Le Stringhe

- Una **stringa**, in informatica, è una sequenza di caratteri.
- In C una stringa viene rappresentata come un array contenente i suoi caratteri, seguiti dal *carattere speciale di fine stringa*: `'\0'`.
- **Esempi:**

```
char stringa1[10] = {'p', 'i', 'p', 'p', 'o', '\0'};
char non_stringa1[2] = {'p', 'i'};
```

  Il secondo array di caratteri non è una stringa (non termina con `'\0'`).
- Si può denotare una *costante* di tipo stringa (o *stringa letterale*) scrivendone i caratteri tra doppi apici (senza carattere di fine stringa).
  Esempio: `"pippo"`, `"corso di Informatica"`.
- Costanti di tipo stringa sono trattate come puntatori a caratteri.

```
1    char *p = "abc";
2    printf("%c%c\n", *p, *(p+1));
```

  provoca la stampa sullo schermo di `a` e `b`, poiché il puntatore `p` viene inizializzato con l'indirizzo del primo elemento dell'array *costante* di *4* caratteri utilizzato per la memorizzazione di `"abc"`.

Attenzione:

- non confondere costanti di tipo stringa e costanti di tipo carattere.
  $\Longrightarrow$ `"a"` e `'a'` sono diversi. Il primo è un array di 2 elementi, i cui valori sono il carattere `'a'` e il carattere speciale `'\0'`.
- Le stringhe costanti non possono essere modificate. **Esempio:**

```
1    char *p = "abc";
2    *p='x';
3    printf("p: %s\n",p);
```

  Stampa: `p: abc`, quindi la stringa non viene modificata.
- I tre comandi che seguono sono del tutto equivalenti:

```
1    char stringa[] = "abc";
2    char stringa[4] = "abc";
3    char stringa[] = {'a', 'b', 'c', '\0'};
```

- La stringa `"abc"` viene usata per inizializzare il vettore `stringa`. Quest'ultimo non è un vettore costante e può essere modificato
  $\Longrightarrow$ `*stringa = 'x'` (equivalente a `stringa[0]='x'`) è lecito.

Ingresso/uscita di stringhe

Stampa di una stringa

- Si deve utilizzare la specifica di formato "\%s". Stampa tutti i caratteri fino al primo '\0' escluso.
- **Esempio:**

```
1 char stringa[] = "abc";
2 char stringa1[] = {'a', '\0', 'b', 'c', '\0'};
3 char non_stringa[] = {'a', 'b'};
4 printf("%s\n", "pippo");
5 printf("%s\n", stringa);
6 printf("%s\n", stringa1);
7 printf("%s\n", non_stringa);
```

```
pippo
abc
a
ab?????...
```

Dove ????? sta per una sequenza di caratteri non significativi.

Lettura di una stringa

- Si deve utilizzare la specifica di formato "\%s".
- **Esempio:**

```
1 char buffer[40];
2 scanf("%s", buffer);
```

1. Vengono letti da input i caratteri in sequenza fino a trovare il primo carattere di spaziatura (spazio, tabulazione, interlinea, ecc.).
2. I caratteri letti vengono messi dentro il vettore `buffer`.
3. Al posto del carattere di spaziatura, viene messo il carattere '\0'.

- Note:
  - il vettore *deve* essere sufficientemente grande da contenere tutti i caratteri letti
  - non si usa \&buffer ma direttamente `buffer` (questo perché `buffer` è di tipo **char***, ovvero è già un indirizzo)

Esempio:

```
1 char buffer[40];
2 printf("Digita una stringa di caratteri: ");
3 scanf("%s", buffer);
4 printf("Ecco la stringa memorizzata in buffer:  %s\n",
5       buffer);
```

```
Digita una stringa di caratteri: abcdefghi
Ecco la stringa memorizzata in buffer:  abcdefghi
```

```
Digita una stringa di caratteri: abcde fghi
Ecco la stringa memorizzata in buffer:  abcde
```


Manipolazione di stringhe

- Per manipolare una stringa bisogna accedere ai singoli caratteri singolarmente (è un vettore come tutti gli altri!).
- **Esempio:**

```
1   for (i = 0; buffer[i] != '\0'; i++) {
2   /* fai qualcosa con buffer[i], ad esempio: */
3   buffer[i]='*';
4   }
```

- **Esempio:** Uguaglianza tra due stringhe.

```
1 typedef char String20[20];
2 String20 s1, s2;
3 ...
4 if (s1==s2) ...
```

- N.B. *Non* si può usare "==" perché questo confronta i puntatori e non le stringhe.
  $\Longrightarrow$ si devono necessariamente scandire le due stringhe.

Una funzione per uguaglianza di stringhe

```
1  boolean uguali(char * str1, char * str2)
2  /* Restituisce 'true' se le due stringhe str1 e str2
3     sono uguali, 'false' altrimenti. */
4  {
5    int i = 0;
6    while (str1[i] == str2[i] && str1[i] != '\0'
7           && str2[i] != '\0')
8      i++;
9    if (str1[i] == '\0' && str2[i] == '\0')
10     return true;
11   else return false;
12 }
```


Esercizio

Lunghezza di una stringa.

```
1  int lunghezza1(char * stringa)
2    /* Restituisce la lunghezza della stringa
3       passata come parametro.  */
4  {
5    int i = 0;
6    while (stringa[i] != '\0')
7      i++;
8    return i;
9  }
```

In realtà queste funzioni, come molte altre, sono disponibili nella libreria `<string.h>`

Per utilizzarle è *necessario:* **#include** `<string.h>`

Funzioni di libreria in <string.h>

Lunghezza di una stringa

unsigned `strlen(const char *str);`

Esempio: `strlen("abc")` restituisce il valore 3 mentre `strlen("")` restituisce il valore 0.

Funzioni di confronto

int `strcmp(const char *s1, const char *s2);`

- confronta le stringhe `s1` ed `s2`
- restituisce:
  - 0 se `s1` = `s2`
  - un valore < 0 se `s1` $<$ `s2`
  - un valore > 0 se `s1` $>$ `s2`
- il confronto è quello lessicografico: i caratteri vengono confrontati uno ad uno, ed il primo carattere diverso (o la fine di una delle due stringhe) determina il risultato
- per il confronto tra due caratteri viene usato il codice numerico corrispondente

Esempio di confronto tra stringhe

```
1 char *s1 = "abc";
2 char *s2 = "abx";
3 char *s3 = "abc altro";
4 printf("%d\n", strcmp(s1, s1));
5 printf("%d\n", strcmp(s1, s2));
6 printf("%d\n", strcmp(s1, s3));
7 printf("%d\n", strcmp(s2, s1));
```

Stampa:

```
0
-1
-1
1
```

Attenzione: per verificare l'uguaglianza di due stringhe usando `strcmp()` bisogna confrontare il risultato con 0

Esempio:

```
1 if (strcmp(s1, s2) == 0)
2   printf("uguali\n");
3 else
4   printf("diverse\n");
```

La funzione `strncmp()`

`int strncmp(const char *s1, const char *s2, size_t n);`

confronta al più `n` caratteri di `s1` ed `s2`

Esempio:

```
1 char *s1 = "abc";
2 char *s3 = "abc altro";
3
4 printf("%d\n", strncmp(s1, s3, 3));
```

Stampa `0`, poiché i primi tre caratteri delle due stringhe sono uguali.

Funzioni di copia

`char *strcpy(char *dest, const char *src);`

copia la stringa `src` nel vettore `dest`

restituisce il valore puntatore `dest`

`dest` dovrebbe essere sufficientemente grande da contenere `src`

se `src` e `dest` si sovrappongono, il comportamento di `strcpy` è indefinito

Esempio:

```
1 char a[10], b[10];
2 char *x = "Ciao";
3 char *y = "mondo";
4 strcpy(a, x);    strcpy(b, y);
5 printf("%s\n", a);          printf("%s\n", b);
```

Stampa:

```
Ciao
mondo
```

char *strncpy(**char** *dest, **const** **char** *src, size_t n);

copia un massimo di `n` caratteri della stringa `src` nel vettore `dest`
(`size_t` è il tipo del valore restituito da **sizeof**, ovvero
unsigned long o **unsigned**, dipendente dal sistema)
restituisce il valore puntatore di dest
Attenzione: il carattere `'\0'` finale di `src` viene copiato *solo* se `n` è $\geq$ della lunghezza di `src+1`

Esempio:

```
1 char a[10], b[10];
2 char *x = "Ciao";
3 char *y = "mondo";
4
5 strncpy(a, x, 5);
6 printf("%s\n", a);
7 strncpy(b, y, 5);
8 b[5] = '\0';
9 printf("%s\n", b);
```


Funzioni di concatenazione

char *strcat(**char** *dest, **const** **char** *src);

accoda la stringa `src` a quella nel vettore `dest` (il primo carattere di `src` si sostituisce al `'\0'` di `dest`) e termina `dest` con `'\0'`
restituisce il valore di `dest`
`dest` dovrebbe essere sufficientemente grande da contenere tutti i caratteri di `dest`, di `src`, ed il `'\0'` finale
se `src` e `dest` si sovrappongono, il comportamento di `strcat` è indefinito

Esempio:

```
1 char a[10], b[10];
2 char *x = "Ciao",  *y = "mondo";
3 strcpy(a, x);
4 strcat(a, y);
5 printf("%s\n", a);
```

Conversione di stringhe

Convertono le strighe formate da cifre in valori interi ed in virgola mobile.

Funzioni `atoi`, `atol`, `atof`

int `atoi(const char *str);`

converte la stringa puntata da `str` in un **int**
la stringa deve contenere un numero intero valido
in caso contrario il risultato è indefinito
spazi bianchi iniziali vengono ignorati
il numero può essere concluso da un qualsiasi carattere che non è valido in un numero intero (spazio, lettera, virgola, punto, . . .)

Esempio: file `stringhe/strtonum.c`

`atoi("123")` restituisce l'intero 123
`atoi("123.45")` restituisce l'intero 123 (`".45"` viene ignorato)
`atoi("~~123.45")` restituisce l'intero 123 (`"~~"` e `".45"` ignorati)

long `atol(const char *str);`

analoga ad `atoi`, solo che restituisce un **long int**

double `atof(const char *str);`

analoga ad `atoi` ed `atol`, solo che restituisce un **double**
il numero può essere concluso da un qualsiasi carattere non valido in un numero in virgola mobile

Esempio: file `stringhe/strtonum.c`

`atof("123.45")` restituisce 123.45
`atof("1.23xx")` restituisce 1.23
`atof("1.23.45")` restituisce 1.23

Funzioni di libreria per I/O di stringhe e caratteri in <stdio.h>

Input/output di caratteri

`int getchar(void);`

legge il prossimo carattere da standard input e lo restituisce

`int putchar(int c);`

manda `c` in standard output

restituisce `EOF` (-1) se si verifica un errore

Input/output di stringhe

`char *gets(char *str);`

legge i caratteri da standard input e li inserisce nel vettore `str`

la lettura termina con un `'\n'` o un `EOF`, che *non* viene inserito nel vettore

la stringa nel vettore viene terminata con `'\0'`

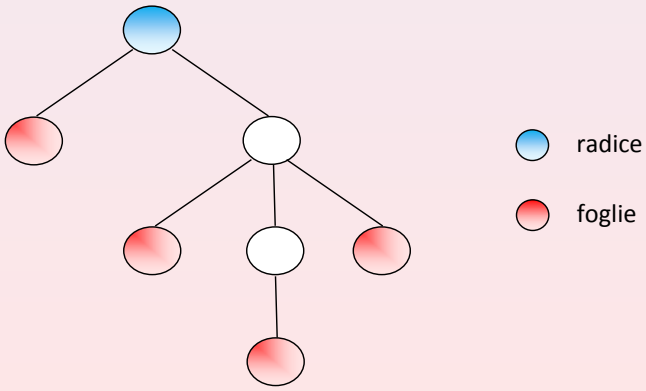
Attenzione: non c'è alcun modo di limitare la lunghezza della sequenza immessa $\Longrightarrow$ vettore allocato potrebbe non bastare

`int puts(const char *str);`

manda la stringa `str` in standard output (inserisce un `'\n'` alla fine)

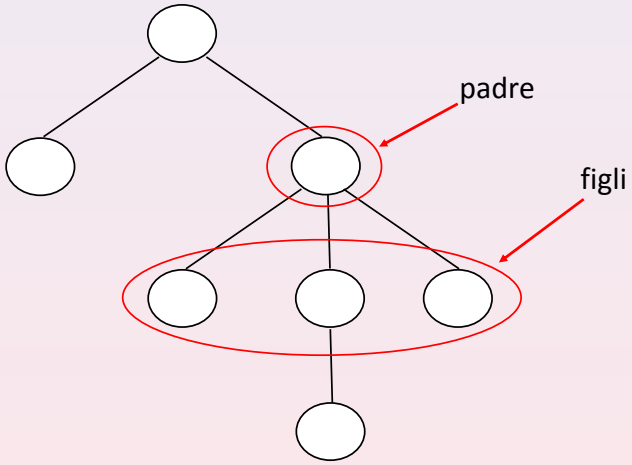
Alberi e alberi binari

- Un albero è un caso particolare di grafo
  - È costituito da un insieme di nodi collegati tra di loro mediante archi
  - Gli archi sono orientati (ogni arco esce da un nodo origine ed entra in un nodo destinazione)
  - Ogni nodo ha al più un arco entrante ed esiste un nodo, la radice dell'albero, che non ha archi entranti
  - I nodi senza archi uscenti sono detti foglie dell'albero
- Questi vincoli ci consentono di rappresentare graficamente un albero come una struttura gerarchica.

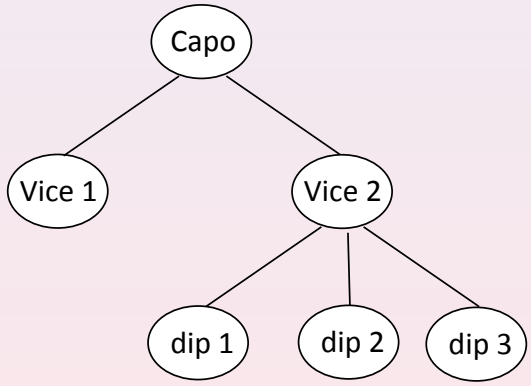


Se un arco esce da un nodo A ed entra in un nodo B si dice che

- A è il padre di B
- B è un figlio di A



- Gli alberi sono utili per rappresentare informazioni che hanno una struttura gerarchica (es. alberi genealogici, organigramma di aziende, ecc.)
- Ai nodi si associano le informazioni di interesse, dette etichette



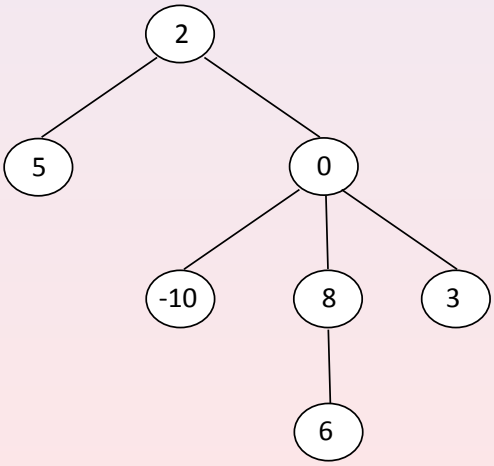
- Nel seguito faremo sempre riferimento ad alberi etichettati e identificheremo un nodo con la sua etichetta, laddove ciò non crei ambiguità!

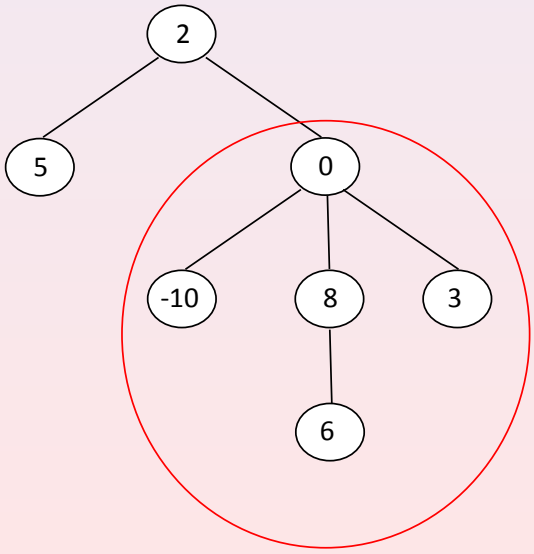
- Dato un albero, definiamo:
  - Cammino nell'albero una sequenza di nodi, in cui ogni nodo è figlio del nodo che lo precede nella sequenza
  - Livello (o profondità) di un nodo, la sua distanza dalla radice (quanto "in basso" si trova nell'albero).

    Il livello di un nodo può essere definito induttivamente come segue:
    - la radice ha livello 0
    - se un nodo ha livello i, allora i suoi figli hanno livello $i+1$
  - Livello k di un albero, come l'insieme di tutti e soli i nodi di livello k.
  - Altezza (o profondità) di un albero come la profondità massima che può avere un nodo dell'albero.
- Osserviamo che ogni nodo di un albero è a sua volta radice di un (sotto) albero

Un albero con etichette intere

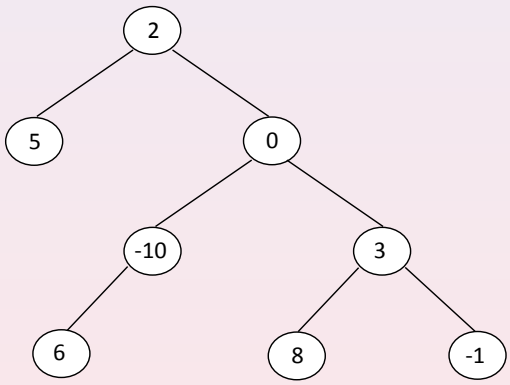
- Alcuni cammini: $\langle 0, 3\rangle$, $\langle 2, 0, 8, 6\rangle$
- Livello: il nodo 0 ha livello 1, il nodo 6 ha livello 3, ...
- Livello 1 dell'albero: {5, 0}
  Livello 2 dell'albero: {-10, 8, 3}
- Un sottoalbero





Alberi binari

- Un albero binario è un albero in cui ogni nodo ha al più 2 figli, detti rispettivamente figlio sinistro e figlio destro



- Per quanto osservato prima il figlio sinistro è a sua volta radice di un sottoalbero binario, detto sottoalbero sinistro. Analogamente per il figlio destro.

Rappresentazione collegata degli alberi binari

- Come possiamo rappresentare in C alberi, e in particolare alberi binari?
- Utilizziamo una rappresentazione collegata simile a quella delle liste
- L'elemento fondamentale è il nodo, che
  - ha un'etichetta
  - è collegato ai sottoalberi sinistro e destro (eventualmente vuoti)
- Possiamo definire una struttura con 3 campi:
  - l'etichetta
  - il puntatore al sottoalbero sinistro
  - il puntatore al sottoalbero destro
- In pratica, rappresentiamo mediante puntatori gli archi che collegano un nodo ai suoi sottoalberi.


```
struct nodoAlberoBinario
{
   TipoInfoAlbero label;
   struct nodoAlberoBinario *left;
   struct nodoAlberoBinario *right;
 }

typedef struct nodoAlberoBinario NodoAlbero;

typedef NodoAlbero *AlberoBinario;
```

- Il tipo `TipoInfoAlbero` definisce il tipo delle etichette dei nodi. Negli esempi
  `typedef int TipoInfoAlbero;`
- Si noti come, analogamente alle liste, un albero binario sia rappresentato dal puntatore al nodo radice

Un esempio

- Vediamo come primo esempio l'implementazione di una procedura che, dato un albero binario di interi, raddoppia l'etichetta di nodi con etichetta pari
- L'implementazione più naturale è di tipo ricorsivo, osservando che un albero binario può essere definito induttivamente come segue:
  - L'albero vuoto è un albero binario
  - Se lt e rt sono alberi binari e n è un intero, allora l'albero con radice un nodo etichettato con n, sottoalbero sinistro lt e sottoalbero destro rt, è un albero binario

```
void raddoppiaPari {AlberoBinario bt)
  if (bt != NULL)
    {
      if even(bt -> label)
         bt -> label = 2 * (bt -> label);
      raddoppiaPari(bt -> left);
      raddoppiaPari(bt -> right);
    }
```


Osservazioni

- Nell'esempio precedente abbiamo scelto di operare analizzando, nell'ordine:
  - la radice dell'albero
  - il sottoalbero sinistro
  - il sottoalbero destro
- Poiché l'analisi dei sottoalberi sinistro e destro avviene utilizzando la stessa procedura ricorsiva, anche la loro analisi opera allo stesso modo (prima la radice, poi il sottoalbero sx, quindi il sottoalbero dx . . .)
- Avremmo potuto procedere diversamente, ad esempio:

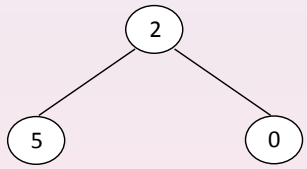
```
void raddoppiaPari {AlberoBinario bt)
  {if (bt != NULL)
    {
      raddoppiaPari(bt -> left);
      raddoppiaPari(bt -> right);
      if even(bt -> label)
         bt -> label = 2 * (bt -> label);
    }}
```

Osservazioni

- Nell'esempio precedente abbiamo scelto di operare analizzando, nell'ordine:
  - la radice dell'albero
  - il sottoalbero sinistro
  - il sottoalbero destro
- Poiché l'analisi dei sottoalberi sinistro e destro avviene utilizzando la stessa procedura ricorsiva, anche la loro analisi opera allo stesso modo (prima la radice, poi il sottoalbero sx, quindi il sottoalbero dx . . .)
- Avremmo potuto procedere diversamente, ad esempio:

```
void raddoppiaPari {AlberoBinario bt)
  {if (bt != NULL)
    {
      raddoppiaPari(bt -> left);
      if even(bt -> label)
         bt -> label = 2 * (bt -> label);
      raddoppiaPari(bt -> right);
    }}
```


- Nel caso della procedura vista, l'ordine è ininfluente ai fini degli effetti finali: tutte le etichette pari vengono comunque raddoppiate
- Ciò non è il caso, però, per altre operazioni
- **Esempio:** : Stampare la sequenza di etichette dell'albero



Possiamo ottenere sequenze diverse:

- 2, 5, 0
- 5, 2, 0
- 5, 0, 2
- . . .

Visita di un albero

- Visitare un albero significa analizzare in sequenza tutti i suoi nodi.
- Molte operazioni sugli alberi possono essere viste come varianti di visite degli stessi.
- Possiamo avere diversi tipi di visita, che differiscono per l'ordine in cui vengono visitati i nodi.
  - Visite depth-first (in profondità)
    - visita anticipata: si analizza la radice, poi si effettua la visita anticipata del sottoalbero sinistro e infine si effettua la visita anticipata del sottoalbero destro
    - visita simmetrica: si effettua la visita simmetrica del sottoalbero sinistro, poi si analizza la radice e infine si effettua la visita simmetrica del sottoalbero destro
    - visita posticipata: si effettua la visita posticipata del sottoalbero sinistro, poi si effettua la visita posticipata del sottoalbero destro, e infine si analizza la radice
  - visita breadth-first (per livelli): si visita prima la radice (livello 0), poi si visitano tutti i nodi di livello 1, poi tutti i nodi di livello 2, ...

Implementazione delle visite

- Vediamo l'implementazione ricorsiva delle visite in profondità (generalizzazione dell'esempio visto), assumendo data una funzione col seguente prototipo
  `void AnalizzaNodo(TipoInfoAlbero)`
- N.B. Se l'analisi del nodo può comportare la modifica dell'etichetta, abbiamo bisogno di una procedura con prototipo
  `void AnalizzaNodo(TipoInfoAlbero *)`.
  Di conseguenza la chiamata nella procedura di visita si modifica in
  `AnalizzaNodo(&(bt -> label))`

Implementazione delle visite (cont.)

Visita simmetrica

```
void visitaSimmetrica {AlberoBinario bt)
 { if (bt != NULL)
    {
      visitaSimmetrica(bt -> left);
      AnalizzaNodo(bt -> label);
      visitaSimmetrica(bt -> right);
    }}
```


Implementazione delle visite (cont.)

Visita anticipata

```
void visitaAnticipata {AlberoBinario bt)
 { if (bt != NULL)
    {
      AnalizzaNodo(bt -> label);
      visitaAnticipata(bt -> left);
      visitaAnticipata(bt -> right);
    }}
```

Implementazione delle visite (cont.)

Visita posticipata

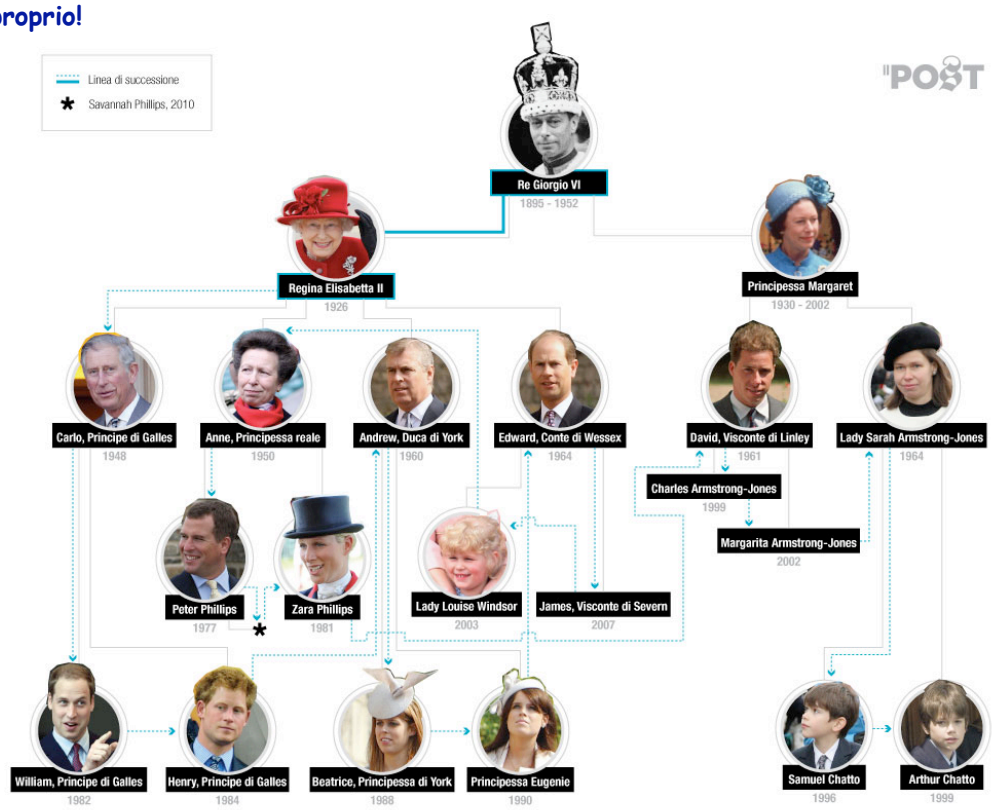
```
void visitaPosticipata {AlberoBinario bt)
  {if (bt != NULL)
    {
      visitaPosticipata(bt -> left);
      visitaPosticipata(bt -> right);
      AnalizzaNodo(bt -> label);
    }}
```


Visita anticipata: una curiosità

La visita in ordine anticipato di un albero genealogico corrisponde all'antichissimo algoritmo usato per determinare l'ordine di successione al titolo in una famiglia nobile o regale.

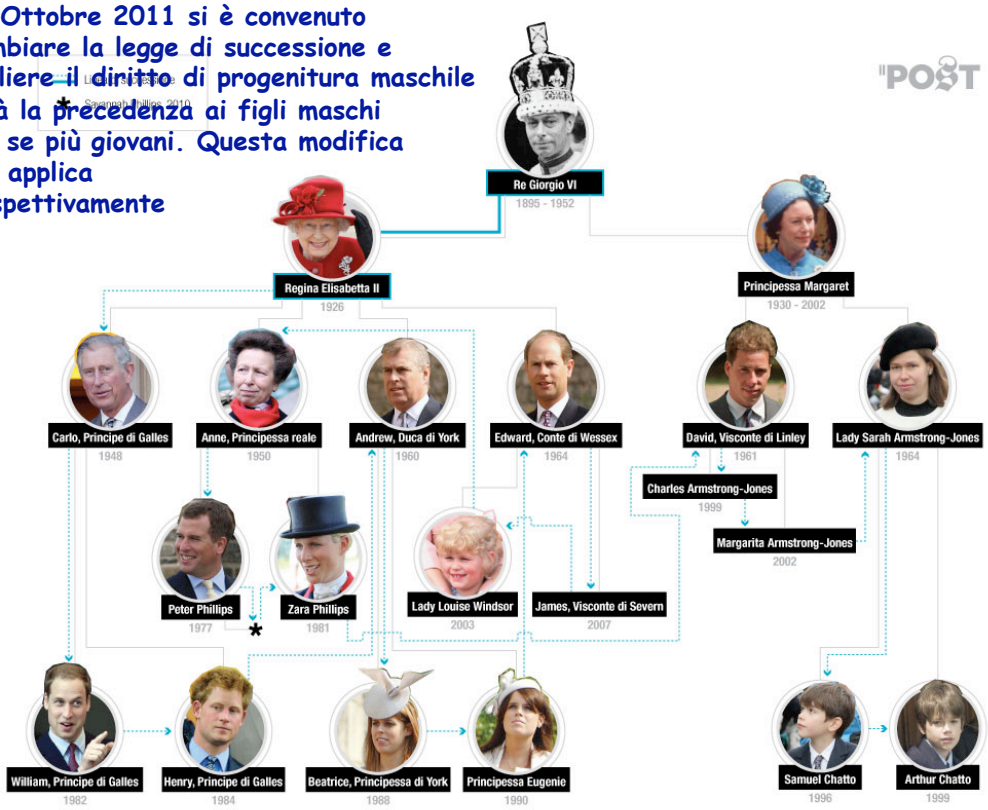
Visita anticipata: una curiosità (2)

Non proprio!



Visita anticipata: una curiosità (3)

Il 28 Ottobre 2011 si è convenuto di cambiare la legge di successione e di togliere il diritto di progenitura maschile che dà la precedenza ai figli maschi anche se più giovani. Questa modifica non si applica retrospettivamente



Esercizi

- Scrivere una funzione che determina se un albero contiene un nodo con una certa etichetta. Il prototipo è

  `boolean member (AlberoBinario, TipoInfoAlbero)`

- Scrivere una funzione che conta il numero di occorrenze di una certa etichetta in un albero binario.

  `int contaOccorrenze (AlberoBinario, TipoInfoAlbero)`

- Per alberi binari con etichette di tipo `int`, scrivere una funzione che calcoli la somma delle etichette
  - di tutti i nodi dell'albero
  - di tutte le foglie dell'albero

Ricerca di un'etichetta

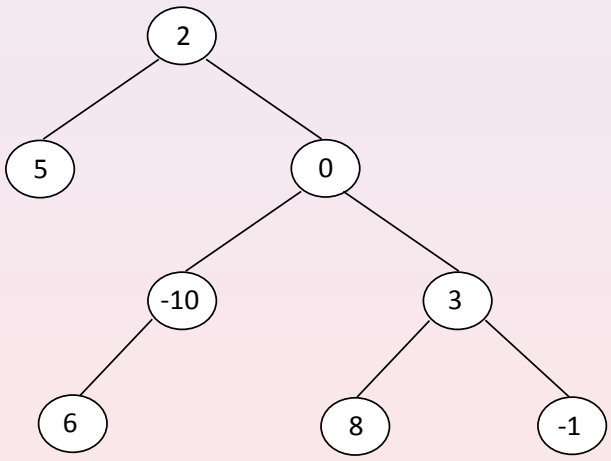
Diamo due tra le tante possibili soluzioni:

```
boolean member (AlberoBinario bt, TipoInfoAlbero etichetta)
{
   boolean risultato = false;
   if (bt != NULL)
      risultato = ((bt -> label) == etichetta) || member(bt -> left, etichetta)
      |I              member(bt -> right, etichetta);
   return risultato;
 }

boolean member (AlberoBinario bt, TipoInfoAlbero etichetta)
{
   boolean risultato = false;
   if (bt != NULL)
      if ((bt -> label)==etichetta)
         risultato = true;
      else  {
              risultato = member(bt->left, etichetta);
              if (!risultato)
                 risultato = member(bt -> right, etichetta);
            }
   return risultato;
 }
```

Anche in questo caso due tra le varie soluzioni possibili (la prima visita in ordine simmetrico, la seconda in ordine posticipato)

```
int contaOccorrenze (AlberoBinario bt, TipoInfoAlbero etichetta)
{
   int risultato = 0;
   if (bt != NULL)
      {
        if ((bt -> label) == etichetta) risultato = risultato + 1;
        risultato = risultato + contaOccorrenze(bt -> left, etichetta) +
                                contaOccorrenze(bt -> right, etichetta);
      }
    return risultato;
 }

int contaOccorrenze (AlberoBinario bt, TipoInfoAlbero etichetta)
{
   int risultato = 0;
   if (bt != NULL)
      {
        risultato = contaOccorrenze(bt -> right, etichetta);
        risultato  = risultato + contaOccorrenze(bt -> left, etichetta);
        if ((bt -> label) == etichetta) risultato = risultato + 1;
      }
    return risultato;
 }
```


- **Esempio:** Dato un albero binario di interi, costruire la lista delle etichette dei suoi nodi, ottenuta visitando l'albero in ordine simmetrico
- In corrispondenza dell'albero



vogliamo dunque ottenere la lista

```
5 --> 2 --> 6 --> -10 --> 0 --> 8 --> 3 --> -1 --> //
```