Reporte tarea 7 - Programación y algoritmos

Erick Salvador Alvarez Valencia, Centro de Investigación en Matemáticas, A.C.

I. Introducción

En en el presente reporte se analizará la creación de un algoritmo que usa listas ligadas para generar un conjunto de palabras únicas ordenadas lexicográficamente y a su vez, contar las frecuencias de aparición de las mismas.
De la misma forma, al final del reporte se verá la manera en la que se programaron algunos ejercicios recursivos.

mds
Septiembre 30, 2017

II. Desarrollo

El programa crea una lista ligada que internamente realiza un algoritmo por inserción para ir ordenando los nodos en función de sus palabras asociadas. Primeramente se define la estructura de un nodo con los siguientes parámetros:

1. **Str:** Un puntero a $char$. Aquí se almacena el texto asociado.
2. **Freq:** Un número entero. El valor de la frecuencia actual del nodo.
3. **Next:** Un puntero a un objeto tipo Nodo. De esta forma se apunta al siguiente valor de la lista.

Una vez que se tiene bien definia la estructura del nodo, se puede proceder a la función que genera un nuevo nodo y lo inicializa con los valores adecuados. Cabe destacar que usando la palabra reservada $typedef$ se creó un tipo que relaciona al apuntador de un nodo.

Algorithm 1 Genera un nuevo nodo.

1: **procedure** CreateNode(str)
2: $res \leftarrow$ Puntero a nodo.
3: $res \leftarrow$ Se asigna memoria al nodo.
4: $res \rightarrow Str$ Se asigna memoria dependiento de la longitud de la palabra.
5: $res \rightarrow Str = str$
6: $res \rightarrow Freq = 1$
7: $res \rightarrow Next = NULL$
8: **return** res

Ahora viene el algoritmo más importante de la lista ligada, la función que añade un nuevo nodo a la lista. Para hacer esto debemos disponer siempre de un nodo que apunte al comienzo de la lista, ahora supongamos que tenemos una lista ordenada con n términos, podemos asumir esto anterior ya que si insertamos un nuevo nodo de forma ordenada en la lista, esta seguirá siendo ordenada, lo anterior se puede demostrar aplicando el principio del buen orden.

Continuando con lo anterior, supongamos que queremos insertar un nodo en la lista de forma ordenada, ahora, debido a que el criterio de orden depende de la palabra asociada al nodo, debemos disponer de una función que mapee un conjunto de dos palabras hacia un valor entero por el que podamos darnos cuenta qué palabra es mayor, y por consiguiente, cuál es menor, para lo anterior podemos usar la función $strcmp$ contenida en la librería $string.h$ la cual hace lo anterior descrito, y por lo mismo podemos realizar la inserción de una forma más sencilla.
Para insertar un nuevo nodo a la lista con n términos se propone el siguiente algoritmo: Se generan dos punteros a nodo: lst y nxt que serviran para ir recorriendo la lista, la idea es que uno siempre esté atrás del otro. Ahora, dentro de un bucle, realizamos una comparación con la palabra asociada al nodo nxt y la palabra que queremos insertar, de aquí se generan tres casos que a su vez tienen algunos sub-casos, los cuales enlistaremos a continuación:

1. **Ambas palabras son iguales:** Para este caso sólo debemos aumentar en 1 el contador de frecuencia asociado al nodo, esto porque no podemos insertar nodos con palabras repetidas en la lista.

2. **La palabra a insertar es mayor lexicográficamente que la del nodo** nxt**:** En esta situación se pueden presentar dos casos:
   a) Nos encontramos al final de la lista. Si estamos en este caso, simplemente insertamos el nuevo nodo a la derecha del nodo nxt ya que la palabra a insertar es mayor que todas las demás.
   b) No nos encontramos al final de la lista. En esta situación aún no podemos determinar cuál es la posición que corresponde al nodo que queremos insertar, esto debido a que la palabra asociada al mismo puede seguir siendo mayor que los nodos consiguientes en la lista. Simplemente hacemos que lst y nxt avancen una posición.

3. **La palabra a insertar es menor lexicográficamente que la del nodo** nxt**:** Al igual que el caso anterior, se nos presentan dos situaciones:
   a) Nos encontramos al principio de la lista. Esta es la situación análoga que la del caso anterior, e indica que la palabra a insertar es menor que todas las que contiene la lista. Para ello generamos un nuevo nodo y lo ponemos atrás del nodo $root$ el cual indica el principio de la lista, y al final apuntamos $root$ hacia este nodo, para no perder la referencia.
   b) No nos encontramos al principio de la lista. En

este caso podemos decir que la palabra a insertar va después del nodo lst y antes del nxt, esto porque la lista se encuentra actualmente ordenada y sabemos que la palabra es menor a la del nodo nxt pero mayor a todas las anteriores, esto por lo que se comentó en el caso anterior. Lo que debemos hacer es asociar el nodo lst a un nuevo nodo que contendrá la palabra que se quiere insertar, posteriormente este nuevo nodo lo asociamos al nodo nxt.

Algo que no se mencionó antes pero que de igual manera hay que considerar es que si el nodo raíz está vacío, entonces la lista está vacía y podemos convertir a la raíz como el nuevo nodo. A continuación se mostrará una implementación del algoritmo antes propuesto, mediante pseudocódigo:

Algorithm 2 Añade un nuevo nodo a la lista.

```
1:  procedure AddNode(root, str)
2:      if root then
3:          lst ← NULL
4:          nxt ← root
5:          while True do
6:              if nxt → Str = str then
7:                  nxt → Freq = nxt → Freq + 1
8:                  Termina ciclo.
9:              else if nxt → Str < str then
10:                 if nxt → Next then
11:                     lst ← nxt
12:                     nxt ← nxt → Next
13:                 else
14:                     nxt ← CreateNode(str)
15:                     Termina ciclo.
16:             else
17:                 if lst! = NULL then
18:                     aux ← CreateNode(str)
19:                     aux → Next ← root
20:                     root ← aux
21:                 else
22:                     lst → Next ← CreateNode(str)
23:                     lst → Next → Next ← nxt
24:                 Termina ciclo.
25:     else
26:     return root
```

Una vez teniendo esta función, ya podemos generar las frecuencias simplemente leyendo palabra por palabra y añadiénto esta a la lista. Cuando se imprima la misma, esta se imprimirá en orden de las palabras y con la frecuencia de aparición de cada una.
Para terminar debemos tener una implementación de una función que libere la memoria de una lista, para ello se propone el siguiente algoritmo iterativo:

Algorithm 3 Libera la memoria de una lista dada.

```
1:  procedure FreeList(root)
2:      if root then
3:          nxt ← root → Next
4:          while nxt → Next do
5:              FreeNode(root)
6:              root ← nxt
7:              nxt ← nxt → Next
8:          if nxt then
9:              FreeNode(nxt)
10:         if root then
11:             FreeNode(root)
```

Donde la función $FreeNode$ libera la memoria de un nodo.

III. Pruebas realizadas

El algoritmo anterior se ejecutó en contra de varios archivos de texto, y se pudo comprobar que produjo una salida correcta, de acuerdo con el orden de las palabras y con la frecuencia de aparición de las mismas. A continuación se muestra un ejemplo de ejecución con un archivo de texto, el cual corresponde a un libro.

```
  1   1    "'Blue
  2   2      "'I
  3   1   "'Never
  4   1     "'Now
  5   1    "'Swim
  6   1  "'That's
  7   2    "'They
  8   1    "'This
  9   1    "'Why,
 10   3       "A
 11   2    "After
 12   1     "Ah,"
 13   3      "All
 14   1       "Am
 15  33      "And
 16   1   "Anyone
 17  13      "Are
 18   1   "Aren't
 19   1       "At
 20   1     "Aunt
 21   3       "Be
 22   7  "Because
 23   1 "Because,"
 24   1    "Bless
 25   1     "Blue
 26   2    "Bring
 27  28      "But
 28   3     "But,
 29   1    "But,"
 30   1       "By
 31   1     "Call
 32   2      "Can
 33   3    "Can't
 34   1    "Carry
 35   2 "Certainly,"
 36   1 "Certainly.
 37   1 "Certainly;
 38   1     "City
 39  11     "Come
 40   1    "Come,
 41   1 "Congratulat
 42   1     "Dear
 43   1 "Defects,"
 44   2      "Did
 45   1   "Didn't
 46   4       "Do
 47   1     "Does
 48   1  "Doesn't
 49   9    "Don't
 50   1   "Don't!
 51   1  "Drink."
 52   2   "Ep-pe,
 53   1     "Even
 54   1 "Everything
 55   1  "Exactly
 56   1      "Fly
 57   1   "Follow
 58   2      "For
```

(a) Figura 1. Resultado de la ejecución del programa.

En la **Figura 1.** podemos apreciar las primeras cincuenta y ocho palabras que imprimió el programa, las cuales están en orden. La segunda columna de la imagen representa la frecuencia de aparición de cada palabra.

Cabe mencionar que el programa se corrió bajo la ejecución del software $valgrind$ y como resultado, no presentó fugas de memoria, o algo por el estilo.

IV. Ejercicios propuestos en la tarea

En esta sección se analizarán algunos algoritmos de caracter recursivo, los cuales fueron propuestos en la definición de la tarea, de la igual forma se brindará el pseudocódigo de los mismos.

IV-A. Ejercicio 1. Suma de números impares

Para este ejercicio se propone un algoritmo que dado un parámetro n, se debe generar la suma de todos los números positivos e impares menores o iguales a n.

Algorithm 4 Suma de números impares.

```
1: procedure GETODDNUMBERS(n)
2:     if n = 1 then
3:         return 1
4:     if n es par then
5:         return GetOddNumbers(n − 1)
6:     else
7:         return n + GetOddNumbers(n − 2)
```

Para el algoritmo anterior, se asume que el caso más pequeño es cuando n es igual a 1, el cual regresa 1. Los casos recursivos son, cuando n es par, lo cual no nos sirve, por lo que llamamos a la función con un $n-1$, y cuando n es impar, para este caso debemos calcular la suma en un estado anterior, refiriéndonos a $n-2$ y a eso le sumamos n ya que sabemos que dicho número es impar.

IV-B. Ejercicio 2. Generación de promedio.

Para este ejercicio se propone un algoritmo que dado un parámetro n se calcula el promedio de todos los números enteros, empezando en 1 y terminando en n.

Algorithm 5 Promedio de números enteros.

```
1: procedure GETAVR(n)
2:     if n = 1 then
3:         return 1
4:     else
5:         return GetAvr(n − 1) + 0,5
```

Se puede demostrar fácilmente que teniendo el promedio hasta $n-1$ podemos sumar $0{,}5$ al mismo y obtener el promedio hasta n.

IV-C. Ejercicio 3. Método de Newton-Raphson

Para este ejercicio se implementó el algoritmo de Newton-Raphson de la siguiente manera: Se da un x_0, un valor de tolerancia y un número máximo de iteraciones. El algoritmo debe encontrar la raíz que cumpla la condición de ser menor a la tolerancia, y debe hacer esto con un número de iteraciones menor al propuesto.

Algorithm 6 Método de Newton-Raphson.

```
1: procedure NEWTON(x, eps, iter)
2:     fx ← f(x)
3:     if iter = 0 then
4:         Imprime No se encontró la raíz.
5:         Termina
6:     if abs(fx) < eps then
7:         return x
8:     x ← x − fx/fp(x, eps)
9:     return Newton(x, eps, iter − 1)
```

En el **Algoritmo 6.** se puede apreciar que recursivamente se calcula el nuevo valor de x_i hasta encontrar uno que se acerque mucho a la raíz.

V. Conclusion

En el presente reporte se pudo apreciar la implementación de una versión particular de las listas ligadas, nos podemos dar cuenta que la función que añade un nuevo nodo a la lista tiene una complejidad $O(n)$, igual a la de una implementación normal, o incluso hay casos en los que puede ser más rápido, debido a que encuentra la posición donde añadirá el nuevo nodo antes de llegar al final de la lista. Aunque esta no es la implementación más eficiente, se pueden usar estructuras como árboles o diccionarios que reducen el tiempo de complejidad del algoritmo. Incluso si se conoce el número de palabras que se tendrán, se puede hacer un algoritmo más austero con un arreglo de estructuras, el cual añade todas las palabras en tiempo $O(n)$ y se ordena en tiempo $O(nlog(n))$.