CSE 326: Data Structures

Topic #7:
Don’t Sweat It, **Splay** It!

Ashish Sabharwal
Autumn, 2003

Today’s Outline

- *TO DO*
  - Finish Homework #1; due Friday at the beginning of class
  - Find a partner for Project #2; send me email by Friday evening
- Review AVL Trees
- **Splay Trees**

2

AVL Trees Revisited

- Balance condition:
  For every node x, $-1 \le \text{balance}(x) \le 1$
  - Strong enough : Worst case depth is $\Theta(\log n)$
  - Easy to maintain : *one* single or double rotation
- Guaranteed $\Theta(\log n)$ running time for
  - Find ?
  - Insert ?
  - Delete ?
  - buildTree ?

3

AVL Trees Revisited

- What extra info did we maintain in each node?
- Where were rotations performed?
- How did we locate this node?

4

Other Possibilities?

- Could use different balance conditions, different ways to maintain balance, different guarantees on running time, …
- Why? Aren’t AVL trees perfect?
- Many other balanced BST data structures
  - Red-Black trees
  - AA trees
  - **Splay Trees**
  - 2-3 Trees
  - **B-Trees**
  - …

5

Splay Trees

- Blind adjusting version of AVL trees
  - Why worry about balances? Just rotate anyway!
- *Amortized* time per operations is $O(\log n)$
- Worst case time per operation is $O(n)$
  - But guaranteed to happen rarely

Insert/Find always rotate node *to the root*!

Subject GRE Analogy question:
AVL is to Splay trees as ___________ is to __________

6

Recall: Amortized Complexity

If a sequence of M operations takes O(M f(*n*)) time, we say the amortized runtime is O(f(*n*)).

- Worst case time *per operation* can still be large, say O(*n*)
- Worst case time for <u>*any*</u> *sequence* of M operations is O(M f(*n*))

Average time *per operation* for *any* sequence is O(f(*n*))

Amortized complexity is *worst-case* guarantee over *sequences* of operations.

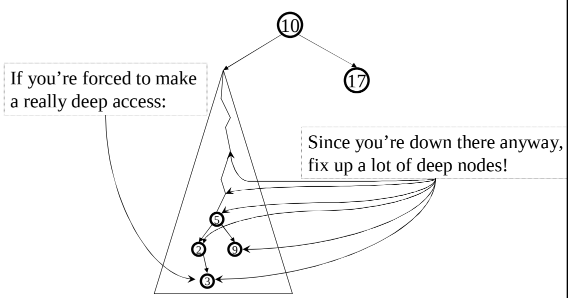
7

Recall: Amortized Complexity

- Is amortized guarantee any weaker than worstcase?
- Is amortized guarantee any stronger than averagecase?
- Is average case guarantee good enough in practice?
- Is amortized guarantee good enough in practice?

8

The Splay Tree Idea



9

Find/Insert in Splay Trees

1. Find or insert a node ***k***
2. **Splay *k* to the root using:**
   zig-zag, zig-zig, or plain old zig rotation

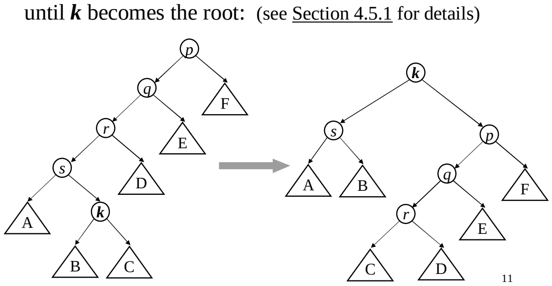
Why could this be good??

1. Helps the new root, ***k***
   - *Great if x is accessed again*
2. And helps many others!
   - *Great if many others on the path are accessed*

10

Splaying node *k* to the root: Need to be careful!

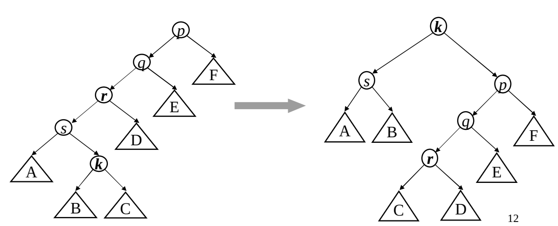
One option is to repeatedly use AVL single rotation until ***k*** becomes the root: (see <u>Section 4.5.1</u> for details)



11

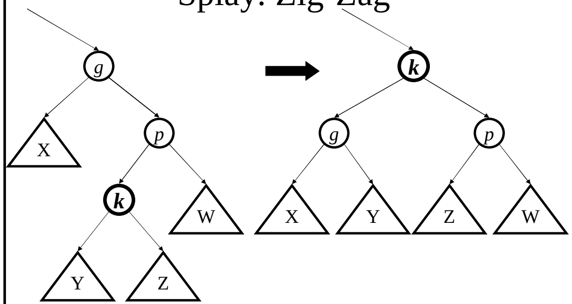
Splaying node *k* to the root: Need to be careful!

What's bad about this process?



12

Splay: Zig-Zag*

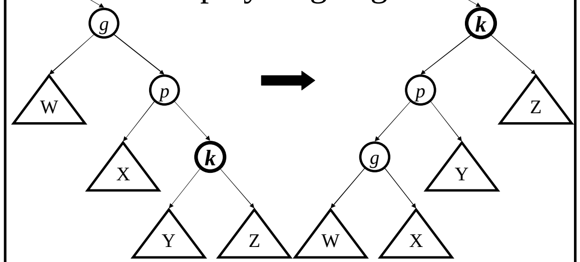


*Just like an…

Which nodes improve depth?

13

Splay: Zig-Zig*



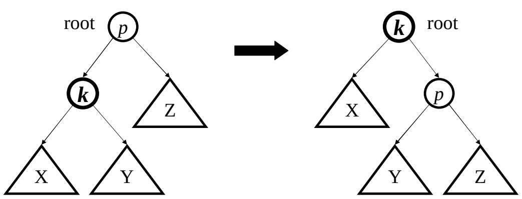
*Is this an AVL zig-zig?

How to implement?

Why does this help?

14

Special Case for Root: Zig



Relative depth of *p*, Y, Z?

Relative depth of everyone else?

Why not drop zig-zig and just zig all the way?

15

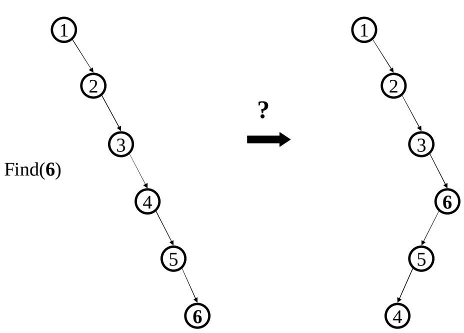
Does Splaying Help *Every* Node?

Only amortized guarantee!

Let's see an example…

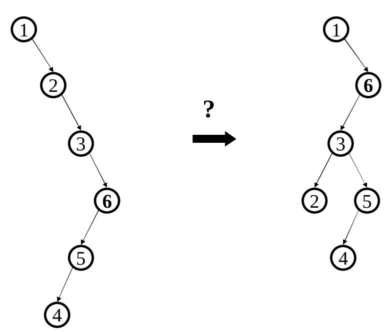
16

Splaying Example: Find(6)



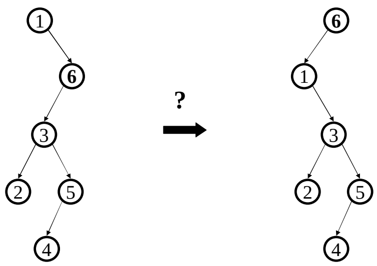
17

Still Splaying 6



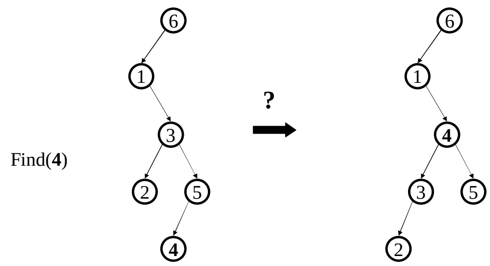
18

Finally…



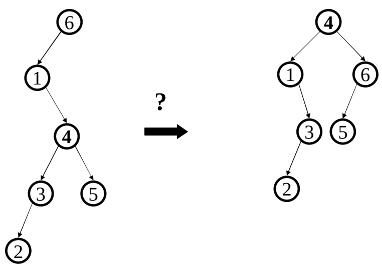
19

Another Splay: Find(4)



20

Example Splayed Out



21

But Wait…

What happened here?

Didn’t *two* find operations take linear time instead of logarithmic?

What about the amortized $\Theta(\log n)$ guarantee?

22

Why Splaying Helps

- If a node *n* on the access path is at depth *d* before the splay, it’s at about depth *d/2* after the splay
  - Exceptions are the root, the child of the root (and descendants), and the node splayed
- Overall, nodes which are low on the access path tend to move closer to the root

23

Practical Benefit of Splaying

- No heights to maintain, no imbalance to check for
  - Less storage per node, easier to code
- Often data that is accessed once, is soon accessed again!
  - Splaying does implicit *caching* by bringing it to the root

24

Splay Operations: Find

- Find the node in normal BST manner
- Splay the node to the root
  - if node not found, splay what would have been its parent

What if we didn't splay?

25

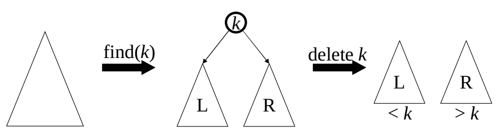
Splay Operations: Insert

- Insert the node in normal BST manner
- Splay the node to the root

What if we didn't splay?

26

Splay Operations: Remove

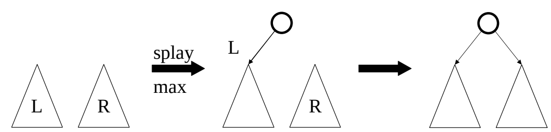


Now what?

27

Join

Join(L, R): given two trees such that L < R, merge them



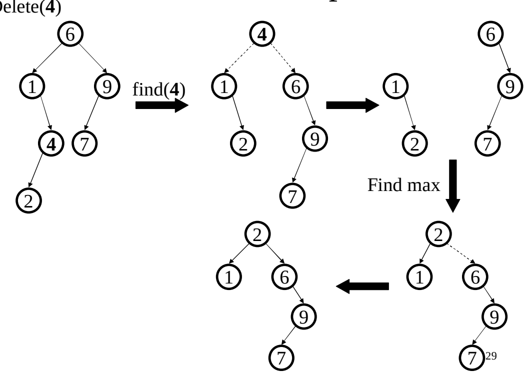
Splay on the maximum element in L, then attach R

Does this work to join *any* two trees?

28

Delete Example

Delete(**4**)



29

A Nifty Splay Operation: Splitting

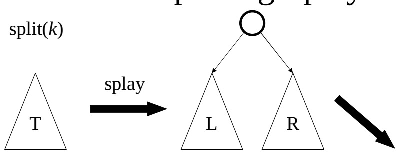
Split(T, *k*) creates two BSTs L and R:

- all elements of T are in either L or R ($T = L \cup R$)
- all elements in L are $\leq k$
- all elements in R are $\geq k$
- L and R share no elements ($L \cap R = \emptyset$)

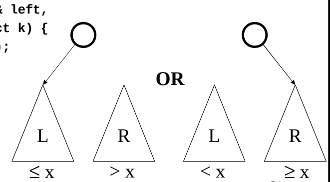
How do we split a splay tree?

30

Splitting Splays

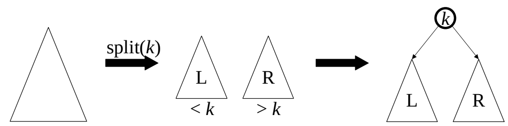



```
void split(Node * root, Node *& left,
           Node *& right, Object k) {
  Node * target = root->find(k);
  splay(target);
  if (target < k) {
    left = target->left;
    target->left = NULL;
    right = target;
  }
  ...
}
```



Aha, Another Way to Insert!

Insert(k)




```
void insert(Node *& root, Object k) {
  Node * left, * right;
  split(root, left, right, k);
  root = new Node(k, left, right);
}
```

Interesting note: split-and-insert was the original algorithm. But insert-and-splay has better constants

Splay Tree Summary

- All operations are in amortized $\Theta(\log n)$ time
- Splaying can be done top-down; better because:
  - only one pass
  - no recursion or parent pointers necessary
  - *we didn't cover top-down in class*
- Splay trees are *very* effective search trees
  - Relatively simple
  - No extra fields required
  - Excellent *locality* properties: frequently accessed keys are cheap to find

To Do

- Finish reading Chapter 4
- Homework #1 due Friday
- Project #2 will be released Friday
  - Pick a partner!