Ultimate Final

Big "O" Notation

- Big "O", how quickly the runtime grows as the input gets larger.
- Big "O" is a tool to analyze the cost of an algorithm primarily on CPU time, memory

O(1) Same amount of time regardless of input size

O(n) Dependent on size of input (printAll) (linear time complex)

O(n^2) n^2 times dependant (2 for loops)

↳ Relation between input & runtime.

Different functions will have different rates of growth
Computation times are different

$$\log_2 x = \frac{\log_{10} x}{\log_{10} 2}$$

Big O function

n : data size	O(1)	O (LOG2(N))	O(N)	O(N^2)
10	CONST	4	10	1.E+02
100	CONST	7	100	1.E+04
1,000	CONST	10	1,000	1.E+06
10,000	CONST	14	10,000	1.E+08
100,000	CONST	17	100,000	1.E+10
1,000,000	CONST	20	1,000,000	1.E+12

Searching

We want to do optimized searching algorithm

1. Sequential (linear) Seach
2. Binary Search.

Sequential Search (go through all elements)

⇒ Easy to implement & understand

⇒ But, is a Big O (N) complexity

How to improve? Sort the array.

Binary Search Compare with middle point. Continue with either side.

⍟ Binary search pseudo-code

```
set first = 0, last = N

                                                {
      set mid = (first + last)/2

      compare search element to element at mid
      if match
          return index
      else if search element is less than term at mid
          set last = mid-1  //Focus on higher order
      else
          set first = mid + 1 //Focus on lower order

      }
   return -1 if no match
```

Change first/last to the midpoint to change the "search radius"

// mid − 1 or mid + 1 to minimize the complexity of search

Binary Search with Recursion

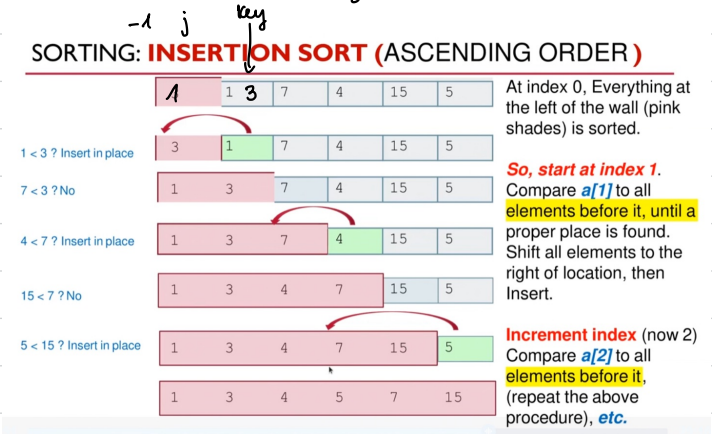
Need to put first & last. Else, need helper.

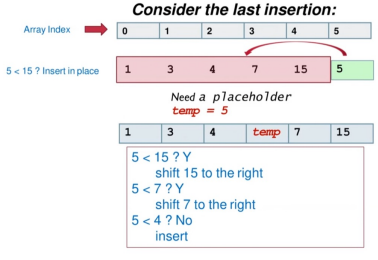
```
int binarySearchRec(int list[], int leftIndex, int rightIndex, int key) {
    if (leftIndex > rightIndex) {
        return -1;
    }

    int middle = (leftIndex + rightIndex) / 2;

    if (list[middle] == key) {
        return middle;
    }

    if (key < list[middle]) {
        // search left subarray
        return binarySearchRec(list, leftIndex, middle - 1, key);
    } else {
        // search right subarray
        return binarySearchRec(list, middle + 1, rightIndex, key);
    }
}
```

⇒ Binary Search is Big O ($\log_2 n$) complexity

Insertion Sort

Insert into an already sorted list, in correct place





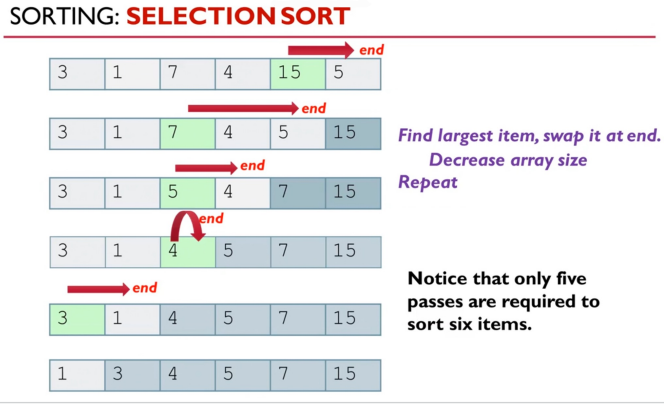

```
void insertionSort (int * list, int size) {
    for (int i = 1; i < size; i++) {
        int item = list[i];

                                        // ascending order
        for (int j = i - 1; j >= 0 && list[j] > item; j--) {
            list[j+1] = list[j];        // shifting to right
        }

        list[j+1] = item;               // inserting
    }
}
```

+) $O(n^2)$ when array is reversed
+) If already sorted, $O(n)$

} Good for almost sorted or small amounts.

Selection Sort

⇒ Start at head/tail, look the entire list to find the smallest / biggest. Swap position.




```
void selectionSort (int list[], int size) {
    for (int i = 0; i < size; i++) {          // picking locations
        int smallest = list[i];

        for (int j = i+1; j < size; j++) {    // finding which position to be at the location
            if (list[j] < smallest) {
                smallest = list[j];
                index = j;
            }
        }
        int temp = list[i];
        list[i] = list[smallest];             // swapping
        list[smallest] = tmp;
    }
}
```

⇒ Very bad. ($O(n^2)$ all the time) (good for only subsection)

Bubble Sort

Compare 2 adjacent values. After one pass, the largest value will be at the bottom limit ⇒ We change the limit down, then pass again.
⇒ After size times, we will complete the sort.

```
void bubbleSort (int * list, int size) {
    for (int i = 0; i < size; i++) {          // size times.
        for (int j = 0; j < size - i - 1; j++) {
            if (list[j] > list[j+1]) {
                // swap
                temp = list[j];
                list[j] = list[j+1];
                list[j+1] = temp;
            }
        }
    }
}
```

the bound for j will shrink as i grows, because last number is already sorted.

+) least efficient ($O(n^2)$)

Quick Sort ⇒ Good for more than 20 elements. Worst case $O(n^2)$, average $O(\log_2 n)$

SORTING: QUICK SORT

```
void quickSortHelper(int list[], int low, int high) {

     if (low < high) { // Base case ... checking the range
          int left = low; // copy org, index as we need to travese the array
          int right = high;
          int pivot = list[low]; // pick a index 0 content as the pivot

          // PARTITION PHASE
          // Goal: to place the pivot in its right place
          // ie. all larger than P on the right, all less than P on the left
          while (left < right){
               while (list[right] >= pivot && left < right)
                    right--;

               list[left] = list[right];

               while (list[left] <= pivot && left < right)
                    left++;

               list[right] = list[left];

          }

          list[left] = pivot; // Here marker left = marker right so we can say list[right] = pivot

          // Recursive SORT PHASE
          quickSortHelper(list,low,left-1);
          quickSortHelper(list,right+1,high);
     }
// COLLECT PHASE
return;
}
```

set the low & high indices Pick a Pivot.

Check the right of pivot side for items < pivot with range check

Main while loop runs till lower and higher indices are equal.

Move smaller item from R to L

Check the left of pivot side for items > pivot with range check

Move larger item from L to R

HERE partitioning for each recursive run ends. Pivot is at the right place

Tree

Unlike linked lists, trees store data in a hierarchical manner (data matters)

- root, parent, child, leaf (no children)
- No circuits

Binary Tree

- # children is 0, 1, or 2. (left / right)
- Unbalanced tree: more nodes on one side than another
- Application: File system, searching.

Binary Search Tree

- Left always smaller, right always bigger.

⊛ Design:

```
typedef struct node {
        int data;
        struct node * left;
        struct node * right;
} Node;
```

```
typedef struct bstree {
        Node* root;
} BSTree;
```

Operations

+) Create an empty BSTree
+) Check for empty BSTree
+) Insert a new item in BSTree (needs: create a new Node)
+) Search the BSTree for a given item
+) Delete an item
+) Traverse

1. Initializing

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

typedef struct node{
    int data;
    struct node* left;
    struct node* right;
}Node;
typedef struct bstree{
    Node* root;
}BSTree;
BSTree * initTree(){
    BSTree* tree = (BSTree*)malloc(sizeof(BSTree));
    tree->root = NULL;

    return tree;
}
Node* newNode (int data){
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return node;
}
bool isEmpty(BSTree* tree){
    return (tree->root == NULL);
}
```

2. Inserting : We will always insert to left or right of leaf, based on the Tree nature.

```
void insert(BSTree* tree, int value){
    if (isEmpty(tree)){
        Node* node = newNode(value);
        tree->root = node;
        return;
    }

    Node* current = tree->root;
    Node* parent = NULL;

    while (current != NULL){
        parent = current;
        if (value < current->data){
            current = current->left;
        }
        else{
            current = current->right;
        }
    }
    if (value < parent->data){
        parent->left = newNode(value);
    }
    else{
        parent->right = newNode(value);
    }
}
```

With insert (normal), we have current & parent.
Note that we need to keep track of where current went, in case it jumps to NULL.
Then, we only need to place parent → left/right.

```
Node* insertRecHelper(Node*root, int value){
    //base case
    if (root==NULL){
        return newNode(value);
    }

    // if root (of main tree or sub-tree) has a node value

    if (value < root->data){
        root->left = insertRecHelper(root->left,value);
    }
    else if (value > root->data){
        root->right = insertRecHelper(root->right,value);
    }
    return root;
}
void insertRec(BSTree* tree, int value){
    tree->root = insertRecHelper(tree->root, value);
}
```

- Use a helper function so that we can pass tree directly
- Since we are always checking and moving left or right, we only need to put recursion to move left or right.
- End case is at the end, where we return (newNode(value)); ⇒ right position

3. Printing ⇒ Requires traversing

```
void printRecHelper(Node* root){
    if (root == NULL)
        return;

    //We want to print all numbers *in order*
    printRecHelper(root->left);
    printf("%d ",root->data);
    printRecHelper(root->right);

}
void print(BSTree* tree){
    printRecHelper(tree->root);
}
```

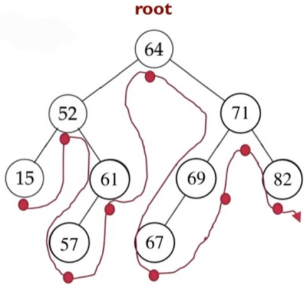
Summary: Recursive algorithms for operating on binary trees processes the root directly
+) An empty root acts as a stopping condition on recursion.

Traversing Binary Trees

There are 3 types of traversing:

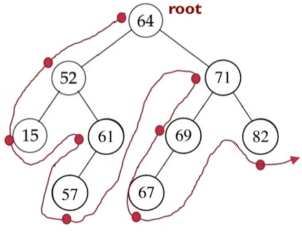
- **In order → Left - root - right**

```
root → left  = func (root →left) ;
printf (... , root → data) ;
root → right = func (root → right) ;
```



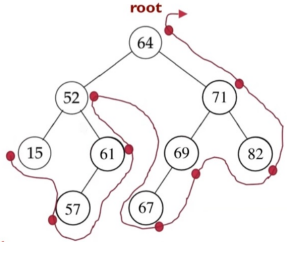
- **Pre order → Root - Left - Right**

```
printf (... , root → data) ;
root → left  = func (root →left) ;
root → right = func (root → right) ;
```



- **Post order → Left - Right - Root**

```
root → left  = func (root →left) ;
root → right = func (root → right) ;
printf (... , root → data) ;
```



4. Search

```
Node* search(BSTree*tree, int value){
    Node* current = tree->root;
    while (current != NULL && current->data != value){
        if (value < current->data){
            current = current->left;
        }
        else{
            current = current->right;
        }
    }
    return current;
}
```

→

```
Node* searchRecHelper(Node* node,int value){
    //pre-order

    if (node != NULL && node->data == value){
        return node;
    }

    if (value < node->data){
        return searchRecHelper(node->left,value);
    }else{
        return searchRecHelper(node->right,value);
    }
}

Node* searchRec(BSTree*tree, int value){
    return (searchRecHelper(tree->root,value));
}
```

⇒ Root → left → Right (Preorder)

(Remember return func)

5. Node* findMin

```
Node* findMinHelper(Node* root){
    if (root == NULL){
        return NULL;
    }
    if (root->left == NULL){
        return root;
    }
    return findMinHelper(root->left);
}

Node* findMin (BSTree*tree){
    return findMinHelper(tree->root);
}
```

6. Delete Note

Case 1 : Delete a leaf

Case 2 : 1 child ⇒ The child points upwards

Case 3 : 2 children ⇒ Replace with ← Biggest item on left - sub
Smallest item on right - sub