

# 17.1 Binary trees

## Binary tree basics

In a list, each node has up to one successor. In a **binary tree**, each node has up to two children, known as a *left child* and a *right child*. "Binary" means two, referring to the two children. Some more definitions related to a binary tree:

- **Leaf**: A tree node with no children.
- **Internal node**: A node with at least one child.
- **Parent**: A node with a child is said to be that child's parent. A node's **ancestors** include the node's parent, the parent's parent, etc., up to the tree's root.
- **Root**: The one tree node with no parent (the "top" node).

Another section discusses binary tree usefulness; this section introduces definitions.

Below, each node is represented by just the node's key, as in B, although the node may have other data.

PARTICIPATION  
ACTIVITY

17.1.1: Binary tree basics.



### Animation captions:

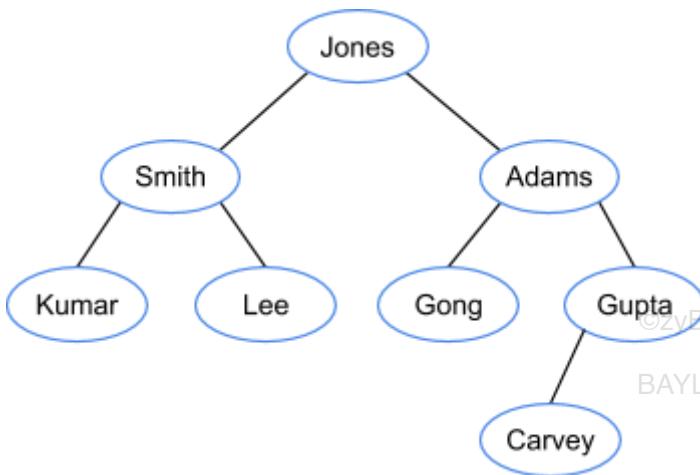
1. In a list, each node has up to one successor.
2. In a binary tree, each node has up to two children.
3. A tree is normally drawn vertically. Edge arrows are optional.
4. A node can have a left child and right child. A node with a child is called the child's parent.
5. First node: Root node. Node without child: Leaf node. Node with child: Internal nodes.

PARTICIPATION  
ACTIVITY

17.1.2: Binary tree basics.



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

1) Root node: \_\_\_\_\_.

**Check****Show answer**

2) Smith's left child: \_\_\_\_\_.

**Check****Show answer**

3) The tree has four leaf nodes: Kumar,  
Lee, Gong, and \_\_\_\_\_.

**Check****Show answer**

4) How many internal nodes does the tree  
have?

**Check****Show answer**

5) The tree has an internal node, Gupta,  
with only one child rather than two. Is  
the tree still a binary tree? Type yes or  
no.

**Check****Show answer**

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



## Depth, level, and height

A few additional terms:

- The link from a node to a child is called an **edge**.
- A node's **depth** is the number of edges on the path from the root to the node. The root node thus has depth 0.
- All nodes with the same depth form a tree **level**.
- A tree's **height** is the largest depth of any node. A tree with just one node has height 0.

©zyBooks 04/25/21 07:54 488201

xiang zhao

PARTICIPATION  
ACTIVITY

17.1.3: Binary tree terminology: height, depth, and level.

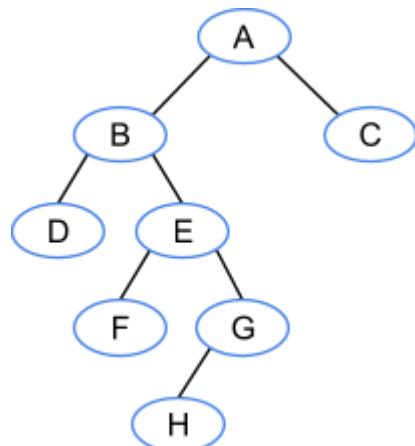


### Animation captions:

1. The binary tree has edges A to B, A to C, and C to D.
2. A node's depth is the number of edges from the root to the node.
3. Nodes with the same depth form a level.
4. A tree's height is the largest depth of any node.

PARTICIPATION  
ACTIVITY

17.1.4: Binary tree height, depth, and level.



1) A's depth is 1.

- True  
 False

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

2) E's depth is 2.

- True  
 False



3) B and C form level 1.



True False

4) D, F, and H form level 2.

 True False

5) The tree's height is 4.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021 True False

6) A tree with just one node has height 0.

 True False

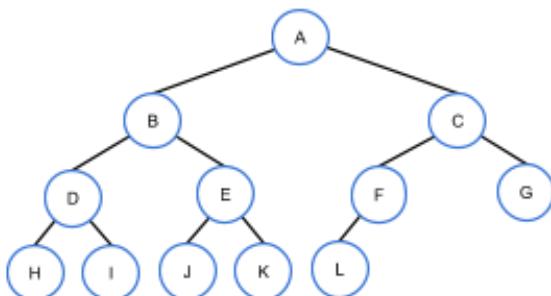
## Special types of binary trees

Certain binary tree structures can affect the speed of operations on the tree. The following describe special types of binary trees:

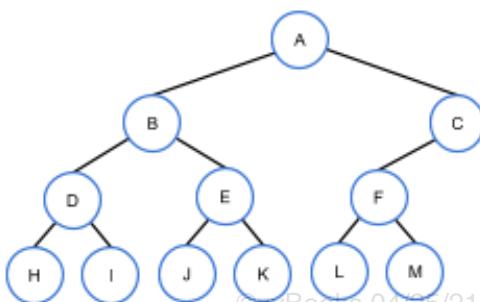
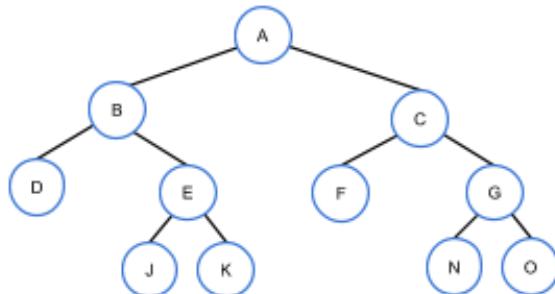
- A binary tree is **full** if every node contains 0 or 2 children.
- A binary tree is **complete** if all levels, except possibly the last level, contain all possible nodes and all nodes in the last level are as far left as possible.
- A binary tree is **perfect**, if all internal nodes have 2 children and all leaf nodes are at the same level.

Figure 17.1.1: Special types of binary trees: full, complete, perfect.

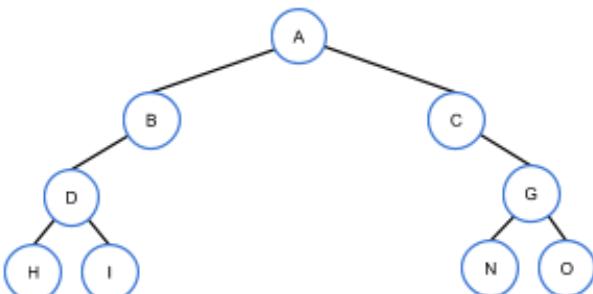
©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



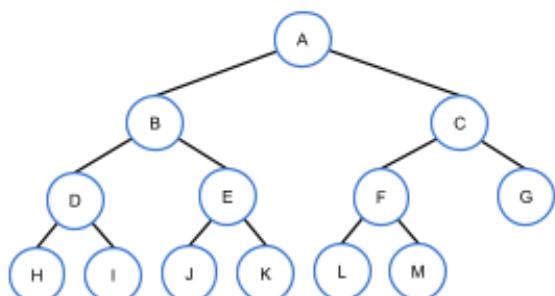
Not full, complete, not perfect

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

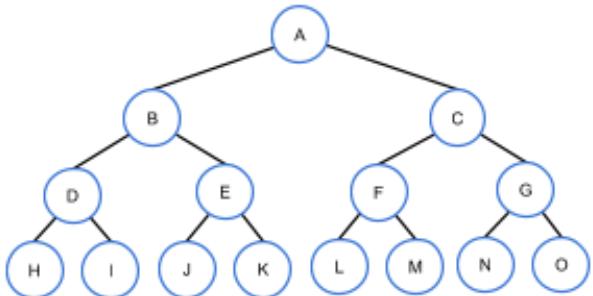
Full, not complete, not perfect



Not full, not complete, not perfect



Full, complete, not perfect

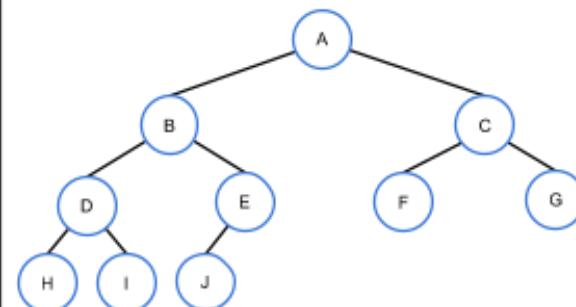
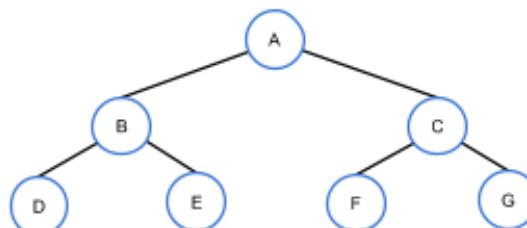
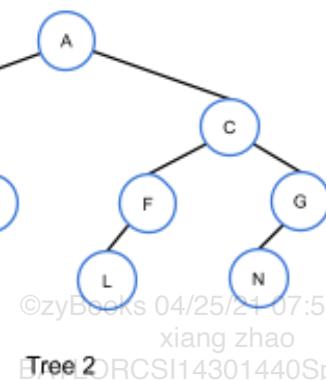
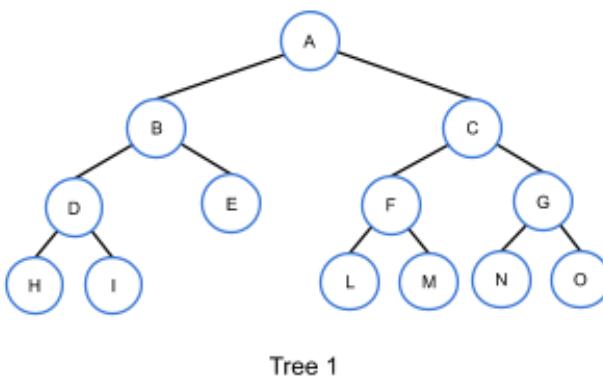


Full, complete, perfect

**PARTICIPATION ACTIVITY**

17.1.5: Identifying special types of binary trees.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

**Full, complete, perfect****Not full, complete, not perfect****Full, not complete, not perfect****Not full, not complete, not perfect**

Tree 1

Tree 2

Tree 3

Tree 4

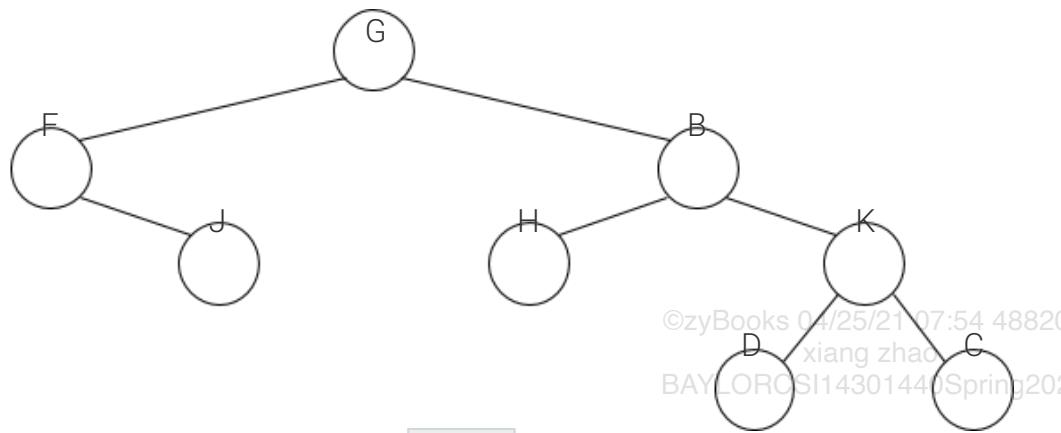
**Reset**

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

**CHALLENGE ACTIVITY**

17.1.1: Binary trees.

**Start**



What is the root node?

Ex: A

What is the parent of node J?

Ex: A

What are the ancestors of node J?

Ex: A, B, C

What are the leaf nodes?

Ex: A, B, C

How many internal nodes does the tree have?

Ex: 9

1

2

3

**Check**

**Next**

## 17.2 Applications of trees

### File systems

Trees are commonly used to represent hierarchical data. A tree can represent files and directories in a file system, since a file system is a hierarchy.

**PARTICIPATION ACTIVITY**

17.2.1: A file system is a hierarchy that can be represented by a tree.



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

### Animation content:

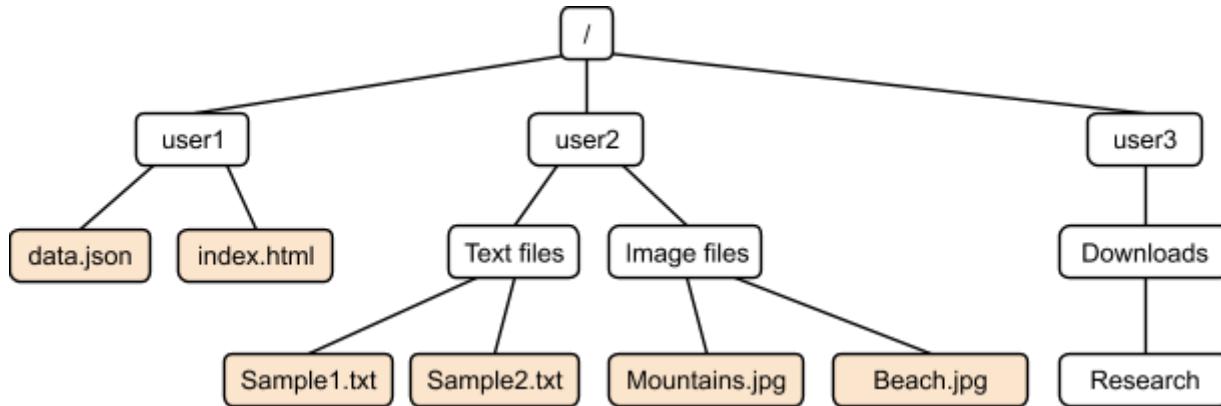
undefined

### Animation captions:

1. A tree representing a file system has the filesystem's root directory ("/"), represented by the root node.
2. The root contains 2 configuration text files and 2 additional directories: user1 and user2.
3. Directories contain additional entries. Only empty directories will be leaf nodes. All files are leaf nodes.

**PARTICIPATION ACTIVITY****17.2.2: Analyzing a file system tree.**

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



1) What is the depth of the "Mountains.jpg" file node?

- 3
- 4

2) What is the height of this tree?

- 3
- 4
- 14

3) What is the parent of the "Text files" node?

- The "user2" directory node
- The "Image files" directory node
- The "Sample1.txt" file node.

4) Which operation would increase the height of the tree?

- Adding a new file into the user1 directory
- Adding a new directory into the "Image files" directory

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

Adding a new directory into the "Research" directory

**PARTICIPATION ACTIVITY**

17.2.3: File system trees.



- 1) A file in a file system tree is always a leaf node.  
 True  
 False
- 2) A directory in a file system tree is always an internal node.  
 True  
 False
- 3) Using a tree data structure to implement a file system requires that each directory node support a variable number of children.  
 True  
 False

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



## Binary space partitioning

**Binary space partitioning (BSP)** is a technique of repeatedly separating a region of space into 2 parts and cataloging objects contained within the regions. A **BSP tree** is a binary tree used to store information for binary space partitioning. Each node in a BSP tree contains information about a region of space and which objects are contained in the region.

In graphics applications, a BSP tree can be used to store all objects in a multidimensional world. The BSP tree can then be used to efficiently determine which objects must be rendered on screen. The viewer's position in space is used to perform a lookup within the BSP tree. The lookup quickly eliminates a large number of objects that are not visible and therefore should not be rendered.

**PARTICIPATION ACTIVITY**

17.2.4: A BSP tree is used to quickly determine which objects do not need to be rendered.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



## Animation content:

undefined

## Animation captions:

1. Data for a large, open 2-D world contains many objects. Only a few objects are visible on screen at any given moment.
2. Avoiding rendering off-screen objects is crucial for realtime graphics. But checking the intersection of all objects with the screen's rectangle is too time consuming.
3. A BSP tree represents partitioned space. The root represents the entire world and stores a list of all objects in the world, as well as the world's geometric boundary.
4. The root's left child represents the world's left half. The node stores information about the left half's geometric boundary, and a list of all objects contained within.
5. The root's right child contains similar information for the right half.
6. Using the screen's position within the world as a lookup into the BSP tree quickly yields the right node's list of objects. A large number of objects are quickly eliminated from the list of potential objects on screen.
7. Further partitioning makes the tree even more useful.

### PARTICIPATION ACTIVITY

#### 17.2.5: Binary space partitioning.



- 1) When traversing down a BSP tree, half the objects are eliminated each level.

- True
- False



- 2) A BSP implementation could choose to split regions in arbitrary locations, instead of right down the middle.

- True
- False



- 3) In the animation, if the parts of the screen were in 2 different regions, then all objects from the 2 regions would have to be analyzed when rendering.

- True
- False



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

- 4) BSP can be used in 3-D graphics as well as 2-D.

- True
- False



## Using trees to store collections

Most of the tree data structures discussed in this book serve to store a collection of values. Numerous tree types exist to store data collections in a structured way that allows for fast searching, inserting, and removing of values.

©zyBooks 04/25/21 07:54 488201

xiang zhao

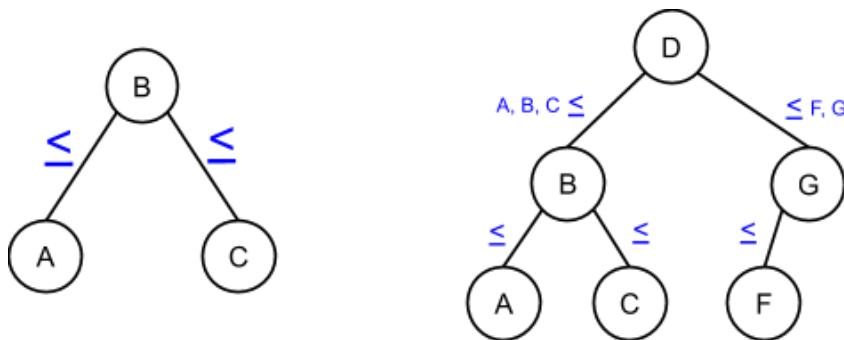
BAYLORCSI14301440Spring2021

## 17.3 Binary search trees

### Binary search trees

An especially useful form of binary tree is a **binary search tree** (BST), which has an ordering property that any node's left subtree keys  $\leq$  the node's key, and the right subtree's keys  $\geq$  the node's key. That property enables fast searching for an item, as will be shown later.

Figure 17.3.1: BST ordering property: For three nodes, left child is less-than-or-equal-to parent, parent is less-than-or-equal-to right child. For more nodes, all keys in subtrees must satisfy the property, for every node.



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

PARTICIPATION  
ACTIVITY

17.3.1: BST ordering properties.



**Animation content:**

**undefined****Animation captions:**

1. BST ordering property: Left subtree's keys  $\leq$  node's key, right subtree's keys  $\geq$  node's key.
2. All keys in subtree must obey the ordering property. Not a BST.
3. All keys in subtree must obey the ordering property. Not a BST.
4. All keys in subtree must obey the ordering property. Valid BST.

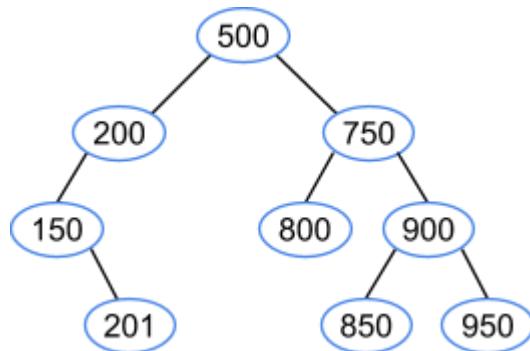
©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

**PARTICIPATION ACTIVITY**

17.3.2: Binary search tree: Basic ordering property.



- 1) Does node 900 and the node's subtrees obey the BST ordering property?

 Yes  
 No

- 2) Does node 750 and the node's subtrees obey the BST ordering property?

 Yes  
 No

- 3) Does node 150 and the node's subtrees obey the BST ordering property?

 Yes  
 No

- 4) Does node 200 and the node's subtrees obey the BST ordering property?

 Yes  
 No©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

- 5) Is the tree a binary search tree?

 Yes

No

6) Is the tree a binary tree?

 Yes No

7) Would inserting 300 as the right child of 200 obey the BST ordering property (considering only nodes 300, 200, and 500)?

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

 Yes No

## Searching

To **search** nodes means to find a node with a desired key, if such a node exists. A BST may yield faster searches than a list. Searching a BST starts by visiting the root node (which is the first currentNode below):

Figure 17.3.2: Searching a BST.

```
if (currentNode->key == desiredKey) {
    return currentNode; // The desired node was found
}
else if (desiredKey < currentNode->key) {
    // Visit left child, repeat
}
else if (desiredKey > currentNode->key) {
    // Visit right child, repeat
}
```

If a child to be visited doesn't exist, the desired node does not exist. With this approach, only a small fraction of nodes need be compared.

**PARTICIPATION ACTIVITY**

17.3.3: A BST may yield faster searches than a list.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

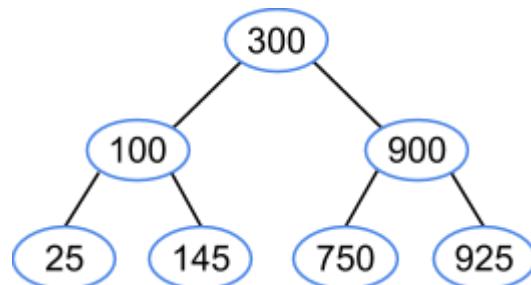
## Animation captions:

1. Searching a 7-node list may require up to 7 comparisons.
2. In a BST, if desired key equals current node's key, return found. If less, descend to left child. If greater, descend to right child.

3. Searching a BST may require fewer comparisons, in this case 3 vs. 7.

**PARTICIPATION ACTIVITY**

## 17.3.4: Searching a BST.



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

- 1) In searching for 145, what node is visited first?

**Check**[Show answer](#)

- 2) In searching for 145, what node is visited second?

**Check**[Show answer](#)

- 3) In searching for 145, what node is visited third?

**Check**[Show answer](#)

- 4) Which nodes would be visited when searching for 900? Write nodes in order visited, as: 5, 10

**Check**[Show answer](#)

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

- 5) Which nodes would be visited when searching for 800? Write nodes in order visited, as: 5, 10, 15



**Check****Show answer**

- 6) What is the worst case (largest) number of nodes visited when searching for a key?

**Check****Show answer**

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

## BST search runtime

Searching a BST in the worst case requires  $H + 1$  comparisons, meaning  $O(H)$  comparisons, where  $H$  is the tree height. Ex: A tree with a root node and one child has height 1; the worst case visits the root and the child:  $1 + 1 = 2$ . A major BST benefit is that an  $N$ -node binary tree's height may be as small as  $O(\log N)$ , yielding extremely fast searches. Ex: A 10,000 node list may require 10,000 comparisons, but a 10,000 node BST may require only 14 comparisons.

A binary tree's height can be minimized by keeping all levels full, except possibly the last level. Such an "all-but-last-level-full" binary tree's height is  $H = \lfloor \log_2 N \rfloor$ .

Table 17.3.1: Minimum binary tree heights for  $N$  nodes are equivalent to  $\lfloor \log_2 N \rfloor$ .

Nodes N	Height H	$\log_2 N$	$\lfloor \log_2 N \rfloor$	Nodes per level
1	0	0	0	1
2	1	1	1	1/1
3	1	1.6	1	1/2
4	2	2	2	1/2/1
5	2	2.3	2	1/2/2
6	2	2.6	2	1/2/3
7	2	2.8	2	1/2/4
8	3	3	3	1/2/4/1
9	3	3.2	3	1/2/4/2
...				

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

15	3	3.9	3	1/2/4/8
16	4	4	4	1/2/4/8/1

**PARTICIPATION ACTIVITY**

17.3.5: Searching a perfect BST with N nodes requires only  $O(\log N)$ .

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

**Animation captions:**

1. A perfect binary tree has height  $\lfloor \log_2 N \rfloor$ .
2. A perfect binary tree search is  $O(H)$ , so  $O(\log N)$ .
3. Searching a BST may be faster than searching a list.

**PARTICIPATION ACTIVITY**

17.3.6: Searching BSTs with N nodes.



What is the worst case (largest) number of comparisons given a BST with N nodes?

1) Perfect BST with  $N = 7$

- $\lfloor \log_2 N \rfloor$
- $\lfloor \log_2 N \rfloor + 1$
- $N$

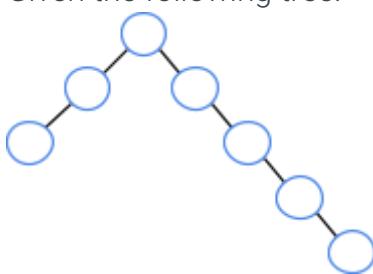


2) Perfect BST with  $N = 31$

- 31
- 4
- 5



3) Given the following tree.



- 3
- 5

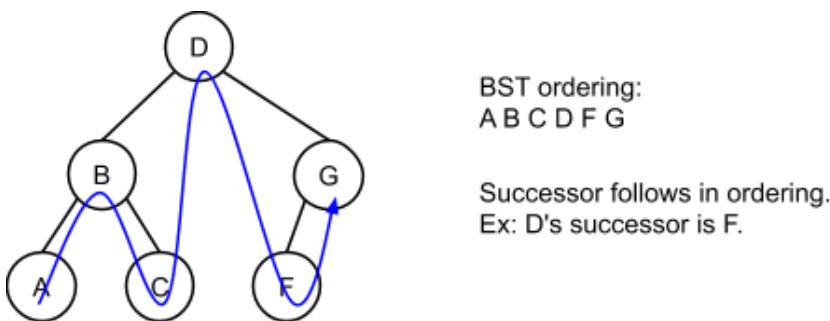
©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

## Successors and predecessors

A BST defines an ordering among nodes, from smallest to largest. A BST node's **successor** is the node that comes after in the BST ordering, so in A B C, A's successor is B, and B's successor is C. A BST node's **predecessor** is the node that comes before in the BST ordering.

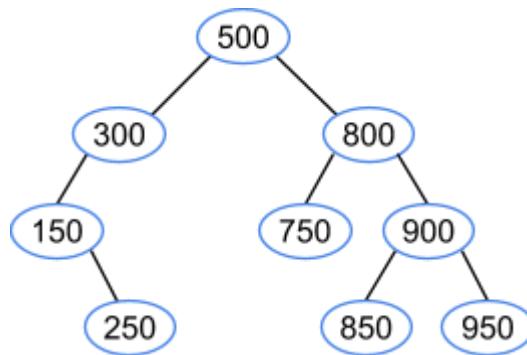
If a node has a right subtree, the node's successor is that right subtree's leftmost child: Starting from the right subtree's root, follow left children until reaching a node with no left child (may be that 88201 subtree's root itself). If a node doesn't have a right subtree, the node's successor is the first ancestor having this node in a left subtree. Another section provides an algorithm for printing a BST's nodes in order.

Figure 17.3.3: A BST defines an ordering among nodes.



### PARTICIPATION ACTIVITY

17.3.7: Binary search tree: Defined ordering.



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

- 1) The first node in the BST ordering is 150.

True

False

- 2) 150's successor is 250.

True



False

3) 250's successor is 300.

 True False

4) 500's successor is 850.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

 True False

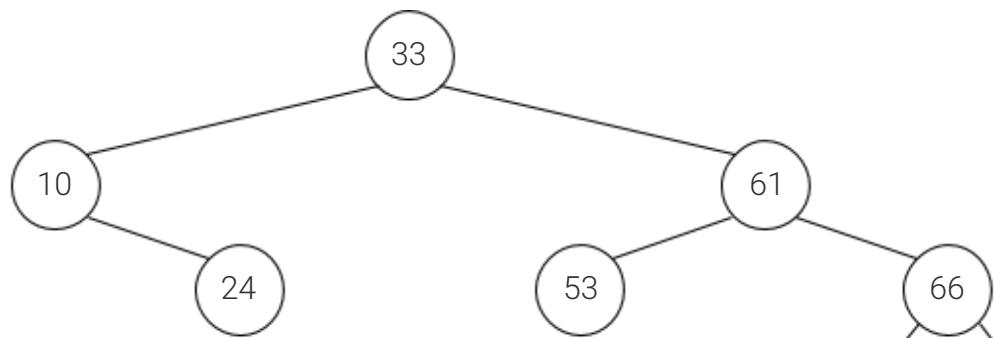
5) 950's successor is 150.

 True False

6) 950's predecessor is 900.

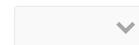
 True False**CHALLENGE ACTIVITY**

17.3.1: Binary search trees.

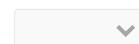
**Start**

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

Does node 10 and the node's subtrees obey the BST ordering property?



Does node 61 and the node's subtrees obey the BST ordering property?



Is the tree a BST?

**1**

2

3

4

5

**Check****Next**

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

## 17.4 BST 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. A simple BST search algorithm checks the current node (initially the tree's root), returning that node as a match, else assigning the current node with the left (if key is less) or right (if key is greater) child and repeating. If such a child is null, the algorithm returns null (matching node not found).

**PARTICIPATION ACTIVITY**

17.4.1: BST search algorithm.



### Animation content:

undefined

### Animation captions:

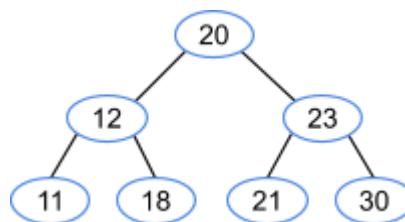
1. BST search algorithm checks current node, returning a match if found. Otherwise, assigns current node with left (if key is less) or right (if key is greater) child and continues search.
2. If the child to be visited does not exist, the algorithm returns null indicating no match found.

**PARTICIPATION ACTIVITY**

17.4.2: BST search algorithm.



Consider the following tree.



©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

- 1) When searching for key 21, what node is visited first?

 20


21

- 2) When searching for key 21, what node is visited second?

 12 23

- 3) When searching for key 21, what node is visited third?

 21 30

- 4) If the current node matches the key, when does the algorithm return the node?

 Immediately Upon exiting the loop

- 5) If the child to be visited is null, when does the algorithm return null?

 Immediately Upon exiting the loop

- 6) What is the maximum loop iterations for a perfect binary tree with 7 nodes, if a node matches?

 3 7

- 7) What is the maximum loop iterations for a perfect binary tree with 7 nodes, if no node matches?

 3 7

- 8) What is the maximum loop iterations for a perfect binary tree with 255 nodes?

 8 255

- 9) Suppose node 23 was instead 21, meaning two 21 nodes exist (which is



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



allowed in a BST). When searching for 21, which node will be returned?

- Leaf
- Internal

**PARTICIPATION ACTIVITY**

## 17.4.3: BST search algorithm decisions.

©zyBooks 04/25/21 07:54 488201

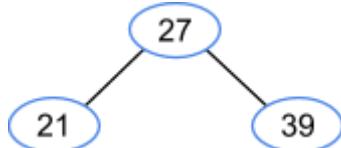
xiang zhao

BAYLORCSI14301440Spring2021



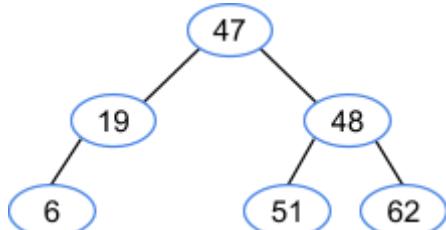
Determine cur's next assignment given the key and current node.

1) key = 40, cur = 27



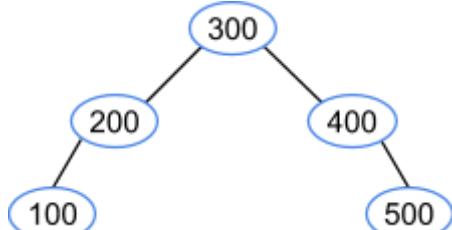
- 27
- 21
- 39

2) key = 6, cur = 47



- 6
- 19
- 48

3) key 350, cur = 400



- Search terminates and returns null.
- 400
- 500

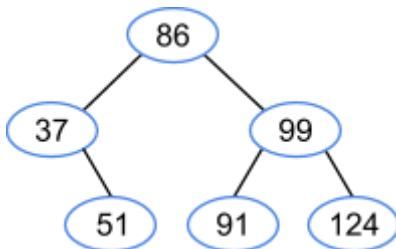
4) key 91, cur = 99



©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021



- 86
- 91
- 124

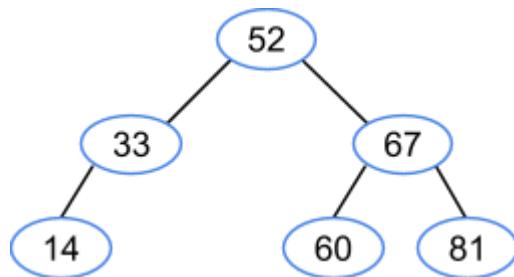
©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

**PARTICIPATION ACTIVITY**

17.4.4: Tracing a BST search.



Consider the following tree. If node does not exist, enter null.



- 1) When searching for key 45, what node is visited first?



[Show answer](#)

- 2) When searching for key 45, what node is visited second?



[Show answer](#)

- 3) When searching for key 45, what node is visited third?



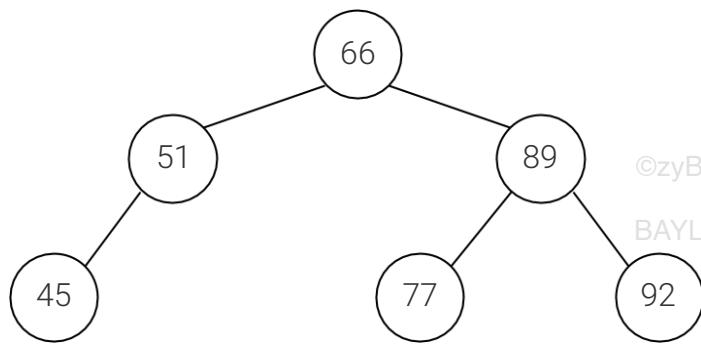
[Show answer](#)

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

**CHALLENGE ACTIVITY**

17.4.1: BST search algorithm.



[Start](#)

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

What does BSTSearch(tree, 92) return?

1

2

[Check](#)[Next](#)

## 17.5 BST insert algorithm

Given a new node, a BST **insert** operation inserts the new node in a proper location obeying the BST ordering property. A simple BST insert algorithm compares the new node with the current node (initially the root).

- *Insert as left child*: If the new node's key is less than the current node, and the current node's left child is null, the algorithm assigns that node's left child with the new node.
- *Insert as right child*: If the new node's key is greater than or equal to the current node, and the current node's right child is null, the algorithm assigns the node's right child with the new node.
- *Search for insert location*: If the left (or right) child is not null, the algorithm assigns the current node with that child and continues searching for a proper insert location.

PARTICIPATION  
ACTIVITY

17.5.1: Binary search tree insertions.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

### Animation content:

undefined

## Animation captions:

1. A node inserted into an empty tree will become the tree's root.
2. The BST is searched to find a suitable location to insert the new node as a leaf node.

**PARTICIPATION ACTIVITY**

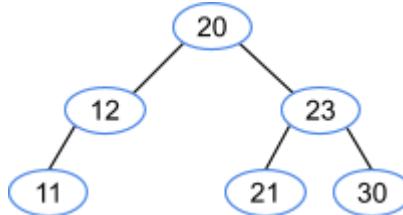
17.5.2: BST insert algorithm.

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

Consider the following tree.



1) Where will a new node 18 be inserted?

- 12's right child
- 11's right child

2) Where will a new node 11 be inserted?

(So two nodes of 11 will exist).

- 11's left child
- 11's right child

3) Assume a perfect 7-node BST. How many algorithm loop iterations will occur for an insert?

- 3
- 7

4) Assume a perfect 255-node BST. How many algorithm loop iterations will occur for an insert?

- 8
- 255

©zyBooks 04/25/21 07:54 488201

xiang zhao

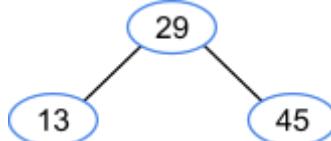
BAYLORCSI14301440Spring2021

**PARTICIPATION ACTIVITY**

17.5.3: BST insert algorithm decisions.

Determine the insertion algorithm's next step given the new node's key and the current node.

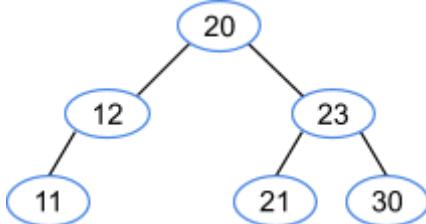
1) key = 7, currentNode = 29



- currentNode->left = node
- currentNode = currentNode->right
- currentNode = currentNode->left

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

2) key = 18, currentNode = 12

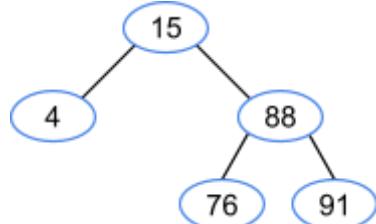


- currentNode->left = node
- currentNode->right = node
- currentNode = currentNode->right

3) key = 87, currentNode = null, tree->root = null (empty tree)

- tree->root = node
- currentNode->right = node
- currentNode->left = node

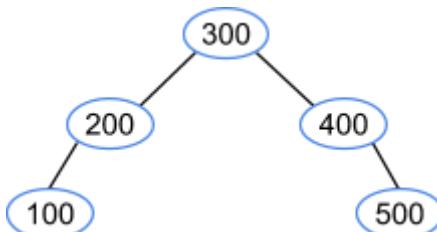
4) key = 53, currentNode = 76



- currentNode->left = node
- currentNode->right = node
- tree->root = node

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

5) key = 600, currentNode = 400



- currentNode->left = node
- currentNode =  
currentNode->right
- currentNode->right = node

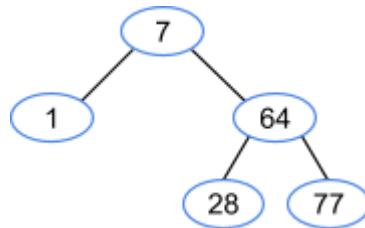
©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

PARTICIPATION  
ACTIVITY

17.5.4: Tracing BST insertions.



Consider the following tree.



- 1) When inserting a new node with key 35,  
what node is visited first?




**Check**

[Show answer](#)

- 2) When inserting a new node with key 35,  
what node is visited second?




**Check**

[Show answer](#)

- 3) When inserting a new node with key 35,  
what node is visited third?




**Check**

[Show answer](#)

- 4) Where is the new node inserted? Type:  
left or right



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

[Check](#)[Show answer](#)

## BST insert algorithm complexity

©zyBooks 04/25/21 07:54 488201

xiang zhao

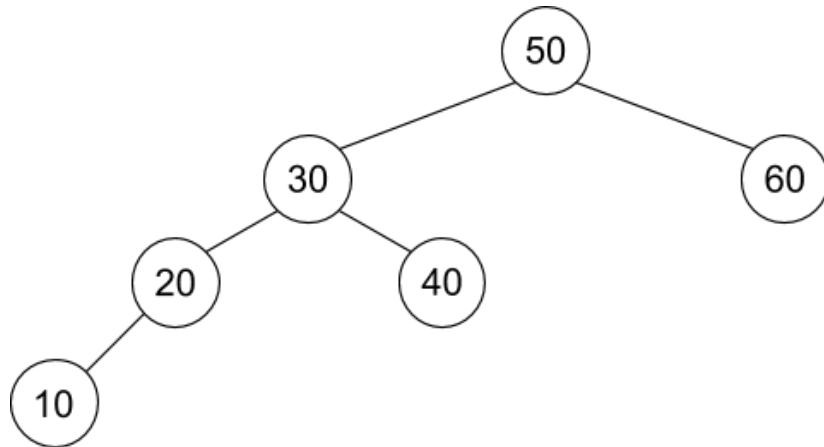
BAYLORCSI14301440Spring2021

The BST insert algorithm traverses the tree from the root to a leaf node to find the insertion location. One node is visited per level. A BST with  $N$  nodes has at least  $\log_2 N$  levels and at most  $N$  levels. Therefore, the runtime complexity of insertion is best case  $O(\log N)$  and worst case  $O(N)$ .

The space complexity of insertion is  $O(1)$  because only a single pointer is used to traverse the tree to find the insertion location.

**CHALLENGE ACTIVITY**

### 17.5.1: BST insert algorithm.

[Start](#)

Where will a new node 21 be inserted?

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021



child of node



1	2	3
---	---	---

[Check](#)[Next](#)

Exploring further:

- [Binary search tree visualization](#)

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

## 17.6 BST remove algorithm

Given a key, a BST **remove** operation removes the first-found matching node, restructuring the tree to preserve the BST ordering property. The algorithm first searches for a matching node just like the search algorithm. If found (call this node X), the algorithm performs one of the following sub-algorithms:

- *Remove a leaf node:* If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with null. Else, if X was the root, the root pointer is assigned with null, and the BST is now empty.
- *Remove an internal node with single child:* If X has a parent (so X is not the root), the parent's left or right child (whichever points to X) is assigned with X's single child. Else, if X was the root, the root pointer is assigned with X's single child.
- *Remove an internal node with two children:* This case is the hardest. First, the algorithm locates X's successor (the leftmost child of X's right subtree), and copies the successor to X. Then, the algorithm recursively removes the successor from the right subtree.

PARTICIPATION ACTIVITY

17.6.1: BST remove: Removing a leaf, or an internal node with a single child.



### Animation captions:

1. Removing a leaf node: The parent's right child is assigned with null.
2. Remove an internal node with a single child: The parent's right child is assigned with node's single child.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

PARTICIPATION ACTIVITY

17.6.2: BST remove: Removing internal node with two children.



### Animation captions:

1. Find successor: Leftmost child in node 25's right subtree is node 27.
2. Copy successor to current node.
3. Remove successor from right subtree.

Figure 17.6.1: BST remove algorithm.

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

```

BSTRemove(tree, key) {
    par = null
    cur = tree->root
    while (cur is not null) { // Search for node
        if (cur->key == key) { // Node found
            if (cur->left is null && cur->right is null) { // Remove leaf
                if (par is null) // Node is root
                    tree->root = null
                else if (par->left == cur)
                    par->left = null
                else
                    par->right = null
            }
            else if (cur->right is null) { // Remove node with only left child
                if (par is null) // Node is root
                    tree->root = cur->left
                else if (par->left == cur)
                    par->left = cur->left
                else
                    par->right = cur->left
            }
            else if (cur->left is null) { // Remove node with only right child
                if (par is null) // Node is root
                    tree->root = cur->right
                else if (par->left == cur)
                    par->left = cur->right
                else
                    par->right = cur->right
            }
            else { // Remove node with two children
                // Find successor (leftmost child of right subtree)
                suc = cur->right
                while (suc->left is not null)
                    suc = suc->left
                successorData = Create copy of suc's data
                BSTRemove(tree, suc->key) // Remove successor
                Assign cur's data with successorData
            }
            return // Node found and removed
        }
        else if (cur->key < key) { // Search right
            par = cur
            cur = cur->right
        }
        else { // Search left
            par = cur
            cur = cur->left
        }
    }
    return // Node not found
}

```

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

## BST remove algorithm complexity

The BST remove algorithm traverses the tree from the root to find the node to remove. When the node being removed has 2 children, the node's successor is found and a recursive call is made. One node is visited per level, and in the worst case scenario the tree is traversed twice from the root to a leaf. A BST with  $N$  nodes has at least  $\log_2 N$  levels and at most  $N$  levels. Therefore, the runtime complexity of removal is best case  $O(\log N)$  and worst case  $O(N)$ .

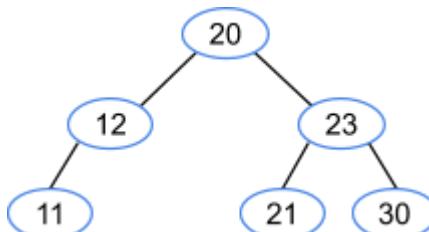
Two pointers are used to traverse the tree during removal. When the node being removed has 2 children, a third pointer and a copy of one node's data are also used, and one recursive call is made. Thus, the space complexity of removal is always  $O(1)$ .

### PARTICIPATION ACTIVITY

#### 17.6.3: BST remove algorithm.



Consider the following tree. Each question starts from the original tree. Use this text notation for the tree: (20 (12 (11, -), 23 (21, 30))). The - means the child does not exist.



1) What is the tree after removing 21?



- (20 (12 (11, -), 23 (-, 30)))
- (20 (12 (11, -), 23))

2) What is the tree after removing 12?



- (20 (- (11, -), 23 (21, 30)))
- (20 (11, 23 (21, 30)))

3) What is the tree after removing 20?



- (21 (12 (11, -), 23 (-, 30)))
- (23 (12 (11, -), 30 (21, -)))

4) Removing a node from an N-node



nearly-full  
BST has what computational complexity?

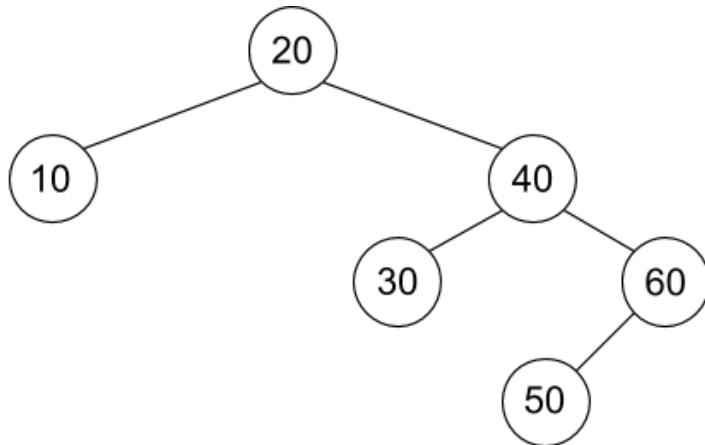
- O( $\log N$ )
- O( $N$ )

## CHALLENGE ACTIVITY

17.6.1: BST remove algorithm.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

Start



BSTRemove(tree, 50) executes.

What is the left child of 60?

What is the right child of 60?

1

2

3

**Check****Next**

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

## 17.7 BST inorder traversal

A **tree traversal** algorithm visits all nodes in the tree once and performs an operation on each node. An **inorder traversal** visits all nodes in a BST from smallest to largest, which is useful for example to print

the tree's nodes in sorted order. Starting from the root, the algorithm recursively prints the left subtree, the current node, and the right subtree.

Figure 17.7.1: BST inorder traversal algorithm.

```
BSTPrintInorder(node) {
    if (node is null)
        return

    BSTPrintInorder(node->left)
    Print node
    BSTPrintInorder(node->right)
}
```

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

**PARTICIPATION ACTIVITY**

17.7.1: BST inorder print algorithm.



### Animation captions:

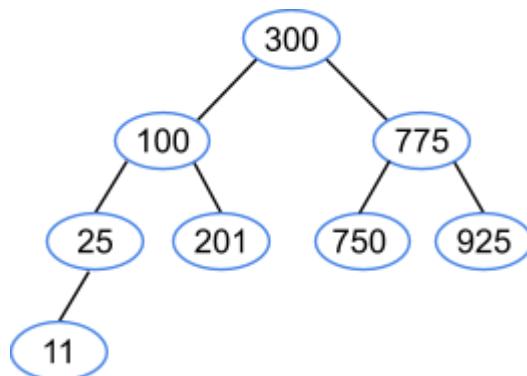
1. An inorder traversal starts at the root. Recursive call descends into left subtree.
2. When left done, current is printed, then recursively descend into right subtree.
3. Return from recursive call causes ascending back up the tree; left is done, so do current and right.
4. Continues similarly.

**PARTICIPATION ACTIVITY**

17.7.2: Inorder traversal of a BST.



Consider the following tree.



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

- 1) What node is printed first?



**Check**

**Show answer**



- 2) Complete the tree traversal after node  
300's left subtree has been printed.

11 25 100 201

**Check****Show answer**

- 3) How many nodes are visited?

**Check****Show answer**

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

- 4) Using left, current, and right, what ordering will print the BST from largest to smallest? Ex: An inorder traversal uses left current right.

**Check****Show answer**

## 17.8 BST height and insertion order

### BST height and insertion order

Recall that a tree's **height** is the maximum edges from the root to any leaf. (Thus, a one-node tree has height 0.)

The *minimum* N-node binary tree height is  $h = \lfloor \log_2 N \rfloor$ , achieved when each level is full except possibly the last. The *maximum* N-node binary tree height is  $N - 1$  (the - 1 is because the root is at height 0).

Searching a BST is fast if the tree's height is near the minimum. Inserting items in random order naturally keeps a BST's height near the minimum. In contrast, inserting items in nearly-sorted order leads to a nearly-maximum tree height.

**PARTICIPATION  
ACTIVITY**

17.8.1: Inserting in random order keeps tree height near the minimum.  
Inserting in sorted order yields the maximum.



### Animation captions:

1. Inserting in random order naturally keeps tree height near the minimum, in this case 3 (minimum: 2)
2. Inserting in sorted order yields the maximum height, in this case 6.
3. If nodes are given beforehand, randomizing the ordering before inserting keeps tree height near minimum.

**PARTICIPATION ACTIVITY**

17.8.2: BST height.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



Draw a BST by hand, inserting nodes one at time, to determine a BST's height.

- 1) A new BST is built by inserting nodes in this order:

6 2 8



What is the tree height? (Remember, the root is at height 0)

**Check****Show answer**

- 2) A new BST is built by inserting nodes in this order:

20 12 23 18 30



What is the tree height?

**Check****Show answer**

- 3) A new BST is built by inserting nodes in this order:

30 23 21 20 18



What is the tree height?

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

**Check****Show answer**

- 4) A new BST is built by inserting nodes in this order:

30 11 23 21 20



What is the tree height?

**Check****Show answer**

- 5) A new BST is built by inserting 255 nodes in sorted order. What is the tree height?

**Check****Show answer**

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

- 6) A new BST is built by inserting 255 nodes in random order. What is the minimum possible tree height?

**Check****Show answer**

## BSTGetHeight algorithm

Given a node representing a BST subtree, the height can be computed as follows:

- If the node is null, return -1.
- Otherwise recursively compute the left and right child subtree heights, and return 1 plus the greater of the 2 child subtrees' heights.

PARTICIPATION  
ACTIVITY

17.8.3: BSTGetHeight algorithm.



### Animation content:

undefined

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

### Animation captions:

1. BSTGetHeight(tree $\rightarrow$ root) is called to get the height of the tree. The height of the root's left child is determined first using a recursive call.
2. BSTGetHeight for node 18 makes a recursive call on node 12. BSTGetHeight on node 12 makes a recursive call on the null left child, which returns -1.

3. Returning to the BSTGetHeight(node 12) call, a recursive call is now made on the right child. Node 14 is a leaf, so both recursive calls return -1.
4. BSTGetHeight(node 14) returns  $1 + \max(-1, -1) = 1 + -1 = 0$ .
5. BSTGetHeight(node 12) has completed 2 recursive calls and returns  $1 + \max(-1, 0) = 1$ . BSTGetHeight(node 18) makes the recursive call on the null right child, which returns -1.
6. A recursive call is made for each node in the tree. BSTGetHeight(tree → root) returns  $1 + \max(2, 1) = 3$ , which is the tree's height.

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

**PARTICIPATION ACTIVITY**

## 17.8.4: BSTGetHeight algorithm.



- 1) BSTGetHeight returns 0 for a tree with a single node.

- True  
 False

- 2) The base case for BSTGetHeight is when the node argument is null.

- True  
 False

- 3) The worst-case time complexity for BSTGetHeight is  $O(\log N)$ , where  $N$  is the number of nodes in the tree.

- True  
 False

- 4) BSTGetHeight would also work if the recursive call on the right child was made before the recursive call on the left child.

- True  
 False



©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

## 17.9 BST parent node pointers

A BST implementation often includes a parent pointer inside each node. A balanced BST, such as an AVL tree or red-black tree, may utilize the parent pointer to traverse up the tree from a particular node

to find a node's parent, grandparent, or siblings. The BST insertion and removal algorithms below insert or remove nodes in a BST with nodes containing parent pointers.

Figure 17.9.1: BSTInsert algorithm for BSTs with nodes containing parent pointers.

```
BSTInsert(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
        }
    }
}
```

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

Figure 17.9.2: BSTReplaceChild algorithm.

```
BSTReplaceChild(parent, currentChild, newChild) {
    if (parent->left != currentChild &&
        parent->right != currentChild)
        return false

    if (parent->left == currentChild)
        parent->left = newChild
    else
        parent->right = newChild

    if (newChild != null)
        newChild->parent = parent
    return true
}
```

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

Figure 17.9.3: BSTRemoveKey and BSTRemoveNode algorithms for BSTs with nodes containing parent pointers.

```

BSTRemoveKey(tree, key) {
    node = BSTSearch(tree, key)
    BSTRemoveNode(tree, node)
}

BSTRemoveNode(tree, node) {
    if (node == null)
        return

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

        // Copy value/data from succNode to node
        node = Copy succNode

        // Recursively remove succNode
        BSTRemoveNode(tree, succNode)
    }

    // 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

        // Make sure the new root, if non-null, has a null parent
        if (tree->root != null)
            tree->root->parent = null
    }

    // Case 3: Internal with left child only
    else if (node->left != null)
        BSTReplaceChild(node->parent, node, node->left)

    // Case 4: Internal with right child only OR leaf
    else
        BSTReplaceChild(node->parent, node, node->right)
}

```

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

#### PARTICIPATION ACTIVITY

17.9.1: BST parent node pointers.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

- 1) **BSTInsert** will not work if the tree's root is null.
  - True
  - False
- 2) **BSTReplaceChild** will not work if the

parent pointer is null.

- True
- False

3) **BSTRemoveKey** will not work if the key is not in the tree.



- True
- False

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

4) **BSTRemoveNode** will not work to remove the last node in a tree.



- True
- False

5) **BSTRemoveKey** uses **BSTRemoveNode**.



- True
- False

6) **BSTRemoveNode** uses **BSTRemoveKey**.



- True
- False

7) **BSTRemoveNode** may use recursion.



- True
- False

8) **BSTRemoveKey** will not properly update parent pointers when a non-root node is being removed.



- True
- False

9) All calls to **BSTRemoveNode** to remove a non-root node will result in a call to **BSTReplaceChild**.



- True
- False

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

# 17.10 BST: Recursion

## BST recursive search algorithm

BST search can be implemented using recursion. A single node and search key are passed as arguments to the recursive search function. Two base cases exist. The first base case is when the node is null, in which case null is returned. If the node is non-null, then the search key is compared to the node's key. The second base case is when the search key equals the node's key, in which case the node is returned. If the search key is less than the node's key, a recursive call is made on the node's left child. If the search key is greater than the node's key, a recursive call is made on the node's right child.

### PARTICIPATION ACTIVITY

17.10.1: BST recursive search algorithm.



### Animation content:

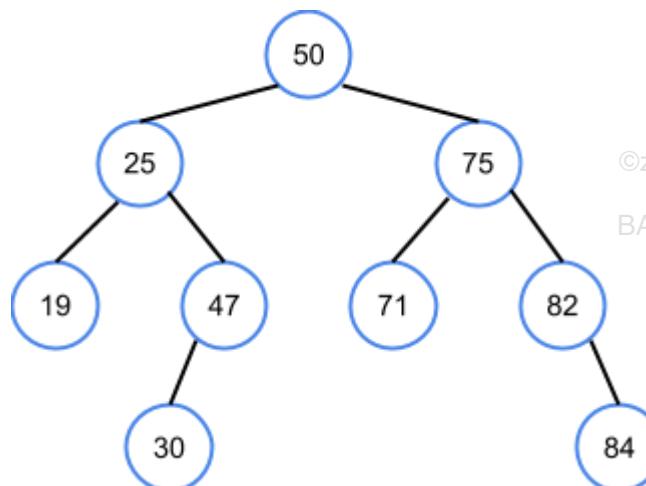
undefined

### Animation captions:

1. A call to `BSTSearch(tree, 40)` calls the `BSTSearchRecursive` function with the tree's root as the node argument.
2. The search key 40 is less than 64, so a recursive call is made on the root node's left child.
3. An additional recursive call searches node 32's right child. The key 40 is found and node 40 is returned.
4. Each function returns the result of a recursive call, so `BSTSearch(tree, 40)` returns node 40.

### PARTICIPATION ACTIVITY

17.10.2: BST recursive search algorithm.



©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



1) How many calls to BSTSearchRecursive are made by calling BSTSearch(tree, 71)?

- 2
- 3
- 4

2) How many calls to BSTSearchRecursive are made by calling BSTSearch(tree, 49)?

- 3
- 4
- 5

3) What is the maximum possible number of calls to BSTSearchRecursive when searching the tree?

- 4
- 5

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021



## BST get parent algorithm

In a BST without parent pointers, a search for a node's parent can be implemented recursively. The algorithm recursively searches for a parent in a way similar to the normal BSTSearch algorithm. But instead of comparing a search key against a candidate node's key, the node is compared against a candidate parent's child pointers.

Figure 17.10.1: BST get parent algorithm.

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

```
BSTGetParent(tree, node) {
    return BSTGetParentRecursive(tree->root, node)
}

BSTGetParentRecursive(subtreeRoot, node) {
    if (subtreeRoot is null)
        return null

    if (subtreeRoot->left == node or
        subtreeRoot->right == node) {
        return subtreeRoot
    }

    if (node->key < subtreeRoot->key) {
        return BSTGetParentRecursive(subtreeRoot->left, node)
    }
    return BSTGetParentRecursive(subtreeRoot->right, node)
}
```

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

**PARTICIPATION ACTIVITY**

## 17.10.3: BST get parent algorithm.



- 1) BSTGetParent returns null when the node argument is the tree's root.

- True
- False



- 2) BSTGetParent always returns a non-null node when searching for a null node.

- True
- False



- 3) The base case for BSTGetParentRecursive is when subtreeRoot is null or is node's parent.

- True
- False

**Recursive BST insertion and removal**

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

BST insertion and removal can also be implemented using recursion. The insertion algorithm uses recursion to traverse down the tree until the insertion location is found. The removal algorithm uses the recursive search functions to find the node and the node's parent, then removes the node from the tree. If the node to remove is an internal node with 2 children, the node's successor is recursively removed.

## Figure 17.10.2: Recursive BST insertion and removal.

```

BSTInsert(tree, node) {
    if (tree->root is null)
        tree->root = node
    else
        BSTInsertRecursive(tree->root, node)
}

BSTInsertRecursive(parent, nodeToInsert) {
    if (nodeToInsert->key < parent->key) {
        if (parent->left is null)
            parent->left = nodeToInsert
        else
            BSTInsertRecursive(parent->left, nodeToInsert)
    }
    else {
        if (parent->right is null)
            parent->right = nodeToInsert
        else
            BSTInsertRecursive(parent->right, nodeToInsert)
    }
}

BSTRemove(tree, key) {
    node = BSTSearch(tree, key)
    parent = BSTGetParent(tree, node)
    BSTRemoveNode(tree, parent, node)
}

BSTRemoveNode(tree, parent, node) {
    if (node == null)
        return false

    // Case 1: Internal node with 2 children
    if (node->left != null && node->right != null) {
        // Find successor and successor's parent
        succNode = node->right
        successorParent = node
        while (succNode->left != null) {
            successorParent = succNode
            succNode = succNode->left
        }

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

        // Recursively remove successor
        BSTRemoveNode(tree, successorParent, succNode)
    }

    // 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
    }

    // Case 3: Internal with left child only
    else if (node->left != null) {
        // Replace node with node's left child
        if (parent->left == node)
            parent->left = node->left
        else
    }
}

```

©zyBooks 04/25/21 07:54 488201  
 xiang zhao  
 BAYLORCSI14301440Spring2021

©zyBooks 04/25/21 07:54 488201  
 xiang zhao  
 BAYLORCSI14301440Spring2021

```

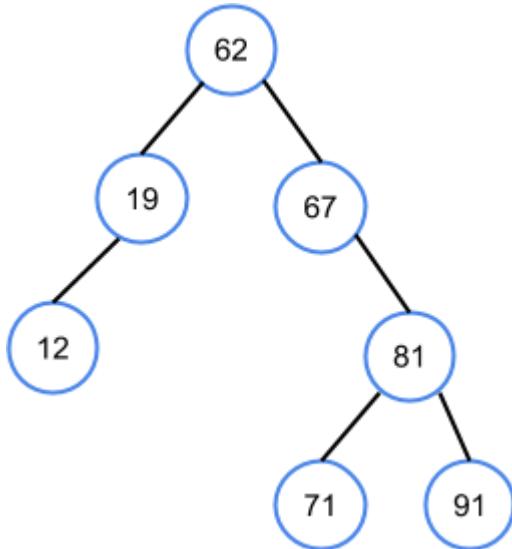
        parent->right = node->left
    }

    // Case 4: Internal with right child only OR leaf
    else {
        // Replace node with node's right child
        if (parent->left == node)
            parent->left = node->right
        else
            parent->right = node->right
    }

    return true
}

```

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

**PARTICIPATION ACTIVITY**
**17.10.4: Recursive BST insertion and removal.**


The following operations are executed on the above tree:

BSTInsert(tree, node 70)  
BSTInsert(tree, node 56)  
BSTRemove(tree, 67)

1) Where is node 70 inserted?



- Node 67's left child
- Node 71's left child
- Node 71's right child

2) How many times is BSTInsertRecursive called when inserting node 56?



- 2
- 3
- 5

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



- 3) How many times is BSTRemoveNode called when removing node 67?

- 1
- 2
- 3

- 4) What is the maximum number of calls to BSTRemoveNode when removing one of the tree's nodes?

- 2
- 4
- 5

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

## 17.11 Tries

### Overview

A **trie** (or **prefix tree**) is a tree representing a set of strings. Each node represents a single character and has at most one child per distinct alphabet character. A **terminal node** is a node that represents a terminating character, which is the end of a string in the trie.

PARTICIPATION ACTIVITY

17.11.1: Trie representing the set of strings: bat, cat, and cats.



### Animation content:

undefined

### Animation captions:

1. The following trie represents a set of two strings. Each string can be determined by traversing the path from the root to a leaf.
2. Suppose "cats" is added to the trie. Adding another child for 'c' would violate the trie requirements.
3. So the existing child for 'c' is reused.
4. Similarly, nodes for 'a' and 't' are reused. Node 't' has a new child added for 's'.
5. Exactly one terminal node exists for each string. Other nodes may be shared by multiple strings.

**PARTICIPATION ACTIVITY**

17.11.2: Trie representing the set of strings: bat, cat, and cats.



Refer to the trie above.

- 1) The terminal nodes can be removed, and instead the last character of a string can be a leaf.

- True  
 False

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

- 2) Adding the string "balance" would create a new branch off the root node.

- True  
 False



- 3) Inserting a string that doesn't already exist in the trie requires allocation of at least 1 new node.

- True  
 False



## Trie insert algorithm

Given a string, a **trie insert** operation creates a path from the root to a terminal node that visits all the string's characters in sequence.

A current node pointer initially points to the root. A loop then iterates through the string's characters. For each character C:

1. A new child node is added only if the current node does not have a child for C.
2. The current node pointer is assigned with the current node's child for C.

After all characters are processed, a terminal node is added and returned.

**PARTICIPATION ACTIVITY**

17.11.3: Trie insert algorithm.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021



### Animation content:

**undefined**

### Animation captions:

1. The trie starts with an empty root node. Adding "APPLE" first adds a new child for 'A' to the root node.
2. New nodes are built for each remaining character in "APPLE".
3. The terminal node is added, completing insertion of "APPLE".
4. When adding "APPLY", the first 4 character nodes are reused.
5. Node L has no child for 'Y', so a new node is added. The terminal node is also added.
6. When adding "APP", only 1 new node is needed, the terminal node.

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

**PARTICIPATION ACTIVITY****17.11.4: Trie insert algorithm.**

Assume a trie is built by executing the following code.

```
trieRoot = new TrieNode()
TrieInsert(trieRoot, "cat")
TrieInsert(trieRoot, "cow")
TrieInsert(trieRoot, "crow")
```

- 1) When inserting "cat", \_\_\_\_ new nodes are created.

- 1
- 3
- 4

- 2) When inserting "crow", \_\_\_\_ new nodes are created.

- 1
- 4
- 5

- 3) If `TrieInsert(trieRoot, "cow")` is called a second time, \_\_\_\_ new nodes are created.

- 0
- 1
- 4

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

## Trie search algorithm

Given a string, a **trie search** operation returns the terminal node corresponding to that string, or null if the string is not in the trie.

**Animation content:****undefined****Animation captions:**

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

1. The search for "PAPAYA" starts at the root and iterates through one node per character. The terminal node is returned, indicating that the string was found.
2. The search for "GRAPE" ends quickly, since the root has no child for 'G'.
3. The search for "PEA" gets to the node for 'A'. No terminal child node exists after 'A', so null is returned.



Refer to the trie above. Assume that to "visit" a node means accessing the node's children in `TrieSearch`.

- 1) `TrieSearch(root, "PINEAPPLE")` returns \_\_\_\_.

- the trie's root node
- the terminal node for "APPLE"
- null

- 2) When searching for "PLUM", \_\_\_\_ visited.

- only the root node is
- the root and the root's 'P' child node are
- all nodes in the root's 'P' child subtree are

- 3) `TrieSearch` will visit at most \_\_\_\_ nodes in a single search.

- 7
- 9
- 41

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

**Trie remove algorithm**

Given a string, a **trie remove** operation removes the string's corresponding terminal node and all non-root ancestors with 0 children.

**PARTICIPATION ACTIVITY**

17.11.7: Trie remove algorithm.


**Animation content:**

undefined

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021

**Animation captions:**

1. TrieRemove is called to remove "BANANA". TrieRemove then calls TrieRemoveRecursive, passing the trie's root, the string "BANANA", and a character index of 0.
2. The root has a child for 'B'. A recursive removal call is made for the child and the next character index.
3. Recursive calls continue until the terminal node's parent is reached.
4. The terminal node is removed from the node's children and true is returned.
5. After returning from each recursive call, child nodes with 0 children are also removed.
6. When removing "APPLE", 4 nodes are removed.
7. Removal operations only remove nodes that are exclusive to the string being removed.

**PARTICIPATION ACTIVITY**

17.11.8: Trie remove algorithm.



Refer to the trie above. Match each operation to the statement that is true when the operation executes. Assume that "BANANA" and "APPLE" have already been removed.

**TrieRemove(root, "CHERRY")  
TrieRemove(root, "PLUM")**

**TrieRemove(root, "AVOCADO")  
TrieRemove(root, "APRICOT")**

**TrieRemove(root, "PEAR")**

**TrieRemove(root, "PAPAYA")**

Remove's the root's child node for 'A'.

©zyBooks 04/25/21 07:54 488201

xiang zhao

BAYLORCSI14301440Spring2021

charIndex is 6 at the moment a terminal node is removed.

Has no effect.

Removes a total of 2 nodes from the trie.

**Reset**

## Trie time complexities

Implementations commonly use a lookup table for a trie node's children, allowing retrieval of a child node from a character in  $O(1)$  time. Therefore, to insert, remove, or search for a string of length  $M$  in a trie takes  $O(M)$  time. The trie's current size does not affect each operation's time complexity.

©zyBooks 04/25/21 07:54 488201  
xiang zhao  
BAYLORCSI14301440Spring2021