자료구조의 정의

- 컴퓨터 프로그램

> **데이터 + 명령어**

- 자료구조

> **데이터를 효율적으로 처리할 수 있도록 만들어진 데이터의 처리 방식**

- 알고리즘

> **데이터를 효율적으로 처리할 수 있는 방법**

→ 프로그램의 목적에 따라 올바른 자료구조를 사용할 경우 얻을 수 있는 이득

1) 메모리의 **효율적 사용**

2) 프로그램 **수행 시간의 단축**

◆ 효율적인 알고리즘 적용하기 위해서는 알맞은 자료구조가 선행 되어야 함

- 자료구조의 분류

1. 단순구조 : 기본적인 데이터타입들이 해당

   → EX) int, float, double, ...

2. 선형구조 : 각각의 자료들이 1:1로 연결된 형태

   → EX) 스택, 큐, 리스트, 덱, ...

3. 비선형구조 : 각각의 자료들이 1:다 또는 다:다로 연결된 형태

   → EX) 트리, 그래프, ...

4. 파일구조 : 일반적으로 메모리에 로드하기 어려운 큰 자료들이 대상

   → 보조기억장치에 저장하는 것을 전제로 만들어졌다.

   → 구성 방식에 따라 순차적, 상대적, 색인, 다중 키 파일구조등이 있다.

- 추상자료형

> **자료형이란?**

- 자료(데이터) + 명령어
- EX) int 자료형 = int + 사칙연산을 필두로 한 명령어들

> **정보은닉**

- 사용자로 하여금 핵심적인 정보만 확인할 수 있게 하고, 내부적 기능이나, 형태는 공개하지 않는 것 → 백조를 생각하면 편할 듯 하다.

> **추상자료형이란?**

- 정보은닉 + 자료형
- 데이터와 명령어로 구성된 자료형에 정보은닉의 개념을 쓰까 만든 것
- 사용에 있어서 핵심적인 데이터나 명령어들의 정의만 공개하고 내부 로직 같은 필수적이지 않은 부분들은 비공개로 정의한 형태

- 알고리즘

> **알고리즘이란?**

- 문제해결을 위한 일련의 절차
- 컴퓨터가 이해할 수 있는 명백한 명령어의 집합이면서 어떤 문제를 해결하기 위한 절차
- 알고리즘은 문제 해결을 위한 5가지 조건이 필요

1) 입력 : 외부에서 제공되는 자료가 0개이상 존재해야 한다.
2) 출력 : 적어도 1개 이상의 결과를 만들어야 한다.
3) 유한성 : 각 명령어는 의미가 모호해서는 안된다.
4) 명백성 : 한정된 수의 단계 뒤에는 반드시 종료되어야 한다.
5) 유효성 : 모든 명령은 실행 가능한 연산이어야 한다.

> **알고리즘의 표현법**

- 대표적으로 4가지의 표현 방법이 존재

  1) 자연어

  2) 순서도

  3) 의사코드

  4) 프로그래밍 언어

> **알고리즘의 분석기준**

- 공간 복잡도
- 시간 복잡도

  → 임베디드 환경을 제외한 대부분의 환경에서 시간 복잡도를 우선시 함

> **시간 복잡도**

- 계산방법
  - 한번의 연산을 1로 잡아 명령어들의 구성을 통해 함수를 도출
  - 반복문의 경우 명령어 동작시 적용되는 연산 마다 1씩 증가 후 * 반복횟수
- 빅오 표기법
  - 시간 복잡도 함수에서 가장 큰 차수의 항만 도출해 표기
  - O(n)의 형태
  - n < logn < nlogn < n^2 < n^3 < 2^n < n!

C프로그래밍

- 컴파일 및 실행 프로그램 작성

→ 빌드 시 출력되는 메시지에서 구성:DebugWin32가 출력 되는 것을 확인 가능

→ 일반적으로 visual Studio의 프로젝트는 2개의 빌드환경을 가짐

1) Debug : C 소스파일을 컴파일 할 때 디버깅 정보를 실행 프로그램에 자동 추가

→ 솔루션 파일이 저장되는 폴더의 하위Debug폴더에 저장

2) Release : 개발이 완료된다음에 실행 프로그램 배포 시에 Release환경에서 빌드 된 실행 파일을 배포

→ Release폴더에 저장

- 단순 자료형

> **정수**

- 소수점이하는 저장하지 않음
- unsigned추가시 음수 값에 할당된 비트까지 가져와서 표현 가능

> **실수**

- 소수를 포함하는 수
- 수의 범위에서 E라는 문자가 표현되는대, E다음에 오는 수가 지수(Exponential)이라는 의미이다.

> **문자**

- 문자를 나타내는대 사용

- 배열

> **같은 자료 형의 데이터를 메모리상에 연속적으로 저장하는 자료형**

> **연속된 각각의 값을 배열의 원소(Element)라고 한다.**

→ 어떤 자료형의 배열수를 추출하는 방법 : Sizeof(배열)/sizeof(배열의 자료형)

- 다차원 배열

> **배열의 원소가 배열 자료형인 배열**

> **2차원 배열에서는 열에 대한 인덱스가 행에 대한 인덱스보다 먼저 나온다**

> **물리적인 실제 메모리 구조는 순차적으로 할당된다.**

- 구조체

> **다른 자료형의 데이터를 하나의 그룹으로 묶은 자료형**

> **사용하려면 먼저 구조체에 대한 선언이 필요**

> **자료형 정의를 통해 하나의 독립된 구조를 가지도록 선언해야 함**

> **자료형 선언이라 후에 변수선언도 따로 해줘야 한다.**

> **Typedef를 통해 하나의 데이터타입으로 정의 하기도 한다.**

- 포인터

> **포인터 변수는 메모리주소값을 저장하는 변수이다.**

> **프로그램의 안정성 차원에서 가능한 변수 선언에 항상 NULL값을 넣어 초기화를 시켜 줘야 한다.**

> **포인터 연산**

1) 주소연산자(&) : 변수의 주소값을 얻기 위한 연산자

2) 참조연산자(*) : 포인터 변수에 저장된 주소를 이용해 해당 주소에 있는 값을 얻는 연산자

> **동적 메모리 할당**

- 프로그램 실행중(런타임)j에 임의의 크기로 메모리를 할당 할 수 있다.
- Void* Malloc(int size)을 사용해 할당 가능
  - 파라미터인 size바이트 만큼 메모리를 할당 후 할당된 메모리 블록의 시작 주소를 반환한다.
  - 할당 불가능시 NULL을 반환 → malloc호출후에는 반드시 NULL이 아닌지 검사를 해야 한다.
  - 할당시킨 메모리는 사용 후 반드시 void free(void *ptr)로 할당을 해제해 줘야 한다.
  - free함수의 경우 파라미터로 NULL또는 쓰레기값이 전달 될 경우 ㅈ 될수 있다.
- 초기화 : void *Memset(void *ptr_start, int value, size_t count)를 사용해 초기화 가능
  - String.h를 선언 시켜 줘야 한다.
  - Ptr_start가 가리키는 메모리 영역을 시작으로 Count크기만큼 value로 설정한다.

- **더블 포인터**

  > **다른 변수의 메모리 주소를 저장하는 포인터 변수의 주소를 저장**

  > **포인터와 배열**

  - 배열을 포인터로 나타내는 것 또한 가능하다.
  - 2차 배열의 경우 더블 포인터를 활용해 표현 가능하다.
  - 배열의 이름 = 포인터변수

  Ex) int arr[10]일 때, arr = 포인터 변수 → 값을 더해주는 것으로 다른 인덱스에 접근 가능

  - 2차배열의 이름 = 더블 포인터 변수

  Ex) int arr[a][b]일 때, arr = 더블 포인터 변수

  → 값을 더해주는 것으로 a의 인덱스에 접근 가능

  → 이후 도출된 값에 수를 더해주는 것으로 b인덱스에 접근 가능

  > **구조체에서도 포인터 사용가능**

  - 포인터 참조연산으로 값 도출 시 (.)연산자가 (*)연산자보다 우선순위가 높으니 주의

리스트

- **리스트**

  > **여러 개의 자료가 일직선으로 서로 연결된 선형 구조를 의미**

  > **리스트의 활용 예시**

  1. 문자리스트 : 문자들이 각각 차례대로 저장
     - 리스트에 저장되는 자료는 각각의 문자가 되고, 문자들은 순서대로 선형 구조를 이루게 된다.
  2. 문자열 리스트 : 저장되는 자료가 문자열
     - 제일 마지막 문자는 널문자('₩0')가 오게된다.

3. 행렬 : 행렬은 각각의 열이 행개수만큼의 자료를 가지는 리스트라 볼 수 있다.

4. 다항식 : 리스트를 활용하여 다항식 또한 저장 할 수 있다.

→ 다항식의 계수값을 저장하는 리스트로 표현 가능하다.

→ 계수가 0인 경우도 저장을 해야 한다.

- 리스트 추상자료형

> **리스트 사용에 필요한 추상자료형**

이름		입력	출력	설명
리스트 생성	createList()	최대 원소 개수 n	리스트 l	최대 n개의 원소를 가지는 공백(Empty)리스트 l을 생성
리스트 삭제	deleteList()	리스트 l	N/A	리스트의 모든 원소를 제거
원소 추가 가능여부판단	isFull()	리스트 l	TRUE/FALSE	리스트의 원소 개수가 최대 원소 개수와 같은지를 반환. 배열 리스트인 경우에만 의미가 있음
원소 추가	addElement()	리스트 l 원소 위치 p 원소 e	성공/실패 여부	원소 e를 리스트의 특정 위치 p에 추가
원소 제거	removeElement()	리스트 l 원소 위치 p	성공/실패 여부	리스트의 위치 p에 있는 원소를 제거
리스트 초기화	clearList()	리스트 l	N/A	리스트의 모든 원소를 제거
원소 개수	getListLength	리스트 l	원소의 개수	리스트의 모든 원소개수를 반환
원소 반환	getElement()	리스트 l 원소 위치 p	원소	리스트의 위치 p에 있는 원소를 반환

→ 경우에 따라서 디버깅 용도로 모든 노드를 출력하는 displayList()가 필요 할 수 있다.

- 배열리스트

> **배열을 통해 리스트를 구현한 것**

> **논리적 순서와 물리적 순서가 같다는 특징이 있다.**

> **원소 추가**

→ 배열리스트의 중간에 새로운 원소를 추가하려면 기존의 원소들을 이동 시켜 줘야 한다.

→ 원소의 논리적 순서를 보장하기 위해 물리적인 순서에 제약을 준 것

→ 배열의 가장 오른쪽 원소부터 왼쪽으로 진행

> **원소 제거**

→ 제거하려는 원소의 다음 인덱스부터 시작해 오른쪽으로 진행

> **구현**

→ github.com/Snack1511/DataStructure/tree/master/ArrayList

- 연결리스트

> **포인터를 이용하여 구현**

> **자료의 순서상으로는 연결된 듯 하지만 물리적으로는 서로 떨어져 있다.**

→ 링크에 의해 논리적으로만 연결되어 있음을 시사한다.

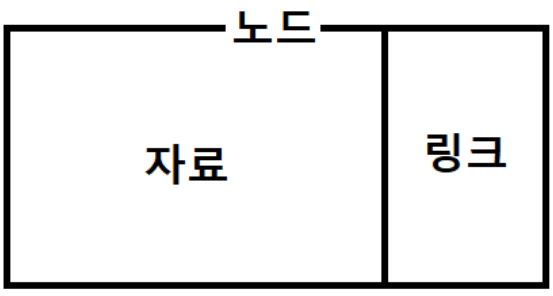
→ 배열리스트와 달리 저장 가능한 최대원소의 개수를 지정 할 필요가 없다.

→ 새로운 원소를 추가할 경우 동적으로 원소를 생성하고 포인터로 이어주면 된다.

> **연결 리스트의 노드구조**

→ 원소 = 노드인 배열리스트와 다르게 노드 = 자료 + 링크 형태로 구성되어 있다.

→ 이는 노드가 원소를 포함하는 개념이다.



→ 위 그림과 같이 노드는 자료를 저장하는 부분과 링크(다음 주소지)를 저장하는 부분으로 나뉘어 있다.

1) 자료 : 정수와 실수 같은 단순데이터 뿐만 아니라 복잡한 구조체 정보도 저장 할 수 있다.

2) 링크 : 포인터를 이용하여 현재 노드와 연결된 다음 노드를 가리킨다.

→ → 제일 마지막 노드는 다음 노드가 더 없으므로 NULL값을 가리킨다.

> **노드추가**

→ 기존에 연결된 링크를 제거하고 새로 추가되는 노드의 링크를 기존의 노드들과 연결시켜 준다.

> **노드제거**

→ 기존의 링크를 제거하고 다음 노드를 연결시켜준다

→ 연결이 끊긴 노드는 메모리를 해제해서 메모리 누수가 발생 하지 않도록 하는 것이 중요하다.

> **연결리스트의 장점**

1) 추가 원소 이동 연산 불필요

→ 배열리스트의 경우 모든 원소를 이동시키는 연산이 반드시 필요하다.

→ 연결리스트는 링크만 새로 연결 하면 되므로 이동연산의 제약에서 자유로워 진다.

2) 최대 원소 개수 지정 불필요

→ 배열리스트는 생성 시 반드시 최대 원소 개수를 지정해줘야 한다.

→ 연결리스트는 동적으로 메모리를 할당하므로 제약에서 자유롭다.

> **연결리스트의 단점**

1) 구현이 어렵다.

→ 동적인 메모리 할당 및 포인터 연산등으로 배열리스트보다 구현의 비용이 높다.

→ 메모리 관리와 관련해서 메모리 누수오류의 발생 가능성이 높다.

2) 탐색연산의 비용이 높다.

→ 배열리스트는 특정 원소에 대한 탐색은 O(1)의 시간복잡도를 가지는 반면 연결리스트는 O(n)의 시간복잡도를 가진다.

→ 이는 연결리스트의 경우 원하는 원소를 찾을 때까지 포인터로 노드를 탐색해야 하기 때문에 시간 비용이 높아질 수 밖에 없다.

> **연결리스트의 종류**

1) 단순 연결리스트

- 연결리스트의 가장 기본이다.
- 첫 노드부터 끝 노드까지 일직선으로 구현하기에 간단한 구조를 지닌다.
- 이전노드로의 링크가 없으므로 이전 노드 탐색 시 새로운 순회를 시작해야 한다.

2) 원형 연결리스트

- 연결리스트의 마지막 노드가 첫 노드를 가리키는 원형의 형태를 가진다.
- 이전노드를 탐색하려면 순회를 지속하면 된다.
  - 이 경우 전체 리스트를 한번 순회해야 한다.

→ 단순 연결리스트와 원형 연결리스트는 링크가 단 방향이라는 것에 주의하자.

3) 이중 연결리스트

- 노드 사이의 링크가 양방향으로 구성되어 있다.
- 이로 인해 이전 노드에 대한 직접 접근이 가능하다.

- 단순연결리스트

> **노드의 구조체 = 현재 노드 개수 + 헤더노드**

> 구현

- 원형연결리스트

> **리스트의 마지막 노드가 리스트의 첫 번째 노드와 연결된 단순연결리스트**

> **마지막 노드가 첫 번째 노드와 연결되어 순환 구조를 지님**

> **원형 연결리스트는 링크를 따라 이동하면 이전 노드에 접근 할 수 있다.**

→ 현재 노드의 이전노드가 현재노드의 다음 노드인지 확인하면 이전노드인지 판별 가

능

> **헤더포인터를 사용한 구현**

→ 헤더 포인터를 사용한 경우 추가 제거 기능에서 여러가지 고려해야 할 사항이 발생할 수 있다.

> **추가**

→ 헤드포인터를 이용하여 고려 시 3가지 경우를 생각해야 한다.

1) 리스트의 첫 번째 노드로 추가하는 경우

1-1) 공백리스트인 경우

1-2) 공백리스트가 아닌 경우

2) 리스트의 중간에 노드를 추가하는 경우

> **제거**

→ 이 또한 3가지 경우를 고려 해야 한다.

1) 리스트의 첫 번째 노드를 제거하는 경우

1-1) 제거하려는 노드가 리스트의 마지막 노드인 경우

1-2) 마지막 노드가 아닌 경우

2) 중간 노드를 제거하는 경우

> **구현**

- 이중연결리스트

> **각각의 노드마다 2개의 링크가 있기 때문에 특정 노드의 다음 노드뿐 아니라 이전 노드에도 직접 접근 가능**

> **이전노드에 직접적인 접근은 장점이나, 이 때문에 추가 메모리 공간을 더 사용한다는 단점이 존재**

> **마지막 노드가 첫 번째 노드를 가리키는 원형 연결 리스트의 속성을 지닌다.**

> **헤더노드를 포함하는 이중연결리스트는 임의의 위치에 있는 노드pNode에 대해 다음식이 성립한다.**

→ pNode == pNode->pLLink->pRLink == pNode->pRLink->pLLink

→ 이 식은 헤드포인터로는 성립될 수 없다.

> **생성**

→ 생성 시 헤더 노드에서 좌우 링크 모두 자기자신을 참조하도록 초기화시켜줘야 한다.

> **추가**

→ 노드가 추가될 적절한 위치를 찾아서 새로 생성한 노드를 삽입만 하면 된다.

> **제거**

→ 삭제하려는 노드의 이전 노드를 찾고 삭제하려는 노드의 좌우 링크를 재설정 해준다.

> **구현**

- 연결리스트의 응용

> **순회**

→ 노드 사이에 연결된 노드의 링크를 이용해, 보다 효율적으로 리스트 순회 가능

> **다른 리스트 연결**

→ 연결 주체 리스트에 연결하려는 리스트를 연결한 후 연결하려는 리스트의 링크를 해제해줘야 한다.

→ 각 리스트들의 노드개수 또한 수정해줘야 한다.

> **역순**

→ 현재 노드와 현재노드의 이전노드가 있다면 현재노드의 다음노드로의 링크를 이전노드로의 링크로 설정해주고, 현재노드의 다음노드로 이동한다.

1) B = A

2) A = C

3) C = C->link

4) A->link = B

> **다항식**

→ 각 항을 노드로 나타내고, 이런 항들의 집합인 다항식을 단순연결리스트로 구현

→ 각 항의 계수와 차수를 멤버변수를 이용해 저장하되, 차수의 내림차순으로 저장하고, 저장공간을 절약하기 위해 계수가 0인 항은 노드를 추가하지 않는다.

→ 다항식의 덧셈연산 : 3가지 경우를 고려

1) 다항식들 주체식의 차수가 높은경우 : 주체식에 대해서만 다음노드로 이동

2) 다항식들 중 주체식의 차수가 낮은 경우 : 더해주려는 식만 다음노드로 이동

3) 같은 경우 : 다항식들 전체의 노드 이동

→ 항들 끼리 더해준 후 남은 항들은 마지막에 연산

→ 연산 후 식들은 모두 메모리를 해제해줘야 한다.

> **옵션함수들 구현**

> **다항식 구현**

스택

- **스택**

> **자료를 쌓아두는 기능**

> **특성**

→ Last In First Out

→ 선형 자료구조 : 저장된 자료들 사이의 선후관계가 모두 1대1

> **제약사항**

→ 자료의 추가 및 반환은 스택의 끝에서만 가능

→ 스택의 끝이란 스택의 제일 위→ 가장 최근에 추가된 자료가 있는곳

→ 후입선출의 특성을 지니는 이유

> **후입선출의 특성은 다양한 알고리즘에서 필수적인 요소로 사용**

→ 수식을 해석하고 이를 계산하는데 사용

→ 복잡한 미로에서 길을 찾는 알고리즘등으로 사용

→ 트리나 그래프등 다른 복잡한 자료구조에서도 내부적으로 사용

> **푸쉬(Push)**

→ 새로운 자료를 스택에 추가하는 것

→ 스택의 제일 위에서만 발생

→ 스택의 크기는 스택이 저장할 수 있는 최대 자료의 개수를 의미

→ 스택의 크기를 초과해 새로운 자료를 추가할 때 자료가 추가될 수 없는 현상을 넘침(Overflow)현상이라 한다.

→ 탑을 어떻게 변경해야 할지를 가장 주의깊게 고려해야 한다.

> **팝(Pop)**

→ 스택에 저장된 자료를 사용하기 위해 스택에서 자료를 꺼내는 것

→ 스택의 제일 위에서만 발생

→ 스택에 아무 자료가 저장되지 않은 상태를 공백(Empty)이라 한다.

→ 공백상태에서 팝연산이 호출되면 부족(Underflow)현상이 발생

> **피크(Peek)**

→ 팝 연산과 달리 자료를 제거하지 않고 해당자료에 접근만 하는 것

→ 최상위 자료를 반환하는 점에서 팝연산과 비슷하지만 스택에서 자료를 제거하지 않고 반환만 시킨다.

> **추상자료형**

이름		입력	출력	설명
스택 생성	createStack()	스택의 크기 n	스택 s	최대 n개의 원소를 가지는 공백(Empty)스택 s를 생성
스택 삭제	deleteStack()	스택 s	N/A	스택제거(메모리 해제)
원소 추가 가능여부판단	isFull()	스택 s	TRUE/FALSE	스택의 원소 개수가 최대 원소 개수와 같은지를 반환. 배열 스택인 경우에만 의미가 있음

공백스택인지 여부 판단	isEmpty()	스택 s	TRUE/FALSE	공백 스택인지 여부를 전달
푸시	Push()	스택 s 원소 e	성공/실패 여부	스택의 맨 위에 새로운 원소를 추가
팝	Pop()	스택 s	원소 e	스택의 맨 위에 있는 원소를 제거한 뒤 이를 반환
피크	Peek()	스택 s	원소 e	스택의 맨 위에 있는 원소를 반환 (스택에서 제거하지는 않음)

→ createStack()과 deleteStack()은 각각 스택을 생성 및 삭제하는 연산이다.

→ isFull은 스택의 크기를 초과하는 overflow가 발생할지에 따라 원소가 추가 가능한지를 알려준다.

→ 구현 시 최대 저장 원소 개수라는 제약조건이 필요하지만, 스택을 어떻게 구현할지에 따라 필요가 없을 수도 있다.

→ isEmpty는 현제 스택이 공백상태인지 확인하는 기능이다.

→ 스택의 가장 최근 자료를 반환하는 팝/피크 연산은 스택에 있어서 가장 기본적인 연산이다.

→ 스택의 구현 방법은 배열을 이용한 방법과 포인터를 이용한 방법으로 구분할 수 있다.

> **배열로 구현한 스택**

→ C언어에서 사용하는 배열을 이용하여 스택을 구현한 것

→ 물리적으로 연결된 C의 배열을 이용하기 때문에 스택을 생성할 때 스택의 크기를 반드시 지정해 주어야 한다.

→ 배열로 구현한 스택은 스택의 크기(최대노드개수)를 내부변수로 저장하고 있다.

→ 스택을 생성해 줄 때에 스택의 크기에 해당하는 개수만큼 노드의 배열을 생성해준다.

→ 스택에 저장된 현재 노드 개수 또한 내부변수에 저장

→ 이를 이용해 스택에서의 탑의 위치를 알수있다. → 탑의 위치 = 현재노드개수 – 1

> **배열스택 구현**

> **연결리스트로 구현한 스택**

→ 배열로 구현시에는 스택의 크기를 미리 지정해야 하고, 최대저장수보다 많은 자료를 저장하려고 할 때 오버플로우가 발생

→ 연결리스트로 스택을 구현할 때에는 노드 추가 때마다 동적으로 메모리를 할당하기에 스택크기를 미리 지정해 줄 필요가 없다.

→ 보다 효율적인 메모리 사용이 가능

→ 탑노드를 포인터 변수가 기리키고 있기 때문에 포인터 변수로 직접 접근 할 수 있다.

→ 노드 추가시에는 새로운 노드를 연결리스트에 추가하고 탑을 가리키는 포인터변수는 새로 추가된 노드를 가리키도록 수정해야 한다.

→ 또한 새로운 탑 노드의 다음노드가 기존의 탑 노드가 되도록 노드사이의 링크를 설정해줘야 한다.

→ 정리 : 연결리스트로 구현한 스택은 최대저장개수로 인한 오버플로우가 발생하지 않는 대신 구현이 어렵다

> **이전에 했던 연결리스트를 조금 변형시키면?**

→ 스택은 탑에서만 자료를 추가/제거 할 수 있기 때문에 범용적인 리스트를 직접 이용하는 것보다 스택에 맞춰 구현해 주는게 훨씬 효율적이다.

→ 기존의 연결리스트를 활용할 경우 가장 마지막에 추가된 노드에 접근하기 위해 리스트 전체를 순회해야 한다.

→ 연결리스트스택의 경우 가장 마지막에 추가된 노드가 스택구조체 내에서 탑포인터에 의해 링크되어 있어서 접근이 쉽다.

→ 기존의 연결리스트와 다르게 추가순서와 연결방향이 반대라는 것 또한 차이점이다.

→ 가장마지막에 추가된 자료의 다음 노드는 그 이전에 추가된 노드가 되고, 스택에 맨 처음 추가된 노드는 다음 노드값이 NULL이 된다.

> **연결리스트스택 구현**

- 스택응용1

> **역순문자열**

→ 입력받은 문자열의 순서를 뒤집은 문자열을 출력하는 함수

→ 스택을 활용해 비교적 간단하게 구현 가능

1) 문자열의 모든 문자를 순서대로 스택에 푸시

2) 공백스택이 될 때까지 스택의 모든 노드를 팝

3) 정반대 순서대로 문자들이 스택에서 추출

> **수식괄호검사**

→ 2가지를 점검해야한다.

1) 여는 괄호와 닫는 괄호가 서로 쌍을 이루는가?

여는 괄호를 만날 때 스택에 푸시 하다, 닫는 괄호를 만날 때마다 팝을 해주면 쉽게 구현 할 수 있다.

2) 여러 개의 괄호가 중첩 될 경우 차례대로 처리되는가?

스택의 후입선출 제약을 이용하면 검사할 수 있다.

> **스택응용1 구현**

- 수식계산과 표기법 변환

> **표기법**

→ 중위 표기법 : a * b

→ 후위표기법 : a b +

→ 변환1 : A*(B+C) -> A B C + *

→ 변환2 : A * B+ C -> A B * C +

→ 후위 표기수식 계산시 3가지 원칙

1) 피연산자를 마나면 스택에 푸시

2) 연산자를 만나면 연산에 필요한개수만큼 피연산자를 스택에서 팝

3) 연산한 결과는 다시 스택에 푸시

→ 스택에 저장된 유일한 노드에서 최종계산 결과를 팝시키면 정상종료

→ 결과 노드가 없다면 수식에 오류

> **지금부터 피연산자와 연산자를 묶어서 요소 혹은 토큰이라 칭한다.**

> **후위 표기수식은 중위 표기와는 달리 수식을 따라 단계별로 토큰을 처리**

> **토큰타입선언**

→ 연산자들의 종류를 열거형타입으로 정의

→ 숫기의 각 단위인 토큰을 표현하기 위해 구조체 선언

→ 데이터 값과 연산자 종류를 멤버변수로 가짐

> **중위에서 후위표기로의 전환**

→ 기본규칙 : 피연산자를 만나면 바로 출력, 연산자를 만나면 일단 스택에 보관

→ 연산자 우선순위 : 스택내부와 스택 외부에서의 연산자 우선순위에 차이가 있다.

→ 연산자 "("는 스택안에서는 우선순위가 가장 낮은 연산자이다.

→ 중위 표기에서 후위표기로의 변환 과정 도중에 연산자 ")"를 만나게 되면 스택에서 여는 괄호를 만날 때까지 스택에 저장된 모든 연산자를 팝시켜야 한다.

→ 중위->후위표기 변환시 적용되는 5가지 규칙

1) 피연산자를 만나면 바로 출력
2) 연산자를 만나게 되면 일단 스택에 보관
3) 단, 스택보관중인 연산자 중에서 우선순위가 높은 연산자는 출력
4) (주의)스택의 내부와 외부에서의 연산자 우선순위는 다르다.
5) 닫는 괄호연산자")"를 만나게 되면, 스택에서 여는 괄호"("를 만날 때까지 스택에 저장된 모든 연산자들을 팝 시키고 이를 출력

> **스택응용2 구현**

- 미로찾기

> **입구와 출구가 주어진 미로에서 입구부터 출구까지의 경로를 찾는 문제**

> **스택의 노드가 위치정보를 저장하도록 디자인**

> **알고리즘**

→ 스택에 기존에 방문한 위치 정보를 차례로 저장하는 방식을 통해 경로를 찾는 알고리즘

→ 입구 – 출구 까지의 경로를 출력하기 위해서는 어떤 경로를 통해 탐색했는지 정보가 저장되어야 한다. → 위치정보의 저장에 스택을 이용

→ 특정방향으로 탐색을 계속 진행하다가 더 이상 진행 할 수 없을 시, 이전의 이정표에 해당하는 위치로 돌아와서 다른 방향으로 탐색

→ 제약사항 : 무한루프가 발생하지 않기 위해서 미로의 위치별로 기존에 방문했던 곳인지를 저장할 필요가 있다.

→ 정리

1) 미로찾기 알고리즘에서는 기존 위치를 저장하고 다른 방향으로 재탐색 하기 위해 스택을 이용한다.

2) 미로의 탐색단계마다 방문위치 정보와 이동방향 정보를 스택에 푸시한다.

3) 출구 탐색에 실패한다면 다시 이전 위치로 돌아오기 위해 스택에 팝을 실시한다.

> **스택응용3 구현**

큐

- 큐의 개념

> **선입선출(First in First Out) 특성을 지니는 선형자료구조.**

- 특징

> **선입선출 : 추가되는 자료를 차례대로 저장 후 추가된 순서대로 반환시킨다.**

> **선형구조 : 큐에 있는 모든 자료는 각각 자신의 앞에도 1개의 자료만 존재하고 뒤에도 1개의 자료만 존재한다.**

> **큐의 제일 앞은 프런트 혹은 전단이라 부르고 제일 뒤는 리어 혹은 후단이라고 한다.**

- 스택과의 비교

> **스택의 탑과 큐의 리어는 자료가 추가되는 리스트의 제일 끝이라는 점에서 공통점을 보인다.**

> **스택은 탑(첫 부분)에서 자료를 반환하는 반면 큐는 리어(끝 부분)에서 자료를 반환한다.**

- 인큐

> **새로운 자료를 큐에 추가하는 것**

> **큐의 최대자료수를 넘어 추가하면 오버플로우 현상이 발생한다.**

> **추가 시 큐의 리어가 가장 나중에 삽입된 자료를 가리킨다.**

- 디큐

> **큐에서 자료를 꺼내는 것**

> **큐의 프런트가 가리키는 자료를 반환한 이후 그 다음 자료를 가리킨다.**

> **한 개의 자료만 저장된 상태에서는 프런트와 리어가 모두 한 자료를 가리킨다.**

→ 의미하는 바

1) 해당 자료가 현재 큐에서 유일한 원소이다.

2) 현재 큐에 있는 자료 중 가장 오래된 자료이다.

3) 현재 큐에 있는 자료 중 가장 최근 자료이다.

→ 반환 시 프런트와 리어는 공백자료를 가리키게 된다.

> **큐에 아무런 자료가 저장되어 있지 않은 상태를 공백상태라 하며, 이 경우에 디큐 연산 시 언더플로우 현상이 발생한다.**

- 피크

> **스택과 마찬가지로 기존 자료를 제거하지 않고 반환만 하는 특성이 있다.**

> **큐의 프런트 자료에 접근하는 점에서는 디큐와 같지만, 실제 큐에서 자료를 반환 시키지는 않는다는 차이점이 존재한다.**

- 추상자료형

이름		입력	출력	설명

큐 생성	createQueue()	스택의 크기 n	큐 q	최대 n개의 원소를 가지는 공백(Empty) 큐 q를 생성
큐 삭제	deleteQueue()	큐 q	N/A	큐 제거(메모리 해제)
원소 추가 가능여부판단	isFull()	큐 q	TRUE/FALSE	큐의 원소 개수가 최대 원소 개수와 같은지를 반환. 배열 큐인 경우에만 의미가 있음
공백 큐 여부 판단	isEmpty()	큐 q	TRUE/FALSE	공백 큐인지 여부를 전달
인큐	Enqueue()	큐 q 원소 e	성공/실패 여부	큐의 리어에 새로운 원소를 추가
디큐	Dequeue()	큐 q	원소 e	큐의 프런트에 있는 원소를 제거한 뒤 이를 반환
피크	Peek()	큐 q	원소 e	큐의 프런트에 있는 원소를 반환 (큐에서 제거하지는 않음)

> **구현 방법에 따른 구분**

1) 배열 : 구현의 복잡도가 낮은 대신 고정된 큐의 크기로 인해 불필요한 메모리 사용량이 증가

2) 리스트 : 구현의 복잡도가 높은 대신 큐의 크기를 가변적으로 추가/삭제할 수 있기에 메모리 사용 효율 면에서는 우수함

- 배열 선형 큐

> **배열을 이용해 큐를 구현한 것**

> **선형 큐는 인큐/디큐를 반복함에 따라 배열에 비어 있는 곳이 있음에도 이를 인식 하지 못하는 경우가 발생한다. → 이는 비효율적인 메모리 사용으로 이어진다.**

> **보통 배열로 구현한 원형 큐 또는 연결리스트로 구현한 큐를 많이 사용**

> **선형큐와 원형 큐모두 같은 구조체를 사용하지만 함수 내부의 알고리즘에서 차이가 발생한다.**

> **배열은 동적할당으로 생성되며 큐 구조체에서는 이를 가리키는 포인터변수를 가지고 있다.**

> **큐에 저장된 현재 노드의 개수를 내부변수로 저장하고, 배열에서의 프런트와 리어 위치를 각각 내부변수로 저장한다.**

> **구현**

- 배열 원형 큐

> **선형 큐에 최대개수까지 인큐하고, 이후 디큐한후 다시 인큐를 하면 오버플로우 현상이 발생**

> **프런트 쪽의 빈노드가 있음에도 리어가 이동할 오른쪽에 빈 노드가 없기 때문에 넘침 현상이 발생한 것**

> **대안으로 배열을 이동시키는 방법을 생각 할 수는 있으나 배열의 크기에 비례해 시간복잡도가 O(n)만큼 증가한다. → 더욱 효과적인 방법이 필요**

> **배열의 오른쪽을 배열의 왼쪽과 연결시킨다면 효율적일 것**

> **리어의 오른쪽과 프런트의 왼쪽을 연결 시킨 것이 원형 큐**

◆ **리어와 프런트의 이동에 mod연산자를 이용하면 구현 가능**

> **인큐**

→ 리어 변수 증가시 : (rear + 1)%maxElementCount

> **디큐**

→ 프런트 변수 증가시 : (front +1)%maxElementCount

> **display함수**

→ mod연산자 때문에 rear의 값이 front보다 작을 수 있기 때문에 리어를 인덱스 계산에 쓰는 것이 아니라 프런트를 기준으로 계산한다.

> **구현**

- 연결리스트 큐

> **단순연결리스트의 제일 앞부분이 프런트, 제일 뒷부분이 리어**

> **새로운 자료 추가 시 리어에 추가하고, 기존 자료를 반환할 때는 가장 앞 프런트에 있는 노드를 제거한 뒤 이를 반환한다.**

> **각각의 노드는 자신의 앞, 뒤 노드와 1대1의 선형관계로 서로 연결되어 있다.**

> **자료의 방향은 프런트에서 리어 순서로 이어진다.**

> **배열큐와의 차이점**

→ 저장 노드 개수 초과로 인한 넘침현상이 발생하지 않는다는 장점이 있다.

→ 하지만 동적으로 메모리를 할당하기 때문에 구현 시 난이도가 있다.

> **구현시 주의점**

→ 연결큐의 제일 마지막인 리어노드의 경우 pLink값이 NULL이다.

> **인큐**

→ 기존의 연결리스트와 유사하게 이미 저장된 자료가 있는 지에 따라 인큐로직에 다소 차이가 있다.

1) 공백일때 : 기존의 프런트와 리어모두 NULL을 가리키고 있다. 이 상태에서 새로운 자료를 추가하면 프런트 노드와 리어노드가 모두 새로 추가된 노드를 가리키도록 수정해야 한다.

2) 공백이 아닐 때 : 새로추가되는 노드를 리어의 다음 노드로 설정한다. 그 다음 리어노드의 포인터를 새로추가된 노드를 가리키도록 재설정한다.

> **디큐와 피크**

→ 디큐를 통해 전달 받은 노드는 기존 큐의 연결리스트에서 제거된 노드이므로 전달받은 쪽(main)에서 노드의 메모리 해제(free)를 책임지지만, 피크는 큐에서 해당노드를 제거한 것이 아니기 때문에 전달받은 쪽에서의 해제에 대한 책임이 없다.

> **디큐**

→ 기존 연결리스트에서 제거하는 연산이 필요하기 때문에 조금 복잡하다.

1) 일반적인 경우 : 기존 큐의 프런트 노드가 연결리스트에서 제거되어 포인터 변수로 반환된다. 또한 기존 프런트노드의 다음 노드가 큐의 새로운 프런트 노드로 설정된다. 반환 되는 포인터 변수 또한 기존의 연결을 NULL로 초기화 시켜줘야 한다.

2) 노드가 1개 남은 경우 : 새로운 노드에 대한 처리는 일반적인 경우와 동일하나, 리어노드의 경우 마지막 남은 노드를 처리하는 과정이라 NULL을 가리키도록 해야 한다.

> **연결리스트 큐 구현**

- 연결리스트 덱

> **덱(Deque)이란?**

→ 두 개의 끝을 가지는 큐(Double-Ended Queue)

→ 양쪽 끝에서 자료의 삽입과 반환이 모두 가능한 선형자료구조

→ 이런 특성 때문에 큐와 스택의 기능을 합친 자료구조로도 설명될 수 있다.

> **덱과 큐, 스택 비교**

연산위치	연산	덱	스택	큐
앞, 프런트	추가	InsertFront	X	X
	반환	DeleteFront	X	Dequeue
뒤, 리어	추가	InsertRear	Push	Enqueue
	반환	DeleteRear	Pop	X

→ 스택의 두 연산 Push/Pop은 덱의 InsertRear/DeleteRear와 매칭된다.

→ 큐의 두 연산 Enqueue/Dequeue또한 InsertRear/DeleteFront로 매칭된다.

> **덱은 기존의 스택 혹은 큐에서 제공하지 않던 기능을 제공한다.**

→ InsertFront

> **덱은 스택과 큐를 일반화 시킨 포괄적인 자료구조로 해설 될 수 있다.**

→ A-Steal스케줄링 알고리즘과 같이 몇몇 알고리즘에서 덱의 이러한 포괄적인 기능이 요구되기도 한다.

> **추상자료형**

이름		Input	Output	설명
덱 생성	CreateDeque()	덱의 크기n	덱 d	최대 n개의 원소를 가질 수 있는 공백(Empty) 덱 d생성

덱 삭제	DeleteDeque()	덱 d	N/A	덱을 제거(메모리 해제)
원소추가 가능 여부 판단	isFull()	덱 d	TRUE/FALSE	덱의 원소 개수가 최대 언소 개수와 같은지를 반환
공백 판단	isEmpty()	덱 d	TRUE/FALSE	공백 덱인지를 전달
앞 추가	InsertFront()	덱 d 원소e	성공/실패여부	덱의 맨 앞(Front)에 새로운 원소를 추가
뒤 추가	InsertRear()	덱 d 원소e	성공/실패여부	덱의 맨 뒤(Rear)에 새로운 원소를 추가
앞 제거	DeleteFront()	덱 d	원소e	덱의 맨 앞(Front)에 있는 원소를 제거 한 뒤 반환
뒤 제거	DeleteRear()	덱 d	원소e	덱의 맨 뒤(Rear)에 있는 원소를 제거한 뒤 반환
앞 반환	PeekFront()	덱 d	원소e	덱의 맨 앞(Front)에 있는 원소를 반환(제거X)
뒤 반환	PeekRear()	덱 d	원소e	덱의 맨 뒤(Rear)의 원소를 반환(제거X)
덱 표시	displayDeque()	덱 d	N/A	덱에 저장된 원소를 화면에 표시

> **덱은 구현 방법에 따라 배열을 이용하거나 연결리스트를 이용하는 두가지 방법으로 구현 될 수 있음**

> **지금은 이중연결리스트를 이용해서 구현**

→ 이중연결리스트를 사용한면서 헤드포인터를 사용해 구현

> **선언**

→ 헤드포인터를 이용하기 때문에 두개의 해드포인터(Front, Rear)에 대한 변수를 가지고 있다.

→ "헤더노드"를 이용하는 이중연결 리스트는 임의의 위치에 있는 노드 pNode에 대해 항상 다음식이 성립한다.

→ pNode == pNode->pLLink->pRLink == pNode->pRLink->pLLink

→ 헤드 포인터를 이용하여 구현하는 이중 연결리스트는 위 식이 성립하지 않는다.

→ 따라서 헤드포인터를 이용하여 이중연결리스트를 구현하면, 노드의 추가와 삭제 시에 고려해야 할 경우의 수가 많아짐을 유념해야 한다.

> **InsertFront/Rear**

→ 덱의 상태가 공백인지에 따라 처리 과정에서 차이가 있다.

1) 공백일때 : 프런트노드와 제일 뒤쪽 리어노드가 모두 이 새로 추가된 노드를 가리키도록 한다.

2) 공백이 아닐 때 :

2-1) 새로운 노드가 기존 프런트 노드의 왼쪽 노드로 설정된다.

2-2) 새로운 노드의 오른쪽 노드는 기존의 프런트 노드가 차지한다.

2-3) 이후 새로 추가된 노드를 덱의 프런트 노드로 설정해 줘야 한다.

→ Rear의 경우 Front를 Rear로 대체하고 반대방향으로 연결해준다.

> **DeleteFront/Rear**

→ delete연산을 통해 전달되는 노드들은 사용이 끝난 후 반드시 메모리를 해제 시켜 줘야 한다.

→ 덱의 노드가 1개인지 1개이상(일반적인 경우)인지에 따라 차이가 발생한다.

1) 일반적인 경우

1-1) 반환되는 노드에 덱의 프런트노드를 대입한다.

1-2) 덱의 새로은 프런트 노드는 기존 프런트 노드의 오른쪽 노드로 대체한다.

1-3) 반환되는 노드의 오른쪽링크 정보는 NULL로 초기화 시킨다.

1-4) 덱의 새 프런트 노드는 왼쪽 링크정보를 NULL로 설정해준다.

2) 노드가 1개인경우 : 남은 노드 1개를 제거했을 때, 프런트 노드는 일반적인 경우와 차이기 없지만 리어 노드의 경우 NULL로 설정해 줘야 한다.

→ Rear의 경우 Front를 Rear로 대체하고 반대방향으로 연결해준다.

> **연결리스트 덱 구현**

- ## 큐의 응용_시뮬레이션

  > **시뮬레이션이란? : 현실 시스템에 대한 모델을 설계하고, 컴퓨터 기반의 실험을 통해, 시스템의 현상을 예측하는 일련의 과정**

  > **시뮬레이션을 구현하는 방법에는 많은 방법이 존재하지만, 시스템에서 시간을 일정하게 증가시키면서, 각각의 시각마다 발생하는 이벤트들을 큐에 저장하고 이를 처리하는 방식을 채택 할것이다.**

  > **하나의 노드를 고객으로 잡고, 노드의 멤버로 도착시간과 서비스시간을 정의한다.**

  > **큐를 통해 대기열, 서비스를 제공할 은행원, 고객의 방문순서를 표현한다.**

  > **시뮬레이션 시작시 도착시각이 미리 정의된 고객들은 시간 순서에 의해 고객의 방문순서를 기록한 큐에 미리 삽입되어 있다고 가정한다.**

  > **구현**

- ## 연습문제

  > **시뮬레이션에서 은행원이 1명 더 추가되었을 때를 구현하자**

  > **구현**

재귀호출

- ## 개념

  > **자기 자신을 호출하는 것**

  > **비선형 자료구조에 기반을 둔 다양한 알고리즘에 사용된다.**

  → 트리 혹은 그래프에 있는 모든 노드를 탐색하거나 다원탐색트리에서 노드를 삭제하거나 병합할 때 사용한다.

  → 문제 해결을 위한 접근방식으로 사용되기도 한다.

  ◆ 재귀적 접근방법은 일단 큰 문제를 여러 개의 작은 문제들로 나누는 것으로 시작

  ◆ 재귀적 접근 방법은 이처럼 하나의 큰 문제를 여러 개의 작은 문제들로 나누는 방

법인 것

- ◆ 이때 나누어진 작은 문제는 모두 같은 타입이어야 한다는 제약이 존재
- ◆ 종료조건 또한 재귀적 접근방법의 다른 특성으로 더 이상 나눌 수 없는 상태에 이를 때 재귀 호출을 종료하고 값을 반환한다.
- ◆ 재귀적 접근방법은 여러 개의 문제들로 쪼개어 해결한다는 점에서 분할정복의 한 방법으로 볼 수 있다.
- ◆ 단순히 자기자신을 호출 할 뿐 아니라 호출 될때마다 문제가 더 작아지는 특징이 존재

→ 성능상의 몇가지 제약사항 때문에 문제해결방법이 어느정도 고정된 후에는 비 재귀적인 방법(반복적인 방법)으로 다시 수정되기도 한다.

> **재귀호출의 내부적 구현**

→ 실제 운영체제에서는 스택을 이용해 재귀호출을 가능케 한다.

→ 호출 시 함수에서 사용되는 모든 지역변수와 전달인자를 저장하는 공간을 활성 레코드라 하는대, 운영체제에서는 하나의 함수가 호출되면, 함수별로 이러한 활성화 레코드에 함수관련 정보를 저장한다.

→ 재귀호출 종료시 활성레코드를 팝하여 처리하는 과정은 문맥변경(ContextSwitch)가 필요하며, 이는 운영체제가 기존에 수행되던 함수의 지역변수와 전달된 인자를 대신하여 새로운 함수의 지역변수와 인자를 적재하기 때문에, 이를 처리하는 과정에서 시간이 소요될 수 있다.

→ 따라서 같은 기능을 수행하는 함수의 경우 반복호출을 사용하는 함수가 재귀호출을 사용하는 함수보다 수행시간이 더 빠를 수 있다.

> **재귀호출의 종료조건**

→ 기본경우 혹은 최소한계라 부른다.

→ 만약 재귀호출에서 문제범위의 최소한계를 정하는 부분이 없다면 시스템 스택이 허용할 대까지 계속 순환적으로 호출 하다가 메모리 에러를 발생시키며 끝날 것이다.

> **재귀호출과 반복호출**

→ 재귀호출과 비교되는 방법으로 반복호출이 존재

→ 정해진 반복 횟수 혹은 완료조건을 만족할 때까지 반복적으로 계산을 실시하는 것이 반복 호출의 기본 개념이다.

→ 재귀호출이 반복호출과 비교하여 가지는 장점은 알고리즘의 간결성과 명화성이다.

→ 그러나 시스템 스택을 사용하므로 수행시간이 더 오래걸리며, 스택 메모리 문제가 발생할 수 있다.

> **재귀 호출 의 예**

→ 팩토리얼

→ 피보나치

→ 하노이탑

트리

- 트리의 개념

> **트리는 노드와 간선의 집합니다.**

→ 노드 : 일반적으로 모델링하려는 시스템의 객체를 나타낸다.

→ 간선 : 모델링한 객체들 사이의 부모-자식 관계를 정의한다.

> **노드의 구분**

구분	용어	내용
트리에서의 위치	루트(Root)노드	트리의 첫번째 노드
	단말(Leaf or Terminal)노드	자식노드가 없는 노드
	내부(Internal)노드	자식노드가 있는 노드
노드사이의 관계	부모(parent)노드	부모노드와 자식노드는 간선으로 연결되어 있음
	자식(Child)노드	
	선조(Ancestor)노드	루트 노드로부터 부모노드까지의 경로상에 있는 모든 노드
	후손(Descendant)노드	특정 노드의 아래에 있는 모든 노드
	형제(Sibling)노드	같은 부모노드의 자식노드

> **노드의 속성**

용어	내용

레벨(Level)	루트 노드로부터의 거리
높이(Height)	루트노드부터 가장 먼 거리에 있는 자식 노드 높이에 1을 더한 값
차수(Degree)	한 노드가 가지는 자식 노드의 수

> **트리의 특징**

→ 계층구조로 자료를 저장

◆ 계층구조란 트리를 구성하는 노드가 부모-자식관계라는 의미 → 특정부모 노드 하나에 여러 개의 자식 노드들이 연결되는 구조

→ 계층구조(하나의 노드에 여러 자식노드가 연결된 구조)인 점에서 비선형 자료임을 알 수 있다.

→ 노드 하나가 여러 자식을 가질 수 는 있지만, 반대로 각 노드의 부모노드는 모두 1개라는 점을 주의해야 한다.

> **트리의 기타 용어**

→ 서브트리 : 트리에 속한 노드들의 부분집합을 서브트리라 한다.

→ 포리스트(Forest) : 트리의 집합 → 여러 개의 루트 노드가 존재하기도 한다.

> **일반적인 트리의 경우 자식노드에 개수제한이 없는 반면, 최대 2개까지 제한을 가진 트리가 존재 → 이진트리**

- 이진트리

> **이진트리란?**

→ 모든 노드의 차수가 2이하인 트리를 말한다.

> **특성**

→ 이진트리의 각 노드는 3가지 경우가 존재한다.

1) 차수가 2(왼쪽, 오른쪽 자식 모두 가지는 경우)이거나

2) 차수가 1(왼쪽 오른쪽 중 하나만 가지는 경우)이거나

3) 차수가 0(단말노드인 경우)

→ 이진트리의 모든 서브트리들은 이진트리이다.

◆ n개의 노드를 가지는 이진트리는 모두 (n-1)개의 간선을 가지는데, 루트노드를 제외

한 나머지 노드들이 모두 부모-자식 관계에 의해 간선이 있기 때문이다.

◆ 정리하면 모든 노드개수에서 루트노드 1개를 제외한 수만큼의 간선이 존재한다.

◆ n-1개 보다 작은 간선을 가진다면 트리가 아니라 포리스트인 경우이다.

> **이진트리의 종류**

→ 트리의 형태는 레벨과 노드 수에 따라 결정된다.

1) 포화이진트리 : 모든레벨의 노드가 꽉 차있는 형태의 이진트리로 노드개수의 일반식은 $n = 2^h-1$이다.

2) 완전이진트리 : 노드가 왼쪽부터 차례로 채워진, 중간에 빈 노드가 없는 트리로 노드개수는 $n < 2^h-1$이다.

3) 편향이진트리 : 같은 높이의 이진트리 중에서 최소개수의 노드개수를 가지면서 왼쪽 혹은 오른쪽 서브트리만을 가지는 이진트리로 노드의 최대개수 범위는 $h \leq n \leq (2^h - 1)$가 된다.

> **이진트리의 추상자료형**

이름		Input	Output	설명
이진트리생성	makeBinTree()	원소e	이진트리bt	원소 e를 데이터로 갖는 루트노드로 구성된 이진트리bt를 생성
루트노드 반환	getRootNode()	이진트리 bt	루트노드rn	이진트리bt의 루트노드 반환
왼쪽 자식노드 추가	insertLeftChildNode()	부모노드 pt 원소e	생성된 노드 n	부모 노드pt의 오른쪽 자식 노드 n추가
오른쪽 자식 노드 추가	insertRightChildNode()	부모노드 pt 원소e	생성된 노드 n	부모노드pt의 왼쪽 자식 노드 n추가
왼쪽 자식 노드 반환	getLeftChildNode()	노드n	왼쪽 자식노드 ln	노드n의 오른쪽 자식노드 반환
오른쪽 자식 노드 반환	getRightChildNode()	노드n	오른쪽 자식 노드 rn	노드n의 왼쪽 자식노드 반환

이진트리루트노드의 데이터	getData()	이진트리 bt	원소e	이진트리bt의 루트노드 데이터를 반환
이진트리 삭제	deleteBinTree()	이진트리 bt		이진트리를 제거(메모리 해제)

> **배열을 이용한 이진트리 구현**

→ 배열을 이용한 트리는 노드의 번호가 인덱스 역할을 하는 점이 중요

> **특징**

→ 장점 : 배열 인덱스를 사용하여 노드접근이 편리

→ 단점 : 편향이진트리와 같이 빈 노드가 많을 경우에 메모리 낭비가 심하다.

- 높이가 h인 트리에 대해서는 미리 최대 저장노드개수보다 한 개 더 많게($2^h - 1 + 1$) 메모리를 할당 해야 한다.

→ 트리의 크기가 증가하는 경우 필요한 메모리 양이 급속히 증가하게 된다.

> **포인터를 이용한 이진트리 구현**

→ 장점 : 필요한 만큼만 메모리를 할당하기 때문에 메모리 효율성 측면에서 우수하다.

→ 단점 : 포인터를 이용해 구현하기 때문에 노드의 탐색과 메모리 관리측면에서 구현이 다소 복잡하다.

> **포인터를 이용한 구현**