CSCA48 – Week 9

BST

Why do we need BSTs?

- Limitations of arrays ⇒ fixed size
  if sorted efficient search $O(\log_2 N)$
- Limitations of linked lists ⇒ traversal takes $O(n)$

Sorting a linked list? $\log N$?

NO

what do I want?

1. dynamic allocation
2. efficient search
3. Keep the structure sorted

Complexity ⇓

array of size m

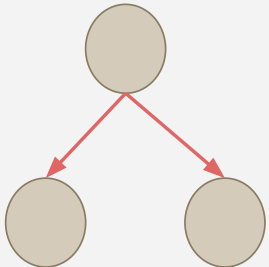
$m < n$

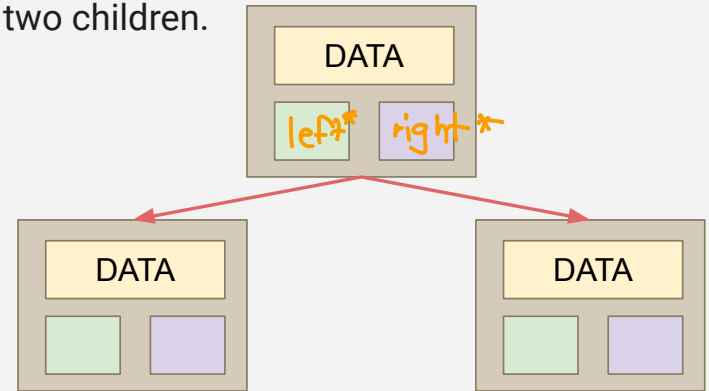
you have to add n new elements to this array of m elements and sort it

$O(n)$ + $O(n \log n)$

Trees, Binary Trees, and Binary Search Trees

- Tree
  - A collection of nodes, where each node is a data structure consisting of a value and a list of references to nodes.
- A binary tree
  - is a tree in which each node has at most two children.
  - Left child and right child.





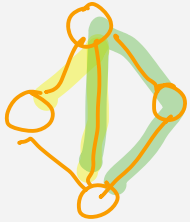
Data structures studies so far

Linear (Data arranged in sequential order) ⇒ linked list, arrays, stack

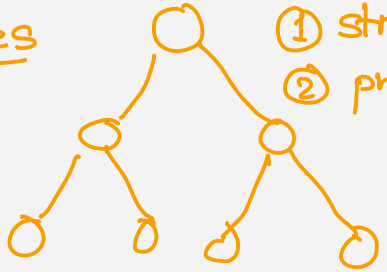
sequential data structure

Non-linear data structures (No linear arrangements of data elements)

Graph



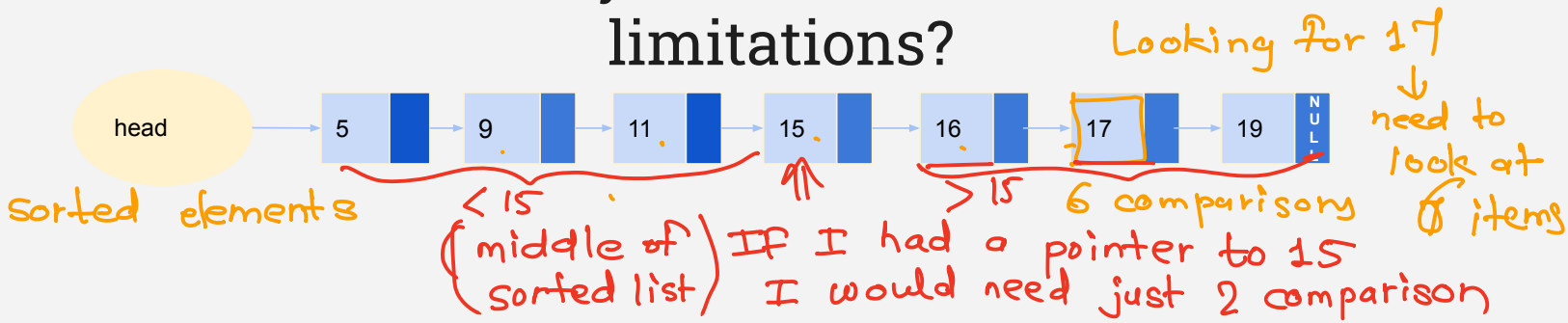
Trees

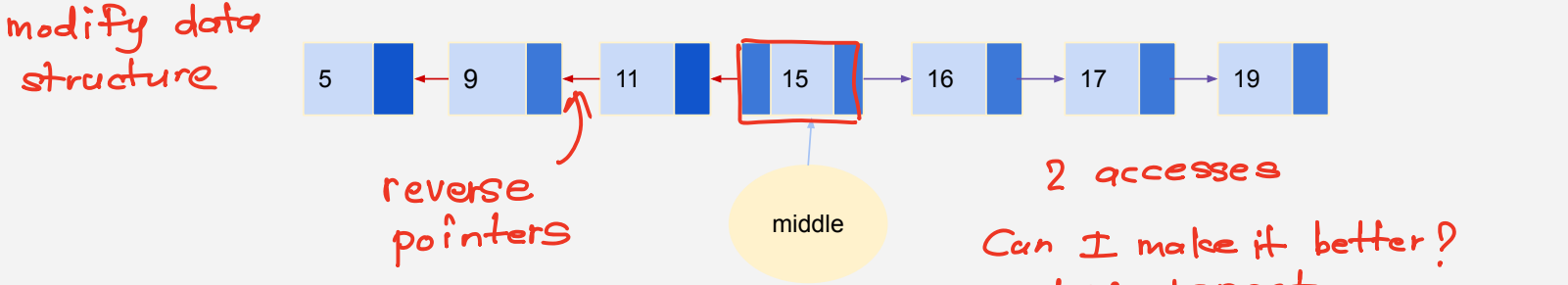


① structure
② property

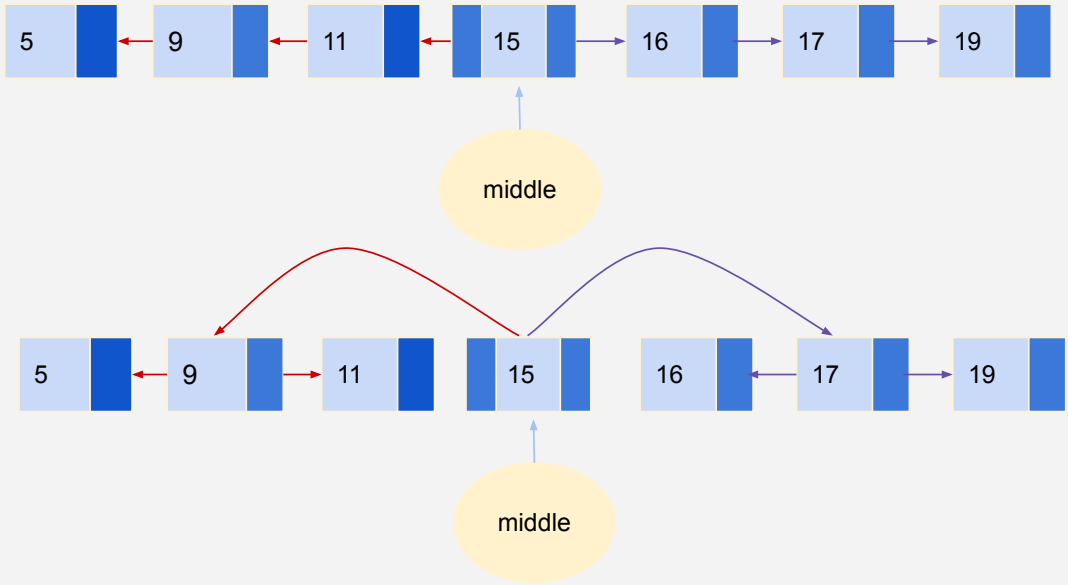
diff. between Graph & trees?

Can we modify Linked Lists to overcome limitations?





Can we modify Linked Lists to overcome limitations?

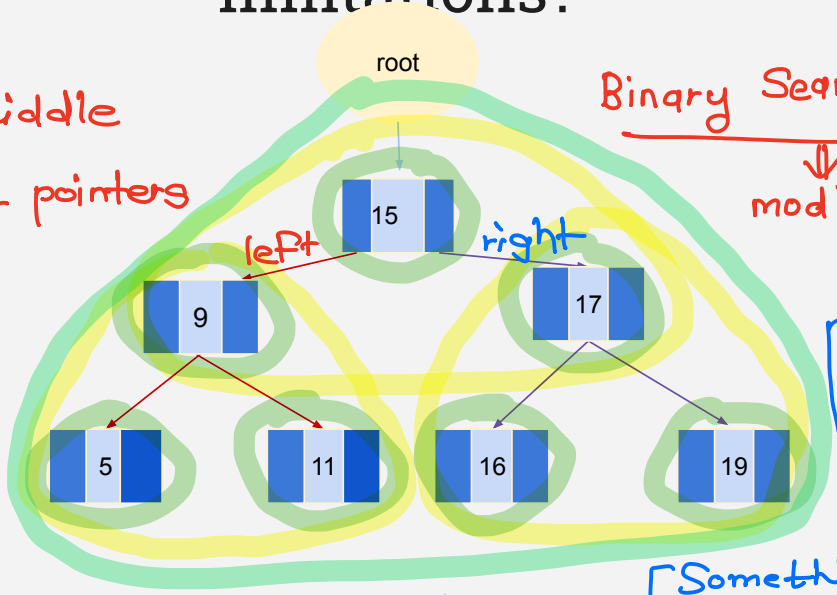


Can we modify Linked Lists to overcome limitations?

① find the middle

② Just add 2 pointers

non-linear arrangement



Binary Search Tree

⇓

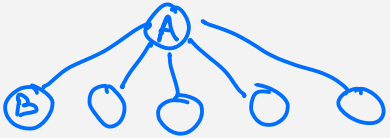
modified linked list

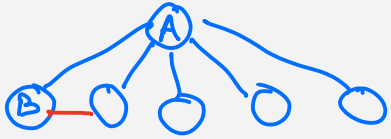
BST ⟺ Recursion

Something defined in terms of itself

Definition of the tree itself + operations on tree

Tree

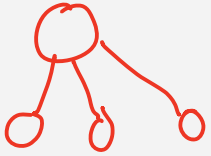




- Set of nodes
- A set of edges that connect nodes
- Constraint: (No cycles, no loops)
  - There is exactly one path between any two nodes

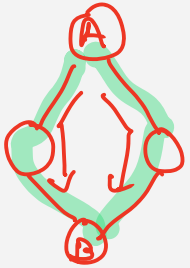
trees ⟷ Graphs

tree ? yes

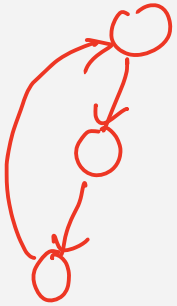


tree? yes

tree

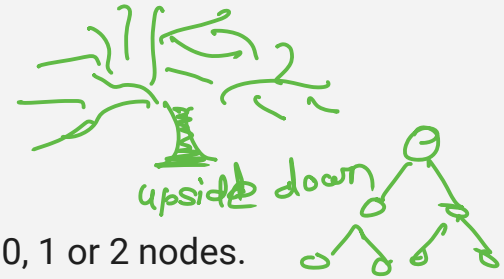


not a tree

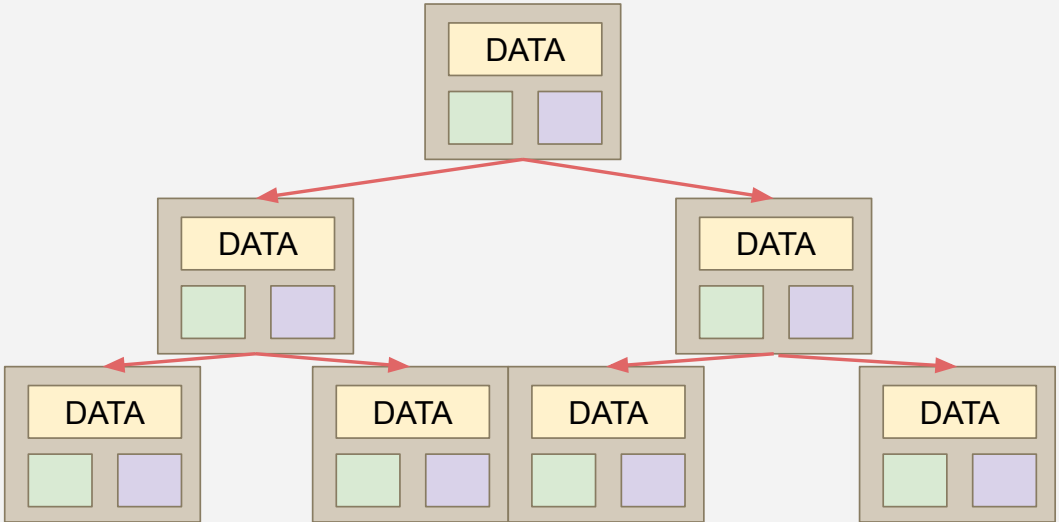


No

Binary Tree



- Binary tree
  - is a nonlinear data structure in which nodes can have 0, 1 or 2 nodes.
  - has the property that each node has two children nodes, the left child, and the right child.



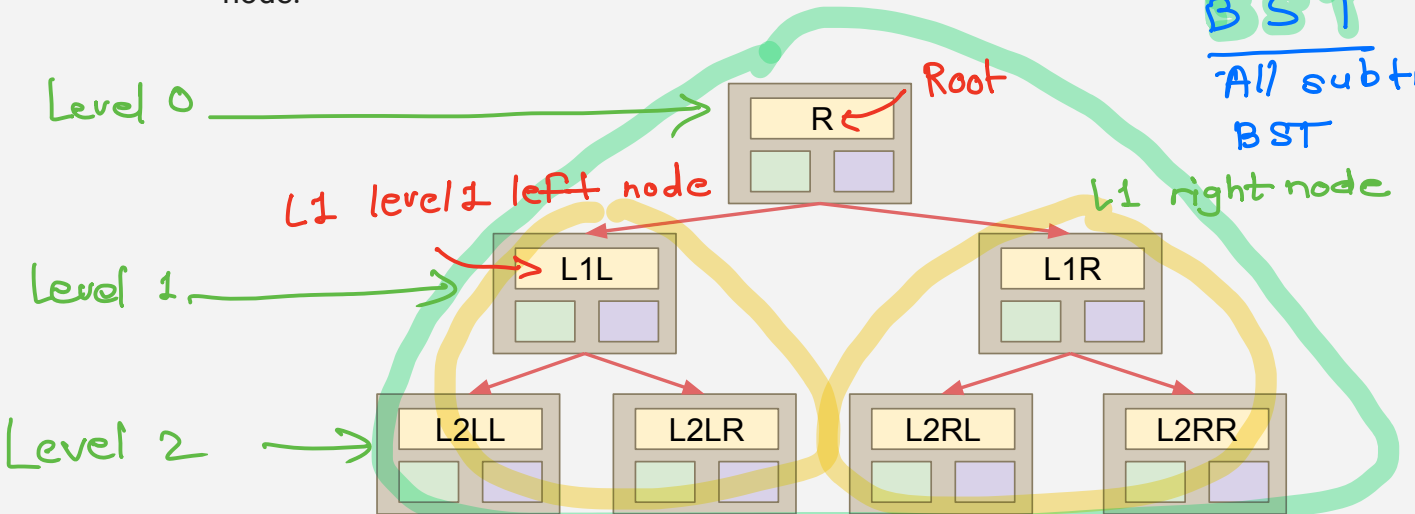
Binary Search Tree (BST)

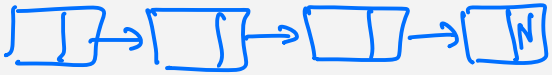
upside down tree

- A Binary Search Tree (BST) is a binary tree with **structured organization** of nodes with **BST property**:
  - Data in the nodes on the **left sub-tree** have value **less than,** or equal to the value of the data in the node.
  - Data in the nodes on the **right sub-tree** have value **greater than** the value of the data in the node.

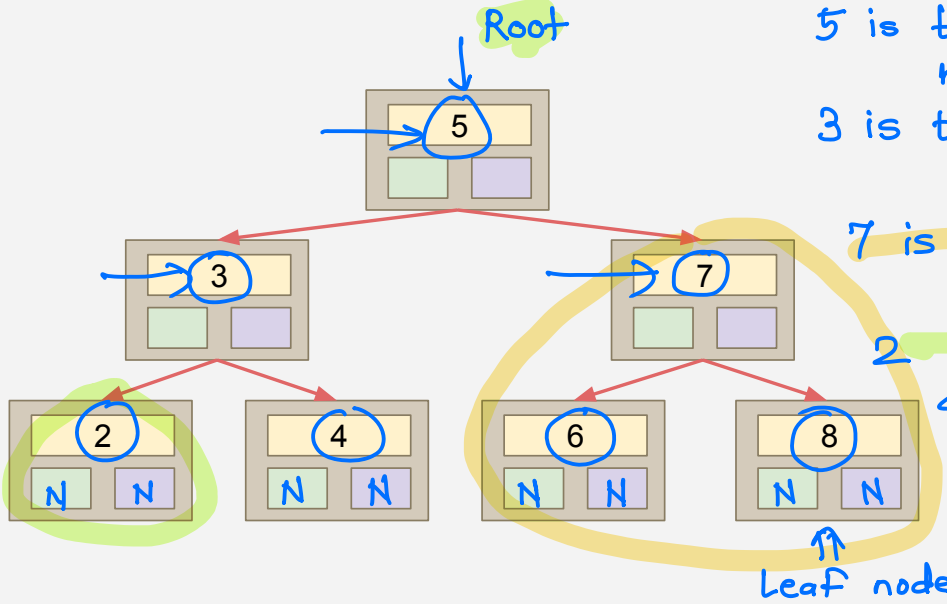
BST

All subtrees are BST





Observations about BST



5 is the root of the subtree rooted at 5

3 is the root of the subtree rooted at 3

7 is the root of the subtree rooted at 7

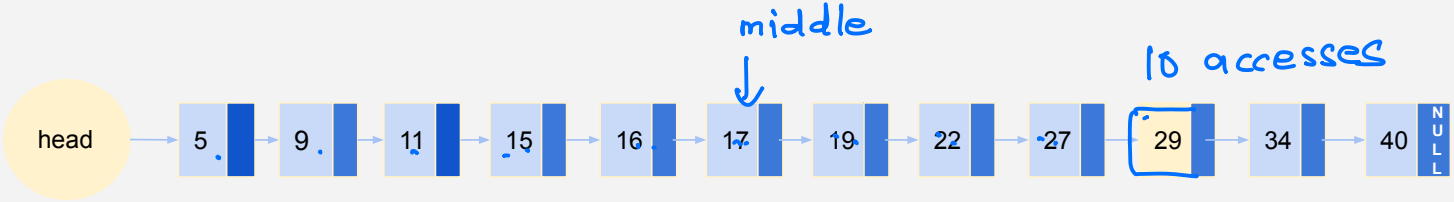
2 " at 2

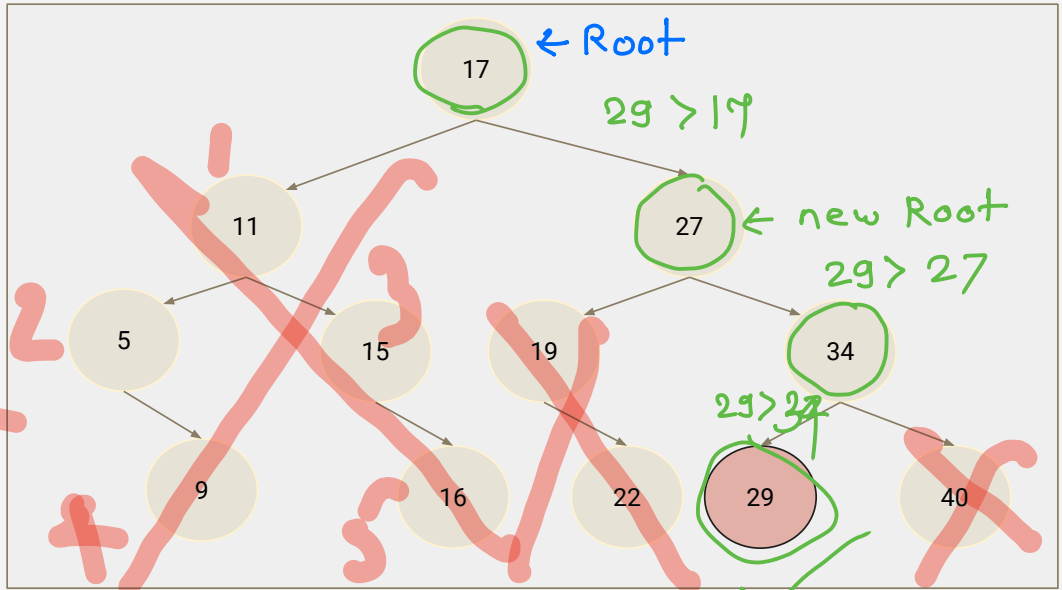
4 " at 4

All the nodes can be roots

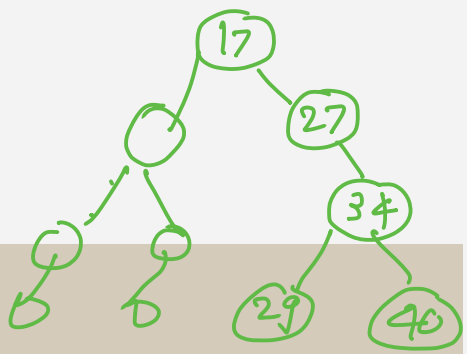
Order of Insertion for BST is Important

Searching in BST vs searching in linked list





Searching in BST



name of function

17 , 29 27, 29 , 34, 29, 29, 29

searchForKey ← subtree, queryKey
if the queryKey is equal to the key at the root node of the subtree
    Found the key! Return a reference to the root node of the subtree
else
    if the queryKey is **less than or equal to the key** in the root node of the subtree
        searchForKey ← left subtree, queryKey
    otherwise
        searchForKey ← right subtree, queryKey

29 , 29

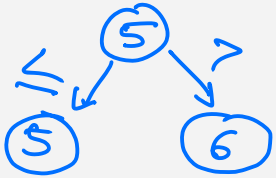
27 , 29

34 , 29

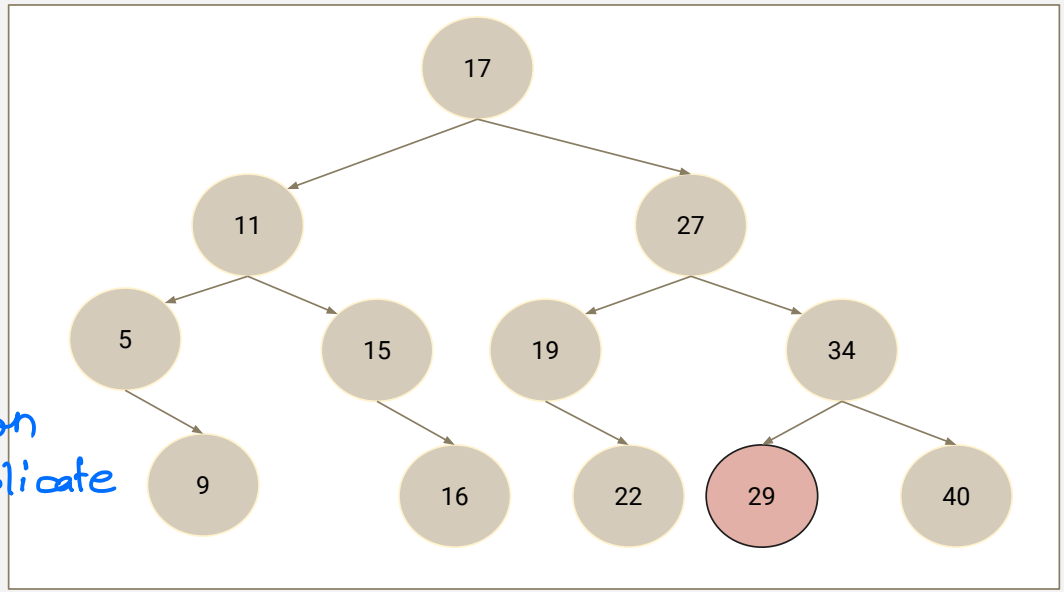
key ⇒ identifies the node uniquely

int / string / float ← key representation for the entire node

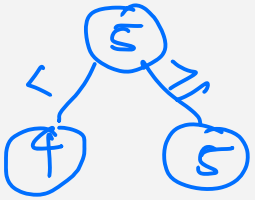
Abstract data type



Searching for 29



OR



Searching efficiency goes down with duplicate keys

Dynamic set ADT (Generally we do not allow duplicate values)

Observations

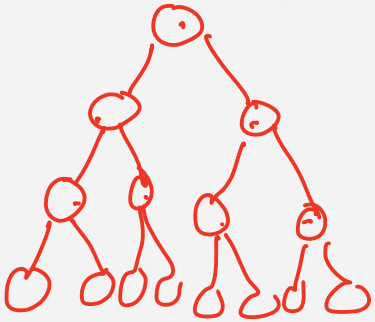
how much work at each level

- Search process discards large portion of values stored in the tree
  - How many exactly?
- Search moves down one level in order to find out the key
  - How many levels?

for N nodes, levels ⇒

top to bottom direction

figure out the complexity



7 nodes
3 Levels

15 nodes
4 levels

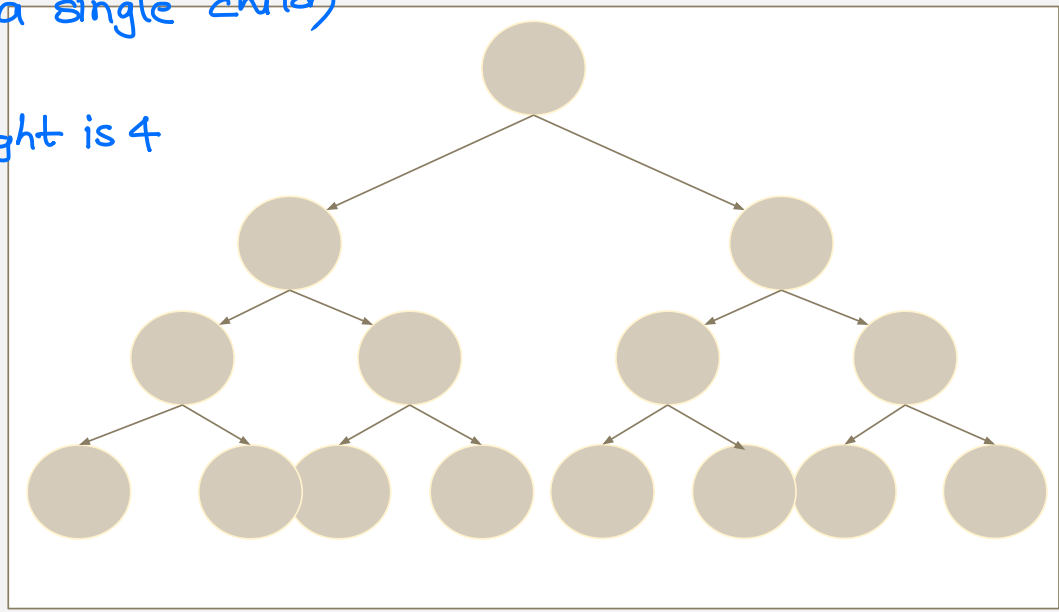
Full Binary Tree

(cannot have a single child)

Every node has 0 or 2 children

For **15** nodes, **4** levels ← height is 4

For **N** nodes, how many levels?



Complete Binary Tree

$k \approx \log_2 N$ ← No. of nodes

k ← no. of levels

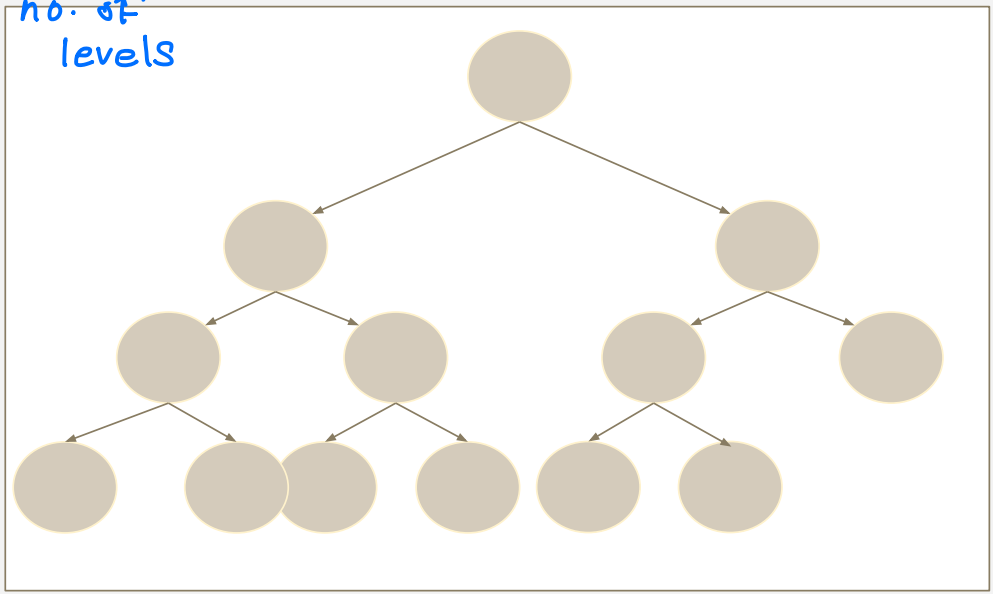
All the levels are completely filled except possibly the last level and the last level has all keys as left as possible

For **13** nodes, **4** levels

For **N** nodes, how many levels?

when we divide in 2 equal parts

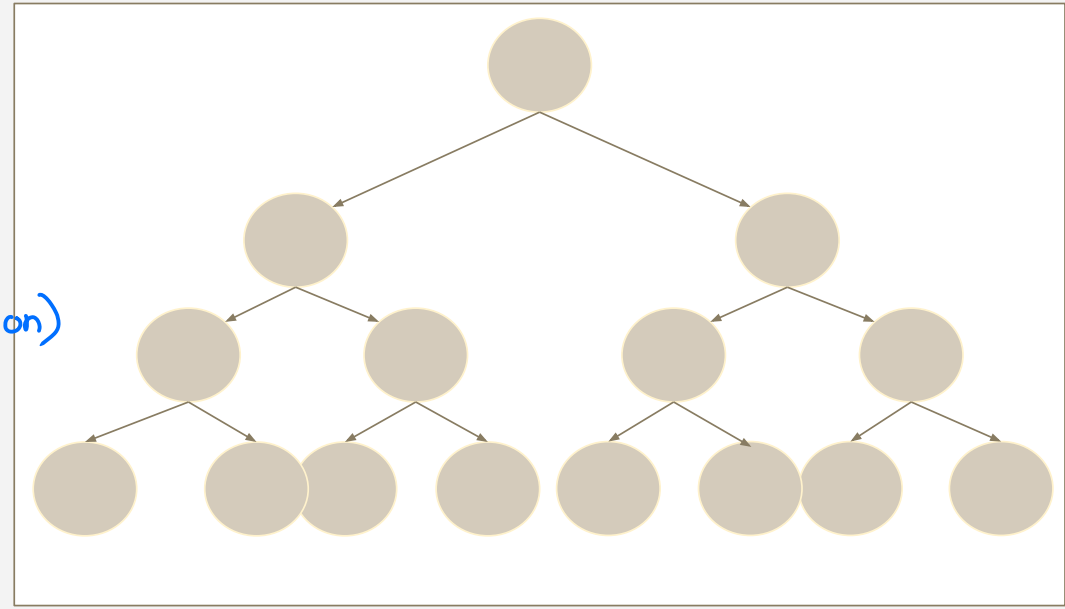
complexity = $\log_2 N$



Full Binary Tree

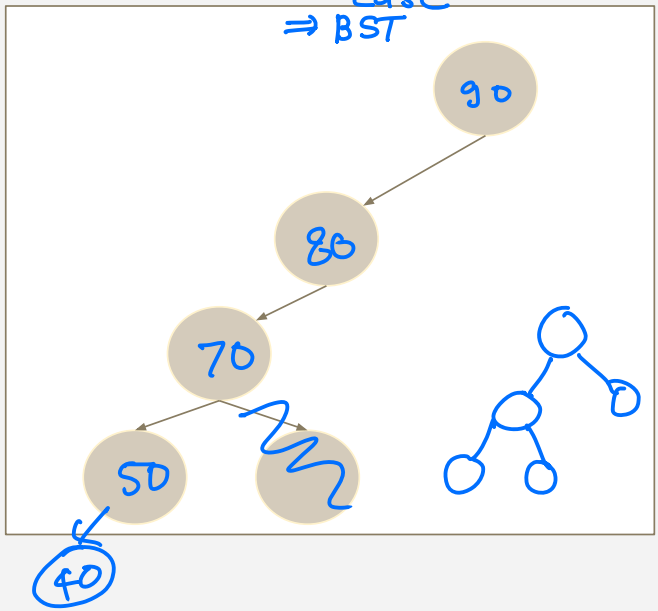
Every node has 0 or 2 children
If BST has **4** levels, how many nodes in BST?
For **k** levels, how many nodes? (flipped question)

$N = 2^k$ nodes



Is BST always full / complete?

randomness ⇒ average case ⇒ BST



No

order in which we receive the nodes, decide the height of the tree

https://www.cs.usfca.edu/~galles/visualization/BST.html

height = no. of levels = 5

no. of nodes = 5

worst case ⇒ BST acts like linked list

Operations on BST

- Initializing an empty binary tree
- Inserting a node into the BST
- Removing (deleting) a node in the BST
- Searching for a specific item in the collection
- traversal

} similar to linked list

Implementing BST

Restaurant review app Kelp

Start by creating a CDT(composite data type for restaurant):

tag

```
typedef struct Restaurant_Score
{
    char restaurant_name[MAX_STRING_LENGTH];
    char restaurant_address[MAX_STRING_LENGTH];
    int score;
} Review;
```

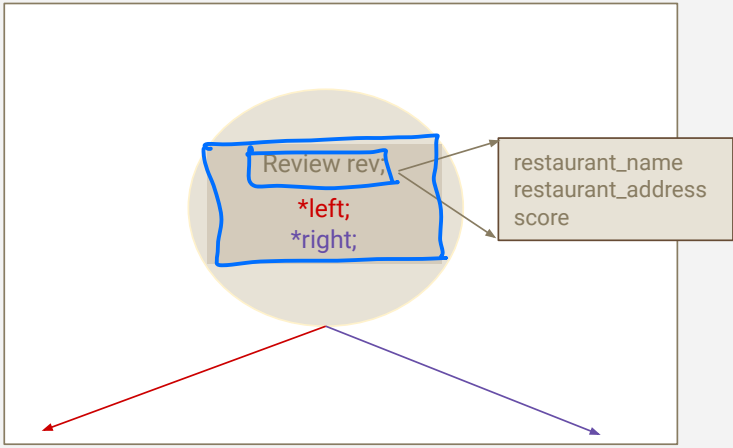
.name

Creating Node

self referential pointer (cannot use nickname BST_node)

```
typedef struct BST_Node_Struct
{
      Review rev; // Stores one review
      struct BST_Node_Struct *left; // A pointer to its left child
      struct BST_Node_Struct *right; // and a pointer to its right child
} BST_Node;
```

CDT inside another CDT

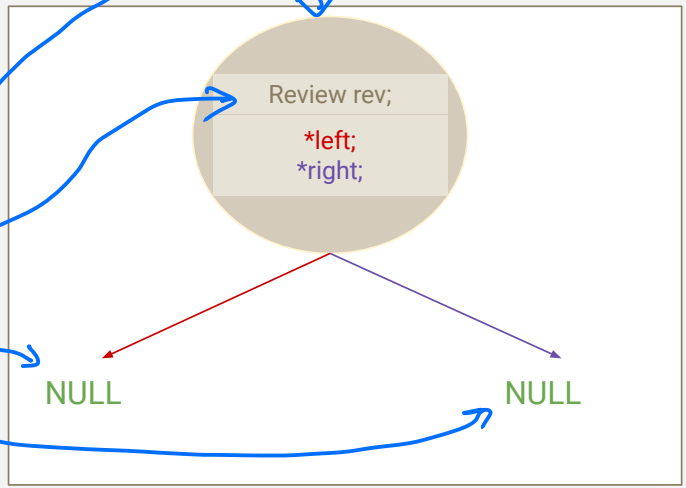


Initializing a node

arrow pointer because pointer variable new_review

similar to linked list

```
BST_Node *new_BST_Node(void)
{
    BST_Node *new_review=NULL; // Pointer to the new node

    new_review=(BST_Node *)calloc(1, sizeof(BST_Node));

    new_review->rev.score=-1;
    strcpy(new_review->rev.restaurant_name,"");
    strcpy(new_review->rev.restaurant_address,"");
    new_review -> left = NULL;
    new_review -> right = NULL;
    return new_review;
}
```

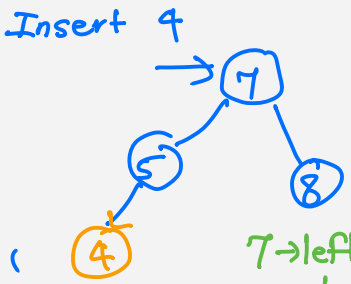


Inserting Nodes in BST

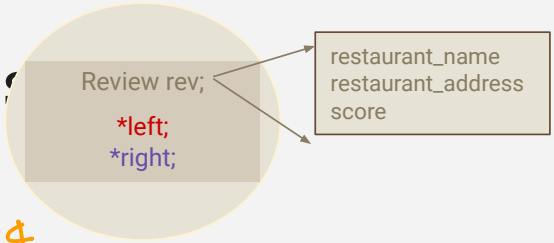
- Differences from inserting into a linked list
  - Pointer to **root** instead of head
  - BST property must be enforced when new **key** is inserted
- What could be a **key** in Review CDT??

↖ keeps the data structure sorted (always)

good idea to keep an id to uniquely identify key

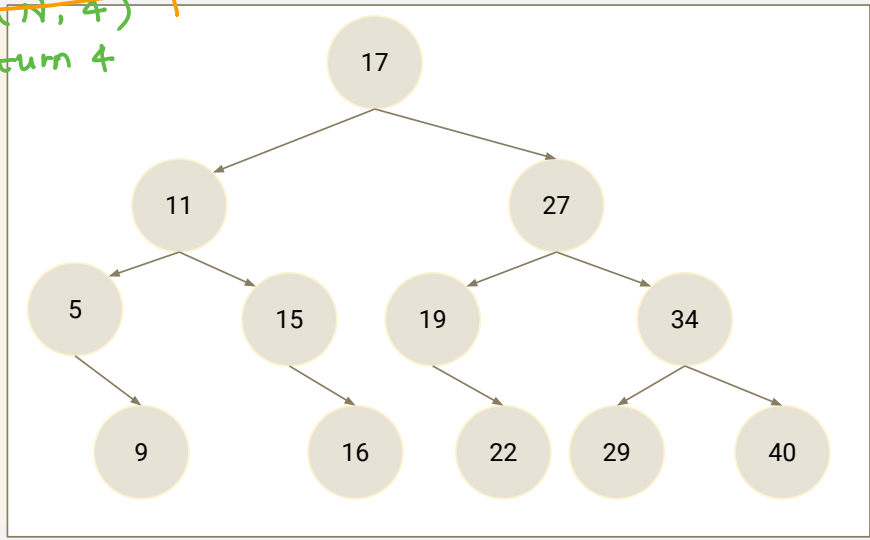


Inserting Nodes



BST_insert (7, 4)
7→left = BST_insert (5, 4)
5→left = BST_insert (N, 4) 4
↓
return 4

```
BST_Node *BST_insert(BST_Node *root, BST_Node
*new_review)
{
      if (root==NULL) // Tree is empty, new node
      becomes
            return new_review; // the root
      // Determine which subtree the new key should be
      in
      if (strcmp(new_review->rev.restaurant_name,\

      root->rev.restaurant_name)<=0)

      root->left=BST_insert(root->left,new_review);
      else

      root->right=BST_insert(root->right,new_review);
      return root;
}
```

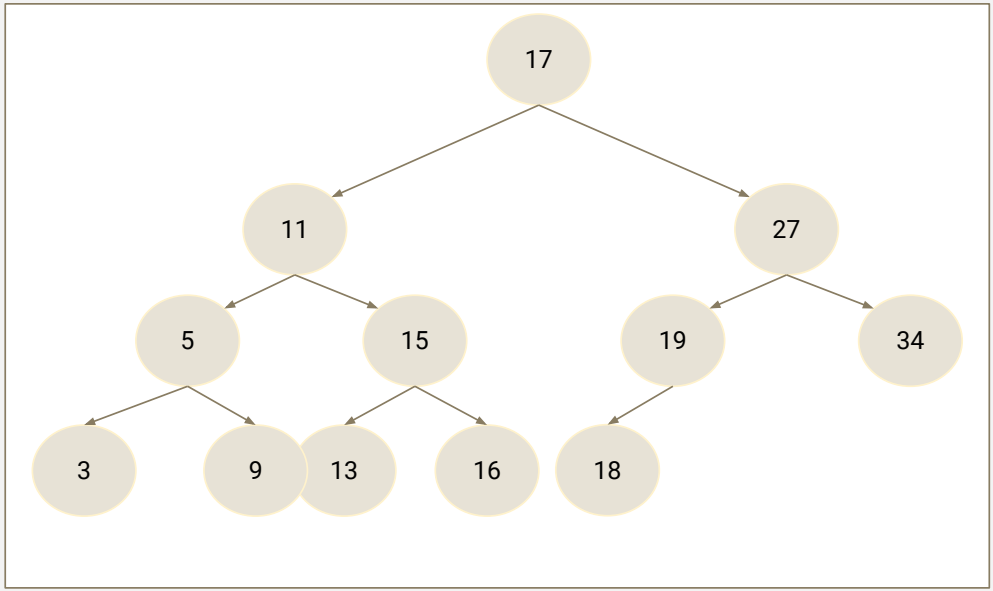


Recursion

Function calls itself directly or indirectly

```
BST_Node *BST_insert(BST_Node *root, BST_Node *new_review)
{
      if (root==NULL) // Tree is empty, new node becomes
            return new_review; // the root
      // Determine which subtree the new key should be in
      if (strcmp(new_review->rev.restaurant_name,\
                        root->rev.restaurant_name)<=0)
            root->left=BST_insert(root->left,new_review);
      else
            root->right=BST_insert(root->right,new_review);
      return root;
}
```

Every subtree of a BST is also a BST

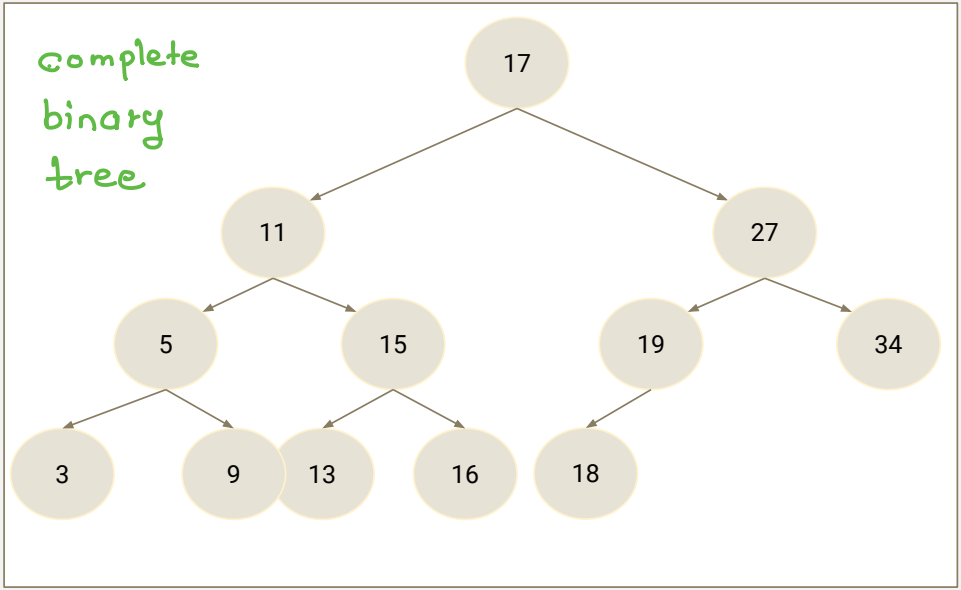


Insertion Example

nodes we want to insert

17 11 27 5 15 19 34 3 9 13 16 18

```
If root == NULL
    Insert at the root
Compare the inserting element with
root,
if less than root,
      then recursively call left
      subtree,
else
      recursively call right subtree.
```

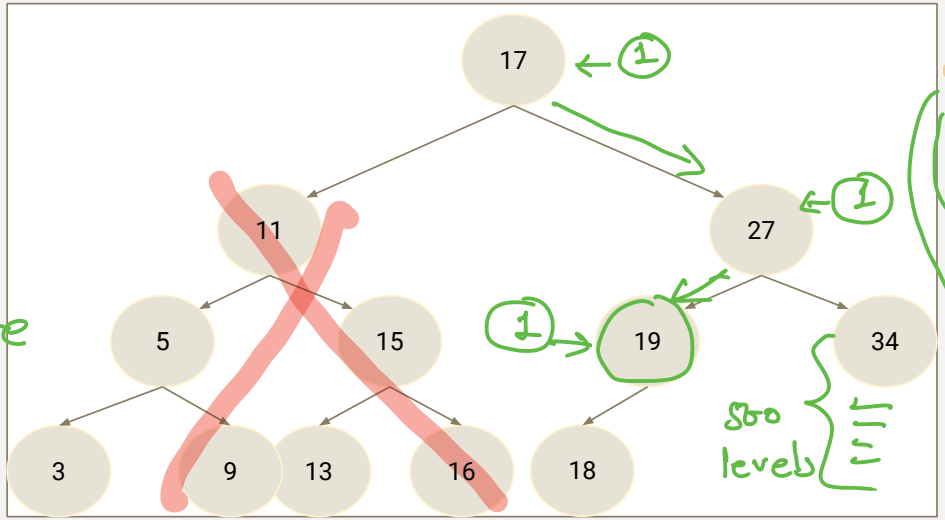


Revisit the complexity of search in BST

How to calculate the search complexity of BST?? $O(\log_2 N)$
depends on the height of the tree, and we also know that on a full BST the height of the tree is $\lceil \text{Log}_2(N) \rceil$,

ceiling function

Insert & search are similar



amount of work at each level
*
no. of levels
constant time * $\log_2 N$
how many comparison?

Search in BST

since it's key

```
BST_Node *BST_search(BST_Node *root, char name[1024])
{
     if (root==NULL) // Tree or subtree is empty
          return NULL;
     // Check if this node contains the review we want, if so,
     return // a pointer to it
     if (strcmp(root->rev.restaurant_name)==0)
          return root;
     // Not in this node, search the corresponding subtree
     if (strcmp(name, root->rev.restaurant_name)==0)
          return BST_search(root->left,name);
     else
          return BST_search(root->right,name);
}
```

success

fail

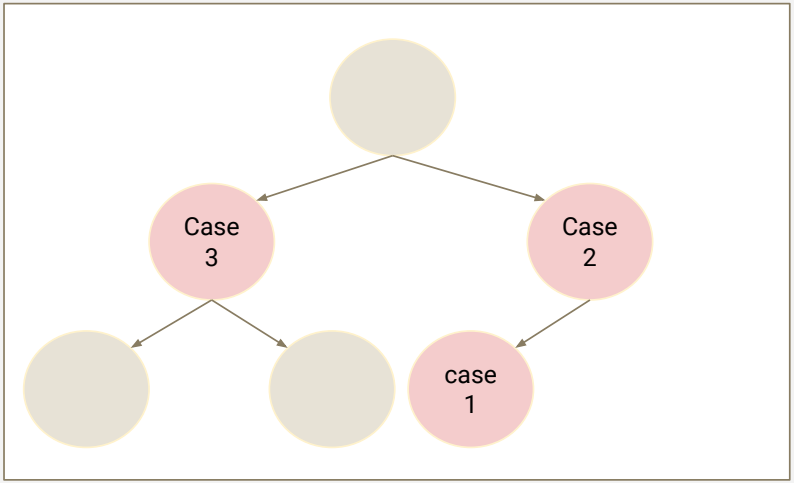
Updating in BST

- Similar to searching followed by updating
- No updates allowed on the keys since we don't want to change the keys
- Why??

Deleting items from BST

- Case 1: Deleting leaf node
- Case 2: Deleting non-leaf node with one child
- Case 3: Deleting non-leaf nodes with two children

Goal:
Always maintain BST property

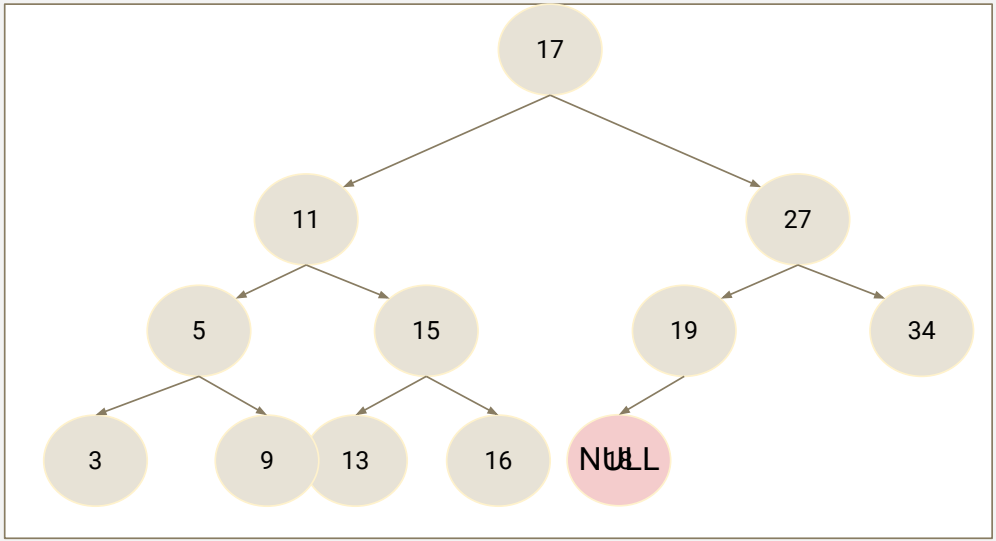


Function to delete a node

bit restructured

```
BST_Node *BST_delete(BST_Node *root, char name[1024])
{
     if (root==NULL)
          return NULL; // Tree or sub-tree is empty

     else if (strcmp(name, root->rev.restaurant_name) < 0)
          root->left = BST_delete(root->left,name);

     else if (strcmp(name, root->rev.restaurant_name) > 0)
          root->right = BST_delete(root->right,name);

     else ← (when you find the node to delete)
     {
        //case1

        //case2

        //case3

     }
     return root;   ??
}
```

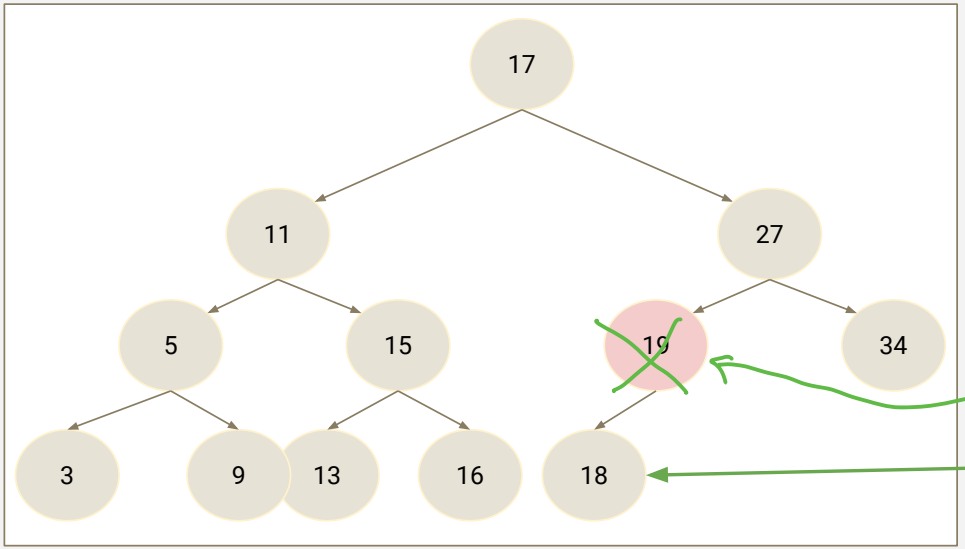
Case 1: Deleting leaf node



Case1:

```
if (root->left==NULL && root->right==NULL)
{
        free(root);
        return NULL;
}
```

Case 2: Deleting non-leaf node with one child



*tmp

Case 2:

pointer to BST_node

```
BST_Node *tmp;
if (root->right==NULL)
{
        tmp = root->left;
        free(root);
        return tmp;
}
```

*tmp

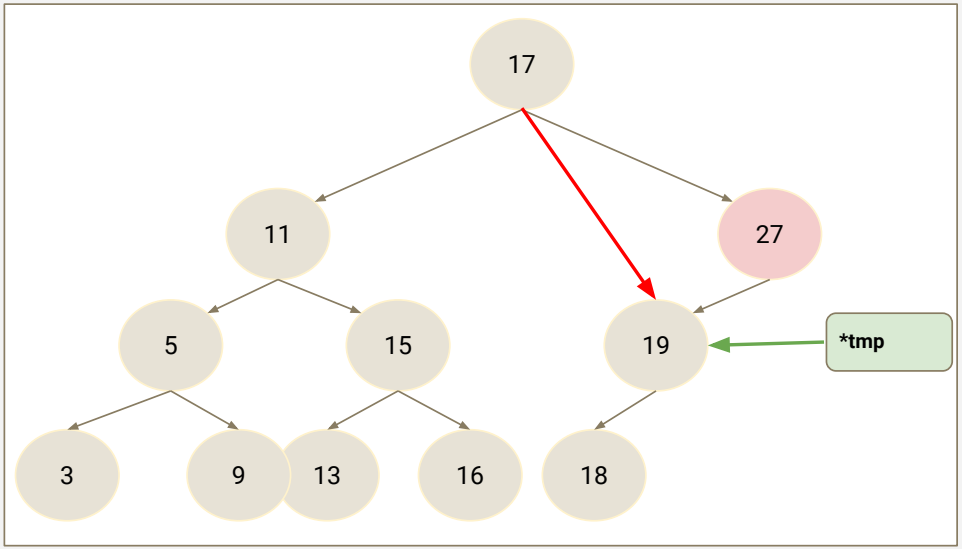
what's root??

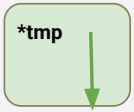
19

Anything missing?

yes connection bet^n 18 and 27

Case 2: Deleting non-leaf node with one child





Case 2:

```
BST_Node *tmp;
else if (root->right==NULL)
{
        tmp = root->left;
        free(root);
        return tmp;
}
```

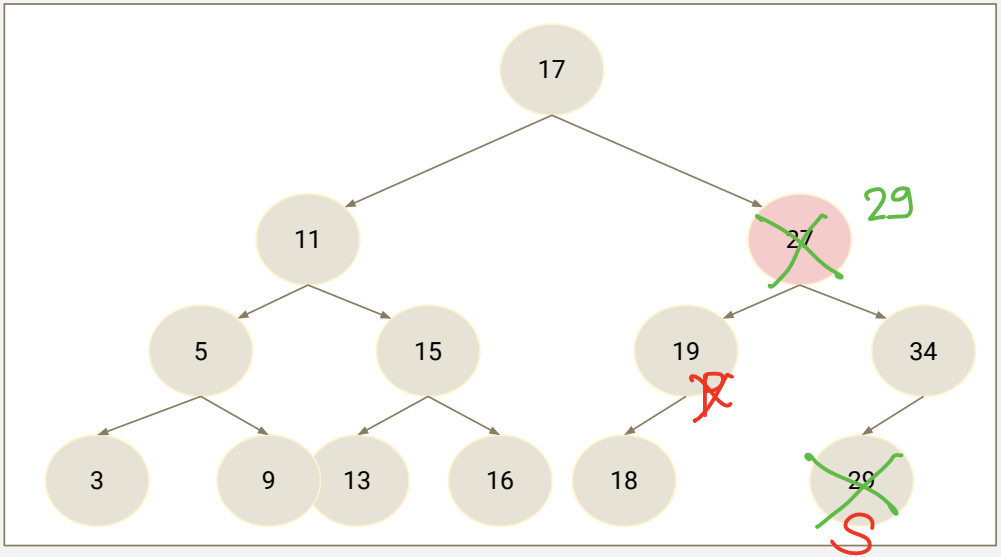
The red arrow is fixed in the recursive call:

```
root->right = BST_delete(root->right,name);
```

Case 3: Deleting non-leaf nodes with two children

predecessor

Always use successor



Which node should replace 27??

Answer: Successor of 27

Successor?

Smallest element in the right subtree of 27

Successor of 27: **29**

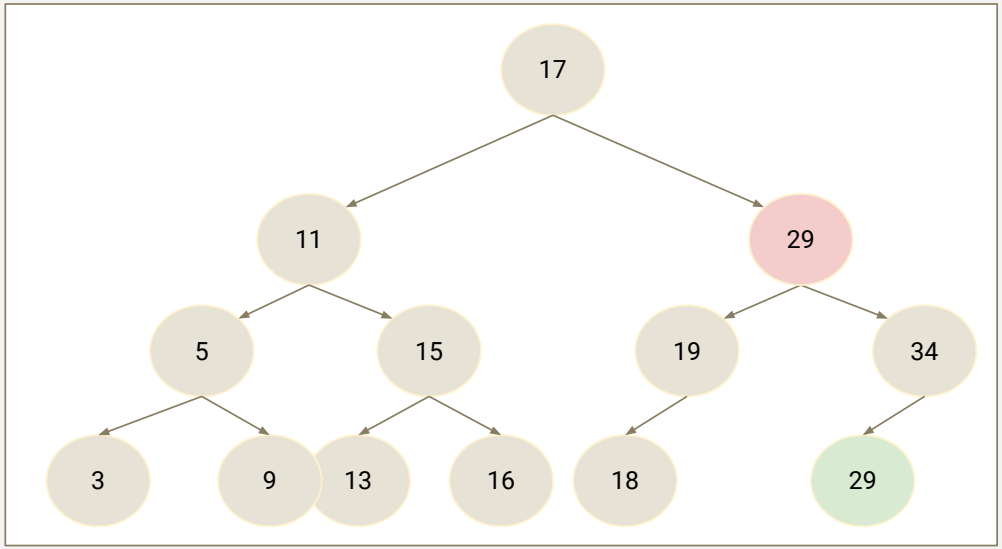
Next?

- Copy content of successor to node we want to delete
- Delete successor

what if 29 had 2 children? NO

Even if it had 1 child ⇒ Is it going to be left or right?

Case 3: Deleting non-leaf nodes with two children



Case 3:

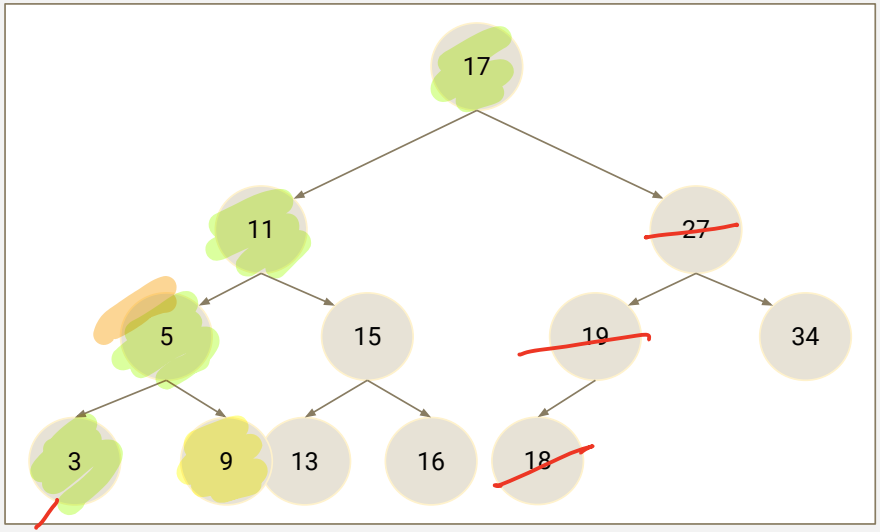
```
//Find the minimum in the right subtree of root(successor)


//copy successor's data to root's data

//BST_delete successor node
```

Tree Traversal

- Iterating over all nodes in some manner
- BST is a self-referential (recursively defined) data structure, hence traversal can be defined using recursion
- Unlike linear data structures, BST can be traversed in multiple ways:
  - In-order
  - Pre-order
  - Post-order

In-order traversal

(left child, root, right child)



Algorithm ***in-order(tree)***

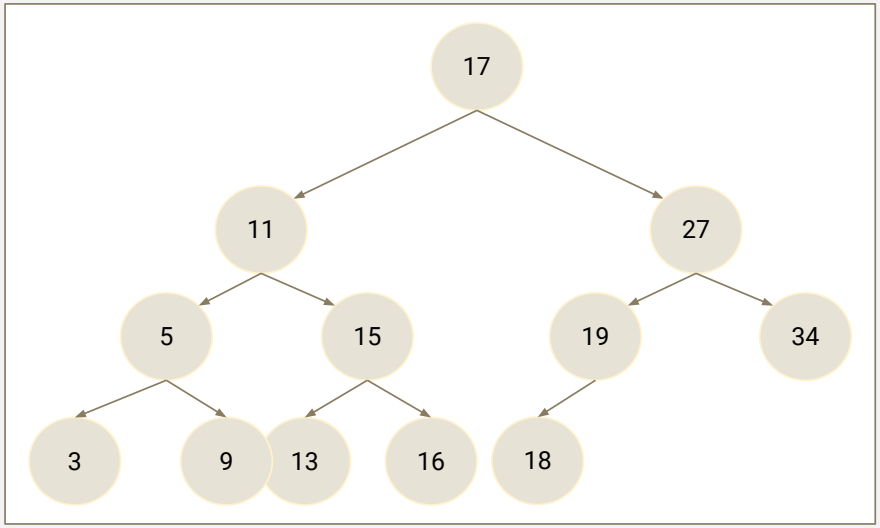
1. Traverse the left subtree ***in-order(left-subtree)***
2. Visit the root.
3. Traverse the right subtree ***in-order(right-subtree)***

in-order(17)
↓
visit 17
↓
in-order(27)
↓
in-order(19)
↓
in-order(18)

Prefix expression

+AB

Pre-order traversal (Root then children)



Algorithm *preorder(tree)*

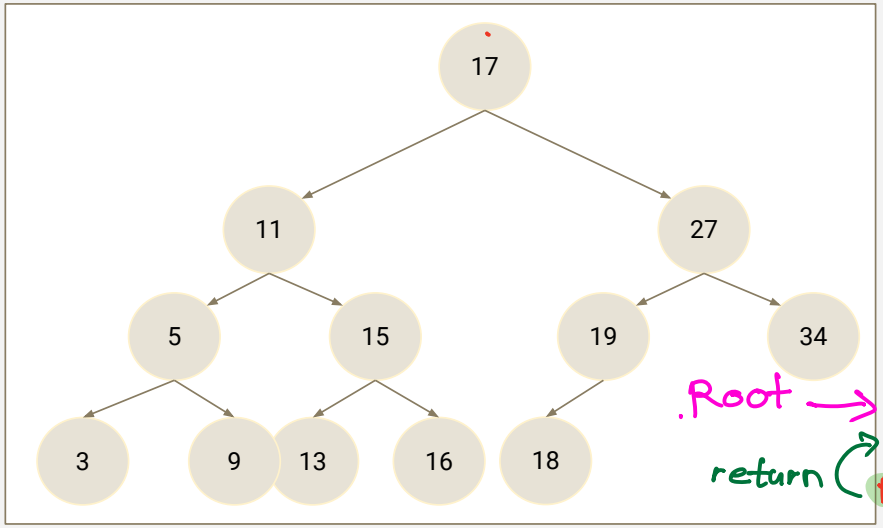
1. Visit the root
2. Traverse the left subtree *preorder(left-subtree)*
3. Traverse the right subtree *preorder(right-subtree)*

17 11 5 3 9 15 13 16 27 19 18 34

Pre-order is used to copy the tree

Post-order traversal

(children first then root)



Algorithm *post-order(tree)*

1. Traverse the left subtree *post-order(left-subtree)*
2. Traverse the right subtree *post-order(right-subtree)*
3. Visit the root

PO(17)
PO(11)
PO(5)
Root
PO(3)
PO(9)
return
PO(N)
PO(N)

3 9 5 13 16 15 11 18 19 34 27 17

Post-order is also used to delete the tree

Efficiency of search op^n using data structure

Have we solved the problem?

	Linked List	BST
Building the data structure	N items complexity of 1 item Insertion complexity : O(1) N items Insertion complexity N * O(1) = O(N) ↑ worst	N items complexity of 1 item Insertion complexity $O(\log_2 N)$ ↖ average N items $N * O(\log_2 N)$ $O(N \log_2 N)$ ← avg

Have we solved the problem?

	Array	BST
Sorting the Data Structure	Bubble sort Quicksort $O(N \log_2 N)$ ⇑ average complexity	Building BST On average $O(N \log_2 N)$

Have we solved the problem?

	Sorted Array	BST	Linked List
Search complexity	Binary Search $O(\log_2 N)$ ⇑ worst	$O(\log_2 N)$ ⇑ average	$O(N)$ ⇑ worst case
	∴ BST is not always the winner		

Have we solved the problem?

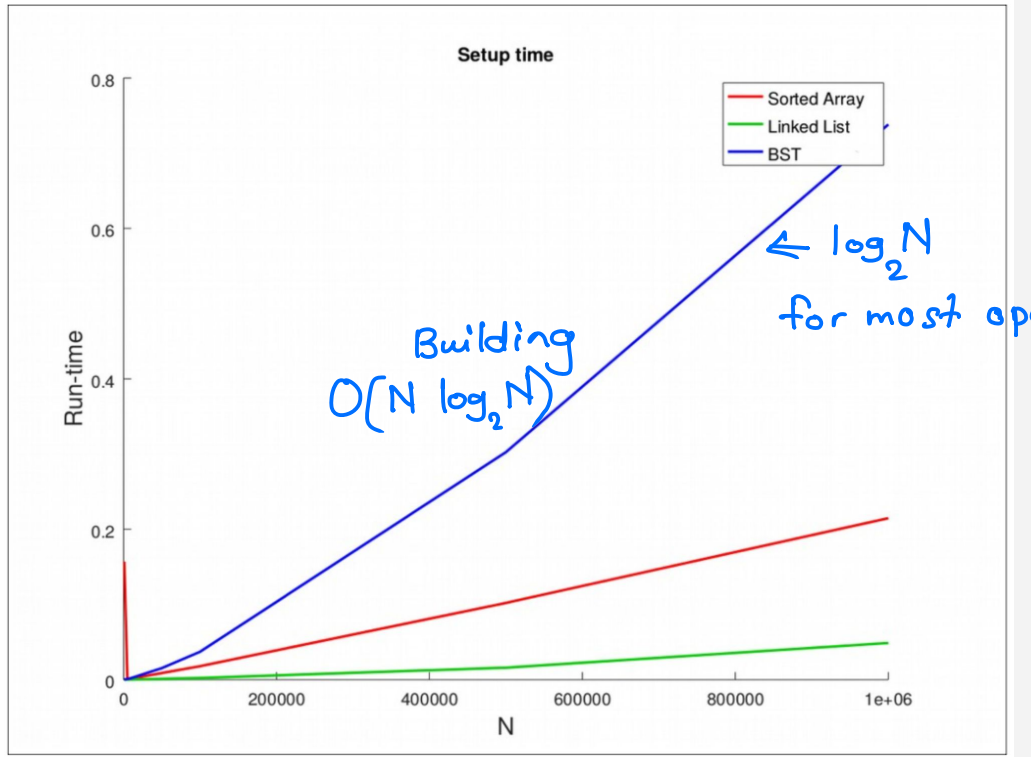
	Sorted Array	BST	Linked List
Storage	 if full copy & insert NOT GOOD OPTION	O(N) ⇓ dynamic ⇓ large cost ⇓ GOOD OPTION	O(N) ⇓ dynamic

How to choose?

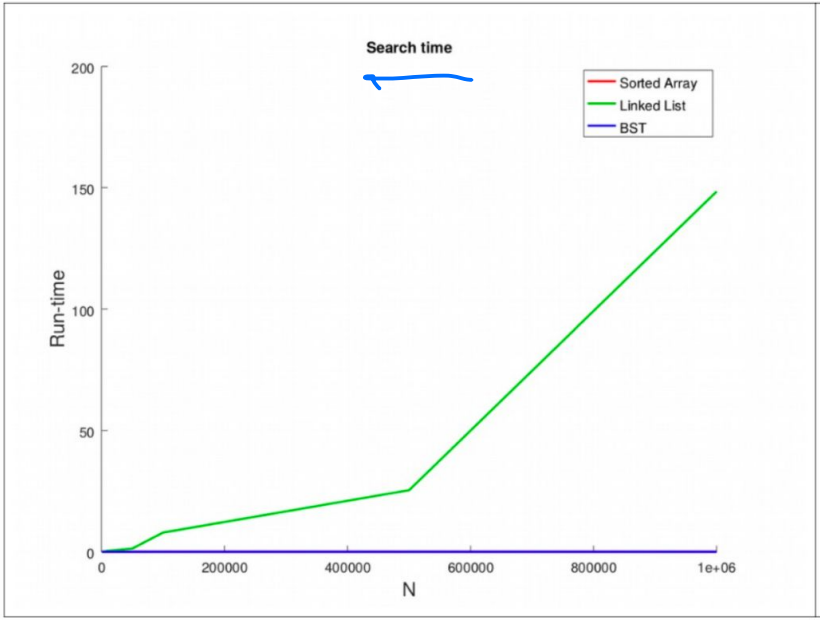
between all data structures?

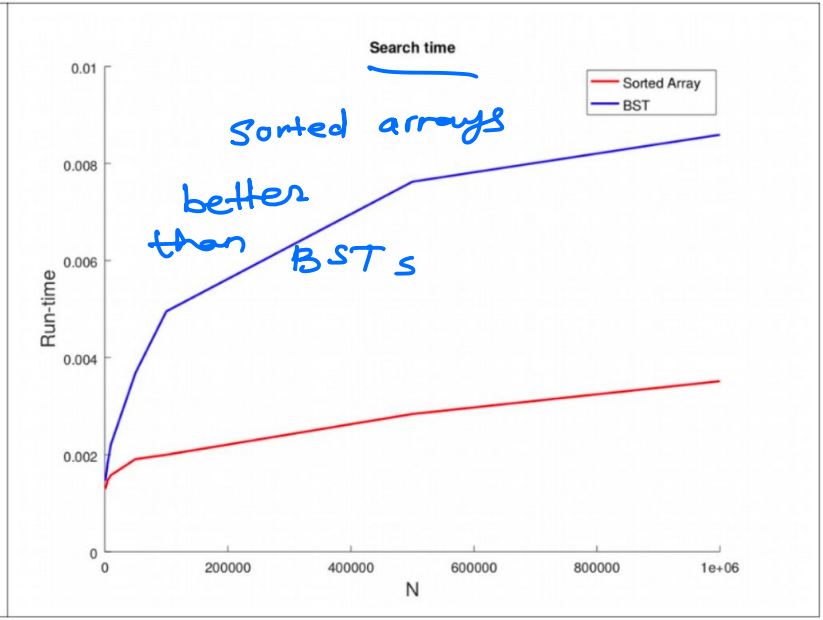
- Size of input(N)
  - Large N (best Big O complexity for common operations like insert, delete, and search) prob. BST
    - What kind of data? (for storage requirement)
      - Numeric data: (arrays not that bad!) wastage will be smaller
      - CDTs: (linked list or BSTs) wastage more
  - Smaller N
    - Ease of implementation
    - Performance of Data structure ⇒ may go for bubble sort
- Kind of operations
  - Access to all items all the time (linked lists or arrays)
  - Frequent insertions, search and deletions, (data structure with best big O for these operations)

(Sorted array+binary search) vs linked list vs BST

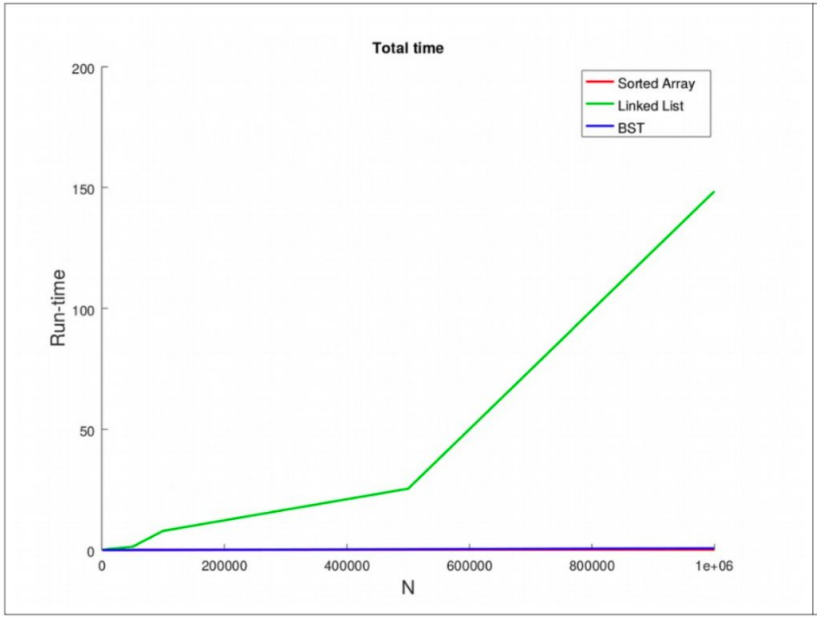


(Sorted array+binary search) vs linked list vs BST

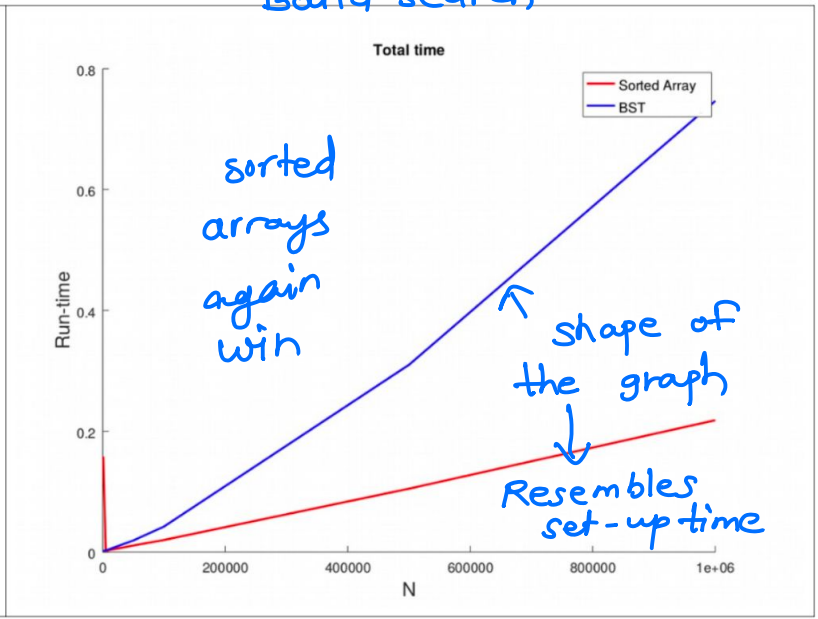




(Sorted array+binary search) vs linked list vs BST



Build+search



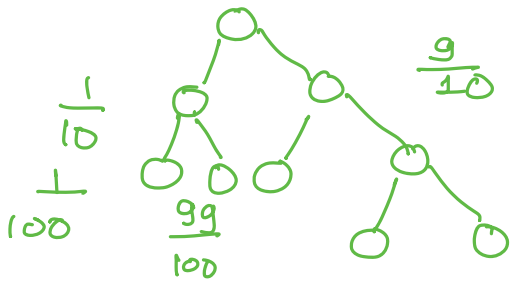
50 iterations for each (10,000) queries

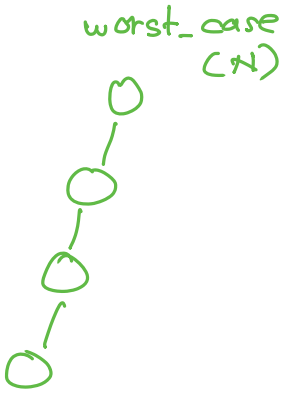
Unit 4 takeaways!

- How to measure complexity of algorithms, problems, data structures
  - Worst case complexity
  - Average case complexity
    - Not reliable for safety critical applications or real time applications
- How to choose an ADT for an application
  - Implementation of BST
  - Linked lists vs Arrays vs BSTs

average case is closer to worst case

$(\log_{10/7} N)$





$(\log_{10/7} N)$

$\Uparrow$ diff. base