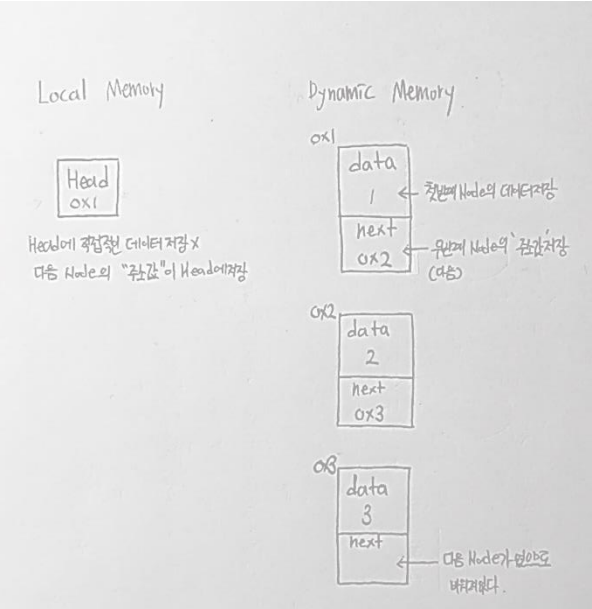
21.03.31

Node를 이용한 LinkedList 기본 구조:

```
//기존에 입력 된 데이터(Node) 중 마지막 Node 의 위치(주소)를 알아야 한다.
//Head(처음 Node)부터 기존의 마지막 Node 까지 순차적으로 위치(주소)를 확인하여 마지막 Node 에 새로운 Node 의 주소를 입력해야 한다.
//따라서 처음 Node 의 위치를 알아야 한다.

//맨 마지막 node 와 그 외의 node 간 차이점 : next 가 비어 있으면(다음 node 의 주소가 없으면) 마지막 node
```



1. Node 클래스 : Node의 기본 구조를 생성

방법1)

```
package _3 강;
public class Node {
	public Node next;//주소를 지칭하는 변수 Nodes 지정(연결고리)
	public int data;//실제 입력 받은 Block 의 데이터(데이터)
	public Node(int d)
	{
		next = null;
		data = d;
	}
}
/* 실제 함수에서 Node 를 변수에 인스턴스로 호출하면 자동으로 생성자로서 각 값이 지정된다.
 */ Node newBlock = new Node(data);// 인스턴스 선언 시, 자동으로 data 값이 Node class 내에 생성
```

방법2)

```
public class Node {
	public Node next;//주소를 지칭하는 변수 next 지정(연결고리)
	public int data;//실제 입력 받은 Block 의 데이터(데이터)
}
/* 방법 2 의 경우엔 실제 함수에서 Node 를 선언할 때마다 data 를 지정해주어야 한다.
 */newBlock.data = data; //newBlock 은 앞에서 이미 Node 로 호출된 인스턴스
```

2. LinkedList 클래스

```
package _4강;
import _3강.Node;//node를 연결하는 행동을 관리하는 클래스입니다.
public class LinkedList {

	public Node head = null; //head의 기본 값을 비워둔다.
	public void insert(int data)//새로운 Node 삽입 함수
	//접근자 public: 누구나 접근가능하도록 , 데이터형태 void: void 리턴 값이 따로 존재 x
	{
		Node newBlock = new Node(data);
		//local에 newBlock 데이터를 만들고, dynamic에  Node인 data와 next를 생성
		newBlock.data = data;
		//newBlock 객체의 data에 insert함수로 입력받은 data 저장
//원리: newBlock의 data에 데이터를 미리 저장
		if(head == null) //head에 다음 Node주소가 없는 경우 : 첫번째 Node인경우
		{
			head=newBlock; //새로운 node를 head로 삼는다는 의미
			return; //if문 탈출
		}
		else
		{

//원리: 첫 Node입력인 경우 head에 첫 Node가 될 NewBlock의 주소를 입력
			Node current = head;
			//Node current를 선언, current의 주소를 head로 지정
			while(current.next != null)
			//current의 next가 0이 아닌 경우 반복
			{
				current = current.next;
				//current에 next주소값을 담는다.
			}
		current.next = newBlock;//while종료 시 (현 node의 마지막) 새 node 추가
		}
	}
//원리: 첫 Node가 아닌경우, current 매개변수에 current.next를 순차적으로 대입하여 null값이 되는
current.next까지 탐색하고, current.next ==null (다음 Node의 주소가 없을 때) newBlock 주소 대입


	public void insertHead(int data) //head의 값을 바꾸는 함수
	{
		Node newBlock = new Node(data);
		newBlock.data = data;

		newBlock.next = head;
		head=newBlock;

	}


	public void print()// LinkedList 내의 모든 Node를 출력
			//모두 출력하므로 (): 인자 x, void: 리턴값 x, public:외부 공개 가능
	{
		Node current = head;
		while(current != null)
		{
			System.out.println(current.data);
			//System.out.println(current.next); →노드의 주소를 출력할 경우
			current = current.next;
		}
	}
```

```
	public int getData(int idx) //특정 index의 Node 출력
	{				//특정번째의 Node를 반환하므로 반환데이터 형태 int 사용
		Node current = head;
		for(int i = 0; i<idx; i++) //찾을 번째 수인 idx 전까지의 i 탐색
		{
			current = current.next;
		}
		return current.data; //current(특정번째 Node)의 data 반환

	}

	public void insertAt(int idx, int data) //특정 index에 Node를 삽입
	{				//삽입할 번째 수(index) : idx , 삽입 데이터 : data
		Node newBlock = new Node(data);
		newBlock.data = data;

		Node current = head;
		for(int i = 0; i<idx; i++)
		{
			current = current.next;

			// 1번째 방법 : 새로운 변수에 임시 저장
			Node temp = current.next;
			current.next = newBlock;
			newBlock.next = temp;
			/* 2번째 방법 : 연산의 순서를 바꾸기
			 * newBlock.next = current.next;
			 * current.next = newBlock;
			 */
		}
	}
/* 원리 :
 * idx 전까지 Head 에서부터 Node를 탐색하다가, idx-1 번째에서 기존(idx-1 번째)Node의 next와 새로 입력한 Node의 next를 바꾼다.
 */(이때 temp라는 임시 저장 변수를 이용하거나, 순서를 활용하여 기존 Node와 newBlock의 next(저장된 주소값)을 바꾼다.

	public void deleteLast() //마지막 번째 Node 삭제하는 함수
	{

		if(head == null)//Node 값이 아예 없는 경우(기존 노드 index=0)
			return;
//원리 : 애초에 Head에 저장된 Node 주소값이 없으므로 아무 작업하지 않고 if문 탈출

		else if(head.next ==null)//Node 값이 하나만 있던 경우(기존 노드 index=1)
			head =null;
/* 원리 :
 * head에 첫번째 Node의 주소값이 저장되어 있고 첫번째 Node의 data만 있던 형태
 */따라서, 첫번째 Node의 주소값이 저장되어 있는 head의 값 초기화
		else //그 외의 모든 경우
		{

			Node current = head;
			while(current.next.next !=null)
			{ //뒤에서 두번째 블록의 특징 : 다음 블록의 next는 null이다.
				current = current.next; //current에 next를 순차적지칭
			}
			current.next= null; //뒤에서 두번째 블록의 next 초기화
		}
	}
```

```
/* 원리:
 * 마지막 Node 를 찾아서, 마지막에서 두번째 Node 의 next 를 삭제
 * 마지막에서 두번째에 위치한 블록의 특징 : 그 다음 블록(마지막 Node)의 next 는 null 이다.
 */장은 deleteLast 전에 선언된 Node 가 지워지지 않지만, 해당 연산 이후 주소값이 다른 Node 에
저장되지 못한 데이터는 garbage collector 가 지운다.

	public void deleteAt(int idx)//특정 index 의 Node 를 삭제하는 함수
	{
		if(head == null) //head 값이 null 인 경우(node 가 입력되지 않은 경우)
			return;

		else if(idx==0) //삭제할 index 가 0 번째 Node 인 경우
		{
			head=head.next;
		}
/* 원리:head 다음번째 node 인 경우 head 의 다음 주소를 가리키는 값(next)를 head 로 만든다.
 */이렇게 되면 기존 head 의 값이 다음 Node 의 주소로 대체되었으므로, 기존에 첫번째였던 Node 는
사라진다.
		else // 그외의 모든 경우
		{
			Node current = head;
			for(int i =0; i<idx; i++)
			{
				current = current.next;
			//idx 보다 1 개 적은 index 의 next 를 current 에 지정한 뒤 탈출
			}
		current.next = current.next.next; //해당 next 에 다음번째 next 저장
		}
	}
// 원리: 찾을 index 직전번째 Node 의 next 를 찾고 해당 next 에 다음 Node 의 next(index 번째 주소를
건너뛰고 다음 Node 의 주소)를 저장하면, index 번째 Node 의 주소를 가지고 있는 Node 가 없기때문에
해당 Node 를 삭제할 수 있다.

	public int getLastData() //마지막 Node 의 데이터 반환
	{
		Node current= head;
		while(current.next!=null)
		{
			current=current.next;
		}
		int data=current.data;
		return data;
	}

	public void clear() //연결리스트 내의 모든 노드 삭제
	{
		head=null;
	}

	public int length() //연결리스트의 Node 개수 반환
	{
		int num=0;
		Node current=head.next;
		while(current.next!=null)
		{
			num+=1;
			current= current.next;
		}
		return num;
	}
}
```

사용 시 주의 사항:

getData
getLastData

이 2가지 메소드를 활용한 함수들(Queue 나 Stack)은 반드시 새로운 변수로 2개 메소드의 값을 저장한 뒤에 print해야 결과 값이 나온다.

21.04.10

Node를 이용한 Queue 기본 구조:

1. Queue 클래스 생성:

```
package 선형자료구조;
//import 선형자료구조.LinkedList;  // -> 동일한 패키지 이므로 import 필요 x
public class Queue {
	private LinkedList buffer = null;//buffer 를 선언하고 초기화

	Queue() //생성자
	{
		buffer= new LinkedList(); //buffer 라는 연결리스트 생성
	}

	public void Enqueue(int data) //Queue 에 값을 추가하는 함수
	{
		//buffer 의 가장 마지막 Node 에 데이터를 추가
		buffer.insert(data);
	}

	public int Dequeue() //queue 에서 값을 빼는 함수
	{
		int data=buffer.getData(0);//buffer 의 0 번째 인덱스 Node 반환
		buffer.deleteAt(0);//buffer 의 0 번째 인덱스 Node 제거
		return data;
	}

	public int length() //queue 의 원소 수를 출력하는 함수
	{
		int num = buffer.length();
		return num;
	}

	public void reverseEnqueue(int data) //뒤에 데이터를 추가
	{
		buffer.insertHead(data); //buffer 의 head 다음 Node 에 데이터 추가
	}

	public int reverseDequeue() //뒤에 데이터를 제거
	{
		int lastNum= buffer.length();// buffer 의
		int data = buffer.getData(lastNum);
		buffer.deleteLast();
		return data;
	}


	public void print() //Queue 에 저장된 원소 전체를 출력하는 함수
	{
		buffer.print();
	}
```

```
	public void clear() //Queue 에 저장된 원소들을 모두 지우고 Queue 를 초기화하는 함수
	{
		buffer.clear();
	}
}
```

21.04.10

Node를 이용한 Stack 기본 구조:

1. Stack 클래스 생성(LinkedList 기반):

```
package 선형자료구조;
//import 선형자료구조.LinkedList;  // -> 동일한 패키지 이므로 import 필요 x
public class Stack {
	private LinkedList buffer = null;//buffer 를 선언하고 초기화

	public Stack()
	{
		buffer = new LinkedList();
	}

	public void push(int data) //데이터를 삽입(뒤로 삽입 시 앞으로 빼야 한다.)
	{
		buffer.insert(data); // 뒤에서부터 데이터를 삽입할 경우
	}

	public int pop() //해당 원소를 반환한 뒤 Node 에서 삭제
	{
		int data = buffer.getLastData(); //가장 뒤의 데이터를 임시 저장
		buffer.deleteLast();//가장 뒤의 데이터를 삭제
		return data;
	}

	public void pushReverse(int data)
	{
		buffer.insertHead(data); //앞에서부터 데이터를 삽입하는 경우
	}

	public int popReverse()
	{
		int data = buffer.getData(0); //가장 앞의 데이터를 임시 저장
		buffer.deleteAt(0);//가장 앞의 데이터를 삭제
		return data;
	}

	public void print() //Stack 의 전체 데이터 출력
	{
		buffer.print();
	}

	public void clear() //Stack 의 전체 데이터 초기화
	{
		buffer.clear();
	}

	public void lenght() //Stack 의 Node 개수 반환
	{
		buffer.length();
	}
```

```
	public int top() //가장 최근에 입력된 데이터 반환
	{
		int data= buffer.getLastData();
		return data;
	}
}
```

2. StackArray 클래스 생성(Array 기반):

```
package 선형자료구조;

public class StackArray {
	private String[] buffer;
	private int position = 0; // 각 데이터의 index 를 의미(주소의 위치)

	StackArray(int size) //Array 를 사용했으므로 배열의 원소 개수 설정이 필요
	{
		buffer = new String[size];
	}

	public void push(String data)
	{
		if(position >= buffer.length)
			return ;
	//배열로 구성했으므로, size 를 초과하는 개수의 원소를 입력받을 경우 데이터를 추가 불가능

		else
		{
		buffer[position]= data;
		position++; //다음에 추가될 데이터의 index 순서를 의미
		}
	}

	public String pop()
	{
		if (position== 0)
			return "empty";
		else
		{
		String data = buffer[position-1]; //position 은 다음 Node 의 index 번호 의미
		return data;
		}
	}
}
```

21.04.14

Node를 이용한 TreeNode 기본 구조:

1. TreeNode 클래스 : Tree의 기본 구조를 생성

```
package _7강;
//트리를 구성하는 기본 단위
public class TreeNode {

	int data;

	TreeNode left;
	TreeNode right;
	//만약 자식이 3개이상이면 더 노드를 추가하면 된다.
}
/*  원리: TreeNode 자체에 int형 data생성
 */ 왼쪽/오른쪽 Node를 지정(TreeNode left 와 TreeNode right로 선언) → 주소를 담을 변수
```

2. Tree 클래스 생성:

```
package _7강;
public class Tree {
	//TreeNode를 가지고 나무 구조를 구성하고 검색하는 이진 탐색 트리를 구현
	private TreeNode root = null; //루트값을 null(빈 값)으로 설정


	public boolean find(int query) //탐색
	{
		return findNode(root, query);
		//탐색 시에는 변수를 따로 고려할 필요가 없으므로 바로 findeNode를 실행
	}


	private boolean findNode(TreeNode current, int query) //검색(탐색하며 비교)
	{
		if(current == null)//쿼리가 트리 안에 없는 경우(값을 찾을 수 없는 경우)
			return false;

		if(current.data == query) //노드가 쿼리와 같은 경우
			return true;

		if(current.data >= query)//노드보다 쿼리보다 큰 경우
		{
			return findNode(current.left, query);
		}

		else //노드가 쿼리보다 작은  경우
			return findNode(current.right, query);
	}
/*  원리: find함수로 쿼리만 사용자에게 입력 받고, 재귀함수를 활용하여 트리 안에서 쿼리를 검색
 *  굳이 find와 findNode로 분리한 이유 : findNode를 재귀로 풀어야 하는데
 *  이때 반드시 현재 노드의 값을 입력받아야 하므로
 *  find함수에는 findNode(root(초기값), query(사용자 입력값)) 을 return하고,
 *  findNode함수는 Node의 경우(query와 current.data의 비교)에 따라
 *  current.left와 current.right으로 나누어 재귀함수 진행 → LinkedList에서 next같은 기능

 *  노드가 쿼리와 같은 경우 : 1) root = query  2) TreeNode = query 인경우 존재
 *  이때 2) TreeNode는 root가 아닌 root의 siblingNode 중 하나이므로 경우를 나누어 재귀호출
 *  재귀를 반복하다가 current.data=query (Node=query)인 경우를 만나면 true 반환
 */  leafNode까지 갔는데, query와 맞는 값을 못 찾은 경우 false를 반환
```

```
        public void insert(int data) //삽입
         {
                  TreeNode newNode = new TreeNode(); //TreeNode 인스턴스 생성
                  newNode.data= data;

                  if(root ==null) //root 값이 비어있을 경우
                  {
                           root = newNode;
                           return;
                  }

                  else// root 의 값이 있는 경우 -> 재귀함수이용해서 검색한 뒤 leafNode 에 저장
                           insertNode(root, newNode);
         }


         private void insertNode(TreeNode current, TreeNode newNode) //검색(탐색하며 삽입)
         {
                  if(current.data >= newNode.data)
                  {
                           if(current.left == null)
                                    current.left = newNode;
                           else
                                    insertNode(current.left, newNode);
                  }
                  else
                  {
                           if(current.right == null)
                                    current.right = newNode;
                           else
                                    insertNode(current.right, newNode);
                  }
         }

}
/*  원리: insert 함수로 쿼리만 사용자에게 입력 받고,
 *  경우 1) root 가 비어있다면 data 를 root 에 저장하고
 *  경우 2) root 가 안비어있으면 재귀함수를 활용하여 트리 안에서 쿼리를 검색
 *
 *  굳이 insert 와 insertNode 로 분리한 이유 : insertNode 를 재귀로 풀어야 하는데
 *  이때 반드시 현재 노드의 값을 입력받아야 하므로
 *  insert 함수에는 insertNode(root(초기값), query(사용자 입력값)) 을 return 하고,
 *  insertNode 함수는 Node 의 경우(query 와 current.data 의 비교)에 따라
 *  current.left 와 current.right 으로 나누어 재귀함수 진행 → LinkedList 에서 next 같은 기능
 *
 *  data 와 TreeNode 의 값 비교 : 1) TreeNode>= data 인경우  2) TreeNode <= data 인경우 존재
 *  이때 1)은 leftNode 에 값 추가이므로, current.left(현재 Node 의 left 값이 있는 지 확인)
 *      1-1)비어있다면, current.left 에 newNode 추가(data 와 주소 값)
 *      1-2)값이 이미 있다면, insertNode 재귀호출하여 다음 Node 의 left 로 이동
 *  똑같이 2)는 rightNode 에 값 추가이므로, current.right(현재 Node 의 right 값이 있는 지 확인)
 *       2-1)비어 있는 경우, current.right 에 newNode 추가(data 와 주소 값)
 */       2-2)값이 이미 있는 경우, insertNode 재귀호출하여 다음 Node 의 right 로 이동
```

트리 순회 :

1)전위순회 : traverse_preorder

2)중위순회 : traverse_inorder

3)후위순회 : traverse_postorder

1)preorder 순회 방식:

```
public void traverse_preorder() //전위순회 결과값 반환
{
        traverse_preorder_node(root);
}// 기준이 current 인 subtree 를 preorder 방식으로 traverse 하는 일반적인 함수

public void traverse_preorder_node(TreeNode current) //전위순회 탐색
{
        if(current == null) //가장 기초적인 경우에 재귀함수를 멈추는 기능
                return;    //(재귀함수는 무한 루프에 빠질 위험 있음)
        System.out.println(current.data);
        if(current.left!= null)
                traverse_preorder_node(current.left);
        if(current.right!= null)
                traverse_preorder_node(current.right);
}
```

2)inorder 순회 방식:

```
public void traverse_inorder() //중위순회 결과값 반환
{
        traverse_inorder_node(root);
}

public void traverse_inorder_node(TreeNode current) //중위순회 탐색
{
        if(current == null) //가장 기초적인 경우에 재귀함수를 멈추는 기능
                return;
        if(current.left!= null)
                traverse_inorder_node(current.left);
        System.out.println(current.data);
        if(current.right!= null)
                traverse_inorder_node(current.right);
}
```

3) postorder 순회 방식:

```
public void traverse_postorder() //후위 순회 결과값 반환
{
        traverse_postorder_node(root);
}

public void traverse_postorder_node(TreeNode current) //후위순회 탐색
{
        if(current == null)
                return;
        if(current.left!=null)
                traverse_postorder_node(current.left);
        if(current.right!=null)
                traverse_postorder_node(current.right);
        System.out.println(current.data);
}
```