單元5 堆疊

type abcc ← dde ← ← ← ef ← fg 新增移除一筆資料
是否為空

while (not end of line)
{ Read a new character ch
if (ch is not '←')
    Add ch to the ADT
else if (the ADT is not empty)
  Remove the last item from the ADT
else
  Ignore '←'
} // while

□ Operation Contract for the ADT stack

isEmpty(): boolean 是否為空

push(in newItem: StackItemType) 新增一筆

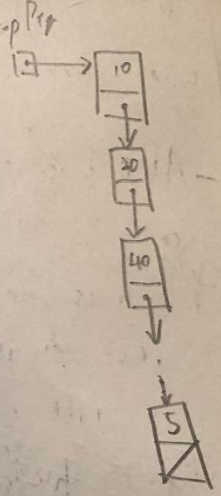
pop() throw StackException 移除最近一筆

getTop(out stakTop: stackItemType) 擷取最近一筆

popout stackTop: stackItemType) 擷取後移除最近一筆

ADT stack – Pointer-based Implementation

```
Void stack::push(const StackItemType & newItem)
{ try
  { StackNode *newPtr = new StackNode;
    newPtr->item = newItem;
    newPtr->next = topPtr
    topPtr = newPtr;
  } // try
} // end push()
```




```
Void stack::pop()
{if(!isEmpty())
 { StackNode *temp = topPtr;
   topPtr = topPtr->next;
   temp->next = NULL;
   delete temp;
 } // if
} // pop()
```

```
Void Stack::getTop(StackItemType & stack
                                    Top)
{ if(!isEmpty)
     StackTop = topPtr->item;
} // getTop()
```

```
Void stack::pop(stackItemType & stackTop)
{if(!isEmpty)
  { getTop();
    pop();
  } // if
} // pop()
```

- Pointer-based implementation is more efficient

- ADT list approach reuses an already implemented class
  - Muck simpler to write
  - Saves programming time

心得：堆疊算是滿直接的存資料方式，也就是先進後出，在某些特別的處理資料就可以利用堆疊來存放資料，尤其是會用到前，中，後序的情況，通常都會使用堆疊。

單元 6 Queue 佇列=排隊

□ ADT queue operations

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

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 = temp->next;
    tempPtr->next = NULL;
    delete tempPtr;
  } // else
} // dequeue()
```

心得：我覺得佇列跟堆疊有點相似，所以在剛學時並不會有太困難的地方，只是需要注意佇列的執行的方式與堆疊相反，佇列是先進先出，主要還是判斷條件的部分需要小心。

單元7 演算法

□ Analysis of algorithms

□ Time efficiency 時間效率
space efficiency 空間效率

□ Difficulties
- Implementation
- Computer
- Dataset

□ Enables comparison between two algorithms
- Algorithm A requires time proportional to n^2
- B to n

1. $O(1)$ 常數
2. $O(\log_2 n)$ 對數
3. $O(n)$ 線性
4. $O(n \log_2 n)$
5. $O(n^2)$ 平方
6. $O(n^3)$ 立方
7. $O(2^n)$ 指數

□ Algorithm B is faster than Algorithm A
- n^2 and n are growth-rate functions
- Algorithm A is $O(n^2)$ - order n^2
- Algorithm B is $O(n)$ - order n
- Big O notation

□ Properties of growth-rate functions
- $O(n^3+3n)$ is $O(n^3)$: ignore low-order terms 忽略低位階
- $O(5f(n)) = O(f(n))$: ignore multiplicative constant 忽略常數
- $O(f(n)) + O(g(n)) = O(f(n)+g(n))$

□ Worst-case analysis 最多
- the maximum time an algorithm require to solve problem of size n

□ Average-case analysis 平均
- the average time 〃

□ Best-case analysis 最小
- the minimum time 〃

□ Sequential search 循序搜尋

□ Look each item, stop when the desired item is found, or the data is end

- Efficiency
Worst : O(n)
average : O(n)
best : O(1)

□ Binary search

- Strategy
  □ divide array in half, determine which half could contain the item, and discard the another

- Efficency
  □ Worst : $O(\log_2 n)$
  best : $O(1)$

□ Simple Sort

氣泡排序：兩兩比較

選擇排序：找到最小的排到前面

插入排序：將 item 插入適當的位置

□ Faster Sort

- Mergesort : 先→分組各自排序
  $O(n)$ 後→迴合併迴排序

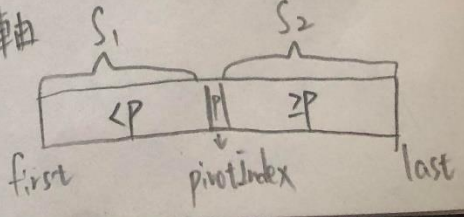
- Quicksort

□ Strategy

- Choose a pivot 樞紐軸

先：分組（軸的位置）

後：遞迴的叫



□ Radix Sort

□ Strategy

- Decompose the sort key by the radix 分解取部分值
  - ㄧ Treats a key as a character string 分配至对應容器
  - ㄧ Repeatedly assign the keys into groups (buckets) according to the i^{th} character

ㄧ得：演算法在程式中扮演非常重要的角色，經常會被使用，对我而言，看到有使用遞迴的演算法經常需要思考一下才能理解，演算法是一項很好用的工具。

單元8 二元樹

□ Trees are composed of nodes and edges 親子關係 祖孫關係
- Parent-child relationship between two nodes
- Ancestor-descendant relationships among nodes

□ Subtree of a tree: Any nodes and its descendants 子樹

□ Ancestor of node B
- A node on path from root to B

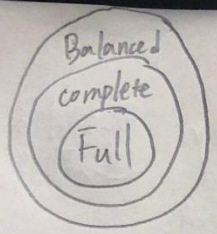
□ Descendant of node B
- // from B to a leaf

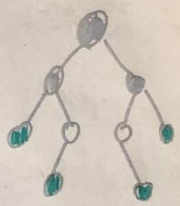
□ Height of Trees 樹高
- Number of nodes along the longest path from root to leaf

Height 3 Height 5 Height 7

最大階層 = 樹高

□ A binary tree of height h is full if 完全樹
- Node at levels < h have two children each





□Notes

N_2 : 3 nodes with two children

N_0 : 4 Leaves

$-N_0 = N_2 + 1$?

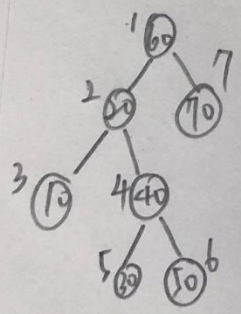
B: 8 Branches (edges)

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

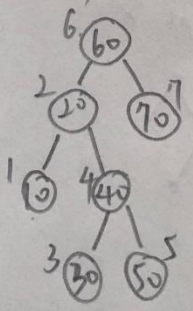
$-N_0 + N_2 + N_1 = |V| = |E| + 1 = 1 * N_2 + 1 * N_1 + 1$

$\Rightarrow N_0 = N_2 + 1$

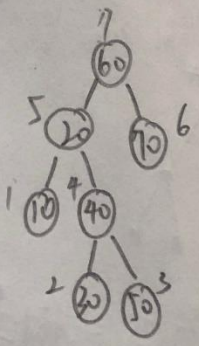
Traversal of a Binary Tree



preorder (60, 20, 10, 40, 30, 50, 70)



Inorder (10, 20, 30, 40, 50, 60, 70)



Postorder (10, 30, 50, 40, 20, 70, 60)

traverse (T
if (r
{
tra
tra
}
}
□ A bin
左<

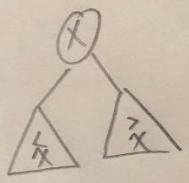
Operation
Retrieval
Insertion
Deletion
Traversal
心得:
3解,
然,
式

```
traverse ( BinaryTree ) {
  if (not empty)
  {
    traverse (Left subtree)
    traverse (Right subtree)
  } // if
} //
```

← 前序
← 中序
← 後序

□ 是 binary search tree

左 < X < 右



Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

心得：在大一時有接觸過二元樹，當時只有表面的了解，但現在又再次接觸後，更加了解到二元樹有其他型態，以及不同的列印順序，我認為這其中困難的是，在寫程式時特別留意不能丟失父節點。