**Sorting**

1. **Insertion sort**
2. Algorithm

A = [5, 2, 4, 6, 1, 3]

// Loop from second position, and compare to sorted array from 0 to current position, if smaller, then loop until find the position that current item is larger or first position. Insert item to the sorted array

function insertionSort(a) {

for (var i = 1; i < a.length; i++) {

var key = a[i]

var j = i - 1

while (j >= 0 && a[j] > key) {

a[j + 1] = a[j]

j = j - 1

}

a[j + 1] = key

}

return a

}

console.log(insertionSort(A))

1. Analysis

The time taken by an algorithm grows with the size of the input

Worst case running time of θ(n2)

1. **Merge sort ( divide and conquer)**
2. algorithm

// divide

A = [5, 2, 4, 7, 1, 3, 2, 6]

function mergeSort(a) {

if(a.length < 2) {

return a

}

var middle = Math.floor(a.length/2)

var leftArr = a.slice(0, middle)

var rightArr = a.slice(middle, a.length)

console.log(leftArr)

console.log(rightArr)

return merge(mergeSort(leftArr), mergeSort(rightArr))

}

//Conquere

function merge(L, R) {

var result = []

while(L.length && R.length) {

if(L[0] <= R[0]) {

result.push(L.shift())

} else {

result.push(R.shift())

}

}

while(L.length) {

result.push(L.shift())

}

while(R.length) {

result.push(R.shift())

}

return result

}

console.log(mergeSort(A))

1. analysis

Worst case running time of θ(nlgn)

1. **Bubble sort**

Simply swapping 2 elements of array. It’s no efficiency

1. **Heap sort**
2. Algorithm

* Build max\_heap\_heap. ( create a function to maintain max-heap properties)
* Then get a max item at position 0, swap position to the last pos and decremeting the array size
* Repeat the process for the size n-2 down to 1. We get the sorted array

var arr = [ 9, 10, 2, 1, 5, 4, 3, 6, 8, 7, 13 ];

var arraylength = 0

function max\_heapify(a, i) {

var left = i\*2 + 1;

var right = i\*2 + 2;

var largest = i;

if (left < arraylength && a[left] > a[largest]) {

largest = left;

}

if (right < arraylength && a[right] > a[largest]) {

largest = right;

}

if (i != largest) {

swap(a, i, largest);

max\_heapify(a, largest)

}

}

function swap(input, index\_A, index\_B) {

var temp = input[index\_A];

input[index\_A] = input[index\_B];

input[index\_B] = temp;

}

function build\_max\_heap(a) {

arraylength = a.length

for(i = Math.floor(a.length/2); i >= 0; i--) {

max\_heapify(a, i)

}

}

function heapSort(a) {

build\_max\_heap(a)

for (var i = a.length - 1; i > 0; i--) {

swap(a, 0, i);

arraylength--;

max\_heapify(a, 0);

}

}

heapSort(arr);

console.log(arr)

1. Analysis

The HEAPSORT procedure, which runs in O(nlgn)

1. **QuickSort**
2. Algorithm

Divide: divide the array into 2 subarray a[p, q-1] and a[q, r] such that element of a[p, q – 1] is less than or equal to a[q], and element of a[q + 1, r] greater than a[q]. Compute the index q

Conquer: repeat sort 2 subarrays by recursive call quicksort

function quickSort(a, p, r) {

if(p < r) {

q = partition(a, p, r)

quickSort(a,p, q - 1)

quickSort(a,q + 1, r)

}

}

function partition(a, p ,r) {

var x = a[r]

var i = p - 1

for (var j = p; j < r; j++) {

if(a[j] <= x){

i = i + 1

swap(a, i, j)

}

}

swap(a, i+ 1, r)

return i + 1

}

quickSort(arr, 0, arr.length - 1)

1. Analysis

Worst-case partitioning : when 1 subarray with n-1 elements and one with 0 elements. The partition cost θ(n) time. With array size 0, the cost is T(0) = θ(1). So recursive running time is

T(n) = T(n-1) + T(0) + θ(n)

Solution is T(n) = θ(n2)

Best-case partitioning: 2 subarrays and each of size no more than n/2

T(n) = 2T(n/2) + θ(n)

Solution is T(n) = θ(nlgn)

**Data Structure**

1. Stacks and queues
2. Stacks: last in, first out.
3. Queues: first in, first out.
4. Linked list

It’s data structre in which the objects are arranged in a linear order. Linked list is a sequence of links which contains items. Each link contains a connection to another link.

1. Singly node

function LinkedList(){

this.head = null;

}

LinkedList.prototype.push= function(val) {

var node = {

value: val,

next: null

}

if(!this.head) {

this.head = node

} else{

var current = this.head

while(current.next) {

current = current.next

}

current.next = node

}

}

LinkedList.prototype.printList = function() {

if(!this.head) {

console.log('empty list')

} else{

var current = this.head

while(current) {

console.log(current.value)

current = current.next

}

}

}

LinkedList.prototype.insertAfter = function(pre\_node, new\_data) {

if(pre\_node === null) {

console.log('given previous node cannot be null')

return

} else{

var node = {

value = new\_data

next = pre\_node.next

}

pre\_node.next = node

}

}

LinkedList.prototype.insertAfter = function(pre\_node, new\_data) {

if(pre\_node === null) {

console.log('given previous node cannot be null')

return

} else{

var node = {

value = new\_data,

next = pre\_node.next

}

pre\_node.next = node

}

}

LinkedList.prototype.insertFirst = function (new\_data) {

var node = {

value = new\_data,

next = null

}

if(!this.head) {

this.head = node

return

}else {

node.next = this.head

this.head = node

}

}

LinkedList.prototype.deleteNode = function (val) {

var current = this.head;

//case-1

if(current.value == val){

this.head = current.next;

}

else{

var previous = current;

while(current.next){

//case-3

if(current.value == val){

previous.next = current.next;

break;

}

previous = current;

current = current.next;

}

//case -2

if(current.value == val){

previous.next == null;

}

}

}

var lls = new LinkedList();

lls.push(2)

lls.push(3)

lls.push(4)

//traver list

lls.printList()

1. Doubly node

A doubly linked list contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.

function DoubleLinkedList() {

this.head = null

}

DoubleLinkedList.prototype.append = function(val){

var node = {

value: val,

pre: null,

next: null

}

if(!this.head) {

this.head = node

} else {

var current = this.head

while(current && current.next) {

current = current.next

}

current.next = node

node.pre = current

}

}

DoubleLinkedList.prototype.push = function(val){

var node = {

value: val,

pre: null,

next: null

}

if(!this.head) {

this.head = node

} else {

var current = this.head

current.pre = node

node.next = current

this.head = node

}

}

DoubleLinkedList.prototype.insertAfter = function(pre\_node, val){

var node = {

value: val,

pre: null,

next: null

}

if(pre\_node === null) {

console.log('not a node')

}else {

node.next = pre\_node.next

node.pre = pre\_node

pre\_node.next = node

if(node.next != null) {

node.next.pre = node

}

}

}

DoubleLinkedList.prototype.printList = function(){

if(!this.head) {

console.log('empty list')

} else {

var current = this.head

while(current) {

console.log(current.pre)

current = current.next

}

}

}

var dll = new DoubleLinkedList();

dll.append(2);

dll.append(3);

dll.append(4);

dll.append(5);

dll.printList()

1. Rooted tree (binary tree)

In binary tree, each node has pointer parent, left, right to pointer to other nodes respectively. If x.p = nil, then x is the root. If node x has no left child or right child then x.left or x.right = nil. The root of the entire tree T is pointed to by the attribute T.root. If T.root = nil, then tree empty

1. **Hash Table**

A hash table is a data structure that maps keys to values for highly efficient lookup. In a very simple implementation of hash table, we use an array of linked list and a hash code function.

To insert a key and value:

* Compute the key’s hash code
* Then, map the hash code to an index in the array. This could be done with something like hash(key) % array\_length.
* At this index, there is a llinked list of key and values. Store the key and value in this index.

1. **Linked list:**

A linked list is a data structure that represents a sequence of nodes. In a singly linked list, each node points to the next node in the linked list. A doubly linked list gives each node pointers to both the next and previous node.

Compare to array:

* Pros: you can add and remove items from the beginning of the list in constant time.
* Cons: a linked list does not provide constant time access to a particular “index” within the list.

Remember to use 2 pointers if the question is to find the middle of linked list or to travel the whole list.

1. **Stacks and queues:**
2. Stack: is a data structure that stores the data in a stack rather than in an array.

A stack uses LIFO (last-in first out) ordering.

Unlike an array, a stack does not offer constant-time access to the ith item. However, it does allow constant-time adds and removes, as it doesn’t require shifting elements around.

* Implementing stack:

var stack = []

stack.push

stack.pop

or using object

var Stack = function () {

this.count = 0

this.storage = {}

}

Stack.prototype.push(val) {

this.storage.[this.count]

this.count ++

}

Stack.prototype.pop() {

If(this.count === 0 {  
 return -1

} else {

Var I = this.storage.[this.count]

Delete this.storage.[this.count]

this.count –

return i

}

}

Stack.prototype.isEmpty