Binary Search Trees

Saikrishna Arcot
M. Hudachek-Buswell

July 18, 2020

Idea of Binary Search Trees

- An array or a list can be used to store data items in ascending or descending order.

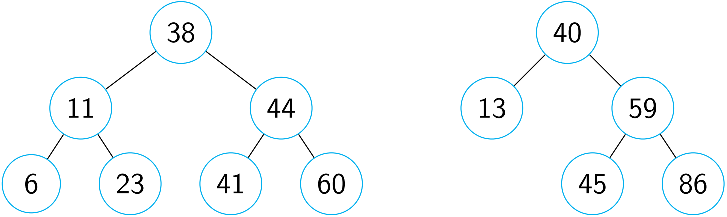
Idea of Binary Search Trees

- An array or a list can be used to store data items in ascending or descending order.
- Binary search can then be used to search for a particular data item in $O(\log n)$ time.

Idea of Binary Search Trees

- An array or a list can be used to store data items in ascending or descending order.
- Binary search can then be used to search for a particular data item in $O(\log n)$ time.
- The idea behind binary search (look at the item in the middle of the list, and based on its value, look at either the left half of the list or the right half of the list) is then turned into a data structure known as a binary search tree (BST).

Example BSTs



Subtree Update Method

- When adding to or removing from a BST (and possibly other tree-like structures), the class representing a single node in the tree may not have a reference to the parent node. This makes updating the left/right child of the parent node a little harder.

Subtree Update Method

- When adding to or removing from a BST (and possibly other tree-like structures), the class representing a single node in the tree may not have a reference to the parent node. This makes updating the left/right child of the parent node a little harder.
- One solution to this is to always save a reference to the parent node, or "work from the parent node". However, this will make the method long and error-prone.

Subtree Update Method

- When adding to or removing from a BST (and possibly other tree-like structures), the class representing a single node in the tree may not have a reference to the parent node. This makes updating the left/right child of the parent node a little harder.
- One solution to this is to always save a reference to the parent node, or "work from the parent node". However, this will make the method long and error-prone.
- A better solution is to always return what the new left/right child of the parent node should be.

Subtree Update Method

- When making a recursive call to your left/right child, pretend that you have some subtree there (or a blob, if you prefer).

Subtree Update Method

- When making a recursive call to your left/right child, pretend that you have some subtree there (or a blob, if you prefer).
- After the recursive call, the structure of the subtree might change, and you need to get the final subtree.

Subtree Update Method

- When making a recursive call to your left/right child, pretend that you have some subtree there (or a blob, if you prefer).
- After the recursive call, the structure of the subtree might change, and you need to get the final subtree.
- The idea is that you assume that the recursive call will always return what the new subtree should be, the new left/right child should be.

Subtree Update Method

- When making a recursive call to your left/right child, pretend that you have some subtree there (or a blob, if you prefer).
- After the recursive call, the structure of the subtree might change, and you need to get the final subtree.
- The idea is that you assume that the recursive call will always return what the new subtree should be, the new left/right child should be.
- When you are actually returning the node in the method, make sure you return the new left/right child of the parent node.

Searching

- Start at the root node.

Searching

- Start at the root node.
- Check the data in the node.

Searching

- Start at the root node.
- Check the data in the node.
  - If the data you're looking for is less than the data in the node, go to the left child.

Searching

- Start at the root node.
- Check the data in the node.
  - If the data you're looking for is less than the data in the node, go to the left child.
  - If the data you're looking for is equal to the data in the node, then you've found the data.

Searching

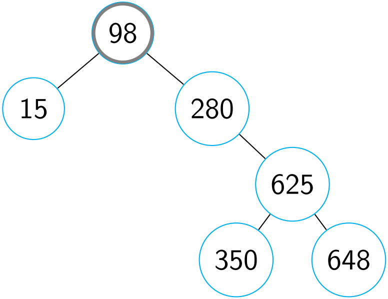
- Start at the root node.
- Check the data in the node.
  - If the data you're looking for is less than the data in the node, go to the left child.
  - If the data you're looking for is equal to the data in the node, then you've found the data.
  - If the data you're looking for is greater than the data in the node, go to the right.

Searching

- Start at the root node.
- Check the data in the node.
  - If the data you're looking for is less than the data in the node, go to the left child.
  - If the data you're looking for is equal to the data in the node, then you've found the data.
  - If the data you're looking for is greater than the data in the node, go to the right.
- Repeat the previous step, until you find the node in the tree or go off of the tree, in which case the data you're looking for isn't in the tree.

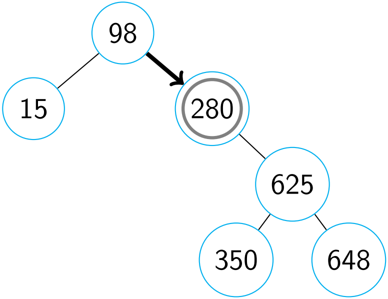
Searching

For example, if you were searching for 625 (a gray circle represents the node being looked at):



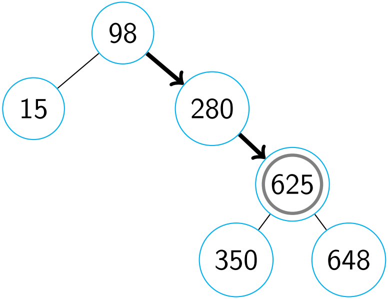
Searching

For example, if you were searching for 625 (a gray circle represents the node being looked at):



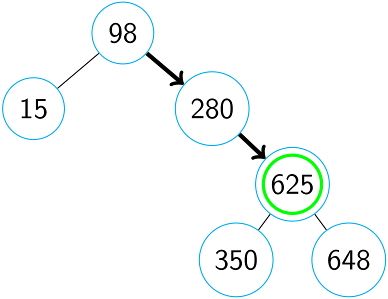
Searching

For example, if you were searching for 625 (a gray circle represents the node being looked at):



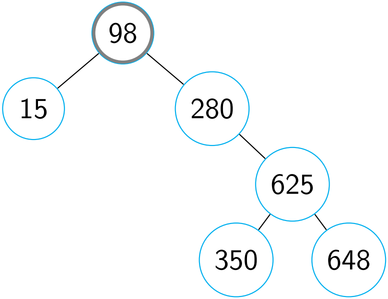
Searching

For example, if you were searching for 625 (a gray circle represents the node being looked at):



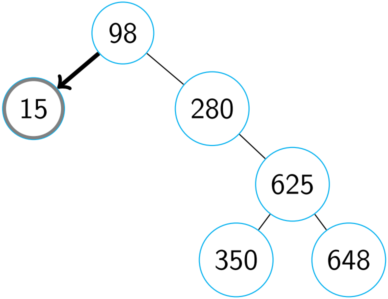
Searching

For example, if you were searching for 40 (a gray circle represents the node being looked at):



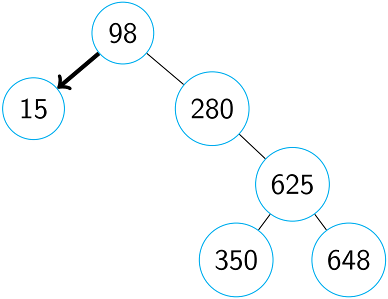
Searching

For example, if you were searching for 40 (a gray circle represents the node being looked at):



Searching

For example, if you were searching for 40 (a gray circle represents the node being looked at):



40 is not in the tree.

Searching

```
procedure SEARCH(data, node)
    if node is not valid then
        return FALSE
    else
        if data = node.data then
            return TRUE
        else if data < node.data then
            return SEARCH(data, node.left)
        else
            return SEARCH(data, node.right)
        end if
    end if
end procedure
```

Adding

- Follow the same steps as searching until you find the data in the tree or reach a leaf node.

Adding

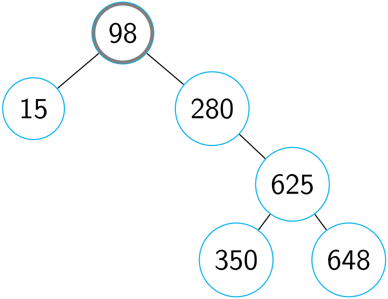
- Follow the same steps as searching until you find the data in the tree or reach a leaf node.
- If the data is not in the tree (i.e. you are at a leaf node), add a new node containing the data as either the left or right child of the leaf node.

Adding

- Follow the same steps as searching until you find the data in the tree or reach a leaf node.
- If the data is not in the tree (i.e. you are at a leaf node), add a new node containing the data as either the left or right child of the leaf node.
- If the data is in the tree, then what happens is implementation-defined. Some implementations may do nothing, some implementations may update the data item, and some implementations may add a duplicate.

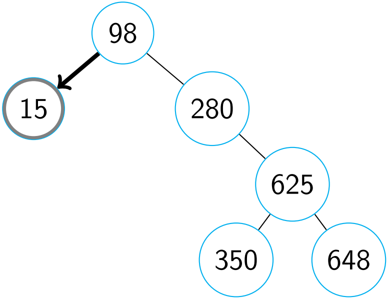
Adding

For example, if you were adding 40 (a gray circle represents the node being looked at):



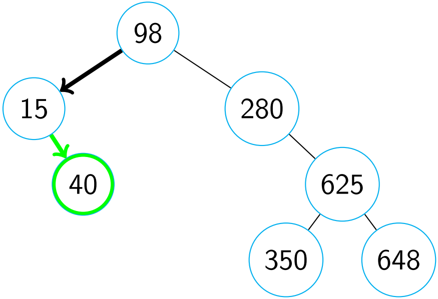
Adding

For example, if you were adding 40 (a gray circle represents the node being looked at):



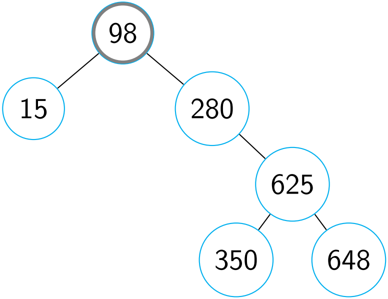
Adding

For example, if you were adding 40 (a gray circle represents the node being looked at):



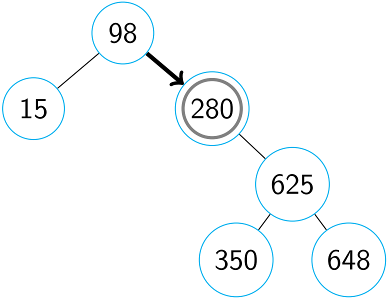
Adding

For example, if you were adding 200 (a gray circle represents the node being looked at):



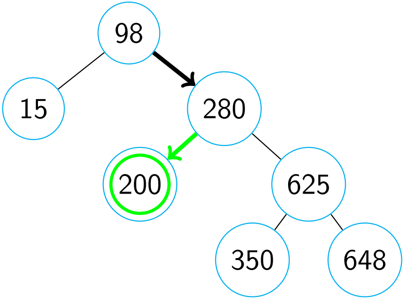
Adding

For example, if you were adding 200 (a gray circle represents the node being looked at):



Adding

For example, if you were adding 200 (a gray circle represents the node being looked at):



Adding

```
procedure Add(data, node)
    if node is not valid then
        return new node containing data
    else
        if data = node.data then
            Implementation-defined behavior
        else if data < node.data then
            node.left = Add(data, node.left)
        else
            node.right = Add(data, node.right)
        end if
        return node
    end if
end procedure
```

Removing

- Follow the same steps as searching until you find the data in the tree or go off of the tree (in which case the data is not in the tree).

Removing

- Follow the same steps as searching until you find the data in the tree or go off of the tree (in which case the data is not in the tree).
- To remove a node:

Removing

- Follow the same steps as searching until you find the data in the tree or go off of the tree (in which case the data is not in the tree).
- To remove a node:
  - If the node to be removed has no children, then that node can simply be removed, and the parent node will no longer have a left/right child.

Removing

- Follow the same steps as searching until you find the data in the tree or go off of the tree (in which case the data is not in the tree).
- To remove a node:
  - If the node to be removed has no children, then that node can simply be removed, and the parent node will no longer have a left/right child.
  - If the node to be removed has one child, then that child node takes the place of this node. The parent node left/right child will be the current node's left/right child.

Removing

- Follow the same steps as searching until you find the data in the tree or go off of the tree (in which case the data is not in the tree).
- To remove a node:
  - If the node to be removed has no children, then that node can simply be removed, and the parent node will no longer have a left/right child.
  - If the node to be removed has one child, then that child node takes the place of this node. The parent node left/right child will be the current node's left/right child.
  - If the node to be removed has two children, then either the predecessor or the successor node takes the place of this node (which one is used is implementation-defined).

Removing

(Assume current node refers to the node being removed.)

- The predecessor node is the node that has the largest data, but is less than the data in the current node (in other words, the node with the largest data in the left subtree of the current node).

Removing

(Assume current node refers to the node being removed.)

- The predecessor node is the node that has the largest data, but is less than the data in the current node (in other words, the node with the largest data in the left subtree of the current node).
- The successor node is the node that has the smallest data, but is greater than the data in the current node (in other words, the node with the smallest data in the right subtree of the current node).

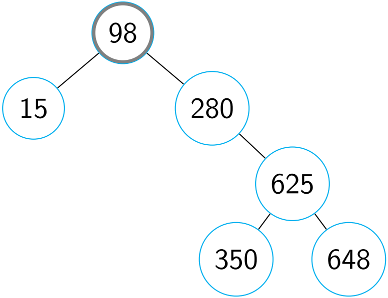
Removing

(Assume current node refers to the node being removed.)

- The predecessor node is the node that has the largest data, but is less than the data in the current node (in other words, the node with the largest data in the left subtree of the current node).
- The successor node is the node that has the smallest data, but is greater than the data in the current node (in other words, the node with the smallest data in the right subtree of the current node).
- To think of this in another way, if you were to do an inorder traversal of the subtree of the current node, then the predecessor data is the data item that is to the left of the current data item, and the successor data is the data item that is to the right of the current data item.

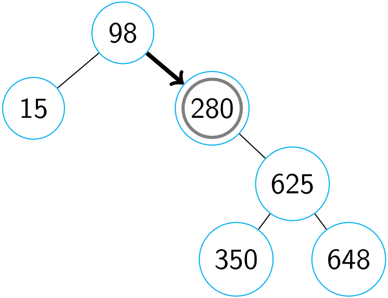
Removing

For example, if you were removing 350:



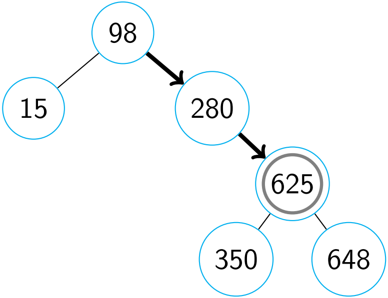
Removing

For example, if you were removing 350:



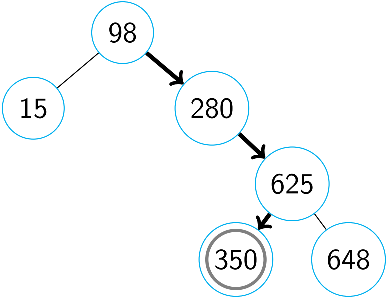
Removing

For example, if you were removing 350:



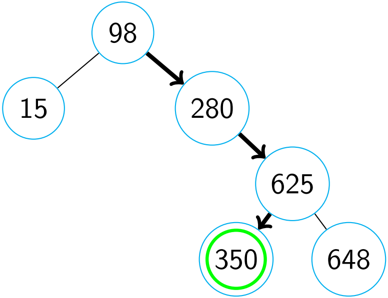
Removing

For example, if you were removing 350:



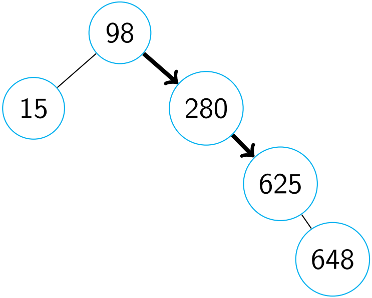
Removing

For example, if you were removing 350:



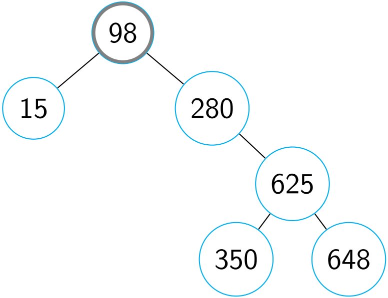
Removing

For example, if you were removing 350:



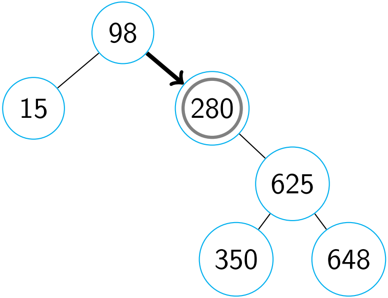
Removing

For example, if you were removing 280:



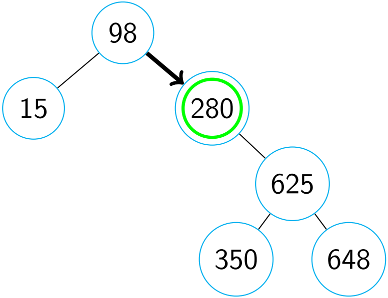
Removing

For example, if you were removing 280:



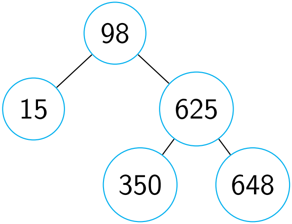
Removing

For example, if you were removing 280:



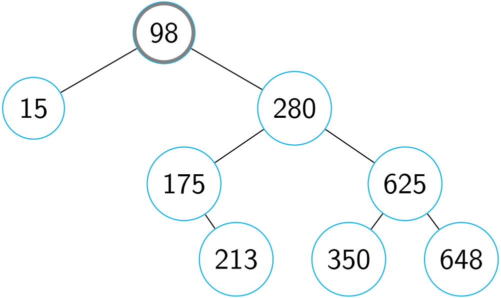
Removing

For example, if you were removing 280:



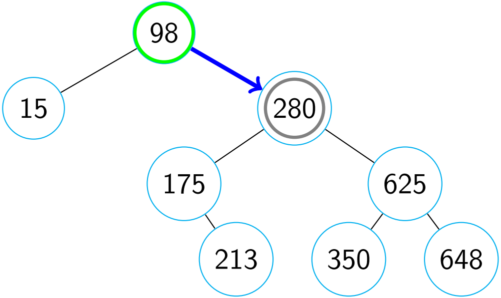
Removing

For example, if you were removing 98 (using the successor):



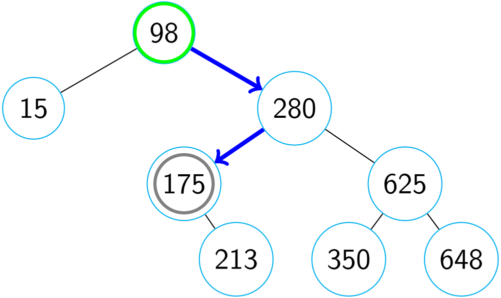
Removing

For example, if you were removing 98 (using the successor):



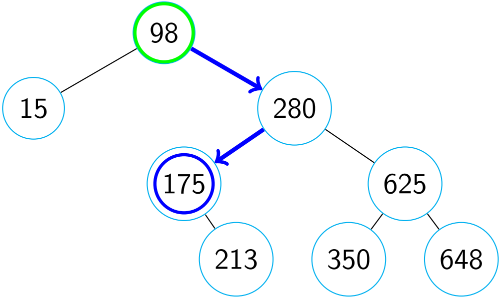
Removing

For example, if you were removing 98 (using the successor):



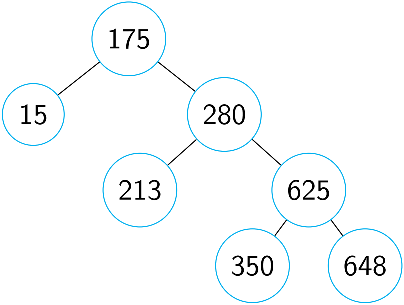
Removing

For example, if you were removing 98 (using the successor):



Removing

For example, if you were removing 98 (using the successor):



The successor of 98 was 175; as a result, 175 was moved up to where 98 was. (Note that the predecessor of 98 was 15.)

Removing

procedure REMOVE(*data*, *node*)
  if *node* is not valid **then**
    data is not in the tree
  else
    if $data = node.data$ **then**
      Remove the node, considering the three cases (0 children, 1 child, 2 children)
      Return the node that should take this position

Removing

```
        else if data < node.data then
            node.left = REMOVE(data, node.left)
        else
            node.right = REMOVE(data, node.right)
        end if
        return node
    end if
end procedure
```

Performance

- In the average case, the BST divides the data in half perfectly at each level. Therefore, insertion, search, and deletion are all $O(\log n)$ in the average case.

Performance

- In the average case, the BST divides the data in half perfectly at each level. Therefore, insertion, search, and deletion are all $O(\log n)$ in the average case.
- In the best case, search can be $O(1)$ if what you search for always happens to be at the root of the tree. (Insertion and deletion will still be $O(\log n)$.)

Performance

- In the average case, the BST divides the data in half perfectly at each level. Therefore, insertion, search, and deletion are all $O(\log n)$ in the average case.
- In the best case, search can be $O(1)$ if what you search for always happens to be at the root of the tree. (Insertion and deletion will still be $O(\log n)$.)
- The worst case, however, is if the BST doesn't divide the data in half. In this case, the BST might be closer to a linked list. (How might this happen?) In this case, insertion, search, and deletion are all $O(n)$.