

Tracce con esercizi e svolgimento

Traccia: 2025-06-09 - Appello IV - Traccia -Svolgimento

Data la struttura dati **albero n-ario**, scrivere specifica sintattica e semantica di due nuovi operatori (quando possibile/utile, fare uso degli operatori esistenti dell'albero):

- **figli_ordinati**, che prende un nodo u e restituisce la lista dei nodi figli di u , ordinati in senso crescente per valore;

```
figli_ordinati(Nodo, Albero)-> Lista<Nodo>
figli_ordinati(u, T)-> L
PRE: T = <N, A>, u ∈ N
POST: se foglia(u,T), allora L = <>
altrimenti L = <a1, a2, ..., an>, per ogni i, 1 ≤ i ≤ n (ai ∈ N AND PADRE(ai, T) = u)
AND per ogni j, 1 ≤ j ≤ n, leggiNodo(aj, T) ≤ leggiNodo(aj+1, T)      // impone ordinamento
AND per ogni i, 1 ≤ i ≤ n, per ogni j, 1 ≤ j ≤ n, con i ≠ j, ai ≠ aj      // evito duplicati nella lista
AND dato m = |{x ∈ N | padre(x, T) = u}|, n = m                         // garantisco di prendere TUTTI i figli
OPPURE
AND per ogni x ∈ N | padre(x, T) = u, esiste i, 1 ≤ i ≤ n, t.c. x = ai    // garantisco di prendere TUTTI i figli
```

- **pari**, che prende un nodo u e restituisce *true*, se il valore contenuto è pari; *false* altrimenti.

```
pari(Nodo, Albero) -> bool
pari(u, T) -> b
PRE: T = <N, A>, u ∈ N
POST: b = true se leggiNodo(u, T) % 2 = 0; b = false altrimenti.
```

Spiegazione:

Questo è un esercizio dove viene chiesto di estendere una struttura dati esistente (l'albero n-ario) con nuovi operatori, definendoli tramite specifica sintattica (input/output) e specifica semantica (PRE e POST condizioni).

L'esercizio richiede due operatori; **figli_ordinati** , **pari**

Operatore 1: **figli_ordinati**

Dato un nodo u , restituisce una lista contenente tutti i suoi figli, ma ordinati in base al valore (dal più piccolo al più grande).

Specifiche sintattica:

```
figli_ordinati(Nodo, Albero) → Lista<Nodo>
```

- Input: un **Nodo** (quello di cui cerchiamo i figli) e l'**albero** in cui si trova.

- Output: una `Lista<Nodo>` (usiamo la Lista e non l'insieme perchè l'ordine dei risultati è crescente).

Specifico semantica:

PRE-condizione:

`PRE: T = <N, A>, u ∈ N`

- L'albero `T` è composta da `N` Nodi e Archi `A`.
- Il nodo `u` deve esistere nell'insieme dei nodi `N`.

POST-condizione

Dobbiamo distinguere due casi: se il nodo non ha figli (foglia) e se li ha.

- Caso A: Il nodo è una foglia.
`SE foglia(u, T), ALLORA L = <>`
 Se `u` non ha figli, la lista risultante `L` è vuota (indicata con `<>`).
- Caso B: Il nodo ha figli (ALTRIMENTI). Dobbiamo dire che la lista `L` contiene nodi che sono figli di `u`, che sono tutti i figli, e che sono ordinati.
`ALTRIMENTI L = <a1, a2, ..., an>`
 (definiamo `L` come sequenza di n elementi)

Elenchiamo le proprietà che questa sequenza deve rispettare:

1. Proprietà di paternità: Sono davvero figli?

`per ogni i, 1 <= i <= n, (ai ∈ N AND PADRE(ai, T) = u)`

Tutti gli elementi della lista (`ai`) devono essere nodi dell'albero e il loro padre deve essere `u`

2. Proprietà di ordinamento(sono crescenti?):

`per ogni j, 1 <= j < n, leggiNodo(aj, T) <= leggiNodo(a(j+1), T)`

Scorrendo la lista, il valore del nodo corrente (`aj`) deve essere minore o uguale del successivo (`a(j+1)`).

3. Proprietà di unicità (non ci siano duplicati):

`per ogni i, j con i != j, ai != aj`

Ogni figlio appare nella lista una volta sola.

4. Proprietà di completezza (lo ho presi tutti?):

`dato m = |{x ∈ N | padre(x, T) = u}|, n = m`

Qui usiamo la cardinalità `|...|`. Calcoliamo `m`, ovvero il numero totale di nodi nell'albero che sono figli di `u`.

La lunghezza della nostra lista (`n`) deve essere uguale a `m`. Se `n < m`, significherebbe che abbiamo dimenticato qualche figlio.

Operatore 2: `pari`

Prende un nodo `u` e restituisce `true` se il valore contenuto è pari, `false` altrimenti.

Specifiche sintattiche: `pari(Nodo, Albero) → bool`

- Input: un `Nodo u` e l'`Albero T` in cui si trova.
- Output: un valore booleano (`bool`), vero o falso.

Specifiche semantiche: `pari(u, T) → b`

PRE-condizione: `PRE: T = <N, A>, u ∈ N`

- L'albero `T` è composto da Nodi `N` e Archi `A`.
- Il nodo `u` deve esistere nell'insieme dei nodi `N`.

POST-condizione: `POST: b = true se leggiNodo(u,T) % 2 = 0; b = false altrimenti.`

Si legge:

- `leggiNodo(u,T)` : Accediamo al valore intero contenuto nel nodo `u`.
- `% 2 = 0` : L'operatore modulo (resto della divisione) per 2 deve essere uguale a 0 (condizione di parità).
- `b = true ... altrimenti b = false` : La variabile risultato `b` sarà vera se la condizione è soddisfatta, falsa se il numero è dispari.

Es. 2. Descrivere le implementazioni del **grafo** basate su **matrice di adiacenza** e **matrice di adiacenza estesa**. Discutere (enunciare e motivare) la complessità computazionale di ogni singolo operatore del grafo implementato attraverso tali approcci, evidenziandone differenze e vantaggi/svantaggi. **(8 punti)**

Risposta:

L'implementazione del grafo tramite matrice di adiacenza utilizza una matrice quadrata M di dimensione $N \times N$ (dove N è il numero di nodi) in cui l'elemento M_{ij} contiene un valore booleano o un peso se esiste un arco dal nodo i al nodo j assumendo che i nodi siano identificati da indici numerici, mentre la variante estesa affianca alla matrice un vettore di dimensione N per memorizzare le informazioni associate ai nodi mappando così i dati reali sugli indici della matrice.

Analizzando la complessità computazionale, entrambe le soluzioni richiedono uno spazio di memoria pari a $O(N^2)$ il che rappresenta il principale svantaggio per grafi sparsi, ma offrono il vantaggio di verificare l'esistenza di un arco o aggiungerne uno in tempo costante $O(1)$ grazie all'accesso diretto, mentre operazioni come l'individuazione dei nodi adiacenti richiedono $O(N)$ per la scansione della riga e la gestione dinamica dei nodi (inserimento o cancellazione) risulta particolarmente onerosa con un costo di $O(N^2)$ dovendo ridimensionare l'intera struttura.

Es. 3. Si vuole progettare una struttura dati per la memorizzazione di un social network dedicato a ricercatori scientifici. Ogni ricercatore è eventualmente associato ad altri ricercatori, nel caso in cui abbiano collaborato alla scrittura di articoli scientifici. È necessario memorizzare il numero di articoli scientifici per cui hanno collaborato. Per ogni ricercatore, è inoltre necessario memorizzare i riconoscimenti ottenuti a livello internazionale. Completare la specifica della struttura dati **social**, fornendo la specifica semantica per mezzo di PRE e POST condizioni, rispetto alla seguente specifica sintattica:

domini: social, ricercatore, riconoscimento (aggiungerne altri se necessario)

social: insieme dei grafi non orientati $G = \langle N, A \rangle$, con nodi di tipo *ricercatore* e valore dei nodi di tipo

insieme<riconoscimento> e valore degli archi di tipo *intero*

operatori:

// crea il social network, inizialmente senza alcun ricercatore (1p)
creaSocial () → social

creaSocial() -> s

PRE: -

POST: $s = \langle \{ \}, \{ \} \rangle$

// aggiunge un ricercatore al social network (1p)

aggiungiRicercatore (social, ricercatore) → social

aggiungeRicercatore(s, r) -> s'
PRE: s = <N, A>, !esisteNodo(r, s)
POST: s' = scriviNodo(r, insNodo(r,s), {})

// memorizza il fatto che due ricercatori hanno collaborato alla scrittura
di un nuovo articolo scientifico (2p)
aggiungiCoautori (social, ricercatore, ricercatore) ->
social

aggiungiCoautori(s, r1, r2) -> s'
PRE: s = <N, A>, esisteNodo(r1, s), esisteNodo(r2, s)
POST:
SE esisteArco(r1, r2, s), s' = scriviArco(r1, r2, s, leggiArco(r1,r2,s)+1)
ALTRIMENTI s' = scriviArco(r1, r2, insArco(r1,r2,s), 1)
OPPURE
dato x = insArco(r1,r2,s), s' = scriviArco(r1, r2, x, 1)

// aggiungi un riconoscimento ricevuto da un ricercatore (2p)
aggiungiRiconoscimento (social, ricercatore, riconoscimento) -> social

aggiungiRiconoscimento(s,r,p) -> s'
PRE: s = <N, A>, esisteNodo(r, s)
POST: s' = scriviNodo(s,r, leggiNodo(s,r) U {p})

// restituisce il ricercatore che ha scritto il maggior numero di articoli con altri ricercatori; in caso di parimerito, restituire tutti quelli a parimerito (3p)
piuArticoli(social) -> ?? (determinare il tipo dell'output) **(9 punti)**

piuArticoli(social)->insieme<ricercatore>
piuArticoli(s)->I
PRE: s = <N, A>
POST: I = { r ∈ N | non esiste y ∈ N t.c. contaArticoli(s,y) > contaArticoli(s,r)}

contaArticoli(social, ricercatore)->intero
contaArticoli(s,r)->i
PRE: s = <N, A>, r ∈ N
POST: i = sommatoria per ogni [x ∈ N AND esisteArco(r,x,s)] di leggiArco(r,x,s)

Premessa:

- **Nodo:** ricercatore
- **Arco:** collaborazione
- **Peso:** numero di articoli scritti insieme

SPIEGAZIONE dell'esercizio con commenti:

creaSocial: (questo sarà l'operatore costruttore, deve creare un grafo vuoto)

creaSocial() → social (restituisce un grafo senza nodi archi)

PRE: nessuna.

`POST: s = <{}, {}>` cioè il grafo `s` è composto da un insieme di vuoto di nodi e un insieme vuoto di archi.

aggiungiRicercatore : (aggiunge un nuovo ricercatore al social network)

Deve verificare che il ricercatore non esista già, lo deve inserire nel grafo e inizializzi il suo valore (i riconoscimenti) come un insieme vuoto.

`PRE: s = <N, A>, !esisteNodo(r, s)` (il ricercatore `r` non deve già esistere nel grafo `s`)

`POST: s' = scriviNodo(r, insNodo(r, s), {})` (`insNodo(r, s)` aggiunge un nodo al grafo, `scriviNodo(r, insNodo(r, s), {})` associa a quel nodo il valore `{}` (insieme vuoto dei riconoscimenti)).

aggiungiCoautori : (registra una collaborazione tra due ricercatori)

`aggiungiCoautori(social, ricercatore, ricercatore) → social` (controlla che entrambi i ricercatori esistano, controlla se esiste già un arco fra loro (hanno collaborato?), se SI incrementa il numero di collaborazioni (peso +1), se NO crea un nuovo arco con peso 1)

`PRE: s = <N, A>, esisteNodo(r1, s), esisteNodo(r2, s)`

`POST: SE esisteArco(r1, r2, s): s' = scriviArco(r1, r2, s, leggiArco(r1, r2, s) + 1)` (Prende il peso corrente con `leggiArco` e aggiunge 1)

`ALTRIMENTI: s' = scriviArco(r1, r2, insArco(r1, r2, s), 1)` (Inserisce l'arco con `insArco` e imposta il peso a 1)

// C'è anche una variante con `OPPURE` nello svolgimento ufficiale che unisce le operazioni usando una variabile temporanea `x`, ma la logica `SE/ALTRIMENTI` è più chiara (secondo me).

OPPURE

dato x = insArco(r1,r2,s), s' = scriviArco(r1, r2, x, 1)

aggiungiRiconoscimento : (aggiunge un premio all'elenco dei riconoscimenti di un ricercatore)

`aggiungiRiconoscimento(social, ricercatore, riconoscimento) → social` (prende l'insieme attuale dei riconoscimenti, fa l'unione con il nuovo riconoscimento e salva il nuovo insieme nel nodo)

`PRE: s = <N, A>, esisteNodo(r, s)`

`POST: s' = scriviNodo(s, r, leggiNodo(s,r) U {p})`

(`leggiNodo(s,r)` ti dà l'insieme corrente (es. `{Nobel}`). `U {p}` aggiunge il nuovo premio `{Turing}` Risultato: `{Nobel, Turing}`)

piuArticoli : (trova il ricercatore/i che ha collaborato di più in assoluto (somma dei pesi degli archi))

piuArticoli(social) → Insieme<ricercatore> : Restituisce un insieme perchè potrebbe esserci più persone a parimerito con lo stesso massimo.

(per ogni ricercatore, calcola la somma dei pesi di tutti i suoi archi adiacenti, confronta queste somme, e tiene solo i ricercatori per cui non esiste nessuno con una somma maggiore)

PRE: $s = \langle N, A \rangle$

POST: $I = \{ r \in N \mid \text{non esiste } y \in N \text{ t.c. } \text{contaArticoli}(s,y) > \text{contaArticoli}(s,r) \}$

(seleziona i nodi r tali che NON esiste un altro nodo y che ha scritto PIÙ articoli ($>$) di r .

Se non c'è nessuno migliore di te, sei il massimo)

contaArticoli Per rendere leggibile la POST condizione sopra, definiamo contaArticoli(s, r):

POST: $i = \text{sommatoria per ogni } [x \in N \text{ AND } \text{esisteArco}(r,x,s)] \text{ di leggiArco}(r,x,s)$ (somma i pesi (leggiArco) di tutti gli archi che collegano r a un qualsiasi altro nodo x)

Es. 4. Tecniche algoritmiche

(7 punti)

Spiegare il problema dell'allocazione di attività ad una risorsa condivisa e proporre due soluzioni algoritmiche, una basata su tecnica enumerativa e una basata su tecnica greedy, confrontandone la complessità computazionale.

Risposta:

Il problema dell'allocazione di attività consiste nel selezionare il sottoinsieme di cardinalità massima di attività mutuamente compatibili che devono utilizzare una risorsa condivisa, dove ogni attività è definita da un tempo di inizio e uno di fine.

Una soluzione basata sulla tecnica enumerativa esplora l'intero spazio di ricerca generando tutti i possibili 2^n sottoinsiemi di attività e verificando per ciascuno la compatibilità degli orari per scegliere il migliore, ma tale approccio ha una complessità esponenziale $O(2^n)$ che lo rende impraticabile al crescere di n .

Una soluzione basata sulla tecnica greedy, invece, ordina le attività per tempo di fine crescente e seleziona iterativamente la prima attività compatibile con l'ultima scelta fatta; questo approccio è molto più efficiente con una complessità di $O(n \log n)$ dominata dall'ordinamento e garantisce comunque di trovare la soluzione ottima globale a differenza dell'enumerazione che è eccessivamente costosa

Es. 5. Si consideri il seguente algoritmo che lavora sugli array A e B, aventi numero di elementi pari a n . Stimare la complessità dell'algoritmo in funzione di n nel caso pessimo e nel caso ottimo, motivando la risposta e illustrando in quali casi l'algoritmo si trova nelle condizioni ottime e pessime.

(4 punti)

```
void analizza_array(int A[], int B[], int n)
{
    int conta = 0, soglia = 5, i = 0;
    while(i < n && conta <= soglia)
    {
        if(A[i] == B[i])
            conta++;
        i++;
    }
    if(conta >= soglia)
        selection_sort(A);
}
```

```
while(i < n && conta <= soglia) {
    if (A[i] != B[i]) conta++;
    i++;
}
```

Qui abbiamo due condizioni unite da un **AND** (`&&`). Il ciclo si ferma se anche una sola delle due diventa falsa.

- **Scenario A:** Se `conta` non sale mai (o sale poco), la condizione `conta <= soglia` rimane sempre VERA. Quindi il ciclo dipende solo da `i < n`.
 - Deve fare n giri.
 - Costo: $O(n)$.
- **Scenario B (Arrays diversi):** Se `conta` sale, appena arriva a 6 la condizione `conta <= soglia` diventa FALSA.
 - Il ciclo si ferma subito (dopo pochi giri, massimo 6).
 - Costo: $O(1)$ (Costante, perché 6 è un numero piccolo che non dipende da n).

```
if (conta >= soglia)
    selection_sort(A);
```

Quanto costa questo pezzo? Qui c'è un bivio decisivo.

- **Ramo "Non Entro" (`conta < 5`):**
 - L'istruzione è saltata.
 - Costo: **0**.
- **Ramo "Entro" (`conta >= 5`):**
 - Viene lanciato il `selection_sort(A)` .
 - Dalla teoria sappiamo che il Selection Sort usa due cicli annidati per ordinare n elementi.
 - Costo: $O(n^2)$.

3. Perché il Caso Pessimo è $O(N^2)$ e non $O(N^2 + 1)$?

La tua spiegazione è corretta, ma per renderla "a prova di bomba" ricorda la **Regola del Dominio**.

Nel caso pessimo, matematicamente accade questo:

$$Costo_{Totale} = Costo_{While} + Costo_{Sort}$$

$$Costo_{Totale} = 5 \text{ (operazioni costanti)} + N^2 \text{ (operazioni del sort)}$$

Quando N diventa un miliardo:

- 5 è invisibile.
- N^2 è gigantesco. Quindi il 5 (o qualsiasi costante derivata dalla soglia) sparisce. Il "costo" del ciclo diventa irrilevante rispetto alla catastrofe dell'ordinamento.

Riposta da dare:

Caso Pessimo ($O(n^2)$): Si verifica quando il ciclo `while` rileva almeno `soglia` differenze. In questo scenario, la variabile `conta` raggiunge il valore 5, rendendo vera la condizione `if (conta >= soglia)` al termine del ciclo. Di conseguenza, viene eseguito l'algoritmo `selection_sort(A)`, che ha una complessità temporale quadratica $O(n^2)$. Questo costo, rendendo irrilevante il tempo lineare speso nel `while`.

Caso Ottimo ($O(n)$): Si verifica quando gli array sono "quasi uguali", ovvero presentano meno di 5 differenze su tutti gli n elementi. In questo caso, il ciclo `while` deve scorrere l'intero array fino alla fine (`i < n`) poiché la condizione `conta <= soglia` rimane sempre vera. Tuttavia, al termine del ciclo, la condizione `if (conta >= soglia)` risulta falsa, evitando l'esecuzione del `selection_sort`. Il costo totale è quindi determinato solo dalla scansione lineare del ciclo.

Laboratorio

La prova di laboratorio non sarà corretta se non si supera la prova scritta.

Si assuma di avere una classe C++ per l'implementazione di alberi n-ari in sola lettura, che presenti le seguenti funzioni:

```
template< class T >
class ReadOnlyTree {
public:
    typedef int Nodo;

    ReadOnlyTree();
    bool vuoto() const;
    Nodo radice() const;
    Nodo padre(Nodo) const;
    Nodo primofiglio(Nodo) const;
    Nodo succfratello(Nodo) const;
    bool ultimofratello(Nodo) const;
    bool foglia(Nodo) const;
    leggi(Nodo) const;
    void scrivi(Nodo, const T &) const;
}
```

Scrivere la funzione *confronta_alberi* (...) che prenda in input due alberi n-ari di interi P e Q , e un valore intero k , e restituisca una lista contenente i nodi di P il cui valore è presente in almeno k nodi dell'albero Q .

```
#include <list>
#include <iostream>

// Assumiamo che la classe sia già definita come nel testo
// template<class T> class ReadOnlyTree { ... };

// --- FUNZIONE AUSILIARIA: Conta occorrenze di 'target' in T ---
// Visita tutto l'albero T per contare quante volte appare 'target'
int conta_occorrenze(const ReadOnlyTree<int>& T, ReadOnlyTree<int>::Nodo u, int target) {
    int count = 0;

    // 1. Processa il nodo corrente
    if (T.leggi(u) == target) {
        count = 1;
    }

    // 2. Processa ricorsivamente i figli (se non è foglia)
    if (!T.foglia(u)) {
        ReadOnlyTree<int>::Nodo curr = T.primofiglio(u);
        while (true) {
            count += conta_occorrenze(T, curr, target);

            // Controllo per il ciclo sui fratelli
            if (T.ultimofratello(curr)) break;
            curr = T.succfratello(curr);
        }
    }
}
```

```

    }

    return count;
}

// --- FUNZIONE AUSILIARIA: Visita P e costruisce la lista ---
void scansiona_P(const ReadOnlyTree<int>& P, ReadOnlyTree<int>::Nodo u,
                 const ReadOnlyTree<int>& Q, int k,
                 std::list<int>& risultato) {

    // A. Leggi il valore del nodo corrente in P
    int valore_P = P.leggi(u);

    // B. Conta quante volte questo valore appare in Q
    // (Solo se Q non è vuoto, partiamo dalla radice di Q)
    int num_occorrenze = 0;
    if (!Q.vuoto()) {
        num_occorrenze = conta_occorrenze(Q, Q.radice(), valore_P);
    }

    // C. Se soddisfa la condizione, aggiungi il NODO alla lista
    if (num_occorrenze >= k) {
        risultato.push_back(u);
    }

    // D. Continua la visita in profondità su P (figli e fratelli)
    if (!P.foglia(u)) {
        ReadOnlyTree<int>::Nodo curr = P.primofiglio(u);
        while (true) {
            scansiona_P(P, curr, Q, k, risultato);

            if (P.ultimofratello(curr)) break;
            curr = P.succfratello(curr);
        }
    }
}

// --- FUNZIONE PRINCIPALE RICHIESTA ---
std::list<int> confronta_alberi(const ReadOnlyTree<int>& P, const ReadOnlyTree<int>& Q, int k) {
    std::list<int> lista_nodi;

    // Se P è vuoto, non c'è nulla da controllare
    if (P.vuoto()) {
        return lista_nodi;
    }

    // Avvia la ricorsione partendo dalla radice di P
}

```

```
scansiona_P(P, P.radice(), Q, k, lista_nodi);
```

```
    return lista_nodi;
```

```
}
```