JOHNS HOPKINS UNIVERSITY | Engineering for Professionals

Review of Data Structures

Foundations of Algorithms
Guven

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Reading Assignment

- Cormen, Chapter 10, 11, 12, 13

Outline

- Dynamic Sets
- Array, Stack, Queue
- Linked List, Trees
- Hash
- Binary Search Trees
- Red Black Trees

Dynamic Sets

- Operations on an ordered set S (e.g. sequence)
  - `Search(S,k)`

    Finds and returns a pointer x to an element such that x.key=k
  - `Insert(S,x)`

    Inserts the element pointed by x to S
  - `Delete(S,x)`

    Removes the element pointed by x from S
  - `Minimum(S)`
  - `Maximum(S)`
  - `Successor(S,x)`

    Returns the pointer x' to the next element (from x) in S
  - `Predessor(S,x)`

    Returns the pointer x' to the previous element (from x) in S

Array

- Contiguous and linear memory buffer
  - MEM-STORE, MEM-LOAD machine level instructions
- Simplest possible data storage
- Pre-allocated in the program
  - Otherwise the compiler/interpreter will dynamically allocate

```
def preallocate():
  array_example = ARRAY_SIZE*[None]
  array_example[42] = 42; array_example[42] = '42'
def allocate():
  array_example = []  # Have to use append
  array_example.append(42)  # array_example += [42]
```

Stack

- A Last-In-First-Out linear data structure
  - Preallocated storage, **top** variable keeps the current position

```
S = STACK_SIZE*[None]; top=0
def push(S,x):
  if top < STACK_SIZE:
    S[top] = x; top += 1
def pop(S):
  if top > 0:
    top -= 1; return S[top+1]
def empty(S):
  if top == 0:
    return True
  return False
```

Queue

- First-In-First-Out linear data structure
  - Preallocated storage, **head** and **tail** keeps the positions

```
Q = QUEUE_SIZE*[None]; head=0; tail=0
def enqueue(Q,x):  # implementation is circular
  if abs(head-tail) < QUEUE_SIZE:
    Q[tail]=x; tail += 1
    if tail == QUEUE_SIZE: tail=0
def dequeue(Q):
  if head != tail:
    x=Q[head]; head += 1
    if head == QUEUE_SIZE: head=0
    return x
  return None
```

Linked List

- A linear data structure
  - Dynamically allocated, **next** and **prev** variables keep track

```
class Node:
  def __init__(self, x):
    self.x=x; self.prev=None; self.next=None
  def setnext(self, next):
    self.next=next
  def getnext(self):
    return self.next
  def getx(self):
    return self.x
```

Linked List

```
def insert(_root, x):
  node=_root
  while node.getnext() != None:
    node=node.getnext()
  node.setnext(x)

def search(_root, x):
  node=_root
  while node.getx() != x:
    if node.getnext() != None:
      node=node.getnext()
    else:
      return None
  return node
```

Trees

- A non-linear data structure
  - Dynamically allocated, **left** and **right** variables keep track

```
class TNode:
  def __init__(self, x):
    self.x=x; self.left=None; self.right=None
  def setleft(self, left):
    self.left=left
  def setright(self, right):
    self.right=right
```

Trees

```
@classmethod
def tsearch(node, x):
  if node == None or node.x == x:
    return node
  if node.x < x:
    return tsearch(node.left,x)
  else:
    return tsearch(node.right,x)
```

Hash

- Arrays have the **index** to address the data
  - i.e. like a coordinate
- Use key instead of an index
  - key value pairs, a dictionary
- Simplest implementation is list of indices
  - In this scenario indices are not sequence of integers
  - Finding the correct index: O(n)
    - Indices may not be sortable
  - Compare to finding the correct array index: O(1)

Hash

```
H={}
def search(H,k):
  return H[k]
def insert(H,k,x):
  H[k]=x
def delete(H,k):
  del H[k]
```

- Also called direct address tables
  - i.e. dictionary
- Does not support multiple values to the same key
- Allows **collision**
  - i.e. if hash values are same for multiple keys
- Hashing and collision internal to Python

Hash Tables

- Hash function h maps keys to memory locations
- Given a hash table T[0:m-1]
  - e.g. T is an array
- $h : U \rightarrow \{0,1,\ldots,m-1\}$
  - Universe of keys
- Collision is handled with **chaining**
  - *i.e.* linked lists

Analysis of Hash Tables

- n data elements
- m memory locations in T, generally n>m
- Define load factor $\alpha = n/m$
- Theorem: Unsuccessful searches take $\Theta(1+\alpha)$
  Proof: Simple uniform hashing assumption $\Longrightarrow$ n/m elements are needed to be searched after h(k) which is O(1)

Hash Functions

- Division method
  - h(k) = k modulo m

```
def h(k):   # M constant
  return k%M
```

- Multiplication method
  - $h(k) = \lfloor m(kA-\lfloor kA \rfloor) \rfloor$, where $0<A<1$
- Universal hashing
  - Select randomized hash functions

Binary Search Trees

- Each node contains the object (i.e. data)
- For each node

```
node.left.key ≤ node.key
node.right.key ≥ node.key
```

- In a way, the tree is kept sorted
- **Theorem:** In-order walk of the tree takes $\Theta(n)$ time

Binary Search Trees

```
def tsearch(node, x):
  if node == None or node.x == x:
    return x
  if node.x < x:
    return tsearch(node.left,x)
  else:
    return tsearch(node.right,x)

def tsearch_iterative(node, x):
  while node != None and node.x != x:
    if node.x < x:
      node = node.right
    else:
      node = node.left
    return x
```

Binary Search Trees

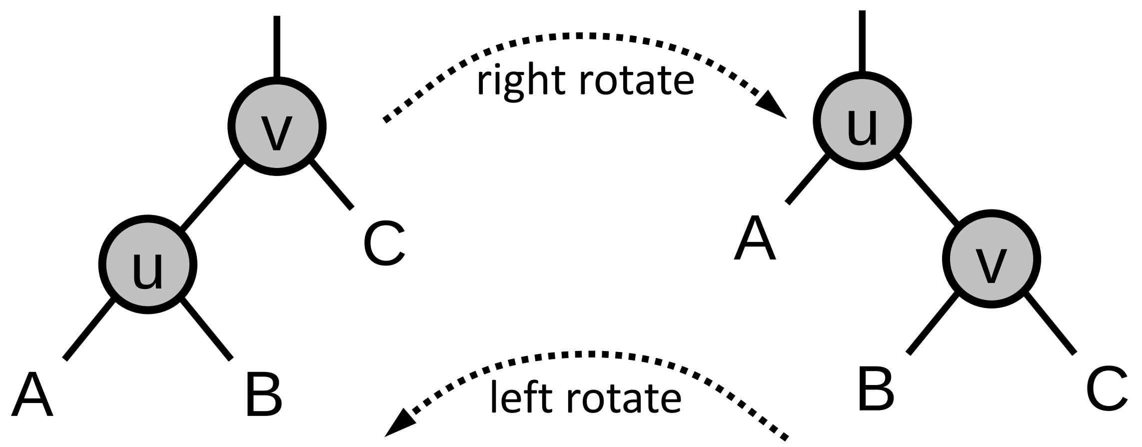
```
def insert(newnode):
  global root
  y = None; node = root
  while node != None:
    y = node
    if newnode.x < node.x:
      node = node.left
    else:
      node = node.right
  if y == None:
    root = newnode
  elif newnode.x < y.x:
    y.left = newnode
  else:
    y.right = newnode
```

AVL Trees

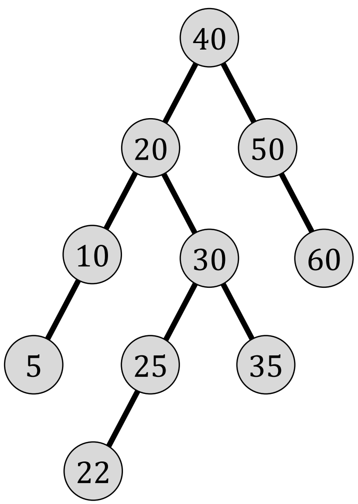
- Self balancing binary search trees - AVL trees (Adelson-Velsky and Landis, 1962)
- Property - Heights of the two child subtrees of any node differ by at most one
- Rebalancing is conducted to retain property
- A variation of AVL is Red Black trees

Rotations

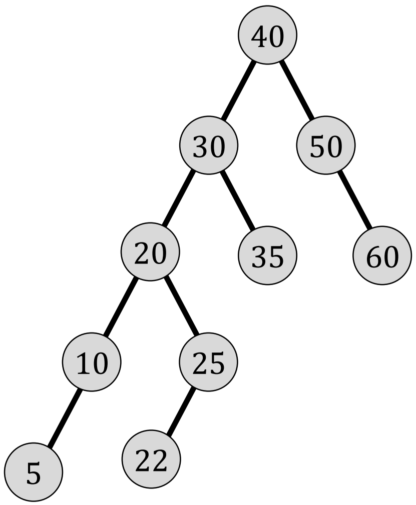
- After insert and delete
  - Keep the tree balanced
  - Keep the tree's {AVL,RB} properties valid



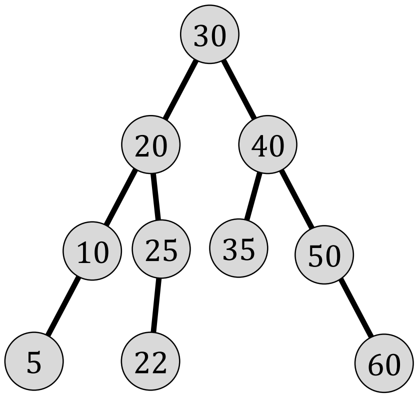
Rotation Example



(1) original



(2) a left rotation



(3) after the right rotation

Rotation

```
def rotate_left(x):
  global root
  y = x.right  # set y
  x.right = y.left  # turn y.left subT into x.right subT
  if y.left != None:
    y.left.p = x
  y.p = x.p  # link x's parent to y
  if x.p == None:
    root = y
  elif x == x.p.left:
    x.p.left = y
  else:
    x.p.right = y
  y.left = x  # put x on y's left
  x.p = y
```

Red Black Trees

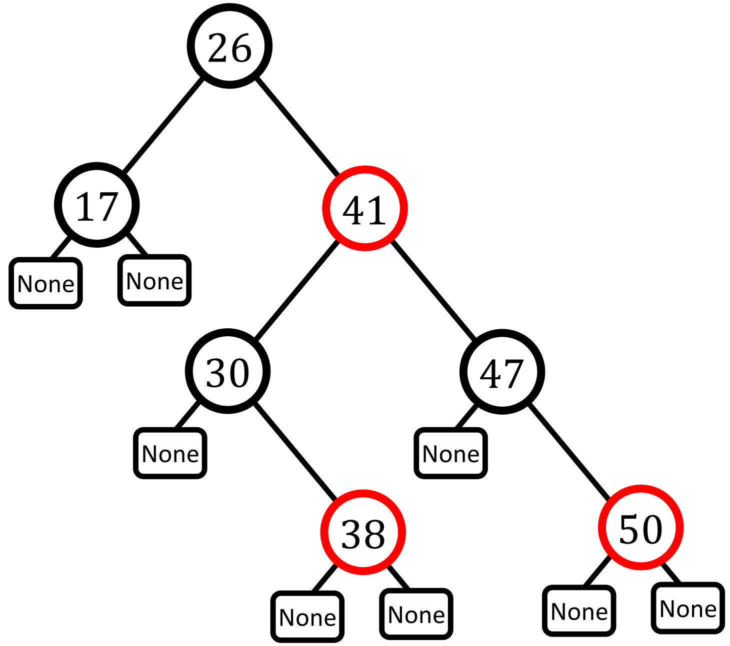
- Extension of AVL binary search trees
  - Each node has a color {red,black}
  - Each node has a parent pointer p
  - Every node is either red or black
  - The root is black
  - Every leaf that is None is black
  - If a node is red then both its children are black
  - For each node all paths to descendant leaves contain the same number of black nodes
  - The goal is to "balance" the tree

Red Black Trees

- **Lemma:** An AVL tree with n internal nodes has height at most $c\log(n+2)+b$

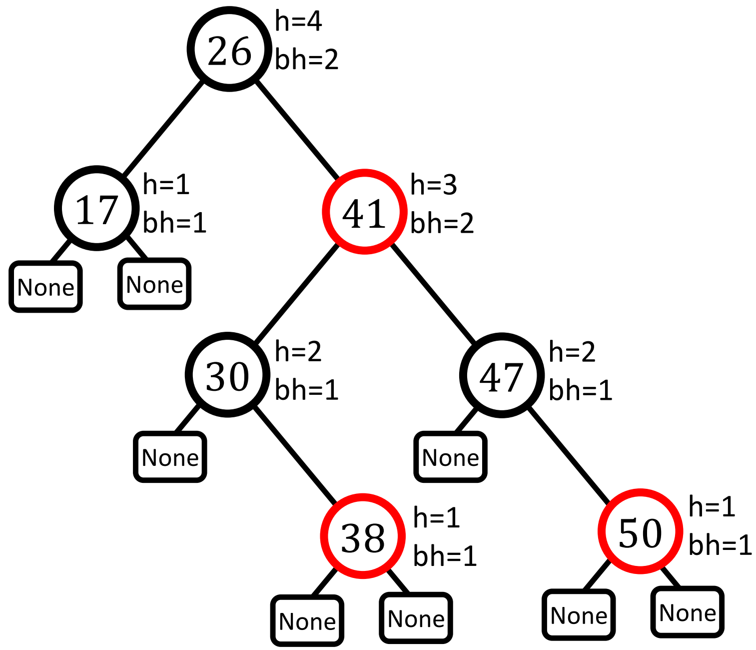
- **Lemma:** A red-black tree with n internal nodes has height at most $2\log(n+1)$

Example RB Tree



- NoneNodes.color = black

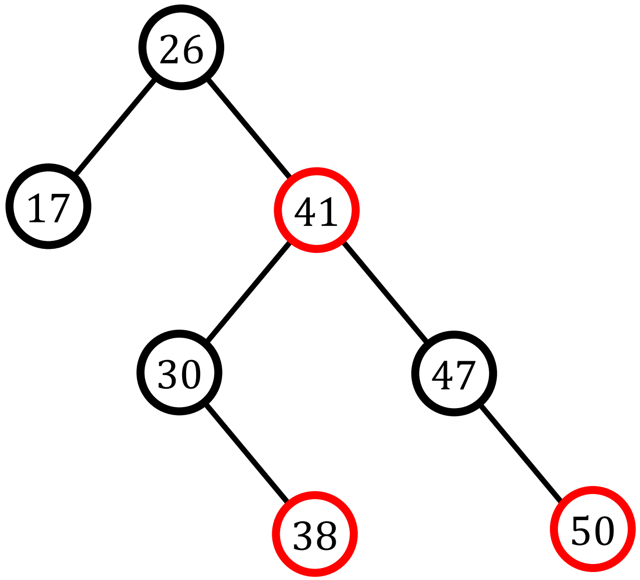
Example RB Tree



- node.h = the number of edges in the longest path to a leaf
- node.bh = the number of black nodes (with NoneNodes) on the path from node_x to a leaf, not counting node_x

RB Tree Insertion & Deletion

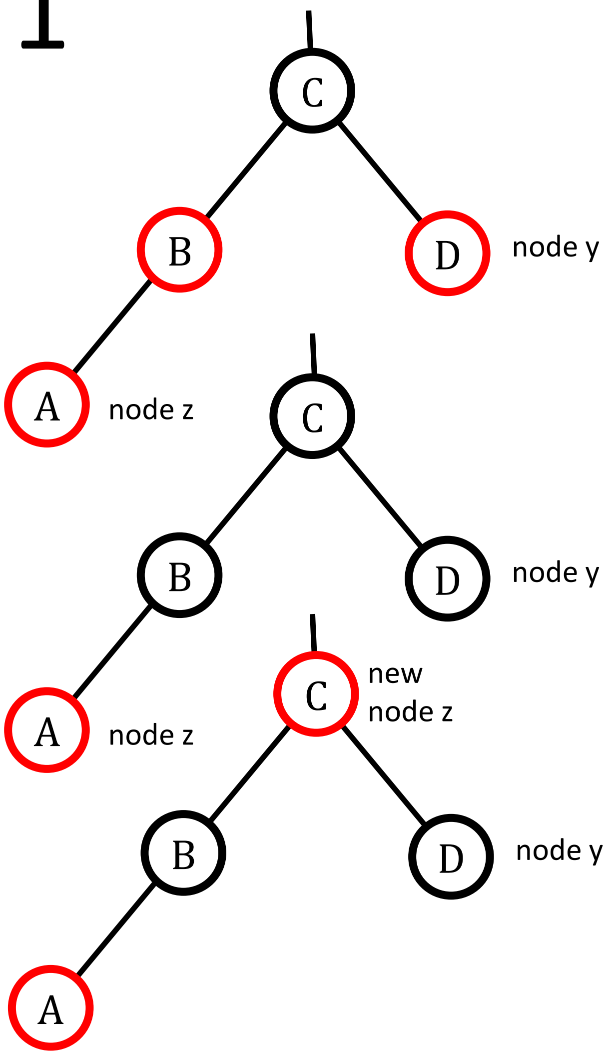
- Insert red 35
  - fix: both children must be black
- Insert black 14
  - fix: paths to its leaves has same # black nodes
- Delete root
  - fix: colors have to be readjusted
- Delete a black node
  - fix: black heights
  - fix: two consecutive reds
- Insert like a BST then Fixup the tree to restore properties



RB Fixup Case 1

- Insert z with default red
- Case 1: z.uncle y is red, z can be either left or right child

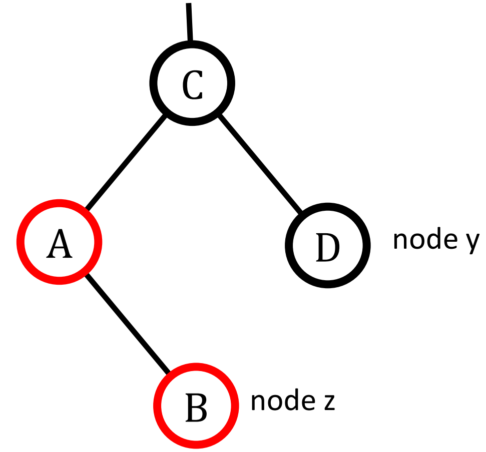
```
# z.p.p must be black
z.p.color = black
y.color = black
z.p.p.color = red
z = z.p.p
# push red violation up the tree
```

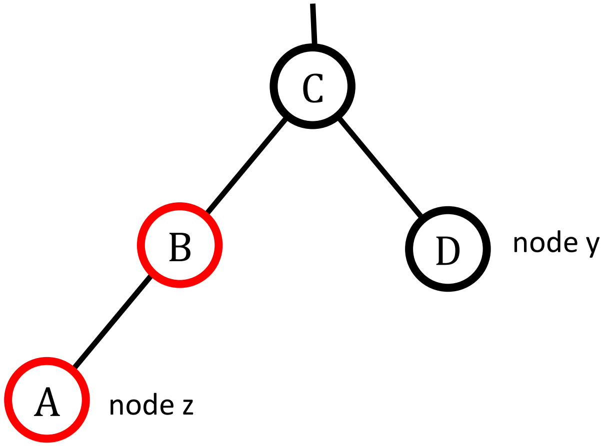


RB Fixup Case 2

- Case 2: z.uncle y is black and z is right child

```
z = z.p
rotate_left(T, z)
# z is left child now
# both z and z.p are red
# continue with case 3
```

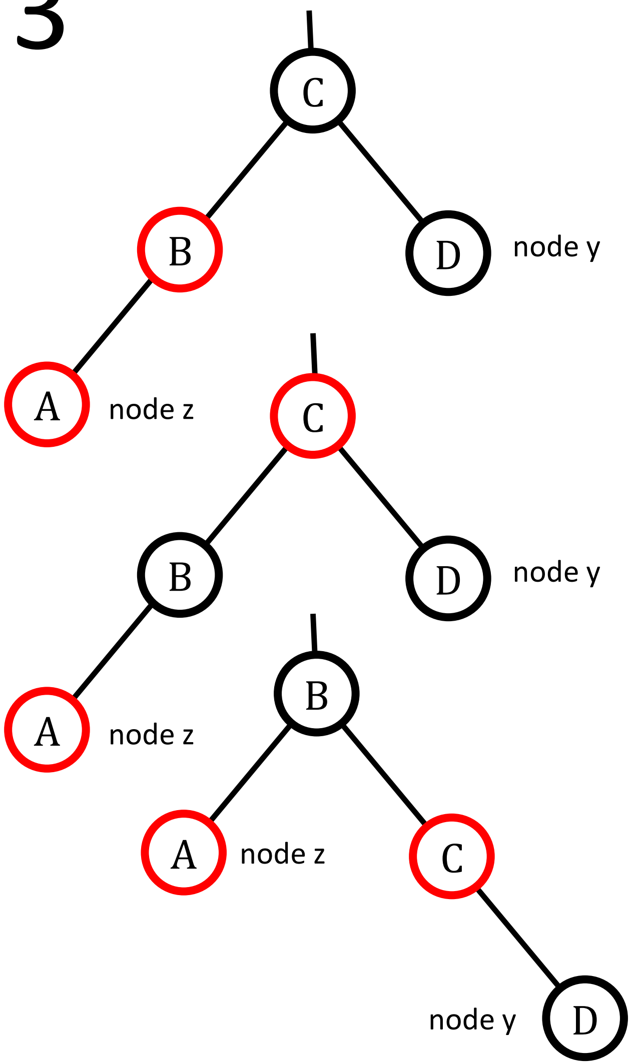




RB Fixup Case 3

- Case 3: z.uncle y is black and z is left child

```
z.p.color = black
z.p.p.color = red
rotate_right(T, z.p.p)
# no two consecutive reds
# z.p is black now
```



RB Insert-Fixup

```
while z.p.color == red:
  if z.p == z.p.p.left:
    y = z.p.p.right
    if y.color == red:
      case1()
    else:
      if z == z.p.right:
        case2() - rotate_left()
      case3() - rotate_right()
  else:
    # similar to z.p == z.p.p.right as above
    # rotations reversed
T.root.color = black  # red reached to root
```

RB Insert-Fixup

- Insert: O(logn)
- Fixup: O(logn)
  - while loop executes only with case 1 and
  - and at most O(logn) times
  - Total insert: O(logn)

RB Deletion

- Case 1: z's sibling is red
- Case 2: z's sibling w is black and both w.left and w.right are black
- Case 3: z's sibling w is black, w.left is red and w.right is black
- Case 4: z's sibling w is black and w.right is red
- Complexity O(logn)