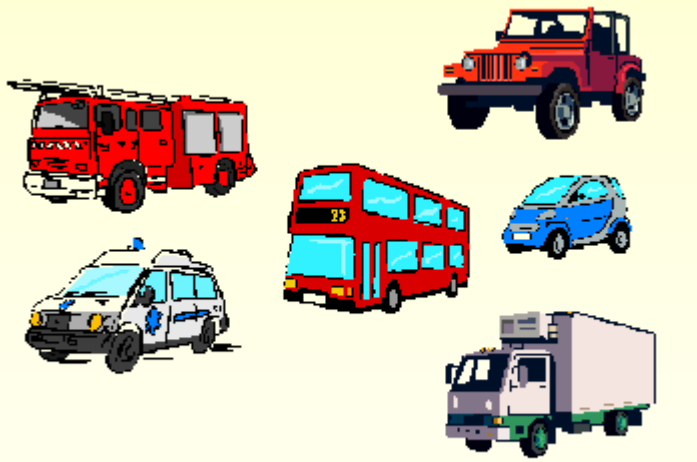
CHAP 8:우선순위 큐

우선순위 큐

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



우선순위 높음 우선순위 낮음

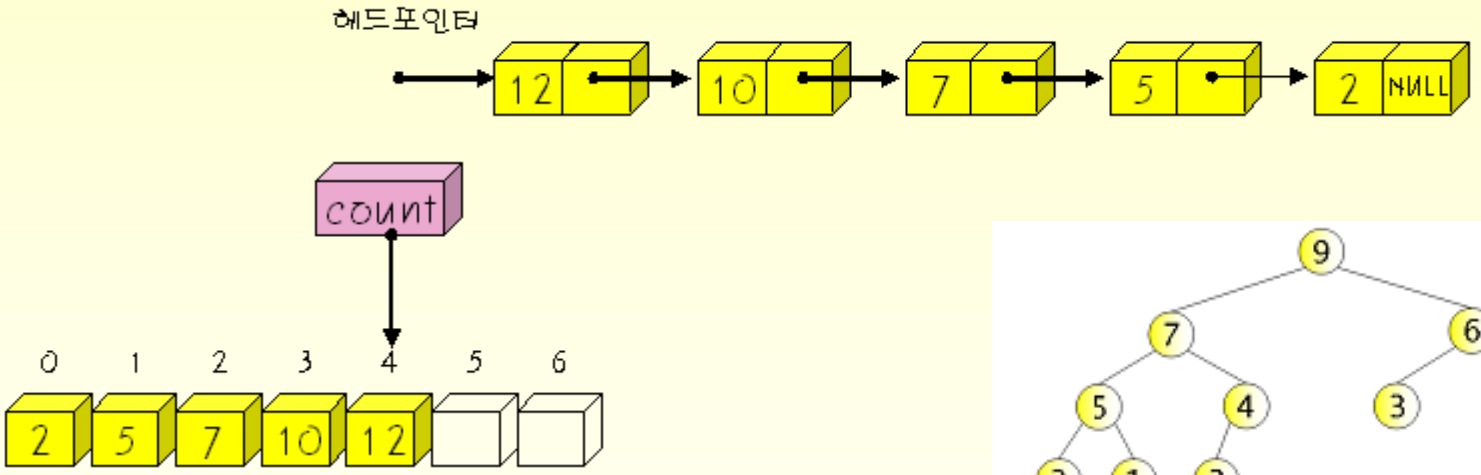
우선순위큐 ADT

- 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.

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

우선순위 큐 구현방법

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

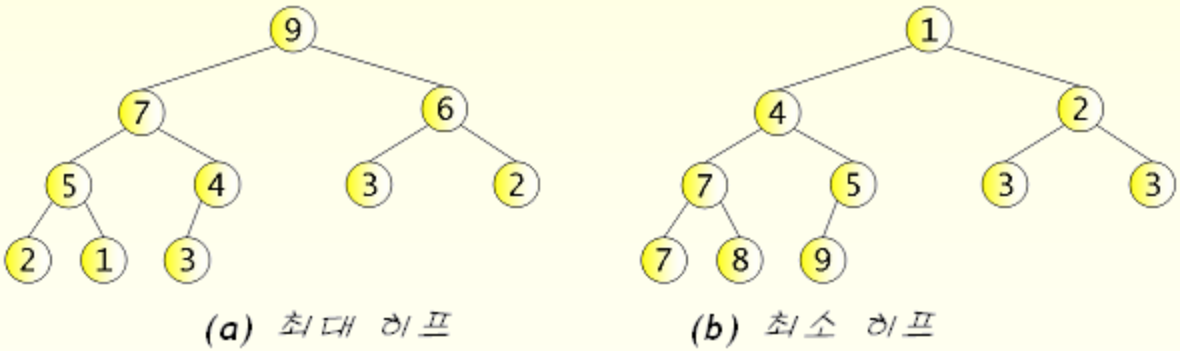


우선순위큐 구현방법

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)

히프(heap)란?

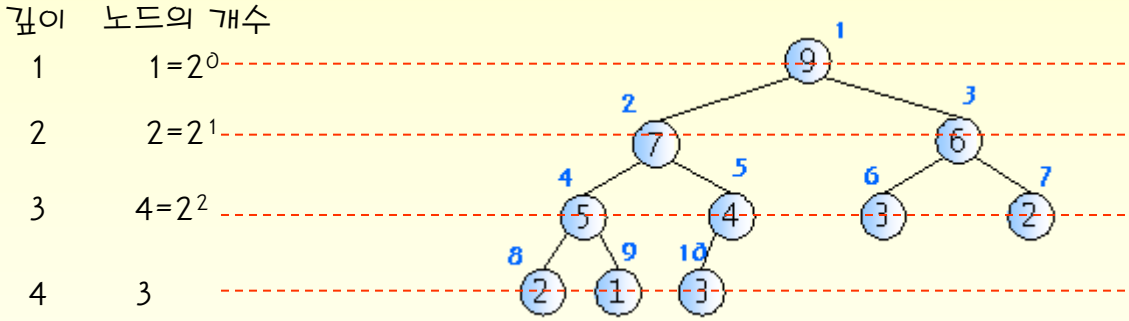
- 히프란 노드들이 저장하고 있는 키들이 다음과 같은 식을 만족하는 완전이진트리
- 최대 히프(max heap)
  - 부모 노드의 키값이 자식 노드의 키값보다 크거나 같은 완전 이진 트리
  - *key(부모노드) ≥key(자식노드)*
- 최소 히프(min heap)
  - 부모 노드의 키값이 자식 노드의 키값보다 작거나 같은 완전 이진 트리
  - *key(부모노드) ≤key(자식노드)*



(a) 최대 히프 (b) 최소 히프

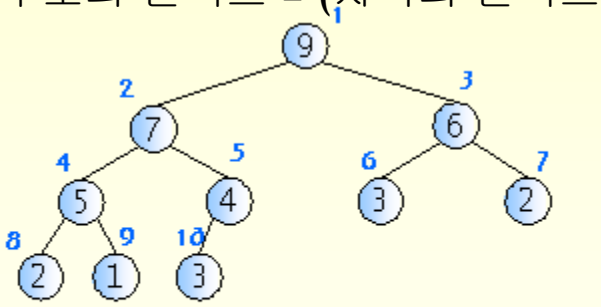
히프의 높이

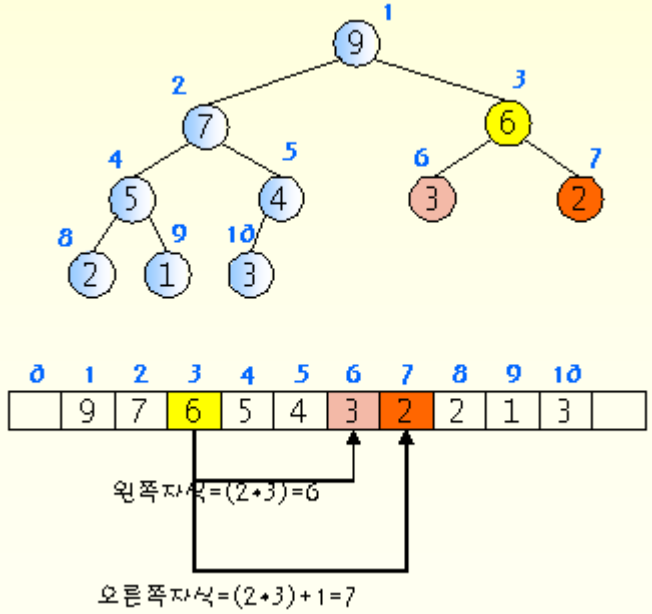
- n개의 노드를 가지고 있는 히프의 높이는 O(logn)
  - 히프는 완전이진트리
  - 마지막 레벨 h을 제외하고는 각 레벨 i에 2^{i-1}개의 노드 존재



히프의 구현방법

- 히프는 배열을 이용하여 구현
  - 완전이진트리이므로 각 노드에 번호를 붙일 수 있다
  - 이 번호를 배열의 인덱스라고 생각
- 부모노드와 자식노드를 찾기가 쉽다.
  - 왼쪽 자식의 인덱스 = (부모의 인덱스)*2
  - 오른쪽 자식의 인덱스 = (부모의 인덱스)*2 + 1
  - 부모의 인덱스 = (자식의 인덱스)/2



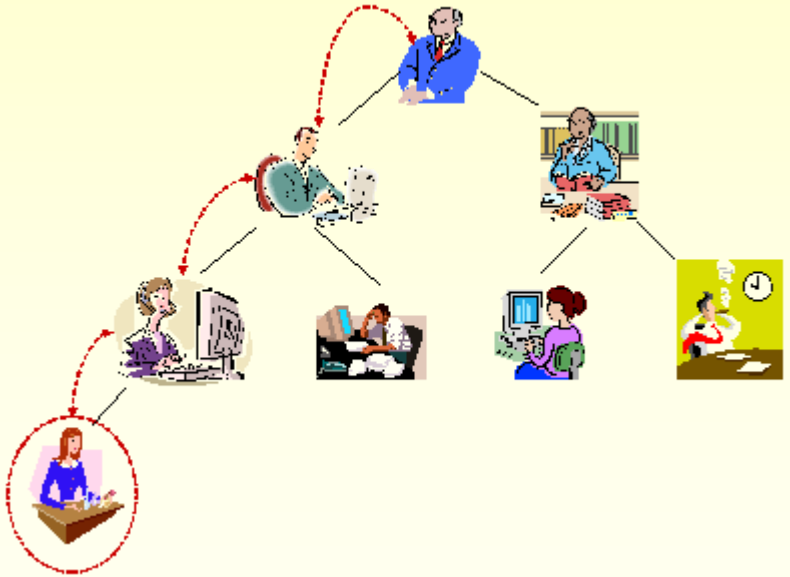


히프에서의 삽입

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

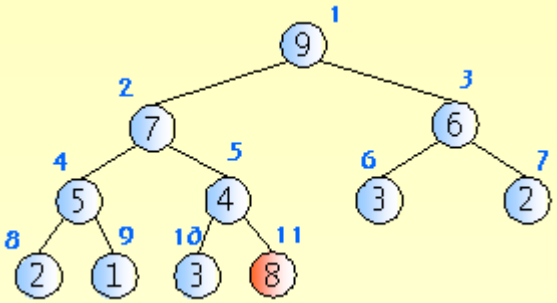
(1)히프에 새로운 요소가 들어 오면, 일단 새로운 노드를 히프의 마지막 노드에 이어서 삽입

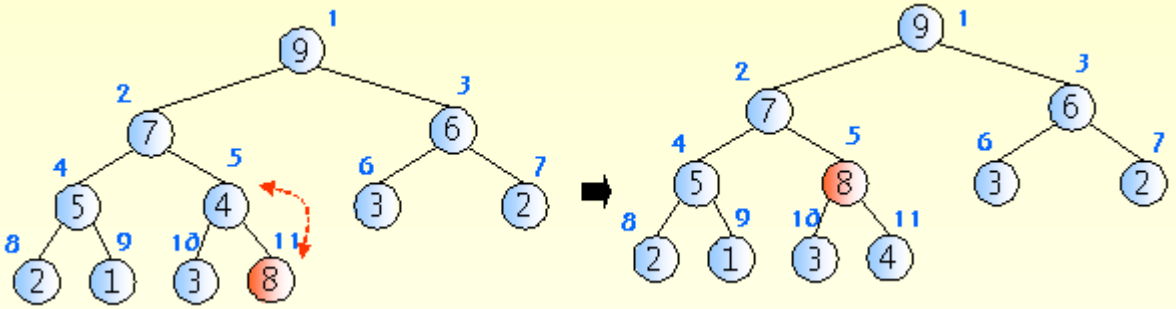
(2)삽입 후에 새로운 노드를 부모 노드들과 교환해서 히프의 성질을 만족

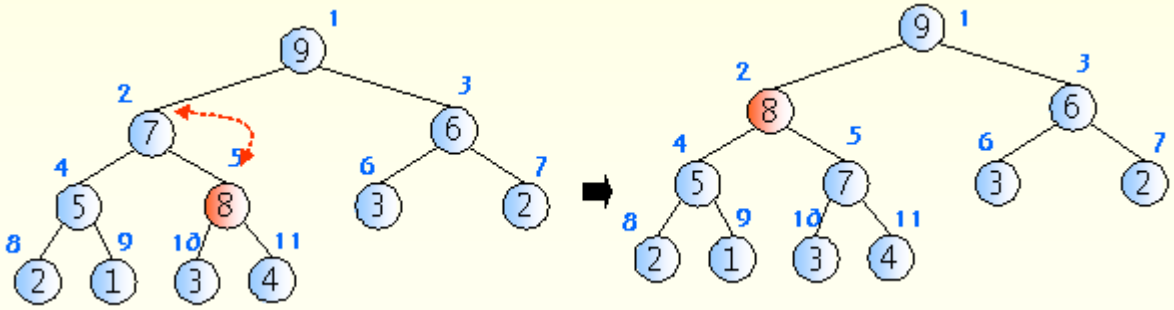


upheap 연산

- 새로운 키 k의 삽입연산후 히프의 성질이 만족되지 않을 수 있다
- upheap는 삽입된 노드로부터 루트까지의 경로에 있는 노드들을 k와 비교, 교환함으로써 히프의 성질을 복원한다.
- 키 k가 부모노드보다 작거나 같으면 upheap는 종료한다
- 히프의 높이가 O(logn)이므로 upheap연산은 O(logn)이다.







upheap 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];
              i ← PARENT(i);
```

삽입 프로그램

```
// 현재 요소의 개수가 heap_size인 히프 h에 item을 삽입한다.
// 삽입 함수
void insert_max_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
```

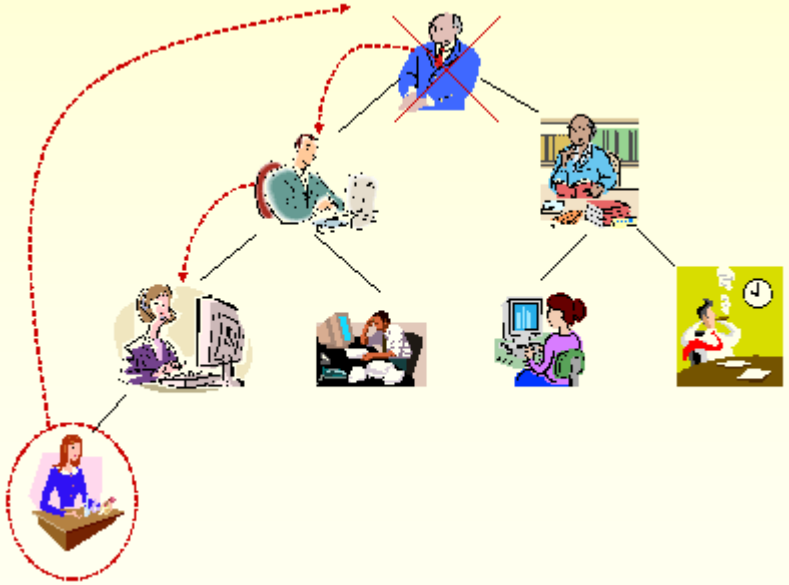
히프에서의 삭제

- 최대 히프에서의 삭제는 가장 큰 키값을 가진 노드를 삭제하는 것을 의미

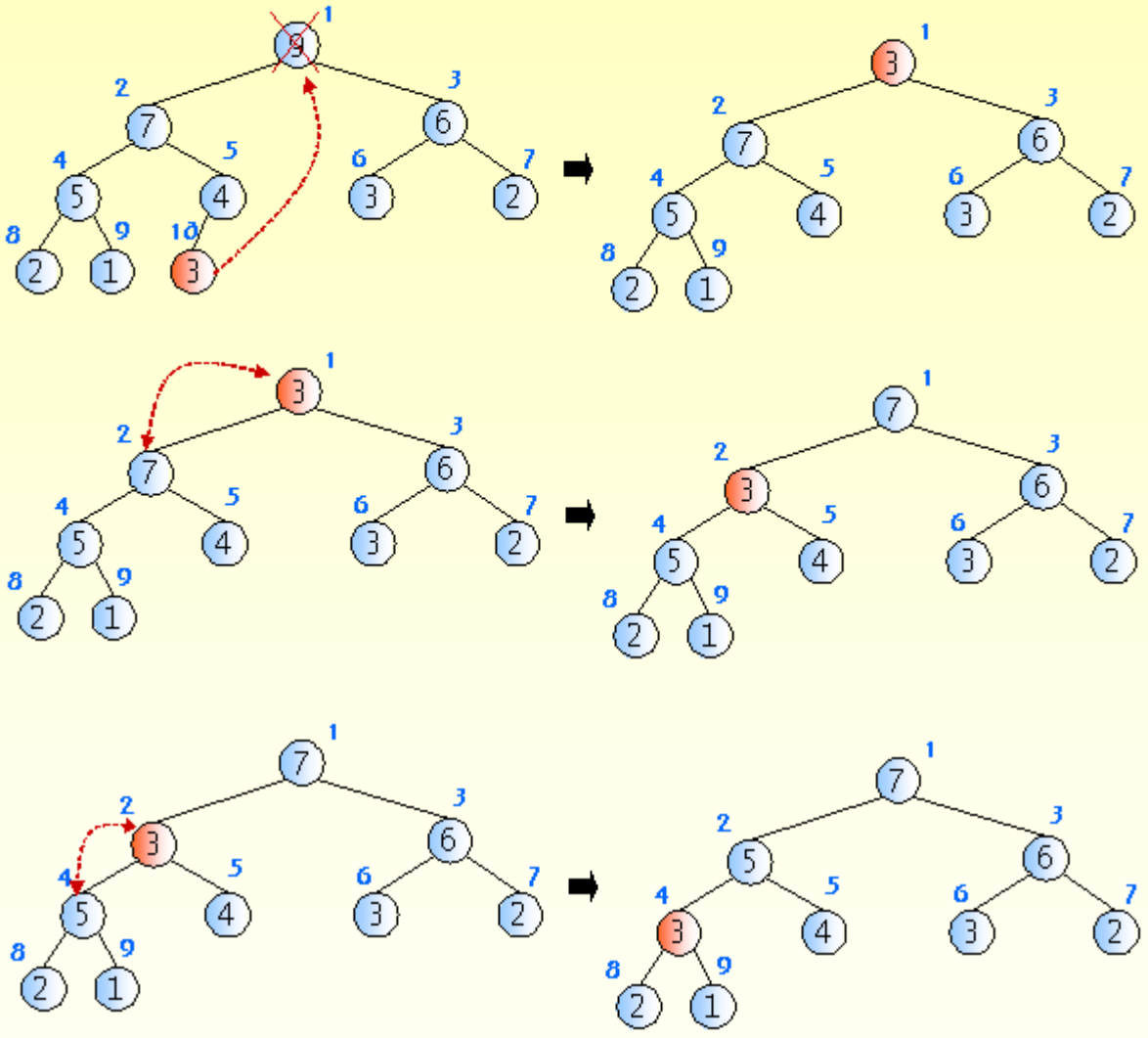
-> 따라서 루트 노드가 삭제된다.

- 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.
(1)루트에서부터 단말 노드까지의 경로에 있는 노드들을 교환하여 히프 성질을 만족시킨다.



downheap 알고리즘



- 히프의 높이가 O(logn)이므로 downheap연산은 O(logn)이다.

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
        if i < heap_size and A[LEFT(i)] > A[RIGHT(i)]
                then largest ← LEFT(i);
                else largest ← RIGHT(i);
        if A[PARENT(largest)] > A[largest]
                then break;
        A[PARENT(largest)] ↔ A[largest];
        i ← CHILD(largest);

return item;
```

삭제 프로그램

```
// 삭제 함수
element delete_max_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].key) < h->heap[child+1].key)
               child++;
           if( temp.key >= h->heap[child].key ) break;
           // 한단계 아래로 이동
           h->heap[parent] = h->heap[child];
           parent = child;
           child *= 2;

    h->heap[parent] = temp;
    return item;
```

전체 프로그램

```
#include <stdio.h>
#define MAX_ELEMENT 200
typedef struct {
        int key;
} element;
typedef struct {
        element heap[MAX_ELEMENT];
        int heap_size;
} HeapType;
// 초기화 함수
init(HeapType *h)
{
        h->heap_size =0;
}
```

전체 프로그램

```
void main()
{
        element e1=10, e2=5, e3=30;
        element e4, e5, e6;
        HeapType heap;   // 히프 생성
        init(&heap);              // 초기화
        // 삽입
        insert_max_heap(&heap, e1);
        insert_max_heap(&heap, e2);
        insert_max_heap(&heap, e3);
//      print_heap(&heap);
        // 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 > ", e6.key);
}
```

< 30 > < 10 > < 5 >

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)

히프 정렬

- 히프를 이용하면 정렬 가능
- 먼저 정렬해야 할 n개의 요소들을 최대 히프에 삽입
- 한번에 하나씩 요소를 히프에서 삭제하여 저장하면 된다.
- 삭제되는 요소들은 값이 증가되는 순서(최소히프의 경우)
- 하나의 요소를 히프에 삽입하거나 삭제할 때 시간이 O(logn) 만큼 소요되고 요소의 개수가 n개이므로 전체적으로 O(nlogn)시간이 걸린다. (빠른편)
- 히프 정렬이 최대로 유용한 경우는 전체 자료를 정렬하는 것이 아니라 가장 큰 값 몇 개만 필요할 때이다.
- 이렇게 히프를 사용하는 정렬 알고리즘을 히프 정렬이라고 한다.

히프 정렬 프로그램

```
// 우선 순위 큐인 히프를 이용한 정렬
void heap_sort(element a[], int n)
{
        int i;
        HeapType h;

        init(&h);
        for(i=0;i<n;i++){
                insert_max_heap(&h, a[i]);
        }
        for(i=(n-1);i>=0;i--){
                a[i] = delete_max_heap(&h);
        }
}
```

이산 이벤트 시뮬레이션

- 이산 이벤트 시뮬레이션에서 모든 시간의 진행은 이벤트의 발생에 의해서 이루어진다.

- 우선순위 큐를 이용하여 이벤트를 저장하고 이벤트의 발생시각을 우선순위로 하여 이벤트를 처리하는 간단한 시뮬레이션을 작성하여 보자.

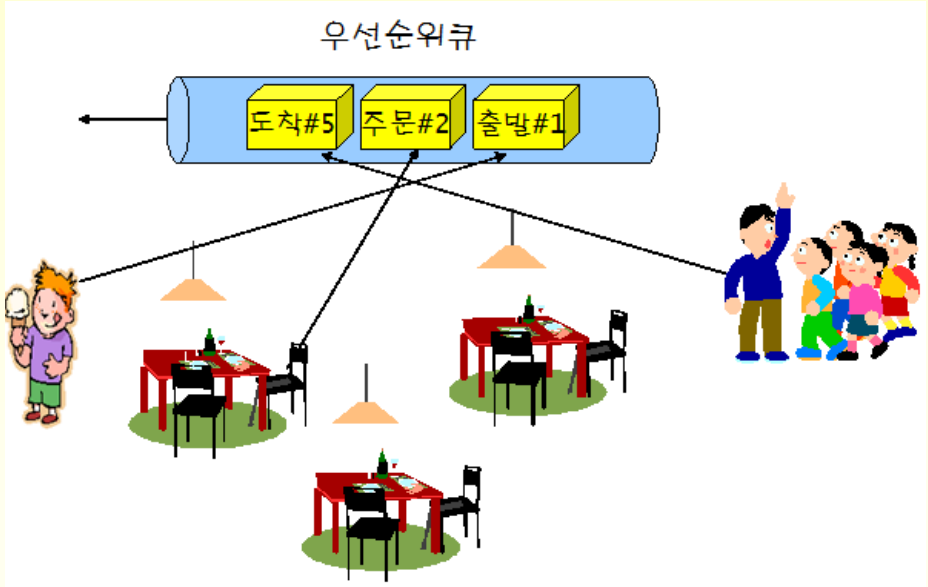


그림 8.18 아이스크림 가게 시뮬레이션

이산 이벤트 시뮬레이션

```
#define ARRIVAL 1
#define ORDER 2
#define LEAVE 3
int free_seats=10;
double profit=0.0;
#define MAX_ELEMENT 100
typedef struct {
        int type;    // 이벤트의 종류
        int key;     // 이벤트가 일어난 시각
        int number;          // 고객의 숫자
} element;
typedef struct {
        element heap[MAX_ELEMENT];
        int heap_size;
} HeapType;
```

이산 이벤트 시뮬레이션

```
// 초기화 함수
void init(HeapType *h)
{
        h->heap_size =0;
}
//
int is_empty(HeapType *h)
{
        if( h->heap_size == 0 )
                return TRUE;
        else
                return FALSE;
}
```

이산 이벤트 시뮬레이션

```
// 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
}
```

이산 이벤트 시뮬레이션

```
// 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 ){
        if( ( child < h->heap_size ) &&
        (h->heap[child].key) > h->heap[child+1].key)
        child++;
        if( temp.key <= h->heap[child].key ) break;
        h->heap[parent] = h->heap[child];
        parent = child;
        child *= 2;
}
h->heap[parent] = temp;
return item;
}
```

이산 이벤트 시뮬레이션

```
// 0에서 n사이의 정수 난수 생성 함수
int random(int n)
{
return rand() % n;
}
// 자리가 가능하면 빈 자리수를 사람수만큼 감소시킨다.
int is_seat_available(int number)
{
        printf("%d명의 고객 도착\n", number);
        if( free_seats >= number ){
                free_seats -= number;
                return TRUE;
        }
        else {
                printf("자리가 없어서 떠남\n");
                return FALSE;
        }
}
```

이산 이벤트 시뮬레이션

```
// 주문을 받으면 순익을 나타내는 변수를 증가시킨다.
void order(int scoops)
{
        printf("아이스크림 %d개 주문 받음\n", scoops);
        profit += 0.35 * scoops;
}
// 고객이 떠나면 빈자리수를 증가시킨다.
void leave(int number)
{
        printf("%d명이 매장을 떠남\n", number);
        free_seats += number;
}
```

이산 이벤트 시뮬레이션

```
// 이벤트를 처리한다.
void process_event(HeapType *heap, element e)
{
        int i=0;
        element new_event;

        printf("현재 시간=%d\n", e.key);
        switch(e.type){
        case ARRIVAL:
                // 자리가 가능하면 주문 이벤트를 만든다.
                if( is_seat_available(e.number) ){
                        new_event.type=ORDER;
                        new_event.key = e.key + 1 + random(4);
                        new_event.number=e.number;
                        insert_min_heap(heap, new_event);
                }
                break;
```

이산 이벤트 시뮬레이션

```
case ORDER:
                    // 사람수만큼 주문을 받는다.
                    for (i = 0; i < e.number; i++){
                              order(1 + random(3));
                    }
                    // 매장을 떠나는 이벤트를 생성한다.
                    new_event.type=LEAVE;
                    new_event.key = e.key + 1 + random(10);
                    new_event.number=e.number;
                    insert_min_heap(heap, new_event);
                    break;
          case LEAVE:
                    // 고객이 떠나면 빈자리수를 증가시킨다.
                    leave(e.number);
                    break;
          }
}
```

```
int main()
{
element event;
HeapType heap;
unsigned int t = 0;
init(&heap);
// 처음에 몇개의 초기 이벤트를 생성시킨다.
while (t < 5) {
t += random(6);
            event.type = ARRIVAL;
            event.key = t;
            event.number = 1 + random(4);
            insert_min_heap(&heap, event);
}
while (!is_empty(&heap)) {
            event = delete_min_heap(&heap);
            process_event(&heap, event);
}
printf("전체 순이익은 =%f입니다.\n ", profit);
}
```

이산 이벤트 시뮬레이션

```
현재 시간=0
3명의 고객 도착
현재 시간=1
4명의 고객 도착
현재 시간=4
아이스크림 1개 주문 받음
아이스크림 3개 주문 받음
아이스크림 3개 주문 받음
현재 시간=4
아이스크림 1개 주문 받음
아이스크림 1개 주문 받음
아이스크림 1개 주문 받음
아이스크림 2개 주문 받음
현재 시간=4
2명의 고객 도착
현재 시간=5
아이스크림 3개 주문 받음
아이스크림 2개 주문 받음
현재 시간=6
4명의 고객 도착
자리가 없어서 떠남
현재 시간=6
4명이 매장을 떠남
현재 시간=7
2명이 매장을 떠남
현재 시간=10
3명이 매장을 떠남
전체 순이익은 =5.950000입니다.
```

허프만 코드

- 이진 트리는 각 글자의 빈도가 알려져 있는 메시지의 내용을 압축하는데 사용될 수 있다.
- 이런 종류의 이진트리를 허프만 코딩 트리라고 부른다.

빈도수 분석

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

그림 8.19 허프만 코드

글자의 빈도수

- 예를 들어보자. 만약 텍스트가 e, t, n, i, s의 5개의 글자로만 이루어졌다고 가정하고 각 글자의 빈도수가 다음과 같다고 가정하자.

글자	빈도수
e	15
t	12
n	8
i	6
s	4

글자	비트코드	빈도수	비트수
e	00	15	30
t	01	12	12
n	10	8	16
i	110	6	18
s	111	4	12
합계			88

허프만 코드 생성 절차

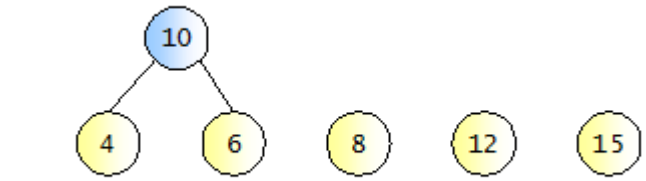


그림 8.20 허프만 코드 생성 과정 #1

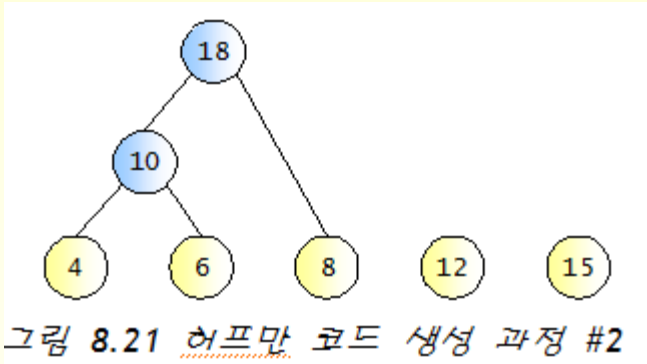


그림 8.21 허프만 코드 생성 과정 #2

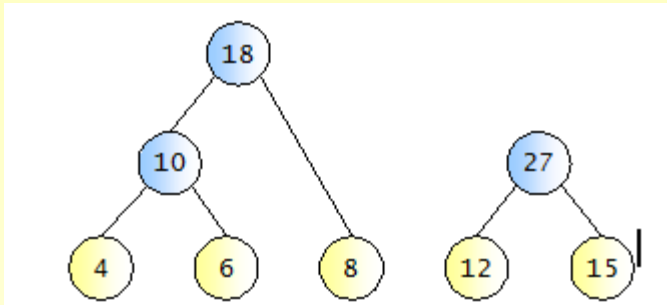


그림 8.22 허프만 코드 생성 과정 #3

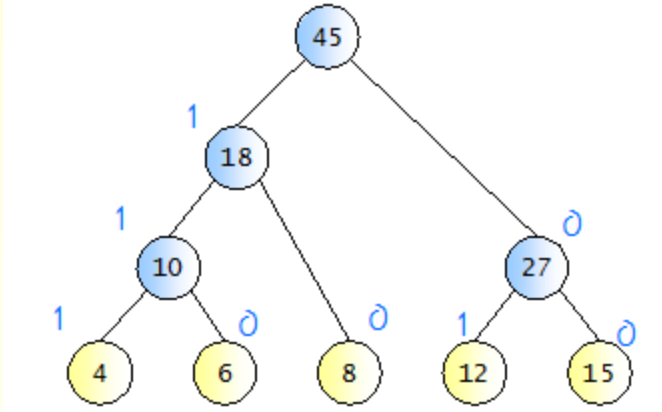


그림 8.23 허프만 코드 생성 과정 #4

허프만 코드 프로그램

```
#define MAX_ELEMENT 100
typedef struct TreeNode {
        int weight;
        struct TreeNode *left_child;
        struct TreeNode *right_child;
} TreeNode;
typedef struct {
        TreeNode *ptree;
        int key;
} element;
typedef struct {
        element heap[MAX_ELEMENT];
        int heap_size;
} HeapType;
```

허프만 코드 프로그램

```
// 초기화 함수
init(HeapType *h)
{
        h->heap_size =0;
}
// 삽입 함수
void insert_min_heap(HeapType *h, element item)
{
// 프로그램 8.5 참조
}
// 삭제 함수
element delete_min_heap(HeapType *h)
{
// 프로그램 8.5 참조
}
```

```
// 이진 트리 생성 함수
TreeNode *make_tree(TreeNode *left, TreeNode *right)
{
        TreeNode *node= (TreeNode *)malloc(sizeof(TreeNode));
        if( node == NULL ){
                fprintf(stderr,"메모리 에러\n");
                exit(1);
        }
        node->left_child = left;
        node->right_child = right;
        return node;
}
// 이진 트리 제거 함수
void destroy_tree(TreeNode *root)
{
        if( root == NULL ) return;
        destroy_tree(root->left_child);
        destroy_tree(root->right_child);
        free(root);
}
```

허프만 코드 프로그램

```
// 허프만 코드 생성 함수
void huffman_tree(int freq[], int n)
{
        int i;
        TreeNode *node, *x;
        HeapType heap;
        element e, e1, e2;
        init(&heap);
        for(i=0;i<n;i++){
                node = make_tree(NULL, NULL);
                e.key = node->weight = freq[i];
                e.ptree = node;
                insert_min_heap(&heap, e);
        }
```

허프만 코드 프로그램

```
for(i=1;i<n;i++){
                    // 최소값을 가지는 두개의 노드를 삭제
                    e1 = delete_min_heap(&heap);
                    e2 = delete_min_heap(&heap);
                    // 두개의 노드를 합친다.
                    x = make_tree(e1.ptree, e2.ptree);
                    e.key = x->weight = e1.key + e2.key;
                    e.ptree = x;
                    insert_min_heap(&heap, e);
        }
        e = delete_min_heap(&heap); // 최종 트리
        destroy_tree(e.ptree);
}
```

허프만 코드 프로그램

```
// 주함수
void main()
{
        int freq[] = { 15, 12, 8, 6, 4 };
        huffman_tree(freq, 5);
}
```