Documentazione

Andrea Berlingieri

Ultimo aggiornamento:2 agosto 2017

Indice

0.1 Note generali valide per tutte le strutture dati

Tutte le strutture dati sono parametriche: questo vuol dire che possono essere usate per gestire qualsiasi tipo di dato. Quando si vuole istanziare un oggetto è necessario passare anche i tipi di dato come parametri tra parentesi <> dopo il nome della struttura dati. Ad esempio:

```
int main()
{
    ...
    List<int> numbers;
    ...
}
```

Per sapere il numero di parametri richiesti dalla struttura dati basta guardare nell'header della struttura dati, che si trova nella cartella *include*.

In genere tutte le funzioni per il tipo di dato sono dichiarate nell'header, che si trova nella cartella *include*, mentre il codice vero è proprio si trova nella cartella *src*. Di solito è sufficiente controllare l'header di una struttura dati per conoscere la funzione dei metodi, dove ogni metodo è commentato con ciò che fa e le eventuali precondizioni.

Per ogni struttura dati si ha un file di esempio di utilizzo della struttura dati, chiamato *main.cpp*. Per compilarlo è sufficiente utilizzare il comando *make*; verrà creato un eseguibile chiamato *Test*.

Capitolo 1

List

List è un'implementazione di una lista bidirezionale, circolare con sentinella realizzata coi puntatori. Il suo costruttore non richiede parametri. Per iterare lungo la lista si può usare la classe *List<T>::iterator* nel seguente modo:

```
int main()
{
    ...
    List<int> numbers;
    ...
    for(List<int>::iterator it = numbers.begin(); it !=
        numbers.end(); it++) // Scorrimento in ordine
    {
        cout << *it << " ";
    }
    for(List<int>::iterator it = --numbers.end(); it !=
        numbers.end(); it--) // Scorrimento in ordine inverso
    {
        cout << *it << " ";
    }
}
```

1.1 Metodi della classe List

List()

Costruttore della classe List. Crea una lista vuota. Complessita: $O(1)$.

List(const List& list)

Copy constructor: data una lista *list* crea una nuova lista con gli stessi elementi di *list* nello stesso ordine in cui appaiono in *list*. Complessità: $O(n)$.

~List()

Distruttore della classe List. Libera la memoria occupata dalla Lista quando va out of scope. Complessita: $O(1)$.

bool empty() const

Ritorna *true* se la lista è vuota, *false* altrimenti. Complessita: $O(1)$.

bool contains(const T v) const

Dato un elemento di tipo T, restituisce *true* se è contenuto nella lista, *false* altrimenti. Complessita: $O(n)$.

Nota: è richiesto per utilizzare questo metodo che sia implementato l'operatore di confronto == per il tipo di dato memorizzato nella lista.

iterator begin() const

Ritorna un iteratore che punta al primo elemento della lista. Il tipo è *List_iterator<T>*. Complessita: $O(1)$.

Nota: la cella puntata da *begin()* contiene un valore della lista.

iterator end() const

Ritorna un iteratore che punta alla cella successiva all'ultima cella contenente un elemento della lista della lista. Il tipo è *List_iterator<T>*. Complessita: $O(1)$.

Nota: la cella puntata da *end()* **non** contiene un valore della lista. Si tratta della sentinella, utilizzare l'operatore * con un iteratore che punta a tale cella darà risultati inaspettati. Da utilizzarsi per controllare se si è finita la lista durante una scansione.

void insert(const iterator p,const T v) const

Dato un iteratore che punta ad una cella della lista (anche la sentinella), inserisce l'elemento v nella posizione precedente rispetto a quella dell'elemento puntato da p. Complessita: $O(1)$.

void insert(const T v) const

Dato un elemento v, lo inserisce in testa alla lista. Complessita: $O(1)$.

void remove(iterator& p) const

Dato un iteratore p che punta ad un elemento della lista, rimuove tale elemento dalla lista e incrementa p. Complessita: $O(1)$.

void write(const iterator p,const T v) const

Dato un iteratore p che punta ad un elemento della lista e un elemento v, scrive l'elemento v al posto di quello puntato da p. Complessita: $O(1)$.

1.2 Metodi della classe List<T>::iterator

iterator(List<T>::Node* node)

Costruttore. Dato l'indirizzo di un nodo crea un puntatore che punta tale nodo. Complessità:$O(1)$.

iterator()

Costruttore di default. Crea un iterator che non punta ad alcuna cella.

iterator(const iterator it)

Copy constructor. Dato un iteratore *it*, crea un iteratore identico a *it*. Usato negli assegnamenti.

T& operator*()

Operatore di dereferanziamento. Dato un List<T>::iterator *it*, **it* restituisce l'elemento contenuto nella cella puntata da *it* per riferimento. Complessità: $O(1)$.

bool operator==(const iterator & rhs) const

Operatore di confronto. Ritorna *true* se due iteratori puntato alla stessa cella, *false* altrimenti. Complessità: $O(1)$.

bool operator!=(const iterator & rhs) const

Ritorna l'opposto di *p1 == p2*. Complessità: $O(1)$.

iterator& operator++()

Operatore di incremento prefisso ($++p$). Applicato ad un iteratore lo incrementa e restituisce il nuovo iteratore incrementato. Complessità: $O(1)$.

iterator& operator++(int)

Operatore di incremento postfisso ($p++$). Applicato ad un iteratore lo incrementa e restituisce il vecchio iteratore prima dell'incremento. Complessità: $O(1)$.

iterator& operator−−()

Operatore di decremento prefisso ($--p$). Applicato ad un iteratore lo decrementa e restituisce il nuovo iteratore decrementato. Complessità: $O(1)$.

iterator& operator−−(int)

Operatore di decremento postfisso ($p--$). Applicato ad un iteratore lo incrementa e restituisce il vecchio iteratore prima del decremento. Complessità: $O(1)$.

1.3 Esempio di utilizzo

```
#include <iostream>
#include "include/List.h"

using namespace std;

template<typename T>
void print(const List<T>& l);

template<typename T>
void print_backwards(const List<T>& l);

template<typename T>
void incrementByOne(const List<T>& l);

int main()
{
   List<int> l;
   for(int i = 0; i < 10; i++)
      l.insert(l.end(),i);
   List<int> l2(l);

    print(l);
    print_backwards(l);
    incrementByOne(l2);
    print(l2);
    print_backwards(l2);
}

template<typename T>
void print(const List<T>& l)
{
    for(typename List<T>::iterator it = l.begin(); it != l.end(); ++it)
   {
      cout << *it << " ";
   }
```

```
    cout << endl;
}

template<typename T>
void print_backwards(const List<T>& l)
{
    for(typename List<T>::iterator it = --l.end(); it != l.end(); --it)
   {
      cout << *it << " ";
   }

    cout << endl;
}

template<typename T>
void incrementByOne(const List<T>& l)
{
    for(T& n : l)
   {
       n++;
   }
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
```

Capitolo 2

Dequeue

Dequeue è un'implementazione di una Double Endend Queue, ovvero di una coda con inserimento/rimozione sia in testa che in fondo.

L'unico parametro richiesto è il tipo di dati da memorizzare nella coda.

2.1 Metodi della classe Dequeue

bool empty() const

Restituisce *true* se la coda è vuota, *false* altrimenti. Complessità: $O(1)$.

void push(const T v) const

Inserisce l'elemento v in testa alla coda. Corrisponde al *push* in uno Stack. Complessità: $O(1)$.

void push_back(const T v) const

Inserisce l'elemento v in fondo alla coda. Corrisponde al *dequeue* in una Queue. Complessità: $O(1)$.

T& pop() const

Restituisce l'elemento in testa alla coda e lo rimuove dalla coda. Corrisponde al *pop* in uno Stack. Complessità: $O(1)$.

T& pop_last() const

Restituisce l'elemento in fondo alla coda e lo rimuove dalla coda. Complessità: $O(1)$.

T& top() const

Restituisce l'elemento in testa alla coda. Complessità: $O(1)$.

T& last() const

Restituisce l'elemento in fondo alla coda. Complessità: $O(1)$.

2.2 Esempio di utilizzo

```
#include <iostream>
#include "include/dequeue.h"

using namespace std;

int main()
{
    Dequeue<int> q;
    for(int i = 0; i < 10; i++)
        q.push(i);
    cout << q.top() << endl;
    while(!q.empty())
        cout << q.pop() << " ";
    cout << endl;
    for(int i = 0; i < 10; i++)
        q.push(i);
    cout << q.last() << endl;
    while(!q.empty())
        cout << q.pop_last() << " ";
    cout << endl;
    for(int i = 0; i < 10; i++)
        q.push_back(i);
    cout << q.last() << endl;
    while(!q.empty())
        cout << q.pop_last() << " ";
    cout << endl;
}
```

Output:

```
9
9 8 7 6 5 4 3 2 1 0
0
0 1 2 3 4 5 6 7 8 9
9
9 8 7 6 5 4 3 2 1 0
```

Capitolo 3

Hashtable

Hashtable è un'implementazione di una tabella hash con memorizzazione esterna basata su liste di trabocco realizzate tramite la struttura dati List che vengono memorizzate all'interno di un vettore. Richiede due parametri: il tipo della chiave ed il tipo del valore.

Mentre per il tipo del valore non è richiesto alcunchè, per il tipo della chiave è necessario che venga implementato il metodo hash del namespace std, di modo che sia possibile generare un codice hash data una chiave. Lo schema generale è il seguente:

```
...
namespace std
{
    template <> struct hash<T>
    {
        size_t operator()(T & n) const
        {
            ...
            Codice che genera l'hash code
            ...
        }
    };
}
...
```

dove T rappresenta un generico tipo. È possibile avvalersi della funzione std::hash, già definita in C++ per diversi tipi primitivi e per le stringhe. Esempio:

```
...
class Node {

    private:

        string name;
```

```
        int weight;
...
};
...

namespace std
{
    template <> struct hash<Node>
    {
        size_t operator()(Node & n) const
        {
            return hash<string>()(n.getName());
        }
    };
}
...
```

Inoltre è necessario implementare l'operatore di confronto == tra due oggetti della classe da utilizzare come chiave. Per farlo è necessario aggiungere la dichiarazione dell'operatore al corpo della classe e implementarlo poi fuori dalla classe, in questo modo:

```
class Node
{
    private:
        string name;
        int weight;
    ...
    public:
        friend bool operator ==(const Node& n1, const Node& n2);
    ...
};

bool operator ==(const Node& n1, const Node& n2)
{
    return (n1.name == n2.name) && (n1.weight == n2.weight);
}
```

Nota: nella dichiarazione è necessaria la keyword *friend*, nella definizione non bisogna metterla.

3.1 Metodi della struct HashPair

HashPair è una struct che mantiene la coppia chiave-valore nella tabella. Richiede due parametri: il tipo della chiave ed il tipo del valore.

bool opeartor ==(const HashPair rhs)

Ritorna *true* sue due coppie chiave-valore hanno la stessa chiave, *false* altrimenti. Complessità: $O(1)$.

bool opeartor ==(const K key)

Ritorna *true* se la chiave di una coppia chiave-valore è uguale a *key*, *false* altrimenti. Complessità: $O(1)$.

Nota: l'HashPair deve apparire come primo elemento nell'operazione di confronto; $k == hashpair$, dove k è un generico elemento di tipo K e $hashpair$ è un generico elemento di tipo $HashPair < K, V >$, darà un errore di compilazione.

3.2 Metodi della classe HashTable<K,V>::iterator

HashTable<K,V>::iterator è una classe utilizzata per iterare in una tabella hash contenente valori di tipo K come chiavi e valori di tipo V come valori.

bool operator == <>(const hash_iterator rhs)

Operatore di confronto tra iteratori. Ritorna *true* se i due iteratori si riferiscono alla stessa tabella ed allo stesso elemento della tabella. Complessità: $O(1)$.

bool operator != <>(const hash_iterator rhs)

Operatore di confronto tra iteratori. Ritorna l'opposto di *it1 == it2*. Complessità: $O(1)$.

iterator operator++()

Operatore di incremento prefisso ($++p$). Applicato ad un iteratore lo incrementa e restituisce il nuovo iteratore incrementato. Complessità: $O(1)$.

iterator operator++(int)

Operatore di incremento postfisso ($p++$). Applicato ad un iteratore lo incrementa e restituisce il vecchio iteratore prima dell'incremento. Complessità: $O(1)$.

ValueType& operator *()

Operatore di dereferenziamento. Restituisce l'HashPair a cui il puntatore si riferisce. Complessità: $O(1)$.

Nota: ValueType è sostanzialmente un alias per HashPair nella maggiore parte dei casi. L'unico caso in cui non lo è è se V è *void*, cosa che nell'utilizzo normale di una tabella hash (memorizzare coppie chiave-valore) non avviene.

L'unico caso è l'implementazione dell'HashSet, che richiede di essere generici sull'elemento dereferenziato.

3.3 Metodi della classe HashTable

HashTable(const int capacity)

Data una dimensione, crea una HashTable con un numero pari a *capacity* di liste di trabocco (**bucket list**). Complessità: $O(m)$.

Nota: si consiglia fortemente una dimensione **dispari**, possibilmente un numero primo distante da potenze di 2. Questo perchè con la funzione di hash usata si avranno delle collisioni nell'usare le potenze di 2 o numeri ad esse vicine; per ottenere una tabella efficiente è meglio evitare dimensioni di questo tipo.

~HashTable()

Distruttore della classe HashTable. Libera la memoria occupata dalla tabella quando va out of scope. Complessità: $O(1)$.

bool contains(K k) const

Ritorna *true* se la tabella contiene la chiave k, *false* altrimenti. Complessità: $O(1)$ nel caso medio.

V lookup(const K k) const

Ritorna l'elemento associato alla chiave k, se presente nella tabella; altrimenti ritorna un oggetto di tipo V costruito col costruttore di default e con i valori di default. Complessità: $O(1)$ nel caso medio.

V operator[](const K k) const

Come l'operatore di lookup, ma la chiave viene passata tra parentesi quadre, come se la tabella fosse un vettore. Esempio: *V value = table[key]*. Complessità: $O(1)$ nel caso medio.

void insert(ValueType e)

Inserisce la coppia chiave-valore nella tabella. Complessità: $O(1)$ nel caso medio.

Nota: la coppia chiave-valore deve essere passata tra parentesi quadre, con chiave per prima e chiave e valore separate da una virgola.

void remove(const K key)

Rimuove, se presente, la chiave key ed il valore ad essa associata dalla tabella. Complessità: $O(1)$ nel caso medio.

iterator begin()

Ritorna un iteratore che punta alla prima coppia chiave-valore della tabella. Complessità: $O(m+n)$ nel caso pessimo.

iterator end()

Ritorna un iteratore che punta alla fine della tabella. Complessità: $O(1)$.

Nota: la cella puntata dall'iteratore non contiene alcun elemento della tabella.

3.4 Esempio di utilizzo

```
#include "include/HashTable.h"
#include <limits.h>
#include <iostream>
#include <string>

using namespace std;

struct var
{
   string name,type,kind;
   int n;
};

class Node {

    private:

        string name;
        int weight;

    public:

        Node()
        {
            name = "";
            weight = INT_MAX;
        }

        Node(const Node& m)
        {
            name = m.name;
            weight = m.weight;
        }

        Node(string name, int weight): name(name), weight(weight)
```

```
        {}

        string getName()
        {
            return this->name;
        }

        void setName(string name)
        {
            this->name = name;
        }

        void setWeight(int weight)
        {
            this->weight = weight;
        }

        int getWeight()
        {
            return this->weight;
        }

        void print()
        {
            cout << name << " " << weight << endl;
        }

        friend bool operator ==(const Node& n1, const Node& n2);

};

namespace std
{
    template <> struct hash<Node>
    {
        size_t operator()(Node & n) const
        {
            return hash<string>()(n.getName());
        }
    };
}

bool operator ==(const Node& n1, const Node& n2)
{
    return (n1.name == n2.name) && (n1.weight == n2.weight);
}

int main()
{
```

```
    HashTable<string,Node> H(17);

    H.insert({"numero 1",{"numero 1", 1}});
    H.insert({"numero 2",{"numero 2", 2}});
    H.insert({"numero 3",{"numero 3", 3}});
    H.insert({"numero 4",{"numero 4", 4}});

    HashTable<string,Node> H2(H);

    Node i;

    i = H.lookup("numero 1");
    i.print();
    i = H.lookup("numero 4");
    i.print();
    i = H["numero 2"];
    i.print();
    i = H["numero 3"];
    i.print();

    H.remove("numero 1");
    i = H.lookup("numero 1");
    if(i == Node())
        cout << "Element not found" << endl;



    for(HashTable<string,Node>::iterator it = H.begin(); it != H.end();
         it++)
        (*it).value.print();

    cout << "H2" << endl;
    H2.insert({"numero 1",{"numero 2",2}});
    for(auto p : H2)
        p.value.print();

}
```

Output:

```
numero 1 1
numero 4 4
numero 2 2
numero 3 3
Element not found
numero 4 4
numero 3 3
numero 2 2
H2
numero 4 4
```

```
numero 3 3
numero 2 2
numero 2 2
```

Capitolo 4

Set

Set contiene l'implementazione di un insieme con liste non ordinate, con tabelle hash e con alberi binari di ricerca bilanciati. Si consiglia l'utilizzo degli HashSet, in quanto più efficienti per le operazioni di inserimento, rimozione degli elementi e di verifica che un elemento appartenga all'insieme ($O(1)$ nel caso medio). In alternativa è possibile utilizzare la realizzazione basata su alberi bilanciati, che richiede $O(\log n)$ nel caso pessimo per tali operazioni e che mantiene l'ordine tra gli elementi, ammesso che sia definita una relazione d'ordine totale tra di essi.

4.1 HashSet

La classe HashSet prende come parametro un solo tipo, quello degli elementi. Per questo tipo di dato è necessario prendere gli stessi accorgimenti richiesti per usare tale tipo come chiave in una HashTable, già specificati nella sezione Hashtable.

4.2 Metodi per la classe HashSet

HashSet(const int capacity)

Data una capacità, crea un insieme con tale dimensione. Complessità: $O(m)$.

Nota: essendo l'insieme realizzato con HashTable con memorizzazione esterna, il numero di elementi all'interno dell'insieme può anche superare la capacità. Si consiglia tuttavia di dare una dimensione almeno doppia rispetto a quella teorica che ci si aspetta, di modo che le operazioni sull'insieme restino efficienti. Se il fattore di carico α della HashTable supera 2, non è più una scelta efficiente l'utilizzo della HashTable per la memorizzazione degli elementi. Inoltre sono richieste le stesse dimestichezze nella scelta della dimensione già specificate in HashTable(const int capacity).

bool isEmpty() const

Ritorna *true* se l'insieme è vuoto, *false* altrimenti. Complessità: $O(1)$.

bool contains(const T x)

Ritorna *true* se l'elemento x è contenuto nell'insieme, *false* altrimenti.

bool insert(const T x)

Dato un elemento x, lo inserisce nell'insieme, se non già presente, e ritorna *true*; se l'elemento è già presente, ritorna *false*. Complessità: $O(1)$.

bool remove(const T x)

Dato un elemento x, lo rimuove dall'insieme, se presente, e ritorna *true*; se l'elemento non fa parte dell'insieme, ritorna *false*. Complessità: $O(1)$ nel caso medio.

iterator begin()

Ritorna un iteratore che punta al primo oggetto dell'insieme. Complessità: $O(m+n)$ nel caso pessimo.

iterator end()

Ritorna un iteratore che punta alla fine dell'insieme. Complessità: $O(1)$.

Nota: questo iteratore non punta ad alcun elemento dell'insieme.

int size() const

Ritorna la cardinalità dell'insieme. Complessità: $O(1)$.

4.3 Metodi della classe HashSet::iterator

I metodi di questa "classe" sono gli stessi di quelli della classe HashTable::iterator (si tratta di un alias). Si veda Metodi della classe HashTable<K,V>::iterator per maggiori dettagli.

4.4 TreeSet

Le operazioni sono fondamentalmente le stesse della sezione Metodi per la classe HashSet. Ciò che cambia è la complessità di alcuni metodi. Il costruttore ha complessità $O(1)$, le operazioni *contains*, *insert* e *remove* richiedono $O(\log n)$ nel caso pessimo. *begin* richiede $O(\log n)$ e *end* richiede $O(1)$.

Il *TreeSet* preserva l'ordine tra gli elementi dell'insieme, purché sia definita una relazione d'ordine totale sugli elementi dell'insieme. Per utilizzare il *TreeSet* con un certo tipo di dato, è necessario fare l'overload degli operatori $<, >, ==, ! =$ per tale tipo di dato in modo da poter far confronti tra gli oggetti; per i tipi primitivi ovviamente non è necessario.

4.5 Esempio di utilizzo

```
#include <limits.h>
#include "include/set.h"
#include <iostream>

using namespace std;

class Integer
{
    private:

        int i;

    public:

        Integer()
        {
            i = INT_MAX;
        }

        Integer(int a):i(a){ };

        friend ostream& operator <<(ostream& out,Integer i);

        int getN()
        {
            return this->i;
        }

        Integer operator++(int)
        {
            Integer old = *this;
            i++;
            return old;
        }

        friend bool operator ==(Integer n, Integer m);
        friend bool operator <(Integer n, Integer m);
        friend bool operator >(Integer n, Integer m);
        friend bool operator !=(Integer n, Integer m);
```

```
};

ostream& operator <<(ostream& out,Integer i)
{
    out << i.i;
}

bool operator ==(Integer n, Integer m)
{
    return (n.i == m.i);
}

bool operator <(Integer n, Integer m)
{
    return (n.i < m.i);
}

bool operator >(Integer n, Integer m)
{
    return (n.i > m.i);
}

bool operator !=(Integer n, Integer m)
{
    return !(n == m);
}

namespace std
{
    template <> struct hash<Integer>
    {
        size_t operator()(Integer i) const
        {
            return hash<int>()(i.getN());
        }
    };
}

int main()
{
    UnorderedListSet<Integer> set;

    for(int i = 0; i < 10; i++)
        set.insert(Integer(i));

    cout << "Unordered List set" << endl;

    for(auto element : set)
    {
        cout << element << " ";
```

```
}
cout << endl;

for(int i = 0; i < 5; i++)
    set.remove(Integer(i));

for(UnorderedListSet<Integer>::iterator it = set.begin(); it !=
     set.end(); it++)
{
    cout << *it << " ";
}
cout << endl;

HashSet<Integer> hashSet(193);
cout << "Hash set" << endl;

for(int i = 0; i < 10; i++)
    hashSet.insert(Integer(i));
for(HashSet<Integer>::iterator it = hashSet.begin(); it !=
     hashSet.end(); it++)
    cout << *it << " ";

cout << endl;

for(int i = 0; i < 5; i++)
    hashSet.remove(Integer(i));

for(Integer i : hashSet)
    cout << i << " ";

cout << endl;
if(hashSet.contains(9))
    cout << "The set contains 9" << endl;
else
    cout << "The set does not contain 9" << endl;

TreeSet<Integer> s;
cout << "Tree set" << endl;

for(int i = 0; i < 10; i++)
{
    s.insert(Integer(i));
}

for(TreeSet<Integer>::iterator it = s.begin(); it != s.end(); it++)
{
    cout << *it << " ";
}
cout << endl;
```

```
    for(int i = 0; i < 5; i++)
        s.remove(Integer(i));

    for(auto e : s)
    {
        cout << e << " ";
    }
    cout << endl;
}
```

Output:

```
Unordered List set
9 8 7 6 5 4 3 2 1 0
9 8 7 6 5
Hash set
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
The set contains 9
Tree set
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
```

Capitolo 5

Tree

Tree è un'implementazione di un albero con un numero arbitrario di figli per nodo realizzato tramite puntatori padre-figlio-fratello. Una gerarchia dall'alto verso il basso e da sinistra verso destra è conseguenza della struttura della classe.

La classe Tree contiene l'albero intero. Per lavorare sui singoli nodi si utilizza un puntatore alla classe ListNode.

Un solo parametro è richiesto, il tipo dei dati da mantenere.

5.1 Metodi della classe Tree

Tree()

Crea un albero vuoto. La radice è **nullptr**. Complessità: $O(1)$.

Tree(T v)

Crea un albero con un solo nodo contenente il valore v. Complessità: $O(1)$.

~Tree()

Distruttore. Libera la memoria occupata dall'albero quando questo va out of scope o viene esplicitamente eliminato. Complessità: $O(n)$.

TreeNode* getRoot()

Resitituisce la radice dell'albero. Complessità: $O(1)$.

5.2 Metodi della classe TreeNode

TreeNode()

Costruttore di default. Crea un nodo vuoto. Il valore contenuto in value è indeterminato. Complessità: $O(1)$.

TreeNode(T value)

Crea un nodo contenente l'elemento *value*. Complessità: $O(1)$.

T getValue()

Restituisce il valore contenuto nel nodo. Complessità: $O(1)$.

void setValue(T value)

Scrive *value* al posto del valore contenuto del nodo. Complessità: $O(1)$.

TreeNode* getParent()

Restituisce un puntatore al padre del nodo. Complessità: $O(1)$.

TreeNode* getChild()

Restituisce un puntatore al primo figlio del nodo (Quello più a sinistra). Complessità: $O(1)$.

TreeNode* getSibling()

Restituisce un puntatore al fratello destro del nodo. Complessità: $O(1)$.

void insertChild(TreeNode* t)

Inserisce il sottoalbero contenuto in t come primo figlio del nodo. Complessità: $O(1)$. **Precondizione:** il padre di t deve essere **nullptr**.

void insertSibling(TreeNode* t)

Inserisce il sottoalbero contenuto in t come fratello destro del nodo. Complessità: $O(1)$. **Precondizione:** il padre di t deve essere **nullptr**.

void deleteChild()

Elimina il sottoalbero contenuto nel primo figlio. Complessità: $O(h)$, dove h rappresenta l'altezza dell'albero. Caso pessimo: $O(n)$.

void deleteSibling()

Elimina il sottoalbero contenuto nel fratello destro. Complessità: $O(n)$ nel caso pessimo.

5.3 Esempio di utilizzo

```
#include "include/tree.h"
#include "../Queue/include/queue.h"
#include <iostream>
#include <string>

using namespace std;

template<typename T>
void preVisit(Tree<T>& t);

template<typename T>
void postVisit(Tree<T>& t);

template<typename T>
void inVisit(Tree<T>& t,int i);

template<typename T>
void levelVisit(Tree<T>& t);

int main()
{
   Tree<string> T("+");
   TreeNode<string>* t2 = new TreeNode<string>("x");
   TreeNode<string>* t3 = new TreeNode<string>("*");
   TreeNode<string>* t4 = new TreeNode<string>("f");
   TreeNode<string>* t5 = new TreeNode<string>("5");
   TreeNode<string>* t6 = new TreeNode<string>("2");
   TreeNode<string>* t7 = new TreeNode<string>("y");
   TreeNode<string>* t8 = new TreeNode<string>("-");
   TreeNode<string>* t9 = new TreeNode<string>("z");
   T.getRoot()->insertChild(t2);
   t2->insertSibling(t3);
   t3->insertChild(t4);
   t4->insertSibling(t5);
   t4->insertChild(t6);
   t6->insertSibling(t7);
   t7->insertSibling(t8);
   t8->insertChild(t9);
    cout << "Formula: x + f(2,y,-z)*5" << endl;
    cout << "Previsit:" << endl;
   preVisit(T);
```

```
    cout << "Postvisit:" << endl;
   postVisit(T);
    cout << "Invisit (i = 1):" << endl;
   inVisit(T,1);
    cout << "Visit by levels: " << endl;
   levelVisit(T);
}

template<typename T>
void preVisit(TreeNode<T>* n, int level)
{
    for(int i = 0; i < level; i++)
        cout << "\t";
    cout << n->getValue() << endl;
    TreeNode<T>* u = n->getChild();
    while (u != nullptr)
    {
        preVisit(u,level + 1);
        u = u->getSibling();
    }
}

template<typename T>
void preVisit(Tree<T>& t)
{
    if(t.getRoot() != nullptr)
        preVisit(t.getRoot(),0);
    cout << endl;
}

template<typename T>
void postVisit(Tree<T>& t)
{
    if(t.getRoot() != nullptr)
        postVisit(t.getRoot());
    cout << endl;
}

template<typename T>
void postVisit(TreeNode<T>* t)
{
    TreeNode<T>* u = t->getChild();
    while (u != nullptr)
    {
        postVisit(u);
        u = u->getSibling();
    }
    cout << t->getValue() << " ";
}
```

```
template<typename T>
void inVisit(Tree<T>& t,int i)
{
    if(t.getRoot() != nullptr)
        inVisit(t.getRoot(),i);
    cout << endl;
}

template<typename T>
void inVisit(TreeNode<T>* t,int i)
{
    int k = 0;
    TreeNode<T>* u = t->getChild();
    while ((u != nullptr) && (k < i))
    {
        inVisit(u,i);
        u = u->getSibling();
        k++;
    }
    cout << t->getValue() << " ";
    while (u != nullptr)
    {
        inVisit(u,i);
        u = u->getSibling();
    }
}

template<typename T>
void levelVisit(Tree<T>& t)
{
    if(t.getRoot() != nullptr)
    {
        Queue<TreeNode<T>*> q;
        q.enqueue(t.getRoot());
        while (!q.isEmpty())
        {
            TreeNode<T>* u = q.dequeue();
            cout << u->getValue() << " ";
            u = u->getChild();
            while (u != nullptr)
            {
                q.enqueue(u);
                u = u->getSibling();
            }
        }
        cout << endl;
    }
}
```

Output:

```
Formula: x + f(2,y,-z)*5
Previsit:
+
   x
   *
      f
         2
         y
         -
            z
      5

Postvisit:
x 2 y z - f 5 * +
Invisit (i = 1):
x + 2 f y z - * 5
Visit by levels:
+ x * f 5 2 y - z
```

Capitolo 6

PriorityQueue

PriorityQueue è l'implementazione di una coda con priorità crescente realizzato con min-heap. La complessità di molti dei metodi è $O(\log n)$.

Un solo parametro è richiesto, quello del tipo di dato da memorizzare. Si richiede che per tale tipo di dato sia fatto l'overload dell'operatore di confronto ==.

6.1 Metodi della classe PriorityItem

int getPriority()

Getter per la priorità dell'elemento. Complessita: $O(1)$.

void setPriority(int p)

Setter per la priorità dell'elemento. Complessita: $O(1)$.

T getValue()

Restituisce il valore contenuto nell'elemento. Complessita: $O(1)$.

void setValue(const T value)

Setter per il valore dell'elemento. Complessita: $O(1)$.

int getPos()

Getter per la posizione dell'elemento nella PriorityQueue. Complessita: $O(1)$.

void setPos(const int p)

Setter per la posizione dell'elemento nella PriorityQueue. Complessita: $O(1)$.

6.2 Metodi per la classe PriorityQueue

PriorityQueue()

Costruttore di defualt. Da non usare.

PriorityQueue()

Distruttore. Libera la memoria occupata dalla coda con priorità una volta che la vita dell'oggetto è arrivata a termine. Complessita: $O(n)$.

PriorityQueue(int n)

Crea una coda con priorità con capacità n. Complessita: $O(n)$.

bool isEmpty()

Ritorna *true* se la coda è vuota, *false* altrimenti. Complessita: $O(1)$.

T min()

Restituisce l'elemento con priorità maggiore nella coda. Complessita: $O(1)$.

T deleteMin()

Restituisce l'elemento con priorità maggiore nella coda e lo rimuove dalla coda. Complessita: $O(\log n)$.

PriorityItem¡T¿ insert(T x, int p)

Inserisce l'elemento x nella coda con priorità p. Restituisce il PriorityItem corrispondente, da utilizzare per modificare la priorità dell'elemento. Complessita: $O(\log n)$.

void decrease(PriorityItem<T>& x,int p)

Cambia la proprità del PriorityItem x, facendola diventare p. Complessita: $O(\log n)$.

void decrease(T x,int p)

Cambia la proprità dell'elemento x, facendola diventare p. Complessita: $O(n)$.

6.3 Esempio di utilizzo

```
#include "include/priorityqueue.h"
#include <iostream>

using namespace std;

int main()
{
    PriorityQueue<int> q(20);

    // Test isEmpty() #1
    cout << q.isEmpty() << endl;
    // Test insert
    for(int i = 9; i >= 0; i--)
        q.insert(i,i);
    // Test min and deleteMin
    cout << q.min() << endl;
    for(int i = 0; i < 5; i++)
        cout << q.deleteMin() << " ";
    cout << endl;
    // Test isEmpty #2
    cout << q.isEmpty() << endl;
    // Test decrease
    PriorityItem<int> a = q.insert(19,19);
    q.decrease(a,0);
    for(int i = 0; i < 6; i++)
        cout << q.deleteMin() << " ";
    cout << endl;
}
```

Output:

```
1
0
0 1 2 3 4
0
19 5 6 7 8 9
```