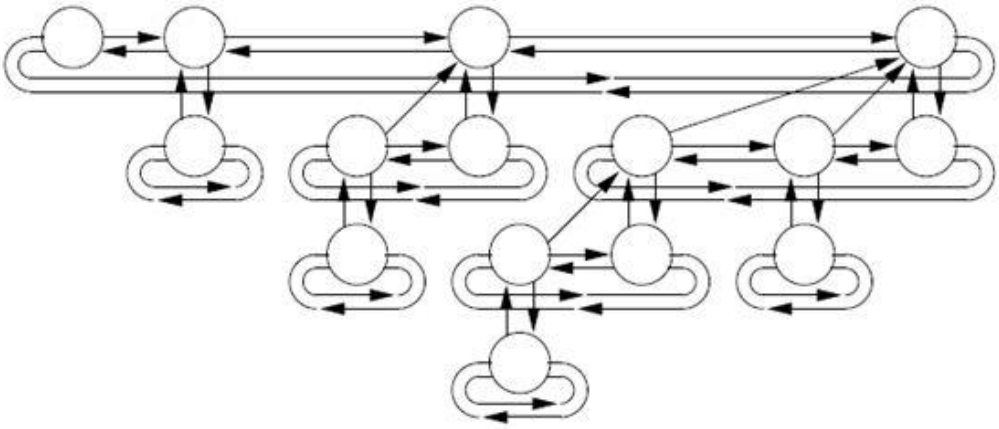


Heap di Fibonacci

Samuele Maria Gallina | Laboratorio di Algoritmi CDL informatica UniCT

Contenuti

Introduzione

Gli heap di Fibonacci sono interessanti strutture dati, dato che molte operazioni vengono eseguite in tempo ammortizzato costante.

Essi supportano le operazioni di quelli che chiamiamo heap riunibili, e sono particolarmente interessanti i suoi tempi di esecuzione in relazione agli Heap Binari.

In realtà essi non sono molto utilizzati per la complessità nella sua implementazione e per i fattori costanti a volte poco idonei per una buona esecuzione.

Se si trovasse una struttura basata sulla stessa idea però più "semplice" sarebbe una rivoluzione, al momento però sono preferiti gli Heap binari.

Heap Riunibile

Un Heap Riunibile è una struttura dati che sommariamente può eseguire queste operazioni(supponendo x sia un elemento):

- Insert(x);
- Minimum();
- ExtractMin();
- Union(H1,H2); (H1 e H2 sono due heap riunibili)
- DecreaseKey(x,k);
- Delete(x);

A livello teorico c'è un vantaggio da parte dell'Heap di fibonacci assolutamente da non poco, infatti,l'Heap di Fibonacci riesce ad eseguire tutte le operazioni in tempi ammortizzati costanti , eccetto la Delete e la ExtractMin in tempi logaritmici.(In realtà ciò vale se non utilizziamo una funzione find/search che deve cercare un nodo nella struttura e ci si può impiegare anche molto tempo)

Avviso: il codice reale sarà adattato per avere la possibilità di usare alberi che abbiano la proprietà di Max Heap oltre a Min Heap.

Heap di Fibonacci

L'Heap di Fibonacci è sostanzialmente una lista doppiamente concatenata e circolare di radici di alberi radicati che hanno la proprietà di Min-Heap ,ovvero la chiave di un padre è maggiore o uguale alla chiave di un figlio.

La struttura prende questo nome per 2 motivi principali:

- I numeri di Fibonacci sono usati nell'analisi dei tempi di esecuzione
- Un nodo ha grado massimo O(logn) (con n numero di nodi nella struttura) e la dimensione della sottostruttura per un nodo con grado k è di almeno $F_k + 2$ dove F_k è il k-esimo elemento della serie di Fibonacci.

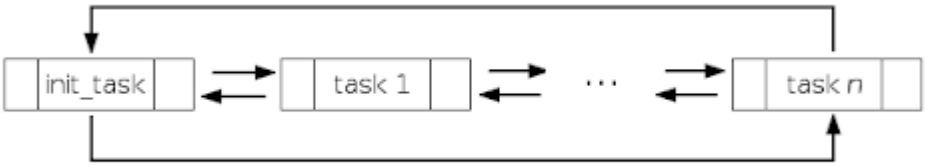
Per cosa può essere utile la struttura? E' utilizzata in alcuni algoritmi molto famosi appunto perché garantisce prestazioni in certe situazioni migliori di altri tipi di Heap. Un esempio di variante di algoritmo in cui è utilizzato è l'algoritmo di Dijkstra utilizzato per trovare i camminimi minimi in un grafo pesato. Questo è anche il motivo per cui spesso la struttura è implementata senza utilizzare la procedura di ricerca di un nodo che può risultare abbastanza pesante,infatti se inserissimo il nodo di un grafo nella struttura nelle successive visite al grafo avremmo diretto accesso al puntatore del nodo che ci permetterebbe di "ottenerlo" con tempo costante invece di ricercarlo nella struttura Heap

Osservare che qui non si usano alberi binari, ma alberi con un numero qualunque di figli.

Dunque a livello implementativo i figli costituiranno sostanzialmente una lista. Abbiamo già detto che la struttura conterrà dei nodi sostanzialmente,in ogni nodo troveremo questicì campi:

- Padre;
- Figlio; (è uno dei figli)
- Right;
- Left;
- Chiave;
- Grado;
- Mark;

Dove i primi 4 campi sono i puntatori agli altri nodi,se un nodo è nella lista delle radici non ha padre, se è una foglia non ha figli e potrebbe anche essere nella lista delle radici ma non avere figli. Invece i nodi fratelli sono sempre definiti dato che ci troviamo in una lista circolare doppiamente concatenata



Chiave: è la parte fondamentale del nodo, se avessimo un Nodo di tipo int ad esempio potrebbe essere 5, è la quantità che esso rappresenta.

Grado: è un numero che rappresenta quanti figli ha il nodo (all'inserimento sarà settato a 0)

Mark: è un attributo booleano, indica se il nodo ha perso un figlio dall'ultima volta che è diventato figlio di qualche altro nodo. (ovviamente un nuovo nodo inserito ha questo mark settato a false)

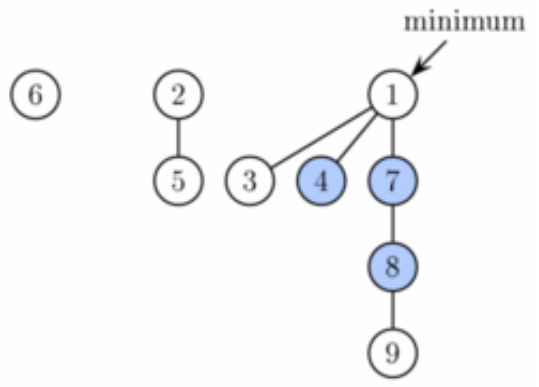
In questa implementazione oltre a questi attributi che sono quelli standard ne associamo un altro sempre di tipo booleano che è utile per la ricerca del Nodo nella struttura.

Da cosa è identificata la struttura?

L'elemento più importante della struttura è quello che chiamiamo Extreme, nella struttura con la proprietà Min Heap sarà il Minimo,mentre nella struttura con proprietà Max Heap sarà il Massimo.

Esso sarà un puntatore che punta all'elemento estremo della struttura(sta sicuramente nella lista più in "alto" che è detta Lista delle radici)

Esempio:



Inseriremo anche un ulteriore campo : numero_nodi che è ovviamente il numero di nodi nella struttura. Ulteriori campi : trovato(booleano) (ci aiuterà con alcune operazioni di ricerca),infinite (puntatore che serve a identificare un nodo che va eliminato(viene usato unicamente nella find,ovviamente di norma è settato a NULL)

Analisi delle prestazioni per la struttura

Come accennato prima, per questa struttura si analizzano i tempi di esecuzione con l'analisi ammortizzata,in questo campo il calcolo del tempo richiesto per eseguire un'operazione su una struttura dati è dato come media dei tempi di tutte le operazioni. Questo tipo di analisi non valuta dunque il caso medio di un'operazione , ma invece valuta le prestazioni medie di ogni operazione nel caso peggiore. Considera le operazioni della struttura come se fossero un "tutt'uno". Essa è utile per dimostrare che, contro le apparenze, il costo medio di un'operazione è più piccolo rispetto al costo calcolato con l'analisi del caso medio.

Per questa struttura si è soliti utilizzare il metodo del potenziale per analizzare le prestazioni di un Heap.

La funzione potenziale che usiamo è la seguente:

- $\Phi(H)=t(H) + 2m(H)$

t(H) : numero di alberi radicati nella struttura

m(H) : numero nodi marcati nella struttura

Nella struttura vedremo che avremo un limite superiore D(n) per il grado massimo che può assumere ogni nodo in una struttura Heap di Fibonacci. Si può dimostrare che questo D(n) è O(logn) , più nel dettaglio $D(n) \leq \text{floor}(\log_{\phi} n)$ dove ϕ è il numero aureo(che tra l'altro è un numero legato alla serie di Fibonacci)

Dimostrazione: capitolo 19-paragrafo .4 di Introduzione agli algoritmi e strutture dati. Cormen.

Inserimento di un nodo

La seguente procedura in pseudocodice inserisce un nodo con chiave key dentro alla struttura(in particolare nella lista delle radici)

Supponiamo di trovarci in una struttura Min heap(il caso max heap o generico è analogo)

Fib-Heap-Insert(key)

1. Sia x un nuovo nodo
2. x.chiave=key
3. x.figlio=NULL
4. x.Mark=false
5. If(Min==NULL)
6. {
7. Crea lista delle radici in cui c'è solo x
8. Min=x
9. }
10. Else
11. {
12. Inserisce x nella lista delle radici (inserimento in testa)
13. If(x.chiave < Min.chiave) Min=x
14. }
15. numero_nodi = numero_nodi + 1

Da come si vede la procedura fa semplicemente un inserimento in testa in una lista doppiamente concatenata circolare dunque ha tempo O(1)

Il potenziale è incrementato di 1

(Un inserimento alla fine della lista non avrebbero causato alcuna variazione ne nella lista nella logica della struttura per intero in quanto l'ordine dei nodi nelle liste è arbitrario)

Trovare il nodo extreme

L'operazione è molto semplice, basta restituire il puntatore a extreme(poi sarà max o min)

FindExtreme()

- return extreme;

La complessità è ovviamente O(1) e il potenziale rimane invariato

Unione di 2 Heap di Fibonacci

L'unione di 2 heap di Fibonacci è molto semplice concettualmente, prevede semplicemente di concatenare(l'ordine come al solito non ha importanza) le due liste delle radici.

Ci saranno 3 conseguenze in seguito all'unione:

1. I 2 heap di input saranno poi inutilizzabili
2. Se essi avevano n ed m nodi, l'heap di output avrà esattamente n+m nodi
3. Se ci troviamo in un heap di fibonacci con proprietà Min Heap, il nodo minimo della struttura di output sarà ovviamente il minore dei 2 minimi delle strutture di input

Pseudocodice:

Unione(H1 , H2)

1. If(H1 == NULL AND H2==NULL) return NULL
2. If(H1==NULL) return H2
3. If(H2==NULL) return H2
4. Allocare nuovo heap H3
5. H3=H1
6. Concatena lista delle radici di H2 con la lista delle radici di H1
7. H3.min=min(H2.min , H1.min)
8. H3.numero_nodi=H2.numero_nodi + H1.numero_nodi
9. Return H3

Il costo è ovviamente lineare e la variazione di potenziale è nulla.

Estrazione nodo extreme

L'estrazione del nodo extreme(valutiamo il caso particolare del min) è la prima operazione che comporta una ridistribuzione dei nodi che chiamiamo consolidamento degli alberi,alla fine nella lista delle radici tutti i nodi avranno un grado diverso.

Pseudocodice:

ExtractMin()

1. If(min==NULL) return NULL //struttura vuota
2. Nuovo nodo z=min
3. Aggiungere tutti i figli di min nella lista delle radici(ovviamente settando successivamente il padre di questi figli a NULL)
4. Rimuovere min dalla lista delle radici
5. If(z==z.right) min=NULL //struttura svuotata
6. Else
7. {
8. min=z.right
9. CONSOLIDATE()
10. }
11. numero_nodi=numero_nodi – 1
12. return z

La procedura CONSOLIDATE esegue il vero e proprio consolidamento ed opera finchè ogni nodo della lista delle radici non abbia grado distinto

L'idea generale consiste in questi due passi

1. Trovare due radici con lo stesso grado ,x e y
2. Il minimo di questi due nodi diventerà il padre dell'altro. (vengono ovviamente aggiornati a dovere puntatori e gradi)

La procedura usa dello spazio di memoria ausiliario , si può usare anche un semplice array,il problema che sorge è: quale è il numero minimo di celle da usare perché la procedura sia fatta sempre in modo safe?
La dimensione è D(numero_nodi) che è il limite superiore sul grado massimo, che come già detto è minore o uguale a floor($\log_\phi$(numero_nodi))

Pseudocodice:

CONSOLIDATE()

```
1.  A[D(numero_nodi)] nuovo array di nodi
2.  Inizializziamo ogni cella di A a NULL
3.  For ogni nodo iter della lista delle radici
4.  {
5.       d = Iter.grado
6.          While(A[d] != NULL AND iter!=A[d])
7.          {
8.              Nuovo nodo y=A[d]
9.              If(iter.chiave  >  y.chiave ) swap(iter,y)
10.             LINK(y,iter)
11.             A[d]=NULL
12.             d=d+1
13.         }
14.     A[d]=iter
15. }
16. min=NULL
17. for i=0 to D(numero_nodi)
18. {
19.     If(A[i]!=NULL)
20.     {
21.         If(min==NULL)
22.         {
23.             Creare lista radici che contiene solo A[i]
24.             min=A[i]
25.         }
26.         Else
27.         {
28.             Inserire A[i] nella lista delle radici
29.             If(A[i].chiave < min.chiave ) min=A[i]
30.         }
31.     }
32. }
```

Idea generale: il primo for serve a rendere ogni radice con un grado diverso, essa si serve della procedura di LINK, molto semplicemente LINK(y,x) fa diventare y un figlio di x(dopo aver rimosso y dalla lista delle radici) e il mark di y viene settato a false per come abbiamo definito le proprietà della struttura. Il secondo for dopo che siamo riusciti a costruire l'array A e abbiamo settato min a NULL(abbiamo fatto una sorta di svuotamento) serve per "ricostruire" la lista delle radici(non l'intera struttura)

Pseudocodice di LINK

LINK(y,x)

1. Togliere y dalla lista delle radici
2. Inserire y nella lista dei figli di x(x.grado aumenta)
3. y.mark=false

Date le 3 procedure vediamo la complessità di ExtractMin

NB le chiamate avverranno così:

- Inizio ->ExtractMin -> Consolidate ->LINK(un certo numero di volte) -> fine

La complessità di Extract Min è O(D(numero_nodi)) dunque al più
O(log(numero_nodi))

Variazione potenziale:

- Prima: t(H) + 2m(H)
- Dopo: D(numero_nodi) + 1 + 2m(H)

Ovviamente essi sono i due limiti, non è detto che coincidano esattamente

CAMBIAMENTO VALORE DI UN NODO

Questa è una delle operazioni che in linea teorica nasce per prendere in input il puntatore al nodo da modificare, cosa assolutamente plausibile dato che molte volte la struttura nasce per essere utilizzata in algoritmi che ci "forniscono" già i puntatori come Dijkstra, in questa versione però sarà preso in input non il puntatore ma un valore che verrà ricercato nella struttura per poi passare al vero e proprio algoritmo.

Questa piccola modifica causa un cambiamento molto importante per la complessità,quindi valuteremo la complessità in tutti e due i casi: ovvero il caso che non utilizza una procedura di ricerca e il caso che la utilizza.

Nella struttura con proprietà di Min Heap si tende a implementare la Decrease Key, al contrario nella struttura con proprietà Max Heap.

Pseudocodice DecreaseKey con ricerca:

DecreaseKey(vecchiaKey , nuovaKey) //DK #1

1. If(min==NULL) return; //struttura vuota
2. DecreaseKey(testa,vecchiaKey,nuovaKey) //è un'altra funzione
3. trovato=false //lo reimpostiamo a false dato che la seconda DK lo ha
   // cambiato a true

DecreaseKey(nodo x , vecchiaKey , nuovaKey)/versione leggermente modificata di find

1. if(x==NULL) return
2. if(trovato==true) return
3. x.C='Y'
4. nodo p1=NULL
5. nodo p2=NULL
6. if(x.chiave==vecchiaKey)
7. {
8. trovato=true
9. p2=x
10. x.C='N'
11. p1=p2
12. DecreaseKey(p1,nuovaKey) //è la DK #3
13. }
14. If(p2==NULL) //chiamate ricorsive
15. {
16. If(x.figlio!=NULL) DecreaseKey(x.figlio, vecchiaKey , nuovaKey)
17. If(x.right.C != 'Y') DecreaseKey(x.right , vecchiaKey , nuovaKey)
18. }
19. x.C='N'

DecreaseKey(nodo x , nuovaKey)

1. If(x.chiave > nuovaKey) return
2. x.chiave=nuovaKey
3. y=x.padre
4. If(y!= NULL AND x.chiave < y.chiave)
5. {
6. Taglio(x , y)
7. Taglio_cascata(y)
8. }
9. If(x.chiave < min.chiave) min=x

La procedura della DecreaseKey senza find è invece molto molto semplice

Pseudocodice:

DecreaseKey(nodo x , nuovaKey)

1. If(x.chiave < nuovaKey) return
2. x.chiave=nuovaKey
3. y=x.padre
4. if(y!=NULL AND x.chiave < y.chiave)
5. {
6. Taglio(x , y)
7. Taglio_cascata(y)
8. }
9. If(x.chiave < y.chiave) min=x

Taglio(nodo x,nodo y)

1. Rimuovere x dai figli di y (y.grado è decrementato)
2. Aggiungere x alla lista delle radici
3. x.padre=NULL
4. x.mark=false

Taglio_cascata(nodo y)

1. nodo z=y.padre
2. If(z!=NULL)
3. {
4. If(y.mark == false) y.mark=true
5. Else { Taglio(y , z) Taglio_cascata(z) }
6. }

Taglio e Taglio_cascata servono per risistemare l'albero dato che con un decremento potrebbero essere state causate delle violazioni(della proprietà Min Heap).

Semplicemente eseguiamo delle procedure di taglio , qui vengono utilizzati per bene i campi di mark

NB: x.mark è true se x ha perso un figlio dall'ultima volta che è diventato figlio di qualcun altro.

Taglio_cascata ci permette di "risistemare" un po' i mark del nostro albero radicato in modo tale da avere un limite superiore non troppo alto(esso dipende anche dal numero di nodi marcati)

Complessità senza Find:

- Taglio richiede tempo O(1)
- Taglio_cascata richiede tempo O(1) se escludiamo le chiamate ricorsive.
- La DecreaseKey impiega tempo O(1) più il numero di chiamate di Taglio_cascata ,supponiamo sia c.
- Si può dimostrare che la variazione di potenziale in seguito a c chiamate di Taglio_cascata è O(1) dunque il costo di DecreaseKey è O(1)

Complessità con Find:

- Taglio richiede tempo O(1)
- Taglio_cascata richiede tempo O(1) se escludiamo le chiamate ricorsive.
- La funzione di ricerca ha un tempo O(numero_nodi)
- La DecreaseKey impiega tempo O(numero_nodi) più il numero di chiamate di Taglio_cascata ,supponiamo sia c.
- Si può dimostrare che la variazione di potenziale in seguito a c chiamate di Taglio_cascata è O(1) dunque il costo di DecreaseKey è O(numero_nodi)

La variazione di complessità è notevole, si passa da avere un tempo costante ad avere un tempo lineare

CANCELLAZIONE DI UN NODO

Come implementazione per la cancellazione scriviamo una procedura chiamata Pisano-Delete (dal suo inventore) nata perché si supponeva che questa fosse una procedura più veloce della classica Delete ,in realtà poi si vede che non è asintoticamente migliore.

Anche qui bisogna precisare che la procedura cambia drasticamente complessità dal caso in cui venga dato un puntatore come imput al caso in cui invece venga data una chiave e debba essere fatta una ricerca del nodo nella struttura.

Pseudocodice(con find)

Pisano-Delete(k)

1. If(min==NULL) return
2. If(min.chiave==k){ extractMin() return }
3. find(testa,k)
4. nodo x = infinite
5. if(x==NULL) return
6. nodo y=x.padre
7. If(y!=NULL)
8. {
9. Taglio(y , x)
10. Taglio_cascata(y)
11. }
12. infinite=NULL
13. Aggiungere figli di x alla lista delle radici
14. Eliminare x

Pseudocodice(senza find)

Pisano-Delete(nodo x)

1. If(min==NULL) return
2. If(min==x){ extractMin() return }
3. if(x==NULL) return
4. nodo y=x.padre
5. If(y!=NULL)
6. {
7. Taglio(y , x)
8. Taglio_cascata(y)
9. }
10. Aggiungere figli di x alla lista delle radici
11. Eliminare x

Anche se non abbiamo ancora visto la find si vede bene che essa non ritorna il puntatore “trovato” ma piuttosto setta il dato infinite al nodo da ricercare(se non c’è è lasciato a NULL)

Dopo, le procedure di taglio servono a risistemare i nodi (cosa dovuta al fatto che x venga cancellato potrebbe causare violazioni)

Complessità: (n: numero_nodi)

Con find:

- O(n) (ovviamente dovuto alla ricerca che ha tempi abbastanza lunghi)

Senza find:

- O(logn)
- Si nota facilmente che il procedimento richiede l'utilizzo di ExtractMin dunque ha tempo O(logn)
- Potremmo avere dei miglioramenti rispetto alla procedura classica solo in casi particolari che non richiedono un numero elevato di "tagli", ma per il resto il limite superiore rimane il medesimo.

La procedura classica a livello teorico è molto semplice da implementare, richiede semplicemente di settare il nodo da cancellare a -∞ e poi eseguire una ExtractMin.

RICERCA DI UN NODO

La funzione di ricerca di un nodo come già accennato trova il nodo da cercare e setta il puntatore infinite a quel nodo, il puntatore infinite sarà dunque poi utilizzabile dalla struttura. Di solito la funzione di ricerca non è presente nelle implementazioni dato che essa a livello pratico per molte operazioni non è richiesta(basta usare i puntatori), ed essa ha un tempo di ricerca lungo, O(n)

Pseudocodice:

Find(nodo x , k)

```
1.  If(x==NULL) return
2.  x.C='Y'
3.  nodo p1=NULL
4.  nodo p2=NULL
5.  if(x.chiave==k)
6.  {
7.      p2=x
8.      x.C='N'
9.      p1=p2
10.     Infinite=x
11. }
12. If(p2==NULL) //chiamate ricorsive
13. {
14.   If(x.figlio!=NULL) Find(x.figlio, k)
15.   If(x.right.C != 'Y' ) Find( x.right , k)
16. }
17. x.C='N'
```

Ovviamente non causa variazioni di potenziale ma ha tempo O(n)

SVUOTAMENTO STRUTTURA

La procedura è molto semplice, eseguiremo semplicemente ExtractMin finchè la struttura non sarà vuota. A livello teorico si potrebbe ricorrere anche semplicemente a settare a NULL testa e min, ma a livello pratico andremmo in contro a un memory leak notevole

Pseudocodice:

Empty()

```
1. While(min!=NULL)
2. {
3.    ExtractMin()
4. }
```

Complessità: O(nlogn) dato che il ciclo è eseguito n volte e la ExtractMin ha tempo O(logn) (considerando n=numero_nodi)

METODI DI VISUALIZZAZIONE

Implementiamo 3 procedure:

- PrintListaradici(): stampa solo le radici
- Print(): stampa tutti i nodi della struttura
- Preorder(nodo x): procedura ricorsiva usata da Print() per la stampa.

Pseudocodici:

PrintListaradici()

```
1. if(min==NULL) return
2. nodo iter=testa;
3. STAMPA: "LISTA RADICI: “ iter.chiave " "
4. while(iter.right!=testa)
5. {
6.          iter=iter.right;
7.          STAMPA: iter.chiave" ";
8. }
```

Il tempo preso da PrintListaradici è O(n) ,poiché il caso peggiore è quello in cui tutti i nodi sono nella lista delle radici

Print()

```
1.  if(min==NULL) return
2.  nodo iter=testa;
3.  STAMPA: "  |radice : ";
4.  preorder(testa);
5.  for(iter=testa.right ; iter!=testa; iter=iter.right)
6.  {
7.          STAMPA:"  |radice : ";
8.          preorder(iter);
9.  }
10. STAMPA: "numNodi=" numero_nodi
```

Preorder(nodo x)

```
1.  if(x!=NULL)
2.  {
3.          STAMPA x.figlio ”  grado= " x.grado" ";
4.          if(x.figlio!=NULL)
5.          {
6.                  nodo iter=x.figlio;
7.                  STAMPA: "->[figli di " x.chiave " : ";
8.                  preorder(iter);
9.                  for(iter=x.figlio.right;iter!=x.figlio;iter=iter.right)
10.                         preorder(iter);
11.                 STAMPA: "] ";
12.         }
13. }
```

Le preorder e la print che cooperano assieme visitano tutti gli n nodi dunque il tempo è O(n)