Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de datos y algoritmos — 2' 2020

Ayudantía 3

1 *Stream* de datos [P1 I1 2011-1]

1.1 Única fuente

Suponga que tenemos una fuente de datos desordenados. Queremos enviar la totalidad de los datos recibidos por un único canal de salida, ordenados de mayor a menor. Se sabe a priori al cantidad de datos que van a llegar. Da un algoritmo que, utilizando un arreglo, resuelva este problema con una complejidad mejor a $O(n^2)$.

1.2 Múltiples fuentes

Supón que tenemos k fuentes de datos. Los datos de cada fuente vienen ordenados, p.ej, de mayor a menor. Queremos enviar la totalidad de los datos recibidos por un único canal de salida, también ordenados de mayor a menor. El dispositivo que debe hacer esta *mezcla ordenada de datos* tiene una capacidad **limitada** de memoria.

Da un algoritmo para este dispositivo, que le permita hacer su tarea empleando un arreglo a de k casilletos, en que cada casillero tiene dos campos:

1. $a[j].data$ se puede almacenar un dato
2. $a[j].num$ se puede almacenar un numero entero entre 1 y k.

Para recibir un dato desde la fuente i se ejecuta $recieve[i]()$, y para enviar un dato x por el canal de salida, se ejecuta $send(x)$.

2 Ejercicio 2

Cree un algoritmo para encontrar la suma de los n elementos menores de un árbol binario de búsqueda. Puede modificar la estructura del árbol para hacerlo más eficiente.

Soluciones

1 *Stream* de datos [P1 I1 2011-1]

1.1 Única fuente

Para resolver este problema, utilizaremos un *heap*. ¿Por qué un heap? Porque las operaciones de ordenación en un heap permiten lograr una complejidad logarítmica, y debido a que conocemos la cantidad de datos que van a llegar, podemos hacer un arreglo arbitrariamente grande. Recuerda que el *heap* es una estructura que 'corre' sobre un arreglo.

Siguiendo la idea anterior, el algoritmo que se utilizará será *heapsort*, que funciona con complejidad $O(n \cdot log(n))$. El paso a paso sería el siguiente:

1. Llega el primer dato, y se almacena en la primera posición, vale decir, la raíz del heap.
2. Llega el siguiente dato, que se agrega en la **primera posición desocupada**, en este caso, la segunda posición.
3. Ahora se realiza un *sift up* sobre el dato insertado, donde el dato agregado 'sube' a la posición que le corresponde, es decir, llega a una posición que mantenga las propiedades del heap. La complejidad de este algoritmo es $O(log(n))$.

Este procedimiento se repite hasta que todos los datos han llegado. Sabemos que todos los datos han llegado, debido a que sabemos a priori cuantos son. Ahora tenemos un heap con los datos, y lo que se debe hacer es extraer los datos ordenadamente. Para esto, se debe hacer lo siguiente:

1. Se saca el elemento en la raíz, y se agrega en la primera posición de la lista.
2. Se coloca el elemento del**último espacio ocupado** en la raíz, y se hace *sift down*, para que el elemento 'baje' a una posición que mantenga las propiedades del heap. La complejidad de este algoritmo es $O(log(n))$.

Este procedimiento se realiza hasta que el heap quede vacío. De esta forma, los datos se van agregando de forma ordenada a la lista.

Este algoritmo tiene complejidad $O(n \cdot log(n) + n \cdot log(n)) = O(n \cdot log(n))$.

1.2 Múltiples fuentes

La idea es armar un (max-)heap binario de tamaño k, inicialmente con el primer dato (el mayor) de cada fuente, de modo que en la raíz quede el mayor de todos los datos:

```
for (i=0; i<k; i= i+1)
      x.data = recieve[i]()
      x.num = i
      insertObject(x)
```

A countinuación, hay que ir sacando el dato que está en la raíz, enviándolo por el canal de salida, y reemplazándolo en el heap por el próximo dato recibido de la misma fuente de donde provenía el que salió:

```
while ( true )
      x = xMax()
      i = x.num
      send(x.data)
      x.data = recieve[i]()
      insertObject(x)
```

2 Suma n elementos menores de un árbol binario de búsqueda

Para hacer que este algoritmo sea más eficiente modificaremos la estructura del árbol binario de búsqueda:

Struct *Nodo* **contains**
- int valor;
- int cantidad_nodos;
- int suma;
- Node* hijo_izquierdo;
- Node* hijo_derecho;

end

El algoritmo lo vamos a definir recursivamente con una función que retorna la suma de los n elementos menores de un nodo en particular. Primero definimos dos casos base:

Function `sumaNMenores`(*nodo, n*)**:**
- **if** $n == 0 \ \vee \ \neg$nodo **then**
  - **return** 0
- **else if** nodo $\rightarrow$ cantididad_nodos $== n$ **then**
  - **return** nodo $\rightarrow$ suma
- ...

Luego, cuando hay un hijo a la izquierda hay que incorporar los elementos menores de este nodo en la suma (porque estos son los menores del nodo original por definición del árbol de búsqueda binario). Para esto tomaremos dos casos:

Caso 1. La cantidad de nodos en el hijo izquierdo es mayor o igual a n

Function `sumaNMenores`(*nodo, n*)**:**
- ...
- **else if** nodo $\rightarrow$ hijo_izquierdo $\wedge$ nodo $\rightarrow$ hijo_izquierdo $\rightarrow$ *cantidaad_nodos* $\geq n$ **then**
  - **return** sumaNMenores(nodo $\rightarrow$ hijo_izquierdo, n)
- ...

Caso 2. si es que la cantidad de nodos en el hijo izquierdo es menor a n, o entonces también hay que tomar los nodos de la derecha. (Las definiciones de variables son solo para hacer más legible el valor retornado.)

Function `sumaNMenores`(*nodo, n*)**:**
- ...
- **else if** nodo $\rightarrow$ hijo_izquierdo $\wedge$ nodo $\rightarrow$ hijo_izquierdo $\rightarrow$ *cantidaad_nodos* $< n$ **then**
  - suma_izquierda = nodo $\rightarrow$ hijo_izquierdo $\rightarrow$ suma
  - nodos_izquierda = nodo $\rightarrow$ hijo_izquierdo $\rightarrow$ *cantidaad_nodos*
  - mi_valor = nodo $\rightarrow$ valor
  - hijo_derecha = nodo $\rightarrow$ hijo_derecho
  - **return** suma_izquierda + mi_valor + sumaNMenores(hijo_derecha, n − nodos_izquierda − 1)
- ...

Luego, el único caso que falta es cuando no existe un nodo en la izquierda. En este caso n tiene que ser mayor o igual a 1 por el primer caso base, y además el nodo mismo es el menor elemento por lo que se entrega su valor y se hace una recursión con los nodos de la derecha.

Function `sumaNMenores`(*nodo, n*)**:**
```
...
else
    return nodo → valor + sumaNMenores(nodo → hijo_derecho, n − 1)
```

El algoritmo final sería entonces:

Function `sumaNMenores`(*nodo, n*)**:**
```
if n == 0 ∨ ¬nodo then
    return 0
else if nodo → cantididad_nodos == n then
    return nodo → suma
else if nodo → hijo_izquierdo ∧ nodo → hijo_izquierdo → cantidaad_nodos ≥ n then
    return sumaNMenores(nodo → hijo_izquierdo, n)
else if nodo → hijo_izquierdo ∧ nodo → hijo_izquierdo → cantidaad_nodos < n then
    suma_izquierda = nodo → hijo_izquierdo → suma
    nodos_izquierda = nodo → hijo_izquierdo → cantidaad_nodos
    mi_valor = nodo → valor
    hijo_derecha = nodo → hijo_derecho
    return suma_izquierda + mi_valor + sumaNMenores(hijo_derecha, n − nodos_izquierda − 1)
else
    return nodo → valor + sumaNMenores(nodo → hijo_derecho, n − 1)
```

En todas las líneas de este algoritmo podemos observar dos tipos de operaciones:

1. Obtener el valor en un puntero: operación O(1)
2. Llamar recursivamente a la función en un nodo distinto en un nivel inferior.

Debido a que en el peor caso se llega al nodo del último nivel del árbol, y nunca se devuelve, la complejidad es de $O(h)$ donde h es la altura del árbol. Luego, en los árboles binarios la altura máxima de un árbol es n, por lo que la complejidad del algoritmo es en el peor caso $O(|\text{árbol}|)$.