Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2133 – Estructuras de Datos y Algoritmos – 2021-1
Estudiante: Diego Bustamante

Tarea 0

Pregunta 1

Sea n la cantidad de personas del árbol de la aplicación.

i) Buscar al paciente 0 se utiliza en los demás algoritmos, se calculará primero.

Problema: Búsqueda del paciente 0
Input: World, Id País e Id Región
Output: Paciente 0
Complejidad: $O(1)$
Pseudocódigo:

```
person = world -> countries[country][region];
return person;
```

Donde "countries" es un arreglo de arreglos de pacientes 0 y los id de los países y regiones son los índices de los arreglos. Entonces, como buscar 2 índices es constante e independiende de la cantidad de personas, este algoritmo es $O(1)$.

ii) También se justificará la complejidad del algoritmo de búsqueda de una persona dado que se utilizará en los eventos.

Problema: SEARCH
Input: World, Id País, Id Región, Ruta de la Persona
Output: Persona buscada
Complejidad: $O(n)$
Pseudocódigo:

```
person = busqueda_del_paciente_0(params);
for (int idx = 0; idx < rute_size; idx++)
{
  current = person -> head;
  while (current)
  {
      if (current -> _id == rute[idx])
      {person = current; current = NULL;}
      else
    {current = current -> _next;}
  }
}
return person;
```

Son 2 loops. Los peores casos individuales son: en el primero, que todo el árbol sea una lista ligada y se tengan que recorrer todas las personas para llegar a la que buscamos ya que busca la profundidad. El segundo, que el raíz tenga todas las demás personas de hijos y el buscado es tail ya que busca por amplitud. Sin embargo, estas condiciones no se pueden cumplir al mismo tiempo. Entre las dos deben sumar n personas y la combinación que da el mayor producto entre los 2 loops son cada uno $\sqrt{n}$. Por lo tanto, el algoritmo es de complejidad $O(n)$.

iii) El primer evento es añadir contactos.

Problema: ADD_CONTACTS
Input: World, Id País, Id Región, Ruta de la Persona y Número de contactos a agregar.
Output: Vacío
Complejidad: $O(n)$
Pseudocódigo:

```
persona = buscar_persona(params);
for (int i = 0; i < n_contactos; i++)
{
  add_contact(person);
}
```

Se debe sumar la complejidad de buscar a una persona, que sabemos que es $O(n)$, más la complejidad de añadir personas. Dicha complejidad sería $O(m)$, donde m es la cantidad de contactos a agregar porque "add_contact" simplemente añade un contacto al final de la lista ligada lo que es $O(1)$. Sin embargo, se asume $m <<< n$ en la aplicación. Y eso justifica que la complejidad sea $O(n) + O(m) = O(n)$.

iv) El segundo evento es actualizar un contacto como recuperado.

Problema: RECOVERED
Input: World, Id País, Id Región y Ruta de la Persona.
Output: Vacío
Complejidad: $O(n)$
Pseudocódigo:

```
persona = buscar_persona(params);
state_update(person, recuperado);
```

Se debe buscar a una persona y de actualizar un contacto. La complejidad de la última parte sería $O(1)$ puesto que es simplemente cambiar un atributo (no depende de n). Por lo tanto, la complejidad es $O(n)$.

v) El tercer evento es actualizar un contacto como recuperado.

Problema: POSITIVE
Input: World, Id País, Id Región y Ruta de la Persona.
Output: Vacío
Complejidad: $O(n)$
Pseudocódigo:

```
Person* person = search(world, country, region, depth, route);
state_update(person, positive);
Person* current = person -> head;
while (current)
{
```

```
    state_update(current, take_test);
    current = current -> _next;
  }
```

Se debe que buscar una persona y actualizar a todos sus contactos. Dicha complejidad sería $O(n)$ más el peor caso de actualizar los contactos estrechos, este es, todas las personas del árbol ($O(n)$). Y, por aritmética de complejidades, $O(n) + O(n) = O(n)$.

vi) El cuarto evento es actualizar un contacto como negativo.

Problema: NEGATIVE
Input: World, Id País, Id Región y Ruta de la Persona.
Output: Vacío
Complejidad: $O(n)$
Pseudocódigo:

```
person = search(world, country, region, depth, route);
parent = person -> parent;
prev = person -> prev;
next = person -> _next;
arreglar_lista_ligada(person, parent, prev, next);
recursive_destroy(person);
```

De nuevo, aparece el término $O(n)$ por buscar una persona y muchas otras constantes por arreglar la lista ligada (siempre interconectar 4 personas). Luego, "recursive_destroy" es:

```
current = person -> head;
while (current)
{
  next = current -> _next;
  recursive_destroy(current);
  current = next;
}
person_destroy(person);
```

Es una función que recorre todos los hijos de una persona. En el peor caso la persona es la raíz y se deben recorrer todas las personas para eliminarlas. Esto es $O(n)$, entonces, todo el algoritmo NEGATIVE es $O(n)$.

vii) El quinto evento es intercambiar los hijos de 2 personas.

Problema: CORRECT
Input: World, Id País, Id Región, Ruta de la Persona 1 y Ruta de la Persona 2.
Output: Vacío
Complejidad: $O(n)$
Pseudocódigo:

```
person_one = search(world, country, region, depth_one, route_one);
person_two = search(world, country, region, depth_two, route_two);
head_one = person_one -> head;
tail_one = person_one -> tail;
person_one -> head = person_two -> head;
person_one -> tail = person_two -> tail;
current = person_one -> head;
while (current)
```

```
{
  current -> parent = person_one;
  state_update(current, take_test);
  current = current -> _next;
}
person_two -> head = head_one;
person_two -> tail = tail_one;
current = person_two -> head;
while (current)
{
  current -> parent = person_two;
  state_update(current, take_test);
  current = current -> _next;
}
```

En este caso es similar a uno anterior. Se tienen 2 búsquedas de personas y 2 recorridos de contactos, así se obtiene una complejidad de $4 * O(n) = O(n)$.

viii) El sexto evento es informar una representación del árbol.

Problema: INFORM
Input: World, Id País y Id Región.
Output: Vacío
Complejidad: $O(n)$
Pseudocódigo:

```
person = busqueda_del_paciente_0(params);
recursive_inform(patient_zero, 0, output_file);
```

Donde "recursive_inform" es:

```
fprintf(output_file, content);
current = person -> head;
while (current)
{
  recursive_inform(current, depth + 1, output_file);
  current = current -> _next;
}
```

La búsqueda es $O(n)$ y "recursive_inform" recorre todos hijos de una persona recursivamente. Como se utiliza sobre el paciente 0, se recorre todo el árbol. Entonces, también es $O(n)$. Por lo tanto, todo el algoritmo es $O(n)$.

ix) El séptimo evento es informar las estadísticas de un árbol.

Problema: STATISTICS
Input: World, Id País y Id Región.
Output: Vacío
Complejidad: $O(n)$
Pseudocódigo:

```
person = busqueda_del_paciente_0(params);
recursive_statistics(patient_zero, stats);
fprintf(output_file, stats);
```

Donde “recursive_statistics” es:

```
update_stats(person -> state, stats);
current = person -> head;
while (current)
{
  recursive_statistics(current, stats);
  current = current -> _next;
}
```

La búsqueda es $O(n)$ y “recursive_statistics” recorre todos hijos de una persona recursivamente. Como se utiliza sobre el paciente 0, se recorre todo el árbol. Además, “update_stats”, es un switch por lo que su complejidad es $O(1)$. Por lo tanto, todo el algoritmo es $O(n)$.

Pregunta 2

Para analizar la búsqueda, inserción y eliminación de personas se utilizarán los algoritmos: SEARCH, ADD_CONTACTS y NEGATIVE.

En el caso de SEARCH analizamos 2 loops. Uno que busca la profundidad y el otro que busca los contactos en amplitud y en el peor caso recorre todos los contactos. Al poder acceder por índice a los contactos de las personas, gracias a conocer el ID, uno de los peores casos deja de ser tal.

Si la raíz posee a todos los contactos estrechos ya no hay que recorrer en amplitud toda la lista ligada para llegar al último. Basta con ir directamente al último índice ya que se conoce el ID. Sin embargo, el caso de que se tengan que recorrer todos los contactos en profundidad sigue siendo igual. Por lo tanto, podemos deducir que la búsqueda en el peor caso continúa siendo $O(n)$ pero en el promedio el tiempo sería un factor menor gracias a acceder por índice.

En el caso de ADD_CONTACTS se debe reanalizar la función “add_contact” pues ya no es una lista ligada sino un arreglo. Para insertar un elemento nuevo en un arreglo de tamaño fijo no existe otra solución que volver a crear un nuevo arreglo recorriendo el anterior y luego añadir los nuevos contactos.

Si bien lo anterior es muy costoso, sigue sin depender de n dado que se asumió que la cantidad de personas agregadas o contactos de una persona es mucho menor que n. Por lo tanto, la complejidad de insertar personas continúa siendo $O(n)$ pero multiplicado por un factor de más costo computacional.

Finalmente, el caso de NEGATIVE es similar al anterior. Si se elimina una persona del arreglo y se deja un hueco, esto deberá ser parchado al insertar recorriendo 2 veces el arreglo. Como deben haber más inserciones que eliminaciones (si no no habría aplicación) es mejor rehacer el arreglo de contactos estrechos al eliminar una persona.

Entonces, “arreglar_lista_ligada” se transformaría en “arreglar_arreglo” y sería de complejidad $O(m)$, donde m es la cantidad de contactos y $m <<< n$. Además, “recursive_destroy” mantendría su complejidad puesto que continúa recorriendo a todas las personas. Así, eliminar a alguien sería complejidad de “search” sumado a la de “arreglar_arreglo” y “recursive_destroy”, esto es, $O(n)$.

En conclusión, dependiendo de qué instrucciones se ocupen más en la aplicación es mejor una estructura de datos. Por ejemplo, si lo principal es buscar personas y producir datos, entonces lo mejor son arreglos.

Sin embargo, si lo principal es insertar y eliminar personas, entonces lo mejor son listas ligadas.

Pregunta 3

Se construyó una tabla para los diferentes tests ejecutados por los programas escritos en Python y C. La segunda columna y la tercera representan el tiempo promedio en segundos de 3 ejecuciones del test por el respectivo programa.

Tabla 1: Promedio de ejecución de Python y C para 6 tests

Test	Python (s)	C (s)
1	1,653	0,475
2	3,320	0,975
3	4,795	1,292
4	6,522	1,892
5	7,823	2,346
6	9,397	2,654

Como se puede apreciar en la Tabla 1, estos tiempos comprueban lo calculado en las partes anteriores. La cantidad de lineas de cada test aumenta linealmente y a cantidad de personas por árbol también (aproximadamente). Y los tiempos de ejecución también aumenten linealmente.

Se nota claramente una diferencia en tiempo de ejecución entre ambos programas. Esto proviene del uso y manejo del lenguaje utilizado. Si bien C presenta desventajas en cuanto a dificultad y programación orientada a objetos, esto permite ventajas en otros aspectos.

Python es interpretado y de alto nivel. La traducción en ejecución de un lenguaje más parecido al natural genera una demora en la ejecución de cada instrucción. En contraste, C es compilado y de bajo nivel, las instrucciones se mapean de forma mas directa al lenguaje de máquina y el compilador se encarga de optimizar mejor un programa.

Existe un último factor que influye indirectamente de una forma bien particular. Los objetos de Python usan mucha más memoria que los structs de C debido a la herencia y su fácil manejo en cuanto a código. Esto perjudica considerablemente el tiempo de ejecución si es que el test crea demasiadas personas. El sistema operativo se verá forzado a hacer swapping entre la ram y el disco duro del computador lo que impacta en el tiempo real de ejecución de un proceso.