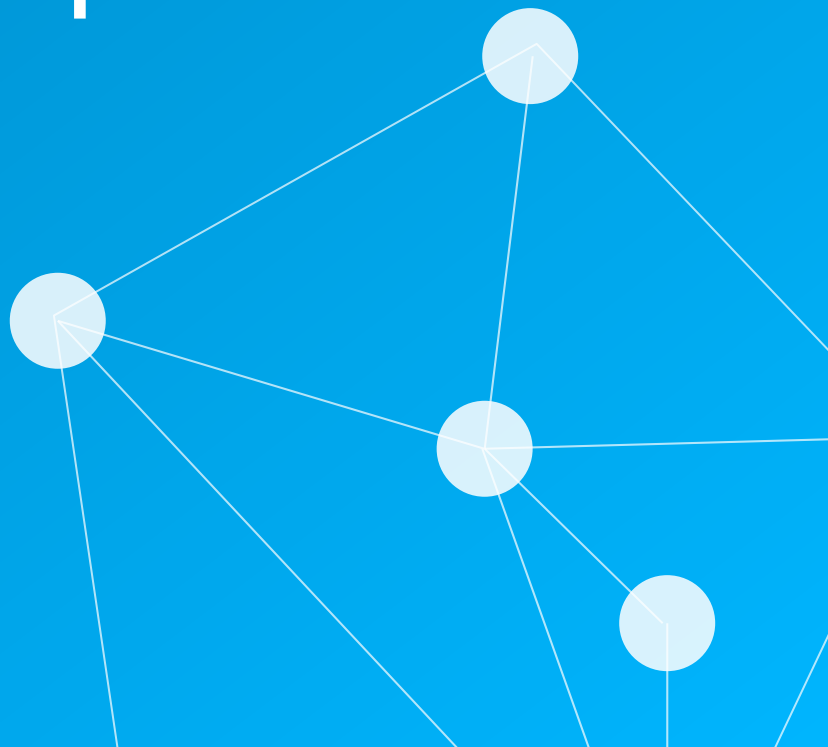
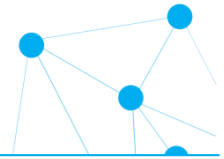
파이썬
자료구조

08 CHAPTER

트리

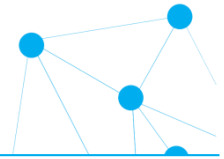


8장. 학습 목표



- 트리의 개념과 용어들을 이해한다.
- 이진트리의 표현 방법을 2가지를 이해한다.
- 이진트리의 순회방법들과 연산들을 이해한다.
- 힙의 동작 원리와 효율성을 이해한다.
- 배열 구조를 이용한 힙의 구현 방법을 이해한다.
- 이진트리와 힙을 문제 해결에 활용할 수 있다.

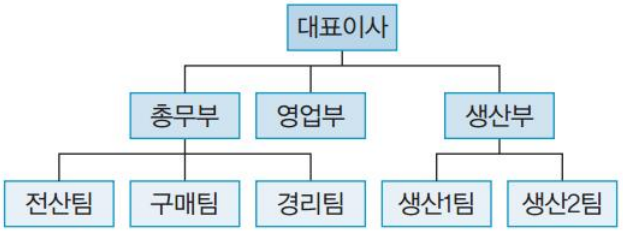
8.1 트리란?



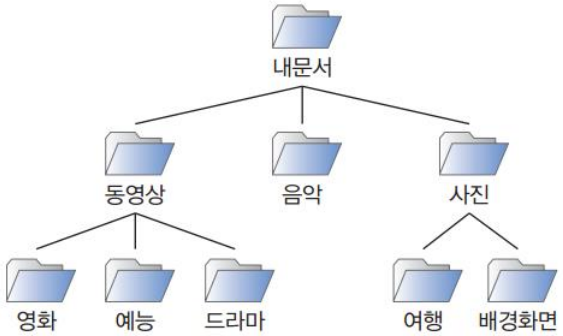
- 계층적인 자료의 표현에 적합한 자료 구조
- 일반트리의 표현 방법

트리란?

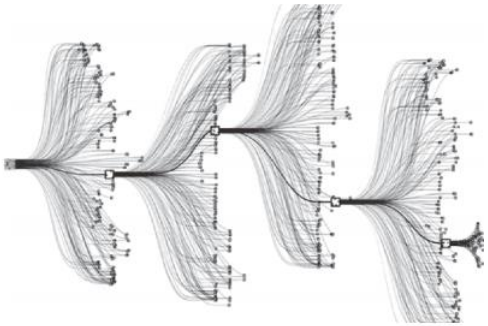
- 계층적인 자료의 표현에 적합한 자료 구조



(a) 회사의 조직도

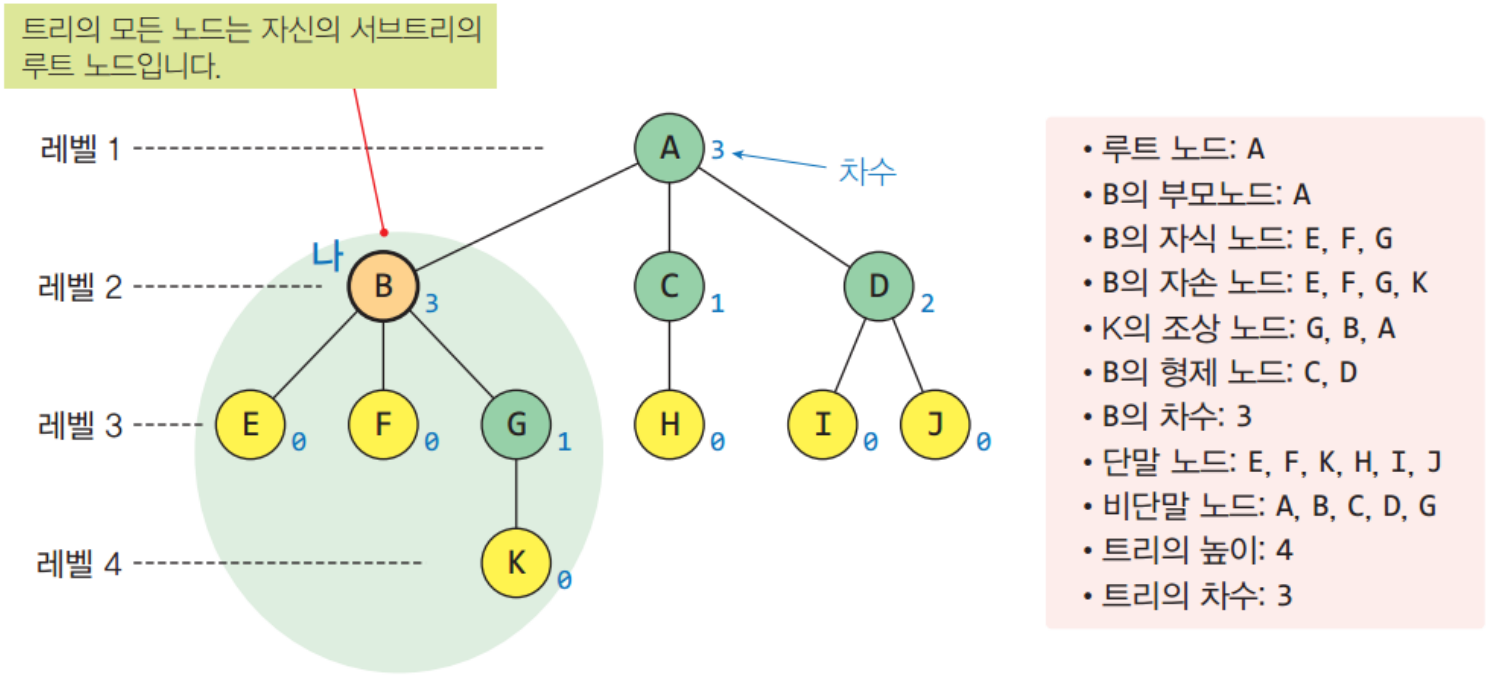


(b) 컴퓨터의 폴더 구조



(c) 인공지능 바둑 프로그램의
거대한 결정 트리(decision tree)

트리의 용어

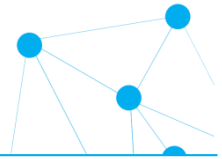


- 루트 노드: A
- B의 부모노드: A
- B의 자식 노드: E, F, G
- B의 자손 노드: E, F, G, K
- K의 조상 노드: G, B, A
- B의 형제 노드: C, D
- B의 차수: 3
- 단말 노드: E, F, K, H, I, J
- 비단말 노드: A, B, C, D, G
- 트리의 높이: 4
- 트리의 차수: 3

- 루트 노드
- 간선 또는 에지
- 부모 / 자식 / 형제
- 조상 / 자손
- 단말 / 비단말 노드

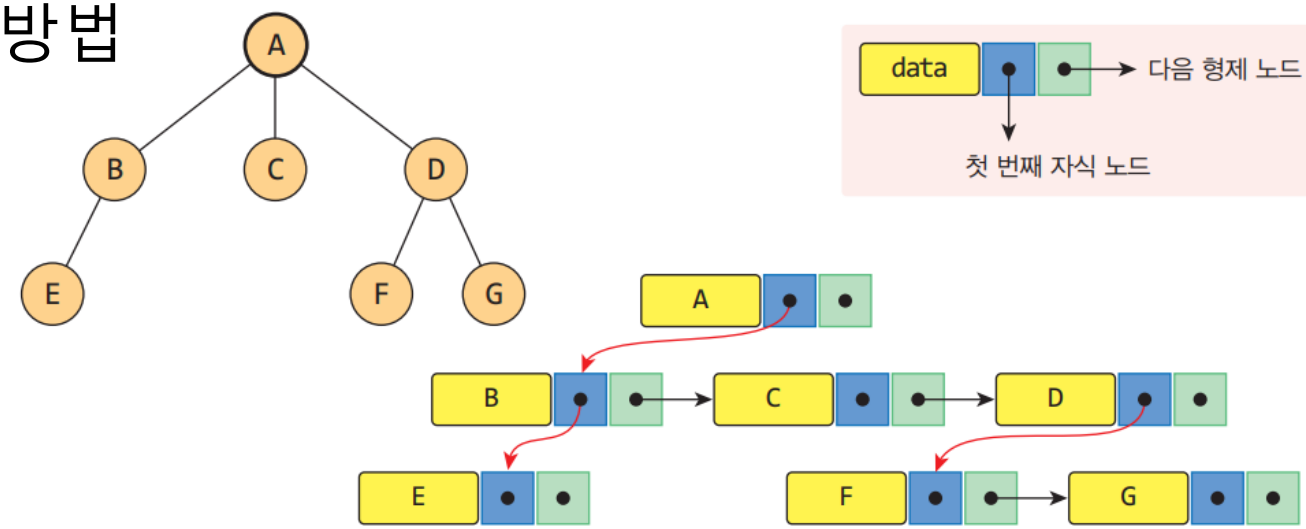
- 노드의 차수
- 트리의 차수
- 레벨
- 트리의 높이
- 포레스트(forest)

일반 트리의 표현 방법

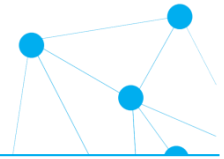


- 일반 트리의 노드

- 다른 방법

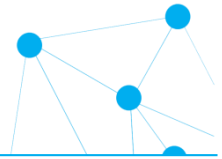


8.2 이진 트리

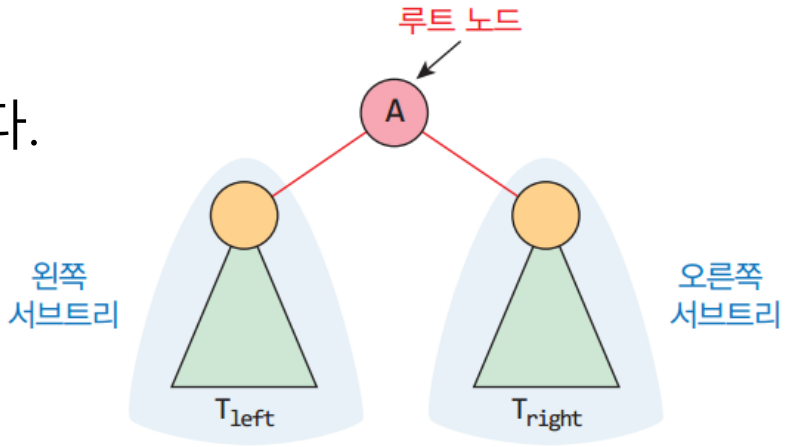


- 이진트리는 순환적으로 정의된다.
- 이진트리의 종류와 성질
- 이진트리의 표현 방법

이진 트리



- 모든 노드가 2개의 서브 트리를 갖는 트리
  - 서브트리는 공집합일수 있다.
  - 이진트리는 순환적으로 정의된다.

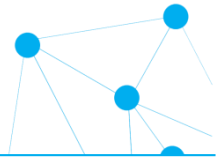


정의 8.1 이진 트리의 정의

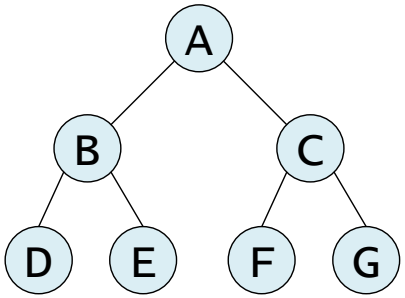
(1) 공집합이거나

(2) 루트와 왼쪽 서브 트리, 오른쪽 서브 트리로 구성된 노드들의 집합. 이진트리의 서브 트리들은 모두 이진트리이어야 함.

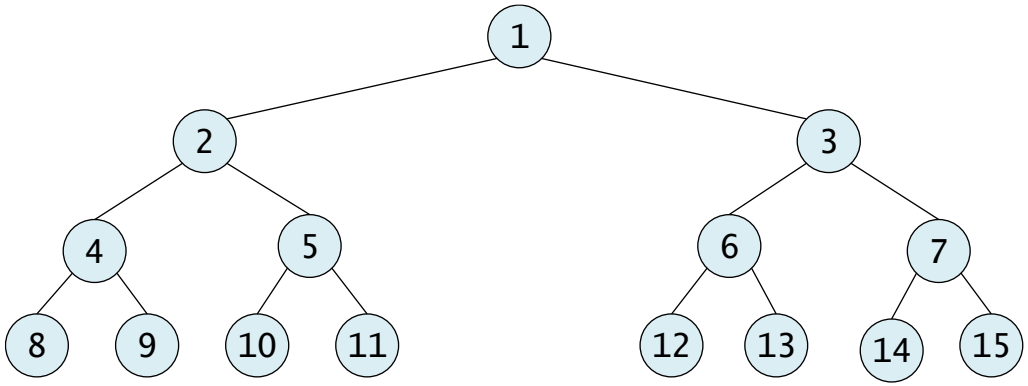
이진 트리의 분류



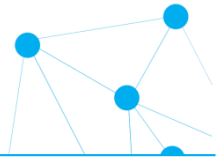
- 포화 이진 트리(full binary tree)
  - 트리의 각 레벨에 노드가 꽉 차있는 이진트리



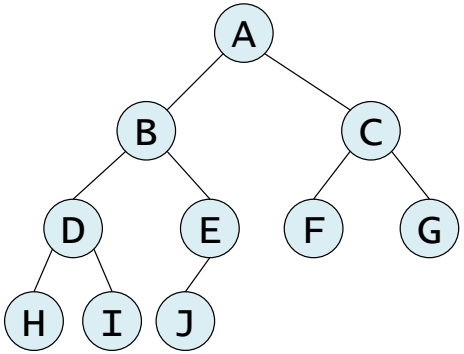
  - 노드의 번호

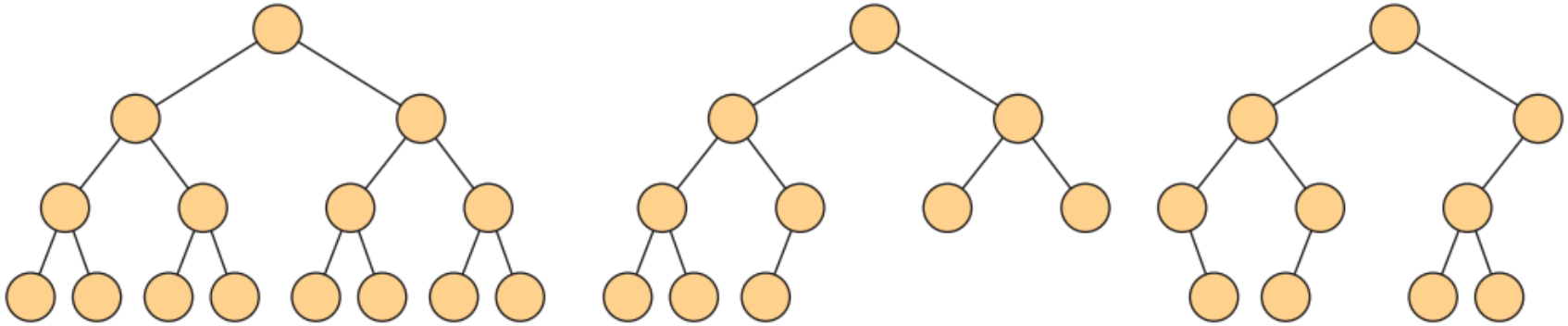


이진 트리의 분류



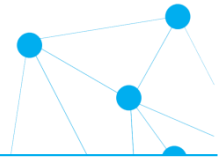
- 완전 이진 트리(complete binary tree)
  - 높이가 h일 때 레벨 1부터 h-1까지는 노드가 모두 채워짐
  - 마지막 레벨 h에서는 노드가 순서대로 채워짐



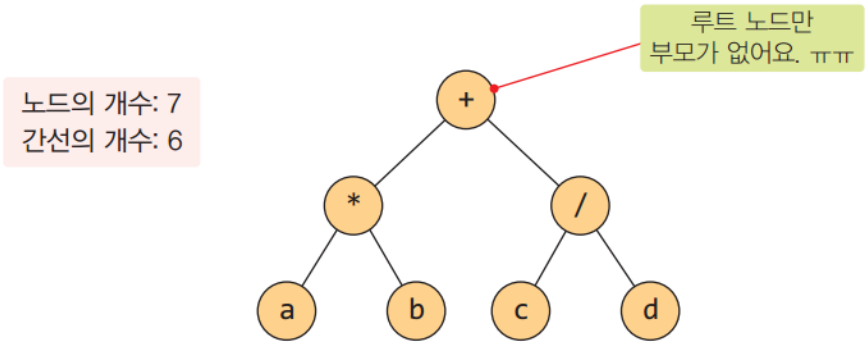


포화이진트리 완전 이진트리 기타이진트리

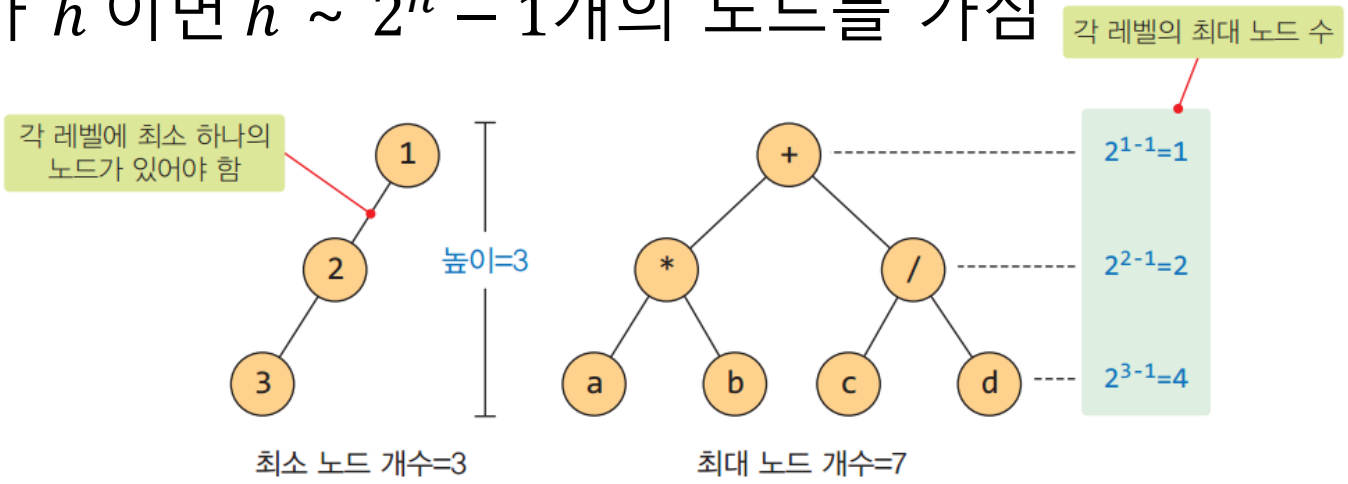
이진 트리의 성질



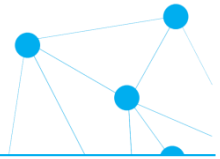
- 노드의 개수가 n개이면 간선의 개수는 n-1



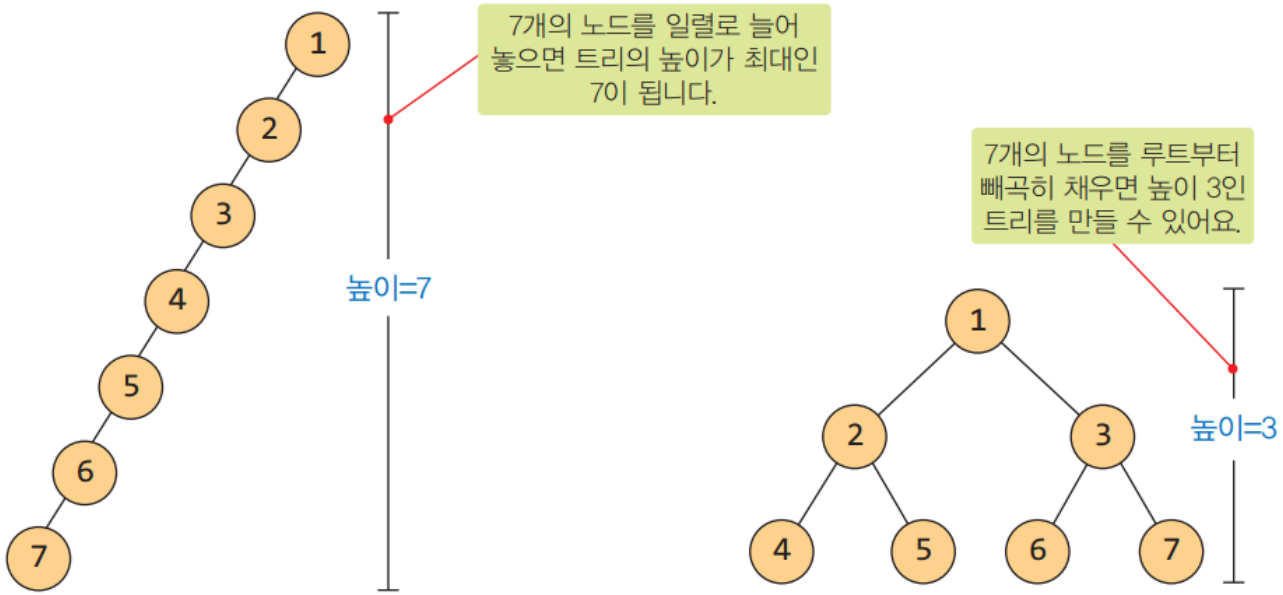
- 높이가 h 이면 $h \sim 2^h - 1$개의 노드를 가짐



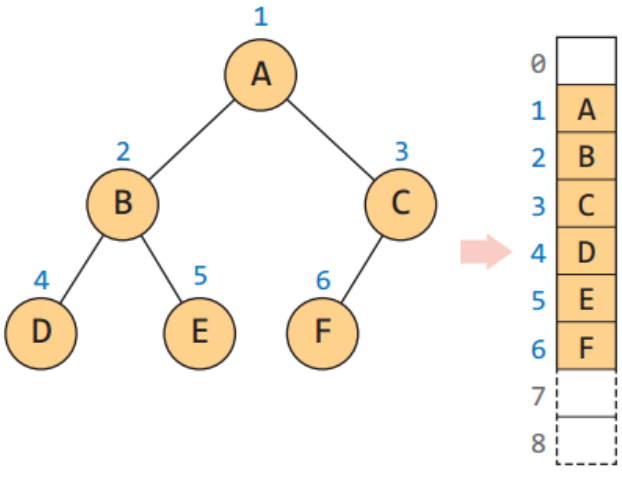
이진트리의 성질



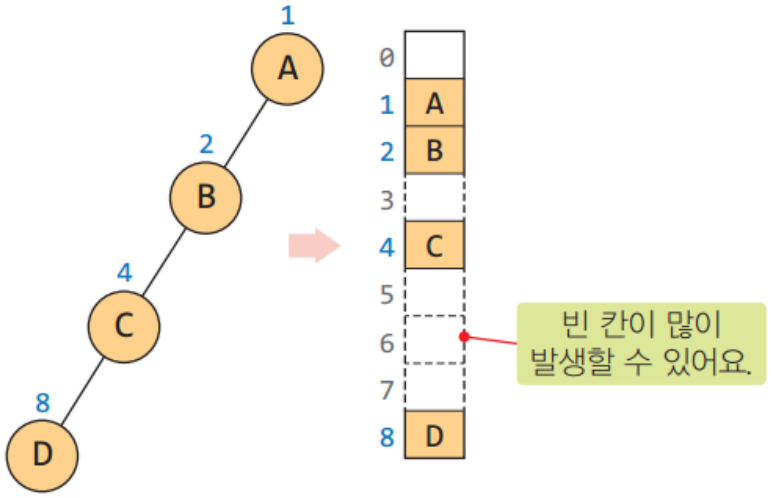
- n개 노드의 이진 트리 높이: $\lceil log_2(n+1) \rceil \sim n$



이진트리의 표현: 배열 표현법



완전이진트리의 배열 표현

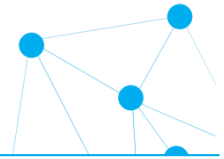


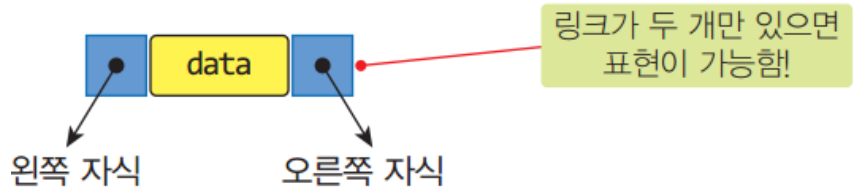
경사이진트리의 배열 표현

- 노드 i의 부모 노드 인덱스 = i/2
- 노드 i의 왼쪽 자식 노드 인덱스 = 2i
- 노드 i의 오른쪽 자식 노드 인덱스 = 2i+1

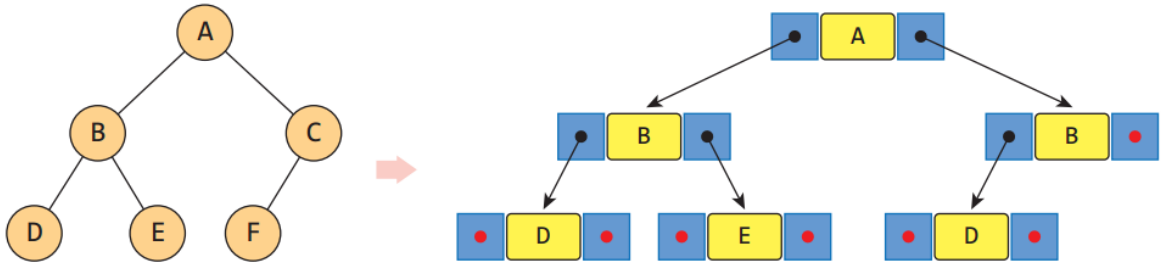
파이썬에서는 나눗셈 연산자가 /와 //로 구분되어 있습니다. 정수 나눗셈을 위해서는 i//2를 써야 합니다.

이진트리의 표현: 링크 표현법

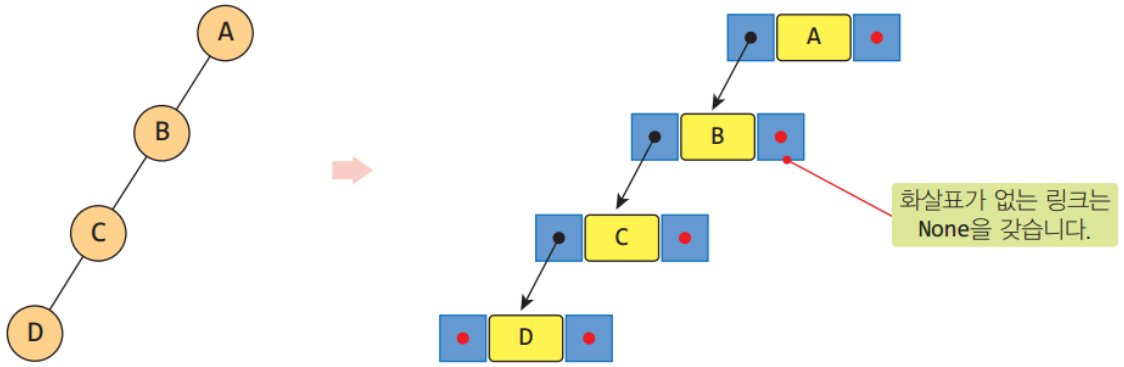





```
class TNode:
    def __init__ (self, data, left, right):
        self.data = data
        self.left = left
        self.right = right
```

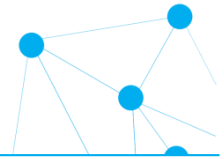


완전이진트리의 링크 표현



경사이진트리의 링크 표현

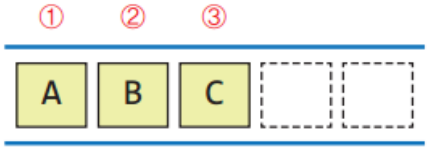
8.3 이진트리의 연산



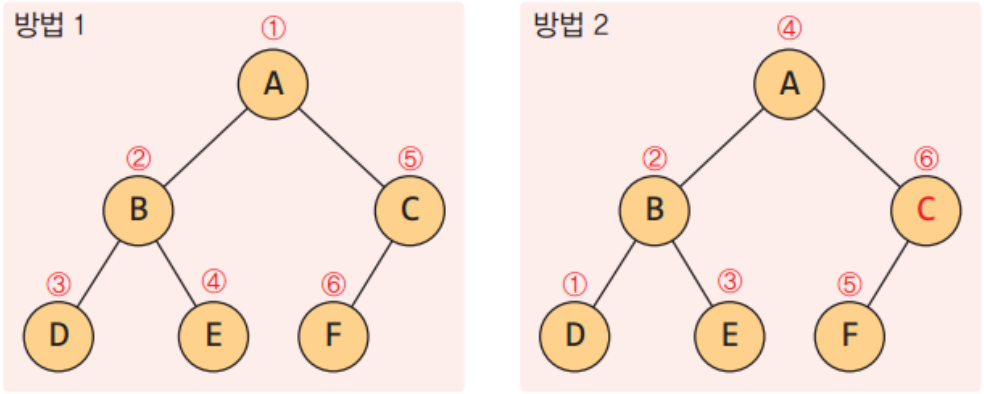
- 순회
  - 전위
  - 중위
  - 후위
  - 레벨
- 전체 노드 개수
- 단말 노드의 수
- 높이 계산

순회(traversal)

- 순회(traversal)
  - 트리에 속하는 모든 노드를 한 번씩 방문하는 것
  - 선형 자료구조는 순회가 단순
  - 트리는 다양한 방법이 있음

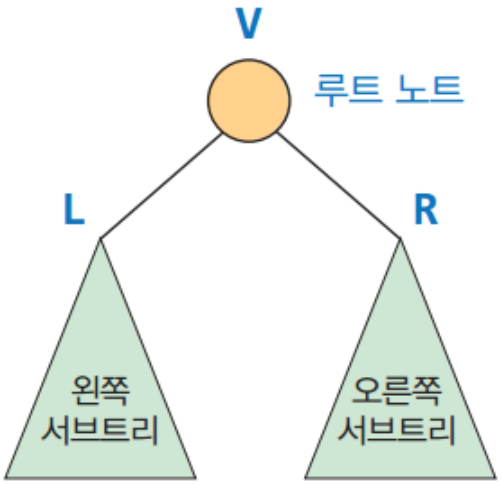


선형자료구조는
순회 방법이 단순하다.



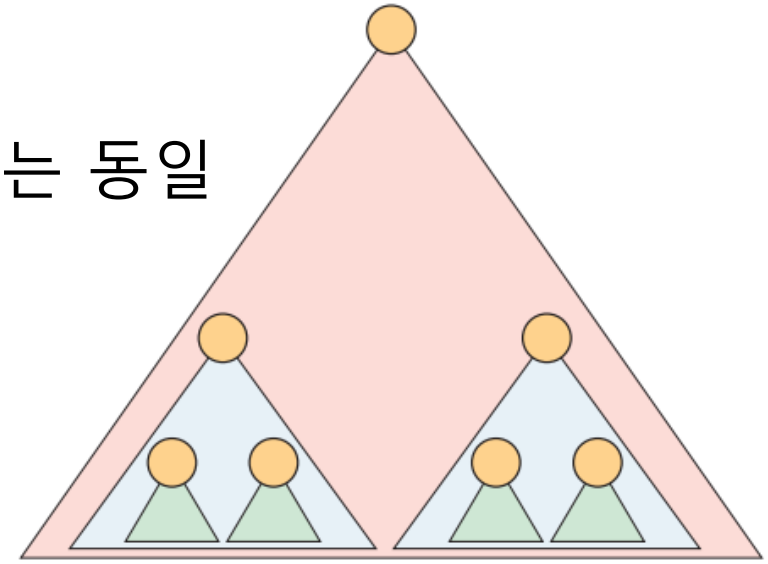
트리는 다양한 방법으로 순회할 수 있다.

이진트리의 기본 순회

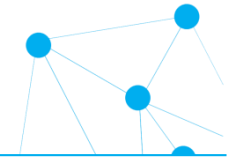


- 전위 순회(preorder traversal) : VLR
- 중위 순회(inorder traversal) : LVR
- 후위 순회(postorder traversal) : LRV

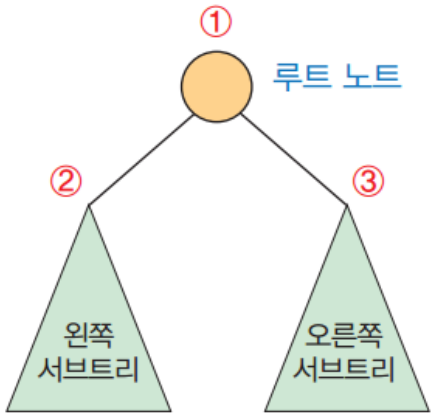
- 전체 트리나 서브 트리나 구조는 동일



전위 순회



- 루트 → 왼쪽 서브트리 → 오른쪽 서브트리

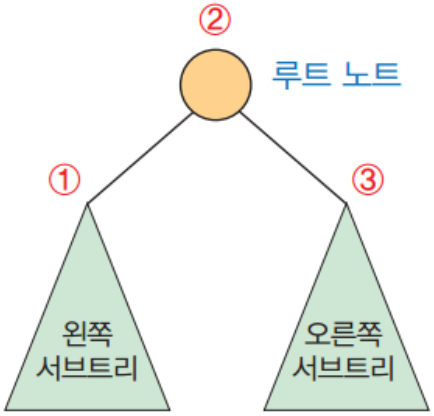



```
def preorder(n) :                    # 전위 순회 함수
    if n is not None :
        print(n.data, end=' ')       # 먼저 루트노드 처리(화면 출력)
        preorder(n.left)             # 왼쪽 서브트리 처리
        preorder(n.right)            # 오른쪽 서브트리 처리
```

- 응용 예
  - 노드의 레벨 계산
  - 구조화된 문서 출력

중위 순회

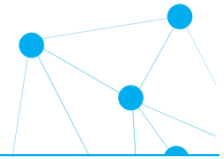
- 왼쪽 서브트리 → 루트 → 오른쪽 서브트리



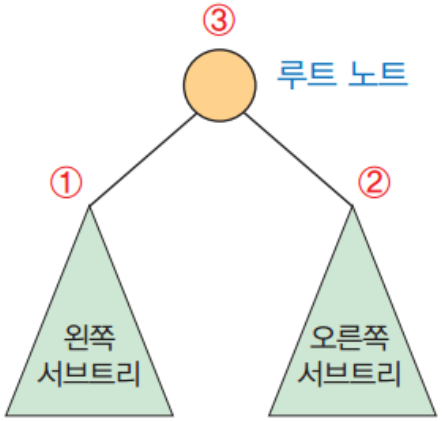

```
def inorder(n) :                    # 전위 순회 함수
    if n is not None :
        inorder(n.left)             # 왼쪽 서브트리 처리
        print(n.data, end=' ')      # 루트노드 처리(화면 출력)
        inorder(n.right)            # 오른쪽 서브트리 처리
```

- 응용 예
  - 정렬

후위 순회



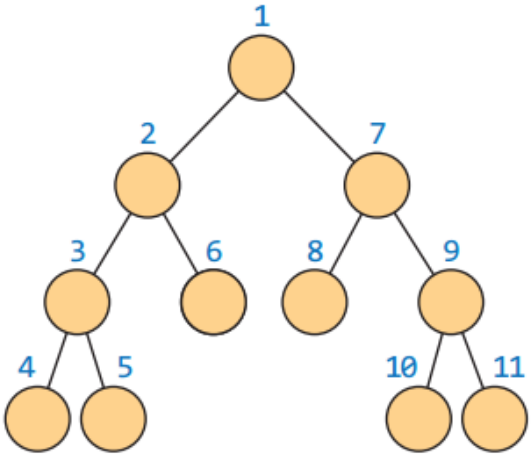
- 왼쪽 서브트리 → 오른쪽 서브트리 → 루트



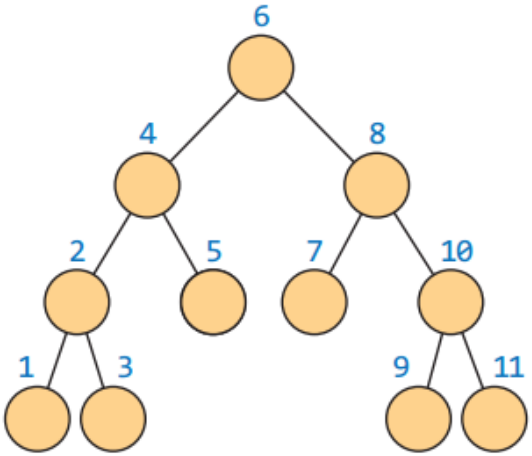

```
def postorder(n) :
    if n is not None :
        postorder(n.left)
        postorder(n.right)
        print(n.data, end=' ')
```

- 응용 예
  - 폴더 용량 계산

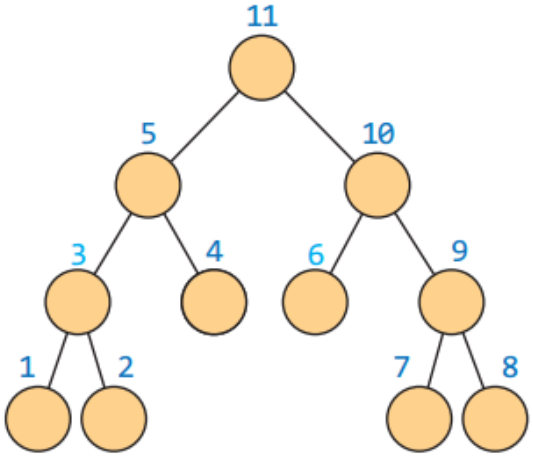
순회 방법에 따른 노드 방문 순서



(a) 전위 순회



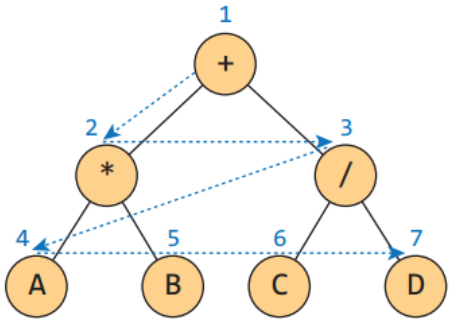
(b) 중위 순회

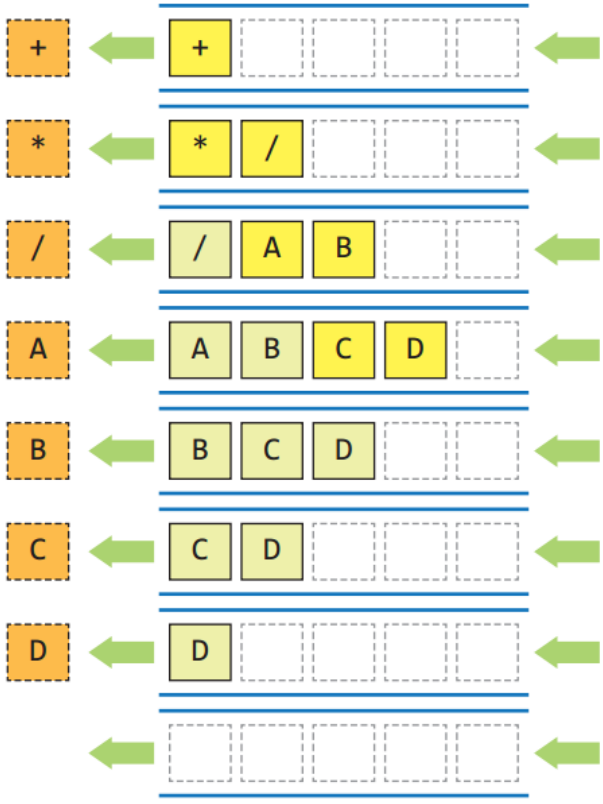


(c) 후위 순회

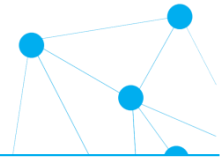
레벨 순회

- 노드를 레벨 순으로 검사하는 순회방법
  - 큐를 사용해 구현
  - 순환을 사용하지 않음





레벨 순회 알고리즘



```
def levelorder(root) :
    queue = CircularQueue()                 # 큐 객체 초기화
    queue.enqueue(root)                     # 최초에 큐에는 루트 노드만 들어있음.
    while not queue.isEmpty() :             # 큐가 공백상태가 아닌 동안,
        n = queue.dequeue()                 # 큐에서 맨 앞의 노드 n을 꺼냄
        if n is not None :
            print(n.data, end=' ')          # 먼저 노드의 정보를 출력
            queue.enqueue(n.left)           # n의 왼쪽 자식 노드를 큐에 삽입
            queue.enqueue(n.right)          # n의 오른쪽 자식 노드를 큐에 삽입
```

이진트리연산: 노드 개수, 단말 노드의 수

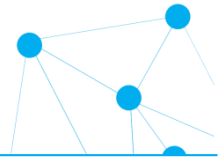
- 노드 개수

```
def count_node(n) :          # 순환을 이용해 트리의 노드 수를 계산하는 함수.
    if n is None :           # n이 None이면 공백 트리 --> 0을 반환
        return 0
    else :                   # 좌우 서브트리의 노드수의 합 + 1을 반환 (순환이용)
        return 1 + count_node(n.left) + count_node(n.right)
```

- 단말 노드의 수

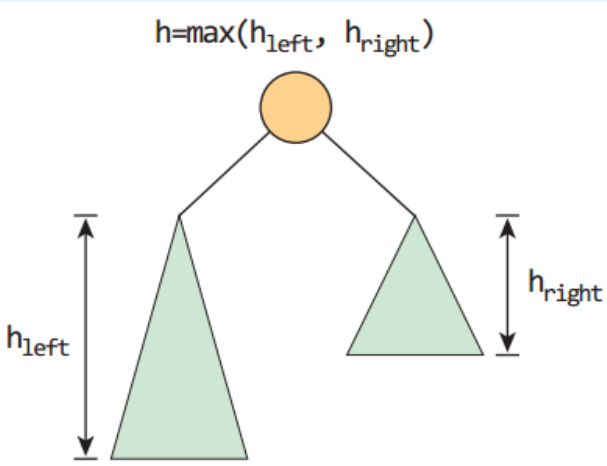
```
def count_leaf(n) :
    if n is None :                              # 공백 트리 --> 0을 반환
        return 0
    elif n.left is None and n.right is None :   # 단말노드 --> 1을 반환
        return 1
    else :                                      # 비단말 노드: 좌우 서브트리의 결과 합을 반환
        return count_leaf(n.left) + count_leaf(n.right)
```

이진트리연산 : 트리 높이



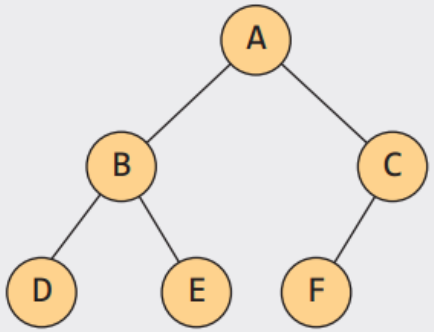
- 트리의 높이

```
def calc_height(n) :
    if n is None :                  # 공백 트리 --> 0을 반환
        return 0
    hLeft = calc_height(n.left)     # 왼쪽 트리의 높이 --> hLeft
    hRight = calc_height(n.right)   # 오른쪽 트리의 높이 --> hRight
    if (hLeft > hRight) :           # 더 높은 높이에 1을 더해 반환.
        return hLeft + 1
    else:
        return hRight + 1
```



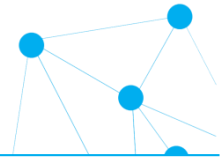
테스트 프로그램

```
d = TNode('D', None, None)
e = TNode('E', None, None)
b = TNode('B', d, e)
f = TNode('F', None, None)
c = TNode('C', f, None)
root = TNode('A', b, c)

print('\n  In-Order : ', end='')
inorder(root)
print('\n  Pre-Order : ', end='')
preorder(root)
print('\n Post-Order : ', end='')
postorder(root)
print('\nLevel-Order : ', end='')
levelorder(root)
print()

print(" 노드의 개수 = %d개" % count_node(root))
print(" 단말의 개수 = %d개" % count_leaf(root))
print(" 트리의 높이 = %d" % calc_height(root))
```



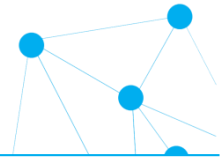

```
C:₩WINDOWS₩system32₩cmd.exe

  In-Order : D B E A F C
 Pre-Order : A B D E C F
Post-Order : D E B F C A
Level-Order : A B C D E F
 노드의 개수 = 6개
 단말의 개수 = 3개
 트리의 높이 = 3
```

8.4 응용: 모르스 코드 결정트리



- 모르스 코드
- 인코딩
- 디코딩 방법 개선 → 결정트리
- 결정 트리 알고리즘

모르스 코드 결정트리



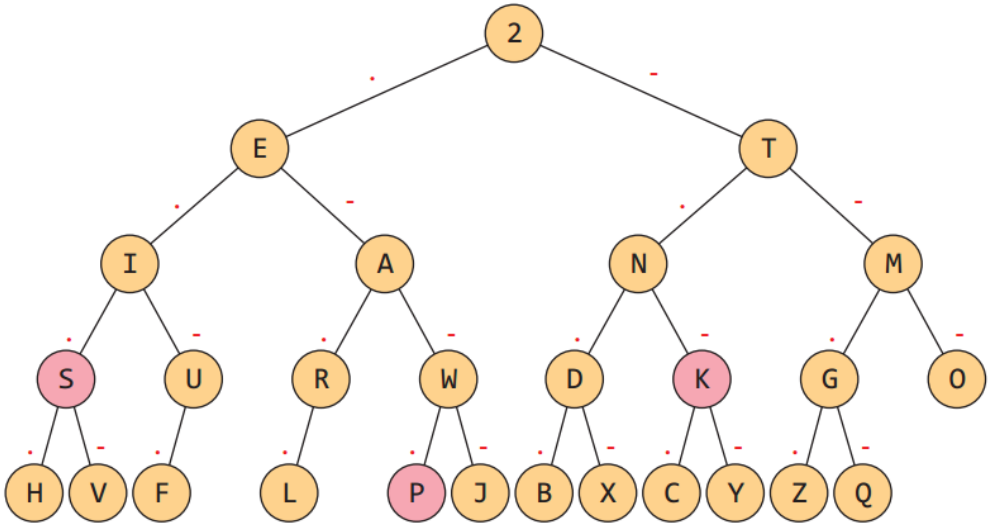
- 모르스 부호
  - 도트(점)와 대시(선)의 조합으로 구성된 메시지 전달용 부호
  - SOS: … --- …
- 모르스 부호 표

A .-	B -...	C -.-.	D -..	E .	F ..-.
G --.	H	I ..	J .---	K -.-	L .-..
M --	N -.	O ---	P .--.	Q --.-	R .-.
S ...	T -	U ..-	V ...-	W .--	X -..-
Y -.--	Z --..				

- 인코딩: 알파벳에서 모르스 코드로 변환
  - 표에서 바로 찾음: $O(1)$
    - 예) PYTHON: .--. -.-- - .... --- -.
- 디코딩: 표에서 순차 탐색
  - 표에서 순차 탐색: $O(n)$
    - 예) .--. -.-- - .... --- -. ???

디코딩 방법 개선→결정트리

- 결정트리(decision tree)
  - 여러 단계의 복잡한 조건을 갖는 문제에 대해 조건과 그에 따른 해결방법을 트리 형태로 나타낸 것
- 모르스 코드를 위한 결정 트리

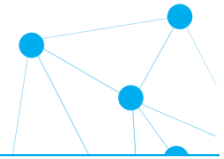


예:

...	: S
-.-	: K
.--.	: P
--...	: 코드 없음

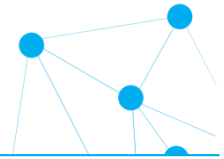
  - 디코딩 시간 복잡도:$O(log_2 n)$

모르스 코드의 결정 트리 알고리즘



```
def make_morse_tree():
    root = TNode( None, None, None )
    for tp in table :
        code = tp[1]                          # 모르스 코드
        node = root
        for c in code :                       # 맨 마지막 문자 이전까지 --> 이동
            if c == '.' :                     # 왼쪽으로 이동
                if node.left == None :        # 비었으면 빈 노드 만들기
                    node.left = TNode (None, None, None)
                node = node.left              # 그쪽으로 이동
            elif c == '-' :                   # 오른쪽으로 이동
                if node.right == None :       # 비었으면 빈 노드 만들기
                    node.right = TNode (None, None, None)
                node = node.right             # 그쪽으로 이동

        node.data = tp[0]                     # 코드의 알파벳
    return root
```

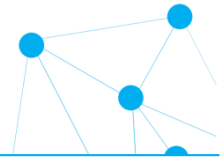
테스트 프로그램



```
morseCodeTree = make_morse_tree()
str = input("입력 문장 : ")
mlist = []
for ch in str:
    code = encode(ch)
    mlist.append(code)
print("Morse Code: ", mlist)
print("Decoding  : ", end='')
for code in mlist:
    ch = decode(morseCodeTree, code)
    print(ch, end='')
print()
```

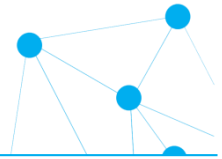
```
C:₩WINDOWS₩system32₩cmd.exe
입력 문장 : GAMEOVER
Morse Code:  ['--.', '.-', '--', '.', '---', '...-', '.', '.-.']
Decoding  : GAMEOVER
```

8.5 힙 트리



- 힙(Heap)이란?
- 힙을 저장하는 효과적인 자료구조는 배열이다.
- 우선순위 큐의 가장 좋은 구현 방법은 힙이다.
  - 힙의 복잡도 분석

힙(Heap)이란?

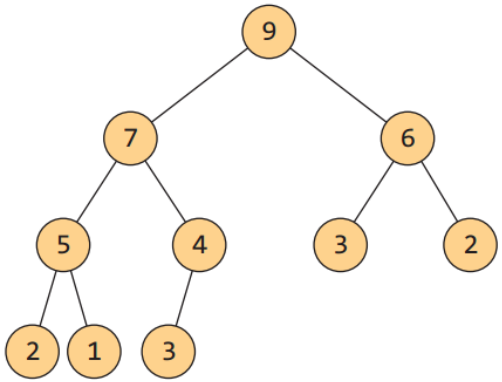


- 힙(Heap)이란?
  - "더미"와 모 습이 비슷한 완전이진트리 기반의 자료 구조
  - 가장 큰(또는 작은) 값을 빠르게 찾아내도록 만들어진 자료 구조
  - 최대 힙, 최소 힙

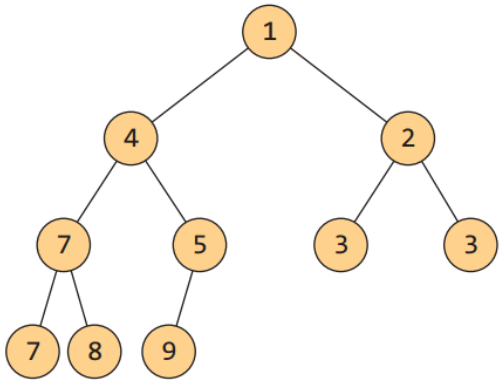
정의 8.2 최대 힙, 최소 힙의 정의

- 최대 힙(max heap): 부모 노드의 키 값이 자식 노드의 키 값보다 크거나 같은 완전이진트리 (key(부모노드) $\geq$ key(자식노드))
- 최소 힙(min heap): 부모 노드의 키 값이 자식 노드의 키 값보다 작거나 같은 완전이진트리 (key(부모노드) $\leq$ key(자식노드))

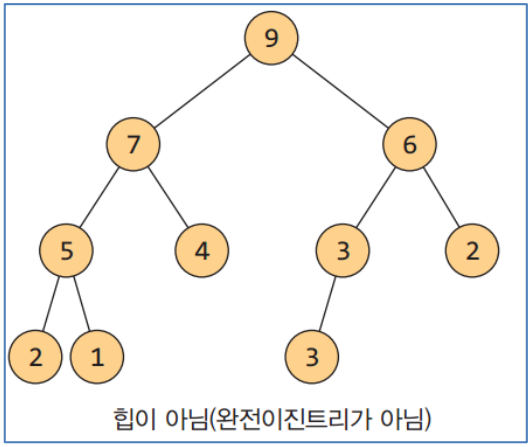
힙의 예



최대 힙(Max Heap)



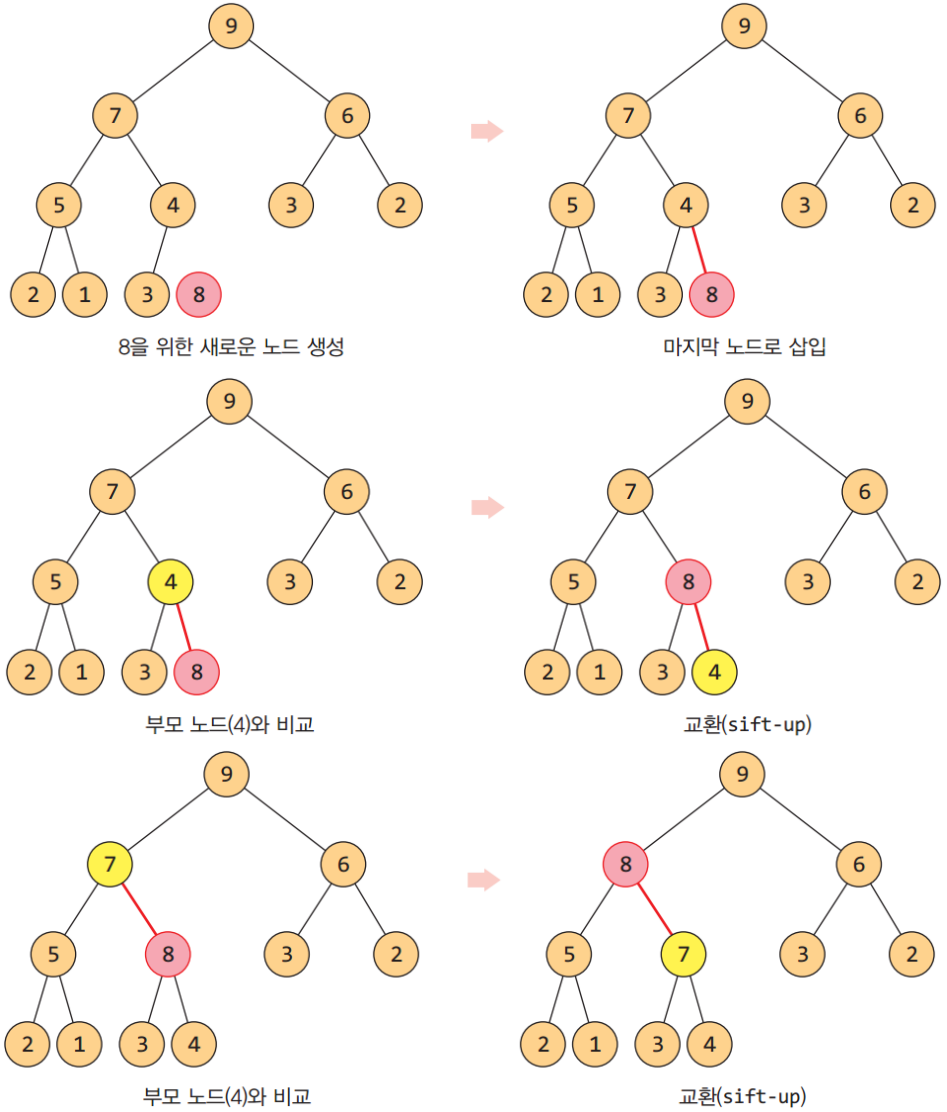
최소 힙(Min Heap)



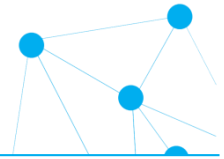
힙이 아님(완전이진트리가 아님)

힙의 연산: 삽입 연산

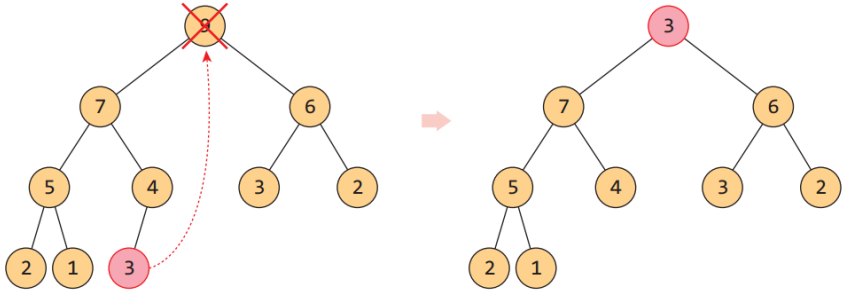
- Upheap
  - 회사에서 신입 사원이 들어오면 일단 말단 위치에 앉힘
  - 신입 사원의 능력을 봐서 위로 승진시킴
  - 시간 복잡도: $O(logn)$



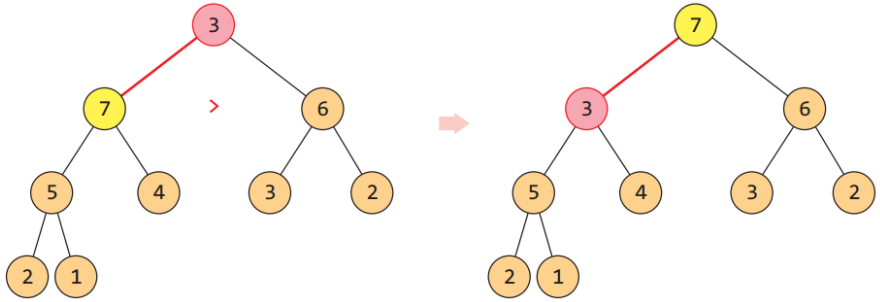
힙의 연산: 삭제 연산



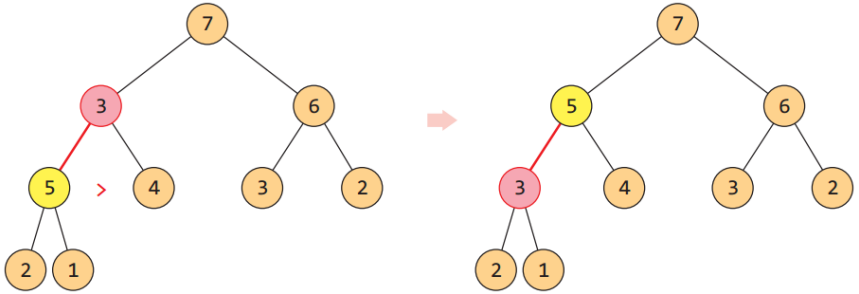
- Downheap
  - 회사에서 사장의 자리가 비면 말단 사원을 사장자리로
  - 순차적으로 강등
  - 시간 복잡도: $O(logn)$



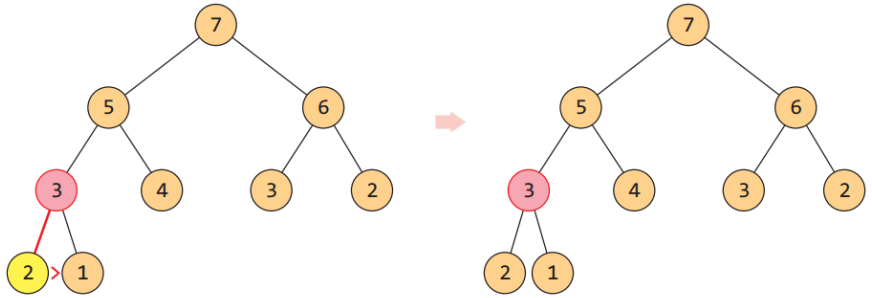
step1



step2

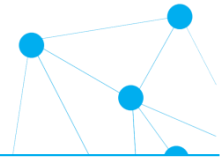


step3

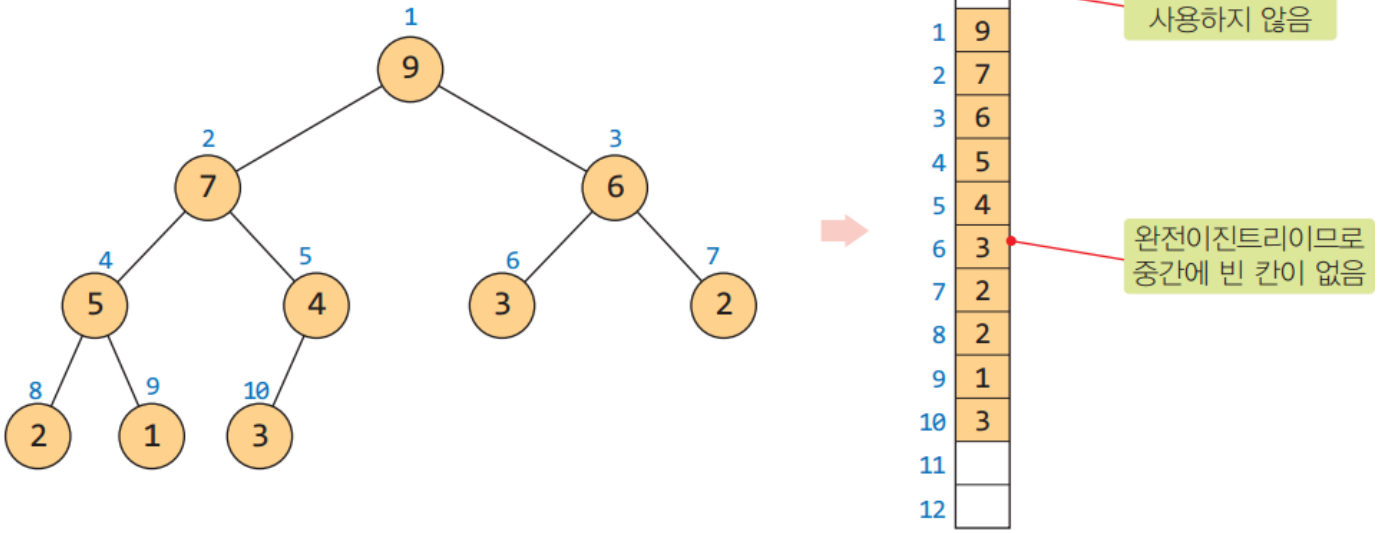


step4

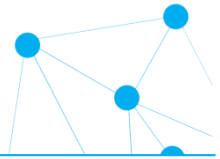
힙의 구현: 배열 구조



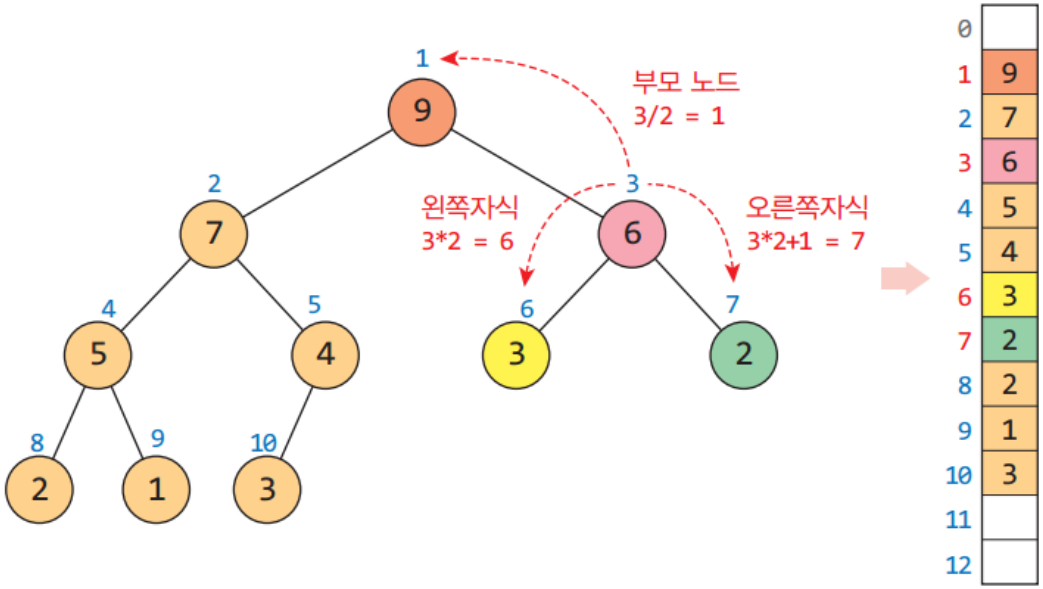
- 힙은 보통 배열을 이용하여 구현
  - 완전이진트리 ➔ 각 노드에 번호를 붙임 ➔ 배열의 인덱스



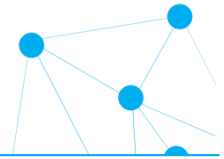
힙의 구현



- 부모노드와 자식노드의 관계
  - 왼쪽 자식의 인덱스 = (부모의 인덱스)*2
  - 오른쪽 자식의 인덱스 = (부모의 인덱스)*2 + 1
  - 부모의 인덱스 = (자식의 인덱스)/2



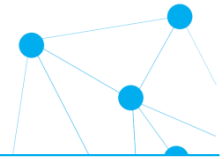
최대 힙의 구현



- 최대 힙 클래스

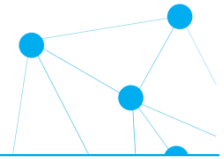
```
class MaxHeap :                                  # 최대힙 클래스
    def __init__ (self) :                        # 생성자
        self.heap = []                           # 리스트(배열)를 이용한 힙
        self.heap.append(0)                      # 0번 항목은 사용하지 않음

    def size(self) : return len(self.heap) - 1   # 힙의 크기
    def isEmpty(self) : return self.size() == 0  # 공백 검사
    def Parent(self, i) : return self.heap[i//2] # 부모노드 반환
    def Left(self, i) : return self.heap[i*2]    # 왼쪽 자식 반환
    def Right(self, i) : return self.heap[i*2+1] # 오른쪽 자식 반환
    def display(self, msg = '힙 트리: ') :
        print(msg, self.heap[1:])                # 파이썬 리스트의 슬라이싱 이용
```

최대 힙: 삽입 연산



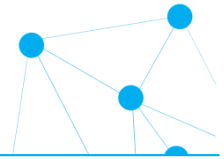
```
def insert(self, n) :
    self.heap.append(n)                    # 맨 마지막 노드로 일단 삽입
    i = self.size()                        # 노드 n의 위치
    while (i != 1 and n > self.Parent(i)):  # 부모보다 큰 동안 계속 업힙
        self.heap[i] = self.Parent(i)      # 부모를 끌어내림
        i = i // 2                         # i를 부모의 인덱스로 올림
    self.heap[i] = n                       # 마지막 위치에 n 삽입
```

최대 힙: 삭제 연산



```
def delete(self) :
    parent = 1
    child = 2
    if not self.isEmpty() :
        hroot = self.heap[1]                    # 삭제할 루트를 복사해 둠
        last = self.heap[self.size()]           # 마지막 노드
        while (child <= self.size()):           # 마지막 노드 이전까지
            # 만약 오른쪽 노드가 더 크면 child를 1 증가 (기본은 왼쪽 노드)
            if child<self.size() and self.Left(parent)<self.Right(parent):
                child += 1
            if last >= self.heap[child] :       # 더 큰 자식이 더 작으면
                break;                          # 삽입 위치를 찾음. down-heap 종료
            self.heap[parent] = self.heap[child] # 아니면 down-heap 계속
            parent = child
            child *= 2;

        self.heap[parent] = last                # 맨 마지막 노드를 parent위치에 복사
        self.heap.pop(-1)                       # 맨 마지막 노드 삭제
        return hroot                            # 저장해두었던 루트를 반환
```

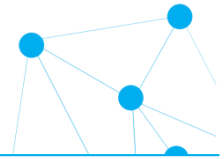
테스트 프로그램



```
heap = MaxHeap()                        # MaxHeap 객체 생성
data = [2, 5, 4, 8, 9, 3, 7, 3]         # 힙에 삽입할 데이터
print("[삽입 연산] : " + str(data))
for elem in data :                      # 모든 데이터를
    heap.insert(elem)                   # 힙에 삽입
heap.display('[ 삽입 후 ]: ')            # 현재 힙 트리를 출력
heap.delete()                           # 한 번의 삭제연산
heap.display('[ 삭제 후 ]: ')            # 현재 힙 트리를 출력
heap.delete()                           # 또 한 번의 삭제연산
heap.display('[ 삭제 후 ]: ')            # 현재 힙 트리를 출력
```

```
선택 C:₩WINDOWS₩system32₩cmd.exe
[삽입 연산] : [2, 5, 4, 8, 9, 3, 7, 3]
[ 삽입 후 ]:  [9, 8, 7, 3, 5, 3, 4, 2]
[ 삭제 후 ]:  [8, 5, 7, 3, 2, 3, 4]
[ 삭제 후 ]:  [7, 5, 4, 3, 2, 3]
```

힙의 복잡도 분석



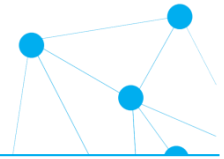
- 삽입 연산에서 최악의 경우
  - 루트 노드까지 올라가야 하므로 트리의 높이에 해당하는 비교 연산 및 이동 연산이 필요하다.

  ➔ O(logn)

- 삭제연산 최악의 경우
  - 가장 아래 레벨까지 내려가야 하므로 역시 트리의 높이 만큼의 시간이 걸린다.

  ➔ O(logn)

8.6 힙의 응용: 허프만 코드



- 허프만 코드란?

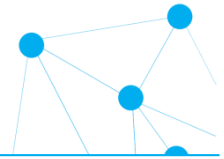
- 허프만 코딩 트리 생성 프로그램

허프만 코드란?

- 이진 트리는 각 글자의 빈도가 알려져 있는 메시지의 내용을 압축하는데 사용될 수 있음
- 이런 종류의 이진트리 ➔ 허프만 코딩 트리

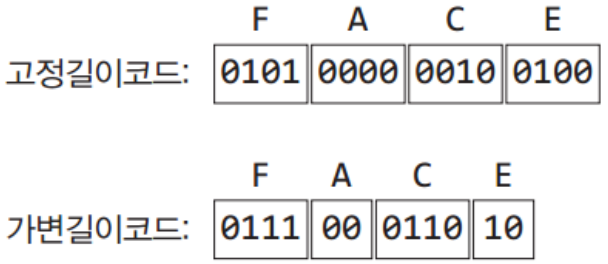
A	80
B	16
C	32
D	36
E	123
F	22
G	16
H	51
I	71
...	
Z	1

문자의 빈도수

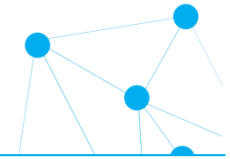


- 고정길이코드와 가변길이코드의 비교

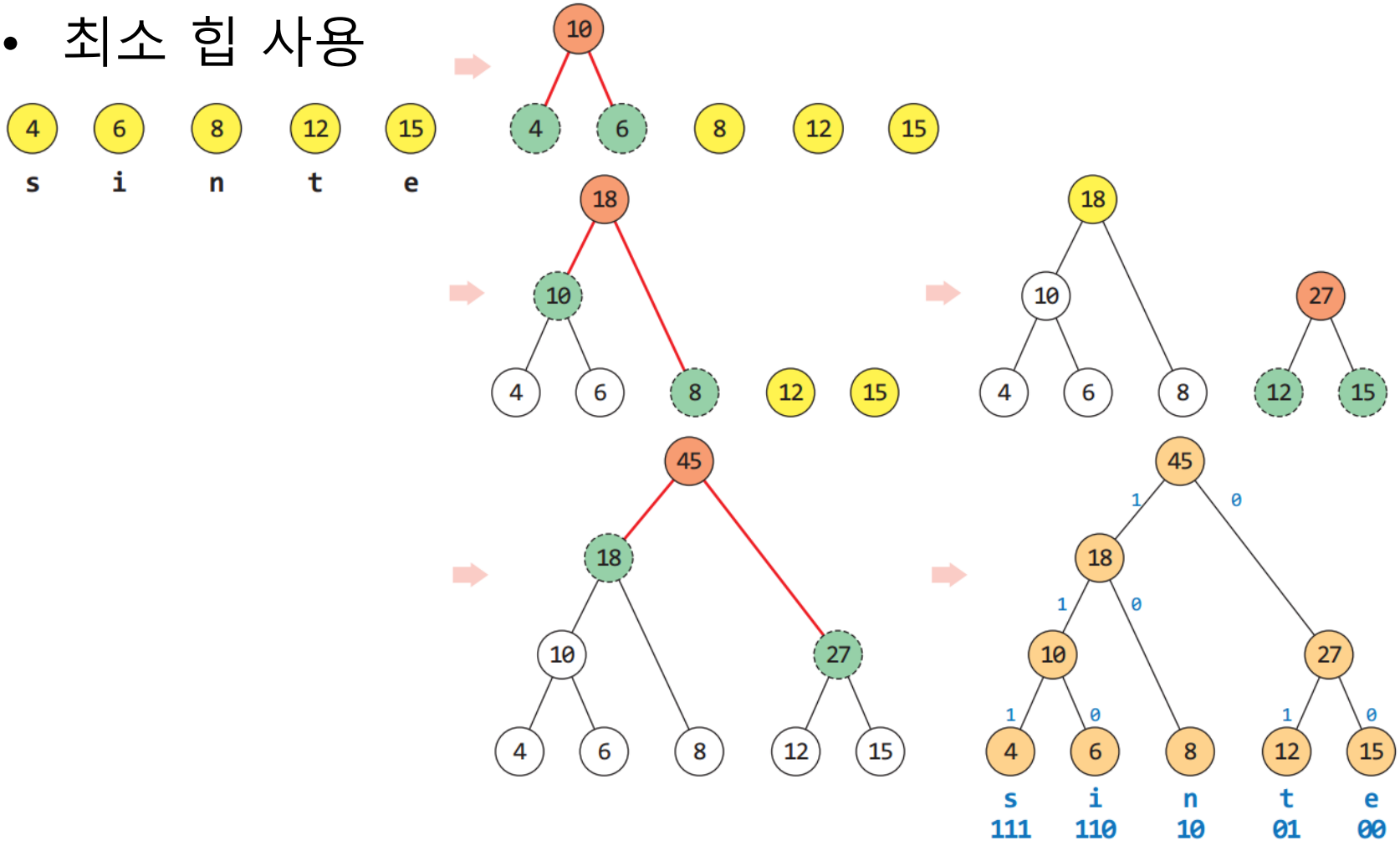
글자	빈도수	고정길이코드			가변길이코드		
		코드	비트수	전체 비트수	코드	비트수	전체 비트수
A	17	0000	4	68	00	2	34
B	3	0001	4	12	11110	5	15
C	6	0010	4	24	0110	4	24
D	9	0011	4	36	1110	4	36
E	27	0100	4	108	10	2	54
F	5	0101	4	20	0111	4	20
G	4	0110	4	16	11110	5	20
H	13	0111	4	52	010	3	39
I	15	1000	4	60	110	3	45
J	1	1001	4	4	11111	5	5
합계	100			400			292



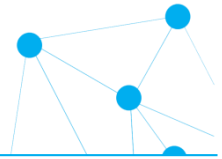
허프만 코드 생성 방법



- 최소 힙 사용



허프만 코딩 트리 생성 프로그램

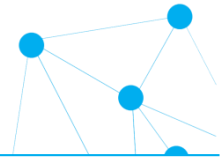


```
def make_tree(freq):
    heap = MinHeap()
    for n in freq :
        heap.insert(n)

    for i in range(0, n) :
        e1 = heap.delete()
        e2 = heap.delete()
        heap.insert(e1 + e2)
        print(" (%d+%d)" % (e1, e2))
```

```
C:₩WINDOWS₩system32₩cmd.exe
(4+6)
(8+10)
(12+15)
(18+27)
```

```
label = [ 'E', 'T', 'N', 'I', 'S' ]
freq  = [15, 12, 8, 6, 4 ]
make_tree(freq)
```

8장 연습문제, 실습문제



감사합니다!