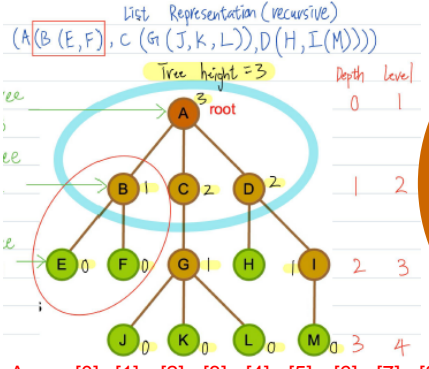

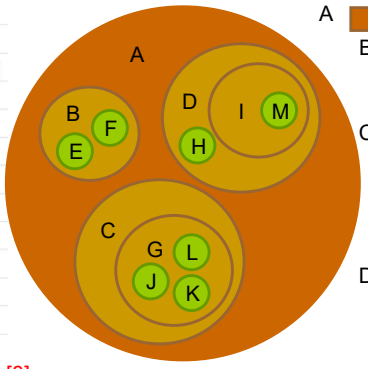

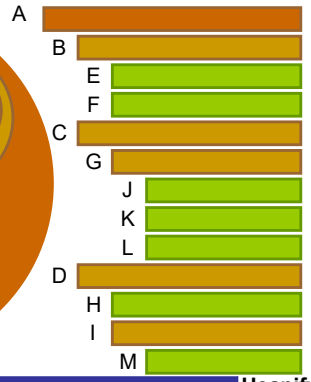

Array: [0] [1] [2] [3] [4] [5] [6] [7] [8]

A	B	C	D	E	F	G	H	I

Depth: 0 1 2 3

	Balanced BST	Sorted array	Unordered array
Find max	O(log(n))	O(1)	O(n)
insert	O(log(n))	O(n)	O(1)
Delete (given key)	O(log(n))	O(n)	O(n)
search	O(log(n))	O(log(n))	O(n)

Binary Tree
Definition: Each Node at most 2 children (0,1,2)
Skewed Tree
Definition: All nodes on left or right
Full Binary Tree
Definition: Every node either 0 or 2 children
Complete Binary Tree
Definition: All nodes completely filled at every level(possibly except last)
Perfect Binary Tree
Definition: All interior nodes hv 2 children, all leaves same depth / level
Properties
Max num of nodes on level m = 2^{m-1}
Max num of nodes = 2^{h+1}-1,h is height
Combination
For n=k, combinations = $\frac{1}{k+1} \times \frac{(2k)!}{k!\,k!}$

Heapify
```
void heapify(vector<int>& data, int i) {
cout << "heapify(data, " << i << ")" << endl;
int l = i * 2 + 0;
int r = i * 2 + 1;
int largest;
if (l <= data.size() - 1 && data[l] > data[i])
{ largest = l; }
else { largest = i; }
if (r <= data.size() - 1 && data[r] > data[largest])
{
largest = r;
}
if (largest != i) {
swap(data[i], data[largest]);
heapify(data, largest);
}
}
```

BST(Recursive)
```
if (p == NULL) return NULL;
if (x == p->info)  return p;
if (x < p->info)
return search(p->left, x);
else
return search(p->right, x);
```
Height
```
int height(treeNode<Type> *p)  {
int h, t;
if (p == NULL)
return -1; // leaf node's height is 0
h = -1;
p = p->left;
while (p != NULL) {
t = height(p);
if (t > h)
h = t;
p = p->right;
}
// h = max height of all subtrees
return h+1;
}
```

Postorder (LRV)
```
void postOrder(Node* root)
{
if (root != NULL && root->data != -1)
{
postOrder(root->left);
 postOrder(root->right);
cout << root->data << " ";
}
}
```
Preorder (VLR)
```
void preOrder(Node* root)
{
if (root != NULL && root->data != -1)
{
cout << root->data << " ";
preOrder(root->left);
preOrder(root->right);
}
}
```
Inorder (LVR)
```
void inOrder(Node* root)
{
if (root != NULL && root->data!= - 1)
{
inOrder(root->left);
cout << root->data << " ";
inOrder(root->right);
}
}
```

Count Leaf Nodes
```
int countLeafNodes(Node* root)
{
if (root == nullptr)
return 0;
if ((root->left == nullptr|| root->left->data == -1) &&
(root->right == nullptr || root->right->data == -1) && root->data!=-1)
return 1;
else
return countLeafNodes(root->left) + countLeafNodes(root->right);
}
```

Check Max Heap (O(1))
```
bool isMaxHeap(vector<int>& heap) {
    for (int i = 0; i <= (heap.size() - 2) / 2; i++) {
        if (heap[i] < heap[2 * i + 1] ||
            (2 * i + 2 != heap.size() && heap[i] < heap[2 * i + 2])) {
            return false;
        }
    }
    return true;
}
```

Insert heap O(log(n))
```
void heap_insert(vector<int>& data, int pos) {
        int k = data[pos]; // the element to be insert
        int i = pos;
        int parent = i / 2;
        while (i > 1 && data[parent] < k) {
                data[i] = data[parent];
                i = parent;
                parent = i / 2;
        }
        data[i] = k; }
```