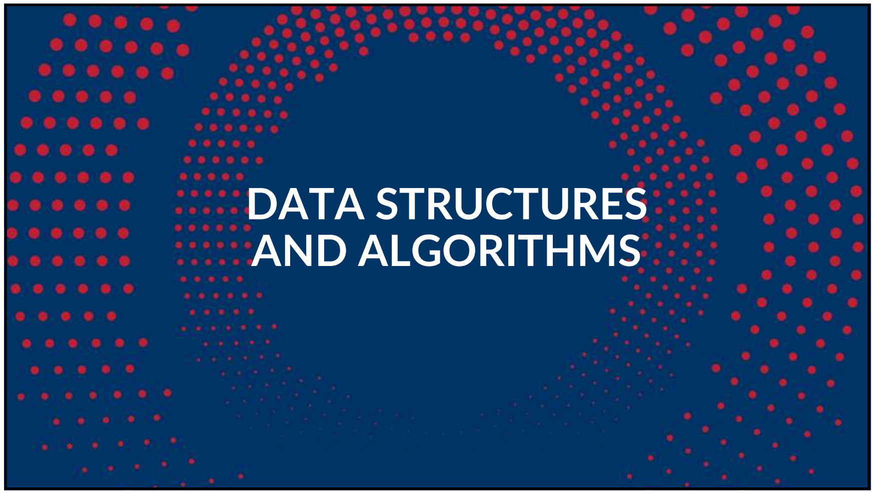
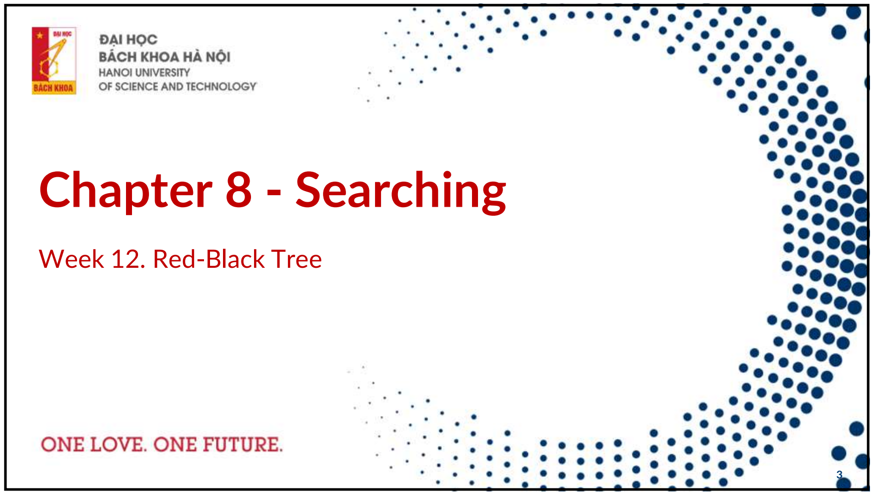
HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.



DATA STRUCTURES AND ALGORITHMS

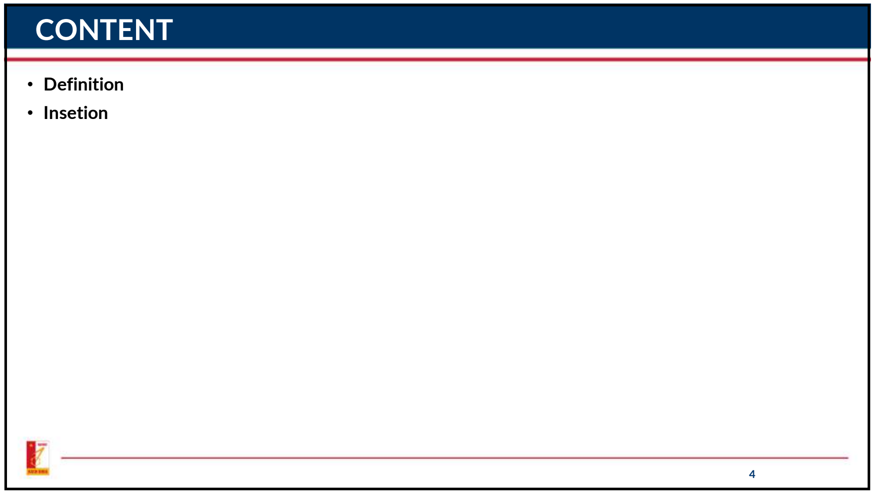


ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Chapter 8 - Searching

Week 12. Red-Black Tree

ONE LOVE. ONE FUTURE.

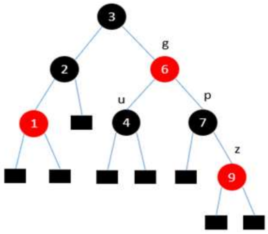


CONTENT

- Definition
- Insetion

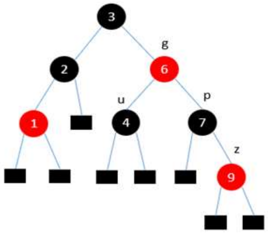
Definition

- Red-Black tree is a binary search tree with properties:
  - Each node has a color (red or black)
  - Color of root is black
  - Leaf (or NULL node) has color black (is presented by a rectangle)
  - A red node has 2 children with color black
  - Paths from a node to leaf nodes have the same number of black nodes



Definition

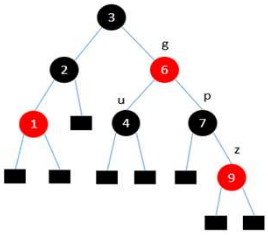
- Red-Black (RB) tree is a binary search tree with properties:
  - (1) Each node has a color (red or black)
  - (2) Color of root is black
  - (3) Leaf (or NULL node) has color black (is presented by a rectangle)
  - (4) A red node has 2 children with color black
  - (5) Paths from a node to leaf nodes have the same number of black nodes
- Notation: *bh*(*x*): number of black nodes (except *x*) on the path from *x* to a leaf
- Lemma 1. A RB tree contains at least $2^{bh(x)}$-1 internal nodes
- Lemma 2. Height of a RB tree containing *n* nodes is at most 2log(*n*+1)



Definition

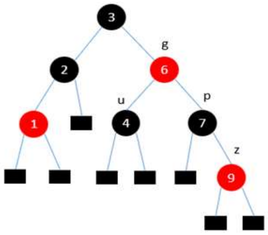
- Typical data structure of a node on a RB tree

```
Node {
   key;  // key of the node
   color; // color of the node
   p; // pointer to the parent
   left;  // pointer to the left-child
   right: // pointer to the right-child
}
```



Insertion

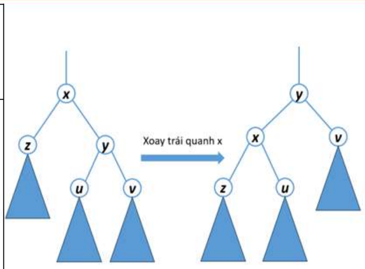
- When inserting a new node (node z) in the RB tree T
  - Insert node z into T as in binary search tree
  - Assign red color to this node z
  - If the RB property is not satisfied, then we perform rotations and change the color of some nodes to recover the RB property
- Notation
  - p: the parent node of z
  - u: sibling node of p (uncle node of z)
  - g: parent of p (grand parent of z)



Insertion: Left Rotation

```
Algorithm leftRotate(r, x)
Input: pointer r to the root of the RB tree T, pointer x to
some node of T
Output: perform left rotation on x, return the pointer to the
root of the resulting tree
```

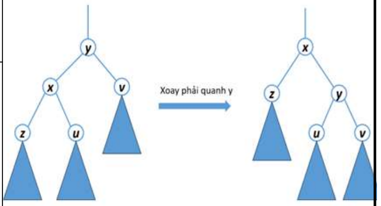
```
y = x.right;
x.right = y.left;
if y.left != NULL then {
      y.left.p = x;
}
y.p = x.p;
if x.p = NULL then {
      r = y;
} else if x = x.p.left then {
      x.p.left = y;
} else {
      x.p.right = y;
}
y.left = x;
x.p = y;
return r;
```



Insertion: Right Rotation

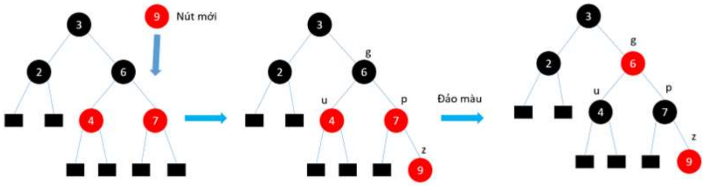
```
Thuật toán rightRotate(r, y)
Input: pointer r to the root of the RB tree T, pointer y to
some node of T
Output: perform right rotation on y, return the pointer to
the root of the resulting tree
```

```
x = y.left;
y.left = x.right;
If x.right != NULL {
     x.right.p = y;
}
x.p = y.p;
if y.p == NULL {
     r = x;
} else if y = y.p.left {
     y.p.left = x;
} else {
     y.p.right = x;
}
x.right = y;
y->p = x;
return r;
```



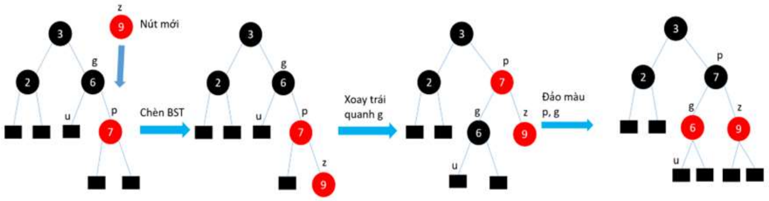
Fix RB property

- Case 1: T is empty, make z as the root, assign black color to z
- Case 2: (**node p – parent of z – with black color**) RB property is satisfied, do nothing
- Case 3: (**node p has color red**), property RB (4) is not satisfied. Node g has color black (as before insertion, T is a RB tree with RB property)
  - Case 3.1: (**node u (if exists) has color red**) → change the color of p and u from red to black and change the color of g by the color red, repeat this process with node g (Figure below).



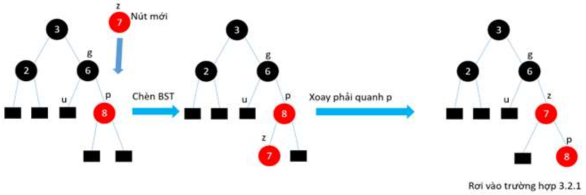
Fix RB property

- Case 3: (**node p has color red**), property RB (4) is not satisfied. Node g has color black (as before insertion, T is a RB tree with RB property)
  - Case 3.2: (node u (if exists) has color black) → perform a single-rotation or double-rotation depending on z is the left child or right child of p
    - Case 3.2.1 (**p is a right child of g and z is a right child of p**), perform left rotation on g and flip the color of p and g (Figure below)



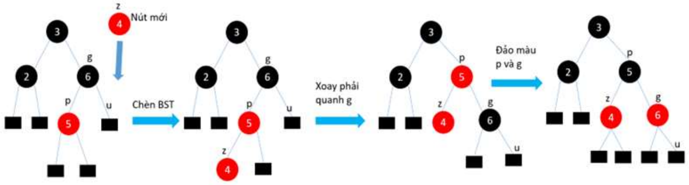
Fix RB property

- Case 3: (**node p has color red**), property RB (4) is not satisfied. Node g has color black (as before insertion, T is a RB tree with RB property)
    - Case 3.2: (node u (if exists) has color black) → perform a single-rotation or double-rotation depending on z is the left child or right child of p
        - Case 3.2.2 (**p is a right child of g and z is a left child of p**), perform right rotation on p (Figure below) return to the case 3.2.1 (process as in the case 3.2.1)



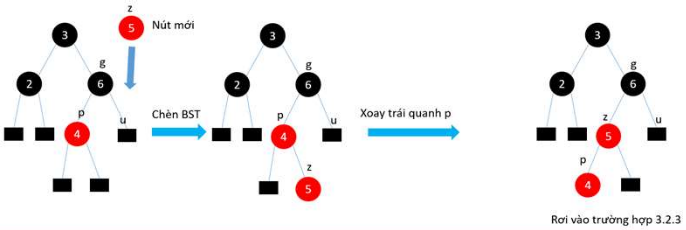
Fix RB property

- Case 3: (**node p has color red**), property RB (4) is not satisfied. Node g has color black (as before insertion, T is a RB tree with RB property)
    - Case 3.2: (node u (if exists) has color black) → perform a single-rotation or double-rotation depending on z is the left child or right child of p
        - Case 3.2.3 (**p is a left child of g and z is a left child of p**), perform right rotation on g and flip the color of p and g (Figure below)



Fix RB property

- Case 3: (**node p has color red**), property RB (4) is not satisfied. Node g has color black (as before insertion, T is a RB tree with RB property)
    - Case 3.2: (node u (if exists) has color black) → perform a single-rotation or double-rotation depending on z is the left child or right child of p
        - Case 3.2.4 (**p is a left child of g and z is a right child of p**), perform left rotation on p (Figure below) return to the case 3.2.3 (process as in the case 3.2.3)



Insertion: Fix RB property

```
Insert(r, k) {
  x = r;
  y = NULL;
  while x !=  NULL do {
    y = x;
    if  k < x.key then
      x = x->left;
    else
      x = x->right;
  }
  z = createNode(k,'RED');
  z.p = y;
  if y = NULL then {
    z.color = 'BLACK';
    return z;
  } else if z.key < y.key then
    y.left = z;
  else
    y.right = z;
  return insertFixUp(r,z);
}
```

```
insertFixUp(r,z) {
  while z.p != NULL and z.p.color = ‘RED' do {
    if z.p = z.p.p.left then {// z->p is a left-child of its parent
      u = z.p.p.right; // y is the uncle of z
      if u != NULL and u.color = 'RED'){// case 3.1
        z.p.color = 'BLACK'; u.color = 'BLACK';  z.p.p.color = 'RED';
        z = z.p.p;// repeat with the grand-parent of z
      } else {
        if z = z.p.right then { z = z.p;  r = leftRotate(r,z);  }
        z.p.color = 'BLACK'; z.p.p.color = 'R';  r = rightRotate(r, z.p.p);
      }
    } else {// z->p is the right-child of its parent
      u = z.p.p.left;
      if u != NULL and u.color = 'RED' then {// case 3.1
        z.p.color = 'BLACK';  u.color = 'BLACK'; z.p.p.color = 'RED'; z = z.p.p;
      } else {
        if z = z.p.left then {  z = z.p;   r = rightRotate(r, z); }
        z.p.color = 'BLACK';  z.p.p.color = 'RED'; r = leftRotate(r, z.p.p);
      }
    }
  }
  r.color = 'BLACK';  return r;
}
```

HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !