**07. 트리(Tree)**

**07-1. 트리의 이해**

**# 트리(Tree)**

* **트리의 정의**

: 원소들 간에 **1: 多** 관계를 가지는 **비선형 자료구조**

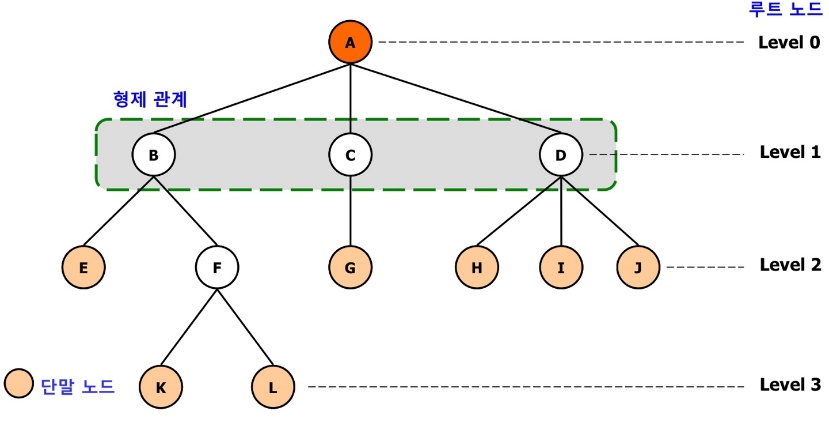
: 원소들 간에 **계층 관계**를 가지는 **계층형 자료구조**

: 상위 원소에서 하위 원소로 내려가면서 확장되는 **나무 모양의 구조**

ex) 컴퓨터 디렉터리 구조, 기업 구조, 족보, 결정 트리

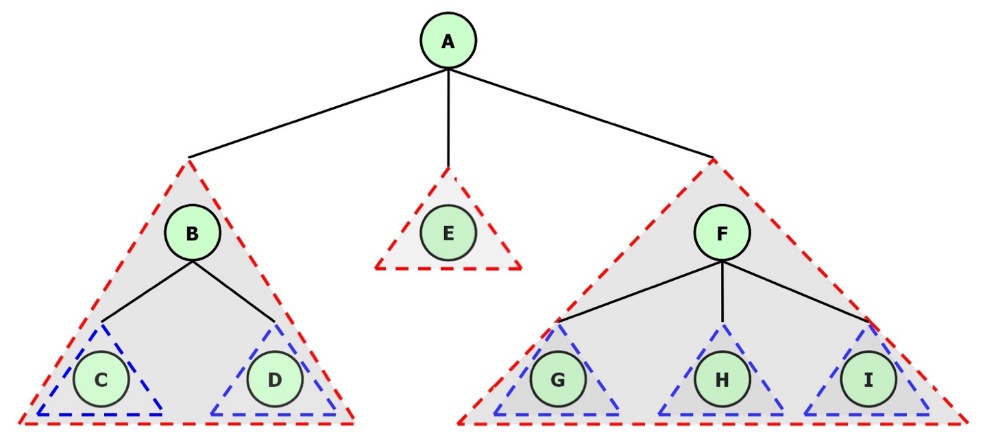
**# 트리 구조**

**- 부모, 자식 관계 : 노드(Node), 간선(Edge)**

****

**- 부분트리(Subtree)**

* + 자식 노드들은 각각 독립하여 새로운 트리를 구성할 수 있다.
  + 각 노드는 자식 노드 수만큼의 서브 트리를 갖는다.



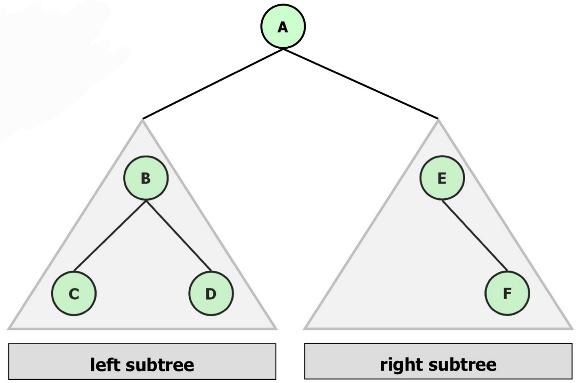
**07-2. 이진트리**

**# 이진 트리(Binary Tree)**

: 최대 두개 까지의 자식 노드를 가질 수 잇는 트리

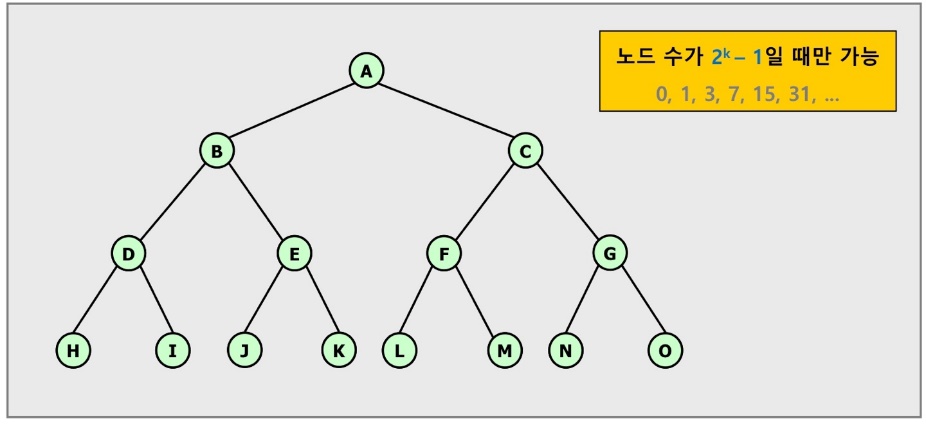
* 하나의 노드는 0, 1, 혹은 2개의 서브 트리를 가질 수 있다.

ex) 좌 서브 트리(left subtree), 우 서브 트리(right subtree), 널 트리(null tree)



**# 포화 이진 트리(Full Binary Tree)**

: 루트로부터 시작해서 모든 노드가 정확히 두 개씩의 자식 노드를 가지도록 꽉 채워진 트리

****

**# 완전 이진 트리(Complete Binary Tree)**

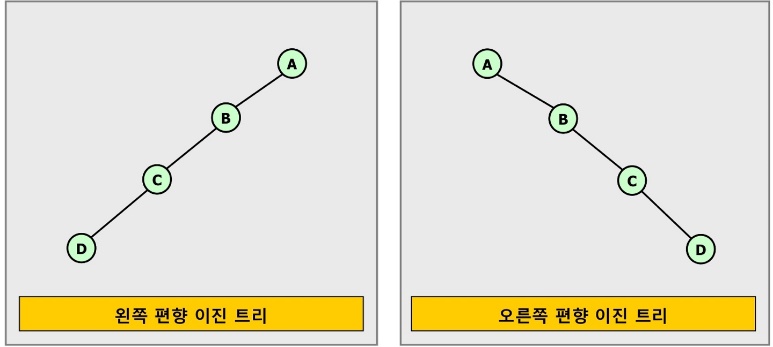
: 노드의 수가 맞지 않아 포화 이진 트리를 만들 수 없으면 맨 마지막 레벨은 왼쪽부터 채워 나간다.

도표, 라인, 디자인이(가) 표시된 사진

자동 생성된 설명

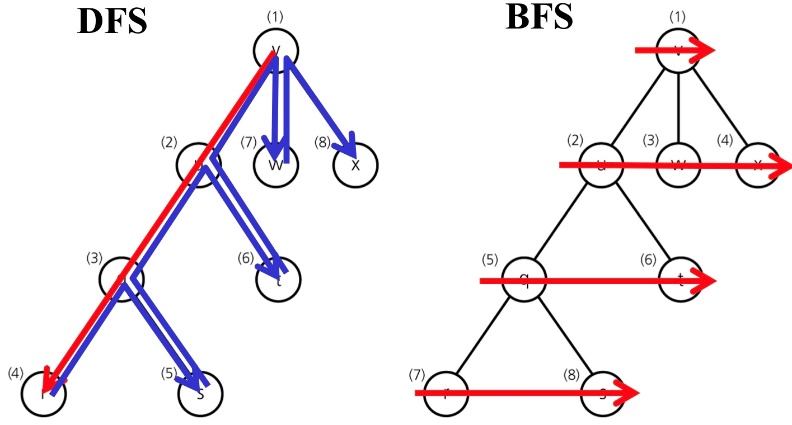
**# 편향 이진 트리(Skewed Binary Tree)**

: 이진 트리 중에서 최소 개수의 노드를 가지면서 왼쪽이나 오른쪽 서브트리만 가지고 있는 트리

****

**07-2-1. 이진트리 순회**

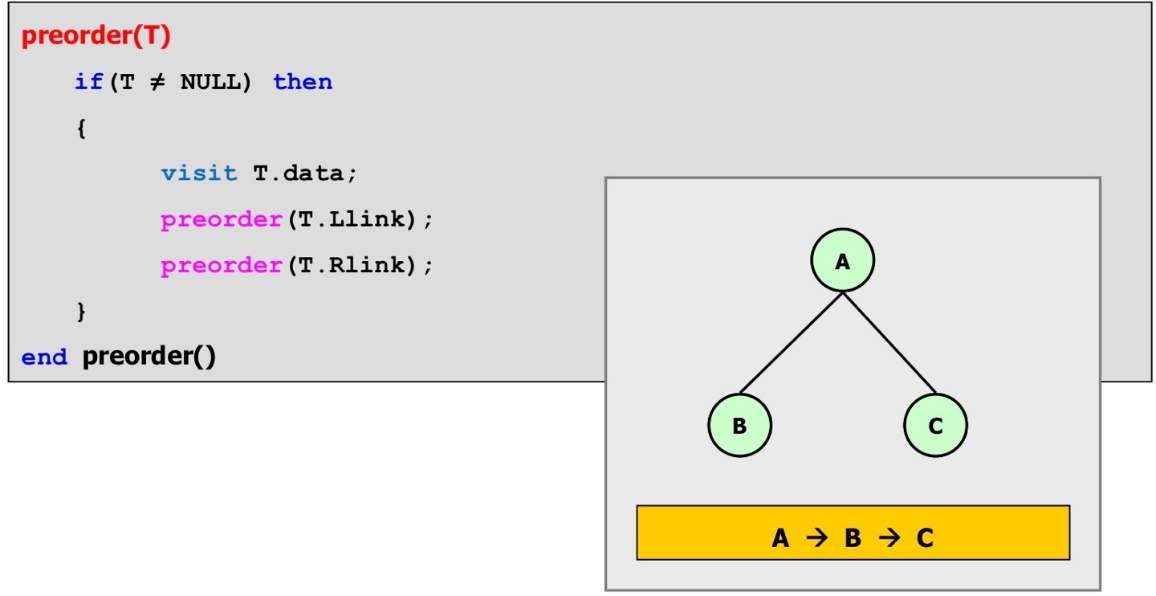
**# 이진 트리 순회(Traversal)**

****

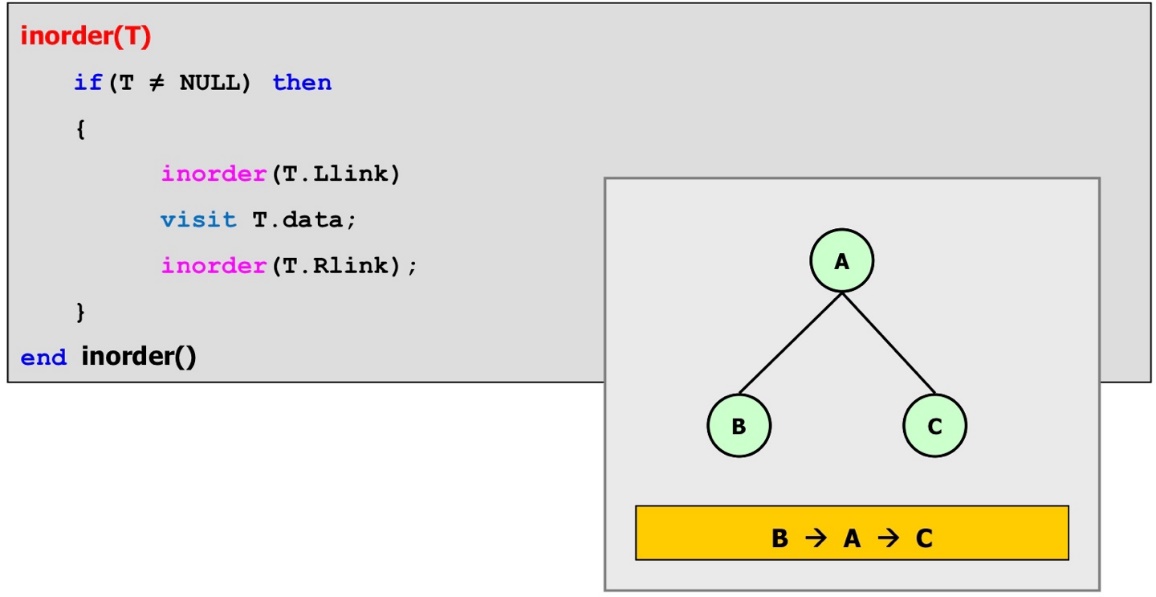
1. **깊이 우선 순회\_DFS**

**: 스택을 이용하여 구현**

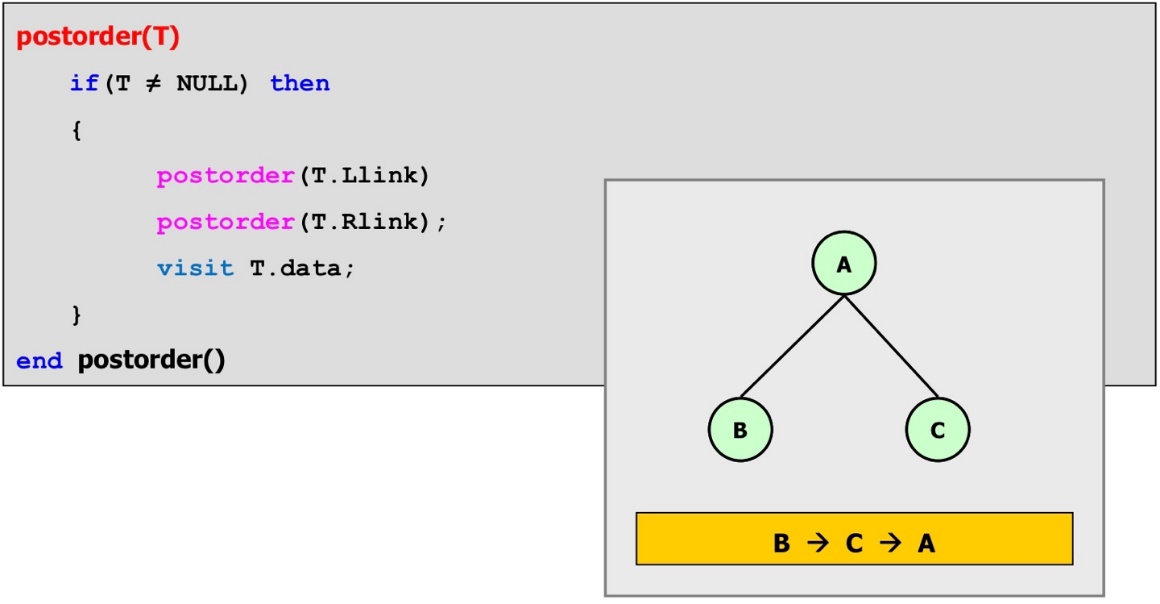
- 전위 순회(preorder traversal)



- 중위 순회(inorder traversal)



- 후위 순회(posterorder traversal)



**2. 너비 우선 순회\_BFS**

**: 큐를 이용하여 구현**

* 다음 레벨의 노드들을 처리하기 전에 노드의 자식 모두를 처리

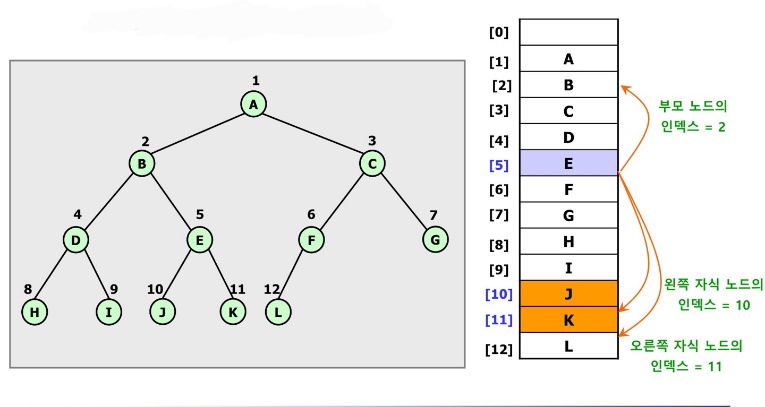
라인, 도표, 스크린샷, 그래프이(가) 표시된 사진

자동 생성된 설명

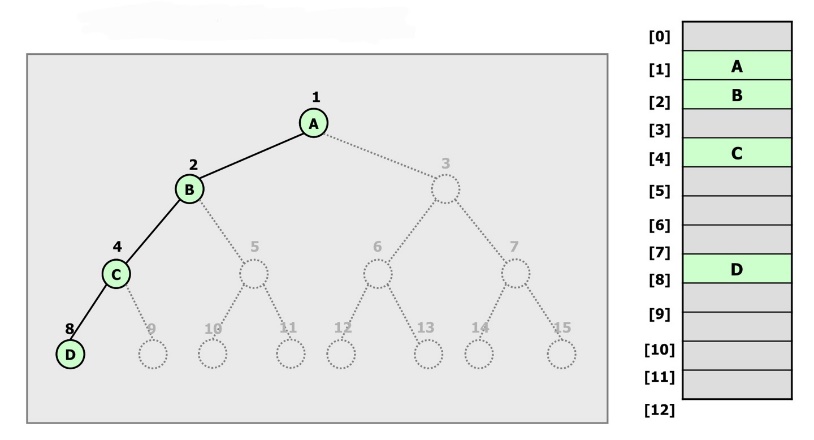
**07-2-2. 이진트리 구현**

**1. 순차 자료 구조**

**# 완전 이진 트리의 배열 표현**

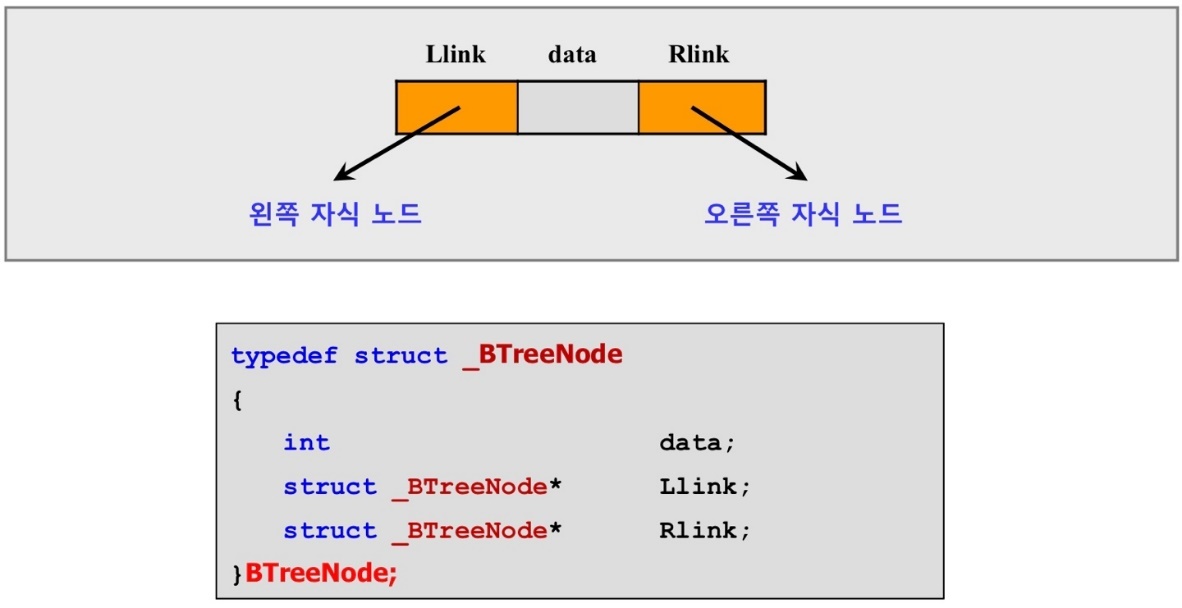
****

**# 편향 이진 트리의 배열 표현**

****

**2. 연결 자료구조**

**# 이진 트리 구현**

****

**# 완전 이진 트리의 연결 자료 구조 형태**

**도표, 라인, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명**

**# 편향 이진 트리의 연결 자료 구조 형태**

**스크린샷, 도표, 라인, 텍스트이(가) 표시된 사진

자동 생성된 설명**

**# 이진 트리 구현 : 연결 자료 구조**

**1. ver.C**

**2. ver.CPP**

**# LinkedBTree.cpp**

/\*

이진 트리: 알고리즘 구현 -- 단순연결리스트

파일명: LinkedBTree(demo).cpp

- main: 이진 트리(수식 트리) 구성 및 순회

\*/

#include <iostream>

#include <string>

#include "LinkedBtree.h" // DNode Class

// LinkedBTree Class

using namespace std;

int main(void)

{

string postfix;

cout << "20215229 이채훈" << endl;

cout << "트리를 구성할 후위 수식: ";

getline(cin, postfix);

// 이진 트리 생성 및 구성

LinkedBTree<char> BT;

DNode<char>\* root = BT.makeLinkedBTree(postfix);

// 깊이 우선 순회: 전위.중위.후위 순회

cout << "\nPreorder : "; BT.Preorder(root); // 전위 순회

cout << "\nInorder : "; BT.Inorder(root); // 중위 순회

cout << "\nPostorder : "; BT.Postorder(root); // 후위 순회

cout << "\nLevelorder : "; BT.Levelorder(root); // 너비 우선 순회

return 0;

}

**# LinkedBTree.h**

#include <iostream>

#include <string>

#include <stack>

#include <queue>

#include "LinkedQueue.h"

using namespace std;

// 연산자 여부를 판단

int isOperator(int op) {

return op == '+' || op == '-' || op == '\*' || op == '/';

}

template <typename E>

class LinkedBTree;

// class DNode

template <typename E>

class DNode {

public:

DNode(const E& data);

private:

E \_\_data;

DNode<E>\* \_\_Llink;

DNode<E>\* \_\_Rlink;

template <typename E> friend class LinkedBTree;

};

//DNode: 생성자와 메소드 정의

template <typename E>

DNode<E>::DNode(const E& data)

:\_\_data(data), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

// class LinkedBTree

template <typename E>

class LinkedBTree {

public:

LinkedBTree();

DNode<E>\* makeLinkedBTree(const string str);

void Preorder(DNode<E>\* root) const;

void Inorder(DNode<E>\* root) const;

void Postorder(DNode<E>\* root) const;

void Levelorder(DNode<E>\* root)const;

private:

DNode<E>\* root;

};

//LinkedBTree class : 생성자와 소멸자

template <typename E>

LinkedBTree<E>::LinkedBTree()

: root(nullptr){}

//이진 트리(수식 트리) 생성

template <typename E>

DNode<E>\* LinkedBTree<E>::makeLinkedBTree(const string str) {

stack<DNode<E>\*> s;

DNode<E>\* tNode;

for (int i = 0; i < str.length(); i++) {

//공백 제거

while (str[i] == ' ')

i++;

//새로운 노드를 생성

tNode = new DNode<E>(str[i]);

//연산자일 경우...

if (isOperator(str[i])) {

tNode->\_\_Rlink = s.top(); s.pop();

tNode->\_\_Llink = s.top(); s.pop();

}

s.push(tNode);

}

tNode = s.top();

s.pop();

//루트 노드 설정

root = tNode;

return tNode;

}

//깊이 우선 순회: 전위 순회(재귀)

template <typename E>

void LinkedBTree<E>::Preorder(DNode<E>\* root)const {

if (root) {

cout.width(3); cout << root->\_\_data;

Preorder(root->\_\_Llink);

Preorder(root->\_\_Rlink);

}

}

//깊이 우선 순회: 중위 순회(재귀)

template <typename E>

void LinkedBTree<E>::Inorder(DNode<E>\* root)const {

if (root) {

Inorder(root->\_\_Llink);

cout.width(3); cout << root->\_\_data;

Inorder(root->\_\_Rlink);

}

}

//깊이 우선 순회: 후위 순회(재귀)

template <typename E>

void LinkedBTree<E>::Postorder(DNode<E>\* root)const {

if (root) {

Postorder(root->\_\_Llink);

Postorder(root->\_\_Rlink);

cout.width(3); cout << root->\_\_data;

}

}

//너비 우선 순회(비재귀)

template <typename E>

void LinkedBTree<E>::Levelorder(DNode<E>\* root)const {

if (root == nullptr)

return;

queue<DNode<E>\*> Q;

Q.push(root);

while (!Q.empty()) {

DNode<E>\* tNode = Q.front();

Q.pop();

cout.width(3);

cout << tNode->\_\_data;

if (tNode->\_\_Llink) Q.push(tNode->\_\_Llink);

if (tNode->\_\_Rlink) Q.push(tNode->\_\_Rlink);

}

cout << endl;

}

**# LinkedQueue.h**

#pragma once

#include <iostream>

using namespace std;

inline void error(const char\* message) {

cout << message;

exit(100);

}

template<typename E>

class SNode {

public:

SNode(const E& data);

E getData(void)const;

private:

E \_\_data;

SNode<E>\* \_\_link;

template<typename E> friend class LinkedQueue;

};

//SNode class: 멤버 함수 외부 정의

template <typename E>

SNode<E>::SNode(const E& data)

: \_\_data(data), \_\_link(nullptr) {}

template <typename E>

E SNode<E>::getData(void) const {

return \_\_data;

}

//LinkedQueue class

template <typename E>

class LinkedQueue {

public:

LinkedQueue();

~LinkedQueue();

void enQueue(const E& e);//enQueue

void deQueue(void); //deQueue

E& front(void) const;

E& back(void) const;

bool empty(void) const;

int size(void)const;

void printQueue(void) const;

private:

SNode<E>\* \_\_front;

SNode<E>\* \_\_rear;

int \_\_count;

};

//LinkedStack class: 생성자(소멸자)와 소멸자

template<typename E>

LinkedQueue<E>::LinkedQueue()

: \_\_front(nullptr), \_\_rear(nullptr), \_\_count(0) {}

template <typename E>

LinkedQueue<E> :: ~LinkedQueue() {

SNode<E>\* temp = \_\_front;

while (temp) {

\_\_front = temp->\_\_link;

delete temp;

temp = \_\_front;

}

}

//push(enQueue) : 큐에 데이터 항목 삽입(맨 마지막 원소)

template <typename E>

void LinkedQueue<E>::enQueue(const E& e) {//enQueue

SNode<E>\* newSNode = new SNode<int>(e);

//newSNode-> \_\_data=e;

if (\_\_front)

\_\_rear->\_\_link = newSNode;

else \_\_front = newSNode;

\_\_rear = newSNode;

}

//pop(deQueue) : 큐에서 맨 첫번째로 삽입된 데이터 항목 삭제

template <typename E>

void LinkedQueue<E>::deQueue(void) {//deQueue

if (empty()) error("STACK IS EMPTY"); //throw "ERROR : STACK IS EMPTY";

SNode<int>\* temp = \_\_front;

//E data = temp->\_\_data;

\_\_front = temp->\_\_link;

if (!\_\_front)

\_\_rear = nullptr;

delete temp;

}

//front(peek) : 큐에서 맨 첫 번째로 삽입된 데이터 항목 확인

template<typename E>

E& LinkedQueue<E>::front(void)const {

if (empty())error("STACK IS EMPTY");//throw "ERROR::STACK IS EMPTY";

return \_\_front->\_\_data;

}

//back(peek) : 큐에서 맨 마지막에 삽입된 데이터 항목 확인

template<typename E>

E& LinkedQueue<E>::back(void)const {

if (empty())error("STACK IS EMPTY"); //throw "ERROR::STACK IS EMPTY";

return \_\_rear->\_\_data;

}

//empty : 스택의 공백 여부 확인

template <typename E>

bool LinkedQueue<E> ::empty(void)const {

return \_\_front == nullptr;

}

//size : 큐의 크기(항목의 개수)

template<typename E>

int LinkedQueue<E> ::size(void)const {

return \_\_count;

}

//printQueue : 큐의 전체 데이터 항목 출력

template <typename E>

void LinkedQueue<E>::printQueue(void)const {

SNode<E>\* temp = \_\_front;

cout << "\nQUEUE [";

while (temp) {

cout << temp->\_\_data;

temp = temp->\_\_link;

}

cout << "]" << endl;

}

**07-3. 우선 순위 큐와 힙**

**# 힙(Heap)**

: 우선 순위 큐를 구현하는 가장 기본적인 자료구조(다음 두 조건을 만족해야 한다.)

1. **완전 이진 트리**
2. **모든 노드는 값을 갖고, 자식 노드(들) 값보다 크거나 같다.**

**# 우선 순위 큐(Priority Queue)**

**: 가장 높은 우선순위를 가진 항목에 접근, 삭제와 임의의 우선순위를 가진 항목을 삽입을 지원하는 자료구조**

**\* 스택이나 큐도 일종의 우선 순위 큐**

스택 : 가장 마지막으로 삽입된 항목이 가장 높은 우선순위를 가진다.

* 따라서, 최근 시간일수록 높은 우선순위를 부여한다.

큐 : 먼저 삽입된 항목이 우선순위가 높다.

* 따라서, 이른 시간일수록 더 높은 우선순위를 부여한다.

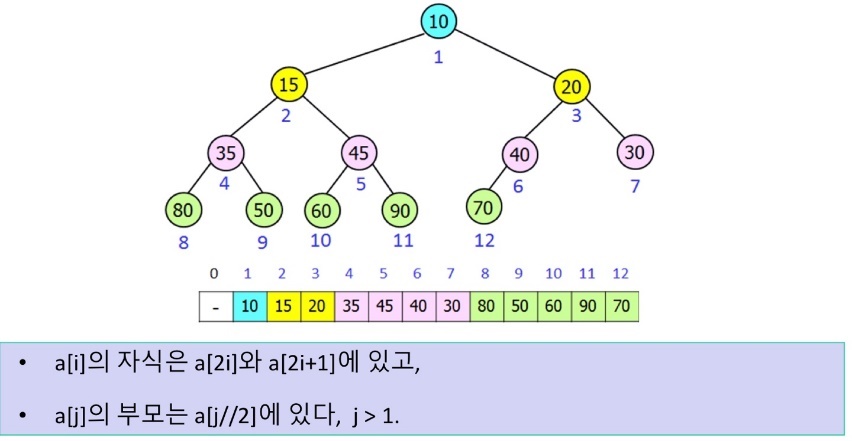
**but, 삽입되는 항목이 임의의 우선순위를 가지면 스택이나 큐는 새 항목이 삽입될 때마다 저장되어 있는 항목들을 우선순위에 따라 정렬해야 하는 문제점이 있음.**

**# 힙 : 완전 이진트리**

**: 완전 이진트리로서 부모의 우선 순위가 자식의 우선 순위보다 높은 자료구조**

**# 힙 : 순차 자료구조**

**: 완전 이진트리의 노드들이 저장된 리스트**



**# 최소 힙**

**: 키 값이 작을수록 높은 우선순위**

* 최소 힙의 루트에는 항상 가장 작은 키가 저장된다.
* 부모에 저장된 키가 자식의 키보다 작다는 규칙
* 루트는 a[1]에 있으므로, O(1)시간에 min 키를 가진 노드 접근

스케치, 원, 도표, 화이트이(가) 표시된 사진

자동 생성된 설명

**1. 삽입 연산**

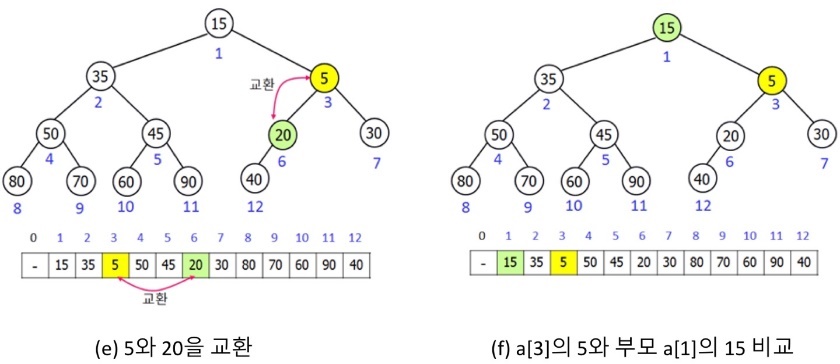
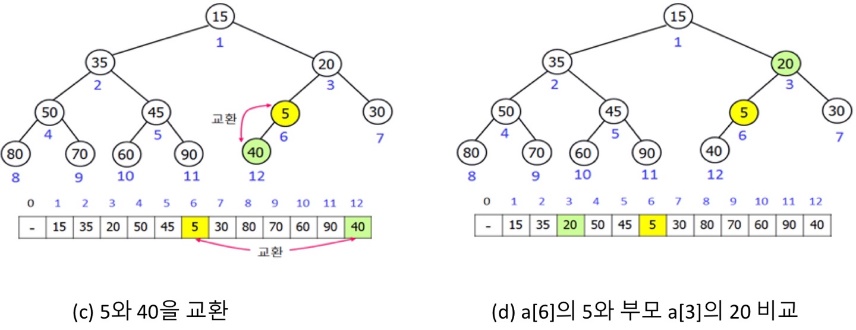
- 힙의 마지막 노드(즉, 리스트의 마지막 항목)의 바로 다음 비어 있는 원소에 새로운 항목을 저장

- 루트 방향으로 올라가면서 부모의 키와 비교하여 힙 속성이 만족될 때 까지 노드를 교환(위로 올라가면 수행되므로, **upheap**이라 부름.

ex) 최소 힙에 5를 삽입하는 과정

**텍스트, 도표이(가) 표시된 사진

자동 생성된 설명**

****

**텍스트, 도표, 라인, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

**2. 삭제 연산**

- 힙의 가장 마지막 노드. 즉, 리스트의 가장 마지막항목을 루트로 옮기고, 힙 크기를 1 감소시킨다.

- 루트로부터 자식들 중에서 작은 값을 가진 자식(승자)와 키를 비교하여 힙 속성이 만족될 때까지 키를 교환하며 이파리 방향으로 진행(루트로부터 아래로 내려가면 진행되므로, **downheap**이라 부름.

ex) 최소 힙에 루트의 키를 삭제시키는 과정

**텍스트, 도표, 라인, 원이(가) 표시된 사진

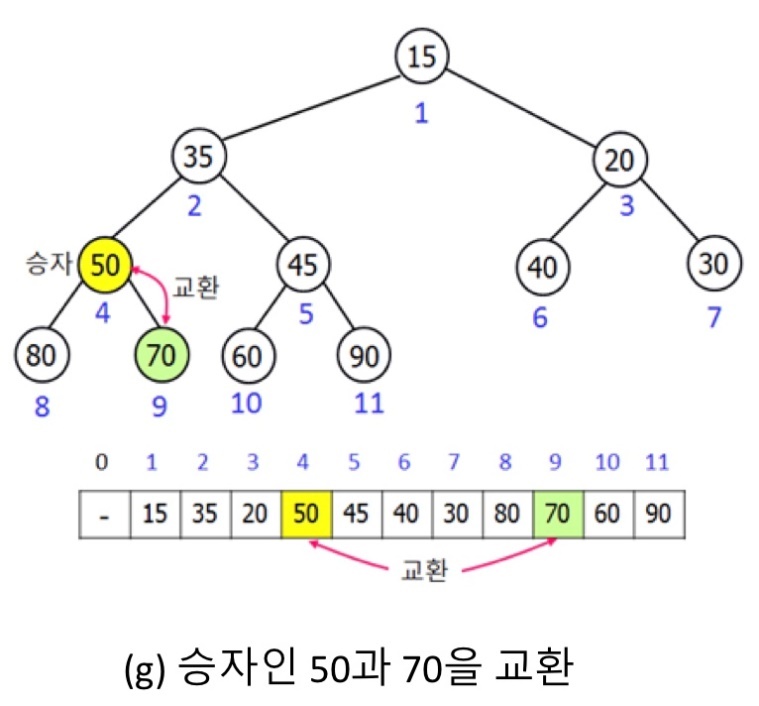
자동 생성된 설명**

**텍스트, 도표, 원, 라인이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 도표, 원, 라인이(가) 표시된 사진

자동 생성된 설명**

****

**# 최대 힙**

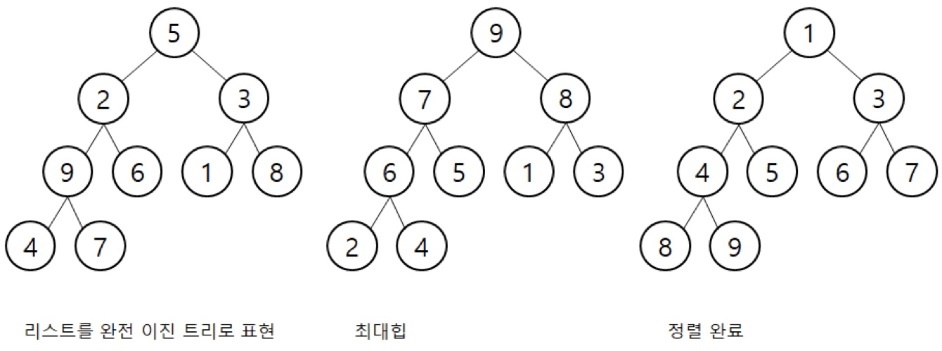
**: 키 값이 클수록 더 높은 우선순위**

스케치, 원, 도표, 화이트이(가) 표시된 사진

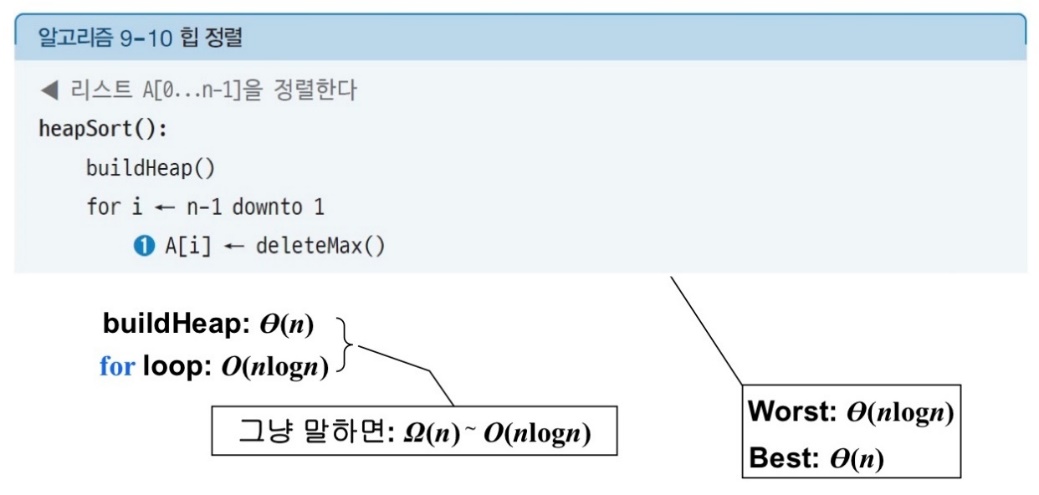
자동 생성된 설명

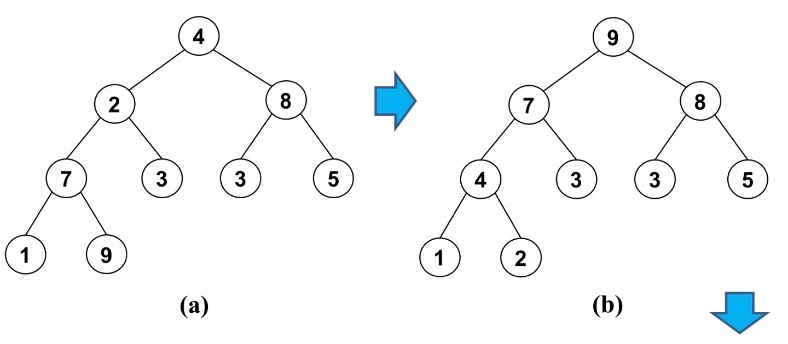
**# 힙 정렬(Heap sort)**

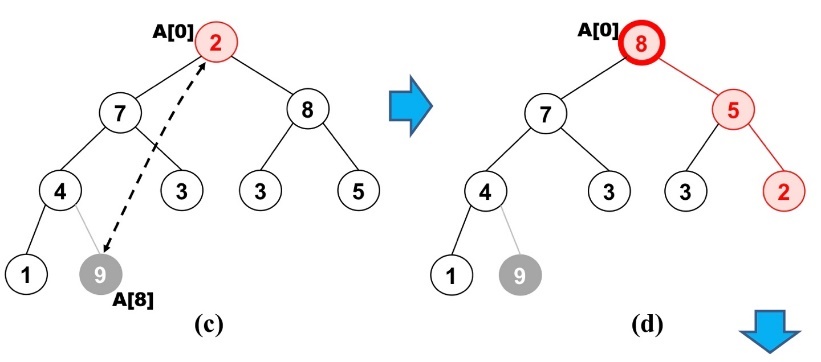
**: 힙을 이용한 정렬과정**

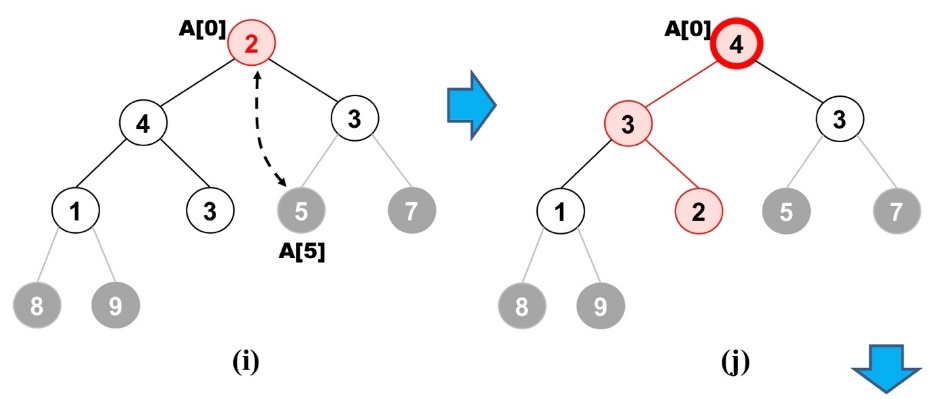
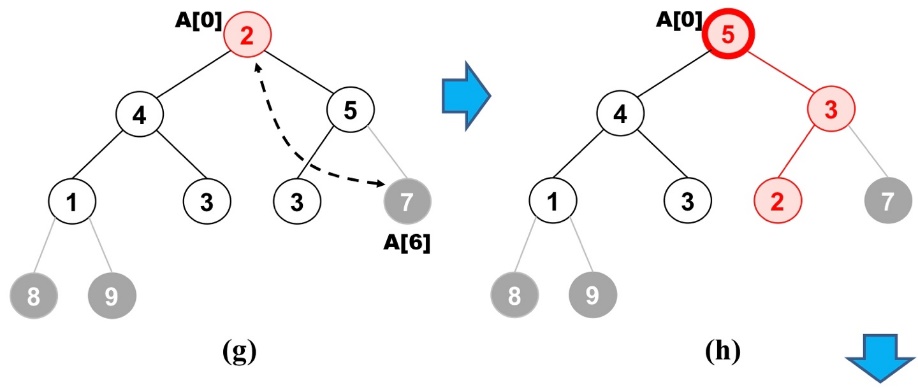
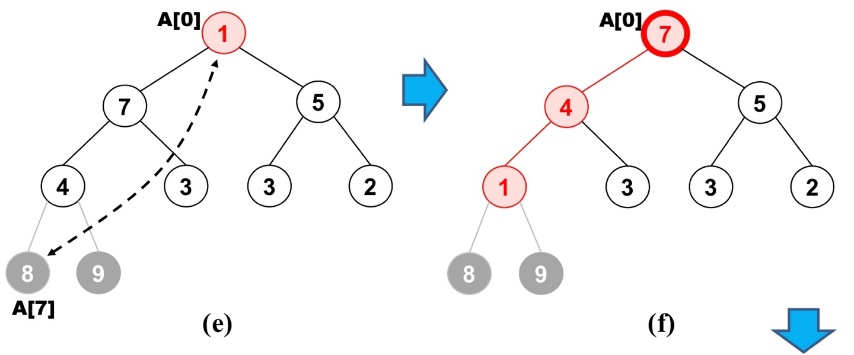


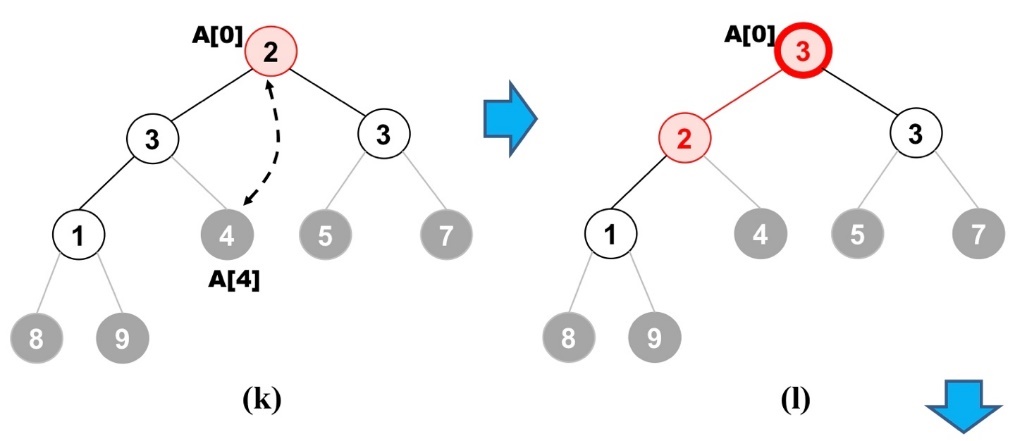
**- 힙정렬 : 알고리즘**

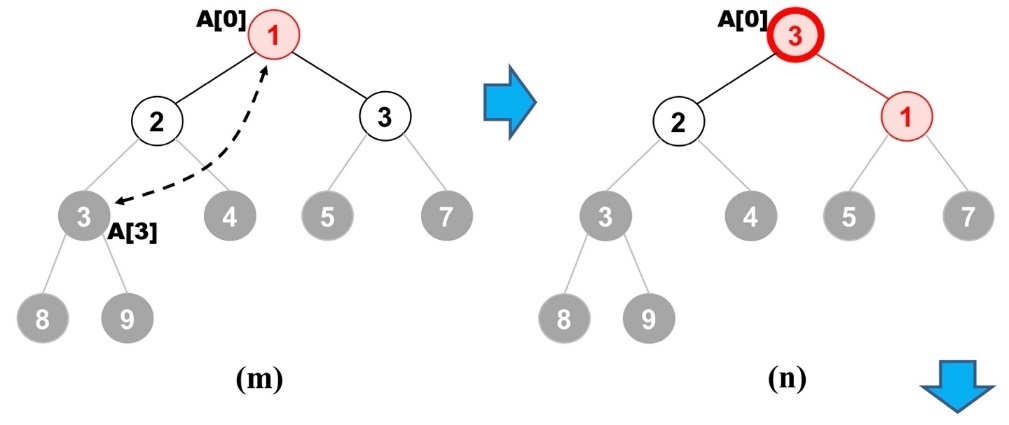


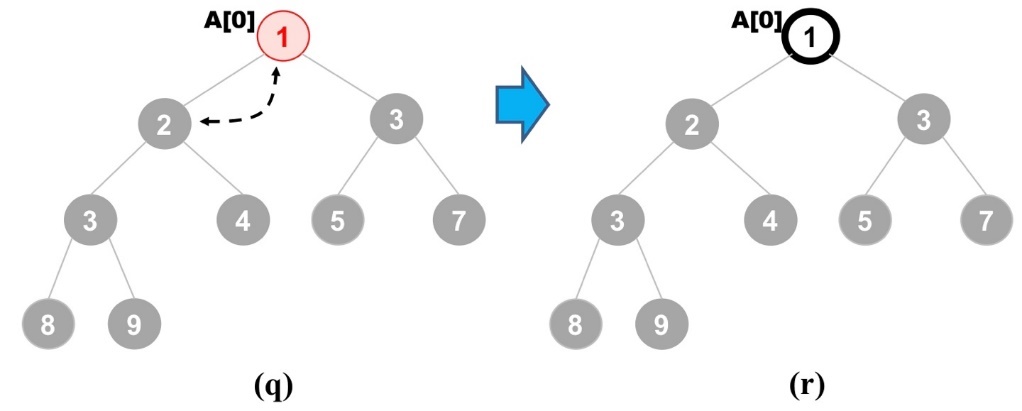
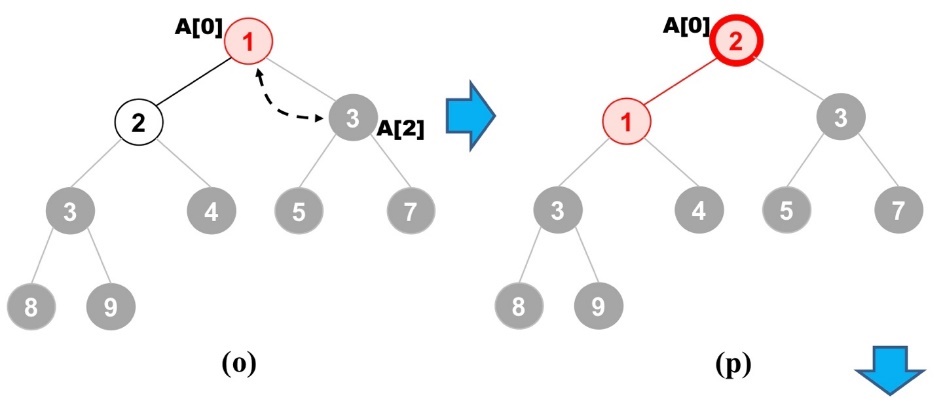












**- 힙 정렬 : 알고리즘 구현**

**: 임의의 정수(난수)를 힙 정렬 알고리즘을 이용하여 정렬**

**텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명**

**1. C.ver**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

#define arrMAXSIZE 10

typedef int element;

void HeapSort(element\* pArr, const int num);

void buildHeap(element\* pArr, const int num);

void percolateDown(element\* pArr, const int root, const int end);

void SWAP(element\* pa, element\* pb);

void PRINT(element\* pArr, const int num);

int main(void) {

int arr[arrMAXSIZE] = { 0 };

//임의의 난수 생성: 0~99사이의 정수

srand((unsigned int)time(NULL));

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = rand() % 100;

printf("정렬 전: "); PRINT(arr, arrMAXSIZE);

HeapSort(arr, arrMAXSIZE);

printf("정렬 후: "); PRINT(arr, arrMAXSIZE);

return 0;

}

//HeapSort: 힙 정렬

void HeapSort(element\* pArr, const int num) {

buildHeap(pArr, num);

for(int i=num-1; i>=0; i--){

//A[i] = deleteMax();

//i번째 위치에 제일 큰 값을 저장

SWAP(pArr, pArr + i);

//pArr[0]의 자료 이동으로 pArr[0:i-1]를 최대 힙으로 재구성

percolateDown(pArr, 0, i - 1);

}

}

//buileheap: 최대 힙 구성

void buildHeap(element\* pArr, const int num) {

for (int i = num / 2; i >= 0; i--)

percolateDown(pArr, i, num);

//print("buildHeap: "); PRINT(pArr, arrMAXSIZE);

}

void percolateDown(element\* pArr, const int root, const int end) {

int child = 2 \* root + 1;//왼쪽 자식

int right = 2 \* root + 2;//오른쪽 자식

if (child <= end) {

if (right <= end && pArr[child] < pArr[right])

child = right;

//child: A[2k+1]와 A[2k+2] 중에 큰 원소의 인덱스

if (pArr[root] < pArr[child]) {

SWAP(pArr + root, pArr + child);

percolateDown(pArr, child, end);

}

}

}

void SWAP(element\* pa, element\* pb) {

element temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

return;

}

void PRINT(element\* pArr, const int num) {

for (int i = 0; i < num; i++)

printf("%3d", \*(pArr + i));

printf("\n");

return;

}

**2. CPP.ver(3차 실습보고서\_연습문제#05\_힙 정렬)**

#include <iostream>

#include <random>

using namespace std;

#define arrMAXSIZE 15

template <typename E> void HeapSort(E\* pArr, const int num);

template <typename E> void buildHeap(E\* pArr, const int num);

template <typename E> void percolateDown(E\* pArr, const int root, const int num);

template <typename E> void SWAP(E\* pa, E\* pb) {

E temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

template <typename E> void PRINT(E\* pArr, const int num) {

for (int i = 0; i < num; i++)

cout << pArr[i] << " ";

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

// C++ 스타일 난수 생성

random\_device rd; // 시드 설정: random\_device 생성

mt19937 gen(rd()); // 난수 생성 엔진(mt19937) 초기화

uniform\_int\_distribution<int> dis(0, 99); // 균등 분포 정의: 범위 지정

for (int i = 0; i < arrMAXSIZE; i++)

\*(arr + i) = dis(gen);

cout << "20215229 이채훈\n" << endl;

cout << "정렬 전: "; PRINT(arr, arrMAXSIZE);

buildHeap(arr, arrMAXSIZE);

cout << "buildheap: "; PRINT(arr, arrMAXSIZE);

HeapSort(arr, arrMAXSIZE);

cout << "정렬 후: "; PRINT(arr, arrMAXSIZE);

return 0;

}

template <typename E>

void percolateDown(E\* pArr, const int root, const int end) {

int parent = root;

int child;

while ((parent \* 2) + 1 <= end) {

// 왼쪽 자식

child = (parent \* 2) + 1;

// 오른쪽 자식이 있고, 오른쪽 자식이 더 크다면 child를 오른쪽 자식으로 할당

if (child + 1 <= end && pArr[child] < pArr[child + 1]) {

++child;

}

// 부모가 자식보다 작으면 교환

if (pArr[parent] < pArr[child]) {

SWAP(&pArr[parent], &pArr[child]);

parent = child;

}

else {

break;

}

}

}

template <typename E>

void buildHeap(E\* pArr, const int num) {

// 마지막 비닐 노드의 인덱스 계산

int startIdx = (num / 2) - 1;

// 마지막 비닐 노드부터 루트까지 힙 속성을 유지하면서 반복

for (int i = startIdx; i >= 0; --i) {

percolateDown(pArr, i, num - 1);

}

}

template <typename E>

void HeapSort(E\* pArr, const int num) {

// 1. 힙 생성

buildHeap(pArr, num);

// 2. 힙 정렬

for (int i = num - 1; i > 0; --i) {

// 최대 값을 배열 끝으로 이동

SWAP(&pArr[0], &pArr[i]);

// 힙 속성 유지하면서 힙 크기 감소

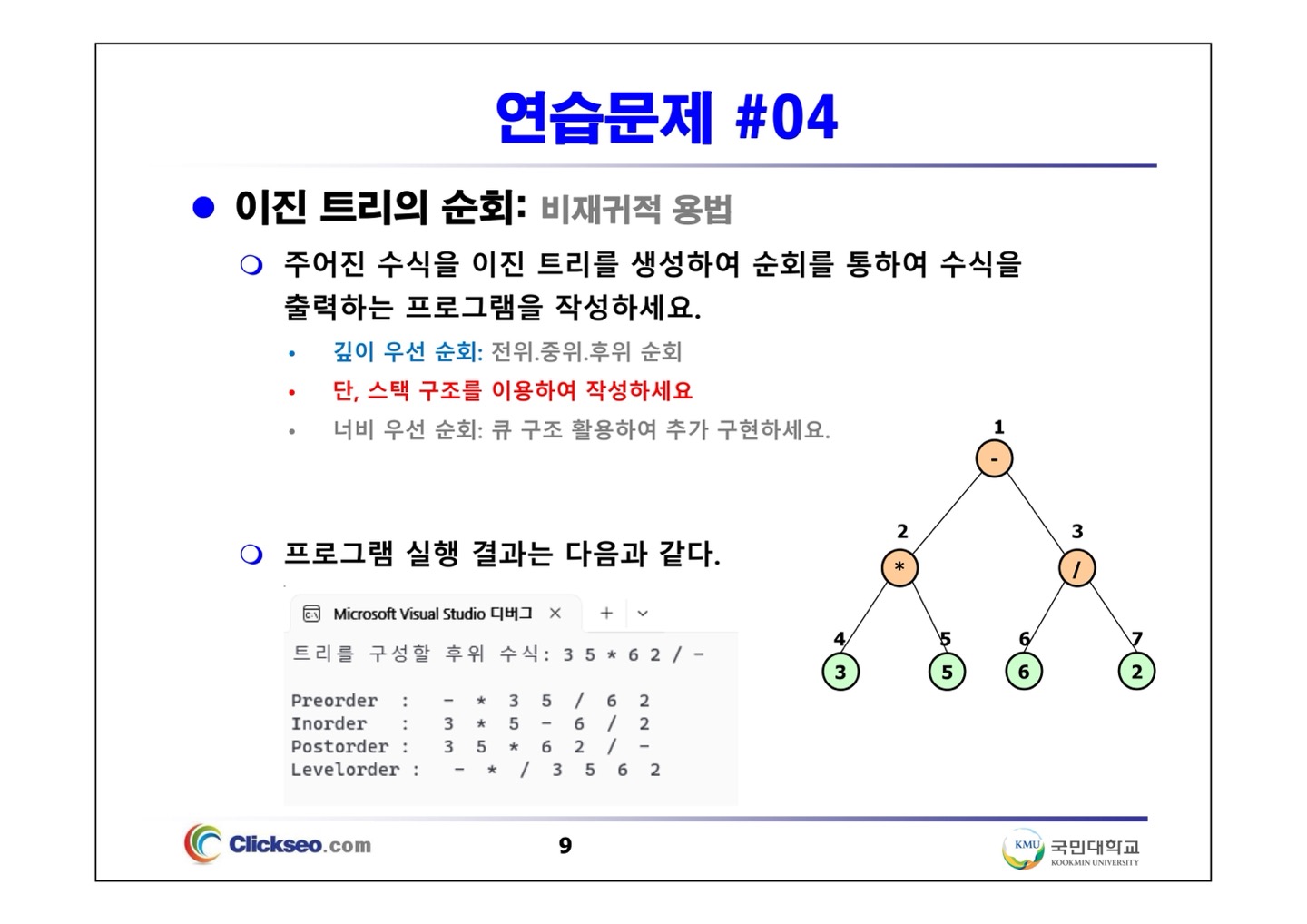
percolateDown(pArr, 0, i - 1);

}

}

**07-4. 3차 실습보고서\_연습문제#04\_이진트리의 순회**

**# 문제**

****

**# 소스코드**

**\* LinkedBtree.cpp**

#include <iostream>

#include <string>

#include "LinkedBtree.h"// class DNode

using namespace std;

int main(void)

{

string postfix;

cout << "20215229 이채훈" << endl;

cout << "\n트리를 구성할 후위 수식: ";

getline(cin, postfix);

// 이진 트리 생성 및 구성

LinkedBTree<char> BT;

DNode<char>\* root = BT.makeLinkedBTree(postfix);

// 깊이 우선 순회: 전위.중위.후위 순회

cout << "\nPreorder : "; BT.Preorder(root); // 전위 순회

cout << "\nInorder : "; BT.Inorder(root); // 중위 순회

cout << "\nPostorder : "; BT.Postorder(root); // 후위 순회

// 너비 우선 순회

cout << "\nLevelorder : "; BT.Levelorder(root);

return 0;

}

**\* LinkedBtree.h**

#include <iostream>

#include <string>

#include <stack>

#include <queue>

#include "LinkedQueue.h"

using namespace std;

// 연산자 여부를 판단

int isOperator(int op) {

return op == '+' || op == '-' || op == '\*' || op == '/';

}

template <typename E>

class LinkedBTree;

// class DNode

template <typename E>

class DNode {

public:

DNode(const E& data);

private:

E \_\_data;

DNode<E>\* \_\_Llink;

DNode<E>\* \_\_Rlink;

template <typename E> friend class LinkedBTree;

template <typename E> friend class LinkedQueue;

};

//DNode: 생성자와 메소드 정의

template <typename E>

DNode<E>::DNode(const E& data)

:\_\_data(data), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

// class LinkedBTree

template <typename E>

class LinkedBTree {

public:

LinkedBTree();

DNode<E>\* makeLinkedBTree(const string str);

void Preorder(DNode<E>\* root) const;

void Inorder(DNode<E>\* root) const;

void Postorder(DNode<E>\* root) const;

void Levelorder(DNode<E>\* root)const;

private:

DNode<E>\* root;

};

//LinkedBTree class : 생성자와 소멸자

template <typename E>

LinkedBTree<E>::LinkedBTree()

: root(nullptr) {}

//이진 트리(수식 트리) 생성

template <typename E>

DNode<E>\* LinkedBTree<E>::makeLinkedBTree(const string str) {

stack<DNode<E>\*> s;

DNode<E>\* tNode;

for (int i = 0; i < str.length(); i++) {

//공백 제거

while (str[i] == ' ')

i++;

//새로운 노드를 생성

tNode = new DNode<E>(str[i]);

//연산자일 경우...

if (isOperator(str[i])) {

tNode->\_\_Rlink = s.top(); s.pop();

tNode->\_\_Llink = s.top(); s.pop();

}

s.push(tNode);

}

tNode = s.top();

s.pop();

//루트 노드 설정

root = tNode;

return tNode;

}

//깊이 우선 순회: 전위 순회(비재귀)

template <typename E>

void LinkedBTree<E>::Preorder(DNode<E>\* root)const {

if (root == nullptr)

return;

stack<DNode<E>\*> s;

s.push(root);

while (!s.empty()) {

DNode<E>\* tNode = s.top();

s.pop();

cout.width(3); cout << tNode->\_\_data;

if (tNode->\_\_Rlink) s.push(tNode->\_\_Rlink);

if (tNode->\_\_Llink) s.push(tNode->\_\_Llink);

}

}

//깊이 우선 순회: 중위 순회(비재귀)

template <typename E>

void LinkedBTree<E>::Inorder(DNode<E>\* root)const {

stack<DNode<E>\*> s;

DNode<E>\* tNode = root;

while (tNode || !s.empty()) {

while (tNode) {

s.push(tNode);

tNode = tNode->\_\_Llink;

}

tNode = s.top();

s.pop();

cout.width(3); cout << tNode->\_\_data;

tNode = tNode->\_\_Rlink;

}

}

//깊이 우선 순회: 후위 순회(비재귀)

template <typename E>

void LinkedBTree<E>::Postorder(DNode<E>\* root)const {

if (root == nullptr)

return;

stack<DNode<E>\*> s1, s2;

s1.push(root);

while (!s1.empty()) {

DNode<E>\* tNode1 = s1.top();

s1.pop();

s2.push(tNode1);

if (tNode1->\_\_Llink) s1.push(tNode1->\_\_Llink);

if (tNode1->\_\_Rlink) s1.push(tNode1->\_\_Rlink);

}

while (!s2.empty()) {

DNode<E>\* tNode2 = s2.top();

s2.pop();

cout.width(3); cout << tNode2->\_\_data;

}

}

//너비 우선 순회

template <typename E>

void LinkedBTree<E>::Levelorder(DNode<E>\* root)const {

if (root == nullptr)

return;

queue<DNode<E>\*> Q;

Q.push(root);

while (!Q.empty()) {

DNode<E>\* tNode = Q.front();

Q.pop();

cout.width(3);

cout << tNode->\_\_data;

if (tNode->\_\_Llink) Q.push(tNode->\_\_Llink);

if (tNode->\_\_Rlink) Q.push(tNode->\_\_Rlink);

}

cout << endl;

}

**# 연구조사**

**# is Operator()**

: 주어진 값이 연산자인지 판단하는 함수

**->** 주어진 값이 '+' 또는 '-' 또는 '\*' 또는 '/' 중 하나인지 확인하여, 맞으면 1을 반환하고, 아니면 0을 반환.

**# Preorder**

1. root를 먼저 방문한 후, 왼쪽 서브트리를 전체적으로 방문하고, 그 다음에 오른쪽 서브트리를 전체적으로 방문함.

2. 스택을 사용하여 노드를 처리하므로, 자식 노드의 방문 순서는 왼쪽에서 오른쪽임.

3. 출력 결과는 전위 순서대로 노드의 데이터가 나열됨.

- 조건 확인

: root 노드가 비어있는지 확인, 비어 있을 경우 함수를 종료함.

- 스택 초기화

: 노드의 순서를 기억하는데 사용됨

- 노드 처리

: 스택이 비어있지 않은 동안 계속 반복함. -> 현재 스택의 맨 위에 있는 노드를 가져옴->스택에서 노드 제거->노드의 데이터 출력

- 자식 노드 처리

: 오른쪽 자식이 있는 경우, 스택에 삽입/왼쪽 자식이 있는 경우, 스택에 삽입

**# Inorder**

1. 왼쪽 서브트리를 먼저 방문한 후 루프틑 방문하고, 그 다음에 오른쪽 서브트리를 방문함.

2. 스택을 사용하여 왼쪽 자식으로 이동하면서 부모 노드를 스택에 저장하고, 그 후에 부모 노드를 방문하여 출력함.

3. 출력 결과는 중위 순서대로 노드의 데이터가 나열됨.

- 루프 시작

: 현재 노드가 비어있지 않거나 스택이 비어있지 않은 동안 계속 반복함.

- 왼쪽 자식 방문

: 현재 노드가 비어있지 않은 동안 반복함. -> 현재 노드를 스택에 push -> 현재노드를 왼쪽 자식으로 이동

- 노드 처리

: 스택의 맨 위에 있는 노드를 가져옴. -> 스택에서 노드 제거

- 오른쪽 자식 방문

: 현재노드를 오른쪽 자식으로 이동

**# Postorder**

1. 왼쪽 서브트리를 먼저 방문한 후 오른쪽 서브트리를 방문하고, 그 다음에 루트를 방문함.

2. 두 개의 스택을 사용하여 루트를 기준으로 왼쪽 자식과 오른쪽 자식을 각각 다른 스택에 저장하고, 두번째 스택에서 노드를 꺼내어 출력함.

3. 출력 결과는 후위 순서대로 노드의 데이터가 나열됨.

- root노드 스택에 저장

: root 노드를 첫 번째 스택에 삽입

- 첫 번째 스택 이용하여 노드 저장

: 첫 번째 스택이 비어있지 않은 동안 계속 반복. -> 첫 번째 스택의 맨 위에 있는 노드를 가져옴. -> 첫 번째 스택에서 노드를 제거함. -> 두번째 스택에 노드를 넣음.

- 자식 노드 처리

: 왼쪽 자식이 있으면, 첫 번째 스택에 왼쪽 자식 push/오른쪽 자식이 있으면, 첫 번째 스택에 오른쪽 자식 push

- 두 번째 스택

: 두 번째 스택이 비어있지 않은 동안 계속 반복. -> 두 번째 스택의 맨 위에 있는 노드를 가져옴. -> 두 번째 스택에서 노드를 제거함.

**# Levelorder**

1. 같은 레벨의 노드를 왼쪽에서 오른쪽으로 순서대로 방문함.

2. 큐를 사용하여 레벨 단위로 노드를 저장하고 처리함.

3. 출력 결과는 레벨별로 노드의 데이터가 나열됨.

- root노드 큐에 저장

: root 노드를 큐에 삽입

- 큐를 이용하여 노드 저장

: 큐가 비어있지 않은 동안 계속 반복. -> 큐의 맨 앞에 있는 노드를 가져옴. -> 큐에서 노드를 제거함.

- 자식 노드 처리

: 왼쪽 자식이 있으면, 왼쪽 자식을 큐에 push/오른쪽 자식이 있으면, 오른쪽 자식을 큐에 push

**07-5. 3차 실습보고서\_연습문제#04\_이진트리의 순회ver.Ong**

**# 코드**

#include <iostream>

#include <string>

#include <stack>

#include <queue>

using namespace std;

template <typename E>

class LinkedBTree;

// 클래스 DNode

template <typename E>

class DNode {

public:

DNode(const E& data);

private:

E \_\_data;

DNode<E>\* \_\_Llink;

DNode<E>\* \_\_Rlink;

template <typename T> friend class LinkedBTree;

};

template <typename E>

DNode<E>::DNode(const E& data) :

\_\_data(data), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

//클래스 LinkedBTree

template <typename E>

class LinkedBTree {

public:

LinkedBTree();

~LinkedBTree();

DNode<E>\* makeLinkedBTree(const string str);

void Preorder(DNode<E>\* root) const;

void Inorder(DNode<E>\* root) const;

void Postorder(DNode<E>\* root) const;

void Levelorder(DNode<E>\* root) const;

private:

DNode<E>\* \_\_root;

};

//생성자

template <typename E>

LinkedBTree<E>::LinkedBTree()

: \_\_root(nullptr){}

//소멸자

template <typename E>

LinkedBTree<E>::~LinkedBTree()

{

if (\_\_root == nullptr)

return;

DNode<E>\* tNode = nullptr;

queue<DNode<E>\*> Q;

Q.push(\_\_root);

while (!Q.empty()) {

tNode = Q.front();

Q.pop();

if (tNode->\_\_Llink) Q.push(tNode->\_\_Llink);

if (tNode->\_\_Rlink) Q.push(tNode->\_\_Rlink);

delete tNode;

}

}

//공백 여부 판단

int isOperator(int op)

{

return op == '+' || op == '-' || op == '\*' || op == '/';

}

//이진트리 생성

template <typename E>

DNode<E>\* LinkedBTree<E>::makeLinkedBTree(const string str)

{

stack<DNode<E>\*> S;

DNode<E>\* tNode;

for (int i = 0; i < str.length(); i++)

{

while (str[i] == ' ')

i++;

tNode = new DNode<E>(str[i]);

if (isOperator(str[i])) {

tNode->\_\_Rlink = S.top(); S.pop();

tNode->\_\_Llink = S.top(); S.pop();

}

S.push(tNode);

}

tNode = S.top();

S.pop();

// 루트 노드 설정

\_\_root = tNode;

return tNode;

}

// 깊이 우선 순회: 전위 순회

template <typename E>

void LinkedBTree<E>::Preorder(DNode<E>\* root) const

{

if (root == nullptr) return;

stack<DNode<E>\*> S;

S.push(root);

while (!S.empty())

{

DNode<E>\* tNode = S.top();

S.pop();

cout.width(3); cout << tNode->\_\_data;

if (tNode->\_\_Rlink) S.push(tNode->\_\_Rlink);

if (tNode->\_\_Llink) S.push(tNode->\_\_Llink);

}

cout << endl;

}

// 깊이 우선 순회: 중위 순회

template <typename E>

void LinkedBTree<E>::Inorder(DNode<E>\* root) const

{

if (root == nullptr) return;

stack<DNode<E>\*> S;

DNode<E>\* ptr = root;

while (ptr != nullptr || !S.empty())

{

while (ptr != nullptr)

{

S.push(ptr);

ptr = ptr->\_\_Llink;

}

ptr = S.top();

S.pop();

cout.width(3); cout << ptr->\_\_data;

ptr = ptr->\_\_Rlink;

}

cout << endl;

}

// 깊이 우선 순회: 후위 순회

template <typename E>

void LinkedBTree<E>::Postorder(DNode<E>\* root) const

{

if (root == nullptr) return;

stack<DNode<E>\*> S1, S2;

S1.push(root);

while (!S1.empty())

{

DNode<E>\* tNode = S1.top();

S1.pop();

S2.push(tNode);

if (tNode->\_\_Llink) S1.push(tNode->\_\_Llink);

if (tNode->\_\_Rlink) S1.push(tNode->\_\_Rlink);

}

while (!S2.empty())

{

DNode<E>\* tNode = S2.top();

S2.pop();

cout.width(3); cout << tNode->\_\_data;

}

cout << endl;

}

// 너비 우선 순회: 비재귀적 용법(큐 구조 활용)

template <typename E>

void LinkedBTree<E>::Levelorder(DNode<E>\* root) const

{

if (root == nullptr)

return;

queue<DNode<E>\*> Q;

Q.push(root);

while (!Q.empty()) {

DNode<E>\* tNode = Q.front();

Q.pop();

cout.width(3);

cout << tNode->\_\_data;

if (tNode->\_\_Llink) Q.push(tNode->\_\_Llink);

if (tNode->\_\_Rlink) Q.push(tNode->\_\_Rlink);

}

cout << endl;

}

int main(void)

{

string postfix;

cout << "트리를 구성할 후위 수식: ";

getline(cin, postfix);

// 이진 트리 생성 및 구성

LinkedBTree<char> BT;

DNode<char>\* root = BT.makeLinkedBTree(postfix);

// 깊이 우선 순회: 전위.중위.후위 순회

cout << "\nPreorder : "; BT.Preorder(root); // 전위 순회

cout << "Inorder : "; BT.Inorder(root); // 중위 순회

cout << "Postorder : "; BT.Postorder(root); // 후위 순회

// 너비 우선 순회

cout << "Levelorder : "; BT.Levelorder(root);

return 0;

}

**# 연구조사**

위의 코드에서 DNode 클래스로 템플릿으로 정의된 이진트리의 노드를 나타냈으며,

makeLinkedBTree 함수로 주어진 후위 수식을 이용하여 이진트리를 생성하는 동작을 수행한다. 이때, 스택을 활용하여 연산자와 피연산자를 처리하며 이진트리를 구성한다.

문제에 주어진 조건대로 큐로 구현한 levelorder를 제외한 preorder, inorder, postorder는 스택 구조로 구현하여 입력받은 후위 수식을 표현하였다. 이때 이진 트리의 순회 방식에 대한 구현은 스택이나 큐를 이용하여 비재귀적으로 수행되어 있다.이 코드의 주요 알고리즘에 대해 간략히 나타낸 그림은 다음과 같다.

|  |  |  |
| --- | --- | --- |
| 깊이 우선 순회 | | |
| 전위 순회 | 중위 순회 | 후위 순회 |
| 도표, 라인, 스크린샷, 원이(가) 표시된 사진  자동 생성된 설명  루트 노드를 먼저 방문하고, 왼쪽 서브트리를 전위 순회한 뒤에 오른쪽 서브트리를 전위 순회한다. | 라인, 도표, 원, 스크린샷이(가) 표시된 사진  자동 생성된 설명  왼쪽 서브트리를 중위 순회한 뒤에 루트 노드를 방문하고, 오른쪽 서브트리를 중위 순회한다. | 라인, 도표, 스크린샷, 원이(가) 표시된 사진  자동 생성된 설명  왼쪽 서브트리를 후위 순회한 뒤에 오른쪽 서브트리를 후위 순회하고, 마지막으로 루트 노드를 방문한다. |
| 너비 우선 순회 | | |
| 원, 라인, 도표, 디자인이(가) 표시된 사진  자동 생성된 설명  루트 노드부터 시작하여 각 레벨 순서대로 좌에서 우로 노드를 방문한다. 이진 트리를 너비 우선으로 탐색하는 방법으로, 노드의 순회 순서는 레벨 단위로 좌에서 우로 이동하면서 출력한다. | | |

**07-6. 3차 실습보고서\_연습문제#05\_힙 정렬ver.Ong**

**# 코드**

#include <iostream>

#include <cstdlib>

#include <ctime>

#define SIZE 10

using namespace std;

// 함수 선언

template <typename E>

void HeapSort(E\* pArr, const int num);

template <typename E>

void buildHeap(E\* pArr, const int num);

template <typename E>

void percolateDown(E\* pArr, const int root, const int num);

template <typename E>

void SWAP(E\* pa, E\* pb);

template <typename E>

void PRINT(E\* pArr, const int num);

template <typename E>

void SWAP(E\* pa, E\* pb)

{

E temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

return;

}

template <typename E>

void PRINT(E\* pArr, const int num)

{

for (int i = 0; i < num; i++)

{

cout.width(3); cout << \*(pArr + i);

}

cout << endl;

return;

}

template <typename E>

void buildHeap(E\* pArr, const int num)

{

for (int i = num / 2; i >= 0; i--) percolateDown(pArr, i, num - 1);

cout << "buildHeap: "; PRINT(pArr, SIZE);

}

template <typename E>

void percolateDown(E\* pArr, const int root, const int num)

{

int child = 2 \* root + 1; // 왼쪽 자식

int right = 2 \* root + 2; // 오른쪽 자식

if (child <= num)

{

if (right <= num && pArr[child] < pArr[right]) child = right;

if (pArr[root] < pArr[child])

{

SWAP(pArr + root, pArr + child);

percolateDown(pArr, child, num);

}

}

}

template <typename E>

void HeapSort(E\* pArr, const int num)

{

buildHeap(pArr, num);

for (int i = num - 1; i >= 0; i--)

{

SWAP(pArr + 0, pArr + i);

percolateDown(pArr, 0, i - 1);

}

}

int main()

{

int arr[SIZE] = { 0 };

//초기 난수 생성해 배열에 넣기

srand((unsigned int)time(NULL));

for (int i = 0; i < SIZE; i++) \*(arr + i) = rand() % 100;

cout << "정렬 전: "; PRINT(arr, SIZE);

HeapSort(arr, SIZE);

cout << "정렬 후: "; PRINT(arr, SIZE);

return 0;

}

**# 연구조사**

위 코드는 생성한 난수 여러 개를 정렬하지 않은 상태에서 처음에 출력,

HeapSort 함수에서 buildHeap 함수를 호출하여 주어진 배열을 최대 힙으로 만든 상태에서 출력, 최대 힙이 구성된 상태에서 HeapSort 함수를 호출하여 힙 정렬을 수행하고 정렬된 최종적인 결과 배열을 출력하는 코드이다.

|  |
| --- |
| // 함수 선언  template <typename E>  void HeapSort(E\* pArr, const int num);  template <typename E>  void buildHeap(E\* pArr, const int num);  template <typename E>  void percolateDown(E\* pArr, const int root, const int num);  template <typename E>  void SWAP(E\* pa, E\* pb);  template <typename E>  void PRINT(E\* pArr, const int num); |

위와 같은 함수의 선언부를 넣어주어야 하는 이유는 함수의 구현부에서 서로 순서가 엉킨 채로 다른 함수를 호출하고 있기 때문이다.

**\* 힙 정렬 알고리즘**

힙은 완전 이진트리 기반의 트리형 자료구조로써 최댓값이나 최솟값을 찾아내기 위해 사용된다. 힙 정렬은 최대 힙 트리나 최소 힙 트리를 구성해 정렬을 하는 방법이다.

힙 정렬을 위한 과정은 다음과 같다.

- n개의 노드에 대한 완전 이진트리를 구성하고, 루트 노드로부터 부모노드, 왼쪽 자식 노드, 오른쪽 자식 노드 순으로 구성한다.

- 최대 힙(부모노드>자식노드인 트리)을 구성한다. 단말노드를 자식 노드로 가진 부모노트부터 구성하며 아래부터 루트까지 올라오며 순차적으로 만들어갈 수 있다.

- 가장 큰 수(루트에 위치)를 가장 끝의 노드와 맞바꾼다.

- 위의 과정을 반복한다.

🡪 힙 정렬의 시간 복잡도는 O(nlogn)이다.

**07-7. 3차 실습보고서\_연습문제#04\_이진트리의 순회ver.Seo**

**# 코드**

#pragma once

#include <iostream>

#include <string>

#include <stack>

#include <queue>

#include "LinkedQueue.h"

using namespace std;

// 연산자 여부를 판단

int isOperator(int op) {

return op == '+' || op == '-' || op == '\*' || op == '/';

}

template <typename E>

class LinkedBTree;

// class DNode

template <typename E>

class DNode {

public:

DNode(const E& data);

private:

E \_\_data;

DNode<E>\* \_\_Llink;

DNode<E>\* \_\_Rlink;

template <typename E> friend class LinkedBTree;

template <typename E> friend class LinkedQueue;

};

//DNode: 생성자와 메소드 정의

template <typename E>

DNode<E>::DNode(const E& data)

:\_\_data(data), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

// class LinkedBTree

template <typename E>

class LinkedBTree {

public:

LinkedBTree();

DNode<E>\* makeLinkedBTree(const string str);

void Preorder(DNode<E>\* root) const;

void Inorder(DNode<E>\* root) const;

void Postorder(DNode<E>\* root) const;

void Levelorder(DNode<E>\* root)const;

private:

DNode<E>\* root;

};

//LinkedBTree class : 생성자와 소멸자

template <typename E>

LinkedBTree<E>::LinkedBTree()

: root(nullptr) {}

//이진 트리(수식 트리) 생성

template <typename E>

DNode<E>\* LinkedBTree<E>::makeLinkedBTree(const string str) {

stack<DNode<E>\*> s;

DNode<E>\* tNode;

for (int i = 0; i < str.length(); i++) {

//공백 제거

while (str[i] == ' ')

i++;

//새로운 노드를 생성

tNode = new DNode<E>(str[i]);

//연산자일 경우...

if (isOperator(str[i])) {

tNode->\_\_Rlink = s.top(); s.pop();

tNode->\_\_Llink = s.top(); s.pop();

}

s.push(tNode);

}

tNode = s.top();

s.pop();

//루트 노드 설정

root = tNode;

return tNode;

}

/\*

//깊이 우선 순회: 전위 순회(재귀적 용법)

template <typename E>

void LinkedBTree<E>::Preorder(DNode<E>\* root)const {

if (root) {

cout.width(3); cout << root->\_\_data;

Preorder(root->\_\_Llink);

Preorder(root->\_\_Rlink);

}

}

\*/

//깊이 우선 순회: 전위 순회(비재귀적 용법)

template <typename E>

void LinkedBTree<E>::Preorder(DNode<E>\* root)const {

if (root == nullptr)

return;

stack<DNode<E>\*> S;

DNode<E>\* tNode;

S.push(root);

while (!S.empty()) {

tNode = S.top(); S.pop();

if (tNode) {

cout.width(3); cout << tNode->\_\_data;

S.push(tNode->\_\_Rlink);

S.push(tNode->\_\_Llink);

}

}

return;

}

/\*

//깊이 우선 순회: 중위 순회(재귀적 용법)

template <typename E>

void LinkedBTree<E>::Inorder(DNode<E>\* root)const {

if (root) {

Inorder(root->\_\_Llink);

cout.width(3); cout << root->\_\_data;

Inorder(root->\_\_Rlink);

}

}

\*/

//깊이 우선 순회: 중위 순회(비재귀적 용법)

template <typename E>

void LinkedBTree<E>::Inorder(DNode<E>\* root)const {

if (root == nullptr) return;

stack<DNode<E>\*> S;

DNode<E>\* tNode;

tNode = root;

while (true) {

//왼쪽 서브 트리를 스택에.

while (tNode != nullptr) {

S.push(tNode);

tNode = tNode->\_\_Llink;

}

if (S.empty())

break;

//스택에서 pop를 실행하여 노드 방문

tNode = S.top(); S.pop();

cout.width(3); cout << tNode->\_\_data;

//오른쪽 서브 트리로 이동

tNode = tNode->\_\_Rlink;

}

return;

}

/\*

//깊이 우선 순회: 후위 순회(재귀적 용법)

template <typename E>

void LinkedBTree<E>::Postorder(DNode<E>\* root)const {

if (root) {

Postorder(root->\_\_Llink);

Postorder(root->\_\_Rlink);

cout.width(3); cout << root->\_\_data;

}

}

\*/

//깊이 우선 순회: 후위 순회(비재귀적 용법)

template <typename E>

void LinkedBTree<E>::Postorder(DNode<E>\* root)const {

if (root == nullptr) return;

stack<DNode<E>\*> S;

DNode<E>\* tNode, \* pre;

tNode = root;

while (true) {

//왼쪽 서브 트리를 스택에..

while (tNode != nullptr) {

S.push(tNode);

tNode = tNode->\_\_Llink;

}

while (!S.empty()) {

pre = tNode;

tNode = S.top(); S.pop();

//오른쪽 서브 트리가 있을 경우..

if (tNode->\_\_Rlink != nullptr) {

if (tNode->\_\_Rlink == pre) {

cout.width(3);

cout << tNode->\_\_data;

}

else {

S.push(tNode);

tNode = tNode->\_\_Rlink;

break;

}

}

//오른쪽 서브 트리가 없으면 노드 방문

else {

cout.width(3);

cout << tNode->\_\_data;

}

}

if (S.empty())

break;

}

return;

}

//너비 우선 순회

template <typename E>

void LinkedBTree<E>::Levelorder(DNode<E>\* root)const {

if (root == nullptr)

return;

queue<DNode<E>\*> Q;

Q.push(root);

while (!Q.empty()) {

DNode<E>\* tNode = Q.front();

Q.pop();

cout.width(3);

cout << tNode->\_\_data;

if (tNode->\_\_Llink) Q.push(tNode->\_\_Llink);

if (tNode->\_\_Rlink) Q.push(tNode->\_\_Rlink);

}

cout << endl;

}