

Árboles binarios

Mauricio Avilés

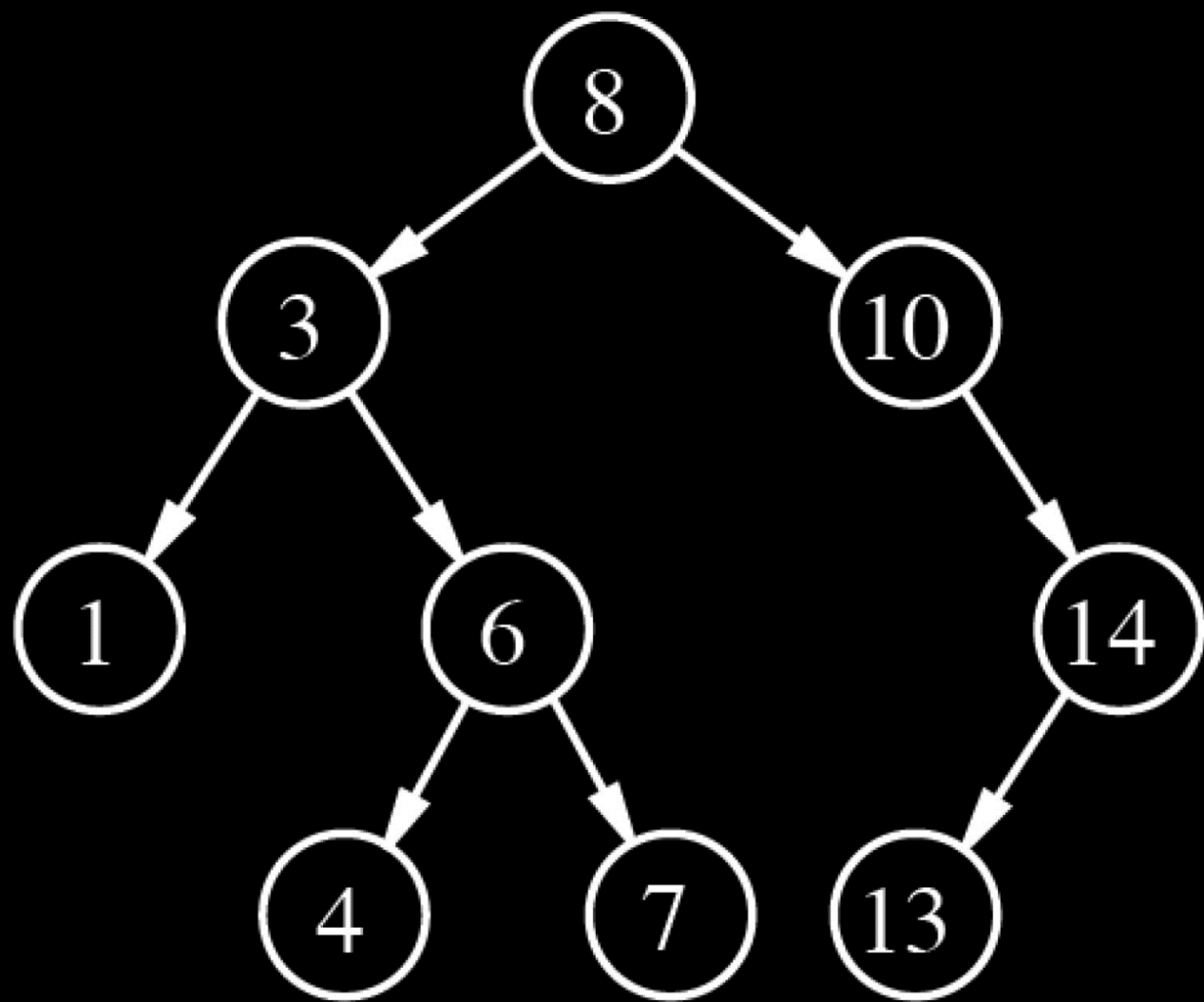
# Contenido

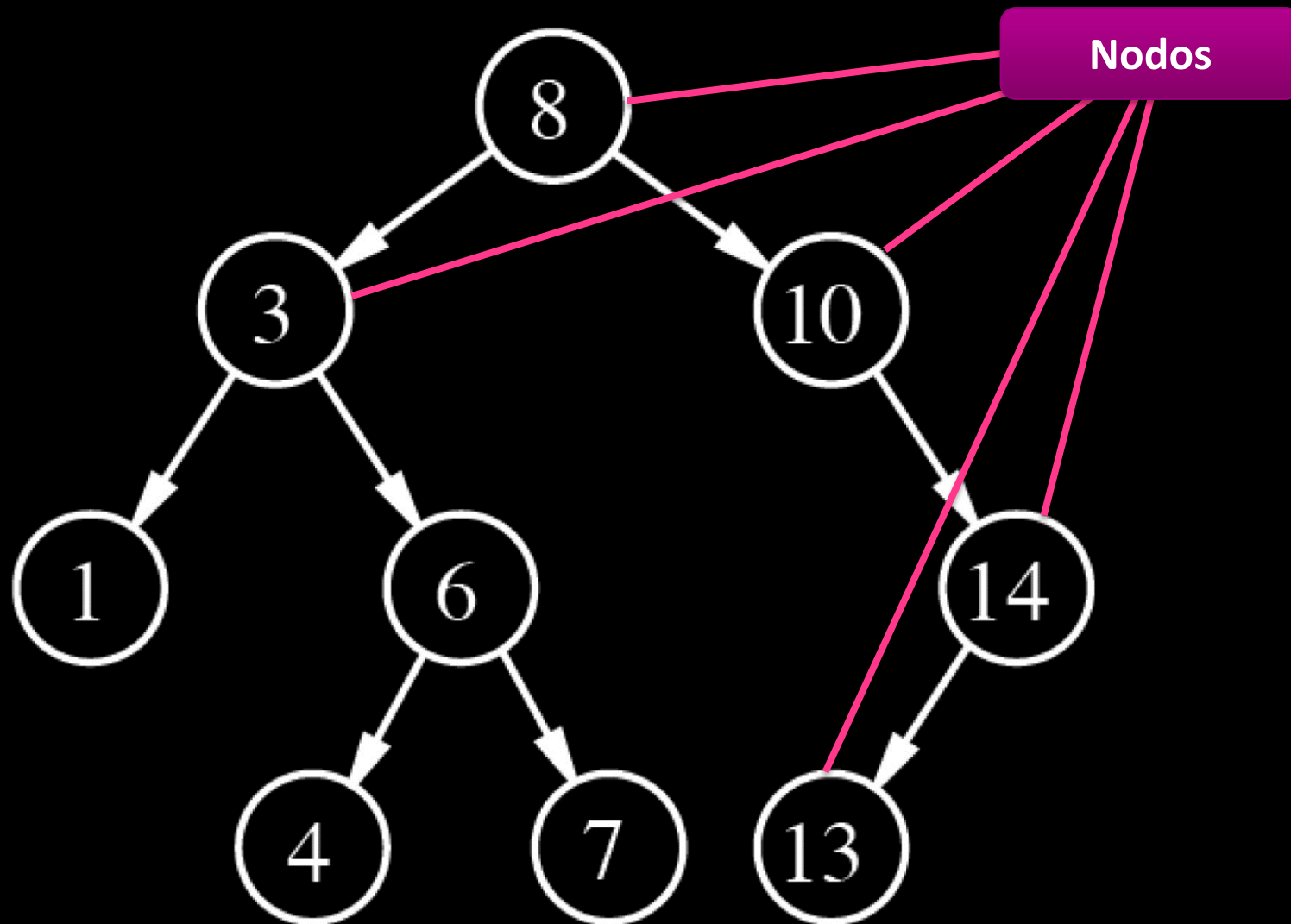
- Definiciones y propiedades
- Recorridos
- Implementación de nodos
- Árboles de búsqueda binaria
- Heaps o colas de prioridad

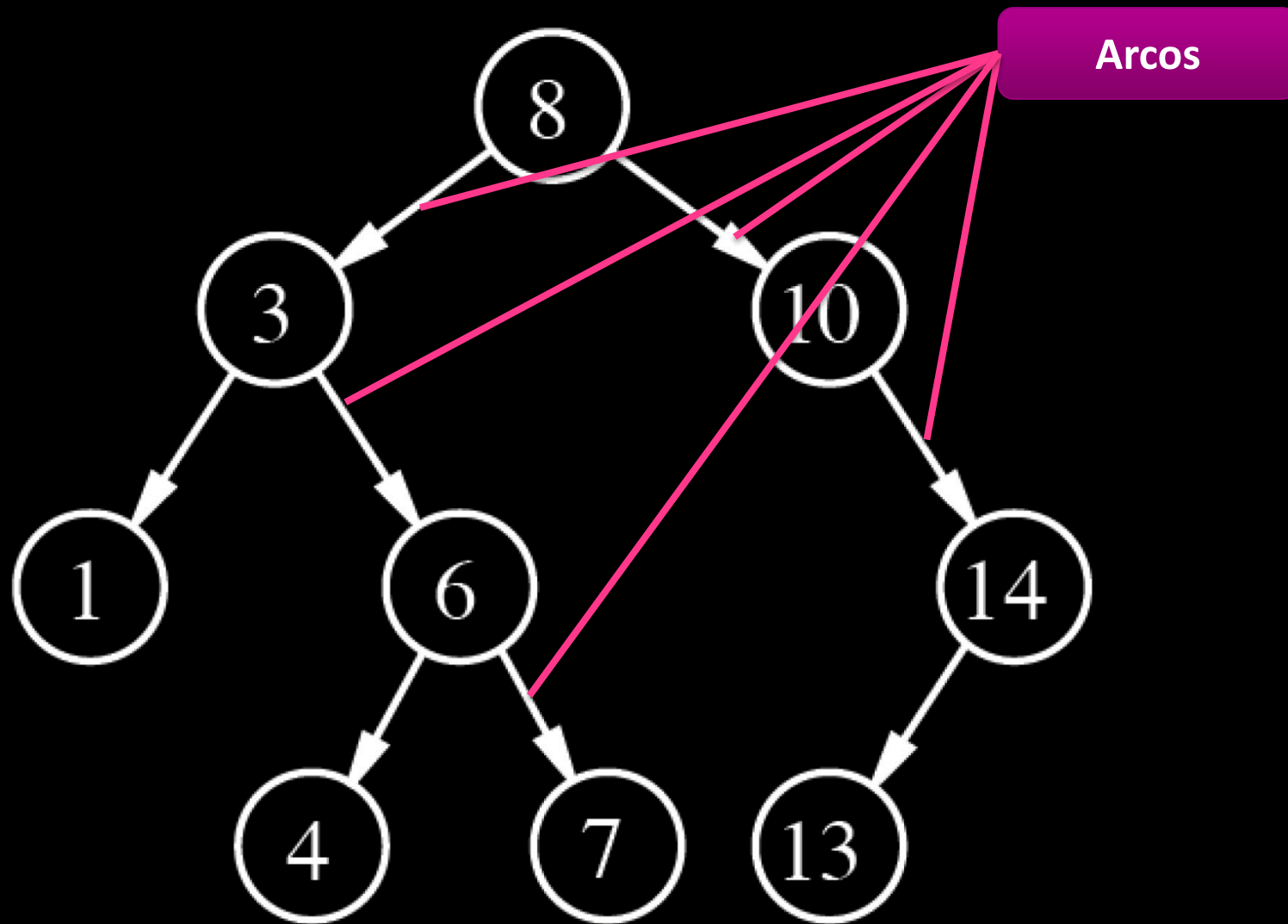
# Árboles binarios

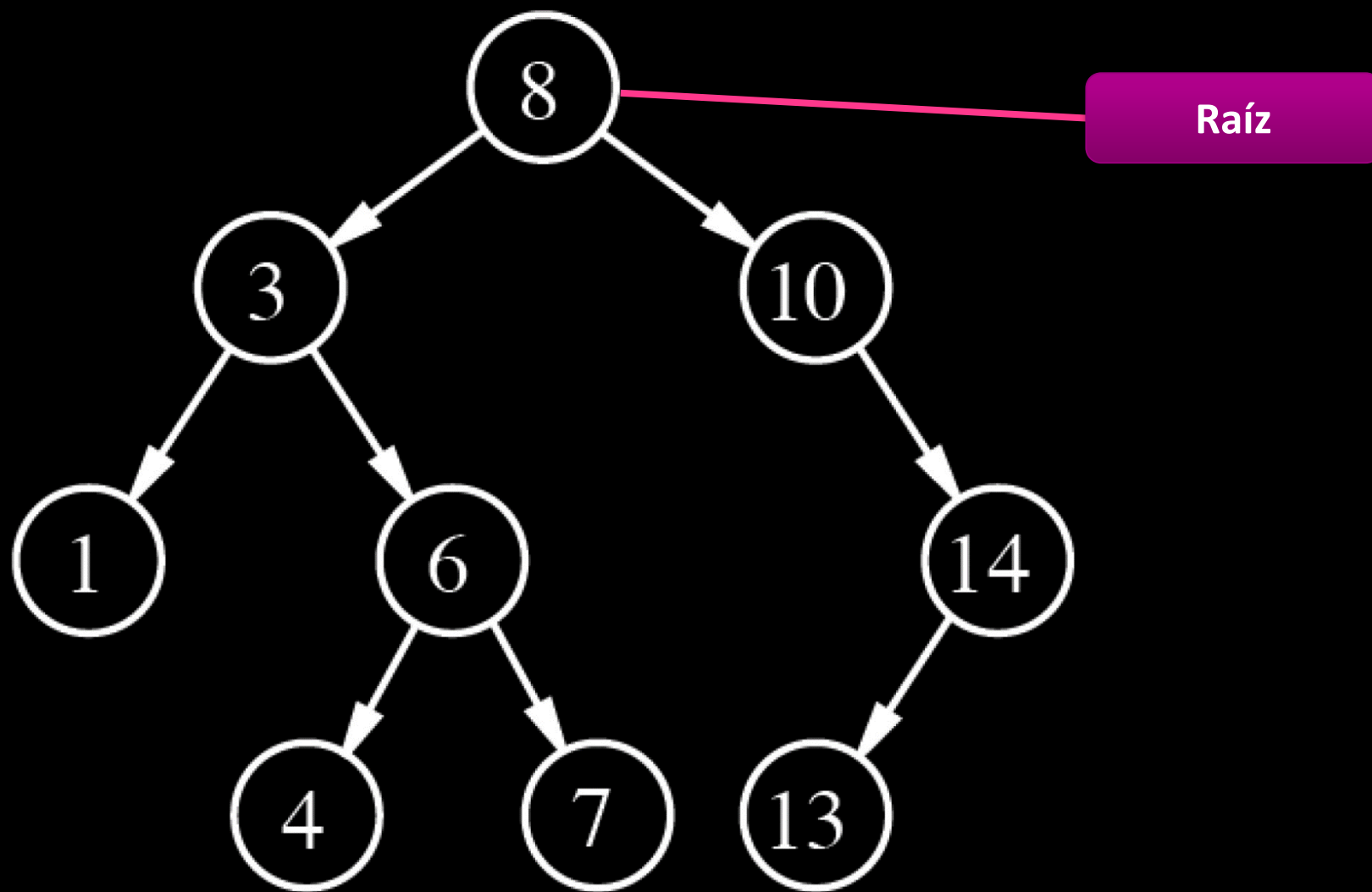
- Limitación fundamental de las listas: inserción, borrado y búsqueda eficientes, pero no todo
- Las estructuras de árbol permiten que esto sea eficiente
- Aplicaciones:
  - Priorización de trabajos
  - Descripción de expresiones matemáticas
  - Descripción de elementos sintácticos en lenguajes de programación
  - Algoritmos de compresión

- **Árbol binario**
  - Formado por un conjunto finito de **nodos**
  - El conjunto puede ser vacío, o
  - Consiste en un **nodo raíz**, que tiene como hijos dos sub-árboles binarios
  - Los **sub-árboles** son disjuntos entre ellos y del nodo raíz
  - Los nodos raíz de estos sub-árboles son los **hijos** del nodo raíz del árbol y este a su vez es el **padre** de ellos
  - Existe un **arco** que une a un nodo padre con su hijo

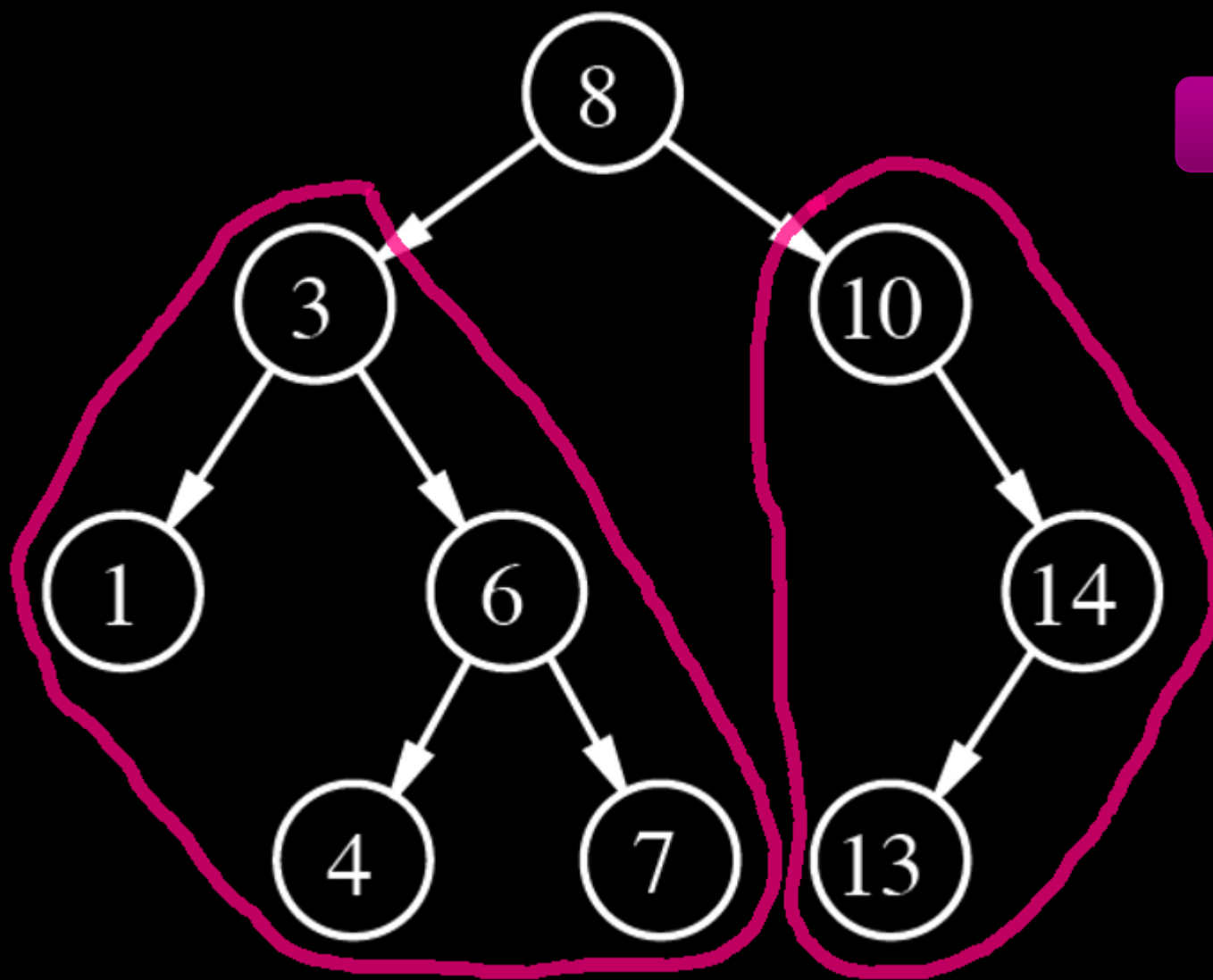




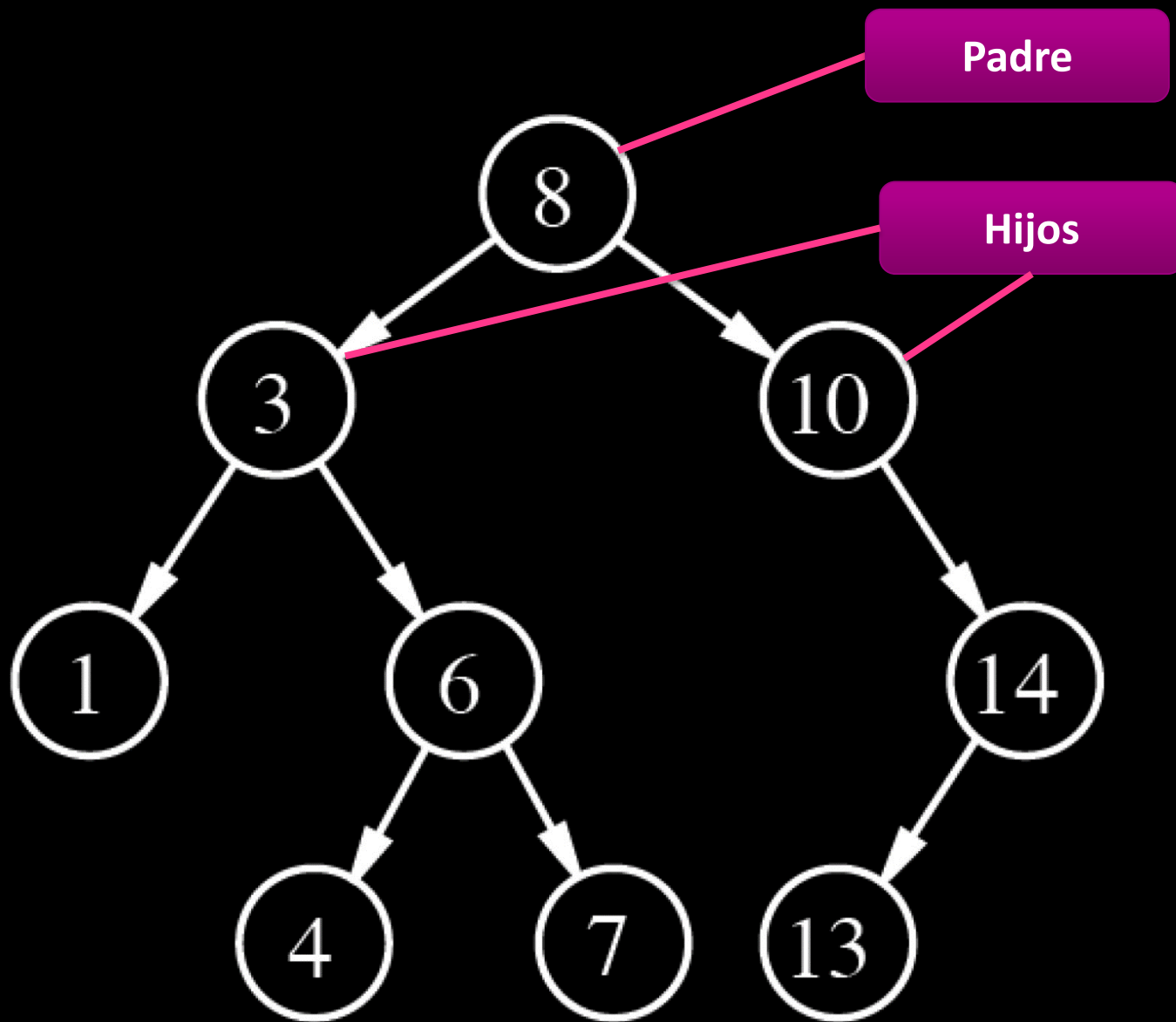




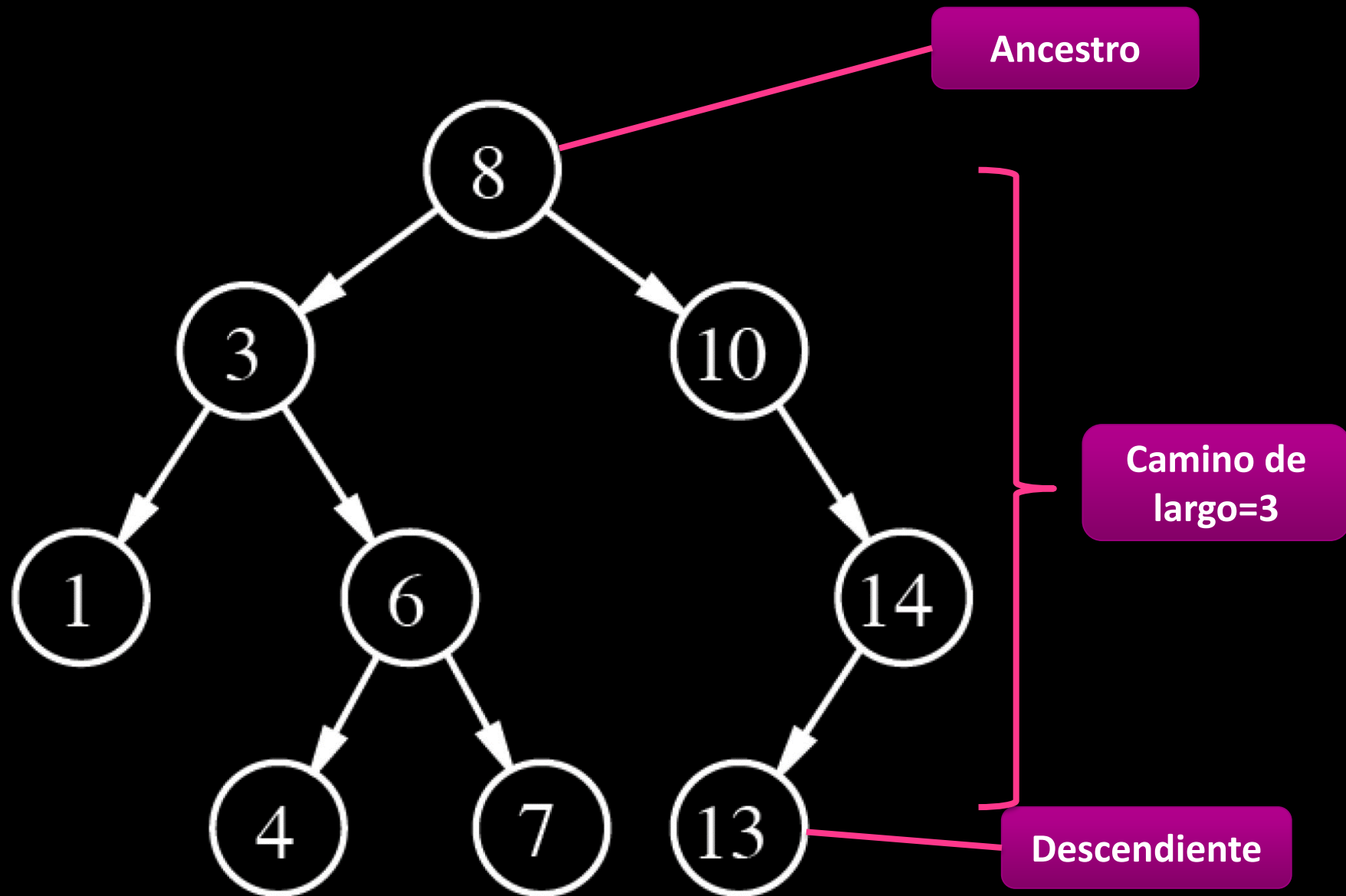




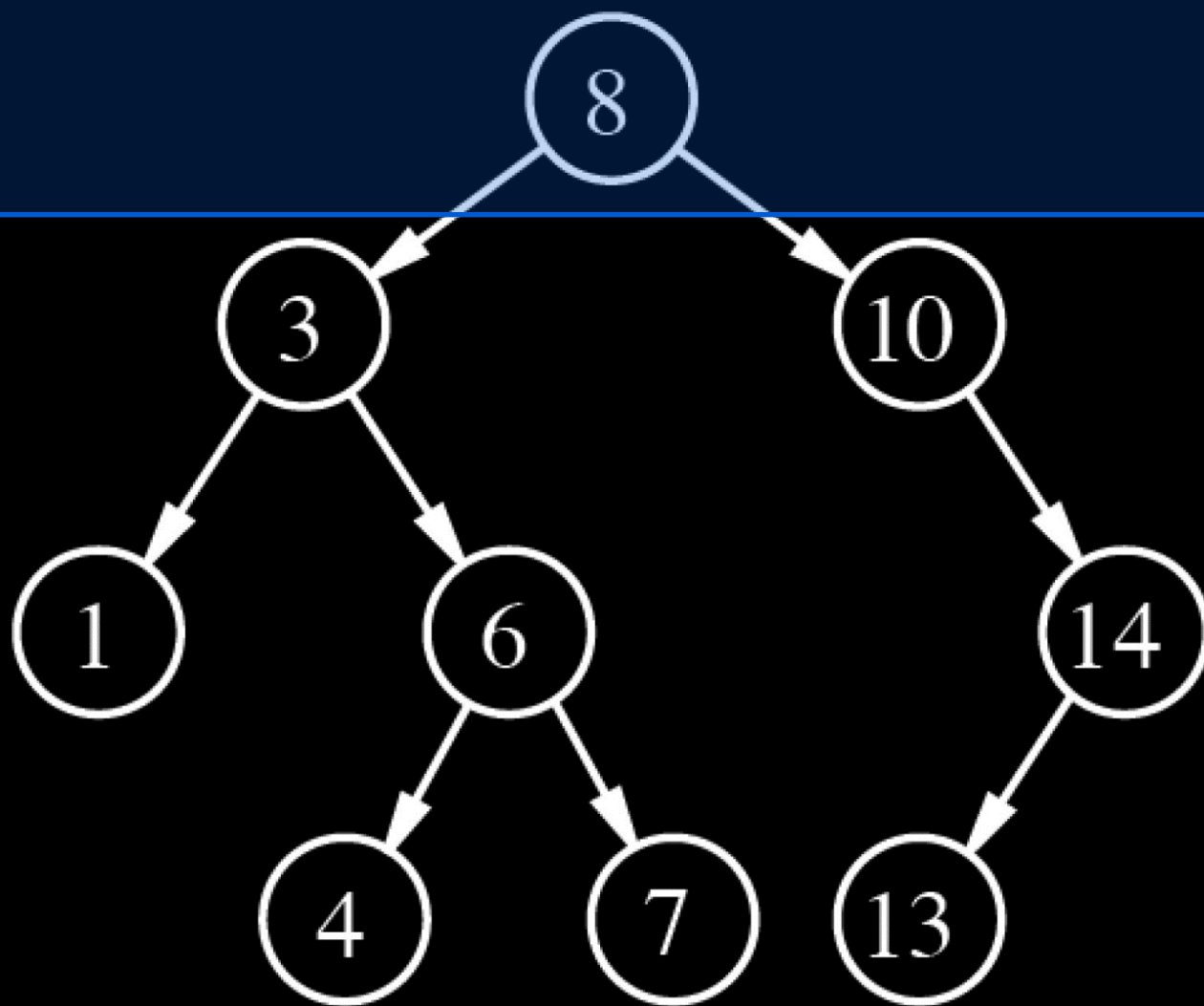
Sub-árboles



- Sea  $n_1, n_2, \dots, n_k$  es una secuencia de nodos en un árbol
- Si  $n_i$  es padre de  $n_{i+1}$  para todo  $1 \leq i \leq k$
- Entonces existe un **camino** o **ruta** desde  $n_1$  hasta  $n_k$
- El **largo** de la ruta es  $k-1$
- Si existe un camino de un nodo  $R$  a otro  $M$ , entonces
  - $R$  es **ancestro** de  $M$
  - $M$  es **descendiente** de  $R$
- Todos los nodos son descendientes de la raíz



- La **profundidad** de un nodo es el largo del camino desde la raíz
- La **altura** de un árbol es uno más la mayor profundidad de sus nodos
- Todos los nodos en una profundidad  $p$  se encuentran en el **nivel**  $p$  del árbol
- La raíz es el único nodo en el nivel 0
- Un **nodo hoja** no tiene hijos
- Un **nodo interno** tiene al menos un hijo

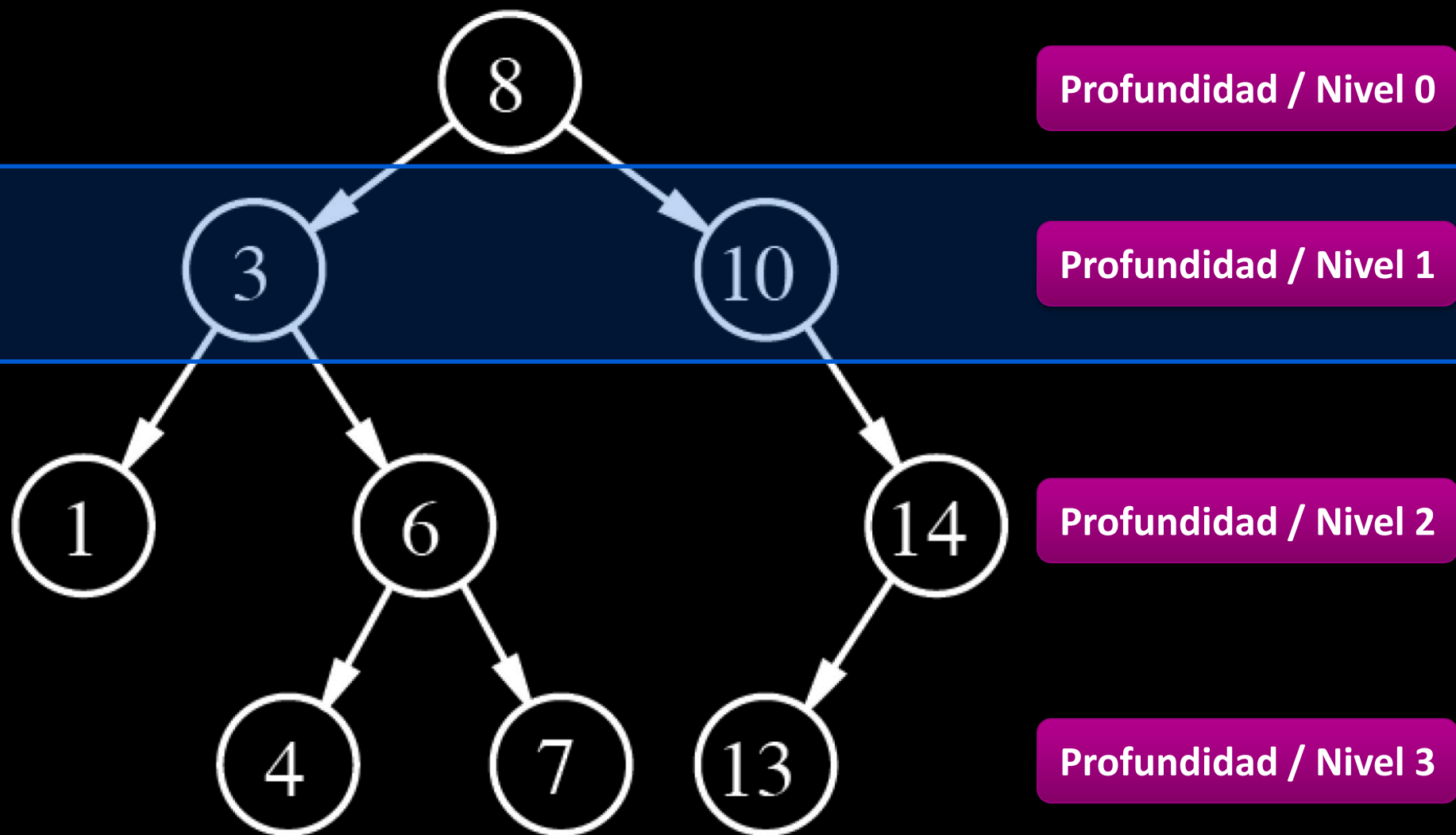


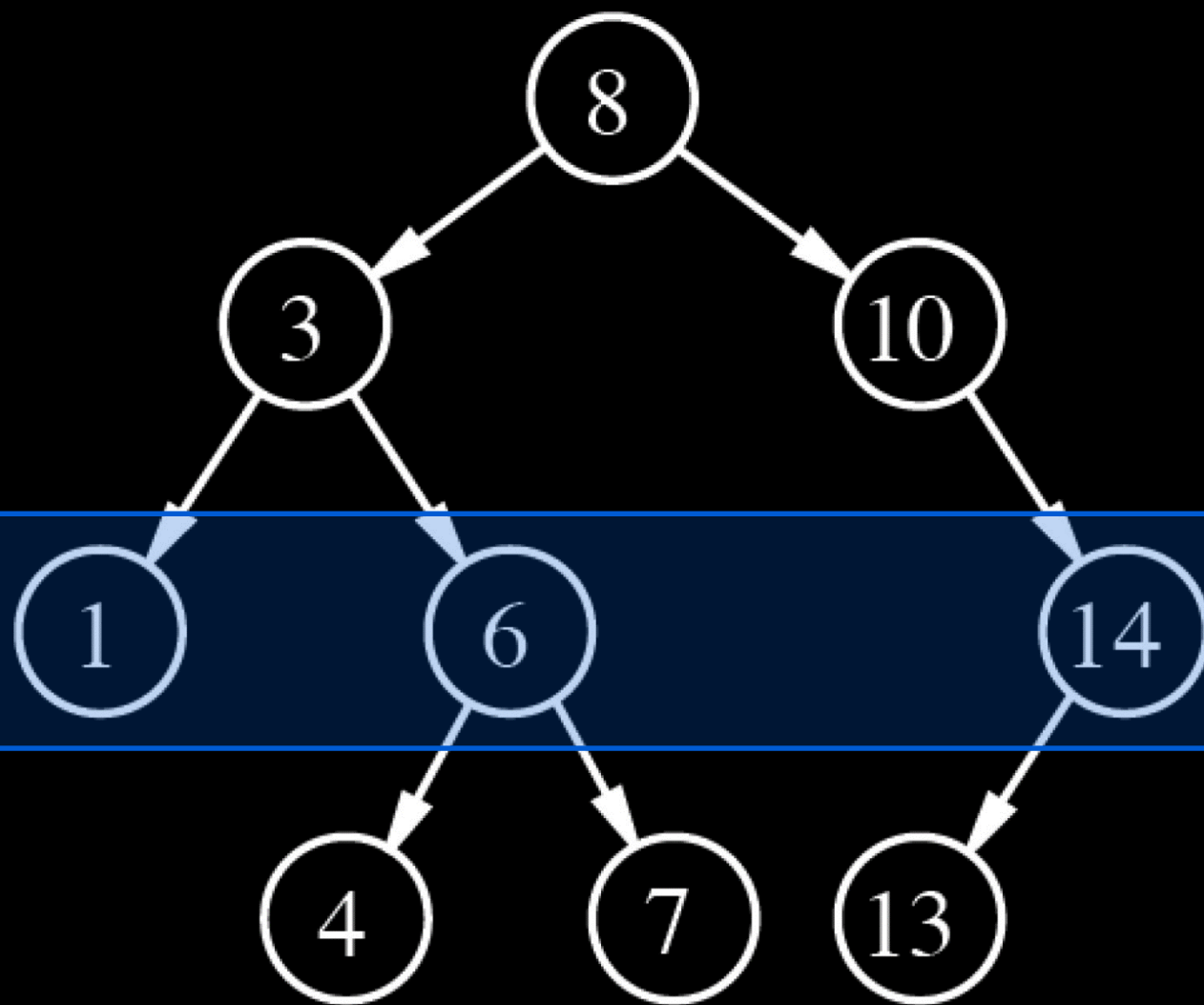
**Profundidad / Nivel 0**

**Profundidad / Nivel 1**

**Profundidad / Nivel 2**

**Profundidad / Nivel 3**





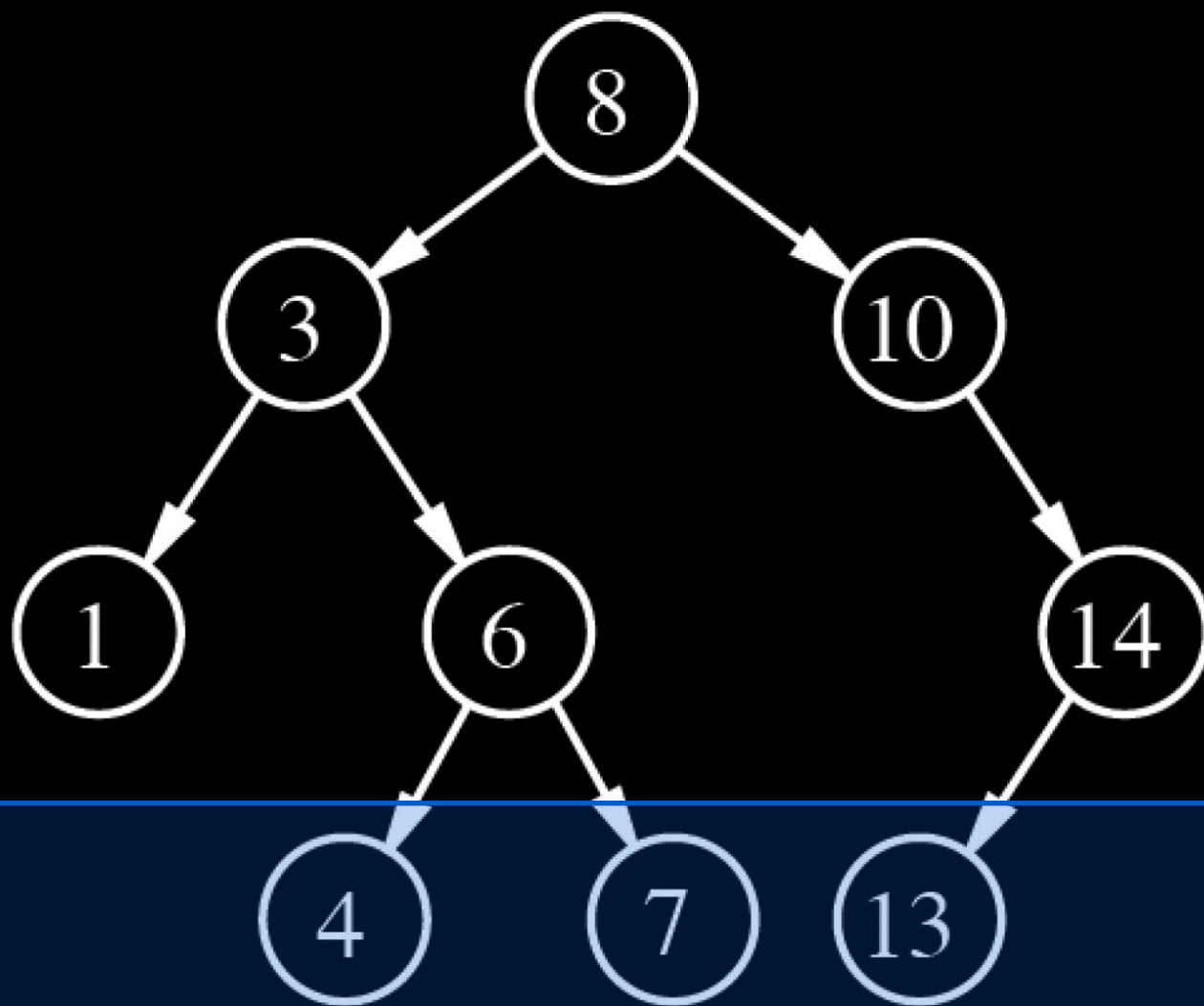
Profundidad / Nivel 0

Profundidad / Nivel 1

Profundidad / Nivel 2

Profundidad / Nivel 3



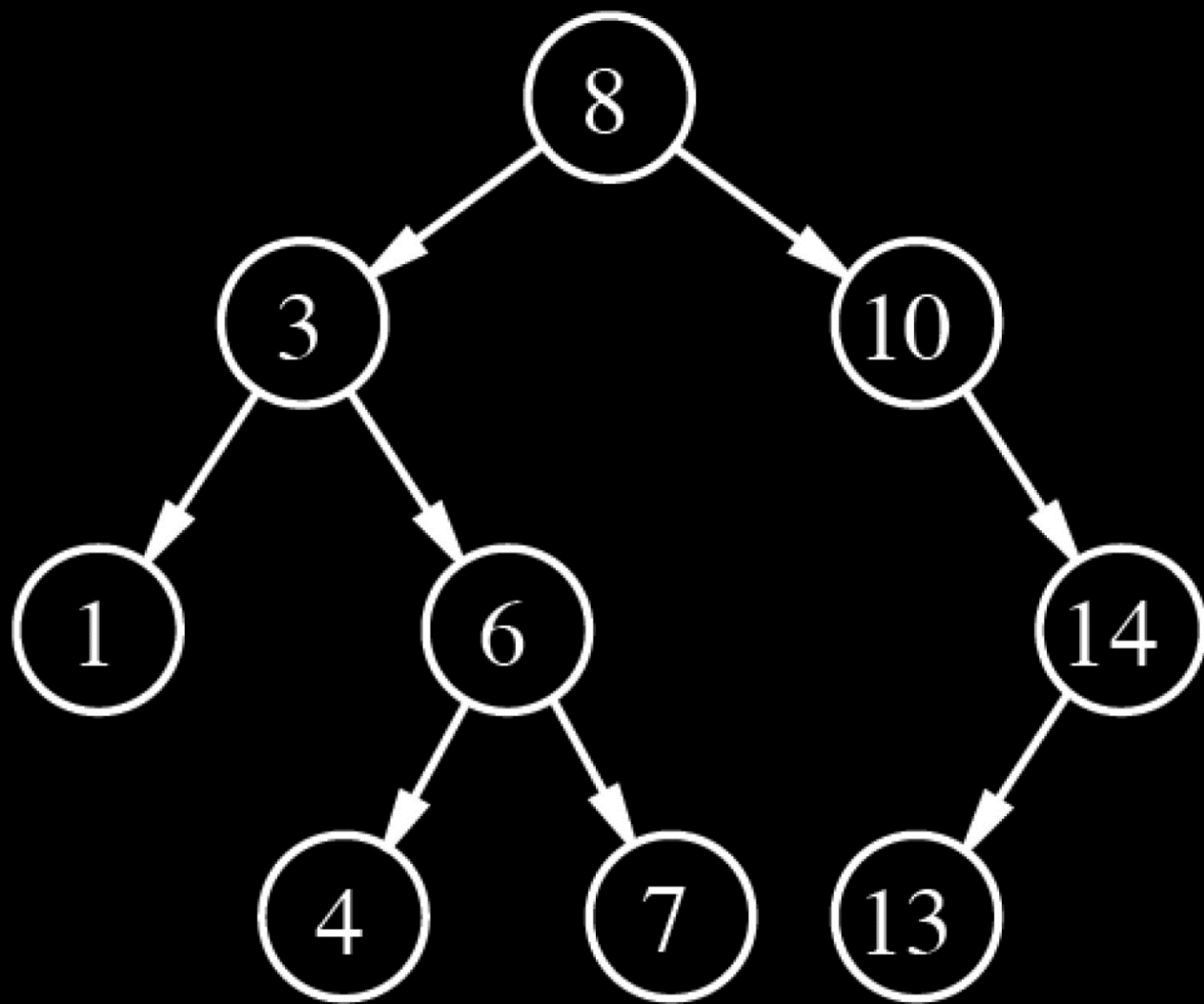


Profundidad / Nivel 0

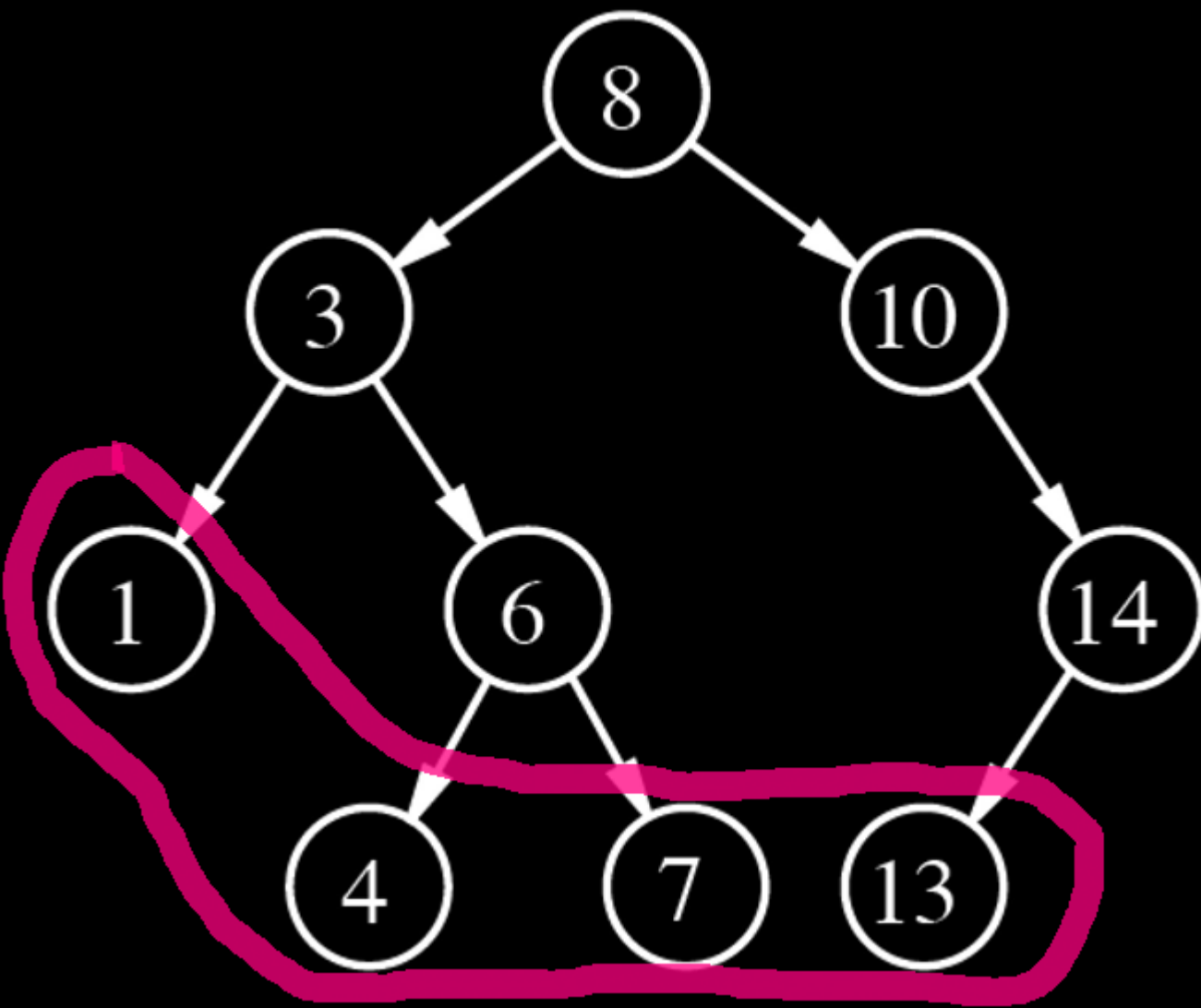
Profundidad / Nivel 1

Profundidad / Nivel 2

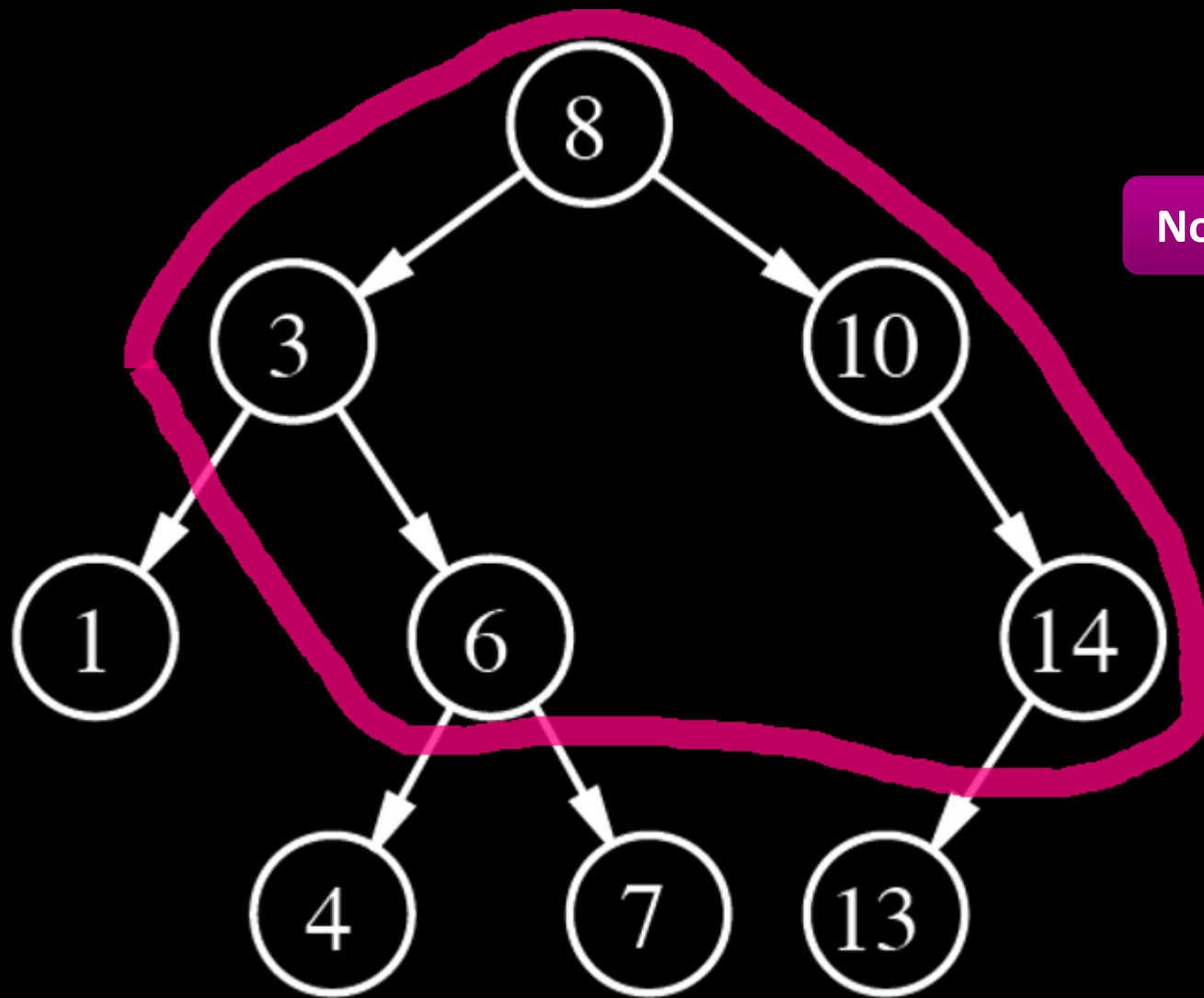
Profundidad / Nivel 3



Altura 4

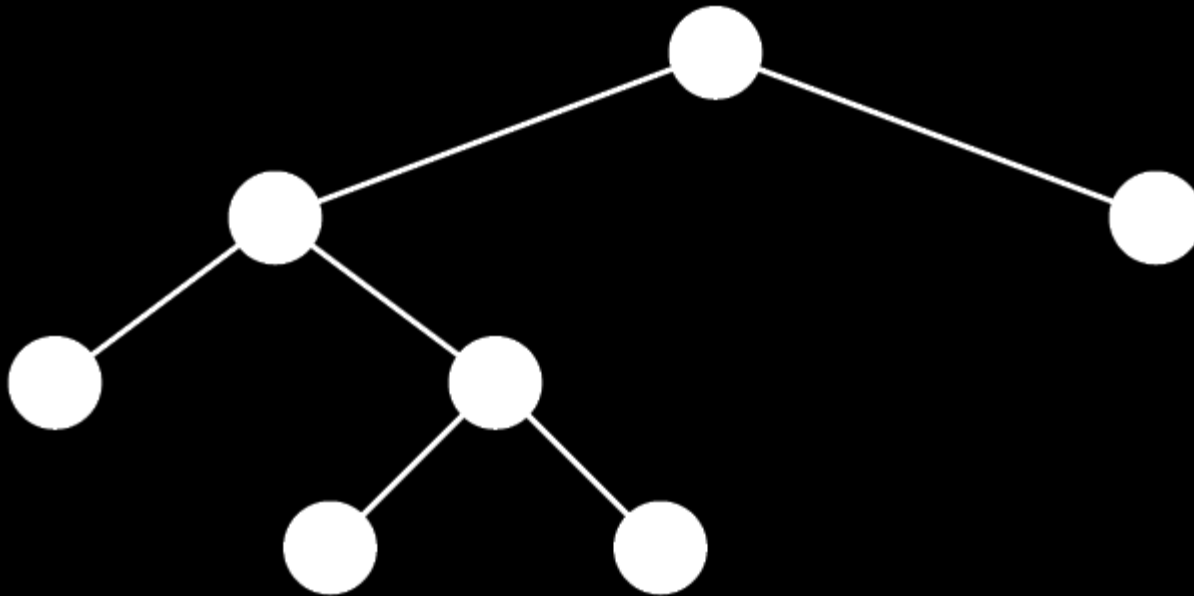


**Nodos hoja**

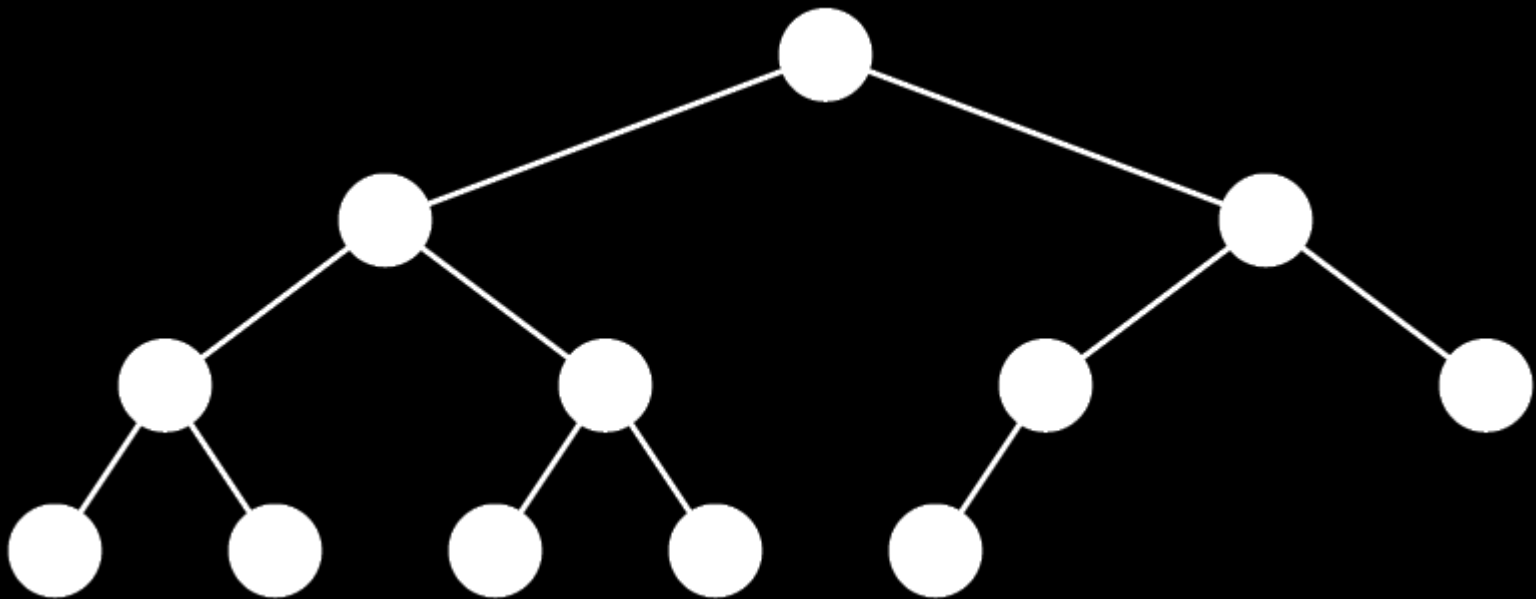


Nodos internos

- Un árbol binario **lleno** es aquel donde todos sus nodos son nodos internos con 2 hijos o un nodo hoja



- Un árbol binario **completo** es aquel donde todos los niveles están completamente llenos excepto por el último, cuyos nodos están agrupados del lado izquierdo



# Operaciones de los nodos

- Constructor y destructor
- Obtener elemento – E getElement()
- Asignar elemento – void setElement(E)
- Obtener hijo izquierdo – \* getLeft()
- Asignar hijo izquierdo – void setLeft(\*)
- Obtener hijo derecho – \* getRight()
- Asignar hijo derecho – void setRight(\*)
- Es hoja – bool isLeaf()

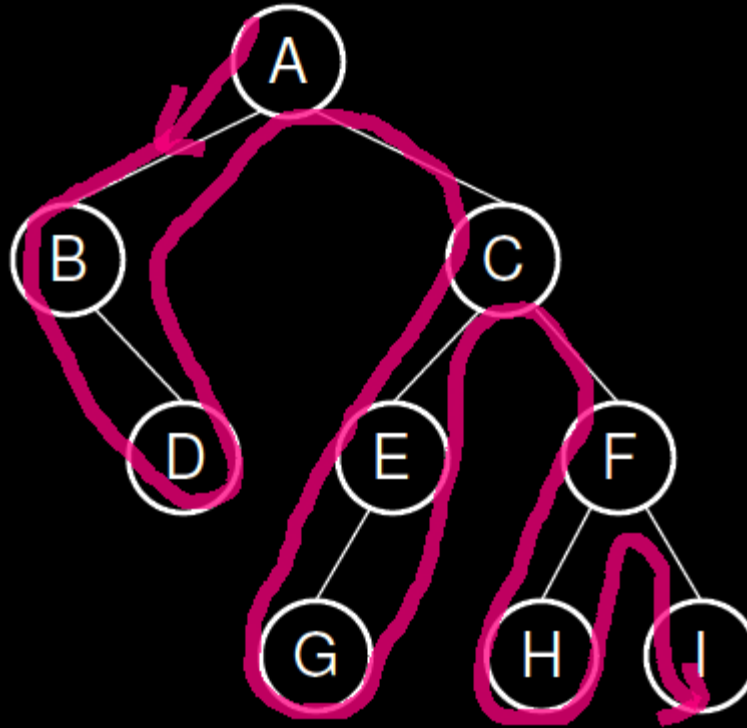
# Recorridos

- Pueden haber diferentes razones para recorrer un árbol, según el problema que se esté resolviendo
- El proceso de visitar cada uno de los nodos de un árbol se denomina **recorrido**
- Un recorrido que visita cada nodo exactamente una vez es una **enumeración** de los nodos



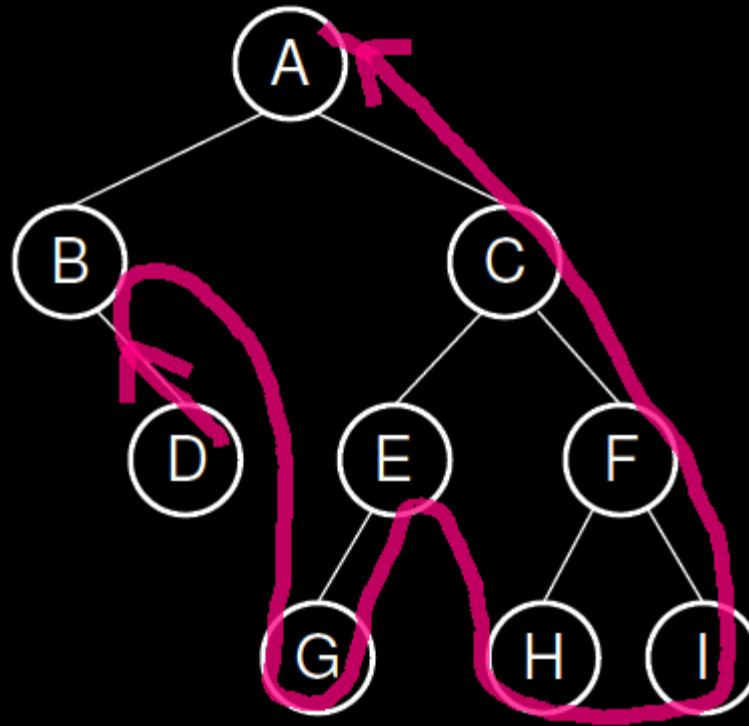
- Un recorrido que visita primero un nodo antes que sus hijos se llama recorrido en **preorden**
- Un recorrido que visita primero los hijos antes que el nodo se llama recorrido en **postorden**
- Un recorrido que visita primero el hijo izquierdo, luego el nodo y luego el hijo derecho se llama recorrido en **inorden**

# Recorrido en preorden



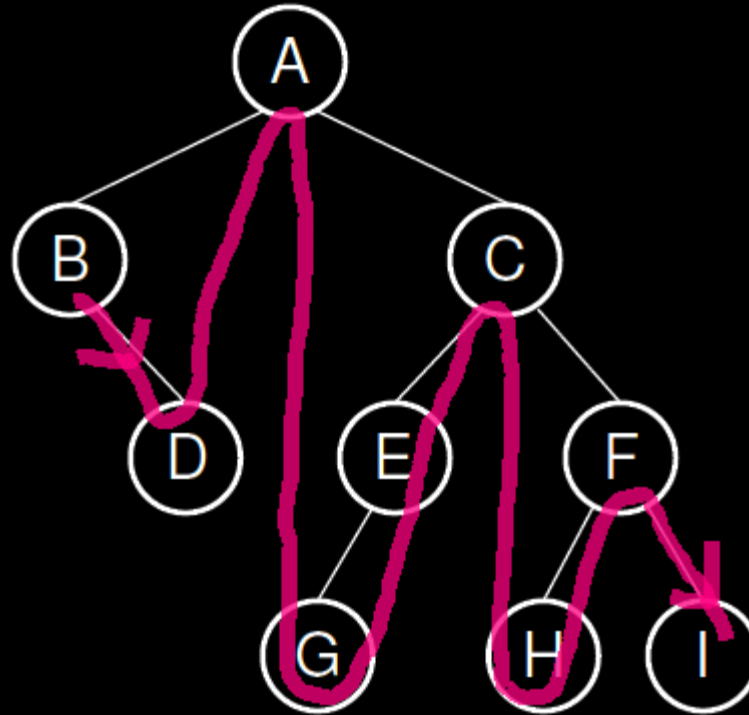
ABDCEGFHI

# Recorrido en postorden



DBGEHIFCA

# Recorrido en inorden

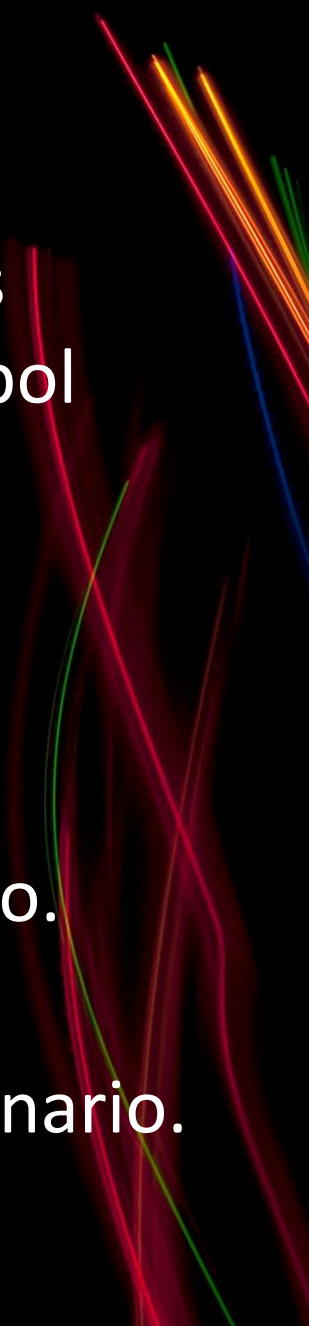


BDAGECHFI

```
void preorder(BinNode<E>* pRoot) {  
    if (pRoot == NULL) {  
        return;  
    }  
    visit(pRoot);  
    preorder(pRoot->getLeft());  
    preorder(pRoot->getRight());  
}
```

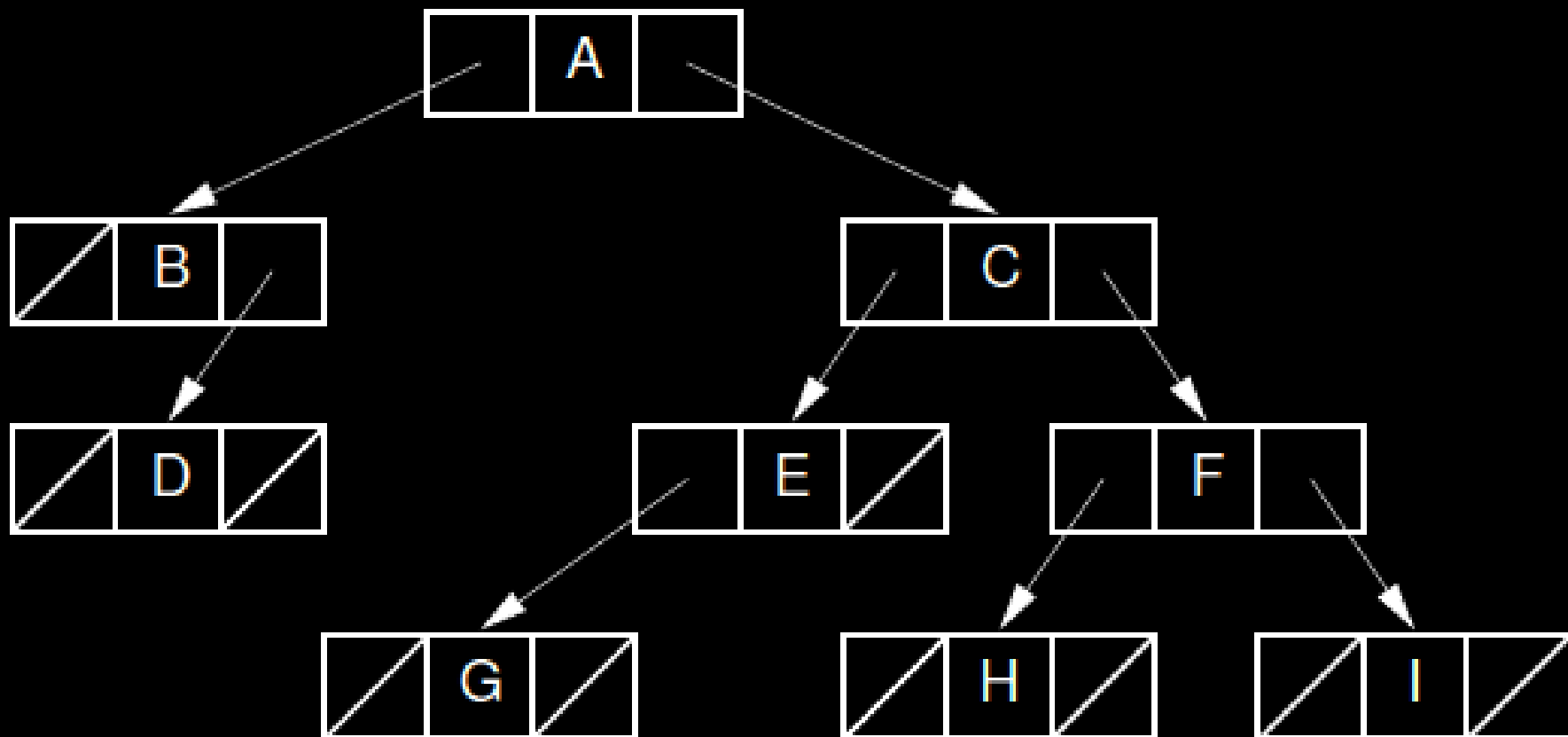
Función en C++ para realizar el recorrido en preorden

# Trabajo en clase

- Escribir funciones recursivas en C++ para los recorridos en **postorden** e **inorden** de un árbol binario.
  - Escribir una función en C++ para **contar** la cantidad de nodos en un árbol binario.
  - Escribir una función en C++ que cuente solamente los **nodos hoja** de un árbol binario.
  - Escribir una función en C++ que cuente solamente los **nodos internos** de un árbol binario.
- 
- A decorative graphic on the right side of the slide, consisting of several thin, overlapping lines in red, orange, yellow, and green, creating a sense of motion or a stylized flame.

# Implementación de nodos

- La implementación más común del nodo incluye:
  - Campo de **valor**
  - Dos punteros a los **hijos**
- Se usa una **llave** para poder ordenar los nodos y un valor que se guarda en cada nodo
- Es común agregar un puntero hacia el nodo padre, pero es innecesario → overhead





```
template <typename Key, typename E>
class BSTNode
{
private:
    Key key;
    E element;
    BSTNode<Key, E>* left;
    BSTNode<Key, E>* right;
```

```
    BSTNode(BSTNode<Key, E>* pLeft = NULL, BSTNode<Key, E>* pRight
= NULL) {
        left = pLeft;
        right = pRight;
    }
    BSTNode(Key pKey, E pElement, BSTNode<Key, E>* pLeft = NULL,
BSTNode<Key, E>* pRight = NULL) {
        key = pKey;
        element = pElement;
        left = pLeft;
        right = pRight;
    }
    ~BSTNode() {}
    E getElement() {
        return element;
    }
}
```

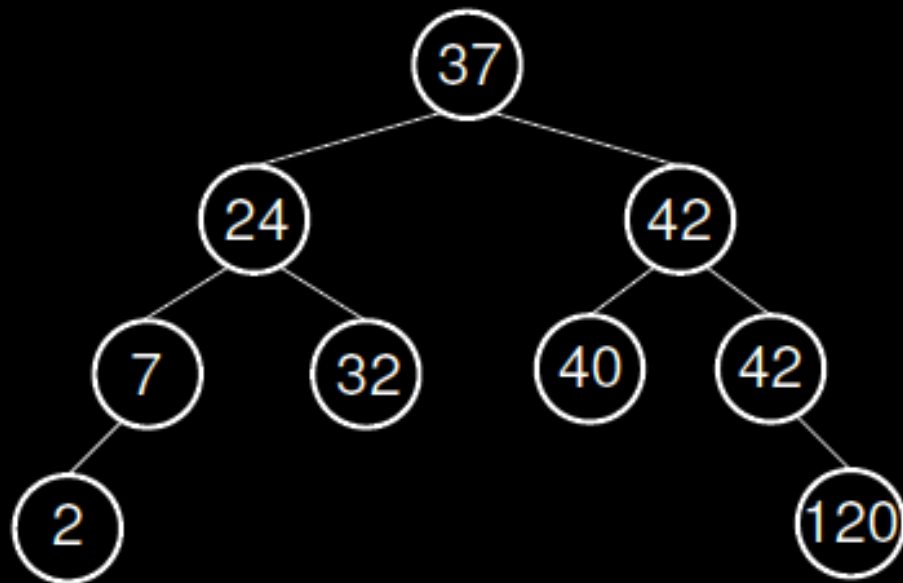
```
void setElement(E pElement) {
    element = pElement;
}
Key getKey() {
    return key;
}
void setKey(Key pKey) {
    key = pKey;
}
BSTNode<Key, E>* getLeft() {
    return left;
}
void setLeft(BSTNode<Key, E>* pLeft) {
    left = pLeft;
}
```

```
BSTNode<Key, E>* getRight() {  
    return right;  
}  
void setRight(BSTNode<Key, E>* pRight) {  
    right = pRight;  
}  
bool isLeaf() {  
    return (left == NULL) && (right == NULL);  
}
```

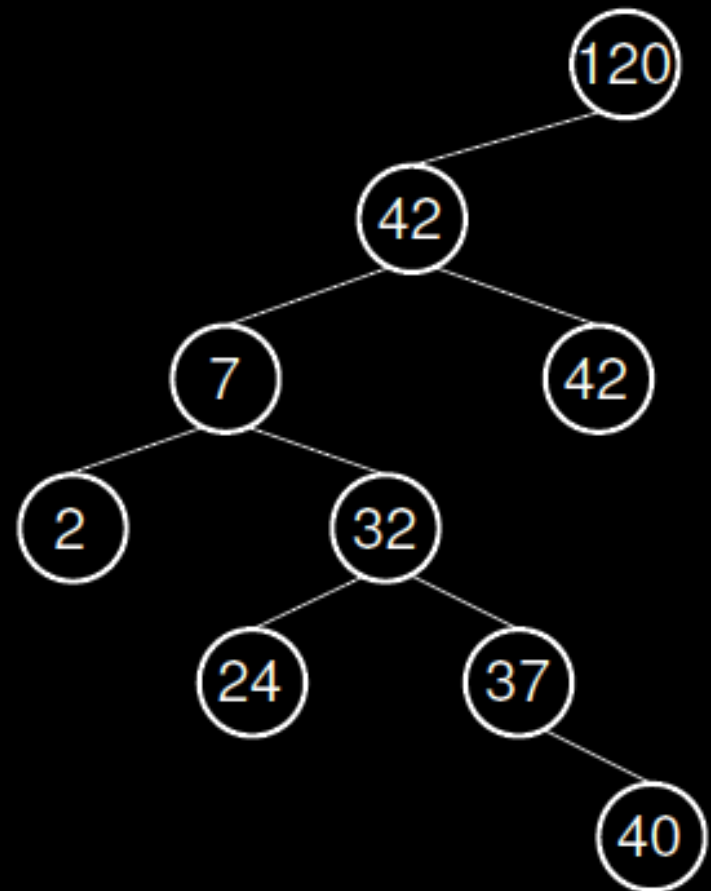
# Árboles de búsqueda binaria

- Son una forma de organizar una colección de registros que permite **búsqueda e inserción** rápida
- Un BST (binary search tree) es un árbol binario que cumple lo siguiente:
  - Todos los nodos en el subárbol izquierdo de un nodo con llave  $K$  tienen valores de llave **menores** que  $K$
  - Todos los nodos en el subárbol derecho de un nodo con llave  $K$  tienen valores **mayores o iguales** que  $K$

[37, 24, 42, 7, 2, 40, 42, 32, 120]



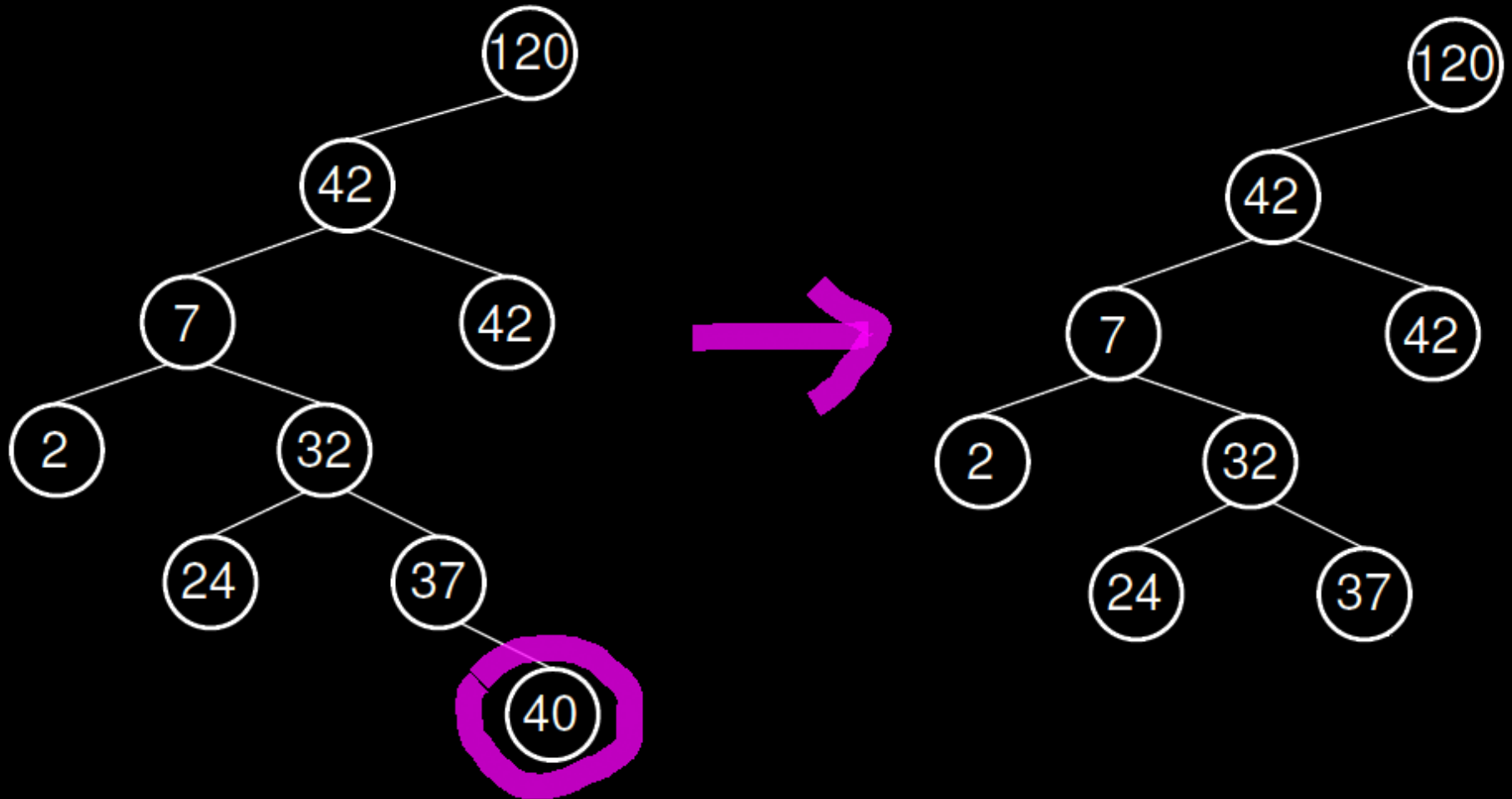
[120, 42, 42, 7, 2, 32, 37, 24, 40]



# Borrado de un nodo

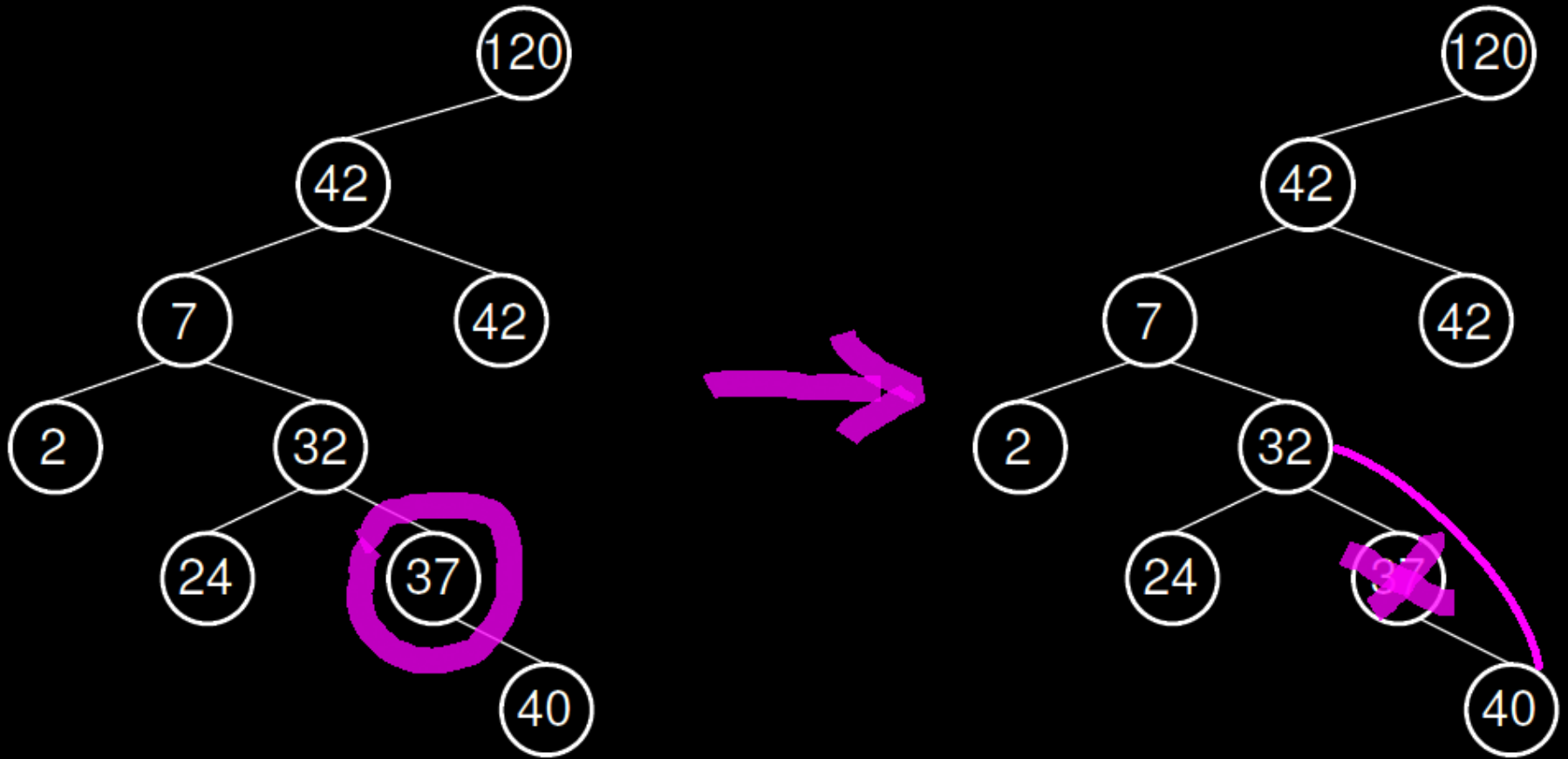
- El borrado de un nodo no es tan trivial como la búsqueda o la inserción
- Tres casos:
  - Si el nodo **no tiene hijos**
  - Si tiene **un hijo**
  - Si tiene **dos hijos**

# Caso 1: nodo sin hijos – Eliminar 40

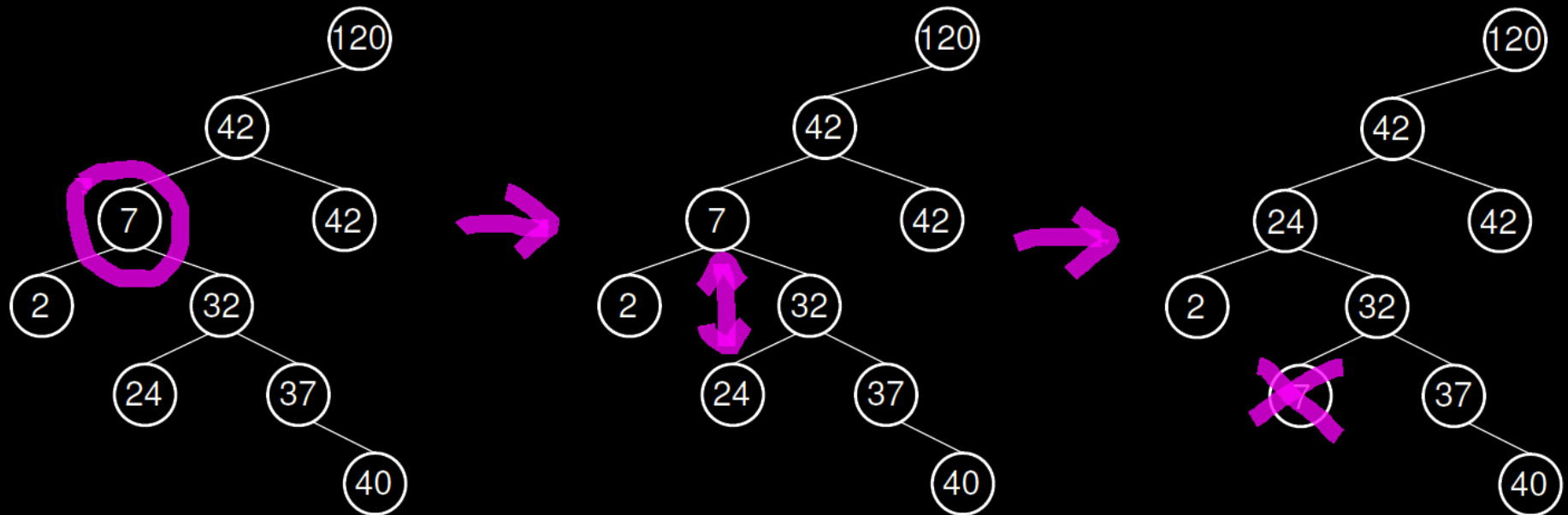




## Caso 2: nodo con un hijo – Eliminar 37



# Caso 3: nodo con dos hijos – Eliminar 7



- Se busca el nodo que se desea eliminar (R)
- Si R no tiene hijos
  - Se asigna NULL al puntero del padre
- Si el nodo tiene un hijo
  - Se asigna al puntero del padre el hijo de R
- Si el nodo tiene dos hijos
  - Se busca el menor nodo en el subárbol derecho de R (S)
  - Se intercambia el valor de S con R
  - Se elimina el nodo según los casos anteriores
- Se elimina R de memoria

# Ejercicios

- Dibuje los árboles generados por las siguientes inserciones. Realice los **borrados** indicados y dibuje el árbol obtenido después de cada borrado.
  - Ins.: [62, 25, 87, 46, 4, 84, 46, 53, 48, 68]  
Bor.: [46, 46, 25, 62]
  - Ins.: [36, 61, 27, 3, 18, 8, 84, 21, 20, 57]  
Bor.: [18, 36, 27, 61]

# Ejercicios

- Dibuje los árboles generados por las siguientes inserciones. Realice los **borrados** indicados y dibuje el árbol obtenido después de cada borrado.
  - Ins.: [62, 25, 87, 46, 4, 84, 46, 53, 48, 68] Bor.: [46, 46, 25, 62]
  - Ins.: [36, 61, 27, 3, 18, 8, 84, 21, 20, 57] Bor.: [18, 36, 27, 61]
- Analice el código fuente de las siguientes funciones (Shaffer, p. 174-177) y **escriba el pseudocódigo** que explique el funcionamiento de cada una de ellas:
  - deletemin: borra el nodo con menor llave en un BST
  - getmin: retorna un puntero al nodo con menor llave en un BST
  - removehelp: busca un nodo con una llave dada y lo elimina de un BST
- Elabore una **diapositiva** por cada función que explique el funcionamiento de su código fuente
- Explique:
  - ¿Porqué al eliminar un nodo con dos hijos, se busca el nodo menor del subárbol derecho? ¿Es posible hacerlo también buscando el nodo mayor del subárbol izquierdo?

```
template <typename Key, typename E>
class BSTree
{
private:
    BSTNode<Key, E>* root;
```

```
public:
    E find(Key pKey) throw (runtime_error) {
        return findAux(root, pKey);
    }

private:
    E findAux(BSTNode<Key, E>* pRoot, Key pKey) throw (runtime_error) {
        if (pRoot == NULL) {
            throw runtime_error("Key not found.");
        }
        if (pKey == pRoot->getKey()) {
            return pRoot->getElement();
        }
        if (pKey < pRoot->getKey()) {
            return findAux(pRoot->getLeft(), pKey);
        } else {
            return findAux(pRoot->getRight(), pKey);
        }
    }
}
```

```
public:
    void insert(Key pKey, E pElement) {
        root = insertAux(root, pKey, pElement);
    }
private:
    BSTNode<Key, E>* insertAux(BSTNode<Key, E>* pRoot, Key pKey, E
pElement) {
        if (pRoot == NULL) {
            return new BSTNode<Key, E>(pKey, pElement);
        }
        if (pKey < pRoot->getKey()) {
            pRoot->setLeft(insertAux(pRoot->getLeft(), pKey,
pElement));
            return pRoot;
        } else {
            pRoot->setRight(insertAux(pRoot->getRight(), pKey,
pElement));
            return pRoot;
        }
    }
}
```



```
public:
    E remove(Key pKey) {
        E result = find(pKey);
        root = removeAux(root, pKey);
        return result;
    }

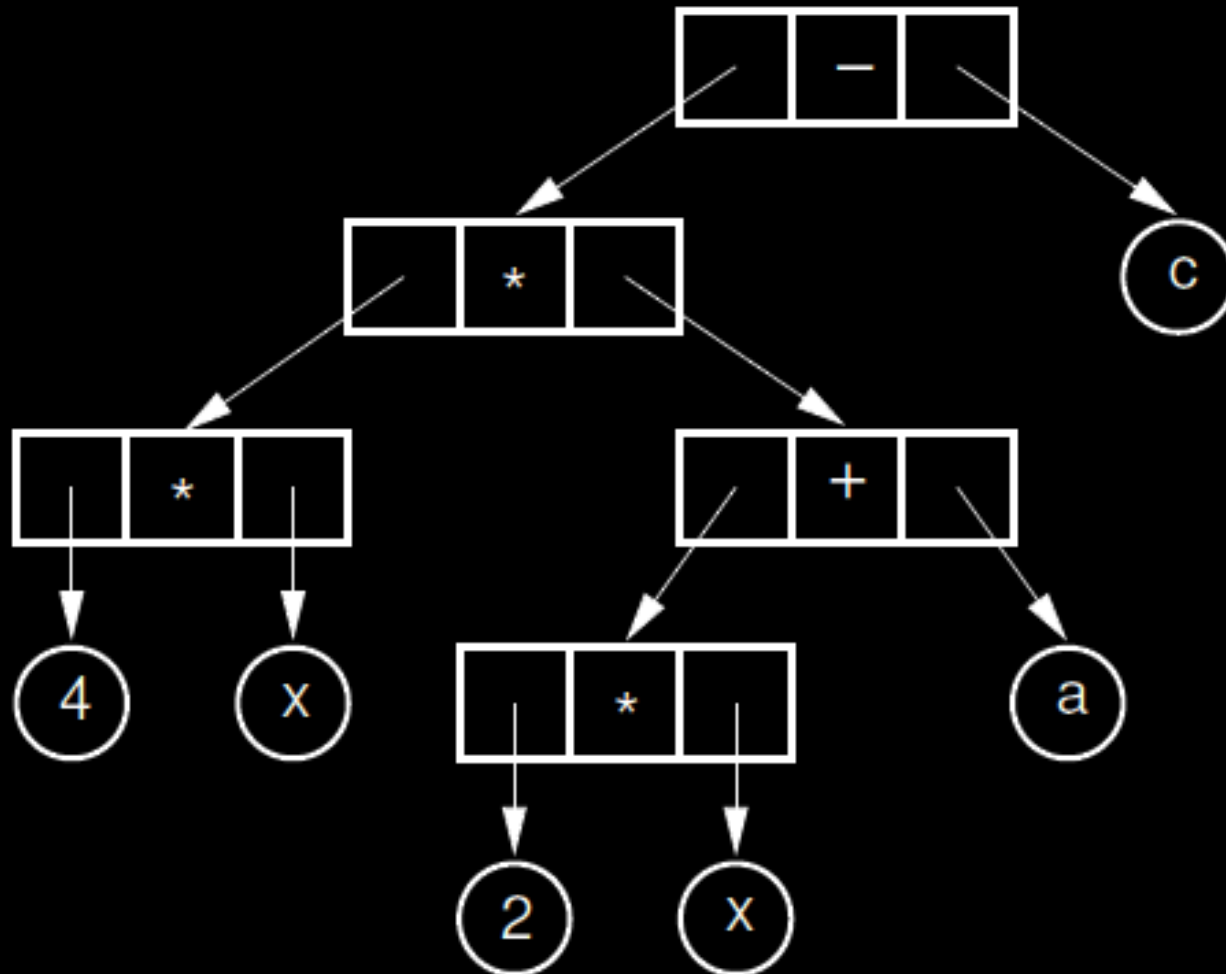
private:
    BSTNode<Key, E>* removeAux(BSTNode<Key, E>* pRoot, Key pKey) throw
(runtime_error) {
    if (pRoot == NULL) {
        throw runtime_error("Key not found.");
    }
    if (pKey < pRoot->getKey()) {
        pRoot->setLeft(removeAux(pRoot->getLeft(), pKey));
        return pRoot;
    }
    if (pKey > pRoot->getKey()) {
        pRoot->setRight(removeAux(pRoot->getRight(), pKey));
        return pRoot;
    } else { ...
```

```
...    if (pRoot->isLeaf()) {
        delete pRoot;
        return NULL;
    }
    if (pRoot->childrenCount() == 1) {
        BSTNode<Key, E>* temp = pRoot->getUniqueChild();
        delete pRoot;
        return temp;
    } else {
        BSTNode<Key, E>* successor = pRoot->getSuccessor();
        swap(pRoot, successor);
        if (pRoot->getRight() == successor) {
            pRoot->setRight(removeAux(successor, pKey));
        } else {
            BSTNode<Key, E>* succesorParent = pRoot->getRight();
            while (succesorParent->getLeft() != successor) {
                succesorParent = succesorParent->getLeft();
            }
            succesorParent->setLeft(removeAux(successor, pKey));
        }
        return pRoot;
    }
}
```

# Árboles con nodos internos y externos diferentes

- Debe considerarse si los nodos hojas pueden ser **iguales** a los internos
- Usar la misma clase **simplifica** la implementación pero puede requerir más espacio
- Hay aplicaciones que requieren que sólo las hojas tengan un valor

$$4x(2x + a) - c$$



- Para poder hacer un árbol que tenga nodos de diferentes tipos hay que aprovechar la **herencia** de la orientación a objetos
- Se declara una **clase base** para el nodo con operaciones básicas (VarBinNode)
- Se construyen dos clases **derivadas** de esa clase base
  - LeafNode
  - IntNode

```
class VarBinNode {  
public:  
    virtual ~VarBinNode() {}  
    virtual bool isLeaf() = 0;  
};
```

```
class VarBinNode {  
public:  
    virtual ~VarBinNode() {}  
    virtual bool isLeaf() = 0;  
};
```

Clase abstracta para los nodos


```
class VarBinNode {  
public:  
    virtual ~VarBinNode() {}  
    virtual bool isLeaf() = 0;  
};
```



Destructor



```
class VarBinNode {  
public:  
    virtual ~VarBinNode() {}  
    virtual bool isLeaf() = 0;  
};
```




Método para determinar  
si el nodo es hoja o  
interno

```
class LeafNode : public VarBinNode {  
private:  
    Operand var;  
public:  
    LeafNode(const Operand& val) { var = val; }  
    bool isLeaf() { return true; }  
    Operand value() { return var; }  
};
```


```
class LeafNode : public VarBinNode {  
private:  
    Operand var;  
public:  
    LeafNode(const Operand& val) { var = val; }  
    bool isLeaf() { return true; }  
    Operand value() { return var; }  
};
```



Clase para el nodo hoja

```
class LeafNode : public VarBinNode {  
private:  
    Operand var;  Valor del operando  
public:  
    LeafNode(const Operand& val) { var = val; }  
    bool isLeaf() { return true; }  
    Operand value() { return var; }  
};
```

```
class LeafNode : public VarBinNode {  
private:  
    Operand var;  
public:  
    LeafNode(const Operand& val) { var = val; }  
    bool isLeaf() { return true; }  
    Operand value() { return var; }  
};
```



A diagram consisting of a purple rectangular box with the word "Constructor" in white text. A purple line originates from the left side of this box and points diagonally down and to the left, ending at the line of code `LeafNode(const Operand& val) { var = val; }` in the code block above.

```
class LeafNode : public VarBinNode {  
private:  
    Operand var;  
public:  
    LeafNode(const Operand& val) { var = val; }  
    bool isLeaf() { return true; }  
    Operand value() { return var; }  
};
```



Dice que es hoja

```
class LeafNode : public VarBinNode {  
private:  
    Operand var;  
public:  
    LeafNode(const Operand& val) { var = val; }  
    bool isLeaf() { return true; }  
    Operand value() { return var; }  
};
```



Retorna el valor del operando


```
class IntlNode : public VarBinNode {
private:
    VarBinNode* left;
    VarBinNode* right;
    Operator opx;
public:
    IntlNode(Operator op, VarBinNode* l, VarBinNode* r)
        { opx = op; left = l; right = r; }
    bool isLeaf() { return false; }
    VarBinNode* leftchild() { return left; }
    VarBinNode* rightchild() { return right; }
    Operator value() { return opx; }
};
```



```
class IntlNode : public VarBinNode {
private:
    VarBinNode* left;
    VarBinNode* right;
    Operator opx;
public:
    IntlNode(Operator op, VarBinNode* l, VarBinNode* r)
        { opx = op; left = l; right = r; }
    bool isLeaf() { return false; }
    VarBinNode* leftchild() { return left; }
    VarBinNode* rightchild() { return right; }
    Operator value() { return opx; }
};
```

Clase para nodos internos

```
class IntlNode : public VarBinNode {
private:
    VarBinNode* left;
    VarBinNode* right;
    Operator opx;
public:
    IntlNode(Operator op, VarBinNode* l, VarBinNode* r)
        { opx = op; left = l; right = r; }
    bool isLeaf() { return false; }
    VarBinNode* leftchild() { return left; }
    VarBinNode* rightchild() { return right; }
    Operator value() { return opx; }
};
```



Hijos del nodo

```
class IntlNode : public VarBinNode {
private:
    VarBinNode* left;
    VarBinNode* right;
    Operator opx;
public:
    IntlNode(Operator op, VarBinNode* l, VarBinNode* r)
        { opx = op; left = l; right = r; }
    bool isLeaf() { return false; }
    VarBinNode* leftchild() { return left; }
    VarBinNode* rightchild() { return right; }
    Operator value() { return opx; }
};
```

Operador de la expresión

```
class IntlNode : public VarBinNode {
private:
    VarBinNode* left;
    VarBinNode* right;
    Operator opx;
public:
    IntlNode(Operator op, VarBinNode* l, VarBinNode* r)
        { opx = op; left = l; right = r; }
    bool isLeaf() { return false; }
    VarBinNode* leftchild() { return left; }
    VarBinNode* rightchild() { return right; }
    Operator value() { return opx; }
};
```



Constructor

```
class IntlNode : public VarBinNode {
private:
    VarBinNode* left;
    VarBinNode* right;
    Operator opx;
public:
    IntlNode(Operator op, VarBinNode* l, VarBinNode* r)
        { opx = op; left = l; right = r; }
    bool isLeaf() { return false; }
    VarBinNode* leftchild() { return left; }
    VarBinNode* rightchild() { return right; }
    Operator value() { return opx; }
};
```

No es nodo hoja

```
class IntlNode : public VarBinNode {
private:
    VarBinNode* left;
    VarBinNode* right;
    Operator opx;
public:
    IntlNode(Operator op, VarBinNode* l, VarBinNode* r)
        { opx = op; left = l; right = r; }
    bool isLeaf() { return false; }
    VarBinNode* leftchild() { return left; }
    VarBinNode* rightchild() { return right; }
    Operator value() { return opx; }
};
```

**Métodos para  
acceder a los  
hijos del nodo**

```
class IntlNode : public VarBinNode {  
private:  
    VarBinNode* left;  
    VarBinNode* right;  
    Operator opx;  
public:  
    IntlNode(Operator op, VarBinNode* l, VarBinNode* r)  
        { opx = op; left = l; right = r; }  
    bool isLeaf() { return false; }  
    VarBinNode* leftchild() { return left; }  
    VarBinNode* rightchild() { return right; }  
    Operator value() { return opx; }  
};
```



Acceso al operador

```
void traverse(VarBinNode *root) {  
    if (root == NULL) return;  
    if (root->isLeaf())  
        cout << "Leaf: "  
            << ((LeafNode *)root)->value() << endl;  
    else {  
        cout << "Internal: "  
            << ((IntlNode *)root)->value() << endl;  
        traverse(((IntlNode *)root)->leftchild());  
        traverse(((IntlNode *)root)->rightchild());  
    }  
}
```



# Heaps y colas de prioridad

- Escoger el siguiente “más importante” de una colección de personas, tareas u objetos
- Los sistemas operativos priorizan los diferentes trabajos a ejecutar
- Los **heaps** son la estructura de datos que permite este tipo de organización
- Propiedades:
  - Es un árbol binario **completo**
  - Los valores están **parcialmente ordenados**

- Existe una relación entre el valor de un nodo y el de sus hijos, no entre nodos “hermanos”
- Tipos:
  - Max-heap
    - El valor de un nodo es **mayor** que sus hijos. La raíz es el nodo de mayor valor
  - Min-heap
    - El valor de un nodo es **menor** que sus hijos. La raíz es el nodo de menor valor
- Similar a una cola, cuando se elimina un elemento, sólo puede eliminarse el que está en la **raíz**

# Inserciones en un heap

- Se insertan los nuevos valores en la **siguiente posición libre** del mayor nivel, de izquierda a derecha
- Si ya no hay espacio en el nivel, se crea uno nuevo
- Si la prioridad del nuevo elemento es mayor que la de su padre, se **intercambian**
- Se repite la comparación con el siguiente padre hasta que ya no haya intercambios

Insertar: 95, 29, 12, 64, 26, 54, 15, 7, 22, 12, 84, 22, 5, 44, 98

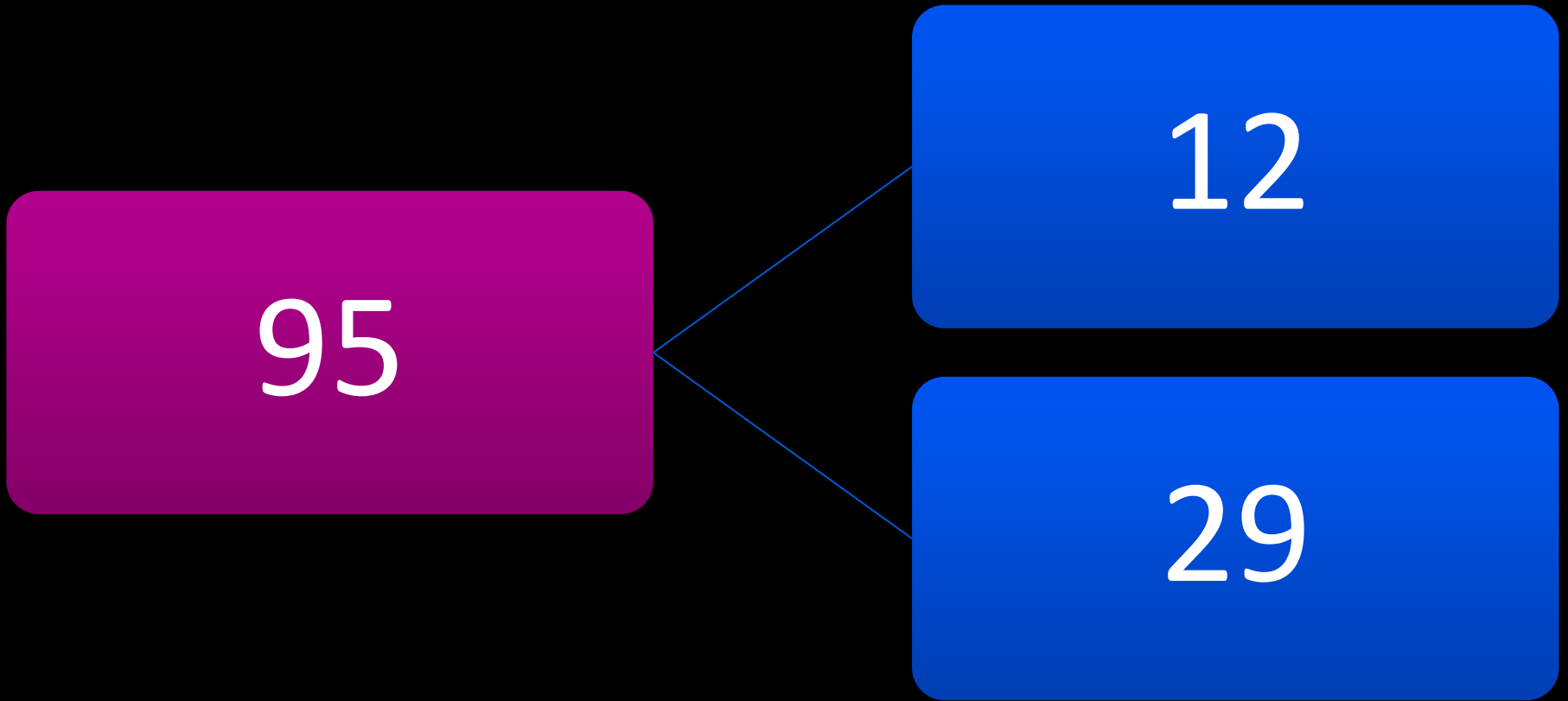
95

Insertar: 95, 29, 12, 64, 26, 54, 15, 7, 22, 12, 84, 22, 5, 44, 98

95

29

Insertar: 29, 12, 64, 26, 54, 15, 7, 22, 12, 84, 22, 5, 44, 98

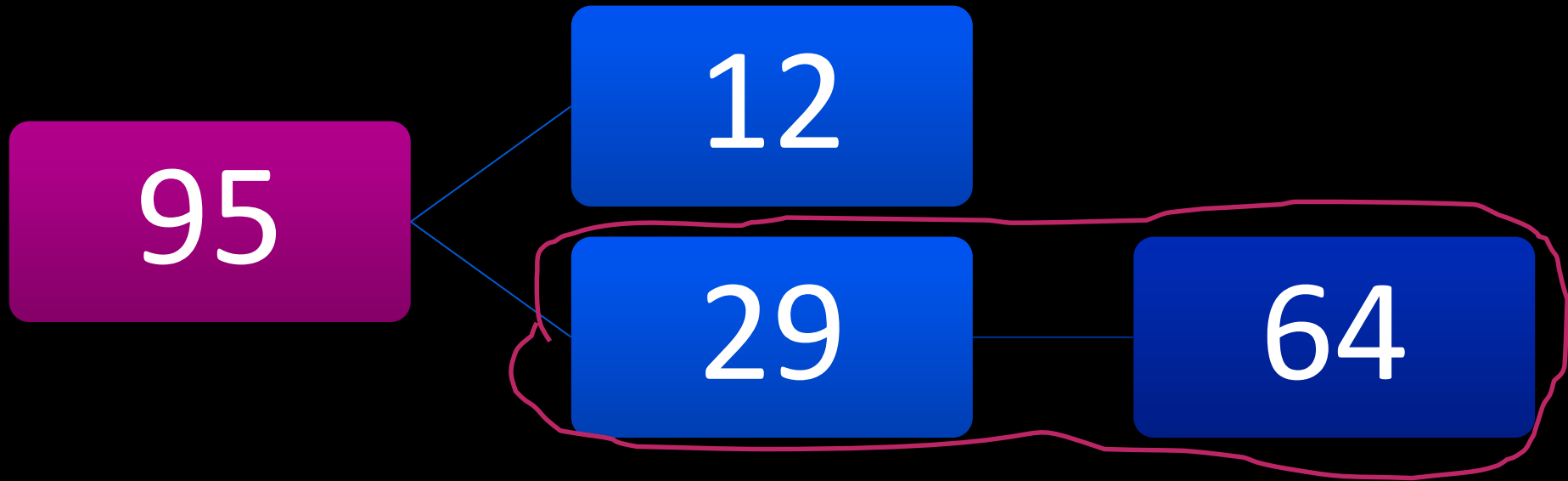


Insertar: 12, 64, 26, 54, 15, 7, 22, 12, 84, 22, 5, 44, 98



Insertar: 64, 26, 54, 15, 7, 22, 12, 84, 22, 5, 44, 98

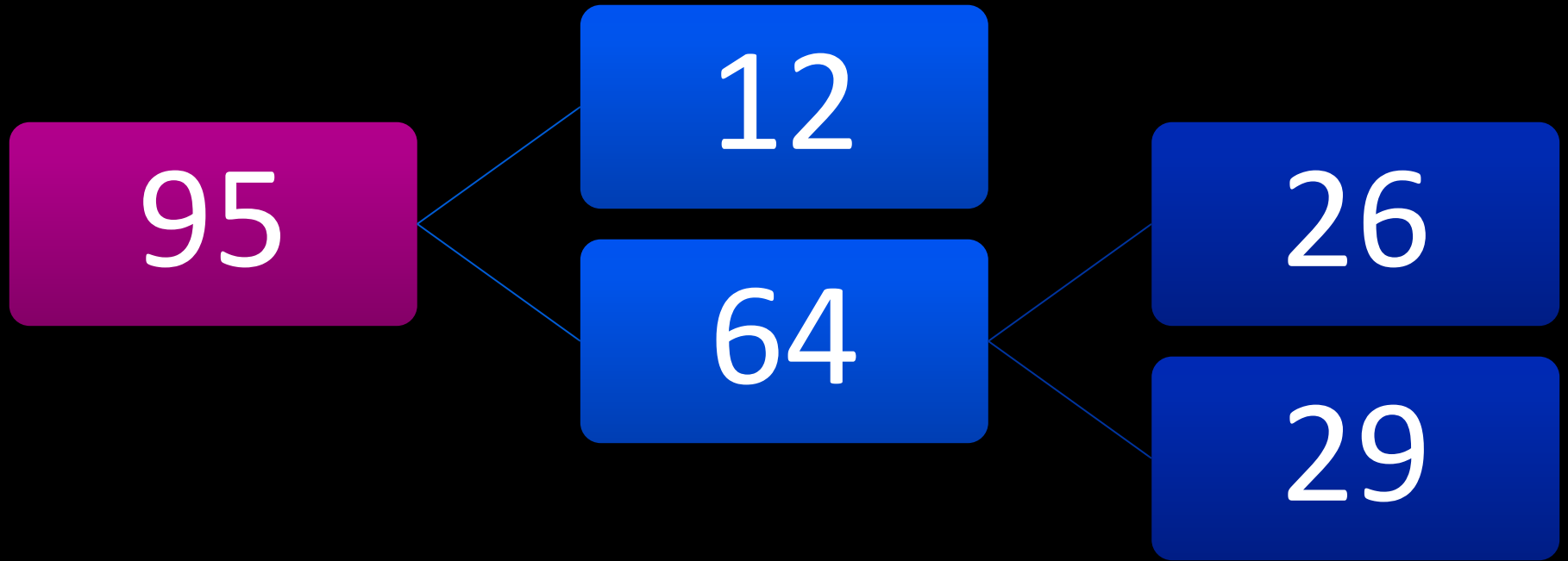




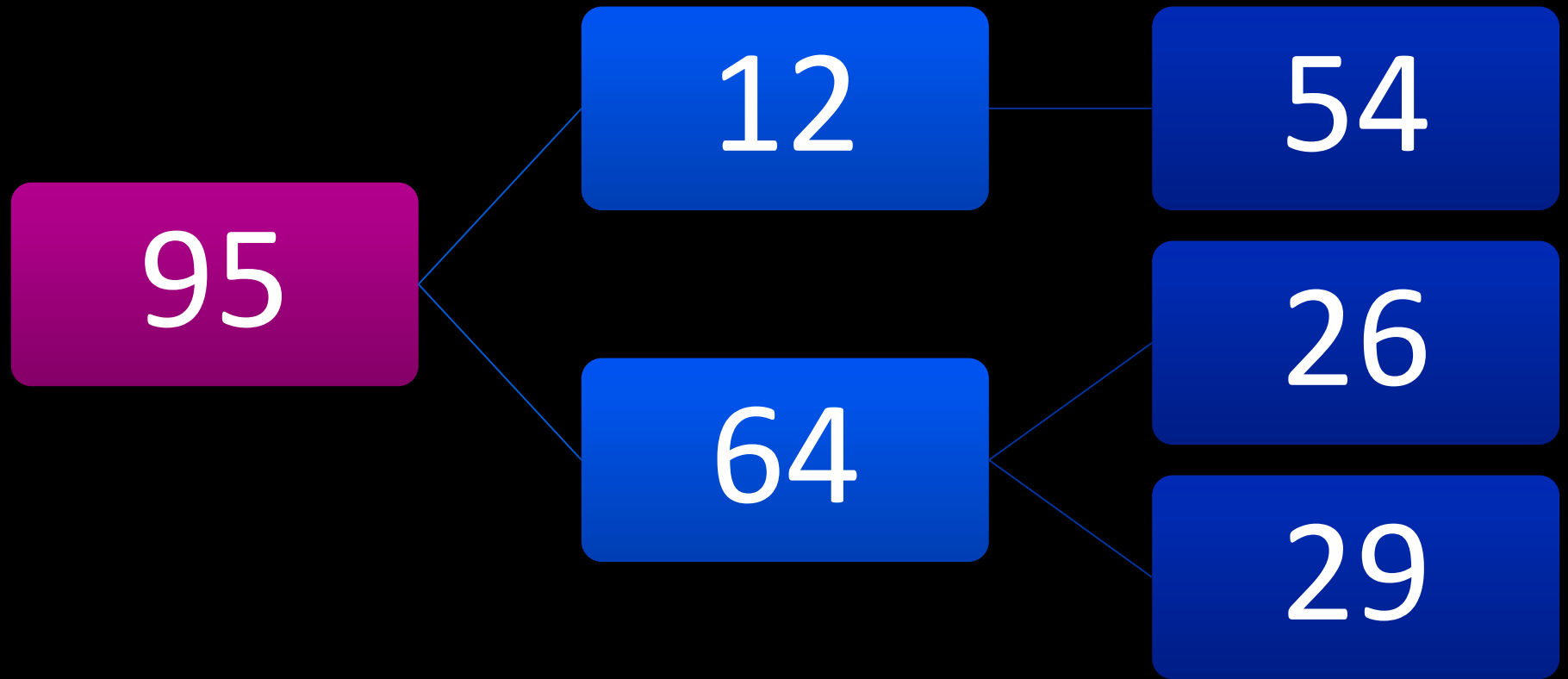
Insertar: 64, 26, 54, 15, 7, 22, 12, 84, 22, 5, 44, 98



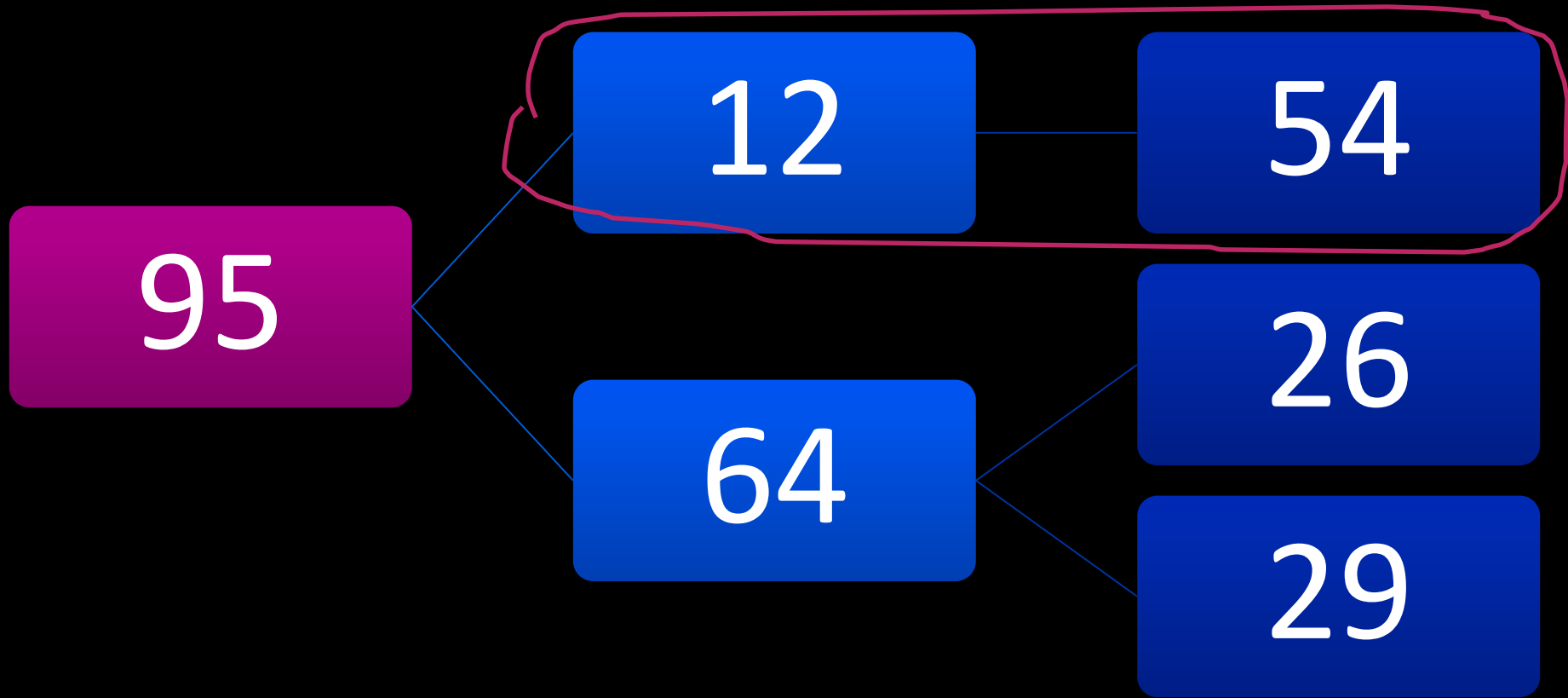
Insertar: 64, 26, 54, 15, 7, 22, 12, 84, 22, 5, 44, 98



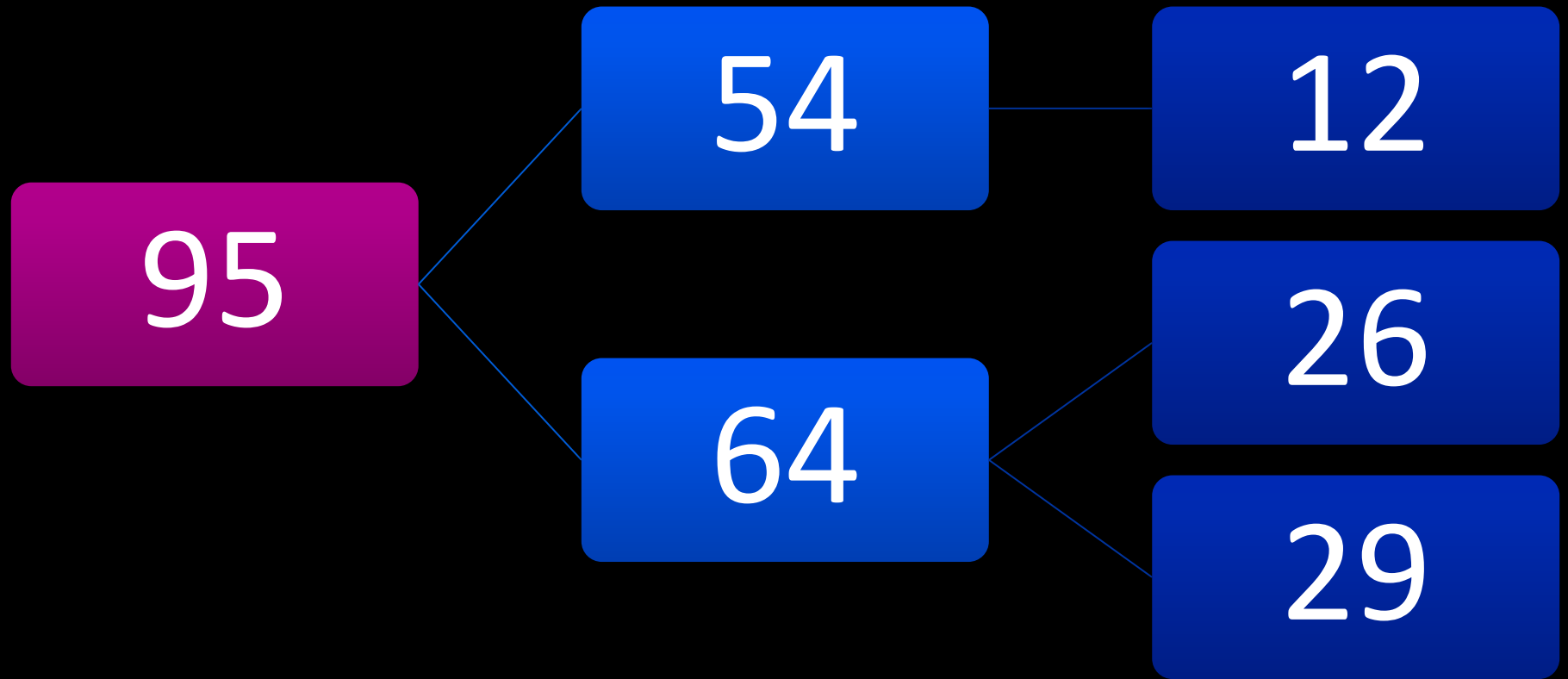
Insertar: 26, 54, 15, 7, 22, 12, 84, 22, 5, 44, 98



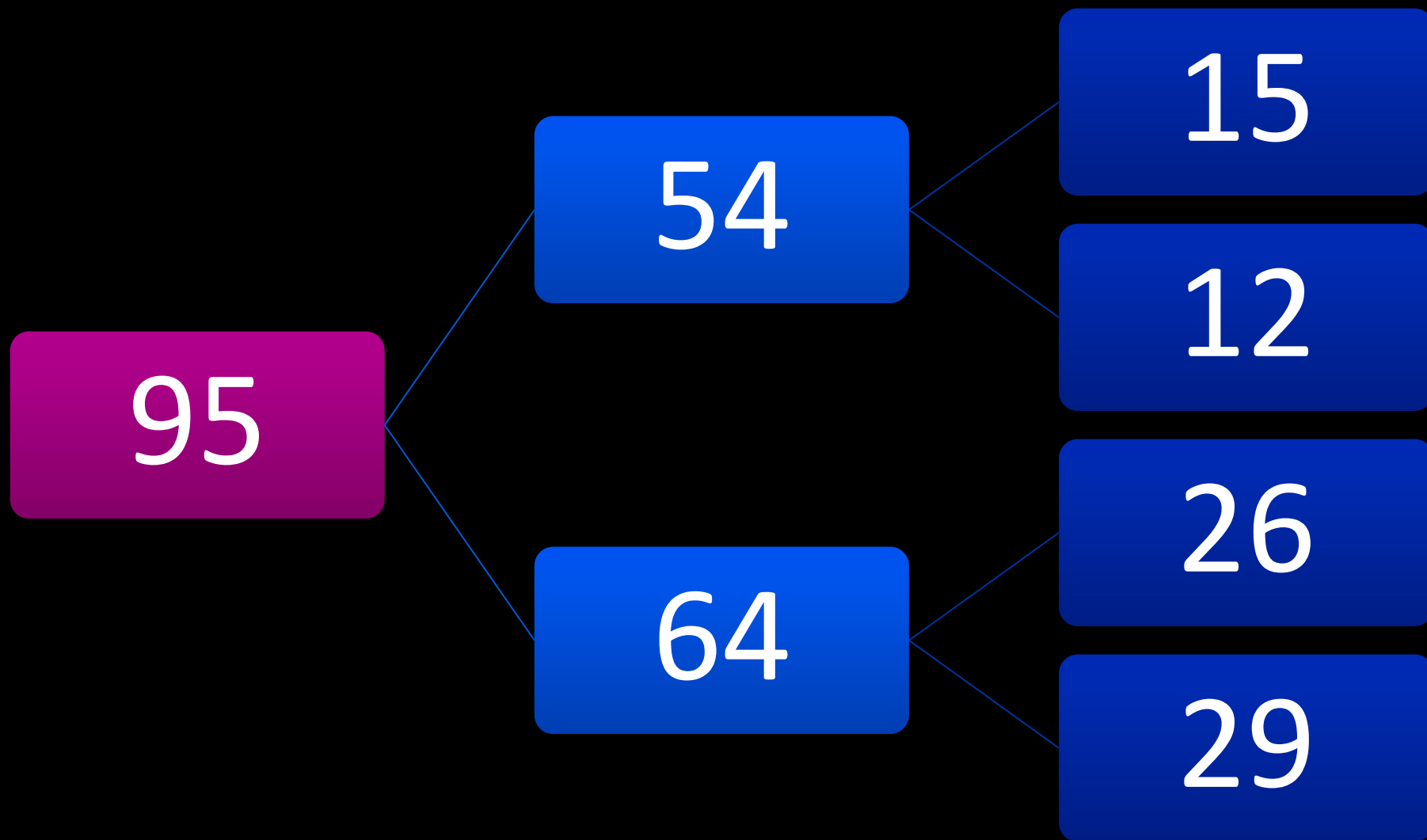
Insertar: 54, 15, 7, 22, 12, 84, 22, 5, 44, 98



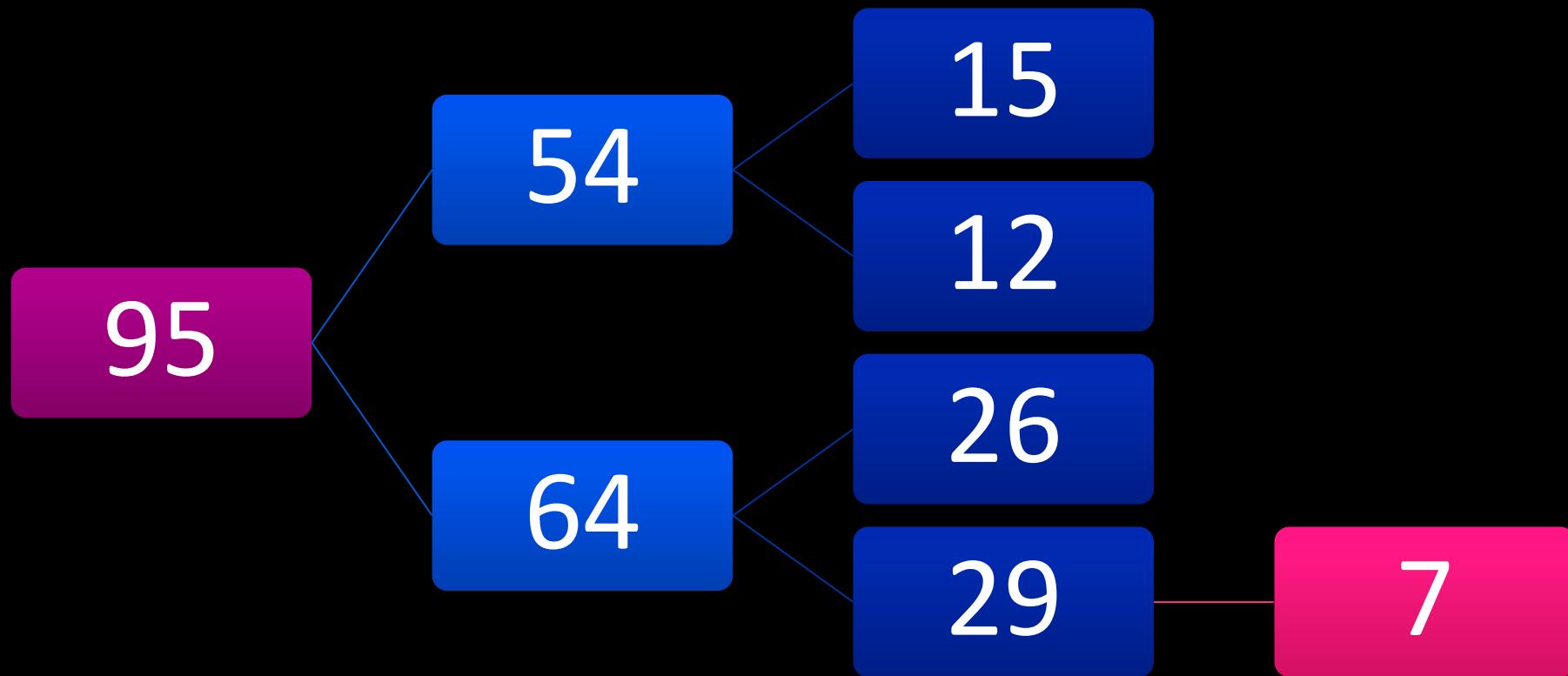
Insertar: 54, 15, 7, 22, 12, 84, 22, 5, 44, 98



Insertar: 54, 15, 7, 22, 12, 84, 22, 5, 44, 98

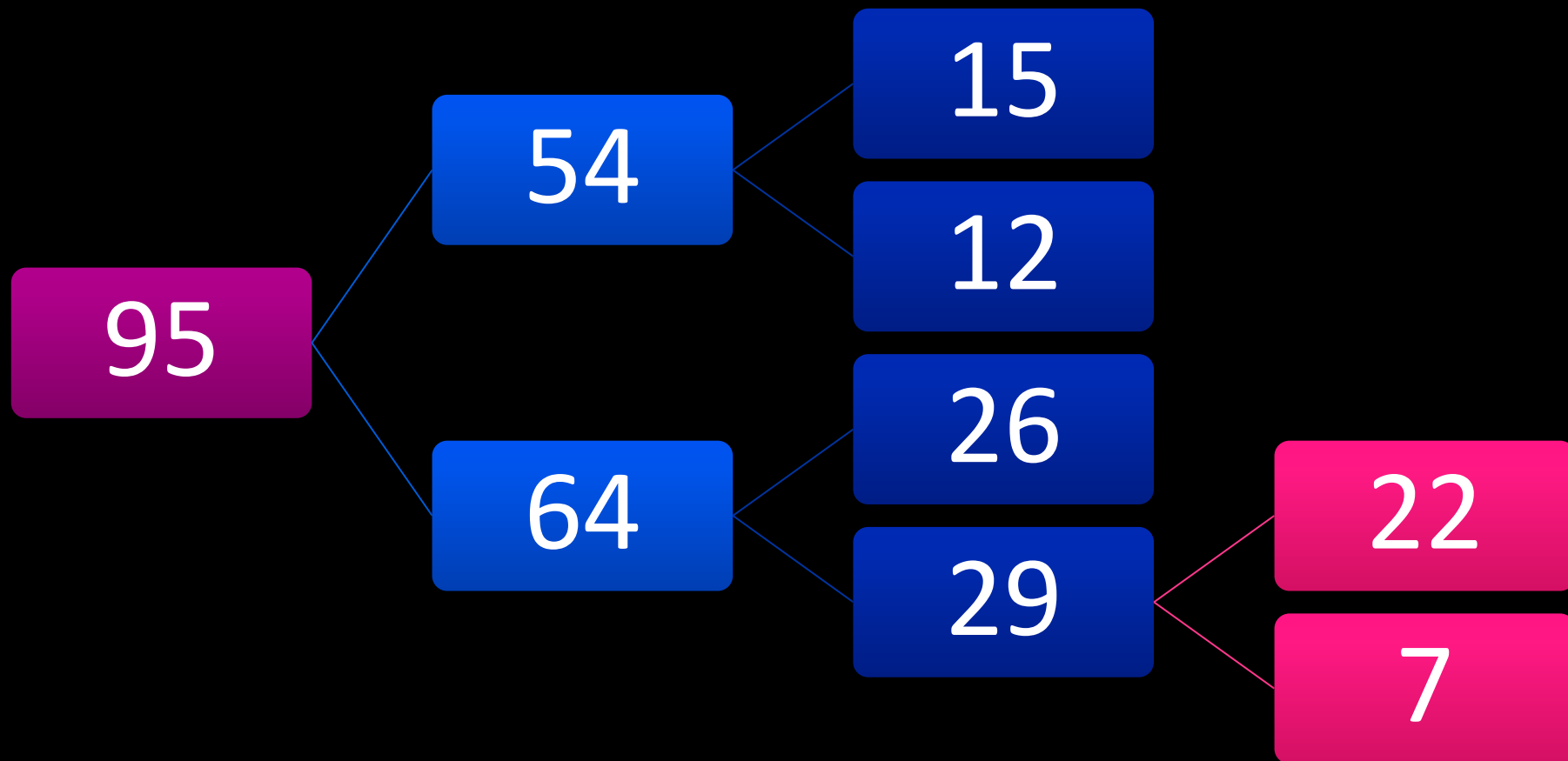


Insertar: 15, 7, 22, 12, 84, 22, 5, 44, 98

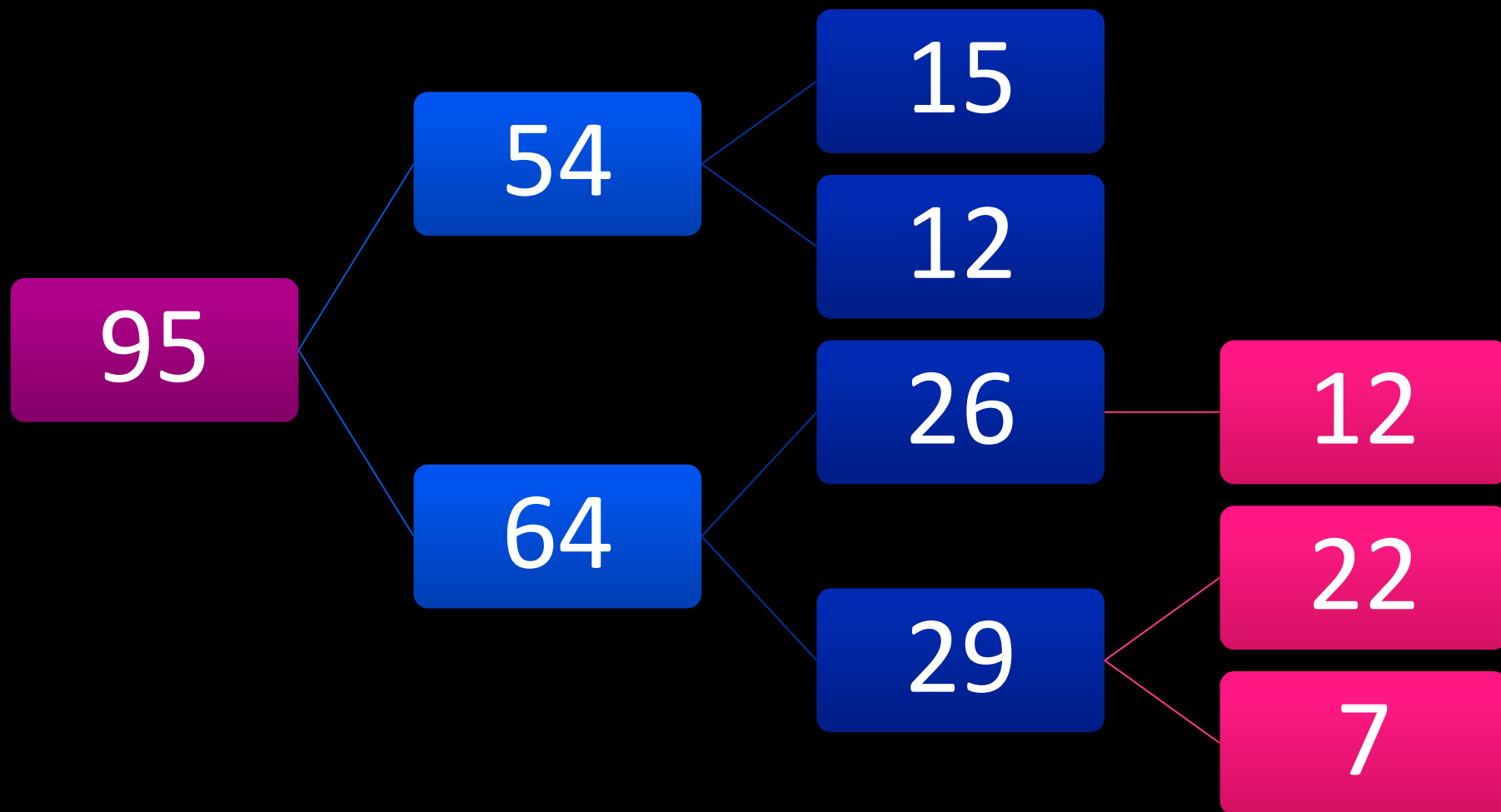


Insertar: 7, 22, 12, 84, 22, 5, 44, 98

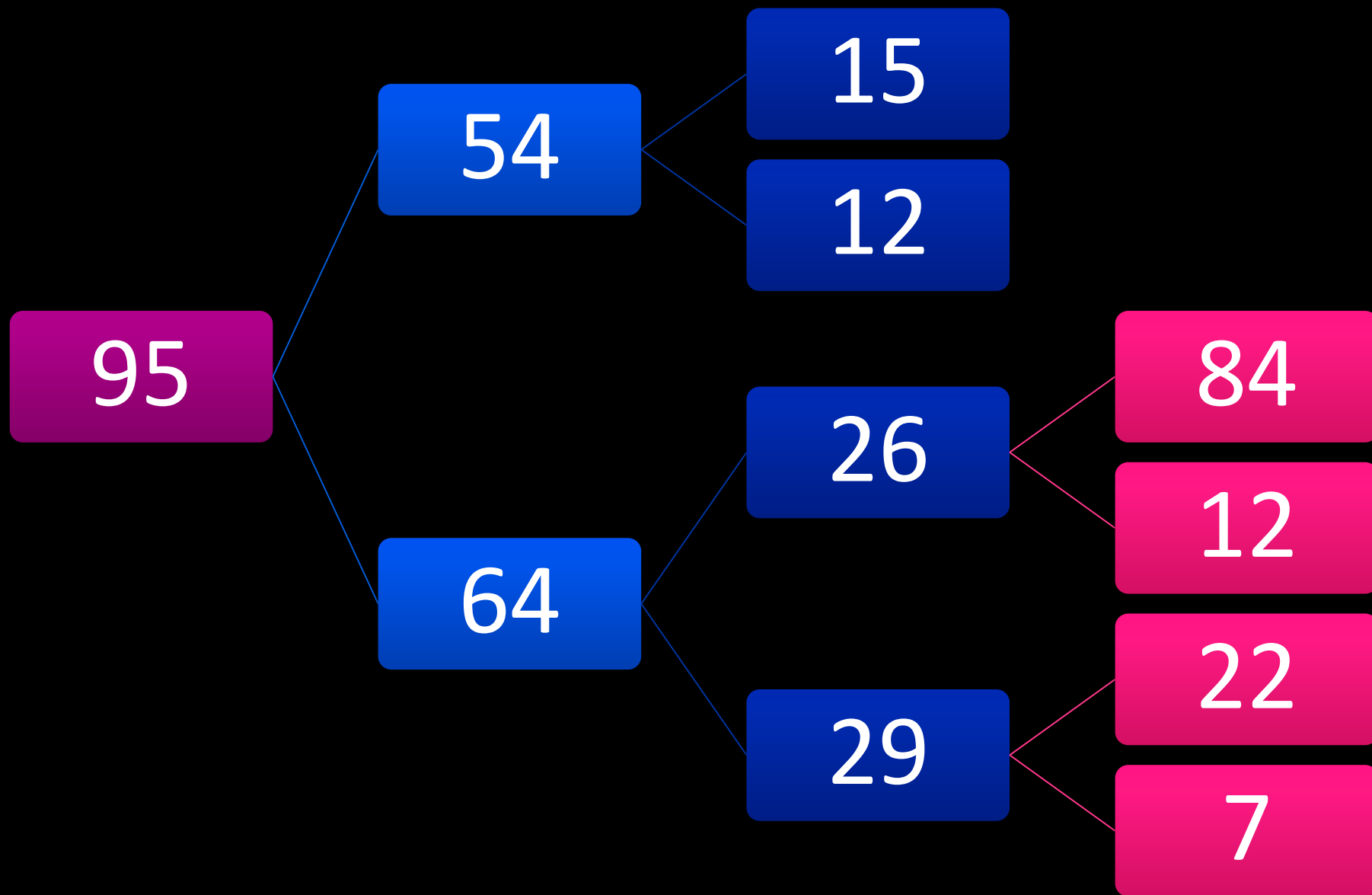




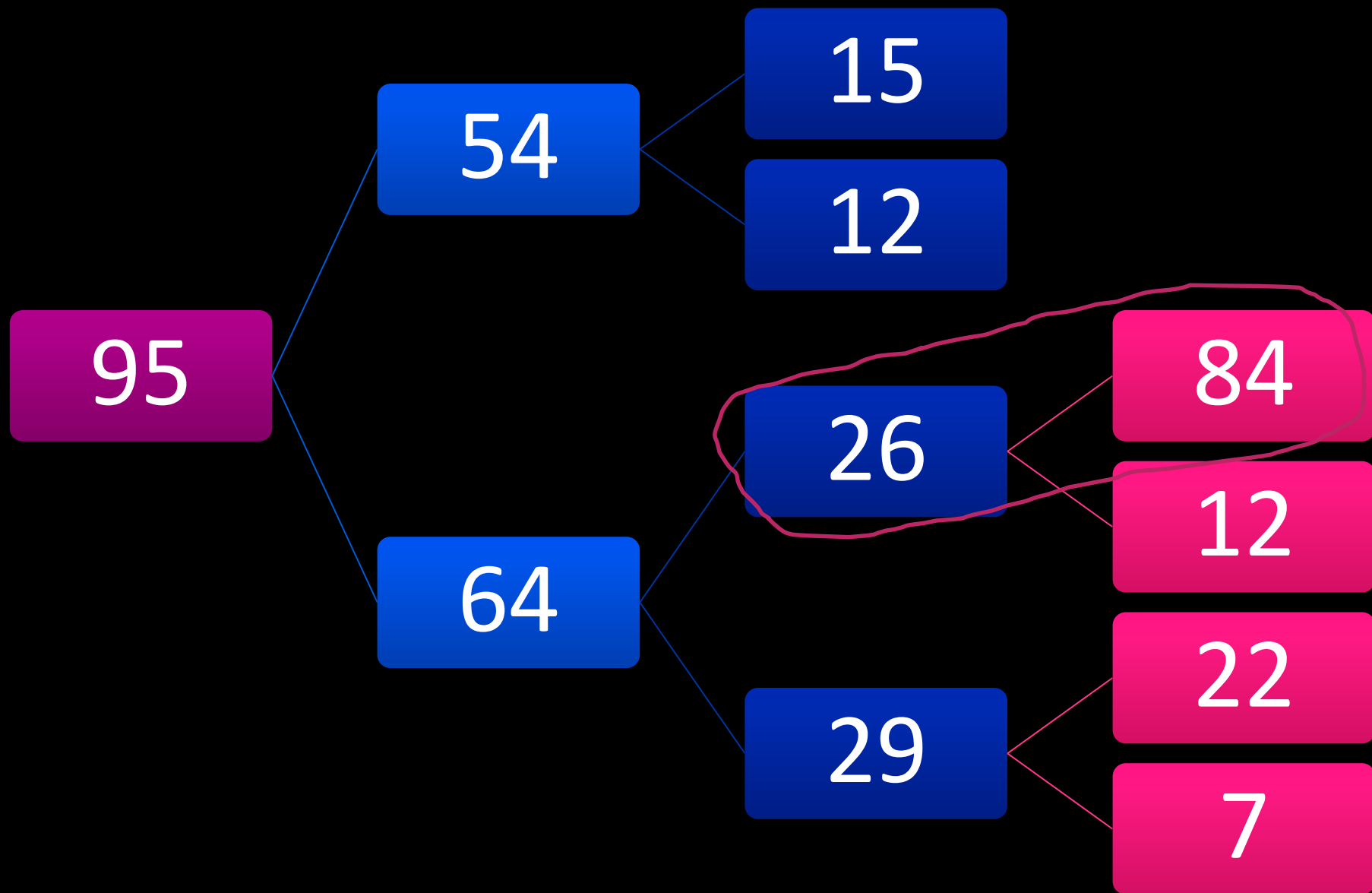
Insertar: 22, 12, 84, 22, 5, 44, 98



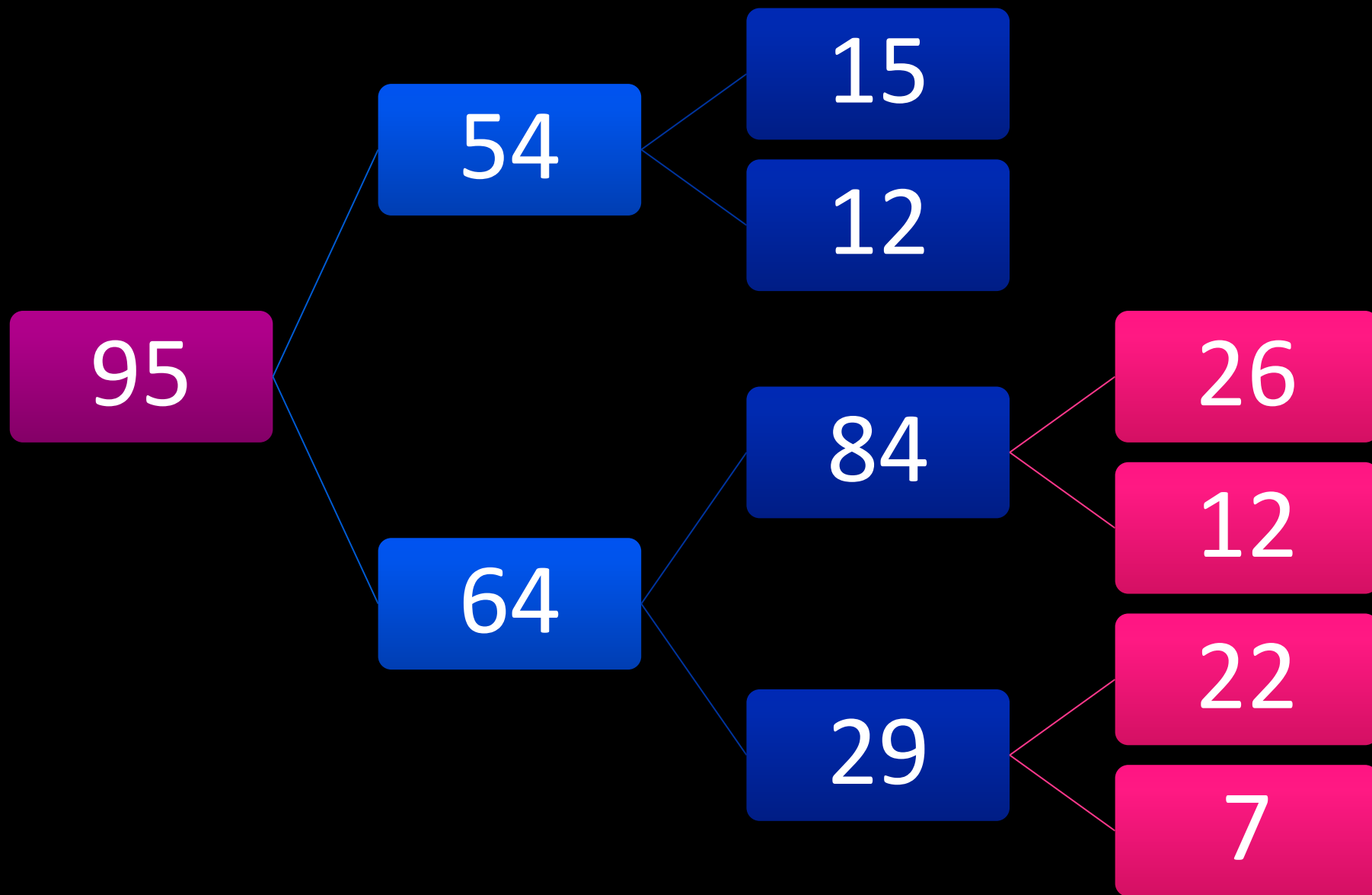
Insertar: 12, 84, 22, 5, 44, 98



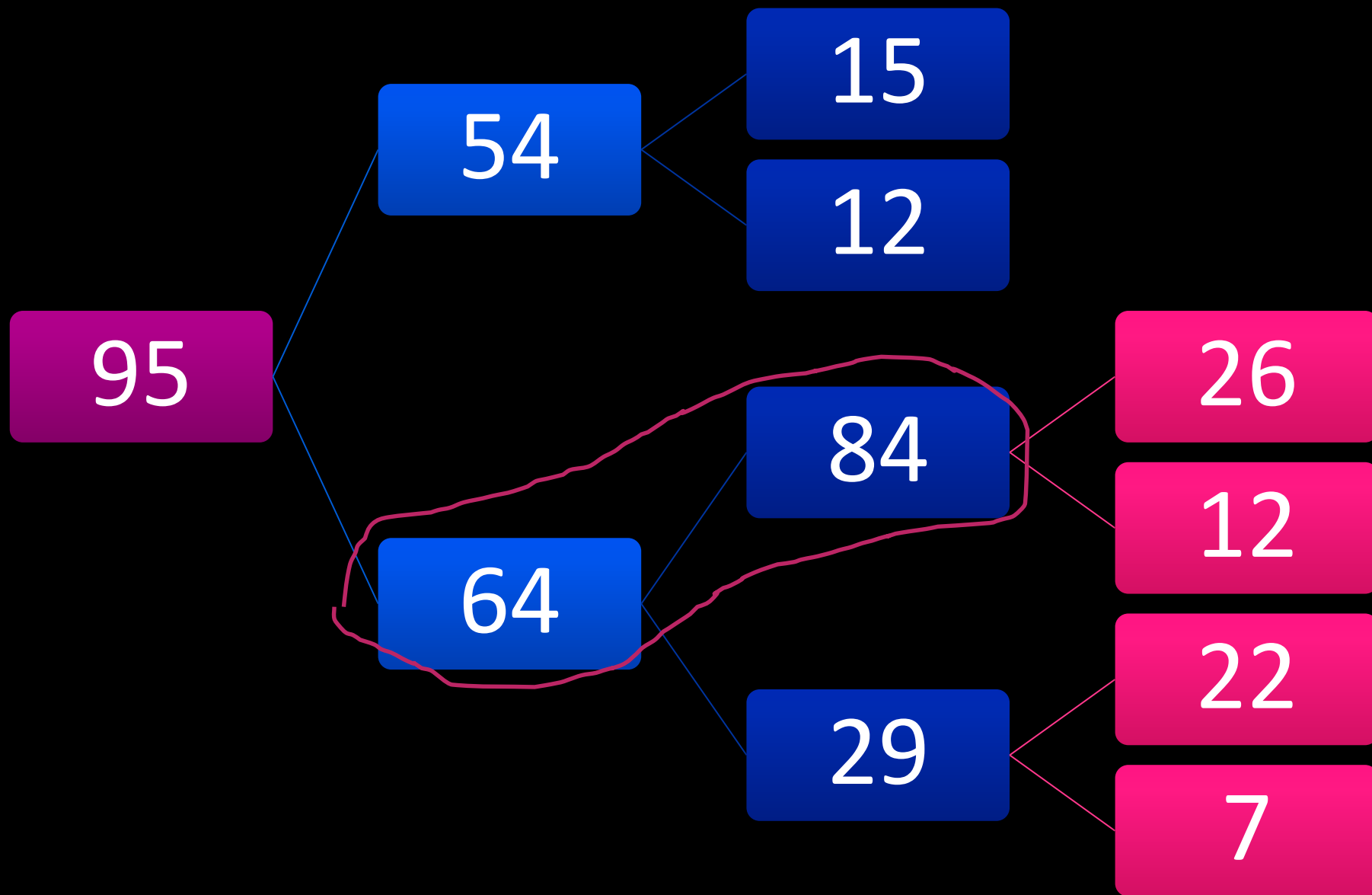
Insertar: 84, 22, 5, 44, 98



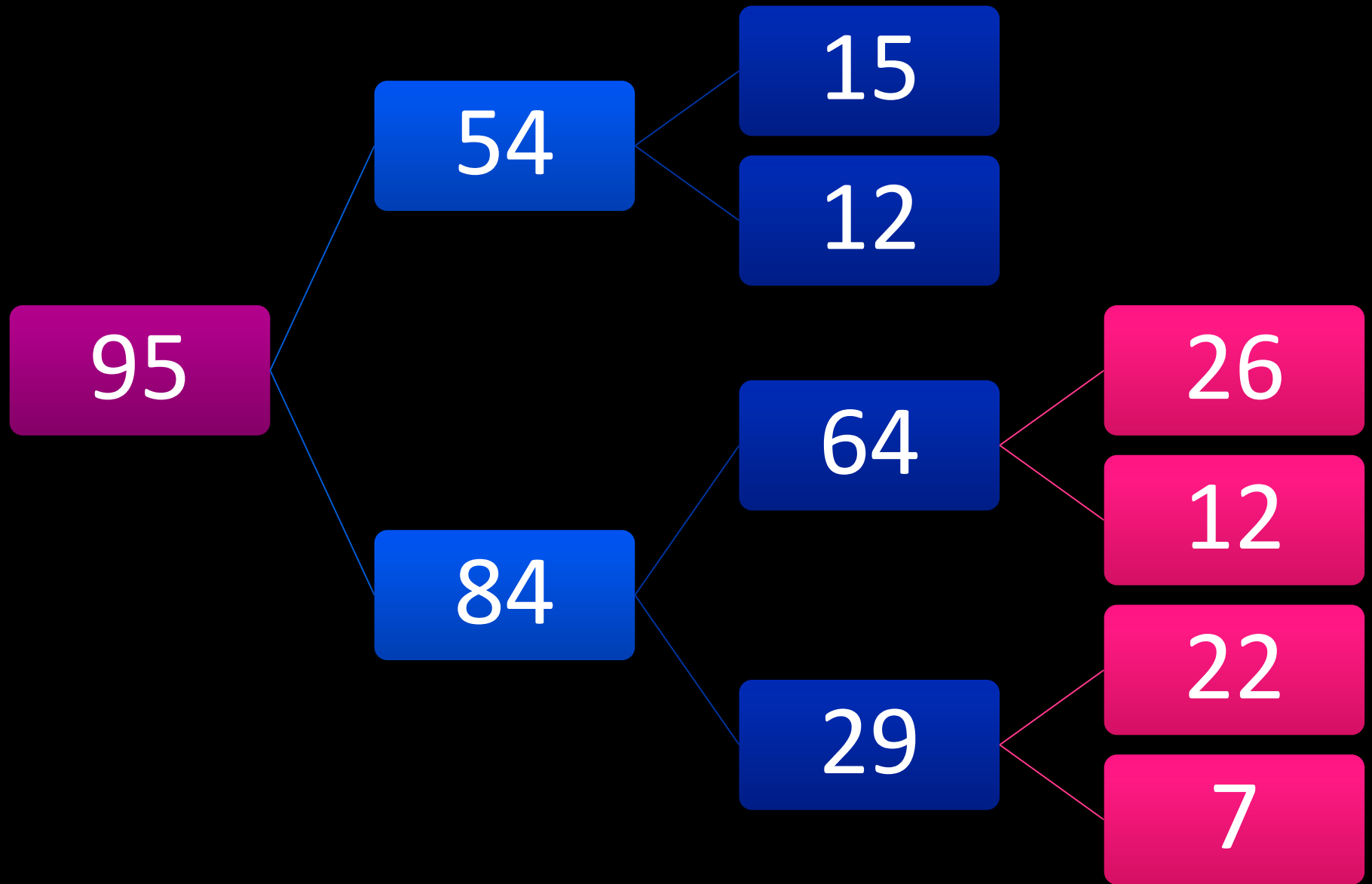
Insertar: 84, 22, 5, 44, 98



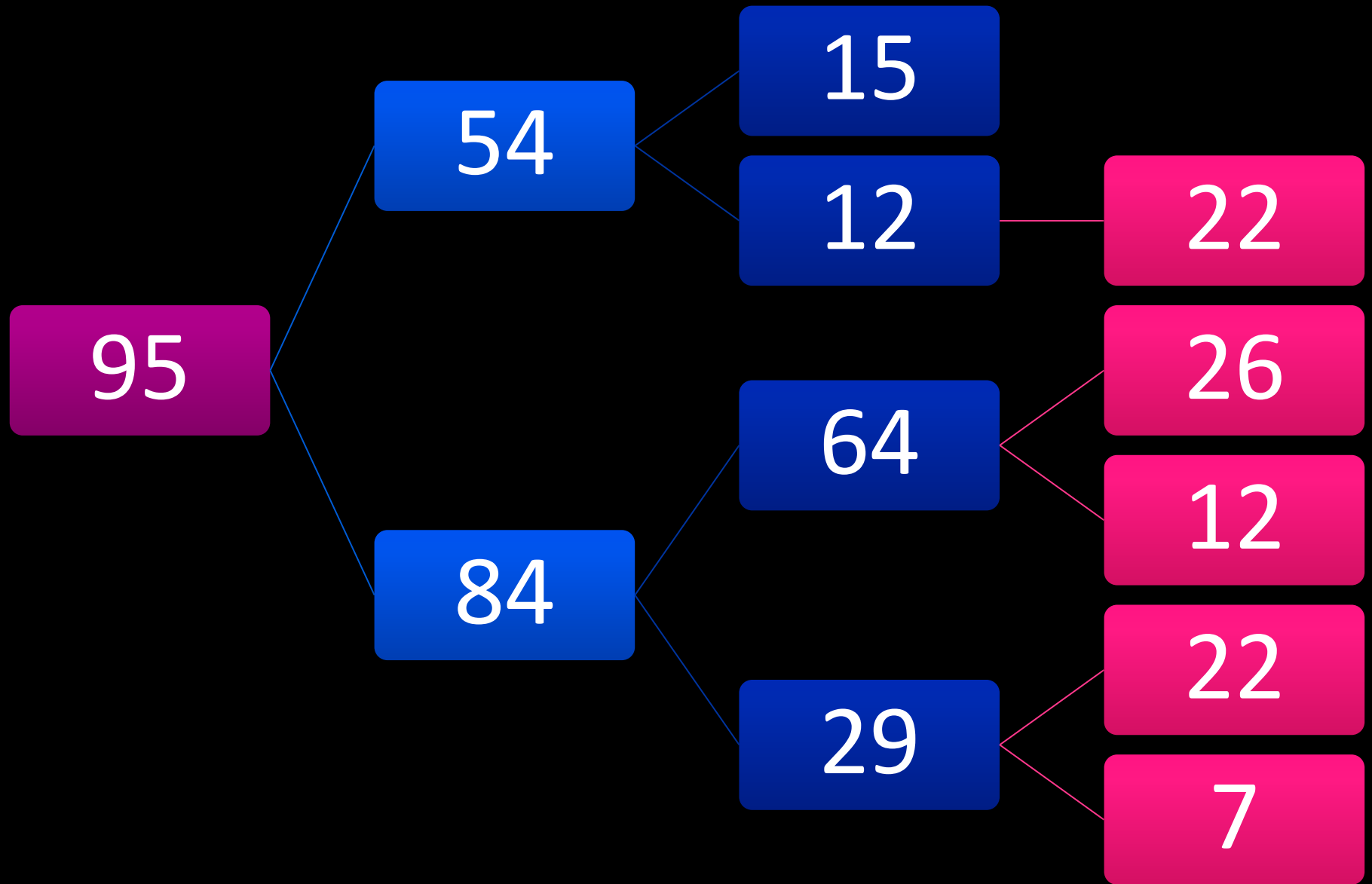
Insertar: 84, 22, 5, 44, 98



Insertar: 84, 22, 5, 44, 98

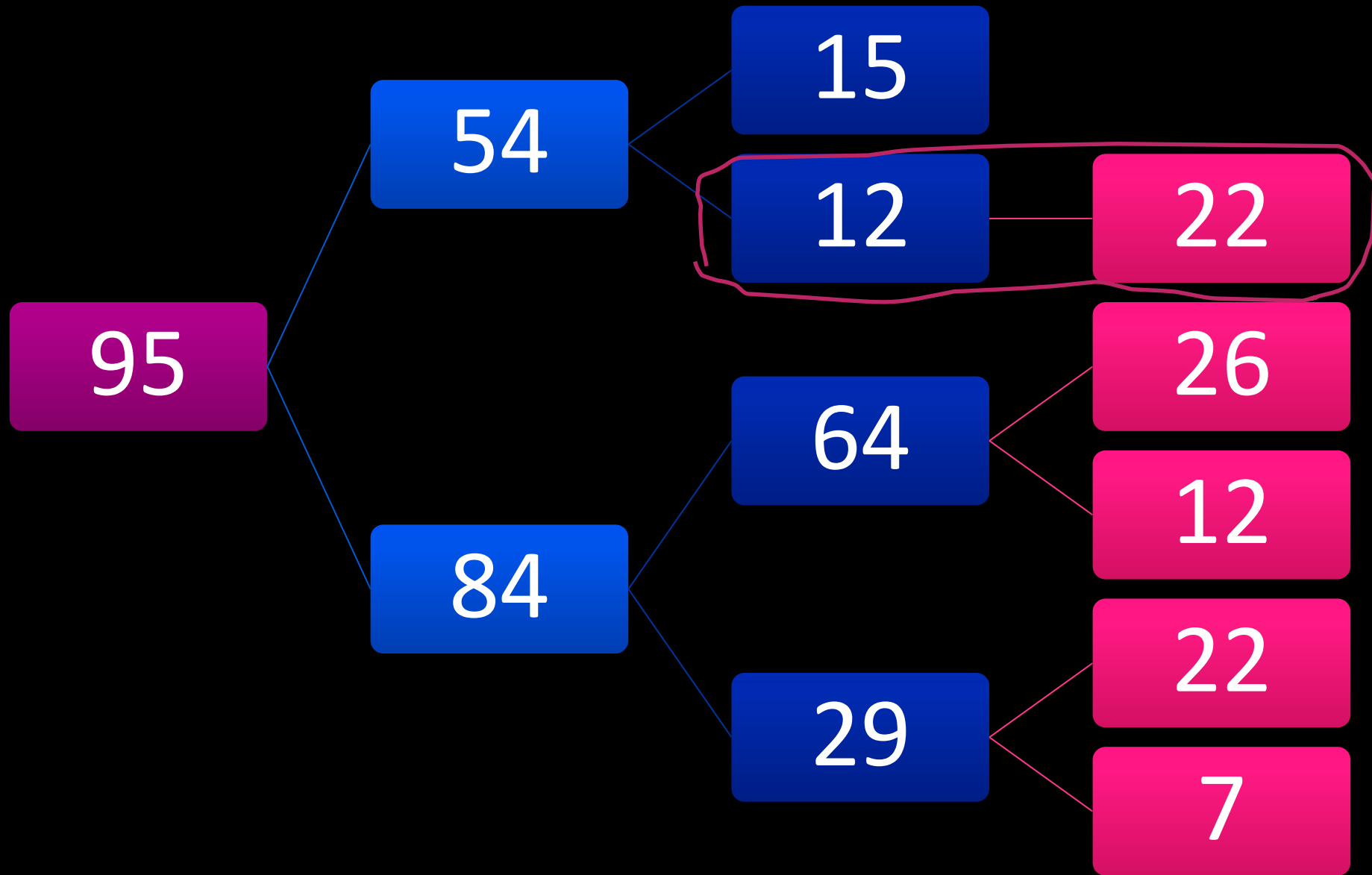


Insertar: 84, 22, 5, 44, 98

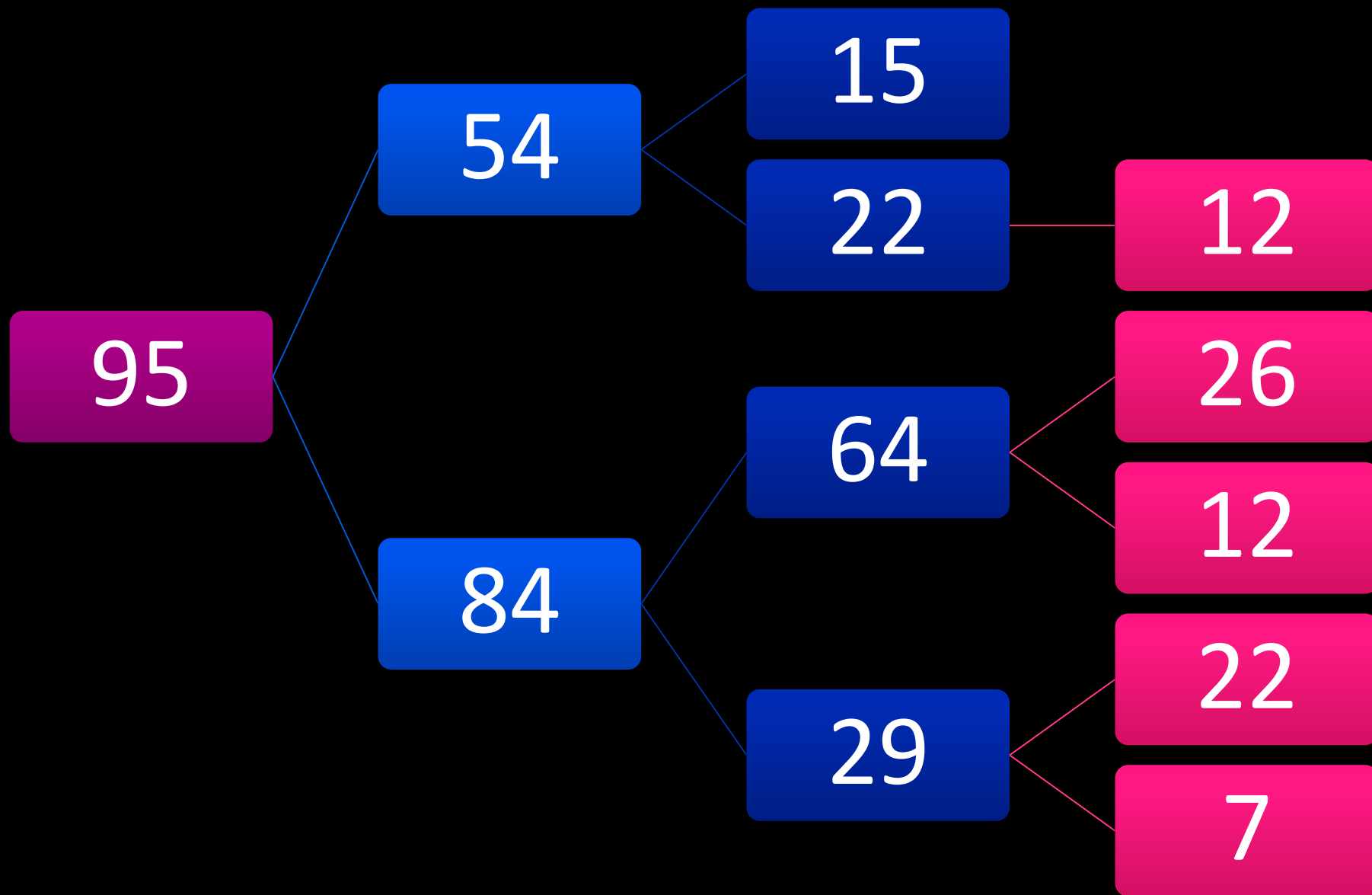


Insertar: 22, 5, 44, 98

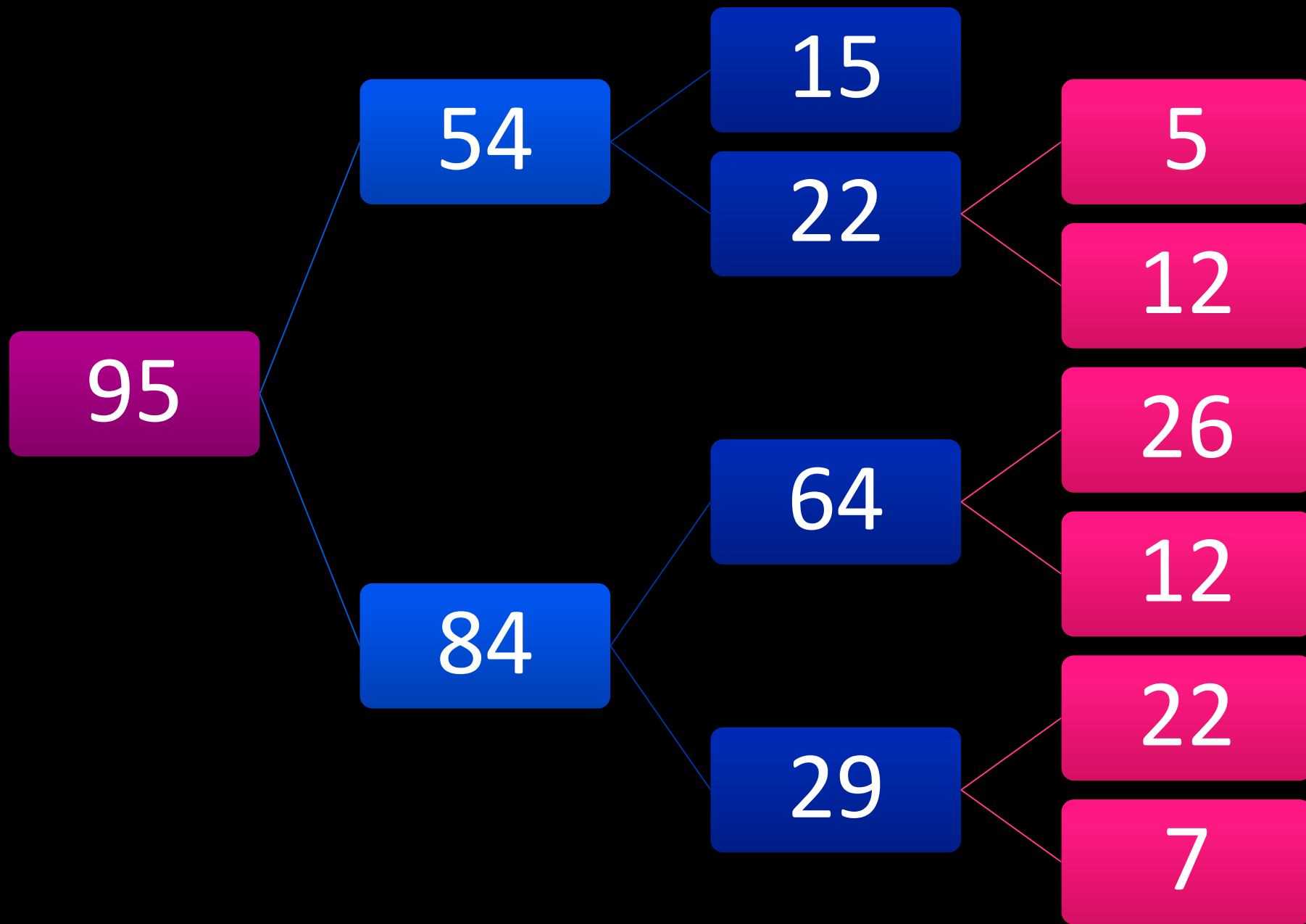




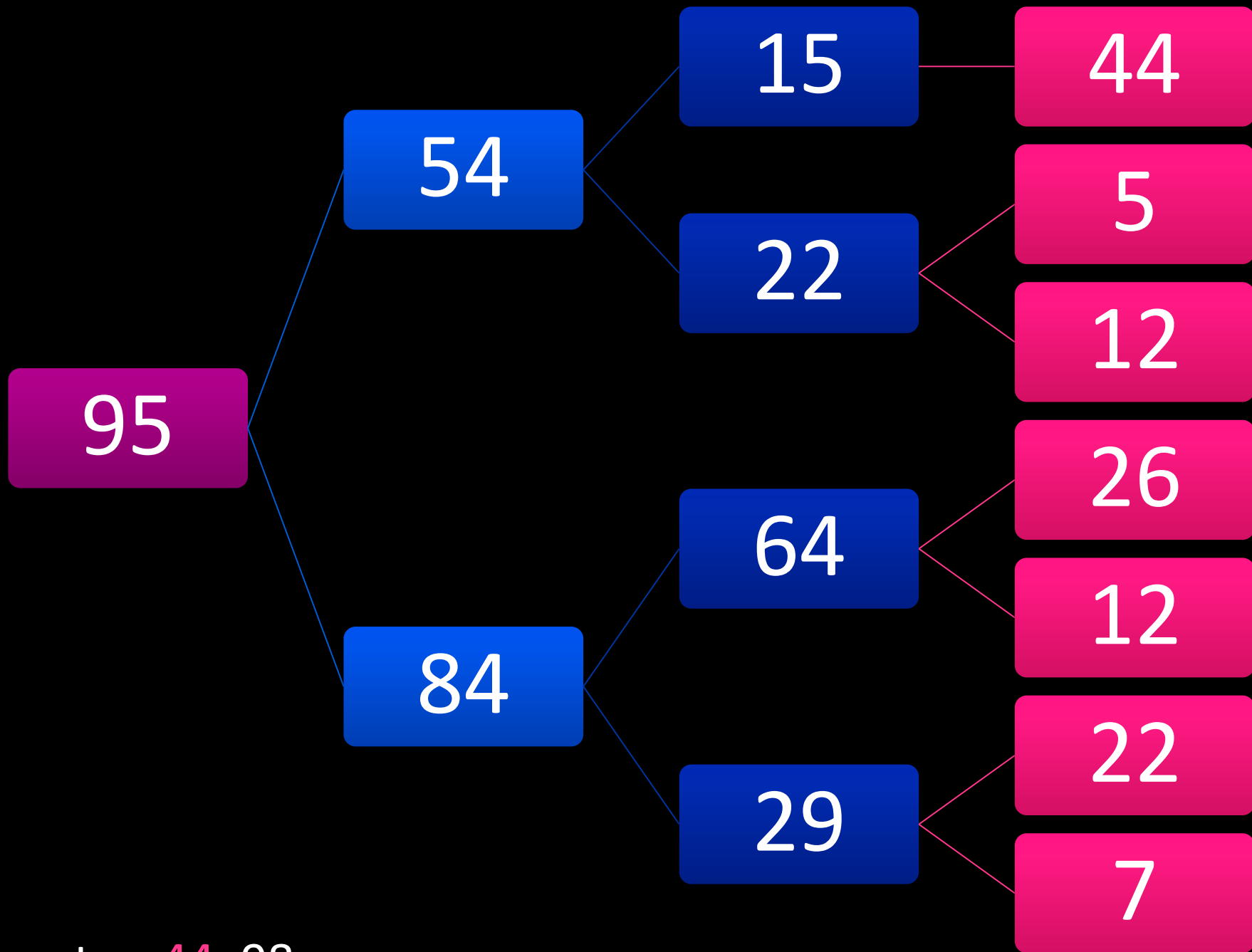
Insertar: 22, 5, 44, 98



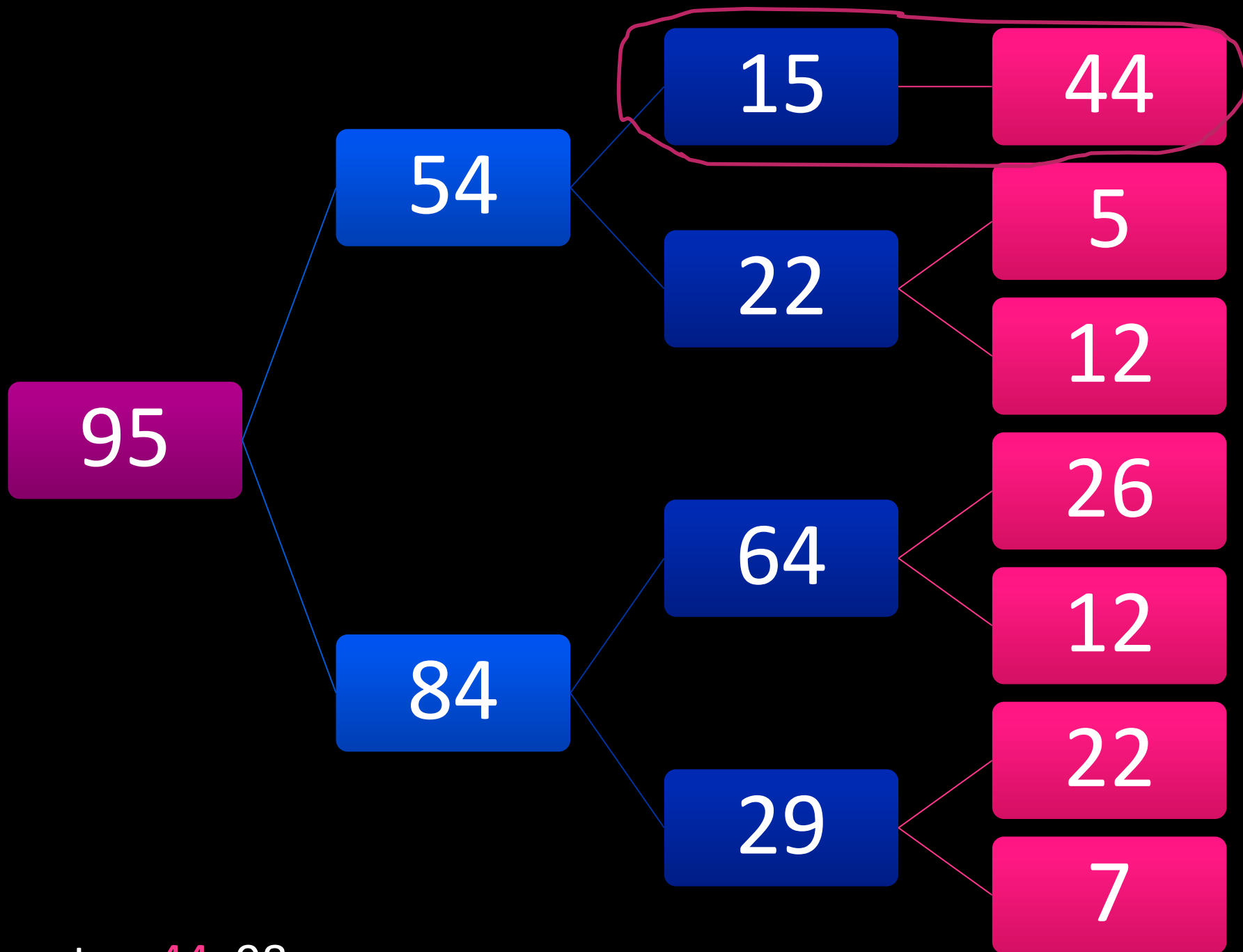
Insertar: 22, 5, 44, 98



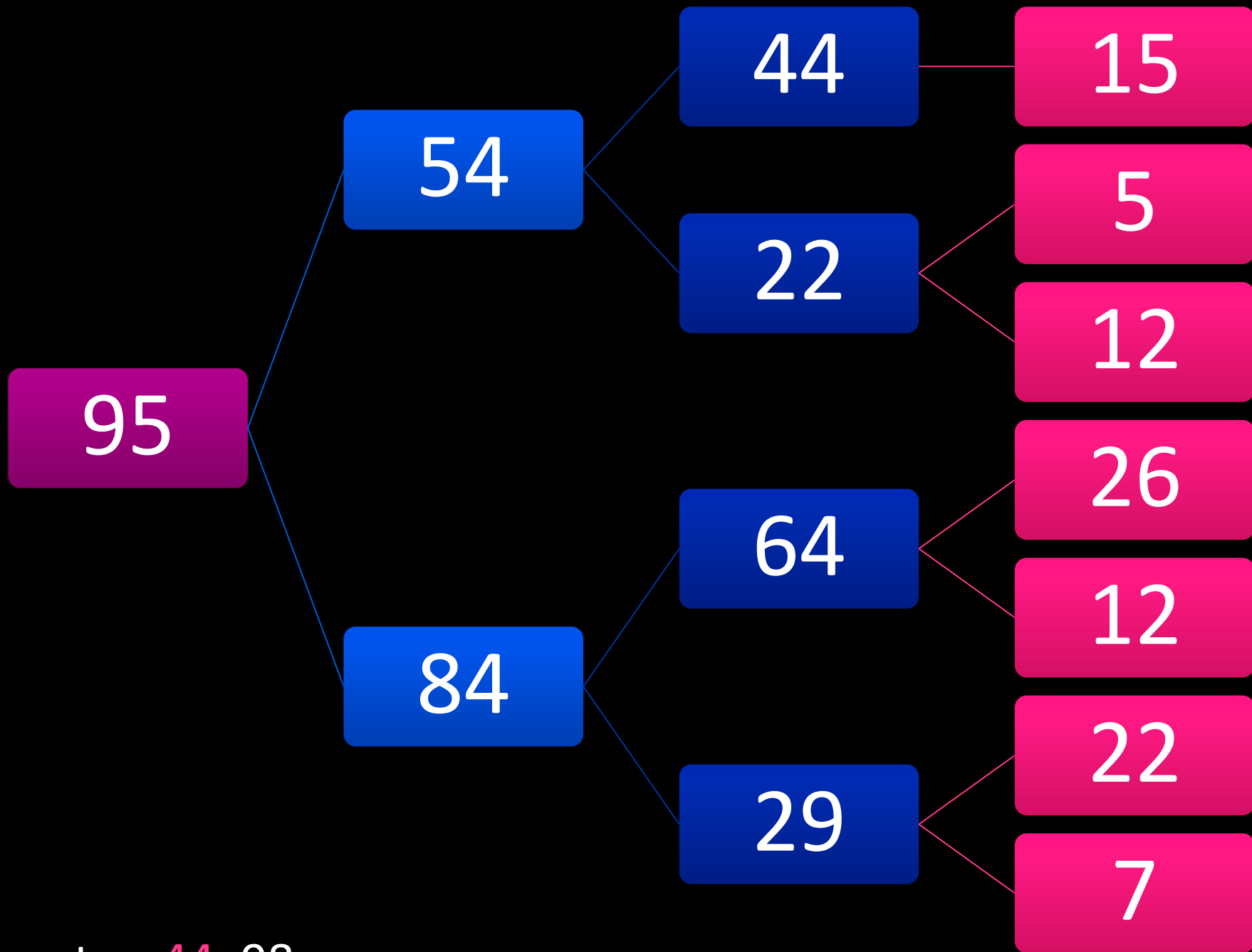
Insertar: 5, 44, 98



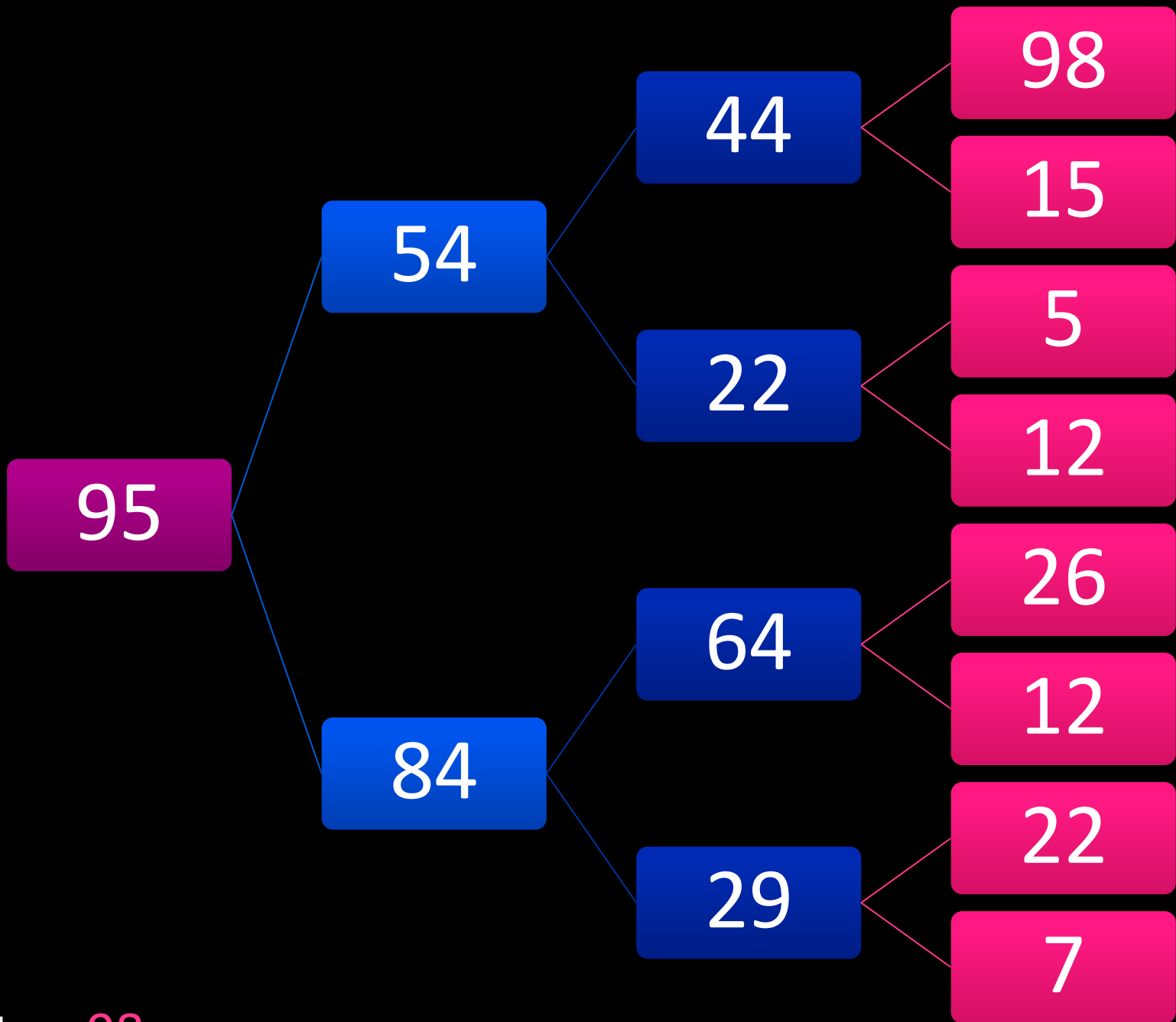
Insertar: 44, 98



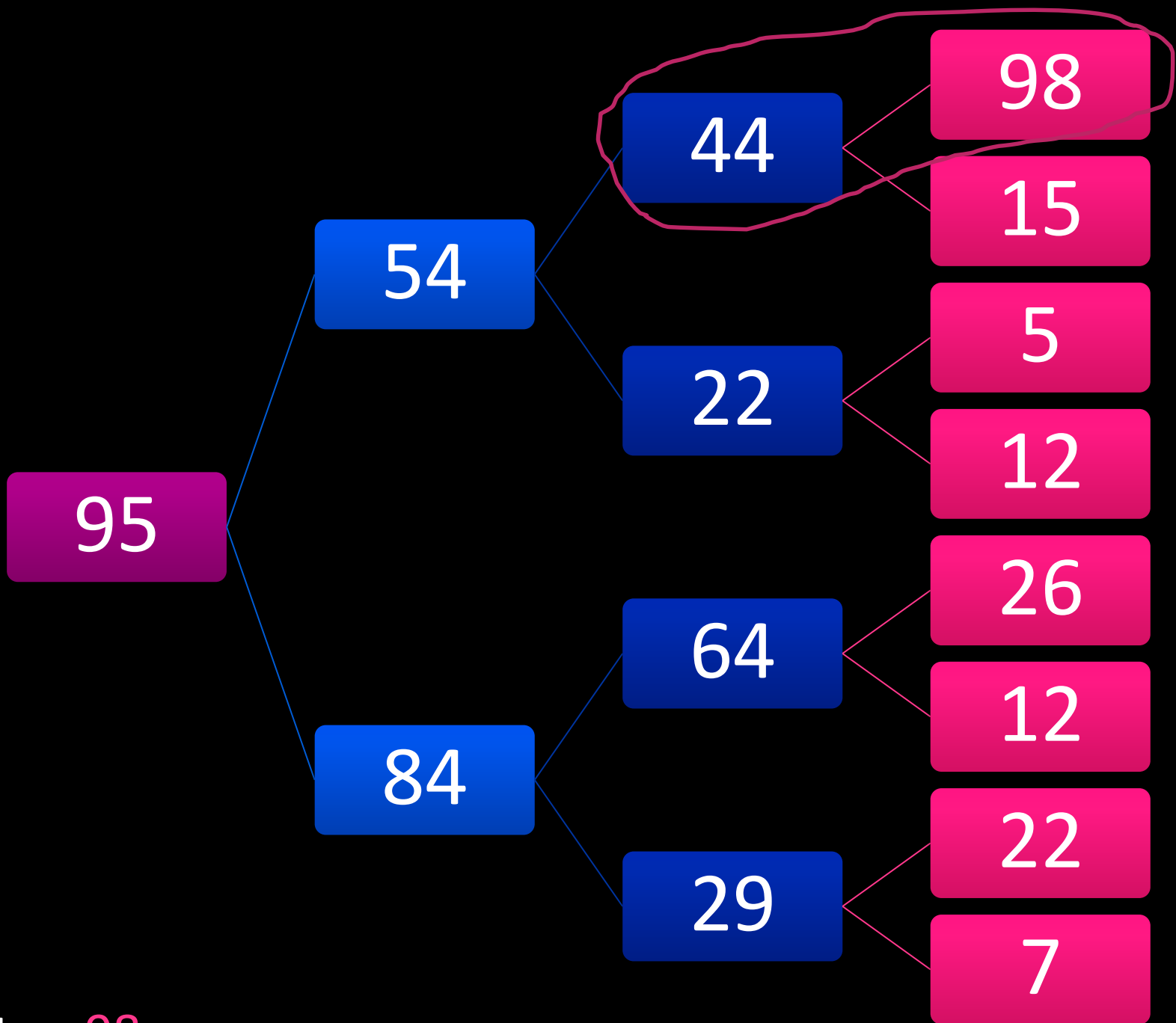
Insertar: 44, 98



Insertar: 44, 98

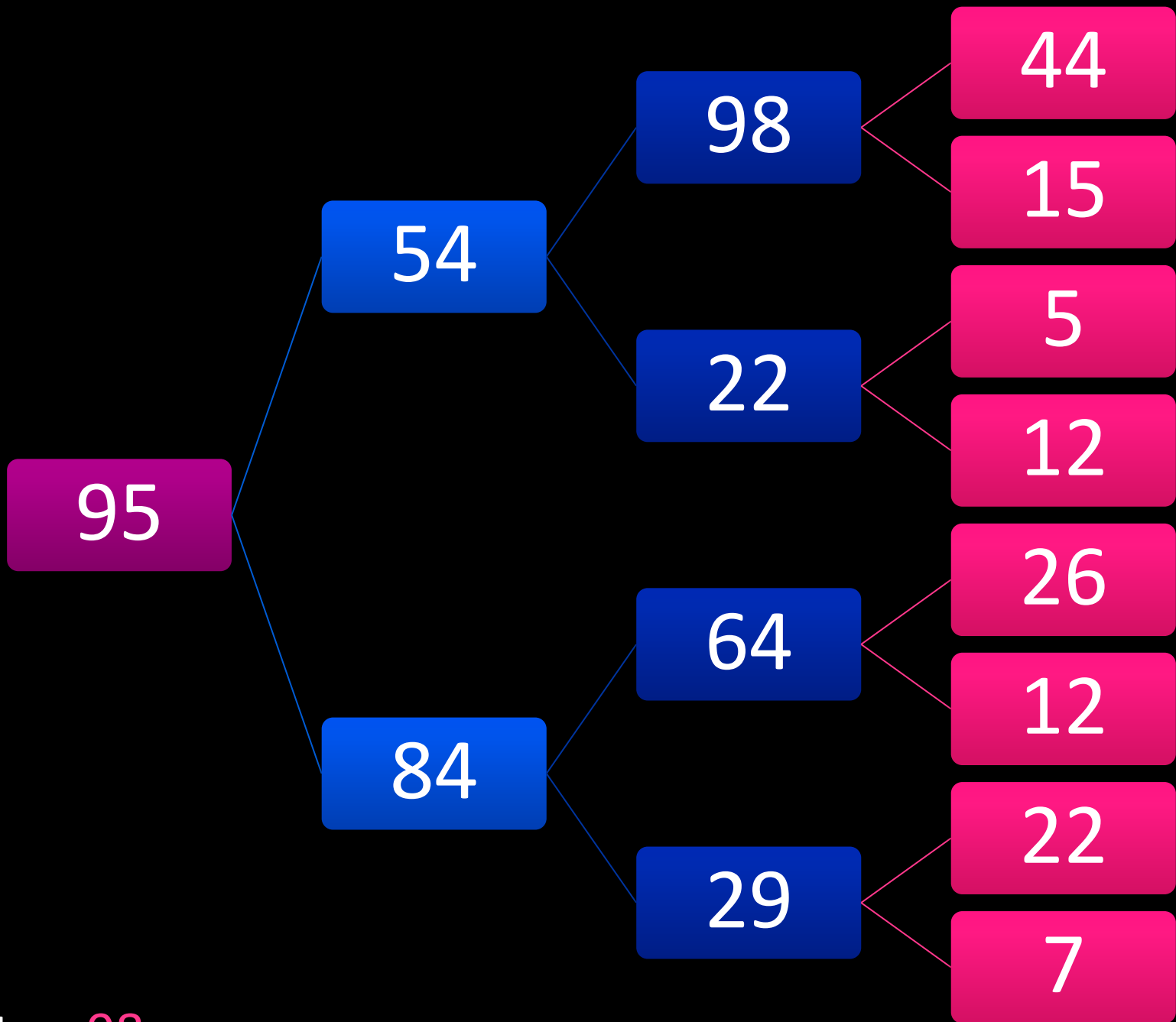


Insertar: 98

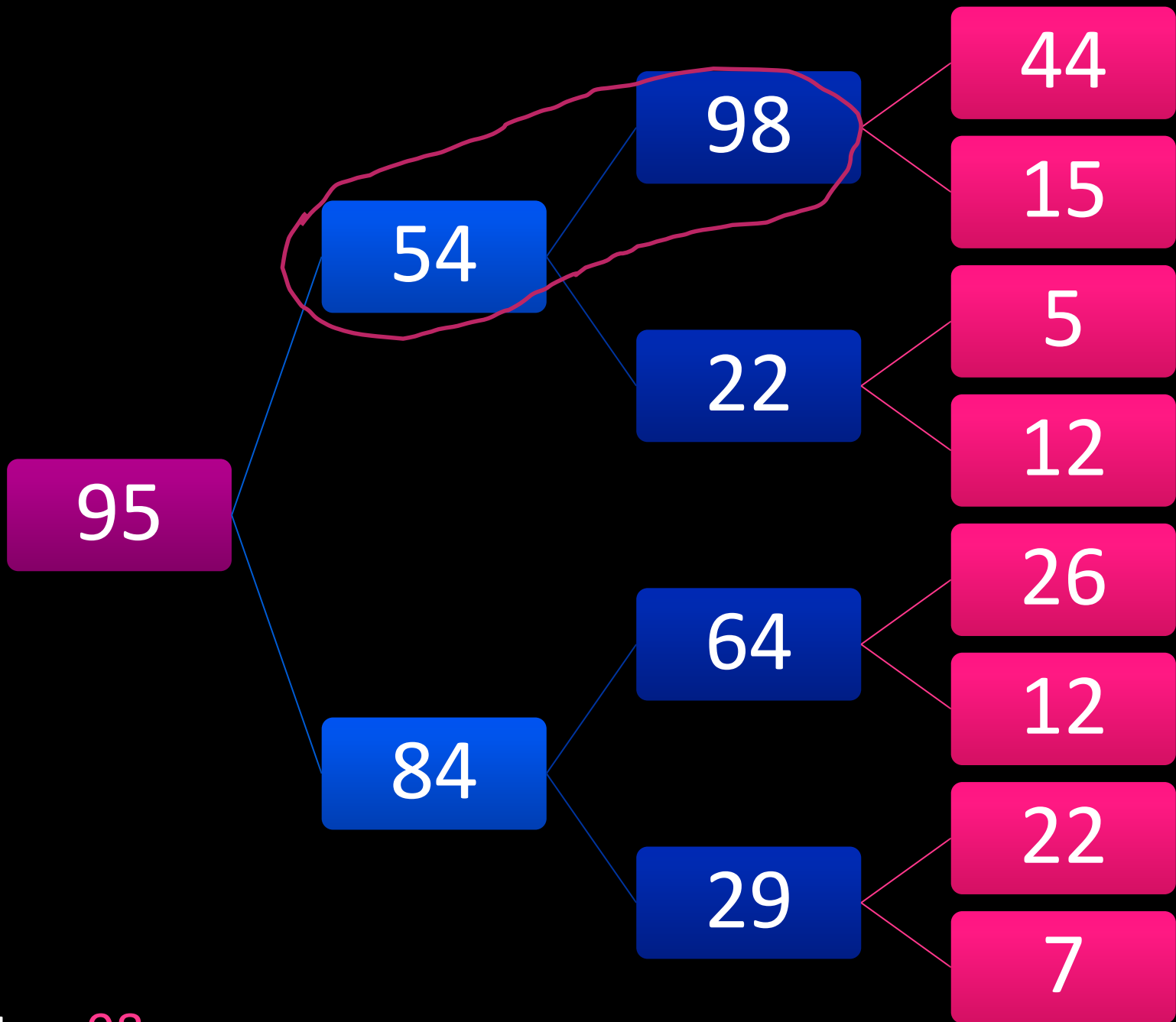


Insertar: 98

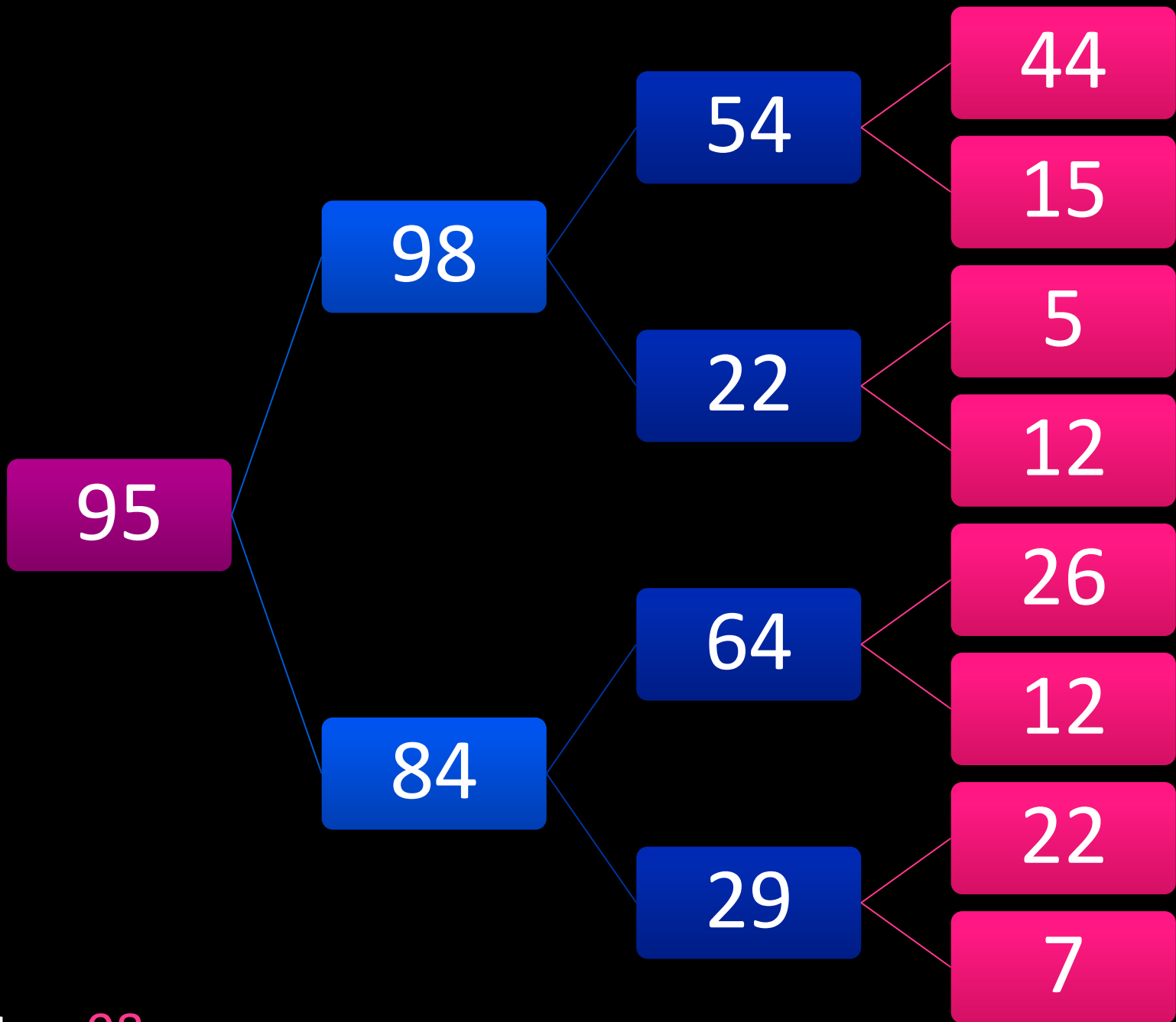




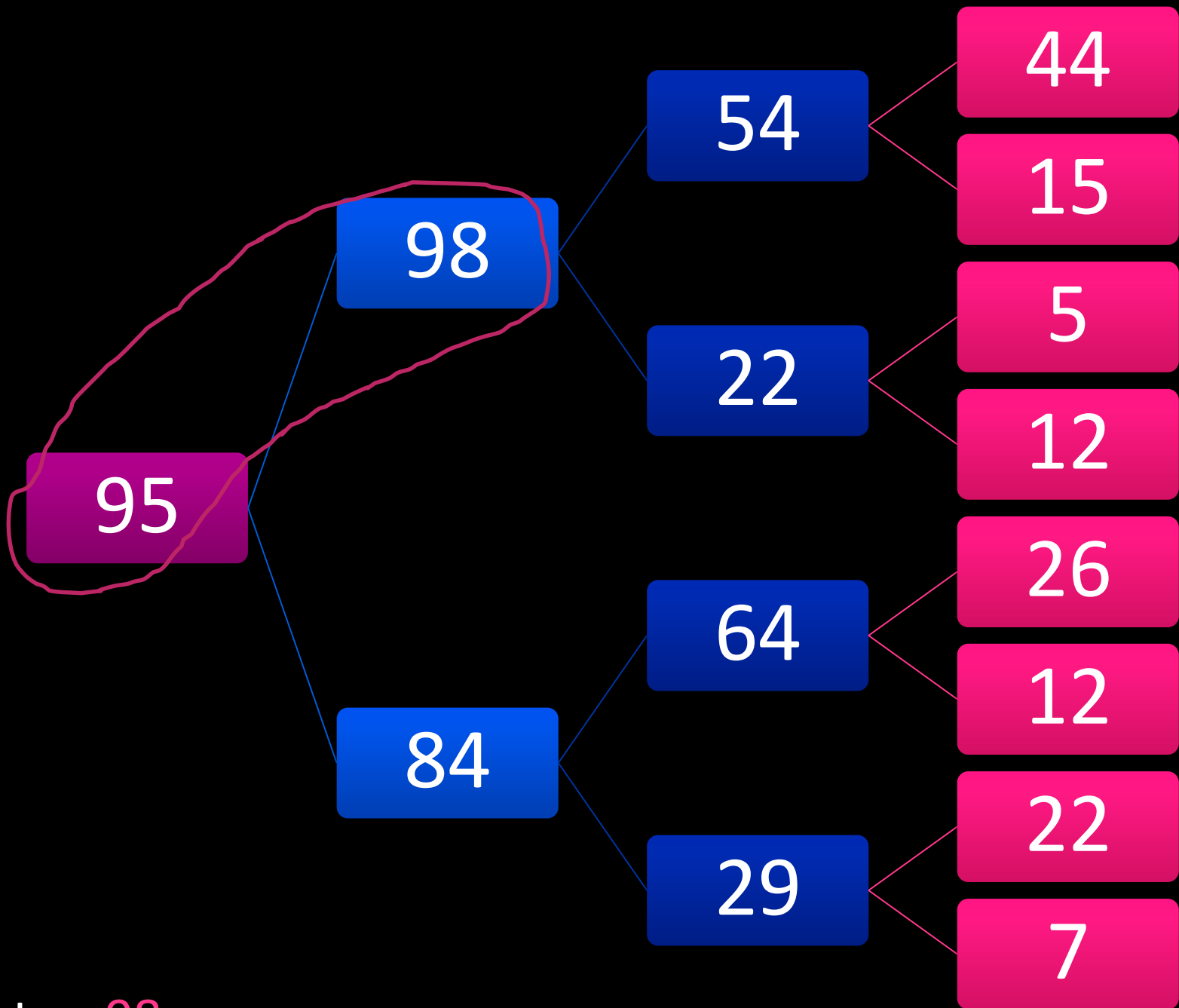
Insertar: 98



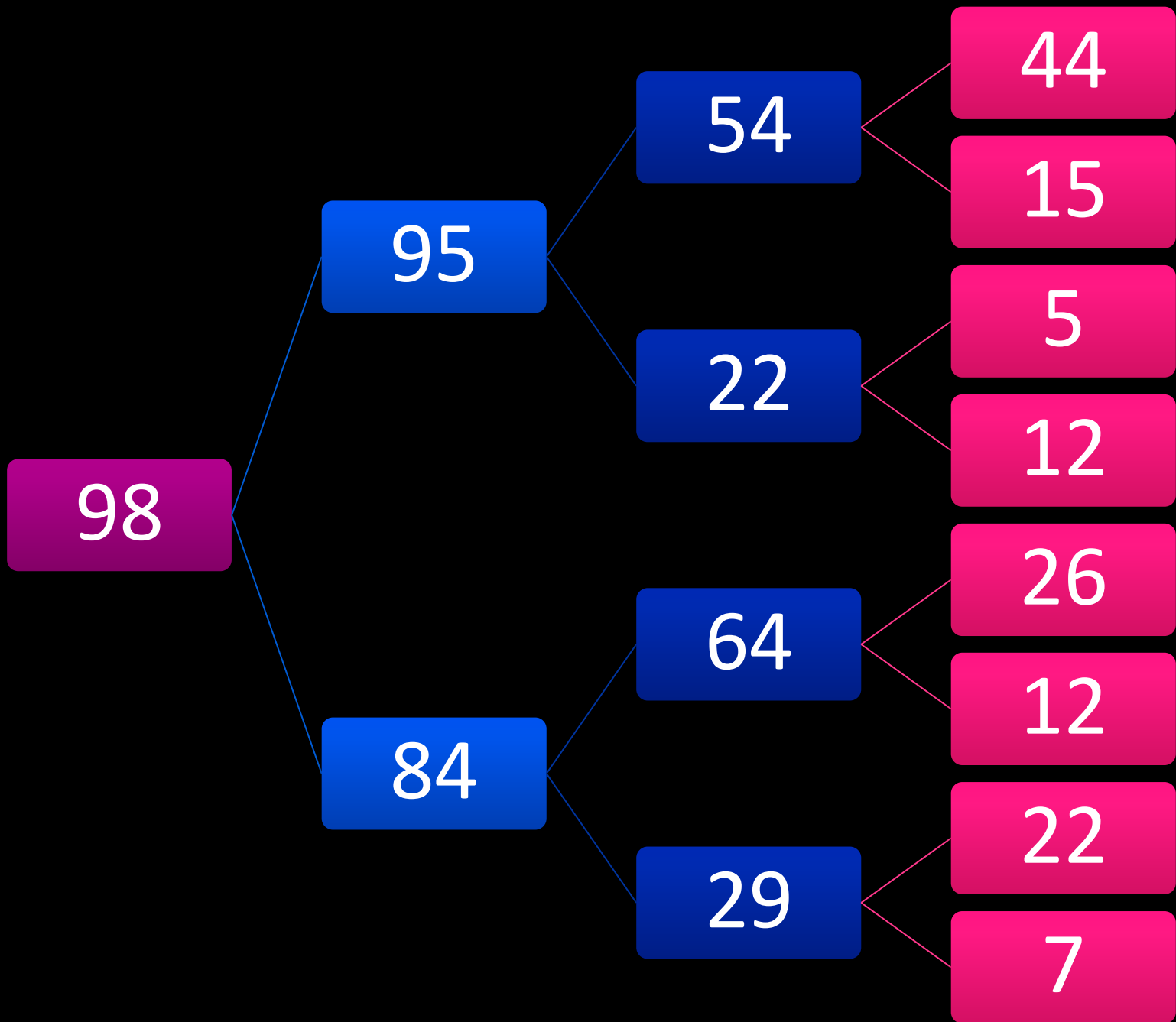
Insertar: 98



Insertar: 98

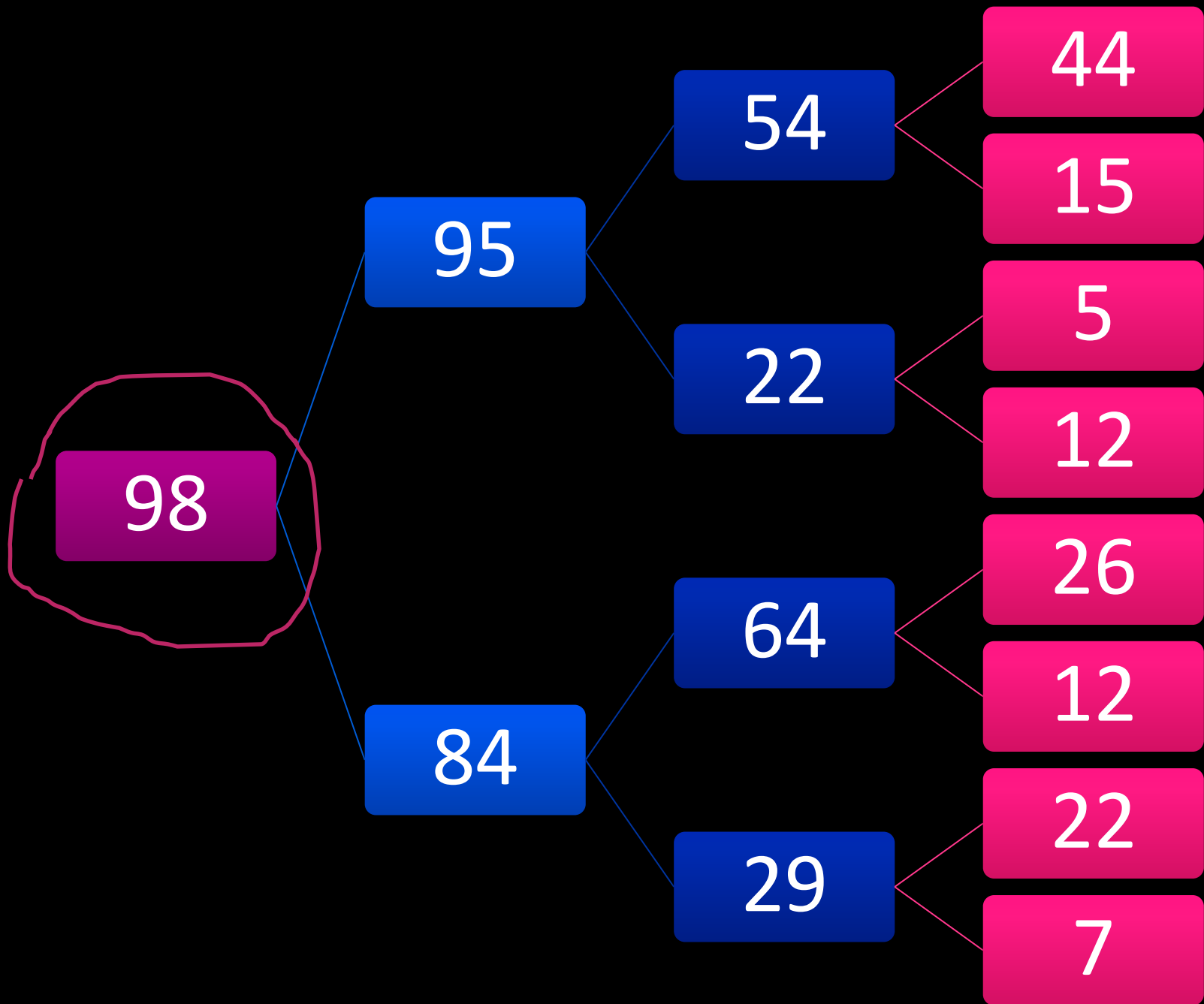


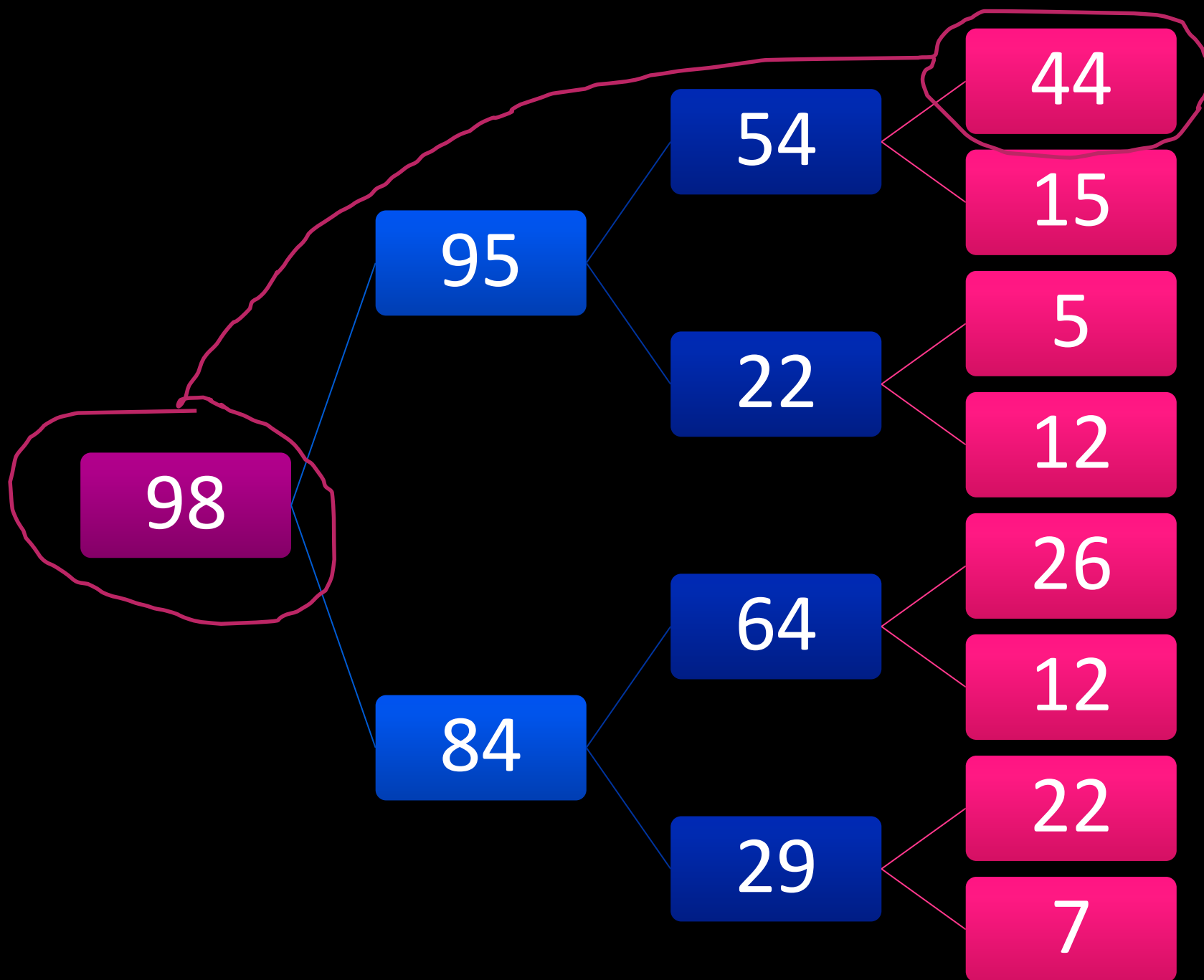
Insertar: 98



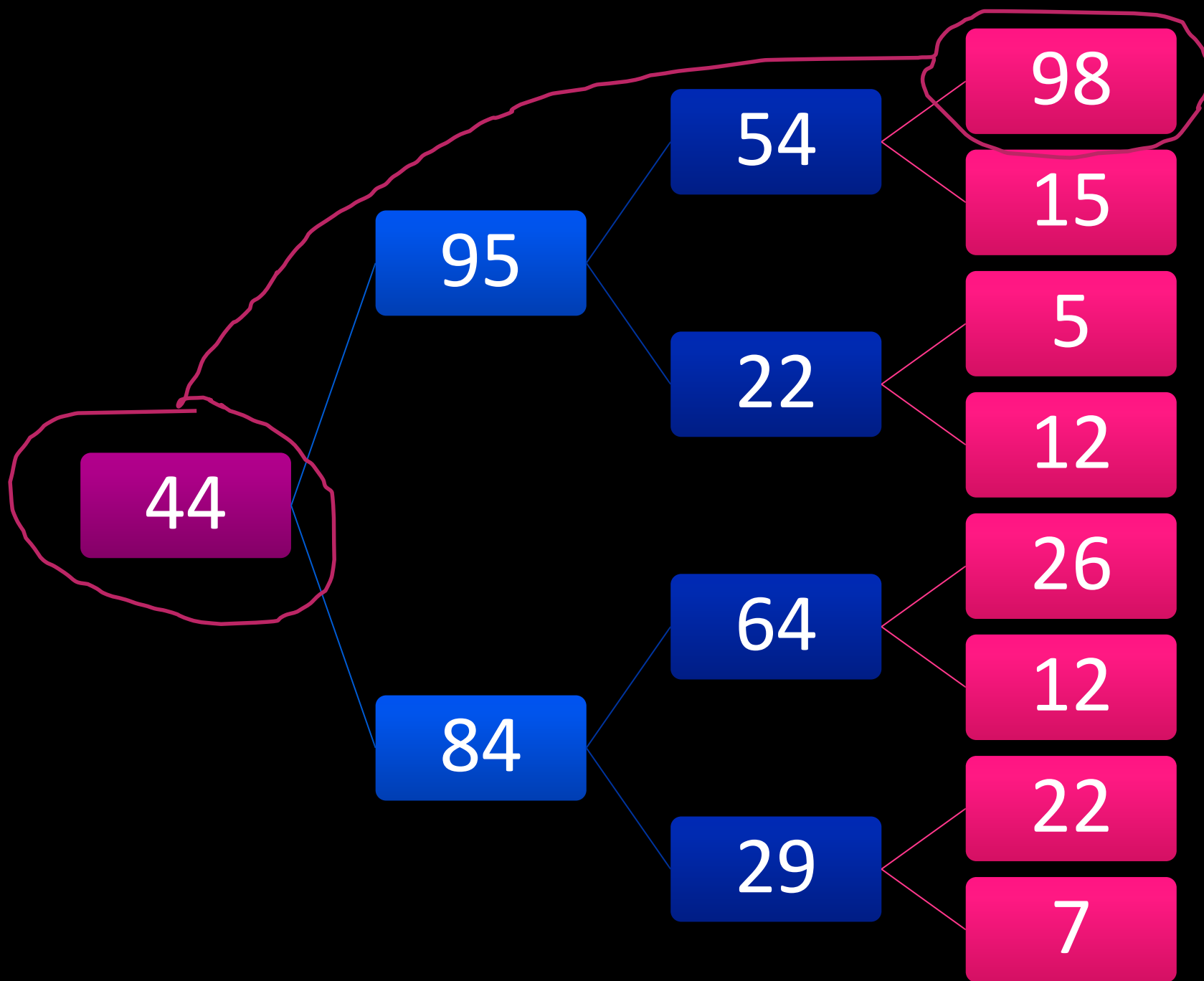
# Borrado de un heap

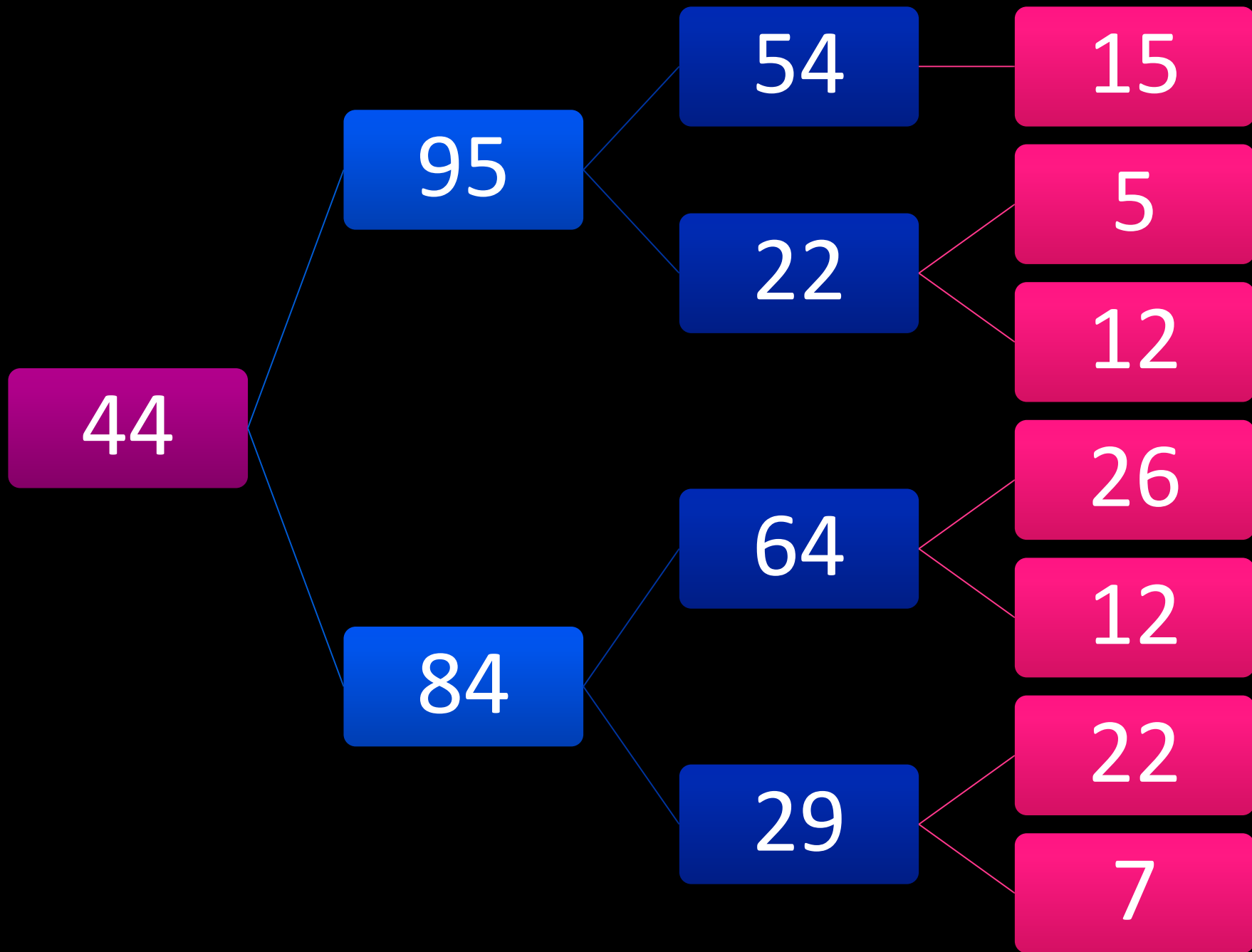
- Sólo puede eliminarse el elemento en la raíz
- Se **intercambia** la raíz con el último elemento del heap
- Se elimina el último elemento
- El nodo raíz debe intercambiarse recursivamente con su hijo mayor hasta que sus hijos sean menores o el nodo sea una hoja

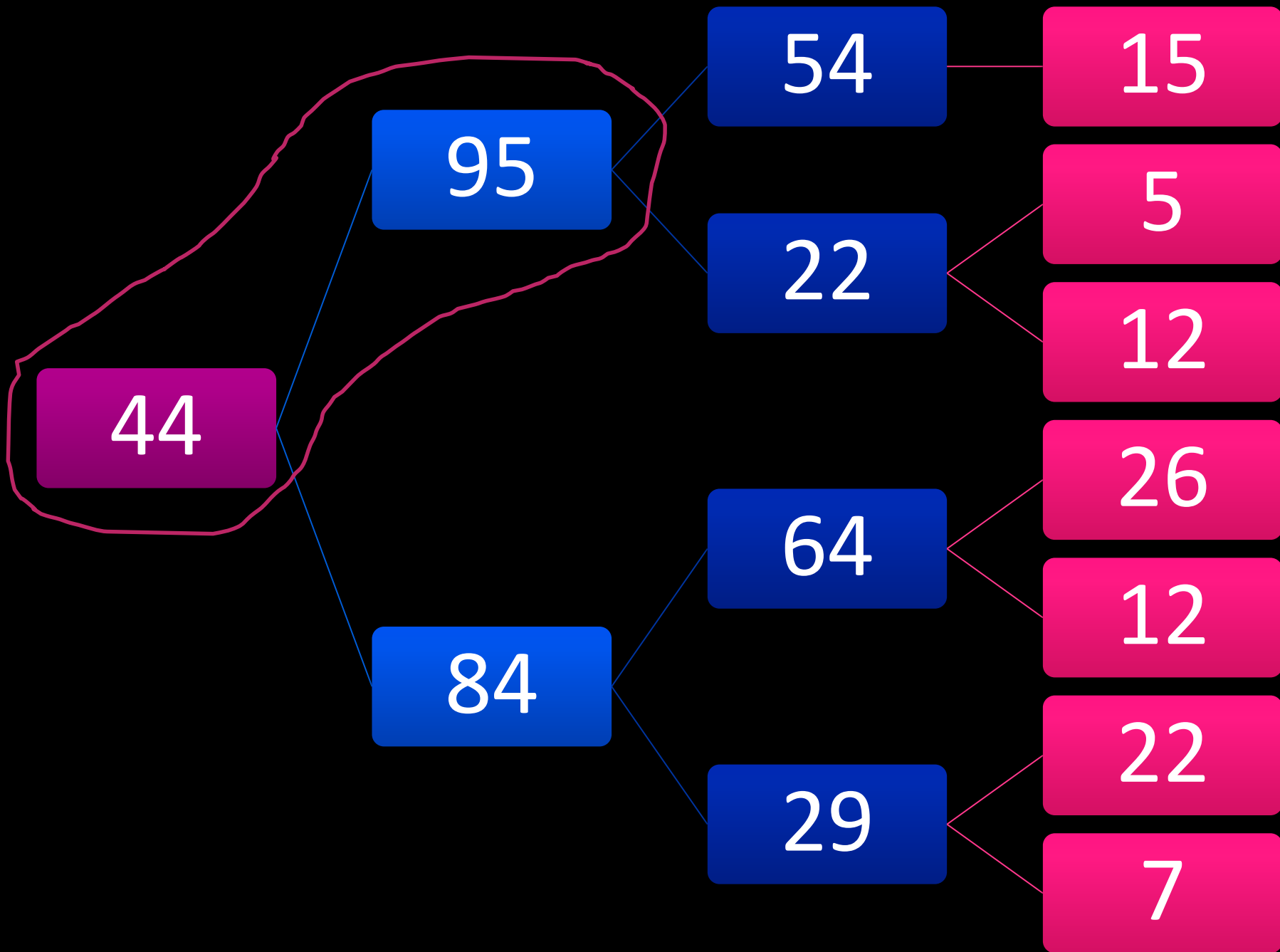


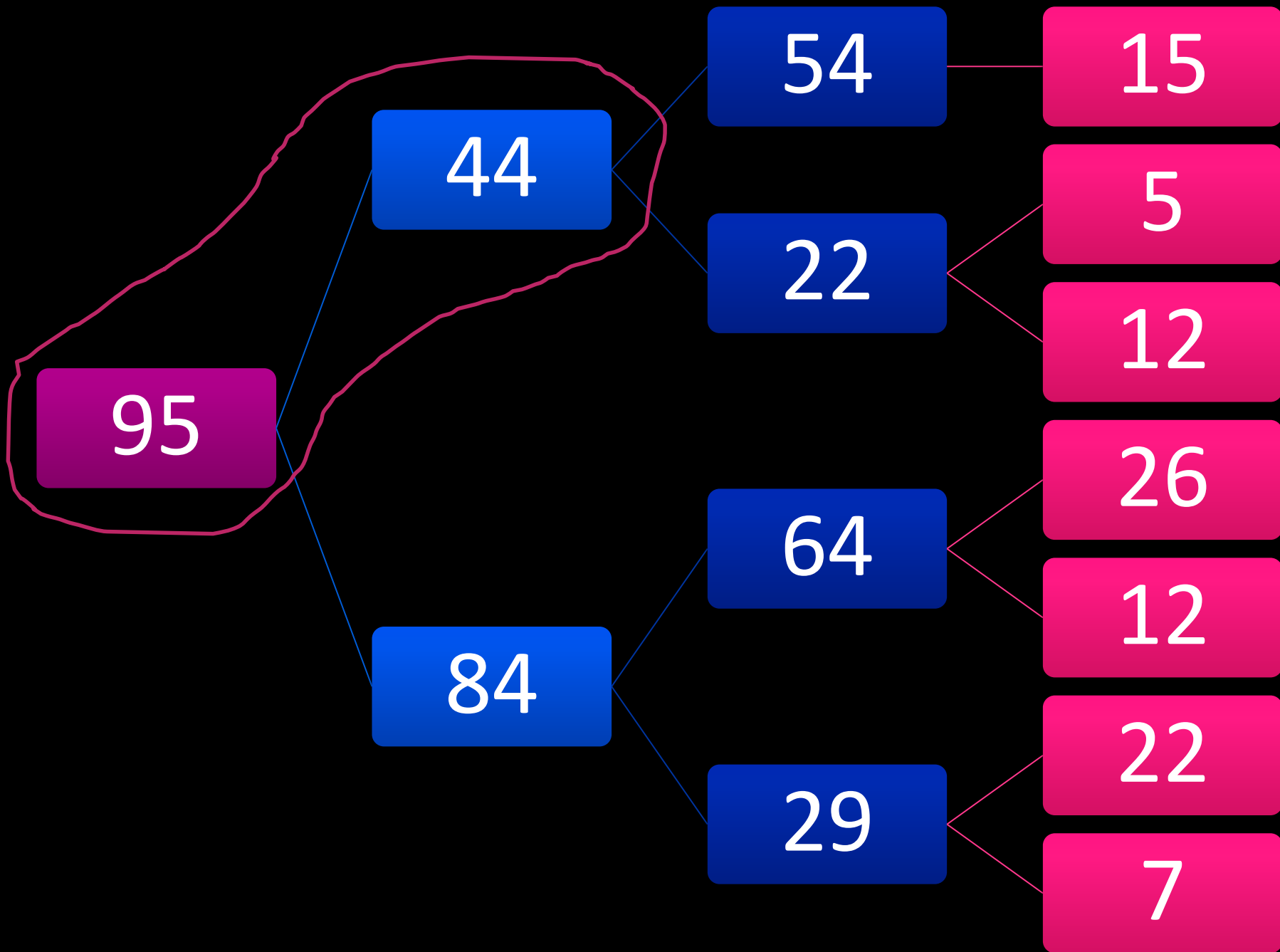


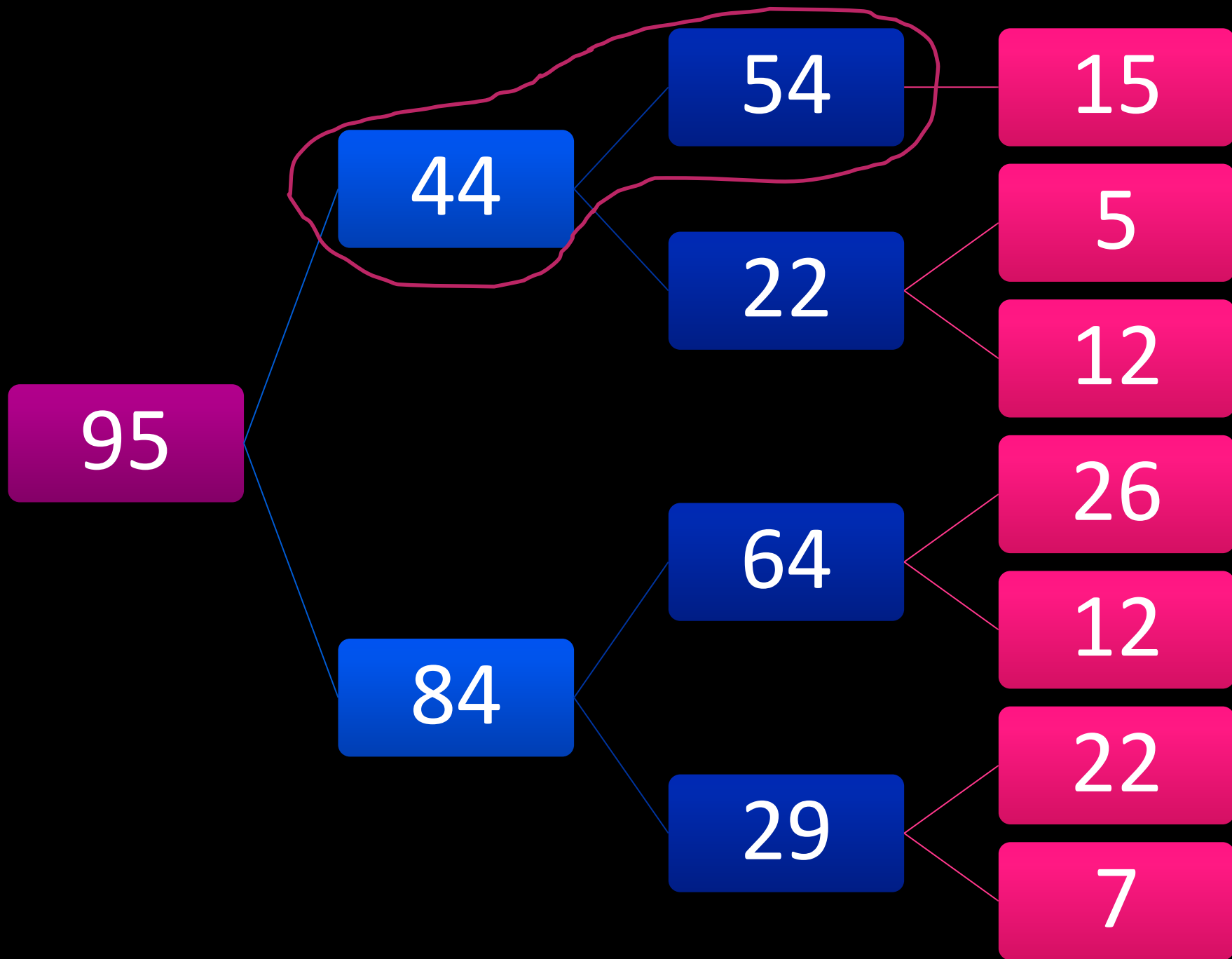


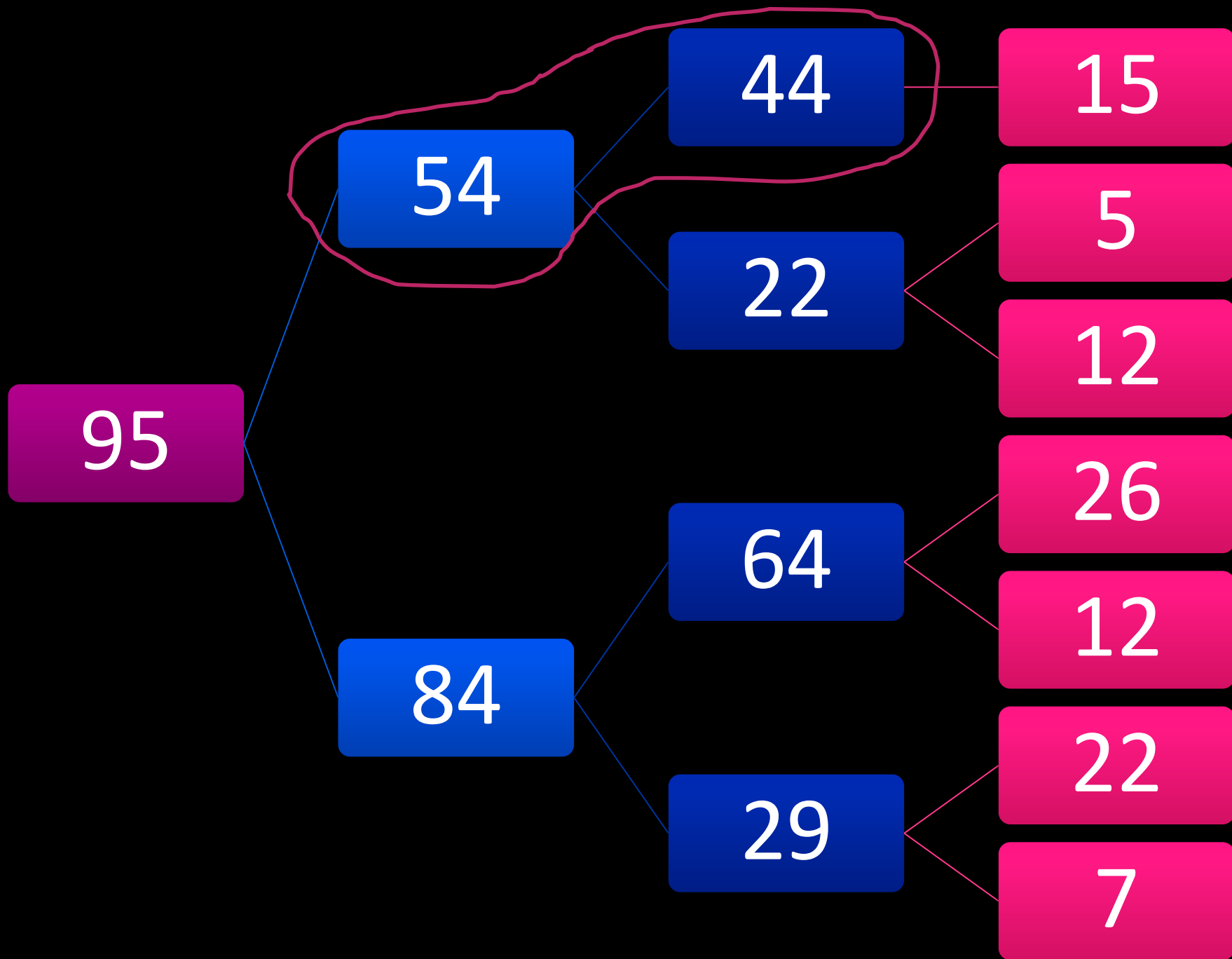


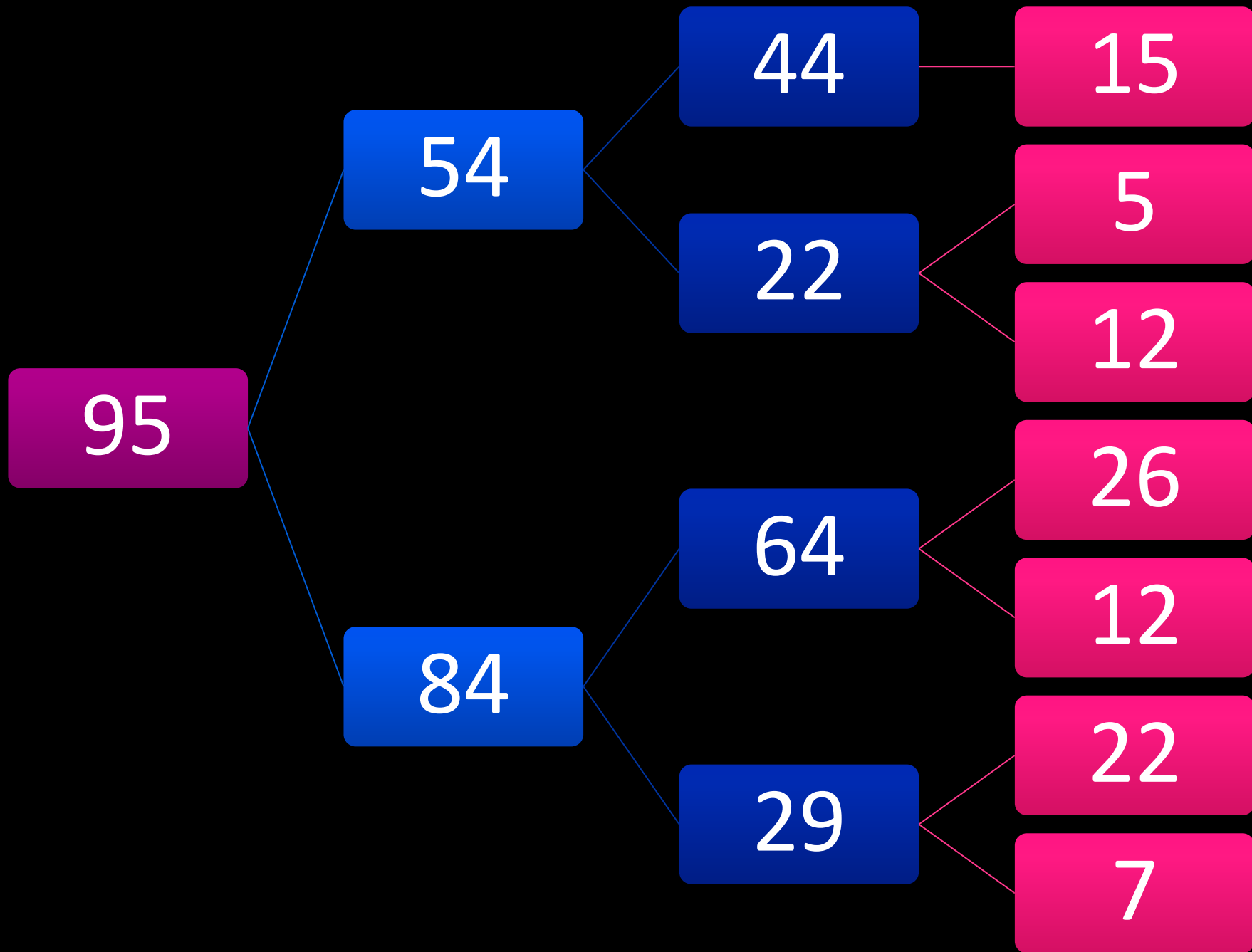


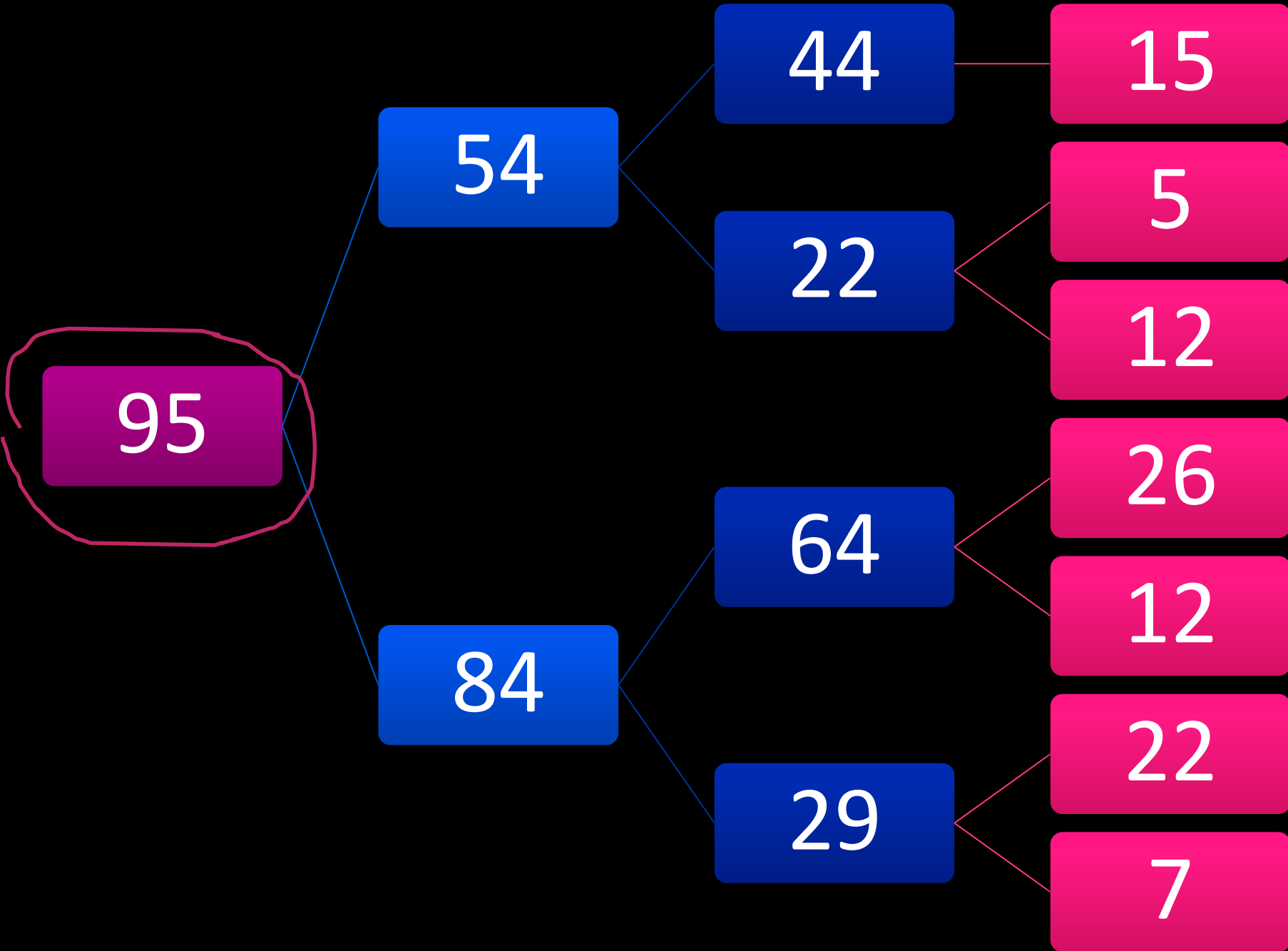




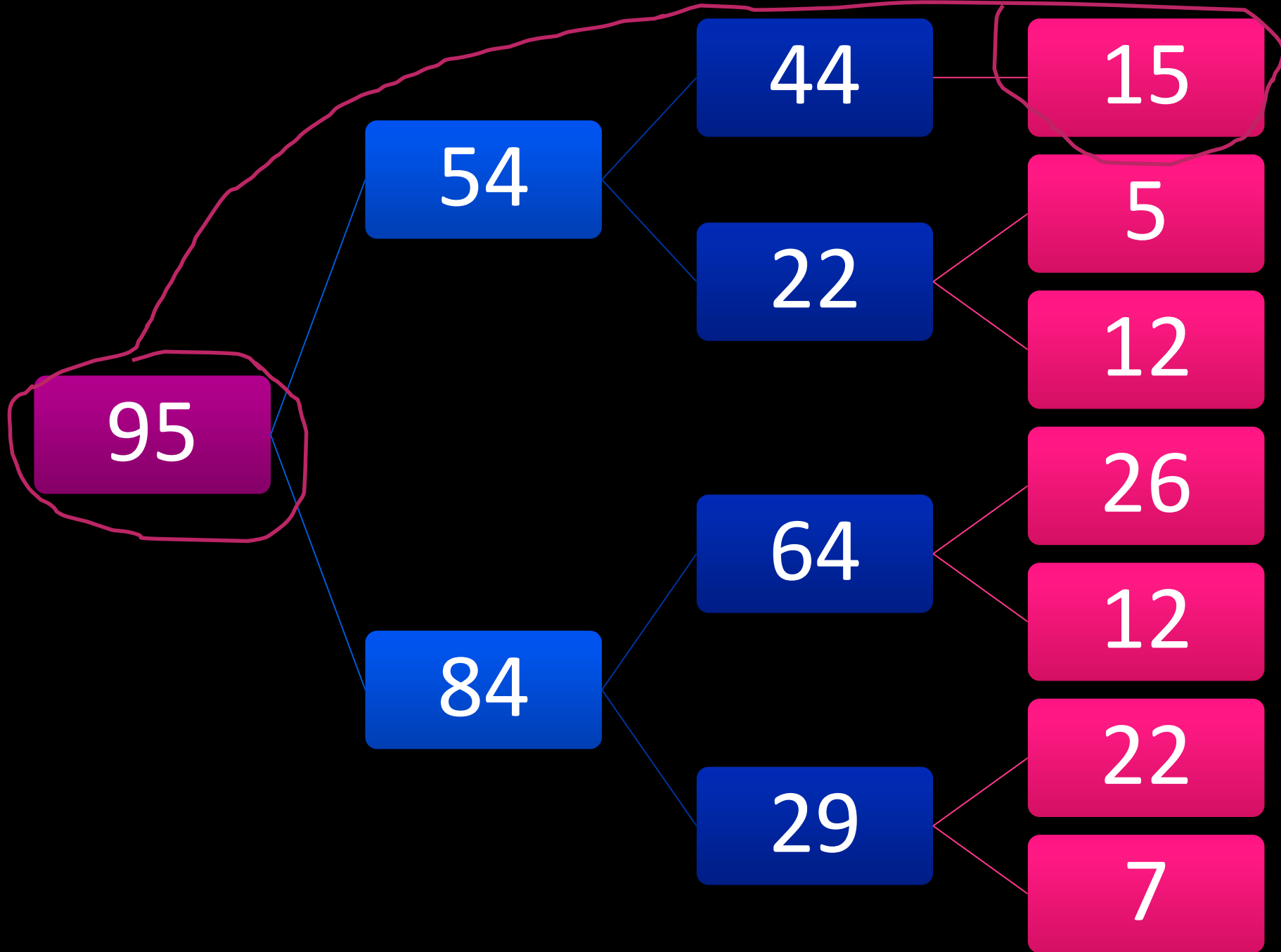


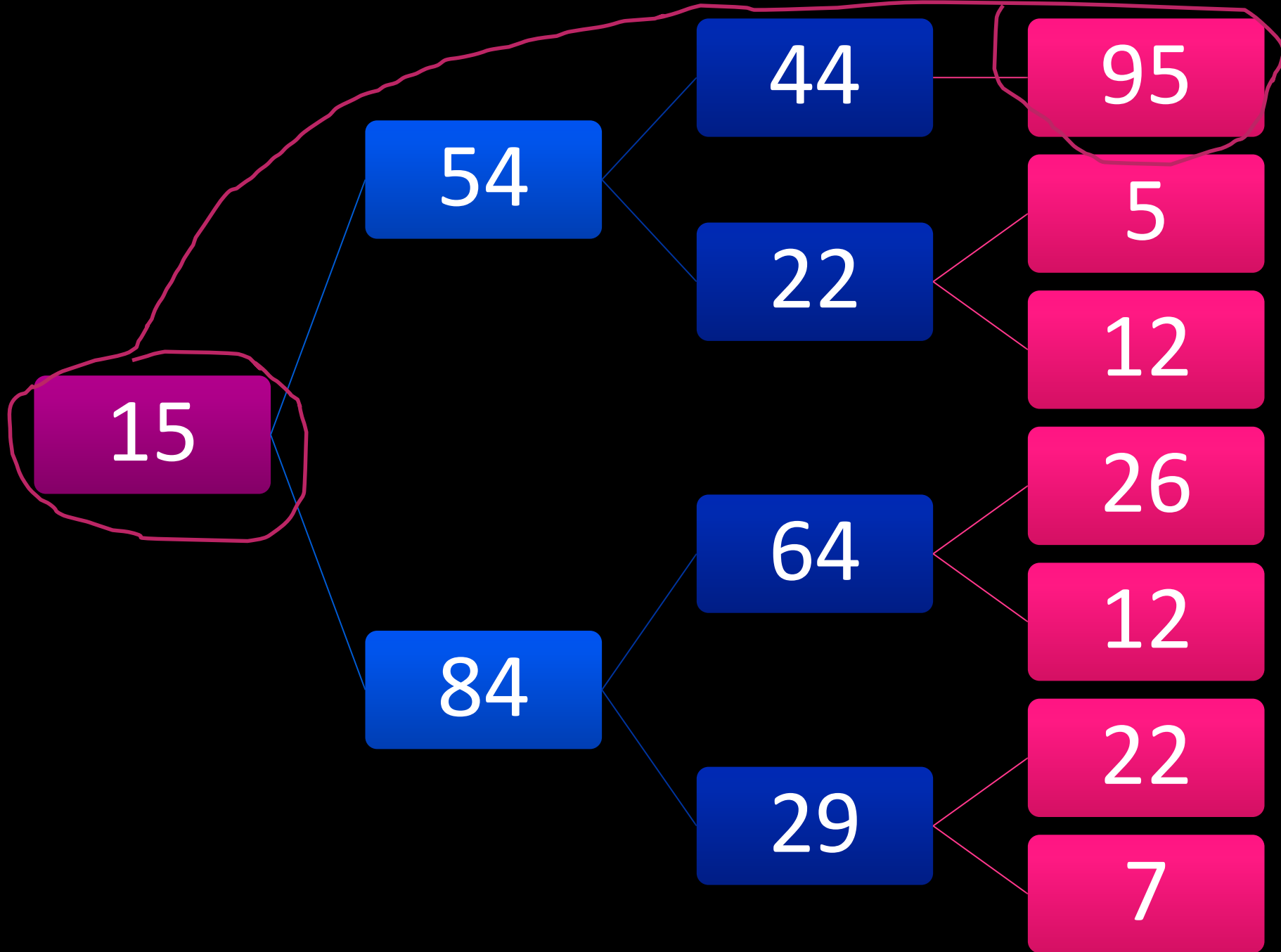


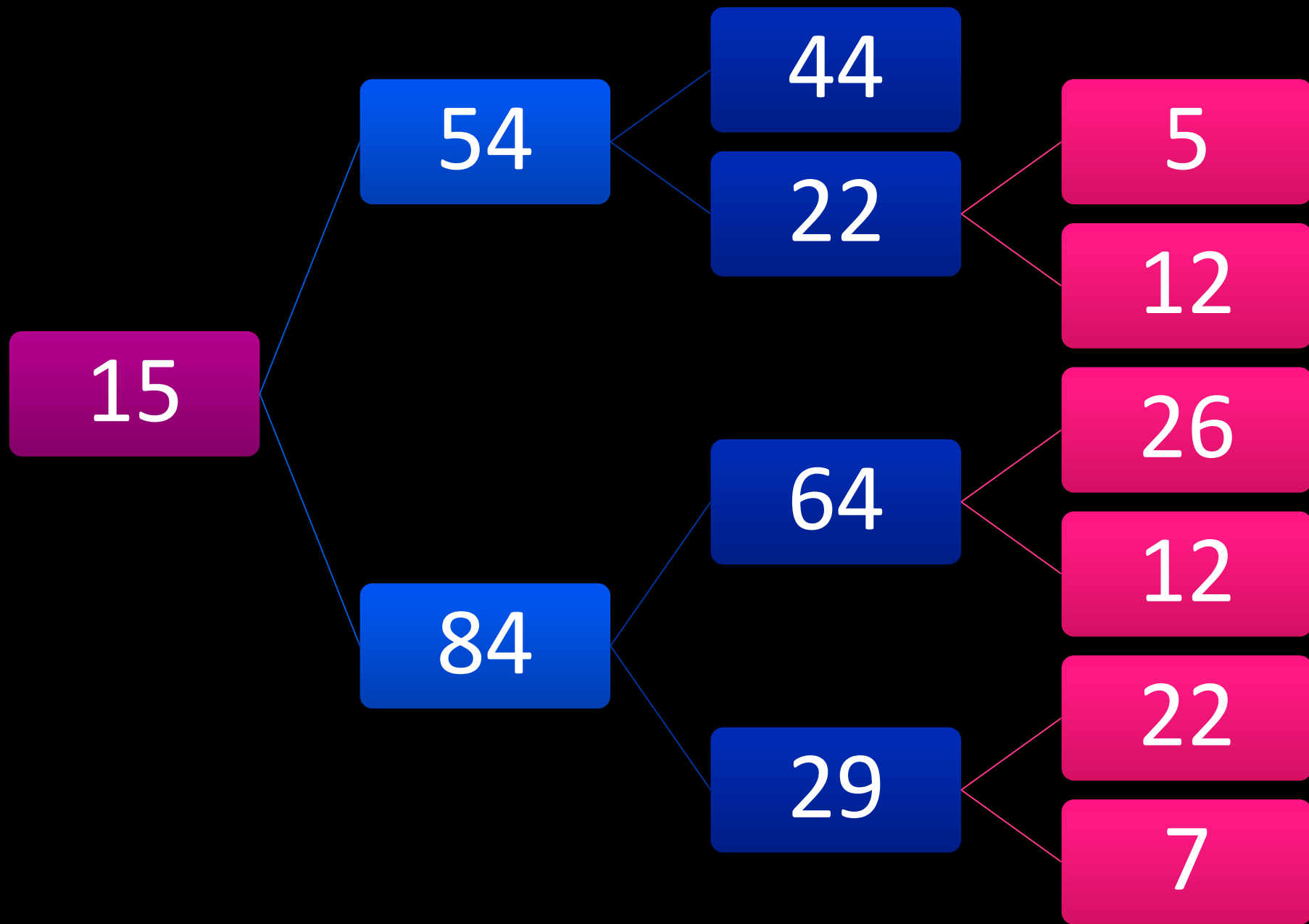


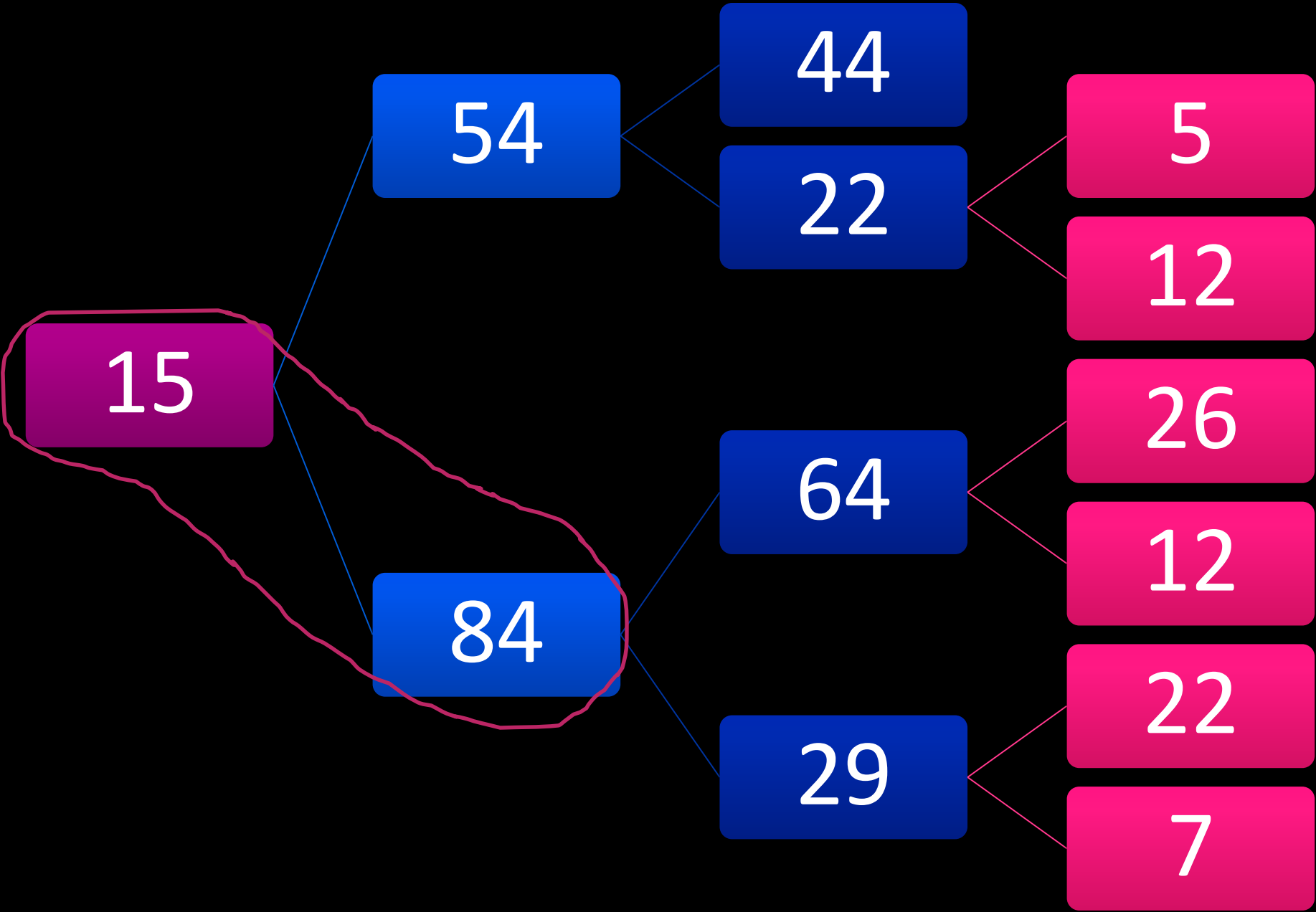


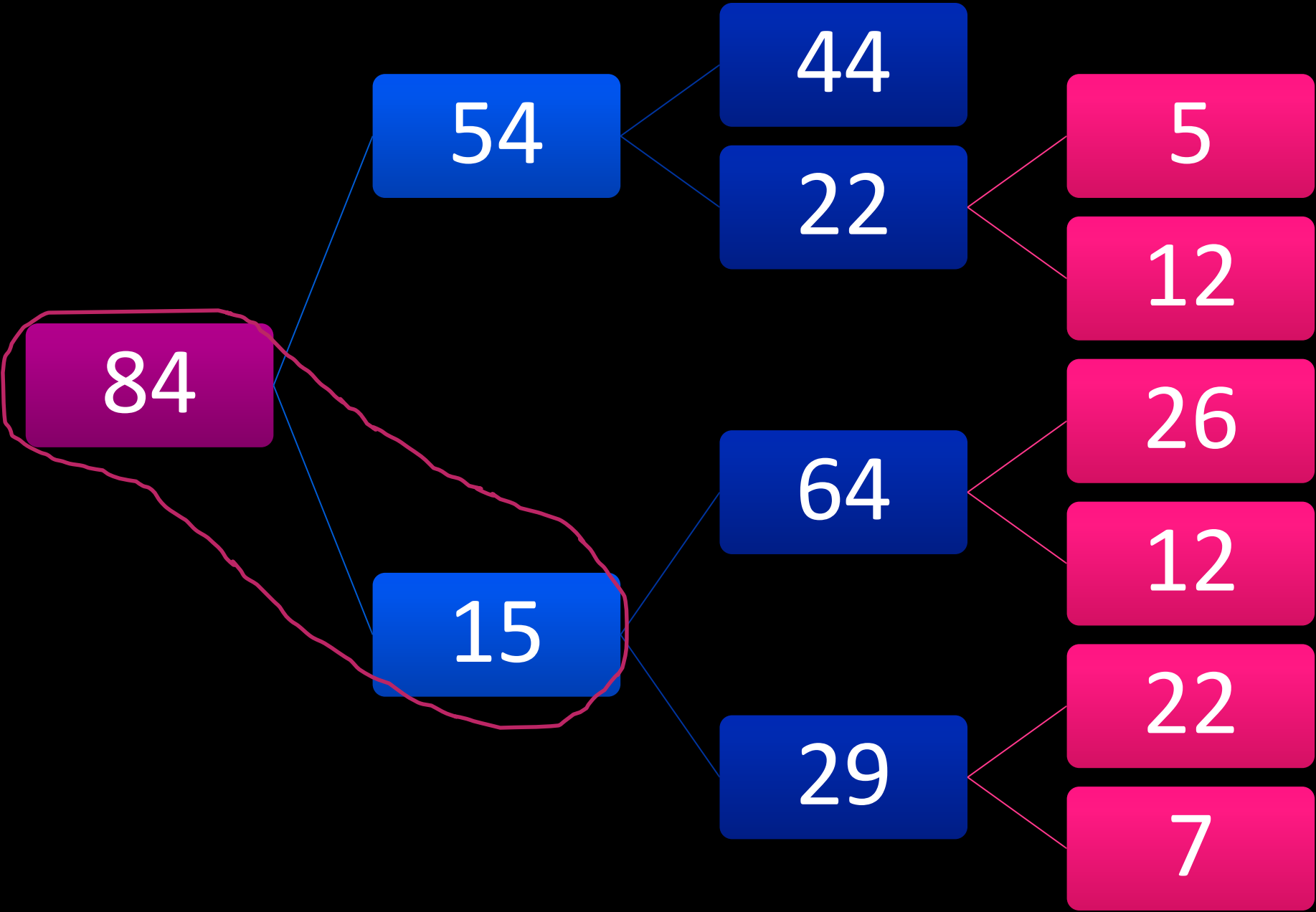


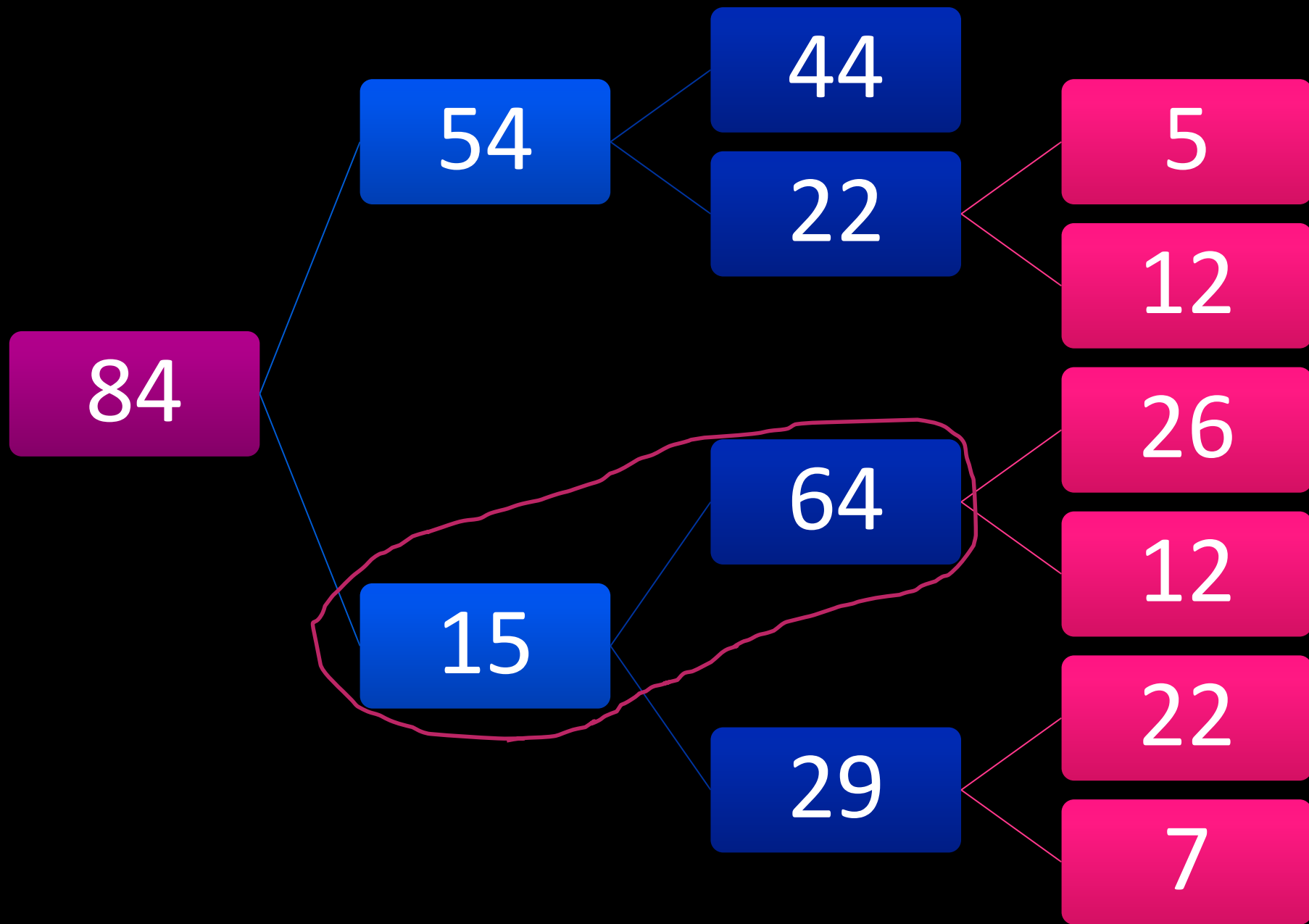


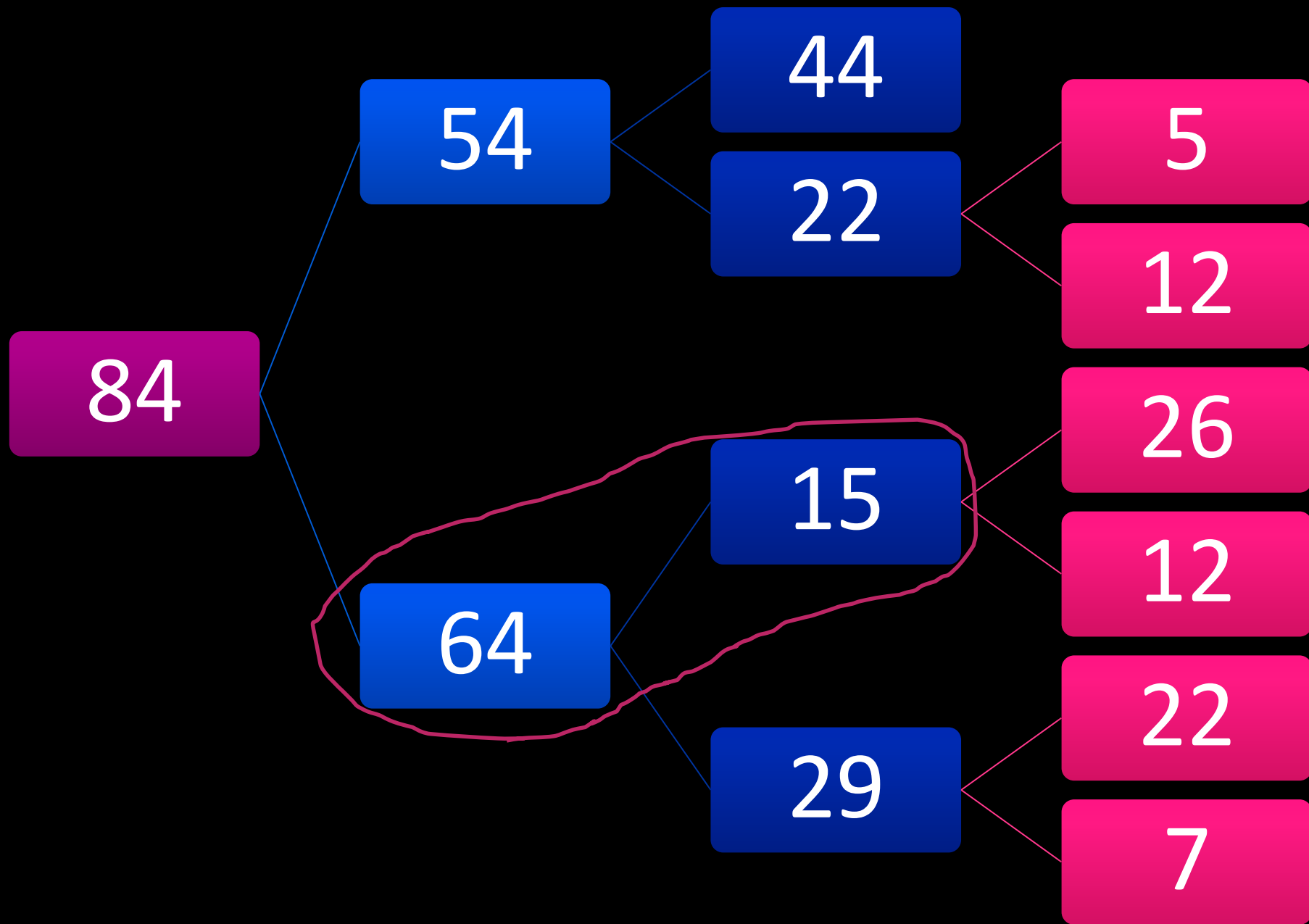


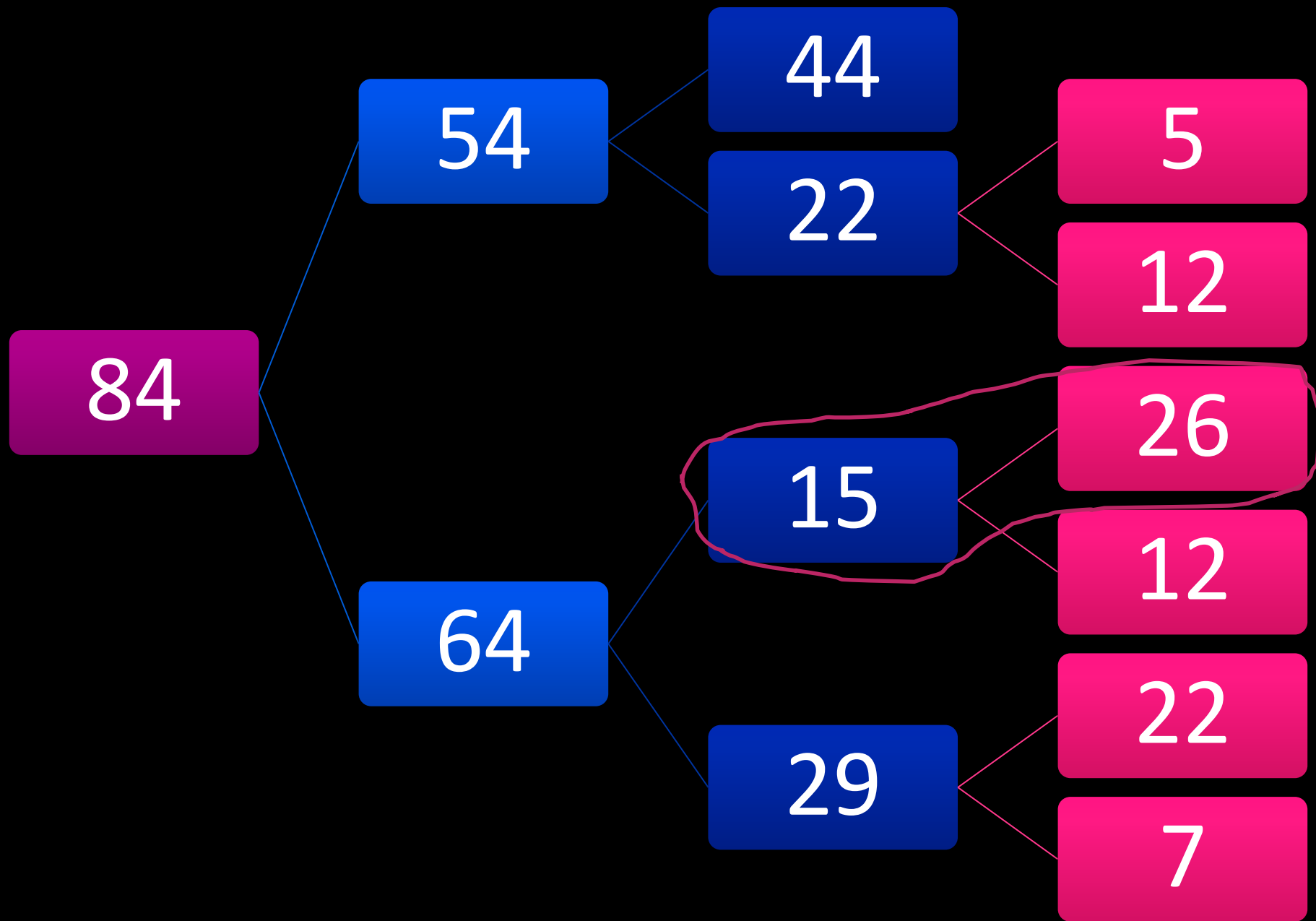




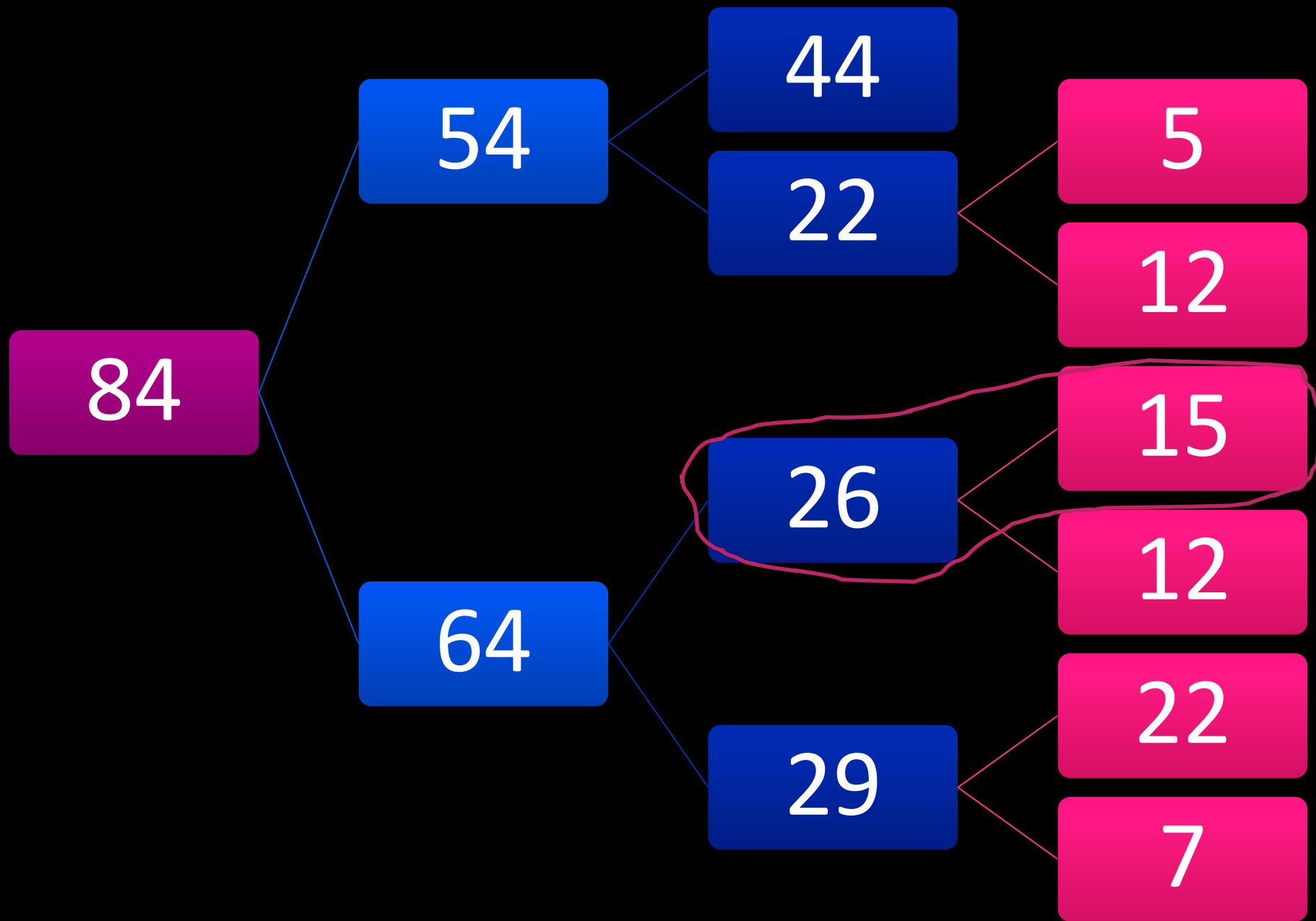


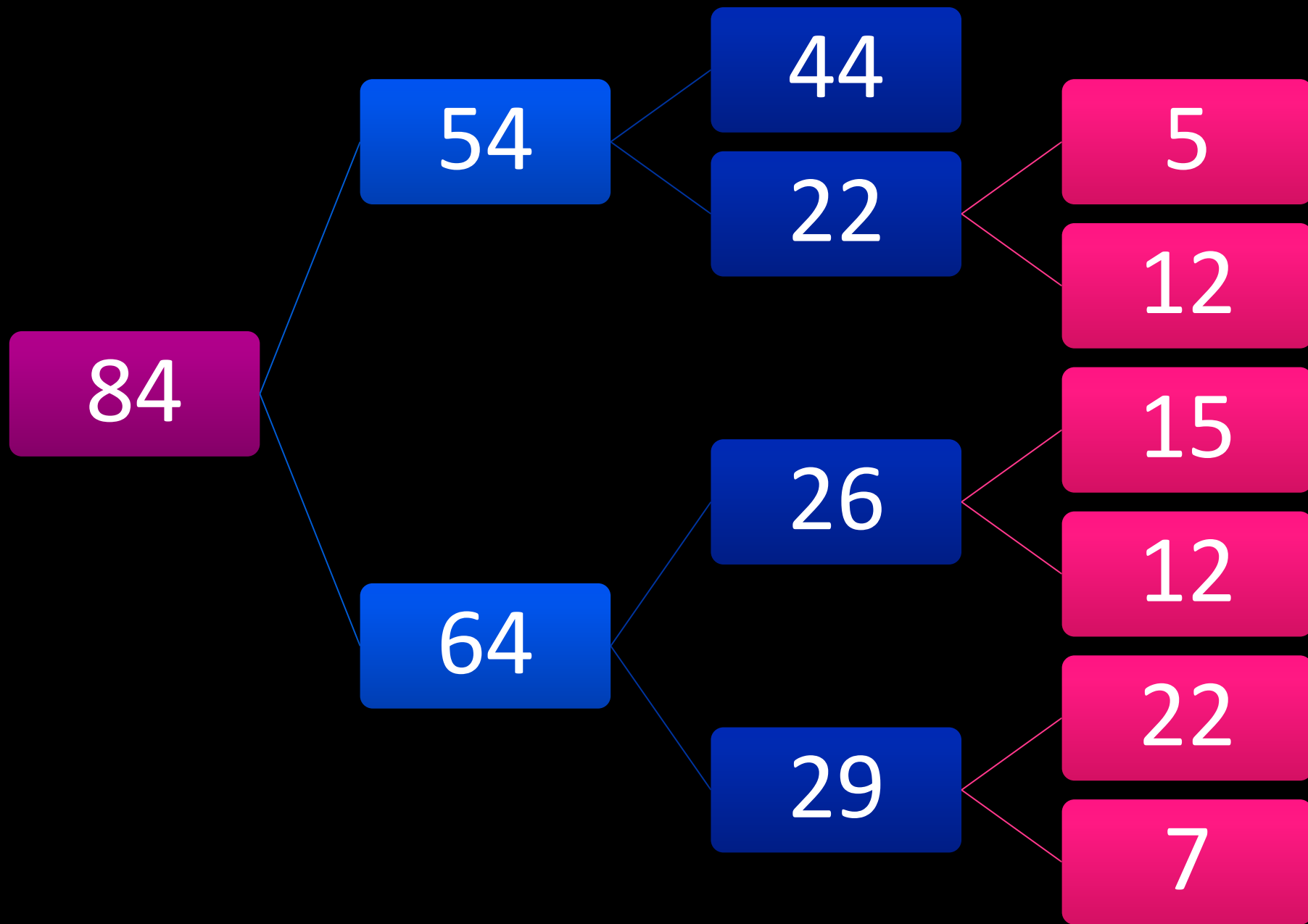








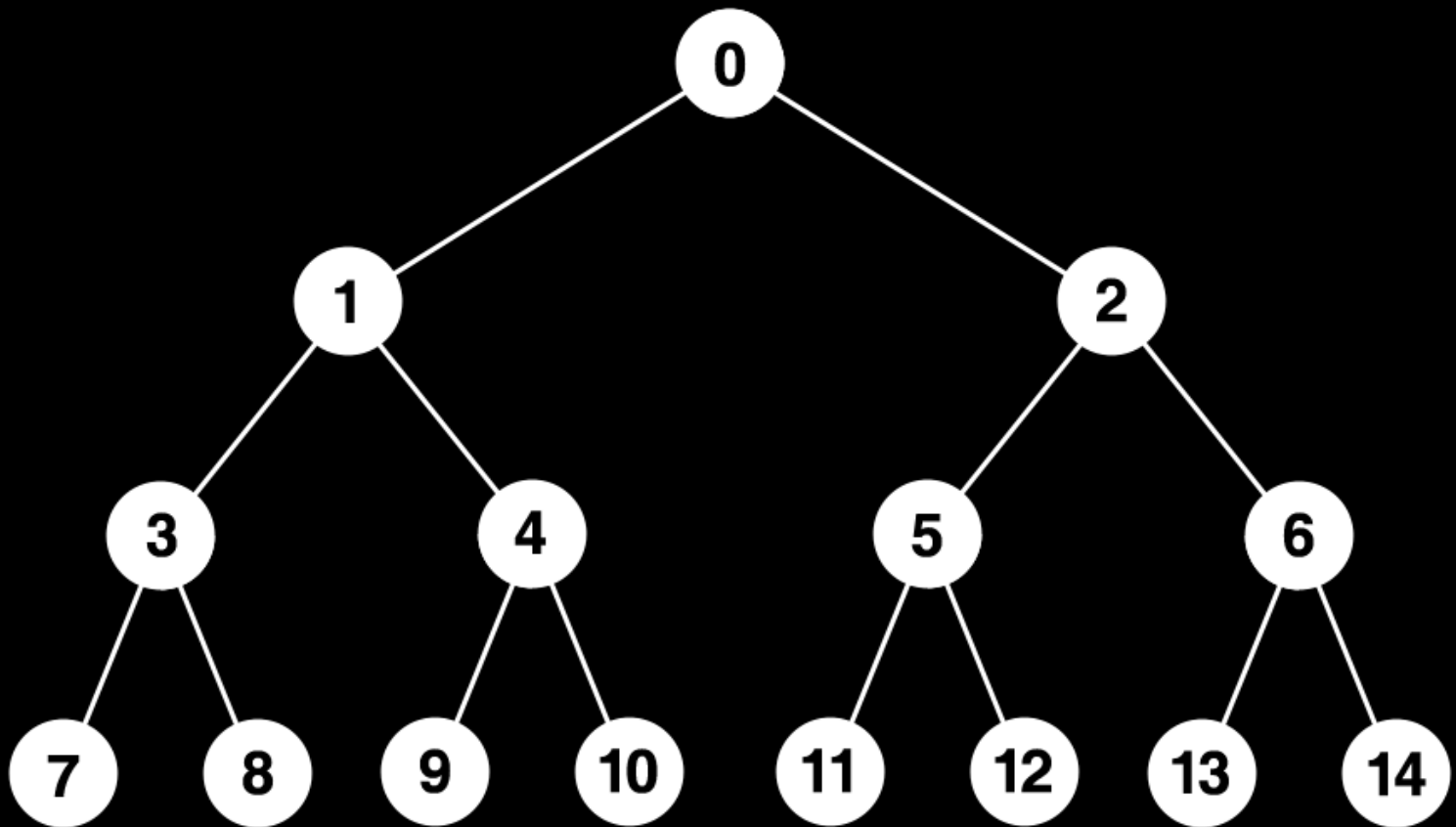




# Implementación de un heap

- No hay que confundir la presentación lógica de un heap con su implementación física
- Normalmente se implementa con un arreglo
- Esto es posible porque un heap siempre es un árbol binario completo

- Los **hijos** de un nodo en la posición  $x$  de un arreglo se encuentran en las posiciones
  - $2x+1$
  - $2x+2$
- El **padre** de un nodo en la posición  $x$  se encuentra con la parte entera de
  - $(x-1)/2$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	m
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	-----	---

n=15

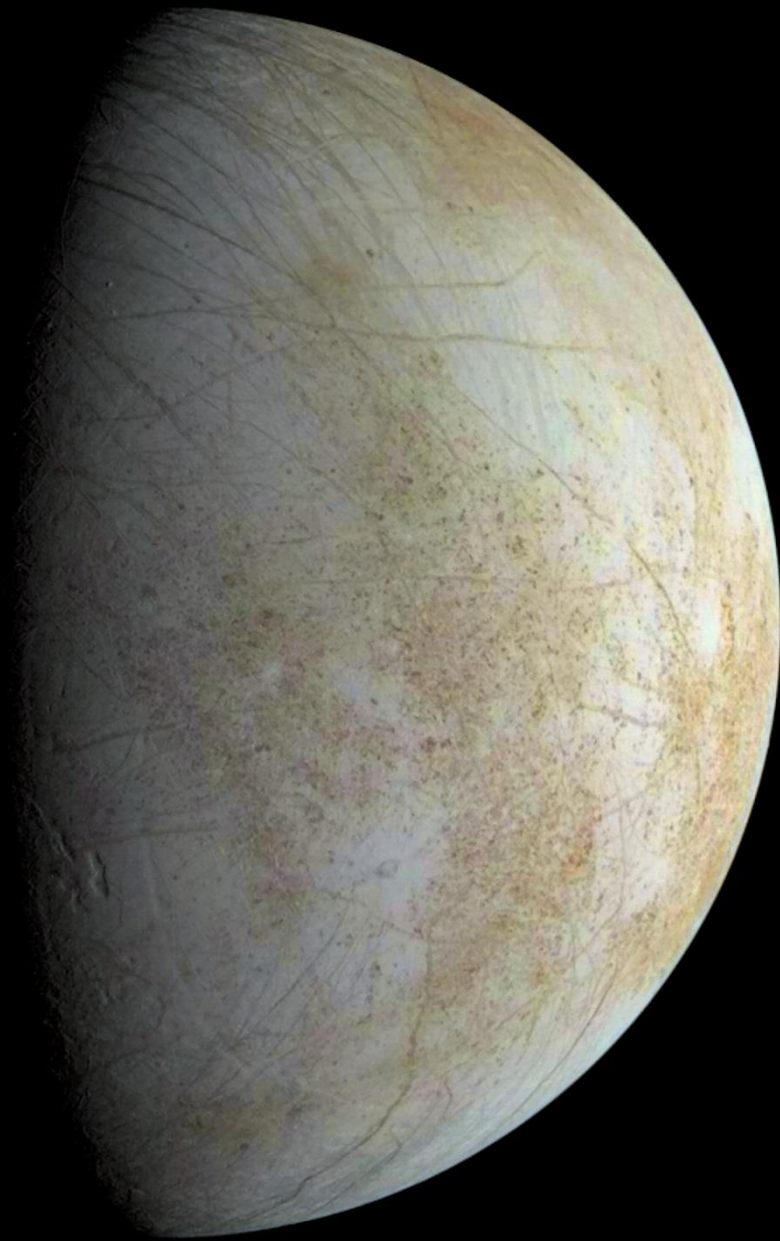
- `isLeaf(pos)`
  - Dice si un elemento es un nodo hoja del heap
- `leftchild(pos)` y `rightchild(pos)`
  - Retornan la posición del hijo izquierdo y derecho de un nodo, respectivamente
- `parent(pos)`
  - Retorna la posición del padre de un nodo
- `insert(it)`
  - Inserta un elemento en la última posición heap y lo sube de nivel hasta la posición correcta
- `siftdown(pos)`
  - Método que baja un elemento a una posición adecuada después de un borrado
- `buildheap()`
  - Reordena un arreglo cualquiera para que sea un heap
- `remove(pos)`
  - Elimina un elemento dentro del heap
- `removefirst()`
  - Elimina el elemento en la raíz del heap

- [72, 46, 24, 44, 39, 18, 21, 18, 56, 57]
- [71, 52, 84, 98, 35, 34, 85, 92, 90, 88]

# Lectura

- Shaffer – Capítulo 5
- Goodrich – Secciones 7.3 y 8.3





Árboles binarios

Mauricio Avilés