Appunti Algoritmi e Strutture Dati

Luca Seggiani

26 Marzo 2024

1 Alberi binari

Riprendiamo la trattazione delgli alberi binari. Avevamo detto che un albero è definito come segue:

- L'albero vuoto è un'albero binario.
- Un nodo p più due alberi binari B_s e B_d formano un'albero binario.

Ad esempio, assumiamo che il livello di un albero sia -1. Il livello della radice sarà 0, e il livello dell'albero totale sarà il più lungo cammino fra la radice e una foglia. Un albero binario si dice inoltre etichettatto quando ad ogni nodo è associato un nome, o etichetta.
Una possibile implementazione della struttura dati albero binario in pseudocodice potrebbe essere:

```
struct Node {
  int inf;
  Node* sx;
  Node* dx;
};
```

Gli alberi binari si prestano all'implementazione, su di essi, di algoritmi ricorsivi. Avremo infatti:

- Caso base: albero vuoto
- Passo ricorsivo: radice e i due sottoalberi

Gli algoritmi più comuni su alberi binari sono quelli di: linearizzazione, ricerca, inserimento e cancellazione di nodi. Vediamo la prima categoria:

Linearizzazione di alberi
Vediamo come possiamo linearizzare una struttura dati ad albero. Una linearizzazione di un albero è una sequenza contenente i nomi dei suoi nodi. Le più comuni linearizzazioni, dette **visite**, sono:

- **Ordine anticipato** (*preorder*):
  Si stampa prima di tutto la radice, e poi si chiama ricorsivamente sui sottoalberi sinistri e destro, stampando quindi i nodi appena trovati e poi procedendo di livello nell'albero:

```
void preOrder(Node* albero) {
  if(albero == NULL) return;
  else {
    print(radice);
    preOrder(albero->sx);
    preOrder(albero->dx);
  }
}

```

- **Ordine differito** (*postorder*):
  Si scorre prima ricorsivamente tutto l'albero, raggiungendo l'ultimo nodo a sinistra e stampandolo. Le stampe si susseguono poi attraverso il meccanismo della ricorsione in coda:

```
void postOrder(Node* albero) {
  if(albero == NULL) return;
  else {
    preOrder(albero->sx);
    preOrder(albero->dx);
    print(radice);
  }
}

```

- **Ordine simmetrico** (*inorder*):
  Si stampa la radice dopo la stampa del sottoalbero sinistro e prima della stampa del sottoalbero destro, rispettando quindi quello che sarebbe l'"ordine" naturale dell'albero:

```
void postOrder(Node* albero) {
  if(albero == NULL) return;
  else {
    preOrder(albero->sx);
    print(radice);
    preOrder(albero->dx);
  }
}

```

Alberi binari bilanciati

Un albero binario si dice bilanciato quando ogni suo nodo (tranne quelli

all'ultimo livello) ha due figli. Il numero di nodi di un albero binario bilanciato è $2^{(k+1)} - 1$, e il numero di foglie è 2^k.

Alberi binari quasi bilanciati

Un albero binario è quasi bilanciato quando tutti i livelli fino al penultimo sono bilanciati. Un albero bilanciato è anche quasi bilanciato.

Alberi pienamente binari Un albero sii dice pienamente binario quando tutti i nodi tranne le foglie hanno 2 figli. Il numero di nodi interni è uguale al numero di foglie meno 1.

Complessità della linearizzazione

Studiamo la complessità di una visita fatta su un albero. Abbiamo chiaramente due passi ricorsivi, più un operazione di confronto e di una di lettura che richiedono entrambe tempo $O(1)$. Il problema è posto dal fatto che non conosciamo accuratamente la dimensione dei sottoproblemi al momento della chiamata ricorsivo. Il caso migliore sarebbe quello di sottoalberi bilanciati, ovvero con numero uguale di nodi figli. Possiamo in ogni caso porre la ricorrenza:

$$T(0) = 0, \quad T(n) = b + T(n_s) + T(n_d), \quad n_s + n_d = n - 1, \quad n > 0$$

Il caso particolare bilanciato è il seguente:

$$T(0) = a, \quad T(n) = b + 2T(\frac{n-1}{2})$$

con complessità in caso migliore di $T(n) \in O(n)$. Possiamo inoltre esprimere la complessità in funzione dei livelli, applicando quanto visto prima sugli alberi binari bilanciati:

$$T(0) = a, \quad T(k) = b + 2T(k-1)$$

dove abbiamo complessità $T(k) \in O(2^k)$. Chiaramente la complessità è maggiore, ma possiamo dire che in genere $k < n$, e anche di molto.

Altre funzioni su alberi binari

Poniamoci il problema di contare nodi e foglie di un albero binario. Una possibile soluzione è:

```
//conta nodi
int nodes(Node* tree) {
  if(tree == NULL) return 0;
  return 1 + nodes(tree->sx) + nodes(tree->dx);
}
//conta foglie
int leaves(Node* tree) {
  if(tree == NULL) return 0;
  if(!tree->sx && !tree->dx) return 1; //foglia
  return leaves(tree->sx) + leaves(tree->dx);
}
```

Entrambe le funzioni hanno formula di ricorrenza simile a quelle già viste sulle visite, e quindi complessità $O(n)$.

Ricerca di un'etichetta su un albero binario

Implementiamo una funzione che restituisce un puntatore al primo nodo con etichetta n che trova in visita anticipata. Se il nodo non viene trovato, la funzione restituisce NULL.

```
Node* findNode(int n, Node* tree) {
  if(tree == NULL) return NULL;
  if(tree->inf = n) return tree;
  Node* a = findNode(n, tree->left);
  if(a) return a;
  else return findNode(n, tree->right);
}
```

La complessità è sempre $O(n)$.

Cancellazione dell'albero binario

Vediamo la cancellazione dell'albero, che avremo per esempio bisogno di fare nel caso l'albero fosse allocato in memoria dinamica e si rendesse necessaria la sua deallocazione:

```
void delTree(Node* &tree) {
  if(tree != NULL) {
    delTree(tree->left);
    delTree(tree->right);
    delete tree;
    tree = NULL;
  }
}
```

Inserzione di un nodo su un albero binario

Vediamo una funzione che inserisce un nodo *child* come figlio di un nodo *parent*, sinistro se $c = l$ e destro se $c = r$. Se l'albero è vuoto, inserisce *child* come radice. Se *parent* non esiste o ha già il figlio richiesto, la funzione non modifica l'albero.

```
int insertNode(Node* &tree, int child, int parent, char c) {
  if(tree == NULL) { //albero vuoto
    tree = new Node;
    tree->label = child;
    tree->left = tree->right = NULL;
    return 1;
  }
  Node* a = findNode(parent, tree); //cerca parent
  if(parent == NULL) return 0;
  if(c == 'l' && !a->left) { //figlio sinistro
    a->left = new Node;
    a->left->inf = child;
    a->left->left = a->left->right = NULL;
```

```
14      return 1;
15    }
16    if(c == 'r' && !a->right) { //figlio destro
17      a->right = new Node;
18      a->right->inf = child;
19      a->right->left = a->right->right = NULL;
20      return 1;
21    }
22  }
```

2 Alberi generici

Generalizziamo la definizione di albero binario introducendo l'albero generico. Il concetto chiave sarà sempre quello di una radice, da cui partiranno però non uno ma una sequenza di sottoalberi. Formalmente:

- Un nodo è un albero
- Un nodo più una sequenza di alberi $A_1, ..., A_n$ è un albero.

Notiamo come fra i sottoalberi di un nodo di albero generico non è stabilito alcun ordinamento, mentre nei sottoalberi di alberi binari c'era una chiara distinzione fra sottoalbero sinistro e sottoalbero destro.
La rappresentazione di una struttura dati di tipo albero generico potrebbe essere complicata: l'approccio naive è infatti quello di immagazzinare insieme ad ogni etichetta, una lista dei sottoalberi presenti. Questo è però particolarmente inefficiente, in quanto l'accesso ad ogni sottoalbero richederebbe una scansione lineare della lista dei sottoalberi. Un approccio migliore è quello figlio-fratello: a partire dal nodo radici, definiamo una struttura effettivamente analoga a quella di un albero binario, con la differenza, però, che il puntatore di sinistra indica adesso il prossimo nodo su un livello più basso (figlio), mentre quello di destra indica il prossimo nodo sullo stesso livello (fratello). L'albero binario così generato prende il nome di trasformato dell'albero generico. Una possibile implementazione della struttura dati albero generico sarà quindi:

```
1  struct Node {
2    int inf;
3    Node* child;
4    Node* sibling;
5  };
```

Molte delle funzioni definite su alberi binari valgono (con qualche modifiche) sugli alberi generici, ad esempio la visita anticipata:

```
void preOrder(albero) {
  print(radice);
  preOrder(albero->1);
  ...
  preOrder(albero->n);
}
```

Proprio per la proprietà riportata prima, invece, la visita simmetrica risulta impossbile: non si può stabilire un sottoalbero "centrale" di un nodo. Notiamo, che nella rappresentazione figlio-fratello, la visita anticipata del trasformato corrisponde alla visita trasformata dell'albero generico, e la visita simmetrica del trasformato corrisponde alla visita in differita dell'albero generico. Come per gli alberi binari, la complessità delle operazioni su alberi generici è di $O(n)$ sul numero di elementi.

Altre funzioni su alberi generici

Poniamoci nuovamente il problema di contare nodi e foglie di un albero, stavolta generico. La soluzione è del tutto analoga a quella dell'albero binario, con la sola differenza che un nodo è classificato come foglia quando il suo puntatore child è NULL, e ciò non significa che non possa avere altri sibling, anch'essi foglie.

```
//conta nodi
int nodes(Node* tree) {
  if(tree == NULL) return 0;
  return 1 + nodes(tree->child) + nodes(tree->sibling);
}
//conta foglie
int leaves(Node* tree) {
  if(tree == NULL) return 0;
  if(!tree->child) return 1 + leaves(tree->sibling); //foglia
  return leaves(tree->child) + leaves(tree->sibling);
}
```

Inserzione di un nodo su un albero generico

Riportiamo una funzione che inserisca un nodo come ultimo fratello di un certo sibling:

```
A CASA!
```

O un altra che inserisca un nodo come ultimo figlio di un certo parent:

```
A CASA!
```