Algoritmi e Strutture di Dati

Alberi binari di ricerca

m.patrignani

Nota di copyright

- queste slides sono protette dalle leggi sul copyright
- il titolo ed il copyright relativi alle slides (inclusi, ma non limitatamente, immagini, foto, animazioni, video, audio, musica e testo) sono di proprietà degli autori indicati sulla prima pagina
- le slides possono essere riprodotte ed utilizzate liberamente, non a fini di lucro, da università e scuole pubbliche e da istituti pubblici di ricerca
- ogni altro uso o riproduzione è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte degli autori
- gli autori non si assumono nessuna responsabilità per il contenuto delle slides, che sono comunque soggette a cambiamento
- questa nota di copyright non deve essere mai rimossa e deve essere riportata anche in casi di uso parziale

Contenuto

- Tipo astratto di dato array associativo
- Alberi binari di ricerca (abr)
  - loro uso per la realizzazione di tipi astratti di dato
  - consultazione di un abr
    - ricerca di un valore
    - calcolo del valore massimo o minimo
  - creazione e manutenzione di un abr
    - inserimento e cancellazione di nodi

Array associativi

- Sappiamo già cos'è un array
  - una sequenza di variabili omogenee accedute tramite un indice intero in un intervallo specificato
    - esempio: A è un array di 10 caratteri da A[0] ad A[9]
- Un *array associativo* (o *mappa*, o *dizionario*) è un insieme di variabili omogenee accedute tramite chiavi (omogenee ma di tipo qualsiasi)
  - la chiave può essere un intero, una stringa, un oggetto, ecc
  - si assume che la chiave sia unica
- Un array associativo è dunque costituito da un insieme di coppie ⟨chiave, valore⟩

Esempi di array associativi

- Dizionario della lingua italiana
  - la chiave è un lemma
    - il valore è un testo che descrive la sua classificazione, i suoi significati, alcuni esempi d'uso, ecc.
- Dati fiscali dei contribuenti
  - la chiave è il codice fiscale
    - il valore è composto dai dati anagrafici, contributivi, ecc
- Dati satellite associati ad oggetti software
  - la chiave è un oggetto
    - il valore è un insieme di dati specifici associati all'oggetto
- Voti degli studenti del corso di ASD
  - la chiave è un numero di matricola
    - il valore è un voto in trentesimi

Il tipo astratto di dato array associativo

- Domini
  - il dominio di interesse è l'insieme A degli array associativi
  - dominio di supporto: le chiavi K dell'array associativo
  - dominio di supporto: i valori V dell'array associativo
    - comprensivo della costante "valore nullo"
  - dominio di supporto: i booleani B = {true, false}
- Costanti
  - l'array associativo vuoto
- Operazioni
  - aggiunge una coppia ⟨chiave,valore⟩ : PUT: $A \times K \times V \rightarrow A$
  - restituisce il valore associato ad una chiave: GET: $A \times K \rightarrow V$
    - può restituire il valore nullo se nessun elemento è associato alla chiave
  - rimuove la coppia ⟨chiave,valore⟩: DELETE: $A \times K \rightarrow A$
  - verifica che un a chiave sia utilizzata: EXISTS: $A \times K \rightarrow B$
  - …

Realizzazioni di array associativi

- Un array associativo potrebbe essere realizzato tramite…
  - un array di coppie chiave-valore non ordinato
  - un array di coppie chiave-valore ordinato in base alle chiavi
  - una lista di coppie chiave-valore non ordinata
  - una lista di coppie chiave-valore ordinata in base alle chiavi
- Nessuna di queste realizzazioni garantisce tempi logaritmici sia per l'inserimento che per la ricerca

Realizzazione tramite array ordinato

- In un array di coppie chiave-valore ordinato in base alle chiavi
  - l'inserimento ha un costo lineare in quanto occorre traslare tutti gli elementi dell'array successivi alla posizione corrente
  - la ricerca ha un costo logaritmico tramite una strategia di ricerca binaria
- Strategia di ricerca binaria:
  - salto a metà dell'intervallo
  - se ho trovato il valore cercato la procedura termina
  - se trovo un valore maggiore ricorro nel primo sottointervallo
  - altrimenti ricorro nel secondo sottointervallo

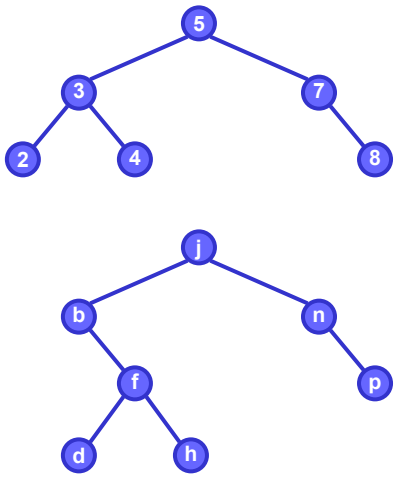
Alberi binari di ricerca

- Detti anche abr o bst (binary search trees)
- Sono strutture di dati utilizzate per implementare dizionari, cioè coppie ⟨chiave, valore⟩ nel caso in cui sulle chiavi, che si suppongono uniche, sia definita una relazione d'ordine totale (o "lineare")
  - sono disponibili due funzioni `MINORE`(x,y) e `UGUALE`(x,y) che calcolano la posizione reciproca di due chiavi
    - per esempio: le chiavi sono interi e la relazione d'ordine è "<"

Alberi binari di ricerca

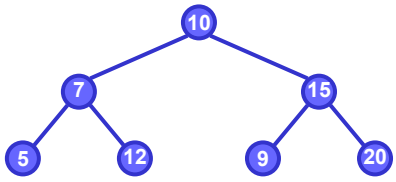
- Gli abr sono realizzati con alberi binari radicati
  - ogni nodo dell'albero rappresenta un elemento e ha i campi
    - `parent`: riferimento al nodo genitore
    - `left`: riferimento al figlio sinistro
    - `right`: riferimento al figlio destro
    - `key`: valore della chiave
      - nel seguito supporremo che sia un intero
    - `value`: il valore associato alla chiave
      - nel seguito ignoreremo questo valore
- Per ogni nodo x dell'albero
  - tutti i nodi del sottoalbero sinistro di x hanno chiave minore di quella di x
  - tutti i nodi del sottoalbero destro di x hanno chiave maggiore di quella di x

Esempi di abr



Esercizi sugli alberi binari di ricerca

- Disegna abr di altezza 2, 3, 4, 5, 6 sull'insieme di chiavi {1, 4, 5, 10, 16, 17, 12}
- Questo albero è un abr?



Operazioni sugli abr

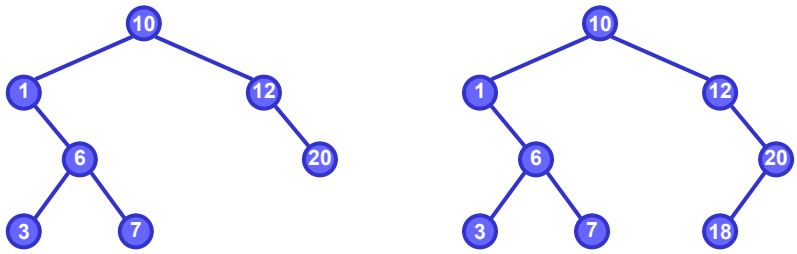
- Gli abr supportano operazioni di
  - modifica
    - inserimento di un elemento nell'inseme
    - cancellazione di un elemento dall'inseme
  - consultazione
    - calcolo del minimo valore contenuto
    - calcolo del massimo valore contenuto
    - ricerca del nodo (se esiste) contentente un particolare valore
  - verifica di consistenza
    - verifica che un albero sia effettivamente un abr

Inserimento di un nuovo elemento nell'abr

- Se l'abr è vuoto
  - crea un nodo che diventa radice dell'abr
- Altrimenti
  - se l'oggetto da inserire è uguale al nodo corrente
    - rifiuta l'inserimento
  - altrimenti
    - se l'oggetto da inserire è < del nodo corrente
      - inserisce nel sottoalbero sinistro
    - altrimenti
      - inserisce nel sottoalbero destro
- Il nuovo nodo sarà sempre inserito come foglia
  - quindi con i due sottoalberi vuoti

Inserimento di un nodo in un abr

- Inserimento del nodo “18”



Inserimento di un nodo in un abr

- Suppongo di avere a disposizione una funzione per la creazione di nuovi nodi

```
NEW_TREE_ELEM(k)     /* costruisco un nodo con chiave k */
1.  /* x è un oggetto con campi p, left, right, e key */
2.  x.p = x.left = x.right = NULL     /* genitore e figli */
3.  x.key = k
4.  return x
```

- Il caso in cui l’abr è vuoto viene trattato separatamente

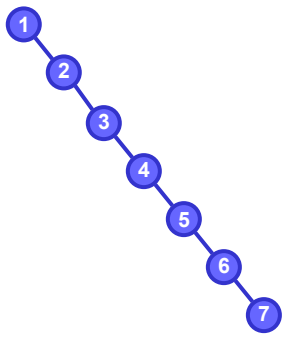
```
TREE_INSERT(t,k)
1.  if (t.root == NULL)
2.      t.root = NEW_TREE_ELEM(k)
3.      return TRUE        /* nodo inserito correttamente */
4.  else return BST_INSERT(t.root, k)
```

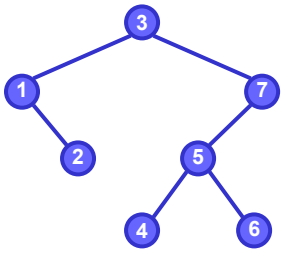


```
BST_INSERT(x,k)    /* ins. k nell'abr radicato a x != NULL */
1.  output = FALSE   /* nodo non inserito */
2.  if (MINORE(k,x.key))    /* devo inserire k a sinistra */
3.      if (x.left == NULL)
4.          x.left = NEW_TREE_ELEM(k)
5.          x.left.p = x
6.          output = TRUE    /* nodo inserito */
6.      else
7.          output = BST_INSERT(x.left,k)
8.  else if (MINORE(x.key,k)) /* devo inserire k a destra */
9.      if (x.right == NULL)
10.         x.right = NEW_TREE_ELEM(k)
11.         x.right.p = x
12.         output = TRUE    /* nodo inserito */
12.     else
13.         output = BST_INSERT(x.right,k)
14. return output /* è FALSE se il nodo era già presente */
```

La forma dell'albero dipende dall'ordine di inserimento

Inserimento di 1, 2, 3, 4, 5, 6, 7

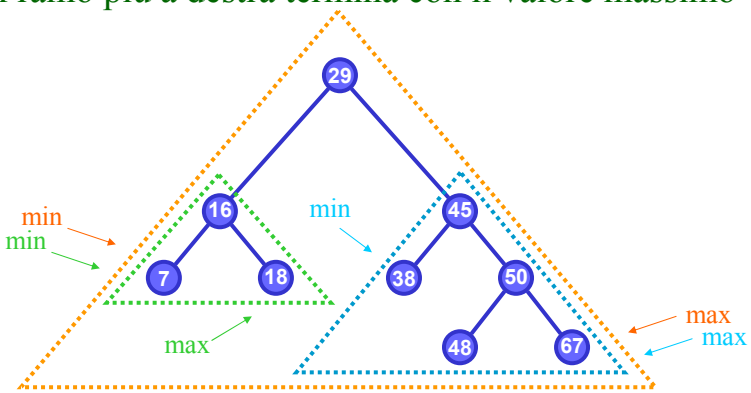




Inserimento di 3, 1, 7, 2, 5, 4, 6

Calcolo del minimo e del massimo

- In un abr
  - il ramo più a sinistra termina con il valore minimo
  - il ramo più a destra termina con il valore massimo



TREE_MINIMUM e TREE_MAXIMUM

- Queste funzioni ritornano il riferimento al nodo che contiene il minimo e il massimo valore dell'abr

```
TREE_MINIMUM(x)        /* minimo del sottoalbero radicato ad x */
1.  while (x.left != NULL)
2.      x = x.left      /* scendo a sinistra finché posso */
3.  return x
```

```
TREE_MAXIMUM(x)        /* massimo del sottoalbero radicato ad x */
1.  while (x.right != NULL)
2.      x = x.right     /* scendo a destra finché posso */
3.  return x
```

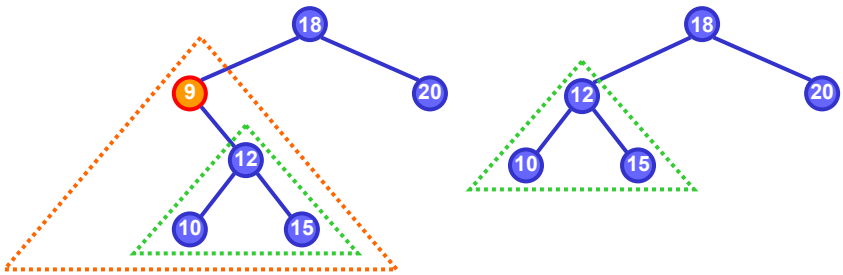
- Osservazione banale (che sarà utile in seguito)
  - il nodo minimo non ha figlio sinistro
  - il nodo massimo non ha figlio destro

Strategia per la cancellazione di un nodo *x*

- Se *x* non ha figli lo posso sempre rimuovere
- Se il nodo *x* ha un figlio
  - elimino *x* collegando il genitore con il figlio
- Se il nodo *x* ha due figli
  - cerco il nodo *y*, successore di *x*, nel sottoalbero destro
    - *y* ha al massimo un figlio
  - rimuovo *y*
  - sostituisco *y* ad *x*
- Sarebbe stato analogo cercare il nodo predecessore di x nel sottoalbero sinistro

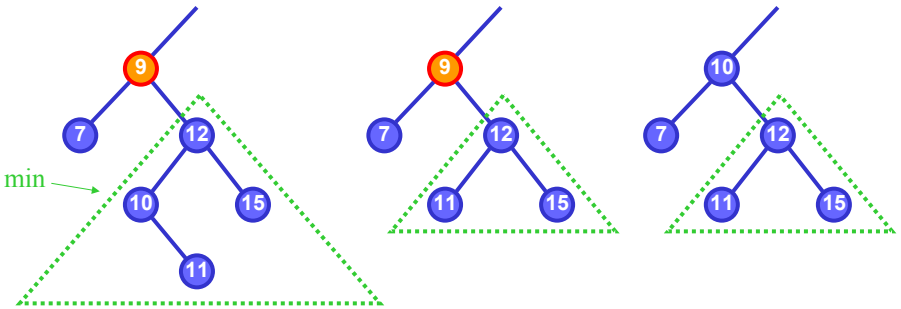
Rimozione di un nodo con un solo figlio

- devo rimuovere il nodo 9 (che ha un solo figlio)
- Il sottoalbero radicato al nodo 12:
  - è contenuto nel sottoalbero radicato al nodo 9
  - contiene tutti valori minori o uguali a 18
  - dunque può essere il sottoalbero sinistro del nodo 18



Rimozione di un nodo con due figli

- Devo rimuovere il nodo 9 che ha due figli
- Il nodo "successore" del nodo 9 è il nodo 10 (minimo del sottoalbero destro del nodo 9)
- Il nodo 10 è rimuovibile con la strategia precedente
- Sostituendo il nodo 10 al nodo 9 si ottiene un abr



Funzione TREE_BYPASS

```
TREE_BYPASS(t,x)     /* x ha al massimo un figlio */
1.  if (x.left != NULL)
2.      figlio = x.left
3.  else
4.      figlio = x.right     /* NULL se x non ha figli */
5.  if (figlio != NULL)
6.      figlio.p = x.p
7.  if (x.p != NULL)  /* c'è il parent di x da aggiornare */
8.      if (x == x.p.left)  /* x era il figlio sinistro */
9.          x.p.left = figlio
10.     else                /* x era il figlio destro */
11.         x.p.right = figlio
12. else t.root = figlio
```

- La complessità è $\Theta(1)$

Funzione TREE-DELETE

```
TREE_DELETE(t,x)     /* x qualsiasi */
1.  if (x.left != NULL) and (x.right != NULL)
2.      y = TREE_MINIMUM(x.right)
3.      x.key = y.key
4.  else
5.      y = x
6.  TREE_BYPASS(t,y)
```

- TREE_DELETE ha complessità $\Theta(h)$
  - la funzione TREE_MINIMUM ha complessità $\Theta(h)$

Ricerca del nodo con chiave *k*

```
ITERATIVE_TREE_SEARCH(x,k) /* suppongo x radice dell'abr */
1.  while (x != NULL) and k != x.key
2.      if (k < x.key)
3.          x = x.left
4.      else
5.          x = x.right
6.  return x      /* ritorna il riferim. al nodo */
```

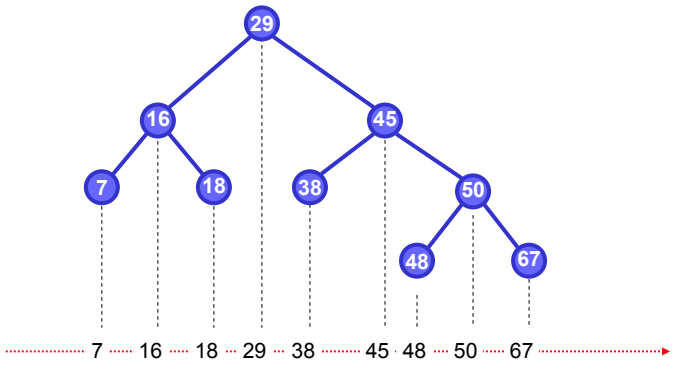
```
RECURSIVE_TREE_SEARCH(x,k) /* suppongo x radice dell'abr */
1.  if (x == NULL) or (k == x.key)
2.      return x  /* ritorna il riferim. al nodo */
3.  if (k < x.key)
4.      return RECURSIVE_TREE_SEARCH(x.left,k)
5.  else
6.      return RECURSIVE_TREE_SEARCH(x.right,k)
```


Complessità delle operazioni sugli abr

- Gli algoritmi visti per l'inserimento, la cancellazione, la ricerca e per il calcolo del minimo e del massimo hanno tutti una complessità asintotica $\Theta(h)$, dove h è la profondità dell'abr
  - nel caso peggiore (albero sbilanciato) $h \in \Theta(n)$
  - nel caso migliore (albero bilanciato) $h \in \Theta(\log n)$
- Sono note delle strategie (alberi rosso-neri) per mantenere bilanciati gli abr

Verifica che un albero sia un abr

- Osservazione
  - se l'albero è un abr una visita simmetrica produce valori in ordine crescente



Visita simmetrica ABR_SYM

- La strategia è la seguente
  - compongo un array con tutti gli elementi dell'abr nell'ordine in cui sono processati da una visita simmetrica
  - verifico che l'array sia non decrescente

```
ABR_SYM(x)
1.  n = CONTA_NODI(x)
2.  /* creo l'array A con n posizioni */
3.  TREE_TO_ARRAY(A,x,0)    /* "riverso" l'albero in A */
4.  return IS_SORTED(A)
```

- La complessità asintotica è $\Theta(n)$ in quanto ogni singola fase ha costo $\Theta(n)$

Funzioni CONTA_NODI e IS_SORTED

```
CONTA_NODI(x)
1.  if (x == NULL) return 0
2.  l = CONTA-NODI(x.left)
3.  r = CONTA-NODI(x.right)
4.  return 1 + l + r
```

```
IS_SORTED(A)
1.  for i = 0 to A.length-2
2.      if A[i] > A[i+1]
3.          return FALSE
4.  return TRUE
```

- La complessità asintotica è $\Theta(n)$ per entrambe

Visita simmetrica TREE_TO_ARRAY

```
TREE_TO_ARRAY(A,x,i) /* uso A a partire dalla posizione i */
1. if (x == NULL) return i
2. i = TREE_TO_ARRAY(A,x.left,i)
3. A[i] = x.key
4. i = TREE_TO_ARRAY(A,x.right,i+1)
5. return i // i è la prossima posizione libera dell'array
```

- La complessità asintotica è $\Theta(n)$
- Questa funzione può essere usata per creare un algoritmo di ordinamento chiamato TREE_SORT
  - costruisco un abr da un array di input
  - lancio TREE_TO_ARRAY sull'abr ottenuto
  - l'array ottenuto è ordinato

Domande

- In un abr bilanciato con *n* nodi quanto costa:
  - l'inserimento di un nodo?
  - la cancellazione di un nodo?
  - la ricerca di un nodo?
- Quanto costano le stesse operazioni se l'abr è fortemente sbilanciato?

Esercizi

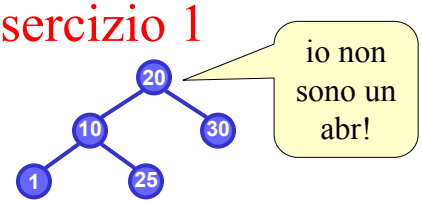
1. Scrivi lo pseudocodice della procedura `IS_ABR_PRE`(t) che verifica se un albero binario t di interi sia un albero binario di ricerca con una visita in preordine
   - qual è la sua complessità nel caso peggiore?
2. Scrivi lo pseudocodice della procedura `IS_ABR_POST`(t) che verifica se un albero binario t di interi sia un albero binario di ricerca con una visita in postordine
   - qual è la sua complessità nel caso peggiore?

Esercizi

3. Scrivi lo pseudocodice della procedura `TREE_SORT`(A) che ordina un array A di interi utilizzando un albero binario di ricerca
   - supponi di avere a disposizione le funzioni
     - `INSERISCI`(t,k) che in tempo lineare nell'altezza dell'albero t inserisce un intero k nell'albero
     - `TREE_TO_ARRAY`(A,t,i) che con una visita simmetrica riversa in tempo lineare l'albero t nell'array A a partire dalla posizione i
   - qual è la complessità di `TREE_SORT`(A) nel caso peggiore?
   - qual è la complessità di `TREE_SORT`(A) nel caso migliore?

Soluzione esercizio 1

- Si noti che il codice seguente è errato




```
IS_ABR_PRE(t)          /* t è un albero */
1.  return IS_ABR_ERRATO(t.root)
```

```
IS_ABR_ERRATO(x)       /* x è un nodo dell'albero */
1.  if x == NULL
2.          return TRUE
3.  else    return (   (x.key >= x.left.key ) and
4.                     (x.key <= x.right.key ) and
5.                     IS_ABR_ERRATO(x.left) and
6.                     IS_ABR_ERRATO(x.right) )
```


Soluzione esercizio 1

- Soluzione con una visita in preordine

```
IS_ABR_PRE(t)        /* t è un albero */
1.  return ABR_PRE_RIC(t.root)
```

```
ABR_PRE_RIC(x)                /* x è un nodo dell'albero */
1.  if x == NULL
2.          return TRUE
3.  else    return (   NO_MAGGIORE(x.left,x.key) and
4.                     NO_MINORE(x.right,x.key) and
5.                     ABR_PRE_RIC(x.left) and
6.                     ABR_PRE_RIC(x.right) )
```


Soluzione esercizio 1

- Dove NO_MAGGIORE e NO_MINORE sono

```
NO_MAGGIORE(x,v)      /* x è un nodo, v è un intero */
1.  if x == NULL
2.          return TRUE
3.  else    return (   (x.key <= v) and
4.                     NO_MAGGIORE(x.left,v) and
5.                     NO_MAGGIORE(x.right,v) )
```

```
NO_MINORE(x,v)        /* x è un nodo, v è un intero */
1.  if x == NULL
2.          return TRUE
3.  else    return (   (x.key >= v) and
4.                     NO_MINORE(x.left,v) and
5.                     NO_MINORE(x.right,v) )
```


Soluzione esercizio 1

- Nel caso peggiore l'abr è un albero completamente sbilanciato

$$T(n) = T(n\text{-}1) + \Theta(n)$$

- Questa equazione di ricorrenza ha la forma

$$T(n) = T(n\text{-}1) + g(n)$$

- Che ammette soluzione

$$T(n) = c + \sum_{k=1}^{n} g(k)$$

- Che nel caso in esame produce

$$T(n) = \Theta(n^2)$$

Soluzione esercizio 2

- Eseguo le operazioni in postordine (prima i figli)
- Quando considero un nodo *x* ho già verificato che i sottoalberi destro e sinistro siano abr
- Avendo già percorso i sottoalberi posso essermi contestualmente calcolato il valore minimo e massimo in essi contenuto
- La funzione `ABR_POST` ritorna un oggetto con tre valori
  - `is_abr`: booleano che mi dice se il sottoalbero è un abr
  - `min`: il minimo valore contenuto nell'abr
  - `max`: il massimo valore contenuto nell'abr

Soluzione esercizio 2

```
ABR_POST_RIC(x) /* ritorna un oggetto (is_abr, min, max) */
1.  if (x == NULL) return NULL
2.  l = ABR_POST_RIC(x.left)  // l ha campi is_abr, min, max
3.  r = ABR_POST_RIC(x.right) // idem
4.  if (l == NULL and r == NULL)  /* x è una foglia */
5.      return (TRUE, x.key, x.key)
6.  if (l == NULL and r != NULL) // x ha il figlio destro
7.      out = r.is_abr and (x.key < r.min)
8.      return (out, x.key, r.max)
9.  if (l != NULL and r == NULL) // x ha il figlio sinistro
10.     out = l.is_abr and (x.key > l.max)
11.     return (out, l.min, x.key)
12. out = l.is_abr and r.is_abr   // x ha entrambi i figli
13. out = out and (x.key < r.min) and (x.key > l.max)
14. return (out, l.min, r.max)
```

Soluzione esercizio 2

- Nel caso della visita in postordine tutti i test (linea 1 e dalla linea 4 alla linea 14) non prevedono chiamate a funzioni
  - la loro complessità è $\Theta(1)$
- Le chiamate ricorsive ad `ABR_POST_RIC` realizzano una visita in postordine
  - la complessità asintotica è $\Theta(n)$ come per tutte le visite in postordine

Soluzione esercizio 3

```
TREE_SORT(A)        ▷ A è un array che deve essere ordinato
1.  t.root = NULL           ▷ t è un nuovo albero
2.  for i = 1 to A.length-1
3.     INSERISCI(t,A[i])    ▷ r ha i campi is_abr, min, max
4.  TREE_TO_ARRAY(A,t,0)
```

- Complessità della procedura `TREE_SORT`
  - nel caso peggiore, poiché l'inserimento ha complessità lineare, la complessità totale è $\Theta(n^2)$
  - se l'albero fosse bilanciato l'inserimento avverrebbe in tempo $\Theta(\log n)$ e la complessità totale sarebbe $\Theta(n \log n)$ nel caso peggiore

