Search Trees and Augmenting Data Structures

CS 4231, Fall 2017 Mihalis Yannakakis

Dictionary
Search/ Index structure

- Maintain set S of items, each item has key and other info
- Basic Operations:
- Insert an item x
- Search for an item with given key k

- Other operations: empty?, size, list all items
- delete an item (specified by handle or key)
- join (union) two sets
- min, max, successor (ceiling), predecessor (floor) (in presence of <)
- Duplicate key issue: may assume no duplicate keys or allow them

Applications

- Dictionary key=word, info=translation, definition etc
- Phone book: key=name, info=number (or vice-versa)
- File system: key=file name -> location on disc
- Book index: key=term -> list of pages
- Web index: keyword -> list of documents/ web pages
- DNS: URL -> IP address
- Routing table: destination -> network route
- Compiler: variable name -> type and value
- Genome database: DNA string (marker) -> positions

Domain (Type) of keys: easy case

- If small domain D = {1,...,k} , i.e. can treat key as an index for array, then easy:
- Array A[ ] where A[i] contains the nullitem (if set S has no item with key=i) or the item with key=i.
- If allow duplicate-key items then A[i]=list of items with key=i
- Special case where no other info, i.e. just want to maintain a set S of keys , then A[] is a bit array:
- A[i] = 1 iff i $\in$S else A[i] =0

- Complexity: O(1) time for search, insert, delete
- Need more to support min, max (if we have deletions)

General Domain of keys

- Dictionary == Associative array = Index structure: Behaves like an array but indexed with keys from a general domain

- Approach 1: Comparison-based.
- Domain has < (besides =) operator
- Supports also some more operations: sorted listing, min, max, successor, predecessor

- Approach 2: Hashing
- Map large domain to small domain {1,...,k}

- Approach 3: Radix-based for keys that are strings (tuples)

Elementary data structures

	Search	Insert
• Unsorted array	N	1
• Sorted array	logN	N
• Unsorted list	N	1
• Sorted list	N	N

- If mostly search operations, or static set (all inserts at the beginning, eg. language dictionary) -> sorted array + binary search good solution (or hash table)
- If mix of operations, want to do better

Binary Search Trees

- Data structure for maintaining a set S of elements with keys drawn from an ordered domain, that supports the dictionary operations
- Search
- Insert
- Delete

\+ other operations

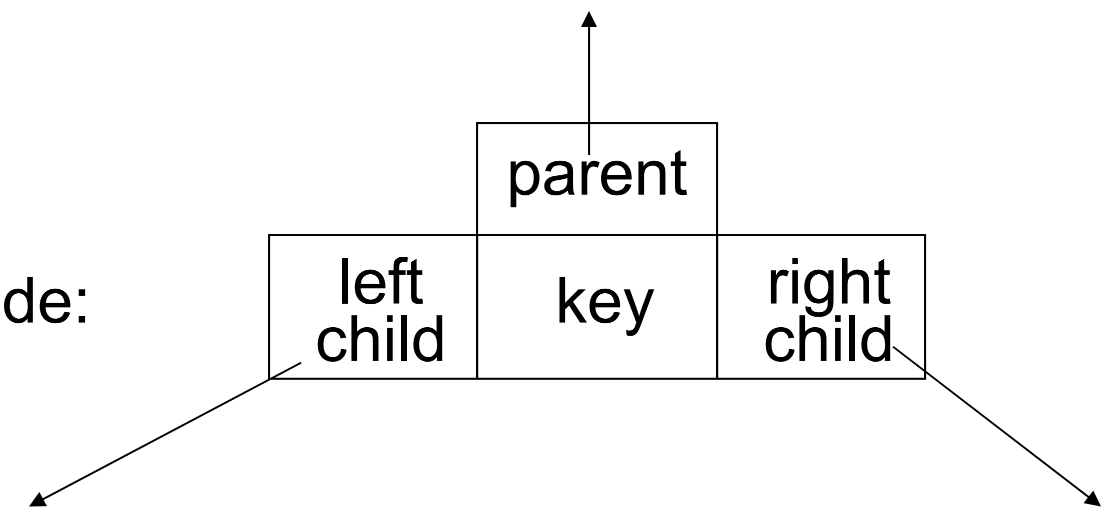
- Min, Max, …

Binary Search Tree

- Binary tree
- Keys at the nodes, 1-1 correspondence
- Order property of BST: $\forall$ node v

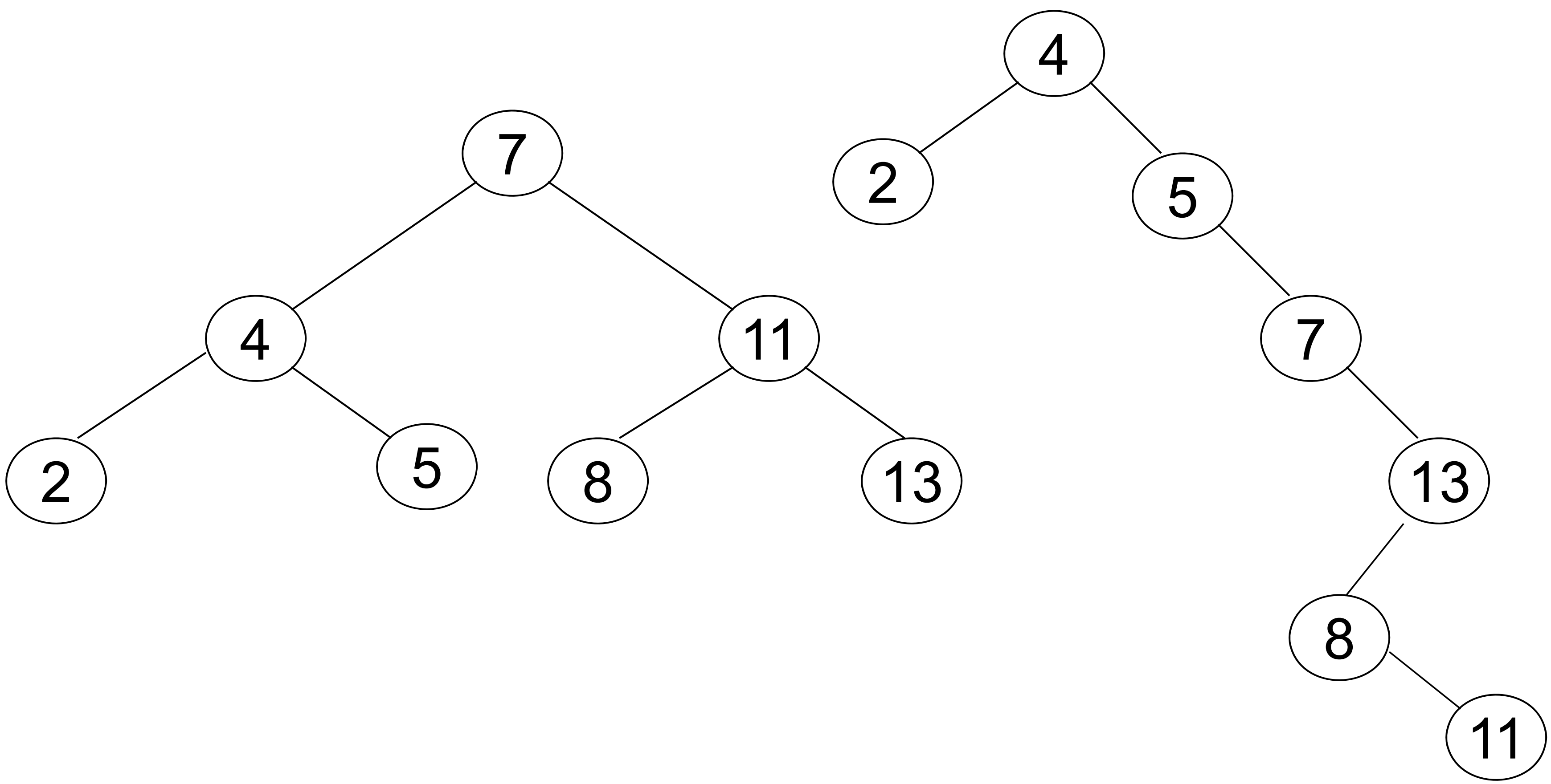
keys in left subtree ≤ key(v) ≤ keys in right subtree

Node:



Examples

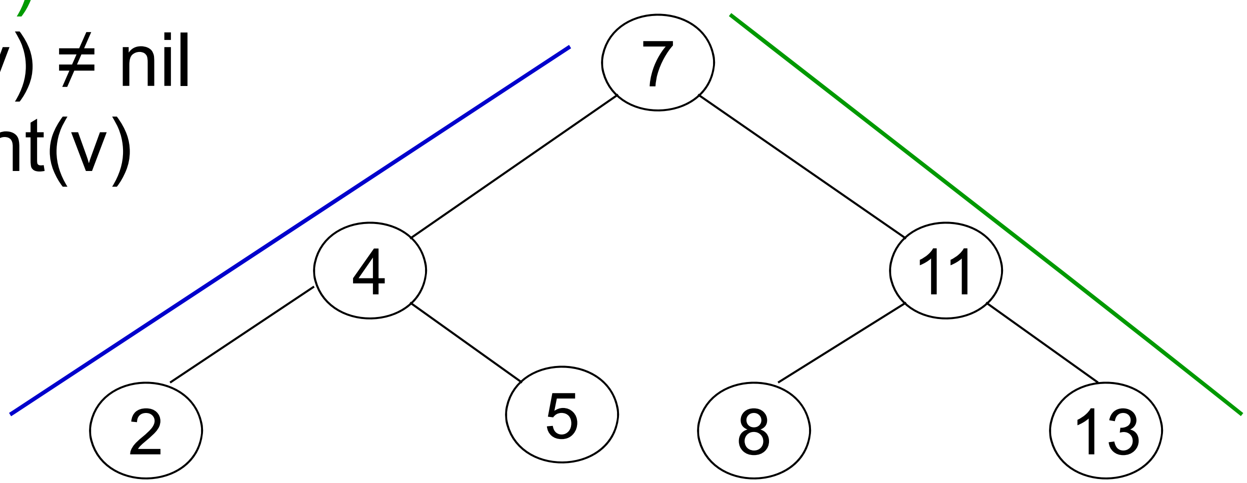
- Tree may not be balanced
- height(T) between logn and n



Min and Max

- Min(node v)
  while left(v) ≠ nil
  do v = left(v)
  return v

- Max(node v)
  while right(v) ≠ nil
  do v = right(v)
  return v

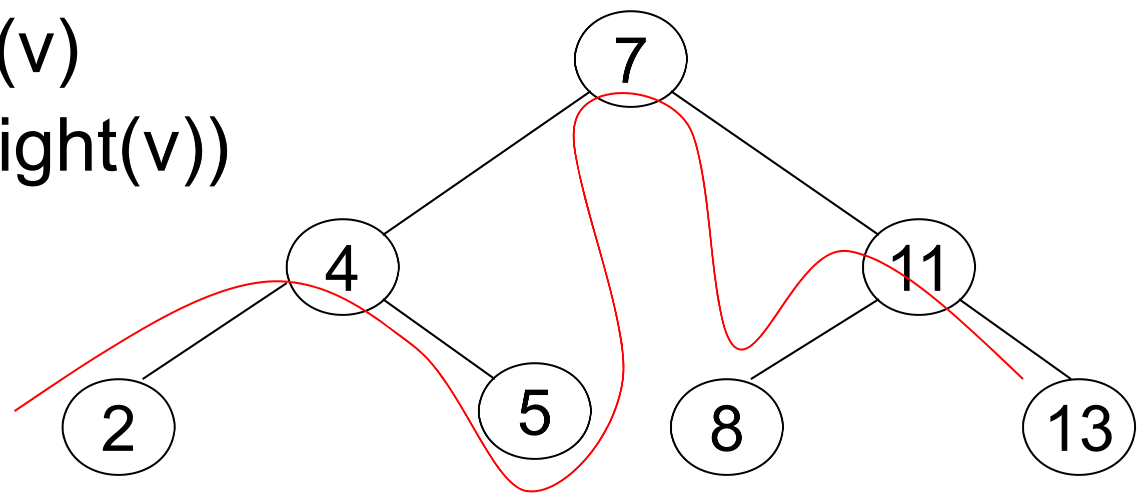


Sorted Listing of Set S

- Inorder walk of the tree : O(n) time

InorderWalk(v)

if v ≠ nil then
    { Inorder(left(v))
      print key(v)
      Inorder(right(v))
    }



Search

- Start at the root
- Search(node v, key k)

  if v=nil or k=key(v) then return v

  if k < key(v) then return Search(left(v), k)

  else return Search(right(v), k)

Complexity : O(h),   h=height of the tree

Search: Iterative version

- Start at the root
- Search(node v, key k)

while v≠nil and k≠key(v) do

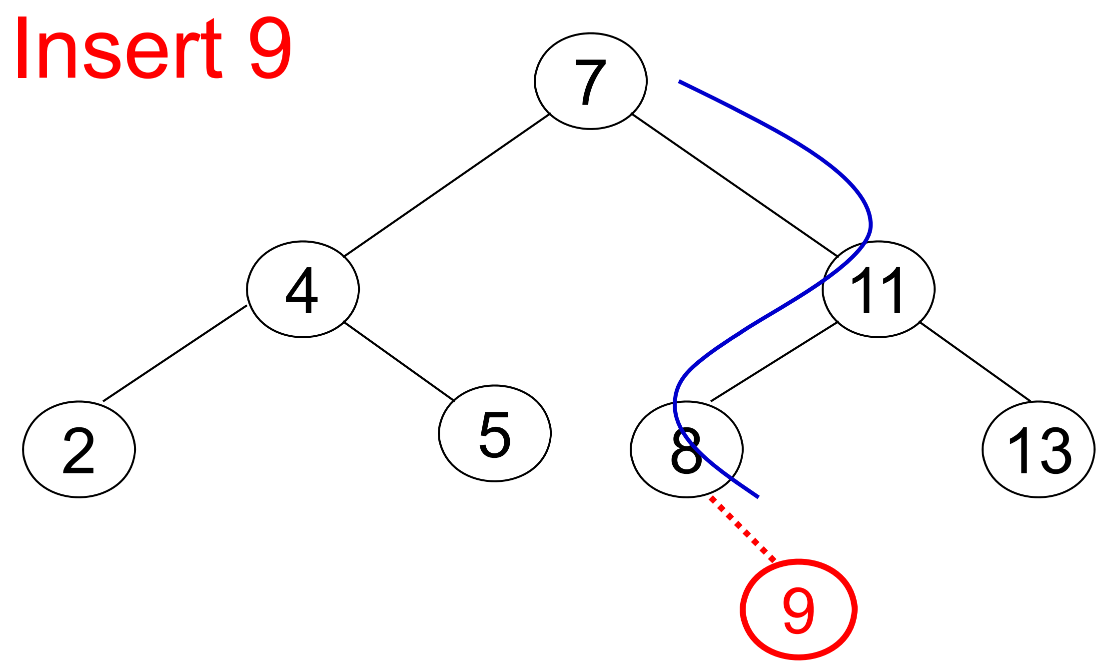
    if k < key(v) then v=left(v) else v= right(v)

return v

Complexity : O(h),   h=height of the tree

Insertion of a new key

- Search for the key, and at the end when it is not found, put it where it should have been

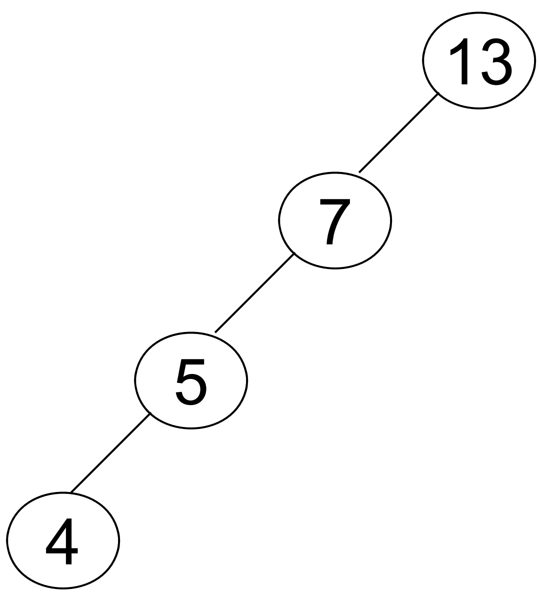


Shape of BST depends on Order of insertion

- Sorted order $\Rightarrow$ unbalanced BST

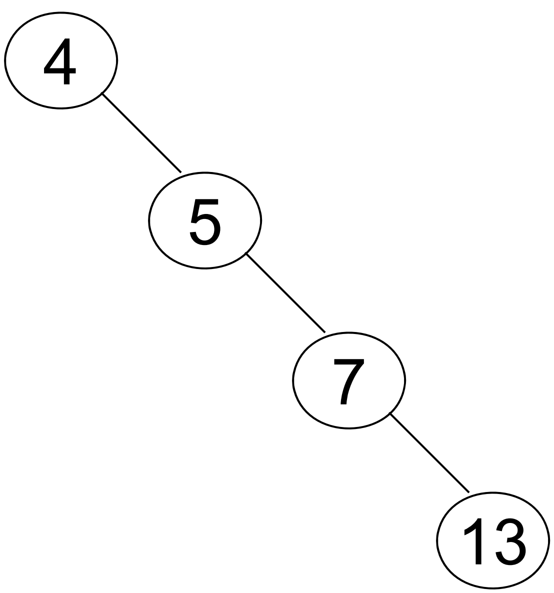
Decreasing order

13 7 5 4



Increasing order

4 5 7 13



Random insertion order $\Rightarrow$ ~Balanced BST

- Correspondence with Randomized-Quicksort
- R-Quicksort: Random pivot key partitions other keys, recursive calls on smaller and larger keys
- Binary Search Tree: Random first key goes at root, partitions other keys: smaller in left subtree, larger in right subtree
- Total BST insertion time = R-Quicksort time

  = Θ(nlogn) expected time (if no duplicate keys)

- BST $\leftrightarrow$ R-Quicksort recursion tree
- height of BST = depth of R-Quicksort recursion
- expected height = O(logn) (no duplicate keys)

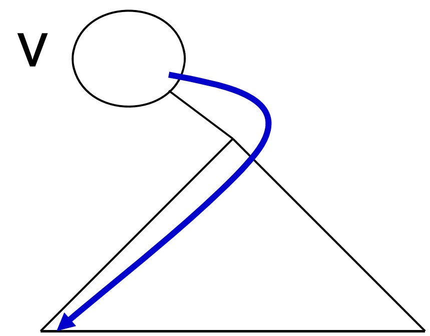
Successor, Predecessor

- successor(v) = node with least key > key(v)
- predecessor(v) = node with greatest key < key(v)

Successor(v)

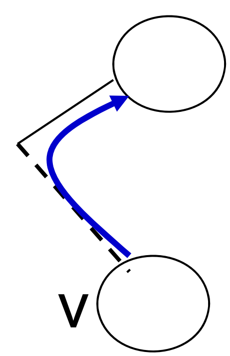
Case 1: v has a right child:

go right then all left



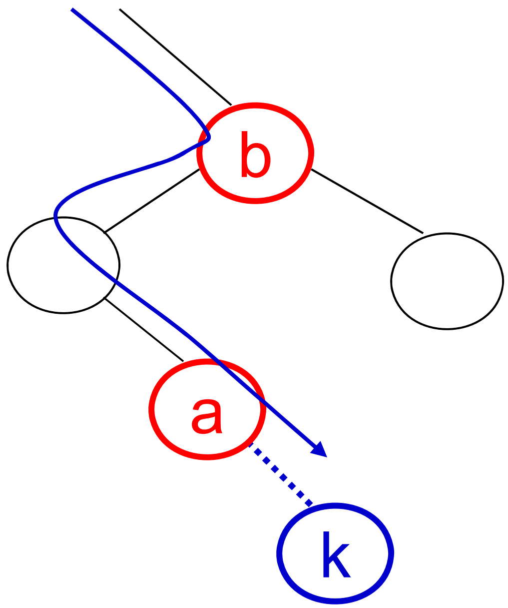
Case 2: v has no right child:

go up till first left child



Nearest matches of a missing key

- Given key k $\notin$S, find nearest keys a,b in S, a<k<b
- Search for key k in BST
- Nearest matches found on the path
- Case 1: k → missing right child



a = key of last node visited

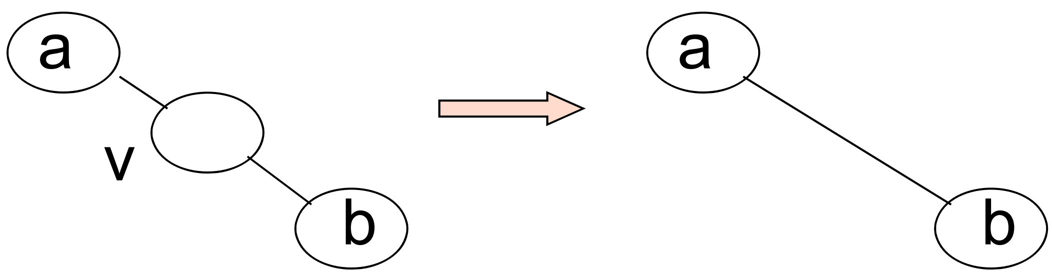
b = key of its successor

- Case 2: k → missing left child , symmetric

Deletion

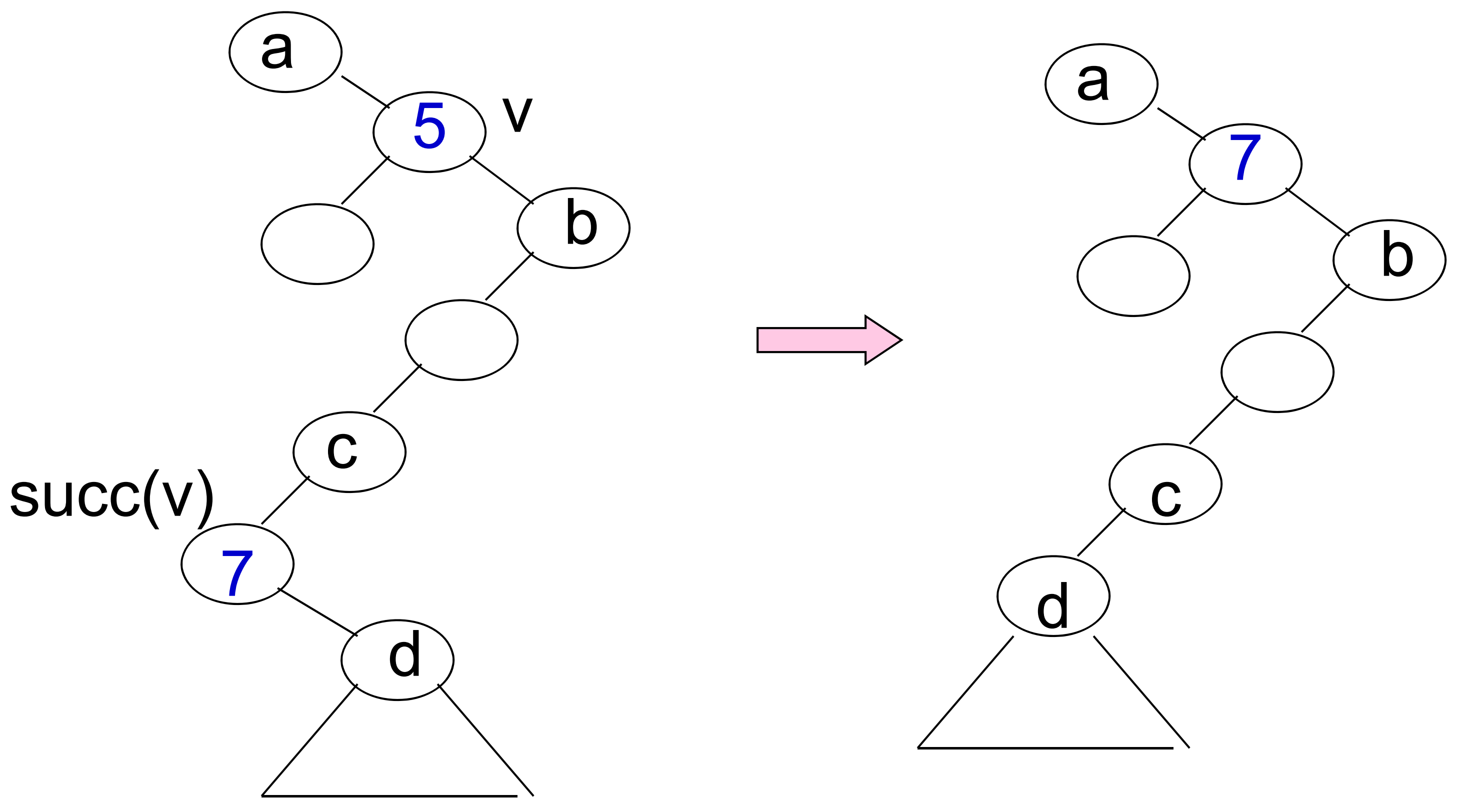
Delete(key(v))

- Case 1: v has no children → delete v
- Case 2: v has one child → shortcut v



Deletion ctd.

- Case 3: v has two children → replace v by its successor(v) (which has no left child ⇒Case 1 or 2) and update links



Balanced Trees

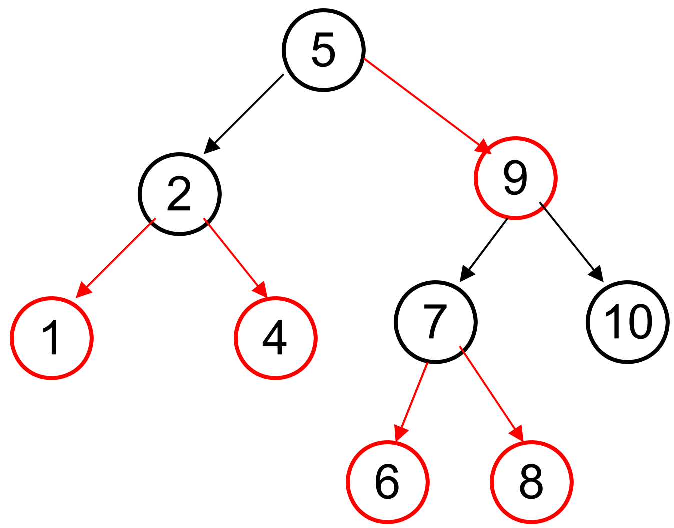
Balanced Trees

- BST trees after random insertions have O(logn) expected height, but
  - insertions may not be random
  - bound does not hold after mix of deletions, insertions
- Ensure O(logn) height by enforcing some form of balance in the tree
  - red-black trees
  - AVL trees
  - 2-3, 2-3-4 trees, B-trees
  - treaps (expected)
  - splay trees (amortized cost)
  - …..

Red-Black Trees

- Red-Black Tree: Binary Search Tree with two types of nodes, red and black
- Root is black
- Children, parent of a red node are black
- All root-to-’nil’ paths same number of black nodes

Example



Height of red-black trees

$$\text{Height} \leq 2\log(n+1)$$

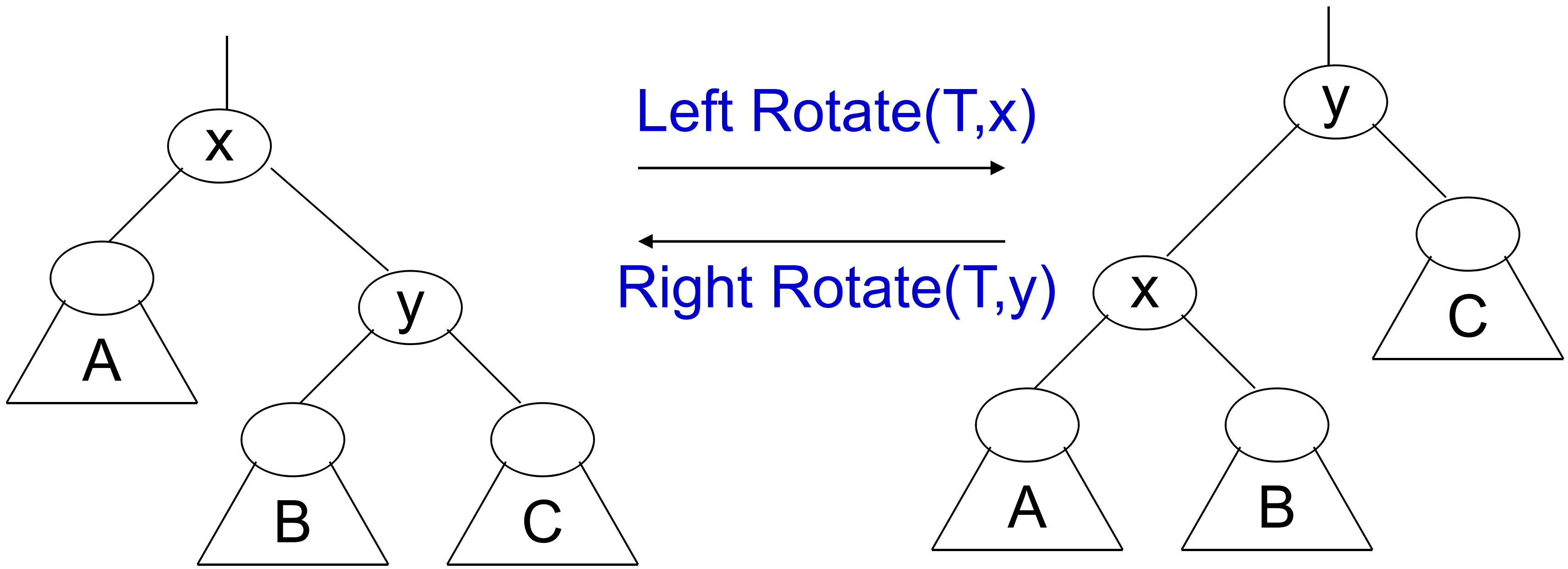
Define black height of a node v:
bh(v)= #black nodes in any path from v to nil

Claim: Subtree rooted at v has $\geq 2^{bh(v)}-1$ nodes
Proof: By induction on height of v.

$\Rightarrow$ bh(root) ≤ log(n+1)

Root-to-leaf paths do not have two red nodes in a row $\Rightarrow$
Height of tree ≤ 2 bh(root) ≤ 2 log(n+1)

Restructuring a Binary Search Tree: Rotations



- BST order property maintained

  keys(A) ≤ key(x) ≤ keys(B) ≤ key(y) ≤ keys(C)

- Can restructure BST, improve balance for BST,

  and/or fix coloring for Red Black tree

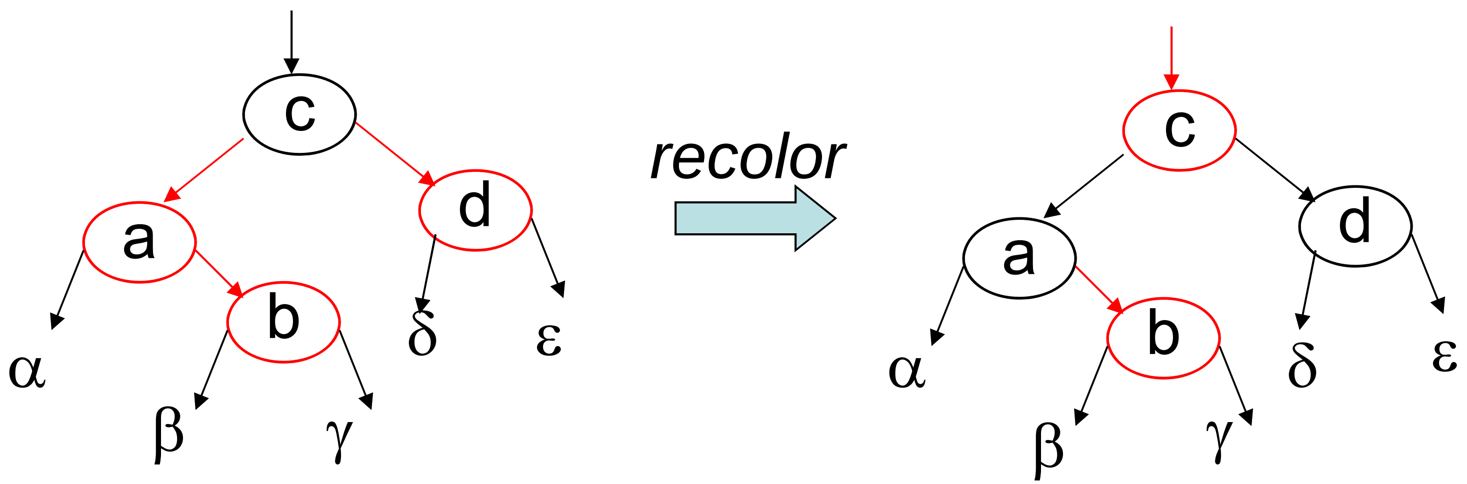
Operations on Red-black trees

- Search: Same as in ordinary BST

- Insertion: Key is inserted as in BST in new leaf colored red
  $\rightarrow$ may result in two red nodes in a row
  Possible violation of property 'no red-red' fixed by recolorings, rotations

- Deletion: Key is deleted as in BST
  Possible violation of red-black properties due to shortcut of removed node are fixed by recolorings, rotations – several cases, see book

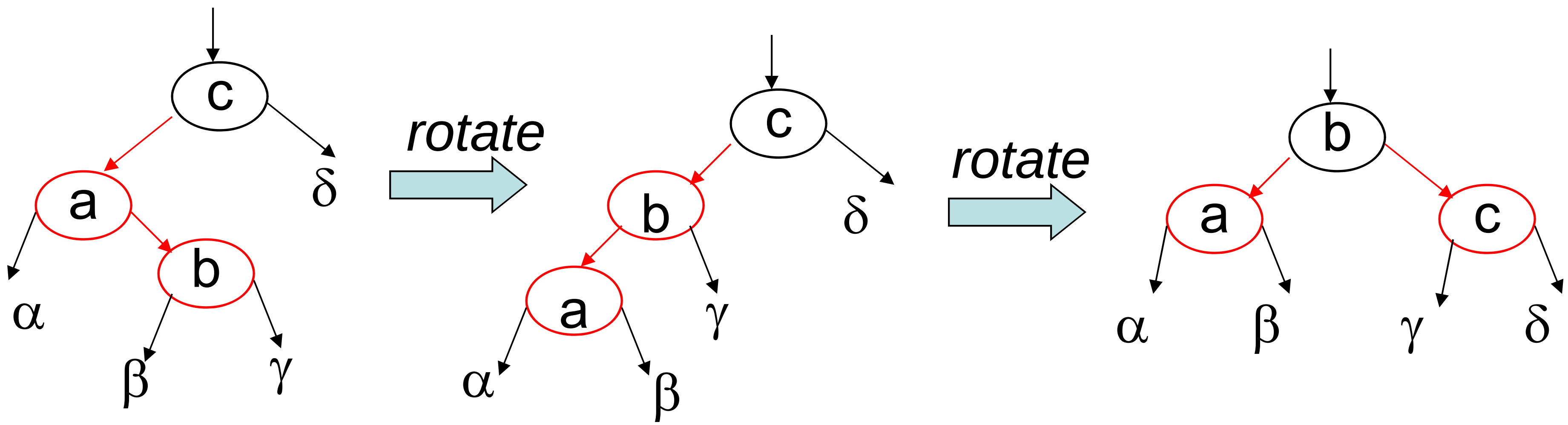
Red node with red child and red sibling



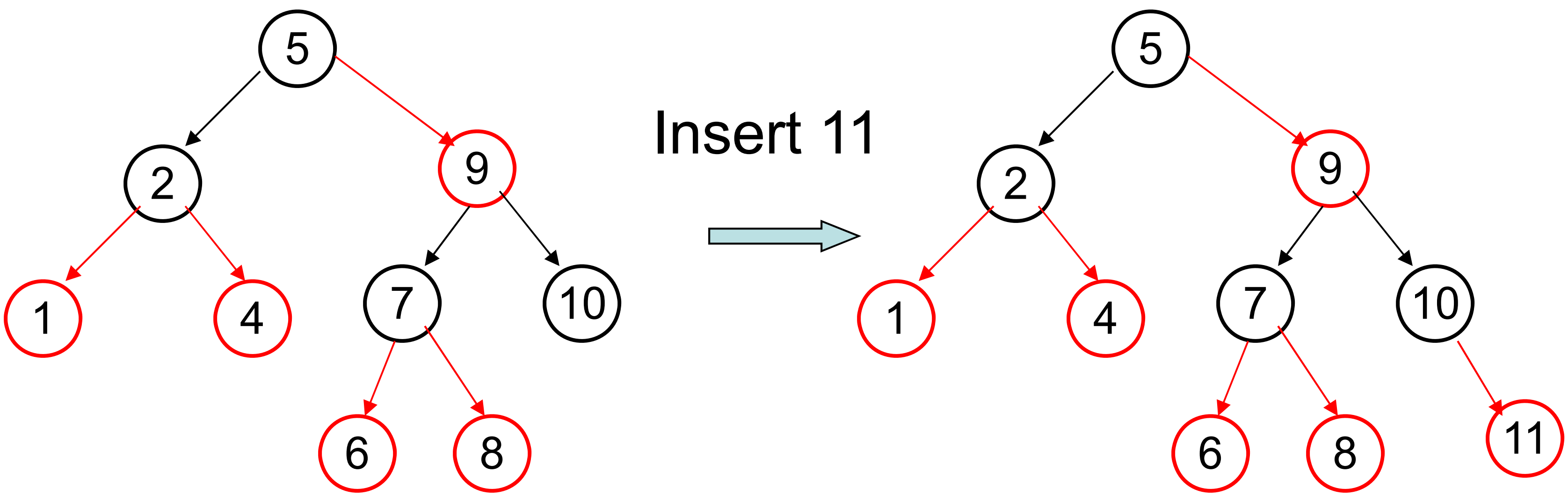
α,β,γ,δ pointers to black nodes or nil

- If c=root, then change its color to black, Done
- If parent(c) is black then Done,
- otherwise violation of two red nodes at c -> Continue fixing up the tree

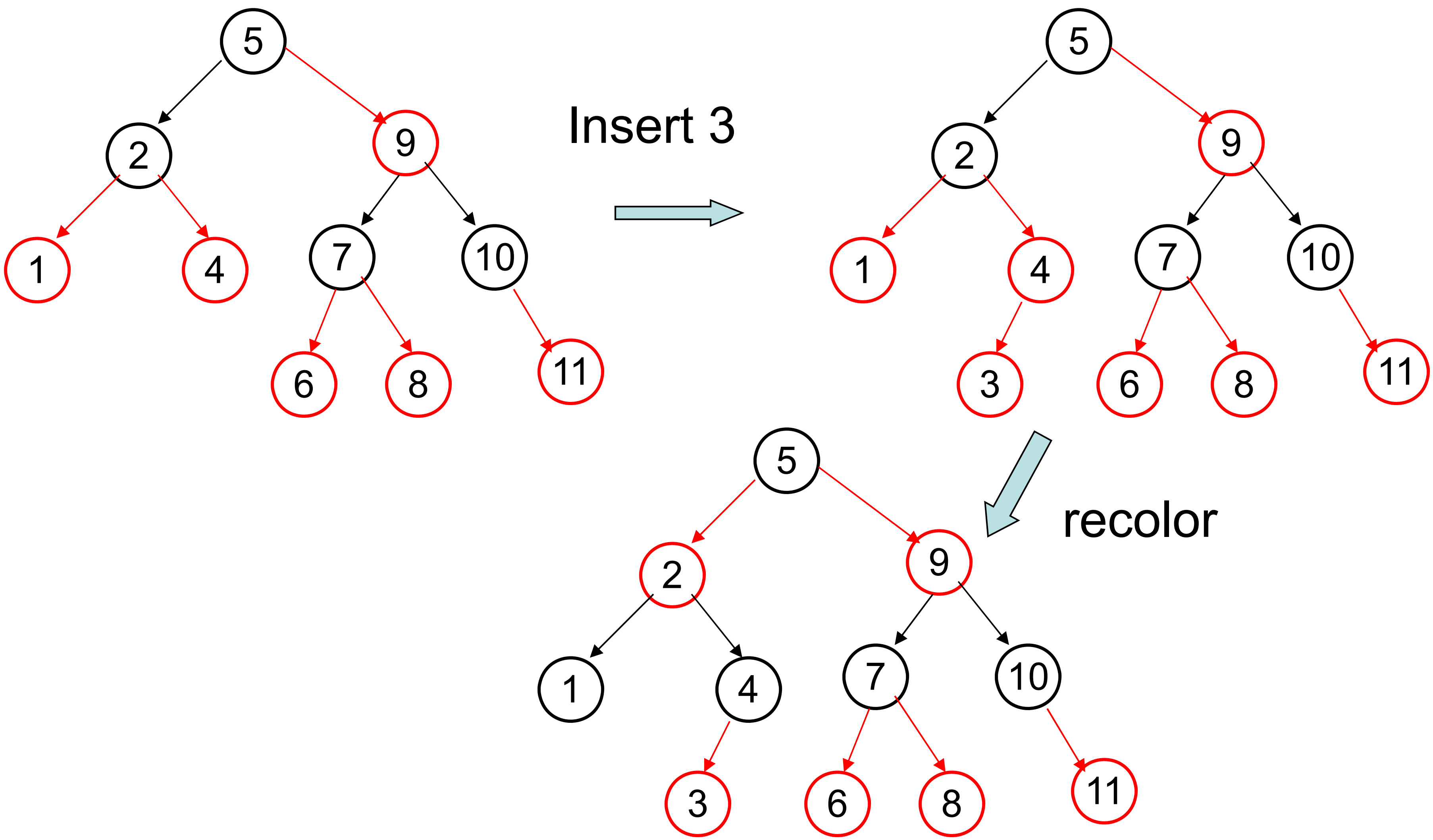
Red node with red child, and black (or no) sibling



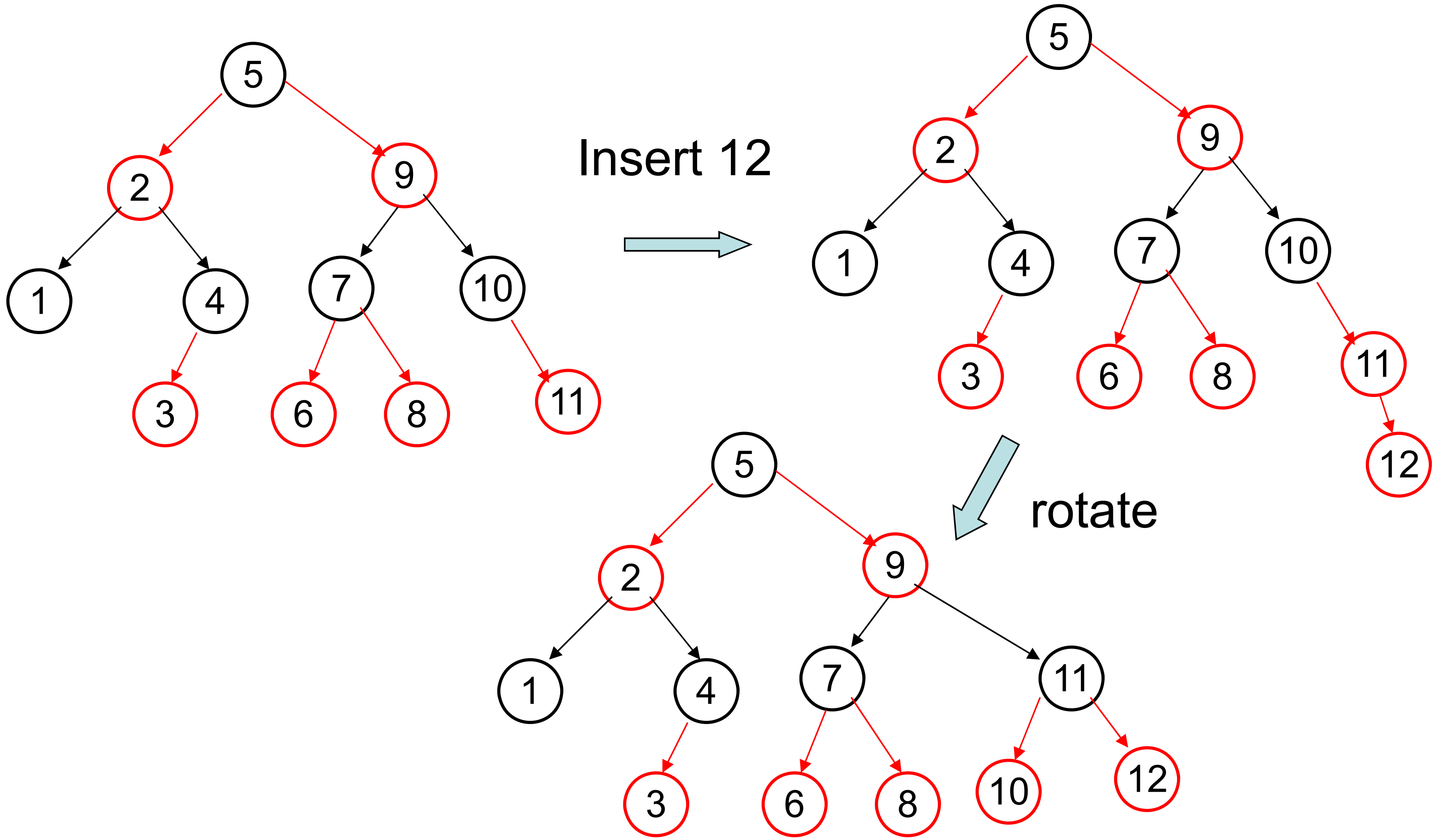
Example (red-black tree insertion)



Example (red-black tree insertion)



Example (red-black tree insertion)



Summary

- Balanced Trees (e.g. red-black trees,..) support the following operations in O(log*n*) worst-case time
- Insertion
- Deletion
- Search
- Max, Min
- Successor, Predecessor

Augmenting Data Structures

Augmenting data structures

- Can add auxiliary information to facilitate other queries, operations, or structuring of tree

- Example:

  - Selection, Order Statistics queries

    Select(i): Find the i-th smallest key

    Rank(x): What is the rank of the key at node x

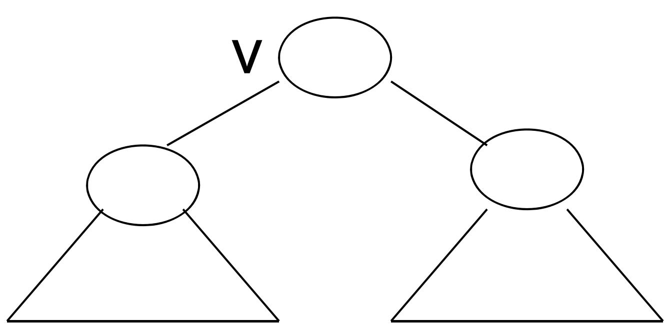
    or Rank(k): How many keys are ≤ k

Augmenting Search Trees (BST, red-black ..) for Order Statistics

- Add size(v): #nodes (keys) of subtree rooted at node v

Select(v,i): select the i-th smallest key in subtree of node v: Complexity O(height(v)).
- If balanced tree (e.g. red-black tree) then O(logn)

case: size(left(v)) = i-1 → return key(v)
size(left(v)) ≥ i → return Select(left(v),i)
size(left(v))< i-1 →return Select(right(v),i-size(left(v))-1)



Can use size() to answer also Rank(node x) and Rank(key k) queries

Maintaining the additional information

- When we change the data structure (eg. insertions, deletions) we have to maintain the auxiliary information

- Size(nodes) in BST can be easily updated during insertions, deletions at same complexity O(height)

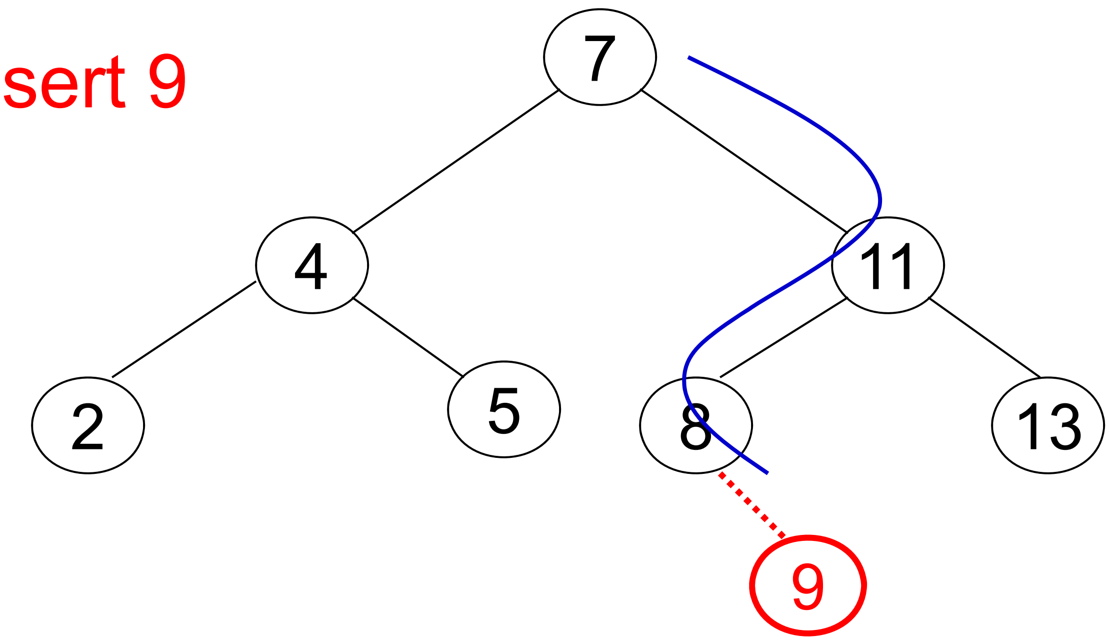
  - The only nodes affected during insertion or deletion are the nodes on the path from root to inserted/deleted node

- Ditto for red-black trees : O(logn)
  - Recolorings have no effect.
  - Rotations ok, can update the size.

Insertion of a new key in a BST

- Search for the key, and at the end when it is not found, put it where it should have been

Insert 9

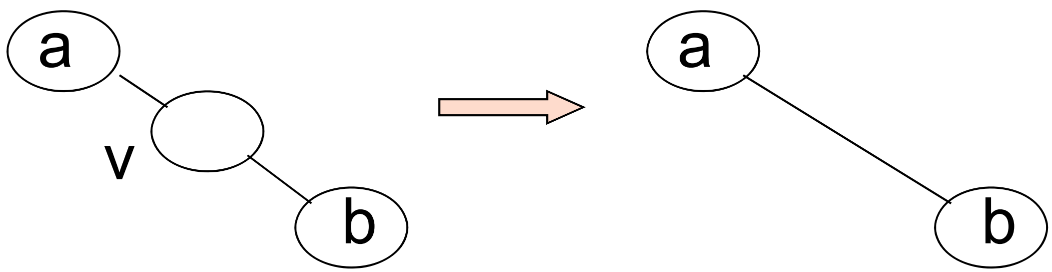


- Increment size along the path to new leaf

Deletion

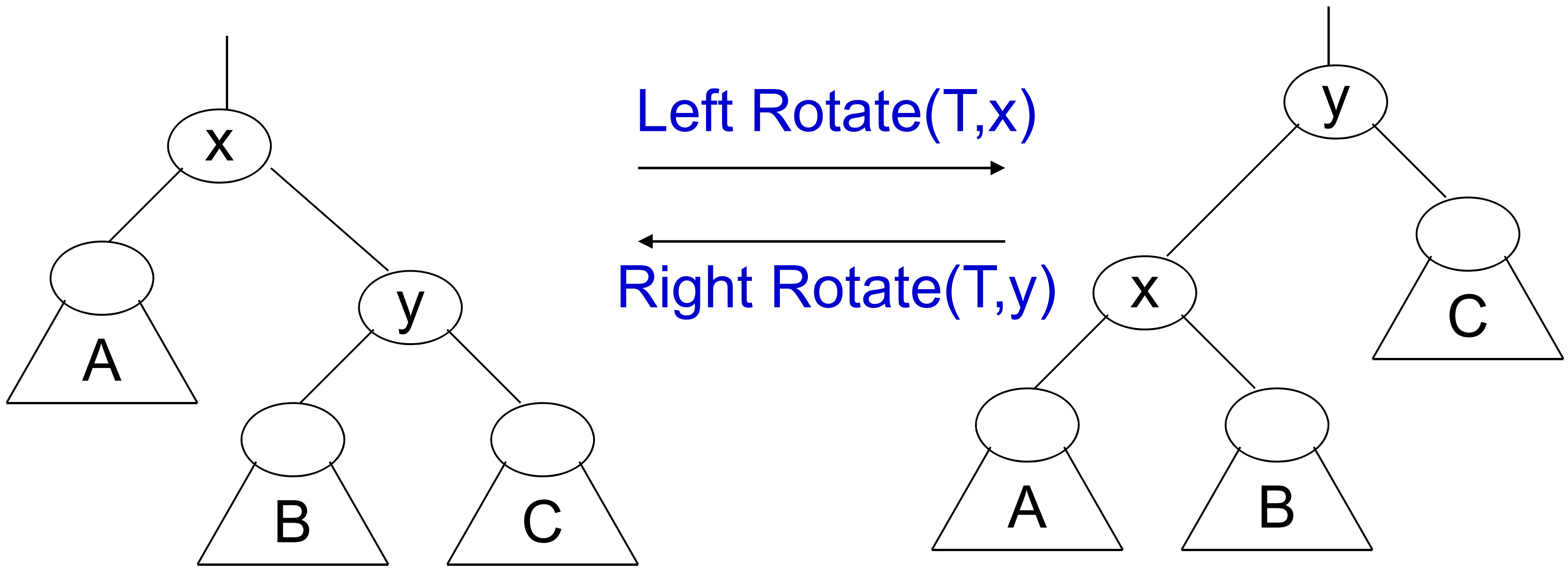
Delete(key(v))

- Case 1: v has no children → delete v
- Case 2: v has one child → shortcut v
- Case 3: v has both children: reduced to case 2



- Decrement size along the path to node a

Rotations



- Sizes of the roots of subtrees A, B, C don't change
- Update size(x) and size(y)

Red-Black Trees

-Same method for red-black trees: O(logn)

During insertion or deletion, changes along the path to the inserted/deleted node: recolorings, rotations

We can update the sizes of all the affected nodes in O(logn) time

- Similar for other types of balanced search trees

Augmenting Search Trees (BST, red-black trees etc)

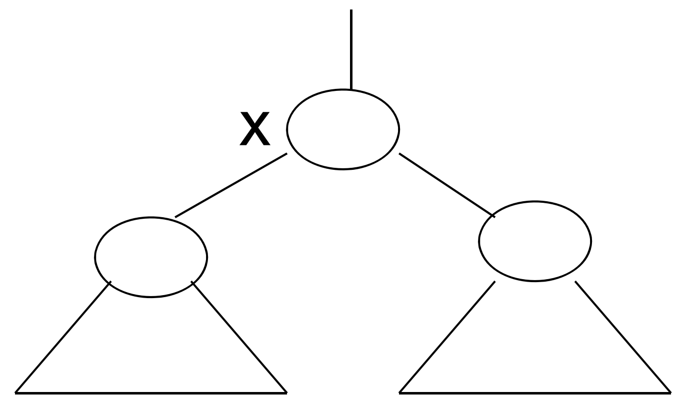
- General Lemma: Suppose augment search tree with field f and that f at node x can be computed from info(x), and value of f at children of x. Then can maintain the values of f at all nodes during insertion and deletion in time O(height) (=O(logn) for balanced trees)
- Proof Idea: Propagate the change in value of f at a node v up the tree during insertion and deletion. Can update f when we do rotations in red-black trees (recolor -> no change)

Examples:

f(x) =size (#nodes) of subtree T[x]

f(x)= sum of keys in subtree T[x]

Not: average of keys in subtree



Summary

- Balanced Trees (e.g. red-black trees and others) support the following operations in O(log*n*) worst-case time
- Insertion
- Deletion
- Search
- Max, Min
- Successor, Predecessor

- (+ others, possibly by augmentation, eg. Selection, rank)

Designing Data Structures

- Determine objects, operations (updates, queries)

- For comparison-based structures:
  - What will serve as the key?
  - What is the comparison operator and what is its cost?
  - What auxiliary information do we need?

Example: Interval trees

- Maintain set of intervals with insert, member, delete + other ops.
- Closed, open, half-open intervals: [a,b], (a,b), [a,b), (a,b]
- a = low (left) endpoint, b = high (right) endpoint
- Assume for simplicity closed intervals

- Basic Search data structure: search, insert, delete
- Can use a comparison-based structure: a balanced search tree (red-black, 2-3-4, AVL tree …) for O(logn) worst case
- What is the key and comparison operator?

Relations between intervals

- I1 left-of I2, I1 right-of-I2 (I1, I2 disjoint)
- I1 overlaps with I2 (I1, I2 intersect: they just overlap or one contains the other)

and the symmetric

Interval Search data structure

- Define Comparison operator (Ordering) among intervals s.t.
  1. nonreflexive : I not < I
  2. total: for all distinct intervals I1, I2 either I1<I2 or I2 <I1 (but not both)
  3. transitive: I1 <I2 and I2 < I3 $\Rightarrow$ I1 < I3

- 'left-of' not appropriate: not total
- order by low endpoint (in book) not total either: cannot do member and delete by value (can delete by handle)

  - Would be ok if we don't need member and delete

- Order by low endpoint, then (break ties) by high endpoint (lexicographic order for pairs) : ok
- Use a red-black tree (could also use 2-3-4 tree etc)

Example query: Longest interval

- Interval tree + auxiliary information:
- maxlength(v) = maximum-length interval in the subtree of v

- f(v)=maxlength(v) can be computed from the info(v) (which is the interval at v) and value of f at the children of v.

  $\Rightarrow$ can maintain the maxlength information with O(logn) time for insert and delete

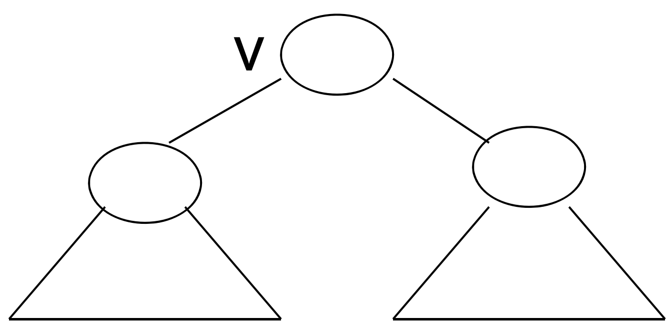
Alternative solution: choose as key=(length, low endpoint)

Then longest interval is just a max query.

Example: Overlap queries

- Given query interval I, determine if there is some interval in the set that overlaps it and return one if so.
- Special case: I =[a,a] (a point).

- Augment interval tree with max(v) = maximum high endpoint of an interval in the subtree T[v]
- max(v) can be computed from I(v) and max at children



Example: Overlap queries

OVER(v,I) [v a node of the tree, I a query interval]

if Int(v) overlaps I then return v else

if left(v) $\neq$nil and max(left(v)) $\geq$low(I) then OVER(left(v),I) else

if right(v) $\neq$nil then OVER(right(v,I)) else return NO

Time: O(logn)

Correctness proof:

- If Int(v) overlaps I then ok, so suppose not, i.e. Int(v) either left or right of I

- if max(left(v)) $\geq$low(I) and Int(v) left of I then there is an intersecting interval in the left subtree because all intervals J in the left subtree have low(J) $\leq$ low(Int(v)) $<$ low(I) and at least one of them has high(J) = max(left(v)) $\geq$low(I) .

Thus the algorithm correctly recurses in the left subtree to find an intersecting interval.

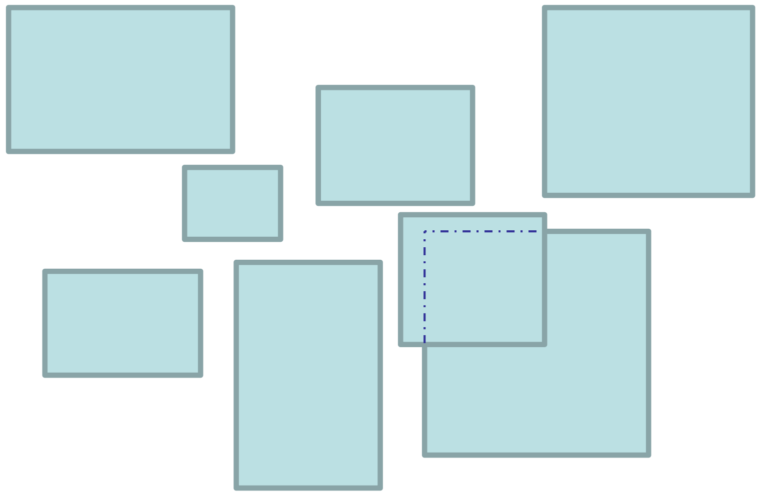
- if max(left(v)) $\geq$low(I) and Int(v) right of I, then all intervals in right subtree are right of I (no overlap) $\Rightarrow$ if there is an overlapping interval it must be in the left subtree

Correctness proof ctd.

- if max(left(v)) < low(I) then all intervals in left subtree are left of I $\Rightarrow$ if there is an overlapping interval it must be in the right subtree

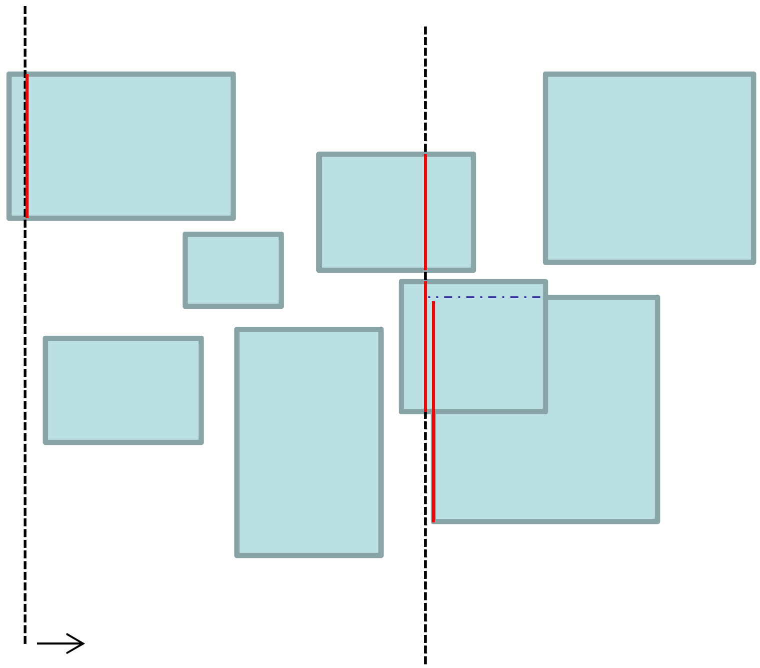
Example: Rectangle overlap

- Given a set of rectangles with sides parallel to the x and y axis, determine if there are any two overlapping rectangles.



Algorithm

- Sort the rectangles by the x coordinate of their left side.
- Sweep a vertical line from left to right, maintaining in an interval tree data structure the set of intervals in which the line intersects the rectangles
- Answer an overlap query for every new rectangle



Time: O(nlogn)