Soluzioni Esercitazione II ASD 2024-25

Lorenzo Cazzaro   Riccardo Maso   Alessandra Raffaetà

Esercizio 1. Sia `t` un albero binario di ricerca (BST) avente **n** nodi con i seguenti campi: `key`, `p`, `left` e `right`.

1. Si scriva una funzione efficiente

   `PTree Modify_key(PTree t, PNode x, int key)`

   che modifica `x->key` con `key` solo se `t` è ancora un `BST` e ritorna `t` se `t` è ancora un `BST`, `nullptr` altrimenti.

2. Valutare e giustificare la complessità della funzione.

I tipi PNode e PTree sono definiti nel modo seguente::

```
struct Node {
       int key;
       Node* p;
       Node* left;
       Node* right;
};

typedef Node* PNode;

struct Tree {
       PNode root;
 };

typedef Tree* PTree;
```

Soluzione:

```
//Funzione che restituisce il minimo del BST radicato nel nodo x.
PNode tree_minimum(PNode x){
    while (x->left)
        x = x->left;
    return x;
}

//Funzione che restituisce il successore del nodo x.
PNode tree_succ(PNode x){
```

```
    if (x->right)
        return tree_minimum(x->right);
    PNode y = x->p;
    while (y && x == y->right){
        x = y;
        y = y->p;
    }
    return y;
}

//Funzione che restituisce il massimo del BST radicato nel nodo x.
PNode tree_maximum(PNode x){
    while (x->right)
        x = x->right;
    return x;
}

//Funzione che restituisce il predecessore del nodo x.
PNode tree_pred(PNode x){
    if (x->left)
        return tree_maximum(x->left);
    PNode y = x->p;
    while (y && x == y->left){
        x = y;
        y = y->p;
    }
    return y;
}

//Funzione che modifica la chiave del nodo x se le condizioni specificate nella
     consegna sono soddisfatte.
PTree modify_key(PTree t, PNode x, int key){

    PNode pred, succ;

    if (x->key == key)
        return t;

    succ = tree_succ(x);
    pred = tree_pred(x);
    if ((succ && succ->key < key) || (pred && pred->key > key))
        return nullptr;
    x->key = key;
    return t;
}
```

La complessità temporale asintotica di `modify_key` dipende dalle complessità delle funzioni `tree_succ` e `tree_pred` utilizzate, che a loro volta dipendono da `tree_maximum` e `tree_minimum`.

In particolare, le complessità temporali di queste funzioni ausialiare sono, come visto a lezione (indichiamo con h l'altezza dell'albero):

- `tree_succ`: $\mathcal{O}(h)$;

- `tree_pred`: $\mathcal{O}(h)$;
- `tree_maximum`: $\mathcal{O}(h)$;
- `tree_minimum`: $\mathcal{O}(h)$.

Dato che all'interno della funzione `modify_key` vengono eseguite operazioni dal costo costante oltre alle due chiamate a funzione, la complessità temporale risultante è $\mathcal{O}(h)$.

Esercizio 2. Sia **arr** un vettore di interi di dimensione **n**. Si assuma che nel vettore **arr** siano stati inseriti numeri interi positivi provenienti da un albero binario di ricerca completo **T**. In particolare, gli elementi di **T** sono stati inseriti in **arr** usando la stessa convenzione che si usa normalmente per la memorizzazione di uno **heap** in un vettore.

1. Scrivere una procedura `stampa` che stampa le chiavi di **T** in ordine non decrescente.

   ```
   void stampa(const vector<int>& arr)
   ```

2. Scrivere una funzione efficiente

   ```
   int maxBST(const vector<int>& arr)
   ```

   che restituisce il massimo di **arr**.

3. Scrivere una funzione efficiente

   ```
   int minBST(const vector<int>& arr)
   ```

   che restituisce il minimo di **arr**.

4. Siano **arr1** ed **arr2** due vettori che memorizzano due alberi binari di ricerca completi **T1** e **T2** aventi ciascuno **n** elementi, con le convenzioni sopra fissate. Sia **k** una chiave intera tale che tutte le chiavi di **T1** sono minori di **k** e tutte le chiavi di **T2** sono maggiori di **k**. Si vuole costruire l'albero binario di ricerca completo **T** che si otterrebbe dalla fusione di **T1**, **k**, e **T2**. Scrivere una funzione efficiente che restituisce l'array che memorizza **T**. La funzione ha il seguente prototipo:

   ```
   vector mergeBST(vector<int>& arr1, vector<int>& arr2, int val)
   ```

5. Determinare e giustificare la complessità delle funzioni `stampa`, `maxBST`, `minBST` e `mergeBST`.

Soluzione:

```
void stampaAux(const vector<int>& arr, size_t i, size_t n){
    if (i < n){
        stampaAux(arr, i*2+1, n);
        cout<< arr[i]<< endl;
        stampaAux(arr, i*2 +2, n);
    }
}

//Procedura che stampa l'albero.
void stampa(const vector<int>& arr){
    stampaAux(arr, 0, arr.size());
}

//Funzione che restituisce il massimo dell'albero.
int maxBST(const vector<int>& arr){
    return arr[arr.size()-1];
}

//Funzione che restituisce il minimo dell'albero.
int minBST(const vector<int>& arr){
    size_t dim = arr.size();
    size_t i = 0;
    while (i*2+1 < dim)
        i = i*2 + 1;

    return arr[i];
}

//Funzione che effettua l'unione di due alberi rappresentati da arr1 e arr2.
vector<int> mergeBST(const vector<int>& arr1, const vector<int>& arr2, int val)
    {
    vector<int> ris;
    ris.push_back(val);
    size_t sx = 0, j = 1, dx = 0;
    while (sx < arr1.size()){
        size_t end = sx+j;
        while (sx < end){
            ris.push_back(arr1[sx]);
            sx++;
        }
        while (dx < end){
            ris.push_back(arr2[dx]);
            dx++;
        }
        j*=2;
    }
    return ris;
}
```

Le complessità delle funzioni e procedure implementate sono le seguenti:

- `stampaAux`: $\Theta(n)$, dato che ciascun elemento dell'heap viene visitato una e una sola volta.

- `stampa`: $\Theta(n)$, dato che viene solo chiamata la procedura `stampaAux`.
- `maxBST`: $\Theta(1)$, dato che viene solamente estratto l'ultimo elemento dell'array `arr` rappresentante l'albero binario di ricerca.
- `minBST`: $\Theta(log(n))$, dato che il ciclo while itera per $log(n)$ volte per raggiungere il minimo elemento dell'albero binario di ricerca che si suppone essere completo. Il minimo elemento è contenuto nella foglia del ramo più a sinistra dell'albero.
- `mergeBST`: $\Theta(n)$, dato che vengono scorsi e inseriti in `ris` tutti gli elementi di `arr1` e `arr2`, che si suppone rappresentino due alberi di **n** nodi.

Esercizio 3. Questa settimana Carlotta ha ricevuto del denaro dai suoi genitori e vuole spenderlo tutto acquistando libri. Per finire un libro Carlotta impiega una settimana e poiché riceve denaro ogni due settimane, ha deciso di acquistare due libri, così potrà leggerli fino a quando riceverà altri soldi. Desidera spendere tutti i soldi così vorrebbe scegliere due libri i cui prezzi sommati sono pari ai soldi che ha ricevuto.

Data la quantità di soldi che Carlotta ha a disposizione e un array contenente i prezzi dei libri (tutti distinti), restituire le coppie di prezzi di libri che soddisfano la condizione. Le coppie di prezzi devono contenere prima il prezzo più basso e poi quello più alto.

Scrivere una funzione efficiente il cui prototipo è il seguente:

```
int libriSelezionati(vector<int>& prezzolibri, double soldi,
                  vector<pair<int,int>>& ris)
```

La funzione restituisce la dimensione dell'array `ris`. Si devono scrivere eventuali funzioni/procedure ausiliari utilizzate.

Soluzione:

```
void my_merge(vector<int>& v, int inf, int med, int sup){

    vector<int> aux;
    int primo = inf, secondo = med +1;

  while (primo <= med && secondo <= sup)
    if (v[primo] > v[secondo]){
      aux.push_back(v[secondo]);
      secondo ++;
    }
    else {
      aux.push_back(v[primo]);
      primo ++;
    }

  if (primo <= med) {
    for (int i = med; i >= primo; i--){
      v[sup] = v[i];
```

```
      sup --;
    }
  }
  for (int i = 0; i < aux.size(); i++)
    v[i + inf] = aux[i];
}

//Implementazione di mergesort
void my_mergesort(vector<int>& v, int inf, int sup){
    int med;

    if (inf < sup){
        med = (inf + sup)/2;
        my_mergesort(v, inf, med);
        my_mergesort(v, med + 1, sup);
        my_merge(v, inf, med, sup);
    }
}

//Funzione che restituisce il numero di coppie di libri che possono essere
    comprate spendendo tutti i soldi.
int libriSelezionati(vector<int>& prezzolibri, double soldi, vector<pair<int,
    int>>& ris){
    int i, j;
    my_mergesort(prezzolibri, 0, prezzolibri.size() - 1);
    i = 0;
    j = prezzolibri.size()-1;
    while (i < j)
        if (prezzolibri[i] + prezzolibri[j] == soldi){
            ris.push_back({prezzolibri[i], prezzolibri[j]});
            i++;
            j--;
        }
        else {
            if (prezzolibri[i] + prezzolibri[j] < soldi)
                i++;
            else
                j--;
        }
    return ris.size();
}
```

La complessità temporale asintotica di `libriSelezionati` dipende dalla complessità della funzione di ordinamento e del ciclo `while` utilizzato. Il ciclo `while` utilizzato per individuare le coppie di prezzi di libri inserite in `ris` presenta complessità $\Theta(n)$, dove n è il numero di elementi del vettore `prezzolibri` in input. La funzione di ordinamento presenta complessità $\Theta(n \cdot logn)$, se la funzione di ordinamento implementata è una di quelle viste a lezione con complessità ottima. Ad esempio, nella soluzione proposta viene implementato l'algoritmo di ordinamento `Mergesort` tramite la procedura `my_mergesort`. Quindi, la complessità di `libriSelezionati` è $\Theta(n) + \Theta(n \cdot logn) = \Theta(n \cdot logn)$.

Esercizio 4. Progettare una realizzazione del tipo di dato Albero binario di ricerca. Si assuma che ciascun nodo `x`, anziché includere l'attributo `x->p`, che punta al padre di `x`, includa `x->succ`, che punta al successore di `x`. Verrà resa disponibile l'interfaccia della classe da completare assieme all'esercitazione su HackerRank. Discutere la complessità in tempo di ciascuna operazione.

Soluzione:

```
#include "tree.hpp"

struct Node{
  int key;
  Node* succ;
  Node* left;
  Node* right;
};

/*post: crea un nodo contenente chiave uguale a val, con tutti i rimanenti
    campi a nullptr */
PNode createNode(int val) {
    return new Node{val, nullptr, nullptr, nullptr};
}

/*post: restituisce un albero vuoto */
Tree::Tree(){
  root = nullptr;
}

void cancella(PNode u){
    if (u){
        cancella(u->left);
        cancella(u->right);
        delete u;
    }
}

/*post: distrugge l'albero */
Tree::~Tree(){
    if (root)
        cancella(root);
}

/*post: restituisce true se l'albero e' vuoto, false altrimenti */
bool Tree::is_empty() const{
  return root==nullptr;
}

/*post: restituisce la radice dell'albero*/
PNode Tree::radice() const{
  return root;
}
```

```
/*pre: u<> nullptr */
/*post: restituisce l'informazione contenuta nel nodo u*/
int Tree::leggiInfo(PNode u) const{
   return u->key;
}

/*pre: u<> nullptr */
/*post: restituisce il figlio sinistro del nodo u */
PNode Tree::figlioSx(PNode u) const{
  return u->left;
}

/*pre: u<> nullptr */
/*post: restituisce il figlio destro del nodo u */
PNode Tree::figlioDx(PNode u) const{
  return u->right;
}

/*pre: u<> nullptr */
/*post: restituisce il puntatore al padre di u*/
PNode Tree::padre(PNode u) const{
  PNode y, z, iter;

  y = u;
  while (y->right != nullptr)
    y = y->right;

  z = y->succ;

  if (z == nullptr)
    iter = root;
  else
    iter = z->left;

  if (iter == u)
    return z;

  while (iter -> right != u)
    iter = iter->right;

  return iter;
}

/*pre: u <> nullptr */
/*post: restituisce il successore di u nell'ordine stabilito in una visita
    simmetrica dell'albero a cui u appartiene. Se u e' il massimo restituisce
    nullptr*/
PNode Tree::treesucc(PNode u) const {
  return u->succ;
}
```

```
/*pre: u <> nullptr */
/*post: restituisce il predecessore di u nell'ordine stabilito in una visita
    simmetrica dell'albero a cui u appartiene. Se u e' il minimo restituisce
    nullptr*/
PNode Tree::treepred(PNode u) const{
  PNode y;

  if (u->left != nullptr)
    return treemax(u->left);

  y = padre(u);
  while (y!= nullptr && y->left == u){
    u = y;
    y = padre(y);
  }
  return y;
}

/*pre: u <> nullptr */
/*post: restituisce il minimo del sottoalbero radicato nel nodo u */
PNode Tree::treemin(PNode u) const{
  while (u->left != nullptr)
    u = u->left;

  return u;
}

/*pre: u <> nullptr */
/*post: restituisce il massimo del sottoalbero radicato nel nodo u */
PNode Tree::treemax(PNode u) const{
  while (u->right != nullptr)
    u = u->right;

  return u;
}

/*post: restituisce un nodo con chiave k nel sottoalbero radicato in u, se
    esiste, altrimenti restituisce nullptr */
PNode Tree::treesearchaux(PNode u, int k) const{
  if (u== nullptr || u->key == k)
    return u;

  if (k < u->key)
    return treesearchaux(u->left, k);

  return treesearchaux(u->right, k);
}

/*post: restituisce un nodo con chiave k, se esiste, altrimenti restituisce
    nullptr */
PNode Tree::treesearch(int k) const {
    return treesearchaux(root, k);
}
```

```
/*post: inserisce il nodo z nell'albero t.
z e' un nodo con left, right e succ con valore nullptr*/
void Tree::treeinsert(PNode z){
   PNode iter, y, ant;

   iter = root;
   y = nullptr;    /* padre di iter */
   ant = nullptr;

   while (iter != nullptr) {
      y = iter;
      if (z->key < iter -> key)
         iter = iter->left;
      else {
         iter = iter->right;
         ant = y;
      }
   }
   if (y == nullptr)
      root = z;
   else
      if (z->key < y->key)
         y->left = z;
      else
         y->right = z;

   if (ant != nullptr){
      z->succ = ant->succ;
      ant->succ = z;
   }
   else
      z->succ = y;
}

void Tree::transplant(PNode u, PNode v){
  PNode p;

  p = padre(u);

  if (p == nullptr)
    root = v;
  else
    if (u ==  p ->left)
      p->left = v;
     else
      p->right = v;
}


/*pre: z <> nullptr */
/*post: rimuove dall'albero il nodo z */
```

```
void Tree::treedelete(PNode z){
  PNode prec, succ;

  prec = treepred(z);
  succ = z->succ;
  if (z->left == nullptr)
    transplant(z, z->right);
  else
    if (z->right == nullptr)
      transplant(z, z->left);
    else {
      if (succ != z->right){
        transplant(succ, succ->right);
        succ->right = z->right;
      }
      transplant(z, succ);
      succ->left = z->left;
    }

  if (prec != nullptr)
    prec->succ = succ;
  delete z;
}
```

Andiamo ora ad anallizarre la complessità delle varie funzioni:

- Le funzioni `createNode`, `is_empty`, `radice`, `leggiInfo`, `figlioSx`, `figlioDx`, `treesucc` ed il costruttore `Tree` hanno complessità costante $\Theta(1)$.
- La procedura `cancella` ha complessità $\Theta(n)$ con n numero dei nodi dell'albero radicato in `u` in quanto effettua una visita dell'albero e ogni nodo è attraversato una sola volta. Il distruttore $\sim$ `Tree` invoca la procedura `cancella` sulla radice dell'albero. Di conseguenza ha complessità $\Theta(n)$, con n numero dei nodi dell'albero.
- Le funzioni `treemin` e `treemax` hanno complessità $\mathcal{O}(h)$ dove h è l'altezza dell'albero in quanto i nodi attraversati dalle funzioni formano un cammino dal nodo `u` verso le foglie.
- La funzione `treesearchaux` ha complessità $\mathcal{O}(h)$ dove h è l'altezza dell'albero in quanto i nodi attraversati dalla funzione formano un cammino dal nodo `u` verso le foglie. La funzione `treesearch` invoca la funzione `treesearchaux` sulla radice dell'albero. Di conseguenza ha complessità $\mathcal{O}(h)$, con h altezza dell'albero.
- La funzione `padre` esegue una serie di operazioni costanti e due cicli `while`. In ciascuno dei due cicli a ogni iterazione si scende di un livello dell'albero. Di conseguenza la funzione ha complessità $\mathcal{O}(h)$, con h altezza dell'albero.
- La funzione `treepred` invoca la funzione `treemax` o la funzione `padre` poi seguita da un ciclo `while` che viene eseguito $\mathcal{O}(h)$ volte in quanto a ogni iterazione si sale di un livello dell'albero verso la radice. Il corpo del `while` invoca la funzione `padre` dunque ha un costo $\mathcal{O}(h)$. La complessità nel caso peggiore è dunque $\mathcal{O}(h) + \mathcal{O}(h) \cdot \mathcal{O}(h) = \mathcal{O}(h^2)$.

- La procedura `treeinsert` ha complessità $\mathcal{O}(h)$ dove h è l'altezza dell'albero in quanto i nodi attraversati dalla procedura formano un cammino dalla radice verso le foglie.

- La procedura `transplant` invoca la funzione `padre` e tutte le altre operazioni hanno costo costante. Quindi ha complessità $\mathcal{O}(h)$, con h altezza dell'albero.

- La procedura `treedelete` invoca la funzione `treepred` e nel caso peggiore invoca due volte la procedura `transplant`. La complessità è dunque $\mathcal{O}(h^2)+2\cdot\mathcal{O}(h) = \mathcal{O}(h^2)$, con h altezza dell'albero.

Esercizio 5. Dato un albero di ricerca T, scrivere una funzione efficiente che restituisca il numero di elementi che occorrono una sola volta e analizzarne la complessità.

Il prototipo della funzione è

`int elementiDistinti(const Tree& t))`

N.B. Non si possono usare strutture ausiliarie di dimensione $\mathcal{O}(n)$ dove n è il numero dei nodi dell'albero e si devono utilizzare le funzioni della classe Tree.

Soluzione:

```
int elementiDistinti(const Tree& t){
    PNode iter;
    int ris;
    if (t.is_empty())
        return 0;
    ris = 0;
    iter = t.treemin(t.radice());
    while (iter){
        PNode succ = t.treesucc(iter);
        int count = 1;
        while (succ && t.leggiInfo(iter) == t.leggiInfo(succ)){
            count++;
            iter = succ;
            succ = t.treesucc(iter);
        }
        if (count == 1)
            ris++;
        iter = succ;
    }
    return ris;
}
```

Il costo della funzione `treemin` è $\mathcal{O}(h)$ dove h è l'altezza dell'albero mentre tutte le altre funzioni hanno costo $\Theta(1)$. I due cicli `while` annidati visitano ogni nodo dell'albero una sola volta. Di conseguenza la complessità della funzione `elementiDistinti` è $\mathcal{O}(h) + \Theta(n) = \Theta(n)$.

Esercizio 6. Sia A un vettore di n interi e si consideri il problema di determinare se esistono due interi che occorrono in A lo stesso numero di volte.

1. Si scriva un algoritmo efficiente per il problema proposto. Il prototipo della funzione è:

   `bool stesseOccorrenze(vector<int>& arr)`

2. Si scriva un algoritmo efficiente per il problema proposto nel caso in cui in A occorrono c valori distinti, dove c è una costante intera positiva (non è nota). Il prototipo della funzione è:

   `bool stesseOccorrenzeValoriDistintiCostanti(vector<int>& arr)`

3. Si determini e giustifichi la complessità degli algoritmi proposti.

Si devono scrivere eventuali funzioni/procedure ausiliarie utilizzate.

Soluzione:

```
void mymerge(vector<int>& arr, int p, int med, int r){
  vector<int> aux;
  int i = p, j = med +1;

  while (i <= med && j <= r)
    if (arr[i] <= arr[j]){
      aux.push_back(arr[i]);
      ++i;
    }
    else {
      aux.push_back(arr[j]);
      ++j;
    }

  if (i <= med){
    for (j = med; j>= i; --j){
      arr[r] = arr[j];
      --r;
    }
  }
  for (i = 0; i < aux.size(); ++i)
    arr[p + i] = aux[i];
}


void mymergesort(vector<int>& arr, int p, int r){
  if (p < r){
    int med = (p + r)/2;
    mymergesort(arr, p, med);
    mymergesort(arr, med + 1, r);
    mymerge(arr, p, med, r);
```

```
  }
}

bool stesseOccorrenze(vector<int>& arr){
    int i, j;
    bool ok = false;
    vector<int> occ(arr.size() + 1);
    if (arr.size() < 2)
        return false;
    mymergesort(arr, 0, arr.size() - 1);
    i = 0;
    while (i < arr.size() && !ok){
        j = i+1;
        while (j < arr.size() && arr[j] == arr[i])
            j++;
        if (occ[j-i] == 0){
            occ[j-i] = 1;
            i = j;
        }
        else
            ok = true;
    }
    return ok;
}

void insertionsort(vector<pair<int, int>>& mappa){
    for (int i = 1; i < mappa.size(); i++){
        pair<int,int> val = mappa[i];
        int j = i - 1;
        while (j >= 0 && mappa[j].second > val.second){
            mappa[j + 1] = mappa[j];
            j--;
        }
        mappa[j + 1] = val;
    }
}

bool stesseOccorrenzeValoriDistintiCostanti(vector<int>& arr){
    int j;
    bool ok;
    vector<pair<int, int>> mappa;
    if (arr.size() < 2)
        return false;

    for (int i=0; i<arr.size();i++){
        j = 0;
        while (j < mappa.size() && mappa[j].first!= arr[i])
            j++;
        if (j == mappa.size())
            mappa.push_back(make_pair(arr[i],1));
        else
            mappa[j].second += 1;
    }
```

```
    insertionsort(mappa);
    ok = false;
    j = 0;
    while (j < mappa.size()-1 && !ok){
        if (mappa[j].second == mappa[j + 1].second)
            ok = true;
        else
            j++;
    }

    return ok;
}
```

Andiamo ad analizzare le complessità delle due soluzioni:

- Nella prima soluzione viene eseguito l'ordinamento *mergesort* che costa $\Theta(n \log n)$ con n numero di elementi dell'array `arr`. Poi viene eseguito il ciclo `while` che scorre ogni elemento dell'array `arr` al più una volta. Dunque la complessità è $\Theta(n \log n) + \mathcal{O}(n) = \Theta(n \log n)$.

- Nella seconda soluzione, si esegue un ciclo `for` che ha un ciclo `while` annidato allo scopo di costruire una mappa che associa a ciascuno dei c valori distinti quante volte occorrono nell'array `arr`. Il costo di questi cicli annidati è $\mathcal{O}(nc)$, in quanto per ogni elemento dell'array `arr` si può scorrere l'intera mappa che contiene al più c elementi. Viene poi invocato l'*insertion sort* sulla mappa, che ha complessità $\mathcal{O}(c^2)$. Infine attraverso l'ultimo ciclo `while` si effettua una ricerca nella mappa di due valori con lo stesso numero di occorrenze, operazione che costa $\mathcal{O}(c)$. Di conseguenza la complessità della seconda soluzione è $\mathcal{O}(nc)+\mathcal{O}(c^2)+\mathcal{O}(c) = \mathcal{O}(n)$ in quanto per ipotesi c è costante.