Reflexión: Actividad integradora #3

Miguel Arriaga Velasco – A01028570

Campus Santa Fe

Programación de estructuras de datos y algoritmos fundamentales

10 de mayo del 2021

1. Árboles binarios

Los árboles binarios son estructuras de datos utilizadas en la programación. El más básico es el "Binary search tree" y se define la siguiente manera:

- Estructura de árbol basada en nodos, cuenta con las siguientes propiedades:
    - El sub-árbol de la izquierda contiene valores menores que el nodo actual
    - El sub-árbol de la derecha contiene valores mayores que el nodo actual
    - Los sub-árboles de la izquierda y de la derecha también deben de ser binary search trees.

Algunas de las variantes de árboles binarios son las siguientes:

- Árbol AVL
- Heap
- Árbol SPLAY

Para esta actividad integradora, implementaremos un Heap para almacenar y ordenar los datos de la bitácora.

2. Heap

Un Heap es un árbol binario que cumple con las siguientes reglas:

- Todo padre del árbol contiene un elemento que tiene un valor de mayor prioridad que los valores de sus nodos hijos.
    - La prioridad está determinada por la aplicación. Ej. a mayor valor, mayor prioridad.
- El árbol está completamente balanceado, todos los niveles del árbol tienen el máximo de nodos posible, excepto el nivel inferior que puede estar incompleto.
- Los nodos hojas del nivel inferior están lo más a la izquierda posible.
- Los nodos hojas del nivel inferior están lo más a la izquierda posible.

Algunas de sus aplicaciones son las siguientes:

- Implementación de colas priorizadas:
    - En nivel físico del ADT Cola priorizada es un Heap.
    - Por lo tanto, muchas de las aplicaciones de Grafos, se pueden implementar óptimamente con Heaps.
- Implementación del método Heap Sort para ordenar información. Éste será el método de ordenamiento de nuestra bitácora.

Podemos distinguir dos tipos de Heap:

- Max Heap: mayor valor = mayor prioridad
- Min Heap: menor valor = mayor prioridad

3. Heap Sort

Según GeeksForGeeks, 2021: "Heap Sort es una técnica de ordenamiento basada basada en la estructura de datos de un Binary Heap. Es similar al ordenamiento por selección donde primero encontramos el elemento mínimo y colocamos el elemento mínimo al principio. Repetimos el mismo proceso para el resto de los elementos.

3.1. Análisis de complejidad

Complejidad espacial:

Heap sort realiza un ordenamiento “in-place”, ya que utiliza la misma estructura de datos que guardaba a los datos sin ordenar, para guardar los datos ordenados. Es por esto que su complejidad espacial es constante: O(1).

Complejidad temporal:

Para analizar la complejidad temporal de nuestro algorítmo, primero debemos de analizar la complejidad que nos toma el método “heapifydel” (figura 1), que se llama dentro del método “heapSort”.

```
void BitacoraHeap::heapifydel(int n, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && vect[l].getIpNoPort() > vect[largest].getIpNoPort())
        largest = l;
    if (r < n && vect[r].getIpNoPort() > vect[largest].getIpNoPort())
        largest = r;
    if (largest != i) {
        swap(i,largest);
        heapifydel(n,largest);
    }
}
```

Figura 1. Algoritmo “heapifydel”

Este algorítmo lo podemos analizar utilizando las técnicas de análisis empírico de algoritmos recursivos. Se puede observar que solo hay una llamada recursiva, y el parámetro de control “largest”, se multiplica por un valor constante (2), en cada llamada. Esto nos dice que la complejidad de este método es de:

$$O(n) = (log_2 n)$$

Ahora observemos el algorítmo heapSort (figura 2).

```
void BitacoraHeap::heapSort()
{
    // One by one extract an element from heap
    for (int i = size - 1; i >= 0; i--) {
        // Move current root to end
        swap(0, i);
        // call max heapify on the reduced heap
        heapifydel(i,0);
    }
}
```

Figura 2. Algoritmo “heapifysort”

Se puede observar que hay un loop que va desde cero, hasta el tamaño del heap menos uno. Observemos que la operación más costosa dentro del heap es “heapfydel”, ya que “swap” corre en tiempo constante. Ya sabemos que el costo de “heapifydel” es de:

$$O(n) = (log_2 n)$$

Esta operación se repite n-1 veces, quitamos las constantes y obtenemos una complejidad temporal final de:

$$O(n) = (nlog_2 n)$$

Esta complejidad aplica para los peores, mejores y casos promedios. Además, es la misma que la de los algoritmos QuickSort y MergeSort, que se utilizaron en entregas pasadas.

Conclusión y reflexión sobre BST en situación problema

La situación problema de este curso es sobre los ataques cibernéticos y la identificación de páginas infectadas. En esta actividad, se logró manipular la bitácora dependiendo de las direcciones IP a través de el uso de heaps. Los heaps son una de las estructuras de datos más utilizadas, ya que almacena los datos según su prioridad. Esto tiene grandes aplicaciones a la hora de detectar ataques cibernéticos, ya que una IP con un gran número de accesos, podría mantenerse con una alta prioridad y estar bajo la mira como “sospechosa”. Ya que hay millones de accesos diarios, sería muy tardado buscar entre todos cuáles podrían ser ataques, es aquí en donde la naturaleza del heap entra en juego. Un alto número de accesos desde una misma IP, accesos consecutivos en un espacio de tiempo reducido y mensajes de error iguales todas son indicaciones de que se está intentando un ataque. Analizar este gigante número de información es una tarea difícil, consecuentemente, es nuestro trabajo como ingenieros de software interpretar la data a través de algoritmos y estructuras de datos.

Referencias

Presentaciones por el Dr. Eduardo Arturo Rodríguez Tello, durante la materia de Programación de estructuras de datos y algoritmos fundamentales

GeeksForGeeks (2021) “Heap Sort”. Recuperado desde: https://www.geeksforgeeks.org/heap-sort/