ELEC 278 FINAL REVIEW

Dec 2022

by Noah Serhan

Big Oh

→ For worst case scenario

Function

$$f(n) = O(n) = O(g(n))$$

where

- $f(n)$ is the actual function that determines colour
- O states that it is in big Oh notation and is simplified
- n is the amount of data inputted in the algorithm
- $g(n)$ is the simplified function for runtime without a constant in front.

(e.g. $f(n) = O(2n)$ → $g(n) = O(n)$ no constant)

Finding the big Oh function

Say $f(n) = 6n^2 + 3n + 2$

When we increase n to a large number, both the $+2$ and the $+3n$ have a much smaller effect on the function (compared to $6n^2$)

↑ low impact too

∴ we can say that

$$f(n) = O(g(n)) = \underline{O(n^2)}$$

Types of big Oh

O(1) – No matter how big the array / data structure / ... the algo. will perform the operation at the same runtime.

e.g. Prepending a linked list. No matter if there are 2 or 200 links already in the list, the prepend operation adds another at the head.

O(n) – The runtime is directly proportional to the size of the data structure.

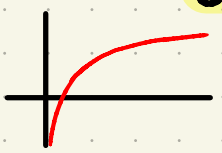
e.g. Traversing through a linked list. The time it takes to traverse is directly proportional to the size of the linked list. This also applies to for loops

`for(i=0; i<n, i++)` → depends on n

O(n^k) – O(n^k) runtime is proportional to the size of the array to the power k (so, $O(n^2)$ or $O(n^3)$)

e.g. A nested for loop is an example as it runs the for loop n times within another for loop. Therefore giving a runtime proportional to $n \times n = n^2$ or $O(n^2)$.

O(log(n)) – O(log(n)) says that if the input increases by a certain amount, then the runtime will increase a whole lot LESS than that of the data input increase.



e.g. Binary search: Select a lower and upper bound – chose the value directly in the middle and compare it to the one we are searching for – if it is bigger/smaller, use the appropriate half of the data and repeat until found.

$O(n \log(n))$ - This is a combination of $O(n)$ and $O(\log(n))$ runtimes.

e.g. Running binary search within a for loop.

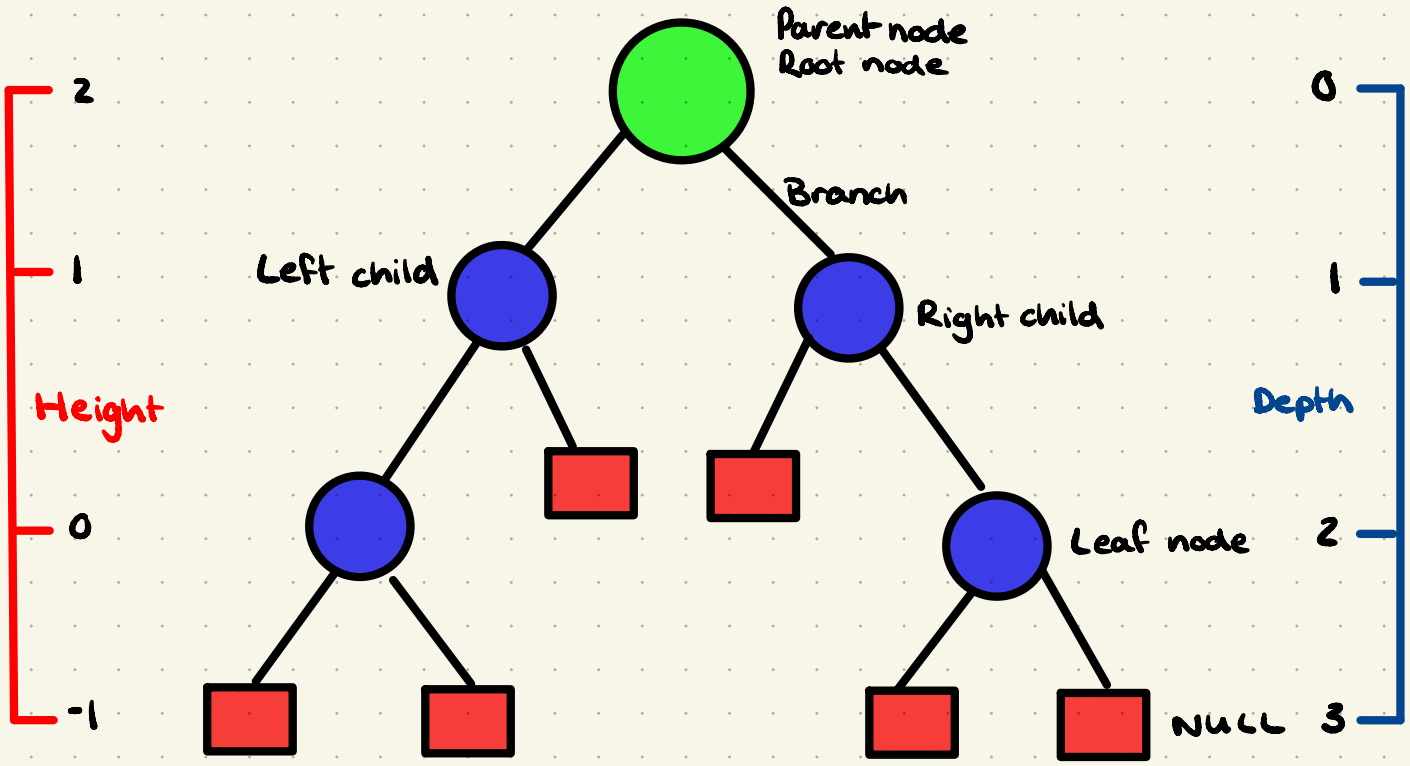
Linked Lists

Check insertion and removal

Check stacks and queues

Circular linked + Circular doubly linked.

Trees



Internal node: a node with one or more child nodes

Internal path length: Sum of the depths of all of the internal nodes.

External node: a node that has no child nodes

External path length: Sum of the depths of all of the external nodes (leaf).

Traversals

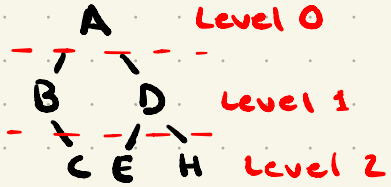
Recursively going right / left

Preorder – <root> <Left> <right>

Inorder – <left> <root> <right>

Postorder – <left> <right> <root>

Depth first / breadth order:



A, B, D, C, E, H

BSTs

- Max 2 children per node
- Left child value < Parent value < Right child value

```
Struct Node {
        int data;
        Struct Node *Left, *right;
}
```

Inserting

```
Struct Node *insertBST (struct Node *root, int data){
        if (!root){ // found the place to insert new data
            return createNode(data);
        }
        if (data < root->data) {
            root->Left = insertBST (root->Left, data);
        }
        else{
            root->right = insertBST(root->right, data);
        }
        return root; // sends each subtree parent up recursion chain
}
```

Deleting (3 cases)

- Node is replaced with leftmost node of right subtree

```
Struct Node *deleteBST (struct Node *root, int data ) {
        if(!root){ // node not in tree
            return root;
        }
        if ( data < root->data) {
            root->Left = deleteBST (root->Left, data);
        }
        if( data > root->data) {
            root->right = deleteBST (root->right, data);
        }
        else { // this is the node to be deleted
```

```
        //node with only ONE or NO children
        if (root -> left != NULL) {
Normal  ->  struct node *temp = root->right;
node            free(root);
                return temp;
        }
        else if (root->left != NULL) {
                struct node *temp = root->left;
                free(root);
                return temp;
        }
        //with two children, replace with leftmost node in right ST.
        Struct Node *replacer = root->right
        while(replacer->left != NULL) { // gets smallest node in right.
                replacer = replacer->left;
        }
    }
}
```

★ Creating a new node

```
Struct node *new = (struct node *) malloc (sizeof (struct node));
new -> data = data
new -> right = NULL
   :
```

AVL Trees

- Similar to binary search tree — but reduces worst-case search scenarios.
- The difference in height of the two subtrees of a parent node cannot differ by more than one.
- Rebalances every time a node is added

Noah's Final Review Package 7

#ifndef max
#define max(a,b) ((a>b)?(a:b))
#endif

Calculating Height

```
int calcHeight(nodeType *root) {
    if(root == NULL) {
        return 0;
    }
    else {
        return max(calcHeight(root->left), calcHeight(root->right));
    }
}
```

+1

don't forget +1

Balancing Factor

```
int calcBalanceFactor(nodeType *root) {
    if(root == NULL) return 0;
    else {
        return(calcHeight(root->left) - calcHeight(root->right));
    }
}
```

return > 0 then left > right

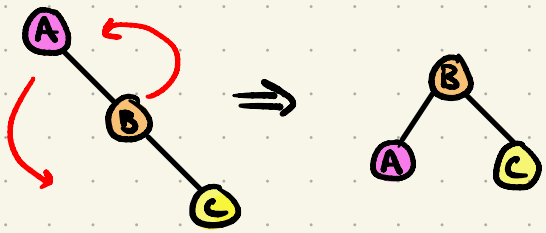
else if return < 0 then right < left

Rotations

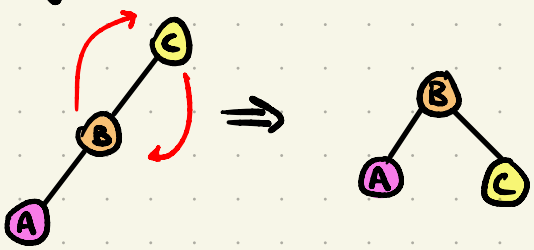
```
nodeType *AVLBalance(nodeType *root) {
    nodeType *temp = root;
    int BF = calcBalanceFactor(temp);
    if(root != NULL) {
        if(BF > 1) {
            int LBF = calcBalanceFactor(temp->left);
            printf("Leftchild balance factor: %d \n", LBF);
            if ((LBF > 0) && (LBF < 2)) {
                temp = LLRotation(temp);
            }
            else if ((LBF > -2) && (LBF < 0)) {
                temp = LRRotation(temp);
            }
            else if (LBF != 0) {
                temp->left = AVLBalance(temp->left);
            }
        }
        else if (BF < -1) {
            int RBF = calcBalanceFactor(temp->right);
```

```
                printf("Rightchild balance factor: %d \n", RBF);
                if(RBF<0 && RBF>-2){
                        temp = RRRotation(temp);
                }
                else if(RBF<2 && RBF>0){
                        temp = RLRotation(temp);
                }
                else if(RBF != 0){
                        temp->right = AVLBalance(temp->right);
                }
        }
        else{
                temp->left = AVLBalance(temp->left);
                temp->right = AVLBalance(temp->right);
        }
    }
return(temp);
}
```

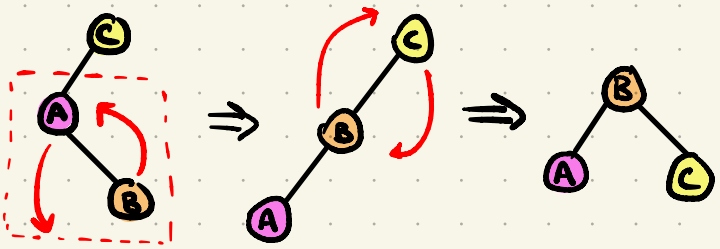
Left Rotation



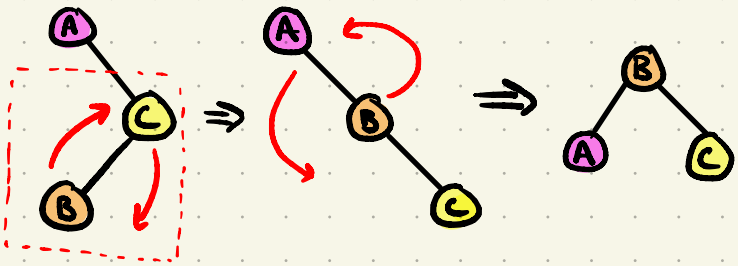
Right Rotation



Left - Right Rotation



Right - Left Rotation



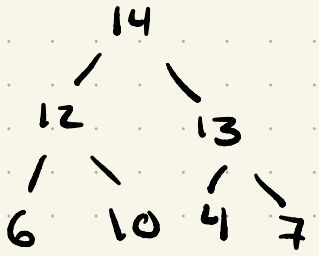
- Inserting into an AVL tree is always followed with a check and a rebalance if needed.

Heaps

- Like a BST where there are two children per node.
- Heaps are always balanced (height diff < ±1)
- Sorted from top to bottom.

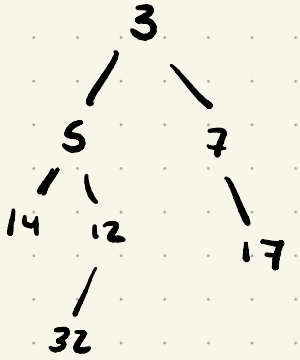
Max Heap

↳ All of the parents' nodes are larger than the childrens'



Min Heap

↳ All of the parents' nodes are smaller than the childrens'



Push

```
void pushHeap(int arr[], int last, int item) {
    int currPos, parentPos;
    currPos = last;
    parentPos = (currPos-1)/2;

    while (currPos != 0) {
        if (item > arr[parentPos]){
            arr[currPos] = arr[parentPos];
            currPos = parentPos;
            parentPos = (currPos-1)/2;
        }
        else break;
    }
```

```
    arr[currPos] = item;
}
```

Pop

```
void popHeap(int arr[], int last) {
    int temp = arr[0];
    arr[0] = arr[last-1];
    arr[last-1] = temp;
    adjustHeap(arr, 0, last-1);
}
```

Adjust Heap

```
void adjustHeap(int arr[], int first, int last) {
    int currPos, childPos;
    int target;
    currPos = first;
    target = arr[first];
    childPos = 2*currPos + 1;
    while (childPos < last) {
        if ((childPos + 1 < last) && arr[childPos]+1 > arr[childPos]) {
            childPos = childPos + 1;
        }
        if (arr[childPos] > target) {
            arr[currPos] = arr[childPos];
            currPos = childPos;
            childPos = 2*currPos + 1;
        }
        else
            break;
    }
    arr[currPos] = target;
}
```

Hashing

- Storing data at a unique index (fast access)
- Arrays used as storage for the keys (which point to the values)

Collisions

→ Hashing function returns the same index for multiple keys.

→ Value at that index needs to be retained, but new value also needs to be inserted

Solutions

Limited by the finite storage of the hash table

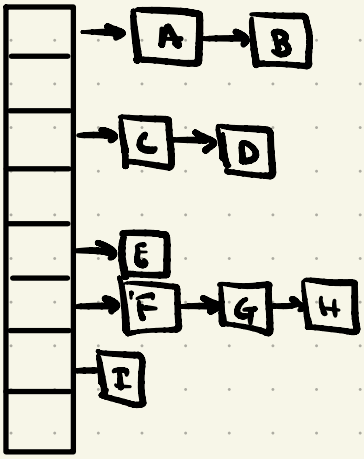
1. **Probing**: Searching through indexes iteratively until a free spot is found.

   e.g. Skip 5 indexes and check if the 6th can be used ... if not, skip another 5 and try again.

2. **Re-Hashing**: Like Probing, but the value at the index where the collision occured is used to hash a second time

most efficient way.

3. **Chaining**: Using a linked list to store the hashed data. Allows new data to be added into the list by linking it to the correct node (start/end)



Adding nodes to the end of others at a specific index.

Objectives of hash function:

→ Minimize collisions

→ Uniform distribution of hash values

→ Easy to calculate

→ Resolve any collisions

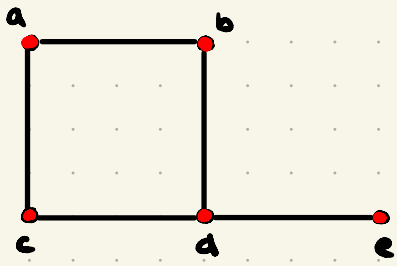
Graphs

Representation of objects that are connected in some way. Objects are nodes or vertices and are connected by edges.

Graphs are represented by a pair of sets V and E

V = (a, b, c, d, e) - vertices

E = (ab, ac, bd, cd, de) - edges



Adjacent: vertices connected by an edge

Path: Sequence of edges connecting two vertices.

Directed graph: Graph with one-way edges.

e.g a ⟶ c ac is an edge, but ca is not.

Weighted graph: Edges have values associated with them.

Steps to create a graph:

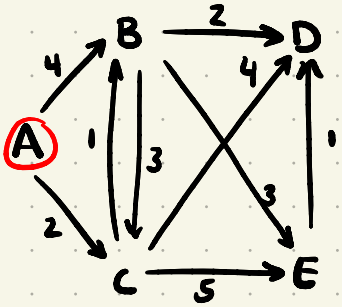
1. Define the graph structure
2. Initialize the graph structure
3. Add vertices to the graph
4. Add edges to the vertices

```
typedef struct {
    int vertex;
    struct Edge* next;
} Edge;

#define maxVertices 100

typedef struct {
    Edge *edges[maxVertices];
    int numVertices;
} Graph;

Graph *createGraph(int numVertices)
    Graph * graph = (Graph *) malloc(sizeof(Graph));
    graph -> numVertices = numVertices;
    for (int i = 0; i < numVertices; i++) {
        graph -> edges[i] = NULL;
    }
    return graph;
}

void printGraph(Graph *graph) {
    for (int i = 0; i < graph -> numVertices; i++) {
        Edge *edge = graph -> edges[i];
        printf("Vertex %d is connected to:", i);
        while(edge != NULL) {
            printf(" %d", edge -> vertex);
            edge = edge -> next;
        }
        printf("\n");
    }
}
```

```
void addEdge(Graph *graph, int src, int dest){
    Edge *edge = (Edge*) malloc(sizeof(Graph));
    edge->vertex = dest;
    edge->next = graph->edges[src];
    graph->edge[src];
}

int main(...
```

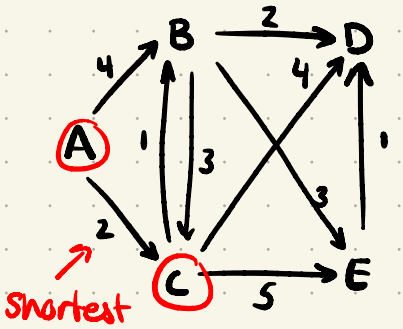
Dijkstra's Algorithm

1. Pick a starting node and create a table with distances.
   Starting node is 0 and all of the others are ∞ (aka unvisited)
2. Travel to the next unvisited node with the smallest cost.
   Update the table ONLY if there are edges leaving the vertex.
3. Repeat for all UNVISITED vertices. Remove visited ones from the unvisited array / list.



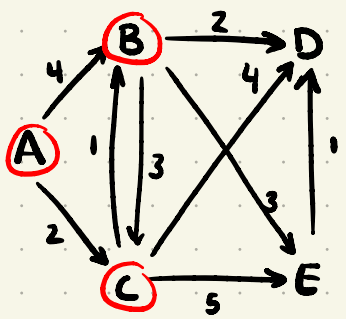
Unvisited = {~~A~~, B, C, D, E}

A: 0 ✓
B: ∞
C: ∞
D: ∞
E: ∞



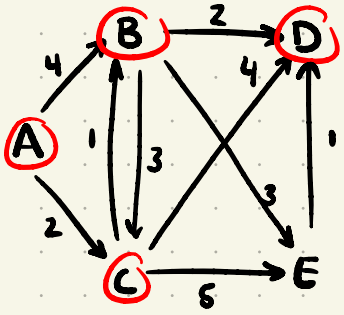
Unvisited = {~~A~~, B, ~~C~~, D, E}

A: 0 ✓
B: 4
C: 2 ✓
D: ∞
E: ∞



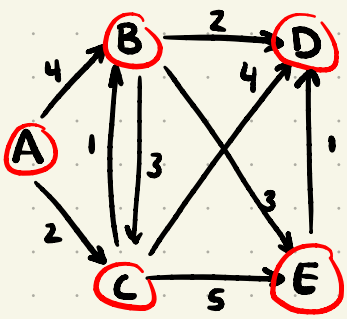
Unvisited = {~~A~~, ~~B~~, ~~C~~, D, E}

A: 0 ✓
B: 2+1 = 3 ✓
C: 2 ✓
D: 2+4 = 6
E: 2+5 = 7



Unvisited = {~~A~~, ~~B~~, ~~C~~, ~~D~~, E}

A: 0 ✓
B: 3 ✓
C: 2 ✓
D: 3+2 = 5 ✓
E: 3+3 = 6

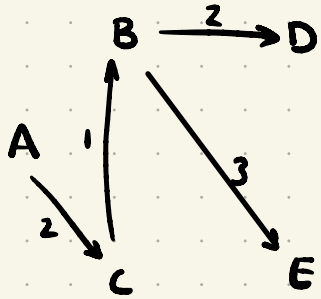


Unvisited = {~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~E~~}

A: 0 ✓
B: 3 ✓
C: 2 ✓
D: 5 ✓
E: 3+3 = 6 ✓

B → E (Not D... no edges leaving D)

Shortest Path from A



Unvisited = {~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~E~~}

A: 0 ✓
B: 3 ✓
C: 2 ✓
D: 5 ✓
E: 6 ✓

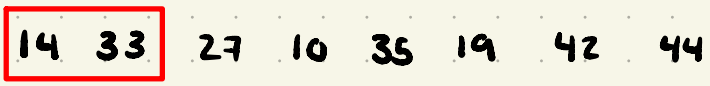
```
void Dijkstra(int Graph[MAX][MAX], int n, int start) {
    int cost[MAX][MAX], distance[MAX], pred[MAX];
    int visited[MAX], count, mindistance, nextnode, i, j;
    // Creating cost matrix
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (Graph[i][j] == 0)
                cost[i][j] = INFINITY;
            else
                cost[i][j] = Graph[i][j];
    for (i = 0; i < n; i++) {
        distance[i] = cost[start][i];
        pred[i] = start;
        visited[i] = 0;
    }
    distance[start] = 0;
    visited[start] = 1;
    count = 1;

    while (count < n - 1) {
        mindistance = INFINITY;
        for (i = 0; i < n; i++)
            if (distance[i] < mindistance && !visited[i]) {
                mindistance = distance[i];
                nextnode = i;
            }
        visited[nextnode] = 1;
```

```
        for (i = 0; i<n; i++)
            if(!visited[i])
                if (mindistance + cost[nextnode][i] < distance[i]) {
                    distance[i] = mindistance + cost[nextnode][i];
                    pred[i] = nextnode;
                }
            count++;
        }
    }
```


Sorting

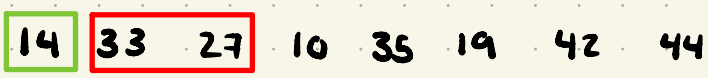
Insertion Sort (two array sections: sorted & unsorted)

- Compare the first two elements



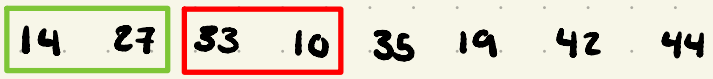
sorted ✓

- If sorted, add the first element to the sorted "sub-array"



x unsorted

- 33 are unsorted. Swap them and, again add the first to the sorted sub-list while ALSO checking to make sure that the sub-list is still sorted (which for 14-27 it is)



sorted ✓ x unsorted

Code implementation:

```
void insertionSort ( int *array , int n){
    for(int i = 1 ; i < n ; i++) {
        for( int j=i ; j>0 && (array[j-1] > array[j]);j--){
            swap(array[j], array[j-1]) ;
        }
    }
}
```

only if [j-1] > [j]

i = 1 since we already have array[0] sorted (alone)

Bubble Sorting

- Start from the bottom of the array and swap adjacent array elements if they are not properly sorted.
- Check all of the array elements once
- Start the sort again from the start
- End when a whole pass was made without any swaps.

Code implementation:

```
void bubbleSort(int *arr, int size){
    for(int i=0; i<=size-2; i++){
        int swapped = 0; // keeps track of swaps
        for(int j = size-1; j<=i+1; j--) {
                    goes from bottom to top
            if(nums[j] < nums[j-1]) {
                swap(j, j-1, nums);
                swapped = 1; // means swap occured
            }
        }
        if(swapped == 0) {
            break;
        }
    }
}
```

Quicksort

- Pick a pivot (arbitrarily or middle or last)
- Assign a left and right selector
- Swap the locations of the pivot and whatever is in the right selector, then move the selector over one.
- Check the value of left and right and move them both until the right finds a value ≤ pivot and left finds a value ≥ pivot.
- Swap the values of the selectors if both left ≥ pivot and right ≤ pivot.
- Eventually right and left will reach the same index.
- Swap the pivot with that value IF the pivot is smaller than the index value. (If it is greater, then it is already in the right spot.)

(Repeat recursively on the left and right)

Code implementation:

```
int partition(int *arr, int left, int right, int pivot) {
    while(1) {
        while(arr[left] ≤ pivot){
            left ++; //moves right until value ≤ pivot found
        }
        while(arr[right] ≥ pivot){
            right --;
        }
        if (left ≥ right) {
            break; //found pivot
        }
        swap(arr, left, right);
    }
    swap(arr, left, right);
    return left;
}
```

```
void quickSort(int *arr, int size, int left , int right){
    if(right-left <=0){
        return; //maximum partitionning
    }
    int pivot = arr[right];
    int partitionIndex = partition(arr, left, right, pivot);
    quickSort(arr, size, left, partitionIndex-1 );
    quickSort(arr, size, partitionIndex, right);
}
```


Merge Sort

- Recursively divide the array in half until there are many sub-arrays of length 1.
- Each element is compared with an adjacent element and they are merged into subarrays of length 2
- Continue until the two last arrays are merged into the full sorted array.

Code implementation:

```
void merge( int *arr, int left, int right, int mid){
    int p1 = left , p2 = mid+1; //indices used to merge
    int mergeIndex = left;
    while (p1 <= mid && p2 ≤ right){
        if (arr[p1] < arr[p2]){
            arr[mergeIndex] = arr[p1];
            p1++;
        }
        else{
            arr[mergeIndex] = arr[p2];
            p2++;
        }
        mergeIndex++;
```

```
    }
    while(p1 <= mid){    //one of the arrays has leftovers
        arr[mergeIndex] = arr[p1];
        p1++;
        mergeIndex++;
    }
    while(p2 <= mid){
        arr[mergeIndex] = arr[p2];
        p2++;
        mergeIndex++;
    }

}
```

```
void mergeSort(int *arr, int left, int right){
    if (left < right){
        int mid = (left+right)/2;
        mergeSort(left, mid);
        mergeSort(mid+1, right);
        merge(arr, left, right, mid);
    }
}
```


Heap Sort

- Heap is made from the original array
- The heap is extracted from and put in successive positions in the sorted array.

Code implementation

```
void maxHeapify(int *arr, int size, int rootIndex){
    int largest = rootIndex;
    int left = 2*rootIndex+1;
    int right = 2*rootIndex+2;
    //if left child in heap and larger than root, set largest to left child
```

```
    if (arr[left] > arr[largest] && left < size) {
        largest = left;
    }
    if (arr[right] > arr[largest] && right < size) {
        largest = right;
    }
    if (largest != rootIndex) { // Largest index not root
        swap(arr[rootIndex], arr[largest]);
        maxHeapify(arr, size, largest);
    }
}
```

```
void heapSort(int *arr, int size) {
    for(int i = size/2 - 1; i >= 0; i--){
        maxHeapify(arr, size, i);
    }
    for(int i = n-1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        maxHeapify(arr, i, 0);
    }
}
```

Sorting Algo Complexity

Algorithm	Best	Average	Worst
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Quick	$O(n\log n)$	$O(n\log n)$	$O(n^2)$
Merge	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Heap	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Radix	$O(n^k)$	$O(n^k)$	$O(n^k)$

good with low amount of items

same

same