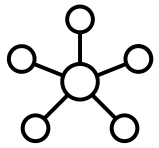


[제04주] 그래프 색칠

충남대학교 컴퓨터공학과 알고리즘 04 분반
제출일: 2021-10-12
학번: 201701975, 이름: 구건모

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

목차

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

과제 설명

- 주요 이론(BFS)
- 과제에서 해야할 일

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

주요 이론 및 과제에서 해야할 일

이번 과제의 주요 이론인 Breadth Frist Search(BFS)란 탐색 알고리즘 중 루트 노드에서부터 인접한 노드들을 우선적으로 탐색하여 모든 노드를 탐색하는 방식입니다. 쉽게 표현하여 깊게 탐색하기 전에 넓게 탐색하는 방식으로 볼 수 있으며 Queue를 이용하여 구현합니다. 최초에는 시작 노드를 Queue에 넣은 상태로 시작하며 Queue에서 element를 꺼내어 해당 노드에 대한 인접한 노드들을 방문하면서 방문한 노드들을 Queue에 삽입합니다. Queue의 element를 하나씩 꺼내어 해당 노드에 대한 인접 노드들에 대해 방문한 적이 없는 노드라면 방문하고 Queue에 넣는 작업을 반복합니다. 더 이상 Queue에서 꺼낼 원소가 없다면 순회를 마치게 됩니다. 이번 과제에서는 BFS를 이용하여 Graph를 순회하면서 vertex에 Color를 부여하는 기능을 구현하는 것입니다. BFS를 위해 필요한 CircularQueue와 Coloring 관련 메서드를 구현하는 것이 이번 과제의 주요 내용입니다.

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

Class별 설명

- Interface Queue, Iterator / Enum vertexColor
- Class LinkedNode
- Class LinkedList
- Class CircularQueue
- Class Coloring
- Class AppController

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

Interface & Enum

<interface **Queue**>

```
public interface Queue<T> {
    public void reset();

    public int size();

    public boolean isEmpty();

    public boolean isFull();

    public boolean add(T anElement);

    public T remove();
}
```

interface Queue

interface Queue는 CircularQueue 구현 시 상속하여 사용합니다.

기본적인 Queue의 연산들이 정의되어 있습니다

<interface **Iterator**>

```
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
}
```

interface Iterator

interface Iterator는 이후 LinkedList에서 상속하여 사용되게 됩니다.

<enum **VertexColor**>

```
public enum VertexColor {
    NONE, RED, BLUE
}
```

enum VertexColor

VertexColor는 Coloring에서 vertex의 색깔을 나타내기 위해 선언되어 있는 enum Class 입니다.

LinkedNode

```java
public class LinkedNode<T> {
    private T _element;
    private LinkedNode<T> _next;

    public T element() {
        return this._element;
    }

    public void setElement(T newElement) {
        this._element = newElement;
    }

    public LinkedNode<T> next() {
        return this._next;
    }

    public void setNext(LinkedNode<T> newNext) {
        this._next = newNext;
    }

    public LinkedNode() {
        this.setElement(null);
        this.setNext(null);
    }

    public LinkedNode(T givenElement, LinkedNode<T> givenNext) {
        this.setElement(givenElement);
        this.setNext(givenNext);
    }
}
```

LinkedNode

LinkedNode는 노드의 Element와 다음 Node를 가리키는 멤버변수가 선언되어 있습니다.

next 메서드를 통해 현재 Node의 다음 Node를 반환합니다.

LinkedList (1)

```
public class LinkedList<T> {
    private LinkedNode<T> _head;
    private int _size;

    private LinkedNode<T> head() {
        return this._head;
    }

    private void setHead(LinkedNode<T> newHead) {
        this._head = newHead;
    }

    public int size() {
        return this._size;
    }

    private void setSize(int newSize) {
        this._size = newSize;
    }

    public LinkedList() {
        this.setSize(0);
        this.setHead(null);
    }

    public boolean isEmpty() {
        return (this.size() == 0);
    }

    public boolean isFull() {
        return false;
    }

    public boolean add(T anElement) {
        LinkedNode<T> newHeadNode = new LinkedNode<T>(anElement, this.head());
        this.setHead(newHeadNode);
        this.setSize(this.size() + 1);
        return false;
    }

    public T removeAny() {
        if (this.isEmpty()) {
            return null;

        } else {
            T removedElement = this.head().element();
            this.setHead(this.head().next());
            this.setSize(this.size() - 1);
            return removedElement;
        }
    }
```

LinkedList 의 멤버변수와 getter/setter

LinkedList 에는 LinkedList의 head node를 의미하는 _head와 현재 LinkedList의 크기를 의미하는 _size가 선언되어 있습니다.

LinkedList는 Node들이 Link 되어 있는 구조로 새로운 노드를 연결하면 기존 LinkedList의 head에 연결됩니다. 따라서 add 메서드를 살펴보면 새로들어온 anElement를 현재의 head를 next Node로 가지도록 설정한 후, LinkedList의 head로 지정하고 있는 것을 알 수 있습니다.

Remove 시에도, 현재의 head의 next node를 새로운 head로 설정하는 방식으로 Node를 제거하게 됩니다. 또한 변수값을 다루기 위한 getter/setter가 선언되어 있습니다.

LinkedList (2)

```
    public IteratorForLinkedList iterator() {
        return new IteratorForLinkedList();
    }

    public class IteratorForLinkedList implements Iterator<T> {
        LinkedNode<T> _nextNode;

        private LinkedNode<T> nextNode() {
            return this._nextNode;
        }

        private void setNextNode(LinkedNode<T> newLinkedNode) {
            this._nextNode = newLinkedNode;
        }

        private IteratorForLinkedList() {
            this.setNextNode(LinkedList.this.head());
        }

        @Override
        public boolean hasNext() {
            return (this.nextNode() != null);
        }

        @Override
        public T next() {
            T nextElement = this.nextNode().element();
            this.setNextNode(this.nextNode().next());
            return nextElement;

        }
    }
}
```

IteratorForLinkedList

LinkedList를 순회하기 위한 Iterator 입니다.

앞에서 구현하였던 interface Iterator를 상송받아 구현되어 있으며

next 메서드를 통해 다음 노드로 이동하는 방식입니다.

CircularQueue

```java
public class CircularQueue<T> implements Queue<T> {
    private T[] _elements;
    private int _capacity;
    private int _front;
    private int _rear;

    private T[] elements() {
        return this._elements;

    }

    private void setElements(T[] newElements) {
        this._elements = newElements;
    }

    private int capacity() {
        return this._capacity;

    }

    private void setCapacity(int newCapacity) {
        this._capacity = newCapacity;
    }

    private int front() {
        return this._front;
    }

    private void setFront(int newFront) {
        this._front = newFront;
    }

    private int rear() {
        return this._rear;
    }

    private void setRear(int newRear) {
        this._rear = newRear;

    }

    private int nextRear() {
        return (this.rear() + 1) % this.capacity();
    }

    private int nextFront() {
        return (this.front() + 1) % this.capacity();
    }
```

```java
    @SuppressWarnings("unchecked")
    public CircularQueue(int maxNumberOfElement) {
        this.setCapacity(maxNumberOfElement);
        this.setElements((T[]) new Object[this.capacity()]);
        this.reset();
    }

    @Override
    public void reset() {
        this.setFront(0);
        this.setRear(0);
    }

    @Override
    public int size() {
        if (this.front() <= this.rear()) {
            return this.rear() - this.front();
        } else {
            return (this.capacity() + this.rear() + this.front());
        }
    }

    @Override
    public boolean isEmpty() {
        return (this.front() == this.rear());
    }

    @Override
    public boolean isFull() {
        return (this.front() == this.nextRear());
    }

    public boolean add(T anElement) {
        if (this.isFull()) {
            return false;
        } else {
            this.setRear(this.nextRear());
            this.elements()[this.rear()] = anElement;
            return true;
        }
    }

    public T remove() {
        if (this.isEmpty()) {
            return null;
        } else {
            this.setFront(this.nextFront());
            return this.elements()[this.front()];
        }
    }
}
```

CircularQueue

CirularQueue는 interface Queue를 상속받아서 Queue의 메서드를 Override 하여 구현합니다.

Queue의 element를 담을 배열 elements가 선언되어 있고, Queue의 front와 rear를 표현하기 위한 int형 변수가 선언되어 있습니다.

BFS를 구현하기 위한 자료구조로 Queue에서 하나씩 꺼내어 해당 노드의 인접 노드들 중

방문한 적이 없었던 노드에 방문하고 방문한 노드를 다시 Queue에 넣어 이후 Queue에서

꺼내어질 때 해당 노드의 인접노드들을 방문하게 됩니다.

Coloring(1)

```
public class Coloring {
    private AdjacencyMatrixGraph _graph;
    private VertexColor[] _vertexColors;
    private int _startingVertex;
    private LinkedList<Edge> _sameColorEdges;

    private AdjacencyMatrixGraph graph() {
        return this._graph;
    }

    private void setGraph(AdjacencyMatrixGraph newGraph) {
        this._graph = newGraph;
    }

    private int startingVertex() {
        return this._startingVertex;
    }

    private void setStartingVertex(int newVertex) {
        this._startingVertex = newVertex;
    }

    private VertexColor[] vertexColors() {
        return this._vertexColors;
    }

    private void setVertexColors(VertexColor[] newVertexColors) {
        this._vertexColors = newVertexColors;
    }

    public VertexColor vertexColor(int aVertex) {
        return this.vertexColors()[aVertex];
    }

    private void setVertexColor(int aVertex, VertexColor newColor) {
        this.vertexColors()[aVertex] = newColor;
    }

    public LinkedList<Edge> sameColorEdges() {
        return this._sameColorEdges;
    }

    private void setSameColorEdges(LinkedList<Edge> newLinkedList) {
        this._sameColorEdges = newLinkedList;
    }
```

Coloring의 멤버변수

Coloring을 AdjacencyMatrixGraph에 대해 수행하므로 _graph 가 선언되어 있고,
각 Vertex의 color를 저장하기 위한 VertexColor 배열로 _vertexColors 가 선언되어 있습니다.
_sameColorEdges는 edge의 vertex가 같은 Edge의 정보를 담기위한 LinkedList 입니다.

getter/setter

외부에서 사용되지 않는 메서드를 private으로 선언하여 getter로 접근하도록하여 외부에서 값에 직접 접근하는 것을 제한하고, 값을 설정하기 위한 setter가 선언되어 있습니다.

Coloring(2)

```
public Coloring(AdjacencyMatrixGraph givenGraph) {
    this.setGraph(givenGraph);
    this.setVertexColors(new VertexColor[this.graph().numberOfVertices()]);
    for (int vertex = 0; vertex < this.graph().numberOfVertices(); vertex++) {
        this.setVertexColor(vertex, VertexColor.NONE);
    }
    this.setSameColorEdges(new LinkedList<Edge>());
    this.setStartingVertex(0);
}

public void runColoring() {
    this.paintColorsOfVertices();
    this.findSameColorEdges();
}

public void paintColorsOfVertices() {
    this.setVertexColor(this.startingVertex(), VertexColor.RED);
    CircularQueue<Integer> breadthFirstSearchQueue = new CircularQueue<Integer>(this.graph().numberOfVertices());
    breadthFirstSearchQueue.add(this.startingVertex());
    while (!breadthFirstSearchQueue.isEmpty()) {
        int tailVertex = breadthFirstSearchQueue.remove();
        VertexColor headVertexColor = (this.vertexColor(tailVertex) == VertexColor.RED) ? VertexColor.BLUE
                : VertexColor.RED;
        for (int headVertex = 0; headVertex < this.graph().numberOfVertices(); headVertex++) {
            Edge visitingEdge = new Edge(tailVertex, headVertex);
            if (this.graph().edgeDoesExist(visitingEdge)) {
                if (this.vertexColor(headVertex) == VertexColor.NONE) {
                    this.setVertexColor(headVertex, headVertexColor);
                    breadthFirstSearchQueue.add(headVertex);
                }
            }
        }
    }
}

private void findSameColorEdges() {
    for (int tailVertex = 0; tailVertex < this.graph().numberOfVertices(); tailVertex++) {
        for (int headVertex = 0; headVertex < this.graph().numberOfVertices(); headVertex++) {
            Edge visitingEdge = new Edge(tailVertex, headVertex);
            if (this.graph().edgeDoesExist(visitingEdge)) {
                if (this.vertexColor(tailVertex) == this.vertexColor(headVertex)) {
                    this.sameColorEdges().add(visitingEdge);
                }
            }
        }
    }
}

}
```

Coloring 생성자

Coloring 생성자는 coloring 할 graph를 받아 해당 graph의 vertex 수 만큼 VertexColors의 크기를 지정해 주고, 초기에는 모든 Vertex의 None으로 초기화 합니다.

void findSameColorEdge(){...}

해당 메서드는 Edge 중에서 tailVertex와 headVertex가 같은 색을 가진 Edge를 찾기 위해 구현된 메서드로, 그래프에 존재하는 Edge 중에 tailVertex와 headVertex의 vertexColor가 같으면 sameColorEdge에 해당 Edge를 추가합니다.

void paintColorsOfVertices(){...}

paintColorsOfVertices 메서드는 graph의 vertex에 색을 칠하는 행위를 하는 메서드로, Breadth First Search를 이용하여 그래프를 순회하며 색을 칠하게 됩니다. BFS를 위해 startingVertex를 Queue에 넣은 후 하나 Queue가 empty가 될 때까지 element를 꺼내면서 해당 vertex를 tailVertex로 지정하고, 만약 해당 tailVertex의 색이 RED 라면 headVertex는 Blue를 부여한 후, edgeDoesExist를 for문으로 순회하며
headVertex를 찾은 후 vertexColor가 NONE이 아니라면 setter를 이용해 vertexColor를 앞의 조건문에서 지정한 Color로 설정해주며, 그 vertex를 Queue에 추가하는 식으로 구현되어 있으며, BFS는 이런식으로 Queue가 빌때까지 순회하면서 Vertex에 색을 설정하여 모든 Vertex가 색칠되게 됩니다.

AppController 추가된 부분

```
public class AppController {
    private AdjacencyMatrixGraph _graph;
    private Coloring _coloring;

    public AppController() {
        this.setGraph(null);
        this.setColoring(null);
    }

    private Coloring coloring() {
        return this._coloring;
    }

    private void setColoring(Coloring aColoring) {
        this._coloring = aColoring;
    }
```

```
private void showColoring() {
    AppView.outputLine("");
    AppView.outputLine("> 각 vertex에 칠해진 색깔은 다음과 같습니다: ");
    for (int vertex = 0; vertex < this.graph().numberOfVertices(); vertex++) {
        AppView.outputLine("[" + vertex + "]" + this.coloring().vertexColor(vertex).name());
    }
    AppView.outputLine("");
    AppView.outputLine("> 양 끝 vertex의 색깔이 같은 edge들은 다음과 같습니다: ");
    if (this.coloring().sameColorEdges().size() == 0) {
        AppView.outputLine("!! 모든 edge의 양 끝 vertex의 색깔이 다릅니다.");
    } else {
        LinkedList<Edge>.IteratorForLinkedList iterator = this.coloring().sameColorEdges().iterator();
        while (iterator.hasNext()) {
            Edge currentEdge = iterator.next();
            AppView.output("(" + currentEdge.tailVertex() + "," + currentEdge.headVertex() + "):");
            AppView.outputLine(" " + this.coloring().vertexColor(currentEdge.tailVertex()).name());
        }
    }
}
```

```
public void run() {

    AppView.outputLine("<<< 입력되는 그래프의 사이클 검사를 시작합니다 >>>");
    this.inputAndMakeGraph();
    this.showGraph();

    this.setColoring(new Coloring(this.graph()));
    this.coloring().runColoring();
    this.showColoring();
    AppView.outputLine("");
    AppView.outputLine("<<< 그래프의 입력과 사이클 검사를 종료합니다 >>>");

}
```

AppController 추가된 부분

앞서 구현한 Coloring을 AppController에서 선언하고 showColoring을 구현하여 vertex들에 적용된 color를 사용자에게 보여주고, sameColorEdges 메서드를 이용하여 양 끝 vertex의 color가 같은 edge를 리턴해주게 됩니다.

run 메소드에서 만든 그래프를 Coloring 해주는 부분을 추가하고, 해당 부분을 showColoring으로 보여주는 부분이 추가되었습니다

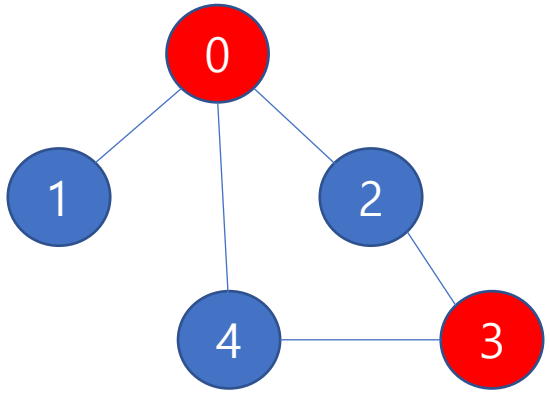
결과분석

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

결과 1

```
Console
<terminated> _Main_AL04_201701975_구건모 [Java Application] C:₩Program Files₩Java₩jdl
<<< 입력되는 그래프의 사이클 검사를 시작합니다 >>>
> 입력할 그래프의 vertex 수와 edge 수를 먼저 입력해야 합니다:
? Vertex 수를 입력하시오: 5
? Edge 수를 입력하시오: 5

> 이제부터 edge를 주어진 수 만큼 입력합니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 4
!새로운 edge (0,4) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 1
!새로운 edge (0,1) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 2
!새로운 edge (0,2) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 3
!새로운 edge (2,3) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 4
? head vertex 수를 입력하시오: 3
!새로운 edge (4,3) 가 그래프에 삽입되었습니다.

> 입력된 그래프는 다음과 같습니다:
[0] -> 1 2 4
[1] -> 0
[2] -> 0 3
[3] -> 2 4
[4] -> 0 3

> 각 vertex에 칠해진 색깔은 다음과 같습니다:
[0]RED
[1]BLUE
[2]BLUE
[3]RED
[4]BLUE

> 양 끝 vertex의 색깔이 같은 edge들은 다음과 같습니다:
!! 모든 edge의 양 끝 vertex의 색깔이 다릅니다.
```

결과1 그래프

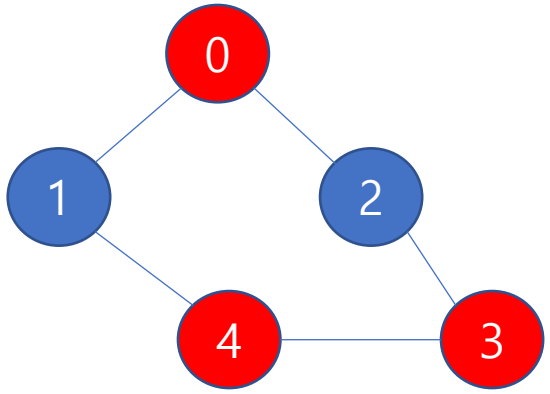


정상적인 동작을 테스트하기 위해 실습자료에서 나온 출력 예시와 동일한 상황으로 테스트 해보았습니다. root 노드인 0번 노드부터 시작하여 BFS를 통해 Coloring이 진행되는데 최초로 인접했던 노드들인 1,2번 노드는 0이 RED이기 때문에 BLUE로 지정되고 이후에 1번 노드는 인접노드가 0뿐이므로 Queue에 add 되지 않고 2번 노드를 방문한 후 3번 노드가 Queue에 add 된 상태이므로 다시 3번 노드를 Queue에서 꺼내어 인접노드 4번을 BLUE로 색칠한 모습입니다. 4번노드 또한 인접노드가 모두 방문처리 된 상태이기 때문에 Queue에 add 되지 않고 Queue가 비었기 때문에 BFS가 종료됩니다.

결과 2

```
Console
<terminated> _Main_AL04_201701975_구건모 [Java Application] C:₩Program Files₩Java₩jd
> 이제부터 edge를 주어진 수 만큼 입력합니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 1
!새로운 edge (0,1) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 4
!새로운 edge (1,4) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 4
? head vertex 수를 입력하시오: 3
!새로운 edge (4,3) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 3
? head vertex 수를 입력하시오: 2
!새로운 edge (3,2) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 0
!새로운 edge (2,0) 가 그래프에 삽입되었습니다.

> 입력된 그래프는 다음과 같습니다:
[0] -> 1 2
[1] -> 0 4
[2] -> 0 3
[3] -> 2 4
[4] -> 1 3

> 각 vertex에 칠해진 색깔은 다음과 같습니다:
[0]RED
[1]BLUE
[2]BLUE
[3]RED
[4]RED

> 양 끝 vertex의 색깔이 같은 edge들은 다음과 같습니다:
(4,3): RED
(3,4): RED

<<< 그래프의 입력과 사이클 검사를 종료합니다 >>>
```

결과2 그래프



두번째 결과화면의 경우 첫번째 실행결과와 동일한 방법으로 BFS를 통해 Coloring을 된 모습을 확인하실 수 있으며 두번째 결과의 경우 3번 verte와 4번 verte의 색깔이 동일하므로 화면에 출력되고 있음을 확인할 수 있습니다. findSameColorEdge가 tail과 head vertex가 동일한 color를 가진 Edge를 sameColorEdge에 add 한 후 해당 정보가 AppController의 showColoring 메서드를 통해서 LinkedList Iterator로 sameColorEdge를 정삭적으로 출력해 주고 있습니다.

과제 질문에 대한 답변

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

과제 안에 질문에 대한 답변- 생각할 점

❑생각할 점

- Breadth-first traversal 방법

- Iterator 는 왜 필요한가?
  - 구현 방법은?

*** Breadth First Traversal의 방법:**

breadth first traversal은 탐색할 때 너비를 우선적으로 탐색하는 방식으로 루트 노드부터 인접 노드들을 방문하고 방문처리를 합니다. 이후 방문한 노드들을 Queue에 넣어두었다가 꺼내어 꺼낸 노드의 인접 노드들 중 방문하지 않는 노드를 방문하고 방문처리 후 Queue에 넣는 과정을 반복하게 되는데 더 이상 Queue에서 꺼낼 원소가 존재하지 않을 경우 BFS가 끝나게 됩니다.

*** Iterator의 필요성과 구현방법**

Iterator는 모든 컬렉션 프레임워크(list,set,map,queue 등)에서 공통으로 사용가능하고 단순한 메서드 구성으로 컬렉션의 값들을 다룰 수 있다는 장점이 있습니다. 어떤 컬렉션이던 공통으로 쉽게 구현 및 사용할 수 있다는 점에서 필요성이 부각된다고 볼 수 있습니다. Iterator는 크게 다음 요소가 존재하는지 반환하는 hasNext와 다음 요소에 접근하는 Next 메서드를 구현하는 것으로 만들어집니다. 이번 과제에서는 iterator interface를 선언하고 LinkedList를 위한 iterator를 만들기 위해 interface Iterator를 상속받은 후 메서드들을 override 하여 구현하였습니다.

과제를 통해 느낀점

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

과제를 통해 느낀점

이번 과제에서는 이론시간에 배웠던 Breadth First Search를 실제로 어떻게 쓰는지에 대해서 체감하게 된 기회였습니다.

또한 BFS를 통한 그래프 색칠하기 기능을 구현하는 과정에서 다루었던 자료구조 LinkedList, CircularQueue 등을 다뤄보면서,

이전에 이론으로만 배웠던 자료구조가 실질적으로 어떻게 쓰이는 것인지도 알게 되었습니다.

과제의 코드양도 이전에 비해 많아진 것 같지만 코드양이 많아진만큼 실습과 과제 수행과정에서 구현하는 부분들을 더 자세히

살펴보게 되는 것 같고, 긴 코드이지만 전체적으로 이해하고 구현해보면서 더 몸에 와닿는 개념이 되어가는 것 같습니다.

감사합니다!

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

감사합니다

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com