Implementing Graphs in C++

Background

Graphs not available in STL (too many different possibilities)

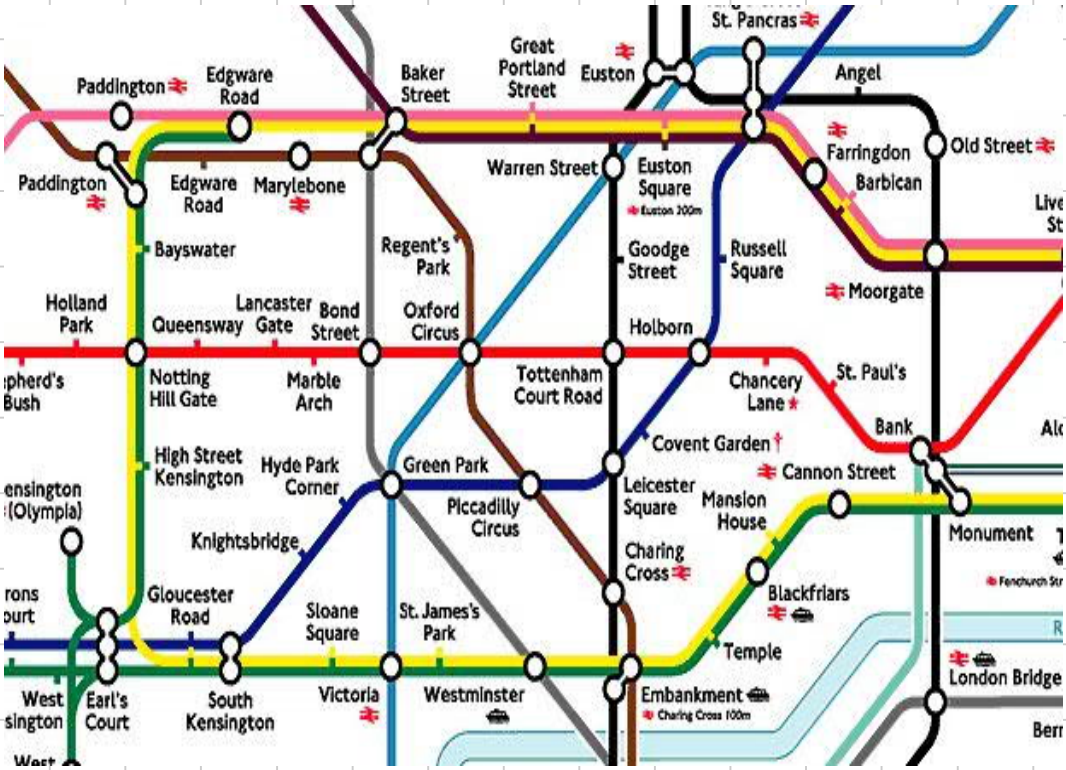
Assumptions

- nodes exist along with accompanying data

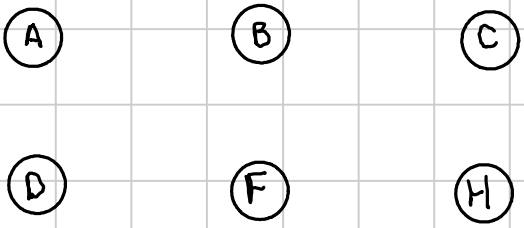
- all nodes already stored in own container

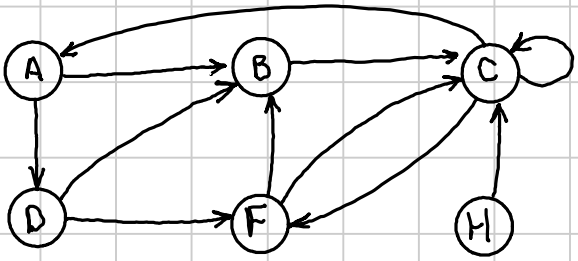
- directed graph

Part of London subway system



Example





Nodes and edges

Nodes

x	A	B	C	D	F	H
x's adjacency set	D, B	C	C, A, F	B, F	B, C	C

□

Implementation 1

```
std::unordered_map<int, Node>
allNodes;

struct Node
{
    // All the data stored in the node
    int id;
    std::string name;
    // other node data ...

    std::vector<Node*> to_neighbours;
};
```

Node struct has

- unique node identifier

- a name

- node's adjacency set (pointers)

Suitable when

- only need to move forward along edges
- edges added or deleted infrequently

Implementation 2

```
struct Node
{
    // All the data stored in the node
    int id;
    std::string name;
    // other node data ...

    std::unordered_set<Node*> to_neighbours;
};
```

Node struct has

- unique node identifier

- a name

- node's adjacency set (pointers)

Suitable when

- only need to move forward along edges
- edges added or deleted frequently

Implementation 3

```
struct Node
{
    // All the data stored in the node
    int id;
    std::string name;
    // other node data ...

    std::vector<Node*> to_neighbours;
    std::vector<Node*> from_neighbours;
};
```

Node struct has

- unique node identifier

- a name

- node's adjacency set (pointers)

- to what other nodes this node is adjacent (pointers)

Suitable when

- need to move both forward and backward along edges
- edges added or deleted infrequently

Implementation 4

```
enum Color { WHITE, GRAY, BLACK };

struct Node
{
    // All the data stored in the node
    int id;
    std::string name;

    int d; // Distance
    Node* parent; // π
    Color color; // node's color

    std::vector<Node*> to_neighbours;
};
```

Node struct has

- unique node identifier
- a name
- node's adjacency set (pointers)
- a distance
- a parent node
- a color

Suitable when

- only need to move forward along edges
- edges added or deleted infrequently
- some graph search/traversal algorithm is performed (BFS, DFS)
- memory use not strict (d, parent, color always included)

Implementation 5

```
enum Color { WHITE, GRAY, BLACK };

struct SearchInfo
{
    int d; // Distance
    Node* path_back; // π
    Color color; // Color
};

struct Node
{
    // All the data stored in the node
    int id;
    std::string name;

    SearchInfo* search_info; // Must be
created!

    std::vector<Node*> to_neighbours;
};
```

Node struct has

- unique node identifier
- a name
- node's adjacency set (pointers)
- pointer to extra data

Suitable when

- only need to move forward along edges
- edges added or deleted infrequently
- some graph search/traversal algorithm is performed (BFS, DFS)
- memory use strict (create extra data when needed, delete after)

Implementation 6

```
enum Color { WHITE, GRAY, BLACK };

struct SearchInfo
{
    int d; // Distance
    int parent; // π
    Color color; // Color
};

vector<Node> allNodes;
vector<SearchInfo> allSearchInfos;

struct Node
{
    // All the data stored in the node
    int id;
    std::string name;

    std::vector<int> to_neighbours;
};
```

Node struct has

- unique node identifier

- a name

- node's adjacency set (indices)

Suitable when

- only need to move forward along edges
- edges added or deleted infrequently
- some graph search/traversal algorithm is performed (BFS, DFS)
- memory use strict (create extra vector when needed, delete after)