Today:
Más sobre iteradores
Hash tables (Diccionarios)
Introducción a TADs no lineales

```
template <class Object>  <T> Provide sample template arguments f
class HashTable
{
  public:
    HashTable( );

    void makeEmpty( );
    void insert( const Object & x );
    void remove( const Object & x );
    Cref<Object> find( const Object & x ) const;
```

```
enum EntryType { ACTIVE, EMPTY, DELETED };
```

```
private:
  struct HashEntry
  {
      Object    element;
      EntryType info;

      HashEntry( const Object & e = Object( ), EntryType i = EMPTY )
        : element( e ), info( i ) { }
  };

  vector<HashEntry> array;
  int occupied;
```

Privado

```
    bool isActive( int currentPos ) const;
    int findPos( const Object & x ) const;
    void rehash( );
};

unsigned int hash( const string & key );
```

```
template <class Object>
HashTable<Object>::HashTable( )
  : array( nextPrime( 101 ) )
{
   makeEmpty( );
}

// Make the hash table logically empty.
template <class Object>
void HashTable<Object>::makeEmpty( )
{
    occupied = 0;
    for( int i = 0; i < array.size( ); i++ )
        array[ i ].info = EMPTY;
}
```

```
template <class Object>
void HashTable<Object>::insert( const Object & x )
{
      // Insert x as active
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        throw DuplicateItemException( );
    array[ currentPos ] = HashEntry( x, ACTIVE );

    if( ++occupied > array.size( ) / 2 )
        rehash( );
}
```

```cpp
// Return true if currentPos exists and is active.
template <class Object>
bool HashTable<Object>::isActive( int currentPos ) const
{
    return array[ currentPos ].info == ACTIVE;
}

////////////////////////////
```

```
// Remove item x from the hash table.
// Throw ItemNotFoundException if x is not present.
template <class Object>
void HashTable<Object>::remove( const Object & x )
{
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        array[ currentPos ].info = DELETED;
    else
        throw ItemNotFoundException( );
}
```

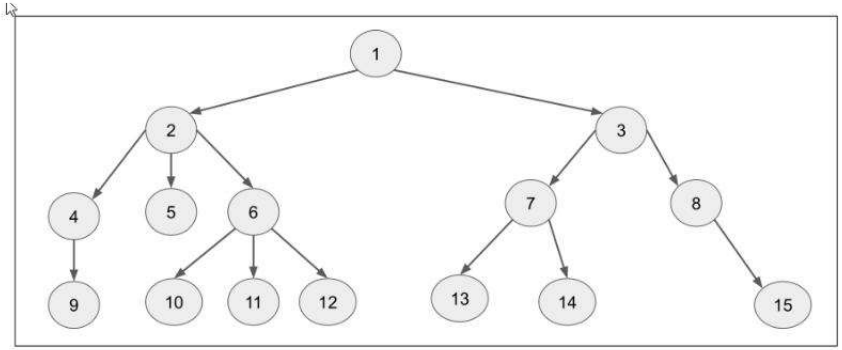
```
// Find item x in the hash table.
// Return the matching item, wrapped in a Cref object.
template <class Object>
Cref<Object> HashTable<Object>::find( const Object & x ) const
{
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        return Cref<Object>( array[ currentPos ].element );
    else
        return Cref<Object>( );
}

// Make the hash table logically empty
```

```
// Method that performs quadratic probing resolution.
// Return the position where the search for x terminates.
template <class Object>
int HashTable<Object>::findPos( const Object & x ) const
{
    int collisionNum = 0;
    int currentPos = ::hash( x ) % array.size( );

    while( array[ currentPos ].info != EMPTY &&
           array[ currentPos ].element != x )
    {
        currentPos += 2 * ++collisionNum - 1;  // Compute ith probe
        if( currentPos >= array.size( ) )
            currentPos -= array.size( );
    }

    return currentPos;
}
```

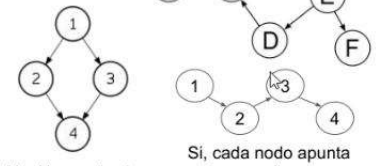
```cpp
// Generic hash function -- used if no other matches.
template <class Object>
unsigned int hash( const Object & key )
{
    unsigned int hashVal = 0;
    const char *keyp = reinterpret_cast<const char *>( & key );

    for( size_t i = 0; i < sizeof( Object ); i++ )
        hashVal =  ( hashVal << 5 ) ^ keyp[ i ] ^ hashVal;

    return hashVal;
}
```

```
// Expand the hash table.
template <class Object>
void HashTable<Object>::rehash( )
{
    vector<HashEntry> oldArray = array;

      // Create new double-sized, empty table
    array.resize( nextPrime( 2 * oldArray.size( ) ) );
    for( int j = 0; j < array.size( ); j++ )
        array[ j ].info = EMPTY;

      // Copy table over
    makeEmpty( );
    for( int i = 0; i < oldArray.size( ); i++ )
        if( oldArray[ i ].info == ACTIVE )
            insert( oldArray[ i ].element );
}
```

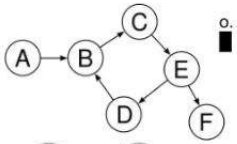
Árboles



¿Cuales son árboles?

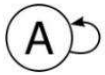


No. Hay un bucle

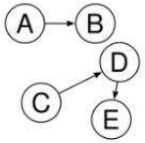


o. Hay un bucle

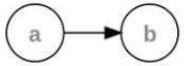
Si, cada nodo apunta a uno o varios y no hay bucles



No. Hay un bucle



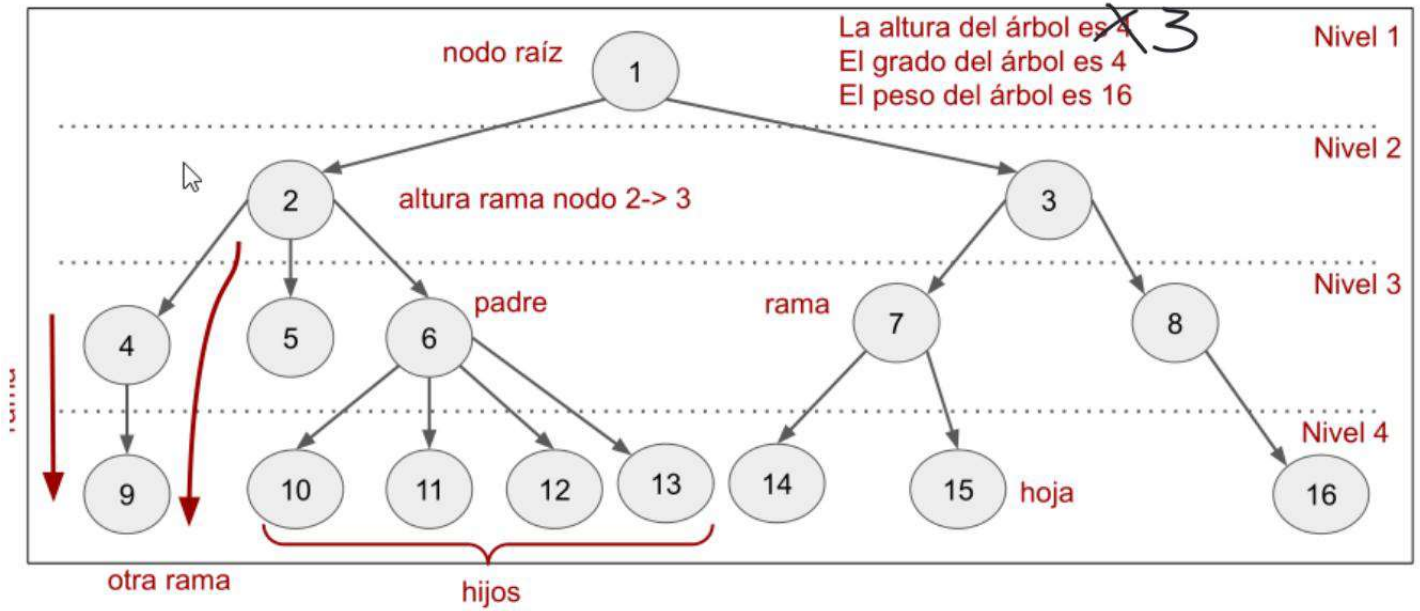
No. Hay 2 estructuras.



Si, el más simple

- Terminología:
  - Nodo: cada elemento que contiene un árbol
  - Padre: nodo del que parten otros nodos(más cerca de la raíz)
  - Hijo: nodos que parten de otro nodo (se alejan de la raíz)
  - Hermanos: nodos de un mismo padre
  - Raíz: El nodo superior (todos los nodos descienden de él y no tiene padre. Sólo puede haber un nodo Raíz)
  - Nodo Hoja: nodo sin hijos
  - Nodo Rama: nodo que no es raíz ni hoja y tienen al menos un hijo
  - Descendiente: Un nodo accesible por descenso repetido de padre a hijo
  - Ancestro: Un nodo accesible por ascenso repetido de hijo a padre.

- Terminología:
  - Bosque: conjunto de árboles disjuntos
  - Camino o rama: secuencia de nodos conectados con un nodo descendiente
  - Orden: nº máximo de hijos que puede tener un nodo
  - Grado: nº de hijos que tiene el elemento con más hijos dentro del árbol
  - Nivel: se define para cada elemento del árbol como la distancia a la raíz, medida en nodos.
  - Altura: nivel del nodo con mayor nivel
  - Peso: número de nodos que tiene un árbol


nodo raíz
1
La altura del árbol es 4 X 3
El grado del árbol es 4
El peso del árbol es 16
Nivel 1
Nivel 2
2
altura rama nodo 2-> 3
3
Nivel 3
padre
rama
4
5
6
7
8
Nivel 4
9
10
11
12
13
14
15
hoja
16
otra rama
hijos

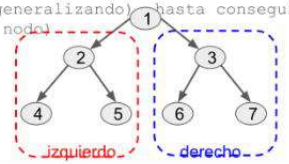
- Árboles binarios:
  - Recorridos:
    - En profundidad, se diferencia según donde se procese el dato:
      - Pre order (antes de recorrer)
      - In order (entre medias)
      - Post order (después de recorrer)
    - En anchura (o amplitud)

Árboles binarios de búsqueda (ABB)

- Conceptos de árboles:
  - Un árbol impone una representación jerárquica sobre los datos que contiene (raíz->nodos->hojas)
  - Ejemplos:
    - En la "vida":
      - Un árbol genealógico
      - El índice de un libro
      - etc
    - En informática:
      - Estructura de datos (para albergar un camino en un grafo)
      - Árbol de sintaxis (lo utilizan los compiladores)
      - Inteligencia artificial (árboles de decisión)
      - Algoritmos de búsqueda (a través del recorrido por el árbol)
      - Algoritmos de clasificación (ordenación)
      - etc

Árboles binarios de búsqueda (ABB)

- Conceptos de árboles:
  - Usaremos un árbol cuando queramos representar información jerarquizada que siempre converge en un punto
    - Punto de entrada: nodo raíz
    - Tendremos varios caminos hacia las hojas
  - Se trata de una estructura recursiva, cada nodo con sus descendientes constituye un subárbol. Dicho subárbol se va "simplificando" (generalizando) hasta conseguir el caso más sencillo (un sólo nodo)



Árboles binarios de búsqueda (ABB)

- ¿Por qué más tipos de árboles binarios? ¿ABB?
  - Buscar en una lista enlazada tiene complejidad O(n)
  - Con los ABB conseguimos una solución eficiente para realizar búsquedas eficientes en colecciones ordenadas de datos
    - Para buscar en listas enlazadas ¿tenemos que recorrerlas enteras? -> SI
    - ¿Y en un árbol binario "a secas"? -> TAMBIÉN (tenemos que recorrer todos los nodos)
    - Pero… si al crear el árbol seguimos algún criterio a la hora de insertar los nodos reduciremos el número de ramas a explorar :)

Árboles binarios de búsqueda

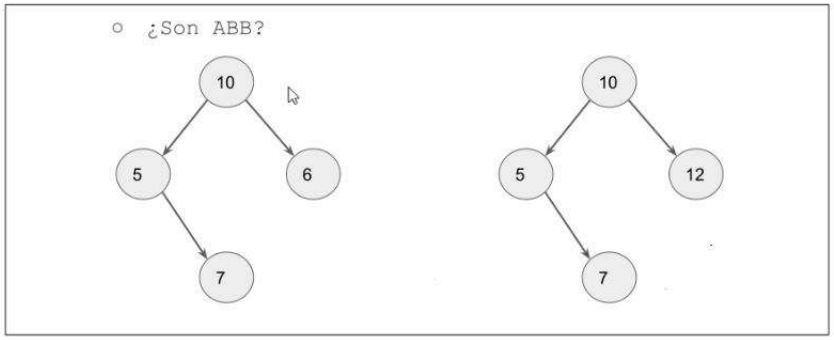
- TAD ABB: es un árbol binario en los que los nodos están ordenados:
  - Para cada nodo los valores del subárbol izquierdo son menores que su valor y los valores de la derecha son mayores.
- Todas las operaciones están basadas en la comparación, por lo que es necesario un procedimiento que compare nodos y establezca una relación de orden.
- Árbol Binario de Búsqueda=Binary Search Tree (BST)

Árboles binarios de búsqueda

- En un ABB (BST) los nodos deben cumplir:
  - Cada nodo debe tener asociado un tipo comparable. Pero puede tener mucha más información. (Por ejemplo la agenda del teléfono)
  - Para cada nodo el hijo izquierdo es menor que el padre.
  - Para cada nodo el hijo derecho es mayor que el padre

Árboles binarios de búsqueda

- ¿Son ABB?



Árboles binarios de búsqueda

- ¿Son ABB?