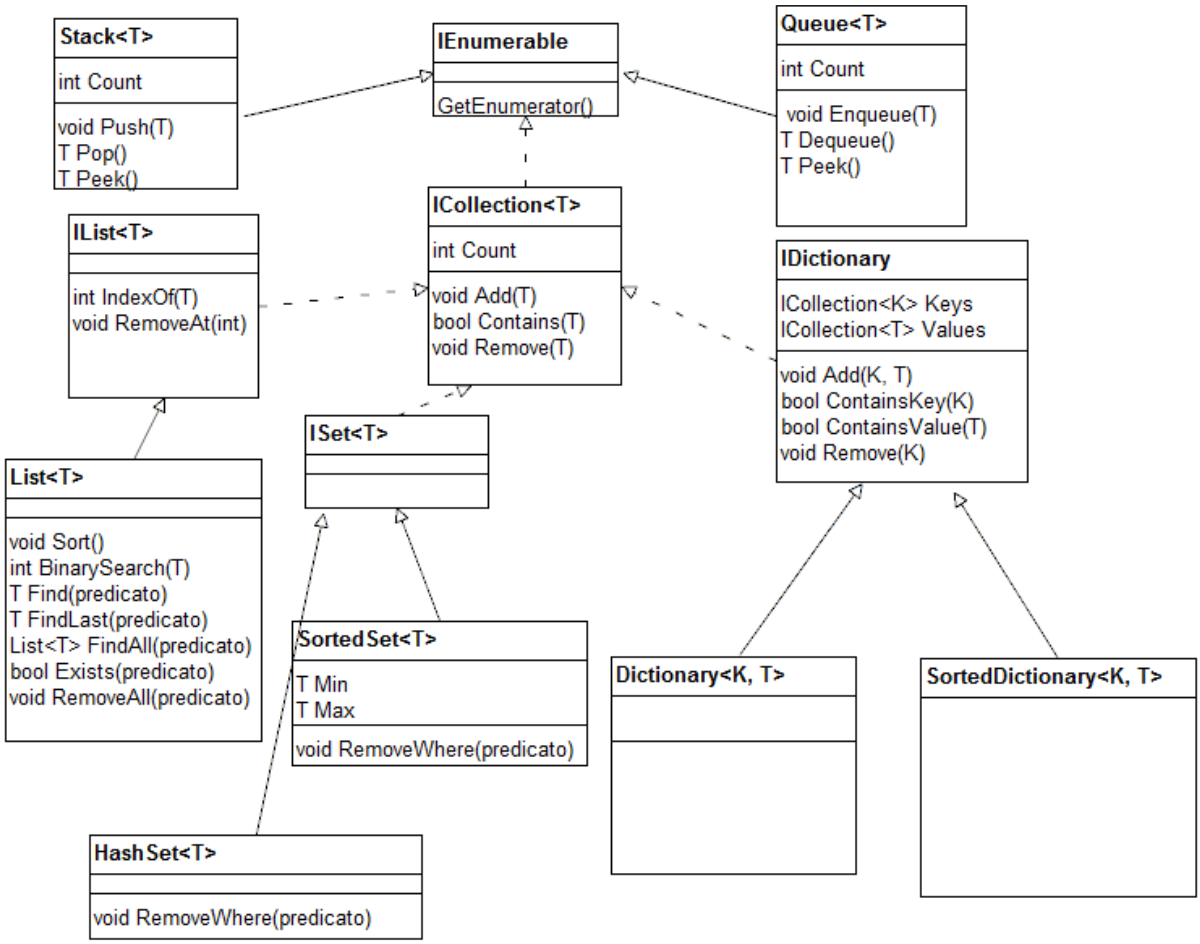
Strutture Dati in RAM e uso della libreria LINQ

Strutture dati in RAM

A partire dalla versione 2 di C#, sono state introdotte le collezioni di tipi generici T.

Al vertice della gerarchia c'è l'interfaccia IEnumerable<T> che prevede il metodo GetEnumerator() il quale in sostanza consente di utilizzare il costrutto "foreach" per visitare una collezione di dati.

Poi c'è l'interfaccia ICollection<T> che unifica tutti i diversi tipi di strutture dati. C'è la categoria rappresentata dall'interfaccia IList<T> che consiste in collezioni di oggetti di tipo generico T, c'è la categoria rappresentata dall'interfaccia ISet<T> che rappresenta insiemi di dati senza ripetizioni e poi quella rappresentata dall'interfaccia IDictionary<K, T> che consiste in collezioni di coppie di valori, costituite da una chiave (di tipo K che di solito è una stringa o un numero intero) e da un valore di tipo generico T. Inoltre ci sono le classi concrete Stack<T>, che rappresenta una pila di oggetti, e Queue<T> che rappresenta una coda di oggetti.



La List<T> è la struttura dati più usata e più semplice, essa consiste sostanzialmente in un array dinamico di oggetti di tipo T. Tipicamente i nuovi elementi vengono aggiunti in coda a quelli presenti, con il metodo Add(), e la ricerca avviene in modo sequenziale (si parte dal primo elemento e si prosegue finché si trova l'elemento cercato).
Per i tipi semplici o per oggetti che implementano il metodo Equals() si può effettuare una ricerca con il metodo IndexOf() che restituisce la posizione del primo elemento trovato che risulta uguale (secondo quanto previsto dal metodo Equals()) all'elemento cercato (-1 se non trovato).
Con lo stesso criterio si può usare il metodo Contains() che restituisce true o false a seconda della presenza o meno dell'elemento cercato.
Per maggiore flessibilità si può usare il metodo Find() che applica come criterio di ricerca una espressione lambda.

Esempio di una lista di stringhe:

```
List<string> animali = new List<string>();

animali.Add("Elefante");   // posizione 0
animali.Add("Leone");      // posizione 1
animali.Add("Tigre");
animali.Add("Giraffa");

// si usa molto comodamente il ciclo foreach
foreach(string a in animali)
{
      Console.WriteLine(a);
}

// si può usare anche il ciclo for
for(int i=0; i<animali.Count; i++)
{
    Console.WriteLine( animali[i] );
}

// ricerca sequenziale
int posizione = animali.IndexOf("Tigre");  // posizione 2


// ordinamento secondo il metodo CompareTo()
animali.Sort();


// ora che i dati sono stati ordinati si può fare la ricerca binaria
int indice = animali.BinarySearch("Tigre");

// L'eliminazione di un elemento avviene sfruttando implicitamente il metodo Equals()
animali.Remove("Elefante");
```

La **pila** Stack è una struttura dati che funziona con il **criterio LIFO** (Last In First Out), ovvero l'ultimo arrivato va in cima alla pila e il prossimo ad essere servito è proprio l'elemento che si trova in cima alla pila. Intuitivamente, si tratta della situazione di chi deve lavare i piatti sporchi che vengono posti in una pila. La pila viene utilizzata anche nei giochi di carte, quando si scartano delle carte che vanno messe in una pila da cui altri giocatori, a loro volta, possono pescare. I metodi sono: Push() per aggiungere un elemento, Pop() per prendere l'elemento in cima, Peek() per guardare l'elemento in cima, senza toglierlo dalla pila.

La **coda** Queue è una struttura dati che funziona con il **criterio FIFO** (First In First Out), ovvero l'ultimo arrivato va in fondo alla coda e il prossimo ad essere servito è l'elemento che si trova in prima posizione. Intuitivamente, si tratta della classica situazione di una fila di persone in coda davanti ad uno sportello. I metodi sono: Enqueue() per inserire in coda, Dequeue() per prendere il primo elemento e toglierlo dalla coda, Peek() per guardare il primo elemento, senza toglierlo dalla coda.

Le principali strutture dati che implementano ICollection<T> sono List<T>, SortedSet<T> e HashSet<T>. Esse vengono tutte utilizzate nello stesso identico modo, cambia solo la loro "implementazione" e quindi i tempi di accesso e lo spazio di memoria che esse richiedono.

All'80% delle volte conviene usare il tipo List<T> che consente di memorizzare un elenco di oggetti e di effettuare ricerche (metodi Find, FindAll) e di ordinarli (metodo Sort) e di visualizzarli tutti con un ciclo "foreach". I tempi di ricerca con una lista sono quelli di una scansione sequenziale di tutti gli elementi a partire dal primo elemento. Mediamente la ricerca termina dopo aver consultato la metà (n/2) degli elementi; ad esempio con 1.000 di elementi una ricerca richiede di consultarne mediamente 500.

Nel caso in cui si abbiano molte centinaia di elementi, ci si può preoccupare se conviene utilizzare un'altra struttura dati come SortedSet o HashSet per conseguire tempi di ricerca più rapidi.

Il SortedSet prevede il mantenimento di un ordine tra gli elementi della collezione, che non possono avere doppioni, e consente ricerche più rapide nel caso in cui si scelga di usare il metodo BinarySearch(). I tempi di ricerca sono dell'ordine del logaritmo in base 2 della numerosità n della collezione: con 1000 elementi una ricerca richiede di consultarne 10. L'implementazione in memoria di un SortedSet viene effettuata mediante un cosiddetto "albero binario" di elementi.

L'HashSet è una struttura dati che richiede uno spazio di memoria almeno doppio rispetto allo spazio occupato dai dati veri e propri. Essa prevede che un dato venga inserito in una posizione che dipende dal valore del dato stesso: si effettua una KAT (Key To Address Transformation) mediante una apposita funzione che transforma il valore di una chiave in un indirizzo di memorizzazione.

La logica di questa struttura dati è quella di sparpagliare il più possibile i dati all'interno della struttura dati in modo da evitare il più possibile di avere valori che occupino posizioni consecutive o peggio che vadano a "collidere" nella stessa posizione. In questo secondo caso si può scegliere di ricalcolare la posizione dell'elemento oppure di posizionarlo nella prima cella libera successiva a quella calcolata.

La ricerca di un singolo valore avviene praticamente sempre in modo immediato, a prescindere dalla numerosità della collezione dati. Si ottengono così tempi di ricerca spettacolarmente bassi, imbattibili da qualsiasi altro tipo di struttura dati. Il difetto è quello di perdere completamente qualsiasi possibile ordinamento dei dati.

Un esempio di KAT è dato dalla funzione che calcola la somma dei codici ASCII dei caratteri che costituiscono una stringa e poi ne calcola il modulo rispetto alla dimensione della collezione, che è opportuno che sia un numero primo.

Qualora la percentuale di utilizzo dello spazio a disposizione della struttura dati arrivi a superare il 50% si comincia ad osservare un deterioramento delle prestazioni in quanto cominciano ad ammassarsi dati in

alcune zone della struttura. E' giunto quindi il momento di ricreare una nuova struttura dati di dimensioni maggiori e trasferirvi tutti i dati.

Esercitazione: controllare la presenza di alcune parole nel vocabolario della lingua italiana e cronometrare il tempo richiesto a seconda della struttura dati utilizzata:List<string>, SortedSet<string>, HashSet<string>.

Il Dictionary è una struttura dati che consente di associare dei valori semplici o degli oggetti complessi a delle chiavi identificatrici. Esempio di un dizionario di parole con il loro significato:

```
Dictionary<string, string> dizionario = new Dictionary<string, string>();

dizionario.Add("adamo", "primo uomo");
dizionario.Add("atomo", "entità non divisibile");
```

La ricerca si fa tramite una chiave. Conviene innanzitutto accertarsi dell'esistenza della chiave per evitare un errore run-time quando si accede al valore associato alla chiave nel caso in cui la chiave cercata non esista.

```
if (dizionario.ContainsKey("eva"))
{
    string significato = dizionario["eva"];
    Console.WriteLine(significato);
}
```

Si noti che l'accesso al valore associato ad una chiave avviene in modo del tutto analogo a quanto avviene negli array statici, dove si scrive arr[i] per accedere all'elemento che si trova nella posizione i-esima; la differenza è che con i dizionari l'accesso è basato non sulla posizione bensì sul valore della chiave, come ad esempio: dizionario["eva"]. E' per questo motivo che i dizionari sono anche detti "array associativi".

Il SortedDictionary funziona nello stesso modo del Dictionary con la differenza che esso usa internamente una struttura ad albero per mantenere sempre ordinate le chiavi al prezzo di un rallentamento dei tempi di ricerca, che sono gli stessi di quelli di un SortedSet.

Il vantaggio principale è quello che si può ottenere facilmente una stampa ordinata delle coppie chiave-valore contenute nel dizionario:

```
foreach (KeyValuePair<string, string> elemento in dizionario)
{
      Console.WriteLine(elemento.Key + elemento.Value);
}
```

Per semplificare la suddetta sintassi si può anche scrivere:

```
foreach (var elemento in dizionario)
{
      Console.WriteLine(elemento.Key + elemento.Value);
}
```

Uso della libreria LINQ

La libreria LINQ (Language Integrated Query) è stata introdotta a partire da C# 3.5.

Essa contiene un insieme di funzioni che agiscono su una collezione che implementa l'interfaccia IEnumerable<T>, che consente di unificare la gestione di tutti i diversi tipi di strutture dati.

Essa consente di semplificare moltissimo (grazie al suo approccio dichiarativo che prevede di esprimere "che cosa" si vuole senza dover specificare passo passo la procedura da attuare per ottenerlo) la ricerca, la selezione, l'ordinamento e, in generale, la manipolazione di qualsiasi tipo di struttura dati presente in memoria.

Inoltre essa consente la composizione di funzioni in modo del tutto naturale, in quanto si tratta di funzioni che si applicano ad una generica collezione IEnumerable<T> e che restituiscono a loro volta una collezione IEnumerable<T>.

Esempio:

```
// creazione e caricamento di una lista di persone
List<Persona> elenco = new List<Persona>();
elenco.Add( new Persona("gianni", 33, "TV") );
elenco.Add( new Persona("mario", 25, "TV") );
elenco.Add( new Persona("laura", 23, "PD") );

var risultatoParziale = elenco.Where(x => x.Città == "TV");

var risultatoFinale = risultatoParziale.OrderBy(x=>x.Nome);
```

scrivendo **var** si lascia al compilatore l'inferenza automatica sul tipo di variabile che in questo caso è IEnumerable<Persona> (si tratta di una soluzione molto comoda!)

Le due suddette istruzioni equivalgono a scrivere su un'unica riga una concatenazione di funzioni:

```
var risultatoFinale = elenco.Where(x => x.Città == "TV").OrderBy(x=>x.Nome);
```

Esiste anche una cosiddetta QUERY SYNTAX, ispirata al linguaggio SQL, per esprimere tale sequenza di operazioni:

```
var risultato Finale = FROM elenco AS x-   x IN elenco
                       WHERE x.Città == "TV"
                       ORDER BY x.Nome;
```

L'uso di tale sintassi può piacere o meno, tuttavia si ricordi che il prezzo della sua apparente maggiore facilità di lettura è una potenza espressiva leggermente inferiore.

Se si vuole effettuare una selezione di elementi mediante criterio composto si possono utilizzare tranquillamente gli operatori logici **&&** (and) e **||** (or)

var risultato = elenco.Where(x => x.Città == “TV” || x.Città == “PD”).OrderBy(x=>x.Nome);

Se si vuole impostare un criterio multiplo di ordinamento, prima per Nome e poi, a parità di nome, per Età:

var risultato = elenco.Where(x => x.Città == “TV”).OrderBy(x=>x.Nome).ThenBy(x =>x.Età);

Se si vogliono estrarre le persone che sono studenti e che abbiano voto >= 6, ipotizzando che il tipo Studente derivi dal tipo Persona:

var risultato = elenco.OfType<Studente>().Where(x=>x.Voto >= 6);

in quest’ultimo caso var corrisponde al tipo IEnumerable<Studente>

Il tipo IEnumerable<T> consente di applicare un ciclo foreach alla collezione di dati per poterli visualizzare o per effettuare qualche elaborazione, e può anche essere direttamente associato a controlli dell’interfaccia grafica come ListBox o ComboBox o DataGrid.

Tuttavia, se si desidera ottenere una lista di oggetti si può convertire il tutto con ToList()

List<Persona> lista = elenco.Where(x=>x.Città == “TV”).ToList();

Esistono molteplici funzioni nella libreria LINQ, come ad esempio Max, Min, Sum, Average, Count.

Se per esempio si volesse ottenere la media dei voti degli studenti con voto positivo, si scriverebbe:

double media = elenco.OfType<Studente>().Where(x=>x.Voto >= 6).Average(x=>x.Voto);

Rif. Bibliografici: Bochicchio Daniele e altri, “C # - Guida completa per lo sviluppatore” – Ed. Hoepli – 2010