Binary Search Trees

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees

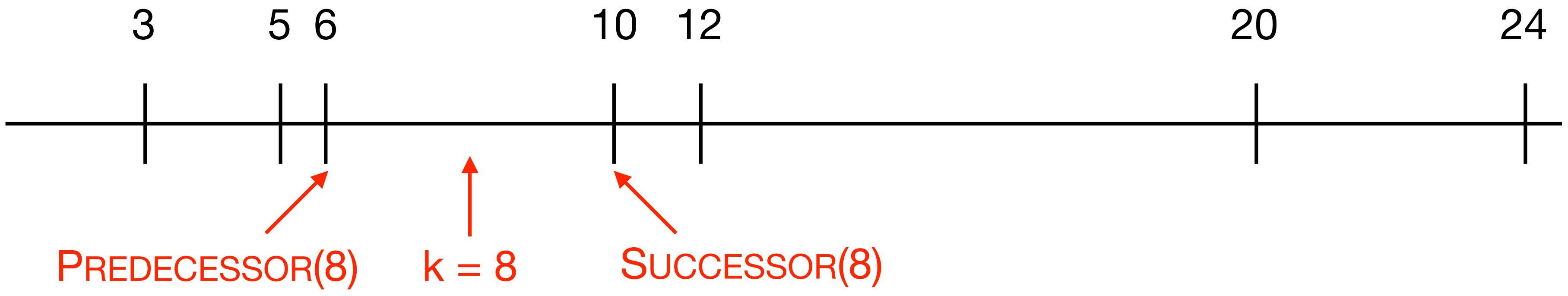
Philip Bille

Binary Search Trees

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees

Nearest Neighbor

- Nearest neighbor. Maintain dynamic set S supporting the following operations. Each element has key x.key and satellite data x.data.
  - PREDECESSOR(k): return element with largest key ≤ k.
  - SUCCESSOR(k): return element with smallest key ≥ k.
  - INSERT(x): add x to S (we assume x is not already in S)
  - DELETE(x): remove x from S.



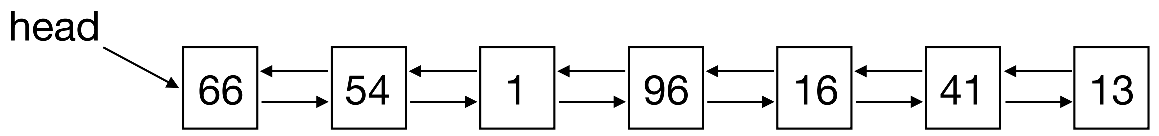
Nearest Neighbor

- Applications.
  - Searching for similar data (typically multidimensional)
  - Routing on the internet.

- Challenge. How can we solve problem with current techniques?

Nearest Neighbor

- Solution 1: linked list. Maintain S in a doubly-linked list.



- PREDECESSOR(k): linear search for largest key ≤ k.
- SUCCESSOR(k): linear for smallest key ≥ k.
- INSERT(x): insert x in the front of list.
- DELETE(x): remove x from list.

- Time.
  - PREDECESSOR and SUCCESSOR in O(n) time (n = |S|).
  - INSERT and DELETE in O(1) time.
- Space.
  - O(n).

Nearest Neighbor

- Solution 2: Sorted array. Maintain S in an sorted array.

1	2	3	4	5	6	7
1	13	16	41	54	66	96

- PREDECESSOR(k): binary search for largest key ≤ k.
- SUCCESSOR(k): binary search for smallest key ≥ k.
- INSERT(x): build new array of size +1 with x inserted.
- DELETE(x): build new array of size -1 with x removed.

- Time.
  - PREDECESSOR and SUCCESSOR in O(log n) time.
  - INSERT and DELETE in O(n) time.
- Space.
  - O(n).

Nearest Neighbor

Data structure	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Space
linked list	O(n)	O(n)	O(1)	O(1)	O(n)
sorted array	O(log n)	O(log n)	O(n)	O(n)	O(n)

- Challenge. Can we do significantly better?

Binary Search Trees

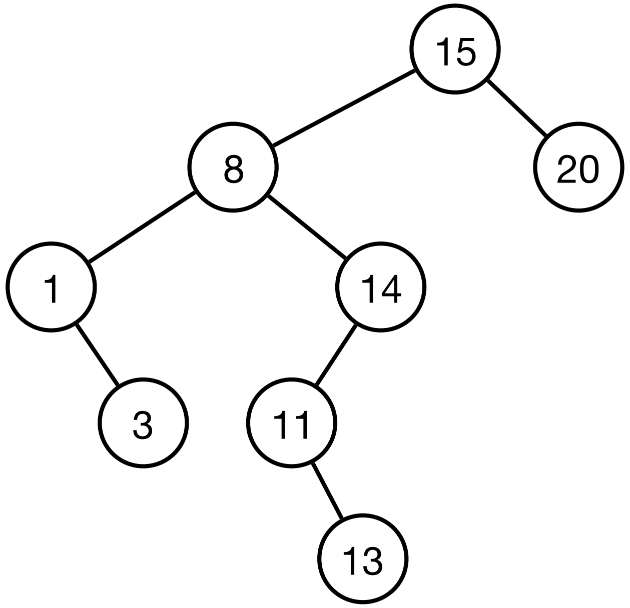
- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees

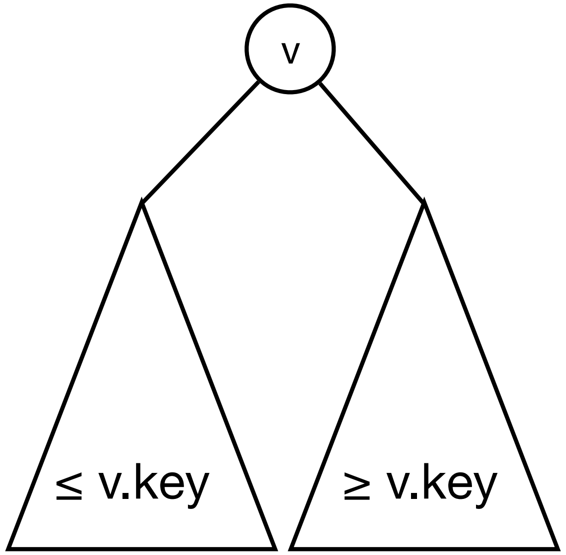
Binary Search Trees

- Binary tree. Rooted tree, where each internal vertex has a left child and/or a right child.

- Binary search tree. Binary tree that satisfies the search tree property.

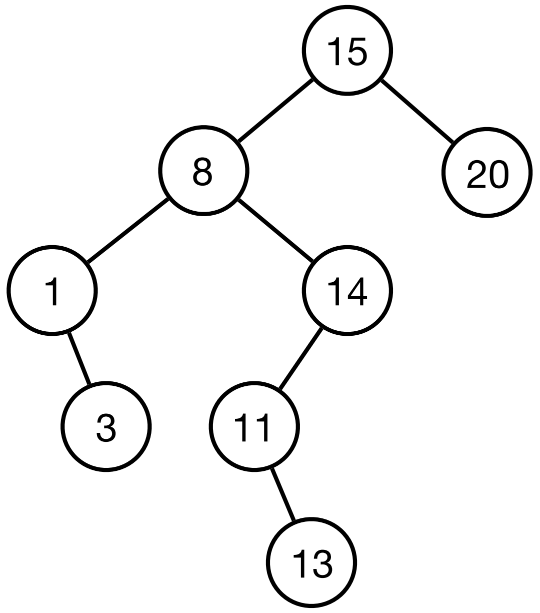
- Search tree property.
  - Each vertex stores an element.
  - For each vertex v:
    - all vertices in left subtree are ≤ v.key.
    - all vertices in right subtree are ≥ v.key.

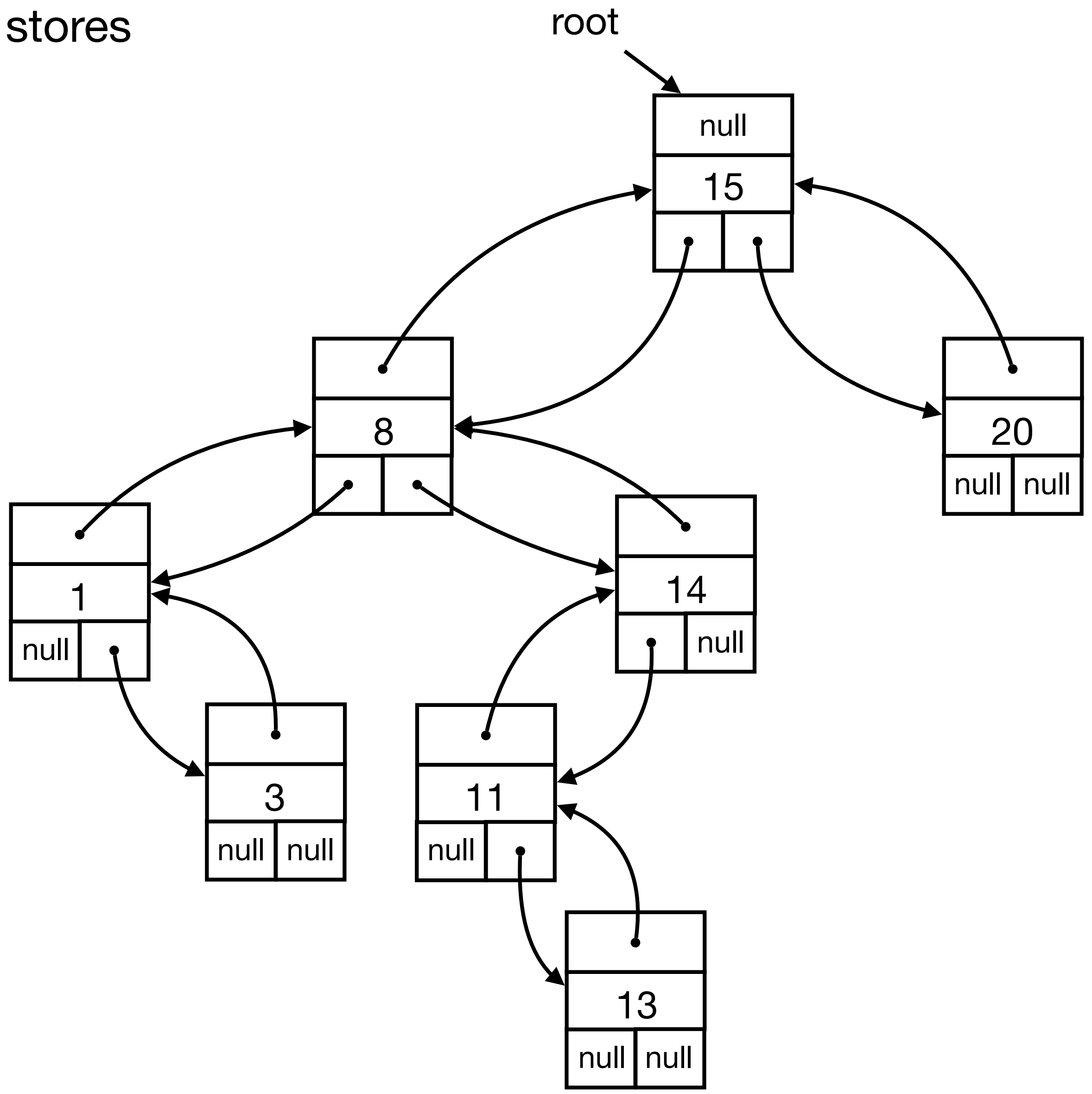




Binary Search Trees

- Representation. Each vertex x stores
  - x.key
  - x.left
  - x.right
  - x.parent
  - (x.data)
- Space. O(n)



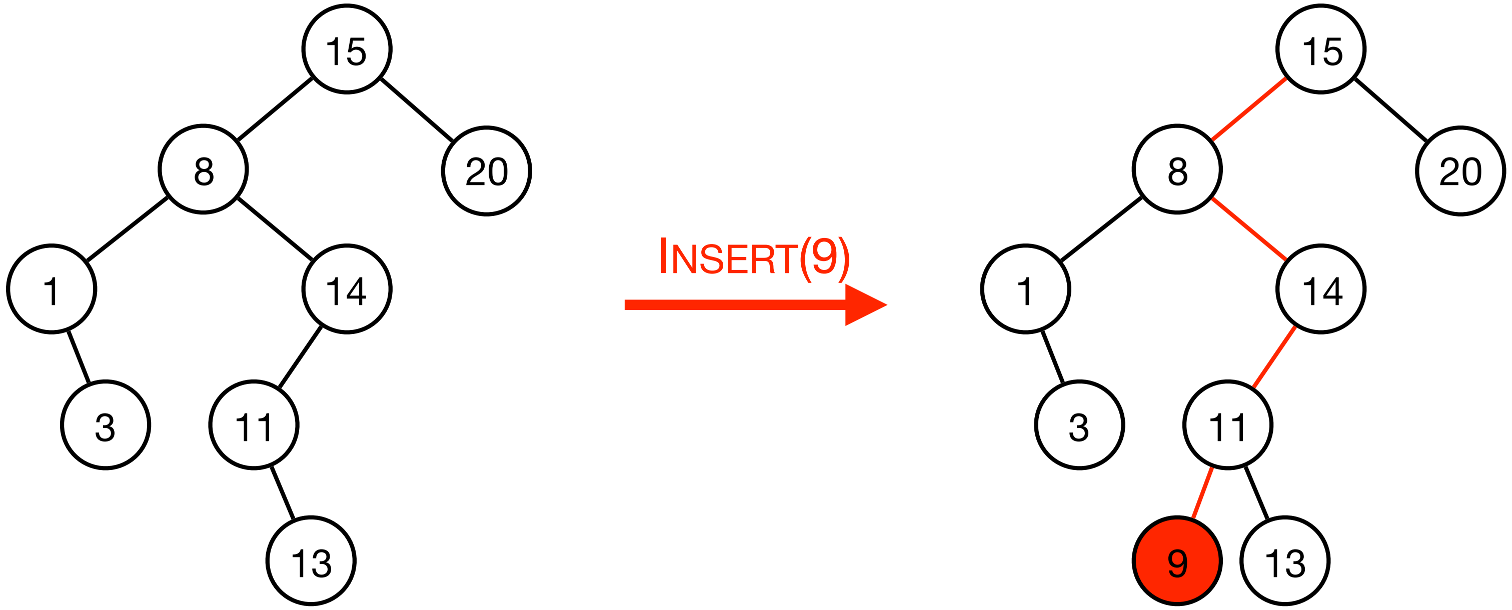


Binary Search Trees

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees

Insertion

- INSERT(x): start in root. At vertex v:
    - if x.key ≤ v.key go left.
    - if x.key > v.key go right.
    - if null, insert x



Insertion

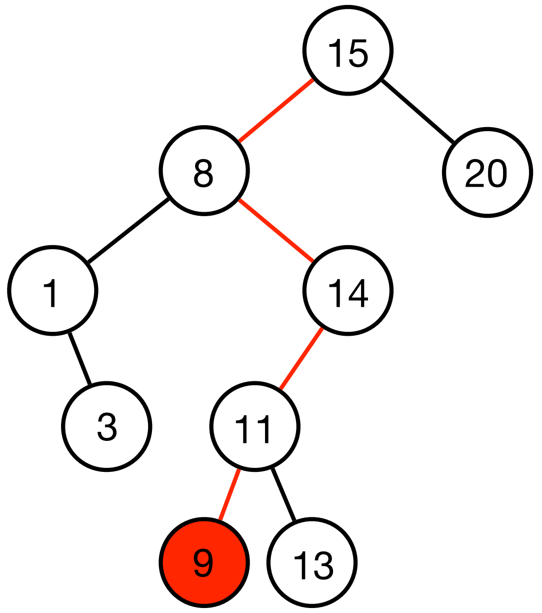
- INSERT(x): start in root. At vertex v:
    - if x.key ≤ v.key go left.
    - if x.key > v.key go right.
    - if null, insert x

- Exercise. Insert following sequence in binary search tree: 6, 14, 3, 8, 12, 9, 34, 1, 7

Insertion

```
INSERT(x,v)
   if (v == null) return x
   if (x.key ≤ v.key)
      v.left = INSERT(x, v.left)
   if (x.key > v.key)
      v.right = INSERT(x, v.right)
```

- Time. O(h)

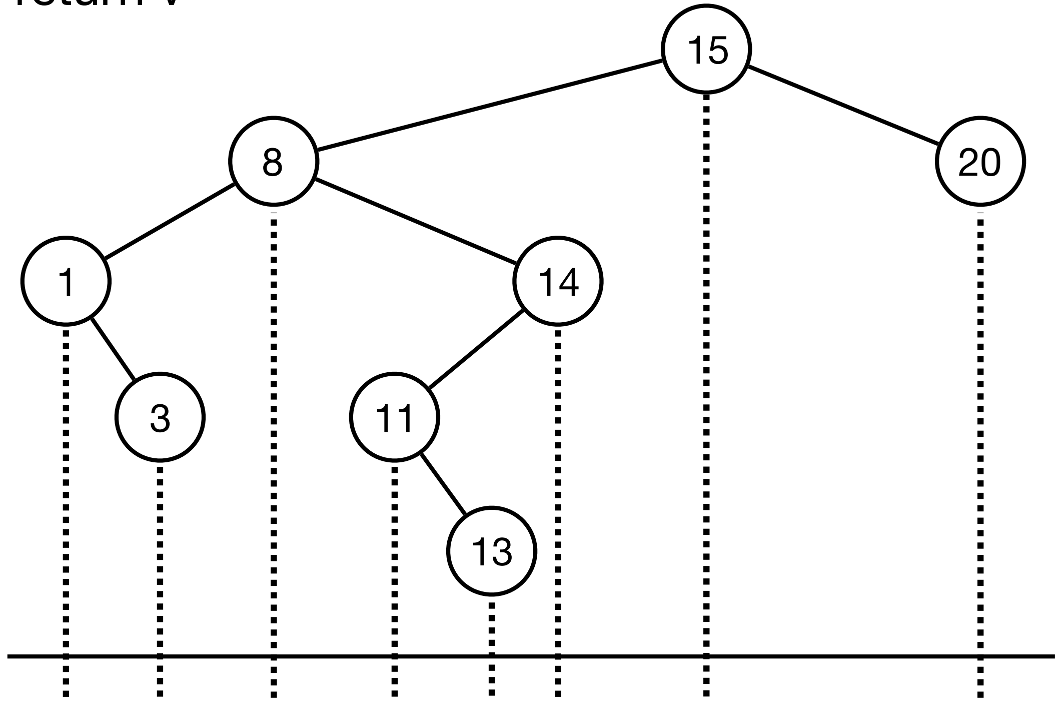


Binary Search Trees

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees

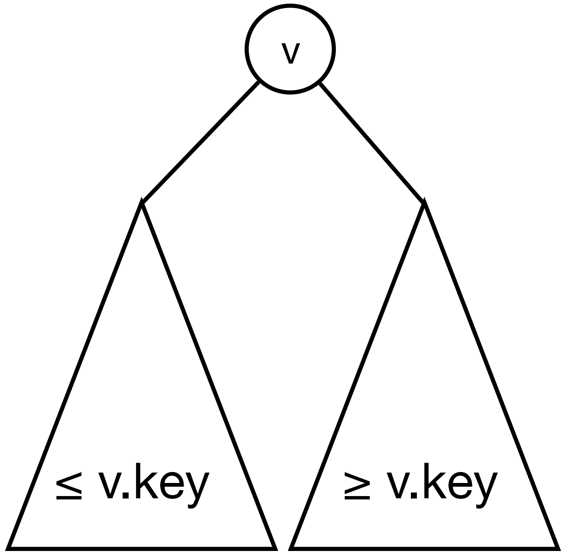
Predecessor

- PREDECESSOR(k): start in root. At vertex v:
  - if v == null: return null.
  - if k == v.key: return v.
  - if k < v.key: go left.
  - if k > v.key: search in right subtree.
    - If element x with key ≤ k in right subtree return x.
    - Otherwise, return v



Predecessor

```
PREDECESSOR(v, k)
   if (v == null) return null
   if (v.key == k) return v
   if (k < v.key)
      return PREDECESSOR(v.left, k)
   t = PREDECESSOR(v.right, k)
   if (t ≠ null) return t
   else return v
```



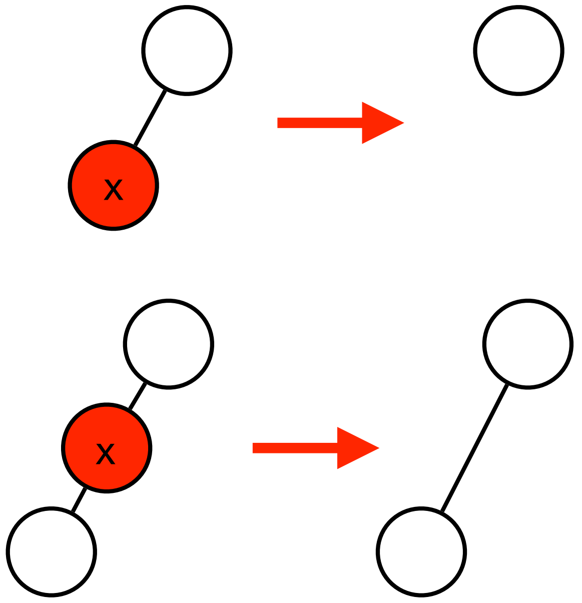
- Time. O(h)
- SUCCESSOR with similar algorithm in O(h) time.

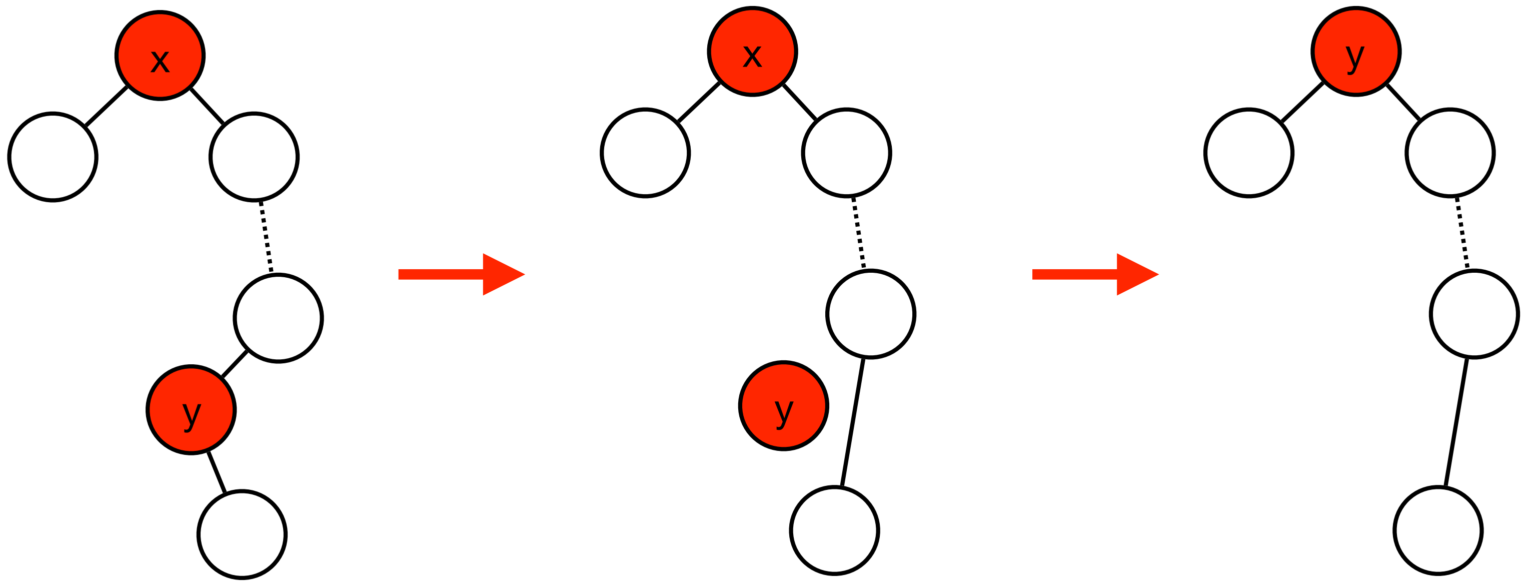
Binary Search Trees

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees

Deletion

- DELETE(x):
    - 0 children: remove x.
    - 1 child: splice x.
    - 2 children: find y = vertex with smallest key > x.key. Splice y and replace x by y.





Deletion

- DELETE(x):
  - 0 children: remove x.
  - 1 child: splice x.
  - 2 children: find y = vertex with smallest key > x.key. Splice y and replace x by y.
- Time. O(h)

Nearest Neighbor

Data structure	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Space
linked list	O(n)	O(n)	O(1)	O(1)	O(n)
sorted array	O(log n)	O(log n)	O(n)	O(n)	O(n)
binary search tree	O(h)	O(h)	O(h)	O(h)	O(n)
balanced binary search tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)

- Height. Depends on sequence of operations.
  - h = Ω(n) worst-case and h = Θ(log n) on average.
- Balanced binary search trees.
  - Possible to efficiently maintain binary search with height O(logn) (2-3 tree, AVL-trees, red-black trees, ..)
  - Even better bounds possible with advanced data structures.

Binary Search Trees

- Nearest neighbor
  - PREDECESSOR(k): return element with largest key ≤ k.
  - SUCCESSOR(k): return element with smallest key ≥ k.
  - INSERT(x): add x to S (we assume x is not already in S)
  - DELETE(x): remove x from S.

- Other operations on binary search trees.
  - SEARCH(k): determine if element with key k is in S and return it if so.
  - TREE-SEARCH(x, k): determine if element with key k is in subtree rooted at x and return it if so.
  - TREE-MIN(x): return the smallest element in subtree rooted at x.
  - TREE-MAX(x): return the largest element in subtree rooted at x.
  - TREE-PREDECESSOR(x): return element with largest key ≤ x.key.
  - TREE-SUCCESSOR(x): returner element with smallest key ≥ x.key.

Binary Search Trees

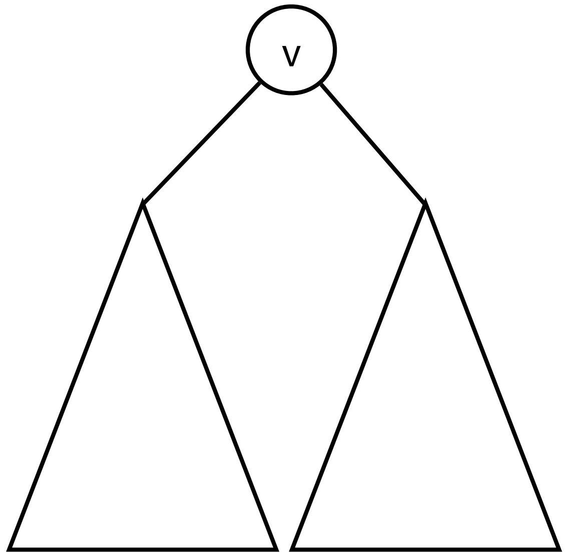
- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees

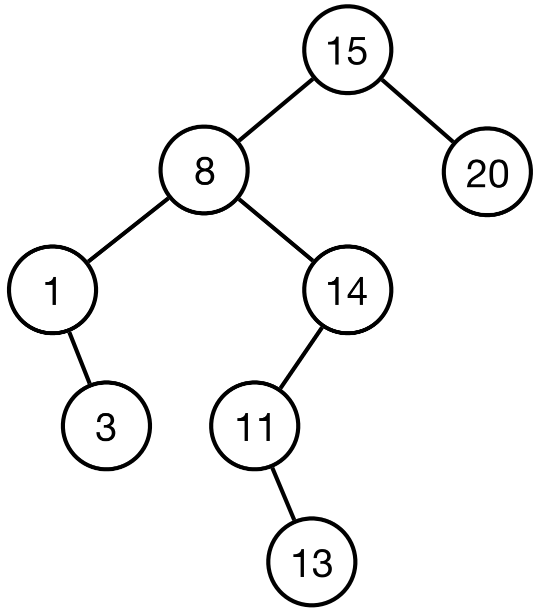
Algorithms on Trees

- Previous algorithms.
    - Heaps (MAX, EXTRACT-MAX, INCREASE-KEY, INSERT, …)
    - Union find (INIT, UNION, FIND, …)
    - Binary search trees (PREDECESSOR, SUCCESSOR, INSERT, DELETE, …)
- Challenge. How do we design algorithms on binary trees?

Algorithms on Trees

- Recursion on binary trees.
  - Solve problem on T(v):
    - Solve problem recursively on T(v.left) and T(v.right).
    - Combine to get solution for T(v).



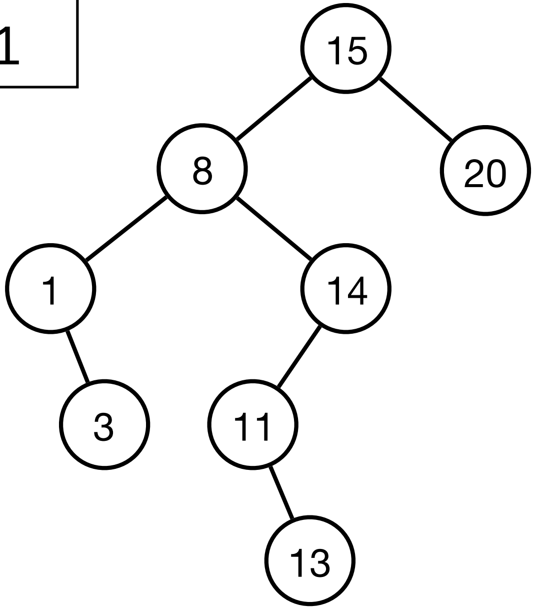


Algorithms on Trees

- Example. Compute size(v) (= number of vertices in T(v)).
  - If v is empty: size(v) = 0
  - Otherwise: size(v) = size(v.left) + size(v.right) + 1.

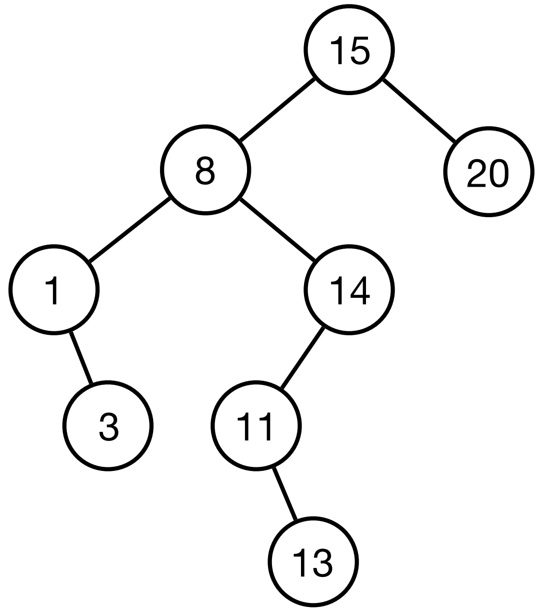
```
SIZE(v)
   if (v == null) return 0
   else return SIZE(v.left) + SIZE(v.right) + 1
```

- Time. O(size(v))



Tree Traversals

- Inorder traversal.
  - Visit left subtree recursively.
  - Visit vertex.
  - Visit right subtree recursively.
- Prints out the vertices in a binary search tree in sorted order.



- Preorder traversal.
  - Visit vertex.
  - Visit left subtree recursively.
  - Visit right subtree recursively.

- Postorder traversal.
  - Visit left subtree recursively.
  - Visit right subtree recursively.
  - Visit vertex.

Inorder: 1, 3, 8, 11, 13, 14, 15, 20

Preorder: 15, 8, 1, 3, 14, 11, 13, 20

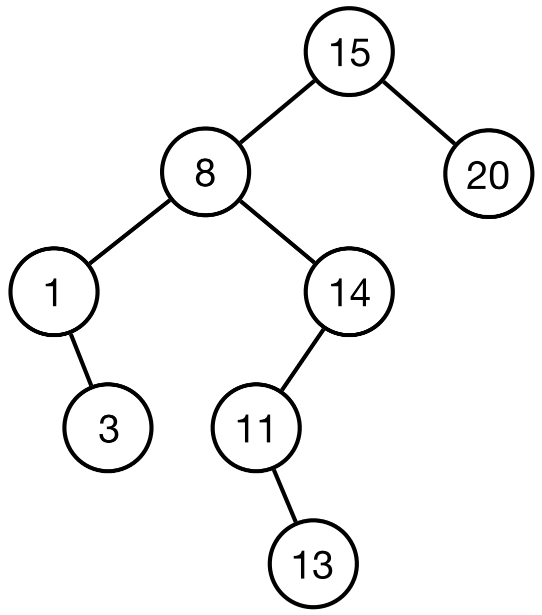
Postorder: 3, 1, 13, 11, 14, 8, 20, 15

Tree Traversals

```
INORDER(v)
    if (v == null) return
    INORDER(v.left)
    print v.key
    INORDER(v.right)
```

```
PREORDER(v)
    if (v == null) return
    print v.key
    PREORDER(v.left)
    PREORDER(v.right)
```

```
POSTORDER(v)
    if (v == null) return
    POSTORDER(v.left)
    POSTORDER(v.right)
    print v.key
```



Inorder: 1, 3, 8, 11, 13, 14, 15, 20

Preorder: 15, 8, 1, 3, 14, 11, 13, 20

Postorder: 3, 1, 13, 11, 14, 8, 20, 15

- Time. O(n)

Binary Search Trees

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees