Lecture 19: Red-Black Trees I

2023/10/11

詹博华（中国科学院软件研究所）

Binary Search Trees

- Classical data structure for maintaining a set of items that can be compared with each other (numbers, strings, etc).
- Also used to maintain a mapping of key-value pairs, where keys can be compared.
- Support insert, delete and search.
- Examples:
  - Student information indexed by name.
  - Transactions indexed by time.
  - Use within other algorithms.

Implementation in C++ (STL)

std::map

Defined in header <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

`std::map` is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function `Compare`. Search, removal, and insertion operations have logarithmic complexity. Maps are usually implemented as red-black trees.

Source: https://en.cppreference.com/w/cpp/container/map

Implementation in Java

java.util

Class TreeMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.TreeMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Implemented Interfaces:

Serializable, Cloneable, Map<K,V>, NavigableMap<K,V>, SortedMap<K,V>

A Red-Black tree based `NavigableMap` implementation. The map is sorted according to the natural ordering of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used.

Source: https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html

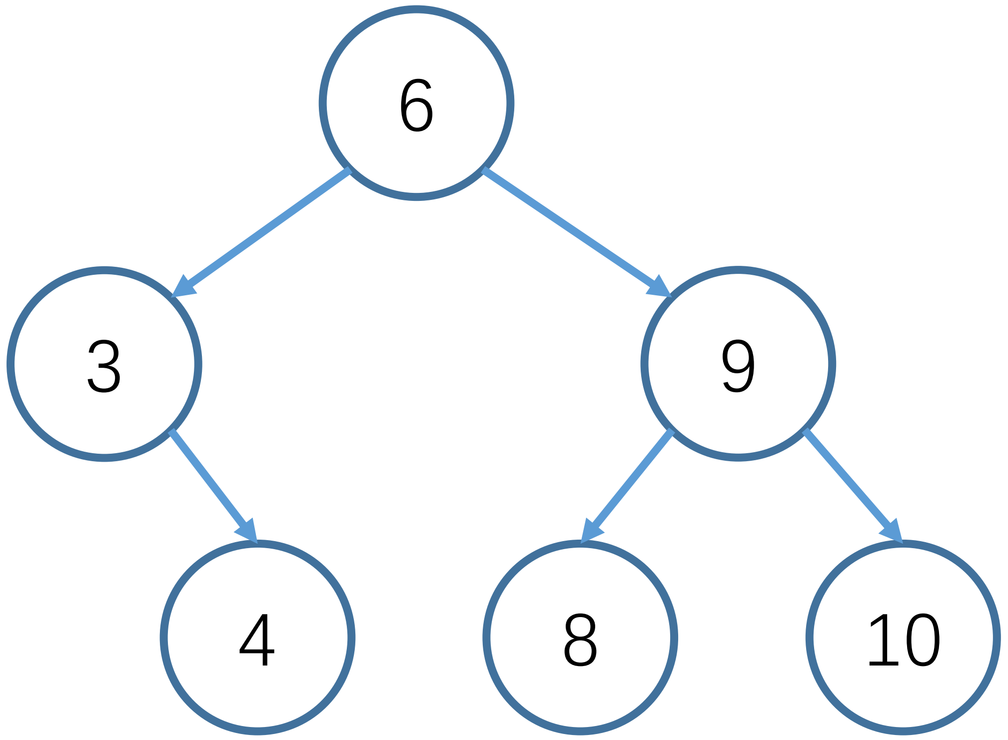
Binary Search Trees

- Each node records a key (or a key-value pair in case of maps), and pointers to two subtrees.
- Main property (invariant): all keys in the left subtree are smaller, and all keys in the right subtree are bigger than the key in the node.
- Insert, delete, search takes $\mathrm{O}(\log n)$ time for average inputs.
- Operations can take $\mathrm{O}(n)$ time in the worst case.

Binary Search Trees: average case

Steps:

- Insert 6
- Insert 3
- Insert 4
- Insert 9
- Insert 8
- Insert 10
- Search 8
- Search 4

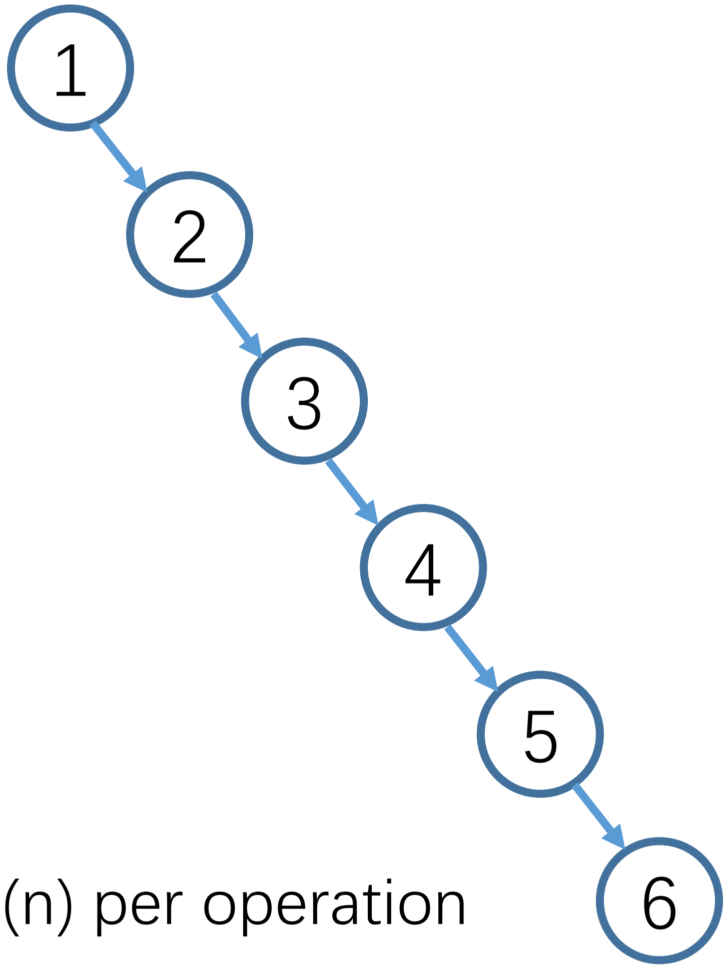


O(log n) per operation

Binary Search Trees: worst case

Steps:

- Insert 1
- Insert 2
- Insert 3
- Insert 4
- Insert 5
- Insert 6
- …



O(n) per operation

Binary Search Trees: implementation

- Implementation of search: recursive version

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

Binary Search Trees: implementation

- Implementation of search: iterative version

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

Binary Search Trees: Insert

- Implementation of insert: iterative version
- Line 3-7: traverse to a leaf y.
- Line 8: set parent of z to y.
- Line 9-13: add z to the tree.

```
TREE-INSERT(T, z)
1   y = NIL
2   x = T.root
3   while x ≠ NIL
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

Self-Balancing Search Trees

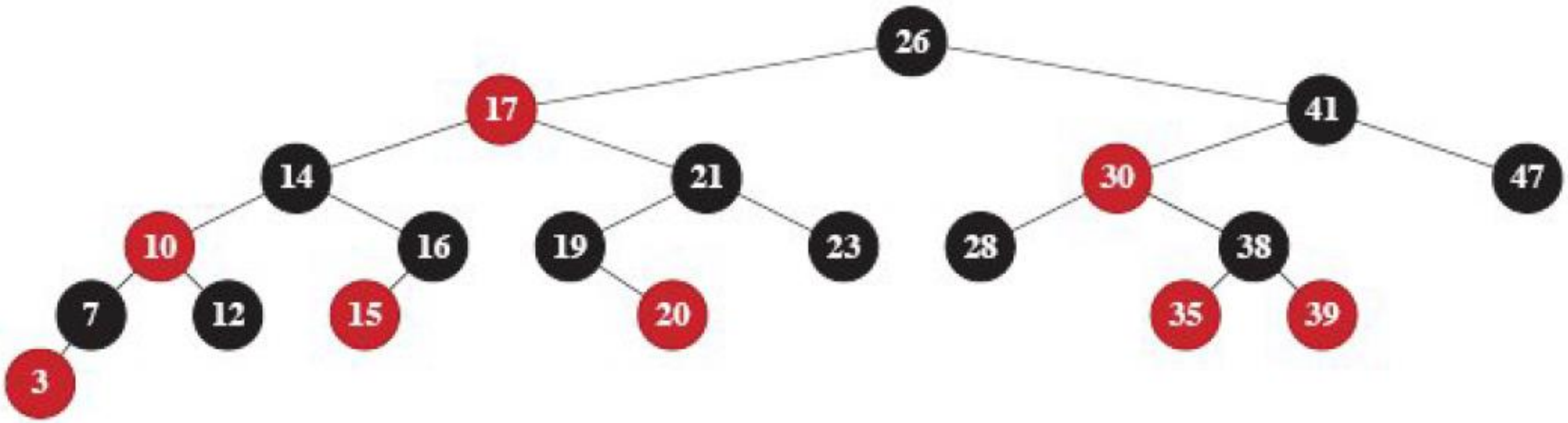
- Bad performance of binary search trees can be attributed to the tree being poorly balanced.
- Signs of poor balance:
  - One side of tree is much larger than the other.
  - Height of the tree grows faster than $\log n$.
- Self-balancing search trees are improvements of binary search trees, including mechanism to make sure that the tree stays balanced, whatever the input is.

Self-Balancing Search Trees: Proposals

- Red-black trees (the most commonly implemented).
- AVL trees (an early proposal, based on recording height).
- Treaps (uses randomization, tree+heap).
- Splay trees (amortized $O(\log n)$ time).

Red-Black Trees

- Colors each node **red** or **black**.
- The root is black.
- If a node is red, then both its children are black.
- All paths from root to leaves have the same number of black nodes.



Height and Black-Height

- Define the *black height* of a red-black tree to be the number of black nodes along any path from root to leave in the tree.

$$bh \text{ – black height of tree}$$

- Define the height of a red-black tree to be the maximum length of any path from root to leave in the tree.

$$h \text{ – height of tree}$$

- We have $h \leq 2 \cdot bh$ (since path do not contain consecutive red nodes).
- There are at least $2^{bh} - 1$ black nodes in the tree (proof next slide).

Height and Black-Height

- There are at least $2^{bh} - 1$ black nodes in the tree.
- **Proof by induction:** for any $n \geq 1$, any tree with black height n has at least $2^n - 1$ black nodes.
  - Base case ($n = 1$): certainly has at least $2^1 - 1 = 1$ black node.
  - Inductive case ($n = k + 1$):
    - If the root is black, then both subtrees have black height k, by induction hypothesis they each have at least $2^k - 1$ black nodes. So there are at least $2 \cdot (2^k - 1) + 1 = 2^{k+1} - 1$ black nodes in total.
    - If the root is red, then both subtrees have black height $k + 1$, and have black roots. Reduce to the case where root is black.

Red-Black Tree: main result

- Let

$$n \text{ – number of nodes in the tree}$$

- From the previous slide, we have:

$$n \geq 2^{bh} - 1$$
$$h \leq 2 \cdot bh$$

- This implies:

$$h \leq 2 \cdot \lg(n+1) = O(\log n)$$

- That is, the height of tree always grows according to $\log n$.

Red-Black Tree: operations

- Search on red-black tree is the same as before, ignoring the color.

Search takes $O(\log n)$ time

- Insertion and deletion is more complicated, making use of rotations. The aim is to always maintain the properties of red-black tree.

Insertion and deletion takes $O(\log n)$ time

- Next: left and right rotations.