Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructuras de Datos y Algoritmos

Informe Tarea 0

9 de abril de 2021

1º semestre 2021 - Profesores C. Gazali - Y. Eterovic

Vicente Espinosa - 17639247

Análisis complejidad

A continuación se mostrará un pseudo código de lo que se ejecuta en cada uno de los casos posibles, junto con una pequeña explicación de por que se le da cierta complejidad.

Estas son unas funciones usadas en distintos casos, se describen antes para disminuir el tamaño del documento y para facilitar su revisión.

Funciones usadas frecuentemente:

−buscarNodoConRuta:

Tomar caso 0 de la región correspondiente.

$for\ \_\ in\ range(depth)$

Recibir *id_contact* del input

Buscar al hijo del nodo con $id == id\_contact$

Cambiar a este nuevo nodo

−destroyTree:

Recibir al nodo inicial de la destrucción

if nodo actual− >hijo ! = $NULL$

$detroyTree$(hijo)

$else$

if nodo actual ! = nodo de primer $detroyTree()$

if nodo actual $- > next == NULL$

liberar nodo actual

$detroyTree(padre)$

$else$

liberar nodo actual

$detroyTree(next)$

$else$

liberar nodo

Se repite desde la segunda linea hasta llegar a el nodo del primer $detroyTree()$

• **ADD_CONTACTS** $\mathcal{O}(n)$

PSEUDO CÓDIGO

nodo = $buscarNodoConRuta(ruta)$

Crear los hijos para el nodo en el que se encuentre en este momento.

EXPLICACIÓN COMPLEJIDAD

La complejidad de este caso es $\mathcal{O}(n)$, porque el crear hijos para el nodo padre solo depende de la cantidad de hijos nuevos, por lo tanto la complejidad vendrá de la función $buscarNodoConRuta$. Esta función tiene complejidad n por que en el peor caso, cada nodo en la ruta tendrá un solo hijo, y se pedirá el ultimo nodo, en este caso habría que pasar por cada nodo que hay una vez, y por lo tanto, el tiempo que tomará será $\mathcal{O}(n)$.

• **RECOVERED** $\mathcal{O}(n)$

PSEUDO CÓDIGO

nodo actual = $buscarNodoConRuta(ruta)$

Cambiar estado del nodo actual a recuperado.

EXPLICACIÓN COMPLEJIDAD

Al igual que ADD_CONTACTS, lo único que depende de la cantidad de datos es $buscarNodoConRuta$, y por lo tanto, tendrá la misma complejidad $\mathcal{O}(n)$

• **POSITIVE** $\mathcal{O}(n)$

PSEUDO CÓDIGO

nodo actual = $buscarNodoConRuta(ruta)$

Recorrer los hijos del nodo actual y cambiarles su estado a sospechoso. Se comienza por *head* y se sigue hacia el *next* del nodo, hasta que *next* sea $NULL$.

EXPLICACIÓN COMPLEJIDAD

Mismo caso que los dos anteriores, la complejidad de cambiar el estado de los hijos depende unicamente de la cantidad de hijos, y por lo tanto, la complejidad viene de $buscarNodoConRuta$, o sea, es $\mathcal{O}(n)$

• **NEGATIVE** $\mathcal{O}(n)$

PSEUDO CÓDIGO

nodo = $buscarNodoConRuta(ruta)$

$detroyTree$(nodo)

EXPLICACIÓN COMPLEJIDAD

Ya sabemos que $buscarNodoConRuta$ tiene complejidad $\mathcal{O}(n)$. Ahora, $detroyTree$ debe recorrer algunos nodos mas de una vez, ya que para eliminar un nodo este no debe tener

hijos restantes, entonces en general cada nodo padre se recorre dos veces, y cada nodo sin hijos se recorre una. Esta complejidad, a pesar de ser mayor, sigue siendo $\mathcal{O}(n)$, y por lo tanto, la complejidad de la función final también será $\mathcal{O}(n)$.

• **CORRECT** $\mathcal{O}(n)$

PSEUDO CÓDIGO

nodo1 = $buscarNodoConRuta(ruta1)$

nodo2 = $buscarNodoConRuta(ruta2)$

A cada hijo de $nodo1$ cambiarle la propiedad $parent$ a $nodo2$

A cada hijo de $nodo2$ cambiarle la propiedad $parent$ a $nodo1$

Intercambiar propiedades $head$ y $tail$ entre $nodo1$ y $nodo2$

EXPLICACIÓN COMPLEJIDAD

El cambiarle las propiedades a el $nodo1$ y $nodo2$ toma un tiempo constante, al igual que cambiarle los padres a los hijos de estos nodos, por lo tanto no son relevantes para el calculo de la complejidad, pues sabemos que $buscarNodoConRuta$ tiene complejidad $\mathcal{O}(n)$, y aunque se ejecuta dos veces, la complejidad se mantiene como $\mathcal{O}(n)$.

• **INFORM** $\mathcal{O}(n)$

PSEUDO CÓDIGO

Se usa un algoritmo similar a $destroyTree$ (depth first search) pero en vez de liberar nodos, se printean en el $output\_file$

EXPLICACIÓN COMPLEJIDAD

Dado que lo único que se hace es recorrer todo los nodos con un algoritmo de DFS, se pasa una vez por cada nodo en la región solicitada. Dado esto, la complejidad será $\mathcal{O}(n)$.

• **STATISTICS** $\mathcal{O}(n)$

PSEUDO CÓDIGO

Se usa un algoritmo similar a *destroyTree* (depth first search) pero en vez de liberar los nodos, se va contando cuantos nodos tienen X tipo de estado.

EXPLICACIÓN COMPLEJIDAD

La complejidad de STATISTICS es igual a la de INFORM, porque la única diferencia entre estos dos es que un cuenta los estados de los nodos, mientras que el otro printea el id junto al estado de cada nodo. O sea, es $\mathcal{O}(n)$.

• **Búsqueda Caso 0** $\mathcal{O}(1)$

La búsqueda de un caso 0 tiene una complejidad constante, ya que siempre se puede encontrar como un atributo de *world*, y por lo tanto, no depende de la cantidad de datos que hay en el problema.

Arreglo vs lista ligada

Si los contactos de una persona se ordenaran como arreglo, y se tuviera conocimiento de su índice con solo conocer el id, se facilitaría la búsqueda de personas, ya que no sería necesario recorrer los hijos de cada persona hasta encontrar al que tiene el id que se busca.

Ahora, con la eliminación y la inserción no sería un beneficio, por que a pesar de que se encontraría más rápido a la persona, no se puede borrar tan eficientemente, pues luego de eliminar un elemento del arreglo, se deben mover todos los elementos que están con índice superior al eliminado un espacio hacia la izquierda, lo que en la mayoría de los casos sería un problema grave. Con la inserción existe un problema similar, pero esta vez en vez de mover los nodos con índice mayor a la izquierda estos se mueven a la derecha.

Por lo tanto, el arreglo seria superior para analizar los datos (buscar), pero la lista ligada es mucho mejor para crear la estructura (eliminar e insertar).

C vs Python

Test	Python	C	Diferencia
1	3.533s	1.993s	1.540s
2	5.701s	3.912s	1.789s
3	7.437s	5.803s	1.634s
4	11.098s	7.350s	3.748s
5	12.580	9.300s	3.280s
6	15.381s	9.772s	5.609s

Claramente C tiene una mejor velocidad de ejecución, la diferencia va aumentando a medida que aumenta el tiempo total, por lo que C es una opción completamente superior a Python para proyectos que manejen grandes cantidades de datos.