HW4 Readme

4-1思路

- 這一題是BST的實作，需要完成插入刪除，並且依照inorder和levelorder做輸出。

CODE分析

Insert函式

```
/* A utility function to
   insert a new node with given key in
 * BST */
node* insert(node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
#  BST的插入規則 : 小於 parent 的 key 要放在leftchilde
#  大於parent要放在rightchilde
```

Delete函式

```
node* minValueNode( node* nd )
{

    node* current = nd ;

    /* loop down to find the leftmost leaf */
    while (current && current->left != NULL)
    {
        current = current->left ;
    }

    return current;
}

# 這邊實作一個從subtree中找最小的函式，用於方便刪除。
# 原因是刪除node時，若刪除的node有leftchild和rightchild
# 此時可以從rightchilde中找最小去補上刪除的node
```

```
node* deleteNode( node* root , int key)
{
    // base case
    if (root == NULL)
        return root;

    // If the key to be deleted
    // is smaller than the root's
    // key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted
    // is greater than the root's
    // key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key,
    // then This is the node
    // to be deleted
    else {
        // node with only one child or no child
        if (root->left == NULL) {
            node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            node* temp = root->left;
            free(root);
            return temp;
        }


        // (smallest in the right subtree)
        node* temp = minValueNode(root->right);

        // Copy the inorder
        // successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
```

```
    return root;
}
# 先找到耀珊的節點，然後再分case處理。
# 刪除有三個case需要分別作處理 :
# 沒有child，只有一個child，以及有兩個child。
```

inorder函式

```
void inorder( node* root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}
# inorder的部分是依照 LVR 的規則去做尋訪 。
```

levelorder函式

- levelorder這邊我有另外寫一個height函式去計算樹高。

```
int height(node* node)
{
    if (node == NULL)
        return 0;
    else {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return (lheight + 1);
        else
            return (rheight + 1);
    }
}
```

```
void printLevelOrder( node* root)
{
    int h = height(root);
    int i;
    for (i = 1; i <= h; i++)
        printCurrentLevel(root, i);
}
/* Print nodes at a current level */

void printCurrentLevel( node* root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        printf("%d ", root->key);
    else if (level > 1) {
        printCurrentLevel(root->left, level - 1);
        printCurrentLevel(root->right, level - 1);
    }
}
# levelorder的實作 : 從root開始 ， 慢慢往下面的level輸出。
# 從root開始往下 ，  由leftchild到rightchild
```

4-2思路

- 這邊需要找最短路徑，我用的方法是先計算**root到key**的路徑，再計算從 **key 到 treasure** 的路徑。
- 計算完路徑後，我使用陣列存起來，接著依序輸出。
- 不過因為我有分兩段輸出，因此需考慮重複輸出富情形。
- 這邊需使用插入節點的實作

CODE分析

插入函式

```
/* A utility function to
   insert a new node with given key in
 * BST */
node* insert(node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
#  BST的插入規則 : 小於 parent 的 key 要放在leftchilde
#  大於parent要放在rightchilde
```

計算最短路徑函式

```
int CalculatePath(node* root , int * path  , int x )
{
    // if root is NULL
    // there is no path
    if ( !root ) return 0 ;


    // push the node's value in 'arr'
    path[ num_patha ] = root -> key ;
    num_patha++ ;

    // if it is the required node
    // return true
    if (root->key == x) return 1 ;

    // else check whether the required node lies
    // in the left subtree or right subtree of
    // the current node
    if ( CalculatePath(root->left , path , x ) ||
         CalculatePath (root->right, path, x) )
    return 1 ;
    // required node does not lie either in the
    // left or right subtree of the current node
    // Thus, remove current node's value from
    // 'patharray'and then return false
    num_patha -- ;
    path[ num_patha ] = -1 ;
    return 0 ;
}

# 這邊有利用backtracking的方式，從樹的左邊右邊尋找key。
```

輸出路徑函式

```
void PrintPath(node* root , int na , int nb )
{
    // array to store the path of
    // first node n1 from root
    // na for original nb for final
    int patha[ 1000 ] ;
    num_patha = 0 ;

    // array to store the path of
    // second node n2 from root
    int pathb[ 1000 ] ;
    num_pathb = 0 ;

    CalculatePatha(root ,  patha   , na );
    CalculatePathb(root ,  pathb   , nb );

    int intersection = -1;


    int i = 0, j = 0;
    while ( i != num_patha || j != num_pathb)
    {
        // Keep moving forward until no repeatnode
        // is found
        if (i == j && patha[ i ] == pathb[ j ] )
        {
            i++;
            j++;
        }
        else
        {
            intersection = j - 1;
            break;
        }
    }

    // Print the required path
    for (int i = num_patha - 1 ; i > intersection; i--)
    {
        printf("%d->",patha[ i ]) ;
    }

    for ( int i = intersection ; i < num_pathb ; i++)
    {
        if( i == num_pathb - 1 )
```

```
46              {
47                  last = pathb[ i ] ;
48              }
49              printf("%d->",pathb[ i ]) ;
50          }
51
52      }
53
54      # 這邊一開始的na即為root ， nb 為key 。
55      # 第二次的na為key ， nb 為 treasure 。
56      # 由於第一次的nb和第二次的na會有重複的情形 。
57      # 因此有計算repeatnode而不去輸出。
```