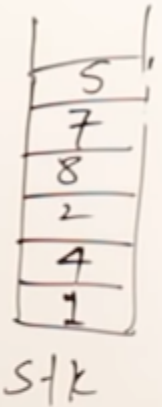
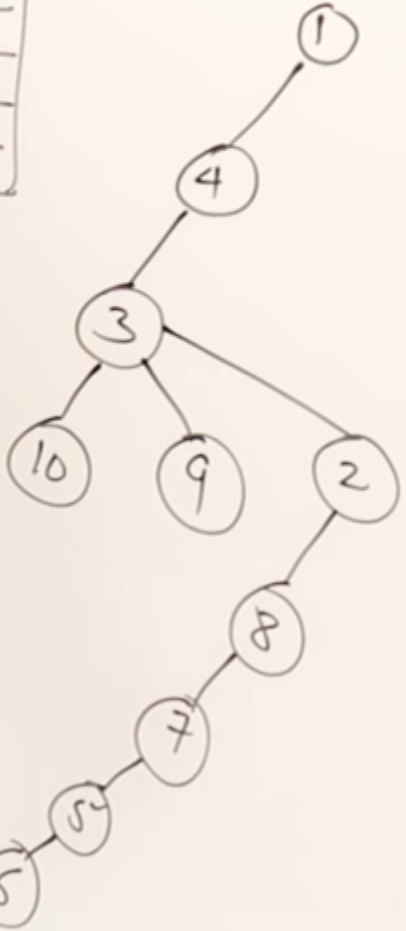
```
template <class T> class Graph {
 public:
   map<T, Vertex<T>> vertices;

   Vertex<T> *addIfNotExists(T item) {
      auto i = vertices.find(item);
      if (i != vertices.end())
         return &(i->second);
      return &(vertices[item] = Vertex<T>(item));
   }
```

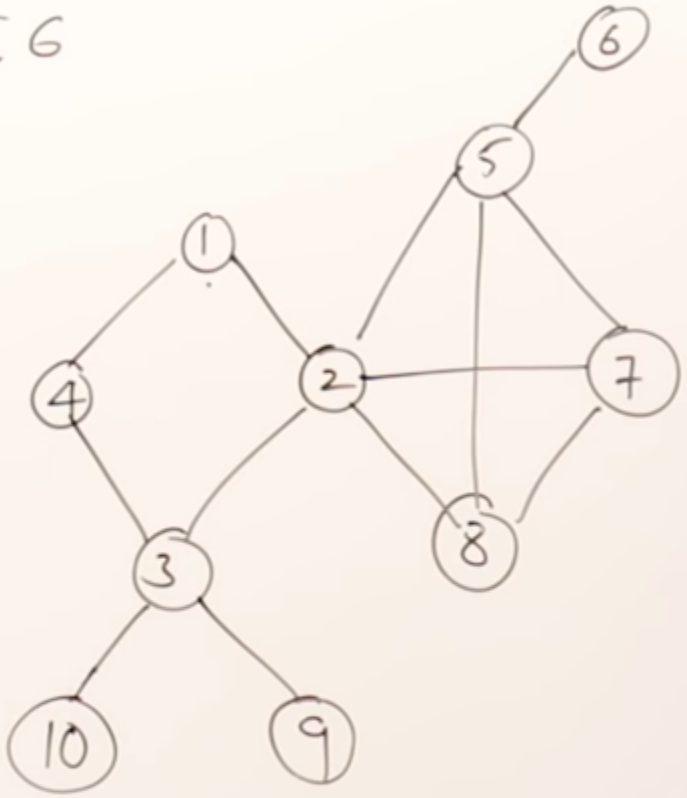
```cpp
template <class T> class Vertex {
 public:
    T item;
    list<Vertex<T> *> neighbors;
    bool visited = false;
    Vertex() {}
    Vertex(T item) : item(item) {}
};
```

DFS: 1, 4, 3, 10, 9, 2, 8, 7, 5, 6




Stk



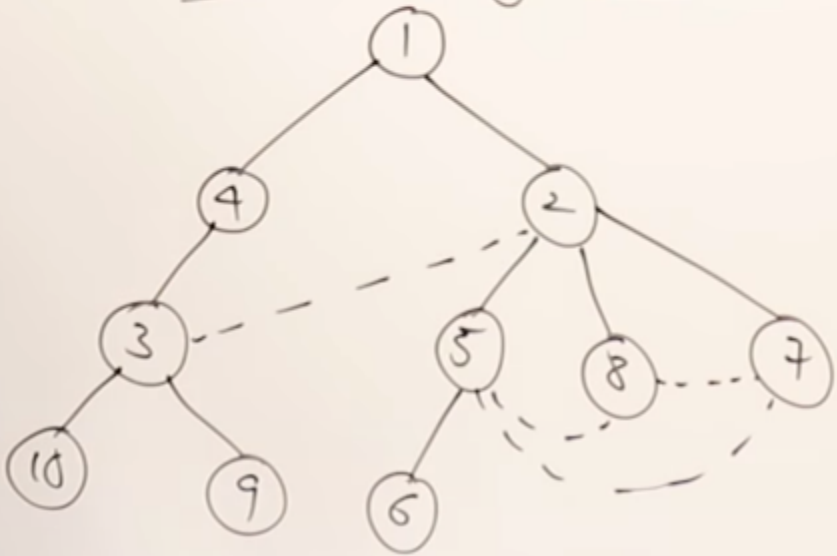
```
void DFS() {
    cout << "DFS ";
    stack<Vertex<T> *> s;
    Vertex<T> *p;
    s.push(&((vertices.begin())->second));
    while (!s.empty()) {
        p = s.top();
        s.pop();
        if (!p->visited) {
            p->visited = true;
            cout << p->item << " ";
            for (auto &v : p->neighbors) {
                if (!(v->visited)) {
                    s.push(v);
                }
            }
        }
    }
    cout << endl;
}
```

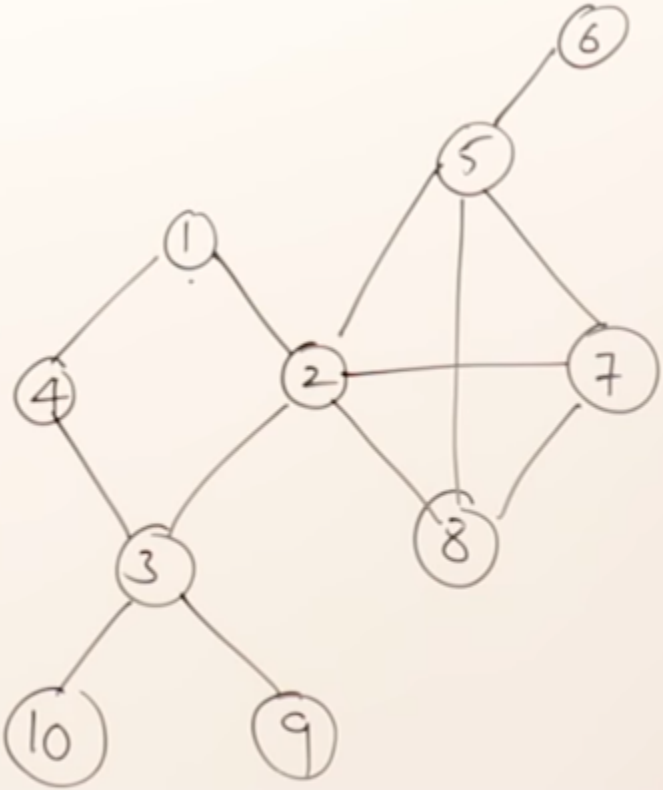
```
void DFSRInternal(Vertex<T> *i) {
    cout << i->item << " ";
    i->visited = true;
    for (auto v : i->neighbors) {
        if (!(v->visited)) {
            DFSRInternal(v);
        }
    }
}

void DFSR() {
    cout << "DFSR ";
    DFSRInternal(&((vertices.begin())->second));
    cout << endl;
}
```

BFS : 1, 4, 2, 3, 5, 8, 7, 10, 9, 6

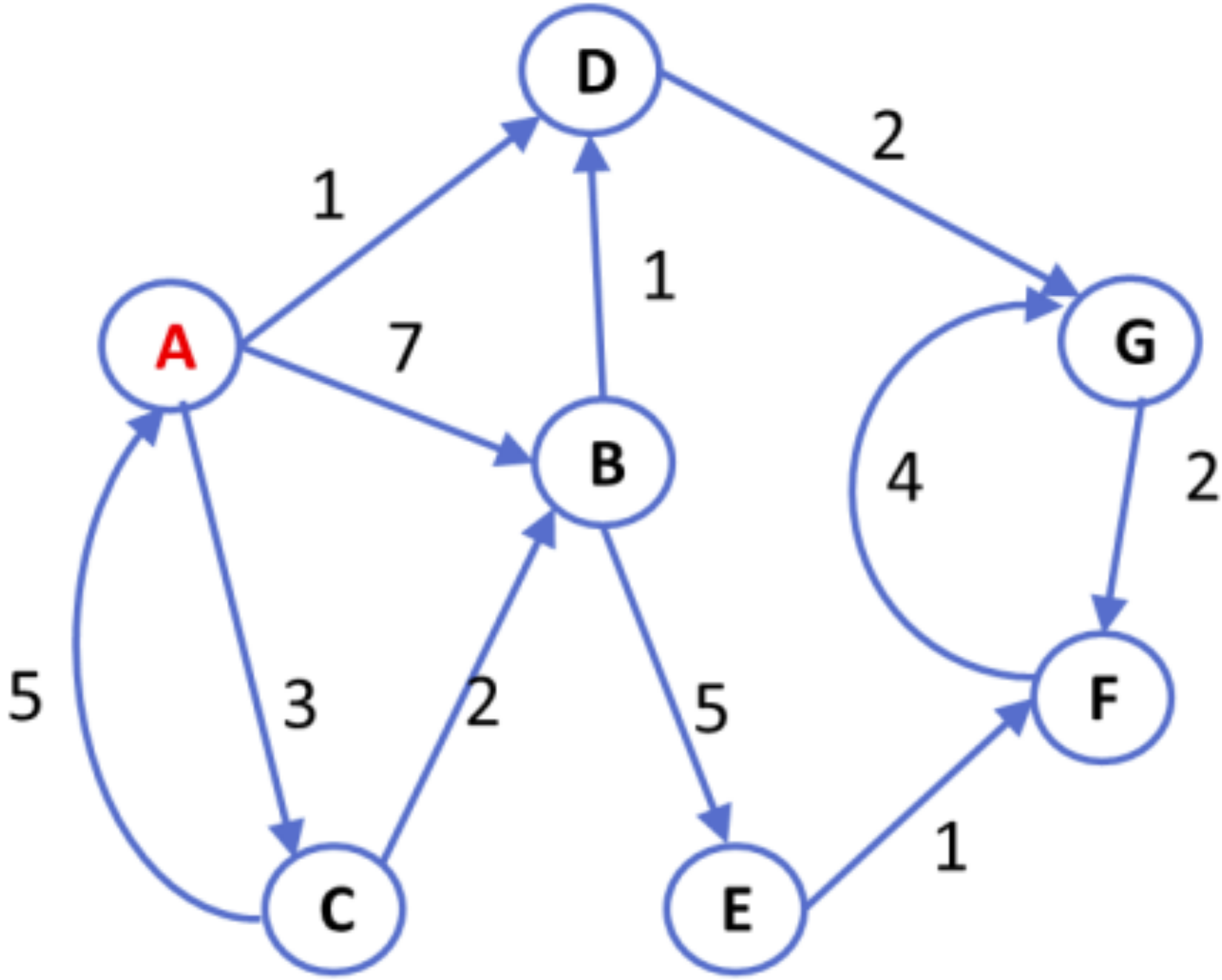
Q | ~~1~~ | ~~4~~ | ~~2~~ | ~~3~~ | ~~5~~ | ~~8~~ | 7 | 10 | 9 | 6 |

BFS Spanning Tree



```
void BFS() {
    cout << "BFS ";
    queue<Vertex<T> *> q;
    Vertex<T> *p;
    p = &((vertices.begin())->second);
    p->visited = true;
    cout << p->item << " ";
    q.push(p);
    while (!q.empty()) {
        p = q.front();
        q.pop();
        for (auto &v : p->neighbors) {
            if (!(v->visited)) {
                v->visited = true;
                cout << v->item << " ";
                q.push(v);
            }
        }
    }
    cout << endl;
}
```


D
2
1
1
A
7
G
B
4
2
5
3
2
5
F
1
C
E

**1. Početno stanje**

**$Q$ = {0, ∞, ∞, ∞, ∞, ∞, ∞}**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Udaljenost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Preth | - | - | - | - | - | - | - |

**2. Uzimamo vrh A iz $Q$ i označavamo da je obiđen.**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Udaljenost | 0 | 7 | 3 | 1 | ∞ | ∞ | ∞ |
| Preth | - | A | A | A | - | - | - |

**$Q$ = {1, 7, 3, ∞, ∞, ∞}**

**3. Uzimamo vrh D iz $Q$ i označavamo da je obiđen. Podešavamo udaljenosti.**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Udaljenost | 0 | 7 | 3 | 1 | ∞ | ∞ | 3 |
| Preth | - | A | A | A | - | - | D |

**$Q$ = {3(C), 7, 3(G), ∞, ∞}**

**4. Uzimamo vrh C iz $Q$ i označavamo da je obiđen. Podešavamo udaljenosti.**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Udaljenost | 0 | 5 | 3 | 1 | ∞ | ∞ | 3 |
| Preth | - | C | A | A | - | - | D |

**$Q$ = {3 (G), 5, ∞, ∞}**

**5. Uzimamo vrh G iz $Q$ i označavamo da je obiđen. Podešavamo udaljenosti.**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Udaljenost | 0 | 5 | 3 | 1 | ∞ | 5 | 3 |
| Preth | - | C | A | A | - | G | D |

**$Q$ = {5 (B), 5 (F), ∞}**

**6. Uzimamo vrh B iz $Q$ i označavamo da je obiđen.**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Udaljenost | 0 | 5 | 3 | 1 | 10 | 5 | 3 |
| Preth | - | C | A | A | B | G | D |

**$Q$ = {5 (F), 10}**

**7. Uzimamo vrh F iz $Q$ i označavamo da je obiđen. Podešavamo udaljenosti.**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Udaljenost | 0 | 5 | 3 | 1 | 10 | 5 | 3 |
| Preth | - | C | A | A | B | G | D |

**$Q$ = { 10 }**

**8. Uzimamo vrh E iz $Q$ i označavamo da je obiđen. Time smo obišli sve vrhove.**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Udaljenost | 0 | 5 | 3 | 1 | 10 | 5 | 3 |
| Preth | - | C | A | A | B | G | D |

**$Q$ = { }**

```
string source = "A";
// initialize Q
auto comparator = [](Vertex<string> *a, Vertex<string> *b) {
    return a->distanceFromSource > b->distanceFromSource;
};
// all template parameters must be types, so decltype returns the type of
// comparator
priority_queue<Vertex<string> *, vector<Vertex<string> *>, decltype(comparator)> Q(comparator);

for (auto &vi : g.vertices) {
    v = &(vi.second);
    if (v->item == source) {
        v->distanceFromSource = 0;
    } else {
        v->distanceFromSource = INFINITY;
    }
    v->previous = nullptr;
    v->visited = false;
    Q.push(v);
}
```

```
// iterate through Q
while (!Q.empty()) {
    u = Q.top();
    // if only interested in distance to a destination
    // if (u->item == destination) break;
    u->visited = true;
    for (auto &n : u->neighbors) {
        if (!n.vertex->visited) {
            newDistance = u->distanceFromSource + n.distance;
            if (newDistance < n.vertex->distanceFromSource) {
                n.vertex->distanceFromSource = newDistance;
                n.vertex->previous = u;
            }
        }
    }
    Q.pop();
}
g.printDistances();
```

```
// actual paths for all nodes
stack<Vertex<string> *> s;
Vertex<string> *p;

for (auto &vi : g.vertices) {
    v = &(vi.second);
    s.push(v);
    for (p = v->previous; p != nullptr; p = p->previous) {
        s.push(p);
    }
    while (!s.empty()) {
        v = s.top();
        cout << v->item << " ";
        s.pop();
    }
    cout << v->distanceFromSource << endl;
}
```