



CSE332: Data Abstractions

Lecture 6: Dictionaries; Binary Search Trees

Ruth Anderson
Winter 2011

Announcements

- **Project 1** – phase B due Tues Jan 18th 11pm via catalyst
- **Homework 1** – due NOW!!
- **Homework 2** – due Friday Jan 21st at **beginning** of class
- No class on Monday Jan 17th
- Ruth's Office hours moved to Tues Jan 18th 12:30-1:30pm

1/14/2011

2

Today

- Dictionaries
- Trees

1/14/2011

3

Where we are

Studying the absolutely essential ADTs of computer science and classic data structures for implementing them

ADTs so far:

1. Stack: `push, pop, isEmpty, ...`
2. Queue: `enqueue, dequeue, isEmpty, ...`
3. Priority queue: `insert, deleteMin, ...`

Next:

4. Dictionary (a.k.a. Map): associate keys with values
 - probably the most common, way more than priority queue

1/14/2011

4

The Dictionary (a.k.a. Map, a.k.a. Associative Array) ADT

- Data:
 - set of (key, value) pairs
 - keys must be *comparable* ($<$ or $>$ or $=$)
- Primary Operations:
 - `insert(key, val)`: places (key, val) in map
 - If key already used, overwrites existing entry
 - `find(key)`: returns val associated with key
 - `delete(key)`

1/14/2011

5

The Dictionary (a.k.a. Map) ADT

- Data:
 - set of (key, value) pairs
 - keys must be comparable
- Operations:
 - `insert(key, value)`
 - `find(key)`
 - `delete(key)`
 - ...

Will tend to emphasize the keys,
don't forget about the stored values



1/14/2011

6

Comparison: Set ADT vs. Dictionary ADT

The *Set* ADT is like a Dictionary without any values

- A key is *present* or not (no repeats)

For **find**, **insert**, **delete**, there is little difference

- In dictionary, values are “just along for the ride”
- So *same data-structure ideas* work for dictionaries and sets
 - Java HashSet implemented using a HashMap, for instance

Set ADT may have other important operations

- **union**, **intersection**, **is_subset**
- notice these are operators on 2 sets

1/14/2011

7

Dictionary data structures

Will spend the next 1.5-2 weeks looking at dictionaries with three different data structures

1. AVL trees
 - Binary search trees with *guaranteed balancing*
2. B-Trees
 - Also always balanced, but different and shallower
 - B!=Binary; B-Trees generally have large branching factor
3. Hashtables
 - Not tree-like at all

Skipping: Other balanced trees (red-black, splay)

But first some applications and less efficient implementations...

1/14/2011

8

A Modest Few Uses

Any time you want to store information according to some key and be able to retrieve it efficiently

- Lots of programs do that!

- Networks: router tables
- Operating systems: page tables
- Compilers: symbol tables
- Databases: dictionaries with other nice properties
- Search: inverted indexes, phone directories, ...
- Biology: genome maps
- ...

1/14/2011

9

Simple implementations

For dictionary with n key/value pairs

insert find delete

- Unsorted linked-list
- Unsorted array
- Sorted linked list
- Sorted array

We'll see a Binary Search Tree (BST) probably does better, but not in the worst case unless we keep it balanced

1/14/2011

10

Simple implementations

For dictionary with n key/value pairs

insert find delete

- Unsorted linked-list $O(1)^*$ $O(n)$ $O(n)$
- Unsorted array $O(1)^*$ $O(n)$ $O(n)$
- Sorted linked list $O(n)$ $O(n)$ $O(n)$
- Sorted array $O(n)$ $O(\log n)$ $O(n)$

We'll see a Binary Search Tree (BST) probably does better, but not in the worst case unless we keep it balanced

*Note: If we do not allow duplicates values to be inserted, we would need to do $O(n)$ work to check for a key's existence before insertion

1/14/2011

11

Lazy Deletion

10	12	24	30	41	42	44	45	50
✓	✗	✓	✓	✓	✓	✗	✓	✓

A *general technique* for making **delete** as fast as **find**:

- Instead of actually removing the item just mark it deleted

Plusses:

- Simpler
- Can do removals later in batches
- If re-added soon thereafter, just unmark the deletion

Minuses:

- Extra *space* for the “is-it-deleted” flag
- Data structure full of deleted nodes wastes *space*
- **find** $O(\log m)$ *time* where m is data-structure size (okay)
- May complicate other operations

1/14/2011

12

Some tree terms (mostly review)

- There are many kinds of trees
 - Every binary tree is a tree
 - Every list is kind of a tree (think of "next" as the one child)
- There are many kinds of binary trees
 - Every binary min heap is a binary tree
 - Every binary search tree is a binary tree
- A tree can be balanced or not
 - A balanced tree with n nodes has a height of $O(\log n)$
 - Different tree data structures have different "balance conditions" to achieve this

1/14/2011

13

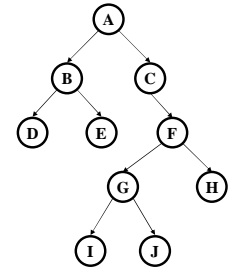
Binary Trees

- Binary tree is empty or
 - a root (with data)
 - a left subtree (maybe empty)
 - a right subtree (maybe empty)

- Representation:

Data	
left pointer	right pointer

- For a dictionary, data will include a key and a value



1/14/2011

14

Binary Tree: Some Numbers

Recall: height of a tree = longest path from root to leaf (count # of edges)

For binary tree of height h :

- max # of leaves:
- max # of nodes:
- min # of leaves:
- min # of nodes:

1/14/2011

15

Binary Trees: Some Numbers

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height h :

- max # of leaves: 2^h
- max # of nodes: $2^{(h+1)} - 1$
- min # of leaves: 1
- min # of nodes: $h + 1$

For n nodes, we cannot do better than $O(\log n)$ height, and we want to avoid $O(n)$ height

1/14/2011

16

Calculating height

What is the height of a tree with root r ?

```

int treeHeight(Node root) {
    ???
}
  
```

1/14/2011

17

Calculating height

What is the height of a tree with root r ?

```

int treeHeight(Node root) {
    if (root == null)
        return -1;
    return 1 + max(treeHeight(root.left),
                  treeHeight(root.right));
}
  
```

Running time for tree with n nodes: $O(n)$ – single pass over tree

Note: non-recursive is painful – need your own stack of pending nodes; much easier to use recursion's call stack

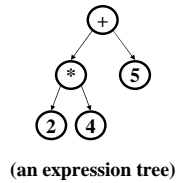
1/14/2011

18

Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

- *Pre-order*: root, left subtree, right subtree
- *In-order*: left subtree, root, right subtree
- *Post-order*: left subtree, right subtree, root

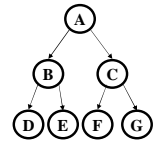


1/14/2011

19

More on traversals

```
void inOrderTraversal(Node t){
    if(t != null) {
        traverse(t.left);
        process(t.element);
        traverse(t.right);
    }
}
```



A
B
D
E
C
F
G

Sometimes order doesn't matter

- Example: sum all elements

Sometimes order matters

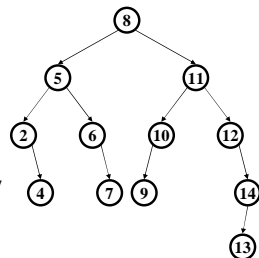
- Example: print tree with parent above indented children (pre-order)
- Example: evaluate an expression tree (post-order)

1/14/2011

20

Binary Search Tree

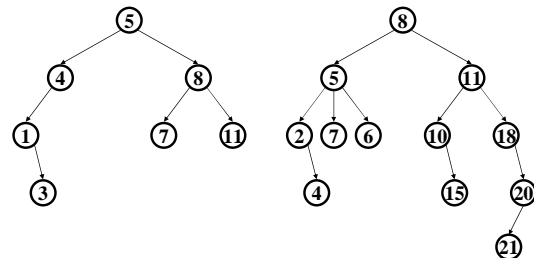
- Structural property ("binary")
 - each node has ≤ 2 children
 - result: keeps operations simple
- Order property
 - all keys in left subtree smaller than node's key
 - all keys in right subtree larger than node's key
 - result: easy to find any given key



1/14/2011

21

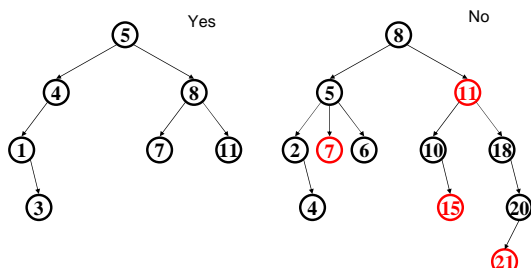
Are these BSTs?



1/14/2011

22

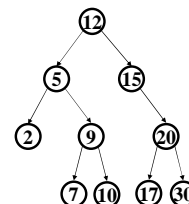
Are these BSTs?



1/14/2011

23

Find in BST, Recursive

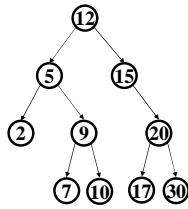


```
Data find(Key key, Node root){
    if(root == null)
        return null;
    if(key < root.key)
        return find(key, root.left);
    if(key > root.key)
        return find(key, root.right);
    return root.data;
}
```

1/14/2011

24

Find in BST, Iterative



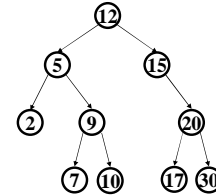
```
Data find(Key key, Node root){
while(root != null
    && root.key != key) {
    if(key < root.key)
        root = root.left;
    else(key > root.key)
        root = root.right;
    }
if(root == null)
    return null;
return root.data;
}
```

1/14/2011

25

Other “finding operations”

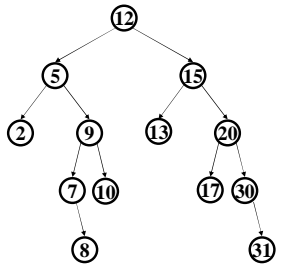
- Find minimum node
- Find maximum node
- Find predecessor?
- Find successor?



1/14/2011

26

Insert in BST



```
insert(13)
insert(8)
insert(31)
```

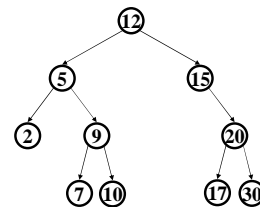
(New) insertions happen only at leaves – easy!

1. Find
2. Create a new node

1/14/2011

27

Deletion in BST



Why might deletion be harder than insertion?

1/14/2011

28

Deletion

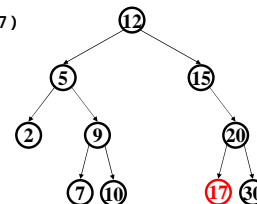
- Removing an item disrupts the tree structure
- Basic idea: **find** the node to be removed, then “fix” the tree so that it is still a binary search tree
- Three cases:
 - node has no children (leaf)
 - node has one child
 - node has two children

1/14/2011

29

Deletion – The Leaf Case

delete(17)

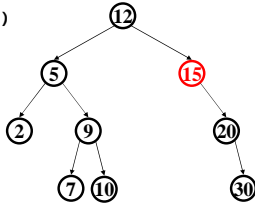


1/14/2011

30

Deletion – The One Child Case

delete(15)

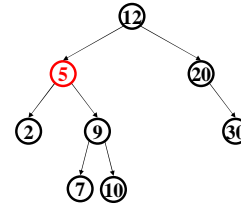


1/14/2011

31

Deletion – The Two Child Case

delete(5)



What can we replace 5 with?

1/14/2011

32

Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees

Options:

- *successor* from right subtree: `findMin(node.right)`
- *predecessor* from left subtree: `findMax(node.left)`
 - These are the easy cases of predecessor/successor

Now delete the original node containing *successor* or *predecessor*

- Leaf or one child case – easy cases of delete!

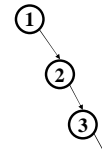
1/14/2011

33

BuildTree for BST

- We had `buildHeap`, so let's consider `buildTree`
- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST

- If inserted in given order, what is the tree?
- What big-O runtime for this kind of sorted input?
- Is inserting in the reverse order any better?



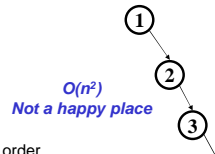
1/14/2011

34

BuildTree for BST

- We had `buildHeap`, so let's consider `buildTree`
- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST

- If inserted in given order, what is the tree?
- What big-O runtime for this kind of sorted input?
- Is inserting in the reverse order any better?



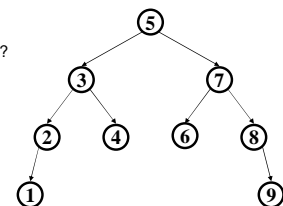
1/14/2011

35

BuildTree for BST

- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
- What we if could somehow re-arrange them
 - median first, then left median, right median, etc.
 - 5, 3, 7, 2, 1, 4, 8, 6, 9


- What tree does that give us?
- What big-O runtime?



1/14/2011

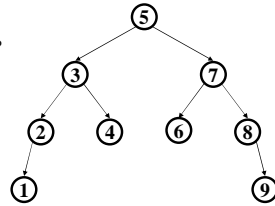
36

BuildTree for BST

- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
 - What we if could somehow re-arrange them
 - median first, then left median, right median, etc.
 - 5, 3, 7, 2, 1, 4, 8, 6, 9
 - What tree does that give us?
 - What big-O runtime?
- 
- ```

graph TD
 5((5)) --- 3((3))
 3 --- L1[]
 style L1 fill:none,stroke:none
 L1 --- L2[]
 style L2 fill:none,stroke:none
 L2 --- L3[]
 style L3 fill:none,stroke:none
 L3 --- L4[]
 style L4 fill:none,stroke:none
 L4 --- L5[]
 style L5 fill:none,stroke:none
 L5 --- L6[]
 style L6 fill:none,stroke:none
 L6 --- L7[]
 style L7 fill:none,stroke:none
 L7 --- L8[]
 style L8 fill:none,stroke:none
 L8 --- L9[]
 style L9 fill:none,stroke:none
 L9 --- L10[]
 style L10 fill:none,stroke:none
 L10 --- L11[]
 style L11 fill:none,stroke:none
 L11 --- L12[]
 style L12 fill:none,stroke:none
 L12 --- L13[]
 style L13 fill:none,stroke:none
 L13 --- L14[]
 style L14 fill:none,stroke:none
 L14 --- L15[]
 style L15 fill:none,stroke:none
 L15 --- L16[]
 style L16 fill:none,stroke:none
 L16 --- L17[]
 style L17 fill:none,stroke:none
 L17 --- L18[]
 style L18 fill:none,stroke:none
 L18 --- L19[]
 style L19 fill:none,stroke:none
 L19 --- L20[]
 style L20 fill:none,stroke:none
 L20 --- L21[]
 style L21 fill:none,stroke:none
 L21 --- L22[]
 style L22 fill:none,stroke:none
 L22 --- L23[]
 style L23 fill:none,stroke:none
 L23 --- L24[]
 style L24 fill:none,stroke:none
 L24 --- L25[]
 style L25 fill:none,stroke:none
 L25 --- L26[]
 style L26 fill:none,stroke:none
 L26 --- L27[]
 style L27 fill:none,stroke:none
 L27 --- L28[]
 style L28 fill:none,stroke:none
 L28 --- L29[]
 style L29 fill:none,stroke:none
 L29 --- L30[]
 style L30 fill:none,stroke:none
 L30 --- L31[]
 style L31 fill:none,stroke:none
 L31 --- L32[]
 style L32 fill:none,stroke:none
 L32 --- L33[]
 style L33 fill:none,stroke:none
 L33 --- L34[]
 style L34 fill:none,stroke:none
 L34 --- L35[]
 style L35 fill:none,stroke:none
 L35 --- L36[]
 style L36 fill:none,stroke:none
 L36 --- L37[]
 style L37 fill:none,stroke:none
 L37 --- L38[]
 style L38 fill:none,stroke:none
 L38 --- L39[]
 style L39 fill:none,stroke:none
 L39 --- L40[]
 style L40 fill:none,stroke:none
 L40 --- L41[]
 style L41 fill:none,stroke:none
 L41 --- L42[]
 style L42 fill:none,stroke:none
 L42 --- L43[]
 style L43 fill:none,stroke:none
 L43 --- L44[]
 style L44 fill:none,stroke:none
 L44 --- L45[]
 style L45 fill:none,stroke:none
 L45 --- L46[]
 style L46 fill:none,stroke:none
 L46 --- L47[]
 style L47 fill:none,stroke:none
 L47 --- L48[]
 style L48 fill:none,stroke:none
 L48 --- L49[]
 style L49 fill:none,stroke:none
 L49 --- L50[]
 style L50 fill:none,stroke:none
 L50 --- L51[]
 style L51 fill:none,stroke:none
 L51 --- L52[]
 style L52 fill:none,stroke:none
 L52 --- L53[]
 style L53 fill:none,stroke:none
 L53 --- L54[]
 style L54 fill:none,stroke:none
 L54 --- L55[]
 style L55 fill:none,stroke:none
 L55 --- L56[]
 style L56 fill:none,stroke:none
 L56 --- L57[]
 style L57 fill:none,stroke:none
 L57 --- L58[]
 style L58 fill:none,stroke:none
 L58 --- L59[]
 style L59 fill:none,stroke:none
 L59 --- L60[]
 style L60 fill:none,stroke:none
 L60 --- L61[]
 style L61 fill:none,stroke:none
 L61 --- L62[]
 style L62 fill:none,stroke:none
 L62 --- L63[]
 style L63 fill:none,stroke:none
 L63 --- L64[]
 style L64 fill:none,stroke:none
 L64 --- L65[]
 style L65 fill:none,stroke:none
 L65 --- L66[]
 style L66 fill:none,stroke:none
 L66 --- L67[]
 style L67 fill:none,stroke:none
 L67 --- L68[]
 style L68 fill:none,stroke:none
 L68 --- L69[]
 style L69 fill:none,stroke:none
 L69 --- L70[]
 style L70 fill:none,stroke:none
 L70 --- L71[]
 style L71 fill:none,stroke:none
 L71 --- L72[]
 style L72 fill:none,stroke:none
 L72 --- L73[]
 style L73 fill:none,stroke:none
 L73 --- L74[]
 style L74 fill:none,stroke:none
 L74 --- L75[]
 style L75 fill:none,stroke:none
 L75 --- L76[]
 style L76 fill:none,stroke:none
 L76 --- L77[]
 style L77 fill:none,stroke:none
 L77 --- L78[]
 style L78 fill:none,stroke:none
 L78 --- L79[]
 style L79 fill:none,stroke:none
 L79 --- L80[]
 style L80 fill:none,stroke:none
 L80 --- L81[]
 style L81 fill:none,stroke:none
 L81 --- L82[]
 style L82 fill:none,stroke:none
 L82 --- L83[]
 style L83 fill:none,stroke:none
 L83 --- L84[]
 style L84 fill:none,stroke:none
 L84 --- L85[]
 style L85 fill:none,stroke:none
 L85 --- L86[]
 style L86 fill:none,stroke:none
 L86 --- L87[]
 style L87 fill:none,stroke:none
 L87 --- L88[]
 style L88 fill:none,stroke:none
 L88 --- L89[]
 style L89 fill:none,stroke:none
 L89 --- L90[]
 style L90 fill:none,stroke:none
 L90 --- L91[]
 style L91 fill:none,stroke:none
 L91 --- L92[]
 style L92 fill:none,stroke:none
 L92 --- L93[]
 style L93 fill:none,stroke:none
 L93 --- L94[]
 style L94 fill:none,stroke:none
 L94 --- L95[]
 style L95 fill:none,stroke:none
 L95 --- L96[]
 style L96 fill:none,stroke:none
 L96 --- L97[]
 style L97 fill:none,stroke:none
 L97 --- L98[]
 style L98 fill:none,stroke:none
 L98 --- L99[]
 style L99 fill:none,stroke:none
 L99 --- L100[]
 style L100 fill:none,stroke:none
 L100 --- L101[]
 style L101 fill:none,stroke:none
 L101 --- L102[]
 style L102 fill:none,stroke:none
 L102 --- L103[]
 style L103 fill:none,stroke:none
 L103 --- L104[]
 style L104 fill:none,stroke:none
 L104 --- L105[]
 style L105 fill:none,stroke:none
 L105 --- L106[]
 style L106 fill:none,stroke:none
 L106 --- L107[]
 style L107 fill:none,stroke:none
 L107 --- L108[]
 style L108 fill:none,stroke:none
 L108 --- L109[]
 style L109 fill:none,stroke:none
 L109 --- L110[]
 style L110 fill:none,stroke:none
 L110 --- L111[]
 style L111 fill:none,stroke:none
 L111 --- L112[]
 style L112 fill:none,stroke:none
 L112 --- L113[]
 style L113 fill:none,stroke:none
 L113 --- L114[]
 style L114 fill:none,stroke:none
 L114 --- L115[]
 style L115 fill:none,stroke:none
 L115 --- L116[]
 style L116 fill:none,stroke:none
 L116 --- L117[]
 style L117 fill:none,stroke:none
 L117 --- L118[]
 style L118 fill:none,stroke:none
 L118 --- L119[]
 style L119 fill:none,stroke:none
 L119 --- L120[]
 style L120 fill:none,stroke:none
 L120 --- L121[]
 style L121 fill:none,stroke:none
 L121 --- L122[]
 style L122 fill:none,stroke:none
 L122 --- L123[]
 style L123 fill:none,stroke:none
 L123 --- L124[]
 style L124 fill:none,stroke:none
 L124 --- L125[]
 style L125 fill:none,stroke:none
 L125 --- L126[]
 style L126 fill:none,stroke:none
 L126 --- L127[]
 style L127 fill:none,stroke:none
 L127 --- L128[]
 style L128 fill:none,stroke:none
 L128 --- L129[]
 style L129 fill:none,stroke:none
 L129 --- L130[]
 style L130 fill:none,stroke:none
 L130 --- L131[]
 style L131 fill:none,stroke:none
 L131 --- L132[]
 style L132 fill:none,stroke:none
 L132 --- L133[]
 style L133 fill:none,stroke:none
 L133 --- L134[]
 style L134 fill:none,stroke:none
 L134 --- L135[]
 style L135 fill:none,stroke:none
 L135 --- L136[]
 style L136 fill:none,stroke:none
 L136 --- L137[]
 style L137 fill:none,stroke:none
 L137 --- L138[]
 style L138 fill:none,stroke:none
 L138 --- L139[]
 style L139 fill:none,stroke:none
 L139 --- L140[]
 style L140 fill:none,stroke:none
 L140 --- L141[]
 style L141 fill:none,stroke:none
 L141 --- L142[]
 style L142 fill:none,stroke:none
 L142 --- L143[]
 style L143 fill:none,stroke:none
 L143 --- L144[]
 style L144 fill:none,stroke:none
 L144 --- L145[]
 style L145 fill:none,stroke:none
 L145 --- L146[]
 style L146 fill:none,stroke:none
 L146 --- L147[]
 style L147 fill:none,stroke:none
 L147 --- L148[]
 style L148 fill:none,stroke:none
 L148 --- L149[]
 style L149 fill:none,stroke:none
 L149 --- L150[]
 style L150 fill:none,stroke:none
 L150 --- L151[]
 style L151 fill:none,stroke:none
 L151 --- L152[]
 style L152 fill:none,stroke:none
 L152 --- L153[]
 style L153 fill:none,stroke:none
 L153 --- L154[]
 style L154 fill:none,stroke:none
 L154 --- L155[]
 style L155 fill:none,stroke:none
 L155 --- L156[]
 style L156 fill:none,stroke:none
 L156 --- L157[]
 style L157 fill:none,stroke:none
 L157 --- L158[]
 style L158 fill:none,stroke:none
 L158 --- L159[]
 style L159 fill:none,stroke:none
 L159 --- L160[]
 style L160 fill:none,stroke:none
 L160 --- L161[]
 style L161 fill:none,stroke:none
 L161 --- L162[]
 style L162 fill:none,stroke:none
 L162 --- L163[]
 style L163 fill:none,stroke:none
 L163 --- L164[]
 style L164 fill:none,stroke:none
 L164 --- L165[]
 style L165 fill:none,stroke:none
 L165 --- L166[]
 style L166 fill:none,stroke:none
 L166 --- L167[]
 style L167 fill:none,stroke:none
 L167 --- L168[]
 style L168 fill:none,stroke:none
 L168 --- L
```


*$O(n \log n)$ , definitely better*

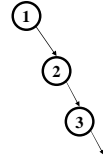


1/14/2011

37

### Unbalanced BST

- Balancing a tree at build time is insufficient, as sequences of operations can eventually transform that carefully balanced tree into the dreaded list
  - At that point, everything is  $O(n)$  and nobody is happy
    - **find**
    - **insert**
    - **delete**
- 
- ```
graph TD; 1((1)) --> 2((2)); 2 --> ...((...))
```



1/14/2011

38

Balanced BST

Observation

- BST: the shallower the better!
- For a BST with n nodes inserted in arbitrary order
 - Average height is $O(\log n)$ – see text for proof
 - Worst case height is $O(n)$
- Simple cases such as inserting in key order lead to the worst-case scenario

Solution: Require a **Balance Condition** that

1. ensures depth is always $O(\log n)$ – strong enough!
2. is easy to maintain – not too strong!

1/14/2011

39

Potential Balance Conditions

1. Left and right subtrees of the *root* have equal number of nodes
2. Left and right subtrees of the *root* have equal *height*

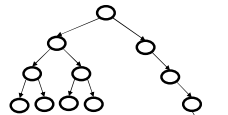
1/14/2011

40

Potential Balance Conditions

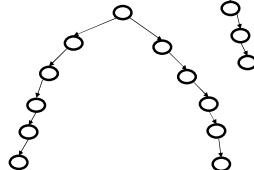
1. Left and right subtrees of the *root* have equal number of nodes

Too weak!
Height mismatch example:



- Left and right subtrees of the *root* have equal *height*

Too weak!
Double chain example:



1/14/2011

41

Potential Balance Conditions

- Left and right subtrees of every node have equal number of nodes
- Left and right subtrees of every node have equal *height*

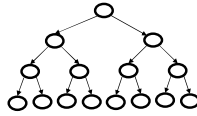
1/14/2011

42

Potential Balance Conditions

3. Left and right subtrees of every node have equal number of nodes

Too strong!
Only perfect trees ($2^n - 1$ nodes)



4. Left and right subtrees of every node have equal height

Too strong!
Only perfect trees ($2^n - 1$ nodes)

The AVL Balance Condition

Left and right subtrees of every node have heights differing by at most 1

Definition: $\text{balance}(\text{node}) = \text{height}(\text{node.left}) - \text{height}(\text{node.right})$

AVL property: for every node x , $-1 \leq \text{balance}(x) \leq 1$

- Ensures small depth
 - Will prove this by showing that an AVL tree of height h must have a number of nodes exponential in h
- Easy (well, efficient) to maintain
 - Using single and double rotations