|  |  |  |  |
| --- | --- | --- | --- |
| **KNUTH-MORRIS-PRATT STRING SEARCH ALGORITHM** | | | |
| **prefixTable helper function *O(n)***  two pointers + table array  1 while loop with conditions  *Arguments:* (pattern)  *Return:* table | | **patternSearch function *O(n)***  two pointers + table array + foundPatterns array  1 while loop with conditions  *Arguments:* (text, pattern)  *Return:* foundPatterns | |
| **BUBBLE SORT** | **SELECTION SORT** | | **INSERTION SORT** |
| **bubbleSort function *O(n^2)***  *1st loop* - shrink the array size backwards  *2nd loop* - check values in a new subarray for a new  max  + check if swapped | **selectionSort function *O(n^2)***  *1st loop* - assign new min indices (to  arr.length-1)  *2nd loop* - iterate to find a new min (from i+1) | | **insertionSort function *O(n^2)***  *1st loop* - select an unsorted element (from 1)  +currVal = arr[i]  *2nd loop* - insert unsorted element in the  correct spot (iterate through the left  backwards) |
| **MERGE SORT** | | | |
| **merge helper function *O(n)***  create two pointers + an empty array to be returned  3 while loops to merge two sorted arrays  *Arguments:* (arr1, arr2)  *Return:* arr | | **mergeSort function *O(nlogn)***  calculate the middle  base case: middle === 0  merge two subarrays created from the passed one  *Arguments:* (arr)  *Return:* arr (recursive) | |
| **QUICK SORT** | | | |
| **placePivot helper function *O(n)***   * create a pivotIdx = start * 1 for loop to check if the pivot is greater than the current element   if yes, pivotIdx++ and swap   * swap the pivot with the pivotIdx element   *Arguments:* (arr, start = 0, end = arr.length - 1)  *Return:* pivotIdx | | **quickSort function *O(n^2)***   * if start < end   place pivot  apply quickSort on both sides of the pivot   * return an array   *Arguments:* (arr, start = 0, end = arr.length - 1)  *Return:* arr | |
| **RADIX SORT** | | | |
| *3 helper functions:*  **getDigit(num, place)**  return the digit in a place index ***O(k)***  **digitCount(num)**  return the number of digits in a number ***O(k)***  **maxDigitCount(nums)**  return the largest number of digits encountered ***O(k)*** | | **radixSort function *O(n \* k)*** \*n – length of arr; k – max number of digits  2 nested for loops:  *1st* change the number index  *2nd* place each number in a bucket according to the digits  concatenate buckets every time after they are filled (after the inner loop)  *Arguments:* (arr)  *Return:* arr | |

|  |  |
| --- | --- |
| **SINGLY LINKED LIST** | **NODE** |
| *3 properties:* head, tail, length | *2 properties:* val, next |
| **PUSH/POP** | |
| **push instance method *O(1)***   * create a new node * if this.length === 0, set the head and tail to be a newly created node * else, set the next property on the tail to be a new node * set the tail property on the list to be a newly created node * this.length++   *Arguments:* (val)  *Return:* this | **pop instance method *O(n)***   * if this.length === 0, return undefined * 1 while loop to reach tail * set the next property on 2nd to last node to be null * set the tail to be the 2nd to the last node * this.length--   *Arguments:* ()  *Return:* the value of the removed node |
| **UNSHIFT/SHIFT** | |
| **unshift instance method *O(1)***   * create a new node * if this.length === 0, set the head and tail to be a new node * otherwise, set the new node’s next property to this.head * set a new head; this.length++   *Arguments:* (val)  *Return:* this | **shift instance method *O(1)***   * if this.length === 0, return undefined * store the current head in a variable * set a new head; set oldHead’s next property to null * this.length--; set this.tail to null if the length is 0   *Arguments:* ()  *Return:* the value of the removed node |
| **GET/SET** | |
| **get instance method *O(n)***   * if idx < 0 || idx >= this.length, return undefined * loop though the list to return the node's value at a specific index   *Arguments:* (idx)  *Return:* foundNode/undefined | **set instance method *O(n)***   * use get() to find a specific node * if nothing found, return false * else, set a new value on the node; return true   *Arguments:* (idx, val)  *Return:* true/false |
| **INSERT/REMOVE** | |
| **insert instance method *O(n)***   * if idx < 0 || idx > this.length, return false * if idx === this.length, use push() * if idx === 0, use unshift() * otherwise, use get() to access node at idx - 1 * set the next properties of the new and previous nodes; this.length++   *Arguments:* (idx, val)  *Return:* true/false | **remove instance method *O(n)***   * if idx < 0 || idx >= this.length, return undefined * if idx === this.length - 1, use pop() * if idx === 0, use shift() * otherwise, use get() to access node at idx - 1 * set the new next property for a node before the deleted one * this.length--   *Arguments:* (idx)  *Return:* the value of the removed node |
| **REVERSE** | |
| **reverse instance method *O(n)***   * swap the head and tail and create *3 pointers*: prev = null, curr = this.tail, next * iterate through the list to alter it   *Arguments:* ()  *Return:* this | |

|  |  |
| --- | --- |
| **DOUBLY LINKED LIST** \*takes up more memory for an extra pointer | **NODE** |
| *3 properties:* head, tail, length | *3 properties:* val, next, prev |
| **PUSH/POP** | |
| **push instance method *O(1)***   * create a new node * if this.length === 0, set the head and tail to be a newly created node * else, set the next property on the tail to be a new node * set the prev property on the new node to be the tail * set the tail property on the list to be a newly created node; this.length++   *Arguments:* (val)  *Return:* this | **pop instance method *O(1)***   * if this.length === 0, return undefined * store the old tail in a variable * set the tail to be the previous node * adjust prev/next properties on the last nodes; this.length— * set this.head to null if the length is 0, else set this.tail to null   *Arguments:* ()  *Return:* the value of the removed node |
| **UNSHIFT/SHIFT** | |
| **unshift instance method *O(1)***   * create a new node * if this.length === 0, set the head and tail to be a new node * otherwise, set prev/next properties on the first nodes * set a new head; this.length++   *Arguments:* (val)  *Return:* this | **shift instance method *O(1)***   * if this.length === 0, return undefined * store the current head in a variable * set a new head * adjust prev/next properties on the last nodes * this.length--; set this.tail to null if the length is 0   *Arguments:* ()  *Return:* the value of the removed node |
| **GET/SET** | |
| **get instance method *O(n)***   * if idx < 0 || idx >= this.length, return undefined * find the middle * depending on the middle, determine the iteration direction; loop though the list to return the node's value at a specific index   *Arguments:* (idx)  *Return:* foundNode/undefined | **set instance method** ***O(n)***   * use get() to find a specific node * if nothing found, return false * else, set a new value to the node; return true   *Arguments:* (idx, val)  *Return:* true/false |
| **INSERT/REMOVE** | |
| **insert instance method *O(n)***   * if idx < 0 || idx > this.length, return false * if idx === this.length, use push() * if idx === 0, use unshift() * otherwise, use get() to access node at idx - 1 * adjust prev/next properties on the appropriate nodes; this.length++   *Arguments:* (idx, val)  *Return:* true/false | **remove instance method *O(n)***   * if idx < 0 || idx >= this.length, return undefined * if idx === this.length - 1, use pop() * if idx === 0, use shift() * otherwise, use get() to access a node at idx - 1 * adjust prev/next properties on the appropriate nodes; this.length--   *Arguments:* (idx)  *Return:* the value of the removed node |
| **REVERSE** | |
| **reverse instance method *O(n)***   * swap the head and tail; iterate through the list to swap nodes' prev/next properties   *Arguments:* ()  *Return:* this | |

|  |  |
| --- | --- |
| **STACKS/QUEUES** | **NODE** |
| *3 properties:* first, last, size | *2 properties:* val, next |
| **PUSH/POP (STACK)** | |
| **push instance method *O(1)*** \*unshift in SLL   * create a new node * if size === 0, set the first and last to be a new node * otherwise, set the new node’s next property * set a new first; this.size++   *Arguments:* (val)  *Return:* this | **pop instance method *O(1)*** \*shift in SLL   * if size === 0, return undefined * store the current first in a variable * set a new first * this.size--; set this.last to null if the size is 0   *Arguments:* ()  *Return:* the value of the removed node |
| **ENQUEUE/DEQUEUE (QUEUE)** | |
| **enqueue instance method *O(1)*** \*push in SLL   * create a new node * if this.size === 0, set the first and last to be a newly created node * else, set the next property on the last to be a new node * set the last property on the list to be a newly created node * this.length++   *Arguments:* (val)  *Return:* this | **dequeue instance method *O(1)*** \*shift in SLL  *same as pop in a stack* |

|  |  |
| --- | --- |
| **MAX BINARY HEAP** | |
| *1 property:* values (array) | |
| **INSERT/EXTRACT MAX** | |
| **insert instance method *O(logn)***   * push the val into this.values array on the heap * create currIdx variable to point on the inserted value (values.length - 1) * create parentIdx variable to point on the inserted value's parent Math.floor((currIdx - 1) / 2) * loop through this.values to bubble up the inserted value   *Arguments:* (val)  *Return:* this | **extractMax instance method *O(logn)***   * create parentIdx = 0, leftIdx, rightIdx, maxIdx pointers + del, arr * if arr.length === 0, return undefined * swap the first and last elements in arr * pop the last element in arr * while maxIdx !== null   + calculate leftIdx, rightIdx; assign maxIdx = null   + if leftIdx < arr.length     - if left child is greater than the parent, maxIdx is leftIdx   + if rightIdx < arr.length     - if (maxIdx === null *AND* right child is greater than the parent) *OR* (maxIdx !== null *AND* right child is greater than the left one), maxIdx is rightIdx   + if maxIdx !== null, swap elements at parentIdx and maxIdx   + parentIdx = maxIdx * this.values = arr   *Arguments:* ()  *Return:* the value of the removed element |
| **NODE** | **PRIORITY QUEUE** |
| *2 properties:* val, priority | *1 property:* values (array) |
| **ENQUEUE/DEQUEUE** | |
| **enqueue instance method *O(logn)***   * create a new node * push the node into this.values array * create currIdx variable to point on the inserted value (this.length - 1) * create parentIdx variable to point on the inserted value's parent Math.floor((currIdx - 1) / 2) * loop through this.values to bubble up the inserted value   *Arguments:* (val, priority)  *Return:* this | **dequeue instance method *O(logn)*** \*compare priority of the nodes   * create parentIdx = 0, leftIdx, rightIdx, minIdx pointers + del, arr * if arr.length === 0, return undefined * swap the first and last elements in arr * pop the last element in arr * while minIdx !== null   + calculate leftIdx, rightIdx; assign minIdx = null   + if leftIdx < arr.length     - if the left child is lesser than the parent, minIdx is leftIdx   + if rightIdx < arr.length     - if (minIdx === null *AND* the right child is lesser than the parent) *OR* (minIdx !== null *AND* the right child is lesser than the left one), minIdx is rightIdx   + if minIdx !== null, swap elements at parentIdx and minIdx   + parentIdx = minIdx * this.values = arr   *Arguments:* ()  *Return:* the value of the removed element |

|  |  |
| --- | --- |
| **HASH TABLE** | |
| *1 property:* keyMap = new Array(size) \*size needs to be a prime number to avoid collisions | |
| **SET/GET** | |
| **set instance method *O(1)***   * hash the key * if the spot is empty   + create a nested array * push val in the nested array   *Arguments:* (key,val)  *Return:* this | **get instance method *O(1)***   * hash the key * if the spot contains nothing, return undefined * if the spot contains 1 value, return it * if the spot contains more than 1 value   + loop to find a pair   *Arguments:* (key)  *Return:* arr \*key-value pair/undefined |
| **KEYS/VALUES** | |
| **keys instance method *O(n)***   * loop through the hash table to return an array of its keys   *Arguments:* ()  *Return:* arr | **val instance method *O(n)***   * loop through the hash table to return an array of unique values   *Arguments:* ()  *Return:* arr |

|  |  |
| --- | --- |
| **BINARY SEARCH TREE** | **NODE** |
| *1 property:* root | *3 properties:* val, left, right |
| **INSERT/FIND** | |
| **insert instance method *O(n)***   * create a helper function (root, val)   + if root === null, return a newly created node   + else if val < root.val, call the function on the left child (result is root.left)   + else if val > root.val, call the function on the right child (result is root.right)   + eventually, return root * call the helper function on the tree root (result is this.root)   *Arguments:* (val)  *Return:* this | **find instance method *O(n)***   * create a helper function (root, val)   + if root === null, return undefined   + if root.val === val, return root   + else if val < root.val, call the function on the left child (return its result)   + else if val > root.val, call the function on the right child (return its result) * call the helper function on the tree root (return its result)   *Arguments:* (val)  *Return:* node/undefined |

|  |  |  |
| --- | --- | --- |
| **BST TRAVERSAL** | | |
| **BREADTH-FIRST SEARCH (BFS)** | | |
| **bfs instance method *O(n)***   * create a queue * create an arr to store visited values * if the tree has no root, return arr * place the root node in the queue * while queue is not empty   + dequeue a node from the queue and push its value into the arr   + if there are left/right properties on the dequeued node, add them to the queue   *Arguments:* ()  *Return:* arr | | |
| **DEPTH-FIRST SEARCH (DFS)** | | |
| **DFS PRE-ORDER** | **DFS IN-ORDER** | **DFS POST-ORDER** |
| \*get the tree structure to export (for easy duplication)  **dfsPreOrder instance method *O(n)***   * create an arr to store visited values * if the tree has no root, return arr * create a *helper function* which accepts node   + ***push the node's value into arr***   + if node has a left property, call the helper function with the left property on the node   + if node has a right property, call the helper function with the right property on the node * invoke the helper function with this.root   *Arguments:* ()  *Return:* arr | \* get nodes' values in ascending order  **dfsInOrder instance method *O(n)***   * create an arr to store visited values * if the tree has no root, return arr * create a *helper function* which accepts node   + if node has a left property, call the helper function with the left property on the node   + ***push the node's value into arr***   + if node has a right property, call the helper function with the right property on the node * invoke the helper function with this.root   *Arguments:* ()  *Return:* arr | \*commonly used with BSTs  \*\*get the nodes' values in the underlying order  **dfsPostOrder instance method *O(n)***   * create an arr to store visited values * if the tree has no root, return arr * create a *helper function* which accepts node   + if node has a left property, call the helper function with the left property on the node   + if node has a right property, call the helper function with the right property on the node   + ***push the node's value into arr*** * invoke the helper function with this.root   *Arguments:* ()  *Return:* arr |

|  |  |
| --- | --- |
| **UNWEIGHTED GRAPH** | |
| *1 property:* adjacencyList | |
| **ADD/REMOVE VERTEX** | |
| **addVertex instance method *O(1)*** (1)   * check if the vertex exists   + add a key to this.adjacencyList with the name of vertex and set its value to be an empty array   *Arguments:* (vertex)  *Return:* - | **removeVertex instance method *O(V + E)*** (4)   * check if the vertex exists   + loop through all vertex's connections     - on each connection call removeEdge function   + delete the key in the adjacency list for that vertex   *Arguments:* (vertex)  *Return:* - |
| **ADD/REMOVE EDGE** | |
| **addEdge instance method *O(1)\**** (2)   * check if both vertices exist and do not have connection established yet   + push vertex2 into this.adjacencyList[vertex1]   + push vertex1 into this.adjacencyList[vertex2]   *Arguments:* (vertex1,vertex2)  *Return:* - | **removeEdge instance method *O(E)*** \*use filter method (3)   * check if both vertices exist and have connection established   + remove vertex2 from this.adjacencyList[vertex1]   + remove vertex1 from this.adjacencyList[vertex2]   *Arguments:* (vertex1,vertex2)  *Return:* - |

|  |
| --- |
| **GRAPH TRAVERSAL** |
| **BREADTH-FIRST SEARCH (BFS)** |
| **bfs instance method *O(V + E)*** \*where V - number of vertices; E - number of edges   * create an arr to store visited vertices * create visited object and queue to track vertices * add the start vertex to queue and mark it as visited * while queue is not empty   + dequeue next vertex from queue   + push the dequeued vertex into arr   + enqueue all adjacent vertices *that have not been visited* into queue and mark them as visited   *Arguments:* (start)  *Return:* arr |
| **DEPTH-FIRST SEARCH (DFS) - RECURSION** |
| **dfsRecursive instance method *O(V + E)*** \*where V - number of vertices; E - number of edges   * create arr to store visited vertices * create visited object to track visited vertices * create a *helper function* which accepts vtx   + if vtx is empty/not valid, return   + place vtx into arr and visited object   + loop over all the values in this.adjacencyList[vtx]     - if a vertex has not been visited, recursively invoke the helper function on the vertex * invoke the helper function with start   *Arguments:* (start)  *Return:* arr |
| **DEPTH-FIRST SEARCH (DFS) - ITERATION** |
| **dfsIterative instance method *O(V + E)*** \*where V - number of vertices; E - number of edges   * create an arr to store visited vertices * create visited object and stack to track vertices * add the start vertex to stack and mark it as visited * while stack is not empty   + pop next vertex from stack   + push the popped vertex into arr   + push all adjacent vertices *that have not been visited* into stack and mark them as visited   *Arguments:* (start)  *Return:* arr |

|  |  |
| --- | --- |
| **WEIGHTED GRAPH** | |
| *1 property:* adjacencyList | |
| **ADD VERTEX/EDGE** | |
| **addVertex instance method *O(1)***   * check if the vertex exists   + add a key to this.adjacencyList with the name of the vertex and set its value to be an empty array   *Arguments:* (vertex)  *Return:* - | **addEdge instance method *O(1)\****   * check if both vertices exist and do not have connection established yet   + push {val: vertex2, weight} into this.adjacencyList[vertex1]   + push {val: vertex1, weight} into this.adjacencyList[vertex2]   *Arguments:* (vertex1,vertex2,weight)  *Return:* - |
| **DIJKSTRA’S ALGORITHM** | |
| **dijkstraAlgorithm instance method *O(V + E \* logV)***   * create distances object to store shortest distances from start for each vertex (copy keys from this.adjacencyList; set the values to be Infinity/0) * enqueue all vertices into priorityQueue * create previous object and set each key to be every vertex in this.adjacencyList with a value of null * while priorityQueue is not empty   + dequeue vtx from priorityQueue   + if vtx === end, return   + else, loop through this.adjacencyList[vtx]     - calculate distance to vtx from start     - if distance is less than one in distances object:       * update distances object with new lower distance       * update previous object to contain vtx       * enqueue vtx with a priority of the total distance from start   *Arguments:* (start,end)  *Return:* previous | |