Universidad Simón Bolívar
Departamento de Computación y Tecnologías de la Información
CI2692 - Laboratorio de Algoritmos y Estructuras II
Enero - Marzo 2017

CI2692 - Laboratorio 3

Heap. Heapsort. Cola de Prioridad

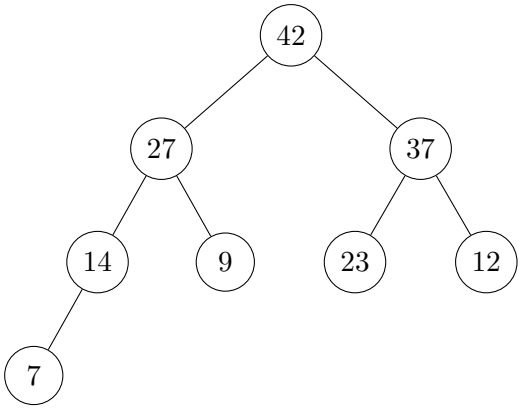
Sartenejas, febrero de 2017

Introducción

El objetivo de este laboratorio representar un *heap* a través de un arreglo. Específicamente el arreglo *ArrayT* previamente usado en los laboratorios. Esto es un requerimiento y estará prohibido usar las listas de *Python*; las mismas no serán necesarias. Después de haber construido una representación de *heap*, usted deberá implementar el algoritmo de ordenamiento *heapsort* para ordenar una serie de objetos comparables bajo cierto criterio y por último habrá un ejercicio conceptual para visualizar la equivalencia entre un *heap* y una cola de prioridad.

Representación del *heap*

En primer lugar hay que definir como será la estructura abstracta que contendrá los valores a manipular. Para éste laboratorio nuestro *heap* será un árbol binario, en el cual se mantendrá la condición de que cada nodo padre será mayor que sus hijos, en lo que respecta a la comparación del valor almacenado en los nodos. Nuestro *heap* binario estará representado por un arreglo *ArrayT*, tomando como referencia las clases de teoría. A continuación se muestra lo que sería un *heap* binario que almacena enteros.



Donde su representación en un arreglo sería de la siguiente manera, tomando en cuenta que la raíz del *heap* se encuentra en la posición 0 del arreglo.

42	27	37	14	9	23	12	7

Ahora, si se desea obtener el índice del hijo izquierdo, hijo derecho o del padre, se deben usar la siguientes funciones; las mismas están definidas en el código base adjunto a este enunciado.

```
1      def parent(i):
2          return (i + 1) // 2 - 1
3
4      def left(i):
5          return (2*(i+1)-1)
6
7      def right(i):
8          return (2*(i+1))
```

Alcanzar la condición de *heap*

La función clave para lograr realizar este laboratorio se llama ***heapify***, la cual mantiene la condición antes mencionada donde todo nodo padre es mayor a sus hijos. Esta función recibe un arreglo al cual se le aplican una serie de intercambios entre sus nodos en caso de ser necesario. Partiendo de un nodo ***i*** y suponiendo que sus hijos, siendo ellos subárboles, cumplen la condición de *heap*, aunque él puede que no la cumpla la condición. A continuación se describe lo que hace la función.

```
1  def heapify(array, i, length):
2      hijo_izquierdo ← left(i)
3      hijo_derecho ← right(i)
4
5      el_mayor ← i
6
7      if hijo_izquierdo < length and array[hijo_izquierdo] > array[i]
8          el_mayor ← hijo_izquierdo
9
10     if hijo_derecho < length and array[hijo_derecho] > array[el_mayor]
11         el_mayor ← hijo_derecho
12
13     if el_mayor ≠ i
14         swap(array,el_mayor,i)
15         heapify(array,el_mayor,length)
```

La funciones ***left*** y ***right*** son las que están especificadas en la sección anterior. La función ***swap*** intercambia dos elementos del arreglo que es pasado como primer parámetro, los otros parámetros son las posiciones de los elementos a intercambiar. Cabe destacar que esta será la única función recursiva del laboratorio.

Crear un *heap*

Teniendo definida la función ***heapify*** no basta para crear un *heap* a partir de un arreglo arbitrario, dado que esta parte de un nodo y desciende por sus hijos. A pesar de ello, gracias a ***heapify***, es posible crear un *heap* aplicando la función sobre todos aquellos nodos que tengan al menos un hijo y buscando el padre con al menos un hijo que se encuentre en el nivel más bajo del árbol. Esto se consigue con el siguiente pseudocódigo.

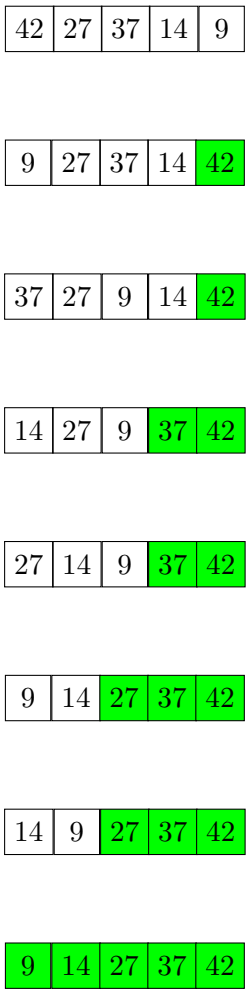
```
1    def build_max_heap(array):
2        for i from (len(array) // 2) downto 0
3            heapify(array,i,len(array))
```

Ordenamiento por *heapsort*

Después de haber creado el *heap*, ya están las condiciones ideales para aplicar el algoritmo de ordenamiento *heapsort*. Ya sabemos que el valor más grande se encuentra en la raíz del árbol, o mejor dicho, en la posición 0 del arreglo. Partiendo de ese hecho, se define el siguiente pseudocódigo.

```
1    def heap_sort(array):
2        build_max_heap(array)
3        for i from (len(array)-1) downto 1
4            swap(array,0,i)
5            heapify(array,0,i)
```

En la próxima página se puede visualizar una pequeña corrida del algoritmo.



Obtener el primero en una cola de prioridad

Finalmente, dejando a un lado el algoritmo de ordenamiento *heapsort*, pero manteniendo la estructura del *heap*, es posible representar una cola de prioridad usando en esencia lo mismo que se tiene hasta el momento. La condición no sería quien es mayor, sino quien tiene mayor prioridad pero a fines prácticos es esencialmente lo mismo. La idea es similar a *heapsort* aunque en este caso se devuelve el nodo con mayor valor pero antes de llamar a la función se debe construir el *heap* a partir del arreglo.

```
1     def dequeue(array):
2         if len(array) < 1
3             error "El arreglo debe tener más de un elemento"
4
5         aux_array = array[1..len(array)-1]
6
7         heapify(aux_array,0,len(aux_array))
8         return (array[0],aux_array)
```

Requerimientos específicos

Todos los archivos base para el desarrollo del laboratorio se encuentran adjuntos a este enunciado. Usted tendrá que modificar el archivo "**heap_functions.py**", donde implementará las funciones ***swap***, ***heapify***, ***build_max_heap***, ***heap_sort*** y ***dequeue***. En el archivo "**ordenamiento.py**" importará solamente la función *heap_sort* del módulo "**heap_functions.py**". Así podrá correr el cliente de ordenamiento con *heap_sort*, y adicionalmente deberá estar su implementación de *merge_sort* para comparar ambos algoritmos. Deberá generar una gráfica comparativa entre los algoritmos con al menos 5 tamaños de arreglos diferentes. Cualquier otro algoritmo de ordenamiento deberá estar comentado. Específicamente se pide:

- Implementar ***swap(ArrayT array, Int i, Int j)***
- Implementar ***heapify(ArrayT array, Int i, Int length)***.
- Implementar ***build_max_heap(ArrayT array)***.
- Implementar ***heap_sort(ArrayT array)***.
- Implementar ***dequeue(ArrayT array)***.

Esta última función, ***dequeue***, se usará sólo en el cliente para la cola de prioridad proporcionado junto a este enunciado, un archivo llamado "**cliente_cola.py**". Donde se simulará la ejecución de ciertas actividades previamente definidas de manera secuencial, usando la función ***dequeue_max***, cuya salida se puede ver a continuación.

```
1   Secuencia de actividades a realizar:
2
3   Realicé la actividad: Recargar teléfono
4   Faltan: 4
5   Realicé la actividad: Hacer currículum
6   Faltan: 3
7   Realicé la actividad: Lavar la ropa
8   Faltan: 2
9   Realicé la actividad: Ir al odontólogo
10  Faltan: 1
11  Realicé la actividad: Comprar comida
12  Terminé
```

Entrega del laboratorio

Al finalizar el desarrollo del laboratorio, deberá comprimir el código fuente y la gráfica en un archivo llamado *lab_3_X_Y.tar.gz* donde X y Y corresponden al carné de cada integrante del grupo. En caso de no tener pareja llame el comprimido *lab_3_X.tar.gz* donde X es su número de carné. La entrega será hasta las 4:30pm del jueves 2 de febrero de 2017.