<9장, 우선순위큐(PRIORITY QUEUE), 201910783, 김성현>

9.1 우선순위 큐 추상 데이터 타입	9.2. 우선순위 큐의 구현 방법	9.3 히프	9.4 히프의 구현	9.5 히프 정렬
			9.7. 허프만 코드	9.6 머신 스케줄링

p.322~324. 9.1 우선순위 큐 추상 데이터 타입.

- 우선순위 큐란?
- 우선순위 큐의 ADT

ex) 도로에서 우선 순위를 가지는 응급차량 등.

ex) 네트워크 패킷 중 네트워크 관리와 관련된 패킷의 우선 순위

ex) 운영체제에서 시스템 process의 우선 순위.

• 우선순위 큐(priority queue)란?

→ 우선 순위 = key → 책에서는 숫자가 클수록 우선순위, key가 높다는 이야기로 설명될 수 있다.

□ 우선순위 큐: 우선순위(key)를 가진 요소들을 저장하는 큐로서 제거 동작이 우선순위 순서(FIFO 아님!!) 로 수행되는 자료구조. (제거 동작이 매우 중요!! FIFO가 아닌 우선 순위 순서로 제거!)

↳ key가 가장 큰 놈이 먼저 나가게 된다!! 중요한 데이터 먼저, 우선 순위 낮은 데이터는 나중에

□ 응용 분야

- 통신 라우터(router)에서 우선 순위를 사용하게 되며 전송량(bandwidth)를 관리.
  - 패킷들을 어디로 보낼지 결정하는 소프트웨어 - 하드웨어.
- 운영체제 커널에서 우선 순위를 가진 작업(task)들을 스케줄링(scheduling)할 때 사용.
  - 작업: 컴퓨터의 여러 프로그램들.
  - 스케줄링: 우선 순위 높은 것을 CPU를 먼저 주어 먼저 사용.
  - 커널: Core가 되는 소프트웨어
- 허프만 코딩(Huffman coding)

이런 분야에서도 많은 곳에서 사용되고 있다!!

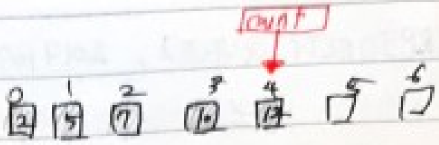
• 우선순위 큐의 ADT.

- 객체: n개의 우선 순위를 가진 요소들의 모임.
- 연산:
- create() ::= 우선 순위 큐를 생성한다.
- init(q) ::= 우선 순위 큐 q를 초기화한다.
- is_empty(q) ::= 우선순위 큐 q가 empty인지 검사한다. (element 갯수 = 0.)
- is_full(q) ::= 우선순위 큐 q가 full인지 검사한다. → array일 경우는 가능, Link list는 불가능
- insert(q,x) ::= 우선 순위 큐 q에 요소 x를 추가한다.
- delete(q) ::= 우선순위 큐 q에서 가장 우선 순위가 높은 요소를 삭제하고 이를 반환한다.
- find(q) ::= 우선 순위 큐 q에서 우선순위가 가장 높은 요소를 반환한다.

(insert, delete: 우선순위 큐에서 가장 중요한 두 개)

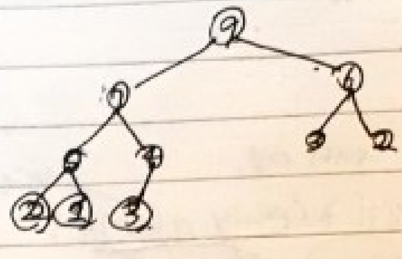
p.304~306. 9.2 우선순위 큐의 구현 방법.

- 배열(array)로 사용한 우선순위 큐



- 연결 리스트(linked list)를 사용한 우선순위 큐.

- 히프(heap tree)를 사용한 우선순위 큐



└ 이 히프 tree도 tree라서
⟨ array
⟨ linked list로 나타낼 수 있다.

└ 보통 array를 사용.
다만, 이 heap tree의 배열구조 표현 구현은
우선순위 큐의 배열구조랑은 당연히 다르다.

- 우선 순위 큐 구현 방법과 시간.

자료구조	우선순위 큐 구현 방법	삽입 시간	삭제 시간.
배열 (array)	순서 없는 (unordered) 배열	O(1)	O(n)
	정렬된(ordered) 배열	O(n)	O(1)
연결 리스트 (linked list)	순서 없는(unordered) 연결 리스트	O(1)	O(n)
	정렬된(ordered) 연결리스트	O(n)	O(1)
	히프(heap tree)	O(logn)	O(logn)

└ heap이라 말하지만, 정확히는 heap tree이다.

시간적 측면에서 가장 장점이 크다.

heap (tree)을 써야 한다!!

→ n보다 logn은 월등히 낫다.

p.326~329 **9.3. 히프(heap)**

• 히프는 사전에서 "더미"라는 뜻이다. → 더미 트리

□ 아래 조건을 만족하는 완전(complete) 이진 트리

(일단 complete여야 한다)

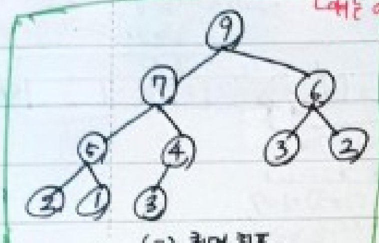
(레벨 별로 node를 꽉 꽉 채우고, 가장 마지막 level의 node는 자유롭게 채운다)

□ 최대 히프(max heap): key(부모 노드) ≥ key(자식 노드)

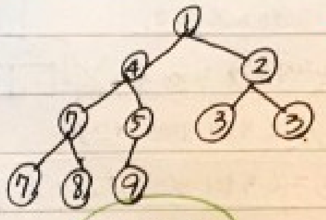
(부모가 자식보다 key값이 크면 최대히프이다. → 모든 node에 다 적용)

□ 최소 히프(min heap): key(부모 노드) ≤ key(자식 노드)

└ 얘는 여기서 사용 X



(a) 최대 히프



(b) 최소 히프

(우리는 최대 히프만 사용한다!!)

얘는 사용 X

Hours 함수 ⇒ 가장 가까운 높은 정수, 내림

$\lceil 2.1 \rceil = 3$ ⟺ $\lfloor 2.1 \rfloor = 2$

(ceil의 설정 ⇒ 가장 가까운 높은 정수 천장)

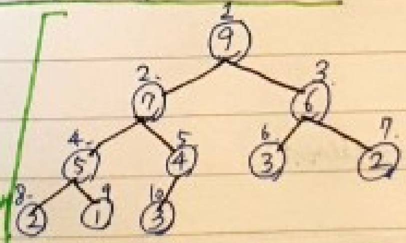
• 히프의 특징

□ n개의 노드를 가지는 히프의 높이는 $\lceil \log_2 (n+1) \rceil$

(컴퓨터에서는 무조건 $\log_2$라고 보아야 한다!!)

□ 레벨 i에는 2^{i-1}개의 노드가 존재(가장 큰 레벨은 제외)

- level 1: $2^{1-1} = 1$
- level 2: $2^{2-1} = 2$.
- level 3: $2^{3-1} = 4$. 노드존재
- level 4:



height = 4 ⇒ $\lceil \log_2 11 \rceil$

p.326~339 9.4 히프의 구현 방법.

- 배열로 사용한 구현
- 히프 구조체 정의
- 히프에서 삽입
- 히프에서 삭제
- 히프의 복잡도 분석

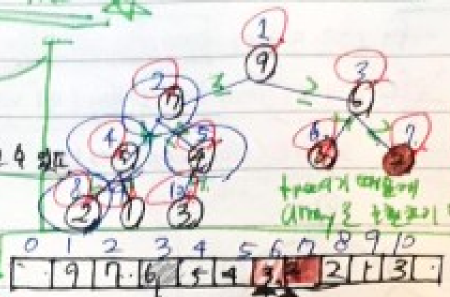
• 배열을 사용한 히프의 구현 (p.326~327) → complete binary tree ☆

<트리는 모르겠 암기할부터>.

□ 히프는 배열을 사용하여 구현

□ 완전 이진 트리이므로 각 노드에 번호를 붙일 수 있다

□ 이 번호를 배열의 인덱스라고 생각..



tree이기 때문에 array로 표현하기 때문에 좋음.

array에서 순차적으로 저장 ☆ => 배열의 Index

□ 배열을 이용하면 부모 노드와 자식 노드를 찾기가 쉽다. ☆

□ 왼쪽 자식의 인덱스 = (부모의 인덱스)*2 ☆

왼쪽 = (2*3)=6

□ 오른쪽 자식의 인덱스 = (부모의 인덱스)*2 + 1. ☆

오른쪽 = (2*3)+1=7.

└ 오른쪽 노드는 ×2 더 +1 하면 됨

□ 부모의 인덱스 = (자식의 인덱스) / 2 ☆

└ 몫, integer, 나누기는 버려서 right, left 비교없이 2를 나누어줌 a!!!

☆ 2 ☆ └ 이진 트리라 그가 궁금히 좋아!!

• 히프 구조체 정의.

```
#define MAX_ELEMENT 200

typedef struct {
    int key;
} element;      (key → element)

typedef struct {
    element heap[MAX_ELEMENT];      — array 형태로 되어있음
    int heap_size;      → heap에 어디까지 차있는지 알도록 size를 저장.
                        └ size, 현재 heap의 갯수, 현재 element의 갯수
} HeapType;

HeapType heap;                      // 정적 메모리 할당 사용.
HeapType *heap = create();          // 동적 메모리 할당 사용.
```

→ 동적 할당으로 메모리 공간 확보할 수 있음.

create() = malloc() ☆

①삽입
↓
②upheap

• 히프에 삽입, ★ heap은 complete binary tree라서 삽입은 꼭 자리를 정해주면서 해야 하므로 맨 끝자리부터에서 받는 것이다.

□ 히프에 삽입 동작은 아래와 같음.

① □ 히프의 마지막 위치에 (키 k)를 삽입한다.

② □ 삽입한 위치에서 루트 노드까지의 경로에 있는 노드들을 교환하여 히프를 재구성한다.
(이 동작을 upheap동작이라고 함). <heap의 조건 만족하도록 노드들의 위치 교환(변경)>

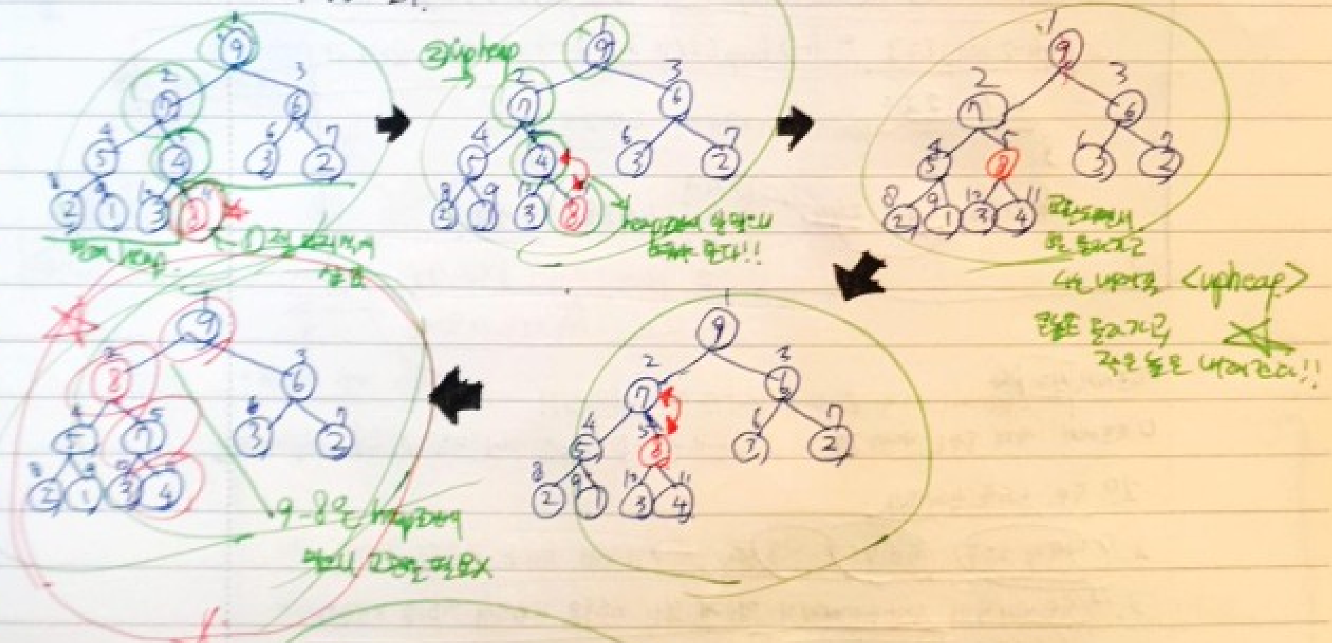
heap tree에서 젤 마지막 레벨에서 올라가면서 하나씩 바꾼다는 말.

참고: 히프의 높이가 O(logn)이므로 upheap동작은 O(logn)임.

search와 비교해보면.

★ 한 level 올라가면서 "루트노드까지 가는 길"을 따라 가면서 heap 조건을 따져 서로 교환해나간다!!

• 히프에 삽입 - 보기.



□ 교환하는 것이 아니고, 교환할 필요가 없는 자리에서 멈추면 된다!!

• 히프에 삽입 – 프로그램

```
// 현재 요소의 개수가 heap_size인 히프 h에 item을 삽입한다.
// 삽입 함수.
void insert_max_heap(HeapType *h, element item)
{
    int i;
    // 새로운 노드를 집어넣을 위치를 만든다
    i = ++(h->heap_size);          ── ① 준비.
    // 트리를 타고 올라가면서 부모 노드와 비교하는 과정.   ② 준비 및 실행.
    while((i != 1) && (item.key > h->heap[i/2].key)) {
        h->heap[i] = h->heap[i/2];   → parent에 있는 것 i에 가져온 뒤 내려버린다!!
        i /= 2;                      → parent로 가야하니까 i/=2로 한다!!
    }
    h->heap[i] = item;    // 새로운 노드를 삽입
}
```

① 시작 + 내린 다음 그 parent 위치에 item 삽입 (i를 위로 바꿨으니.)

• 히프에서 삭제 (tree에서 가장 큰 요소(root)를 삭제해야 한다!!)

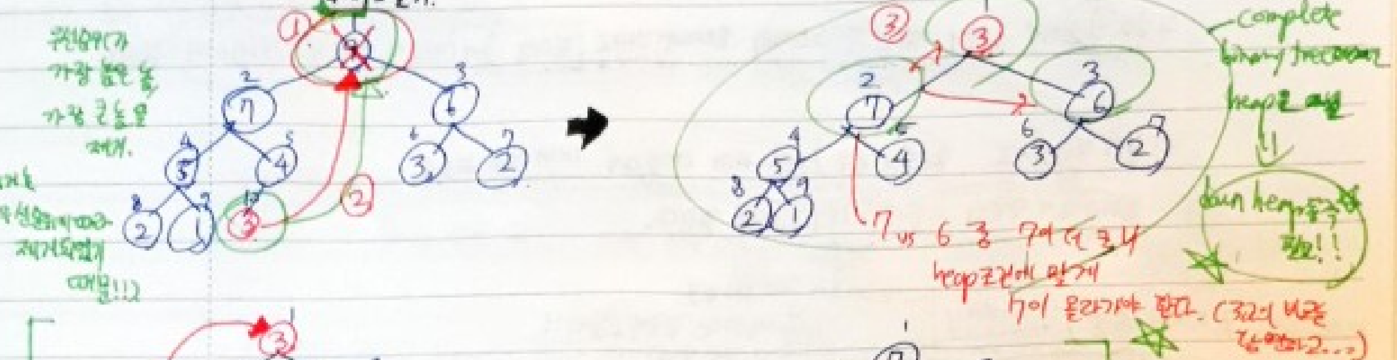
□ 히프에서 삭제 동작은 다음과 같음. ↳ 그 다음 heap을 다시 만든다. → 이게 어려움.

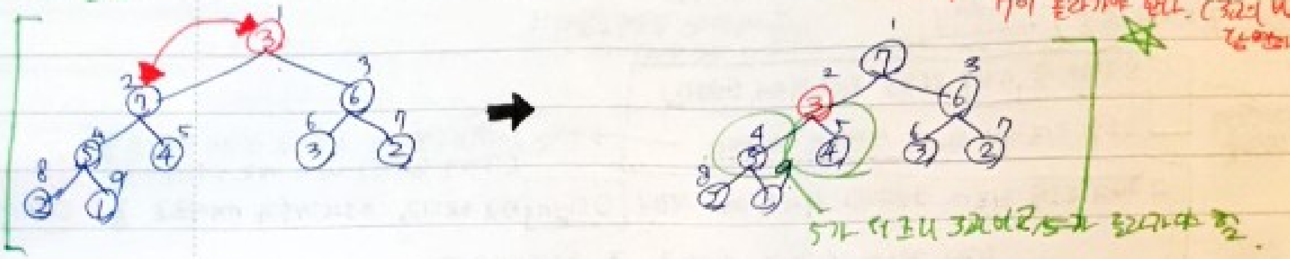
1. 루트 노드를 삭제한다.
2. 마지막 노드를 루트 노드로 이동한다. → 이러면 tree는 구성됨.
3. 루트에서부터 단말 노드까지의 경로에 있는 노드들을 교환하여 히프를 구성한다.
   (이 동작을 downheap 동작이라고 함). → <heap 조건에 맞게 차곡 내려오게 함>
   (내려가면서 downheap!!) <heap 조건 안맞으면 또 교환!!>

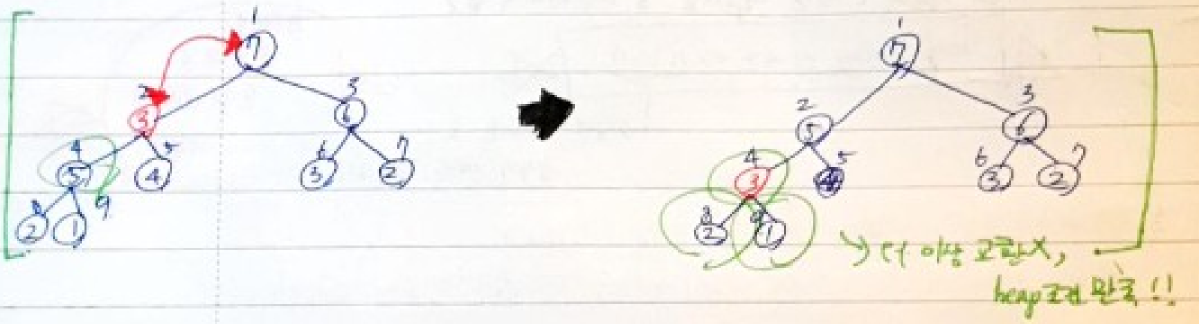
참고: 히프의 높이가 O(lgn)이므로 downheap 동작은 O(lgn)임.

삽입할 때도 $O(\log_2 n)$, 삭제할 때도 $O(\log_2 n)$

• 히프에서 삭제 _ 보기.







• 히프에서 삭제 _ 프로그램.

```
element delete_max_heap(HeapType *h)
{
    int parent, child;        → downheap을 하면서 윗놈은 parent, 아랫놈은 child로
    element item, temp;

    item = h->heap[1];                     // item = 루트노드
    temp = h->heap[(h->heap_size)--];      // temp = 마지막 노드
    parent = 1; child = 2;
    while(child <= h->heap_size){
        // 현재 노드의 자식 노드 중 더 큰 자식 노드를 찾는다.
        if((child < h->heap_size) && (h->heap[child].key) < h->heap[child+1].key)
            child++;      오른쪽께 크니 오른쪽자식으로 이동!!
        if(temp.key >= h->heap[child].key) break;
        // 한 단계 아래로 이동.
        h->heap[parent] = h->heap[child];      // child를 parent로 올림
        parent = child; child *= 2;
    }
    h->heap[parent] = temp;
    return item;                // 루트 노드 리턴
}
```

→ 하나를 삭제 후 크기를 줄여야 하니까 '--'가 존재

→ 현재 루트노드를 제거 후 return해야 하기 때문에 item으로 저장해 놓는다

(왼쪽께 더 크면 이 과정을 그냥 넘어가서 왼쪽 것이 교환되게 됨).

→ 맨 마지막 노드가 child보다 크면 교환하지 말고 바로 나옴

└ 밑으로 내려가니 child로, └ 2를 곱해서 다시 child의 left child로 되게 함.

log n → 굉장히 천천히 올라가는 함수. ⇒ 알고리즘에서 굉장히 좋은 그래프임. (완만하기 때문)

heap : 굉장히 빠른 구조

• 히프의 복잡도 분석.

- 삽입 연산에서 최악의 경우 루트 노드까지 올라가야 하므로 트리의 높이 만큼의 시간인 $O(\log n)$이 걸린다.

- 삭제 연산에서도 최악의 경우 가장 아래 레벨까지 내려가야 하므로 트리의 높이 만큼의 시간인 $O(\log n)$이 걸린다.

p.340~342 9.5 히프 정렬 → sorting ~ 알고리즘시 많이 쓰임 → heap 사용하면 굉장히 효율적!!

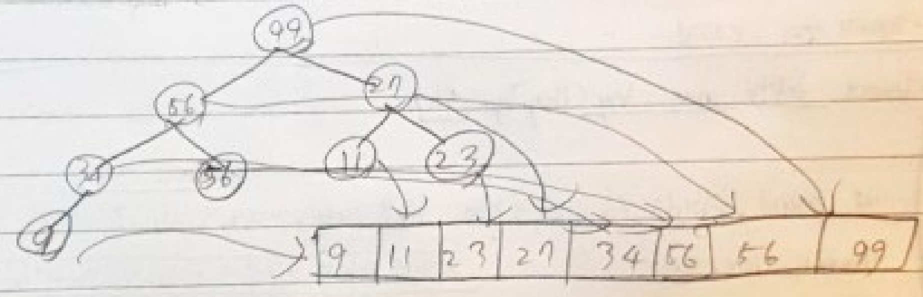
heap sorting의 알고리즘

- 정렬해야 할 n개의 요소들을 최대 히프에 삽입한다!
- 요소를 히프에서 하나씩 삭제하여 저장한다. → 삭제된 순서대로 나열하면 큰 순서로 정렬이 자동으로 됨. (삭제된 순서대로 나오기 때문 + 우선순위 큐)
- 하나의 요소를 히프에 삽입하거나 삭제할 때 시간이 $O(\log n)$ 만큼 소요되고, 요소의 개수가 n개이므로 총 $O(n\log n)$ 시간이 걸린다. (정렬 알고리즘 중 빠른 편에 속함.)

heap sorting 의 시간 ⇒ $n\log n$ → 정렬 알고리즘 중 굉장히 빠른 편이다!!

heap의 큰 순서로 트리 구성을 이용!!

즉, 정렬되어야 할 것들을 최대히프에 넣어서 히프 트리를 구성했다가, 한번에 하나씩 요소를 히프에서 꺼내서 배열의 뒤쪽부터 저장해서 정렬하면 된다!!



• 히프 정렬 – 프로그램.

```
// 우선순위 큐인 히프를 이용한 정렬 프로그램
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);
    }
}
```

heap은 완전한 binary tree이다. (큰 순서로 트리(배열) 구성해서 쓸 때도 많고, 성능도 좋다.)

p.343~347 9.6 머신스케줄링(machine schedling)

- 공장에 동일한 기계 m개와 처리해야하는 작업 n개를 가지고있다고 해보자. + 각 작업이 필요로 하는 기계의사용시간이 다르다고 가정.

- 우리의 목표는 모두 기계를 골고루하여 가장 최소의 시간 안에 작업들을 모두 끝내는 것
  - => 이걸 정확하게 결정을 짜는 것을 "머신 스케줄링(machine schedling)"이라 한다

- 이 문제는 알고리즘 분야에서 상당히 유서 깊은 문제고 많은 응용분야
  - => 서버가 여러 개 있어서 서버에 작업을 분배할 때도 사용.
    - → 최적의 해 찾기 어려움
    - → 하지만 근사의 해를 찾는 방법은 있음
      - → 이중 한가지가 LPT(longest pro-cessing time first)

- LPT : 가장 긴 작업을 우선적으로 기계에 할당하는 것.
  (자세한 설명과 구현은 교재 p. 343~346 참고)

- LPT 알고리즘 : 각 작업들을 가장 먼저 사용가능하게 되는 기계에 할당하는것
  - → 최대 히프가 아닌 최소 히프 사용

☆ 예산수학시험에 나왔었다!! ☆

그냥 저장하면 메모리 공간의 낭비가 심함.

p.347~355 9.7 허프만 코드(Huffman code)

□ 허프만 코드는 각 글자의 빈도수가 알려져 있는 문서를 minimum redundancy(가장 작은 중복)로 표현하는 코드임 (1952년 MIT 학생)

(문서(파일, 텍스트) → 글자의 빈도수가 알려진 문서를, 컴퓨터에 표현(encoding)하기 위해 최소한으로 표현하는 코드(중복 비트 수를 최소화하며 효율적으로 표현하는 것).)

중복을 줄여야 함

□ 예를 들어 어떤 문서가 e, t, n, i, s의 5개의 글자로만 이루어져 있다고 가정하고, 각 글자의 빈도수가 아래와 같을 때, 각 글자에 대한 비트 코드(허프만 코드)는 아래와 같음.

글자	빈도수	비트 코드		비트 수
e	15	00	×2 =	30
t	12	01	×2 =	24
n	8	10	×2 =	16
i	6	110	×3 =	18
s	4	111	×3 =	12
합계	45			88

주어진 것들 → Huffman code.

=> 11 byte만 있으면 표현 가능! ☆

• 허프만 코드 - 알고리즘

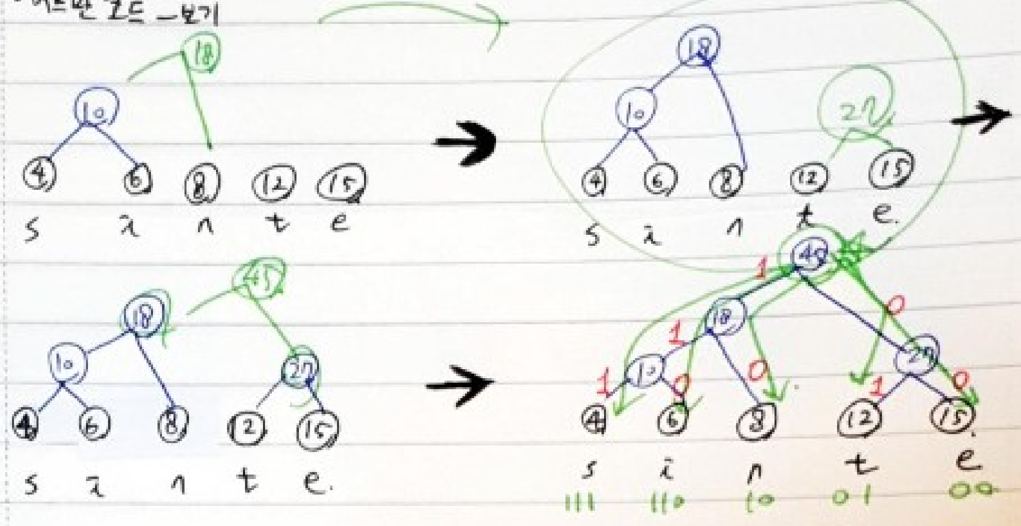
☆ 이미 다 표현했을 거니까 기억을 잘 되살려봐!! ☆

1. 모든 문자를 출현 빈도수에 따라 정렬한다. (여기서 각 문자는 앞으로 구성될 허프만 트리의 단말 노드임)
2. 가장 빈도수가 낮은 2개의 문자를 탐색한다.
3. 이들 2개 문자를 자식 노드로 가지는 루트 노드를 생성하고 이 루트 노드에 2개 자식 노드의 빈도수를 더한다.
4. 앞으로 2개 자식 노드는 탐색할 문자에서 제거하고, 새로 생성된 루트 노드는 탐색할 문자에 추가한다.
5. 탐색할 2개 문자가 있으면 다시 2로 간다.
6. 최종 생성된 허프만 트리의 left 링크에 1 및 right 링크에 0을 할당하여 코드(루트노드 → 단말노드)를 생성한다.

☆ 빈도수가 낮은 순서대로 정렬을 진행한다!! ☆

루프로 2~5 여러번 돌아가는

• 허프만 코드 — 보기



(나는 큰 순서부터 정도로 배열했지만
여기 그림은 작은 순서부터
그렸는데 반대방향으로 생각하면 된다!!).
(원래 0은 왼쪽 바꿀뿐인 거를
⇒ 왼쪽 바꿈 되도록)

⇓

이 말을 이제
정확하게 표현하면
"루트에서 자신에게 오는 경로를
합치하면 된다!!"라는 뜻이다!!