Data Structure

실습 7

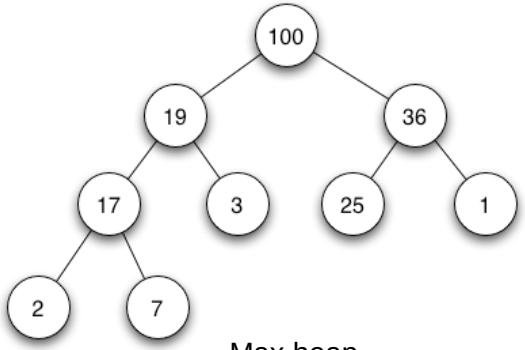
0. 이번 주 실습 내용

- **Heap & Priority Queue**
  - Heap의 정의
  - Priority Queue의 정의
  - Priority Queue와 Heap의 관계
- **Heap 구현 & 실습**
  - Heap의 구현 (Max heap)
  - Init, Print, Insert, Delete 연산 (Max heap)

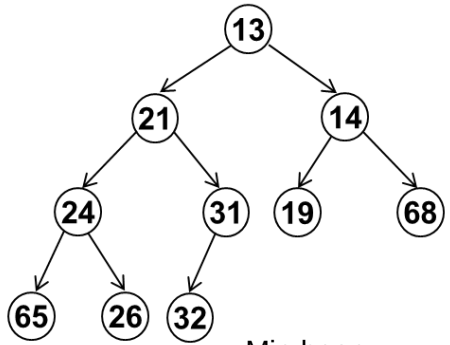
2. Heap & Priority Queue

- **Heap (정의): 여러 개의 값들 중에서 가장 큰 값, 또는 가장 작은 값을 빠르게 탐색하도록 만들어진 Binary Tree 자료구조**

- 조건
  - Tree는 **Complete** 해야한다
  - 각 Node의 key 값은 자식 Node 들의 Key 값보다 **크거나 같아야 한다** (Max heap)
    cf.) 각 Node의 key 값이 자식 Node 들의 Key 값보다 **작거나 같아야 한다** (Min heap)



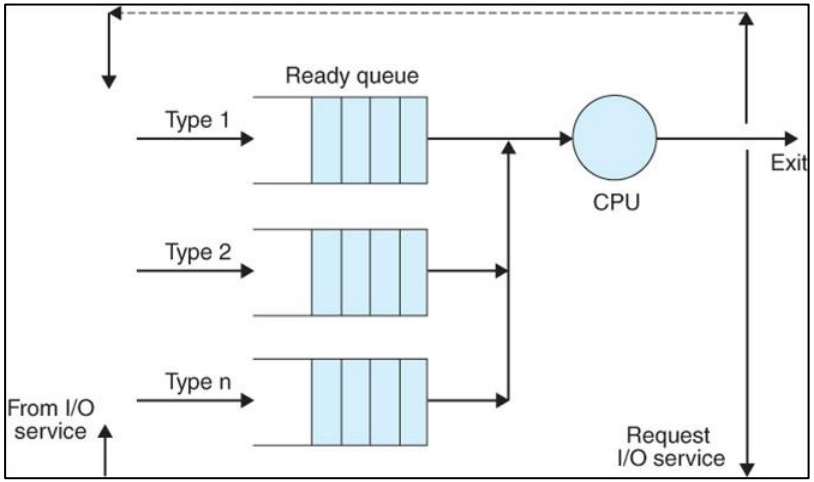
Max heap



Min heap

2. Heap & Priority Queue

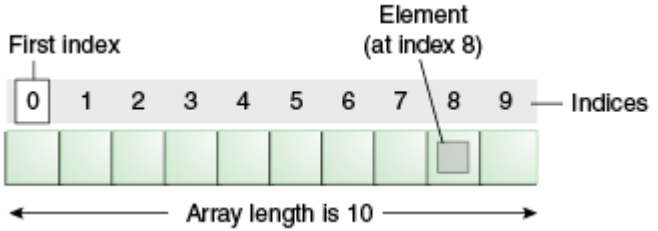
- **Priority Queue(우선순위 큐)**
  - Queue (큐)
    - 먼저 들어간 데이터가 먼저 나오는 FIFO(First In First Out)구조의 자료구조
  - Priority Queue
    - 들어간 순서에 상관없이 우선순위가 높은 데이터가 먼저 나오는 자료구조



대표적인 활용 예
Operating System
(Process Scheduling)

2. Heap & Priority Queue

- **Priority Queue(우선순위 큐) 구현 방법**
  1. 배열(Array)을 이용한 구현

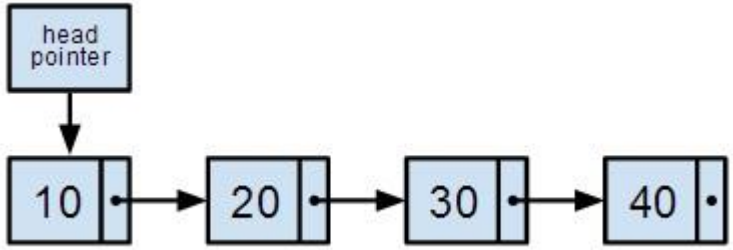


  - 우선순위가 높을수록 배열의 앞쪽에 데이터를 위치시킴

  - **장점**
    - 직관적으로 보았을 때 우선순위가 높은 데이터를 알기 쉽다
  - **단점**
    - 데이터를 추가 및 삭제하는 과정에서 우선순위에 따라 위치가 달라지므로 나머지 데이터들의 위치이동을 해야하는 불필요한 연산이 발생한다
    - 배열의 길이가 정해져 있어 일정 크기 만큼의 데이터만 넣을 수 있다

2. Heap & Priority Queue

- **Priority Queue(우선순위 큐) 구현 방법**
  2. 연결 리스트(Linked List)를 이용한 구현



  - 우선순위가 가장 높은 데이터를 헤더 노드가 가리키고 있고 그 뒤로 순서대로 데이터들이 연결되어 있다

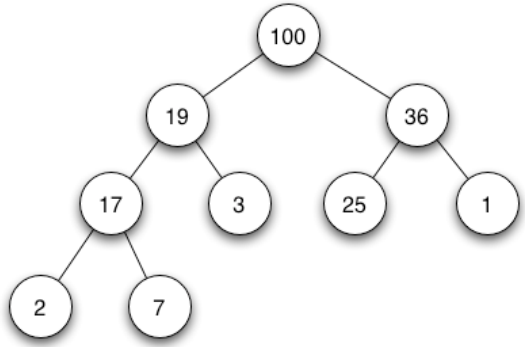
  - **장점**
    - 배열과 달리 데이터를 추가 및 삭제하는 과정에서 불필요한 연산이 추가되지 않는다
  - **단점**
    - 우선순위에 따라 데이터가 들어갈 위치를 처음부터 마지막까지 비교해야 하므로 시간이 걸린다 (O(n))

➔ **따라서 우선순위 큐는 일반적으로 힙(heap)이라는 자료구조를 이용해서 구현한다**

2. Heap & Priority Queue

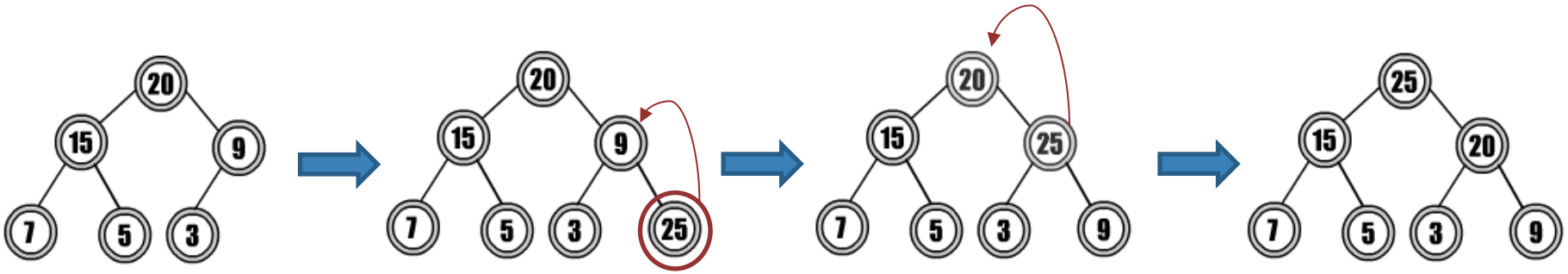
- **Priority Queue(우선순위 큐) 구현 방법**
  3. 힙(heap)을 이용한 구현

    - 우선순위가 가장 높은 데이터를 가리키고있는 완전 이진 트리(complete binary tree)
    - **장점**
      - 트리 구조이므로 데이터의 추가나 삭제 연산을 할 때 데이터의 이동이 빠르다 (O(log n))
      - 데이터의 추가나 삭제 연산이 생각보다 간단해 구현이 쉽다



3. Heap – 구현 (Max heap)

- **insert (데이터 추가 연산) O(log n)**
  1. Tree에 Node를 하나 확장 시켜 추가할 데이터를 넣어준다
  2. 부모 Node의 데이터와 값을 비교한다
     2-1. 부모 Node의 데이터보다 값이 클 경우 부모와 위치를 바꿔준 다음 다시 2번을 수행한다
     2-2. 부모 Node의 데이터보다 값이 작을 경우 연산을 종료한다.



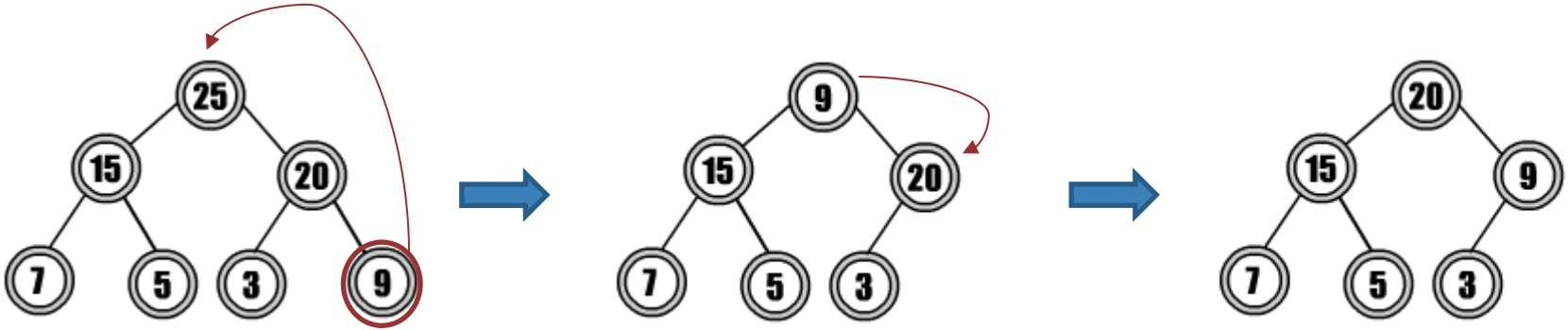
insert 25

3. Heap – 구현 (Max heap)

- **delete (데이터 삭제 연산) O(log n)**
  1. root Node의 데이터를 삭제하고 마지막 Node를 root Node로 이동시킨다
  2. 자식 Node들의 데이터 값을 비교한다

     2-1. 자식 Node들의 데이터 값 중 가장 큰 값을 가진 Node와 위치를 바꾸고 다시 2번을 수행한다

     2-2. 이동이 이루어지지 않았을 경우 연산을 종료한다

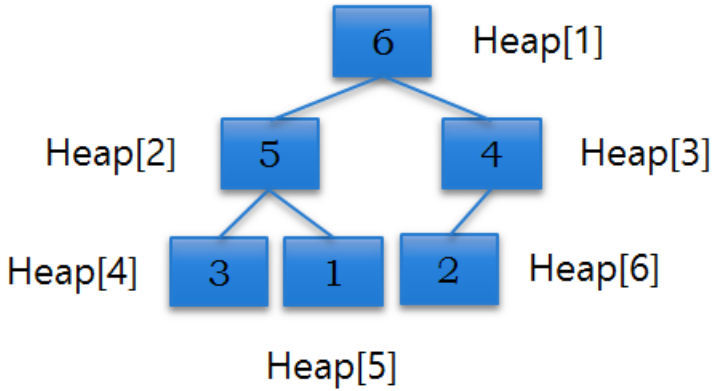


delete 25

3. Heap – 구현 (Max heap)

- **Heap 구현 방법**
  - **배열**을 이용하여 간단하게 구현할 수 있음
  - 일반적으로 배열의 첫 번째 요소는 사용하지 않음 (구현의 편의를 위해)

Heap[0]	Heap[1]	Heap[2]	Heap[3]	Heap[4]	Heap[5]	Heap[6]
x	6	5	4	3	1	2

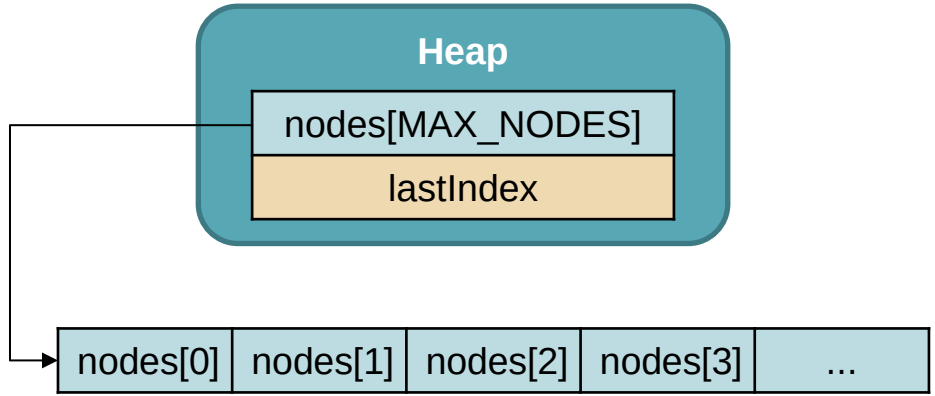


왼쪽 자식 Node의 index = (부모 Node의 index) * 2
오른쪽 자식 Node의 index = (부모 Node의 index) * 2 + 1

부모 Node의 index = (자기 자신 Node의 index) / 2

3. Heap – 실습

- **Heap Data Type**




```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   #define MAX_NODES 50
6   typedef struct Heap
7   {
8       int nodes[MAX_NODES];
9       int lastIndex;
10  }Heap;
11
12  void initHeap(Heap* heap);
13  void insertData(Heap* heap, int data);
14  void printHeap(Heap heap);
15  void deleteData(Heap* heap);
16
17
```

3. Heap – 실습

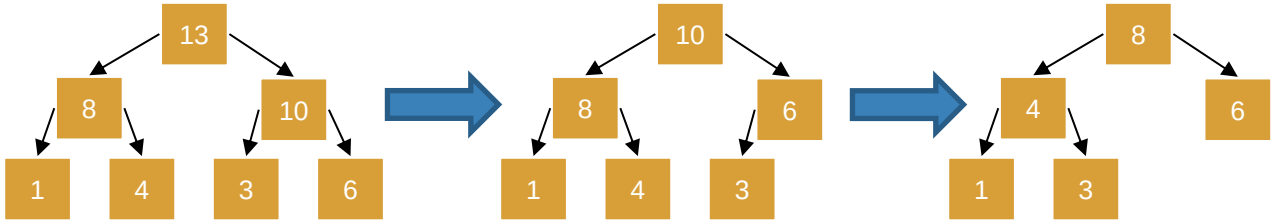
- **main**

```
C:₩WINDOWS₩system32₩cmd.exe
13
8       10
1       4       3       6


10
8       6
1       4       3

8
4       6
1       3

계속하려면 아무 키나 누르십시오 . . .
```




```
18  int main()
19  {
20      //MAX heap
21      Heap heap;
22
23      initHeap(&heap);
24
25      insertData(&heap, 1);
26      insertData(&heap, 3);
27      insertData(&heap, 8);
28      insertData(&heap, 13);
29      insertData(&heap, 4);
30      insertData(&heap, 10);
31      insertData(&heap, 6);
32
33      printHeap(heap);
34
35      deleteData(&heap);
36      printHeap(heap);
37
38      deleteData(&heap);
39      printHeap(heap);
40
41      return 0;
42  }
43
```

3. Heap – 실습

- **init (heap 초기화)**
  1. 모든 데이터를 0으로 초기화 시켜주고
     heap tree의 마지막 node index를 0으로 지정

- **print (heap 출력)**
  1. 각 Level에 해당하는 node들은 같은 줄에 출력

```
45  void initHeap(Heap* heap)
46  {
47      int i;
48      for (i = 1; i < MAX_NODES; i++)
49          heap->nodes[i] = 0;
50      heap->lastIndex = 0;
51
52  }
53
54
55  void printHeap(Heap heap)
56  {
57      int i, count, newLineIndex;
58      count = 1;
59      newLineIndex = 0;
60      for (i = 1; i <= heap.lastIndex; i++)
61      {
62          printf("%d\t", heap.nodes[i]);
63          //heap tree의 각 level은 한 줄에 출력되도록 함
64          if (pow(2, newLineIndex) == count)
65          {
66              printf("\n");
67              newLineIndex++;
68              count = 0;
69          }
70          count++;
71      }
72      printf("\n\n");
73  }
```

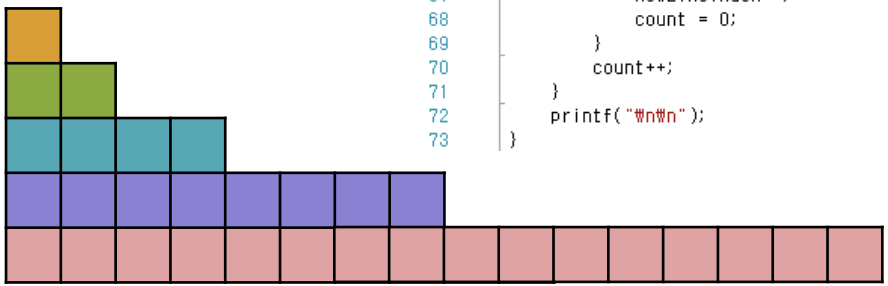
3. Heap – 실습

- **init (heap 초기화)**
  1. 모든 데이터를 0으로 초기화 시켜주고
     heap tree의 마지막 node index를 0으로 지정

- **print (heap 출력)**
  1. 각 Level에 해당하는 node들은 같은 줄에 출력

(Binary tree의 level별)

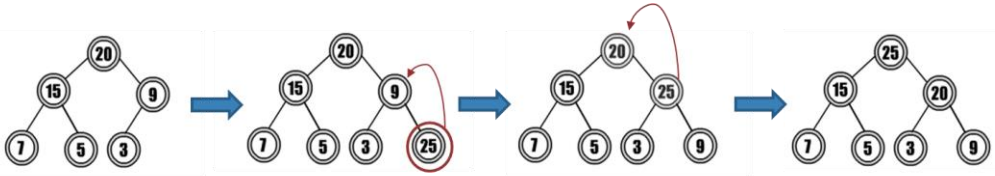
Node 개수	Node 개수 누적
2^n	$2^{n+1}-1$
2^0	2^0
2^1	2^1+2^0
2^2	$2^2+2^1+2^0$
2^3	$2^3+2^2+2^1+2^0$
2^4	$2^4+2^3+2^2+2^1+2^0$



```
45  void initHeap(Heap* heap)
46  {
47      int i;
48      for (i = 1; i < MAX_NODES; i++)
49          heap->nodes[i] = 0;
50      heap->lastIndex = 0;
51
52  }
55  void printHeap(Heap heap)
56  {
57      int i, count, newLineIndex;
58      count = 1;
59      newLineIndex = 0;
60      for (i = 1; i <= heap.lastIndex; i++)
61      {
62          printf("%d₩t", heap.nodes[i]);
63          //heap tree의 각 level은 한 줄에 출력되도록 함
64          if (pow(2, newLineIndex) == count)
65          {
66              printf("₩n");
67              newLineIndex++;
68              count = 0;
69          }
70          count++;
71      }
72      printf("₩n₩n");
73  }
```

3. Heap – 실습

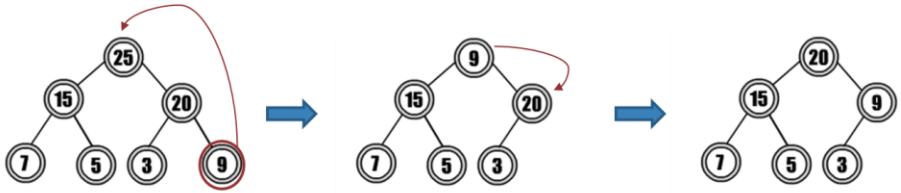
- **insert (데이터 추가)**
  1. heap에 데이터를 넣을 수 있는지 검사
  2. heap에 node를 확장 시켜서 데이터를 추가
  3. (확장시킨 node의 위치에서부터) 부모 node의 데이터를 비교
     - 3-1. 부모 node보다 값이 클 경우 부모 node와 자리를 바꾸고 다시 3번을 수행
     - 3-2. 부모 node보다 값이 작을 경우 연산을 종료




```
76  void insertData(Heap* heap, int data)
77  {
78      int index;
79      //heap이 꽉 차있는지 검사
80      if (heap->lastIndex == MAX_NODES - 1)
81      {
82          printf("Heap is full₩n");
83          return;
84      }
85
86      //heap에 node를 확장시켜 데이터를 추가
87
88
89
90      //부모 node의 데이터를 확인해가면서 업데이트
91      while (                                    )
92      {
93
94
95      }
96
97
98  }
99
```

3. Heap – 실습

- **delete (데이터 삭제)**
  1. heap에 데이터를 지울 수 있는지 검사
  2. heap의 root node에만 데이터가 있는 경우 값 삭제
  3. heap의 마지막 node를 root node로 대체 후 마지막 node 삭제
  4. (root node에서부터) 자식 node들과 값을 비교
     - 4-1. 자식 node의 값이 클 경우 자식 node와 자리를 바꾸고 다시 4번을 수행
     - 4-2. 자식 node의 값이 작을 경우 연산을 종료



delete 25

```
101  void deleteData(Heap* heap)
102  {
103      int temp, parentIndex, childIndex;
104      //heap이 비어있는지 검사
105      if (heap->lastIndex == 0)
106      {
107          printf("Heap is empty₩n");
108          return;
109      }
110
111      //root node에만 데이터가 존재하는 경우
112      if (heap->lastIndex == 1)
113      {
114          heap->nodes[heap->lastIndex] = 0;
115          heap->lastIndex = 0;
116          return;
117      }
118
119      //heap의 마지막 node의 데이터를 임시 변수에 저장
120
121
122
123
124      parentIndex = 1;
125      childIndex = 2;
126      //root node부터 시작해서 자식 노드들의 데이터 값과 비교하여 업데이트
127      while (childIndex <= heap->lastIndex)
128      {
129          //sibling node 중 값이 큰 node를 선택
130          if (
131
132
133          //임시 변수에 저장된 값과 비교
134          if (
135              break;
136
137          //자식 node의 값이 더 클 경우 부모 node와 교체
138
139
140
141      }
142
143
144  }
```