13.1 B-trees

This section has been set as optional by your instructor.

Introduction to B-trees

In a binary tree, each node has one key and up to two children. A ***B-tree*** with order K is a tree where nodes can have up to K-1 keys and up to K children. The ***order*** is the maximum number of children a node can have. Ex: In a B-tree with order 4, a node can have 1, 2, or 3 keys, and up to 4 children. B-trees have the following properties:

- All keys in a B-tree must be distinct.
- All leaf nodes must be at the same level.
- An internal node with N keys must have N+1 children.
- Keys in a node are stored in sorted order from smallest to largest.
- Each key in a B-tree internal node has one left subtree and one right subtree. All left subtree keys are < that key, and all right subtree keys are > that key.

PARTICIPATION ACTIVITY 13.1.1: Order 3 B-trees.

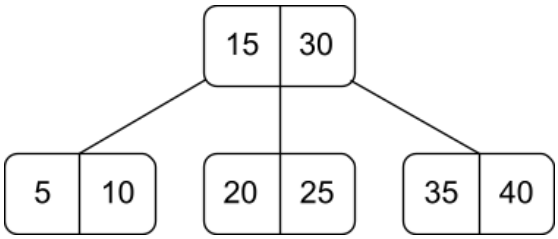
Animation captions:

1. A single node in a B-tree can contain multiple keys.
2. An order 3 B-tree can have up to 2 keys per node. This root node contains the keys 10 and 20, which are ordered from smallest to largest.
3. An internal node with 2 keys must have three children. The node with keys 10 and 20 has three children nodes, with keys 5, 15, and 25.
4. The root's left subtree contains the key 5, which is less than 10.
5. The root's middle subtree contains the key 15, which is greater than 10 and less than 20.
6. The root's right subtree contains the key 25, which is greater than 20.
7. All left subtree keys are < the parent key, and all right subtree keys are > the parent key.

PARTICIPATION ACTIVITY 13.1.2: Validity of order 3 B-trees.

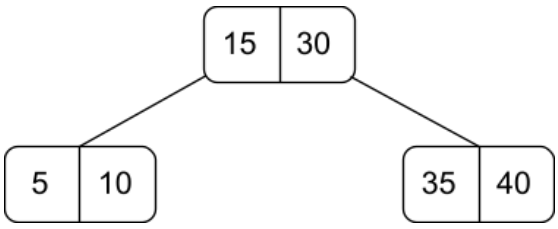
Determine which of the following are valid order 3 B-trees.

1)



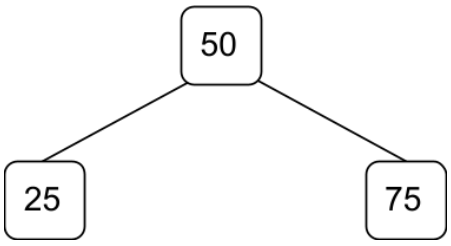
- Valid
- Invalid

2)



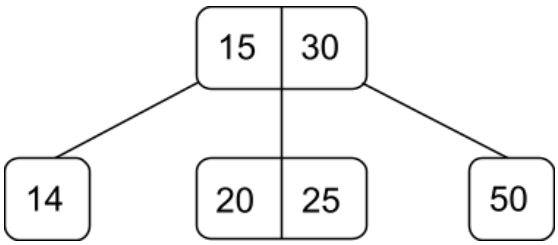
- Valid
- Invalid

3)



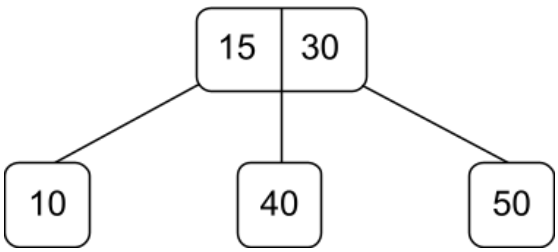
- Valid
- Invalid

4)



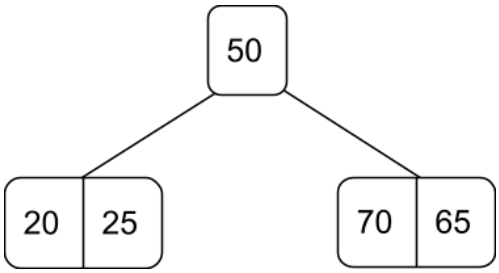
- Valid
- Invalid

5)



- Valid
- Invalid

6)

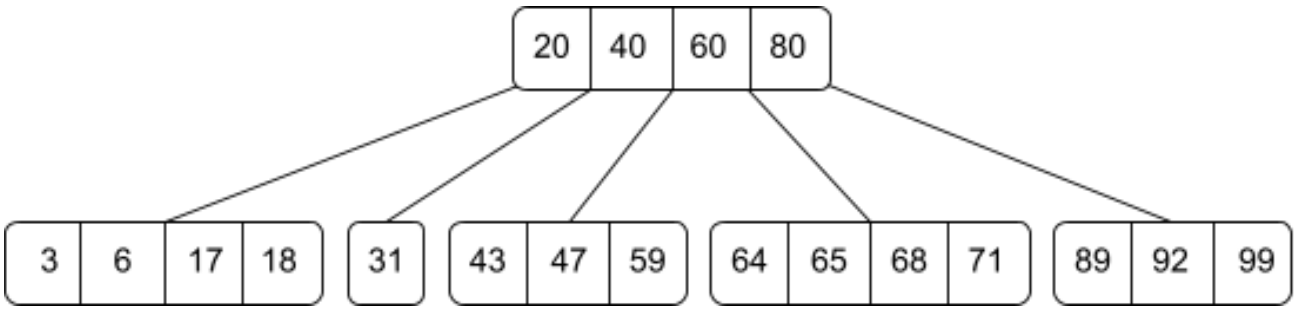


- Valid
- Invalid

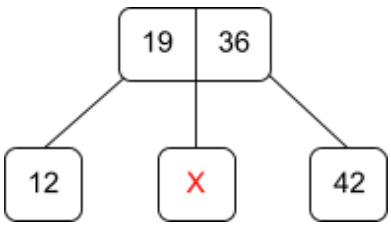
Higher order B-trees

As the order of a B-trees increases, the maximum number of keys and children per node increases. An internal node must have one more child than keys. Each child of an internal node can have a different number of keys than the parent internal node. Ex: An internal node in an order 5 B-tree could have 1 child with 1 key, 2 children with 3 keys, and 2 children with 4 keys.

Example 13.1.1: A valid order 5 B-tree.



PARTICIPATION ACTIVITY 13.1.3: B-tree properties.



1) What is the minimum possible order of this B-tree?

Check Show answer

2) What is the minimum possible integer value for the unknown key X?

Check Show answer

3) What is the maximum possible integer value for the unknown key X?

Check Show answer

2-3-4 Trees

A 2-3-4 tree is an order 4 B-tree. Therefore, a 2-3-4 tree node contains 1, 2 or 3 keys. A leaf node in a 2-3-4 tree has no children.

Table 13.1.1: 2-3-4 tree internal nodes.

Number of keys	Number of children
1	2
2	3
3	4

PARTICIPATION ACTIVITY 13.1.4: 2-3-4 tree properties.

1) A 2-3-4 tree is a B-tree of order _____.

Check Show answer

2) What is the minimum number of children that a 2-3-4 internal node with 2 keys can have?

Check Show answer

3) What is the maximum number of children that a 2-3-4 internal node with 2 keys can have?

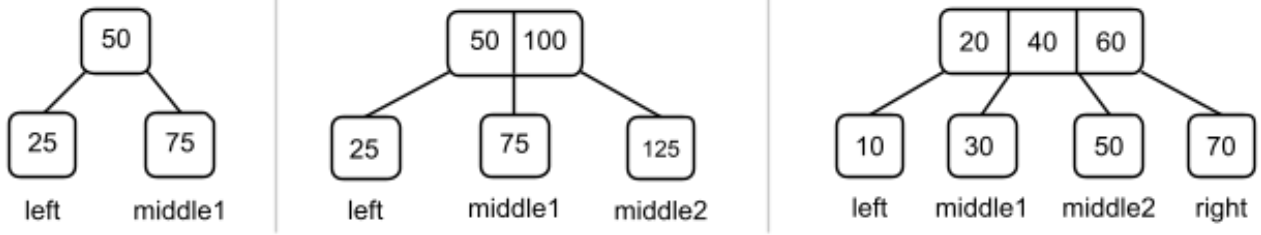
Check Show answer

2-3-4 tree node labels

The keys in a 2-3-4 tree node are labeled as A, B and C. The child nodes of a 2-3-4 tree internal node are labeled as left, middle1, middle2, and right. If a node contains 1 key, then keys B and C, as well as children middle2 and right, are not used. If a node contains 2 keys, then key C, as well as the right child, are not used. A 2-3-4 tree node containing exactly 3 keys is said to be ***full***, and uses all keys and children.

A node with 1 key is called a ***2-node***. A node with 2 keys is called a ***3-node***. A node with 3 keys is called a ***4-node***.

Figure 13.1.1: 2-3-4 child subtree labels.



PARTICIPATION ACTIVITY 13.1.5: 2-3-4 tree nodes.

1) Every 2-3-4 tree internal node will have children left and _____.

Check Show answer

2) The right child is only used by a 2-3-4 tree internal node with _____ keys.

Check Show answer

3) A node in a 2-3-4 tree that contains no children is called a _____ node.

Check Show answer

4) A 2-3-4 tree node with _____ keys is said to be full.

Check Show answer

13.2 2-3-4 tree search algorithm

This section has been set as optional by your instructor.

Given a key, a ***search*** algorithm returns the first node found matching that key, or returns null if a matching node is not found. Searching a 2-3-4 tree is a recursive process that starts with the root node. If the search key equals any of the keys in the node, then the node is returned. Otherwise, a recursive call is made on the appropriate child node. Which child node is used depends on the value of the search key in comparison to the node's keys. The table below shows conditions, which are checked in order, and the corresponding child nodes.

Table 13.2.1: 2-3-4 tree child node to choose based on search key.

Condition	Child node to search
key < node's A key	left
node has only 1 key or key < node's B key	middle1
node has only 2 keys or key < node's C key	middle2
none of the above	right

PARTICIPATION ACTIVITY 13.2.1: 2-3-4 tree search algorithm.

Animation content:

Static figure: A code block and a 2-3-4 tree labeled BTreeSearch(tree⋯→root, 70).
Begin pseudocode:

```
BTreeSearch(node, key) {
  if (node is not null) {
    if (node has key) {
      return node
    }
    if (key < node⋯→A) {
      return BTreeSearch(node⋯→left, key)
    }
    else if (node⋯→B is null || key < node⋯→B) {
      return BTreeSearch(node⋯→middle1, key)
    }
    else if (node⋯→C is null || key < node⋯→C) {
      return BTreeSearch(node⋯→middle2, key)
    }
    else {
      return BTreeSearch(node⋯→right, key)
    }
  }
  return null
}
```

End pseudocode.

Static figure: Code for BTreeSearch() function is shown on the left. On the right is a 2-3-4 tree with four nodes and six keys. Root's keys are 25 and 50. Root's left child has key 10. Root's middle1 child has keys 35 and 40. Root's middle2 child has key 70. Above the tree is the search function call: BTreeSearch(tree⇢root, 70). A label with text "node" has an accompanying arrow that points to node 70.

Step 1: Execution of BTreeSearch(tree⇢root, 70) begins. Execution highlight is shown on function signature, not yet executing code in the function bode. A label "node" appears with an accompanying arrow pointing to the root node.

Step 2: The first if statement executes and the condition is true since node is not null. Code highlight proceeds to the next if statement's condition, "node has key". The comparisons that occur are shown: 70 != 25 and 70 != 50. Neither is true, so the code highlight proceeds to the closing curly brace for the if statement.

Step 3: The next if statement's condition is evaluated and is false because 70 is not less than 25. The next else-if statement's condition is also false because 70 is not less than 50. The following else-if statement's condition is true because node⇢C is null.

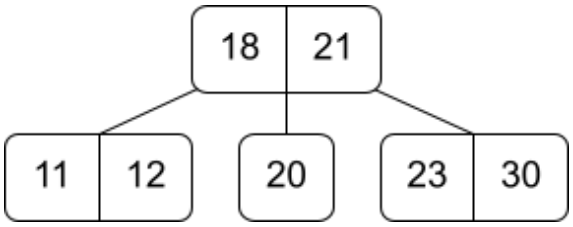
Step 4: The following recursive call executes: BTreeSearch(node⇢middle2, key). Code execution moves back to the function's top line and the node label and arrow move to node 70.

Step 5: The first two if statement's condition is evaluated and is true since node is not null. The next if statement's condition is also true since 70 == 70, meaning the node has the key. So execution reaches the "return node" statement and the search is done.

Animation captions:

1. Search for 70 starts at the root node.
2. node is not null, so the search compares 70 with the node's two keys, 25 and 50. No match occurs, so the node does not have key 70.
3. Since no match was found in the root node, the search algorithm compares the key to the node's keys to determine the recursive call.
4. 70 is greater than 50, and the node does not contain a key C, so a recursive call to the middle2 child node occurs.
5. node is not null, so 70 is compared with the node's A key. A match occurs, so the node is returned.

PARTICIPATION ACTIVITY 13.2.2: 2-3-4 tree search.



1) When searching for key 23, what node is visited first?

- Root
- Root's left child
- Root's middle1 child
- Root's middle2 child

2) When searching for key 23, how many keys in the root are compared against 23?

- 1
- 2
- 3
- 4

3) When searching for key 23, the root node will be the only node that is visited.

- True
- False

4) When searching for key 23, what is the total number of nodes visited?

- 1
- 2
- 3
- 4

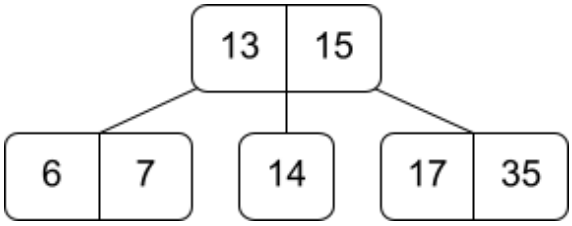
5) When searching for key 20, what is returned by the search function?

- Null
- Root's left child
- Root's middle1 child
- Root's middle2 child

6) When searching for key 19, what is returned by the search function?

- Null
- Root's left child
- Root's middle1 child
- Root's middle2 child

PARTICIPATION ACTIVITY 13.2.3: 2-3-4 tree search algorithm.



1) When searching for key 6, search starts at the root. Since the root node does not contain the key 6, which recursive search call is made?

- `BTreeSearch(node→left, key)`
- `BTreeSearch(node→middle1, key)`
- `BTreeSearch(node→middle2, key)`
- `BTreeSearch(node→right, key)`

2) When searching for key 6, after making the recursive call on the root's left node, which return statement is executed?

- `return BTreeSearch(node→left, key)`
- `return node→A`
- `return node`
- `return null`

3) When searching for key 15, which recursive search call is made?

- `BTreeSearch(node→left, key)`
- `BTreeSearch(node→middle1, key)`
- no recursive call is made

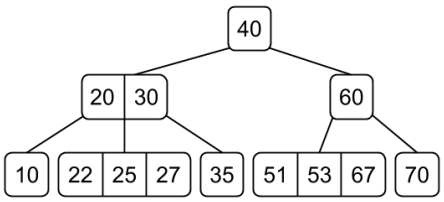
CHALLENGE ACTIVITY 13.2.1: 2-3-4 tree search algorithm.

502696.2096894.qx3zqy7

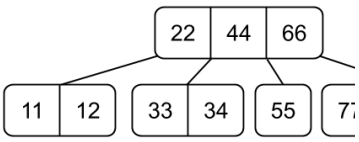
Start

Select all valid 2-3-4 trees.

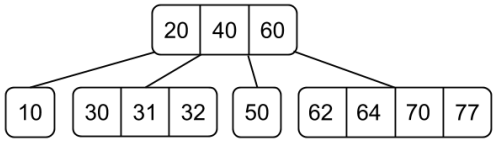
A



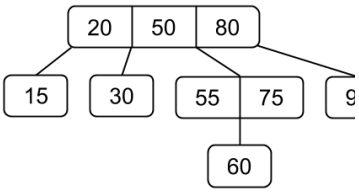
D



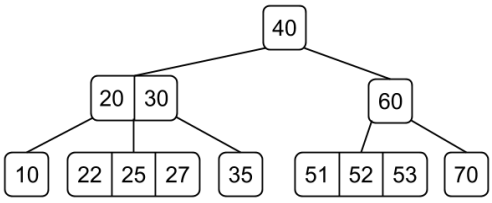
B



E



C



1 2 3 4

Check Next

13.3 2-3-4 tree insert algorithm

This section has been set as optional by your instructor.

2-3-4 tree insertions and split operations

Given a new key, a 2-3-4 tree ***insert*** operation inserts the new key in the proper location such that all 2-3-4 tree properties are preserved. New keys are always inserted into leaf nodes in a 2-3-4 tree. Insertion returns the leaf node where the key was inserted, or null if the key was already in the tree.

An important operation during insertion is the ***split*** operation, which is done on every full node encountered during insertion traversal. The split operation moves the middle key from a child node into the child's parent node. The first and last keys in the child node are moved into two separate nodes. The split operation returns the parent node that received the middle key from the child.

PARTICIPATION ACTIVITY 13.3.1: Split operation.

Animation captions:

1. To split the full root node, the middle key moves up, becoming the new root node with a single value.
2. To split a full, non-root node, the middle value is moved up into the parent node.
3. Compared to the original, the tree contains the same values after the split, and all 2-3-4 tree requirements are still satisfied.

PARTICIPATION ACTIVITY 13.3.2: Split operation.

1) During insertion, only a full node can be split.
   - True
   - False

2) During insertion of a key K, after splitting a node, the key K is immediately inserted into the node.
   - True
   - False

3) What is the result of splitting a full root node?

- The total number of nodes in the tree decreases by 1.
- The total number of nodes in the tree does not change.
- The total number of nodes in the tree increases by 1.
- The total number of nodes in the tree increases by 2.

4) When a full internal node is split, which key moves up into the parent node?

- First
- Middle
- Last

Split operation algorithm

Splitting an internal node allocates 2 new nodes, each with a single key, and the middle key from the split node moves up into the parent node. Splitting the root node allocates 3 new nodes, each with a single key, and the root of the tree becomes a new node with a single key.

PARTICIPATION ACTIVITY 13.3.3: B-tree split operation.

Animation content:

undefined

Animation captions:

1. Splitting a node starts by verifying that the node is full. A pointer to the parent node is also needed when splitting an internal node.
2. New node allocation is necessary. splitLeft is allocated with a single key copied from node⇢A, and two null child pointers copied from node⇢left and node⇢middle1.
3. splitRight is allocated with a single key copied from node⇢C, and null child pointers copied from node⇢middle2 and node⇢right.
4. Since nodeParent is not null, the key 37 moves from node into nodeParent and the two newly allocated children are attached to nodeParent as well.
5. Splitting the root node allocates 3 new nodes, one of which becomes the new root.

During a split operation, any non-full internal node may need to gain a key from a split child node. This key may have children on either side.

Figure 13.3.1: Inserting a key with children into a non-full parent node.

```
BTreeInsertKeyWithChildren(parent, key, leftChild,
rightChild) {
   if (key < parent→A) {
      parent→C = parent→B
      parent→B = parent→A
      parent→A = key
      parent→right = parent→middle2
      parent→middle2 = parent→middle1
      parent→middle1 = rightChild
      parent→left = leftChild
   }
   else if (parent→B is null || key < parent→B) {
      parent→C = parent→B
      parent→B = key
      parent→right = parent→middle2
      parent→middle2 = rightChild
      parent→middle1 = leftChild
   }
   else {
      parent→C = key
      parent→right = rightChild
      parent→middle2 = leftChild
   }
}
```


PARTICIPATION ACTIVITY 13.3.4: B-tree split operation.

1) Like searching, the split operation in a 2-3-4 tree is recursive.
   - True
   - False

2) If a non-full node is passed to BTreeSplit, then the root node is returned.
   - True
   - False

3) Allocating new nodes is necessary for the split operation.

- True
- False

4) The root node is split in the same way a non-root node is split.

- True
- False

5) When splitting a node, a pointer to the node's parent is required.

- True
- False

6) The split function should always split a node, even if the node is not full.

- True
- False

Inserting a key into a leaf node

A new key is always inserted into a non-full leaf node. The table below describes the 4 possible cases for inserting a new key into a non-full leaf node.

Table 13.3.1: 2-3-4 tree non-full-leaf insertion cases.

Condition	Outcome
New key equals an existing key in node	No insertion takes place, and the node is not altered.
New key is < node's first key	Existing keys in node are shifted right, and the new key becomes node's first key.
Node has only 1 key or new key is < node's middle key	Node's middle key , if present, becomes last key, and new key becomes node's middle key.
None of the above	New key becomes node's last key.

PARTICIPATION ACTIVITY 13.3.5: Insertion of key into leaf node.

1) A non-full leaf node can have any key inserted.
   - ○ True
   - ○ False

2) When the key 30 is inserted into a leaf node with keys 20 and 40, 30 becomes which node value?
   - ○ A
   - ○ B
   - ○ C

3) When the key 50 is inserted into a leaf node with key 25, 50 becomes which node value?
   - ○ A
   - ○ B
   - ○ C

4) When inserting a new key into a node with 1 key, the new key can become the A, B, or C key in the node.
   - ○ True
   - ○ False

5) When the key 50 is inserted into a leaf node with keys 10, 20, and 30, 50 becomes which value?
   - ○ A
   - ○ B
   - ○ C
   - ○ none of the above

B-tree insert with preemptive split

Multiple insertion schemes exist for 2-3-4 trees. The ***preemptive split*** insertion scheme always splits any full node encountered during insertion traversal. The preemptive split insertion scheme ensures that any time a full node is split, the parent node has room to accommodate the middle value from the child.

PARTICIPATION ACTIVITY 13.3.6: B-tree insertion with preemptive split algorithm.

Animation content:

undefined

Animation captions:

1. Insertion of 60 starts at the root. A series of checks are executed on the node.
2. 60 is inserted and the root node is returned.
3. Insertion of 20 again begins at the root. The search ensures that 20 is not already in the node.
4. The full root node is split and the return value from the split is assigned to node.
5. The root node is not a leaf, so a recursive call is made to insert into the left child of the root.
6. After the series of checks, 20 is inserted and the left child of the root is returned.

PARTICIPATION ACTIVITY 13.3.7: Preemptive split insertion.

1) When arriving at a node during insertion, what is the first check that must take place?
   - ○ Check if the node is a leaf
   - ○ Check if the node already contains the key being inserted
   - ○ Check to see if the node is full

2) After any insertion operation completes, the root node will never have 3 keys.
   - ○ True
   - ○ False

3) During insertion, a parent node can temporarily have 4 keys, if a child node is split.
   - ○ True
   - ○ False

4) If a node has 2 keys, 20 and 40, then only keys > 20 and < 40 could be inserted into this node.

- True
- False

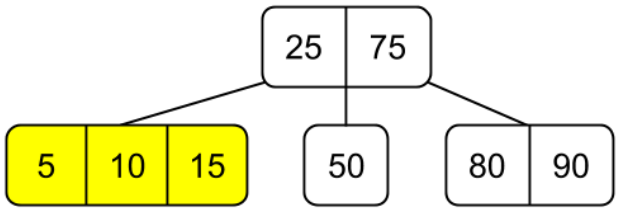
5) During insertion, how does a 2-3-4 expand in height?

- When a value is inserted into a leaf, the tree will always grow in height.
- When splitting a leaf node, the tree will always grow in height.
- When splitting the root node, the tree will always grow in height.
- Any insertion that does NOT involve splitting any nodes will cause the tree to grow in height.

CHALLENGE ACTIVITY 13.3.1: 2-3-4 tree insert algorithm.

502696.2096894.qx3zqy7

Start



The node (5, 10, 15) is split.
Enter each node's keys after the split, or "none" if the node doesn't exist.

Root: Ex: 10, 20, 30

Root's left child:

Root's middle1 child:

Root's middle2 child:

Root's right child:

Height of tree: Ex: 5

1 2 3 4

Check Next

13.4 2-3-4 tree rotations and fusion

This section has been set as optional by your instructor.

Rotation concepts

Removing an item from a 2-3-4 tree may require rearranging keys to maintain tree properties. A ***rotation*** is a rearrangement of keys between 3 nodes that maintains all 2-3-4 tree properties in the process. The 2-3-4 tree removal algorithm uses rotations to transfer keys between sibling nodes. A ***right rotation*** on a node causes the node to lose one key and the node's right sibling to gain one key. A ***left rotation*** on a node causes the node to lose one key and the node's left sibling to gain one key.

PARTICIPATION ACTIVITY 13.4.1: Left and right rotations.

Animation content:

undefined

Animation captions:

1. A right rotation on the root's left child moves 23 into the root, and 27 into the root's middle1 child.
2. A left rotation on the root's right child moves 73 into the root, and 55 into the root's middle1 child.

PARTICIPATION ACTIVITY 13.4.2: 2-3-4 tree rotations.

1) A rotation on a node changes the set of keys in one of the node's children.
   - True
   - False

2) A rotation on a node changes the set of keys in the node's parent.
   - True
   - False

3) A left rotation can only be performed on a node that has a left sibling.
   - True
   - False

4) A rotation operation may change the height of a 2-3-4 tree.
   - True
   - False

Utility functions for rotations

Several utility functions are used in the rotation operation.

- `BTreeGetLeftSibling` returns a pointer to the left sibling of a node or null if the node has no left sibling. BTreeGetLeftSibling returns null, left, middle1, or middle2 if the node is the left, middle1, middle2, or the right child of the parent, respectively. Since the parent node is required, a precondition of this function is that the node is not the root.
- `BTreeGetRightSibling` returns a pointer to the right sibling of a node or null if the node has no right sibling.
- `BTreeGetParentKeyLeftOfChild` takes a parent node and a child of the parent node as arguments, and returns the key in the parent that is immediately left of the child.
- `BTreeSetParentKeyLeftOfChild` takes a parent node, a child of the parent node, and a key as arguments, and sets the key in the parent that is immediately left of the child.
- `BTreeAddKeyAndChild` operates on a non-full node, adding one new key and one new child to the node. The new key must be greater than all keys in the node, and all keys in the new child subtree must be greater than the new key. Ex: If the node has 1 key, the newly added key becomes key B in the node, and the child becomes the middle2 child. If the node has 2 keys, the newly added key becomes key C in the node, and the child becomes the right child.
- `BTreeRemoveKey` removes a key from a node using a key index in the range [0,2]. This process may require moving keys and children to fill the location left by removing the key. The pseudocode for BTreeRemoveKey is below.

Figure 13.4.1: BTreeRemoveKey pseudocode.

```
BTreeRemoveKey(node, keyIndex)
{
   if (keyIndex == 0) {
      node→A = node→B
      node→B = node→C
      node→C = null
      node→left = node→middle1
      node→middle1 =
node→middle2
      node→middle2 = node→right
      node→right = null
   }
   else if (keyIndex == 1) {
      node→B = node→C
      node→C = null
      node→middle2 = node→right
      node→right = null
   }
   else if (keyIndex == 2) {
      node→C = null
      node→right = null
   }
}
```


PARTICIPATION ACTIVITY 13.4.3: Utility functions for rotations.

If unable to drag and drop, refresh the page.

BTreeGetRightSibling | BTreeAddKeyAndChild | BTreeGetParentKeyLeftOfChild | BTreeSetParentKeyLeftOfChild | BTreeRemoveKey | BTreeGetLeftSibling

	Removes a node's key by index.
	Adds a new key and child into a node that has 1 or 2 keys.
	Returns a pointer to a node's right-adjacent sibling.
	Returns a pointer to a node's left-adjacent sibling.

	Returns the key of the given parent that is immediately left of the given child.
	Replaces the parent's key that is immediately left of the child with the specified key.

Reset

Rotation pseudocode

The rotation algorithm operates on a node, causing a net decrease of 1 key in that node. The key removed from the node moves up into the parent node, displacing a key in the parent that is moved to a sibling. No new nodes are allocated, nor existing nodes deallocated during rotation. The code simply copies key and child pointers.

PARTICIPATION ACTIVITY 13.4.4: Left rotation pseudocode.

Animation content:

undefined

Animation captions:

1. A left rotation is performed on the root's middle1child. leftSibling is assigned with a pointer to node's left sibling, which is the root's left child.
2. keyForLeftSibling is assigned with 44, which is the key in parent's that is left of the node. Then, that key and the node's left child are added to the left sibling.
3. The node's leftmost key 66 is copied to the node's parent and then removed from the node.

PARTICIPATION ACTIVITY 13.4.5: Rotation Algorithm.

1) A rotation is a recursive operation.

- True
- False

2) A rotation will in some cases dynamically allocate a new node.

- True
- False

3) Any node that has an adjacent right sibling can be rotated right.

- True
- False

4) One child of the node being rotated will have a change of parent node.

- True
- False

Fusion

When rearranging values in a 2-3-4 tree during deletions, rotations are not an option for nodes that do not have a sibling with 2 or more keys. Fusion provides an additional option for increasing the number of keys in a node. A ***fusion*** is a combination of 3 keys: 2 from adjacent sibling nodes that have 1 key each, and a third from the parent of the siblings. Fusion is the inverse operation of a split. The key taken from the parent node must be the key that is between the 2 adjacent siblings. The parent node must have at least 2 keys, with the exception of the root.

Fusion of the root node is a special case that happens only when the root and the root's 2 children each have 1 key. In this case, the 3 keys from the 3 nodes are combined into a single node that becomes the new root node.

PARTICIPATION ACTIVITY 13.4.6: Root fusion.

Animation content:

undefined

Animation captions:

1. Fusion of the root happens without allocating any new nodes. First, the A, B, and C keys are set to 41, 63, and 76, respectively.
2. The 4 child pointers of the root are copied from the child pointers of the 2 children.

PARTICIPATION ACTIVITY 13.4.7: Root fusion.

1) How many nodes are allocated in the root fusion pseudocode?

- 0
- 1
- 2
- 3

2) From where does the final B key in the root after fusion come?

- The A key in the root's left child.
- The A key in the root's right child.
- The original A key in the root.
- The original C key in the root.

3) How many keys will the root have after root fusion?

- 1
- 2
- 3
- 4

4) How many child pointers are changed in the root node during fusion?

- 0
- 2
- 3
- 4

Non-root fusion

For the non-root case, fusion operates on 2 adjacent siblings that each have 1 key. The key in the parent node that is between the 2 adjacent siblings is combined with the 2 keys from the two siblings to make a single, fused node. The parent node must have at least 2 keys.

In the fusion algorithm below, the `BTreeGetKeyIndex` function returns an integer in the range [0,2] that indicates the index of the key within the node. The `BTreeSetChild` functions sets the left, middle1, middle2, or right child pointer based on an index value of 0, 1, 2, or 3, respectively.

PARTICIPATION ACTIVITY 13.4.8: Non-root fusion.

Animation content:

undefined

Animation captions:

1. leftNode is the node with key 20 and rightNode is the node with key 54. The fuse operation starts by getting a pointer to the parent.
2. The parent node is root, but does not have 1 key, so BTreeFuseRoot is not called.
3. middleKey is assigned with 30, which is the parent's key between the left and right nodes' keys.
4. The fused node is allocated with keys 20, 30, and 54. The child pointers are assigned with the left and right node's children.
5. The parent's leftmost key and child are removed. Then the parent's left child pointer is assigned with fusedNode.

PARTICIPATION ACTIVITY 13.4.9: Non-root fusion.

1) If the parent of the node being fused is the root, then BTreeFuseRoot is called.
   - ○ True
   - ○ False

2) How many keys will the returned fused node have?
   - ○ 1
   - ○ 2
   - ○ 3
   - ○ Depends on the number of keys in the parent node

3) The leftmost key from the parent node is always moved down into the fused node.
   - ○ True
   - ○ False

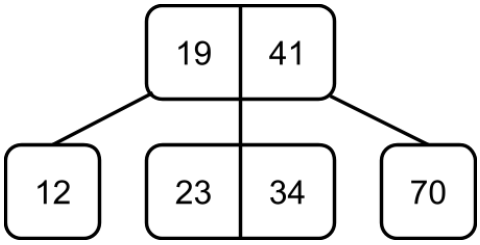
4) When the parent node has a key removed, how many child pointers must be assigned with new values?

- Only 1
- At most 2
- 3 or 4
- 2, 3, or 4

CHALLENGE ACTIVITY 13.4.1: 2-3-4 tree rotations and fusion.

502696.2096894.qx3zqy7

Start



A right rotation occurs on node (23, 34).
Enter each node's keys after the rotation, or **none** if the node doesn't exist.

Root: Ex: 10, 20, 30, or none

Root's left child:

Root's middle1 child:

Root's middle2 child:

Root's right child:

1 2 3 4

Check Next

13.5 2-3-4 tree removal

This section has been set as optional by your instructor.

Merge algorithm

A B-Tree ***merge*** operates on a node with 1 key and increases the node's keys to 2 or 3 using either a rotation or fusion. A node's 2 adjacent siblings are checked first during a merge, and if either has 2 or more keys, a key is transferred via a rotation. Such a rotation increases the number of keys in the merged node from 1 to 2. If all adjacent siblings of the node being merged have 1 key, then fusion is used to increase the number of keys in the node from 1 to 3. The merge operation can be performed on any node that has 1 key and a non-null parent node with at least 2 keys.

PARTICIPATION ACTIVITY 13.5.1: Merge algorithm.

Animation content:

undefined

Animation captions:

1. To merge the node with the key 25, a left rotation is performed on the right-adjacent sibling.
2. Since all siblings of the node with key 12 have 1 key, the merge operation is done with a fusion.

PARTICIPATION ACTIVITY 13.5.2: Merge algorithm.

If unable to drag and drop, refresh the page.

2 or 3 keys **Exactly 3 keys** **1, 2, or 3 keys** **Exactly 1 key**

	Number of keys a node must have to be merged.
	Number of keys a node must have to transfer a key to an adjacent sibling during a merge.

	Number of keys a node has after fusion.
	After a node is merged, the parent of the node will be left with this number of keys.

Reset

Utility functions for removal

Several utility functions are used in a B-tree remove operation.

- `BTreeGetMinKey` returns the minimum key in a subtree.
- `BTreeGetChild` returns a pointer to a node's left, middle1, middle2, or right child, if the childIndex argument is 0, 1, 2, or 3, respectively.
- `BTreeNextNode` returns the child of a node that would be visited next in the traversal to search for the specified key.
- `BTreeKeySwap` swaps one key with another in a subtree. The replacement key must be known to be a key that can be used as a replacement without violating any of the 2-3-4 tree rules.

Figure 13.5.1: BTreeGetMinKey pseudocode.

```
BTreeGetMinKey(node) {
   cur = node
   while (cur→left != null)
{
      cur = cur→left
   }
   return cur→A
}
```

Figure 13.5.2: BTreeGetChild pseudocode.

```
BTreeGetChild(node, childIndex)
{
   if (childIndex == 0)
      return node→left
   else if (childIndex == 1)
      return node→middle1
   else if (childIndex == 2)
      return node→middle2
   else if (childIndex == 3)
      return node→right
   else
      return null
}
```


Figure 13.5.3: BTreeNextNode pseudocode.

```
BTreeNextNode(node, key) {
   if (key < node→A)
      return node→left
   else if (node→B == null || key <
node→B)
      return node→middle1
   else if (node→C == null || key <
node→C)
      return node→middle2
   else
      return node→right
}
```

Figure 13.5.4: BTreeKeySwap pseudocode.

```
BTreeKeySwap(node, existing, replacement) {
   if (node == null)
      return false

   keyIndex = BTreeGetKeyIndex(node, existing)
   if (keyIndex == -1) {
      next = BTreeNextNode(node, existing)
      return BTreeKeySwap(next, existing,
replacement)
   }

   if (keyIndex == 0)
      node→A = replacement
   else if (keyIndex == 1)
      node→B = replacement
   else
      node→C = replacement

   return true
}
```


PARTICIPATION ACTIVITY 13.5.3: Utility functions for removal.

1) The BTreeGetMinKey function always returns the A key of a node.
   - True
   - False

2) The BTreeGetChild function returns null if the childIndex argument is greater than three or less than zero.
   - True
   - False

3) The BTreeNextNode function takes a key as an argument. The key argument will be compared to at most ____ keys in the node.
   - 1
   - 2
   - 3
   - 4

4) What happens if the BTreeKeySwap function is called with an existing key parameter that does not reside in the subtree?

- The tree will not be changed and true will be returned.
- The tree will not be changed and false will be returned.
- The key in the tree that is closest to the existing key parameter will be replaced and true will be returned.
- The key in the tree that is closest to the existing key parameter will be replaced and false will be returned.

5) The pseudocode for BTreeGetMinKey, BTreeGetChild, and BTreeNextNode have a precondition of the node parameter being non-null.

- True
- False

Remove algorithm

Given a key, a 2-3-4 tree ***remove*** operation removes the first-found matching key, restructuring the tree to preserve all 2-3-4 tree rules. Each successful removal results in a key being removed from a leaf node. Two cases are possible when removing a key, the first being that the key resides in a leaf node, and the second being that the key resides in an internal node.

A key can only be removed from a leaf node that has 2 or more keys. The ***preemptive merge*** removal scheme involves increasing the number of keys in all single-key, non-root nodes encountered during traversal. The merging always happens before any key removal is attempted. Preemptive merging ensures that any leaf node encountered during removal will have 2 or more keys, allowing a key to be removed from the leaf node without violating the 2-3-4 tree rules.

To remove a key from an internal node, the key to be removed is replaced with the minimum key in the right child subtree (known as the key's successor), or the maximum key in the leftmost child subtree. First, the key chosen for replacement is stored in a temporary variable, then the chosen key is removed recursively, and lastly the temporary key replaces the key to be removed.

PARTICIPATION ACTIVITY

13.5.4: BTreeRemove algorithm: leaf case.

Animation content:

undefined

Animation captions:

1. Removal of 33 begins by traversing through the tree to find the key.
2. All single-key, non-root nodes encountered during traversal must be merged.
3. The key 33 is found in a leaf node and is removed by calling BTreeRemoveKey.

PARTICIPATION ACTIVITY 13.5.5: BTreeRemove algorithm: non-leaf case.

Animation content:

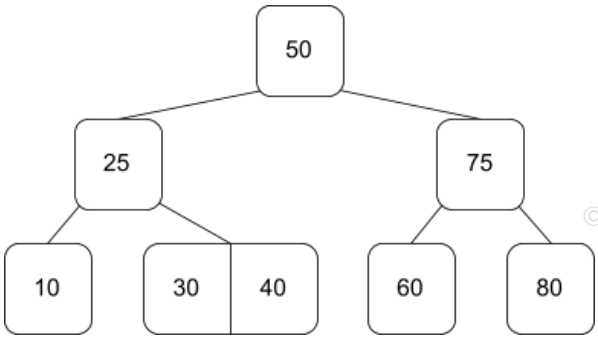
undefined

Animation captions:

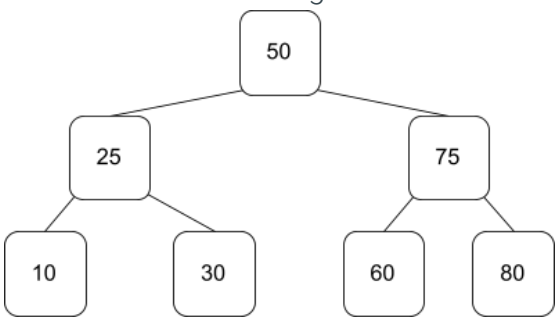
1. When deleting 60, the process is more complex due to the key being found in an internal node.
2. The key 62 is a suitable replacement for 60, but 62 must be recursively removed before the swap.
3. After the recursive removal completes, 60 is replaced with 62.

PARTICIPATION ACTIVITY 13.5.6: BTreeRemove algorithm.

Tree before removal:



1) The tree after removing 40 is:



- True
- False

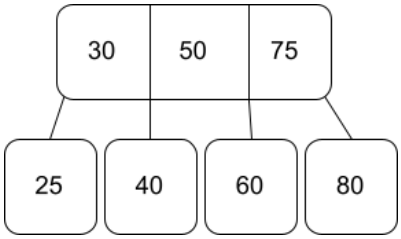
2) Calling BTreeRemove to remove any key in this tree would cause at least 1 node to be merged.

- True
- False

3) Calling BTreeRemove to remove a key NOT in this tree would cause at least 1 node to be merged.

- True
- False

4) The tree after removing 10 is:



- True
- False

5) Calling BTreeRemove to remove key 50 would result in 75 being recursively removed and then used to replace 50.

- True
- False

PARTICIPATION ACTIVITY 13.5.7: BTreeRemove algorithm.

1) If a key in an internal node is to be removed, which key(s) in the tree may be used as replacements?

- Only the minimum key in right child subtree.
- Only the maximum key in left child subtree.
- Either the minimum key in the right child subtree or the maximum key in the left child subtree.
- Any adjacent key in the same node.

2) During removal traversal, if the root node is encountered with 1 key, then the root node will be merged.

- True
- False

3) During removal traversal, any non-root node encountered with 1 key will be merged.

- True
- False

©zyBooks 11/20/23 11:12 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

4) When removing a key in an internal node, a replacement key from elsewhere in the tree is chosen and stored in a temporary variable. What is true of the replacement key?

- The replacement key came from a leaf node.
- The replacement key is either the minimum or maximum key in the entire tree.
- The replacement key will be swapped with the key to remove and then the replacement key will be recursively removed.
- No nodes will be merged during the recursive removal of the replacement key.

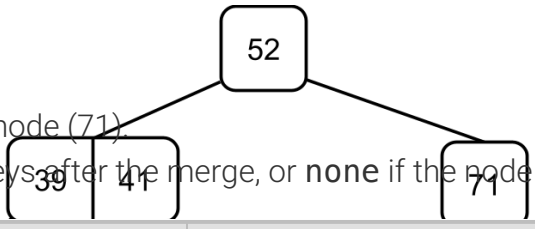
5) Removal pseudocode has the check: "if (keyIndex != -1)". What is implied about the node pointed to by cur when the condition evaluates to true?

- cur is null.
- cur has only 1 key.
- cur has no parent node.
- cur contains the key being removed.

CHALLENGE ACTIVITY 13.5.1: 2-3-4 tree removal.

502696.2096894.qx3zqy7

Start



A merge occurs on node (71).
Enter each node's keys after the merge, or **none** if the node doesn't exist:

Node	Keys
root	Ex: 10, 20, 30, or none
root⋯→left	
root⋯→middle1	
root⋯→left⋯→middle1	
root⋯→middle1⋯→left	

1 2 3 4

Check Next

13.6 Python: 2-3-4 trees

This section has been set as optional by your instructor.

Node234 class initialization and get methods

The Node234 class, representing a 2-3-4 tree node, contains data members A, B, and C for keys, and left, middle1, middle2, and right for children. The __init__() method has a required key_A argument, since a 2-3-4 tree node must always have at least 1 key. Optional left_child and middle1_child arguments allow the node to be initialized with children on either side of the A key. Both left_child and middle1_child default to None if omitted.

The Node234 class allows retrieval of children and keys by index. The get_child() method returns the node's left, middle1, middle2, or right child if the child_index argument is 0, 1, 2, or 3, respectively. The get_key() method returns the node's A, B, or C key, if the key_index argument is 0, 1, or 2, respectively. Both methods return None if the index argument is out of range.

The get_child_index() method returns 0, 1, 2, or 3 if the child argument is the node's left, middle1, middle2, or right child, respectively. -1 is returned if the child argument is not a child of the node. The

get_key_index() method returns 0, 1, or 2, if the key argument is the node's A, B, or C key, respectively. -1 is returned if the key argument is not a key of the node.

Figure 13.6.1: Node234 class initialization and get methods.

©zyBooks 11/20/23 11:12 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

©zyBooks 11/20/23 11:12 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
# Node234 class - represents a node in a 2-3-4 tree
class Node234:
    def __init__(self, key_A, left_child = None, middle1_child = None):
        self.A = key_A
        self.B = None
        self.C = None
        self.left = left_child
        self.middle1 = middle1_child
        self.middle2 = None
        self.right = None

    # Returns the left, middle1, middle2, or right child if the
child_index
    # argument is 0, 1, 2, or 3, respectively.
    # Returns None if the child_index argument is < 0 or > 3.
    def get_child(self, child_index):
        if child_index == 0:
            return self.left
        elif child_index == 1:
            return self.middle1
        elif child_index == 2:
            return self.middle2
        elif child_index == 3:
            return self.right
        return None

    # Returns 0, 1, 2, or 3 if the child argument is this node's left,
    # middle1, middle2, or right child, respectively.
    # Returns -1 if the child argument is not a child of this node.
    def get_child_index(self, child):
        if child is self.left:
            return 0
        elif child is self.middle1:
            return 1
        elif child is self.middle2:
            return 2
        elif child is self.right:
            return 3
        return -1

    # Returns this node's A, B, or C key, if the key_index argument is
    # 0, 1, or 2, respectively.
    # Returns None if the key_index argument is < 0 or > 2.
    def get_key(self, key_index):
        if key_index == 0:
            return self.A
        elif key_index == 1:
            return self.B
        elif key_index == 2:
            return self.C
        return None

    # Returns 0, 1, or 2, if the key argument is this node's A, B, or
    # C child, respectively.
    # Returns -1 if the key argument is not a key of this node.
    def get_key_index(self, key):
        if key == self.A:
            return 0
        elif key == self.B:
            return 1
```

```
        elif key == self.C:
            return 2
        return -1
```

PARTICIPATION ACTIVITY 13.6.1: Node234 class initialization and get methods.

Assume the following code is run to initialize a node:

```
node = Node234(21)
node.B = 54
node.C = 87
# Assume node12, node36, node 61, and node 88 all reference valid nodes
node.left = node12
node.middle1 = node36
node.middle2 = node61
node.right = node88
```

Match each return value to the corresponding method call.

If unable to drag and drop, refresh the page.

87 | -1 | node36 | 2 | 0 | None

	node.get_key(2)
	node.get_child(1)
	node.get_key_index(36)
	node.get_child_index(node12)
	node.get_key_index(87)
	node.get_child(36)

Reset

Other Node234 class methods

The Node234 class also contains methods for various operations on nodes:

Table 13.6.1: Other Node234 class method descriptions.

Method name	Description
append_key_and_child()	Appends 1 key and 1 child to this node, which must have 1 or 2 keys. The key must also be greater than all keys in this node, and the child subtree must contain only keys greater than the appended key.
count_keys()	Returns the number of keys in this node, which will be 1, 2, or 3.
has_key()	Returns True if this node has the specified key, False otherwise.
insert_key()	Inserts a new key into the proper location in this node, which must be a leaf and must not be full.
insert_key_with_children()	Inserts a new key into the proper location in this node, and sets the children on either side of the inserted key. This node must have 2 or fewer keys.
is_leaf()	Returns True if this node is a leaf node, False otherwise.
next_node()	Returns the child of this node that would be visited next in the traversal to search for the specified key.
remove_key()	Removes key A, B, or C from this node, if key_index is 0, 1, or 2, respectively. Other keys and children are shifted as necessary.
remove_rightmost_child()	If this node has 3 or 4 children, the rightmost child is removed and returned.
remove_rightmost_key()	If this node has 2 or 3 keys, the rightmost key is removed and returned.
set_child()	Sets the left, middle1, middle2, or right child if the child_index argument is 0, 1, 2, or 3, respectively. Does nothing if the child_index argument is < 0 or > 3.
set_key()	Sets this node's A, B, or C key if the key_index argument is 0, 1, or 2, respectively. Does nothing if the key_index argument is < 0 or > 2.

Figure 13.6.2: Other Node234 class methods.

©zyBooks 11/20/23 11:12 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
class Node234:
    # Appends 1 key and 1 child to this node.
    # Preconditions:
    # 1. This node has 1 or 2 keys
    # 2. key > all keys in this node
    # 3. Child subtree contains only keys > key
    def append_key_and_child(self, key, child):
        if self.B == None:
            self.B = key
            self.middle2 = child
        else:
            self.C = key
            self.right = child

    # Returns the number of keys in this node, which will be 1, 2, or 3.
    def count_keys(self):
        if self.C != None:
            return 3
        elif self.B != None:
            return 2
        return 1

    # Returns True if this node has the specified key, False otherwise.
    def has_key(self, key):
        return self.A == key or self.B == key or self.C == key

    # Inserts a new key into the proper location in this node.
    # Precondition: This node is a leaf and has 2 or fewer keys
    def insert_key(self, key):
        if key < self.A:
            self.C = self.B
            self.B = self.A
            self.A = key
        elif self.B == None or key < self.B:
            self.C = self.B
            self.B = key
        else:
            self.C = key

    # Inserts a new key into the proper location in this node, and
    # sets the children on either side of the inserted key.
    # Precondition: This node has 2 or fewer keys
    def insert_key_with_children(self, key, leftChild, rightChild):
        if key < self.A:
            self.C = self.B
            self.B = self.A
            self.A = key
            self.right = self.middle2
            self.middle2 = self.middle1
            self.middle1 = rightChild
            self.left = leftChild
        elif self.B == None or key < self.B:
            self.C = self.B
            self.B = key
            self.right = self.middle2
            self.middle2 = rightChild
            self.middle1 = leftChild
        else:
            self.C = key
            self.right = rightChild
```

```
            self.middle2 = leftChild

    # Returns True if this node is a leaf, False otherwise.
    def is_leaf(self):
        return self.left == None

    # Returns the child of this node that would be visited next in the
    # traversal to search for the specified key
    def next_node(self, key):
        if key < self.A:
            return self.left
        elif self.B == None or key < self.B:
            return self.middle1
        elif self.C == None or key < self.C:
            return self.middle2
        return self.right

    # Removes key A, B, or C from this node, if key_index is 0, 1, or 2,
    # respectively. Other keys and children are shifted as necessary.
    def remove_key(self, key_index):
        if key_index == 0:
            self.A = self.B
            self.B = self.C
            self.C = None
            self.left = self.middle1
            self.middle1 = self.middle2
            self.middle2 = self.right
            self.right = None
        elif key_index == 1:
            self.B = self.C
            self.C = None
            self.middle2 = self.right
            self.right = None
        elif key_index == 2:
            self.C = None
            self.right = None

    # Removes and returns the rightmost child. Two possible cases exist:
    # 1. If this node has a right child, right is set to None, and the
    #    previous right value is returned.
    # 2. Else if this node has a middle2 child, middle2 is set to None, and
    #    the previous right value is returned.
    # 3. Otherwise no action is taken, and None is returned.
    # No keys are changed in any case.
    def remove_rightmost_child(self):
        removed = None
        if self.right != None:
            removed = self.right
            self.right = None
        elif self.middle2 != None:
            removed = self.middle2
            self.middle2 = None
        return removed

    # Removes and returns the rightmost key. Three possible cases exist:
    # 1. If this node has 3 keys, C is set to None and the previous C value
is returned.
    # 2. If this node has 2 keys, B is set to None and the previous B value
is returned.
    # 3. Otherwise no action is taken and None is returned.
    # No children are changed in any case.
```


©zyBooks 11/20/23 11:12 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
    # No children are changed in any case.
    def remove_rightmost_key(self):
        removed = None
        if self.C != None:
            removed = self.C
            self.C = None
        elif self.B != None:
            removed = self.B
            self.B = None
        return removed

    # Sets the left, middle1, middle2, or right child if the child_index
    # argument is 0, 1, 2, or 3, respectively.
    # Does nothing if the child_index argument is < 0 or > 3.
    def set_child(self, child, child_index):
        if child_index == 0:
            self.left = child
        elif child_index == 1:
            self.middle1 = child
        elif child_index == 2:
            self.middle2 = child
        elif child_index == 3:
            self.right = child

    # Sets this node's A, B, or C key if the key_index argument is 0, 1, or
    # 2, respectively.
    # Does nothing if the key_index argument is < 0 or > 2.
    def set_key(self, key, key_index):
        if key_index == 0:
            self.A = key
        elif key_index == 1:
            self.B = key
        elif key_index == 2:
            self.C = key
```


PARTICIPATION ACTIVITY 13.6.2: Other Node234 methods.

1) append_key_and_child() may set the middle1, middle2, or right members of the node.
   - ○ True
   - ○ False

2) count_keys() returns a value in the range [1, 3].
   - ○ True
   - ○ False

3) insert_key(), insert_key_with_children(), and remove_key() may shift existing keys in the node.

- True
- False

4) remove_rightmost_child() sets at most 1 child to None, and remove_rightmost_key() sets at most 1 key to None.

- True
- False

5) The range of valid indices for set_child()'s child_index argument is [1, 3].

- True
- False

6) set_key() compares the key argument to the node's existing keys, and inserts the key in the proper spot.

- True
- False

Tree234 class - initialization, search, and insertion

The Tree234 class uses the Node234 class to implement a 2-3-4 tree. The __init__() method initializes the tree by setting the root member to None. The search() method takes a key as an argument, and returns the node in the tree containing the key, or None if the key is not found. Searching is implemented recursively with the search_recursive() helper method.

The insert() method inserts a new key into the tree. During insertion, full nodes are preemptively split using the split() method. The split() method splits a full node, moving the middle key up into the parent node.

Figure 13.6.3: Tree234 class __init__, insert, search, search_recursive, and split methods.

©zyBooks 11/20/23 11:12 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
class Tree234:
    # Initializes the tree with the root node reference set to None.
    def __init__(self):
        self.root = None

    # Inserts a new key into this tree, provided the tree doesn't already
    # contain the same key.
    def insert(self, key, node = None, node_parent = None):
        # Special case for empty tree
        if self.root == None:
            self.root = Node234(key)
            return self.root

        # If the node argument is null, recursively call with root
        if node == None:
            return self.insert(key, self.root, None)

        # Check for duplicate key
        if node.has_key(key):
            # Duplicate keys are not allowed
            return None

        # Preemptively split full nodes
        if node.C != None:
            node = self.split(node, node_parent)

        if not node.is_leaf():
            if key < node.A:
                return self.insert(key, node.left, node)
            elif node.B == None or key < node.B:
                return self.insert(key, node.middle1, node)
            elif node.C == None or key < node.C:
                return self.insert(key, node.middle2, node)
            else:
                return self.insert(key, node.right, node)

        # key can be inserted into leaf node
        node.insert_key(key)
        return node

    # Searches this tree for the specified key. If found, the node
containing
    # the key is returned. Otherwise None is returned.
    def search(self, key):
        return self.search_recursive(key, self.root)

    # Recursive helper method for search.
    def search_recursive(self, key, node):
        if node == None:
            return None

        # Check if the node contains the key
        if node.has_key(key):
            return node

        # Recursively search the appropriate subtree
        if key < node.A:
            return self.search_recursive(key, node.left)
        elif node.B == None or key < node.B:
            return self.search_recursive(key, node.middle1)
```

```
        elif node.C == None or key < node.C:
            return self.search_recursive(key, node.middle2)
        return self.search_recursive(key, node.right)

    # Splits a full node, moving the middle key up into the parent node.
    def split(self, node, node_parent):
        split_left = Node234(node.A, node.left, node.middle1)
        split_right = Node234(node.C, node.middle2, node.right)
        if node_parent is not None:
            node_parent.insert_key_with_children(node.B, split_left,
split_right)
        else:
            # Split root
            node_parent = Node234(node.B, split_left, split_right)
            self.root = node_parent

        return node_parent
```


PARTICIPATION ACTIVITY 13.6.3: Tree234 class insert(), search(), search_recursive(), and split() methods.

1) The insert() method is recursive. Which is one of the base cases?
   - The node argument is None.
   - The node contains the key being inserted.
   - The node is not a leaf.

2) The search() method calls search_recursive(), passing _____ as the node argument.
   - None
   - self.root
   - key

3) The split() method always allocates a minimum of _____ nodes.
   - 1
   - 2
   - 3

Tree234 class removal methods

The remove() method removes the specified key from the tree. Each non-root node with 1 key that is encountered during removal is preemptively merged using the merge() method. The merge() method

attempts to take a key from an adjacent sibling using the rotate_left() or rotate_right() methods. If no adjacent siblings have more than 1 key, then the fuse() method is called to fuse a parent and two child nodes into one node. The fuse_root() method implements the special case of root fusion, and is called by fuse() in the case when the root must be fused.

When removing a key from an internal node, remove() finds and stores the key's successor, recursively removes the successor, and then replaces the key to remove with the stored successor key. The get_min_key() is called on the subtree to the right of the key being removed and gets the minimum key in a subtree to obtain the key's successor. After recursively removing the successor, the key_swap() method is used to replace the key to remove with the key's successor.

Figure 13.6.4: Tree234 class removal-oriented methods.

©zyBooks 11/20/23 11:12 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
class Tree234:
    # Fuses a parent node and two children into one node.
    # Precondition: Each of the three nodes must have one key each.
    def fuse(self, parent, left_node, right_node):
        if parent is self.root and parent.count_keys() == 1:
            return self.fuse_root()

        left_node_index = parent.get_child_index(left_node)
        middle_key = parent.get_key(left_node_index)
        fused_node = Node234(left_node.A)
        fused_node.B = middle_key
        fused_node.C = right_node.A
        fused_node.left = left_node.left
        fused_node.middle1 = left_node.middle1
        fused_node.middle2 = right_node.left
        fused_node.right = right_node.middle1
        key_index = parent.get_key_index(middle_key)
        parent.remove_key(key_index)
        parent.set_child(fused_node, key_index)
        return fused_node

    # Fuses the tree's root node with the root's two children.
    # Precondition: Each of the three nodes must have one key each.
    def fuse_root(self):
        old_left = self.root.left
        old_middle1 = self.root.middle1
        self.root.B = self.root.A
        self.root.A = old_left.A
        self.root.C = old_middle1.A
        self.root.left = old_left.left
        self.root.middle1 = old_left.middle1
        self.root.middle2 = old_middle1.left
        self.root.right = old_middle1.middle1
        return self.root

    # Searches for, and returns, the minimum key in a subtree
    def get_min_key(self, node):
        current = node
        while current.left != None:
            current = current.left
        return current.A

    # Finds and replaces one key with another. The replacement key must
    # be known to be a key that can be used as a replacement without
violating
    # any of the 2-3-4 tree rules.
    def key_swap(self, node, existing, replacement):
        if node == None:
            return False

        key_index = node.get_key_index(existing)
        if key_index == -1:
            next = node.next_node(existing)
            return self.key_swap(next, existing, replacement)

        if key_index == 0:
            node.A = replacement
        elif key_index == 1:
            node.B = replacement
        else:
```

```
            node.C = replacement

        return True

    # Rotates or fuses to add 1 or 2 additional keys to a node with 1 key.
    def merge(self, node, node_parent):
        # Get references to node's siblings
        node_index = node_parent.get_child_index(node)
        left_sibling = node_parent.get_child(node_index - 1)
        right_sibling = node_parent.get_child(node_index + 1)

        # Check siblings for a key that can be transferred
        if left_sibling != None and left_sibling.count_keys() >= 2:
            self.rotate_right(left_sibling, node_parent)
        elif right_sibling != None and right_sibling.count_keys() >= 2:
            self.rotate_left(right_sibling, node_parent)
        else: # fuse
            if left_sibling == None:
                node = self.fuse(node_parent, node, right_sibling)
            else:
                node = self.fuse(node_parent, left_sibling, node)

        return node

    # Finds and removes the specified key from this tree.
    def remove(self, key):
        # Special case for tree with 1 key
        if self.root.is_leaf() and self.root.count_keys() == 1:
            if self.root.A == key:
                self.root = None
                return True
            return False

        current_parent = None
        current = self.root
        while current != None:
            # Merge any non-root node with 1 key
            if current.count_keys() == 1 and current is not self.root:
                current = self.merge(current, current_parent)

            # Check if current node contains key
            key_index = current.get_key_index(key)
            if key_index != -1:
                if current.is_leaf():
                    current.remove_key(key_index)
                    return True

                # The node contains the key and is not a leaf, so the key
is
                # replaced with the successor
                tmp_child = current.get_child(key_index + 1)
                tmp_key = self.get_min_key(tmp_child)
                self.remove(tmp_key)
                self.key_swap(self.root, key, tmp_key)
                return True

            # Current node does not contain key, so continue down tree
            current_parent = current
            current = current.next_node(key)

        # key not found
```

```
        return False

    def rotate_left(self, node, node_parent):
        # Get the node's left sibling
        node_index = node_parent.get_child_index(node)
        left_sibling = node_parent.get_child(node_index - 1)

        # Get the key from the parent that will be copied into the left
sibling
        key_for_left_sibling = node_parent.get_key(node_index - 1)

        # Append the key to the left sibling
        left_sibling.append_key_and_child(key_for_left_sibling, node.left)

        # Replace the parent's key that was appended to the left sibling
        node_parent.set_key(node.A, node_index - 1)

        # Remove key A and left child from node
        node.remove_key(0)

    def rotate_right(self, node, node_parent):
        # Get the node's right sibling
        node_index = node_parent.get_child_index(node)
        right_sibling = node_parent.get_child(node_index + 1)

        # Get the key from the parent that will be copied into the right
sibling
        key_for_right_sibling = node_parent.get_key(node_index)

        # Shift key and child references in right sibling
        right_sibling.C = right_sibling.B
        right_sibling.B = right_sibling.A
        right_sibling.right = right_sibling.middle2
        right_sibling.middle2 = right_sibling.middle1
        right_sibling.middle1 = right_sibling.left

        # Set key A and the left child of right_sibling
        right_sibling.A = key_for_right_sibling
        right_sibling.left = node.remove_rightmost_child()

        # Replace the parent's key that was prepended to the right sibling
        node_parent.set_key(node.remove_rightmost_key(), node_index)
```


PARTICIPATION ACTIVITY 13.6.4: Tree234 class removal methods.

1) Which is NOT a precondition of the fuse() method?

- parent, left_node, and right_node must each have one key.
- left_node and right_node must be adjacent siblings.
- The parent node must not be the root.

2) get_min_key() may return a node's _____.

- A key only
- A or B key
- A, B, or C key

3) If a valid key_swap() call passes 22 as the existing argument and 32 as the replacement argument, then the tree must not contain any keys _____.

- less than 22
- greater than 22 but less than 32
- greater than 32

4) merge() calls fuse() only if both siblings _____.

- are None or have one key
- have two or more keys
- have three keys

5) If the key is not found in the tree, remove() _____.

- fails after accessing a member of an object set to None
- returns False without altering the tree in any way
- returns False without removing any keys

6) Both rotate_left() and rotate_right() have a precondition that _____.

- node has a left sibling
- node_parent is not None
- node is not a leaf

zyDE 13.6.1: Exploring the Tree234 class.

The following program inserts some nodes into a 2-3-4 tree and then removes some nod
The first line of input is a space-separated list of numbers to insert. The second line of inp
a space-separated list of numbers to remove. Try the following activities:

- Try several reorderings of the same list of numbers. Ex: Sort the input numbers in ascending order, sort the input numbers in descending order, and randomize the orc input numbers. Does the height of the tree change based on insertion order?
- How many times will split be called when adding the numbers? Insert a print statem in the split() method in the BTree.py file to display each time a node is split.
- How many times will merge be called when removing the numbers? Insert a print statement in the merge() method in the BTree.py file to display each time a node is merged.

Current file: main.py Load default templat

```
1  from BTree import Tree234
2
3  user_values = input('Enter insert values with spaces between: ')
4  print()
5
6  # Create a 2-3-4 tree and add the values
7  tree = Tree234()
8  for value in user_values.split():
9      tree.insert(int(value))
10
11 # Display the height after all inserts are complete.
12 print("2-3-4 tree with " + str(len(tree)) + " keys has height " + str
13
14 # Read user input to get a list of values to remove
15 user_values = input('Enter remove values with spaces between: ')
```



```
15 42 77 18 90 21 36 19 89 47 28 39 55 73 99
15 47 40 19 55 77 58 89
```

Run

13.7 LAB: Iterable 2-3-4 tree

This section has been set as optional by your instructor.

This section's content is not available for print.