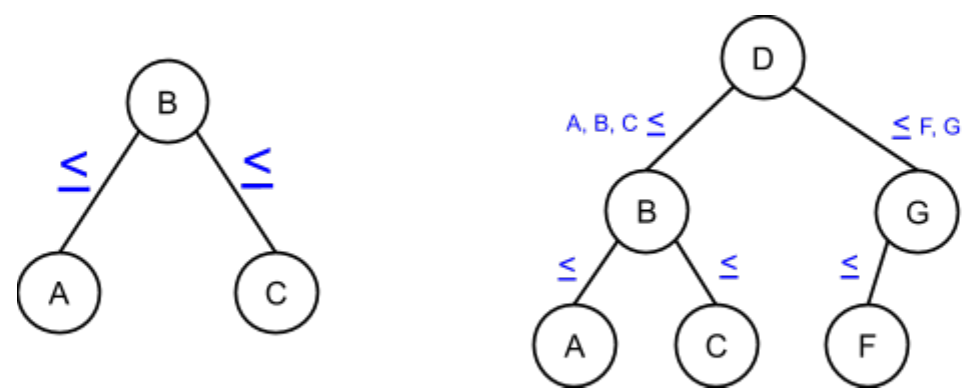
7.1 Binary search trees

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 ≤ the node's key, and the right subtree's keys ≥ the node's key. That property enables fast searching for an item, as will be shown later.

Figure 7.1.1: 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.



PARTICIPATION ACTIVITY | 7.1.1: BST ordering properties.

Animation content:

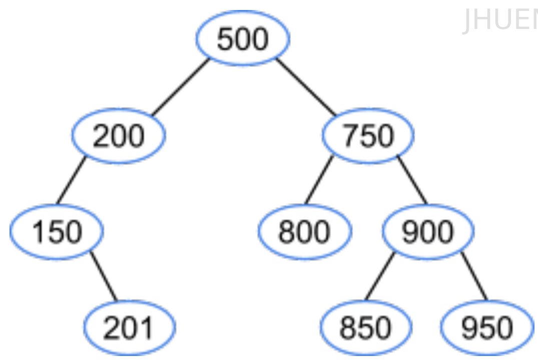
undefined

Animation captions:

1. BST ordering property: Left subtree's keys ≤ node's key, right subtree's keys ≥ 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.

PARTICIPATION ACTIVITY 7.1.2: 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

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)?
   - 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 7.1.2: 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
}
```

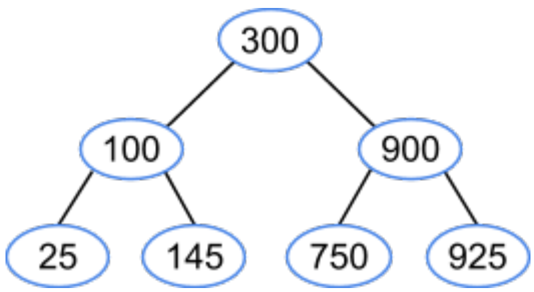

If a child to be visited doesn't exist, the desired node does not exist. With this approach, only a small fraction of nodes need be compared.

PARTICIPATION ACTIVITY | 7.1.3: A BST may yield faster searches than a list.

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 | 7.1.4: Searching a BST.



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

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

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($logN$), 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 7.1.1: 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
...				
15	3	3.9	3	1/2/4/8
16	4	4	4	1/2/4/8/1

PARTICIPATION ACTIVITY 7.1.5: Searching a perfect BST with N nodes requires only O($logN$) comparisons.

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($logN$).

3. Searching a BST may be faster than searching a list.

PARTICIPATION ACTIVITY | 7.1.6: Searching BSTs with N nodes.

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

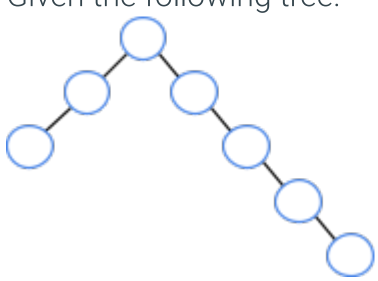
1) Perfect BST with N = 7

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

2) Perfect BST with N = 31

- ○ 31
- ○ 4
- ○ 5

3) Given the following tree.



- ○ 3
- ○ 5

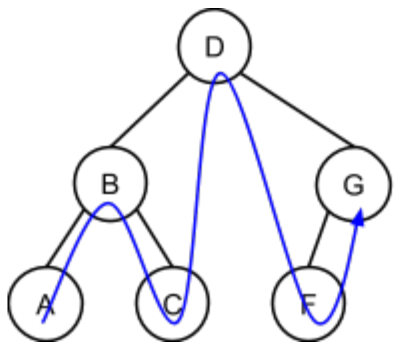
Successors and predecessors

©zyBooks 03/29/21 23:16 926259
NAT FOSTER
JHUEN605202JavaSpring2021

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 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.

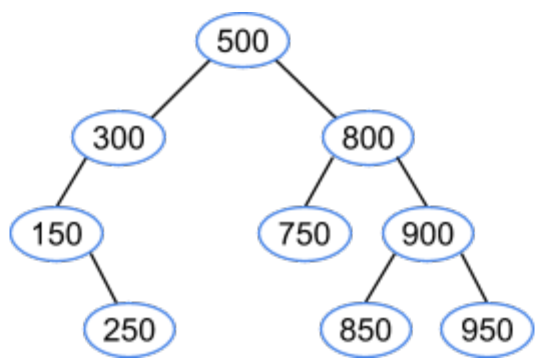
Figure 7.1.3: A BST defines an ordering among nodes.



BST ordering:
A B C D F G

Successor follows in ordering.
Ex: D's successor is F.

PARTICIPATION ACTIVITY 7.1.7: Binary search tree: Defined ordering.



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.

True

False

5) 950's successor is 150.

True

False

6) 950's predecessor is 900.

True

False

CHALLENGE ACTIVITY | 7.1.1: Binary search trees.

7.2 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 | 7.2.1: BST recursive search algorithm.

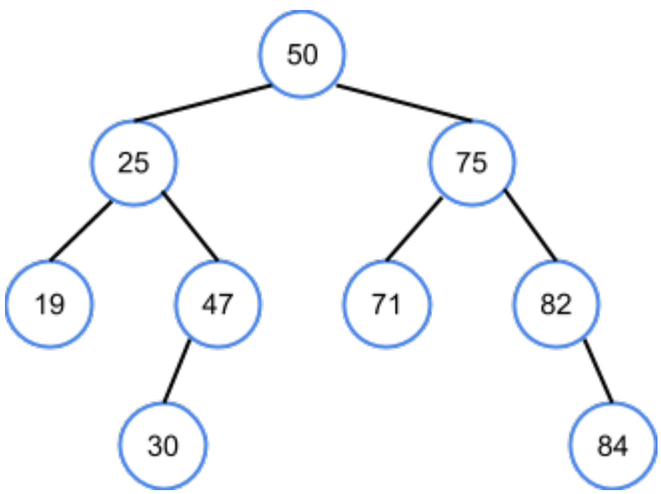
Animation content:

`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 | 7.2.2: BST recursive search algorithm.



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

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 7.2.1: BST get parent algorithm.

```
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)
}
```

PARTICIPATION ACTIVITY | 7.2.3: 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

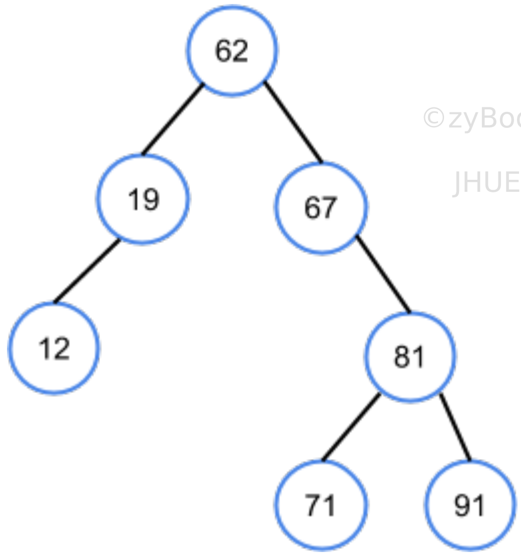
Recursive BST insertion and removal

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 7.2.2: 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
```

PARTICIPATION ACTIVITY | 7.2.4: 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

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

7.3 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 7.3.1: A file system is a hierarchy that can be represented by a tree.

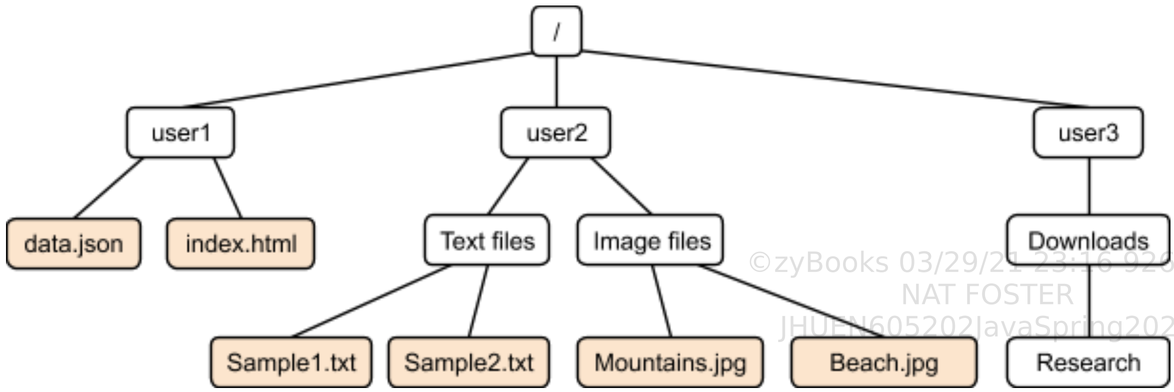
Animation content:

`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 7.3.2: Analyzing a file system tree.



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
- Adding a new directory into the "Research" directory

PARTICIPATION ACTIVITY | 7.3.3: 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

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 | 7.3.4: A BSP tree is used to quickly determine which objects do not need to be rendered.

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 7.3.5: 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

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.

7.4 Minimum spanning tree

This section has been set as optional by your instructor.

Overview

A graph's ***minimum spanning tree*** is a subset of the graph's edges that connect all vertices in the graph together with the minimum sum of edge weights. The graph must be weighted and connected. A ***connected*** graph contains a path between every pair of vertices.

PARTICIPATION ACTIVITY 7.4.1: Using the minimum spanning to minimize total length of power lines connecting cities.

Animation captions:

1. A minimum spanning tree can be used to find the minimal amount of power lines needed to connect cities. Each vertex represents a city. Edges represent roads between cities. The city P has a power plant.
2. Power lines are along roads, such that each city is connected to a powered city. But power lines along every road would be excessive.
3. The minimum spanning tree, shown in red, is the set of edges that connect all cities to power with minimal total power line length.
4. The resulting minimum spanning tree can be viewed as a tree with the power plant city as the root.

PARTICIPATION ACTIVITY 7.4.2: Minimum spanning tree.

1) If no path exists between 2 vertices in a weighted and undirected graph, then no minimum spanning tree exists for the graph.

○ True

○ False

2) A minimum spanning tree is a set of vertices.

○ True

○ False

3) The "minimum" in "minimum spanning tree" refers to the sum of edge weights.

○ True

○ False

4) A minimum spanning tree can only be built for an undirected graph.

○ True

○ False

Kruskal's minimum spanning tree algorithm

Kruskal's minimum spanning tree algorithm determines subset of the graph's edges that connect all vertices in an undirected graph with the minimum sum of edge weights. Kruskal's minimum spanning tree algorithm uses 3 collections:

- An edge list initialized with all edges in the graph.
- A collection of vertex sets that represent the subsets of vertices connected by current set of edges in the minimum spanning tree. Initially, the vertex sets consists of one set for each vertex.
- A set of edges forming the resulting minimum spanning tree.

The algorithm executes while the collection of vertex sets has at least 2 sets and the edge list has at least 1 edge. In each iteration, the edge with the lowest weight is removed from the list of edges. If the removed edge connects two different vertex sets, then the edge is added to the resulting minimum spanning tree, and the two vertex sets are merged.

PARTICIPATION ACTIVITY | 7.4.3: Minimum spanning tree algorithm.

Animation content:

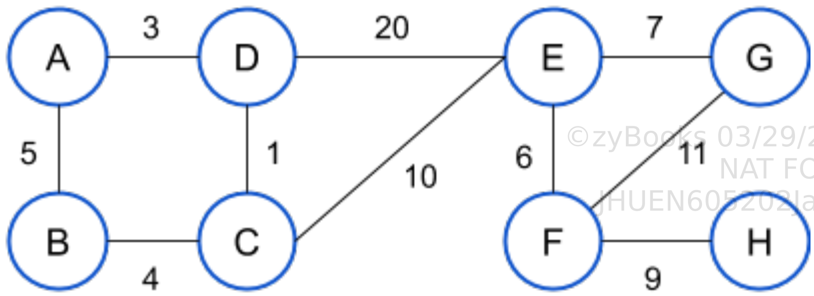
`undefined`

Animation captions:

1. An edge list, a collection of vertex sets, and an empty result set are initialized. The edge list contains all edges from the graph.
2. Edge AD is removed from the edge list and added to resultList, which will contain the edges forming the minimum spanning tree.
3. The next 5 edges connect different vertex sets and are added to the result.
4. Edges AB and CE both connect 2 vertices that are in the same vertex set, and therefore are not added to the result.
5. Edge EF connects the 2 remaining vertex sets.
6. One vertex set remains, so the minimum spanning tree is complete.

PARTICIPATION ACTIVITY | 7.4.4: Minimum spanning tree algorithm.

Consider executing Kruskal's minimum spanning tree algorithm on the following graph:



1) What is the first edge that will be added to the result?

- AD
- AB
- BC
- CD

2) What is the second edge that will be added to the result?

- AD
- AB
- BC
- CD

3) What is the first edge that will NOT be added to the result?

- BC
- AB
- FG
- DE

4) How many edges will be in the resulting minimum spanning tree?

○ 5
○ 7

PARTICIPATION ACTIVITY 7.4.5: Minimum spanning tree - critical thinking.

1) The edge with the lowest weight will always be in the minimum spanning tree.
   - ○ True
   - ○ False

2) The minimum spanning tree may contain all edges from the graph.
   - ○ True
   - ○ False

3) Only 1 minimum spanning tree exists for a graph that has no duplicate edge weights.
   - ○ True
   - ○ False

4) The edges from any minimum spanning tree can be used to create a path that goes through all vertices in the graph without ever encountering the same vertex twice.
   - ○ True
   - ○ False

Algorithm efficiency

Kruskal's minimum spanning tree algorithm's use of the edge list, collection of vertex sets, and resulting edge list results in a space complexity of $O(|E| + |V|)$. If the edge list is sorted at the beginning, then the minimum edge can be removed in constant time within the loop. Combined with a mechanism to map a vertex to the containing vertex set in constant time, the minimum spanning tree algorithm has a runtime complexity of $O(|E| log|E|)$.

7.5 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 7.5.1: 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 | 7.5.2: BST height.

Draw a BST by hand, inserting nodes one at a 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?

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

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 7.5.3: BSTGetHeight algorithm.

Animation content:

```
undefined
```

Animation captions:

1. BSTGetHeight(tree⇢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.

PARTICIPATION ACTIVITY 7.5.4: 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

7.6 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 7.6.1: 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
      }
   }
}
```

Figure 7.6.2: 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
}
```

Figure 7.6.3: 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 03/29/21 23:16 926259
NAT FOSTER
JHUEN605202JavaSpring2021

PARTICIPATION ACTIVITY 7.6.1: BST parent node pointers.

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

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

7.7 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 | 7.7.1: Binary search tree insertions.

Animation content:

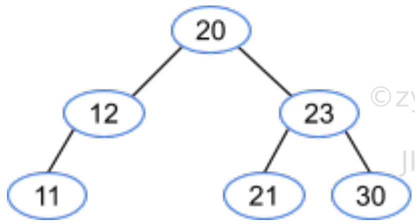
`undefined`

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 7.7.2: BST insert algorithm.

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

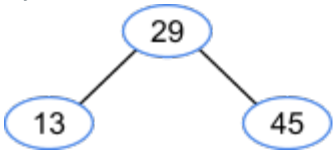
4) Assume a perfect 255-node BST. How many algorithm loop iterations will occur for an insert?

- 8
- 255

PARTICIPATION ACTIVITY 7.7.3: BST insert algorithm decisions.

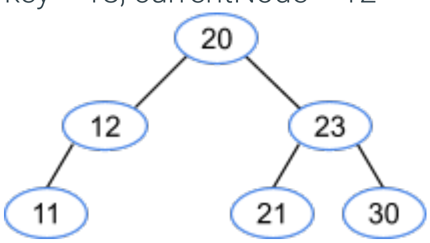
Determine the insertion algorithm's next step given the new node's key and the current node.

1) key = 7, currentNode = 29



- currentNode⇢left = node
- currentNode = currentNode⇢right
- currentNode = currentNode⇢left

2) key = 18, currentNode = 12

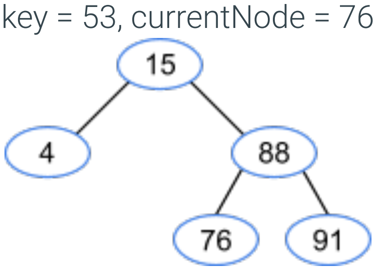


- currentNode⇢left = node
- currentNode⇢right = node
- currentNode = currentNode⇢right

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

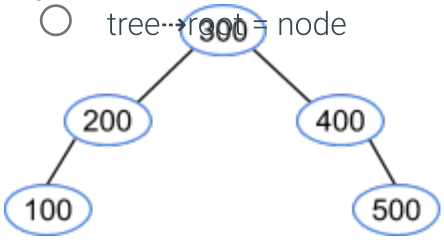
- tree⇢root = node
- currentNode⇢right = node
- currentNode⇢left = node

4) key = 53, currentNode = 76



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

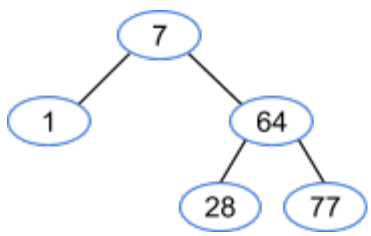
5) key = 600, currentNode = 400



- currentNode→left = node
- currentNode = currentNode→right
- currentNode→right = node

PARTICIPATION ACTIVITY 7.7.4: 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

Check Show answer

BST insert algorithm complexity

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 $O(logN)$ 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 7.7.1: BST insert algorithm.

Exploring further:

- Binary search tree visualization

7.8 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 | 7.8.1: BST search algorithm.

Animation content:

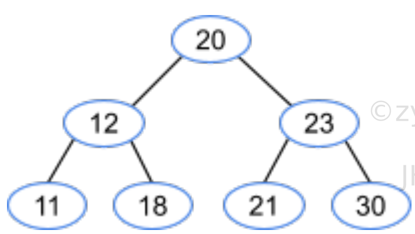
undefined

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 | 7.8.2: BST search algorithm.

Consider the following tree.



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

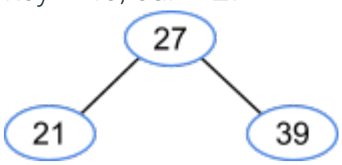
9) Suppose node 23 was instead 21, meaning two 21 nodes exist (which is allowed in a BST). When searching for 21, which node will be returned?

- ○ Leaf
- ○ Internal

PARTICIPATION ACTIVITY | 7.8.3: BST search algorithm decisions.

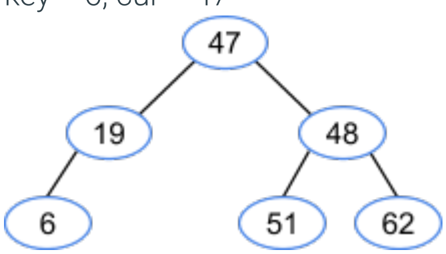
Determine cur's next assignment given the key and current node.

1) key = 40, cur = 27



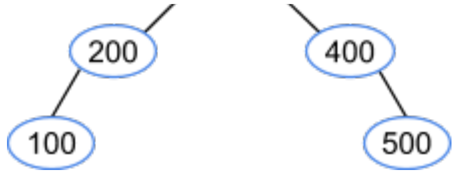
- ○ 27
- ○ 21
- ○ 39

2) key = 6, cur = 47



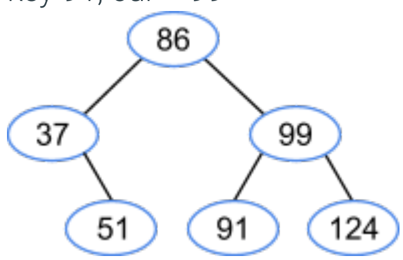
- ○ 6
- ○ 19
- ○ 48

3) key 350, cur = 400



- ○ Search terminates and returns null.
- ○ 400
- ○ 500

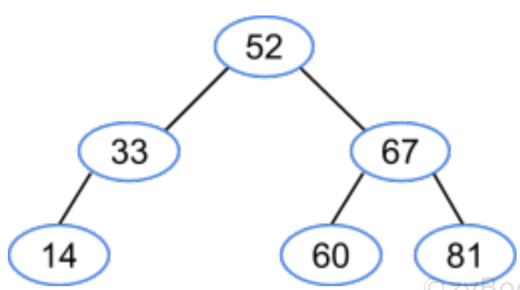
4) key 91, cur = 99



- ○ 86
- ○ 91
- ○ 124

PARTICIPATION ACTIVITY | 7.8.4: 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?

Check Show answer

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

Check Show answer

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

Check Show answer

CHALLENGE ACTIVITY 7.8.1: BST search algorithm.

7.9 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 7.9.1: BST remove: Removing a leaf, or an internal node with a single

ACTIVITY 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.

PARTICIPATION ACTIVITY 7.9.2: 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 7.9.1: BST remove algorithm.

```
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 03/29/21 23:16 926259
NAT FOSTER
JHUEN605202JavaSpring2021

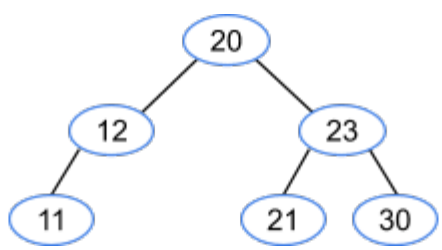
BST remove algorithm complexity

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 $O(logN)$ 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 7.9.3: 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($logN$)
   - O(N)

CHALLENGE ACTIVITY 7.9.1: BST remove algorithm.

7.10 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 7.10.1: BST inorder traversal algorithm.

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

  BSTPrintInorder(node⇢left)
  Print node
  BSTPrintInorder(node⇢right)
}
```

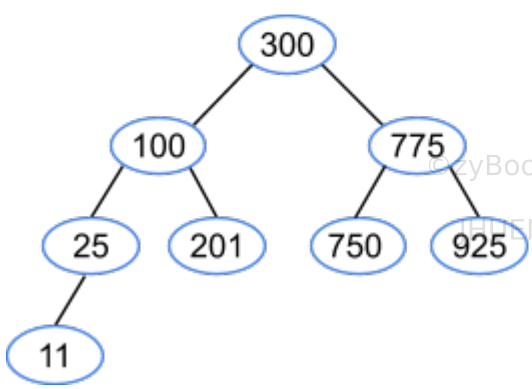

PARTICIPATION ACTIVITY | 7.10.1: BST inorder print algorithm.

Animation captions:

1. An inorder traversals 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 | 7.10.2: Inorder traversal of a BST.

Consider the following tree.



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

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

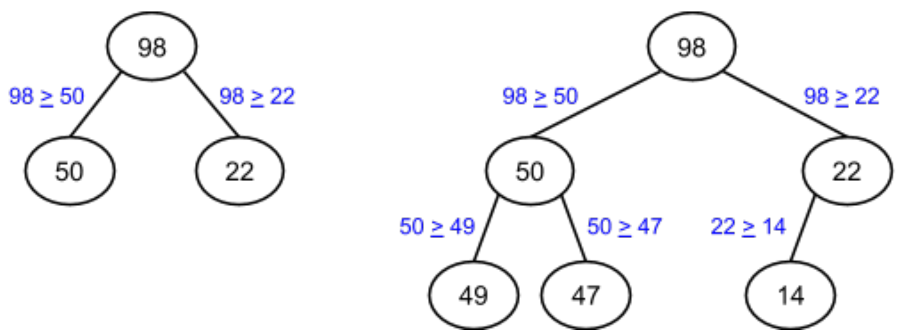
7.11 Heaps

Heap concept

Some applications require fast access to and removal of the maximum item in a changing set of items. For example, a computer may execute jobs one at a time; upon finishing a job, the computer executes the pending job having maximum priority. Ex: Four pending jobs have priorities 22, 14, 98, and 50; the computer should execute 98, then 50, then 22, and finally 14. New jobs may arrive at any time.

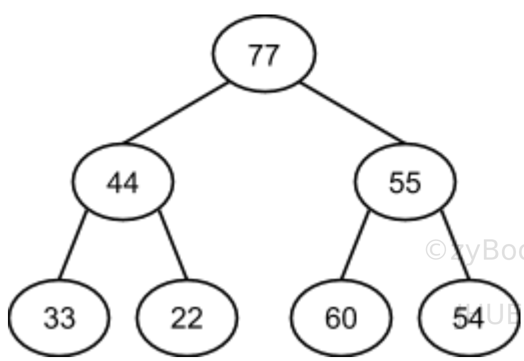
Maintaining jobs in fully-sorted order requires more operations than necessary, since only the maximum item is needed. A ***max-heap*** is a complete binary tree that maintains the simple property that a node's key is greater than or equal to the node's childrens' keys. (Actually, a max-heap may be any tree, but is commonly a binary tree). Because x ≥ y and y ≥ z implies x ≥ z, the property results in a node's key being greater than or equal to all the node's descendants' keys. Therefore, *a max-heap's root always has the maximum key in the entire tree.*

Figure 7.11.1: Max-heap property: A node's key is greater than or equal to the node's childrens' keys.



PARTICIPATION ACTIVITY | 7.11.1: Max-heap property.

Consider this binary tree:



1) 33 violates the max-heap property due to being greater than 22.

- True
- False

2) 54 violates the max-heap property due to being greater than 44.

- True
- False

3) 60 violates the max-heap property due to being greater than 55.

- True
- False

4) A max-heap's root must have the maximum key.

- True
- False

Max-heap insert and remove operations

An ***insert*** into a max-heap starts by inserting the node in the tree's last level, and then swapping the node with its parent until no max-heap property violation occurs. Inserts fill a level (left-to-right) before adding another level, so the tree's height is always the minimum possible. The upward movement of a node in a max-heap is called ***percolating***.

A ***remove*** from a max-heap is always a removal of the root, and is done by replacing the root with the last level's last node, and swapping that node with its greatest child until no max-heap property violation occurs. Because upon completion that node will occupy another node's location (which was swapped upwards), the tree height remains the minimum possible.

PARTICIPATION ACTIVITY 7.11.2: Max-heap insert and remove operations.

Animation captions:

1. This tree is a max-heap. A new node gets initially inserted in the last level...
2. ...and then percolate node up until the max-heap property isn't violated.
3. Removing a node (always the root): Replace with last node, then percolate node down.

PARTICIPATION ACTIVITY | 7.11.3: Max-heap inserts and deletes.

1) Given N nodes, what is the height of a max-heap?

- $\lfloor logN \rfloor$
- N
- Depends on the keys

2) Given a max-heap with levels 0, 1, 2, and 3, with the last level not full, after inserting a new node, what is the maximum possible swaps needed?

- 1
- 2
- 3

3) Given a max-heap with N nodes, what is the worst-case complexity of an insert, assuming an insert is dominated by the swaps?

- O(N)
- O($logN$)

4) Given a max-heap with N nodes, what is the complexity for removing the root?

- O(N)
- O($logN$)

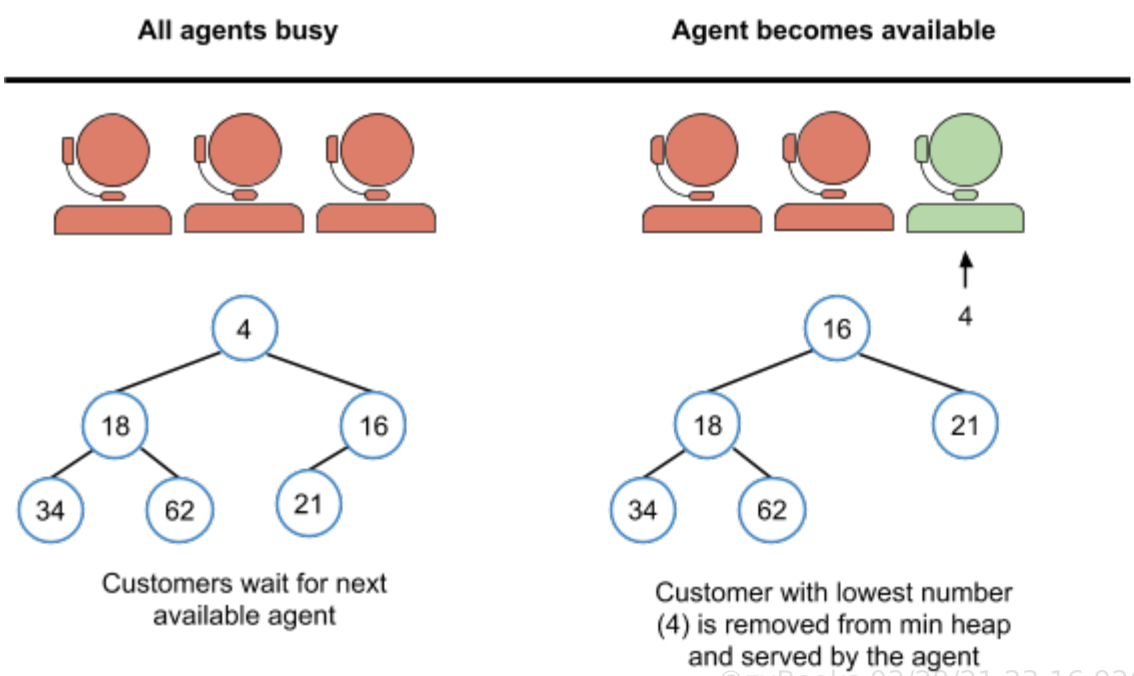
Min-heap

A ***min-heap*** is similar to a max-heap, but a node's key is less than or equal to its children's keys.

Example 7.11.1: Online tech support waiting lines commonly use min heaps.

Many companies have online technical support that lets a customer chat with a support agent. If the number of customers seeking support is greater than the number of available agents, customers enter a virtual waiting line. Each customer has a priority that determines their place in line. The customer with the highest priority is served by the next available agent.

A min heap is commonly used to manage prioritized queues of customers awaiting support. Customers that entered the line earlier and/or have a more urgent issue get assigned a lower number, which corresponds to a higher priority. When an agent becomes available, the customer with the lowest number is removed from the heap and served by the agent.



PARTICIPATION ACTIVITY 7.11.4: Min heaps and customer support.

1) A customer with a higher priority has a lower numerical value in the

min heap.

- True
- False

2) If 2,000 customers are waiting for technical support, removing a customer from the min heap requires about 2,000 operations.

- True
- False

CHALLENGE ACTIVITY 7.11.1: Heaps.

7.12 Tries

This section has been set as optional by your instructor.

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 7.12.1: 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 7.12.2: 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

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 | 7.12.3: Trie insert algorithm.

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.

PARTICIPATION ACTIVITY | 7.12.4: 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

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.

PARTICIPATION ACTIVITY 7.12.5: Trie search algorithm.

Animation content:

`undefined`

Animation captions:

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.

PARTICIPATION ACTIVITY 7.12.6: Trie search algorithm.

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

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 | 7.12.7: Trie remove algorithm.

Animation content:

`undefined`

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 | 7.12.8: 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, "PAPAYA")` **`TrieRemove(root, "PEAR")`**

`TrieRemove(root, "AVOCADO")` **`TrieRemove(root, "CHERRY")`**
`TrieRemove(root, "APRICOT")` **`TrieRemove(root, "PLUM")`**

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

`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.

7.13 Treaps

This section has been set as optional by your instructor.

Treap basics

A BST built from inserts of N nodes having random-ordered keys stays well-balanced and thus has near-minimum height, meaning searches, inserts, and deletes are O($logN$). Because insertion order may not be controllable, a data structure that somehow randomizes BST insertions is desirable. A ***treap*** uses a main key that maintains a binary search tree ordering property, and a secondary key generated randomly (often called "priority") during insertions that maintains a heap property. The combination usually keeps the tree balanced. The word "treap" is a mix of tree and heap. This section assumes the heap is a max-heap. Algorithms for basic treap operations include:

- A treap ***search*** is the same as a BST search using the main key, since the treap is a BST.
- A treap ***insert*** initially inserts a node as in a BST using the main key, then assigns a random priority to the node, and percolates the node up until the heap property is not violated. In a heap, a node is moved up via a swap with the node's parent. In a treap, a node is moved up via a *rotation at the parent*. Unlike a swap, a rotation maintains the BST property.
- A treap ***delete*** can be done by setting the node's priority such that the node should be a leaf (-∞ for a max-heap), percolating the node down using rotations until the node is a

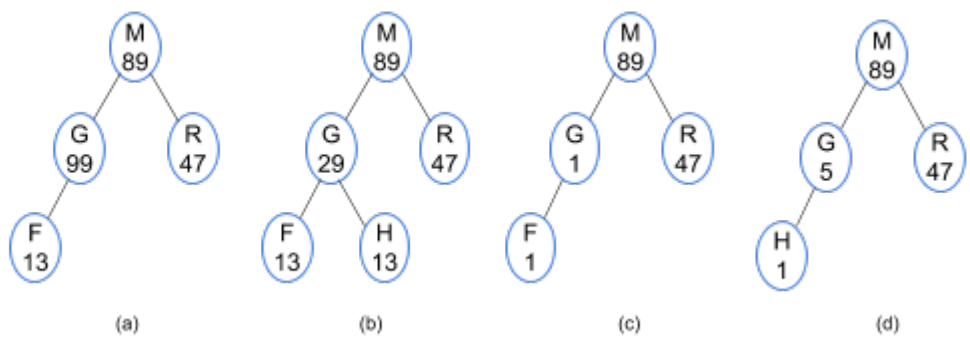
leaf, and then removing the node.

PARTICIPATION ACTIVITY 7.13.1: Treap insert: First insert as a BST, then randomly assign a priority and use rotations to percolate node up to maintain heap.

Animation captions:

1. The keys maintain a BST, the priorities a heap. Insert B as a BST...
2. Assign random priority (70). Rotate (which keep a BST) the node up until the priorities maintain a heap: 20 not > 70: Rotate. 47 not > 70: Rotate. 80 > 70: Done.

PARTICIPATION ACTIVITY 7.13.2: Recognizing treaps.



1) (a)

- Treap
- Not a treap

2) (b)

- Treap
- Not a treap

3) (c)

- Treap
- Not a treap

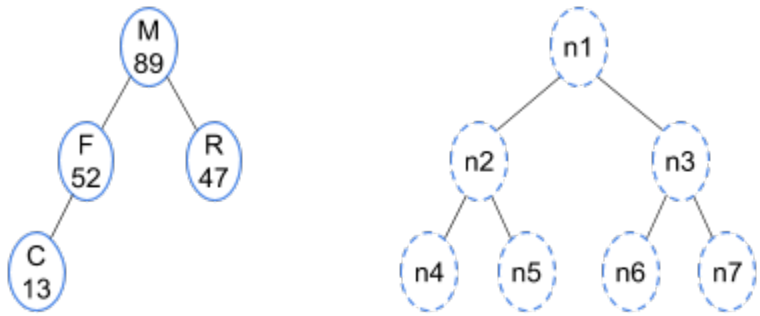
4) (d)

◯ Treap

◯ Not a treap

PARTICIPATION ACTIVITY | 7.13.3: Treap insert.

When performing an insert, indicate each node's new location using the template tree's labels (n1...n7).



1) Where will a new node H first be inserted?

Check Show answer

2) H is assigned a random priority of 20. To where does H percolate?

Check Show answer

3) Where will a new node P first be inserted?

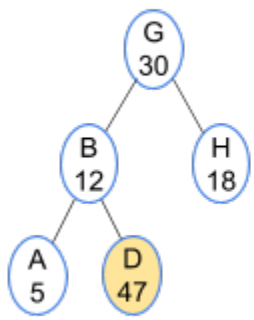
Check Show answer

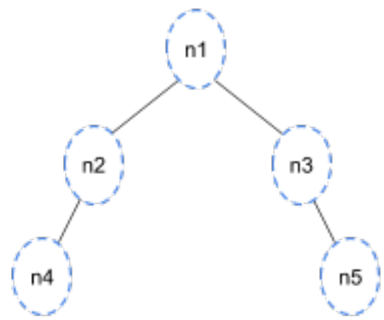
4) P is assigned a random priority of 65. To where does P percolate?

Check Show answer

PARTICIPATION ACTIVITY | 7.13.4: Treap insert.

Node D was just inserted, and assigned a random priority of 47. Rotations are needed to not violate the heap property. Match the node value to the corresponding location in the tree template on the right after the rotations are completed.





G, 30 **H, 18** **B, 12** **A, 5** **D, 47**

n1

n2

n3

n4

n5

Reset

Treap delete

A treap delete could be done by first doing a BST delete (copying the successor to the node-to-delete, then deleting the original successor), followed by percolating the node down until the heap property is not violated. However, a simpler approach just sets the node-to-delete's priority to -∞ (for a max-heap), percolates the node down until a leaf, and removes the node. Percolating the node down uses rotations, not swaps, to maintain the BST property. Also, the node is rotated in the direction of the lower-priority child, so that the node rotated up has a higher priority than that child, to keep the heap property.

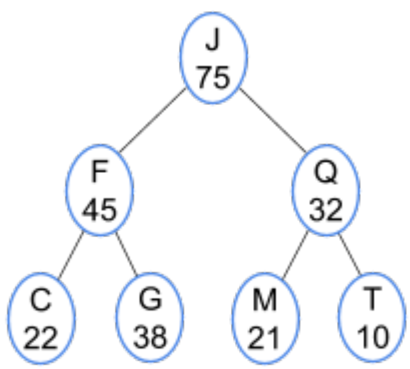
PARTICIPATION ACTIVITY | 7.13.5: Treap delete: Set priority such that node must become a leaf, then percolate down using rotations.

Animation captions:

1. Node F is to be deleted. First set F's priority to -∞.
2. Rotate (to keep a BST) until the node becomes a leaf node. 29 > 13: Rotate right.
3. Rotate until node becomes a leaf node. Rotate left (the only option).
4. Remove leaf node.

PARTICIPATION ACTIVITY | 7.13.6: Treap delete algorithm.

Each question starts from the original tree. Use this text notation for the tree: (J (F (C, G), Q (M, T))). A - means the child does not exist.



1) What is the tree after removing G?

- (J (F (C, -), Q (M, T)))
- (J (C (-, F), Q (M, T)))

2) What is the tree after removing Q?

- (J (F (C, G), M(-, T))
- (J (F (C, G), T(M, -))

PARTICIPATION ACTIVITY | 7.13.7: Treaps.

1) A treap's nodes have random main keys.

- True
- False

2) A treap's nodes have random priorities.

- True
- False

3) Suppose a treap is built by inserting nodes with main keys in this order: A, B, C, D, E, F, G. The treap will have 7 levels, with each level having one node with a right child.

- True
- False

7.14 Sorting: Introduction

Sorting is the process of converting a list of elements into ascending (or descending) order. For example, given a list of numbers (17, 3, 44, 6, 9), the list after sorting is (3, 6, 9, 17, 44). You may have carried out sorting when arranging papers in alphabetical order, or arranging envelopes to have ascending zip codes (as required for bulk mailings).

The challenge of sorting is that a program can't "see" the entire list to know where to move an element. Instead, a program is limited to simpler steps, typically observing or swapping just two elements at a time. So sorting just by swapping values is an important part of sorting algorithms.

PARTICIPATION ACTIVITY | 7.14.1: Sort by swapping tool.

Sort the numbers from smallest on left to largest on right. Select two numbers then click "Swap values".

PARTICIPATION ACTIVITY | 7.14.2: Sorted elements.

1) The list is sorted into ascending order:
(3, 9, 44, 18, 76)

- True
- False

2) The list is sorted into descending order:
(20, 15, 10, 5, 0)

- True
- False

3) The list is sorted into descending order:
(99.87, 99.02, 67.93, 44.10)

- True
- False

4) The list is sorted into descending order:
(F, D, C, B, A)

- True
- False

5) The list is sorted into ascending order:
(chopsticks, forks, knives, spork)

- True
- False

6) The list is sorted into ascending order:
(great, greater, greatest)

- True
- False

7.15 Bubble sort

Bubble sort is a sorting algorithm that iterates through a list, comparing and swapping adjacent elements if the second element is less than the first element. Bubble sort uses nested loops. Given a list with N elements, the outer i-loop iterates N - 1 times. Each iteration moves the i^{th} largest element into sorted position. The inner j-loop iterates through all adjacent pairs, comparing and swapping adjacent elements as needed, except for the last i pairs that are already in the correct position,.

Because of the nested loops, bubble sort has a runtime of $O(N^2)$. Bubble sort is often considered impractical for real-world use because many faster sorting algorithms exist.

Figure 7.15.1: Bubble sort algorithm.

```
BubbleSort(numbers, numbersSize) {
   for (i = 0; i < numbersSize - 1; i++) {
      for (j = 0; j < numbersSize - i - 1; j++) {
         if (numbers[j] > numbers[j+1]) {
            temp = numbers[j]
            numbers[j] = numbers[j + 1]
            numbers[j + 1] = temp
         }
      }
   }
}
```


PARTICIPATION ACTIVITY 7.15.1: Bubble sort.

1) Bubble sort uses a single loop to sort the list.

- True
- False

2) Bubble sort only swaps adjacent elements.

- True
- False

3) Bubble sort's best and worst runtime complexity is O(N^2).

- True
- False

7.16 Quicksort

Quicksort

Quicksort is a sorting algorithm that repeatedly partitions the input into low and high parts (each part unsorted), and then recursively sorts each of those parts. To partition the input, quicksort chooses a pivot to divide the data into low and high parts. The ***pivot*** can be any value within the array being sorted, commonly the value of the middle array element. Ex: For the list (4, 34, 10, 25, 1), the middle element is located at index 2 (the middle of indices [0, 4]) and has a value of 10.

Once the pivot is chosen, the quicksort algorithm divides the array into two parts, referred to as the low partition and the high partition. All values in the low partition are less than or equal to the pivot value. All values in the high partition are greater than or equal to the pivot value. The values in each partition are not necessarily sorted. Ex: Partitioning (4, 34, 10, 25, 1) with a pivot value of 10 results in a low partition of (4, 1, 10) and a high partition of (25, 34). Values equal to the pivot may appear in either or both of the partitions.

PARTICIPATION ACTIVITY | 7.16.1: Quicksort partitions data into a low partition with values ≤

pivot and a high partition with values ≥ pivot.

Animation content:

undefined

Animation captions:

1. The pivot value is the value of the middle element.
2. lowIndex is incremented until a value greater than the pivot is found.
3. highIndex is decremented until a value less than the pivot is found.
4. Elements at indices lowIndex and highIndex are swapped, moving those elements to the correct partitions.
5. The partition process repeats until indices lowIndex and highIndex reach or pass each other, indicating all elements have been partitioned.
6. Once partitioned, the algorithm returns highIndex, which is the highest index of the low partition. The partitions are not yet sorted.

Partitioning algorithm

The partitioning algorithm uses two index variables lowIndex and highIndex, initialized to the left and right sides of the current elements being sorted. As long as the value at index lowIndex is less than the pivot value, the algorithm increments lowIndex, because the element should remain in the low partition. Likewise, as long as the value at index highIndex is greater than the pivot value, the algorithm decrements highIndex, because the element should remain in the high partition. Then, if lowIndex >= highIndex, all elements have been partitioned, and the partitioning algorithm returns highIndex, which is the index of the last element in the low partition. Otherwise, the elements at indices lowIndex and highIndex are swapped to move those elements to the correct partitions. The algorithm then increments lowIndex, decrements highIndex, and repeats.

PARTICIPATION ACTIVITY | 7.16.2: Quicksort pivot location and value.

Determine the midpoint and pivot values.

1) numbers = (1, 2, 3, 4, 5), lowIndex = 0, highIndex = 4

`midpoint =`

Check Show answer

2) (1, 2, 3, 4, 5), lowIndex = 0, highIndex = 4

`pivot =`

Check Show answer

3) numbers = (200, 11, 38, 9), lowIndex = 0, highIndex = 3

`midpoint =`

Check Show answer

4) numbers = (200, 11, 38, 9), lowIndex = 0, highIndex = 3

`pivot =`

Check Show answer

5) numbers = (55, 7, 81, 26, 0, 34, 68, 125), lowIndex = 3, highIndex = 7

`midpoint =`

Check Show answer

6) numbers = (55, 7, 81, 26, 0, 34, 68, 125), lowIndex = 3, highIndex = 7

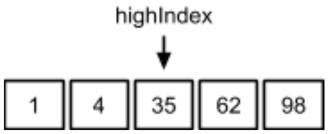
`pivot =`

Check Show answer

PARTICIPATION ACTIVITY 7.16.3: Low and high partitions.

Determine if the low and high partitions are correct given highIndex and pivot.

1) pivot = 35

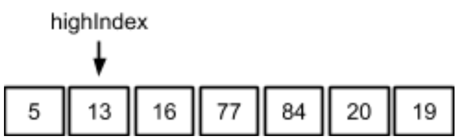


- Correct
- Incorrect

2) pivot = 65

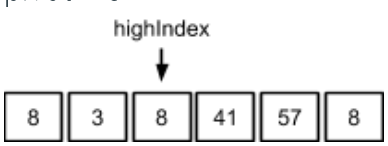
- Correct
- Incorrect

3) pivot = 5



- Correct
- Incorrect

4) pivot = 8



- Correct
- Incorrect

Recursively sorting partitions

Once partitioned, each partition needs to be sorted. Quicksort is typically implemented as a recursive algorithm using calls to quicksort to sort the low and high partitions. This recursive sorting process continues until a partition has one or zero elements, and thus is already sorted.

PARTICIPATION ACTIVITY | 7.16.4: Quicksort.

Animation content:

`undefined`

Animation captions:

1. The list from low index 0 to high index 4 has more than 1 element, so Partition is called.
2. Quicksort is called recursively to sort the low and high partitions.
3. The low partition has more than one element. Partition is called for the low partition, followed by recursive calls to Quicksort.
4. Each partition that has one element is already sorted.
5. The high partition has more than one element and thus is partitioned and recursively sorted.
6. The low partition with two elements is partitioned and recursively sorted.
7. Each remaining partition with only one element is already sorted.
8. All elements are sorted.

Below is the recursive quicksort algorithm, including quicksort's key component, the partitioning function.

Figure 7.16.1: Quicksort algorithm.

```
Partition(numbers, lowIndex, highIndex) {
   // Pick middle element as pivot
   midpoint = lowIndex + (highIndex - lowIndex) / 2
   pivot = numbers[midpoint]

   done = false
   while (!done) {
      // Increment lowIndex while numbers[lowIndex] < pivot
      while (numbers[lowIndex] < pivot) {
         lowIndex += 1
      }

      // Decrement highIndex while pivot < numbers[highIndex]
      while (pivot < numbers[highIndex]) {
         highIndex -= 1
      }

      // If zero or one elements remain, then all numbers are
      // partitioned. Return highIndex.
      if (lowIndex >= highIndex) {
         done = true
      }
      else {
         // Swap numbers[lowIndex] and numbers[highIndex]
         temp = numbers[lowIndex]
         numbers[lowIndex] = numbers[highIndex]
         numbers[highIndex] = temp

         // Update lowIndex and highIndex
         lowIndex += 1
         highIndex -= 1
      }
   }

   return highIndex
}

Quicksort(numbers, lowIndex, highIndex) {
   // Base case: If the partition size is 1 or zero
   // elements, then the partition is already sorted
   if (lowIndex >= highIndex) {
      return
   }

   // Partition the data within the array. Value lowEndIndex
   // returned from partitioning is the index of the low
   // partition's last element.
   lowEndIndex = Partition(numbers, lowIndex, highIndex)

   // Recursively sort low partition (lowIndex to lowEndIndex)
   // and high partition (lowEndIndex + 1 to highIndex)
   Quicksort(numbers, lowIndex, lowEndIndex)
   Quicksort(numbers, lowEndIndex + 1, highIndex)
}

main() {
```


©zyBooks 03/29/21 23:16 926259
NAT FOSTER
JHUEN605202JavaSpring2021

Quicksort activity

The following activity helps build intuition as to how partitioning a list into two unsorted parts, one part <= a pivot value and the other part >= a pivot value, and then recursively sorting each part, ultimately leads to a sorted list.

PARTICIPATION ACTIVITY | 7.16.5: Quicksort tool.

Select all values in the current window that are less than the pivot for the left part, then press "Partition". If a value equals pivot, you can choose which part, but each part must contain at least one number. Yellow means current window. Green means sorted.

Quicksort runtime

The quicksort algorithm's runtime is typically O(N log N). Quicksort has several partitioning levels, the first level dividing the input into 2 parts, the second into 4 parts, the third into 8 parts, etc. At each level, the algorithm does at most N comparisons moving the lowIndex and highIndex indices. If the pivot yields two equal-sized parts, then there will be log N levels, requiring the N * log N comparisons.

PARTICIPATION ACTIVITY | 7.16.6: Quicksort runtime.

Assume quicksort always chooses a pivot that divides the elements into two equal parts.

1) How many partitioning levels are required for a list of 8 elements?

Check Show answer

2) How many partitioning levels are required for a list of 1024 elements?

Check Show answer

3) How many total comparisons are required to sort a list of 1024 elements?

Check Show answer

Worst case runtime

For typical unsorted data, such equal partitioning occurs. However, partitioning may yield unequally sized parts in some cases. If the pivot selected for partitioning is the smallest or largest element, one partition will have just 1 element, and the other partition will have all other elements. If this unequal partitioning happens at every level, there will be N - 1 levels, yielding N + N-1 + N-2 + ... + 2 + 1 = $(N + 1) \cdot (N/2)$, which is O(N^2). So the worst case runtime for the quicksort algorithm is O(N^2). Fortunately, this worst case runtime rarely occurs.

PARTICIPATION ACTIVITY | 7.16.7: Worst case quicksort runtime.

Assume quicksort always chooses the smallest element as the pivot.

1) Given numbers = (7, 4, 2, 25, 19), lowIndex = 0, and highIndex = 4, what are the contents of the low partition? Type answer as: 1, 2, 3

Check Show answer

2) How many partitioning levels are required for a list of 5 elements?

3) How many partitioning levels are required for a list of 1024 elements?

Check Show answer

4) How many total calls to the Quicksort() function are made to sort a list of 1024 elements?

Check Show answer

CHALLENGE ACTIVITY 7.16.1: Quicksort.