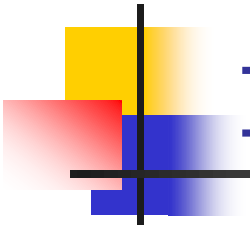
Il paradigma greedy

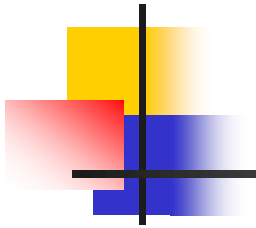
Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino



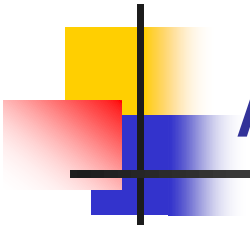
Il paradigma greedy

Il paradigma greedy è un'alternativa all'esplorazione esaustiva dello spazio delle possibilità per i problemi di ottimizzazione:

- a ogni passo: per trovare una soluzione globalmente ottima si scelgono soluzioni *localmente ottime*
- le scelte fatte ai singoli passi non vengono successivamente riconsiderate (no *backtrack*).
- scelte localmente ottime sulla base di una funzione di appetibilità (costo).



- Vantaggi
  - algoritmo molto semplice
  - tempo di elaborazione molto ridotto
- Svantaggi
  - soluzione non necessariamente ottima.



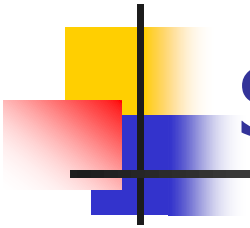
Algoritmo

Appetibilità note in partenza e non modificate:

- partenza: soluzione vuota
- ordina le scelte per appetibilità decrescenti
- esegui le scelte in ordine decrescente, aggiungendo, ove possibile, il risultato alla soluzione.

Appetibilità modificabili:

- come prima con modifica delle appetibilità e coda a priorità.

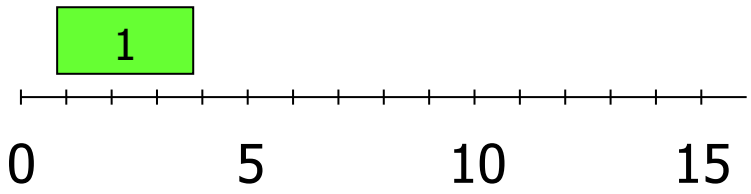


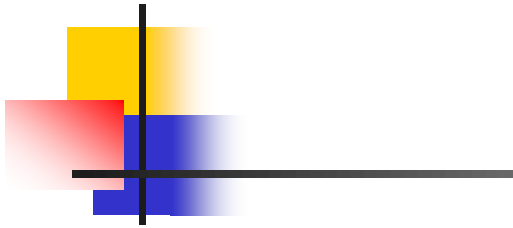
Selezione di attività

- Input: insieme di n attività caratterizzate da tempo di inizio e tempo di fine [s, f)
- Output: insieme con il massimo numero di attività compatibili
- Compatibilità: $[s_i, f_i)$ e $[s_j, f_j)$ non si sovrappongono, cioè $s_i \geq f_j$ oppure $s_j \geq f_i$
- Approccio greedy: ordinamento delle attività per tempo di fine crescente.

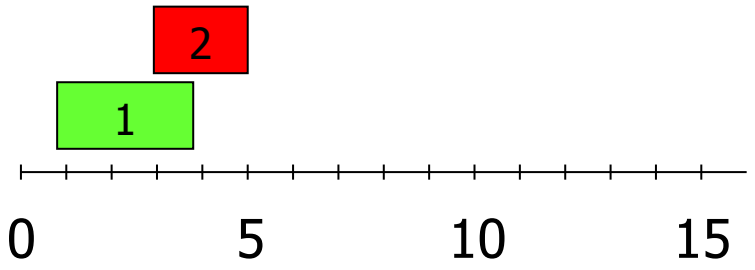
Esempio

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

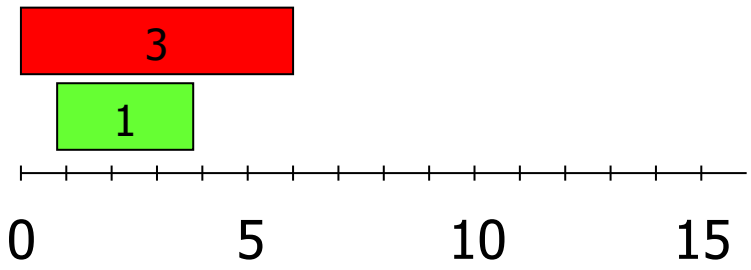




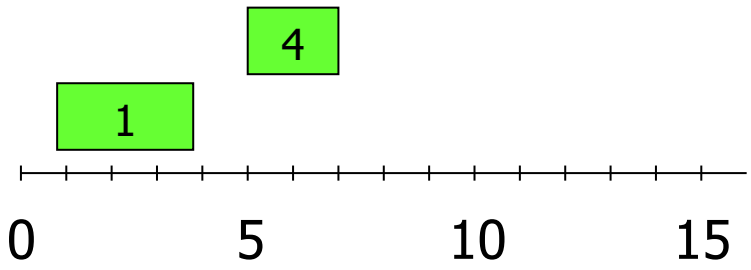
i	S_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



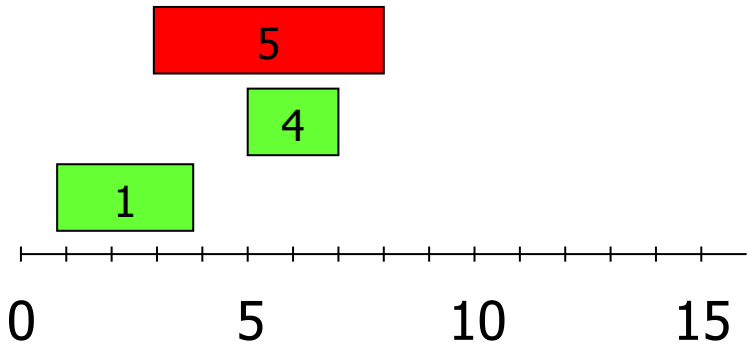
i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



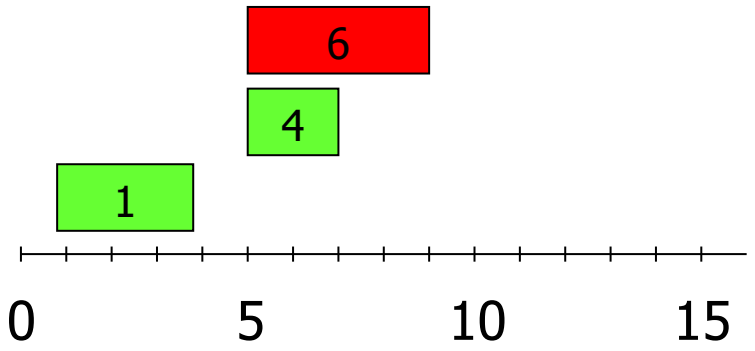
i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



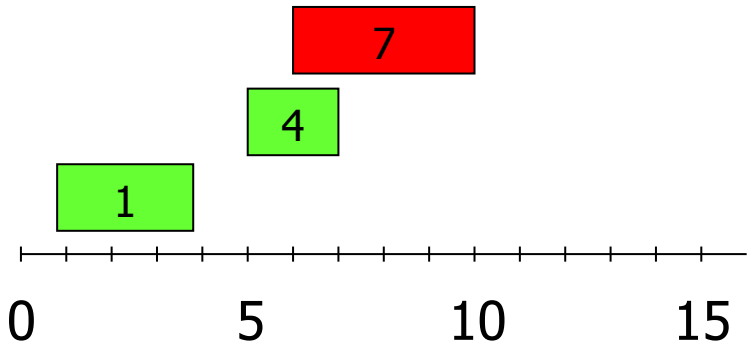
i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



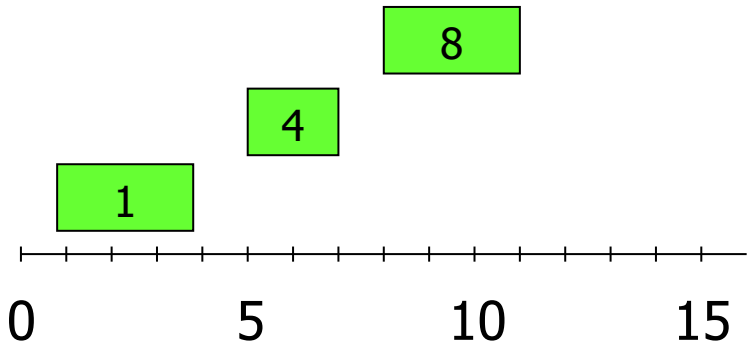
i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



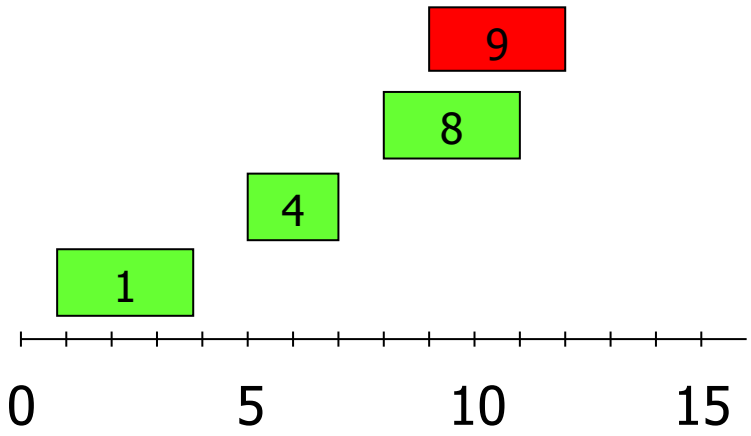
i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



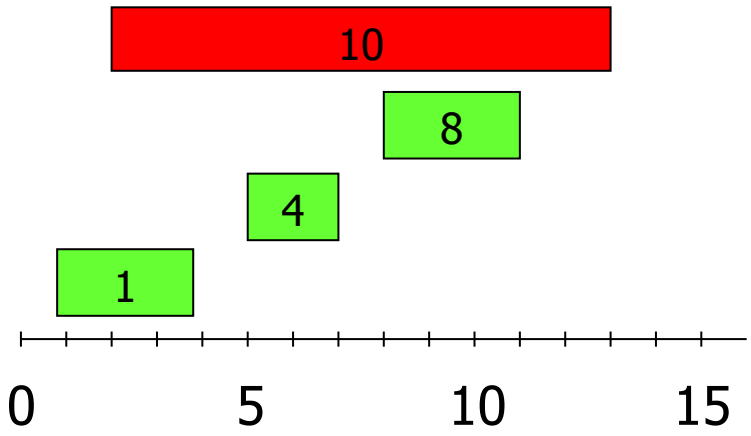
i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



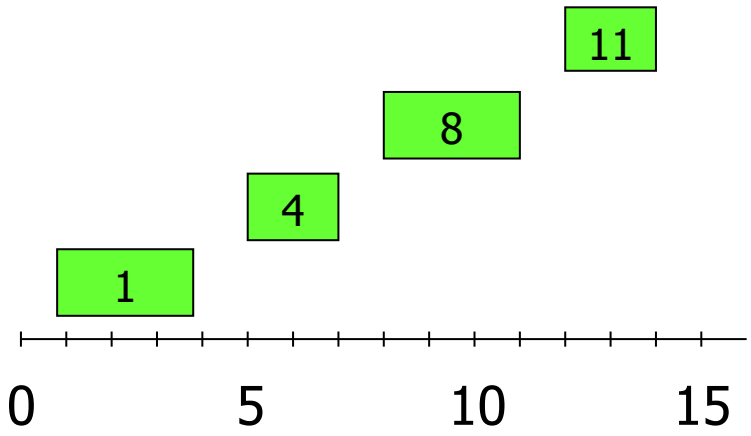
i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



01coinchanger.c

Il cambiamonete

- Input: monetazione, resto da erogare
- Output: resto con numero minimo di monete
- Appetibilità: valore della moneta
- Approccio greedy: a ogni passo moneta di maggior valore inferiore al resto residuo.

```
for (i=0; i < numden; i++) {
  coins[i] = amount / den[i];
  amount = amount - (amount/den[i])*den[i];
  printf("n. of  %d cent coins = %d\n",den[i],coins[i]);
}
```

Esempio

Monetazione:

20, 10, 5, 2, 1

Resto:

55

Passo	Resto residuo	Moneta scelta
0	55	2x20
1	15	10
2	5	5
3	0	

Esempio

Monetazione:

6, 4, 1

Resto:

9

soluzione ottima:
4,4,1

Passo	Resto residuo	Moneta scelta
0	9	6
1	3	3x1
2	0	

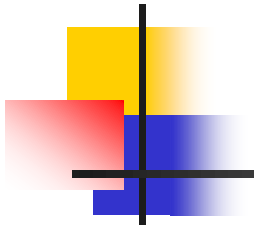
soluzione
non ottima!

Il problema dello zaino (discreto)

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap, determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq$ cap
- $\sum_{j \in S} v_j x_j =$ MAX
- $x_j \in \{0,1\}$

Ogni oggetto o è preso ($x_j = 1$) o lasciato ($x_j = 0$).



Appetibilità: valore specifico v_j / w_j decrescente.

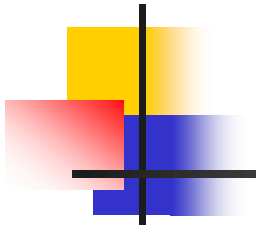
Approccio greedy: a ogni passo aggiungo l'oggetto a massimo valore specifico compatibile con il peso disponibile.

Esempio

cap = 50

		i=1	i=2	i=3
Valore	v_i	60	100	120
Peso	w_i	10	20	30
Val. spec.	v_i/w_i	6	5	4

Passo	Res	Oggetto	Valore
0	50	1	60
1	40	2	100
2	20		



Oggetti: 1 e 2
Valore 160:

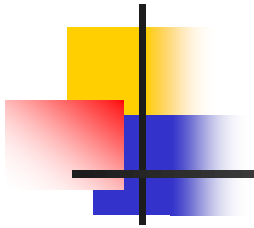
Oggetti: 2 e 3
Valore 220:

Il problema dello zaino (continuo)

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap, determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w\, x_j \leq P$
- $\sum_{j \in S} v_j\, x_j = MAX$
- $0 \leq x_j \leq 1$

Ogni oggetto può essere preso per una frazione x_j.



Appetibilità: valore specifico v_j / w_j decrescente.

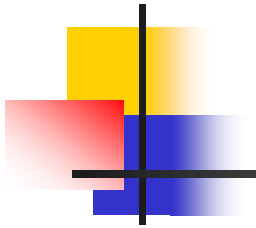
Approccio greedy: ad ogni passo aggiungo la frazione di oggetto a massimo valore specifico compatibile con il peso disponibile.

Esempio

cap = 50

		i=1	i=2	i=3
Valore	v_i	60	100	120
Peso	w_i	10	20	30
Val. spec.	v_i/w_i	6	5	4

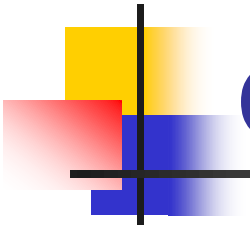
Passo	res	Oggetto	Valore	Frazione
0	50	1	60	1.00
1	40	2	100	1.00
2	20	3	120	0.66



Oggetti: 1, 2 e 2/3 di 3
Valore 240:

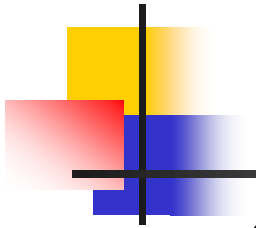
02fractionalknapsack.c

```
void knapsack(int n, float w[], float v[], float cap){
  float fract[maxobj], stolen = 0.0, res = cap;
  int i;
  for (i=0; i<n; i++)
    fract[i]=0.0;
  for (i=0; i<n && (w[i] <= res);i++) {
    fract[i] = 1.0;
    stolen = stolen + v[i];
    res = res - w[i];
  }
  fract[i] = res/w[i];
  stolen = stolen + (fract[i]*v[i]);
  printf("Results: \n");
  for(i = 0; i < n; i++)
    printf("Fraction of object = %f\n", fract[i]);
  printf("Total amount stolen: %f \n", stolen);
}
```



Codici di Huffman (1950)

- Codice: stringa di bit associata a un simbolo $s \in S$
  - a lunghezza fissa
  - a lunghezza variabile
- Codifica: da simbolo a codice
- Decodifica: da codice a simbolo



Codici a lunghezza fissa:

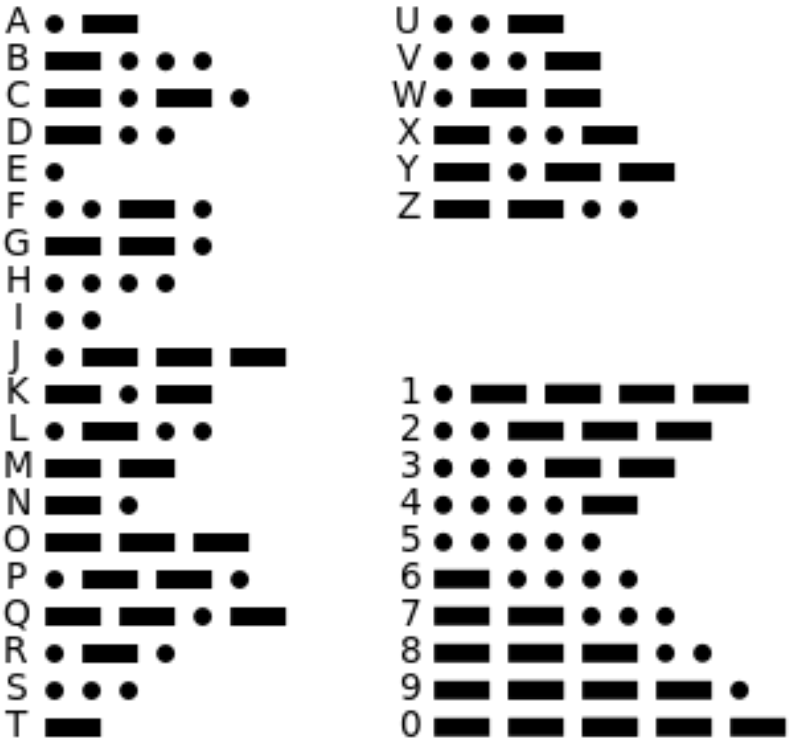
- numero di bit n = $\lceil \log_2 (\text{card}(S)) \rceil$
- vantaggio: facilità di decodifica
- uso: simboli isofrequenti

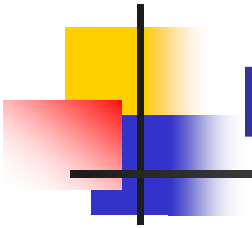
Codici a lunghezza variabile:

- svantaggio: difficoltà di decodifica
- vantaggio: risparmio di spazio di memoria
- uso: simboli con frequenze diverse
- esempio: alfabeto Morse (con pause tra parole).

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.





Esempio

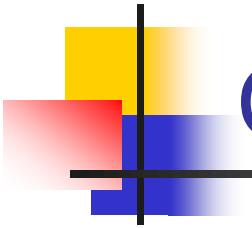
	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
codice fisso	000	001	010	011	100	101
codice variabile	0	101	100	111	1101	1100

file con 100.000 caratteri

codice fisso: 3 x 100.000 = 300.000 bit

codice variabile:

(45x1 + 13x3 + 12x3 + 16x3 + 9x4 + 5x 4) x 1.000 = 224.000 bit



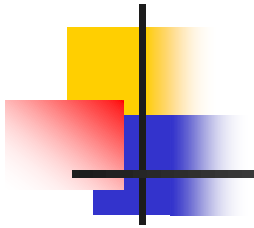
Codici prefissi

Codice (libero da) prefisso:

nessuna parola di codice valida è un prefisso di un'altra parola di codice.

Codifica: giustapposizione di stringhe

Decodifica: percorrimento di albero binario.



a = 0

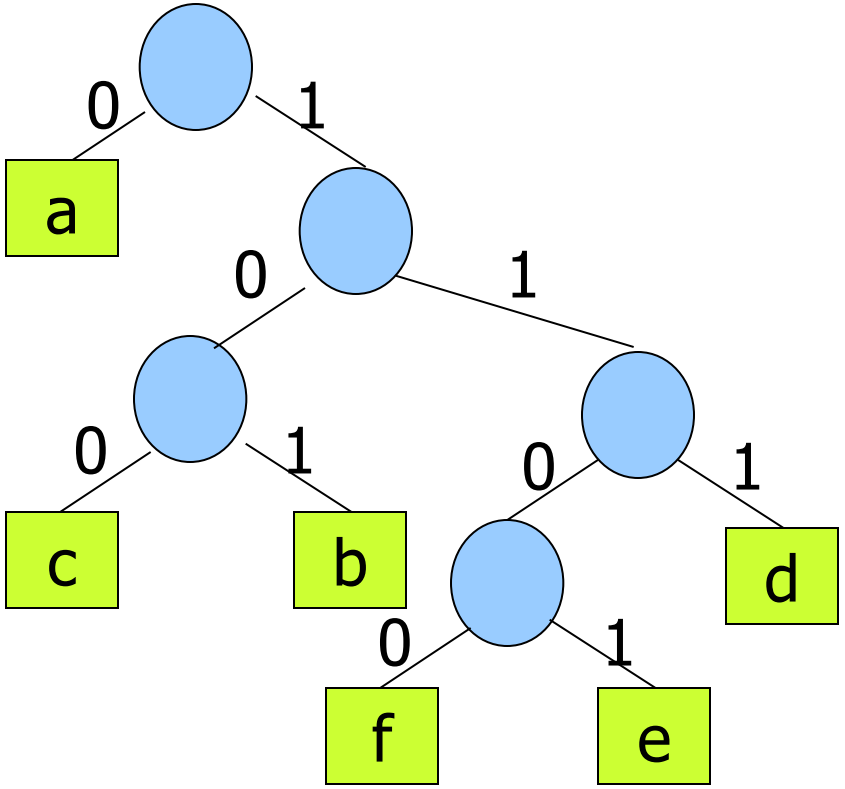
b = 101

c = 100

d = 111

e = 1101

f = 1100



Esempio

Codifica:

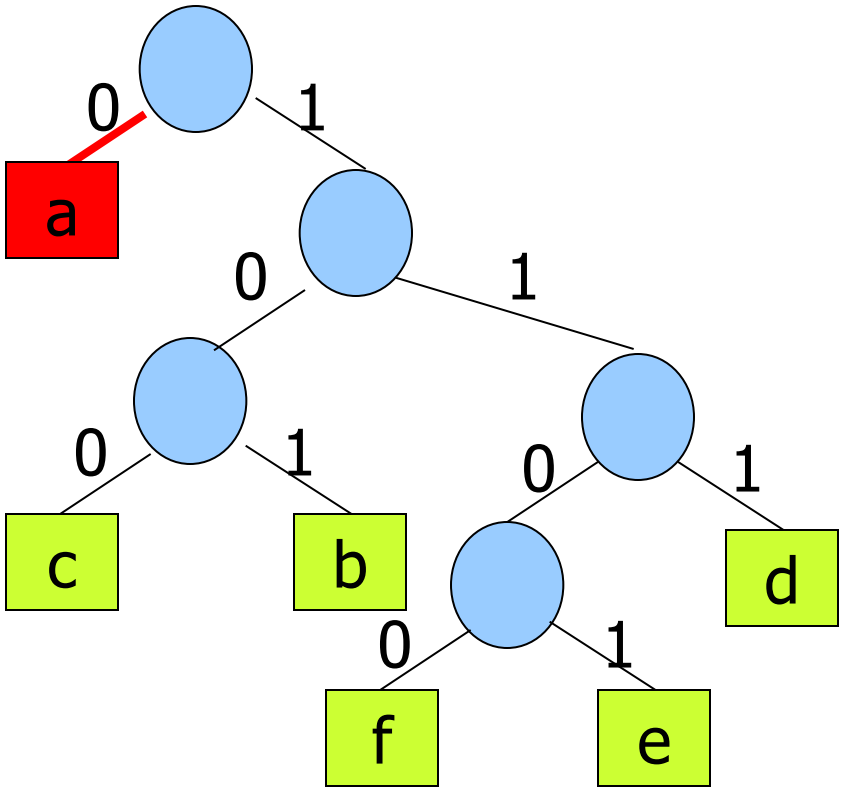
a b f a a c

0101110000100

Decodifica:

0101110000100

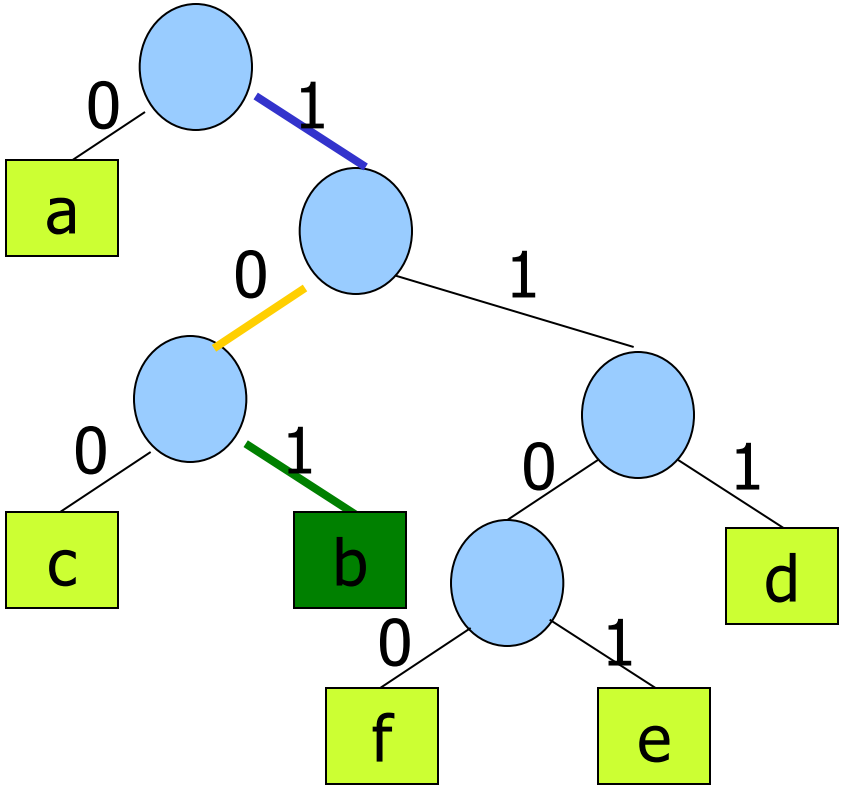
a



Decodifica:

101110000100

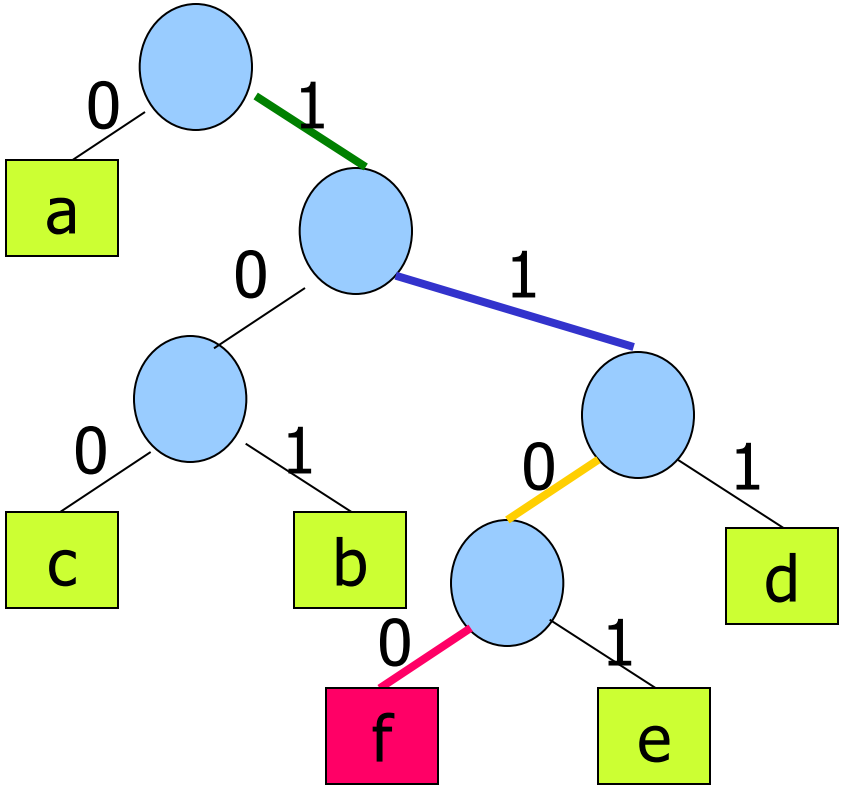
ab



Decodifica:

110000100

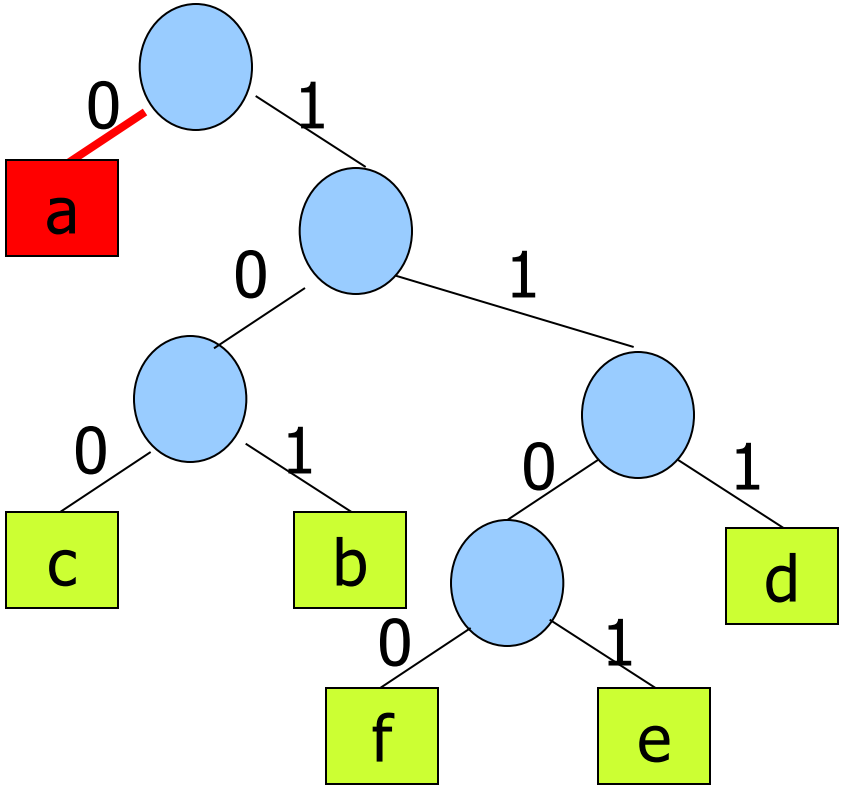
abf



Decodifica:

00100

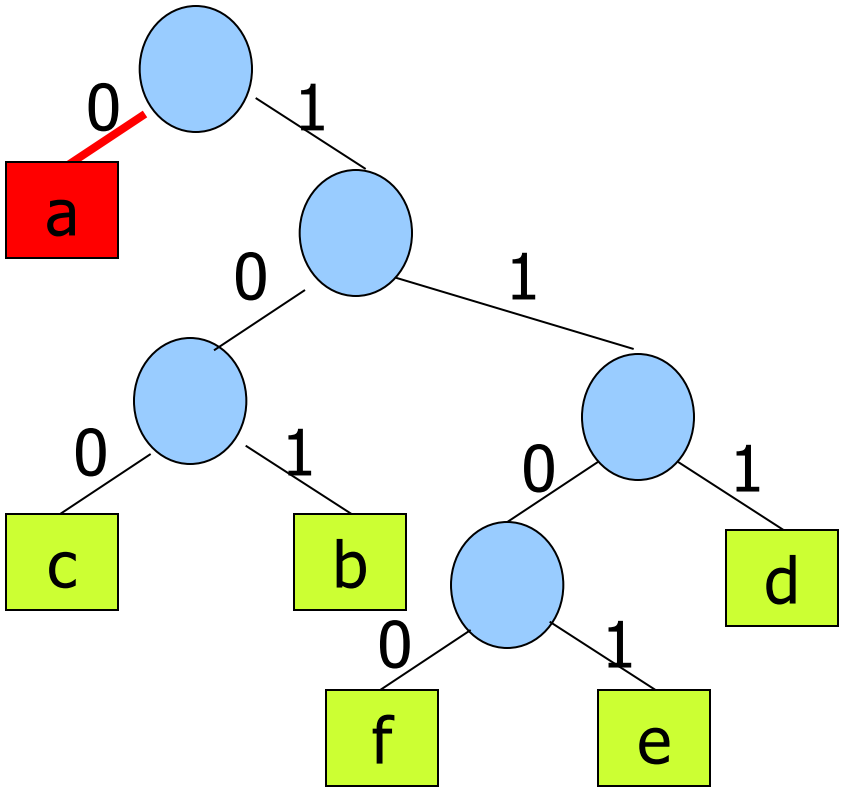
abfa



Decodifica:

0100

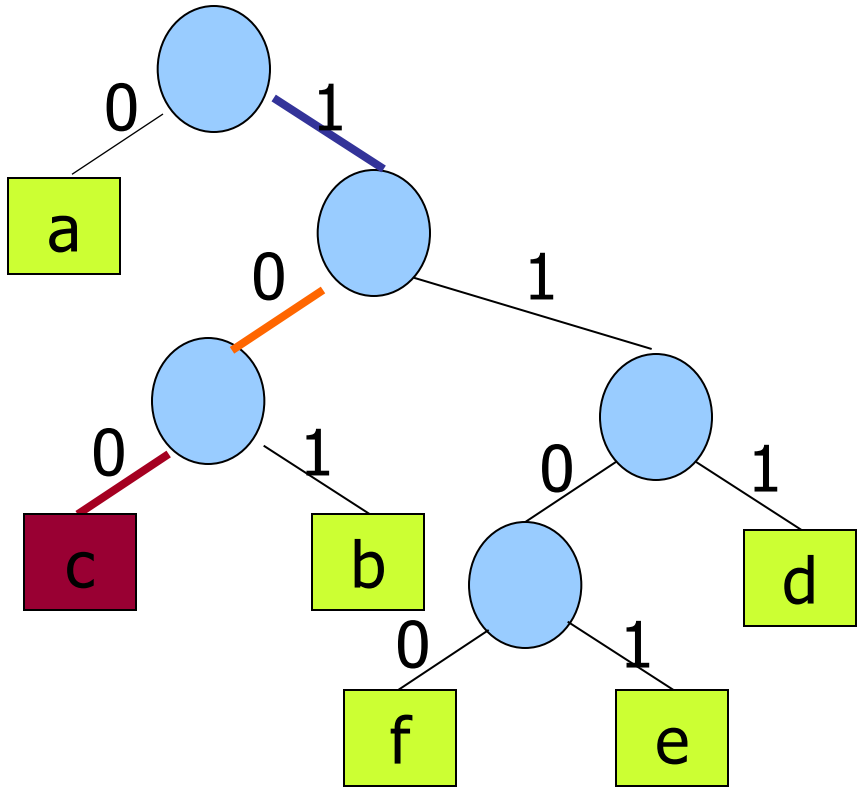
abfaa

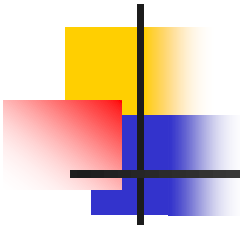


Decodifica:

100

abfaac



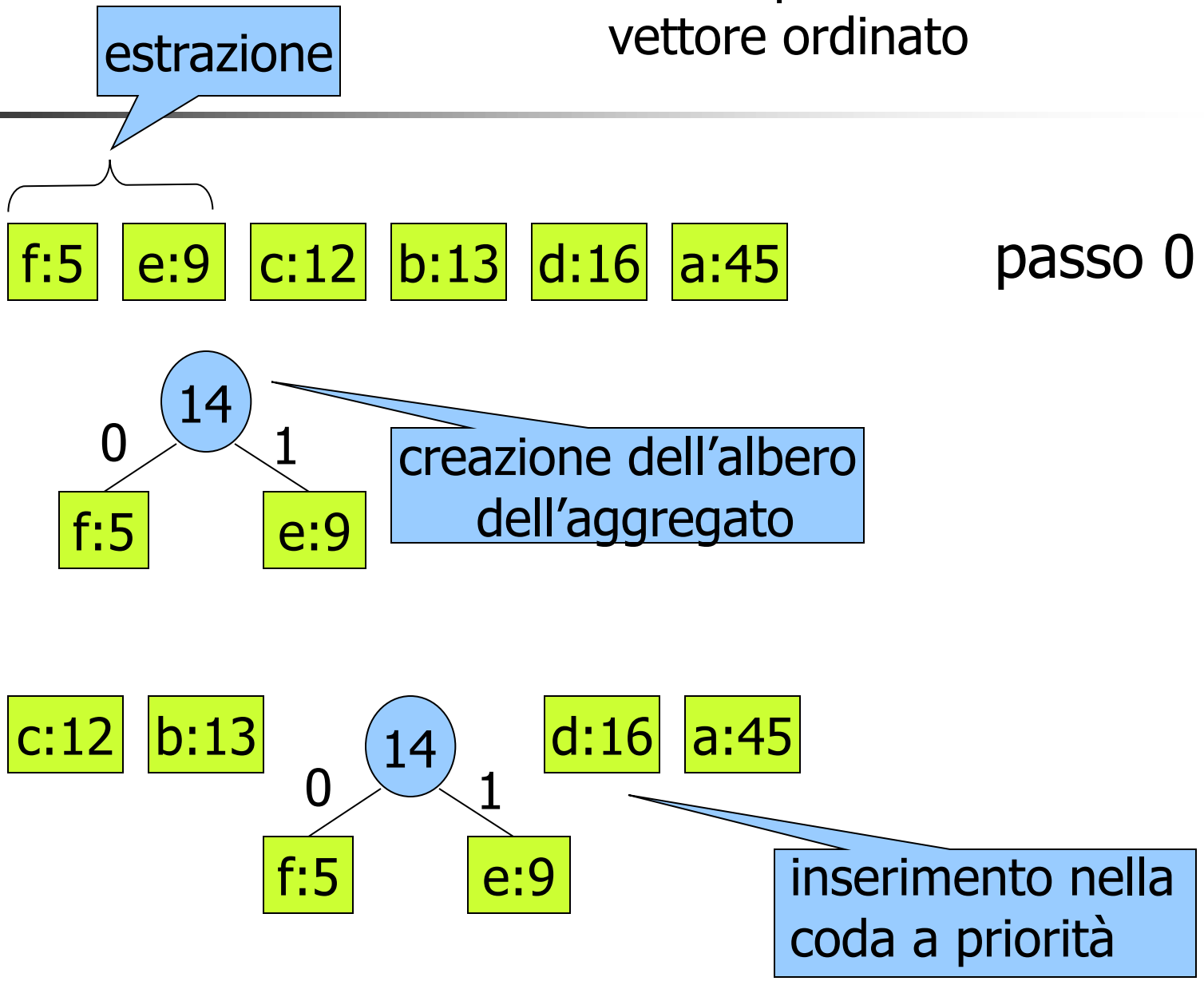


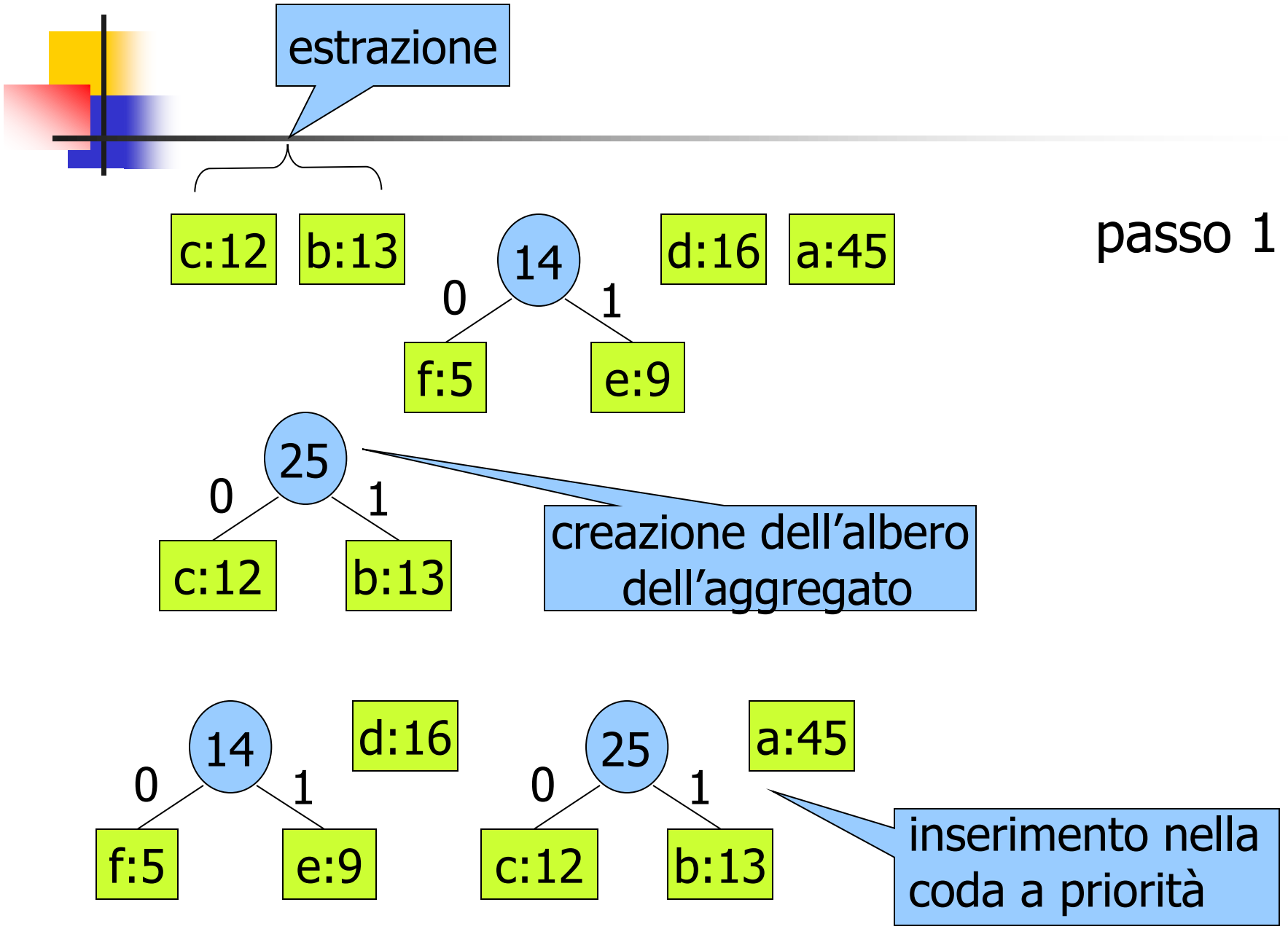
Costruzione dell'albero binario

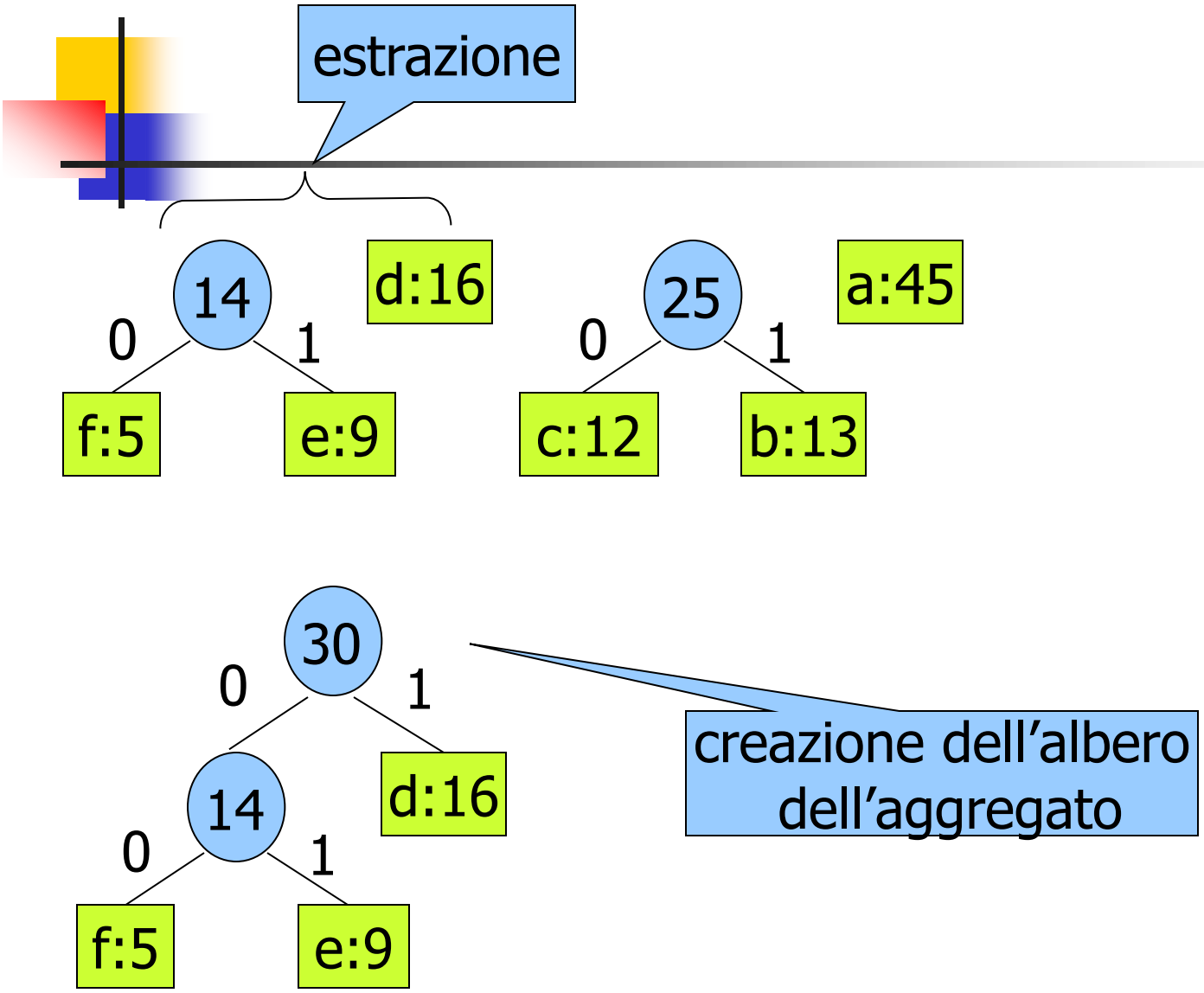
Struttura dati: coda a priorità (frequenze crescenti, proprietà greedy)

- Inizialmente: simbolo = foglia
- Passo i-esimo:
  - estrazione dei 2 simboli (o aggregati) a minor frequenza
  - costruzione dell'albero binario (aggregato di simboli): nodo = simbolo o aggregato, frequenza = somma delle frequenze
  - inserimento nella coda a priorità
- Terminazione: coda vuota.

coda a priorità come vettore ordinato

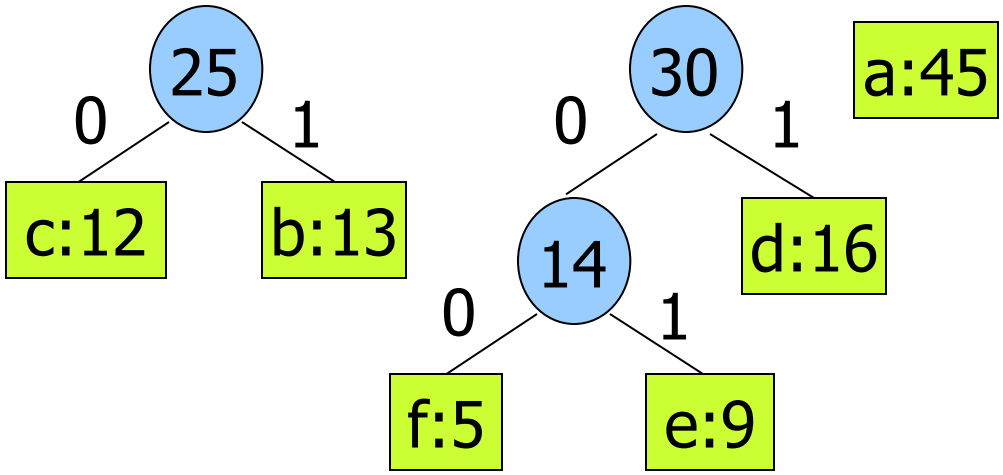


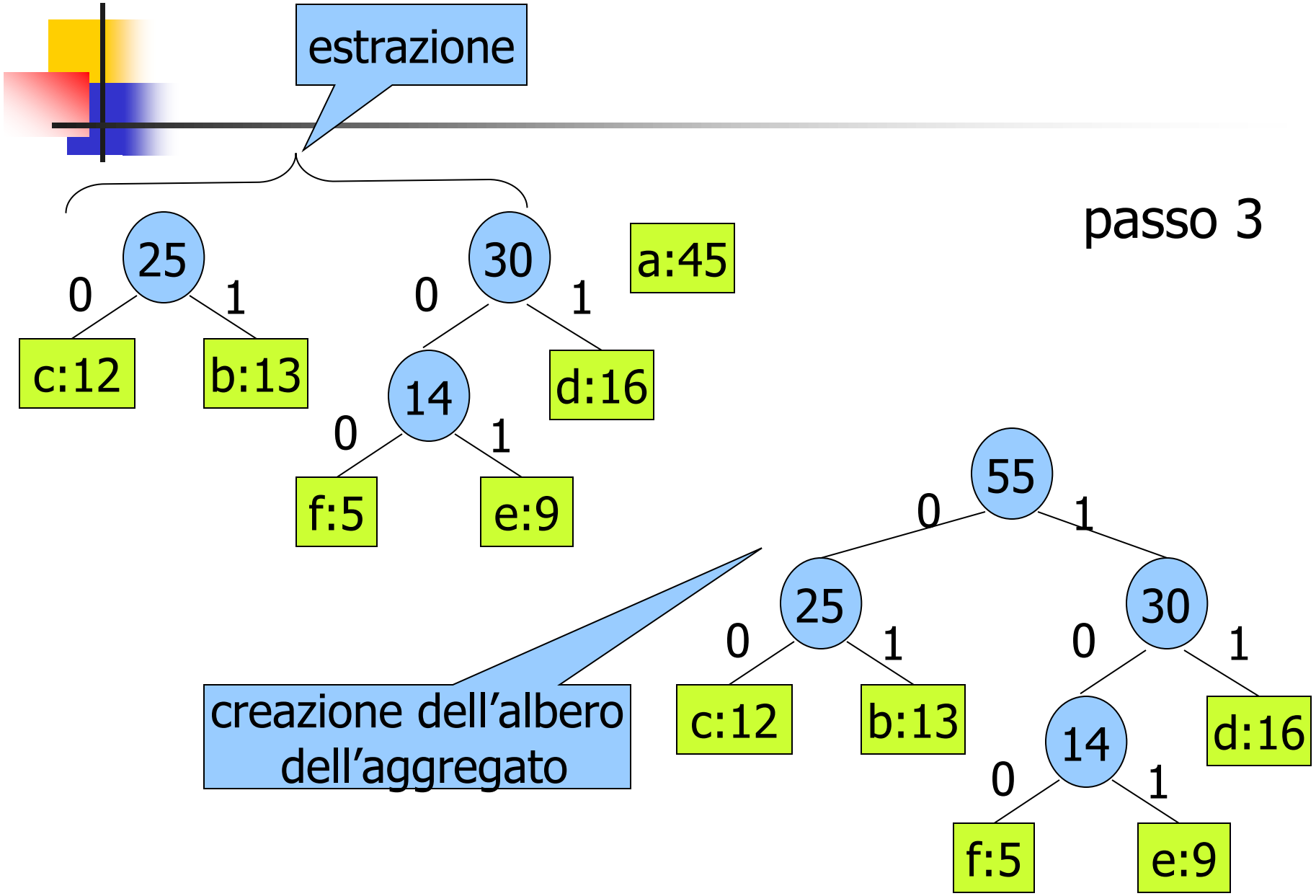

estrazione
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
passo 1
25
0
1
c:12
b:13
creazione dell'albero dell'aggregato
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
inserimento nella coda a priorità

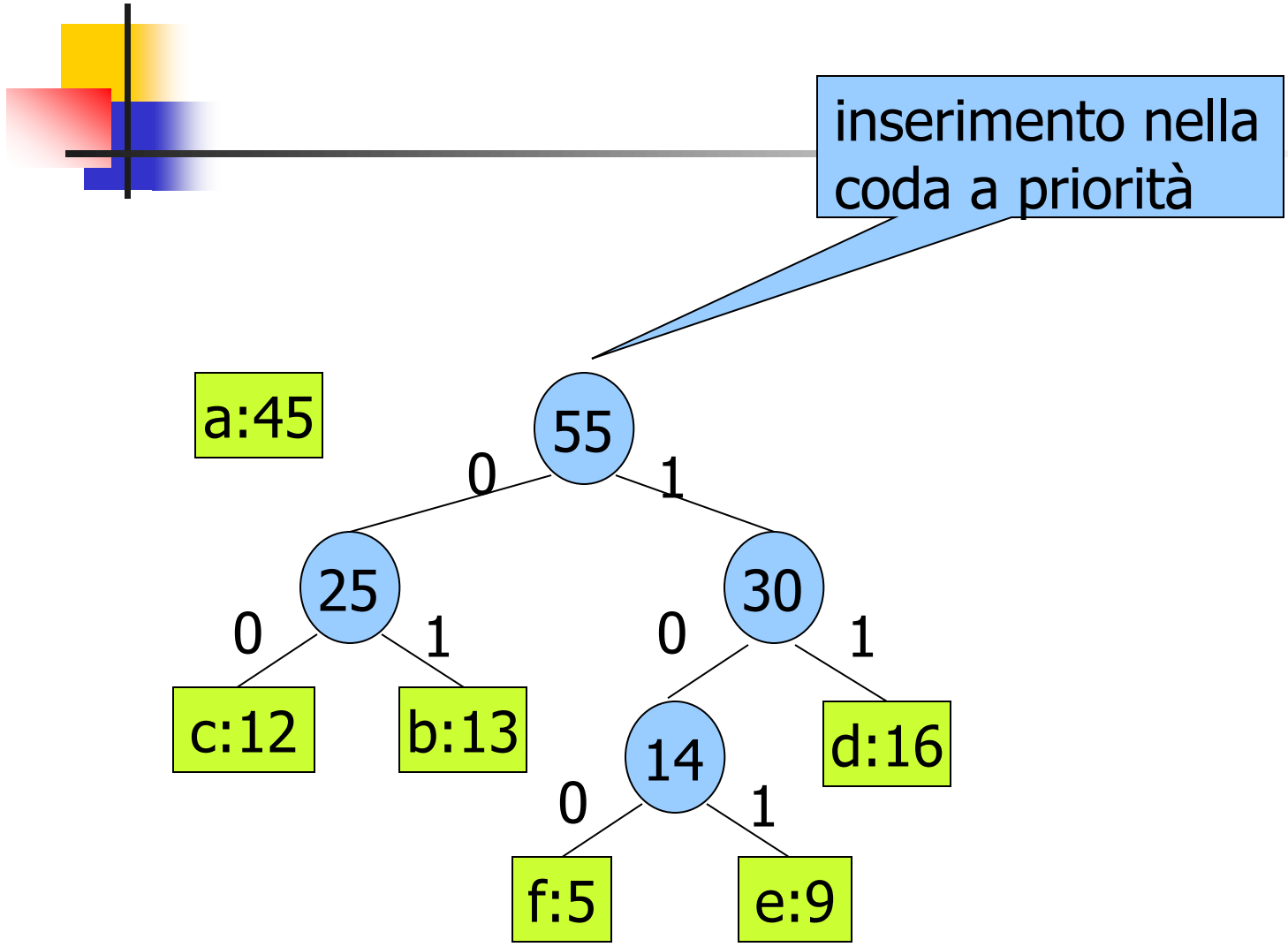

estrazione
14
d:16
0
1
f:5
e:9
25
a:45
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
creazione dell'albero
dell'aggregato

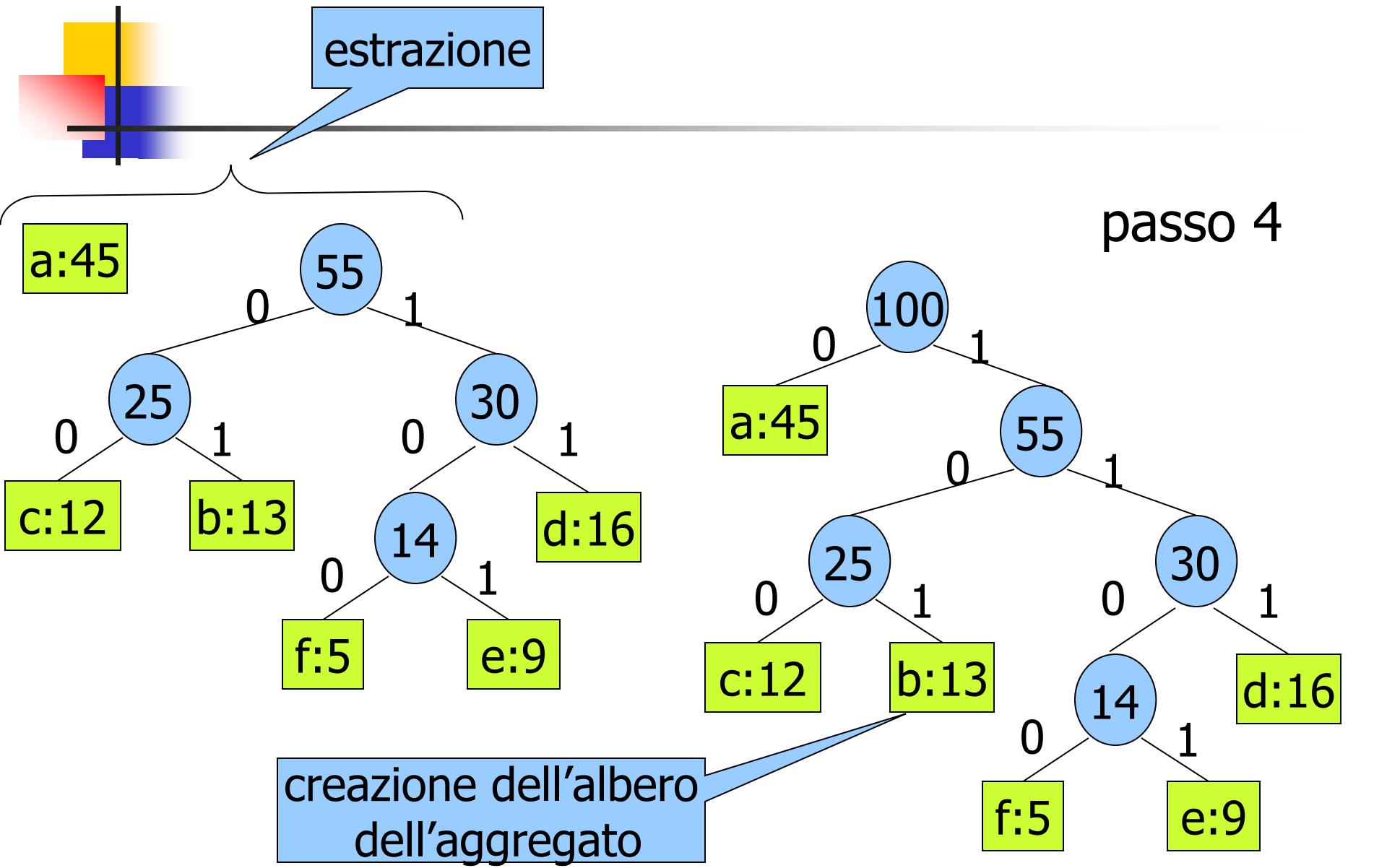
passo 2

inserimento nella coda a priorità

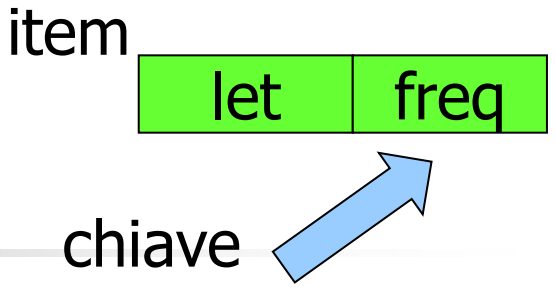



estrazione
passo 3
25
30
a:45
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9
creazione dell'albero
dell'aggregato

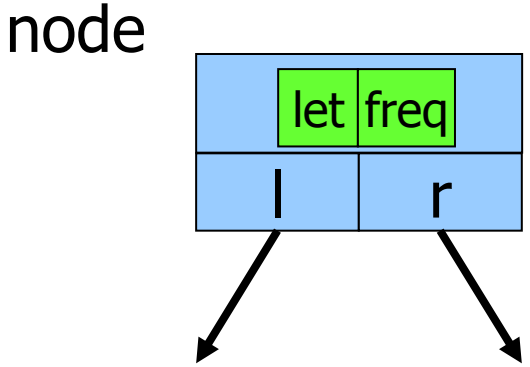

inserimento nella coda a priorità
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9


estrazione
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9
passo 4
100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9
creazione dell'albero
dell'aggregato

Implementazione



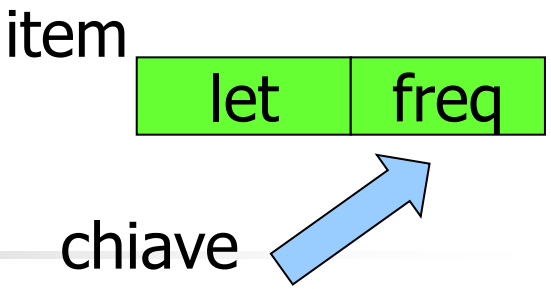
- ADT di I cat. Item per i dati (lettera e frequenza). La chiave è la frequenza, ma è necessario poter accedere alla lettera per visualizzarla
- ADT I cat. PQ che contiene i puntatori alla radice dei diversi alberi.



ADT di I categoria Item

Item.h

```
#define MAXC 3

typedef struct item* Item;
typedef int Key;
typedef char* Let;

Item ITEMscan();
int ITEMcheckvoid(Item data);
Item ITEMsetvoid();
int ITEMgreater(Item data1, Item data2);
int ITEMless(Item data1, Item data2);
Item ITEMnew(Item data1, Item data2);
int KEYcompare(Key k1, Key k2);
Key KEYget(Item data);
Let LETget(Item data);
```



Item.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Item.h"
struct item { char *let; int freq; };
Item ITEMscan() {
  char let[MAXC];
  int freq;
  printf("letter = "); scanf("%s", let);
  printf("frequency = "); scanf("%d", &freq);
  Item tmp = (Item) malloc(sizeof(struct item));
  if (tmp == NULL) return ITEMsetvoid();
  else {
    tmp->let = strdup(let);
    tmp->freq = freq;
  }
  return tmp;
}
```

```
Item ITEMnew(Item data1, Item data2) {
  char let[MAXC]="";
  Item tmp = (Item) malloc(sizeof(struct item));
  if (tmp == NULL) return ITEMsetvoid();
  else {
    tmp->let = strdup(let);
    tmp->freq = data1->freq + data2->freq;
  }
  return tmp;
}

int ITEMcheckvoid(Item data) {
  Key k1, k2=-1;
  k1 = KEYget(data);
  if (KEYcompare(k1,k2)==0) return 1;
  else return 0;
}
```

```
Item ITEMsetvoid() {
  char let[MAXC]="";
  Item tmp = (Item) malloc(sizeof(struct item));
  if (tmp != NULL) {
    tmp->let = strdup(let);
    tmp->freq = -1;
  }
  return tmp;
}
int ITEMgreater (Item data1, Item data2) {
  Key k1 = KEYget(data1), k2 = KEYget(data2);
  if (KEYcompare(k1, k2) == 1) return 1;
  else return 0;
}
int ITEMless (Item data1, Item data2) {
  Key k1 = KEYget(data1), k2 = KEYget(data2);
  if (KEYcompare(k1, k2) == -1) return 1;
  else return 0;
}
```

```
int KEYcompare(Key k1, Key k2) {
  if (k1 < k2)
    return -1;
  else if (k1 == k2)
    return 0;
  else
    return 1;
}

Key KEYget(Item data) {
  return data->freq;
}

Let LETget(Item data) {
  return data->let;
}
```

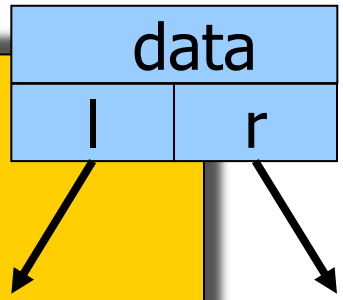
ADT di I categoria Coda a Priorità

PQ.h

```
typedef struct node* link;

typedef struct pqueue *PQ;

PQ      PQinit(int);
int     PQempty(PQ);
void    PQinsert(PQ, link);
link    PQextractMin(PQ);
int     PQsize(PQ);
Item    NODEextract(link);
link    NEW(Item, link, link);
void    CODEdisplay(link char*, int);
```

PQ.c

```
#include <stdlib.h>
#include <stdio.h>
#include "Item.h"
#include "PQ.h"
#define LEFT(i)     ((i*2) + 1)
#define RIGHT(i)    ((i*2) + 2)
#define PARENT(i)   ((i-1) / 2)
struct node { Item data; link l; link r; } ;
struct pqueue { link *array; int heapsize; };
link NEW(Item data, link l, link r) {
  link node = malloc(sizeof *node);
  node->data = data;   node->l = l; node->r = r;
  return node;
}
PQ PQinit(int maxN){
  int i;  PQ pq = malloc(sizeof(*pq));
  pq->array = (link *)malloc(maxN*sizeof(link));
  for (i=0; i< maxN; i++)
    pq->array[i] = NEW(ITEMsetvoid(), NULL, NULL);
  pq->heapsize = 0;
  return pq;
}
```



```
int PQempty(PQ pq) { return pq->heapsize == 0; }

int PQsize(PQ pq) { return pq->heapsize; }

void PQinsert (PQ pq, link node) {
  int  i  = pq->heapsize++;
  while((i>=1) &&
        (ITEMgreater(pq->array[PARENT(i)]->data,
                     node->data)) ) {
    pq->array[i] = pq->array[PARENT(i)];
    i = (i-1)/2;
  }
  pq->array[i] = node;
  return;
}

void Swap(PQ pq, int n1, int n2) {
  link temp = pq->array[n1];
  pq->array[n1]=pq->array[n2];  pq->array[n2]=temp;
  return;
}
```

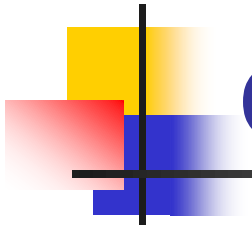
```
void HEAPify(PQ pq, int i) {
  int l, r, largest;
  l = LEFT(i);
  r = RIGHT(i);
  if ( (l < pq->heapsize) &&
       (ITEMless(pq->array[l]->data,
                 pq->array[i]->data)) )
    largest = l;
  else
    largest = i;
  if ( (r < pq->heapsize) &&
       (ITEMless(pq->array[r]->data,
                 pq->array[largest]->data)))
    largest = r;
  if (largest != i) {
    Swap(pq, i,largest);
    HEAPify(pq, largest);
  }
  return;
}
```

```
link PQextractMin(PQ pq) {
  link node = malloc(sizeof(node));
  Swap (pq, 0,pq->heapsize-1);
  node = pq->array[pq->heapsize-1];
  pq->heapsize--;
  HEAPify(pq, 0);
  return node;
}
Item NODEextract(link node) { return node->data; }
void CODEdisplay(link n, char *code, int depth) {
  if (n->l==NULL && n->r==NULL) {
    code[depth] = '\0';
    printf("%s : %s\n", LETget(NODEextract(n)), code);
    return;
  }
  code[depth] = '0';
  CODEdisplay(n->l, code, depth+1);
  code[depth] = '1';
  CODEdisplay(n->r, code, depth+1);
}
```

client.c

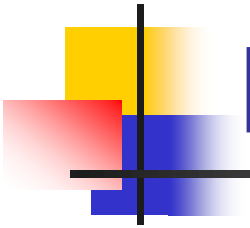
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Item.h"
#include "PQ.h"

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Error: missing argument\n");
    printf("correct format:\n");
    printf("%s maxN \n", argv[0]);
    return 0;
  }
  int i, maxN = atoi(argv[1]);
  link node, nodeL, nodeR, root;
  Item item;
  char code[maxN+1];
```

```
  PQ pq = PQinit(maxN);

  for (i=0; i<maxN; i++) {
    node = NEW(ITEMscan(), NULL, NULL);
    PQinsert(pq, node);
  }
  while (PQsize(pq) > 1) {
    nodeL = PQextractMin(pq);
    nodeR = PQextractMin(pq);
    item = ITEMnew(NODEextract(nodeL), NODEextract(nodeR));
    node = NEW(item, nodeL, nodeR);
    PQinsert(pq, node);
  }
  root = PQextractMin(pq);

  printf("\nHuffman code is:\n");
  CODEdisplay(root, code, 0);

  return 1;
}
```



Complessità

- Heap implementato come albero binario
- Operazioni di estrazione e inserzione in coda a priorità
- $T(n) = O(n \lg n)$.



Riferimenti

- Selezione di attività:
  - Cormen 17.1
- Problema del ladro e dello zaino:
  - Cormen 17.2
- Codici di Huffman
  - Cormen 17.3