Auxiliar Extra - Control 2

Profesores: Jeremy Barbay
Patricio Poblete
Auxiliares: Daniela Campos, Cristóbal Muñoz
Sven Reisenegger, Bernardo Subercaseaux

P1. *P3 2017-1*

El algoritmo usual de inserción en un árbol de búsqueda binaria (ABB) deja al nuevo elemento en el borde inferior del árbol. En este problema vamos a programar un algoritmo alternativo que deja al nuevo elemento como raíz del árbol resultante. Para esto, se compara el nuevo elemento con la raíz del árbol y, dependiendo de si es menor o mayor que ella, se inserta (recursivamente) en el subárbol izquierdo o derecho, respectivamente. Observando que después de esto el nuevo elemento debe estar como hijo directo de la raíz, se hace una rotación simple para dejarlo como raíz.

Escriba un método en Java con encabezamiento "Nodo insertarRaiz(int x, Nodo r)" que inserte un nuevo elemento x en un árbol cuya raíz es r, y que retorne un puntero a la raíz del árbol resultante. Suponga que la llave x es distinta de todas las que están en el árbol.

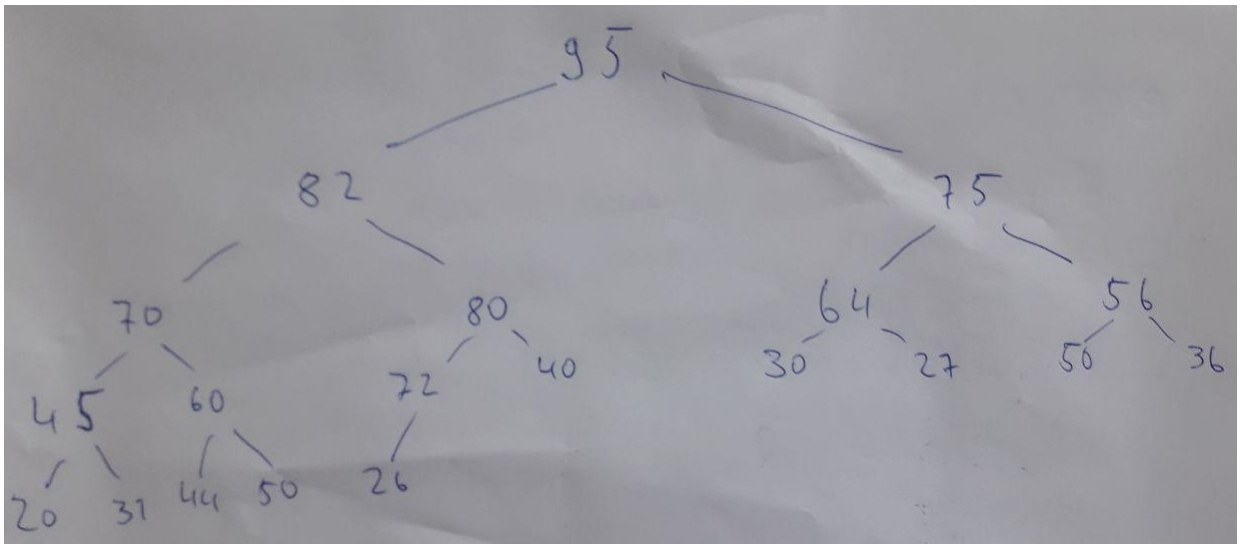
```
Nodo insertarRaiz(int x, Nodo r){
  //caso base, si se inserta en un arbol vacio
  //se crea un nodo y se retorna
  if(r==null)return new Nodo(x);
  Nodo puntero; //se crea un puntero a la nueva raiz
  if(x > r.info){
    puntero = insertarRaiz(x,r.der); //se inserta a la derecha
    //se hace una rotacion simple para dejar el nodo nuevo como raiz
    r.der = puntero.izq;
    puntero.izq = r;
  }
  else{ //analogo al caso anterior, pero se rota e inserta al otro lado
    puntero = insertarRaiz(x, r.izq);
    r.izq = puntero.der;
    puntero.der = r;
  }
  return puntero; //se retorna la nueva raiz
}
```

P2. *P1a 2017-1* Considere el siguiente heap:

95	82	75	70	80	64	56	45	60	72	40	30	27	50	36	20	31	44	50	26
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Dibuje el árbol correspondiente a este heap. Luego modifique el elemento del casillero 10, cam- biando su valor de 72 a 85, indique las operaciones necesarias para preservar la condición de heap y muestre el heap (arreglo) resultante.

El arbol:



Si se cambia el 72 al 85 entonces se rompe la condición de heap. Sería necesario cambiar el del indice 10 (85) por su padre (índice 5: 80) y luego de eso cambiar el del índice 5 (ahora 85) por el del índice 2 (82).

P3. *P4 2016-1*

Considere una tabla de hashing A con Linear Probing, de tamaño m, con n elementos almacenados y suponga que la función h está dada. Suponga que las llaves son enteros positivos y que una llave en cero indica que la celda respectiva está desocupada. Programe un algoritmo que, dado un subíndice k, elimine la llave almacenada en A[k]. Puede suponer que en ese lugar hay una llave distinta de cero (no necesita chequearlo).

Observe que para eliminar un elemento no basta con poner la celda respectiva en cero, porque algunas búsquedas terminarían erróneamente ahí, cuando el elemento buscado podría estar más a la derecha. Lo que hay que hacer es recorrer el bloque hacia la derecha buscando alguna llave, si hay, que se pueda mover hacia el lugar libre (eso depende de su valor de hashing, no cualquiera se puede mover, ¿por qué?). Si se encuentra alguna, se mueve y para rellenar el lugar que ahora quedó libre, se repite el proceso.

Nota: Para evitar complicaciones, en su programa suponga que el bloque termina antes de llegar al extremo derecho de la tabla. Discuta qué pasaría en el caso general.

```
void eliminar(int k){
  A[k] = 0;
  int i = k+1;
  while(A[i] != 0){
    //se busca hacia delante para ver si
    // hay algun elemento que pueda reemplazar al eliminado
    if(h(A[i]) <= k){
```

```
            //si se encuentra uno que lo pueda reeplazar
            A[k] = A[i];//se mueve a la posicion k de la tabla
            eliminar(i); //se repite el proceso de forma recursiva
            return;
        }
        i++;
    }
}
```

En el caso general, al dar la vuelta habría que ver que que el elemento que se trae de vuelta tenga un hash que sea mayor que el límite izquierdo del bloque en el que se está eliminando además de que sea menor que k. Pero el límite izquierdo del bloque puede estar a la derecha de k (porque el bloque puede darse la vuelta).

No se si se entendió bien la discusión (pero no le den tantas vueltas si no se entiende, puede ser bastante enredado).

P4. *P2 2017-1* En clases vimos que es fácil mezclar ("merge") dos secuencias ordenadas, cada una de largo n, si se dispone de un área de salida de largo 2n. En este problema veremos que el "merge" se puede hacer incluso si sólo tenemos n celdas auxiliares disponibles, en lugar de 2n. Para esto, considere que se tiene un arreglo a de tamaño 3n, el que imaginaremos dividido en tres segmentos, cada uno de largo n (correspondientes a los casilleros 0..(n − 1), n..(2n − 1) y 2n..(3n − 1), respectivamente). Suponga que el segundo y el tercer segmento contienen cada uno secuencias de elementos ordenadas ascendentemente. El objetivo de este problema es mezclar estos dos segmentos, dejando el resultado ocupando los 2n primeros casilleros del arreglo. Por ejemplo, si el arreglo inicial es (con n = 5):

					21	36	40	61	80	10	50	70	74	95
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

entonces el arreglo resultante debería contener:

10	21	36	40	50	61	70	74	80	95					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

(Los casilleros que aparecen como vacíos podrían contener información, pero ésta no es relevante para el algoritmo). Excepto por algunas variables auxiliares, no se puede usar memoria extra y el algoritmo debe funcionar en tiempo $\theta(n)$.

(a) Programe en Java un método de encabezamiento "void mezclar(int[ ] a, int n)" que realice la tarea descrita. Indique claramente el invariante de su proceso.

(b) Demuestre que en el curso del merge, el algoritmo nunca sobreescribe un dato útil ("nunca se pisa la cola").

```
void mezclar(int[a], int n){
  int i = n;
  int j= 2*n;
  int k = 0;
  while(i<2n && j<3n){
    if(a[i] > a[j]){
      a[k] = a[i];
      i++;
    }
    else{
      a[k] = a[j];
      j++;
    }
    k++;
  }
  while(j<3n){
    a[k] = a[j];
    i++; k++;
```

```
    }
    //el while con i<2n es innecesario porque si se
    //copio todo el ultimo tercio
    // i = k;
}
```

El algoritmo no se pisa la cola, porque una vez que se copia todo el último tercio, k se hace igual a i, y el arreglo desde i hasta 2n-1 ya está ordenado. De este modo los elementos del último tercio del arreglo no sobreescriben los del segundo tercio.