+\cdots+\alpha}{\alpha} = \frac{1+\alpha}{2} \approx 1 + \frac{\alpha}{2}$。
  - 失敗搜尋平均比較次數為 α。
- Linear probing：易發生 primary clustering problem，即相同 hashing address 的 data 易儲存在附近，增加 searching time。
- Quadratic probing：Overflow 發生時，改變 hashing function 為

$$(H(x) \pm i^2) \,\%\, B,\ \forall i = 1, 2, \cdots, \lceil \frac{B-1}{2} \rceil \tag{8}$$

  其中 B 為 bucket 數，i 找到有空 bucket 或是所有格皆滿為止。解決 primary clustering problem，但易發生 secondary clustering problem，即相同 hashing address 的 data overflow probe 的位置距規律性，增加 searching time。

- Double hashing：Overflow 發生時，改變 hashing function 為

$$\begin{aligned} &(H(x) + i \times H'(x)) \,\%\, B,\ \forall i = 1, 2, \cdots \\ &H'(x) = R - (x \,\%\, R),\ R \text{ is prime} \end{aligned} \tag{9}$$

  其中 B 為 bucket 數，i 找到有空 bucket 或是所有格皆滿為止。解決 secondary clustering problem，但不保證 table 充分利用。

20. **Theorem (234, 236, 240)** Graph：

- Adjacency multilists：每個節點儲存 v_i，v_j，$link\_for\_v_i$ 指向 v_i 下一個相鄰的點所在的節點，$link\_for\_v_j$ 指向 v_j 下一個相鄰的點所在的節點。

Adjacency matrix and Adjacency lists		
Operation	Adjacency matrix	Adjacency lists
Lots of vertices		$\surd$
# of edges or if it's connected, etc.		$\surd$ $(O(\lvert V\rvert + \lvert E\rvert))$

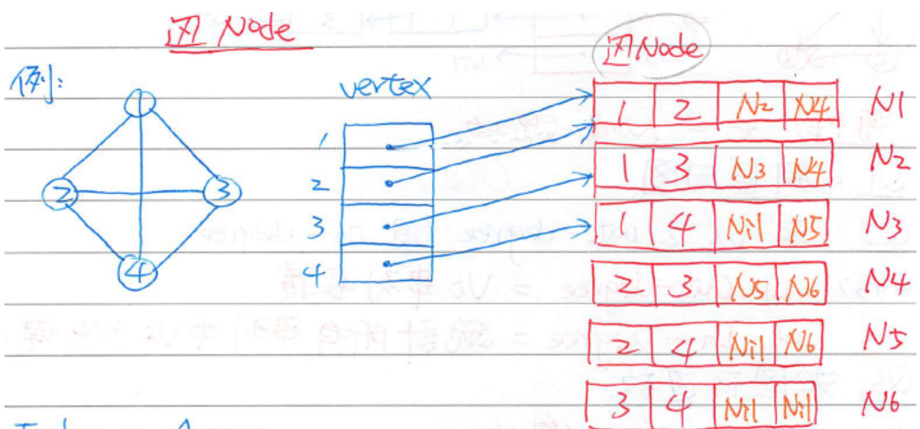

Figure 1: Example for adjacency multilists.

21. **Theorem (257)** 尋找 articulation point：

- dfn 為 DFS number。
-

$$low(v) = \begin{cases} dfn(v) \\ \min\{low(u) | u \text{ is child of } v\} \\ dfn(u), u \text{ is descendant of } v\text{, which is reachable by a back edge} \end{cases} ,\forall v \in G \tag{10}$$

- 若 root 有 ≥ 2 子節點，則 root 為 articulation point；非 root 節點 u，若 $\exists v$ 為 u 子節點，且 $low(v) \geq dfn(u)$，則 u 為 articulation point。

ZHEJIANG UNIVERSITY 1897

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3 edition, 2009.

[2] wjungle@ptt. 資料結構 @tkb 筆記. https://drive.google.com/file/d/0B8-2o6L73Q2VeFpGejlYRk1WeFk/view?usp=sharing, 2017.