CMSC 341
B-Trees

Based on slides from previous iterations of this course

Today's Topics

- Exam Overview
- B-Trees
  - M-Way Trees
- B-Tree Operations
  - Searching

Introduction to B-Trees

An Alternative to BSTs

- Up until now we assumed that each **node** in a BST stored the data
- What about having the data stored only in the **leaves** of the tree?
- The internal nodes simply guide our search to the leaf, which contains the data we want
- (We'll restrict this discussion of such trees to those in which all leaves are at the same level)

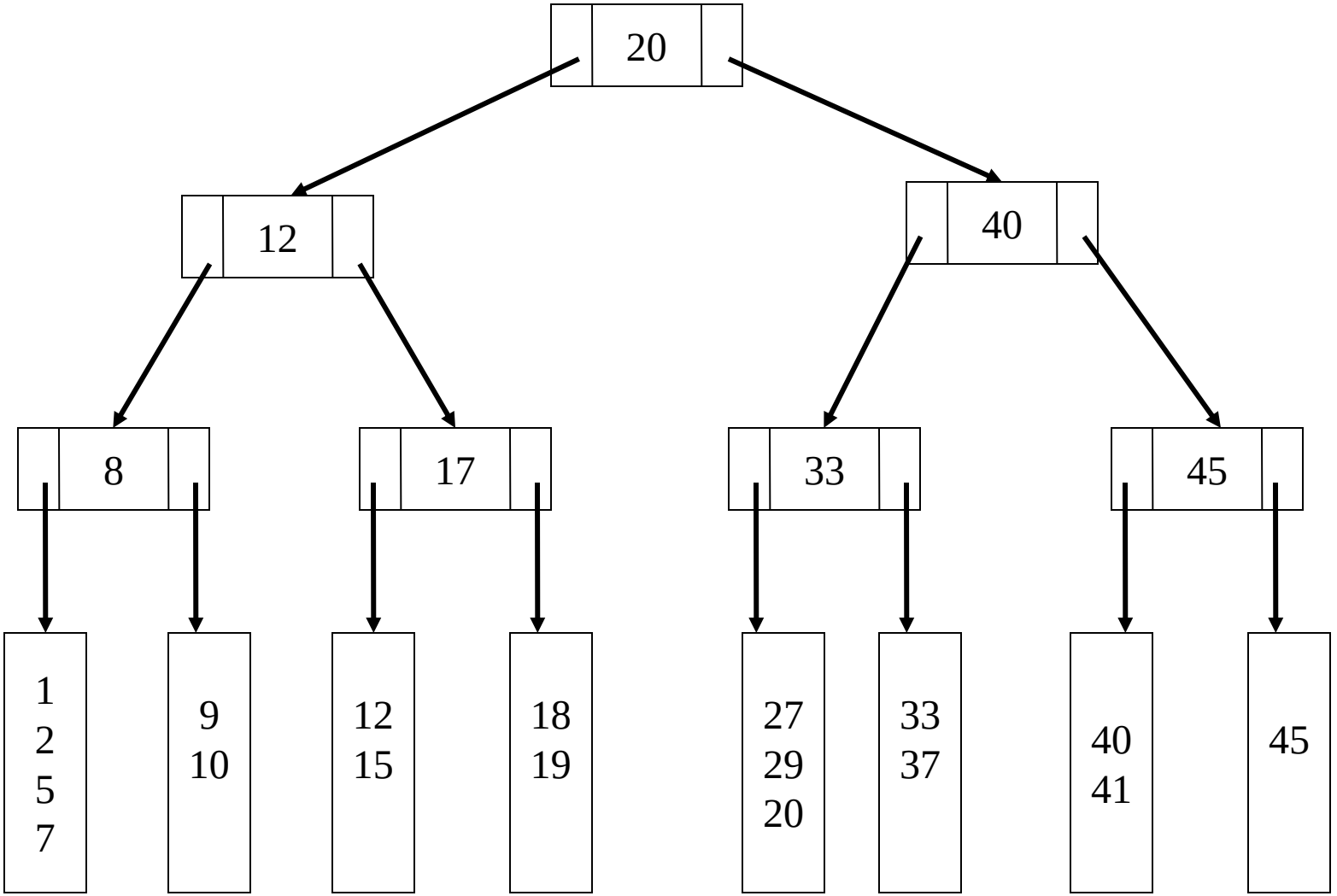
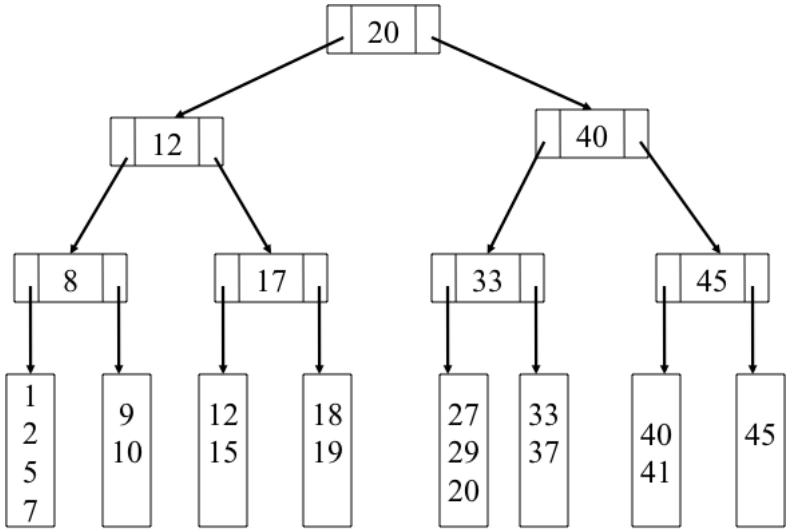


Figure 1 - A BST with data stored in the leaves

Properties

- Store data only at leaves; all leaves at same level
  - Interior and exterior nodes have different structure
  - Interior nodes store one key and two subtree pointers
  - All search paths have same length: $\lceil \log n \rceil$ (assuming one element per leaf)
  - Can store multiple data elements in a leaf



M-Way Trees

- A generalization of the previous BST model
  - Each interior node has M subtrees pointers and M-1 keys
  - *e.g.*, “2-way tree” or “M-way tree of order 2”
- As M increases, height decreases: $\lceil \log_M n \rceil$ (assuming one element per leaf)
- A perfect M-way tree of height h has M^h leaves

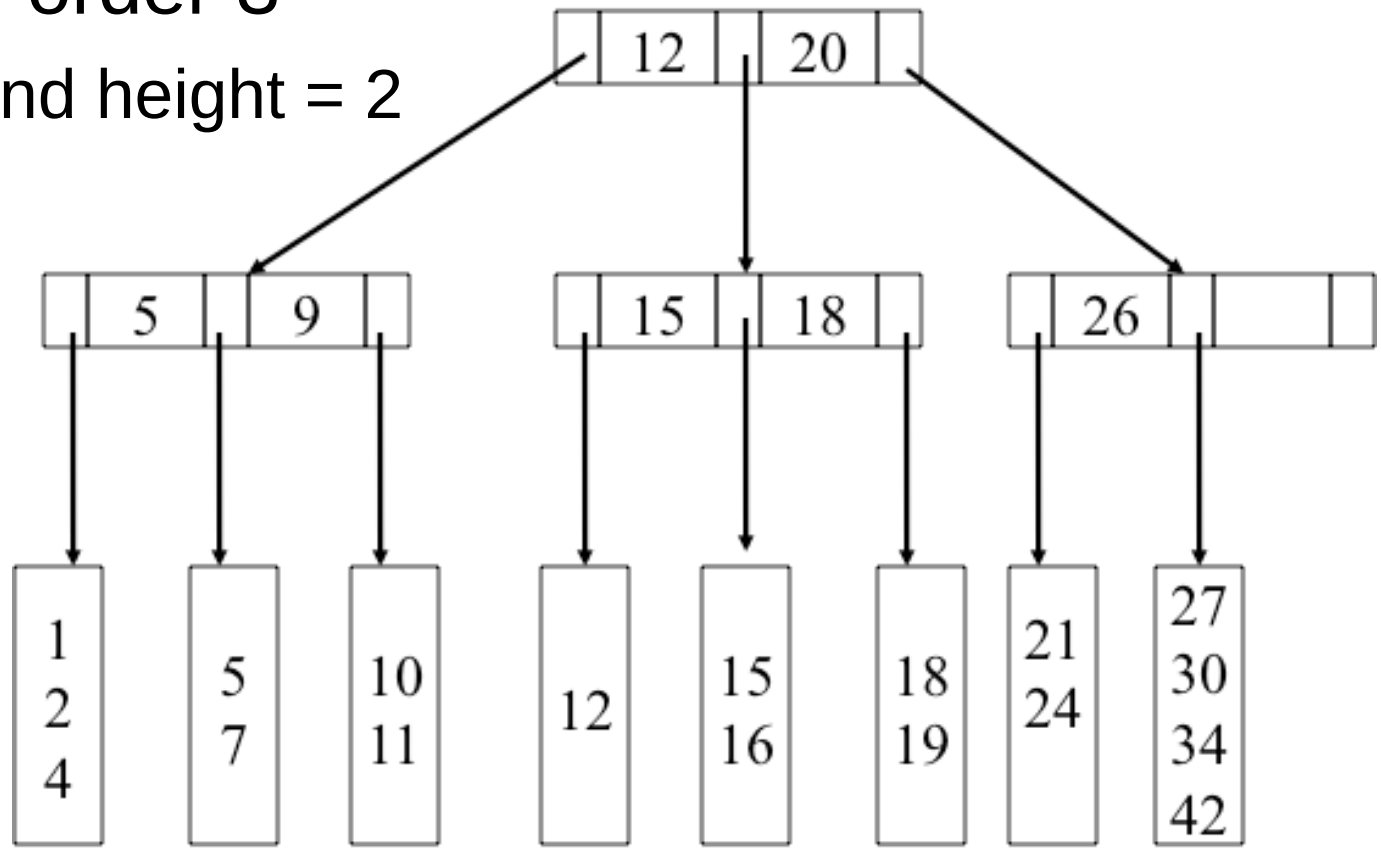
B-Trees

- A B-Tree is an M-Way Tree that satisfies two important properties:

1. It is perfectly balanced
   (All leaves are at the same height)
2. Every node is at least half full (>= M values)
   (Possible exception for the root)

A B-Tree of Order 3

- B-Tree of order 3
  - M = 3 and height = 2



- Tree can support 9 leaves (but it has only 8)

Searching in a B-Tree

- Different from standard BST search
  - Search always terminates at a leaf node
  - May scan more than one element at a leaf
  - May scan more than one key at an interior node

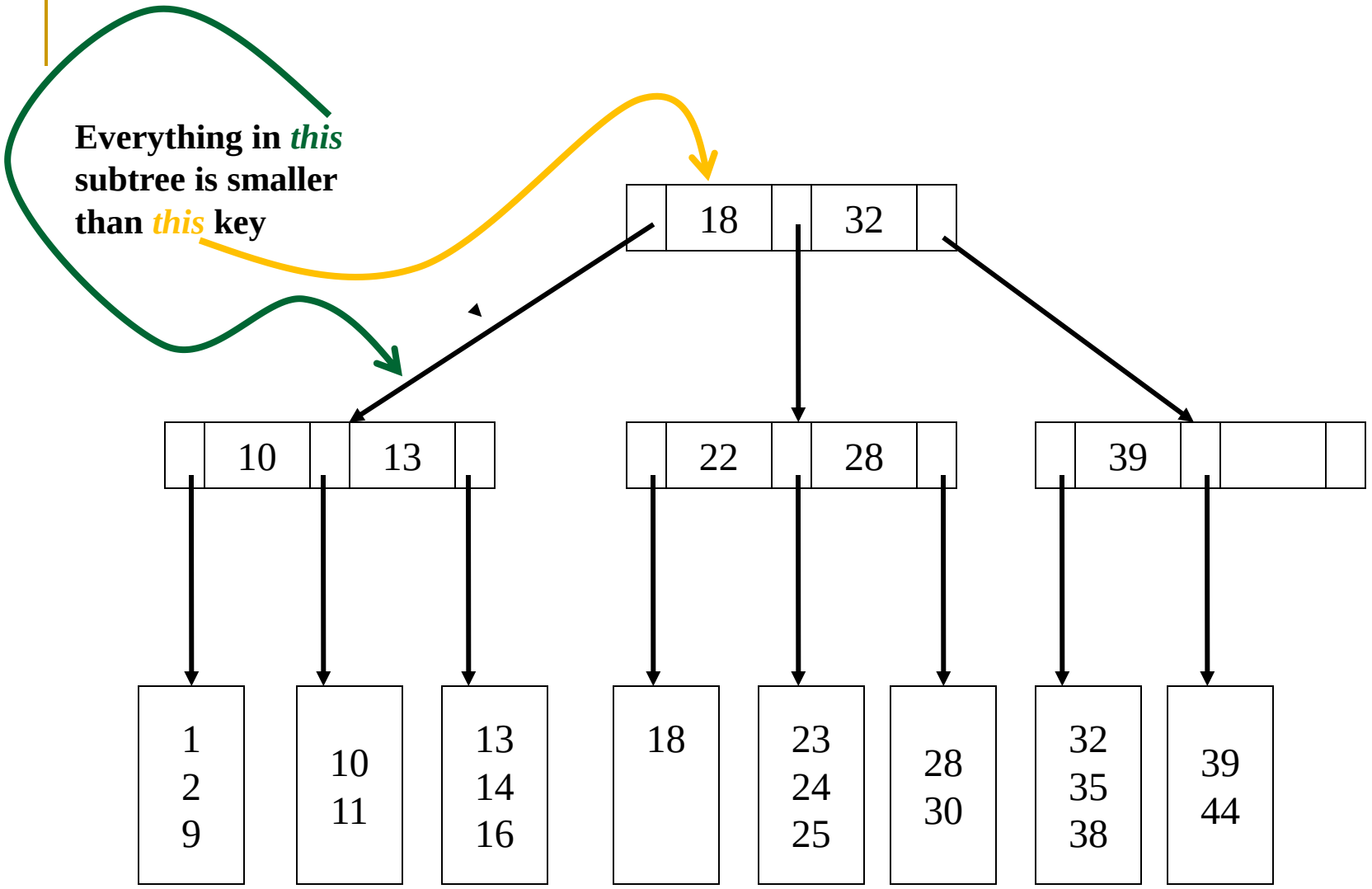
- Trade-offs
  - Tree height decreases as M increases
  - Computation at each node during search increases as M increases

Searching in a B-Tree: Code

```
Search (MWayNode v, DataType element)
{
   if (v == NULL) { return failure; }

   if (v is a leaf) {
      // search the list of values looking for element
      // if found,  return success
      // otherwise, return failure
   }

   else {   // if v is an interior node
      // search the keys to find subtree element is in
      // recursively search the subtree
   }
}
```



In any interior node, find the *first* key > search item, and traverse the link to the left of that key. Search for any item >= the last key in the subtree pointed to by the rightmost link. Continue until search reaches a leaf.

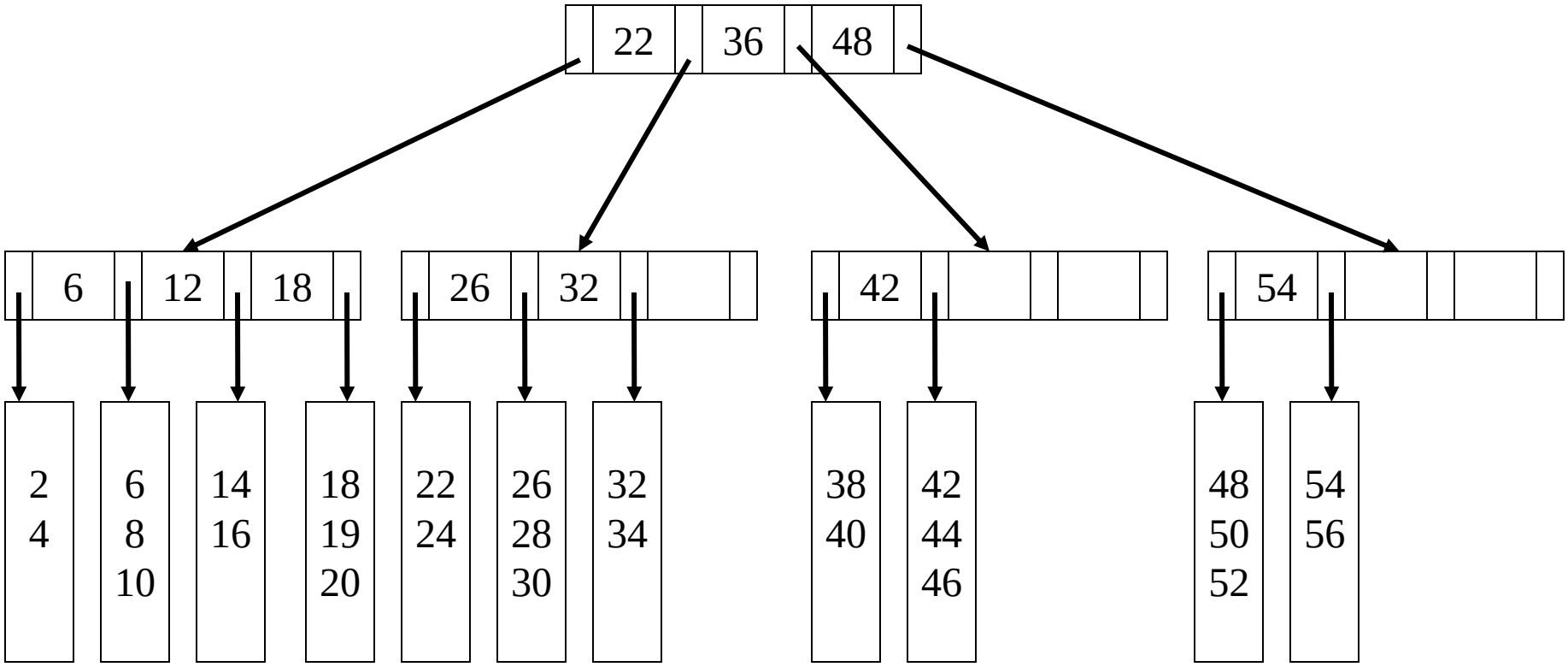


Figure 3 – searching in an B-Tree of order 4

Is It Worth It?

- Is it worthwhile to reduce the height of the search tree by letting M increase?
- Although the number of nodes visited decreases, the amount of computation at each node increases
- Where's the payoff?

An Example

- Consider storing 10^7 = 10,000,000 items in a balanced BST or in an B-Tree of order 10
- The height of the BST will be $\log_2(10^7) \approx 24$.
- The height of the B-Tree will be $\log_{10}(10^7) = 7$ (assuming that we store just 1 record per leaf)
- In the BST, just one comparison will be done at each interior node
- In the B-Tree, 9 will be done (worst case)

Why Use B-Trees?

- If it takes longer to descend the tree than it does to do the extra computation
  - This is exactly the situation when the nodes are stored externally (*e.g.*, on disk)
  - Compared to disk access time, the time for extra computation is insignificant
- We can reduce the number of accesses by sizing the B-Tree to match the disk block and record size

A Generic M-Way Tree Node

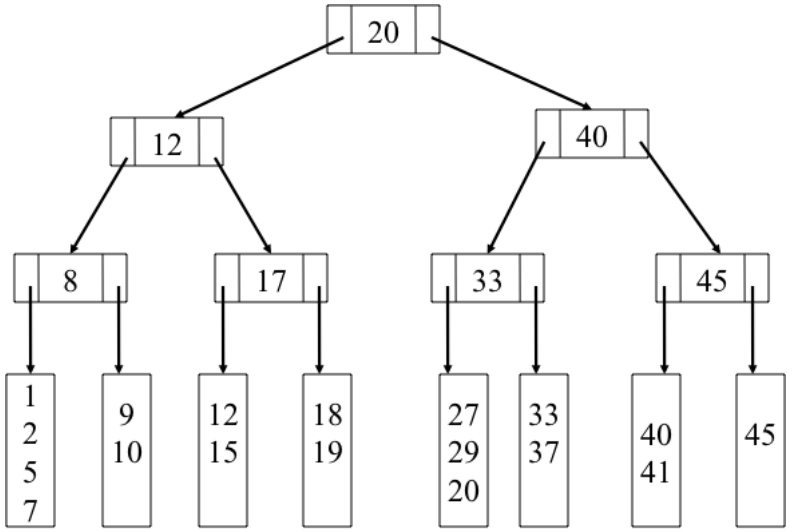
```
public class MwayNode<Ktype, Dtype>
{
    // code for public interface here
    // constructors, accessors, mutators

    private boolean isLeaf;          // true if node is a leaf
    private int m;                   // the "order" of the node
    private int nKeys;               // nr of actual keys used
    private ArrayList<Ktype> keys;   // array of keys(size = m - 1)
    private MWayNode subtrees[ ];    // array of pts (size = m)
    private int nElems;              // nr poss. elements in leaf
    private List<Dtype> data;        // data storage if leaf
}
```

M-Way Trees and B-Trees

Review: M-Way Trees

- Data is stored only at the leaves
  - A leaf may have multiple elements of data
- Interior nodes are used for "navigation"
  - Locating a value
  - But do not store any information themselves
- As M increases, the height decreases



B-Tree Definition

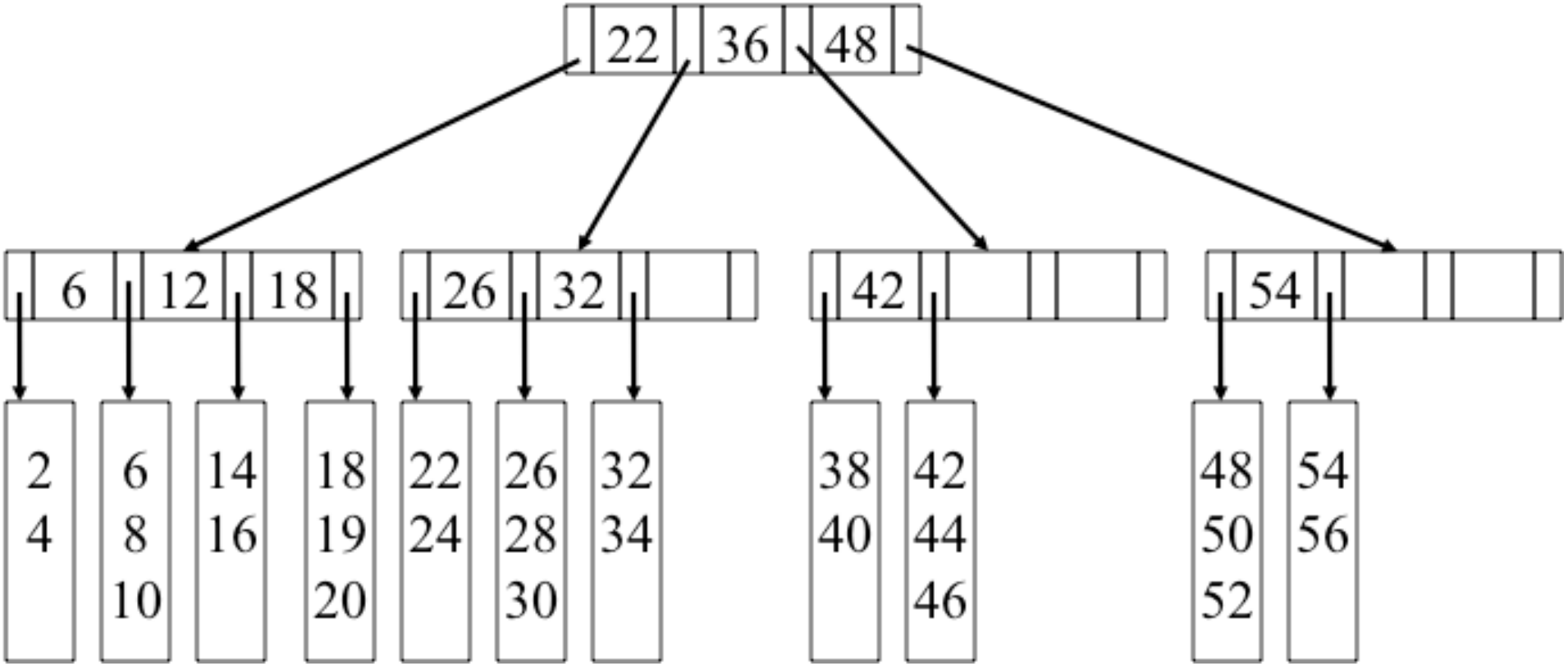
- A B-Tree of order M is an M-Way tree with the following constraints
  - The root is either a leaf or has between 2 and M subtrees
  - All interior node (except maybe the root) have between $\lceil M / 2 \rceil$ and M subtrees
    - Each interior node is at least “half full”
  - All leaves are at the same level
  - A leaf stores between $\lceil L / 2 \rceil$ and L data elements
    - Except when the tree has fewer than L/2 elements
    - L is a fixed constant >= 1

B-Tree Example

- For a B-Tree with M = 4 and L = 3

- The root node can have between 2 and 4 subtrees
- Each interior node can have between
  - 2 and 4 subtrees → $\lceil M / 2 \rceil = 2$
  - Up to 3 keys → $M - 1 = 3$
- Each exterior node (leaf) can hold between
  - 2 and 3 data elements → $\lceil L / 2 \rceil = 2$

B-Tree Example



Designing a B-Tree

Why Use B-Trees?

- B-trees are often used when there is too much data to fit in memory
- Each node/leaf access costs one disk access

- When choosing M and L, keep in mind
  - The size of the data stored in the leaves
  - The size of the keys
  - Pointers stored in the interior nodes
  - The size of a disk block

Example: B-Tree for Students Records

- B-Tree stores student records:
  - Name, address, other data, etc.

- Total size of records is 1024 bytes
- Assume that the key to each student record is 8 bytes long (SSN)
- Assume that a pointer (really a disk block number) requires 4 bytes
- Assume that our disk block is 4096 bytes

Example B-Tree: Calculating L

- L is the number of data records that can be stored in each leaf
- Since we want to do just one disk access per leaf, this should be the same as the number of data records per disk block
- Since a disk block is 4096 and a data record is 1024, we choose 4 data records per leaf
  - L = $\lfloor$ 4096 / 1024 $\rfloor$

Example B-Tree: Calculating M

- To keep the tree flat and wide, we want to maximize the value of M
  - Also want just one disk access per interior node
- Use the following relationship:
  - 4(M) + 8(M – 1) <= 4096
- So 342 is the largest possible M that makes the tree as shallow as possible

Example B-Tree: Performance

- With M = 342 the height of our tree for N students will be $\lceil \log_{342} \lceil N / L \rceil \rceil$
- For example, with N = 100,000 the height of the tree with M = 342 would be no more than 2, because $\lceil \log_{342} \lceil 100000 / 4 \rceil \rceil = 2$
- So any record can be found in 3 disk accesses
  - If the root is stored in memory, then only 2 disk accesses are needed

B-Tree Operations
Insertion

Inserting Into a B-Tree

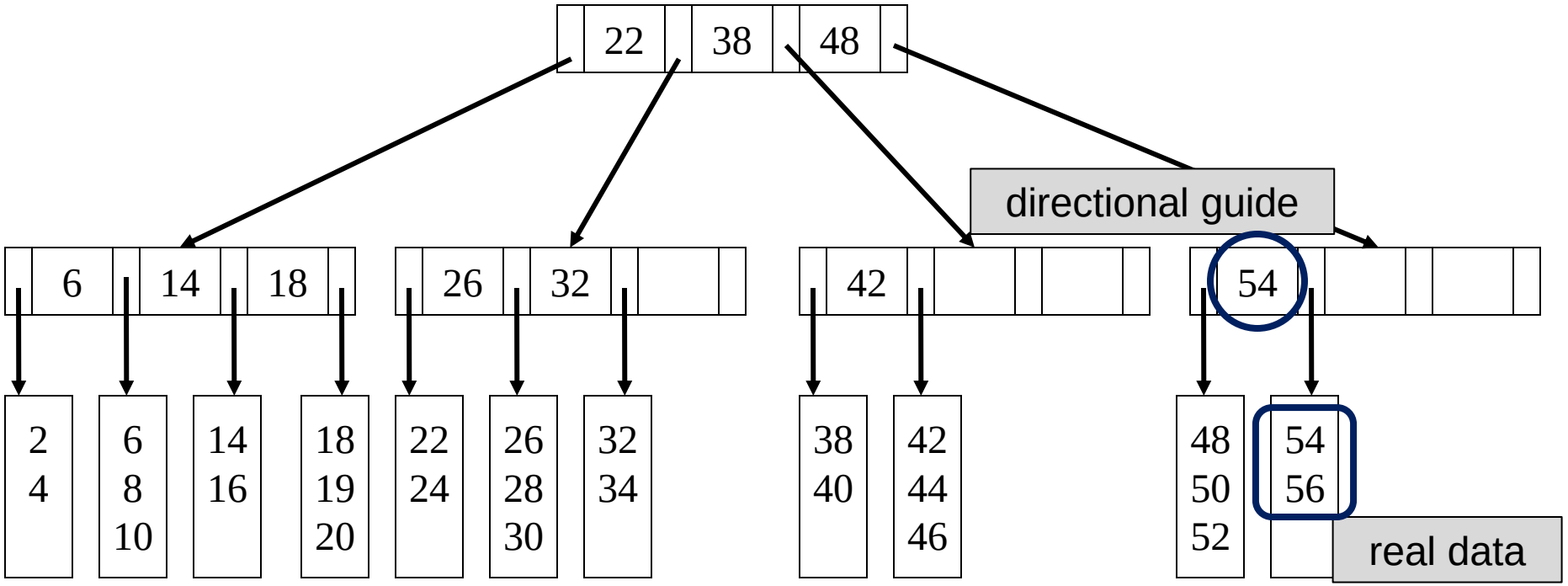
- Search to find the leaf into which the new value (X) should be inserted
- If the leaf has room (fewer than L elements), insert X and write the leaf back to the disk
- If the leaf is full, split it into two leaves, each containing half of the elements
  - Insert X into the appropriate new leaf

Inserting Into a B-Tree

- If a leaf has been split, we need to update the keys in the parent interior
- To choose a new key for the parent interior node, there are a variety of methods
  - One is to use the median data value as the key
- If the parent node is already full, split it in the same manner; splits propagate up to the root
  - This is how the tree grows in height

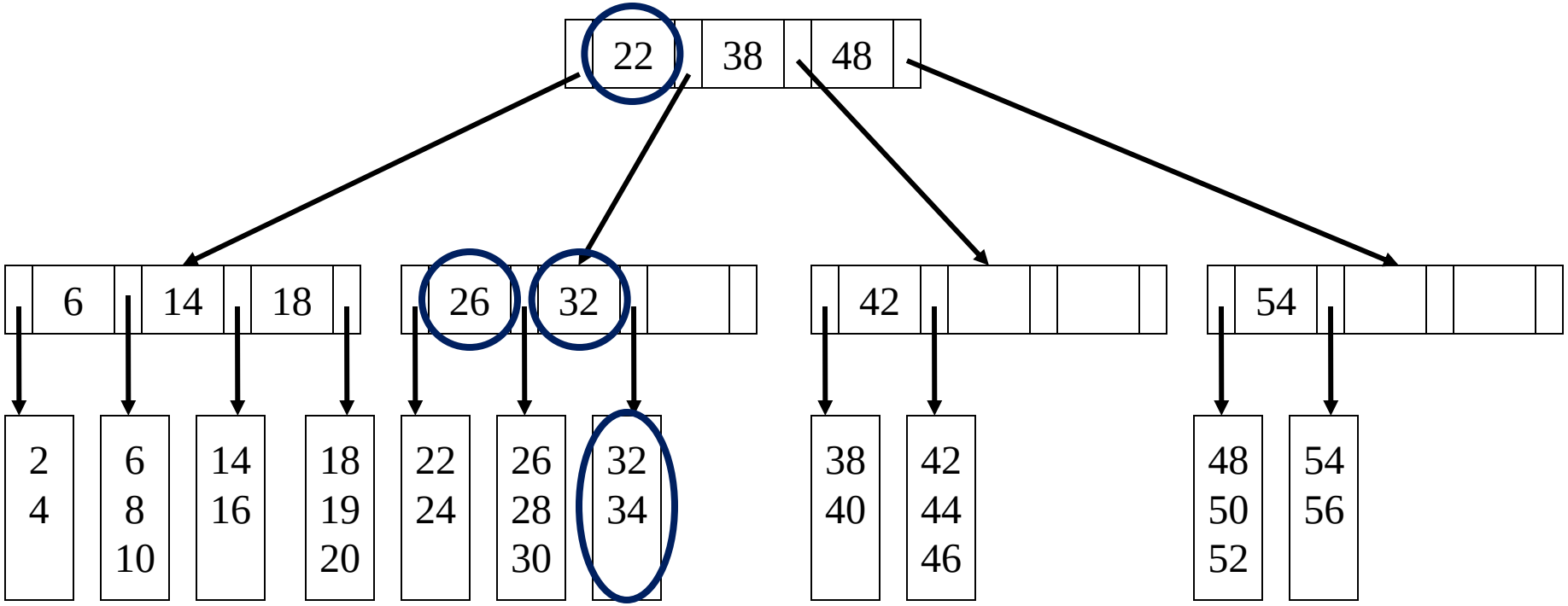
Insertion Example

- Insert 33, 35, and 21 into the tree below



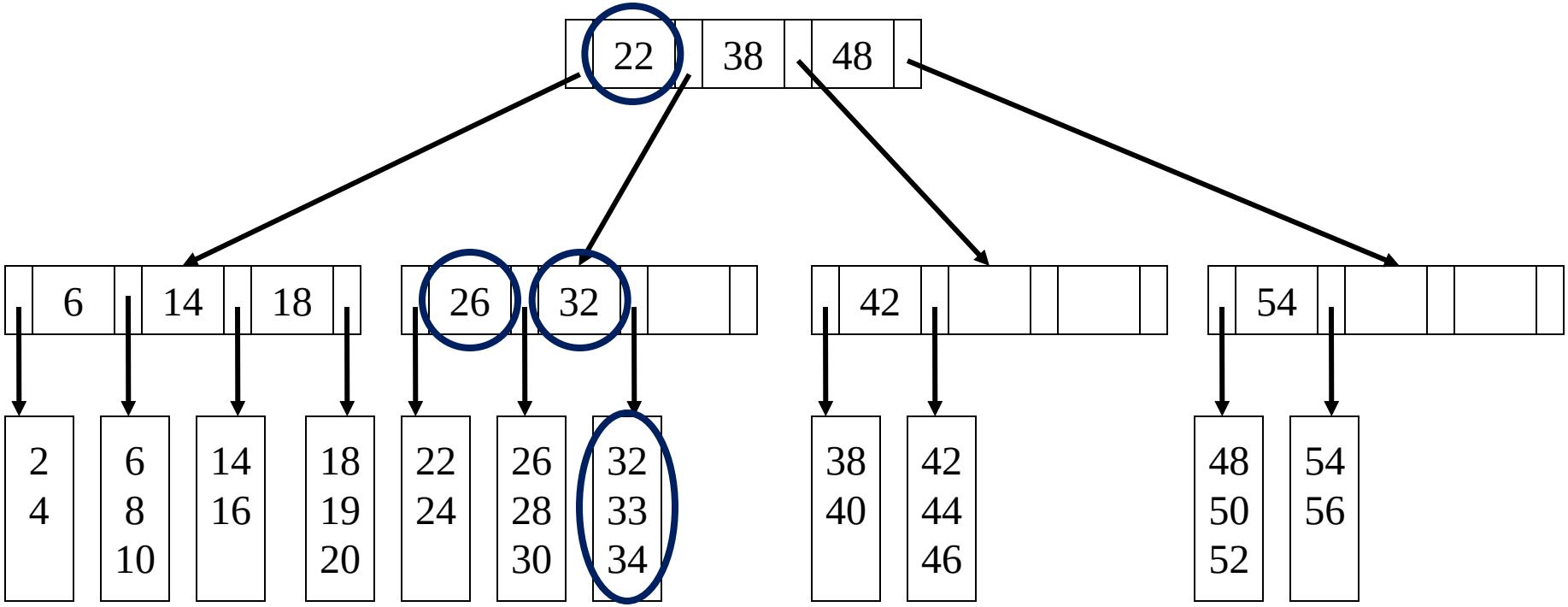
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Traverse to find place to insert



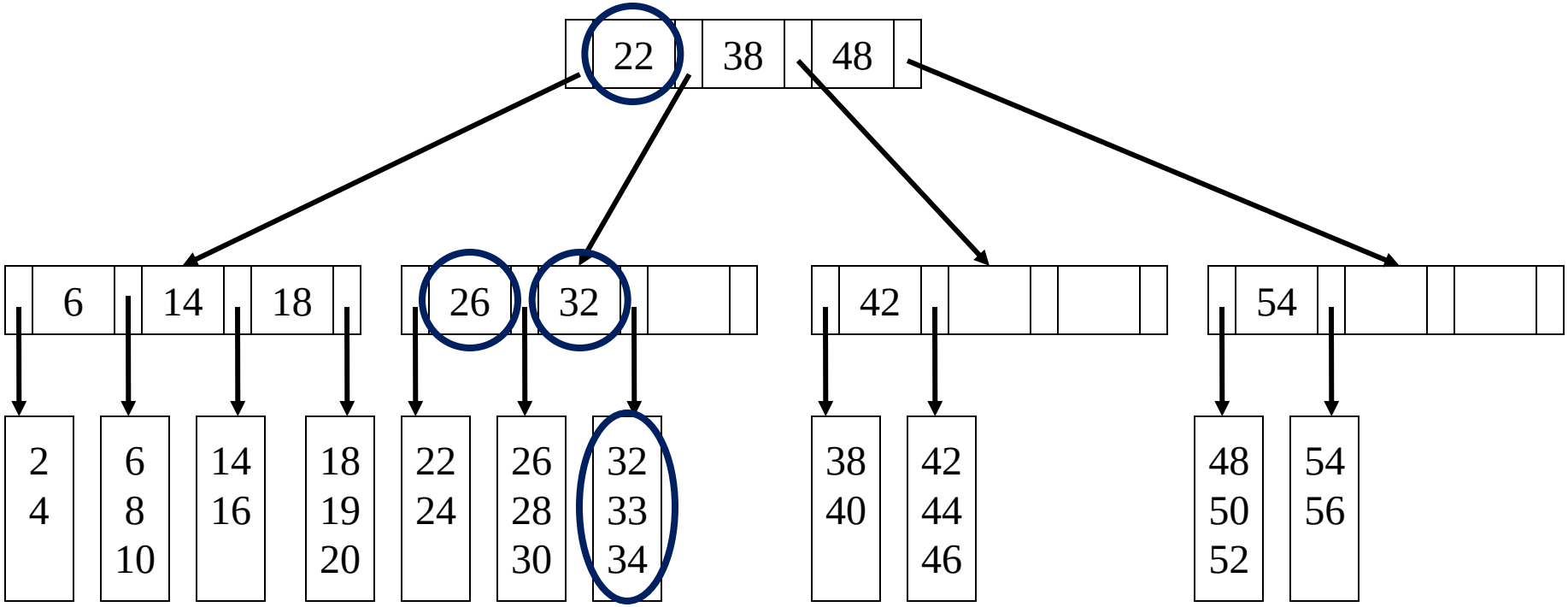
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Insert value



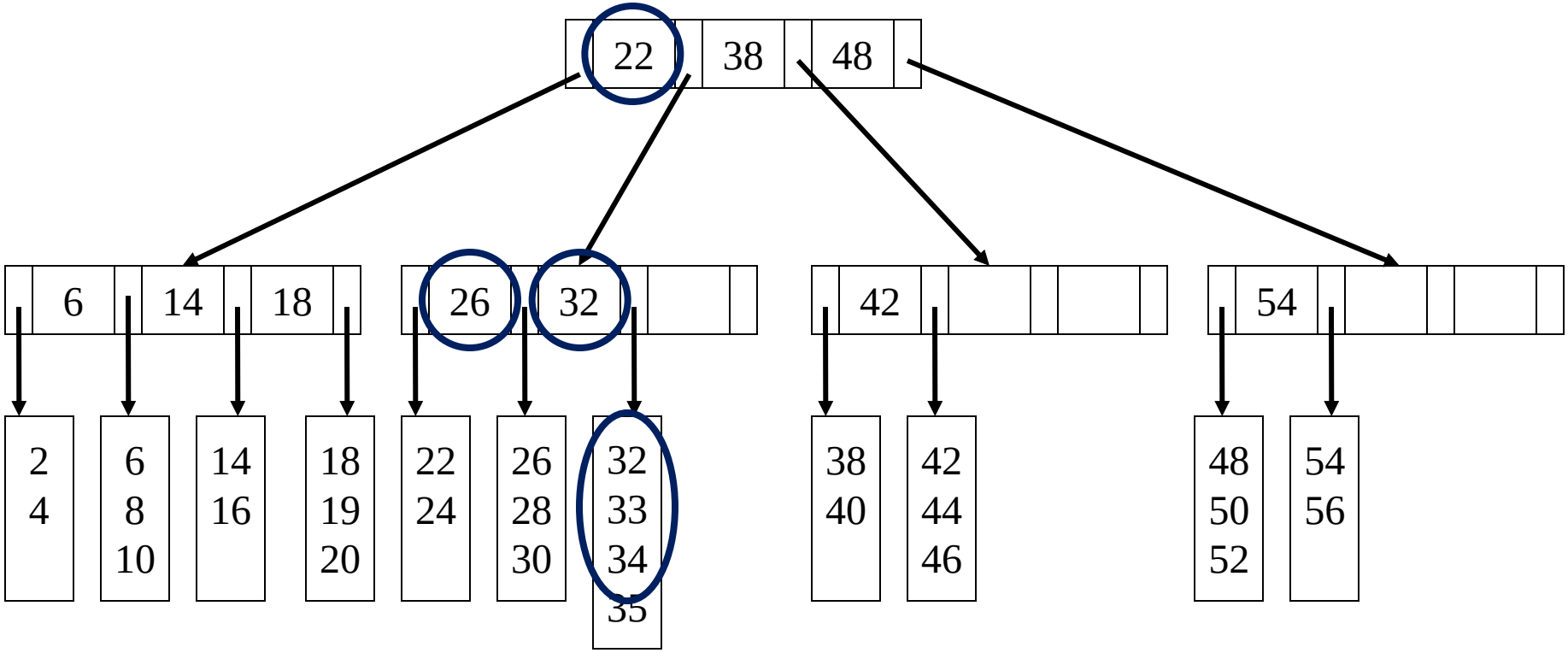
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Traverse to find place to insert



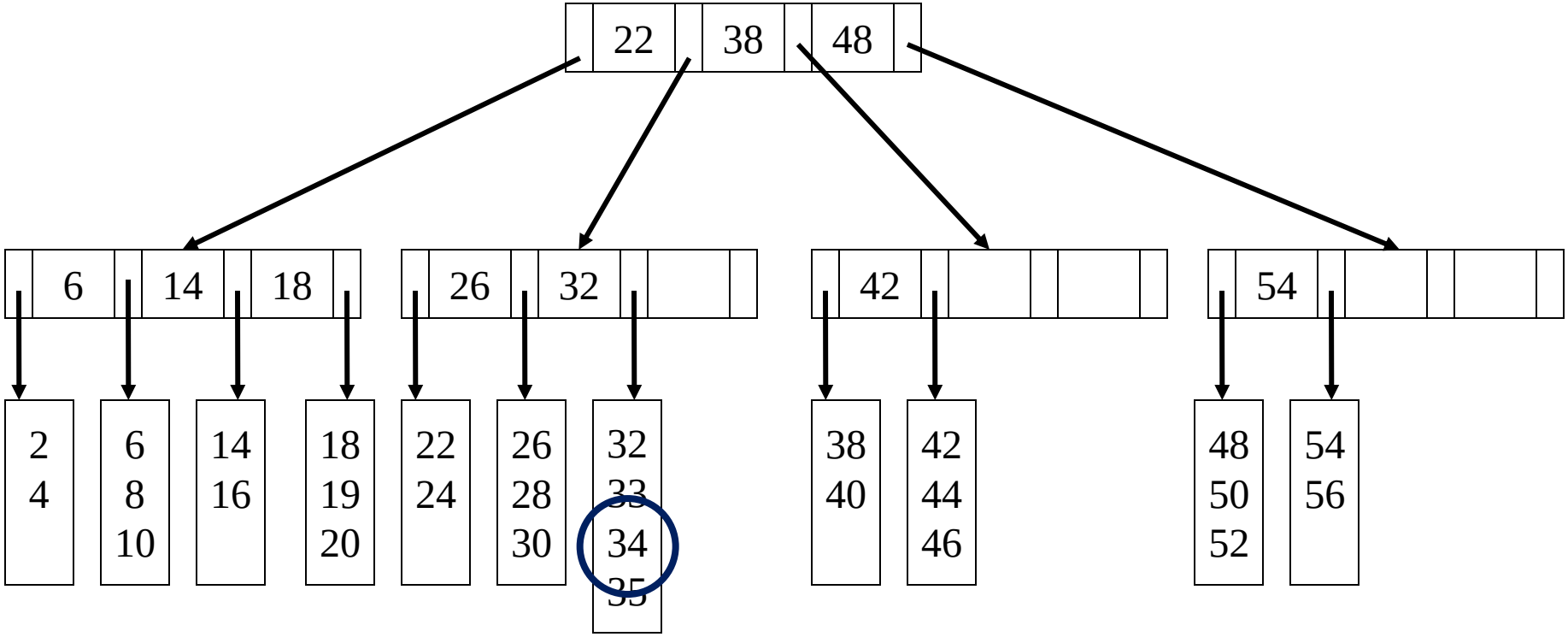
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Insert value – no room!



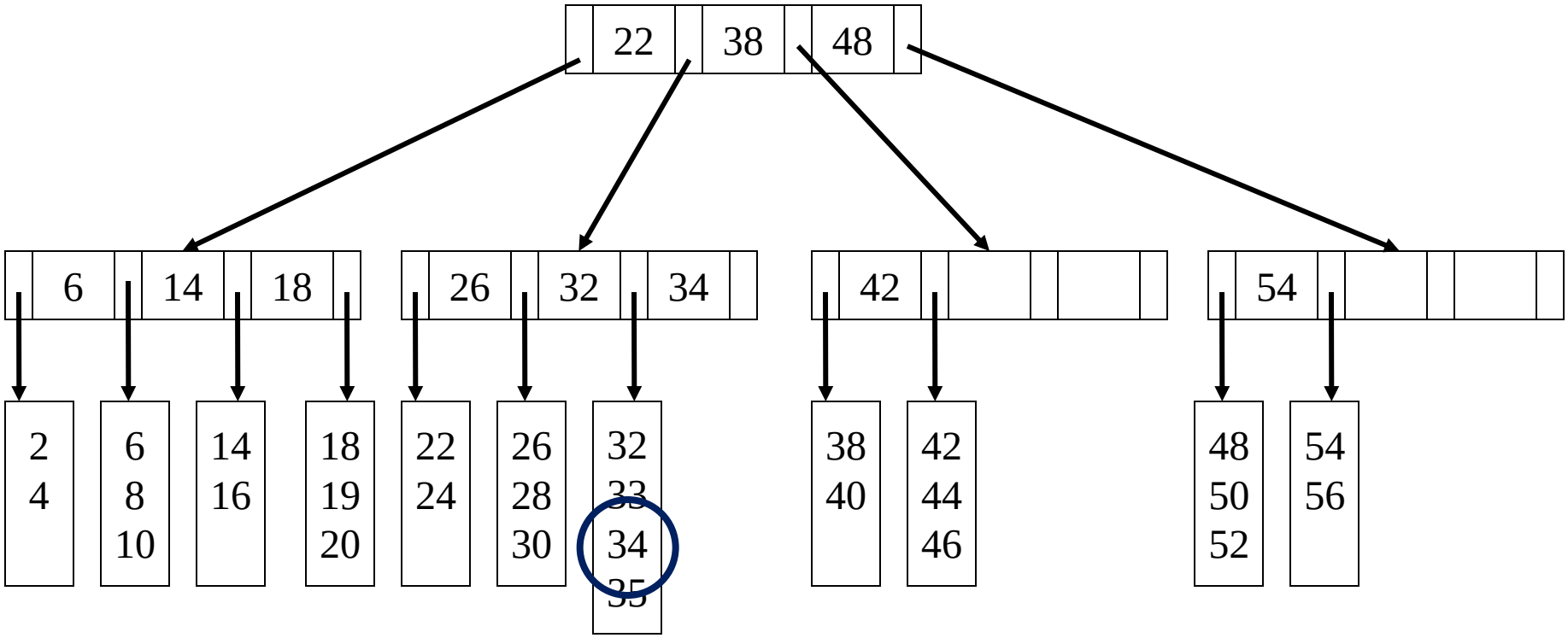
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Find the median for the new key in the parent



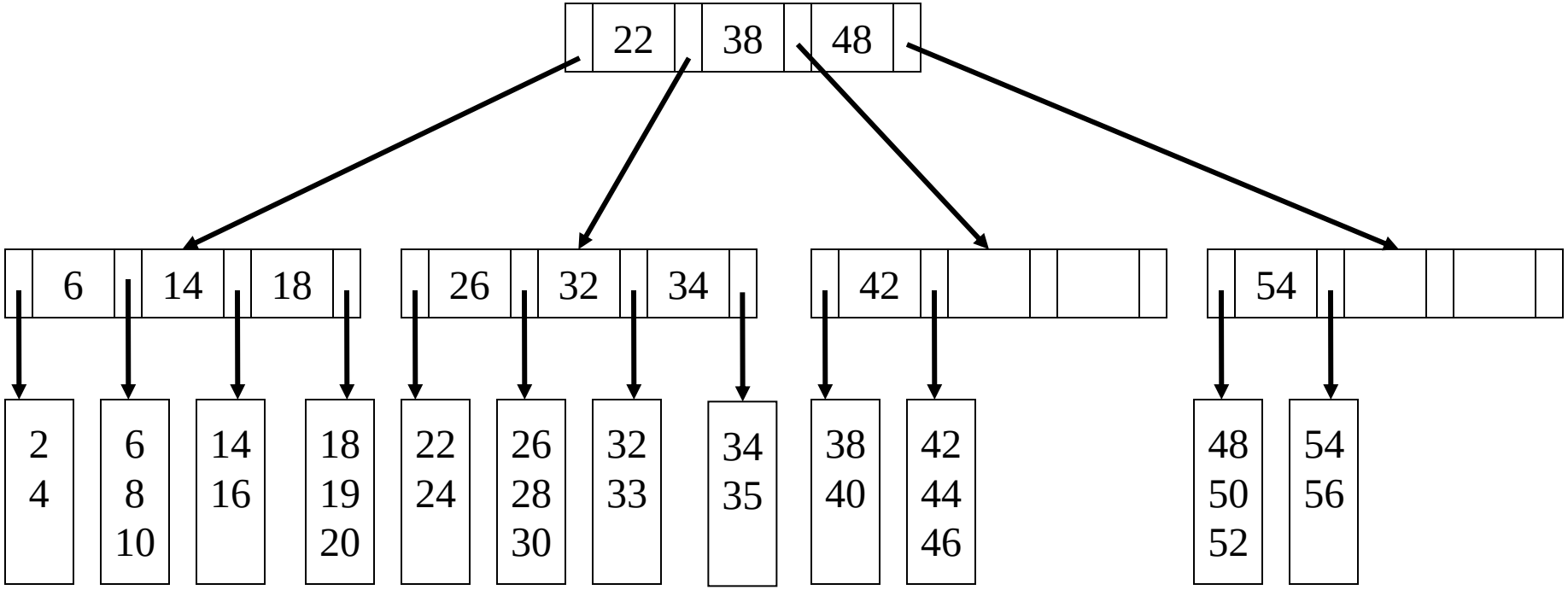
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Update the parent node



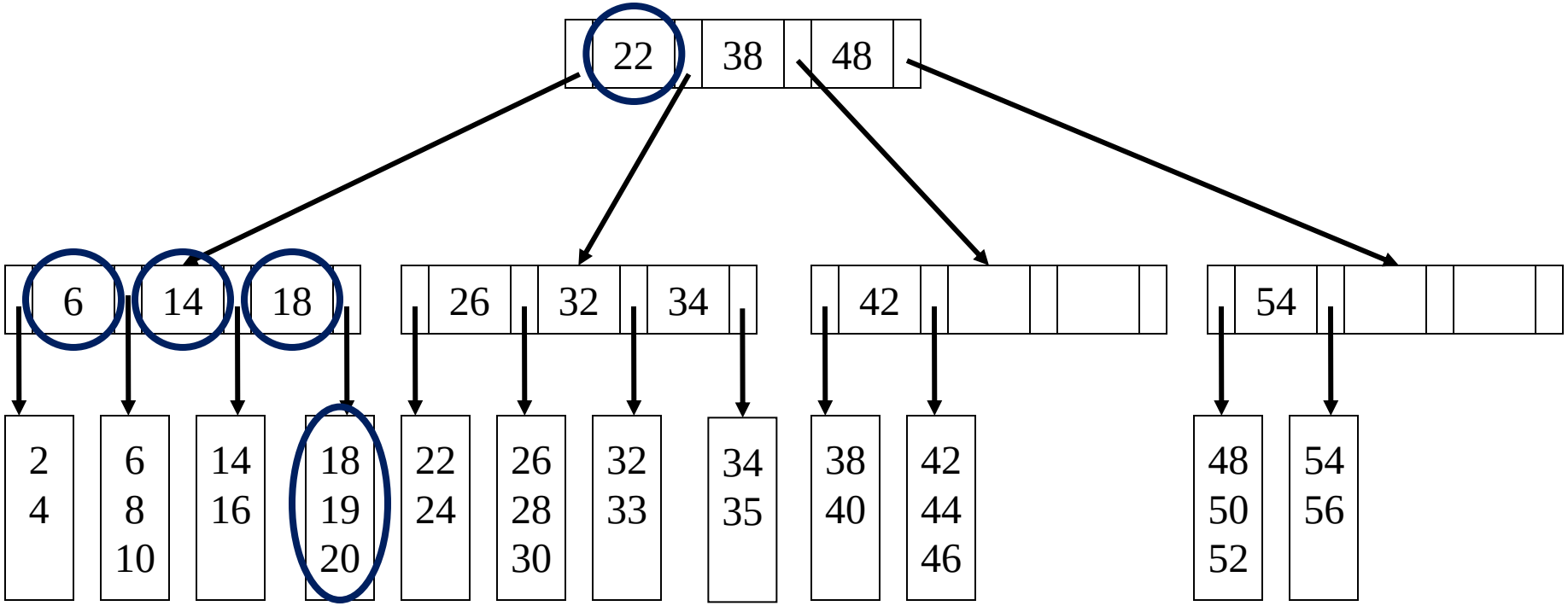
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Split the leaf into two leaves



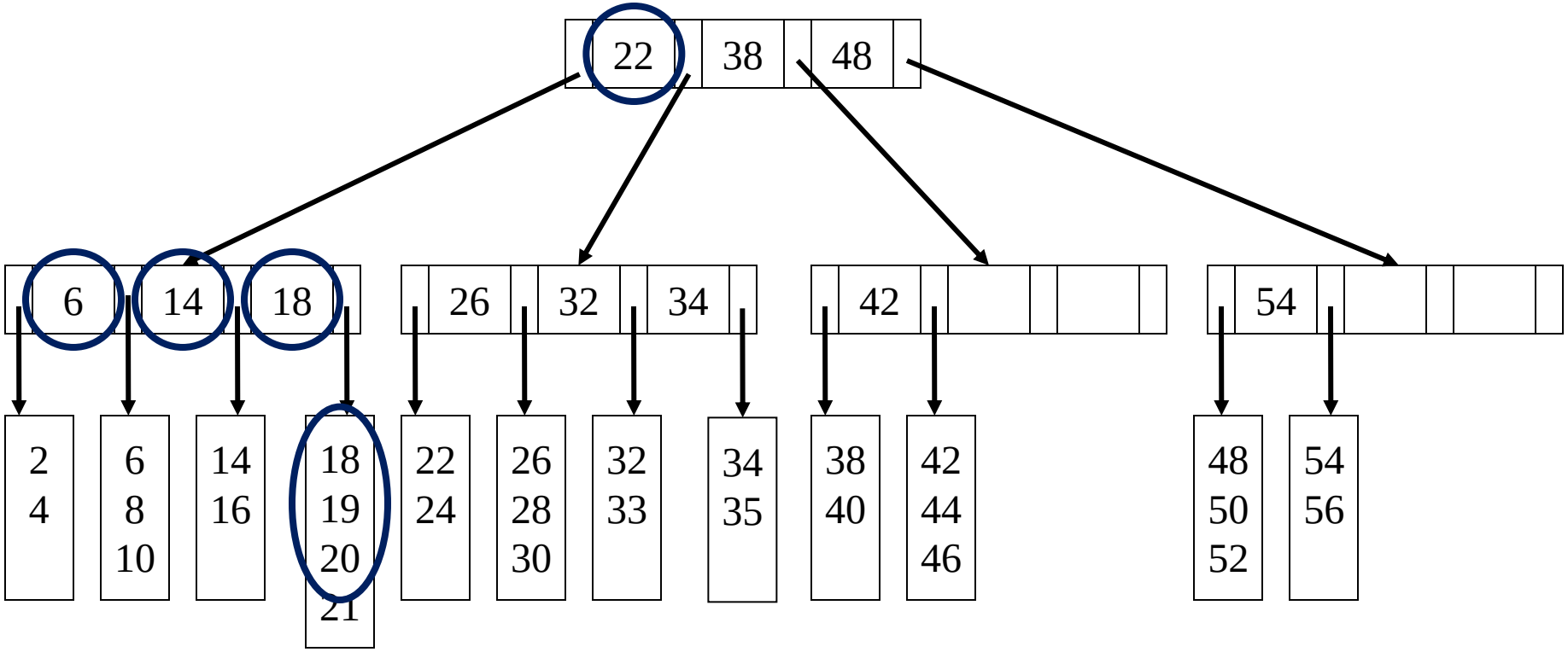
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Traverse to find place to insert



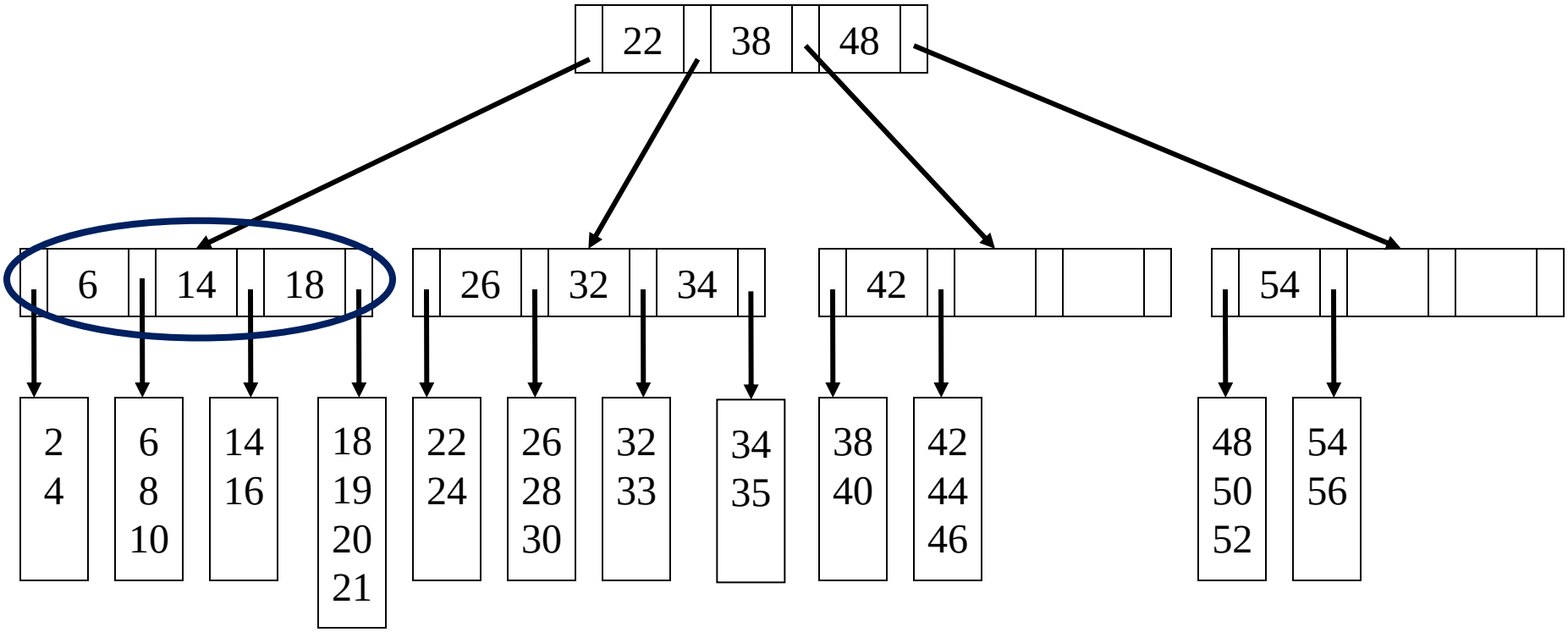
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Insert value – no room!



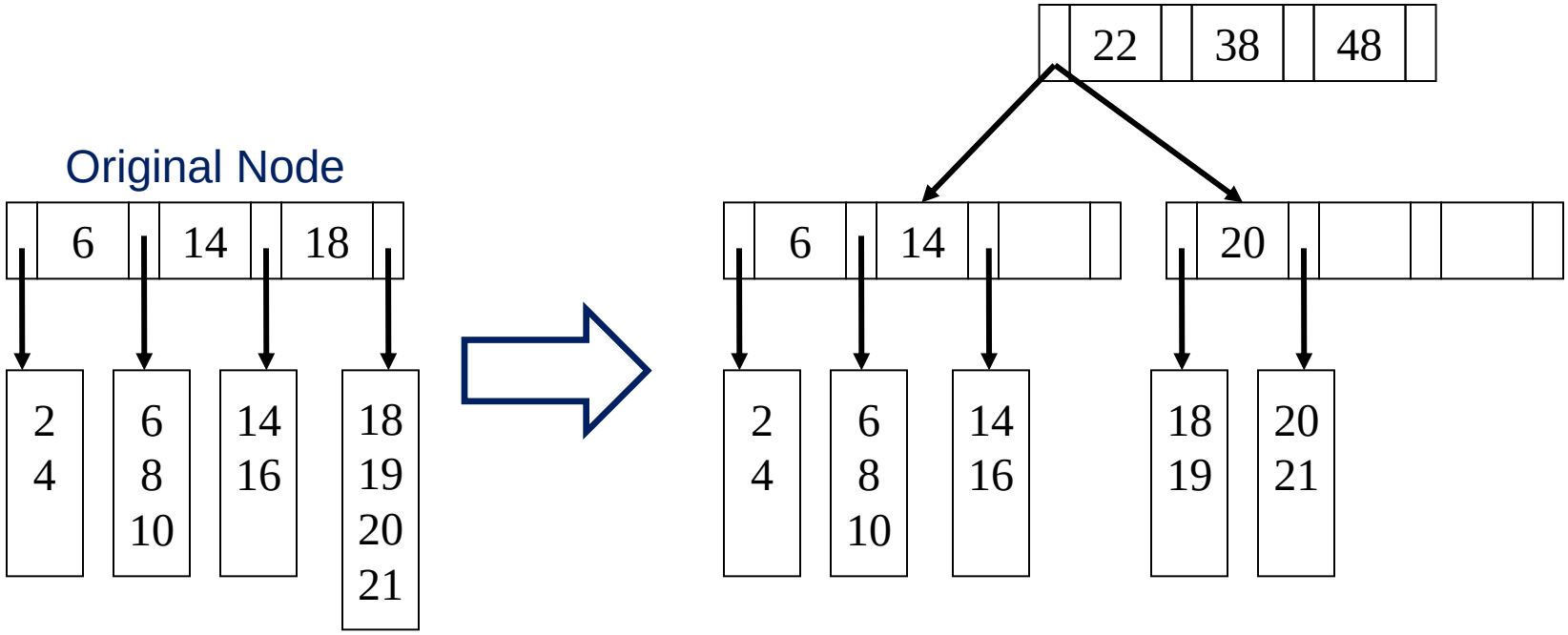
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - And no room to add directly to the parent node!



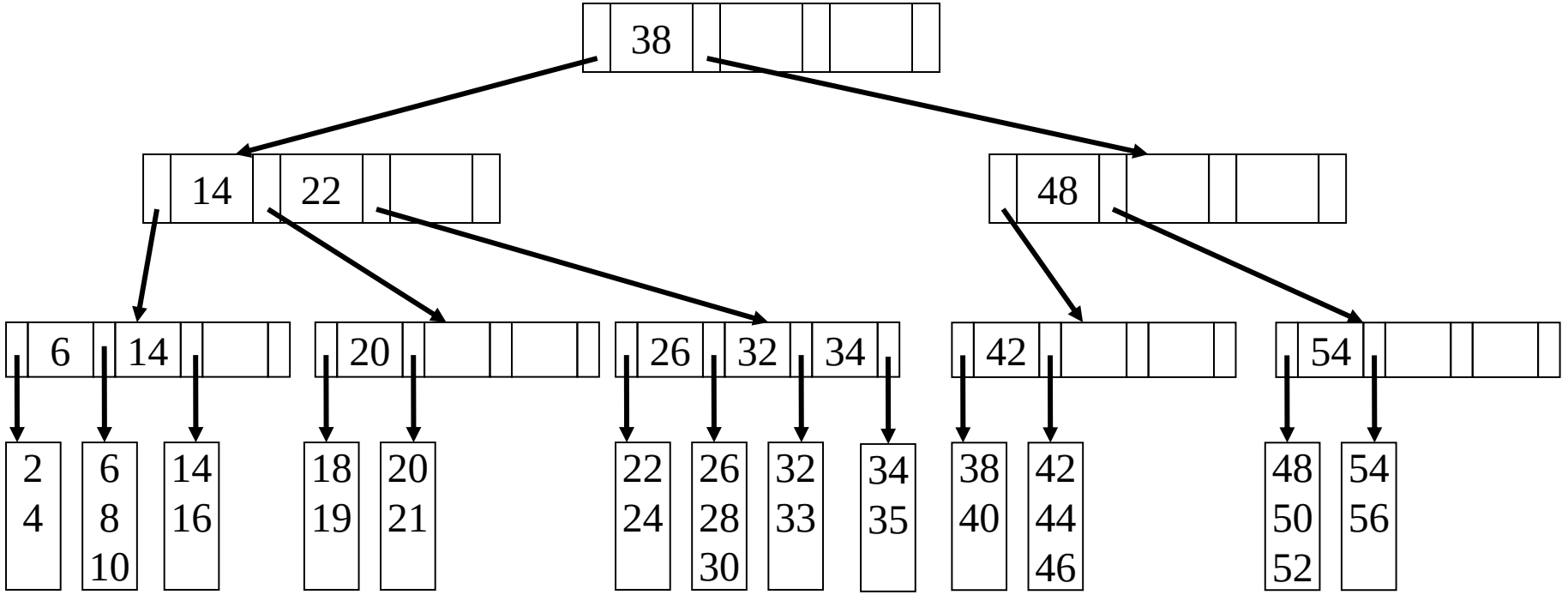
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Split the root as well
  - But now the parent of these needs to be split



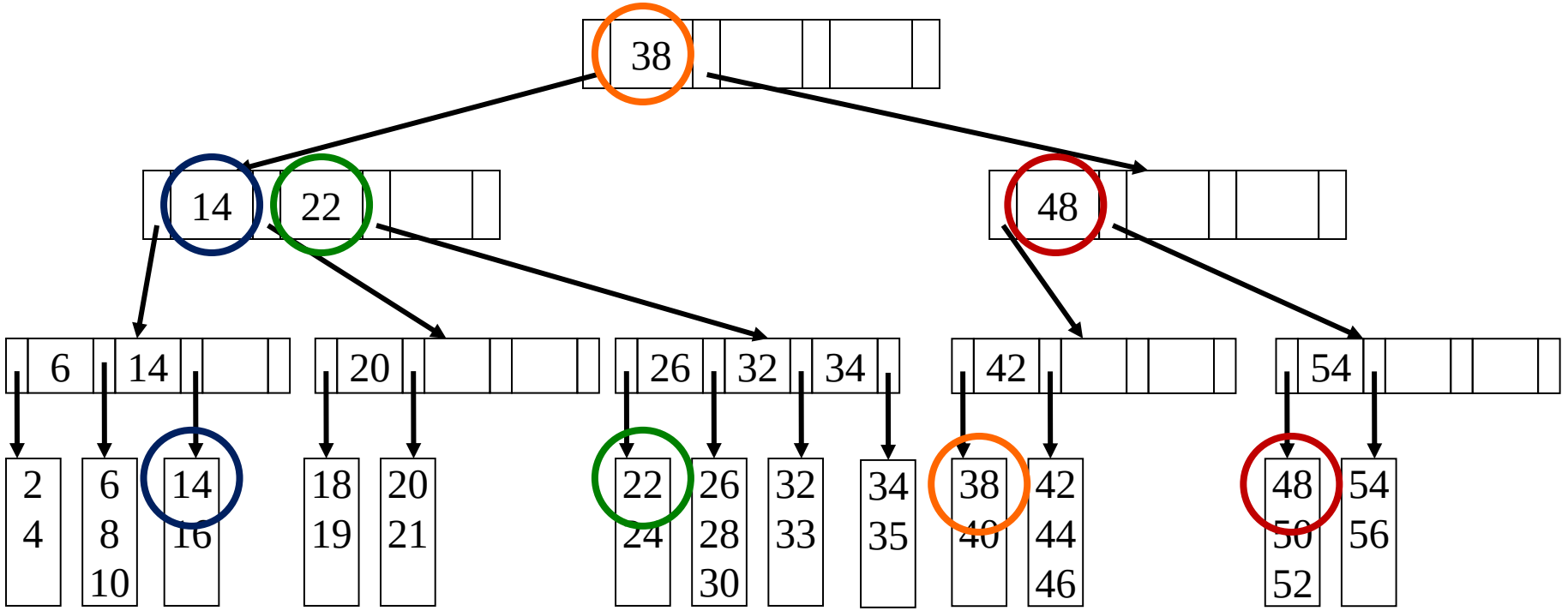
Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Keep splitting up the tree as needed



Insertion Example

- Insert 33, 35, and 21 into the tree below
  - Keep splitting up the tree as needed



B-Tree Operations
Deletion

B-Tree Deletion

- Find leaf containing element to be deleted
- If that leaf is still full enough (still has $\lceil L / 2 \rceil$ elements after remove) write it back to disk without that element
  - And change the key in the ancestor if necessary
- If leaf is now too empty (has less than $\lceil L / 2 \rceil$ elements after remove), take an element from a neighbor

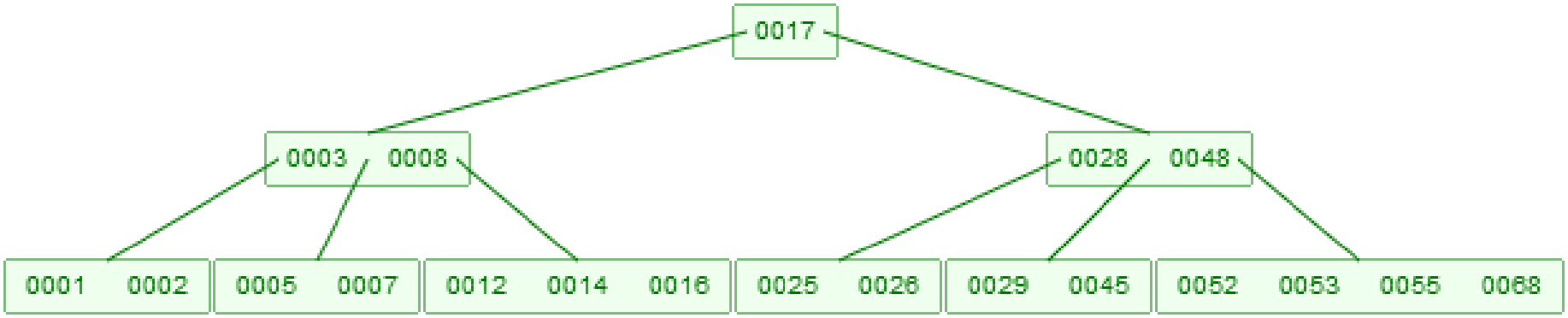
B-Tree Deletion

- When “taking” an element from a neighbor
- If neighbor would be too empty, combine two leaves into one
  - This combining requires updating the parent which may now have too few subtrees
  - If necessary, continue the combining up the tree
- Does it matter which neighbor we borrow from?

Other Implementations of B-Trees

Interior Nodes Store Data

- There are often multiple ways to implement a given data structure
- B-Trees can also be implemented where the interior nodes store data as well
  - The leaves can store much more, however



Effects on Data in Interior Nodes

- What kind of effect would this have on performance and implementation?
  - Does it change the way that insert works?
  - What about deletion? Is it simpler or does this change make it more complicated?
- Why would you choose one implementation over another?

Announcements

- Homework 4
  - Due Thursday, October 26th at 8:59:59 PM
- Project 3
  - Due Tuesday, October 31st at 8:59:59 PM

- Next Time:
  - Heaps