Notes - Binary Search Tree Usage - JavaScript

- Dr Nick Hayward

Basic outline for working with a binary search tree in JavaScript.

Contents

- Intro
- Links and usage
    - search pattern
- BST with JavaScript
    - custom methods
    - has()
    - add()
    - size()
- Removing nodes from a BST
    - delete() - part 1
    - delete() - part 2
    - delete() - part 3

Intro

A binary search tree (BST) has a non-linear insertion algorithm.

BST is similar in nature to a doubly-linked list.

This type of list is a linked data structure, which includes a set of sequentially linked records, commonly known as nodes. Each node defines three fields,

- link field - `previous` node in sequence of nodes
- link field - `next` node in sequence of nodes
- one `data` field

The link fields may be represented using a common example of a convoy, a train &c. e.g. Linked ships sailing in a convoy.

Links and usage

A binary search tree defines its pointers as `left` and `right` to help indicate any duplication of logic &c.

The algorithm may be detected, thereby providing support for left and right traversal.

A binary search tree node's pointers are typically called "left" and "right" to indicate subtrees of values relating to the current value.

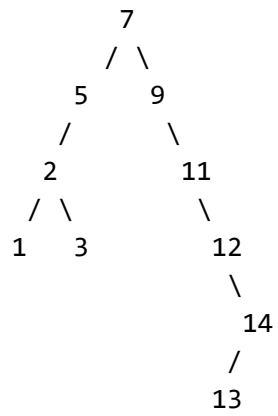
A simple JavaScript implementation of such a node is as follows:

```
const node = {
    value: 123,
    left: null,
    right: null
}
```

So, the BST is a unique tree due to its inherent ordering of nodes based on value.

Any child nodes in a left subtree are always less than the parent node's value. Whilst the converse holds for a right subtree, values in the subtree will always be greater.

For example,



As we traverse the tree, we may check the key of the current node. If the search key is less than the current node's value, follow the left link. Otherwise, follow the right link.

So, position of node values is based on a few factors.

- value of node
- value of root
- order of insertion.

e.g.

- root is set to 7
- 5 is less than root - insert as left link
- 9 is greater than root - insert as right link
- 2 is less than root - follow left link
    - 2 is less than 5 - left link is null - insert as left link
- ...

& repeat for additional inserts.

search pattern

With a clearly defined pattern, searching a BST becomes easier to reduce to a repeatable pattern.

Traverse left for a value less than the current node, and right for a greater value.

This simple pattern is helped by the exclusion of duplicates.

Sample BSTs may be found at the following URL,

- https://en.wikipedia.org/wiki/File:Binary_search_tree.svg

In the above example, we currently have a depth of 5. This means the furthest we need to travel is 5 nodes from the root to find a value.

BSTs also have a natural sorted order due to thd insertion algorithm. This makes BSTs particularly useful for quick searching as we eliminate options at each node.

BST with JavaScript

There are a few options for implementing BSTs in JavaScript.

For example, we may extend an object's prototype to include various custom functions to manage a BST.

A non-map implementation might include the following,

- define initial data and props for constructor
- extend Prototype - add custom methods
    - add
    - has
    - delete
    - size
- expose universal interface

e.g.

```
/*
* Constructor BST
*/
function BinarySearchTree() {
    // instantiated object - private prop - root default...
    this._root = null;
}

/*
* Prototype
* -extend with custom functions
* - methods
*/
BinarySearchTree.prototype = {
    // extend - custom functions
    add: function(value) {

    },
    has: function(value) {
```

```
    },
    delete: function(value) {

    },
    size: function(

    }
};
```

custom methods

Commonly, it's easiest to begin such custom methods with an outline of a `has()` method.

This allows us to define a general structure for querying a BST.

This method defines a single parameter for value, and returns `true` if the value is found and `false` if null found.

Its logic follows a basic binary search algorithm to determine presence of a value.

has()

For example, we might consider the following initial outline for querying keys.

```
has: function(value){
    const found = false,
        current = this._root;

    // check node is available for search...
    while(!found && current){

        // check node - if value less than current node's, go LEFT
        if (value < current.value){
            // update 'current' prop
            current = current.left;
        // check node - if value greater than current node's, go RIGHT
        } else if (value > current.value){
            // update 'current' prop
            current = current.right;
        //check node - values are equal, found node...
        } else {
            // update boolean...
            found = true;
        }
    }
    // return search status...
    return found;
},
```

The above search starts at the root, initially checking that there is a `root` key and that the key has not been found. `current` is initially set to `root` to beging BST traversal.

This sets the `while` loop running, which allows the following checks to be executed,

- check search value against current node value
    - if less than - set current to left link
- check search value again...
    - if greater than - set current to right link
- otherwise - value has been `found`
    - `found` updated to true
    - `while` loop broken
- `contains` now complete...

add()

We may also use the same underlying pattern for node insertion. However, we need to modify the search for a place to insert a node instead of returning an existing value.

```
add: function(value){
    // define a new node - placeholder object & props...
    const node = {
            value: value,
            left: null,
            right: null
        },
        // variable for current node - use during BST traversal...
        current;

    // CHECK -  no items yet in the BST
    if (this._root === null){
        // ROOT - BST empty - set root to current node -
        this._root = node;
    } else {
        // update current prop - set to root node
        current = this._root;

        // TRAVERSE - begin traversal of BST from current node - start at root
        while(true){

            // check node - if value less than current node's, go LEFT
            if (value < current.value){
                // check node - if no node in left link
                if (current.left === null){
                    // update current prop - set `left` to new node
                    current.left = node;
                    // EXIT - node inserted as `left` link
                    break;
                } else {
                    // node set to existing left link...
                    current = current.left;
```

```
                }

            // check node - if value greater than current node's, go RIGHT
            } else if (value > current.value){
                // chevk node = if no node in right link
                if (current.right === null){
                    // update current prop - set `right` to new node
                    current.right = node;
                    // EXIT - node inserted as `right` link
                    break;
                } else {
                    // node set to existing right link...
                    current = current.right;
                }

            // if new value = current one - ignore
            } else {
                break;
            }
        }
    }
},
```

size()

To determine just the size of a BST, we may traverse the tree in any order.

However, we may also need to flatten the tree to an array, map &c. For such purposes, we may abstract a `traverse()` function to ensure an ordered traversal.

```
traverse: function(process){
    // inner scope helper function - pass node...call recursively
    function inOrder(node){
        // check node exists
        if (node) {
            // check node - if left link exists
            if (node.left !== null){
                // call recursively - pass current node from subtree - checks
extent of subtree...
                inOrder(node.left);
            }
            // call the passed process method on this node
            process.call(this, node);

            // traverse the right subtree
            if (node.right !== null){
                inOrder(node.right);
            }
        }
    }
```

```
    // define start node - pass root
    inOrder(this._root);
},
```

This `traverse()` function defines a single parameter, `process`. The passed argument should be a function, which may be executed on every node in the tree.

The function `inOrder`, is used to recursively traverse the tree. However, the recursion only works when left and right links exist. This rule is designed to reduce reference to `null` nodes to a bare minimum.

The `traverse()` function begins the traversal from the root, and the passed `process()` function handles each node.

We may now use this abstracted function, `traverse()`, with a custom `size()` function

```
size: function(){
    const length = 0;

    this.traverse(function(node){
        length++;
    });

    return length;
},
```

The `size()` function calls the above abstracted `traverse()` function passing a custom function for counting the nodes.

Removing nodes from a BST

Removing or deleting nodes from a BST can become complex due to necessary balancing of the tree.

For each node removed, we need to check if it's the root.

The removal of the root node is handled in a similar manner to other nodes, except it will also need to be replaced. This is a simple matter of tree integrity, and may be handled as a special case in the logic.

delete() - part 1

The first part of a node's removal is checking it exists in the defined BST.

```
delete: function(value){

    let found = false,
        parent = null,
        current = this._root,
        childCount,
        replacement,
```

```
            replacementParent;

        // check node - if not found & node still exists
        while(!found && current){

            // check value - if less than current - traverse left
            if (value < current.value){
                parent = current;
                current = current.left;
            }
            // check value - if greater than current - travers right
            else if (value > current.value){
                parent = current;
                current = current.right;
            }
            // value found...
            else {
                found = true;
            }
        }

        // continue - value found...
        if (found){
            // continue
        }

    },
```

We may check the required node using a standard *binary search*. As expected, we traverse left if the value is less than the current node, and right if the value is greater.

As part of the traversal, we also monitor the parent node as the passed node will need to be removed from its parent.

When the requested node is found, `current` defines the node to remove. This is the initial part of defining a remove or delete option for binary search trees.

delete() - part 2

Once we've found a node in the BST, we need to consider options for removing the node.

In effect, there are three applicable conditions we need to consider

- a leaf node
- a node with one child
- a node with two children

The first two cases are easy to implement. A leaf node may simply be deleted from the tree, whilst a child may replace the parent leaf node upon deletion.

The third condition is more involved in its modification of the tree. We need to determine if the selected node has any children, how many, and if it is the root.

If the leaf node selected for deletion is a root, the updated decision tree is relatively simple.

For example,

```
// root node - special case
if (current === this._root){
    // check no. of child nodes - execute matching case
    switch(childCount){

        // no children - erase root
        case 0:
            this._root = null;
            break;

        // one child - child  is now root
        case 1:
            this._root = (current.right === null ?
                           current.left : current.right);
            break;

        // two children -
        case 2:

            //TODO

        // no default - one of above cases always matched...

    }
}
```

So, for a root leaf node the deletion is simple to handle and implement. However, for a child leaf node, we need to check and update the tree. For example,

- value lower than parent
    - left pointer must be reset
    - reset to `null` (no children) or node's left child pointer
- value higher than parent
    - right pointer must be reset
    - reset to `null` (no children) or node's right child pointer

For a child node, we may initially check for 0 or 1 children of the current selected node.

```
switch (childCount){

    // no children - delete from tree
    case 0:// check delete value relative to parent
        if (current.value < parent.value){
            // value < parent - null parent's left pointer
            parent.left = null;
        } else {
```

```
            // else - null parent's right pointer
            parent.right = null;
        }
        break;

    // one child - replace deleted parent node
    case 1: // check value relative to parent
        if (current.value < parent.value){
            // value < parent - reset left pointer
            parent.left = (current.left === null ?
                           current.right : current.left);
        } else {
            // value > parent - reset right pointer
            parent.right = (current.left === null ?
                           current.right : current.left);
        }
        break;

}
```

We need to update the pointer on the parent based on the value of the node to delete.

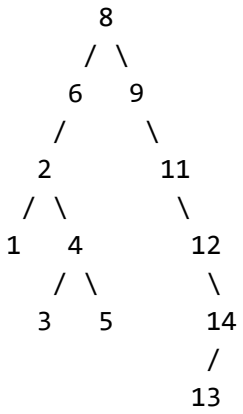
If the deleted node's value was less than the parent, we need to reset the left pointer either to `null` or to the left pointer of the deleted node.

Likewise, if the deleted node's value was greater than the parent, we need to reset the right pointer.

delete() - part 3

The most complex deletion is a node with two children.

If we consider the following tree, we can see the issue with deletion of node `2`,



The issue, of course, is how we now update the tree based on the child nodes of the deleted node.

There are two common options to consider for such trees relative to the deleted node,

- in-order predecessor - left child

- in-order successor - left-most child of the right subtree

So, we may end up with either `1` or `3` replacing `2` in the tree.

Either option is acceptable, and may be used to update the tree. For example, we may consider the following

- in-order predecessor = value before deleted value
    - examine left subtree of deleted node and select right-most descendant
- in-order successor = value immediately after deleted value
    - examine right subtree of deleted node and select left-most descendant

Each option will also require a traversal of the tree to find the required node.

For a root node with two children, we may consider the update as follows

```
/* root - two children
* - in-order predecessor
*       - check left subtree
*               - select right most descendant
*
*             8
*            / \
*           6   9
*          / \   \
*         2   7   11
*        / \       \
*       1   4       12
*          / \       \
*         3   5      14
*                    /
*                   13
*/
case 2: // e.g. delete root node - 8
                // check left subtree - get left of root (6)
    replacement = this._root.left;

    // check right-most child node - if not null
    while (replacement.right !== null){ // (7)
        replacementParent = replacement; // (6)
        replacement = replacement.right; // (7)
    }

    // check replacement parent
    if (replacementParent !== null){ // (6)

        // check for left node of replace - if exists, move to right of parent
        replacementParent.right = replacement.left; // (null)

        // new root - update with child nodes from existing root node
        replacement.right = this._root.right;
        replacement.left = this._root.left;
    } else {
        // new root - assign existing root's child nodes
```

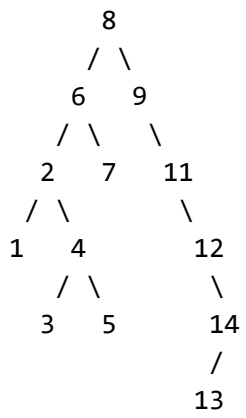
```
        replacement.right = this._root.right;
    }

    // new root - UPDATE root value after deletion of root...
    this._root = replacement;
```

This example will always look for the *in-order predecessor*, which means

- check left subtree
- select right most descendant

So, if the previous tree was modified as follows



If we deleted the root node `8`, we would update the root to the node `7`. As expected, this follows the pattern

- check left subtree = `6`
- select right most descendant = `7`

Likewise, if we deleted the `6` node using *in-order predecessor*, we would check the left subtree, node `2`, and then traverse following the right pointers.

We would now replace the deleted node, `6`, with the node `5`. If we used *in-order successor*, we would end up replacing the deleted node in this tree with node `7`.

For a child node with two children, we may add the following case to the existing `switch` statement

```
/* child node - two children
* - in-order predecessor
*       - check left subtree
*       - select right most descendant
*/
case 2:
    // two children - reset pointers for new traversal
    replacement = current.left;
    replacementParent = current;

    //find the right-most node
```

```
    while(replacement.right !== null){
        replacementParent = replacement;
        replacement = replacement.right;
    }

    replacementParent.right = replacement.left;

    // assign - children to replacement
    replacement.right = current.right;
    replacement.left = current.left;

    // add replacement to correct node in tree
    if (current.value < parent.value){
        // current < parent - add replacement to parent's left pointer
        parent.left = replacement;
    } else {
        // current > parent - add replacement to parent's right pointer
        parent.right = replacement;
    }
```

Either of these options will work to update the tree.

For example, our initial tree may be represented as follows,

```
initial BST = {
  "_root": {
    "value": 8,
    "left": {
      "value": 6,
      "left": {
        "value": 2,
        "left": {
          "value": 1,
          "left": null,
          "right": null
        },
        "right": {
          "value": 4,
          "left": {
            "value": 3,
            "left": null,
            "right": null
          },
          "right": {
            "value": 5,
            "left": null,
            "right": null
          }
        }
      },
      "right": {
        "value": 7,
```

```
        "left": null,
        "right": null
      }
    },
    "right": {
      "value": 9,
      "left": null,
      "right": {
        "value": 11,
        "left": null,
        "right": {
          "value": 12,
          "left": null,
          "right": {
            "value": 14,
            "left": {
              "value": 13,
              "left": null,
              "right": null
            },
            "right": null
          }
        }
      }
    }
  }
}
```

If we now delete node 6, using *in-order predecessor*, our updated tree may be rendered as follows

```
after node deletion = {
  "_root": {
    "value": 8,
    "left": {
      "value": 5,
      "left": {
        "value": 2,
        "left": {
          "value": 1,
          "left": null,
          "right": null
        },
        "right": {
          "value": 4,
          "left": {
            "value": 3,
            "left": null,
            "right": null
          },
          "right": null
        }
      },
```

```
        "right": {
          "value": 7,
          "left": null,
          "right": null
        }
      },
      "right": {
        "value": 9,
        "left": null,
        "right": {
          "value": 11,
          "left": null,
          "right": {
            "value": 12,
            "left": null,
            "right": {
              "value": 14,
              "left": {
                "value": 13,
                "left": null,
                "right": null
              },
              "right": null
            }
          }
        }
      }
    }
  }
```

However, by just using one option exclusively, we may end up with an unbalanced tree.

We may, however, consider modifying the logic to ensure we monitor the tree to maintain a *self-balancing search tree*.