Algorithms

Red-Black Trees

Review: Binary Search Trees

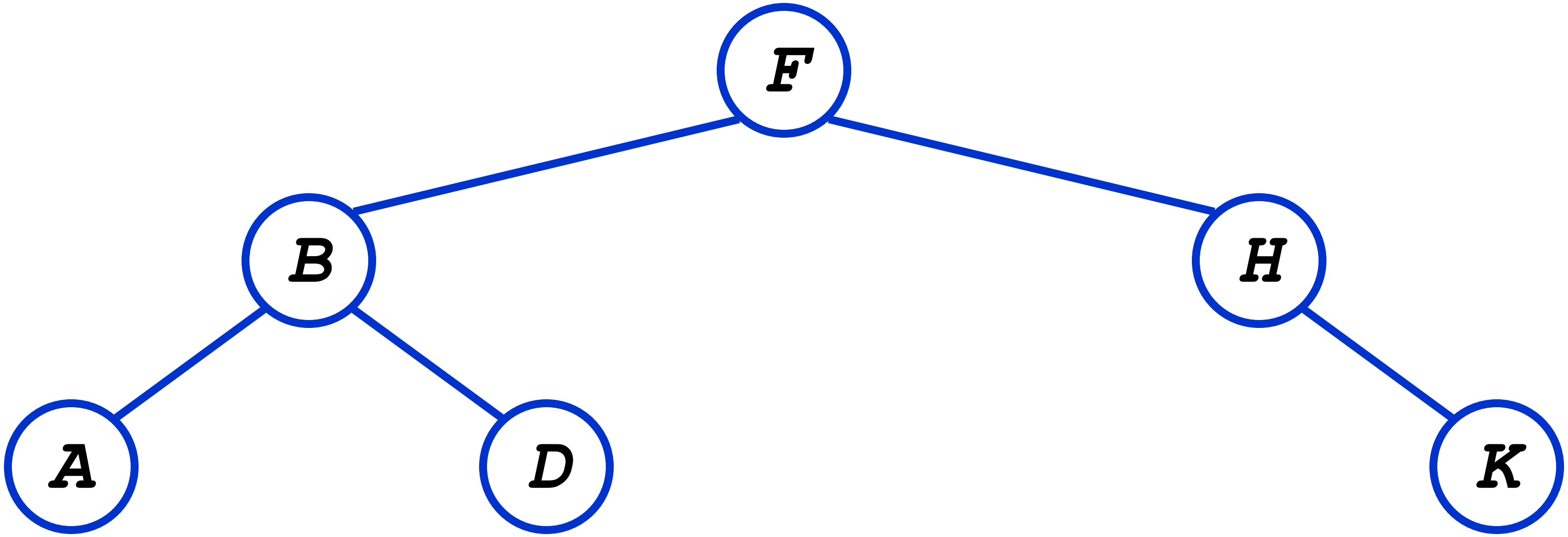
- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, eleements have:
  - *key*: an identifying field inducing a total ordering
  - *left*: pointer to a left child (may be NULL)
  - *right*: pointer to a right child (may be NULL)
  - *p*: pointer to a parent node (NULL for root)

Review: Binary Search Trees

- BST property:

  key[left(x)] ≤ key[x] ≤ key[right(x)]

- Example:



Review: Inorder Tree Walk

- An *inorder walk* prints the set in sorted order:

```
TreeWalk(x)
    TreeWalk(left[x]);
    print(x);
    TreeWalk(right[x]);
```

  - Easy to show by induction on the BST property
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

Review: BST Search

```
TreeSearch(x, k)
    if (x = NULL  or  k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);
```

Review: BST Search (Iterative)

```
IterativeTreeSearch(x, k)
    while (x != NULL  and  k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

Review: BST Insert

- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a “trailing pointer” to keep track of where you came from (like inserting into singly linked list)
- Like search, takes time O(*h*), *h* = tree height

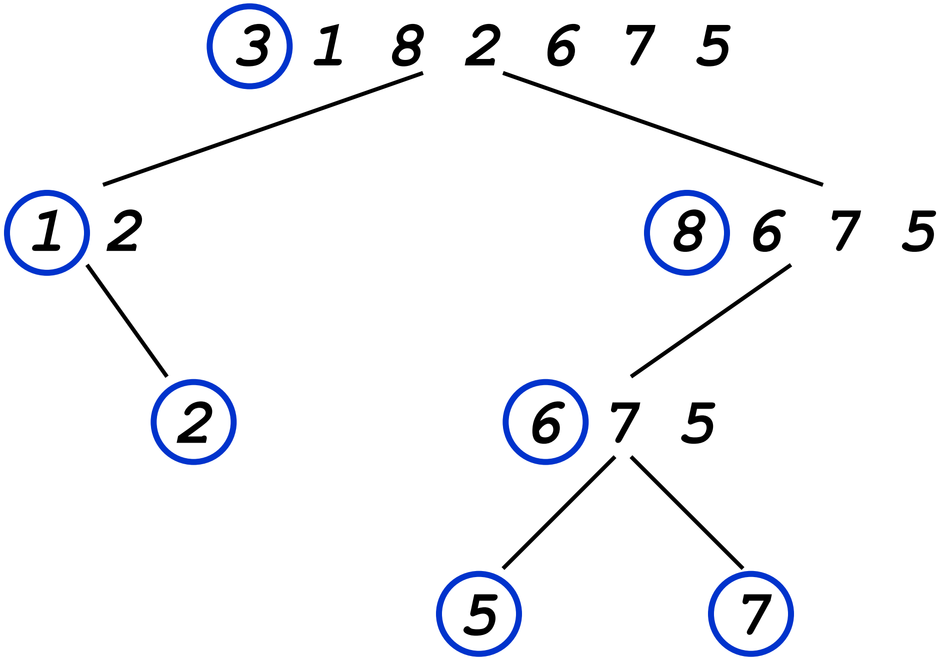
Review: Sorting With BSTs

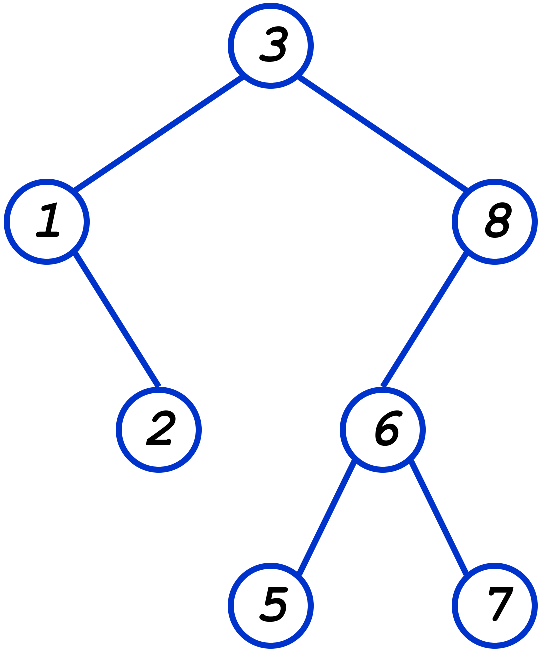
- Basic algorithm:
  - Insert elements of unsorted array from 1..n
  - Do an inorder tree walk to print in sorted order
- Running time:
  - Best case: $\Omega(n \lg n)$ (it's a comparison sort)
  - Worst case: $O(n^2)$
  - Average case: $O(n \lg n)$ (it's a quicksort!)

Review: Sorting With BSTs

- Average case analysis
  - It's a form of quicksort!

```
for i=1 to n
     TreeInsert(A[i]);
InorderTreeWalk(root);
```



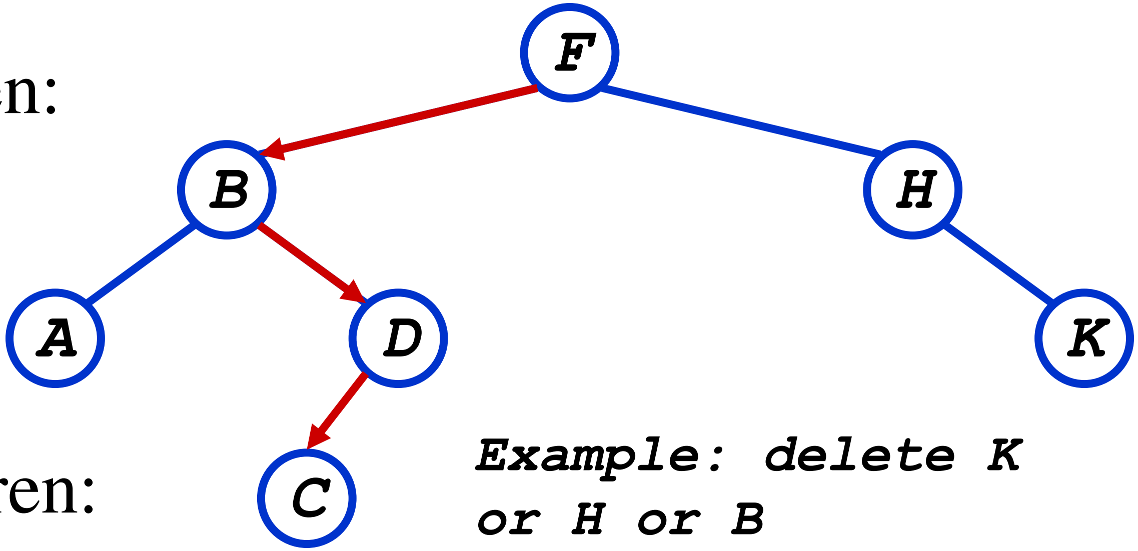


Review: More BST Operations

- Minimum:
  - Find leftmost node in tree
- Successor:
  - x has a right subtree: successor is minimum node in right subtree
  - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you’re visiting smaller nodes.
- Predecessor: similar to successor

Review: More BST Operations

- Delete:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Swap x with successor
    - Perform case 1 or 2 to delete it



Example: delete K or H or B

Red-Black Trees

- *Red-black trees*:
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height $h = O(\lg n)$
- First: describe the properties of red-black trees
- Then: prove that these guarantee $h = O(\lg n)$
- Finally: describe operations on red-black trees

Red-Black Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every “real” node has 2 children
  3. If a node is red, both children are black
     - Note: can’t have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

Red-Black Trees

- Put example on board and verify properties:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
  3. If a node is red, both children are black
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black
- *black-height*: # black nodes on path to leaf
  - Label example with *h* and bh values

Height of Red-Black Trees

- *What is the minimum black-height of a node with height h?*
- A: a height-h node has black-height $\geq h/2$
- Theorem: A red-black tree with n internal nodes has height $h \leq 2 \lg(n + 1)$
- *How do you suppose we'll prove this?*

RB Trees: Proving Height Bound

- Prove: n-node RB tree has height $h \leq 2\lg(n+1)$
- Claim: A subtree rooted at a node x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes
  - Proof by induction on height h
  - Base step: x has height 0 (i.e., NULL leaf node)
    - *What is bh(x)?*

RB Trees: Proving Height Bound

- Prove: n-node RB tree has height $h \leq 2 \lg(n+1)$
- Claim: A subtree rooted at a node x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes
  - Proof by induction on height h
  - Base step: x has height 0 (i.e., NULL leaf node)
    - *What is bh(x)?*
    - A: 0
    - So…subtree contains $2^{\mathrm{bh}(x)} - 1$
      $= 2^0 - 1$
      $= 0$ internal nodes (TRUE)

RB Trees: Proving Height Bound

- Inductive proof that subtree at node x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes
  - Inductive step: x has positive height and 2 children
    - Each child has black-height of $\mathrm{bh}(x)$ or $\mathrm{bh}(x)-1$ *(Why?)*
    - The height of a child = (height of x) - 1
    - So the subtrees rooted at each child contain at least $2^{\mathrm{bh}(x) - 1} - 1$ internal nodes
    - Thus subtree at x contains
      $(2^{\mathrm{bh}(x) - 1} - 1) + (2^{\mathrm{bh}(x) - 1} - 1) + 1$
      $= 2 \bullet 2^{\mathrm{bh}(x)-1} - 1 = 2^{\mathrm{bh}(x)} - 1$ nodes

RB Trees: Proving Height Bound

- Thus at the root of the red-black tree:

  $n \geq 2^{\mathrm{bh}(root)} - 1$ *(Why?)*

  $n \geq 2^{h/2} - 1$ *(Why?)*

  $\lg(n+1) \geq h/2$ *(Why?)*

  $h \leq 2 \lg(n + 1)$ *(Why?)*

  Thus $h = O(\lg n)$

RB Trees: Worst-Case Time

- So we’ve proved that a red-black tree has O(lg *n*) height
- Corollary: These operations take O(lg *n*) time:
  - Minimum(), Maximum()
  - Successor(), Predecessor()
  - Search()
- Insert() and Delete():
  - Will also take O(lg *n*) time
  - But will need special care since they modify tree