9.1 Searching and algorithms

Algorithms

An ***algorithm*** is a sequence of steps for accomplishing a task. ***Linear search*** is a search algorithm that starts from the beginning of a list, and checks each element until the search key is found or the end of the list is reached.

PARTICIPATION ACTIVITY | 9.1.1: Linear search algorithm checks each element until key is found.

Animation content:

`undefined`

Animation captions:

1. Linear search starts at first element and searches elements one-by-one.
2. Linear search will compare all elements if the search key is not present.

Figure 9.1.1: Linear search algorithm.

```
LinearSearch(numbers, numbersSize, key) {
   i = 0

   for (i = 0; i < numbersSize; ++i) {
      if (numbers[i] == key) {
         return i
      }
   }

   return -1 // not found
}

main() {
   numbers = {2, 4, 7, 10, 11, 32, 45, 87}
   NUMBERS_SIZE = 8
   i = 0
   key = 0
   keyIndex = 0

   print("NUMBERS: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   print("Enter a value: ")
   key = getIntFromUser()

   keyIndex = LinearSearch(numbers, NUMBERS_SIZE, key)

   if (keyIndex == -1) {
      printLine(key + " was not found.")
   }
   else {
      printLine("Found " + key + " at index " + keyIndex + ".")
   }
}
```



```
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 10
Found 10 at index 3.
...
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 17
17 was not found.
```


PARTICIPATION ACTIVITY 9.1.2: Linear search algorithm execution.

Given list: (20, 4, 114, 23, 34, 25, 45, 66, 77, 89, 11).

1) How many list elements will be compared to find 77 using linear search?

Check Show answer

2) How many list elements will be checked to find the value 114 using linear search?

Check Show answer

3) How many list elements will be checked if the search key is not found using linear search?

Check Show answer

Algorithm runtime

An algorithm's ***runtime*** is the time the algorithm takes to execute. If each comparison takes 1 μs (1 microsecond), a linear search algorithm's runtime is up to 1 s to search a list with 1,000,000 elements, 10 s for 10,000,000 elements, and so on. Ex: Searching Amazon's online store, which has more than 200 million items, could require more than 3 minutes.

An algorithm typically uses a number of steps proportional to the size of the input. For a list with 32 elements, linear search requires at most 32 comparisons: 1 comparison if the search key is found at index 0, 2 if found at index 1, and so on, up to 32 comparisons if the search key is not found. For a list with N elements, linear search thus requires at most N comparisons. The algorithm is said to require "on the order" of N comparisons.

PARTICIPATION ACTIVITY | 9.1.3: Linear search runtime.

1) Given a list of 10,000 elements, and if each comparison takes 2 μs, what is the fastest possible runtime for linear search?

μs

Check Show answer

2) Given a list of 10,000 elements, and if each comparison takes 2 μs, what is the longest possible runtime for linear search?

μs

Check Show answer

9.2 Binary search

Linear search vs. binary search

Linear search may require searching all list elements, which can lead to long runtimes. For example, searching for a contact on a smartphone one-by-one from first to last can be time consuming. Because a contact list is sorted, a faster search, known as binary search, checks the middle contact first. If the desired contact comes alphabetically before the middle contact, binary search will then search the first half and otherwise the last half. Each step reduces the contacts that need to be searched by half.

PARTICIPATION ACTIVITY | 9.2.1: Using binary search to search contacts on your phone.

Animation captions:

1. A contact list stores contacts sorted by name. Searching for Pooja using a binary search starts by checking the middle contact.
2. The middle contact is Muhammad. Pooja is alphabetically after Muhammad, so the binary search only searches the contacts after Muhammad. Only half the contacts now need to be searched.
3. Binary search continues by checking the middle element between Muhammad and the last contact. Pooja is before Sharod, so the search continues with only those contacts between Muhammad and Sharod, which is one fourth of the contacts.
4. The middle element between Muhammad and Sharod is Pooja. Each step reduces the number of contacts to search by half.

PARTICIPATION ACTIVITY 9.2.2: Using binary search to search a contact list.

A contact list is searched for Bob.
Assume the following contact list: Amy, Bob, Chris, Holly, Ray, Sarah, Zoe

1) What is the first contact searched?

Check Show answer

2) What is the second contact searched?

Check Show answer

Binary search algorithm

Binary search is a faster algorithm for searching a list if the list's elements are sorted and directly accessible (such as an array). Binary search first checks the middle element of the list. If the search key is found, the algorithm returns the matching location. If the search key is not found, the algorithm repeats the search on the remaining left sublist (if the search key was less than the middle element) or the remaining right sublist (if the search key was greater than the middle element).

PARTICIPATION ACTIVITY | 9.2.3: Binary search efficiently searches sorted list by reducing the search space by half each iteration.

Animation captions:

1. Elements with indices between low and high remain to be searched.
2. Search starts by checking the middle element.
3. If search key is greater than element, then only elements in right sublist need to be searched.
4. Each iteration reduces search space by half. Search continues until key found or search space is empty.

Figure 9.2.1: Binary search algorithm.

```
BinarySearch(numbers, numbersSize, key) {
   mid = 0
   low = 0
   high = numbersSize - 1

   while (high >= low) {
      mid = (high + low) / 2
      if (numbers[mid] < key) {
         low = mid + 1
      }
      else if (numbers[mid] > key) {
         high = mid - 1
      }
      else {
         return mid
      }
   }

   return -1 // not found
}

main() {
   numbers = { 2, 4, 7, 10, 11, 32, 45, 87 }
   NUMBERS_SIZE = 8
   i = 0
   key = 0
   keyIndex = 0

   print("NUMBERS: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   print("Enter a value: ")
   key = getIntFromUser()

   keyIndex = BinarySearch(numbers, NUMBERS_SIZE, key)

   if (keyIndex == -1) {
      printLine(key + " was not found.")
   }
   else {
      printLine("Found " + key + " at index " + keyIndex + ".")
   }
}
```



```
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 10
Found 10 at index 3.
...
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 17
17 was not found.
```

PARTICIPATION ACTIVITY | 9.2.4: Binary search algorithm execution.

Given list: (4, 11, 17, 18, 25, 45, 63, 77, 89, 114).

1) How many list elements will be checked to find the value 77 using binary search?

Check Show answer

2) How many list elements will be checked to find the value 17 using binary search?

Check Show answer

3) Given an array with 32 elements, how many list elements will be checked if the key is less than all elements in the list, using binary search?

Check Show answer

Binary search efficiency

Binary search is incredibly efficient in finding an element within a sorted list. During each iteration or step of the algorithm, binary search reduces the search space (i.e., the remaining elements to search within) by half. The search terminates when the element is found or the search space is empty (element not found). For a 32 element list, if the search key is not found, the search space is halved to have 16 elements, then 8, 4, 2, 1, and finally none, requiring only 6 steps. For an N element list, the maximum number of steps required to reduce the search space to an empty sublist is $\lfloor log_2 \ N \rfloor + 1$. Ex: $\lfloor log_2 \ 32 \rfloor + 1 = 6$.

PARTICIPATION ACTIVITY | 9.2.5: Speed of linear search versus binary search to find a number within a sorted list.

Animation captions:

1. A binary search begins with the middle element of the list. Each subsequent search reduces the search space by half. Using binary search, a match was found with only 3 comparisons.
2. Using linear search, a match was found after 6 comparisons. Compared to a linear search, binary search is incredibly efficient in finding an element within a sorted list.

If each comparison takes 1 µs (1 microsecond), a binary search algorithm's runtime is at most 20 µs to search a list with 1,000,000 elements, 21 µs to search 2,000,000 elements, 22 µs to search 4,000,000 elements, and so on. Ex: Searching Amazon's online store, which has more than 200 million items, requires less than 28 µs; up to 7,000,000 times faster than linear search.

PARTICIPATION ACTIVITY | 9.2.6: Linear and binary search efficiency.

1) Suppose a list of 1024 elements is searched with linear search. How many distinct list elements are compared against a search key that is less than all elements in the list?

[ ] elements

Check Show answer

2) Suppose a sorted list of 1024 elements is searched with binary search. How many distinct list elements are compared against a search key that is less than all

elements in the list?

elements

Check Show answer

CHALLENGE ACTIVITY 9.2.1: Binary search.

Start

A last names list is searched for West using binary search.
Last names list: (Boyd, Cruz, Hall, Hill, Lee, Nava, Pena, Vega, Webb, West)

What is the first last name searched?

Ex: West

What is the second last name searched?

1 2 3 4 5

Check Next

9.3 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 nodes 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 | 9.3.1: Order 3 B-trees.

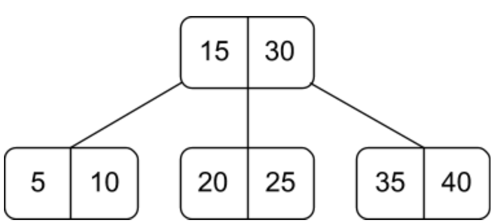
Animation captions:

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 | 9.3.2: Validity of order 3 B-trees.

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

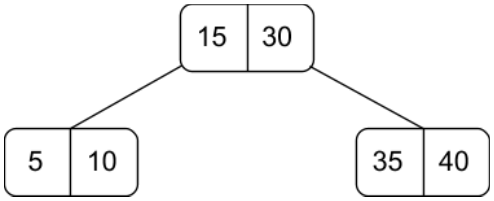
1)



○ Valid

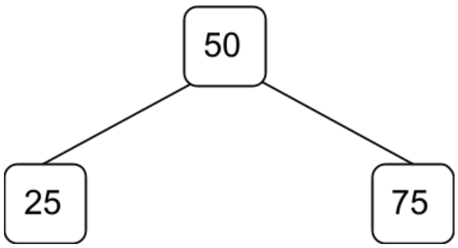
○ Invalid

2)



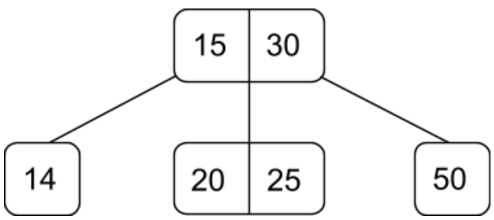
- ◯ Valid
- ◯ Invalid

3)



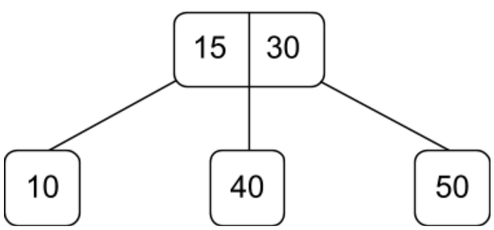
- ◯ Valid
- ◯ Invalid

4)



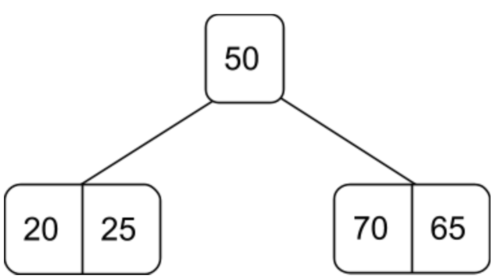
- ◯ Valid
- ◯ Invalid

5)



- ◯ Valid
- ◯ Invalid

6)

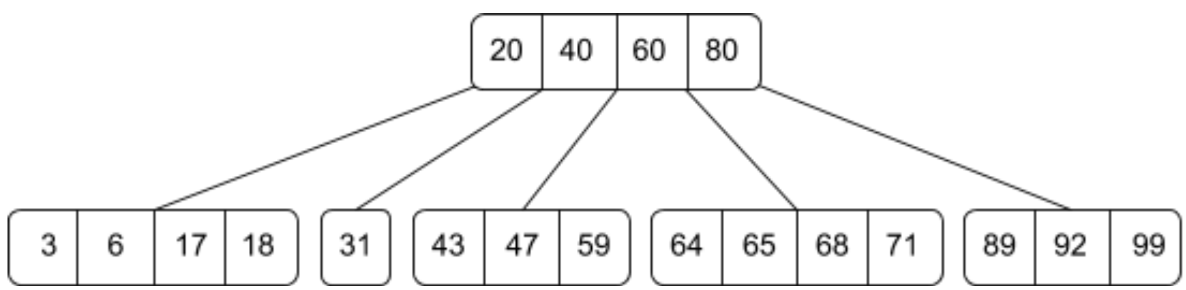


○ Valid

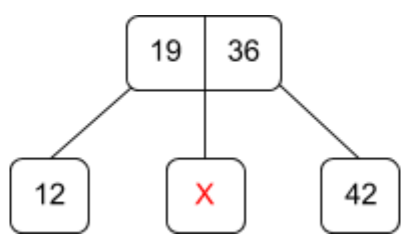
Higher order B-trees

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 children 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 9.3.1: A valid order 5 B-tree.



PARTICIPATION ACTIVITY 9.3.3: 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 9.3.1: 2-3-4 tree internal nodes.

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

PARTICIPATION ACTIVITY 9.3.4: 2-3-4 tree properties.

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

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?

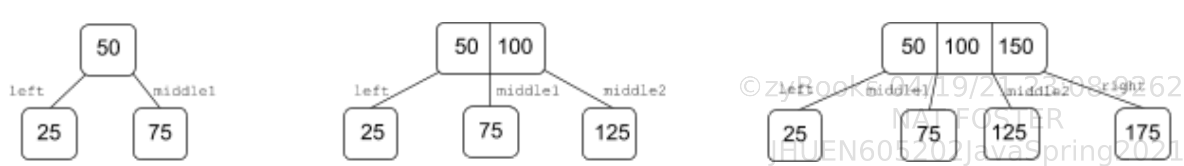
Check Show answer

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 9.3.1: 2-3-4 child subtree labels.



PARTICIPATION ACTIVITY 9.3.5: 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

9.4 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 9.4.1: 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 | 9.4.1: 2-3-4 tree search algorithm.

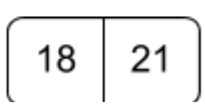
Animation content:

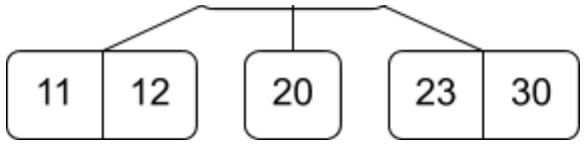
`undefined`

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 | 9.4.2: 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

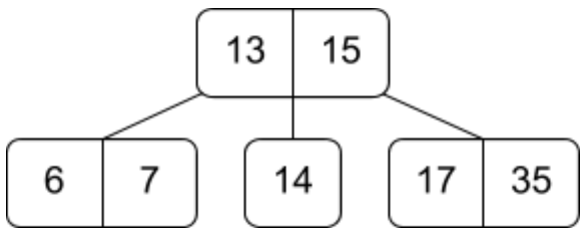
5) When searching for key 20, what is returned by the search function?

- ○ 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 | 9.4.3: 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

9.5 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 | 9.5.1: 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 9.5.2: 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

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 | 9.5.3: 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 9.5.1: Inserting a key with children within a 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 9.5.4: 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 9.5.1: 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 9.5.5: 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 9.5.6: 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 9.5.7: 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.

9.6 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 9.6.1: 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 9.6.2: 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
- 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 9.6.1: 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 | 9.6.3: Utility functions for rotations.

BTreeSetParentKeyLeftOfChild **BTreeGetRightSibling**

BTreeAddKeyAndChild **BTreeGetLeftSibling**

BTreeGetParentKeyLeftOfChild **BTreeRemoveKey**

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 | 9.6.4: 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 | 9.6.5: 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 | 9.6.6: 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 | 9.6.7: 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 | 9.6.8: 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 | 9.6.9: 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

9.7 2-3-4 tree removal

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 | 9.7.1: 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 child, the merge operation is done with a fusion.

PARTICIPATION ACTIVITY | 9.7.2: Merge algorithm.

1, 2, or 3 keys   2 or 3 keys   Exactly 1 key   Exactly 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 9.7.1: BTreeGetMinKey pseudocode.

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

Figure 9.7.2: 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 9.7.3: 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 9.7.4: 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 9.7.3: 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 9.7.4: 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 | 9.7.5: BTreeRemove algorithm: non-leaf case.

Animation content:

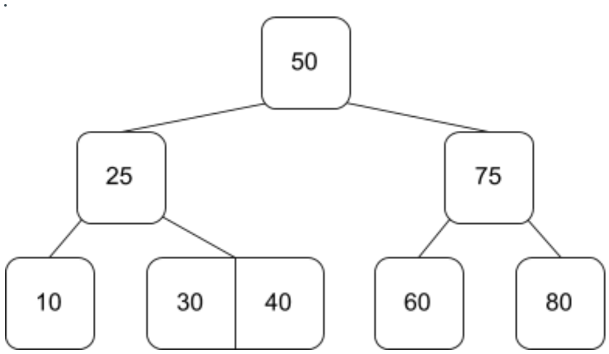
`undefined`

Animation captions:

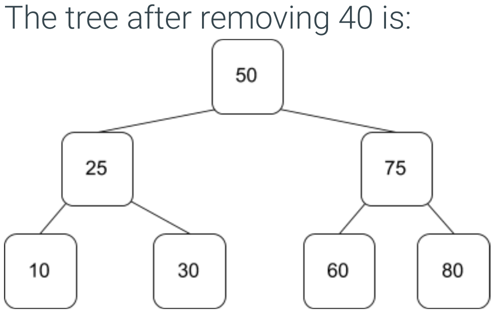
1. When deleting 60, the process is more complex due to the key being found in an internal node.
2. The key 62 is a suitable replacement for 60, but 62 must be recursively removed before the swap.
3. After the recursive removal completes, 60 is replaced with 62.

PARTICIPATION ACTIVITY | 9.7.6: 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

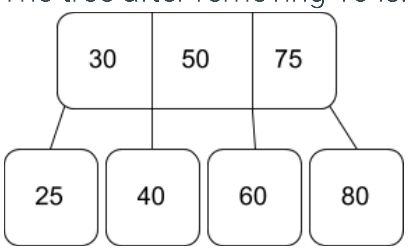
False

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 | 9.7.7: 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.

9.8 AVL: A balanced tree

Balanced BST

An ***AVL tree*** is a BST with a height balance property and specific operations to rebalance the tree when a node is inserted or removed. This section discusses the balance property; another section discusses the operations. A BST is ***height balanced*** if for any node, the heights of the node's left and right subtrees differ by only 0 or 1.

A node's ***balance factor*** is the left subtree height minus the right subtree height, which is 1, 0, or -1 in an AVL tree.

Recall that a tree (or subtree) with just one node has height 0. For calculating a balance factor, a non-existent left or right child's subtree's height is said to be -1.

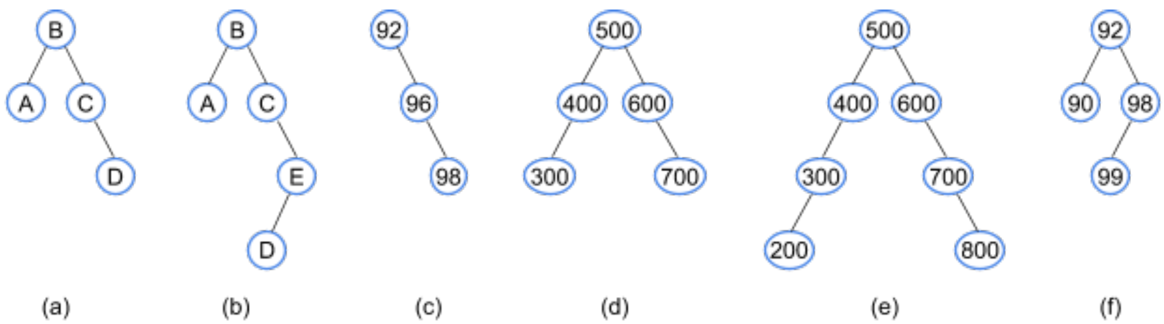
PARTICIPATION ACTIVITY | 9.8.1: An AVL tree is height balanced: For any node, left and right subtree heights differ by only 0 or 1.

Animation captions:

1. Every AVL tree node's balance factor (left minus right heights) is -1, 0, or 1.
2. If any node's subtree heights differ by 2 or more, the entire tree is not an AVL tree.

PARTICIPATION ACTIVITY 9.8.2: AVL trees.

Indicate whether each tree is an AVL tree.



1) (a)

- Yes
- No

2) (b)

- Yes
- No

3) (c)

- Yes
- No

4) (d)

- Yes
- No

5) (e)

- Yes
- No

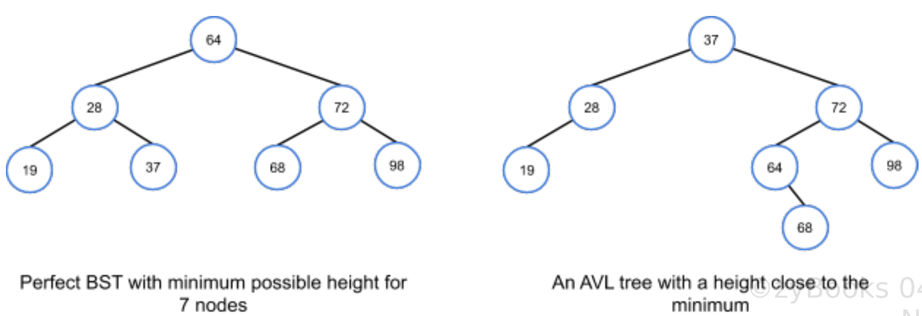
6) (f)

- Yes
- No

AVL tree height

Minimizing binary tree height yields fastest searches, insertions, and removals. If nodes are inserted and removed dynamically, maintaining a minimum height tree requires extensive tree rearrangements. In contrast, an AVL tree only requires a few local rotations (discussed in a later section), so is more computationally efficient, but doesn't guarantee a minimum height. However, theoreticians have shown that an AVL tree's worst case height is no worse than about 1.5x the minimum binary tree height, so the height is still O(log N) where N is the number of nodes. Furthermore, experiments show that AVL tree heights in practice are much closer to the minimum.

Figure 9.8.1: An AVL tree doesn't have to be a perfect BST.



PARTICIPATION ACTIVITY 9.8.3: AVL tree height.

1) An AVL tree maintains the

minimum possible height.

- True
- False

2) What is the minimum possible height of an AVL tree with 7 nodes?

- 2
- 3
- 5
- 7

3) What is the maximum possible height of an AVL tree with 7 nodes?

- 2
- 3
- 5
- 7

Storing height at each AVL node

An AVL tree implementation can store the subtree height as a member of each node. With the height stored as a member of each node, the balance factor for any node can be computed in O(1) time. When a node is inserted in or removed from an AVL tree, ancestor nodes may need the height value to be recomputed.

PARTICIPATION ACTIVITY | 9.8.4: Storing height at each AVL node.

©zyBooks 04/19/21 23:08 926259
NAT FOSTER
JHUEN605202JavaSpring2021

Animation captions:

1. Adding node 55 requires height values for nodes 76 and 47 to be updated.
2. With updated height values at each node, balance factors can be computed. The height of any null subtree is -1.

PARTICIPATION ACTIVITY 9.8.5: Storing height at each AVL node.

1) What relationship does a node's height have to the node's balance factor?

- Height equals balance factor.
- A negative height implies a negative balance factor and a positive height implies a positive balance factor.
- Absolute value of balance factor equals height.
- No relationship.

2) When adding a new node, what is true about the order in which the ancestor height values must be updated?

- Height values must be updated starting at the node's parent, and moving up to the root.
- Height values must be updated starting at the root and moving down to the node.
- Height values can be updated top-down or bottom-up.
- Height values can be updated in any order.

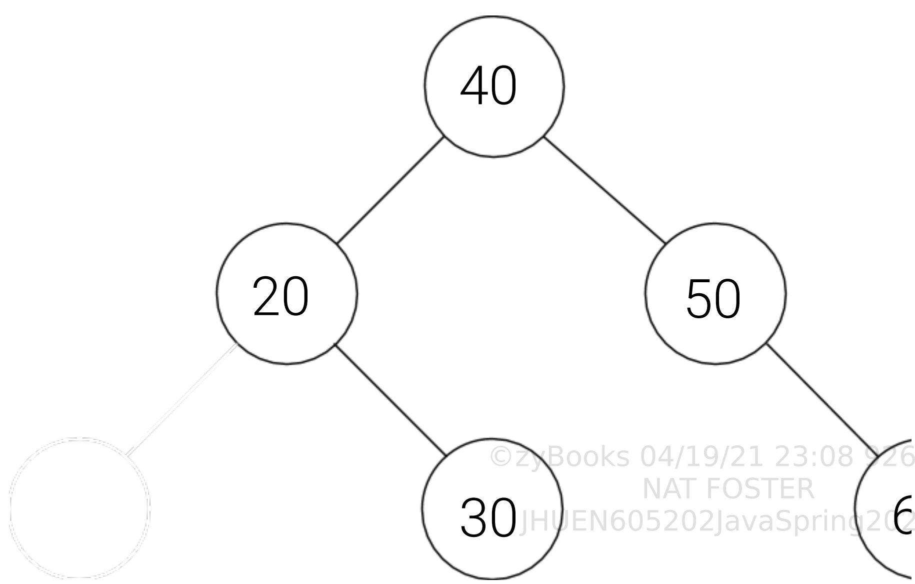
3) When would inserting a new node

to an AVL tree result in no height value changes for all ancestors?

- None. Inserting a new node always changes the height in at least 1 ancestor node.
- A new node is inserted as a child of a leaf node.
- A new node is inserted as a child of an internal node with 1 child.
- The new node is inserted as a child of an internal node with 2 children.

CHALLENGE ACTIVITY 9.8.1: AVL tree properties.

Start



What is the height of 50? Ex: 5

50's left subtree height:

1	2	3	4	5

Check Next

Exploring further:

- AVL is named for inventors Adelson-Velsky and Landis, who described the data structure in a 1962 paper: "An Algorithm for the Organization of Information", often cited as the first balanced tree structure. AVL tree: Wikipedia

9.9 AVL rotations

Tree rotation to keep balance

Inserting an item into an AVL tree may require rearranging the tree to maintain height balance. A ***rotation*** is a local rearrangement of a BST that maintains the BST ordering property while rebalancing the tree.

PARTICIPATION ACTIVITY | 9.9.1: A simple right rotation in an AVL tree.

Animation content:

undefined

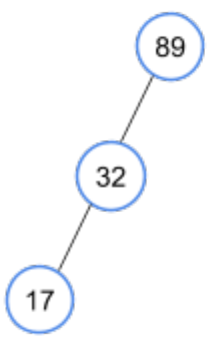
Animation captions:

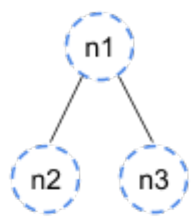
1. This BST violates the AVL height balance property.
2. A right rotation balances the tree while maintaining the BST ordering property.

Rotating is said to be done "at" a node. Ex: Above, the rotation is at node 86.

PARTICIPATION ACTIVITY | 9.9.2: AVL rotate right: 3 nodes.

Rotate right at node 89. Match the node value to the corresponding location in the rotated AVL tree template on the right.





17 **89** **32**

n1

n2

n3

Reset

In the above animation, node 75 becomes the root, and node 86 becomes node 75's new right child. If node 75 had a right child node, that node becomes node 86's left child, to maintain the BST ordering property.

Below, the leaf nodes may instead be subtrees, and the rotation moves the entire subtree along with the node. Likewise, the shown tree may be a subtree, meaning the shown root may have a parent.

PARTICIPATION ACTIVITY | 9.9.3: In a right rotate, B's former right child C becomes D's left child, to maintain the BST ordering property.

Animation captions:

1. If right-rotating B to the root, B's former right child becomes D's left child to maintain the BST ordering property.

PARTICIPATION ACTIVITY | 9.9.4: AVL rotate right: 4 nodes.

Rotate right at node 53. Match the node value to the node location in the rotated AVL tree template on the right.

9 **53** **27** **29**

n1

n2

n3

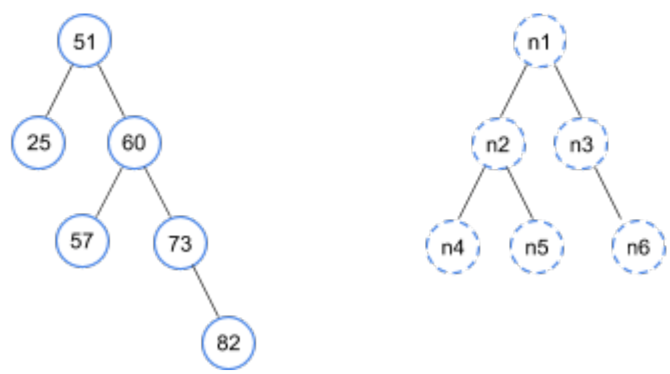
n4

Reset

A left rotation is also possible and is symmetrical to the right rotation.

PARTICIPATION ACTIVITY 9.9.5: AVL rotate left.

Rotate left at node 51. Match the node value to the node location in the AVL tree template.



25 **51** **60** **82** **73** **57**

n1

n2

n3

n4

n5

n6

Reset

Algorithms supporting AVL trees

The `AVLTreeUpdateHeight` algorithm updates a node's height value by taking the maximum of the child subtree heights and adding 1.

The `AVLTreeSetChild` algorithm sets a node as the parent's left or right child, updates the child's parent pointer, and updates the parent node's height.

The `AVLTreeReplaceChild` algorithm replaces one of a node's existing child pointers with a new value, utilizing `AVLTreeSetChild` to perform the replacement.

The `AVLTreeGetBalance` algorithm computes a node's balance factor by subtracting the right subtree height from the left subtree height.

Figure 9.9.1: AVLTreeUpdateHeight, AVLTreeSetChild, AVLTreeReplaceChild, and AVLTreeGetBalance algorithms.


```
AVLTreeUpdateHeight(node) {
   leftHeight = -1
   if (node⇢left != null)
      leftHeight = node⇢left⇢height
   rightHeight = -1
   if (node⇢right != null)
      rightHeight = node⇢right⇢height
   node⇢height = max(leftHeight, rightHeight) + 1
}


AVLTreeSetChild(parent, whichChild, child) {
   if (whichChild != "left" && whichChild != "right")
      return false

   if (whichChild == "left")
      parent⇢left = child
   else
      parent⇢right = child
   if (child != null)
      child⇢parent = parent

   AVLTreeUpdateHeight(parent)
   return true
}


AVLTreeReplaceChild(parent, currentChild, newChild) {
   if (parent⇢left == currentChild)
      return AVLTreeSetChild(parent, "left", newChild)
   else if (parent⇢right == currentChild)
      return AVLTreeSetChild(parent, "right", newChild)
   return false
}


AVLTreeGetBalance(node) {
   leftHeight = -1
   if (node⇢left != null)
      leftHeight = node⇢left⇢height
   rightHeight = -1
   if (node⇢right != null)
      rightHeight = node⇢right⇢height
   return leftHeight - rightHeight
}
```

PARTICIPATION ACTIVITY | 9.9.6: AVL tree utility algorithms.

1) AVLTreeGetBalance has a precondition that the node parameter is non-null.

- True
- False

2) AVLTreeGetBalance has a precondition that the node's children are both non-null.

- True
- False

3) AVLTreeUpdateHeight has a precondition that the node's children both have correct height values.

- True
- False

4) AVLTreeSetChild has a precondition that the child's height value is correct.

- True
- False

5) AVLTreeSetChild calls AVLTreeReplaceChild.

- True
- False

Right rotation algorithm

A right rotation algorithm is defined on a subtree root (node D) which must have a left child (node B). The algorithm reassigns child pointers,assigning B's right child with D, and assigning D's left child with C (B's original right child, which may be null). If D's parent is non-null, then the parent's child D is replaced with B. Other tree parts (T1..T4 below) naturally stay with their parent nodes.

PARTICIPATION ACTIVITY | 9.9.7: Right rotation algorithm.

Animation content:

`undefined`

Animation captions:

1. A right rotation at node D changes P's child from D to B, D's left child from B to C, and B's right child from C to D.

PARTICIPATION ACTIVITY | 9.9.8: Right rotation algorithm.

Refer to the above AVL tree right rotation algorithm.

1) The algorithm works even if node B's right child is null.

- True
- False

2) The algorithm works even if node D's left child is null.

- True
- False

3) Node D may be a subtree of a larger tree. The algorithm updates node D's parent to point to node B, the new root of the subtree.

- True
- False

AVL tree balancing

When an AVL tree node has a balance factor of 2 or -2, which only occurs after an insertion or removal, the node must be rebalanced via rotations. The `AVLTreeRebalance` algorithm updates the height value at a node, computes the balance factor, and rotates if the balance factor is 2 or -2.

Figure 9.9.2: AVLTreeRebalance algorithm.

```
AVLTreeRebalance(tree, node) {
   AVLTreeUpdateHeight(node)
   if (AVLTreeGetBalance(node) == -2) {
      if (AVLTreeGetBalance(node⇢right) == 1) {
         // Double rotation case.
         AVLTreeRotateRight(tree, node⇢right)
      }
      return AVLTreeRotateLeft(tree, node)
   }
   else if (AVLTreeGetBalance(node) == 2) {
      if (AVLTreeGetBalance(node⇢left) == -1) {
         // Double rotation case.
         AVLTreeRotateLeft(tree, node⇢left)
      }
      return AVLTreeRotateRight(tree, node)
   }
   return node
}
```


PARTICIPATION ACTIVITY 9.9.9: AVLTreeRebalance algorithm.

1) AVLTreeRebalance rebalances all ancestors from the node up to the root.

- True
- False

2) AVLTreeRebalance recomputes the height values for each non-null child.

- True
- False

3) AVLTreeRebalance recomputes the height for the node.

- True
- False

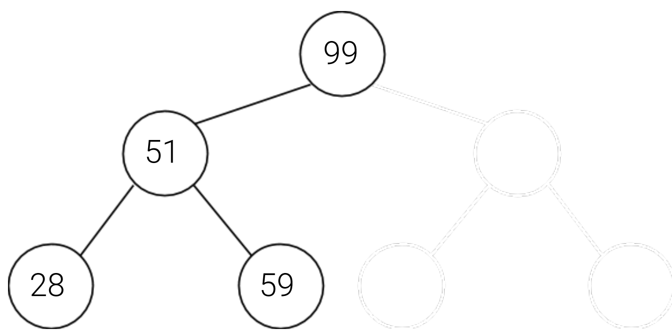
4) AVLTreeRebalance takes no action if a node's balance factor is 1, 0, or -1.

- True
- False

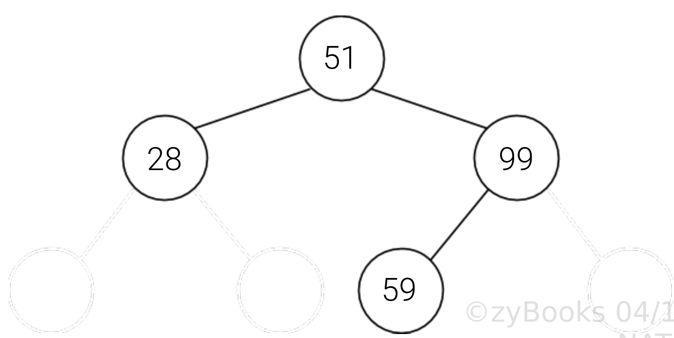
CHALLENGE ACTIVITY 9.9.1: AVL rotations.

Start

Given the following BST violating the AVL height balance property:



A left tion at node 28 ds:



1 2 3

Check Next

9.10 AVL insertions

Insertions requiring rotations to rebalance

Inserting an item into an AVL tree may cause the tree to become unbalanced. A rotation can rebalance the tree.

PARTICIPATION ACTIVITY | 9.10.1: After an insert, a rotation may rebalance the tree.

Animation captions:

1. Inserting a node may temporarily violate the AVL height balance property.
2. A rotation, at the problem node closest to the new node, restores balance.

Sometimes, the imbalance is due to an insertion on the *inside* of a subtree, rather than on the *outside* as above. One rotation won't rebalance. A double rotation is needed.

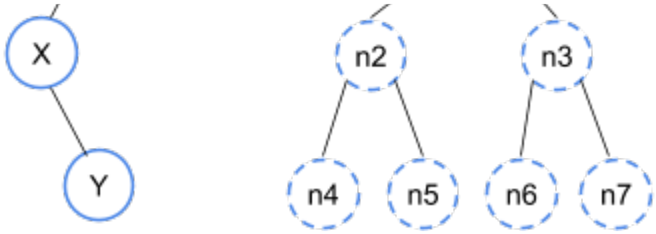
PARTICIPATION ACTIVITY | 9.10.2: Sometimes a double rotation is necessary to rebalance.

Animation captions:

1. Inserting a node may temporarily violate the AVL height balance property.
2. In this case, after a single rotate, the tree is still unbalanced.
3. A double "left-then-right" rotate is necessary. First rotate left at A...
4. ...and then rotate right at C. Tree is now balanced.

PARTICIPATION ACTIVITY | 9.10.3: Double rotate: Left-then-right.

When performing a double rotation (left-then-right), indicate each node's new location using the template tree's labels (n1, n2, n3, n4, n5, n6, or n7).



1) After the initial left rotation, where is Y?

Check Show answer

2) After the initial left rotation, where is X?

Check Show answer

3) After the left rotation, where is the right rotation performed: at X, Y, or Z?

Check Show answer

4) After the left rotation and then the right rotation, where is Z?

Check Show answer

5) After the left rotation and then the right rotation, where is Y?

Check Show answer

6) After the left rotation and then the right rotation, where is X?

Check Show answer

7) If the left rotation had NOT first been performed, a right rotation at Z would have made X the new root, and made Z X's right child. To where would Y have been moved?

Check Show answer

Four imbalance cases

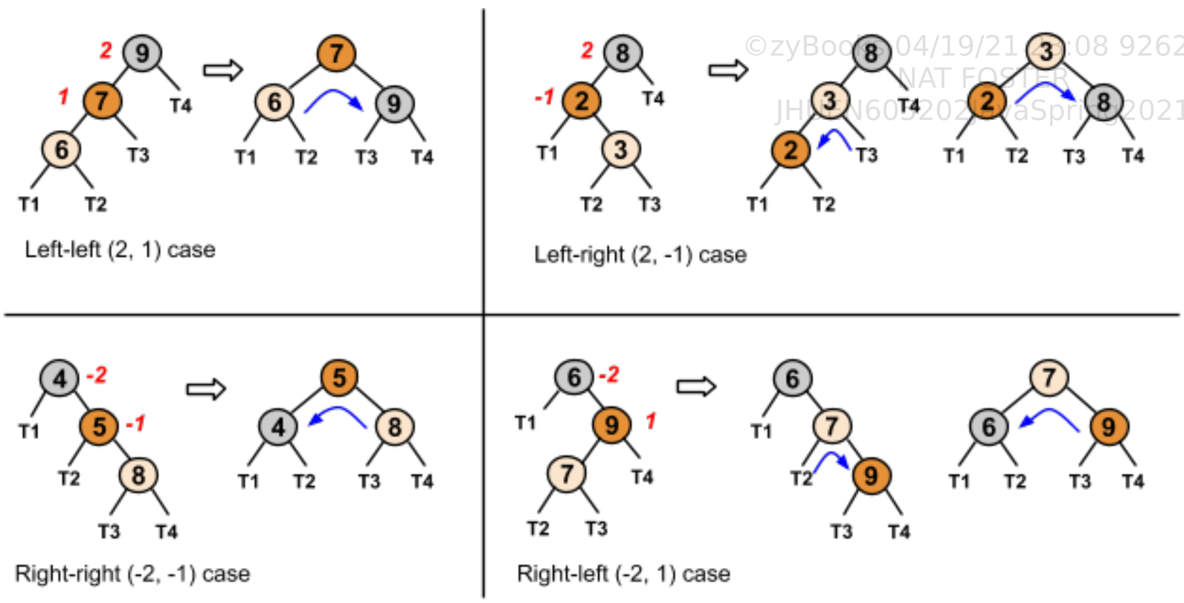
After inserting a node, nodes on the path from the new node to the root should be checked for a balance factor of 2 or -2. The first such node P triggers rebalancing. Four cases exist, distinguishable by the balance factor of node P and one of P's children.

PARTICIPATION ACTIVITY | 9.10.4: Four AVL imbalance cases are possible after inserting a new node.

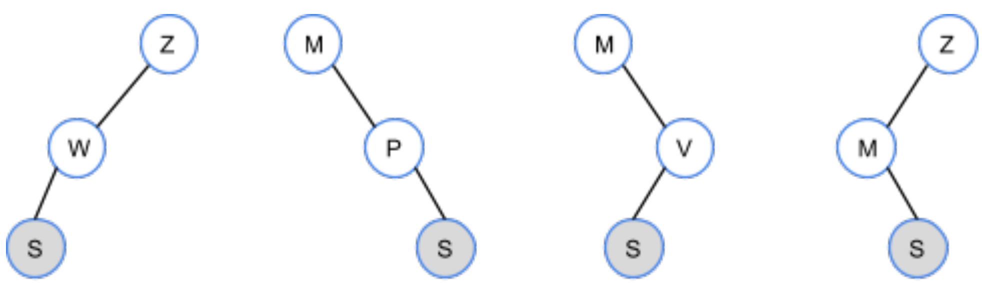
Animation captions:

1. Four imbalance cases can arise from inserting a new node into an AVL tree. Inserting node 12 leads to a left-left imbalance case.
2. Inserting node 38 to the original AVL tree results in a left-right imbalance case.
3. Similarly, right-right and right-left imbalance cases also exist.

Figure 9.10.1: Four imbalance cases and rotations (indicated by blue arrow) to rebalance.



PARTICIPATION ACTIVITY 9.10.5: Determine the imbalance case and appropriate rotations.



(a) (b) (c) (d)

1) (a)

- ○ (2, 1)
- ○ (2, -1)
- ○ (-2, -1)
- ○ (-2, 1)

2) (b)

- (2, 1)
- (2, -1)
- (-2, -1)
- (-2, 1)

3) (c)

- (2, 1)
- (2, -1)
- (-2, -1)
- (-2, 1)

4) (d)

- (2, 1)
- (2, -1)
- (-2, -1)
- (-2, 1)

5) What is the proper rotation for (a)?

- Left at Z
- Right at Z
- Right at W, then right at Z.

6) What is the proper rotation for a 2, -1 case?

- Right
- Right-left
- Left-right

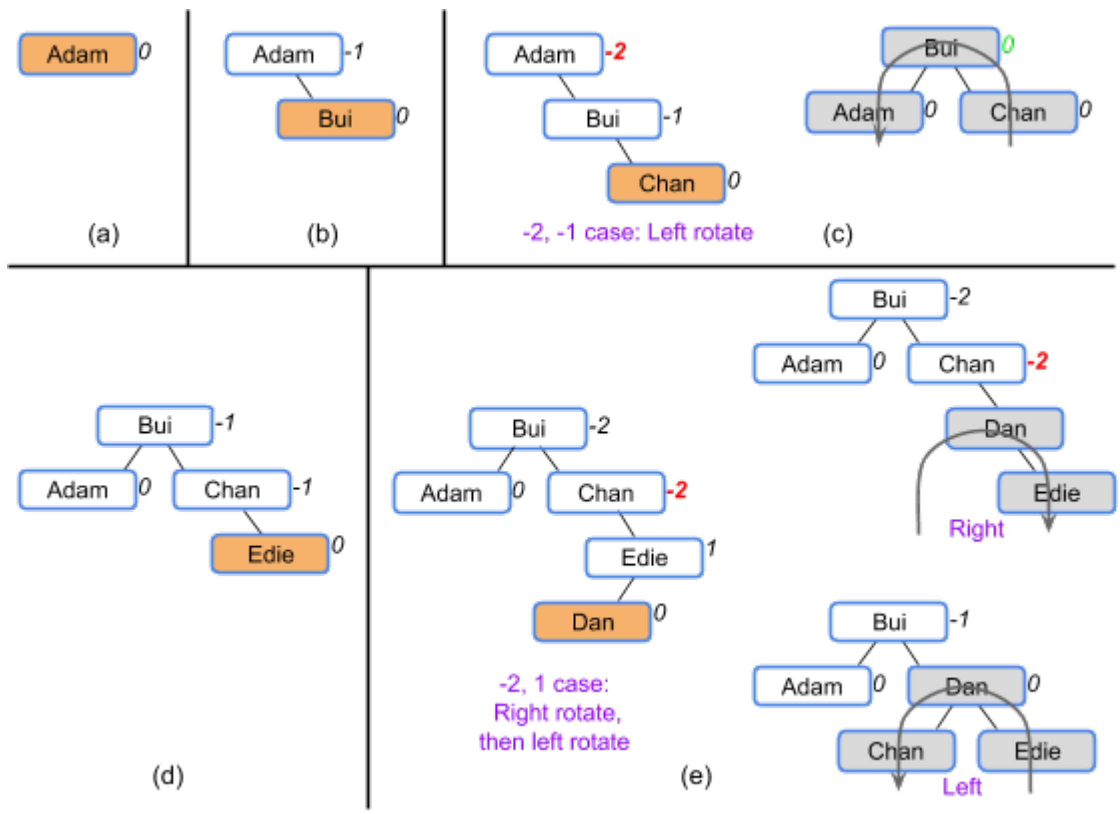
Insertion with rebalancing

An AVL tree insertion involves searching for the insert location, inserting the new node, updating balance factors, and rebalancing.

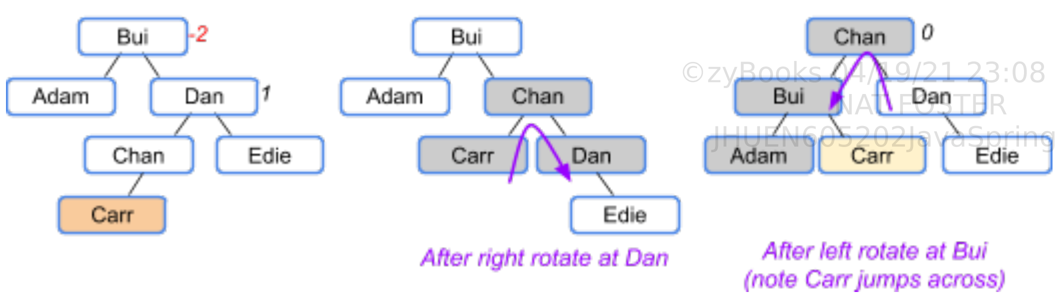
Balance factor updates are only needed on nodes ascending along the path from the inserted node up to the root, since no other nodes' balance could be affected. Each node's balance factor can be recomputed by determining left and right subtree heights, or for speed can be stored in each node and then incrementally updated: +1 if ascending from a left child, -1 if from a right child. If a balance factor update yields 2 or -2, the imbalance case is determined via that node's left (for 2) or right (for -2) child's balance factor, and the appropriate rotations performed.

Example 9.10.1: AVL example: Phone book.

A phone book program stores names in an AVL tree. A program user enters a series of names, which just happen to be in sorted order (perhaps copying from a previously sorted list). The tree is kept balanced by occasional rebalancing rotations, thus enabling fast searches. Without rebalancing, the BST's final height would be 4 instead of just 2.



Adding another item shows the interesting case of rotations occurring higher up in the tree.



PARTICIPATION ACTIVITY | 9.10.6: AVL balance.

1) If an AVL tree has X levels, the first X-1 levels will be full.

- True
- False

2) For n nodes, an AVL tree has height equal to floor(log(n)).

- True
- False

3) For n nodes, an AVL tree has height O(log(n)).

- True
- False

4) An AVL insert operation involves a search, an insert, and possibly some rotations. An insert operation is thus O(log(n)).

- True
- False

5) After inserting a node into a tree, all tree nodes must have their balance factors updated.

- True
- False

6) Conceivably, inserting 100 items into an AVL tree may not require *any* rotations.

- True
- False

AVL insertion algorithm

Insertion starts with the standard BST insertion algorithm. After inserting a node, all ancestors of the inserted node, from the parent up to the root, are rebalanced. A node is rebalanced by first computing the node's balance factor, then performing rotations if the balance factor is outside of the range [-1,1].

Figure 9.10.2: AVLTreeInsert algorithm.

```
AVLTreeInsert(tree, node) {
   if (tree⇢root == null) {
      tree⇢root = node
      node⇢parent = null
      return
   }

   cur = tree⇢root
   while (cur != null) {
      if (node⇢key < cur⇢key) {
         if (cur⇢left == null) {
            cur⇢left = node
            node⇢parent = cur
            cur = null
         }
         else {
            cur = cur⇢left
         }
      }
      else {
         if (cur⇢right == null) {
            cur⇢right = node
            node⇢parent = cur
            cur = null
         }
         else
            cur = cur⇢right
      }
   }

   node = node⇢parent
   while (node != null) {
      AVLTreeRebalance(tree, node)
      node = node⇢parent
   }
}
```


PARTICIPATION ACTIVITY 9.10.7: AVLTreeInsert algorithm.

1) AVLTreeInsert updates heights on all ancestors before inserting the node.

- True
- False

2) The node passed to AVLTreeInsert must be a leaf node.

- True
- False

3) AVLTreeInsert works to insert a node into an empty tree.

- True
- False

4) AVLTreeInsert adds the new node as a child to an existing node in the tree, but the new node's parent pointer is not set and must be handled outside of the function.

- True
- False

5) AVLTreeInsert sets the height in the newly inserted node to 0 and the node's left and right child pointers to null.

- True
- False

AVL insertion algorithm complexity

The AVL insertion algorithm traverses the tree from the root to a leaf node to find the insertion point, then traverses back up to the root to rebalance. One node is visited per level, and at most 2 rotations are needed for a single node. Each rotation is an O(1) operation. Therefore, the runtime complexity of insertion is O(log N).

Because a fixed number of temporary pointers are needed for the AVL insertion algorithm, including any rotations, the space complexity is O(1).

Exploring further:

- AVL tree simulator

9.11 AVL removals

Removing nodes in AVL trees

Given a key, an AVL tree ***remove*** operation removes the first-found matching node, restructuring the tree to preserve all AVL tree requirements. Removal begins by removing the node using the standard BST removal algorithm. After removing a node, all ancestors of the removed node, from the nodes' parent up to the root, are rebalanced. A node is rebalanced by first computing the node's balance factor, then performing rotations if the balance factor is 2 or -2.

PARTICIPATION ACTIVITY 9.11.1: AVL tree removal.

Animation captions:

1. Removing node 63 begins with the standard BST removal. A pointer to node 63's parent is kept.
2. After removal, the balance factor of each node from the parent up to the root is checked. Rotations are used if the balance factor (BF) is 2 or -2.
3. Removing node 84 replaces the node with node 84's successor, node 89. Node 89 is then removed from the right subtree. No rotations are necessary because the root's balance factor changes to 1.
4. After the standard BST removal algorithm removes node 93, the root is left with a balance factor of 2. A right rotation at 21 rebalances the tree.

PARTICIPATION ACTIVITY | 9.11.2: AVL tree removal.

1) The BST removal algorithm is used as part of AVL tree removal.

- True
- False

2) After removing a node from an AVL tree using the standard BST removal algorithm, all nodes in the tree must be rebalanced.

- True
- False

3) During rebalancing, encountering nodes with balance factors of 2 or -2 implies that a rotation must occur.

- True
- False

4) Removal of an internal node with 2 children always requires a rotation to rebalance.

- True
- False

AVL tree removal algorithm

To remove a key, the AVL tree removal algorithm first locates the node containing the key using `BSTSearch`. If the node is found, AVLTreeRemoveNode is called to remove the node. Standard BST removal logic is used to remove the node from the tree. Then AVLTreeRebalance is called for all ancestors of the removed node, from the parent up to the root.

PARTICIPATION ACTIVITY

9.11.3: AVL tree removal algorithm.

Animation content:

undefined

Animation captions:

1. Removal of 75 starts with the standard BST removal, replacing the child and updating the height at node 50.
2. The node has not changed, so the balancing begins at the parent and continues up parent pointers until null.
3. Removal of 88 again starts with the standard BST removal. The root's balance factor changes to 2, requiring a rotation to rebalance.
4. After removals, the tree maintains O(log n) height.

Figure 9.11.1: AVLTreeRebalance algorithm.

```
AVLTreeRebalance(tree, node) {
   AVLTreeUpdateHeight(node)
   if (AVLTreeGetBalance(node) == -2) {
      if (AVLTreeGetBalance(node⇢right) == 1) {
         // Double rotation case.
         AVLTreeRotateRight(tree, node⇢right)
      }
      return AVLTreeRotateLeft(tree, node)
   }
   else if (AVLTreeGetBalance(node) == 2) {
      if (AVLTreeGetBalance(node⇢left) == -1) {
         // Double rotation case.
         AVLTreeRotateLeft(tree, node⇢left)
      }
      return AVLTreeRotateRight(tree, node)
   }
   return node
}
```

Figure 9.11.2: AVLTreeRemoveKey algorithm.

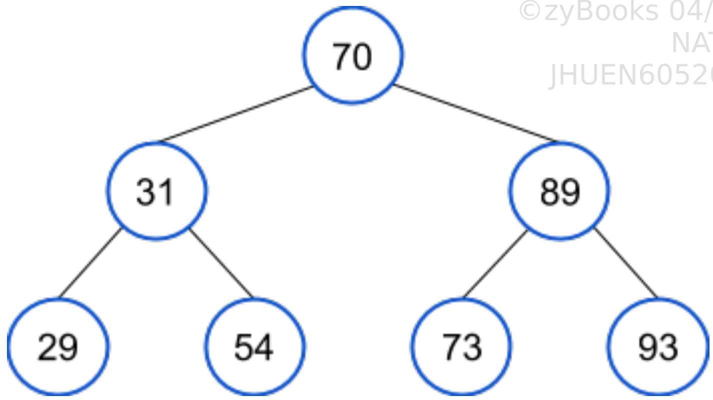
```
AVLTreeRemoveKey(tree, key) {
   node = BSTSearch(tree, key)
   return AVLTreeRemoveNode(tree, node)
}
```

Figure 9.11.3: AVLTreeRemoveNode algorithm.

```
AVLTreeRemoveNode(tree, node) {
   if (node == null)
      return false

   // Parent needed for rebalancing
   parent = node⇢parent

   // Case 1: Internal node with 2 children
   if (node⇢left != null && node⇢right != null) {
      // Find successor
      succNode = node⇢right
      while (succNode⇢left != null)
         succNode = succNode⇢left

      // Copy the value from the node
      node = Copy succNode

      // Recursively remove successor
      AVLTreeRemoveNode(tree, succNode)

      // Nothing left to do since the recursive call will have rebalanced
      return true
   }

   // Case 2: Root node (with 1 or 0 children)
   else if (node == tree⇢root) {
      if (node⇢left != null)
         tree⇢root = node⇢left
      else
         tree⇢root = node⇢right

      if (tree⇢root)
         tree⇢root⇢parent = null

      return true
   }

   // Case 3: Internal with left child only
   else if (node⇢left != null)
      AVLTreeReplaceChild(parent, node, node⇢left)

   // Case 4: Internal with right child only OR leaf
   else
      AVLTreeReplaceChild(parent, node, node⇢right)

   // node is gone. Anything that was below node that has persisted is already
correctly
   // balanced, but ancestors of node may need rebalancing.
   node = parent
   while (node != null) {
      AVLTreeRebalance(tree, node)
      node = node⇢parent
   }
   return true
}
```

PARTICIPATION ACTIVITY | 9.11.4: AVL tree removal algorithm.

Select the order of tree-altering operations that occur as a result of calling AVLTreeRemoveKey to remove 70 from this tree:



2 **5** **3** **Never** **4** **6** **1**

Node 89's left child is set to null.

The tree's root pointer is set to the root's left child.

The node being removed is compared against the tree's root and is not equal.

Node 73 is found as the successor to node 70.

AVLTreeRebalance is called on node 73, which has a balance factor of 0, so no rotations are necessary.

Key 73 is copied into the root node.

AVLTreeRebalance is called on node 89, which has a balance factor of -1, so no rotations are necessary.

Reset

AVL removal algorithm complexity

In the worst case scenario, the AVL removal algorithm traverses the tree from the root to the lowest level to find the node to remove, then traverses back up to the root to rebalance. One node is visited per level, and at most 2 rotations are needed for a single node. Each rotation is an O(1) operation. Therefore, the runtime complexity of an AVL tree removal is O(log N).

Because a fixed number of temporary pointers are needed for the AVL removal algorithm, including any rotations, the space complexity is O(1).

9.12 Red-black tree: A balanced tree

This section has been set as optional by your instructor.

A ***red-black tree*** is a BST with two node types, namely red and black, and supporting operations that ensure the tree is balanced when a node is inserted or removed. The below red-black tree's requirements ensure that a tree with N nodes will have a height of O(log N).

- Every node is colored either red or black.
- The root node is black.
- A red node's children cannot be red.
- A null child is considered to be a black leaf node.

- All paths from a node to any null leaf descendant node must have the same number of black nodes.

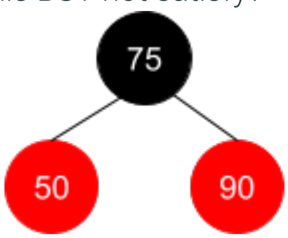
PARTICIPATION ACTIVITY | 9.12.1: Red-black tree rules.

Animation captions:

1. The null child pointer of a leaf node is considered a null leaf node and is always black. Visualizing null leaf nodes helps determine if a tree is a valid red-black tree.
2. Each requirement must be met for the tree to be a valid red-black tree.
3. A tree that violates any requirement is not a valid red-black tree.

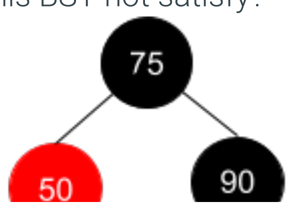
PARTICIPATION ACTIVITY | 9.12.2: Red-black tree rules.

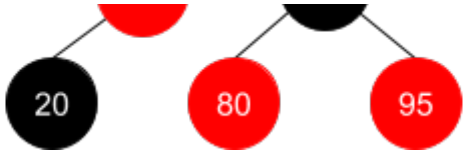
1) Which red-black tree requirement does this BST not satisfy?



- Root node must be black.
- A red node's children cannot be red.
- All paths from a node to null leaf nodes must have the same number of black nodes.
- None.

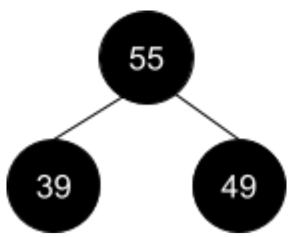
2) Which red-black tree requirement does this BST not satisfy?





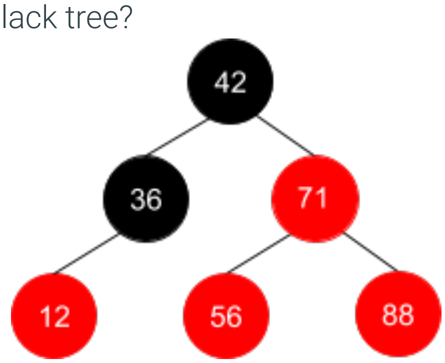
- Root node must be black.
- A red node's children cannot be red.
- Not all levels are full.
- All paths from a node to null leaf nodes must have the same number of black nodes.

3) The tree below is a valid red-black tree.



- True
- False

4) What single color change will make the below tree a valid red-black tree?



- Change node 36's color to red.
- Change node 71's color to black.
- Change node 88's color to black.
- No single color change will make this a valid red-black tree..

5) A black node's children will always be the same color.

- True
- False

6) All valid red-black trees will have more red nodes than black nodes.

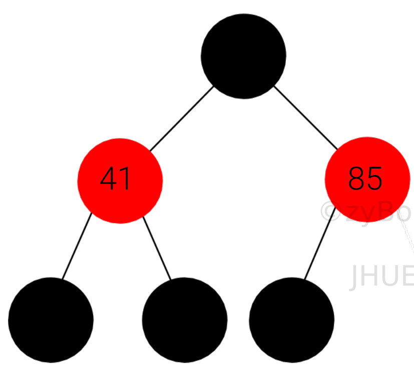
- True
- False

7) Any BST can be made a red-black tree by coloring all nodes black.

- True
- False

CHALLENGE ACTIVITY 9.12.1: Red-black tree: A balanced tree.

Start



Is the above a valid red-black tree? Select all that apply.

- [ ] Yes
- [ ] No, root node must be black

1	2	3

Check Next

9.13 Red-black tree: Rotations

This section has been set as optional by your instructor.

Introduction to rotations

A rotation is a local rearrangement of a BST that maintains the BST ordering property while rebalancing the tree. Rotations are used during the insert and remove operations on a red-black tree to ensure that red-black tree requirements hold. Rotating is said to be done "at" a node. A left rotation at a node causes the node's right child to take the node's place in the tree. A right rotation at a node causes the node's left child to take the node's place in the tree.

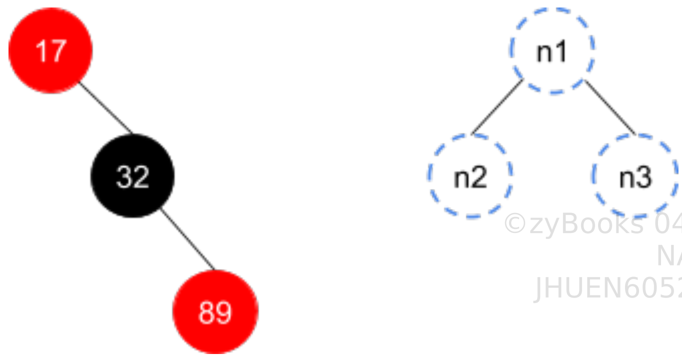
PARTICIPATION ACTIVITY | 9.13.1: A simple left rotation in a red-black tree.

Animation captions:

1. This BST is not a valid red-black tree. From the root, paths down to null leaves are inconsistent in terms of number of black nodes.
2. A left rotation at node 16 creates a valid red-black tree.

PARTICIPATION ACTIVITY | 9.13.2: Red-black tree rotate left: 3 nodes.

Rotate left at node 17. Match the node value to the corresponding location in the rotated red-black tree template on the right.



17 **32** **89**

n2

n1

n3

Reset

Left rotation algorithm

A rotation requires altering up to 3 child subtree pointers. A left rotation at a node requires the node's right child to be non-null. Two utility functions are used for red-black tree rotations. The `RBTreeSetChild` utility function sets a node's left child, if the whichChild parameter is "left", or right child, if the whichChild parameter is "right", and updates the child's parent pointer. The `RBTreeReplaceChild` utility function replaces a node's left or right child pointer with a new value.

Figure 9.13.1: RBTreeSetChild utility function.

```
RBTreeSetChild(parent, whichChild, child) {
   if (whichChild != "left" && whichChild != "right")
      return false

   if (whichChild == "left")
      parent⇢left = child
   else
      parent⇢right = child
   if (child != null)
      child⇢parent = parent
   return true
}
```


Figure 9.13.2: RBTreeReplaceChild utility function.

```
RBTreeReplaceChild(parent, currentChild, newChild) {
   if (parent⇢left == currentChild)
      return RBTreeSetChild(parent, "left", newChild)
   else if (parent⇢right == currentChild)
      return RBTreeSetChild(parent, "right", newChild)
   return false
}
```

The `RBTreeRotateLeft` function performs a left rotation at the specified node by updating the right child's left child to point to the node, and updating the node's right child to point to the right child's former left child. If non-null, the node's parent has the child pointer changed from node to the node's right child. Otherwise, if the node's parent is null, then the tree's root pointer is updated to point to the node's right child.

Figure 9.13.3: RBTreeRotateLeft pseudocode.

```
RBTreeRotateLeft(tree, node) {
   rightLeftChild = node⇢right⇢left
   if (node⇢parent != null)
      RBTreeReplaceChild(node⇢parent, node, node⇢right)
   else { // node is root
      tree⇢root = node⇢right
      tree⇢root⇢parent = null
   }
   RBTreeSetChild(node⇢right, "left", node)
   RBTreeSetChild(node, "right", rightLeftChild)
}
```


PARTICIPATION ACTIVITY 9.13.3: RBTreeRotateLeft algorithm.

Node with null left child **Node with null right child** **Red node** **Root node**

RBTreeRotateLeft will not work when called at this type of node.

RBTreeRotateLeft called at this node requires the tree's root pointer to be updated.

After calling RBTreeRotateLeft at this node, the node will have a null left child.

After calling RBTreeRotateLeft at this node, the node will be colored red.

Reset

Right rotation algorithm

Right rotation is analogous to left rotation. A right rotation at a node requires the node's left child to be non-null.

PARTICIPATION ACTIVITY | 9.13.4: RBTreeRotateRight algorithm.

Animation content:

`undefined`

Animation captions:

1. A right rotation at node 80 causes node 61 to become the new root, and nodes 40 and 80 to become the root's left and right children, respectively.
2. The rotation results in a valid red-black tree.

PARTICIPATION ACTIVITY | 9.13.5: Right rotation algorithm.

1) A rotation will never change the root node's value.

- True
- False

2) A rotation at a node will only change properties of the node's descendants, but will never change properties of the node's ancestors.

- True
- False

3) RBTreeRotateRight works even if

the node's parent is null.

- True
- False

4) RBTreeRotateRight works even if the node's left child is null.

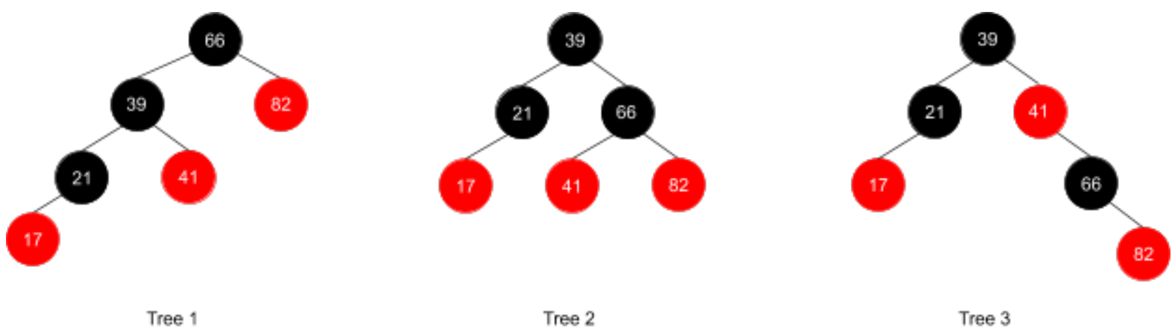
- True
- False

5) RBTreeRotateRight works even if the node's right child is null.

- True
- False

PARTICIPATION ACTIVITY 9.13.6: Red-black tree rotations.

Consider the 3 trees below:



1) Which trees are valid red-black trees?

- Tree 1 only.
- Tree 2 only.
- Tree 3 only.
- All are valid red-black trees.

2) Which operation on tree 1 would produce tree 2?

- Rotate right at node 82.
- Rotate left at node 66.
- Rotate right at node 66.
- Rotate left at node 39.

3) Which operation on tree 3 would produce tree 2?

- Rotate left at node 21.
- Rotate left at node 39.
- Rotate left at node 41.
- Rotate left at node 66.

4) A right rotation at node 21 in tree 2 would result in a valid red-black tree.

- True
- False

9.14 Red-black tree: Insertion

This section has been set as optional by your instructor.

Given a new node, a red-black tree ***insert*** operation inserts the new node in the proper location such that all red-black tree requirements still hold after the insertion completes.

Red-black tree insertion begins by calling `BSTInsert` to insert the node using the BST insertion rules. The newly inserted node is colored red and then a balance operation is performed on this node.

Figure 9.14.1: RBTreeInsert algorithm.

```
RBTreeInsert(tree, node) {
   BSTInsert(tree, node)
   node⇢color = red
   RBTreeBalance(tree, node)
}
```


The red-black balance operation consists of the steps below.

1. Assign `parent` with `node`'s parent, `uncle` with `node`'s uncle, which is a sibling of `parent`, and `grandparent` with `node`'s grandparent.
2. If `node` is the tree's root, then color `node` black and return.
3. If `parent` is black, then return without any alterations.
4. If `parent` and `uncle` are both red, then color `parent` and `uncle` black, color `grandparent` red, recursively balance `grandparent`, then return.
5. If `node` is `parent`'s right child and `parent` is `grandparent`'s left child, then rotate left at `parent`, assign `node` with `parent`, assign `parent` with `node`'s parent, and go to step 7.
6. If `node` is `parent`'s left child and `parent` is `grandparent`'s right child, then rotate right at `parent`, assign `node` with `parent`, assign `parent` with `node`'s parent, and go to step 7.
7. Color `parent` black and `grandparent` red.
8. If `node` is `parent`'s left child, then rotate right at `grandparent`, otherwise rotate left at `grandparent`.

The RBTreeBalance function uses the RBTreeGetGrandparent and RBTreeGetUncle utility functions to determine a node's grandparent and uncle, respectively.

Figure 9.14.2: RBTreeGetGrandparent and RBTreeGetUncle utility functions.

```
RBTreeGetGrandparent(node) {
   if (node⇢parent == null)
      return null
   return node⇢parent⇢parent
}

RBTreeGetUncle(node) {
   grandparent = null
   if (node⇢parent != null)
      grandparent = node⇢parent⇢parent
   if (grandparent == null)
      return null
   if (grandparent⇢left == node⇢parent)
      return grandparent⇢right
   else
      return grandparent⇢left
}
```


PARTICIPATION ACTIVITY | 9.14.1: RBTreeBalance algorithm.

Animation content:

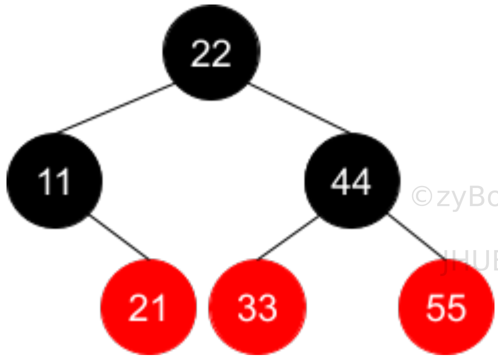
undefined

Animation captions:

1. Insertion of 22 as the root starts with the normal BST insertion, followed by coloring the node red. The balance operation simply changes the root node to black.
2. Insertion of 11 and 33 do not require any node color changes or rotations.
3. Insertion of 55 requires recoloring the parent, uncle, and grandparent, then recursively balancing the grandparent.
4. Inserting 44 requires two rotations. The first rotation is a right rotation at the parent, node 55. The second rotation is a left rotation at the grandparent, node 33.

PARTICIPATION ACTIVITY | 9.14.2: Red-black tree: insertion.

Consider the following tree:



1) Starting at and including the root node, how many black nodes are encountered on any path down to and including the null leaf nodes?

- 1
- 2
- 3
- 4

2) Insertion of which value will require at least 1 rotation?

- 10
- 20
- 30
- 45

3) The values 11, 21, 22, 33, 44, 55 can be inserted in any order and the above tree will always be the result.

- True
- False

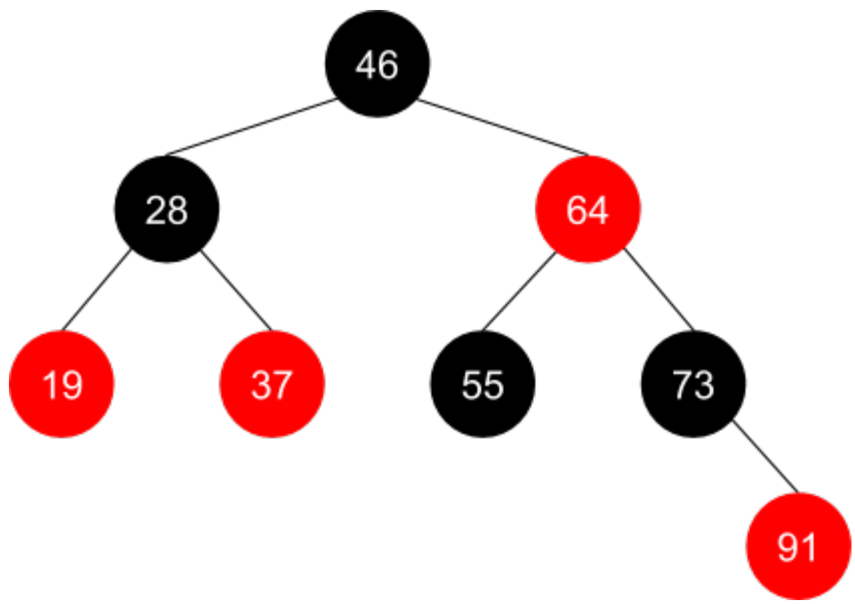
4) All red nodes could be recolored to black and the above tree would still be a valid red-black tree.

True

False

PARTICIPATION ACTIVITY | 9.14.3: RBTreeInsert algorithm.

Select the order of tree-altering operations that occur as a result of calling RBTreeInsert to insert 82 into this tree:



3 **5** **4** **2** **1** **Never**

Rotate left at node 73.

Insert red node 82 as node 91's left child.

Color grandparent node red.

Color parent node black.

Rotate right at node 91.

Call RBTreeBalance

recursively on node 73.

Reset

9.15 Red-black tree: Removal

This section has been set as optional by your instructor.

Removal overview

Given a key, a red-black tree ***remove*** operation removes the first-found matching node, restructuring the tree to preserve all red-black tree requirements. First, the node to remove is found using `BSTSearch`. If the node is found, RBTreeRemoveNode is called to remove the node.

Figure 9.15.1: RBTreeRemove algorithm.

```
RBTreeRemove(tree, key) {
   node = BSTSearch(tree, key)
   if (node != null)
      RBTreeRemoveNode(tree, node)
}
```

The RBTreeRemove algorithm consists of the following steps:

1. If the node has 2 children, copy the node's predecessor to a temporary value, recursively remove the predecessor from the tree, replace the node's key with the temporary value, and return.
2. If the node is black, call `RBTreePrepareForRemoval` to restructure the tree in preparation for the node's removal.
3. Remove the node using the standard BST `BSTRemove` algorithm.

Figure 9.15.2: RBTreeRemoveNode algorithm.

```
RBTreeRemoveNode(tree, node) {
   if (node⇢left != null && node⇢right != null) {
      predecessorNode = RBTreeGetPredecessor(node)
      predecessorKey = predecessorNode⇢key
      RBTreeRemoveNode(tree, predecessorNode)
      node⇢key = predecessorKey
      return
   }

   if (node⇢color == black)
      RBTreePrepareForRemoval(node)
   BSTRemove(tree, node⇢key)
}
```


Figure 9.15.3: RBTreeGetPredecessor utility function.

```
RBTreeGetPredecessor(node) {
   node = node⇢left
   while (node⇢right != null) {
      node = node⇢right
   }
   return node
}
```

PARTICIPATION ACTIVITY 9.15.1: Removal concepts.

1) The red-black tree removal algorithm uses the normal BST removal algorithm.

- ○ True
- ○ False

2) RBTreeRemove uses the BST search algorithm.

- True
- False

3) Removing a red node with RBTreeRemoveNode will never cause RBTreePrepareForRemoval to be called.

- True
- False

4) Although RBTreeRemoveNode uses the node's predecessor, the algorithm could also use the successor.

- True
- False

Removal utility functions

Utility functions help simplify red-black tree removal code. The `RBTreeGetSibling` function returns the sibling of a node. The `RBTreeIsNonNullAndRed` function returns true only if a node is non-null and red, false otherwise. The `RBTreeIsNullOrBlack` function returns true if a node is null or black, false otherwise. The `RBTreeAreBothChildrenBlack` function returns true only if both of a node's children are black. Each utility function considers a null node to be a black node.

Figure 9.15.4: RBTreeGetSibling algorithm.

```
RBTreeGetSibling(node) {
   if (node⇢parent != null) {
      if (node == node⇢parent⇢left)
         return node⇢parent⇢right
      return node⇢parent⇢left
   }
   return null
}
```

Figure 9.15.5: RBTreeIsNonNullAndRed algorithm.

```
RBTreeIsNonNullAndRed(node) {
   if (node == null)
      return false
   return (node⇢color == red)
}
```


Figure 9.15.6: RBTreeIsNullOrBlack algorithm.

```
RBTreeIsNullOrBlack(node) {
   if (node == null)
      return true
   return (node⇢color == black)
}
```

Figure 9.15.7: RBTreeAreBothChildrenBlack algorithm.

```
RBTreeAreBothChildrenBlack(node) {
   if (node⇢left != null && node⇢left⇢color == red)
      return false
   if (node⇢right != null && node⇢right⇢color == red)
      return false
   return true
}
```


PARTICIPATION ACTIVITY 9.15.2: Removal utility functions.

1) Under what circumstance will RBTreeAreBothChildrenBlack always return true?

- When both of the node's children are null
- When both of the node's children are non-null
- When the node's left child is null
- When the node's right child is null

2) RBTreeIsNonNullAndRed will not work properly when passed a null node.

- True
- False

3) What will be returned when RBTreeGetSibling is called on a node with a null parent?

- A pointer to the node
- null
- A pointer to the tree's root
- Undefined/unknown

4) RBTreeIsNullOrBlack requires the node to be a leaf.

- True
- False

5) Which function(s) have a precondition that the node parameter must be non-null?

- All 4 functions have a precondition that the node parameter must be non-null
- RBTreeGetSibling only
- RBTreeIsNonNullAndRed and RBTreeIsNullOrBlack
- RBTreeGetSibling and RBTreeAreBothChildrenBlack

6) If RBTreeGetSibling returns a non-null, red node, then the node's parent must be non-null and black.

- True
- False

Prepare-for-removal algorithm overview

Preparation for removing a black node requires altering the number of black nodes along paths to preserve the black-path-length property. The `RBTreePrepareForRemoval` algorithm uses 6 utility functions that analyze the tree and make appropriate alterations when each of the 6 cases is encountered. The utility functions return true if the case is encountered, and false otherwise. If case 1, 3, or 4 is encountered, `RBTreePrepareForRemoval` will return after calling the utility function. If case 2, 5, or 6 is encountered, additional cases must be checked.

Figure 9.15.8: RBTreePrepareForRemoval pseudocode.

```
RBTreePrepareForRemoval(tree, node) {
   if (RBTreeTryCase1(tree,node))
      return

   sibling = RBTreeGetSibling(node)
   if (RBTreeTryCase2(tree, node, sibling))
      sibling = RBTreeGetSibling(node)
   if (RBTreeTryCase3(tree, node, sibling))
      return
   if (RBTreeTryCase4(tree, node, sibling))
      return
   if (RBTreeTryCase5(tree, node, sibling))
      sibling = RBTreeGetSibling(node)
   if (RBTreeTryCase6(tree, node, sibling))
      sibling = RBTreeGetSibling(node)

   sibling⇢color = node⇢parent⇢color
   node⇢parent⇢color = black
   if (node == node⇢parent⇢left) {
      sibling⇢right⇢color = black
      RBTreeRotateLeft(tree, node⇢parent)
   }
   else {
      sibling⇢left⇢color = black
      RBTreeRotateRight(tree, node⇢parent)
   }
}
```


PARTICIPATION ACTIVITY 9.15.3: Prepare-for-removal algorithm.

1) If the condition for any of the first 6 cases is met, then an adjustment specific to the case is made and the algorithm returns without processing any additional cases.

- ○ True
- ○ False

2) Why is no preparation action

required if the node is red?

- ○ A red node will never have children
- ○ A red node will never be the root of the tree
- ○ A red node always has a black parent node
- ○ Removing a red node will not change the number of black nodes along any path

3) Re-computation of the sibling node after case RBTreeTryCase2, RBTreeTryCase5, or RBTreeTryCase6 implies that these functions may be doing what?

- ○ Recoloring the node or the node's parent
- ○ Recoloring the node's uncle or the node's sibling
- ○ Rotating at one of the node's children
- ○ Rotating at the node's parent or the node's sibling

4) RBTreePrepareForRemoval performs the check `node⇢parent == null` on the first line. What other check is equivalent and could be used in place of the code `node⇢parent == null`?

○ tree->root == null

Prepare-for-removal algorithm cases

Preparation for removing a node first checks for each of the six cases, performing the operations below.

1. If the node is red or the node's parent is null, then return.
2. If the node has a red sibling, then color the parent red and the sibling black. If the node is the parent's left child then rotate left at the parent, otherwise rotate right at the parent. Continue to the next step.
3. If the node's parent is black and both children of the node's sibling are black, then color the sibling red, recursively call on the node's parent, and return.
4. If the node's parent is red and both children of the node's sibling are black, then color the parent black, color the sibling red, then return.
5. If the sibling's left child is red, the sibling's right child is black, and the node is the left child of the parent, then color the sibling red and the left child of the sibling black. Then rotate right at the sibling and continue to the next step.
6. If the sibling's left child is black, the sibling's right child is red, and the node is the right child of the parent, then color the sibling red and the right child of the sibling black. Then rotate left at the sibling and continue to the next step.
7. Color the sibling the same color as the parent and color the parent black.
8. If the node is the parent's left child, then color the sibling's right child black and rotate left at the parent. Otherwise color the sibling's left child black and rotate right at the parent.

Table 9.15.1: Prepare-for-removal algorithm case descriptions.

Case #	Condition	Action if condition is true	Process additional cases after action?
1	Node is red or node's parent is null.	None.	No
2	Sibling node is red.	Color parent red and sibling black. If node is left child of parent, rotate left at parent node, otherwise rotate right at parent node.	Yes
3	Parent is black and both of sibling's children are black.	Color sibling red and call removal preparation function on parent.	No
4	Parent is red and both of sibling's children are black.	Color parent black and sibling red.	No
5	Sibling's left child is red, sibling's right child is black, and node is left child of parent.	Color sibling red and sibling's left child black. Rotate right at sibling.	Yes
6	Sibling's left child is black, sibling's right child is red, and node is right child of parent.	Color sibling red and sibling's right child black. Rotate left at sibling.	Yes

©zyBooks 04/19/21 23:08 926259
NAT FOSTER
JHUEN605202JavaSpring2021

Table 9.15.2: Prepare-for-removal algorithm case code.

Case #	Code

1	`RBTreeTryCase1(tree, node) {`
	`   if (node→color == red || node→parent == null)`
	`      return true`
	`   else`
	`      return false // not case 1`
	`}`

2	`RBTreeTryCase2(tree, node, sibling) {`
	`   if (sibling→color == red) {`
	`      node→parent→color = red`
	`      sibling→color = black`
	`      if (node == node→parent→left)`
	`         RBTreeRotateLeft(tree, node→parent)`
	`      else`
	`         RBTreeRotateRight(tree, node→parent)`
	`      return true`
	`   }`
	`   return false // not case 2`
	`}`

3	`RBTreeTryCase3(tree, node, sibling) {`
	`   if (node→parent→color == black &&`
	`       RBTreeAreBothChildrenBlack(sibling)) {`
	`      sibling→color = red`
	`      RBTreePrepareForRemoval(tree, node→parent)`
	`      return true`
	`   }`
	`   return false // not case 3`
	`}`

4	`RBTreeTryCase4(tree, node, sibling) {`
	`   if (node→parent→color == red &&`
	`       RBTreeAreBothChildrenBlack(sibling)) {`
	`      node→parent→color = black`
	`      sibling→color = red`
	`      return true`
	`   }`
	`   return false // not case 4`
	`}`

	`RBTreeTryCase5(tree, node, sibling) {`
	`   if (RBTreeIsNonNullAndRed(sibling-→left) &&`
	`       RBTreeIsNullOrBlack(sibling→right) &&`
	`       node == nodeparent→left) {`
	`      sibling→color = red`

©zyBooks 04/19/21 23:08 926259
NAT FOSTER
JHUEN605202JavaSpring2021

PARTICIPATION ACTIVITY | 9.15.4: Removal preparation, case 4.

Animation content:

undefined

Animation captions:

1. In the above tree, all paths from root to null leaves have 3 black nodes.
2. Preparation for removal of node 62 encounters case 4, since the node's parent is red and both children of the sibling are black (null).
3. The parent is colored black and the sibling is colored red.
4. The preparation leaves the tree in a state where node 62 can be removed and all red-black tree requirements would be met.

PARTICIPATION ACTIVITY | 9.15.5: Removal preparation for a node can encounter more than 1 case.

Animation content:

undefined

Animation captions:

1. Preparation for removal of node 75 first encounters case 2 in RBTreePrepareForRemoval.
2. After making alterations for case 2, the code proceeds to additional case checks, ending after case 4 alterations.
3. In the resulting tree, node 75 can be removed via BSTRemove and all red-black tree requirements will hold.

PARTICIPATION ACTIVITY | 9.15.6: Prepare-for-removal algorithm cases.

RBTreeTryCase4 **RBTreeTryCase5** **RBTreeTryCase3** **RBTreeTryCase6**

RBTreeTryCase2 **RBTreeTryCase1**

This case function always returns true if passed a node with a red sibling.

This case function finishes preparation exclusively by recoloring nodes.

This case function never returns true if the node is the right child of the node's parent.

This case function never alters the tree.

When this case function returns true, a left rotation at the node's sibling will have just taken place.

This case function recursively calls RBTreePrepareForRemoval if the node's parent and both children of the node's sibling are black.

Reset