Homework #3

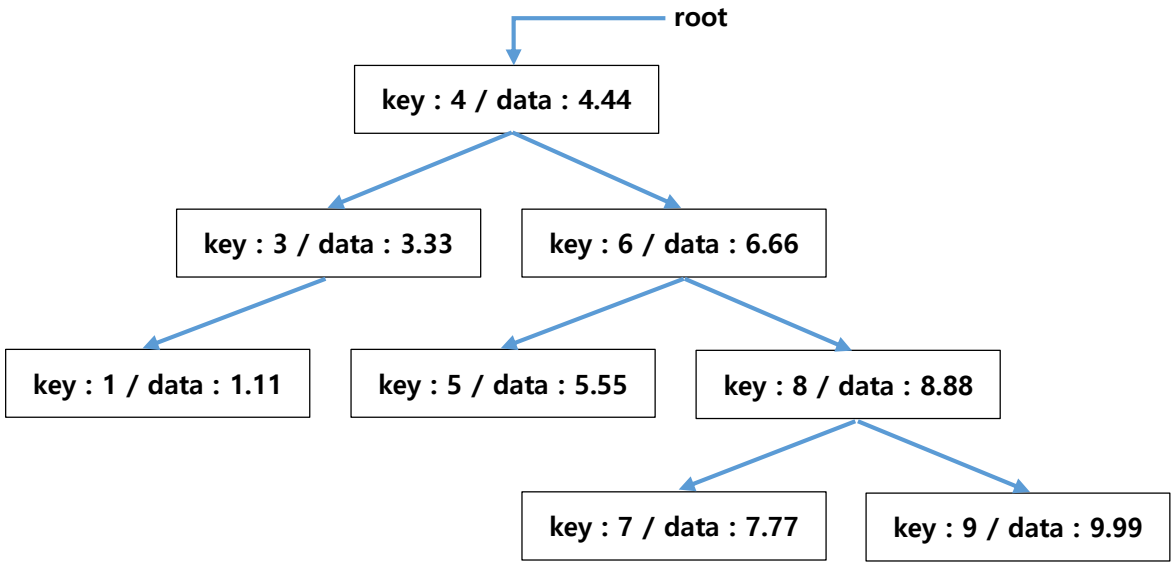
2018. 5. 11 (금) ~ 5. 25 (금) 23:59

*** 마지막 장의 주의사항을 참고할 것.**

P1 (40 점). [BST] Binary Search Tree 를 이용하여 Dictionary 구현을 해보고자 한다. 다음 조건을 만족하도록 BST Class 를 작성하시오. (필요하면 HW2 에서 작성한 Queue class 사용 가능)

제출 파일 : HW3_BST.h (**※ main() 함수는 작성하지 마시오**)

Binary Search Tree 예시)



TreeNode Class

- **Definition**
  - Binary Search Tree의 Node를 나타내는 Class
- **Member Variable (Public으로 정의)**
  - **int key** : Dictionary의 Key
  - **double data** : Dictionary의 Value
  - **TreeNode *left** : Node의 Left Child
  - **TreeNode *right** : Node의 Right Child
- **Constructor (Public으로 정의)**
  - **TreeNode(int k, double d)**
    - **key**를 **k**로 설정하고, **data**를 **d**로 설정
    - **left, right**는 **nullptr**로 초기화

BST Class

- **Definition**
  - Binary Search Tree의 구조의 Dictionary를 나타내는 Class
- **Member Variable (Protected로 정의)**
  - **TreeNode *root** : Root Node
- **Constructor (Public으로 정의)**
  - **BST()**
    - **root**를 **nullptr**로 초기화
- **Member Function (Protected로 정의)**
  - **int getLevelFromLeaf(TreeNode *curr_node);**
    - 가장 먼 Leaf node로부터 curr_node까지의 level 반환

      예) 예시 그림에서 key=7인 node의 경우 0, key=4인 node의 경우 3 반환
  - **void inorder(TreeNode *curr_node)**
    - **curr_node**로부터 Inorder Traversal 수행하여 그 결과 출력
      1. **curr_node**의 Left Sub-tree에 대해 Inorder Traversal 수행
      2. **curr_node**의 **key**와 **data** 출력 (**<key, data>**)
      3. **curr_node**의 Right Sub-tree에 대해 Inorder Traversal 수행
  - **void preorder(TreeNode *curr_node)**
    - **curr_node**로부터 Preorder Traversal 수행하여 그 결과 출력
      1. **curr_node**의 **key**와 **data** 출력 (**<key, data>**)
      2. **curr_node**의 Left Sub-tree에 대해 Preorder Traversal 수행
      3. **curr_node**의 Right Sub-tree에 대해 Preorder Traversal 수행
  - **void postorder(TreeNode *curr_node)**
    - **curr_node**로부터 Postorder Traversal 수행하여 그 결과 출력
      1. **curr_node**의 Left Sub-tree에 대해 Postorder Traversal 수행
      2. **curr_node**의 Right Sub-tree에 대해 Postorder Traversal 수행
      3. **curr_node**의 **key**와 **data** 출력 (**<key, data>**)
  - **void levelorder(TreeNode *curr_node)**
    - **curr_node**를 Root로 하는 Sub-tree의 모든 Node에 대해

      낮은 Level부터 차례대로 Traversal 수행하여 그 결과 출력

- **TreeNode * find(TreeNode *&curr_node, const int& key)**
  - **curr_node**를 Root로 하는 Sub-tree에서 Key 값을 **key**로 가지는 Node 반환
  - Key 값을 **key**로 가지는 Node가 없는 경우 **nullptr** 반환
- **void insert(TreeNode *&curr_node, const int& key, const double& data)**
  - **curr_node**를 Root로 하는 Sub-tree에
    Key 값으로 **key**를, Data 값으로 **data**를 가지는 Node 추가
  - Key 값이 동일한 Node가 이미 있는 경우, Data 값만 **data**로 변경
  - Insert 후 모든 Node에 대해 다음의 Property를 만족해야 함
    - Left Child Node의 Key 값은 Parent Node의 Key 값보다 작아야 함
    - Right Child Node의 Key 값은 Parent Node의 Key 값보다 커야 함
- **TreeNode *& rightMostChild(TreeNode *&curr_node)**
  - **curr_node**를 Root로 하는 Sub-tree에서 가장 오른쪽의 Node 반환
  - 가장 오른쪽의 Node는 Sub-tree에서 Key 값이 가장 큰 Node임
- **void remove(TreeNode *&curr_node, const int& key)**
  - **curr_node**를 Root로 하는 Sub-tree에서 Key 값이 **key**인 Node 삭제
  - 아래에 명세된 **doRemoval()** 함수를 이용하여 구현
- **void doRemoval(TreeNode *&curr_node)**
  - 아래에 명세된 **noChildRemove(), oneChildRemove(), twoChildRemove()**
    함수를 이용하여 구현
- **void noChildRemove(TreeNode *&curr_node)**
  - **curr_node** 단독으로 삭제
- **void oneChildRemove(TreeNode *&curr_node)**
  - 기존의 **curr_node**는 삭제하고 Child Node가 **curr_node**를 대체
- **void twoChildRemove(TreeNode *&curr_node)**
  - **rightMostChild()**를 이용하여 **curr_node**를 삭제하고
    curr_node의 Left Sub-tree에서의 Right Most Child Node가 **curr_node**를 대체

- **Member Function (Public으로 정의)**
  - **bool isEmpty()**
    - BST가 비어있으면 **1**, 비어있지 않으면 **0** 반환
  - **int getHeight()**
    - BST의 Height 반환 (빈 BST의 경우 **-1** 반환)
  - **void inorder()**
    - BST의 Node를 **Inorder Traversal** 순서로 출력
  - **void preorder()**
    - BST의 Node를 **Preorder Traversal** 순서로 출력
  - **void postorder()**
    - BST의 Node를 **Postorder Traversal** 순서로 출력
  - **void levelorder()**
    - BST의 Node를 **Levelorder Traversal** 순서로 출력
  - **void insert(const int& key, const double& data)**
    - BST에 Key 값으로 **key**를, Data 값으로 **data**를 가지는 Node 추가
    - Key 값이 동일한 Node가 이미 있는 경우, Data 값만 **data**로 변경
  - **void remove(const int& key)**
    - BST에서 Key 값이 **key**인 Node 삭제
  - **double find(const int& key)**
    - BST에서 Key 값을 **key**로 가지는 Node의 Data 반환
    - Key 값을 **key**로 가지는 Node가 없는 경우 **-1** 반환

Traversal 예시) 맨 첫 페이지의 Binary Search Tree에 대해

Inorder Traversal from Root :

<1, 1.11> <3, 3.33> <4, 4.44> <5, 5.55> <6, 6.66> <7, 7.77> <8, 8.88> <9, 9.99>

Preorder Traversal from Root :

<4, 4.44> <3, 3.33> <1, 1.11> <6, 6.66> <5, 5.55> <8, 8.88> <7, 7.77> <9, 9.99>

Postorder Traversal from Root :

<1, 1.11> <3, 3.33> <5, 5.55> <7, 7.77> <9, 9.99> <8, 8.88> <6, 6.66> <4, 4.44>

Levelorder Traversal from Root :

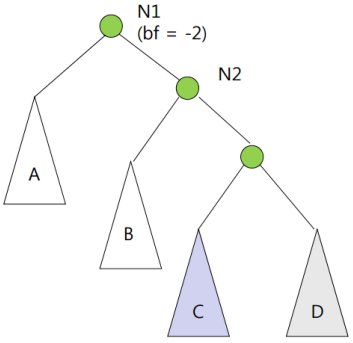
<4, 4.44> <3, 3.33> <6, 6.66> <1, 1.11> <5, 5.55> <8, 8.88> <7, 7.77> <9, 9.99>

P2 (40 점). [AVL Tree] 위에서 구현한 Binary Search Tree 를 이용하여 AVL Tree 를 구현해보고자 한다. 다음 조건을 만족하도록 AVL Class 를 작성하시오.

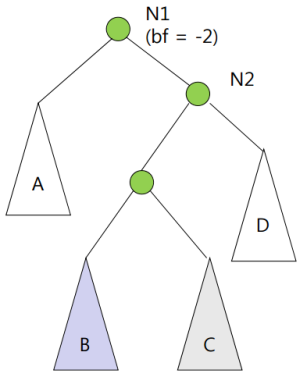
제출 파일 : HW3_AVL.h, HW3_AVL.cpp (**※ main() 함수는 작성하지 마시오**)

AVL Class

- **Definition**
  - Height Balanced BST를 나타내는 Class
  - BST Class를 확장시킴 **(class AVL : public BST{})**
- **Constructor (Public으로 정의)**
  - **AVL()**
    - BST의 Constructor(**BST()**)를 호출하여 **root**를 **nullptr**로 초기화
- **Member Function (Private으로 정의)**
  - **int BF(TreeNode * curr_node)**
    - **curr_node**의 Balanced Factor를 계산하여 반환
  - **void insert(TreeNode *& curr_node, const int& key, const double& data)**
    - **curr_node**를 Root로 하는 Sub-tree에 Key 값으로 **key**를, Data 값으로 **data**를 가지는 Node 추가
    - Key 값이 동일한 Node가 이미 있는 경우, Data 값만 **data**로 변경
    - Insert 후 모든 Node에 대해 다음의 Property를 만족해야 함
      - Left Child Node의 Key 값은 Parent Node의 Key 값보다 작아야 함
      - Right Child Node의 Key 값은 Parent Node의 Key 값보다 커야 함
    - 위에서 명세된 **BF()** 함수를 이용하여 Tree의 Imbalance를 계산하고 아래에 명세된 **rotateSingle()**, **rotateDouble()** 함수를 이용하여 Insert 완료된 Tree가 Height Balanced Tree를 만족하도록 Rotate 수행
  - **void rotateSingle(TreeNode *& curr_node)**
    - 오른쪽과 같은 Imbalance 상황에서 **N2** Node를 Root Node로 하는 Height Balanced BST가 되도록 Rotate 수행
    - 강의자료 PPT 참조



■ **void rotateDouble(TreeNode *& curr_node)**



◆ 오른쪽과 같은 Imbalance 상황에서
N2 Node의 Left Child Node를 Root Node로 하는
Height Balanced BST가 되도록 Rotate 수행

◆ 강의자료 PPT 참조

■ **void remove(TreeNode *& curr_node, const int& key)**

◆ **curr_node**를 Root로 하는 Sub-tree에서 Key 값이 key인 Node 삭제

◆ BST Class의 **doRemoval()** 함수 사용 가능

◆ BST Class의 **remove()**와 동일하게 Node 삭제

◆ 삭제 후 Height Balanced BST가 되도록 Propagated Rotation 과정 필요

◆ **rotateSingle()**과 **rotateDouble()** 모두 가능한 경우, **rotateSingle()** 우선 수행

◆ 강의자료 PPT 참조

- **Member Function (Public으로 정의)**

■ **void insert(const int& key, const double& data)**

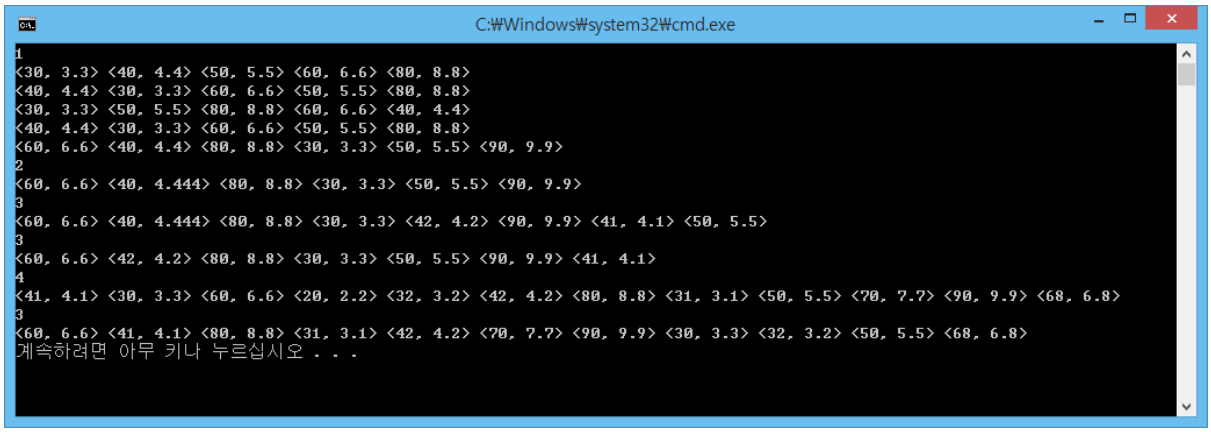
◆ BST에 Key 값으로 **key**를, Data 값으로 **data**를 가지는 Node 추가

◆ Key 값이 동일한 Node가 이미 있는 경우, Data 값만 **data**로 변경

■ **void remove(const int& key)**

◆ BST에서 Key 값이 **key**인 Node 삭제

주어진 HW3_Main_AVL.cpp를 실행하여 간단하게 AVL Class가 잘 구현되었는지 테스트 할 수 있다. 정상적인 결과가 나오기 위해서는 BST Class가 잘 구현되어 있어야 한다. (채점 시에는 더욱 복잡한 Case를 가지고 테스트할 예정)



P3 (20 점). [Hashing] integer 형의 key 값을 이용해 data 를 저장하는 Hash table 을 구현하고자 한다. Key 사이의 Collision 이 발생하는 경우 Linear-Probe based hashing 을 이용하여 data 를 저장하고자 할 때, 다음 조건을 만족하도록 HashTable Class 를 작성하시오.

제출 파일 : HW3_HashTable.h (**※ 헤더 파일 안에 main() 함수를 작성하지 마시오**)

HashTable Class

- **Definition**
  - Key값을 이용하여 data를 저장하는 HashTable
- **Member Variables (Private으로 정의됨)**
  - **HashNode class**
    - Hashtable에 저장되는 node의 class를 의미하며, integer형의 key와 template V형의 value값이 HashNode에 저장된다.
  - **HashNode<V> **table** : Hash node들이 저장될 Hashtable
  - **HashNode<V> *dummy** : dummy node로 node deletion에 사용됨
  - **int capacity** : HashTable이 저장할 수 있는 HashNode의 개수
  - **int number** : 현재 HashTable에 저장되어 있는 HashNode의 개수
- **Constructor**
  - **HashTable (int cap)**
    - capacity를 cap으로, number를 0으로 초기화
    - HashNode class의 constructor를 이용하여 key가 -1, value가 NULL이 되도록 dummy 생성
    - Table 변수에 HashNode<V>* 형의 node들을 capacity 개수만큼 담을 수 있는 array 생성
- **Destructor**
  - **~HashTable()** : table 및 table의 모든 HashNode와 dummy delete
- **Member Functions**
  - **int hashFunction (int key)**
    - key값을 받아들여 HashTable 내 address로 mapping하는 hash function
    - h(key) = key % capacity로 정의되어 있음
  - **void tableDoubling()**
    - HashTable이 많이 찰 경우 hashing의 성능이 떨어질 수 있으므로 capacity의 절반 이상으로 node가 저장될 경우 capacity size doubling 진행 (ex) capacity가 7일 경우 4개째 저장될 때, capacity가 8일 경우 5개째 저장될 때 진행

- ◆ insertNode() 함수 안에서 호출되어 진행됨
- ◆ 현재 capacity보다 두 배의 capacity를 가지는 table을 새로 생성
- ◆ 기존에 저장 되어있던 HashNode들을 새로운 capacity값에 대해 re-hashing을 진행하여 새로운 table에 저장 (단순히 Array 복사하는 것 아님)
- ◆ 기존에 존재하였던 table은 delete

■ **void insertNode(int key, V value)**

- ◆ key값을 받아들여 hashFunction()을 거쳐 나온 table의 address에 key, value값을 가지는 HashNode를 생성하여 저장
- ◆ 이미 해당 address에 HashNode가 존재하는 경우 Linear-probing을 이용하여 다음 칸으로 한 칸씩 이동해보며 빈 곳 혹은 dummy가 저장된 곳에 저장
- ◆ 만약 key값이 동일한 HashNode가 이미 존재하는 경우 해당 HashNode에 value값만 덮어씀
- ◆ HashNode 저장 후 number값 update, 이후 capacity의 절반 이상 node가 저장된 경우 tableDoubling() 호출

■ **V deleteNode(int key)**

- ◆ HashTable내에서 key값을 가지는 HashNode 삭제 후 number값 update
- ◆ hashFunction()을 이용하여 address 탐색, 만약 해당 위치에 key값을 가지는 node가 존재하지 않는 경우 다음 칸으로 한 칸씩 이동해가며 탐색
- ◆ key값을 가지는 HashNode가 존재하는 경우 해당 HashNode를 delete하고 그 address에 dummy 저장 후 HashNode의 value값 return
- ◆ 탐색 과정에서 비어있는 table address를 찾거나, 모든 address를 탐색했는데도 찾지 못한 경우는 찾고자 하는 HashNode가 없는 경우이므로 key값을 가지는 HashNode가 존재하지 않음을 출력하고 NULL return

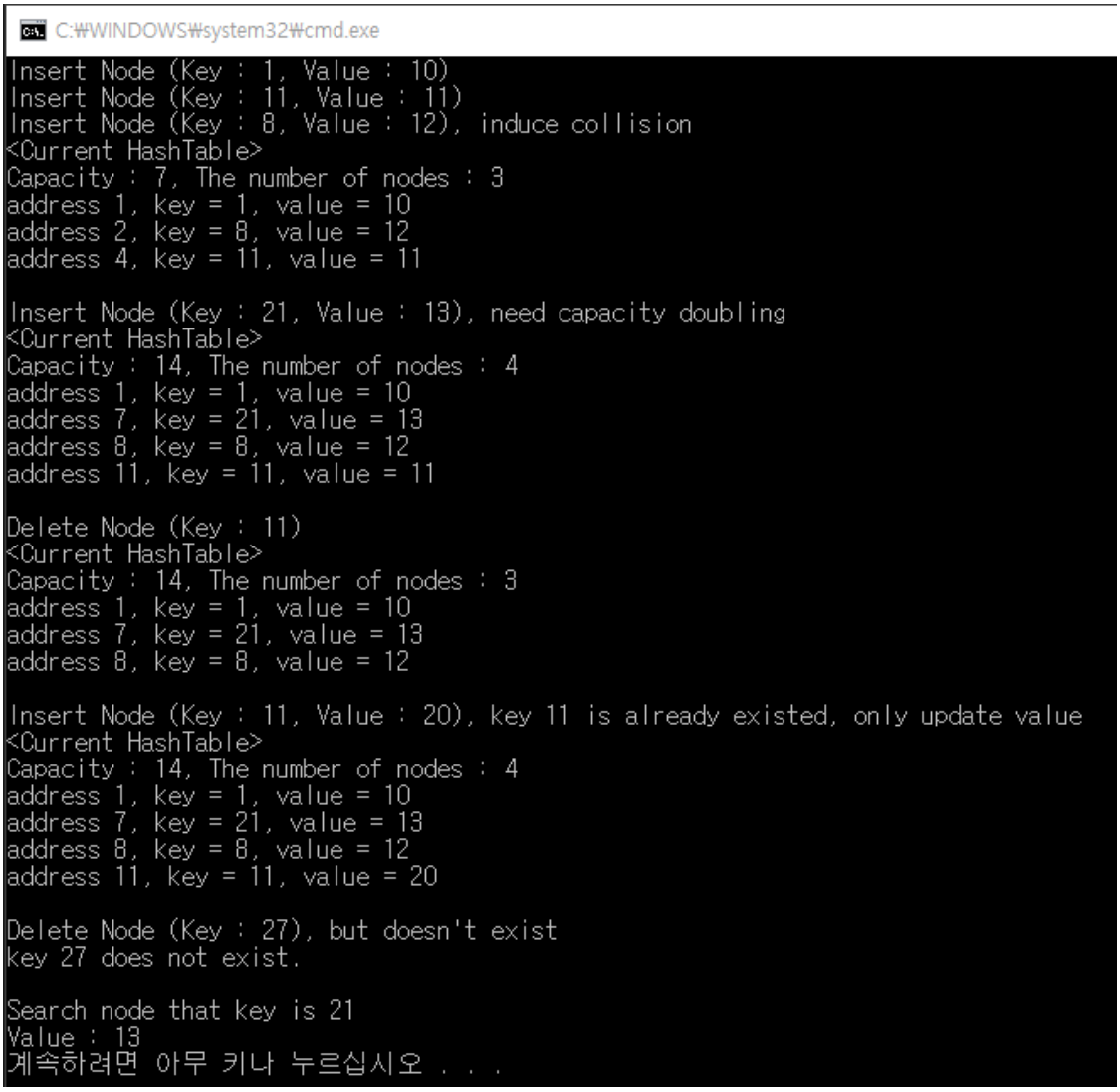
■ **V search(int key)**

- ◆ hashFunction()을 이용하여 key값을 가지는 HashNode 탐색
- ◆ key값을 가지는 HashNode가 존재하는 경우, 해당 HashNode의 value값 return
- ◆ 존재하지 않는 경우 NULL return

■ **void display()**

- ◆ 현재 HashTable의 capacity, number 출력
- ◆ HashTable 안에 저장되어 있는 HashNode들의 key, value값 출력
- ◆ 다음 장의 출력 양식 참고

주어진 HW3_Main_Hash.cpp를 실행하여 간단하게 HashTable Class가 잘 구현되었는지 테스트할 수 있다. (채점 시에는 더욱 복잡한 Case를 가지고 테스트할 예정)



※ Dummy가 필요한 이유 : 동일한 address를 가지는 노드는 Linear-probing을 이용하여 연속적으로 배치되는데, 중간에 어떤 노드가 삭제가 되어 nullptr로 바뀐다면 그 뒤의 노드는 Linear-probing으로 탐색될 수 없다. 이러한 상황을 방지하기 위하여 dummy를 두어 Linear-probing을 가능하게 한다.

- **참고**
  - **과제 채점은 Microsoft Visual Studio 2017에서의 동작을 기준으로 함**
  - **주어진 변수, 함수 이름 절대 고치지 말것! (대소문자 구분) 이를 어길 시 감점**
  - **필요한 경우 새로운 멤버 변수 또는 함수 추가 가능**
  - **주어진 예시와 같은 양식으로 입/출력되게 할 것! 이를 어길 시 감점**
  - **채점은 문제에 주어진 예시보다 복잡한 Test Case로 진행**
  - **Compile 오류 시 해당 문제 0점 처리할 예정이므로, 충분한 Test 후 제출 요망**
  - **표절 금지. 표절 적발 시 해당 과제 0점 처리 및 교수님께 통보**
  - **질문이 있는 경우, 검색을 충분히 해 본 후에도 해결되지 않는 것에 한해 질문 요망 (단순히 소스코드를 보내고 디버그 요청하는 질문에는 답변하지 않을 예정)**
- **풀이 및 제출 방법**
  - 각 문제에 대한 코드는 문제에서 특별한 언급이 없으면 **선언과 구현을 분리**하여 각각의 .h / .cpp 파일에 작성 **(대소문자 반드시 구분, 이를 어길 시 감점)**
  - 파일 이름은 각 문제에 주어진 대로 정의 **(이를 어길 시 감점)**
  - 작성한 코드를 하나의 압축 파일로 압축
    - 압축 파일 이름은 "**HW3_(이름)_(학번).zip**"으로, zip 방식 압축 사용
    - 예) "**HW3_김태환_2017-11111.zip**"
  - 만든 압축 파일을 **eTL** 강의 페이지에 마련된 **"과제3" 제출함**으로 제출
- **제출 기한**
  - **5월 25일 (금) 오후 11시 59분**까지
  - 오류 발생으로 eTL 과제함에 제출이 되지 않는 경우 **이메일**로 제출
    E-mail : ds@snucad.snu.ac.kr
  - **지연 제출은 받지 않음(eTL / 이메일 모두), 지연 제출 또는 미제출 시 0점**
  - **파일이 첨부되어 제출되었는지 반드시 확인! 첨부되지 않은 채로 제출 시 0점**