12.1 B-trees

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 | 12.1.1: Order 3 B-trees.

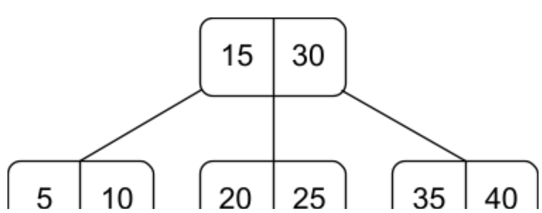
Animation captions:

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 | 12.1.2: Validity of order 3 B-trees.

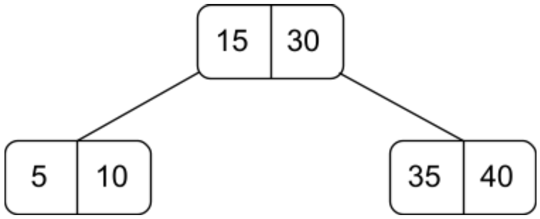
Determine which of the following are valid order 3 B-trees.

1)



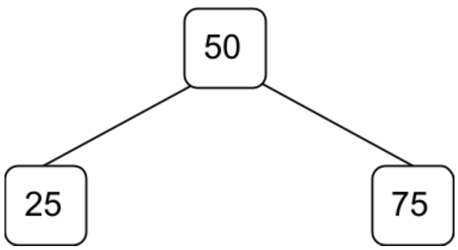
Valid

Invalid

2) 

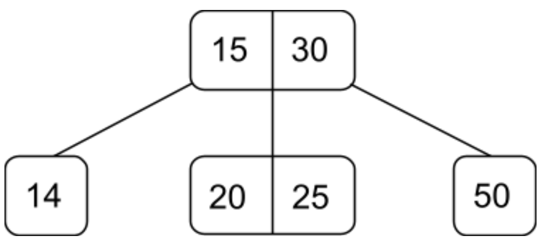
Valid

Invalid

3) 

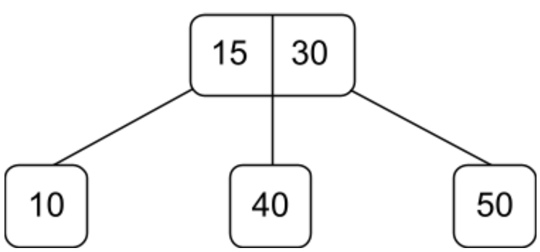
Valid

Invalid

4) 

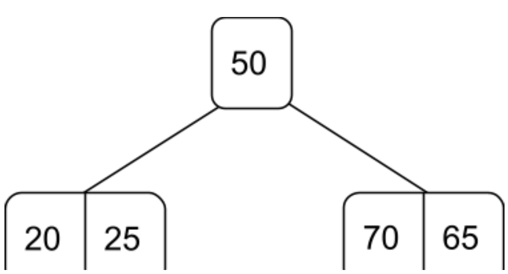
Valid

Invalid

5) 

Valid

Invalid

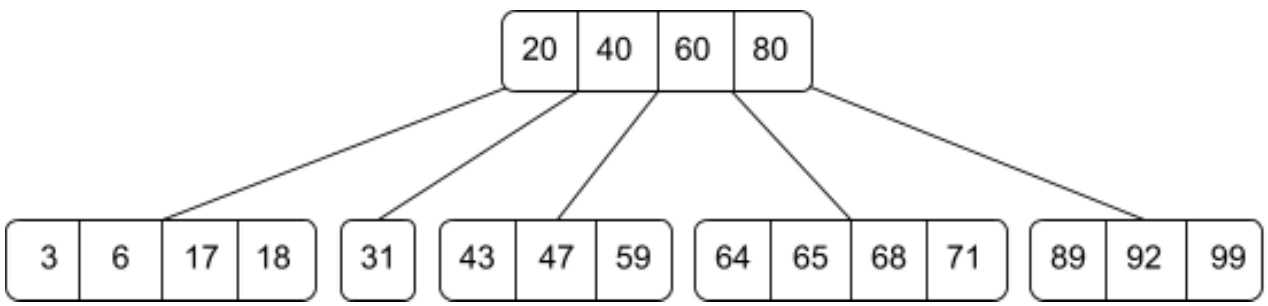
6) 

- Valid
- Invalid

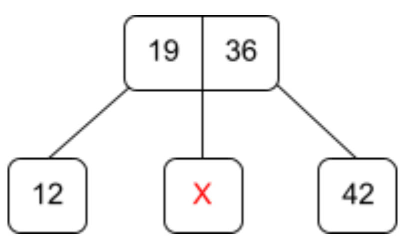
Higher order B-trees

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 12.1.1: A valid order 5 B-tree.



PARTICIPATION ACTIVITY 12.1.3: 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 12.1.1: 2-3-4 tree internal nodes.

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

PARTICIPATION ACTIVITY 12.1.4: 2-3-4 tree properties.

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

©zyBooks 12/08/22 22:17 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

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?

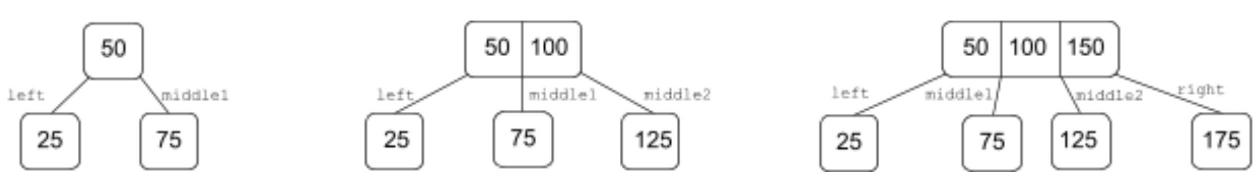
Check Show answer

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 12.1.1: 2-3-4 child subtree labels.



PARTICIPATION ACTIVITY 12.1.5: 2-3-4 tree nodes.

©zyBooks 12/08/22 22:17 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

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

12.2 2-3-4 tree search algorithm

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 12.2.1: 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 12.2.1: 2-3-4 tree search algorithm.

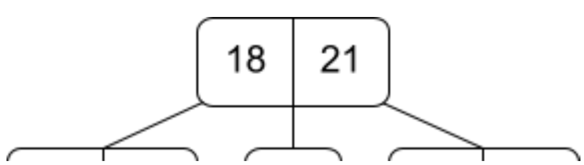
Animation content:

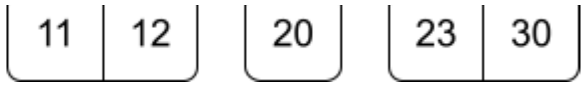
undefined

Animation captions:

1. Search for 70 starts at the root node.
2. node is not null, so the search will check all keys in the current node. The keys 25 and 50 in the root node are compared to 70, and no match is found.
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 is made.
5. node is not null, so 70 is compared with the node's A key. A match is found, and the node is returned.

PARTICIPATION ACTIVITY 12.2.2: 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?

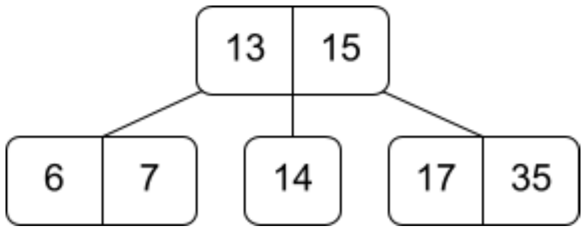
- Null
- Root's left child
- Root's middle1 child
- Root's middle2 child

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 | 12.2.3: 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

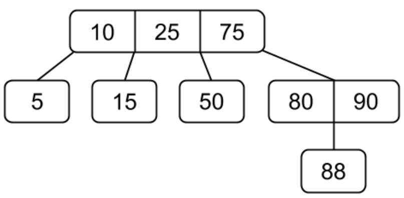
CHALLENGE ACTIVITY | 12.2.1: 2-3-4 tree search algorithm.

422352.2723990.qx3zqy7

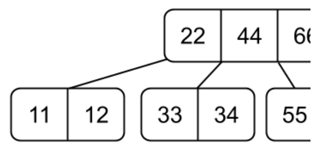
Start

Select all valid 2-3-4 trees.

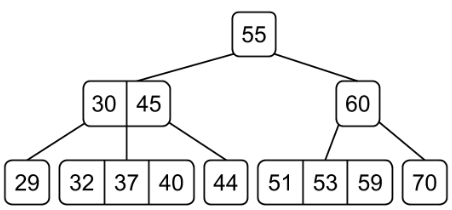
☐ A



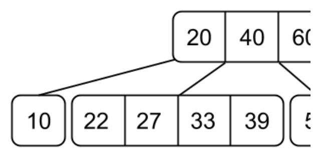
☐ D



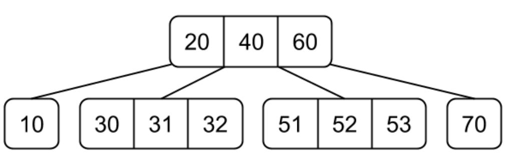
☐ B



☐ E



☐ C



©zyBooks 12/08/22 22:17 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

1	2	3	4

Check Next

12.3 2-3-4 tree insert algorithm

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 12.3.1: 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 12.3.2: 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 12.3.3: 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 12.3.1: 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 12.3.4: 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) All internal nodes are split in the same way.

True

False

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

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 12.3.1: 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 12.3.5: 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 12.3.6: 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 12.3.7: 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?

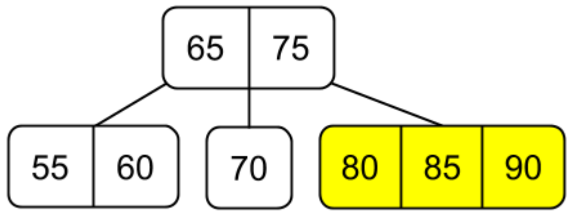
- When a value is inserted into a leaf, the tree will always grow in height.
- When splitting a leaf node, the tree will always grow in height.
- When splitting the root node, the tree will always grow in height.

- Any insertion that does NOT involve splitting any nodes will cause the tree to grow in height.

CHALLENGE ACTIVITY | 12.3.1: 2-3-4 tree insert algorithm.

422352.2723990.qx3zqy7

Start



The node (80, 85, 90) 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

12.4 2-3-4 tree rotations and fusion

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 | 12.4.1: 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 | 12.4.2: 2-3-4 tree rotations.

1) A rotation on a node changes the set of keys in of 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

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.

©zyBooks 12/08/22 22:17 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

Figure 12.4.1: 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 12.4.3: Utility functions for rotations.

If unable to drag and drop, refresh the page.

BTreeRemoveKey **BTreeGetLeftSibling** **BTreeAddKeyAndChild**

BTreeGetRightSibling **BTreeSetParentKeyLeftOfChild**

BTreeGetParentKeyLeftOfChild

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 | 12.4.4: 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 | 12.4.5: 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 | 12.4.6: Root fusion.

Animation content:

©zyBooks 12/08/22 22:17 1361995
John Farrell
COLOSTATECS165WakefieldFall2022

`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 | 12.4.7: 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 | 12.4.8: 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 | 12.4.9: 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

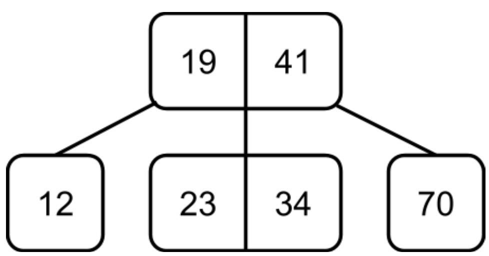
4) When the parent node has a key removed, how many child pointers must be assigned with new values?

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

CHALLENGE ACTIVITY 12.4.1: 2-3-4 tree rotations and fusion.

422352.2723990.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

12.5 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 | 12.5.1: 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 | 12.5.2: Merge algorithm.

If unable to drag and drop, refresh the page.

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

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 12.5.1: BTreeGetMinKey pseudocode.

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

Figure 12.5.2: 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 12.5.3: 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 12.5.4: 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 12.5.3: 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 | 12.5.4: 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 | 12.5.5: BTreeRemove algorithm: non-leaf case.

Animation content:

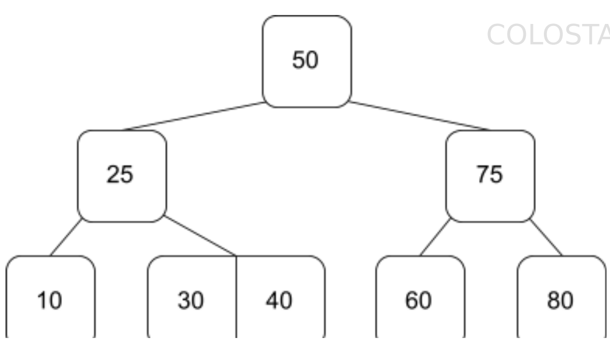
undefined

Animation captions:

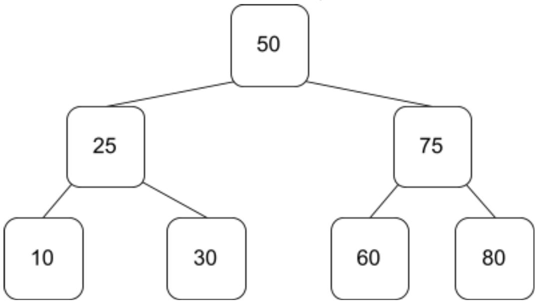
1. When deleting 60, the process is more complex due to the key being found in an internal node.
2. The key 62 is a suitable replacement for 60, but 62 must be recursively removed before the swap.
3. After the recursive removal completes, 60 is replaced with 62.

PARTICIPATION ACTIVITY | 12.5.6: BTreeRemove algorithm.

Tree before removal:



1) The tree after removing 40 is:



- True
- False

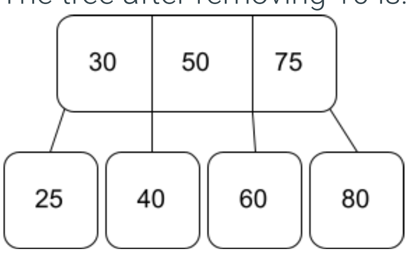
2) Calling BTreeRemove to remove any key in this tree would cause at least 1 node to be merged.

- 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 12.5.7: 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

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.

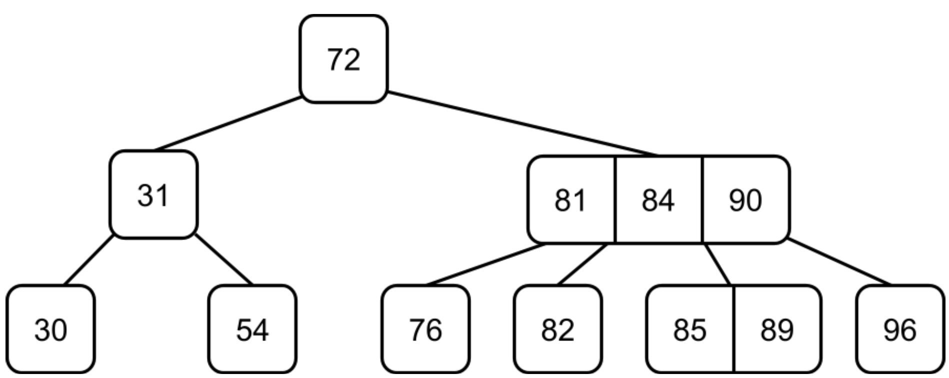
5) Removal pseudocode has the check: "if (keyIndex != -1)". What is implied about the node pointed to by cur when the condition evaluates to true?

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

CHALLENGE ACTIVITY 12.5.1: 2-3-4 tree removal.

422352.2723990.qx3zqy7

Start



A merge occurs on node (31).
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⇢left⇢middle2	

1	2	3	4

Check Next

12.6 Lab 22 - Building a B+ Tree

Module 11: Lab 22: Building a B+ Tree

This lab includes the following .java files:

L22/
└── BPlusTree.java
└── BPlusTest.java*

* This is the **main** class in zyBooks and the only one that will run.

Note: you may download BPlusTree.java below and code in your preferred environment. It has its own main that is similar to BPlusTest.java. (BPlusTest has more than one tree, so it requires an input of '1' ,'2', or '3' in zyBooks.)

This is one of the most complicated data structures you will be asked to implement in this course. This is also the only lab in this module, since it is quite an involved and difficult one.

The most fundamental of the B+ operations is the insert, which is what this lab will focus on. You will essentially be implementing the insert function from scratch on a simplified B+ tree. Don't get too comfortable - this is a deceptively complicated assignment.

This lab assumes you are already familiar with the basics of B+ trees. Be sure to watch the lecture videos first.

Our B+ Implementation

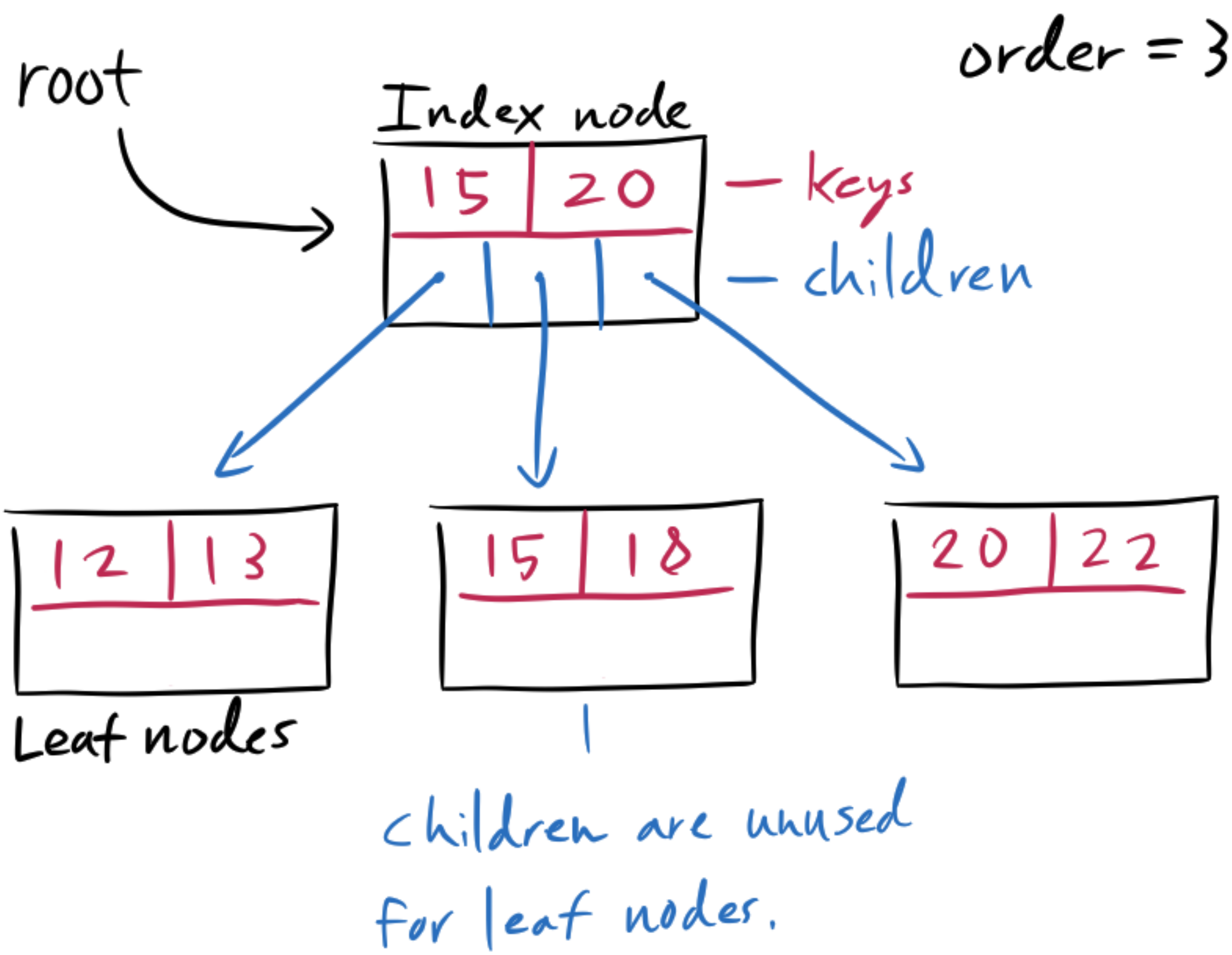
Let's take a quick tour of the code in the lab, which will explain how our B+ tree is set up in Java.

You should notice the Node class right away. For the sake of simplicity, this Node class represents both index nodes *and* leaf nodes. To help you tell when a Node is being used as an index or as a leaf, it holds a boolean called `isLeaf`. Whenever you make a Node, you will have to specify the type by passing either `Type.LEAF` or `Type.INDEX` to the constructor, which will set this boolean accordingly.

Remember that leaf nodes are where the actual data is, and index nodes just contain keys to help you find the data. The keys are a lot like road signs pointing you in the right direction at each intersection - they're not your destination, but they let you know where to go to get there. The leaf nodes, or more specifically, the data in them, are your actual destination.

Since the same class is being used to represent leaf and index nodes, note that the index node's keys and the leaf node's data are both stored in the `keys` ArrayList. Don't let that confuse you!

For a little more clarity, this is what a simple B+ tree might look like, with some of the parts labelled:

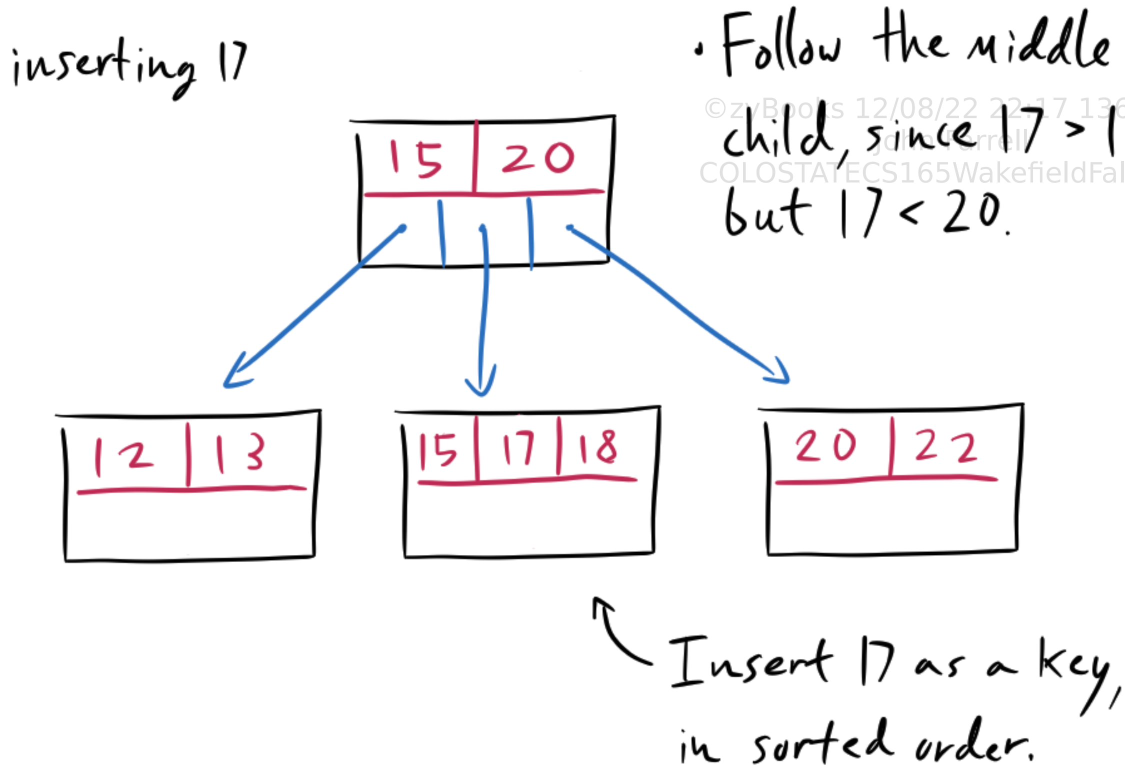


This is a tree of order 3, meaning it can a maximum of 2 keys in each node and a minimum of 1. In the code, the order of the tree is stored as part of the `BPlusTree` class and set in the constructor, meaning you have access to it everywhere.

Notice how each node has a list of keys and a list of children. For leaf nodes, the list of children is unused - for index nodes, the list of children contains references to each of the child nodes.

Keeping the Invariants

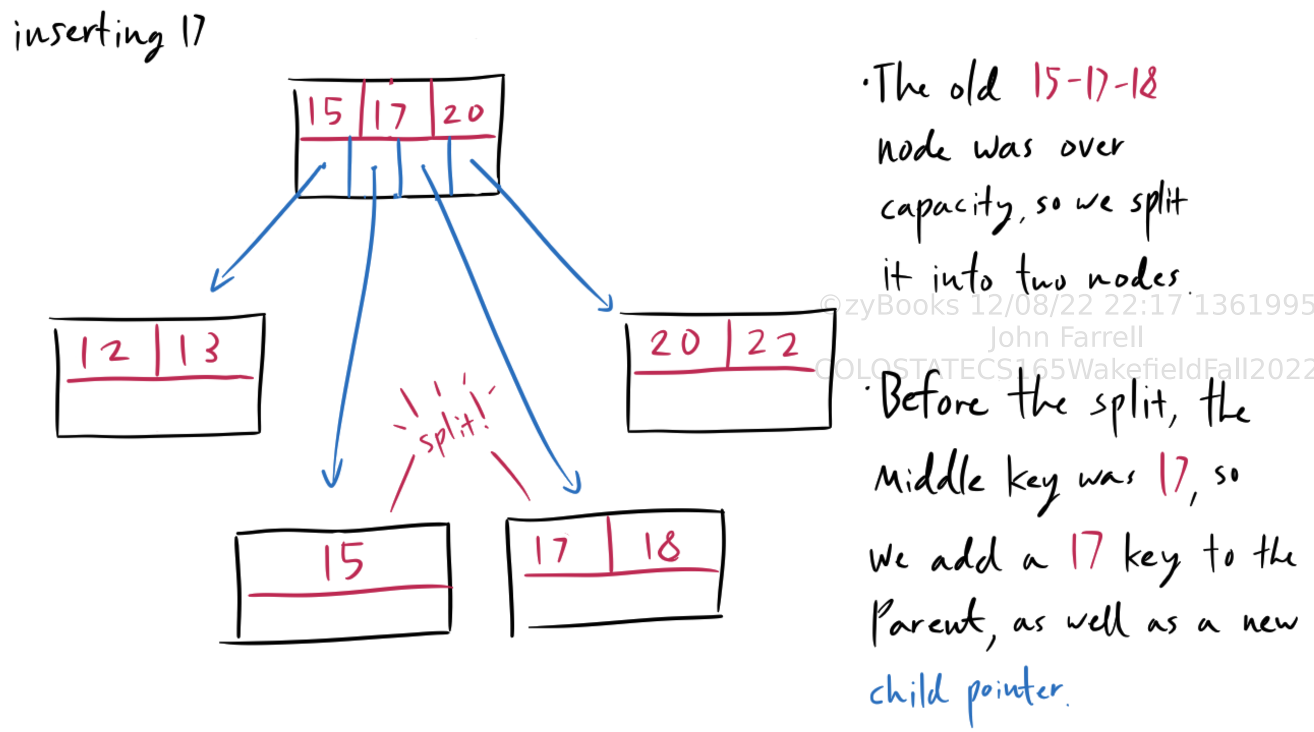
You probably already know the basic idea of adding to a B+ tree, but I'd like to walk through a fairly complicated example step-by-step so we have a better idea of how to actually program it. While we do this, I'll point out some considerations for how these steps might look in Java. Taking the tree above, let's insert the key 17, and watch the resulting chaos unfold.



The first thing we need to do is figure out *where* the 17 should go. Our strategy will be to recursively search through the tree, following the correct child pointers until we arrive at a leaf node.

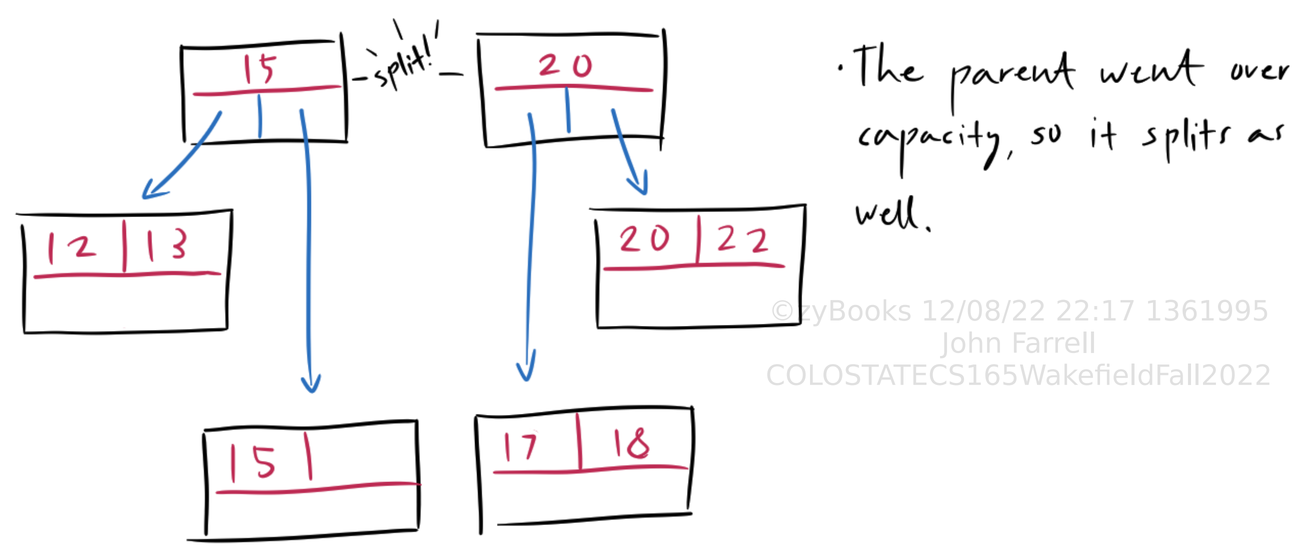
Starting at the root, we find the root is not a leaf node, so we turn to examining the keys. There are two keys: 15 and 20. This means that the first child pointer will bring us to leaf nodes containing values below 15 - not good. The second child pointer will bring us to leaf nodes containing values greater than or equal to 15, but less than 20 - hey, that sounds like us.

Following that child pointer, we arrive at a leaf node. Awesome! Let's insert our 17 in sorted order. Since the keys are stored in array lists, this is as simple as saying `node.keys.add(17)` and then `node.keys.sort(null)` (or perhaps `Collections.sort(node.keys)`).



If this insert didn't bring the node above its max capacity of 2, we would be done. However, with the 17, the node exceeded its max capacity, and it must be split into two nodes. A new leaf node is created, and the lower half of the original node's keys are moved into the new one, rounded down. In this case, we removed the 15, created a new leaf node, and shuffled the 15 into that new node.

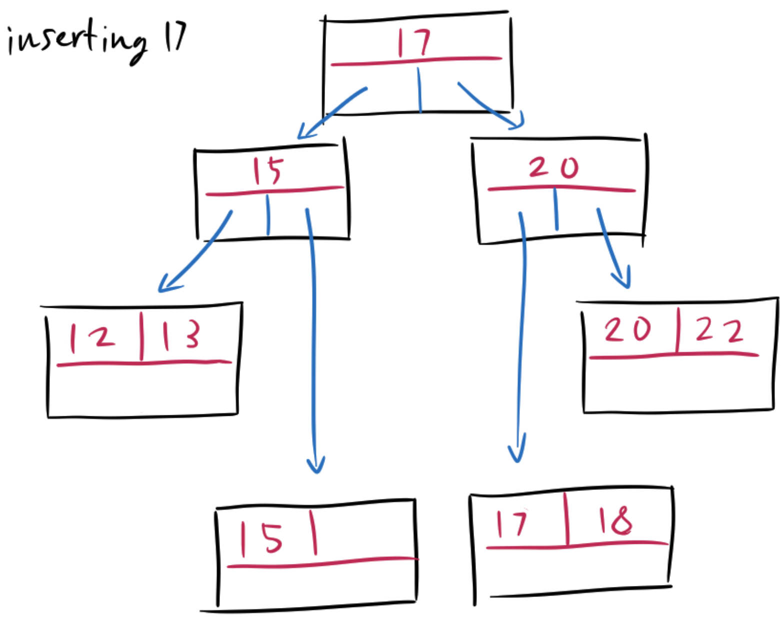
Before the split, the middle key of the node was 17. Therefore, we need to *raise* that key up to the parent node, and insert it in sorted order. We also need to add a new child pointer in the parent for the new node we just created. The new child pointer needs to go *just before* the one we originally followed, since the original child pointer is now pointing to the "larger" of the two nodes, and the new child pointer is pointing to the "smaller" of the two.



Of course, you may have noticed the *parent* is now over capacity, so it *also* has to split into two nodes. A similar procedure is in order; we split the keys and the child pointers in half, giving one half to a freshly-created index node and leaving the other half in the original node. However, this time, **the middle key is not kept in either of the nodes**. It still moves up to the parent node and is inserted in sorted order, but it doesn't stay in the left or right split nodes. This is the only difference between splitting and index and a leaf node, but it's a very important one.

17 was the middle key in this case, so it leaves both split nodes and moves up to the parent... node? Well, we just split the root node, which by definition has no parent! What now?

We have to tend to a special case here, which is that **when the root node splits, we must create a new root**. We wanted to insert 17, so our new root has 17 as its only key, and two child pointers pointing to both of the split nodes.



· 17 was the middle element before the split, so it moves up. But since there's no node to move to, a new one is created, which becomes the new root.

· The insert is now done.

Finally, we're done. Order and balance have been restored to the tree, and 17 is now part of the bottom-layer leaf nodes.

Hopefully, this gives you some sense of how to approach the insert. Be careful, deliberate, and plan out your approach. I would highly recommend doing this recursively with a helper method, as this makes it easier to keep track of each node's parent (when a function working on a node returns, it returns into another instance of the function that was working on the node's parent - so if your recursive insert returns something, it can essentially communicate with its parent!)

Wrapping Up

Inside the lab code is the `void insert(int key)` function. When someone calls this, it should add the key to the tree and return nothing. **You may write as many helper functions or other methods as you see fit, but do not modify any code given to you, and do not add to or modify the Node class or the toString() methods.**

You should always store the root of the tree in the `root` variable in the BPlusTree class. The root is initialized to an empty leaf node - your first few inserts should add to this leaf node, which should split once it becomes too full, creating a new index node which then becomes the root. This means small trees will be just a single leaf node.

A recursive toString method is given to the BPlusTree which shows the entire tree in text. Print the tree to help you see what it's doing as you add to it.

The main method contains some simple test code. In BPlusTest, there are three different trees you will be tested on. To test tree 1, type in a '1' for your input, and to test tree 2, type in a '2' for your input, and type a '3' for tree 3. A successful implementation will produce this output when you test tree 1:


```
INDEX NODE
key 0 = 15
child 0 =
  INDEX NODE
  key 0 = 5, key 1 = 10
  child 0 =
    LEAF NODE
    key 0 = 0, key 1 = 1, key 2 = 2, key 3 = 4
  child 1 =
    LEAF NODE
    key 0 = 5, key 1 = 7, key 2 = 8, key 3 = 9
  child 2 =
    LEAF NODE
    key 0 = 10, key 1 = 11, key 2 = 13
child 1 =
  INDEX NODE
  key 0 = 18, key 1 = 20
  child 0 =
    LEAF NODE
    key 0 = 15, key 1 = 16, key 2 = 17
  child 1 =
    LEAF NODE
    key 0 = 18, key 1 = 19
  child 2 =
    LEAF NODE
    key 0 = 20, key 1 = 23, key 2 = 25, key 3 = 26
```


We have added some comments in the code to help you understand the algorithms, and we have some helper methods you may want to use in your code as well. Keep in mind that we are always available to answer your questions. We are happy to help with anything. Good luck.

422352.2723990.qx3zqy7

LAB ACTIVITY | 12.6.1: Lab 22 - Building a B+ Tree 0 / 7

Downloadable files

BPlusTree.java Download

Current file: BPlusTree.java Load default template...

```
import java.util.*;

public class BPlusTree {
    enum Type {
        LEAF,
        INDEX
    }

    /* Do not change this class! */
    private static class Node {
        ArrayList<Integer> keys;
        ArrayList<Node> children;
        boolean isLeaf;

        Node(Type nt) {
            isLeaf = nt == Type.LEAF;

```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **BPlusTree.java** (Your program) →

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```