1. 기본 데이터 구조 설명

char key 를 가지는 binary tree node 를 다음과 같이 정의한다.

```
#define KEYLENGTH 3
#define BULK_SIZE 4096
struct BTNode {
  char bulk[BULK_SIZE];
  struct BTNode *left, *right;
};
```

- key 는 KEYLENGTH 의 길이를 가지는 문자열이다.
  (예: "abc", "123", ...)
  "bulk"에는 key 가 숨겨져 있다. 따라서 key 를 찾기 위해서는 평균 BULK_SIZE/2 번의 검색을 해야 한다.
- left, right 는 각각 left subtree, right subtree 에 대한 포인터이다.

---

2. 함수정의(주어진 것들)

```
const char* getkey(struct BTNode *a);
```
- 이 함수를 통해서만 key 값을 얻어올 수 있다.

```
int setkey(struct BTNode *a, const char kw[]);
```
- 이 함수를 통해서만 node 의 key 값을 바꿀 수 있다.

```
int copykey(struct BTNode *dst, struct BTNode *src)
{ return setkey(dst,getkey(src)); }
```
- 노드간의 key 값 qhrtk

```
int comparekey(struct BTNode *a, struct BTNode *b);
// return value: (by character comparison)
//   -1 if a's key < b's key
//   0 if a's key == b's key
//   +1 if a's key > b's key
```
- key 값 비교

```
struct BTNode *generate_btnode(const char kw[]);
```
- 주어진 key 값을 가지는 노드를 하나 생성한다

```
void free_bt_recursive (struct BTNode *bt);
```
- 트리 메모리 전체 반환

struct BTNode *copy_bt_recursive (struct BTNode *bt);
- 트리 복사(복제)

struct BTNode *insert_left_bcnode(struct BTNode *parent, struct BTNode *newPtr);
- 왼쪽에 집어 넣는다. LHBT (left-half binary tree)를 만들때 사용한다.

struct BTNode *readkeys_textfile_LHBT(const char infile[], int *pN);
- 주어진 파일에서 LHBT (left-half binary tree)의 형태로 리스트를 만들어 반환한다.

```
////////////////////////////////////////////////////////////
// FILL 1: generate a binary search tree using insertion
////////////////////////////////////////////////////////////
struct BTNode *insert_to_BST_leaf(struct BTNode *bst, struct BTNode *newPtr)
{
  if ( bst == NULL ) return newPtr; // new bst as the input node
  else if ( newPtr == NULL ) return bst; // nothing to add
  else {
    if ( comparekey(bst, newPtr) < 0 ) {
      /* FILL */
    }
    else {
      /* FILL */
    }
  }
  return bst;
}

struct BTNode *generate_BST_by_insertion(struct BTNode *lhbt)
{
  /* FILL */
  /* (hint: use insert_to_BST_leaf repeatedly) */
}
```

---

3. 출력예제

예) input/i9.txt -> output/o9.txt

```
====================================
100-400-000-800-900-800-500-700-900
total 9 nodes
====================================
```

```
000 100 400 500 700 800 800 900 900
total 9 nodes (sorted)
====================================
              900
                  900
          800
              800
                      700
                  500
     400
100
     000
BST height 6
====================================
100/400/800/900
                 +900
             +800
                 +500/700
   +000
BST height 6
====================================
100/400/800/900
   |          |     +900
   |          +800
   |               +500/700
   +000
BST height 6
====================================
800/900/900
   |     +800
   +500/700
        +100/400
             +000
Complete BST height 4
====================================
100/400/800/900
   |          |     +900
   |          +800
   |               +500/700
   +000
TIME 0.00613 seconds
```

output 폴더에 더 많은 입력 예제들이 있다.

---
4. 구현할 것들

1) BST (bineary search tree) 를 하나씩 삽입하여 만들기

struct BTNode *insert_to_BST_leaf(struct BTNode *bst, struct BTNode *newPtr)
- binary search tree 에서 적절한 위치를 찾아서 left node 에 삽입한다.

struct BTNode *generate_BST_by_insertion(struct BTNode *lhbt)
- 주어진 LHBT (리스트)에서 노드를 하나씩 뽑아내서 삽입한다.

2) BST 를 정렬순서로 화면에 표시

```
==================================
000 100 400 500 700 800 800 900 900
total 9 nodes (sorted)
==================================
int print_BST_sortedorder(FILE *fp, struct BTNode *bst, int level)
  // (hint: inorder traversal)
  // INPUT
  //    fp: 화면에 출력할 경우 stdout, 아니면 file pointer
  //    bst: root node of the BST, should satisfy the property of
  //       binary search tree, left <= center < right
  //    level: 현재 bst 의 level 이다. 0 부터 시작하고, 필요없으면 안 써도 됨
  // RETURNs number of NODES in the list
```

3) BST 를 아래와 같이 화면에 표시(node/line)

```
==================================
            900
                900
        800
            800
                    700
                500
    400
100
    000
BST height 6
==================================
int print_BST_right_center_left(FILE *fp, struct BTNode *bst, int level)
```

```
  // INPUT
  //    (same as print_BST_sortedorder)
  // RETURNs HEIGHT-1 of the printed tree (2 in the above example)
  //    (hint: printing order is right -> center -> left
  //     carefully count the number of spaces)
```
* hint: 표시순서는 right -> center -> left

4) BST 를 아래와 같이 화면에 표시(center 와 right 를 한 줄로, left 는 아래)
right 를 표시할 때문 "/"를 앞에 붙이고, left 를 표시할 때는 parent node 다음의
수직위치에 오도록 빈 같을 띄우고 "+"를 앞에 붙인다.
(위의 3 줄은 *11 월 2 일에 수정함.)

```
====================================
100/400/800/900
                +900
            +800
                +500/700
   +000
BST height 6
====================================
int print_BST_1(FILE *fp, struct BTNode *bst, int level)
  // prints a binary search tree, rotated by 270 degrees, with less lines
  //   1) center and right are in the same line
  //   2) left subtree is below the center
  //   3) right is connected by '/' and left by '+'
  // Note: key's length is fixed to KEYLENGTH,
  // so left and right begins at (KEYLENGTH+1)*level+1
  //    (hint: printing order is center -> right -> left)
```
* hint: 표시순서에 따라 recursive 함수로 구현할 수 있다.

5) BST 를 아래와 같이 화면에 표시(수직으로 right subtree 까지 line 을 그린다)
위의 4)번과 표시 방법은 같고 "|"만 추가됨

```
====================================
100/400/800/900
   |         |    +900
   |         +800
   |              +500/700
   +000
BST height 6
====================================
```

```
int print_BST_2(FILE *fp, struct BTNode *bst, int level)
  // Hint: stack or some extra variables may help.
  //       static variable can be used as well
  //       You may add additional parameter to the function if necessary
```
* 필요하다면 함수에 인수들을 추가해도 된다.

6) BST 를 complete BST 로 바꾸어서 height 를 최소화한다.

==========

```
800/900/900
   |    +800
   +500/700
       +100/400
           +000
```

Complete BST height 4

==========

```
struct BTNode *BST_to_completeBST(struct BTNode *bst, int numNodes)
  // convert a BST to complete BST (minimum height, filling in left first)
  // INPUT
  //    bst: root node of the BST, should satisfy the property of
  //       binary search tree, left <= center < right
  //    numNodes: number of nodes in the bst
  //       if not necessary in your implementation, do not have to use it
  // RETURNs a COMPLETE BST
  // (hint: using extra memory (arrays or lists) may help,
  //   array's rule for parent-child, sorted list, etc.)
```
* 필요하다면 함수에 인수들을 추가하거나 삭제해도 된다.
* hint: complete binary tree 저장을 위한 array 를 중간에 사용할 수도 있다.