

B+ Tree 1

Today's Lecture

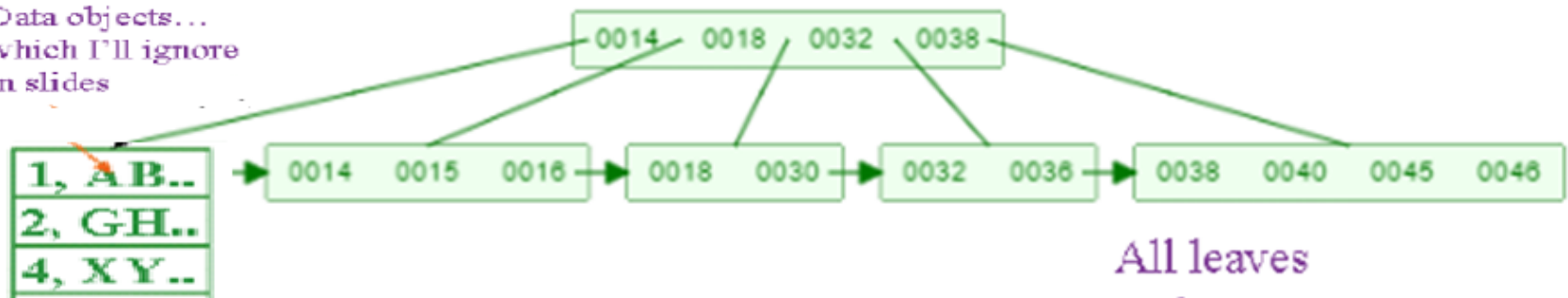
- Introduction to B+ tree
- Properties of B+ tree
- Insertion into B+ tree
- Examples

Definition of a B+ tree

- Internal nodes contain only keys
- Only leaf nodes contain keys and actual data
- Much of key structure loaded into memory irrespective of data object size
- Data actually resides in disk

An example B+ -Tree

Data objects...
which I'll ignore
in slides



All leaves
at the same
depth

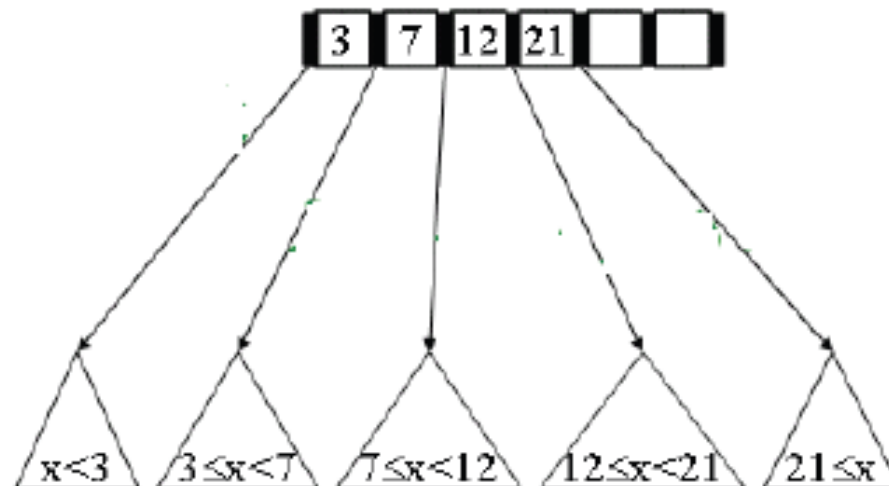
Properties of B+ -trees

1. **Balanced** : All leaves are at the same depth
2. **Root (special case) :**
 - ✓ Has between 2 and M children (or root could be leaf)
3. **Internal nodes :**
 - ✓ Store up to M-1 keys (max) and $M/2-1$ (min)
 - ✓ Have between $M/2$ and M children
4. **Leaf nodes :**
 - ✓ Where data is stored
 - ✓ All at the same depth
 - ✓ Contains between $M/2-1$ and M-1 data items

6. Keys are stored in ascending order inside node.

7. Order Property :

- ✓ Sub tree between two keys x and y contains leaves with values v such that $x \leq v < y$



8. Leaf nodes are connected to each other

	min	max
Children	• $M/2$ (all internal nodes except root node)	• M (all nodes)
Keys	• $M/2-1$ • $M/2-1, M/2$	• $M-1$ • $M-1$


Insertion into B plus-tree

1. Insert the key in its leaf in sorted order
2. If the leaf ends up with L keys, **overflow!**
 - Split the leaf into two nodes:
 - original with $\lfloor M/2 \rfloor$
 - new one with $\lceil M/2 \rceil$
 - Add the new child to the parent
 - If the parent ends up with M keys, **overflow!**
3. If an internal node ends up with M keys, **overflow!**
 - Split the node into two nodes:
 - original with $\lfloor M/2 \rfloor$ keys
 - new one with $\lceil M/2 \rceil$ keys
 - Add the new child to the parent
 - If the parent ends up with M keys, **overflow!**

4. Split an overflowed root in two and hang the new nodes under a new root
5. Propagate keys up tree.

Example :

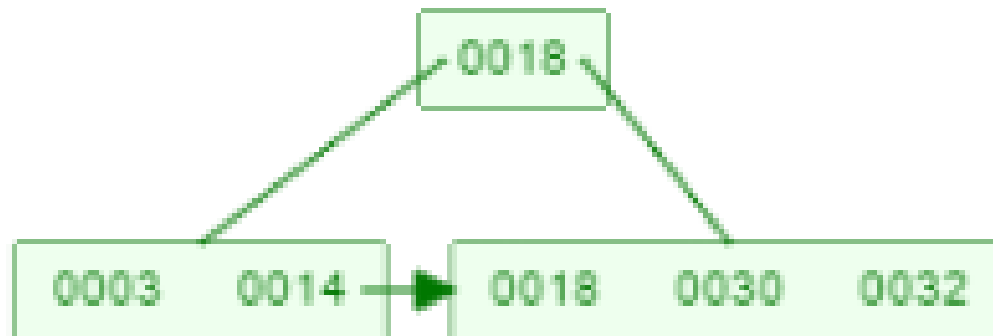
- Suppose we start with an empty B+-tree and keys arrive in the following order: 3 , 18, 14, 30, 32, 36, 15, 16, 12, 40, 45, 38
- We want to construct a **B+-tree of order 5**
- Insert 3,18, 14, 30



0003 0014 0018 0030

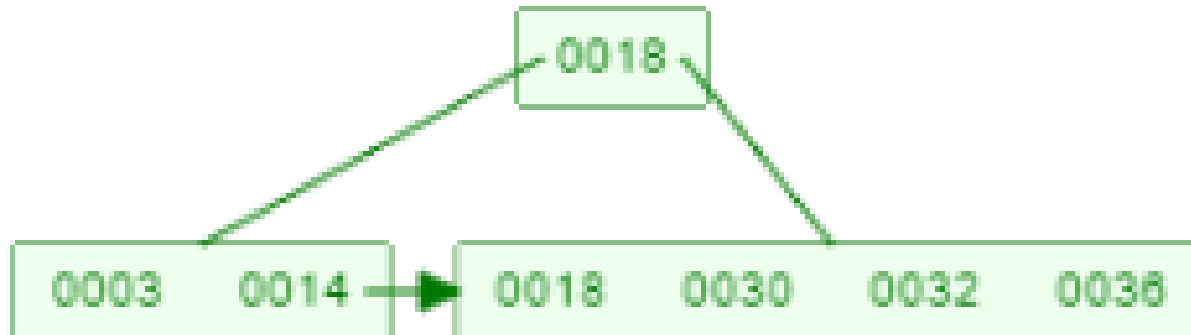
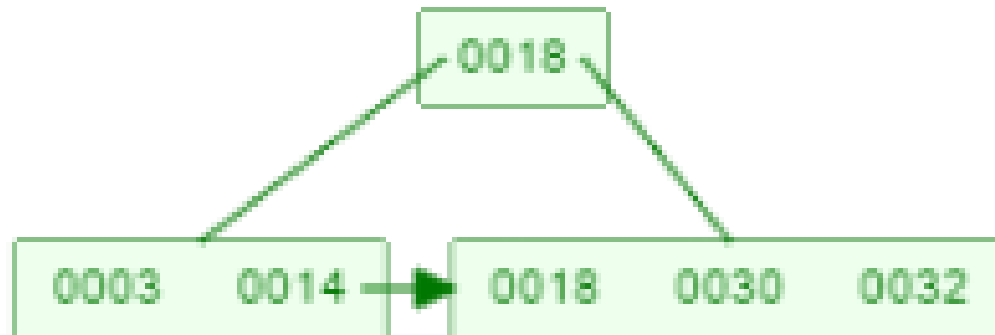
Example :

- Insert 32



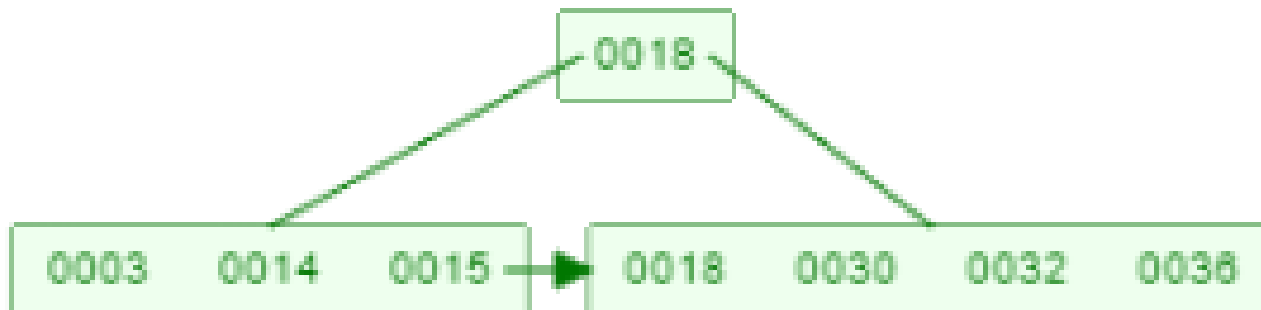
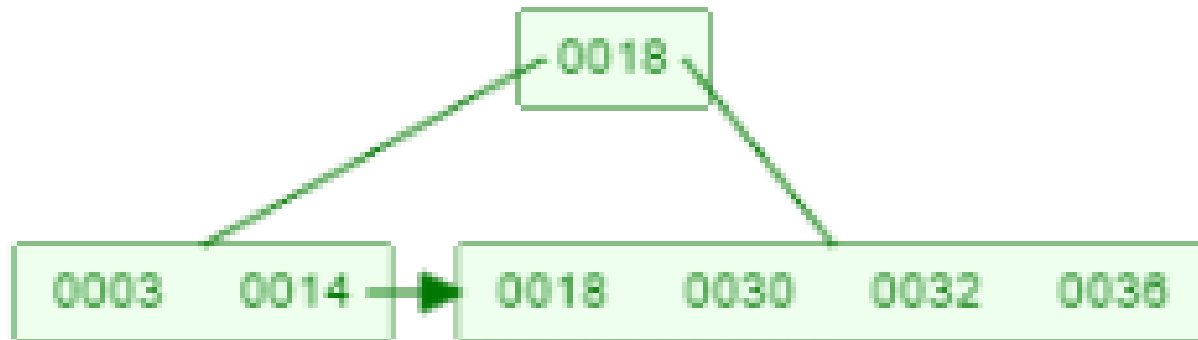
Example :

- Insert 36



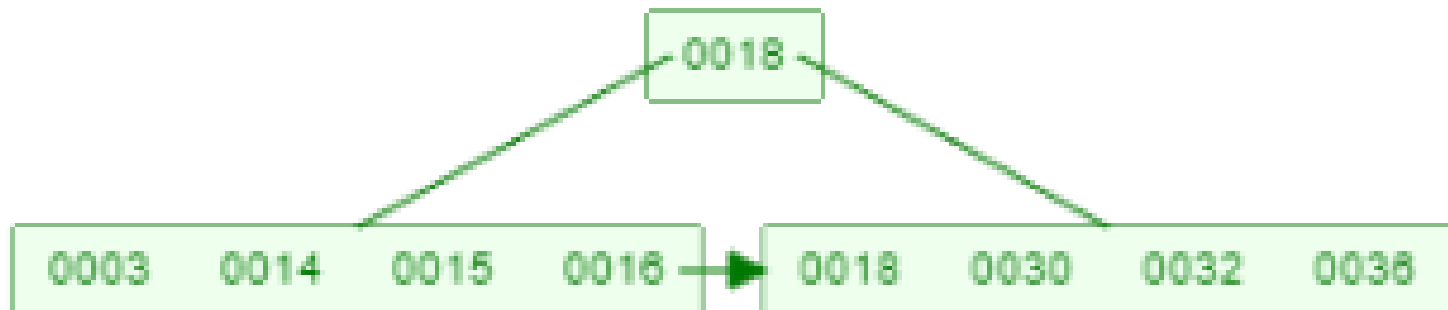
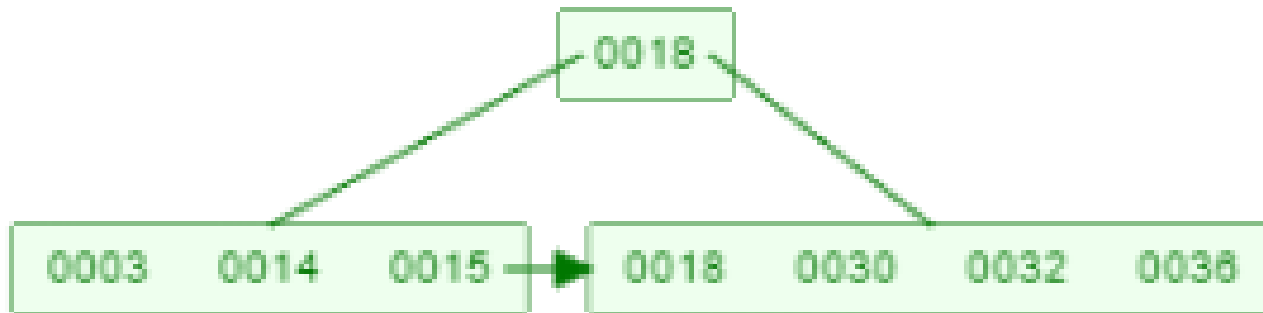
Example :

- Insert 15



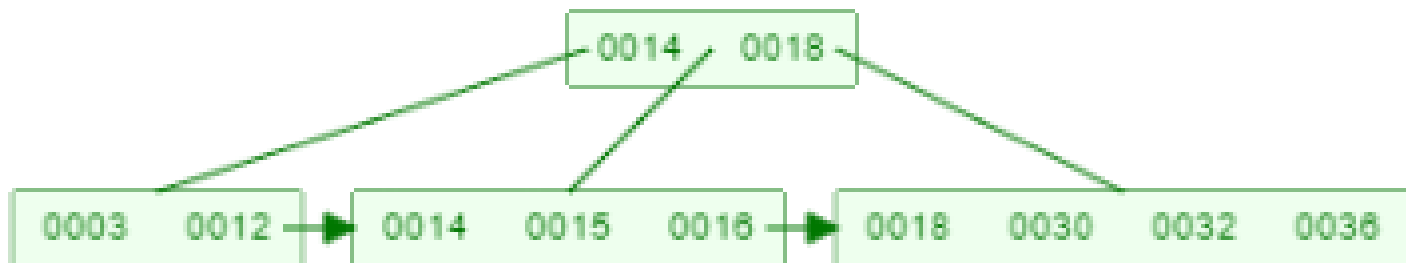
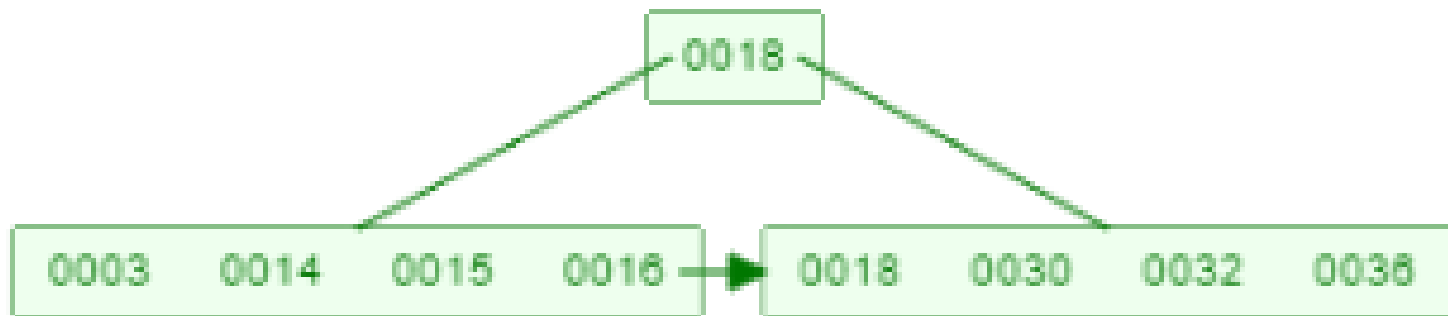
Example :

- Insert 16



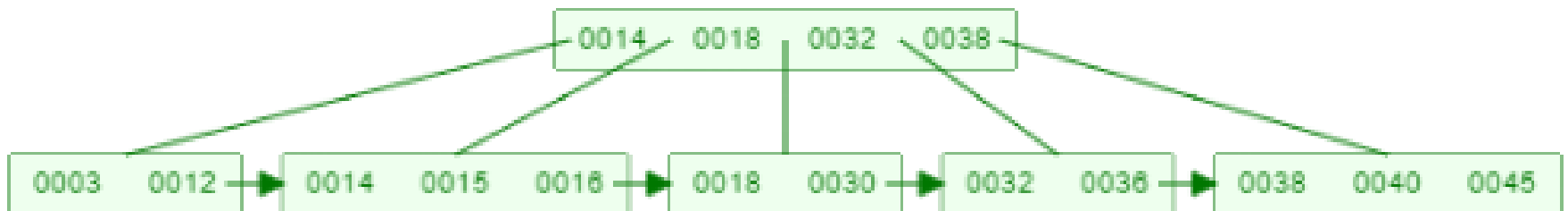
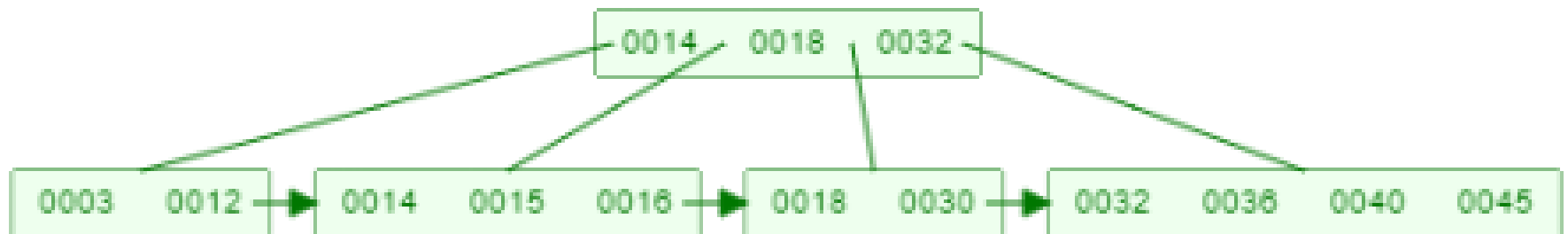
Example :

- Insert 12



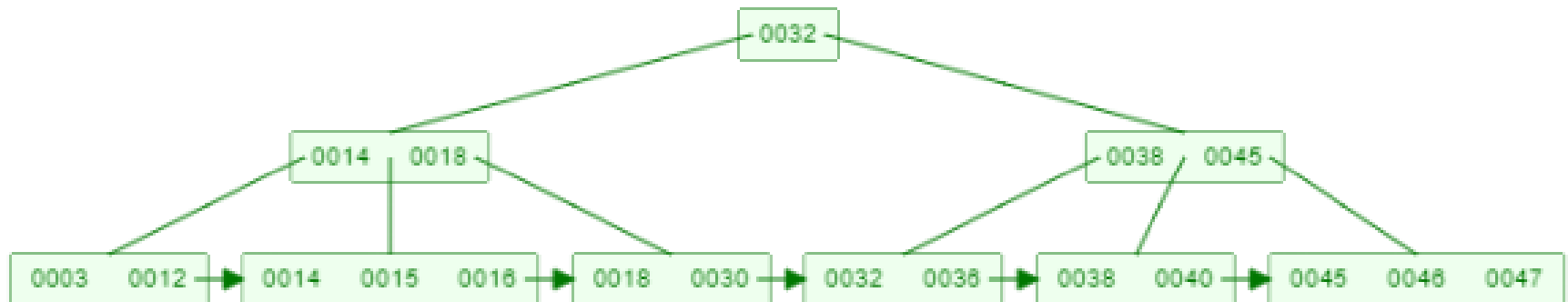
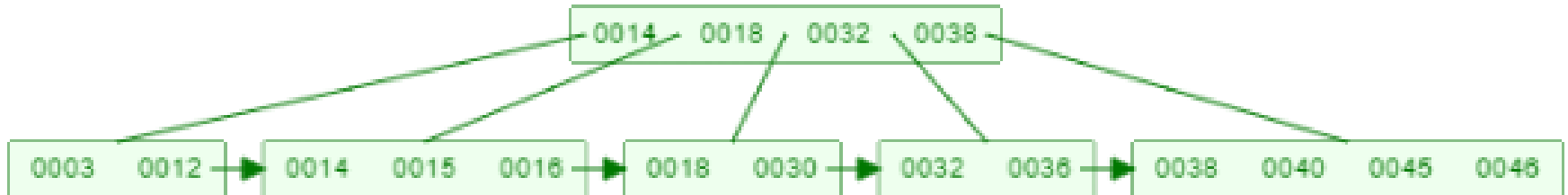
Example :

- Insert 38



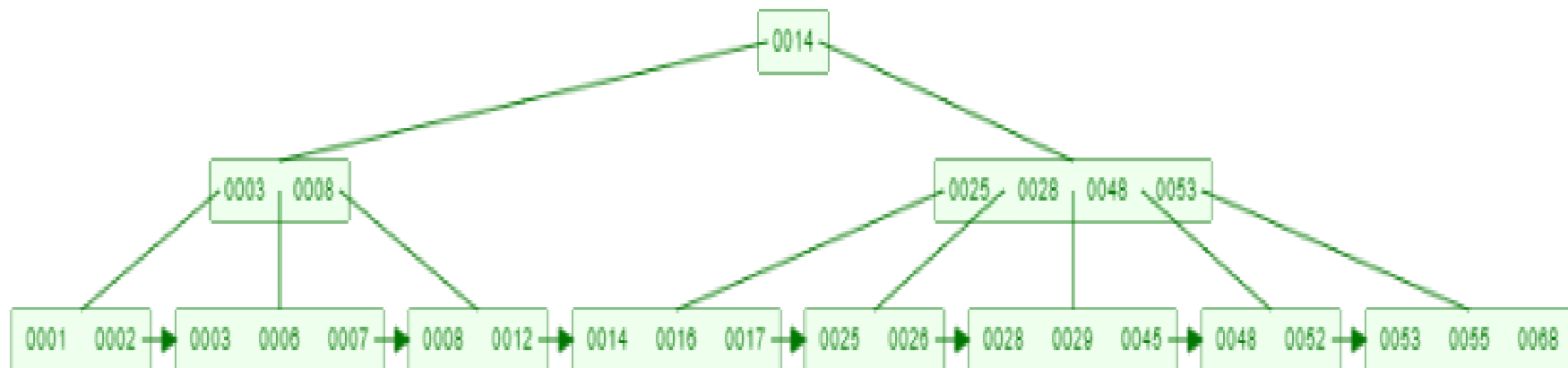
Example :

- Insert 47



Task

- 1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Advantages of B+ trees

- B+ trees don't have data associated with interior nodes, hence more keys can fit on a page of memory. Therefore, it will require fewer cache misses in order to access data that is on a leaf node.
- (*Range or sequential search*) The leaf nodes of B+ trees are linked, so doing a full scan of all objects in a tree requires just one linear pass through all the leaf nodes. A B-tree, on the other hand, would require a traversal of every level in the tree. This *full-tree traversal* will likely involve more cache misses than the linear traversal of B+ leaves.

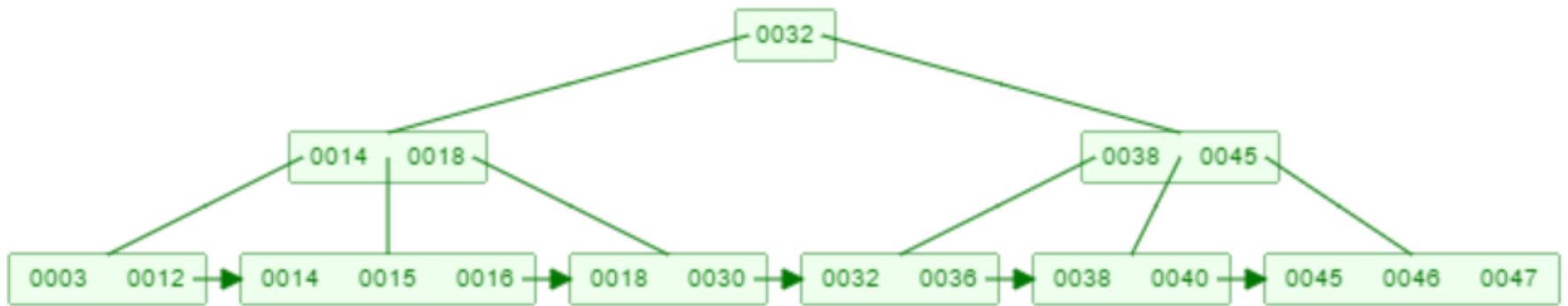
Deletion Pseudo code

1. Search the key 'k' in the tree. Mark the node in the path from root to leaf containing 'k'. Remove the 'k' from leaf.
2. If the leaf ends up with fewer than $(M/2 - 1)$ keys, underflow
 - a) Adopt data from a neighbor; update the parent
 - b) If adopting won't work, delete node and merge with neighbor, update the parent (delete entry pointing to Leaf)
 - c) If the parent ends up with fewer than $(M/2 - 1)$ keys, underflow!
3. If an internal node ends up with fewer $(M/2 - 1)$ keys, underflow!
 - a) Adopt from a neighbor through parent
 - b) If adopting won't work, delete node and merge with neighbor, update the parent (take the key from parent if its not 'k', if it is k then delete it and replace it with appropriate key)
 - c) If the parent ends up with fewer than $(M/2 - 1)$ keys, underflow!

Deletion Pseudo code

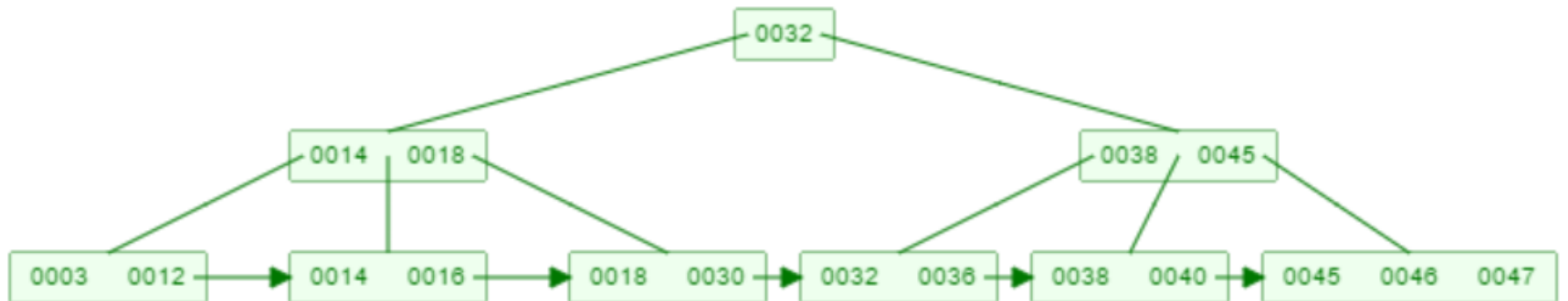
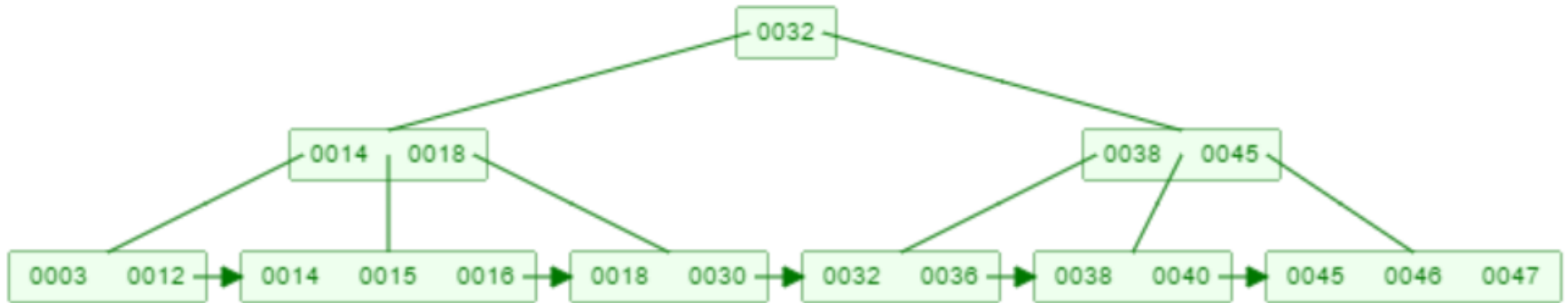
4. If the root ends up with only one child, merge child and root, make resulting node the new root of the tree
5. If procedure end up without reaching the marked node in first step then replace the 'k' in that node with appropriate key

Example



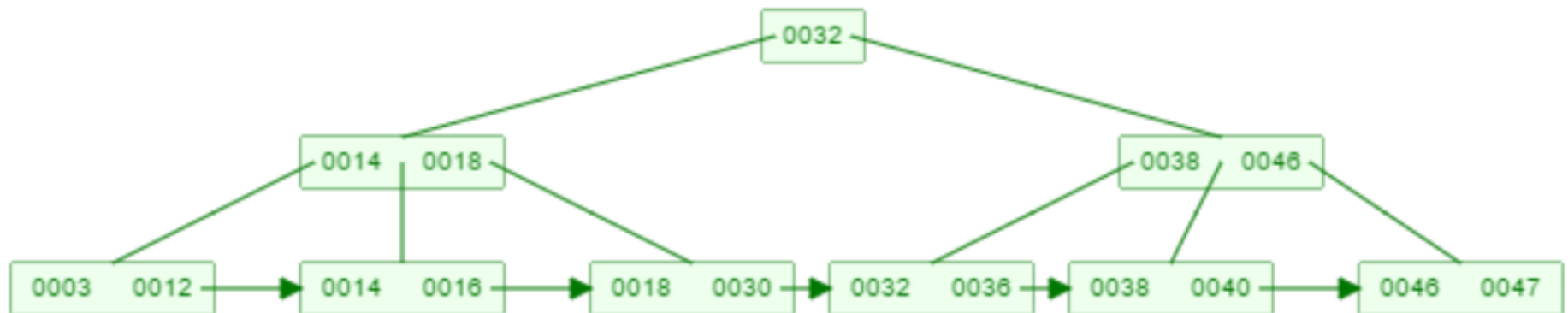
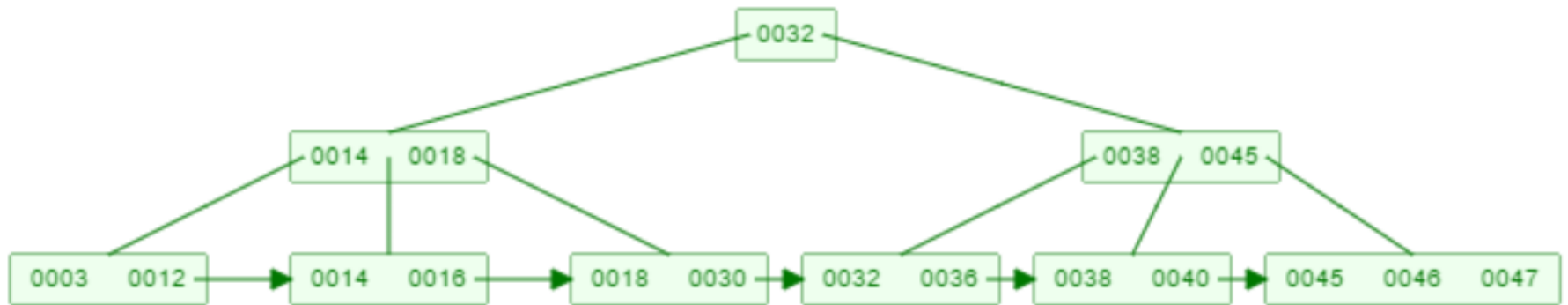
Delete 15

Delete from leaf (parent is not updated)

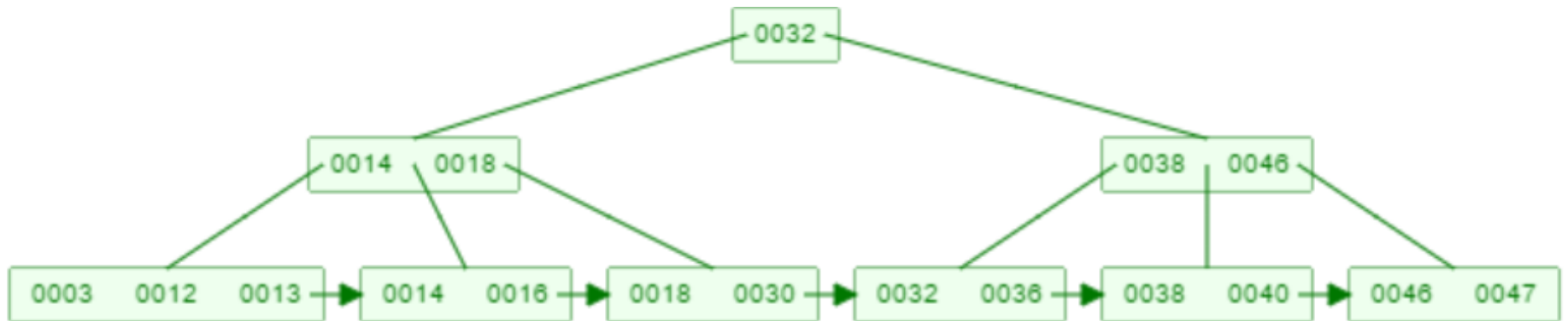


Delete 45

Delete from leaf (parent is updated)



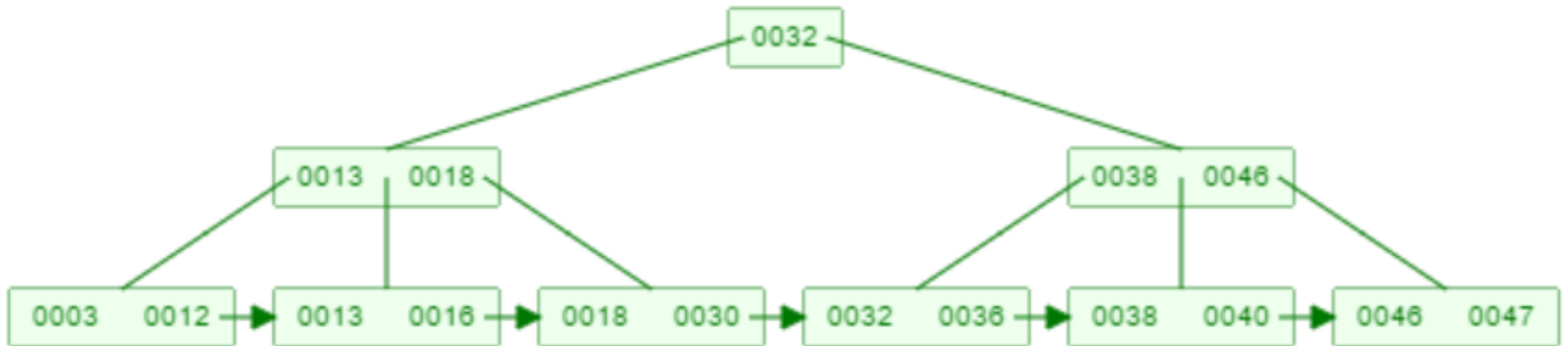
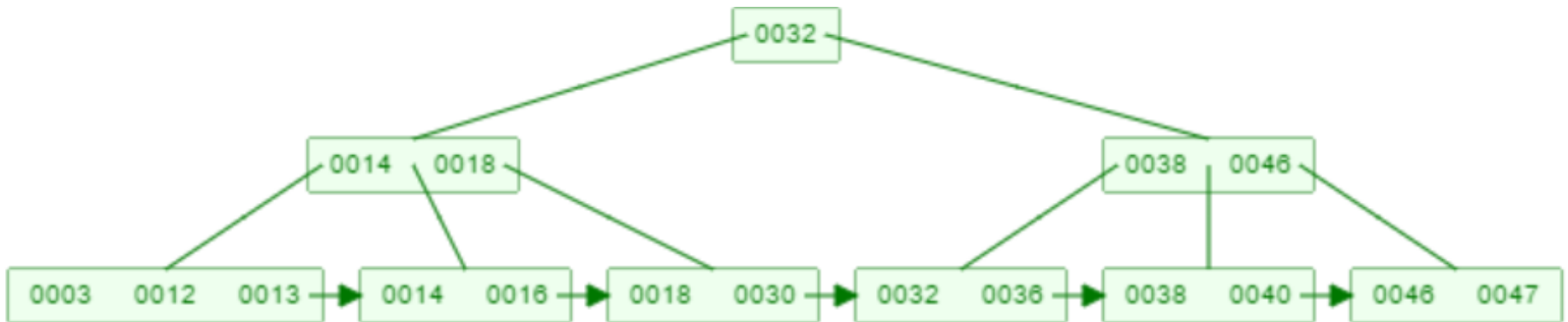
New Tree



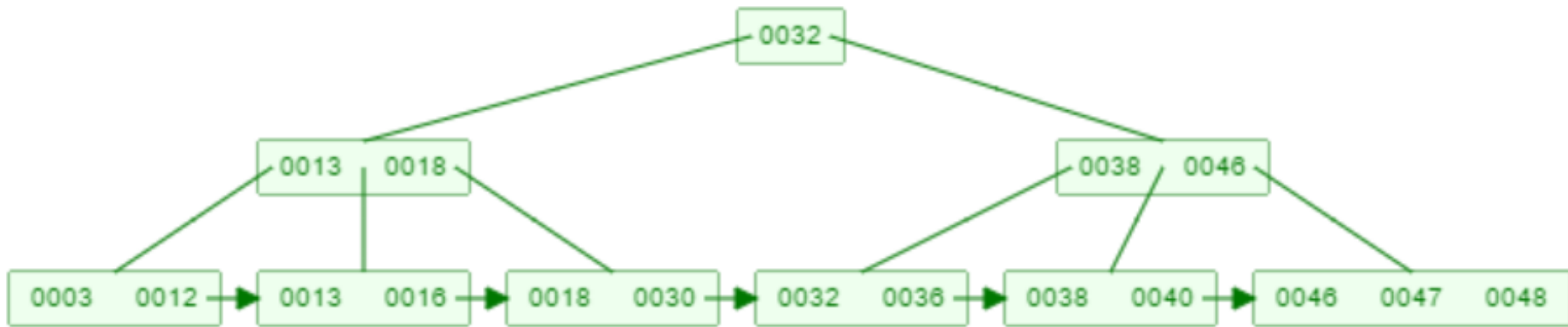
Delete 14

After Deletion

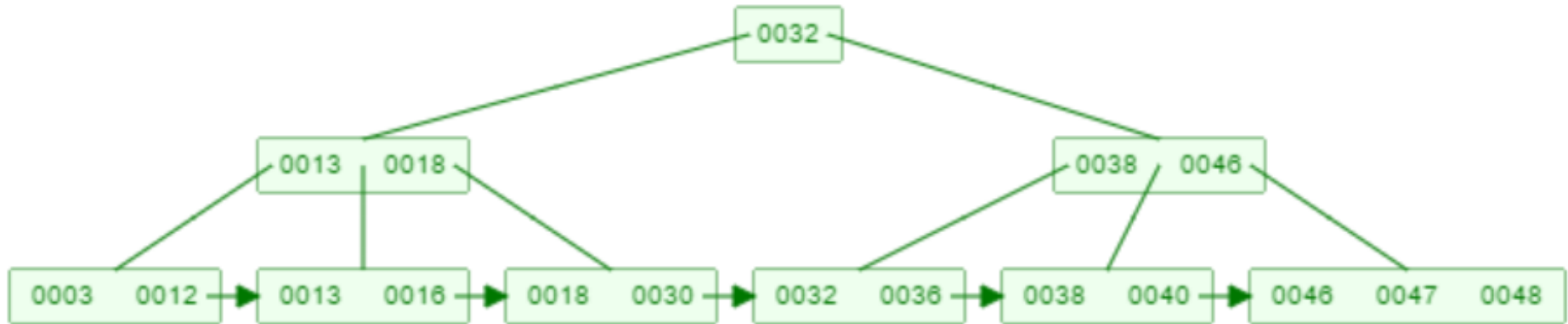
- Leaf Underflow
- Borrow in leaf from left neighbor



New Tree

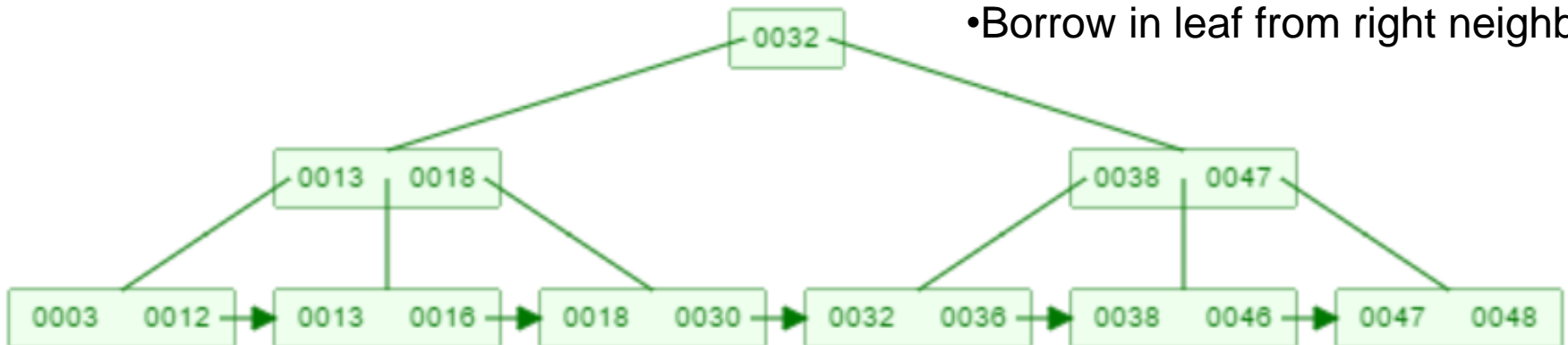


Delete 40 ????

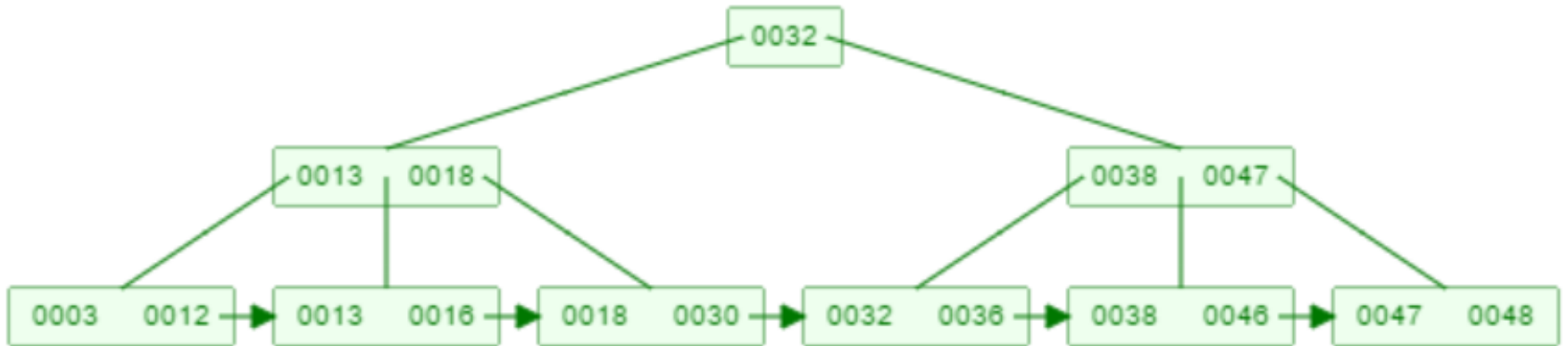


After Deletion

- Leaf Underflow
- Borrow in leaf from right neighbor

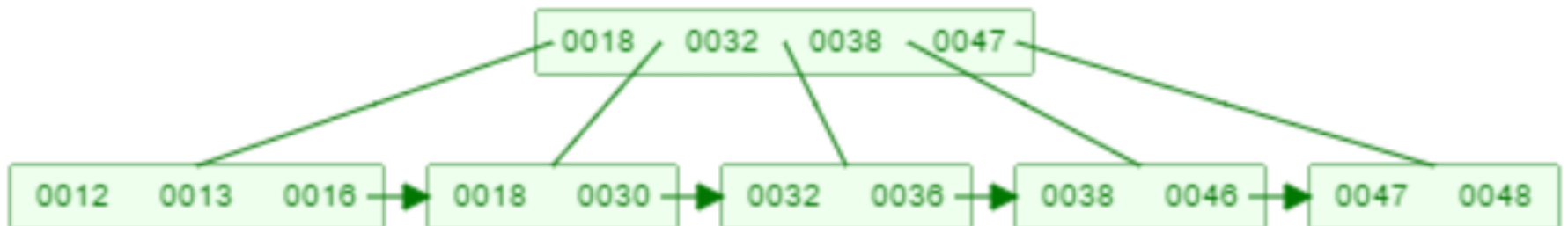


Delete 3

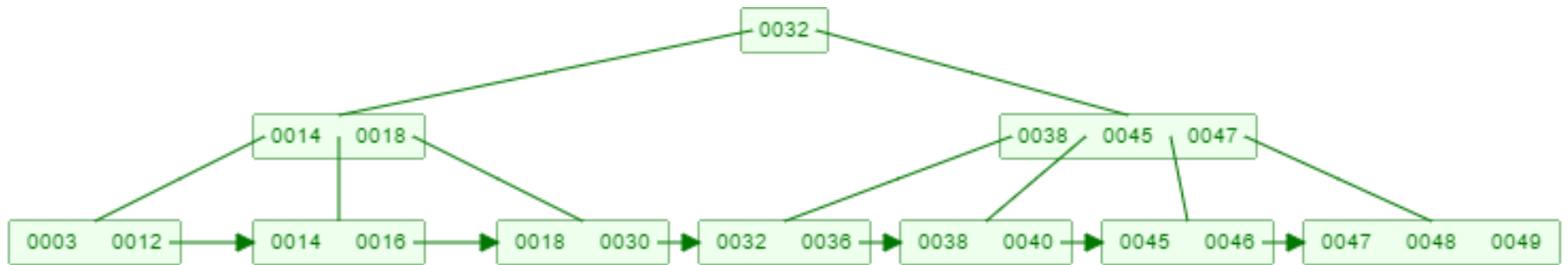


After Deletion

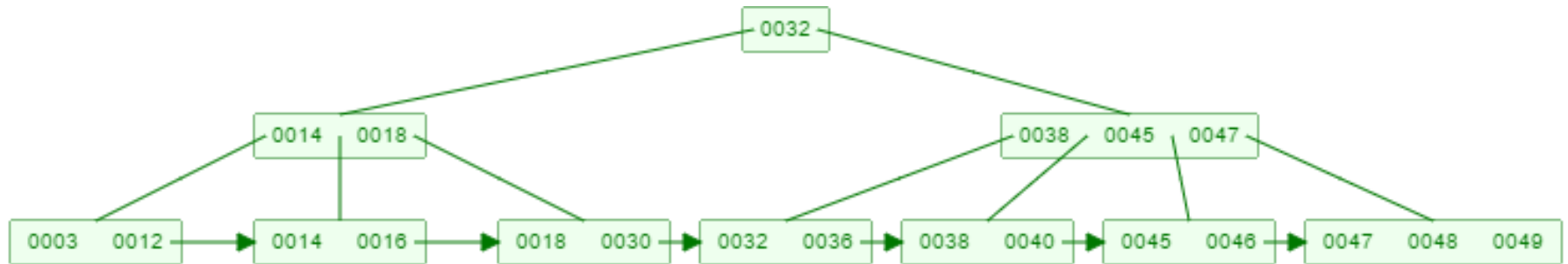
- Leaf underflow
- Parent (Internal node) underflow
-> merge



New Tree

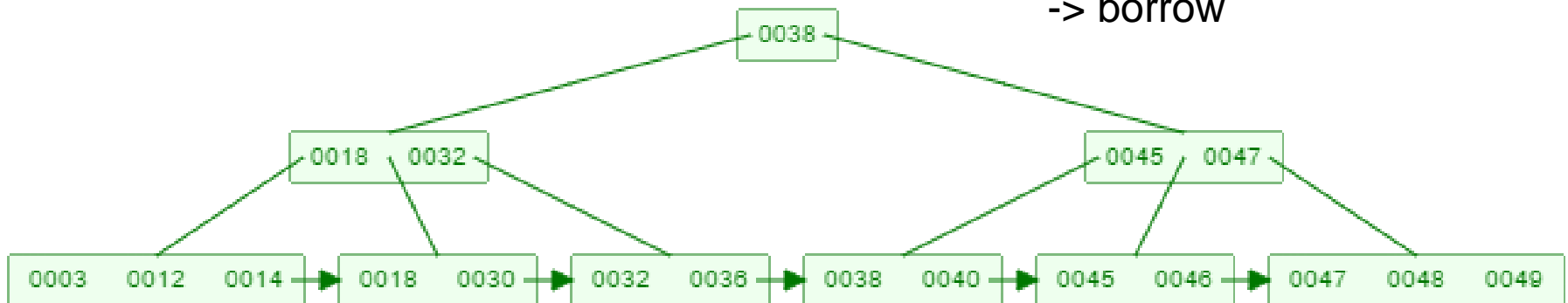


Delete 16

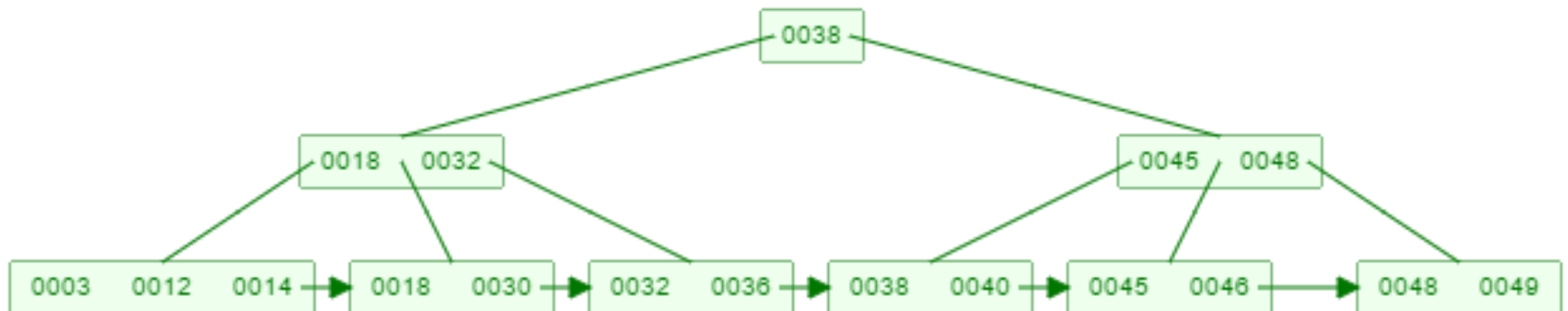
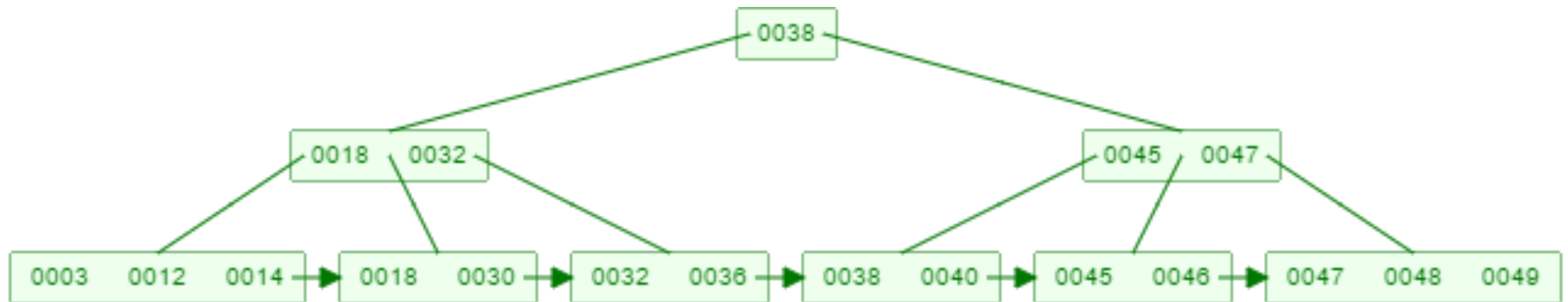


After Deletion

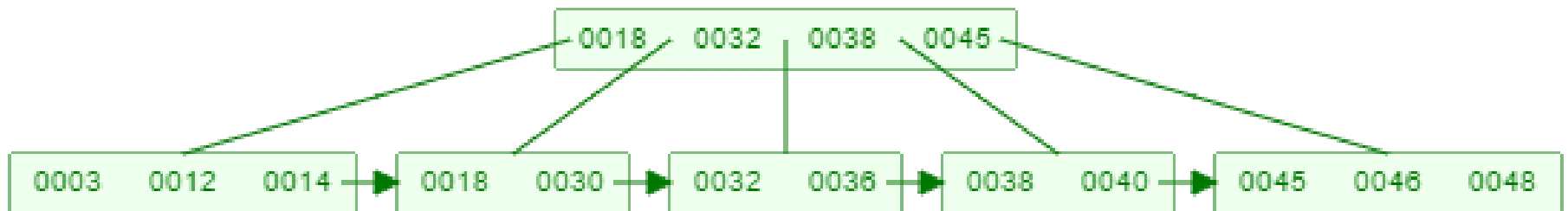
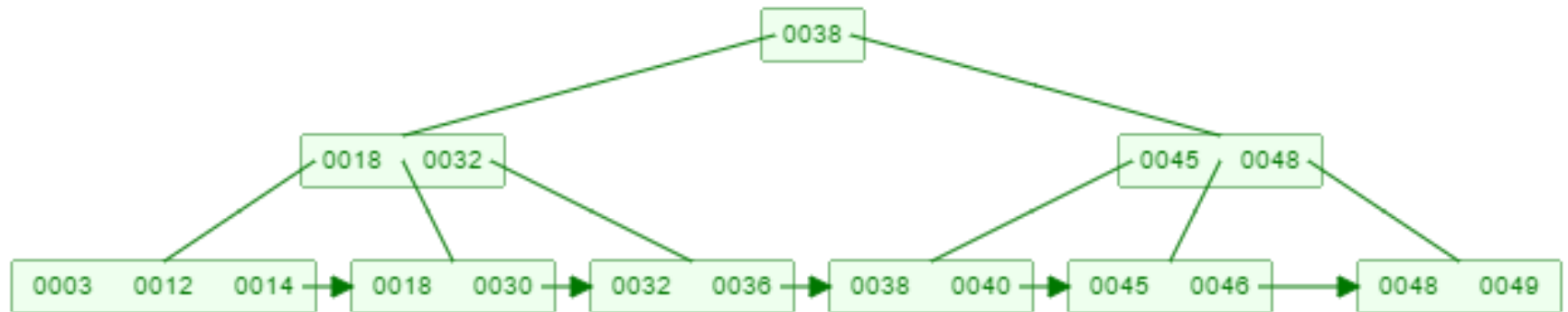
- Leaf underflow
- internal node underflow
-> borrow



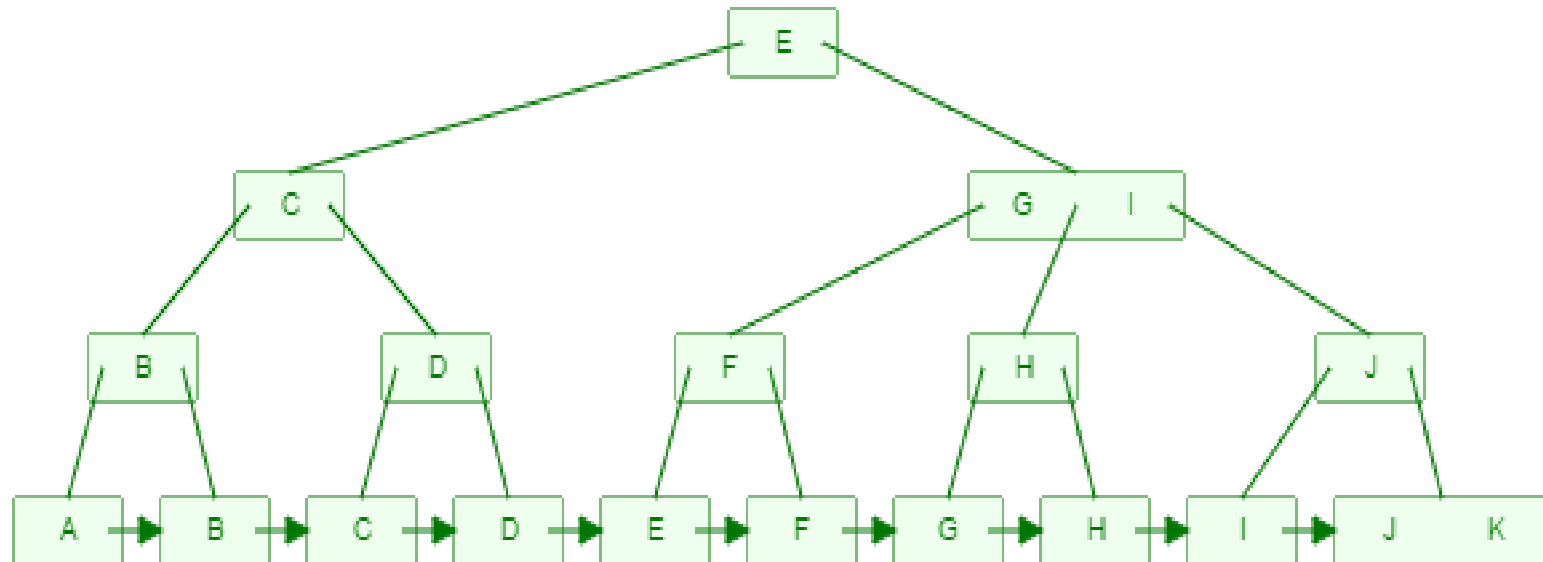
Delete 47



Delete 49

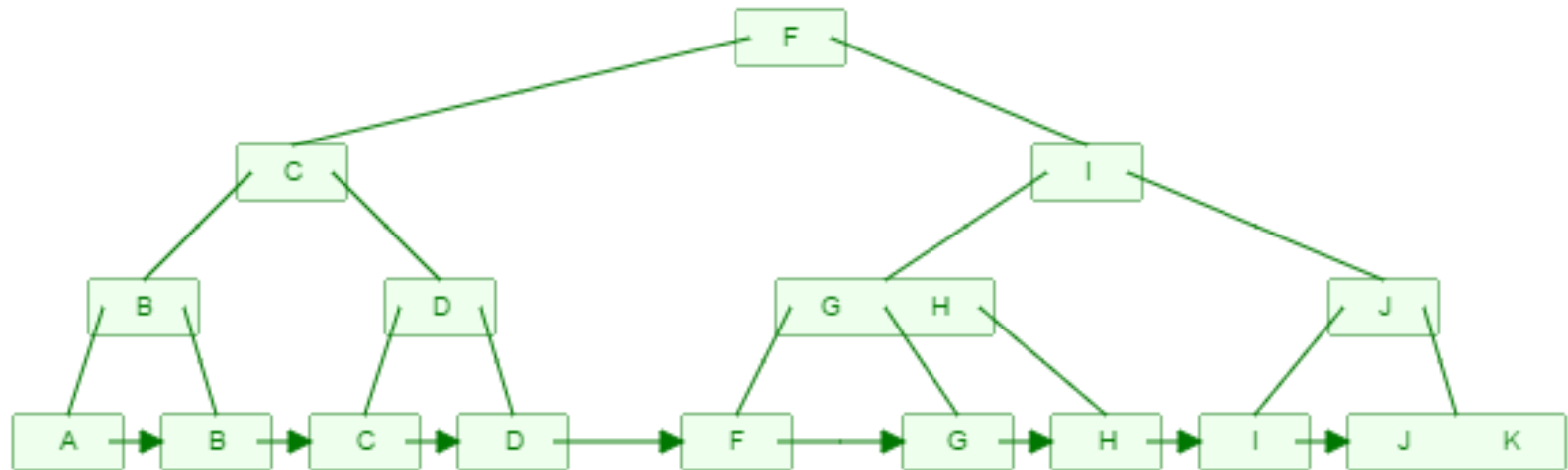


Task



- Delete E

Task



- Delete I
- Delete C

Analysis

- Search $O(\log_m n)$
- Insert $O(\log_m n)$
- Delete $O(\log_m n)$
- Range $O(\log_m n + k)$

Task

- Write a pseudocode to search data in B+tree?

```
BplusSearch (x,k)
{
    i=1;
    while (i ≤ x.n && k ≥ x.keyi)
        if(x.leaf != True & x != x.keyi)
        {
            i = i+1
        }
    If(i ≤ x.n && x.leaf && k==x.keyi)
        return (x,i)
    else if x.leaf
        return null
    else
        BplusSearch(x.ci, k)
}
```

Summary

- Analysis : similar to B tree except disk accesses.
- The **leaf nodes of B+ trees are linked**, so doing a full scan of all objects in a tree requires just one linear pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree.
- B+ trees don't have data associated with interior nodes, **more keys can fit on a main memory**.
- ***Range or sequential search:*** Providing support for range queries in an efficient manner.
- Because B trees contain data with each key, nodes closer to the root, can be accessed more quickly.

Summary

- B+ tree is used for an obvious reason and that is search speed. As we know that there are space limitations when it comes to memory, and not all of the data can reside in memory, and hence majority of the data has to be saved on disk. Disk as we know is a lot slower as compared to memory because it has moving parts.