Chapter 4. (Part 2)

The Greedy Approach

경북대학교 배준현 교수
(joonion@knu.ac.kr)

Contents

4.3 Scheduling Problem

- The ***Scheduling*** Problem:
  - The *time in the system* is
    - the time spent both waiting and being served.
  - The problem of *minimizing* the *total time in the system*
    - has many applications.
    - ex) scheduling users' access to a bank counter or a disk drive.
  - You would learn it in detail
    - when you study the schedulers of operating system.

4.3 Scheduling Problem

- ***Scheduling with Deadlines***:
  - Another scheduling problem
    - occurs when *each job* takes the *same amount of time* to complete,
    - but has a *deadline* by which it must start to *yield a profit*
      - associated with the job.
  - The goal is
    - to schedule the jobs to *maximize* the *total profit*.

4.3 Scheduling Problem

- The Problem of *Scheduling with Deadlines*:
  - Each job *takes one unit* of time to finish
    - and has a *deadline* and a *profit*.
  - If the job starts *before or at* its deadline, the profit is obtained.
    - Therefore, not all jobs have to be scheduled.
  - A schedule is called ***impossible***,
    - if a job is scheduled *after its deadline*.
  - We need not consider any impossible schedule
    - because the job in that schedule does not yield any profit.

4.3 Scheduling Problem

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

1	2	3	4

Job 1	Job 2	(impossible)
Job 1	Job 3	
Job 1	Job 4	(impossible)
Job 2	Job 1	
Job 2	Job 3	
Job 2	Job 4	(impossible)

Schedule	Total Profit
[1, 3]	55
[2, 1]	65
[2, 3]	60
[3, 1]	55
[4, 1]	70 **(optimal)**
[4, 3]	65

$$profit([1,3]) = 30 + 25 = 55$$

$$profit([4,1]) = 40 + 30 = 70$$

4.3 Scheduling Problem

- The Greedy Approach to the Problem:
  - To consider all schedules, a brute-force approach,
    - takes *factorial* time. (worse than exponential)
  - A reasonable greedy approach for solving the problem would be:
    - First, *sort* the jobs in *non-increasing* order *by profit*.
    - Next, *inspect* each job in sequence
      - and *add* it to the *schedule* if it is *possible*.

4.3 Scheduling Problem

- The Greedy Approach to the Problem:
  - A *sequence* is called a ***feasible sequence***
    - if all the jobs in the sequence *start by their deadlines*.
    - ex) [4, 1]: feasible sequence, [1, 4]: not a feasible sequence.
  - A *set of jobs* is called a ***feasible set***
    - if there exists *at least one feasible sequence* for the jobs in the set.
    - ex) {1, 4}: feasible set, {2, 4} not a feasible set.
  - Our goal is to find an *optimal* sequence,
    - which is *a feasible sequence with maximum total profit*.
  - An *optimal set of jobs* is the set of jobs in an optimal sequence.

4.3 Scheduling Problem

- High-level greedy algorithm for the problem:

```
sort the jobs in nonincreasing order by profit;
S = ∅;
while (the instance is not solved) {
   select next job;
   if (S is feasible with this job added)
       add this job to S;
   if (there are no more jobs)
       the instance is solved;
}
```

4.3 Scheduling Problem

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

1. $S = \phi$
2. $S = \{1\}$, [1] is feasible
3. $S = \{1, 2\}$, [2, 1] is feasible
4. $S = \{1, 2, 3\}$, rejected
   there is no feasible sequence for this set : $S = \{1, 2\}$
5. $S = \{1, 2, 4\}$, [2, 1, 4] is feasible
6. $S = \{1, 2, 4, 5\}$, rejected
   $S = \{1, 2, 4\}$
7. $S = \{1, 2, 4, 6\}$, rejected
   $S = \{1, 2, 4\}$
8. $S = \{1, 2, 4, 7\}$, rejected
   $S = \{1, 2, 4\}$ (feasible set) [2, 1, 4] (feasible sequence)

4.3 Scheduling Problem

- An efficient way to *determine* whether a set is *feasible*:
  - Lemma:
    - Let S be *a set of jobs*, then S is *feasible*
      - *if and only if* the sequence obtained by ordering
      - the jobs in S according to *nondecreasing deadlines*
      - is *feasible*.

We need only check the feasibility of the sequence:

$S = \{1,\ \ 2,\ \ 4,\ \ 7\}$: feasible?

not feasible

[2, 7, 1, 4]

1 2 3 3

not feasible

4.3 Scheduling Problem

ALGORITHM 4.4: Scheduling with Deadlines

```
void schedule(int n, int dealine[], sequence_of_integer &J) {
    int i;
    sequence_of_integer K;

    J = [1];
    for (i = 2; i <= n; i++) {
        K = J with i added according to nondecreasing values of deadline[i];
        if (K is feasible)
            J = K;
    }
}
```

4.3 Scheduling Problem

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

The jobs are already sorted by the profit

1. $J = [1]$
2. $K = [2,1]$, K is feasible
   $J = [2,1]$
3. $K = [2,3,1]$ is rejected, because K is not feasible
4. $K = [2,1,4]$, K is feasible
   $J = [2,1,4]$
5. $K = [2,5,1,4]$ is rejected
6. $K = [2,1,4,6]$ is rejected
7. $K = [2,7,1,4]$ is rejected

- $J = [2, 1, 4]$ is the final result
- $Total\ Profit = 35 + 40 + 25 = 100$

4.3 Scheduling Problem


```cpp
typedef vector<int> sequence_of_integer;

bool is_feasible(sequence_of_integer& K, sequence_of_integer& deadline) {
    for (int i = 1; i < K.size(); i++)
        if (i > deadline[K[i]])
            return false;
    return true;
}
```

4.3 Scheduling Problem

```
void schedule(int n, sequence_of_integer& deadline, sequence_of_integer &J) {
    sequence_of_integer K;
    J.clear();
    J.push_back(0); // for an empty job
    J.push_back(1);
    for (int i = 2; i <= n; i++) {
        // K = J with i added according to nondecreasing values of deadline[i];
        K.resize(J.size());
        copy(J.begin(), J.end(), K.begin());
        int j = 1;
        while (j < K.size() && deadline[K[j]] <= deadline[i])
            j++;
        K.insert(K.begin() + j, i);
        if (is_feasible(K, deadline)) {
            // J = K
            J.resize(K.size());
            copy(K.begin(), K.end(), J.begin());
        }
    }
}
```

4.3 Scheduling Problem

- **Time Complexity of Algorithm 4.4 (Worst-Case)**
  - Basic Operation: the operation of comparisons to *sort*, to do $K = J$, and to *check* if K is *feasible*.
  - Input Size: n, the $number\ of\ jobs$.
  - In each iteration of the $\texttt{for}-i$ loop, we need to do
    - at most $i - 1$ comparisons to add the ith job of K,
    - and at most i comparisons to *check* if K is *feasible*.
  - Therefore, the worst case is

$$W(n) = \sum_{i=2}^{n} [(i - 1) + i] = n^2 - 1 \in \Theta(n^2)$$

4.4 Huffman Code

- The problem of *data compression*
  - is to find an *efficient method* for *encoding a data file*.
  - A *binary code* is a common way to represent a data file.
  - A *codeword* is a *unique binary string*
    - representing *each character* in a binary code.
  - A ***fixed-length*** binary code
    - represents each character using the *same number of bits*.
  - A ***variable-length*** binary code is a more efficient coding.
    - It represents different characters using *different number of bits*.

4.4 Huffman Code

- The Problem of the ***Optimal Binary Code***:
  - Given a file (or a string of characters),
    - find a *binary character code* for the characters in the file,
    - which represents the file in the *least number of bits*.
  - We discuss the encoding method, called *Huffman code*,
    - then we develop *Huffman's algorithm* for solving this problem.

4.4 Huffman Code

$$File = ababcbbbc, character\ set = \{a, b, c\}$$

Character	Fixed-lengh Binary Code
a	00
b	01
c	10

Character	Variable-length Binary Code
a	10
b	0
c	11

a	b	a	b	c	b	b	b	c
00	01	00	01	10	01	01	01	10

- It takes 18 bits with this encoding

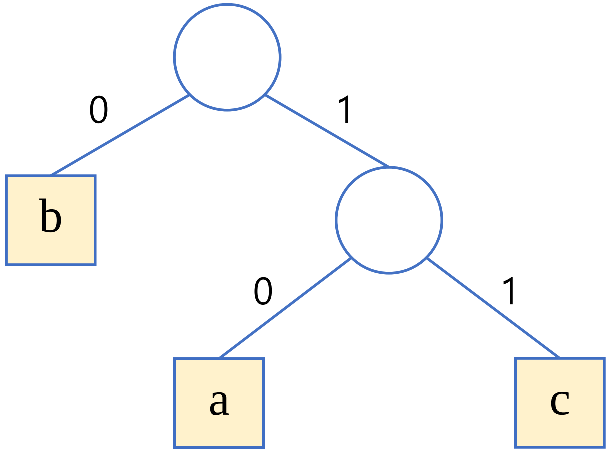
a	b	a	b	c	b	b	b	c
10	0	10	0	11	0	0	0	11

- 'b' occurs *most frequently*:
  - Encode 'b' with one bit (0)
- Encode 'a' and 'c' starting with 1 to distinguish from ' bb'
  - 'a': 10, 'c': 11
- It takes only 13 bits with this encoding

4.4 Huffman Code

- Prefix Code
  - is one particular type of variable-length code.
  - In a prefix code, *no codeword* for one character
    - constitutes the *beginning of the codeword* for another character.
  - Every prefix code can be represented by
    - a *binary tree* whose *leaves* are *the characters* that are to be encoded.
  - The advantage of a prefix code is that
    - we *need not look ahead* when parsing the file.

- b: 0
- a: 10
- c: 11



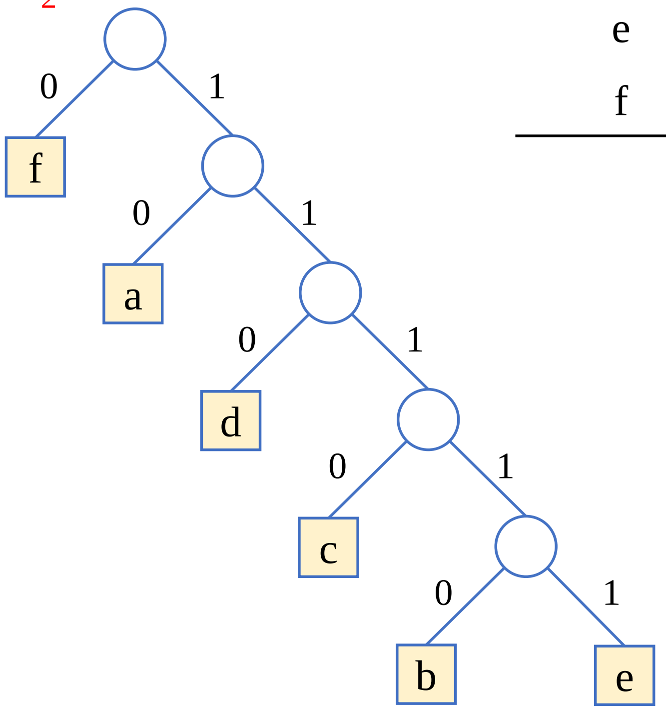
4.4 Huffman Code

$S = \{a, b, c, d, e, f\}$

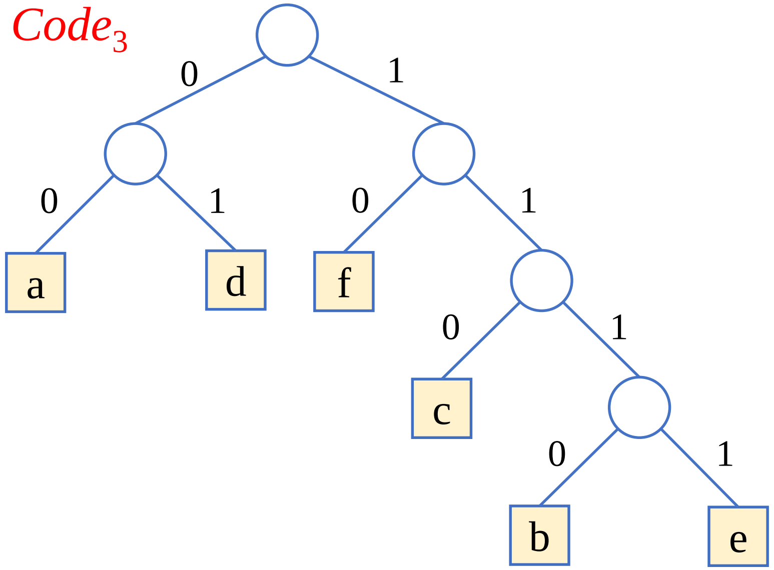
Character	Frequency	$Code_1$	$Code_2$	$Code_3$
a	**16**	000	10	00
b	**5**	001	11110	1110
c	**12**	010	1110	110
d	**17**	011	110	01
e	**10**	100	11111	1111
f	**25**	101	0	10

- $Code_1$: Fixed-Length
- $Code_3$: Huffman code

$Code_2$



$Code_3$



4.4 Huffman Code

- Computing the *number of bits* for *encoding*:
  - Given the binary tree T corresponding to some code
    - the number of bits it takes to encode a file is given by
    - $bits(T) = \sum_{i=1}^{n} frequency(v_i) \times depth(v_i)$
      - where $\{v_1, v_2, \cdots, v_n\}$ is the set of characters in the file,
      - $frequency(v_i)$ is the number of times v_i occurs in the file,
      - and $depth(v_i)$ is the depth of v_i in T.

- $bits(Code_1) = 255$
- $bits(Code_2) = 231$
- $bits(Code_3) = 212$

- Huffman's Algorithm
  - *Huffman* developed a *greedy* algorithm
    - that produces an *optimal binary character code* by constructing
    - a *Huffman code*, a *binary tree* corresponding to an *optimal code*.

  - We need a *type declaration* for the node of binary tree.
  - We also need to use a *priority queue*
    - in which the character with the *lowest frequency* is *removed next*.
    - It can be implemented as a *min-heap*.

4.4 Huffman Code

- High-level pseudo-code for the Huffman's algorithm:

```
n = number of characters in the file;
Arrange n pointers to nodetype records in a PQ;

for (i = 1; i <= n - 1; i++) {
     remove(PQ, p);
     remove(PQ, q);
     r = new nodetype;
     r->left = p;
     r->right = q;
     r->frequency = p->frequency + q->frequency;
     insert(PQ, r);
}
remove(PQ, r);
return r;
```

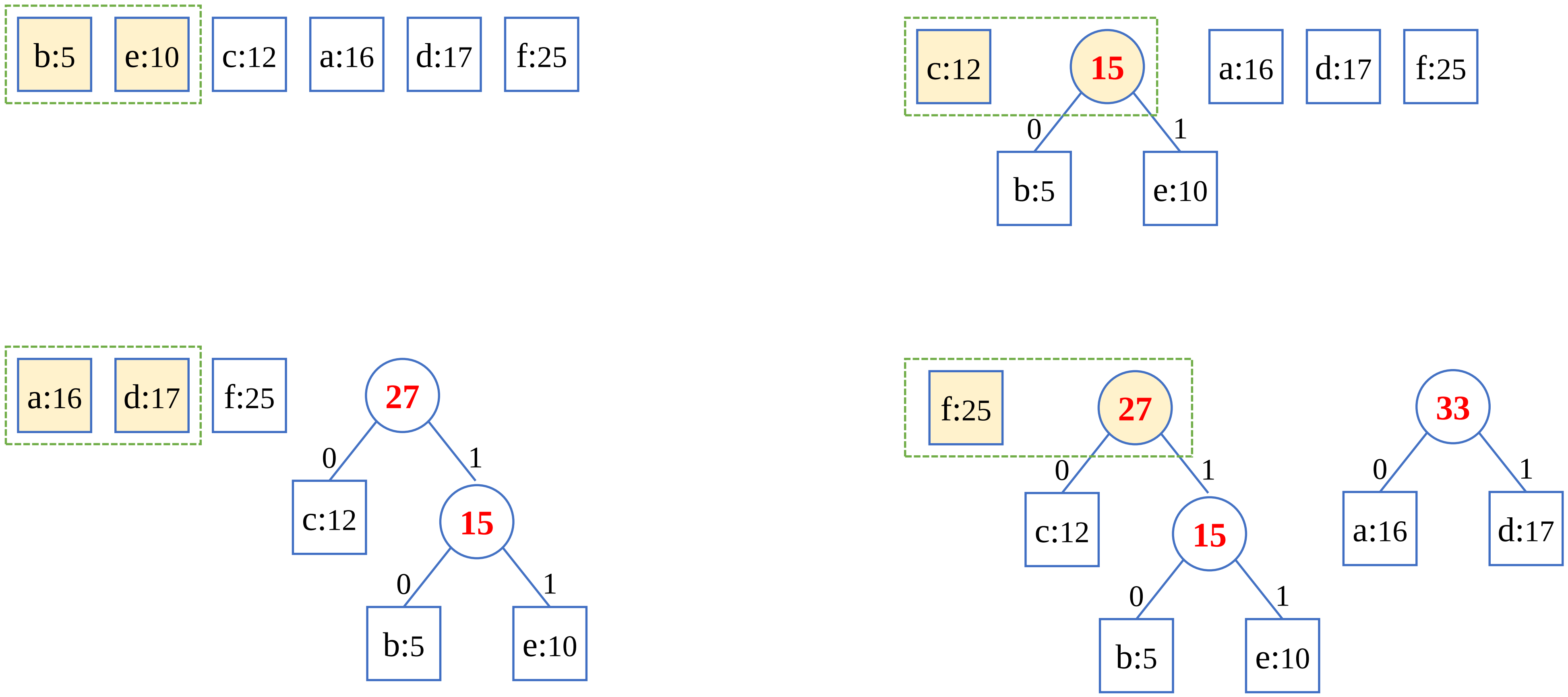
4.4 Huffman Code

```
typedef struct node *node_ptr;
typedef struct node {
    char symbol;    // the value of a character.
    int frequency;  // the number of times the character is in the file.
    node_ptr left;
    node_ptr right;
} node_t;


struct compare {
    bool operator()(node_ptr p, node_ptr q) {
        return p->frequency > q->frequency;
    }
};

typedef priority_queue<node_ptr, vector<node_ptr>, compare> PriorityQueue;
```

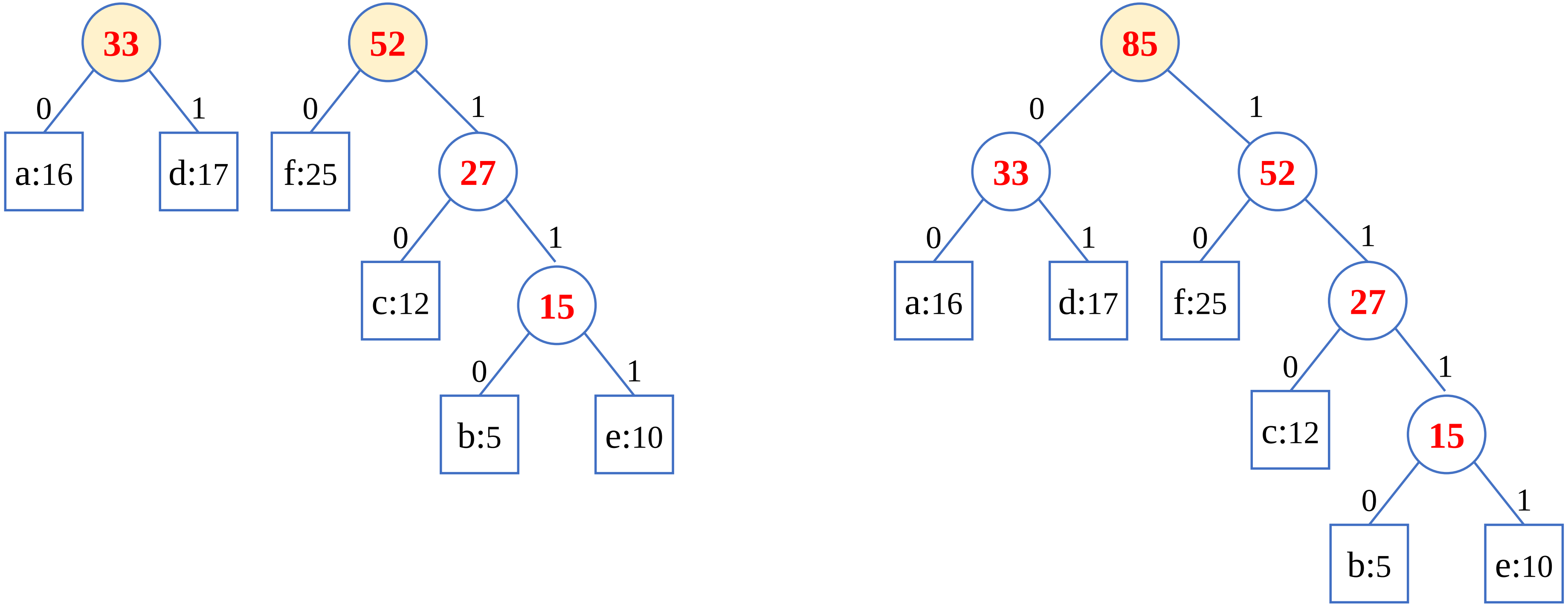
4.4 Huffman Code


```
void huffman(int n, PriorityQueue& PQ)
{
    for (int i = 1; i <= n - 1; i++) {
        node_ptr p = PQ.top(); PQ.pop();
        node_ptr q = PQ.top(); PQ.pop();
        node_ptr r = create_node('+', p->frequency + q->frequency);
        r->left = p;
        r->right = q;
        PQ.push(r);
    }
}
```

4.4 Huffman Code



4.4 Huffman Code



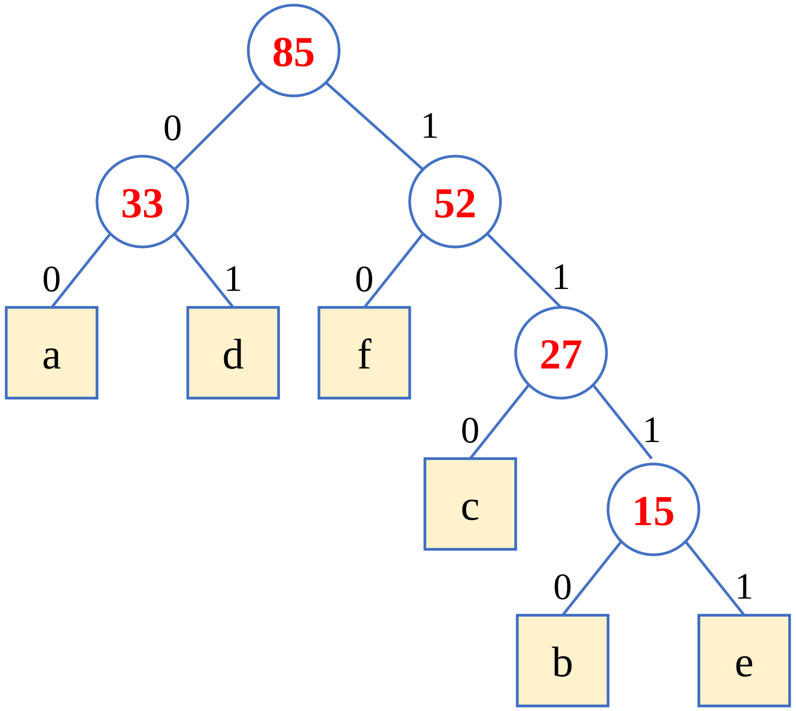
4.4 Huffman Code

- Time Complexity of Huffman's Algorithm
  - If a priority queue is implemented *as a min-heap*,
    - each heap operation (insert & remove) requires $\Theta(\lg n)$ time.
  - Since there are $n - 1$ passes through the `for`-i loop,
    - the algorithm runs in $\Theta(n \lg n)$ time.

- It is *provable* that
  - Huffman's algorithm always produces an optimal binary code.
  - based on Lemma:
    - The binary tree corresponding to an *optimal binary prefix code* is *full*.
    - That is, *every nonleaf node has two children*.

4.4 Huffman Code

- How to *encode* a string of characters?
  - Make a *hash map* of characters and binary codes, then,
  - Given with a string of characters, *afdbce*,
    - you can *transform* it directly into 0010011110110111 1.



a → 00
d → 01
f → 10
c → 110
b → 1110
e → 1111

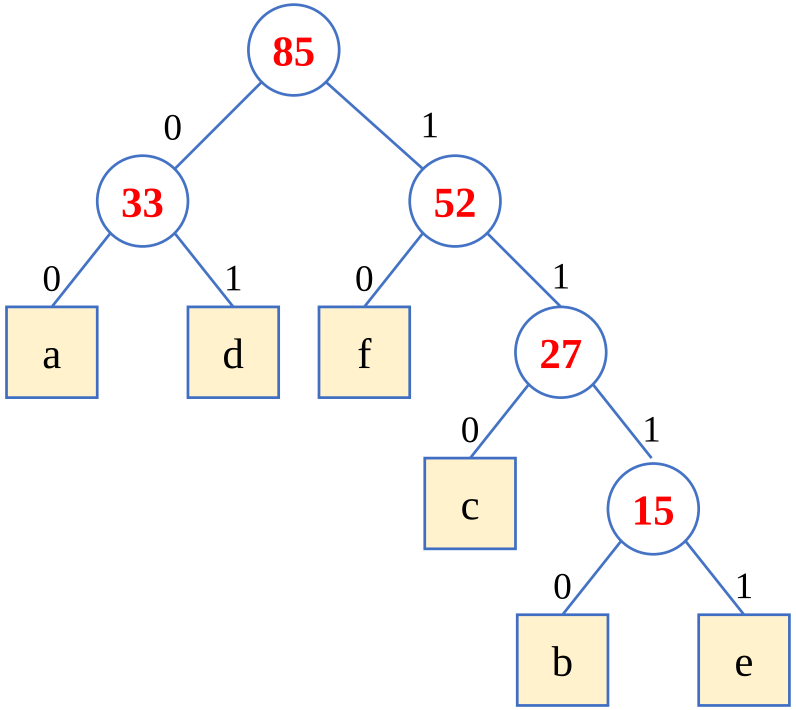
a	f	d	b	c	e
↓	↓	↓	↓	↓	↓
00	10	01	1110	110	1111

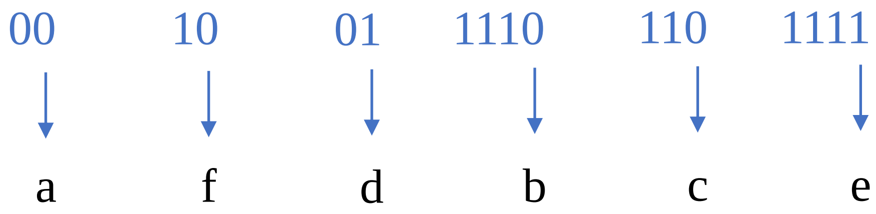
4.4 Huffman Code

```
void make_encoder(node_ptr node, string code, map<char, string> &encoder) {
    if (node->symbol != '+') { // leaf node
        encoder[node->symbol] = code;
    } else { // internal node
        make_encoder(node->left, code + "0", encoder);
        make_encoder(node->right, code + "1", encoder);
    }
}
```

4.4 Huffman Code

- How to *decode* an encoded binary string?
  - Given with an encoded binary string, 0010011110110111 1,
    - you can *traverse* the binary tree to decode it into $afdbce$.





4.4 Huffman Code

```
void decode(node_ptr root, node_ptr node, string s, int i) {
    if (i <= s.length()) {
        if (node->symbol != '+') { // leaf node
            cout << node->symbol;
            decode(root, root, s, i);
        } else { // internal node
            if (s[i] == '0')
                decode(root, node->left, s, i + 1);
            else // s[i] == '1'
                decode(root, node->right, s, i + 1);
        }
    }
}
```

Any Questions?