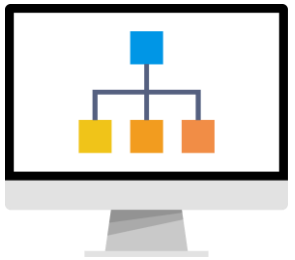


# Data Structure & Algorithm
# 자료구조 및 알고리즘

## 16. 우선순위 큐와 힙 (Priority queue and Heap)

# 우선순위 큐의 이해

# 우선순위 큐와 우선순위의 이해

일반 큐의 두 가지 연산

- enqueue 큐에 데이터를 삽입하는 행위
- dequeue 큐에서 데이터를 꺼내는 행위

Enqueue된 순서대로 dequeue 연산이 진행된다.

우선순위 큐의 두 가지 연산

- enqueue 우선순위 큐에 데이터를 삽입하는 행위
- dequeue 우선순위 큐에서 데이터를 꺼내는 행위

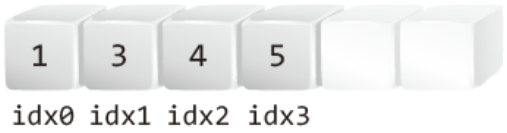
Enqueue된 순서에 상관 없이 우선순위대로 dequeue 연산이 진행된다.

데이터 별 우선순위의 비교기준은 프로그래머가 결정할 몫이다! 따라서 우선순위 큐 자료구조를 활용하는 프로그래머가 직접 우선순위 비교기준을 결정할 수 있도록 구현이 되어야 한다.

# 우선순위 큐의 구현 방법

우선순위 큐를 구현하는 세 가지 방법

- 배열을 기반으로 구현하는 방법
- 연결 리스트를 기반으로 구현하는 방법
- 힙(heap)을 이용하는 방법



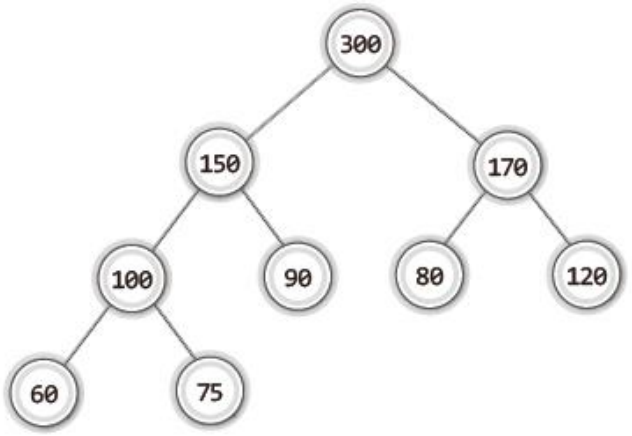

두 가지 방법 모두 최악의 경우 새 데이터의 위치를 찾기 위해서 기존에 저장된 모든 데이터와 비교를 진행해야 한다.


▶ [그림 09-1: 단순 배열과 연결 리스트 기반의 우선순위 큐 모델]

우선순위에 알맞은 위치를 찾아서 데이터를 저장하는 방식

O(N) 시간이 필요하다.

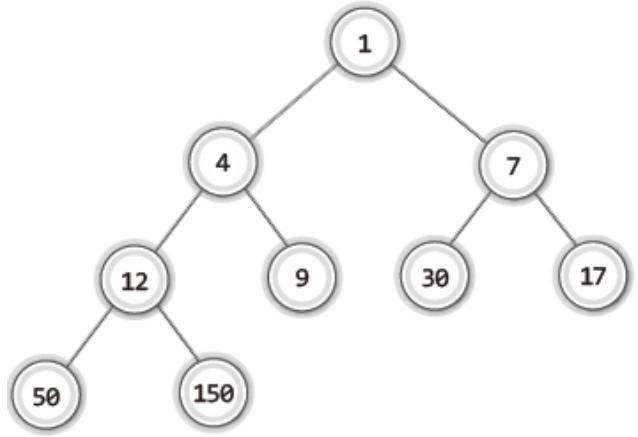
# 힙(Heap)의 소개




모든 노드에 저장된 값은 자식 노드에 저장된 값보다 크거나 같아야 한다.
즉 루트 노드에 저장된 값이 가장 커야 한다.

▶ [그림 09-2: 최대 힙(max heap)]

힙은 '완전 이진 트리'이다!



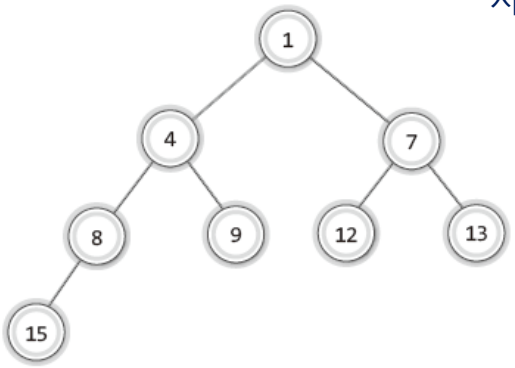

모든 노드에 저장된 값은 자식 노드에 저장된 값보다 크거나 같아야 한다.
즉 루트 노드에 저장된 값이 가장 커야 한다.

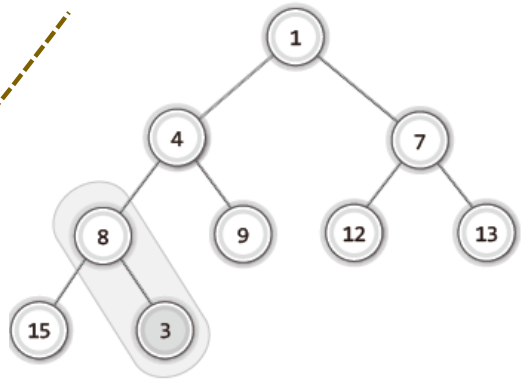
▶ [그림 09-3: 최소 힙(min heap)]

# 힙의 구현과
# 우선순위 큐의 완성

# 힙에서의 데이터 저장과정

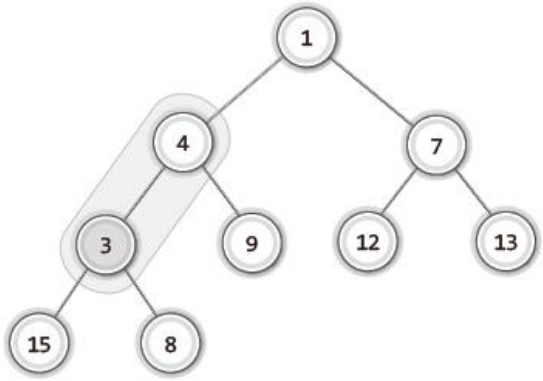
자식 노드 데이터의 우선순위 ≤ 부모 노드 데이터의 우선순위



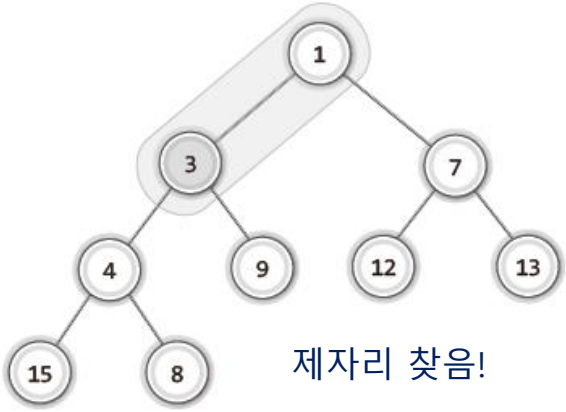




새 데이터는 우선순위가 낮다는 가정하에 끝!

에 저장 그리고 부모 노드와 비교를 진행!

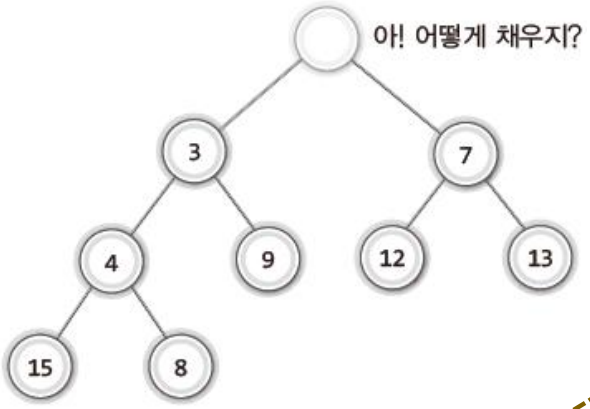


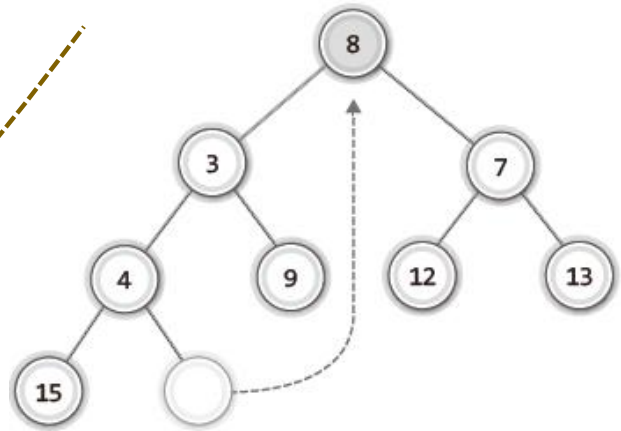

부모 노드와 비교 및 자리 바꿈



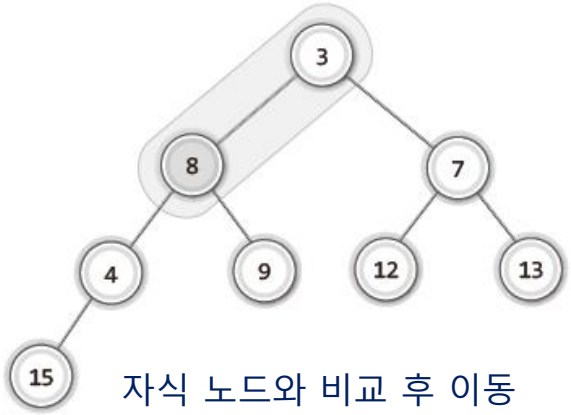

제자리 찾음!

# 힙에서의 데이터 삭제과정

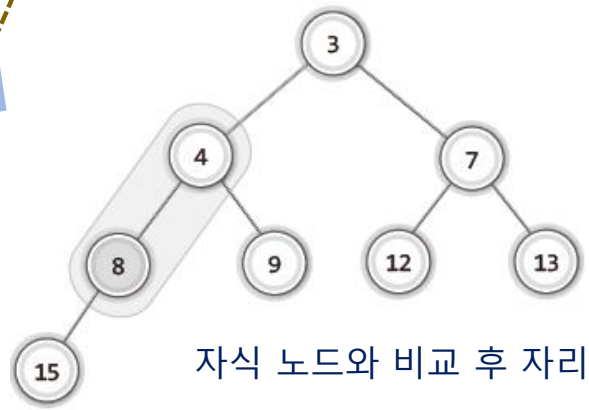



마지막 노드를 루트 노드로 이동!




자식 노드와 비교 후 이동




자식 노드와 비교 후 자리 확정!

# 삽입과 삭제의 과정에서 보인 성능의 평가

배열 기반 우선순위 큐의 시간 복잡도

- 배열 기반 데이터 삽입의 시간 복잡도 *O(n)*
- 배열 기반 데이터 삭제의 시간 복잡도 *O(1)*

* 배열 기반의 경우 배열의 요소를 한 칸씩 뒤로 밀거나 당길 때 O(N)이 소요된다.

연결 리스트 기반 우선순위 큐의 시간 복잡도

- 연결 리스트 기반 데이터 삽입의 시간 복잡도 *O(n)*
- 연결 리스트 기반 데이터 삭제의 시간 복잡도 *O(1)*

# 삽입과 삭제의 과정에서 보인 성능의 평가

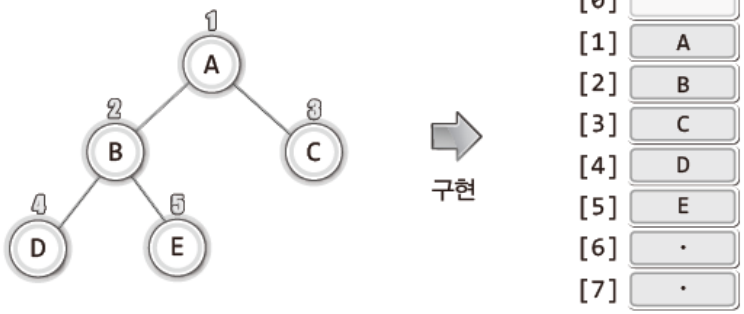
힙 기반 우선순위 큐의 시간 복잡도

- 힙 기반 데이터 삽입의 시간 복잡도 $O(\log_2 n)$
- 힙 기반 데이터 삭제의 시간 복잡도 $O(\log_2 n)$

힙에서 삽입/삭제의 최대 비교 횟수는 (힙의 높이) 번이다.

힙의 높이 h를 저장된 데이터의 수 n으로 나타내면?

# 배열을 기반으로 힙을 구현하는데 필요한 지식들




"연결 리스트를 기반으로 힙을 구현하면,

새로운 노드를 힙의 '마지막 위치'에

추가하는 것이 쉽지 않다."

배열 기반에서 인덱스 값 구하기!

- 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

나눗셈은 정수형 나눗셈

# 원리 이해 중심의 힙 구현: 헤더파일의 소개

```
typedef char HData;
typedef int Priority;

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

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

void HeapInit(Heap * ph);
int HIsEmpty(Heap * ph);

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

힙을 구현하는 것이므로 함수의 이름이 enqueue, dequeue가 아니다!

우선순위 큐의 구현을 목적으로 하는 힙의 헤더파일 정의!

우선순위가 가장 높은 데이터 삭제되도록 정의!

# 원리 이해 중심의 힙 구현: 숙지할 내용

- 힙은 완전 이진 트리이다.
- 힙의 구현은 배열을 기반으로 하며 인덱스가 0인 요소는 비워둔다.
- 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.
  - 1순위, 2순위, ...

배열을 기반으로 하는 경우! 힙에 저장된 노드의 개수와 마지막 노드의 고유번호가 일치하기 때문에 마지막 노드의 인덱스 값을 쉽게 얻을 수 있다! 이것은 중요한 특징이다!

# 원리 이해 중심의 힙 구현: 초기화와 Helper

```
void HeapInit(Heap * ph)
{
    ph->numOfData = 0;
}
```

초기화!

```
int HIsEmpty(Heap * ph)
{
    if(ph->numOfData == 0)
        return TRUE;
    else
        return FALSE;
}
```

비었는지 확인

```
int GetParentIDX(int idx)
{
    return idx/2;
}
```

부모 노드의 인덱스 값 반환

*Helper!*

```
int GetLChildIDX(int idx)
{
    return idx*2;
}
```

왼쪽 자식 노드의 인덱스 값 반환

*Helper!*

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

오른쪽 자식 노드의 인덱스 값 반환

# 원리 이해 중심의 힙 구현: Helper

우선 순위가 높은 자식의 인덱스 값 반환!

numOfData는 마지막 노드의 고유번호이니,
자식 노드의 값이 이보다 크면 존재하지 않는 자식 노드이다.

자식 노드 없으면 0 반환!

자식 노드가 하나 존재하면 이는 왼쪽 자식 노드이다.
완전 이진 트리 이므로!

```
int GetHiPriChildIDX(Heap * ph, int idx)
{
    // 자식 노드가 존재하지 않는다면,
    if(GetLChildIDX(idx) > ph->numOfData)
        return 0;

    // 자식 노드가 왼쪽 자식 노드 하나만 존재한다면,
    else if(GetLChildIDX(idx) == ph->numOfData)
        return GetLChildIDX(idx);

    // 자식 노드가 둘 다 존재한다면,
    else
    {
        // 오른쪽 자식 노드의 우선순위가 높다면,
        if(ph->heapArr[GetLChildIDX(idx)].pr > ph->heapArr[GetRChildIDX(idx)].pr)
            return GetRChildIDX(idx);    // 오른쪽 자식 노드의 인덱스 값 반환

        // 왼쪽 자식 노드의 우선순위가 높다면,
        else
            return GetLChildIDX(idx);    // 왼쪽 자식 노드의 인덱스 값 반환
    }
}
```

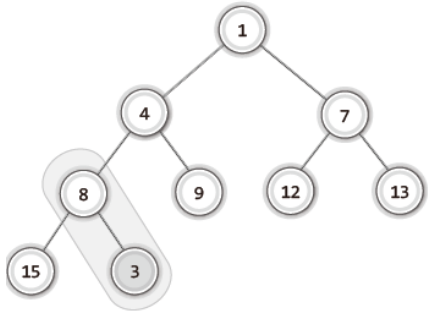
# 원리 이해 중심의 힙 구현: HInsert

```
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 올리지는 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;
}
```

새 노드를 마지막 위치에 직접 저장하지 않아도 된다. 어차피 이동을 하니!




새 노드의 인덱스 정보를 갱신만 하자! 그림처럼 실제 저장까지 할 필요는 없다! 어차피 이동해야 하므로!

# 원리 이해 중심의 힙 구현: HDelete

```
HData HDelete(Heap * ph)
{
    HData retData = (ph->heapArr[1]).data;      // 반환을 위해서 삭제할 데이터 저장
    HeapElem lastElem = ph->heapArr[ph->numOfData];    // 힙의 마지막 노드 저장
```

마지막 노드를 임시 저장하여 그에 맞는 자리를 찾아나간다!

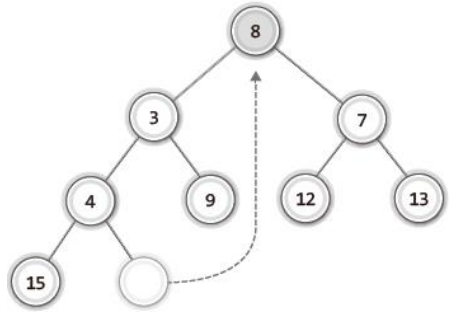
```
    // 아래의 변수 parentIdx에는 마지막 노드가 저장될 위치정보가 담긴다.
    int parentIdx = 1;    // 루트 노드가 위치해야 할 인덱스 값 저장
    int childIdx;

    // 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 올림
        ph->heapArr[parentIdx] = ph->heapArr[childIdx];
        parentIdx = childIdx;     // 마지막 노드가 저장될 위치정보를 한 레벨 내림
    }      // 반복문 탈출하면 parentIdx에는 마지막 노드의 위치정보가 저장됨

    ph->heapArr[parentIdx] = lastElem;      // 마지막 노드 최종 저장
    ph->numOfData -= 1;
    return retData;
}
```

굳이 루트 노드의 자리로 옮기지 않아도 된다!



# 힙의 확인을 위한 main 함수!

```
int main(void)
{
    Heap heap;
    HeapInit(&heap);                // 힙의 초기화

    HInsert(&heap, 'A', 1);         // 문자 'A'를 최고의 우선순위로 저장
    HInsert(&heap, 'B', 2);         // 문자 'B'를 두 번째 우선순위로 저장
    HInsert(&heap, 'C', 3);         // 문자 'C'를 세 번째 우선순위로 저장
    printf("%c \n", HDelete(&heap));

    HInsert(&heap, 'A', 1);         // 문자 'A' 한 번 더 저장!
    HInsert(&heap, 'B', 2);         // 문자 'B' 한 번 더 저장!
    HInsert(&heap, 'C', 3);         // 문자 'C' 한 번 더 저장!
    printf("%c \n", HDelete(&heap));

    while(!HIsEmpty(&heap))
        printf("%c \n", HDelete(&heap));

    return 0;
}
```

SimpleHeap.h
SimpleHeap.c
SimpleHeapMain.c

**실행결과**

```
A
A
B
B
C
C
```

데이터를 저장할 때 우선순위 정보를 별도로 전달하는 것은 적합하지 않은 경우가 많다.
데이터 자체를 비교하는 것으로 어떤 데이터가 우선하는지 알 수 있기 때문이다.

# 구조체 변경

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

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

구조체의 변경!

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

typedef int PriorityComp(HData d1, HData d2);

```
void HeapInit(Heap * ph, PriorityComp pc)
{
     ph->numOfData = 0;
     ph->comp = pc;
}
```

구조체의 변경에 따른 초기화 함수의 변경!

프로그래머가 힙의 우선순위 판단 기준을 설정할 수 있어야 한다!

# 우선 순위 결정 함수: PriorityComp

PriorityComp 형 함수의 정의 기준

- 첫 번째 인자의 우선순위가 높다면, 0보다 큰 값 반환!
- 두 번째 인자의 우선순위가 높다면, 0보다 작은 값 반환!
- 첫 번째, 두 번째 인자의 우선순위가 동일하다면, 0이 반환!

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

우선 순위 정보를 별도로 받지 않는다.

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

PriorityComp형 함수가 등록되면! HInsert 함수는 등록된 함수를 활용하여 우선순위를 비교 판단한다.

# Helper 함수의 변경

```
int GetHiPriChildIDX(Heap * ph, int idx)
{
    if(GetLChildIDX(idx) > ph->numOfData)
        return 0;

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

    else
    {
    //  if(ph->heapArr[GetLChildIDX(idx)].pr
    //              > ph->heapArr[GetRChildIDX(idx)].pr)
        if(ph->comp(ph->heapArr[GetLChildIDX(idx)],
                    ph->heapArr[GetRChildIDX(idx)]) < 0)
            return GetRChildIDX(idx);
        else
            return GetLChildIDX(idx);
    }
}
```

comp에 등록된 함수의 호출결과를 통해서 우선순위를 판단한다.

# HInsert의 변경

```
void HInsert(Heap * ph, HData data)
{
    int idx = ph->numOfData+1;

    while(idx != 1)
    {
    //  if(pr < (ph->heapArr[GetParentIDX(idx)].pr))
        if(ph->comp(data, ph->heapArr[GetParentIDX(idx)]) > 0)
        {
            ph->heapArr[idx] = ph->heapArr[GetParentIDX(idx)];
            idx = GetParentIDX(idx);
        }
        else
        {
            break;
        }
    }

    ph->heapArr[idx] = data;
    ph->numOfData += 1;
}
```

comp에 등록된 함수의 호출결과를
통해서 우선순위를 판단!

# HDelete의 변경

```
HData HDelete(Heap * ph)
{
    HData retData = ph->heapArr[1];
    HData lastElem = ph->heapArr[ph->numOfData];

    int parentIdx = 1;
    int childIdx;

    while(childIdx = GetHiPriChildIDX(ph, parentIdx))
    {
    //  if(lastElem.pr <= ph->heapArr[childIdx].pr)
        if(ph->comp(lastElem, ph->heapArr[childIdx]) >= 0)
            break;

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

    ph->heapArr[parentIdx] = lastElem;
    ph->numOfData -= 1;
    return retData;
}
```

comp에 등록된 함수의 호출결과를 통해서 우선순위를 판단!

# main 함수

```c
int main(void)
{
    Heap heap;
    HeapInit(&heap, DataPriorityComp);

    HInsert(&heap, 'A');
    HInsert(&heap, 'B');
    HInsert(&heap, 'C');
    printf("%c \n", HDelete(&heap));

    HInsert(&heap, 'A');
    HInsert(&heap, 'B');
    HInsert(&heap, 'C');
    printf("%c \n", HDelete(&heap));

    while(!HIsEmpty(&heap))
        printf("%c \n", HDelete(&heap));

    return 0;
}
```

```c
int DataPriorityComp(char ch1, char ch2)
{
    return ch2-ch1;
//  return ch1-ch2;
}
```

아스키 코드 값이 작은 문자의 우선순위가 더 높다!

UsefulHeap.h
UsefulHeap.c

UsefulHeapMain.c

```
A
A
B
B
C
C
```

**실행결과**

# 개선된 힙을 이용한 우선순위 큐의 구현

```
#include "UsefulHeap.h"

typedef Heap PQueue;
typedef HData PQData;

void PQueueInit(PQueue * ppq, PriorityComp pc);
int PQIsEmpty(PQueue * ppq);

void PEnqueue(PQueue * ppq, PQData data);
PQData PDequeue(PQueue * ppq);
```

힙의 함수를 사실상 우선순위 큐의 내용으로
구현해 놓았기 때문에 실제 할 일은 별것 없다!

```
void PQueueInit(PQueue * ppq, PriorityComp pc)
{
    HeapInit(ppq, pc);
}

int PQIsEmpty(PQueue * ppq)
{
    return HIsEmpty(ppq);
}

void PEnqueue(PQueue * ppq, PQData data)
{
    HInsert(ppq, data);
}

PQData PDequeue(PQueue * ppq)
{
    return HDelete(ppq);
}
```

# 요약

- 우선순위 큐는 데이터가 삽입된 순서가 아니라, 정해진 규칙에 의해 dequeue 순서가 정해지는 **추상 데이터 타입**이다.

- 우선순위 큐를 구현할 때 heap이라는 **자료 구조**를 흔히 사용한다.
  - 힙은 내부적으로 완전 이진 트리를 사용하며 부모-자식 노드의 관계를 우선 순위에 따라 유지한다.
  - 추가: 맨 마지막에 추가하고 위치 찾아 올리기
  - 삭제: 루트를 돌려주고, 맨 마지막을 루트로 올리고 위치 찾아 내리기
  - 순서를 갱신해야 하는 경우에도 O(log N) 시간이면 충분하다.

# 출석 인정을 위한 보고서 작성

- 아래 질문에 대해 A4 반장 이상으로 답한 후 포털에 제출 (다음 페이지도 있음)

1) 길이 N인 배열에 N개의 정수가 있다. 이 정수를 커지는 순서대로 나열하는 작업을 “오름차순 정렬”이라고 한다.

**최소 힙**을 하나 사용하여 오름차순 정렬을 구현할 수 있을까? 또, 이 때 시간복잡도는 얼마일까?

예) 배열에 10, 3, 5, 6, 2가 저장되어 있다면 이를 오름차순 정렬하면 2, 3, 5, 6, 10이 된다.

# 출석 인정을 위한 보고서 작성

2) 배열에 1이 하나 들어가 있다. 매 번 두 개의 정수가 배열에 추가된다. 이 때마다 배열의 중간 값을 출력하고 싶다.

- 정렬을 사용해서 해결할 수 있을까?
- 최소 힙 하나와 최대 힙 하나를 사용해서 더 효율적으로 수행할 수 있을까?

배열: [1], 중간 값: 1

배열: [1, 5, 2], 중간 값: 2

배열: [1, 5, 2, 4, 3], 중간 값: 3

배열: [1, 5, 2, 4, 3, 100, 2], 중간 값: 3

빨간 색으로 표시된 수는 추가된 두 개의 정수이다.

중간 값: 순서대로 나열했을 때 중간에 오는 값