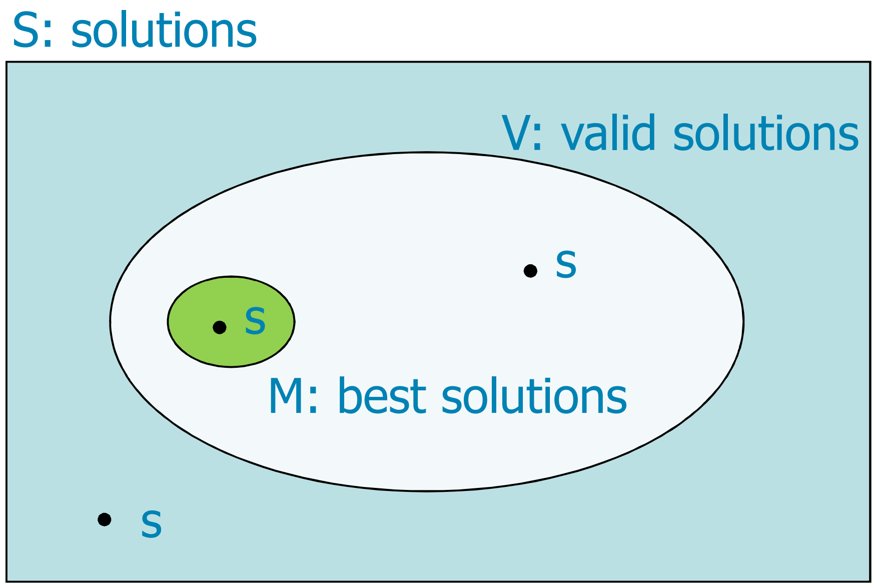
Greedy Algorithms

Greedy Algorithms

Paolo Camurati and 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

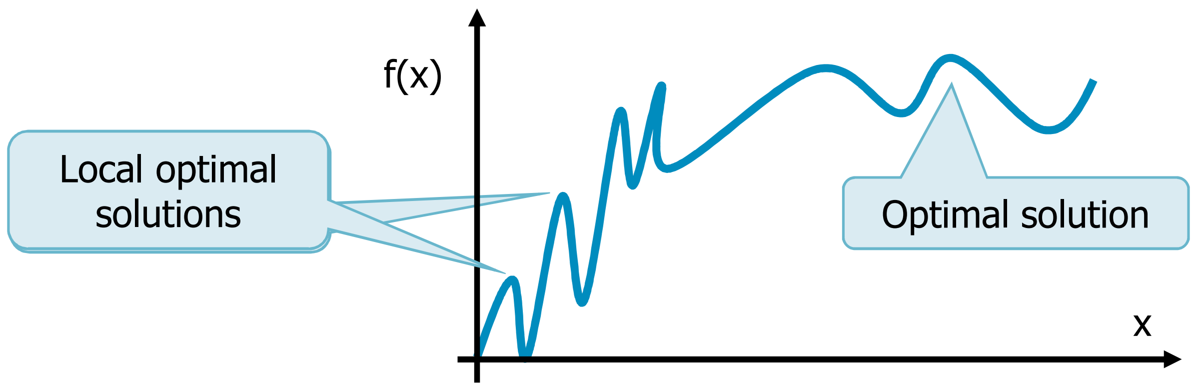


Greedy Algorithms

- 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

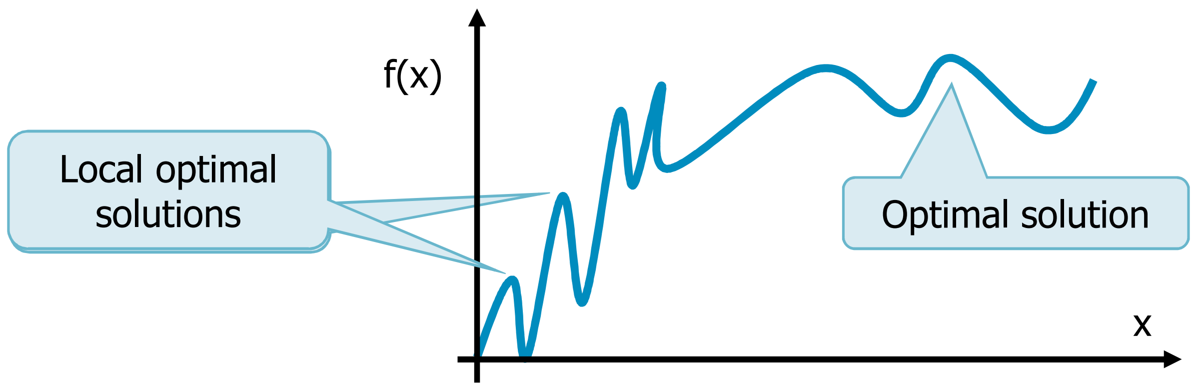
Greedy Algorithms

- 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

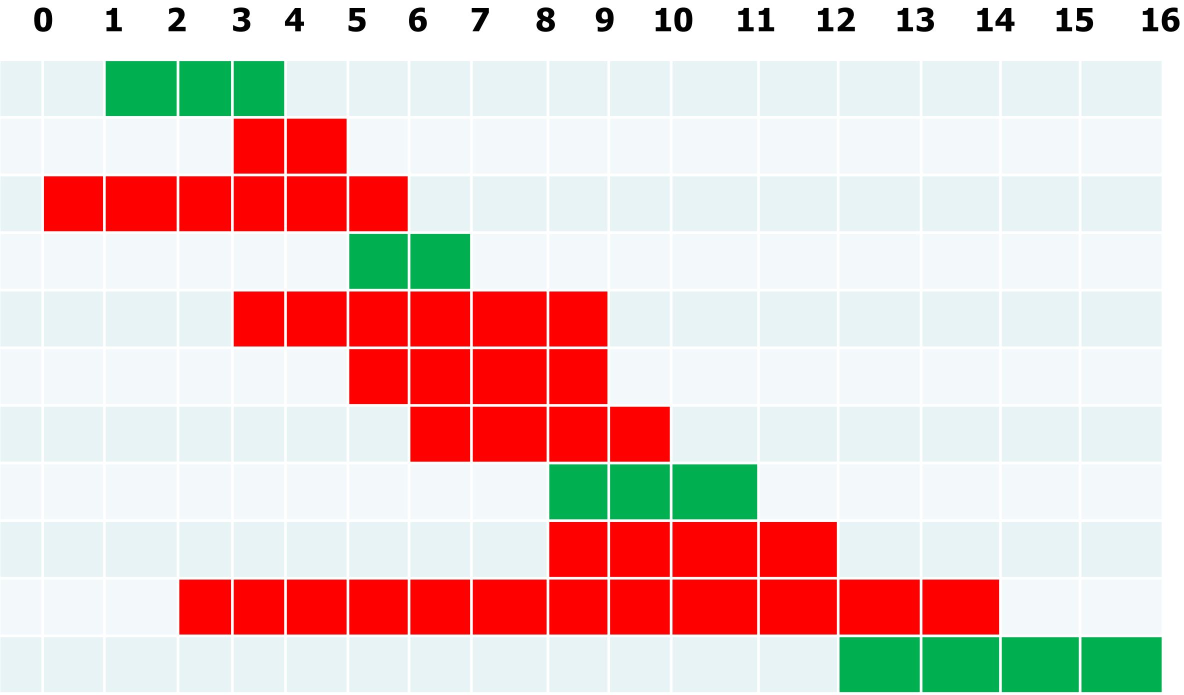
- Input
  - Set of n activities with start time and end time [s, f)
- Output
  - Set with the maximum number of compatible activities
- Compatibility
  - $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
  - That is $s_i \geq f_j$ or $s_j \geq f_i$
- Greedy approach
  - Sort the activities by increasing end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

An example

Initial activity (sorted)

Selected activity

k	s_k	f_k
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



Algorithm

```
/* 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

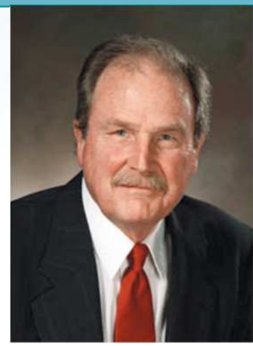
C Standard Library

Algorithm

```
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

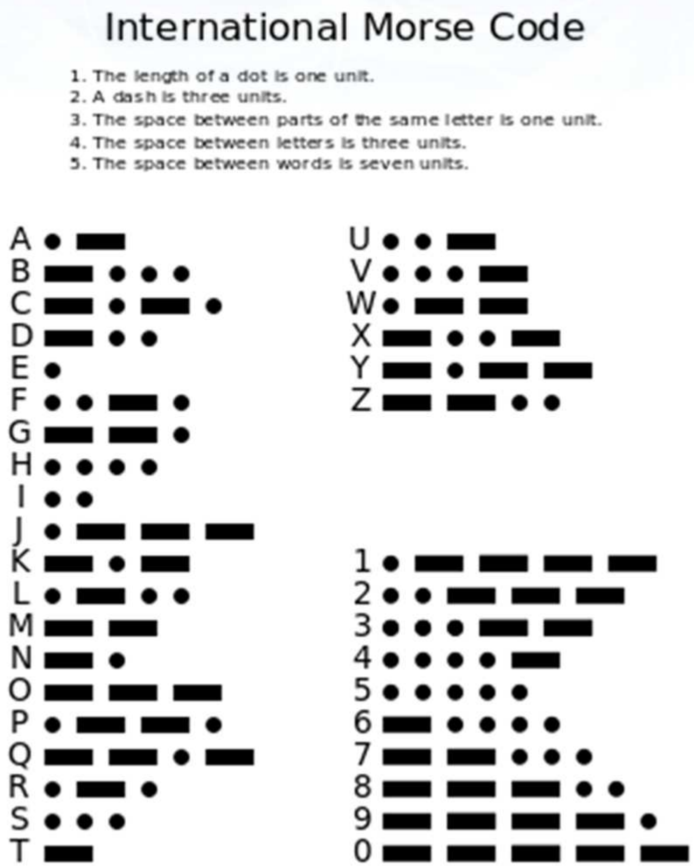


- Huffman in 1950 invented a greedy algorithm that construct an optimal prefix code
- Codeword
  - String of bits associated to a symbol $s \in S$
  - Fixed length
  - Variable length
- Encoding
  - From symbol to codeword
- Decoding
  - From codeword to symbol

Huffman Codes

- Fixed-length codes
  - Codewords with n = $\lceil \log2 (card(S)) \rceil$ 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



Example

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

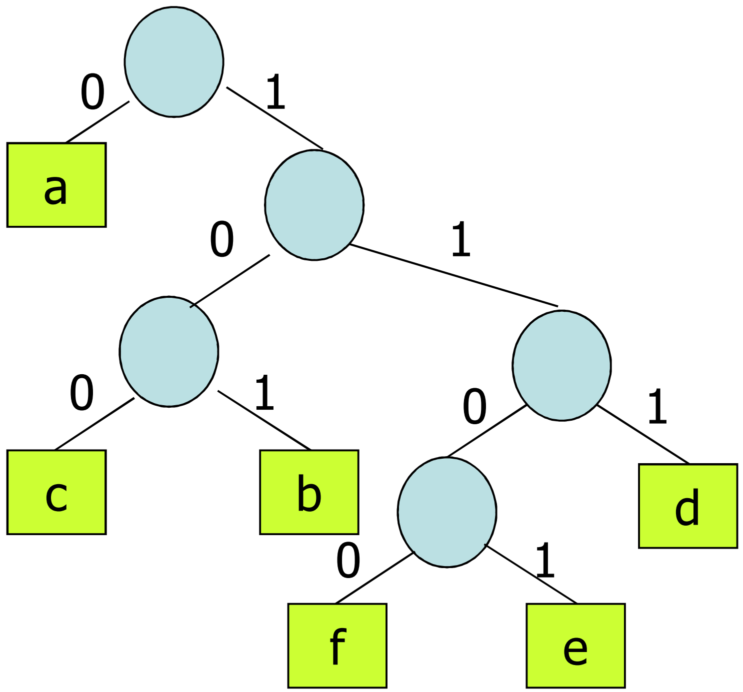
	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length	000	001	010	011	100	101
Variable-length	0	101	100	111	1101	1100

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

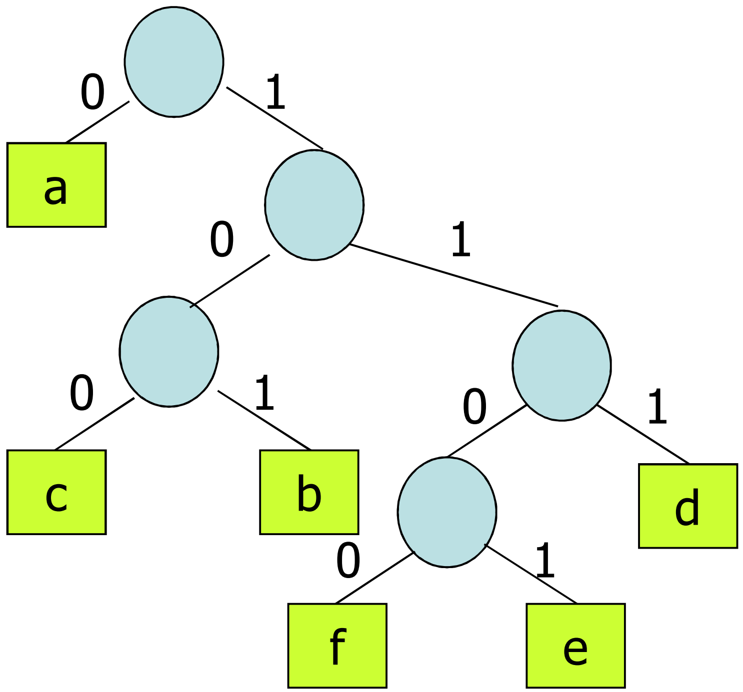
Example

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



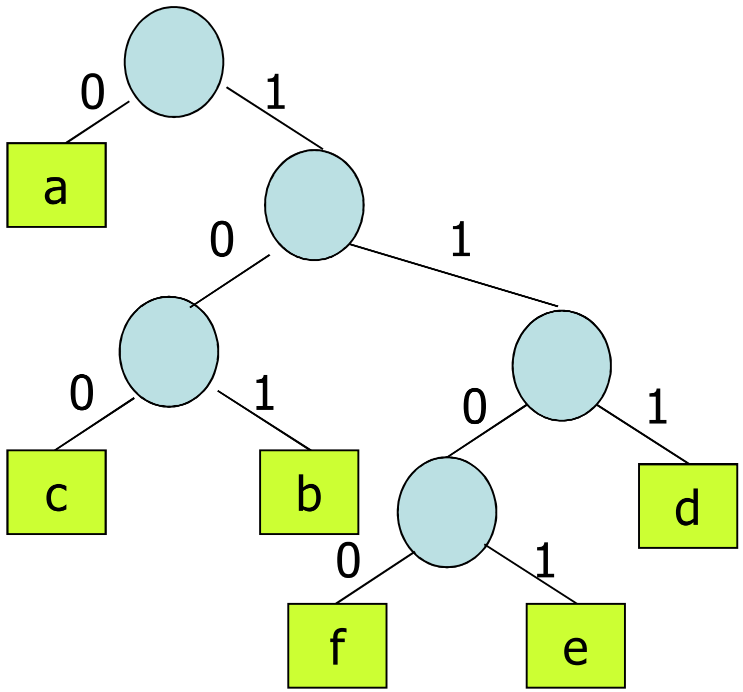
Example: Encoding

- From symbols to code (encoding)
  - abfaac → 0101110000100



Example: Decoding

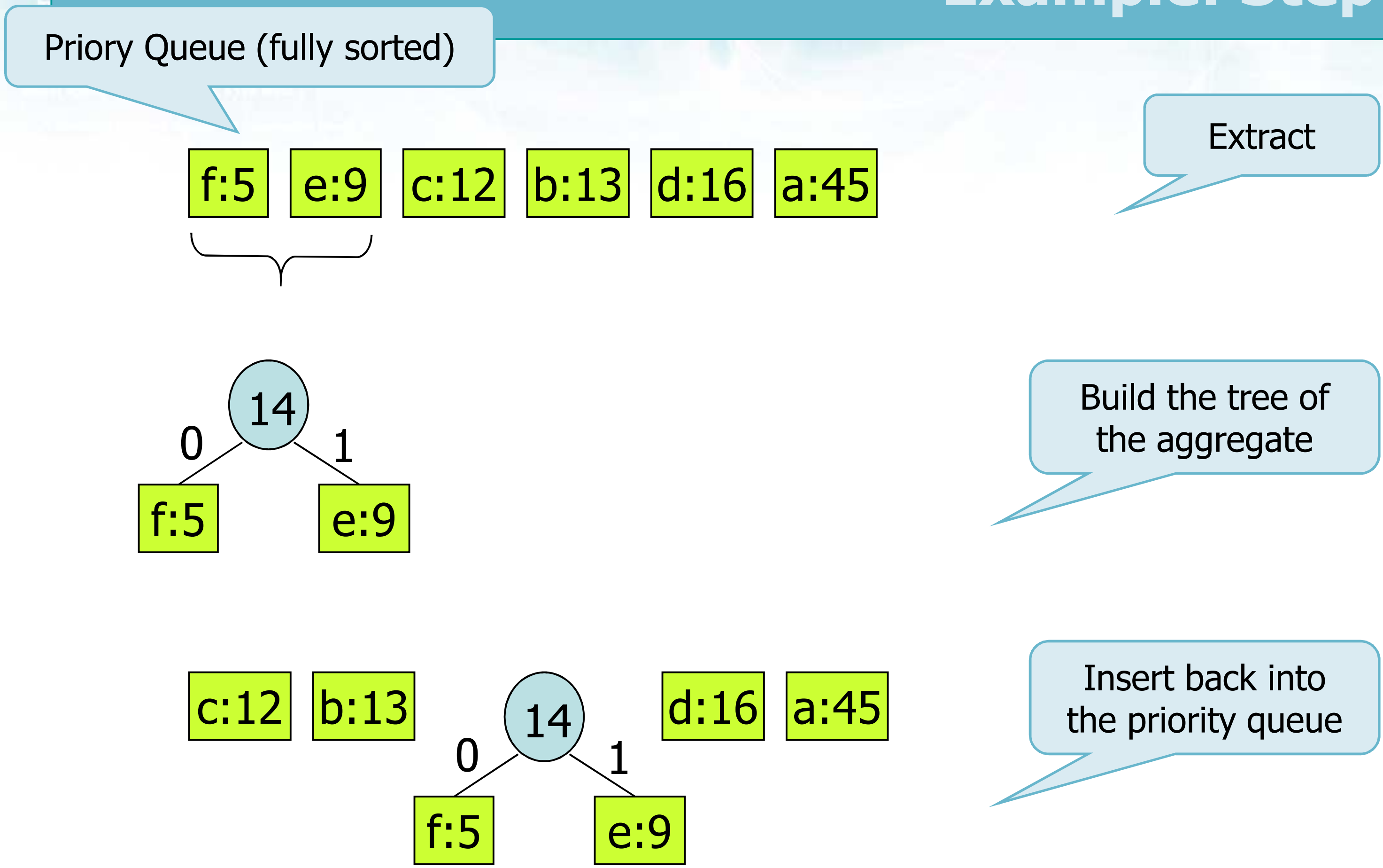
- From code to symbols (decoding)
  - 0101110000100 → abfaac



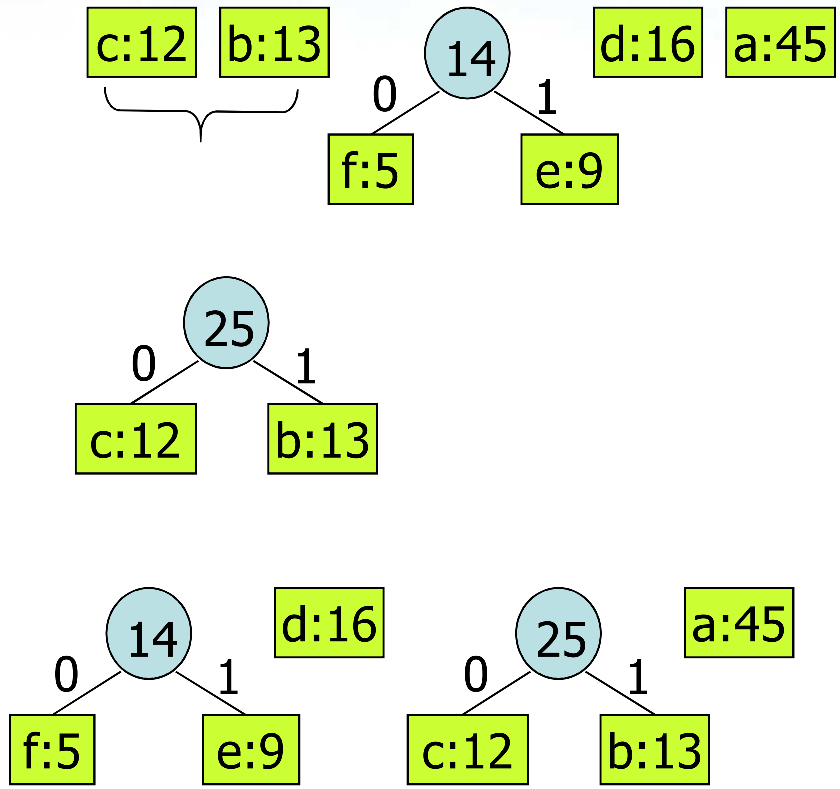
Building the tree

- 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



Example: Step 2

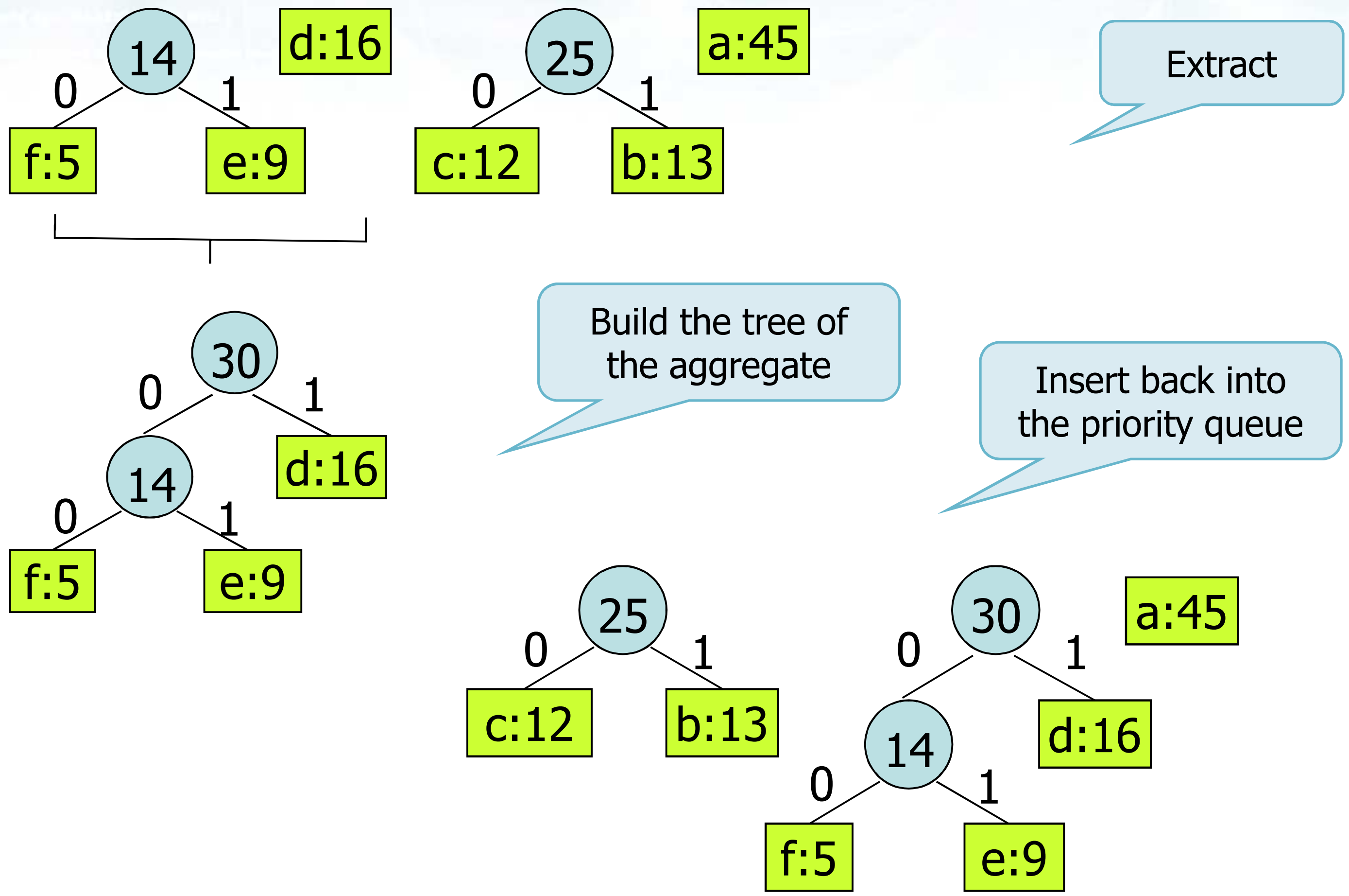


Extract

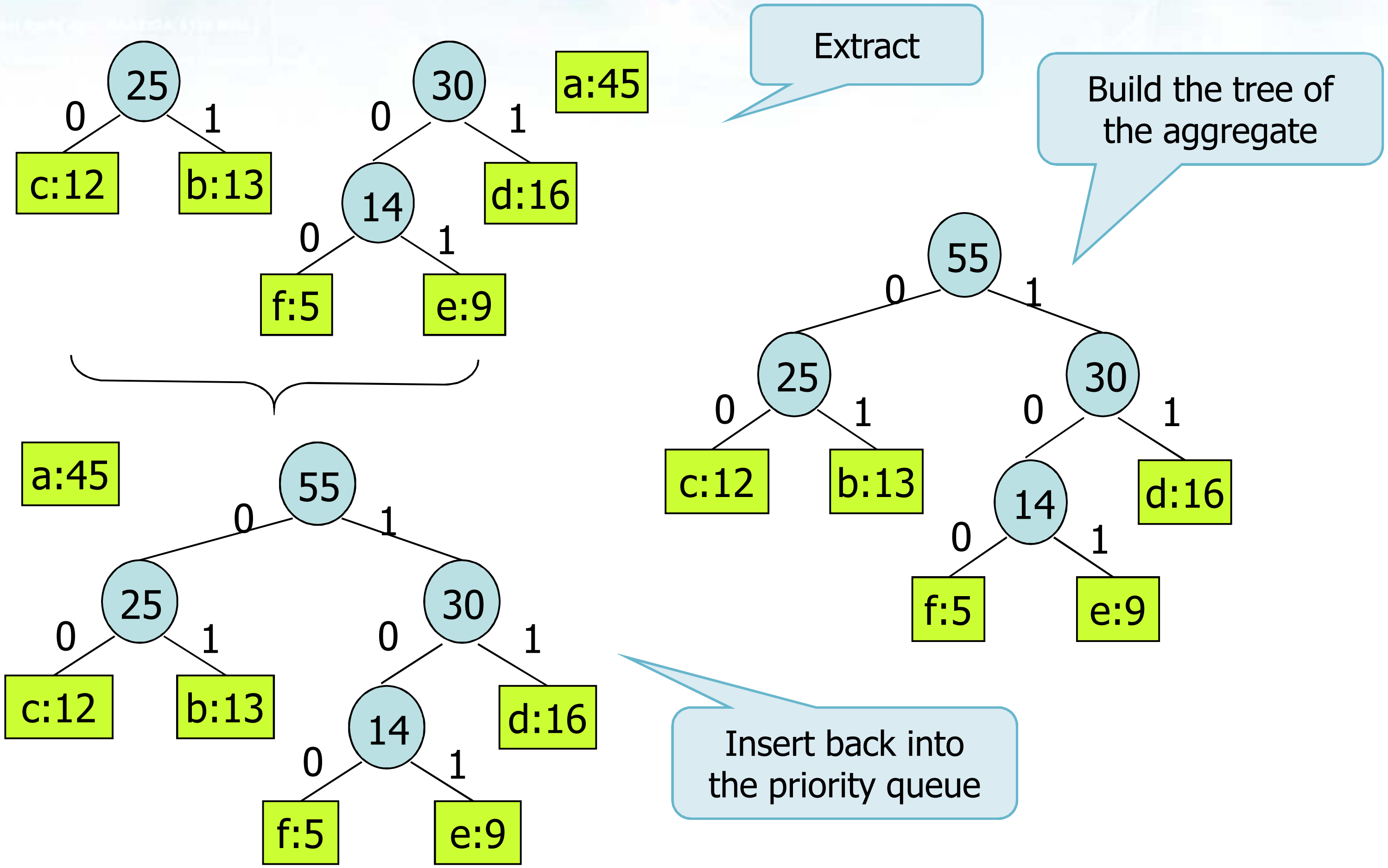
Build the tree of the aggregate

Insert back into the priority queue

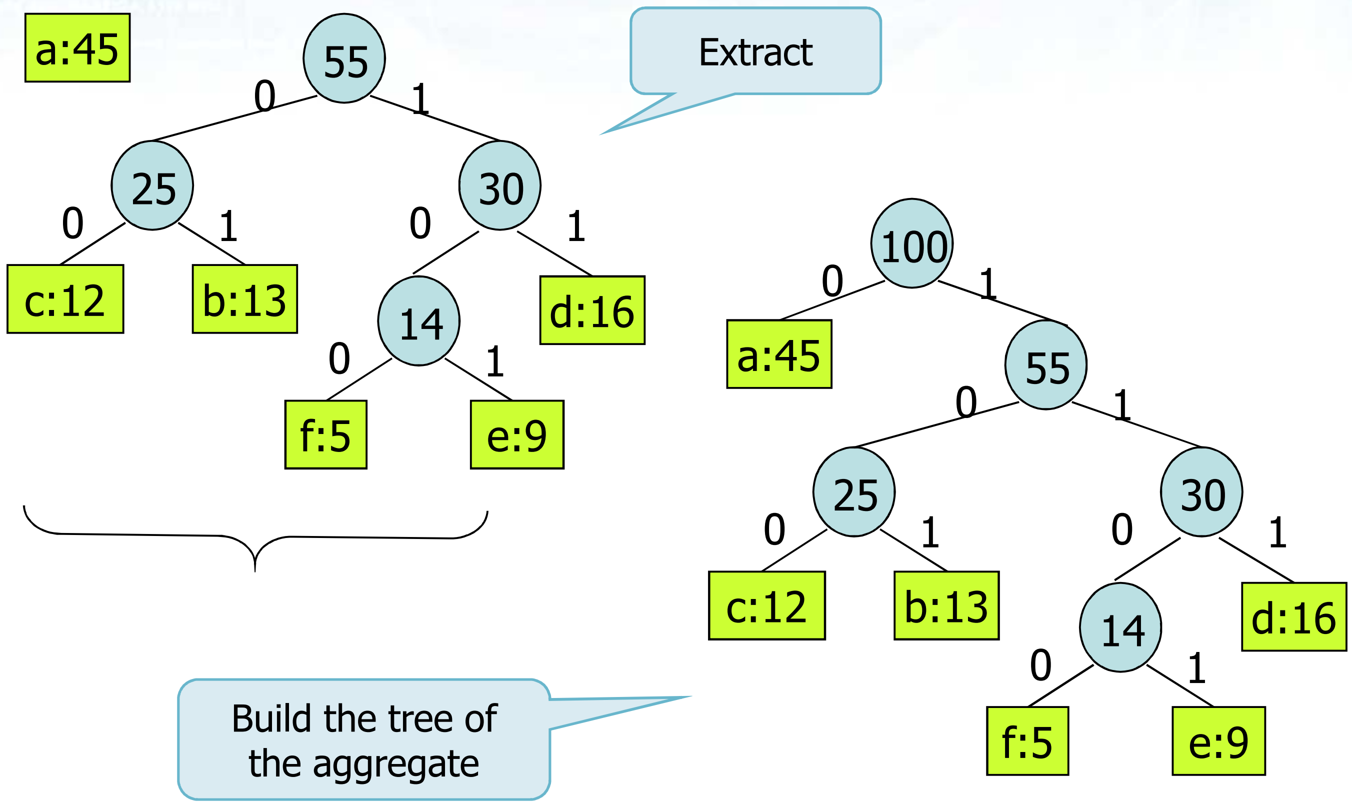
Example: Step 3



Example: Step 4



Example: Step 5



Algorithm

Init Heap / Code

```
PQ *pq;

pq = PQUEUEinit(maxN, ITEMcompare);

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

  tmp = ITEMnew(letter, freq);

  PQUEUEinsert(pq, tmp);
}
```

Algorithm

```
while (PQUEUEsize(pq) > 1) {
  l = PQUEUEextract(pq); r = PQUEUEextract(pq);
  tmp = ITEMnew('!', l->freq + r->freq);
  tmp->left = l;  tmp->right = r;
  PQUEUEinsert(pq, tmp);
}

root = PQUEUEextract(pq);

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

Generate code

Visit tree

Complexity

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