

**มหาวิทยาลัยเทคโนโลยีสุรนารี**

**โปรแกรมสำหรับจัดเก็บข้อมูลกราฟไม่มีทิศทาง (Undirected Graph)**

**และการหาวงจรออยเลอร์ (Euler Circuits)**

**กลุ่ม G21**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **นายสุรวิช** | **ธิษาไชย** | **B6505677** | **เลขที่ 11** |  |
| **นายณัฐวุฒิ** | **ถินราช** | **B6506919** | **เลขที่ 16** | **(0650093625)** |
| **นายพูลทรัพย์** | **นานาวัน** | **B6525163** | **เลขที่ 60** | **(0985944576)** |
| **นายสิทธิโชค** | **เชยงูเหลือม** | **B6526399** | **เลขที่ 67** |  |

**อาจารย์ที่ปรึกษา**

**อาจารย์ ดร.สุภาพร บุญฤทธิ์**

**รายงานฉบับนี้เป็นส่วนหนึ่งของรายวิชา**

**โครงสร้างข้อมูลและขั้นตอนวิธี รหัสวิชา ENG23 2031**

**หลักสูตรวิศวกรรมศาสตร์บัณฑิต สาขาวิศวกรรมคอมพิวเตอร์**

**สำนักวิชาวิศกรรมศาสตร์มหาวิทยาลัยเทคโนโลยีสุรนารี**

**ประจำภาคการศึกษาที่ 2 ปีการศึกษา 2566**

**คำนำ**

รายงานฉบับนี้จัดทำขึ้นเพื่อเป็นส่วนหนึ่งของรายวิชา โครงสร้างข้อมูลและขั้นตอนวิธี รหัสวิชา ENG23 2031 ปีการศึกษา 2/2566 เพื่อการศึกษากราฟที่ไม่มีทิศทาง (Undirected Graph) เป็นหนึ่งในแนวทางที่น่าสนใจในวงการคณิตศาสตร์ของกราฟ โดยที่ไม่มีการกำหนดทิศทางในการเชื่อมต่อระหว่างจุด (vertices) ซึ่งมีแนวคิดที่เชื่อมโยงกับหลายด้านของวิชาเอกชนิดนี้ เช่น การวิเคราะห์โครงสร้างข้อมูล การสื่อสารในเครือข่าย และอัลกอริทึมที่เกี่ยวข้อง ในรายงานนี้จะพิจารณาคุณสมบัติพื้นฐานของ Undirected Graph แนวคิดทางคณิตศาสตร์ที่เกี่ยวข้อง และการใช้งานทั้งในทฤษฎีและปฏิบัติทางคอมพิวเตอร์ได้อย่างละเอียด

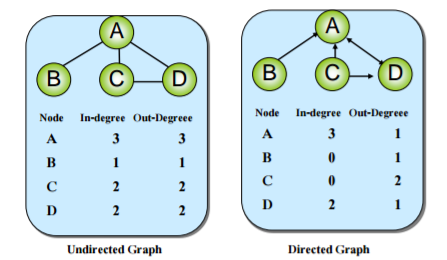
ขอขอบพระคุณ อ.ดร.สุภาพร บุญฤทธิ์ ที่ได้มอบความรู้ และชี้แนวทางการศึกษา ประกอบกับการค้นคว้าหาข้อมูลของสมาชิกในกลุ่มรวบรวมข้อมูลจากหลายๆแหล่งต่างๆ

ผู้จัดทำหวังเป็นอย่างยิ่งว่ารายงานฉบับนี้จะสามารถทำให้ได้รับข้อมูลที่เป็นประโยชน์แก่ผู้อ่าน

**สารบัญ**

|  |  |
| --- | --- |
|  | **หน้า** |
| คำนำ  สารบัญ  **1.ทฤษฎี**   * 1. Degree (องศา)   2. Path (เส้นทาง)   3. Indirected Graph (กราฟไม่ระบุทิศทาง)   4. Circuit (วงลูป)   5. Adjacency Matrix (เมทริกซ์ความใกล้ชิด)   6. Euler Circuits (ทฤษฎีวงลูปออยเลอร์)   **2.ฟังก์ชันหลักภายในโปรแกรม**  - ฟังก์ชัน addNode  - ฟังก์ชัน removeNode  - ฟังก์ชัน findEuler  - ฟังก์ชัน addEdge  - ฟังก์ชัน printGraph  - ฟังก์ชัน printCircuit  - ฟังก์ชัน findCircuit  - ฟังก์ชัน isEuler  - ฟังก์ชัน setAdMatrix  - Function Main()  - ตัวอย่างการแสดงผลกรณีกราฟไม่มีวงจร Euler  - ตัวอย่างการแสดงผลกรณีกราฟมีวงจร Euler  **3.ฟังก์ชันหลักในโปรแกรมแปลง Infix เป็น Postfix โดยใช้ Stack**  - ฟังก์ชัน push  - ฟังก์ชัน pop  - ฟังก์ชัน stacktop  - ฟังก์ชัน checkpr  - ฟังก์ชัน checkoper  - ฟังก์ชัน printstack  - ฟังก์ชัน infixtopostfix  - Function main()  - ตัวอย่างการแสดงผล  **4.ฟังก์ชันหลักในโปรแกรมประมวลผลนิพจน์แบบ Postfix ให้เป็นผลลัพธ์**  - ฟังก์ชัน pushStr  - ฟังก์ชัน popStr  - ฟังก์ชัน calculateStr  - ฟังก์ชัน printstackStr  - Function Main()  - ตัวอย่างการแสดงผล | ก  ข  1  1  1  2  2  3  3  4  4  5  7  9  9  10  11  13  14  15  16  17  18  18  19  20  21  22  23  25  27  27  29  30  31  32  33  34  35 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**ทฤษฎีที่เกี่ยวข้อง**

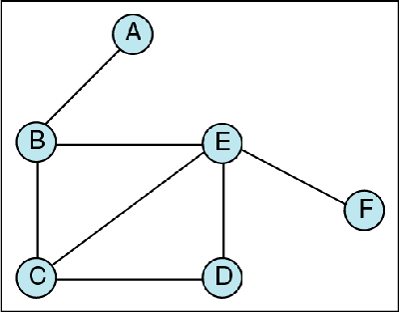
* 1. **Degree (องศา)**

องศาของจุดในกราฟแบบ Undirected คือจำนวนของเส้นเชื่อมที่เชื่อมต่อกับจุดนั้นๆ ซึ่งสามารถแบ่งเป็น องศาเข้า (in-degree) และ องศาออก (out-degree)

**1.2 Path (เส้นทาง)**

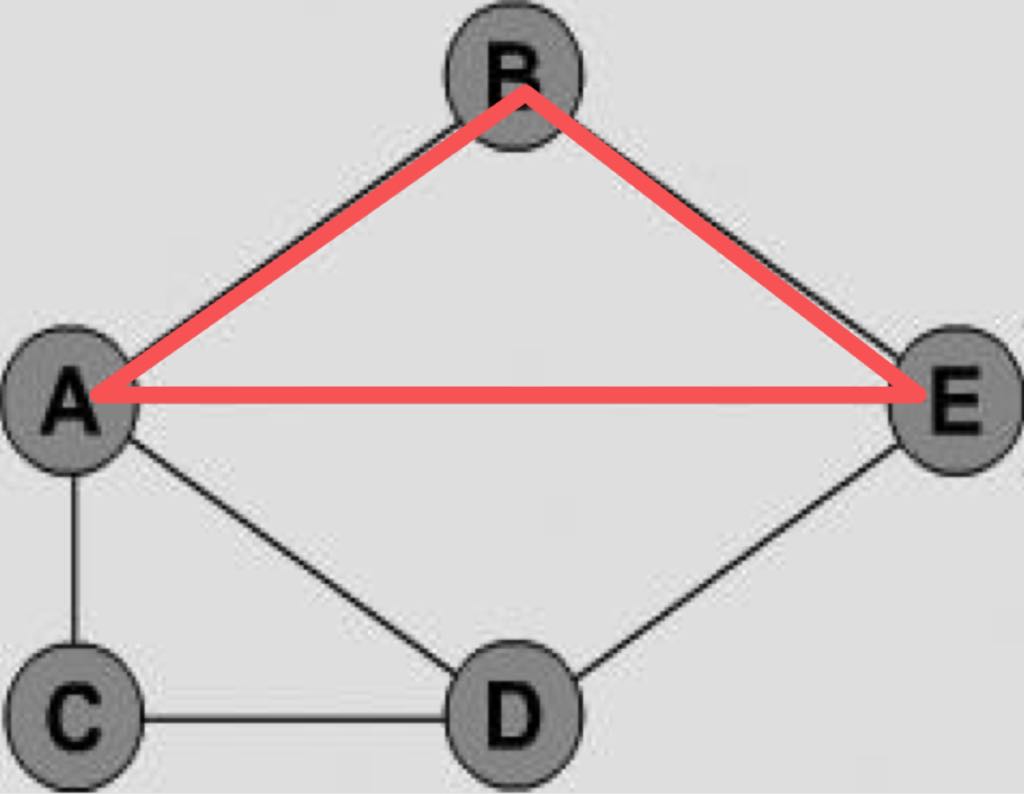
เส้นทางใน Undirected Graph คือลำดับของจุดที่เชื่อมต่อกัน โดยที่ไม่มีเส้นทางที่ทิศทางเดียว

**1.3 Indirected Graph (กราฟไม่ระบุทิศทาง)**

****

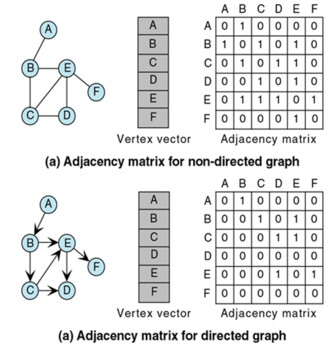
กราฟที่ไม่ระบุทิศทางหรือ Indirected Graph เป็นโครงสร้างข้อมูลทางคณิตศาสตร์ที่ประกอบด้วยจุด (หรือโหนด) และเส้น (หรือเส้นเชื่อม) โดยไม่มีการกำหนดทิศทางในการเชื่อมต่อระหว่างจุดทั้งสอง

**1.4 Circuit (วงลูป)**

****

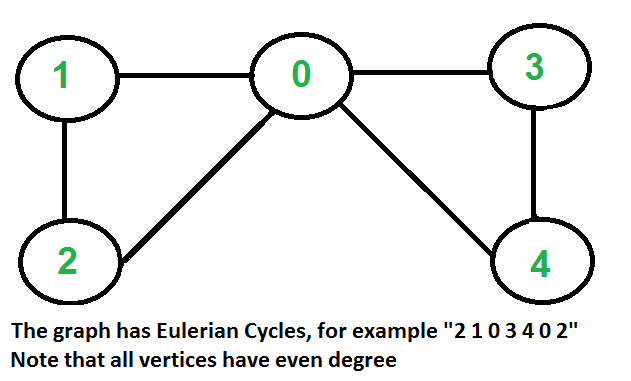
เป็นเส้นทางที่เริ่มต้นและสิ้นสุดที่จุดเดิมในกราฟ โดยไม่ผ่านเส้นเชื่อมใดที่ซ้ำซ้อน นั่นคือวงลูปที่ไม่มีเส้นเชื่อมที่ซ้ำ

**1.5 Adjacency Matrix (เมทริกซ์ความใกล้ชิด)**

****

ทฤษฎี Adjacency Matrix เป็นหนึ่งในวิธีการแทนกราฟในรูปแบบของเมทริกซ์ที่ใช้เก็บข้อมูลเกี่ยวกับความเชื่อมโยงระหว่างโหนดในกราฟ โดยที่แถวและคอลัมน์ของเมทริกซ์แสดงถึงโหนดทั้งหมดในกราฟ และค่าในเซลล์ (cell) ของเมทริกซ์บ่งบอกถึงการเชื่อมต่อระหว่างโหนดที่แถวและคอลัมน์ที่กำหนด

**1.6 Euler Circuits (ทฤษฎีวงลูปออยเลอร์)**

****

ทฤษฎี Euler Circuits เกี่ยวกับการแบ่งกราฟได้เป็นวงรอบที่ผ่านทุกระหว่างเส้นเชื่อม (edges) ของกราฟ โดยในกราฟที่มี Euler Circuit จะสามารถทำการเดินทางไปทุกระหว่างเส้นเชื่อมโดยไม่ต้องผ่านเส้นเชื่อมใดซ้ำซ้อน

**ฟังก์ชันหลักในโปรแกรมสำหรับจัดเก็บข้อมูล**

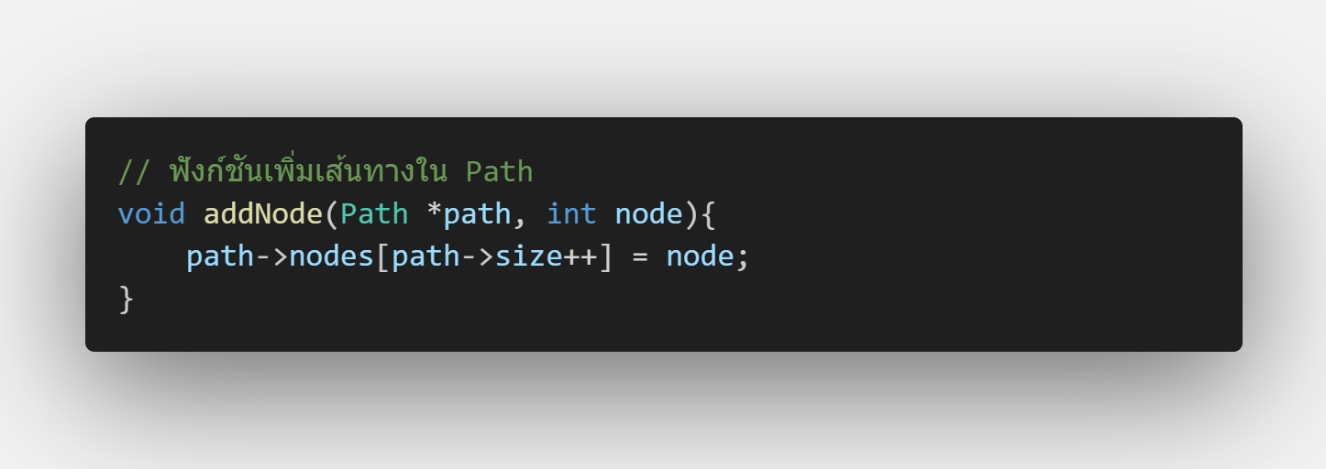
**กราฟไม่มีทิศทาง (Undirected Graph) และการหาวงจรออยเลอร์ (Euler Circuits)**



โครงสร้างข้อมูล Path ที่กำหนดด้วย typedef struct นี้ใช้เพื่อเก็บข้อมูลของเส้นทางในกราฟที่ถูกค้นพบหรือใช้งานในทางไปมา

**โดยประกอบด้วยสองสมาชิกหลักคือ**

1. nodes (int pointer): เป็นตัวแปรที่ชี้ไปยังพื้นที่หน่วยความจำที่เก็บเส้นทางในกราฟ คือลำดับของ nodes ที่เป็นส่วนหนึ่งของเส้นทาง
2. size (int): เก็บขนาดของเส้นทาง หรือจำนวนของ nodes ที่อยู่ในเส้นทาง



ฟังก์ชัน addNode ในโปรแกรมนี้ถูกใช้เพื่อเพิ่มเส้นทาง (node) ลงในโครงสร้างข้อมูล Path ที่ถูกสร้างขึ้นเพื่อเก็บเส้นทางในกราฟ

**หลักการทำงานของฟังก์ชัน**

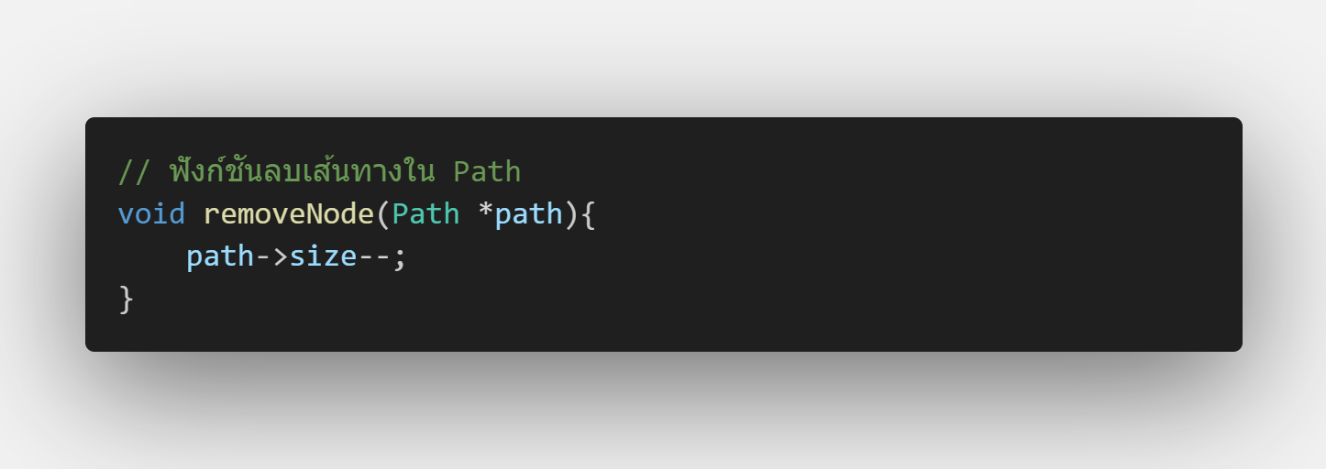
1. การรับข้อมูล:

* รับอาร์กิวเมนต์ path ที่เป็นตัวแปรชนิด Path ที่ถูกส่งเข้ามา
* รับอาร์กิวเมนต์ node ที่เป็น node ที่ต้องการเพิ่มในเส้นทาง

1. การเข้าถึงข้อมูลใน Path:

* path->nodes เป็นการเข้าถึงอาร์เรย์ของ nodes ใน Path
* path->size เป็นขนาดปัจจุบันของ Path หลังจากการเพิ่ม node

3.การเพิ่มเส้นทาง:

* path->nodes[path->size++] = node; ทำการเพิ่ม node ลงใน Path และเพิ่มขนาดของ Path ทีละหนึ่งหลัก

ฟังก์ชัน removeNode ในโปรแกรมนี้ถูกใช้เพื่อลบเส้นทางล่าสุดออกจากโครงสร้างข้อมูล Path ที่ถูกสร้างขึ้นเพื่อเก็บเส้นทางในกราฟ

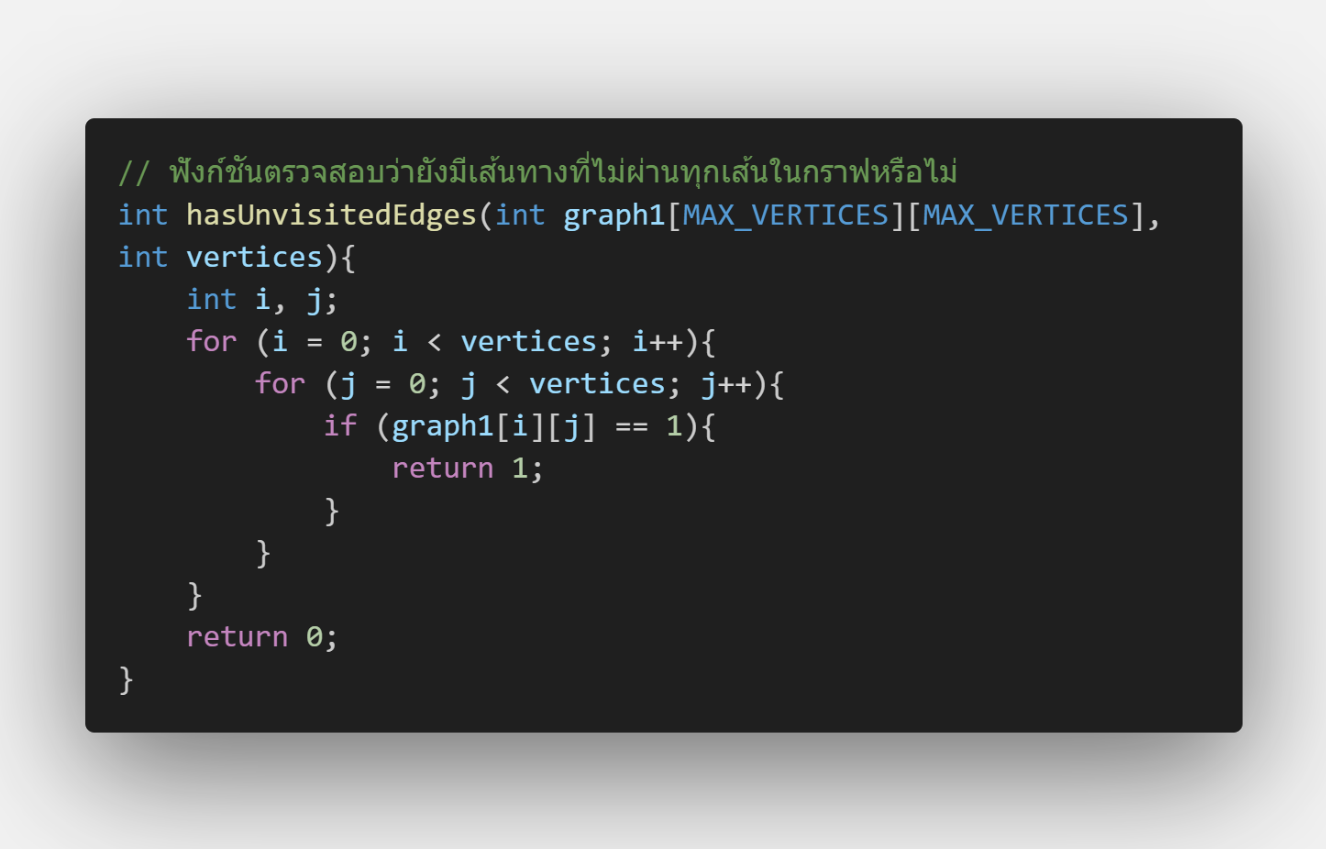
**หลักการทำงานของฟังก์ชัน**

1. การลบเส้นทาง:

* path->size--; ทำการลดขนาดของ Path ทีละหนึ่งหลัก

1. การเข้าถึงข้อมูลใน Path:

* path->size เป็นขนาดปัจจุบันของ Path ก่อนที่จะทำการลบเส้นทาง



ฟังก์ชัน hasUnvisitedEdges ถูกใช้เพื่อตรวจสอบว่ายังมีเส้นทางที่ไม่ได้ผ่านทุกเส้นในกราฟ

หรือไม่ โดยในกราฟที่เป็นวงจร Euler, เส้นทางทุกเส้นจะถูกใช้เพียงครั้งเดียว ดังนั้นเมื่อทุกเส้นทางถูกผ่านแล้ว ฟังก์ชันจะคืนค่า 0 แสดงว่าไม่มีเส้นทางที่ยังไม่ได้ผ่านทุกเส้นในกราฟ

**หลักการทำงานของฟังก์ชัน**

1. การรับข้อมูล:

* graph1 เป็น adjacency matrix ที่ใช้เก็บข้อมูลการเชื่อมต่อระหว่าง nodes ในกราฟ
* vertices เป็นจำนวนของ nodes ในกราฟ

1. การตรวจสอบเส้นทางที่ไม่ผ่านทุกเส้นในกราฟ:

* for (i = 0; i < vertices; i++) ทำการวนลูปตาม nodes ในกราฟ
* for (j = 0; j < vertices; j++) ทำการวนลูปตาม nodes ในกราฟอีกครั้ง
* if (graph1[i][j] == 1) ตรวจสอบว่ามีเส้นทางที่เชื่อมต่อระหว่าง nodes i และ j หรือไม่
* หากพบเส้นทางที่ยังไม่ผ่านทุกเส้น ฟังก์ชันจะทันทีคืนค่า 1

1. การคืนค่า:

* หากไม่มีเส้นทางที่ยังไม่ผ่านทุกเส้นในกราฟ, ฟังก์ชันจะคืนค่า 0



ฟังก์ชัน findEuler ในโปรแกรมนี้ถูกใช้เพื่อหาวงจร Euler ในกราฟ

**หลักการทำงานของฟังก์ชัน**

1. การเตรียมข้อมูล:

* สร้างโครงสร้างข้อมูล Path ที่ชื่อ path เพื่อเก็บเส้นทางในกราฟ
* ใช้ malloc ในการจองพื้นที่หน่วยความจำสำหรับ path.nodes เพื่อเก็บ nodes ในเส้นทาง
* กำหนด path.size เป็น 0 เนื่องจากยังไม่มีเส้นทางในตำแหน่งใด

1. การเพิ่มจุดเริ่มต้นในเส้นทาง:

* เพิ่มจุดเริ่มต้น (startNode) ใน path โดยใช้ฟังก์ชัน addNode

1. การวนลูปเพื่อหาวงจร Euler:

* ใช้ while (hasUnvisitedEdges(graph1, numVertices)) เพื่อวนลูปจนกว่าจะไม่มีเส้นทางที่ยังไม่ได้ผ่านทุกเส้นในกราฟ
* หากยังมีเส้นทางที่ไม่ได้ผ่านทุกเส้น, ทำการวนลูปเพื่อหาว่าจะเดินไปยัง node ถัดไปได้อย่างไร
* int currentNode = path.nodes[path.size - 1]; ใช้เข้าถึง node ปัจจุบันที่อยู่ในทางเดิน
* for (nextNode = 0; nextNode < numVertices; nextNode++) ใช้วนลูปเพื่อหา node ถัดไปที่เชื่อมต่อกับ node ปัจจุบัน
* if (graph1[currentNode][nextNode] == 1) ตรวจสอบว่ามีเส้นทางเชื่อมต่อไปยัง node ถัดไปหรือไม่
* addNode(&path, nextNode); เพิ่ม node ถัดไปลงใน path
* graph1[currentNode][nextNode] = 0; และ graph1[nextNode][currentNode] = 0; ลบเส้นทางที่ถูกใช้ไปแล้วจาก adjacency matrix

1. การแก้ไขกรณีไม่พบเส้นทางที่เชื่อมต่อกับโหนดปัจจุบัน:

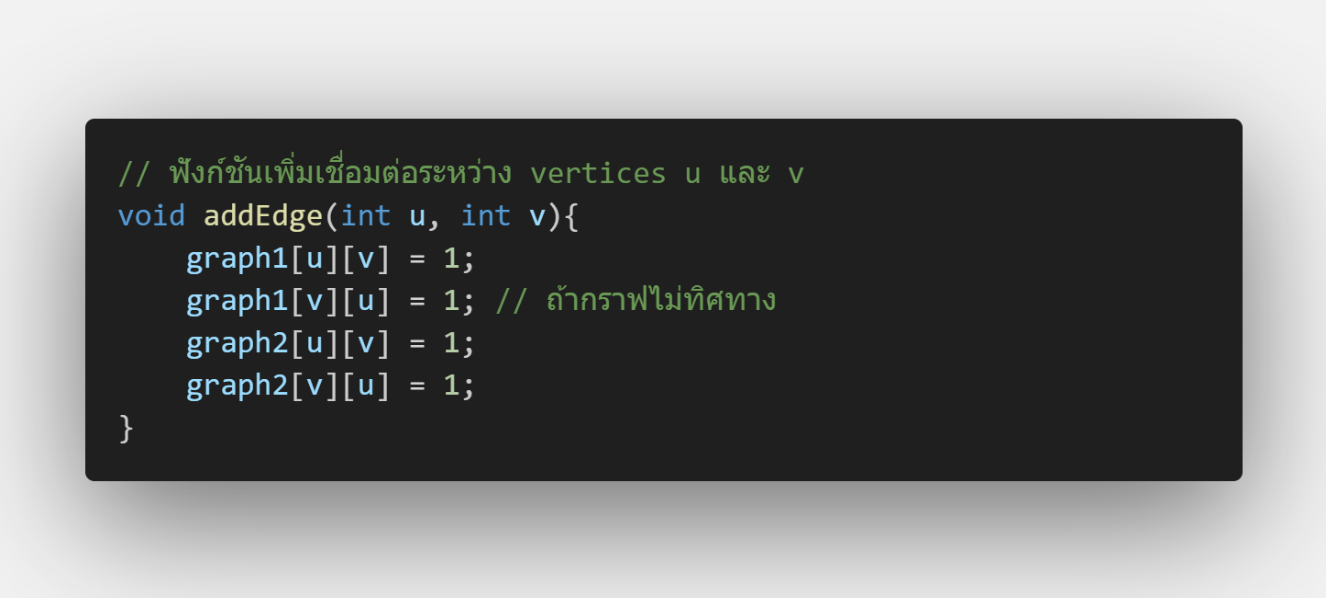
* ถ้า nextNode ไม่เป็นไปตามลูป for ทั้งหมด, แสดงว่าไม่พบเส้นทางที่เชื่อมต่อกับโหนดปัจจุบัน
* ในกรณีนี้, ให้ลบ node ล่าสุดที่เพิ่มเข้าไปใน path ออก โดยใช้ฟังก์ชัน removeNode

1. การแสดงผลลัพธ์:

* เมื่อไม่มีเส้นทางที่ยังไม่ผ่านทุกเส้นในกราฟ, ทำการแสดงผลลัพธ์ที่ได้จากการหาวงจร Euler
* ใช้ลูป for เพื่อแสดง nodes ทั้งหมดใน path ซึ่งเป็นวงจร Euler

1. การคืนทรัพยากร:

* ใช้ free(path.nodes); เพื่อคืนทรัพยากรที่ถูกจองไว้สำหรับ path.nodes

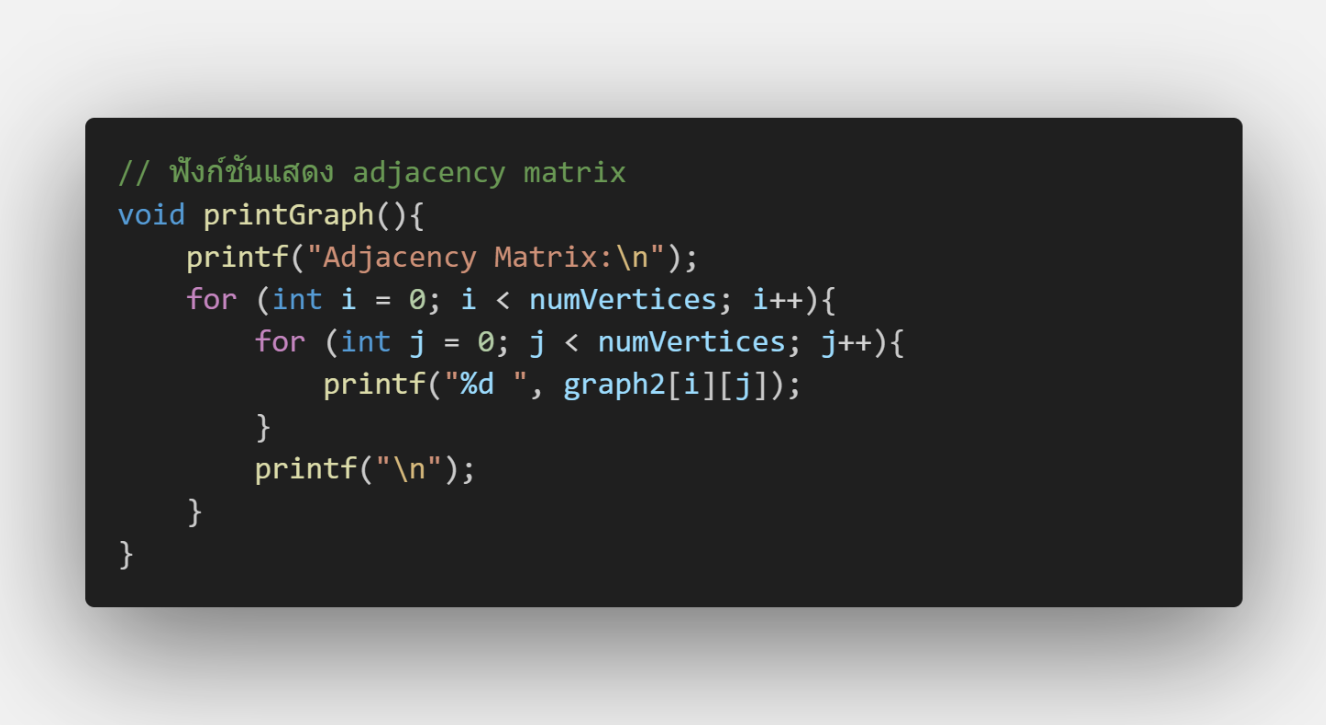
ฟังก์ชัน addEdge ถูกใช้เพื่อเพิ่มเชื่อมต่อ (edge) ระหว่าง nodes u และ v ในกราฟ

**หลักการทำงานของฟังก์ชัน**

1. การเพิ่มเชื่อมต่อใน adjacency matrix:

* graph1[u][v] = 1; ทำการเปลี่ยนค่าใน adjacency matrix graph1 ที่ตำแหน่ง (u, v) เป็น 1 หมายถึงมีเชื่อมต่อระหว่าง nodes u และ v
* graph1[v][u] = 1; ทำการเปลี่ยนค่าใน adjacency matrix graph1 ที่ตำแหน่ง (v, u) เป็น 1 เนื่องจากกราฟไม่มีทิศทาง

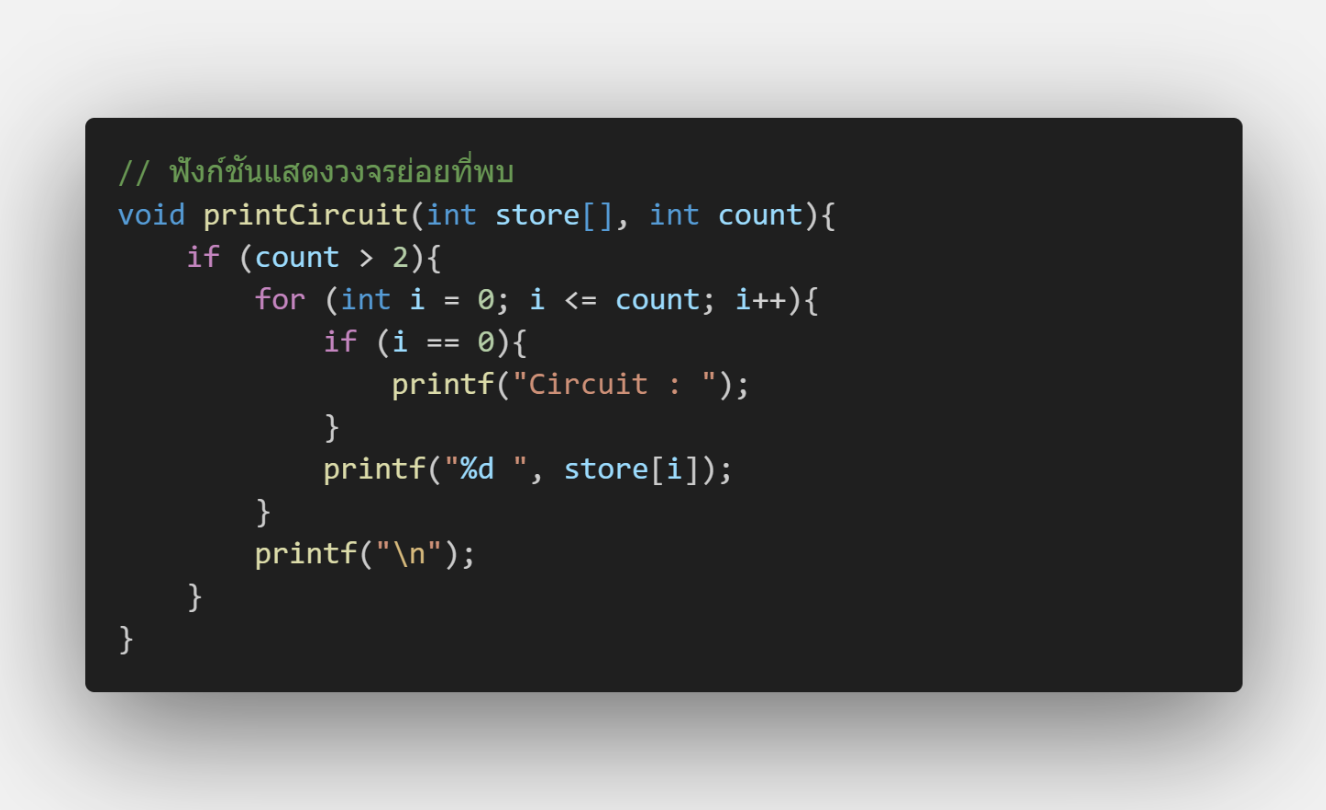
1. การเพิ่มเชื่อมต่อใน adjacency matrix สำหรับการแสดงผล:

* graph2[u][v] = 1; และ graph2[v][u] = 1; ทำการเปลี่ยนค่าใน adjacency matrix graph2 เพื่อให้นำไปใช้ในการแสดงผล adjacency matrix

ฟังก์ชัน printGraph ถูกใช้เพื่อแสดง adjacency matrix ของกราฟที่ถูกเก็บไว้ใน graph2

หลักการทำงานของฟังก์ชัน

**การแสดงผล:**

* printf("Adjacency Matrix:\n"); แสดงข้อความ "Adjacency Matrix:" เพื่อแจ้งให้ผู้ใช้ทราบว่าข้อมูลที่ตามมาเป็น adjacency matrix
* for (int i = 0; i < numVertices; i++) ใช้ลูปเพื่อวนตาม nodes ในกราฟ
* for (int j = 0; j < numVertices; j++) ใช้ลูปเพื่อวนตาม nodes ที่เชื่อมต่อกับ node i
* printf("%d ", graph2[i][j]); แสดงค่าที่ตำแหน่ง (i, j) ใน adjacency matrix ซึ่งหมายถึงการเชื่อมต่อระหว่าง nodes i และ j
* printf("\n"); ทำการขึ้นบรรทัดใหม่หลังจากแสดงค่าทุกค่าในแถวนั้นเสร็จ

ฟังก์ชัน printCircuit ในโปรแกรมนี้ถูกใช้เพื่อแสดงวงจรย่อยที่พบในกราฟ

**หลักการทำงานของฟังก์ชัน**

1. การตรวจสอบความเหมาะสมของวงจรย่อย:

* if (count > 2) ตรวจสอบว่าจำนวน nodes ที่เก็บใน store เป็นมากกว่า 2 หรือไม่ เพื่อที่จะถือว่าเป็นวงจร, จำเป็นต้องมีอย่างน้อย 3 nodes

1. การแสดงผล:

* ถ้าจำนวน nodes ที่มีใน store มากกว่า 2:
* for (int i = 0; i <= count; i++) ใช้ลูปเพื่อวนตาม nodes ที่เก็บใน store
* if (i == 0) ทำการตรวจสอบว่าเป็นการแสดงผลครั้งแรกหรือไม่
* ถ้าเป็นครั้งแรก printf("Circuit : "); จะแสดงข้อความ "Circuit : " เพื่อแสดงถึงว่าจะเริ่มแสดง nodes ในวงจร
* printf("%d ", store[i]); แสดงค่า node ที่ตำแหน่ง i ใน store
* printf("\n"); ทำการขึ้นบรรทัดใหม่หลังจากแสดงผล nodes ทั้งหมดใน store



ฟังก์ชัน findCircuit ในโปรแกรมนี้ถูกใช้เพื่อหาวงจรย่อยในกราฟ

**หลักการทำงานของฟังก์ชัน**

1. การวนลูปเพื่อหาวงจรย่อย:

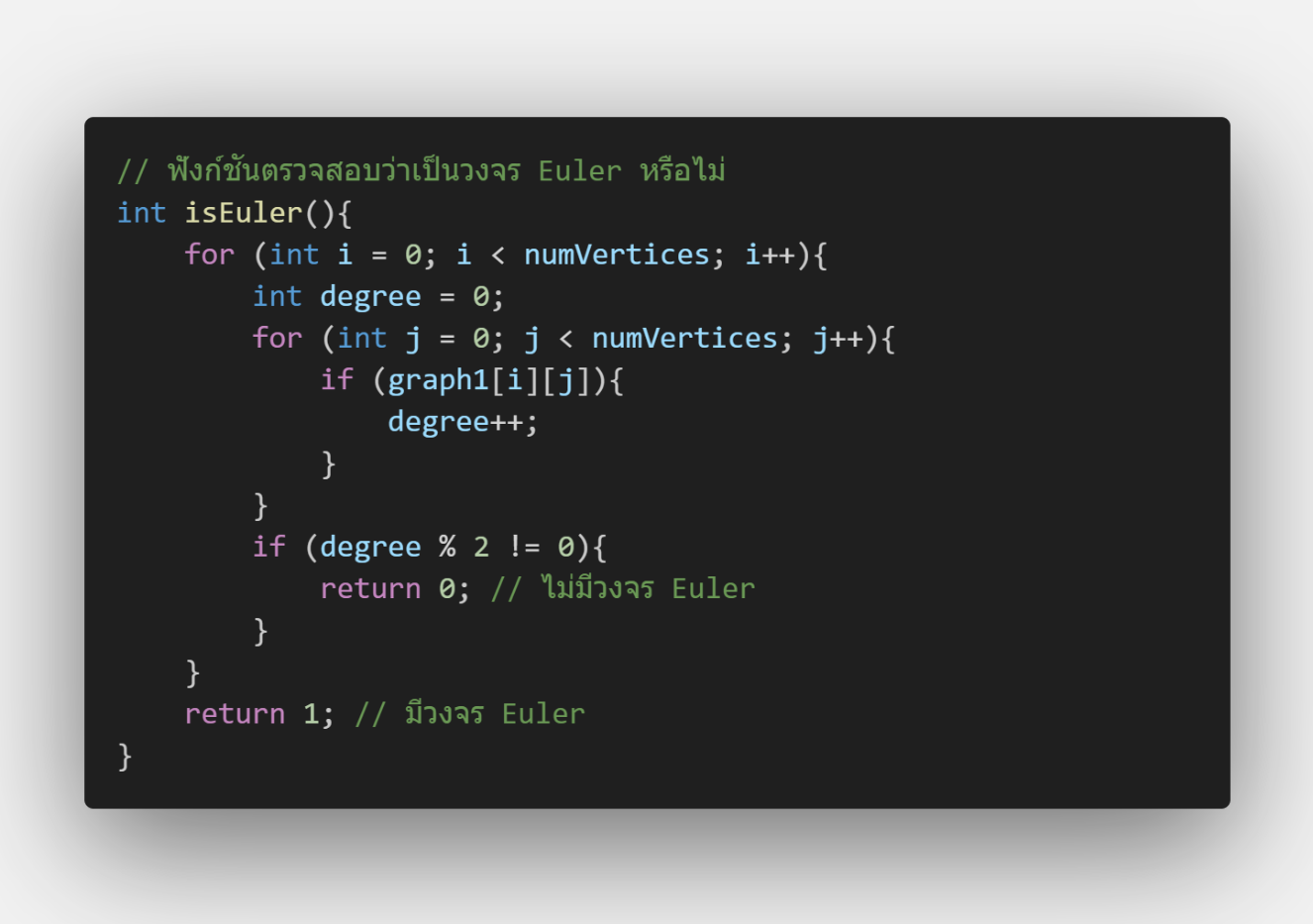
* for (int i = 0; i < numVertices; i++) ใช้ลูปเพื่อวนตาม nodes ในกราฟ
* สำหรับแต่ละ node i, ทำการตรวจสอบการเชื่อมต่อระหว่าง nodes i และ j ทั้งในทิศทางจาก i ไปยัง j และจาก j ไปยัง i
* ถ้ามีการเชื่อมต่อทั้งสองทิศทาง (graph2[i][j] == 1 และ graph2[j][i] == 1):
* เพิ่ม j ลงใน store และลดค่าเชื่อมต่อทั้งสองทิศทางใน graph2 ลงเป็น 0
* ทำการวนลูปต่อไปเพื่อตรวจสอบการเชื่อมต่อทั้งสองทิศทางต่อไป
* หลังจากนั้น, ทำการตรวจสอบว่าวงจรย่อยที่พบมีความยาวมากกว่า 2 หรือไม่

1. การตรวจสอบเพิ่มเติม:

* for (int w = 0; w < count; w++) ใช้ลูปเพื่อตรวจสอบ nodes ที่เก็บใน store
* int node1 = store[w]; และ int node2 = store[w + 1]; เก็บค่า nodes ที่ตำแหน่ง w และ w + 1
* ถ้ายังมีการเชื่อมต่อทั้งสองทิศทาง (graph2[node1][node2] == 1 และ graph2[node2][node1] == 1):
* เพิ่ม i ลงใน store เพื่อทำให้เป็นวงจรปิด
* ลดค่าเชื่อมต่อทั้งสองทิศทางใน graph2 ลงเป็น 0

1. การแสดงผล:

* หลังจากที่เก็บ nodes ใน store เสร็จ, ทำการเรียกใช้ฟังก์ชัน printCircuit(store, count) เพื่อแสดงผลลัพธ์



ฟังก์ชัน isEuler ในโปรแกรมนี้ถูกใช้เพื่อตรวจสอบว่ากราฟมีวงจร Euler หรือไม่

**หลักการทำงานของฟังก์ชัน**

1. การตรวจสอบระดับของทุกโหนด:

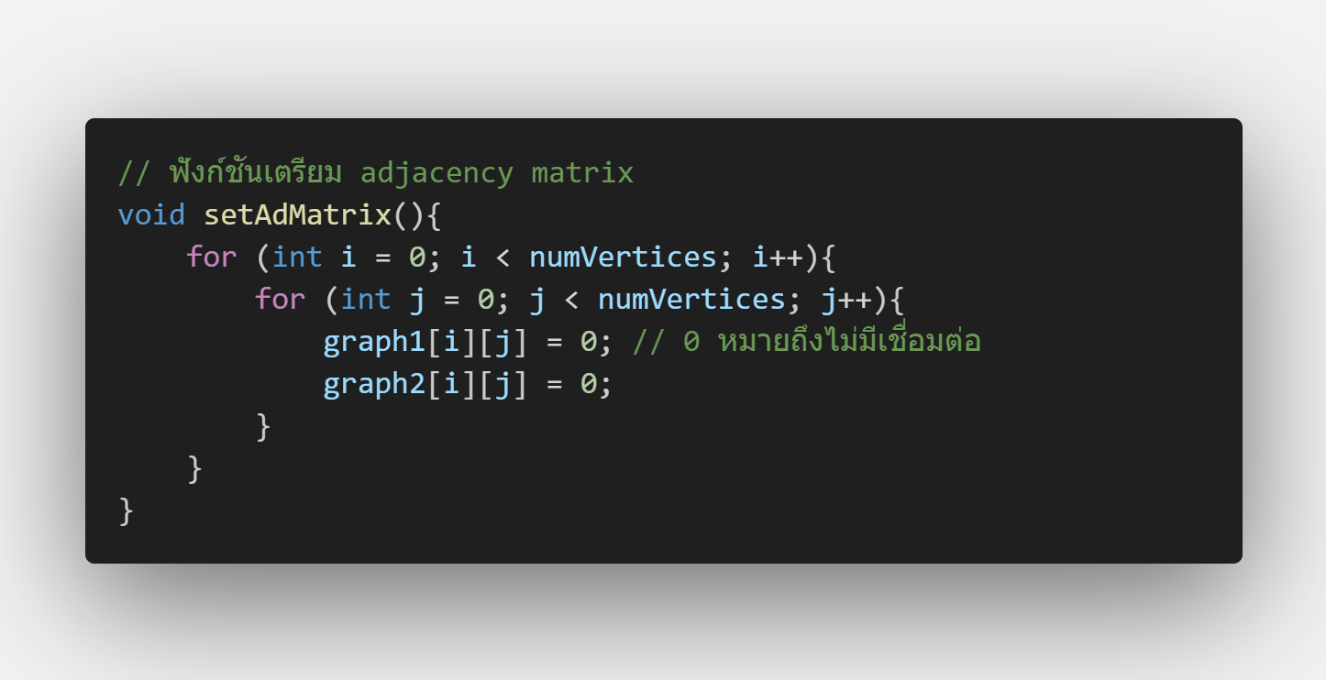
* for (int i = 0; i < numVertices; i++) ใช้ลูปเพื่อวนตามทุกโหนดในกราฟ
* สำหรับแต่ละโหนด i, ทำการนับจำนวนเชื่อมต่อที่เชื่อมกับโหนด i (degree)
* ถ้าจำนวน degree ของโหนด i เป็นเลขคู่ (degree % 2 == 0), แสดงว่าโหนดนี้มี degree เป็นเลขคู่

1. การตรวจสอบความเหมาะสมของทุกโหนด:

* ถ้ามีโหนดใดๆ ที่มี degree เป็นเลขคี่, ฟังก์ชันจะทันทีส่งค่ากลับ (return 0;) แสดงว่าไม่มีวงจร Euler ในกราฟ

1. การคืนค่า:

* ถ้าทุกโหนดมี degree เป็นเลขคู่, ฟังก์ชันจะคืนค่า 1 (return 1;) แสดงว่ากราฟมีวงจร Euler



ฟังก์ชัน setAdMatrix ในโปรแกรมนี้ถูกใช้เพื่อเตรียม adjacency matrix โดยกำหนดค่าเริ่มต้นให้ทุกระยะห่างของเส้นเชื่อมเป็น 0

**หลักการทำงานของฟังก์ชัน**

1. การลูปเพื่อกำหนดค่าใน adjacency matrix:

* for (int i = 0; i < numVertices; i++) ใช้ลูปเพื่อวนตามทุกโหนดในกราฟ
* for (int j = 0; j < numVertices; j++) ใช้ลูปเพื่อวนตามทุกระยะห่างของเส้นเชื่อมที่เป็นไปได้
* graph1[i][j] = 0; และ graph2[i][j] = 0; กำหนดค่าใน adjacency matrix ที่ตำแหน่ง (i, j) เป็น 0 หมายถึงไม่มีเส้นเชื่อมระหว่าง nodes i และ j

1. การตรวจสอบความเหมาะสมของทุกโหนด:

* หลังจากลูปทั้งสองชั้นเสร็จ, adjacency matrix ในทั้ง graph1 และ graph2 จะถูกกำหนดให้ทุกระยะห่างของเส้นเชื่อมเป็น 0

**Function Main()**



**ตัวอย่างการแสดงผลกรณีกราฟไม่มีวงจร Euler**

กำหนด

numVertices = 5;

numEdges = 7;

หากทำการเพิ่มเส้นเชื่อมดังนี้

addEdge(0, 1);

addEdge(0, 2);

addEdge(1, 3);

addEdge(1, 4);

addEdge(2, 1);

addEdge(2, 4);

addEdge(3, 4);

การแสดงผลที่ได้

Adjacency Matrix:

0 1 1 0 0

1 0 1 1 1

1 1 0 0 1

0 1 0 0 1

0 1 1 1 0

No Eulerian circuit exists.

Subcircuits included in the graph

Circuit : 0 1 2 0

Circuit : 1 3 4 1

**ตัวอย่างการแสดงผลกรณีกราฟมีวงจร Euler**

กำหนด

numVertices = 7;

numEdges = 12;

หากทำการเพิ่มเส้นเชื่อมดังนี้

addEdge(0, 1);

addEdge(0, 5);

addEdge(1, 5);

addEdge(1, 6);

addEdge(1, 2);

addEdge(2, 6);

addEdge(2, 3);

addEdge(2, 4);

addEdge(3, 4);

addEdge(4, 6);

addEdge(4, 5);

addEdge(5, 6);

การแสดงผลที่ได้

Adjacency Matrix:

0 1 0 0 0 1 0

1 0 1 0 0 1 1

0 1 0 1 1 0 1

0 0 1 0 1 0 0

0 0 1 1 0 1 1

1 1 0 0 1 0 1

0 1 1 0 1 1 0

Eulerian circuit.

Enter the starting node: 0

Euler Circuit: 0 1 2 3 4 2 6 1 5 4 6 5 0

Subcircuits included in the graph

Circuit : 0 1 5 0

Circuit : 1 2 6 1

Circuit : 2 3 4 2

Circuit : 4 5 6 4

**ฟังก์ชันหลักในโปรแกรมแปลง Infix เป็น Postfix โดยใช้ Stack**

****

โครงสร้างข้อมูลที่ชื่อ node ที่ถูกสร้างขึ้นในโปรแกรมนี้มีลักษณะเป็นโหนดของ stack ที่ใช้เก็บตัวอักษร (characters) และมีการเชื่อมโยงกับโหนดถัดไปของ stack ด้วยพอยน์เตอร์ next

**ลักษณะหลักของโครงสร้างข้อมูล:**

1. c (char):

* เป็นตัวแปรที่เก็บตัวอักษร (character) ในโหนดนั้น

1. next (struct nd):\*

* เป็นพอยน์เตอร์ที่ชี้ไปยังโหนดถัดไปใน stack

1. top (struct nd):\*

* เป็นตัวแปรที่เก็บพอยน์เตอร์ที่ชี้ไปยังโหนดบนสุดของ stack

****

ฟังก์ชัน push ถูกใช้เพื่อเพิ่มข้อมูลลงใน stack ของตัวแปรชนิด node

**หลักการทำงานของฟังก์ชัน**

1. การสร้างโหนดใหม่:

* node \*n = malloc(sizeof(node)); ทำการจองหน่วยความจำใน heap เพื่อสร้างโหนดใหม่ขนาดของโครงสร้าง node

1. การเชื่อมโยงโหนดใหม่กับ stack:

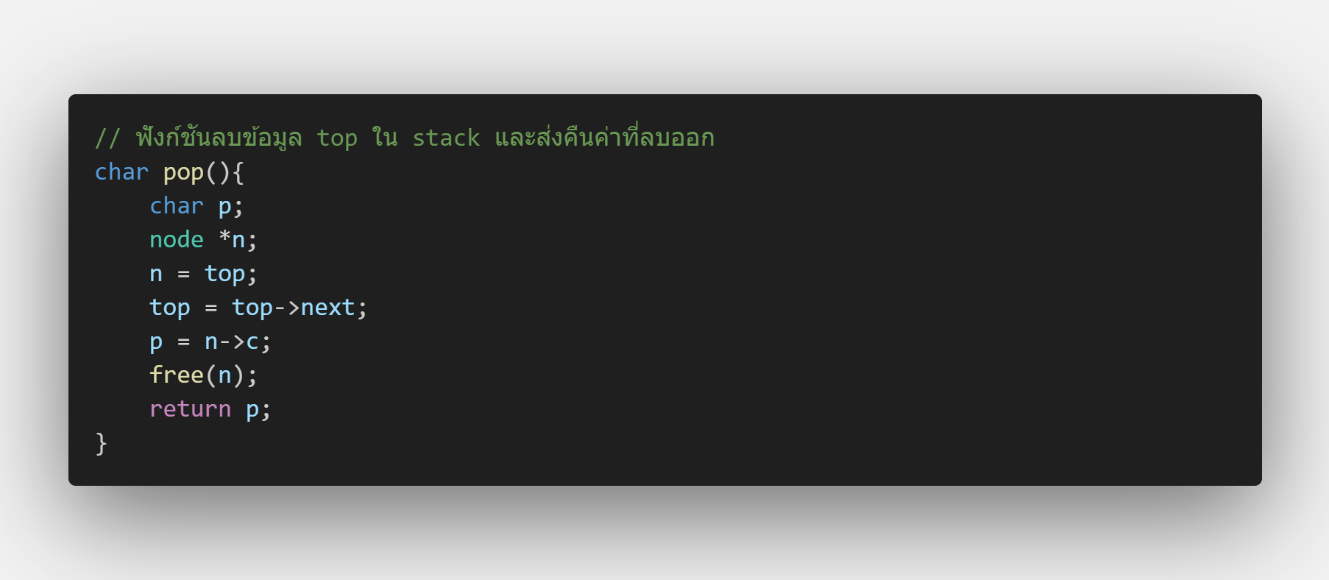
* n->next = top; กำหนดพอยน์เตอร์ next ของโหนดใหม่ให้ชี้ไปยังโหนดบนสุดของ stack เดิม

1. การทำให้ top ชี้ไปยังโหนดใหม่:

* top = n; ทำให้ตัวแปร top ชี้ไปยังโหนดใหม่ที่ถูกเพิ่มเข้า stack

1. การเก็บข้อมูลลงในโหนด:

* n->c = x; กำหนดค่าข้อมูลในโหนดใหม่เป็นค่า x ที่รับเข้ามา

****

ฟังก์ชัน pop ถูกใช้ในการลบข้อมูลจาก stack และส่งคืนค่าที่ถูกลบออก

**หลักการทำงานของฟังก์ชัน**

1. การเก็บค่าที่จะส่งคืน:

* char p; ประกาศตัวแปร p สำหรับเก็บค่าที่จะส่งคืน

1. การเก็บโหนดที่จะถูกลบ:

* node \*n; ประกาศพอยน์เตอร์ n ที่ใช้เก็บโหนดที่จะถูกลบ

1. การเลื่อน top ไปยังโหนดถัดไป:

* top = top->next; กำหนด top ให้ชี้ไปยังโหนดถัดไปใน stack.

1. การเก็บค่าข้อมูลที่ถูกลบ:

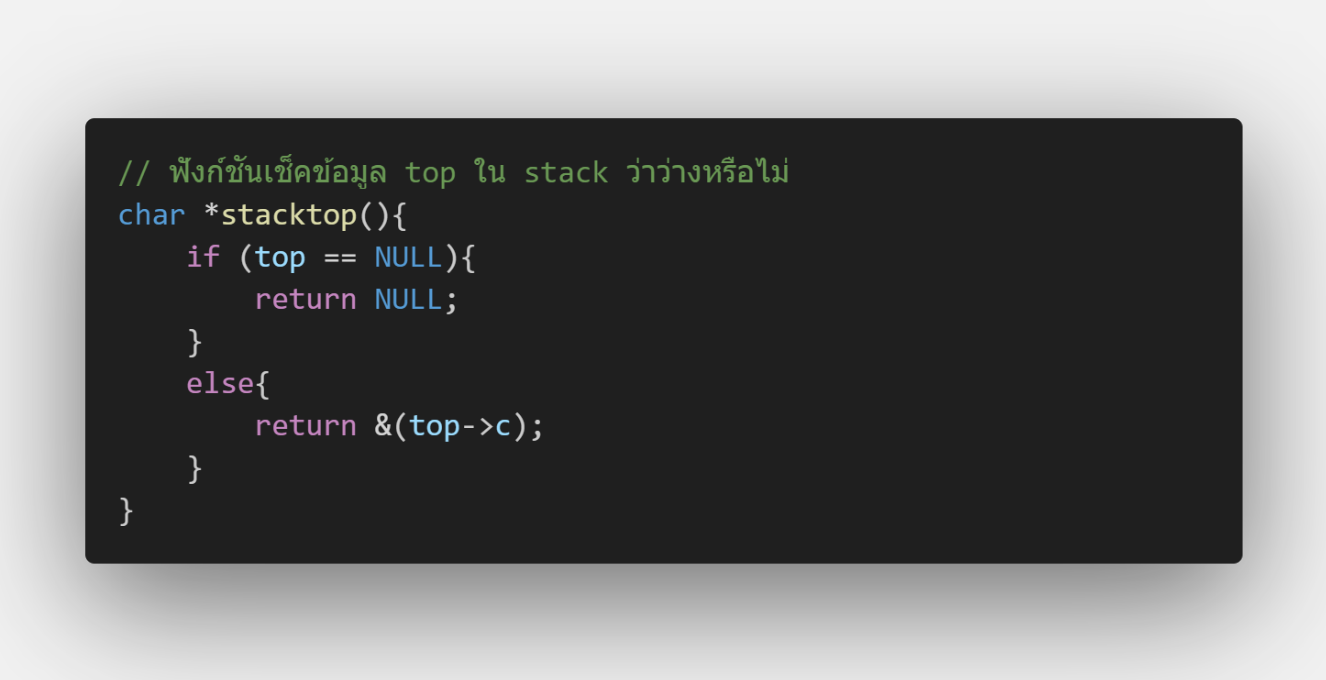
* p = n->c; เก็บค่าข้อมูลที่ถูกลบจากโหนดไว้ในตัวแปร p

1. การลบโหนด:

* free(n); ทำการคืนหน่วยความจำที่ถูกจองสำหรับโหนดที่ถูกลบ

1. การส่งคืนค่าที่ถูกลบ:

* return p; ส่งคืนค่าข้อมูลที่ถูกลบ

****

ฟังก์ชัน stacktop ถูกใช้ในการตรวจสอบว่า stack ว่างหรือไม่ และถ้าไม่ว่างจะส่งคืนข้อมูลที่อยู่ที่ top ของ stack

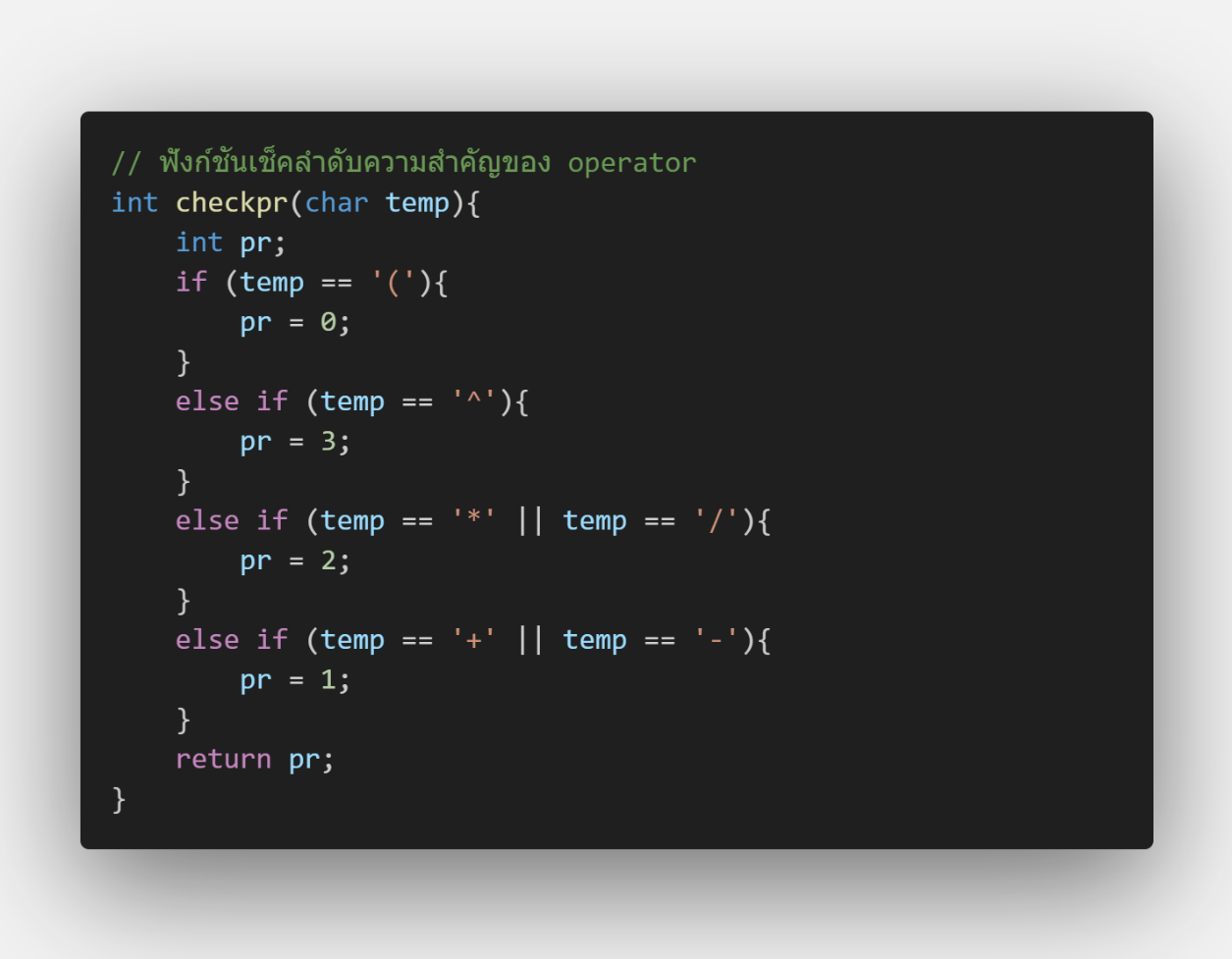
**หลักการทำงานของฟังก์ชัน**

1. การตรวจสอบว่า stack ว่างหรือไม่:

* if (top == NULL) ทำการตรวจสอบว่าตัวแปร top ที่ใช้เก็บหน่วยความจำของโหนดบนสุดของ stack มีค่าเป็น NULL หรือไม่

1. การส่งคืนค่าตามเงื่อนไข:

* ถ้า top เป็น NULL ส่งคืนค่า NULL เพื่อแสดงว่า stack ว่าง
* ถ้า top ไม่เป็น NULL ส่งคืนตำแหน่งของข้อมูลที่อยู่ที่ top ของ stack โดยใช้ &(top->c)

****

ฟังก์ชัน checkpr ถูกใช้ในการตรวจสอบลำดับความสำคัญของ operator

**หลักการทำงานของฟังก์ชัน**

1. การรับค่าพารามิเตอร์:

* char temp: รับค่า operator ที่ต้องการตรวจสอบลำดับความสำคัญ

1. การตรวจสอบค่า operator และกำหนดลำดับความสำคัญ:

* ถ้า temp เป็น '(' กำหนด pr เป็น 0
* ถ้า temp เป็น '^' กำหนด pr เป็น 3
* ถ้า temp เป็น '\*' หรือ '/' กำหนด pr เป็น 2
* ถ้า temp เป็น '+' หรือ '-' กำหนด pr เป็น 1

1. การส่งคืนค่าลำดับความสำคัญ:

* return pr; ส่งคืนค่าลำดับความสำคัญที่ได้จากการตรวจสอบ

****

ฟังก์ชัน checkoper ถูกใช้ในการตรวจสอบและจัดการกับ operator ที่ป้อนเข้ามา

**หลักการทำงานของฟังก์ชัน**

1. การรับค่าพารามิเตอร์:

* char \*ck รับค่า operator ที่ต้องการตรวจสอบ
* char ops[] รับค่าสตริงที่ใช้เก็บผลลัพธ์หลังจากการตรวจสอบ operator

1. การตรวจสอบ stack ว่าว่างหรือไม่:

* if (stacktop() == NULL): ถ้า stack ว่าง
* push(\*ck) ทำการเพิ่ม operator ลงใน stack

1. การตรวจสอบลำดับความสำคัญของ operator:

* else: ถ้า stack ไม่ว่าง
* if (checkpr(\*ck) <= checkpr(\*stacktop())): ถ้าลำดับความสำคัญของ operator ที่เข้ามาน้อยกว่าหรือเท่ากับลำดับความสำคัญของ operator ที่อยู่ที่ top ของ stack
* while ((stacktop() != NULL) && (checkpr(\*ck) <= checkpr(\*stacktop()))) ทำการ loop ในขณะที่ stack ไม่ว่าง และลำดับความสำคัญของ operator ที่เข้ามาน้อยกว่าหรือเท่ากับลำดับความสำคัญของ operator ที่อยู่ที่ top ของ stack
* char temp = pop(); ทำการ pop operator จาก stack
* strncat(ops, &temp, 1); เพิ่ม operator ที่ pop ได้ลงในสตริงผลลัพธ์

1. บันทึก operator ลงใน stack:

* push(\*ck); ทำการบันทึก operator ลงใน stack เนื่องจากลำดับความสำคัญของ operator ที่เข้ามามีลำดับความสำคัญมากกว่า operator ที่อยู่ที่ top ของ stack

****

ฟังก์ชัน printstack ถูกใช้ในการแสดงค่าที่อยู่ใน stack

**หลักการทำงานของฟังก์ชัน**

1. การรับค่าพารามิเตอร์:

* ไม่มีพารามิเตอร์ที่รับเข้ามา

1. การนับจำนวน node ใน stack:

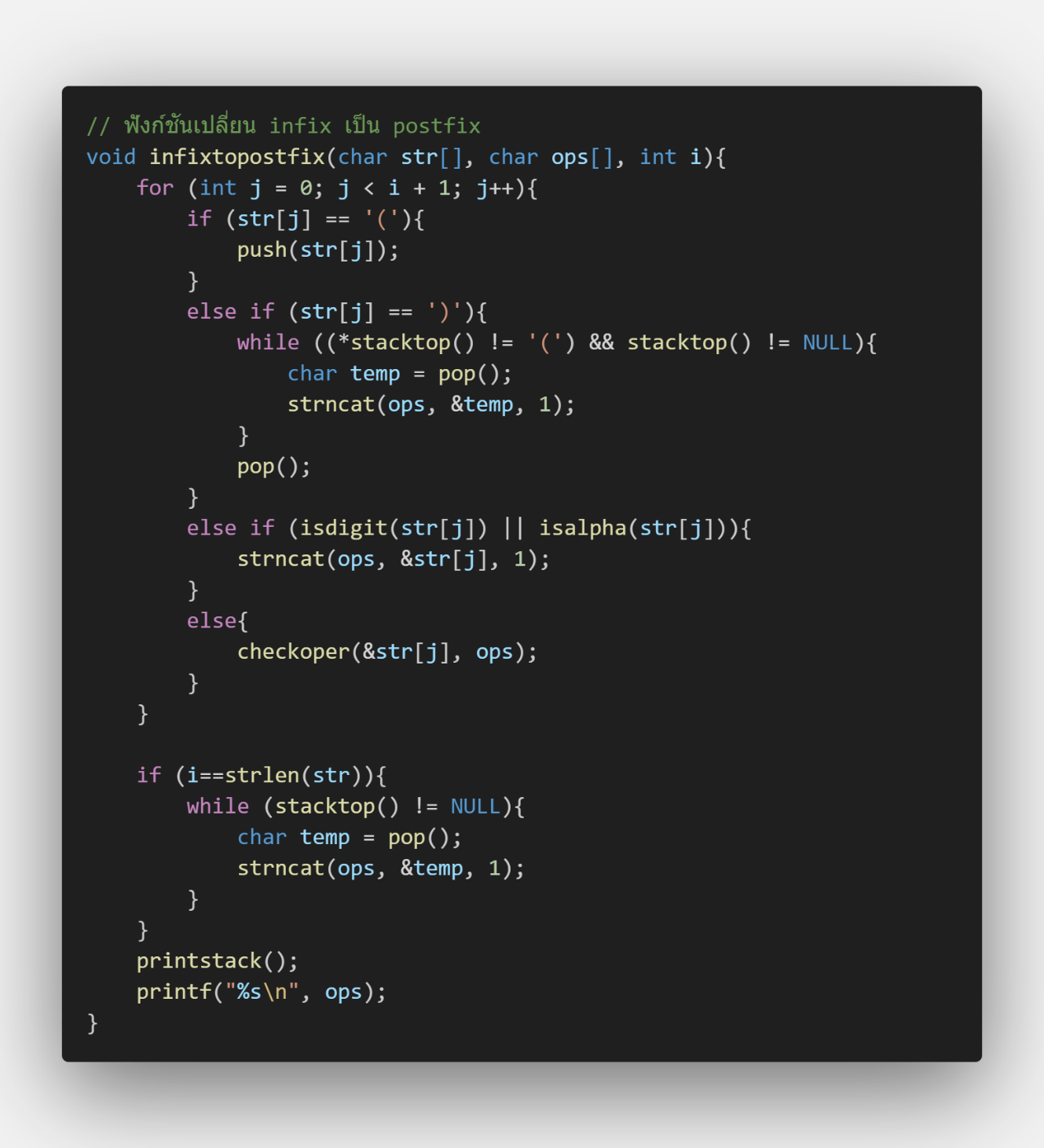
* node \*current = top; กำหนด pointer current ให้ชี้ไปที่ top ของ stack
* int count = 0; กำหนดตัวแปร count เพื่อเก็บจำนวน node ใน stack เริ่มต้นที่ 0
* while (current != NULL): วน loop ขณะที่ current ไม่ชี้ไปที่ NULL (node สุดท้ายของ stack)
* count++; เพิ่มค่า count ขึ้นทีละ 1
* current = current->next; ย้าย pointer current ไปยัง node ถัดไปใน stack

1. การเก็บค่าจาก stack ไว้ในสตริง:

* char oper[MAX\_STR] = ""; กำหนดสตริง oper เพื่อเก็บค่าที่จะแสดง
* for (int i = 0; i < count; i++): วน loop จำนวนครั้งเท่ากับจำนวน node ใน stack
* node \*step = top; กำหนด pointer step ให้ชี้ไปที่ top ของ stack ในแต่ละรอบของ loop
* for (int j = count - i - 1; j > 0; j--) วน loop เพื่อเลื่อน pointer step ไปยัง node ที่ต้องการใน stack
* strncat(oper, &step->c, 1) เพิ่มค่าของ node ที่ step ชี้ไปใน stack ลงในสตริง oper

1. การแสดงค่าที่เก็บไว้ในสตริง:

* printf("%-13s", oper); แสดงค่าที่อยู่ในสตริง oper ในรูปแบบที่กำหนด

****

ฟังก์ชัน infixtopostfix ทำหน้าที่เปลี่ยนนิพจน์ infix เป็นนิพจน์ postfix

**หลักการทำงานของฟังก์ชัน**

1. การรับค่าพารามิเตอร์:

* char str[] เป็น array ที่เก็บนิพจน์ infix
* char ops[] เป็น array ที่เก็บนิพจน์ postfix
* int i: เป็นตัวแปรที่บ่งบอกถึงตำแหน่งปัจจุบันในนิพจน์ infix

1. การแปลง infix เป็น postfix:

* for (int j = 0; j < i + 1; j++) วน loop ตามตำแหน่งปัจจุบันในนิพจน์ infix
* if (str[j] == '(') ถ้าตัวอักษรที่พบคือ '(' ให้ทำการ push '(' เข้า stack
* else if (str[j] == ')') ถ้าตัวอักษรที่พบคือ ')' ให้ทำการ pop จนกว่าจะเจอ '(' และเพิ่มค่าที่ pop ได้ลงในนิพจน์ postfix
* else if (isdigit(str[j]) || isalpha(str[j])) ถ้าตัวอักษรที่พบเป็นตัวเลขหรือตัวอักษร ให้เพิ่มค่านั้นลงในนิพจน์ postfix
* else ถ้าตัวอักษรที่พบเป็น operator ให้ใช้ฟังก์ชัน checkoper เพื่อตรวจสอบลำดับความสำคัญของ operator และทำการ push หรือ pop ตามลำดับ

1. การทำงานเพิ่มเติมหลังจบนิพจน์ infix:

* if (i == strlen(str)) หลังจากวน loop เสร็จสิ้นทุกรอบ ให้ทำการ pop ทุกอย่างใน stack และเพิ่มค่าที่ pop ได้ลงในนิพจน์ postfix
* printstack() เรียกใช้ฟังก์ชัน printstack เพื่อแสดง stack ในแต่ละขั้นตอน
* printf("%s\n", ops) แสดงนิพจน์ postfix ที่ได้หลังจากแปลง

**Function main()**

****

**ตัวอย่างการแสดงผล**

**Example01:**

|  |  |  |  |
| --- | --- | --- | --- |
| Enter infic expression: A+B\*C/D | | | |
| Step | Symbol | Stack | Output |
| 1 | A |  | A |
| 2 | + | + | A |
| 3 | B | + | AB |
| 4 | \* | +\* | AB |
| 5 | C | +\* | ABC |
| 6 | / | +/ | ABC\* |
| 7 | D | +/ | ABC\*D |
| 8 |  |  | ABC\*D/+ |
| ผลลัพธ์ของนิพจน์แบบ Postfix คือ ABC\*D/+ | | | |

**Example02:**

|  |  |  |  |
| --- | --- | --- | --- |
| Enter infic expression: (A+B)/C\*D-E | | | |
| Step | Symbol | Stack | Output |
| 1 | ( | ( |  |
| 2 | A | ( | A |
| 3 | + | (+ | A |
| 4 | B | (+ | AB |
| 5 | ) |  | AB+ |
| 6 | / | / | AB+ |
| 7 | C | / | AB+C |
| 8 | \* | \* | AB+C/ |
| 9 | D | \* | AB+C/D |
| 10 | - | - | AB+C/D\* |
| 11 | E | - | AB+C/D\*E |
| 12 |  |  | AB+C/D\*E- |
| ผลลัพธ์ของนิพจน์แบบ Postfix คือ AB+C/D\*E- | | | |

**Example03:**

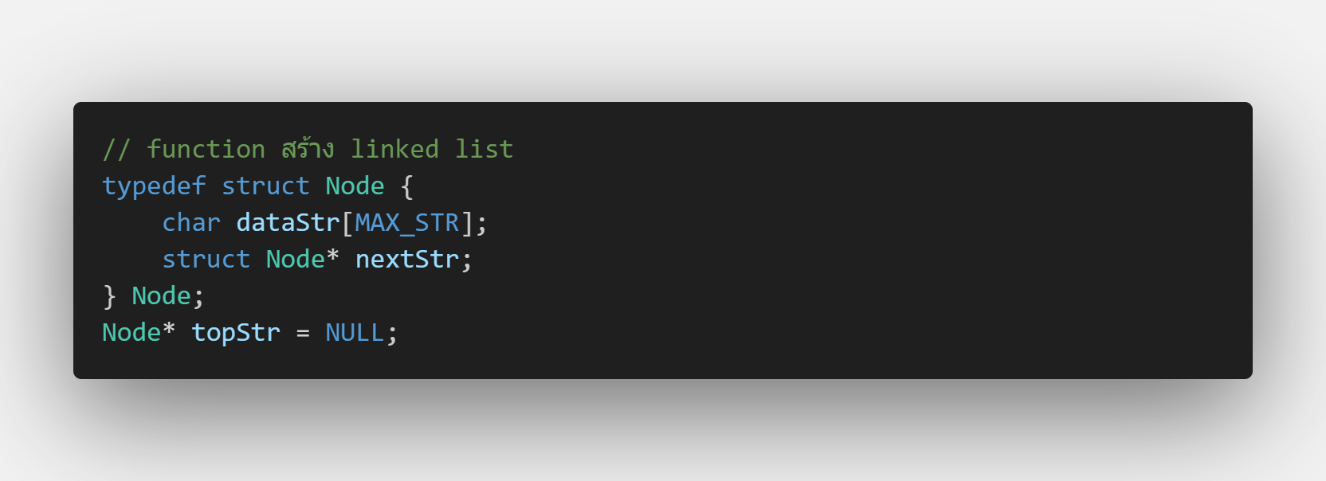
|  |  |  |  |
| --- | --- | --- | --- |
| Enter infic expression: (A+B)/C\*D-E | | | |
| Step | Symbol | Stack | Output |
| 1 |  |  | A |
| 2 |  | \* | A |
| 3 |  | \*( | A |
| 4 |  | \*( | AB |
| 5 |  | \*(+ | AB |
| 6 |  | \*(+ | ABC |
| 7 |  | \*(+^ | ABC |
| 8 |  | \*(+^ | ABCD |
| 9 |  | \* | ABCD^+ |
| 10 |  | - | ABCD^+\* |
| 11 |  | - | ABCD^+\*E |
| 12 |  | -^ | ABCD^+\*E |
| 13 |  | -^ | ABCD^+\*EF |
| 14 |  | -\* | ABCD^+\*EF^ |
| 15 |  | -\*( | ABCD^+\*EF^ |
| 16 |  | -\*( | ABCD^+\*EF^G |
| 17 |  | -\*(/ | ABCD^+\*EF^G |
| 18 |  | -\*(/ | ABCD^+\*EF^GH |
| 19 |  | -\* | ABCD^+\*EF^GH/ |
| 20 |  |  | ABCD^+\*EF^GH/\*- |
| ผลลัพธ์ของนิพจน์แบบ Postfix คือ ABCD^+\*EF^GH/\*- | | | |

**ฟังก์ชันหลักในโปรแกรมประมวลผลนิพจน์แบบ Postfix ให้เป็นผลลัพธ์**

**มีการใช้ฟังก์ชันหลักเหมือนกับโปรแกรมแปลง Infix เป็น Postfix โดยใช้ Stack ดังนี้**

1. สร้างโครงสร้างข้อมูลที่ชื่อ node
2. ฟังก์ชัน push(int x)
3. ฟังก์ชัน pop()
4. ฟังก์ชัน calculate(char \*ck)
5. ฟังก์ชัน printstack()

**ฟังก์ชันหลักที่ใช้เพิ่มเติม**



ในส่วนนี้ของโค้ดเป็นการประกาศโครงสร้างข้อมูลที่ใช้เพื่อสร้าง linked list สำหรับเก็บสตริง (string) ในโปรแกรม

**หลักการทำงานของส่วนนี้**

1. ประกาศโครงสร้างข้อมูล Node:

* typedef struct Node ใช้ในการประกาศโครงสร้างข้อมูลที่มีชื่อว่า Node
* โครงสร้างนี้ประกอบด้วย
* char dataStr[MAX\_STR] เป็นอาเรย์ขนาด MAX\_STR ที่ใช้เก็บข้อมูลสตริง.
* struct Node\* nextStr เป็นตัวแปรชนิดข้อมูล pointer ที่ใช้เก็บที่อยู่ของ Node ถัดไปใน linked list

1. ประกาศตัวแปร topStr:

* Node\* topStr = NULL; ใช้สร้างตัวแปร pointer ชื่อ topStr ที่ใช้เก็บที่อยู่ของ Node แรกของ linked list สำหรับสตริง
* เริ่มต้นที่ NULL แสดงว่า linked list ว่าง

****

ฟังก์ชัน pushStr ทำการเพิ่มข้อมูลลงใน stack สำหรับสตริงในโปรแกรม

**หลักการทำงานของฟังก์ชัน**

1. การสร้าง Node ใหม่:

* Node\* n = malloc(sizeof(Node)); ใช้ในการสร้าง Node ใหม่โดยใช้ฟังก์ชัน malloc เพื่อจองพื้นที่หน่วยความจำสำหรับ Node นั้นๆ
* Node นี้จะเป็น Node ที่จะถูกเพิ่มลงใน stack

1. การคัดลอกข้อมูลสตริง:

* strcpy(n->dataStr, x); ใช้ในการคัดลอกข้อมูลสตริงที่ถูกส่งเข้ามาผ่านพารามิเตอร์ x ไปยังสมาชิก dataStr ของ Node ที่ถูกสร้าง

1. การเชื่อม Node ใหม่กับ Stack:

* n->nextStr = topStr; ใช้ในการเชื่อม Node ใหม่กับ Node ที่อยู่ด้านบนของ stack (Node ที่เป็น topStr)
* นั้นหมายความว่า Node ใหม่นี้จะเป็น Node ที่อยู่ด้านบนสุดของ stack

1. การปรับ topStr:

* topStr = n; ทำการปรับ topStr เพื่อให้ชี้ไปที่ Node ใหม่ที่ถูกเพิ่มลงใน stack
* Node ใหม่นี้กลายเป็น Node ที่อยู่ด้านบนสุดของ stack

****

ฟังก์ชัน popStr ใช้ในการลบข้อมูลตัวสุดท้ายหรือตัวบนสุดใน stack สำหรับสตริง และส่งคืนค่าที่ถูกลบออกมา

**หลักการทำงานของฟังก์ชัน**

1. การจองหน่วยความจำสำหรับสตริง:

* char\* p = malloc(MAX\_STR \* sizeof(char)); ใช้ในการจองหน่วยความจำขนาด MAX\_STR สำหรับเก็บสตริงที่จะส่งคืน

1. การชี้ไปยัง Node ที่จะถูกลบ:

* Node\* n = topStr; ใช้ในการสร้างตัวแปร pointer n เพื่อชี้ไปยัง Node ที่ต้องการลบ (Node ที่อยู่ที่ topStr)

1. การปรับ topStr:

* topStr = topStr->nextStr; ทำการปรับ topStr เพื่อชี้ไปยัง Node ถัดไปใน linked list ซึ่งก็คือ Node ที่เป็นตัวบนสุดของ stack

1. การคัดลอกข้อมูลสตริง:

* strcpy(p, n->dataStr); ใช้ในการคัดลอกข้อมูลสตริงจาก Node ที่ถูกลบลงในตัวแปร p

1. การลบ Node:

* free(n); ใช้ในการลบ Node ที่อยู่ใน n, เนื่องจาก Node นี้ไม่ได้ใช้แล้วหลังจากที่ข้อมูลถูกคัดลอก

1. การส่งคืนค่าสตริง:

* return p; ทำการส่งคืนสตริงที่ถูกลบออกจาก stack

****

ฟังก์ชัน calculateStr ในโปรแกรมทำหน้าที่การดำเนินการทางคณิตศาสตร์หรือการเชื่อมต่อสตริงสองตัวด้วยตัวดำเนินการที่ถูกส่งเข้ามา (อย่างเช่น +, -, \*, /, ^), และนำผลลัพธ์ไปเก็บใน stack สำหรับสตริง

**หลักการทำงานของฟังก์ชัน**

1. การลบ Operand จาก Stack:

* char\* operand2 = popStr(); และ char\* operand1 = popStr(); ใช้ในการดึงข้อมูลสตริง operand ทั้งสองตัวจาก stack สำหรับสตริง

1. การดำเนินการทางคณิตศาสตร์หรือการเชื่อมต่อสตริง:

* ใช้ sprintf เพื่อสร้างสตริง temp ที่เป็นผลลัพธ์ของการดำเนินการ โดยใส่สตริง operand1, operator (\*ck) และ operand2 ในรูปแบบ (operand1 operator operand2) และเก็บไว้ในตัวแปร temp

1. การเพิ่มผลลัพธ์ลงใน Stack:

* pushStr(temp); เพื่อเพิ่มสตริงที่เป็นผลลัพธ์ลงใน stack สำหรับสตริง

****

ฟังก์ชัน printstackStr ทำหน้าที่แสดงค่าที่อยู่ใน stack สำหรับสตริง

**หลักการทำงานของฟังก์ชัน**

1. การหาจำนวน Node ใน Stack:

* ใช้ลูป while เพื่อหาจำนวน Node ทั้งหมดที่อยู่ใน stack สำหรับสตริง
* ในแต่ละรอบของลูป, ทำการเคลื่อนที่ไปยัง Node ถัดไปใน linked list โดยใช้ current = current->nextStr และเพิ่มตัวแปร count ขึ้นทีละ 1

1. การแสดงค่าจาก Stack:

* ใช้ลูป for เพื่อแสดงค่าจากทุก Node ที่อยู่ใน stack
* ในแต่ละรอบของลูป, ใช้ตัวแปร step ในการเคลื่อนที่ไปยัง Node ที่ต้องการด้วยการใช้ for loop สำหรับความสูงของ stack
* ตัวแปร j ใช้ในการเลื่อน Node ไปยังตำแหน่งที่ถูกต้องใน linked list
* step = step->nextStr; ใช้ในการเลื่อน Node ไปยัง Node ถัดไปใน linked list
* แสดงค่าที่อยู่ใน step->dataStr ที่เป็นข้อมูลสตริงของ Node นั้นๆ

1. การแสดง Separator:

* ในทุกรอบของลูป, ถ้าไม่ใช่รอบสุดท้าย (คือ i != count - 1), จะแสดง ‘, ’ เพื่อทำให้ผลลัพธ์มีลักษณะ value1, value2, ....

**Function Main()**

****

**ตัวอย่างการแสดงผล**

**Example01:**

|  |  |  |
| --- | --- | --- |
| Enter postfic expression: ABC\*D/+ | | |
| Step | Symbol | Stack |
| 1 | A | A |
| 2 | B | A, B |
| 3 | C | A, B, C |
| 4 | \* | A, (B\*C) |
| 5 | D | A, (B\*C), D |
| 6 | / | A, (B\*C)/D) |
| 7 | + | (A+((B\*C)/D)) |
| ผลลัพธ์ก็คือค่าสุดท้ายที่ได้ใน Stack ซึ่งคือ (A+((B\*C)/D)) | | |

**Example02:**

|  |  |  |
| --- | --- | --- |
| Enter postfic expression: 753\*5/+ | | |
| Step | Symbol | Stack |
| 1 | 7 | 7 |
| 2 | 5 | 7, 5 |
| 3 | 3 | 7, 5, 3 |
| 4 | \* | 7, 15 |
| 5 | 5 | 7, 15, 5 |
| 6 | / | 7, 3 |
| 7 | + | 10 |
| ผลลัพธ์ก็คือค่าสุดท้ายที่ได้ใน Stack ซึ่งคือ 10 | | |