Algoritmi e strutture dati

Codici di Huffman

Memorizzazione dei dati

- **Quando un file viene memorizzato, esso va memorizzato in qualche formato binario**
- **Modo più semplice: memorizzare il codice ASCII per ogni carattere**
  - **Ogni codice ASCII è lungo 1 byte (8 bits)**
  - **Se il nostro file contiene *n* caratteri, la dimensione del file memorizzato sarà *n* byte, o *8n* bit**

Compressione dei dati

- **Spesso è importante trovare un *modo più efficiente per rappresentare i dati* – cioè così da *impiegare meno bit per la rappresentazione***
  - **si risparmia *spazio disco***
  - **si risparmia *tempo per i trasferimenti* (ad esempio in rete)**
- **La conversione però può impiegare più tempo**
  - **In genere è utilizzate solo per la memorizzazione – non quando si lavora effettivamente sul file**
  - **I dettagli della compressione possono essere nascosti all'utente – non è necessario che sappia se sta leggendo un file compresso o meno**

Rappresentazione della compressione

Un semplice schema di compressione è costituito tramite l'*applicazione di funzioni*.

- **Ogni carattere ha un valore ASCII tra 0 e 255**
- **Ogni carattere necessita di essere rappresentato come un numero (binario) nel file compresso**

- ***Funzione di compressione*: $f(c) = x$**
  - **Il carattere rappresentato dal codice ASCII *c* viene invece rappresentato nel file compresso da *x***

Semplice Compressione

- **Un esempio di *funzione di codifica*:**
  - ***f(‘t’)=11***
  - ***f(‘r’)=01***
  - ***f(‘e’)=10***
- **Per concatenazione delle stringhe di bit possiamo allora calcolare: *f(“tre”)=110110***
- **Ovviamente, con questa codifica viene richiesto *meno spazio* per memorizzare ogni lettera rispetto al caso di utilizzo del codice ASCII di 8-bit**

Unicità della codifica

- **Nella scelta del codice, è importante che ogni carattere venga *mappato* su di un valore differente**
  - **Supponiamo**
    - ***f('a') = 110***
    - ***f('%')= 110***
  - **quando si decomprime, come possiamo sapere quale dei due caratteri è rappresentato da *110*?**
- **In termini matematici, la funzione di codifica deve essere *iniettiva***
  - ***Elementi differenti $x \neq y$ hanno immagini differenti $f(x) \neq f(y)$***

Il problema

É possibile definire uno *schema di compressione* tale che ogni possibile carattere (valore ASCII) venga rappresentato da un *numero* di *bits minore* di *8*?

- **Quanti caratteri ASCII ci sono?**
  - **I caratteri ASCII differenti sono 256**
- **Quanti sono i numero che possiamo rappresentare con al più *7* bits?**
  - **2^7-1 = 127**
- **Ci serve un *valore diverso per ogni carattere* – ma *non* ci sono abbastanza valori differenti rappresentabili con al più *7* bit**
- **Almeno un carattere deve essere rappresentato da *più di 7 bit***
  - **In effetti, *128* caratteri verranno rappresentati con 8 o più bit**

Tipi di codice

- **Codici a *lunghezza fissa*: tutti caratteri sono codificati da stringhe di bit della stessa lunghezza.**
- **Codici a *lunghezza variabile*: i caratteri sono codificati da stringhe di bit la cui lunghezza può variare da carattere a carattere (ad esempio a seconda della *frequenza dei caratteri* nel file).**

***Esempio*:**		***a***	***b***	***c***	***d***	***e***	***f***
frequenze	**:**	**45**	**13**	**12**	**16**	**9**	**5**
lunghezza fissa	**:**	**000**	**001**	**010**	**011**	**100**	**101**
lunghezza variabile	**:**	**0**	**101**	**100**	**111**	**1101**	**1100**

- **fisso: per 100.000 caratteri servono 300.000 bit**
- **variabile: per 100.000 caratteri servono 224.000 bit**

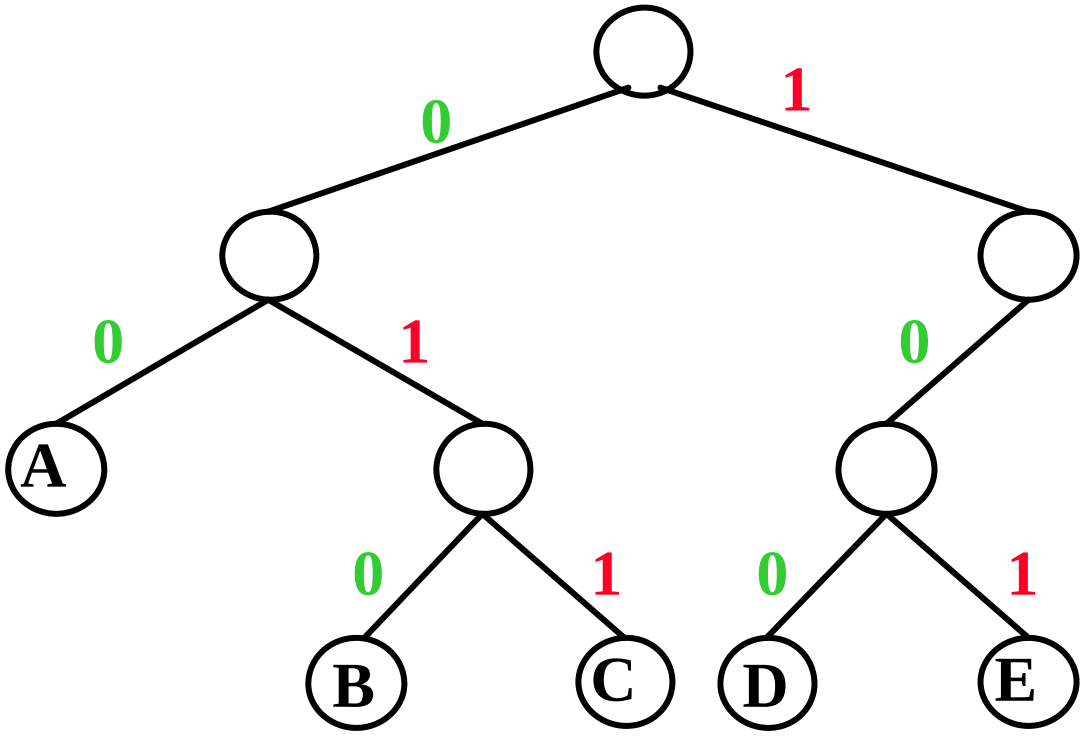
Un problema per la decodifica

- **Supponiamo:**
  - *f(‘a’) = 11*
  - *f(‘b’) = 01*
  - *f(‘x’) = 1101*
- **Ora supponiamo che si tenti di decomprimere il file contenente *1101***
  - **Questo va tradotto in “*ab*” o in ‘*x*’?**
- **Questo problema si verifica perché questo è un *codice a lunghezza variabile***
  - **ASCII è un *codice a lunghezza fissa* ...**
  - **… e sappiamo che ogni 8 bit inizia un *nuovo carattere***

Codici a prefisso

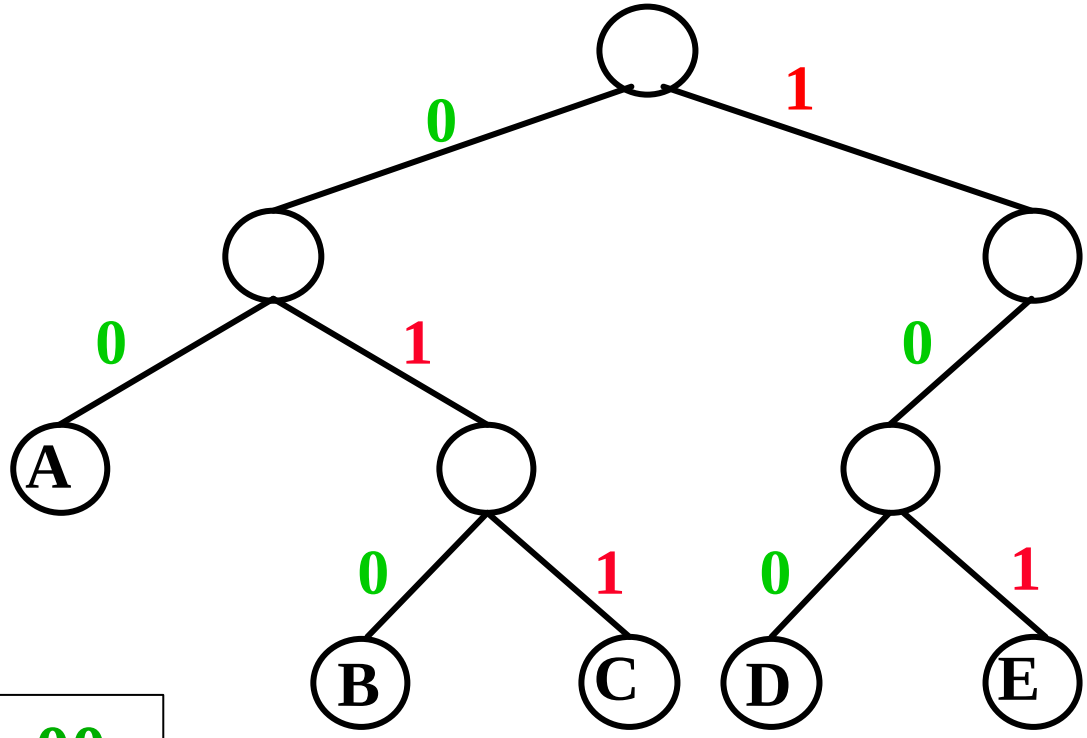
- **Un *codice a prefisso* è un codice in cui *nessuna codifica* è *prefisso* per una qualsiasi altra codifica**
  - **Se *f(‘a’)=110*, allora *nessuna altra codifica inizia* con *110***
- **In questo caso trovare le interruzioni tra caratteri diventa facile:**
  1. **Si leggono i bit finché essi formano una codifica legittima per un carattere**
  2. **Li si trasforma nel carattere corrispondente e si ricomincia a leggere il codice del carattere successivo**
- **I *codici a prefisso* possono essere rappresentati come *alberi binari***

Rappresentazione di codici a prefisso



- **Ogni *foglia* rappresenta un *carattere***
- **Ogni *arco sinistro* è etichettato con *0*, e ogni *arco destro* è etichettato con *1***
- **Il *codice di ogni foglia* è l'*etichetta del percorso* che la raggiunge a partire dalla radice**

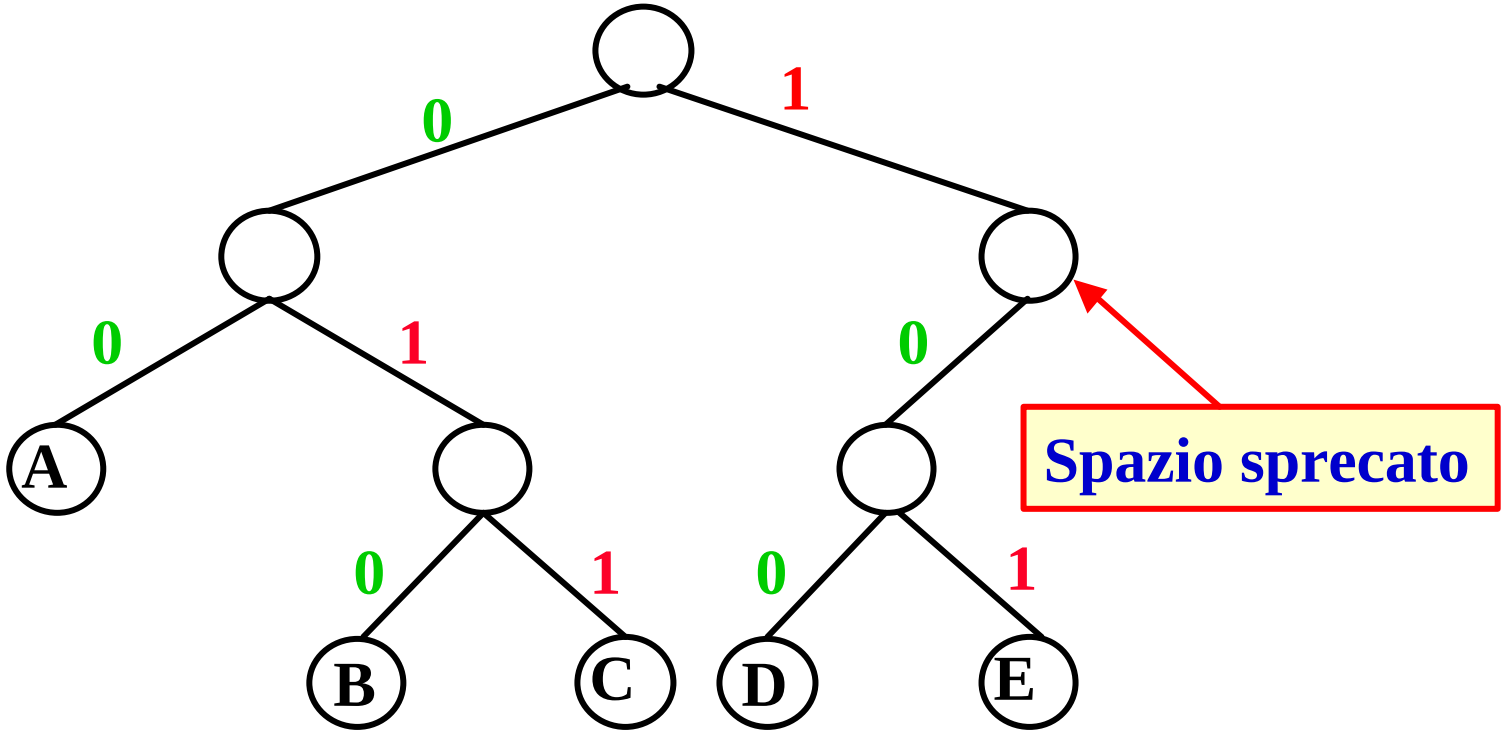
Rappresentazione di codici a prefisso



A: 00
B: 010
C: 011
D: 100
E: 101

Poiché ogni carattere corrisponde a una foglia, questo deve essere un *codice a prefisso*

Rappresentazione di codici a prefisso

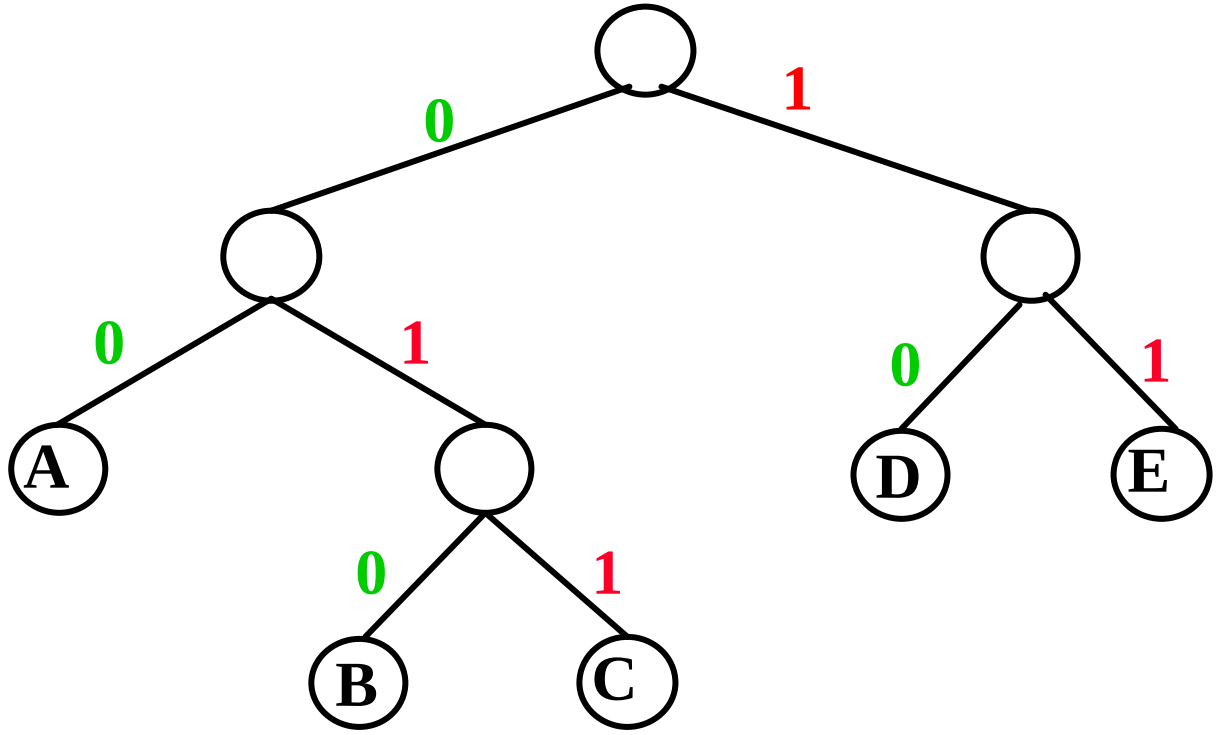


Non c’è motivo di avere un nodo non figlia con un solo figlio

Rappresentazione di codici a prefisso

A: 00
B: 010
C: 011
D: 100
E: 101

A: 00
B: 010
C: 011
D: 10
E: 11



Possiamo quindi assumere che ogni nodo interno abbia due figli.
In altre parole, l'*albero è completo*

Codici ottimi

***Definizione*: Dato un file *F*, diciamo che un *codice C è ottimo* per *F* se non esiste un altro codice tramite il quale *F* possa essere compresso impiegando un numero inferiore di bits.**

- ***Nota*: Esiste in genere *più di un codice ottimo* per un dato file - cioè il *codice ottimo per F non è unico***

***Teorema*: Dato in file *F*, esiste un codice ottimo *C* che è un *codice a prefisso* (che può cioè essere rappresentato come un *albero completo*)**

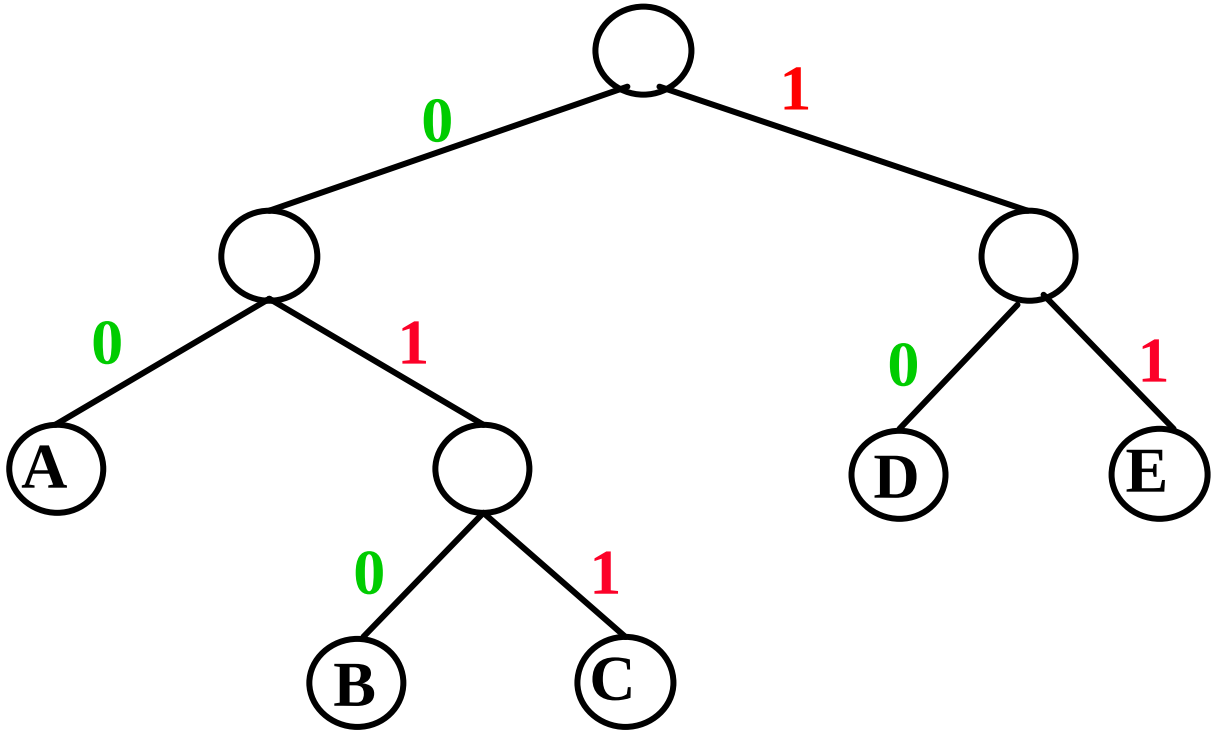
Dimensione del file compresso

- **Supponiamo di avere:**
  - **un file composto da caratteri nell'alfabeto S**
  - **un albero T che rappresenta la codifica**
- **Quanti bit sono richiesto per codificare il file?**
  - **Per ogni c Î S, sia $d_T(c)$ la profondità in T della foglia che rappresenta c**
  - **Il codice per c richiederà allora $d_T(c)$ bits**
  - **Se $f(c)$ è il numero di volte che c occorre nel file, allora la dimensione della codifice è**

$$f(c_1)\, d_T(c_1) + f(c_2)\, d_T(c_2) + \ldots$$

sommatoria su tutti i caratteri c Î S

Esempio



Un file con 7 A, 3 B, 6 C, 2 D e 8 E richiederà

7(2) + 3(3) + 6(3) + 2(2) + 8(2) = 61 bit

***Nota*: Se avessimo scambiato i nodi C and D, avremmo avuto bisogno di soli 57 bit**

Progettazione di un codice

- **Quando si progetta un codice per un file:**
  - ***Minimizzare*** **la lunghezza dei caratteri che compaiono *più frequentemente***
  - **Assegnare ai caratteri con la *frequenza minore* i codici corrispondenti ai *percorsi più lunghi* all'interno dell'albero**
- **Questo è il *principio* sottostante i *codici di Huffman*!!!**

Codici di Huffman

- **Un codice (generalmente) è progettato per un file specifico**
  - **Costruito da un algoritmo specifico**
- **L'*albero binario "completo"* che rappresenta il codice creato dall'algoritmo è chiamato *albero di Huffman***

Costruzione del codice

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

Passo 1: Costruire un nodo foglia per ogni lettera, etichettato con la frequenza della lettera nel file

Costruzione del codice

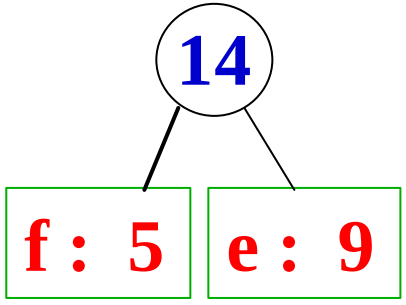
c : 12	b : 13	d : 16	a : 45

f : 5	e : 9

Passo 2: Rimuovere i due nodi “*più piccoli*” (con frequenze minori)

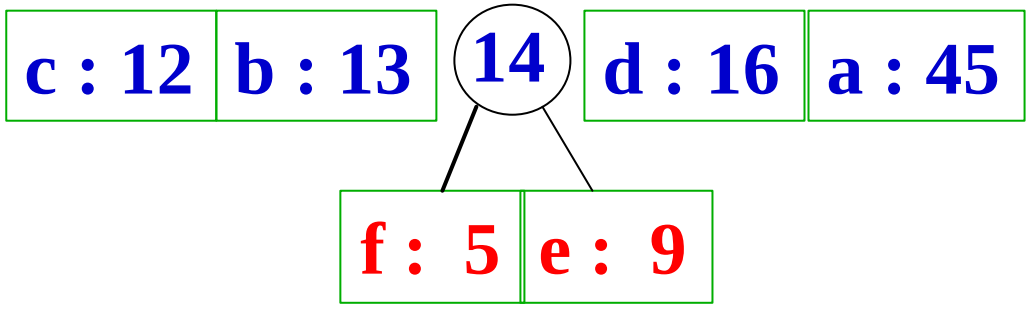
Costruzione del codice

c : 12	b : 13	d : 16	a : 45



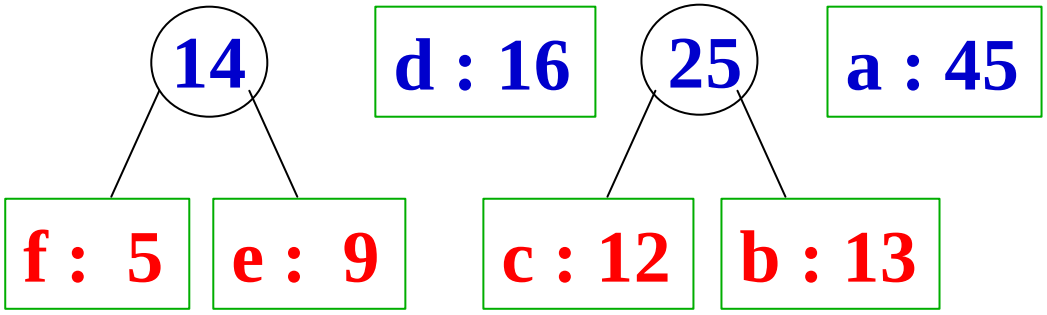
Passo 3: Collegarli ad un nodo padre etichettato con la frequenza combinata (sommata)

Costruzione del codice



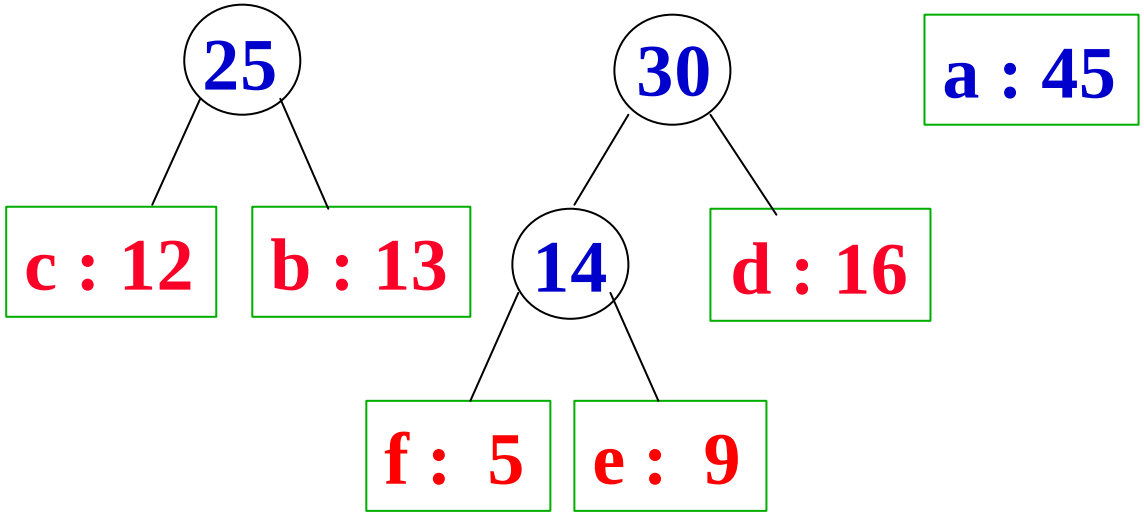
Passo 4: Aggiungere il nodo alla lista.

Costruzione del codice



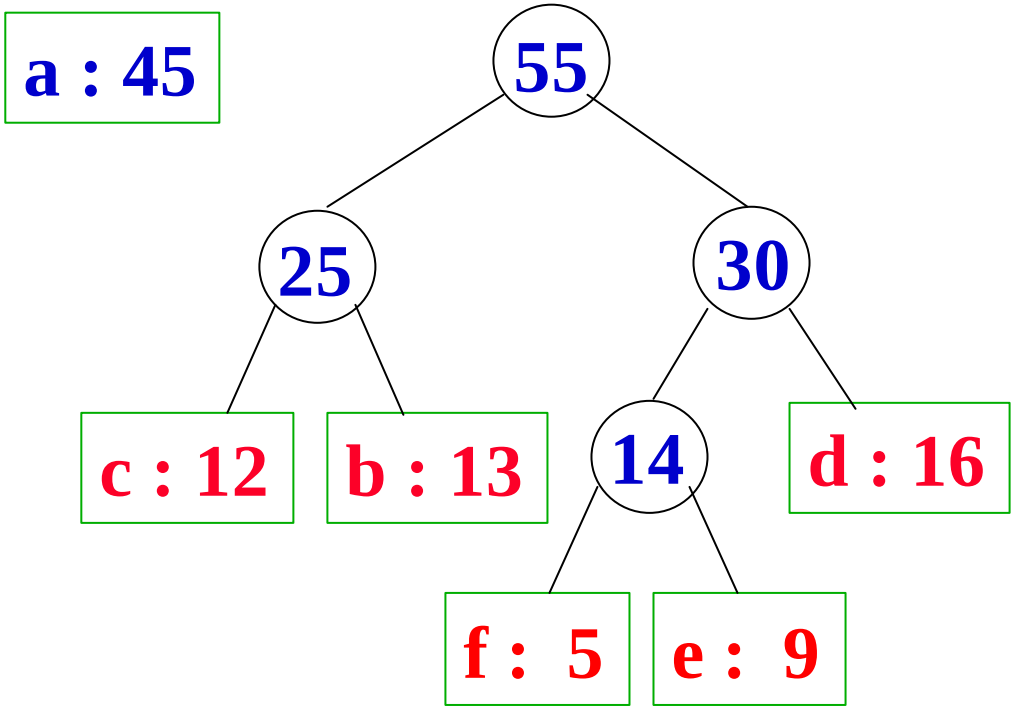
Passo 5: Ripetere i passi 2-4 finché resta un solo nodo nella lista

Costruzione del codice



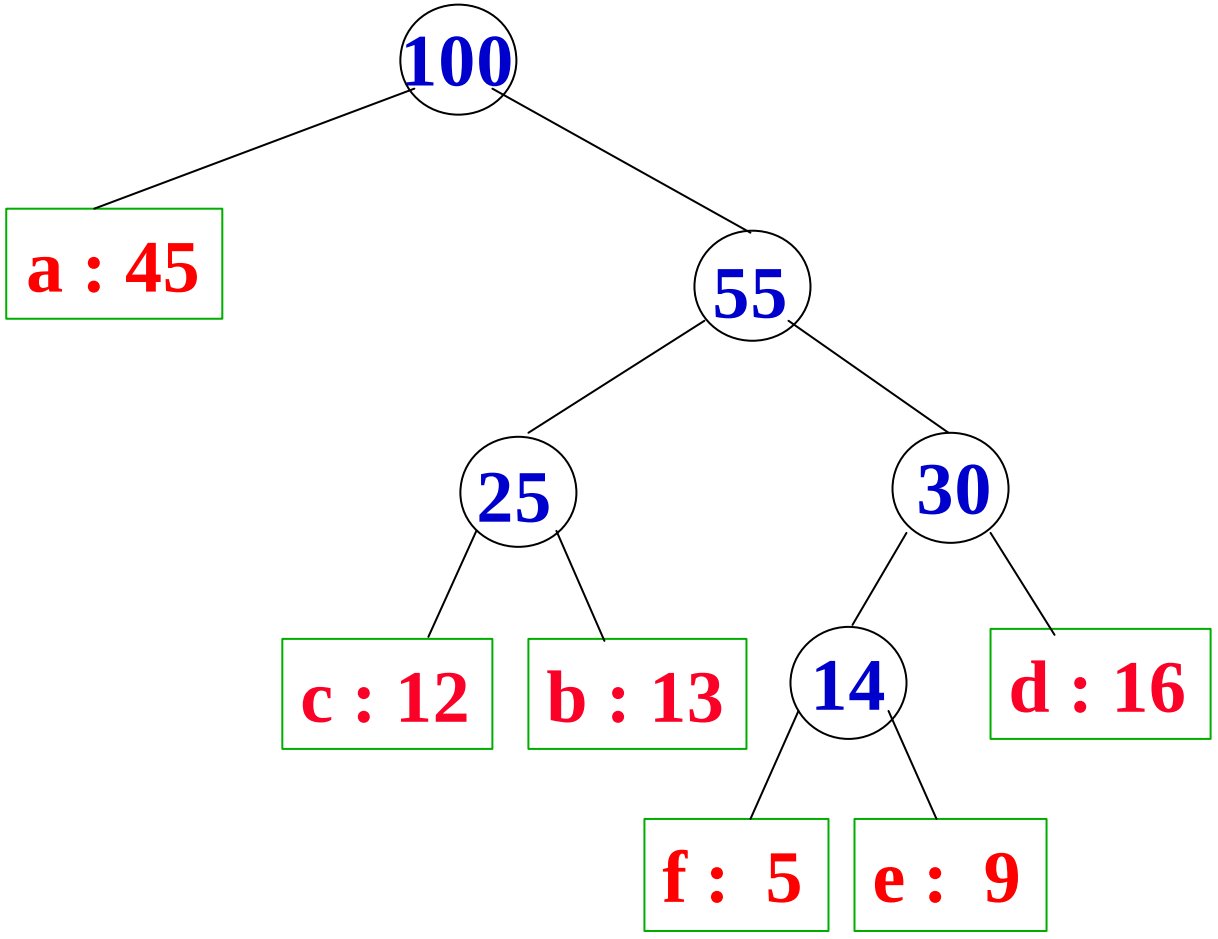
Passo 3: Continua finché resta un solo nodo nella lista

Costruzione del codice



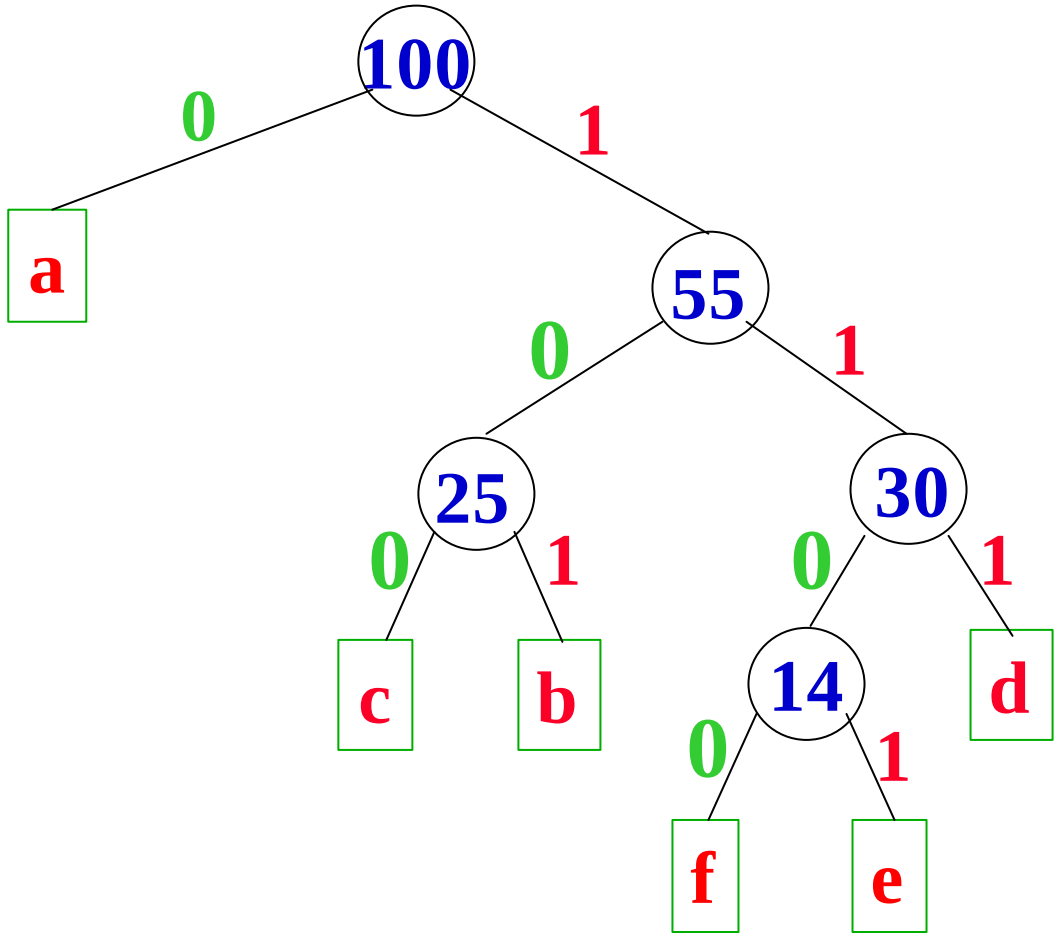
Passo 3: Continua finché resta un solo nodo nella lista

Costruzione del codice



Passo 3: Continua finché resta un solo nodo nella lista

Costruzione del codice



Gli archi dei percorsi dalla radice alle foglie presi in sequenza ci danno i codici associati ai caratteri

Algoritmo di Huffman

```
Huffman(A: array; n: intero)
  Q: Priority_Queue;
  for i = 1 to n
       Insert(Q,A[i])
  for i = 1 to n - 1
     x = Extract-Min(Q)
     y = Extract-Min(Q)
     "crea un nuovo albero con radice z
      e valore Freq[x] + Freq[y] e con
      figli destro x e sinistro y"
     Insert(Q,z)
  return Extract-Min(Q)
```

Tempo di esecuzione

- **Con una *coda a priorità* basata su *lista*:**
  - **Primo ciclo:**
    - **Esegue una operazione di accodamento per ognuna delle *n* iterazioni – *n O(1)***
  - **Secondo ciclo:**
    - **Esegue due operazione di estrazione dalla coda e una di accodamento per ognuna delle *n-1* iterazioni – *(n-1) O(n)***
  - **Totale: $O(n^2)$**
- **Con una *coda a priorità* basata su *heap*:**
  - **Totale: *O(n log n). Perché?***

Codici di Huffman: correttezza

Teorema: **Il *codice di Huffman* per in dato file è un *codice a prefisso ottimo***

- **Schema della *dimostrazione*:**
  - **Sia *C* un qualsiasi codice a prefisso ottimo per il file**
  - **Mostriamo che vale la *proprietà di sottostruttura ottima***
  - **Mostriamo che che vale la proprietà della *scelta greedy***
- **Cioè il *codice di Huffman* prodotto dall'algoritmo comprime il file tanto quanto un *codice ottimo C***

Codici di Huffman: correttezza

Lemma 1: **Sia** T **un** ***albero binario completo*** **che rappresenta un** ***codice a prefisso ottimo*** **su un alfabeto** S**, con frequenza** $f(c)$ **per ogni** $c \in S$**. Siano** x**,** y **le due foglie figlie di** z **in** T**.**

Se eliminiamo x **e** y **da** T **e consideriamo** z **un carattere con frequenza** $f(z) = f(x) + f(y)$**, otteniamo un nuovo albero** $T' = T - \{x,y\}$**.**

T' **rappresenta un** ***codice a prefisso ottimo*** **per l'alfabeto** $S' = S - \{x,y\} \cup \{z\}$**.**

Proprietà di sottostruttura ottima

Codici di Huffman: correttezza

Dimostrazione L1: **Per ogni carattere $c \in S - \{x,y\}$ sappiamo che $d_T(c) = d_{T'}(c)$, quindi anche che $f(c)d_T(c) = f(c)d_{T'}(c)$.**

Inoltre $d_T(x) = d_T(y) = d_{T'}(z)+1$.

Quindi $f(x)d_T(x) + f(y)d_T(y) = [f(x) + f(y)](d_{T'}(z)+1) =$

$$= f(z)d_{T'}(z) + [f(x) + f(y)]$$

Segue che $B(T)=B(T')+ [f(x) + f(y)]$

Ora se T' non fosse ottimo, esisterebbe T'' migliore di T' per $S' = S - \{x,y\} \cup \{z\}$ (cioè tale che $B(T'')<B(T')$).

Codici di Huffman: correttezza

Dimostrazione L1: **Ora se T' non fosse ottimo, esisterebbe T'' migliore di T' per $S'=S - \{x,y\} \cup \{z\}$ (cioè tale che $B(T'')<B(T')$).**

Poiché z è un carattere per S', il nodo per z sarà una foglia di T''.

Ma aggiungendo x e y a T'' come figli di z quello che otteniamo è un codice a prefisso per S.

Il costo del nuovo codice sarebbe

$$B(T'')+[f(x)+f(y)] < B(T)$$

che *contraddice l'ottimalità* di T!

$$B(T)=B(T')+ [f(x) + f(y)]$$

Codici di Huffman: correttezza

Lemma 2: **Sia S un alfabeto ai cui caratteri c sia associata la frequenza $f(c)$. Siano x e y due caratteri di S con le *frequenze minori*.**

Allora esiste un *codice prefisso ottimo* per S nel quale i codici per x e y hanno la *stessa lunghezza* e che *differiscono solo per l'ultimo bit*.

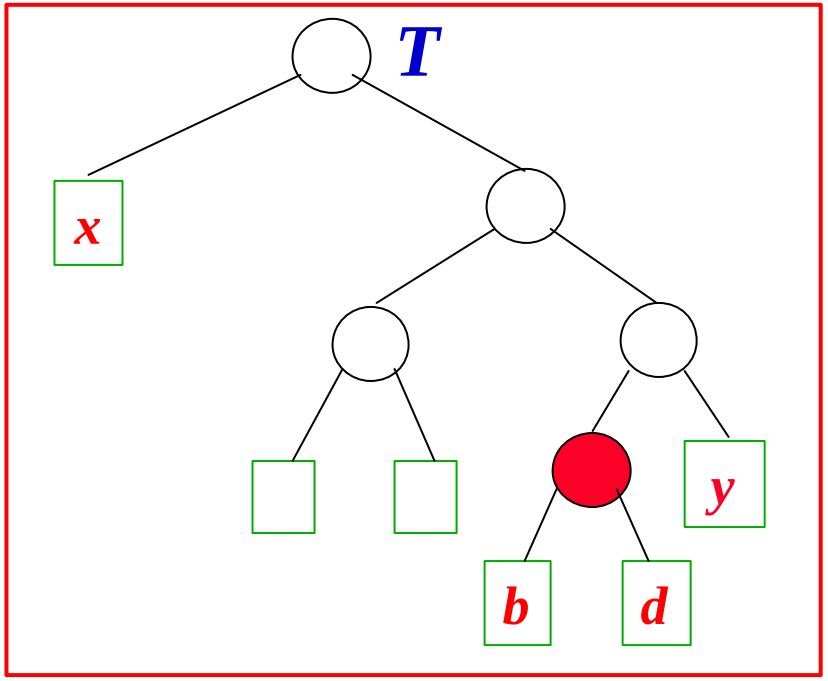
Proprietà della scelta greedy

Codici di Huffman: correttezza

Dimostrazione L2: **Partiamo da un qualunque albero *T* che rappresenta un codice ottimo e ne costruiremo uno diverso che sia ancora ottimo ma in cui i caratteri *x* e *y* siano foglie figlie dello stesso padre e non a profondità massima in *T*.**

Chiamiamo *b* e *d* due foglie di *T* a profondità massima e con lo stesso padre, mentre *x* e *y* saranno due foglie qualsiasi di *T*.

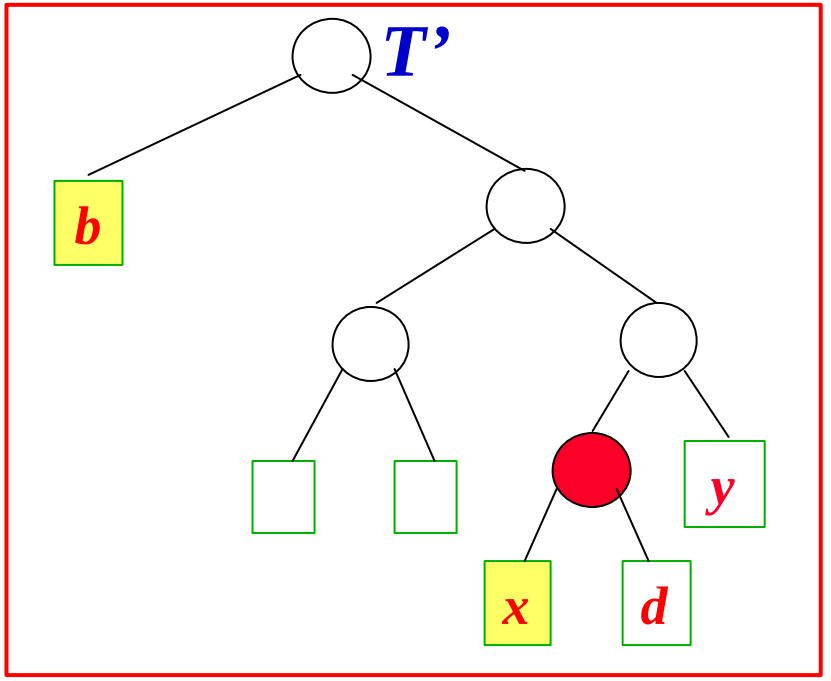
Assumiamo *f(b)* £ *f(d)* e *f(x)* £ *f(y)*



Codici di Huffman: correttezza

Dimostrazione L2: **Per le ipotesi del lemma, *x* e *y* hanno le frequenze minori in assoluto, mentre *b* e *d* sono arbitrarie.**
Quindi varranno anche: *f(x)* £ *f(b)* e *f(y)* £ *f(d)*.
Operiamo la trasformazione in due passi:

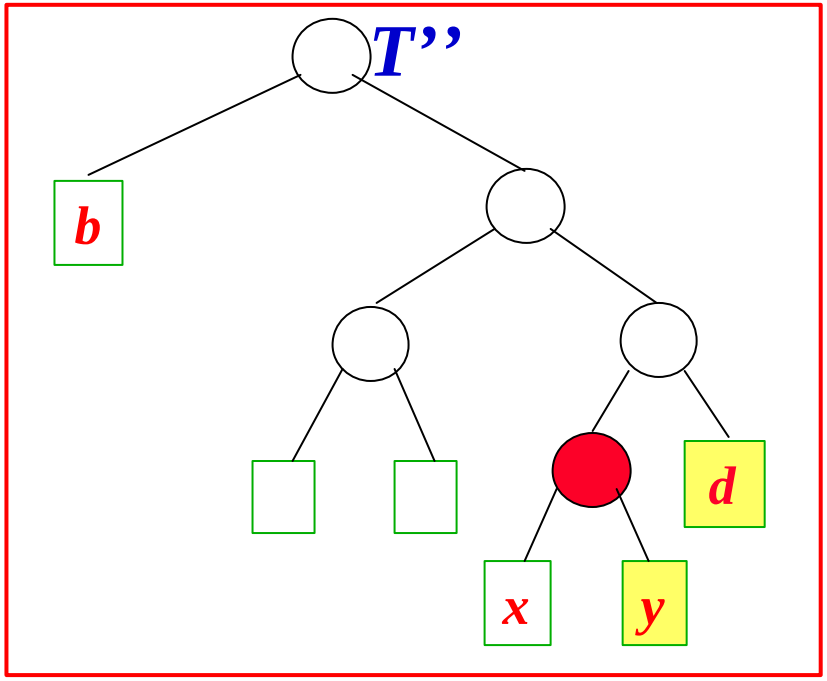
1- Scambiamo la posizione di *x* e *b* ottenendo il nuovo albero *T' (passo 1)*.



Codici di Huffman: correttezza

Dimostrazione L2: **Per le ipotesi del lemma, *x* e *y* hanno le frequenze minori in assoluto, mentre *b* e *d* sono arbitrarie.**
Quindi varranno anche: *f(x)* £ *f(b)* e *f(y)* £ *f(d)*.
Operiamo la trasformazione in due passi:

2- Poi scambiamo la posizione di *y* e *d* ottenendo il nuovo albero *T'' (passo 2)*.



Dobbiamo ora verificare che *T''* rappresenti ancora un *codice ottimo*.

Codici di Huffman: correttezza

Dimostrazione L2: **Calcoliamo la differenza di costo tra *T* e *T'*:**

$$B(T) - B(T') = \sum_{c \in S} f(c)d_T(c) - \sum_{c \in S} f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b)$$

$$- f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

$$= f(x)d_T(x) + f(b)d_T(b)$$

$$- f(x)d_T(b) - f(b)d_T(x)$$

$$= [f(b) - f(x)] \times [d_T(b) - d_T(x)]$$

$$\geq 0$$

Poiché le posizioni di *x* e *b* in *T'* sono quelle di *b* e *x* rispettivamente in *T*

Poiché sia $f(b) - f(x)$ che $d_T(b) - d_T(x)$ sono ≥ 0
$f(x) \leq f(b)$ e $d_T(x) \leq d_T(b)$

Cioè: $B(T) \geq B(T')$

Codici di Huffman: correttezza

Dimostrazione L2: **In modo analogo a quanto appena fatto, si dimostra che passando da *T'* a *T''* il *costo dell'albero* (del codice) *non può aumentare* (*B(T') - B(T'')* $\geq$ 0, cioè *B(T')* $\geq$ *B(T'')*).**

In conclusione: *B(T)* $\geq$ *B(T'')*.

Ma poiché *T* era ottimo per ipotesi, allora deve necessariamente valere *B(T) = B(T'')*.

Ora *T''* è un albero ottimo in cui *x* e *y* sono foglie a *profondità massima* dello *stesso padre*.
I *codici* di *x* e *y* avranno la *stessa lunghezza* e *differiranno* quindi solo per l'*ultimo bit*.

Codici di Huffman: correttezza

In che senso però l'algoritmo di Huffman è greedy?

- **Cioè perché *scegliere* ad ogni passo i *caratteri con* la *frequenza minore* è una *scelta greedy*?**

Il *costo di una fusione di due caratteri* può essere visto come la *somma delle frequenze dei due caratteri fusi assieme*.

- **Si può *dimostrare* che il *costo totale di un albero* (codice) può essere definito equivalentemente come *somma dei costi delle operazioni di fusione* necessarie.**

Huffman ad ogni passo sceglie quindi tra tutte le fusioni quella che costa di meno.

Algoritmi Greedy

- **L'*algoritmo di Huffman* è in esempio di *algoritmo greedy*:**
  - **Fonde ad ogni iterazione i due nodi più piccoli (per frequenza), senza considerare affatto gli altri**
  - **Una volta che la scelta è stata fatta, la mantiene fino alla fine – l'algoritmo non cambia mai decisione**

Concusione

- ***Alberi binari "completi"*** **possono rappresentare** ***codici di compressione***
- **Una** ***albero di Huffman*** **rappresenta un particolare** ***codice*** **per un dato file:**
  - **che è un** ***codice ottimo*** **per il file**
  - **che è generato da un** ***algoritmo greedy***