Corso di Laurea in Informatica – Algoritmi e Strutture Dati – Corso M-Z – 2024/2025 II Appello – 07/02/2025, 9.00-11.00

Consegnare un unico file PDF chiamato COGNOME_NOME.pdf

Es. 1 Estensione di una struttura dati esistente, con nuovi operatori (5 punti)

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):

- **zii,** che prende un nodo *u* e restituisce l'insieme dei nodi fratelli del padre di *u*;

zii(Nodo, Albero)->Insieme<Nodo>

zii(u, T) -> I
PRE: T = <N, A>, u ∈ N
POST: SE livello(u) < 2, I = {}, ALTRIMENTI
I = { x ∈ N | PADRE(x,T) = PADRE(PADRE(u,T),T)} – {PADRE(u,T)}

- **min_a_radice_k,** che scrive nella radice il valore più piccolo contenuto nei nodi a livello k, se il livello k esiste.

min_a_radice_k(Intero, Albero)->Albero
min_a_radice_k(k, T) -> T'
PRE: T = <N, A>, N != {}, esiste u ∈ N, t.c. livello(u) = k
POST: dato y = leggiNodo(x,T) t.c.
x ∈ N AND livello (x) = k
AND non esiste i ∈ N, con livello(i) = k t.c. leggiNodo(i,T) < leggiNodo(x,T)
T' = scriviNodo(T, RADICE(T), y)

Es. 2. Descrivere l'implementazione del **grafo** basata su **matrice di adiacenza**. Discutere (enunciare e motivare) la complessità computazionale di ogni singolo operatore del grafo implementato attraverso tale approccio. **(8 punti)**

Es. 3. Si vuole progettare una struttura dati per la memorizzazione dello storico dei pagamenti dei dipendenti di una azienda. In particolare, per ogni dipendente si memorizza lo storico degli ultimi 24 stipendi mensili ricevuti, ciascuno dei quali rappresentati da un float. Completare la specifica della struttura dati **storico**, fornendo la specifica semantica per mezzo di PRE e POST condizioni, rispetto alla seguente specifica sintattica:

domini: storico, dipendente, intero, float (aggiungerne altri se necessario)
storico: insieme di dizionari con chiave dipendente e valore coda di float

operatori:
// crea la struttura per memorizzare lo storico dei pagamenti (1p)
creaStorico () → storico

creaStorico() -> s
PRE: -
POST: s = { }

// aggiunge un dipendente allo storico (1p)
aggiungiDipendente (storico, dipendente) → storico

aggiungiDipendente(S,d) -> S'
PRE: non esiste <x,v> ∈ S t.c. x = d *OPPURE* ! APPARTIENE(d,S)
POST: S' = S U {<d,<>>} *OPPURE* S' = S U {<d,CREACODA()>} *OPPURE* S' = INSERISCI(S, <d, <>>)

// aggiungi un pagamento ricevuto da un dipendente, mantenendo uno storico di massimo 24 pagamenti (2p)
aggiungiPagamento (storico, dipendente, float) -> storico

aggiungiPagamento(S, d, pag) -> S'
PRE: APPARTIENE(d,S)
POST: dato v = RECUPERA(d,S) = <a1, a2, …., an>
SE n < 24, v' = < pag, a1, a2, …., an>, ALTRIMENTI v' = <pag, a1, a2, …., a(n-1)>
S' = INSERISCI(d,v')

// restituisce la somma degli ultimi k (terzo argomento) pagamenti ricevuti dal dipendente (2p)
totaleUltimiKPagamenti (storico, dipendente, intero) -> float
totaleUltimiKPagamenti(S, d, k) -> somma
PRE: APPARTIENE(d,S)
POST: dato v = RECUPERA(d,S) = <a1, a2, …., an>
somma = sommatoria per i che va da 1 a min(k,n) di a(i)

// restituisce il dipendente (o i dipendenti, in caso di parimerito) che ha ricevuto la somma maggiore complessiva negli ultimi 12 pagamenti (3p)
piùPagati(storico) -> insieme<dipendente> **(9 punti)**
piùPagati(S)->I
PRE: -
POST: SE S = {}, I = {} ALTRIMENTI
I = {d | APPARTIENE(d,S) AND per ogni d' t.c. APPARTIENE(d',S),
totaleUltimiKPagamenti(S,d',12) <= totaleUltimiKPagamenti(S,d,12)}
OPPURE
I = {d | APPARTIENE(d,S) AND non esiste d' t.c. APPARTIENE(d',S),
totaleUltimiKPagamenti(S,d',12) > totaleUltimiKPagamenti(S,d,12)}

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.

Es. 5. Si consideri il seguente algoritmo che lavora sugli array A (non ordinato) e B (ordinato), 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)
  {
     int elemA = A[i];
     int indexB = ricerca_binaria(B, elemA); // restituisce l'indice dell'elemento (-1 se non trovato)
```

```
if(indexB == -1)
        conta++;
i++;
  }
  if(conta > soglia)
    selection_sort(A);

}
```

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 *lista_speciale (...)* che prenda in input due alberi n-ari di interi X e Y e restituisca una lista contenente i nodi dell'albero X aventi almeno un figlio contenente un valore presente almeno una volta tra i valori dei nodi dell'albero Y. Formalmente:

lista_speciale(Albero, Albero) =>
lista lista_speciale(X, Y) => L
PRE: X = <N, A>, Y = <N', A'>
POST: L è una lista ottenuta prendendo una sola volta tutti gli elementi dell'insieme

$$\{x \in N \mid \exists x' \in N\, t.c.\ padre(x', X) = x\ AND\ \exists y \in N\, t.c.\ leggiNodo(x', X) = leggiNodo(y, Y)\}$$

```
void visita(const ReadOnlyTree<int> &X, const ReadOnlyTree<int> &Y,
const ReadOnlyTree<int>::Nodo &u, Lista<ReadOnlyTree<int>::Nodo &lista)
{
        // ESAMINA u
if(soddisfa(X, Y, u))
              lista.insLista(u, lista.primoLista());

// PARTE STANDARD DELLA VISITA
        if(!X.foglia(u))
        {
              ReadOnlyTree<int>::Nodo f = X.primofiglio(u);
              while(!X.ultimofratello(f))
              {
                    visita(X, Y, f, lista);
```

```
                f = X.succfratello(f);
            }
      visita(X, Y, f, lista);
}
}


bool soddisfa(const ReadOnlyTree<int> &X, const ReadOnlyTree<int> &Y,
const ReadOnlyTree<int>::Nodo &u)
{
      if(X.foglia(u))
            return false;

      ReadOnlyTree<int>::Nodo f = X.primofiglio(u);
bool trovato = false;
      while(!X.ultimofratello(f))
      {
            trovato = trova(Y, Y.radice(), X.leggiNodo(f));
            if(trovato)
                  return true;
            f = X.succfratello(f);
      }
trovato = trova(Y,Y.radice(), X.leggiNodo(f));
return trovato;
}

bool trova(const ReadOnlyTree<int> &Y, const ReadOnlyTree<int>::Nodo &u,
int daTrovare)
{
      // ESAMINA u
if(Y.leggiNodo(u) == daTrovare)
             return true;

// PARTE (SEMI-)STANDARD DELLA VISITA
if(!Y.foglia(u))
      {
            bool res = false;
      ReadOnlyTree<int>::Nodo f = Y.primofiglio(u);
            while(!Y.ultimofratello(f))
            {
                  res = trova(Y, f, daTrovare);
                  if(res == true)
                        return true;
                  f = Y.succfratello(f);
            }
            res = trova(Y, f, daTrovare);
      return res;
}
}


Lista<ReadOnlyTree<int>::Nodo> listaSpeciale(const ReadOnlyTree<int> &X, const
ReadOnlyTree<int> &Y)
{
      Lista<ReadOnlyTree<int>::Nodo> lista;
      visita(X, Y, X.radice(), lista);
      return lista;
}
```