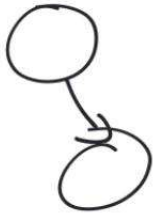
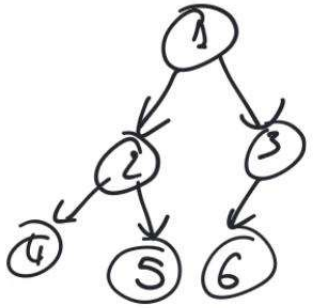
Hoy:
Heaps
Recorridos árboles, repaso
Complejidad algortímica, repaso
Hablar del parcial
Práctica





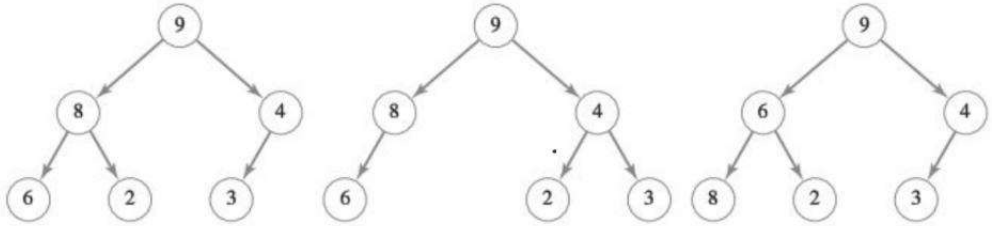
Heap

Colección de datos

Es un árbol binario con las siguientes características:

- Es *completo*, es decir, cada nivel del árbol está completamente lleno, excepto posiblemente al último nivel, y en dicho nivel los nodos se encuentran en las posiciones situadas más a la izquierda.
- Satisface la *propiedad de ordenación de montículo*: el elemento almacenado en cada nodo es mayor o igual que los elementos almacenados en sus hijos.

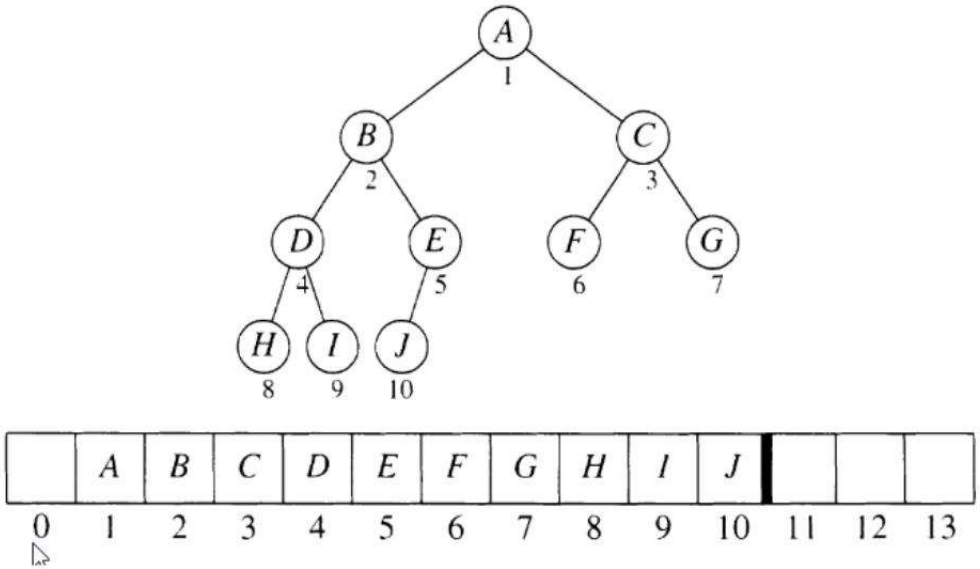
¿Heaps?



¿Max heap? ¿Min heap?

Operaciones básicas

- Construir un montículo vacío.
- Comprobar si un montículo es vacío.
- Insertar un elemento.
- Obtener el mayor elemento.
- Eliminar el mayor elemento.


A
1
B
2
C
3
D
4
E
5
F
6
G
7
H
8
I
9
J
10
A
B
C
D
E
F
G
H
I
J
0
1
2
3
4
5
6
7
8
9
10
11
12
13

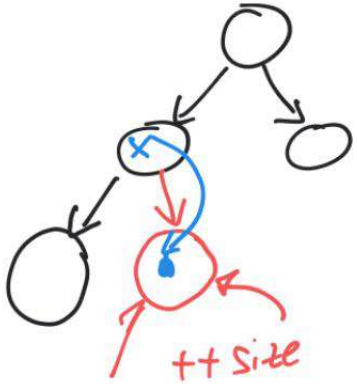
c

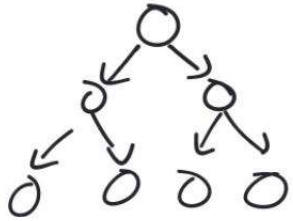
int i = 7;

7/2 ⇒ 3

We also need to maintain an integer that tells us the number of nodes currently in the tree. Then for any element in array position i, its left child can be found in position $2i$. If this position extends past the number of nodes in the tree, we know that the left child does not exist. Similarly, the right child is located immediately after the left child; thus it resides in position $2i + 1$. We again test against the actual tree size to be sure that the child exists. Finally, the parent is in position $\lfloor i/2 \rfloor$.

```
// Construct the binary heap.
template <class Comparable>
BinaryHeap<Comparable>::BinaryHeap( )
  : array( 11 ), theSize( 0 )
{
}
```





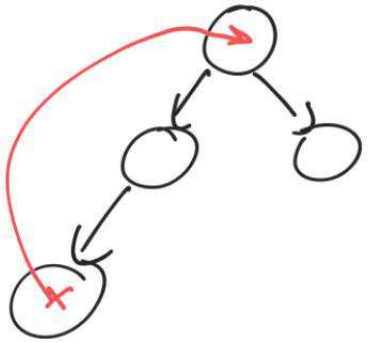
Min heap

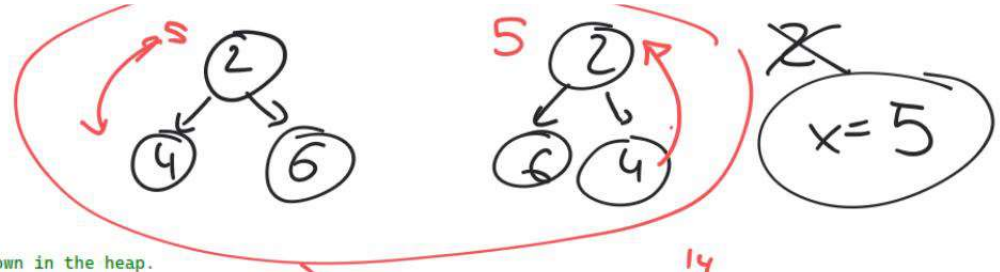
```
// Insert item x into the priority queue, maintaining heap order.
// Duplicates are allowed.
template <class Comparable>
void BinaryHeap<Comparable>::insert( const Comparable & x )
{
    array[ 0 ] = x;  // initialize sentinel
    if( theSize + 1 == array.size( ) )
        array.resize( array.size( ) * 2 + 1 );

      // Percolate up
    int hole = ++theSize;
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}
```

el contenido del padre

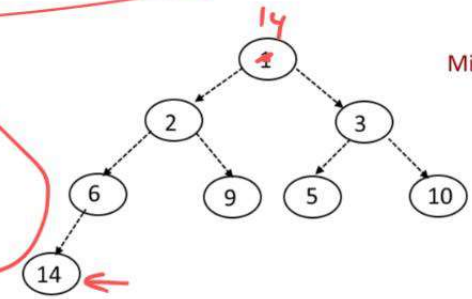
```
// Find the smallest item in the priority queue.
// Return the smallest item, or throw UnderflowException if empty.
template <class Comparable>
const Comparable & BinaryHeap<Comparable>::findMin( ) const
{
    if( isEmpty( ) )
        throw UnderflowException( );
    return array[ 1 ];
}
```

```
// Remove the smallest item from the priority queue.
// Throw UnderflowException if empty.
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin( )
{
    if( isEmpty( ) )
        throw UnderflowException( );

    array[ 1 ] = array[ theSize-- ];
    percolateDown( 1 );
}
```



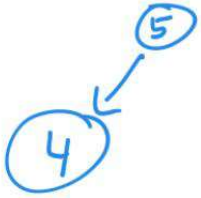



```
// Internal method to percolate down in the heap.
// hole is the index at which the percolate begins.
template <class Comparable>
void BinaryHeap<Comparable>::percolateDown( int hole )
{
    int child;
    Comparable tmp = array[ hole ];

    for( ; hole * 2 <= theSize; hole = child )
    {
        child = hole * 2;
        if( child != theSize && array[ child + 1 ] < array[ child ] )
            child++;
        if( array[ child ] < tmp )
            array[ hole ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}
```



Min Heap

0	1	2	3	4	5	6	7	8
	1	2	3	6	9	5	10	14



no hay hijo dcha

código marcado en azul se ejecuta

rojo no es cierto

```
// Test if the priority queue is logically empty.
// Return true if empty, false otherwise.
template <class Comparable>
bool BinaryHeap<Comparable>::isEmpty( ) const
{
    return theSize == 0;
}

// Make the priority queue logically empty.
template <class Comparable>
void BinaryHeap<Comparable>::makeEmpty( )
{
    theSize = 0;
}
```