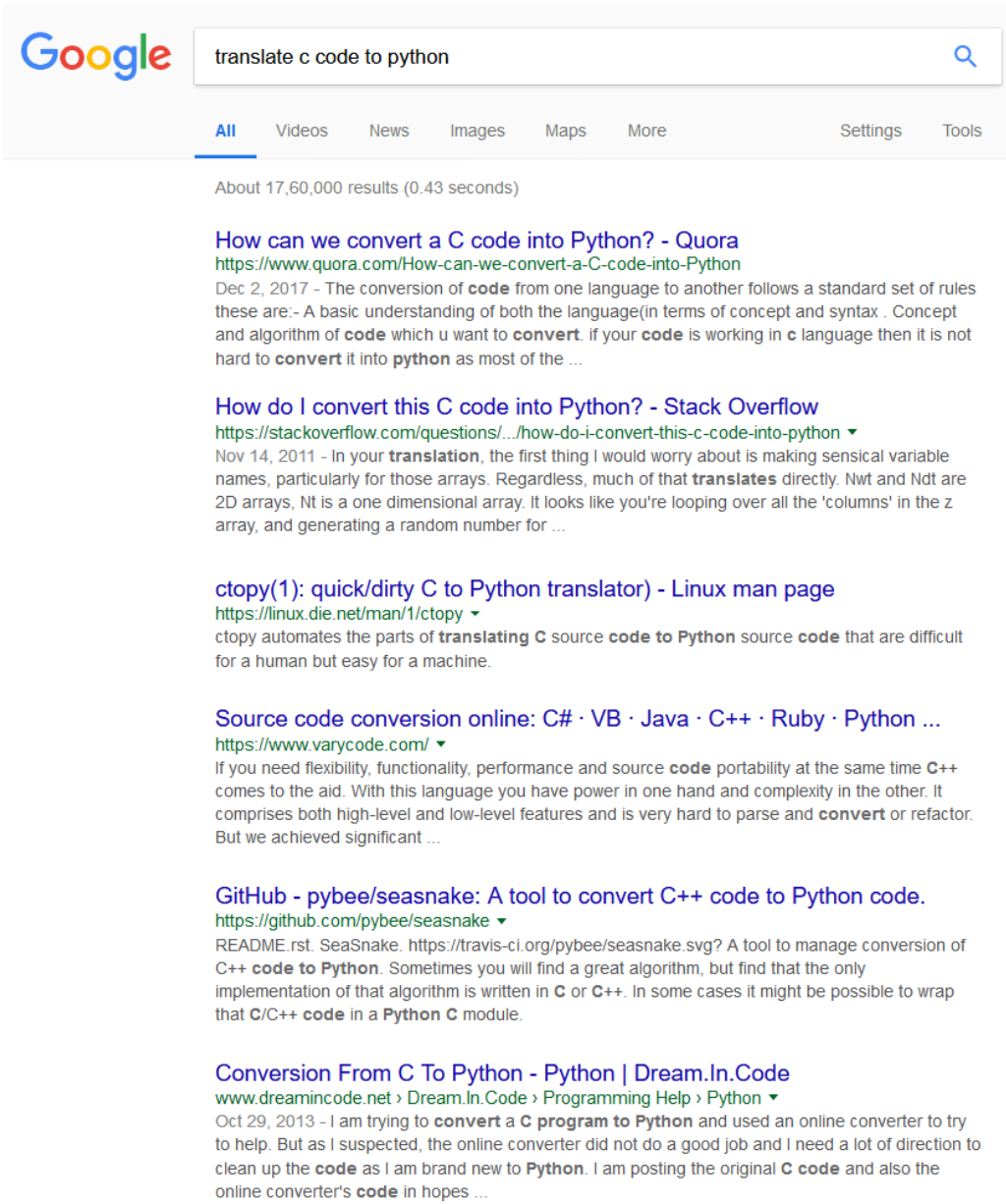
# Search and BSTs

ESC101: Fundamentals of Computing

Nisheeth

# Internet Searches




Most relevant webpage

Google sorts webpages in decreasing relevance to the query you asked!

Less relevant webpage

# Brute Force Search


- Is the element 4 present in the array?
  - Can search the array from left to right or right to left
  - `for(i=0;i<11;i++) if(a[i]==4) return i; return -1;`
  - `for(i=10;i>=0;i--) if(a[i]==4) return i; return -1;`
  - Searching from left seems faster for the query 4
- Is the element 3 present in the array?
- Is the element 5 present in the array?
- If there are N elements in the array we have to do at least N operations (to verify absence) - can we do any better?

# Binary Search

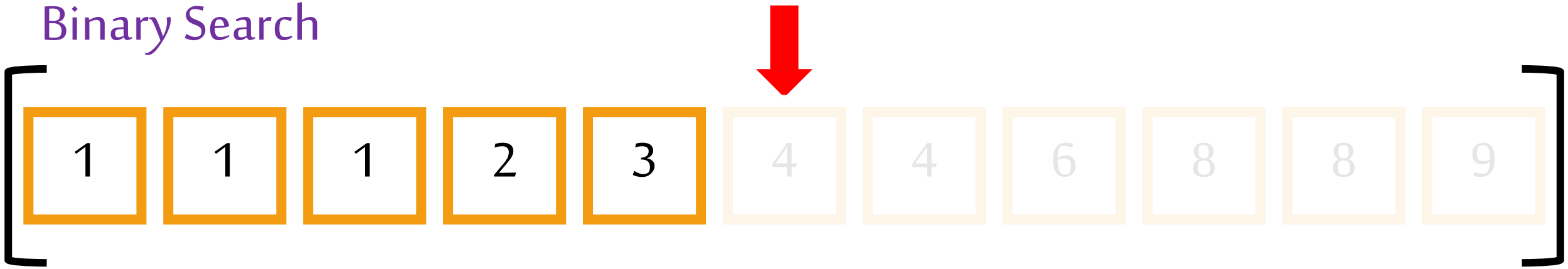

- The above array is sorted in ascending order $a[i] \leq a[i+1]$
- Can sort arrays in descending order too i.e. $a[i] \geq a[i+1]$
- Now lets try searching again by exploiting sortedness
- Crucial insight: if we are searching for the element $K$ and if we know $a[j] < K$ and array is sorted in increasing order then $a[i] < K$ for all $i < j$
  - Proof: $a[i] < a[j]$ since $i < j$ and array is sorted and we know $a[j] < K$
  - Similarly, if $a[j] > K$ then we also know $a[i] > K$ for all $i > j$
- We will use the above to eliminate vast swathes of the array

# Binary Search

| 1 | 1 | 1 | 2 | 3 | 4 | 4 | 6 | 8 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|

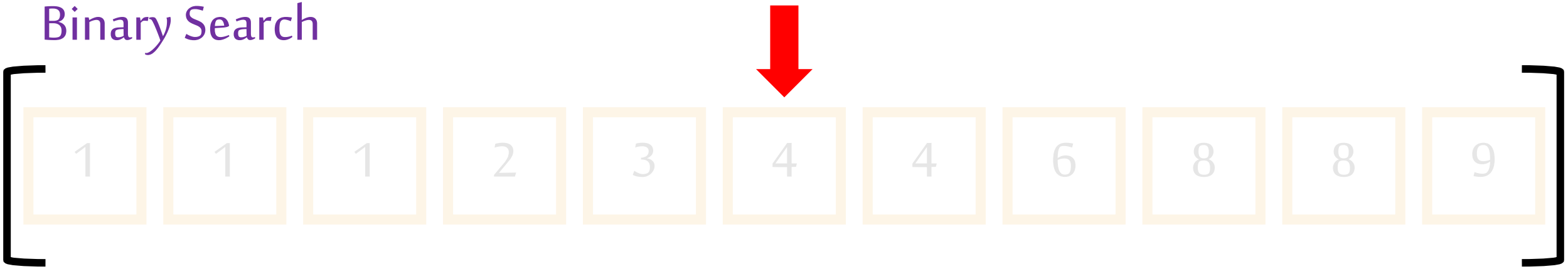
- Suppose we check $a[6] == K$? Three possible outcomes
  - Case 1: $a[6] = K$. Great we have found $K$. Go home and rest
  - Case 2: $a[6] < K$ (e.g. $K = 5$). The left half of the array can never contain $K$
    - Continue search on $a[7:11]$ -- use the same trick again
  - Case 3: $a[6] > K$ (e.g. $K = 2$). The right half of the array can never contain $K$
    - Continue search on $a[1:5]$ -- use the same trick again
- So a win-win situation – we either find the element or else reduce the search space to only half of the array
- Example of the *Divide and Conquer* technique – divide original problem into smaller instances of the same problem

# Binary Search




- Lets take an example – search for the element 1 in the array
- We will always maintain an *active range* $a[L:R]$ with $0 \leq L \leq R < N$
  - Initially the active range is entire array i.e. $L = 0, R = N - 1$
  - At every time step, we check the middle element of active range
- We will ensure two things
  - At all points of time, if the key we are searching for is at all present in the array, it must be present within our chosen active range
  - At every time step, we will halve the size of the active range
- Will need to be careful about termination criterion – more later

# Binary Search




- Lets take an example – search for the element 7 in the array
- We will always maintain an *active range* $a[L:R]$ with $0 \leq L \leq R < N$
  - Initially the active range is entire array i.e. $L = 0, R = N - 1$
  - At every time step, we check the middle element of active range
- Invariants: we will ensure two things
  - At all points of time, if the key we are searching for is at all present in the array, it must be present within our chosen active range
  - At every time step, we will halve the size of the active range
- Will need to be careful about termination criterion – more later

# Binary Search

Exercise: write a recursive version

Exercise: convert this to proper C code

## BINARY SEARCH

1. Given: Sorted array $a$ with $N$ elements, key to search $K$
2. Let $L \leftarrow 0$ and $R \leftarrow N-1$ *//Initial active range is full array*
3. While $L \leq R$
    1. Let $M \leftarrow \text{ceil}((L+R)/2)$
    2. If $a[M] == K$, return $M$ *//Found key, return location*
    3. If $a[M] > K$, set $R \leftarrow M-1$ *//Right portion can't host K*
    4. If $a[M] < K$, set $L \leftarrow M+1$ *//Left portion can't host K*
4. Return $-1$ *//We failed to find the key ☹*

The above is often known as *pseudo code*, something that gives details of an algorithm but does not strictly follow rules of C or any other programming language

# Asymptotic Time Complexity

- An effort to quantify the *speed* of algorithms in a manner that is independent of the computer on which they are executed
- Arguably binary search seems "faster" than brute force search
- We saw that in the worse case, brute force search on an unsorted array must check all $N$ elements before answering
- Can binary search on sorted arrays also be forced to do so?
- Let $T(N)$ denote the time taken by binary search to search for a key in a sorted array with $N$ elements
- We know that at every iteration of the while loop, binary search either discovers the element being searched or else reduces the length of the active range by a factor of 2

# Asymptotic Time Complexity

- Thus, we must have $T(N) \leq c + T(N/2)$
  - $c$ is the time taken to compare the middle element and update $L, R$
  - Note that $c$ does not depend on $N$ at all. Also note that $T(1) \leq c$
  - The above is a *recurrence relation*. It expresses $T$ in terms of itself
- Applying the above to $T(N/2)$ gives us $T(N/2) \leq c + T(N/4)$ i.e. $T(N) \leq 2c + T(N/4) = 2c + T(N/2^2)$
- Repeating this gives us $T(N) \leq cm + T(N/2^m)$ for any $m > 0$
- However for $m \geq \text{ceil}(\log_2 N)$ we have $2^m \geq N$
- This means that $T(N) \leq c \cdot \text{ceil}(\log N) + T(1) \leq c \cdot \text{ceil}(\log N) + c$
- For all $N \geq 4$ we have $\text{ceil}(\log N) + 1 \leq 2 \log N$ which gives us

$$T(N) \leq 2c \cdot \log N$$

# Big-Oh Notation

- Suppose we have two functions $f, g: \mathbb{R}_+ \to \mathbb{R}_+$ such that there exists a constant $c > 0$ so that for all "large" values of $x \in \mathbb{R}_+$ i.e. for all $x \geq M$ for some $M > 0$, we have

$$f(x) \leq c \cdot g(x)$$

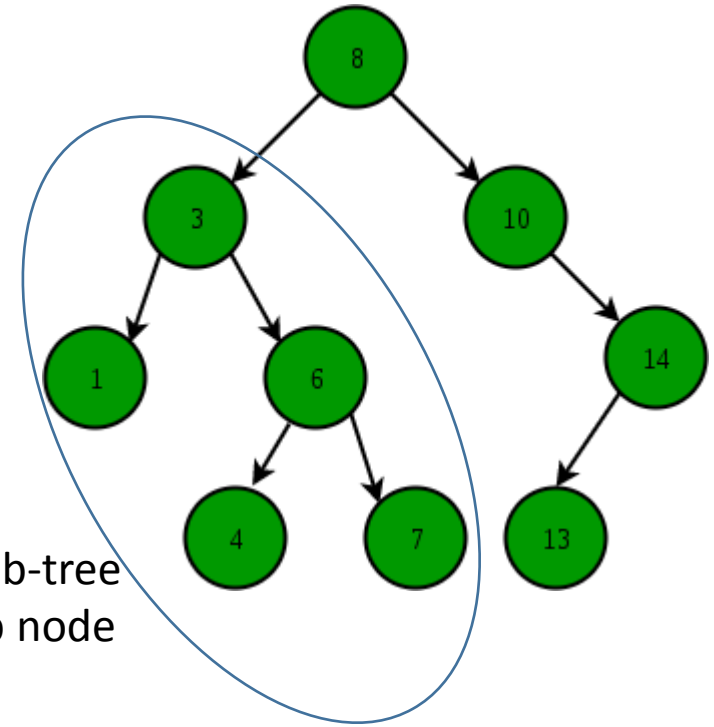
  Then we say that $f(x) = \mathcal{O}\big(g(x)\big)$
- Be careful that $c$ must not depend on $x$ for the above statement
- The above discussion shows that the runtime complexity of Binary search is $\mathrm{T}(N) = \mathcal{O}(\log N)$ since for some constant $c$ that doesn't depend on $N$ we have $T(N) \leq 2c \cdot \log N$ for all $N \geq 4$
- Exercise: show that the runtime of brute force search is $O(N)$

# Practice problems

- Given a array `int a[N];` sorted in ascending order
  - Find the number of occurrences of a given number $K$ in the array
    - Generalizes the search problem we just studied
  - Find the predecessor of a given number $K$ in the array
    - Largest number in the array that is strictly smaller than $K$. Return `NULL` if none.
    - Be careful, the key $K$ may itself occur say $N/2$ times in the array
  - Given a positive integer $M \leq N$, find the element of the array which is greater than or equal to exactly $M$ elements of the array
    - $M = 1$ gives the smallest element, $M = N$ the largest element, $M = N/2$ the median
    - If you are interested, look up the term *quantile* on the internet for more info
- Make sure your algorithms take no more than $O(\log N)$ steps!
- Can you do the above operations as fast if the array is not sorted?

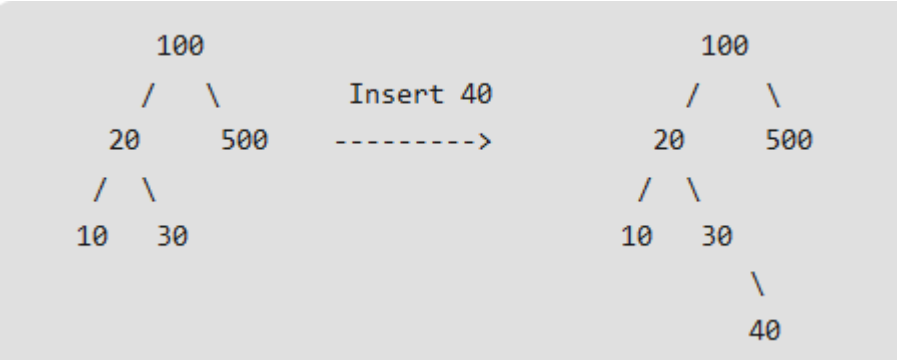
# Binary search trees

- Searching sorted arrays is much faster than unsorted arrays
- But sorting arrays itself is an expensive operation
- If you keep updating your data between searches, sorting the array over and over may not be optimal
- Binary search trees (BSTs) to the rescue:
  - Each node has two possible children (can be empty)
  - The left sub-tree of each node must only contain values smaller than the node value
  - The right sub-tree of each node must only contain values larger than the node value
  - There should be no duplicate nodes



# Building a Binary Tree

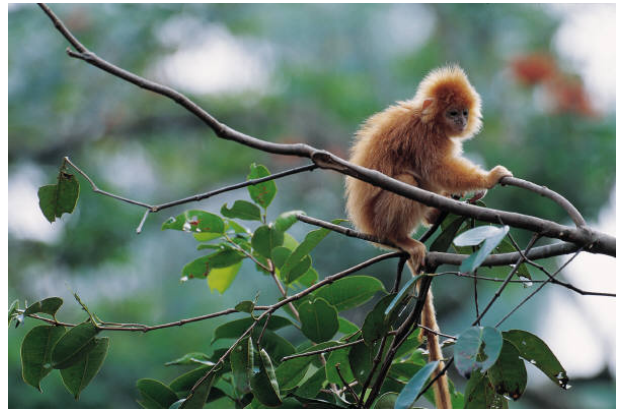
- Given some numbers, we can build a binary tree with each number being at one of the nodes (internal nodes or leaf nodes)



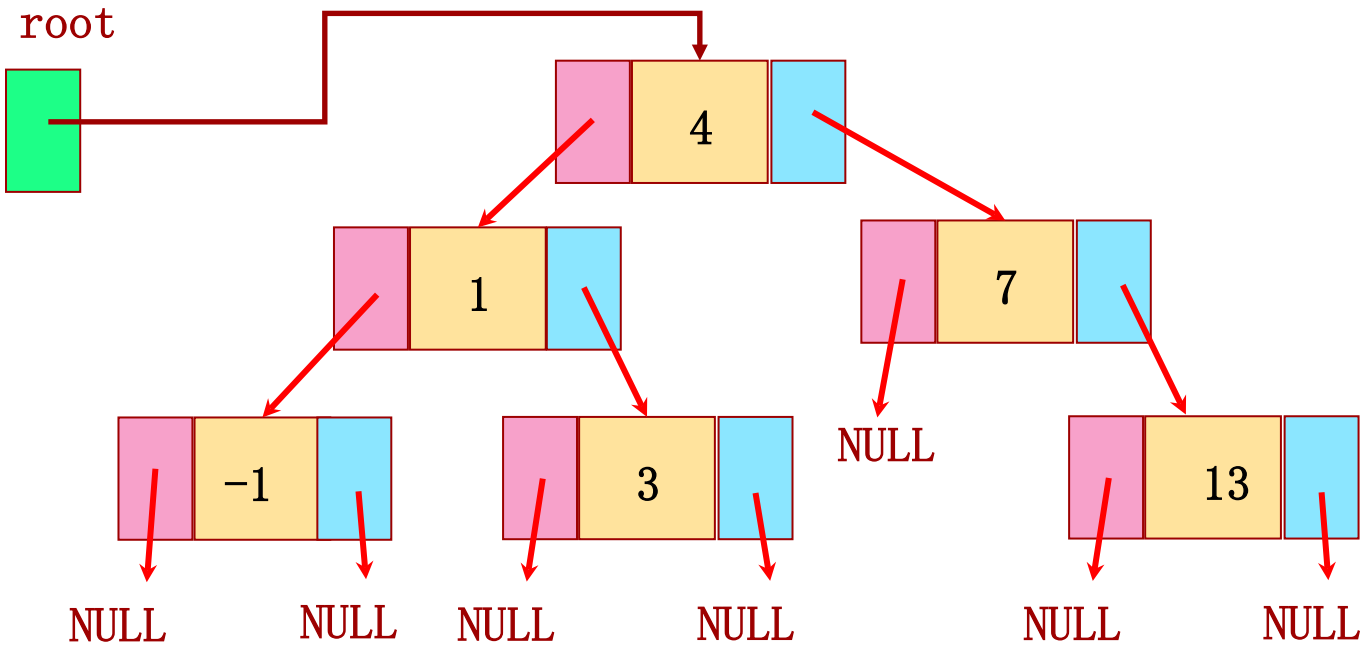

- Many ways to build the tree
- How we build it depends on how we want to organize the numbers in this tree. But in general
  - Decide which number will be at the root node, use a structure to store the number and pointers (initially NULL) to left and right subtrees
  - Send each subsequent number along the left or right branch and add to an existing leaf node

# Traversing a Binary Tree

- We won't discuss building the binary tree. Suppose we are given an already built tree
- Traversal: Visit each node in the binary tree exactly once
- Easy to traverse recursively
- Three common ways of visit
  - inorder: left, root, right
  - preorder: root, left, right
  - postorder: left, right, root



# Inorder Traversal

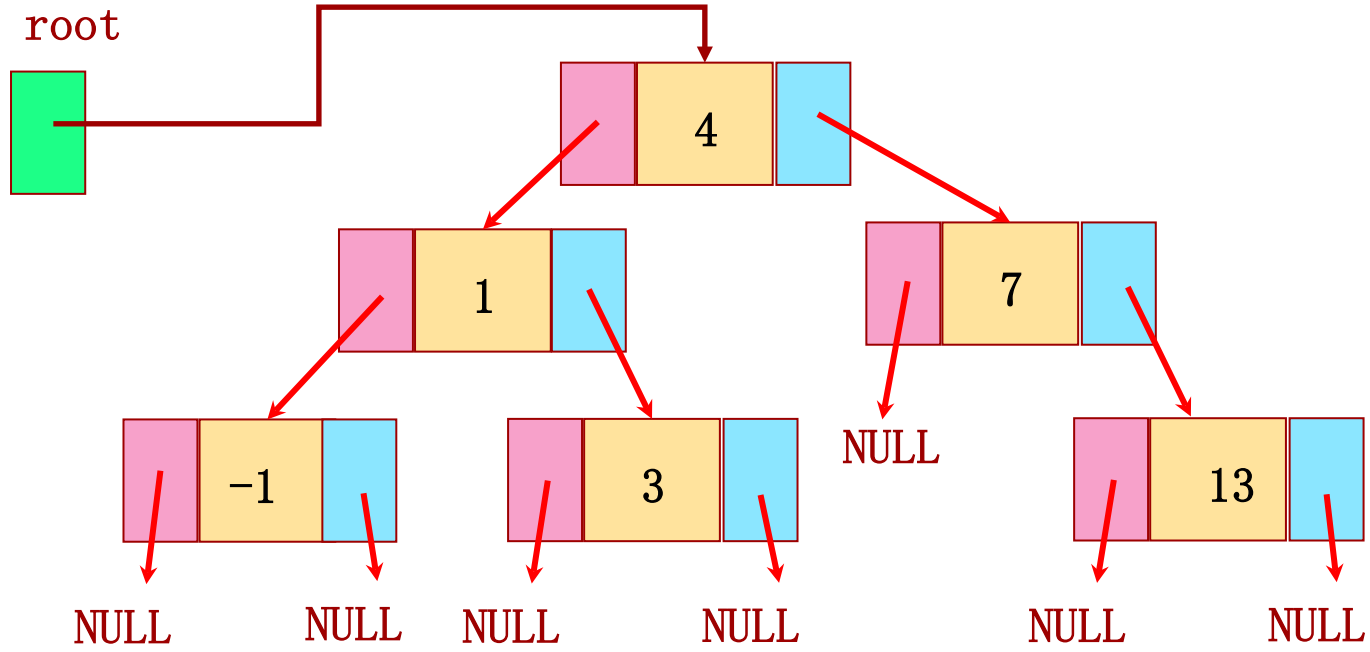



```
void inorder(tree  t)
{
    if (t == NULL)
return;
    inorder(t->left);
    printf("%d ", t-
>data);
    inorder(t->right);
}
```

## Result

-1 1 3 4 7 13

# Preorder Traversal

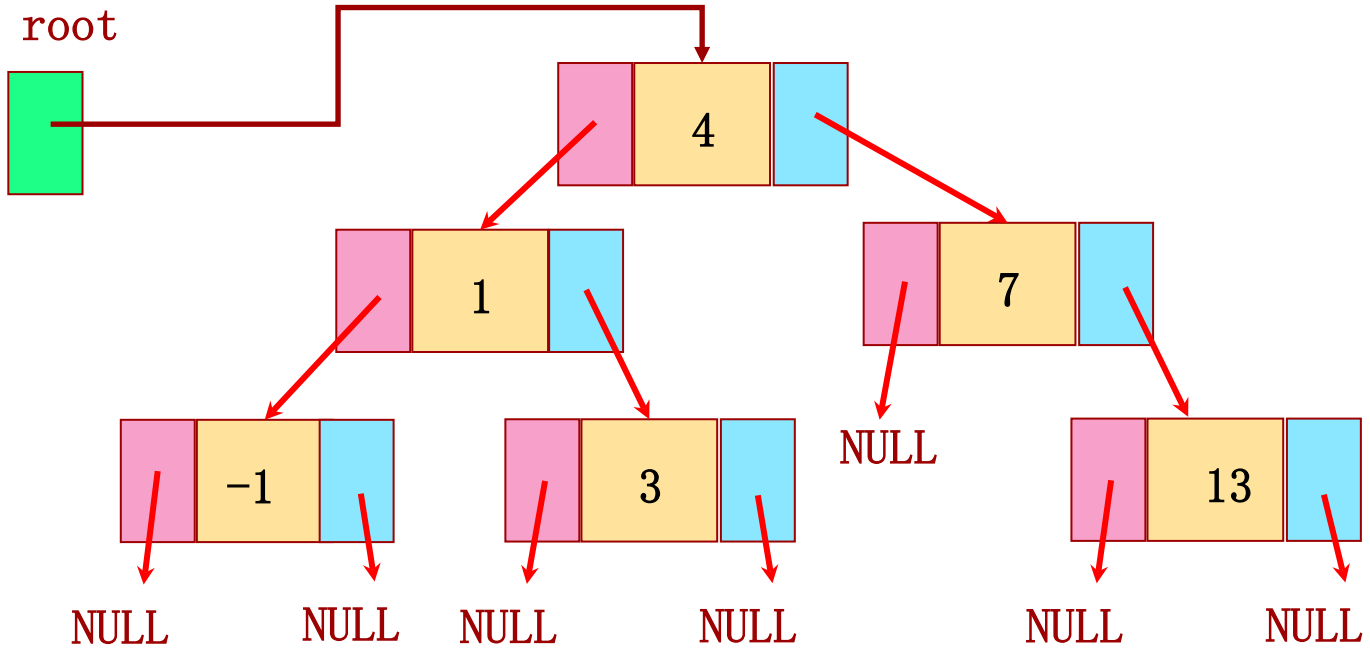



```
void preorder(tree  t)
{
    if (t == NULL)
return;
    printf("%d ", t-
>data);
    preorder(t->left);
    preorder(t->right);
}
```

## Result

4  1  -1  3  7  13

# Postorder Traversal




```
void postorder(tree  t)
{
    if (t == NULL)
return;
    postorder(t->left);
    postorder(t->right);
    printf("%d ", t-
>data);
}
```

## Result

-1  3  1  13  7  4

# Binary search vs binary search trees

You will know you're ready for your programming interview if you can intelligently answer which you'd prefer to use for any given application

Lots of considerations affect the choice: memory caching policy, application resource intensity, etc.

| | Binary search | Binary search tree |
|---|---|---|
| Search | O(log N) | O(log N) |
| Insertion | O(N) | O(log N) |
| Update | O(1) | O(log N) |
| Deletion | O(N) | O(log N) |