No. :

Date : ..../..../....

Subject : 堆疊

檢查成對括弧．辨識語言．代數運算式求解、搜尋-條路徑

isEmpty() : boolean {query} 是否為空

push (in newItem : StackItemType) 新增一筆
    throw StackException

pop() throw StackException 移除最近一筆

getTop (out stackTop : StackItem Type) {query} 擷取最近一筆
    throw StackException

pop (out stackTop : StackItemType) 擷取後移除最近一筆
    throw StackException

5-01 堆疊原理

注1.

1) 是否成對 { }, [ ], ( ) 是否成對

balancedSoFar = true

while (not end of string && balancedSoFar)

{ Read next character ch

if (ch is '{') if (ch is '{') || (ch is '[') || (ch is '(')

aStack . push (ch)

else if (ch is '}') else if (ch is '}') || (ch is ']') || (ch is ')')

if (! aStack . isEmpty())

aStack . pop() aStack . pop (StackItem)
if (! isMatch (stackItem, ch))
balancedSoFar = false

else balancedSoFar = false } // end while

if (balancedSoFar && aStack . isEmpty()) return true

else return false

No. :
Date : / /
Subject :

```
count = 0 ;
while ( not end of string && count >= 0)
  { Read next character ch
    switch (ch)
    { case '{':     count ++ ;
                    break ;
      case '}':     count -- ;
    }
  } // end while
  if (count == 0)    return true
  else               return false
```

(5-02) 以堆疊檢查是否成對括弧

迴文 abc$cba

```
aStack.createStack()
ch = first character
while(ch is not '$')
{ aStack.push(ch)
  ch = next character
} // end while
inLanguage = true
```

```
while( not end of string && inLanguage)
{ if (!aStack.isEmpty())
  { aStack.pop( stackTop)
    ch = next character
    if (stackTop != ch)
        inLanguage = false }
  else inLanguage = false
} // end while
if ( inLanguage && aStack.isEmpty())
    return tre
else return false
```

(5-03) 以堆疊判斷字串

萬國牌

No. :

Date : ..../..../....

Subject : Array-based Implementation

```
const int MAX_STACK = maximum_size_of_stack;
typedef desired_type_of_stack_item StackItemType;
class Stack
{ public:
    Stack();
    bool isEmpty();
    void push(const StackItemType& newItem);
    void pop();
    void getTop(StackItemType& stackTop);
    void pop(StackItemType& stackTop);
  private:
    StackItemType items[MAX_STACK];   堆疊內容
    int           top;                堆疊頂端
};  // end Stack
Stack::Stack() : top(-1)   建構
{
}  // end default constructor
bool Stack::isEmpty() const.   是否為空
{
  return top < 0;
}
```

No. :

Subject : ..... Date : ...../...../.....

```
void Stack :: push ( const StackItemType & newItem )   新增
  { if ( top < MAX_STACK-1)
      { top++;
        items [top] = newItem;
      }
  }

void Stack :: pop()   移除
{ if( ! isEmpty() )
        top--;
}

void Stack :: getTop( StackItemType & stackTop )   擷取
  { if ( ! isEmpty() )
        stackTop = items [ top ];
  }

void Stack :: pop ( StackItemType & stackTop )   擷取後移除
  { if ( ! isEmpty() )
      { stackTop = items [ top ];
        top--;
      }
  }
```

No. :

Date : ..../..../....

Subject : .....

Pointer-based Implementation

之前的筆記有一些的觀念。

List-based Implementation

```
class Stack
{ public:
      Stack();
      Stack(const Stack & aStack);
      ~Stack();
      bool isEmpty() const;
      void push(const StackItemType & newItem);
      void pop();
      void pop(StackItemType & stackTop);
      void getTop(StackItemType & stackTop);
  private:
      List aList;
};
bool Stack::isEmpty() const
{ return aList.isEmpty();
}
void Stack::push(const StackItemType & newItem)
{ try
  { aList.insert(1, newItem);
  }
```

Subject : ..... Date : ...../...../.....

```
    catch ( ListException e )
        { ... }
    catch ( ListIndexOutOfRangeException e )
        { ... }
}
void Stack :: getTop ( StackItemType & stackTop )
{ try
    { aList.retrieve ( 1, stackTop );
      aList.remove(1);
    }
    catch ...
}
```

5-04 實作堆疊

後序式求解 → 簡化假設 { 一元運算子 / 指數運算子 / 小寫字母

中序式轉後序式

operand (運算元)　　operator (運算子)　　5-06 以堆疊解算運算式

後序式求解

前序式求解 (倒過來做)

5-07 後序式求解

Subject : .....

No. :

Date : ...../...../.....

中序轉後序

- 放 operand 在一個空字串裡
- Push (onto a stack
- precedence higher than or equal to 7= 5-08 中序轉後序的程序

依序寫出過程中的堆疊內容 5-09 中序轉後序的範例

Non-recursive Solution (搜尋兩點間的路徑)

```
bool Map::isPath(int originCity,
    int destination)
{ Stack aStack;  堆疊：目前路線
  int topCity, nextCity;
  bool success;
  unvisitAll(); // clear marks
  aStack.push(originCity);
  markVisited(originCity); 初始沒有起點
  aStack.getTop(topCity);
  while(!aStack.isEmpty() &&
        (topCity != destination))
  { success = getNextCity(topCity, nextCity);
    if(!success) aStack.pop(); // back track
    else
    { aStack.push(nextCity);
      markVisited(nextCity);
    } // end if-else
    if(!aStack.isEmpty())
      aStack.getTop(topCity);
  } // end while
  if(aStack.isEmpty())
    return false; // no path exists
  else return true;
} // end isPath
```

5-10 以堆疊搜尋兩點間的路徑

No. :
Date : ...../...../.....

Subject : 佇列

佇列三排隊

是否為空：isEmpty() : boolean {query}

新增：enqueue (in newItem : QueueItemType) throw QueueException

移除：dequeue () throw QueueException

擷取：getFront(out queueFront : QueueItemType)
{query} throw QueueException

擷取後移除：dequeue (out queueFront : QueueItemType)
throw QueueExeption

aQueue.createQueue() 建構
while (not end of line)
aQueue.enqueue (ch) 新增

將字串轉為數值

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()) 是否為空
done = TRUE
else aQueue.dequeue (ch) 移除
}

Subject : .....

No. :

Date : ...../...../.....

有小數點

```
if ( ! done and ch = '.' )
{ aQueue.dequeue(ch)
  p = 0
  while( ! done and ch is digit)
  { n = n x 10 + integer of ch
    p++
    if ( aQueue.isEmpty() )
        done = TRUE
    else aQueue.dequeue(ch)
  }
  n = n x (0.1)^p
}
```

(b-01 作例題理

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

Subject : .....

Date : ...../...../.....

```
while( !aQueue.isEmpty() && charEqual )
{ aQueue.dequeue(front)
  aStack.pop(top)
  if( front != top )
    charEqual = False
}
```

b-02 以符列解謎迴文

```
{ aQueue.getFront(front)
  aStack.getTop(top)
  if( front = top )
  { aQueue.dequeue()
    aStack.pop()
  }
  else charEqual = False
}
```

linked list 有 front 和 back

circular linked list 只有 back (b-03) 以指標實作符列 (b-04) 以指標實作環狀符列

Insert

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

Delete

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

空的條件：count = 0

全滿的條件：count = MAX_QUEUE

多宣告一個空間：MAX_QUEUE+1

是否全滿：isFull

Subject : ..... Date : ...../...../.....

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

移除：dequeue() : aList.remove(1)

擷取：getFront(queueFront) : aList.retrieve(1, queueFront)

6-05 以陣列實作佇列ADT

Simulate() 事件驅動

```
Create an empty bankQueue; // represent the bank line
Create an empty eventList; // keep the future events
Get the earliest 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();
}
```

6-07 以事件驅動模擬

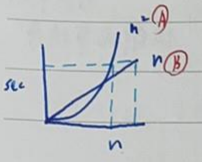
No. :
Date : ...../...../.....

Subject : 演算法效率

空間效率
時間效率

實作
電腦
資料

1-01 演算法的基本觀念



A：指數成長 較慢
B：線性成長 較快

1-02 大O表示法

1. O(1) 最好
2. $O(\log_2 n)$
3. O(n)
4. $O(n \log_2 n)$
5. $O(n^2)$
6. $O(n^3)$
7. $O(2^n)$ 最差

1-03 演算法複雜度分析

Stable Sort vs. Unstable sort

bubble 氣泡
insertion 插入
merge
radix

selection 選擇
quicle
heap

1-04 排序演算法的效率

No. :

Date : ...../...../.....

Subject : .....

```
void bubbleSort(int A[], int n){
    bool sorted = FALSE;
    for(pass=1; pass<n && !sorted; ++pass){      第幾回合
        sorted = TRUE;
        for(int index=0; index<n-pass; index++){      第幾筆資料
            if(A[index] > A[index+1])      相鄰兩筆排序
                swap(A[index], A[index+1]);
                sorted = FALSE
                >= not stable
        }
    }
}
```

Comparisons :

$(n)+(\Sigma_{pass=1\ldots n-1}(n-pass+1))+\Sigma_{pass=1\ldots n-1}(n-pass)$

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

精心比較次數：

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

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

1-12 氣泡排序法的複雜度分析

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

Subject : .....

No. :

Date : ...../...../.....

```
int indexOfLargest(int A[], int size){        找出最大的位置(预设)
    int indexSoFar = 0;
    for (index = 1; index < size; ++index)
        if (A[indexSoFar] < A[index])          发现较大值
            indexSoFar = index;
    return indexSoFar;
}
```

Comparisons: $\Sigma_{size=n\dots 2}(size-1)$

$= (n-1)+(n-2)+\dots+1 = n(n-1)/2 \rightarrow O(n^2)$ (7-13) 选择排序法的复杂度分析

```
void insertionSort(int A[], int n){
    for (unsorted = 1; unsorted < n; ++unsorted){      n-1个回合
        int loc = unsorted, nextItem = A[unsorted];
        for (; (loc > 0) && (A[loc-1] > nextItem); --loc)
            A[loc] = A[loc-1];                逐一往后搬动
        A[loc] = nextItem;                    >= not stable 直到 nextItem的位置
    }
}
```

(7-14) 插入排序法的复杂度分析

Subject : .....

No. :

Date : ...../...../.....

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

it is not stable (1-15 希爾排序:5)

MergeSort

先：分組

各組排序

後：合併

```
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);
        merge (theArray, first, mid, last); 3xn-1
    }
}
```

No. :
Date : ...../...../.....
Subject : .....

```
void merge1 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;
        }
        else {
            tempArray[index] = theArray[first2];
            ++first2;
        }
```

< = stable

Q-16 合併排序

Quicksort

pivot 樞紐、軸

先：分組（軸的位置）

items < pivot

items >= pivot

Pivot is now in correct sorted position

後：遞迴呼叫

Subject : ..... Date : ...../...../.....

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

swap(A[first], A[lastS1]); ++lastS1;
pivotIndex = 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);
    }
}
```

7-11 快速排序

QuickSort : it is not stable 7-12 快速排序的複雜度分析

Double A

Subject : .....

No. :

Date : ...../...../.....

Radix Sort：基數排列 (1-2) 基數排序

MSD (Most Significant Digit)：排序時最重要的數

先：分組

後：串接

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

void radixSort(int A[], int first, int last){ ...
    for(int base=1; (maxData/base)>0; base*=10){
        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];
    }
}
```

Double A

Subject : .....

No. :

Date : ...../...../.....

```
void radixSort (int A[], int first, int last) {
    int temp[MAX_D][MAX_SIZE], maxData;
    int counter[10] = {0}, i, j;
    for ( maxData = A[first], i= first+1; i<last; i++)
        if (maxData < A[i])  maxData = A[i];
    for (int base =1; (maxData / base) >0; base *=10){
        for (i=first; i<=last; i++) {
            int LSD= (A[i] / base) % 10;
            temp[LSD][counter[LSD]]=A[i];
            counter[LSD]++;
        }
    }
```

LSD (分組)

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
Quicksort	n^2	$n * \log n$
Radix sort	n	n

7-22 基數排序的演算法設計

Double A

Subject : 樹

Date : ...../...../.....

位置導向：list, stack, queue, binary tree

內容導向：sorted list, binary search tree

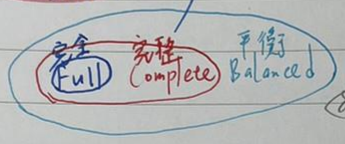
Parent - child 親子關係

Ancestor - descendant 祖孫關係

Subtree 子樹

8-01 樹狀結構簡介

二元樹 (binary tree)



8-03 平衡樹簡介

```
const int MAX_NODES = 100;
class TreeNode
{ private:
      TreeItemType item;
      int leftChild;
      int rightChild;
};
TreeNode tree[MAX_NODES];
int root; // 樹根
int free; // 閒置串列
```

leftChild = 2 x parent + 1

rightChild = 2 x parent + 2

parent = (child - 1) / 2

8-04 二元樹 ADT

Double A

No. :
Date : ...../...../.....

Subject : .....

最小樹高： $h = \lceil \log_2(n+1) \rceil$

最大樹高： $h = \lfloor \log_2(n) \rfloor + 1$

N_2：有2個 children

N_0：leaves

$N_0 = N_2 + 1$

邊 $= 2 \times N_2 + 1 \times N_1$

點 $=$ 邊 $+ 1$

(8-05) = 元樹的特性

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

```
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;
     }
     aStack.pop( treePtr);
  }
}
```

8-07 非遞迴走訪二元樹

Subject : 個人的學習心得

No. :

Date : ...../...../.....

雖然不是實體課，但是透過這4單元的影片使我更了解每個單元所講述的內容，第5單元使我了解到堆疊是先進後出，而在第6單元使我了解到佇列是先進先出，由單元5和6可以看出這兩個方法的不同，而在單元7我了解到不同方法使用上的效率不同，最後在單元8我了解到樹的概念。從這4單元經過老師的講述，使我更清楚明瞭！