



게임 자료구조와 알고리즘

-CHAPTER9-

SOULSEEK



목차

1. 트리(Tree)

2. 우선순위 큐(Priority Queue)와 힙(Heap)



트리(TREE)

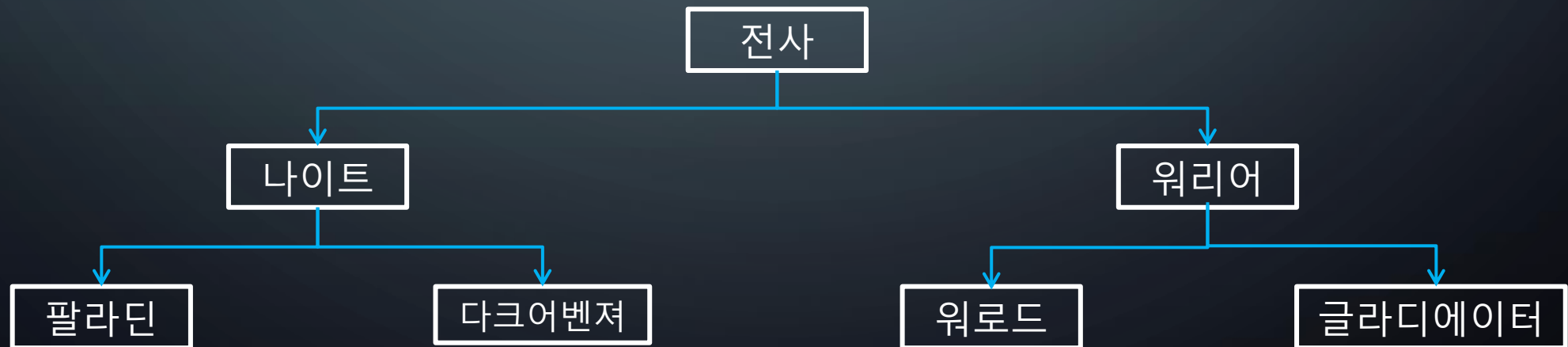
1. 트리(TREE)

- 계층적 관계를 표현한 자료구조
- 저장과 반환 + 표현

테크 "트리"



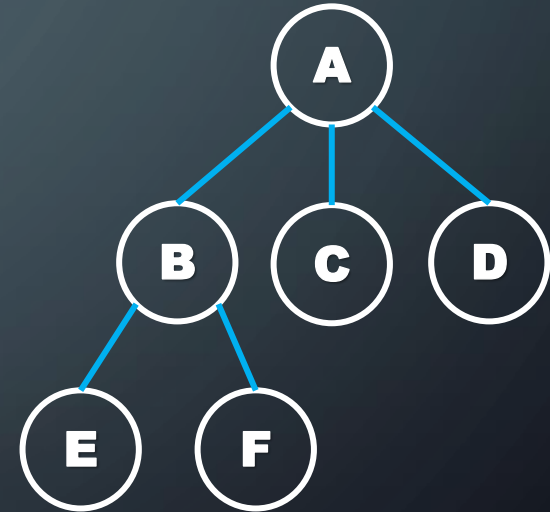
전직 "트리"



1. 트리(TREE)

트리 관련 용어

- **노드 : Node**
 - 트리의 구성요소에 해당하는 **A, B, C, D, E, F**와 같은 요소
- **간선 : Edge**
 - 노드와 노드를 연결하는 연결선
- **루트 노드 : Root Node**
 - 트리 구조에서 최상위에 존재하는 **A**와 같은 노드
- **단말 노드 : Terminal Node**
 - 아래로 또 다른 노드가 연결되어 있지 않은 **E, F, C, D**와 같은 노드
- **내부 노드 : Internal Node**
 - 단말 노드를 제외한 모든 노드로 **A, B**와 같은 노드



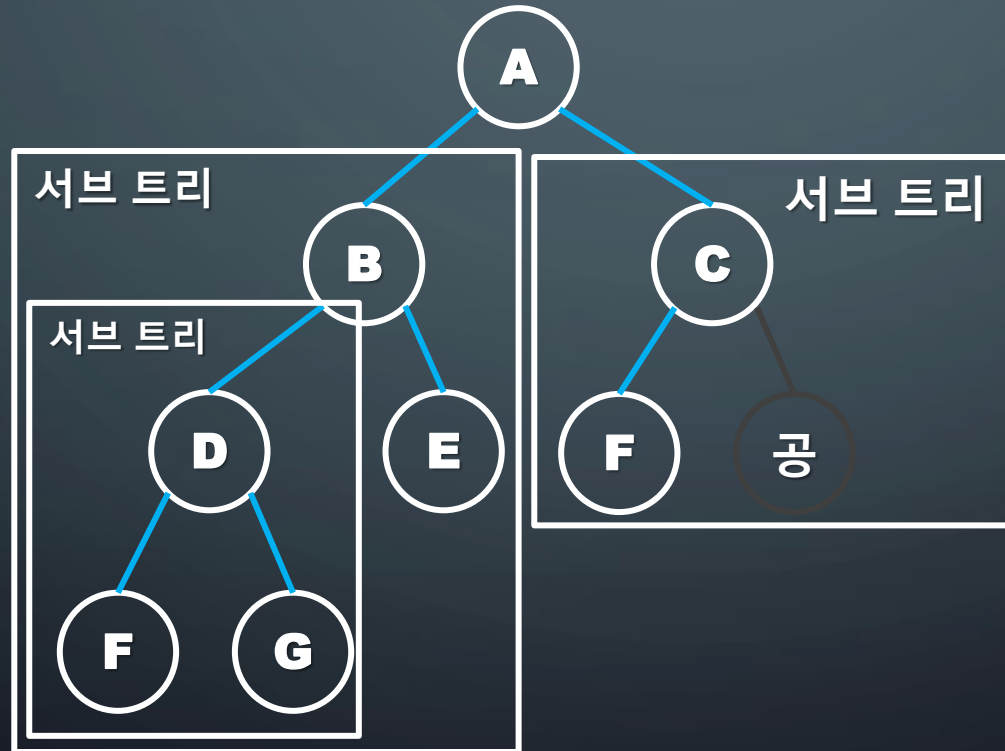
서로부모, 자식, 형제, 조상, 후손의 관계로 명명하기도 한다.

- 노드 **A**는 노드 **B, C, D**의 부모 노드다.
- 노드 **B, C, D**는 노드 **A**의 자식 노드다.
- 노드 **B, C, D**는 부모 노드가 같으므로, 서로 형제 노드다.
- 노드 **B, C, D, E, F**는 모두 노드 **A**의 후손 노드다.

1. 트리(TREE)

- 메인 트리가 아닌 자식이 가지고 있는 또 다른 트리를 서브 트리라고 한다.
- 자식이 두 개씩 달린 트리를 이진 트리라고 한다.
- 노드가 위치할 수 있는 곳에 노드가 존재하지 않는다면, 공집합 노드가 존재하는 것으로 생각한다. 이 공집합을 포함해서 “이진 트리”노드인지 함께 판단한다.

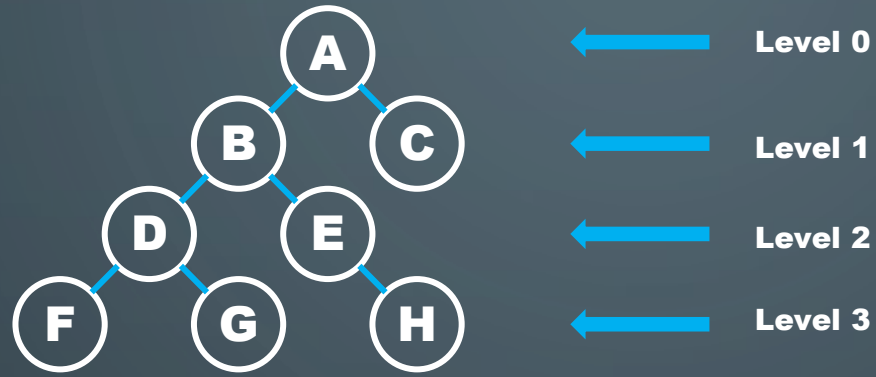
이진 트리



1. 트리(TREE)

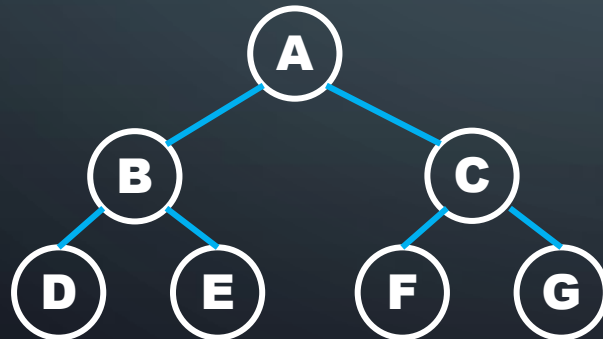
레벨과 높이

- **Tree**에서 각 층별로 숫자를 매겨서 이를 **Tree**의 “레벨”이라고 말한다.
- **Tree**의 최고 레벨을 가리켜 “높이”라고 말한다.

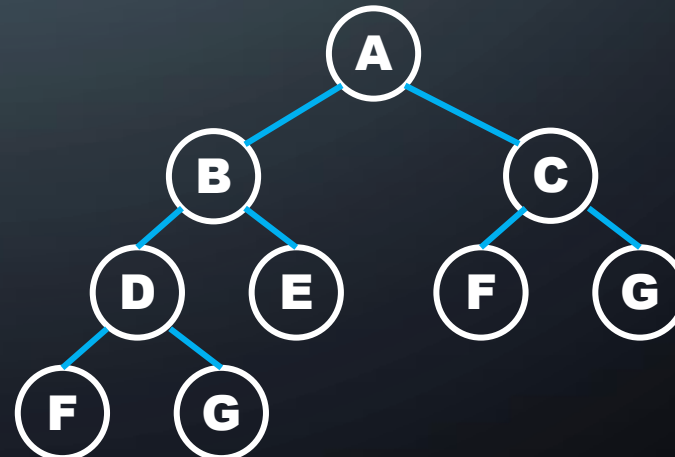


- 차곡차곡 왼쪽에서 오른쪽으로 정렬되어진 이진 트리를 포화, 완전 이진 트리라고 한다.

포화 이진 트리



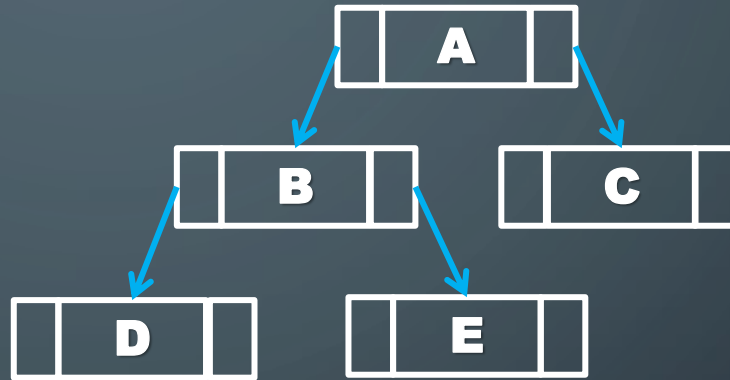
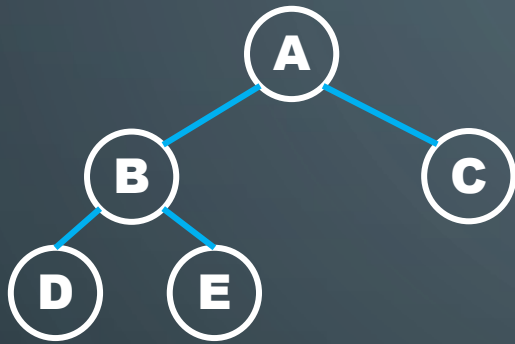
완전 이진 트리



1. 트리(TREE)

이진 트리의 구현

- 배열 기반, 연결 리스트 기반 모두 고려할 수 있다.
- 연결 리스트와 참조 형태가 비슷하기 때문에 연결 리스트를 선호한다.



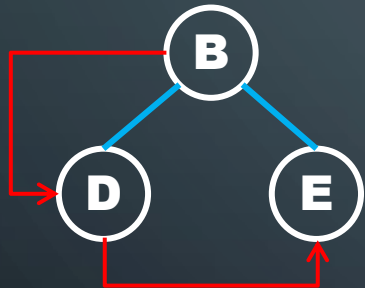
```
typedef struct _bTreeNode
{
    BTDData data;
    struct _bTreeNode * left;
    struct _bTreeNode * right;
} BTreeNode;
```

- 이진 트리를 표현하는 구조체는 존재하지 않는다.
- 공집합 노드가 이미 존재 하고 있는 상태이다.

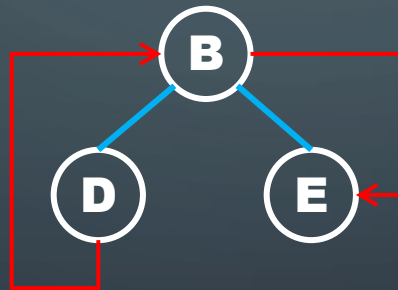
1. 트리(TREE)

이진 트리의 탐색 - 순회라고 부른다.

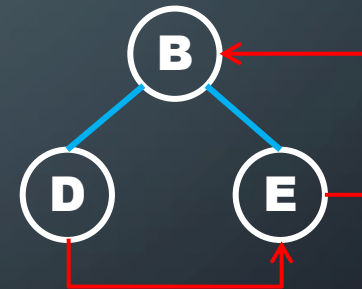
- 전위 순회 - 루트 노드를 먼저.
- 중위 순회 - 루트 노드를 중간에.
- 후위 순회 - 루트 노드를 마지막에.



전위 순회



중위 순회

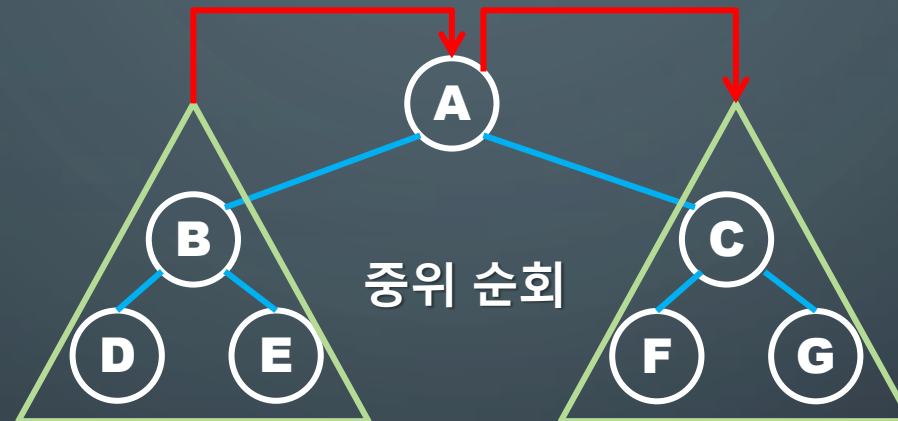


후위 순회

1. 트리(TREE)

높이가 Level2 이상일 때는?

- 재귀 호출을 이용해서 구현하면 해결 할 수 있다.
- 서브 로드 -> 루트로드 -> 서브로드



```
Void InorderTraverse(BTreeNode* bt)
{
    InorderTraverse(bt->left);
    printf("%d \n", bt->data);
    InorderTraverse(bt->right);
}
```

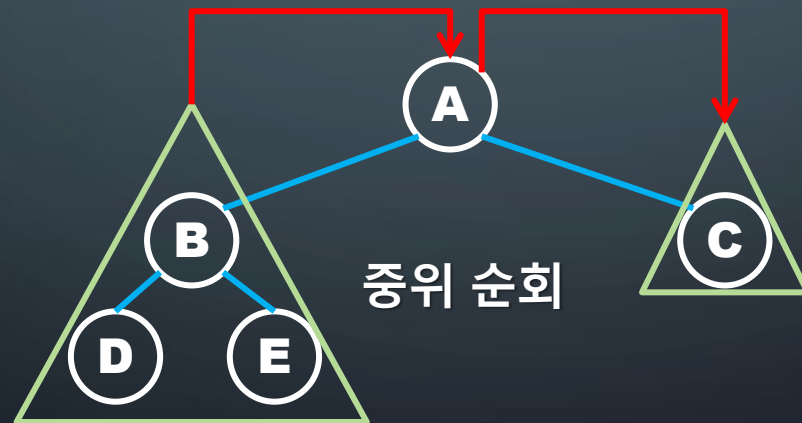
// 이진 트리 전체를

// 1단계 왼쪽 서브 트리의 순회

// 2단계 루트 노드의 방문

// 3단계 오른쪽 서브 트리의 순회

```
void InorderTraverse(BTreeNode* bt)  
{  
    if(bt == NULL)  
        return;  
  
    InorderTraver(bt->left);  
    printf(“%d, \n“, bt->data);  
    InorderTraver(bt->right);  
}
```



재귀 탈출이 필요!

1. 트리(TREE)

노드의 방문을 자유롭게!

- 데이터를 출력하는 것만이 방문의 전부가 아니기 때문에.
- 노드의 방문 목적은 상황에 따라 많이 달라진다.
- 함수 포인터를 이용하면 이를 극대화 할 수 있다.

```
typedef void VisitFuncPtr(BTData data); //방문 목적을 전달받을 함수 포인터
```

```
void InorderTraverse(BTreeNode* bt, VisitFuncPtr action)
```

```
{
```

```
    if(bt == NULL)  
        return;
```

```
    InorderTraverse(bt->left, action);
```

```
    action(bt->data); //노드의 방문.
```

```
    InorderTraverse(bt->right, action);
```

```
}
```

1. 트리(TREE)

학습과제

소멸과 관련된 함수를 제공하지 않았다 이진 트리에서 소멸과 관련된 함수를 작성하자.

```
void DeleteTree(BTreeNode* bt);
```

```
Int main(void)  
{  
    BTreeNode* bt1 = MakeBTreeNode();  
  
    DeleteTree(bt1);  
}
```

우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

- 큐의 단순 확장 개념이 아니다.
- 들어간 순서와 상관없이 우선순위가 높은 데이터가 먼저 나온다.
- 우선순위에 대한 표기가 필요하다.
- 배열, 연결 리스트, 힙을 이용하는 방법이 존재 한다.

배열과 연결리스트를 기반으로..

- 데이터 우선순위로 정렬을 해서 앞쪽부터 위치시킨다.
- 데이터를 삽입 및 삭제하는 과정에서 한 칸씩 밀거나 당기는 연산을 수반해야 한다.
- 삽입위치를 찾기 위해서 모든 데이터와 비교 해야 한다.

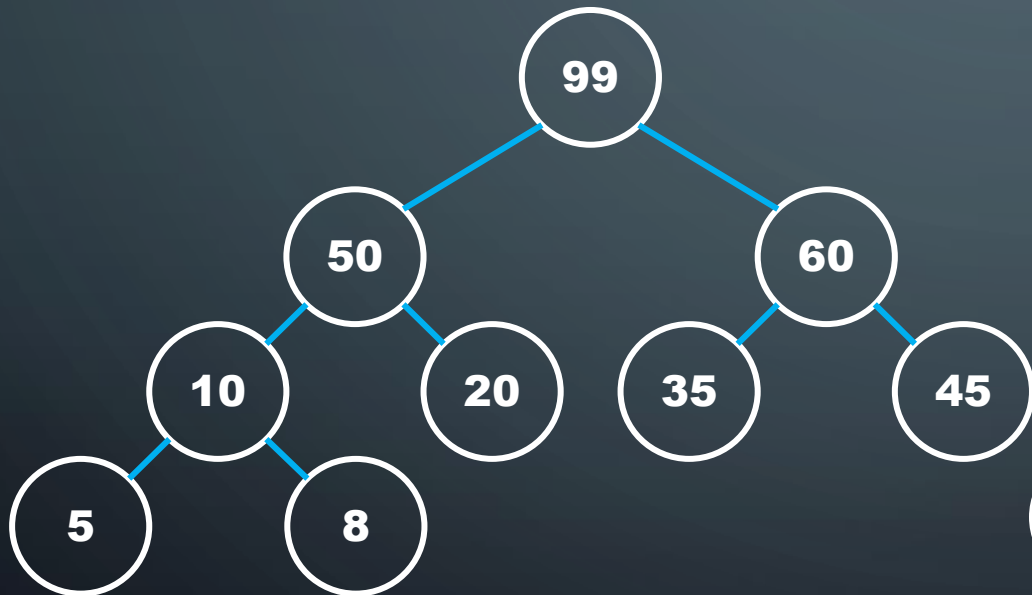
힙을 이용한 우선순위 큐

- 힙은 완전 이진 트리 이다.
- 모든 노드에 저장된 우선순위의 조건이 자식 노드의 조건보다 루트 노드의 조건이 우선시 되어야 한다.

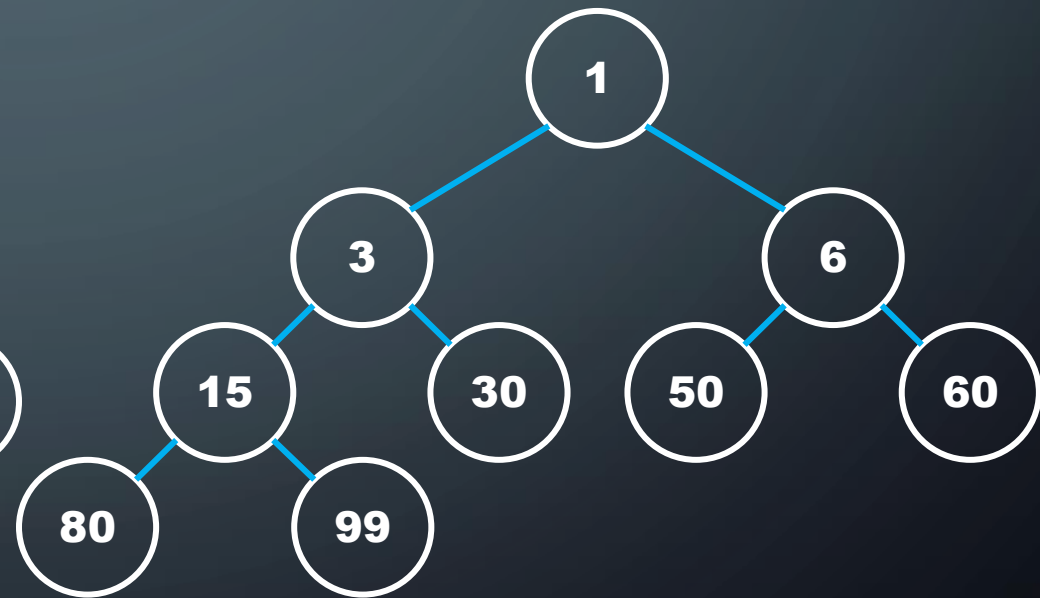
2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

- 루트 노드로 올라갈 수록 **저장된 값(우선순위가 될 수도 있는)이 커지는** 완전 이진 트리를 **최대 힙**이라 한다.
- 루트 노드로 올라갈 수록 **저장된 값(우선순위가 될 수도 있는)이 작아지는** 완전 이진 트리를 **최소 힙**이라 한다.

최대 힙(MAX)

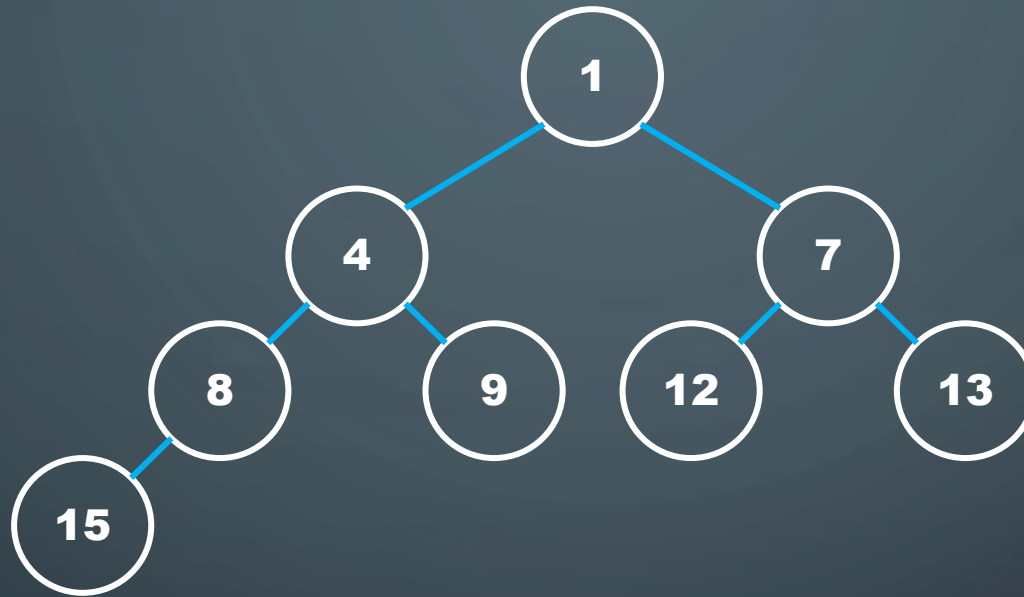


최소 힙(MIN)



2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

힙의 구현



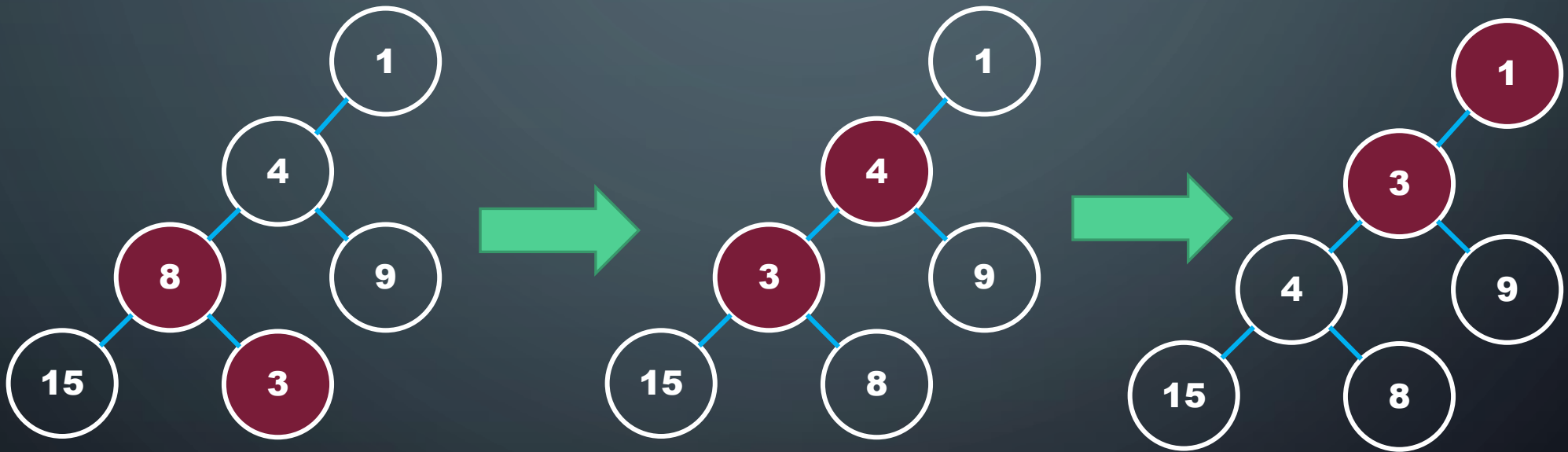
1. 최소 힙(MIN)이라 가정
2. 데이터가 정수이면서 작은 수를 우선순위로 가지는 조건임을 가정

자식 노드 데이터의 우선순위 \leq 부모 노드 데이터의 우선순위

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

힙에 데이터 삽입

- **우선순위의 조건대로 삽입**이 되어야한다.
- 새로운 데이터가 추가 될 때 우선순위가 낮다고 가정하고 마지막 위치에 저장한다. 그리고 부모 노드와 비교하여 위치가 바뀌어야 한다면 바꿔준다. 그렇게 계속해서 부모의 노드와 우선순위를 비교해서 처리해서 제대로 된 자리에 위치하게 해준다.



3이라는 데이터가 추가 된다고 가정하면
3이 마지막으로 그리고 그 부모와 비교

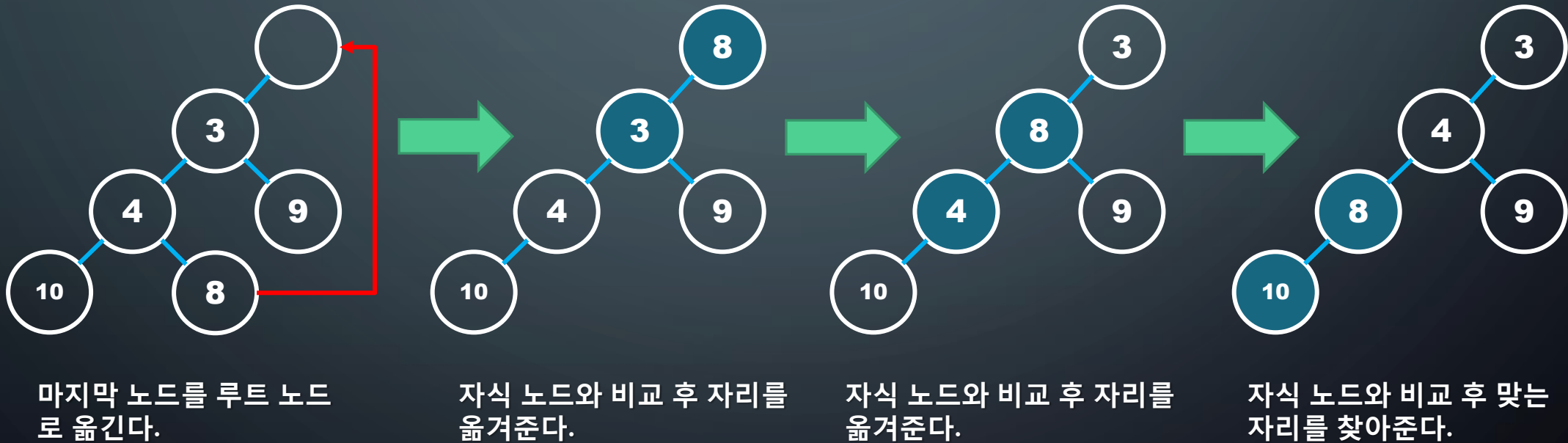
비교에 의해 자리를 정해준다.
다시 부모와 비교를 해준다.

비교에 의해 자리를 정해준다.
다시 부모와 비교를 해준다.

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

힙에서 데이터 삭제

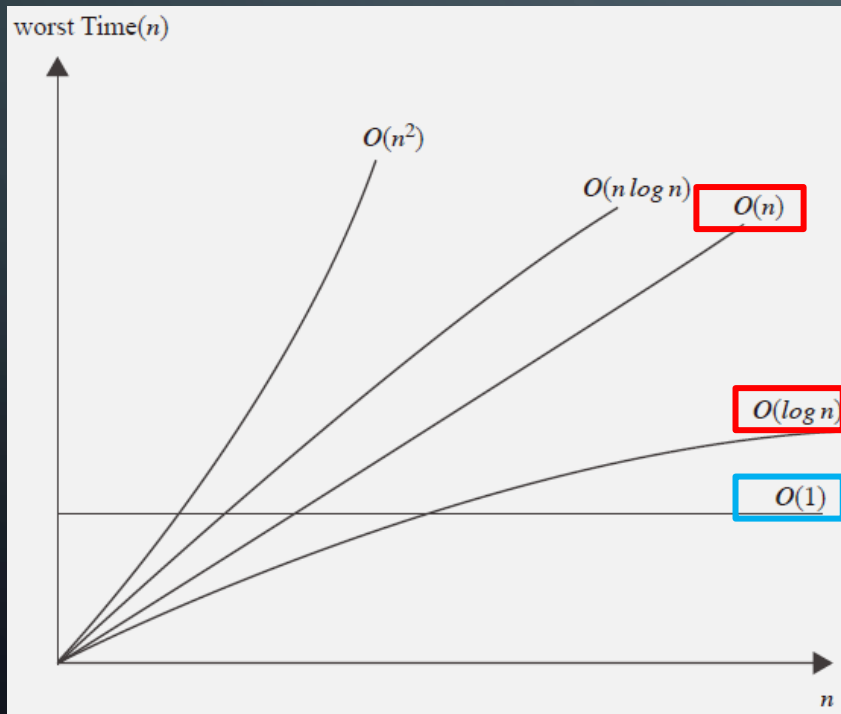
- 우선순위대로 데이터가 삭제된다. – 루트 노드부터 삭제된다.
- 마지막 노드를 루트 노드의 자리로 옮긴 다음에, 자식 노드와의 비교를 통해서 제자리를 찾아가게 한다.



2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

배열, 연결 리스트, 힙의 시간 복잡도 비교

- 배열과 연결 리스트 모두 모든 데이터와 우선 순위를 비교해야 하므로 **$O(n)$** 이 된다.
- 배열과 연결 리스트 모두 삭제 시 제일 앞의 데이터부터 삭제하므로 **$O(1)$** 이 된다.
- 힙은 비교연산이 부모 노드와 자식 노드 사이에만 일어나기 때문에 삽입, 삭제 모두 트리의 높이에 해당하는 수만큼만 비교연산을 진행하면 되기때문에 **$O(\log_2 n)$** 이 된다.



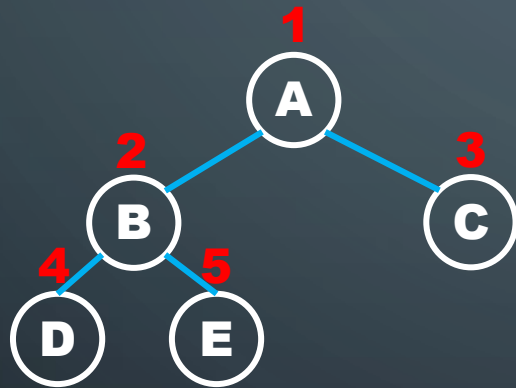
Big - O에 의해서 두 값의 비교가 차이가 많이 난다.

n	순차 탐색 연산횟수	이진 탐색 연산횟수
500	500	9
5,000	5,000	13
50,000	50,000	16

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

힙의 구현

- 연결리스트로 힙을 구현하면 마지막 위치를 특정하기 어렵다.
- 힙은 노드에 **우선순위를 추가해줘야 하는 부분이 배열 인덱스로 활용**하면 되기 때문에 배열을 기반으로 구현한다.



[0]	
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	.
[7]	.

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

왼쪽, 오른쪽 자식 노드의 인덱스와 부모 노드의 인덱스 구하기

- 이진 트리라면 레벨이 증가함에 따라서 추가할 수 있는 자식 노드의 수가 두 배씩 증가하는 구조
- 배수를 나누고 곱하는 방식으로 부모 노드와 자식 노드의 인덱스를 구할 수 있다.

이진 노드라면..

- 왼쪽 자식 노드의 인덱스 값 = 부모 노드의 인덱스 값 * **2**
- 오른쪽 자식 노드의 인덱스 값 = 부모 노드의 인덱스 값 * **2 + 1**
- 부모 노드의 인덱스 값 = 자식 노드의 인덱스 값 / **2**

힙에 저장될 데이터의 모델을 정의하는 구조체

```
typedef struct _heapElem
{
    Priority pr; // 값이 작을수록 높은 우선순위
    HData data;
} HeapElem;
```

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

```
#define HEAP_LEN100
```

```
typedef char HData;  
typedef int Priority;
```

```
typedef struct _heapElem  
{  
    Priority pr;  
    HData data;  
} HeapElem;
```

```
typedef struct _heap  
{  
    int numOfData;  
    HeapElem heapArr[HEAP_LEN];  
} Heap;
```

```
/** Heap 관련 연산들 ***/
```

```
void HeapInit(Heap * ph);
```

```
int HIsEmpty(Heap * ph);
```

// 힙의 초기화

// 힙이 비었는지 확인

```
void Hinsert(Heap * ph, Hdata data, Priority pr); // 힙에 데이터 삽입
```

```
HData HDelete(Heap * ph); // 힙의 데이터 삭제
```

- 힙은 완전 이진 트리이다.
- 힙의 구현은 배열을 기반으로 하며 인덱스가 0인 요소는 비워 둔다.
- 따라서 힙에 저장된 노드의 개수와 마지막 노드의 고유번호는 일치한다.
- 노드의 고유번호가 노드가 저장되는 배열의 인덱스 값이 된다.
- 우선순위를 나타내는 정수 값이 작을수록 높은 우선순위를 나타낸다고 가정한다.

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

힙 연산을 도와주는 함수들

//부모 노드의 인덱스 값을 반환

```
int GetParentIDX(int idx)
```

```
{  
    return idx / 2;  
}
```

//왼쪽 자식 노드의 인덱스 값을 반환

```
int GetLChildIDX(int idx)
```

```
{  
    return idx * 2;  
}
```

//오른쪽 자식 노드의 인덱스 값을 반환

```
int GetRChildIDX(int idx)
```

```
{  
    return GetLChildIDX(idx) + 1;  
}
```

//두 개의 자식 노드 중 높은 우선순위의 자식 노드 인덱스 값 반환

```
int GetHiPriChildIDX(Heap* ph, int idx)
```

```
{  
    생략...  
}
```


2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

HDelete를 도와주는 핵심 함수

- **GetHiPriChildIDX** 함수에 노드의 인덱스 값을 전달하면, 이 노드의 두 자식 중에서 우선순위가 높은 것의 인덱스 값을 반환한다.
- **GetHiPriChildIDX** 함수는 인자로 전달된 노드에 자식 노드가 없으면 **0**을 반환하고, 자식 노드가 하나인 경우에는 그 노드의 인덱스 값을 반환한다.

```
int GetHiPriChildIDX(Heap * ph, int idx)
{
    if (GetLChildIDX(idx) > ph->numOfData) // 자식 노드가 없다면
    {
        return 0;
    }
    else if (GetLChildIDX(idx) == ph->numOfData) // 왼쪽 자식 노드가 마지막 노드라면
    {
        return GetLChildIDX(idx);
    }
    else // 자식 노드가 둘 다 존재하므로 왼쪽 자식 노드와 오른쪽 자식 노드의 우선순위를 비교
    {
        // 오른쪽 자식 노드의 우선순위가 높다면,
        if (ph->heapArr[GetLChildIDX(idx)].pr > ph->heapArr[GetRChildIDX(idx)].pr)
            return GetRChildIDX(idx);
        else
            return GetLChildIDX(idx);
    }
}
```

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

- 힙은 완전 이진 트리이기 때문에 오른쪽 자식만 존재할 수 없다. 따라서 왼쪽 자식이 없다면 오른쪽 자식도 존재하지 않는 것이다.
- 부모 인덱스에 자식이 둘이라면 $\text{인덱스} * 2$ 만큼의 데이터 수 이어야 한다.

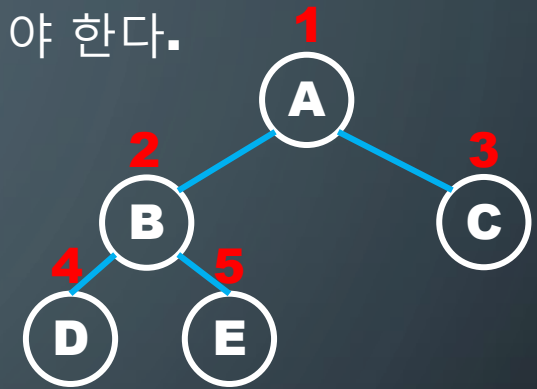
// 자식이 존재 하지 않는 경우의 코딩.

```
if(GetLGChildIDX(idx) > ph->numOfData)
    return 0;
```

- 오른쪽 자식은 왼쪽 자식 + 1의 데이터 수가 된다.
- 왼쪽 자식만 있다면 2의 배수가 된다.
- 데이터 수와 정확히 일치하는 왼쪽 자식은 하나이자 마지막 노드이다.

// 자식 노드가 하나인 경우의 코딩

```
else if(GetLChildIDX(idx) == ph->numOfData)
    return GetLChildIDX(idx);
```



2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

- 가장 우선순위가 높은 루트 노드부터 차례로

```
HData HDelete(Heap * ph)
```

```
{
```

```
    HData retData = (ph->heapArr[1]).data; // 삭제할 데이터 임시 저장
```

```
    HeapElem lastElem = ph->heapArr[ph->numOfData]; // 힙의 마지막 노드 저장
```

```
    // 아래의 변수 parentIdx에는 마지막 노드가 저장될 위치정보가 담긴다.
```

```
    Int parentIdx = 1; // 루트 노드의 Index
```

```
    Int childIdx;
```

```
    // 루트 노드의 우선순위가 높은 자식 노드를 시작으로 반복문 시작.
```

```
    While (childIdx = GetHiPriChildIDX(ph, parentIdx))
```

```
    {
```

```
        If (lastElem.pr <= ph->heapArr[childIdx].pr) // 마지막 노드와 우선순위 비교
```

```
            break; // 마지막 노드의 우선순위가 높으면 반복문 탈출
```

```
        //마지막 노드보다 우선순위 높으니, 비교대상 노드의 위치를 한 레벨 올린다.
```

```
        ph->heapArr[parentIdx] = ph->heapArr[childIdx];
```

```
        parentIdx = childIdx; // 마지막 노드가 저장될 위치 정보를 한 레벨 내림.
```

```
    } // 반복문 탈출하면 parentIdx에는 마지막 노드의 위치정보가 저장된다.
```

```
    ph->heapArr[parentIdx] = lastElem; // 마지막 노드 최종 저장
```

```
    ph->numOfData -= 1;
```

```
    return retData;
```

```
}
```

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

- 마지막 위치로 새로운 데이터를 추가하고 우선순위를 비교하면서 위치를 찾아간다.

```
void HInsert(Heap * ph, HData data, Priority pr)
```

```
{  
    int idx = ph->numOfData + 1;      // 새 노드가 저장될 인덱스 값을 idx에 저장  
    HeapElem nelem = { pr, data };    // 새 노드의 생성 및 초기화  
  
    // 새 노드가 저장될 위치가 루트 노드의 위치가 아니라면 while문 반복  
    While (idx != 1)  
    {  
        //새 노드와 부모 노드의 우선순위 비교  
        if (pr < (ph->heapArr[GetParentIDX(idx)].pr)) // 새 노드의 우선순위 높다면  
        {  
            // 부모 노드를 한 레벨 내림, 실제로 내림  
            ph->heapArr[idx] = ph->heapArr[GetParentIDX(idx)];  
  
            // 새 노드를 한 레벨 올림, 실제로 올리지 않고 인덱스 값만 갱신  
            idx = GetParentIDX(idx);  
        }  
        else // 새 노드의 우선순위가 높지 않다면  
            break;  
    }  
  
    ph->heapArr[idx] = nelem; // 새 노드를 배열에 저장.  
    ph->numOfData += 1;  
}
```

The background is a dark blue gradient with a large, faint, light blue circle in the center. The corners are decorated with white circuit-like lines and small circles, resembling a stylized electronic board.

SimpleHeap을 확인해보자

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

힙 구현의 개선

```
typedef struct _heapElem
{
    Priority pr;
    HData data;
} HeapElem;
```

```
void HInsert(Heap * ph, HData data, Priority pr);
```

- 데이터를 입력하기 전에 우선순위를 정해줘야 하는 상황.
- 데이터의 우선순위를 직접 기입해야 하는 불편한 상황.

```
typedef struct _heap
{
    PriorityComp * comp;
    int numOfData;
    HData heapArr[HEAP_LEN];
} Heap;
```

```
void HeapInit(Heap * ph, PriorityComp pc);
```

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

UsefulHeap을 확인해보자

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

우선순위 큐 자료구조의 ADT

- **void PQueueInit(PQueue* ppq, PriorityComp pc);**
 - 우선순위 큐의 초상화를 진행한다.
 - 우선순위 큐 생성 후 제일 먼저 호출되어야 하는 함수
- **int PQIsEmpty(PQueue* ppq);**
 - 우선순위 큐가 빈 경우 **TRUE(1)**을, 그렇지 않은 경우 **FALSE(0)**을 반환한다.
- **void Penqueue(PQueue* ppq, PQData data);**
 - 우선순위 큐에 데이터를 저장한다. 매개변수 **data**로 전달된 값을 저장한다.
- **PQData Pdequeue(PQueue* ppq);**
 - 우선순위가 가장 높은 데이터를 삭제한다.
 - 삭제된 데이터는 반환된다.
 - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

2. 우선순위 큐(PRIORITY QUEUE)와 힙(HEAP)

학습과제

- **PriorityQueue**를 확인해 보자.
- 우선순위 큐를 이용해 다수의 문자열을 저장하고, 저장된 문자열을 꺼내어 출력하는 프로그램을 작성해보자.

조건1. 힙에 저장되는 문자열은 길이가 짧을수록 우선순위가 높다고 가정한다.