COSC160: Data Structures
B-Trees

Jeremy Bolton, PhD
Assistant Teaching Professor

GEORGETOWN UNIVERSITY

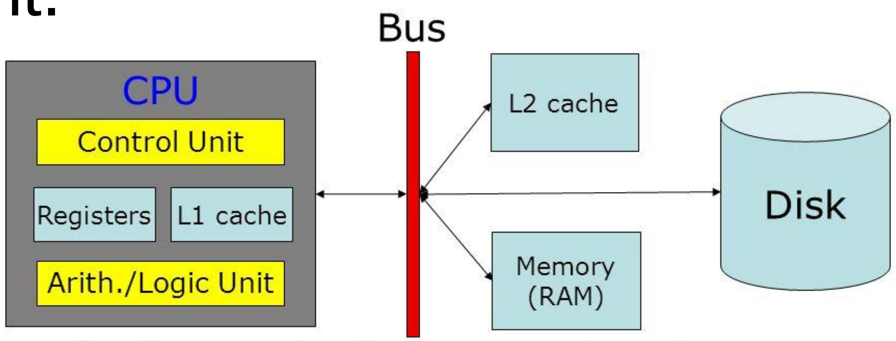
Outline

- I. B-Trees
  - I. Motivation
    - I. Memory hierarchy
  - II. m-way trees
  - III. B-Trees
    - I. Insert / Split
    - II. Remove / Merge
  - IV. Family of B-Trees

Depth, Time Complexity, and Disk Access

- Logarithmic time complexity is quite efficient.
  - Can we improve efficiency further?
  - How?

Computer Architecture



- Von Neumann described a memory hierarchy
  - Fast-----------------------------------> slow
  - $$$------------------------------------> cheap

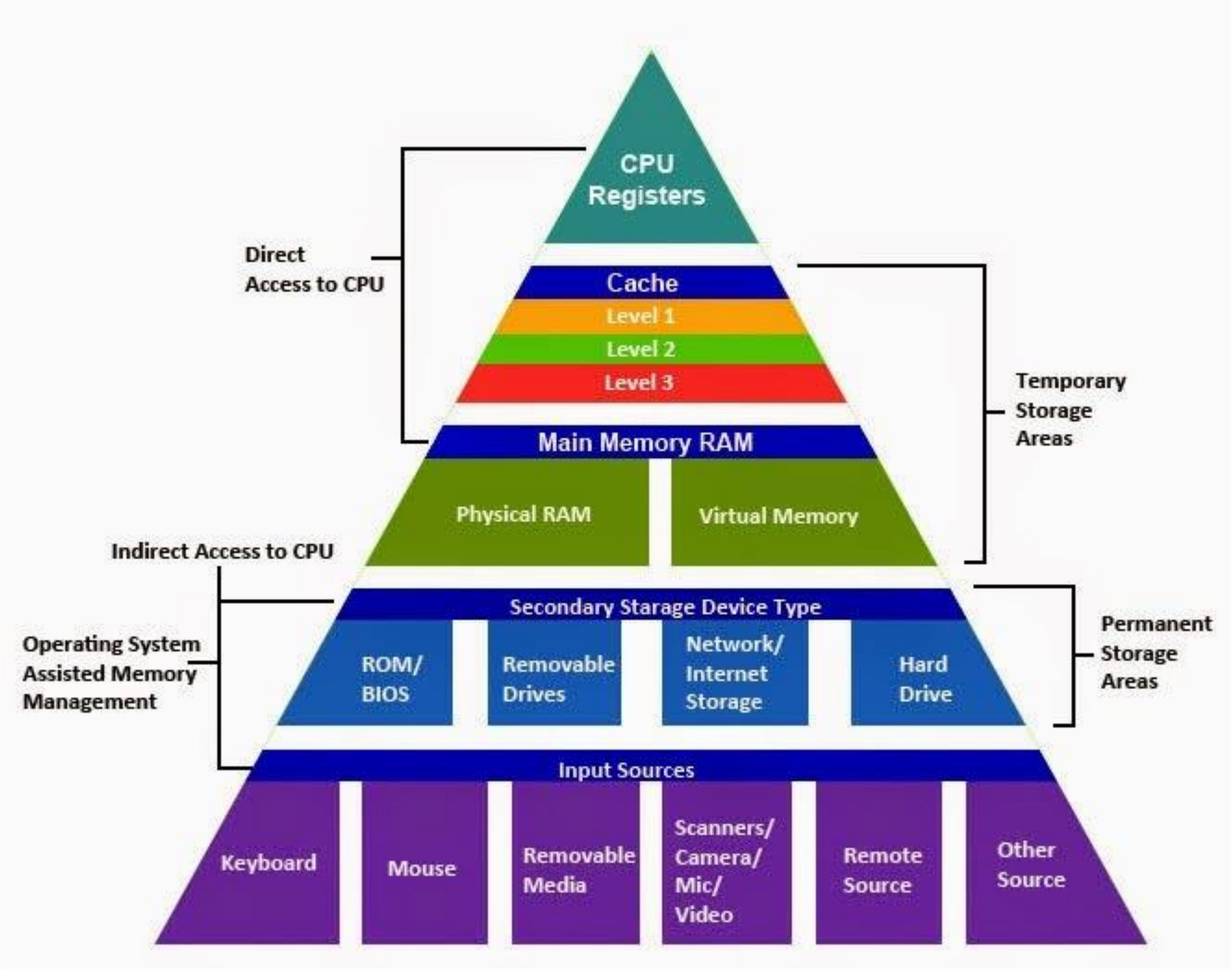
- Motivation
  - Practical Improvements: Disk, Memory and Cache delays
    - Useful for databases.
    - Linear search (with no memory delays) may be better than a logarithmic search with delays
    - Von Neuman Bottleneck: accessing low levels of memory hierarchy is slow
  - How can we reduce the computational steps associated with a search tree?
    - Reduce its height?

Memory Penalties

- Memory is hierarchical to mitigate the effects of the Von Neumann bottleneck

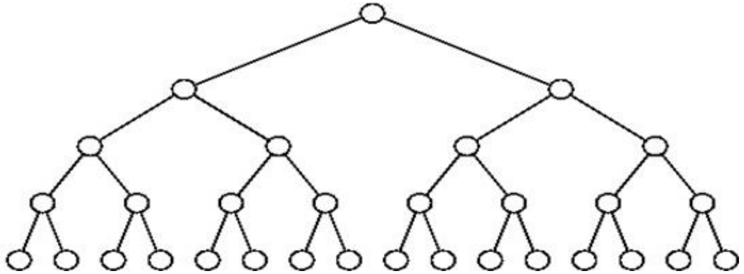
- However: Accessing secondary memory still incurs a significant penalty (delay!)

- This penalty is high enough to offset many orders of magnitude of theoretical time complexity reduction

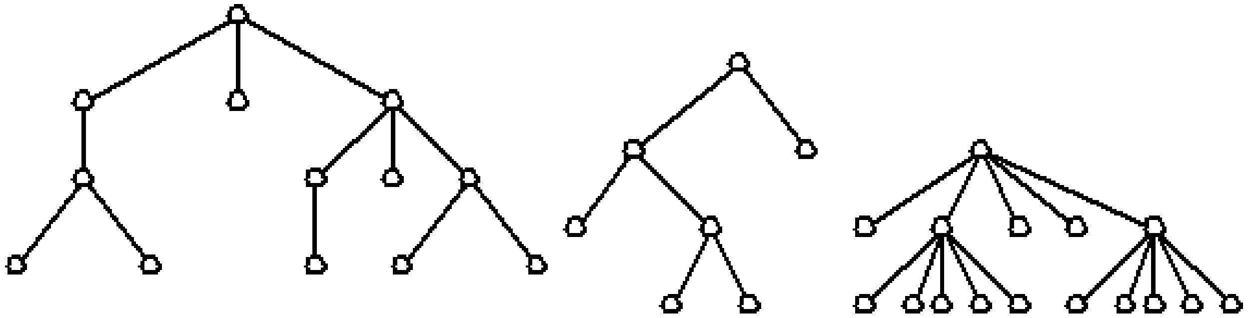


GEORGETOWN UNIVERSITY

m-way Search Tree



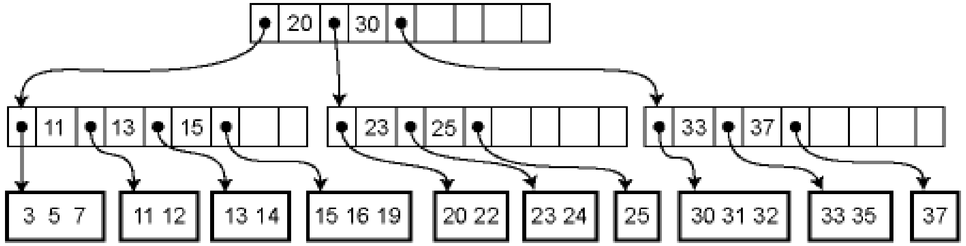
- Observations of multiple branches
  - Height
    - Height of balanced binary (2-way) tree is $O(\log_2(n))$
    - Height of balanced m-way tree is $O(\log_m(n))$
  - As m increases the height of tree decreases. “Less” node traversals.
  - However this design allows (and may require) multiple keys (m-1) stored at each node. The search time at each node increases



m-way Search Tree

- Contiguous vs Non-Contiguous storage:
  - Each time a nodes keys are accessed, they are loaded from memory
  - THUS Reduce the height of the tree, reduce memory accesses

- Notes:
  - But multiple keys in each node will increase searching time over keys in each node
    - If keys in each node are stored contiguously, this is likely done with only 1 access to memory chuck.
  - Crux: May not reduce step count but may reduce the total number of disk access (where disk access might have a memory delay).



Using keys to represent large data file

- A Key should uniquely identify a data entry
  - Example database entries
    - Data entry may be a tuple (ssn, first name, last name, image, …)
    - **Key** should uniquely identify each data entry, e.g. ssn
- Keys should be “light weight” and stored contiguously in node structure.
- Bulky data can then be accessed by pointers associated with each key
  - We neglect these pointers in our conceptual introduction here, but they will be necessary for a real-world implementation.

GEORGETOWN UNIVERSITY

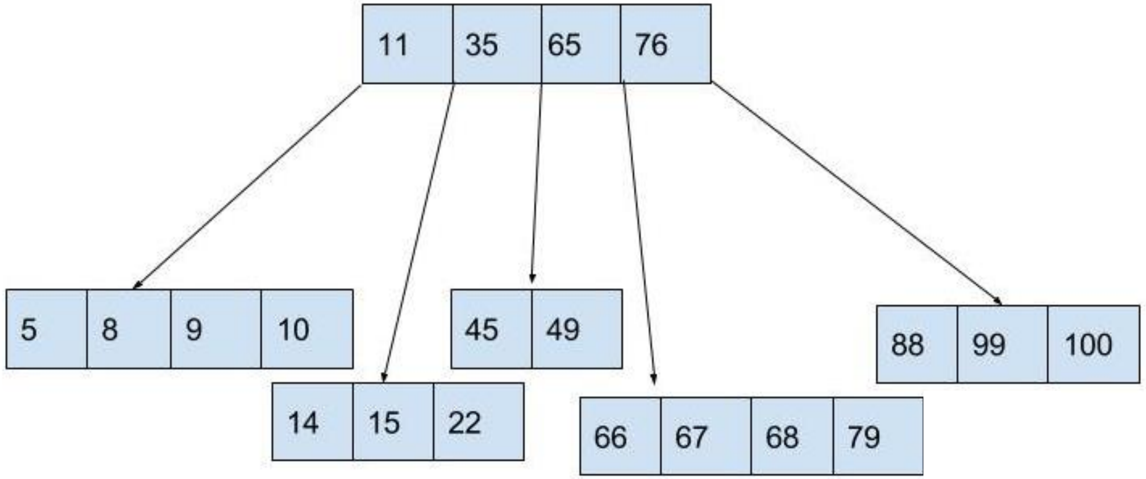
Operations on m-way Search Tree

- Similar to BST, but may have m children at most.

- m-way or m-ary tree
  - each node has up to m children and m-1 keys
  - keys are in some order
  - All keys within first i children are less than the i^{th} key
  - All keys within last m-i children are greater than the i^{th} key

- Operations:
  - Search
  - Insert
  - Remove

Example: 5-way tree



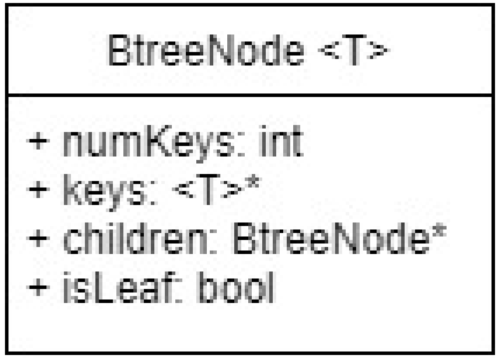
B-Trees

- Definition:
  - A ***b-tree*** of ***order m*** is a m-ary ($m \geq 4$) search tree, with the following properties.
    - The root has between 2 and m children (unless it is a leaf)
    - All non-leaf nodes (except the root) have between ⌈m/2⌉ and m children
    - All non-root nodes contain k-1 keys and k pointers to children where ⌈m/2⌉ $\leq k \leq m$
    - All leaf nodes are the same depth

- Strict balance constraint.
  - Note: The depth of all leaf nodes are the same
    - How is this maintained? Height only changes by adding or removing root.
  - Rotations (reorganization) “similar” to AVL but a bit more complex

B-Tree Nodes

- Each node in a B-Tree of order m has the following information.
  1. Up to m-1 keys
  2. The number of current keys stored
  3. m pointers to children
  4. isLeaf: is the node a leaf node



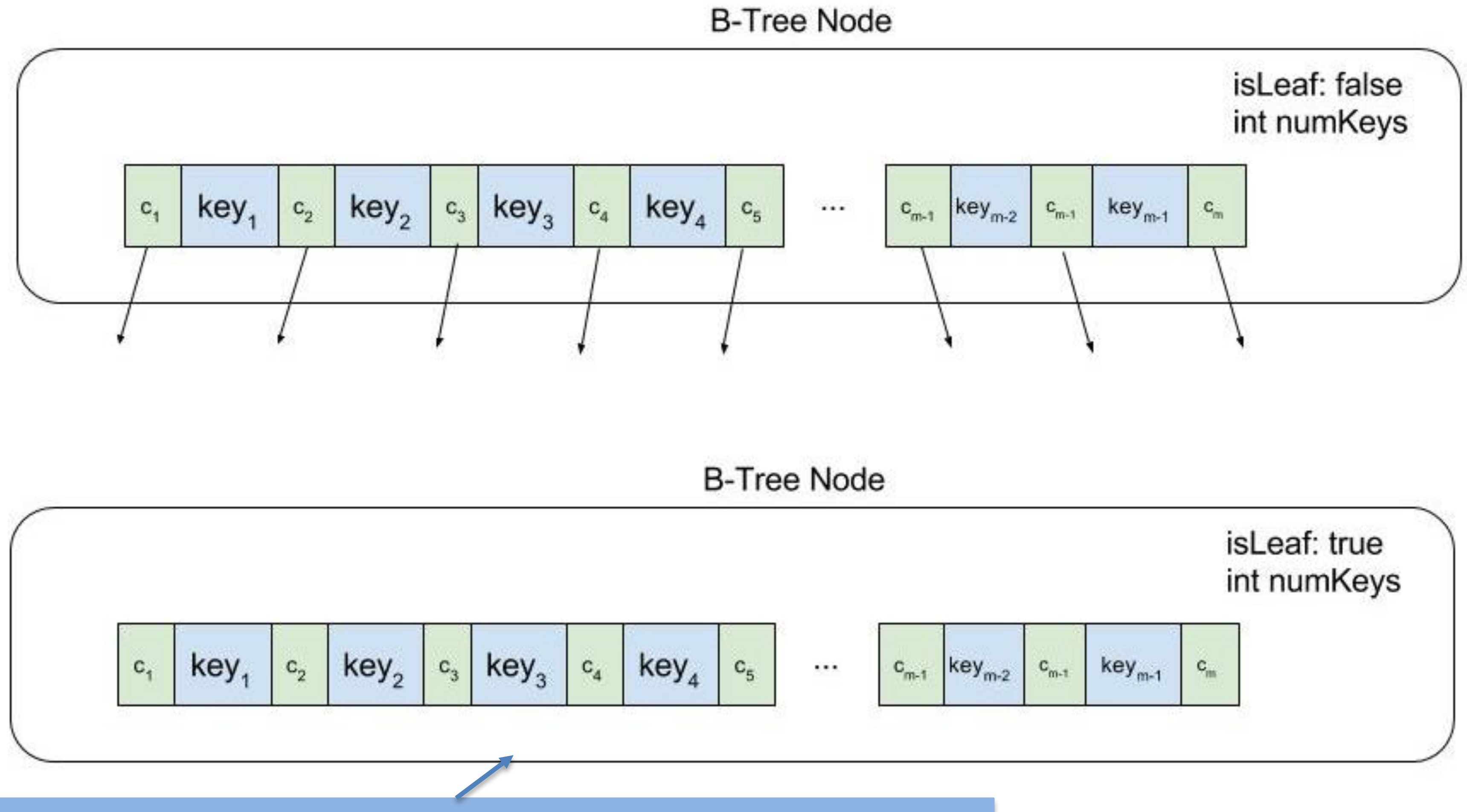
BtreeNode <T>

+ numKeys: int
+ keys: <T>*
+ children: BtreeNode*
+ isLeaf: bool

- Even given these standard constraints, there are some variations and design decisions to make. We will discuss some later.
  – Family of B-Trees

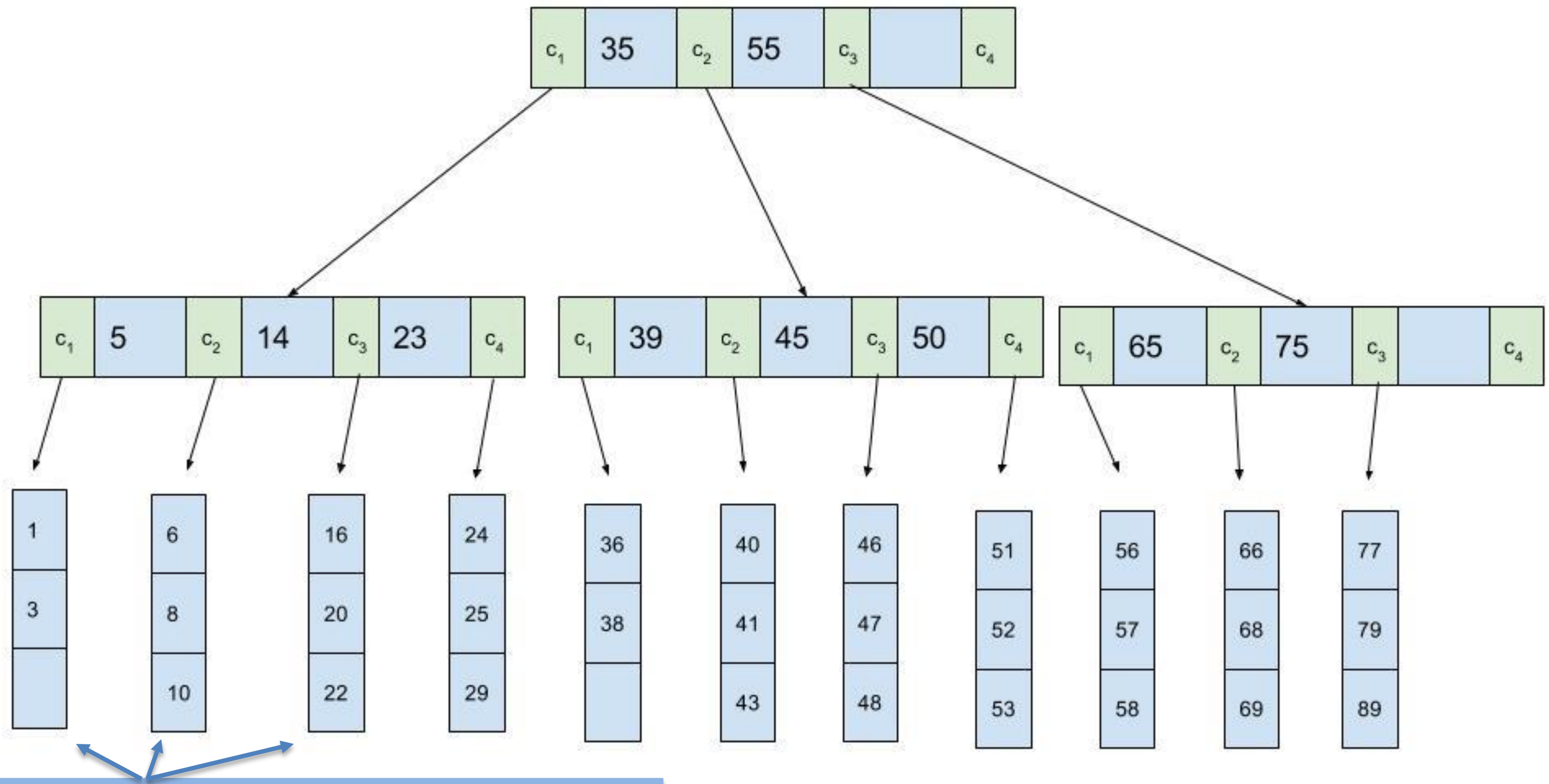
B-Tree Node Illustration

- Each node contains children array and key array
- Intuitively these two arrays are often illustrated in an interleaved fashion. *Keep in mind these are two distinct arrays in implementation!!*



Leaf Nodes: the children array can be absent or all values are set to NULL

Example: 4-way B-Tree



Leaf Nodes: the children arrays are absent in this example

Searching in a B-Tree

```
// assumes subtrtree rooted at root, searching for key value k.
// returns pointer to “data object” specified by key k
// assumes design 2
$function\ searchBtree(root, k)$
$i := 1$
$while\ i \leq root.numChildren\ \ AND\ k > root.keys[i]$
    $i := i + 1$
$if\ i \leq root.numChildren\ AND\ k == root.keys[i]$
    $return\ root.data[i]$ // base case: found it
$if\ root\ is\ a\ leaf$
    $return\ NULL$ // base case: did not find it!
$else$ // recursive case: keep traversing down
    $return\ searchBtree(root.child[i], k)$
```

Complexity Analysis (worst case)

Traversing down the tree $\Theta(h)$ = $\Theta(\log_m n)$

At each node in the tree, sequentially search keys:$\Theta(m)$

Total computational steps: $\Theta(m \log_m n)$

Searching B-Tree Complexity

- Practical Notes:
  - Not a theoretical improvement $O(m \log_m n)$
  - Pro: number of disk accesses is reduced!
    - $\Theta(\log_m n)$
- Can we improve upon this?
  - Try binary search on m-1 keys at each node
    - Given the overhead, we may not gain much here

Complexity Analysis (assume m = L)

Traversing down the tree $\Theta(h)$ = $\Theta(\log_m n)$

At each node in the tree, sequentially search keys: $O(m)$

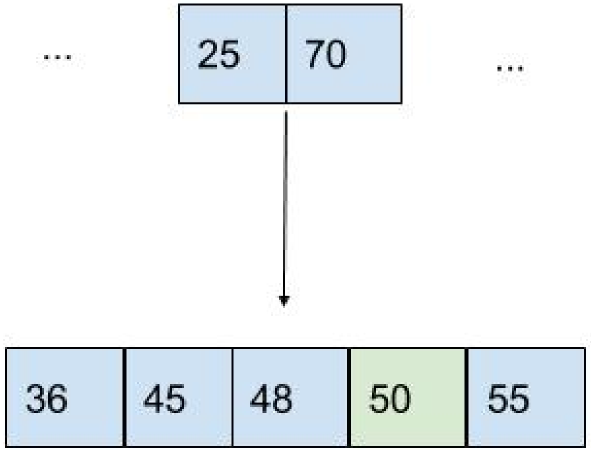
Total computational steps: $O(m \log_m n)$

Inserting a key into a B-Tree

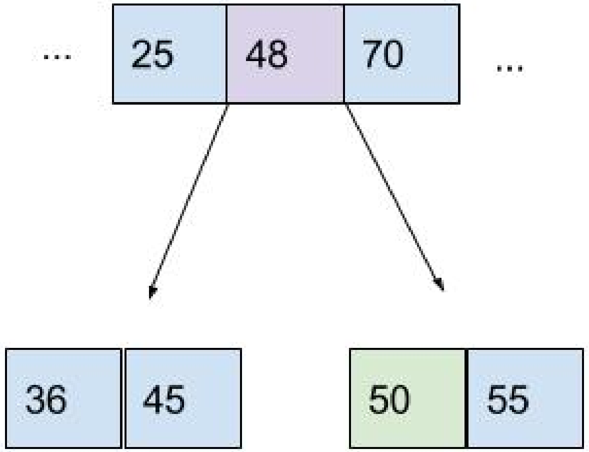
- Insertion is always done at a leaf node
- Insertion is done in a single pass down the tree
- Uses ***splitChildBtree*** function
  - This function assures b-tree constraints are not violated.
  - During traversal to leaf node for insertion, all nodes encountered that have a maximum number of keys are split (otherwise a violation may occur).

Inserting into a B-Tree

- Inserting into a B-Tree is not simple
- If a node's keys are full, ***then the node is split into two nodes***
  - Splitting around the median key value is intuitive
  - Median value is moved to parents key list
  - Must maintain a valid B-Tree after the split
    - NOTE: promotion of child median value to parent may cause parent to have too many keys!



Assume m = 5
Max num keys is 4
Assume we insert 50, thus going over max
1) Split child node at median and
2) "promote" median value to parent
3) Update children pointers

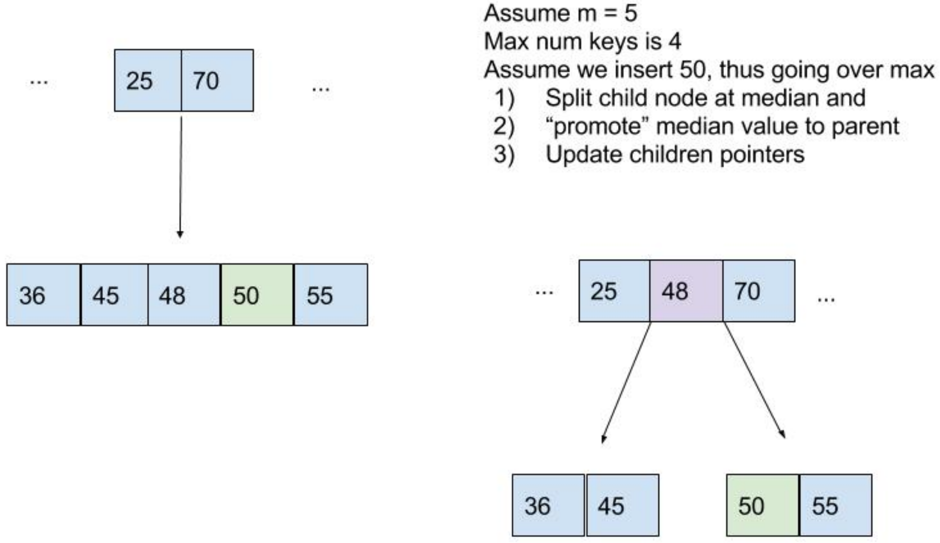


Splitting a node (High-Level)

```
// parent is a "non-full" internal node
// child is a "full" child of root
// i is index into key

function splitChildBtree(parent, child, i)
1.  Create new node
2.  Identify median entry in child.keys
3.  Copy right half (right of median) of  key values into new node
4.  Copy  right half of children pointers to new node
5.  Update child.numKeys and newNode.numKeys
      –  account for median removal
6.  Promote Median to Parent
      –  Make room for and insert median value into parent.key
      –  Make room for and insert pointer for newNode in parent.child
      –  Update parent.numKey
```

Inserting: Inserting into a non-full node
High-Level

```
// assumes is leaf
$function\ insertNonFullBtree(node, k)$

If node is leaf // perform insert

    1. Find correct index i for insertion of k into node.key[i]
    2. Insert k into node.key[i]
    3. Update node.numKeys

$else$ // continue down to leaf and confirm internal nodes are not full!

    1. Find correct index i for traversal node.children[i]
    2. if node.child[i] is full
           $splitChildBtree(node, node.children[i], i)$
           update i as needed, if split occurred
    3. $insertNonFullBtree(node.children[i], k)$
```



Assume m = 5
Max num keys is 4
Assume we insert 50, thus going over max
1) Split child node at median and
2) “promote” median value to parent
3) Update children pointers

Inserting into a B-Tree High-Level

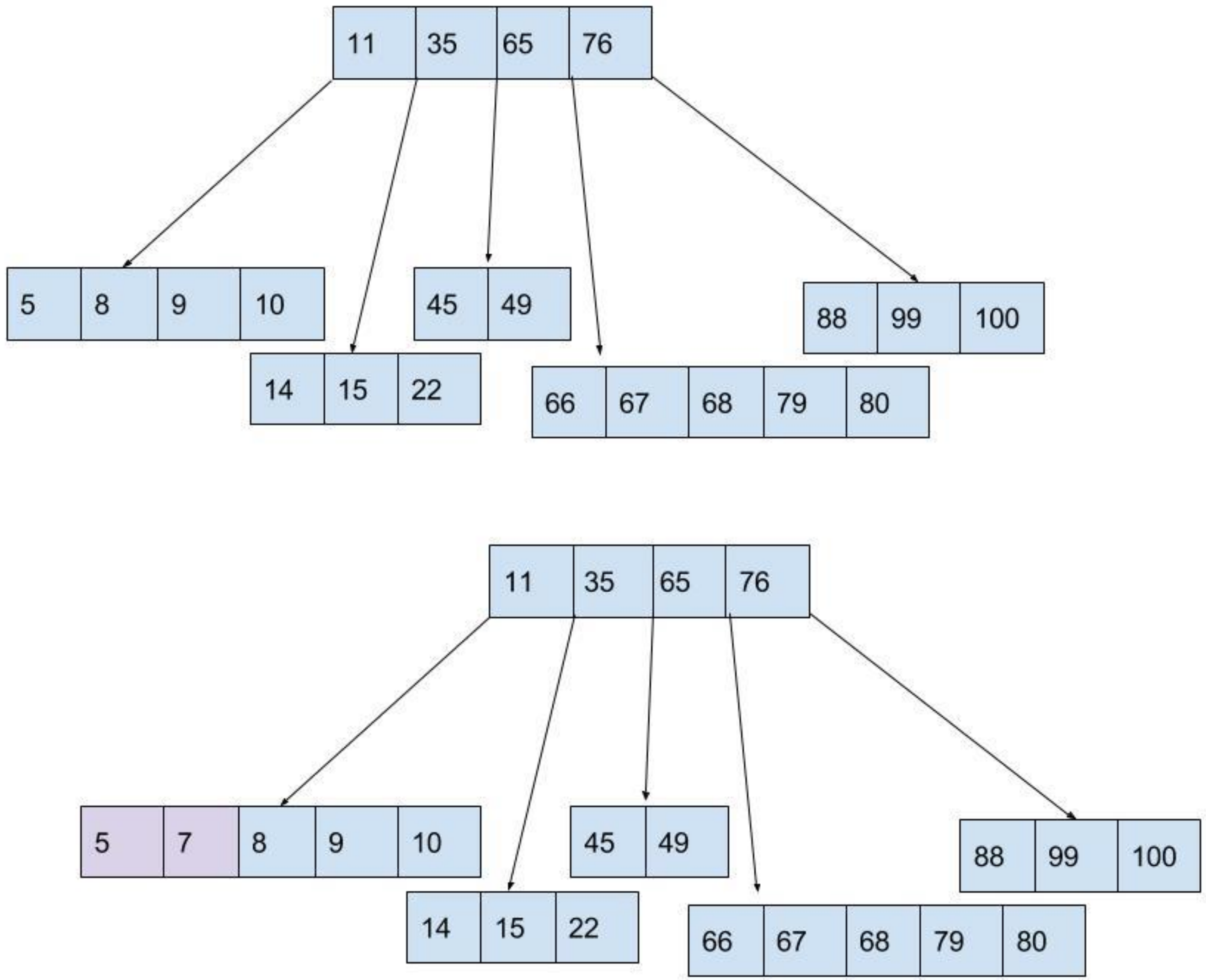
```
// The only reason we need this “wrapper” function is to account for the
// case where the root is full! The main work is being done by the helper
// methods previously defined

$function\ insertBtree(T, k)$
$oldRoot \coloneqq Tree.root$
// if root is full, we must create a new root and increase tree depth by 1
$if\ root\ is\ full$
1.      Create newRoot
2.      Make oldRoot the first child of newRoot
3.      $splitChildBtree(newRoot, root, 1)$
        // the old root is full, need to split it before we can insert
        $insertNonFullBtree(newRoot, k)$
$else$ // if root is not full, we can use standard insert
        $insertNonFullBtree(root, k)$
```

Insert Examples

- B-tree, m = 6

  max keys is 5

- Example Insert 7



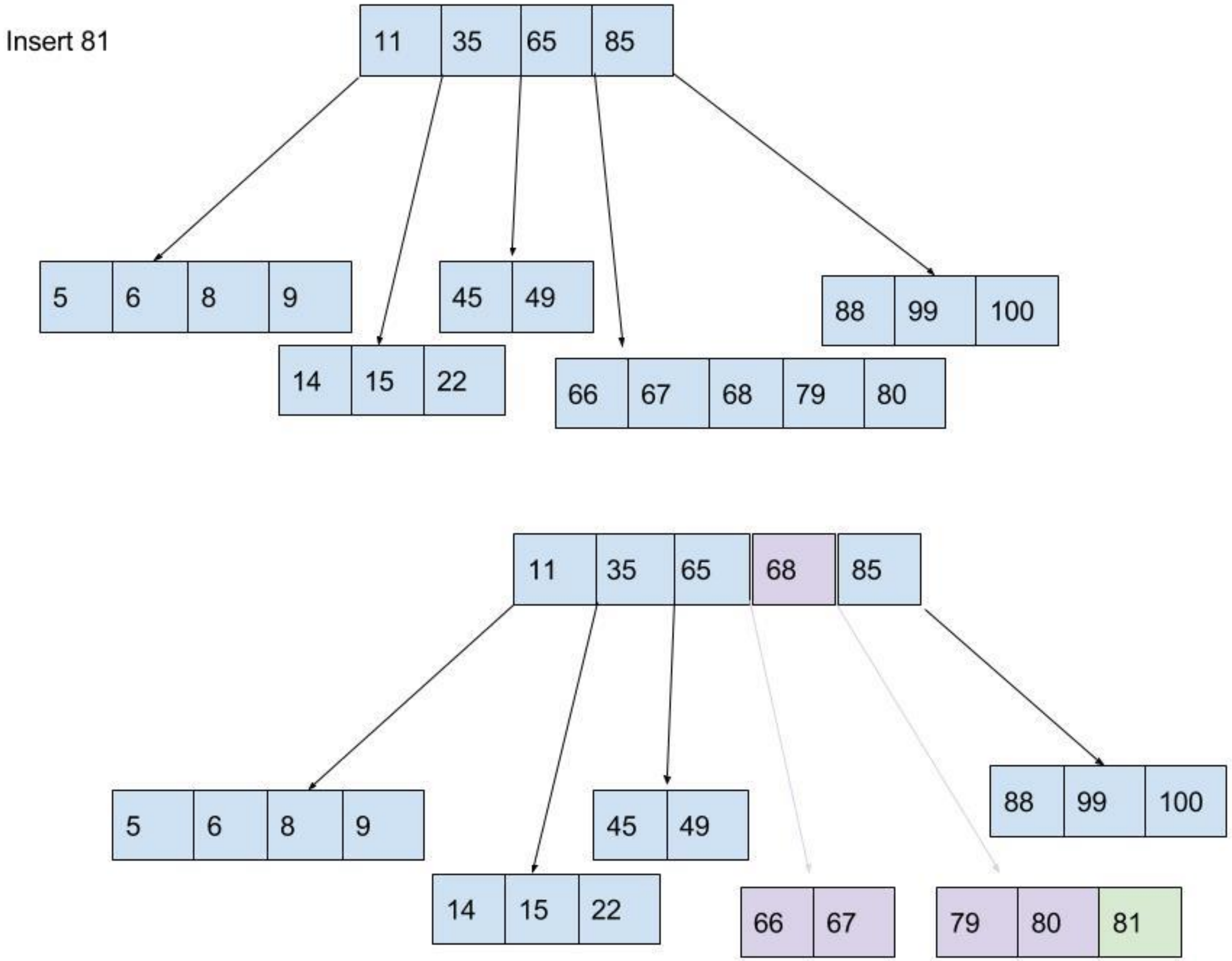
Insert Example

- B-tree, m = 6

  max keys is 5

- Insert into full node.

- Run split node procedure



Deleting a Key from a B-tree

- Similar to insertion, but a few more cases to consider
- Single pass down the tree, key to be deleted is “moved” to the leaf and deletion occurs at the leaf
- If key is deleted from internal node, then there are a few cases of concern.
  - Is resulting b-tree valid
  - Constraint: All non-leaf nodes (except the root) have between m/2 and m children

Deleting a Key from a B-tree

- During traversal for deletion there are 3 Cases
  - General idea:
  - Traverse down the tree in search of key k, **at each node identify the case and proceed appropriately**
    - If a node with a min number of keys is encountered, we will “adjust” keys so that the number is not min. (case 3)
      - (Why? Removal in the subtree may decrease the number of keys in a parent potentially causing a violation.)
    - Key to remove is found in internal node, recursively demote the key down to a leaf node for deletion. (case 2)
    - Key is found in leaf node. Delete key (case 1)

Deleting a Key from a B-tree

- Cases 1 + 2
- Key k is deleted from node thisNode

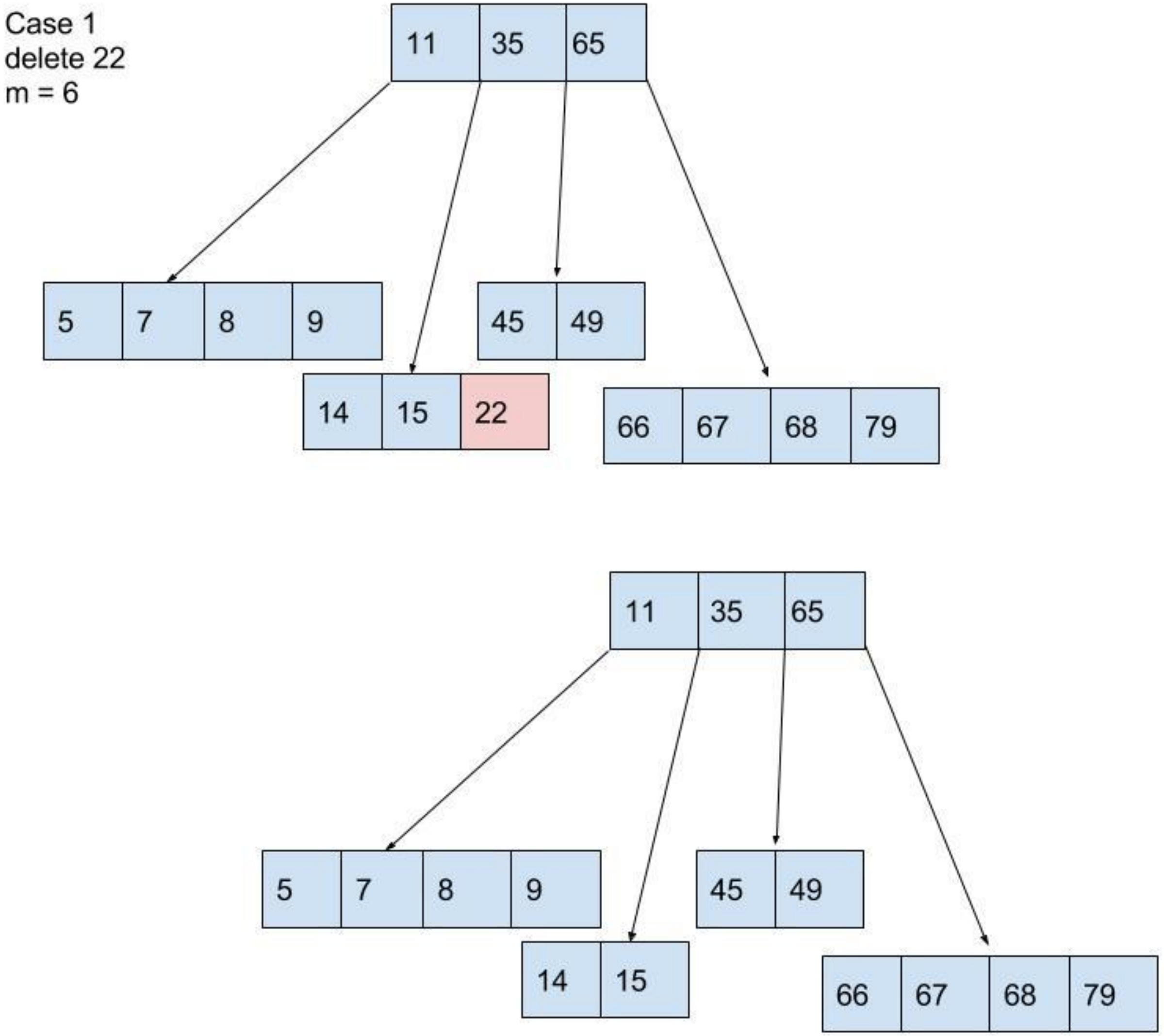
1. thisNode is a leaf node. Simple – just delete k. base case.
2. thisNode is an internal node
   1. Assume k is the i^{th} key in thisNode. thisNode.child_i has more than the minimum number of keys, find predecessor key k' in subtree rooted by thisNode.child_i . Delete k' and replace k with k' in thisNode. (Goal: repeatedly demote k down to a leaf node with a series of "swaps"). Continue traversal down tree (to continue demotion).
   2. ELSE if thisNode.child_i does not have more than minimum number of keys: perform the step above with the $\text{thisNode.child}_{i+1.}$ Continue traversal down tree (to continue demotion).
   3. ELSE if both thisNode.child_i and $\text{thisNode.child}_{i+1}$ do not have the minimum number of keys, demote k and merge k and the contents of $\text{thisNode.child}_{i+1}$ into $\text{thisNode.child}_{i.}$ Adjust thisNodes keys appropriately. Continue traversal down tree (to continue demotion).

Deleting a Key from a B-tree

- Final Case: traversing down tree, searching for key k. Ensure appropriate number of keys on the way down
    3. k is not contained in internal node iNode. Determine which child iNode.child$_i$ roots the subtree that contains k
        1. If iNode.child$_i$ has the minimum number of keys, but has a sibling with more than the minimum number of keys. Transfer an extra key into iNode.child$_i$ from iNode: move a key from sibling iNode.child$_{i+1}$ or iNode.child$_{i-1}$ to iNode , and “promote” the appropriate child from sibling to iNode. Continue traversal down tree (in search of k).
        2. If all children of iNode have the minimum number of keys, merge two of the sibling into one. Move a key down from iNode to the new merged node to become the median key for the new node. Adjust iNodes appropriately. Continue traversal down tree (in search of k).
        3. iNode.child$_i$ has the MORE THAN minimum number of keys, Continue traversal down tree (in search of k).

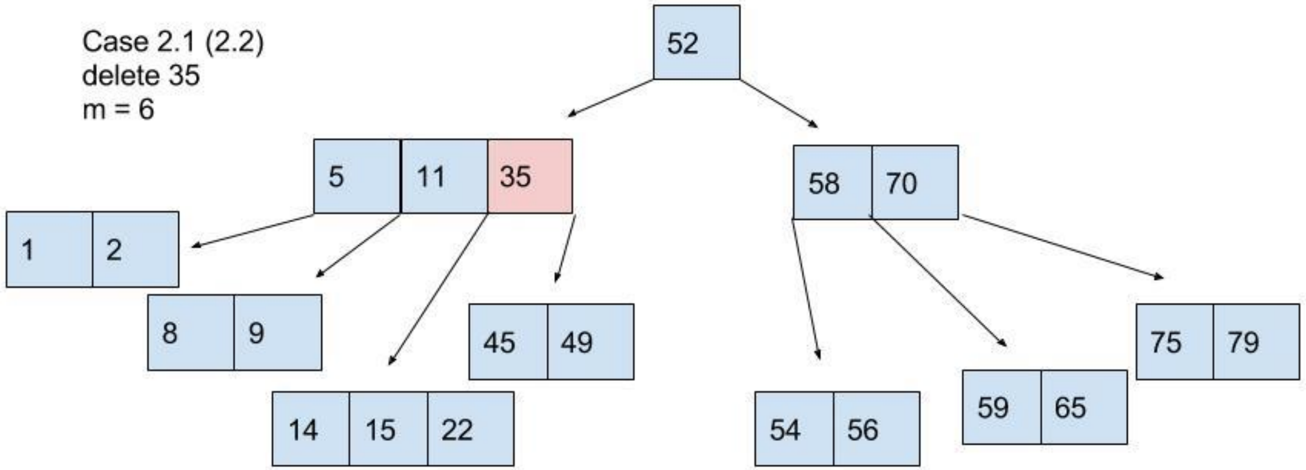
Example: Deleting from a B-Tree

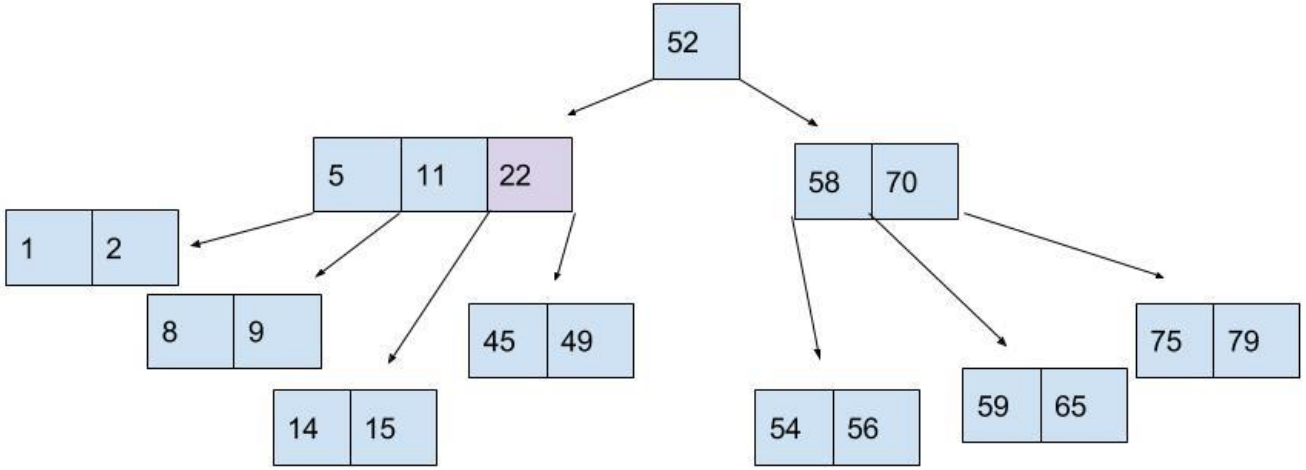
- Case 1:
- Remove 22



Example: Deleting from a B-Tree

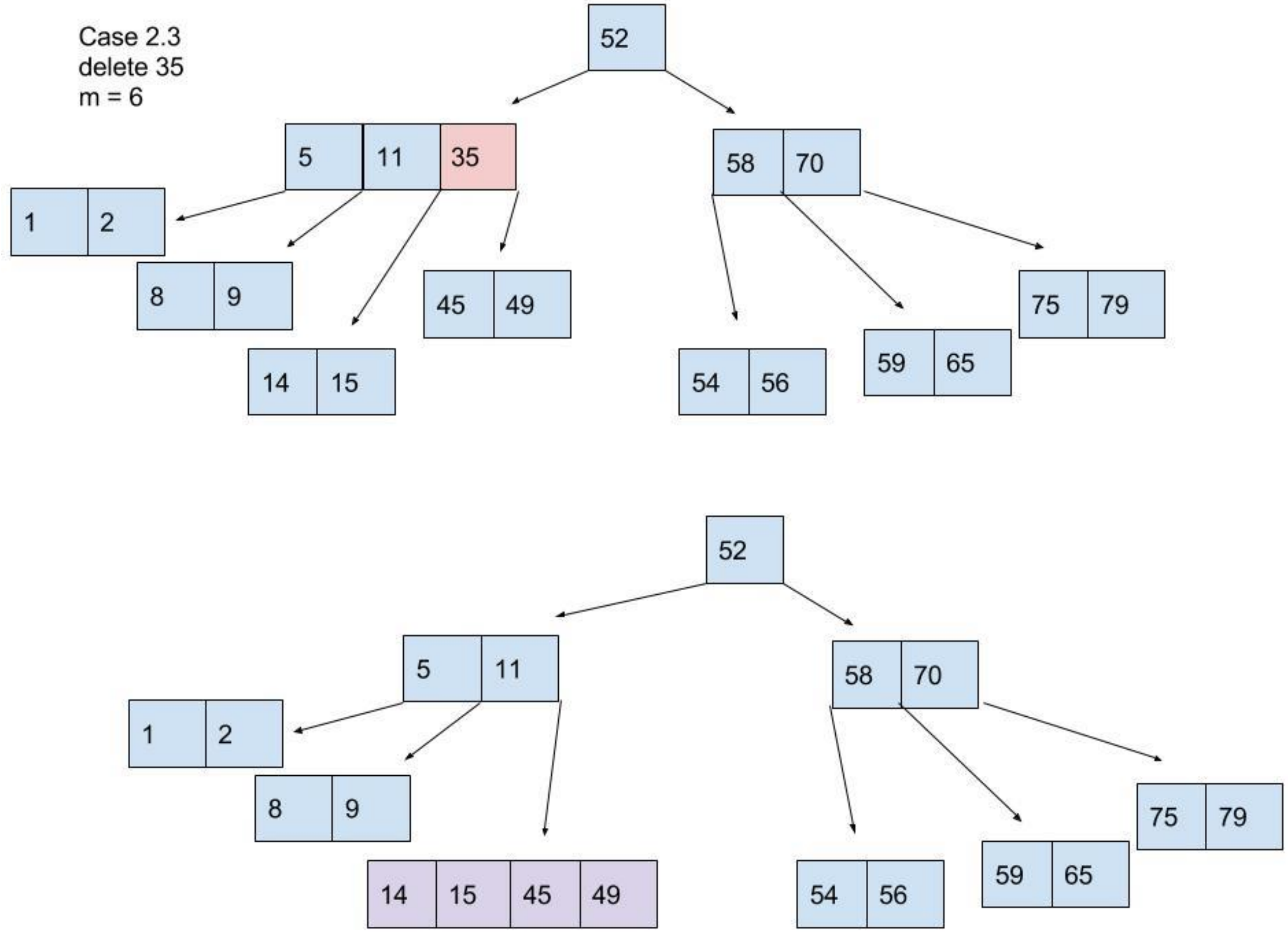
Case 2.1 (2.2)
delete 35
m = 6





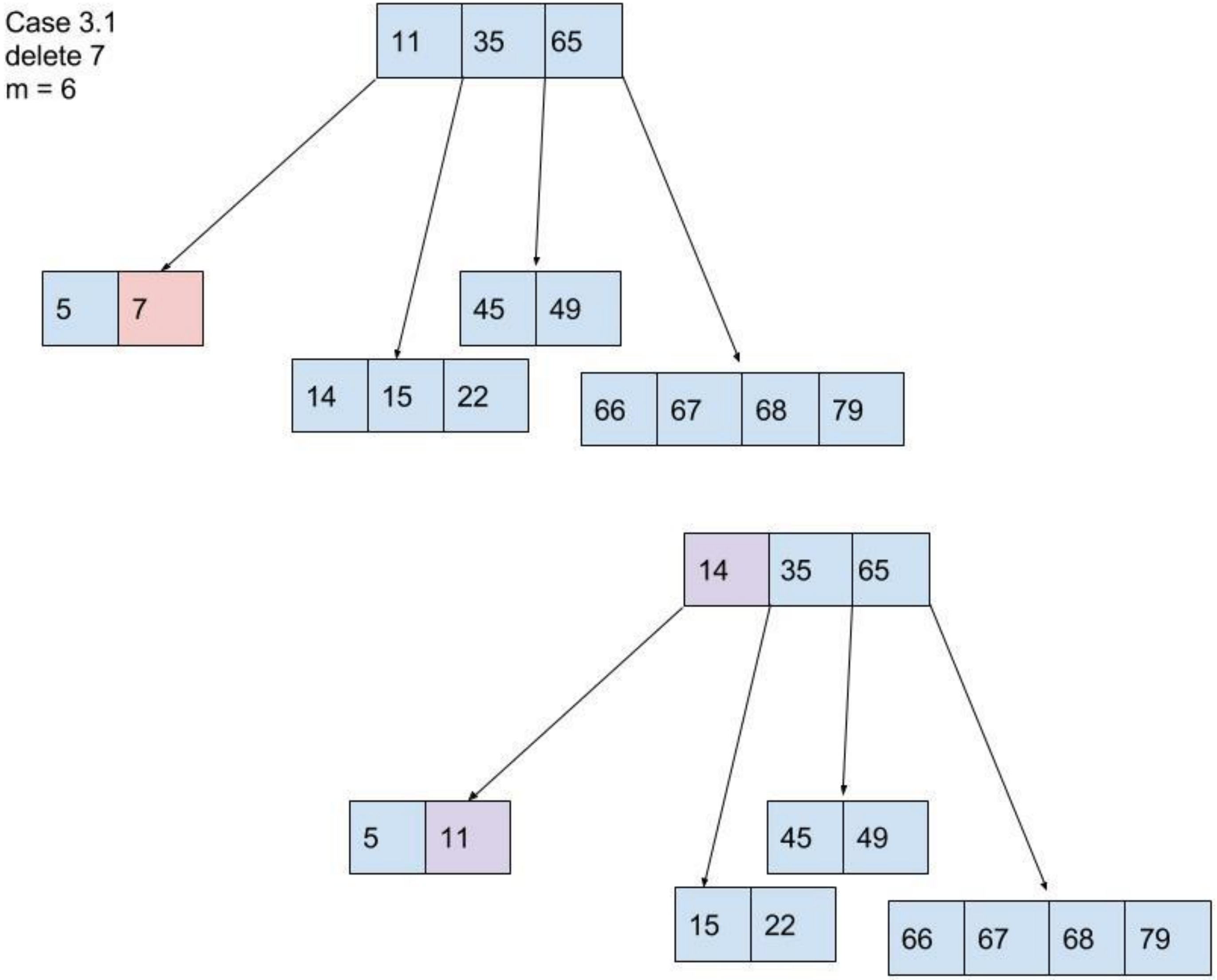
Example: Deleting from a B-Tree

- Case 2.3
- Note that thisNode’s children do not have more than the minimum number of keys
- Therefore merge two children into a new node



Example: Deleting from a B-Tree

- Case 3.1
- Remove 7
- The resulting node, thisNode, will not have a sufficient number of keys.
- Therefore “demote” key from iNode and “promote” key from sibling



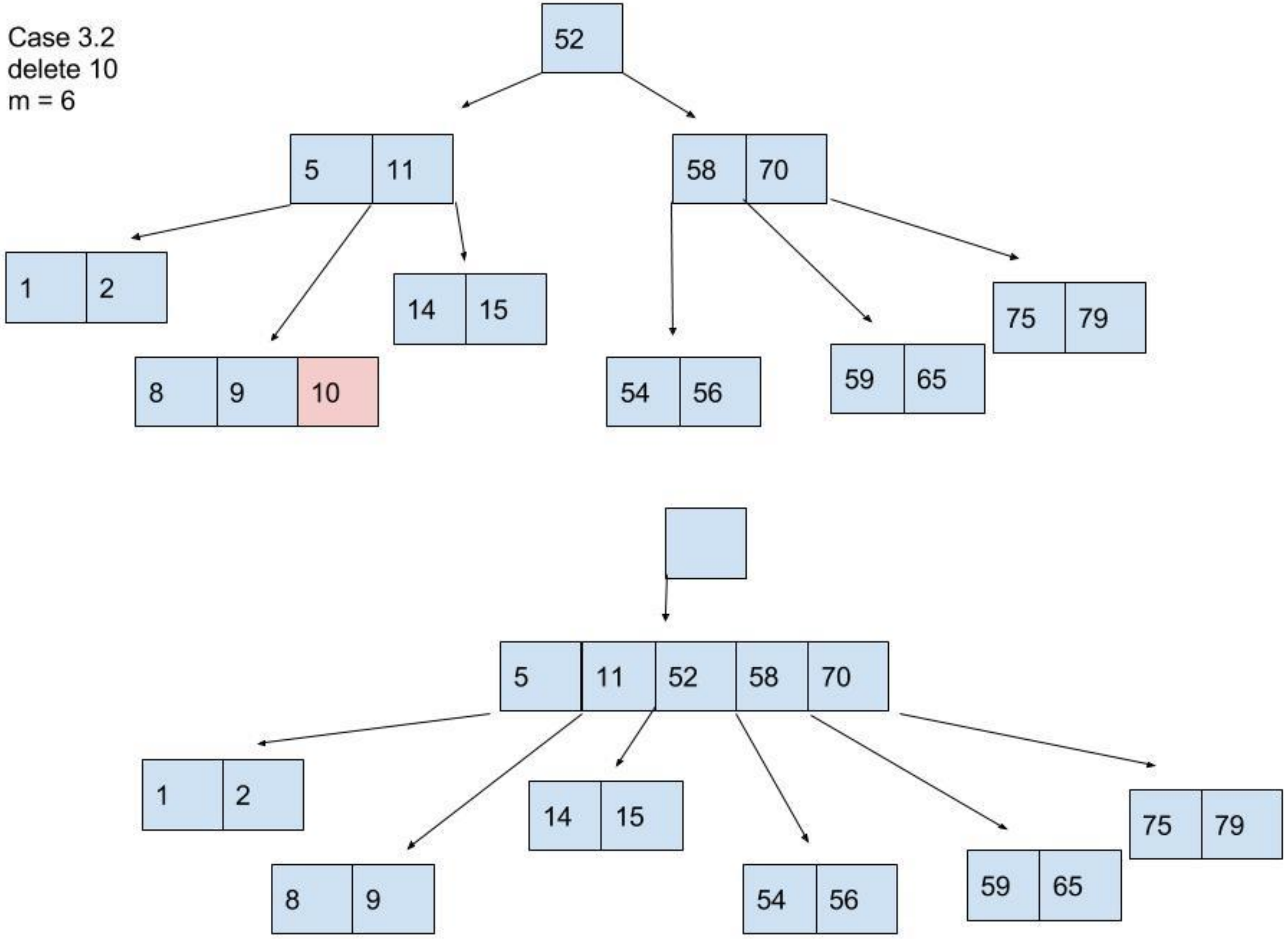
Example: Deleting from a B-Tree

- Case 3.2

- Since both siblings (5,11) and (58,70) do not have more than the min number of keys we cannot simply perform a demotion-promotion swap (case 3.1)

- Therefore we must merge node (5,11) with one of its siblings and use iNodes key as the new median
  - Special case: demoting the root. Since the only key in iNode has been demoted. All other keys at iNode must be adjusted (but there are no others!)
    - Delete oldRoot
  - Observe the height of the tree changes only in this case – with the deletion of the oldRoot.

- Once this merge occurs, the recursive search for the key 10 can proceed. 10 is finally removed (case 1)



Readings

- Family of B-Trees
  - B* Trees
    - If node is full, only split if sibling is also full; otherwise swap values with parent and sibling to avoid split.
      - Idea: reduces the frequency of splitting which can be time consuming.
    - Variation of "fullness" constraint for splitting nodes.
    - Result minimum fullness is $\frac{2}{3}$ instead of $\frac{1}{2}$.
    - B^n Trees (generalization): a node is full at ratio (n+1)/(n+2)

  - B+ Trees
    - Observe: B-tree still have a very inefficient in-order traversal.
    - Internal nodes contain keys (only)
    - Leaf nodes contain keys and pointers associated with corresponding data
      - Leaves also generally contain pointer to the next leaf node (in order)
    - Result
      - Leaves contain all keys and all data references
      - Internal nodes are only used as indices to search for data.
      - Thus a key found in an internal node is also found in a leaf node!

Summary of B-Trees

- Practical benefits to be gained when storing large structures in memory

- Accessing memory off chip is slow! B-trees reduce the number of memory accesses in the worst case as compared to BSTs
  - $O(m \log_m n)$ vs $O(\log_2 n)$
  - Number of disk accesses is reduced for large m.
    - $\Theta(\log_m n)$

- Investigation: How does the choice of m affect the overall complexity?
  - Theoretically?
  - In practice?

Appendix

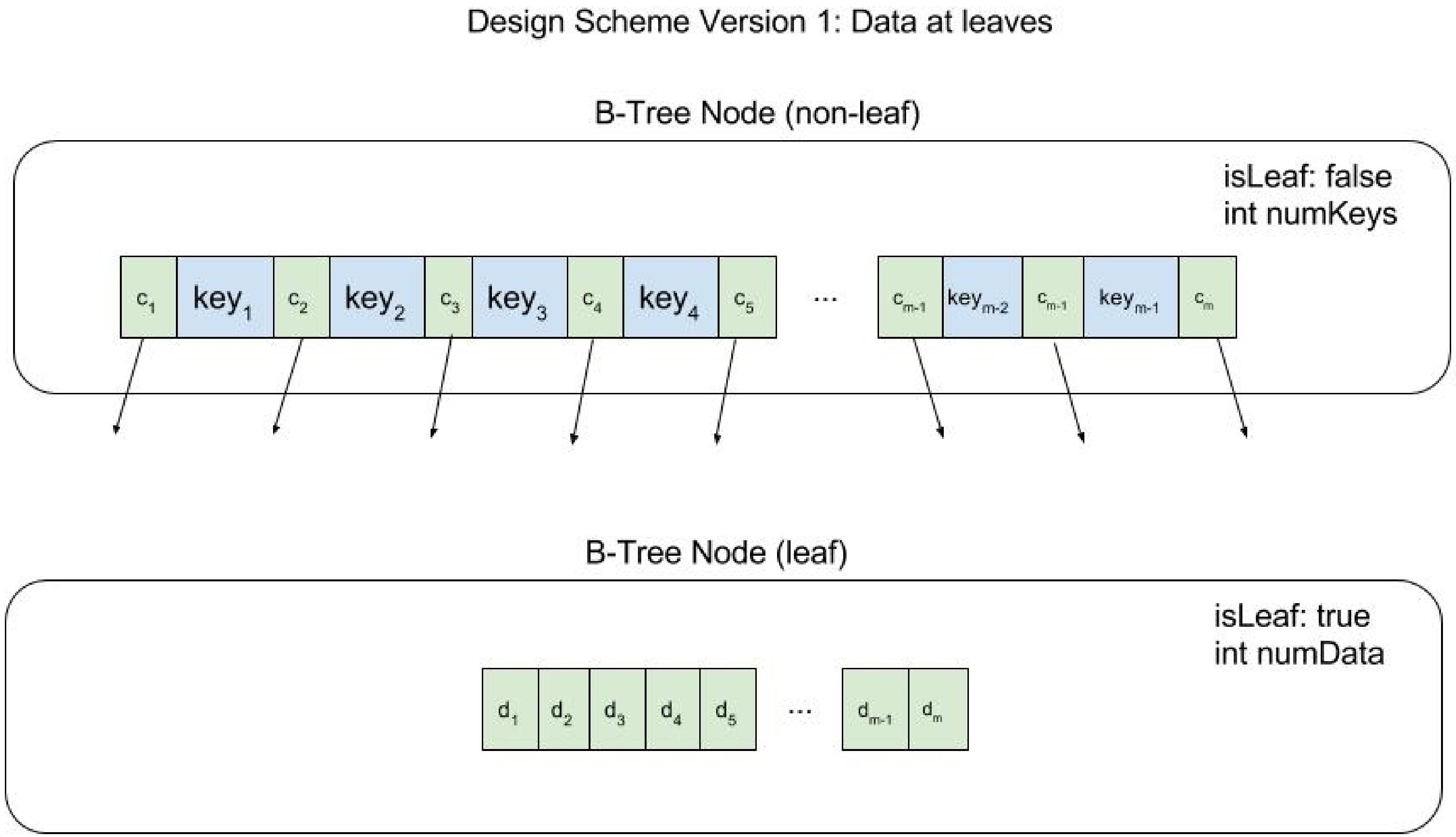
Jeremy Bolton, PhD

Assistant Teaching Professor

GEORGETOWN UNIVERSITY

Design Scheme: B+ trees

- Data only at the leaves
- Keys are only used to guide a search to data
- Key values are by standard the smallest value in the right subtree



Design Scheme: B+ trees

- Data only at the leaves
- Keys are only used to guide a search to data
- Key values are by standard the smallest value in the right subtree
- Here m = 4