algoritmi e strutture di dati

alberi binari di ricerca

r. de virgilio   m.patrignani

contenuto

- tipi astratti 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 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: $A \times K \rightarrow B$
  - …

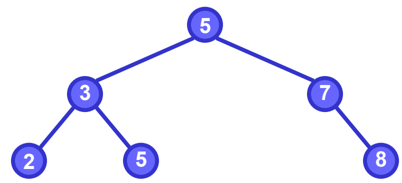
alberi binari di ricerca

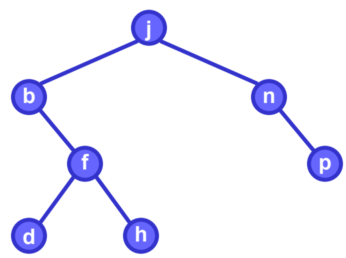
- detti anche abr o bst (binary search trees)
- sono strutture di dati utilizzate per implementare dizionari, cioè coppie ⟨chiave, valore⟩ nel caso in cui sulle chiavi, che si suppongono uniche, sia definita una relazione d'ordine totale (o "lineare")
  - sono disponibili due funzioni `MINORE`(x,y) e `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

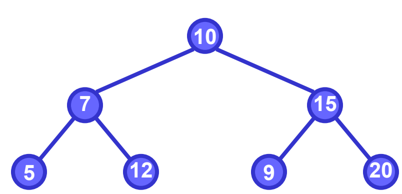
esempi di abr





esercizio 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

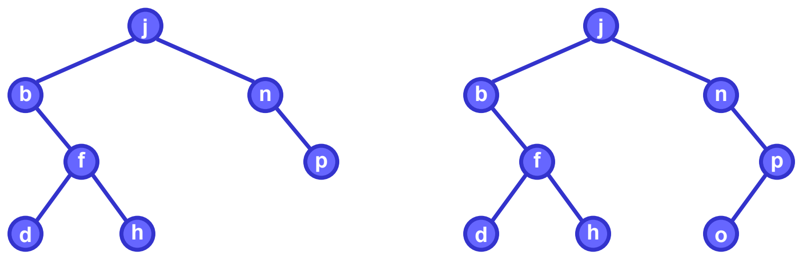
- gli abr supportano operazioni di
  - modifica
    - inserimento di un elemento nell’inseme
    - cancellazione di un elemento dall’inseme
  - consultazione
    - calcolo del minimo valore contenuto
    - calcolo del massimo valore contenuto
    - ricerca del nodo (se esiste) contentente un particolare 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 “o”



inserimento di un nodo in un abr

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

```
MK-TREE-ELEM(k)       ▷ costruisco un nodo con chiave k
   ▷ x è un oggetto con campi p, left, right, e key
   x.p = x.left = x.right = NULL      ▷ genitore e figli
   x.key = k
   return x
```

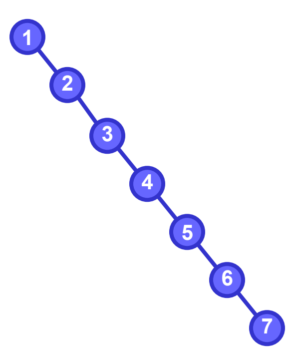
- il caso in cui l’abr è vuoto viene trattato separatamente

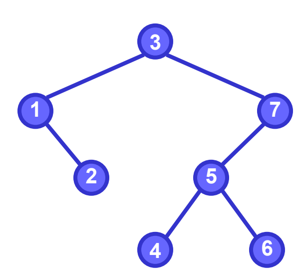
```
INSERISCI(t,k)
   if (t.root == NULL)
       t.root = MK-TREE-ELEM(k)
       return TRUE              ▷ nodo inserito correttamente
   else return BST-INSERT(t.root, k)
```

```
BST-INSERT(x,k)          ▷ ins. k nell'abr radicato a x != NULL
    output = FALSE       ▷ nodo non inserito
    if (MINORE(k,x.key))          ▷ devo inserire k a sinistra
        if (x.left == NULL)
            x.left = MK-TREE-ELEM(k)
            x.left.p = x
            output = TRUE     ▷ nodo inserito
        else
            output = BST-INSERT(x.left,k)
    else if (MINORE(x.key,k))  ▷ devo inserire k a destra
        if (x.right == NULL)
            x.right = MK-TREE-ELEM(k)
            x.right.p = x
            output = TRUE     ▷ nodo inserito
        else
            output = BST-INSERT(x.right,k)
    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

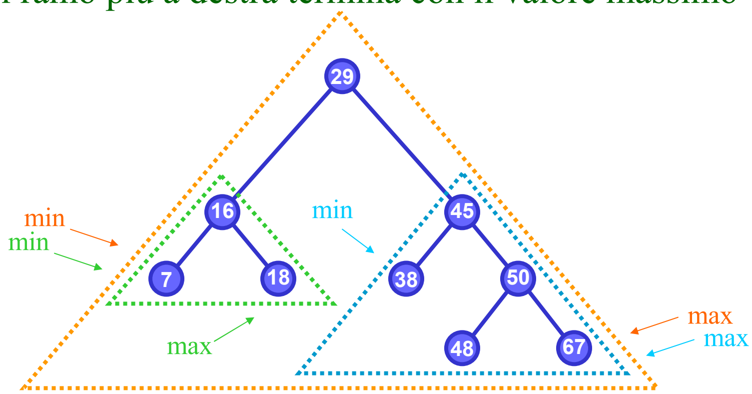




inserimento di 3, 1, 7, 2, 5, 4, 6

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)           ▷ suppongo x radice dell'abr
  while (x.left != NULL)
      x = x.left          ▷ scendo a sinistra finché posso
  return x
```

```
TREE-MAXIMUM(x)           ▷ suppongo x radice dell'abr
  while (x.right != NULL)
      x = x.right             ▷ scendo a destra finché posso
  return x
```

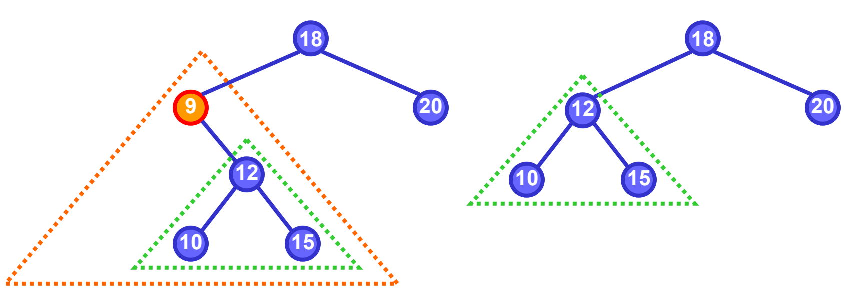
- osservazione (che sarà utile in seguito)
  - il nodo minimo non ha figlio sinistro
  - il nodo massimo non ha figlio destro

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

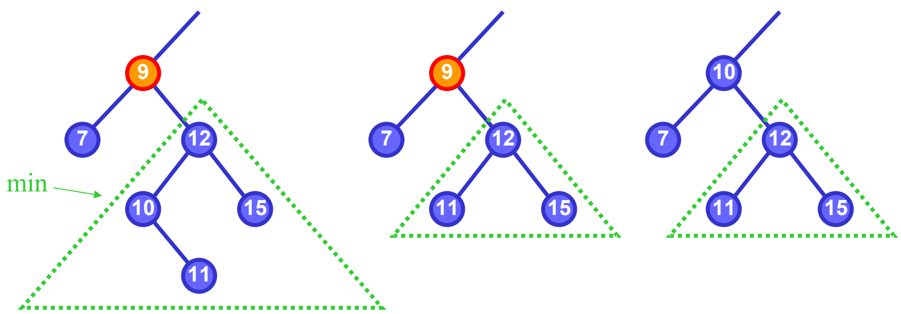
rimozione di un nodo con un solo figlio

- 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
if (x.left != NULL)
    figlio = x.left
else
    figlio = x.right      ▷ NULL se x non ha figli
if (figlio != NULL)
    figlio.p = x.p
if (x.p != NULL)      ▷ c'è il parent di x da aggiornare
    if (x == x.p.left)   ▷ x era il figlio sinistro
        x.p.left = figlio
    else                 ▷ x era il figlio destro
        x.p.right = figlio
▷ dealloca x se il linguaggio lo prevede
```

- la complessità è $\Theta(1)$

funzione TREE-DELETE

```
TREE-DELETE(t,x)      ▷ x qualsiasi
if (x.left != NULL) and (x.right != NULL)
    y = TREE-MINIMUM(x.right)
    x.key = y.key
else
    y = x
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
while (x != NULL) and k != x.key
    if (k < x.key)
        x = x.left
    else
        x = x.right
return x       ▷ ritorna il riferim. al nodo
```

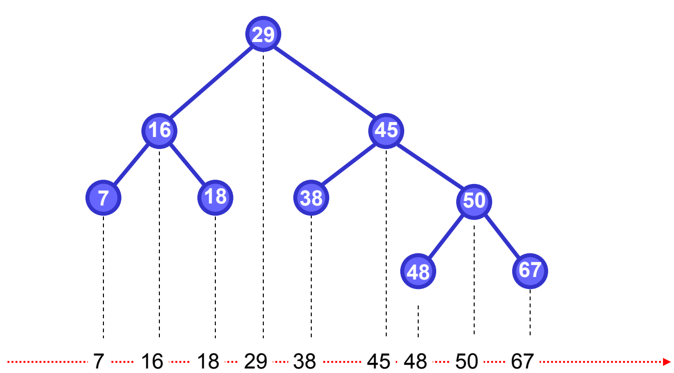
```
RECURSIVE-TREE-SEARCH(x,k)  ▷ suppongo x radice dell'abr
if (x == NULL) or (k == x.key)
    return x   ▷ ritorna il riferim. al nodo
if (k < x.key)
    return RECURSIVE-TREE-SEARCH(x.left,k)
else
    return RECURSIVE-TREE-SEARCH(x.right,k)
```

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
if (x == NULL) return i
i = TREE-TO-ARRAY(A,x.left,i)
A[i] = x.key
i = TREE-TO-ARRAY(A,x.right,i+1)
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

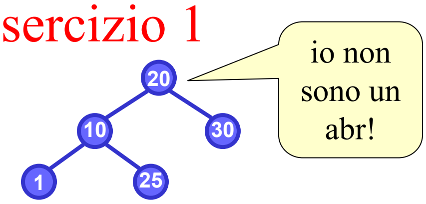
1. scrivi lo pseudocodice della procedura `IS-ABR-PRE`(*t*) che verifica se un albero binario *t* di interi sia un albero binario di ricerca con una visita in preordine
   - qual è la sua complessità nel caso peggiore?
2. scrivi lo pseudocodice della procedura `IS-ABR-POST`(*t*) che verifica se un albero 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 inserisce un intero *k* nell'albero *t*
     - `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?
   - quale sarebbe la complessità se ogni inserimento avvenisse su un albero bilanciato?

soluzione esercizio 1

- si noti che il codice seguente è errato




```
IS-ABR-PRE(t)           ▷ t è un albero
   return IS-ABR-ERRATO(t.root)
```

```
IS-ABR-ERRATO(x)        ▷ x è un nodo dell'albero
   if x == NULL
          return TRUE
   else   return (  (x.key >= x.left.key ) and
                    (x.key <= x.right.key ) and
                    IS-ABR(x.left) and
                    IS-ABR(x.right) )
```


soluzione esercizio 1

- soluzione con una visita in preordine

```
IS-ABR-PRE(t) ▷ t è un albero
   return ABR-PRE-RIC(t.root)
```

```
ABR-PRE-RIC(x)                 ▷ x è un nodo dell'albero
   if x == NULL
          return TRUE
   else   return (  NO-MAGGIORE(x.left,x.key) and
                    NO-MINORE(x.right,x.key) and
                    ABR-PRE-RIC(x.left) and
                    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
if x == NULL
        return TRUE
else    return (  (x.key <= v) and
                  NO-MAGGIORE(x.left,v) and
                  NO-MAGGIORE(x.right,v) )
```

```
NO-MINORE(x,v)           ▷ x è un nodo, v è un intero
if x == NULL
        return TRUE
else    return (  (x.key >= v) and
                  NO-MINORE(x.left,v) and
                  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)
if (x == NULL) return NULL
l = ABR-POST-RIC(x.left) ▷ l ha i campi is_abr, min, max
r = ABR-POST-RIC(x.right)▷ r ha i campi is_abr, min, max
if (l == NULL and r == NULL)   ▷ x è una foglia
    return (TRUE, x.key, x.key)
if (l == NULL and r != NULL)   ▷ x ha il figlio destro
    out = r.is_abr and (x.key <= r.min)
    return (out, x.key, r.max)
if (l != NULL and r == NULL)   ▷ x ha il figlio sinistro
    out = l.is_abr and (x.key >= l.max)
    return (out, l.min, x.key)
out = l.is_abr and r.is_abr    ▷ x ha entrambi i figli
out = out and (x.key <= r.min) and (x.key >= l.max)
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 Θ(1)
- le chiamate ricorsive ad `ABR-POST-RIC` realizzano una visita in postordine
  - la complessità asintotica è Θ(*n*) come per tutte le visite in postordine

soluzione esercizio 3

```
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 `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 Θ(log n) e la complessità totale sarebbe Θ(n log n) nel caso peggiore