

## CS430 Lecture 8 Activities - KEY

### Opening Questions

1. Order Statistics: Select the  $i$ th smallest of  $n$  elements (the element with rank  $i$ )

$i = 1$ : minimum;

$i = n$ : maximum;

$i = \text{floor}((n+1)/2)$  or  $\text{ceiling}((n+1)/2)$ : median

How fast can we solve the problem for various  $i$  values?

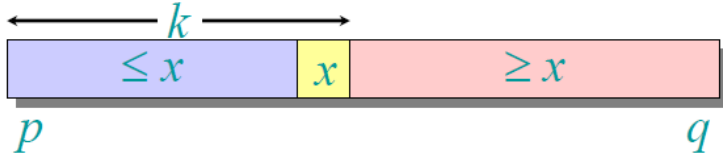
Min/max:  $O(n)$

General  $i$ :  $O(n \log n)$  by sorting

We will see how to do it in  $O(n)$  time

### Randomized Algorithm for finding the $i$ th Element

1. Think about partition (with a random choice of the pivot) from quicksort. Can you think of a way to use that and comparing the final location of the pivot to  $i$ , and then divide and conquer?



If  $i < k$ , recurse on the left; If  $i > k$ , recurse on the right; Otherwise, output  $x$

**RAND-SELECT**( $A, p, r, i$ )

if  $p = r$  then return  $A[p]$

$q \leftarrow \text{RAND-PARTITION}(A, p, r)$

$k \leftarrow q - p + 1$

if  $i = k$  then return  $A[q]$

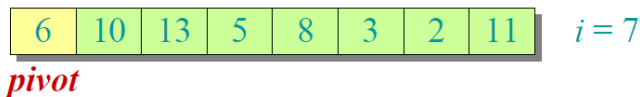
if  $i < k$  then return **RAND-SELECT**( $A, p, q - 1, i$ )

else return **RAND-SELECT**( $A, q + 1, r, i - k$ )

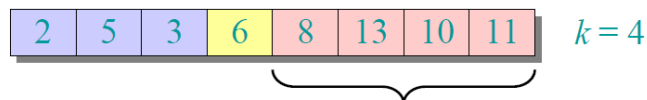
2. Demonstrate on this array to find  $i$ th smallest element

6	10	13	5	8	3	2	11
---	----	----	---	---	---	---	----

Select the  $i = 7$ th smallest:



Partition:



Select the  $7 - 4 = 3$ rd smallest recursively.

3. What is the worst case running time if you find the  $i$ th smallest element?

$T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$  if unlucky (solved by master method)

$T(n) = T(9n/10) + \Theta(n) = \Theta(n)$  if lucky (solved by master method)

Is there an algorithm to find the  $i$ th smallest element that runs in linear time in the worst case?

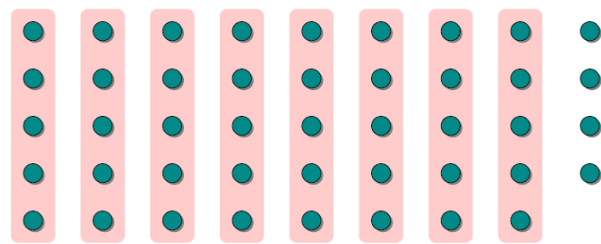
SELECT( $i, n$ )

1. Divide the  $n$  elements into groups of 5. Find the median of each 5-element group by hand.
2. Recursively SELECT the median  $x$  of the  $\lfloor n/5 \rfloor$  group medians to be the pivot.
3. Partition around the pivot  $x$ . Let  $k = \text{rank}(x)$ .
4. **if**  $i = k$  **then return**  $x$   
     **elseif**  $i < k$   
         **then** recursively SELECT the  $i$ th smallest element in the lower part  
     **else** recursively SELECT the  $(i-k)$ th smallest element in the upper part

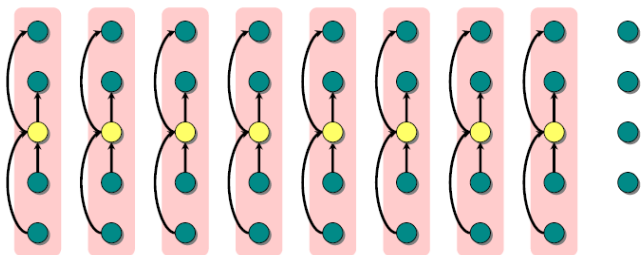
## Choosing the pivot



## Choosing the pivot

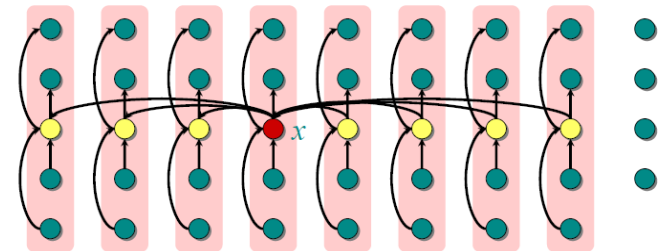


1. Divide the  $n$  elements into groups of 5.



1. Divide the  $n$  elements into groups of 5. Find the median of each 5-element group by rote.

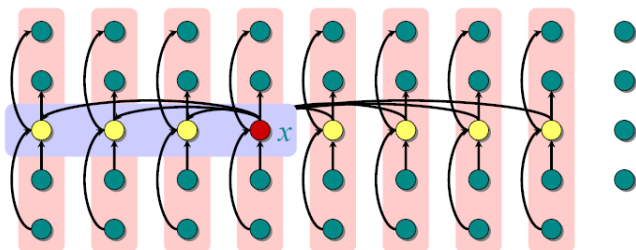
lesser  
↑  
greater



1. Divide the  $n$  elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median  $x$  of the  $\lfloor n/5 \rfloor$  group medians to be the pivot.

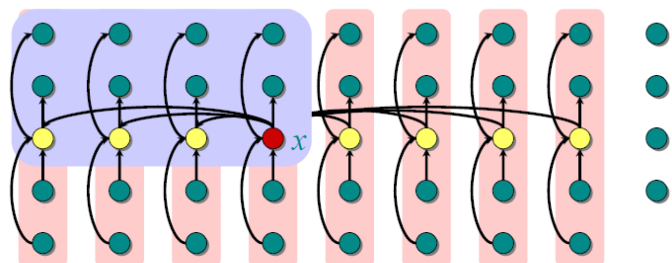
lesser  
↑  
greater

## Analysis



At least half the group medians are  $\leq x$ , which is at least  $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$  group medians.

## Analysis

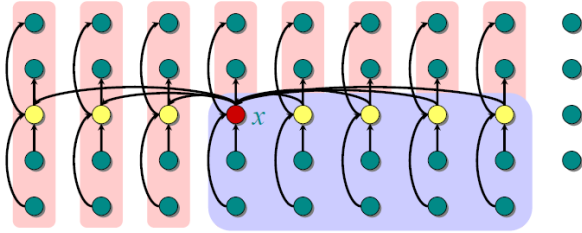


At least half the group medians are  $\leq x$ , which is at least  $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$  group medians.

- Therefore, at least  $3 \lfloor n/10 \rfloor$  elements are  $\leq x$ .

## Developing the recurrence

### Analysis



At least half the group medians are  $\leq x$ , which is at least  $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$  group medians.

- Therefore, at least  $3 \lfloor n/10 \rfloor$  elements are  $\leq x$ .
- Similarly, at least  $3 \lfloor n/10 \rfloor$  elements are  $\geq x$ .

$$\begin{array}{lcl}
 T(n) & \text{SELECT}(i, n) & \\
 \Theta(n) & \left\{ \begin{array}{l} 1. \text{ Divide the } n \text{ elements into groups of } 5. \text{ Find} \\ \text{the median of each } 5\text{-element group by rote.} \end{array} \right. & \\
 T(n/5) & \left\{ \begin{array}{l} 2. \text{ Recursively SELECT the median } x \text{ of the } \lfloor n/5 \rfloor \\ \text{group medians to be the pivot.} \end{array} \right. & \\
 \Theta(n) & \left\{ \begin{array}{l} 3. \text{ Partition around the pivot } x. \text{ Let } k = \text{rank}(x). \\ 4. \text{ if } i = k \text{ then return } x \\ \text{elseif } i < k \\ \text{then recursively SELECT the } i\text{th} \\ \text{smallest element in the lower part} \\ \text{else recursively SELECT the } (i-k)\text{th} \\ \text{smallest element in the upper part} \end{array} \right. & \\
 T(7n/10) & & 
 \end{array}$$

## Solving the recurrence

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n\right) + \Theta(n)$$

**Substitution:**

$$T(n) \leq cn$$

$$T(n) \leq \frac{1}{5}cn + \frac{7}{10}cn + \Theta(n)$$

$$= \frac{18}{20}cn + \Theta(n)$$

$$= cn - \left(\frac{2}{20}cn - \Theta(n)\right)$$

$$\leq cn$$

if  $c$  is chosen large enough to handle the  $\Theta(n)$ .

In practice, this algorithm runs slowly, because the constant in front of  $n$  is large.

Would we use this approach to find the median to partition around in Quicksort, and achieve in worst-case Theta ( $n \log n$ ) time?

**Unfortunately, the additional labor involved causes a very large constant factor “hidden” in the Theta notation, yielding an algorithm that is not useful in practice. Nevertheless, this approach to worst-case linear-time selection is useful and has many applications.**

## CS430 Lecture 9 Activities – KEY

### Opening Questions

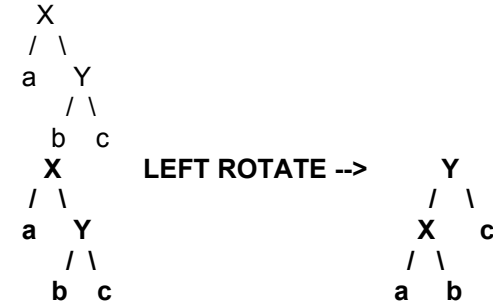
1. In your own words explain how you insert a new key in a binary search tree.

**Start at root, compare new key to root, go left if smaller, go right if bigger. Then do that again until you get to a leaf to insert it.**

2. Give an example of a series of 5 keys inserted one at a time into a binary search tree that will yield a tree of height 5.

**A B C D E**

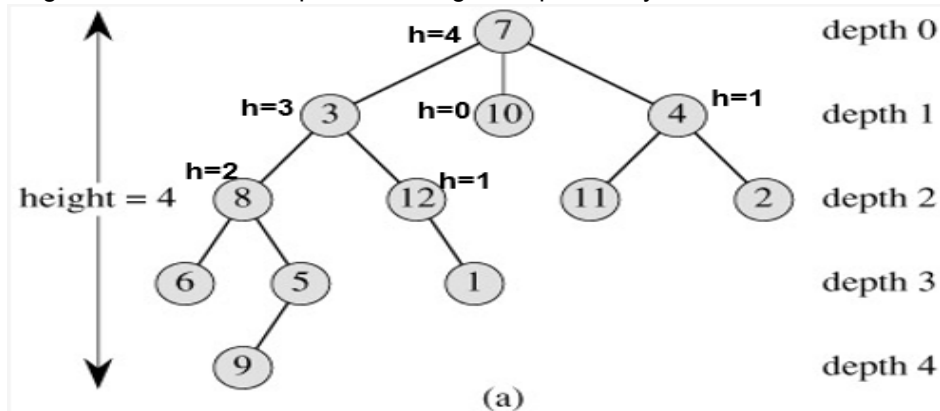
3. If we do a left rotate on node X below, explain which left and/or right child links need to be changed



**b used to be leftchild of Y, now it rightchild of X**  
**Y used to be rightchild of X, now X is leftchild of Y**

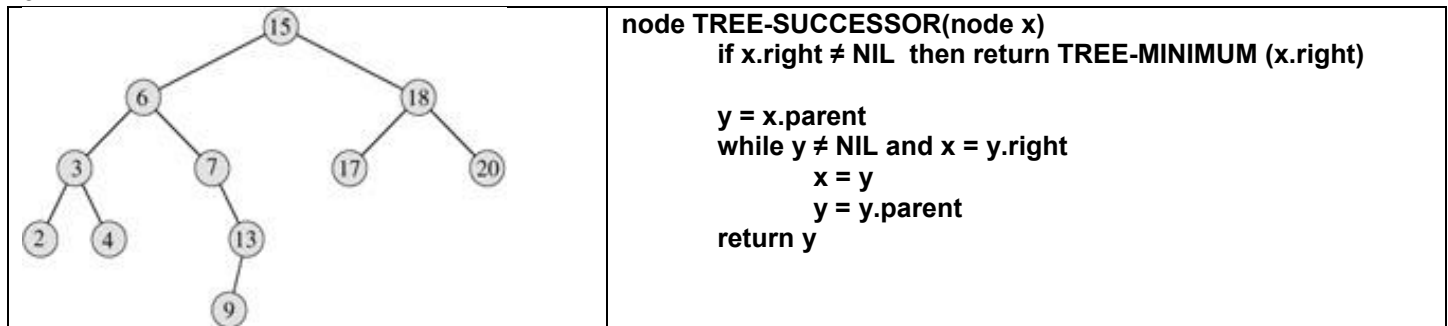
### Tree depth vs Tree height

The length of the path from the root  $r$  to a node  $x$  is the depth of  $x$  in  $T$ . The height of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf, and the height of a tree is the height of its root. The height of a tree is also equal to the largest depth of any node in the tree.

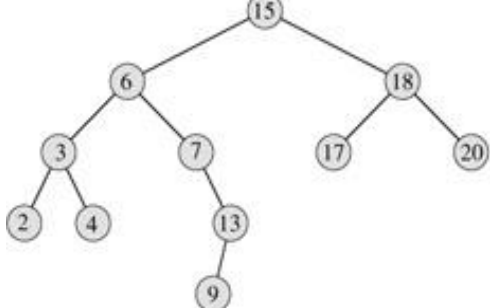


### BST Operations

1. Write pseudocode for BST Successor (or Predecessor). Demonstrate on the below tree from node 15 and then node 13.



2. Write pseudocode for BST Insert. Demonstrate on the below tree to insert 5 and then 19.

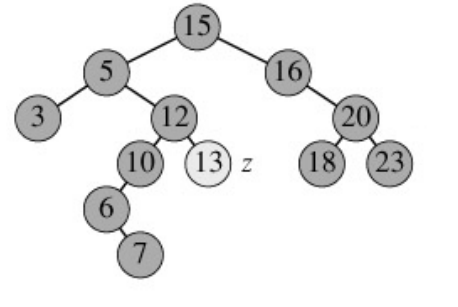
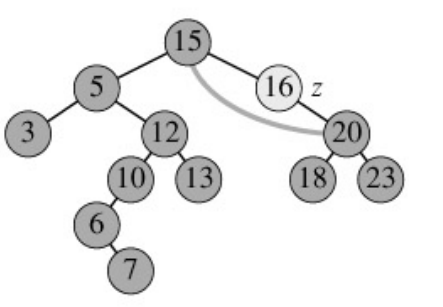
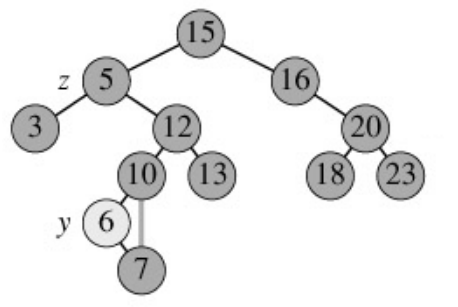
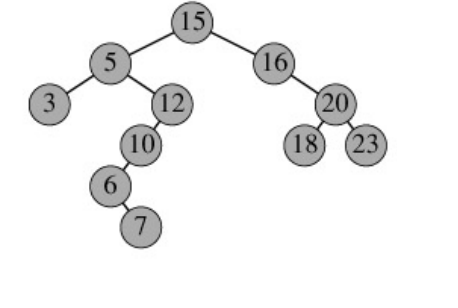
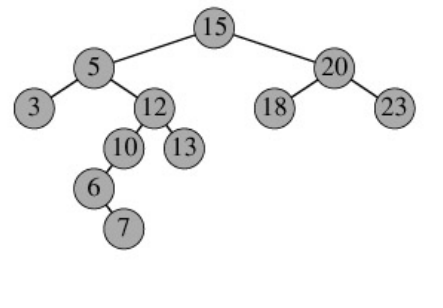
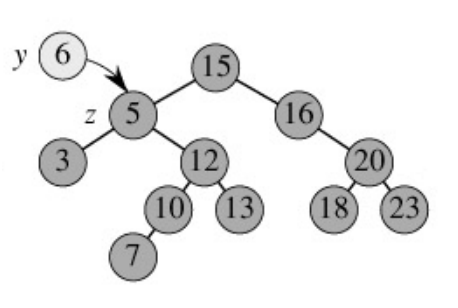


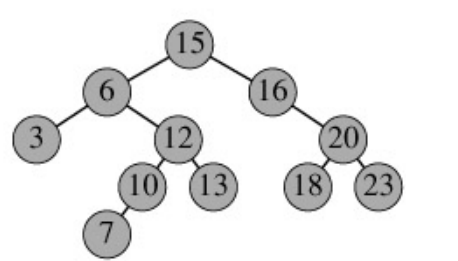
	<pre> <b>node INSERT</b>(node x, int key)  // x is root in initial call   if x == NIL     x = new node(key)     return x;   else     if (key &lt; x.key)       x.left = INSERT(x.left, key)     else if (key &gt; x.key)       x.right = INSERT(x.right, key)     return x         </pre>
---	---

3. What are the three possible cases when deleting a node from a BST?

**The node to be deleted has no children – delete the node**

**The node to be deleted has one child – make the parent of the node point to the child of the node**

**The node to be deleted has two children – swap the node with its predecessor (or successor), then delete the predecessor (or successor)**

No Children	One Child	Two Children
		
		
		

4. Write pseudocode for BST delete (assume you already have a pointer to the node)

**node TREE-DELETE(T, z) // assumes z points to node**

```

if left[z] = NIL or right[z] = NIL
  then y = z
  else y = TREE-SUCCESSOR(z) // O(h)
if left[y] ≠ NIL
  then x = left[y]
  else x = right[y]
if x ≠ NIL
  then p[x] = p[y]
if p[y] = NIL

```

```

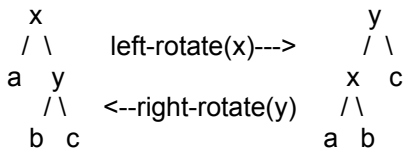
    then root[T] = x
    else if y = left[p[y]]
        then left[p[y]] = x
        else right[p[y]] = x
if y ≠ z
    then key[z] = key[y]
    copy y's satellite data into z
return y

```

### BST Rotations

Local operation in a search tree that maintains the BST property and possibly alters the height of the BST.

x and y are nodes; a, b, c are sub trees



5. Write pseudocode for LeftRotate (or RightRotate). What is the worst case runtime?

```

LEFT-ROTATE(T, x)
y ← right[x] // Set y
right[x] ← left[y]
//Turn y's left subtree into x's right subtree
p[left[y]] ← x
p[y] ← p[x] //Link x's parent to y
if p[x] = nil[T]
    root[T] ← y
else
    if x = left[p[x]]
        left[p[x]] ← y
    else right[p[x]] ← y
left[y] ← x // Put x on y's left
p[x] ← y

```

O(1)

## CS430 Lecture 10 Activities - KEY

### Opening Questions

1. What is the main problem with Binary Search Trees that Red-Black Trees correct? Explain briefly (2-3 sentences) how Red-Black Trees correct this problem with Binary Search Trees.

**Height of Binary Search Tree is not ensured to be  $O(\lg n)$  because BST not ensured to be balanced, depending on add/delete of keys order. Red-Black trees maintain more-or-less balance and therefore dictionary operations are  $O(\lg n)$**

**Red-Black trees maintain more-or-less balance by requiring patterns in the coloring of nodes red or black that result in no path from root to leaf being more than twice as long as any other path. Rotations/recolorings within the tree are used to maintain this balance.**

2. For the balanced binary search trees, why is it important that we can show that a rotation at a node in a is  $O(1)$  (i.e. not dependent on the size the BST)

**There certainly is more work to maintain a BST balanced, using rotations can achieve this. But if each rotation runs in constant time, then we can maintain the balance of the BST in  $O(\lg n)$  because any insert at most traverses the height of the tree to insert an item**

### Red-Black Trees

#### Red-Black Properties

1. Every node is colored either red or black
2. The root is black
3. Every null pointer descending from a leaf is considered to be a null black leaf node
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes (black height)

black-height of a node  $bh(x)$  - The number of black nodes on any path from, but not including, a node "x" down to a null black leaf node

1. If a node x has  $bh(x)=3$ , what is its largest and smallest possible height (distance to farthest leaf) in the BST?

**Largest height is 6  $x=B R B R B R B(\text{nil leaf})$**

**Smallest height is 3  $x=B B B B(\text{nil leaf})$**

2. Prove using induction and red-black tree properties. A red-black tree with n internal nodes (n key values) has height at most  $2\lg(n+1)$

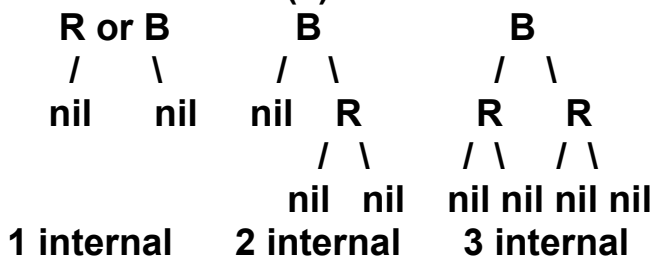
Part A - First show the sub-tree rooted at node "x" has at least  $2^{bh(x)}-1$  internal nodes. Use induction

**Base case -  $bh(x)=0$**

**x is a nil leaf node; no key**

**$2^{bh(x)}-1 = 2^0-1 = 0$  internal nodes minimum**

**Base Case -  $bh(x)=1$**



**$2^{bh(x)}-1 = 2^1-1 = 1$  internal nodes minimum**

**Now consider node x with  $bh(x)=k$**

**All paths to leafs have k nodes black. What about the children of node x?**

If child(x) is red, then  $bh(x) = bh(\text{child}(x)) = k$

If child(x) is black, then  $bh(x) - 1 = bh(\text{child}(x)) = k - 1$

Assume what we are trying to prove is true for a child of x, that it has at least  $2^{bh(x)} - 1$  internal nodes,  $2^{k-1} - 1$  at least (if the child is red it has more)

So if we add up the “at least” internal nodes for both children of x and add in x we get . .

Internal nodes of x

$$\geq (2^{k-1} - 1) + (2^{k-1} - 1) + 1$$

$$\geq 2 * 2^{k-1} - 2 + 1$$

Internal nodes of x  $\geq 2^k - 1$       If  $bh(x) = k$

Part B – Let “h” be height of R-B Tree, by property 4 at least half the nodes on path from root to leaf are black

$$bh(\text{root}) \geq h/2$$

Use that and part A to show  $h \leq 2\log(n+1)$

If  $bh(x) = k$ , Internal nodes of x  $\geq 2^k - 1$

$$2^{bh(\text{root})} - 1 \leq n$$

$$2^{h/2} - 1 \leq n$$

$$2^{h/2} \leq n+1 \text{ (take } \log_2 \text{ of both sides)}$$

$$h/2 \leq \log(n+1)$$

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

3. Which BST operations change for a red-black tree and which do not change? What do the operations that change need to be aware of and why?

search, insert, delete, predecessor, successor, minimum, maximum, rotations

**Red-Black Tree search, predecessor, successor, minimum, maximum operations identical to BST (ignore the color of a node)**

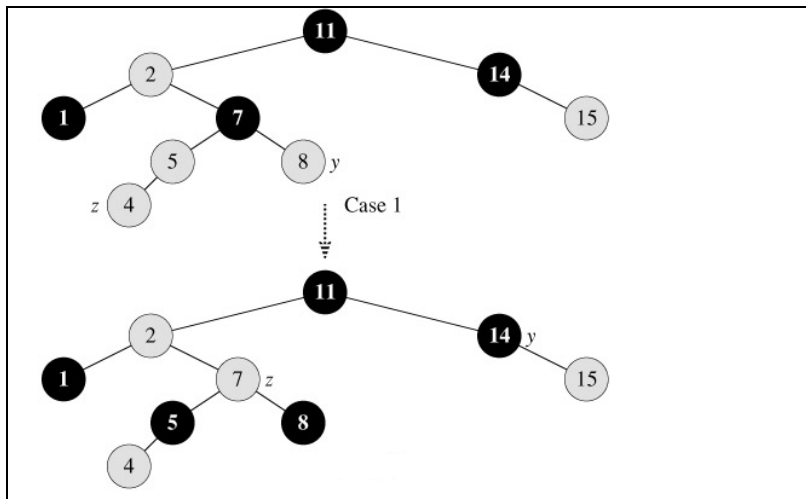
**Red-Black Tree insert and delete and rotations, since they modify the tree, require changing the colors of some nodes and rotations in the tree to maintain the Red-Black Properties and the approximate balanced property of the tree.**

Red-Black Tree Insert - Similar to BST insert, assume we start with a valid red-black tree.

1. Locate leaf position to insert new node
2. Color new node red and create 2 new black nil leaves below newly inserted red node
3. If parent of new insert was \_\_\_\_\_ (fill in the blank, black or red), then done. ELSE procedure to recolor nodes and perform rotations to maintain red-black properties.

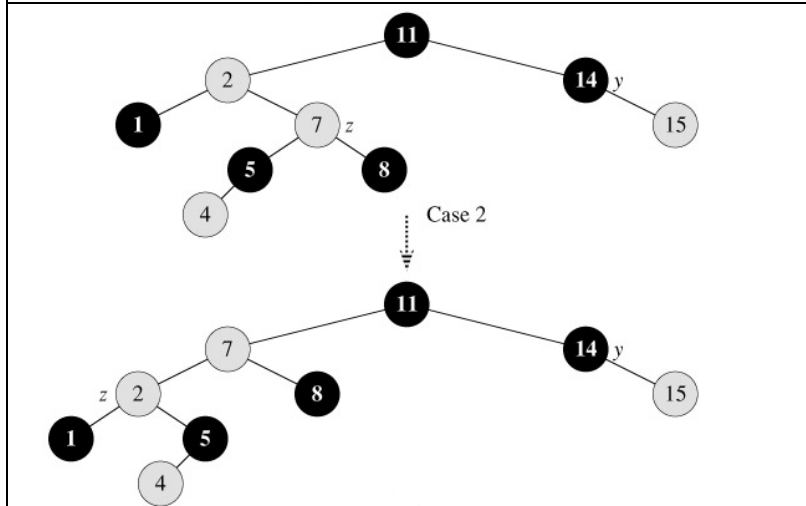
There are three cases if R-B Property #4 broken when insert a red node "Z" (or changed color of a node to red) and its parent is also red.





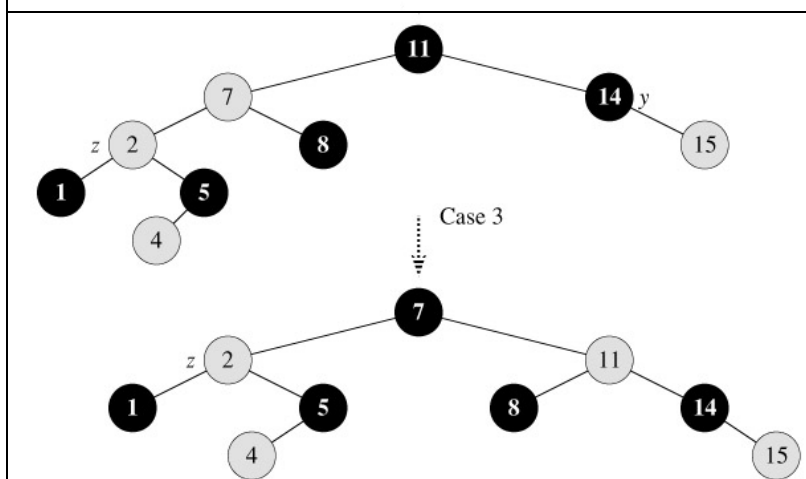
Case 1. Node "Z" (red) is a left or right child and its parent is red and its uncle is red

- Change Z's parent and uncle to black
- Change Z's grandparent to red
- No effect on black height on any node
- Z's grandparent is now Z and recursion to top if #4 still broken at node Z



Case 2. Node "Z" is a right child and its parent is red and its uncle is NOT red

- Rotate left on parent of Z
- Re-label old parent of Z as Z and continue to case #3



Case 3. Node "Z" is a left child and its parent is red and its uncle is NOT red

- Rotate right on grandparent of Z
- Color old parent of Z black
- Color old grandparent of Z red

## CS430 Lecture 11 Activities - KEY

### Opening Questions

1. What do you think the issue we need to handle when deleting a node from a red-black tree? How does red-black delete differ from a BST delete?

#### Find node to delete

**Delete node as in a regular BST (Node actually removed will have at most one child)**

**Need to make sure we don't leave two reds in a row AND maintain the black height of the nodes above the deleted node all the way to the root.**

**If we delete a Red node, STOP, tree still is a Red-Black tree**

**If we delete a black node. Let V be the child of deleted node (if exists, otherwise V is a black nil leaf)**

**If V is red, color it black, STOP**

**If V is black, mark it double black and perform a series of confusing,  $O(\text{height of tree})$  operations to walk the double black up the tree and eliminate it**

### Red-Black Tree Delete

Think of V as having an "extra" unit of blackness. This extra blackness must be absorbed into the tree (by a red node), or propagated up to the root and out of the tree. There are four cases – our examples and "rules" assume that V is a left child. There are symmetric cases for V as a right child

#### Terminology in Examples

- The node just deleted was U
- The node that replaces it is V, which has an extra unit of blackness
- The parent of V is P
- The sibling of V is S



Black Node



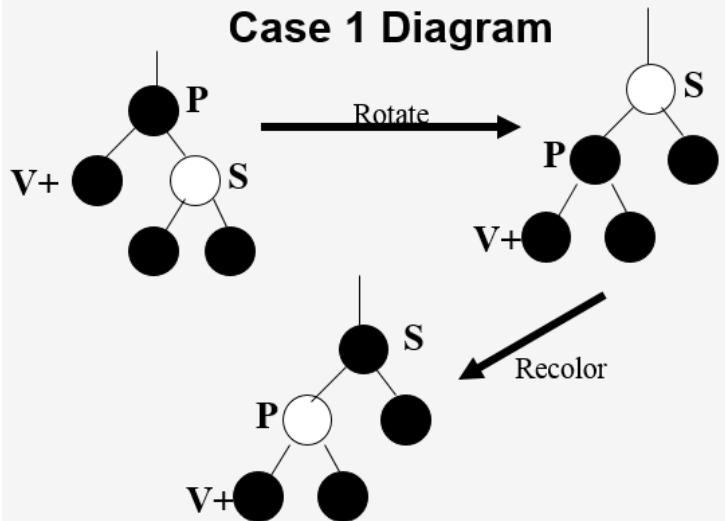
Red or Black and don't care

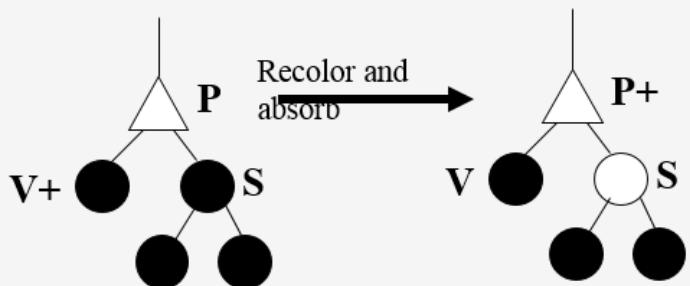
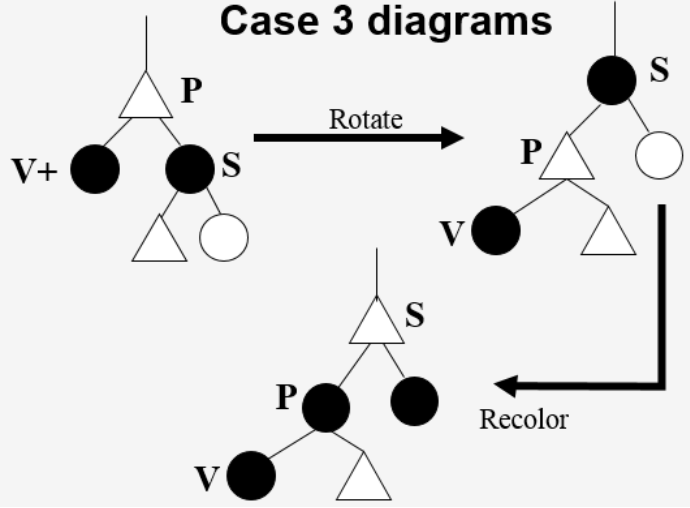
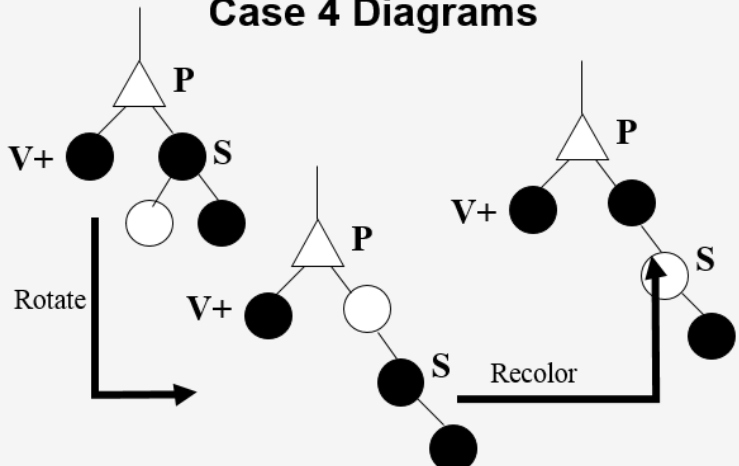


Red Node

- V's sibling, S, is Red
  - Rotate S around P and recolor S & P
- NOT a terminal case – One of the other cases will now apply
- All other cases apply when S is Black

#### Case 1 Diagram



<ul style="list-style-type: none"> <li>▪ V's sibling, S, is black and has <u>two black children</u>.             <ul style="list-style-type: none"> <li>– Recolor S to be Red</li> <li>– P absorbs V's extra blackness                 <ul style="list-style-type: none"> <li>• If P was Red, make it black, we're done</li> <li>• If P was Black, it now has extra blackness and problem has been propagated up the tree</li> </ul> </li> </ul> </li> </ul>	<p style="text-align: center;"><b>Case 2 diagram</b></p>  <p style="text-align: center;">Either extra black absorbed by P or P now has extra blackness</p>
<ul style="list-style-type: none"> <li>▪ S is black</li> <li>▪ S's RIGHT child is RED (Left child either color)             <ul style="list-style-type: none"> <li>– Rotate S around P</li> <li>– Swap colors of S and P, and color S's Right child Black</li> </ul> </li> <li>▪ This is the terminal case – we're done</li> </ul>	<p style="text-align: center;"><b>Case 3 diagrams</b></p> 
<ul style="list-style-type: none"> <li>▪ S is Black, S's right child is Black and S's left child is Red             <ul style="list-style-type: none"> <li>– Rotate S's left child around S</li> <li>– Swap color of S and S's left child</li> <li>– Now in case 3</li> </ul> </li> </ul>	<p style="text-align: center;"><b>Case 4 Diagrams</b></p> 

Red Black Visualization - <http://gauss.eecs.uc.edu/RedBlack/redblack.html>

### AVL Trees

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-ness" is the difference between the heights of sub-trees of every node in the tree. The "height" of tree is the "number of levels" in the tree.

An AVL tree is a binary tree in which the difference between the height of the right and left sub-trees (of any node) is never more than one.

1. How do you think we could keep track of the height of the right and left sub-trees of every node?

Add two ints to each node. Whenever you are inserting or deleting a node (recursively) you must walk past the node twice, once on the way down and once on the way back up (unwinding the recursion). So you can update the heights of the subtrees on the way back up in the recursion.

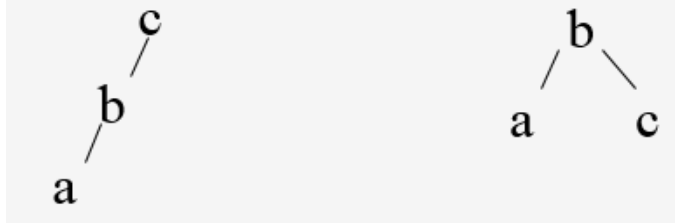
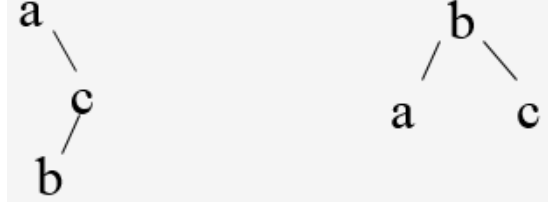
OR

To detect when a "violation" of the AVL criteria occurs, each node must keep track of the difference in height between its right and left sub-trees. We call this "difference" the "balance" factor and define it to be the height of the right sub-tree minus the height of the left sub-tree of a node.

As long as the "balance" factor of each node is never  $>1$  or  $<-1$  we have an AVL tree. As soon as the balance factor of a node becomes 2 (or -2) we need to perform one or more rotations to ensure that the resultant tree satisfies the AVL criteria.

2. If we find an imbalance, how can we correct it without adding any significant cost to the insert or delete?

The idea behind maintaining the "AVL-ness" of an AVL tree is that whenever we insert or delete an item, if we have "violated" the "AVL-ness" of the tree in anyway, we must then restore it by performing a set of manipulations (called "rotations") on the tree. These rotations come in two flavors: single rotations and double rotations (and each flavor has its corresponding "left" and "right" versions).

<p>Single Rotations</p> <p>The imbalance is left-left (or right-right)</p>  <p>Perform single right rotation at "c" (R-rotation) Similar idea for single left rotation (L-Rotation)</p>	<p>Double Rotations</p> <p>The imbalance is left-right (or right-left)</p>  <p>Perform right rotation at "c" then left rotation at "a" (RL-rotation) Similar idea for left rotation then right rotation (LR-Rotation)</p>
--	---

AVL Visualization <http://www.cs.umd.edu/class/spring2002/cmsc420-0401/demo/avltree/>

## CS430 Lecture 12 Activities - KEY

### Opening Questions

Previously we discussed the order-statistic selection problem: given a collection of data, find the  $k$ th largest element. We saw that this is possible in  $O(n)$  time for each  $k$ th element you are trying to find. The naïve method would just be to sort the data in  $O(n \lg n)$  and then access each  $k$ th element in  $O(1)$  time.

1. Which of these two methods would you use if you knew you would be asked to find multiple  $k$ th largest elements from a set of static data?

**If the number of queries is more than  $\lg n$ , use the sort the data first method**

2. What if our collection of data is changing (dynamic), would either these approaches work efficiently for a collection of data that has inserts and deletes happening?

**You would need to redo the  $O(n)$  method after inserts/deletes, there is no way to update it for finding the  $k$ th largest element. For the sorting method you would need to maintain sorted order with each insert/delete (worst case  $O(n)$ ).**

### Augmenting Data Structures

For particular applications, it is useful to modify a standard data structure to support additional functionality. Sometimes the modification is as simple as by storing additional information in it, and by modifying the ordinary operations of the data structure to maintain the additional information.

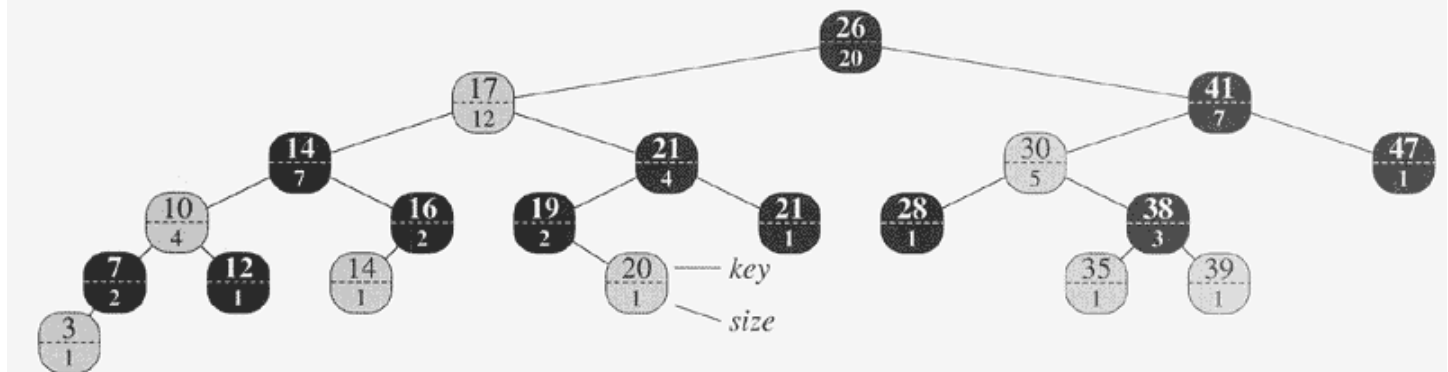
### Dynamic Order-Statistic Trees (Augmenting Balanced Trees)

1. What can we do with a binary search tree (and more efficiently with a balanced binary search tree)?

**Dictionary operations search, insert, delete (more efficient if balanced) and sort. Handles dynamic data better than a list.**

2. Consider the naïve method of finding the  $k$ th largest item is to sort the array and then access the  $k$ th item in  $O(1)$  time. Can we do this with a (balanced) BST? What if we augment it (HINT: recall how counting sort worked)?

**The sorted data is implicit in the BST, but there is no way to easily access the  $k$ th largest by index as in an array. We augment the each node in the red-black tree with its size, where  $\text{size}(x)$  is defined as the number of items in the subtree rooted at  $x$  (which is also the number of items less than (or greater than) the parent of  $x$ ). With this new information, finding the  $k$ th largest element of a tree is a simple process which can be performed in  $O(\text{height}(T))$  time—and since red-black trees are of height  $O(\log n)$ , the running time of the selection algorithm is  $O(\log n)$**



**Figure 14.1** An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual fields, each node  $x$  has a field  $\text{size}[x]$ , which is the number of nodes in the subtree rooted at  $x$ .

3. What is the recursive formula to find the size of a subtree at node  $x$

**Size of a nil node is zero  $x.\text{size} = x.\text{left}.\text{size} + x.\text{right}.\text{size} + 1$**

4. Discuss in detail how would you keep the size at a node correct when you insert a new node, and possibly rotate to keep the tree balanced?

<b>TREE-INSERT(<math>T, z</math>)</b> 1 $y = \text{NIL}$ 2 $x = T.\text{root}$ 3 <b>while</b> $x \neq \text{NIL}$ 4 $y = x$ 5 <b>if</b> $z.\text{key} < x.\text{key}$ 6 $x = x.\text{left}$ 7 <b>else</b> $x = x.\text{right}$ 8 $z.p = y$ 9 <b>if</b> $y == \text{NIL}$ 10 $T.\text{root} = z$ // tree $T$ was empty 11 <b>elseif</b> $z.\text{key} < y.\text{key}$ 12 $y.\text{left} = z$ 13 <b>else</b> $y.\text{right} = z$	<b>LEFT-ROTATE(<math>T, x</math>)</b> 1 $y = x.\text{right}$ // set $y$ 2 $x.\text{right} = y.\text{left}$ // turn $y$ 's left subtree into $x$ 's right sub 3 <b>if</b> $y.\text{left} \neq T.\text{nil}$ 4 $y.\text{left}.p = x$ 5 $y.p = x.p$ // link $x$ 's parent to $y$ 6 <b>if</b> $x.p == T.\text{nil}$ 7 $T.\text{root} = y$ 8 <b>elseif</b> $x == x.p.\text{left}$ 9 $x.p.\text{left} = y$ 10 <b>else</b> $x.p.\text{right} = y$ 11 $y.\text{left} = x$ // put $x$ on $y$ 's left 12 $x.p = y$
--	--

This size field must be updated whenever the tree is modified. Recall that insertion into a red-black tree occurs in two phases. In the first phase, the new node is inserted as the child of an existing node, and sizes can be updated by adding 1 to all size values along the path from the new node to the root. Since the height of the tree is  $O(\log n)$ , this takes  $O(\log n)$  time. In the second phase, rotations are performed in order to restore the red-black properties of the tree. Fortunately, rotations make only local changes to the nodes around the edge on which the rotation takes place. Since at most two rotations are required for an insertion, maintaining the size information takes  $O(1)$  additional time. For similar reasons, updating the size field after a deletion also requires only  $O(\log n)$  time.

13  $y.\text{size} = x.\text{size}$

14  $x.\text{size} = x.\text{left}.\text{size} + x.\text{right}.\text{size} + 1$

5. Discuss in general how would you keep the size at a node correct when you delete a node, and possibly rotate to keep the tree balanced?

Deletion from a red-black tree also consists of two phases: the first operates on the underlying search tree, and the second causes at most three rotations and otherwise performs no structural changes. (See Section 13.4.) The first phase either removes one node  $y$  from the tree or moves upward it within the tree. To update the subtree sizes, we simply traverse a simple path from node  $y$  (starting from its original position within the tree) up to the root, decrementing the size attribute of each node on the path. Since this path has length  $O(\lg n)$  in an  $n$ -node red-black tree, the additional time spent maintaining size attributes in the first phase is  $O(\lg n)$ . We handle the  $O(1)$  rotations in the second phase of deletion in the same manner as for insertion.

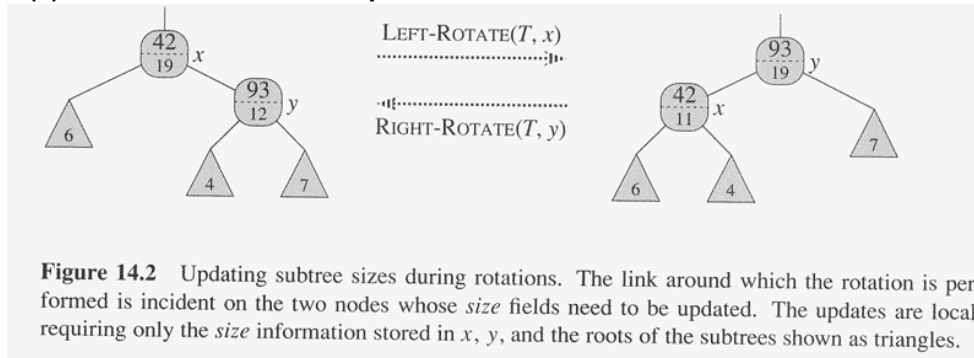


Figure 14.2 Updating subtree sizes during rotations. The link around which the rotation is performed is incident on the two nodes whose *size* fields need to be updated. The updates are local, requiring only the *size* information stored in  $x$ ,  $y$ , and the roots of the subtrees shown as triangles.

6. How can we use the augmented data at each node (size) in a balanced binary search tree to solve the  $k$ th largest item problem?

OS-SELECT( $x, i$ )

1  $r = x.\text{left}.\text{size} + 1$

2 **if**  $i == r$

3     **return**  $x$

4 **elseif**  $i < r$

5     **return** OS-SELECT( $x.\text{left}, i$ )

6 **else** **return** OS-SELECT( $x.\text{right}, i - r$ )

7. How can we use the augmented data at each node (size) in a balanced binary search tree so when given a pointer to a node in the tree we can determine its rank (the index position of the node of the tree data in sorted order)?

OS-RANK( $T, x$ )

```
1   $r \leftarrow \text{size}[\text{left}[x]] + 1$ 
2   $y \leftarrow x$ 
3  while  $y \neq \text{root}[T]$ 
4      do if  $y = \text{right}[p[y]]$ 
5          then  $r \leftarrow r + \text{size}[\text{left}[p[y]]] + 1$ 
6           $y \leftarrow p[y]$ 
7  return  $r$ 
```

8. Why not just use this approach always (not just with dynamically changing data) instead of the  $O(n)$  one?

**Cost to build red-black tree is  $O(n \lg n)$  basically sorting the data**

9. Can we use this approach on a regular BST?

**Yes but not efficiently because of height variability**

## CS430 Lecture 13 Activities - KEY

### Opening Questions

1. Briefly explain what two properties a problem must have so that a dynamic programming algorithm approach will work.

### Optimal Substructure Overlapping Sub-problems

2. Previously we have learned that divide-and-conquer algorithms partition a problem into independent sub-problems, solve each sub-problem recursively, and then combine their solutions to solve the original problem. Briefly, how are dynamic programming algorithms similar and how are they different from divide-and-conquer algorithms?

**Similarities: both techniques analyze a problem recursively, solve sub-problems to arrive at a final answer, exploits the “optimal substructure” of problems/sub-problems**

**Differences: Dynamic programming problems you need to solve all the possible sub-problems (not just sub-problem pairs). But DP takes advantage of overlapping sub-problems for efficiency and usually fills in a table of values of an objective function in optimization problems.**

3. Why does it matter how we parenthesize a chain of matrix multiplications? We get the right answer any way we associate the matrices for multiplication. i.e. If A, B and C are matrices of correct dimensions for multiplication

$(A B) C = A (B C)$

**The total number of multiplications is different depending on how you associate the matrices for multiplication.**

$(A \quad B) \quad C = \quad A \quad (B \quad C)$

$2 \times 5 \quad 5 \times 10 \quad 10 \times 4 \quad 2 \times 5 \quad 5 \times 10 \quad 10 \times 4$

$100 + 80 \text{ mult} \quad 40 + 200 \text{ mult}$

### Dynamic Programming

#### Dynamic Programming Steps

1. Define structure of optimal solution, including what are the largest sub-problems.
2. Recursively define optimal solution
3. Compute solution using table bottom up
4. Construct Optimal solution

Optimal Matrix Chain Multiplication (optimal parenthesization)

1. How many ways are there to parenthesize (two at a time multiplication) 4 matrices  $A*B*C*D$ ?

**exponential,  $(n-1)!$  Permutations of the order of multiplications is an upper bound, but some repetition**

**4 matrices can be parenthesized 5 ways**

$(A) ((BC) D) 231 \quad (A) (B (CD)) 321 \quad (AB)(CD) 132 \text{ same as } 312$

$((AB) C) (D) 123 \quad (A (BC)) (D) 213$

**actually catalan numbers**

$$P(1) = 1$$

$$P(2) = 1$$

$$P(3) = P(1)P(2) + P(2)P(1) = 2$$

$$P(4) = P(1)P(3) + P(2)P(2) + P(3)P(1) = 5$$

$$P(5) = P(1)P(4) + P(2)P(3) + P(3)P(2) + P(4)P(1) = 14$$

$$P(6) = P(1)P(5) + P(2)P(4) + P(3)P(3) + P(4)P(2) + P(5)P(1) = 42$$

It turns out that this is an extremely messy recurrence to solve. Nevertheless, this is a very well-known sequence of numbers called the **Catalan Numbers**. In fact:

$$P(n) = \frac{1}{n} \binom{2n-2}{n-1}$$

which is just the  $n^{th}$  row center (from the top) of Pascal's Triangle divided by  $n$ . Pretty neat, no?

2. Step 1: Generically define the structure of the optimal solution to the Matrix Chain Multiplication problem.

The optimal way to multiply  $n$  matrices  $A_1$  through  $A_n$  is:

**The final multiplication is between matrix  $k$  and matrix  $k+1$**

$(A_1 * A_2 * A_3 \dots A_k) * (A_{k+1} \dots A_n)$



3. Step 2: Recursively define the optimal solution. Assume  $P(1,n)$  is the optimal cost answer. Make sure you include the base case.

$$P(1,n) = P(1,k) + P(k+1,n) + \text{row}_1 * \text{col}_k * \text{col}_n$$

(dimensions of final mult)

Rewriting this over all possible  $k$  (final multiplication possibilities)

$$P(i,i) = 0 \text{ for all } i$$

$$P(1,n) = \min_{1 \leq k < n} \{P(1,k) + P(k+1,n) + \text{row}_1 * \text{col}_k * \text{col}_n\}$$

4. Use proof by contradiction to show that Matrix Chain Multiplication problem has optimal substructure, i.e. the optimal answer to problem must contain optimal answers to sub-problems.

$$P(1,n) = \min_{1 \leq k < n} \{P(1,k) + P(k+1,n) + \text{row}_1 * \text{col}_k * \text{col}_n\}$$

Must  $P(1,k)$  be optimal for  $A_1 \dots A_k$  ?

Assume you have the optimal paren. of matrixes 1 to  $n$  and the last paren is at position  $k$  (between matrix  $k$  and matrix  $k+1$ ); Optimal answer is  $(A_1 \dots A_k) (A_{k+1} \dots A_n)$

$P(1,n)$  is minimized for this “ $k$ ”

To prove  $(A_1 \dots A_k)$  and  $(A_{k+1} \dots A_n)$  must also be optimally paren’d we use a proof by contradiction

If  $(A_1 \dots A_k)$  is not optimally paren’d, then we must have a different paren value  $P'(1,k) < P(1,k)$

But if this is true, then  $P(1,n)$  above could not have been optimal because we could have substituted  $P'(1,k)$  for  $P(1,k)$  and gotten a smaller answer.

Therefore,  $P'(1,k) < P(1,k)$  is not true and  $P(1,k)$  must also be optimal

IF YOU NEED TO SHOW THE PROOF BY CONTRADICTION APPROACH IS VALID

**Optimal solution to problem**  $\longrightarrow$  **Contains optimal solutions to sub-problems**

$$p \rightarrow q \implies (p \wedge \neg q) \rightarrow (r \wedge \neg r)$$

Assume  $p$  and  $\neg q$  and argue  $\neg q \rightarrow \neg p$ . Thus  $p \wedge \neg p$ , contradiction, proves  $\neg q$  is false, therefore  $q$  is true

$p$	$q$	$r$	$p \rightarrow q$	$p \wedge \neg q$	$r \wedge \neg r$	$p \wedge \neg q \rightarrow r \wedge \neg r$
T	T	T	T	F	F	T
T	T	F	T	F	F	T
T	F	T	F	T	F	F
T	F	F	F	T	F	F
F	T	T	T	F	F	T
F	T	F	T	F	F	T
F	F	T	T	F	F	T
F	F	F	T	F	F	T

CS430 Topic 09a 1/15

5. Step 3: Compute solution using a table bottom up for the Matrix Chain Multiplication problem. Use your answer to question 3 above. Note the overlapping sub-problems as you go. .Step 4: Construct Optimal solution

A	B	C	D
2*4	4*6	6*3	3*7

$$P(1,1)=0 \quad P(2,2)=0 \quad P(3,3)=0 \quad P(4,4)=0$$

$$P(1,2)=48 \quad P(2,3)=72 \quad P(3,4)=126$$

$P(1,3)=84$        $A(BC) = 0 + 72 + 2.4.3 = 96$   
 $ROOT(1,3) = 2$     $(AB)C = 48 + 0 + 2.6.3 = 84$

$P(2,4) = 156$      $B(CD) = 0 + 126 + 4.6.7 = 294$   
 $ROOT(2,4)=3$     $(BC)D = 72 + 0 + 4.3.7 = 156$

$P(1,4)=126$        $A(BCD)=0 + 156 + 2.4.7 = 212$   
 $ROOT(1,4)=3$     $(AB)(CD) = 48+126 + 2.6.7$   
                       $(ABC)D = 84 + 0 + 2.3.7 = 126$

$((A\ B)\ C)\ D)$

P, root	1	2	3	4
1	0	48,1	84,2	126,3
2		0	72,2	156,3
3			0	126,3
4				0

## CS430 Lecture 14 Activities - KEY

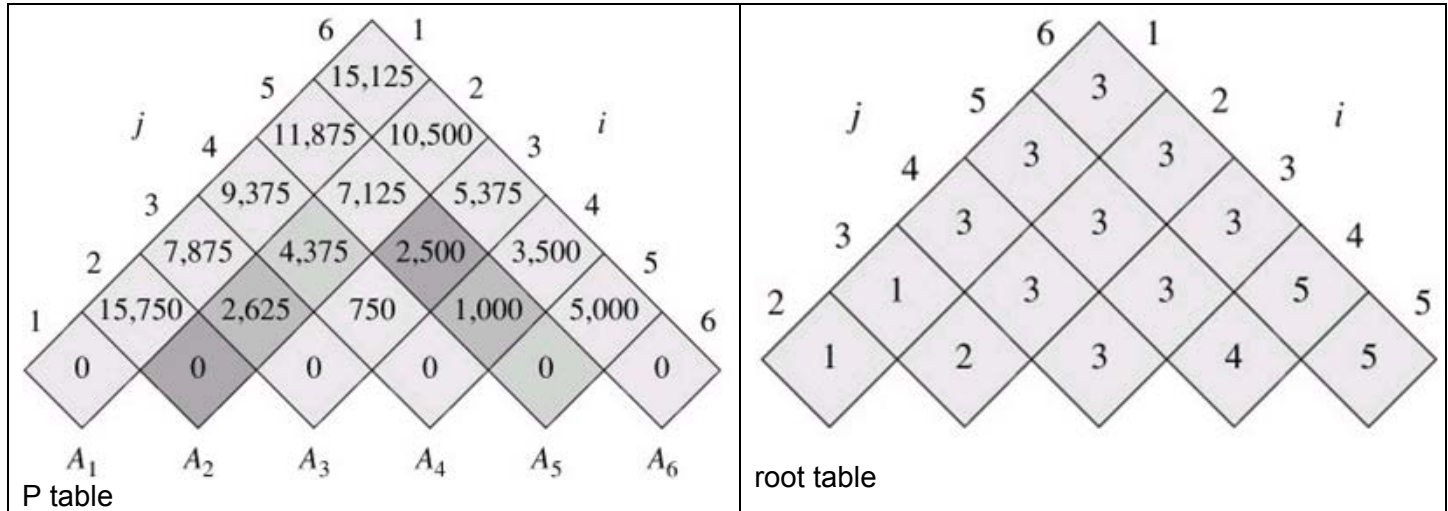
### Opening Questions

1. Why are optimal solutions to sub-problems stored in a table in dynamic programming solutions?

**Because of the problem trait of overlapping sub-problems you will need to know the solutions to sub-problems more than once, but you only need to calculate once and save in a table to look up again later.**

Constructing the answer for the Optimal Matrix Chain Multiplication (optimal parenthesization) from the dynamic programming table. See the solution to the example problem

A1    A2    A3    A4    A5    A6  
30x35   35x15   15x5   5x10   10x20   20x25



2. Write the optimal parenthesization.

**((A1 (A2 A3)) ((A4 A5) A6))**

**#2 #1 #5 #3 #4**

3. Write pseudocode to use the root table to print the optimal parenthesization. Then write pseudocode to use the root table to actually perform the multiplications in the optimal parenthesization.

<b>PRINT-OPT-PARENS</b> (root, i, j) //i=1, j=n if i = j then print "A <sub>i</sub> " else print "(" <b>PRINT-OPT-PARENS</b> (root, i, root[i, j]) print "*" <b>PRINT-OPT-PARENS</b> (root, root[i, j] + 1, j) print ")"	<b>Matrix MULT-OPT-PARENS</b> (root, i, j) //i=1, j=n if i = j then return A <sub>i</sub> else x= <b>MULT-OPT-PARENS</b> (root, i, root[i, j]) y= <b>MULT-OPT-PARENS</b> (root, root[i, j] + 1, j) return <b>MULT_MATRIX</b> (x,y)
--	---

### Longest Common Subsequence

Example: X[1.....m]    Y[1.....n]

X : ABCBDAB    Y : BDCABA

Length of LCS = 4    BCBA    or    BCAB

1. The brute force approach would be to find all subsequences of one input, see if each exist in other input. How many are there?

**$O(n * 2^m)$      $m < n$**

2. Step 1: Generically define the structure of the optimal solution to the Longest Common Subsequence problem. The longest common subsequence of sequence X[1.....m] and sequence Y[1.....n] is:

**IF X(m) = Y(n)**

**One longer than longest common subsequence of sequence X[1.....m-1] and sequence Y[1.....n-1]**

**ELSE THE LONGER OF**

**longest common subsequence of sequence X[1.....m] and sequence Y[1.....n-1]**

**OR longest common subsequence of sequence X[1.....m-1] and sequence Y[1.....n]**

3. Step 2: Recursively define the optimal solution. Assume C(i,j) is the optimal answer for up to position i in X and position j in Y. Make sure you include the base case.

**C[i,j] is LCS of X[1..i] .. m    Y[1..j] ..n    j < m    j < n**

**C[0,0] = C[0,?] = C[?,0] = 0**

$$C[i, j] = \begin{cases} C[i-1, j-1] + 1 & X[i] = Y[j] \\ \text{Max} \begin{cases} C[i, j-1] \\ C[i-1, j] \end{cases} & \text{else} \end{cases}$$

4. Use proof by contradiction to show that Longest Common Subsequence problem has optimal substructure, i.e. the optimal answer to problem must contain optimal answers to sub-problems.

**Proof when  $X[i] = Y[j]$**

Assume  $Z[1..k]$  is LCS of  $X[1..i]$  and  $Y[1..j]$  (assume you have optimal answer)

$C[i, j] = k$   $k \leq i, k \leq j$

If  $X[i] = Y[j]$  then they must equal  $Z[k]$  (discuss)

But then  $Z[1, k-1]$  must also be the longest common subsequence of  $X[1..i-1]$ ,  $Y[1..j-1]$  because if not then  $Z[1..k]$  wouldn't be LCS of  $X[1..i]$ ,  $Y[1..j]$  (contradiction of first assumption).

If there was a longer common subsequence than  $Z[1, k-1]$  of  $X[1..i-1]$ ,  $Y[1..j-1]$ , then we could have added  $X[i]$  (or  $Y[j]$ ) to it a gotten something longer than  $Z[1..k]$

similar of  $X[i] \neq Y[j]$

ANSWER TO  
THIS  $Z[1..k-1]$  IS  
A L.C. COMMON SEQUENCE  
OF  $X[1..i-1]$  &  $Y[1..j-1]$  ← SMALLER PROBLEM

---

IF THERE EXIST A LONGER  
COMMON SUBSEQUENCE OF (THAN  
 $X[1..i-1]$   $Y[1..j-1]$   $Z[1..k-1]$ )  
 $\Rightarrow Q[1..m]$   
 $m > k-1$

THEN  $Q + X[i]$   
 $Y[j]$  would  $m+1 > k$   
BE LONGER THAN  $Z[1..k]$   
IMPOSSIBLE BECAUSE  
 $Z$  WAS ASSUMED LCS  
 $X[1..i]$   
 $Y[1..j]$

5. Step 3: Compute solution using a table bottom up for the Longest Common Subsequence problem. Use your answer to question 3 above. Note the overlapping sub-problems as you go. .Step 4: Construct Optimal solution

$X = \text{ABCBDAB}$

$Y = \text{BDCABA}$

X

C \ Y	0	1	2	3	4	5	6
	B	D	C	A	B	A	
0	0	0	0	0	0	0	0
1 A	0	0	0	0	1	1	1
2 B	0	1	1	1	2	2	2
3 C	0	1	2	2	2	2	2
4 B	0	1	2	2	3	3	3
5 D	0	2	2	2	3	3	3
6 A	0	1	2	2	3	4	4
7 B	0	1	2	3	4	4	4

$$C(7, 6) = 4$$

X: A B C B D A B  $n=7$

Y: B D C A B A  $n=6$

$$C(0, 0) = 0 \quad C(1, 0) = 0$$

$$C(0, 1) = 0$$

$$C(0, 0) = 0 \quad C(1, 0) = 0$$

$$C(0, 1) = 0$$

$$C(1, 1) = 0$$

$$x[2] = y[1]$$

$$C(2, 1) = C(1, 0) + 1 = 1$$

$$C(1, 2) = 0$$

$$C(2, 2) = \max \{ C(2, 1), C(1, 2) \} = 1$$

3  $m+n$  nodes

$$C(3, 4)$$

$$x[1 \dots 3] \\ y[1 \dots 4]$$

$$C(2, 3)$$

$$C(2, 4)$$

$$C(3, 3)$$

$$C(1, 2)$$

$$C(1, 3)$$

$$C(0, 2)$$

$$1, 3 \quad 1, 4$$

$$C(2, 3)$$

$$2, 2 \quad 2, 3$$

$$3, 2$$

## CS430 Lecture 15 Activities - KEY

### Opening Questions

1. In the Optimal Binary Search Tree problem we are not just trying to balance the tree, instead we are trying to minimize what?

**The expected search time for any key, high probability search keys will be nearer the root of the tree.**

2. What is different about the 0-1 knapsack problem as compared to the other problems we solved with dynamic programming?

**All the other problems have some fixed order to the elements, where 0-1 knapsack does not**

### Optimal Binary Search Tree

The Optimal Binary Search Tree problem is a special case of a BST where the data is static (no inserts or deletes) and we know the probability of each key in the data being searched for. We want to minimize total expected search time for the BST. Recall expectation

$$E(X) = \sum_{s \in S} X(s)p(s)$$

1. Apply the expectation formula to the Optimal Binary Search Tree problem with n keys and C(i) is how deep item "i" is. P(i) is probability of search for item "i"

$$\sum P(i) * C(i)$$

2. The brute force approach would be to find all possible BSTs, find the expected search time of each and pick the minimum one. How many BSTs are there?

1. Total no of Binary Trees are =  $\frac{(2n)!}{(n+1)!n!}$
2. Summing over i gives the total number of binary search trees with n nodes.

$$t(n) = \sum_{i=1}^n t(i-1) t(n-i).$$

The base case is  $t(0) = 1$  and  $t(1) = 1$ , i.e. there is one empty BST and there is one BST with one

$$t(2) = t(0)t(1) + t(1)t(0) = 2$$

$$t(3) = t(0)t(2) + t(1)t(1) + t(2)t(0) = 2 + 1 + 2 = 5$$

$$t(4) = t(0)t(3) + t(1)t(2) + t(2)t(1) + t(3)t(0) = 5 + 2 + 2 + 5 = 14$$

node.

3. Step 1: Generically define the structure of the optimal solution to the Optimal Binary Search Tree problem.

The optimal binary search tree with n keys n keys and C(i) is how deep item "i" is and P(i) is probability of search for item "i" is:

$$A[1,n] = \text{Min over all choices for root } 1 \leq k \leq n \left\{ \begin{array}{l} \text{Optimal left subtree expected} \\ + \\ \text{Optimal right subtree expected} \\ + \\ \text{Left subtree increase search} \\ + \\ \text{Right subtree increase search} \\ + \\ \text{Root node search} \end{array} \right\}$$

4. Step 2: Recursively define the optimal solution. Assume  $A(i,j)$  is the optimal answer for keys  $i$  to  $j$ . Make sure you include the base case.

$A[i,j] = \text{Min expected search time for keys } i \text{ to } j$

$A[i,i] = p(i) * 1$

$$A[i,j] = \min_{\text{roots } t} \left\{ A[i,t-1] + A[t+1,j] + \sum_{q=i}^j p(q) * 1 \right\}$$

5. Use proof by contradiction to show that Optimal Binary Search Tree problem has optimal substructure, i.e. the optimal answer to problem must contain optimal answers to sub-problems.

**Assume you have optimal BST, say item  $k$  may be the root.**

**Items 1 to  $(k-1)$  must be in left subtree**

**Items  $(k+1)$  to  $n$  must be in right subtree**

**Those items in those respective subtrees must also be organized optimally for those items.**

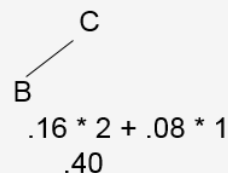
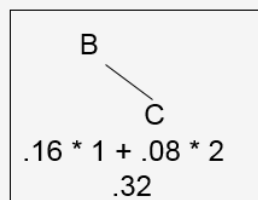
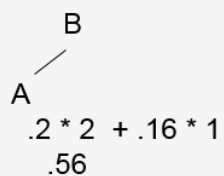
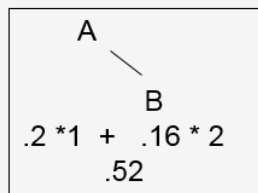
**If not then original BST was not optimal**

6. Step 3: Compute solution using a table bottom up for the Optimal Binary Search Tree problem. Use your answer to question 4 above. Note the overlapping sub-problems as you go. .Step 4: Construct Optimal solution

A B C D E F  
.2 .16 .08 .22 .21 .13

$A =$

	0	1	2	3	4	5	6
1		.2	.52				
2			.16	.32			
3				.08	.38		
4					.22		
5						.21	
6							.13



$$A[3,4] = \text{MIN} \left\{ \begin{array}{l} A[3,2] + A[4,4] \quad t=3 \\ \text{or} \\ A[3,3] + A[5,4] \quad t=4 \end{array} \right\}$$

$$= (0.08 + .22) = 0.38$$

Optimal Binary Search Tree <http://oopweb.com/Algorithms/Documents/AnimatedAlgorithms/VolumeFrames.html>

One commonality to all the dynamic programming solutions we explored is that all the problems had some sort of ordering restriction. Here is an example that does not.

### 0-1 Knapsack Problem

Given  $N$  items and a total weight limit  $W$ ,  $v_i$  is the value and  $w_i$  is the weight of item  $i$ , maximize the total value of items taken.

The dynamic programming algorithm computes entries for a matrix  $C[0..N, 0..W]$

$C[i,j]$  = the optimal value of the items put into the knapsack from among items  $\{1,2,\dots,i\}$  with total weight  $\leq j$   
with  $C[0,?] = C[?,0] = 0$

7. When you think about calculating  $C[i,j]$  there are two options. The  $i$ th item is in that optimal answer or is not. Write the recurrence relation.

$C[i,j] = \max \{ C[i-1,j], v_i + C[i-1,j-w_i] \}$   
ith item not used vs. ith item used

8. Write pseudocode to fill in the  $C[i,j]$  matrix, use your answer from #7.

**initialize C matrix row 0 and column 0 to zero**

```
for i = 1 to N
  for j = 1 to W
    if  $w_i \leq j$ 
      if  $v_i + C[i-1,j-w_i] > C[i-1,j]$ 
         $C[i,j] = v_i + C[i-1,j-w_i]$ 
      else  $C[i,j] = C[i-1,j]$ 
    else  $C[i,j] = C[i-1,j]$ 
```



## CS430 Lecture 16 Activities - KEY

### Opening Questions

1. Briefly explain what two properties a problem must have so that a greedy algorithm approach will work.

**Optimal Substructure and Greedy Choice Property (local optimal choice leads to global optimal solution)**

2. A good cashier gives change using a greedy algorithm to minimize the number of coins they give back. Explain this greedy algorithm.

**Give as many of the largest coins first, then next largest, etc. This will minimize total number of coins (if the coin set was designed correctly).**

3. For what types of optimization problems does optimal substructure fail?

**Where the sub-problems are NOT independent.**

### Activity Selector Problem

Given a set  $S$  of  $n$  activities each with start  $S_i$ , Finish  $F_i$ , find the maximum set of Compatible Activities (non-overlapping). (or could be jobs scheduled with fixed start and end times instead of meetings)

1. The brute force approach would be to find all possible subsets of  $n$  activities, eliminate the ones with non-compatible meetings, and find the largest subset. How many subsets are there?

$2^n$

2. Prove the Activity Selector Problem has optimal substructure (using similar proof by contradiction approach that we used for dynamic programming: assume you have an optimal answer, remove something to get to the largest sub-problem, show that the sub-problem must also be solved optimally).

**A: Optimal Subset    S: Set of MTGS    Assume A is solution to S**

**$\{2,4,k,8,11\}$  No order     $\{1,2,3,\dots,n\}$  No Order**

**Remove a meeting from the optimal answer, it does not matter which one you remove but if you remove the earliest ending or latest starting the math is easier.**

**Remove earliest ending activity from A    (Happens to be MTG  $k$ )  $f_k < f_2 < f_4 < f_i$**

**$A' = \{2,4,8,11\}$  Must be optimal solution to sub-problem  $S' = \{S: s_i > f_k\}$     All MTGS start after finish of  $k$**

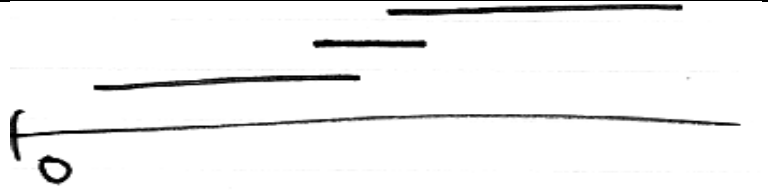
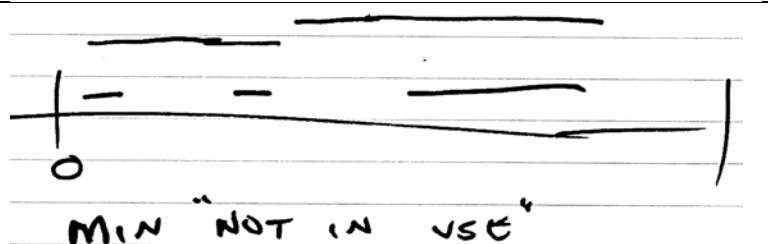
**$A'$  must be solution to  $S'$ , if it isn't then original solution A was not optimal for S. If X is bigger than  $A'$  (and X is valid subset for  $S'$ ), then  $X + \{k\} > A$  which is contradiction. So X cannot exist, A is optimal.**

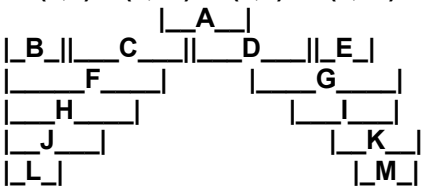
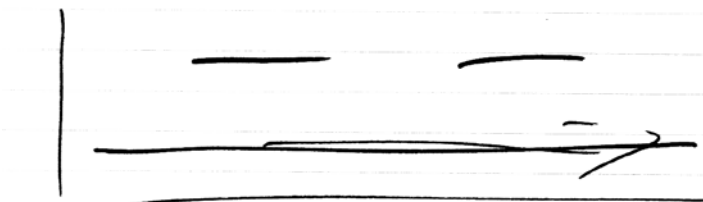
### Proving a Greedy Choice Property

To prove a Greedy Choice Property for a problem (that local optimal choice leads to global optimal solution) use the following "cut and paste" proof.

1. Assume you have an optimal answer that does not contain the greedy choice you are trying to prove.
2. Show that you can "cut" something out of that optimal answer and "paste" in the greedy choice you are trying to prove. Therefore either the assumed optimal answer already contained the greedy choice, or you can make the assumed optimal contain the greedy choice.
3. Therefore there is always an optimal answer that contains the greedy choice (can be continued like an inductive proof).

3. Try various "common sense" greedy approaches that divide the problem into a sub-problem(s) and try to come up with counter-examples or prove the greedy choice is correct.

<b>Pick Shortest Meeting - Counter Example</b>	
<b>Minimize "Not in use" - Counter Example</b>	

<b>Pick Least Conflict - Counter Example</b>	<p> <math>A=(4,6)</math> <math>B=(0,1)</math> <math>C=(1,5)</math> <math>D=(5,9)</math> <math>E=(9,10)</math> <math>F=(0,4)</math> <math>G=(6,10)</math>  <math>H=(0,3)</math> <math>I=(7,10)</math> <math>J=(0,2)</math> <math>K=(8,10)</math> <math>L=(0,1)</math> <math>M=(9,10)</math> </p>  <p> A overlaps 2 mtgs, while everything else overlaps at least 3 mtgs  If we choose A, we exclude C and D, and the max we can get is 3 meetings (A, and one from each side of A)  The optimal answer is 4 mtgs B, C, D, E (other similar answers length 4) </p>
<b>Earliest Start Time First - Counter Example</b>	<p> <b>GREEDY</b>  <b>EARLIEST</b>  <b>START TIME FIRST</b> </p> 
<b>Earliest Finish Time first (or latest start time first)</b>	<p> Assume MTGS ordered earliest finish time first  assume optimal solution A for problem S  If A doesn't have Greedy choice (Earliest finish Time), we want to show there is an equal optimal solution B that does have the greedy choice  meeting K ends first in solution A  <math>B = A - \{K\} + \{1\}</math>  Replace non greedy first choice in A with Greedy choice  This can be done because <math>f_1 &lt; f_k</math> </p> <p> Therefore, There is always an answer that contains the greedy choice </p>

Does every Optimization Problem exhibit Optimal Substructure?

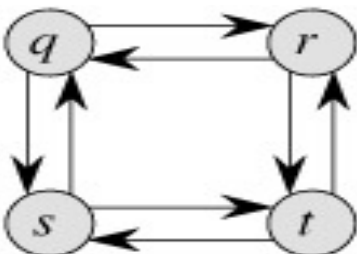
Consider the following two problems in which we are given a directed graph  $G = (V, E)$  and vertices  $u, v \in V$ .

1. Un-weighted shortest path: Find a path from  $u$  to  $v$  consisting of the fewest edges.
2. Un-weighted longest simple path: Find a simple path from  $u$  to  $v$  consisting of the most edges.

4. Try to prove Optimal Substructure for the above two problems.

**Un-weighted shortest path proof** - If  $w$  is on the shortest path from  $u$  to  $v$ , then along that path from  $u$  to  $w$  must be the shortest path from  $u$  to  $w$ . Also, along that path from  $w$  to  $v$  must be the shortest path from  $w$  to  $v$ .

**Un-weighted longest simple path counterexample** - The path  $q \rightarrow r \rightarrow t$  is a longest simple path from  $q$  to  $t$ , but the sub-path  $q \rightarrow r$  is not a longest simple path from  $q$  to  $r$ , nor is the sub-path  $r \rightarrow t$  a longest simple path from  $r$  to  $t$ .



5. Why does optimal substructure fail for some optimization problems?

- Although two sub-problems are used in a solution to a problem for both longest and shortest paths, the sub-problems in finding the longest simple path are not *independent*, whereas for shortest paths they are.
- What do we mean by sub-problems being independent? We mean that the solution to one sub-problem does not affect the solution to another sub-problem of the same problem.

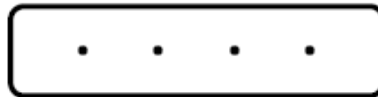
### ANOTHER EXAMPLE


Suppose you are given a set of points on the plane. Consider the problem of connecting each point to exactly one other point in such a way that the cost, the sum of the lengths of the edges, is minimized.

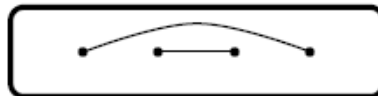
A brute force approach to this problem is to generate each possible matching of points and to find the corresponding costs. The matching with the minimum cost is thus the solution to the problem. While this approach will inarguably produce correct results, it does require exponential time. This encourages us to search for a more efficient algorithm.

One simple algorithm that comes to mind is a greedy algorithm. Find the two nearest points that are not yet connected (to other points) and connect them. Repeat this process exhaustively. In the case of a tie, select a pair of points arbitrarily. How efficient is this algorithm? If there are  $n$  points, it will take time  $\binom{n}{2} + \binom{n-2}{2} + \binom{n-4}{2} + \dots = \Theta(n^3)$  to iteratively look for two nearest points.

Before declaring success, we must ask if our algorithm is correct: does it, in fact, give optimal results? Unfortunately it does not. Consider four points equally spaced along a line:



In the first iteration, there are three optimal pairs of points, so the algorithm chooses a pair arbitrarily. Say it chooses the two points in the middle and is then forced to connect the two remaining points: 



## CS430 Lecture 17 Activities - KEY

### Fractional Knapsack Problem

n items       $w_i$  weight       $v_i$  values  
Maximize value      Total weight  $\leq W$

Total weight limit =  $W$   
Fractional amounts of items are allowed

1. Prove that the Fractional Knapsack Problem has optimal substructure.

Given optimal solution **A** to fractional Knapsack problem ( $w'_i \leq w_i$  for all  $i$ )

$A = \{0, w'_3, 0, w'_i, w'_5, \dots, w'_n\}$

Remove item  $i$  (weight  $w'_i$ ) from **A**

**A'** is same as **A** without  $w'_i$

$A' = \{0, w'_3, 0, w'_5, \dots, w'_n\}$

**A'** is be optimal solution for sub-problem, **S'** Total weight =  $W - w_i$  original  $n-1$  items (excluding item  $i$ )

But what if **B** is solution to **S'** and **B** was better than **A'** (value  $B >$  value **A'**)

Then item  $i$  plus **B** would be better than **A**. Contradiction. **B** cannot exist and **A'** must be optimal or **S'**

2. Try various "common sense" greedy approaches that divide the problem into a sub-problem(s) and try to come up with counter-examples or prove the greedy choice is correct using the "cut and paste" proof.

Max value first counterexample	<div><math>W = 5</math></div> <table><tr><td>1</td><td>✓</td><td><math>w</math></td></tr><tr><td>2</td><td>10</td><td>5</td></tr><tr><td>3</td><td>6</td><td>3</td></tr><tr><td></td><td>5</td><td>2</td></tr></table>	1	✓	$w$	2	10	5	3	6	3		5	2
1	✓	$w$											
2	10	5											
3	6	3											
	5	2											
Min weight first counterexample	<div><math>W = 5</math></div> <table><tr><td>1</td><td>✓</td><td><math>w</math></td></tr><tr><td>2</td><td>15</td><td>5</td></tr><tr><td>3</td><td>6</td><td>3</td></tr><tr><td></td><td>5</td><td>2</td></tr></table>	1	✓	$w$	2	15	5	3	6	3		5	2
1	✓	$w$											
2	15	5											
3	6	3											
	5	2											
Max value/weight first	<p>Prove global optimal solution can be found from a local greedy choice</p> <p>Assume we have optimal answer A has <math>0 \leq w'_i \leq w_i</math> weight of each item (not necessarily created from greedy choice)</p> <p>And assume we have the items in greedy choice order</p> $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \frac{v_3}{w_3} \geq \dots \geq \frac{v_n}{w_n}$ $A = \{w'_1, w'_2, \dots, w'_n\}$ <p>If item 1 has the largest value/weight, either A has the greedy choice (<math>w'_1 &gt; 0</math>) or A doesn't have Greedy Choice (<math>w'_1 = 0</math>)</p> <p>Show that there is an equally optimal solution B that does have greedy choice (removing some weight of item k and adding same weight of item 1)</p> <p><math>B = A - \{\text{item } k\} + \{\text{item 1 same weight as removed}\}</math></p> <p><math>W_A = W_B = W</math> and total <math>V_B \geq</math> total <math>V_A</math></p> <p>Replacing min {weight of k or weight of 1} Lbs item k with min {weight of k or weight of 1} Lbs item 1 Replace the rest of k (if exist) with item 2 (continuing greedy choice)</p> <div><math display="block">\frac{V_1}{W_1} \geq \frac{V_k}{W_k}</math><p>(items in greedy order) and <math>B_{\text{VALUE}} \geq A_{\text{VALUE}}</math></p></div> <p>EXAMPLES</p>												

	10	20	30	W=50
	\$60	\$100	\$120	
$\frac{v}{w} =$	$\frac{\$6}{lb}$	$\frac{\$5}{lb}$	$\frac{\$4}{lb}$	
All	All			
10lbs + 20lbs + 20lbs = 50				
\$60 + \$100 + \$80 = \$240				
	10	20	30	
	\$40	\$100	\$150	
$\frac{v}{w} =$	$\frac{\$4}{lb}$	$\frac{\$5}{lb}$	$\frac{\$5}{lb}$	\$250

### Huffman Codes Problem

#### Data Encoding Background

- Data is a sequence of characters
- Fixed Length – Each character is represented by a unique binary string. Easy to encode, concatenate the codes together. Easy to decode, break off 3-bit codewords and decode each one.
- Variable Length – Give frequent characters shorter codewords, infrequent characters get long codewords. However, how do we decode if the length of the codewords are variable?
- Prefix Codes - No codeword is a prefix of another codeword. Easy to encode, concatenate the codes together. Easy to decode, since no codeword is a prefix of another, strip off one bit a time and match to unique prefix code

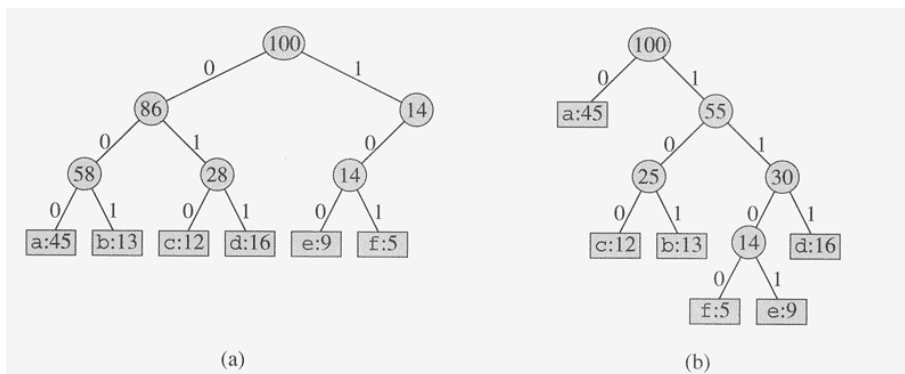
#### Huffman Codes

- Are a Data Compression technique using a greedy algorithm to construct an optimal variable length prefix code
- Use frequency of occurrence of characters to build an optimal way to represent each character as a binary string
- Use a Binary tree method – 0 means go to left child, 1 means go to right child (not a binary search tree).
- Cost of Tree in bits  $B(T) = \sum_{\text{for all } c \in C} \text{freq}(c) * \text{depth}(c)$

#### Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

**Figure 16.3** A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If each character is assigned a 3-bit codeword, the file can be encoded in 300,000 bits. Using the variable-length code shown, the file can be encoded in 224,000 bits.



**Figure 16.4** Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. (a) The tree corresponding to the fixed-length code  $a = 000, \dots, f = 101$ . (b) The tree corresponding to the optimal prefix code  $a = 0, b = 101, \dots, f = 1100$ .

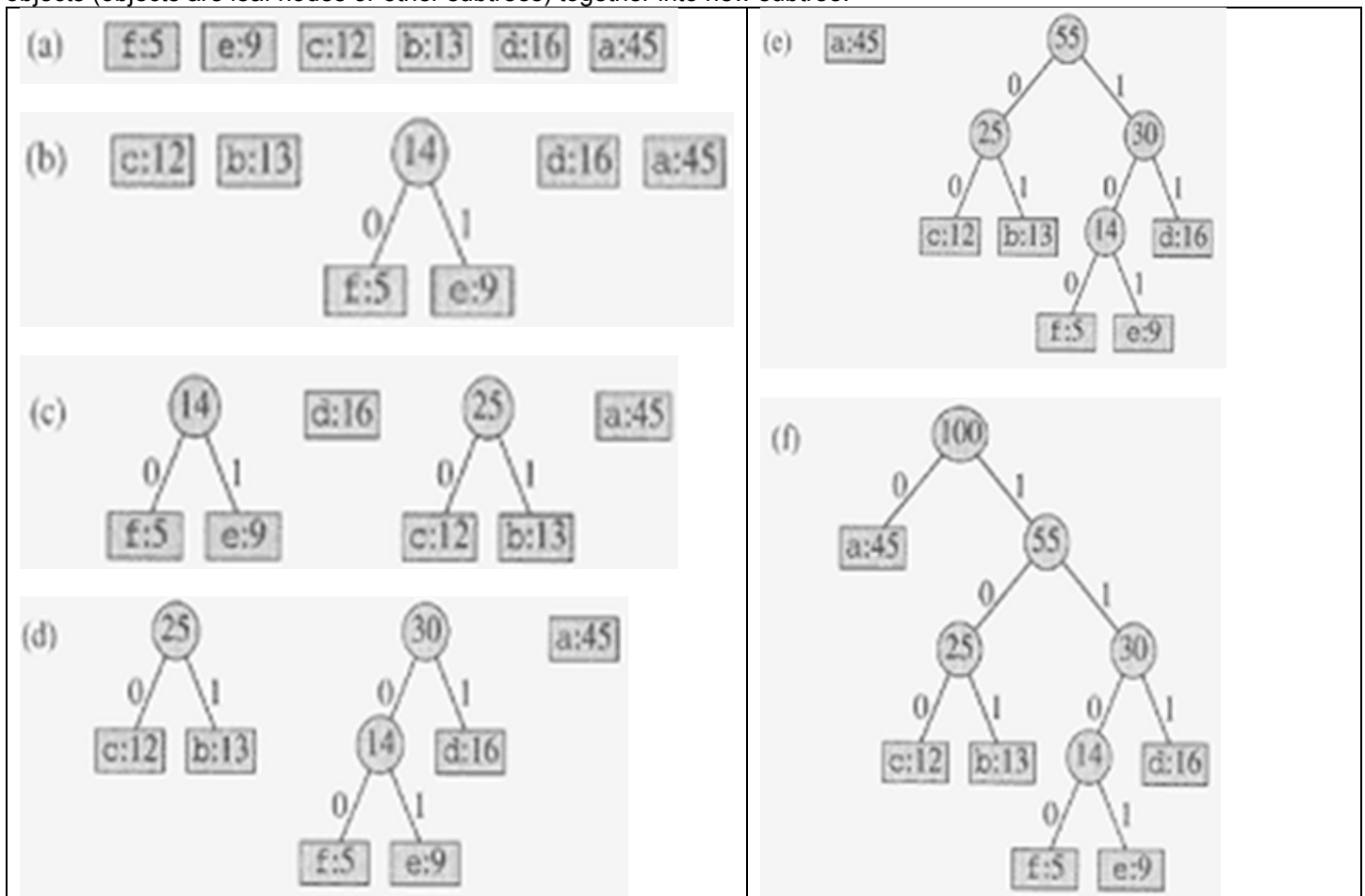
3. Prove that the Huffman Codes Problem has optimal substructure.

**Assume you have an optimal Huffman Code tree  $T$  for a set of characters  $C$  along with each character frequency  $F$ . Replace the two lowest freq characters (must be siblings) from the optimal answer (assume it is characters  $c_j$  and  $c_k$ ) with a new character  $z$  with  $f_z = f_j + f_k$ .**

**You have a new problem with characters  $C' = C - c_j - c_k + z$  along with their frequencies  $F' = F - f_j - f_k + f_z$ .  $T'$  is the new Huffman code tree, is it optimal for  $C'$  and  $F'$ ?**

**Let's say someone tells you they have a more optimal coding than  $T'$ , call it  $T''$ . If that is true then you could replace  $z$  in that tree with a new subtree containing characters  $c_j$  and  $c_k$  and the result tree would have same characters as  $T$  but smaller total cost. Impossible  $T$  was assumed to be optimal.**

The greedy approach that works is: Build the tree bottom up by using a minimum priority queue to merge 2 least frequent objects (objects are leaf nodes or other subtrees) together into new subtree.



4. Proof this greedy approach leads to an optimal Huffman Code tree: Build the tree bottom up by using a minimum priority queue to merge 2 least frequent objects (objects are leaf nodes or other subtrees) together into new subtree. **Assume you have an optimal Huffman Code tree  $T$  for a set of characters  $C$  along with each character frequency  $F$ . Assume character  $x$  and character  $y$  are the two lowest frequencies but are NOT the two lowest leaves. You can swap character  $x$  and character  $y$  with the two lowest leaves (shown as  $a$  and  $b$  below) and get a new Huffman Code Tree with at least, if not smaller, total cost.**

