Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 9

Dr Nick Hayward

DEV Week Assessment

Course total = 15%

- continue development of application
  - *built from scratch*
  - *continue design and development of initial project outline and design*
  - *working app (as close as possible...)*
  - *NO blogs, to-do lists, note-taking...*
  - *...*
- outline research conducted
- describe data chosen for application
- define algorithms and data structures used in app
  - *why choose these options?*
  - *how have they been used?*
  - *define current performance &c.?*
  - *define testing of implementation & usage*
- show any prototypes, patterns, and designs
- how did you respond to peer reviews?

DEV week assessment will include the following:

- brief presentation or demonstration of current project work
  - *~ 5 to 10 minutes per group*
  - *analysis of work conducted so far*
    - e.g. during semester & DEV week
  - *presentation and demonstration*
    - outline current state of app
    - explain what works & does not work
    - show implemented designs since project outline & mockup
    - show latest designs and updates
    - responses to peer reviews
    - ...
  - *due Tuesday 17th & Thursday 19th March 2020 @ 10am*

binary search tree - usage - search pattern

- clearly defined pattern - searching a BST becomes easier to reduce to a repeatable pattern
- traverse left for a value less than current node, and right for a greater value
- simple pattern is helped by exclusion of duplicates
- in last example - currently have a depth of 5 means furthest we need to travel is 5 nodes from root to find a value
- BSTs also have a natural sorted order
  - *due to the insertion algorithm*
- makes BSTs particularly useful for quick searching
  - *eliminate options at each node*

Algorithms and Data Structures

binary search tree - usage - BST with JavaScript - part 1

- a few options for implementing BSTs in JavaScript
- e.g. 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*

Algorithms and Data Structures

binary search tree - usage - BST with JavaScript - part 2

- initial coded example

```
/*
* 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(

    }
};
```

Algorithms and Data Structures

binary search tree - usage - BST with JavaScript - part 3

- easiest to begin such custom methods with an outline of a `has()` method
- may define a general structure for querying a BST
- method defines single parameter for value
  - *returns `true` if value is found and `false` if null found*
- logic follows a basic binary search algorithm
  - *to determine presence of a value*

binary search tree - usage - BST with JavaScript - part 4

- coded example for `has()` method
- e.g. consider 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;
},
```

binary search tree - usage - BST with JavaScript - part 5

- search starts at root
  - *initially checking there is a `root` key and key has not been found*
  - *`current` is initially set to `root` to begin BST traversal*
- sets `while` loop running
- allows 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...*

Video - Algorithms and Data Structures

binary search trees - part 3

Binary Search Trees - has/add - UP TO 7:24

Source - Trees - Java - YouTube

Algorithms and Data Structures

binary search tree - usage - BST with JavaScript - part 6

- may also use same underlying pattern for node insertion
  - *defining* `add()` *method*
- need to modify 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;
            }
        }
    }
},
```

Algorithms and Data Structures

binary search tree - usage - BST with JavaScript - part 7

- to determine just size of BST
  - *traverse tree in any order*
- may also need to flatten tree to an array, map &c.
  - *e.g. abstract a `traverse()` function to ensure ordered traversal*

```javascript
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);
},
```

Video - Algorithms and Data Structures

binary search trees - part 4

Binary Search Trees - Traverse - UP TO 3:55

Source - Trees - Java - YouTube

binary search tree - usage - BST with JavaScript - part 8

- `traverse()` function defines single parameter, `process`
  - *passed argument should be a function*
  - *may be executed on every node in tree*
- function `inOrder`
  - *used to recursively traverse tree*
  - *n.b. recursion only works when left and right links exist*
- rule is designed to reduce reference to `null` nodes to a bare minimum
- `traverse()` function begins traversal from root
  - *passed `process()` function handles each node...*

binary search trees - part 5

Binary Search Trees - traversal function - UP TO 9:27

Source - Trees - Java - YouTube

binary search tree - usage - BST with JavaScript - part 9

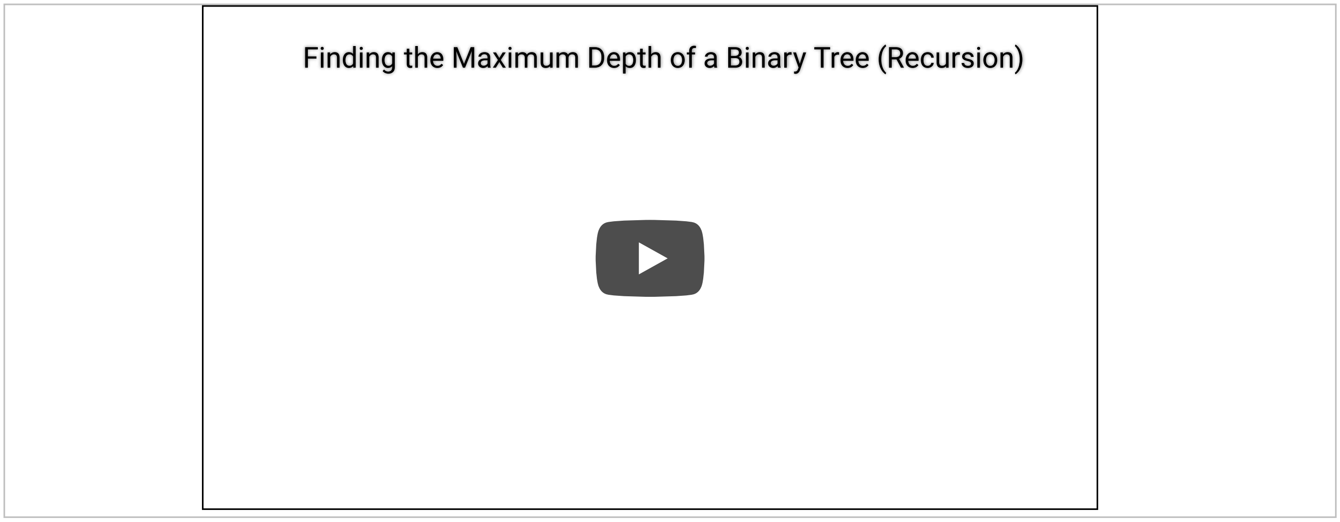
- now use abstracted function, `traverse()`
  - *e.g. use with a custom `size()` function*

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

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

    return length;
},
```

- `size()` function calls above abstracted `traverse()` function
  - *passing a custom function for counting nodes*

Video - Algorithms and Data Structures

binary search trees - maximum depth



Maximum depth of binary tree with recursion - UP TO 2:34

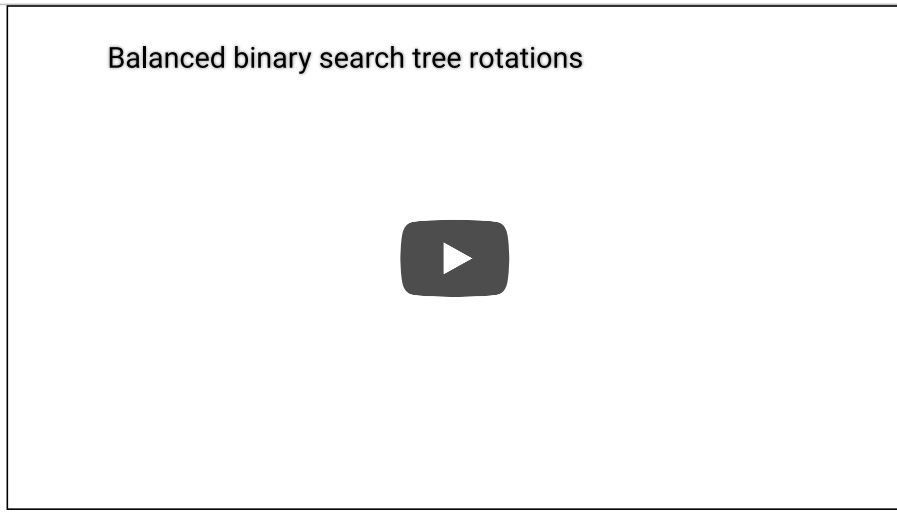
Source - Trees - Max height using recursion - YouTube

binary search tree - usage - BST with JavaScript - part 10

- removing or deleting nodes from a BST can become complex
  - *due to necessary balancing of tree*
- for each node removed
  - *need to check if it's root*
- removal of root node is handled in a similar manner to other nodes
  - *except it will also need to be replaced*
- simple matter of tree integrity
  - *may be handled as a special case in the logic*

Video - Algorithms and Data Structures

binary search trees - integrity and balance - part 1



Trees - Balancing - UP TO 1:22

Source - Trees - Balancing - YouTube

Algorithms and Data Structures

binary search tree - usage - BST with JavaScript - part 11

delete() - part 1

- first part of a node's removal is checking it exists in defined BST

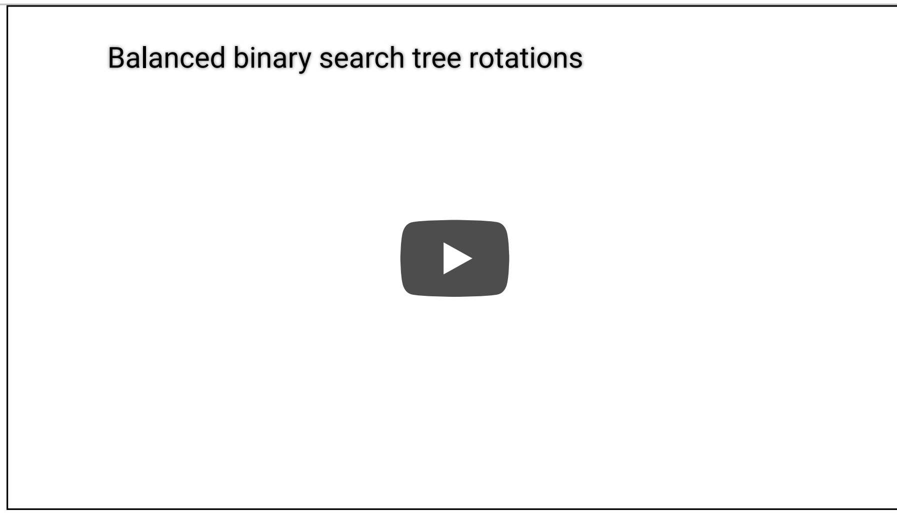
```
delete: function(value){

    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
    }

},
```

binary search tree - usage - BST with JavaScript - part 12

- may check required node using a standard *binary search*
- traverse left if value is less than current node
- traverse right if value is greater
- as part of traversal
    - *monitor parent node as passed node will need to be removed from its parent*
- when requested node is found
    - `current` *defines node to remove*
    - *initial part of defining a remove or delete option for binary search trees*

Video - Algorithms and Data Structures

binary search trees - integrity and balance - part 2



Trees - Balancing and Rotation - UP TO 4:21

Source - Trees - Balancing & Rotation- YouTube

Algorithms and Data Structures

binary search tree - usage - BST with JavaScript - part 13

delete() - part 2

- node in BST is fooound
  - *consider options for removing node*
- i.e. three applicable conditions we need to consider
  - *a leaf node*
  - *a node with one child*
  - *a node with two children*
- first two cases are easy to implement
  - *leaf node may simply be deleted from tree*
  - *a child may replace parent leaf node upon deletion*
- third condition is more involved in its modification of the tree
  - *need to determine if selected node has children, how many, and if it's root…*
- if leaf node selected for deletion is root
  - *updated decision tree is relatively simple*

binary search tree - usage - BST with JavaScript - part 14

- 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...

    }
}
```

- for a root leaf node
  - *deletion is simple to handle and implement*

binary search tree - usage - BST with JavaScript - part 15

- for a child leaf node
  - *need to check and update tree*

- e.g.
  - *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

Algorithms and Data Structures

binary search tree - usage - BST with JavaScript - part 16

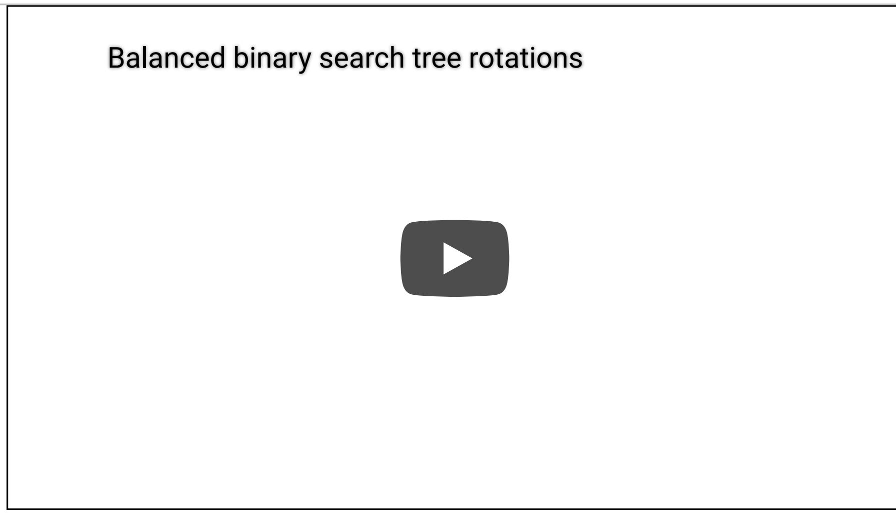
- for a child node
  - *initially check for 0 or 1 children of current selected node*

```
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;

}
```

- need to update pointer on parent based on value of node to delete
- if deleted node's value was less than parent
  - *reset left pointer either to `null` or to left pointer of deleted node*
- if deleted node's value was greater than parent
  - *need to reset right pointer*

Video - Algorithms and Data Structures

binary search trees - integrity and balance - part 3



Trees - Balancing and Rotation - UP TO 6:08

Source - Trees - Balancing & Rotation- YouTube

binary search tree - usage - BST with JavaScript - part 17

delete() - part 3

- most complex deletion is a node with two children
- e.g. consider following tree - issue with deletion of node 2

```
          8
         / \
        6   9
       /     \
      2       11
     / \       \
    1   4       12
       / \       \
      3   5       14
                  /
                13
```

- issue is how we now update tree based on child nodes of deleted node

binary search tree - usage - BST with JavaScript - part 18

- two common options to consider for such trees relative to deleted node
  - *in-order predecessor - left child*
  - *in-order successor - left-most child of sright subtree*
- may end up with either 1 or 3 replacing 2 in tree
- either option is acceptable
  - *i.e. may be used to update tree*
- e.g. 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
- n.b. each option also requires traversal of tree to find required node

Video - Algorithms and Data Structures

binary search trees - in-order traversal

Binary Search Trees - In-Order Traversal - UP TO 2:48

Source - Trees - In-Order Traversal - YouTube

binary search tree - usage - BST with JavaScript - part 19

- for a root node with two children

```
/* 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 always looka for *in-order predecessor*
  - *check left subtree*
  - *select right most descendant*

binary search tree - usage - BST with JavaScript - part 20

- if previous tree was modified as follows

```
      8
     / \
    6   9
   / \   \
  2   7   11
 / \       \
1   4       12
   / \       \
  3   5       14
             /
            13
```

- if we deleted root node 8
  - *update the root to the node 7*
- follows the pattern
  - *check left subtree = 6*
  - *select right most descendant = 7*

binary search tree - usage - BST with JavaScript - part 21

- if we deleted 6 node using *in-order predecessor*
  - *check left subtree, node 2*
  - *then traverse following right pointers*
- now replace deleted node 6 with node 5
- n.b. if we used *in-order successor*
  - *end up replacing deleted node in this tree with node 7*

Video - Algorithms and Data Structures

binary search trees - review of deletion

Trees - Review of Deletion - UP TO 7:56

Source - Trees - Deletion - YouTube

binary search tree - usage - BST with JavaScript - part 22

- for a child node with two children
  - *add following case to 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;
    }
```

Algorithms and Data Structures

binary search tree - usage - BST with JavaScript - part 23

- either of these options will work to update tree
- e.g. 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
          }
        }
      }
    }
  }
}
```

binary search tree - usage - BST with JavaScript - part 24

- now delete node 6
  - *using in-order predecessor*
- 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
        }
      }
    }
  }
 }
}
```

- n.b. by just using one option exclusively, we may end up with an unbalanced tree
  - *may consider modifying logic to ensure monitor of tree to maintain a self-balancing search tree...*

tables - intro

- *Hash tables* are a particularly useful, and fast, data structure
- conceptually define a hash table data structure as follows
  - *store each item in an easily determined location*
    - so no need to search for item
  - *no ordering to maintain*
    - for insertion and deletion of items
- this data structure has impressive performance
  - *i.e. time is concerned*
- there is a tradeoff with additional memory requirements
  - *conceptually harder implementation for custom patterns*

abstract data type - intro

- storage options and patterns described conceptually as an abstract data type
- need to define a specification for this particular abstract data type
- after defining specification
  - *then choose data structure*
- data structure as foundation for this implementation

Algorithms and Data Structures

abstract data type - table

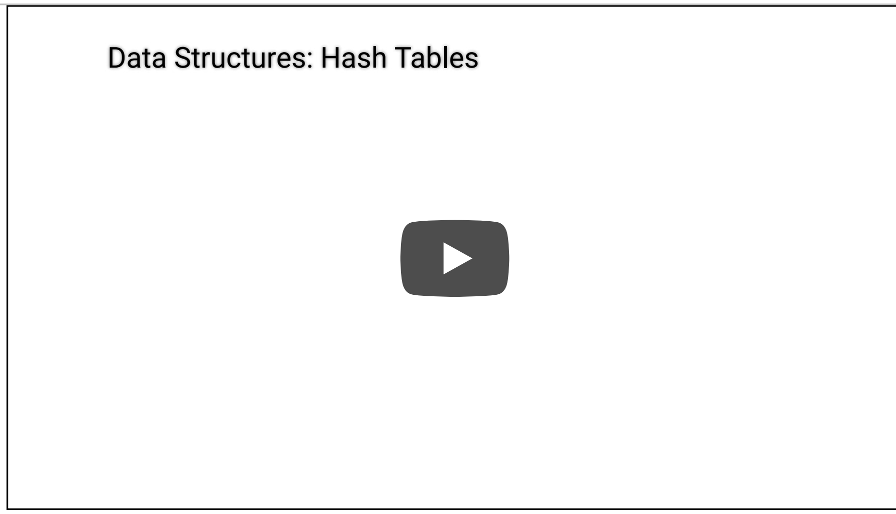
- initial specification for abstract data type outlined
- table may be used to store objects
  - *e.g.*

id	place	country
5	philae	egypt
21	athens	greece
37	rome	italy
24	sparta	greece

- objects may be arbitrarily complicated, e.g.
  - *each object has unique key*
  - *keys may be compared for equality*
  - *keys used to identify objects*
- assume there are methods &c. for the following
  - *check for empty or full table*
  - *insert object into table - assuming table is not already full*
  - *for a given key - retrieve object for that key*
  - *for a given key - update object for that key*
    - commonly replace current object at key with new object
  - *for a given key - delete object for that key*
    - assumes key is already stored in table
  - *traverse or list each item in current table*
    - if order defined - should follow increasing order...
- outline predicated on simple assumption
  - *each object is uniquely identified by its key*

Video - Algorithms and Data Structures

hash tables - part 1



Hash tables - intro - UP TO 1:15

Source - Hash tables - intro - YouTube

implementations of table data structure - intro

- consider three common approaches for implementation
  - *custom design and development of table data structure*
- e.g. might use one of the following options
  - *sorted arrays*
  - *binary search trees*
  - *hash tables*

implementations of table data structure - sorted array implementation - part 1

- if we choose a sorted *array* for *table* data structure
  - *determine full or empty in constant time* `O(1)`
  - *assuming we maintain a variable for its size*
- insert an element
  - *need to find its correct position*
  - *on average takes same time as finding an element*
- find an element
  - *crucial for all operations except traversal itself*
  - *use binary search*
  - *e.g. takes* `O(Log n)`*, logarithmic time*
- consider complexity for *retrieval* and *update*
  - *also produce* `O(Log n)`*, logarithmic, times*

Algorithms and Data Structures

implementations of table data structure - sorted array implementation - part 2

- if we need to delete or insert an item
  - *need to shift following element*
  - *left for delection*
  - *right for insertion*
- e.g.

```
[3, 6, 2, 33. 17. 97]
```

- delete node `33`
  - *element `17` will need to shift to its left*
- insert node at position of node `2`
  - *element `33` &c. will need to shift to the right*
- takes average `n/2` steps
  - *such operations will have a complexity of `O(n)`.*
- ordered traversal is simple for this type of data structure
  - *may also see complexity of `O(n)`.*

implementations of table data structure - binary search tree implementation

- alternative to sorted arrays might use *binary search trees*
- whilst this is certainly possible
    - *worst case may also produce a tree that is very deep and narrow*
- unbalanced trees will have *linear* complexity for lookups
- *self-balancing binary search tree*
    - *able to produce a worst case same as average case*
- for such trees commonly see time complexity of `O(log n)`, logarithmic
    - *for insertion, deletion, search, retrieval, and update*
    - *may also see complexity of* `O(n)`*, linear, for traversal*
- downside of such self-balancing trees
    - *sheer complexity of implementation, management, and initial comprehension*

Video - Algorithms and Data Structures

stable marriage problem

A fun diversion - Stable Marriage Problem - UP TO 27:07

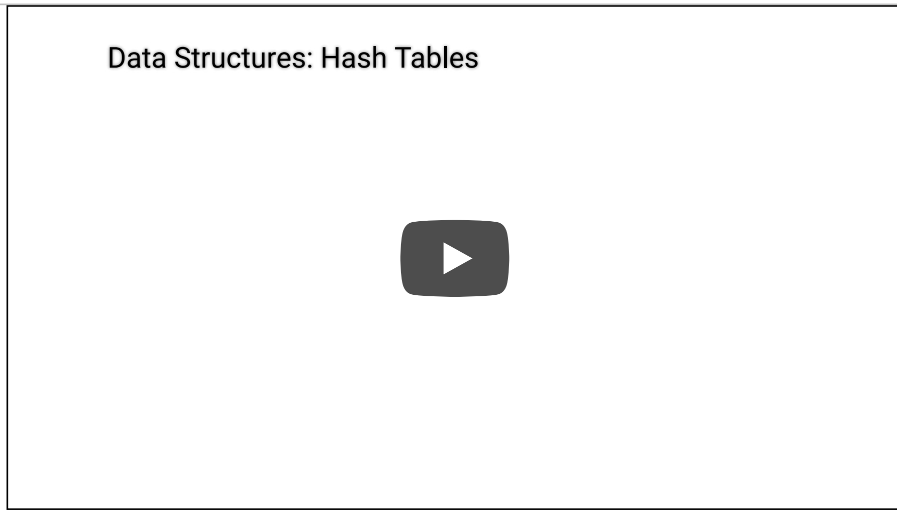
Source - Stable Marriage Problem - YouTube

Further details - The Stable Marriage Problem *implementations of table data structure - hash table implementation*

- hash tables
  - *provide a benefit for such table data structure usage*
- may expend more space
  - *i.e. than actually required or necessary*
- extra space may also be beneficial
  - *i.e. speed up inherent operations of the table*

Video - Algorithms and Data Structures

hash tables - part 2



Hash tables - hash function and index - UP TO 2:33

Source - Hash tables - hash function - YouTube

Resources

various

- The Stable Marriage Problem

videos

- Hash tables - Java - YouTube
- Hash tables - real-world usage - YouTube
- Quicksort - Java - YouTube
- So how does the browser actually render a website - YouTube
- Stable Marriage Problem - YouTube
- Symbol Table API - YouTube
- Trees - Balancing - YouTube
- Trees - Deletion - YouTube
- Trees - In-Order Traversal - YouTube
- Trees - Java - YouTube
- Trees - Max height using recursion - YouTube