Módulos Basicos (v3 - 08/11/2023)

Algoritmos y Estructuras de Datos

Este documento contiene los módulos que se pueden usar en la resolución de ejercicios sin tener que definirlos. Estos van a servir para implementar estructuras y módulos más complejos.

- ListaEnlazada
- PilaSobreLista
- ColaSobreLista
- Vector
- ConjuntoLineal
- ConjuntoLog
- ConjuntoDigital
- DiccionarioLineal
- DiccionarioLog
- DiccionarioDigital
- ColaDePrioridad (pendiente)

1. ListaEnlazada

El módulo Lista Enlazada implementa el TAD Secuencia. Permite la inserción, modificación, borrado y acceso eficiente del primer y último elemento. En cambio, el acceder en forma directa a un elemento arbitrario (llamado 'acceso directo' o 'acceso aleatorio') tiene un costo lineal. Este módulo implementa lo que se conoce como una lista doblemente enlazada, con punteros al inicio y al fin. Su estructura se apoya en el tipo NodoLista, que contiene el dato a guardar y punteros al próximo nodo y al anterior.

La complejidad de cada una de sus operaciones, de acuerdo a lo definido en el TAD Secuencia, es la siguiente:

ListaEnlazada

proc	complejidad
`listaVacía`	$O(1)$
`longitud`	$O(1)$
`vacía`	$O(1)$
`agregarAdelante`	$O(cp(e))$
`agregarAtrás`	$O(cp(e))$
`fin`	$O(1)$
`comienzo`	$O(1)$
`primero`	$O(1)$
`último`	$O(1)$
`obtener`	$O(n)$
`eliminar`	$O(n)$
`modificarPosicion`	$O(n + cp(e))$
`concatenar`	$O(m * cp(e))$
`iterador`	$O(1)$

donde n es la cantidad de elementos de la lista, m es la cantidad de elementos de la segunda lista interviniente (si la hubiera), y $cp(e)$ es el costo de copiar el elemento e a la lista.

IteradorBidireccional

proc	complejidad
haySiguiente	$O(1)$
hayAnterior	$O(1)$
siguiente	$O(1)$
anterior	$O(1)$

Por otra parte, su estructura está dada por la siguiente definición:

```
NodoLista<T> es struct<
    val: T,
    siguiente: NodoLista,
    anterior: NodoLista
>

Modulo ListaEnlazada<T> implementa Secuencia<T> {
    var cabeza: NodoLista<T> // puntero al primer elemento
    var último: NodoLista<T> // puntero al último elemento
    var longitud: int        // cantidad total de elementos

    ...
}
```

El puntero al primer elemento, junto con el campo `siguiente` de la estructura texttttNodoLista nos permite iterar en orden creciente e insertar adelante en O(1). El puntero al último elemento, por otra parte, nos permite iterar de atrás para adelante e insertar al final. Finalmente el campo `longitud` nos permite conocer el tamaño en O(1).

2. PilaSobreLista

El módulo PilaSobreLista implementa el TAD Pila. Provee una pila en la que sólo se puede acceder al tope de la misma. Por este motivo, no incluye iteradores. Se representa con una ListaEnlazada lo que nos permite realizar todas las operaciones en O(1).

proc	complejidad
pilaVacía	$O(1)$
vacía	$O(1)$
encolar	$O(cp(e))$
desencolar	$O(1)$
tope	$O(1)$

donde $cp(e)$ es el costo temporal de copiar el elemento que se apila. El costo es lineal si se tratara de un tipo básico o un string de tamaño acotado.

Como mencionamos, su representación es simplemente una ListaEnlazada:

```
Modulo PilaSobreLista<T> implementa Pila<T> {
    var lista: ListaEnlazada<T>
    ...
}
```

3. ColaSobreLista

En forma análoga a la Pila, el módulo ColaSobreLista implementa el TAD Cola utilizando una ListaEnlazada como representación. No incluye iteradores y todas sus operaciones son O(1).

proc	complejidad
colaVacía	$O(1)$
vacía	$O(1)$
encolar	$O(cp(e))$
desencolar	$O(1)$
proximo	$O(1)$

donde $cp(e)$ es el costo temporal de copiar el elemento que se encola. El costo es lineal si se tratara de un tipo básico o un string de tamaño acotado.

Como mencionamos, su representación es simplemente una ListaEnlazada:

```
Modulo ColaSobreLista<T> implementa Cola<T> {
    var lista: ListaEnlazada<T>
    ...
}
```

4. Vector

El módulo Vector provee una secuencia que permite obtener el i-ésimo elemento de forma eficiente. La inserción de elementos es eficiente cuando se realiza al final de la misma, si se utiliza un análisis amortizado (i.e., n inserciones consecutivas cuestan $O(n)$), aunque puede tener un costo lineal en peor caso. La inserción en otras posiciones no es tan eficiente, ya que requiere varias copias de elementos. El borrado de los últimos elementos es eficiente, no así el borrado de los elementos intermedios.

Una consideración a tener en cuenta, es que el espacio utilizado por la estructura es el máximo espacio utilizado en cualquier momento del programa. Es decir, si se realizan n inserciones seguidas de n borrados, el espacio utilizado es $O(n)$ por el espacio del tipo de los elementos guardados (T). Si fuera necesario borrar esta memoria, se puede crear una copia del vector con los elementos sobrevivientes, borrando la copia vieja.

En cuanto al recorrido de los elementos, como los mismos se pueden recorrer con un índice, no se proveen iteradores.

Para describir la complejidad de las operaciones, vamos a utilizar

$$f(n) = \begin{cases} n & \text{si } n = 2^k \text{ para algún } k \\ 1 & \text{en caso contrario} \end{cases}$$

para describir el costo de inserción de un elemento. Vale la pena notar que $\sum_{i=1}^{n} \frac{f(j+i)}{n} \to 1$ cuando $n \to \infty$, para todo $j \in \mathbb{N}$. En otras palabras, la inserción consecutiva de n elementos costará $O(1)$ operaciones por elemento, en términos asintóticos.

Las complejidades de sus operaciones son por lo tanto:

proc	complejidad
`vectorVacío`	$O(1)$
`longitud`	$O(1)$
`vacía`	$O(1)$
`agregarAdelante`	$O(f(n) + cp(e))$
`agregarAtrás`	$O(f(n) + cp(e))$
`fin`	$O(n)$
`comienzo`	$O(n)$
`primero`	$O(1)$
`último`	$O(1)$
`obtener`	$O(1)$
`eliminar`	$O(n)$
`modificarPosición`	$O(1)$
`concatenar`	$O(m)$

donde n es la cantidad de elementos del vector, y m es la cantidad de elementos del segundo vector interviniente (si lo hubiera).

La representación por la que optamos es un Array, el cual se irá incrementando su tamaño a medida que se agregan nuevos elementos. Así, se podrá acceder en forma directa a los elementos, de forma de que se pueda acceder al i-ésimo elemento en O(1).

Ademas, se necesita que agregar elementos tome O(1) en forma amortizada (i.e., $O(f(n))$ operaciones). Para lograr esto, podemos duplicar el tamaño del arreglo cuando este se llena, hacer la copia correspondiente de los elementos anteriores y agregar el nuevo al final.

```
Modulo Vector<T> implementa Secuencia<T> {
    var elems: Array<T>  // Contiene los elementos del conjunto
    var longitud: int    // La cantidad de elementos contenidos efectivamente
    ...
}
```

5. ConjuntoLineal

El módulo ConjuntoLineal provee un conjunto básico en el que se puede insertar, eliminar, y testear pertenencia en tiempo lineal (de comparaciones y/o copias). En cuanto al recorrido de los elementos, se provee un iterador bidireccional.

Las complejidades de las operaciones son las siguientes:

ConjuntoLineal

proc	complejidad
`conjVacío`	$O(1)$
`tamaño`	$O(1)$
`pertenece`	$O(n * eq(T))$
`agregar`	$O(n * eq(T) + cp(e))$
`agregarRápido`	$O(cp(e))$
`sacar`	$O(n * eq(T))$
`unir`	$O(n * m * cp(T) * eq(T))$
`restar`	$O(n * m * eq(T))$
`intersecar`	$O(n * m * eq(T))$
`iterador`	$O(1)$

donde n es la cantidad de elementos del conjunto, m es la cantidad de elementos del segundo conjunto interviniente (si lo hubiera), $eq(T)$ y $cp(T)$ son los costos de comparar y copiar dos elementos del tipo T, respectivamente.

IteradorBidireccional

proc	complejidad
haySiguiente	$O(1)$
hayAnterior	$O(1)$
siguiente	$O(1)$
anterior	$O(1)$

La representación por la que optamos es una lista enlazada.

```
Modulo ConjuntoLineal<T> implementa Conjunto<T> {
    var elems: ListaEnlazada<T>        // los elementos del conjunto.
    ...
}
```

6. ConjuntoLog

El módulo ConjuntoLog provee un conjunto básico en el que se puede insertar, eliminar, y testear pertenencia en tiempo logarítmico (de comparaciones y/o copias). En cuanto al recorrido de los elementos, se provee un iterador bidireccional.

A diferencia del ConjuntoLineal, la operación agregarRápido no permite bajar el costo de agregar un elemento al conjunto.

Las complejidades de las operaciones son las siguientes:

ConjuntoLog

proc	complejidad
`conjVacío`	$O(1)$
`tamaño`	$O(1)$
`pertenece`	$O(\log n * eq(T))$
`agregar`	$O(\log n * eq(T) + cp(e))$
`agregarRápido`	$O(\log n * eq(T) + cp(e))$
`sacar`	$O(\log n * eq(T))$
`unir`	$O((n+m)log(n+m) * cp(T) * eq(T))$
`restar`	$O((n+m)log(n+m) * eq(T))$
`intersecar`	$O((n+m)log(n+m) * cp(T) * eq(T))$
`iterador`	$O(1)$

donde n es la cantidad de elementos del conjunto, m es la cantidad de elementos del segundo conjunto interviniente (si lo hubiera), y $cp(T)$ es el costo de copiar un elemento del tipo T.

IteradorBidireccional

proc	complejidad
haySiguiente	$O(\log n)$
hayAnterior	$O(\log n)$
siguiente	$O(\log n)$
anterior	$O(\log n)$

Es importante destacar que, más allá de la cota de cada operación individual, si se debe recorrer un conjunto por completo, el costo total del recorrido es $O(n)$.

La representación por la que optamos es un árbol AVL. Aquí suponemos T es un tipo básico o, en caso de no serlo (e.g., es una tupla o struct), el orden de los elementos quedará definido sólo por la primera componente.

```
Modulo ConjuntoLog<T> implementa Conjunto<T> {
    var elems: AVL<T>        // los elementos del conjunto.
    var tamaño: int
    ...
}
```

7. ConjuntoDigital

Conjunto implementado con Tries.

El módulo ConjuntoDigital provee un conjunto básico en el que se puede insertar, eliminar, y testear pertenencia en tiempo logarítmico (de comparaciones y/o copias). En cuanto al recorrido de los elementos, se provee un iterador bidireccional.

A diferencia del ConjuntoLineal, la operación agregarRápido no permite bajar el costo de agregar un elemento al conjunto.

Las complejidades de las operaciones son las siguientes:

ConjuntoLog

proc	complejidad
`conjVacío`	$O(1)$
`tamaño`	$O(1)$
`pertenece`	$O(\|e\| * \|a\|)$
`agregar`	$O(\|e\| * \|a\| * eq(a) + cp(e))$
`agregarRápido`	$O(\|e\| * \|a\| * eq(a) + cp(e))$
`sacar`	$O(\|e\| * \|a\| * eq(a))$
`unir`	$O(m * \text{máx}(c2) * \|a\| * eq(a)))$
`restar`	$O(n * m * \|a\|)$
`intersecar`	$O(n * \text{máx}(c1) * m\,\text{máx}(c2))$

donde $|e|$ es la longitud de la clave usada para indexar (i.e., el propio elemento, si fuera simple, o la primera componente, si fuera algún tipo compuesto), n es la cantidad de elementos y máx($c1$) la clave más grande del primer conjunto, y m es la cantidad de elementos y máx($c2$) la clave más grande del segundo conjunto interviniente (si lo hubiera). Para un uso estándar, donde cada símbolo es un carácter que está acotado para algún idioma determinado, el tamaño del alfabeto y el costo de comparación tienen costo constante, por lo que se desestiman en el análisis asintótico. Finalmente, para casos en lo que lo elementos son de longitud acotada, el costo de todas las operaciones es $O(1)$.

IteradorBidireccional

proc	complejidad
haySiguiente	$O(\text{máx}(c) * \|a\| * eq(a))$
hayAnterior	$O(\text{máx}(c) * \|a\| * eq(a))$
siguiente	$O(\text{máx}(c) * \|a\| * eq(a))$
anterior	$O(\text{máx}(c) * \|a\| * eq(a))$

donde $max(c)$ es el máximo tamaño de clave del conjunto. A diferencia de lo que ocurría con los anteriores diccionarios, recorrerlo por completo tiene costo $O(n * \text{máx}(c) * |a| * eq(a))$.

La representación por la que optamos es un árbol Trie. Aquí suponemos T es un tipo que tiene un orden parcial. En caso de ser algún tipo definido por una tupla o struct, el orden será definido sólo por la primera componente.

```
Modulo ConjuntoDigital<T> implementa Conjunto<T> {
    var elems: Trie<T>        // los elementos del conjunto.
    var tamaño: int
    ...
}
```

8. DiccionarioLineal

El módulo DiccionarioLineal implementa el TAD Diccionario. Provee un diccionario básico en el que se puede definir, borrar, y verificar si una clave está definida en tiempo lineal. Cuando ya se sabe que la clave a definir no esta definida en el diccionario, la definición se puede hacer en tiempo $O(1)$.

En cuanto al recorrido de los elementos, se provee un iterador bidireccional que permite recorrer los elementos como si fuera una secuencia de pares `<clave,valor>`.

Las complejidades de las operaciones son las siguientes:

DiccionarioLineal

proc	complejidad
`diccionarioVacío`	$O(1)$
`está`	$O(n)$
`definir`	$O(n * eq(K) + cp(k) + cp(v))$
`definirRápido`	$O(n * eq(K) + cp(k) + cp(v))$
`obtener`	$O(n)$
`borrar`	$O(n)$
`tamaño`	$O(1)$
`iterador`	$O(1)$

donde n es la cantidad de claves definidas en el diccionario, $eq(K)$ es el costo de comparar dos claves entre sí, y $cp(k)$ y $cp(v)$ son los costos de copiar la clave y el valor de la definición en la estructura.

IteradorBidireccional

proc	complejidad
haySiguiente	$O(1)$
hayAnterior	$O(1)$
siguiente	$O(1)$
anterior	$O(1)$

La representación que optamos consiste en definir al diccionario como dos listas, una de claves y otra de significados. La lista de claves no puede tener repetidos, mientras que la de significados si puede. Ademas, la i-ésima clave de la lista se asocia al i-ésimo significado.

```
Modulo DiccionarioLineal<K, V> implementa Diccionario<K, V> {
    var claves: ListaEnlazada<K>    // Lista de claves. No tiene que tener repetidos
    var valores: ListaEnlazada<V>   // Lista de valores. Puede haber repetidos
    ...
}
```

9. DiccionarioLog

El módulo DiccionarioLog implementa el TAD Diccionario. Provee un diccionario básico en el que se puede definir, borrar, y verificar si una clave está definida en tiempo logarítmico. A diferencia del DiccionarioLineal, no cambia el costo en definirRápido con respecto a definir (i.e., no cambia el costo saber que la clave no está definida). En cuanto al recorrido de los elementos, se provee un iterador bidireccional que permite recorrer los elementos como si fuera una secuencia de pares `<clave,valor>`.

Las complejidades de las operaciones son las siguientes:

DiccionarioLog

proc	complejidad
`diccionarioVacío`	$O(1)$
`está`	$O(\log n * eq(K)))$
`definir`	$O(\log n * eq(K) + cp(k) + cp(v))$
`definirRápido`	$O(\log n * eq(K) + cp(k) + cp(v))$
`obtener`	$O(\log n * eq(K))$
`borrar`	$O(\log n * eq(K))$
`tamaño`	$O(1)$
`iterador`	$O(1)$

donde n es la cantidad de claves definidas en el diccionario.

IteradorBidireccional

proc	complejidad
haySiguiente	$O(\log n)$
hayAnterior	$O(\log n)$
siguiente	$O(\log n)$
anterior	$O(\log n)$

Como ocurría con ConjuntoLog, si fuera necesario recorrer el diccionario por completo, el costo total del recorrido es $O(n)$.

La representación por la que optamos es un árbol AVL de tuplas, donde la primera componente es la clave, y la segunda es el valor. Aquí suponemos K es un tipo básico (e.g., no es tupla ni struct) y tiene un orden parcial.

```
Modulo DiccionarioLog<K, V> implementa Diccionario<K, V> {
    var definiciones: AVL<tupla<K,V>>
    var tamaño: int
    ...
}
```

10. DiccionarioDigital

El módulo DiccionarioDigital implementa el TAD Diccionario. Provee un diccionario básico en el que se puede definir, borrar, y verificar si una clave está definida en tiempo lineal con respecto a la longitud de la clave (suponiendo un alfabeto finito). A diferencia del DiccionarioLineal, no cambia el costo en definirRápido con respecto a definir (i.e., no cambia el costo saber que la clave no está definida).

En cuanto al recorrido de los elementos, se provee un iterador bidireccional que permite recorrer los elementos como si fuera una secuencia de pares `<clave,valor>`.

Las complejidades de las operaciones son las siguientes:

DiccionarioLog

proc	complejidad
`diccionarioVacío`	$O(1)$
`está`	$O(\|k\| * \|a\| * eq(a)$
`definir`	$O(\|k\| * \|a\| * eq(a) + cp(k) + cp(v)$
`definirRápido`	$O(\|k\| * \|a\| * eq(a) + cp(k) + cp(v)$
`obtener`	$O(\|k\| * \|a\| * eq(a))$
`borrar`	$O(\|k\| * \|a\| * eq(a))$
`tamaño`	$O(1)$

donde $|k|$ es el tamaño de la clave, $|a|$ es el tamaño del alfabeto, y $eq(a)$ es el costo de comparar dos elementos de este último. Es habitual que el alfabeto usado sea finito, ya que se utilizan caracteres del alfabeto latino más acentos, tildes, otros modificadores y caracteres especiales, por lo que, en esos casos, $|a|$ es constante. Además, en algunas situaciones podemos encontrarnos con que las claves están acotadas en su longitud, lo que implica que $|k|$ es constante, por lo que todos los procedimientos anteriores pasan a tener costo constante ($O(1)$).

IteradorBidireccional

proc	complejidad
haySiguiente	$O(\text{máx}(k) * \|a\| * eq(a))$
hayAnterior	$O(\text{máx}(k) * \|a\| * eq(a))$
siguiente	$O(\text{máx}(k) * \|a\| * eq(a))$
anterior	$O(\text{máx}(k) * \|a\| * eq(a))$

donde $max(k)$ es el máximo tamaño de clave del diccionario. A diferencia de lo que ocurría con los anteriores diccionarios, recorrerlo el diccionario por completo tiene costo $O(n * \text{máx}(k) * |a| * eq(a))$.

La representación por la que optamos es un árbol Trie de tuplas, donde la primera componente es la clave, que debe ser de tipo string o Secuencia, y la segunda es el valor. Aquí suponemos K es un tipo simple (e.g., no es tupla ni struct) y tiene un orden parcial.

```
Modulo DiccionarioDigital<K, V> implementa Diccionario<K, V> {
    var definiciones: Trie<tupla<K,V>>
    var tamaño: int
    ...
}
```

..

11. ColaDePrioridad

Cola de prioridad implementada con Heaps.

...