1.

part 1.

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

struct Edge {
    int source;
    int destination;
    int weight;
};

struct Graph {
    int V; // num of vertices
    int E; // num of edges
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E){ // graph constructor
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

void print(int dist[], int n, int source){
    cout << "Vertex \tDistance from Source(" << source << ")\n";
    for (int i = 0; i < n; ++i)
        cout << i << "\t\t" << dist[i] << "\n";
    cout << "\n";
}

void BellmanFord(struct Graph* graph, int source){
```

part 2.

```cpp
    int V = graph->V;
    int E = graph->E;
    int dist[V];
    // initialize each distance to infinity but not the source
    for (int i = 0; i < V; i++) dist[i] = INT_MAX;
    dist[source] = 0;
    // find the shortest path for each node
    for (int i=1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].source;
            int v = graph->edge[j].destination;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight <
                dist[v])
                dist[v] = dist[u] + weight;
        }
    }
    // check for negative cycle
    // the above loop already checked for the shortest path
    // if there still are a shorter distance then there's a
       negative cycle
    for (int i=0; i < E; i++) {
        int u = graph->edge[i].source;
        int v = graph->edge[i].destination;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Negative cycle detected.\n\n");
            return; //return if a negative cycle is detected
        }
    }
    print(dist, V, source);
```

```cpp
    graph->edge[6].destination = 1;
    graph->edge[6].weight = -3;

    BellmanFord(graph, 0);

    return 0;
}
```

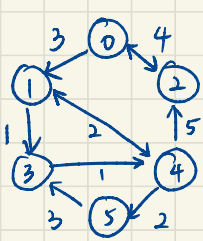
line 61 ~ 124 is used to set up the graphs in main.

```
Vertex  Distance from Source(0)
0       0
1       3
2       4
3       4
4       5
5       7

Vertex  Distance from Source(1)
0       11
1       0
2       7
3       1
4       2
5       4

Negative cycle detected.

Program ended with exit code: 0
```

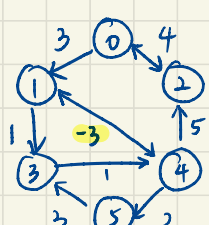
→ result from graph 1
source = ⓪

↳ result from graph 1
source = ①

result from graph 2



Graph 1



Graph 2

Graph 2 creates a negative cycle among ① ③ ④.

2. {12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18} Heap Sort

```
for (int i = n/2 ; i ≥ 1 ; i--)   (build max heap)
    Adjust (a, i, n);   → max_heapify
```

↗ must be 30 ↗ sorts list into a somewhat descending order

⇒ {30 → 20 → 28 → 12 → 18 → 16 → 4 → 10 → 2 → 6 → 8}

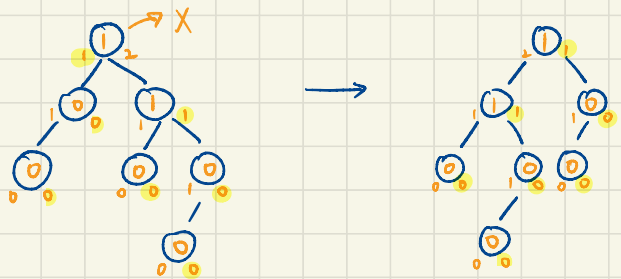
↳ the sequence may vary among different codes.

```
for (int i = n-1 ; i ≥ 1 ; i--) {
    swap(a[1], a[i+1]);   → swapping the first & last elements
    Adjust (a, 1, i);   heapify again and repeat the process
}
```

⇒ {8 → 20 → 28 → 12 → 18 → 16 → 4 → 10 → 2 → 6 → 30}
⇒ {6 → 20 → 16 → 12 → 18 → 8 → 4 → 10 → 2 → 28 → 30}
⇒ {2 → 18 → 16 → 12 → 6 → 8 → 4 → 10 → 20 → 28 → 30}
⇒ {2 → 12 → 16 → 10 → 6 → 8 → 4 → 18 → 20 → 28 → 30}
⇒ {4 → 12 → 8 → 10 → 6 → 2 → 16 → 18 → 20 → 28 → 30}
⇒ {2 → 10 → 8 → 4 → 6 → 12 → 16 → 18 → 20 → 28 → 30}
⇒ {2 → 6 → 8 → 4 → 10 → 12 → 16 → 18 → 20 → 28 → 30}
⇒ {4 → 6 → 2 → 8 → 10 → 12 → 16 → 18 → 20 → 28 → 30}
⇒ {2 → 4 → 6 → 8 → 10 → 12 → 16 → 18 → 20 → 28 → 30}
⇒ {2 → 4 → 6 → 8 → 10 → 12 → 16 → 18 → 20 → 28 → 30} ⇒ Final sort.

These are the list after swap()

3. This binary tree is not a leftist tree since the root node doesn't satisfy the requirement of "the left shortest route must be greater or equal to that of the right side". However, if we swap the branches of the root, then it would become a valid leftist tree.



4. Using mathematical induction.

1° The binomial tree B_0 is the base binomial tree for $k=0$ so B_0 has $2^0 = 1$ node. ⇒ Correct.

2° Assume a binomial tree B_{k-1}, $k-1 \geq 1$, has 2^{k-1} nodes. Then another binomial tree B_k can be constructed by merging two B_{k-1} trees which is $2^{k-1} + 2^{k-1} = 2^k$ nodes.

Thus by mathematical induction, a binomial tree B_k has 2^k nodes ($k \geq 0$)

5.

part 1.

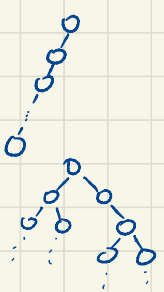
```
BST::BST(int x, int y){
    key = x;
    data = y;
    left = NULL;
    right = NULL;
}

BST* BST::Insert(BST* root, int x, int y){
    if (root == NULL) return new BST(x, y);

    if (x > root->key) root->right =
        Insert(root->right, x, y);
    else root->left = Insert(root->left, x,
        y);

    return root;
}

BST* BST::Delete(BST* root, int key){
    if (root == NULL) return root;
    if (key < root->key) root->left =
        Delete(root->left, key);
    else if (key > root->key) root->right =
        Delete(root->right, key);
    else{ // found the key
        if(root->right == NULL &&
            root->left == NULL){ // node
            has no child
            delete root;
            return NULL;
        }
```

part 2.

```
        else if(root->left == NULL){ // node has only right child
            BST *temp;
            temp = root->right;
            delete root;
            return temp;
        }
        else if(root->right == NULL){ // node has only left child
            BST *temp;
            temp = root->left;
            delete root;
            return temp;
        }
        else{ // node has two children
            BST *current;
            current = root->right;
            while(current->left != NULL) current = current->left;// gets the
                smallest value in the right subtree
            root->key = current->key; // copy the info to its new place
            root->data = current->data;
            root->right = Delete(root->right, root->key); // delete the inorder
                successor
        }
    }
    return root;
}

void BST ::Inorder(BST* root){
    if (root == NULL) return;
    Inorder(root->left);
    cout << " (" << setw(2) << root->key << ", " << setw(3) << root->data << ") ";
    Inorder(root->right);
}
```

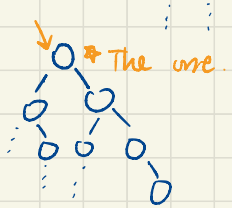
Time complexity T:

For the worst scenario where nodes ⇒ then $T = O(n)$, n = # of nodes.



For the average scenario where nodes ⇒ then $T = O(h)$, h = height of tree.

$h \approx \log_2 n$.



For the best scenario where nodes ⇒ then $T = O(1)$.

```
92      b.Insert(root, 5, 80);
93      b.Insert(root, 16, 90);
94      b.Insert(root, 11, 100);
95      b.Insert(root, 15, 110);
96      b.Inorder(root);
97      cout << "\n";
98      b.Delete(root, 3);
99      b.Inorder(root);
100     cout << "\n";
101     b.Delete(root, 8);
102     b.Inorder(root);
103     cout << "\n";
104     b.Delete(root, 15);
105     b.Inorder(root);
106     cout << "\n";
107     return 0;
108 }
```

The results from some tests.

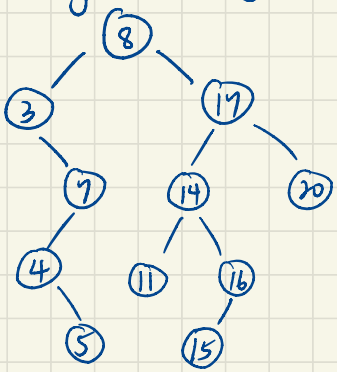
→ Delete (3)

→ Delete (8)

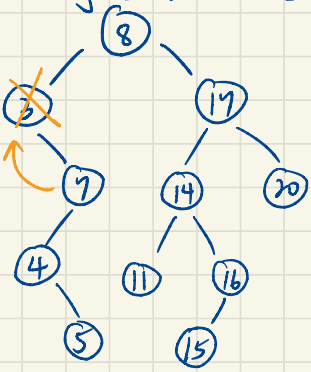
→ Delete (15)

```
( 3,  20) ( 4,  70) ( 5,  80) ( 7,  40) ( 8,  10) (11, 100) (14,  50) (15, 110) (16,  90) (17,  30) (20,  60)
( 4,  70) ( 5,  80) ( 7,  40) ( 8,  10) (11, 100) (14,  50) (15, 110) (16,  90) (17,  30) (20,  60)
( 4,  70) ( 5,  80) ( 7,  40) (11, 100) (14,  50) (15, 110) (16,  90) (17,  30) (20,  60)
( 4,  70) ( 5,  80) ( 7,  40) (11, 100) (14,  50) (16,  90) (17,  30) (20,  60)
Program ended with exit code: 0
```

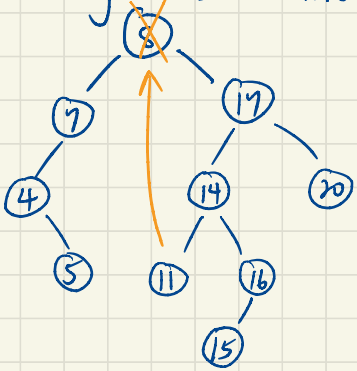
Original Tree



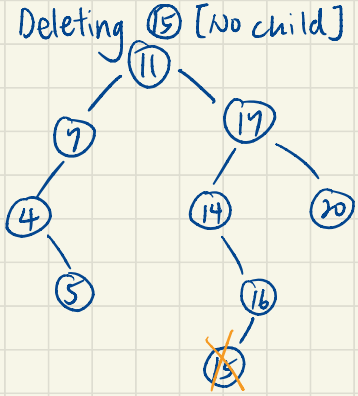
Deleting (3) [one child]



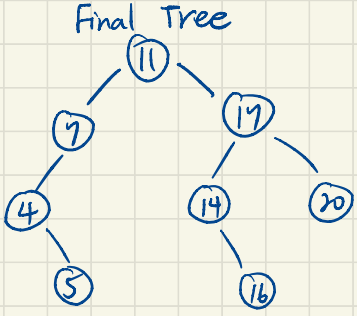
Deleting (8) [two children]



Deleting (15) [No child]



Final Tree



6.

part 1.

```
class Graph{
    int V; // # of vertices
    list<int> *adj;
    int **weight;

public:
    Graph(int);
    void addEdge(int, int, int);
    void BFS(int);
    void print(int*, int);
};

Graph::Graph(int num){
    this->V = num;
    adj = new list<int>[V];
    weight = new int*[num];
    for(int i=0; i<num; i++) weight[i] = new int[num];
}

void Graph::addEdge(int x, int y, int w){
    adj[x].push_back(y);
    adj[y].push_back(x);
    weight[x][y] = w;
    weight[y][x] = w;
}

void Graph::print(int *dist, int source){
    cout << "Vertex \tDistance from Source(" << source << ")\n";
    for (int i = 0; i < V; ++i)
        cout << i << "\t\t" << dist[i] << "\n";
    cout << "\n";
```

part 2.

```
void Graph::BFS(int source){
    bool *visited = new bool[V];
    for(int i=0; i<V; i++) visited[i] = false;

    list<int> queue;
    int dist[V];
    for(int i=0; i<V; i++) dist[i] = INT_MAX;
    dist[source] = 0;

    visited[source] = true;
    queue.push_back(source);

    list<int>::iterator it;

    while(!queue.empty()){
        int current = queue.front();
        queue.pop_front();

        for(it = adj[current].begin(); it !=
            adj[current].end(); ++it){
            if(!visited[*it]){
                visited[*it] = true;
                queue.push_back(*it);
                if(dist[current] != INT_MAX && dist[*it] >
                    dist[current] + weight[current][*it])
                    dist[*it] = dist[current] +
                        weight[current][*it];
            }
        }
    }
    print(dist, source);
```

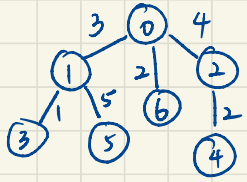
```
int main(){
    Graph g(7);
    g.addEdge(0, 1, 3); g.addEdge(0, 2, 4);
    g.addEdge(0, 6, 2); g.addEdge(1, 3, 1);
    g.addEdge(1, 5, 5); g.addEdge(2, 4, 2);

    g.BFS(0);
    g.BFS(3);

    return 0;
}
```

The result when source = (0)

```
Vertex  Distance from Source(0)
0       0
1       3
2       4
3       4
4       6
5       8
6       2
```

The result when source = (3)

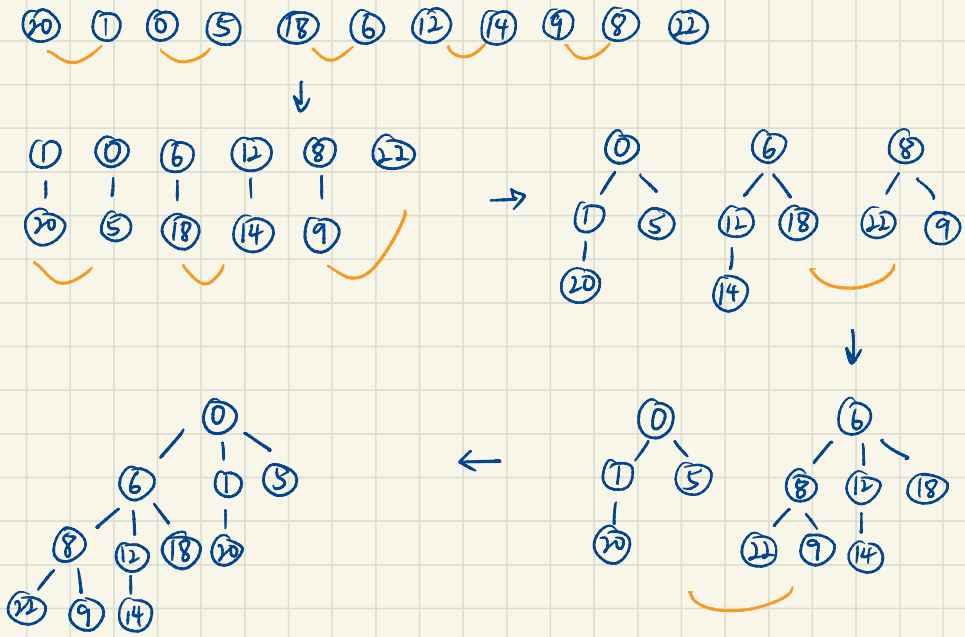
```
Vertex  Distance from Source(3)
0       4
1       1
2       8
3       0
4       10
5       6
6       6

Program ended with exit code: 0
```



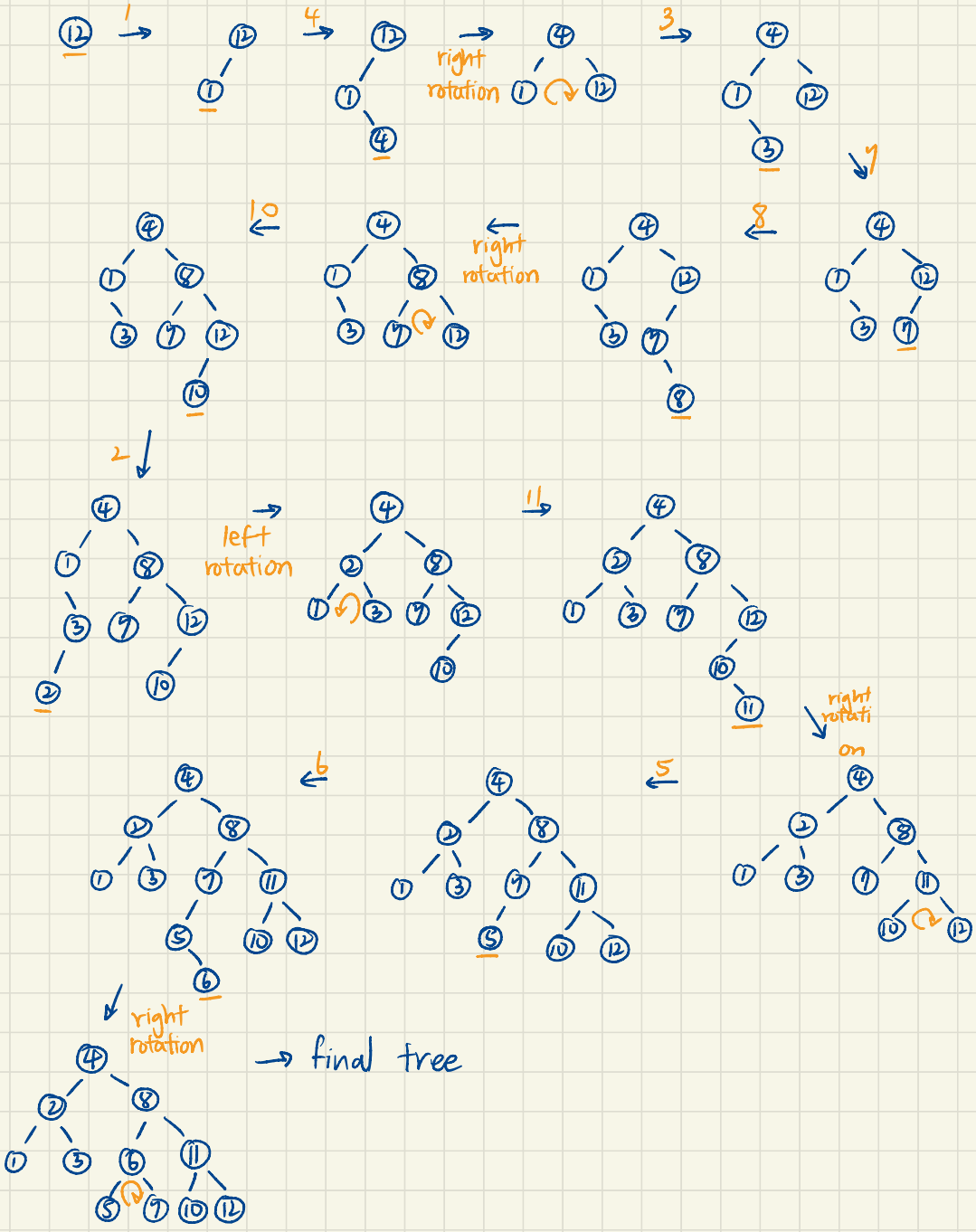
The tree used.

The time complexity of a BFS is O(V+E) when adjancency list is used where V = # of vertices and E = # of edges. In the question, V=n and E is unknown, but if E is a constant then O(V+E) = O(V) = O(n).

7. {20, 1, 0, 5, 18, 6, 12, 14, 9, 8, 22} pairing heap



8. { Dec(12), Jan(1), Apr(4), Mar(3), Jul(7), Aug(8), Oct(10), Feb(2), Nov(11), May(5), Jun(6) }



9.

part 1.

```
class Node{
public:
    int key;
    string data;
    int height;
    Node *left;
    Node *right;
};

int h(Node *N){ // returns the node's height
    if (N == NULL) return 0;
    else return N->height;
}

int max(int a, int b){
    return (a > b)? a : b;
}

int getBalance(Node *N){ // returns the difference in height
   if (N == NULL) return 0;
   else return h(N->left) - h(N->right);
}

int getHeight(Node *N){
    return max(h(N->left), h(N->right)) + 1;
}

Node* newNode(int key, string data){
    Node* node = new Node();
    node->key = key;
    node->data = data;
```

part 2.

```
    node->left = NULL;
    node->right = NULL;
    node->height = 1;

    return node;
}

Node *rightRotate(Node *y){
    Node *x = y->left;
    Node *T = x->right;

    x->right = y;
    y->left = T;

    x->height = getHeight(x);
    y->height = getHeight(y);

    return x;
}

Node *leftRotate(Node *y){
    Node *x = y->right;
    Node *T = x->left;

    x->left = y;
    y->right = T;

    x->height = getHeight(x);
    y->height = getHeight(y);

    return x;
```

part 3.

```
Node* insert(Node* node, int key, string data){ // returns the
    updated root

    if (node == NULL) return(newNode(key, data));

    if (key < node->key) node->left = insert(node->left, key,
        data);
    else node->right = insert(node->right, key, data);

    node->height = getHeight(node);

    // check the balance factor
    int balance = getBalance(node);

    // Left Left
    if (balance > 1 && key < node->left->key) return
        rightRotate(node);

    // Right Right
    else if (balance < -1 && key > node->right->key) return
        leftRotate(node);

    // Left Right
    else if (balance > 1 && key > node->left->key){
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    // Right Left
    if (balance < -1 && key < node->right->key){
        node->right = rightRotate(node->right);
        return leftRotate(node);
```

part 4.

```
    return node; // if nothing's changed
}

void inOrder(Node *root){
    if(root != NULL){
        inOrder(root->left);
        cout << root->data << "\t";
        inOrder(root->right);
    }
}

int main(void){
    Node *root = NULL;

    root = insert(root, 12, "Dec");
    root = insert(root, 1, "Jan");
    root = insert(root, 4, "Apr");
    root = insert(root, 3, "Mar");
    root = insert(root, 7, "Jul");
    root = insert(root, 8, "Aug");
    root = insert(root, 10, "Oct");
    root = insert(root, 2, "Feb");
    root = insert(root, 11, "Nov");
    root = insert(root, 5, "May");
    root = insert(root, 6, "Jun");

    inOrder(root);
    cout << "\n";

    return 0;
}
```

```
Jan Feb Mar Apr May Jun Jul Aug Oct Nov Dec
Program ended with exit code: 0
```

Results in ascending order.

Inorder traversal would show the elements in ascending order since the is a type of binary search tree where key(left) < key(parent) < key(right).

10. {8, 9, 12, 10, 15, 4, 1, 13, 17, 3, 5, 6} B-tree, Max degree = 3