UCSC Silicon Valley Extension

Advanced C Programming

B-Trees

Instructor: Radhika Grover

Latency numbers every programmer should know
https://gist.github.com/jboner/2841832

Overview

- B-trees
- Memory vs disk storage
- Storing BST on disk
- B-tree examples and exercise
- B+ tree

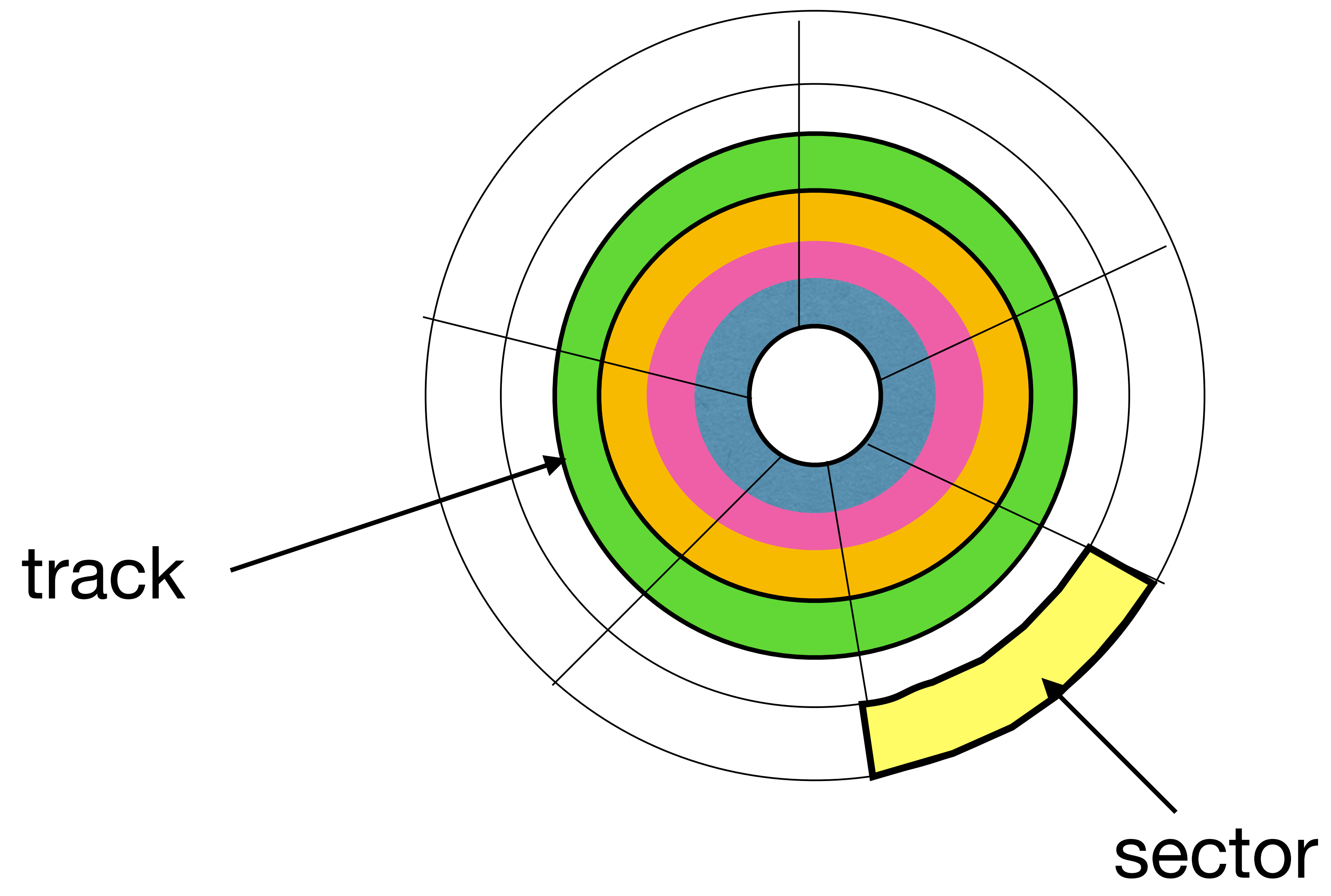
Memory vs disk based

- Binary trees are not used for disk-based storage because they are not efficient for retrieving data from disks.
- Disk access (takes milliseconds) is much slower main memory (takes nanoseconds).
- But RAM is volatile and cannot be used for persistent storage.
- B-trees, B+ trees and other data structures are designed to retrieve data from disks instead of main memory.
- B-trees are used in database systems for disk based storage.

Disk storage

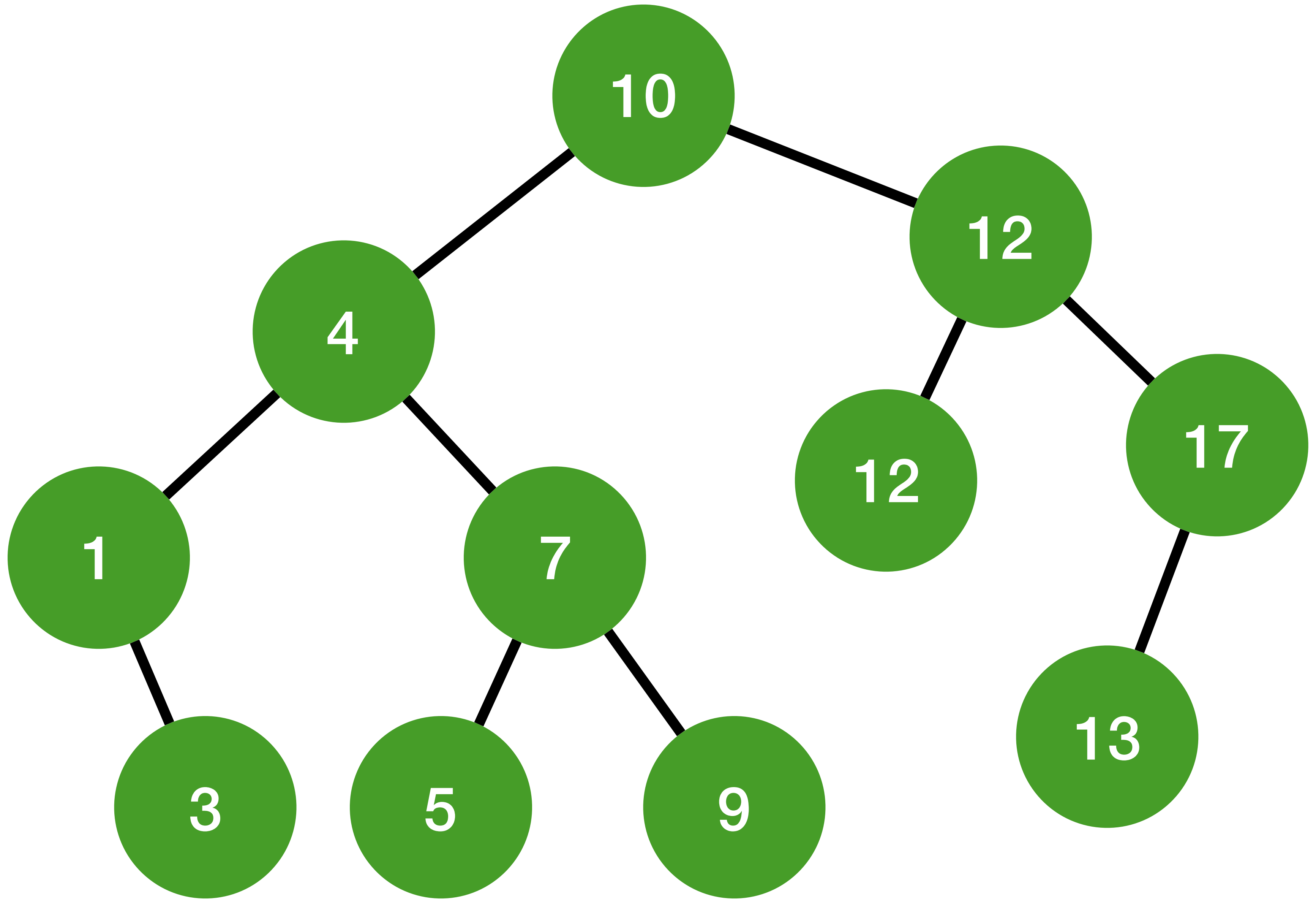
- Sector size is typically 512 bytes or 4096 bytes.
- Programs read and write data in terms of logical *block* size; by default, logical block size is equal to page size.
- A logical block can be one or more contiguous sectors long and its size is determined by the file system.
- Disc access time = seek time (to find track) + rotational time (to find sector) + transfer time (to transfer a block)

Disk platter



Storing BST on disk

- Example: Suppose we store the BST on the right on the disk so that each node is in one disk sector.
- Each node contains data (a record) and two pointers to two nodes.
- Assume that 1 sector is 512 bytes, 1 record takes 20 bytes and a pointer takes 6 bytes.
- What are the problems?



Note: Tree height is the number of edges on longest path from root node to a leaf

Storing BST on disk continued

- Inefficient use of space on disk - each sector uses 20 bytes (integer) + 12 bytes (2 pointers) = 32 bytes and the remainder 256-32 = 224 bytes is wasted.
  - *Need to pack the data more efficiently*
- Time to retrieve data items varies and the number of disk accesses is equal to the height of the tree in the worst case.
  - *Need to limit the height of the tree*

An improved data structure for disk storage

1. *Pack the data more efficiently*

   - Data is stored in leaf nodes.

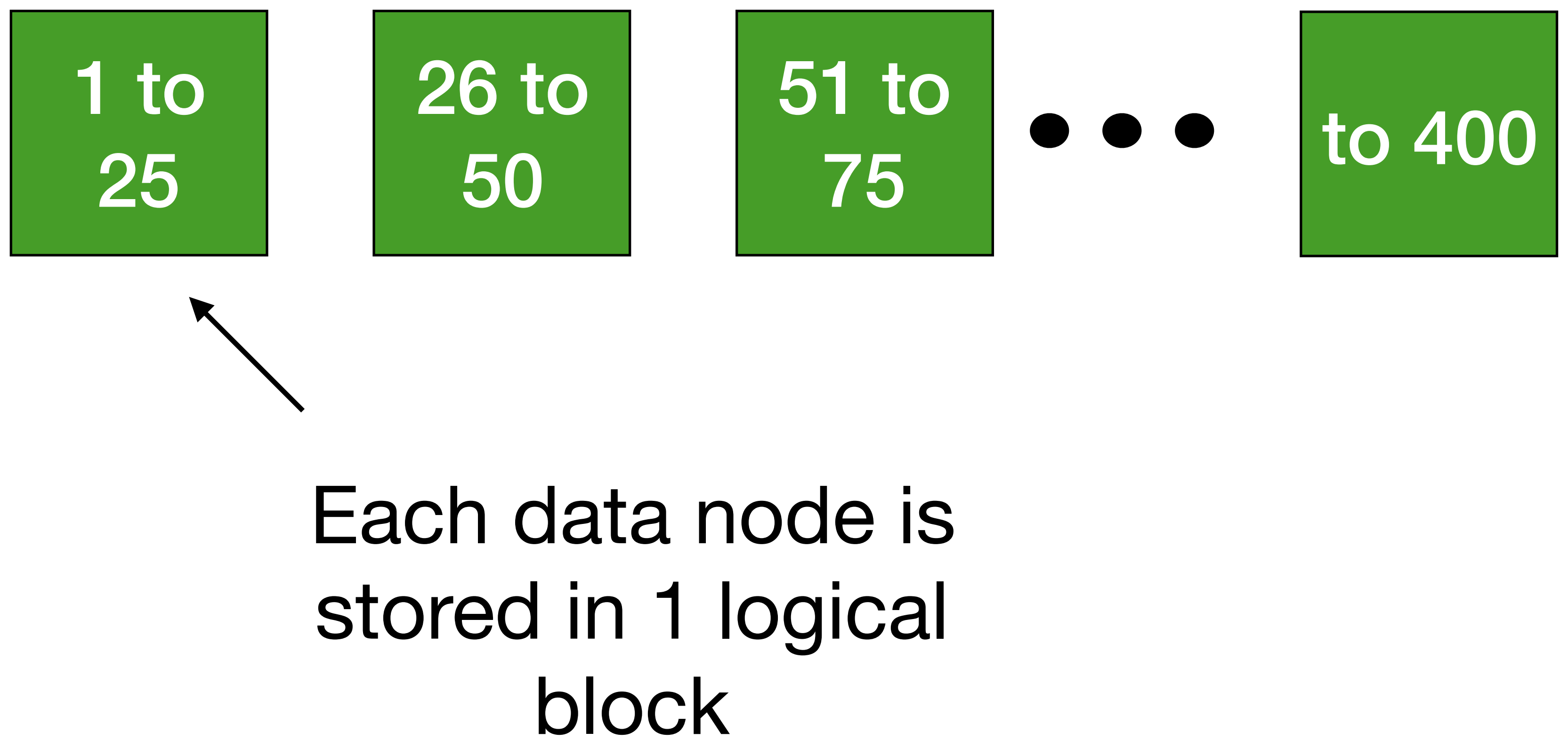
   - The maximum number of elements in a *data* node fits in one disk block.

2. *Limit the height of the tree*

   - Use an *m*-ary tree (instead of a binary tree) of *index* nodes to limit the tree height.

B-tree example 1- data nodes

- Example: Store data records with keys numbered from 1 to 400 on the disk.
- Suppose that logical block size = 4096 byes, sector size = 512 bytes, and each record is 160 bytes.
- Each logical block has 8 sectors.
- We can store 4096/160 = 25 records in each logical block, and need 16 data nodes in total.



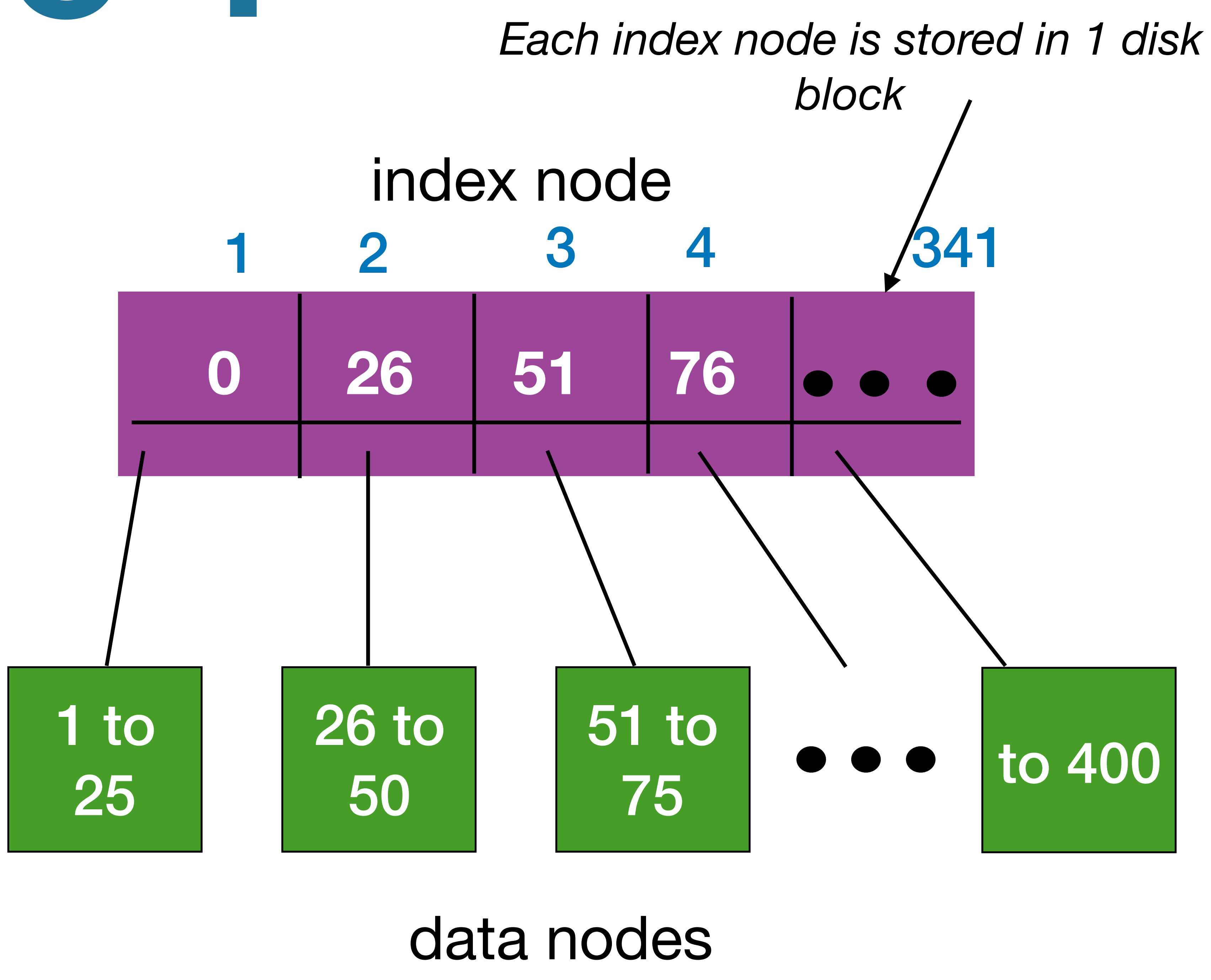
B-tree: index nodes

- Need to access the data in the leaf nodes efficiently - use index nodes.

- Each index node contains:
  - pointers to at most m child nodes p_1, p_2, …, p_m, and
  - m values representing the smallest keys in p_2, p_3, …, p_m

B-tree example 1

- Example: Store data records with keys from 1 to 400 on the disk.
- Block size = 4096, sector size = 512 bytes, and each key is 4 bytes, pointer is 8 bytes.
- An index node can fit *m* pointers and *m* values in 1 block.
- To fit *m* pointers and *m* keys solve $8*m + 4 * m = 4096$, which gives $m = 341$.
- There are 16 data nodes in this example, so only one index node is needed as it can fit up to 341 pointers to the data nodes.



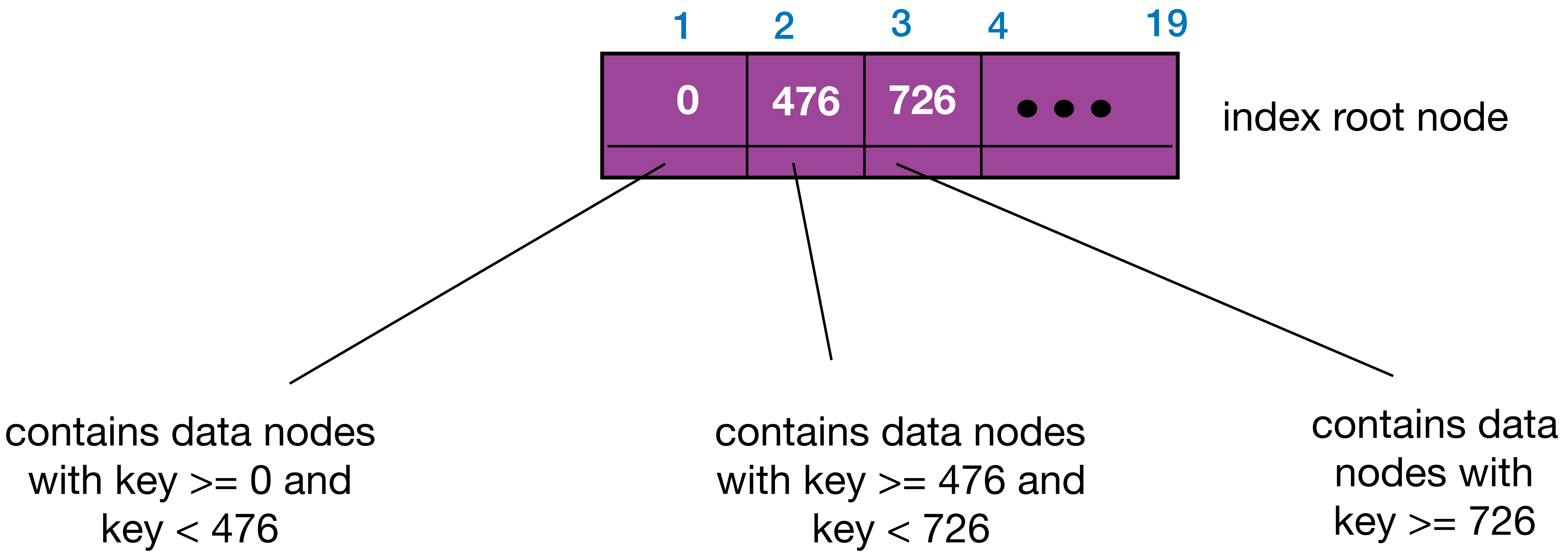
B-tree order m = 42

How many disk accesses are needed to retrieve record 51?

B-tree example 2

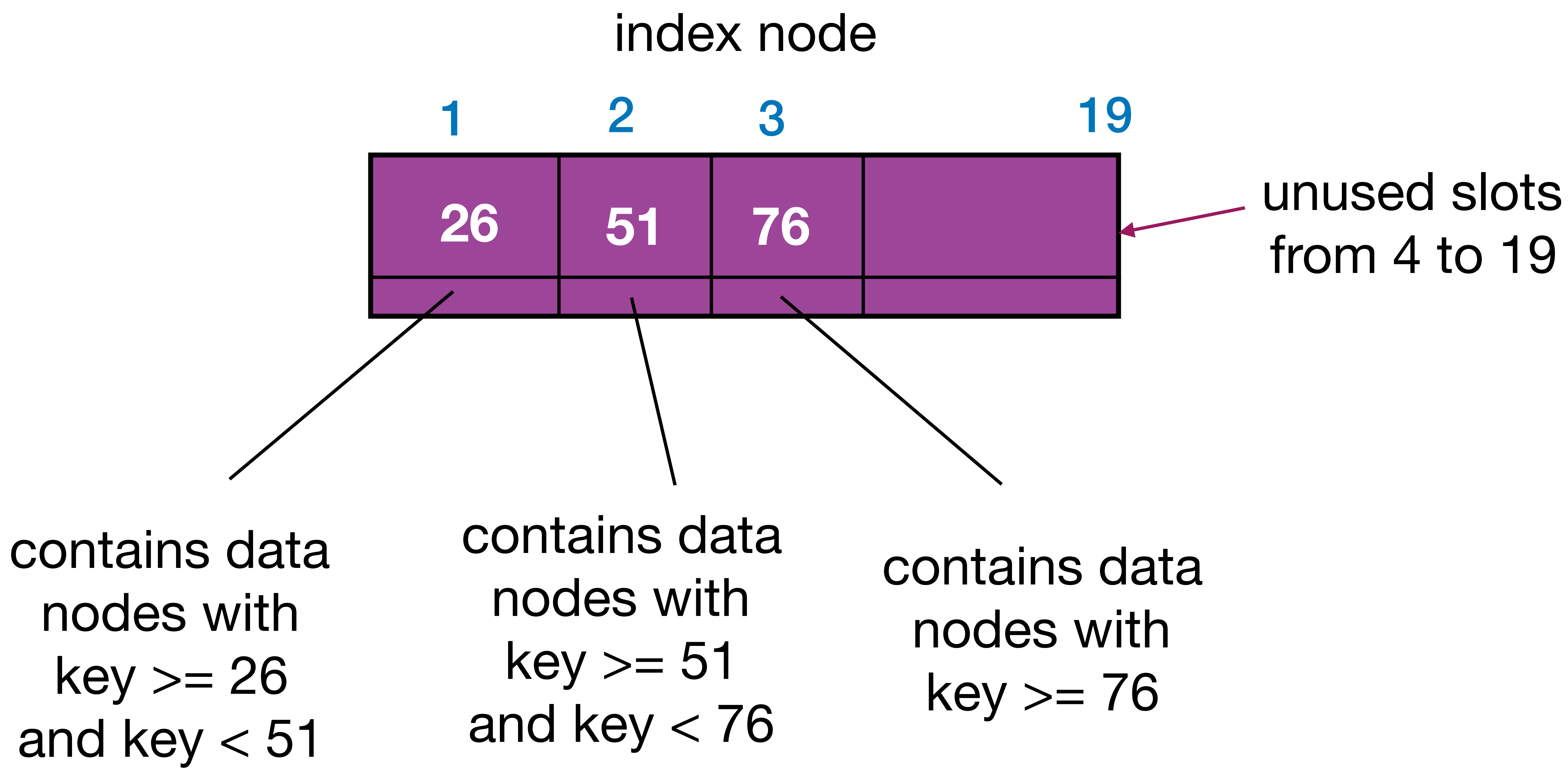
- Suppose we want to store 1000 records in a a B-tree with order $m = 19$.
- Then the B-tree has more than one index node.
- The index nodes are organized hierarchically.

B-tree example 2 continued: index root node values



index root node contains 2 to 19 pointers

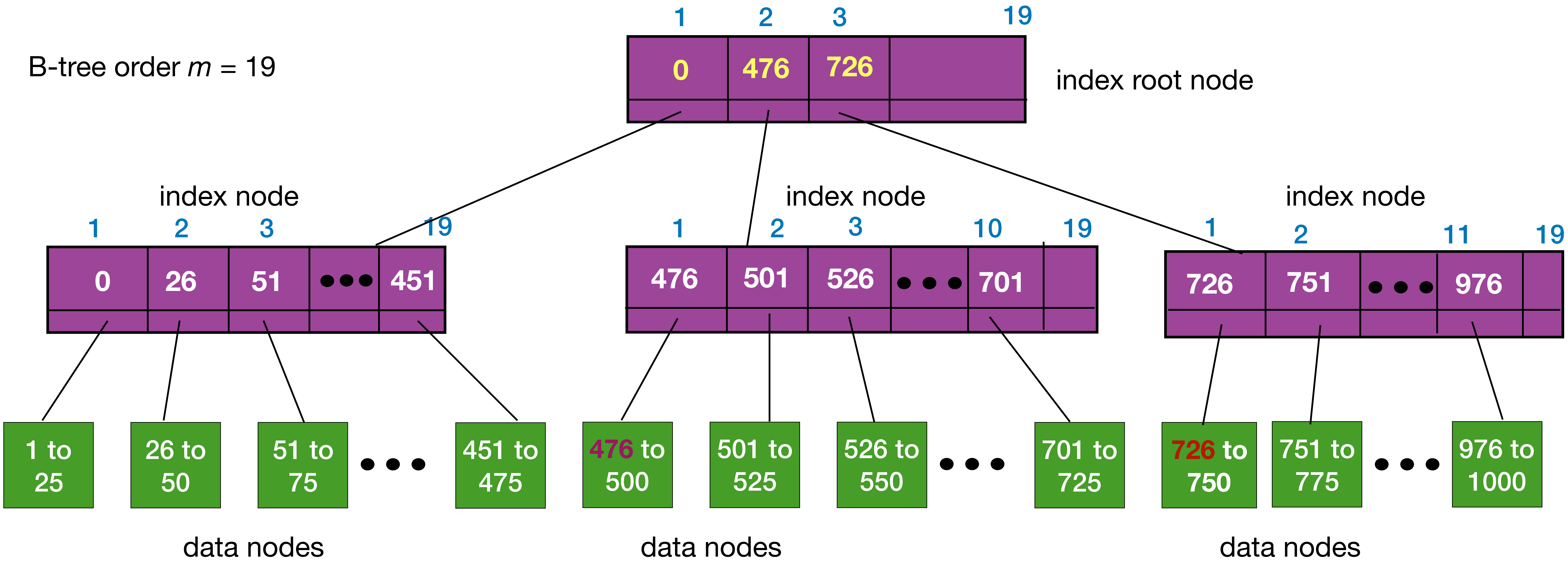
B-tree example 2 continued: index non-root node values



index non-root node contains ⌈19/2⌉ to 19 pointers

B-tree example 2 continued: hierarchical structure of index nodes

B-tree order m = 19



How many disk accesses are needed to retrieve a record?

B-tree properties

- B-tree of order m has these properties:
  - Root node is either a data node or an index node with between 2 and m pointers (index nodes or data nodes).
  - All other nodes have between $\lceil m/2 \rceil$ to m pointers.
  - All data nodes are at the same depth and contain $\lceil m/2 \rceil$ to m data records (or pointers to data records).
  - The data nodes contain sorted keys.

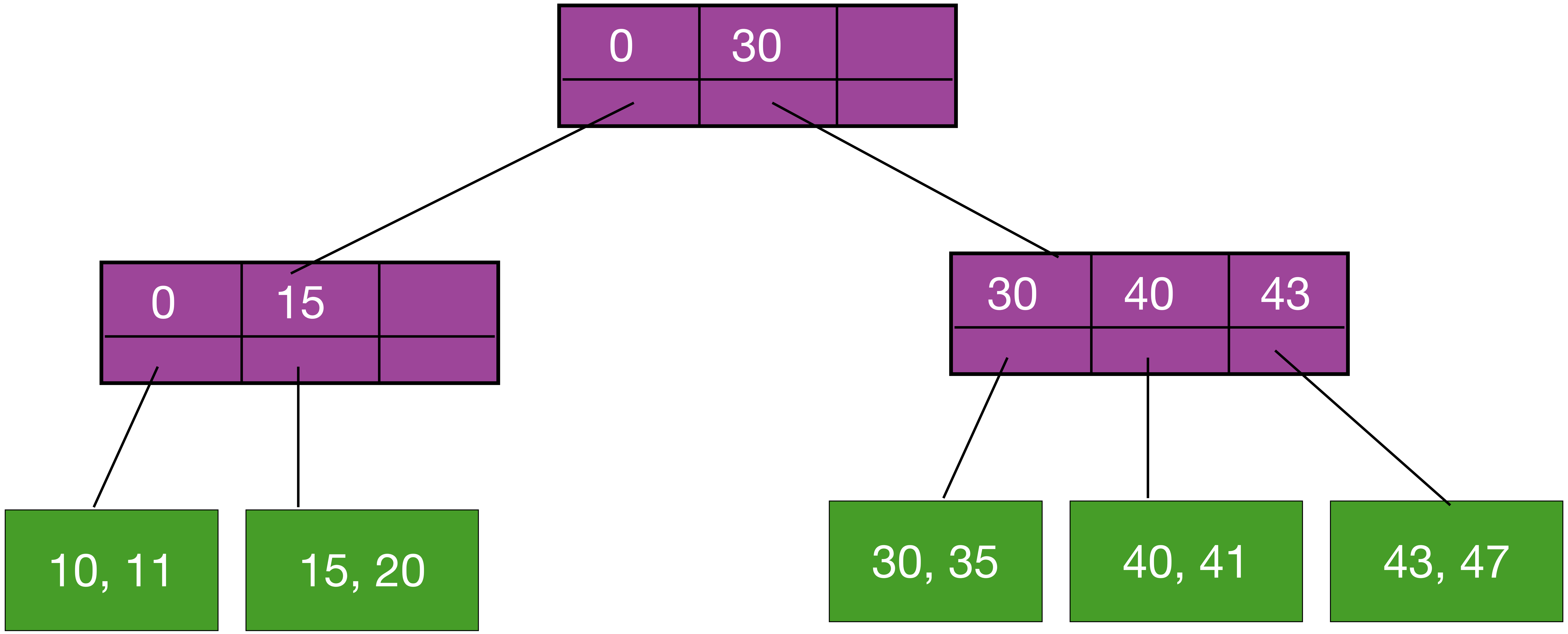
B-tree node insert operation

- Use a find operation to find the data node where the key x can be inserted.
- If data node fewer than m keys, insert x in this node.
- Otherwise, if the data node already has m keys, split it into two nodes with $\lceil (m+1)/2 \rceil$ and $\lfloor (m+1)/2 \rfloor$ keys respectively.
- This gives parent an extra node - split parent if it already has m pointers.
- Continue splitting parent nodes until we find a parent with less than m pointers. If root node is split, create a new root with 2 children.

Exercise 1

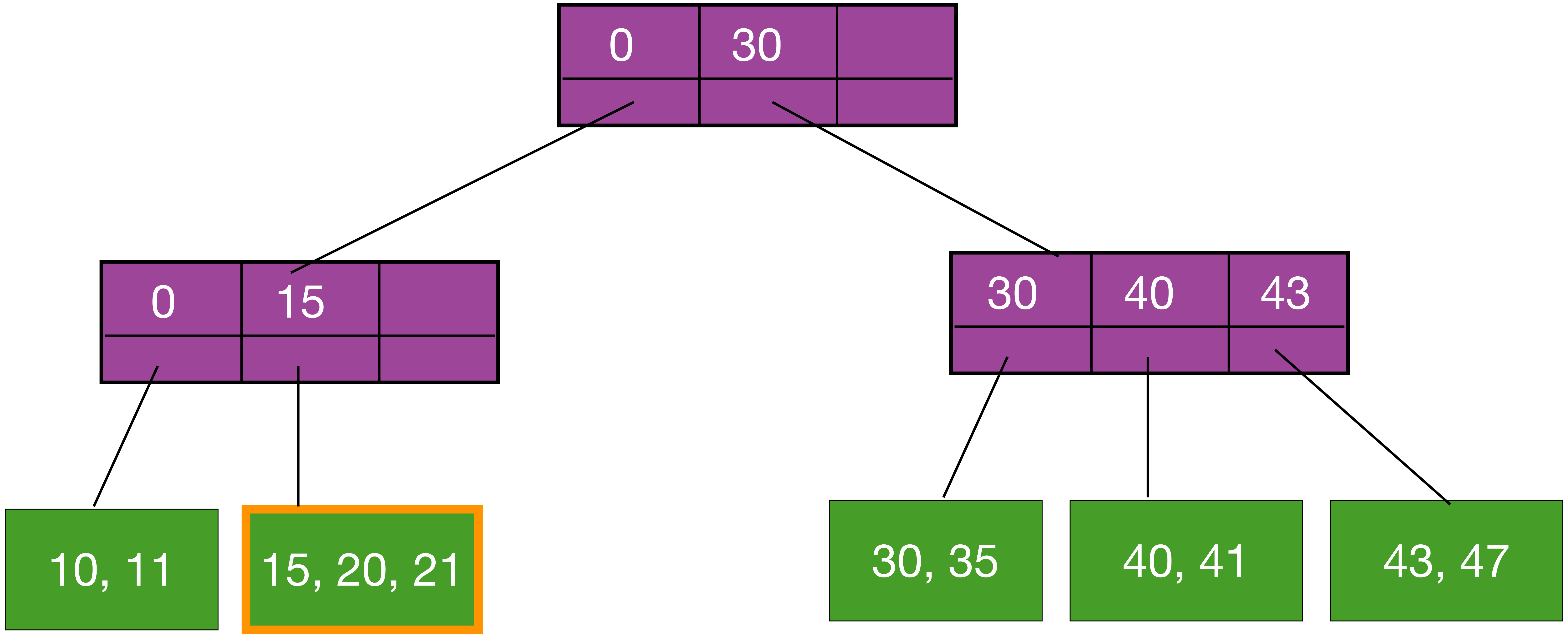
- Insert the following keys into a B-tree of **order 3**.
  - 10, 11, 15, 20, 30, 35, 40, 41, 43, 47
- After building the tree, insert these keys:
  - 21, 25, 28, 29, 38, 39

Exercise 1: One possible solution



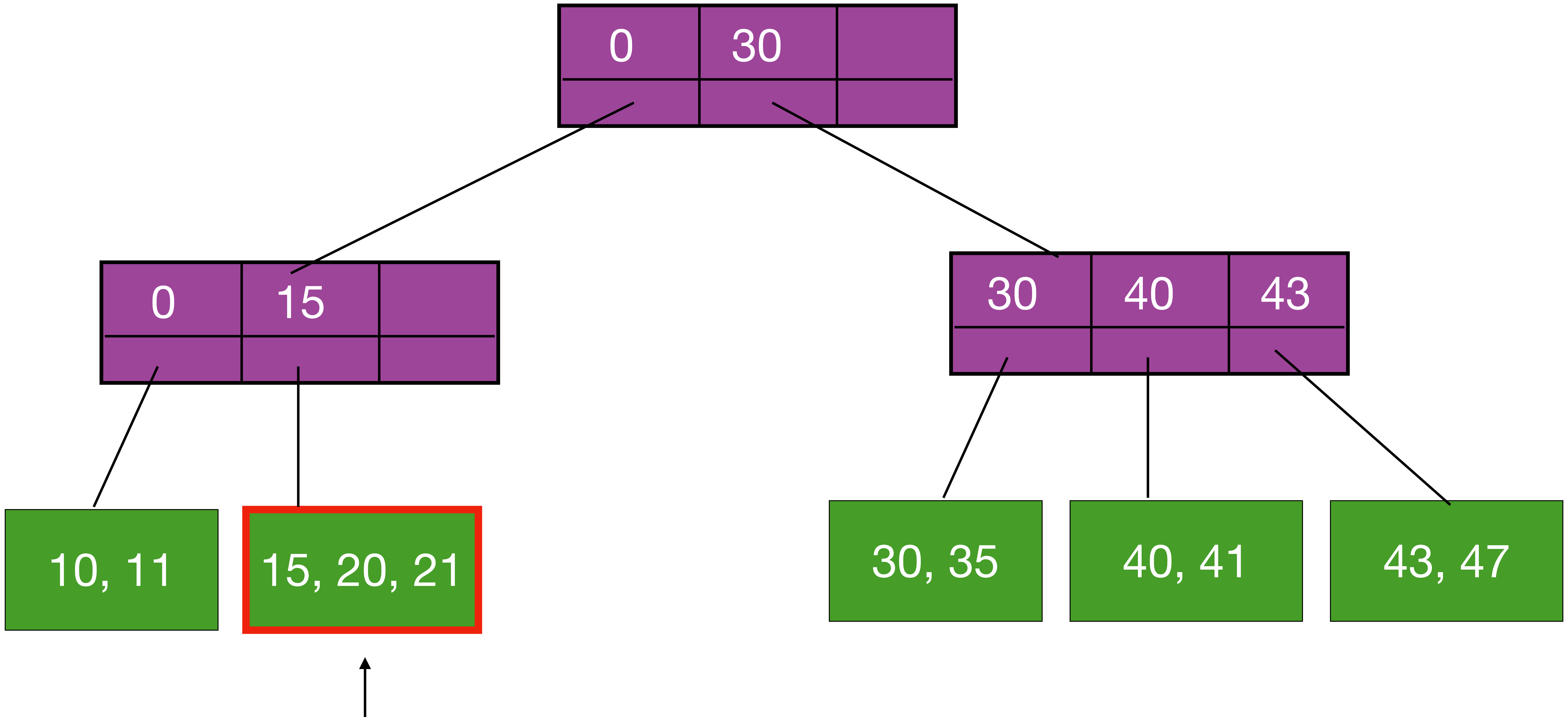
Next, insert 21

Exercise 1: One possible solution



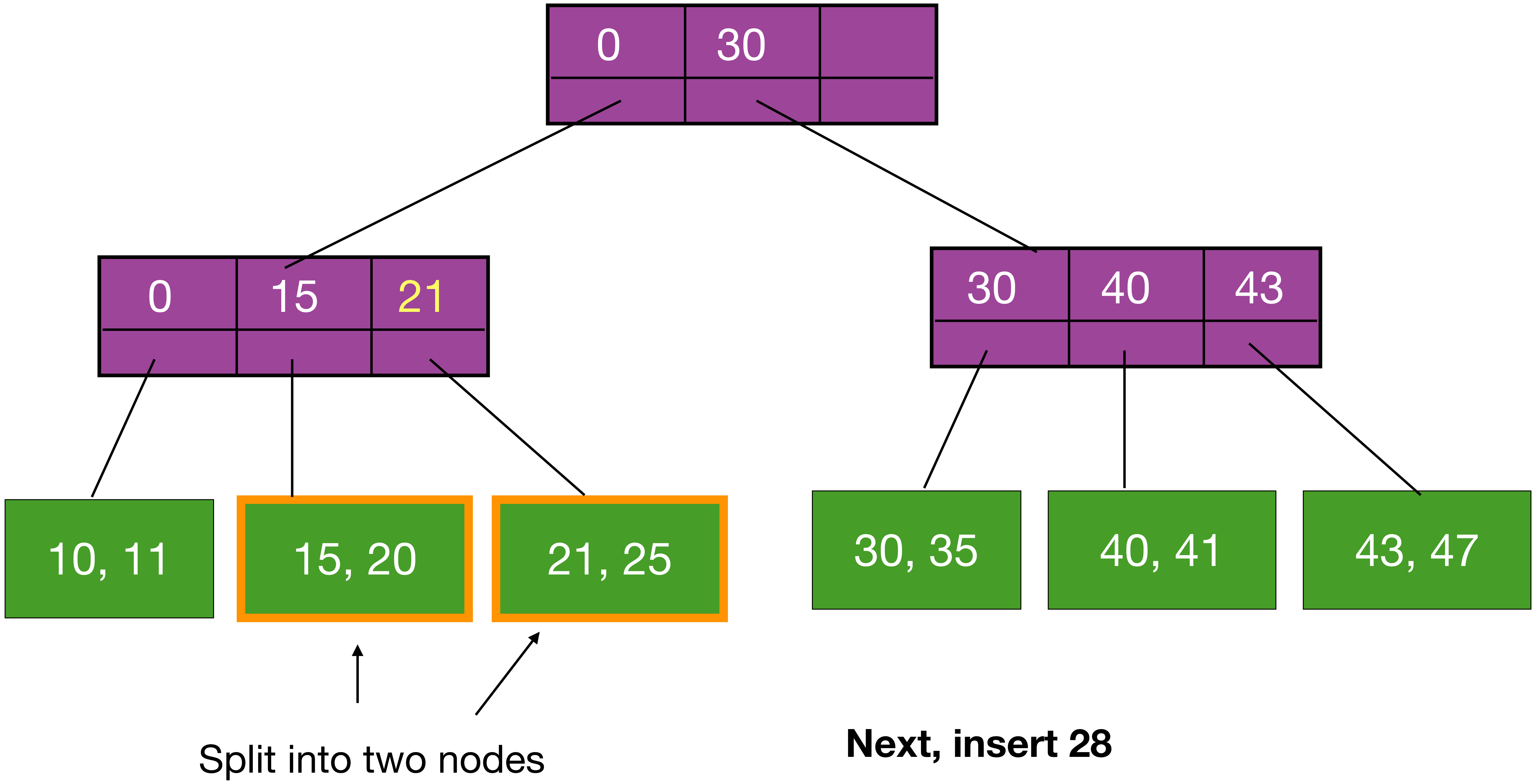
Next, insert 25

Exercise 1: One possible solution



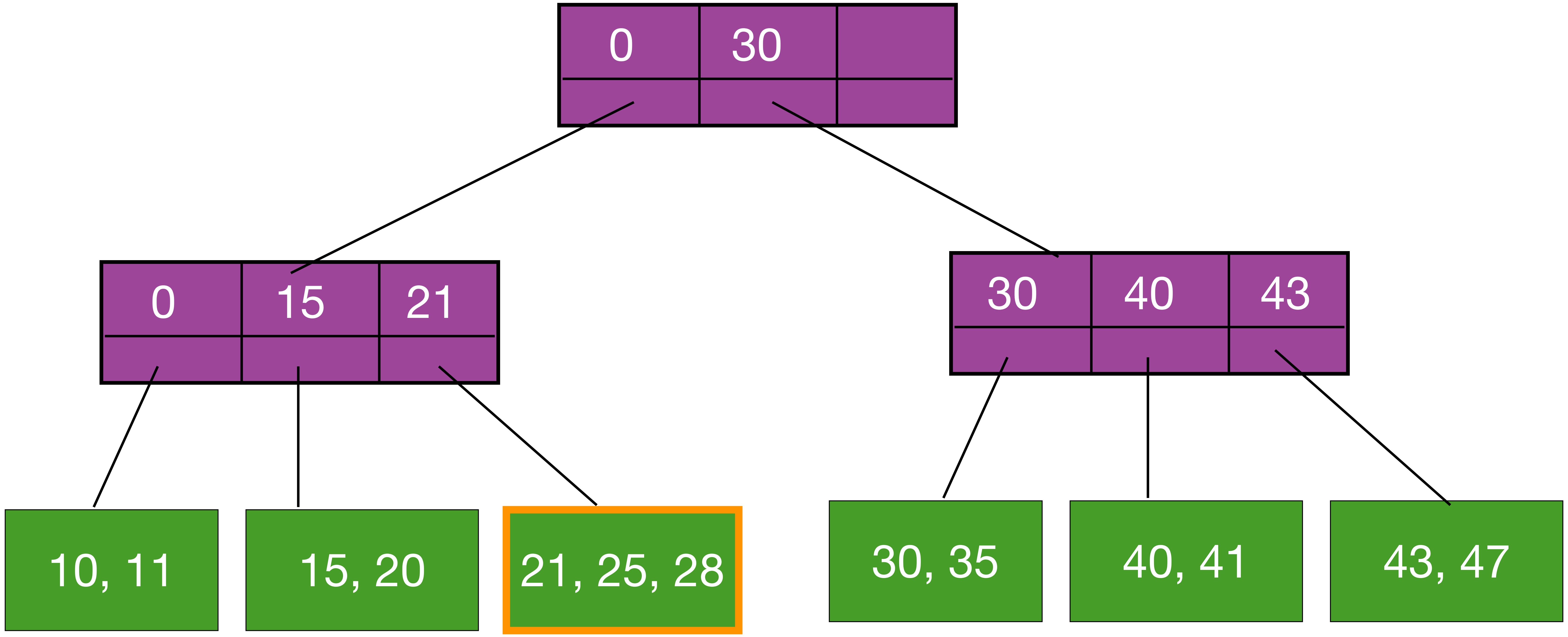
25 will be inserted here, but this node is full - split it

Exercise 1: One possible solution



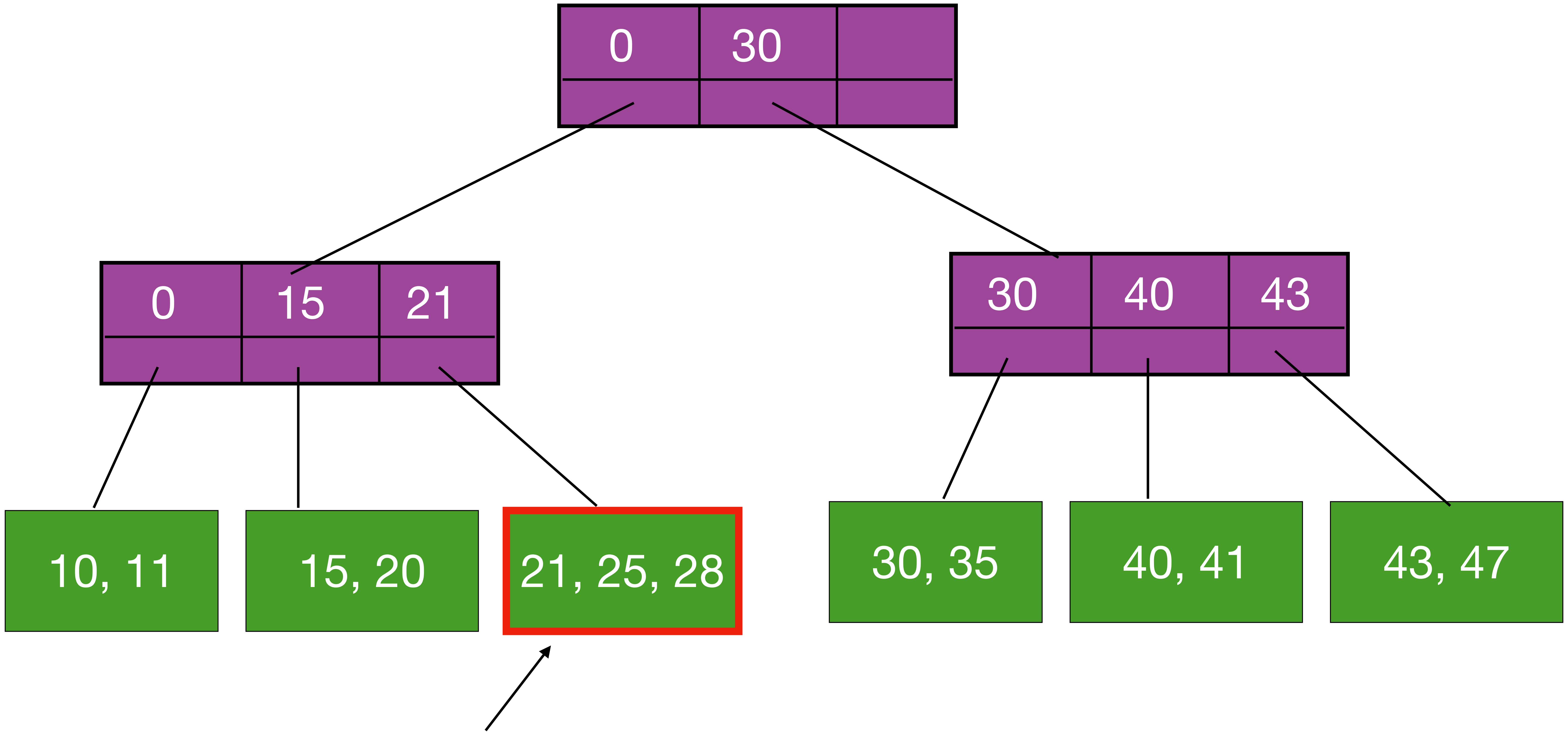
Next, insert 28

Exercise 1: One possible solution



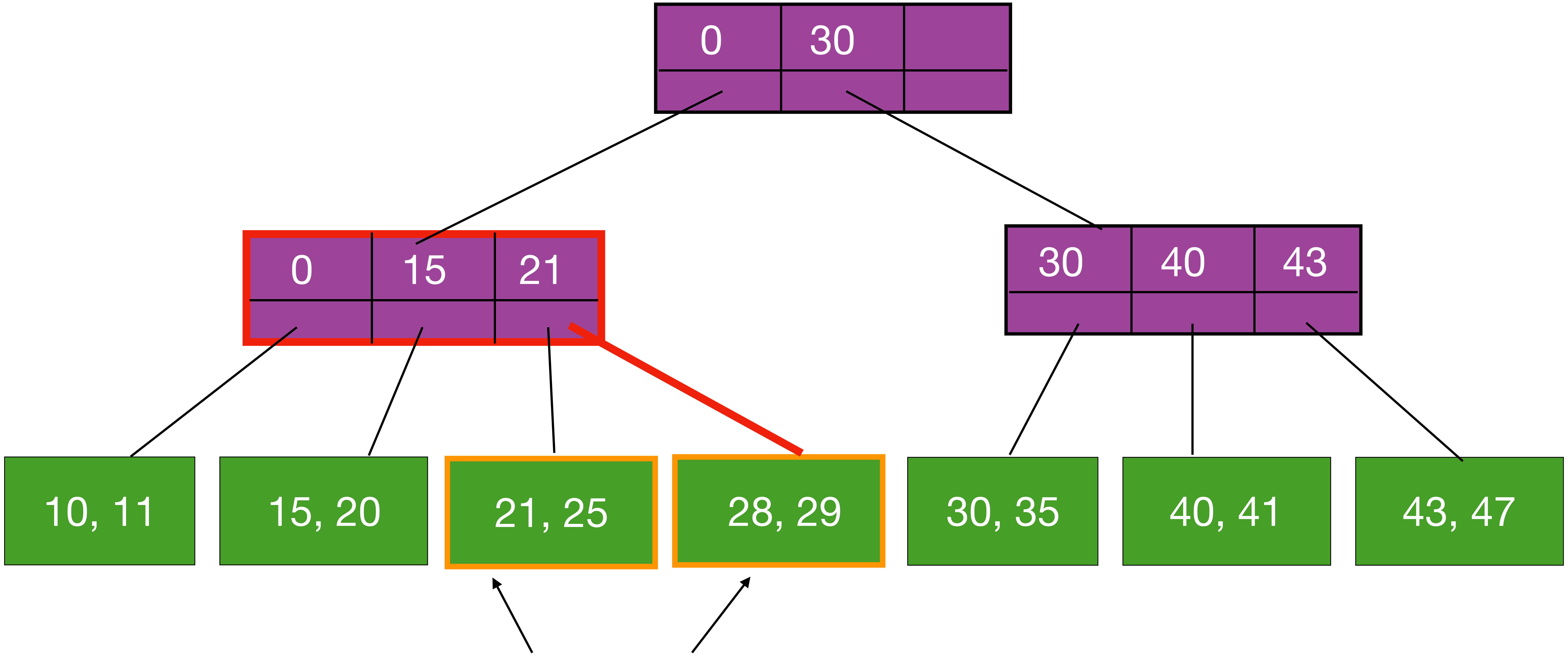
Next, insert 29

Exercise 1: One possible solution



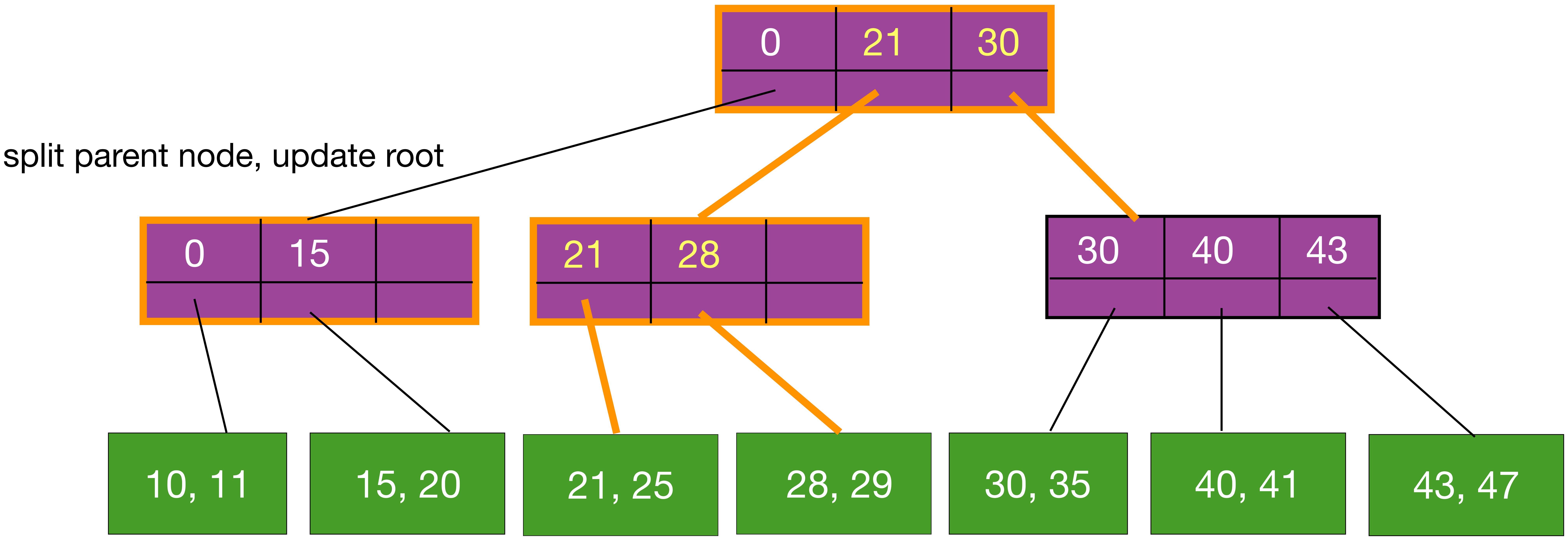
29 will be inserted here, but this node is full - split it

Exercise 1: One possible solution



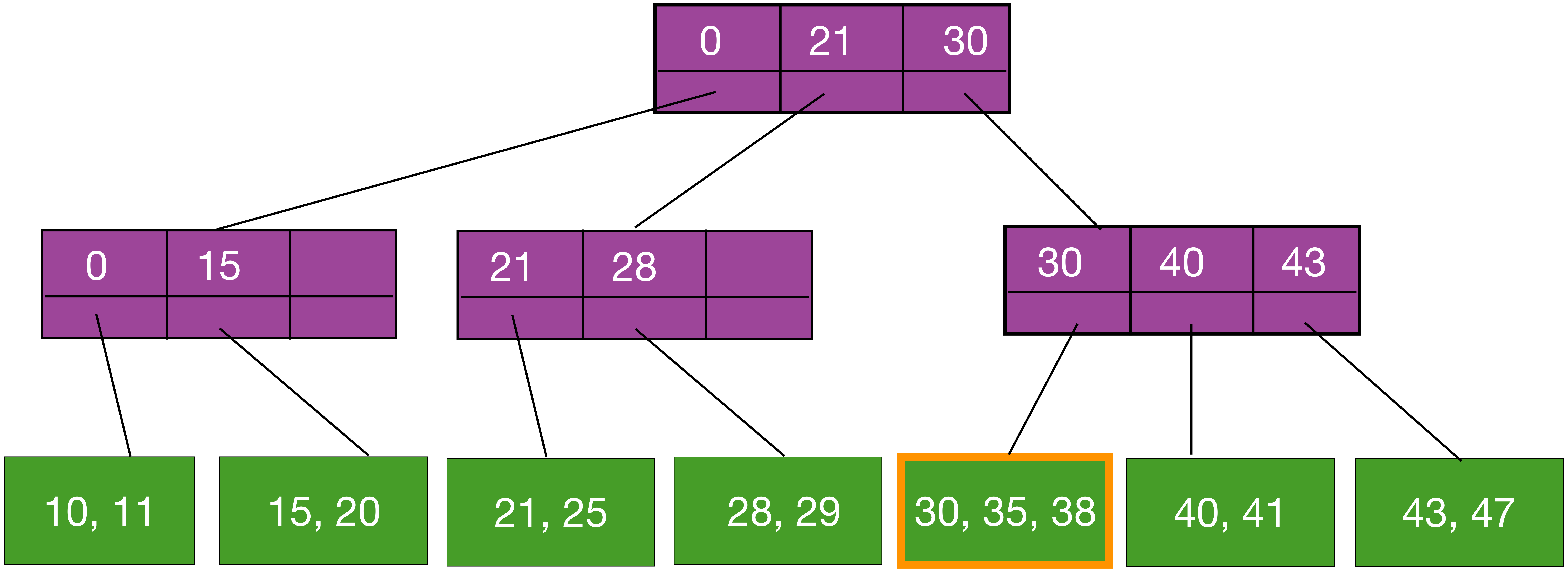
split into two nodes, but parent has overflow - split it

Exercise 1: One possible solution



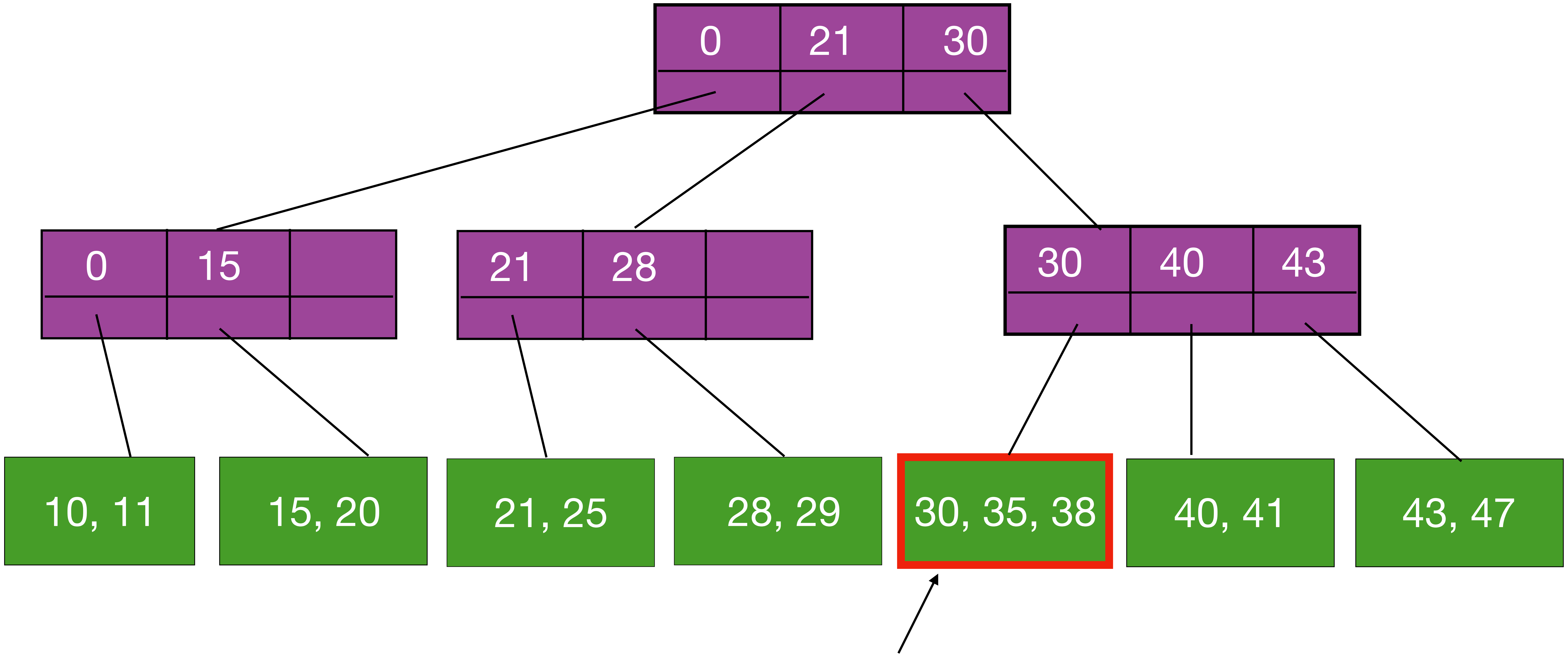
Next, insert 38

Exercise 1: One possible solution



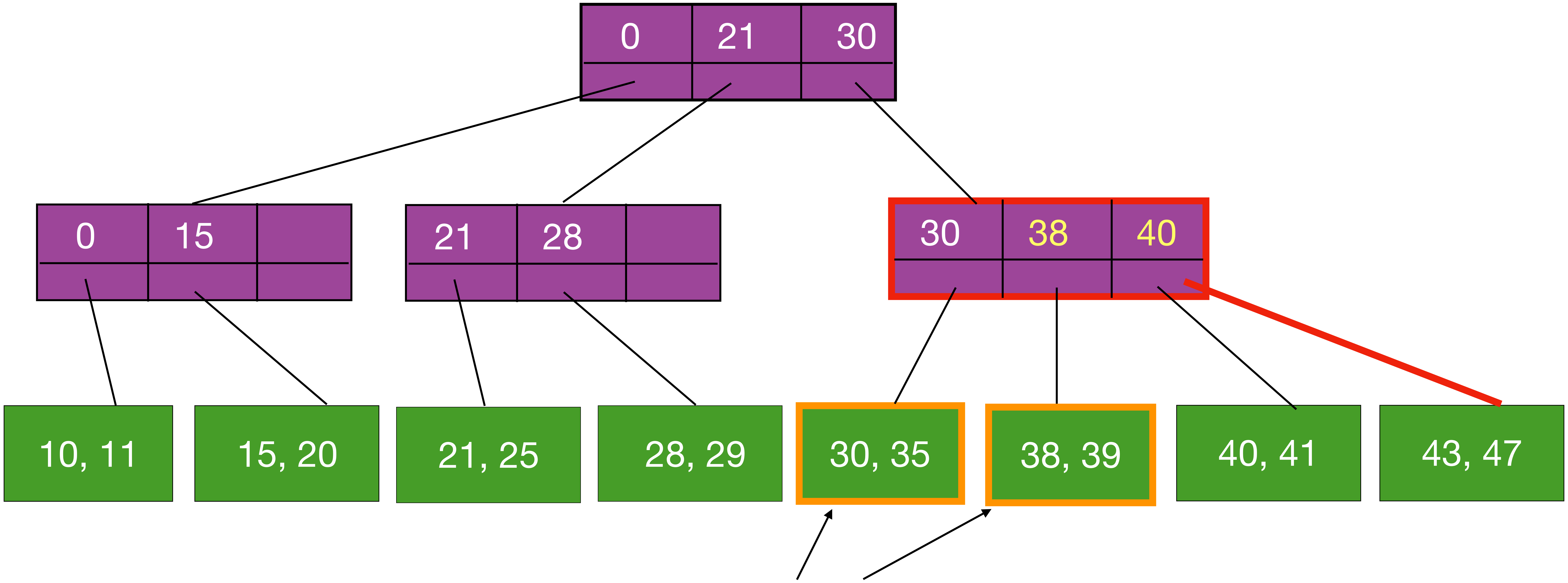
Next, insert 39

Exercise 1: One possible solution



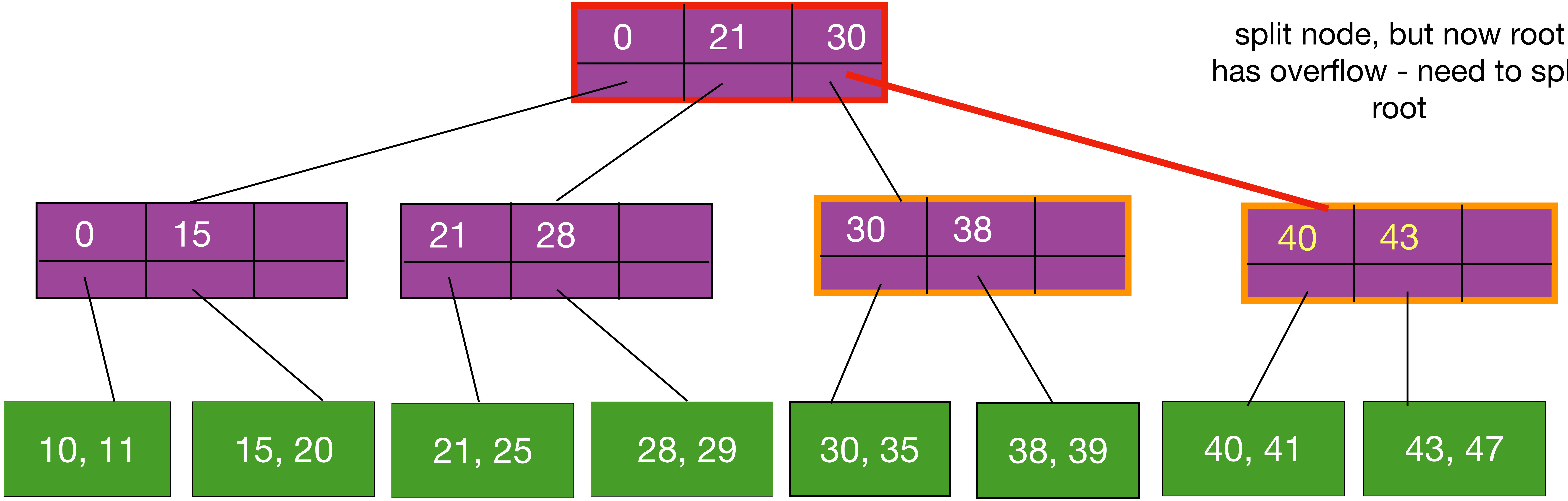
39 is inserted here, need to split this node

Exercise 1: One possible solution



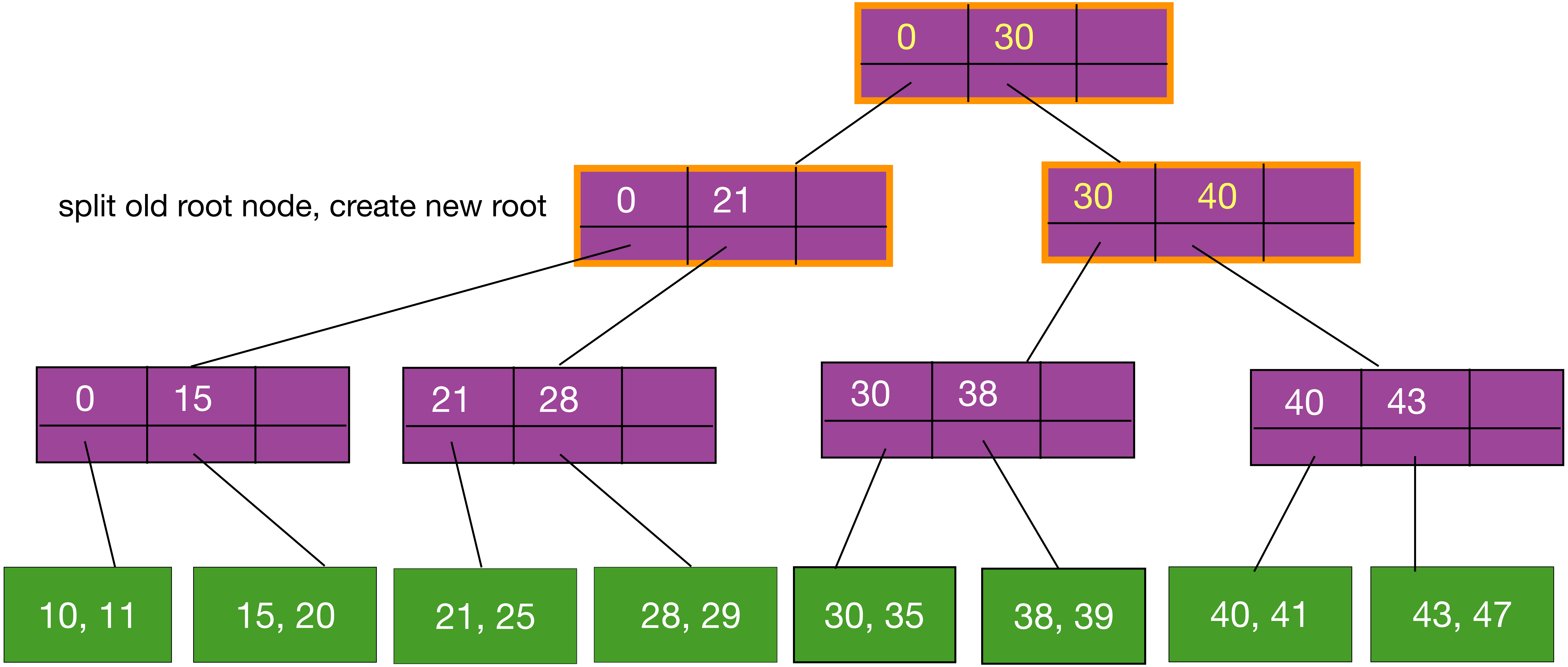
split node, but parent has overflow - need to split parent

Exercise 1: One possible solution



split node, but now root has overflow - need to split root

Exercise 1: One possible solution



Running time

- For a B-tree of order m, each disk access has an overhead of $O(\log m)$ assuming a binary search is performed to determine which direction to branch.

- The number of disk accesses is $O(\log_m n)$.

- Insert and delete operations require $O(m)$ time at each node.

- Example, m = 341 and N = 125 million: the height of tree is 4, and keeping root node in memory requires only 3 disk accesses.

- B-trees will generally be about ln 2 = 69% full.

Page size

- Processors generally support a page (logical block) size of 4KiB.

- To find the page size of the system on Linux, use this command on the terminal:

```
$getconf PAGE_SIZE
```

C code for B-tree

```
/* The Btree code is taken from Algorithms in C, 3rd edition, by Robert Sedgewick */

#define M 4  //  tree order

typedef struct STnode * link;
typedef int Key;

// data in data nodes
typedef struct {
  Key key;
  float value1;
  float value2;
}Item;

// key and pointer in index node
typedef struct {
  Key key;
  union {
    link next; // pointer to next nodes, used in index nodes
    Item item; // actual record, used in data nodes
  } ref ;
}entry;

// index node
struct STnode {
  entry b[M]; // array of M (pointer) key pairs
  int m;     // number of entries
};
```

BTree- create a new node

```
// create a new BTree
void STinit() {
  head = newLink();
  height = 0;
  N = 0;
}


link newLink() {
  link x = malloc(sizeof *x);
  x->m = 0;
  return x;
}
```

B-tree: algorithm to search for item with a given key v

- Searches recursively from root to leaf nodes
- If height is zero, a leaf node has been reached
  - check if key v is present in leaf node and return the corresponding item
- If height is non-zero, an index node has been reached
  - check each key in the node until the last key is reached or the next key is greater than v; suppose this is pointer at index j
  - recursively search the child pointed to by the jth pointer.

B-tree: C code to search for item with a given key v

```
Item searchR(link h, Key v, int height) {
   int j;

   // if at data node, return the item
   if (height == 0) {
      for (j = 0; j < h->m; j++) {
         if(eq(v, h->b[j].key)) // found key
            return h->b[j].ref.item; // return item
      }
   }

   // if an index node
   if (height != 0) {
      // check all m items in the node
      for (j = 0; j < h->m; j++) {
         // at last item or next item has greater key
         if ( j+1 == h->m || less(v, h->b[j+1].key))
            return searchR(h->b[j].ref.next, v, height - 1);
      }
   }

   return NULLItem;
}
```

B-tree: algorithm to split a node

- Splits node h to create a new node t:

1. Move the larger half of the keys from h to the new node t
2. Adjust sizes of both nodes
3. Return the new node t

- Assumptions:
  - the order M is even
  - the maximum number of keys in a node is M-1, when a node gets M keys, we split it into two nodes with M/2 keys each

B-tree : C code for node split

```
link split(link h) {
  int j;
  link t = newLink(); // new node

  // copy the last M/2 items from h to t
  for (j = 0; j < M/2; j++)
    t->b[j] = h->b[M/2+j];

  // nodes h and t contain M/2 items
  h->m = M/2;
  t->m = M/2;

  return t;
}
```

B-tree variant

- Index nodes store keys along with their corresponding data values.
  - This scheme is widely described in several textbooks.
  - However the number of accesses to reach a leaf node is very small, so there is no significant advantage to this scheme.

B+ tree

- A variant of the B-tree stores values in index nodes, which reduces the number of keys that can be stored in these nodes.

- B+ tree stores only keys and pointers in index nodes.

- Leaf nodes are linked to make it easier to access the data items in order.

    - Each leaf node contains an additional pointer to the next leaf.

B+ tree