Tarea 08 Programación

Marco Antonio Flores Pérez
Natalia Huitzil Santamaria
Daniela Isis Flores Silva

Octubre 2020

1 Implementación del montículo ternario

Definimos la siguiente estructura donde definimos la capacidad del árbol, n nos indica el numero de elementos que hay en el árbol y en el arreglo *data* los datos del árbol ordenados.

```
typedef struct _THeap{
  unsigned int capacity;
  int n;
  int *data;
}THeap;
```

1.1 Función THeap_new

La siguiente función crea un árbol con la capacidad dada en el parámetro, observar que en el primer parámetro de calloc se pide una unidad más, esto es porque en el arreglo el elemento $data[0]$ no se toma en cuenta.

```
THeap *THeap_new(unsigned int cpt){
  THeap *new=NULL;
  new=(THeap *)calloc(1,sizeof(THeap));

  if(new!=NULL){
    new->capacity=cpt;
    new->n=0;
    new->data=(int *)calloc(cpt+1,sizeof(int));//Sumar el 0
  }else{
    printf("Solicitud de memoria denegada\n");
  }
  return new;
}
void free_THeap(THeap ** hptr){
  free((*hptr)->data);
  free(*hptr);
}
```

Complejidad

Para la función *THeap_new* tiene complejidad $O(1)$ ya que no se tiene ningún ciclo que tome más de tiempo constante para leer cada línea, al igual que para la función *free_THeap*.

1.2 Función bottomUpHeapify

A diferencia del caso binario dado un elemento en el árbol k, su padre se encuentre en $(k+1)/3$.

Observar que las siguientes funciones tienen un parámetro más, **char** ord, el cual se toma como una bandera para decidir que orden aplicar, esto es si el árbol se ordena con llave mínima o llave máxima.

```
void bottomUpHeapify(int *arr, int k, char ord){

  if(ord==1){
    while (k > 1 && arr[(k+1)/3]>arr[k]) {
      swap(arr ,k, (k+1)/3);
      k=(k+1)/3;
    }
  }else{
    while (k > 0 && arr[(k+1)/3]<arr[k]) {
      swap(arr ,k, (k+1)/3);
      k=(k+1)/3;
    }
  }
}
```

Complejidad

Para la función *bottomUpHeapify*, el árbol tiene altura $h = \lfloor \log_3 n \rfloor$, entonces la complejidad de esta función es $O(\log n)$ ya que sólo entra uno de los dos ciclos que se muestran según la condicional a la que entra, la cuál se lee en $O(1)$, y cada uno recorre el árbol por niveles comparando dos datos, uno en cada nivel.

1.3 Función topDownHeapify

```
void topDownHeapify(int * arr, int k, int n, char ord){
  if(ord == 0){
    while(3*k-1 <= n){
      int j = 3*k - 1;
      if(j<n && arr[j]<arr[j+1]){
        j++;
      }
      else if(j<n-1 && arr[j]<arr[j+2]){
        j += 2;
```

```
      }

      if(arr[k] >= arr[j]) break;

      swap(arr, k, j);
      k = j;
    }
  }
  else{
    while(3*k-1 <= n){
      int j = 3*k - 1;
      if(j<n && arr[j]>arr[j+1]){// j<n para j++
        j++;
      }
      else if(j<n-1 && arr[j]>arr[j+2]){//j<n-1 para j = j+2
        j += 2;
      }

      if(arr[k] <= arr[j]) break;

      swap(arr, k, j);
      k = j;
    }
  }
}
```

Complejidad

Al igual que la función anterior, sólo uno de los ciclos es el que dictamina la complejidad de la función y en cada ciclo *while* el número máximo de iteraciones será la altura del árbol, por lo que su complejidad es $O(\log n)$.

1.4 Funciones getMax,insert y removeMax

La función getMax en la misma que la implementada en arboles binarios. Para las funciones insert y removeMax, la estructura también es la misma que para arboles binarios, sin embargo, se agregó un dato de entrada **char** ord para distinguir de que tipo de árbol se trata (0 para max y 1 para min).

```
int getMax(THeap *h){
  return h->data[1];
}

void insert(THeap *h, int data, char ord){
  if (h->n==h->capacity-1)
    return;
  h->data[++h->n]=data;
  bottomUpHeapify(h->data,h->n,ord);
```

```
}
int removeMax(THeap *h, char ord){
  if (h->n<1)
    return -1;
  int tmp = h->data[1];
  h->data[1]= h->data[h->n--];
  topDownHeapify(h->data,1,h->n,ord);
  return tmp;
}
```

Complejidad

Para la función *getMax* la complejidad es $O(1)$ porque sólo se lee una línea dentro de la función. En el caso de las funciones *insert* y *removeMax* la única línea que no se lee en $T(n) = O(1)$ es en la línea que llama a las funciones *bottomUpHeapify* y *topDownHeapify* respectivamente, por lo que la complejidad de ambas es $O(\log n)$.

2 Aplicación al cálculo de la mediana en streaming

```

double theap_median(THeap * thmin, THeap * thmax, int data){
    double mactual=0;
    if(thmax->n>thmin->n){
        mactual=getMax(thmax);
        if(data<mactual){
            insert(thmin,removeMax(thmax,0),1);
            insert(thmax,data,0);
        }
        else insert(thmin,data,1);
        mactual=(double)(getMax(thmax)+getMax(thmin))/2.0;
        return mactual;
    }
    else{
        if(thmax->n<thmin->n){
            mactual=getMax(thmin);
            if(data>mactual){
                insert(thmax,removeMax(thmin,1),0);
                insert(thmin,data,1);
            }
            else insert(thmax,data,0);
            mactual=(double)(getMax(thmax)+getMax(thmin))/2.0;
            return mactual;

        }
        else{
            if(thmax->n==0){
```

```
28                  insert(thmax,data,0);
29                  mactual=data;
30                  return mactual;
31              }
32              else{
33                   mactual=(double)(getMax(thmax)+getMax(thmin))/2.0;
34                   if(data<mactual){
35                      mactual=getMax(thmax);
36                      return mactual;
37                   }
38                   else{
39                      insert(thmin,data,1);
40                      mactual= getMax(thmin);
41                      return mactual;
42                   }
43              }
44          }
45      }
46
47    return -1;
48  }
```

Complejidad

Las líneas que no toman tiempo constante ($O(1)$) en su lectura son las líneas 7, 8, 18, 19, 28 y 39, las cuales se leen en un tiempo $T = O(\log n)$ y dado que se llaman máximo dos veces en algún caso de las condicionales, la complejidad de toda la función es $O(\log n)$.

En efecto la aplicación pasa la prueba base.

```
linux21@ubuntu:~/ALGPROI/Tarea8/javier.g-ricardo.n$ make main
gcc -g -Wall -lm main.c -o main.app
./main.app < input.txt > output.tmp
diff --side-by-side --report-identical-files output.txt output.tmp
1.000                                                           1.000
2.500                                                           2.500
4.000                                                           4.000
3.500                                                           3.500
Los archivos output.txt y output.tmp son idénticos
```

Figure 1: main.c

Se diseñaron distintos tests para la función *insert*() y *mediana*() con números positivos, negativos y combinación de ambos.

Se añade una prueba extra para el caso de heap con orden mínimo.

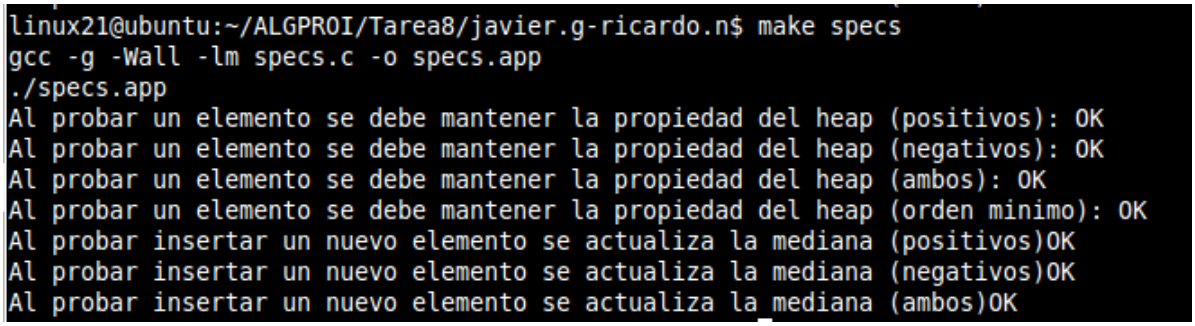

Figure 2: specs.c