Estructuras de datos avanzadas
HEAPS

Realizado por:
Prof. Rhadamés Carmona
Última revisión: 26 / 03 / 2019

Agenda

- Introducción
- Definición de heaps
- Operaciones
  - Inserción
  - Desencolar
  - Heapify
  - Heap Sort
  - Otras: eliminar, incremento de clave
- Tarea: quick sort versus heap sort

Introducción

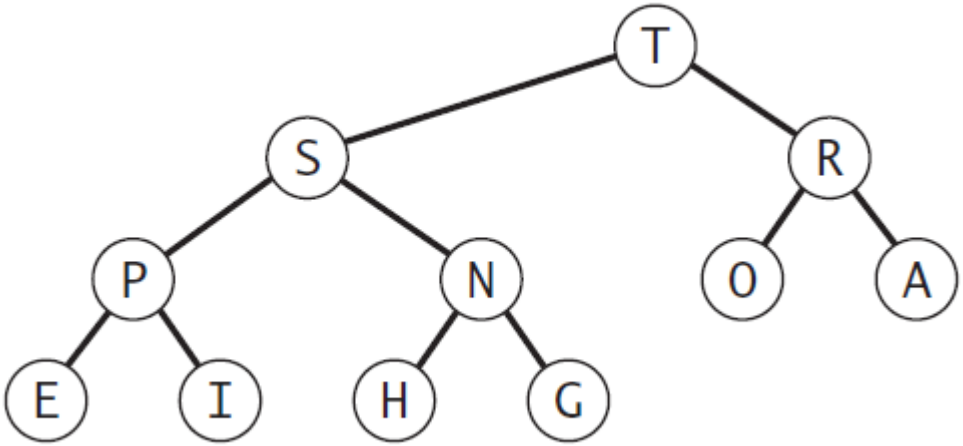
- Hay problemas que requieren de una cola de prioridad.
- En una cola se atiende (elimina) el primer elemento, mientras se insertan al final.
- En una cola de prioridad, los elementos se insertan según su prioridad (no siempre al final).
- Ejemplos: cuando los elementos se procesan en cronológico (y no en orden de inserción), cuando los elementos se procesan según el nivel de error, etc.

Heaps

- Es una estructura de datos que puede soportar eficientemente las operaciones de una cola de prioridad.
- Todo nodo interno, con la posible excepción de un nodo especial, tiene exactamente dos hijos. El nodo especial, si existe, está situado en el nivel K-1 y posee un hijo izquierdo, pero no derecho. Todas la hojas están en el nivel K o en los niveles K y K-1. Ninguna hoja de nivel K-1 está a la izquierda de un nodo interno del mismo nivel

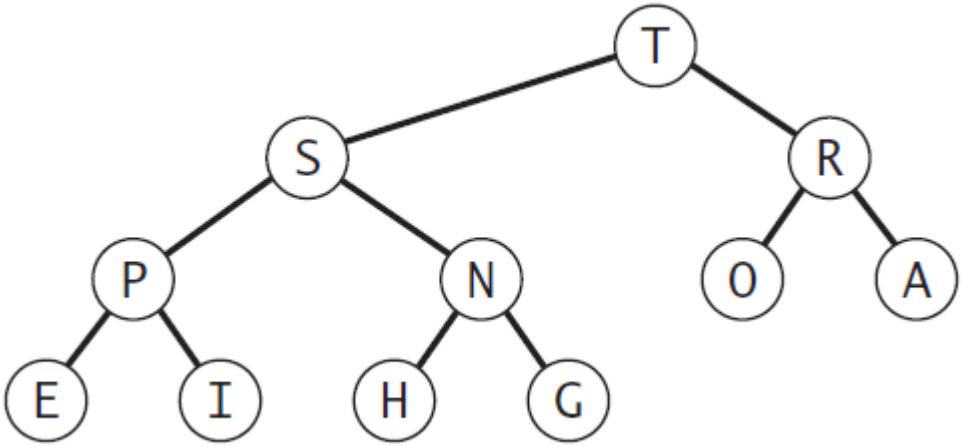
Heaps

- Intuitivamente es un árbol binario en el que los nodos internos han subido lo más posible en el árbol y los nodos hoja del último nivel están lo más a la izquierda posible. La altura es $h = \lfloor \log_2 n \rfloor$.



Heaps

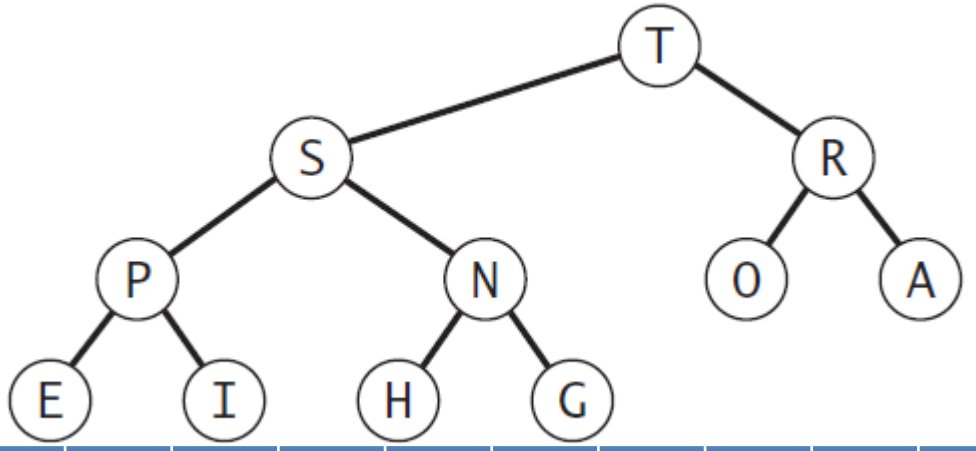
- Note que para cada nodo, la clave de sus hijos son menores (maxHeap). También se puede hacer al revés (minHeap).



Heaps

- Estos árboles pueden almacenarse en un arreglo. La raíz está en la posición 1, el hijo izquierdo del nodo p está en 2p y su derecho en 2p+1.
- El padre del nodo de posición p está en p/2. La posición cero contiene el número de elementos o no se usa.
- Nivel0 – Nivel 1 – Nivel 2 … Nivel K

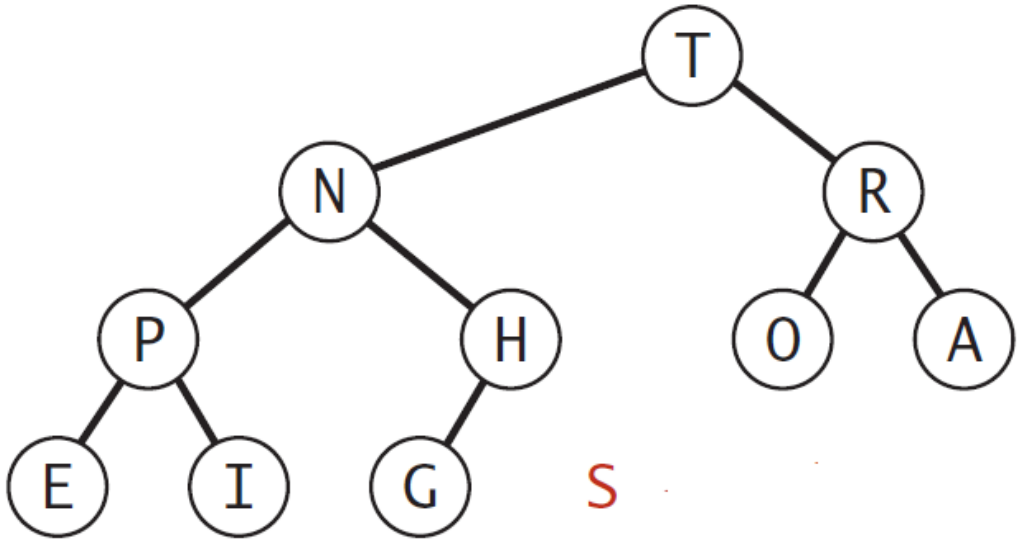
A[0] no se usa ó guarda n



12	T	S	R	P	N	O	A	E	I	H	G
0	1	2	3	4	5	6	7	8	9	10	1º

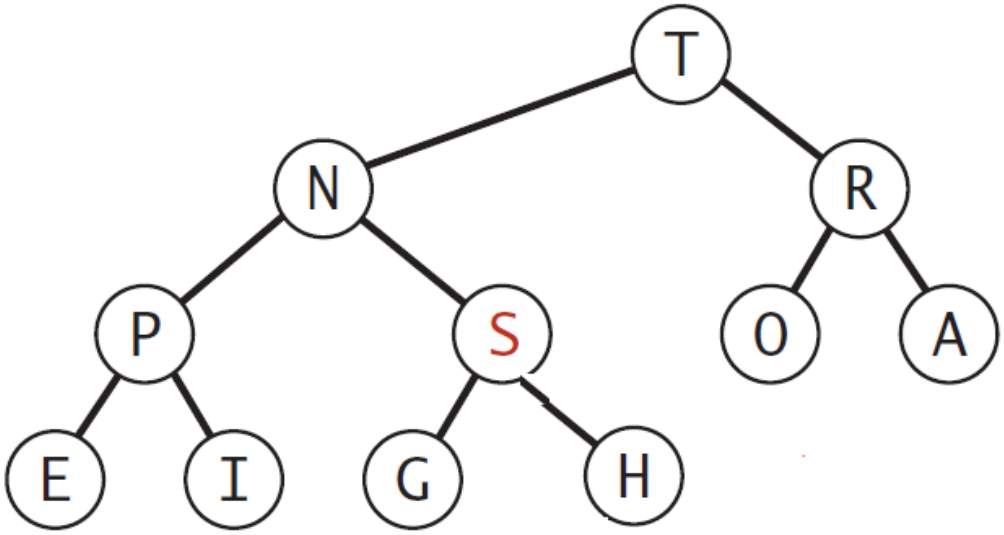
Heaps: operaciones

- **Inserción**: se coloca el nuevo elemento como último, y “flota” hasta no violar la relación mayor-menor entre padre e hijo.



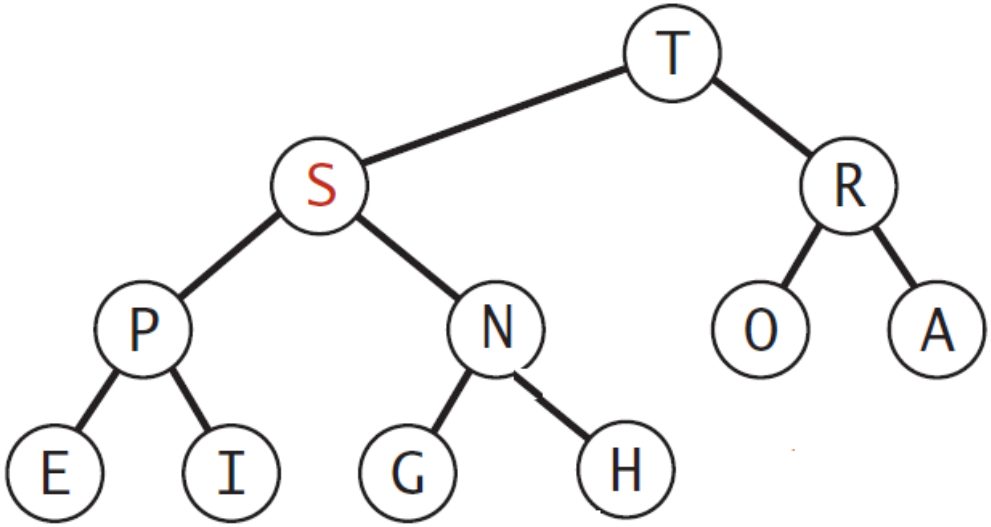
Heaps: operaciones

- **Inserción**: se coloca el nuevo elemento como último, y “flota” hasta no violar la relación mayor-menor entre padre e hijo.



Heaps: operaciones

- **Inserción**: se coloca el nuevo elemento como último, y “flota” hasta no violar la relación mayor-menor entre padre e hijo.



Heaps: operaciones

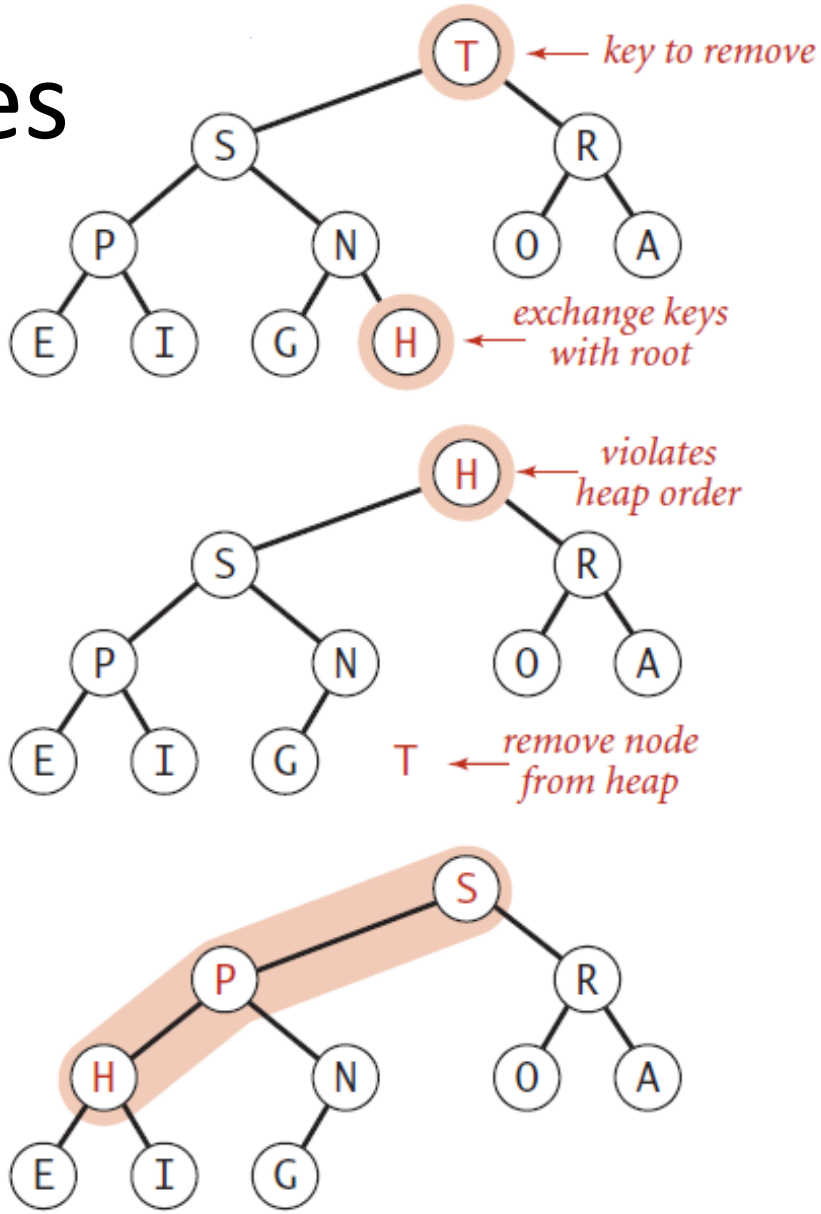
- **Inserción**: se coloca el nuevo elemento como último, y “flota” hasta no violar la relación mayor-menor entre padre e hijo, o llegar a la raíz.

```
void insertar(int  *A, int e, int &n) {
        A[n] = e;
        flotar(A,n);
        n++;
}

void flotar(int *A, int k) {
        while (k > 1 && A[k/2]<A[k])
        {
                swap(A[k/2],A[ k]);
                k = k/2;
        }
}
```

Heaps: operaciones

- **Desencolar**: se elimina la raíz. El último elemento se coloca en la raíz, y se hunde hasta no violar la relación mayor-menor entre padre e hijo, o llegar a una hoja.



Heaps: operaciones

- **Desencolar**: se elimina la raíz. El último elemento se coloca en la raíz, y se hunde hasta no violar la relación mayor-menor entre padre e hijo, o llegar a una hoja.

```
void erase(int *A, int &n) {
  swap(A[1], A[ n ]); n--;
  hundir(A,1,n);
}
void hundir(int *A, int k, int n) {
  while (2*k <= n)  {
    int j = 2*k;
    // bajo por el más grande de los hijos
    // si es un maxheap
    if (j < n && A[j]<A[j+1]) j++;
    if (A[k]>=A[j]) break;
    swap(A[k],A[ j]);
    k = j;
  }
}
```

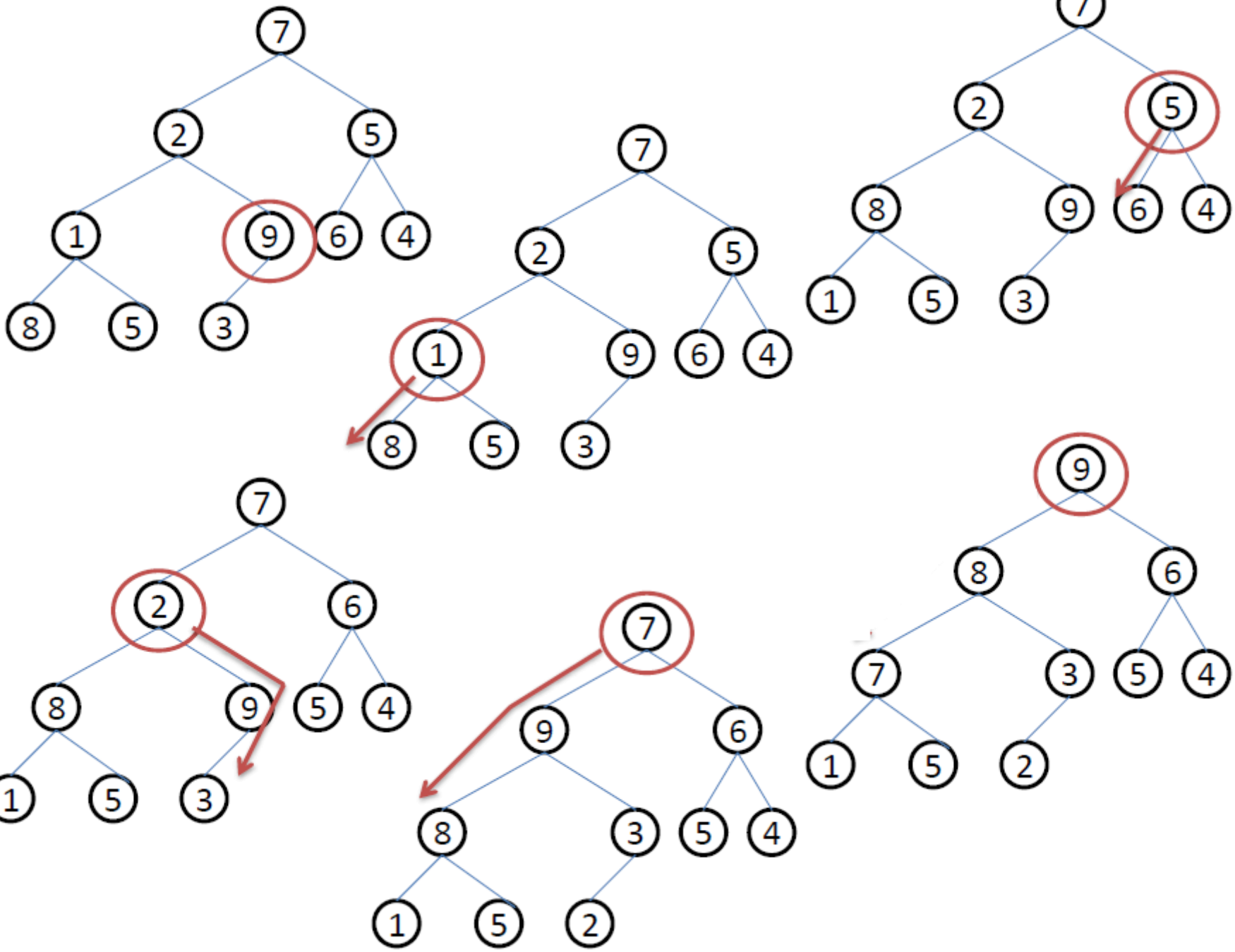
Heaps: operaciones

- **Heapify**: proceso de convertir un arreglo desordenado en un heap.
- Se basa en lograr la condición heap en los dos últimos niveles (k-, y k), hundiendo elementos del nivel k-1 hacia el nivel k.
- Luego hace lo mismo para nivel k-2. Y así sucesivamente hasta hundir el nodo raíz.

Heaps: operaciones

- **Heapify**: proceso de convertir un arreglo desordenado en un heap.

```
// se asume que T[0] no se usa, o guarda el valor  de n
void heapify(int *A, int n)
{
        for (int i = n/2; i>0; i--)
                hundir(A, i, n);
}
```


7
2
5
1
9
6
4
8
5
3
7
2
5
1
9
6
4
8
5
3
7
2
5
8
9
6
4
1
5
3
7
2
6
8
9
5
4
1
5
3
9
8
6
7
3
5
4
1
5
2
7
9
6
8
3
5
4
1
5
2

Heaps: operaciones

- **Heapsort**: proceso para ordenar un arreglo de n elementos usando un heap.
- Tomo el máximo (raíz) y lo intercambio con el último. Ya el último es el máximo. Entonces, decremento el tamaño del arreglo, y hundo la raíz. Repito el proceso para los N-1 elementos. Luego para los N-2 … hasta que N=1, en cuyo caso, la raíz es el mínimo del arreglo (el arreglo queda ordenado ascendentemente).

Heaps: operaciones

- **Heapsort**: proceso para ordenar un arreglo de n elementos usando un heap.

```
void heapSort(int *A, int n) {
        Heapify(A,n);
        for(i = n; i > 1; i--)   {
                swap(A[1], A[i]);
                hundir(A,1,--n);
        }
}
```

Heaps: otras operaciones

- **Eliminar(x)**: dado el apuntador a x, simplemente colocamos su key en $+\infty$ y lo hacemos flotar hasta la raíz. Luego desencolamos. O(log*n*).
- **Incrementar clave(x)**: al incrementar la clave del nodo x, el nodo podría flotar en el peor de los casos hasta la raíz. O(log*n*)

Heaps: operaciones

- Estudio de la complejidad de las operaciones
  - Insertar es $O(\log_2 n)$, porque se coloca de último, y luego flota hasta la raíz en el peor caso, y la altura del árbol es $\lfloor \log_2 n \rfloor$.
  - Eliminar es $O(\log_2 n)$, porque el último se coloca en la posición de la raíz, y luego se hunde hasta ser hoja (peor caso, $\lfloor \log_2 n \rfloor$ pasos).
  - Heapify: una cota muy superior es $O(n.\log_2 n)$, pues se hunden n/2 elementos con un muy peor caso de $\log_2 n$ intercambios por elemento para hundir. Sin embargo, O(n) se ajusta mejor (ver sig. lámina)
  - Heapsort: es $O(n.\log_2 n)$, pues requiere de un heapify de $O(n.\log_2 n)$ y de hundir n elementos con un muy peor caso de $\log_2 n$ intercambios por elemento para hundir.
  - Eliminar: el nodo flota hasta la raíz en $O(\log_2 n)$ y luego se desencola en $O(\log_2 n)$. Así, es $O(\log_2 n)$.
  - Incrementar clave: el nodo puede flotar hasta la raíz en el peor caso. Así es $O(\log_2 n)$.

Heaps: operaciones

- Calculando el orden de heapify. El tiempo de "hundir" depende de la altura del nodo.
- Cuántos nodos de altura h (con h=0..$\log_2 n$) hay a lo sumo? Esto es $\lceil n/2^{h+1} \rceil$
- Luego: $T(n) = \sum_{h=0}^{\log(n)} \left\lceil \frac{n}{2^{h+1}} \right\rceil * h = O\left(n * \sum_{h=0}^{\log(n)} \frac{h}{2^{h+1}} \right)$
- Note que la sumatoria resulta 0+1/4+1/4+3/16+4/32+5/64+…<= 1
- Así, T(n)=O(n)

Heaps versus Quick Sort

- El algoritmo de heap sort es O($n.\log_2 n$) al igual que el merge sort. Sin embargo, las operaciones del heap sort son bastantes simples de implementar.
- El algoritmo **original** de quick sort es O(n^2), y funciona así cuando el arreglo viene ordenado al revés o “casi al revés”. Por lo tanto, no se recomienda a utilizar cuando una espera de n^2 representa un riesgo importante.

Tarea

- **Tarea**: implementar una clase template en c++ con ciertas operaciones de heap. Instanciar la clase heap con un elemento que tenga varios atributos, y un operador de comparación “menor que”(<). Sobrecargue el operador < dentro de la clase Elem. Utilice la clase heap, y realice pruebas. Compare heap sort con quick sort con arreglos ordenados (asc y desc) y desordenados (random), variando n entre 1K y 10M. Ordene solo ascendentemente. Mida tiempo en microsegundos. Haga un pequeño informe con gráficas discutiendo los resultados. Entregar Lunes 03/06.

- **bool** operator < (**const** Elem &e) **const** {…}

Tarea

```
template <class Elem>
class heap<Elem>: public vector< Elem >  {
    public: heap();
    public: heap(const heap<Elem> &h);
    public: heap(const vector<Elem> &a);
    public: ~heap();
    public: void insert(const Elem &e);
    public: void remove();
    public: Elem &head();                    // retorna referencia
    public: Elem head() const;               // retorna copia
    public: static void sort(vector< Elem > &A);
    public: static void heapify(vector< Elem > &A);
    private: static void sink(vector< Elem > &A, int k, int n);// hundir
    private: void swim(int k);                                 // flotar
};
```

Good luck!