16.2 Elements of Greedy Algorithms

- greedy algorithm 만들기

1. 하나의 (greedy) 선택을 하면 나머지 부분도 하나의 subproblem 만 남도록 최적화 문제를 세워라.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice.
3. Demonstrate optimal substructure. (greedy choice 와 subproblem 의 optimal solution 을 결합하면 전체 문제의 optimal solution 을 얻는다는 것을 보임)

When can we use a greedy algorithm?

- greedy-choice property + optimal substructure

Greedy-choice property

"We can assemble a globally optimal solution by making locally optimal (=greedy) choices."

= 어떤 선택을 할지 고려할 때 부분 문제들의 결과를 고려할 필요없이 현재 고려 중인 문제에서 최적인 문제를 선택해도 된다.

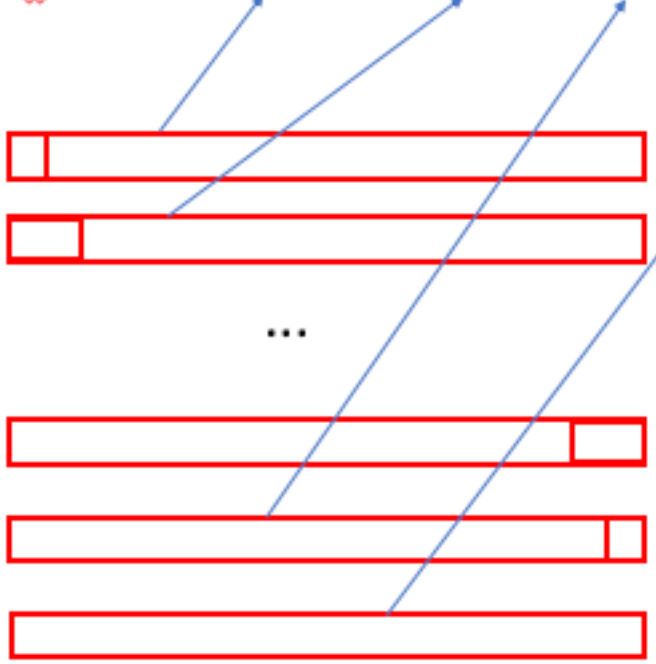
- difference between dynamic programming and greedy algorithm
  - dynamic programming : subproblem 들의 해를 먼저 구한다. (bottom-up)
  - greedy algorithm : choice 를 먼저 한 다음 나머지 subproblem 을 푼다. (top-down)

Greedy-choice property

"We can assemble a globally optimal solution by making locally optimal (=greedy) choices."

- 예를 들면 rod-cutting problem 은 greedy-choice property 를 가지지 않았으므로 dynamic programming 으로 풀어야한다.

$$r_n = \max(p_1+r_{n-1}, p_2+r_{n-2}, \ldots, p_{n-1}+r_1, p_n)$$



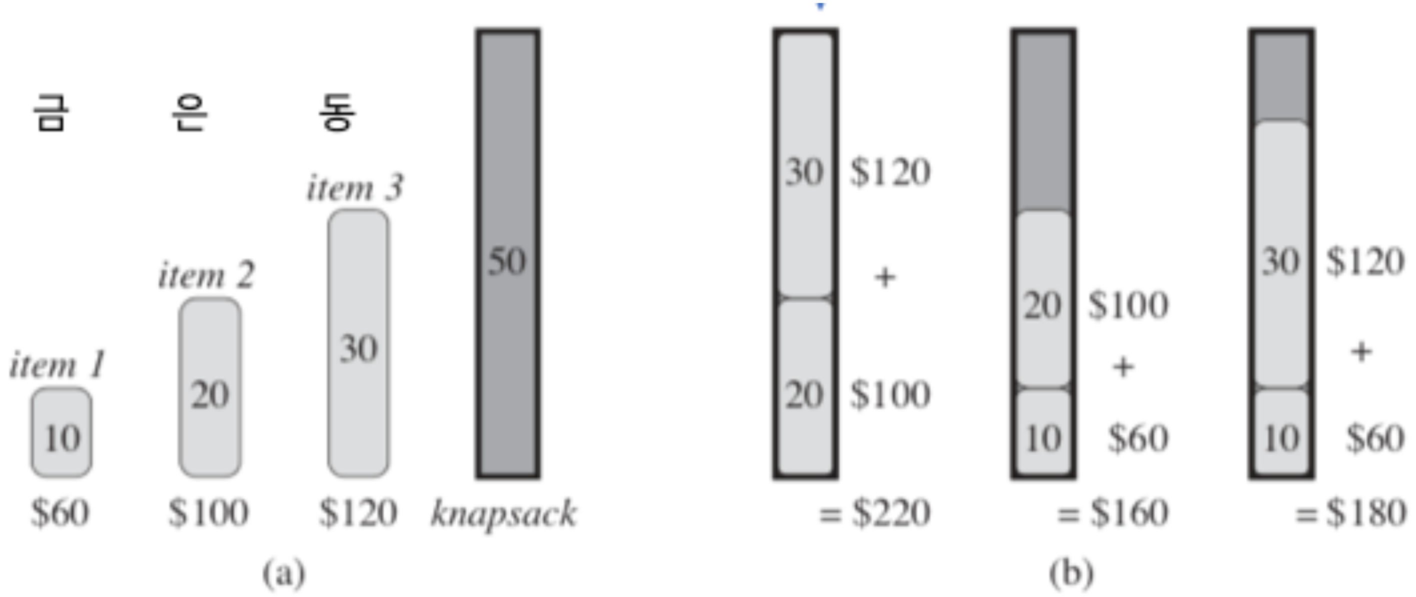
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Optimal Substructure

“An optimal solution to the problem contains optimal solutions to subproblems.”

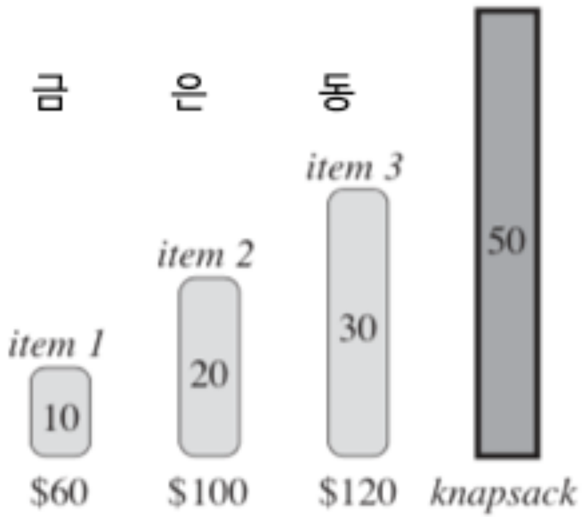
0-1 knapsack problem

- n items : w_i is a weight of i-th item, v_i is a value of i-th item
- W : knapsack capacity
- problem : choose a set of items maximizing total value, and not exceeding knapsack capacity

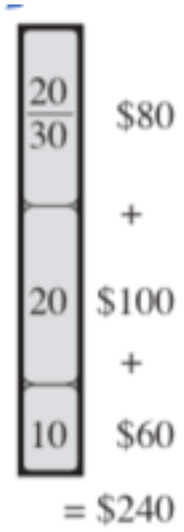


fractional knapsack problem

- n items : w_i is a weight of i-th item, v_i is a value of i-th item
- W : knapsack capacity
- problem : choose a set of fractional items maximizing total value, and not exceeding knapsack capacity



(a)



(c)

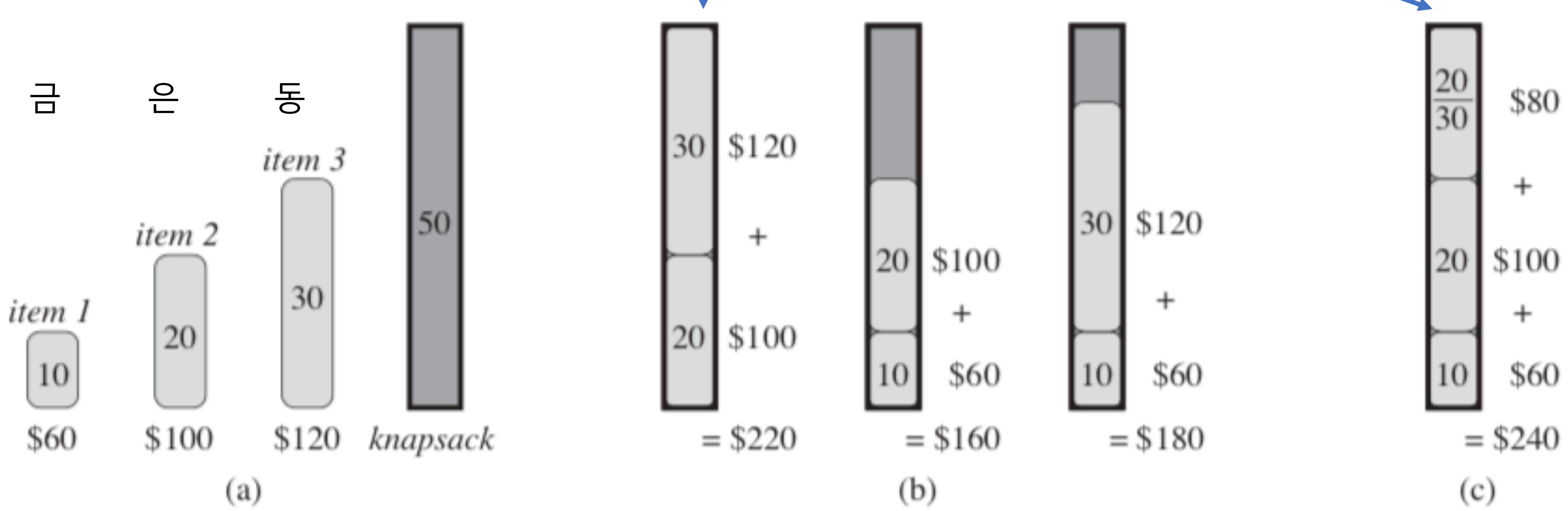
- 0-1 knapsack problem has optimal substructure, but not greedy-choice property → dynamic programming
- fractional knapsack problem has optimal substructure, and greedy-choice property → greedy algorithm $O(n \lg n)$

  1. sort items in value/weight ($=v_i/w_i$)
  2. FRACTIONAL-KNAPSACK(v, w, W)

```
load = 0
i = 1
while load < W and i ≤ n
    if w_i ≤ W − load
        take all of item i
    else take (W − load)/w_i of item i
    add what was taken to load
    i = i + 1
```

Greedy vs. dynamic programming

- greedy algorithm 으로 충분한데 dynamic programming 으로 풀려고 하거나 → i.e. fractional knapsack problem
- dynamic programming 으로 풀어야하는데 greedy algorithm 으로 풀려고 하거나 → i.e. 0-1 knapsack problem



16.3 Huffman codes (for data compression)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- a~f 문자를 100,000 개 포함한 화일의 길이
  - fixed-length (3-bit) code 사용 : 3 x 100,000 = 300,000 bits
  - variable-length code 를 사용하여 화일 크기를 줄일 수 있다.

    1x45000 + 3x(13+12+16)x1000 + 4x(9+5)x1000 = 224,000 bits
- variable-length code 는 codeword 의 끝을 어떻게 알 수 있을까?

001011101 → aabe

Prefix code

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100
					1011	

- Prefix codes : 어느 codeword 도 다른 codeword 의 prefix 가 아닌 코드

(prefix-free codes). → guarantees unambiguity in decoding a variable-length code.

001011101 → 0 0 101 1101

Problem : 주어진 문자 분포에 대해 $B(T)$ 를 최소화하는 최적의 prefix code 를 만들어라.

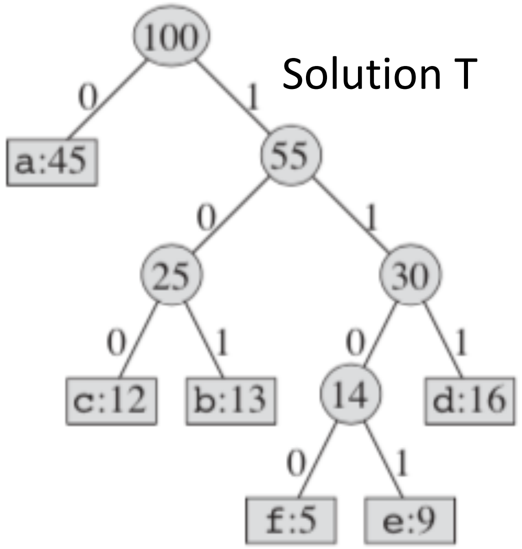
$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

C : 화일에 사용된 문자 집합
$c.freq$: 문자 c 가 화일에서 사용된 빈도 (사용된 횟수/전체문자개수)
$d_T(c)$ = 만들어진 code tree 에서 문자 c 를 나타내는 노드의 depth (root 에서부터 edge 의 갯수)

Problem C

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

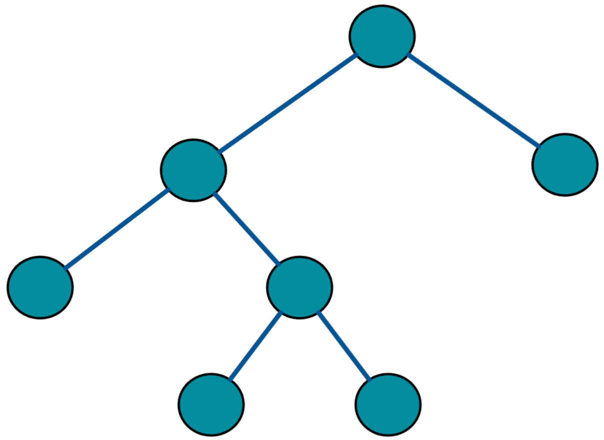
Solution T



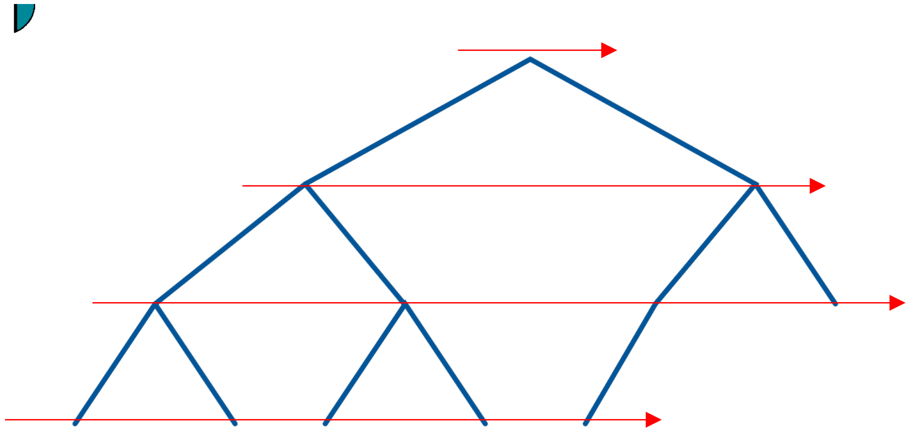
최적의 prefix code는 항상 full binary tree 로 표현된다. (exercise 16.3-2)
full binary tree : 모든 tree node 의 child 가 0개 아니면 2개인 binary tree (complete binary tree 와 다름)

types of binary tree

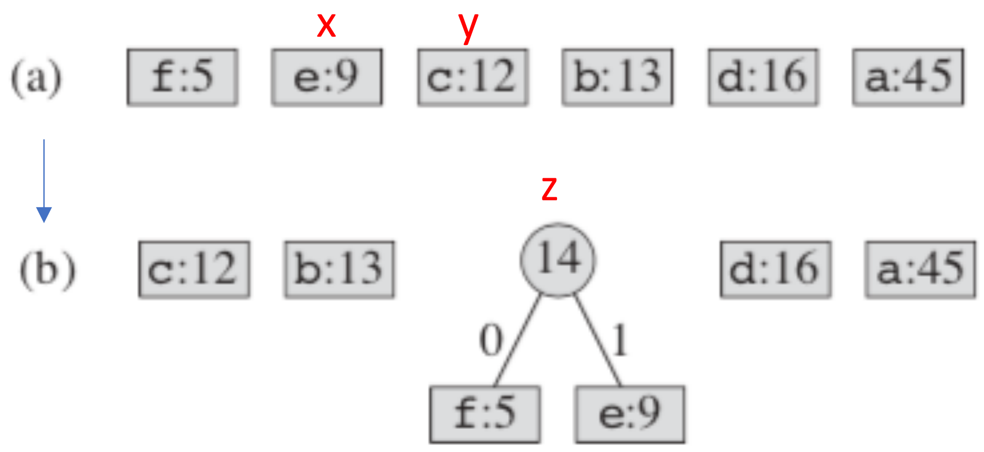
- full binary tree : 모든 노드가 0 혹은 2개의 child 를 가진 트리



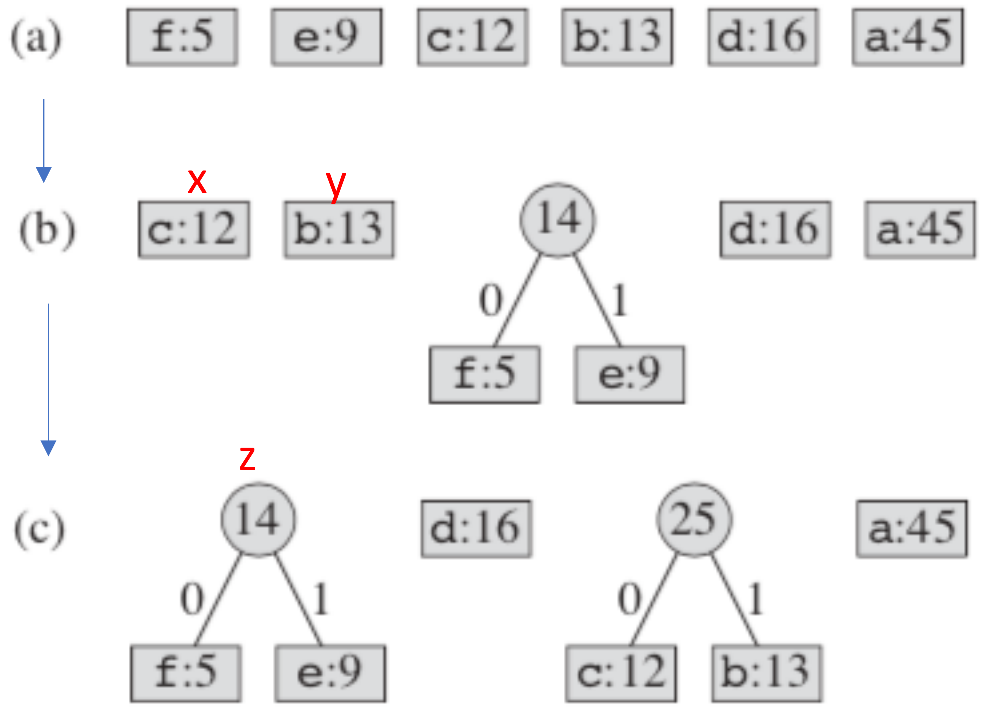
- complete binary tree (완전 이진 트리) 가장 낮은 레벨을 제외하고 모든 레벨이 완전히 차 있고 가장 낮은 레벨은 왼쪽부터 차있는 트리



```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
        allocate a new node z
        z.left = x = EXTRACT-MIN(Q)        greedy choice
        z.right = y = EXTRACT-MIN(Q)
        z.freq = x.freq + y.freq
        INSERT(Q, z)
return EXTRACT-MIN(Q)        // return the root of the tree
```

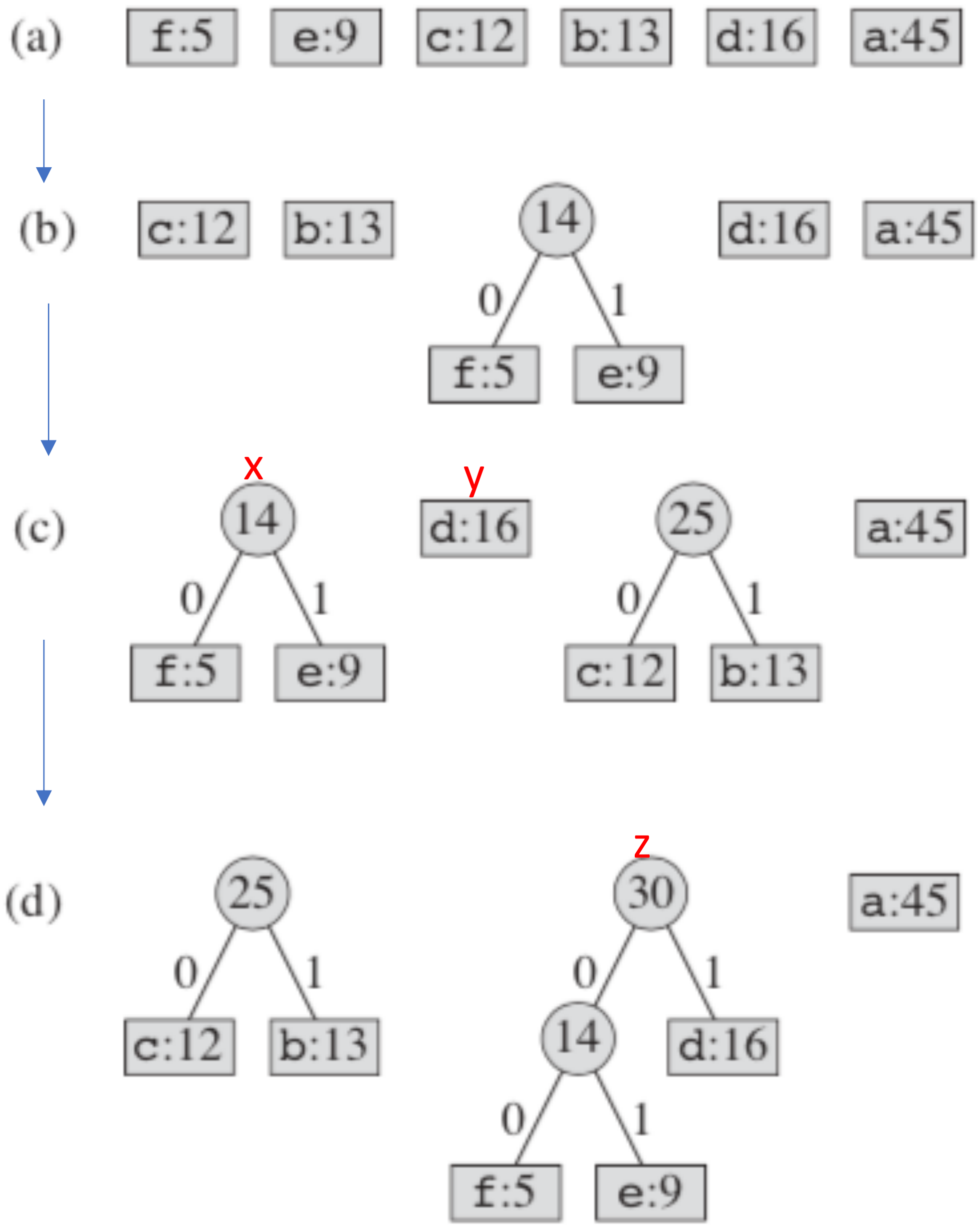


```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
        allocate a new node z
        z.left = x = EXTRACT-MIN(Q)        greedy choice
        z.right = y = EXTRACT-MIN(Q)
        z.freq = x.freq + y.freq
        INSERT(Q, z)
return EXTRACT-MIN(Q)        // return the root of the
```



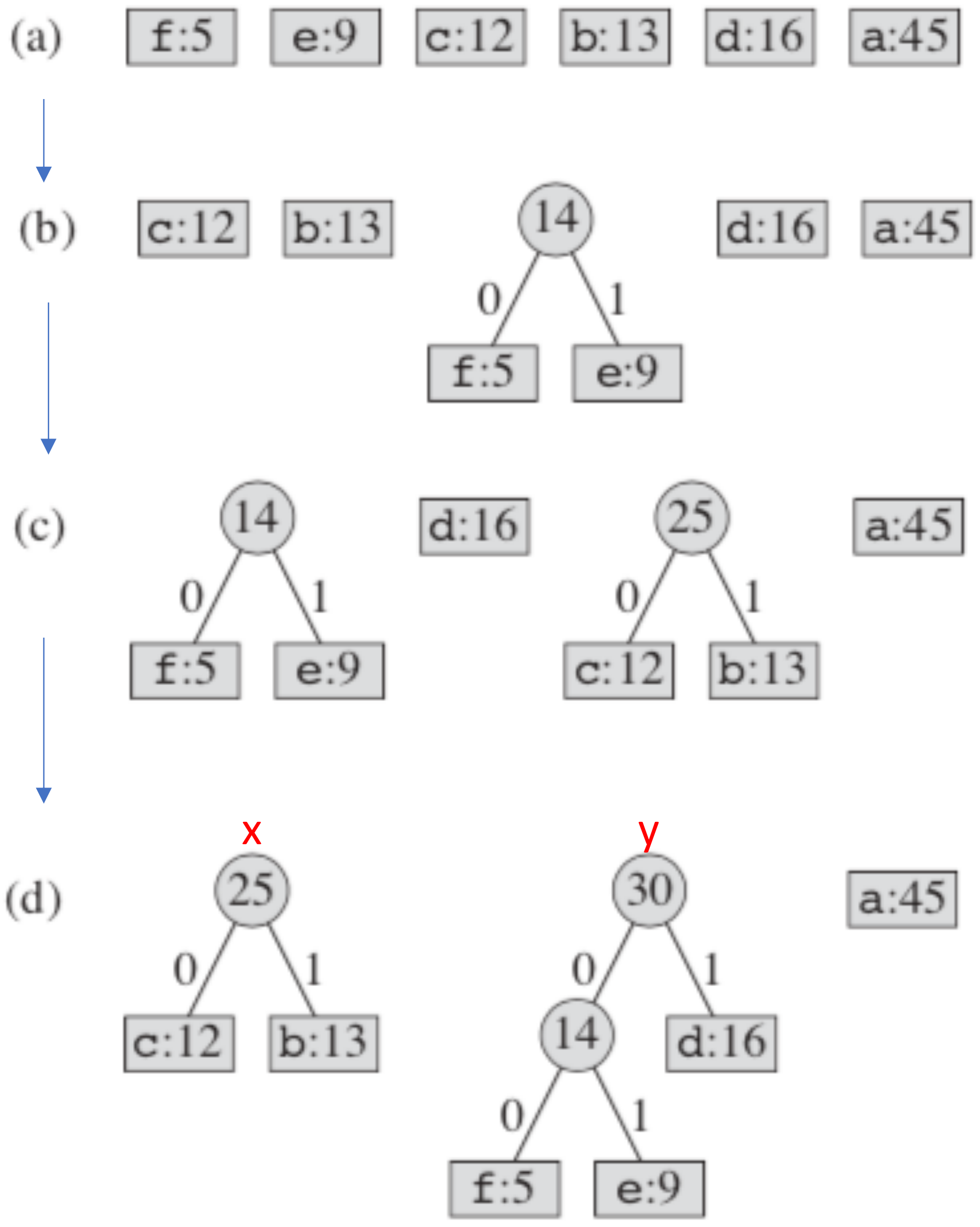
```
HUFFMAN(C)
  n = |C|
  Q = C
  for i = 1 to n - 1
      allocate a new node z
      z.left = x = EXTRACT-MIN(Q)
      z.right = y = EXTRACT-MIN(Q)
      z.freq = x.freq + y.freq
      INSERT(Q, z)
  return EXTRACT-MIN(Q)      // return the root of the
```

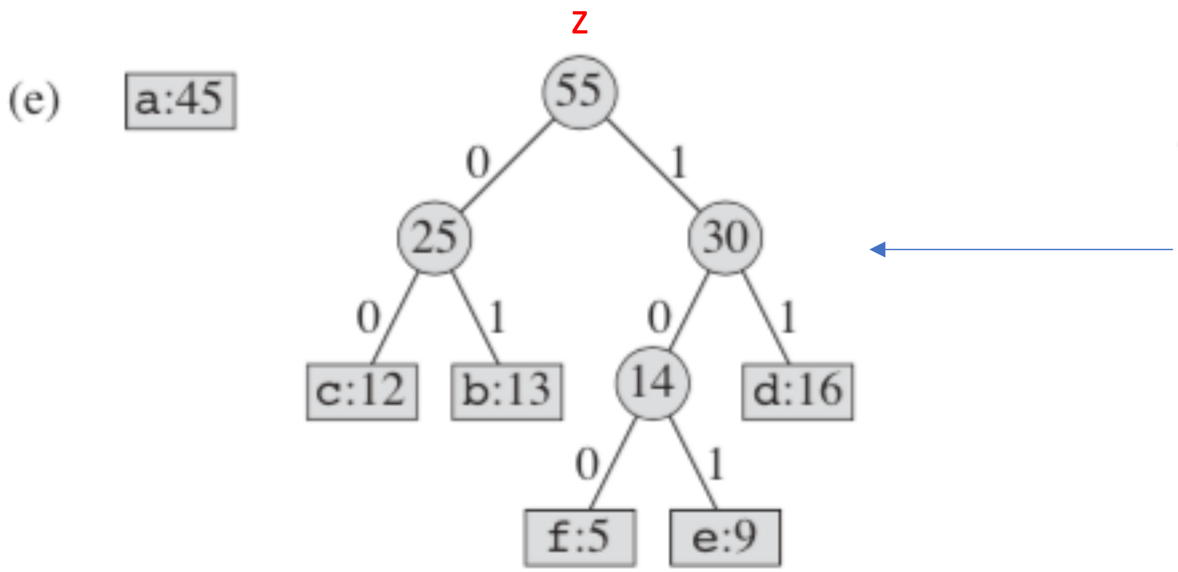
greedy choice



```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)        // return the root of the
```

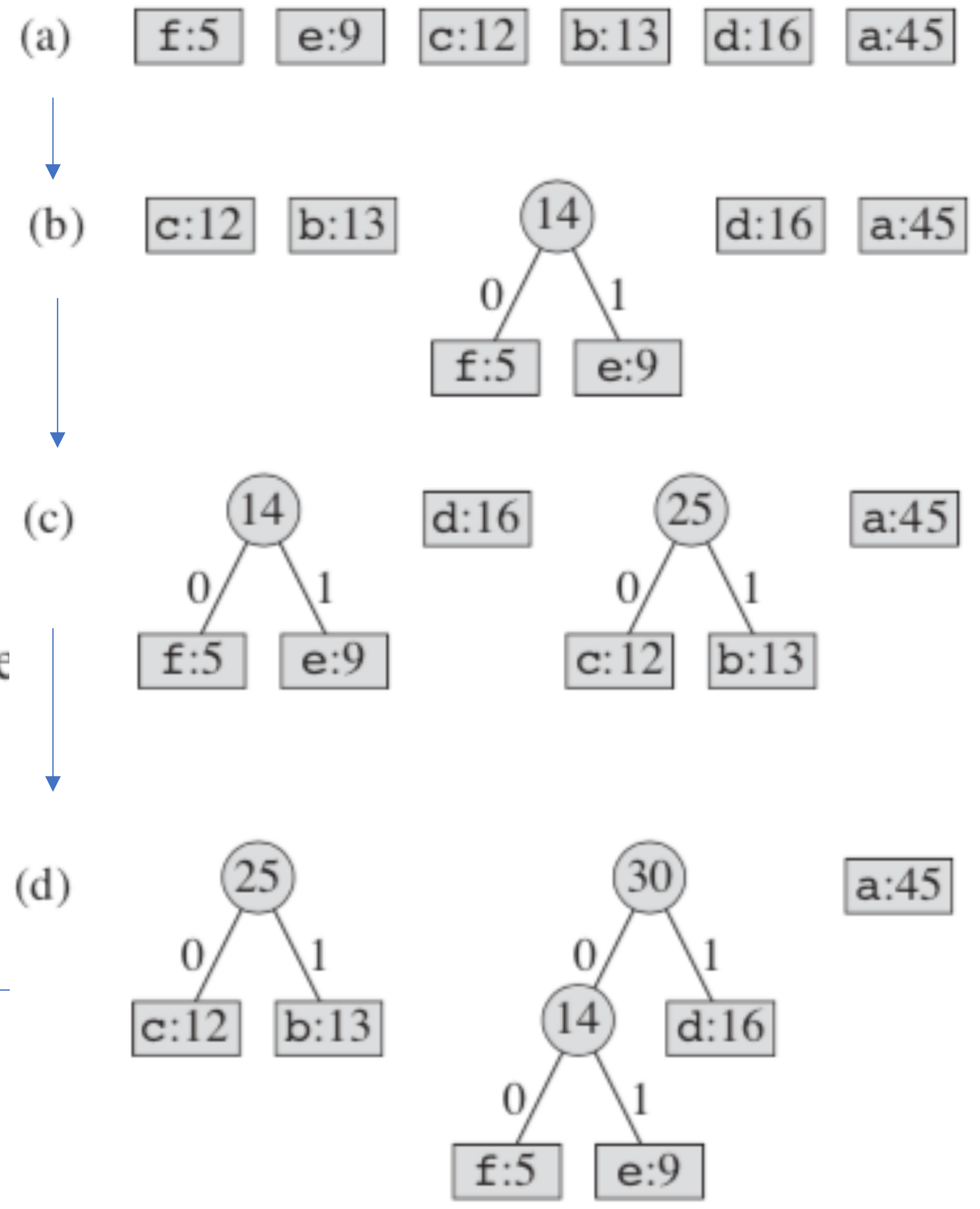
greedy choice

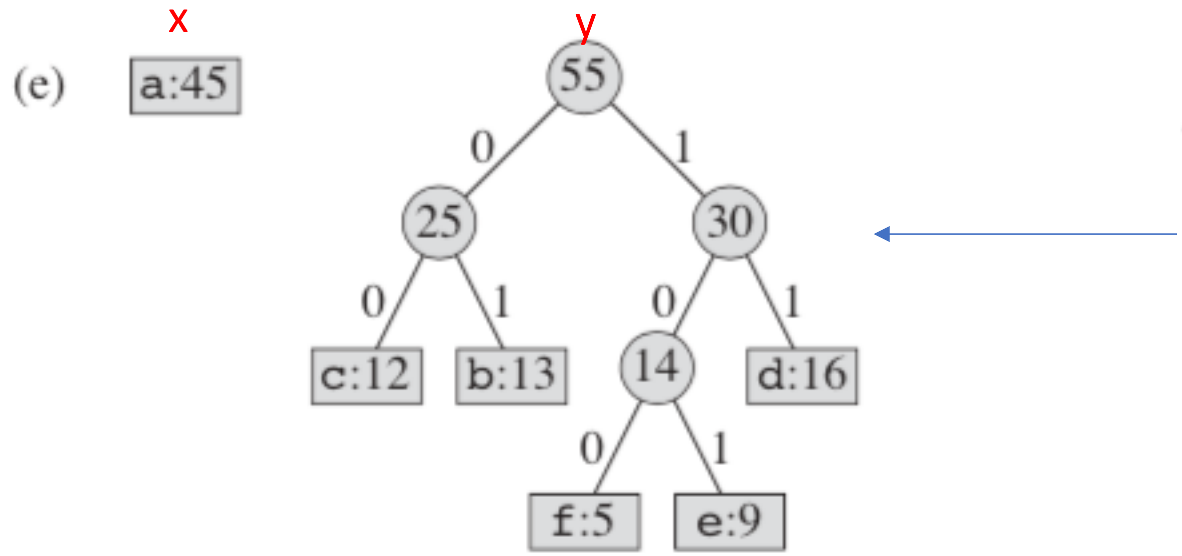


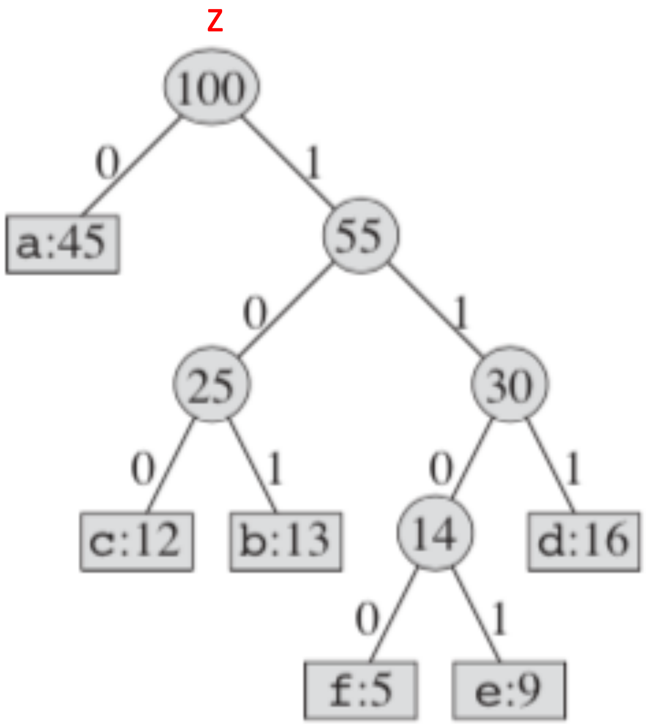


```
HUFFMAN(C)
  n = |C|
  Q = C
  for i = 1 to n - 1
      allocate a new node z
      z.left = x = EXTRACT-MIN(Q)
      z.right = y = EXTRACT-MIN(Q)
      z.freq = x.freq + y.freq
      INSERT(Q, z)
  return EXTRACT-MIN(Q)      // return the root of the
```

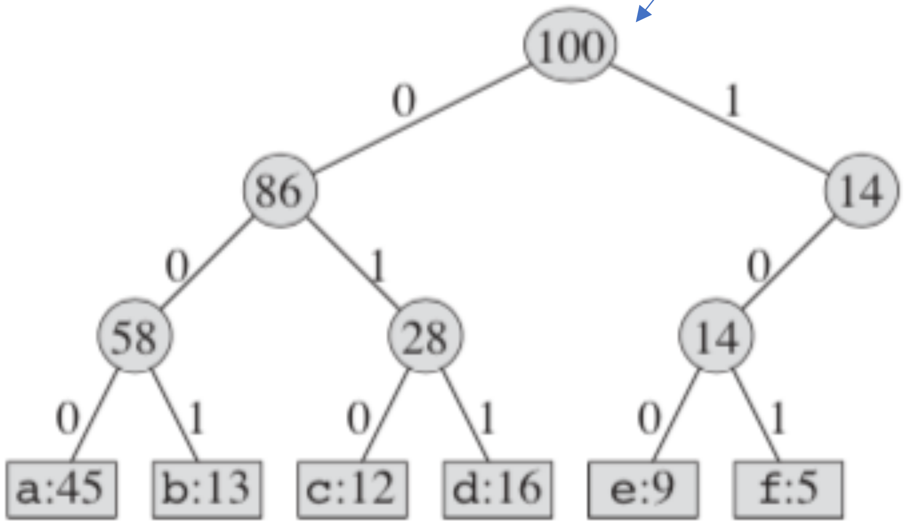
greedy choice

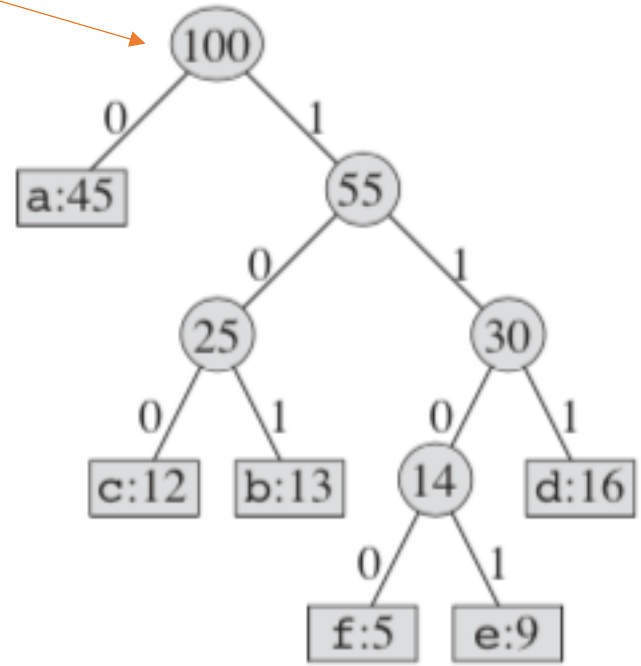





z
100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

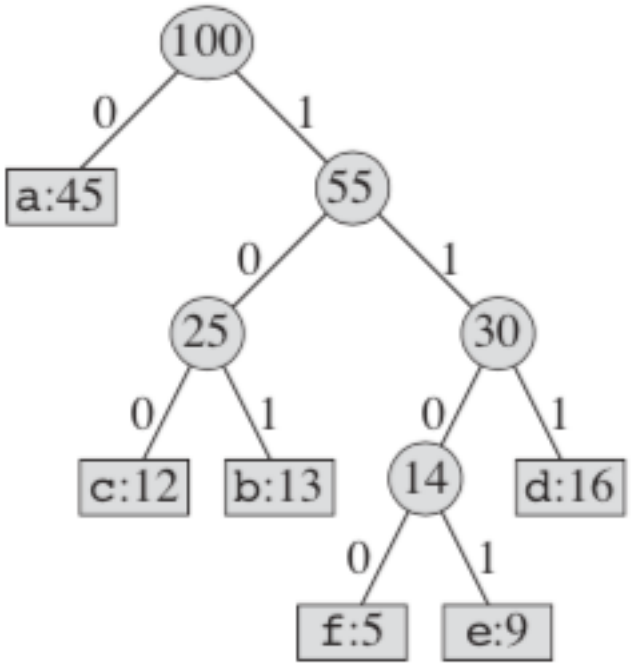




Huffman code

```
HUFFMAN(C)
n = |C|
Q = C                                  O( n ) BUILD_MIN_HEAP
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)                       O( lg n )
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return the root of the tree
```

(f)

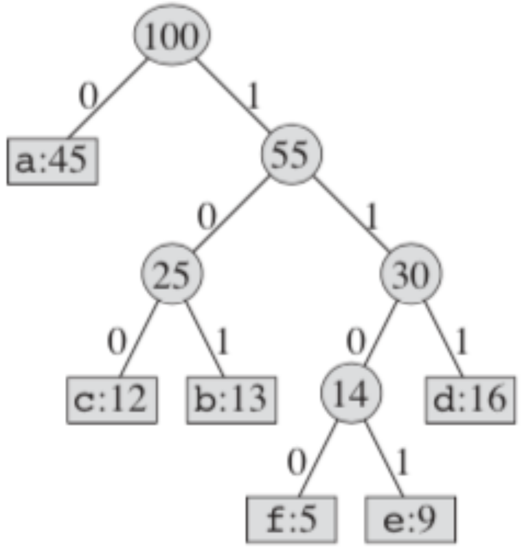


→Q is implemented in a binary min-heap

→ $O(n \lg n)$

Optimizing prefix code with respect to $B(T)$

Tree representation of code



$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

C : 화일에 사용된 문자 집합 = {a,b,c,d,e,f} in the example
$c.freq$: 문자 c 가 화일에서 사용된 빈도 (사용된 횟수/전체문자개수)
$d_T(c)$ = 만들어진 code tree 에서 문자 c 를 나타내는 노드의 depth
= root 에서부터 edge 의 갯수
= codeword 의 길이

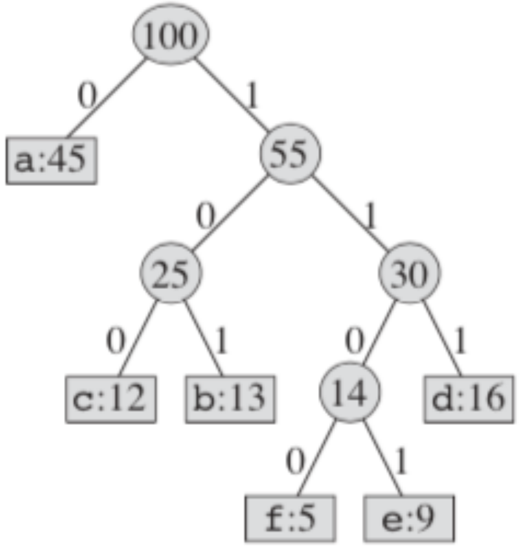
	a	b	c	d	e	f
$c.freq$	0.45	0.13	0.12	0.16	0.09	0.05
$d_T(c)$	1	3	3	3	4	4

Greedy Choice Property of Optimal Prefix Code Problem

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies.

Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



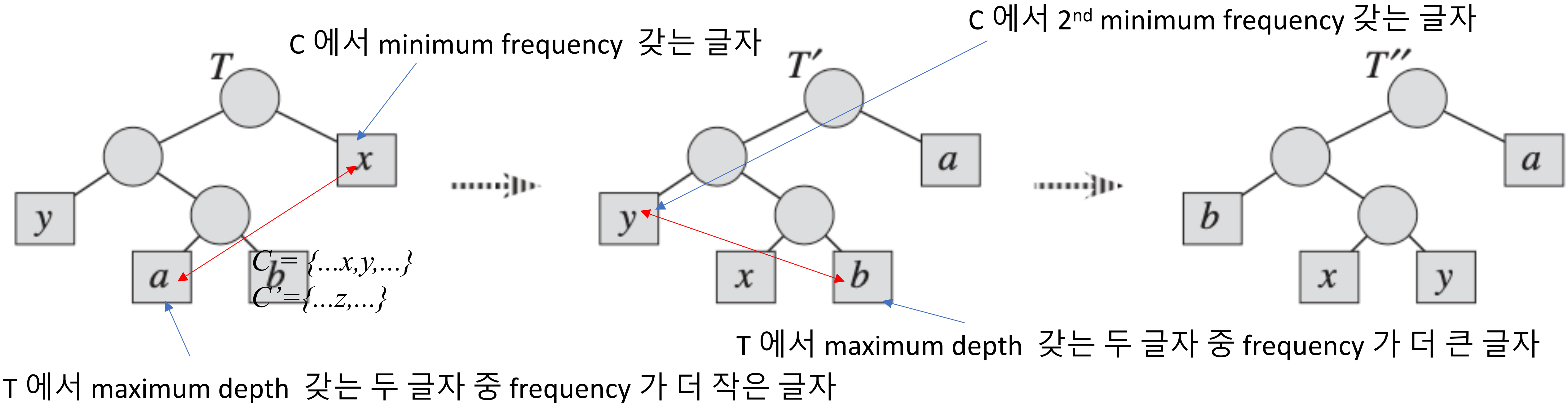
choice of x and y is a greedy choice

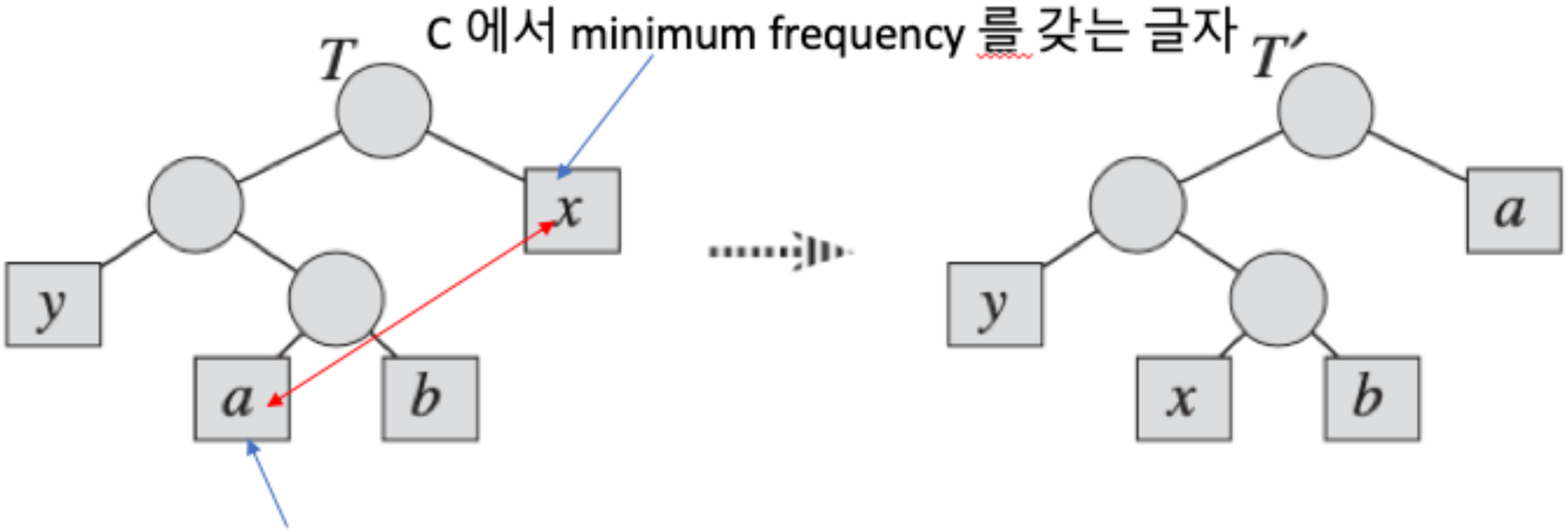
	a	b	c	d	e	f
$c.freq$	0.45	0.13	0.12	0.16	0.09	0.05
$d_T(c)$	1	3	3	3	4	4

Proof of Lemma 16.2

- 주어진 문제에 대한 임의의 optimal prefix code 를 나타내는 tree T를 변형하여 x와 y가 최대 깊이를 갖는 sibling leaf node 가 되는 T'' 을 만들면 T''도 optimal prefix code 를 나타냄을 보인다.

즉 $B(T) = B(T'')$ 임을 보임

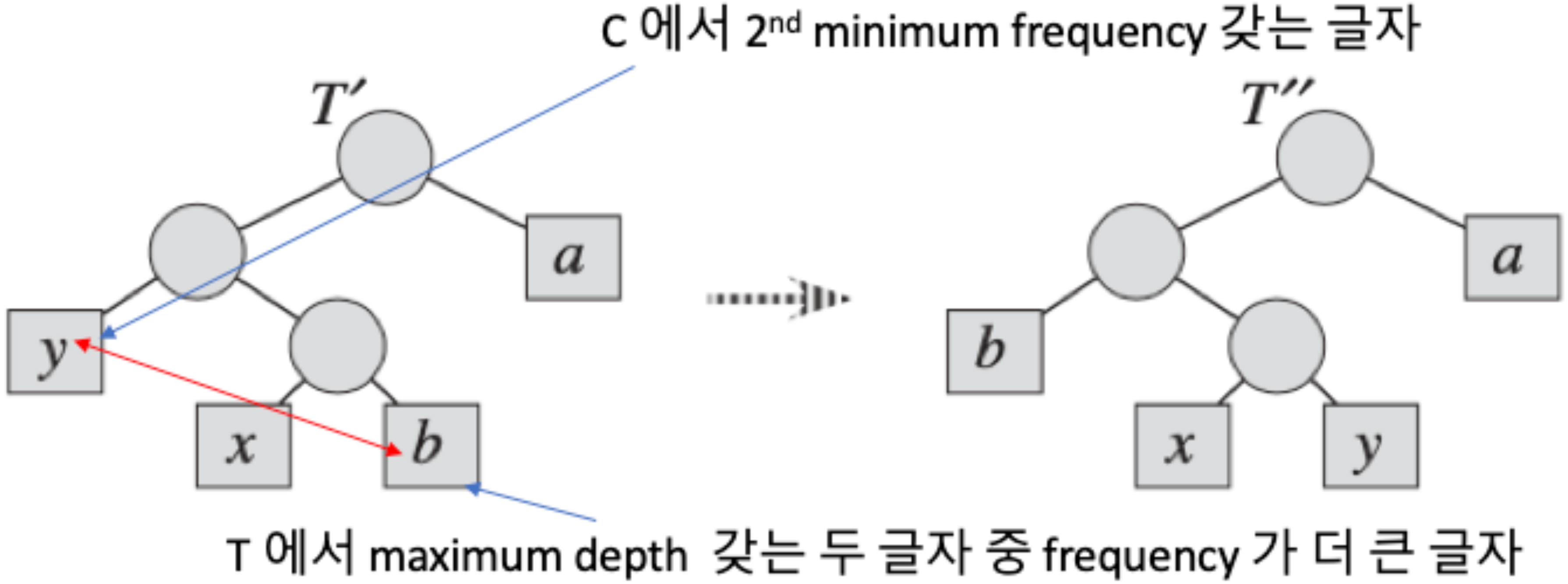




T 에서 maximum depth 갖는 두 글자 중 frequency 가 더 작은 글자

$$
\begin{aligned}
B(T) - B(T') & \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\geq 0 ,
\end{aligned}
$$

T는 optimal prefix code 를 표현하므로 $B(T) = B(T')$



마찬가지로 $B(T') = B(T'')$

따라서 $B(T) = B(T'')$

Optimal Substructure of Optimal Prefix Code Problem

Lemma 16.3

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$.

Let x and y be two characters in C having the lowest frequencies.

Let $C' = C - \{x, y\} \cup \{z\}$. In C'

$z.freq = x.freq + y.freq$ and $c.freq$ are same as in C for all other characters.

Let T' be any tree representing an optimal prefix code for C'.

Huffman code algorithm

Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

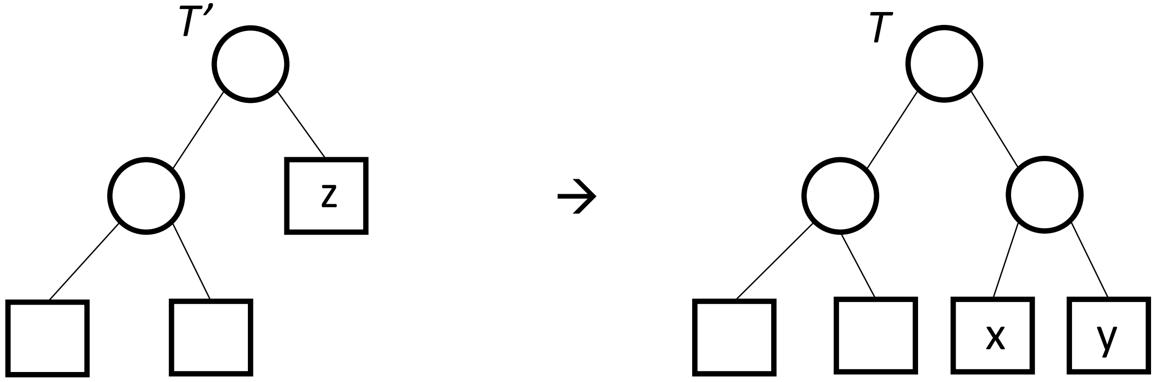
Optimal Substructure of Optimal Prefix Code Problem

Lemma 16.3

$C' = \{......,z\}$
$C=\{......,x,y\}$

T': *optimal prefix code for* C'
→ T *is an optimal prefix code for* C

In C' $z.freq = x.freq + y.freq$ and $c.freq$ are same as in C for all other characters.



Optimal Substructure of Optimal Prefix Code Problem

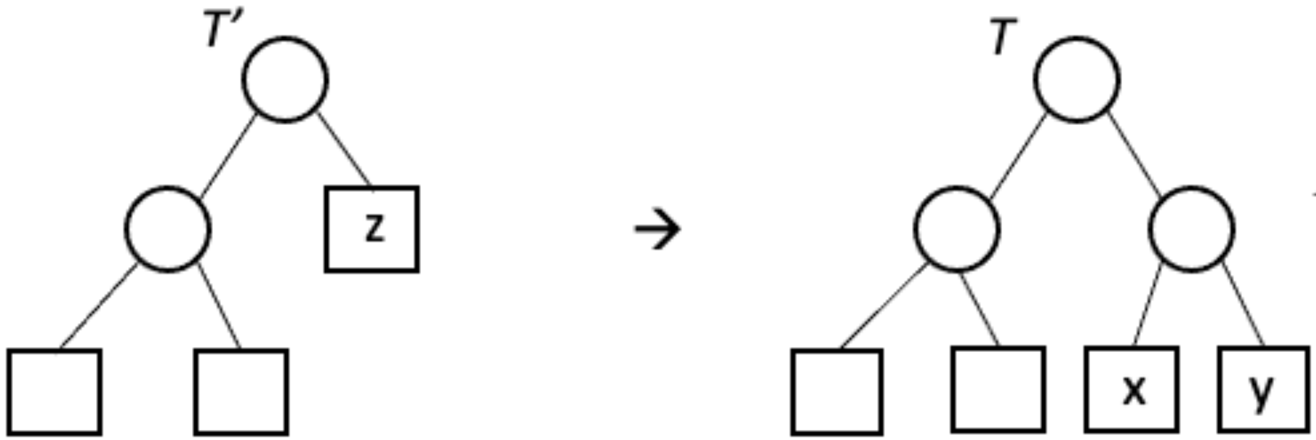
Lemma 16.3

For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c) \rightarrow \quad c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$

$d_T(x) = d_T(y) = d_{T'}(z) + 1 \longrightarrow$

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

$$B(T) = B(T') + x.freq + y.freq$$



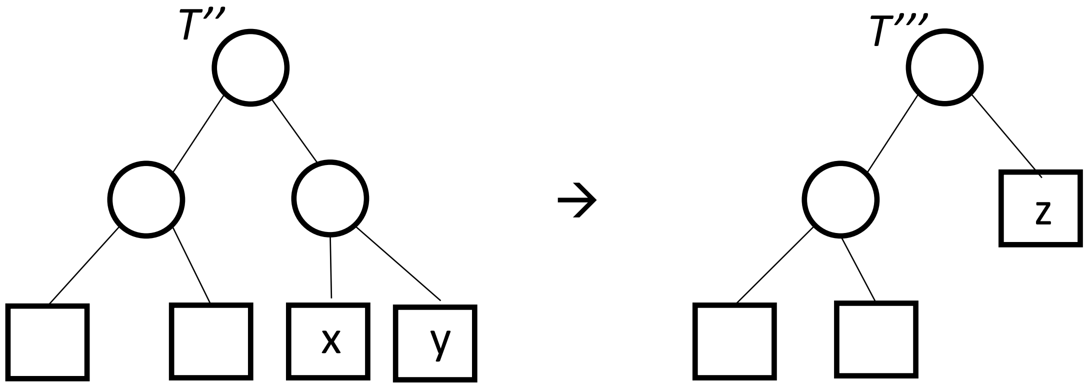
$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

Proof of Lemma 16.3 by contradiction

- T 가 C의 optimal prefix code 를 나타내지 않는다고 가정

T, T'' for C
T', T''' for C'

→ C 의 optimal prefix code 를 나타내는 T'' s.t. $B(T'') < B(T)$ 가 있음
→ T'' 은 x와 y가 sibling leaf node 임 by lemma 16.2
→ T''을 변형하여 x와 y의 공통 부모 노드를 z 로 바꾼 트리 T'''을 만들면 C'의 prefix code
→ T' 은 C'의 optimal prefix code 가 아님 (모순)



Proof of Lemma 16.3 by contradiction

- T 가 C의 optimal prefix code 를 나타내지 않는다고 가정

T, T'' for C
T', T''' for C'

→ C 의 optimal prefix code 를 나타내는 T'' s.t. $B(T'') < B(T)$ 가 있음

→ T'' 은 x와 y가 sibling leaf node 임 by lemma 16.2

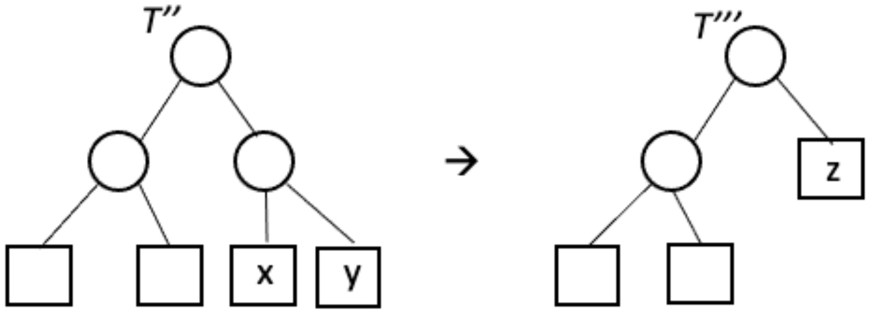
→ T''을 변형하여 x와 y의 공통 부모 노드를 z 로 바꾼 트리 T'''을 만들면 C'의 prefix code

→ $\boxed{B(T''') < B(T')}$ 라서 T'은 C'의 optimal prefix code 가 아님 (모순)

For each character $c \in C - \{x, y\}$, we have that $d_{T''}(c) = d_{T'''}(c)$ → $c.freq\ d_{T''}(c) = c.freq\ d_{T'''}(c)$

$d_{T''}(x) = d_{T''}(y) = d_{T'''}(z) + 1$ → $x.freq\ d_{T''}(x) + y.freq\ d_{T''}(y) = (x.freq + y.freq)(d_{T'''}(z) + 1)$

$= z.freq\ d_{T'''}(z) + x.freq + y.freq$



$$B(T'') = B(T''') + x.freq + y.freq$$

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T'), \end{aligned}$$

Huffman code algorithm 은 optimal prefix code 를 만든다.

- from Lemma 16.2 and 16.3

Lemma 16.3

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$.

Let x and y be two characters in C having the lowest frequencies.

Let $C' = C - \{x, y\} \cup \{z\}$. In C'

$z.freq = x.freq + y.freq$ and $c.freq$ are same as in C for all other characters.

Let T' be any tree representing an optimal prefix code for C'.

Then the tree T , obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C .

Huffman code algorithm