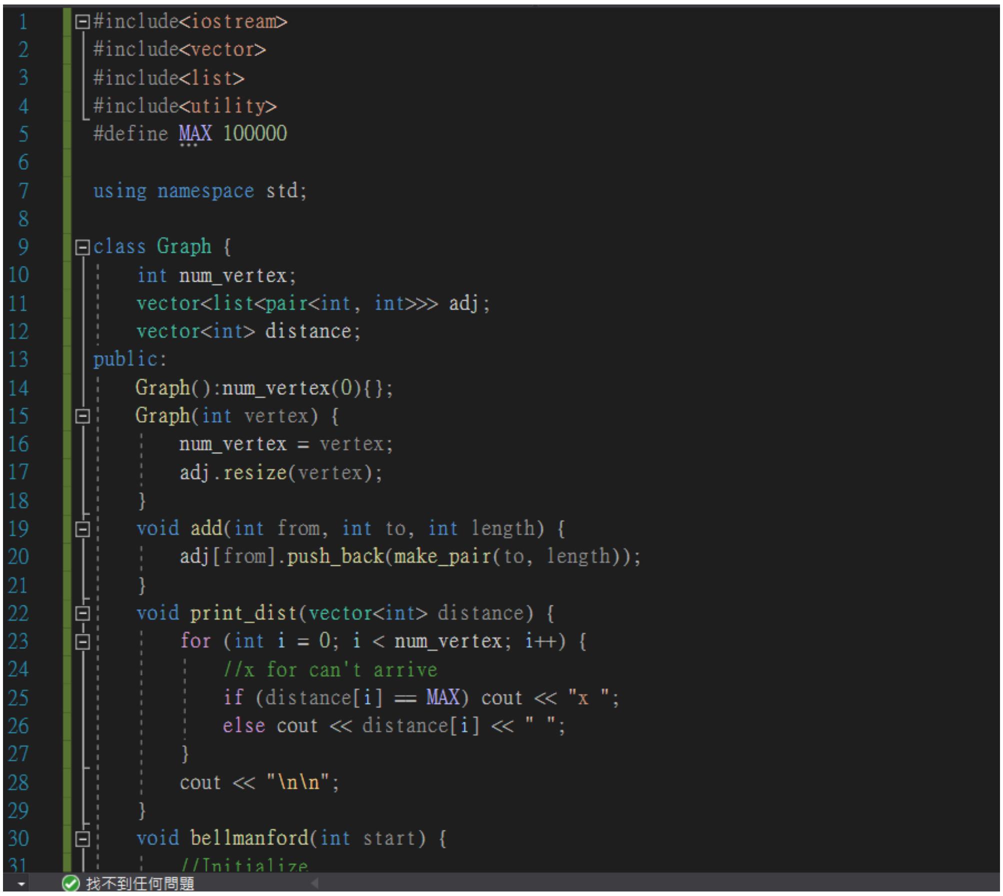
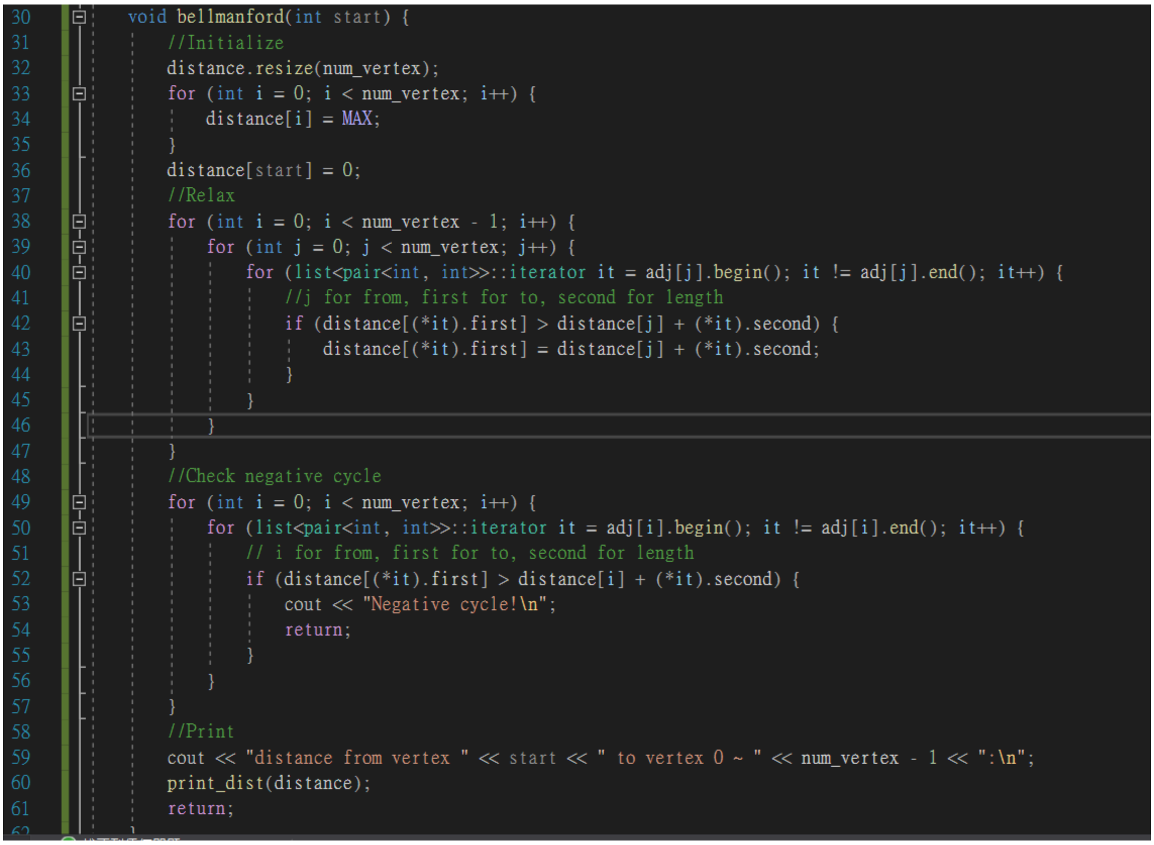
HW3 110550071 田松翰

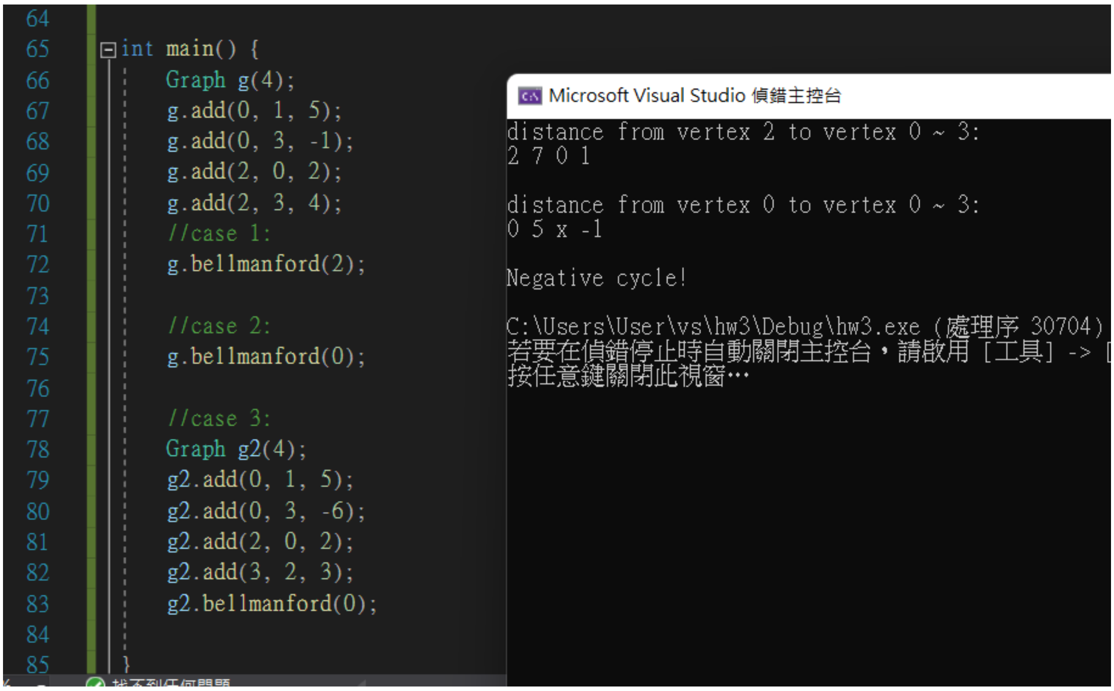
1. Bellmanford



```cpp
#include<iostream>
#include<vector>
#include<list>
#include<utility>
#define MAX 100000

using namespace std;

class Graph {
    int num_vertex;
    vector<list<pair<int, int>>> adj;
    vector<int> distance;
public:
    Graph():num_vertex(0){};
    Graph(int vertex) {
        num_vertex = vertex;
        adj.resize(vertex);
    }
    void add(int from, int to, int length) {
        adj[from].push_back(make_pair(to, length));
    }
    void print_dist(vector<int> distance) {
        for (int i = 0; i < num_vertex; i++) {
            //x for can't arrive
            if (distance[i] == MAX) cout << "x ";
            else cout << distance[i] << " ";
        }
        cout << "\n\n";
    }
    void bellmanford(int start) {
        //Initialize
```



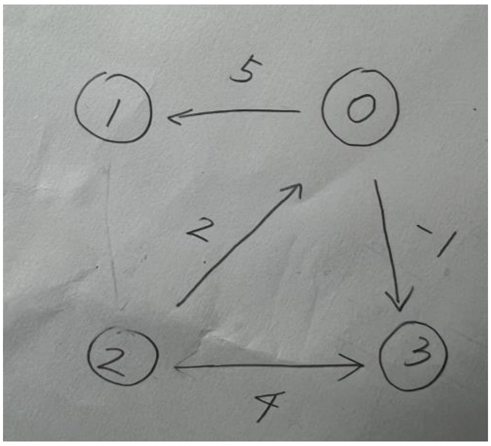
```cpp
    void bellmanford(int start) {
        //Initialize
        distance.resize(num_vertex);
        for (int i = 0; i < num_vertex; i++) {
            distance[i] = MAX;
        }
        distance[start] = 0;
        //Relax
        for (int i = 0; i < num_vertex - 1; i++) {
            for (int j = 0; j < num_vertex; j++) {
                for (list<pair<int, int>>::iterator it = adj[j].begin(); it != adj[j].end(); it++) {
                    //j for from, first for to, second for length
                    if (distance[(*it).first] > distance[j] + (*it).second) {
                        distance[(*it).first] = distance[j] + (*it).second;
                    }
                }
            }
        }
        //Check negative cycle
        for (int i = 0; i < num_vertex; i++) {
            for (list<pair<int, int>>::iterator it = adj[i].begin(); it != adj[i].end(); it++) {
                // i for from, first for to, second for length
                if (distance[(*it).first] > distance[i] + (*it).second) {
                    cout << "Negative cycle!\n";
                    return;
                }
            }
        }
        //Print
        cout << "distance from vertex " << start << " to vertex 0 ~ " << num_vertex - 1 << ":\n";
        print_dist(distance);
        return;
```



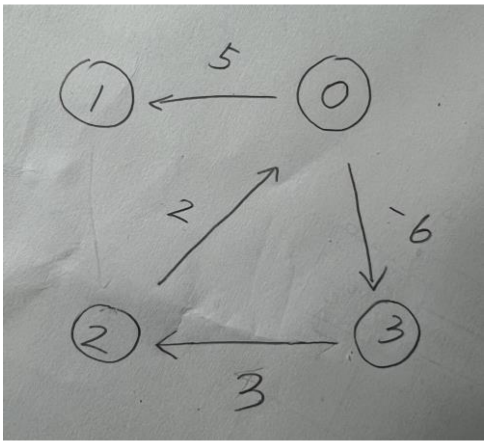
```
int main() {
    Graph g(4);
    g.add(0, 1, 5);
    g.add(0, 3, -1);
    g.add(2, 0, 2);
    g.add(2, 3, 4);
    //case 1:
    g.bellmanford(2);

    //case 2:
    g.bellmanford(0);

    //case 3:
    Graph g2(4);
    g2.add(0, 1, 5);
    g2.add(0, 3, -6);
    g2.add(2, 0, 2);
    g2.add(3, 2, 3);
    g2.bellmanford(0);

}
```

```
distance from vertex 2 to vertex 0 ~ 3:
2 7 0 1

distance from vertex 0 to vertex 0 ~ 3:
0 5 x -1

Negative cycle!

C:\Users\User\vs\hw3\Debug\hw3.exe (處理序 30704)
若要在偵錯停止時自動關閉主控台，請啟用 [工具] -> [
按任意鍵關閉此視窗…
```

Case1,2:



Case3:

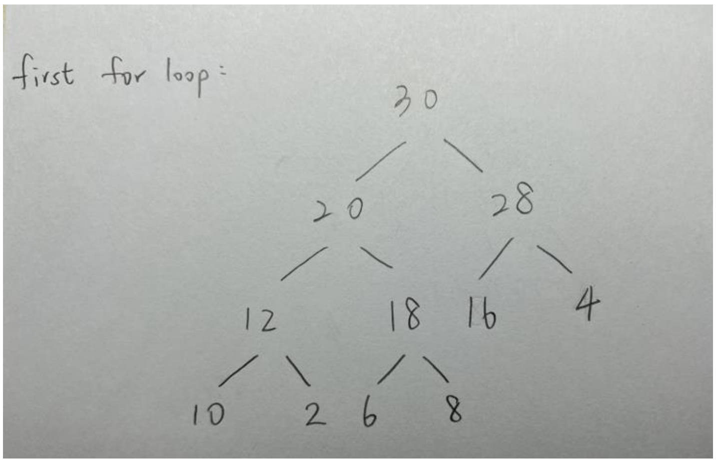


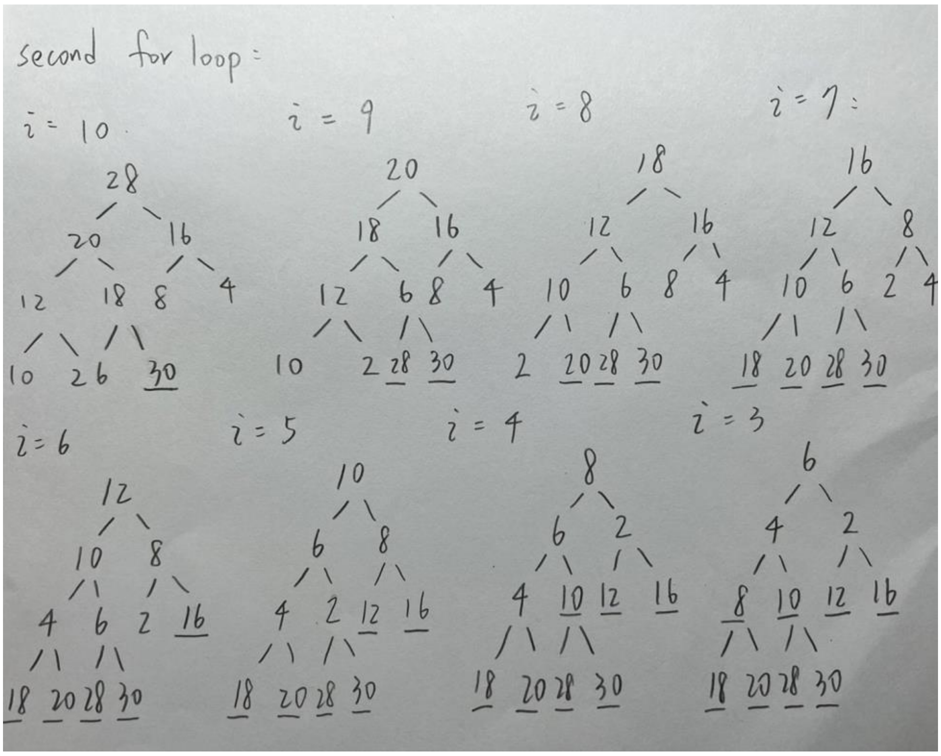
Case1: normal case

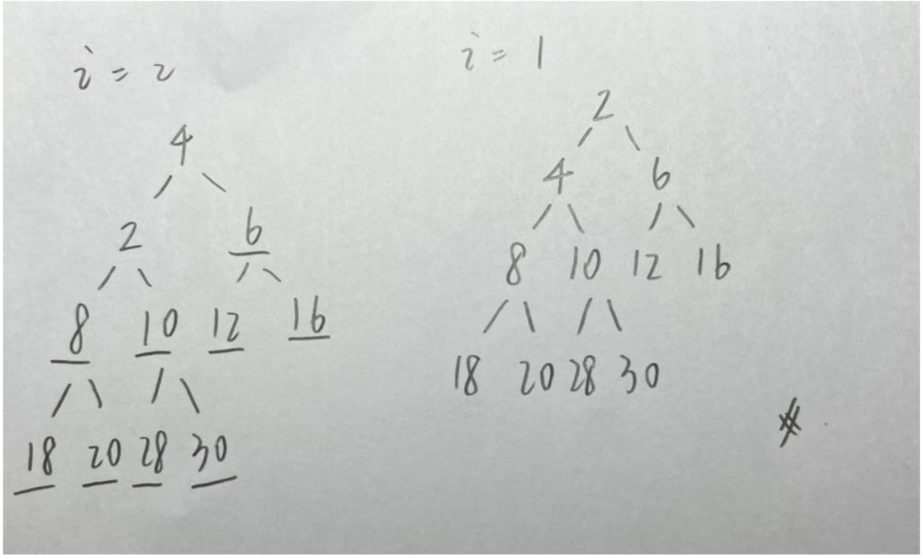
Case2: vertex 0 can’t get to vertex 2

Case3: negative cycle

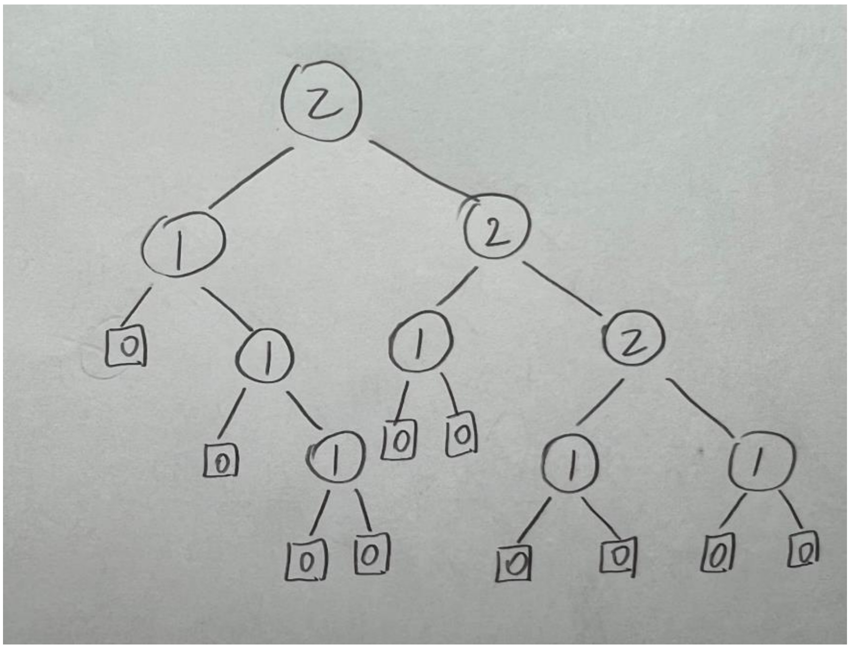
2. Heapify


first for loop:
30
20 28
12 18 16 4
10 2 6 8

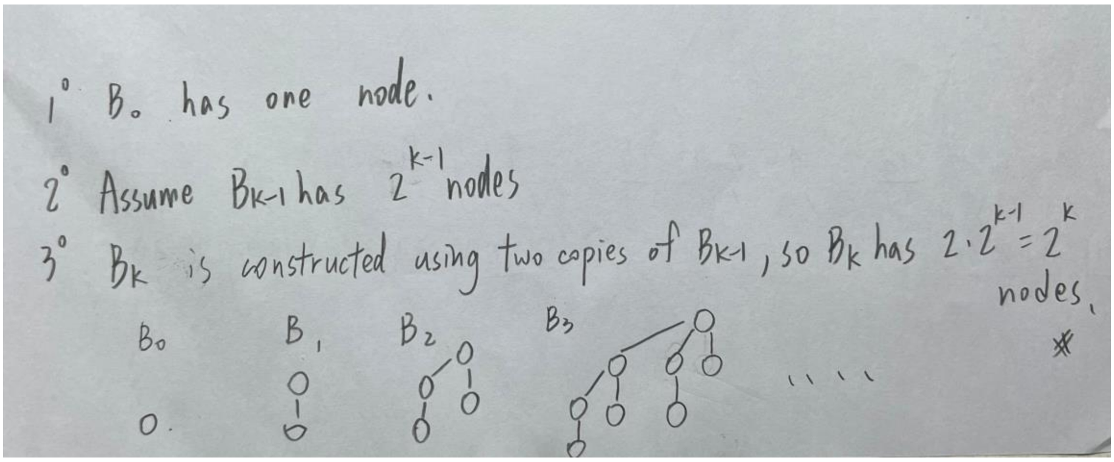

second for loop:
i = 10
28
20 16
12 18 8 4
10 2 6 30
i = 9
20
18 16
12 6 8 4
10 2 28 30
i = 8
18
12 16
10 6 8 4
2 20 28 30
i = 7:
16
12 8
10 6 2 4
18 20 28 30
i = 6
12
10 8
4 6 2 16
18 20 28 30
i = 5
10
6 8
4 2 12 16
18 20 28 30
i = 4
8
6 2
4 10 12 16
18 20 28 30
i = 3
6
4 2
8 10 12 16
18 20 28 30


i = 2
4
2 6
8 10 12 16
18 20 28 30
i = 1
2
4 6
8 10 12 16
18 20 28 30
#

3. Shortest



4. Binomial tree


1° $B_0$ has one node.

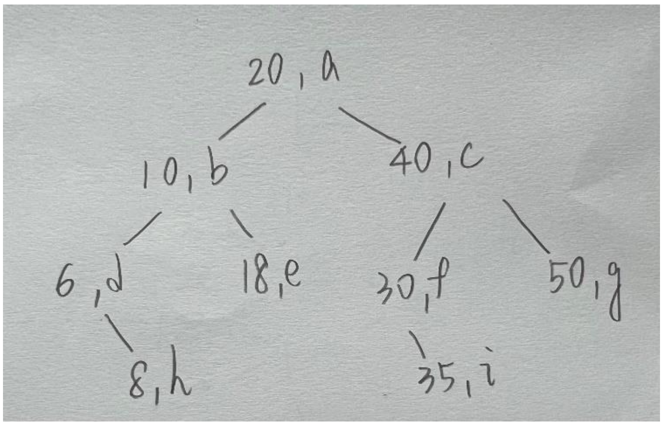
2° Assume $B_{k-1}$ has $2^{k-1}$ nodes

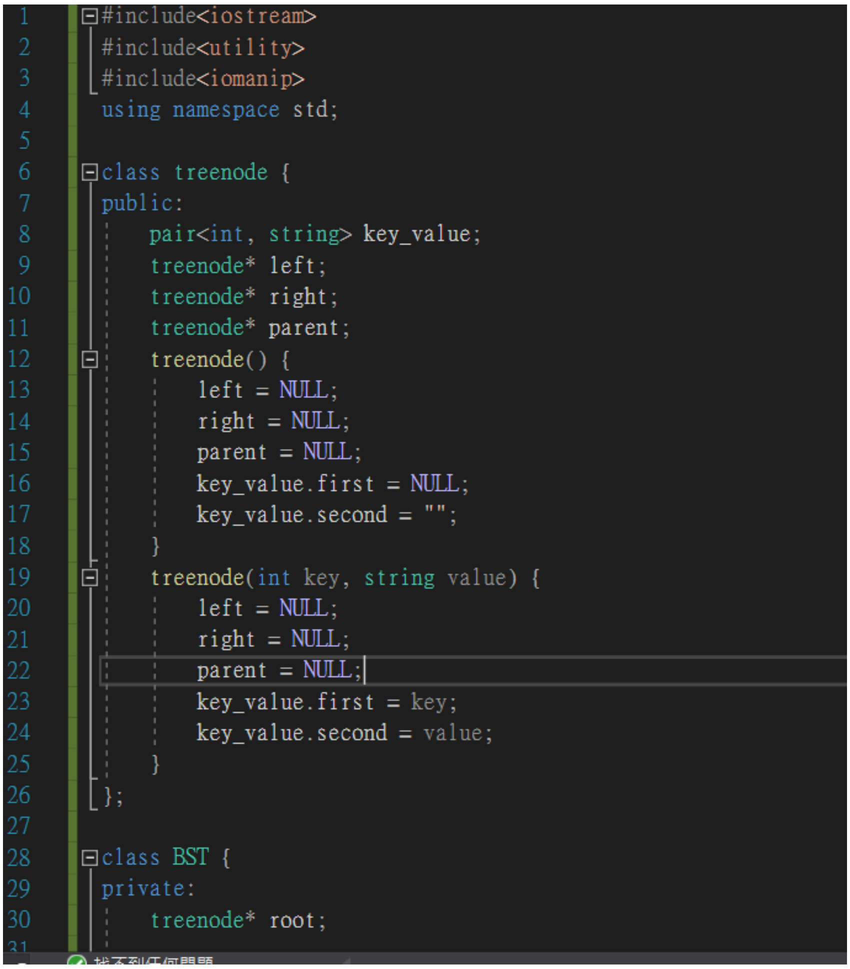
3° $B_k$ is constructed using two copies of $B_{k-1}$, so $B_k$ has $2 \cdot 2^{k-1} = 2^k$ nodes.

#

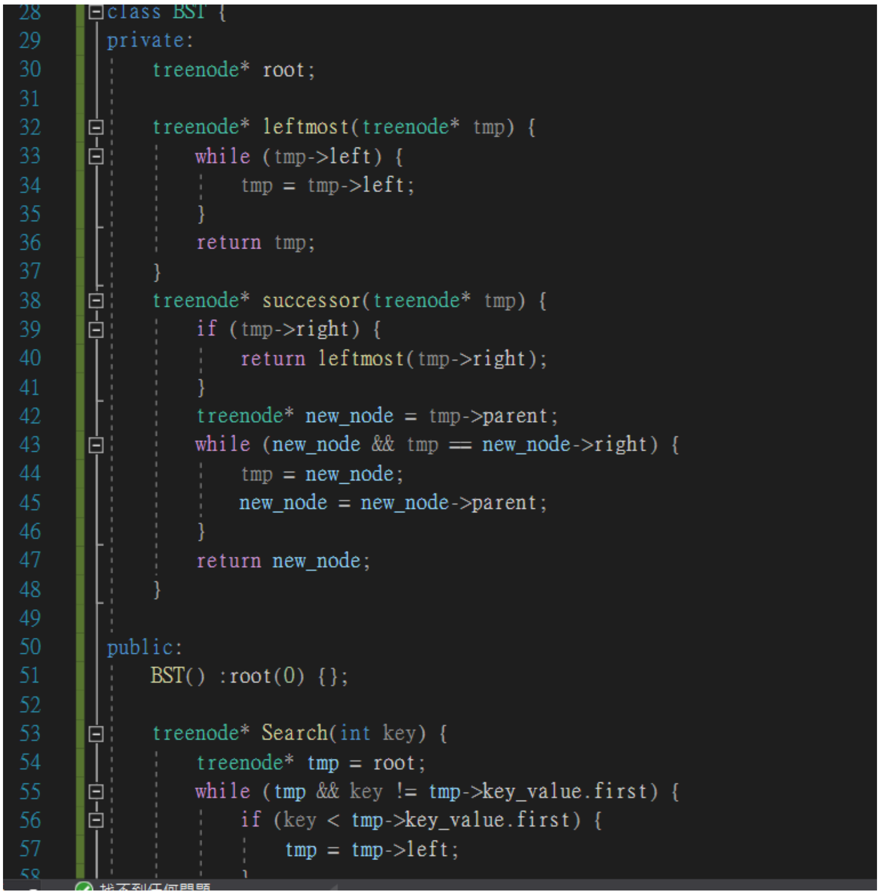
5. BST delete

Original tree ->

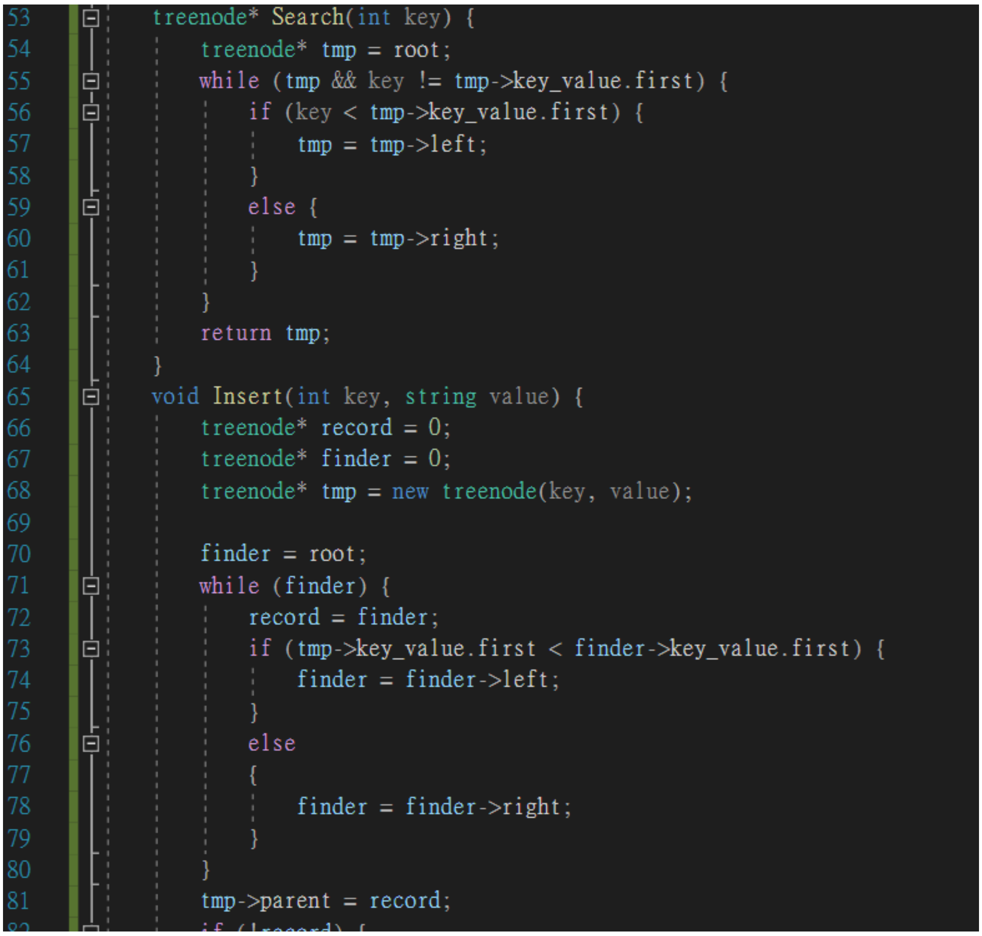




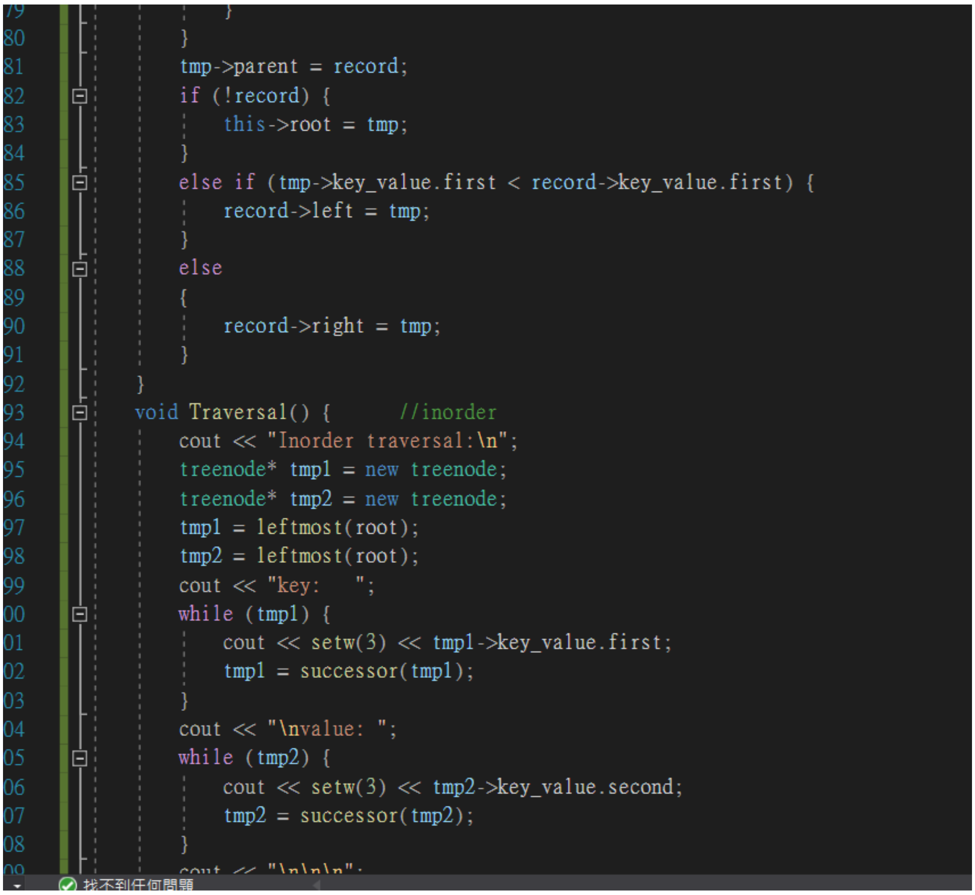
```cpp
#include<iostream>
#include<utility>
#include<iomanip>
using namespace std;

class treenode {
public:
    pair<int, string> key_value;
    treenode* left;
    treenode* right;
    treenode* parent;
    treenode() {
        left = NULL;
        right = NULL;
        parent = NULL;
        key_value.first = NULL;
        key_value.second = "";
    }
    treenode(int key, string value) {
        left = NULL;
        right = NULL;
        parent = NULL;
        key_value.first = key;
        key_value.second = value;
    }
};

class BST {
private:
    treenode* root;
```



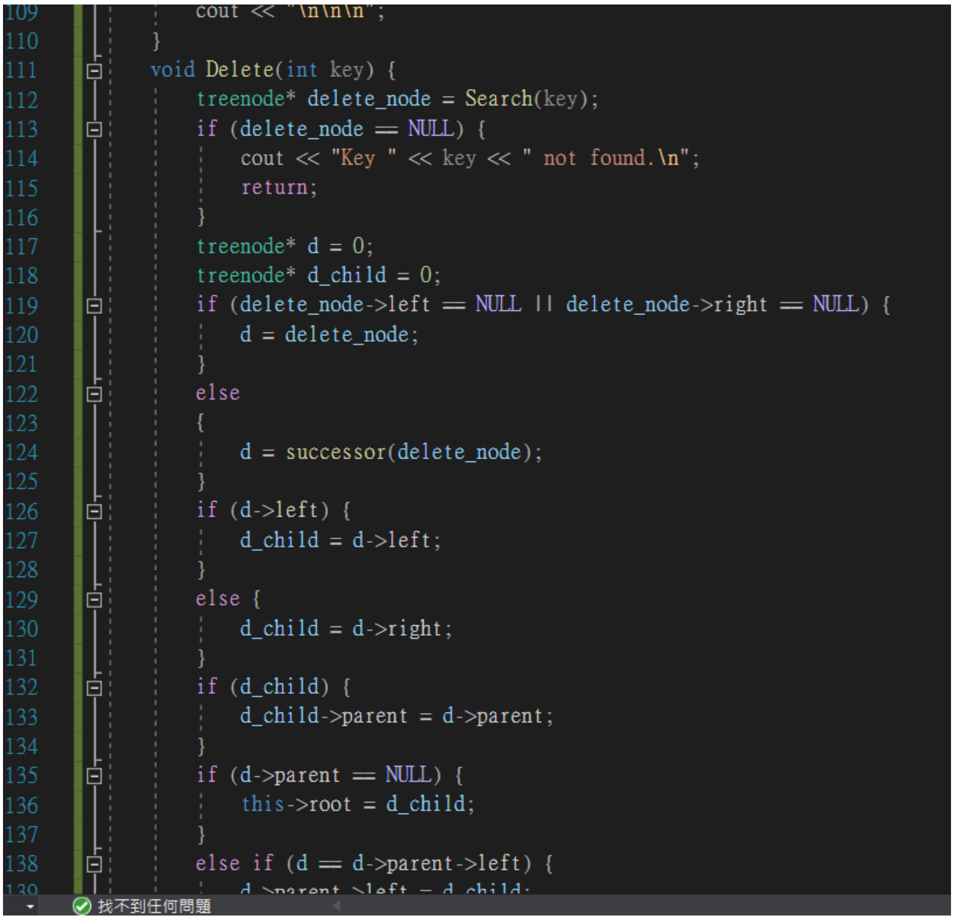
```cpp
class BST {
private:
    treenode* root;

    treenode* leftmost(treenode* tmp) {
        while (tmp->left) {
            tmp = tmp->left;
        }
        return tmp;
    }
    treenode* successor(treenode* tmp) {
        if (tmp->right) {
            return leftmost(tmp->right);
        }
        treenode* new_node = tmp->parent;
        while (new_node && tmp == new_node->right) {
            tmp = new_node;
            new_node = new_node->parent;
        }
        return new_node;
    }

public:
    BST() :root(0) {};

    treenode* Search(int key) {
        treenode* tmp = root;
        while (tmp && key != tmp->key_value.first) {
            if (key < tmp->key_value.first) {
                tmp = tmp->left;
```



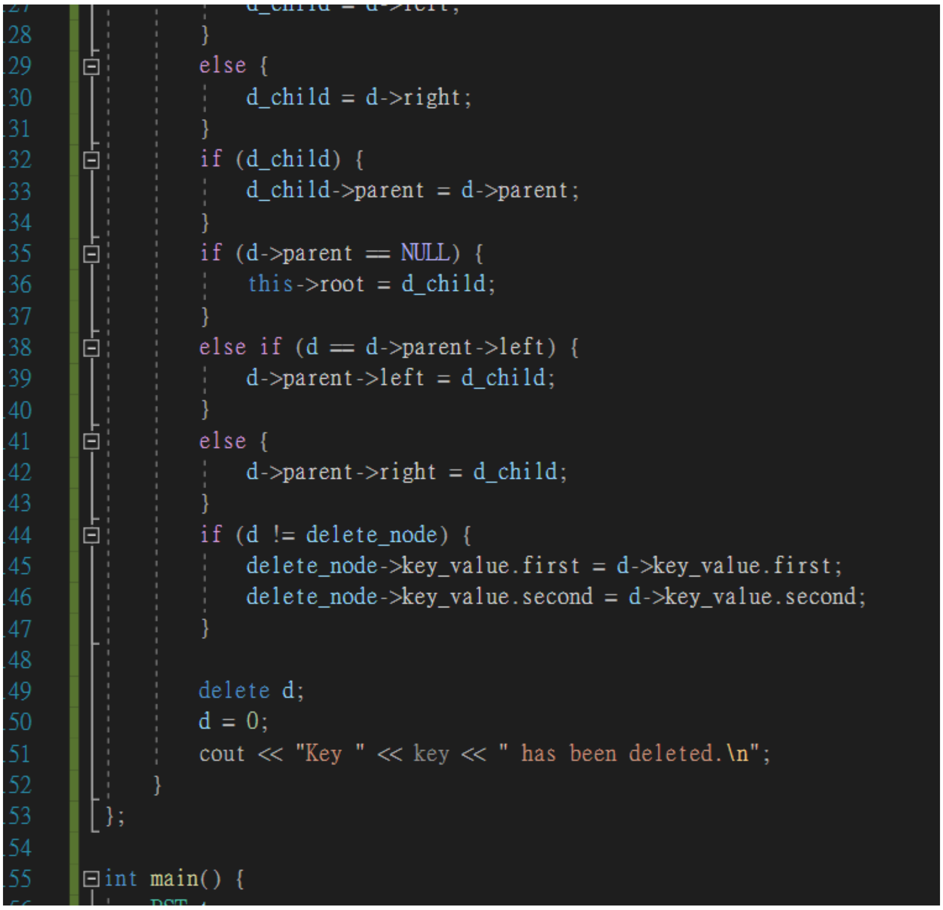
```
    treenode* Search(int key) {
        treenode* tmp = root;
        while (tmp && key != tmp->key_value.first) {
            if (key < tmp->key_value.first) {
                tmp = tmp->left;
            }
            else {
                tmp = tmp->right;
            }
        }
        return tmp;
    }
    void Insert(int key, string value) {
        treenode* record = 0;
        treenode* finder = 0;
        treenode* tmp = new treenode(key, value);

        finder = root;
        while (finder) {
            record = finder;
            if (tmp->key_value.first < finder->key_value.first) {
                finder = finder->left;
            }
            else
            {
                finder = finder->right;
            }
        }
        tmp->parent = record;
        if (!record) {
```



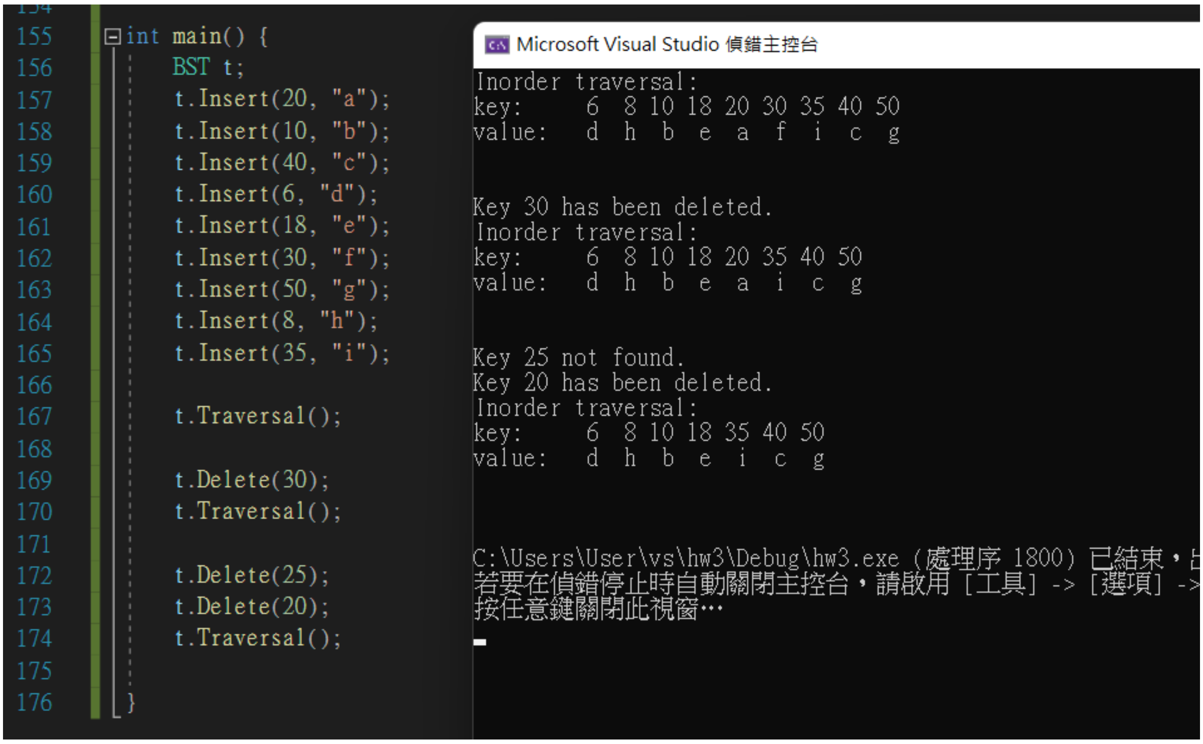
```
            }
        }
        tmp->parent = record;
        if (!record) {
            this->root = tmp;
        }
        else if (tmp->key_value.first < record->key_value.first) {
            record->left = tmp;
        }
        else
        {
            record->right = tmp;
        }
    }
    void Traversal() {      //inorder
        cout << "Inorder traversal:\n";
        treenode* tmp1 = new treenode;
        treenode* tmp2 = new treenode;
        tmp1 = leftmost(root);
        tmp2 = leftmost(root);
        cout << "key:    ";
        while (tmp1) {
            cout << setw(3) << tmp1->key_value.first;
            tmp1 = successor(tmp1);
        }
        cout << "\nvalue: ";
        while (tmp2) {
            cout << setw(3) << tmp2->key_value.second;
            tmp2 = successor(tmp2);
        }
        cout << "\n\n\n";
```



```
        cout << "\n\n\n";
    }
    void Delete(int key) {
        treenode* delete_node = Search(key);
        if (delete_node == NULL) {
            cout << "Key " << key << " not found.\n";
            return;
        }
        treenode* d = 0;
        treenode* d_child = 0;
        if (delete_node->left == NULL || delete_node->right == NULL) {
            d = delete_node;
        }
        else
        {
            d = successor(delete_node);
        }
        if (d->left) {
            d_child = d->left;
        }
        else {
            d_child = d->right;
        }
        if (d_child) {
            d_child->parent = d->parent;
        }
        if (d->parent == NULL) {
            this->root = d_child;
        }
        else if (d == d->parent->left) {
            d->parent->left = d_child;
```



```
            d_child = d->left;
        }
        else {
            d_child = d->right;
        }
        if (d_child) {
            d_child->parent = d->parent;
        }
        if (d->parent == NULL) {
            this->root = d_child;
        }
        else if (d == d->parent->left) {
            d->parent->left = d_child;
        }
        else {
            d->parent->right = d_child;
        }
        if (d != delete_node) {
            delete_node->key_value.first = d->key_value.first;
            delete_node->key_value.second = d->key_value.second;
        }

        delete d;
        d = 0;
        cout << "Key " << key << " has been deleted.\n";
    }
};

int main() {
    BST t
```

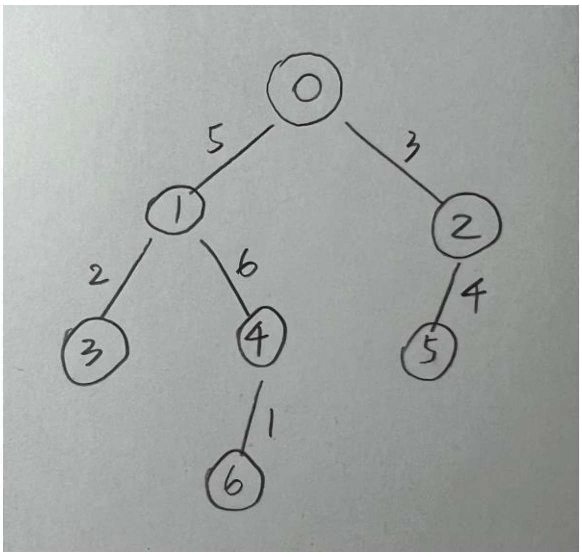


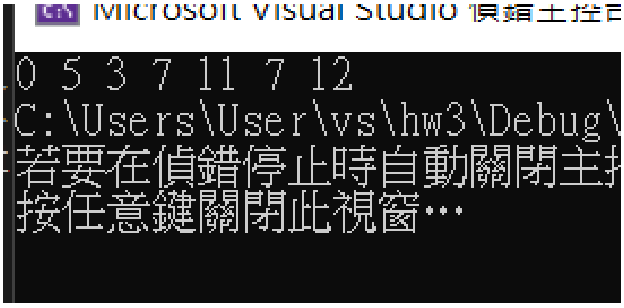
```
int main() {
    BST t;
    t.Insert(20, "a");
    t.Insert(10, "b");
    t.Insert(40, "c");
    t.Insert(6, "d");
    t.Insert(18, "e");
    t.Insert(30, "f");
    t.Insert(50, "g");
    t.Insert(8, "h");
    t.Insert(35, "i");

    t.Traversal();

    t.Delete(30);
    t.Traversal();

    t.Delete(25);
    t.Delete(20);
    t.Traversal();

}
```

```
Inorder traversal:
key:     6  8 10 18 20 30 35 40 50
value:   d  h  b  e  a  f  i  c  g


Key 30 has been deleted.
Inorder traversal:
key:     6  8 10 18 20 35 40 50
value:   d  h  b  e  a  i  c  g


Key 25 not found.
Key 20 has been deleted.
Inorder traversal:
key:     6  8 10 18 35 40 50
value:   d  h  b  e  i  c  g


C:\Users\User\vs\hw3\Debug\hw3.exe (處理序 1800) 已結束，
若要在偵錯停止時自動關閉主控台，請啟用 [工具] -> [選項] ->
按任意鍵關閉此視窗…
```

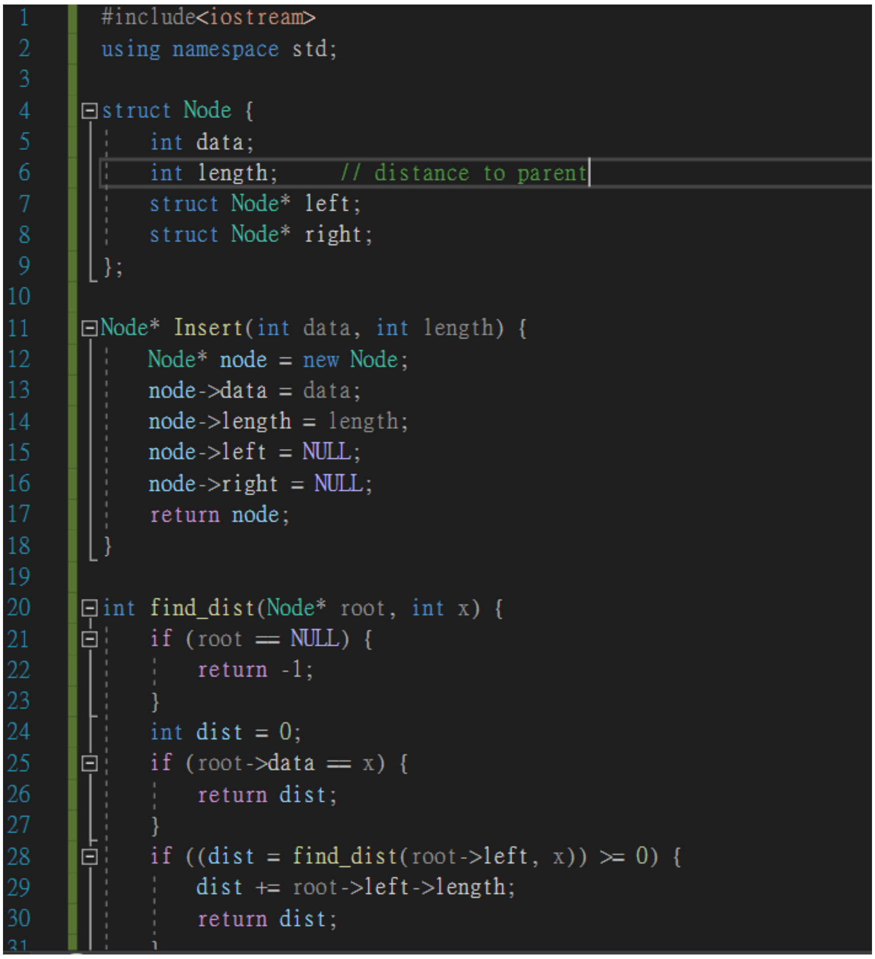
Delete runs for O(log(n)), that n stands for the number of vertices.

6. Shortest path from root

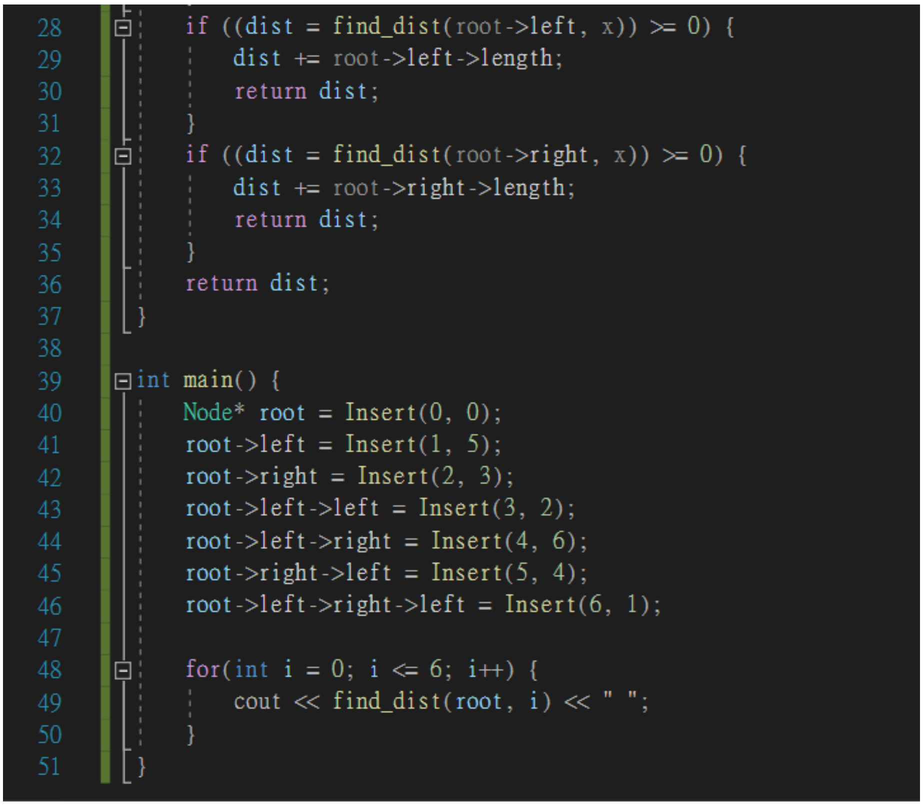




```
0 5 3 7 11 7 12
C:\Users\User\vs\hw3\Debug\
若要在偵錯停止時自動關閉主
按任意鍵關閉此視窗…
```



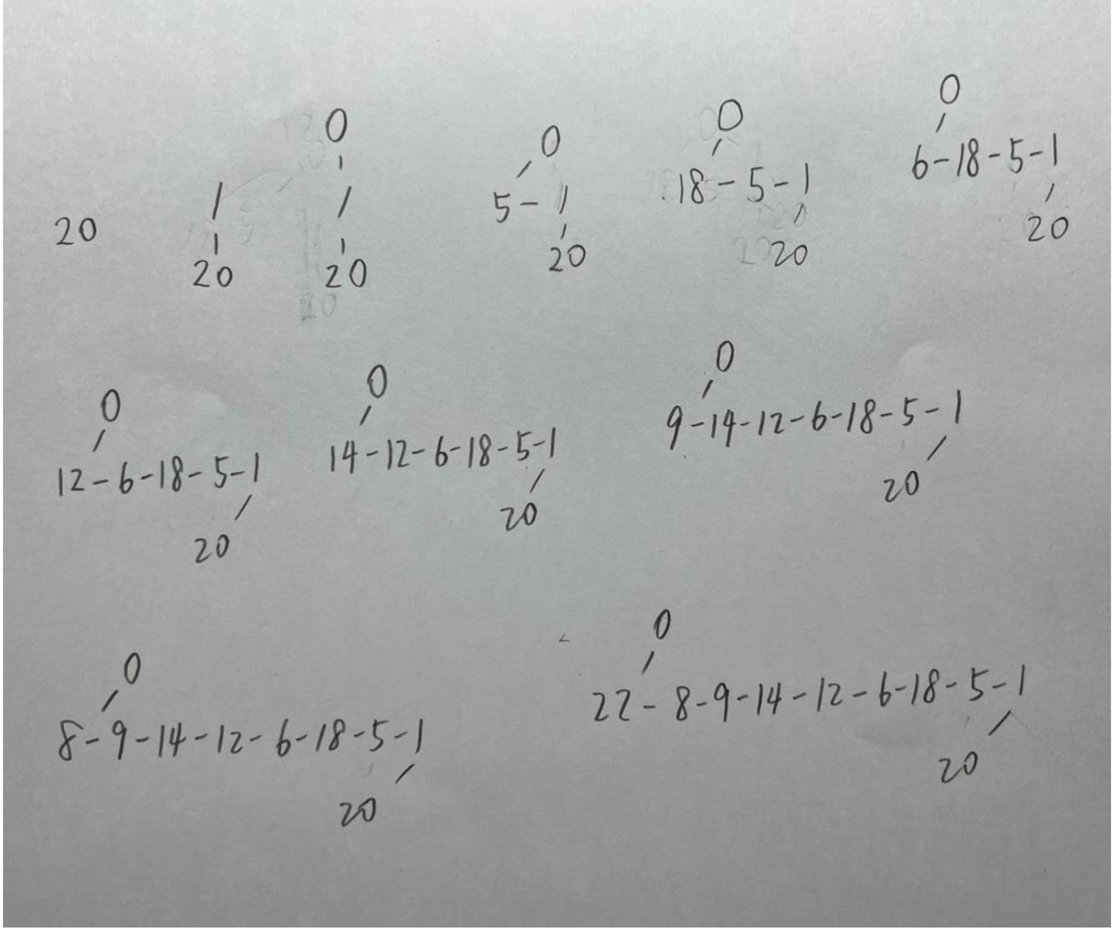
```cpp
#include<iostream>
using namespace std;

struct Node {
    int data;
    int length;     // distance to parent
    struct Node* left;
    struct Node* right;
};

Node* Insert(int data, int length) {
    Node* node = new Node;
    node->data = data;
    node->length = length;
    node->left = NULL;
    node->right = NULL;
    return node;
}

int find_dist(Node* root, int x) {
    if (root == NULL) {
        return -1;
    }
    int dist = 0;
    if (root->data == x) {
        return dist;
    }
    if ((dist = find_dist(root->left, x)) >= 0) {
        dist += root->left->length;
        return dist;
```



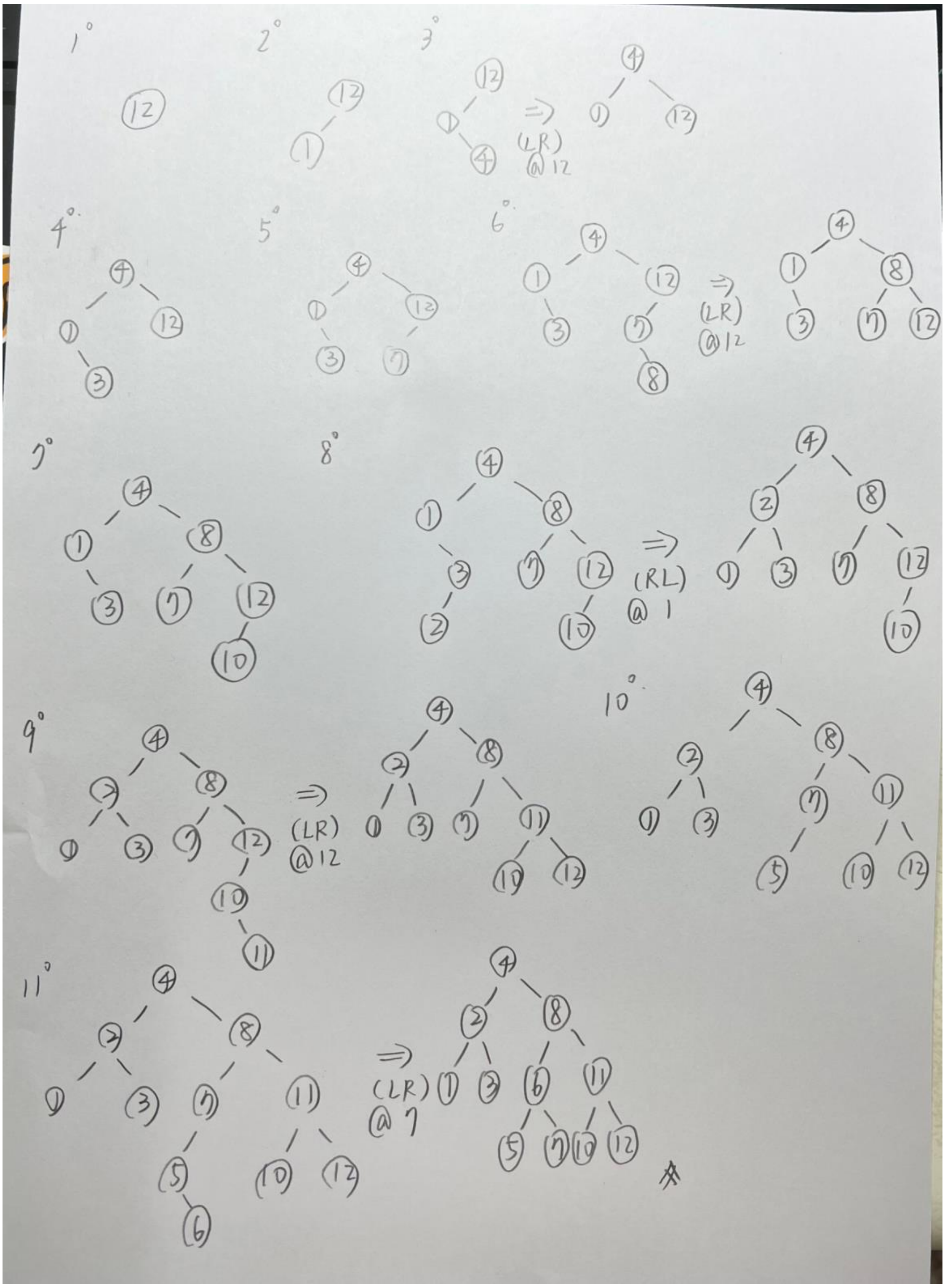
```cpp
    if ((dist = find_dist(root->left, x)) >= 0) {
        dist += root->left->length;
        return dist;
```

```cpp
    }
    if ((dist = find_dist(root->right, x)) >= 0) {
        dist += root->right->length;
        return dist;
    }
    return dist;
}

int main() {
    Node* root = Insert(0, 0);
    root->left = Insert(1, 5);
    root->right = Insert(2, 3);
    root->left->left = Insert(3, 2);
    root->left->right = Insert(4, 6);
    root->right->left = Insert(5, 4);
    root->left->right->left = Insert(6, 1);

    for(int i = 0; i <= 6; i++) {
        cout << find_dist(root, i) << " ";
    }
}
```

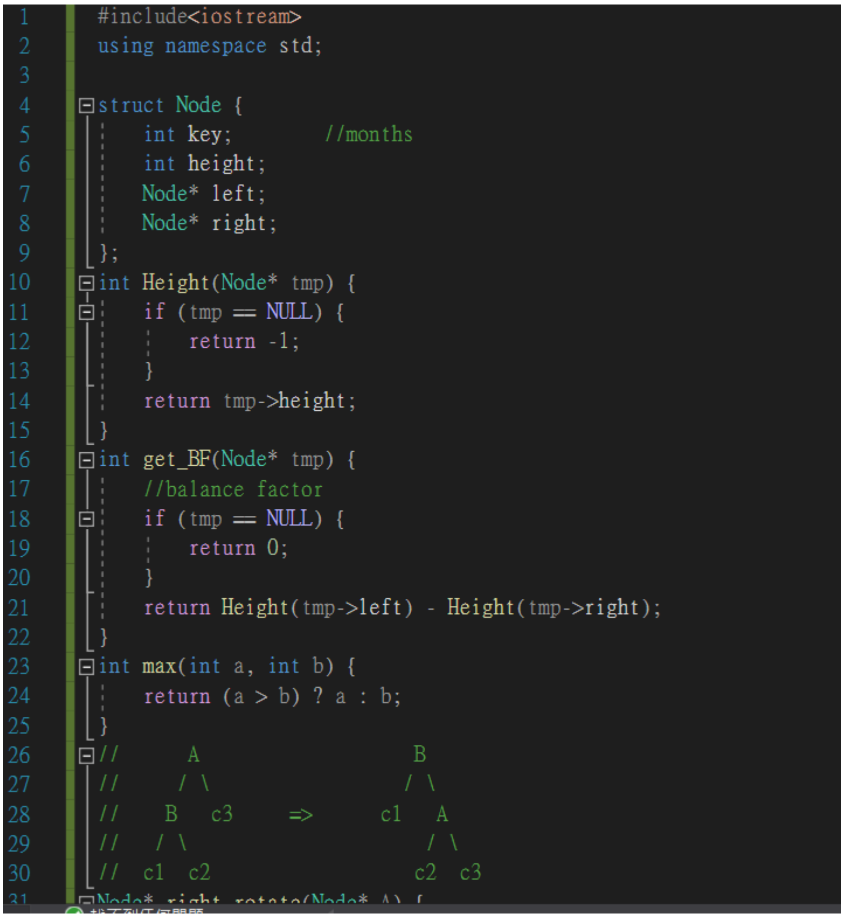
7. Min pairing heap



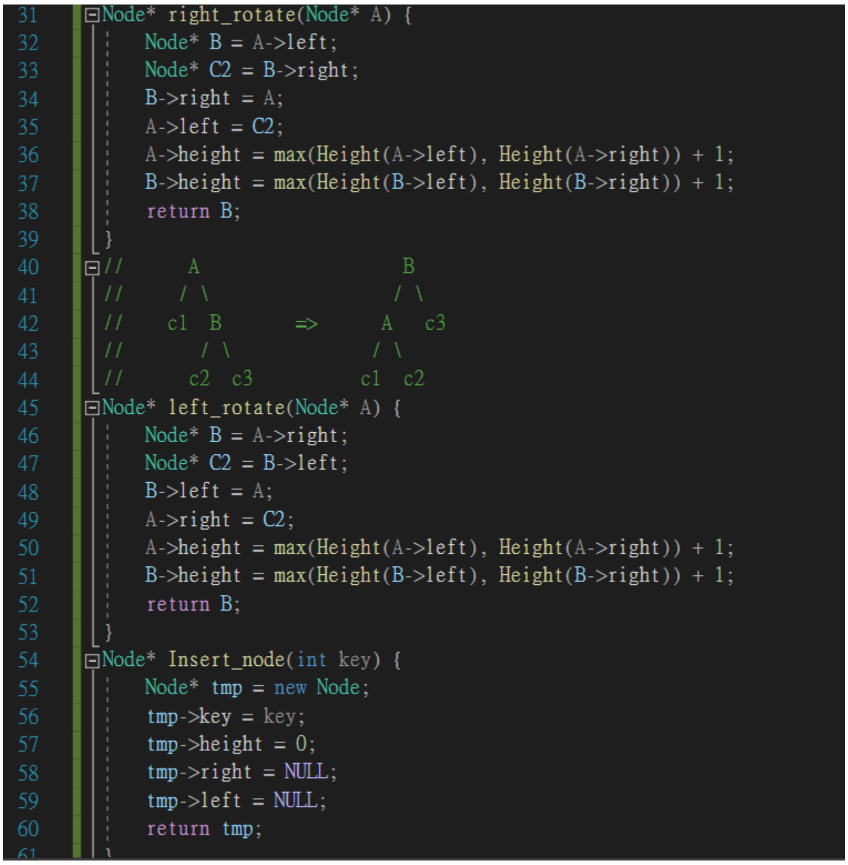
8. AVL tree


1°
12
2°
12
1
3°
12
1
4
⇒
(LR)
@12
4
1
12
4°
4
1
12
3
5°
4
1
12
3
7
6°
4
1
12
3
7
8
⇒
(LR)
@12
4
1
8
3
7
12
7°
4
1
8
3
7
12
10
8°
4
1
8
3
7
12
2
10
⇒
(RL)
@1
4
2
8
1
3
7
12
10
9°
4
2
8
1
3
7
12
10
11
⇒
(LR)
@12
4
2
8
1
3
7
11
10
12
10°
4
2
8
1
3
7
11
5
10
12
11°
4
2
8
1
3
7
11
5
10
12
6
⇒
(LR)
@7
4
2
8
1
3
6
11
5
7
10
12

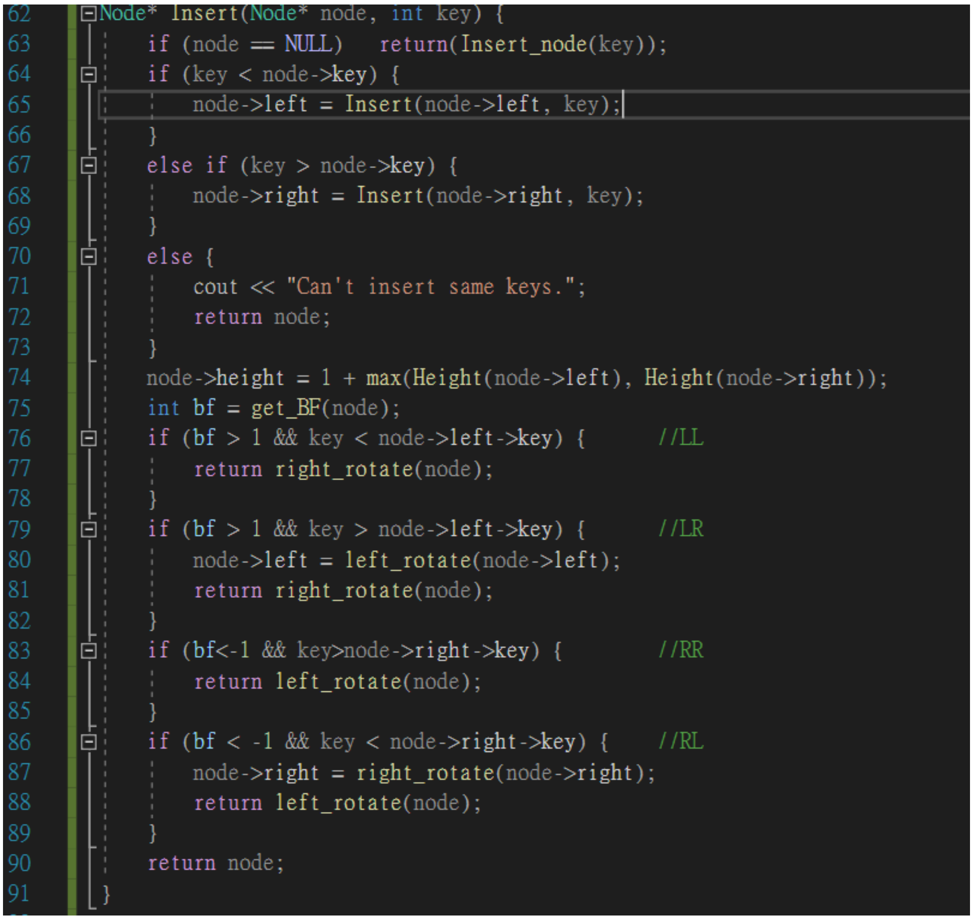
9. AVL tree



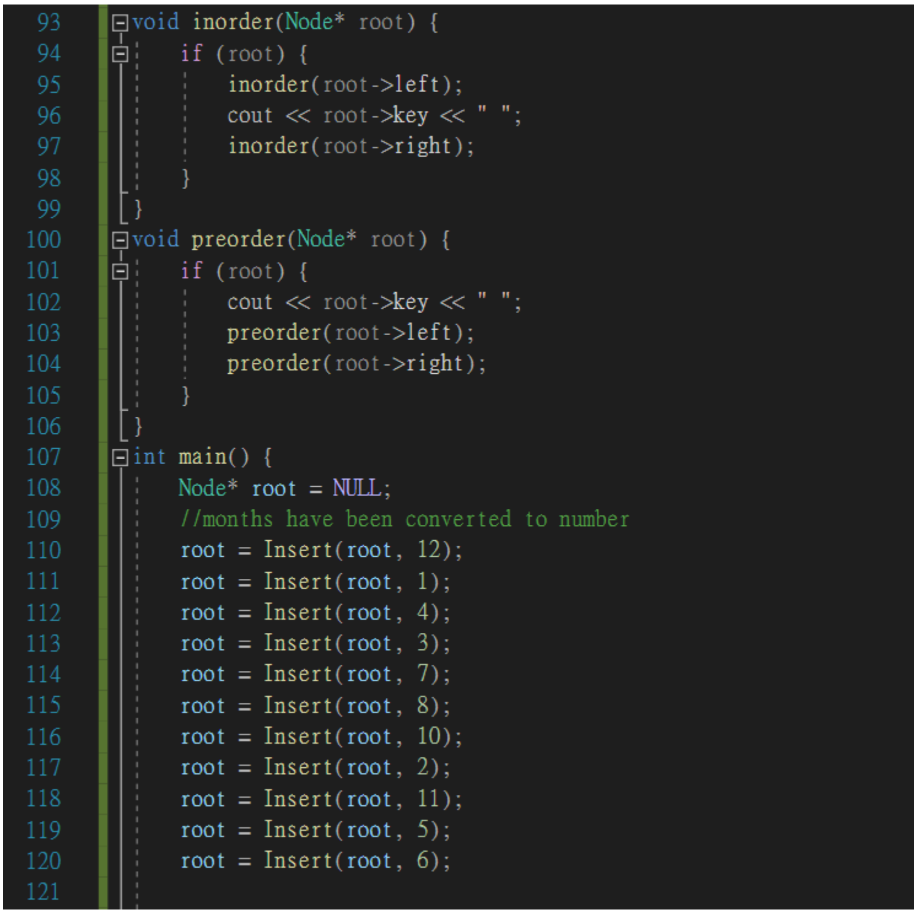
```cpp
#include<iostream>
using namespace std;

struct Node {
    int key;         //months
    int height;
    Node* left;
    Node* right;
};
int Height(Node* tmp) {
    if (tmp == NULL) {
        return -1;
    }
    return tmp->height;
}
int get_BF(Node* tmp) {
    //balance factor
    if (tmp == NULL) {
        return 0;
    }
    return Height(tmp->left) - Height(tmp->right);
}
int max(int a, int b) {
    return (a > b) ? a : b;
}
//      A                   B
//     / \                 / \
//    B   c3     =>      c1    A
//   / \                     / \
//  c1  c2                  c2  c3
```



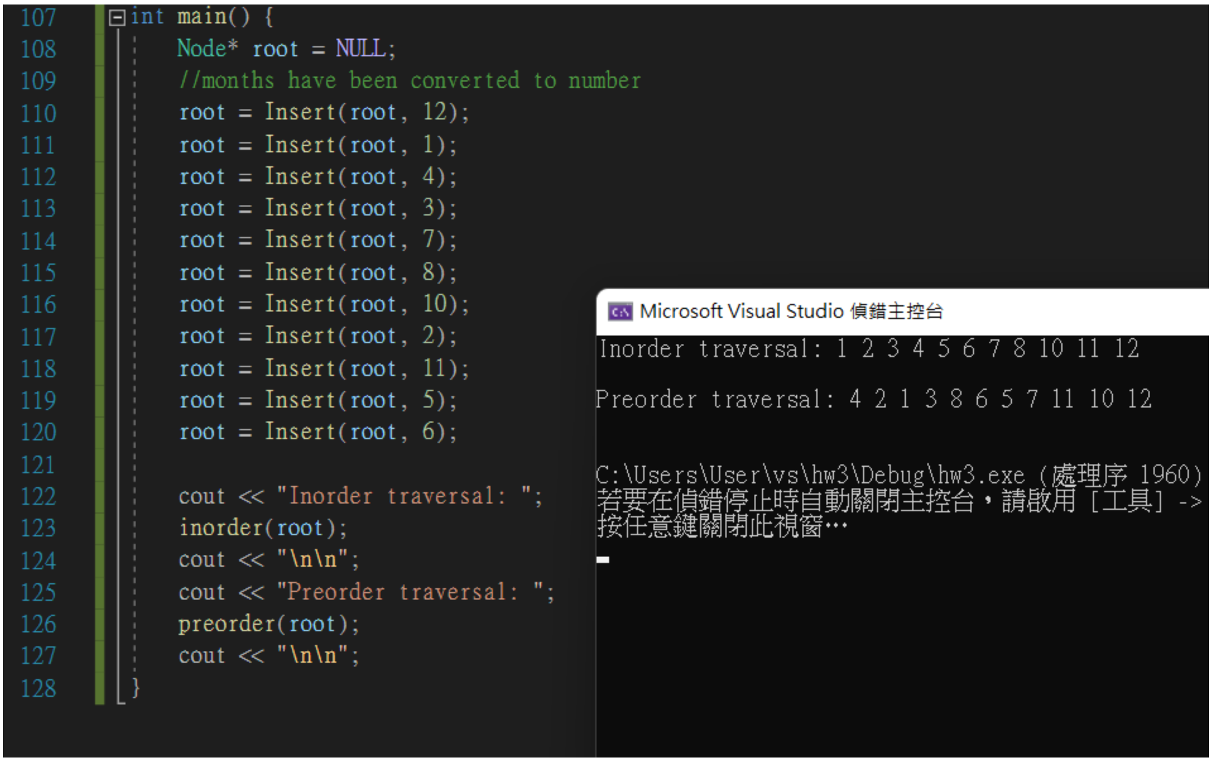
```cpp
Node* right_rotate(Node* A) {
    Node* B = A->left;
    Node* C2 = B->right;
    B->right = A;
    A->left = C2;
    A->height = max(Height(A->left), Height(A->right)) + 1;
    B->height = max(Height(B->left), Height(B->right)) + 1;
    return B;
}
//      A                   B
//     / \                 / \
//    c1  B       =>      A   c3
//       / \             / \
//      c2  c3          c1  c2
Node* left_rotate(Node* A) {
    Node* B = A->right;
    Node* C2 = B->left;
    B->left = A;
    A->right = C2;
    A->height = max(Height(A->left), Height(A->right)) + 1;
    B->height = max(Height(B->left), Height(B->right)) + 1;
    return B;
}
Node* Insert_node(int key) {
    Node* tmp = new Node;
    tmp->key = key;
    tmp->height = 0;
    tmp->right = NULL;
    tmp->left = NULL;
    return tmp;
}
```



```cpp
Node* Insert(Node* node, int key) {
    if (node == NULL)    return(Insert_node(key));
    if (key < node->key) {
        node->left = Insert(node->left, key);
    }
    else if (key > node->key) {
        node->right = Insert(node->right, key);
    }
    else {
        cout << "Can't insert same keys.";
        return node;
    }
    node->height = 1 + max(Height(node->left), Height(node->right));
    int bf = get_BF(node);
    if (bf > 1 && key < node->left->key) {       //LL
        return right_rotate(node);
    }
    if (bf > 1 && key > node->left->key) {       //LR
        node->left = left_rotate(node->left);
        return right_rotate(node);
    }
    if (bf<-1 && key>node->right->key) {         //RR
        return left_rotate(node);
    }
    if (bf < -1 && key < node->right->key) {     //RL
        node->right = right_rotate(node->right);
        return left_rotate(node);
    }
    return node;
}
```



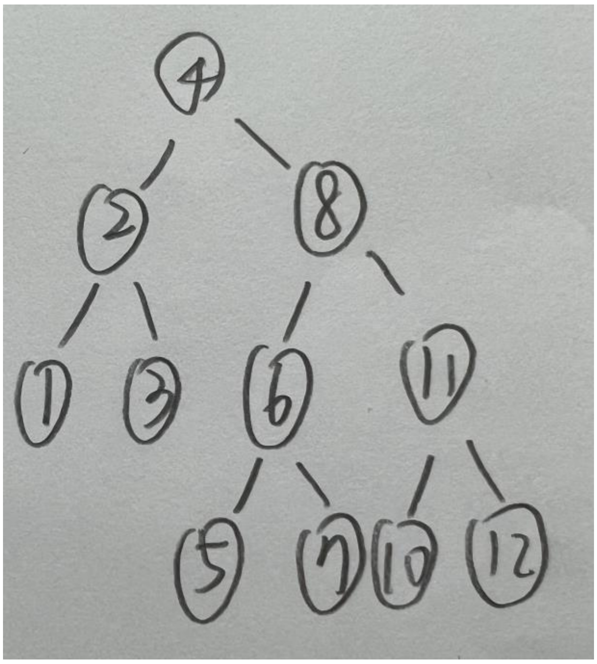
```cpp
void inorder(Node* root) {
    if (root) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}
void preorder(Node* root) {
    if (root) {
        cout << root->key << " ";
        preorder(root->left);
        preorder(root->right);
    }
}
int main() {
    Node* root = NULL;
    //months have been converted to number
    root = Insert(root, 12);
    root = Insert(root, 1);
    root = Insert(root, 4);
    root = Insert(root, 3);
    root = Insert(root, 7);
    root = Insert(root, 8);
    root = Insert(root, 10);
    root = Insert(root, 2);
    root = Insert(root, 11);
    root = Insert(root, 5);
    root = Insert(root, 6);

```



```
int main() {
    Node* root = NULL;
    //months have been converted to number
    root = Insert(root, 12);
    root = Insert(root, 1);
    root = Insert(root, 4);
    root = Insert(root, 3);
    root = Insert(root, 7);
    root = Insert(root, 8);
    root = Insert(root, 10);
    root = Insert(root, 2);
    root = Insert(root, 11);
    root = Insert(root, 5);
    root = Insert(root, 6);

    cout << "Inorder traversal: ";
    inorder(root);
    cout << "\n\n";
    cout << "Preorder traversal: ";
    preorder(root);
    cout << "\n\n";
}
```

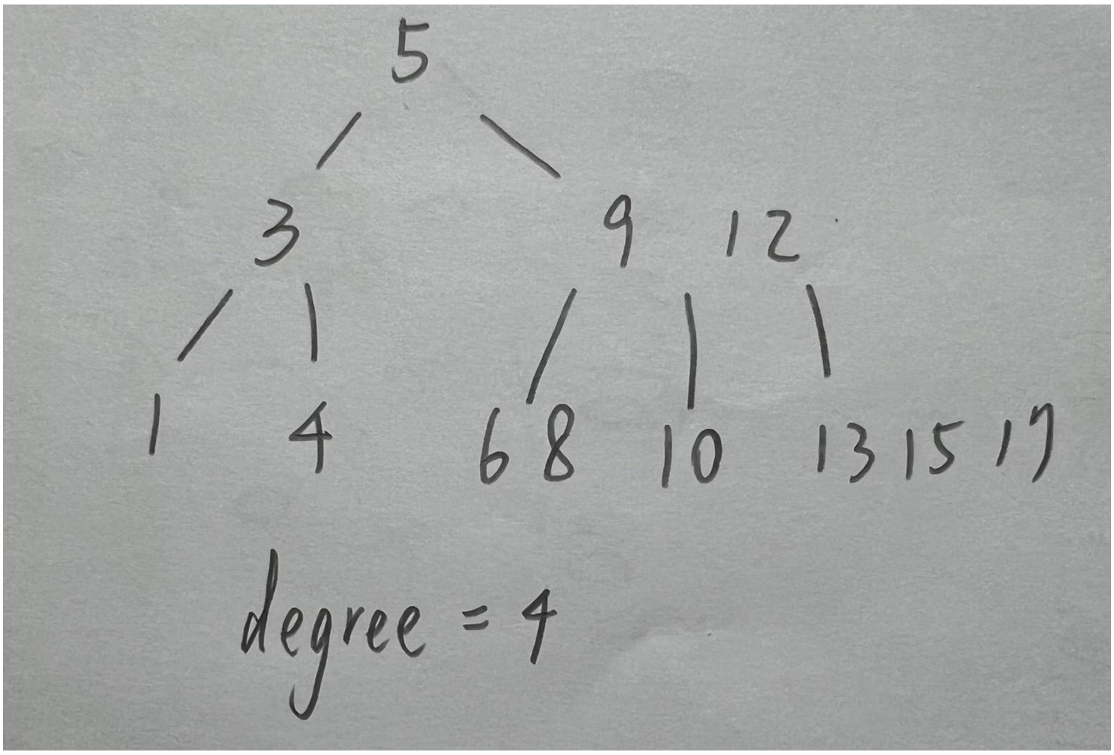
```
Inorder traversal: 1 2 3 4 5 6 7 8 10 11 12

Preorder traversal: 4 2 1 3 8 6 5 7 11 10 12


C:\Users\User\vs\hw3\Debug\hw3.exe (處理序 1960)
若要在偵錯停止時自動關閉主控台，請啟用 [工具] ->
按任意鍵關閉此視窗…
```

We can get the unique tree from inorder and preorder traversal, which constructs the tree below.

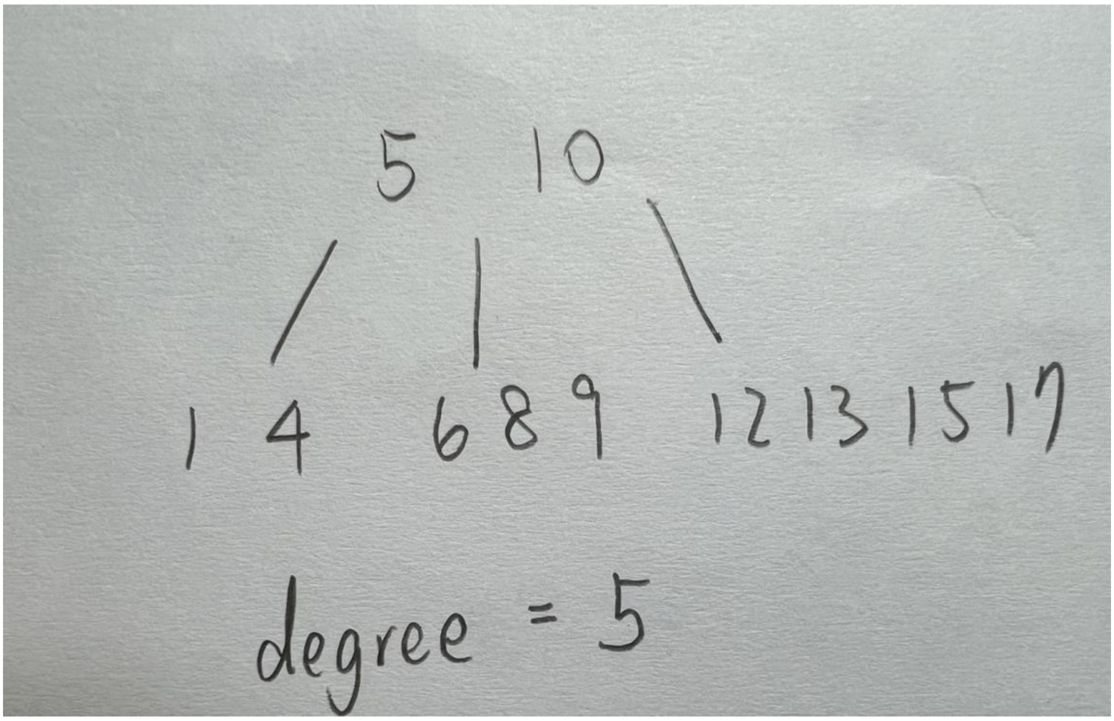


10. B-tree

Degree=4:



Degree=5:


5 10
1 4
6 8 9
12 13 15 17
degree = 5