

1. Bellmanford

```
1  #include<iostream>
2  #include<vector>
3  #include<list>
4  #include<utility>
5  #define MAX 100000
6
7  using namespace std;
8
9  class Graph {
10     int num_vertex;
11     vector<list<pair<int, int>>> adj;
12     vector<int> distance;
13 public:
14     Graph():num_vertex(0){};
15     Graph(int vertex) {
16         num_vertex = vertex;
17         adj.resize(vertex);
18     }
19     void add(int from, int to, int length) {
20         adj[from].push_back(make_pair(to, length));
21     }
22     void print_dist(vector<int> distance) {
23         for (int i = 0; i < num_vertex; i++) {
24             //x for can't arrive
25             if (distance[i] == MAX) cout << "x ";
26             else cout << distance[i] << " ";
27         }
28         cout << "\n\n";
29     }
30     void bellmanford(int start) {
31         //Initialize
32         distance.resize(num_vertex);
33         for (int i = 0; i < num_vertex; i++) {
34             distance[i] = MAX;
35         }
36         distance[start] = 0;
37         //Relax
38         for (int i = 0; i < num_vertex - 1; i++) {
39             for (int j = 0; j < num_vertex; j++) {
40                 for (list<pair<int, int>>::iterator it = adj[j].begin(); it != adj[j].end(); it++) {
41                     //j for from, first for to, second for length
42                     if (distance[*it].first > distance[j] + (*it).second) {
43                         distance[*it].first = distance[j] + (*it).second;
44                     }
45                 }
46             }
47         }
48         //Check negative cycle
49         for (int i = 0; i < num_vertex; i++) {
50             for (list<pair<int, int>>::iterator it = adj[i].begin(); it != adj[i].end(); it++) {
51                 // i for from, first for to, second for length
52                 if (distance[*it].first > distance[i] + (*it).second) {
53                     cout << "Negative cycle!\n";
54                     return;
55                 }
56             }
57         }
58         //Print
59         cout << "distance from vertex " << start << " to vertex 0 ~ " << num_vertex - 1 << ":\n";
60         print_dist(distance);
61         return;
62     }
```

```
64
65 int main() {
66     Graph g(4);
67     g.add(0, 1, 5);
68     g.add(0, 3, -1);
69     g.add(2, 0, 2);
70     g.add(2, 3, 4);
71     //case 1:
72     g.bellmanford(2);
73
74     //case 2:
75     g.bellmanford(0);
76
77     //case 3:
78     Graph g2(4);
79     g2.add(0, 1, 5);
80     g2.add(0, 3, -6);
81     g2.add(2, 0, 2);
82     g2.add(3, 2, 3);
83     g2.bellmanford(0);
84
85 }
```

Microsoft Visual Studio 偵錯主控台

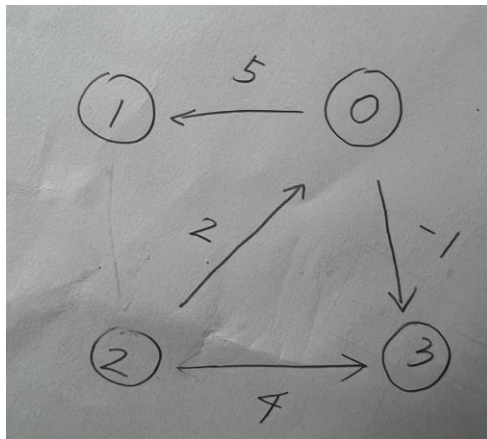
distance from vertex 2 to vertex 0 ~ 3:
2 7 0 1

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

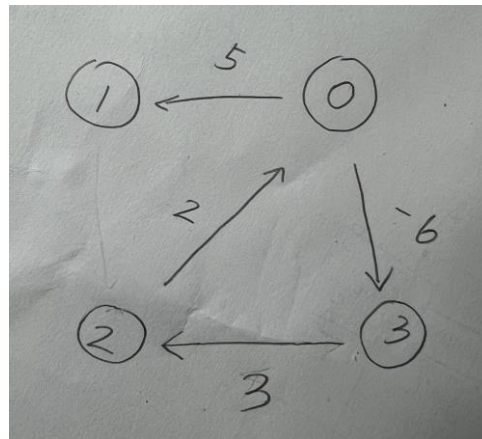
Negative cycle!

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:



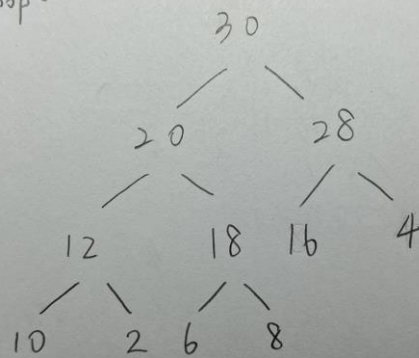
Case1: normal case

Case2: vertex 0 can't get to vertex 2

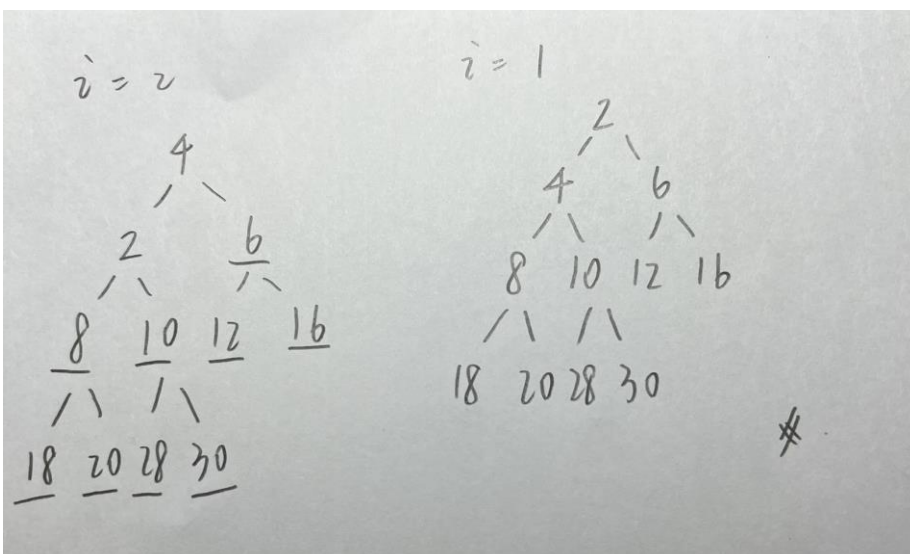
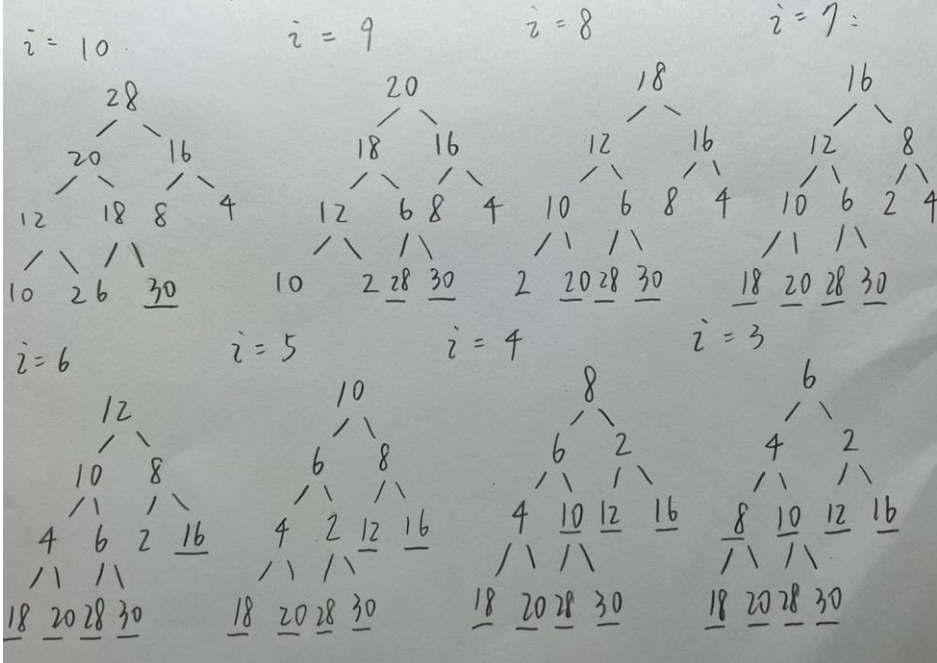
Case3: negative cycle

2. Heapify

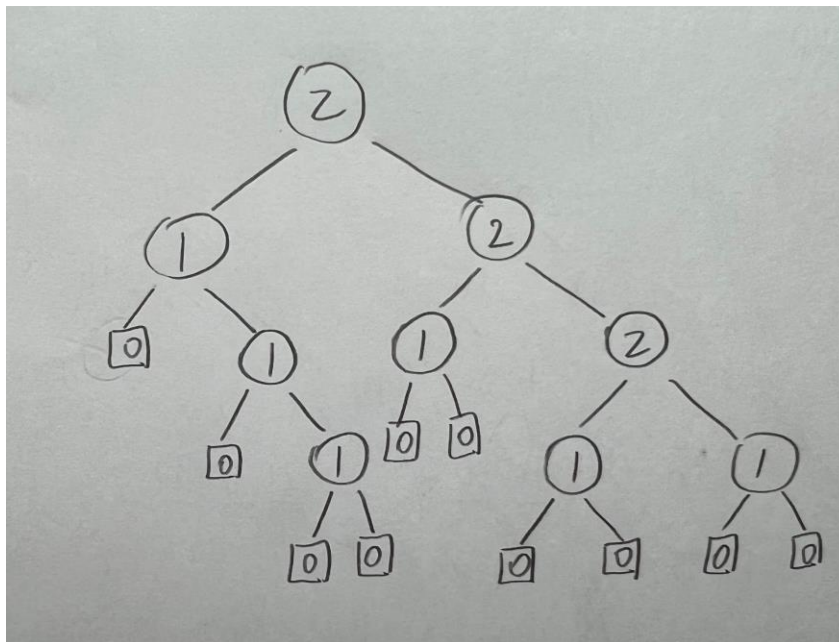
first for loop:



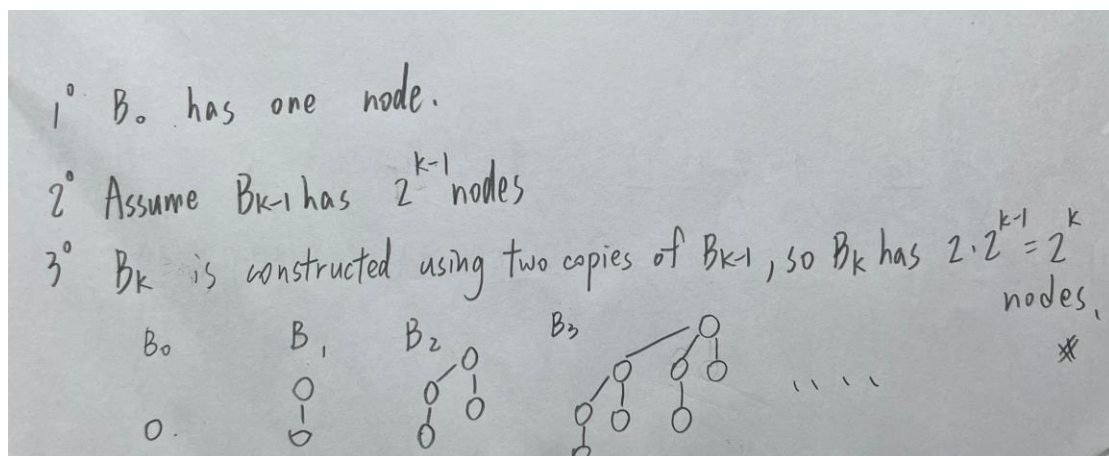
second for loop:



3. Shortest

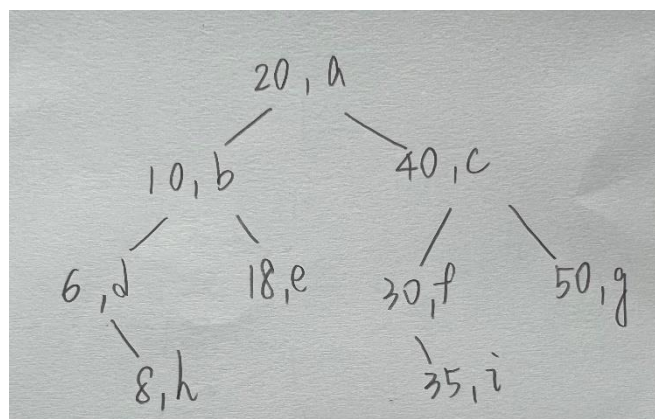


4. Binomial tree



5. BST delete

Original tree ->



```

1  #include<iostream>
2  #include<utility>
3  #include<iomanip>
4  using namespace std;
5
6  class treenode {
7  public:
8      pair<int, string> key_value;
9      treenode* left;
10     treenode* right;
11     treenode* parent;
12     treenode() {
13         left = NULL;
14         right = NULL;
15         parent = NULL;
16         key_value.first = NULL;
17         key_value.second = "";
18     }
19     treenode(int key, string value) {
20         left = NULL;
21         right = NULL;
22         parent = NULL;
23         key_value.first = key;
24         key_value.second = value;
25     }
26 };
27
28 class BST {
29 private:
30     treenode* root;
31

```

```

28 class BST {
29 private:
30     treenode* root;
31
32     treenode* leftmost(treenode* tmp) {
33         while (tmp->left) {
34             tmp = tmp->left;
35         }
36         return tmp;
37     }
38     treenode* successor(treenode* tmp) {
39         if (tmp->right) {
40             return leftmost(tmp->right);
41         }
42         treenode* new_node = tmp->parent;
43         while (new_node && tmp == new_node->right) {
44             tmp = new_node;
45             new_node = new_node->parent;
46         }
47         return new_node;
48     }
49
50 public:
51     BST() :root(0) {};
52
53     treenode* Search(int key) {
54         treenode* tmp = root;
55         while (tmp && key != tmp->key_value.first) {
56             if (key < tmp->key_value.first) {
57                 tmp = tmp->left;
58             }
59             else {
60                 tmp = tmp->right;
61             }
62         }
63         return tmp;
64     }
65
66     void insert(int key, string value) {
67         treenode* new_node = new treenode(key, value);
68         if (root == 0) {
69             root = new_node;
70         }
71         else {
72             treenode* tmp = root;
73             while (tmp->key_value.first != key) {
74                 if (key < tmp->key_value.first) {
75                     tmp = tmp->left;
76                 }
77                 else {
78                     tmp = tmp->right;
79                 }
80             }
81             if (tmp->key_value.first == key) {
82                 cout << "Duplicate key. Not inserted." << endl;
83             }
84             else {
85                 new_node->parent = tmp;
86                 if (key < tmp->key_value.first) {
87                     tmp->left = new_node;
88                 }
89                 else {
90                     tmp->right = new_node;
91                 }
92             }
93         }
94     }
95
96     void inorder() {
97         inorder(root);
98     }
99
100    void preorder() {
101        preorder(root);
102    }
103
104    void postorder() {
105        postorder(root);
106    }
107
108    void levelorder() {
109        levelorder(root);
110    }
111
112    void deleteNode(int key) {
113        deleteNode(root, key);
114    }
115
116    void deleteNode(treenode* root, int key) {
117        if (root == 0) {
118            return;
119        }
120        if (root->key_value.first == key) {
121            if (root->left == 0 && root->right == 0) {
122                root = 0;
123            }
124            else if (root->left != 0 && root->right == 0) {
125                root = root->left;
126            }
127            else if (root->left == 0 && root->right != 0) {
128                root = root->right;
129            }
130            else {
131                treenode* tmp = successor(root);
132                root->key_value.first = tmp->key_value.first;
133                root->key_value.second = tmp->key_value.second;
134                deleteNode(tmp, tmp->key_value.first);
135            }
136        }
137        else if (key < root->key_value.first) {
138            deleteNode(root->left, key);
139        }
140        else if (key > root->key_value.first) {
141            deleteNode(root->right, key);
142        }
143    }
144
145    void inorder(treenode* root) {
146        if (root == 0) {
147            return;
148        }
149        inorder(root->left);
150        cout << root->key_value.first << " " << root->key_value.second << endl;
151        inorder(root->right);
152    }
153
154    void preorder(treenode* root) {
155        if (root == 0) {
156            return;
157        }
158        cout << root->key_value.first << " " << root->key_value.second << endl;
159        preorder(root->left);
160        preorder(root->right);
161    }
162
163    void postorder(treenode* root) {
164        if (root == 0) {
165            return;
166        }
167        postorder(root->left);
168        postorder(root->right);
169        cout << root->key_value.first << " " << root->key_value.second << endl;
170    }
171
172    void levelorder(treenode* root) {
173        if (root == 0) {
174            return;
175        }
176        queue<treenode*> q;
177        q.push(root);
178        while (!q.empty()) {
179            treenode* tmp = q.front();
180            q.pop();
181            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
182            if (tmp->left != 0) {
183                q.push(tmp->left);
184            }
185            if (tmp->right != 0) {
186                q.push(tmp->right);
187            }
188        }
189    }
190
191    void levelorder_helper(treenode* root, int level) {
192        if (root == 0) {
193            return;
194        }
195        if (level == 0) {
196            cout << root->key_value.first << " " << root->key_value.second << endl;
197        }
198        levelorder_helper(root->left, level + 1);
199        levelorder_helper(root->right, level + 1);
200    }
201
202    void levelorder_helper(treenode* root, int level) {
203        if (root == 0) {
204            return;
205        }
206        if (level == 0) {
207            cout << root->key_value.first << " " << root->key_value.second << endl;
208        }
209        levelorder_helper(root->left, level + 1);
210        levelorder_helper(root->right, level + 1);
211    }
212
213    void deleteNode(treenode* root, int key) {
214        if (root == 0) {
215            return;
216        }
217        if (root->key_value.first == key) {
218            if (root->left == 0 && root->right == 0) {
219                root = 0;
220            }
221            else if (root->left != 0 && root->right == 0) {
222                root = root->left;
223            }
224            else if (root->left == 0 && root->right != 0) {
225                root = root->right;
226            }
227            else {
228                treenode* tmp = successor(root);
229                root->key_value.first = tmp->key_value.first;
230                root->key_value.second = tmp->key_value.second;
231                deleteNode(tmp, tmp->key_value.first);
232            }
233        }
234        else if (key < root->key_value.first) {
235            deleteNode(root->left, key);
236        }
237        else if (key > root->key_value.first) {
238            deleteNode(root->right, key);
239        }
240    }
241
242    void inorder(treenode* root) {
243        if (root == 0) {
244            return;
245        }
246        inorder(root->left);
247        cout << root->key_value.first << " " << root->key_value.second << endl;
248        inorder(root->right);
249    }
250
251    void preorder(treenode* root) {
252        if (root == 0) {
253            return;
254        }
255        cout << root->key_value.first << " " << root->key_value.second << endl;
256        preorder(root->left);
257        preorder(root->right);
258    }
259
260    void postorder(treenode* root) {
261        if (root == 0) {
262            return;
263        }
264        postorder(root->left);
265        postorder(root->right);
266        cout << root->key_value.first << " " << root->key_value.second << endl;
267    }
268
269    void levelorder(treenode* root) {
270        if (root == 0) {
271            return;
272        }
273        queue<treenode*> q;
274        q.push(root);
275        while (!q.empty()) {
276            treenode* tmp = q.front();
277            q.pop();
278            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
279            if (tmp->left != 0) {
280                q.push(tmp->left);
281            }
282            if (tmp->right != 0) {
283                q.push(tmp->right);
284            }
285        }
286    }
287
288    void levelorder_helper(treenode* root, int level) {
289        if (root == 0) {
290            return;
291        }
292        if (level == 0) {
293            cout << root->key_value.first << " " << root->key_value.second << endl;
294        }
295        levelorder_helper(root->left, level + 1);
296        levelorder_helper(root->right, level + 1);
297    }
298
299    void deleteNode(treenode* root, int key) {
300        if (root == 0) {
301            return;
302        }
303        if (root->key_value.first == key) {
304            if (root->left == 0 && root->right == 0) {
305                root = 0;
306            }
307            else if (root->left != 0 && root->right == 0) {
308                root = root->left;
309            }
310            else if (root->left == 0 && root->right != 0) {
311                root = root->right;
312            }
313            else {
314                treenode* tmp = successor(root);
315                root->key_value.first = tmp->key_value.first;
316                root->key_value.second = tmp->key_value.second;
317                deleteNode(tmp, tmp->key_value.first);
318            }
319        }
320        else if (key < root->key_value.first) {
321            deleteNode(root->left, key);
322        }
323        else if (key > root->key_value.first) {
324            deleteNode(root->right, key);
325        }
326    }
327
328    void inorder(treenode* root) {
329        if (root == 0) {
330            return;
331        }
332        inorder(root->left);
333        cout << root->key_value.first << " " << root->key_value.second << endl;
334        inorder(root->right);
335    }
336
337    void preorder(treenode* root) {
338        if (root == 0) {
339            return;
340        }
341        cout << root->key_value.first << " " << root->key_value.second << endl;
342        preorder(root->left);
343        preorder(root->right);
344    }
345
346    void postorder(treenode* root) {
347        if (root == 0) {
348            return;
349        }
350        postorder(root->left);
351        postorder(root->right);
352        cout << root->key_value.first << " " << root->key_value.second << endl;
353    }
354
355    void levelorder(treenode* root) {
356        if (root == 0) {
357            return;
358        }
359        queue<treenode*> q;
360        q.push(root);
361        while (!q.empty()) {
362            treenode* tmp = q.front();
363            q.pop();
364            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
365            if (tmp->left != 0) {
366                q.push(tmp->left);
367            }
368            if (tmp->right != 0) {
369                q.push(tmp->right);
370            }
371        }
372    }
373
374    void levelorder_helper(treenode* root, int level) {
375        if (root == 0) {
376            return;
377        }
378        if (level == 0) {
379            cout << root->key_value.first << " " << root->key_value.second << endl;
380        }
381        levelorder_helper(root->left, level + 1);
382        levelorder_helper(root->right, level + 1);
383    }
384
385    void deleteNode(treenode* root, int key) {
386        if (root == 0) {
387            return;
388        }
389        if (root->key_value.first == key) {
390            if (root->left == 0 && root->right == 0) {
391                root = 0;
392            }
393            else if (root->left != 0 && root->right == 0) {
394                root = root->left;
395            }
396            else if (root->left == 0 && root->right != 0) {
397                root = root->right;
398            }
399            else {
400                treenode* tmp = successor(root);
401                root->key_value.first = tmp->key_value.first;
402                root->key_value.second = tmp->key_value.second;
403                deleteNode(tmp, tmp->key_value.first);
404            }
405        }
406        else if (key < root->key_value.first) {
407            deleteNode(root->left, key);
408        }
409        else if (key > root->key_value.first) {
410            deleteNode(root->right, key);
411        }
412    }
413
414    void inorder(treenode* root) {
415        if (root == 0) {
416            return;
417        }
418        inorder(root->left);
419        cout << root->key_value.first << " " << root->key_value.second << endl;
420        inorder(root->right);
421    }
422
423    void preorder(treenode* root) {
424        if (root == 0) {
425            return;
426        }
427        cout << root->key_value.first << " " << root->key_value.second << endl;
428        preorder(root->left);
429        preorder(root->right);
430    }
431
432    void postorder(treenode* root) {
433        if (root == 0) {
434            return;
435        }
436        postorder(root->left);
437        postorder(root->right);
438        cout << root->key_value.first << " " << root->key_value.second << endl;
439    }
440
441    void levelorder(treenode* root) {
442        if (root == 0) {
443            return;
444        }
445        queue<treenode*> q;
446        q.push(root);
447        while (!q.empty()) {
448            treenode* tmp = q.front();
449            q.pop();
450            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
451            if (tmp->left != 0) {
452                q.push(tmp->left);
453            }
454            if (tmp->right != 0) {
455                q.push(tmp->right);
456            }
457        }
458    }
459
460    void levelorder_helper(treenode* root, int level) {
461        if (root == 0) {
462            return;
463        }
464        if (level == 0) {
465            cout << root->key_value.first << " " << root->key_value.second << endl;
466        }
467        levelorder_helper(root->left, level + 1);
468        levelorder_helper(root->right, level + 1);
469    }
470
471    void deleteNode(treenode* root, int key) {
472        if (root == 0) {
473            return;
474        }
475        if (root->key_value.first == key) {
476            if (root->left == 0 && root->right == 0) {
477                root = 0;
478            }
479            else if (root->left != 0 && root->right == 0) {
480                root = root->left;
481            }
482            else if (root->left == 0 && root->right != 0) {
483                root = root->right;
484            }
485            else {
486                treenode* tmp = successor(root);
487                root->key_value.first = tmp->key_value.first;
488                root->key_value.second = tmp->key_value.second;
489                deleteNode(tmp, tmp->key_value.first);
490            }
491        }
492        else if (key < root->key_value.first) {
493            deleteNode(root->left, key);
494        }
495        else if (key > root->key_value.first) {
496            deleteNode(root->right, key);
497        }
498    }
499
500    void inorder(treenode* root) {
501        if (root == 0) {
502            return;
503        }
504        inorder(root->left);
505        cout << root->key_value.first << " " << root->key_value.second << endl;
506        inorder(root->right);
507    }
508
509    void preorder(treenode* root) {
510        if (root == 0) {
511            return;
512        }
513        cout << root->key_value.first << " " << root->key_value.second << endl;
514        preorder(root->left);
515        preorder(root->right);
516    }
517
518    void postorder(treenode* root) {
519        if (root == 0) {
520            return;
521        }
522        postorder(root->left);
523        postorder(root->right);
524        cout << root->key_value.first << " " << root->key_value.second << endl;
525    }
526
527    void levelorder(treenode* root) {
528        if (root == 0) {
529            return;
530        }
531        queue<treenode*> q;
532        q.push(root);
533        while (!q.empty()) {
534            treenode* tmp = q.front();
535            q.pop();
536            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
537            if (tmp->left != 0) {
538                q.push(tmp->left);
539            }
540            if (tmp->right != 0) {
541                q.push(tmp->right);
542            }
543        }
544    }
545
546    void levelorder_helper(treenode* root, int level) {
547        if (root == 0) {
548            return;
549        }
550        if (level == 0) {
551            cout << root->key_value.first << " " << root->key_value.second << endl;
552        }
553        levelorder_helper(root->left, level + 1);
554        levelorder_helper(root->right, level + 1);
555    }
556
557    void deleteNode(treenode* root, int key) {
558        if (root == 0) {
559            return;
560        }
561        if (root->key_value.first == key) {
562            if (root->left == 0 && root->right == 0) {
563                root = 0;
564            }
565            else if (root->left != 0 && root->right == 0) {
566                root = root->left;
567            }
568            else if (root->left == 0 && root->right != 0) {
569                root = root->right;
570            }
571            else {
572                treenode* tmp = successor(root);
573                root->key_value.first = tmp->key_value.first;
574                root->key_value.second = tmp->key_value.second;
575                deleteNode(tmp, tmp->key_value.first);
576            }
577        }
578        else if (key < root->key_value.first) {
579            deleteNode(root->left, key);
580        }
581        else if (key > root->key_value.first) {
582            deleteNode(root->right, key);
583        }
584    }
585
586    void inorder(treenode* root) {
587        if (root == 0) {
588            return;
589        }
590        inorder(root->left);
591        cout << root->key_value.first << " " << root->key_value.second << endl;
592        inorder(root->right);
593    }
594
595    void preorder(treenode* root) {
596        if (root == 0) {
597            return;
598        }
599        cout << root->key_value.first << " " << root->key_value.second << endl;
600        preorder(root->left);
601        preorder(root->right);
602    }
603
604    void postorder(treenode* root) {
605        if (root == 0) {
606            return;
607        }
608        postorder(root->left);
609        postorder(root->right);
610        cout << root->key_value.first << " " << root->key_value.second << endl;
611    }
612
613    void levelorder(treenode* root) {
614        if (root == 0) {
615            return;
616        }
617        queue<treenode*> q;
618        q.push(root);
619        while (!q.empty()) {
620            treenode* tmp = q.front();
621            q.pop();
622            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
623            if (tmp->left != 0) {
624                q.push(tmp->left);
625            }
626            if (tmp->right != 0) {
627                q.push(tmp->right);
628            }
629        }
630    }
631
632    void levelorder_helper(treenode* root, int level) {
633        if (root == 0) {
634            return;
635        }
636        if (level == 0) {
637            cout << root->key_value.first << " " << root->key_value.second << endl;
638        }
639        levelorder_helper(root->left, level + 1);
640        levelorder_helper(root->right, level + 1);
641    }
642
643    void deleteNode(treenode* root, int key) {
644        if (root == 0) {
645            return;
646        }
647        if (root->key_value.first == key) {
648            if (root->left == 0 && root->right == 0) {
649                root = 0;
650            }
651            else if (root->left != 0 && root->right == 0) {
652                root = root->left;
653            }
654            else if (root->left == 0 && root->right != 0) {
655                root = root->right;
656            }
657            else {
658                treenode* tmp = successor(root);
659                root->key_value.first = tmp->key_value.first;
660                root->key_value.second = tmp->key_value.second;
661                deleteNode(tmp, tmp->key_value.first);
662            }
663        }
664        else if (key < root->key_value.first) {
665            deleteNode(root->left, key);
666        }
667        else if (key > root->key_value.first) {
668            deleteNode(root->right, key);
669        }
670    }
671
672    void inorder(treenode* root) {
673        if (root == 0) {
674            return;
675        }
676        inorder(root->left);
677        cout << root->key_value.first << " " << root->key_value.second << endl;
678        inorder(root->right);
679    }
680
681    void preorder(treenode* root) {
682        if (root == 0) {
683            return;
684        }
685        cout << root->key_value.first << " " << root->key_value.second << endl;
686        preorder(root->left);
687        preorder(root->right);
688    }
689
690    void postorder(treenode* root) {
691        if (root == 0) {
692            return;
693        }
694        postorder(root->left);
695        postorder(root->right);
696        cout << root->key_value.first << " " << root->key_value.second << endl;
697    }
698
699    void levelorder(treenode* root) {
700        if (root == 0) {
701            return;
702        }
703        queue<treenode*> q;
704        q.push(root);
705        while (!q.empty()) {
706            treenode* tmp = q.front();
707            q.pop();
708            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
709            if (tmp->left != 0) {
710                q.push(tmp->left);
711            }
712            if (tmp->right != 0) {
713                q.push(tmp->right);
714            }
715        }
716    }
717
718    void levelorder_helper(treenode* root, int level) {
719        if (root == 0) {
720            return;
721        }
722        if (level == 0) {
723            cout << root->key_value.first << " " << root->key_value.second << endl;
724        }
725        levelorder_helper(root->left, level + 1);
726        levelorder_helper(root->right, level + 1);
727    }
728
729    void deleteNode(treenode* root, int key) {
730        if (root == 0) {
731            return;
732        }
733        if (root->key_value.first == key) {
734            if (root->left == 0 && root->right == 0) {
735                root = 0;
736            }
737            else if (root->left != 0 && root->right == 0) {
738                root = root->left;
739            }
740            else if (root->left == 0 && root->right != 0) {
741                root = root->right;
742            }
743            else {
744                treenode* tmp = successor(root);
745                root->key_value.first = tmp->key_value.first;
746                root->key_value.second = tmp->key_value.second;
747                deleteNode(tmp, tmp->key_value.first);
748            }
749        }
750        else if (key < root->key_value.first) {
751            deleteNode(root->left, key);
752        }
753        else if (key > root->key_value.first) {
754            deleteNode(root->right, key);
755        }
756    }
757
758    void inorder(treenode* root) {
759        if (root == 0) {
760            return;
761        }
762        inorder(root->left);
763        cout << root->key_value.first << " " << root->key_value.second << endl;
764        inorder(root->right);
765    }
766
767    void preorder(treenode* root) {
768        if (root == 0) {
769            return;
770        }
771        cout << root->key_value.first << " " << root->key_value.second << endl;
772        preorder(root->left);
773        preorder(root->right);
774    }
775
776    void postorder(treenode* root) {
777        if (root == 0) {
778            return;
779        }
780        postorder(root->left);
781        postorder(root->right);
782        cout << root->key_value.first << " " << root->key_value.second << endl;
783    }
784
785    void levelorder(treenode* root) {
786        if (root == 0) {
787            return;
788        }
789        queue<treenode*> q;
790        q.push(root);
791        while (!q.empty()) {
792            treenode* tmp = q.front();
793            q.pop();
794            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
795            if (tmp->left != 0) {
796                q.push(tmp->left);
797            }
798            if (tmp->right != 0) {
799                q.push(tmp->right);
800            }
801        }
802    }
803
804    void levelorder_helper(treenode* root, int level) {
805        if (root == 0) {
806            return;
807        }
808        if (level == 0) {
809            cout << root->key_value.first << " " << root->key_value.second << endl;
810        }
811        levelorder_helper(root->left, level + 1);
812        levelorder_helper(root->right, level + 1);
813    }
814
815    void deleteNode(treenode* root, int key) {
816        if (root == 0) {
817            return;
818        }
819        if (root->key_value.first == key) {
820            if (root->left == 0 && root->right == 0) {
821                root = 0;
822            }
823            else if (root->left != 0 && root->right == 0) {
824                root = root->left;
825            }
826            else if (root->left == 0 && root->right != 0) {
827                root = root->right;
828            }
829            else {
830                treenode* tmp = successor(root);
831                root->key_value.first = tmp->key_value.first;
832                root->key_value.second = tmp->key_value.second;
833                deleteNode(tmp, tmp->key_value.first);
834            }
835        }
836        else if (key < root->key_value.first) {
837            deleteNode(root->left, key);
838        }
839        else if (key > root->key_value.first) {
840            deleteNode(root->right, key);
841        }
842    }
843
844    void inorder(treenode* root) {
845        if (root == 0) {
846            return;
847        }
848        inorder(root->left);
849        cout << root->key_value.first << " " << root->key_value.second << endl;
850        inorder(root->right);
851    }
852
853    void preorder(treenode* root) {
854        if (root == 0) {
855            return;
856        }
857        cout << root->key_value.first << " " << root->key_value.second << endl;
858        preorder(root->left);
859        preorder(root->right);
860    }
861
862    void postorder(treenode* root) {
863        if (root == 0) {
864            return;
865        }
866        postorder(root->left);
867        postorder(root->right);
868        cout << root->key_value.first << " " << root->key_value.second << endl;
869    }
870
871    void levelorder(treenode* root) {
872        if (root == 0) {
873            return;
874        }
875        queue<treenode*> q;
876        q.push(root);
877        while (!q.empty()) {
878            treenode* tmp = q.front();
879            q.pop();
880            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
881            if (tmp->left != 0) {
882                q.push(tmp->left);
883            }
884            if (tmp->right != 0) {
885                q.push(tmp->right);
886            }
887        }
888    }
889
890    void levelorder_helper(treenode* root, int level) {
891        if (root == 0) {
892            return;
893        }
894        if (level == 0) {
895            cout << root->key_value.first << " " << root->key_value.second << endl;
896        }
897        levelorder_helper(root->left, level + 1);
898        levelorder_helper(root->right, level + 1);
899    }
900
901    void deleteNode(treenode* root, int key) {
902        if (root == 0) {
903            return;
904        }
905        if (root->key_value.first == key) {
906            if (root->left == 0 && root->right == 0) {
907                root = 0;
908            }
909            else if (root->left != 0 && root->right == 0) {
910                root = root->left;
911            }
912            else if (root->left == 0 && root->right != 0) {
913                root = root->right;
914            }
915            else {
916                treenode* tmp = successor(root);
917                root->key_value.first = tmp->key_value.first;
918                root->key_value.second = tmp->key_value.second;
919                deleteNode(tmp, tmp->key_value.first);
920            }
921        }
922        else if (key < root->key_value.first) {
923            deleteNode(root->left, key);
924        }
925        else if (key > root->key_value.first) {
926            deleteNode(root->right, key);
927        }
928    }
929
930    void inorder(treenode* root) {
931        if (root == 0) {
932            return;
933        }
934        inorder(root->left);
935        cout << root->key_value.first << " " << root->key_value.second << endl;
936        inorder(root->right);
937    }
938
939    void preorder(treenode* root) {
940        if (root == 0) {
941            return;
942        }
943        cout << root->key_value.first << " " << root->key_value.second << endl;
944        preorder(root->left);
945        preorder(root->right);
946    }
947
948    void postorder(treenode* root) {
949        if (root == 0) {
950            return;
951        }
952        postorder(root->left);
953        postorder(root->right);
954        cout << root->key_value.first << " " << root->key_value.second << endl;
955    }
956
957    void levelorder(treenode* root) {
958        if (root == 0) {
959            return;
960        }
961        queue<treenode*> q;
962        q.push(root);
963        while (!q.empty()) {
964            treenode* tmp = q.front();
965            q.pop();
966            cout << tmp->key_value.first << " " << tmp->key_value.second << endl;
967            if (tmp->left != 0) {
968                q.push(tmp->left);
969            }
970            if (tmp->right != 0) {
971                q.push(tmp->right);
972            }
973        }
974    }
975
976    void levelorder_helper(treenode* root
```

```

53:     treenode* Search(int key) {
54:         treenode* tmp = root;
55:         while (tmp && key != tmp->key_value.first) {
56:             if (key < tmp->key_value.first) {
57:                 tmp = tmp->left;
58:             }
59:             else {
60:                 tmp = tmp->right;
61:             }
62:         }
63:         return tmp;
64:     }
65: void Insert(int key, string value) {
66:     treenode* record = 0;
67:     treenode* finder = 0;
68:     treenode* tmp = new treenode(key, value);
69:
70:     finder = root;
71:     while (finder) {
72:         record = finder;
73:         if (tmp->key_value.first < finder->key_value.first) {
74:             finder = finder->left;
75:         }
76:         else
77:         {
78:             finder = finder->right;
79:         }
80:     }
81:     tmp->parent = record;
82:     if (!record) {

```

```

79:     }
80: }
81: tmp->parent = record;
82: if (!record) {
83:     this->root = tmp;
84: }
85: else if (tmp->key_value.first < record->key_value.first) {
86:     record->left = tmp;
87: }
88: else
89: {
90:     record->right = tmp;
91: }
92: }
93: void Traversal() { //inorder
94:     cout << "Inorder traversal:\n";
95:     treenode* tmp1 = new treenode;
96:     treenode* tmp2 = new treenode;
97:     tmp1 = leftmost(root);
98:     tmp2 = leftmost(root);
99:     cout << "key: ";
100:    while (tmp1) {
101:        cout << setw(3) << tmp1->key_value.first;
102:        tmp1 = successor(tmp1);
103:    }
104:    cout << "\nvalue: ";
105:    while (tmp2) {
106:        cout << setw(3) << tmp2->key_value.second;
107:        tmp2 = successor(tmp2);
108:    }
109:    cout << "\n\n";

```

```

109         cout << "\n\n\n";
110     }
111     void Delete(int key) {
112         treenode* delete_node = Search(key);
113         if (delete_node == NULL) {
114             cout << "Key " << key << " not found.\n";
115             return;
116         }
117         treenode* d = 0;
118         treenode* d_child = 0;
119         if (delete_node->left == NULL || delete_node->right == NULL) {
120             d = delete_node;
121         }
122         else
123         {
124             d = successor(delete_node);
125         }
126         if (d->left) {
127             d_child = d->left;
128         }
129         else {
130             d_child = d->right;
131         }
132         if (d_child) {
133             d_child->parent = d->parent;
134         }
135         if (d->parent == NULL) {
136             this->root = d_child;
137         }
138         else if (d == d->parent->left) {
139             d->parent->left = d_child;

```

找不到任何問題

```

27         d_child = d->right;
28     }
29     else {
30         d_child = d->right;
31     }
32     if (d_child) {
33         d_child->parent = d->parent;
34     }
35     if (d->parent == NULL) {
36         this->root = d_child;
37     }
38     else if (d == d->parent->left) {
39         d->parent->left = d_child;
40     }
41     else {
42         d->parent->right = d_child;
43     }
44     if (d != delete_node) {
45         delete_node->key_value.first = d->key_value.first;
46         delete_node->key_value.second = d->key_value.second;
47     }
48
49     delete d;
50     d = 0;
51     cout << "Key " << key << " has been deleted.\n";
52 }
53 };
54
55 int main() {

```



```

154
155 int main() {
156     BST t;
157     t.Insert(20, "a");
158     t.Insert(10, "b");
159     t.Insert(40, "c");
160     t.Insert(6, "d");
161     t.Insert(18, "e");
162     t.Insert(30, "f");
163     t.Insert(50, "g");
164     t.Insert(8, "h");
165     t.Insert(35, "i");
166
167     t.Traversal();
168
169     t.Delete(30);
170     t.Traversal();
171
172     t.Delete(25);
173     t.Delete(20);
174     t.Traversal();
175
176 }

```

Microsoft Visual Studio 偵錯主控台

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

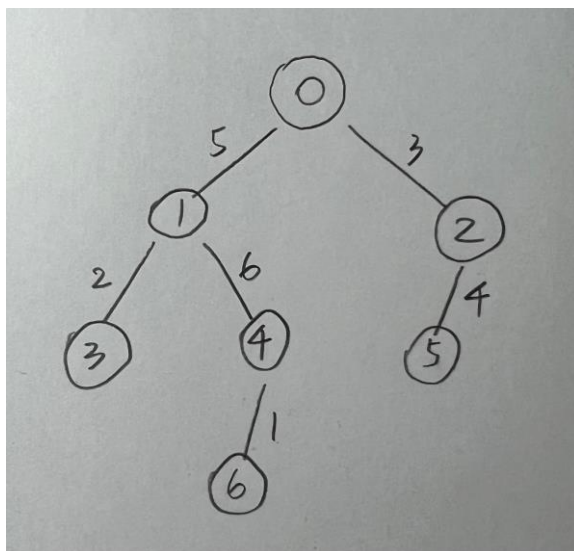
Key 30 has been deleted.
Inorder traversal:
key: 6 8 10 18 20 35 40 50
value: d h b e a i c g

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



Microsoft Visual Studio 偵錯主控台

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控台，請啟用 [工具] -> [選項] ->
按任意鍵關閉此視窗...


```

1  #include<iostream>
2  using namespace std;
3
4  struct Node {
5      int data;
6      int length;    // distance to parent
7      struct Node* left;
8      struct Node* right;
9  };
10
11 Node* Insert(int data, int length) {
12     Node* node = new Node;
13     node->data = data;
14     node->length = length;
15     node->left = NULL;
16     node->right = NULL;
17     return node;
18 }
19
20 int find_dist(Node* root, int x) {
21     if (root == NULL) {
22         return -1;
23     }
24     int dist = 0;
25     if (root->data == x) {
26         return dist;
27     }
28     if ((dist = find_dist(root->left, x)) >= 0) {
29         dist += root->left->length;
30         return dist;
31     }

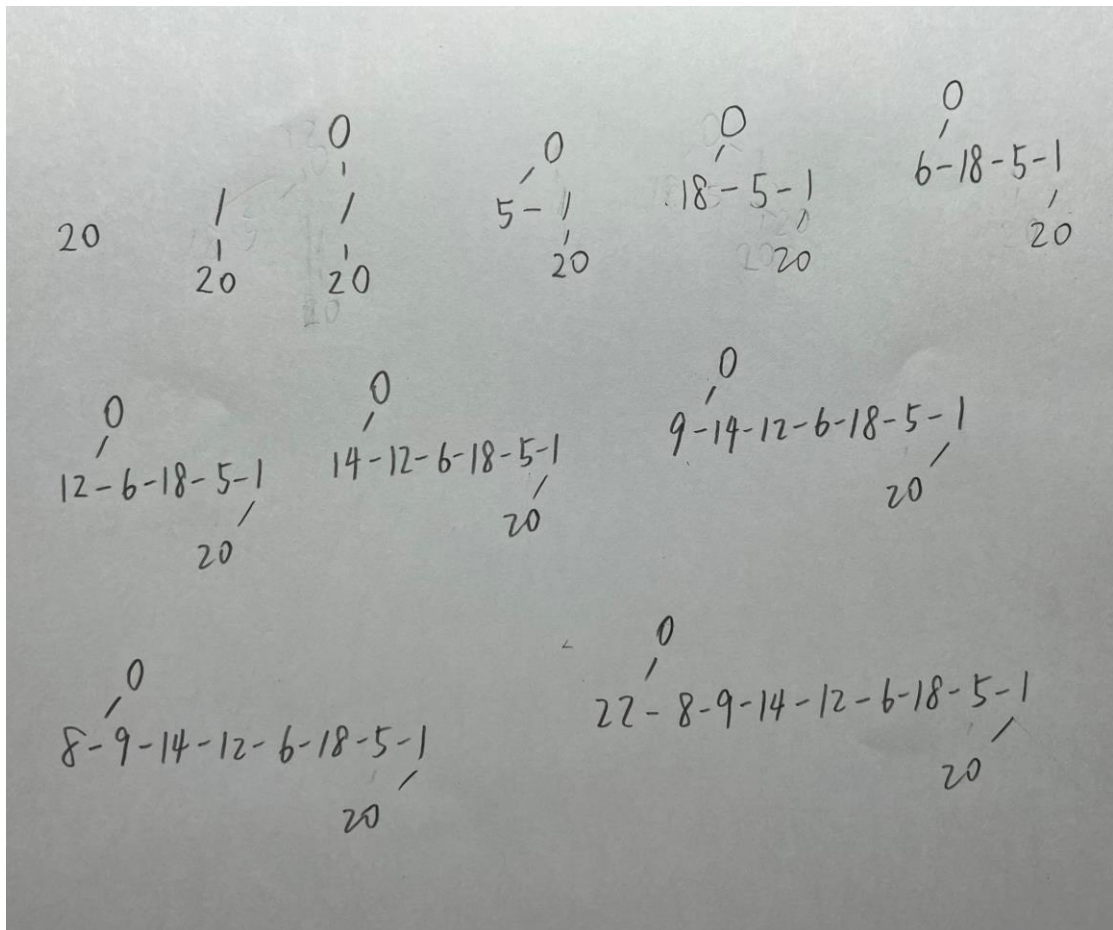
```

```

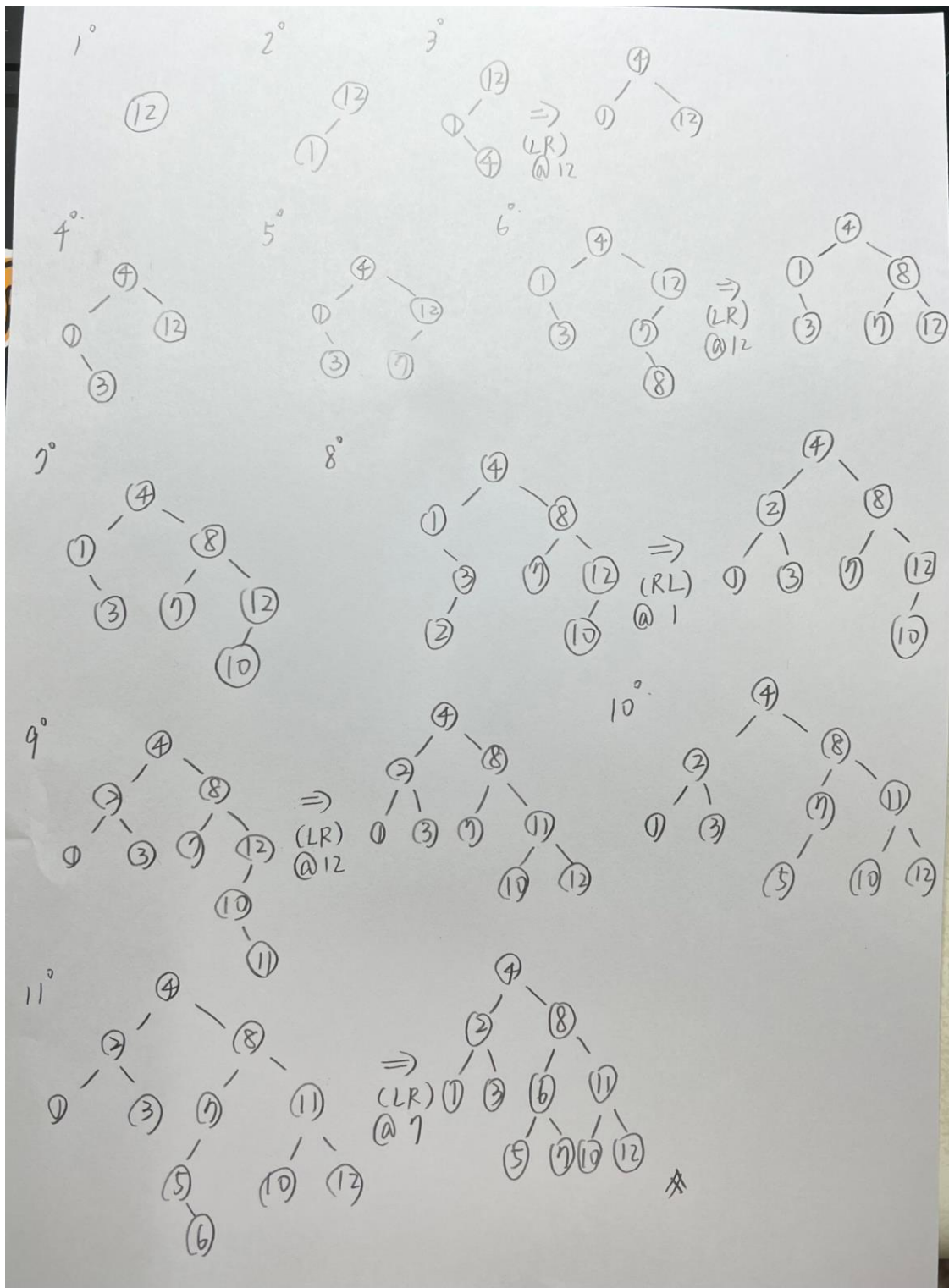
28     if ((dist = find_dist(root->left, x)) >= 0) {
29         dist += root->left->length;
30         return dist;
31     }
32     if ((dist = find_dist(root->right, x)) >= 0) {
33         dist += root->right->length;
34         return dist;
35     }
36     return dist;
37 }
38
39 int main() {
40     Node* root = Insert(0, 0);
41     root->left = Insert(1, 5);
42     root->right = Insert(2, 3);
43     root->left->left = Insert(3, 2);
44     root->left->right = Insert(4, 6);
45     root->right->left = Insert(5, 4);
46     root->left->right->left = Insert(6, 1);
47
48     for(int i = 0; i <= 6; i++) {
49         cout << find_dist(root, i) << " ";
50     }
51 }

```

7. Min pairing heap



8. AVL tree



9. AVL tree

```

1  #include<iostream>
2  using namespace std;
3
4  struct Node {
5      int key;          //months
6      int height;
7      Node* left;
8      Node* right;
9  };
10 int Height(Node* tmp) {
11     if (tmp == NULL) {
12         return -1;
13     }
14     return tmp->height;
15 }
16 int get_BF(Node* tmp) {
17     //balance factor
18     if (tmp == NULL) {
19         return 0;
20     }
21     return Height(tmp->left) - Height(tmp->right);
22 }
23 int max(int a, int b) {
24     return (a > b) ? a : b;
25 }
26 //      A          B
27 //    / \        / \
28 //   B  c3    =>  c1  A
29 //  / \        / \
30 // c1 c2      c2 c3
31 Node* right_rotate(Node* A) {
32     Node* B = A->left;
33     Node* C2 = B->right;
34     B->right = A;
35     A->left = C2;
36     A->height = max(Height(A->left), Height(A->right)) + 1;
37     B->height = max(Height(B->left), Height(B->right)) + 1;
38     return B;
39 }
40 //      A          B
41 //    / \        / \
42 //   c1  B    =>  A  c3
43 //  / \        / \
44 // c2 c3      c1 c2
45 Node* left_rotate(Node* A) {
46     Node* B = A->right;
47     Node* C2 = B->left;
48     B->left = A;
49     A->right = C2;
50     A->height = max(Height(A->left), Height(A->right)) + 1;
51     B->height = max(Height(B->left), Height(B->right)) + 1;
52     return B;
53 }
54 Node* Insert_node(int key) {
55     Node* tmp = new Node;
56     tmp->key = key;
57     tmp->height = 0;
58     tmp->right = NULL;
59     tmp->left = NULL;
60     return tmp;
61 }

```

```

62 Node* Insert(Node* node, int key) {
63     if (node == NULL) return(Insert_node(key));
64     if (key < node->key) {
65         node->left = Insert(node->left, key);
66     }
67     else if (key > node->key) {
68         node->right = Insert(node->right, key);
69     }
70     else {
71         cout << "Can't insert same keys.";
72         return node;
73     }
74     node->height = 1 + max(Height(node->left), Height(node->right));
75     int bf = get_BF(node);
76     if (bf > 1 && key < node->left->key) { //LL
77         return right_rotate(node);
78     }
79     if (bf > 1 && key > node->left->key) { //LR
80         node->left = left_rotate(node->left);
81         return right_rotate(node);
82     }
83     if (bf < -1 && key > node->right->key) { //RR
84         return left_rotate(node);
85     }
86     if (bf < -1 && key < node->right->key) { //RL
87         node->right = right_rotate(node->right);
88         return left_rotate(node);
89     }
90     return node;
91 }

```

```

93 void inorder(Node* root) {
94     if (root) {
95         inorder(root->left);
96         cout << root->key << " ";
97         inorder(root->right);
98     }
99 }
100 void preorder(Node* root) {
101     if (root) {
102         cout << root->key << " ";
103         preorder(root->left);
104         preorder(root->right);
105     }
106 }
107 int main() {
108     Node* root = NULL;
109     //months have been converted to number
110     root = Insert(root, 12);
111     root = Insert(root, 1);
112     root = Insert(root, 4);
113     root = Insert(root, 3);
114     root = Insert(root, 7);
115     root = Insert(root, 8);
116     root = Insert(root, 10);
117     root = Insert(root, 2);
118     root = Insert(root, 11);
119     root = Insert(root, 5);
120     root = Insert(root, 6);
121 }

```

```

107 int main() {
108     Node* root = NULL;
109     //months have been converted to number
110     root = Insert(root, 12);
111     root = Insert(root, 1);
112     root = Insert(root, 4);
113     root = Insert(root, 3);
114     root = Insert(root, 7);
115     root = Insert(root, 8);
116     root = Insert(root, 10);
117     root = Insert(root, 2);
118     root = Insert(root, 11);
119     root = Insert(root, 5);
120     root = Insert(root, 6);
121
122     cout << "Inorder traversal: ";
123     inorder(root);
124     cout << "\n\n";
125     cout << "Preorder traversal: ";
126     preorder(root);
127     cout << "\n\n";
128 }

```

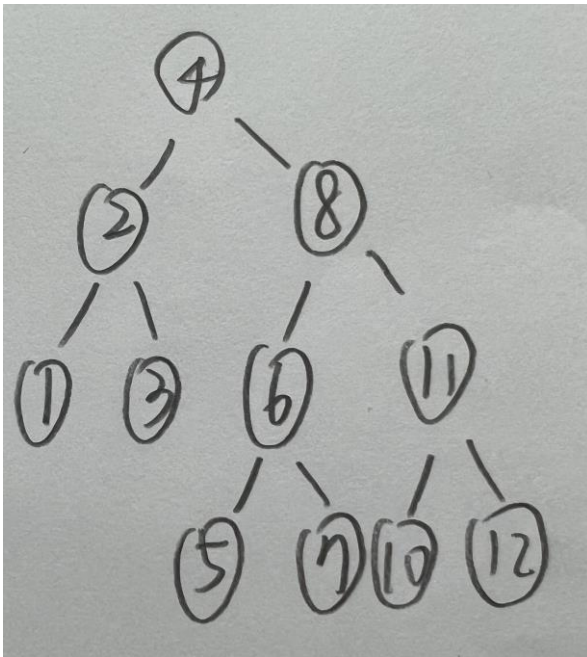
Microsoft Visual Studio 偵錯主控台

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

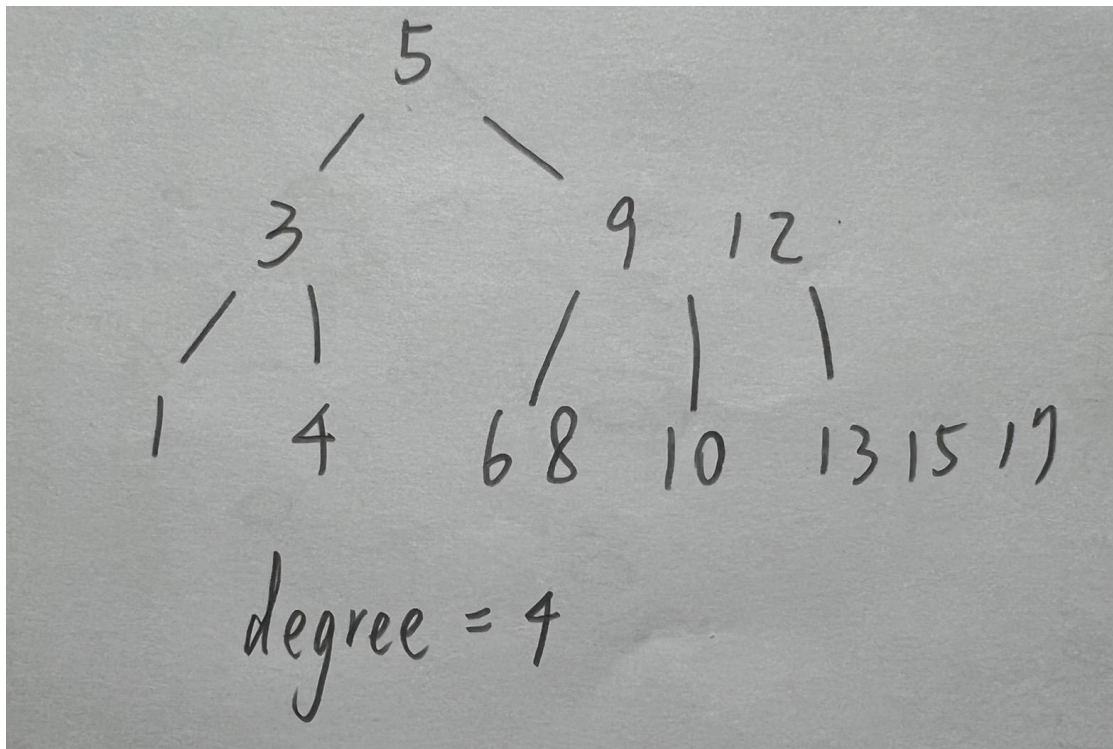
C:\Users\User\vs\hw3\Debug\hw3.exe (處理序 1960)
 若要在偵錯停止時自動關閉主控台，請啟用 [工具] ->
 按任意鍵關閉此視窗...

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



10. B-tree

Degree=4:



Degree=5:

