Huffman Coding

Another Greedy Algorithm

Encoding

	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

Encoding: Replace characters by corresponding codewords.

Example: Encode the word faded

Fixed-length Code: 101000011100011

Variable-length Code: 11000111101111

Encoding

	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

Encoding: Replace characters by corresponding codewords.

Q: How can one design a code minimizing length of encoded message?

Ex: For a file with 100,000 characters that appear with the frequencies given in the table,

The fixed-length code requires

$$3 \cdot 100{,}000 = 300{,}000 \text{ bits}$$

The variable-length code requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224{,}000 \text{ bits}$$

Decoding

Decoding: Replace codewords by corresponding characters.

$$C_1 = \{\texttt{a = 00, b = 01, c = 10, d = 11}\}.$$
$$C_2 = \{\texttt{a = 0, b = 110, c = 10, d = 111}\}.$$
$$C_3 = \{\texttt{a = 1, b = 110, c = 10, d = 111}\}$$

A message is uniquely decodable if it can only be decoded in one way.

Ex:

- Relative to C_1, **010011** is uniquely decodable to **bad**.
- Relative to C_2, **1100111** is uniquely decodable to **bad**.
- But, relative to C_3, **1101111** is not uniquely decipherable since it could have encoded to either **bad** or **acad**.

In fact, one can show that every message encoded using C_1 or C_2 is uniquely decodable.

- C_1: Because it is a fixed-length code.
- C_2: Because it is a prefix-free code.

Prefix Codes

Def: A code is called a prefix (free) code if no codeword is a prefix of another codeword.

Theorem: Every message encoded by a prefix free code is uniquely decodable.

Pf: Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Ex: code: {`a = 0, b = 110, c = 10, d = 111`}.

`01101100 = 0 110 110 0 = abba`

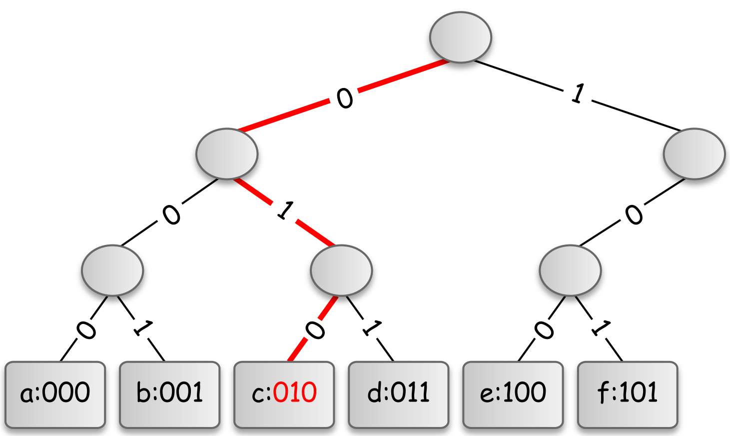
Note: There are other kinds of codes that are also uniquely decodable.

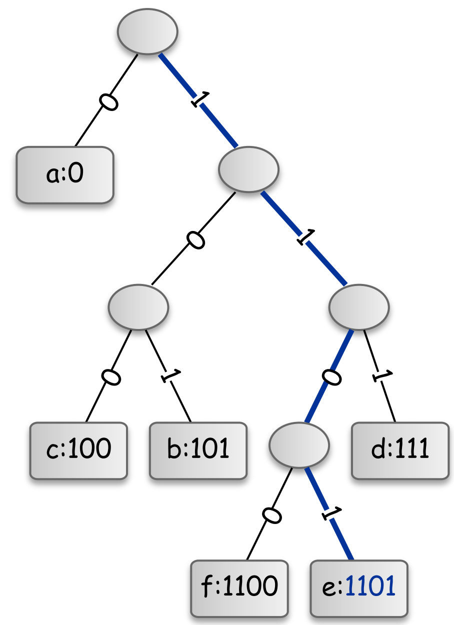
Theorem (proof omitted): The best prefix code can achieve the optimal data compression among any code that is uniquely decodable.

Problem: For a given input file, find the (a) prefix code that results in the smallest encoded message. (Compression)

Correspondence between Binary Trees and Prefix Codes

	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





Left edge labeled 0; right edge is labeled 1.

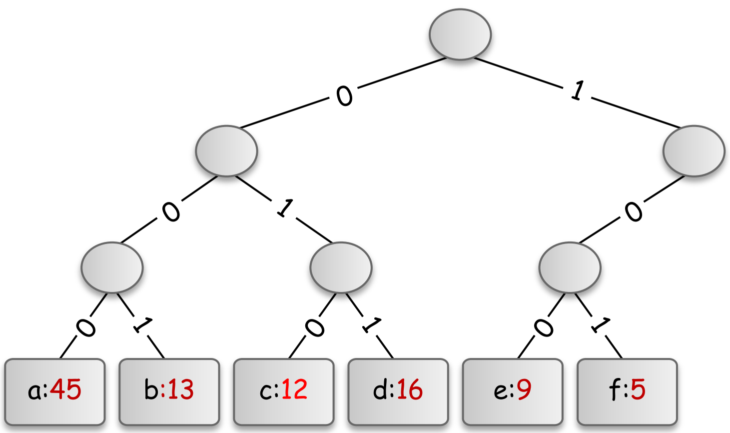
Binary string read off on path from root to a leaf
is the codeword associated with the character at that leaf.

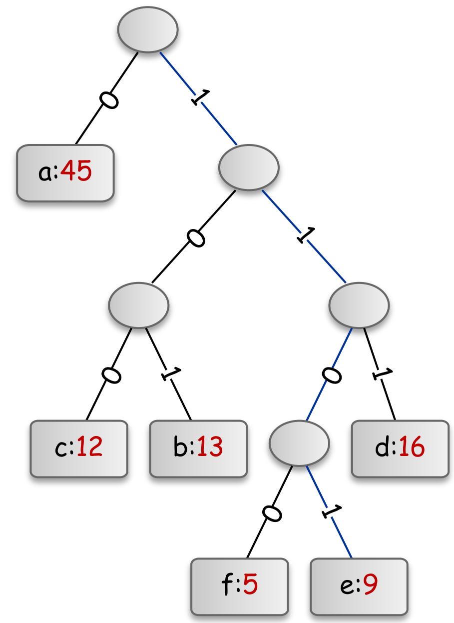
Depth of a leaf is equal to the length of the codeword.

Weighted External Path Length

Given a tree with n leaves labeled $a_1, \dots, a_n$ and associated leaf weights $f(a_1), \dots, f(a_n)$, the *weighted external path length* of the tree is

$$B(T) = \sum_{i=1}^{n} f(a_i) d(a_i)$$



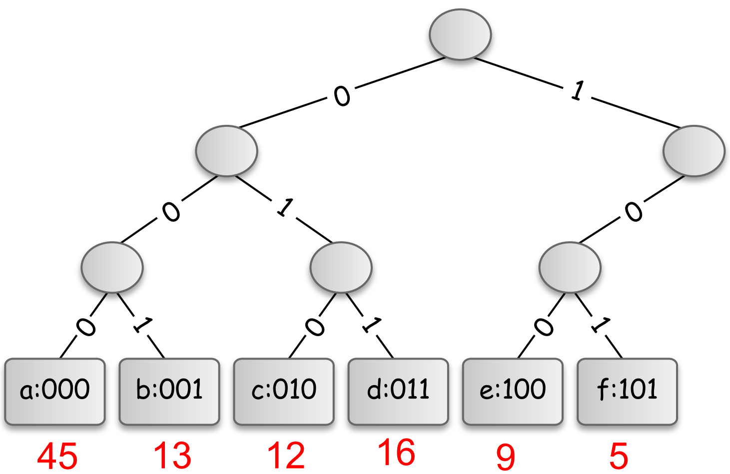


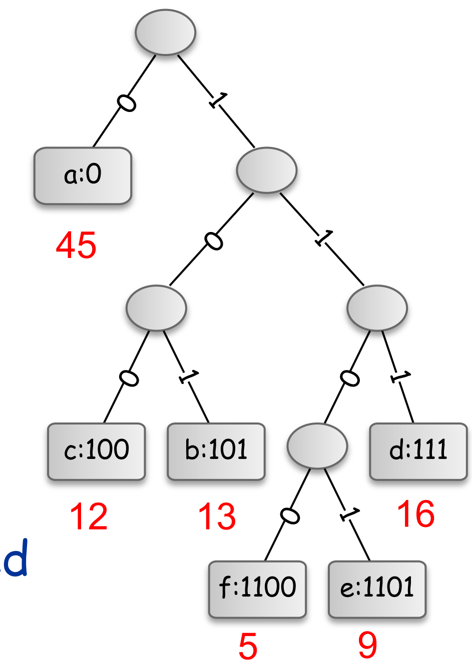
$$(45 + 13 + 12 + 16 + 9 + 5) * 3 = 300$$

$$45 * 1 + (12 + 13 + 16) * 3 + (9 + 5) * 4 = 224$$

Correspondence between Binary Trees and Prefix Codes

	a	b	c	d	e	f	Total Cost
Frequency (in thousands)	45	13	12	16	9	5	
Fixed-length codeword	000	001	010	011	100	101	300
Variable-length codeword	0	101	100	111	1101	1100	224





Set weight of leaf to be frequency of associated code word

External Weighted Path Length (cost) of tree is EXACTLY total cost of code

=> Finding min cost code is same as finding min-cost tree!

Problem Restated

Problem definition: Given an alphabet A of n characters $a_1, \ldots, a_n$ with weights $f(a_1), \ldots, f(a_n)$, find a binary tree T with n leaves labeled $a_1, \ldots, a_n$ such that

$$B(T) = \sum_{i=1}^{n} f(a_i) d(a_i)$$

is minimized, where $d(a_i)$ is the depth of a_i.

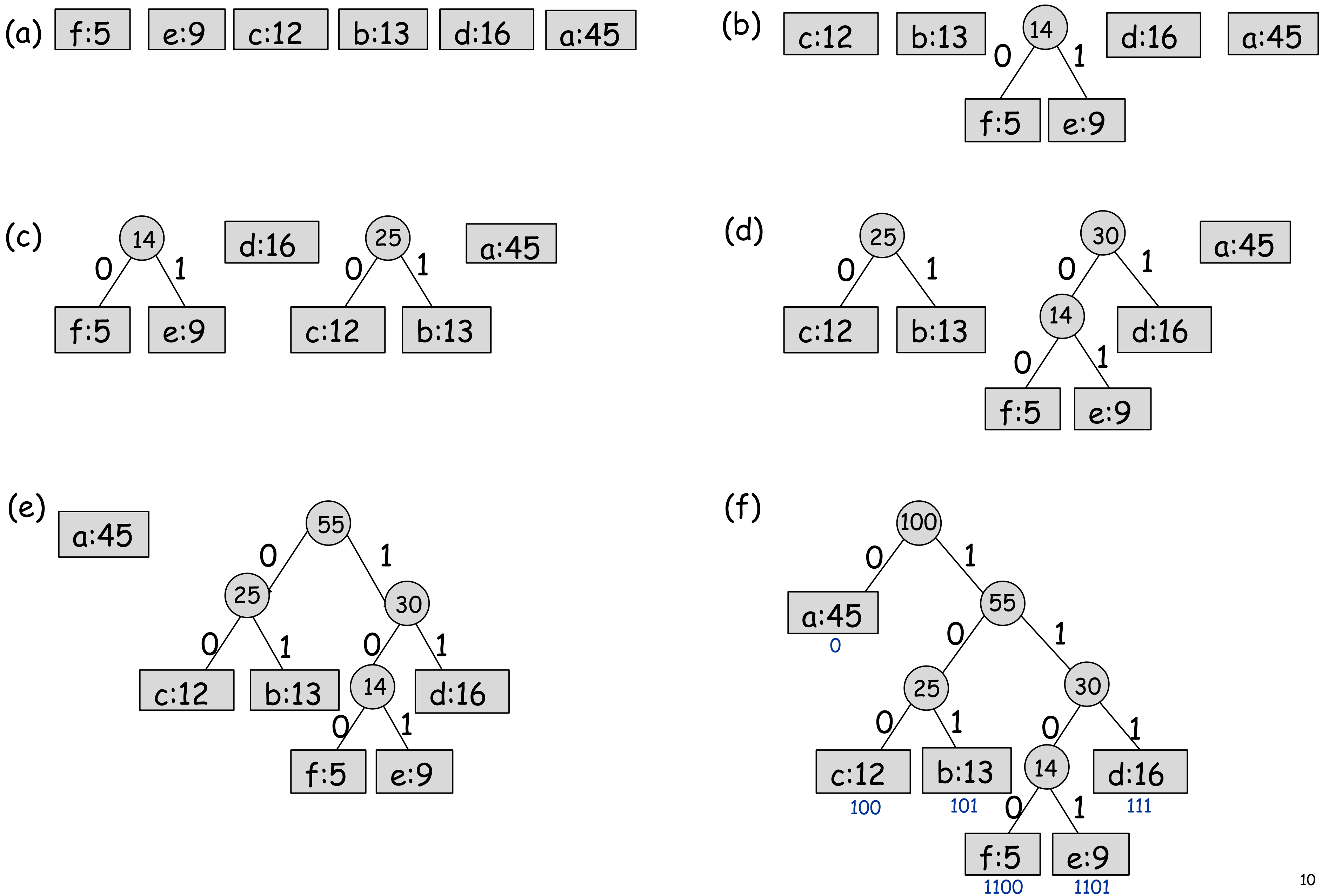
Greedy idea:

- Pick two characters x, y from A with the smallest weights
- Create a subtree that has these two characters as leaves.
- Label the root of this subtree as z.
- Set frequency $f(z) \leftarrow f(x) + f(y)$.

- Remove x, y from A and add z to A.

- Repeat the above procedure (called a "merge"), until only one character is left.

Example



(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
0
1
f:5
e:9
d:16
a:45
(e)
a:45
55
0
1
25
0
1
c:12
b:13
30
0
1
14
0
1
f:5
e:9
d:16
(f)
100
0
1
a:45
0
55
0
1
25
0
1
c:12
100
b:13
101
30
0
1
14
0
1
f:5
1100
e:9
1101
d:16
111

The Algorithm

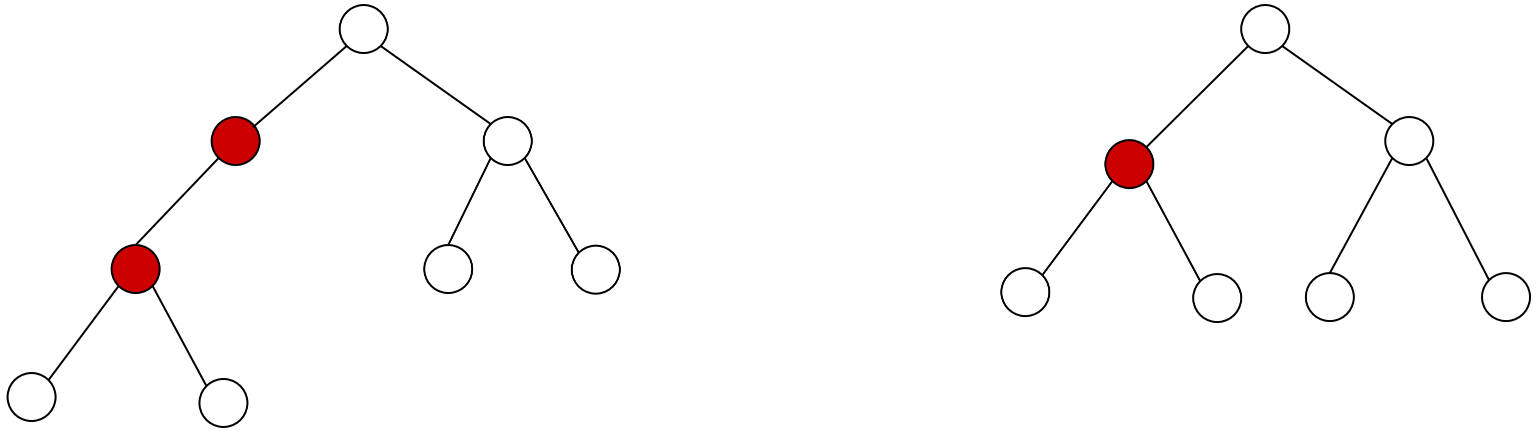
```
Huffman(A):
create a min-priority queue Q on A, with weight as key
for i ← 1 to n − 1
    allocate a new node z
    x ← Extract-Min(Q)
    y ← Extract-Min(Q)
    z.left ← x
    z.right ← y
    z.weight ← x.weight + y.weight
    Insert(Q,z)
return Extract-Min(Q) // return the root of the tree
```

Running time: $O(n \log n)$

Huffman Coding: Correctness

Lemma 1: An optimal prefix code tree must be "full", i.e., every internal node has exactly two children.

Pf: If some internal node had only one child,



then we could simply get rid of this node, replacing it with its child. This would decrease the total cost of the encoding

(because no leaf increases depth and some leaves(s) decrease depth)

Huffman Coding: Correctness

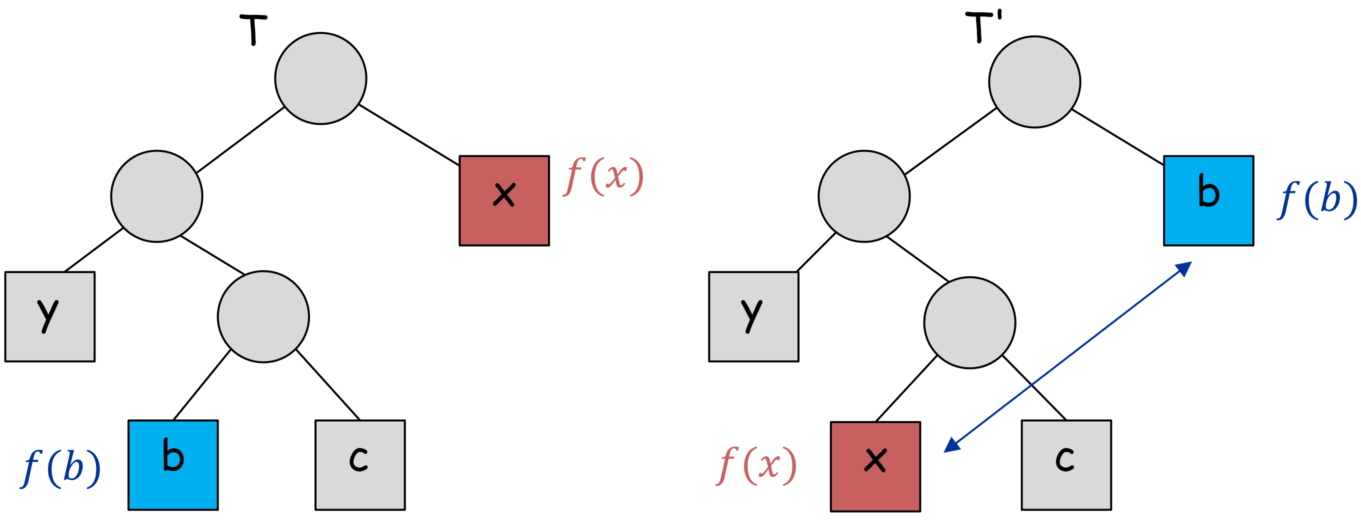
Observation: Moving a small-frequency character downward in T doesn't increase tree cost.

Lemma 2: Let T be prefix code tree and
T' be another obtained from T by swapping two leaf nodes x and b.
If,

$$f(x) \leq f(b), \qquad d(x) \leq d(b)$$

then,

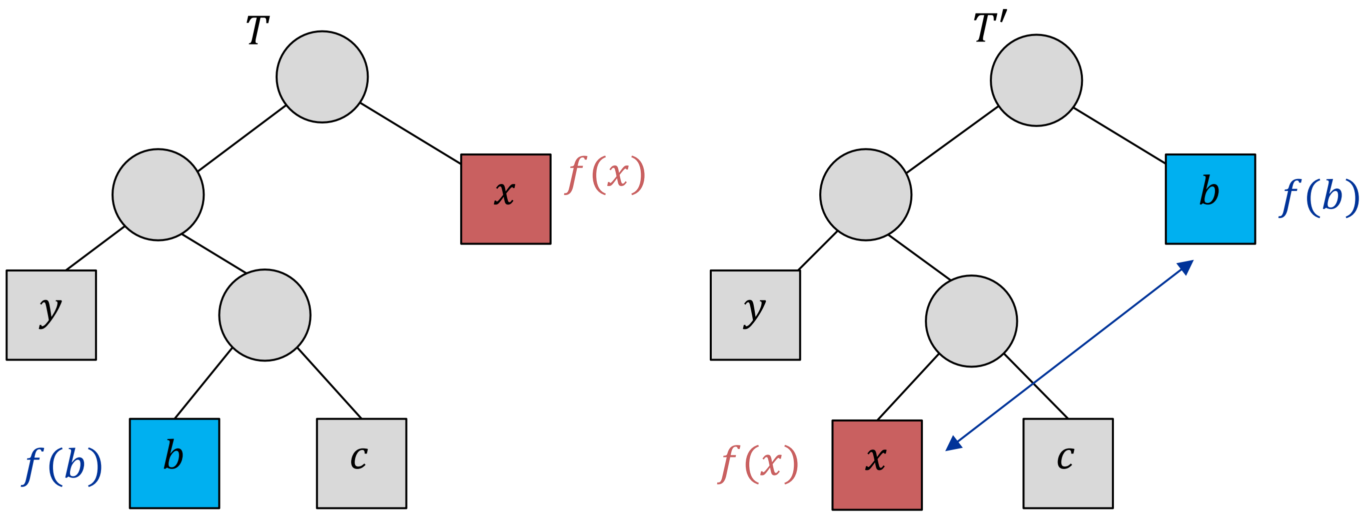
$$B(T') \leq B(T).$$



Huffman Coding: Correctness

Pf:

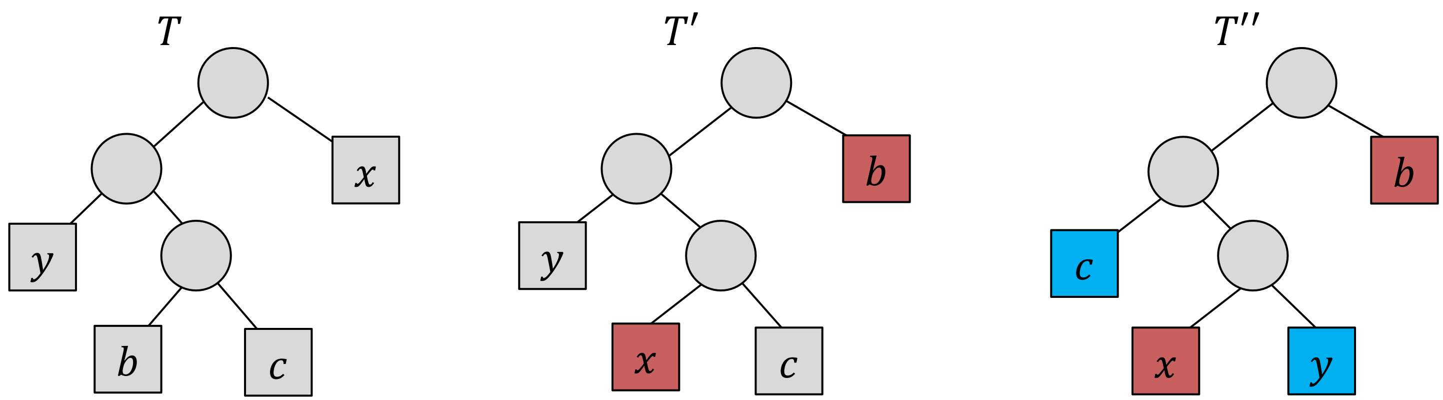
$$\begin{aligned} B(T') &= B(T) - f(x)d(x) - f(b)d(b) + f(x)d(b) + f(b)d(x) \\ &= B(T) + \underbrace{(f(x) - f(b))}_{\le 0} \cdot \underbrace{(d(b) - d(x))}_{\ge 0} \\ &\le B(T). \end{aligned}$$



Huffman Coding: Correctness

Lemma 3: Consider the two characters x and y with the smallest frequencies. There is an optimal code tree in which these two letters are sibling leaves at the deepest level of the tree.

Pf: Let T be any optimal prefix code tree, b and c be two siblings at the deepest level of the tree (must exist because T is full).



Assume without loss of generality that $f(x) \le f(b)$ and $f(y) \le f(c)$

- (If necessary) swap x with b and swap y with c.
- Proof follows from Lemma 2.

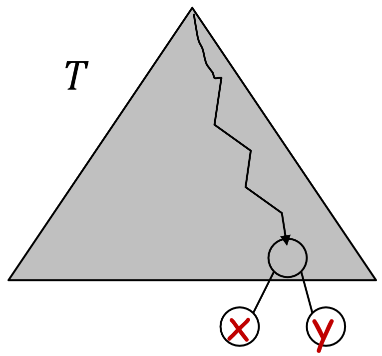
  (Lemma 2 => cost can't increase => since old tree optimal, new one is also)

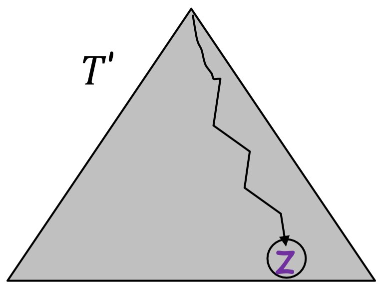
Huffman Coding: Correctness

Lemma 4: Let T be a prefix code tree in which x and y are two *sibling* leaves. Let T' be obtained from T by removing x and y, naming their parent z, and setting $f(z) = f(x) + f(y)$. Then

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

Pf: $$\begin{aligned} B(T) &= B(T') - f(z)d(z) + f(x)(d(z)+1) + f(y)(d(z)+1) \\ &= B(T') - f(z)d(z) + \big(f(x)+f(y)\big)d(z) + \big(f(x)+f(y)\big) \\ &= B(T') + f(x) + f(y). \end{aligned}$$



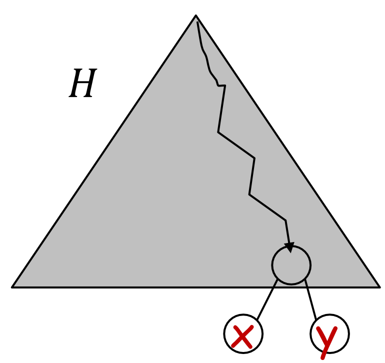


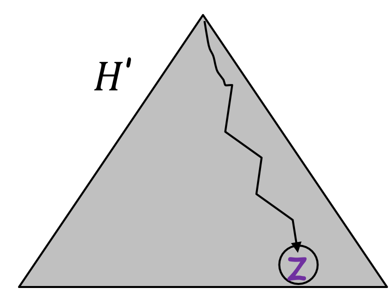
Huffman Coding: Correctness

Observation:

- Let H be the tree produced by Huffman's algorithm for alphabet A.
- Let x and y be the first two items merged together by the algorithm. Note that these are siblings in H.
- Let z be a new character with $f(z) = f(x) + f(y)$. Set $A' = A \cup \{z\} - \{x, y\}$
- Let H' be the tree obtained from H by removing x and y, naming their parent z, and setting $f(z) = f(x) + f(y)$.

Then H' is exactly the tree constructed by the Huffman algorithm on A'.





Huffman Coding: Correctness

Theorem: The Huffman tree is optimal.

Pf: (By induction on n, the number of characters)

- Base case $n = 2$: Tree with two leaves. Obviously optimal.

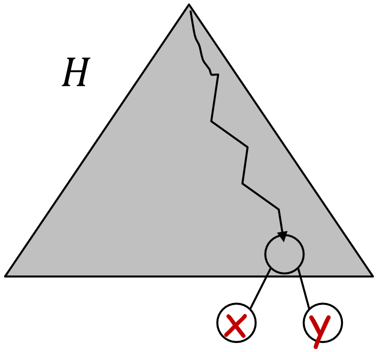
Huffman Coding: Correctness

Theorem: The Huffman tree is optimal.

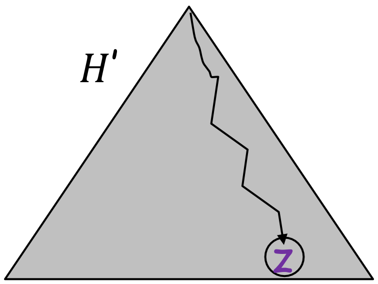
Pf: (By induction on n, the number of characters)

- Induction hypothesis: Huffman's algorithm produces optimal tree for all inputs case of $n-1$ characters.

- Induction step: Consider input of n characters:
  - Let H be the tree produced by Huffman's algorithm.
  - Need to show: H is optimal.



- From operation of Huffman's algorithm:
  - There exist two characters x and y with two smallest frequencies that are sibling leaves in H.
- Let H' be obtained from H by
  (i) removing x and y,
  (ii) naming their parent z, and
  (iii) setting $f(z) = f(x) + f(y)$

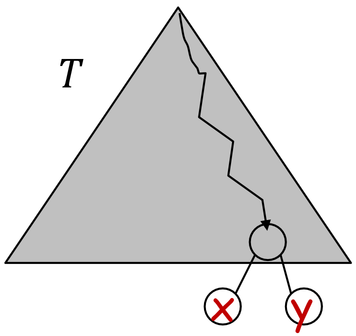


Alphabet for H: A; Alphabet for H' : $A' = A - \{x, y\} \cup \{z\}$

By Lemma 4, $B(H) = B(H') + f(x) + f(y)$.

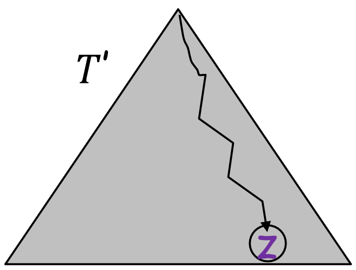
Huffman Coding: Correctness

- H is the tree produced by Huffman's algorithm for A (with x,y)
- H' is the tree produced by Huffman's algorithm for A' (with z, without x,y)
- **By the induction hypothesis, H' is optimal for A'.**



- By Lemma 3, there exists some optimal tree T for which x and y are sibling leaves.

- Let T' be obtained from T by
  (i) removing x, y,
  (ii) naming the parent z, and
  (iii) setting $f(z) = f(x) + f(y)$.
- T' is a prefix code tree for alphabet A'.



- By Lemma 4, $B(T) = B(T') + f(x) + f(y)$.

$$B(H) = B(H') + f(x) + f(y)$$
$$\leq B(T') + f(x) + f(y) \qquad (H' \text{ is optimal for } A')$$
$$= B(T).$$

=> H must be optimal!

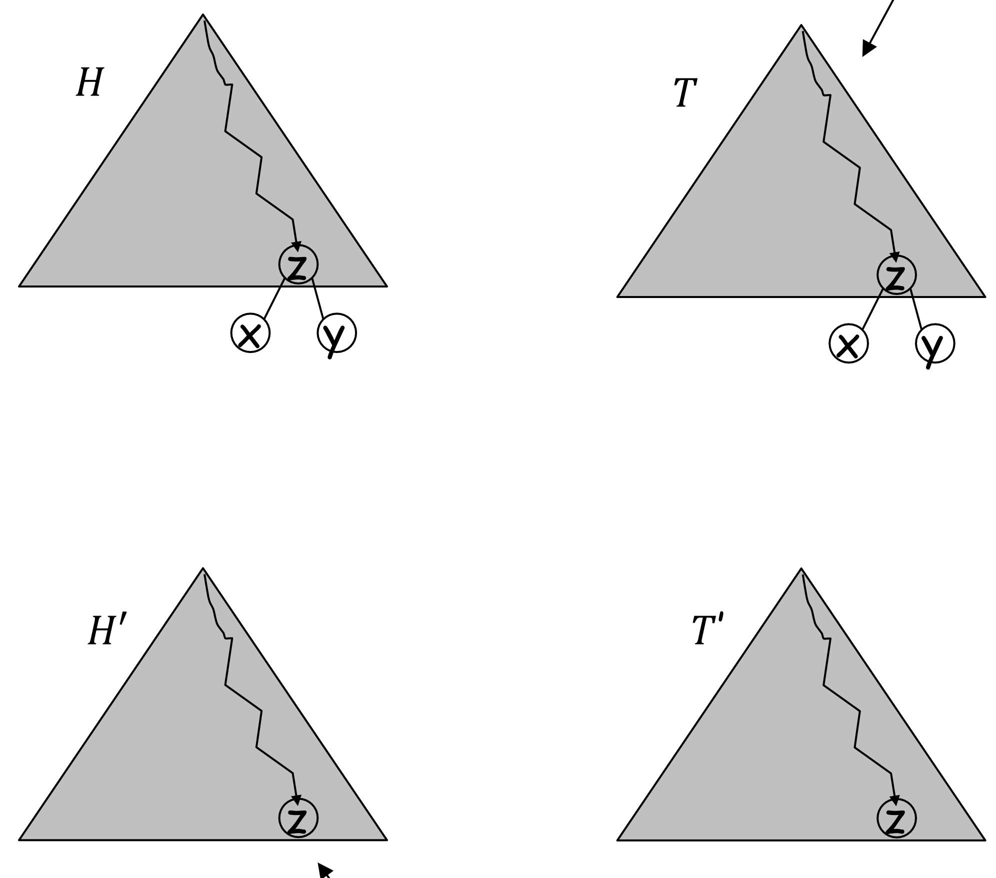
Diagram of the proof

Optimal By Assumption

A is original character set
$A' = A \cup \{z\} - \{x, y\}$

H built by Huffman Alg on A
=> H' built by Huff Alg on A'
$\Rightarrow$ By induction H' optimal for A'

$$B(H) = B(H') + f(x) + f(y)$$



T chosen as Optimal tree for A
T' built from T as tree on A'

$$B(T) = B(T') + f(x) + f(y)$$

$$\begin{aligned} B(H) &= B(H') + f(x) + f(y) \\ &\le B(T') + f(x) + f(y) \\ &= B(T). \end{aligned}$$

=> H is optimal for A

Optimal By Induction

Optimal 2-way Merge

We are given n sorted lists $L_1, L_2, \dots, L_n$, which need to be merged into a combined sorted list, but we can merge only two at a time. Find an optimal merge pattern which minimizes the total number of comparisons

Example. Suppose there are 3 sorted lists L_1, L_2, L_3 of sizes 30, 20, and 10, respectively.

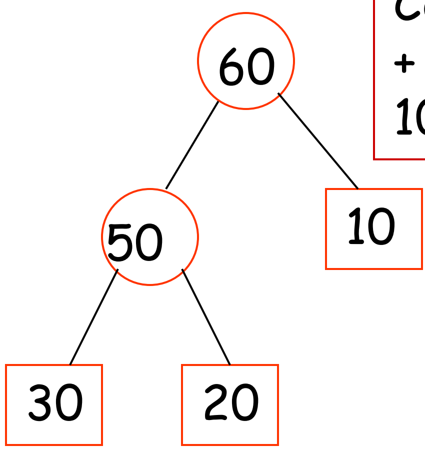
- We can first merge L_1 and L_2, which uses 30 + 20 = 50 comparisons resulting in a list of size 50. We can then merge this list with list L_3, using another 50 + 10 = 60 comparisons, so the total number of comparisons is 50 + 60 = 110.
- Alternatively, we can merge lists L_2 and L_3, using 20 + 10 = 30 comparisons, the resulting list (size 30) can then be merged with list L_1, for another 30 + 30 = 60 comparisons. So the total number of comparisons is 30 + 60 = 90.
- You could also merge L_1 and L_3, first and then the result with L_2. The total number of comparisons is 100.

Binary Merge Tree

Equivalent problem: You are given a set of leaf nodes $a_1, \ldots, a_n$ and associated leaf weights $w(a_1), \ldots, w(a_n)$ (the leaf nodes correspond to the initial lists, and the weights to their sizes).

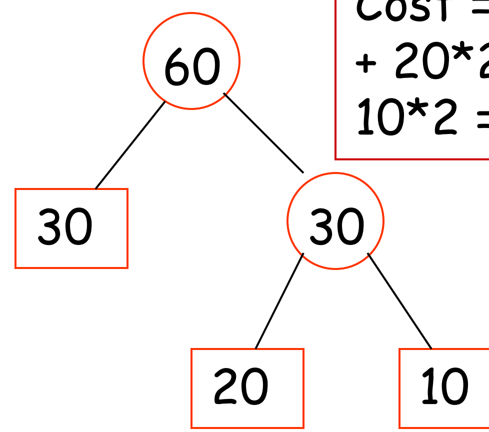
Create a binary tree from the leaf nodes towards the root, in which the size of each node is the sum of the sizes of the two children.

A binary merge tree is **optimal** if it minimizes the weighted external path length. The *weighted external path length* of the tree is $B(T) = \sum_{i=1}^{n} w(a_i) d(a_i)$



Cost = 30*2 + 20*2 + 10*1 = 110

Merge L_1 and L_2, then with L_3



Cost = 30*1 + 20*2 + 10*2 = 90

Merge L_2 and L_3, then with L_1

Optimal Binary Merge Tree Algorithm

Input: $n \geq 2$ leaf nodes, each with a size (i.e., # list elements) .

Output: a binary tree with the given leaf nodes which has a minimum total weighted external path lengths

Algorithm:

Create a min-heap $T[1..n]$ based on the n initial sizes.

While (the heap size $\geq$ 2) do

  extract from the heap two smallest values a and b

  create intermediate node of size $a + b$

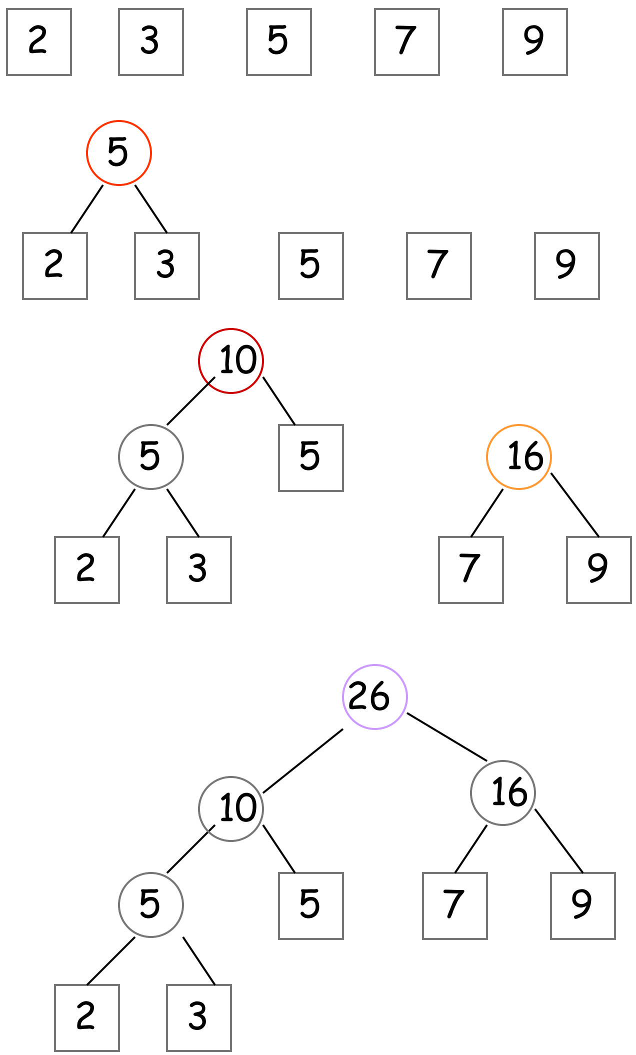
    whose children are a and b

  insert the value $(a + b)$ into the heap

Time complexity $O(n \log n)$

It can be shown that the Binary Merge Tree is optimal

Example of Optimal Merge Tree



Initially, 5 leaf nodes with sizes

Iteration 1: merge 2 and 3 into 5

Iteration 2: merge 5 and 5 into 10

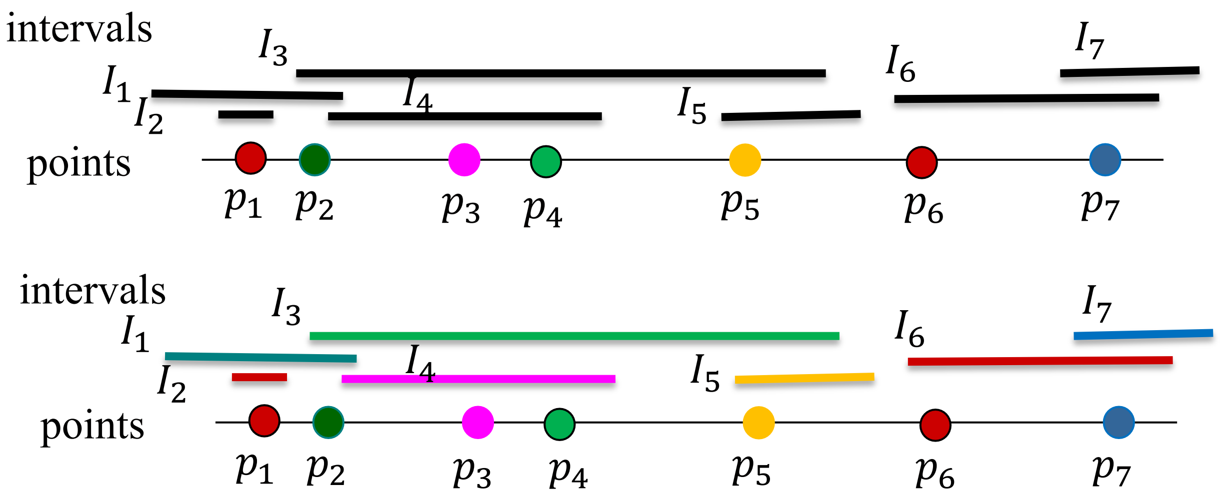
Iteration 3: merge 7 and 9 (chosen among 7, 9, and 10) into 16

Iteration 4: merge 10 and 16 into 26

Cost = 2*3 + 3*3 + 5*2 + 7*2 + 9*2 = 57.

Exercise on Matching Points and Covering Intervals

Given n points x_i $(1 \leq i \leq n)$ on the real line and n intervals $I_j = [s_j, f_j]$, $(1 \leq j \leq n)$, design an algorithm to determine if each point can be assigned to a distinct interval that covers it.



Sort points in non-decreasing order $x_1 \leq \cdots \leq x_i \leq \cdots x_n$
For $i = 1$ to n in the sorted order
  Find interval I_j s.t. $s_j \leq x_i$ and f_j is min among intervals covering x_i
  If such interval exists assign x_i to f_j
    else return false // no assignment possible

$G = \{(p_1, I_2), (p_2, I_1), (p_3, I_4), (p_4, I_3), \ldots\}$

$O = \{(p_1, I_2), (p_2, I_1), (p_3, I_3), \ldots\} \rightarrow O^* = \{(p_1, I_2), (p_2, I_1), (p_3, I_4), \ldots\}$

Exercise on Tiling Path

Let X be a set of n intervals on the real line; each interval x has a starting $x.s$ and a finishing time $x.f$. A subset of intervals $Y \subseteq X$ is called a tiling path if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The size of a tiling cover is just the number of intervals. Design an algorithm to compute the minimal tiling path of X. Assume that all start and finishing times are distinct.

A set of intervals. The seven shaded intervals form a tiling path.

Q: Is the above tiling path minimal?

A: No. The 2^{nd} and 3^{rd} intervals in the path can be replaced by a single one.

Q: In which order you consider the intervals of the tiling path?

A: Increasing order of starting time.

Q: When an interval of the tiling path reaches its finishing time, which interval you would select to succeed it in the path?

A: The interval among those encountered with the largest finishing time.

Exercise on Tiling Path - Algorithm

$\{x_1, x_2, \dots x_n\} \leftarrow$ Sort intervals in increasing starting time.

Insert x_1 to tilling path Y

$last \leftarrow x_1$; $next \leftarrow x_1$;

For $i = 2$ to n

    if $x_i.s < last.f$ then // $last$ covers the beginning of x_i

        if $x_i.f > next.f$, then $next \leftarrow x_i$ // x_i may be next in Tiling Path (i)

    else // *last* does not cover the beginning of x_i

        if $next \neq last$, then insert $next$ to Y

        if $next.f > x_i.s$, then // $next$ covers the beginning of x_i

            $last \leftarrow next$

            if $x_i.f > next.f$, then $next \leftarrow x_i$ (ii)

        else // $next$ does not cover the beginning of x_i

            insert x_i to Y; $last \leftarrow x_i$; $next \leftarrow x_i$ (iii)

if $next \neq last$, then insert $next$ to Y