Documentazione

Andrea Berlingieri

Ultimo aggiornamento:15 luglio 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_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: $O(1)$.