Thursday, October 22, 2015
2:23 PM

outlineW7R

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW7R.pdf>

CS 367 Announcements
Thursday, October 22, 2015

Program p2 due 10 pm tomorrow, Friday, October 23rd
- submit java files to your `in` directory
- make sure to name your source files as specified in the submission section
- do not submit as a project/package/folder
- verify that you've submitted the correct files (ls, more, javac, java)
- partners? only ONE submits source but BOTH submit README.txt

Program p3 assigned

Last Time
- exam mechanics
- sample questions

Today

Recursion
- writing recursive code
- practice writing recursive code
- complexity of recursive methods
- practice analyzing complexity

Next Time

Read: finish *Recursion*, *Search*

Recursion
- more practice writing/analyzing
- execution tree tracing

Searching

Exams Returned

Recall Recursion

Recursion solves a problem by breaking it down into

smaller and smaller problems of the Same kind ________________

Known/obvious

until the problem is so small that it has a __________________________ **solution**

→ Why use recursion?

Simpler/concise code

→ How do you tell that a method is recursive?

Calls itself directly

Rules:

1. Every recursive method must have at least one base case (implicit or explicit).
2. Every recursive method call must make progress towards a base case.

Constructing Recursive Code

→ **Write a recursive method that computes n^m**
that is, it computes `double` n raised to an `int` power m?

recursive definition:

n! = *(N-1)!

$N^m = n * n^{m-1}$ if m > 0
$= 1$ if m = 0
$= 1/n^{-m}$ if m < 0

recursive implementation:

```
Double power (double n, int m){
        If (m == 0) return 1;
        If (m > 0) return n*power(n, m-1);
        Return 1/power(n - m);
}
```

Key Questions:

1.
How can you solve the problem in terms of smaller problems of the kind

2.
What instances of the problem can serve as base cases

3.
How does the problem size decrease with each recursive call

4.
As the problem size decreases will a base case be reached

Practice – ListADT

→ Write a recursive method that displays the values in a (non-null) list of strings.

1. To display a list, print item in current position and then <u>display</u> the remaining list.

2. A list with no (remaining) items displays nothing

3. As current advanes down the list the remaining list decreases in size by 1

4. Eventually there will be no remaining list

```
void display(ListADT<String> list) {
    Display(list, 0);}
```

```
If (list.isEmpty()) return;
System.out.println(list.remove(0));
Display(list);
```

Don't use this approach, not enough information

```
Void display(ListADT<String> list, int current){
        If (current >= list.size()) return;
        System.out.println(list.get(current));
        Display(list, current + 1);
}
```

Explicit base

```
Void display(…){
        If (current  < list.size) {
        …
        …
        }
}
```

Implicit base

Practice – Array

→ **Write a recursive method that counts the number of even values in an (non-null) array filled with integers.**

1. To count even check if current element's value is even, if so, add 1 to count evens of the remaining array Otherwise add 0

2. An array with no (remaining) elements has 0 evens

3. As the current element advances the remaining array decreases in size by one

4. Eventually there will be no remaining array to be counted

```
int evenCount(int[] array) {
        Return evenCount(array, 0);
}
Int evenCount (int[] array, int current) {
        If (current >= array.length) return 0;
        If (array[current] % 2 == 0)
                Return 1 + evenCount(array, current + 1);
        Return 0 + evenCount(array, current + 1);
}
```

Recursive companion method

*sometimes a companion method is needed to allow additional parameters to be passed in the recursive method

Analyzing Complexity of Recursive Methods

Options:

1. Informal reasoning
2. Recurrence equation

Need to determing what aspect of the problem controls the problem size

Steps

1. Write equations

Base case(s): T(problem size of base case) = Growth Rate Function for work of base case

Recursive case form: T(N) = Growth Rate Function for work of recursive case excluding calls
+ T(problem size of the recursive call)

2. Expand the equation in a table
   - -look for a pattern between N & T(N)
   - -Gives a solution based on the pattern

3. Verify the guessed solution by substituting it back into the recurrence equation

4. Do complexity simplification on the verified solution

Practice – Complexity of Recursive `evenCount`

Problem size N is Number of element in the array

1. Equations

T(0) = 1
T(N) = 1 + T(N-1)

2. Table

N	T(N)
0	1
1	1+T(0)
2	1 + T(2-1) = 1 + T(1) = 1 + 2 = 3
3	1 + T(3-1) = 1 + T(2) = 1 + 3 = 4
K	K + 1

Confessed solution

3. Verify

T(N)/K+1 = 1 + T(N-1)/K+1
N+1 = 1 + (N-1) + 1

Same, so verified

4. Complexity

O(N + 1) = O(N)

Towers of Hanoi

Algorithm

```
solveTowers(count, src, dest, spare) {
    If count == 1
        move disr from src to dest
    Else
        solveTowers(count-1, src, spare, dest)
        solveTowers(1,src,Dest,Spare)
        solveTowers(count-1,sphere,dest,src)
}
```

Complexity

Problem size N is number of disks

1. Equations

   T(1) = 1
   T(N) = 1 + T(N-1) + T(1) + T(N-1)
   = ~~2~~ * 1 + 2 * T(N-1) = 1 + 2T(N-1)

2. Table

N	T(N)
1	1
2	1 + 2T(2-1) = 1+ 2T(1) = 1 + 2*= 3
3	1 + 2T(2) = 7
4	15
5	31
k	2^k-1

3 Verify

T(N) = 1 + 2T(N-1)
------ -------
2^N-1 2^{N-1}-1
2^N-1 = 1+ 2* (2^{N-1} - 1)
= 1 + 2^N - 2
=2^N -1

4. Complexity

O(2^N-1) -> O(2^N)

outlineW8T

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW8T.pdf>

CS 367 Announcements
Tuesday, October 27, 2015

Homework h6 due 10 pm, Friday, October 30th

Program p3 due 10 pm, Sunday, November 8th

Last Time

Recursion

- practice writing recursive code
- complexity of recursive methods
- practice analyzing complexity

Today

Recursion

- more practice writing/analyzing recursion
- execution tree tracing

Searching

Exams Returned

Next Time

Read: *Trees*

Categorizing ADTs

Tree Terms

General Trees

- implementing
- determining tree height

Binary Trees

- implementing

Tree Traversals

Practice – Strings

→ Write a recursive method that determines if a string is a palindrome.

Examples:

- eye
- mom
- radar
- racecar
- Rise to vote, sir!
- Never odd or even!
- A nut for a jar of tuna.
- Campus Motto: Bottoms up, Mac.
- Ed, I saw Harpo Marx ram Oprah W aside!
- Doc note: I dissent. A fast never prevents a fatness. I diet on cod.

Assumptions: non-null input string, all spaces and punctuation removed, all lower-case

Useful string methods:

- char charAt(int index)
- int length()
- String substring(int begin, int one_past_last)

1. A string is a palindrome if the first and last chars are the same and the remaining string is a palindrome
2. An Empty string or a string with one char are both palindromes

```
Boolean isPal(String s){
        If (s.length() == 0 || s.length() == 1) return true;
        Return s.charAt(0) == s.charAt(s.length() - 1 )
                && isPal(s.substring(1, s.length() - 1 ));
```

Analyzing Recursive `isPalindrome`

Problem size N is Length of the string

1. Equations

T(0) = 1
T(1) = 1
T(N) = 1 + T(N - 2)

2. Table

N	T(N)
0	1
1	1
2	2
3	2
4	3
5	3
6	4
7	4
K	k/2 + 1

k/2 has to be integer division

3. Verify

T(N) = 1 + T(N-2)

N/2 + 1 == 1 + (N-2)/2 + 1

(N-2)/2 + 2
(N/2) -1 + 2
N/2 + 1

4. Complexity

O(N/2 + 1) = O(N)

Picking Lottery Numbers

What are your odds of winning the lottery? It depends on the number of possible combinations given how many numbers you have to pick and over what range:

Supercash - choose 6 out of 39 numbers (range 1 – 39)
Megabucks - choose 6 out of 49 numbers (range 1 – 49)
*Order doesn't matter
*Duplicates aren't allowed

N Choose K: How many combinations of K things can you make from N things?

Recursive Definition:

c(n,k) =
1) c(n-1, k-1) -> count of combinations including favorite number
+ c(n-1, k) -> count of combinations excluding favorite number

2) c(n,k) = 1 if k = n
c(n,k) = 1 if k = 0
c(n,k) = 0 if k > n

3) Range n is always decreasing by 1
Picks K is either staying same or decreasing by 1

4) c(n-1, k-1) reaches k = 0
c(n-1, k) reaches n = 0

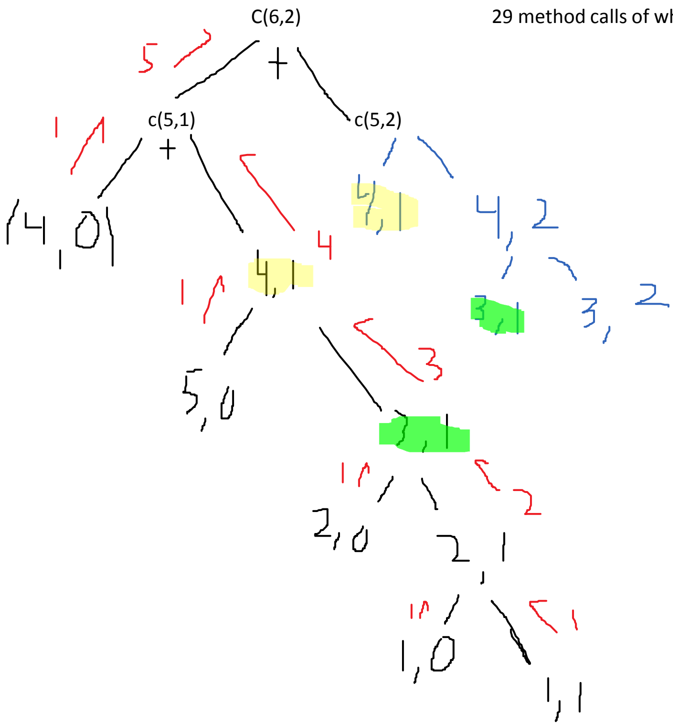
→ Implement the c(n,k) method.

```
Int c(int n, int k){
    If (k == n || k == 0) return 1;
    If (k>n) return 0;
    Return c(n-1, k-1) + c(n-1, k);
}
```

Solution N! / (K!(N-K)!)

Execution Tree Tracing of c(n,k)

29 method calls of which only 14 are unique



*used to diagram the execution of recursive code that does multiple recursive calls in recursive case
*Can reveal inefficiencies in your algorithm that can then be addressed in a number of ways
For example) add base case c(n,kk) = n IF k = 1

Searching

N is size of list searching through

Linear Search:

O(N)

```
LinSearch(L, pos, x)
        If (pos >= L.size()) return false;
        If (x == L.get(pos)) return true;
        Return LinSearch(L, pos+1, x);
```

*Search one by one ue on unsorted list

Binary Search: Divide and conquer requires a sorted list

```
BinSearch(L, first, last, x)
        If (first > last)
                return false;
        Center = (first + last) / 2
        If (x == L.get(center))
                return true
        If (x < L.get(center))
                return BinSearch(L, First, Center-1, x);
        Else
                return BinSearch(L, Ceneter+1, Last, x);
```

O(LogN)

outlineW8R

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW8R.pdf>

CS 367 Announcements
Thursday, October 29, 2015

Homework h6 due by 10 pm tomorrow, October 30th
- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h6.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Program p3 due 10 pm, Sunday, November 8th

Last Time

Recursion
- more practice writing/analyzing recursion
- execution tree tracing

Searching
Exams Returned

Today

Categorizing ADTs Part 1
Tree Terms
General Trees
- implementing
- determining tree height

Binary Trees
- implementing

Tree Traversals

Next Time

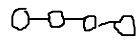
Read: start *Binary Search Trees*
Categorizing ADTs Part 2
Comparable Interface
Binary Search Tree (BST)
- BSTnodes
- BST class
- implementing print

CS Options/Courses

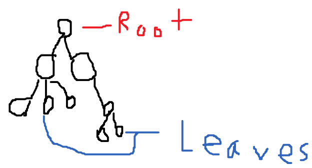
Categorizing ADTs Part 1

Based on their layout

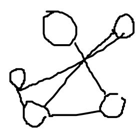
-Linear Next/Previous Relationship
1 predecessor except for First
1 successor except for last



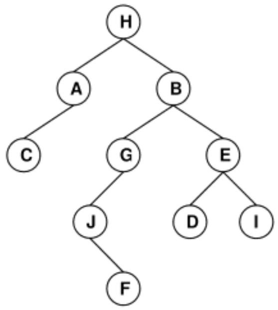
-Hierarchical (Tree)
Parent/Child Relationship
1 Predecessor except root
1 (or more) successor except leaves



-Graphical
Pairwise Relationship
0 or more predecessor
0 or more successor



Tree Terminology



1. Which is the **root**? H

2. How many **leaves** are there? 4

3. How many nodes are in the right **branch**/**subtree** of B? 3

4. Which is the **parent** of G? B

5. How many **children** does E have (**degree** of E)? 2

6. Which is the **sibling** of E? G

7. How many **descendants** does B have? 6

8. What are the **ancestors** of C? 3

9. What is the **length** of the **path** from B to D? 3

10. What is the **height** of the tree? 5

11. What is the **depth**/**level** of J? 4

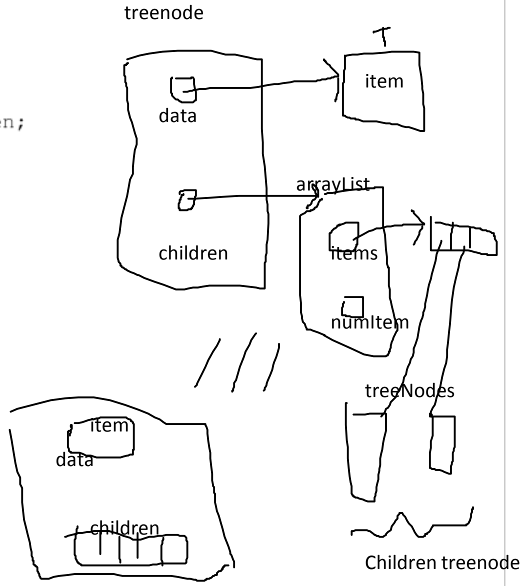
General Tree

- Each node can store an arbitrary number of children

The Tree Node Class:

```
Package class Treenode<T> {
   private T data;
   private ListADT<Treenode<T>> children;
   ...    getData, setData, getChildren
```

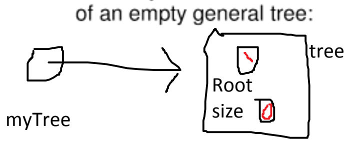
→ **Draw a picture** of the memory layout of a Treenode (assume an ArrayList is used for the ListADT):



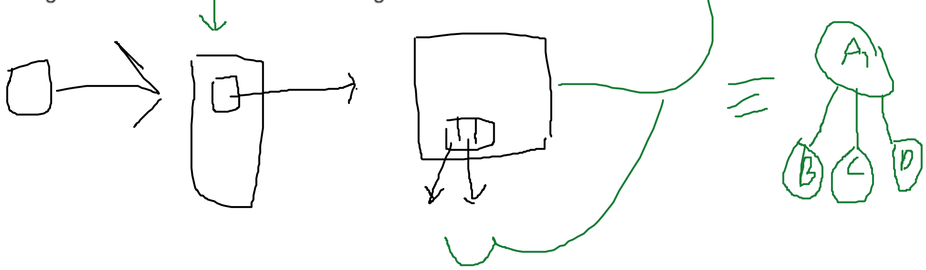
The Tree Class:

```
public class Tree<T> {
   private Treenode<T> root;
   private int size;

   public Tree() {
      root = null;
      size = 0;
   }
   ...
```

→ **Draw a picture** of the memory layout of an empty general tree:



→ **Draw a picture** of the memory layout of a general tree with a root node having 3 children:



Determining Height of a General Tree

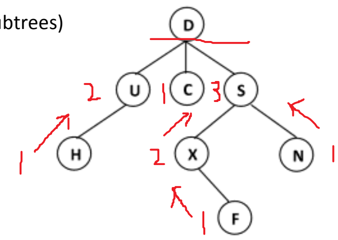
Recall the height of a tree is the length of a path from the root to the deepest leaf.

→ **Write a recursive definition** for the height of a general tree.

1+max(2,1,3) = 4

Height(t) = 0 if t is null
Height(t) = 1 if t is a leaf (has no children)

= 1 + max(height of children subtrees)

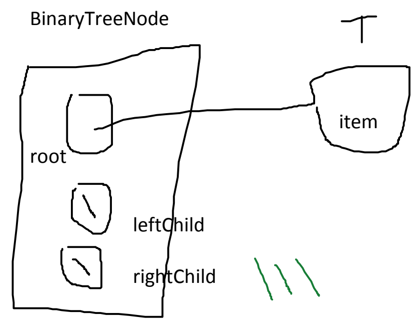


→ **Complete the recursive height method** based on the recursive definition.
Assume the method is added to a Tree class having a root instance variable.

```
public int height() {
        Return height(root);
        }

Private int height(Treenode<T>  t) {
        If (t == null) return 0;
        If (t.getChildren().isEmpty()) return 1;
        Int maxHt = 0;
        Iterator<Treenode<T>> itr  = t.getChildren().iterator();
        While (itr.hasNext()) {
                Int childHt = height(itr.next());
                If (childHt > maxHt) maxHt = childHt;
        }
        Return 1 + maxHt;
}
```

Binary Tree

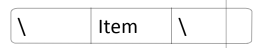
- Each node has at MOST 2 children



The Tree Node Class:

```
class BinaryTreenode<T> {
  private T data;
  private BinaryTreenode<T> leftChild;
  private BinaryTreenode<T> rightChild;

  public BinaryTreenode(T info) {
    data = info;
    leftChild  = null;
    rightChild = null;
  }
  ...
```

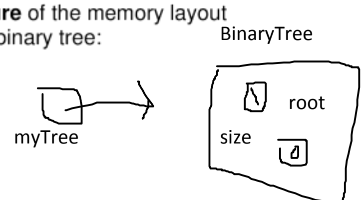
→ **Draw a picture** of the memory layout of a BinaryTreenode:



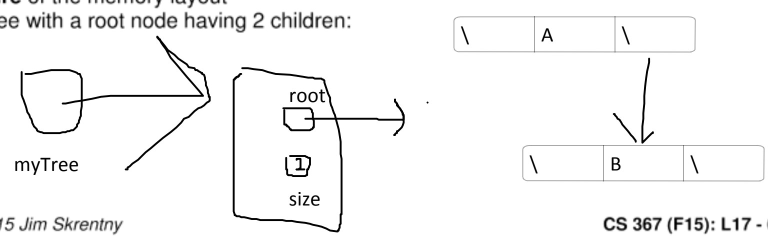
The Tree Class:

```
public class BinaryTree<T> {
  private BinaryTreenode<T> root;
  private int size;

  public BinaryTree() {
    root = null;
    size = 0;
  }
  ...
```

→ **Draw a picture** of the memory layout of an empty binary tree:



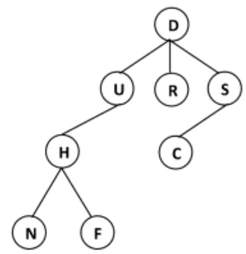
→ **Draw a picture** of the memory layout of a binary tree with a root node having 2 children:



Tree Traversals

Goal: visit every node in the tree exactly once

Visit means to do something with the node's data
(ex. Output)
Traversing means to step through the list of children from left to right



V = visit
C = transverse children

Level-order

D U R S H C N F
Top to bottom
Left to right on each level

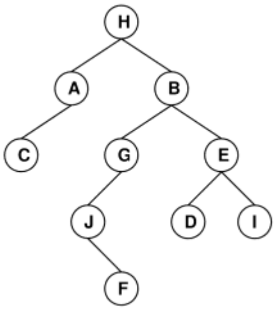
	General Tree	Binary Tree
Pre-order	V C D U H N F R S C	V L R
Post-order	C V	N F H U R C S D L R V
In-order	Not possible	L V R

*Use tree diagram for an execution tree trace

Practice - Tree Traversals

→ **List the nodes** using a pre-order traversal.

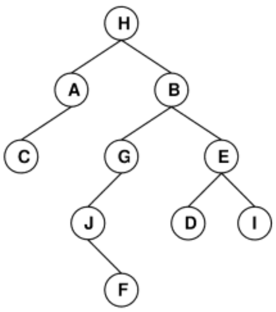
V L R



H A C …

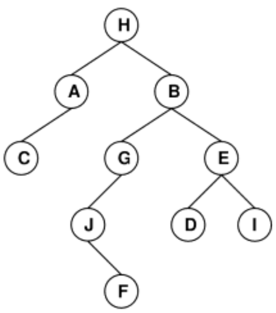
→ **List the nodes** using a post-order traversal.

L R V



…. I E B H

→ **List the nodes** using an in-order traversal.



C A H J F G B D E I

outlineW9T

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW9T.pdf>

CS 367 Announcements
Tuesday, November 3, 2015

Homework h7 due 10 pm, Friday, November 6th

Program p3 due 10 pm, Sunday, November 8th

Last Time

Categorizing ADTs Part 1
Tree Terms
General Trees
- implementing
- determining tree height

Binary Trees
- implementing

Today

Finish Traversals (last lecture)
Categorizing ADTs Part 2
Comparable Interface
Binary Search Tree (BST)
- BSTnodes
- BST class
- implementing print

CS Options/Courses

Next Time

Read: finish *Binary Search Trees*
Binary Search Tree (BST)
- implementing lookup, insert, delete
- complexities of BST methods

Balanced Search Trees
Classifying Binary Trees

Categorizing ADTs Part 2

Based on how the operations are done

Position oriented: Operations occur at a specified position in the ADT
ListADT, StackADT, QueueADT

Value Oriented: operations occur at a position in the ADT that's based on a key value in the item.
SortedListADT, MapADT

`Comparable` Interface

Use to determine the relative ordering of items

-in java.lang package
-specifies one method
Public int compareTo(T other)
-use a.compareTo(b)
Returns 0 if a = b (are the same)
< 0 if a < b (comes before)
> 0 if a > b (comes after)

- Implementation should be compatible with .equals()
  - a.compareTo(b) == 0
    - Then a.equals(b) == true
  - EXCEPT WHEN a.compareTo(null)
    - Throws NullPointerExceptio
    - a.equals(null) == true

Binary Search Tree (BST)

- Value oriented duplicate keys are not allowed

Goal
- Fast Lookup, insert, remove operations
- Combine speed of binary search on an Array of sorted values with the speed of linking/unlinking in a chain of nodes

Example **2 3 6 7 10 12 13 15 17 19 22 24 26 27 30**

```
            15
           /  \
          7    24
         / \   / \
        3  12  19  27
       / \ / \ / \ / \
      2  6 10 13 17 22 26 30
```

This is ideal shape
For BST storing these values
BUT BST doesn't guarantee this shape

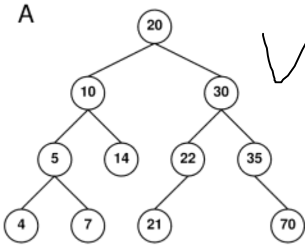
Ordering Constraint

BST Requires
-Binary nodes
-For each Node N having a key value K
L < K For every key L in the left subtree of K
R > K For every key in R in the right subtree of K

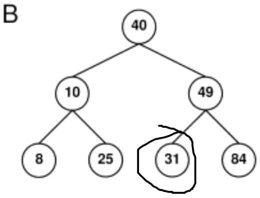
Practice - Identifying Binary Search Trees

→ **Identify which trees below are valid BSTs.**

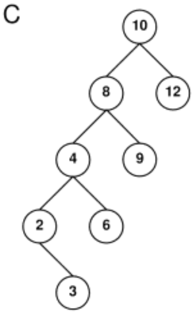
A



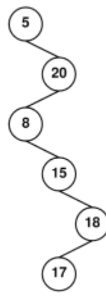
B



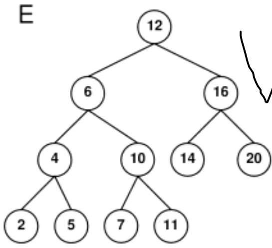
C



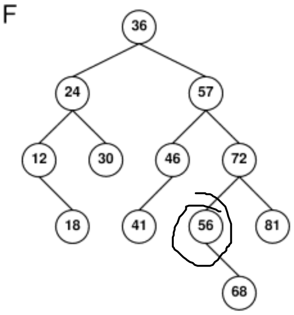
D



E



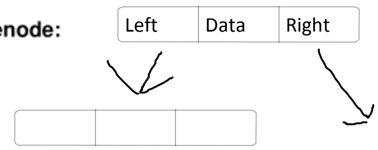
F



BSTnodeS

See readings for modifications to BSTnodes so that it also stores an associated value

→ Draw a picture of the memory layout of a Treenode:




```
class BSTnode<K> {

   private K key;
   private BSTnode<K> left, right;

   public BSTnode(K key, BSTnode<K> left, BSTnode<K> right) {
      this.key = key;
      this.left = left;
      this.right = right;
   }

   public K getKey() { return key; }
   public BSTnode<K> getLeft() { return left; }
   public BSTnode<K> getRight() { return right; }

   public void setKey(K newK) { key = newK; }
   public void setLeft(BSTnode<K> newL) { left = newL; }
   public void setRight(BSTnode<K> newR) { right = newR; }
}
```

BST Class

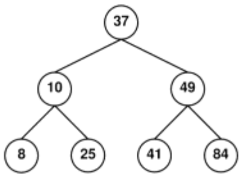
Import Java.io.*;

Restricts K type to only Comparable classes

```
public class BST<K extends Comparable<K>> {

   private BSTnode<K> root;

   public BST() { root = null; }

   public void insert(K key)
                 throws DuplicateException {
      Root = insert(root, key);

   }

   public void delete(K key) {
      Root = delete (root, key);
   }

   public boolean lookup(K key) {
      Return lookup(root, key);

   }

   public void print(PrintStream p) {
      Print(root, p);
   }

   //add helpers ...

      Private companion methods that recursively do the operation

}
```

Implementing `print`

→ **Write a recursive definition** to print a binary tree.



N refs treenodes
If n is null return — Base
Print (n's left subtree)
Output n's key value — Recursive
Print (n's right subtree)

→ **Complete the recursive print method based** on the recursive definition.

```
public void print(PrintStream p) {
   print(root, p);
}

private void print(BSTnode<K> n, PrintStream p) {
```

If (n == null) return;
Print(n.getleft(), p);
p.println(n.getkey());
Print(n.getRight(), p)

8 10 25 37 41 49 84

CS Options

CS Certificate

6 Courses
- Programming – CS 302
- Data Structures – CS 367
- 2 Courses >=400 level
- 2 Other CS Courses

CS Major

Basic CS
- Discrete Math – CS 240
- Programming + Data Structures – CS 302, CS 367
- Basic Systems – (CS 252), CS 352, CS 354

Math
- Calculus – MA 221, MA 222
- 2 Beyond Calc – MA 331/431 (probability), MA 340 (linear algebra)

Group A Theory
- Algorithms – CS 577

Group B Hardware/Software
- OS – CS 537

Group C Applications
- AI – CS 540

Group D Electives
- 2 CS Courses >=400 level

CS Double Major

- Must complete major requirements
- Easy for Computer Engineering Majors

CS Courses

Take Next

- CS 240 Introduction to Discrete Mathematics
- (CS 252) Introduction to Computer Engineering (prereq for CS 352)
- CS 352 Digital Systems Fundamentals
- CS 354 Machine Organization and Basic Systems (prereq for many group B)
- (CS 368) Learning a New Programming Language (C++ for CS 537)

>= 400 can take after CS 367

- CS 407 Foundations of Mobile Systems (spring, popular)
- CS 540 Introduction to Artificial Intelligence
- CS 570 Human Computer Interaction (spring)

>= 400 can take after CS 367 + Math

- CS 412 Introduction to Numerical Methods – MA 222 + MA 234 or CS 240
- CS 435 Introduction to Cryptography – MA 320 or MA 340
- CS 525 Linear Programming Methods – MA 320 or MA 340 or MA 443
- CS 533 Image Processing – MA 320 or MA 340 (fall)
- CS 559 Computer Graphics – MA 320 or MA 340
- CS 576 Introduction to Bioinformatics – MA 222 (fall)
- CS 577 Introduction to Algorithms – CS 240

outlineW9R

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW9R.pdf>

CS 367 Announcements
Thursday, November 5, 2015

Homework h7 due by 10 pm tomorrow, November 6th
- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h6.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Homework h8 assigned 11/10

Program p3 due 10 pm tomorrow, Sunday, November 8th
- submit java files to your `in` directory
- make sure to name your source files as specified in the submission section
- do not submit as a project/package/folder
- verify that you've submitted the correct files (ls, more, javac, java)
- partners? only ONE submits source but BOTH submit README.txt

Program p4 assigned Monday 11/11

Last Time

Finish Traversals
Categorizing ADTs Part 2
Comparable Interface
Binary Search Tree (BST)
- BSTnodes
- BST class
- implementing print

CS Options/Courses

Today

Binary Search Tree (BST)
- implementing print (from last time)

Binary Search Tree (BST)
- implementing lookup, insert, delete
- complexities of BST methods

Balanced Search Trees

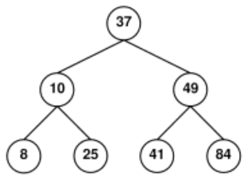
Next Time

Read: *Red Black Trees*
Classifying Binary Trees
Red Black Trees
- tree properties
- print, lookup
- insert
- cascaded fixing

Implementing `lookup`

Pseudo-Code Algorithm

```
private boolean lookup(BSTnode<K> n, K key) {
```



If n is null return false; //not found
If n's key equals key return true; //found
If key < n's key
    Return lookup (n's left subtree, key);
Else
    Return lookup(n's right subtree, key);

This recursive approach to searching down the tree is used in insert and delete
*lookup could be implemented using a loop

Implementing `insert`

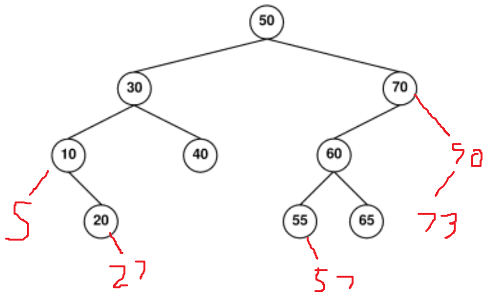
High-Level Algorithm

```
private BSTnode<K> insert(BSTnode<K> n, K key)
                                throws DuplicateException {
```

Search down tree as done in lookup
But
If n's key equals key throw duplicateExeception

When we get to the end of that tree where lookup would expect to find key we'll insert a new leaf node containing key

Practice - Inserting into a BST



→ **Insert 5, 27, 90, 73, 57 into the tree above.**

→ **What can you conclude about the shape of a BST when values are inserted in sorted order?**

15, 17, 22, 45, 97

15
\
17
\
22….

→ **Will you get that shape only if values are inserted in sorted order?**

*The shape of a binary search tree depends on the sequence of inserts and deletes

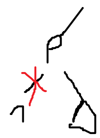
Implementing `delete`

High-Level Algorithm

```
private BSTnode<K> delete(BSTnode<K> n, K key) {
```

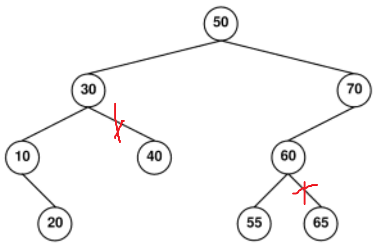
Search down tree as done in lookup
If n is null return null; //not found
If n's key equals key //found

Case 1: n has no children
Delete n by setting the appropriate child of n's parent P to null

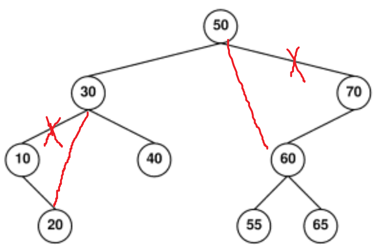


Case 2: n has 1 child
Delete n by setting the appropriate child of n's parent P to n's child c

Practice - Deleting from a BST

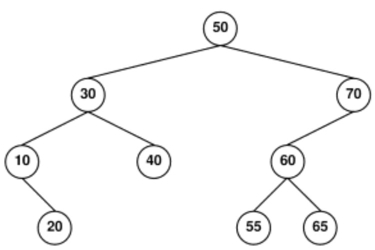


→ **Delete 40 and 65 from the tree above.**



50
30 60
20 40 55 65

→ **Delete 10 and 70 from the tree above and redraw the tree.**



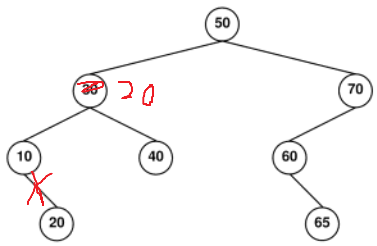
→ **How do you delete 50 or 30 from the tree above?**

Implementing `delete` (cont.)

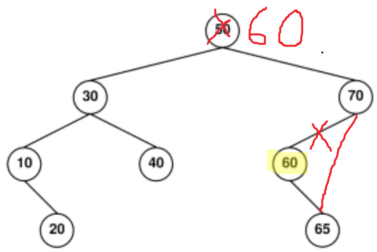
Case 3: n has 2 children
    not so easy since root/parents child reference can't hold on to both child subtrees
Solution:
- Find a replacement value check in either n's left or right subtree
- copy check into n's key
- recursively can delete check in n's subtree

2 replacements work
- In order predecessor
  - Largest value in left subtree
  - Step into left subtree then as far right as possible
- in order successor
  - smallest value in right subtree
  - step into right subtree then as far left as possible

Practice - Deleting from a BST



→ **Delete 30 from the tree above using the** Inorder predecessor.



→ **Delete 50 from the tree above using the** Inorder successor.

*Deleting the node with the replacement value will always be an easy case (0 or 1 child)
*either replacement works

Complexities of BST Methods

Problem size: N = Number of nodes/keys

print: O(N)

lookup:

insert:

delete:

O(H)
Where H is the BST height

O(logN)
Best Case
Good balanced shapes

O(N)
Worse case
Bad linear shape

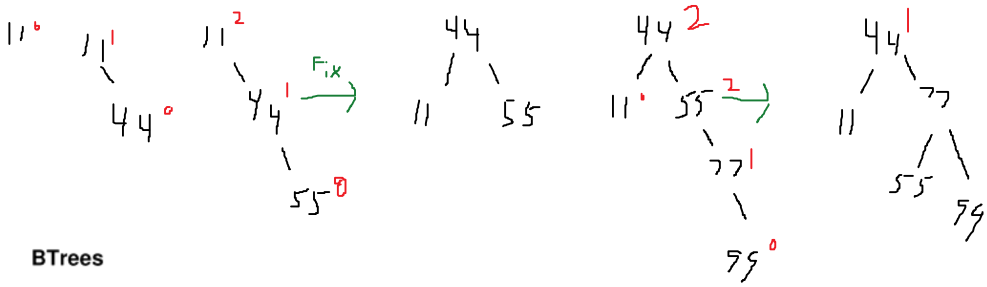
Balanced Search Trees

Goal: Keep height O(log N) where N is # of nodes so insert, lookup, delete are fast O(log N)

Idea: Make insert and delete restructure the tree when its shape goes out of balance

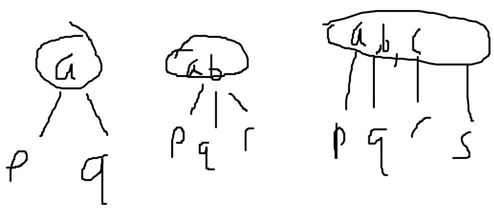
Detect imbalance and fix it

AVL Height balanced
Keep a balanced value in each node -1, 0 , 1
Detect: when nodes balance value +2 or -2
Fix: one technique called **rotation**



BTrees

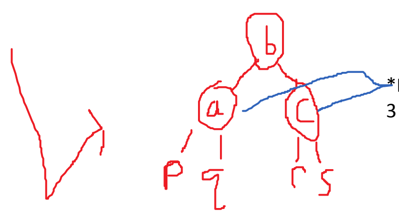
Relax Binary Tree Structure



2 node 3 node 4 nodes

BTree of order 3 (2-3 tree) uses only 2 and 3 nodes

Btree of order 4 (2-3-4 Tree) uses only 2,3, and 4 nodes



*Now these 2 nodes can grow to
3 nodes to accommodate the inserted key

For a 2-3-4 Tree
Detect: During insert look for 4 node

Fix: split 4 nodes

outlineW10T

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW10T.pdf>

CS 367 Announcements
Tuesday, November 10, 2015

Homework h8 due 10 pm, Friday, November 13th

Program p4 due 10 pm, Sunday, November 29th

Last Time

Binary Search Tree (BST)

- implementing print
- implementing lookup, insert, delete
- complexities of BST methods

Balanced Search Trees

Today

Balanced Search Trees (from last time)
Classifying Binary Trees
Red-Black Trees

- tree properties
- print, lookup
- insert

Next Time

Read: *Priority Queues*
Red-Black Trees

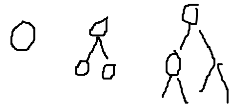
- cascaded fixing
- complexity

Priority Queue ADT

- concept
- operations
- implementation options
- Heap Data Structure

Classifying Binary Trees

Full "No missing nodes"

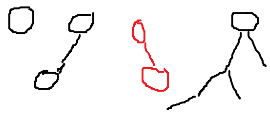


All leaves are at same depth
All non-leaf (interior) node must have 2 children

Height H
Nodes $N = 2^H - 1$
$N + 1 = 2^H$
$Log_2(N+1) = H$
$O(Log_2(N+1))$
$O(Log(N))$

Complete Priority Queues

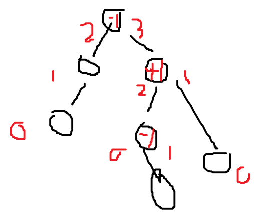
Full to depth H - 1
Depth H is filled from left to right



Full tree is complete

Height-balanced (AVL Tree)

For each node the difference in height of its left and right substrees is at most 1



Full and complete trees are height tree

Balanced (Red-Black)

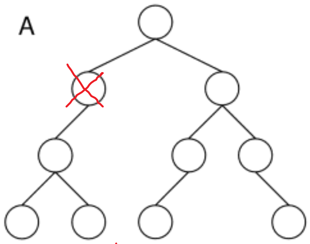
A tree having a height of O(Log(N)) where N is the number of nodes

Full, complete, and height balanced trees are balanced

Practice - Classifying Binary Trees

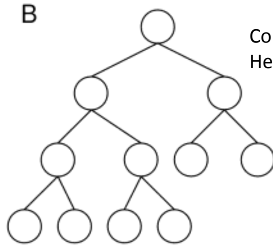
→ Identify which trees below are full, complete and/or height balanced.

A



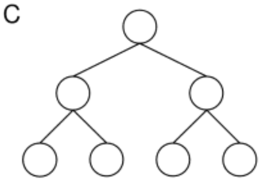
No

B



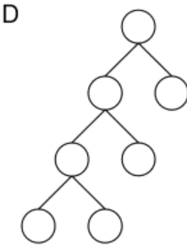
Complete
Height Balanced

C



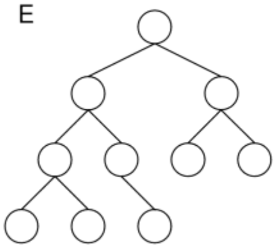
Full, Complete, Height Balanced

D



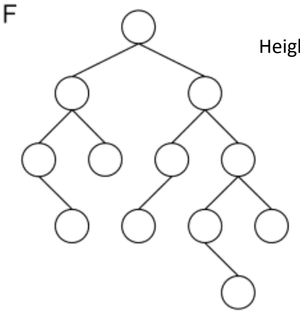
Tending to be stacky O(N)

E



Height balanced

F

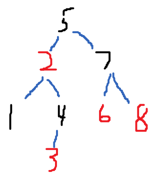


Height Balanced

Red-Black Trees (RBT)

RBT: Binary Search tree that is modified to keep a balanced shape
Height O(Log(N))

Example:



Red-Black Tree Properties

root property Root Node must be Black

red property Red Nodes must have black children

black property Every path from the root to a leaf must have the same number of black nodes

Red-Black Tree Operations

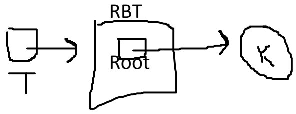
print
lookup Same as Binary Search Tree

insert
delete Similar to Binary Search Tree but with rebalancing code

Inserting into a Red-Black Tree

Goal: insert key value K into red-black tree T
and Maintain Red-Black Tree properties.

If T is Empty Add a black leaf



Except for root
All new nodes added a leaf nodes

If T is Non-Empty

- step down tree as done for BST
- add a leaf node containing K as done for BST, and Color it Red
- Restore RBT properties if needed

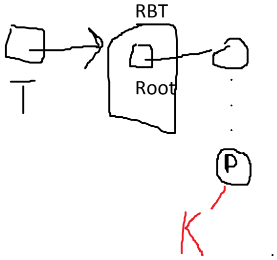
→ Which of the properties might be violated as a result of inserting a red leaf node?

~~root property~~ A non-empty tree already has a black root

~~black property~~ adding a red node doesn't affect the number of black nodes

red property Adding a red node will violate the red property if the parent is red
*use RPV (Red property violation) to detect imbalance

Non-Empty Case 1: K's parent P is black

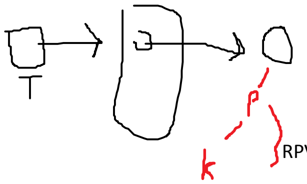


No RPV so done inserting

Mirror Images wont be shown

Non-Empty Case 2

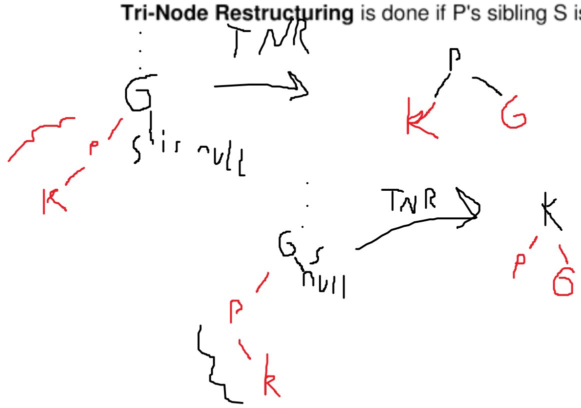
Non-Empty Case 2: K's parent P is red



RPV   Fix depends on the parent's sibling

Fixing an RBT

Tri-Node Restructuring is done if P's sibling S is null

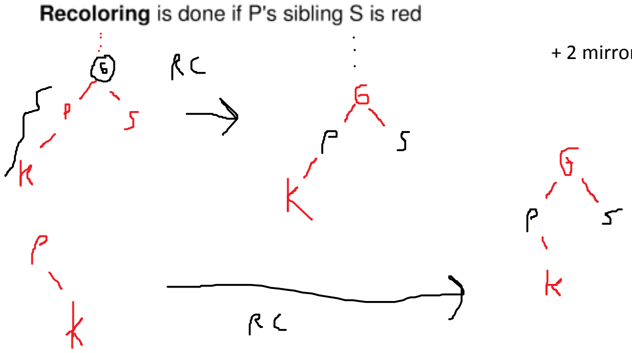


+ 2 mirror images

1) Middle value becomes black parent
2) Smallest value becomes red left child
3) Largest value becomes red right child

*Grandparent might be root

Recoloring is done if P's sibling S is red

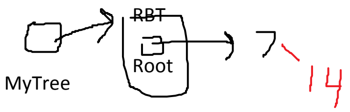


+ 2 mirror images

1) Change parent and sibling to black
2) If G is root, then insert is done
   Else, change G to red

Practice

→ 1. Starting with an empty RBT, show the RBT that results from inserting 7 and 14.

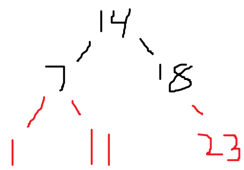


→ 2. Redraw the tree from above and then show the result from inserting 18.

→ 3. Redraw the tree from above and then show the result from inserting 23.

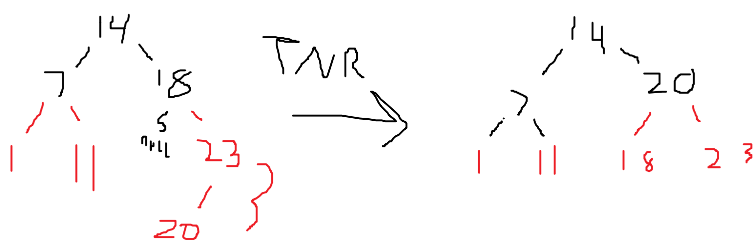
G is root so no color change for G

→ 4. Redraw the tree from above and then show the result from inserting 1 and 11.



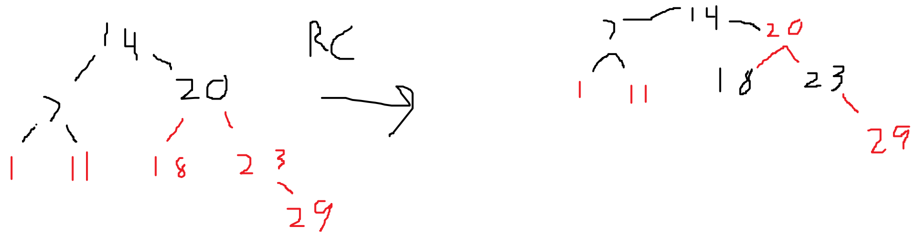
Some inserts require no fixing

→ 5. Redraw the tree from above and then show the result from inserting 20.

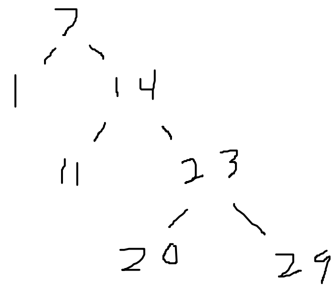


More Practice!

→ 6. Redraw the tree from the previous page and then show the result from inserting 29.



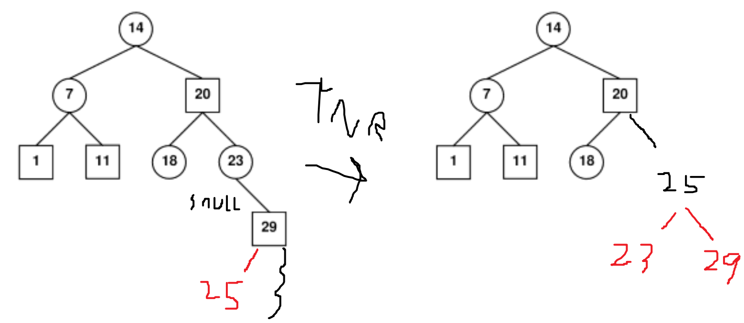
→ 7. Insert the same list of values into an empty BST: 7, 14, 18, 23, 1, 11, 20, 29



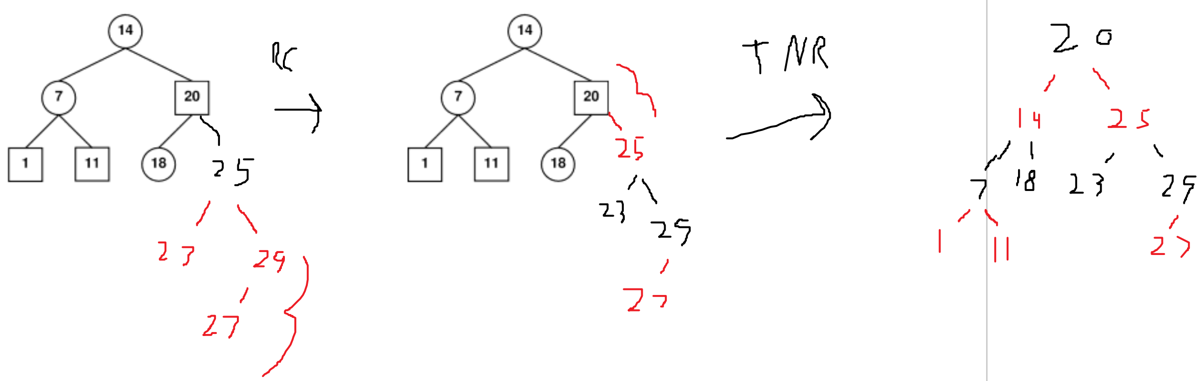
→ What does this demonstrate about the differences between a BST and RBT?

More Practice?

→ 8. Show the result from inserting 25 in the RBT below.



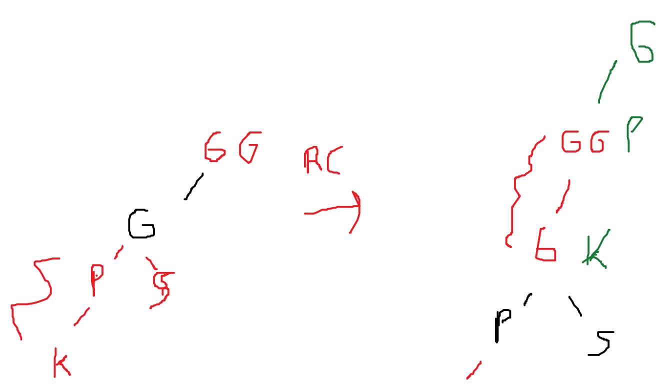
→ 9. Redraw the tree from above and then show the result from inserting 27.



CASCADING FIXES

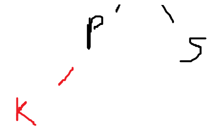
Fixing RBY updated

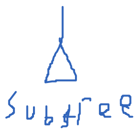
Recoloring is done if P's sibling S is red



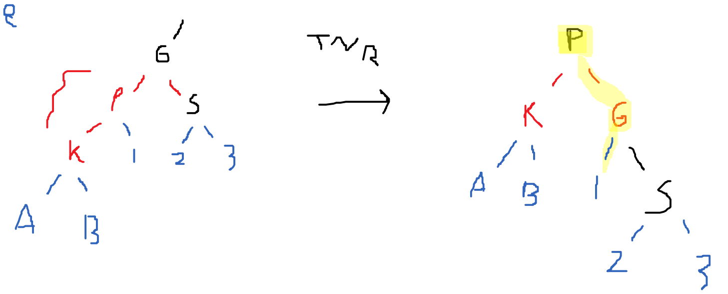
1) Change P and S to black
2) If G is Root, DONE
3) Else, change G to red
   a. And if GG is black DONE
4) Else, RBV G and GG which will fix

2) If G is Root, DONE
3) Else, change G to red
    a. And if GG is black DONE
4) Else, RPV G and GG which will fix recursively starting at G

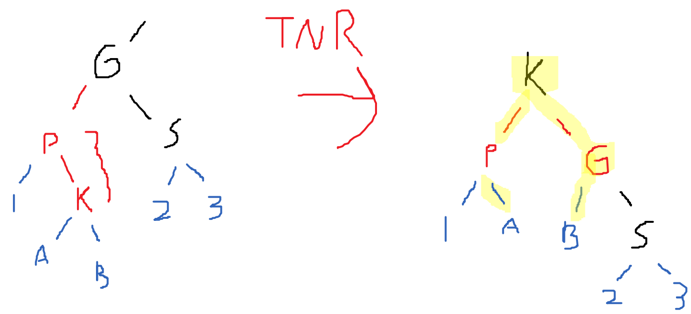




Tri Node Restructuring is done if parent P's sibling S is null <u>or Black</u>



Changes
2 colors
2 links



Change
2 colors
4 links

RBT Complexity

Print = same as BST O(N)

Lookup = same code BST, but worse case (Log(N)) since RBT maintains balance

Insert = 1) insert new red leaf node , in worse case is O(Log(N)) since RBT height is guaranteed to be O(Log(N))
2) restoring RBT properties in worse case recoloring cascades back to Root O(Log(N))
Overall - 1 + 2 = O(Log(N) + O(Log(N)) = O(2Log(N)) = **O(Log(N))**

outlineW10R

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW10R.pdf>

CS 367 Announcements
Thursday, November 12, 2015

Homework h8 due 10 pm, Friday, November 13th

Program p4 due 10 pm, Sunday, November 29th

Last Time

Balanced Search Trees
Classifying Binary Trees
Red-Black Trees
- tree properties
- print, lookup
- insert

Today

Red-Black Trees (from last time)
- cascaded fixing
- complexity

Priority Queue ADT
- concept
- operations
- implementation options

Heap Data Structure

Next Time

Read: start *Hashing*
Heap Data Structure
- insert
- removeMax

Hashing
- terminology
- designing a good hash function

Priority Queue ADT

Priorities

Used to store items by their importance
-Each item stores a number for its priority
-Duplicate priorities **are** allowed
-Hightest priority can be either the smallest or largest number

Concept Priority Queue is an ADT where items are removed in order of their priorities

goal: Fast access O(1) to highest priority

Operations

Void insert(comparable item)

Comparable getMax()   O(1)

Comparable removeMax()

Boolean isEmpty()

Options for Implementing a Priority Queue ADT

data structure	`insert`	`removeMax`
unordered array	O(1) At rear w/ shadow	O(N) worse case Linear Search If removing first item - don't shift fill gap with last item
ordered array	O(N) worse case = O(Log(N)) Binary Search+O(N) shift	O(1) Max priority at rear
unordered chain of nodes	O(1) Insert at Head	O(N) worse case linear search
ordered chain of nodes	O(N) worse case = O(N) linear search + O(1) linking	O(1) Max priority at head
HEAP	O(Log(N))	O(Log(N))

Implementing a Priority Queue ADT using a Heap

Heap

min heap Smallest value is highest priority
max heap Largest value is highest priorty

Shape Constraint

Complete binary tree
1) Full from root to second last level
2) Last level is filled from left to right

Ordering Constraint (max)

P
≤ P ≤ P

Top
↓
Bottom

For every node N, N's priority P is >= the priorities of N's descendants

Implementing Heaps

Root is at index 1 (not using element @ index 0) for each node N at index i
N's left child is at index 2*I
N's right child is at index 2*I + 1
N's Parent is at i/2 Integer divison

Max Heap Example:

	1	2	3	4	5	6	7	8	9	10
X	56	42	37	38	14	12	26	29	16	8

→ Draw the corresponding binary tree:

56
42 37
38 14 12 26
29 16 8 .

outlineW11T

CS 367 Announcements
Tuesday, November 17, 2015

Midterm Exam 2
- Tuesday, November 24th, 5:00 pm
- Exam information posted
- Sample questions on Learn@UW
- UW IDs are required

Homework h9 due 10 pm, Friday, November 20th

Program p4 due 10 pm, Sunday, November 29th

Last Time
Red-Black Trees
- cascaded fixing
- complexity

Priority Queue ADT
- concept
- operations
- implementation options

Heap Data Structure

Today
Heap Data Structure
- insert
- removeMax

Hashing
- terminology
- designing a good hash function

Next Time
Read: finish *Hashing*
Hashing
- choosing table size
- expanding a hash table
- handling collisions

Inserting into a Max Heap

Algorithm

1) Put new item in next free element - O(1)
2) Restore heap ordering constraint
   a. Reheapify by Swapping new item with its smaller parent

Given the following max heap:

	64	52	35 57	46	17 36	~~15~~ 35	34	12	23	14	~~36~~ 17	15

→ Show the heap after inserting 36:

	64	52	35	46	36	15	34	12	23	14	17	

→ Show the heap after inserting 57:

	64	52	57	46	36	35	34	12	23	14	17	15

64
52 35
46 17 15 34
12 23 14 36 .

64
52 35
46 36 15 34
12 23 14 17 .

64
52 57
46 36 35 34 .
12 23 14 17 15 .

Inserting into a Max Heap (cont.)

PriorityQueue Class Instance Variables:

```
private Comparable[] items;
private int nextLoc;
```

Pseudo-code

```
public void insert(Comparable data) {
```

O(1)

If (data ==null) throw exception
//1.
 if (array is full) expand O(1)
Items[nextLoc] = data;
nextLoc++;
//2
Int child - nextLoc - 1;
Boolean done = false;
While (!done) {
    Int parent = child / 2;
    If (parent == 0) done = true;
    Else if (items[child].compareTo(items[parent]) <= 0) done = true;
    Else {
        Swap child and parent items
        Child = parent;
    }
}

O(Log(N))

Complexity

Removing from a Max Heap

Algorithm

1. Remove root item by replacing it with last item in array - O(1)
2. Restore heap ordering constraint
    a. Reheapify by swapping down with largest child

Heap after adding 36 and 57:

	~~64~~ ~~15~~ 57	52	57	46	36	35	34	12	23	14	17	15

→ What will the heap look like after doing a removeMax?

	57	52	35	46	36	15	34	12	23	14	17	

→ What will the heap look like after doing another removeMax?

	52	46	35	23	36	15	34	12	17	14		

Complexity

```
              64                                57
         52         57                    52          35
     46     36     35  34  .          46     36     15  34  .
  12  23  14 17   15          .   12  23  14 17               .
```