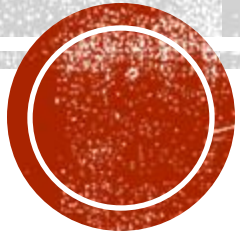
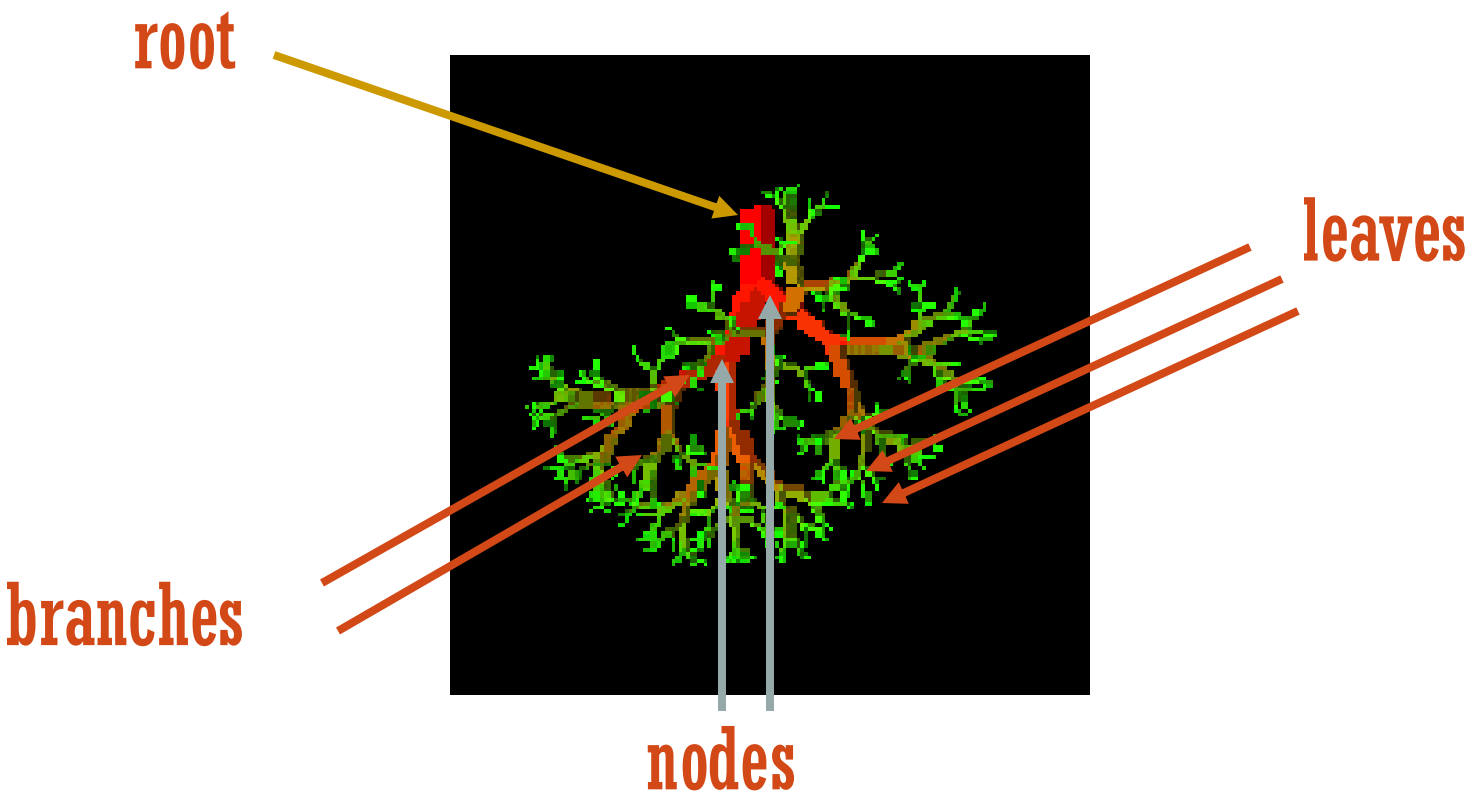
TREES

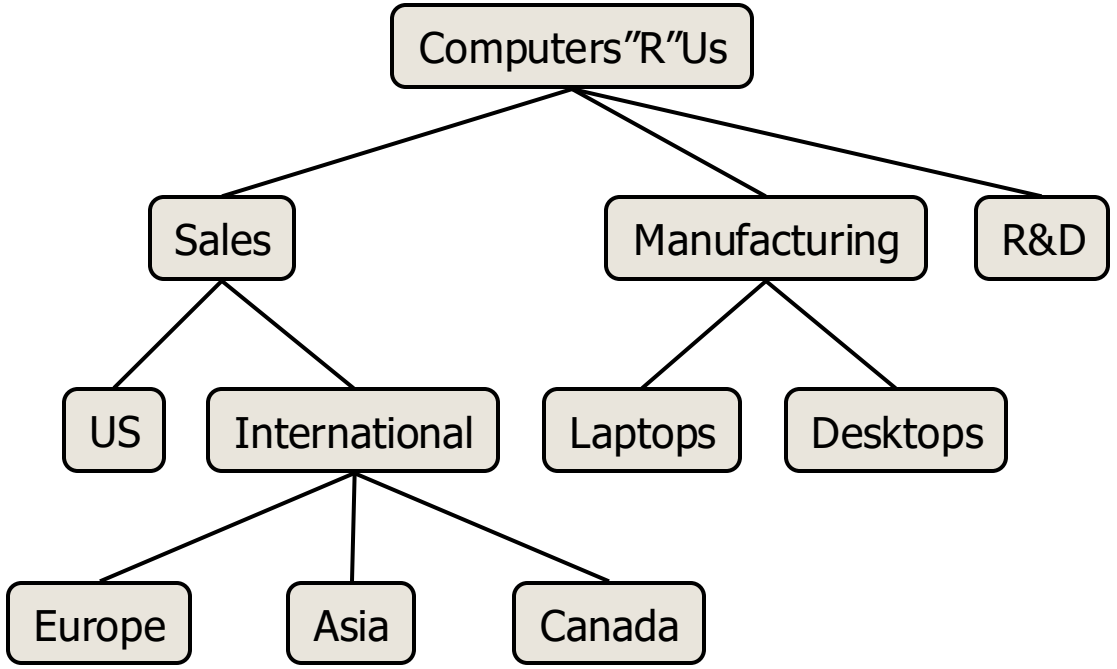


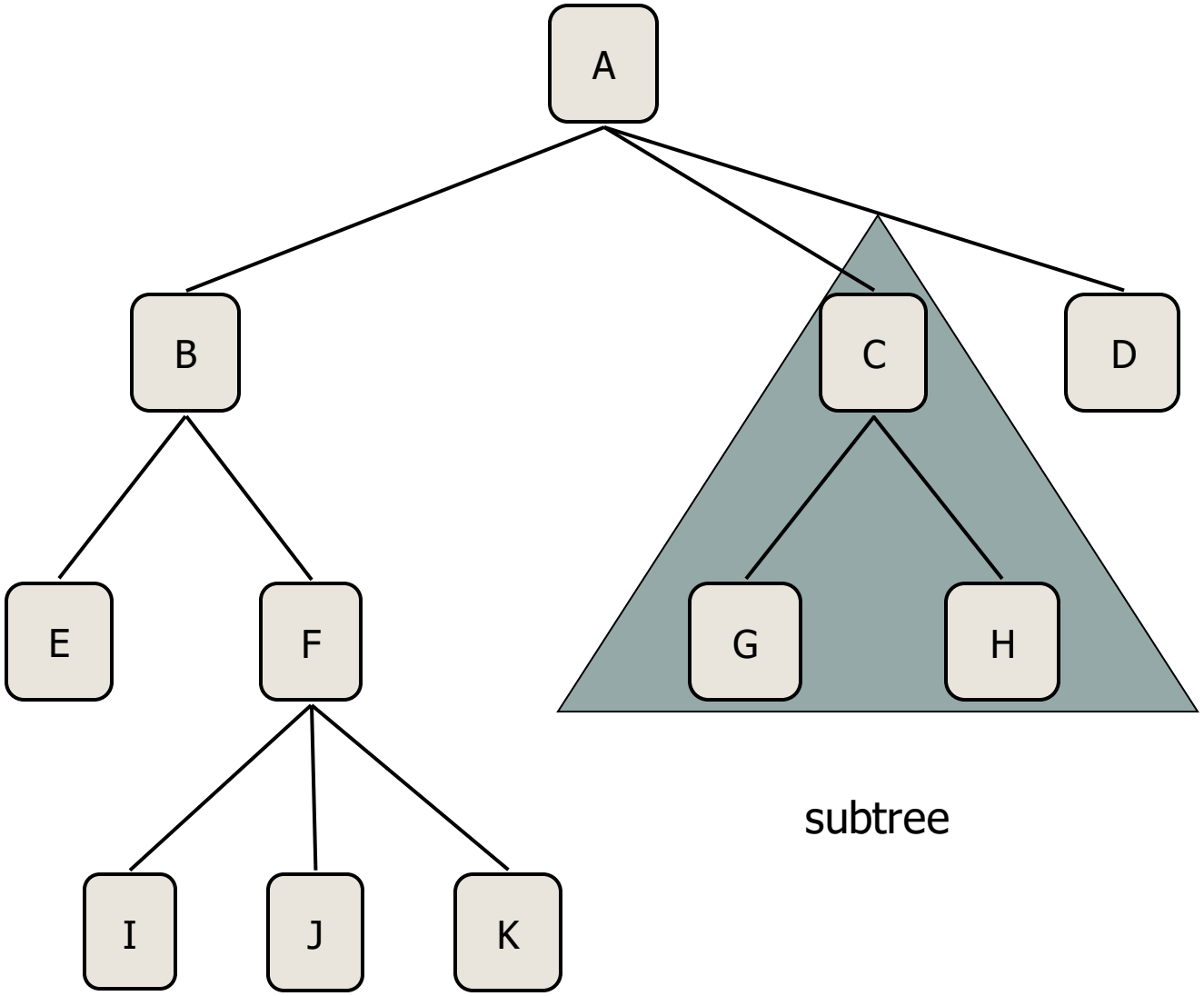
TREE VIEW



WHAT IS A TREE?

- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- Consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems
  - Programming environments

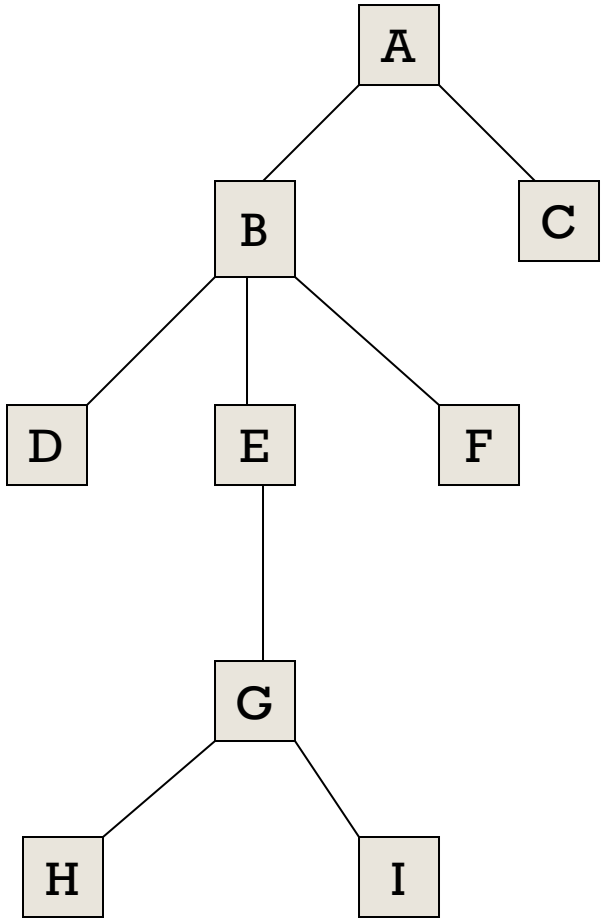



A
B
C
D
E
F
G
H
I
J
K
subtree

TERMINOLOGY

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, great-grandparent, etc.
- **Descendant** of a node: child, grandchild, great-grandchild, etc.
- **Height** of a tree: maximum depth of a node (4)
- **Degree** of a node: the number of its children
- **Degree** of a tree: the maximum number of its node.
- **Subtree**: tree consisting of a node and its descendants

EXAMPLE



Property	Value
Number of nodes	9
Height	5
Root Node	A
Leaves	D,H,I,F,C
Interior nodes	A,B,E,G
Ancestors of H	G,E,B,A
Descendants of B	D,E,G,H,I,F
Siblings of E	D,F
Right subtree of A	C
Degree of this tree	3

TREE ADT

- We use positions to abstract nodes
- Generic methods:
  - integer **size**()
  - boolean **isEmpty**()
  - objectIterator **elements**()
  - positionIterator **positions**()
- Accessor methods:
  - position **root**()
  - position **parent**(p)
  - positionIterator **children**(p)
- Query methods:
  - boolean **isInternal**(p)
  - boolean **isExternal**(p)
  - boolean **isRoot**(p)
- Update methods:
  - **swapElements**(p, q)
  - object **replaceElement**(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

BINARY TREES

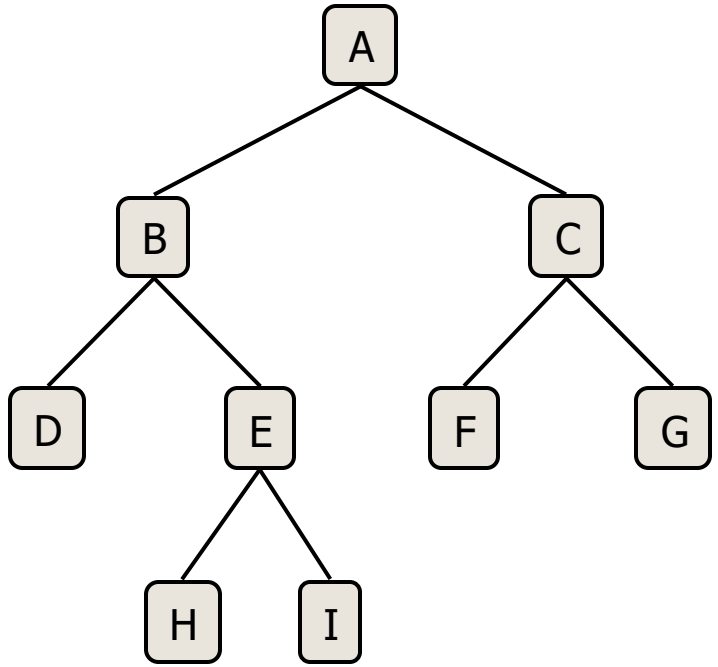
BINARY TREES

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (degree of two)
  - The children of a node are an ordered pair
    (Position of left and right child matter)

- We call the children of an internal node left child and right child

- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, OR
  - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
  - Arithmetic expressions
  - Decision processes
  - Searching

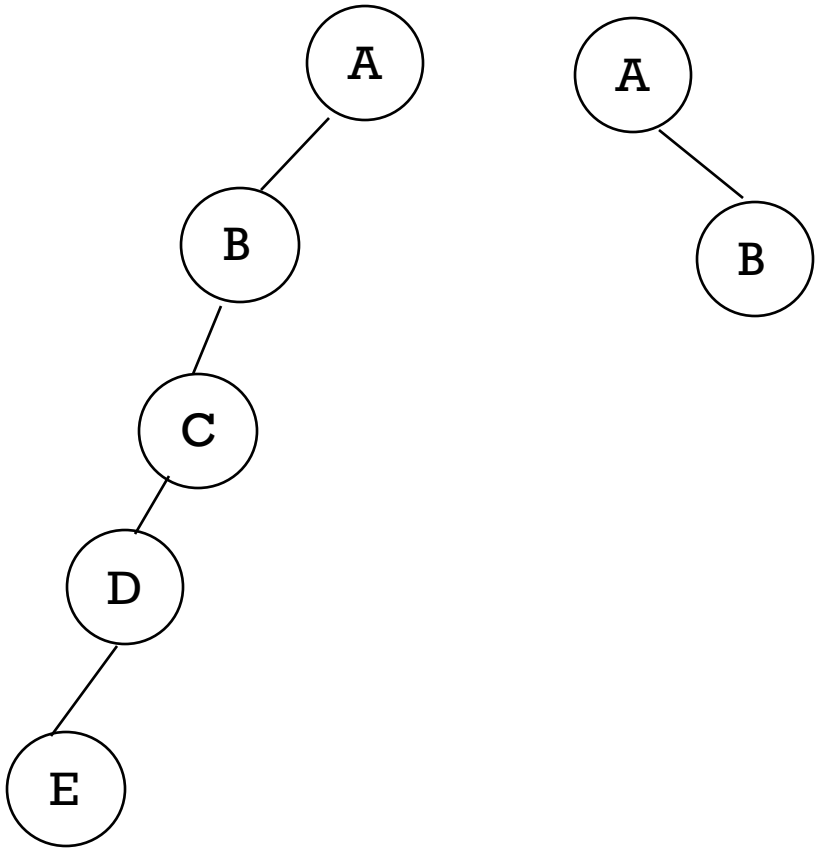


BINARYTREE ADT

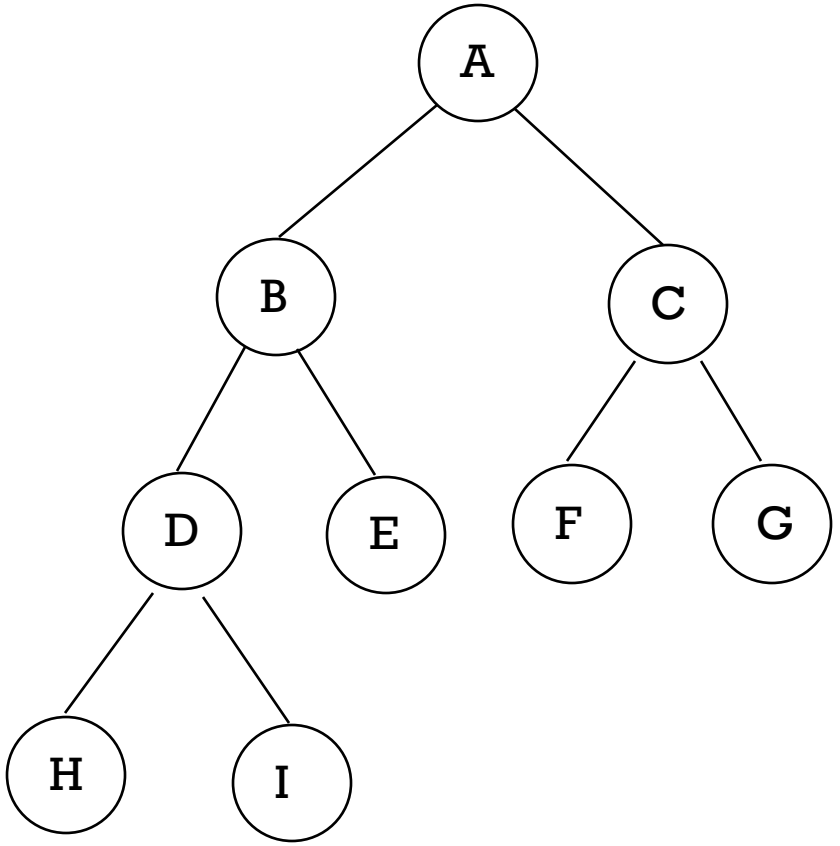
- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position leftChild(p)
  - position rightChild(p)
  - position sibling(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

EXAMPLES OF BINARY TREE

Skewed Binary Tree



Complete Binary Tree



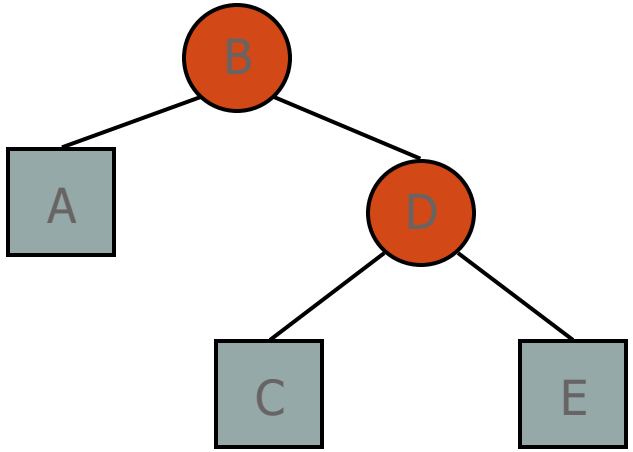
TREE VS BINARY TREE

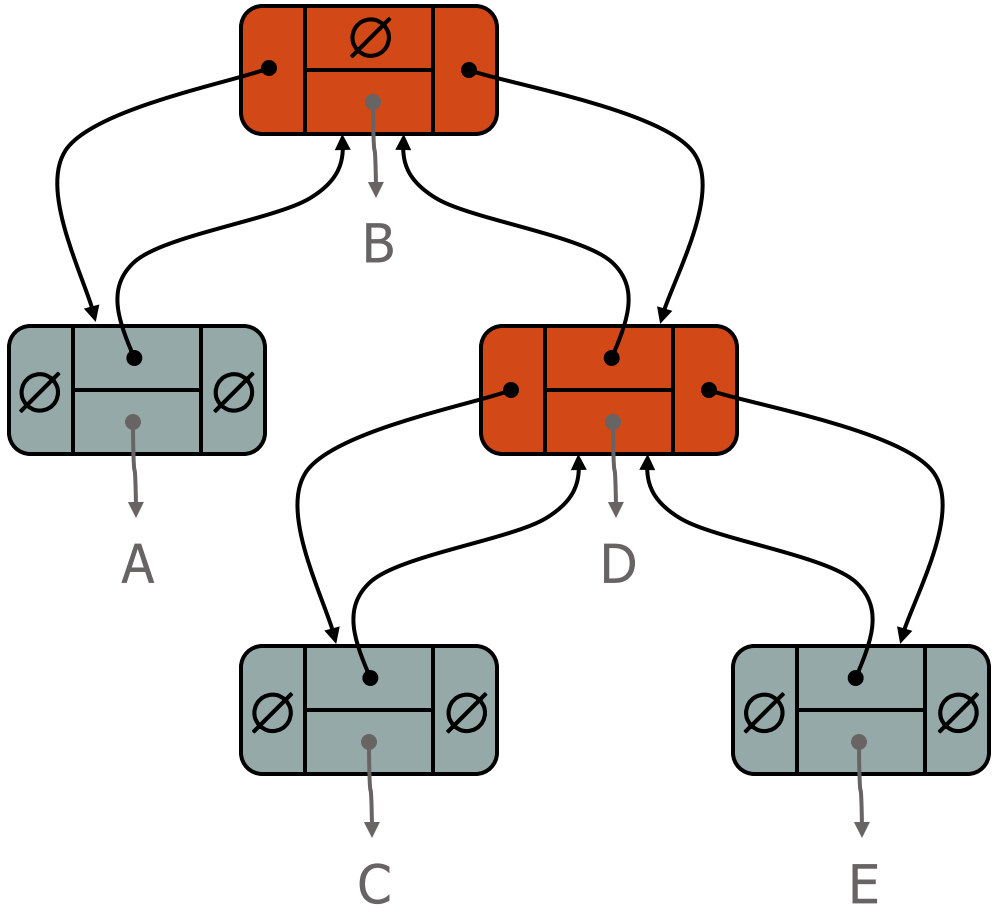
- The subtrees of a binary tree are ordered; those of a tree are not ordered.

- Are different when viewed as binary trees.
- Are the same when viewed as trees.

REPRESENTATION OF A BINARY TREE

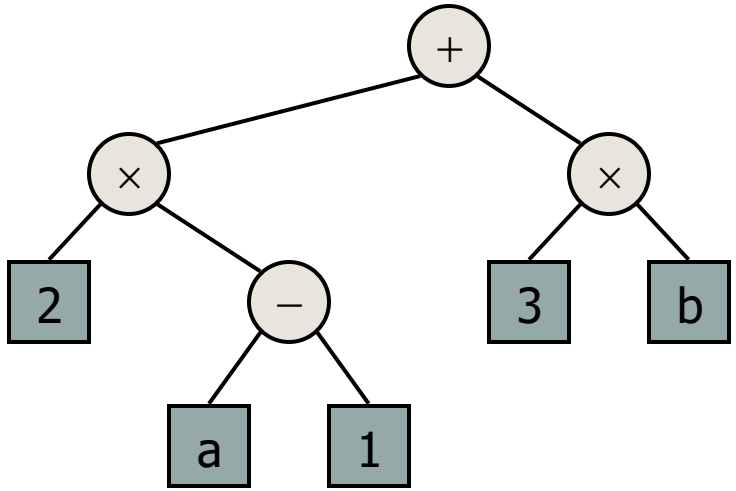
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node





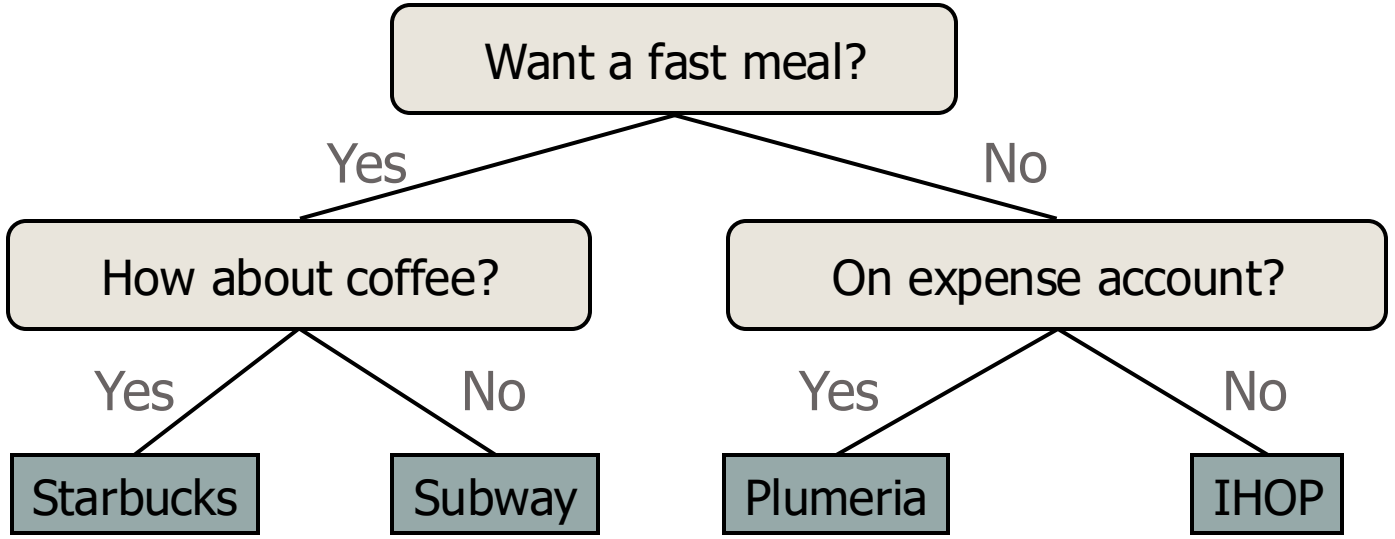
APPLICATIONS: ARITHMETIC EXPRESSION

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression (2 x (a - 1) + (3 x b))



APPLICATIONS: DECISION TREE

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



NUMBER OF NODES IN A BINARY TREE

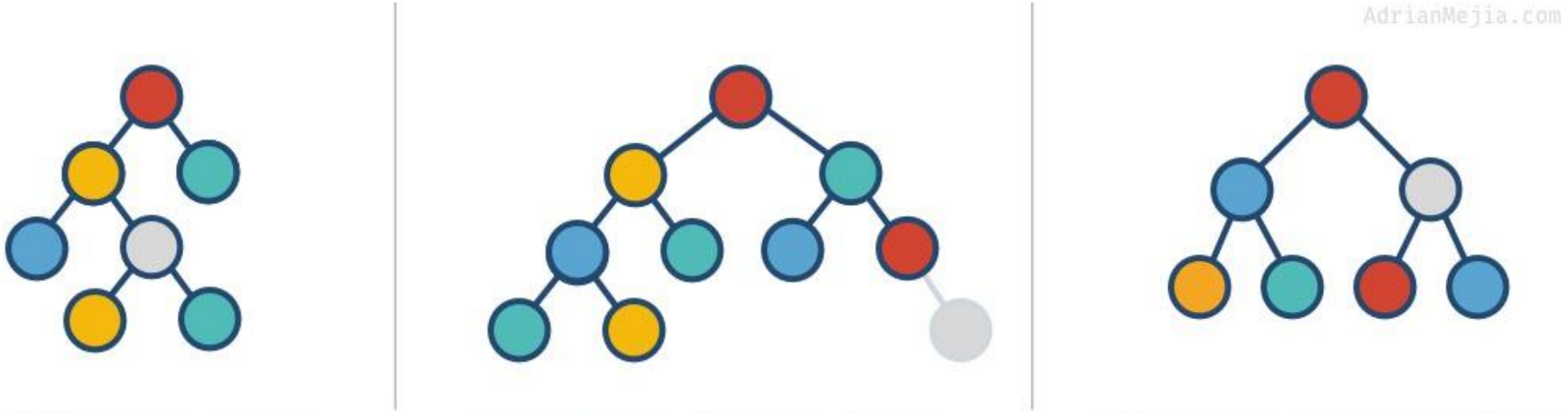
Height of a tree with single node is considered as 1.
A tree has maximum nodes if all levels have maximum nodes.
Maximum number of nodes in a binary tree of height h is:

$1 + 2 + 4 + .. + 2^{h-1}$.

This is a GP series with h terms
Number of nodes in a binary tree of height h = $2^h - 1$.

COMPLETE VS FULL BINARY TREES

- **Full binary tree :** Every node has either zero or two children. The nodes can be organized in any way.
- **Complete binary tree**: All levels are filled except possibly the last, and all child nodes are to the left.
- **Perfect binary tree**: All internal nodes have two children, and all leaf nodes are at the same level.



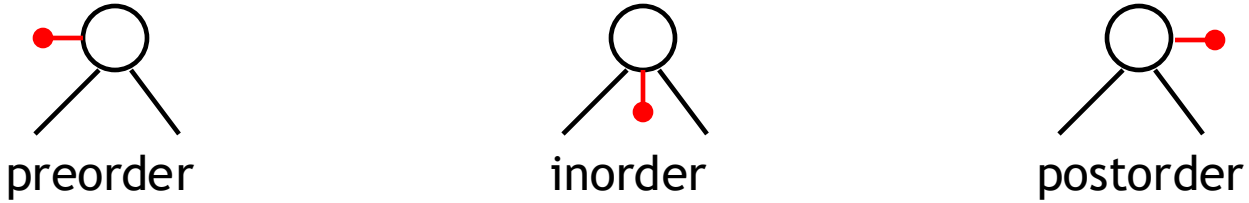
BINARY TREE TRAVERSAL

- Let l, R, and r stand for moving left, visiting the node and moving right.

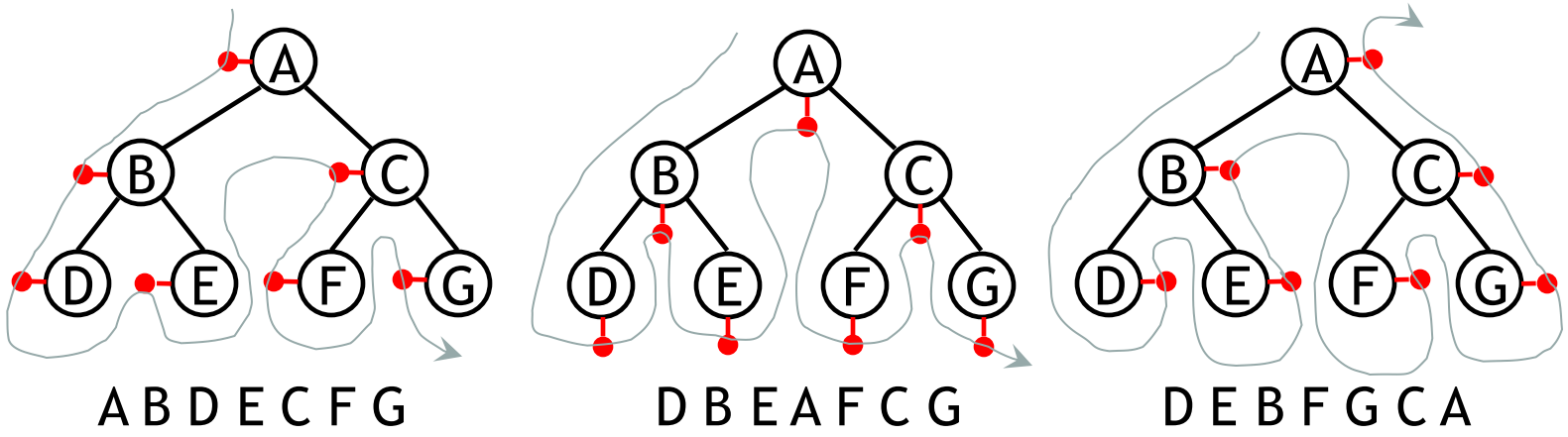
- There are six possible combinations of traversal
  - lRr, lrR, Rlr, Rrl, rRl, rlR

- Adopt convention that we traverse left before right, only 3 traversals remain
  - lRr, lrR, Rlr
  - Inorder, Postorder, Preorder

The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:



To traverse the tree, collect the flags:



19

BINARY SEARCH TREES

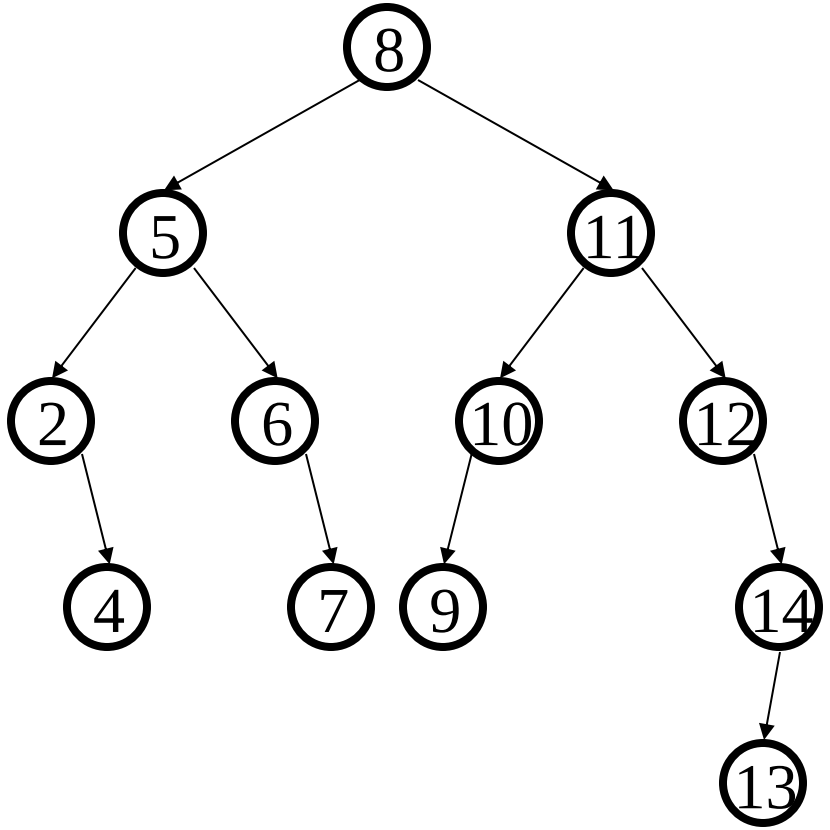
PROPERTIES OF A BST

Binary tree property

- Each node has ≤ 2 children
- Result:
  - storage is small
  - operations are simple
  - average depth is small

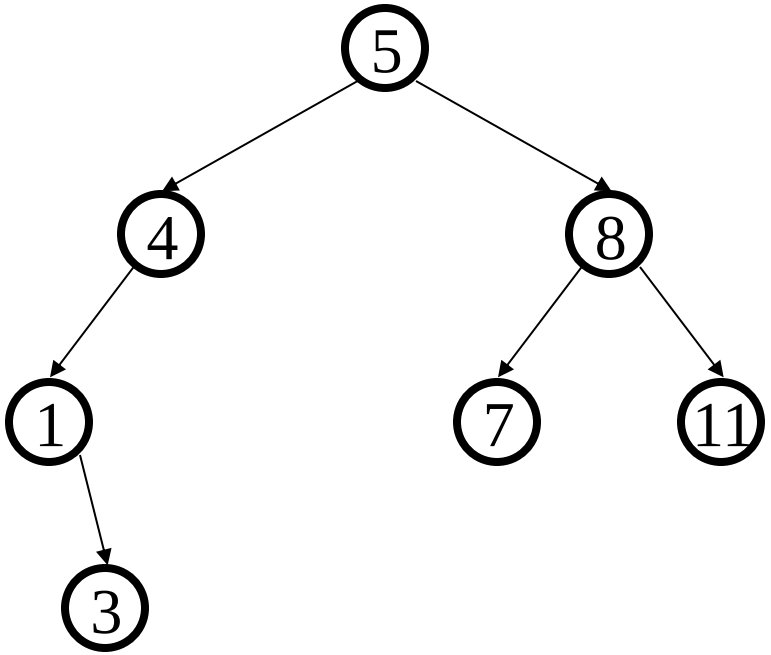
Search tree property

- All keys in left subtree smaller than root's key
- All keys in right subtree larger than root's key
- Result:
  - Easy to find any given key
  - Insert/delete by changing links

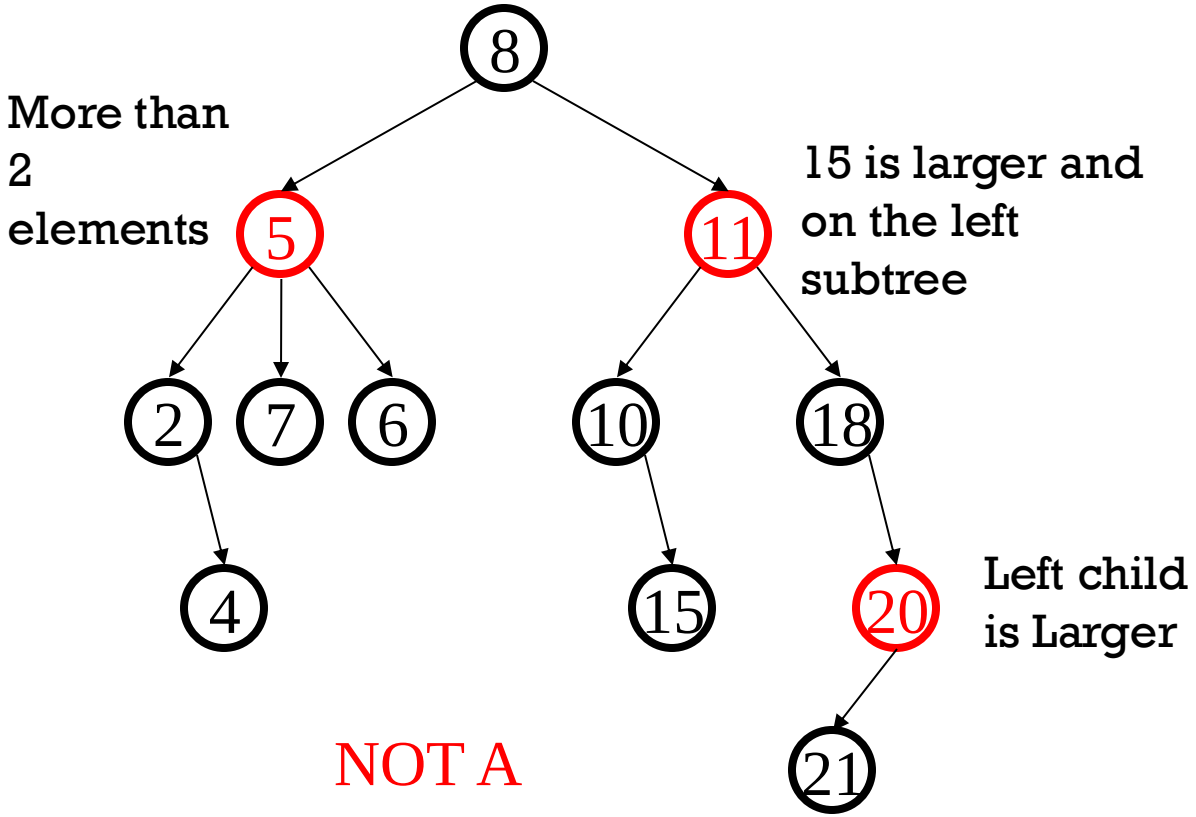


Binary Search Tree does not allow duplicate values

EXAMPLES

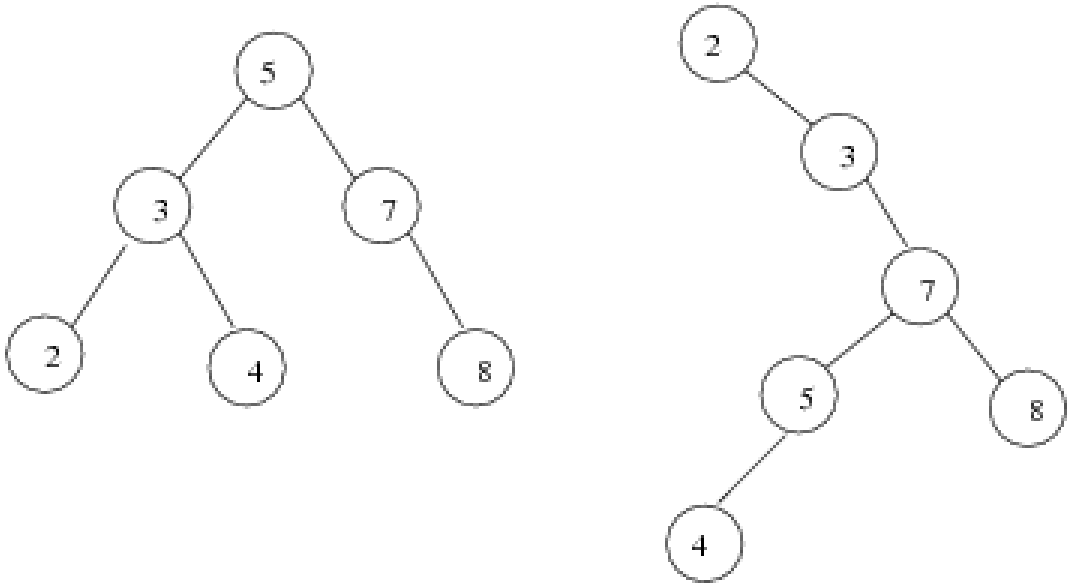


BINARY SEARCH TREE



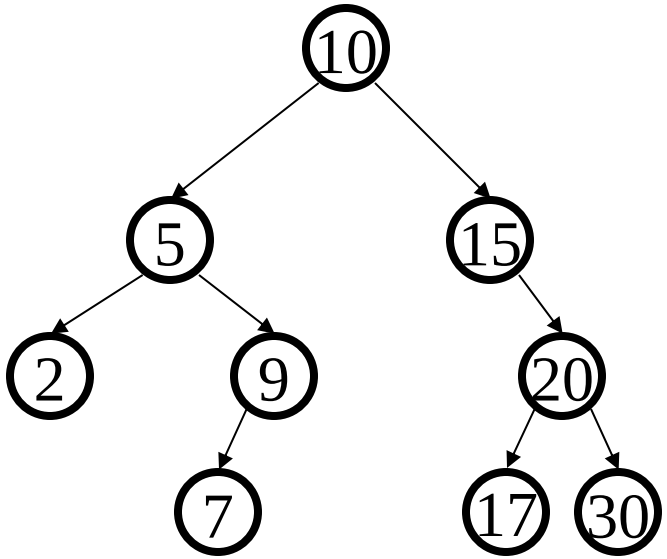
NOT A
BINARY SEARCH TREE

MULTIPLE TREES FOR SAME SET OF DATA



- Average depth of a node is O(log N); maximum depth of a node is O(N)

SEARCH - RECURSIVE

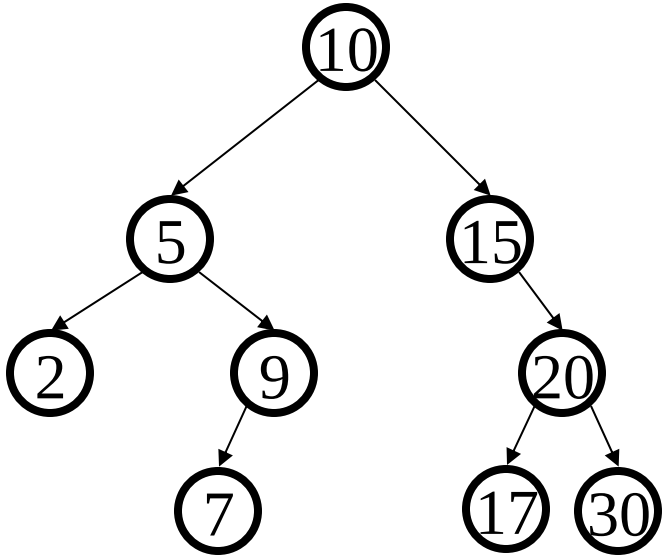



```
find(key, Node* t)
{
  if (t->key == key) return t;
  else if (key < t->key && t->left!=null)
    return find(key, t->left);
  else if (key > t->key && t->right!=null)
    return find(key, t->right);
  else
    return t;
}
```

Runtime:

- Best-worse case? O(log n)
- Worst-worse case? O(n)

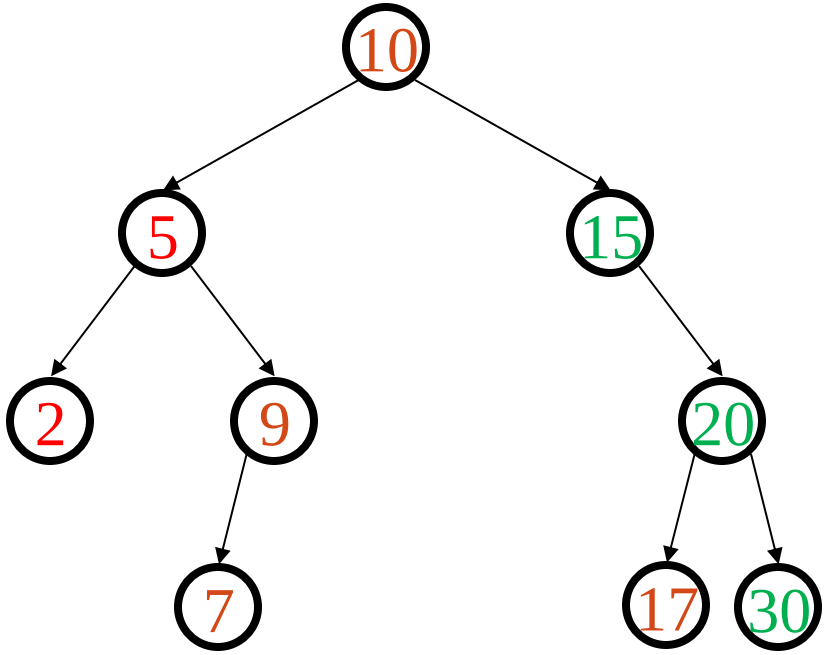
ITERATIVE FIND




```
find(key, Node* t)
{
  while (t != NULL && t->key != key)
  {
    if (key < t->key)
      t = t->left;
    else
      t = t->right;
  }

  return t;
}
```

FINDING MAX & MIN OF A BST

- Find minimum
  - Start from the root
  - Traverse left subtree recursively till you reach the left most leaf.
  - That is the minimum
- Find maximum
  - Start from the root
  - Traverse the right subtree recursively till you reach the right most leaf.
  - That is the maximum



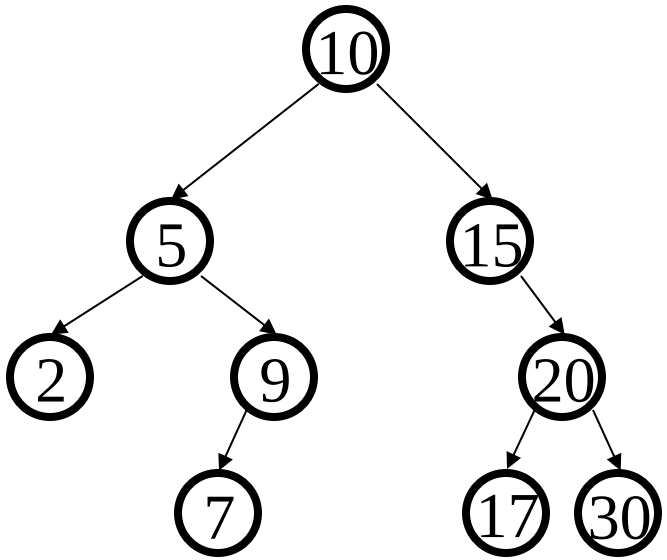
INSERT IN BINARY SEARCH TREE

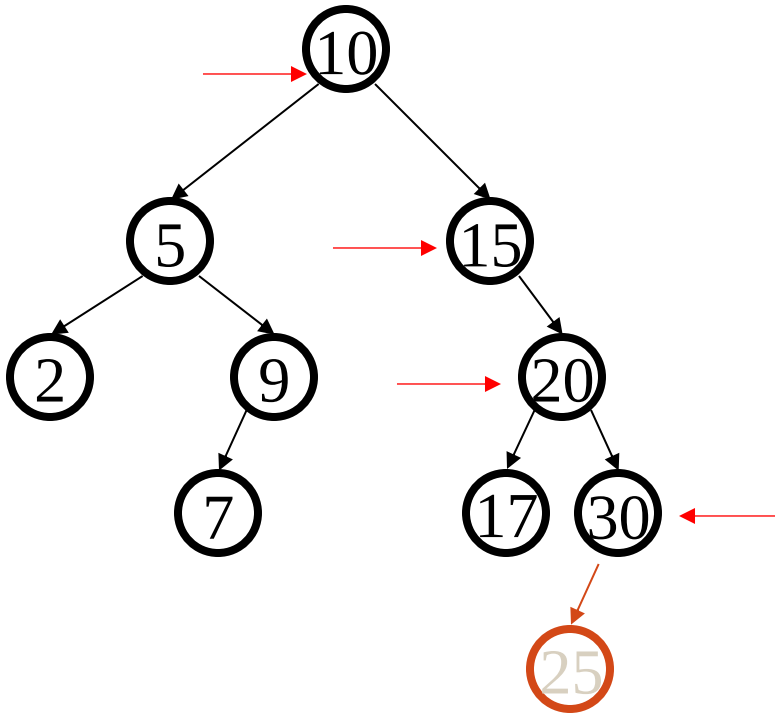
Concept:

- Proceed down tree as in Find
- If new key not found, then insert a new node at last spot traversed

```
insert(x, Node* t)
{
  if ( t == NULL ) {
    t = new Node(x);

  } else if (x < t->key) {
    insert( x, t->left );

  } else if (x > t->key) {
    insert( x, t->right );

  }
}
```

Insert(25)



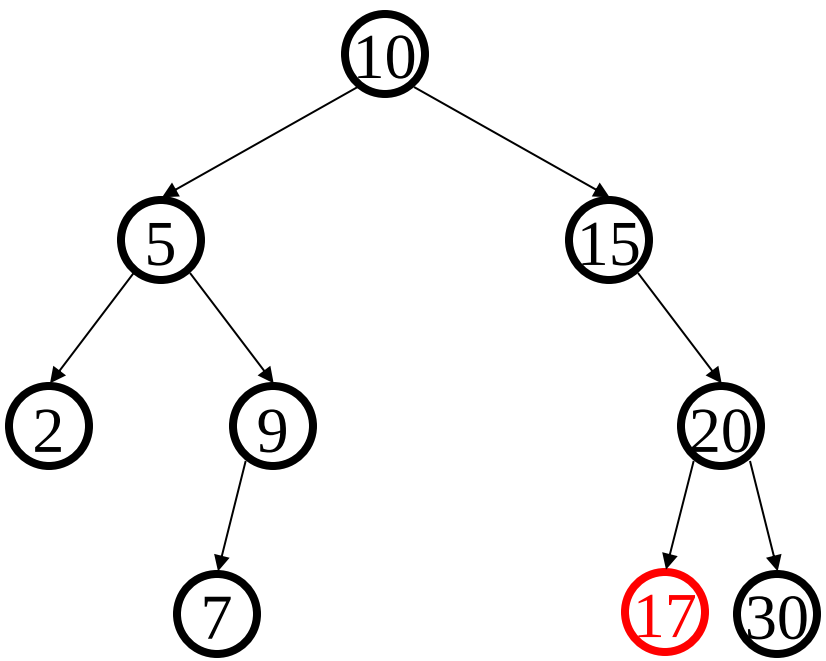


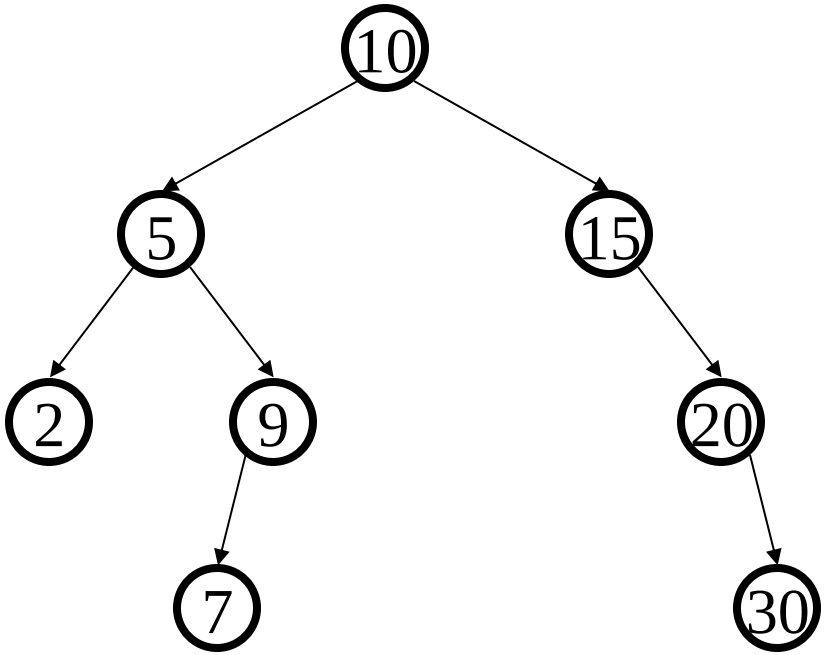
DELETE IN BST

- Case 1: Node to be deleted is a leaf
- Case 2: Node to be deleted has one child
- Case 3: Node to be deleted has two children

DELETION - LEAF CASE

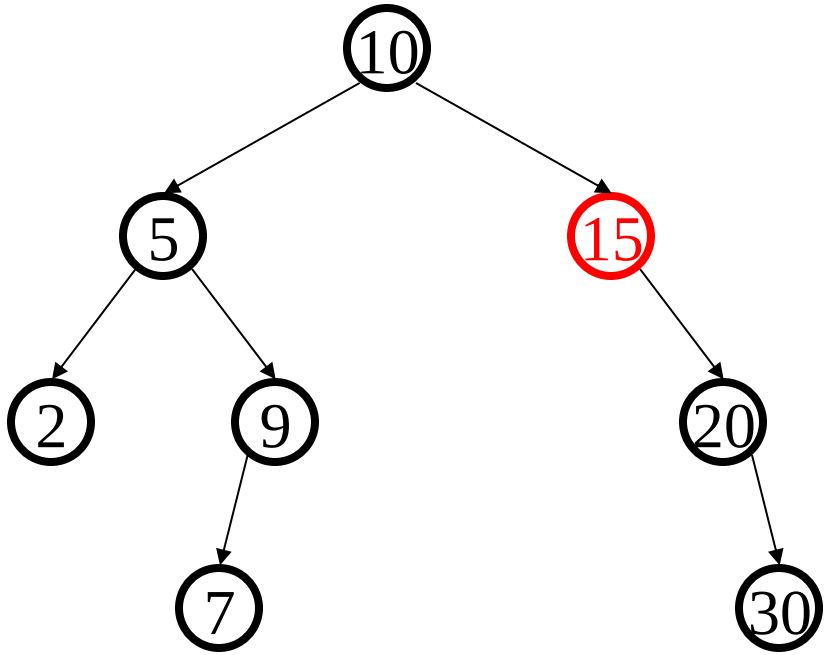
Delete(17)

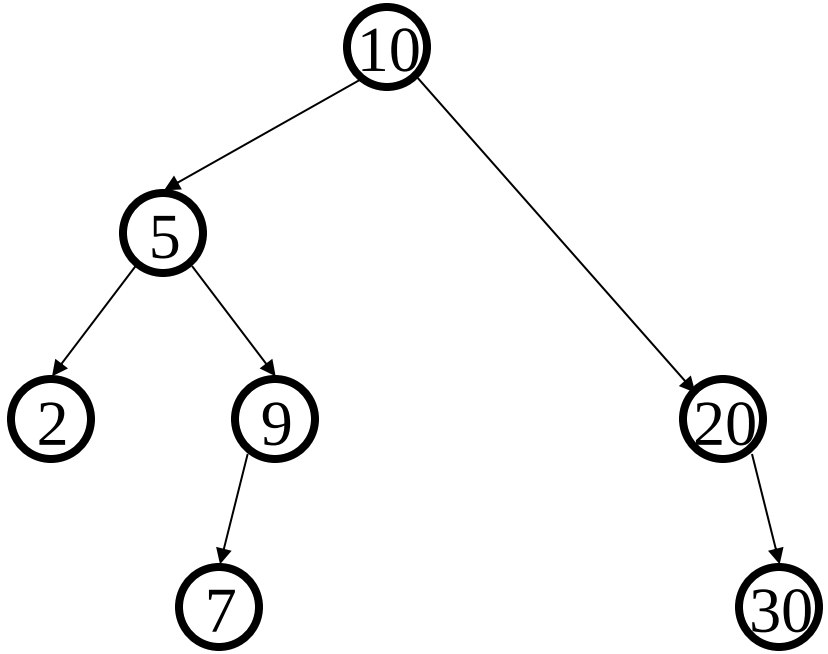




DELETION - ONE CHILD CASE

Delete(15)



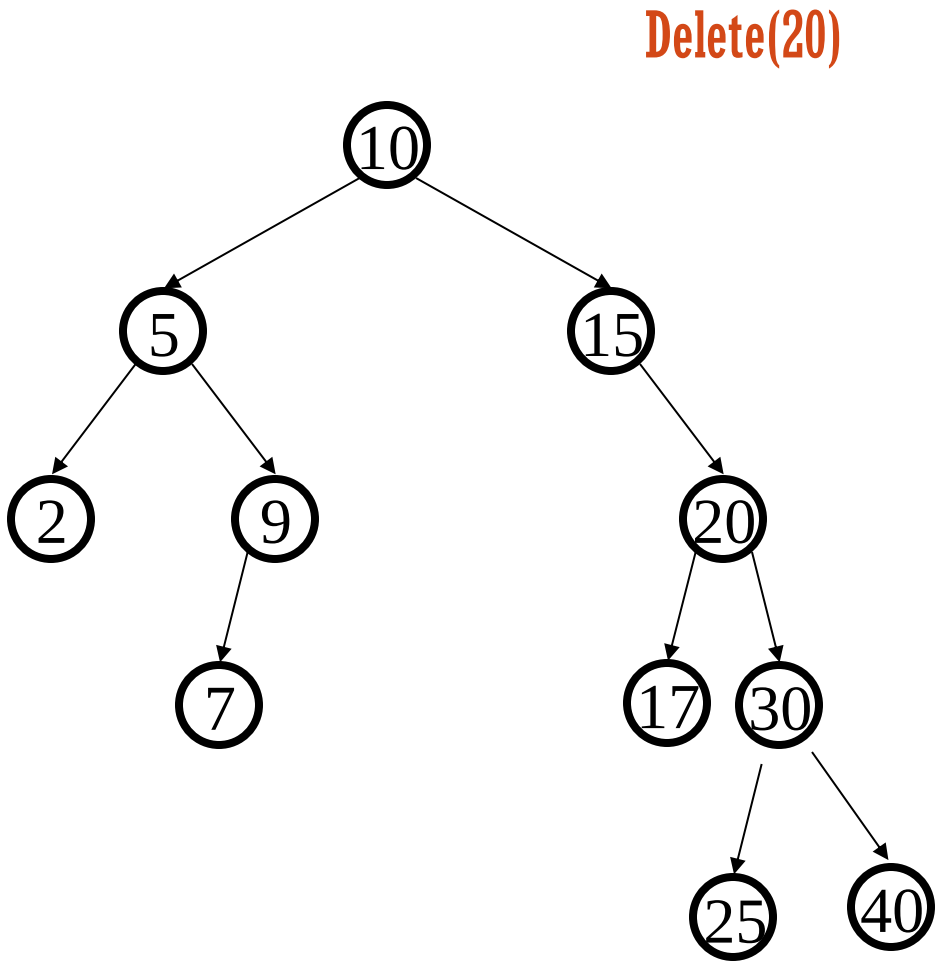


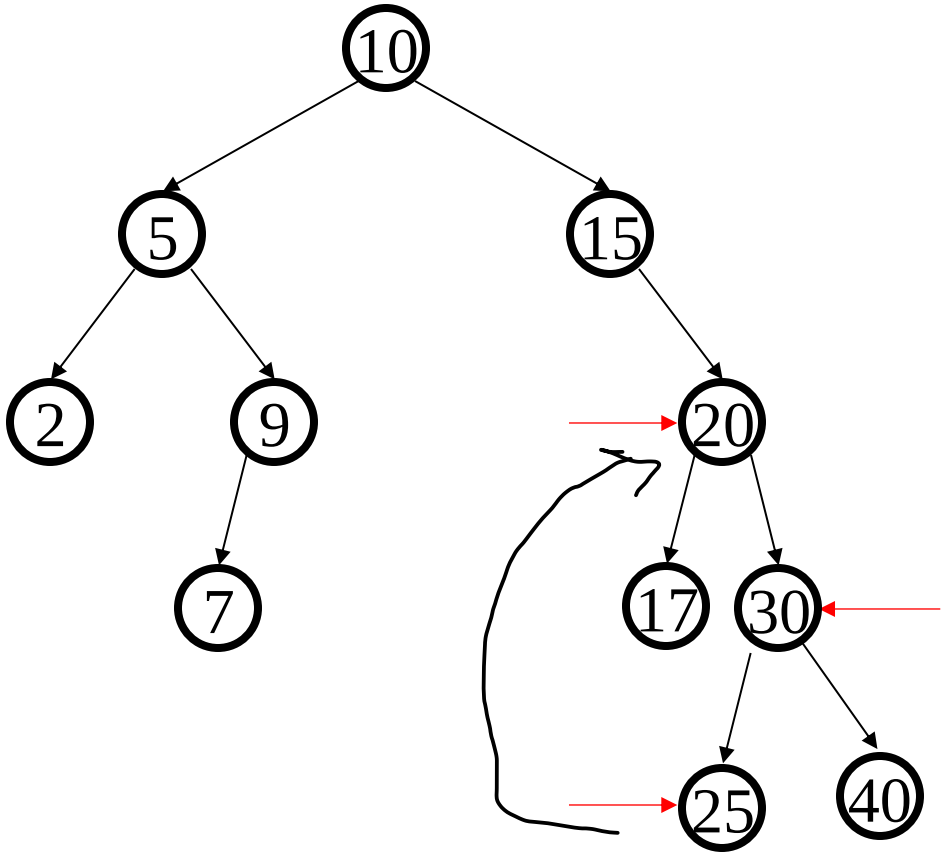
DELETION - TWO CHILD CASE

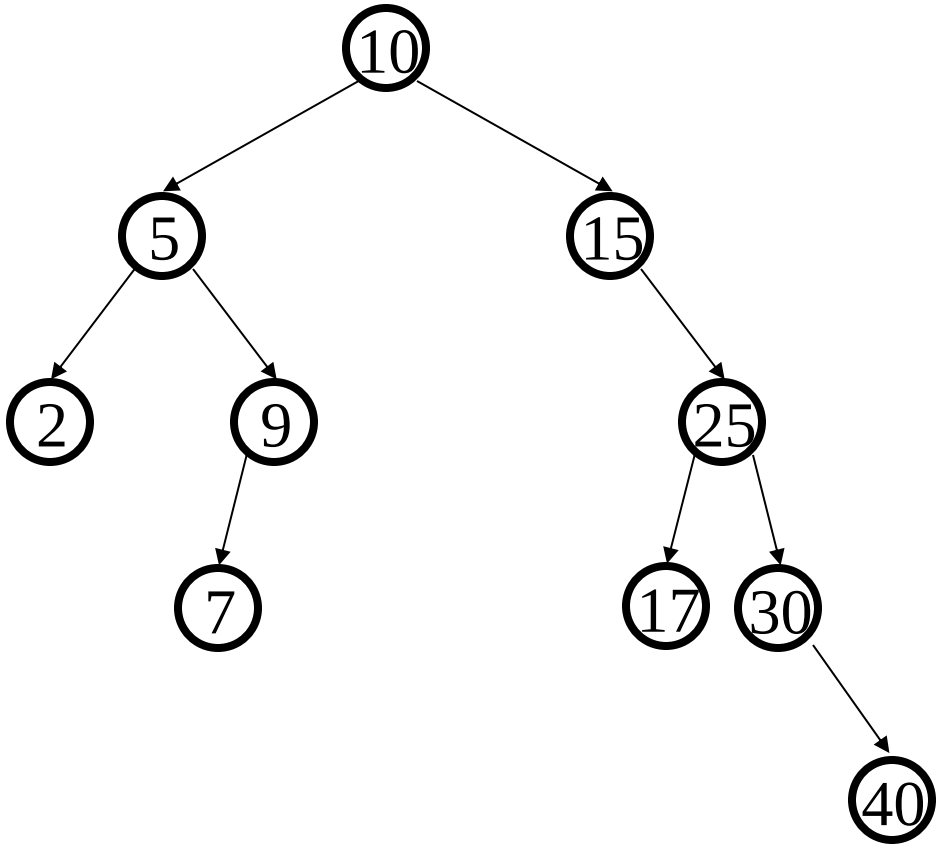
Replace node with descendant whose value is guaranteed to be between left and right subtrees: the successor

Steps:

1. Start from the right child of the node to be deleted.
2. Traverse down find the left most leaf.
3. Replace the node to be deleted with the leaf.




10
5
15
2
9
20
7
17
30
25
40


10
5
15
2
9
25
7
17
30
40

BINARY SEARCH TREE

Observations

- Elements (even siblings) may be scattered in memory
- Binary search trees are fast *if they're shallow*
- For large data sets, disk accesses dominate runtime
- Some deep and some shallow BSTs exist for any data

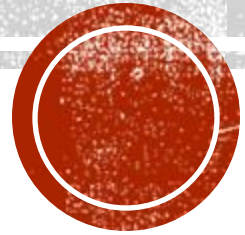
Binary Search Trees are fast if they're shallow:

- Full
- complete – possibly missing some "fringe" (leaves)
- any other good cases?
  - Shorter branches

TIME COMPLEXITIES

- **Searching:** worst case complexity of O(n). In general, time complexity is O(h) where **h** is height of BST.
- **Insertion:** worst case complexity of O(n). In general, time complexity is O(h).
- **Deletion:** worst case complexity of O(n). In general, time complexity is O(h).
- Best case for all three is O(logn)

DICTIONARY (MAP) ADT



DICTIONARIES

- A dictionary is a collection of elements each of which has a unique search key
  - Uniqueness criteria may be relaxed
- Keep track of current members, with periodic insertions and deletions into the set
- Examples
  - Membership in a club, course records
  - Language dictionary
- Similar to database

DICTIONARY ADT

Simple container methods: **size(), isEmpty(), elements()**

Query methods: **findElement(k) , findAllElements(k)**

Update methods: **insertItem(k, e), removeElement(k), removeAllElements(k)**

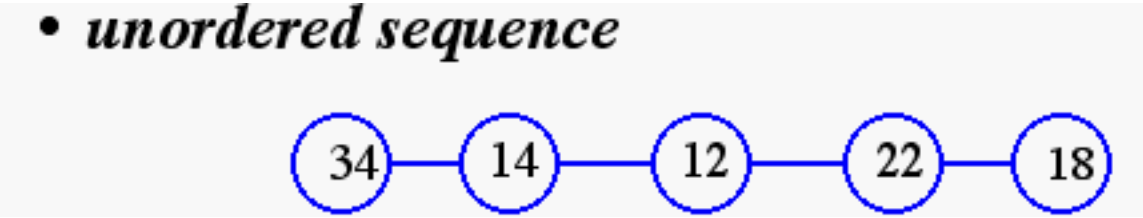
Special element : **NO_SUCH_KEY**, returned by an unsuccessful search

IMPLEMENTATION OF DICTIONARIES

- Sequences / Arrays
  - ordered
  - unordered
- Binary Search Trees
- Skip lists
- Hashtables

USING UNORDERED ARRAYS

- Unordered array



- Searching and Removing takes O(n) time
- Inserting takes O(1) time
- Application : Log files (frequent insertions, rare searches and removals)

USING ORDERED ARRAYS

- Searching takes O(log n) time (binary search)
- Inserting and Removing takes O(n) time
  - Not logn because even if the item can be found in log n time, array elements need to shifted down or up (arrays are static memory not linked)
- Application : Look-up tables (frequent searches, rare insertions and removals)

USING BINARY SEARCH TREES

- Implement a dictionary with a BST
  - Each internal node stores an item (k, e) of a dictionary.
  - Keys stored at nodes in the left subtree of v are less than k.
  - Keys stored at nodes in the right subtree of v are greater than or k.