***PREGUNTAS PREPARATORIAS***

**a) Teniendo en cuenta cada uno de los requerimientos ¿Cuántos índices implementaría en el Reto? y ¿Por qué?**

R/ Usaría dos índices, uno para los artistas y otro para las obras.

**b) Según los índices propuestos ¿en qué caso usaría Linear Probing o Separate Chaining en estos índices? y ¿Por qué?**

R/ Usaría en ambas *linear probing* ya que se me facilitara más usarlas ya que no tengo la restricción de solo tener un bucket por posición que tenga al tener un conflicto, solo se buscara la siguiente posición inmediatamente vacía.

**c) Dado el número de elementos de los archivos MoMA, ¿Cuál sería el factor de carga para estos índices según su mecanismo de colisión?**

R/ Teniendo el posible total de datos que nos dé MoMA, se toma el primo más cercano al doble del tamaño de los datos y usar un factor de carga de 0.5.

**d) ¿Qué diferencias en el tiempo de ejecución notan al ejecutar la cargar los datos al cambiar la configuración de Linear Probing a Separate Chaining?**

R/ Toma menos tiempo de ejecución a la hora de usar una tabla con *linear probing.*

1. **e) ¿Qué configuración de ADT Map escogería para el índice de técnicas o medios?, especifique el mecanismo de colisión, el factor de carga y el numero inicial de elementos.**

R/ El mecanismo de colisión que se usaría aquí sería *linear probing*, ya que su uso es más sencillo, como se explicó antes, el factor de carga sería de 0.5, y el número inicial de elementos sería el primo más cercano al doble del tamaño de los datos por lo estudiado en clase.

1. **f) ¿Qué configuración de ADT Map escogería para el índice de nacionalidades?, especifique el mecanismo de colisión, el factor de carga y el numero inicial de elementos.**

R/ El mecanismo de colisión aquí también sería *linear probing* por las razones antes dadas, el factor de carga sería el mismo (de 0.5), y el número inicial de elementos (de nuevo) sería el primo más cercano al doble del tamaño de los datos por lo estudiado en clase.

**Documento de análisis (Reto 2)**

*José Daniel Montero Gutiérrez:*

*código 202012732,* [*j.monterog@uniandes*](mailto:j.monterog@uniandes)*.edu.co*

*Wyo Hann Chu Mendez:*

*Código 202015066,* [*w.chu@uniandes*](mailto:w.chu@uniandes)*.edu.co*



|  |  |  |
| --- | --- | --- |
|  | **Máquina 1** | **Máquina 2** |
| **Procesador** | AMD Ryzen 5 2600X Six-Core Processor 3.60 GHz |  |
| **Memoria RAM (en GB)** | 8,00 GB |  |
| **Sistema operativo** | Windows 10 Pro 64-Bits |  |

**Análisis de complejidad**

**Requerimiento 2**

|  |
| --- |
| Primero se organizan los elementos dentro de la lista de *artworks* a través de un *Shell sort* para optimizar el tiempo de ejecución de la función. Por esto, tomaremos su complejidad temporal promedio **O(n^1.25)**. |
| *def* addAcquireDate(*aDList*, *date*, *piece*):      posAcquireDate = lt.isPresent(aDList, date)      if posAcquireDate > 0:          acquireDate = lt.getElement(aDList, posAcquireDate)      else:          acquireDate = newAcquireDate(date)          lt.addLast(aDList, acquireDate)      lt.addLast(acquireDate[‘pieces’], piece[‘Title’]) |
| \* La función newAcquireDate() solo crea una nueva lista, así que no influye en el tiempo de ejecución de manera significativa. |
| Este algoritmo toma **m + 1 + 1 + 1 + 1 + 1 = m + 5**  Que se puede reescribir como: **m**  El orden de crecimiento temporal de este algoritmo es aproximadamente **~O(m)** |
| *def* listByAcquireDate(*catalog*, *startDate*, *endDate*):      artworkList = lt.newList(‘ARRAY\_LIST’)      byDatePurchase = lt.newList(‘ARRAY\_LIST’, *cmpfunction*=compareAD)      for artwork in lt.iterator(catalog[‘artworks’]):          if (artwork[‘DateAcquired’][0:3] >= startDate[0:3]) and (artwork[‘DateAcquired’][0:3] <= endDate[0:3]):              if (artwork[‘DateAcquired’][5:6] >= startDate[5:6]) and (artwork[‘DateAcquired’][5:6] <= endDate[5:6]):                  if (artwork[‘DateAcquired’][8:9] >= startDate[8:9]) and  (artwork[‘DateAcquired’][8:9] <= endDate[8:9]):                      lt.addLast(artworkList, artwork)                      addAcquireDate(byDatePurchase, artwork[‘DateAcquired’], artwork)      totalAmmount = lt.size(artworkList)      return totalAmmount, artworkList, byDatePurchase |
| Este algoritmo toma **1 + 1 + n \* 1 \* 1 \* 1 \* 1 \* 1 \* m + 1 = 3 + n \* m**  Que se puede reescribir como: **n \* m**  El orden de crecimiento temporal de este algoritmo es aproximadamente **~O(n\*m)** |

**Requerimiento 4**

|  |
| --- |
| *def* topTenCountries(*catalog*):      n = 1      while n <= 10:          nationality = lt.getElement(catalog['nationalitiesList'], n)          print('La nacionalidad', nationality['nationality'], 'tiene un total de',  *str*(lt.size(nationality['artworks'])), 'obras.')          n += 1 |
| Este algoritmo toma **1 + 10 + 1 + 1 + 1**  Que se puede reescribir como: **14**  El orden de crecimiento temporal de este algoritmo es aproximadamente **~O(1)** |
| *def* topCountryArtworks(*catalog*):      nationality = lt.getElement(catalog['nationalitiesList'], 1)      fThreePiecesRq4(nationality['artworks'])      lThreePiecesRq4(nationality['artworks']) |
| \* Las dos funciones fThreePiecesRq4 y lThreePiecesRq4 tienen complejidad temporal aproximadamente de **O(1)**, debido a que solo se dedican a imprimir información del catálogo.  Este algoritmo toma **1 + 1 + 1**  Que se puede reescribir como: **3**  El orden de crecimiento temporal de este algoritmo es aproximadamente **~O(1)** |

**Pruebas de tiempo de ejecución:**

**Requerimiento 2:**

El rango de tiempo usado durante las pruebas fue de 1985-10-18 a 1990-10-18.

|  |  |
| --- | --- |
| **Porcentaje de la muestra [pct]** | **Tiempo de ejecución (en ms)** |
| **small** | 30640.625 |
| **5.00%** | --- |
| **10.00%** | --- |
| **20.00%** | --- |
| **30.00%** | --- |
| **50.00%** | --- |
| **80.00%** | --- |
| **large** | --- |

Intentar ejecutar la función del requerimiento 2 resultó imposible, debido a que al intentar usarla con los archivos de 5% para arriba se demoraba más de 15 minutos en ejecutar.

**Requerimiento 4:**

|  |  |
| --- | --- |
| **Porcentaje de la muestra [pct]** | **Tiempo de ejecución (en ms)** |
| **small** | 0.0\* |
| **5.00%** | 15.625 |
| **10.00%** | 15.625 |
| **20.00%** | 15.625 |
| **30.00%** | 15.625 |
| **50.00%** | 15.625 |
| **80.00%** | 31.250 |
| **large** | 31.250 |

\* Se puede asumir que se demoró tan poco tiempo en ejecutar, que Visual Studio despreció el tiempo, y por eso muestra que se demoró 0.0 ms.