CMSC 341

B- Trees

D. Frey with apologies to
Tom Anastasio

Large Trees

- Tailored toward applications where tree doesn't fit in memory
  - operations much faster than disk accesses
  - want to limit levels of tree (because each new level requires a disk access)
  - keep root and top level in memory

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? The internal nodes just 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.

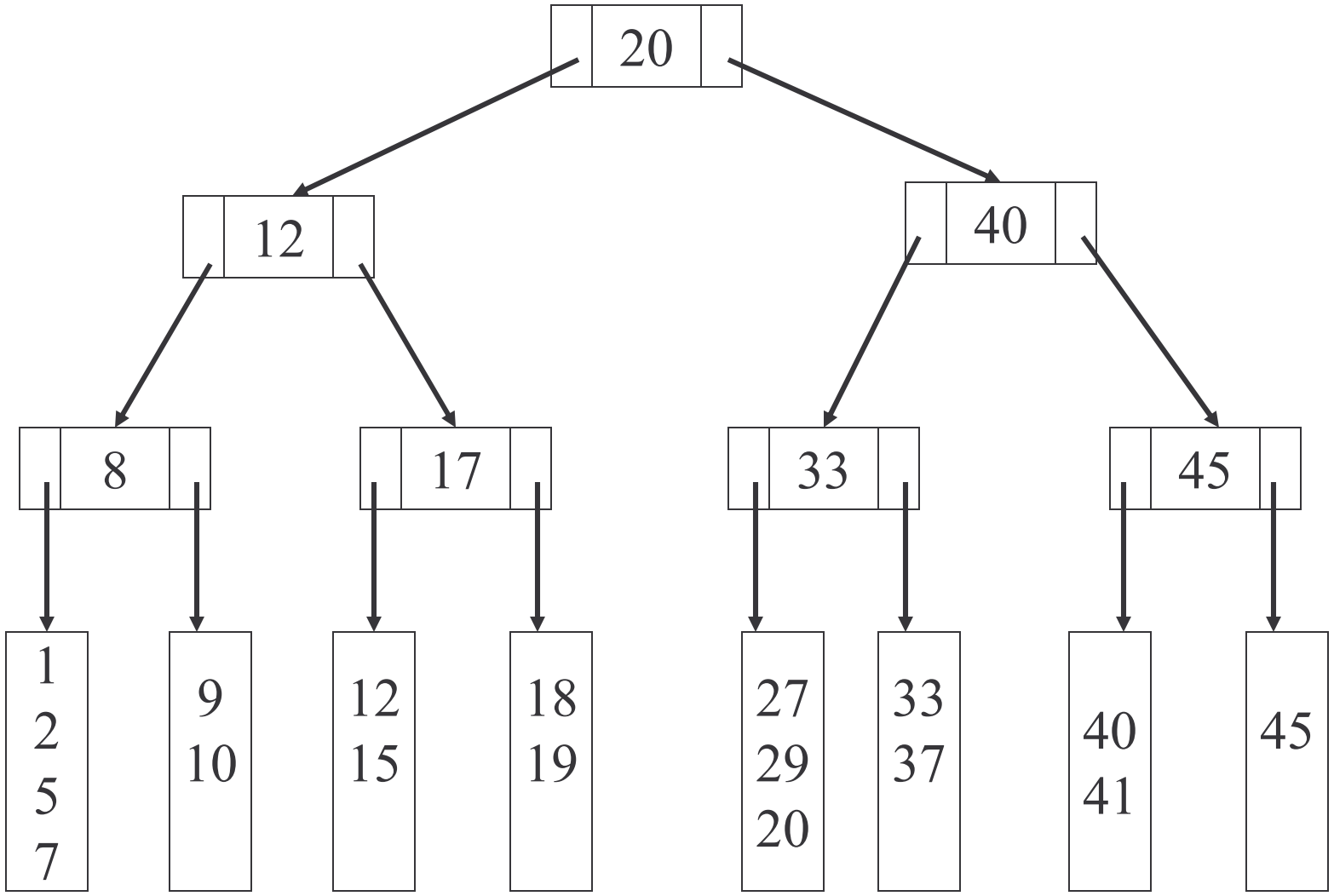


Figure 1 - A BST with data stored in the leaves

Observations

- 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: $\lceil \lg n \rceil$
    (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
    - the previous BST would be called a "2-way tree" or "M-way tree of order 2"
  - as M increases, height decreases: $\lceil \lg_M n \rceil$ (assuming one element per leaf)
  - A perfect M-way tree of height h has M^h leaves

An M-way tree of order 3

Figure 2 (next page) shows the same data as figure 1, stored in an M-way tree of order 3. In this example M = 3 and $h = 2$, so the tree can support 9 leaves, although it contains only 8.

One way to look at the reduced path length with increasing M is that the number of nodes to be visited in searching for a leaf is smaller for large M.

We'll see that when data is stored on the disk, each node visited requires a disk access, so reducing the nodes visited is essential.

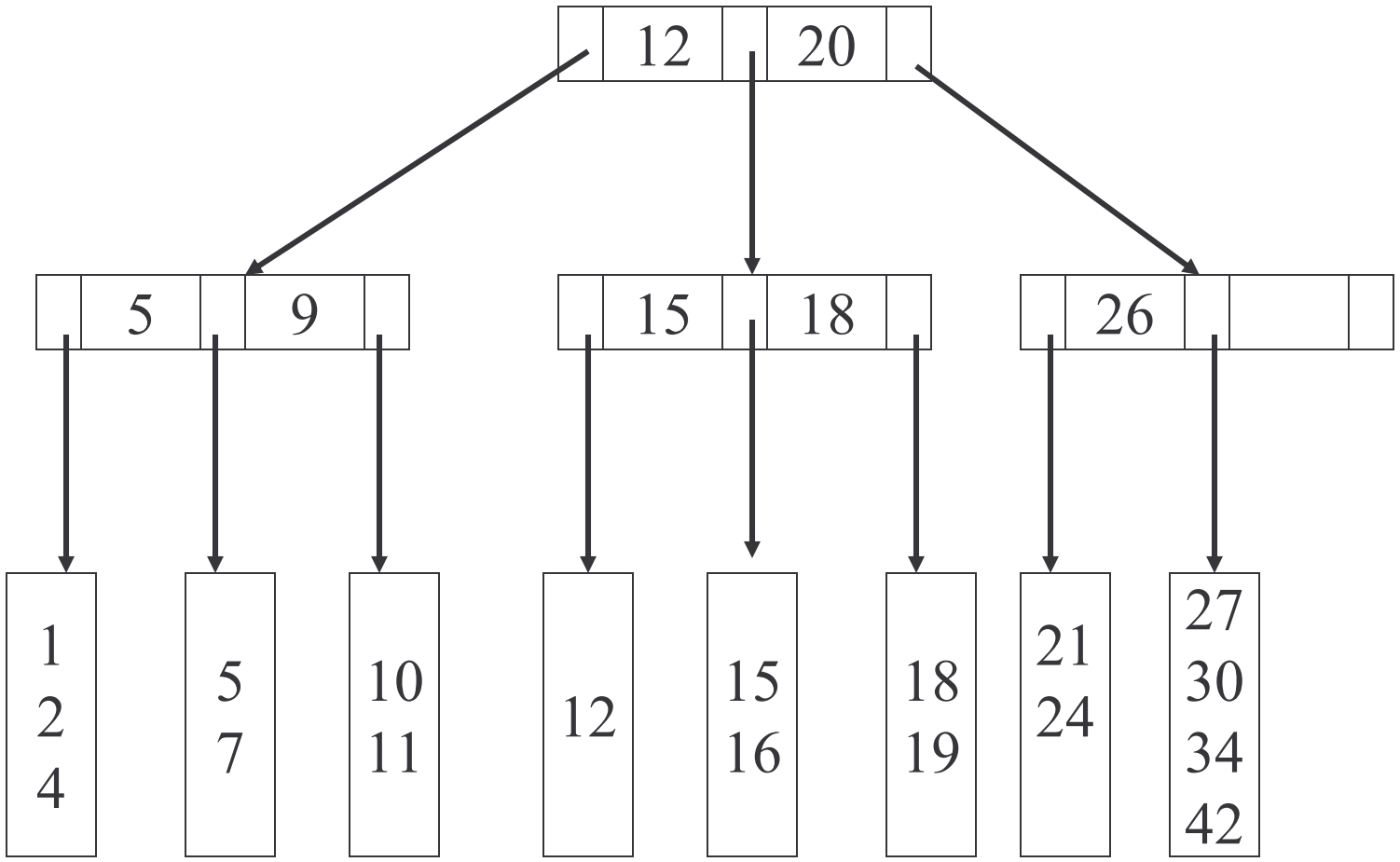


Figure 2 -- An M-Way tree of order 3

Searching in an M-way tree

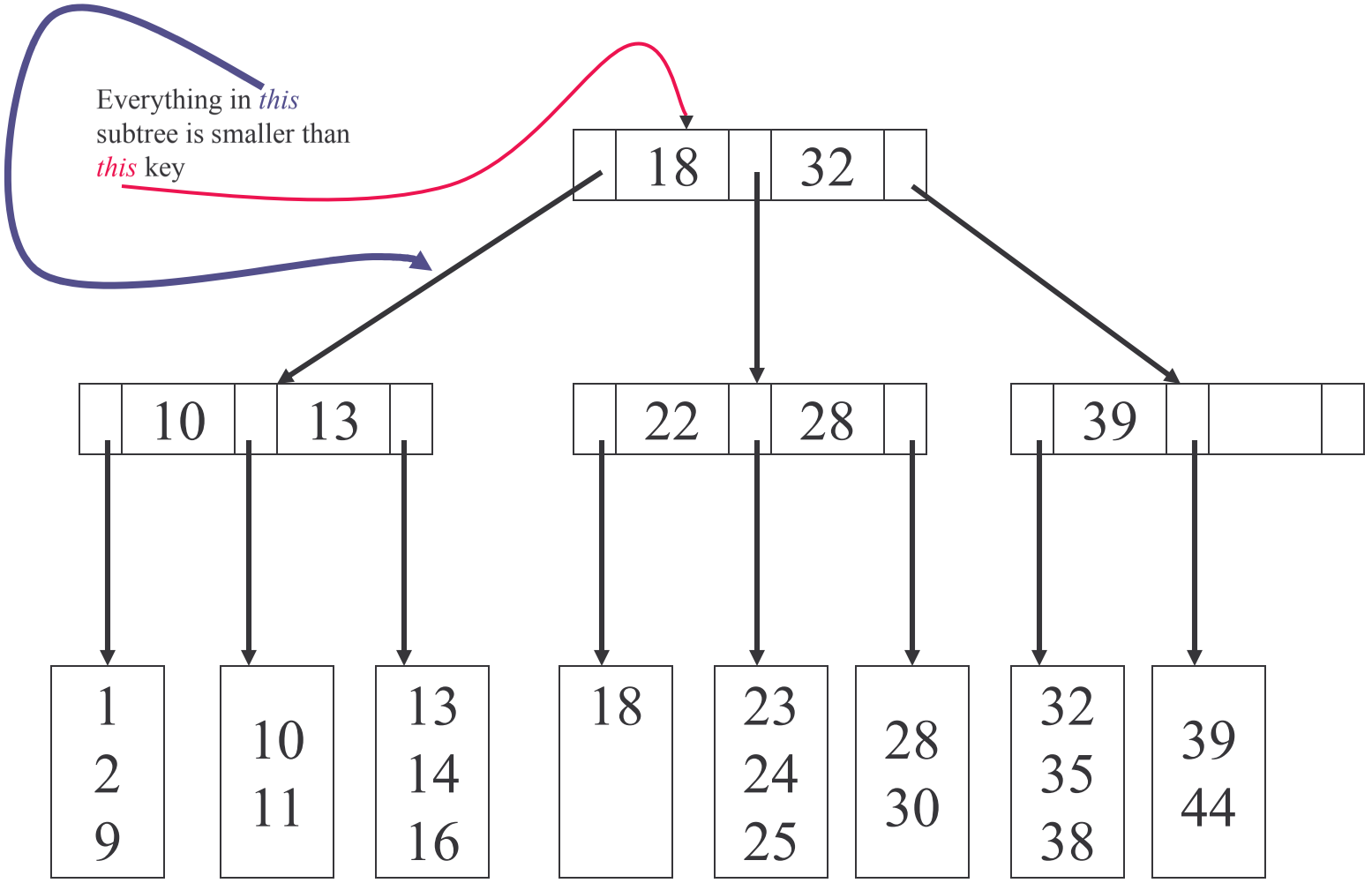
- Different from standard BST search
  - search always terminates at a leaf node
  - might need to scan more than one element at a leaf
  - might need to scan more than one key at an interior node
- Trade-offs
  - tree height decreases as M increases
  - computation at each node during search increases as M increases

Searching an M-way tree

```
Search (MWayNode *v, DataType element, bool& foundIt)
    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 which subtree element is in
            recursively search the subtree
```

- For "real" code, see Dr. Anastasio's postscript notes

Search Algorithm: Traversing the M-way Tree



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.

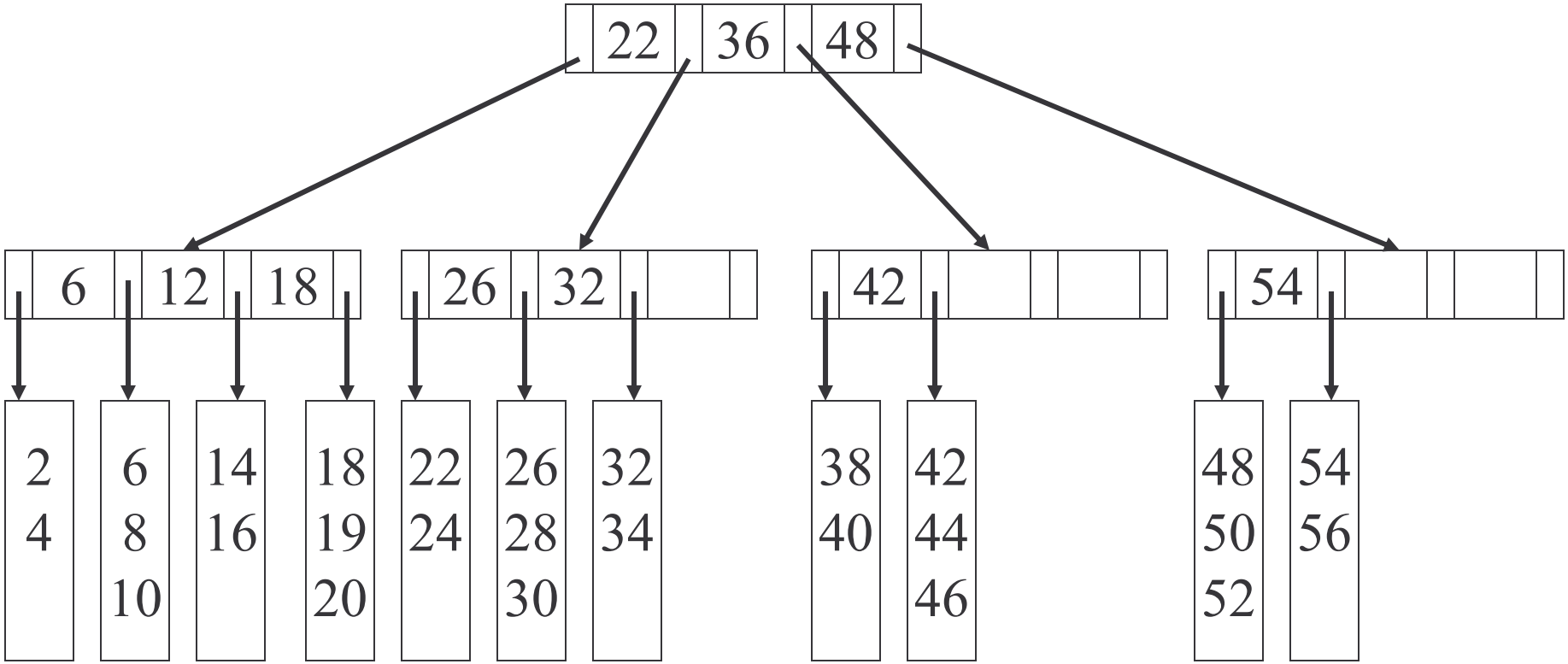


Figure 3 – searching in an M-way tree of order 4

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 items in a balanced BST and in an M-way tree of order 10.
- The height of the BST will be $\lg(10^7) \sim 24$.
- The height of the M-Way tree will be $\log(10^7) = 7$ (assuming that we store just 1 record per leaf)
- However, in the BST, just one comparison will be done at each interior node, but in the M-Way tree, 9 will be done (worst case)

How can this be worth the price?

- Only if it somehow 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 M-way tree to match the disk block and record size.

A generic M-Way Tree Node

```
template <class Ktype, class Dtype>
class MWayNode
{
    public:
        // constructors, destructor, accessors, mutators
    private:
        bool          isLeaf;       // true if node is a leaf
        int           m;            // the "order" of the node
        int           nKeys;        // nr of actual keys used
        Ktype         *keys;        // array of keys (size = m - 1)
        MWayNode *subtrees;         // array of pts (size = m)
        int           nElems;       // nr possible elements in leaf
        List<Dtype> data;           // data storage if leaf
};
```

B-Tree Definition

A B-Tree of order M is an M-Way tree with the following constraints

1. The root is either a leaf or has between 2 and M subtrees
2. All interior node (except maybe the root) have between $\lceil M / 2 \rceil$ and M subtrees (i.e. each interior node is at least "half full")
3. All leaves are at the same level. A leaf must store between $\lceil L / 2 \rceil$ and L data elements, where L is a fixed constant >= 1 (i.e. each leaf is at least "half full", except when the tree has fewer than L/2 elements)

A B-Tree example

- The following figure (also figure 3) shows a B-Tree with M = 4 and L = 3
- The root node can have between 2 and M=4 subtrees
- Each other interior node can have between $\lceil M / 2 \rceil = \lceil 4 / 2 \rceil = 2$ and M = 4 subtrees and up to M – 1 = 3 keys.
- Each exterior node (leaf) can hold between $\lceil L / 2 \rceil = \lceil 3 / 2 \rceil = 2$ and L = 3 data elements

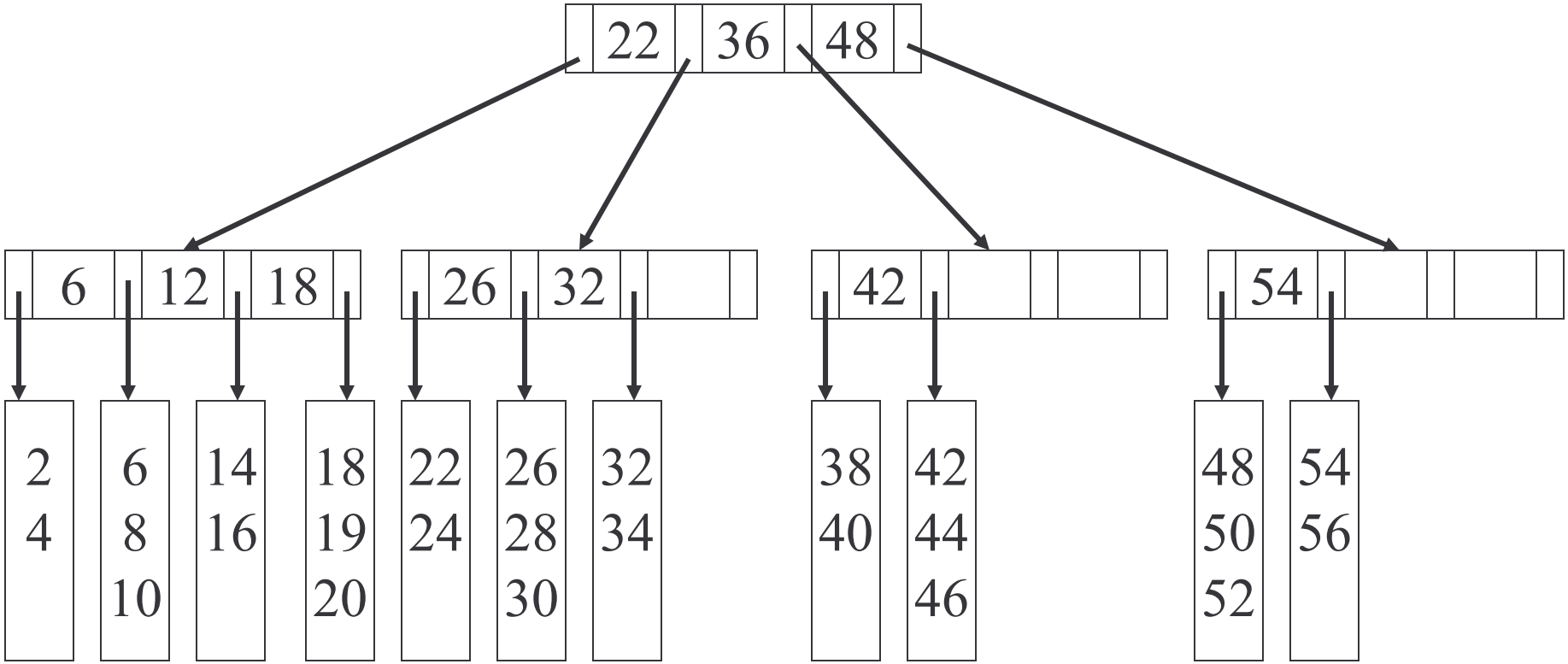


Figure 4 – A B-Tree with M = 4 and L = 3

Designing a B-Tree

- Recall that M-way trees (and therefore B-trees) are often used when there is too much data to fit in memory. Therefore each node and leaf access costs one disk access.
- When designing a B-Tree (choosing the values of M and L), we need to consider the size of the data stored in the leaves, the size of the keys and pointers stored in the interior nodes, and the size of a disk block.

Student Record Example

Suppose our B-Tree stores student records which contain name, address, etc. and other data totaling 1024 bytes.

Further assume that the key to each student record (ssn??) is 8 bytes long.

Assume also that a pointer (really a disk block number, not a memory address) requires 4 bytes

And finally, assume that our disk block is 4096 bytes

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 is 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 $L = \lfloor 4096 / 1024 \rfloor = 4$ data records per leaf.

Calculating M

Each interior node contains M pointers and M-1 keys. To maximize M (and therefore keep the tree flat and wide) and yet do just one disk access per interior node, we have the following relationship

$$4M + 8 (M - 1) <= 4096$$

$$12M <= 4104$$

$$M <= 342$$

So choose the largest possible M (making tree as shallow as possible) of 342.

Performance of our B-Tree

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 (about 10 times the size of UMBC student population) the height of the tree with M = 342 would be no more than 2, because $\lceil \log_{342}(25000) \rceil = 2$.

So any student record can be found in 3 disk accesses. If the root of the B-Tree is stored in memory, then only 2 disk accesses are needed .

Insertion of X in a B-Tree

- Search to find the leaf into which 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 is leaf full, split it into two leaves, each with half of elements. Insert X into the appropriate new leaf and write new leaves back to the disk.
  - Update the keys in the parent
  - If the parent node is already full, split it in the same manner
  - Splits may propagate all the way to the root, in which case, the root is split (this is how the tree grows in height)

Insert 33 into this B-Tree

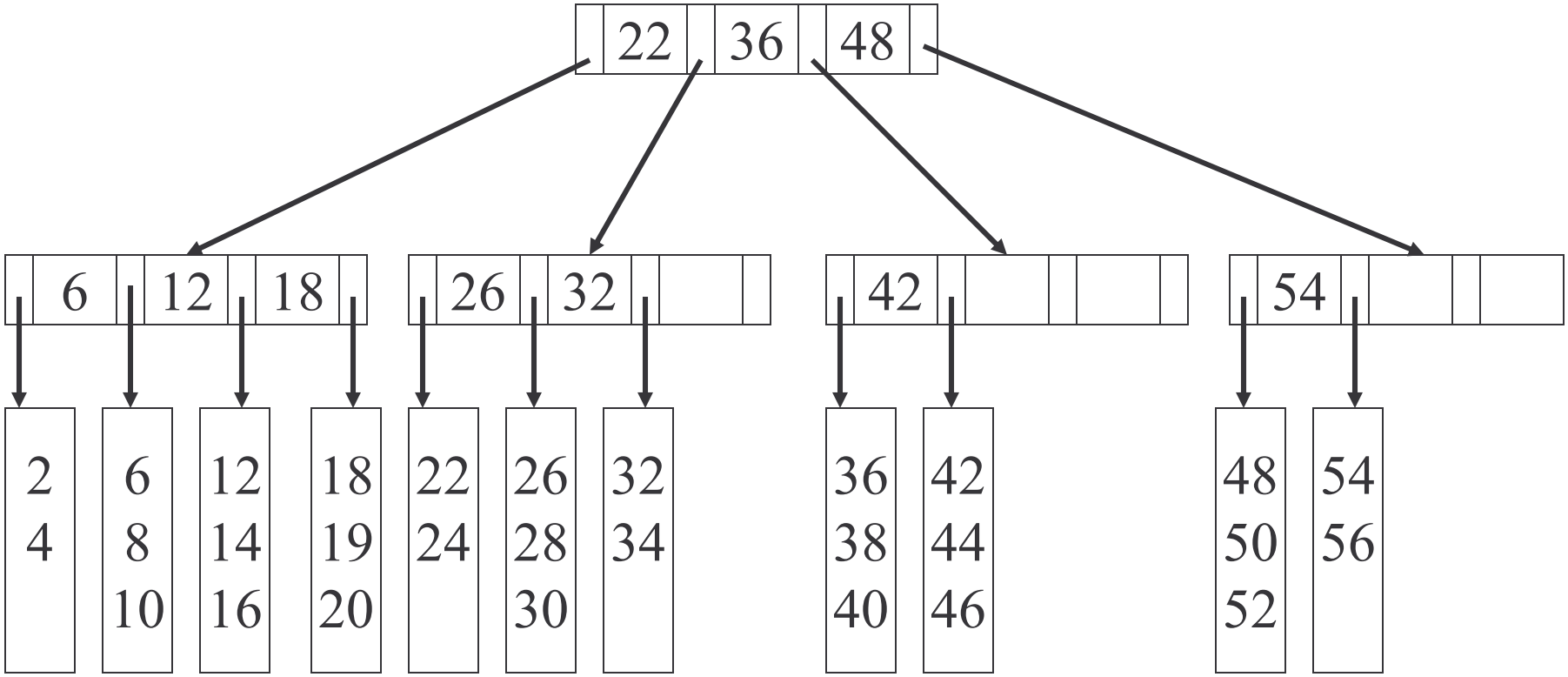


Figure 5 – before inserting 33

Inserting 33

- Traversing the tree from the root, we find that 33 is less than 36 and is greater than 22, leading us to the 2nd subtree. Since 33 is greater than 32 we are led to the 3rd leaf (the one containing 32 and 34).
- Since there is room for an additional data item in the leaf it is inserted (in sorted order which means reorganizing the leaf)

After inserting 33

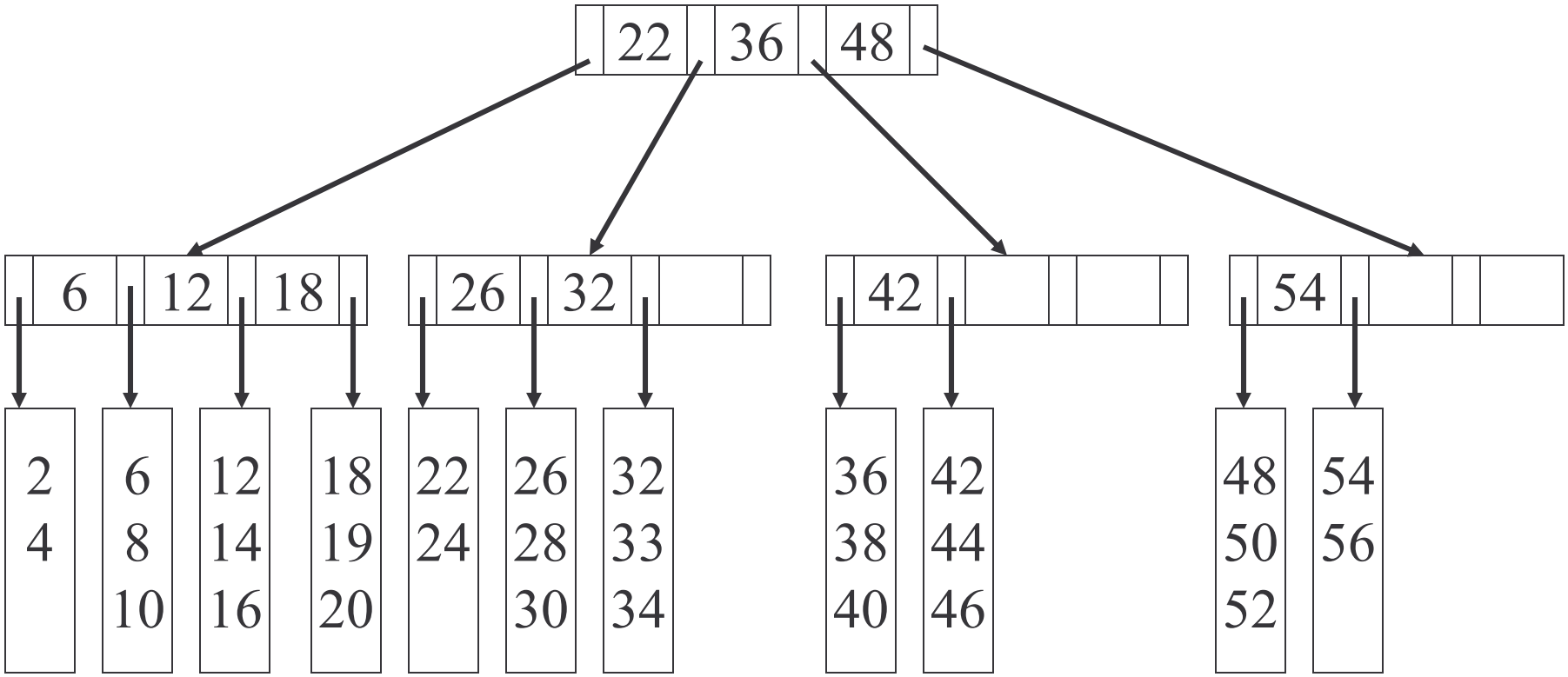


Figure 6 – after inserting 33

Now insert 35

- This item also belongs in the 3rd leaf of the 2nd subtree. However, that leaf is full.
- Split the leaf in two and update the parent to get the tree in figure 7.

After inserting 35

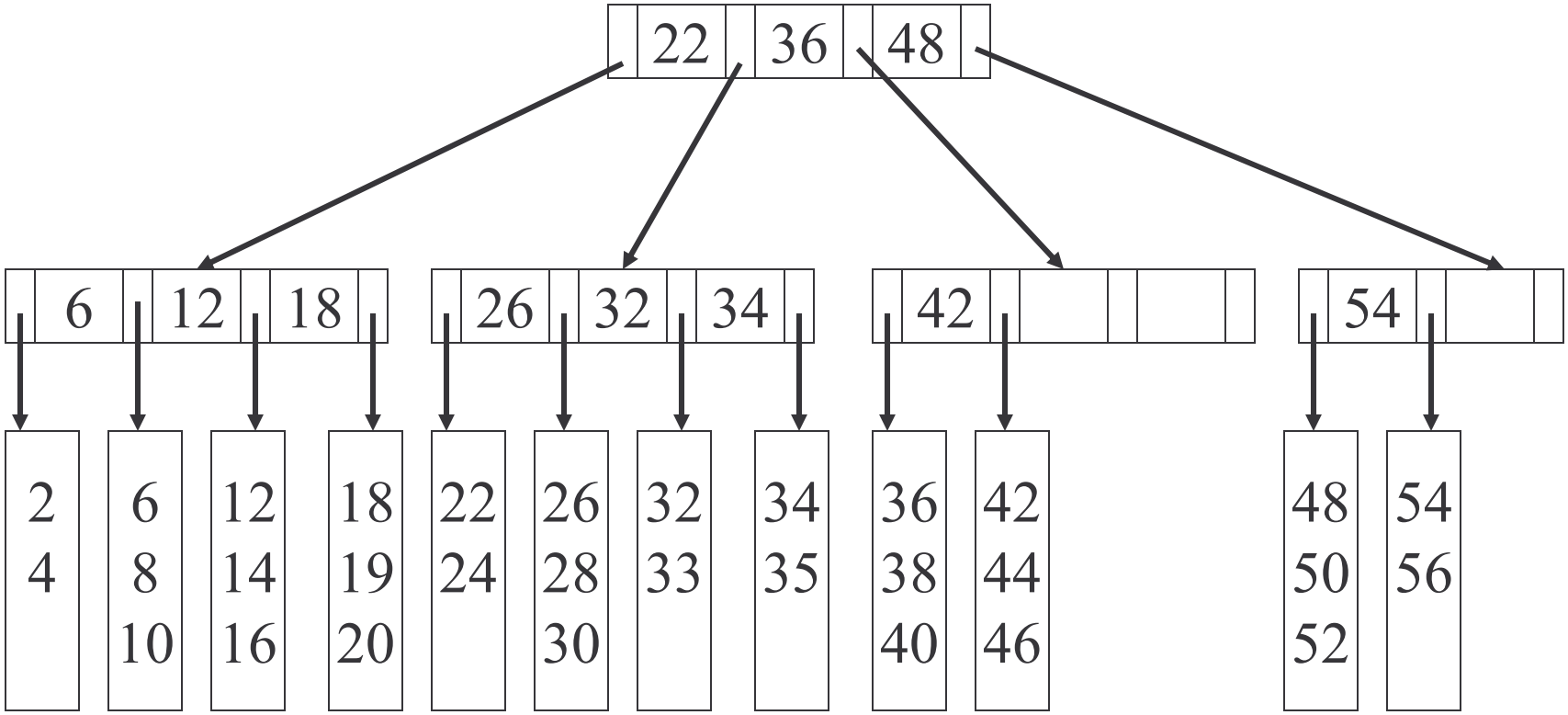


Figure 7 – after inserting 35

Inserting 21

- This item belongs in the 4^{th} leaf of the 1^{st} subtree (the leaf containing 18, 19, 20).
- Since the leaf is full, we split it and update the keys in the parent.
- However, the parent is also full, so it must be split and its parent (the root) updated.
- But this would give the root 5 subtrees which is not allowed, so the root must also be split.
- This is the only way the tree grows in height

After inserting 21

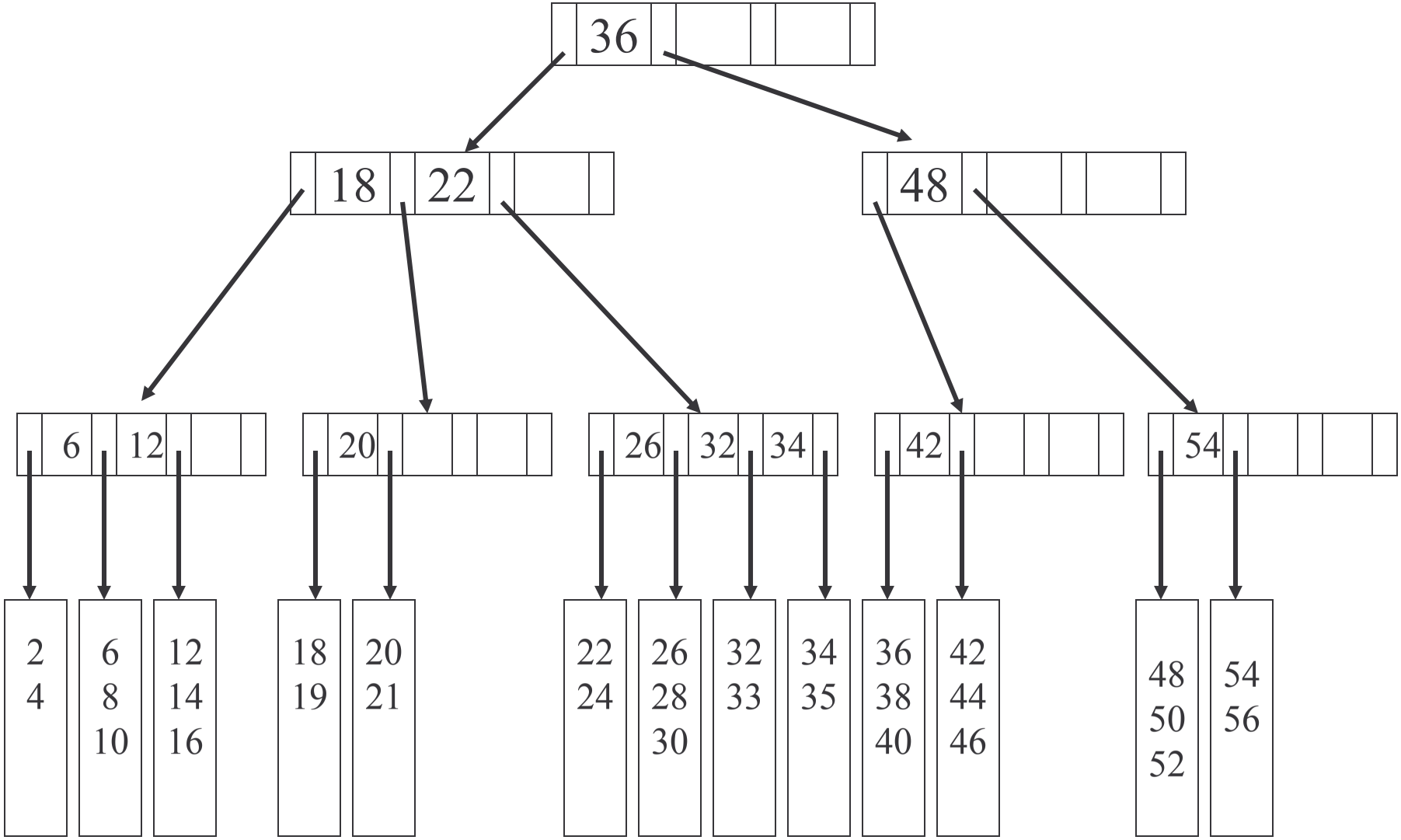


Figure 8 – after inserting 21

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. Then change the key in the ancestor if necessary.
- If leaf is now too empty (has less than $\lceil L / 2 \rceil$ elements), borrow 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