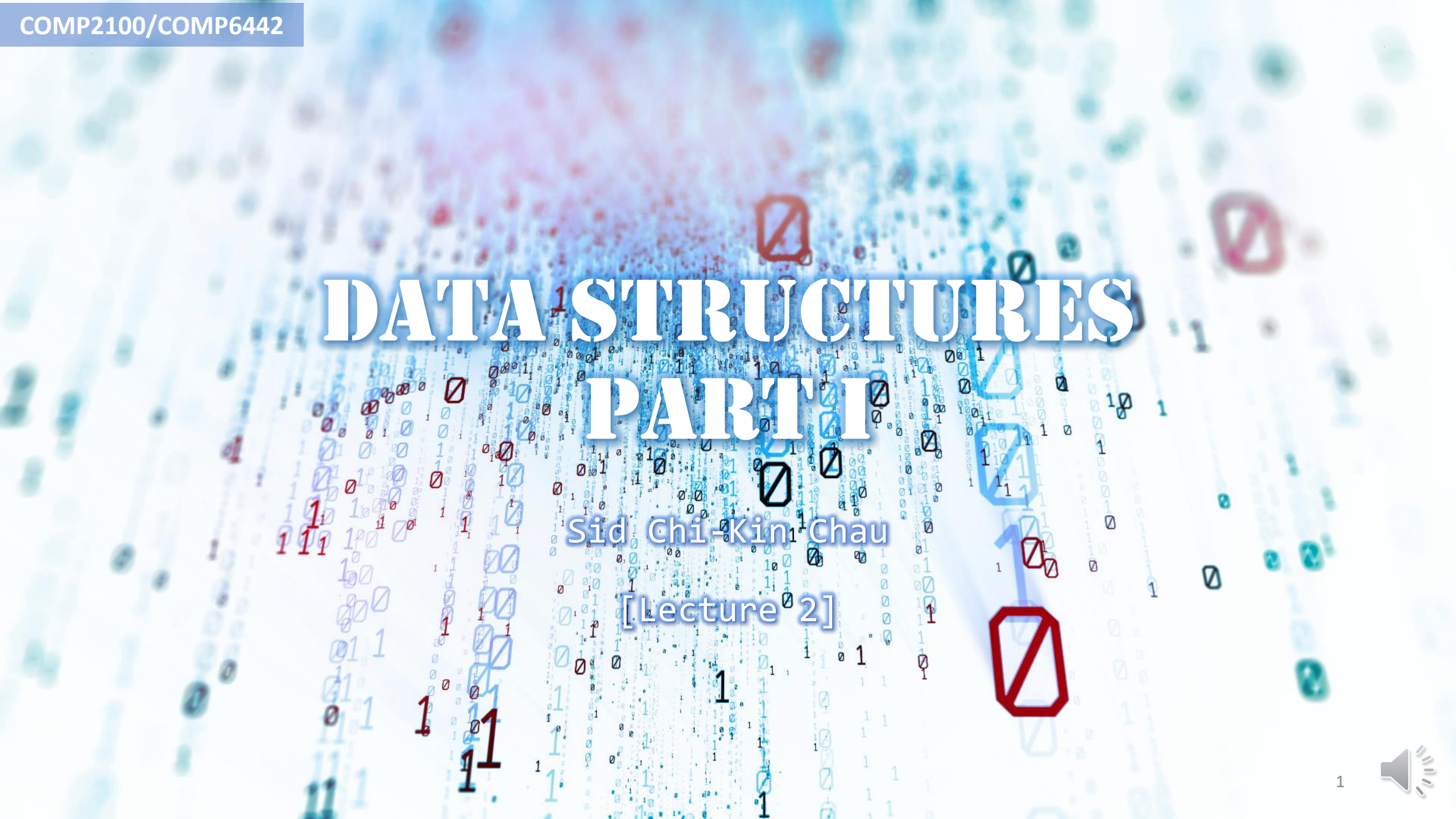


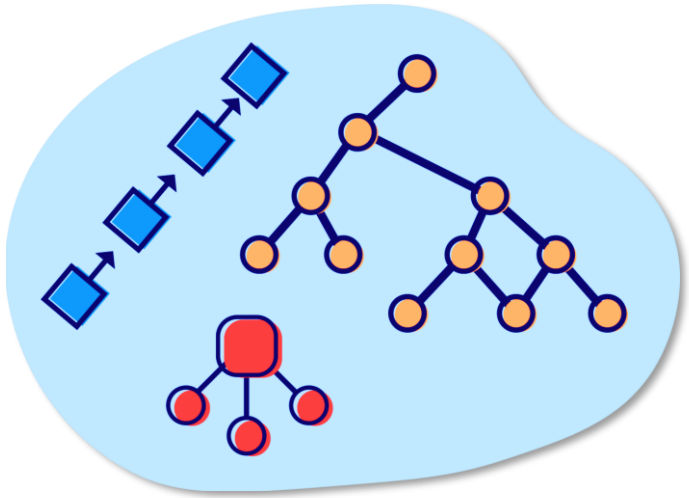
DATA STRUCTURES PART I

Sid Chi-Kin Chau

[Lecture 2]

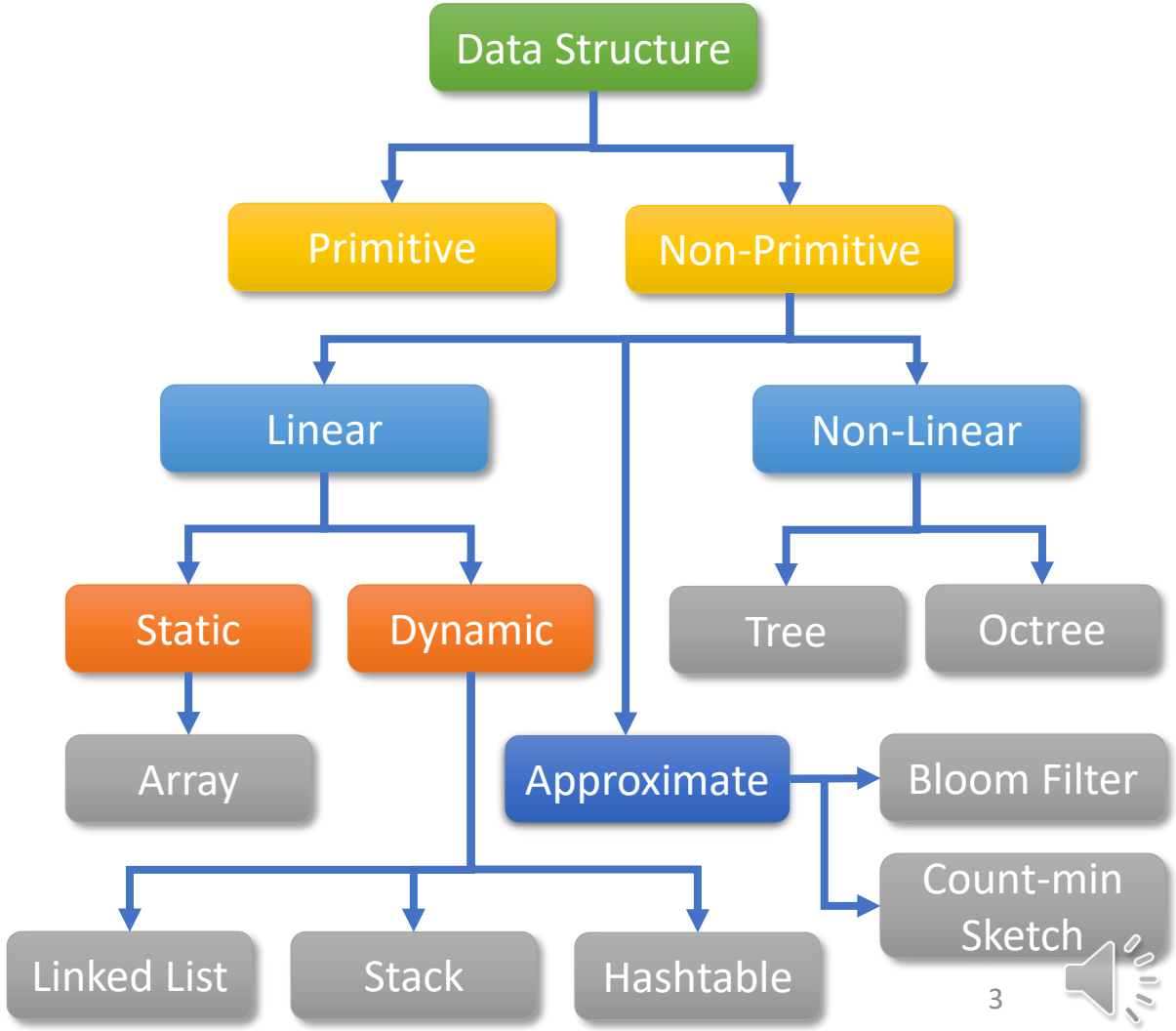
Why Data Structures

- No matter how efficient the programming language is, if the chosen data structure is not appropriate, the performance still suffers
  - Data Structures are universal!
- What is the purpose of data structures?
  - Data structures facilitate data management and retrieval
- What is the best data structure?
  - It can be a linked list or a tree, depends on applications
- Most common data structures in Java are available in the Java Collections Framework
  - `java.util package: Collection, List, Hashtable, TreeMap`



Types of Data Structures

- Primitive
  - Atomic/simplest form to store data/predefined by the language
  - Integer, float, char, string
- Non-primitive
  - Non-atomic/multidimensional
  - Linear
    - Array, linked list, stack, hashtable
  - Non-linear
    - Heap, tree, octree, graph
  - Approximate
    - Possible errors but space/time-efficient

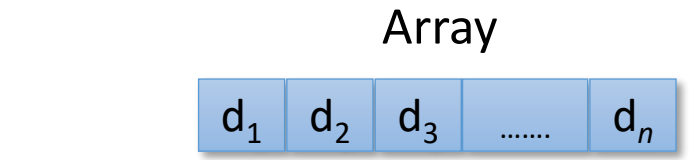


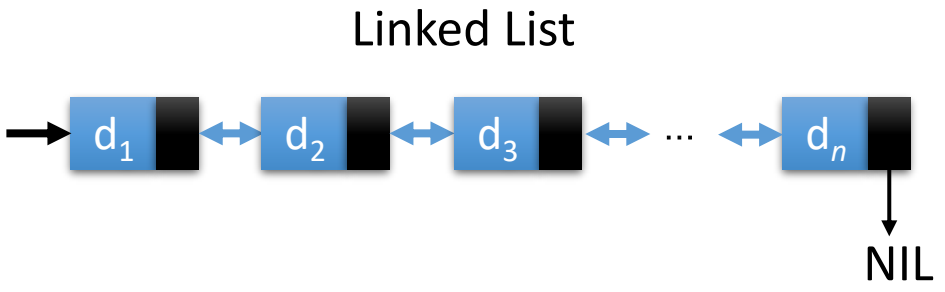
Goals of This Lecture

- Learn basic data structures
  - Linear data structures
    - Linked list, stack, hash table,
  - Approximate data structures
    - Bloom filter, count-min sketch
  - Non-linear data structures
    - Binary search tree
    - Search, traversing, successor and predecessor, insertion, deletion
- Something else to learn
  - Applications of different data structures

Linked List

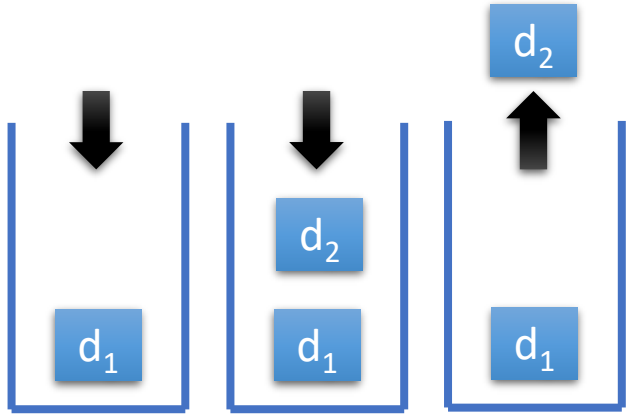
- Items are attached adjacently
- Traversing needs to be sequential
  - Items are arranged in sorted sequence
- Linked list can grow dynamically
- Use a pointer to identify the next and previous items in the list
  - Arrays need one contiguous memory block
- Do not need a contiguous memory block
- Inefficient searching
  - Need to at most look at every item



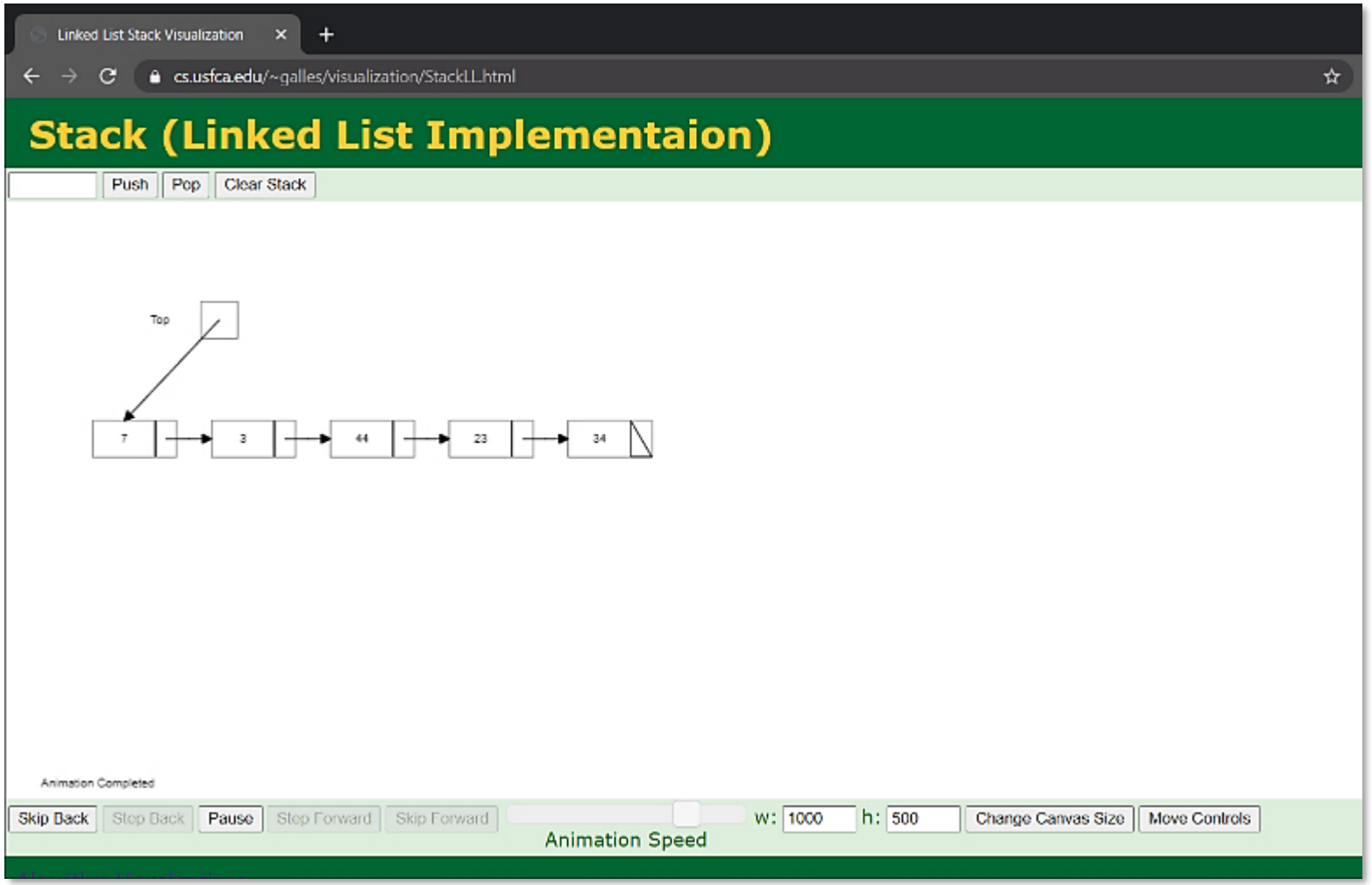


Stack

- Items are stored in a stack
  - Items are **popped** in and **pushed** out of the stack in a last-in-first-out manner
- Use a pointer to the next item in the stack
- Do not need a contiguous memory block
- Inefficient searching
- Applications:
  - Syntax parsing
  - Backtracking (for depth-first search)



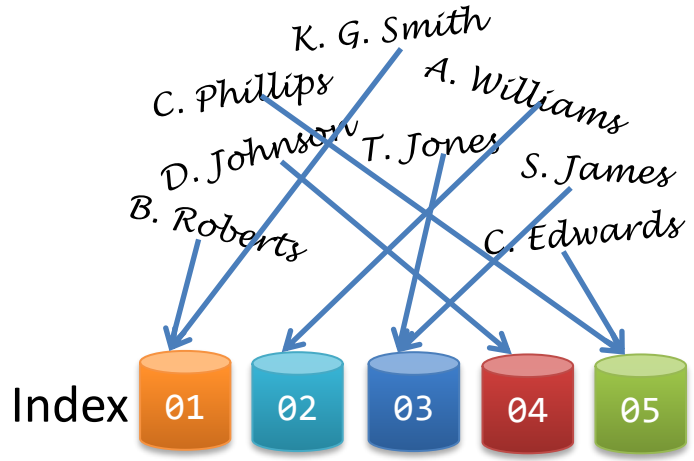
Demo



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

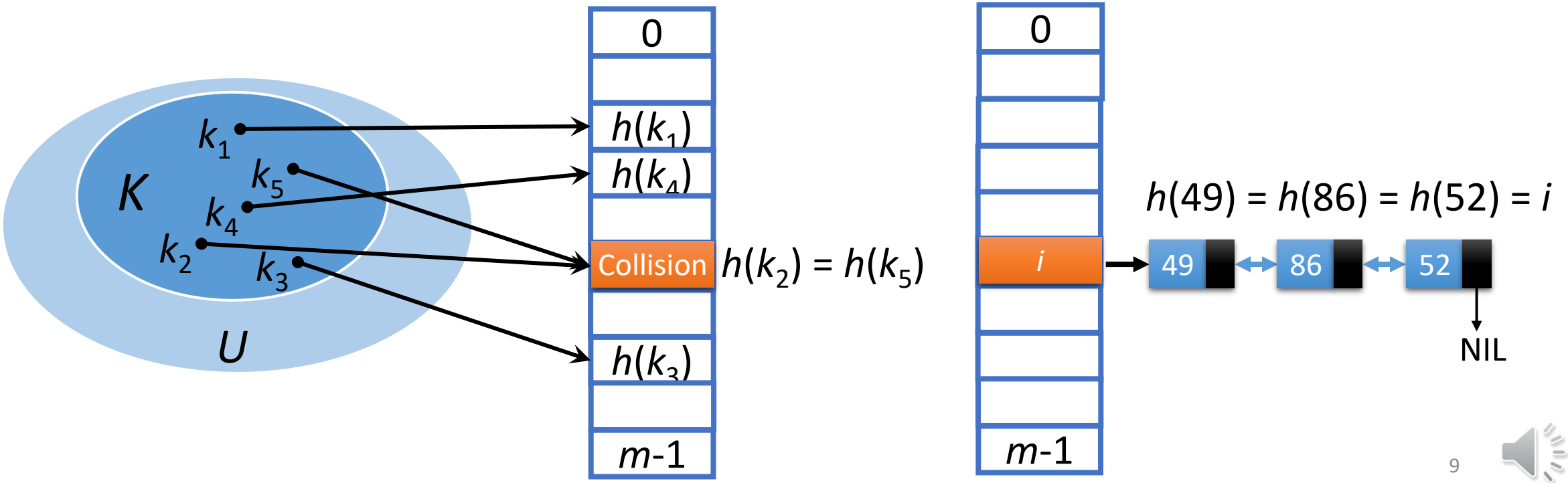
HashTable

- Items are mapped to indexes by a hash function deterministically
  - The number of indexes is a constant, independent of the number of items
- Items with the same index are stored together (e.g., in a linked list)
  - Increase searching efficiency
- Common hash functions
  - Universal hash, one-way hash (SHA-2 (SHA-256))
- Collision-resistant hash function
  - Each item has a uniform probability to be mapped to any one of the indexes



Hash Function

- Hash function h to map the universe U of all keys into {0, 1, …, m-1}:
  - Collision: When an item to be inserted maps to an already occupied slot
  - Resolving collisions by chaining using listed links



Hash Function

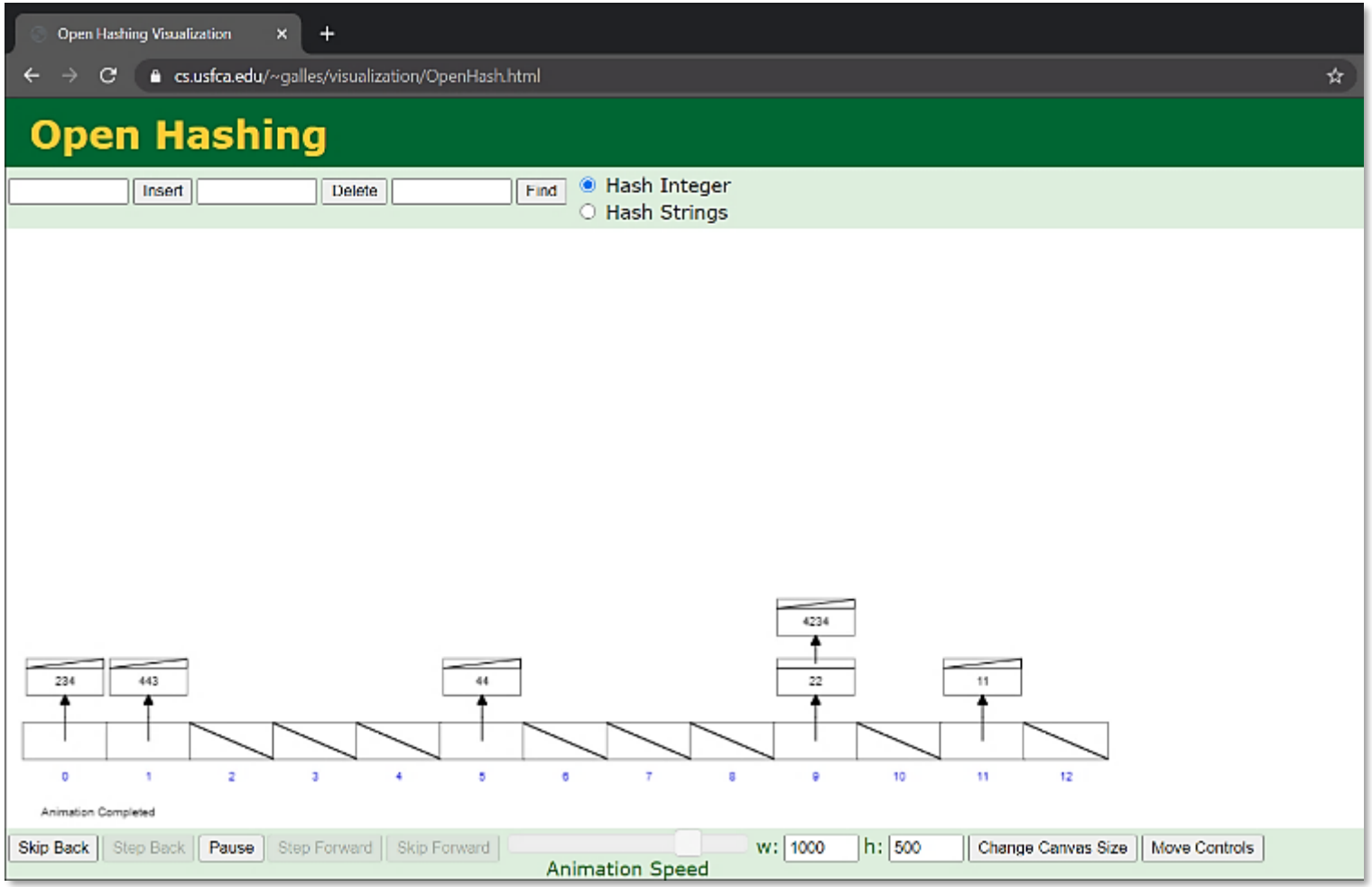
- Assume all keys are integers, and define hash function:

$$h(k) = k \bmod m$$

  - Caveat: If $m = 2^r$, then the hash doesn't even depend on all the bits of k:
    - If $k = 1011000111011010$ and $r = 6$, then $h(k) = 011010$
- Randomized strategy
  - Let m be prime. Decompose key k into $r + 1$ integers, each with value in the set {0, 1, …, m-1}. That is, let $k = \langle k_0, k_1, \ldots, k_r \rangle$, where $0 \leq k_i < m$
    - For example, $k_i = k \bmod m_i$, for some $0 \leq m_i < m$
  - Pick $a = \langle a_0, a_1, \ldots, a_r \rangle$ where each a_i is chosen randomly from {0, 1, …, m–1}.
  - Define hash function:

$$h_a(k) = \sum_{i=0,\ldots,r} a_i k_i \bmod m$$

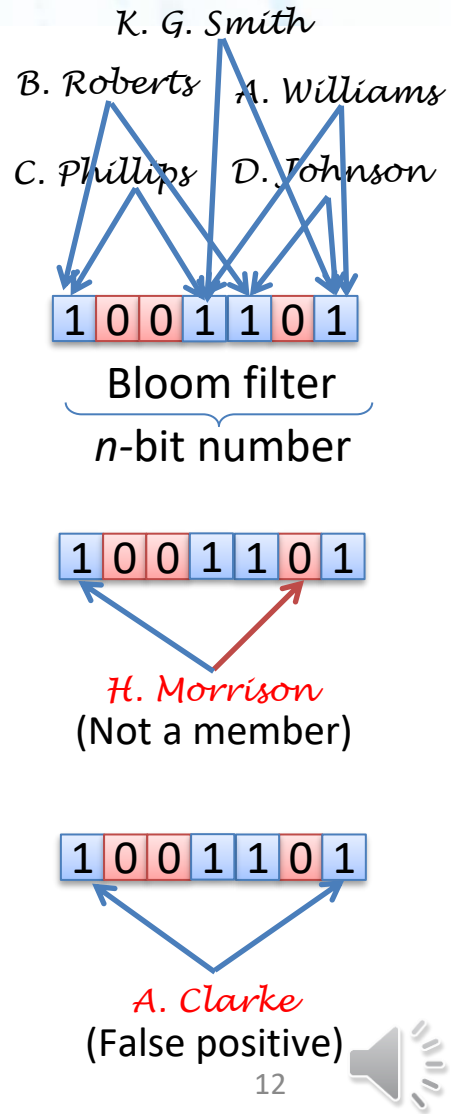
Demo



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

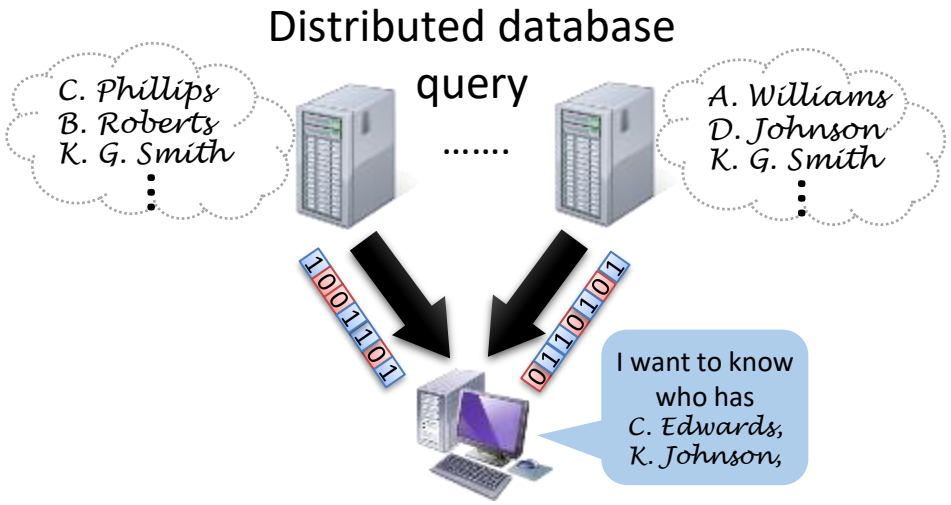
Bloom Filter: Approximate Data Structure

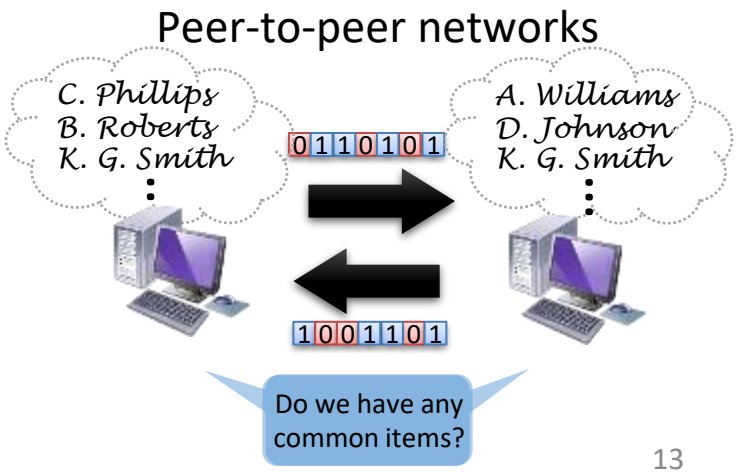
- Bloom filter is compact representation of a set of strings
  - Bloom filter is an n-bit string, initially set all zeros
  - For the k-th hash function, h_k maps a member string to a value in {1,..,n}
  - If we want include a string s in the Bloom filter, we set the $h_k(s)$-th bit in the Bloom filter to be one for every k
  - To validate whether a string s is in a Bloom filter, we check if the $h_k(s)$-th bit in the Bloom filter is one for every k
- A string not in the Bloom filter cannot be a member
  - There is no false negative
- But a string that is not a member may be in the Bloom filter
  - There may be false positive



Applications of Bloom Filter

- False positives can be reduced by increase the size of Bloom filter
- Bloom filter is useful to applications that can tolerate minor false positives:
  - Spell and password checkers with a set recorded words
  - Distributed database query
  - Content distribution and web cache
  - Peer-to-peer networks
  - Packet filtering and measurement of pre-defined flows

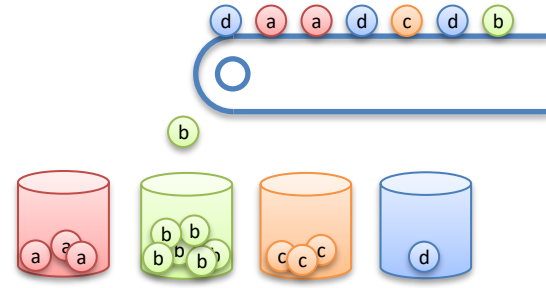


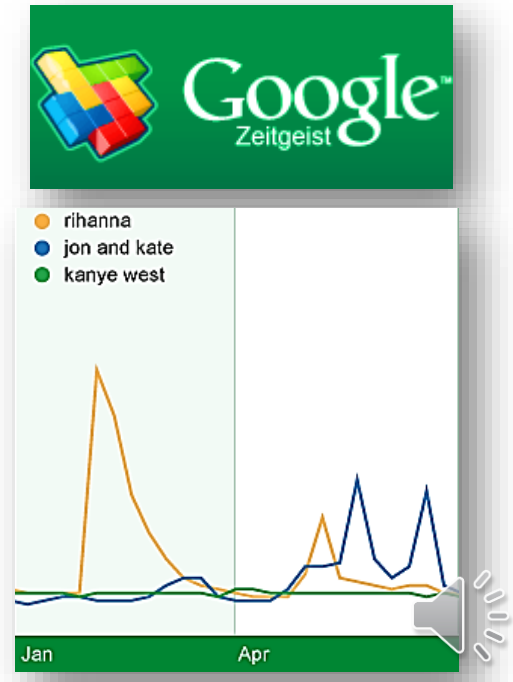


Count-min Sketch

- Find the most frequent items in a stream
  - Find the most purchased items from the transactions of a supermarket
  - Find the users who consume the most bandwidth by observing a stream of packets in the network
  - Find the most queried keywords in a search engine
- Heavy hitter problem
  - There is a stream of items with multiple occurrences
  - We want to find the items with the most occurrences, when observing the stream continuously
  - We do not know the number of distinct items
  - We are only allowed to use compact storage space

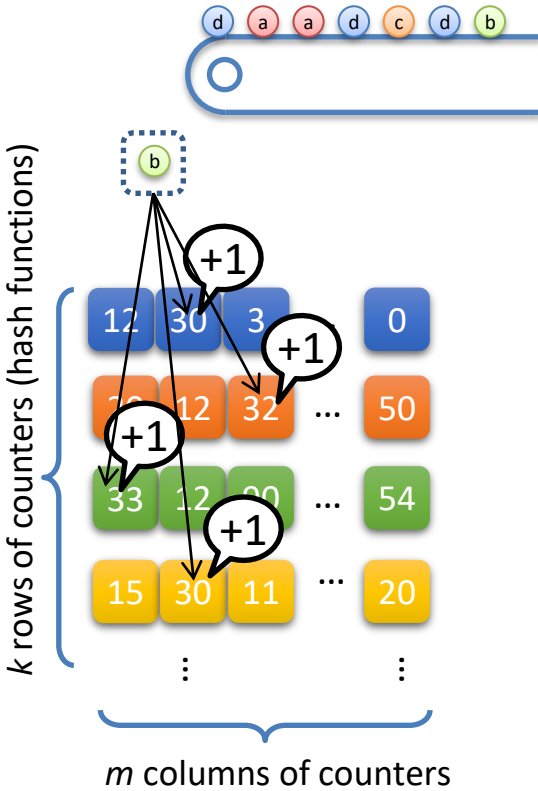
A stream of items with multiple occurrences





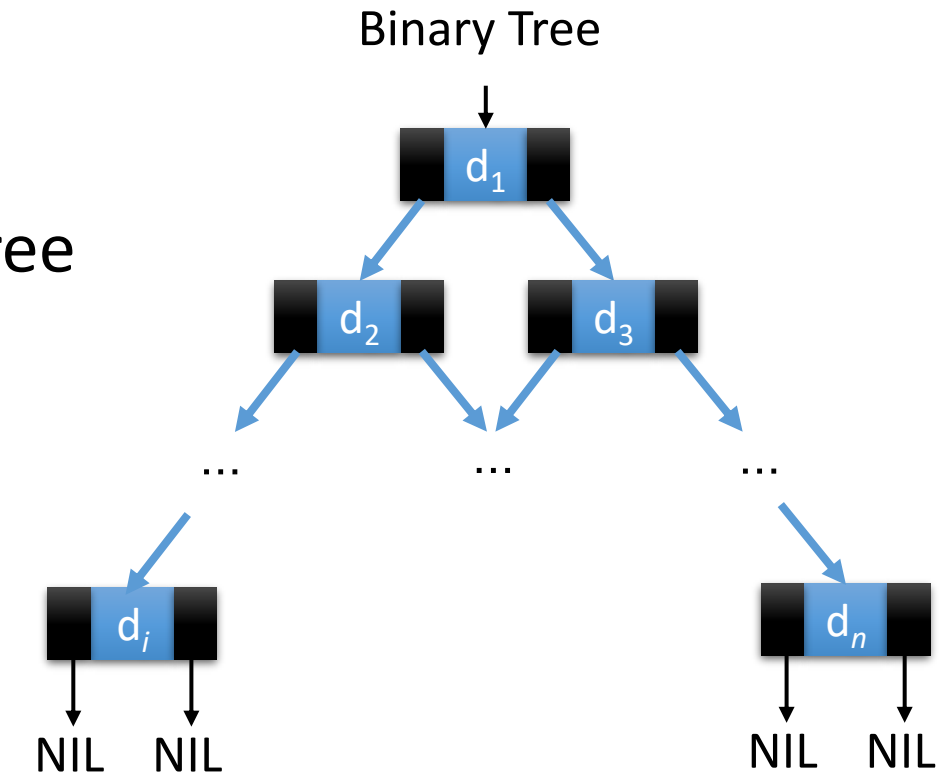
Count-min Sketch

- Count-min sketch can track approximate occurrences
  - A sketch is an array of *k*×*m* counters $\{C_{i,j}\}$
  - There are *k* hash functions, each *i*-th hash function h_i maps an item to a value in {1,..,*m*}
  - Initially set all counters to be zero ($C_{i,j} = 0$)
  - When we observe an item *s* in the stream, increase the $h_i(s)$-th counter ($C_{i,h_i(s)} \leftarrow C_{i,h_i(s)}+1$) for every *i*
  - At the end, we obtain the number of occurrences of an item *s* by the minimum of all the counters that are mapped by *s* as $N(s) = \min\{C_{i,h_i(s)} : i = 1, \ldots, k\}$
  - *N*(*s*) is an estimate of the true number of occurrences
    - Because multiple items can be mapped to the same counter by a hash function

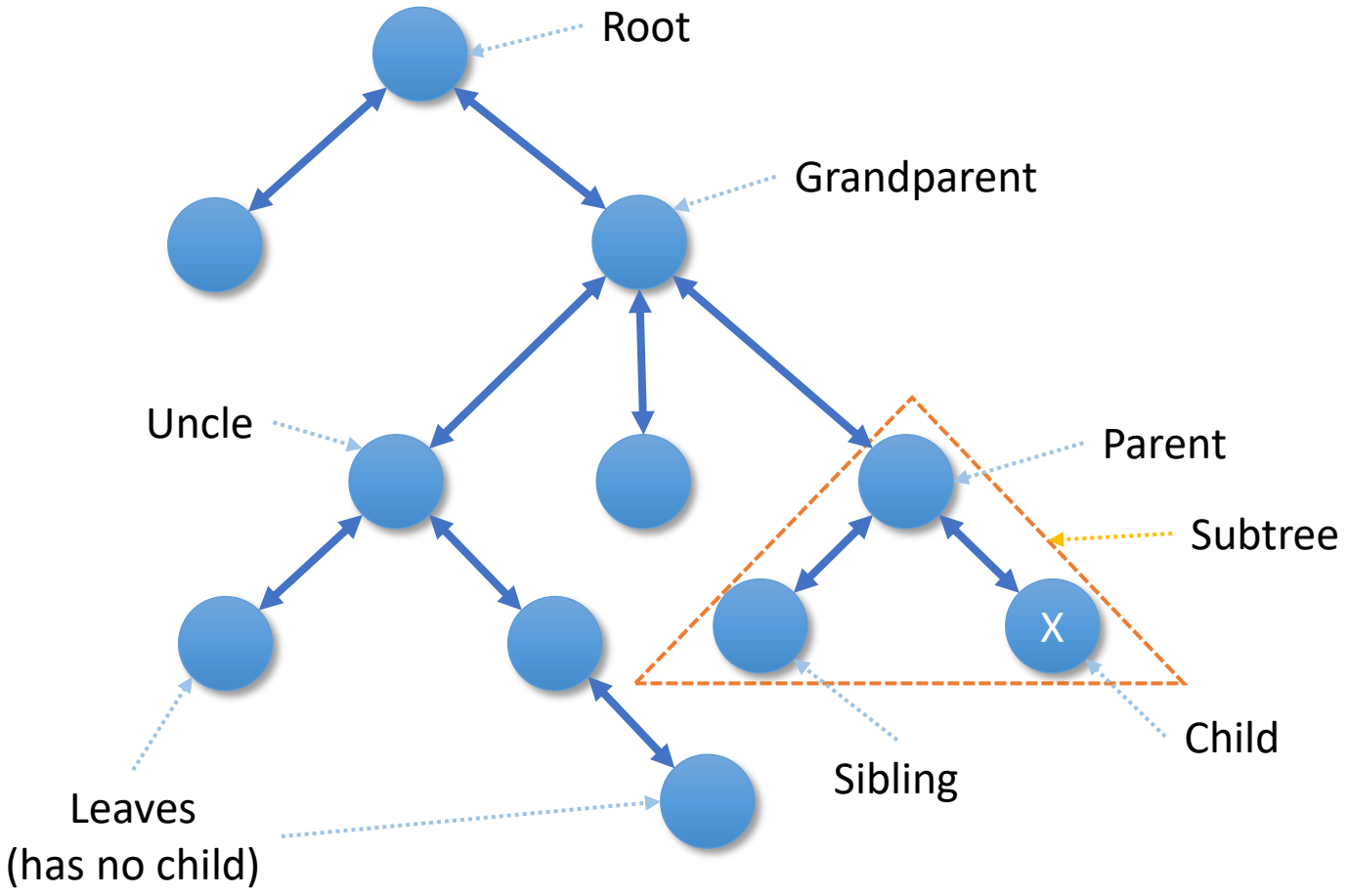


Tree

- Items are connected to two or more items
- Not organized in a sequence
- Use pointers connect to other items in a tree
  - No self-loop nor cycle
- More complicated implementation
- Can grow dynamically
- Can be more efficient for searching
  - Require sorting in place
- Hierarchical data format (e.g., XML)



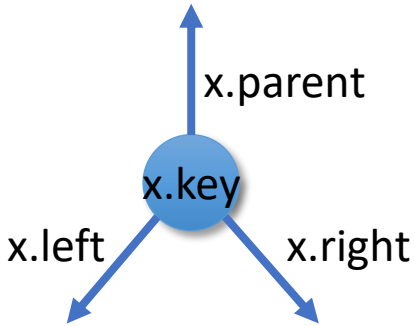
Tree



Height
(the number of **edges** on the longest path between root and a leaf)

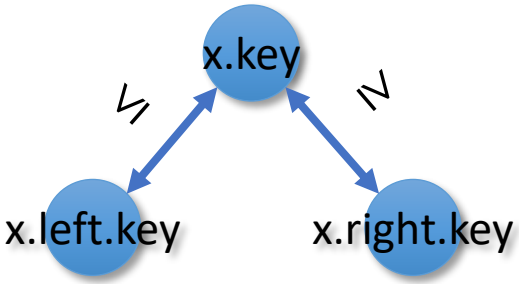
Tree Representation

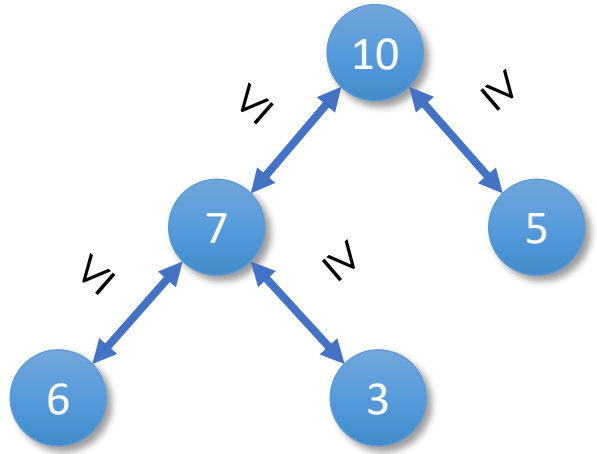
- Represented by a linked data structure of nodes
- T.root points to the root of tree T
- Each node contains fields:
  - key (and data)
  - left: pointer to left child: root of left subtree
  - right: pointer to right child: root of right subtree
  - parent: pointer to parent
    - Root: T.root.parent = NIL



Heap

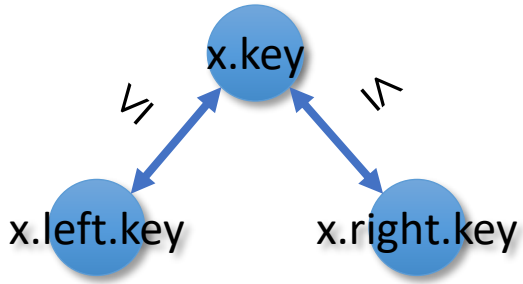
- Heap is a binary tree (with at most two children), satisfying the heap property:
  - Keys of the child nodes are smaller than the key of the parent node
    - x.left.key ≤ x.key and x.right.key ≤ x.key
  - Some similarity and difference with binary search tree. Don't confuse with binary search tree!
- Applications
  - Priority queue
  - Sorting

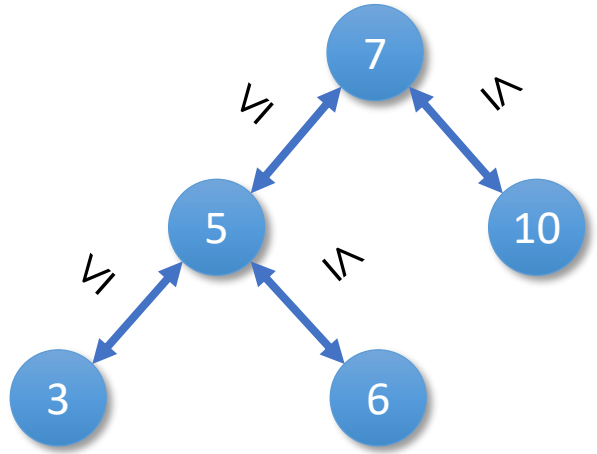




Binary Search Tree (BST)

- Nodes of (key, value) pairs (e.g. phone book)
  - Keys can be sorted (e.g. numbers, strings)
  - Edges relate the keys
- BST: Tree with at most two children for each node
  - Key of the left child node is smaller than that of its parent node
  - Key of the right child node is greater than that of its parent node
- Can support dynamic set operations
  - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete





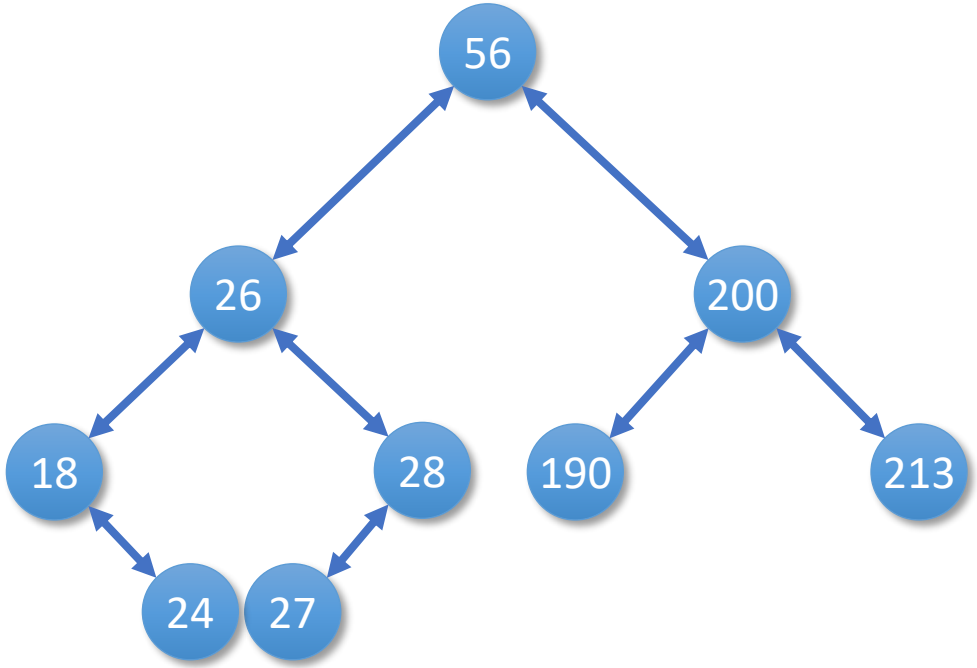
Also for children of children

smaller or equal

Keys can be unique or not

BST Property

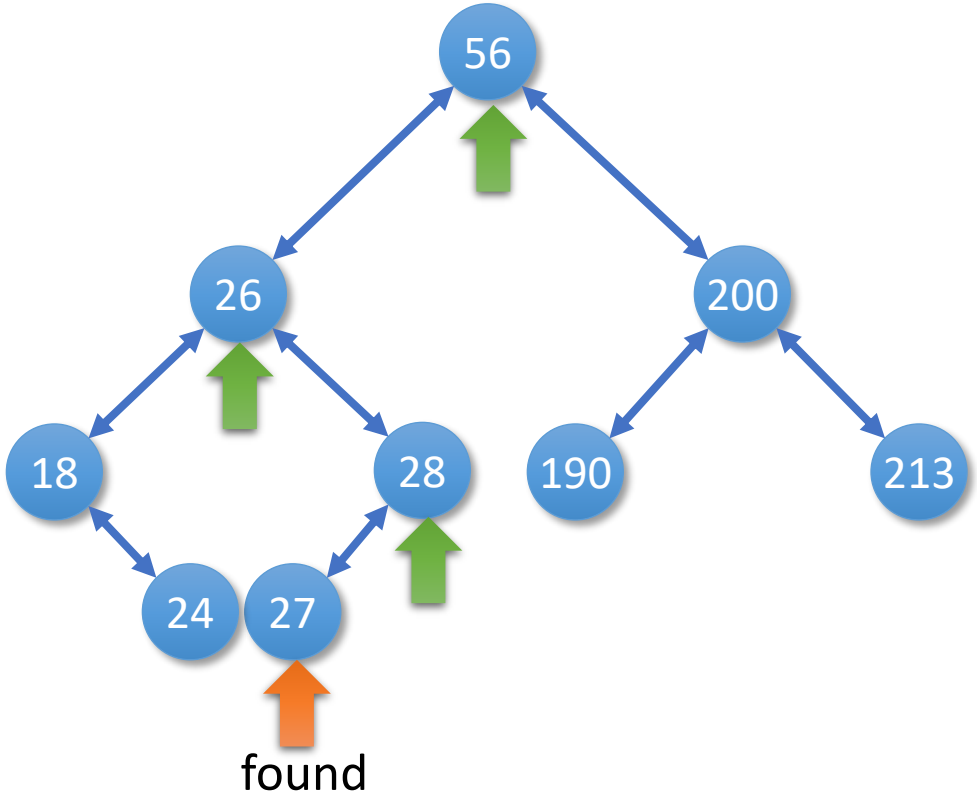
- Stored keys must satisfy the binary search tree property
- Let x be a node in a BST
- If y is a node in the left subtree
  - For all y in left subtree of x, then $y.key \leq x.key$
- If y is a node in the right subtree
  - For all y in right subtree of x, then $y.key \geq x.key$



Searching BST

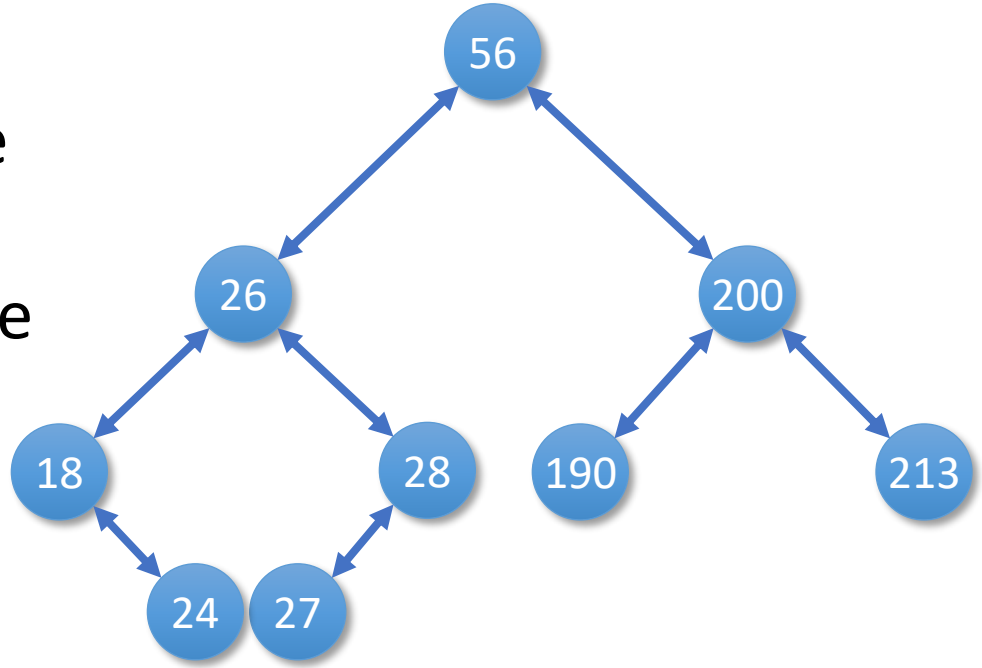
```
BST-Search[x,Key]
If x ≠ null Then
  If x.key = Key Then
     Return x
  Else If x.key > Key Then
     BST-Search[x.left, Key]
  Else
     BST-Search[x.right, Key]
Return null
```

- Recursive searching with a given key
  - If the given key is smaller than that of the current node, search the left subtree
  - If the given key is larger than that of the current node, search the right subtree



Traversing BST

- How to enumerate all the nodes (traverse tree from the root)?
  - It is used to print out the data in a tree in a certain order
  - Or apply some operations to each node
- Recursive traversing methods:
  - In-order traversing
  - Pre-order traversing
  - Post-order traversing

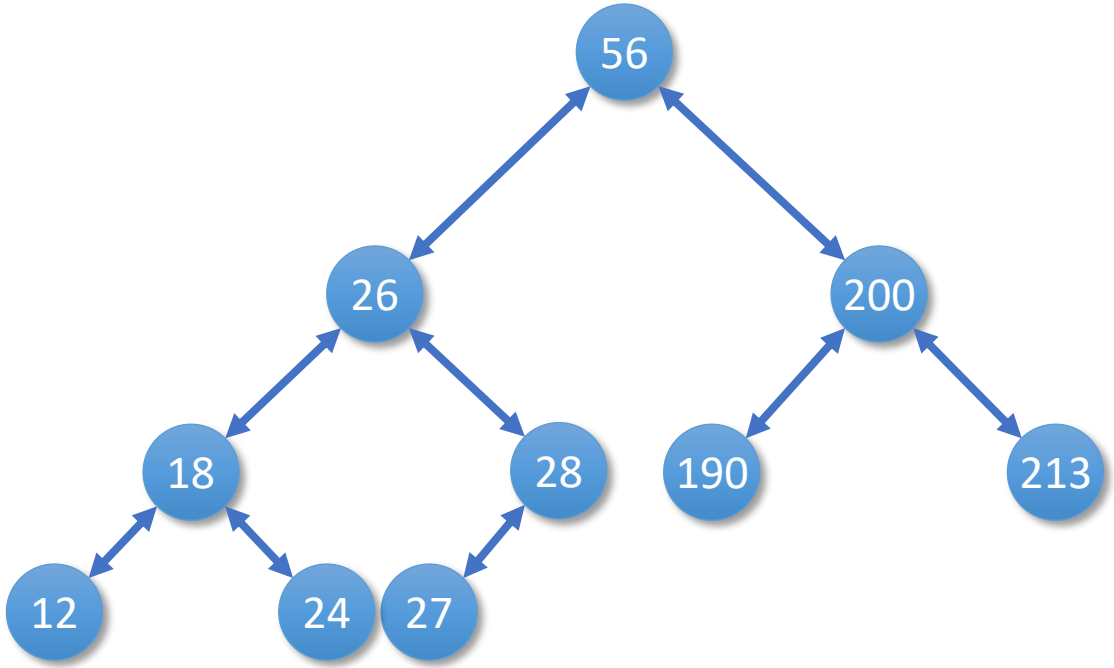


Order： the order of the keys. In order should be from the smallest to the largest.

In-order Traversal

```
Inorder-BST-Walk[x]
If x ≠ null Then
  Inorder-BST-Walk[x.left]
  Print x.key
  Inorder-BST-Walk[x.right]
```

Output:
12, 18, 24, 26, 27, 28, 56, 190, 200, 213

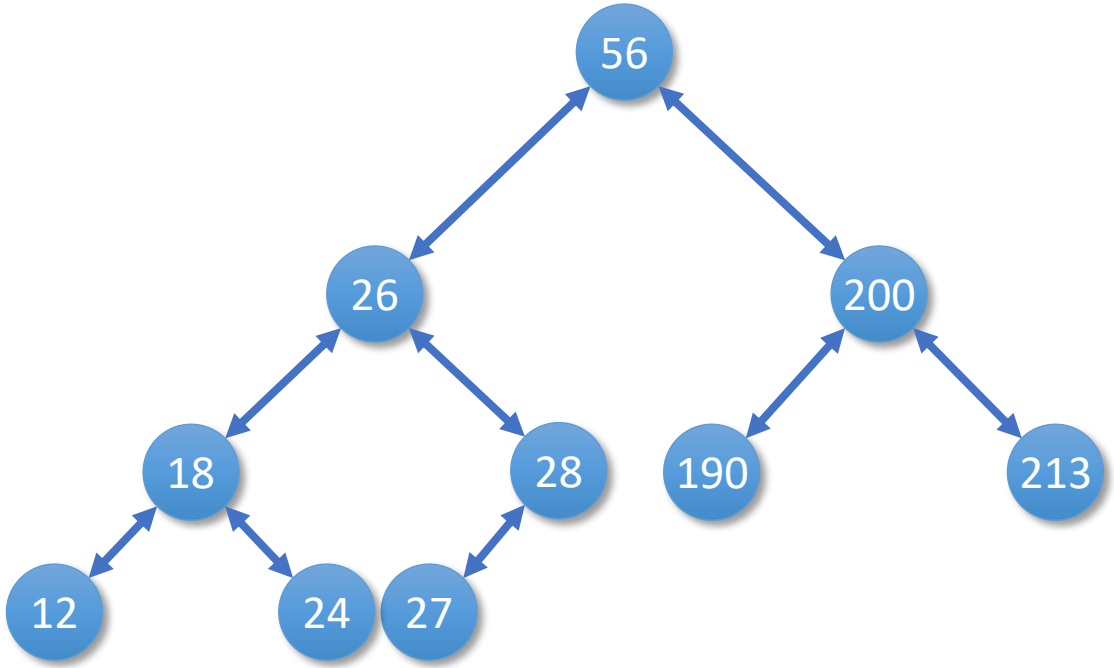


- Inorder-BST-Walk[56]; //key=56,x≠null
- Inorder-BST-Walk[26]; //left=26,x≠null
- Inorder-BST-Walk[18]; //left=18,x≠null
- Inorder-BST-Walk[12]; //left=12,x≠null
- Inorder-BST-Walk[null]; //left=null
- Print 12; //output
- Inorder-BST-Walk[null]; //right=null
- ……

Pre-order Traversal

```
Preorder-BST-Walk[x]
If x ≠ null Then
  Print x.key
  Preorder-BST-Walk[x.left]
  Preorder-BST-Walk[x.right]
```

Output:
56, 26, 18, 12, 24, 28, 27, 200, 190, 213

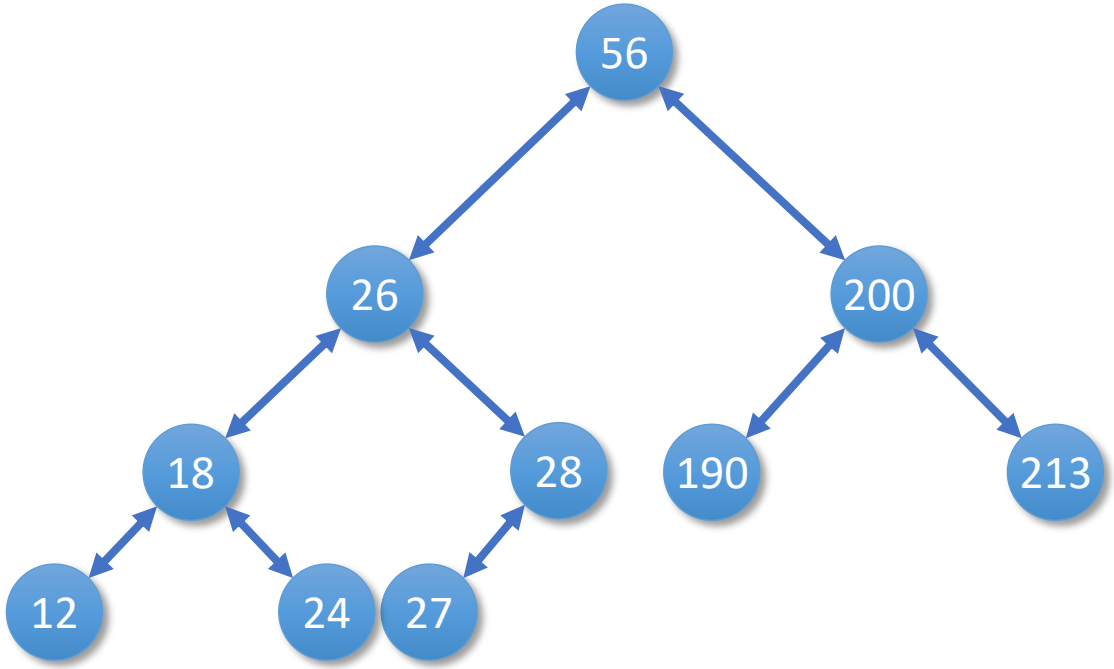


- Preorder-BST-Walk[56]; //key=56,x≠null
- Print 56; //output
- Preorder-BST-Walk[26]; //left=26,x≠null
- Print 26; //output
- Preorder-BST-Walk[18]; //left=18,x≠null
- Print 18; //output
- Preorder-BST-Walk[12]; //left=12,x≠null
- ……

Post-order Traversal

```
Postorder-BST-Walk[x]
If x ≠ null Then
  Postorder-BST-Walk[x.left]
  Postorder-BST-Walk[x.right]
  Print x.key
```

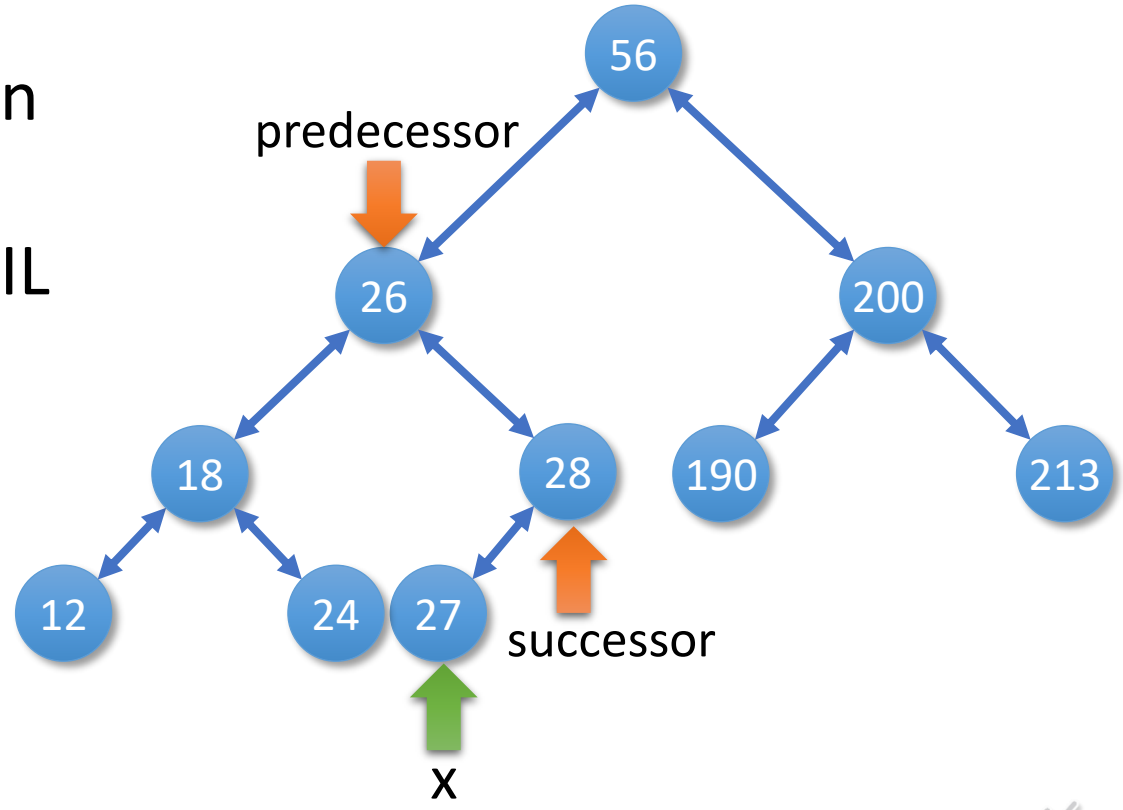
Output:
12, 24, 18, 27, 28, 26, 190, 213, 200, 56



- Postorder-BST-Walk[56]; //key=56,x≠null
- Postorder-BST-Walk[26]; //left=26,x≠null
- Postorder-BST-Walk[18]; //left=18,x≠null
- Postorder-BST-Walk[12]; //left=12,x≠null
- Postorder-BST-Walk[null]; //left=null
- Postorder-BST-Walk[null]; //right=null
- Print 12; //output
- Postorder-BST-Walk[24]; //right:24,x≠null ……

Successor and Predecessor

- Successor
  - Node y is the successor of node x, if y.key is the **smallest key greater** than x.key
  - The successor of the largest key is NIL
- Predecessor
  - Node y is the predecessor of node x, if y.key is the **largest key smaller** than x.key
  - The predecessor of the smallest key is NIL



Finding Successor

- Finding the successor consists of two cases

Case 1: If node x has a non-empty right subtree, then

  - x's successor is the minimum in the right subtree of x

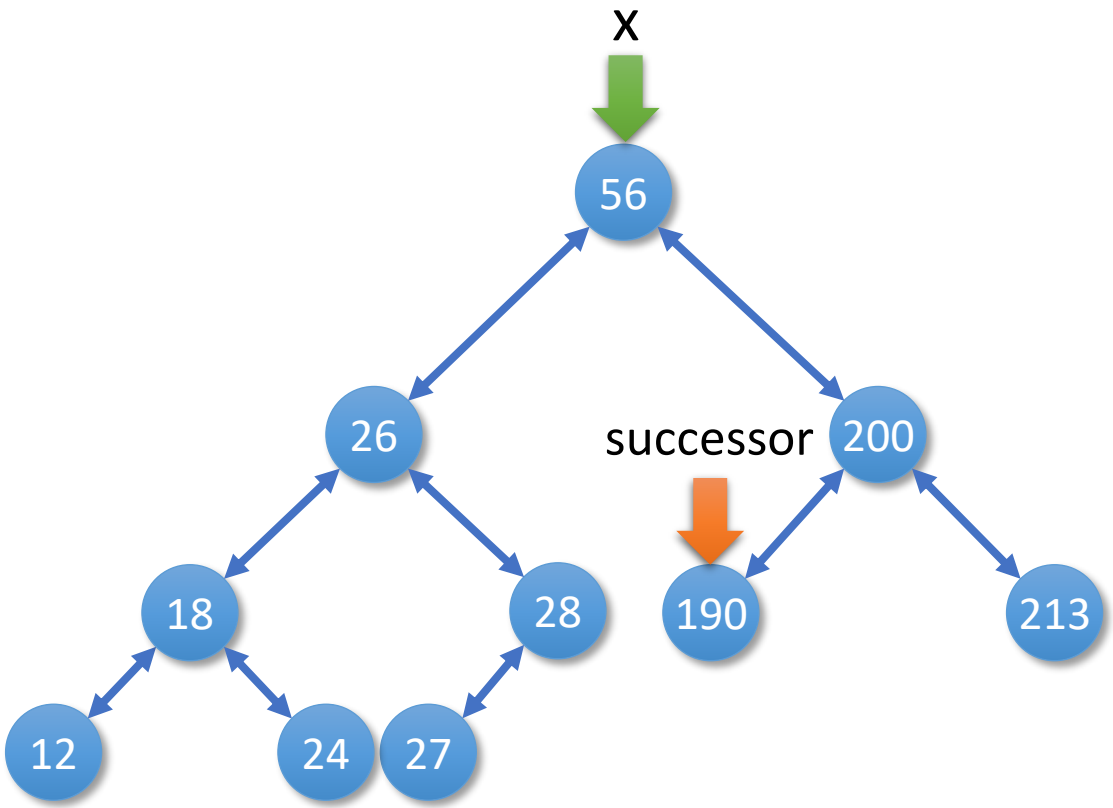
Case 2: If node x has an empty right subtree, then

  - As long as we move up the tree, we can find bigger keys
  - In other words, x's successor y, is the lowest ancestor of x whose left child is also an ancestor of x

Finding Successor

```
BST-Successor[x]
If x.right ≠ null Then
    Return BST-Minimum[x.right]
y ← x.parent
While y ≠ null and x = y.right do
    x ← y
    y ← y.parent
Return y
```

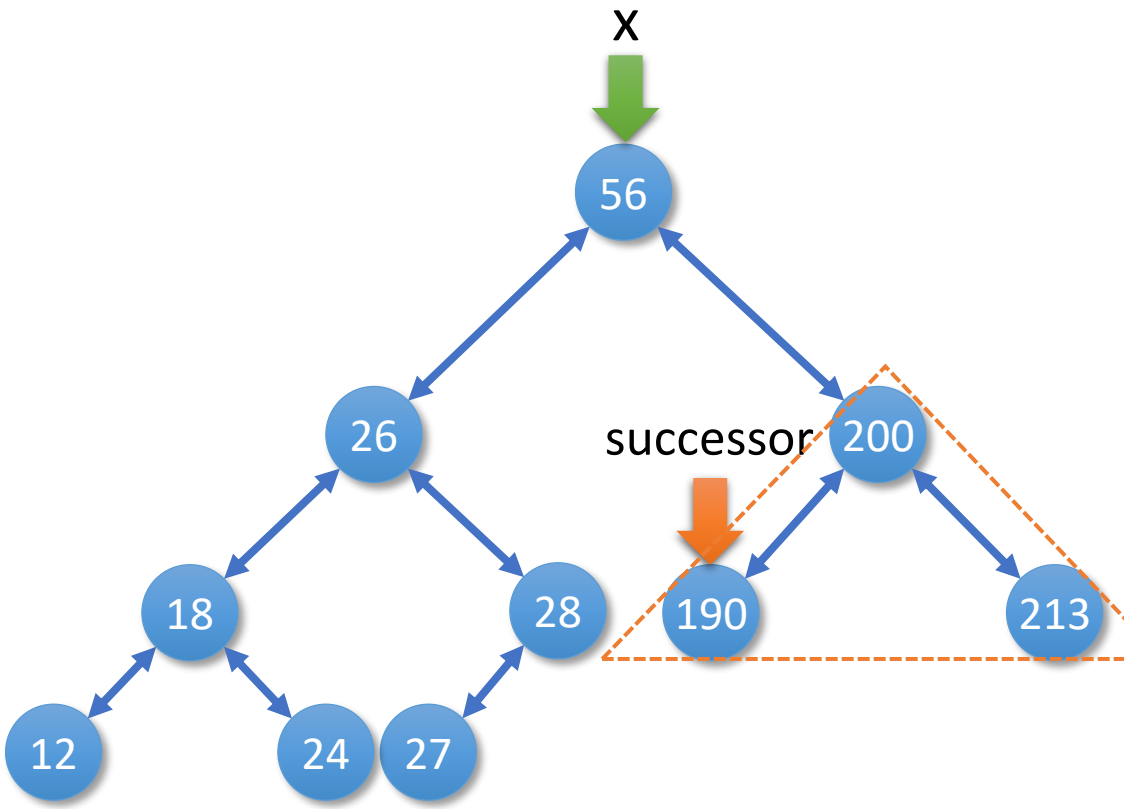
- If x.right ≠ null, the successor is the smallest in x's right subtree
- Otherwise, the successor is an ancestor



Finding Successor

```
BST-Successor[x]
If x.right ≠ null Then
   Return BST-Minimum[x.right]
y ← x.parent
While y ≠ null and x = y.right do
   x ← y
   y ← y.parent
Return y
```

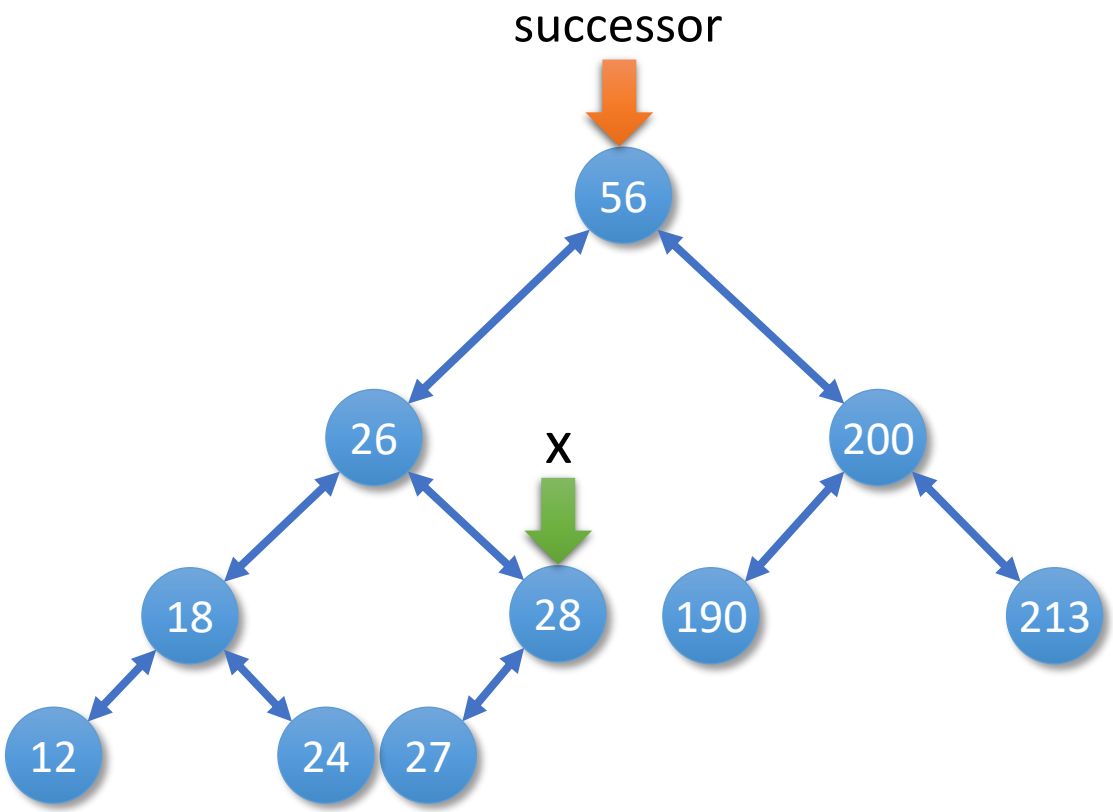
- If x.right ≠ null, the successor is the smallest in x's right subtree
- Otherwise, the successor is an ancestor



Finding Successor

```
BST-Successor[x]
If x.right ≠ null Then
    Return BST-Minimum[x.right]
y ← x.parent
While y ≠ null and x = y.right do
    x ← y
    y ← y.parent
Return y
```

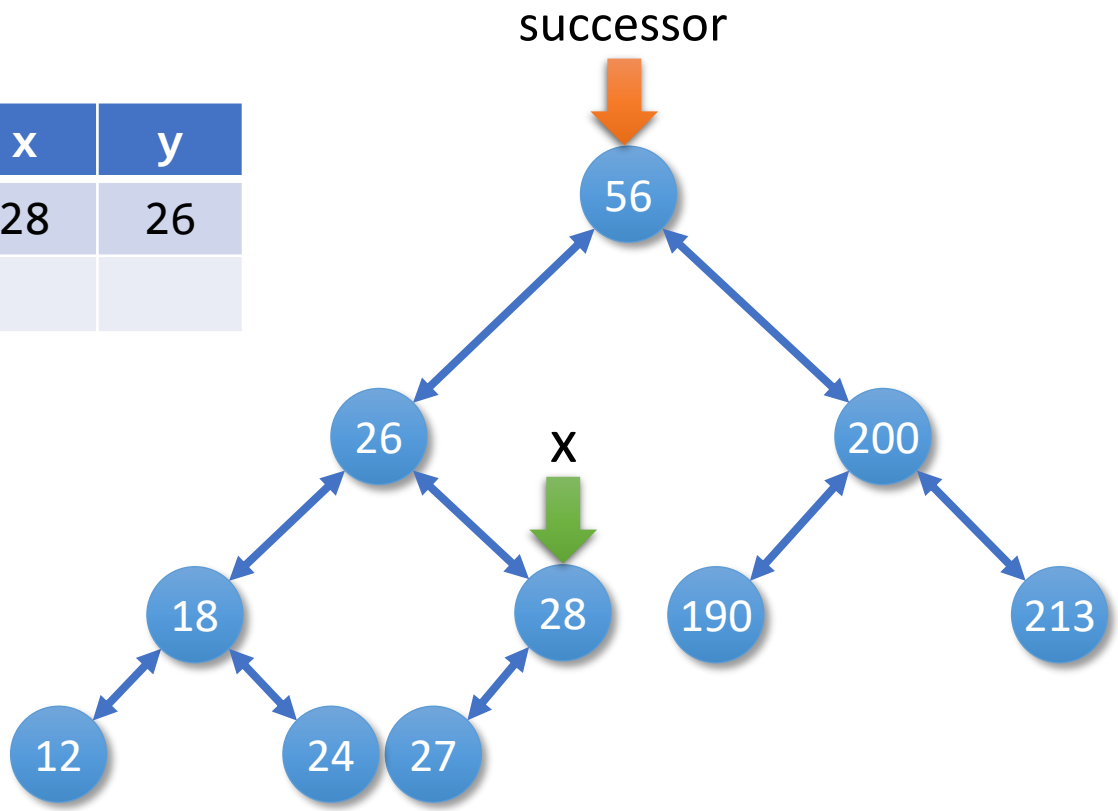
- If x.right ≠ null, the successor is the smallest in x's right subtree
- Otherwise, the successor is an ancestor



Finding Successor

```
BST-Successor[x]
If x.right ≠ null Then
    Return BST-Minimum[x.right]
y ← x.parent
While y ≠ null and x = y.right do
    x ← y
    y ← y.parent
Return y
```

x	y
28	26

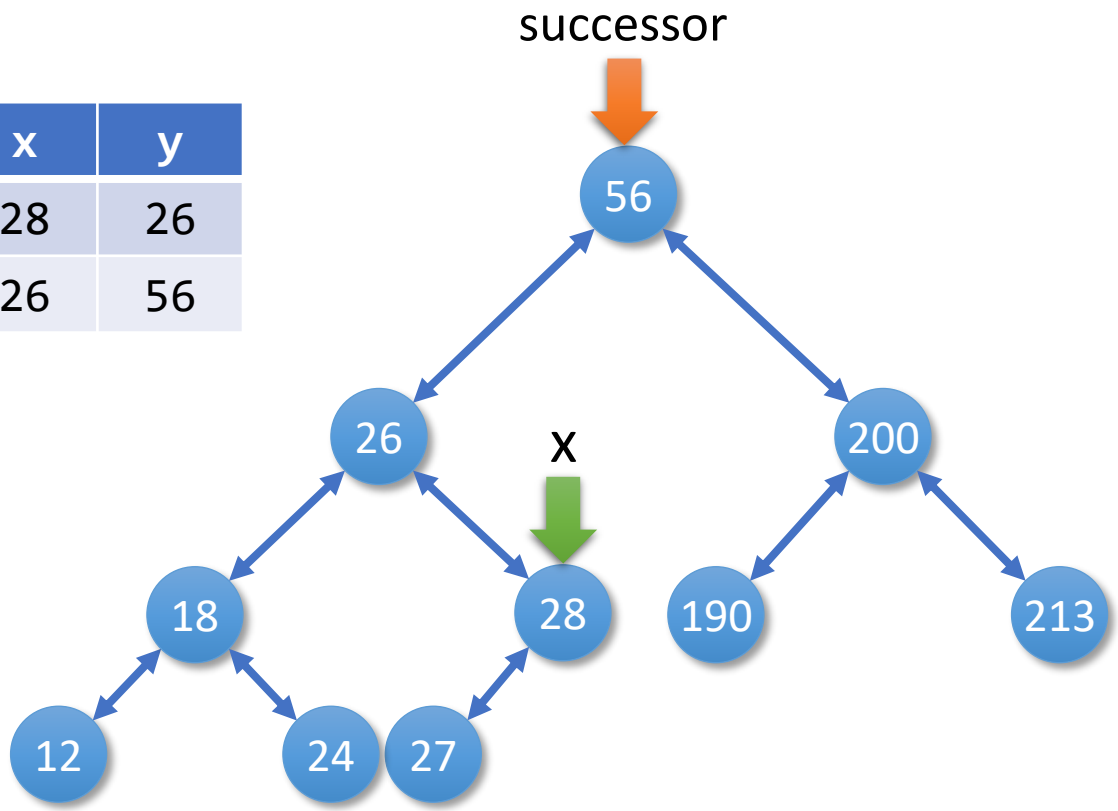


- If x.right ≠ null, the successor is the smallest in x's right subtree
- Otherwise, the successor is an ancestor

Finding Successor

```
BST-Successor[x]
If x.right ≠ null Then
    Return BST-Minimum[x.right]
y ← x.parent
While y ≠ null and x = y.right do
    x ← y
    y ← y.parent
Return y
```

x	y
28	26
26	56

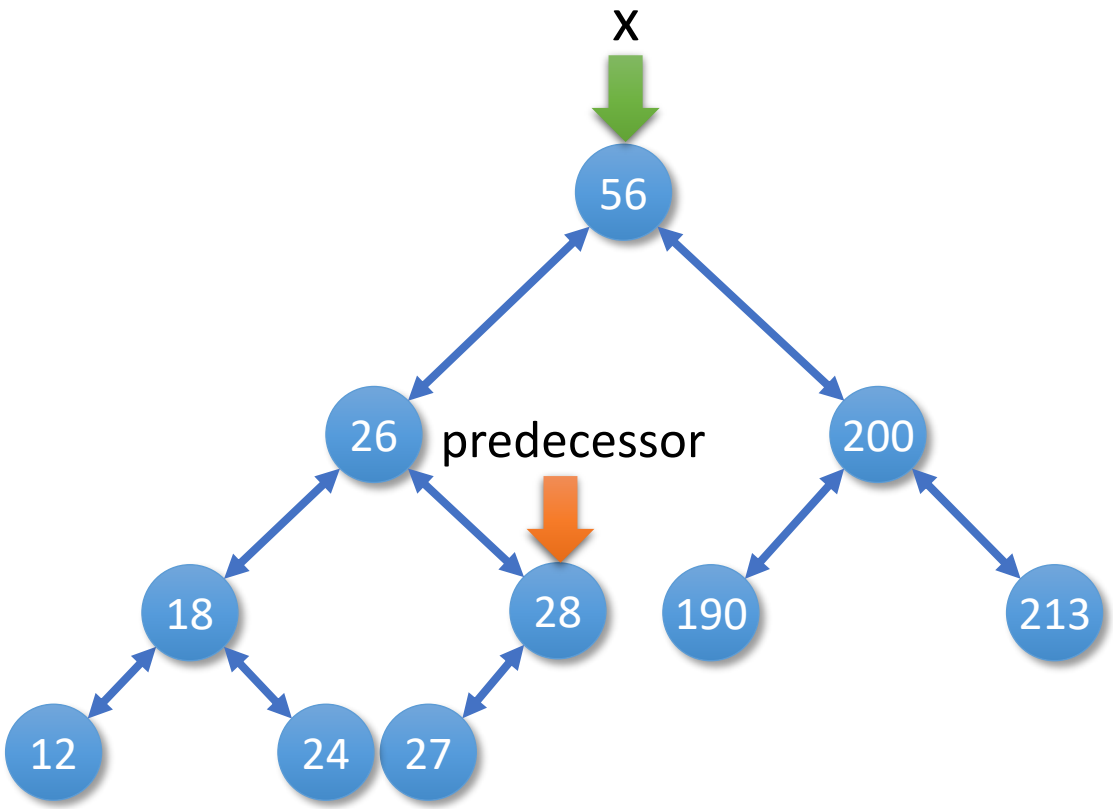


- If x.right ≠ null, the successor is the smallest in x's right subtree
- Otherwise, the successor is an ancestor

Finding Predecessor

```
BST-Predecessor[x]
If x.left ≠ null Then
   Return BST-Maximum[x.left]
y ← x.parent
While y ≠ null and x = y.left do
   x ← y
   y ← y.parent
Return y
```

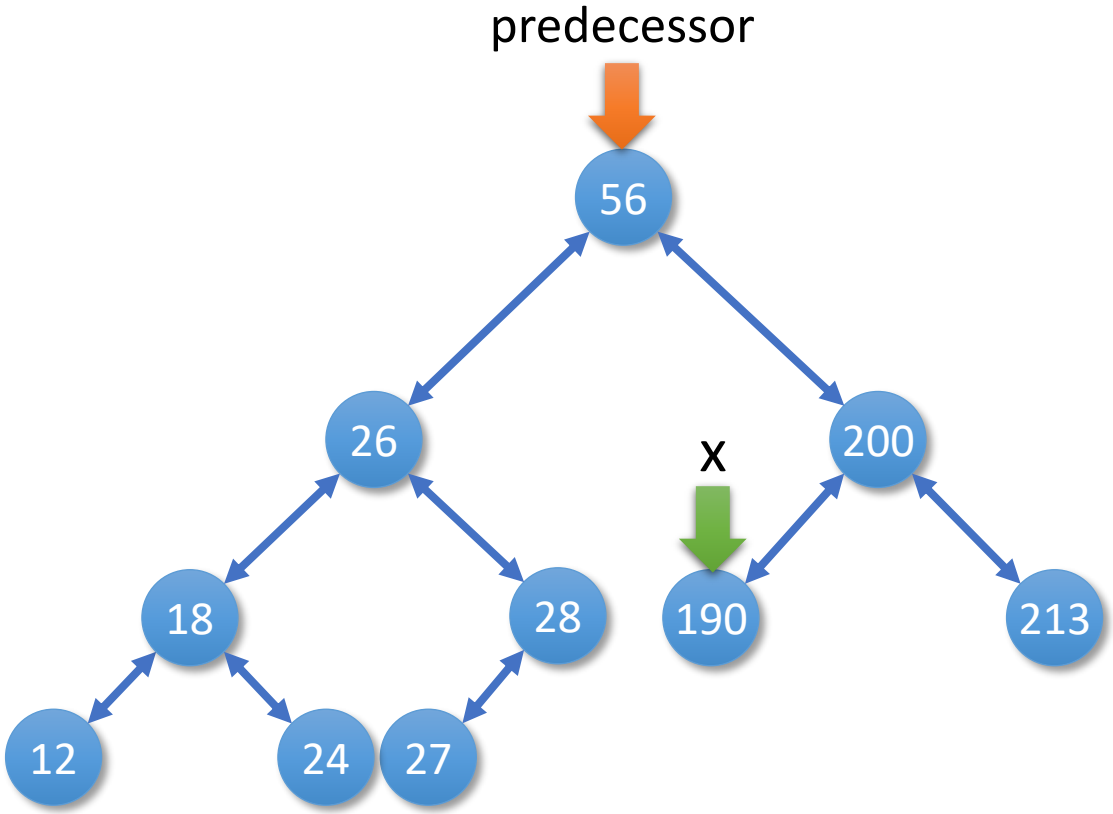
- If x.left ≠ null, the predecessor is the largest in x's left subtree
- Otherwise, the predecessor is an ancestor



Finding Predecessor

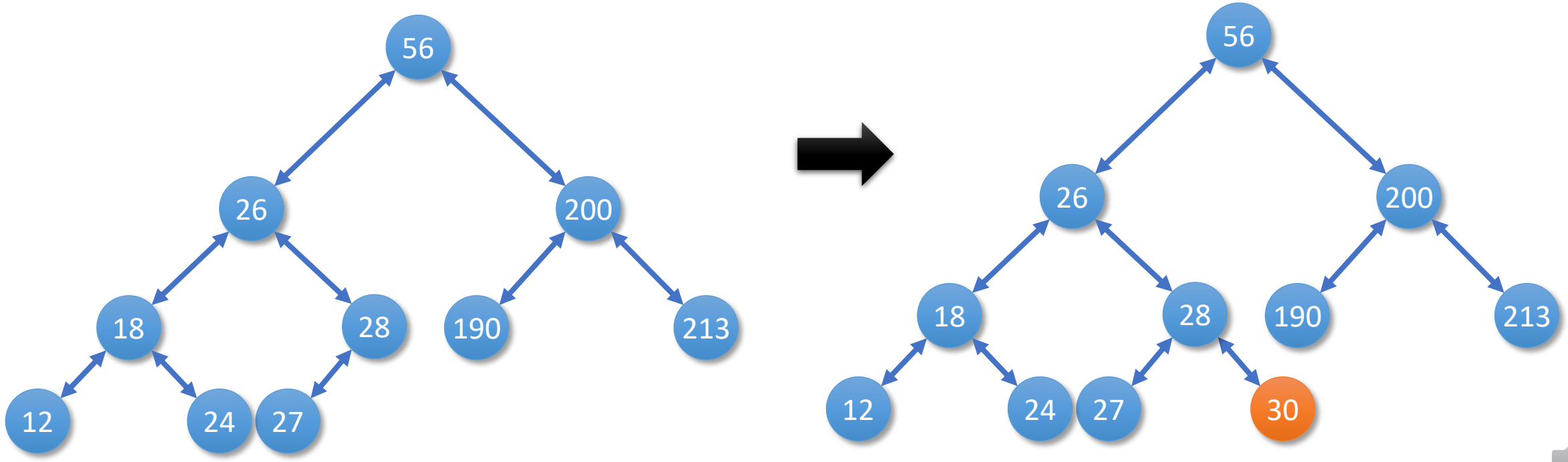
```
BST-Predecessor[x]
If x.left ≠ null Then
   Return BST-Maximum[x.left]
y ← x.parent
While y ≠ null and x = y.left do
   x ← y
   y ← y.parent
Return y
```

- If x.left ≠ null, the predecessor is the largest in x's left subtree
- Otherwise, the predecessor is an ancestor



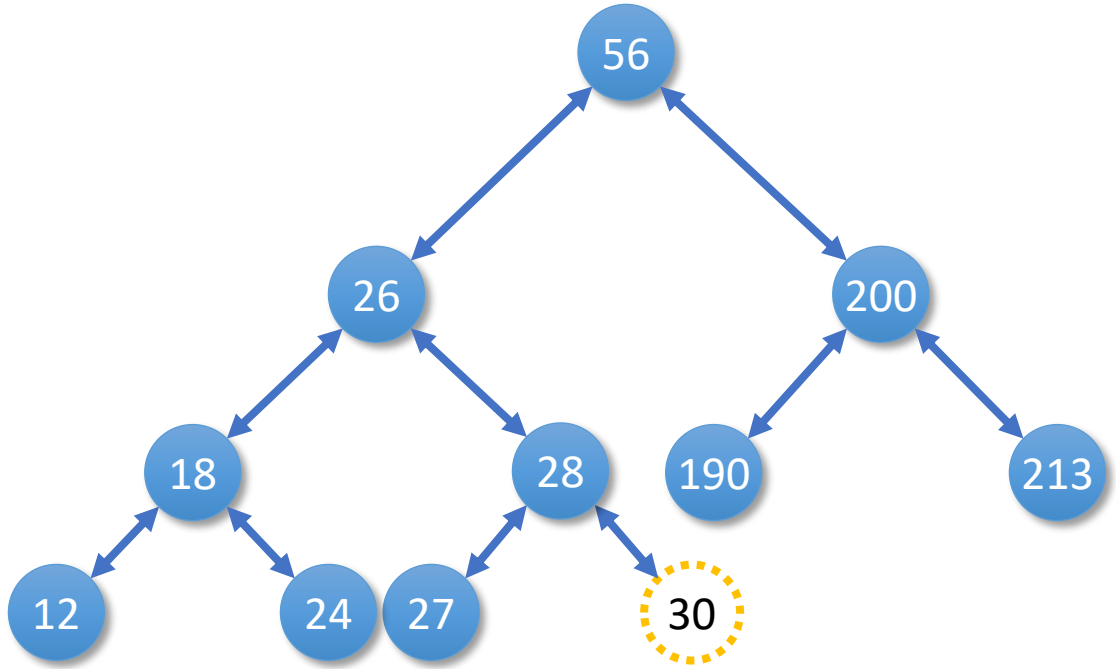
BST Insertion

- Ensure the binary-search-tree property holds after insertion
- A new key is always inserted at the leaf node



BST Insertion

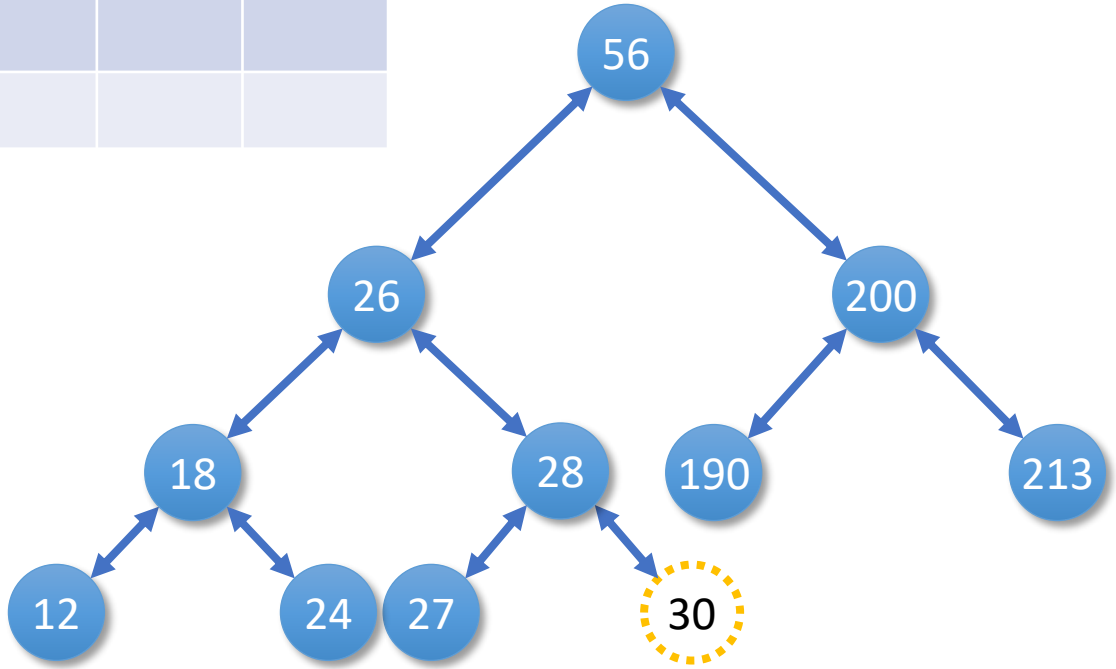
```
BST-Insert[T,z]
y ← null
x ← T.root

// Find a suitable leave
While x ≠ null Do
    y ← x
    If z.key < x.key Then
       x ← x.left
    Else x ← x.right

z.parent ← y
If y = null Then
      T.root ← z
// Update parent's child pointer
Else If z.key < y.key Then
           y.left ← z
      Else y.right ← z
```



BST Insertion

```
BST-Insert[T,z]
y ← null
x ← T.root

// Find a suitable leave
While x ≠ null Do
    y ← x
    If z.key < x.key Then
      x ← x.left
    Else x ← x.right

z.parent ← y
If y = null Then
      T.root ← z
// Update parent's child pointer
Else If z.key < y.key Then
            y.left ← z
      Else y.right ← z
```

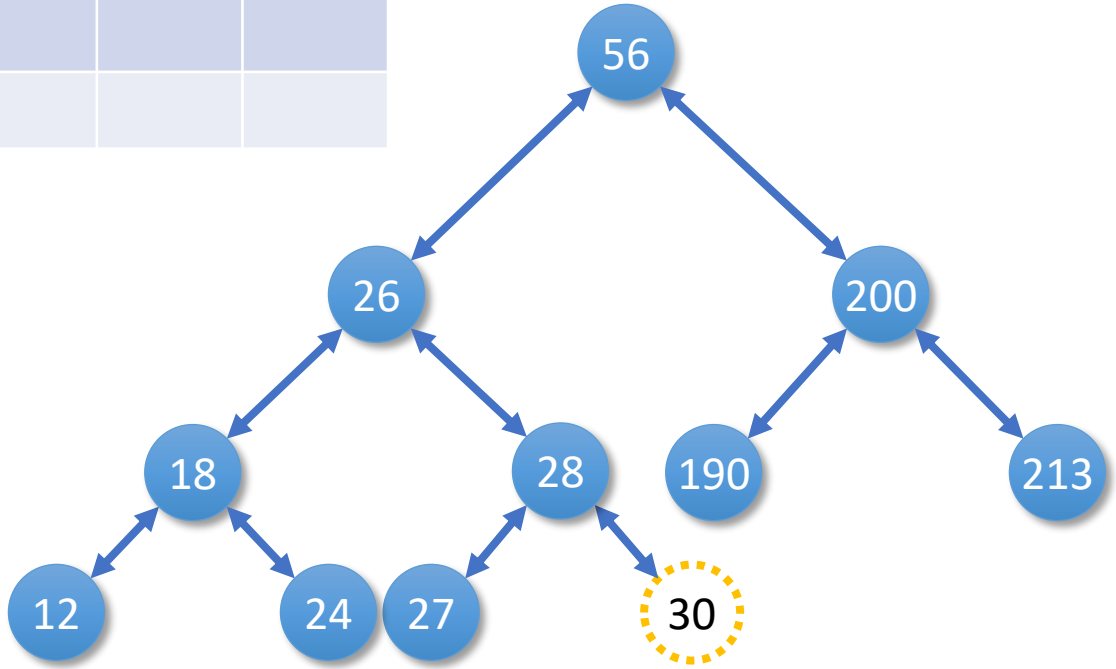
z	x	y
30	56	null



BST Insertion

```
BST-Insert[T,z]
y ← null
x ← T.root

// Find a suitable leave
While x ≠ null Do
    y ← x
    If z.key < x.key Then
       x ← x.left
    Else x ← x.right

z.parent ← y
If y = null Then
       T.root ← z
// Update parent's child pointer
Else If z.key < y.key Then
            y.left ← z
       Else y.right ← z
```

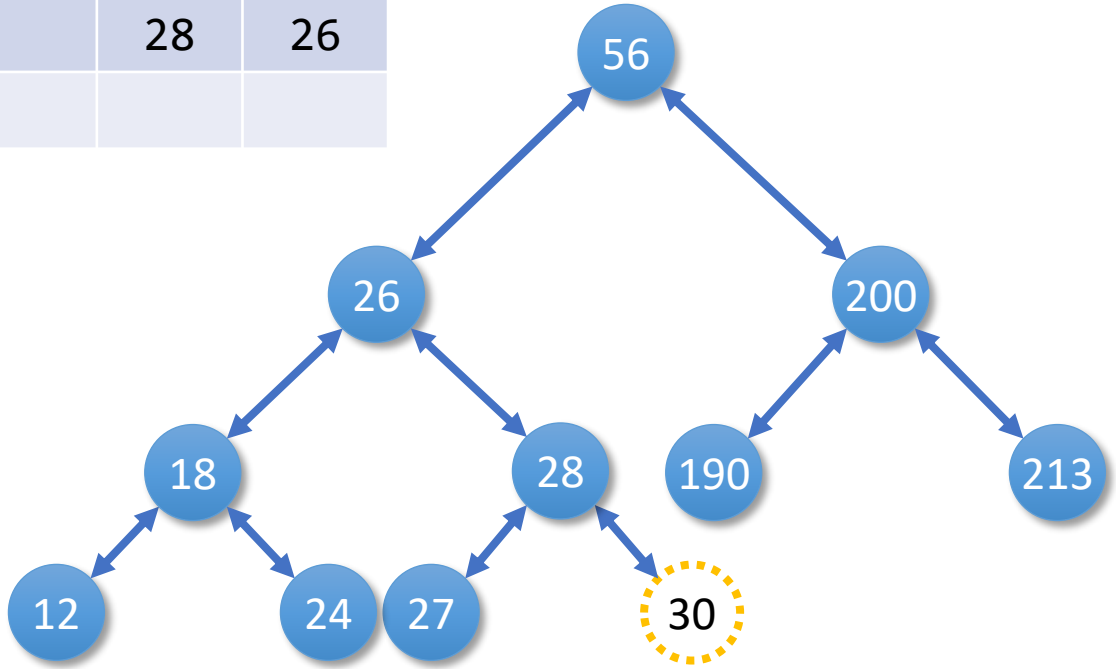
z	x	y
30	56	null
	26	56



BST Insertion

```
BST-Insert[T,z]
y ← null
x ← T.root

// Find a suitable leave
While x ≠ null Do
    y ← x
    If z.key < x.key Then
       x ← x.left
    Else x ← x.right

z.parent ← y
If y = null Then
       T.root ← z
// Update parent's child pointer
Else If z.key < y.key Then
            y.left ← z
       Else y.right ← z
```

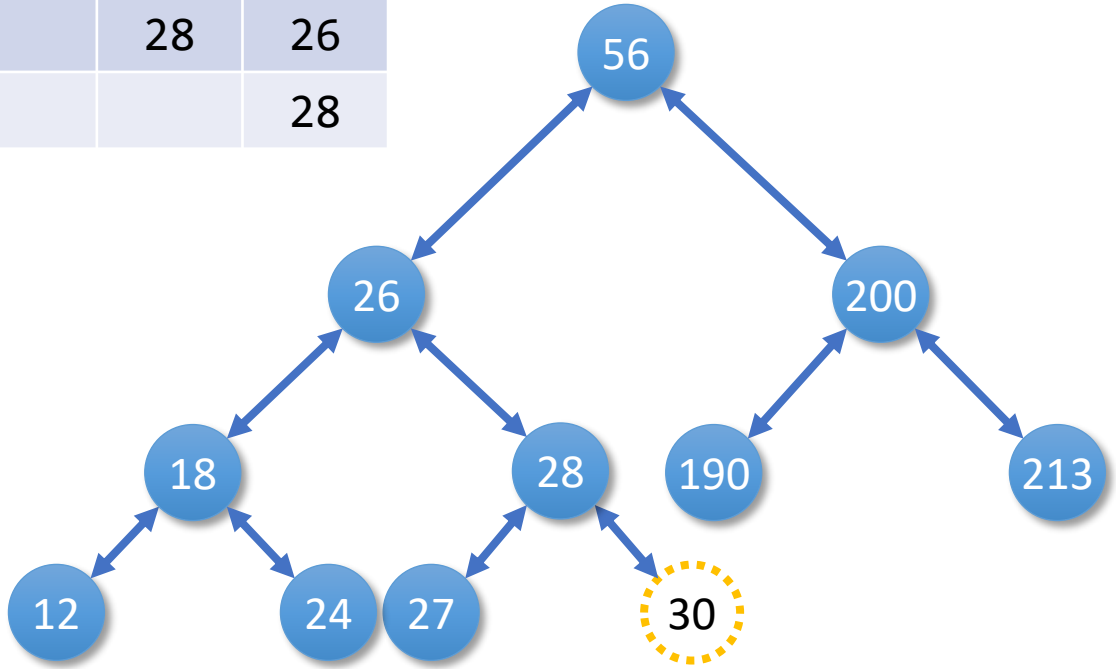
z	x	y
30	56	null
	26	56
	28	26



BST Insertion

```
BST-Insert[T,z]
y ← null
x ← T.root

// Find a suitable leave
While x ≠ null Do
    y ← x
    If z.key < x.key Then
       x ← x.left
    Else x ← x.right

z.parent ← y
If y = null Then
       T.root ← z
// Update parent's child pointer
Else If z.key < y.key Then
            y.left ← z
       Else y.right ← z
```

z	x	y
30	56	null
	26	56
	28	26
		28

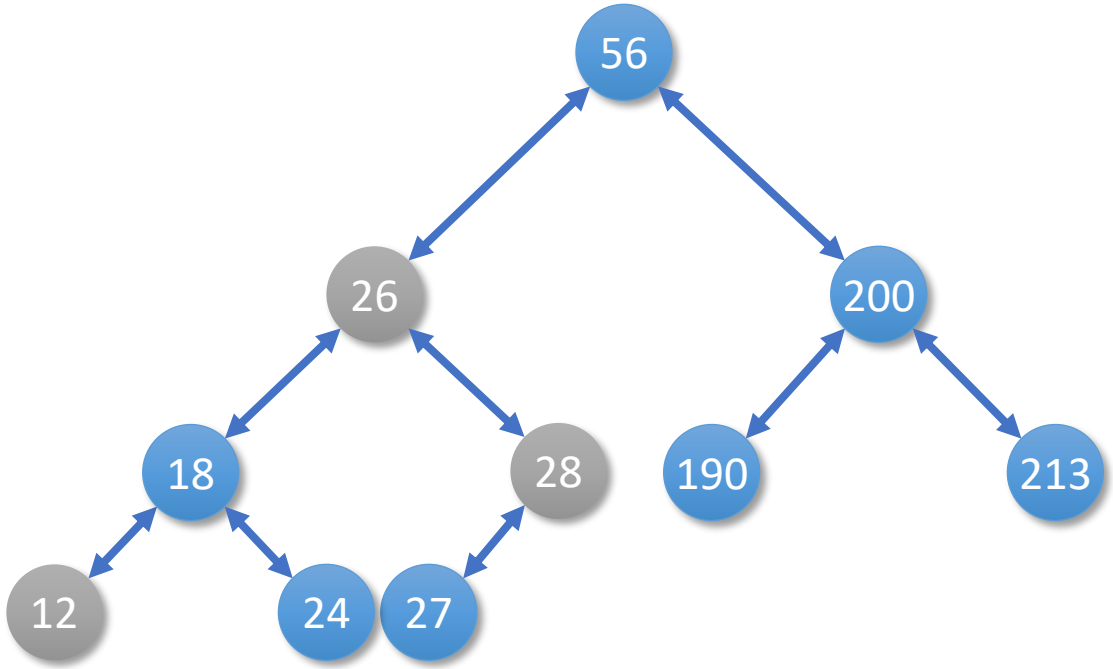


BST Deletion

- `BST-Delete[T,x]`
  - Case 0:
    - If x has no child
  - Case 1:
    - If x has one child
  - Case 2:
    - If x has two children (i.e., substree)

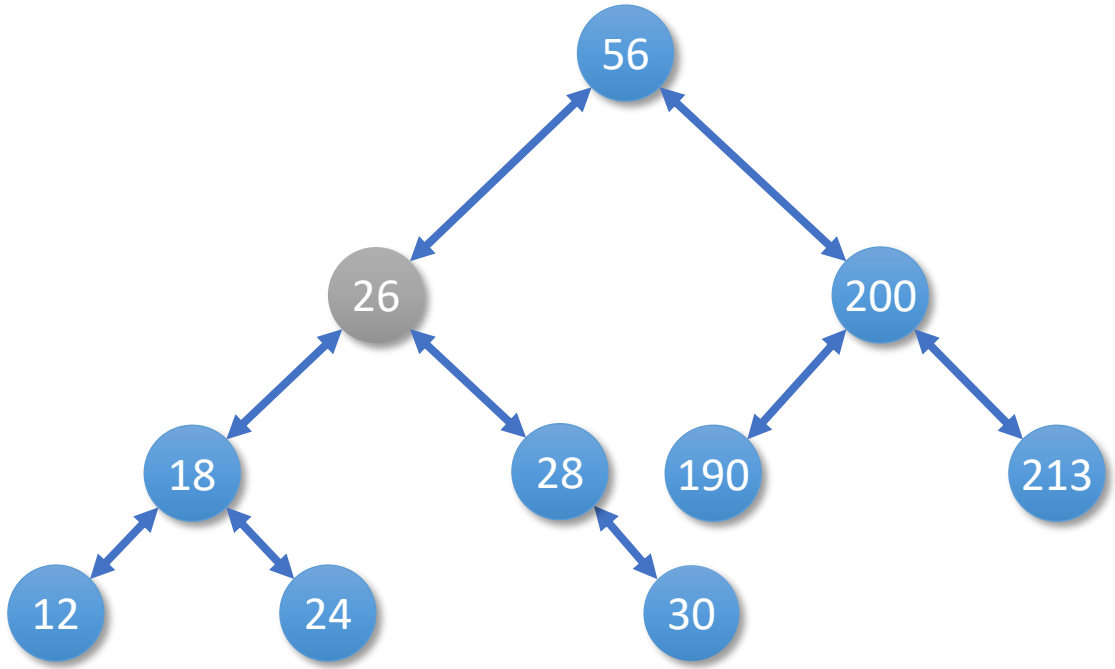
BST Deletion

- Example
  - Case 0
    - Remove 12
  - Case 1
    - Remove 28
      - Set [27].parent ← 26
      - Set [26].right ← 27
  - Case 2
    - Remove 26
      - Replace 26 by 27
      - Remove 27 from subtree



BST Deletion

- Example
  - Case 2:
    - Remove 26
      - Replace 26 with 28
      - Remove 28 from subtree (Case 1)



BST Deletion

- Basic Idea:
  - Case 0:
    - If x has no children
    - Then remove x
  - Case 1:
    - If x has one child
    - Then make x.parent point to the child
  - Case 2:
    - If x has two children (subtrees)
    - Then replace x with its successor
    - Delete the successor in subtree, which gives Case 0 or 1

Exercise

- Consider the following operations to binary search tree
  - Insert 12, 43, 34, 11, 44, 1, 4
  - Delete 44, 1
  - Insert 55, 23, 11, 13
  - Delete 12, 43
- What is the tree height of the final tree?
- What is the key of the root of the final tree?

Implement a BST

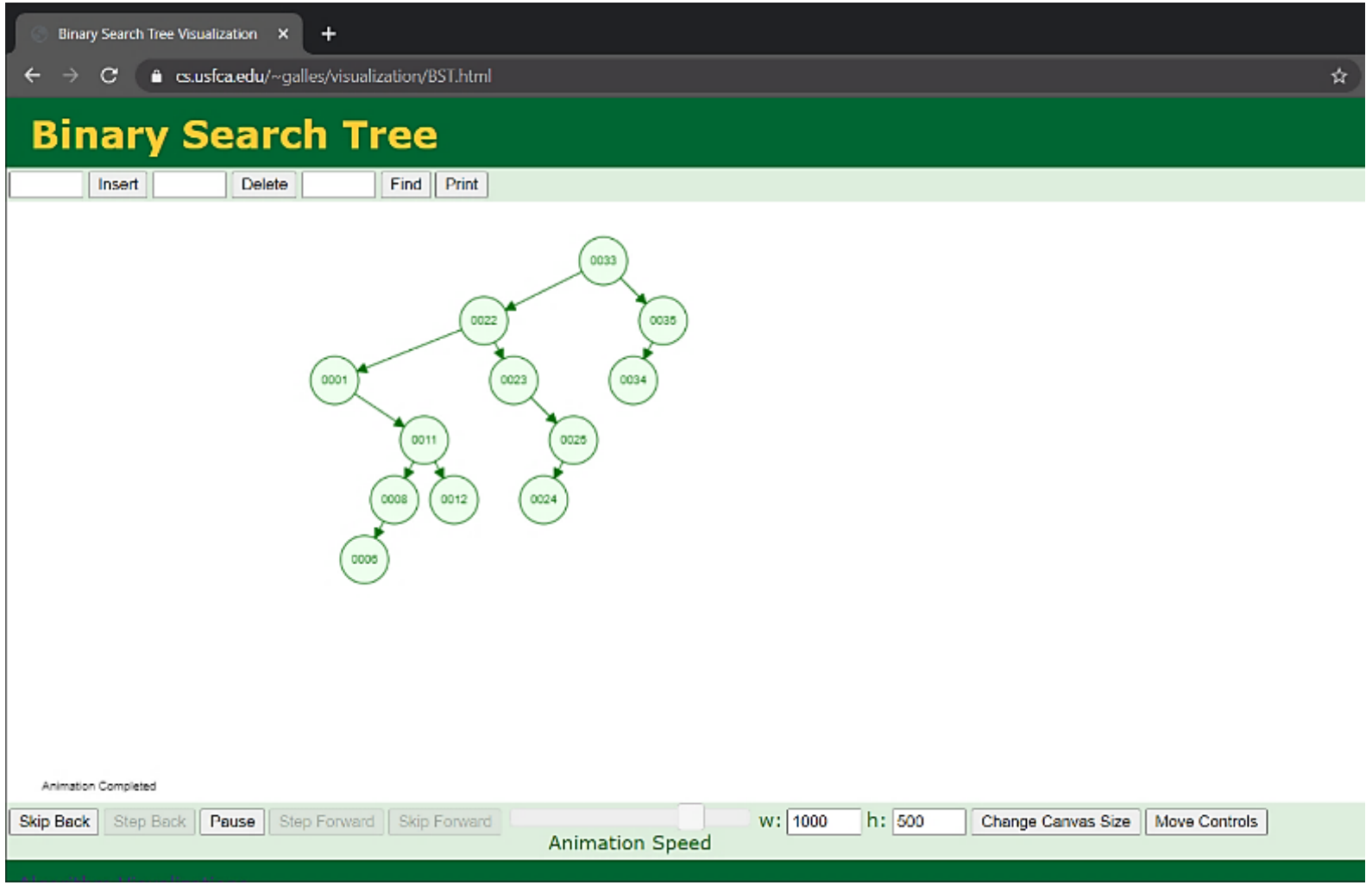
Inner Class Node

```
public class BST {

  public class Node{
        Integer key;     // Node key
        Node left;       // Left child
        Node right;      // Right child
        Node parent;     // Parent node
        Node(Integer key){ . . . }
}

  Node root;

  public Node find(Integer key) { . . . }
  public Node insert(Integer key) { . . . }
  public void delete(Integer key) { . . . }
}
```


Implement a BST

```
public abstract class BinaryTree <T extends Comparable<T>>
{

    public abstract boolean isEmpty(); // check if the tree is empty
    public abstract T biggest(); // find the biggest element in the tree
    public abstract T smallest(); // find the smallest element in the

    public abstract boolean find(T d); // check if the element is in the tree

    public abstract BinaryTree<T> insert(T d);
    // add an element to the tree, this returns the new/modified tree

    public abstract BinaryTree<T> delete(T d);
    // remove an element from the tree, this return the new/modified tree
}
```

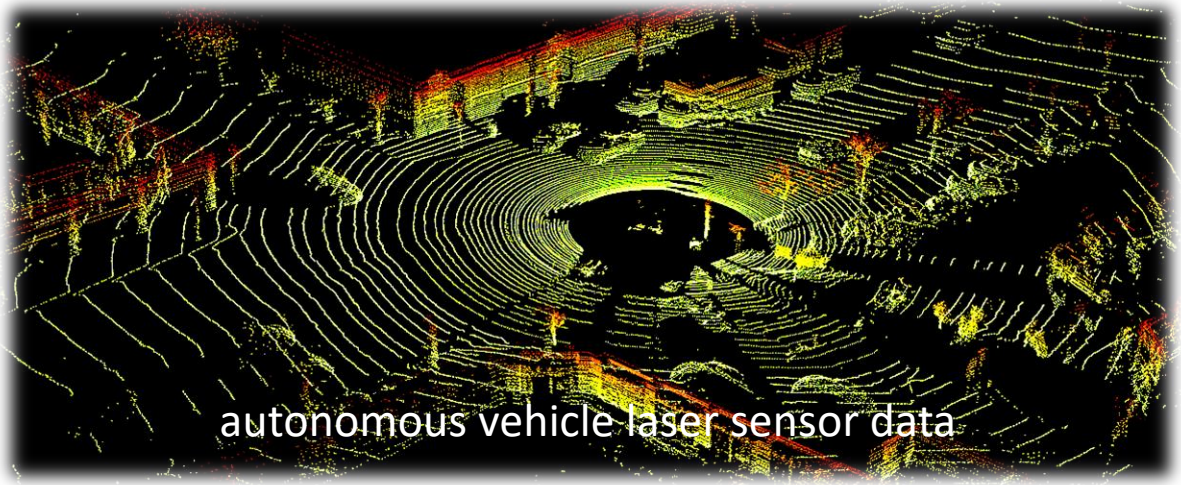

Demo

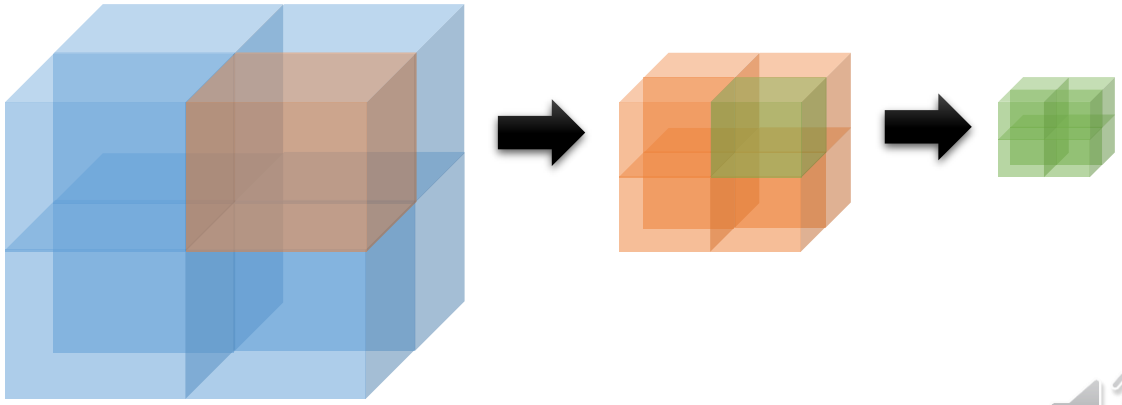


https://www.cs.usfca.edu/~galles/visualization/BST.html

Octree for 3D Space Data

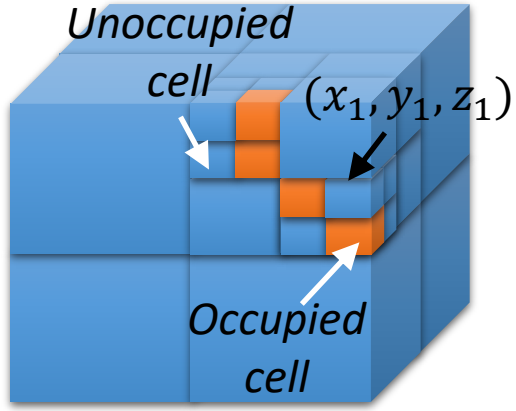
- How to record 3D spatial data?
  - e.g., from autonomous vehicle laser sensors
- Octree is tree-based data structure for 3D point data
- 3D space is sub-dividing into 8 spatial cells recursively
  - Each cell can be represented by a node in a tree
  - There are 8 child nodes of each node, which are the sub-divided cells





Octree for 3D Space Data

- Consider a set of 3D points
  - Each point occupies in the 3D space
- Octree is a tree-based data structure for 3D points
  - A node in Octree has a binary value {0, 1}
  - An occupied cell contains a point or a set of points, and is labeled by ‘1’, otherwise is labeled by ‘0’



Geometric representation

Octree representation

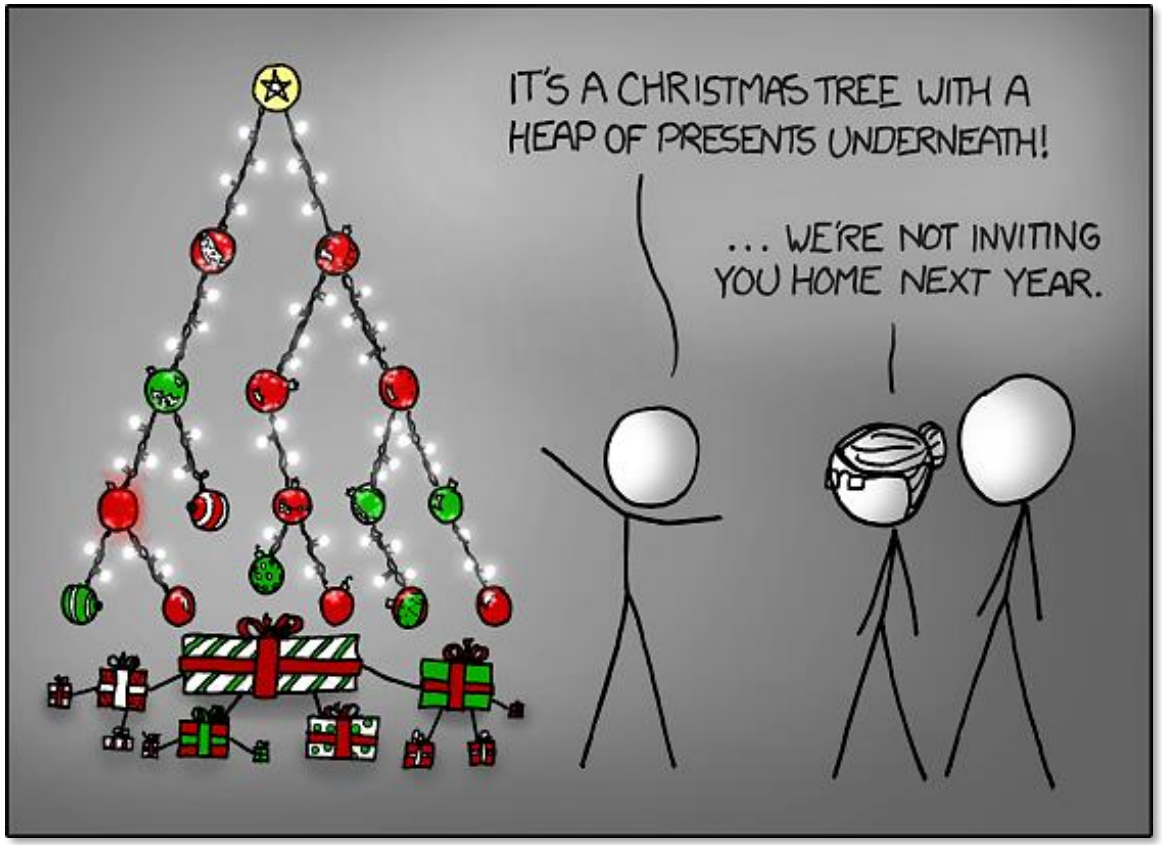
Summary

- Linear data structures
  - Linked list, stack, hashtable
- Approximate data structures
  - Bloom filter, count-min sketch
- Non-linear data structures
  - Binary search tree
    - Search and traversing
    - Successor and predecessor
    - Insertion
    - Deletion
  - Heap, octree

Reference

- Visualizations
  - https://www.cs.usfca.edu/~galles/visualization/StackLL.html
  - https://www.cs.usfca.edu/~galles/visualization/HashTable.html
  - https://www.cs.usfca.edu/~galles/visualization/BST.html

- Reference:
  - Chapters 12-13 in Introduction to Algorithms (by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest)


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.