Queue

```
#include <iostream>
#define LIMIT 5

struct Queue {
        int value;
        Queue* next = nullptr;

        Queue(int a, Queue* b) : value(a), next(b) {} // 👻 CPP Moment
};

void enqueue(Queue*& antrian, int value) {
        Queue* pelanggan = new Queue(value, nullptr);
        // atau
        // pelanggan->value = value;

        // Baru pertama kali
        if (antrian == nullptr) {
                antrian = pelanggan;
                return;
        }
        Queue* read = antrian;
        while (read->next != nullptr) {
                read = read->next;
        }
        read->next = pelanggan;
}

// Sekalian ngeprint why not
void dequeue(Queue*& antrian) {
        // baru bikin langsung dequeue maka return aja memory safe juga...
        if (antrian == nullptr) {
                return;
        }

        Queue* temp = antrian;
        std::cout << temp->value << std::endl;
        antrian = antrian->next;
        delete temp;
}

int queue() {
        Queue* antrian = nullptr;

        enqueue(antrian, 0);
        enqueue(antrian, 1);
        enqueue(antrian, 2);
        enqueue(antrian, 3);
        enqueue(antrian, 4);

        dequeue(antrian);
        dequeue(antrian);
        dequeue(antrian);
        dequeue(antrian);
        dequeue(antrian);
        return 0;
}
```

Binary Tree

```
#include <iostream>
#include <queue>

typedef struct BinaryTree {
        int value;
        BinaryTree* left = nullptr;
        BinaryTree* right = nullptr;

        BinaryTree(int a, BinaryTree* b = nullptr, BinaryTree* c = nullptr) : value(a), left(b), right(c) {}
        // atau
        // BinaryTree(int a) {
        //      value = a;
        //      left = right = nullptr;
        // }
}BT;


void addBTree(BT*& tree, int value) {
        if (tree == nullptr) {
                tree = new BT(value);
                return;
        }

        std::queue<BT*> queue;
        queue.push(tree);

        // 9,7,8,5,3,4,1 (no dupe solution)
        // Level Order Insertion
        while (!queue.empty()) {
                BT* current = queue.front(); // 7
                queue.pop();

                if (current->left == nullptr) {
                        current->left = new BT(value); // 9 L7; 7 L5
                        return;
                }
                else {
                        queue.push(current->left); // 5
                }

                if (current->right == nullptr) { // 9 L7 R8; 7 L5 R3;
                        current->right = new BT(value);
                        return;
                }
                else {
                        queue.push(current->right); // 8
                }
        }
}

void inorderPrint(BT* tree) {
        if (tree == nullptr) return;
        inorderPrint(tree->left);
        std::cout << tree->value << " ";
        inorderPrint(tree->right);
}

// 9,7,8,5,3,4,1 (no dupe solution)
// Gini lah gambarannya... :
//        9
//    7   |   8
//  5 | 3 | 4 | 1

int BTree() {
        BT* tree = nullptr;
        addBTree(tree, 9);
        addBTree(tree, 7);
        addBTree(tree, 8);
        addBTree(tree, 5);
        addBTree(tree, 3);
        addBTree(tree, 4);
        addBTree(tree, 1);

        std::cout << "Inorder Traversal: ";
        inorderPrint(tree);
        std::cout << std::endl;

        return 0;
}
```

Graph

```cpp
#include <iostream>

const int MAX_NODES = 5; // misal 5 simpul (0,1,2,3,4)

struct DoubleLinkedList {
    int data;
    DoubleLinkedList* next;
    DoubleLinkedList* prev;
    DoubleLinkedList(int d) : data(d), next(nullptr), prev(nullptr) {}
};

struct AdjacencyList {
    DoubleLinkedList* head;
    AdjacencyList() : head(nullptr) {}
};

void insertEdge(AdjacencyList* graph, int from, int to) {
    DoubleLinkedList* newNode = new DoubleLinkedList(to);

    if (graph[from].head == nullptr) {
        graph[from].head = newNode;
    }
    else {
        DoubleLinkedList* temp = graph[from].head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

void printGraph(AdjacencyList* graph, int nodes) {
    for (int i = 0; i < nodes; ++i) {
        std::cout << "Node " << i << " -> ";
        DoubleLinkedList* temp = graph[i].head;
        while (temp != nullptr) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << std::endl;
    }
}

void clearGraph(AdjacencyList* graph, int nodes) {
    for (int i = 0; i < nodes; ++i) {
        DoubleLinkedList* current = graph[i].head;
        while (current != nullptr) {
            DoubleLinkedList* toDelete = current;
            current = current->next;
            delete toDelete;
        }
    }
}

int main() {
    AdjacencyList graph[MAX_NODES];

    // Membuat graf:
    // 0 -> 1, 4
    // 1 -> 2, 3
    // 2 -> 3
    // 3 -> 4
    // 4 -> (kosong)

    insertEdge(graph, 0, 1);
    insertEdge(graph, 0, 4);
    insertEdge(graph, 1, 2);
    insertEdge(graph, 1, 3);
    insertEdge(graph, 2, 3);
    insertEdge(graph, 3, 4);

    std::cout << "Adjacency List of Graph:\n";
    printGraph(graph, MAX_NODES);

    clearGraph(graph, MAX_NODES);
    return 0;
}
```