I tipi di dato astratti

La nozione di tipo di dato astratto

Quando si affrontano problemi complessi è necessario procedere in due fasi separate.

1. **specifica** dell'algoritmo
2. **implementazione** dell'algoritmo in un linguaggio di programmazione

La fase (1) coinvolge la specifica di due aspetti:

- specifica dei dati che l'algoritmo deve manipolare
- specifica delle operazioni da eseguire per realizzare l'algoritmo

La fase (2) richiede di implementare nel LDP (realizzare):

- i dati, attraverso le strutture di dati del LDP
- le operazioni, attraverso le istruzioni del LDP

Per esprimere i dati in maniera independente dai vincoli imposti dal LDP si utilizzano i tipi di dato astratti.

Un **tipo di dato astratto** è costituito da tre componenti:

- il **dominio di interesse**, che rappresenta l'insieme di valori che costituiscono il tipo astratto, ed eventuali **altri domini**, che rappresentano altri valori coinvolti
- un insieme di **costanti**, che denotano valori del dominio di interesse
- un insieme di **operazioni** che si effettuano sugli elementi del dominio di interesse, utilizzando eventualmente elementi degli altri domini.

Tutte e tre le componenti sono indipendenti dalla rappresentazione e dall'uso del tipo stesso nei LDP (la fase di specifica è indipendente dai vincoli imposti dal LDP).

Concettualizzazione (o specifica) dei tipi di dato astratti

La concettualizzazione di un tipo di dato astratto richiede di specificare separatamente le tre componenti:

- gli insiemi che rappresentano il **dominio di interesse** e gli eventuali **altri domini**
- l'insieme delle **costanti** del tipo
- le **operazioni**, attraverso un insieme di **funzioni**:
  - la **segnatura** della funzione (ovvero il suo nome e il tipo dei parametri di ingresso ed uscita) specifica di cosa necessita un'operazione e cosa restituisce
  - il **significato** di un'operazione può venire specificato algebricamente in modo rigoroso. Noi descriveremo il significato delle operazioni in modo informale.

Esempio: Concettualizzazione del tipo astratto $Booleano$.

- dominio di interesse: $\{vero, falso\}$
- costanti: $true$ e $false$, che denotano rispettivamente $vero$ e $falso$
- operazioni:
$$\begin{aligned} and &: Booleano \times Booleano \rightarrow Booleano \\ or &: Booleano \times Booleano \rightarrow Booleano \\ not &: Booleano \rightarrow Booleano \end{aligned}$$

Esempio: Concettualizzazione del tipo astratto "insieme di lettere alfabetiche maiuscole".

- *dominio di interesse:* $InsLettere = \{\emptyset, \{A\}, \{B\}, \ldots, \{A, B, \ldots, Z\}\}$
  altri *domini:* $Lettera = \{A, B, \ldots, Z\}$
  $Booleano = \{vero, falso\}$
- *costanti:* $\emptyset$ (denota l'insieme vuoto)
- *operazioni:*
$$\begin{aligned} &VerificaInsiemeVuoto : InsLettere \rightarrow Booleano \\ &Inserisci : InsLettere \times Lettera \rightarrow InsLettere \\ &Cancella : InsLettere \times Lettera \rightarrow InsLettere \\ &VerificaAppartenenza : InsLettere \times Lettera \rightarrow Booleano \\ &Unione : InsLettere \times InsLettere \rightarrow InsLettere \\ &Intersezione : InsLettere \times InsLettere \rightarrow InsLettere \\ &\ldots \end{aligned}$$

  Le prime quattro sono le **operazioni di base**. Le altre operazioni possono essere ottenute componendo in modo opportuno le operazioni di base.

Realizzazione (o implementazione) dei tipi di dato astratti

Il tipo di dato astratto deve essere realizzato usando i costrutti del LDP:

1. rappresentazione dei **domini** usando i **tipi concreti** del LDP
2. codifica delle **costanti** attraverso i **costrutti** del LDP
3. realizzazione delle **operazioni** attraverso opportune **funzioni** del LDP

Esempio: Realizzazione del tipo astratto "insieme di lettere alfabetiche maiuscole".

1) $InsLettere$ rappresentato dal suo vettore caratteristico:

Es.: $\{B, C, Y\}$

0	1	1	0	0	$\cdots$	0	1	0
0	1	2	3	4	$\cdots$	23	24	25
A	B	C	D	E	$\cdots$	X	Y	Z

```
typedef int bool;
#define TRUE  1
#define FALSE 0

typedef char TipoLettera;
typedef bool TipoInsLettere['Z' - 'A' + 1];
```

2) Codifica della costante $\emptyset$

- per rappresentare $\emptyset$ si potrebbe usare un vettore di interi costanti; questo vettore non può però essere usato direttamente per effettuare inserimenti di elementi $\Longrightarrow$
- conviene definire una funzione che inizializza in modo opportuno un parametro passato per indirizzo

Es.:
```
void InitInsLettere(TipoInsLettere insieme)
{ TipoLettera ch;
  for (ch = 'A'; ch <= 'Z'; ch++)
    insieme[ch - 'A'] = FALSE;
}
```

3) Realizzazione delle operazioni

Es.:
```
void Inserisci(TipoInsLettere insieme, TipoLettera lettera)
{ insieme[lettera - 'A'] = TRUE; }
```

Esercizio: Fornire l'implementazione delle rimanenti operazioni.

Soluzione: file `tipidato/inslett.c`

Osservazioni:

- Per uno stesso tipo astratto si possono avere **più realizzazioni** diverse.

  Esempio: "Insieme di lettere alfabetiche maiuscole" può essere realizzato utilizzando il vettore caratteristico oppure una lista collegata di caratteri.

- Una realizzazione potrebbe avere delle **limitazioni** rispetto al tipo di dato astratto.

  Esempio: "Insieme di interi" potrebbe essere realizzato in modo tale da poter rappresentare come elementi solo gli interi rappresentabili in C.

Esercizio: Realizzare il tipo di dato astratto "Insieme di interi" (con domini, costanti e operazioni analoghe a $InsLettere$) usando liste concatenate.

Soluzione: file `tipidato/insint.c`

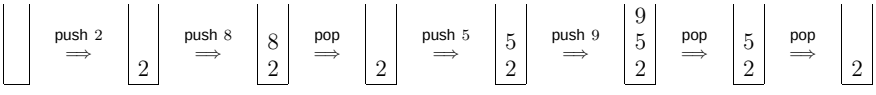
Il tipo di dato pila

Una **pila** è una sequenza di elementi (tutti dello stesso tipo) in cui l'inserimento e l'eliminazione di elementi avvengono secondo la regola:

> L'elemento che viene eliminato tra quelli presenti nella pila deve essere quello che è stato inserito per ultimo.

Si parla di gestione **LIFO** (per "**L**ast **I**n, **F**irst **O**ut").

Esempio:



- L'elemento in cima alla pila viene detto **elemento affiorante**.
- Una pila senza elementi viene detta **pila vuota**.

Utilizzo delle pile

Uno degli utilizzi delle pile si ha nella soluzione di problemi:

- che richiedono di effettuare più scelte in successione, e
- per cui può essere necessario ritrattare le scelte fatte per provare altre alternative
  $\Longrightarrow$ si deve tornare indietro sull'ultima scelta fatta (backtracking)

Si utilizza una pila per memorizzare la sequenza di scelte fatte:

- $Push$ di una nuova scelta;
- Pop ed eventuale $Push$ della scelta alternativa quando di deve fare backtracking.

Esempio: Ricerca di un cammino nella palude (file `ricorsio/palude.c`).
Nella soluzione ricorsiva viene utilizzata implicitamente la pila dei record di attivazione:

- quando si avanza, si effettua una chiamata ricorsiva ($Push$ di un nuovo RDA)
- quando non si può più proseguire, si esce dalla ricorsione per provare le scelte alternative (Pop del RDA ed eventuale $Push$ del RDA di una nuova chiamata ricorsiva)

Concettualizzazione del tipo di dato pila

- *domini:* $Pila$, $Elemento$, $Booleano$
- *costanti:* $PilaVuota$
- *operazioni:*
  - $TestPilaVuota : Pila \rightarrow Booleano$
    restituisce $vero$ se la pila è vuota, $falso$ altrimenti
  - $TopPila : Pila \rightarrow Elemento$
    restituisce l'elemento affiorante di una pila
  - $Push : Pila \times Elemento \rightarrow Pila$
    inserisce un elemento in cima ad una pila e restituisce la pila modificata
  - $Pop : Pila \rightarrow Pila \times Elemento$
    estrae l'elemento affiorante dalla cima della pila e lo restituisce (insieme alla pila modificata)

Realizzazione in C del tipo di dato pila

Indipendentemente da come verrà realizzato il tipo astratto, si possono specificare i prototipi delle funzioni che realizzano le diverse operazioni.

- *domini:*
```
typedef int bool;
typedef ... TipoElemPila;
typedef ... TipoPila;
```
- *costanti:*
```
void InitPila(TipoPila *pp);
```
- *operazioni:*
  - `bool TestPilaVuota(TipoPila p);`
  - `void TopPila(TipoPila p, TipoElemPila *pv);`
  - `void Push(TipoPila *pp, TipoElemPila v);`
  - `void Pop(TipoPila *pp, TipoElemPila *pv);`

Rappresentazione sequenziale delle pile

Una pila è rappresentata mediante:

- un **vettore di elementi** del tipo degli elementi della pila, riempito fino ad un certo indice con gli elementi della pila
- un **indice** che rappresenta la posizione nel vettore dell'elemento affiorante della pila

Esempio: Pila

8
34
12

rappresentata tramite:

`p.pila`

0	12
1	34
2	8
3	?
⋮	⋮
99	?

`p.pos`

2

Per la pila vuota: `p.pos` è pari a -1.

È necessario fissare la dimensione del vettore a tempo di compilazione. Si ha quindi un limite superiore al numero di elementi nella pila. $\Longrightarrow$

Si rende necessaria un'operazione di verifica di pila piena:
– `bool TestPilaPiena(TipoPila p);`

Implementazione: file `tipidato/pileseq.c`

Dichiarazioni di tipo:

```
#define MaxPila 100
typedef ... TipoElemPila;
struct tipoPila {
  TipoElemPila pila[MaxPila];
  int pos;
};
typedef struct tipoPila TipoPila;
```

Inizializzazione di una pila:

```
void InitPila(TipoPila *pp)
{
  (*pp).pos = -1;
}
```

Operazioni: sono tutte molto semplici e di **costo costante** (ovvero indipendente dal numero di elementi nella pila)

Esercizio: Implementare le operazioni sulle pile.

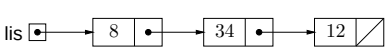
Rappresentazione collegata delle pile

Una pila è rappresentata mediante una **lista** nella quale il primo elemento è l'elemento affiorante della pila.

Esempio: Pila

8
34
12

rappresentata tramite:



Per rappresentare la pila vuota: lista vuota

Non è più necessario imporre un limite al numero massimo di elementi nella pila.
$\Longrightarrow$ `TestPilaVuota` non serve più.

Implementazione per **esercizio**: file `tipidato/pile.c`

Anche in questo caso le funzioni sono molto semplici e di costo costante:

- `Push`: tramite un inserimento in testa ad una lista
- `Pop`: tramite la cancellazione del primo elemento di una lista

Perchè non abbiamo usato una lista in cui l'elemento affiorante è l'ultimo della lista?

Rappresentazione sequenziale tramite vettori dinamici

Per utilizzare un vettore senza dover fissare la dimensione massima della pila a tempo di compilazione si può utilizzare un **vettore allocato dinamicamente**:

- si fissa una dimensione iniziale del vettore (che determina anche quella minima)
- la dimensione può crescere e decrescere a seconda delle necessità

Dichiarazioni di tipo:

```
#define DimInizialePila 10
typedef ... TipoElemPila;
struct tipoPila { TipoElemPila *pila;
                  int pos;
                  int dimCorrente; };
typedef struct tipoPila TipoPila;
```

Inizializzazione di una pila: richiede l'allocazione dinamica della memoria per il vettore

```
void InitPila(TipoPila *pp)
{ (*pp).pila = malloc(DimInizialePila * sizeof(TipoElemPila));
  (*pp).pos = -1;
  (*pp).dimCorrente = DimInizialePila;
}
```


Implementazione delle operazioni: file `tipidato/piledin.c`

- verifica di pila vuota e analisi dell'elemento affiorante:

  Come per la rappresentazione sequenziale statica.

- inserimento di un elemento:

  Se la pila è piena se ne raddoppia la dimensione.

  Si usa la funzione `realloc`, con prototipo:

  ```
  void *realloc(void *p, size_t dim);
  ```

  Alloca una nuova zona di memoria di dimensione specificata, copiandoci il contenuto di una zona di memoria precedentemente allocata dinamicamente.

- eliminazione dell'elemento affiorante:

  Se il numero di elementi nella pila è meno di un terzo della dimensione massima (e più della dimensione iniziale), allora si alloca una nuova zona di memoria di metà dimensione (usando `realloc`).

Confronto tra le rappresentazioni sequenziali e quella collegata

Rappresentazione sequenziale statica:

+ implementazione semplice
+ può essere usata anche con linguaggi privi di allocazione dinamica della memoria (ad ed. FORTRAN)
– è necessario fissare a priori il numero massimo di elementi nella pila
– l'occupazione di memoria è sempre pari alla dimensione massima

Rappresentazione sequenziale dinamica:

+ pila può crescere indefinitamente
– le operazioni di $Push$ e Pop possono richiedere la copia di tutti gli elementi presenti nella pila $\Longrightarrow$ non sono più a tempo di esecuzione costante

Rappresentazione collegata:

+ pila può crescere indefinitamente
+ occupazione di memoria è proporzionale al numero di elementi nella pila
– maggiore occupazione di memoria dovuta ai puntatori
– implementazione leggermente più complicata

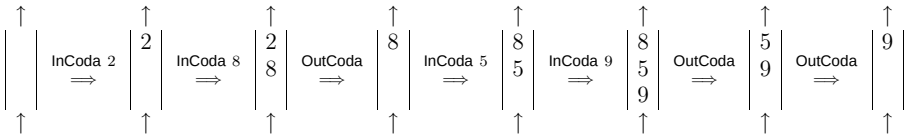
Il tipo di dato coda

Una **coda** è una sequenza di elementi (tutti dello stesso tipo) in cui l'inserimento e l'eliminazione di elementi avvengono secondo la regola:

> L'elemento che viene eliminato tra quelli presenti nella coda deve essere quello che è stato inserito per primo.

Si parla di gestione **FIFO** (per "**F**irst **I**n, **F**irst **O**ut").

Esempio:



Utilizzo delle code

Quando si vuole disaccoppiare il processamento di dati dalla loro generazione, ma si vuole mantenere l'ordinamento, ovvero processare i dato nello stesso ordine in cui sono stati generati.

Concettualizzazione del tipo di dato coda

- *domini:* $Coda$, $Elemento$, $Booleano$
- *costanti:* $CodaVuota$
- *operazioni:*
  - $TestCodaVuota : Coda \rightarrow Booleano$
    restituisce $vero$ se la coda è vuota, $falso$ altrimenti
  - $InizioCoda : Coda \rightarrow Elemento$
    restituisce il primo elemento di una coda
  - $InCoda : Coda \times Elemento \rightarrow Coda$
    inserisce un elemento in una coda e restituisce la coda modificata
  - $OutCoda : Coda \rightarrow Coda \times Elemento$
    estrae l'elemento dalla coda e lo restituisce (insieme alla coda modificata)

Realizzazione in C del tipo di dato coda

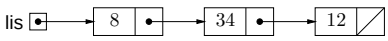
Indipendentemente da come verrà realizzato il tipo astratto, si possono specificare i prototipi delle funzioni che realizzano le diverse operazioni.

- *domini:*
```
typedef int bool;
typedef ... TipoElemCoda;
typedef ... TipoCoda;
```
- *costanti:*
```
void InitCoda(TipoCoda *pc);
```
- *operazioni:*
  - `bool TestCodaVuota(TipoCoda c);`
  - `void InizioCoda(TipoCoda c, TipoElemCoda *pv);`
  - `void InCoda(TipoCoda *pc, TipoElemCoda v);`
  - `void OutCoda(TipoCoda *pc, TipoElemCoda *pv);`

Rappresentazione collegata delle code

Come per le pile, si può rappresentare una coda tramite una **lista** nella quale il primo elemento è il primo elemento della coda (ovvero quello che deve uscire per primo).

Esempio: Coda $\uparrow 8\ 34\ 12 \uparrow$ rappresentata tramite:



Per rappresentare la coda vuota: lista vuota

Le operazioni $InitCoda$, $TestCodaVuota$, $InizioCoda$ e $OutCoda$ sono semplici da realizzare e di costo costante (ovvero, indipendente dalla lunghezza della coda).

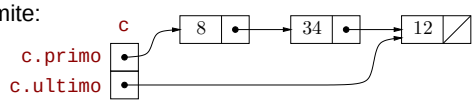
Per $InCoda$ bisogna fare un inserimento in coda ad una lista. $\Longrightarrow$
È necessaria una scansione di tutta la lista (costo non è più costante).

Si può modificare la rappresentazione in modo che tutte le operazioni vengano realizzate a costo costante?

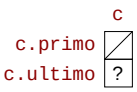
Si usa una rappresentazione mediante **lista e due puntatori**:

- un puntatore al primo elemento della coda
  (permette di realizzare $InizioCoda$ e $OutCoda$ a costo costante)
- un puntatore all'ultimo elemento della coda
  (permette di realizzare $InCoda$ a costo costante)

Esempio: Coda $\uparrow 8\ 34\ 12 \uparrow$ rappresentata tramite:



Per rappresentare la coda vuota:



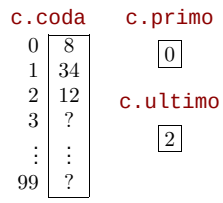
- vantaggio: tutte le operazioni hanno costo costante
- svantaggio: funzioni `InCoda` e `OutCoda` devono tenere conto dei due puntatori

Implementazione: file `tipidato/code.c`

Rappresentazione sequenziale delle code

Una coda è rappresentata mediante **un vettore** e **due indici**.

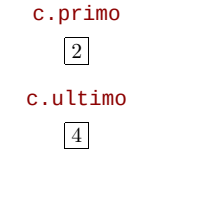
Esempio: Coda $\uparrow 8\ 34\ 12 \uparrow$ rappresentata tramite:



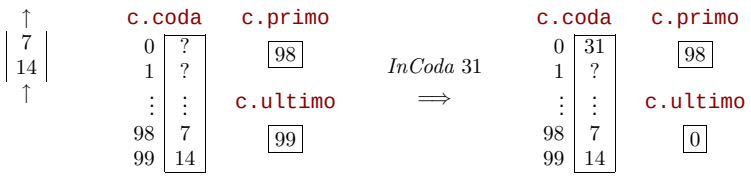
Per la coda vuota: `c.primo` è pari a -1.

Usando due indici si evita di spostare tutti gli elementi verso l'alto ad ogni $OutCoda$.

Esempio: $InCoda\ 25$, $InCoda\ 76$, $OutCoda$, $OutCoda$ $\uparrow 12\ 25\ 76 \uparrow$



Esempio: Dopo una serie di inserimenti ed estrazioni si potrebbe arrivare alla situazione:



$\Longrightarrow$ Il vettore deve essere **gestito in modo circolare**.

Verifica di coda piena: due possibilità

- `(c.primo - c.ultimo == 1)`
- `((c.ultimo - c.primo) == (MaxCoda-1))`

Implementazione delle operazioni: file `tipidato/codeseq.c`

Anche per le code si può usare una rappresentazione sequenziale tramite vettore dinamico.

Implementazione: per **esercizio**

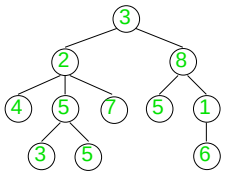
Gli alberi binari

Un **albero** è una struttura di data organizzata gerarchicamente.

È costituito da un insieme di **nodi** collegati tra di loro:

- ogni nodo contiene dell'informazione, detta **etichetta** del nodo
- i nodi collegati ad un certo nodo n vengono detti **figli** di n
- i nodi che non hanno figli sono detti **foglie** dell'albero
- c'è un unico nodo che non è figlio di alcun altro nodo, detto **radice** dell'albero (scendendo a partire dalla radice si raggiungono tutti i nodi dell'albero)

Esempio: Albero in cui le etichette sono degli interi.



la radice ha etichetta 3
i figli della radice hanno etichette 2 e 8
le foglie hanno etichette 4, 3, 5, 7, 5, 6

Albero vuoto: non ha nessun nodo (nemmeno la radice)

Cammino in un albero: sequenza di nodi, in cui ogni nodo è figlio del nodo che lo precede nella sequenza

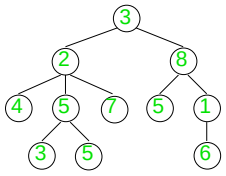
Esempio: (3, 8, 1, 6) è un cammino dalla radice ad una foglia
(8, 1) è un cammino

Livello (o **profondità**) di un nodo è la sua distanza dalla radice (quanto "in basso" si trova nell'albero).

Definizione formale (induttiva) di livello di un nodo:

- la radice ha livello 0
- se un nodo ha livello i, allora i suoi figli hanno livello $i + 1$

Esempio: Il nodo di etichetta 4 ha livello 2.
Il nodo di etichetta 6 ha livello 3.



Il **livello** i di un albero è formato da tutti i nodi a livello i.

Esempio: Il livello 2 è formato dai nodi con etichette 4, 5, 7, 5, 1.

Alberi binari

Un **albero binario** è un albero in cui ogni nodo ha **al massimo 2 figli**.

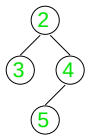
Esempio: Togliendo il nodo 7 dall'albero precedente otteniamo un albero binario.

- i due figli vengono detti **figlio sinistro** e **figlio destro**
- se consideriamo la parte di albero costituita dal figlio sinistro, dai suoi figli, dai figli dei suoi figli, . . . , questa è di nuovo un albero binario, detto **sottoalbero sinistro** (analogamente si definisce il **sottoalbero destro**)

Per rappresentare un albero binario in forma testuale si usa la **notazione parentetica**:

- albero vuoto: ( )
- albero non vuoto: (*radice* *sottoalbero-sinistro* *sottoalbero-destro*)
- se manca un figlio si mette l'albero vuoto come figlio

Esempio:




```
( 2  (3 () ())
     (4 (5 () ())
        ()
     )
)
```


Rappresentazione collegata degli alberi binari

L'elemento fondamentale è il nodo, che

- ha un'etichetta
- è collegato ai sottoalberi sinistro e destro (eventualmente vuoti)

$\Longrightarrow$ usiamo una struct a 3 campi:

- etichetta
- due puntatori ai sottoalberi sinistro e destro

```
typedef ... TipoInfoAlbero;                /* tipo dell'etichetta dei nodi */
struct nodoAlbero {
  TipoInfoAlbero info;
  struct nodoAlbero *destro, *sinistro;
}
typedef struct nodoAlbero NodoAlbero;
typedef NodoAlbero *TipoAlbero;
```


Stampa su schermo di un albero binario in rappresentazione parentetica

Implementazione: file `tipidato/albbin.c`, funzione `StampaAlbero`

Lettura da file di un albero binario in rappresentazione parentetica e creazione dell'albero in memoria

algoritmo lettura di un albero da file
  leggi '('
  leggi un carattere
  if il carattere è ')'
  then restituisci `NULL`
  else leggi la radice
    leggi il sottoalbero sinistro da file
    leggi il sottoalbero destro da file
    leggi ')'

Implementazione per **esercizio**: file `tipidato/albbin.c`, funzione `LeggiAlbero`

Visita di un albero

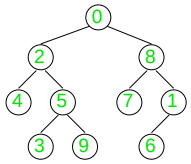
È l'analisi in sequenza di tutti i nodi dell'albero.

La maggior parte delle operazioni sugli alberi possono essere considerate delle varianti di visite.

Diversi **tipi di visita**, che differiscono per l'ordine in cui vengono visitati i nodi.

- visita in **preordine**: si analizza la radice, poi si visita in preordine il sottoalbero sinistro, poi il sottoalbero destro
- visita in **postordine**: si visita in postordine il sottoalbero sinistro, poi il sottoalbero destro, infine si analizza la radice
- visita **per livelli**: si visita prima la radice (livello 0), poi tutti i nodi a livello 1, poi tutti i nodi a livello 2, ...

Esempio:



preordine: 0 2 4 5 3 9 8 7 1 6
postordine: 4 3 9 5 2 7 6 1 8 0
per livelli: 0 2 8 4 5 7 1 3 9 6

Implementazione delle visite

Supponiamo di aver definito una funzione `Analizza`, a cui passiamo l'etichetta di un nodo, la quale effettua l'operazione richiesta dalla visita (assumiamo che sia semplicemente la stampa dell'etichetta).

- in preordine: banale, usando una funzione ricorsiva

  Implementazione: file `tipidato/albbin.c`, funzione `VisitaPreordine`

- in postordine: idem

  Implementazione: file `tipidato/albbin.c`, funzione `VisitaPostordine`

- per livelli:
  problema: dopo aver visitato un nodo, dobbiamo visitare i suoi fratelli, ma dobbiamo anche ricordarci i suoi figli (per visitarli dopo)
  $\Longrightarrow$ utilizziamo una coda, i cui elementi sono i nodi ancora da analizzare

  Implementazione: file `tipidato/albbin.c`, funzione `VisitaLivelli`

Esercizi:

- implementare le visite in preordine ed in postordine utilizzando una pila (invece che tramite una funzione ricorsiva)
- funzione che determina se l'albero contiene un nodo con una certa etichetta
- funzione che stampa tutte le foglie dell'albero
- per alberi con etichette di tipo `int`, funzione che stampa la somma delle etichette
  - di tutti i nodi dell'albero
  - di tutte le foglie dell'albero
  - di tutti i nodi ad un certo livello
  - dei nodi del sottoalbero la cui radice ha un'etichetta data
- funzione che costruisce la lista
  - dei nodi dell'albero visitati in preordine, postordine, per livelli
  - delle foglie dell'albero
  - dei nodi di un certo livello
- funzione che determina se due alberi sono uguali
- funzione che effettua la copia di un albero

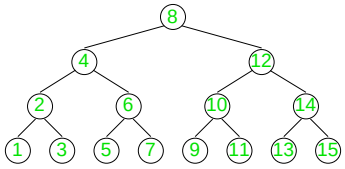
Alberi binari di ricerca

Consideriamo alberi binari per i quali è definito un **ordinamento** sull'informazione sui nodi (ad esempio interi, stringhe, ma anche strutture complesse con un campo chiave).

Un tale albero è un **albero binario di ricerca** se per **ogni** nodo N dell'albero:

- l'informazione associata ad **ogni** nodo nel sottoalbero sinistro di N è strettamente minore dell'informazione associata ad N
- l'informazione associata ad **ogni** nodo nel sottoalbero destro di N è strettamente maggiore dell'informazione associata ad N

Esempio:



L'albero di dice **bilanciato** se, detta P la profondità dell'albero, tutte le foglie stanno a livello P o $P-1$, e tutti i nodi interni hanno due figli, tranne al più un nodo al livello $P-1$ che ne ha uno solo.

La ricerca di un elemento *elem* in un albero binario di ricerca può essere fatta efficientemente (se l'albero è bilanciato):

algoritmo cerca *elem* nell'albero binario di ricerca *Alb*
  if *Alb* è vuoto
  then *elem* non è presente in *Alb*
  else if *elem* coincide con l'etichetta della radice di *Alb*
  then *elem* è stato trovato
  else if *elem* precede l'etichetta della radice di *Alb*
  then cerca *elem* nel sottoalbero sinistro di *Alb*
  else cerca *elem* nel sottoalbero destro di *Alb*

Implementazione: file `tipidato/albinric.c`

Costo: è **pari alla profondità** dell'albero. Sia n il numero di nodi dell'albero.

- caso migliore: l'albero è **bilanciato** $\Longrightarrow$ la profondità è proporzionale a $\log_2 n$
- caso peggiore: l'albero è una catena $\Longrightarrow$ la profondità è pari ad n