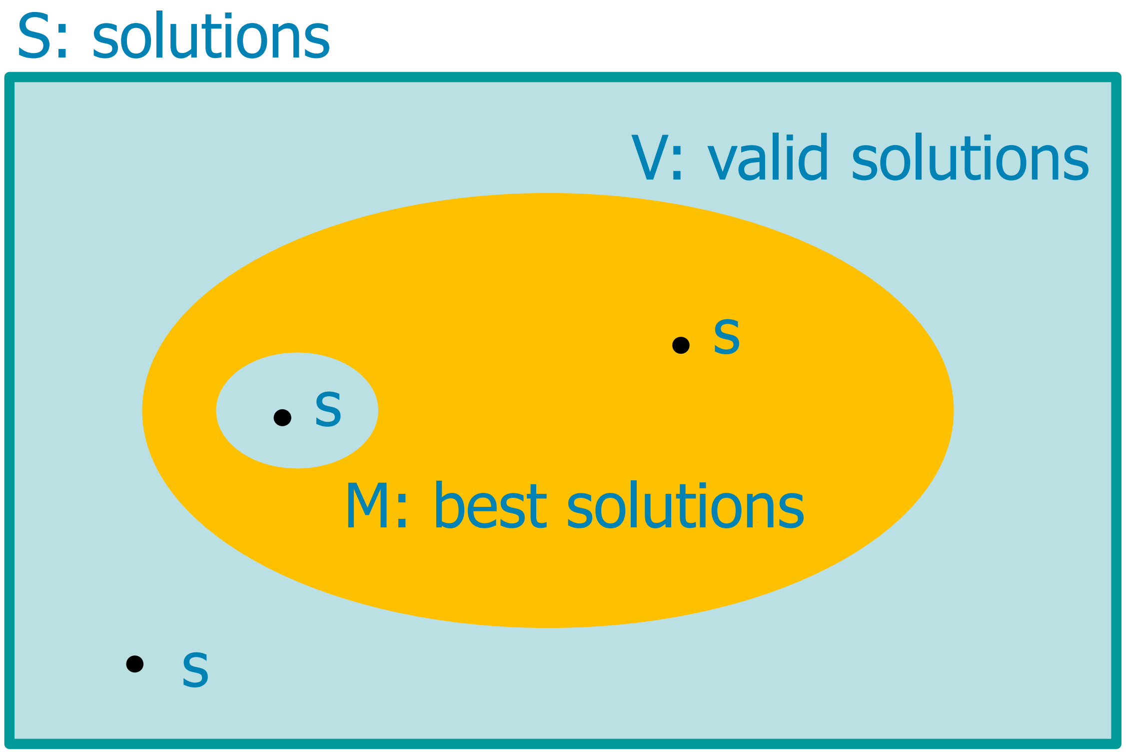
Greedy Algorithms

Greedy Algorithms

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Optimization Algorithms

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step

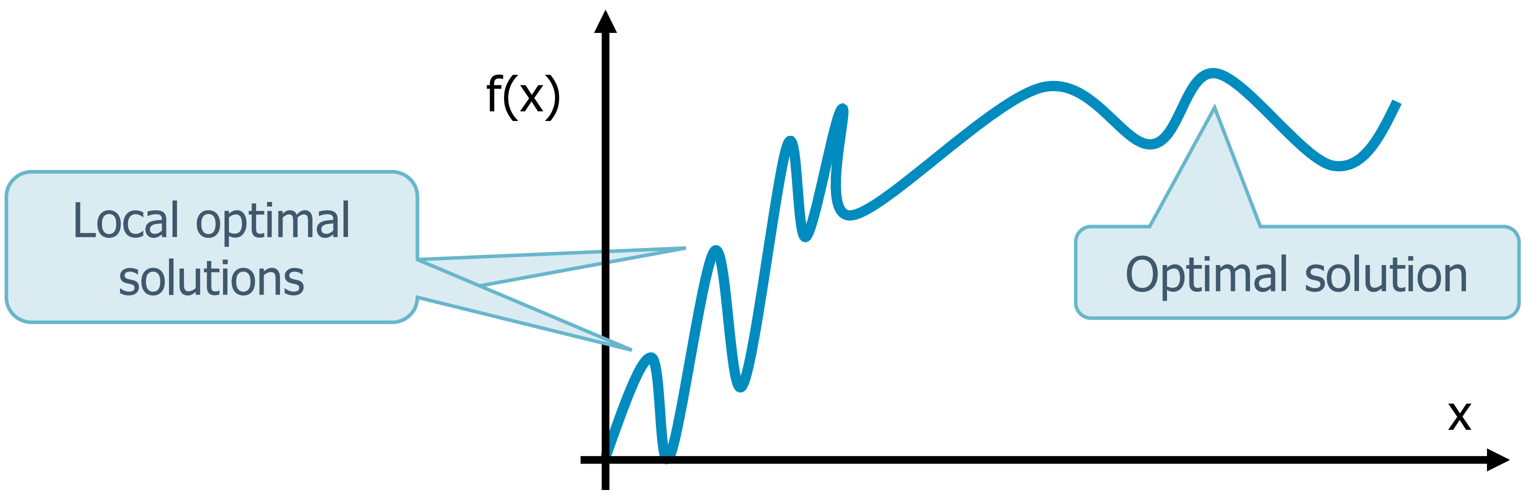


Greedy Algorithms

- For many optimization problems, using brute-force recursion or dynamic programming to determine the best choices is overkill
- Sometimes simpler, more efficient algorithms will solve the problem efficiently
- A **greedy algorithm** always makes the choice that looks best at the moment

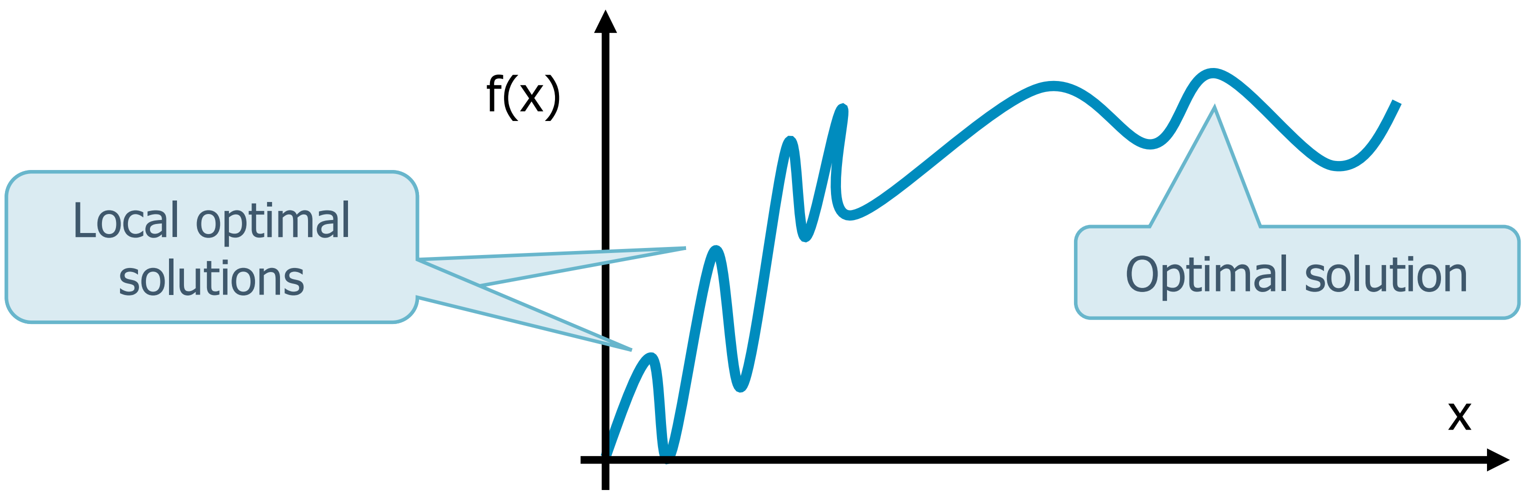
Greedy Algorithms

- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
  - Optimal solution
    - Best possible solution
  - Locally optimal solution
    - Best possible solution within a contiguous domain



Greedy Algorithms

- Greedy algorithms do not always yield optimal solutions, but for many problems they do
- The greedy method is quite powerful and works well for a wide range of problems



Greedy Algorithms

- At each step
  - To find globally optimal solutions locally optimal solutions are selected
  - Decisions taken at each step are never reconsidered (no backtrack)
  - Decisions are considered locally optimal based on an appetibility/cost function
- Advantages
  - Very simple algorithm
  - Limited processing time
- Disadvantages
  - Global solution is not necessarily optimal

Greedy Algorithms

- Appetibility values known a priori and never changed thereafter
  - Start
    - Empty solution
  - Sort choices according to decreasing appetibility values
  - Execute choices in descending appetibility order, adding, if possible, the result to the partial solution.
- Modifyiable appetibility values
  - As before, but appetibility values are stored in a priority queue

Greedy Algorithms

- In this unit we will analyse two algorithms
  - The activity-selection problem
  - The Huffman codes generation

Activity Selection Problem

- Input
  - Set of n activities with start time and end time [s, f)
- Output
  - Set with the maximum number of compatible activities
- Compatibility
  - $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
  - That is $s_i \geq f_j$ or $s_j \geq f_i$
- Greedy approach
  - Sort the activities by increasing end time

An example

Initial activity (sorted)

Selected activity

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

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

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);
...
```

Compare Function

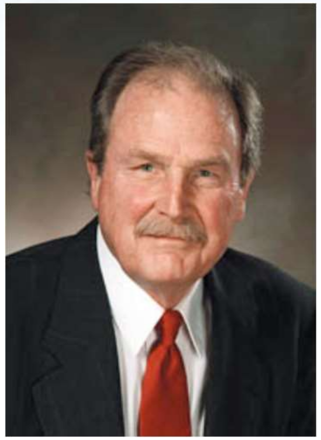
C Standard Library

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;
    }
  }
}
```

Huffman Codes



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

Huffman Codes

- Fixed-length codes
  - Codewords with $n = \lceil \log 2 \ (card(S)) \rceil$ bits
  - Pro: easy to decode
  - Use: symbol occurring with the same frequency
- Variable-length codes
  - Con: difficult to decode
  - Pro: memory savings
  - Use: symbols occurringwith different frequencies
  - Example
    - Morse alphabet (with pauses between words)

The Morse Code

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

A	· −	U	· · −
B	− · · ·	V	· · · −
C	− · − ·	W	· − −
D	− · ·	X	− · · −
E	·	Y	− · − −
F	· · − ·	Z	− − · ·
G	− − ·		
H	· · · ·		
I	· ·		
J	· − − −		
K	− · −	1	· − − − −
L	· − · ·	2	· · − − −
M	− −	3	· · · − −
N	− ·	4	· · · · −
O	− − −	5	· · · · ·
P	· − − ·	6	− · · · ·
Q	− − · −	7	− − · · ·
R	· − ·	8	− − − · ·
S	· · ·	9	− − − − ·
T	−	0	− − − − −

Example

- Give a file with 100.000 characters
- Fixed-length code
  - $3 \cdot 100.000 = 300.000$ bits
- Variable-length code
  - $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1.000 = 224.000$ bits

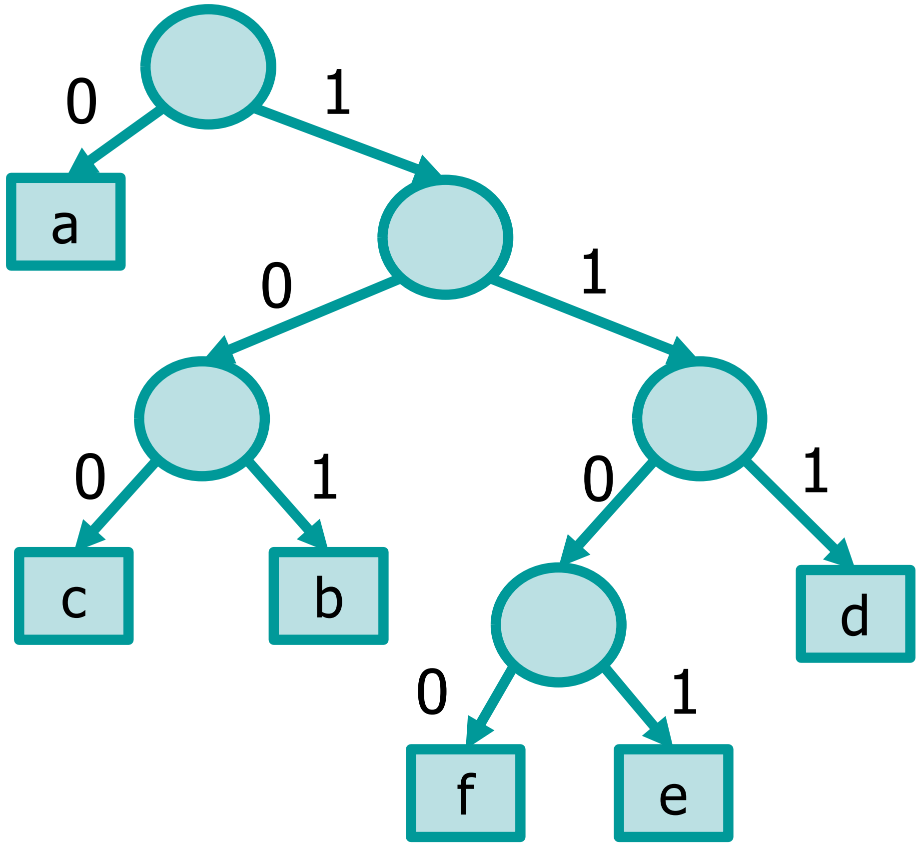
	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length	000	001	010	011	100	101
Variable-length	0	101	100	111	1101	1100

Prefix code

- Prefix-(free) code
  - No valid codeword is a prefix of another valid codeword
  - Encoding
    - Juxtapposition of strings
  - Decoding
    - Path on a binary tree

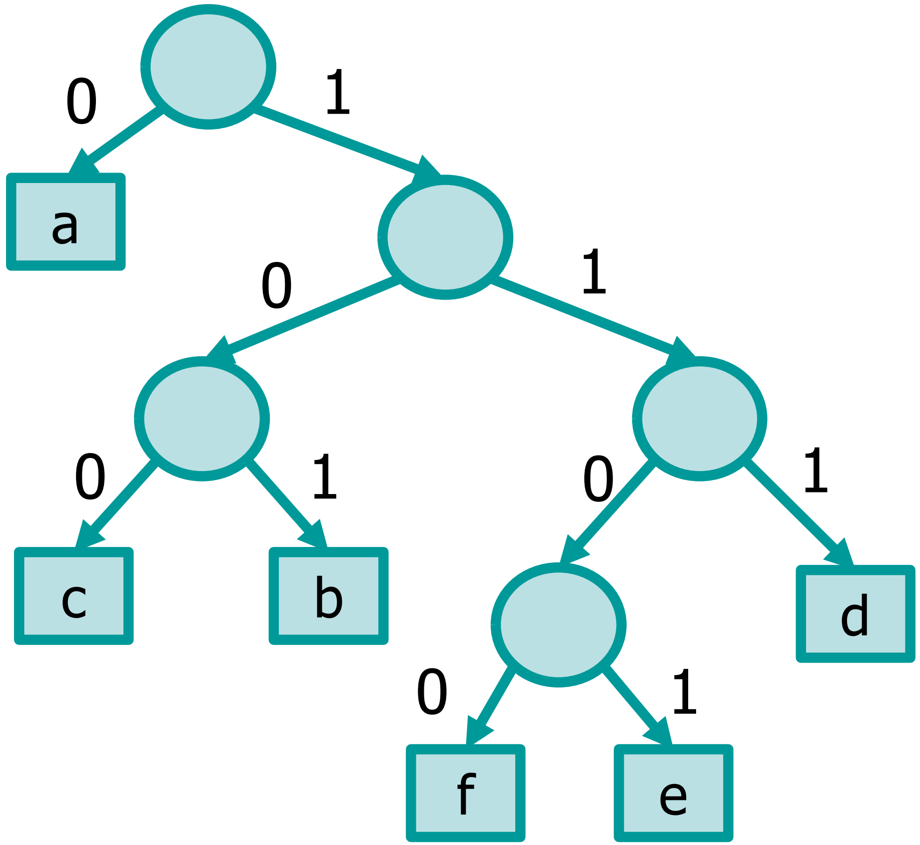
Example

- Symbols to codes correspondence (tree)
  - a=0
  - b=101
  - c=100
  - d=111
  - e=1101
  - f=1100



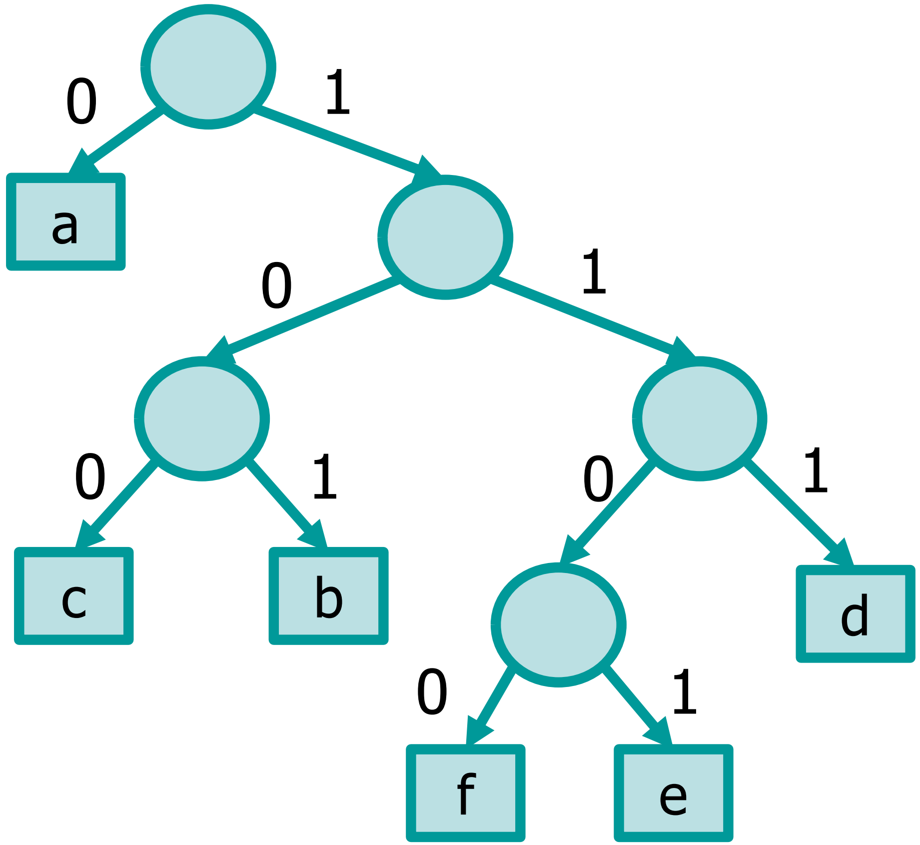
Example: Encoding

- Encoding: From symbols to code
  - abfaac → 010111000100



Example: Decoding

- Decoding: From code to symbols
  - 0101110000100 → abfaac



Building the tree

- Data structure
  - Priority queue
- Initially
  - Symbol = leaf
- Intermediate step
  - Extract the 2 symbols (or aggregates) with minimum frequency
  - Build the binary tree (aggregate of symbols)
  - Node = symbol or aggregate
  - Frequency = sum of frequencies
  - Insert into priority queue
- Termination
  - Empty queue

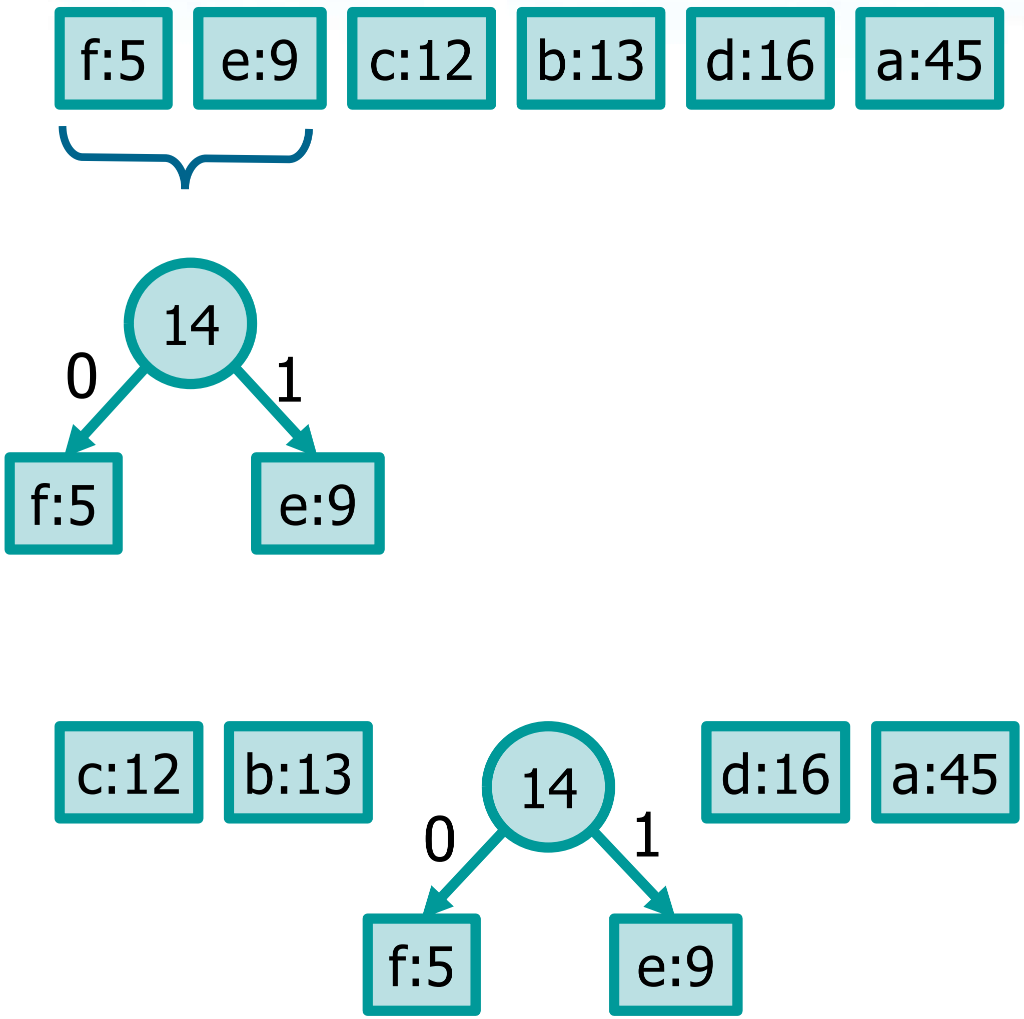
Example: Step 1

Priory Queue (fully sorted)

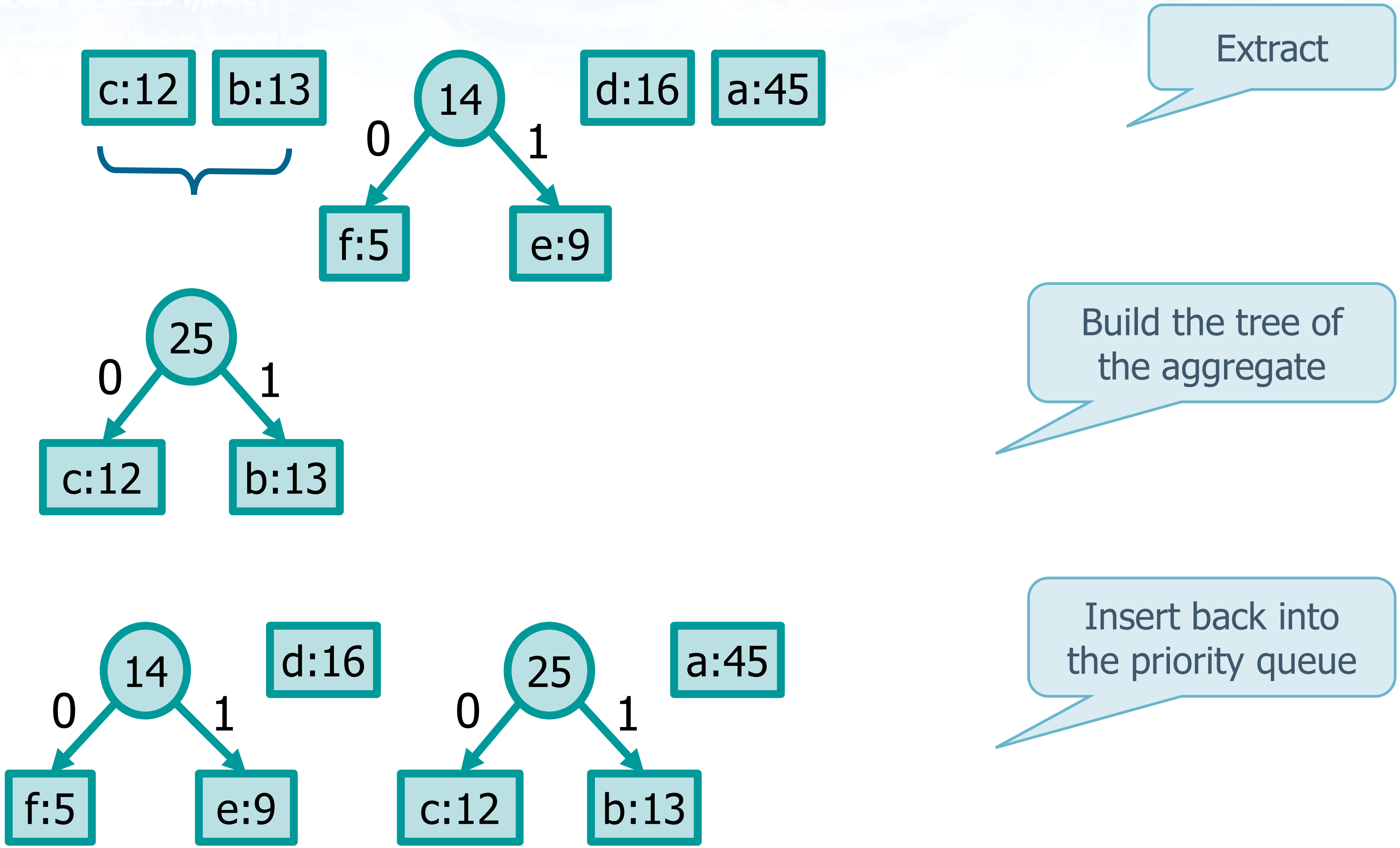
Extract

Build the tree of the aggregate

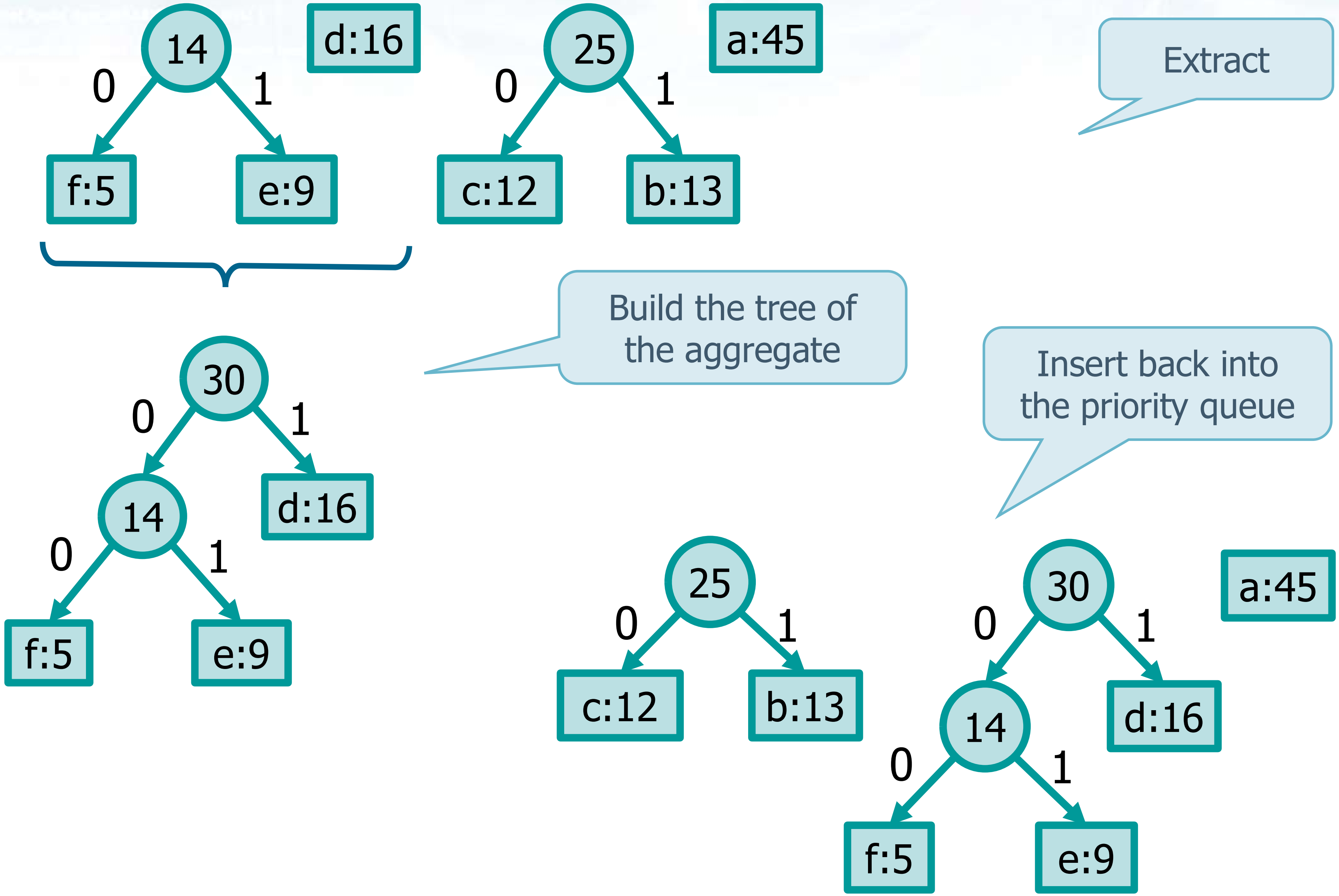
Insert back into the priority queue



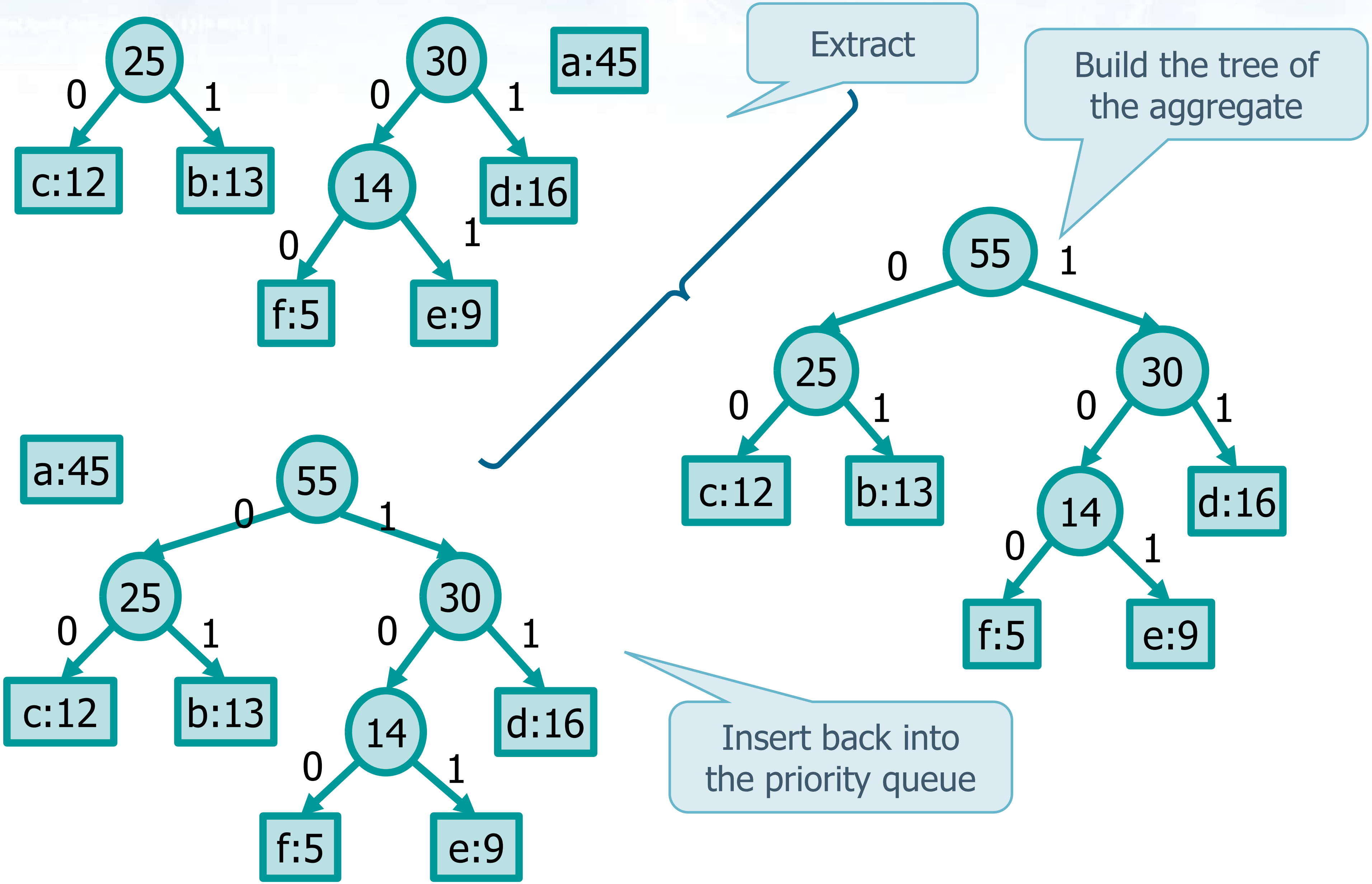
Example: Step 2



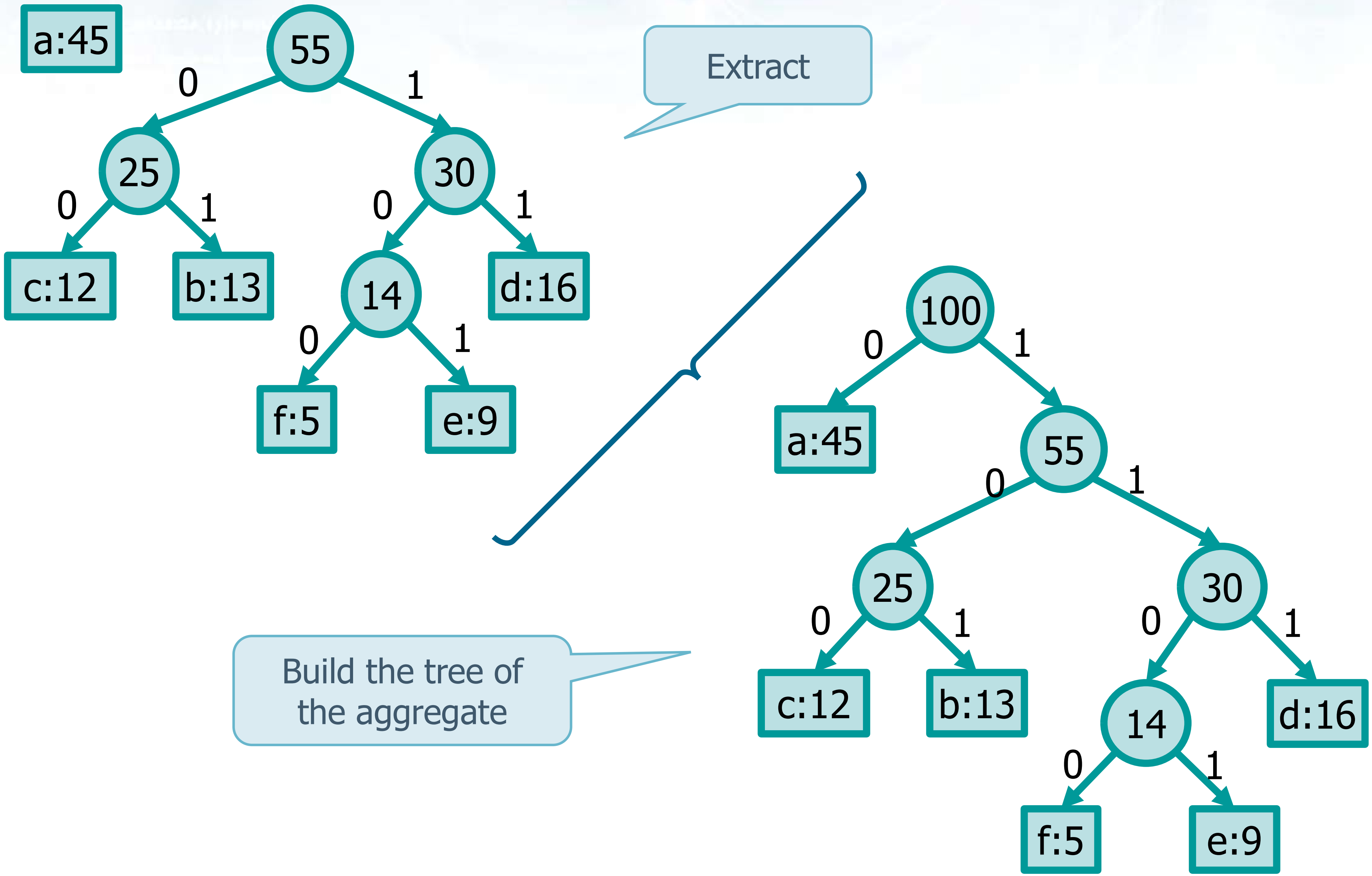
Example: Step 3



Example: Step 4



Example: Step 5



Implementation

Functions **pq_*** are within 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 within 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

- Heap implemented as a binary tree
- Extract and insert operations in priority queues
  - $T(n) = O(n \log n)$

Exercise

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

Activity	s_i	f_i
P_1	1	4
P_2	3	6
P_3	1	8
P_4	3	5
P_5	4	9
P_6	5	11
P_7	2	9

Exercise

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

Activity	s_i	f_i
P_1	1	4
P_2	3	5
P_3	1	7
P_4	3	6
P_5	1	9
P_6	2	11
P_7	5	8
P_8	4	9

Exercise

- Using a greedy algorithm find an optimal Huffman code for the following symbols with the specified frequencies
  - g:13 f:29 h:35 i:8 j:20 k:60 l:27 m:50

Exercise

- Using a greedy algorithm find an optimal Huffman code for the following symbols with the specified frequencies
  - a:46 c:20 g:14 k:3 i:35 n:13 o:24 r:19 t:12 u:17