Universidad Centroamericana “José Simeón Cañas”
Análisis de Algoritmos Ciclo 02/2019
Semana 5 - 9 al 13 de septiembre del 2019

Guía de laboratorio 4

QuickSort
Heapsort

Quicksort

Conocido como método rápido y de ordenación por partición.

Aplicaciones:

Uso comercial de quicksort

- generalmente corre rápido
- no se requiere memoria adicional

Vida crítica:

1. Monitoreo médico
2. Soporte vital en aeronaves y naves espaciales)

Misión crítica :

1. Monitorización y control en plantas industriales y de investigación que manejan material peligroso
2. Control para aeronaves.
3. Defensa.

El algoritmo consiste en:

1. Tomar un elemento X de una posición cualquiera del arreglo.

2. Ubicar a X en la posición correcta del arreglo, de tal forma que todos los elementos que se encuentren a su izquierda sean menores o iguales a X y todos los que se encuentren a su derecha sean mayores o iguales a X.

3. Se repiten los pasos anteriores , pero ahora para los conjuntos de datos que se encuentran a la izquierda y a la derecha de la posición X en el arreglo.

4. El proceso termina cuando todos los elementos se encuentra en su posición correcta en el arreglo.

Pseudocódigo:

```
Quicksort( A, P, r )
        if ( p < r )
                q = partition ( A, P, r )
                Quicksort ( A, P, q - 1 )
                Quicksort ( A, q + 1, r )



Partition( A, P, r )
        x = A [ r ]
        i = p - 1
        for j ∈ { p, . . . , r - 1} {
                if ( A [ j ] ≤ x ){
                        i = i + 1
                        temp = A [ i ]
                        A [ i ] = A [ j ]
                        A [ j ] = temp
                }
        }

        temp = A [ i + 1 ]
        A [ i +1 ] = A [ r ]
        A [ r ] = temp
        return i + 1
```

quicksort.cpp

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;

// Una función para cambiar dos elementos
void swap(int* a, int* b)
{
```

```
        int t = *a;
        *a = *b;
        *b = t;
}

/* Esta función toma el último elemento como pivote, lo coloca en su posición
correcta en el arreglo y posiciona todos los elementos menores a la izquierda y los
mayores a la derecha de el mismo*/
int partition (int arr[], int low, int high)
{
        int pivot = arr[high]; // pivote
        int i = (low - 1); // posición del primer elemento

        for (int j = low; j <= high - 1; j++)
        {
                // Si el el elemento actual es menor que el pivote
                if (arr[j] < pivot)
                {
                        i++; // incrementar el index del menor elemento
                        swap(&arr[i], &arr[j]);
                }
        }
        swap(&arr[i + 1], &arr[high]);
        return (i + 1);
}

/* La función principal que corre QuickSort
arr[] --> arreglo a ordenar,
low --> límite inicial,
high --> límite final */
void quickSort(int arr[], int low, int high)
{
        if (low < high)
        {
                /*pi es la partición del índice, arr[p] está en el lugar correcto */
                int pi = partition(arr, low, high);

                // Ordenando separadamente los elementos antes y después de la
                // partición
                quickSort(arr, low, pi - 1);
                quickSort(arr, pi + 1, high);
        }
```

```
}

/* Función para imprimir arreglo */
void printArray(int arr[], int size)
{
        int i;
        for (i = 0; i < size; i++)
                cout << arr[i] << " ";
        cout << endl;
}

// Código principal para utilizarlo
int main()
{
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = sizeof(arr) / sizeof(arr[0]);
        quickSort(arr, 0, n - 1);
        cout << "Sorted array: \n";
        printArray(arr, n);
        return 0;
}
```

HeapSort

También conocido como montículo, es uno de los métodos más eficientes de ordenación que trabajan con árboles.

Aplicaciones:

1. **Colas de prioridad:** Las colas de prioridad se pueden implementar de manera eficiente utilizando Binary Heap porque admite las operaciones insert(), delete() y extractmax(), disminución de clave() en tiempo O (logn). Binomoial Heap y Fibonacci Heap son variaciones de Binary Heap. Estas variaciones realizan la unión también en el tiempo O (logn), que es una operación O (n) en Binary Heap. Las colas de prioridad de montón implementado se utilizan en algoritmos de gráficos como el algoritmo de Prim y el algoritmo de Dijkstra.

2. **Estadísticas de pedidos:** la estructura de datos del montón se puede utilizar para encontrar eficientemente el k ésimo elemento más pequeño (o más grande) en una matriz. Vea los métodos 4 y 6 de esta publicación para más detalles.

La idea central del algoritmo se basa en dos operaciones:

1. Construir un montículo.

2. Eliminar la raíz del montículo en forma repetida.

Pseudocódigo:

```
Heapsort ( A )
        Build-heap ( A )
        for i ∈ { n . . . 2 }
                temp = A [ 1 ]
                A [ 1 ] = A [ i ]
                A [ i ] = temp
                heap-size -= 1
                max-heapify ( A, 1)

Build-heap ( A )
        heap-size = length ( A )
        for i ∈ { ⌊n/2⌋ . . . 1 }
                max-heapify ( A, i )

Max-heapify ( A, i )

        L = left ( i )
        r = right ( i )

        if ( L ≤ heap-size [ A ] && A [ L ] > A [ i ] )
                largest = L
        else
                largest = i

        if ( r ≤ heap-size [ A ] && A [ r ] > A [ largest ] )
                largest = r

        if ( largest != i )
                temp = A [ largest ]
```

```
            A [ largest ] = A [ i ]
            A [ i ] = temp
            max-heapify ( A, largest )
```

heapsort.cpp

```
// C++ program for implementation of Heap Sort
#include <iostream>
using namespace std;

// Crear un sub-arbol con raiz i que es
// un índice en arr[] siendo n el tamaño de la división que se tendrá
void heapify(int arr[], int n, int i)
{
        int largest = i; // Inicializar el más grande como la raíz
        int l = 2 * i + 1; // izquierda= 2*i + 1
        int r = 2 * i + 2; // derecha= 2*i + 2

        // Si el hijo izquierdo es más largo que la raíz
        if (l < n && arr[l] > arr[largest])
                largest = l;

        // Si el hijo derecho es más largo que el más largo total
        if (r < n && arr[r] > arr[largest])
                largest = r;

        // Si el mayor no es la raíz
        if (largest != i) {
                swap(arr[i], arr[largest]);

                // Recursivamente crear el sub-arbol y usar la función heapify en el
                heapify(arr, n, largest);
        }
}


void heapSort(int arr[], int n)
{
        // Construir la estructura de árbol (reorganizar el arreglo)
        for (int i = n / 2 - 1; i >= 0; i--)
                heapify(arr, n, i);
```

```
        // Extraer los elementos uno por uno de la estructura
        for (int i = n - 1; i >= 0; i--) {
                // Mover la raíz actual al final
                swap(arr[0], arr[i]);

                // llamar la función heapify en la estructura reducida
                heapify(arr, i, 0);
        }
}

/* Función para imprimir el arreglo */
void printArray(int arr[], int n)
{
        for (int i = 0; i < n; ++i)
                cout << arr[i] << " ";
        cout << "\n";
}

//  Código principal para utilizarlo
int main()
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int n = sizeof(arr) / sizeof(arr[0]);

        heapSort(arr, n);

        cout << "Sorted array is \n";
        printArray(arr, n);
}
```

Ejercicios:

1. Con la lógica del heapsort ordenar paso a paso el arreglo 88,85,83,72,73,42,57,6,48,60 se necesitan 19 pasos para llegar al arreglo ordenado.