Documentazione

Andrea Berlingieri

Ultimo aggiornamento:16 luglio 2017

Indice

0.1 Note generali valide per tutte le strutture dati 1

1 List **2**
1.1 Metodi della classe List 2
1.2 Metodi della classe List_iterator 4
1.3 Esempio di utilizzo 5

2 Hashtable **6**
2.1 Hash::Set 7

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_iterator* nel seguente modo:

```
int main()
{
    ...
    List<int> numbers;
    ...
    for(List_iterator<int> it = numbers.begin(); it !=
        numbers.end(); it++)
    {
        cout << *it << " ";
    }
}
```

1.1 Metodi della classe List

List()

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

~List()

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

bool empty()

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

bool finished(iterator p)

Dato un List_iterator p, ritorna *true* se punta alla fine della lista, *false* altrimenti. Complessita: $O(1)$.

bool contains(T v)

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

iterator begin()

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()

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(iterator p,T v)

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

void insert(T v)

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

void remove(iterator& p)

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

void write(iterator p,T v)

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_iterator

List_iterator(ListNode<T>* node)

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

List_iterator()

Costruttore di default.

T& operator*()

Operatore di dereferanziamento. Dato un List_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;

int main()
{
    List<int> l;
    List_iterator<int> it;
    for(int i = 0; i < 10; i++)
        l.insert(i);
    for(it = l.begin(); it != l.end(); it++)
    {
        cout << *it << " ";
    }

    cout << endl;

    for(it = --l.end(); it != l.end(); it--)
    {
        cout << *it << " ";
    }
    cout << endl;

}
```

Output:

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

Capitolo 2

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, che 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());
        }
    };
}
...
```

Oltre alla versione che richiede due parametri sotto al namespace *keyOnly* ce n'è una che richiede solo un parametro, quello della chiave. Viene utilizzata per realizzare un insieme basato su Hashtable, non ha altri usi utili, si consiglia, se serve memorizzare una collezione di valori in modo da effettuare in modo efficiente ($O(1)$ nel caso medio) operazioni di inserimento, eliminazione e lookup, si consiglia di utilizzare la struttura dati Hash::Set.

2.1 Hash::Set