Capitolo 1
Array, liste e alberi

Lucidi tratti da
P. Crescenzi · G. Gambosi · R. Grossi · G. Rossi
Strutture di dati e algoritmi
Progettazione, analisi e visualizzazione
Addison-Wesley, 2012
`http://algoritmica.org`

1. I lucidi sono utilizzabili dai soli docenti e se ne sconsiglia la distribuzione agli studenti: oltre al rischio di violare una qualche forma di copyright, il problema principale è che gli studenti studino in modo superficiale la materia senza il necessario approfondimento e la dovuta riflessione che la lettura del libro fornisce
2. Il simbolo [alvie] nei lucidi indica l'uso di ALVIE per visualizzare il corrispettivo algoritmo: per un proficuo rendimento dello strumento, conviene esaminare in anticipo la visualizzazione per determinare i punti salienti da mostrare a lezione (l'intera visualizzazione potrebbe risultare altrimenti noiosa)

Sequenze lineari

- n elementi $a_0, a_1, \ldots, a_{n-1}$ dove $a_j = (j+1)$-esimo elemento $(0 \leq j \leq n-1)$
- È importante il loro ordine relativo.
- Due modalità di accesso:
  - **diretto** in cui, dato j, si accede solo ad a_j (**array**, costo *costante*)
  - **sequenziale** in cui, dato j, si accede ad $a_0, a_1, \ldots, a_j$ (**liste**, costo $O(j+1)$ e costo $O(k)$ partendo da a_{j-k})

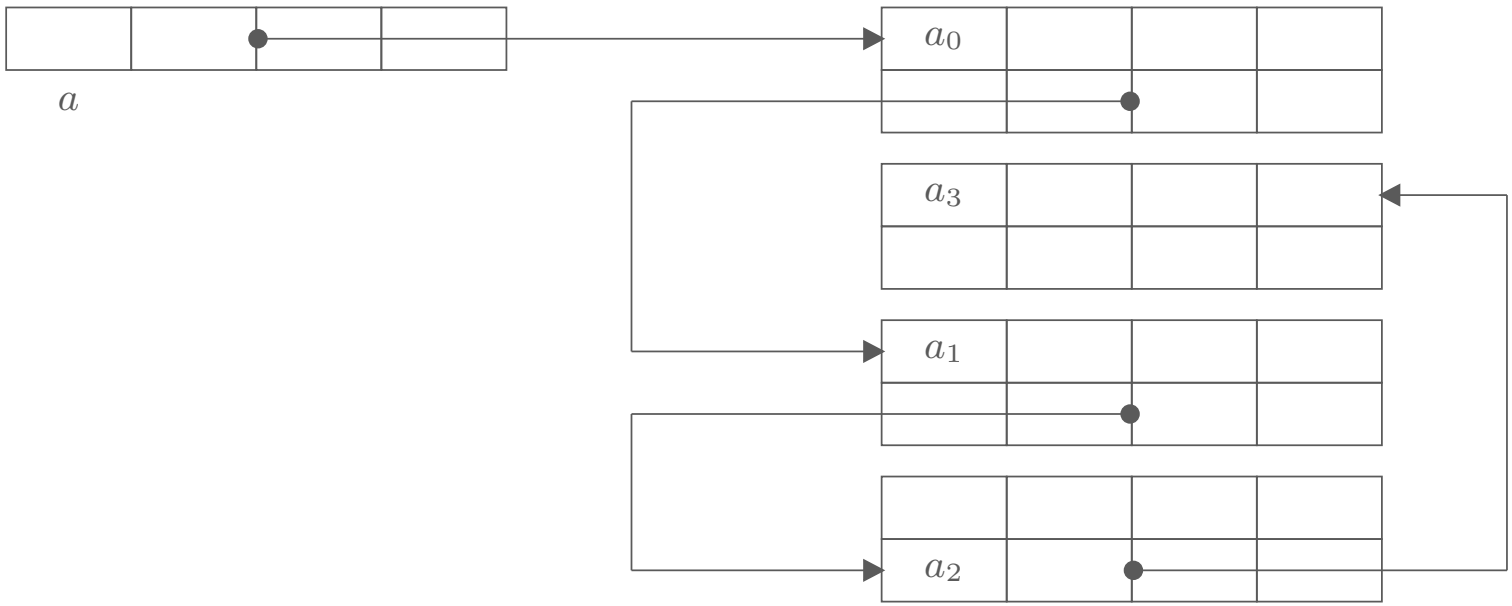
Sequenze: allocazione in memoria

- **Memoria** del calcolatore: celle contigue e indirizzabili singolarmente (numerate globalmente a partire da 0)
- Array = indirizzo a + posizioni **consecutive** (ad esempio, 4 byte per elemento, byte indirizzabili singolarmente)

- a: indirizzo di $a[0]$
- $a[j]$: $a + j \times \text{dim. elemento} = a + j \times 4$
  Accesso all'elemento a_j in $O(1)$ tempo

Sequenze: allocazione in memoria

- Lista = indirizzo a + posizioni **sparse** (a causa della gestione dinamica della memoria)



- a: indirizzo del primo elemento a_0
- Elemento a_{j-1}: indirizzo elemento a_j se esiste
- Accesso ad a_j: scandire i primi j elementi, iniziando con a_0 e accedendo via via al successivo, in $O(j+1)$ tempo totale

Array dinamici

- Alcuni linguaggi (C++, C#, Java) prevedono **array** che possono essere **ridimensionati**
- Ridimensionare un array a: aggiungere o eliminare una posizione in fondo all'array
- Approccio inefficiente richiede $O(n)$ tempo:
  - Crea un nuovo array b
  - Copia gli elementi di a in b
  - Dealloca a (o fallo deallocare) dalla memoria
  - Ridenomina b come a
- È possibile pagare tale costo ogni $\Omega(n)$ ridimensionamenti?

Array dinamici: approccio "spalmato"

- Idea: abbondare nel raddoppio e dimezzamento
  n elementi in un array di d posizioni [alvie]

```
1  VerificaRaddoppio( ):
2    IF (n == d) {
3      b = NuovoArray( 2 × d );
4      FOR (i = 0; i < n; i = i+1)
5        b[i] = a[i];
6      a = b;
7    }
```

- Con un **raddoppio**, $n = d + 1$ elementi sono copiati in un array b di $2d$ elementi
- Occorrono **almeno** $n - 1$ **inserimenti** prima di un ulteriore raddoppio

Array dinamici: approccio "spalmato"

```
1  VerificaDimezzamento( ):
2    IF ((d > 1) &&  (n == d/4)) {
3      b = NuovoArray( d/2 );
4      FOR (i = 0; i < n; i = i+1)
5        b[i] = a[i];
6      a = b;
7    }
```

- Con un **dimezzamento**, $n = d/4$ (perché?) elementi sono copiati in un array b di $d/2$ elementi
- Occorrono **almeno** $n/2$ **cancellazioni** prima di un ulteriore dimezzamento

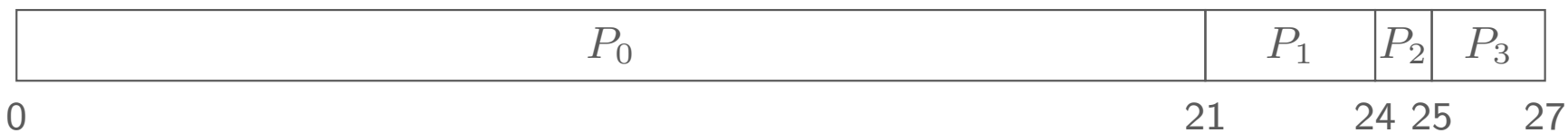
Array dinamici: approccio "spalmato"

- Riassumendo:
  - $n = d + 1$: raddoppia
  - $n = d/4$: dimezza
- Dopo un raddoppio o dimezzamento:
  - occorrono almeno $n - 1$ inserimenti per un ulteriore raddoppio
  - occorrono almeno $n/2$ cancellazioni per un ulteriore dimezzamento
- Costo $O(n)$ di ridimensionamento è spalmato su almeno $n/2$ operazioni: costo $O(1)$ "ammortizzato" in più per operazione

Opus libri: scheduling della CPU

- Task "indivisibili" P_0, P_1, P_2, P_3 da eseguire sulla CPU del calcolatore
- Tempi previsti (ms): $t_0 = 21$, $t_1 = 3$, $t_2 = 1$, $t_3 = 2$
- **Scheduling**: sequenza di esecuzione dei task (ipotesi: i task sono disponibili al tempo 0)

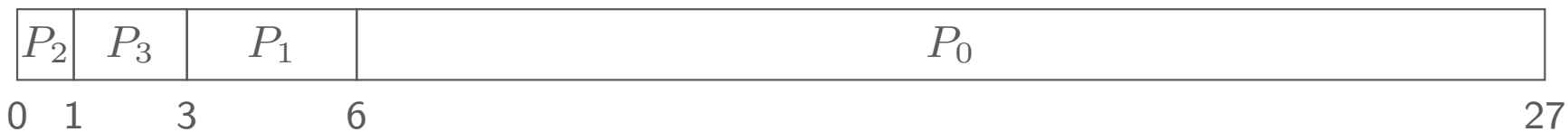
First Come First Served (FCFS)



Tempo medio d'attesa: $(0 + 21 + 24 + 25)/4 = 17,5$ ms

Scheduling della CPU

- Esempio: coda di stampa
- È possibile migliorare il tempo medio d'attesa?
- Task P_j con valori t_j piccoli vanno eseguiti prima
  Shortest Job First (SJF): **OTTIMO** (perché?)



Tempo medio d'attesa: $(0 + 1 + 3 + 6)/4 = 2,5$ ms

Ordinamento (sorting)

- Richiesto per realizzare SJF e miliardi di altre applicazioni.
  Ad esempio, $21, 3, 1, 2$ diviene $1, 2, 3, 21$

  > *Dato un array di n elementi e una loro relazione d'ordine $\leq$, disporli in modo che risultino ordinati (per esempio, in modo crescente) secondo la relazione $\leq$*

- Vediamo due semplici algoritmi:
  - SelectionSort
  - InsertionSort
- In seguito mostreremo algoritmi più efficienti

SelectionSort

- Passo i: **seleziona** l'elemento di rango $(i+1)$ ossia il minimo tra i rimanenti $n-i$ elementi

```
1  SelectionSort( a ):
2    FOR (i = 0; i < n; i = i+1) {
3      minimo = a[i];
4      indiceMinimo = i;
5      FOR (j = i+1; j < n; j = j+1) {
6        IF (a[j] < minimo) {
7          minimo = a[j];
8          indiceMinimo = j;
9        }
10     }
11     a[indiceMinimo] = a[i];
12     a[i] = minimo;
13   }
```

- [alvie]

Analisi del SelectionSort

- Usiamo le regole viste: al passo i del `FOR` esterno, il costo t_i è dominato dal costo del ciclo `FOR` interno
- Il ciclo `FOR` interno richiede meno di $n - i$ iterazioni, ciascuna di costo costante
- Qundi $t_i = O(n - i)$ per il `FOR` esterno
- In totale SelectionSort richiede $O(n^2)$ tempo perché questo è proporzionale a

$$\sum_{i=0}^{n-1}(n-i) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

InsertionSort

- Passo i: **inserisci** l'elemento in posizione i al posto giusto tra i primi i elementi (già ordinati)

```
1  InsertionSort( a ):
2    FOR (i = 0; i < n; i = i+1) {
3      prossimo = a[i];
4      j = i;
5      WHILE ((j > 0) && (a[j-1] > prossimo)) {
6        a[j] = a[j-1];
7        j = j-1;
8      }
9      a[j] = prossimo;
10   }
```

- [alvie]

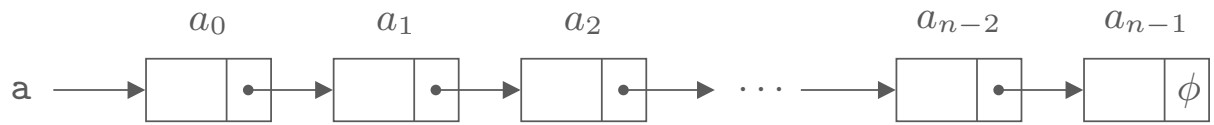
Analisi dell'InsertionSort

- Usiamo le regole viste: al passo i del `FOR` esterno, il costo t_i è dominato dal costo del ciclo `WHILE` interno
- Il ciclo `WHILE` interno richiede al massimo $i + 1$ iterazioni, ciascuna di costo costante
- Quindi $t_i = O(i + 1)$ per il `FOR` esterno
- In totale InsertionSort richiede $O(n^2)$ tempo perché questo è proporzionale a

$$\sum_{i=0}^{n-1}(i+1) = \frac{n(n+1)}{2}$$

- Osservazione: può richiedere $O(n)$ operazioni (quando?)

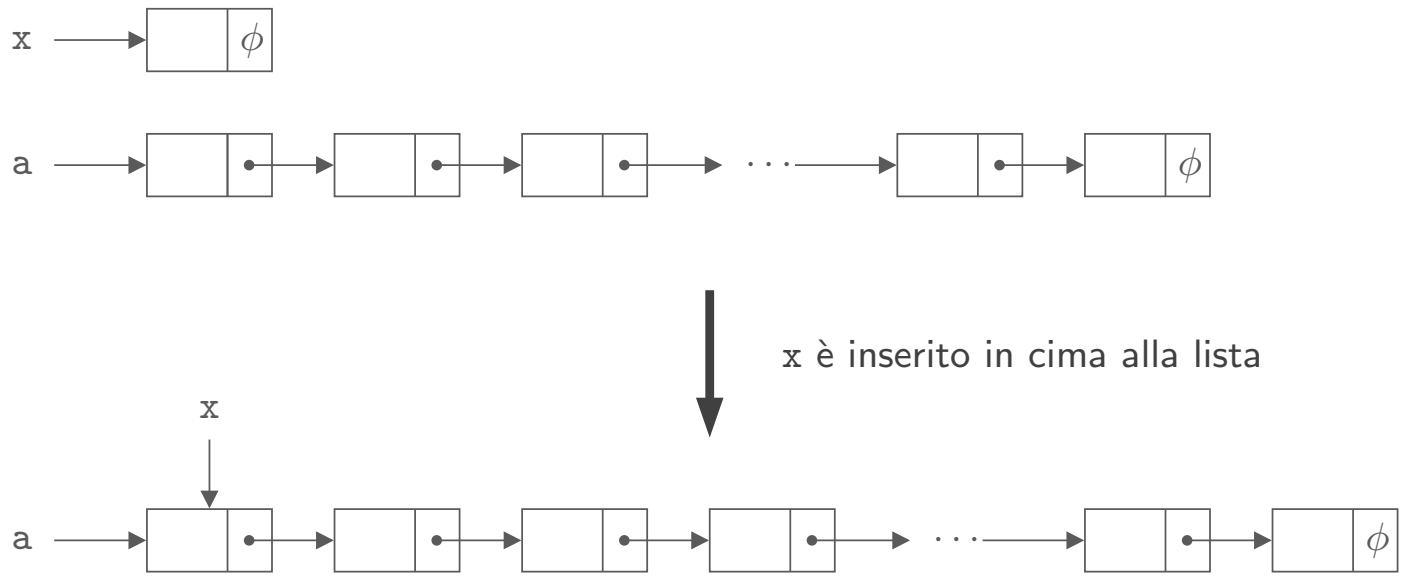
Liste



- elemento `x`
  - `x.succ` è il successore di `x` (`null` se non esiste)
  - `x.dato` è il contenuto informativo di `x`
- accesso all'elemento in posizione i in tempo $O(i+1)$:

```
p = a;
j = 0;
WHILE ((p != null) && (j < i)) {
  p = p.succ;
  j = j+1;
}
```

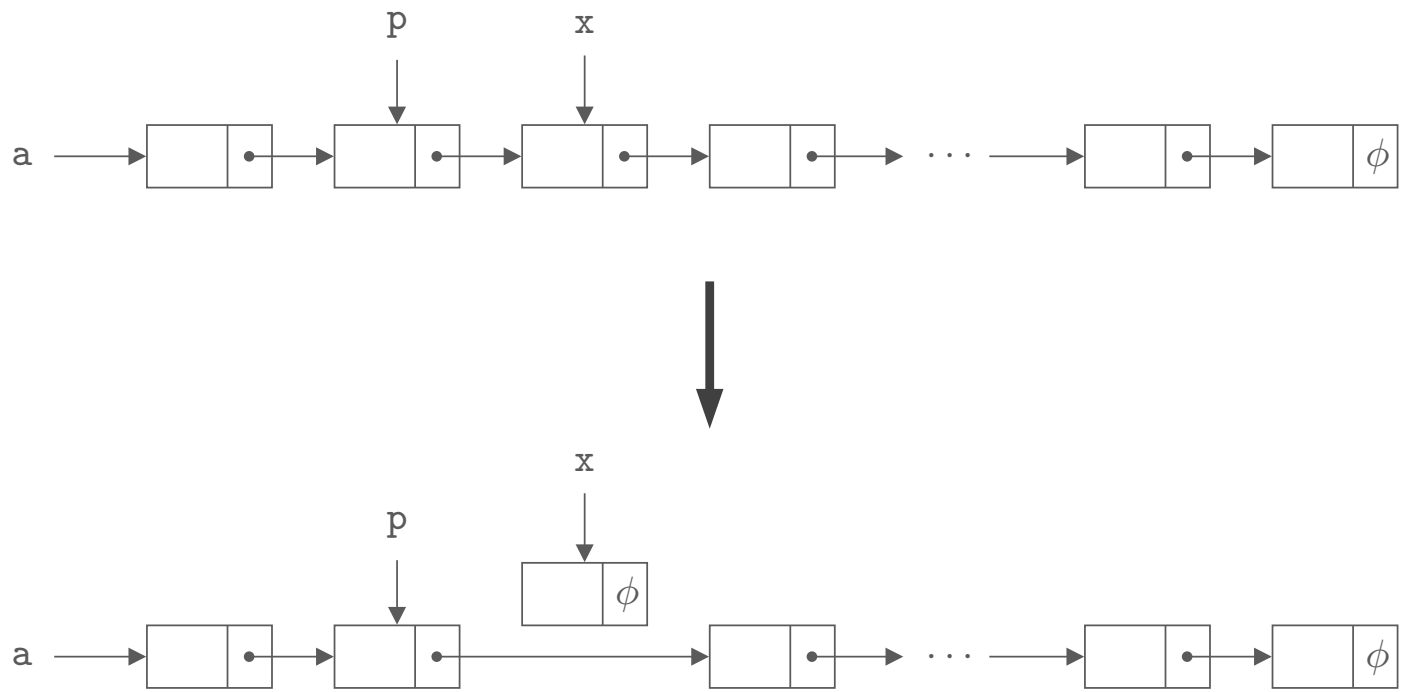
Inserimento in cima




```
x.succ = a;
a = x;
```

- Esercizio: inserimento in una posizione interna

Cancellazione di un elemento interno

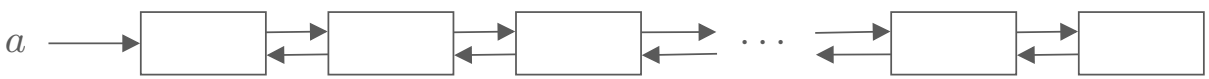



```
p.succ = x.succ;
x.succ = null;
```

- Esercizio: cancellazione del primo elemento, dell'ultimo o dell'unico elemento di una lista

Liste doppie

- `x.pred` è il predecessore

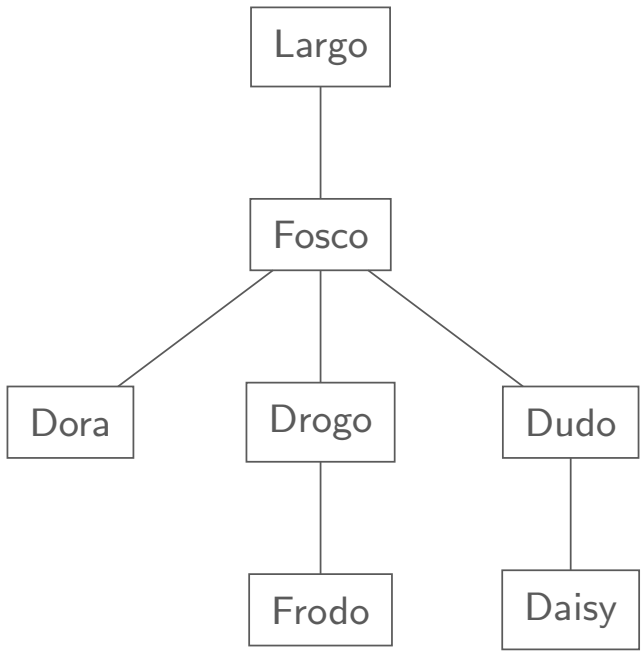


- Esempio: cancellazione di un elemento interno

```
x.succ.pred = x.pred;
x.pred.succ = x.succ;
x.succ = null;
x.pred = null;
```

Alberi

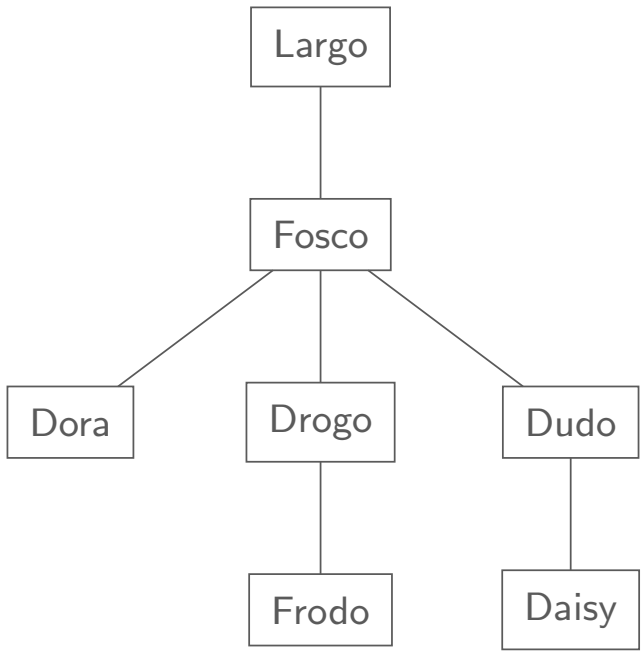
- Generalizzazione delle liste: più successori



- Elemento dell'albero: **nodo**
- Collegamento tra due nodi: **arco**
- `Largo`: **radice**
- `Dora`, `Frodo` e `Daisy`: **foglie**
- Tutti gli altri: **nodi interni**

Alberi

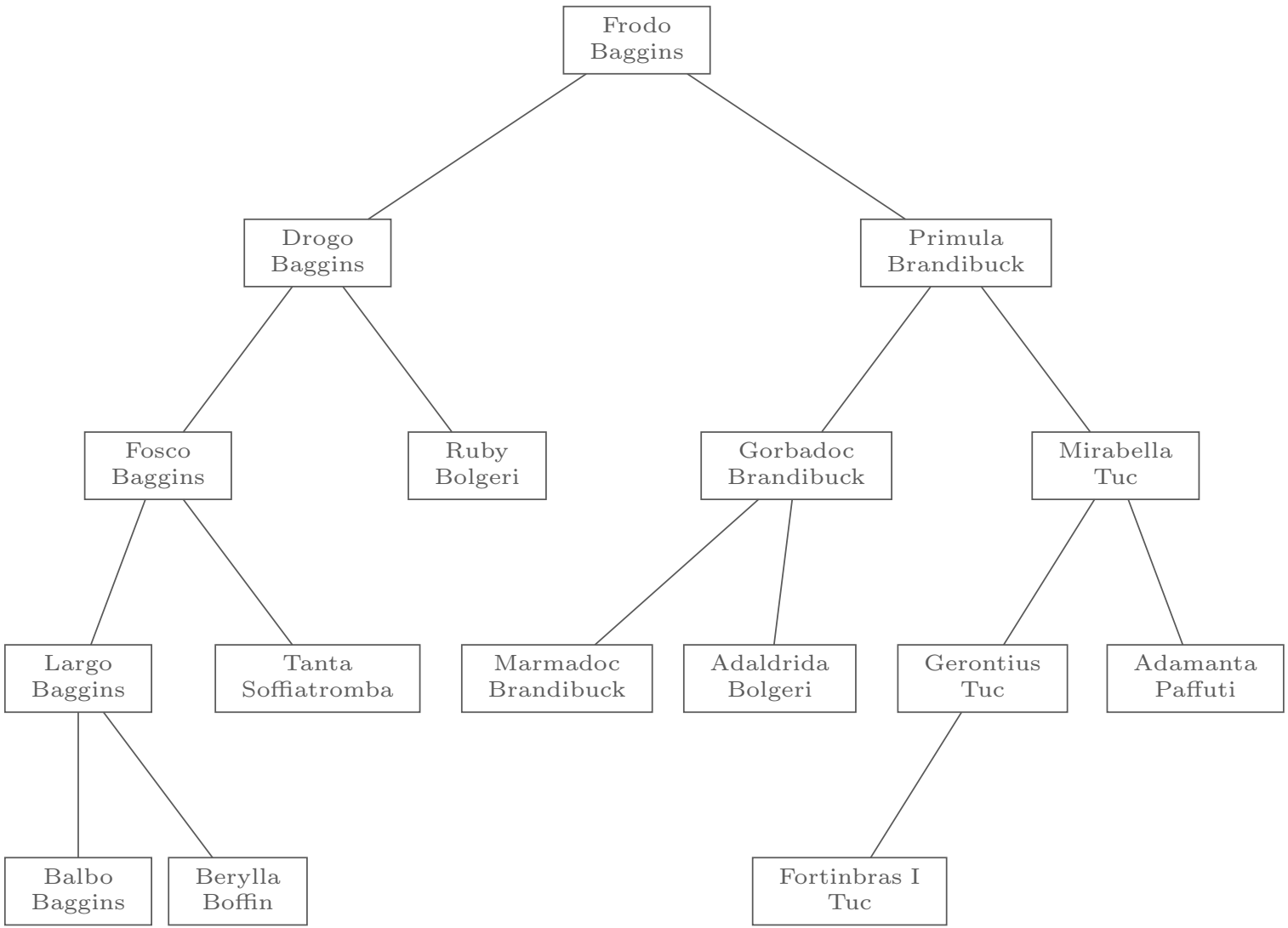
- Generalizzazione delle liste: più successori



- `Largo` è **antenato** di `Frodo`
- `Fosco` è **padre** di `Drogo`
- `Dudo` è **fratello** di `Drogo`
- `Frodo` è **discendente** di `Largo`
- Albero con radice `Fosco`: **sottoalbero**

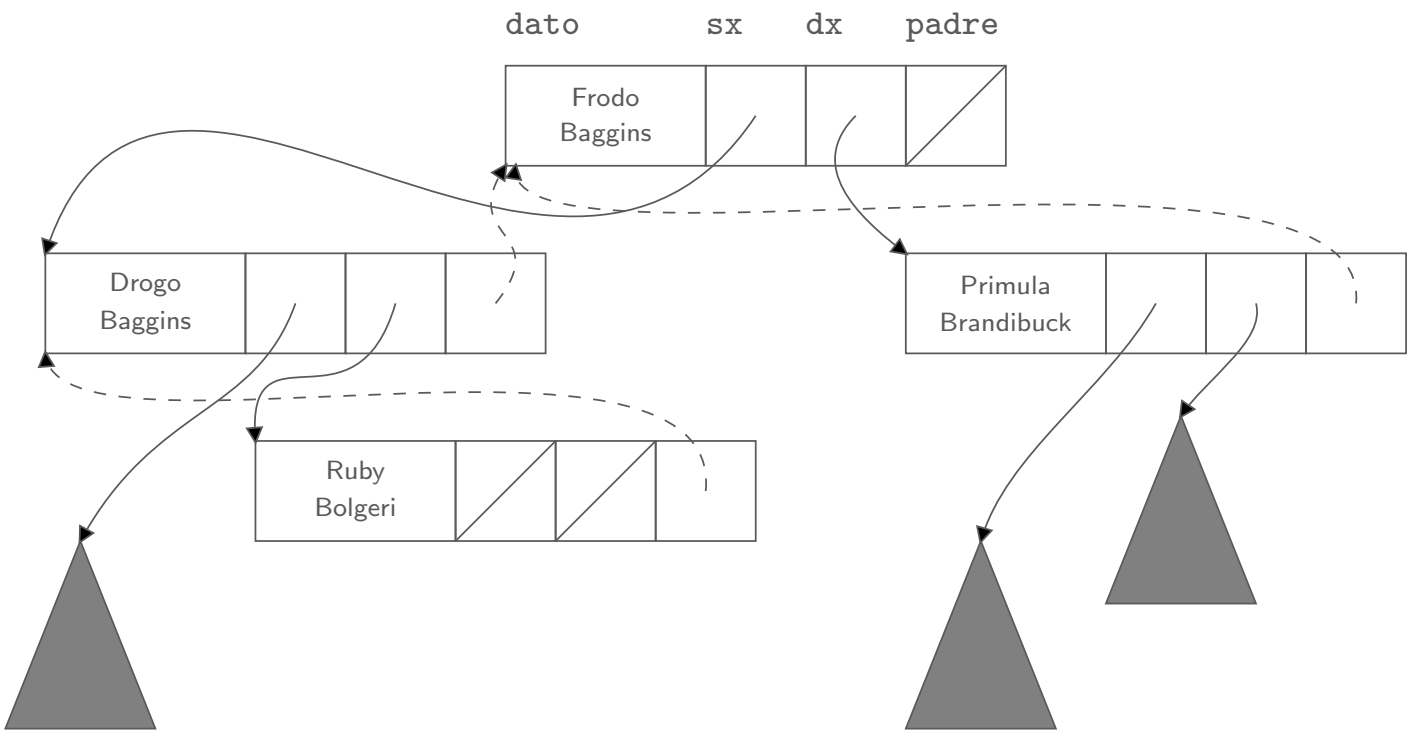
Alberi binari

- Ogni nodo ha due figli (`null` = vuoto): sinistro e destro



Rappresentazione in memoria

- Nodo `u`: campi `u.dato`, `u.sx`, `u.dx`, `u.px`



Alberi cardinali e ordinali

Alberi cardinali

- Ogni nodo ha k riferimenti ai figli, i quali sono enumerati da 0 a $k-1$
- I riferimenti ai figli di un nodo sono memorizzati in un array di dimensione k
- Elemento i-esimo dell'array: riferimento (eventualmente uguale a `null`) all'i-esimo figlio

Alberi ordinali

- Ogni nodo memorizza soltanto la lista ordinata dei riferimenti *non vuoti* ai suoi figli.

Sono due strutture di dati *differenti*, nonostante l'apparente somiglianza

Memorizzazione binarizzata

- Usata per memorizzare alberi k-ari mediante nodi di dimensione fissa (a due riferimenti).