Soluzioni Esercitazione III ASD 2023-24

Lorenzo Cazzaro Riccardo Maso Alessandra Raffaetà

Esercizio 1. Scrivere una funzione **EFFICIENTE** `order` che ordini in modo non crescente n numeri interi compresi nell'intervallo da 0 a $n^4 - 1$ nel tempo $\mathcal{O}(n)$.
Il prototipo della procedura è:

```
void order(vector<int>& arr)
```

Analizzare e motivare in modo chiaro, preciso ed approfondito la complessità della procedura.

Soluzione:

```
/*Funzione che opera il countingsort su una singola cifra degli elementi del
    vettore da ordinare.*/
void countingsort(vector<vector<int>>& tab, vector<vector<int>& out, size_t d){
    size_t dim = tab.size();
    vector<int> occ(dim, 0);

    for (size_t i = 0; i < dim; i++)
        occ[tab[i][d]]++;

    for (int i = dim - 2; i >= 0; i--)
        occ[i]+= occ[i + 1];

    for (int i = dim - 1; i >= 0; i--){
        out[occ[tab[i][d]] - 1]= tab[i];
        occ[tab[i][d]]--;
    }
}

/*Funzione che opera il cambiamento di base da base 10 a base dim, dove dim e'
    la dimensione di arr. Il numero di cifre che compone un numero in base dim
     e' 4 dato che i numeri in arr variano tra 0 e dim^4-1.*/
void cambioBase(const vector<int>& arr, vector<vector<int>>& tab){
    for (size_t i = 0; i < arr.size(); i++){
        tab[i].push_back(arr[i]%(arr.size()));
        tab[i].push_back((arr[i]/arr.size())%arr.size());
        tab[i].push_back((arr[i]/(arr.size()*arr.size()))%arr.size());
        tab[i].push_back(arr[i]/(arr.size()*arr.size()*arr.size()));
    }
}

/*Funzione che opera il cambimento di base da base dim a base 10.*/
```

```
void backDecimali(const vector<vector<int>>& tab, vector<int>& arr){
    size_t dim = arr.size();
    for (size_t i = 0; i < dim; i++)
        arr[i] = tab[i][0] + tab[i][1]*dim + tab[i][2]*pow(dim, 2) + tab[i][3]*
    pow(dim,3);
}

void order(vector<int>& arr){
    size_t dim = arr.size();
    vector<vector<int>> tab(dim); // rappresentazione in base dim dei numeri in
     arr
    vector<vector<int>> aux(dim, vector<int>(4)); // vettore di appoggio per
    countingsort

    cambioBase(arr, tab);
    for (size_t d = 0; d < 4; d++)
        if (d%2 == 0)
            countingsort(tab, aux, d);
        else
            countingsort(aux, tab, d);
    backDecimali(tab, arr);
}
```

Per il calcolo della complessità temporale asintotica nel caso peggiore della procedura `order`, consideriamo le chiamate alle procedure `cambioBase`, `countingsort` e `backDecimali`. Le procedure `cambioBase` e `backDecimali` presentano complessità temporale $\Theta(n)$ in quanto operano su tutti gli elementi contenuti in `arr` tramite delle operazioni dal costo costante. Per quanto riguarda la complessità di `radix sort`, si esegue d volte la procedura `countingsort` con $d = 4$ costante. La complessità di `countingsort` è $\Theta(n)$ in quanto l'inizializzazione di `occ` ha costo $\Theta(n)$ e i tre cicli hanno tutti costo $\Theta(n)$ in quanto ciascun ciclo effettua al più n iterazioni eseguendo operazioni costanti. Dunque la complessità di `radix sort` è $\Theta(n)$.

Esercizio 2. Sia **A** un vettore in cui ogni elemento contiene due campi: **A[i].value** contiene un numero intero ed **A[i].color** contiene un colore (**'b'** o **'n'**). Gli elementi di **A** sono ordinati in ordine crescente rispetto al campo value.

1. Si consideri il problema di ordinare gli elementi di **A** rispetto al campo color secondo l'ordinamento $b < n$, facendo in modo che gli elementi dello stesso colore siano ordinati rispetto al campo value. Si scriva una procedura **EFFICIENTE** per risolvere il problema proposto. Si valuti e giustifichi la complessità.
   Il prototipo della procedura è:

   `void ordinaPerColore(vector<ValCol>& arr)`

2. Si consideri il problema di ordinare gli elementi di **A** rispetto al campo value, facendo in modo che gli elementi che hanno stesso value siano ordinati rispetto al campo colore (sempre con la convenzione $b < n$). Si scriva una procedura **EFFICIENTE** per risolvere il problema proposto. Si valuti e giustifichi la complessità.

Il prototipo della procedura è:

```
void ordinaPerValore(vector<ValCol>& arr)
```

Il tipo ValCol è così definito:

```
typedef struct {
        int value;
        char colour;
} ValCol;
```

Soluzione:

```
typedef struct{
    int value;
    char colour;
} ValCol;

void ordinaPerColore(vector<ValCol>& arr){
    vector<ValCol> aux(arr.size());
    int bianchi =0, neri = 0;
    //conta i colori
    for (int i = 0; i < arr.size(); i++)
        if (arr[i].colour == 'b')
            bianchi++;
        else
            neri++;

    neri += bianchi;
    //ordina per colore
    for (int i = arr.size()-1; i >= 0; i--)
        if (arr[i].colour == 'b'){
            aux[bianchi - 1] = arr[i];
            bianchi--;
        }
        else {
            aux[neri - 1] = arr[i];
            neri--;
        }
  for (int i = 0; i < arr.size(); i++)
       arr[i] = aux[i];
}

//riorganizza nell'intervallo "inizio" e "fine"
void riorganizza(vector<ValCol>& arr, int inizio, int fine){
    ValCol temp;
    int bianchi = inizio - 1;
    for (int neri = inizio; neri <= fine; neri++)
        if (arr[neri].colour == 'b'){
            bianchi++;
            temp = arr[neri];
```

```
            arr[neri] = arr[bianchi];
            arr[bianchi] = temp;
        }
}

void ordinaPerValore(vector<ValCol>& arr){
    int i, j;

    i = 0;
    //itera l'array e cerca i blocchi di valori uguali e poi li riorganizza
    while (i < arr.size()) {
        j = i+1;
        while (j < arr.size() && arr[i].value == arr[j].value)
            j++;
        riorganizza(arr, i, j - 1);
        i = j;
    }
}
```

1. Per il calcolo della complessità di `ordinaPerColore` analizziamo la sua struttura, la funzione presenta tre cicli `for` che iterano n volte. Quindi la complessità temporale asintotica della procedura `ordinaPerColore` è $T(n) = \Theta(3n) = \Theta(n)$.

2. Per il calcolo della complessità di `ordinaPerColore` iniziamo calcolando la complessità di `riorganizza`. Al suo interno è presente un ciclo `for` che itera sugli elementi del vettore nella porzione da `inizio` a `fine` eseguendo operazioni costanti. Quindi la sua complessità è $T(m) = \Theta(m)$ con $m = fine - inizio + 1$. Ora nella procedura `ordinaPerValore` è presente un ciclo `while` che itera l'array e trova le sequenze di valore uguali, tali sequenze vengono passate alla funzione `riordina`. Abbiamo che ogni elemento dell'array appartiene ad una ed una sola sequenza e che ogni sequenza viene passata alla funzione `riordina` una sola volta, quindi avremo che la complessità temporale asintotica della procedura `ordinaPerValore` è $T(n) = \Theta(n)$ con n numero di elementi dell'array.

Esercizio 3. Fornire un'implementazione per il tipo di dato `Insieme`, il cui insieme di operazioni è specificato nel file `insieme.hpp`, utilizzando una tabella Hash con concatenamento. Ciascuna entry della tabella Hash deve utilizzare una lista singolarmente concatenata per salvare gli elementi inseriti.

Deve essere completato il file `implHashIndChaining.cpp`.

Per ciascuna operazione specificare e motivare in modo dettagliato la complessità (caso pessimo e caso medio) in modo da rendere le operazioni il più efficienti possibili.

Soluzione:

```
/* insieme.hpp */
class Insieme {
```

```
    public:
        Insieme();
        Insieme(const Insieme& set);
        ~Insieme();
        void inserisci(int val);
        void cancella(int val);
        bool contiene(int val) const;
        int numElementi() const;
        Insieme unione(const Insieme& I_2) const;
        Insieme intersezione(const Insieme& I_2) const;
        Insieme differenza(const Insieme& I_2) const;
        void stampa() const;
    private:
        struct Impl;
        Impl* pimpl;
};

/* implHashChainingDaComp.cpp */
#include "insieme.hpp"
#include <vector>
#include <iostream>

constexpr int hashsize = 8;
using namespace std;

struct Elem{
    int info;
    Elem * succ;
};
typedef Elem * PElem;

struct Insieme::Impl {
    vector<PElem> content;
    int numelementi;
};

int funHash(int val){
    return val%hashsize;
}

/*post: costruisce un insieme vuoto */
Insieme::Insieme(){
    pimpl = new Impl;
    pimpl->numelementi = 0;

    for (size_t i = 0; i < hashsize; i++)
        pimpl->content.push_back(nullptr);
}


/*post: costruisce un insieme contenente gli elementi di set */
Insieme::Insieme(const Insieme& set) {
    pimpl = new Impl;
```

```
    pimpl->numelementi = set.pimpl->numelementi;

    for (size_t i = 0; i < hashsize; i++) {
        PElem elem = set.pimpl->content[i];
        if (elem) {
            PElem new_head = new Elem{elem->info, nullptr};
            pimpl->content.push_back(new_head);
            while(elem->succ) {
                new_head->succ = new Elem{elem->succ->info, nullptr};
                new_head = new_head->succ;
                elem = elem->succ;
            }
        } else {
            pimpl->content.push_back(nullptr);
        }
    }
}

/*post: rimuove l'insieme*/
Insieme::~Insieme(){
    for (size_t i= 0; i < hashsize; i++) {
        PElem elem_to_delete = pimpl->content[i];
        while(elem_to_delete) {
            PElem succ_to_delete = elem_to_delete->succ;
            delete elem_to_delete;
            elem_to_delete = succ_to_delete;
        }
    }
    delete pimpl;
}

/*post: inserisce il valore val nell'insieme se non e' presente */
void Insieme::inserisci(int val){
    if(!contiene(val)) {
        int index = funHash(val);
        PElem x = new Elem{val, pimpl->content[index]};
        pimpl->content[index] = x;
        pimpl->numelementi += 1;
    }
}

/*post: rimuove l'elemento val dall'insieme */
void Insieme::cancella(int val){
    int index = funHash(val);
    PElem elem = pimpl->content[index];
    PElem pred = nullptr;
    bool found = false;
    while (elem && !found)
        if (elem->info == val)
            found = true;
        else {
            pred = elem;
            elem = elem -> succ;
```

```
        }
    if (found){
        if (pred)
            pred->succ = elem->succ;
        else
            pimpl->content[index] = elem->succ;
        delete elem;
        pimpl->numelementi--;
    }
}

/*post: restituisce il numero di elementi nell'insieme*/
int Insieme::numElementi() const{
    return pimpl->numelementi;
}

/*post: restiusice true se l'elemento e' presente nell'insieme, altrimenti
    false */
bool Insieme::contiene(int val) const {
    int index = funHash(val);
    PElem elem = pimpl->content[index];
    bool found = false;
    while(elem && !found) {
        if (elem->info == val) {
            found = true;
        }
        elem = elem->succ;
    }
    return found;
}

/*post: restituisce un nuovo insieme che contiene gli elementi che appartengono
     ad almeno uno dei due insiemi*/
Insieme Insieme::unione(const Insieme& I_2) const {
    Insieme insieme_unione(*this);
    for (size_t i = 0; i < hashsize; i++) {
        PElem elem = I_2.pimpl->content[i];
        while(elem) {
           if (!(this->contiene(elem->info))){
                insieme_unione.pimpl->content[i] = new Elem{elem->info,
    insieme_unione.pimpl->content[i]};
                    insieme_unione.pimpl->numelementi+=1;
            }
            elem = elem->succ;
        }
    }
    return insieme_unione;
}

/*post: restituisce un nuovo insieme che contiene gli elementi in comune tra se
     stesso e I_2*/
Insieme Insieme::intersezione(const Insieme& I_2) const {
    Insieme insieme_intersez;
```

```
    for (size_t i = 0; i < hashsize; i++) {
        PElem elem = pimpl->content[i];
        while(elem) {
            if(I_2.contiene(elem->info)){
                insieme_intersez.pimpl->content[i] = new Elem{elem->info,
    insieme_intersez.pimpl->content[i]};
                insieme_intersez.pimpl->numelementi+=1;
            }
            elem = elem->succ;
        }
    }
    return insieme_intersez;
}


/*post: restituisce un nuovo insieme che contiene gli elementi che appartengono
     ad almeno uno dei due insiemi*/
Insieme Insieme::differenza(const Insieme& I_2) const {
    Insieme insieme_diff;
    for (size_t i = 0; i < hashsize; i++) {
        PElem elem =pimpl->content[i];
        while(elem) {
            if(!I_2.contiene(elem->info)){
                insieme_diff.pimpl->content[i] = new Elem{elem->info,
    insieme_diff.pimpl->content[i]};
                insieme_diff.pimpl->numelementi+=1;
            }
            elem = elem->succ;
        }
    }
    return insieme_diff;
}



/*post: stampa il contenuto dell'insieme */
void Insieme::stampa() const {
    cout << "{ ";
    for (size_t i = 0; i < hashsize; i++) {
        PElem elem = pimpl->content[i];
        while(elem) {
            cout << elem->info << " ";
            elem = elem->succ;
        }
    }
    cout << "}" << endl;
}
```

Indichiamo con $\mathbf{n}$ il numero di elementi memorizzati nella tabella Hash e con $\mathbf{m}$ la dimensione della tabella Hash. Nel caso di operazioni che coinvolgono un altro insieme (`unione`, `intersezione` e `differenza`), $\mathbf{n_2}$ è il numero di elementi memorizzati nella tabella Hash del secondo insieme. Di seguito vengono elencate le complessità nel caso pessimo e nel caso medio per tutte le funzioni:

- `Insieme`: $\Theta(m)$ nel caso pessimo e nel caso medio.
- `copia Insieme`: $\Theta(m+n)$ nel caso pessimo e nel caso medio, dato che è necessario scorrere tutta la tabella che ha `m` celle e gli elementi sono memorizzati all'esterno della tabella e devono essere tutti copiati. Il numero delle chiavi è `n`.
- $\sim$`Insieme`: $\Theta(m+n)$ nel caso pessimo e nel caso medio, dato che è necessario scorrere tutta la tabella che ha `m` celle e gli elementi sono memorizzati all'esterno della tabella e devono essere tutti cancellati. Il numero delle chiavi è `n`.
- `contiene`: $\Theta(n)$ nel caso pessimo e $\Theta(1+\alpha)$ nel caso medio con $\alpha = \frac{n}{m}$.
- `inserisci`: $\Theta(n)$ nel caso pessimo e $\Theta(1+\alpha)$ nel caso medio con $\alpha = \frac{n}{m}$.
- `cancella`: $\Theta(n)$ nel caso pessimo e $\Theta(1+\alpha)$ nel caso medio con $\alpha = \frac{n}{m}$.
- `numElementi`: $\Theta(1)$ nel caso pessimo e nel caso medio.
- `unione`: consideriamo il caso peggiore per l'unione, cioè i due insiemi da unire sono disgiunti e, di conseguenza, l'insieme unione è costituito da tutti gli elementi dei due insiemi. La complessità della copia dell'insieme su cui viene chiamata la funzione è quella di `copia Insieme`, cioè $\Theta(n+m)$ sia nel caso peggiore sia nel caso medio. Vengono poi effettuati n_2 inserimenti che prevedono anche una ricerca senza successo nell'insieme I su cui è invocata l'operazione di unione. La complessità del caso peggiore nello scenario considerato è quindi $\Theta(m + n_2 * n)$ mentre nel caso medio è $\Theta(m + n_2 * (1+\alpha))$, dove $\alpha = \frac{n}{m}$.
- `intersezione`: consideriamo il caso peggiore per l'intersezione, cioè il primo insieme è contenuto nel secondo e, di conseguenza, tutti gli elementi dell'insieme su cui viene chiamata la funzione devono essere inseriti nell'intersezione. La complessità per scorrere tutta la tabella Hash dell'insieme corrente è $\Theta(m+n)$ nel caso peggiore e nel caso medio. Inoltre, viene eseguita `contiene` nel secondo insieme per ogni elemento dell'insieme corrente, quindi il costo di tutte le operazioni di ricerca è $\Theta(n * n_2)$ nel caso peggiore e $\Theta(n * (1+\alpha_2))$ nel caso medio, dove $\alpha_2 = \frac{n_2}{m}$. Vengono infine effettuati n inserimenti all'interno dell'insieme intersezione e ciascun inserimento ha complessità $\Theta(1)$, quindi la complessità di tutti gli inserimenti è $\Theta(n)$ nel caso peggiore e nel caso medio. La complessità dell'intersezione nel caso peggiore è quindi $\Theta(m + n * n_2)$, mentre nel caso medio è $\Theta(m + n * (1+\alpha_2))$.
- `differenza`: consideriamo il caso peggiore per la differenza, cioè i due insiemi sono disgiunti e, di conseguenza, tutti gli elementi dell'insieme su cui viene chiamata la funzione devono essere inseriti nella differenza. La complessità per scorrere tutta la tabella Hash dell'insieme corrente è $\Theta(m+n)$ nel caso peggiore e nel caso medio. Inoltre, viene eseguita `contiene` nel secondo insieme per ogni elemento dell'insieme corrente, quindi il costo di tutte le operazioni di ricerca è $\Theta(n*n_2)$ nel caso peggiore e $\Theta(n*(1+\alpha_2))$ nel caso medio, dove $\alpha_2 = \frac{n_2}{m}$. Vengono infine effettuati n inserimenti all'interno dell'insieme differenza e ciascun inserimento ha complessità $\Theta(1)$, quindi la complessità di tutti gli inserimenti è $\Theta(n)$ nel caso peggiore e nel caso medio. La complessità di `differenza` nel caso peggiore è quindi $\Theta(m + n * n_2)$, mentre nel caso medio è $\Theta(m + n * (1+\alpha_2))$.

- `stampa`: $\Theta(m+n)$ nel caso pessimo e nel caso medio, dato che è necessario scorrere tutta la tabella che ha `m` celle e gli elementi sono memorizzati all'esterno della tabella e devono essere tutti stampati. Il numero delle chiavi è `n`.

Esercizio 4. Fornire un'implementazione per il tipo di dato `Insieme` il cui insieme di operazioni è specificato nel file `insieme.hpp`, utilizzando una tabella `Hash con indirizzamento aperto` di dimensione $\mathbf{m} = 8$ e `doppio Hashing` basato sulle funzioni $\mathbf{h1(k)} = \mathbf{k \bmod m}$ e $\mathbf{h2(k)} = \mathbf{1 + 2 * (k \bmod (m - 3))}$.

Deve essere completato il file `implHashIndAperto.cpp`.

Per ciascuna operazione specificare e motivare in modo dettagliato la complessità (caso pessimo e caso medio) in modo da rendere le operazioni il piú efficienti possibili.

Soluzione:

```
/* insieme.hpp */
class Insieme {
    public:
        Insieme();
        Insieme(const Insieme& set);
        ~Insieme();
        void inserisci(int val);
        void cancella(int val);
        bool contiene(int val) const;
        int numElementi() const;
        Insieme unione(const Insieme& I_2) const;
        Insieme intersezione(const Insieme& I_2) const;
        Insieme differenza(const Insieme& I_2) const;
        void stampa() const;
    private:
        struct Impl;
        Impl* pimpl;
};

/* implHashIndApertoDaComp.cpp */
#include "insieme.hpp"
#include <vector>
#include <iostream>

constexpr int hashsize = 8;
constexpr int hashaux = 5;

using namespace std;

int funHash(int val, int i){
    return (val%hashsize + i*(1+ 2 * (val%hashaux)))%hashsize;
}

struct Elem{
    int info;
```

```
};
typedef Elem *PElem;

struct Insieme::Impl{
    vector<PElem> content;
    int numelementi;
    PElem DEL;
    void inseriscisenzaCheck(int val){
        int index, i;
        bool ok = false;
        i = 0;
        do{
            index = funHash(val, i);
            if (content[index] == nullptr || content[index] == DEL) {
                content[index] = new Elem{val};
                numelementi += 1;
                ok = true;
            }
            else
                i++;
        } while(i < hashsize && !ok);

        if (!ok)
            cout<<"Set pieno\n";
    };
};

/*post: costruisce un insieme vuoto */
Insieme::Insieme(){
    pimpl = new Impl;
    pimpl->numelementi = 0;
    pimpl->DEL = new Elem;

    for (size_t i = 0; i < hashsize; i++)
        pimpl->content.push_back(nullptr);
}


/*post: costruisce un insieme contenente gli elementi di set */
Insieme::Insieme(const Insieme& set) {
    pimpl = new Impl;
    pimpl->numelementi = set.pimpl->numelementi;
    pimpl->DEL = new Elem;

    for (size_t i = 0; i < hashsize; i++)
        pimpl->content.push_back(nullptr);

    for (size_t i = 0; i < hashsize; i++){
        PElem curr_elem = set.pimpl->content[i];
        if (curr_elem != nullptr && curr_elem != set.pimpl->DEL)
            this->pimpl->inseriscisenzaCheck(curr_elem->info);
    }
```

```
}

/*post: rimuove l'insieme */
Insieme::~Insieme(){
    for (size_t i= 0; i < hashsize; i++)
        if (pimpl->content[i] != nullptr && pimpl->content[i] != pimpl->DEL)
            delete pimpl->content[i];
    delete pimpl->DEL;
    delete pimpl;
}

/*post: inserisce il valore val nell'insieme se non presente */
void Insieme::inserisci(int val){
    if(!contiene(val)) {
        this->pimpl->inseriscisenzaCheck(val);
    }
}

/*post: rimuove l'elemento val dall'insieme */
void Insieme::cancella(int val){
    int index, i;
    bool ok;
    i = 0;
    ok = false;
    do{
        index = funHash(val, i);
        if (pimpl->content[index]!= nullptr && pimpl->content[index]!= pimpl->
    DEL && pimpl->content[index]->info == val)
            ok = true;
        else
            i++;
    } while(i < hashsize && !ok && pimpl->content[index]!= nullptr);

    if (ok) {
        PElem elem = pimpl->content[index];
        pimpl->content[index] = pimpl->DEL;
        delete elem;
        pimpl->numelementi -= 1;
    }
}

/*post: restituisce il numero di elementi nell'insieme */
int Insieme::numElementi() const {
  return pimpl->numelementi;
}


/*post: restiusice true se l'elemento e' presente nell'insieme, altrimenti
    false */
bool Insieme::contiene(int val) const {
    int index, i;
    bool ok;
    i = 0;
```

```
    ok = false;
    do{
        index = funHash(val, i);
        if (pimpl->content[index]!= nullptr && pimpl->content[index]!= pimpl->
    DEL && pimpl->content[index]->info == val)
            ok = true;
        else
            i++;
    } while(i < hashsize && !ok && pimpl->content[index]!= nullptr);

    return ok;
}

/*post: restituisce un nuovo insieme che contiene gli elementi che appartengono
     ad almeno uno dei due insiemi*/
Insieme Insieme::unione(const Insieme& I_2) const {
    Insieme insieme_unione(*this);
    for(int i = 0; i < I_2.pimpl->content.size(); i++) {
        PElem e = I_2.pimpl->content[i];
        if(e != nullptr && e != I_2.pimpl->DEL && !(contiene(e->info))) {
            insieme_unione.pimpl->inseriscisenzaCheck(e->info);
        }
    }
    return insieme_unione;
}

/*post: restituisce un nuovo insieme che contiene gli elementi in comune tra se
     stesso e I_2*/
Insieme Insieme::intersezione(const Insieme& I_2) const {
    Insieme insieme_intersez;
    for(int i = 0; i < hashsize; i++) {
        PElem e = pimpl->content[i];
        if(e != nullptr && e != pimpl->DEL && I_2.contiene(e->info)) {
                insieme_intersez.pimpl->inseriscisenzaCheck(e->info);
            }
    }
    return insieme_intersez;
}

/*post: restituisce un nuovo insieme  che contiene gli elementi nell'insieme
    che non sono presenti in I_2*/
Insieme Insieme::differenza(const Insieme& I_2) const {
    Insieme insieme_diff;
    for(int i = 0; i < hashsize; i++) {
        PElem e = pimpl->content[i];
        if(e != nullptr && e != pimpl->DEL && !I_2.contiene(e->info)) {
                insieme_diff.pimpl->inseriscisenzaCheck(e->info);
                }
    }
    return insieme_diff;
}

/*post: stampa il contenuto dell'insieme */
```

```
void Insieme::stampa() const {
    cout << "{ ";
    for (size_t i = 0; i < hashsize; i++)
        if (pimpl->content[i] != nullptr && pimpl->content[i] != pimpl->DEL)
            cout << pimpl->content[i]->info << " ";
    cout << "}" << endl;
}
```

Indichiamo con **n** il numero di elementi memorizzati nella tabella Hash e con **m** la dimensione della tabella Hash. Nel caso di operazioni che coinvolgono un altro insieme (`unione`, `intersezione` e `differenza`), $\mathbf{n_2}$ è il numero di elementi memorizzati nella tabella Hash del secondo insieme. Di seguito vengono elencate le complessità nel caso pessimo e nel caso medio per tutte le funzioni nell'ipotesi che non ci siano elementi cancellati nella tabella Hash.

- `inseriscisenzaCheck`: $\Theta(n)$ nel caso pessimo e $\Theta(\frac{1}{1-\alpha})$ nel caso medio con $\alpha = \frac{n}{m}$.
- `Insieme`: $\Theta(m)$ nel caso pessimo e nel caso medio.
- `copia Insieme`: $\Theta(m + n^2)$ nel caso pessimo e $\Theta(m + \sum_{i=0}^{n-1} \frac{1}{1-\frac{i}{m}})$ nel caso medio.
- ~`Insieme`: $\Theta(m)$ nel caso pessimo e nel caso medio.
- `cancella`: $\Theta(n)$ nel caso pessimo e $\Theta(\frac{1}{1-\alpha})$ nel caso medio con $\alpha = \frac{n}{m}$.
- `contiene`: $\Theta(n)$ nel caso pessimo e $\Theta(\frac{1}{1-\alpha})$ nel caso medio con $\alpha = \frac{n}{m}$.
- `inserisci`: richiama la funzione `contiene` per verificare l'assenza dell'elemento e poi invoca la procedura `inseriscisenzaCheck`. Dunque la complessità è $\Theta(n)$ nel caso pessimo e $\Theta(\frac{1}{1-\alpha})$ nel caso medio con $\alpha = \frac{n}{m}$.
- `numElementi`: $\Theta(1)$ nel caso pessimo e nel caso medio.
- `unione`: consideriamo il caso peggiore per l'unione, cioè i due insiemi da unire sono disgiunti e, di conseguenza, tutti gli elementi del secondo insieme devono essere inseriti nell'insieme unione che contiene inizialmente gli elementi del primo insieme. La complessità della copia dell'insieme su cui viene chiamata la funzione è quella di `copia Insieme`, cioè $\Theta(m+n^2)$ nel caso peggiore e $\Theta(m+\sum_{i=0}^{n-1} \frac{1}{1-\frac{i}{m}})$ nel caso medio. Vengono poi effettuati $\mathbf{n_2}$ inserimenti che prevedono anche una ricerca senza successo dentro l'insieme a cui è applicata l'operazione. Il costo delle ricerche senza successo è $n_2 * n$ nel caso peggiore e $n_2 * \frac{1}{1-\frac{n}{m}}$ nel caso medio. La complessità di ciascun inserimento dipende dal numero di elementi inseriti all'interno dell'insieme unione al momento dell'inserimento. La complessità della sequenza di inserimenti è $\Theta(\sum_{i=n}^{n+n_2-1} i+1)$ nel caso peggiore e $\Theta(\sum_{i=n}^{n+n_2-1} \frac{1}{1-\frac{i}{m}})$ nel caso medio. La complessità del caso peggiore nello scenario considerato è quindi $\Theta(m + n^2 + n_2 * n + n_2^2)$ mentre nel caso medio è $\Theta(m + n_2 * \frac{1}{1-\frac{n}{m}} + \sum_{i=0}^{n+n_2-1} \frac{1}{1-\frac{i}{m}})$.

- `intersezione`: consideriamo il caso peggiore per l'intersezione, cioè il primo insieme è incluso nel secondo e, di conseguenza, tutti gli elementi dell'insieme su cui viene chiamata la funzione devono essere inseriti nell'intersezione. La complessità per scorrere tutta la tabella Hash dell'insieme corrente è $\Theta(m)$ nel caso peggiore e nel caso medio. Inoltre, viene eseguita la funzione `contiene` nel secondo insieme per ogni elemento dell'insieme corrente per verificare la presenza di tale elemento in I_2. Quindi il costo di tutte le ricerche è $\Theta(n*n_2)$ nel caso peggiore e $\Theta(n*(\frac{1}{\alpha_2}\log\frac{1}{1-\alpha_2}))$ nel caso medio, con $\alpha_2 = \frac{n_2}{m}$. Vengono infine effettuati n inserimenti all'interno dell'insieme intersezione e la complessità dell'inserimento varia in funzione del numero di elementi presenti all'interno dell'insieme intersezione al momento dell'inserimento che varia tra 0 e $n-1$. La complessità dell'inserimento degli n elementi è $\Theta(\sum_{i=0}^{n-1} 1+i)$ nel caso peggiore, e $\Theta(\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{m}})$ nel caso medio, dato che viene riempito un insieme inizialmente vuoto. La complessità di `intersezione` nel caso peggiore è quindi $\mathcal{O}(m+n*n_2+n^2)$, mentre nel caso medio è $\Theta(m+n*(\frac{1}{\alpha_2}\log\frac{1}{1-\alpha_2})+\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{m}})$.

- `differenza`: consideriamo il caso peggiore per la differenza, cioè i due insiemi sono disgiunti e, di conseguenza, tutti gli elementi dell'insieme su cui viene chiamata la funzione devono essere inseriti nella differenza. La complessità per scorrere tutta la tabella Hash dell'insieme corrente è $\Theta(m)$ nel caso peggiore e nel caso medio. Inoltre, viene eseguita la funzione `contiene` nel secondo insieme per ogni elemento dell'insieme corrente per verificare l'assenza di tale elemento in I_2. Quindi il costo di tutte le ricerche è $\Theta(n*n_2)$ nel caso peggiore e $\Theta(n*\frac{1}{1-\alpha_2})$ nel caso medio, con $\alpha_2 = \frac{n_2}{m}$. Vengono infine effettuati n inserimenti all'interno dell'insieme differenza e la complessità dell'inserimento varia in funzione del numero di elementi presenti all'interno dell'insieme differenza al momento dell'inserimento che varia tra 0 e $n-1$. La complessità dell'inserimento degli n elementi è $\Theta(\sum_{i=0}^{n-1} 1+i)$ nel caso peggiore, e $\Theta(\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{m}})$ nel caso medio, dato che viene riempito un insieme inizialmente vuoto. La complessità di `differenza` nel caso peggiore è quindi $\Theta(m+n*n_2+n^2)$, mentre nel caso medio è $\Theta(m+n*(\frac{1}{1-\alpha_2})+\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{m}})$

- `stampa`: $\Theta(m)$ nel caso pessimo e nel caso medio.