9장 우선순위 큐

9.1 우선순위 큐 추상 데이터 타입

- 우선순위 큐(priority queue)
  - 우선순위를 가진 항목들을 저장하는 큐
  - FIFO 순서가 아니라 우선 순위가 높은 데이터가 먼저 나가게 된다.

우선순위 큐

- 가장 일반적인 큐: 스택이나 FIFO 큐를 우선순위 큐로 구현할 수 있다.

자료구조	삭제되는 요소
스택	가장 최근에 들어온 데이터
큐	가장 먼저 들어온 데이터
우선순위큐	가장 우선순위가 높은 데이터

- 응용분야
  - 시뮬레이션 시스템(여기서의 우선 순위는 대개 사건의 시각이다.)
  - 네트워크 트래픽 제어
  - 운영 체제에서의 작업 스케쥴링

우선순위큐 ADT

·객체: n개의 element형의 우선 순위를 가진 요소들의 모임

· 연산:

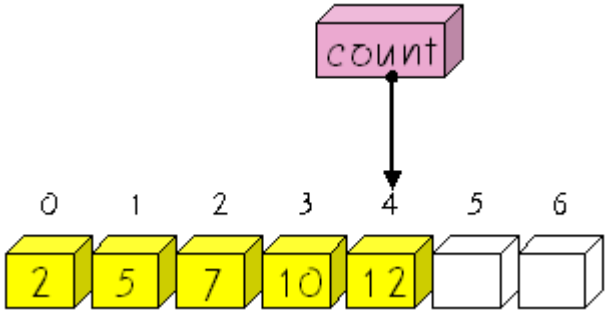
- create() ::= 우선 순위큐를 생성한다.
- init(q) ::= 우선 순위큐 q를 초기화한다.
- is_empty(q) ::= 우선 순위큐 q가 비어있는지를 검사한다.
- is_full(q) ::= 우선 순위큐 q가 가득 찼는가를 검사한다.
- insert(q, x) ::= 우선 순위큐 q에 요소 x를 추가한다.
- delete(q) ::= 우선 순위큐로부터 가장 우선순위가 높은 요소를 삭제하고 이 요소를 반환한다.
- find(q) ::= 우선 순위가 가장 높은 요소를 반환한다.

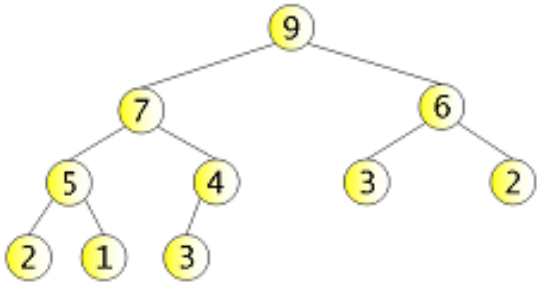
우선순위큐 연산

- 가장 중요한 연산은 insert 연산(요소 삽입), delete 연산(요소 삭제)이다.
- 우선순위 큐는 2가지로 구분
  - 최소 우선순위 큐: 우선순위가 낮은 요소부터 삭제
  - 최대 우선순위 큐: 우선순위가 높은 요소부터 삭제

9.2 우선순위 큐 구현 방법

- 배열을 이용한 우선순위 큐
- 연결리스트를 이용한 우선순위 큐
- 히프(heap)를 이용한 우선순위 큐



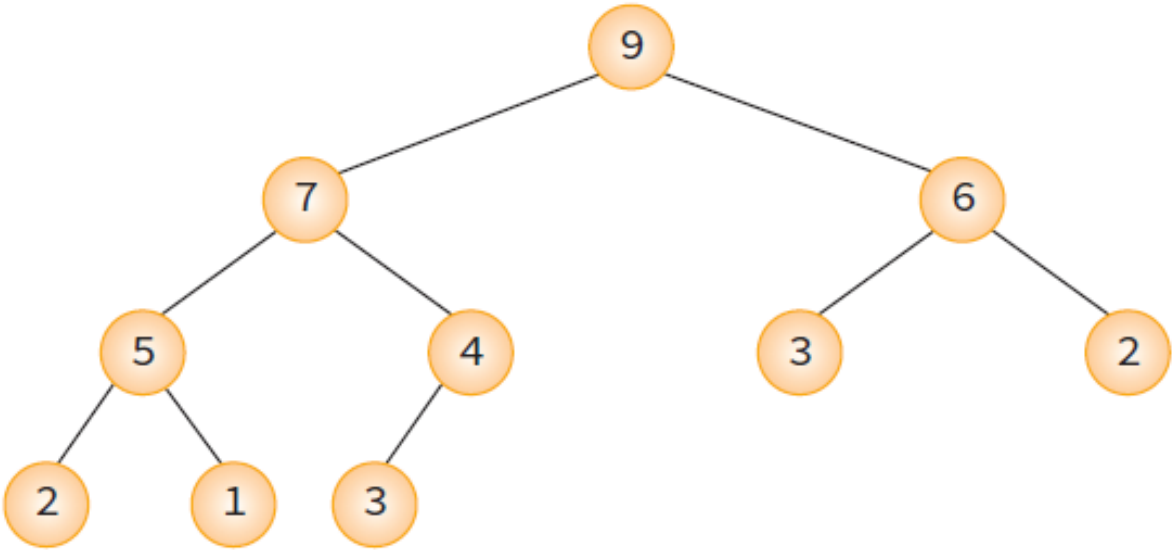


우선순위 큐 구현에 대한 복잡도 비교

표현 방법	삽 입	삭 제
순서없는 배열	O(1)	O(n)
순서없는 연결 리스트	O(1)	O(n)
정렬된 배열	O(n)	O(1)
정렬된 연결 리스트	O(n)	O(1)
히프	O(logn)	O(logn)

9.3 히프(heap)

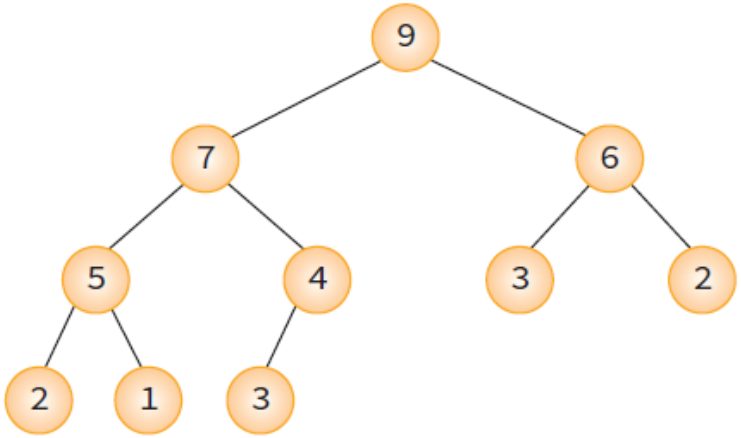
- 히프 정의
  - 노드의 키들이 다음 식을 만족하는 완전이진트리
  - *key(부모노드) ≥ key(자식노드)*



히프의 종류

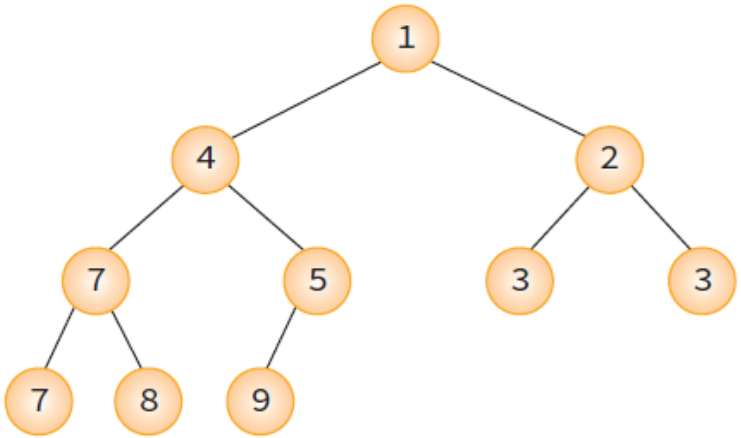
최대 히프(max heap):
부모 노드의 키값이 자식 노드의 키값보다 크거나 같은 완전 이진 트리

$key(\text{부모 노드}) \geq key(\text{자식 노드})$



최소 히프(min heap):
부모 노드의 키값이 자식 노드의 키값보다 작거나 같은 완전 이진 트리

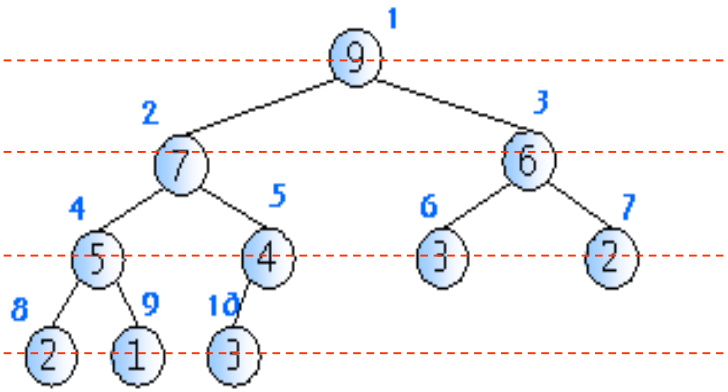
$key(\text{부모 노드}) \leq key(\text{자식 노드})$



히프의 높이

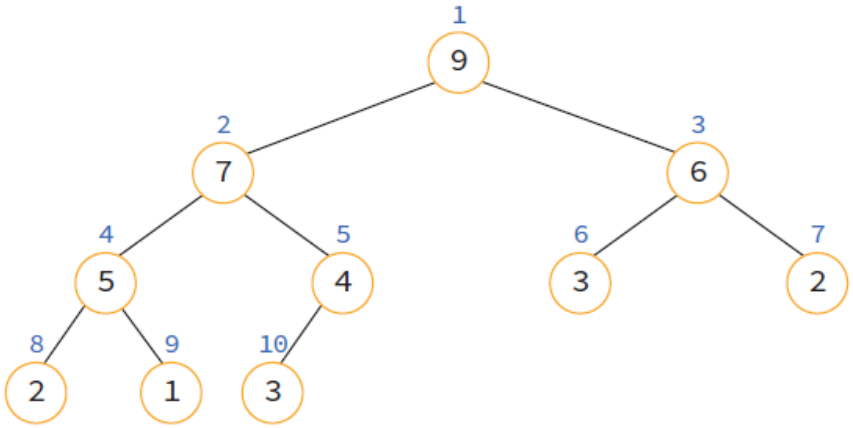
- n개의 노드를 가지고 있는 히프의 높이는 O(logn)
  - 히프는 완전이진트리
  - 마지막 레벨 h을 제외하고는 각 레벨 i에 2^{i-1}개의 노드 존재

깊이	노드의 개수
1	$1=2^0$
2	$2=2^1$
3	$4=2^2$
4	3



힙의 구현방법: 배열

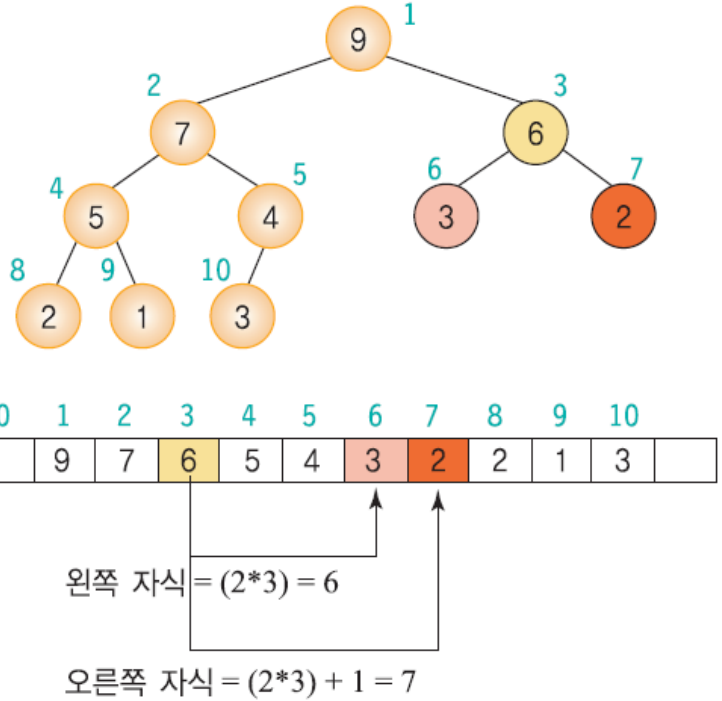
- 힙프는 배열을 이용하여 구현하는 것이 편리함
  - 완전이진트리이므로 각 노드에 번호를 붙일 수 있다
  - 이 번호를 배열의 인덱스라고 생각



0	1	2	3	4	5	6	7	8	9	10	
	9	7	6	5	4	3	2	2	1	3	

배열 구현의 장점

- 부모노드와 자식노드를 찾기가 쉽다.
  - 왼쪽 자식의 인덱스 = (부모의 인덱스)*2
  - 오른쪽 자식의 인덱스 = (부모의 인덱스)*2 + 1
  - 부모의 인덱스 = (자식의 인덱스)/2



9.4 히프의 구현

- 히프 정의

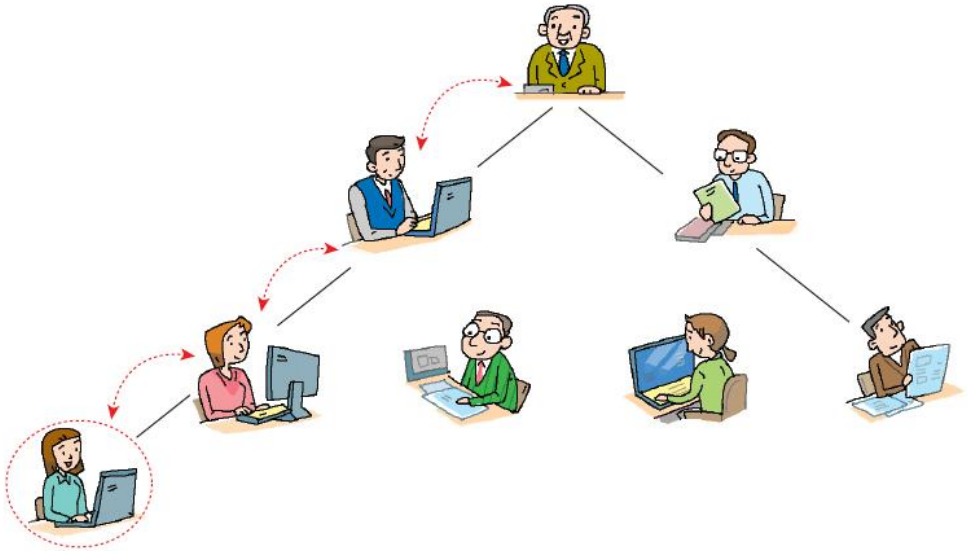
```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define MAX_ELEMENT 200
4   typedef struct {
5       int key;
6   } element;
7   typedef struct {
8       element heap[MAX_ELEMENT];
9       int heap_size;
10  } HeapType;
```

히프에서의 삽입 (최대히프)

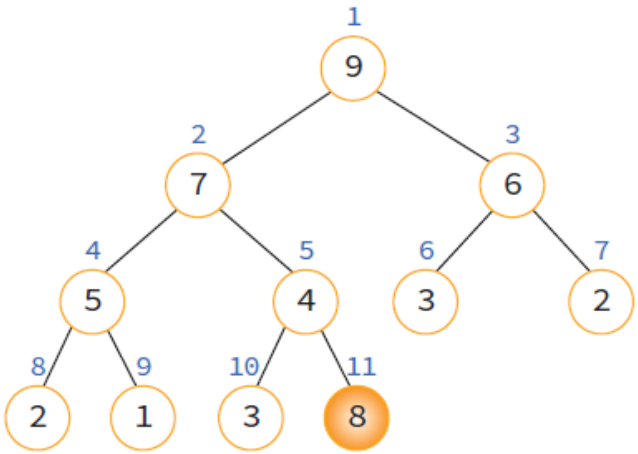
- 히프에 있어서 삽입 연산은 회사에서 신입 사원이 들어오면 일단 말단 위치에 앉힌 다음에, 신입 사원의 능력을 봐서 위로 승진시키는 것과 흡사

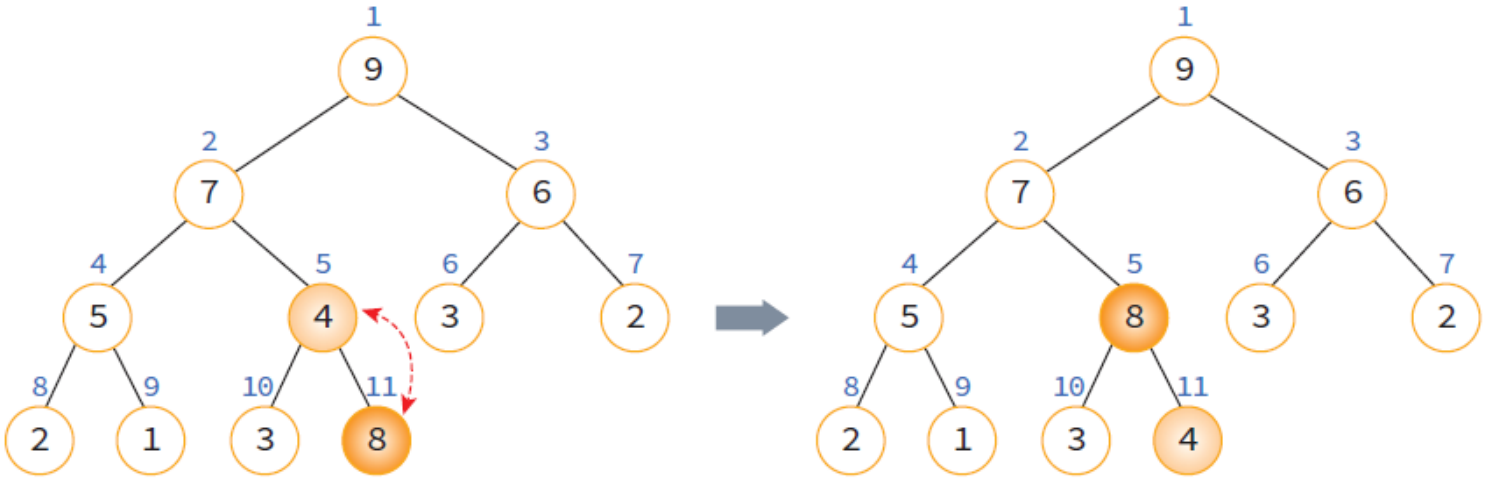
(1) 히프에 새로운 요소가 들어 오면, 일단 새로운 노드를 히프의 마지막 노드에 이어서 삽입

(2) 삽입 후에 새로운 노드를 부모 노드들과 교환해서 히프의 성질을 만족 (upheap 연산)

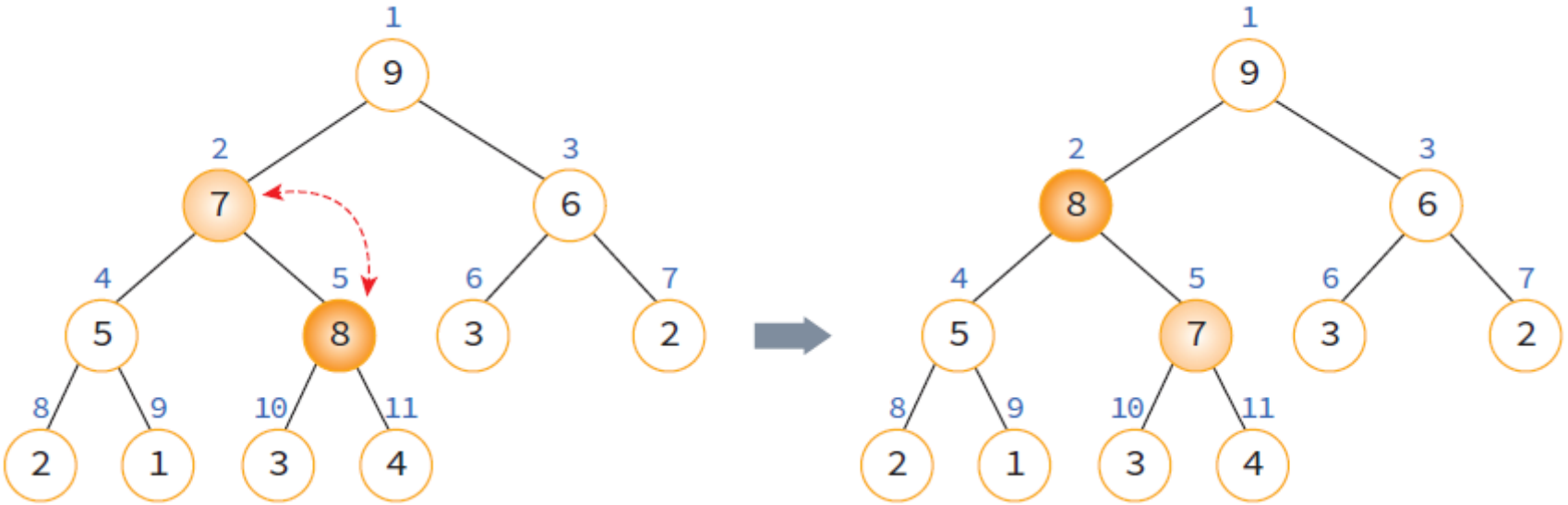


upheap 연산 (1/2)


1
9
2
7
3
6
4
5
5
4
6
3
7
2
8
2
9
1
10
3
11
8


1
9
2
7
3
6
4
5
5
4
6
3
7
2
8
2
9
1
10
3
11
8
1
9
2
7
3
6
4
5
5
8
6
3
7
2
8
2
9
1
10
3
11
4

upheap 연산 (2/2)



히프트리 삽입 알고리즘


```
insert_max_heap(A, key)


 heap_size ← heap_size + 1;
 i ← heap_size;
 A[i] ← key;
 while i ≠ 1 and A[i] > A[PARENT(i)] do
            A[i] <-> A[PARENT];          // 교환(swap)
            i ← PARENT(i);
```

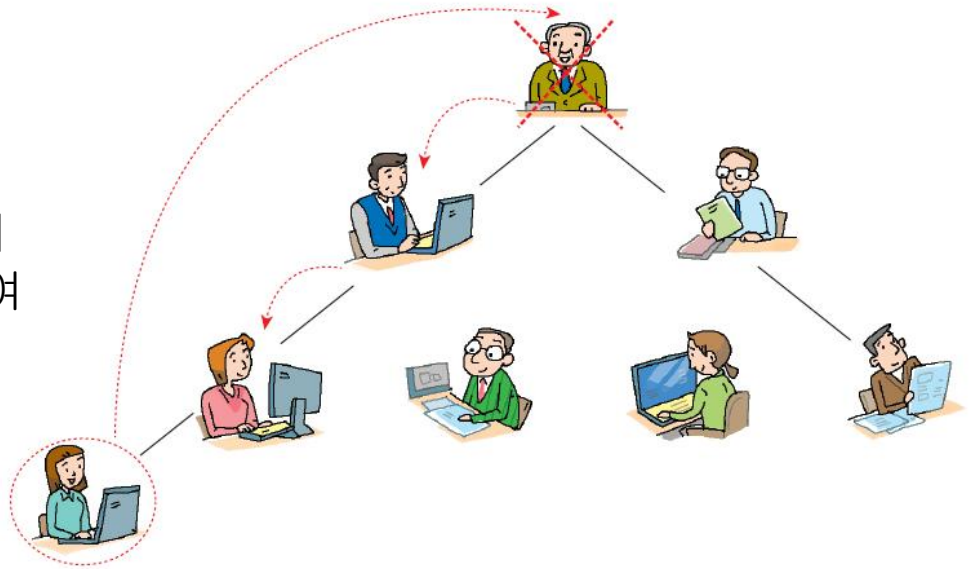
삽입 함수 구현: heap1.c


```
25  // 현재 요소의 개수가 heap_size인 히프 h에 item을 삽입한다.
26  // 삽입 함수,
27  void insert_max_heap(HeapType* h, element item)
28  {
29      int i;
30      i = ++(h->heap_size);
31
32      //  트리를 거슬러 올라가면서 부모 노드와 비교하는 과정
33      while ((i != 1) && (item.key > h->heap[i/2].key)) {
34          h->heap[i] = h->heap[i/2];
35          i /= 2;
36      }
37      h->heap[i] = item;      // 새로운 노드를 삽입
38  }
```

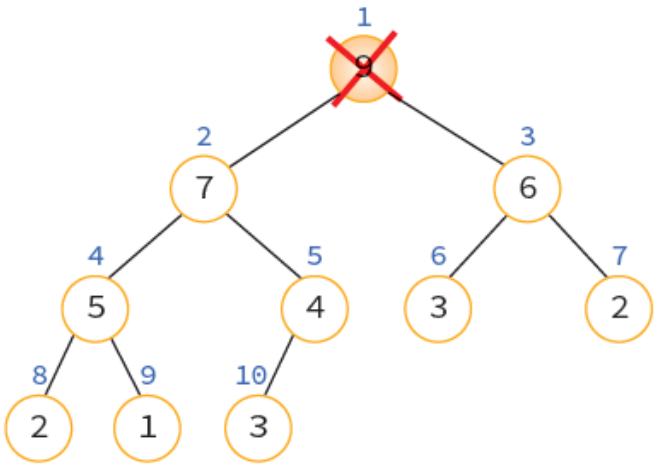
히프에서의 삭제

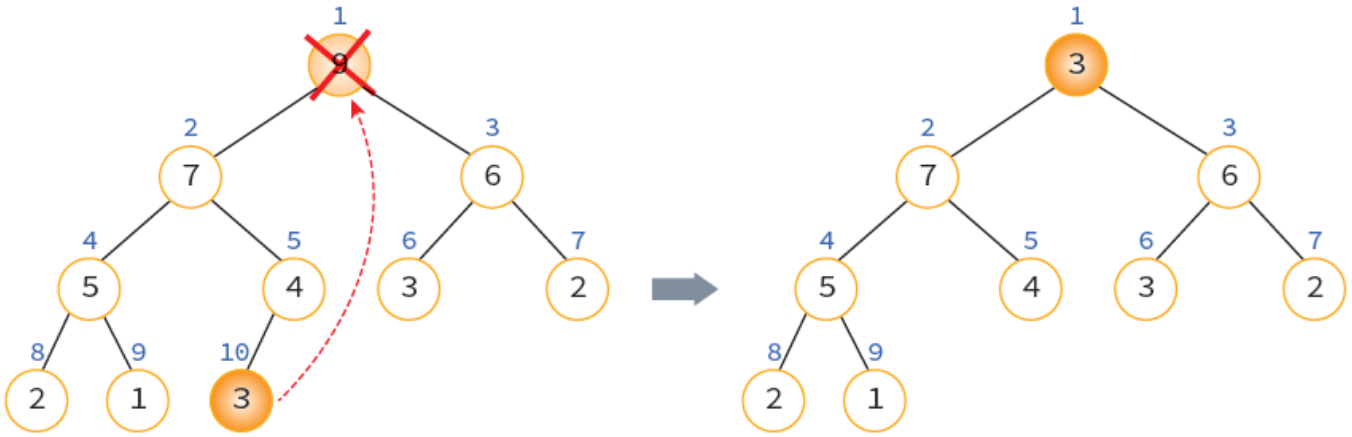
- 최대히프에서의 삭제는 가장 큰 키 값을 가진 노드를 삭제하는 것을 의미
  -> 따라서 루트 노드가 삭제된다.
- 삭제 연산은 회사에서 사장의 자리가 비게 되면 먼저 제일 말단 사원을 사장 자리로 올린 다음에, 능력에 따라 강등시키는 것과 비슷하다.

1) 루트 노드를 삭제한다
2) 마지막 노드를 루트 노드로 이동한다.
3) 루트에서부터 단말 노드까지의 경로에 있는 노드들을 교환하여 히프 성질을 만족시킨다. (downheap 연산)

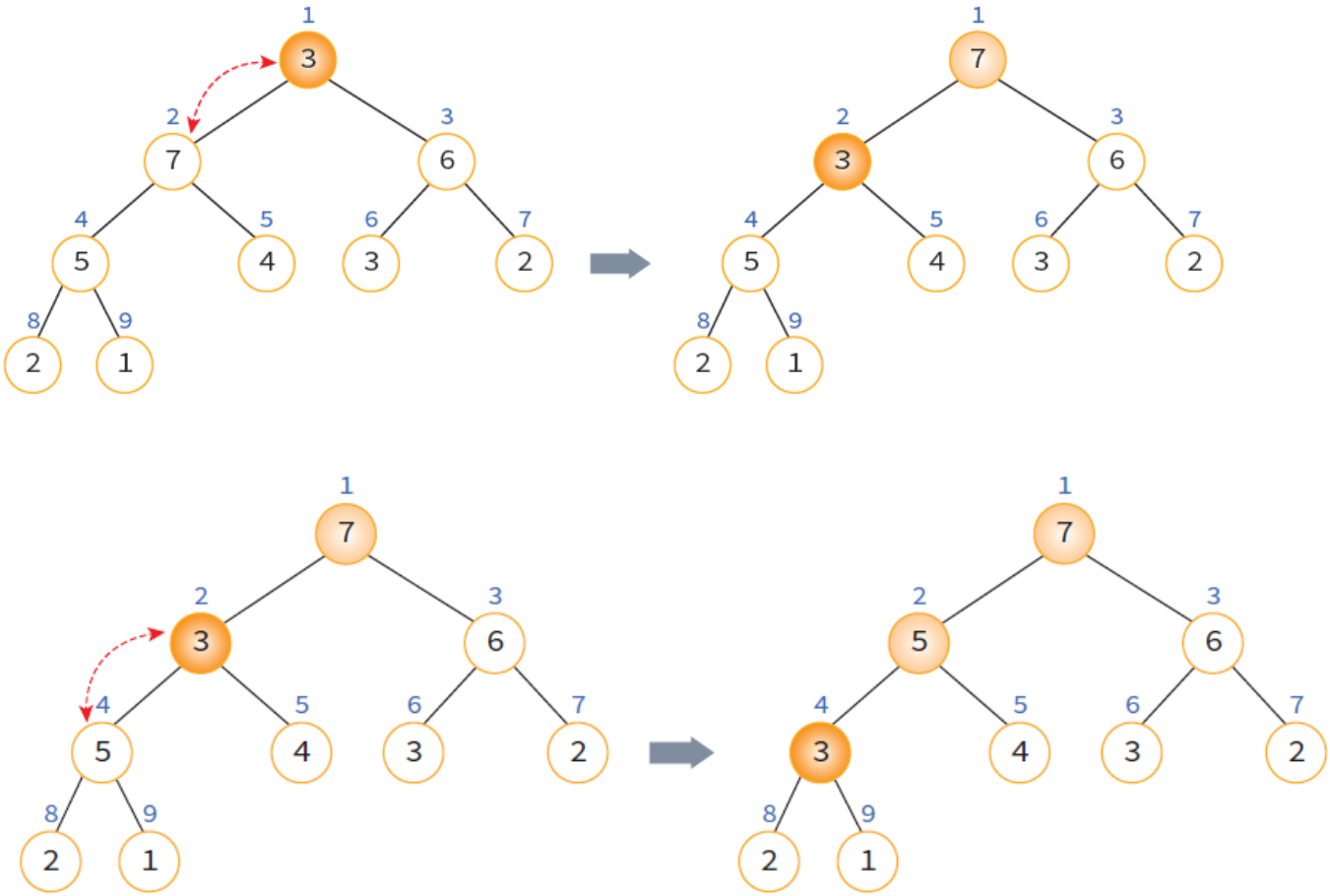


downheap 알고리즘





downheap 알고리즘



downheap 알고리즘


```
delete_max_heap(A):

item ← A[1];
A[1] ← A[heap_size];
heap_size←heap_size-1;
i ← 2;
while i ≤ heap_size do      // A[i/2] 와 max(A[i], A[i+1]) 비교하며 동작
        if i < heap_size and A[i+1] > A[i]
                then largest ← i+1;
                else largest ← i;
        if A[PARENT(largest)] > A[largest]   // A[PARENT(i)] == A[i/2]
                then break;
        A[PARENT(largest)] <-> A[largest];
        i ← CHILD(largest);                  // i = i*2;

return item;
```

삭제 함수 구현: heap1.c


```
40  // 삭제 함수
41  element delete_max_heap(HeapType* h)
42  {
43      int parent, child;
44      element item, temp;
45
46      item = h->heap[1];
47      temp = h->heap[(h->heap_size)--];
48      parent = 1;
49      child = 2;
50      while (child <= h->heap_size) {
51          // 현재 노드의 자식노드 중 더 작은 자식노드를 찾는다.
52          if ((child < h->heap_size) &&
53              (h->heap[child].key) < h->heap[child + 1].key)
54              child++;
55          if (temp.key >= h->heap[child].key) break;
56          // 한 단계 아래로 이동
57          h->heap[parent] = h->heap[child];
58          parent = child;
59          child *= 2;
60      }
61      h->heap[parent] = temp;
62      return item;
63  }
```

이외 프로그램: heap1.c (1/2)


```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENT 200

typedef struct {
    int key;
} element;
typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
} HeapType;

// 생성 함수
HeapType* create()
{
    return (HeapType*)malloc(sizeof(HeapType));
}

// 초기화 함수:
void init(HeapType* h)
{
    h->heap_size = 0;
}
```

이외 프로그램: heap1.c (2/2)


```c
int main(void)
{
    element e1 = { 10 }, e2 = { 5 }, e3 = { 30 };
    element e4, e5, e6;
    HeapType* heap;

    heap = create();     // 히프 생성
    init(heap); // 초기화

                // 삽입
    insert_max_heap(heap, e1);
    insert_max_heap(heap, e2);
    insert_max_heap(heap, e3);

    // 삭제
    e4 = delete_max_heap(heap);
    printf("< %d > ", e4.key);
    e5 = delete_max_heap(heap);
    printf("< %d > ", e5.key);
    e6 = delete_max_heap(heap);
    printf("< %d > \n", e6.key);

    free(heap);
    return 0;
}
```

```
< 30 > < 10 > < 5 >

--------------------------------
Process exited after 0.01257 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```

히프의 복잡도 분석

- 삽입 연산에서 최악의 경우, 루트 노드까지 올라가야 하므로 트리의 높이에 해당하는 비교 연산 및 이동 연산이 필요하다. ->O(logn)

- 삭제도 최악의 경우, 가장 아래 레벨까지 내려가야 하므로 역시 트리의 높이 만큼의 시간이 걸린다. ->O(logn)

9.5 히프 정렬

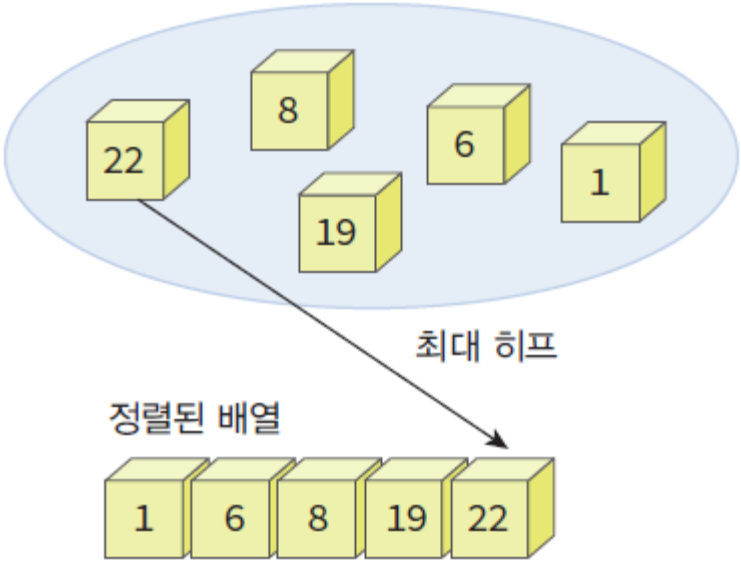
- 히프를 이용하면 데이터 정렬
  - 먼저 정렬해야 할 n개의 요소들을 최대 히프에 삽입
  - 한번에 하나씩 요소를 히프에서 삭제하여 배열에 역순(만약 최소히프일 경우에는 순서대로)저장하면 된다.

- 히프 정렬 복잡도
  - 하나의 요소를 히프에 삽입하거나 삭제할 때 시간이 O(logn) 소요되고
  - 요소의 개수가 n개이므로 전체 복잡도는 O(nlogn)
  - 다른 정렬 알고리즘(12장)에 비해 우수한 편

- 히프 정렬이 최대 유용한 경우
  - 전체 자료를 정렬하는 것이 아니라 가장 큰 값 몇 개만 필요할 경우

히프 정렬

- 히프를 이용하면 정렬 가능



히프 정렬 프로그램: heapsort.c

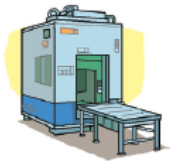

```
65  void heap_sort(element a[], int n)
66  {
67      int i;
68      HeapType* h;
69
70      h = create();
71      init(h);
72      for (i = 0; i<n; i++) {
73          insert_max_heap(h, a[i]);
74      }
75      for (i=(n-1); i >= 0; i--) {
76          a[i] = delete_max_heap(h);
77      }
78      free(h);
79  }
80
81  #define SIZE 8
82  int main(void)
83  {
84      int i;
85      element list[SIZE] = { 23, 56, 11, 9, 56, 99, 27, 34 };
86      heap_sort(list, SIZE);
87      for (i = 0; i < SIZE; i++) {
88          printf("%d ", list[i].key);
89      }
90      printf("\n");
91      return 0;
92  }
```

```
C:₩Users₩wjlee₩Downloads₩Test₩heapsort.exe
9 11 23 27 34 56 56 99

--------------------------------
Process exited after 0.06633 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```

9.6 머신 스케줄링

- 머신 스케줄링
  - 동일 기계 m개, 서로 다른 작업 시간을 가진 n개의 작업
  - 목표: m개의 기계를 풀가동하여 n개의 모든 작업을 최소 시간에 종료
- 최적의 해: 알고리즘 고안이 비교적 난해함
- 근사의 해: LPT(Longest Processing Time first) 알고리즘 (heap 이용)



LPT(longest processing time first)

- 최소 히프(작업 시간을 key)를 이용
  - 각 기계 노드[ID(기계 번호), Avail(작업시간(초기값=0))] 를 순서대로 최소 히프에 삽입
  - 긴 작업시간부터 정렬된 각 Job에 대해 아래를 수행
    - 최소히프(노드의 Avail 값에 의해 구성)에서 노드 삭제 후,
    - 노드의 기계에 작업을 할당(노드 Avail 값에 작업의 시간 합산)
    - 다시 최소히프에 노드를 삽입
  - 가장 큰 Key 값 노드의 시간이 최종 전체 종료시간

LPT(longest processing time first) 예

J1	J2	J3	J4	J5	J6	J7
8	7	6	5	3	2	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M1															
M2															
M3															

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M1															
M2															
M3															

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M1															
M2															
M3															

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M1															
M2															
M3															

.

.

.

LPT 최적 할당 예

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M1															
M2															
M3															

LPT 구현: lpt.c (1/4)

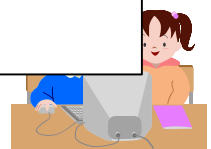

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENT 200

typedef struct {
    int id;
    int avail;
} element;

typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
} HeapType;

// 생성 함수
HeapType* create()
{
    return (HeapType*)malloc(sizeof(HeapType));
}

// 초기화 함수
void init(HeapType* h)
{
    h->heap_size = 0;
}
```



LPT 구현: lpt.c (2/4)


```c
void insert_min_heap(HeapType* h, element item)
{
    int i;
    i = ++(h->heap_size);

    // 트리를 거슬러 올라가면서 부모 노드와 비교하는 과정
    while ((i != 1) && (item.avail < h->heap[i / 2].avail)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item;      // 새로운 노드를 삽입
}

// 삭제 함수:
element delete_min_heap(HeapType* h)
{
    int parent, child;
    element item, temp;

    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
    while (child <= h->heap_size) {
        // 현재 노드의 자식노드중 더 작은 자식노드를 찾는다.
        if ((child < h->heap_size) &&
            (h->heap[child].avail) > h->heap[child + 1].avail)
            child++;
        if (temp.avail < h->heap[child].avail) break;
        // 한 단계 아래로 이동:
        h->heap[parent] = h->heap[child];
        parent = child;
        child *= 2;
    }
    h->heap[parent] = temp;
    return item;
}
```

LPT 구현: lpt.c (3/4)


```c
#define JOBS 7
#define MACHINES 3

int main(void)
{
    int jobs[JOBS] = { 8, 7, 6, 5, 3, 2, 1 };   // 작업은 정렬되어 있다고 가정
    element m = { 0, 0 };
    HeapType* h;
    int i;

    h = create();
    init(h);

    // 여기서 avail 값은 기계가 사용 가능하게 되는 시간이다.
    for (i = 0; i<MACHINES; i++) {
        m.id = i + 1;
        m.avail = 0;
        insert_min_heap(h, m);
    }
    // 최소 히프에서 기계를 꺼내서 작업을 할당하고 사용가능 시간을 증가 시킨 후에
    // 다시 최소 히프에 추가한다.
    for (i = 0; i< JOBS; i++) {
        m = delete_min_heap(h);
        printf("JOB %d을 시간 =%d부터 시간 =%d까지 기계 %d번에 할당한다. \n",
            i, m.avail, m.avail + jobs[i]-1, m.id);
        m.avail += jobs[i];
        insert_min_heap(h, m);
    }
    return 0;
}
```

LPT 구현: lpt.c (4/4)


```
JOB 0을 시간=0부터 시간=7까지 기계 1번에 할당한다.
JOB 1을 시간=0부터 시간=6까지 기계 2번에 할당한다.
JOB 2을 시간=0부터 시간=5까지 기계 3번에 할당한다.
JOB 3을 시간=6부터 시간=10까지 기계 3번에 할당한다.
JOB 4을 시간=7부터 시간=9까지 기계 2번에 할당한다.
JOB 5을 시간=8부터 시간=9까지 기계 1번에 할당한다.
JOB 6을 시간=10부터 시간=10까지 기계 2번에 할당한다.

--------------------------------
Process exited after 0.04455 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
M1															
M2															
M3															

9.7 허프만 코드

- 허프만 코딩 트리
  - 각 글자의 빈도가 알려져 있는 메시지의 내용을 압축하기 위해 글자 코딩에 사용하는 이진트리
- 허프만 코딩
  - 글자 빈도 수에 따라 가변-길이 코딩을 수행
  - 빈도 수가 큰 글자: 짧은 길이 비트 코드 할당, 작은 글자: 긴 길이 비트 코드 할당

빈도수 분석

A	80
B	16
C	32
D	36
E	123
F	22
G	26
H	51
I	71
...	
Z	1

글자의 빈도수

- 예를 들어보자. 만약 텍스트가 e, t, n, i, s의 5개의 글자로만 이루어졌다고 가정하고 각 글자의 빈도수가 다음과 같다고 가정하자.
- 동일한 3비트 코드 사용 시: 45*3 = 135비트 필요

글자	비트 코드	빈도수	비트 수
e	00	15	30
t	01	12	24
n	11	8	16
i	100	6	18
s	101	4	12
합계		45	**88**

허프만 코드

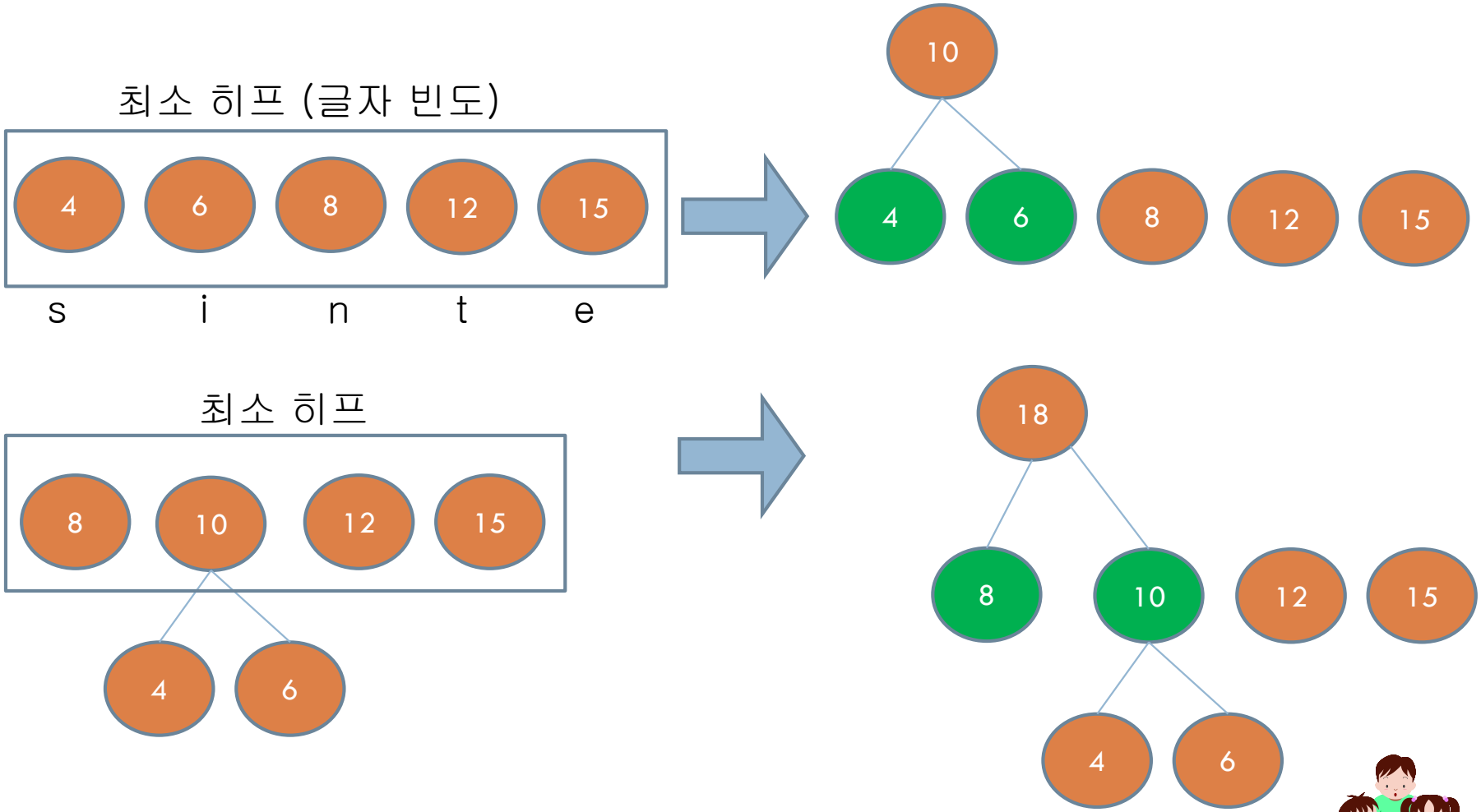
- 허프만 코드 특성
  - 앞의 코딩 예처럼, 2 또는 3 비트 코딩 시에, 3 비트 코드의 앞 2 비트가 2 비트 코드와 같지 않아야 한다.

글자	비트 코드
e	00
t	01
n	11
i	100
s	101
합계	

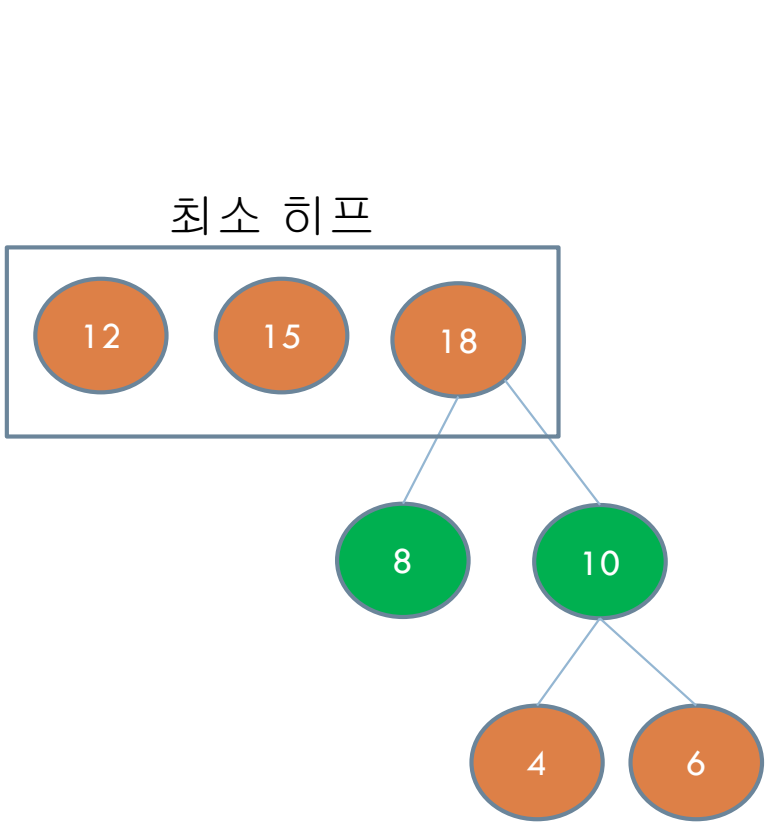
'100' 또는 '101'의 앞 2 비트인 '10' 비트는 두 비트 코드 값들('00', '01', '10')과 다름

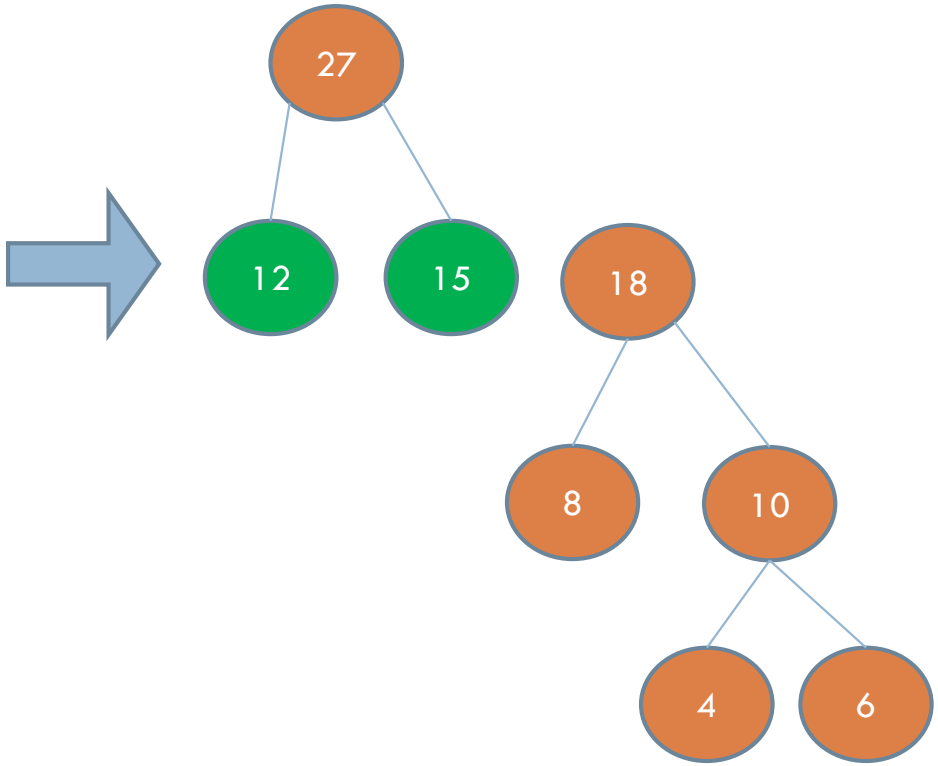
  - 이러한 특성을 만족시키는 허프만 코딩에는 허프만 코딩트리를 사용
  - 허프만 코딩트리 생성에 히프 자료구조를 사용한다

허프만 코드 생성 절차


최소 히프 (글자 빈도)
4
6
8
12
15
s
i
n
t
e
10
4
6
8
12
15
최소 히프
8
10
12
15
4
6
18
8
10
12
15
4
6

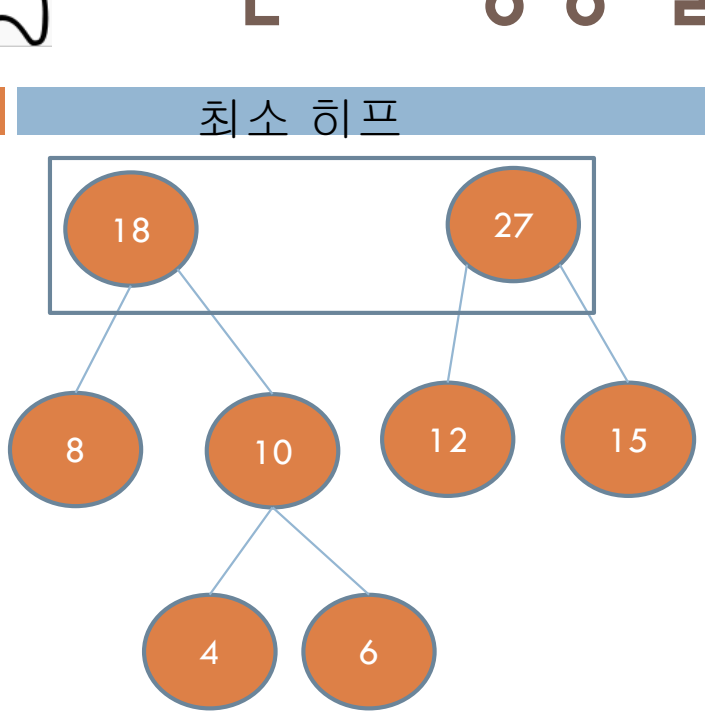
허프만 코드 생성 절차

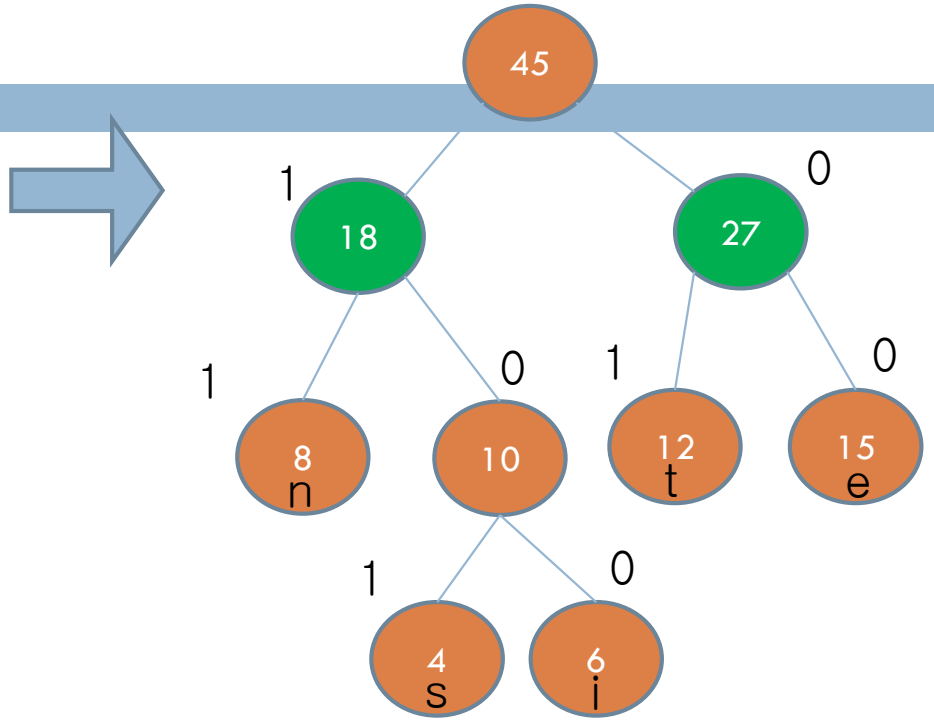




허프만 코드 생성 절차

최소 히프





글자	비트 코드	빈도수	비트 수
e	00	15	30
t	01	12	24
n	11	8	16
i	100	6	18
s	101	4	12
합계			88

허프만 코드 구현: huffman.c (1/6)


```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENT 200

typedef struct TreeNode {
    int weight;
    char ch;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;

typedef struct {
    TreeNode* ptree;
    char ch;
    int key;
} element;

typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
} HeapType;

// 생성 함수
HeapType* create()
{
    return (HeapType*)malloc(sizeof(HeapType));
}
// 초기화 함수
void init(HeapType* h)
{
    h->heap_size = 0;
}
```


허프만 코드 구현: huffman.c (2/6)


```c
// 현재 요소의 개수가 heap_size인 히프 h에 item을 삽입한다.
// 삽입 함수
void insert_min_heap(HeapType* h, element item)
{
    int i;
    i = ++(h->heap_size);

    // 트리를 거슬러 올라가면서 부모 노드와 비교하는 과정
    while ((i != 1) && (item.key < h->heap[i / 2].key)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item;      // 새로운 노드를 삽입
}

// 삭제 함수:
element delete_min_heap(HeapType* h)
{
    int parent, child;
    element item, temp;

    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
```

허프만 코드 구현: huffman.c (3/6)

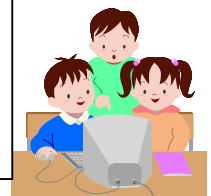

```
59      while (child <= h->heap_size) {
60          // 현재 노드의 자식노드중 더 작은 자식노드를 찾는다.
61          if ((child < h->heap_size) &&
62              (h->heap[child].key) > h->heap[child + 1].key)
63              child++;
64          if (temp.key < h->heap[child].key) break;
65          // 한 단계 아래로 이동
66          h->heap[parent] = h->heap[child];
67          parent = child;
68          child *= 2;
69      }
70      h->heap[parent] = temp;
71      return item;
72  }
73
74  // 이진 트리 생성 함수
75  TreeNode* make_tree(TreeNode* left,
76      TreeNode* right)
77  {
78      TreeNode* node =
79          (TreeNode*)malloc(sizeof(TreeNode));
80      node->left = left;
81      node->right = right;
82      return node;
83  }
```

허프만 코드 구현: huffman.c (4/6)


```
85  // 이진 트리 제거 함수
86  void destroy_tree(TreeNode* root)
87  {
88      if (root == NULL) return;
89      destroy_tree(root->left);
90      destroy_tree(root->right);
91      free(root);
92  }
93
94  int is_leaf(TreeNode* root)
95  {
96      return !(root->left) && !(root->right);
97  }
98
99  void print_array(int codes[], int n)
100 {
101     int i;
102     for (i = 0; i < n; i++)
103         printf("%d", codes[i]);
104     printf("\n");
105 }
106
107 void print_codes(TreeNode* root, int codes[], int top)
108 {
109     // 1을 저장하고 순환호출한다.
110     if (root->left) {
111         codes[top] = 1;
112         print_codes(root->left, codes, top + 1);
113     }
```

허프만 코드 구현: huffman.c (5/6)


```
115         // 0을 저장하고 순환호출한다.
116         if (root->right) {
117             codes[top] = 0;
118             print_codes(root->right, codes, top + 1);
119         }
120
121         // 단말노드이면 코드를 출력한다.
122         if (is_leaf(root)) {
123             printf("%c: ", root->ch);
124             print_array(codes, top);
125         }
126     }
127
128     // 허프만 코드 생성 함수
129     void huffman_tree(int freq[], char ch_list[], int n)
130     {
131         int i;
132         TreeNode *node, *x;
133         HeapType* heap;
134         element e, e1, e2;
135         int codes[100];
136         int top = 0;
137
138         heap = create();
139         init(heap);
140         for (i=0; i<n; i++) {
141             node = make_tree(NULL, NULL);
142             e.ch = node->ch = ch_list[i];
143             e.key = node->weight = freq[i];
144             e.ptree = node;
145             insert_min_heap(heap, e);
146         }
```



허프만 코드 구현: huffman.c (6/6)


```
147     for (i=1; i<n; i++) {
148         // 최소값을 가지는 두개의 노드를 삭제
149         e1 = delete_min_heap(heap);
150         e2 = delete_min_heap(heap);
151         // 두개의 노드를 합친다.
152         x = make_tree(e1.ptree, e2.ptree);
153         e.key = x->weight = e1.key + e2.key;
154         e.ptree = x;
155         printf("%d+%d->%d \n", e1.key, e2.key, e.key);
156         insert_min_heap(heap, e);
157     }
158     e = delete_min_heap(heap); // 최종 트리
159     print_codes(e.ptree, codes, top);
160     destroy_tree(e.ptree);
161     free(heap);
162 }
163
164 int main(void)
165 {
166     char ch_list[] = { 's', 'i', 'n', 't', 'e' };
167     int freq[] = { 4, 6, 8, 12, 15 };
168     huffman_tree(freq, ch_list, 5);
169     return 0;
170 }
```

```
4+6->10
8+10->18
12+15->27
18+27->45
n: 11
s: 101
i: 100
t: 01
e: 00

--------------------------------
Process exited after 0.5926 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```