Alberi generali

Gli alberi visti fino a questo momento erano insiemi di nodi collegati fra di loro, in cui ogni nodo poteva avere al massimo due figli. Negli alberi generali, ogni nodo può avere un numero arbitrario di figli. Non esiste quindi un limite al numero dei figli di ogni nodo.

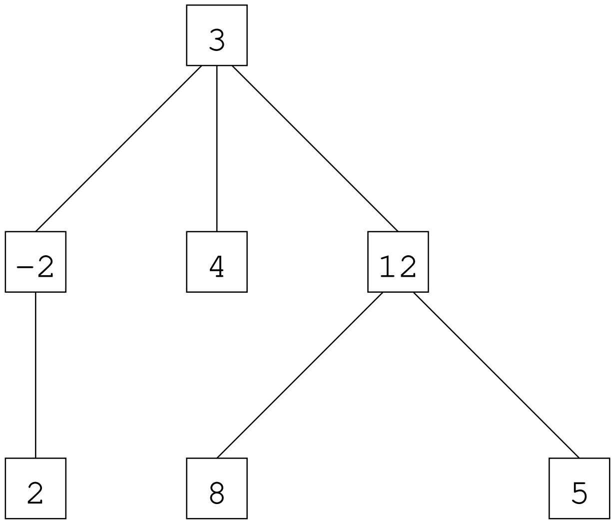
Definizione di albero generale

Gli alberi generali sono strutture dati collegate, ossia sono insiemi di nodi in cui esistono dei legami fra i nodi. Le strutture dati collegate viste fino a questo momento sono le liste e gli alberi binari, che hanno le seguenti caratteristiche:

liste
: ogni nodo ha un successore, tranne l'ultimo della lista che ne ha zero;

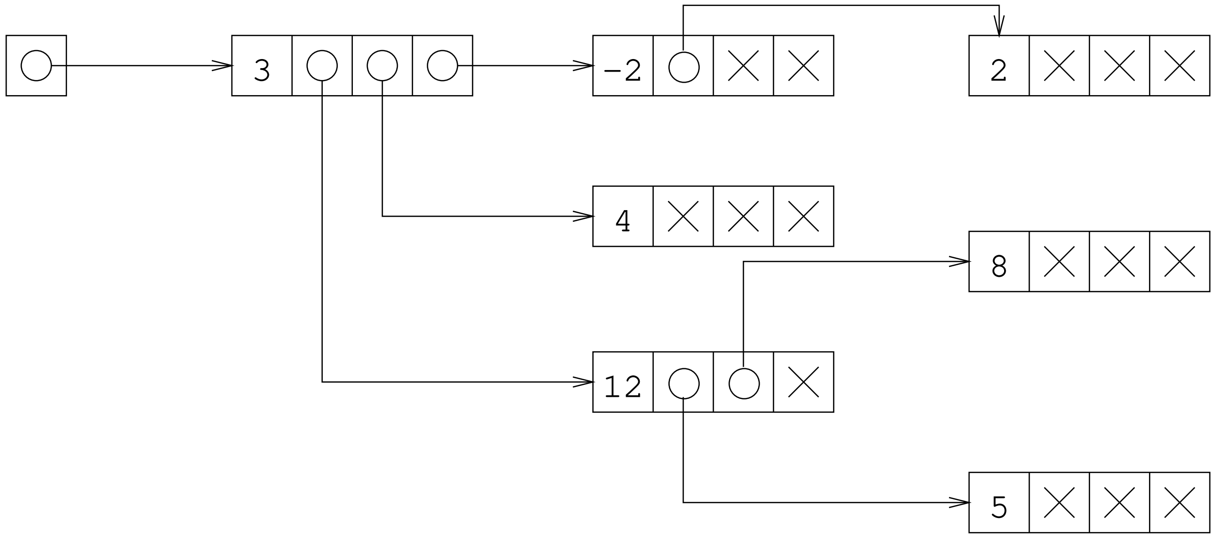
alberi binari
: ogni nodo ha zero, uno oppure due figli

Possiamo quindi dire che l'unica differenza è che nelle liste ogni nodo ha zero oppure un nodo collegato (successore), mentre negli albero ogni nodo può essere collegato a zero, uno oppure due nodi (figli). La differenza è quindi soltanto nel numero massimo dei figli. Elevando questo numero si possono definire altri tipi di struttura dati: per esempio, se si fissa il numero massimo di successori a tre, otteniamo una struttura dati simile agli alberi binari, in cui ogni nodo ha un numero di figli che va da zero a tre. Questo è un possibile esempio.



Come si vede, la radice ha tre figli, ossia i nodi `-2`, `4` e `12`. A loro volta, `-2` ha un solo figlio, `4` non ne ha nessuno, mentre `12` ne ha due. Questo è quindi un albero ternario, dato che ogni nodo ha un numero di figli che non supera il tre.

La rappresentazione in memoria di questi alberi non è difficile. Si tratta infatti di estendere i principi di rappresentazione usati per gli alberi binari. Nel caso della rappresentazione con strutture e puntatori, ogni nodo è ancora rappresentato da una struttura. Soltanto che ora ogni nodo ha tre figli, quindi la struttura deve avere, oltre al campo per il nodo, tre campi per i figli (invece di due come per gli alberi binari). Come al solito, quando un figlio è assente, si può usare il valore `NULL` invece del puntatore al figlio. La rappresentazione in memoria dell'albero di sopra è quindi la seguente. Si noti che l'albero può comunque essere vuoto, per cui l'intero albero viene rappresentato con un puntatore: se vale `NULL`, l'albero è vuoto (viceversa, per rappresentare l'albero vuoto usiamo il valore `NULL`).



Il tipo di dati necessario per ottenere questa rappresentazione è semplicemente una estensione della definizione di tipo usata per gli alberi binari: l'unica cosa che cambia è il fatto che la struttura ha un quarto campo puntatore.

```
struct NodoAlbero {
  int info;
  struct NodoAlbero *sinistro;
  struct NodoAlbero *centrale;
  struct NodoAlbero *destro;
};

typedef struct NodoAlbero *TipoAlbero;
```

Le funzioni usate sugli alberi binari si estendono in modo ovvio agli alberi ternari, semplicemente considerando che ogni nodo ha tre sottoalberi invece di due. Per esempio, per stampare tutti i nodi di un albero, basta modificare la funzione di stampa in preordine, aggiungendo la stampa ricorsiva del terzo sottoalbero.

```
int StampaAlberoTernario(TipoAlbero a) {
  if(a==NULL)
    return;

  printf("%d\n", a->info);

  StampaAlberoTernario(a->sinistro);
  StampaAlberoTernario(a->centrale);
  StampaAlberoTernario(a->destro);
}
```

Per la rappresentazione usando vettori, partiamo dal concetto di completare l'albero con nodi ''virtuali'', e poi di mettere i nodi dell'albero nel vettore procedendo per livelli (prima la radice, poi i suoi figli, ecc). Nel vettore si mettono anche i nodi che nell'albero non c'erano, ma si segnala nel campo booleano la loro assenza. Nel caso dell'albero della figura di sopra, i livelli sono i seguenti (`X` indica che il nodo non esiste).

```
3
-2 4 12
2 X X X X X 8 X 5
```

Il vettore contiene quindi questi elementi in ordine, in cui la `X` diventa un elemento del vettore in cui il campo `esiste` vale `FALSE`. La definizione del tipo di dato è uguale a quella per gli alberi binari. Dato un nodo, è possibile risalire ai suoi tre figli usando una regola lineare che è facile da derivare.

Alberi generali

Gli alberi generali hanno nodi con un numero arbitrario di figli. Un nodo può quindi avere zero figli, o ne può avere cento.

La differenza fondamentale fra gli alberi generali e gli alberi binari (o ternari) non è tanto il fatto che qui un nodo può avere un maggiore numero di figli, quanto il fatto che questo numero non è limitato a priori. In altre parole: quando si va a scrivere la definizione del tipo, non è noto nessun limite sul numero dei figli.

A causa di questo, le due rappresentazioni classiche degli alberi non si possono più usare, per i seguenti motivi:

vettori
: La rappresentazione con vettori richiede un passo preliminare di completamento dell'albero, che consiste nel mettere dei nodi fittizi in tutti i punti in cui un figli non c'è. In altre parole, se un nodo non ha tutti i figli che potrebbe avere (due per gli alberi binari, tre per quelli ternari), aggiungiamo i nodi che mancano. Nel caso degli alberi binari, non esiste un limite al numero di figli; questo passo di completamento non si può quindi fare (richiederebbe di aggiungere un numero infinito di nodi).

strutture
: Quando la struttura viene definita, viene specificato il numero di campi che ha. Nel caso degli alberi binari o ternari, si definiva ogni struttura come avente il massimo numero di figli, e poi si usava `NULL` quando un figli mancava. Nel caso degli alberi generali, il numero di figli non ò limitato: un albero potrebbe avere al massimo un nodo con dieci figli, un altro potrebbe avere un nodo con cento figli, ecc. In ogni caso, il numero massimo di figli non è noto quando viene scritta la definizione di tipo.

Per poter dare una rappresentazione in memoria per gli alberi generali, usiamo la seguente definizione: ogni nodo ha un certo numero di figli, che sono anche essi dei nodi. Per ogni nodo, dobbiamo quindi rappresentare due cose:

1. il suo valore
2. l'insieme dei suoi successori

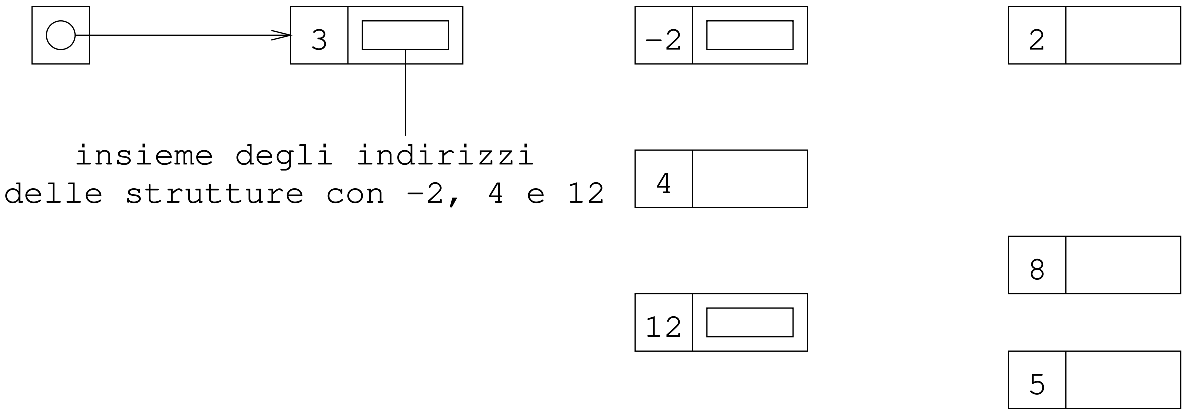
Dal momento che abbiamo due cose da rappresentare, usiamo certamente una struttura per ogni nodo. Il primo campo è il valore del nodo. Come al solito, assumiamo che sia un intero, ma potrebbe essere un tipo qualsiasi.

```
struct NodoAlbero {
  int info;
  ...            // rappresentazione insieme successori
};
```

Per quello che riguarda l'insieme dei successori, le sue caratteristiche sono:

1. è un insieme i cui membri sono di tipo `struct NodoAlbero *`; in altre parole, è un insieme di indirizzi: ogni indirizzo è l'indirizzo di una struttura `NodoAlbero`
2. il numero dei suoi elementi è variabile: può essere vuoto, oppure composto da uno o più elementi;
3. il numero massimo di elementi non è noto.

L'albero di sopra (che è ternario) si può chiaramente anche rappresentare come albero generale. Abbiamo la seguente rappresentazione parziale: per ogni nodo c'è una struttura; il primo campo contiene l'informazione del nodo; il resto della struttura indica l'insieme dei successori, ed è quindi un insieme di altre strutture (le strutture che rappresentano i nodi figli).

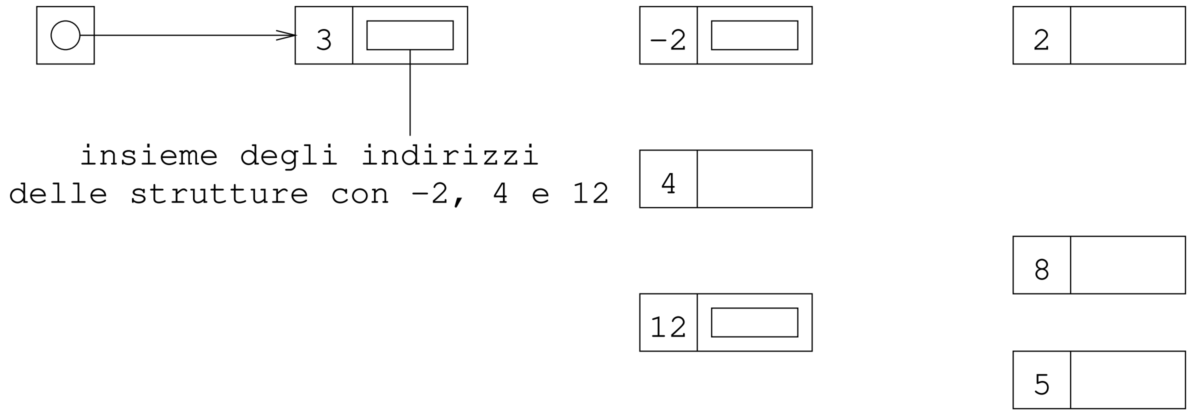


Per quello che riguarda la rappresentazione dell'insieme dei successori di ogni nodo, si tratta di trovare un modo di rappresentare un insieme di dati di cui non si conosce un numero massimo. D'altra parte, conosciamo già due modi per rappresentare insiemi di questo genere: i vettori dinamici e le liste collegate.

Iniziamo quindi con la rappresentazione con vettori dinamici, e vediamo poi la rappresentazione con liste.

Alberi generali con vettori

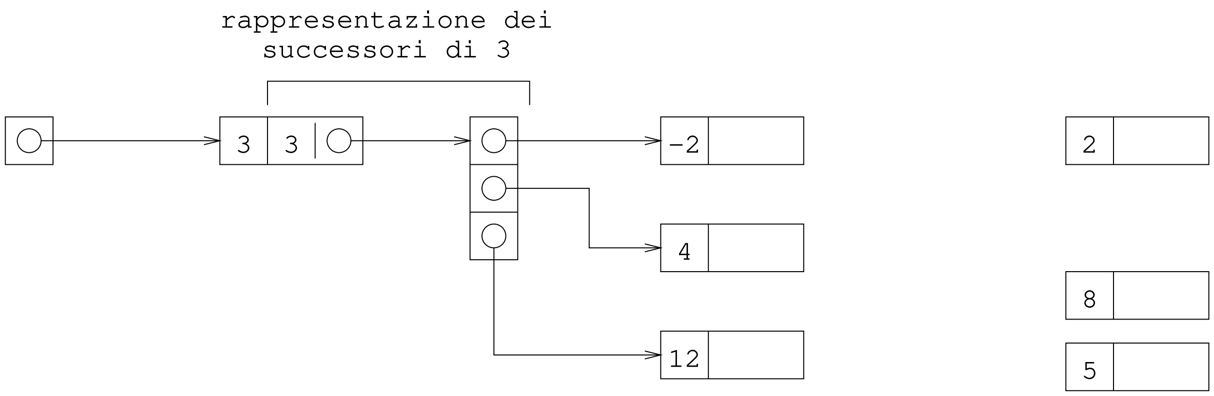
Dal momento che ogni nodo di un albero viene rappresentato come una struttura, il problema da risolvere è quello di specificare i figli di un nodo. Dal momento che i figli sono nodi, si tratta quindi di rappresentare un insieme di indirizzi di strutture. Un esempio è qui sotto.



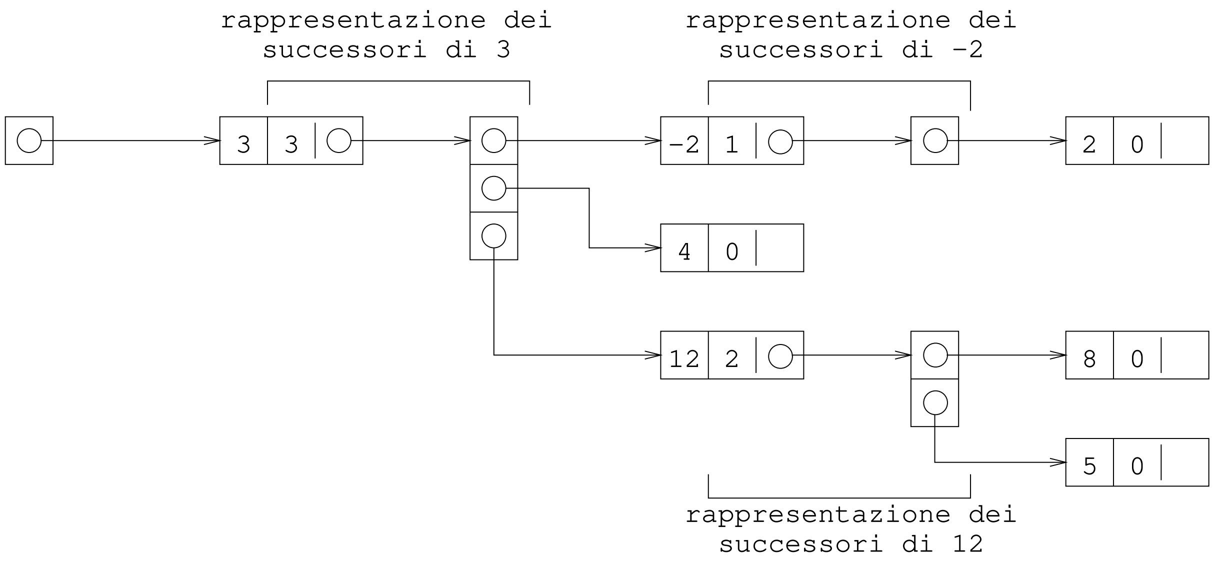
Ci serve un modo per dire che i successori del nodo `3` sono le strutture che contengono i nodi `-2`, `4` e `12`, ossia dobbiamo rappresentare l'insieme composto dall'indirizzo di queste tre strutture.

Il metodo che vediamo in questa pagina è quello che usa vettori dinamici. Per semplicità, consideriamo il caso più semplice, ossia quello di vettore completamente utilizzato. In altre parole, un vettore dinamico di questo tipo è caratterizzato da un puntatore e da un intero che dice la dimensione del vettore, che coincide con il numero di elemente correntemente usati del vettore. Prima di vedere quale è esattamente la definizione di tipo, consideriamo come viene rappresentato in questo modo l'albero di sopra.

Cominciamo con la radice. Abbiamo una struttura che contiene il valore del nodo (`3`), e poi un vettore dinamico i cui elementi sono gli indirizzi delle strutture che contengono i figli di `3`. Ogni vettore dinamico viene rappresentato da un numero di elementi e dal puntatore al vettore stesso. Dato che la radice ha tre figli, il numero di elementi è `3`, e il puntatore contiene l'indirizzo di un vettore di tre elementi. Ogni elemento di questo vettore contiene l'indirizzo di una delle strutture che sono i figli della radice. La struttura che contiene la radice è quindi composta di tre campi, di cui il secondo e il terzo complessivamente indicano quanti e quali sono i figli del nodo.



La stessa rappresentazione vale anche per i figli: per ogni figlio, ho una struttura che contiene il valore del nodo e altri due campi che indicano i successori. Come prima, questi due campi contengono il numero di successori e il puntatore alla zona di memoria dove si trova il vettore.



Questa figura si ottiene considerando che il nodo `−2` ha un unico figlio, quindi per i suoi successori serve un vettore composto di un unico elemento; per il nodo `12`, dal fatto che ha due figli si ottiene che serve un vettore dinamico di due elementi. I nodi `2`, `8` e `5` non hanno figli, quindi basta mettere `0` nel campo che indica la dimensione del vettore (che è infatti anche il numero dei figli).

Lo schema finale sembra piuttosto complesso, ma si ottiene invece molto facilmente:

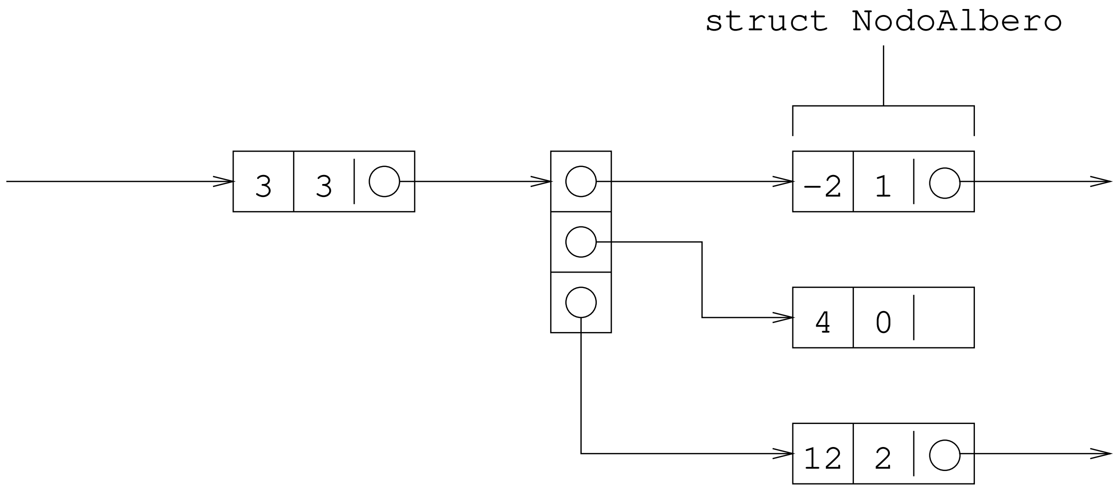
- per ogni nodo ho una struttura;
- la struttura contiene il valore del nodo e l'indicazione dei figli;
- l'insieme dei figli di un nodo è rappresentato come un vettore dinamico che contiene gli indirizzi dei figli;
- nella struttura ci metto i dati fondamentali del vettore dinamico, ossia il numero di elementi e il puntatore al vettore.

Abbiamo quindi capito come viene rappresentato in memoria l'albero. Ci serve ora la dichiarazione dei tipi. La struttura è composta di tre campi, i cui primi due elementi sono interi, e il terzo è un puntatore:

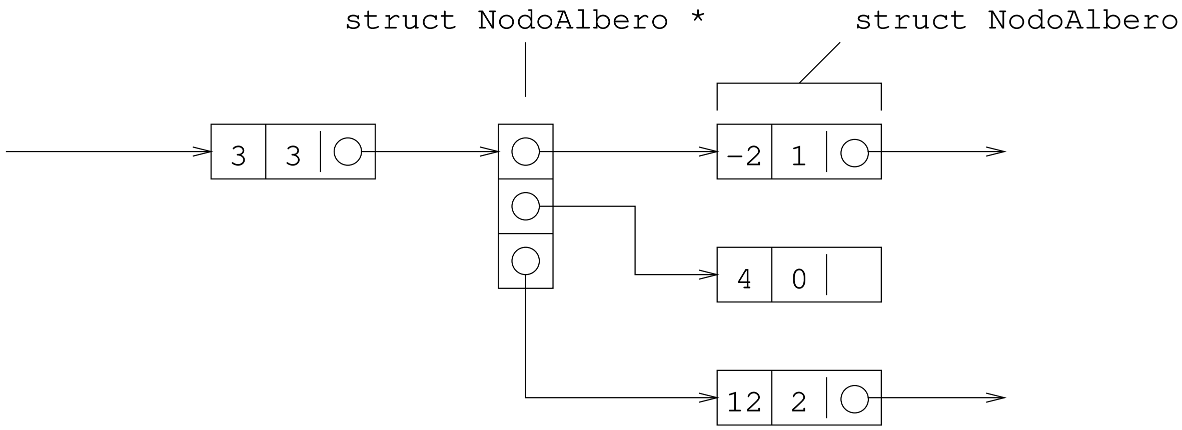
```
struct NodoAlbero {
  int info;
  int numfigli;
  ??? figli;
};
```

L'unica cosa che ci manca è il tipo del puntatore. Questo si può ottenere in questo modo: il puntatore contiene un indirizzo; il suo tipo è fatto dal tipo del dato che si trova a quell'indirizzo seguito da `*`. Per esempio, se ho una variabile che contiene l'indirizzo di una zona di memoria in cui ho un `int`, la variabile deve essere di tipo `int *`.

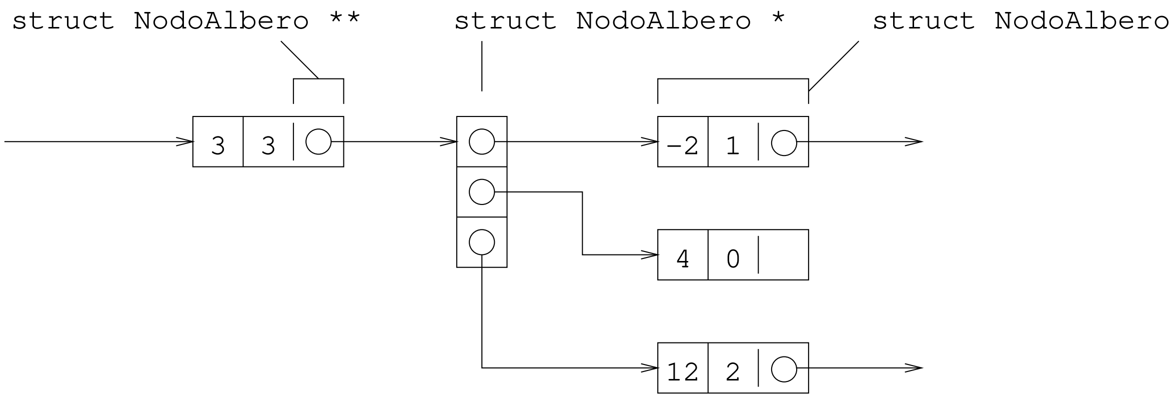
Nel nostro caso, il terzo elemento della struttura è un puntatore a una zona di memoria che contiene a sua volta un puntatore. Più precisamente, questo è un puntatore a una `struct NodoAlbero`. Quindi, gli elementi del vettore sono di tipo `struct NodoAlbero *`, e il puntatore che sta dentro la struttura è di tipo `struct NodoAlbero **`. Si può stabilire questo un passo per volta. Primo, la struttura è di tipo `struct NodoAlbero`.



Dato la struttura è di tipo `struct NodoAlbero`, e il vettore contiene indirizzi di queste strutture, ogni elemento del vettore è di tipo `struct NodoAlbero *`:



A sua volta, nella struttura c'è l'indirizzo del vettore. Il tipo di questo campo è dato semplicemente dal tipo dell'elemento del vettore a cui va aggiunto `*`. Questo perchè il terzo elemento della struttura è un puntatore, e il suo tipo è quello dell'oggetto puntato a cui si aggiunge `*`.



Abbiamo quindi la seguente dichiarazione di dati:

```
struct NodoAlbero {
  int info;
  int numfigli;
  struct NodoAlbero **figli;
};
```

Si può stabilire il tipo del terzo elemento anche facendo questo ragionamento alternativo: si tratta di rappresentare un vettore dinamico, e noi sappiamo come si fa nel caso dei vettori di intero. Stabiliamo quindi i tipi nel caso di vettori di interi, e poi facciamo le modifiche che servono per il caso degli alberi. Per rappresentare un vettore dinamico di interi, ci servono due variabili:

```
  int *p;       // puntatore alla zona di memoria
                // che contiene gli interi
  int dim;      // numero di elementi del vettore
```

Si noti che il secondo dato è sempre un intero, dato che è la dimensione del vettore. La prima variabile `p` dipende invece dal tipo degli elementi del vettore. Per un vettore di interi, `p` è di tipo `int *`. Se vogliamo rappresentare un vettore di reali, avremmo `float *`. Per vettore di caratteri, avremmo usato `char *`. In generale, se vogliamo rappresentare un vettore i cui elementi sono di tipo `TipoElemento`, allora la variabile deve essere di tipo `TipoElemento *`. Questo è dovuto al fatto che una variabile puntatore a un tipo contiene l'indirizzo di una zona di memoria dove sono memorizzati uno o più dati di quel tipo.

Nel nostro caso, gli elementi del vettore non sono di tipo intero. Come si è detto prima, devo rappresentare un insieme di nodi. Ogni nodo lo rappresento come un puntatore a una struttura. Quindi, ogni figlio è rappresentato dall'indirizzo di una struttura `struct NodoAlbero`. Gli elementi del vettore sono quindi di tipo `struct NodoAlbero *`. La variabile `p` deve quindi essere di tipo `struct NodoAlbero **p`. Ripetiamo:

1. voglio rappresentare un insieme di nodi
2. ogni nodo è rappresentato come l'indirizzo di una struttura `struct NodoAlbero`
3. quindi, ogni figlio lo rappresento come un dato di tipo `struct NodoAlbero *`, ossia puntatore a struttura `NodoAlbero`
4. mi serve un vettore in cui ogni elemento rappresenta un nodo, quindi ogni elemento è di tipo `struct NodoAlbero *`
5. per rappresentare un vettore i cui elementi sono di tipo `struct NodoAlbero *`, mi serve una variabile intera e una variabile puntatore al tipo, ossia una variabile `struct NodoAlbero **`

Questo ragionamento alternativo permette ugualmente di arrivare alla stessa dichiarazione di dati che si ottiene guardando la figura e mettendo uno `*` ogni volta che si risale una freccia.

Stampa di tutti i nodi

L'esercizio consiste nella stampa di tutti i nodi di un albero generale, rappresentato usando vettori dinamici.

Procediamo per raffinamenti successivi. Primo, se l'albero è vuoto non si stampa niente. Altrimenti, si stampa la radice e poi (ricorsivamente) i sottoalberi.

1. Se l'albero è vuoto, esci.
2. Altrimenti, stampa la radice, e poi i sottoalberi.

Fino a questo livello, l'algoritmo è lo stesso degli alberi binari: si stampa il primo nodo, e poi si procede ricorsivamente a partire da ognuno dei figli. La differenza è che ora non posso più tradurre la frase ''stampa i sottoalberi'' in ''stampa il sottoalbero sinistro e poi il destro''. Infatti, un sottoalbero è l'albero che si ottiene prendendo un figlio e tutti i nodi al di sotto. Quindi, il numero dei sottoalberi è uguale al numero dei figli, che non è noto a priori nel caso di alberi generali.

Nel caso di alberi generali, la frase ''stampa i sottoalberi'' si può tradurre in un ciclo del tipo: per ogni sottoalbero, stampalo. Abbiamo quindi il seguente algoritmo raffinato.

1. Se l'albero è vuoto, esci.
2. Altrimenti,
    1. stampa la radice;
    2. per ogni sottoalbero, stampalo.

Siamo ora vicini alla implementazione: i sottoalberi sono gli alberi che hanno come radice l'indirizzo di uno dei figli. Ma un albero è l'indirizzo di una struttura `struct NodoAlbero`. Quindi, devo fare la chiamata ricorsiva sull'indirizzo delle strutture che rappresentano i figli. Ma questi sono esattamente gli elementi del vettore dinamico.

Si tratta quindi di fare la chiamata ricorsiva su tutti gli elementi del vettore dinamico. Il numero di elementi che contiene è in `a->numfigli`, mentre gli elementi del vettore sono `a->figli[0]`, `a->figli[1]`, ecc.

```
void Stampa(TipoAlbero a) {
  int i;

  if(a==NULL)
    return;

  printf("%d ", a->info);
  for(i=0; i<a->numfigli; i++)
    Stampa(a->figli[i]);
}
```

Stampa in rappresentazione parentetica

L'esercizio consiste nella stampa di tutti i nodi di un albero generale, rappresentato usando vettori dinamici, in rappresentazione parentetica. La rappresentazione parentetica è simile a quella degli alberi binari: l'albero vuoto si stampa come `()`. Per ogni altro albero, si stampa `(`, poi la radice, poi i sottoalberi, e infine `)`.

Si tratta in sostanza di fare la stampa ricorsivamente:

1. Se l'albero è vuoto, stampa `()`
2. Altrimenti, stampa `(`, poi la radice, i sottoalberi e infine `)`.

Come si vede chiaramente, si tratta della stampa dei nodi dell'albero, in cui però a ogni passo stampo prima `(` e poi `)`.

Nel caso di alberi generali, per stampare i sottoalberi si usa un ciclo: il numero di figli (e quindi di sottoalberi) si trova in `a->numfigli`, mentre i puntatori ai nodi figli (ossia i sottoalberi) si trovano nel vettore `a->figli`.

```
void StampaAlbero(TipoAlbero a) {
  int i;

  if(a==NULL) {
    printf("()");
    return;
  }

  printf("( %d ",a->info);
  for(i=0; i<a->numfigli; i++)
    StampaAlbero(a->figli[i]);
  printf(" ) ");
}
```

Alberi random

Per generare un albero casuale, usare queste due funzioni:

```
TipoAlbero GeneraAlbero(int n) {
  TipoAlbero a;

  if(rand()%(n+1)==0)
    return NULL;

  return GeneraAlberoRic(n);
}

TipoAlbero GeneraAlberoRic(int n) {
  TipoAlbero a;
  int i;

  a=malloc(sizeof(struct NodoAlbero));
  a->info=rand()%100-30;
  if(n>0)
    a->numfigli=rand()%5;
  else
    a->numfigli=0;
  a->figli=malloc(sizeof(struct NodoAlbero *)*a->numfigli);

  for(i=0; i<a->numfigli; i++)
    a->figli[i]=GeneraAlberoRic(n-1);

  return a;
}
```

Per testare le funzioni: generare un albero casuale, stamparlo, chiamare la funzione e verificare il risultato. Non è necessario sapere perchè queste due funzioni di sopra sono corrette.

Presenza in un albero

Esercizio: scrivere una funzione che verifica se un albero contiene un certo elemento. Si assuma che i nodi siano interi: il prototipo è quindi:

```
int Presenza(TipoAlbero, int);
```

Soluzione: si tratta di implementare la stessa funzione ricorsiva che si è vista per gli alberi binari:

1. se l'albero e' vuoto, ritorna `FALSE` (l'elemento non c'è)
2. se l'elemento si trova nella radice dell'albero, ritorna `TRUE`, dato che sappiamo che l'elemento sta nell'albero
3. altrimenti, ritorna `TRUE` se l'elemento si trova in almeno uno dei sottoalberi

I primi due passi sono identici a quelli degli alberi binari. Per il terzo passo, occorre solo considerare che si tratta in effetti di fare la chiamata ricorsiva su ognuno dei sottoalberi, ritornando `TRUE` se qualche chiamata ritorna `TRUE`. L'ultimo passo si può quindi implementare come:

1. per ogni sottoalbero:
   1. se l'elemento sta nel sottoalbero, ritorna `TRUE`
2. ritorna `FALSE`

Chiaramente, si ritorna `FALSE` soltanto se nessuna delle chiamate ricorsive ha tornato `TRUE`. In altre parole, il `return FALSE;` va messo fuori dal ciclo (si esegue quindi soltanto se il `return` dentro il ciclo non viene mai eseguito).

```
int Presenza(TipoAlbero a, int e) {
  int i;

  if(a==NULL)
    return FALSE;

  if(a->info==e)
    return TRUE;

  for(i=0; i<a->numfigli; i++)
    if(Presenza(a->figli[i], e))
      return TRUE;

  return FALSE;
}
```

Somma delle foglie

Esercizio: scrivere una funzione che ritorna la somma delle foglie di un albero. Si noti che le foglie sono, come sempre, definite come i nodi che non hanno figli. Il prototipo della funzione è:

```
int SommaFoglie(TipoAlbero);
```

Soluzione: si procede come per gli alberi binari, ossia si considera che un albero, se non è vuoto, è composto da un nodo e da un certo numero di sottoalberi. Si considerano quindi i vari casi significativi:

albero vuoto
: la somma vale zero

albero composto da un solo nodo
: si tratta di un albero composto da una sola foglia: la somma coincide con il valore della foglia

albero composto da un nodo con figli
: il nodo non è una foglia: le foglie dell'albero stanno in uno dei sottoalberi. Si calcola quindi la somma in ognuno dei sottoalberi con la chiamata ricorsiva, e si sommano tutti i valori ottenuti

L'algoritmo si ottiene mettendo le opportune istruzioni condizionali:

1. se l'albero è vuoto, ritorna `0`
2. se la radice non ha figli, ritorna la radice
3. altrimenti, fai la somma in ognuno dei sottoalberi, e ritorna la somma di tutti questi valori.

La somma nel sottoalbero `a->figli[i]` si fa con la chiamata ricorsiva `SommaFoglie(a->figli[i])`. Questo calcolo va fatto con `i` che va da `0` a `a->numfigli-1`.

Come faccio a sommare tutti questi valori? Si tratta di sommare un certo numero di valori, quindi posso usare il metodo dell'accumulatore. In questo caso, posso usare il metodo perchè devo sommare un insieme di valori `SommaFoglie(a->figli[0])`, `SommaFoglie(a->figli[1])`, `SommaFoglie(a->figli[2])`, ecc. che sono tutti noti all'interno della funzione (sono valori di ritorno di funzioni chiamate tutte all'interno dello stesso record di attivazione). Il metodo non è corretto sulle funzioni ricorsive se si cerca di sommare dei valori che si trovano in record di attivazioni diversi.

```
int SommaFoglie(TipoAlbero a) {
  int somma, i;

  if(a==NULL)
    return 0;

  if(a->numfigli==0)
    return a->info;

  somma=0;
  for(i=0; i<a->numfigli; i++)
    somma+=SommaFoglie(a->figli[i]);

  return somma;
}
```

Si ricorda che la regola da seguire, nella scrittura delle funzioni ricorsive è la seguente:

> se la chiamata ricorsiva viene fatta su un problema più semplice di quello di partenza, è corretta

Nel caso degli alberi, si fa di solito la chiamata ricorsiva sui sottoalberi. A questo punto, si può pensare come se al posto della chiamata ricorsiva `SommaFoglie(a->figli[i])` ci sia una funzione di libreria, di cui la correttezza è stata già dimostrata. Possiamo quindi riscrivere la funzione mettendo, al posto della chiamata ricorsiva, la chiamta di una funzione di libreria (che in realtà non esiste) `leaf_sum`,

```
int SommaFoglie(TipoAlbero a) {
  int somma, i;

  if(a==NULL)
    return 0;

  if(a->numfigli==0)
    return a->info;

  somma=0;
  for(i=0; inumfigli; i++)
```

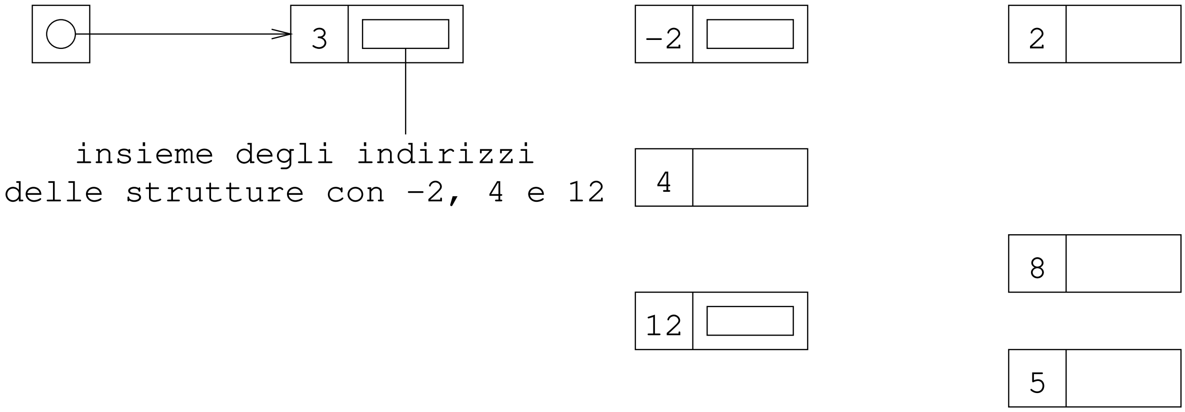
```
    somma+=leaf_sum(a->figli[i]);

  return somma;
}
```

Assumendo che `leaf_sum` sia una funzione di libreria che risolve il problema, la funzione completa è corretta. Questo ci dice che la funzione di partenza era corretta.

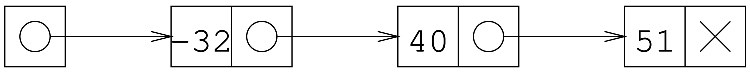
Alberi generali con liste

Consideriamo ancora una volta il problema di specificare l'insieme dei figli di ogni nodo. Per ogni nodo c'è una struttura. Abbiamo quindi bisogno di specificare, per ogni struttura, quali sono le strutture che contengono i figli. Quindi, per ogni struttura dobbiamo rappresentare l'insieme degli indirizzi delle strutture che rappresentano i figli.

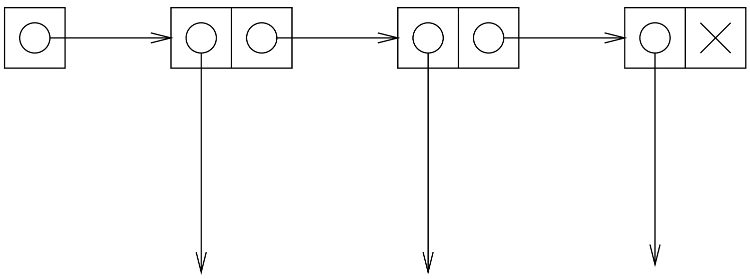


I vettori dinamici sono uno dei metodi che abbiamo visto per rappresentare insieme di cui non conosciamo a priori il numero o il numero massimo di elementi. Un secondo metodo è quello delle liste collegate. In questa pagina vediamo questo secondo metodo.

Con una lista possiamo rappresentare un insieme di interi. Per esempio, per l'insieme di tre elemento `-32`, `40` e `51` avremmo una rappresentazione del genere:

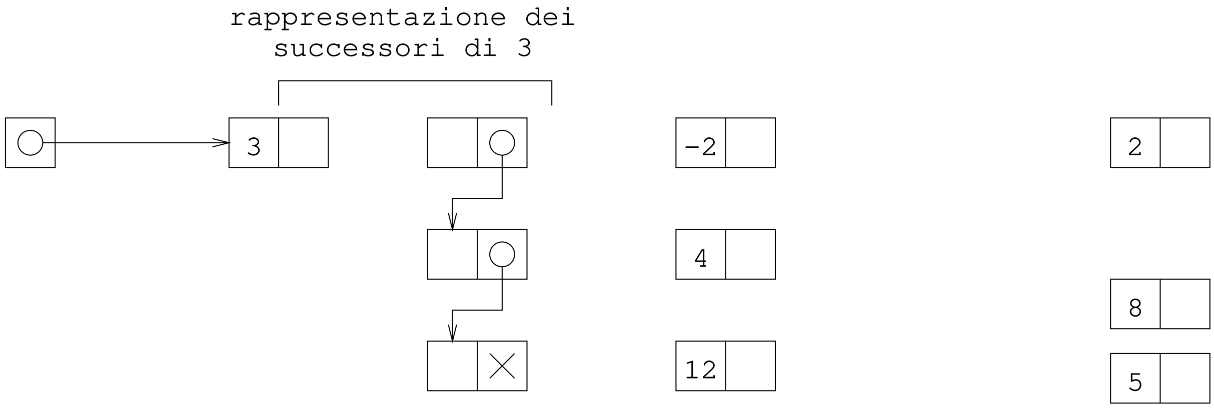


È chiaro che al posto degli interi si può mettere un qualsiasi altro tipo di dati, e al posto di un insieme di tre interi avremmo rappresentato un insieme di tre dati. Nel nostro caso, abbiamo bisogno di rappresentare un insieme di indirizzi. Ma questo non crea problemi: al posto del tipo `int`, avremo che il primo campo delle strutture sarà un indirizzo:

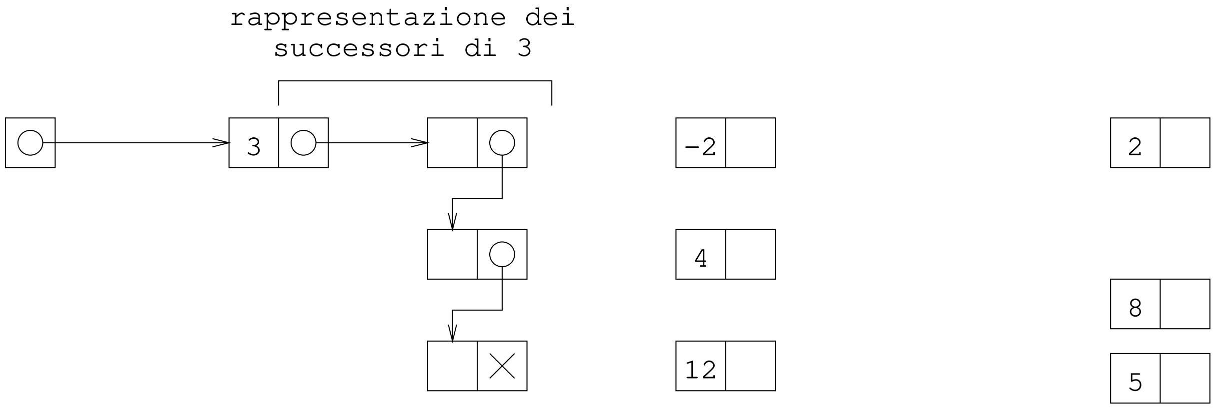


La dichiarazione di dati per le lista sarà poi modificata in modo tale che le strutture contengano un puntatore al posto di un intero, come si vedrà più avanti.

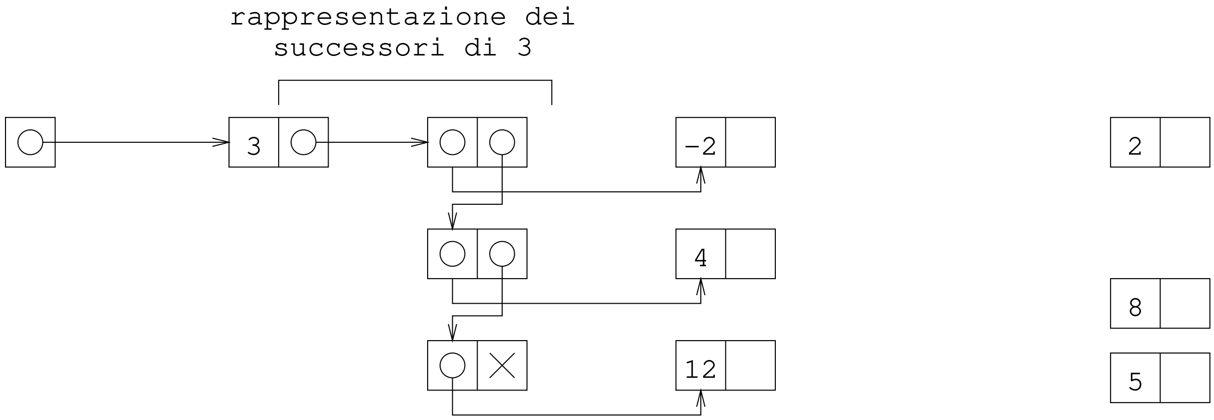
Per rappresentare i figli della radice abbiamo bisogno di una lista con tre elementi, dato che la radice ha tre figli.



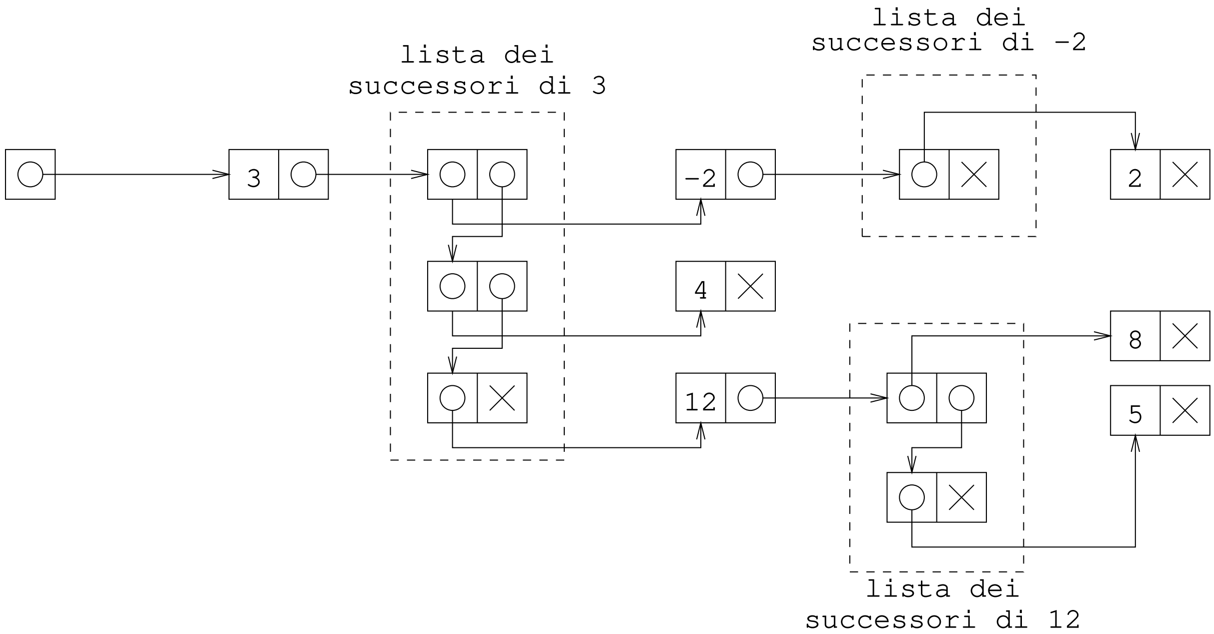
La struttura che rappresenta il nodo `3` deve indicare quali sono i suoi successori, quindi il puntatore iniziale della lista lo mettiamo in un campo di questa struttura.



La lista deve contenere gli indirizzi delle strutture che rappresentano i successori della radice. Quindi, nelle posizioni della lista che abbiamo lasciato bianche, mettiamo questi indirizzi.



L'idea è che il secondo campo della struttura `struct NodoLista` è il puntatore iniziale a una lista, i cui elementi sono, invece che interi, gli indirizzi delle strutture che rappresentano i figli della struttura. La stessa cosa va fatta per tutti gli altri nodi: il nodo `-2` ha un solo figlio, quindi avremo una lista con un solo nodo. Per il nodo `4`, che non ha figli, si usa la lista vuota, ossia `NULL`.



Anche se tutte le strutture della figura hanno due elementi, non sono tutte uguali. Infatti, la struttura che rappresenta un nodo ha come campi un intero (il valore del nodo) e un puntatore a una struttura dell'altro tipo (questo è il puntatore iniziale della lista che contiene i figli del nodo).

```
struct NodoAlbero {
  int info;
  struct NodoLista *figli;
};

typedef struct NodoAlbero *TipoAlbero;
```

Le strutture delle lista sono invece composte di due puntatori. Il primo è un puntatore a uno dei figli, ossia è l'indirizzo di una struttura di quelle di sopra. Il secondo campo è invece il solito campo next, ossia un puntatore al prossimo elemento della lista (ossia contiene l'indirizzo della prossima struttura).

```
struct NodoLista {
  struct NodoAlbero *figlio;
  struct NodoLista *next;
};
```

Si può anche vedere questa dichiarazione come una variante della definizione di struttura standard delle liste, in cui al posto del campo `info` si è messo un puntatore.

Stampa di tutti i nodi

L'esercizio consiste nella stampa di tutti i nodi di un albero generale, rappresentato usando liste per i figli.

L’algoritmo ad alto livello se è lo stesso degli alberi binari e degli alberi generali rappresentati con vettori.

1. Se l’albero è vuoto, esci.
2. Altrimenti, stampa la radice, e poi i sottoalberi.

Nel caso di alberi generali, la frase ‘‘stampa i sottoalberi’’ si può tradurre in un ciclo del tipo: per ogni sottoalbero, stampalo. Abbiamo quindi il seguente algoritmo raffinato, che è diverso da quello degli alberi binari, ma è lo stesso degli alberi generali rappresentati con vettori dinamici.

1. Se l’albero è vuoto, esci.
2. Altrimenti,
   1. stampa la radice;
   2. per ogni sottoalbero, stampalo.

I sottoalberi sono gli alberi che hanno come radice uno dei figli. Dal punto di vista della implementazione nel linguaggio, un albero è l’indirizzo della struttura che rappresenta la radice dell’albero. Nel nostro caso, questi indirizzi sono memorizzati nella lista dei figli. Si tratta quindi di fare una scansione della lista dei figli della radice, effettuando la chiamata ricorsiva per ognuno di essi.

```
void Stampa(TipoAlbero a) {
  TipoLista l;

  if(a==NULL)
    return;

  printf("%d ",a->info);

  l=a->figli;

  while(l!=NULL) {
    Stampa(l->figlio);
    l=l->next;
  }
}
```

Alberi random e stampa parentetica

Le seguenti due funzioni generano un albero random di un certo livello massimo. Si possono usare per generare alberi di test per verificare se le funzioni scritte sono corrette.

```
TipoAlbero GeneraAlberoRic(int n) {
  TipoAlbero a;
  TipoLista l;
  int i;
  int numfigli;

  a=malloc(sizeof(struct NodoAlbero));
  a->info=rand()%100-30;
  a->figli=NULL;

  if(n>0)
    numfigli=rand()%5;
  else
    numfigli=0;
```

```
  for(i=0; i<numfigli; i++) {
    l=malloc(sizeof(struct NodoLista));
    l->figlio=GeneraAlberoRic(n-1);
    l->next=a->figli;
    a->figli=l;
  }

  return a;
}

TipoAlbero GeneraAlbero(int n) {
  TipoAlbero a;

  if(rand()%(n+1)==0)
    return NULL;

  return GeneraAlberoRic(n);
}
```

La stampa parentetica di un albero è simile a quella che si fa nel caso della rappresentazione con vettori. A parte le parentesi, si tratta comunque della stampa dei nodi, stampando prima la radice e poi i figli. L'unica differenza è che i figli di un nodo si trovano con una scansione di una lista invece che con un ciclo su un vettore.

```
void StampaAlbero(TipoAlbero a) {
  int i;

  if(a==NULL) {
    printf("()");
    return;
  }

  printf("( %d ",a->info);
  for(i=0; i<a->numfigli; i++)
    StampaAlbero(a->figli[i]);
  printf(" ) ");
}
```

Presenza in un albero

Rivediamo ora l'esercizio di verifica se un albero contiene un elemento, in cui però l'albero viene rappresentato usando le liste invece dei vettori.

La prima versione dell'algoritmo è sempre la stessa: se l'albero è vuoto, non contiene l'elemento; se la radice contiene l'elemento, si ritorna `TRUE`; altrimenti, si fa la verifica su tutti i sottoalberi, ritornando `TRUE` se almeno una di queste verifiche ritorna `TRUE`.

L'unica differenza rispetto al caso della verifica di presenza con alberi rappresentati con liste è che ora la scansione dei figli di un nodo, che si usa per fare la ricerca sui sottoalberi, va fatta come la scansione di una lista invece della scansione di un vettore. In altre parole, per fare una stessa operazione su tutti i figli di un albero `a`, il ciclo viene fatto come la scansione della lista `a->figli`:

```
l=a->figli;

while(l!=NULL) {
  // operazione sul figlio, che e' l->figlio
  l=l->next;
}
```

Nel nostro caso, occorre fare la chiamata ricorsiva sul figlio `l->figlio`, ritornando `TRUE` se la chiamata ritorna `TRUE`. Se si arriva alla fine del ciclo, vuol dire che tutte le chiamate ricorsive hanno ritornato `FALSE`, e quindi l'elemento non si trova nell'albero.

```
int Presenza(TipoAlbero a, int e) {
  TipoLista l;

  if(a==NULL)
    return FALSE;

  if(a->info==e)
    return TRUE;

  l=a->figli;
  while(l!=NULL) {
    if(Presenza(l->figlio, e))
      return TRUE;

    l=l->next;
  }

  return FALSE;
}
```

Si noti che l'algoritmo coincide con quello della rappresentazione con liste, e l'unica cosa che cambia è il modo in cui si fa la scansione dei figli di un nodo. Per il resto, la funzione è la stessa.

Somma delle foglie

Rivediamo ora l'esercizio di somma delle foglie di un albero quando questo è rappresentato usando liste per i figli di un nodo.

L'algoritmo è sempre lo stesso: l'albero vuoto ha somma zero, l'albero composto da un solo nodo ha come somma delle foglie il valore del nodo, mentre negli altri casi va fatta la somma su tutti i sottoalberi.

Quando l'insieme dei figli è rappresentato con liste, la somma si trova facendo la chiamata ricorsiva sui figli, che si trovano facendo la scansione della lista `a->figli`.

```
int SommaFoglie(TipoAlbero a) {
  TipoLista l;
  int somma;

  if(a==NULL)
    return 0;

  if(a->figli==NULL)
    return a->info;

  somma=0;
```

```
  l=a->figli;
  while(l!=NULL) {
    somma+=SommaFoglie(l->figlio);

    l=l->next;
  }

  return somma;
}
```