date . . No.

佇列 Queue

ex: 讀入字串、辨識迴文 (palindrome)

- New item enter at the back, or rear, of the queue
  Items leave from the front of the queue
- First-in, first-out (FIFO) ex: 排隊

operation
- Create 建構
- Destroy 解構
- Determine whether is empty 是否為空
- Add a new item 新增
- Remove that was added earlist 移除
- Retrieve that was added earlist 檢取

isEmpty()
enqueue()
dequeue()
getFront(out....)
dequeue(out...) 檢取後移除

ex: 將字串轉為數值 (所寫已在 aQueue)

```
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)
}
```

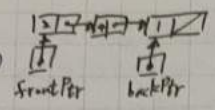
移除

- 迴文 (佇列前端 v.s 堆疊頂端)

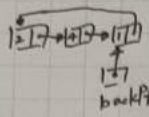
ex: 辨識迴文

```
isPal (in str: string): boolen
aQuene.createQuene()
aStack.createStack()
for (the next character ch in str) {
    aQuene.enqueue(ch)
    aStack.push(ch)
}
charEqual = true
while (!aQueue.isEmpty() &&
       charEqual) {
  aQueue.getFront(&ront)
  aStack.getTop(top)
  if (&ront == top) {
    aQueue.dequeue()
    aStack.pop()
  } else charEqual = FALSE
}
```

• linear linked
前端、後端 (reference)



• circular linked
2有後端 (reference)



▼環狀佇列[新增]

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

▼環狀佇列[移除]

```
Void Queue::dequeue() throw (QueueException){
    if (isEmpty())
        throw ... ;
    else {
        QueueNode *tempPtr = backPtr->next;
        if (backPtr == backPtr->next)
            backPtr = NULL;
        else backPtr->next = tempPtr->next;
        tempPtr->next = NULL;
        delete tempPtr;
    }
}
```

• 環狀佇列的困難
[兩個不同情況(全空、全滿),
條件一樣]

1. 計數器: count
2. 多宣告一個空間: MAX_QUEUE+1
3. 設定旗標: 是否全滿 (full or empty)
[isFull]

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

▼Position-oriented ADTs 位置導向
- List — All positions can be accessed
- Stack
- Queue

] only the end positions can be accessed

• enqueue()
aList.insert(aList.getLength()+1, newItem) — 新增

• dequeue()
aList.remove(1) — 移除

• getFront(queueFront)
aList.retrieve(1, queueFront) — 擷取

▼三種架構
- single teller / single queue
- Multiple tellers / single queue
- Multiple tellers / multiple queues

• Simulation 模擬 (modeling the behavior) 行為模型
目標— 統計、預測

▼ A time-driven simulation
• One time unit 時間驅動

▼ An event-driven simulation
• time of next event 事件驅動

• Arrival event 輸入決定時間

• Departure event 模擬決定時間

演算法

時間效率 空間效率
· Time efficiency, space efficiency

• Big O

• 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 $f(n)$

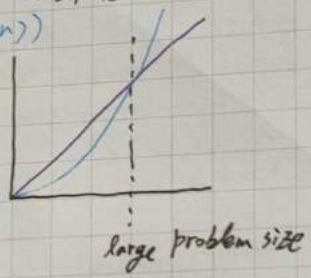
· 計算動作次數

```
ex: -n=10
    -n=100
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
```

$\Sigma(t*5*a)$ for $a=1$ to n
$\Rightarrow 5*t*n*(n+1)/2$
$\Rightarrow t*(2.5n^2+2.5n)$
$\Rightarrow n=10$, Ans $=275t$
$n=100$, Ans $=25250t$

(growth-rate functions)
• 成長速率 [problem size]
1. 指數成長 $[n^2]$
2. 線性成長 $[n]$

• 排位階 (order $f(n)$)
• $O(n^2)$ - order n^2
• $O(n)$ - order n



ex: $2.5n^2-2.5*n$ is $O(?)$, $k=?$ $n_0=?$
$\forall n \geq n_0, (2.5n^2-2.5*n) \leq K*f(n)$
$\forall n \geq 10, (2.5n^2-2.5*n) \leq 1*n^{10}$
$\forall n \geq n_0, (2.5n^2-2.5*n) \leq k*n^2$
$\forall n \geq 0, (2.5n^2-2.5*n) \leq 3*n^2$

(1) $O(1)$ constant time 常數 (最快)
(2) $O(\log_2 n)$ logarithmic time 對數
(3) $O(n)$ linear time 線性
(4) $O(n \log_2 n)$
(5) $O(n^2)$ quadratic time 平方
(6) $O(n^3)$ cubic time 立方
(7) $O(2^n)$ exponential time 指數

[忽略低位階、忽略常數]

- $O(n^3+3n)$ is $O(n^3)$
- $O(5(6(n)))$ $= O(5(n))$

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

★ Worst-case (最多) maximum
★ Average-case (平均) average
Best-case (最少) minimum

• Sequential search 循序搜尋
▲ 看完每一個資料
worst $O(n)$
average $O(n)$
best $O(1)$

• Binary search 二天搜尋
▲ 把 array 分成一半
worst $O(\log_2 n)$
average $O(\log_2 n)$
best $O(1)$

• Sort key 排序鍵
The part of a data item that we consider when sorting a data collection

• internal sort 內部排序

• external sort 外部排序 (資訊量大)

Stable Sort vs. Unstable sort

[最小/大資料在正確位置上]
bubble worst $O(n^2)$ Best $O(n)$
insertion worst $O(n^2)$ Best $O(n)$
merge worst $O(n*\log_2 n)$ Average $O(n*\log_2 n)$
radix
[相同值維持不變的排序]

selection [選後, 做交換] worst $O(n^2)$ Best $O(n^2)$
quick [樞紐, 先分組, 後遞迴呼叫]
heap
worst $O(n^2)$ Average $O(n\log n)$

shell sort → [跟前面做比較, 插入]
[先分組, 各自排序, 後合併]
$O(n)$ moves
[分解取部分值, 分配至數據容器]

• 筆資料太大時, 搬動費時

- Bubble Sort

1+(n-1) assignment, n comparisons

In each pass 1+(n-pass) assignment, n-pass+1 comparisons

```
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 each array item, 1 comparison

▲ comparisons:

$n + \sum_{pass=1 \dots n-1} (n-pass+1) + \sum_{pass=1 \dots n-1} (n-pass)$

$= n + 2[n*(n-1) - n*(n-1)/2] + (n-1) = n^2 + n - 1 \rightarrow O(n^2)$

▲ major comparison 指比較運算 => 比較次數

$(n-1) + (n-2) + (n-3) \dots + 1$

$= n*(n-1)/2 = 0.5n^2 - 0.5n \rightarrow O(n^2)$

-
```
Void mergeSort (DataType theArray[], int first, int last){
    if (first < last){
        int mid = (first + last)/2;
        mergeSort(theArray , first, mid );
        mergeSort (theArray , mid+1, last );   // 先遞迴呼叫(分組)
        merge (theArray, first, mid, last);    // 後合併
    }
}
```

$3*n - 2^i$ $O(\log_2 n)$ levels

```
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 <=last 2);++index)
        if ( theArray[first1] < theArray[first2] ) {
            thempArray[index]= theArray [first1];
            ++first1;
        }
        else{
            tempArray[index] = theArray[first2];
            ++ first2;
        }
    for( ; first1 <= last 1; ++ first1, ++ index )
        tempArray[index] = theArray[first1];
    for( ; first2 <= last 2; ++first2 , ++ index )
        tempArray [index] = theArray[first 2];
    for (index = first ;index <= last ; ++ index)
        theArray [index] = tempArray [index];   // major Operation O(n)
}
```

- QuickSort
  - pivot (樞紐、軸)
  - items < pivot
  - items >= pivot
  - pivot is now in corrected sorted position
- 先依軸分組，後遞迴呼叫

```
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);
    }
}
```

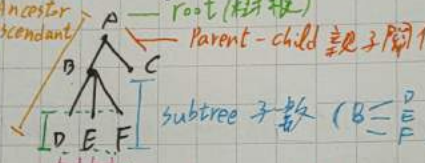
- Radix Sort
- Base 基數

LSD (Least Significant Digit)
依最右側數字分組、串接
MSD (Most Significant Digit)

二元樹 (Binary Trees)

1. position-oriented 位置導向
   ex: list, stack, queue, binary tree 超強關係
2. value-oriented 內容導向
   ex: sorted list, binary search tree
   (競爭較小)

- Tree — nodes 節點 ex: A, B, C, D, E, F
  - edge 連結 (link) ex: A-B, A-C, B-D...

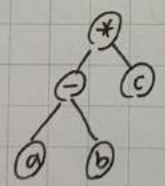


Leaf 葉節點 (no child)
- sibling 兄弟節點 (common parent)

- A is ancestor of node D
  D is descendant of node A

- Binary Tree
  - left subtree and right subtree
    左子樹 右子樹

ex: (a-b)*c



樹高遞迴定義

- If T is empty, its height is 0
- If T is not empty,
  height(T) = 1 + max{height(T_L), height(T_R)}

▸ 樹高

↳ Height 3
(longest path (最長路徑) 需多久)
[越長效率越差]

← Level of a node in a tree (階層)

(Maximum level == Height (T), if tree is not empty)
最大階層 == 樹高

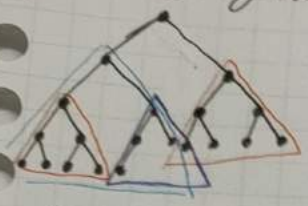
- Full Binary Trees 完全樹

▲ 每個 level < Hight 的節點，都有2個子節點 (填滿)
▲ 實用性不高 (數量固定)

- Complete Binary Trees 完整數

▲ 盡可能填滿，盡量往左邊靠 (最底層)
▲ levels <= h-2，都有2個子節點
level h-1，有2子節點的話，靠左塞滿
level h-1，有1子節點的話，靠左塞

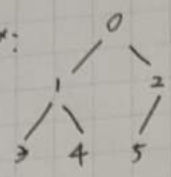
• Balanced Binary Trees 平衡二叉樹



• 左右兩邊樹高差距不超過一 (no more than 1)

Balanced ⊆ Complete ⊆ Full

ex:



- leftchild = 2*parent+1
- rightchild = 2*parent+2
- parent = (child-1)/2

▶ 保持完整二元樹

```
class TreeNode {
  private:
    TreeItemType item;
    int leftchild;
    int rightchild;
};
TreeNode tree[MAX_NODES];
int root;
int tree; // 開置樹根
```

// -1 表示 null

- array-based representation 陣列表示法
- pointer-based representation 指標表示法

• 走過一遍

```
traverse (in binTree: BinaryTree)
    is (binTree is notempty) {
  traverse(Left subtree of binTree's root)
  traverse(Right subtree of binTree's root)
}
```

- preorder 前序 →
- inorder 中序 →
- postorder 後序 →

• successor 後繼者

• 透過兩條件，可還原"唯一"二元樹

性質

- 一個二元樹最多 2^h-1 個資料
- 完整二元樹最小樹高

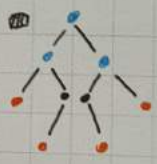
$n \le 2^h-1 \rightarrow \log_2(n+1) \le \log_2(2^h)$

$\rightarrow h \ge \log_2(n+1) \rightarrow h = (\log_2(n+1))$

最大樹高：

$(2^{h-1}-1)+1 \le n \rightarrow \log_2(2^{h-1}) \le \log_2(n)$

$\rightarrow h \le \log_2(n)+1 \rightarrow h = (\log_2(n))+1$



N_2：3個有兩小孩的Nodes

N_0：4個葉節點

$N_0 = N_2 + 1$

B：8個邊 (edges)

$B = |E| = 2*N_2 + 1*N_1$

▲ |leaf nodes| - |internal node| = 1

• 二元搜尋樹

(Binary Search tree)

• 左 < 才 < 右

Average case : O(log n)

Worst case : O(n)

• 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	Worst
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

• 刪掉 nodes

→ 當 nodes 有兩 childs 時 child ≤ 1

↳ Locate another node M that is easier to be deleted (leftmost node)

• 儲存後還原

original → 原來的樹 [preorder]

balance → 平衡樹 [inorder]

▼把任意樹存成二元樹 [空間較有效率]

最左側小孩與右邊的兄弟