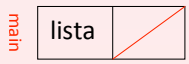
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



Ricerca di un elemento in una lista

- Ricordiamo la ricerca lineare incerta su vettori

```
i = 0;      /* indice del primo elemento */
trovato = false;

while (! trovato && i < DIM)
 {
  if (vet[i] == elem)   /* elemento corrente */
     trovato = true;
  else
     i = i + 1;
 }
```

- sostituiamo l'indice i con un puntatore alla lista che ci permette di scorrerla
- Incapsuliamo questo codice in una funzione a valori booleani

Ricerca di un elemento in una lista

Esempio: Versione Iterativa

```
boolean Ricerca(ListaDiElementi lis, TipoElementoLista  elem)
{
 boolean trovato = false;
  while (lis != NULL && ! trovato)
    {
     if  (lis->info == elem)
          trovato = true;
     else
     lis = lis->next;
    }
return trovato;
}
```

- Non c'è bisogno di un puntatore ausiliario per scorrere la lista
  $\Longrightarrow$ il passaggio per valore consente di scorrere utilizzando il parametro formale!
- Abbiamo assunto che sul tipo `TipoElementoLista` sia definito l'operatore di uguaglianza ==

Ricerca di un elemento in una lista

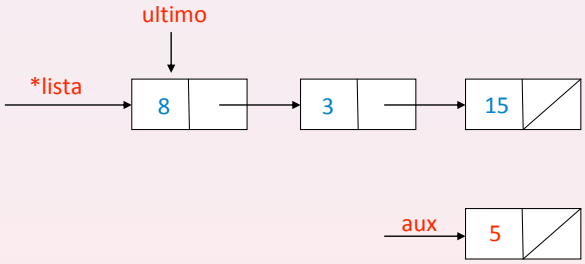
Esempio: Versione Ricorsiva

```
boolean RicercaRic(ListaDiElementi lis, TipoElementoLista  elem)
{
   if (lis == NULL)
      return false;
   else
        if (lis->info == elem) return true;
        else return RicercaRic(lis->next,elem);
  }
```

- Un elemento elem
  - non appartiene alla lista vuota
  - appartiene alla lista con testa x se elem coincide con x
  - appartiene alla lista con testa x diversa da elem e resto L se e solo se appartiene a L

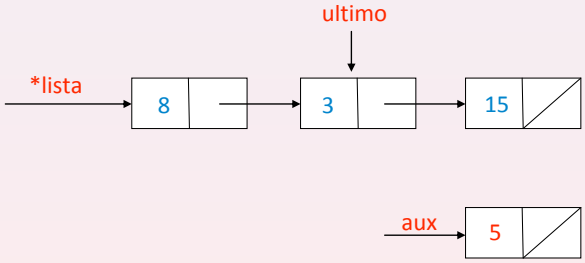
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



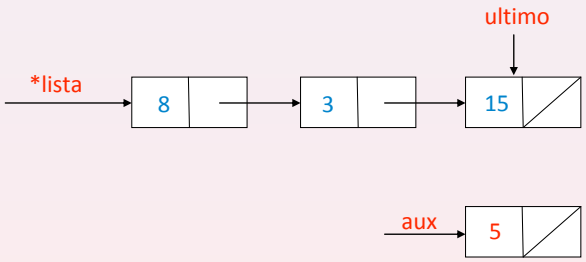
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



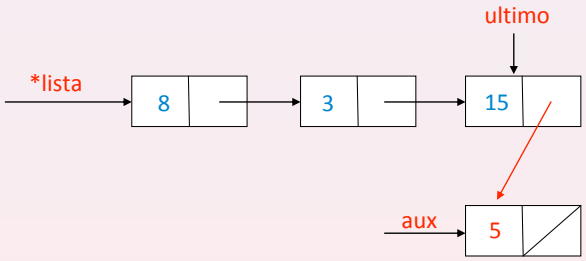
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



Codice della versione iterativa

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi ultimo;   /* puntatore usato per la scansione */
  ListaDiElementi aux;
                            /* creazione del nuovo elemento */
  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = NULL;

  if (*lista == NULL)
    *lista = aux;
  else {
    ultimo = *lista;
    while (ultimo->next != NULL)
      ultimo = ultimo->next;
                    /* concatenazione del nuovo elemento in coda alla lista */
    ultimo->next = aux;
  }
}
```


Inserimento ricorsivo di un elemento in coda

- Caratterizzazione induttiva dell'inserimento in coda

  Sia nuovaLista la lista ottenuta inserendo elem in coda a lista.

  1. se lista è vuota, allora nuovaLista è costituita dal solo `elem` (caso base)
  2. altrimenti nuovaLista è ottenuta da lista facendo l'inserimento di elem in coda al resto di lista (caso ricorsivo)

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
  else
    InserisciCodaLista(       ??       , elem);
}
```

Inserimento ricorsivo di un elemento in coda

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
  else
    InserisciCodaLista(  (*lista)->next , elem);
}
```


Inserimento ricorsivo di un elemento in coda

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
  else
    InserzioneInCoda(&((*lista)->next), elem);
}
```

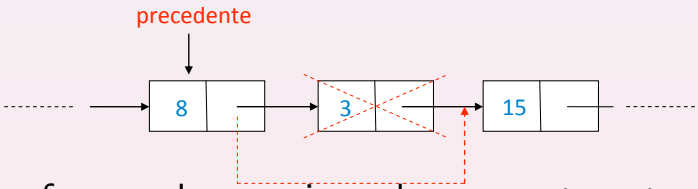
N.B.: Potremmo qui sostituire `*lista == NULL` con `ListaVuota(*lista)`

Cancellazione della prima occorrenza di un elemento

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla
- altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
  1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo
     $\Longrightarrow$ passaggio per indirizzo!!
  2. l'elemento non è né il primo né l'ultimo: si aggiorna il campo next dell'elemento che precede quello da cancellare in modo che punti all'elemento che segue
  3. l'elemento è l'ultimo: come (2), solo che il campo next dell'elemento precedente viene posto a `NULL`
- in tutti e tre i casi bisogna liberare la memoria occupata dall'elemento da cancellare

Osservazioni:

- per poter aggiornare il campo next dell'elemento precedente, bisogna fermare la scansione sull'elemento precedente (e non su quello da cancellare)



- per fermare la scansione dopo aver trovato e cancellato l'elemento, si utilizza una sentinella booleana
- Seguendo questa idea, fare per esercizio la versione iterativa della cancellazione.

Versione iterativa:

```
void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi prec;    /* puntatore all'elemento precedente */
  ListaDiElementi corr;    /* puntatore all'elemento corrente */
  boolean trovato;        /* usato per terminare la scansione */

  if (*lista != NULL)
    if ((*lista)->info==elem)
        { /* cancella il primo elemento */
            CancellaPrimo(lista);
        }
    else    /* scansione della lista e cancellazione dell'elemento */
            prec = *lista; corr = prec->next; trovato = false;
            while (corr != NULL && !trovato)
              if (corr->info == elem)
                {   /* cancella l'elemento */
                    trovato = true;         /* provoca l'uscita dal ciclo */
                    prec->next = corr->next;
                    free(corr);  }
              else  {
                    prec = prec->next;    /* avanzamento dei due puntatori */
                    corr = corr->next;   }
}
```


Versione ricorsiva:

```
void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  if (*lista != NULL)
    if ((*lista)->info== elem)
        { /* cancella il primo elemento */
            CancellaPrimo(lista);
        }
    else   /* cancella elem dal resto */
        CancellaElementoLista(&((*lista)->next), elem);
}
```

Cancellazione di tutte le occorrenze di un elemento

Versione iterativa

- analoga alla cancellazione della prima occorrenza
- però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione
- ci si ferma solo quando si è arrivati alla fine della lista

  $\Longrightarrow$ non serve la sentinella booleana per fermare la scansione

Cancellazione di tutte le occorrenze di un elemento

Caratterizzazione induttiva

Sia ris la lista ottenuta cancellando tutte le occorrenze di elem da lista. Allora:

1. se lista è la lista vuota, allora ris è la lista vuota (caso base)
2. altrimenti, se il primo elemento di lista è uguale ad elem, allora ris è ottenuta da lista cancellando il primo elemento e tutte le occorrenze di elem dal resto di lista (caso ricorsivo)
3. altrimenti ris è ottenuta da lista cancellando tutte le occorrenze di elem dal resto di lista (caso ricorsivo)

Esercizio

Implementare le due versioni

Versione iterativa

```
void CancellaTuttiLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi prec;    /* puntatore all'elemento precedente */
  ListaDiElementi corr;    /* puntatore all'elemento corrente */
  boolean trovato = false;
  while ((*lista != NULL) && ! trovato)  /* cancella le occorrenze */
    if ((*lista)->info!=elem)           /* di elem in testa       */
        trovato = true;
    else  CancellaPrimo(lista);

   if (*lista != NULL)
        {
            prec = *lista; corr = prec->next;
            while (corr != NULL)
              if (corr->info == elem)
                {   /* cancella l'elemento */
                    prec->next = corr->next;
                    free(corr);
                    corr = prec->next;}
              else  {
                    prec = prec->next;    /* avanzamento dei due puntatori */
                    corr = corr->next;   }
        }
}
```


Versione ricorsiva

```
void CancellaTuttiLista(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi aux;

  if (*lista != NULL)
    if ((*lista)->info==elem)
    {                                  /* cancellazione del primo elemento */
      CancellaPrimo(lista);
                               /* cancellazione di elem dal resto della lista */
      CancellaTuttiLista(lista, elem);
    }
    else
      CancellaTuttiLista(&((*lista)->next), elem);
}
```

Inserimento di un elemento in una lista ordinata

- Data una lista (ad es. di interi) già ordinata (in ordine crescente), si vuole inserire un nuovo elemento mantenendo l'ordinamento.

  Versione iterativa: per **esercizio**

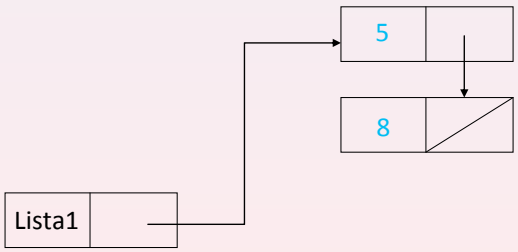
  Versione ricorsiva

- Caratterizziamo il problema induttivamente
- Sia ris la lista ottenuta inserendo l'elemento elem nella lista ordinata lista.
  1. se lista è la lista vuota, allora ris è costituita solo da elem (caso base)
  2. se il primo elemento di lista è maggiore o uguale a elem, allora ris è ottenuta da lista inserendo elem in testa (caso base)
  3. altrimenti ris è ottenuta da lista inserendo ordinatamente elem nel resto di lista (caso ricorsivo)


```
void InserzioneOrdinata(ListaDiElementi *lista, int elem)
{
if (*lista == NULL)
   InserisciTestaLista(lista, elem);
else
    if ((*lista) --> info >= elem)
        InserisciTestaLista(lista, elem);
    else
        InserzioneOrdinata(&((*lista)->next), elem);
}
```

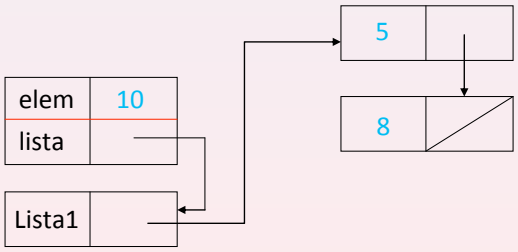
InserzioneOrdinata(&Lista1, 10)

PILA HEAP



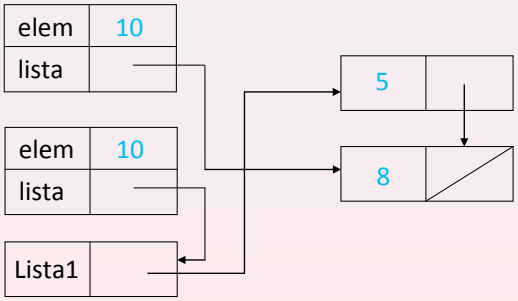
InserzioneOrdinata(&Lista1, 10)

PILA HEAP



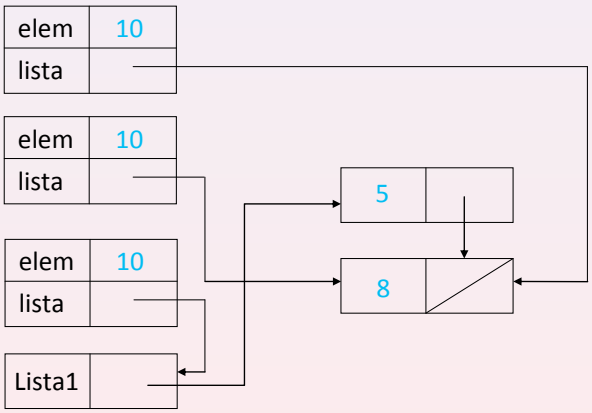
InserzioneOrdinata(&Lista1, 10)

PILA HEAP

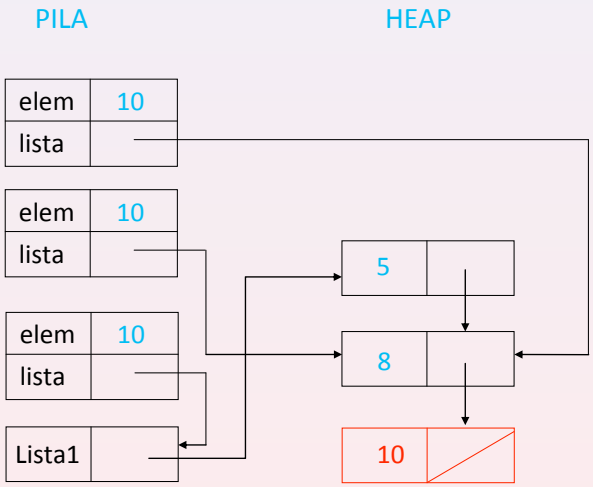


InserzioneOrdinata(&Lista1, 10)

PILA HEAP

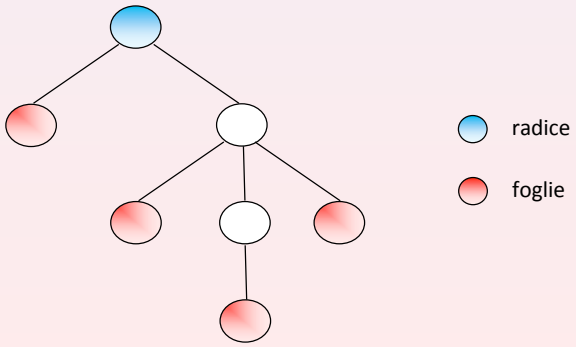


InserzioneOrdinata(&Lista1, 10)



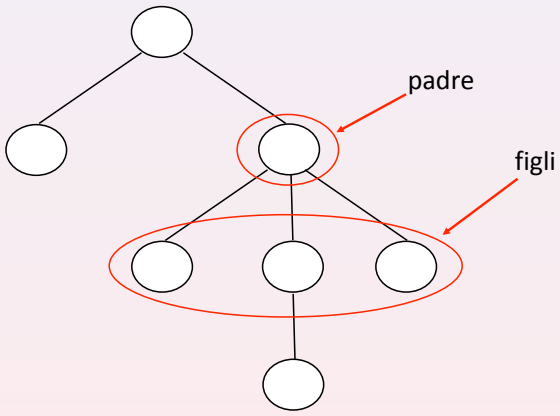
Alberi e alberi binari

- 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.

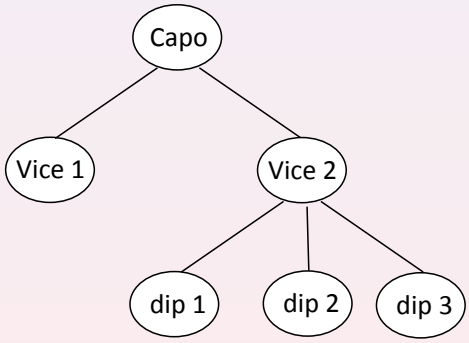


Se un arco esce da un nodo A ed entra in un nodo B si dice che

- 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



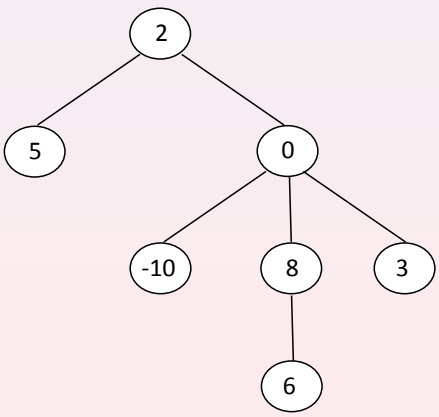
- Nel seguito faremo sempre riferimento ad alberi etichettati e identificheremo un nodo con la sua etichetta, laddove ciò non crei ambiguità!

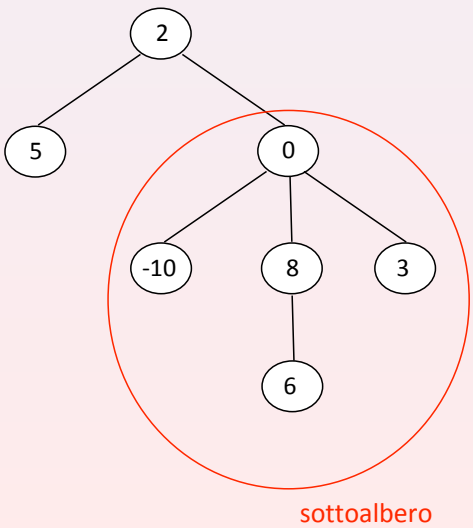
- Dato un albero, definiamo:
  - Cammino nell'albero una sequenza di nodi, in cui ogni nodo è figlio del nodo che lo precede nella sequenza
  - Livello (o profondità) di un nodo, la sua distanza dalla radice (quanto "in basso" si trova nell'albero).

    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

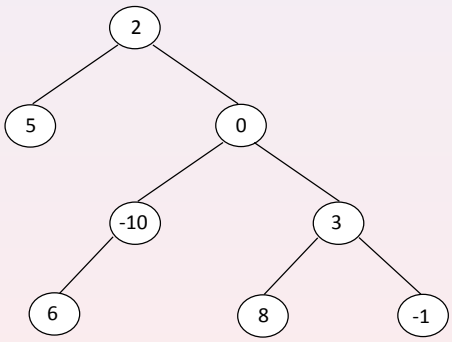
- Alcuni cammini: $\langle 0, 3\rangle$, $\langle 2, 0, 8, 6\rangle$
- Livello: il nodo 0 ha livello 1, il nodo 6 ha livello 3, ...
- Livello 1 dell'albero: {5, 0}
  Livello 2 dell'albero: {-10, 8, 3}
- Un sottoalbero





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:

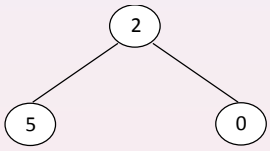
```
void raddoppiaPari {AlberoBinario bt)
  {if (bt != NULL)
    {
      raddoppiaPari(bt -> left);
      raddoppiaPari(bt -> right);
      if even(bt -> label)
         bt -> label = 2 * (bt -> label);
    }}
```


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

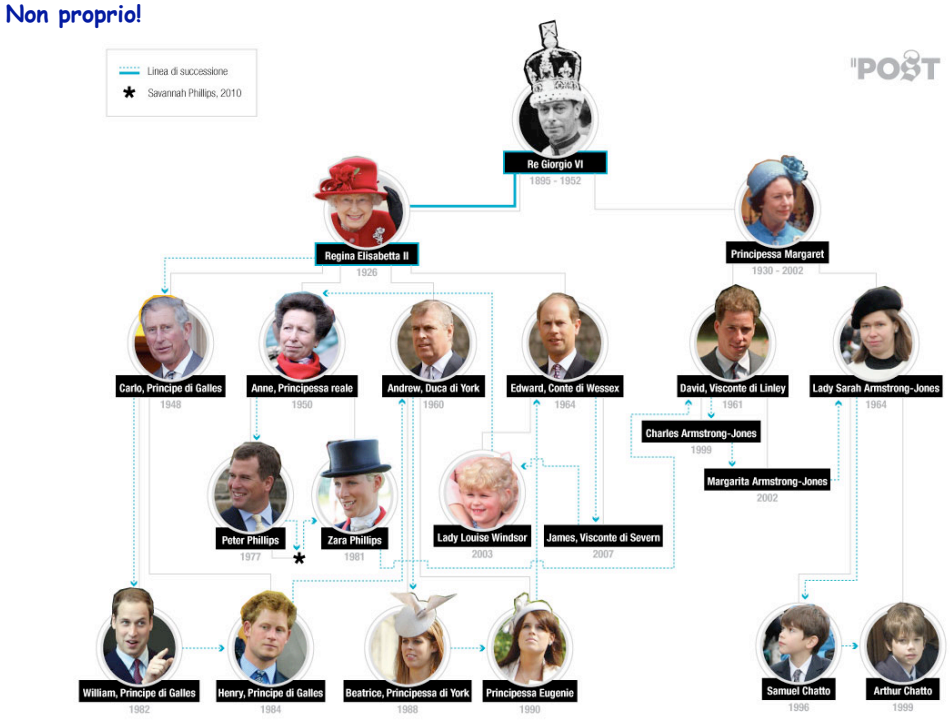
```
void visitaPosticipata {AlberoBinario bt)
  {if (bt != NULL)
    {
      visitaPosticipata(bt -> left);
      visitaPosticipata(bt -> right);
      AnalizzaNodo(bt -> label);
    }}
```

Visita anticipata: una curiosità

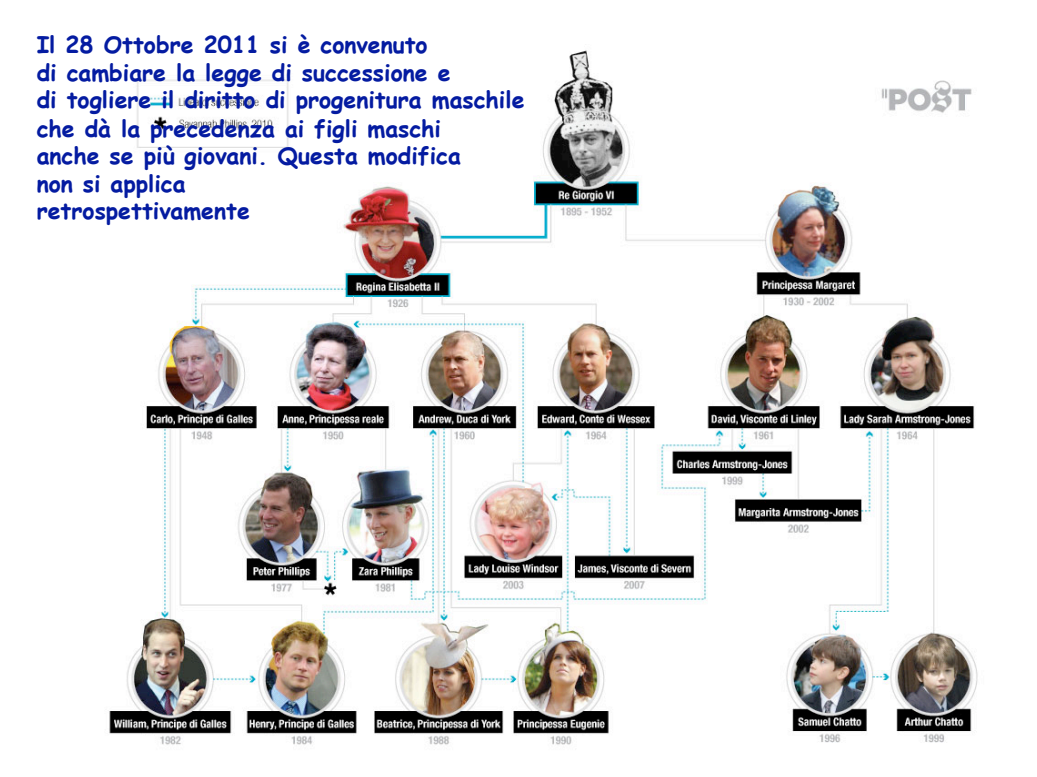
La visita in ordine anticipato di un albero genealogico corrisponde all'antichissimo algoritmo usato per determinare l'ordine di successione al titolo in una famiglia nobile o regale.

Visita anticipata: una curiosità (2)

Non proprio!



Visita anticipata: una curiosità (3)



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

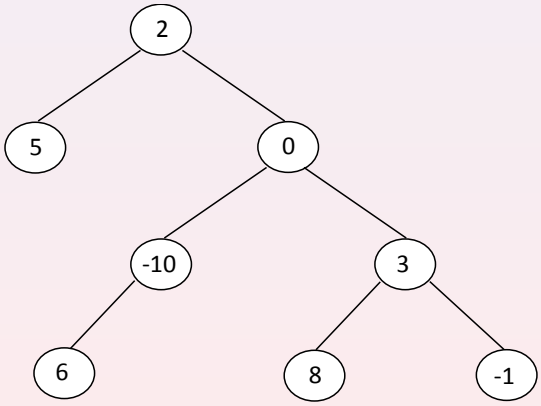
Diamo due tra le tante possibili soluzioni:

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

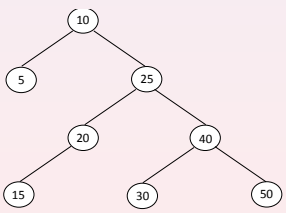


vogliamo dunque ottenere la lista

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


Alberi binari di ricerca

- Sono alberi binari per i quali è definito un ordinamento sull'informazione contenuta nelle etichette (ad esempio interi, stringhe, ma anche strutture complesse). Nel seguito, date due etichette N ed M diciamo semplicemente N è minore di M, N è maggiore di M ...
- Un albero è un albero binario di ricerca se per ogni nodo etichettato N dell'albero valgono le seguenti proprietà:
  - l'etichetta di ogni nodo nel sottoalbero sinistro di N è minore di N
  - l'etichetta di ogni nodo nel sottoalbero destro di N è maggiore di N



- Osserviamo subito che visitando un albero binario di ricerca in ordine simmetrico otteniamo una lista di elementi ordinati in senso crescente
- L'operazione di ricerca di un'etichetta è chiaramente semplice ed efficiente
- l'inserimento o la cancellazione di elementi richiedono invece attenzione, perché non devono alterare le proprietà dell'albero.