Greedy Algorithms

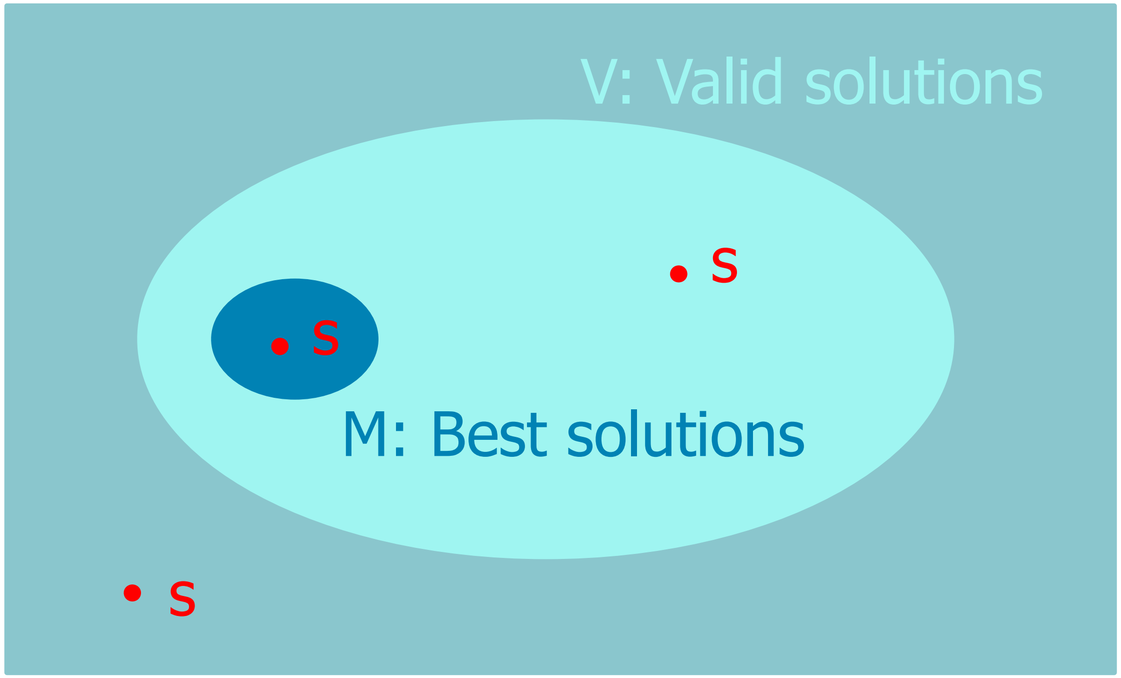
Greedy Algorithms

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Optimization Algorithms

- Algorithms for optimization problems typically look for optimal solutions and
  - Go through a sequence of steps
  - Make a set of choices at each step to reach the desired target

S: All Solutions



Greedy Algorithms

- Many optimization problems
  - Have very expensive solutions adopting brute-force recursion or dynamic programming
  - Sometimes, they may be easily solved with simpler and more efficient algorithms
    - Instead of making exhaustive choices, it is possible to make the choice that looks best at the moment hoping that locally optimal choices will lead to a globally optimal solution
    - This is the strategy followed by **greedy algorithms**
    - Greedy algorithms may be quite powerful and it may work well for a wide range of problems

Greedy Algorithms

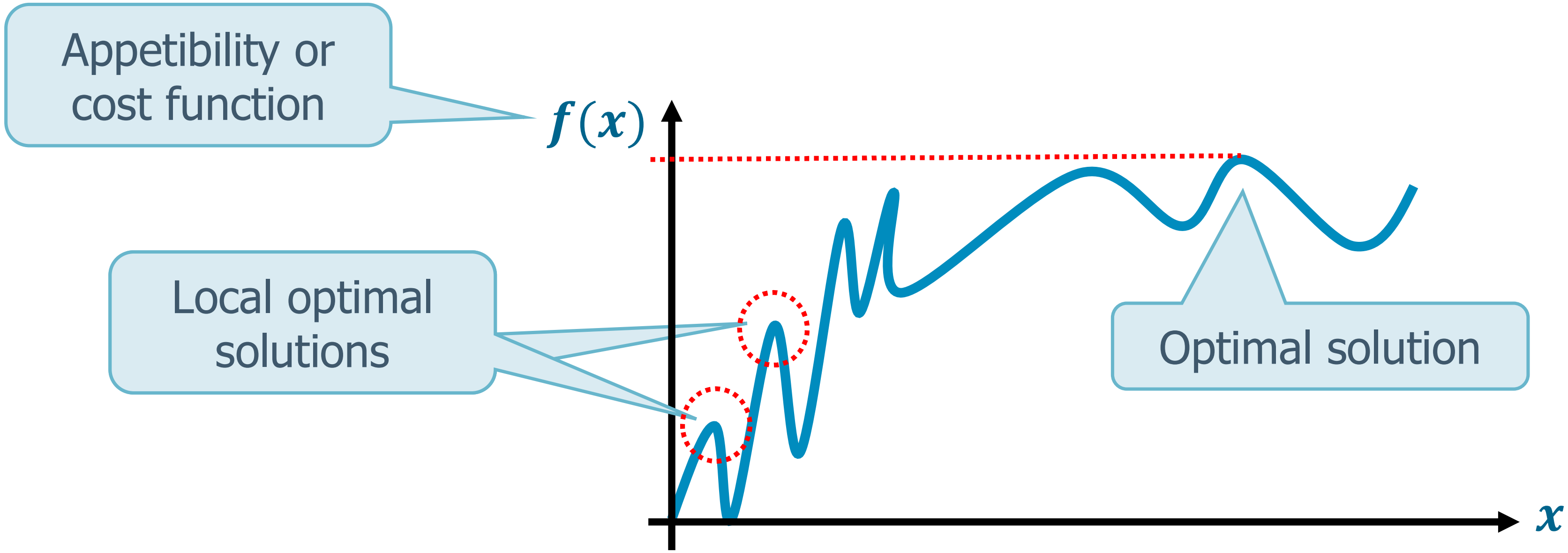
- Greedy algorithms
  - Seek globally optimal solutions by making locally optimal choices
    - Decisions are considered locally optimal based on an appetibility (or cost) function
  - Never reconsider previously taken decisions
    - They never perform backtracking
  - For the above reason greedy algorithms
    - Are very simple and efficient
    - Have limited process time

Greedy Algorithms

- The cost function may be
  - Selected a priori and never changed thereafter
    - We start from the empty solution
    - We sort choices according to the cost function
    - We make choices in descending appetibility order, adding, if possible, the result to the partial solution
  - Modifyiable during the process
    - The process proceeds as before, but choices are stored in a priority queue
    - The appetibility value (cost function) represents the priority used to select the choices and it varies from step to step

Greedy Algorithms

- The solution is not always optimal, but for many problems it is
  - Optimal solution
    - Best possible solution
  - Locally optimal solution
    - Best possible solution within a contiguous domain

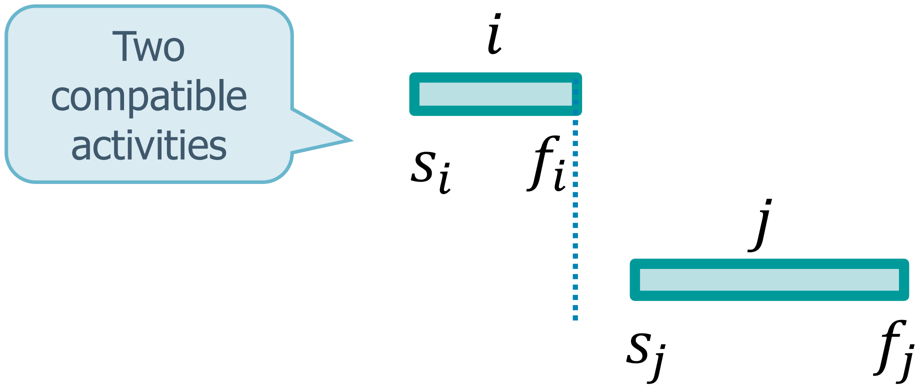


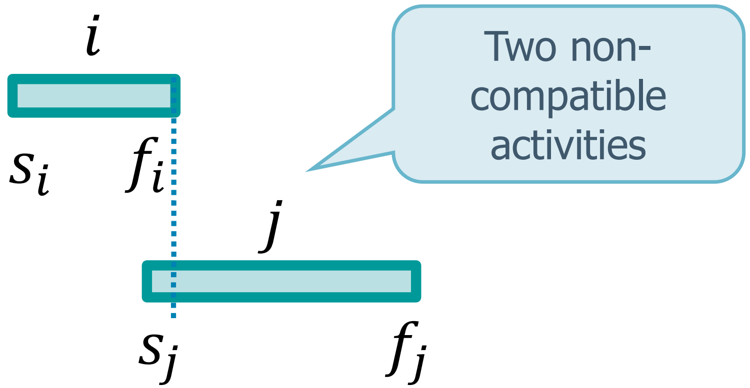
Algorithms

- In this unit, we analyze two algorithms is which greedy choices lead to optimal solutions
  - The Activity Selection problem
  - The Huffman code to perform data compression
- In the graph units we are going to see a few other greedy algorithms
  - Prim’s and Kruskal’s Minimum-Spanning-Tree
  - Dijkstra’s Single-Source-Shortes-Path

Activity Selection Problem

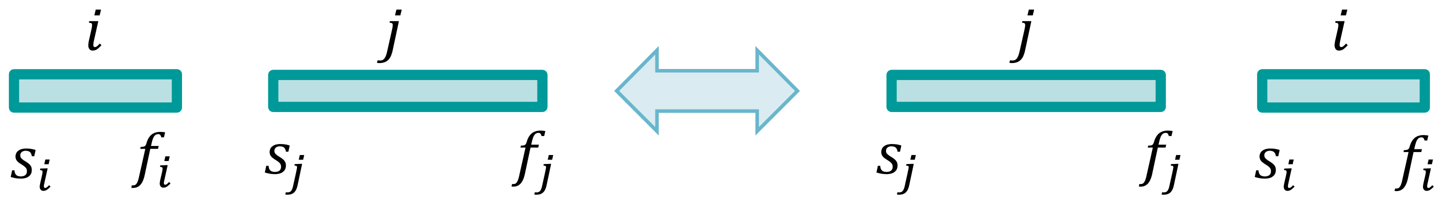
- In the activity selection problem, we
  - Have to schedule several competing activities that requires exclusive usage of a common resource
    - Each activity has a start and an finish time
  - The goal is to select a maximum-size subset of mutually compatible activities
    - Two activities are compatible if they do not overlap in time as they must use the **same** resource (e.g., a classroom, a specific device, etc.)





Activity Selection Problem

- To build an optimal solution, we define
  - Input
    - Set of n activities with start time (s) and finish (f) time $[s, f)$ (or $[s, f[$)
  - Output
    - Sub-set with the maximum number of compatible activities $[s_i, f_i)$
  - Constraints
    - No selected activity $[s_i, f_i)$ overlaps with another one $[s_j, f_j)$, that is, $s_j \geq f_i$ or $s_i \geq f_j$



Activity Selection Problem

- Greedy iterative approach
  - Set A is the initial set of activities
  - Set S is the final sub-set of activities
  - Sort the activities in A by increasing finish time
  - Initialize S to the empty set $\emptyset$
  - While the set of activity A is not empty
    - Select the earliest activity s in A to finish
    - If s is compatible with every activity in S, add s to S
  - Return the set A

Example

Initial activity
(sorted by increasing finish time)

Selected activity

k	s_k	f_k
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16

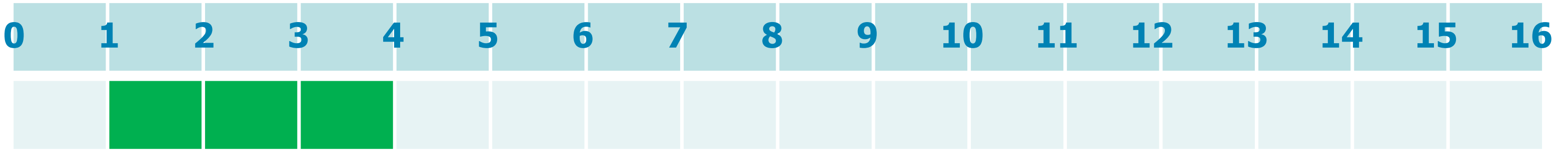
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Solution

Initial activity
(sorted by increasing finish time)

Selected activity
1

k	s_k	f_k
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16

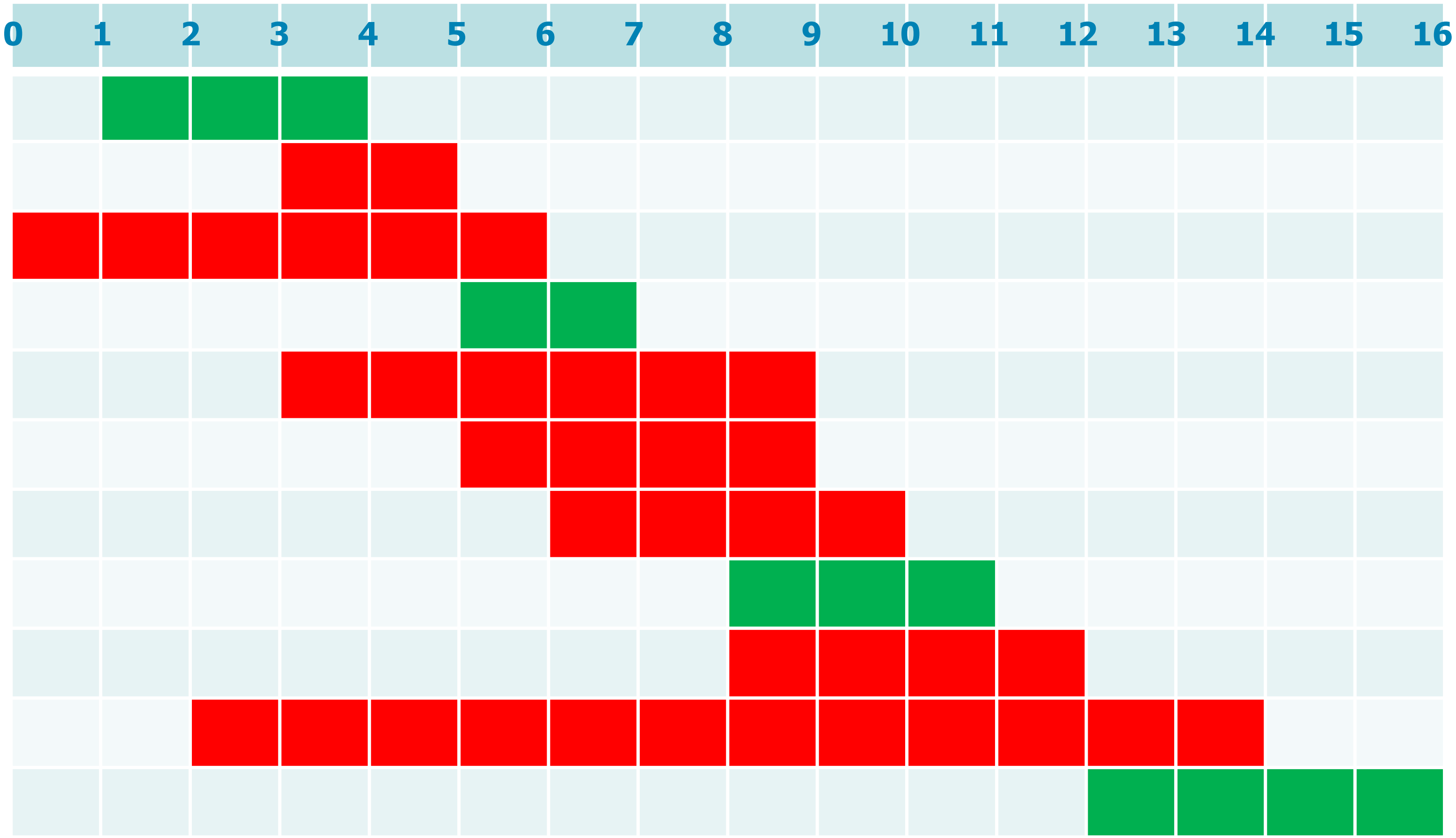


Solution

Initial activity
(sorted by increasing finish time)

Selected activity
1, 4, 8, 11

k	s_k	f_k
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16



Implementation

The iterative C implementation

Data-base definition

```
typedef struct activity {
  char name[MAX];
  int start, stop;
  int selected;
} activity_t;
...
```

Compare Function

```
int cmp (const void *p1, const void *p2);

...

acts = load(argv[1], &n);
qsort ((void *)acts, n, sizeof(activity_t), cmp);
choose (acts, n);
display (acts, n);
...
```

load activity array

asc sort by stop

C Standard Library

Implementation

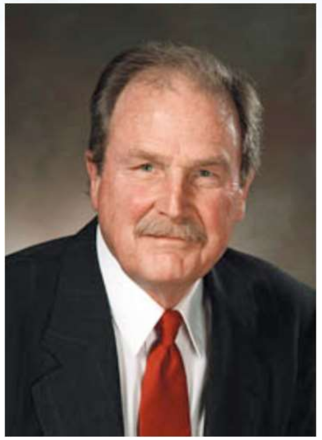
```
int cmp (const void *p1, const void *p2) {
  activity_t *a1 = (activity_t *)p1;
  activity_t *a2 = (activity_t *)p2;
  return a1->stop - a2->stop;
}
void choose (activity_t *acts, int n) {
  int i, stop;

  acts[0].selected = 1;
  stop = acts[0].stop;
  for (i=1; i<n; i++) {
    if (acts[i].start >= stop) {
      acts[i].selected = 1;
      stop = acts[i].stop;
    }
  }
}
```

we check from the beginning since we already sorted

prototype quicksort: qsort(void *, int, int, int *f(void *a. void *b))

Huffman Codes

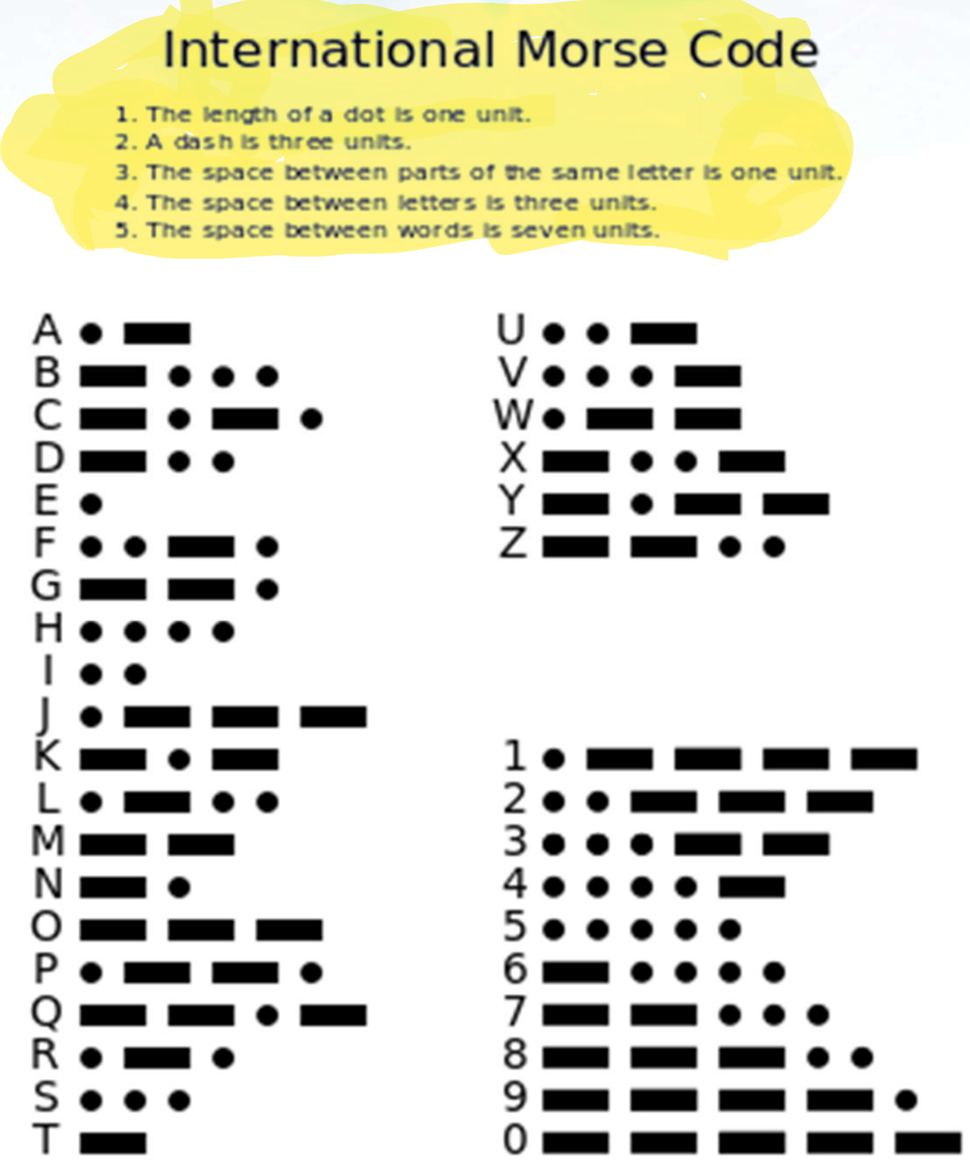


- Huffman in 1950 invented a greedy algorithm that construct an optimal prefix code give a set of symbols S
- Codeword
  - String of bits associated to a symbol $s \in S$
- Encoding
  - From symbol to codeword
- Decoding
  - From codeword to symbol

Huffman Codes

- It is possible to have fixed-length and variable-length codes
  - Fixed-length codes
    - Codewords with $n = \lceil log_2(|S|) \rceil$ bits
    - Pro: easy to decode
    - Use: symbol occurring with the same frequency
  - Variable-length codes
    - Con: difficult to decode
    - Pro: memory savings
    - Use: symbols occurring with different frequencies
    - Example
      - Morse alphabet (with pauses between words)

The Morse Code



Example

- Give a file with only 6 different characters but storing 100.000 characters overall

3 bits → 8 different values > 6

- We can encode the file using
  - A fixed-length code with 3 bits per code, storing

$$3 \cdot 100000 = 300000\ bits$$

  - A variable-length code with 1-4 bits per code, storing

$$(0.45 \cdot 1 + 0.13 \cdot 3 + 0.12 \cdot 3 + 0.16 \cdot 3 + 0.09 \cdot 4 + 0.05 \cdot 4) \cdot 1000 = \\ = 224000\ bits$$

000

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

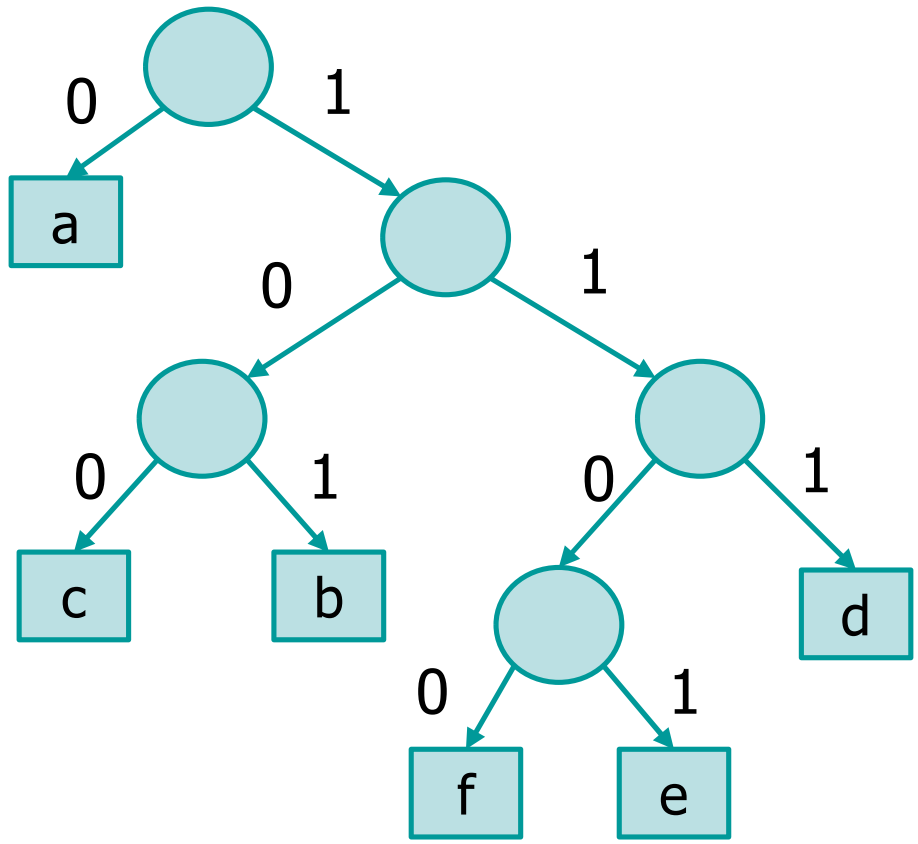
Prefix-codes

- We consider codes in which no codeword is also a prefix of another valid codeword
- These codes are called prefix-codes
  - Prefix-codes can always achieve the optimal data compression among any character code
  - We suffer no loss of generality
- For prefix-codes
  - Encoding
    - Juxtapposition of strings
  - Decoding
    - Path on a binary tree

Example

- The correspondence symbols-codes can be stored in a tree

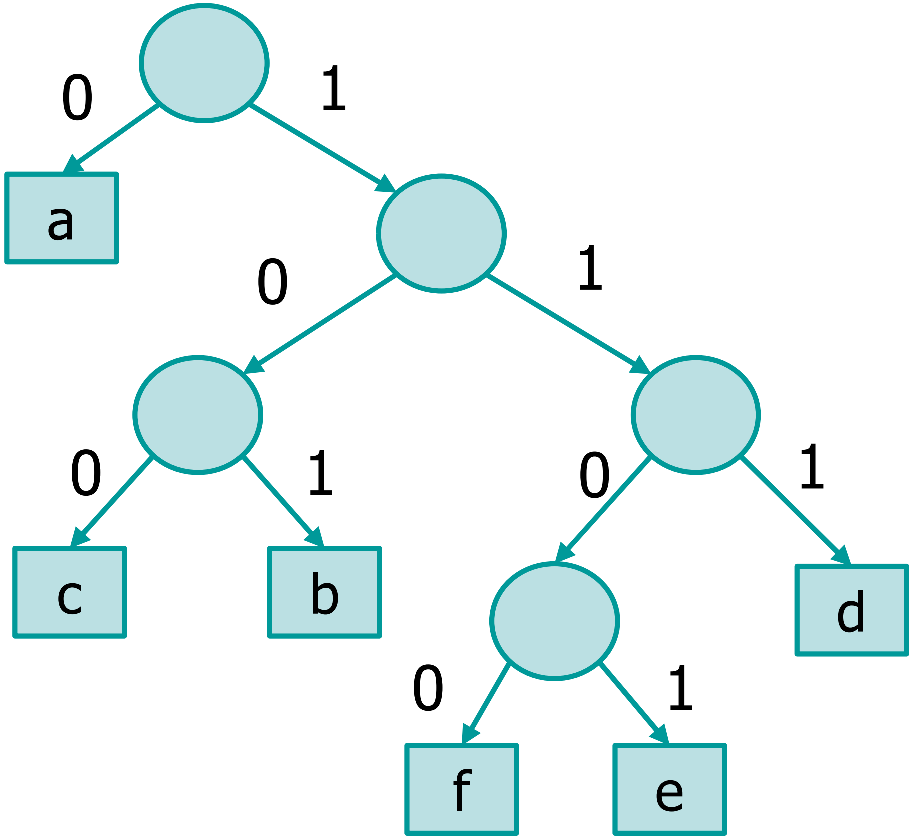
$$
\begin{aligned}
a &= 0 \\
b &= 101 \\
c &= 100 \\
d &= 111 \\
e &= 1101 \\
f &= 1100
\end{aligned}
$$



Example: Encoding

- Encoding
  - Is the evaluation of the codes starting from the symbols

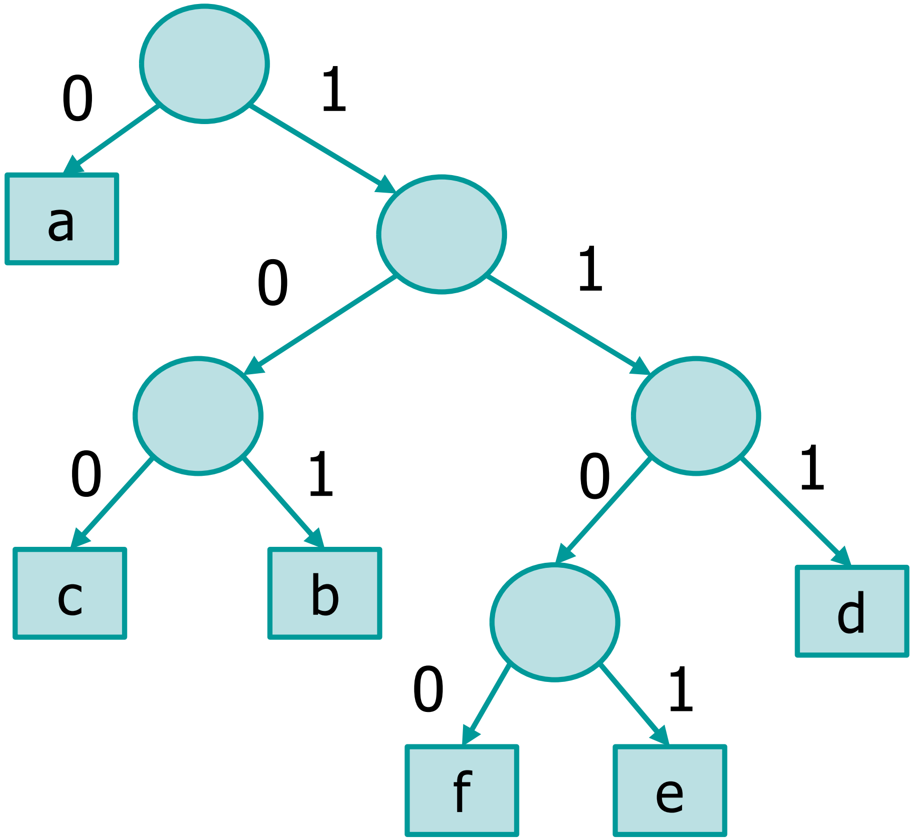
$$abfaac \rightarrow 0101110000100$$



Example: Decoding

- Decoding
  - Is the evaluation of the symbols starting from the codes

$$0101110000100 \rightarrow abfaac$$



Building the tree

- To implement the previous algorithm, we must create the correspondence tree
- Data structure
  - Priority queue used to store tree nodes with their frequency
  - Each tree node represent an initial code or an aggregate
  - The frequency of the code or the aggregate is the priority

Building the tree

- Algorithm
  - Initially
    - Each symbol is a tree leaf
  - Intermediate step
    - Extract the 2 symbols (or aggregates) with minimum frequency
    - Build the binary tree (aggregate of symbols)
      - Each node is a symbol or aggregate
      - Its frequency is the sum of the frequencies
    - Insert the new aggregate into priority queue
  - Termination
    - Empty queue

Example

- Given the following frequencies

letter: relative frequency

$f: 5 \quad e: 9 \quad c: 12 \quad b: 13 \quad d: 16 \quad a: 45$

  - Find an optimal Huffman code for all symbols in the set using a greedy algorithm
  - Indicate the code that must be used to represent all symbols in the set
- Note
  - In all following steps, we store symbols and aggregates in a **sorted array**
  - Anyway, a sorted array is **far less efficient** than a priority queue in practice

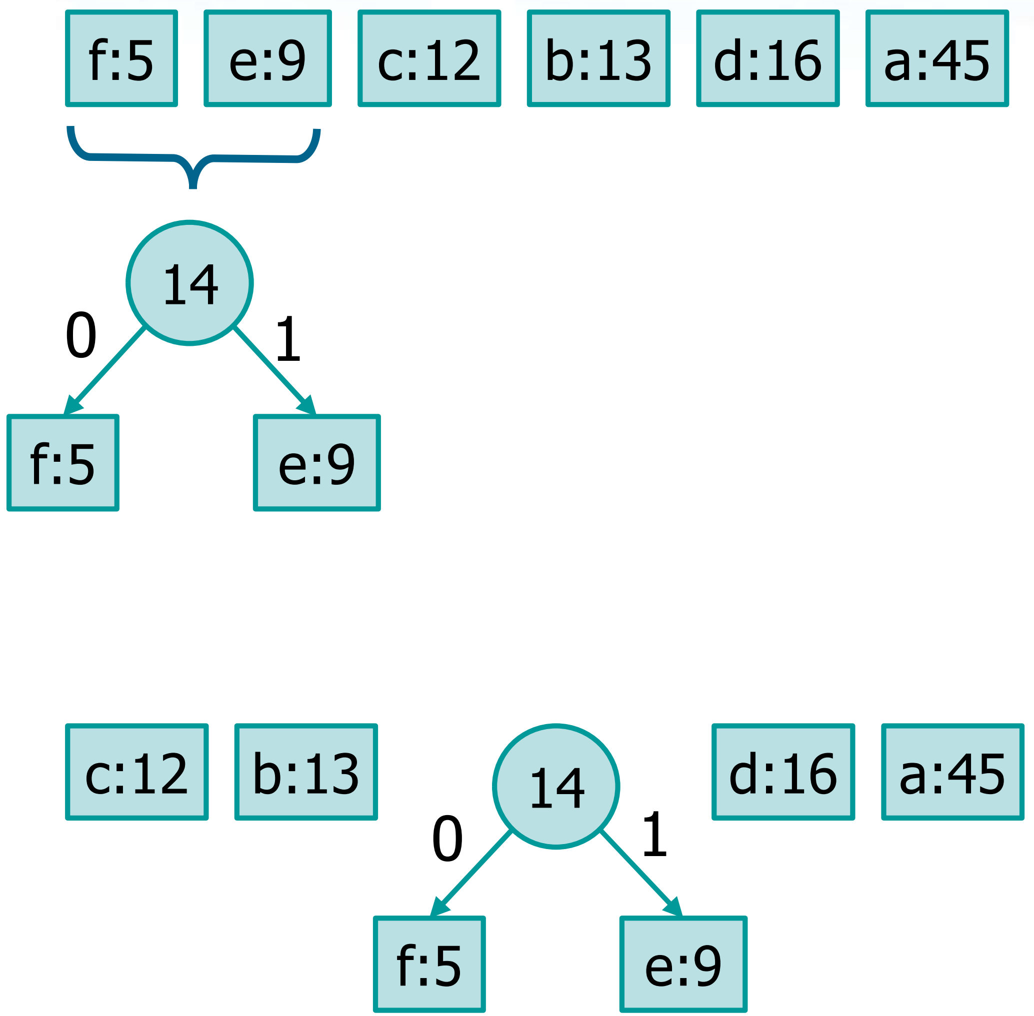
Solution: Step 1

Priory Queue (fully sorted)

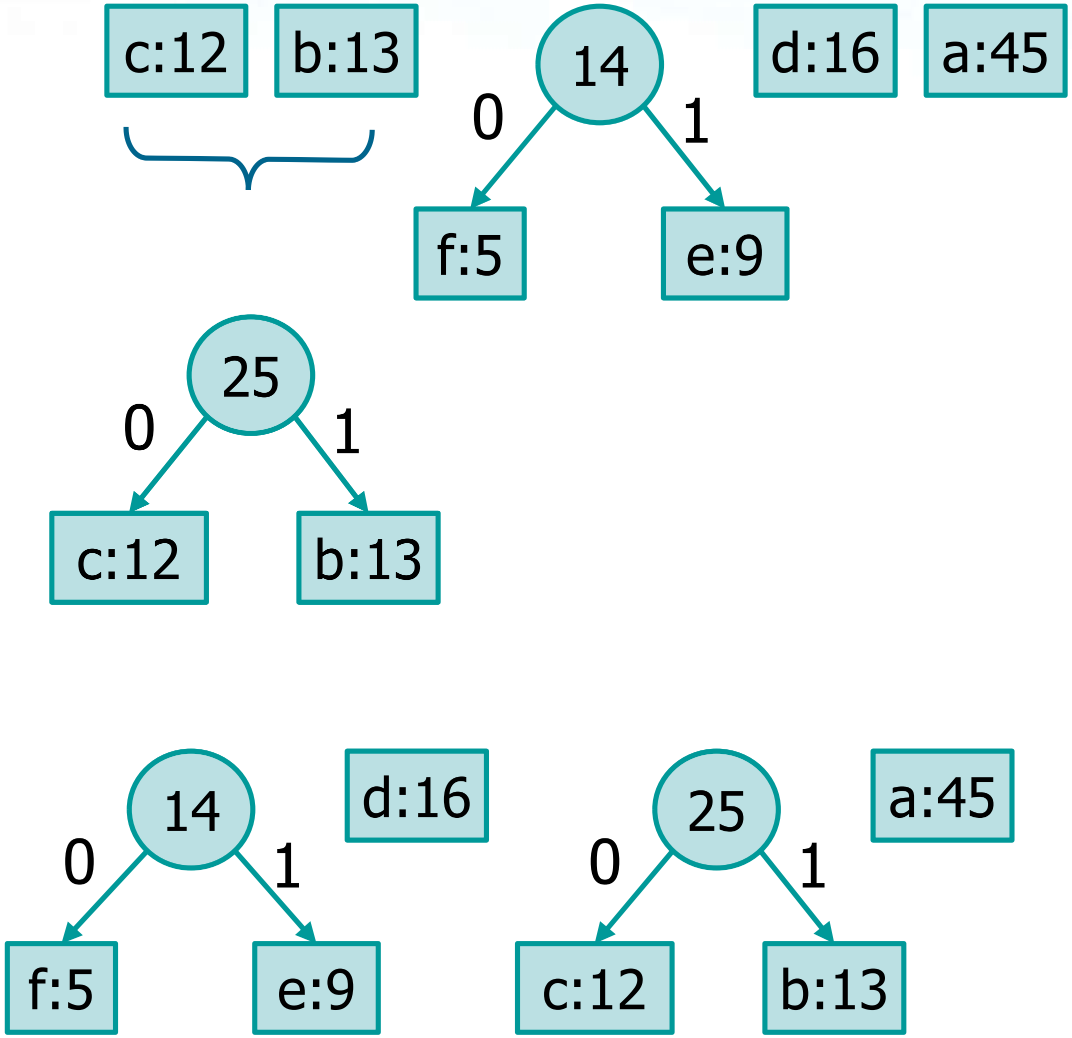
Extract

Build the tree of the aggregate

Insert the aggregate back into the priority queue



Solution: Step 2

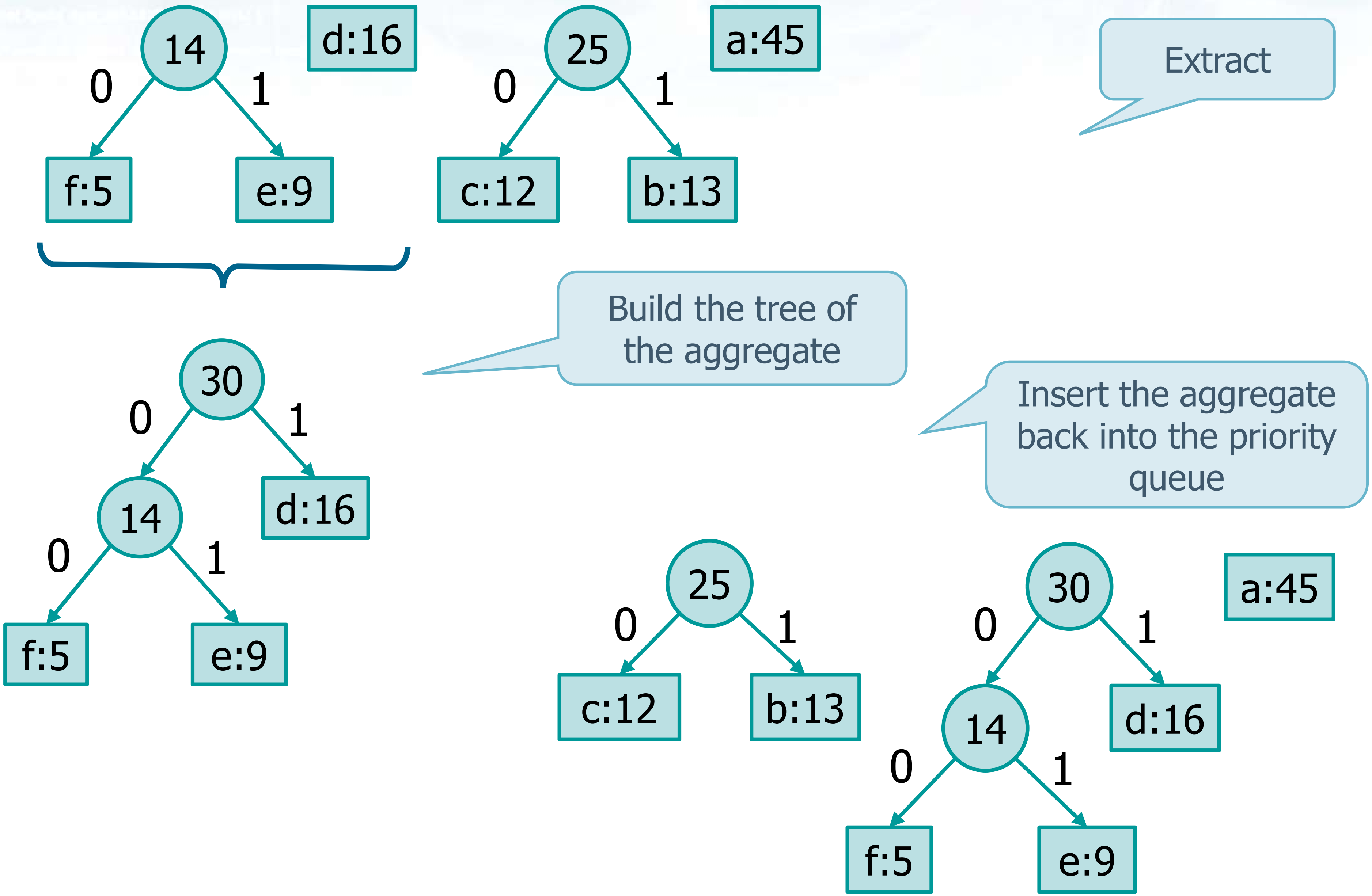


Extract

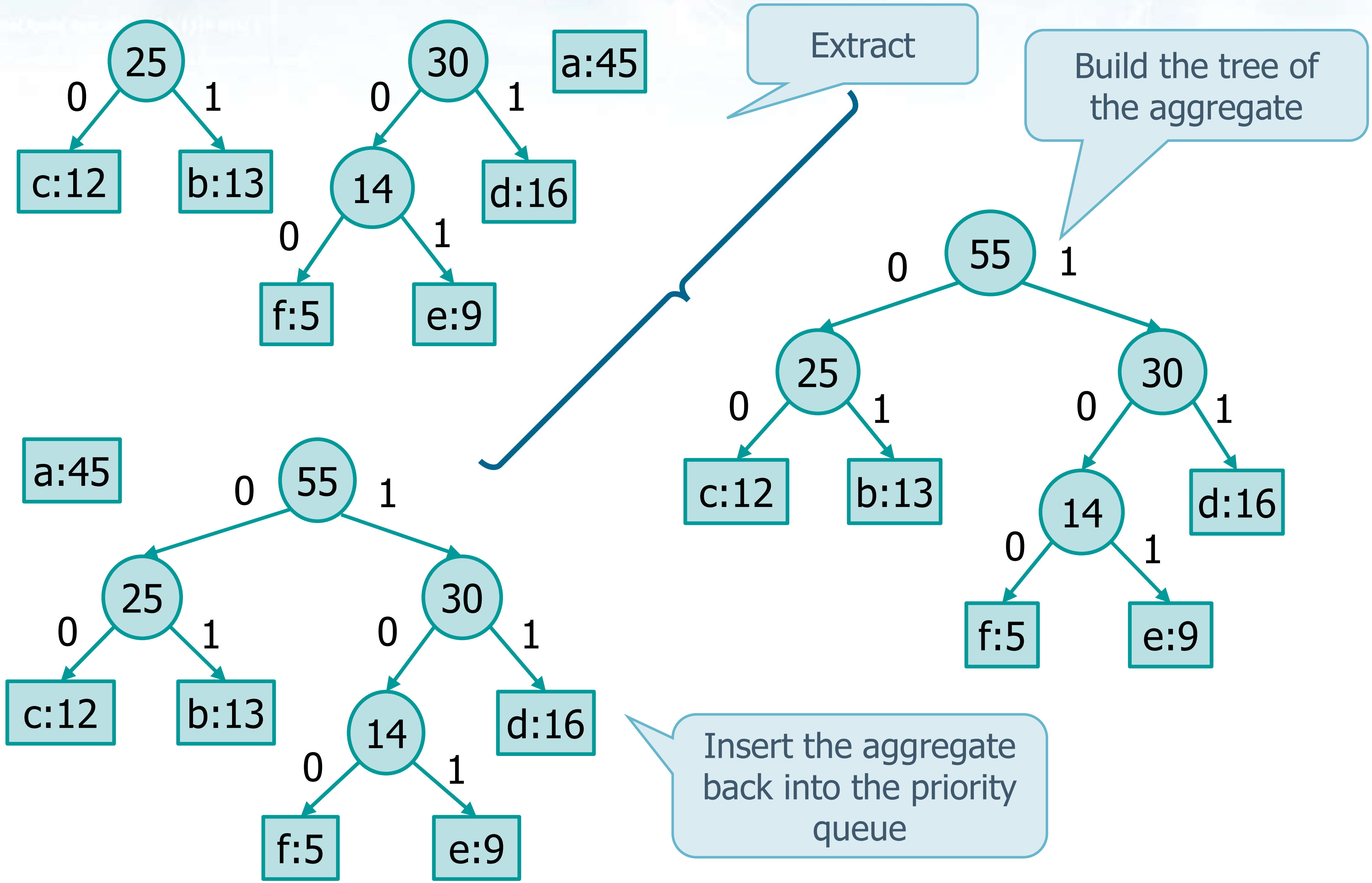
Build the tree of the aggregate

Insert the aggregate back into the priority queue

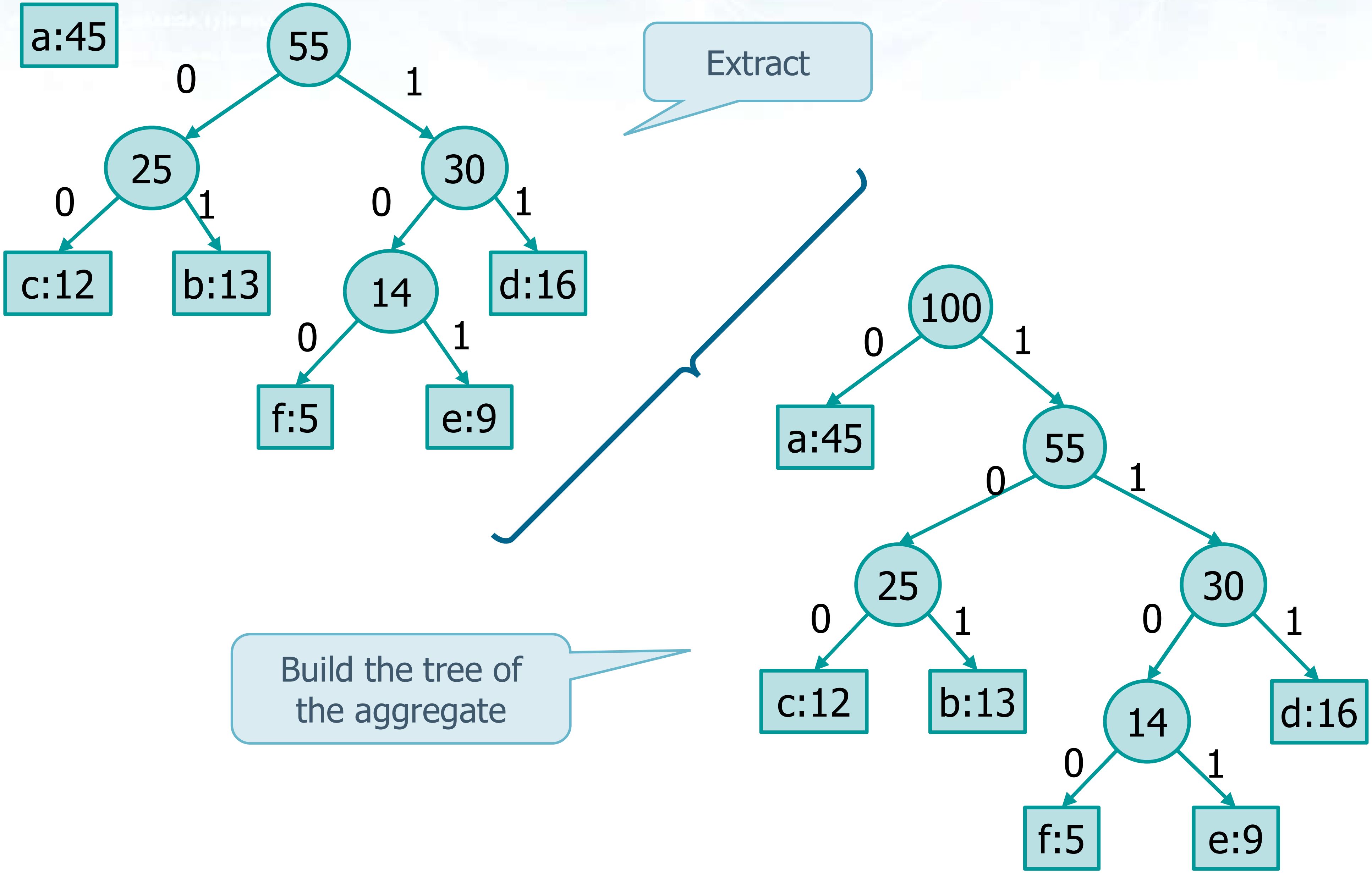
Solution: Step 3



Solution: Step 4



Solution: Step 5



Implementation

Functions **pq_*** are in the priority queue (PQ) library

Init Heap / Code

```
PQ *pq;

pq = pq_init (maxN, item_compare);

for (i=0; i<maxN; i++) {
  printf ("Enter letter & frequency: ");
  scanf ("%s %d", &letter, &freq);

  tmp = item_new (letter, freq);

  pq_insert (pq, tmp);
}
```

Functions **item_*** are in the data-item library

Implementation

pq_extract_max: Maximum priority minimum frequency

Generate code

```
while (pq_size(pq) > 1) {

  l = pq_extract_max (pq);
  r = pq_extract_max (pq);
  tmp = item_new ('!', l->freq + r->freq);
  tmp->left = l;
  tmp->right = r;
  pq_insert (pq, tmp);

}

root = pq_extract_max (pq);

pq_display (root, code, 0);
```

Visit and print tree (and codes)

Complexity

- When
  - Heap is implemented as a binary tree
  - Extract and insert operations manipulate a priority queues

the complexity of the algorithm is

$$T(n) = O(n \cdot log_2 n)$$

Exercise

- Given the following set of activities, find the a maximum-size subset of mutually compatible activities

Exercise A

Activity	s_i	f_i
P_1	1	4
P_2	3	5
P_3	7	15
P_4	6	9
P_5	11	13
P_6	11	12
P_7	5	8
P_8	4	9

Exercise B

Activity	s_i	f_i
P_1	7	8
P_2	21	23
P_3	20	24
P_4	4	5
P_5	15	17
P_6	0	3
P_7	6	7
P_8	27	31
P_9	8	12
P_{10}	26	32
P_{11}	3	8
P_{12}	29	31
P_{13}	9	11

Solution

- Selected activities

Solution A

Activity	s_i	f_i
P_1	1	4
P_7	5	8
P_6	11	12

Solution B

Activity	s_i	f_i
P_6	0	3
P_4	4	5
P_7	6	7
P_1	7	9
P_{13}	9	11
P_5	15	17
P_2	21	23

Exercise

- Given the following frequencies

Exercise A

$A{:}\,13 \quad B{:}\,29 \quad C{:}\,35 \quad D{:}\,8 \quad E{:}\,20 \quad F{:}\,6 \quad G{:}\,17 \quad H{:}\,5$

Exercise B

$A{:}\,6 \quad B{:}\,20 \quad C{:}\,14 \quad D{:}\,3 \quad E{:}\,35 \quad F{:}\,13 \quad G{:}\,24 \quad H{:}\,19 \quad I{:}\,12 \quad J{:}\,17$

  - Find an optimal Huffman code for all symbols in the set using a greedy algorithm
  - Indicate the code that must be used to represent all symbols in the set

Solution A

$A: 13 \quad B: 29 \quad C: 35 \quad D: 8 \quad E: 20 \quad F: 6 \quad G: 17 \quad H: 5$

Frequencies

Algorithmic steps

```
{H}(5)[-1][-1] + {F}(6)[-1][-1] = {H+F}(11)[0][1]
{D}(8)[-1][-1] + {H+F}(11)[0][1] = {D+H+F}(19)[2][3]
{A}(13)[-1][-1] + {G}(17)[-1][-1] = {A+G}(30)[4][5]
{D+H+F}(19)[2][3] + {E}(20)[-1][-1] = {D+H+F+E}(39)[6][7]
{B}(29)[-1][-1] + {A+G}(30)[4][5] = {B+A+G}(59)[8][9]
{C}(35)[-1][-1] + {D+H+F+E}(39)[6][7] =
   = {C+D+H+F+E}(74)[10][11]
{B+A+G}(59)[8][9] + {C+D+H+F+E}(74)[10][11] =
   = {B+A+G+C+D+H+F+E}(133)[12][13]
```

B	00
A	010
G	011
C	10
D	1100
H	11010
F	11011
E	111

Final encoding

Solution B

$A: 6 \quad B: 20 \quad C: 14 \quad D: 3 \quad E: 35 \quad F: 13 \quad G: 24 \quad H: 19 \quad I: 12 \quad J: 17$

Frequencies

Algorithmic steps

```
{D}(3)[-1][-1] + {A}(6)[-1][-1] = {D+A}(9)[0][1]
{D+A}(9)[0][1] + {I}(12)[-1][-1] = {D+A+I}(21)[2][3]
{F}(13)[-1][-1] + {C}(14)[-1][-1] = {F+C}(27)[4][5]
{J}(17)[-1][-1] + {H}(19)[-1][-1] = {J+H}(36)[6][7]
{B}(20)[-1][-1] + {D+A+I}(21)[2][3] = {B+D+A+I}(41)[8][9]
{G}(24)[-1][-1] + {F+C}(27)[4][5] = {G+F+C}(51)[10][11]
{E}(35)[-1][-1] + {J+H}(36)[6][7] = {E+J+H}(71)[12][13]
{B+D+A+I}(41)[8][9] + {G+F+C}(51)[10][11] =
   {B+D+A+I+G+F+C}(92)[14][15]
{E+J+H}(71)[12][13] + {B+D+A+I+G+F+C}(92)[14][15] =
   {E+J+H+B+D+A+I+G+F+C}(163)[16][17]
```

E 00
J 010
H 011
B 100
D 10100
A 10101
I 1011
G 110
F 1110
C 1111

Final encoding