

Strutture dati non lineari

Spicoli Piersilvio

Dipartimento di Informatica – bari

Parte 2

1. Alberi binari
2. Coda con priorità
3. Grafi

Alberi Binari

Definizione di albero Binario

Un albero Binario è un albero ordinato e radicato in cui ogni nodo ha al più due figli, opportunamente denominati come figlio destro e figlio sinistro di un nodo. Due alberi Binari T e U aventi gli stessi nodi, gli stessi figli per ogni nodo e la stessa radice, vengo definiti distinti qualora un nodo u sia designato come figlio sinistro di un nodo v in T e come figlio destro del medesimo nodo in U.

Equivalenza tra alberi n-ari e Binari

È sempre possibile rappresentare un albero n-ario ordinato T con un albero Binario B avente gli stessi nodi e la stessa radice: Infatti in B ogni nodo ha come figlio sinistro il primo figlio in T e come figlio destro il fratello successivo in T. Tale equivalenza vale esclusivamente ai fini della previsita in quanto le sequenze dei nodi esaminati su T e B coincidono se T e B sono visitati in previsita.

Visita degli alberi Binari

In aggiunta alle visite degli alberi n-ari, esiste la visita simmetrica: richiede prima la visita del sottoalbero sinistro (effettuata sempre con lo stesso metodo), poi l'analisi della radice, e poi la visita del sottoalbero destro.

Implementazioni Alberi Binari

Vettore: Utilizziamo un vettore che ha la radice in prima posizione e per ogni nodo p memorizzato in posizione i, se esiste il figlio sx è memorizzato in posizione $2*i$, se esiste un figlio destro è memorizzato in posizione $2 * i + 1$. Il figlio sx è sempre memorizzato in una posizione precedente rispetto al figlio destro. Per sapere i figli di un nodo mi basterà moltiplicare per 2 l'indice del nodo, così da conoscere il figlio sx, e moltiplicare per 2+1 per conoscere il figlio dx, complessità $O(1)$. Per conoscere il padre basterà fare il procedimento inverso. Potremmo avere sprechi di memoria se l'albero non è completo in quanto si avranno valori non associati. Un'ulteriore implementazione potrebbe essere quella di inserire una cella contenente valore booleano per indicare se il nodo è vuoto o meno.

Liste: Se l'albero è vuoto la lista sarà vuota, altrimenti la lista conterrà tre elementi (radice e due sotto liste che rappresentano i due sottoalberi). Se i due sottoalberi hanno dei figli, anche queste due sotto liste avranno delle sotto liste. Se un nodo non ha figli allora la sotto lista sarà vuota, questo meccanismo facilita i meccanismi di inserimento/cancellazione di alberi, in ricerca $O(n)$.

Liste per mezzo di array: Realizzazione simile a quella con cursori delle liste dove abbiamo un vettore in cui rappresentiamo la posizione, il valore del nodo, e se hanno un figlio dx/sx il loro indice e una variabile che indica l'indice del nodo radice. Facilita la lettura ma la scrittura diventa più complessa in quanto bisognerebbe fare degli shift $O(n)$.

Liste per mezzo dei puntatori: È una rappresentazione dinamica con lista attraverso puntatori: creo delle celle che contengono il valore, il puntatore al figlio sx, il puntatore al figlio dx e il puntatore al padre. Aggiornamenti facili ma con spreco di memoria.

Alberi Binari di Ricerca

Prima di definire il concetto di albero binario di ricerca, facciamo un richiamo all'algoritmo di ricerca binaria su un array ordinato A. L'algoritmo prevede che al primo passo si confronta l'elemento che si vuole ricercare con l'elemento al centro dell'array ordinato, ossia $A(n/2)$: dopo aver effettuato il primo passo, ricorsivamente, abbiamo la possibilità di restringere il campo di ricerca o alla destra o a sinistra dell'elemento centrale $A(n/2)$ in array. Immaginiamoci adesso di dover associare all'array A un albero T ordinato, dove la sua radice è proprio $A(n/2)$ e i valori successivi o precedenti compongono dei sottoalberi rispettivamente destri o sinistri. È facile quindi notare che, applicando una visita simmetrica sull'albero, corrisponderebbe alla ricerca sequenziale dell'array A, e che ricercare un elemento nell'array corrisponde a tracciare un cammino da un nodo radice alla foglia.

Definizione di Albero Binario di Ricerca

Un albero binario di ricerca è un albero composto per ogni nodo n presente, un elemento detto Elem(n), a cui è associata una chiave, detta chiave(n), appartenente ad un dominio di qualsiasi tipo di dato, purché sia ordinato. Tale albero si basa su due principali proprietà di ricerca che garantiscono, durante una visita simmetrica, le chiavi associate agli elementi in modo crescente:

- Le chiavi nel sottoalbero sinistro di n sono \leq chiave(n)
- Le chiavi nel sottoalbero destro di n sono \geq chiave(n)

Implementazione di un dizionario per mezzo di un ABR

Sia T un albero binario ordinato, con n – nodi e h altezza (profondità). Vediamo adesso come implementare un dizionario in modo efficiente per mezzo di un ABR; elenchiamo quindi tutte le operazioni principali:

Search ():

Partendo dalla radice, si traccia un cammino per cercare l'elemento con chiave k e per ogni nodo attraversato, si determina se proseguire la ricerca a sx o a dx verificando le proprietà di ricerca.

L'idea:

Si confronta la chiave k che stiamo cercando con chiave(n) dei nodi attraversati. Se $k = \text{chiave}(n)$, allora si restituisce l'elemento Elem(n). Altrimenti se $k < \text{chiave}(n)$, procediamo in modo ricorsivo con la ricerca a sx del nodo, oppure se $k > \text{chiave}(n)$ procediamo in modo ricorsivo con la ricerca a dx del nodo, finché non si trova l'elemento oppure se l'albero non ha più nodi da esaminare. Si nota che, similmente alla ricerca binaria, dopo ogni confronto fallito su k, la ricerca "scenderà" di livello ad esaminare i successivi sottoalberi (o a sinistra o a destra), perciò la complessità sarà pari all'altezza di T, ossia $O(h)$. Per poter determinarla, notiamo due casi:

- Se l'albero è molto profondo e sbilanciato ad uno dei lati, risulta $O(n)$ ed equivarrebbe ad utilizzare una lista in quanto si ricerca linearmente k
- Se l'albero è bilanciato sia in altezza che ai lati, avremo che per ogni nodo $|n| = 2^h$ e risolvendo la relazione con il numero di nodi si ottiene: $2^h \leq n \leftrightarrow h = O(\log(n))$, ossia complessità logaritmica.

Insert ():

Dato in input un certo nodo che contiene una coppia $\langle k, e \rangle$, crea un nuovo nodo e lo aggiunge all'albero come foglia nell'opportuna ricerca, che segue la relazione d'ordine.

L'idea:

- Si cerca il nodo n che diventerà padre del nuovo nodo u
- Si crea il nodo u con l'elemento "e" e chiave k . Dopodiché si appende u come figlio destro o sinistro di n rispettando la relazione d'ordine

Per poter realizzare (b), si utilizza una procedura che permettere di effettuare "append" del nuovo nodo (tale operatore è utilizzabile anche per la cancellazione). Per poter stabilire dove appendere u come figlio di n , abbiamo due casi:

- Se u è diverso da NULL \Rightarrow imposto n come padre di u
- Se n è diverso da NULL \Rightarrow controllo dove inserire u in base al valore di k , in modo ricorsivo
- Se $n = \text{NULL}$ \Rightarrow u sarà la nuova radice di T

Il passo (a) equivale ad effettuare una ricerca di k e richiede di conseguenza $O(h)$. Se dovessimo, inoltre, rappresentare T con dei puntatori ai figli, la (b) richiederebbe una modifica su un numero costante di puntatori e quindi $O(1)$.

Delete ():

Se il nodo n contenente l'elemento immesso in input ha al più un figlio, elimina n collegando il figlio di n all'eventuale padre. Altrimenti si scambia n con il suo predecessore ed elimina il predecessore scambiato. Per implementare tale operatore, useremo l'operatore di ricerca del predecessore di un nodo:

Richiamo: il predecessore di un nodo n è un nodo $u \in N: chiave(u) \leq chiave(n)$.

Sia quindi n il nodo contenente Elem(n) da cancellare. Distinguiamo tre casi:

- Se n è foglia basterà cancellare il nodo stesso
- Se n ha un unico figlio u suo figlio, e se n è anche radice, u diventerà la nuova radice dopo la cancellazione di n . Altrimenti, se n non è radice, si ricerca linearmente il padre di n , si cancella il nodo n e si appende u al genitore di n .
- Se n ha entrambi i figli, dobbiamo cercare di ricondurci ad uno dei casi precedenti. Per far questo si individua il predecessore di n , denotato u , e poiché il suo predecessore non può avere due figli, in quanto è il massimo del sottoalbero sx, non ci resta che "copiare" la chiave di u e metterla in n ($chiave(n) = chiave(u)$) e cancellare fisicamente il nodo u dall'albero

La complessità risulta $O(h)$ in quanto richiede la ricerca dell'Elem(n) da cancellare.

SearchMax () /SearchMin ():

- Per effettuare la ricerca del massimo in T basterà esaminare il cammino scendendo verso destra del sottoalbero destro, in modo ricorsivo in quanto T è ordinato.
- Per effettuare la ricerca del minimo in T basterà esaminare il cammino scendendo verso sinistra del sottoalbero sinistro, in modo ricorsivo in quanto T è ordinato.

Il tempo è pari a $O(h)$ pari all'altezza dell'albero.

Pred () /Succ ():

- a. Il successore di un nodo n è il più piccolo nodo $u \in N: chiave(n) \leq chiave(u)$. Per effettuare la ricerca del successore notiamo due casi: se n ha un figlio destro, allora $Succ(n) = u$ che è proprio il minimo del sottoalbero destro di n ; se n non ha un figlio destro, allora $Succ(n)$ sarà il primo alto antenato di u tale che n appartiene nel sottoalbero sinistro.
- b. Il predecessore di un nodo n è il più grande nodo $u \in N: chiave(u) \leq chiave(n)$. Per ricercare il predecessore, analogamente: se n ha un figlio sinistro, allora $Pred(n) = u$ che è proprio il massimo del sottoalbero sinistro di n ; se n non ha un figlio sinistro, allora $Pred(n)$ sarà il più basso antenato n sta nel sottoalbero destra di u .

In generale, $O(h)$.

Teorema: Il costo computazionale di modifica di tutti gli operatori precedenti di un albero T con altezza h è confinato ai nodi posizionati sull'eventuale cammino di ricerca, ossia $O(h)$.

■

Bilanciamento di un ABR (AVL)

Come già visto in Search(), Un albero T può avere una complessità computazionale variabile quando:

- a. T non è bilanciato, allora la complessità nel caso pessimo generale è $O(h) = O(n)$
- b. T è bilanciato, allora la complessità nel caso pessimo generale è $O(h) = O(\log(n))$

Considerando quindi un albero T bilanciato, a fronte di operazioni di modifica (Insert () / Delete ()), il suo bilanciamento può risultare mal condizionato all'aumentare del numero di operazioni e di conseguenza tale implementazione sul dizionario può risultare inefficiente. Possiamo dire quindi che il bilanciamento di un albero è dato se per ogni altezza di ogni sottoalbero sx e dx di ogni nodo interno differiscono al più un'unità. Denotiamo quindi l'albero Bilanciato con AVL. Tale albero manterrà per ogni nodo un "peso" di bilanciamento, definito come fattore di bilanciamento:

definizione: il fattore di bilanciamento associato al nodo $\beta(n)$ è dato dalla differenza fra l'altezza del sottoalbero sx e quella di dx di n :

$$\beta(n) = h(sx(n)) - h(dx(n))$$

Tale fattore è tanto migliore quanto basso il $|\beta(n)|$, mentre $\beta(n) = h$ quando T è sbilanciato. Quindi, un ABR è denotato AVL se $|\beta(n)| \leq 1$ per ogni nodo.

Code con priorità

Una coda con priorità è un tipo di dato che permette di mantenere il minimo (o il massimo) in un insieme di chiavi su cui è definita una relazione d'ordine totale. Le operazioni di modifica avvengono esclusivamente sull'elemento minimo o il massimo e le altre operazioni possibili sono la ricerca del minimo, l'estrazione del minimo ecc... Tale priorità è basata semplicemente sul rispettare una relazione d'ordine per la quale è basata la coda. Per semplicità, faremo coincidere le priorità con gli elementi stessi dell'insieme ed essa costituisce una caratteristica aggiuntiva associata ad ogni elemento.

Implementazioni coda con priorità (sequenziale)

La struttura dati può essere rappresentata con priorità n elementi utilizzando strutture sequenziali come liste ordinate o non ordinate. La complessità rimane invariata all'implementazione stessa della lista.

Implementazioni coda con priorità (ABR)

È possibile realizzare tale struttura mediante un heap binario:

Un heap binario è un albero binario radicato con le seguenti proprietà:

- a) È un AVR Perfettamente bilanciato
- b) Se k è il livello massimo delle foglie → l'albero ha esattamente $2^k - 1$ nodi di livello minore di k . Tutte le sue foglie di livello k sono addossate a sinistra. (da sinistra verso destra).
- c) Ogni nodo contiene un elemento maggiore di quello del padre

Min (): L'operatore restituisce il contenuto della radice dell'albero. La complessità è $O(1)$ in quanto la radice contiene l'elemento minimale.

Insert ():

L'operatore di inserimento si basa su diverse considerazioni che riguardano inizialmente il posizionamento del nuovo nodo (a) e il ripristino dell'ordinamento fra i livelli (b). Per poter effettuare l'inserimento, il nuovo nodo viene inserito come foglia dopo l'ultima trovata (da sinistra verso destra!). In tali circostanze quindi, si evince che:

- a) Se l'albero è vuoto → il nuovo nodo inserito sarà la nuova radice dell'albero
- b) Se l'albero è costituito dalla sola radice → il nuovo nodo sarà il figlio sinistro della radice
- c) Se l'ultima foglia dell'albero è un nodo figlio sinistro → il nuovo nodo inserito sarà il suo nuovo fratello

Tali circostanze sono banali, ma nel caso in cui siamo in presenza di un albero completo (ovvero che tutti i livelli dell'albero abbiano entrambi i figli), l'inserimento prevede una prima scansione che parte da destra, ovvero l'ultimo elemento foglia, e si propaga verso l'alto fintantoché non si raggiunge la radice oppure il nodo scansionato corrente sia un figlio destro; una volta che la scansione è arrivata alla radice, allora si procede con una seconda scansione che parte dalla radice stessa e scende fintantoché non si trovi una foglia. A quel punto sarà possibile aggiungere il nuovo nodo. Nel caso in cui siamo in presenza di un albero non completo (ossia che almeno un nodo dello stesso livello sia foglia e fratello di un nodo padre), allora si parte dall'ultima foglia inserita e si effettua una scansione procedendo verso l'alto a suo padre e si procede successivamente scansionando da sinistra verso destra tutti i fratelli del nodo padre fintantoché non si trova un fratello foglia. È facile notare che ci sia la possibilità che il nuovo nodo inserito non rispetti la (c) in quanto gli ordini di priorità non vengano sistemati. A questo punto, quindi, per poter ripristinare la (c) è possibile:

- a) Far partire la scansione dall'ultimo nodo foglia inserito ed effettuare il confronto con ogni nodo padre precedente. Nel caso in cui i confronti non rispettino (c) → la foglia viene scambiata col padre.
- b) Questo processo si ripete finché la (c) non è rispettata in tutti i livelli oppure che si è raggiunta la radice.

deleteMin ():

L'operatore consente di cancellare il nodo foglia di livello massimo più a destra dell'albero (in tal caso, la cancellazione avviene da destra verso sinistra!). Analogamente all'inserimento, si evince che se:

- L'albero è costituito dalla sola radice → l'albero viene cancellato tutto
- La radice ha un solo figlio, ossia il sinistro → l'albero sarà composto solo e soltanto dalla sua radice
- La radice ha entrambi i figli → la radice avrà come ultimo figlio solo e soltanto il figlio sinistro

Tali circostanze sono banali, ma se nel caso in cui siamo in presenza di un albero completo (ovvero che tutti i livelli dell'albero abbiano entrambi i figli), la cancellazione prevede una prima scansione che parte da sinistra e procede verso destra salendo di padre in padre finché il nodo scansionato non sia un figlio sinistro oppure non è stata raggiunta la radice; se nel caso in cui la prima scansione raggiunge solo la radice, allora si procede con una seconda scansione che scende di padre in padre finché non si trova il nodo foglia sinistro. Allora a quel punto sarà possibile cancellare il nodo. Nel caso in cui l'albero non sia completo, allora la ricerca partirà dall'ultima foglia inserita e si procederà da destra verso sinistra per i nodi fratelli al padre, finché non si raggiunge la precedente foglia di quella cancellata correntemente.

Teorema: In quanto l'inserimento e la cancellazione hanno bisogno di una prima scansione da destra (risp. Sinistra) e di una seconda scansione finché non si trovi una foglia fino a sinistra (risp. Destra), la complessità è pari a $O(\log_2(n))$ in quanto il campo di ricerca si restringe a destra (risp. Sinistra) verso sinistra (risp. Destra).

■

Grafi

Il grafo è una struttura di dati astratta composta da nodi e archi che rappresenta una relazione binaria sull'insieme costituito da nodi. In generale, i nodi sono usati per rappresentare oggetti e gli archi per rappresentare relazioni tra coppie di oggetti. Definiremo successivamente due tipologie di grafi:

- Grafi orientati: rappresentano delle relazioni orientate fra le coppie di oggetti. Ogni arco, quindi, è orientato da uno dei due nodi verso l'altro nodo, e si disegna con una freccia che ne indica la direzione (la commutatività delle coppie non è valida).
- Grafi non orientati: in questo caso, le coppie dei nodi non hanno una orientazione. Ogni coppia di nodo, quindi, è composta da un nodo simmetrico, privo di direzione e "commutativo" (nel senso che cambiando l'ordine dei nodi, la relazione è valida).

Un grafo non orientato può essere rappresentato come grafo orientato definendo due archi invece che uno, da un nodo all'altro.

Definizione: Un grafo $G = (N, A)$ dove N è l'insieme dei nodi finito e non vuoto e $A \subseteq N \times N$ l'insieme finito e non vuoto di archi che formano le coppie fra i nodi.

- Con $n = |N|$ per indicare il numero di nodi.
- Con $m = |A|$ per indicare il numero di archi.
- Con $N(G)$ l'insieme dei vertici del grafo G .
- Con $A(G)$ l'insieme degli archi del grafo G .
- Con $A(n)$ l'insieme degli archi che congiungo al nodo n .
- con $\text{Elem}(n)$ indicheremo il peso del nodo (l'informazione associata al nodo) se G è pesato.
- con $\text{Arc}(n)$ indicheremo l'arco uscente da n contenente il peso dell'arco, se G è pesato.

Definizioni preliminari

- a) incidenza: Se esiste un arco che congiunge due nodi distinti, diremo che l'arco (n, u) è incidente sui nodi n e u . Se G è orientato ed esiste l'arco fra n e u , diremo che l'arco (n, u) esce da n ed entra in u .
- b) Adiacenza: Se G non è orientato ed esiste un arco (n, u) che congiunge due nodi, diremo che n è adiacente a u e u è adiacente a n . Se G è orientato, diremo (solo!) che u è adiacente a n .
- c) Grado di un nodo: Se G non è orientato, il grado di un nodo è dato dal numero di archi incidenti su un nodo n , e verrà denotato con $\delta(n)$. Se G è orientato, un nodo avrà a disposizione un grado in entrata di archi e un grado in uscita di archi e saranno denotati come $\delta_{in}(n)$ e $\delta_{out}(n)$.
- d) Cammini e cicli: vedi paragrafo alberi.

Implementazioni Grafi

Matrice di adiacenza:

Tale implementazione consente di verificare più velocemente la presenza di un arco, ma d'altro canto è da considerare il fatto che ci sia spreco di memoria per alcune celle che non rappresentano l'arco. Assumiamo quindi che i nodi del grafo corrispondano ai numeri che vanno da $1 \dots n$ e consideriamo una matrice quadrata $n \times n$ cui le righe e le colonne rappresentano i nodi del grafo: le celle della matrice $M[n, u]$, con (n, u) arco che esce da n e va verso u , varranno 0 se non esiste l'arco che unisce (n, u) , altrimenti varrà 1. Per i grafi non orientati la matrice di adiacenza sarà simmetrica in quanto $M[n, u] = M[u, n]$ per ogni nodo e per i grafi pesati, si memorizzeranno nelle celle della matrice i pesi degli archi denotati come $p_{(n,u)}$ se e solo se l'arco esiste fra n e u , altrimenti varrà più o meno infinito. La forza di questa struttura è che è possibile determinare in tempo $O(1)$ la presenza di un arco fra due nodi in quanto basta accedere direttamente con $M[n, u]$ ma, se dobbiamo esaminare tutte le adiacenze di un nodo il tempo sarà $O(n)$ in quanto dovremmo scandire l'intera riga del nodo. Per inserire e cancellare un nodo nella matrice il tempo è di $O(n^2)$ se la struttura è statica, ma con il raddoppiamento e dimezzamento della memoria possiamo rendere dinamica la riallocazione dei nodi, in tempo ridotto a $O(n)$.

Matrice di incidenza:

la matrice di incidenza è una matrice costituita $n \times m$ elementi dove le righe rappresentano i nodi e le colonne rappresentano ogni arco del grafo. La cella della matrice varrà 1 se l'arco e il nodo sono incidenti, 0 altrimenti se G non è orientato (ogni colonna avrà almeno due volte 1!); se G è orientato, la cella della matrice varrà +1 se l'arco entra nel nodo, -1 se l'arco esce dal nodo, 0 altrimenti. Poiché con m rappresentiamo tutti i possibili archi dei nodi, la matrice non sarà quadrata e ne simmetrica. Per poter inserire un nodo o un arco il tempo di esecuzione sarà $O(nm)$ in quanto all'aumentare del numero di nodi e archi, lo spazio cresce. Per rimuovere e trovare un arco, il tempo è $O(n)$ poiché basta scandire la colonna dell'arco corrispondente. Per rimuovere un nodo il tempo sarà $O(nm)$ e per verificare la presenza di uno o più archi fra due nodi basterà scansionare la\e colonna\e della matrice il tempo sarà $O(m)$.

Vettore di adiacenza:

risolve il problema della raccolta e la gestione dei nodi adiacenti ad un nodo n . Rappresentiamo quindi un grafo con due array monodimensionali, uno per i nodi e uno per gli archi. Nel primo array memorizziamo la posizione relativa al secondo array con cui sarà possibile cercare i nodi adiacenti all'elemento del primo array, dove per ogni elemento del primo array avremo 1 o più celle del secondo array. Sapere quale sia il nodo adiacente di n dipenderà dal grado medio dei nodi. Per aggiungere o rimuovere un arco il tempo sarà lineare poiché bisogna aggiornare il secondo array "shiftando" gli elementi. Per grafi non orientati la dimensione del secondo array cresce in modo proporzionale in quanto l'adiacenza è simmetrica. In generale tale implementazione è utile se e solo se non effettui inserimenti o cancellazioni degli elementi.

Liste di adiacenza:

Ogni nodo n del grafo ha una lista contenente i suoi nodi adiacenti, ovvero tutti i vertici u per cui esiste (n, u) . Risulta quindi più semplice trovare gli archi incidenti su un particolare nodo e per questo è sufficiente scansionare la lista di adiacenza del nodo in tempi lineari. È possibile osservare che:

- a) Per ogni nodo n corrispondono n liste di nodi adiacenti
- b) Se G è orientato \rightarrow se la lunghezza della lista è m ossia il numero di archi, in quanto gli archi non sono simmetrici, allora il tempo di ricerca del nodo adiacente è $O(m)$
- c) Se G non è orientato \rightarrow se la lunghezza della lista è $2m$ ossia il doppio per ogni arco in quanto ogni arco è simmetrico, allora il tempo di ricerca del nodo adiacente sarà $O(2m)$
- d) In entrambi i casi la complessità risulta lineare ma in termini di spazio la complessità è pari alla somma del numero di nodi e il numero di archi, $O(m + n)$

Per visitare il grafo, e quindi ricercare tutti i nodi incidenti di un nodo n , il tempo necessario sarà pari al grado del nodo n $O(\delta(n))$ sia se G è orientato o meno. Per verificare la presenza di un arco fra due nodi, il tempo necessario nel caso peggiore risulterà il minimo fra il grado del primo nodo e del secondo, quindi $O(\min\{\delta(n), \delta(u)\})$. Per rendere efficiente questa operazione basta mantenere le liste ordinate. Per inserire e cancellare un arco o un nodo il tempo risulta $O(1)$.