Strutture dati non lineari

Spicoli Piersilvio

Dipartimento di Informatica – Bari

Parte 1

1. **Insieme**
2. **Dizionario**
3. **Introduzione ai grafi e le loro proprietà**
4. **Albero n - ari**

Insiemi

Definizione di insieme

Un insieme è una collezione finita o infinita (o famiglia di elementi) di tipo omogeneo. A differenza di una lista, essa non consente di memorizzare elementi duplicati e ogni elemento non è caratterizzato da una posizione al suo interno. Formalmente, un insieme può essere definito o estensionalmente enumerando ogni elemento al suo interno:

$$Nomi\ propri\ = \{Piersilvio, Ilenia, Giulio, Marco\}$$

Oppure intensionalmente definendone una proprietà (assioma) che ne definisce l'appartenenza a tale insieme:

$$(insieme\ dei\ numeri\ reali\ compresi\ fra\ 0\ e\ 1)\ A\ = \{x\ \in\ R : 0\ \leq\ x\ \leq\ 1\}$$

Per indicare il numero di elementi presenti in un insieme, denotiamo come segue la sua cardinalità:

$$|nomi\ propri|\ =\ 4$$

E la relazione fondamentale su cui si basa il concetto di insieme è quella di appartenenza di un elemento, ovvero se l'elemento appartiene o meno all'insieme ($x\ \in\ R$), da cui ne deriva il concetto di sottoinsieme, dove può essere ritenuto tale se il sottoinsieme contiene gli stessi elementi dell'insieme originale. Le operazioni che si possono effettuare fra due insiemi sono l'unione, intersezione e differenza. Diamo un'idea illustrativa:

Implementazioni insieme

Esistono due tipologie di implementazioni: una con array booleani o con liste, ma esistono altre più efficienti, ossia per dizionario o per albero bilanciato.

Array booleano: Tale implementazione viene utilizzato solo per interi all'interno di un insieme A cui la dimensione è compresa fra [1, n] per mezzo di un array di n – elementi di tipo booleano che consente di memorizzare lo stato di un singolo elemento in vero, se l'elemento appartiene all'insieme, oppure in falso, se l'elemento non appartiene. In questo modo si definisce un array caratteristico basato sul concetto di appartenenza di un insieme. Le operazioni principali (unione, intersezione e differenza) hanno costo computazionale $O(1)$ (atomicamente) in quanto è possibile utilizzare gli operatori di confronto messi a disposizione del linguaggio, ma nel caso in cui vengono applicate su due insiemi, la complessità risulterà $O(n)$. Le operazioni di inserimento e cancellazione hanno costo $O(1)$ in quanto l'array prevede l'accesso diretto e l'operatore di appartenenza è in tempi costanti. Come già detto, l'operatore di appartenenza ha costo $O(1)$. Tale discorso è analogo al controllo per l'insieme vuoto, ma per aggirare la complessità lineare, è possibile implementare un contatore che rimane aggiornato a inserimento e cancellazione, affinché l'insieme vuoto controlli tale contatore potendo avere così $O(1)$. Il principale svantaggio è che un array inizializzato peserebbe con celle a false in quanto vuoto, che porterebbe ad uno spreco di memoria.

Liste: tale rappresentazione ha due tipologie di vie: con liste ordinate o non. Può essere una valida soluzione, essendo che la lista è una struttura dinamica, si limiterebbe a memorizzare solo gli elementi che sono attualmente presenti nell'insieme. Con la lista, la memoria aumenta linearmente agli spostamenti fatti durante la cancellazione e l'inserimento di un elemento, avendo complessità quindi di $O(n)$. Ogni elemento è rappresentato per mezzo di una coppia elemento – indirizzo successivo, in modo collegato e l'inserimento può avvenire esclusivamente in prima posizione della lista, dopo aver effettuato il controllo di appartenenza dell'elemento, avendo così una complessità di $O(n)$ in quanto si scandirebbe l'intera lista. Nel caso in cui la non appartenenza dell'elemento sia nota a priori, l'inserimento vale $O(1)$. Analogo discorso è valido per la cancellazione. Gli operatori di unione, intersezione e differenza hanno complessità $O(n\ *\ m)\ =\ O(n)$ (con m dimensione massima del secondo insieme!) in quanto bisogna scandire entrambe le liste di due insiemi. L'implementazione descritta è basata su liste collegate non ordinate (non efficiente). Se invece è definita una relazione di ordinamento sugli elementi di un insieme, ossia una lista collegata e ordinata, la complessità di unione, intersezione e differenza scenderebbe a $O(n\ +\ m)$. Gli altri operatori avrebbero la complessità invariata a $O(n)$ (efficiente).

Dizionario

Definizione di dizionario

L'applicazione precedente di un insieme può essere contestualizzata in base alla sua applicazione, in quanto, c'è possibilità che non tutte le operazioni possano essere utilizzate. Basti considerare per esempio un insieme di parole cui è definita il suo significato semantico. Certamente, questo problema non include l'utilizzo di operatori come unione ecc... Per questo è possibile definirne un dizionario, ossia un sottotipo con operazioni limitate di tipo insieme. Ogni elemento è un tipo strutturato ai quali è possibile accedere per mezzo di un riferimento a un campo chiave e ogni elemento in un dizionario è rappresentato per mezzo di una coppia <chiave, valore>. La chiave di un elemento è legata alla tipologia di applicazione del dizionario, che possa indicare un oggetto oppure un semplice "intero". L'utilizzo specifico avviene nelle tavole dei simboli, dove la chiave può essere utilizzata come identificatore di una parola-chiave di un linguaggio di programmazione. Le operazioni principali devono consentire quindi:

- La verifica di appartenenza di una chiave
- L'inserimento di una nuova <chiave, valore>
- La cancellazione di un valore per mezzo della chiave
- La ricerca di un valore per mezzo della sua corrispondente chiave
- La modifica di un valore per mezzo della sua chiave

Implementazioni dizionari

Essendo un sottotipo di insieme, possiamo rappresentare i dizionari anche per mezzo del array booleano e di liste ordinate o no. A differenza, però esistono realizzazioni più efficaci ovvero con array ordinato o con tabelle hash.

Array ordinato: l'implementazione prevede un cursore d'appoggio che punta all'ultima posizione occupata dell'array, e con una relazione di ordinamento totale applicato sulle chiavi, memorizziamo ogni chiave in modo contiguo. Ogni cella dell'array contiene un riferimento aduna coppia chiave/valore (un oggetto di tipo Pair) e le operazioni di inserimento e cancellazione avvengo in modo diretto, cosicché ogni qualvolta venga cancellato o inserito un elemento, la relazione d'ordine rimanga attiva. L'operazione di ricerca di un elemento prevede la ricerca di appartenenza di una chiave ad un elemento, e per fare ciò, assumiamo di voler utilizzare la ricerca binaria, in quanto abbia una complessità in termini di caso pessimo $O(log(n))$. La ricerca avviene in questo modo: si consideri una generica chiave k e la chiave centrale rispetto l'ordinamento x.

L'algoritmo prevede il confronto fra k e x:

- Se $k = x$ la chiave che stavamo cercando è stata trovata
- Se $k < x$ => spezzo l'intervallo su cui ho ricercato a destra di x e ricerco ricorsivamente
- Se $k > x$ => spezzo l'intervallo su cui ho ricercato a sinistra di x e ricerco ricorsivamente

La complessità di inserimento e cancellazione hanno $O(1)$ in quanto la struttura prevede l'accesso diretto ad una posizione e se le relazioni d'ordine sono rispettate, ma ha complessità $O(n)$ se nel caso in cui la chiave dell'elemento inserito/cancellato non rispetti la relazione d'ordine presente in array, producendo quindi uno spostamento degli elementi.

Tabelle hash: l'implementazione prevede l'utilizzo della tecnica di hashing dove le operazioni di ricerca e modifica avvengono in modo diretto e indipendente sia dalle dimensioni del dizionario che dagli elementi in tempo costante. Il principio su cui si basa l'implementazione è il seguente: data la chiave di un dizionario, ricavare la posizione corrispondente all'interno di un array. Esistono due tipologie di varianti sulla quale è possibile lavorarci in modo statico oppure in modo dinamico: il primo metodo fa uso di una struttura a dimensione prefissata mentre il secondo fa uso di una struttura a dimensione variabile nel momento in cui viene aggiunto o tolto un elemento. Incentriamo l'attenzione riguardo l'hashing di tipo statico. A sua volta prevede due tipologie di implementazione, ossia:

- Hash con struttura chiusa, dove abbiamo possibilità di poter inserire un insieme limitato di elementi in uno spazio a dimensione fissa
- Hash con struttura aperta, dove abbiamo la possibilità di memorizzare un qualunque numero di elementi in uno spazio "illimitato"

Analizziamo la tecnica di hash chiuso: dato un dizionario composto da coppie <chiave, valore>, possiamo rappresentare ogni elemento per mezzo di un array monodimensionale composto da n-celle (maxbucket) che fungono da contenitori di dimensioni uguali, denominati bucket. Ogni bucket può contenere al suo interno un numero massimo di coppie <chiave, valore>, denominato nb, che per semplicità assumiamo che nb = 1 (una singola coppia per bucket). Per poter gestire tali contenitori, viene utilizzata una funzione di mapping che calcola la posizione del bucket in corrispondenza della chiave contenuta nel dizionario all'interno dell'array. La lunghezza dell'array è data dal numero delle chiavi che il dizionario contiene e denotiamo con |k| il numero di chiavi che ammette il dizionario, n la dimensione dell'array. Per poter gestire tale array la funzione di mapping è la seguente:

$$h: k \ \rightarrow \{1, \dots, n\}$$

La posizione della chiave del dizionario corrisponde al valore h(k) che corrisponde alla posizione all'interno dell'array. Come abbiamo già detto precedentemente, se assumiamo che n = |k| si ha una garanzia bigettiva in quanto ogni chiave avrà una posizione corrispondente all'interno dell'array, ma se nel caso in cui $|k| >> 0$, potremmo avere un vero e proprio spreco di memoria. Il miglior compromesso quindi per garantire la bigettività è che il numero di celle dell'array sia $1 < n < |k|$.

Le operazioni che possiamo effettuare sono le seguenti:

- La ricerca di appartenenza di un elemento ha complessità $O(1)$ in quanto è possibile mappare la sua chiave in modo diretto
- La ricerca di una cella libera ha costo $O(n)$ in quanto bisogna effettuare una scansione delle celle che hanno una chiave fittizia detta "libero", analogo per le celle cancellate
- la cancellazione ha costo $O(1)$ in quanto si accede in modo diretto alla chiave che si vuole cancellare e sostituire la sua posizione con chiave fittizia "cancellato". Il problema che sorge è che se si dovesse eliminare un elemento, automaticamente verrebbe segnato la chiave fittizia "cancellato", non vuol dire che quella cella non sarà mai inutilizzata, e che la scansione avvenga esclusivamente alle chiavi fittizie "libero" per poter inserire una nuova chiave. Bisogna quindi estendere tale scansione, non solo alle chiavi "libere", ma anche a quelle "cancellate".
- durante la fase di inserimento, qualora la chiave che si sta inserendo sia già esistente, può avvenire una collisione fra gli elementi in quanto date due chiavi diverse, esse corrispondono alla stessa cella nell'array. In generale non esistono funzioni hash che evitano tale problema, ma è possibile aggirarlo per mezzo delle tecniche di scansione. Assumiamo per semplicità di voler implementare una scansione lineare: se durante la fase di inserimento la posizione all'interno dell'array è già occupata si effettua una ricerca al suo successivo e così via, fintantoché non si raggiunge la fine dell'array, e una volta raggiunta, si effettua la scansione a ritroso. È chiaro che in termini di complessità per poter ricercare celle con "libero" otteniamo $O(n)$. Tale tecnica nel tempo può produrre agglomerati di chiavi nelle stesse celle.

Abbiamo visto quindi che indipendentemente dalle funzionalità, per gestire una collisione occorre:

- una buona funzione hash che in termini di computazione deve essere rapida e sicura
- una buona tecnica di scansione sicura e veloce per distinguere le chiavi fittizie (ossia lineare, quadratica ecc..)
- il numero di contenitori (n) deve essere una sovrastima delle chiavi che il dizionario deve contenere (ma sapendo che può produrre spreco di memoria)

Analizziamo la tecnica con hash aperto: l'implementazione prevede sempre un array di contenitori predefiniti solo che ogni contenitore sarà implementato come se fosse una lista collegata, o detta a "trabocchi".

Alberi n – ari

In molte situazioni reali trattiamo come collezione di oggetti, delle reali situazioni su cui è definita di solito una relazione gerarchica. Possiamo fare degli esempi: uno di questi può essere la progettazione top-down di problem solving, un albero genealogico di ogni persona oppure un albero di chiamate ricorsive dove ogni nodo richiama una funzione propria. Gli alberi sono definiti sotto la teoria dei grafi, per le quali valgono, in determinate situazioni, le loro proprietà.

Definizione di albero n – ario e le sue proprietà

Un albero (radicato) è una coppia T = <N, A> costituita da un insieme N di nodi e da un insieme di coppie ordinate di nodi A ⊆ NxN dette archi. In un albero n – ario:

- ogni nodo v (tranne la radice) ha un solo genitore u (detto padre) tale che <u, v> ∈ A.
- Un nodo padre u può avere zero o più figli v tali che <u, v> ∈ A e il loro numero di figli di un nodo padre u viene detto grado di u.
- Un nodo senza figli è detta foglia
- Un nodo che non sono né foglie né radice, ossia i nodi padri, sono detti nodi interni
- Un "antenato" di un nodo è ritenuto tale se composto da un cammino che prosegue da una foglia verso il padre (basso verso l'alto)
- Un "discendente" di un nodo è ritenuto tale composto da un cammino che prosegue da un nodo padre ad una foglia (alto verso il basso)
- La profondità di un albero è data dal numero di archi che parte dalla radice fino ad un certo nodo (radice r ha profondità 0, un nodo k ha profondità k + 1)
- Il numero di archi è dato da $|A| = |N| - 1$
- T si dice radicato se è possibile disegnare un albero libero (con coppie non ordinate, ossia un grafo) arbitrariamente un nodo radice r e ordinando i nodi per livelli
- Nodi con lo stesso genitore sono detti fratelli con ugual numero di profondità e l'altezza di un albero è data dalla sua profondità maggiore
- T è un grafo aciclico e connesso (ossia che ogni nodo eccetto la radice r ha un solo arco entrante)
- Può essere ordinato stabilendo per ogni livello una relazione di ordinamento

Definizione di albero n – ario radicato ricorsivo (rigorosa)

Sia dato un albero T ordinato. T è un particolare di grafo orientato (anche vuoto) tale che esista un nodo radice senza predecessori e composto per ogni n ≥ 0 nodi successori a1, …, an e i nodi successori sono definiti ricorsivamente in n – sottoalberi disgiunti T1, ..., Tn aventi i nodi successori a1, …, an rispettivamente come radice.

Visita degli alberi

La visita di un albero consiste nel seguire una "regola" di esplorazione (ricerca) dei nodi di un albero in modo che ogni nodo sia visitato esattamente una volta. Esistono quindi due tipologie di algoritmi per le visite:

- in profondità (DFS) dove vengono visitati i rami uno dopo l'altro da sinistra verso destra
- in ampiezza (BFS) dove la visita avviene a livelli partendo dalla radice

Sia quindi T un albero non vuoto con radice r. Se r non è foglia e ha k figli (k>0), e i sottoalberi T1, …, Tk aventi come radici i figli di r, gli ordini di visita sono:

- prevista (ordine anticipato): consiste nell'esaminare partendo dalla radice r e poi ricercare in ordine i sottoalberi T1, …, Tk leggendo i nodi successivi da sinistra verso destra.
- Post visita (ordine differito): consiste nel leggere in post ordine i nodi dei sottoalberi T1..., Tk e poi esaminare la radice r alla fine
- invisita (ordine simmetrico): consiste nell'effettuare l'ordine nell'ordine, la invisita di T1..., Ti, nell'esaminare r, e poi effettuare, nell'ordine, la invisita di Ti+1, …, Tk per ogni i ≥ 1

I primi tre algoritmi sono basati sull'utilizzo di profondità ricorsiva di BFS. Per quanto riguarda BFS, la tecnica algoritmica per effettuare la visita avviene per livelli, da sinistra verso destra. Si consideri che per ogni tecnica utilizzata la complessità in termini di tempo vale O(n) in quanto bisogna visitare tutti i nodi precedenti per arrivare alla foglia.

Implementazioni albero n-ario (indicizzata)

Vettore – padre:

la rappresentazione è di tipo indicizzata e richiede un certo spazio per memorizzare n-nodi, avendo quindi in termini di spazio una complessità $O(n)$. L'idea di base è quella di rappresentare ogni nodo dell'albero con una cella di un array che contiene l'informazione associata al nodo. L'implementazione a riguardo prevede quindi un array che contiene per ogni nodo presente nell'albero una struttura composta da una coppia <padre, informazione> dove il campo padre punta all'indice del nodo padre e informazione contiene il valore del nodo figlio. Nel caso in cui si tratti di un nodo radice, padre punterà a NULL. Osserviamo che da ogni nodo è possibile risalire, sapendo la posizione corrente, al proprio nodo padre, avendo $O(1)$. Se nel caso in cui la ricerca (visita) avvenga su un figlio avremo complessità $O(n)$ in quanto scandiremmo tutti i nodi. In termini di inserimento e cancellazione risulterebbe difficoltosa l'aggiunta di nuovi nodi con altrettanto sottoalberi. L'operatore padre (u, T) risulterebbe costante $O(1)$ in ogni caso.

Implementazioni albero n-ario (collegata)

La rappresentazione in questione rimane la più flessibile in quanto permettono di tenere in modo costante inserimento e cancellazioni. L'idea di base è quella di rappresentare ogni nodo dell'albero con un record che contiene l'informazione associata al nodo, più altri puntatori che consentono di raggiungere gli altri nodi. Vediamole in specifico:

Liste di figli:

Se il numero di figli m di ogni nodo non è noto a priori, è possibile associare ad ogni nodo padre una lista collegata ai nodi figli. Per poter visitare un nodo la complessità risulterebbe $O(n)$, sapendo dove si trova il padre (padre(u,T) = $O(n)$), in quanto scandiremmo la lista collegata ai figli per poterci risalire. Se invece non sappiamo la posizione del padre, bisognerà scandire tutte le liste alla ricerca dell'unica lista che contiene il nodo padre, che è in prima posizione, la complessità rimane invariata.

Liste di primi figli/fratelli:

Per evitare di mettere ulteriore carne alla brace, senza dover implementare un ulteriore struttura dati come ad esempio la lista (è chiaro che è possibile farlo, nulla ci vieta!), possiamo associare ad ogni nodo presente nell'albero un puntatore al primo figlio (posto a NULL se non ha figli) e un puntatore al primo fratello (posto a NULL se non ha fratelli). In termini di spazio avremo in ogni caso $O(n)$ in quanto, oltre a sprecare memoria allocando e deallocando, la dimensione sarà comunque proporzionale al numero di nodi che ha. Per poter accedere ai nodi successivi, sapendo la posizione del nodo padre corrente, è possibile accedervi in modo rapido conoscendo il puntatore al figlio e al fratello. Se nel caso in cui non sappiamo dove si trovi il nodo padre, la complessità risulterebbe $O(n)$ in quanto scandiremmo per due volte entrambi i puntatori. Una soluzione a tale problema sarebbe quella di aggiungere un nodo ausiliario che consente di memorizzare la posizione del padre, aumentando d'altro canto lo spreco di memoria in quanto verrà associato un array di puntatori con i riferimenti ai suoi figli. Per scandire tutti i figli di un padre, basta posizionarsi sul nodo del primo figlio e scorrere tutti i suoi fratelli successivi, avendo quindi $O(n)$.

Vettori di figli: tale implementazione prevede per ogni nodo un puntatore che punta al padre (NULL se il nodo stesso è radice) e un array monodimensionale di puntatori che puntano in modo ricorsivo ai nodi successivi. I riferimenti dei figli vengono memorizzati da sinistra verso destra e se nel caso in cui un nodo non abbia figli, allora la cella corrispondente punterà a NULL. In termini di complessità spaziale, l'implementazione risulta efficiente se viene stabilito l'ordine di ogni nodo (il numero di figli per ogni nodo), ma se l'ordine di ogni nodo non fosse noto a priori, allora otterremmo spreco di memoria. In termini di operazioni, per ritrovare il padre otterremmo $O(1)$ in quanto è possibile accedere in modo diretto e $O(n)$ per ritrovare un figlio di un nodo, in quanto si effettua la scansione del vettore dei figli del nodo.

Puntatore padre, primo figlio/fratello: l'implementazione prevede per ogni nodo una struttura composta da tre puntatori:

a) un puntatore al padre (NULL se il nodo stesso è radice)
b) un puntatore al primo figlio (NULL se il nodo stesso non ha alcun figlio
c) un puntatore al fratello, che si collega in modo ricorsivo ai fratelli successivi (NULL se il nodo stesso non ha alcun fratello.

In termini di complessità temporale, per poter risalire ad un nodo interno dell'albero, ci basterà scandire tutti i fratelli successivi al primo figlio, quindi $O(n)$. Nel caso in cui si voglia invece risalire al primo figlio, il tempo sarà O(1) e analogo per risalire al padre.

Implementazioni albero n-ario (collegata e dinamica)

In questa implementazione, è possibile aggiungere una lista dinamica definita su ogni livello dell'albero dove la radice sarà il primo elemento della lista e i successivi saranno puntatori ai sottoalberi della radice, e ricorsivamente, ogni radice del sottoalbero sarà il primo elemento della lista. La rappresentazione prevede, all'interno della lista, un numero di nodi ausiliari pari al numero delle radici dei sottoalberi (oppure delle foglie) e i nodi effettivi, che sarà composta da una coppia <informazione, pt> con pt che punterà al primo nodo ausiliario del sottoalbero (NULL se foglia). I nodi ausiliari sono composti invece da due puntatori: uno a inizio lista del sottoalbero, e l'altro al fratello.

Sì ha comunque uno spreco di memoria, ma abbiamo il vantaggio di poterci muovere liberamente fra i nodi.