Topic 7

Tree (Part I)

資料結構與程式設計
Data Structure and Programming

11/14/2018

What we have learned before...

- We have learned some linear type data structures --- list, array, queue, stack, etc.
- However, in real life, many data types are NOT stored in a linear sequence. For example,
  - Directories and files
  - Employee structure in a company

In additon, complexity tradeoffs ---

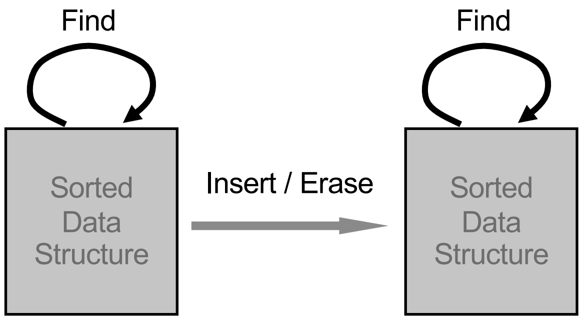
- Remember in "List and Array" topic we compare the complexity of the following functions

	DList	Array
Insert (any pos)	O(1)	O(n) or O(1)
Erase (any pos)	O(1)	O(n) or O(1)
Find	O(n)	O(n log n) to sort the array, O(log n) to find
Memory Overhead	16*n + 16	24

Remember the difference between → O(1), O(log n), O(n)

- To have better "find" performance
  → Data needs to be sorted
- List → fast in insert/erase, slow in find
  - Data cannot be sorted efficiently
- Array → not good in insert/erase, OK in find
  - Takes O(n log n) to update the order
- What if we need to
  - Many "find()" operations
  - Some "insert/erase" operation from time to time

In other words...



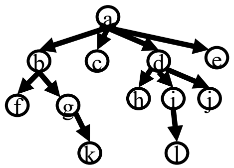
Better DS for "find"

- We will introduce several data types that have good "find" complexity (O(log n)), and OK "insert/erase" complexity (also O(log n))
  - Heap
  - Set
  - Map

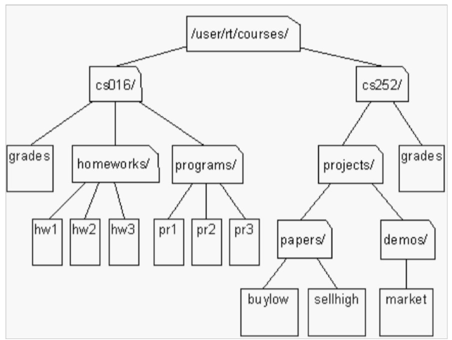
→ They are all different variations of "Tree" data structure

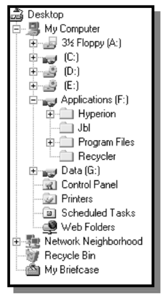
Tree

- In a logic view, it's actually an upside-down tree
- Usually used in representing hierarchy or relationship
  - e.g. File directories
- Root
  → branches
  → leaves
  - No re-convergence



Unix or DOS/Windows file system

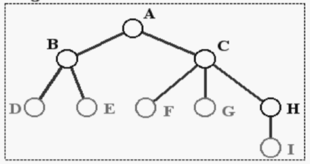




Prof. Yen, "Data Structure" class

TERMINOLOGY

- *A* is the ***root*** node.
- *B* is the ***parent*** of D and E.
- *C* is the ***sibling*** of B
- *D* and *E* are the ***children*** of B
- *D, E, F, G, I* are *external nodes*, or *leaves*
- *A, B, C, H* are ***internal nodes***
- The ***depth*** (***level***) of *E* is *2*
- The ***height*** of the tree is *3*
- The ***degree*** of node *B* is *2*

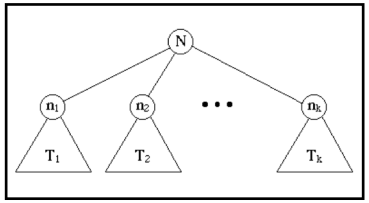


Property: (*#edges*) = (*#nodes*) - 1

Prof. Yen, "Data Structure" class

Definition of a Tree

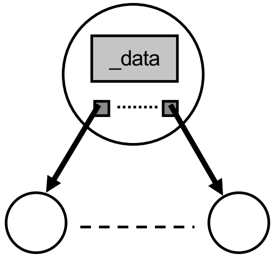
- This definition is "recursive" and "constructive".
  1) A single node is a tree. It is "root."
  2) Suppose N is a node and T_1, T_2, ..., T_k are trees with roots n_1, n_2, ...,n_k, respectively. We can construct a new tree T by making N the parent of the nodes n_1, n_2, ..., n_k. Then, N is the root of T, and T_1, T_2, ..., T_k are subtrees.



Prof. Yen, "Data Structure" class

Trees Implementation (1)

```
struct TreeNode
{
   MyClass     _data;
   TreeNode*   _child1;
   TreeNode*   _child2;
      .
      .
   TreeNode*   _childm;
};
```



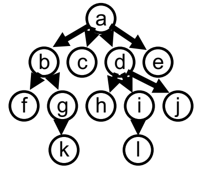
- Straightforward implementation
- Mem usage: d * n + 8 * (n * m)
- Problem
  - ➔ Not flexible in number of children
  - ➔ Good for fixed number of children (e.g. Binary Tree)

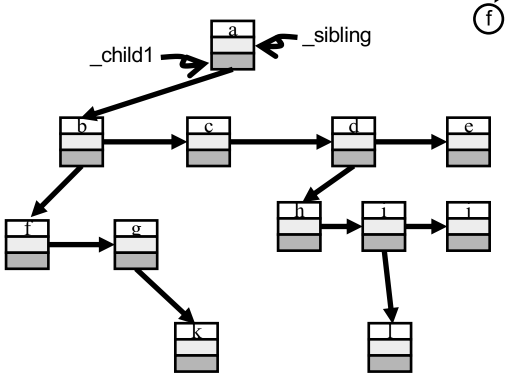
Trees Implementation (2)

```
struct TreeNode
{
   MyClass     _data;
   TreeNode*   _child1;   // head to a list
   TreeNode*   _sibling;  // head to a list
};
```

- Flexible in number of children
- Save memory?
  - Mem usage: d * n + 8 * 2n
- Problem
  - Not straightforward in interpretation
  - Not friendly in child and sibling traversal

Data Structure View





Prof. Yen, “Data Structure” class

Trees Implementation (3)

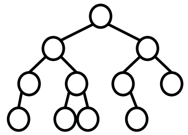
```
class TreeNode
{
   MyClass               _data;
   Array<TreeNode *>     _children;
};
```

- Straightforward view
- Flexible in number of children
- Mem usage: d * n + 8 * (3n – 1) (why?)
- Problem
  - Not easy to access siblings (but is that really a problem?)

Trees Implementation (4)

```
template <class T>
class TreeNode
{
   T                       _data;
   Array<TreeNode<T> *>    _children;
};

template <class T>
class Tree
{
   TreeNode<T>*    _root;
};
```

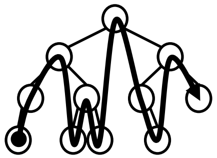

Traversal of Trees



1. Preorder: Process the node, then recursively process the left and right subtrees.
2. Inorder: Process the left subtree, the node, and the right subtree. 🡐 for binary tree
3. Postorder: Process the left subtree, the right subtree, and the node.
4. Levelorder: top-to-bottom, left-to-right order

Prof. Yen, “Data Structure” class

Tree Traversal: InOrder

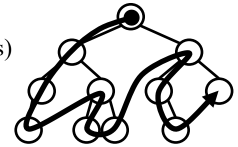


In Order is easily described recursively:
- Visit left subtree (if there is one) In Order
- Visit root
- Visit right subtree (if there is one) In Order

```
algorithm inOrder(TreeNode t)
   Input: a tree node (can be considered to be a tree)
   Output: None.

   if t has a left child
      inOrder(left child of t)
   Visit node t
   if t has a right child
      inOrder(right child of t)
```


Tree Traversal: PreOrder

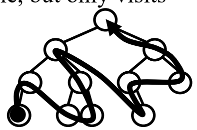
Another common traversal is PreOrder.
It goes as deep as possible (visiting as it goes)
- Visit root
- Visit left subtree in PreOrder
- Visit right subtree in PreOrder



```
algorithm preOrder(TreeNode t)
   Input: a tree node (can be considered to be a tree)
   Output: None.

   Visit node t    // Numbering, action, etc
   if t has a left child
      preOrder(left child of t)
   if t has a right child
      preOrder(right child of t)
```


Tree Traversal: PostOrder

PostOrder traversal also goes as deep as possible, but only visits internal nodes during backtracking.
- Visit left subtree in PostOrder
- Visit right subtree in PostOrder
- Visit root

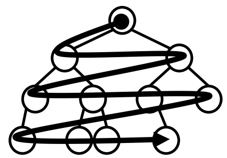


```
algorithm postOrder(TreeNode t)
   Input: a tree node (can be considered to be a tree)
   Output: None.

   if t has a left child
      postOrder(left child of t)
   if t has a right child
      postOrder(right child of t)
   Visit node t
```


LevelOrder Traversal

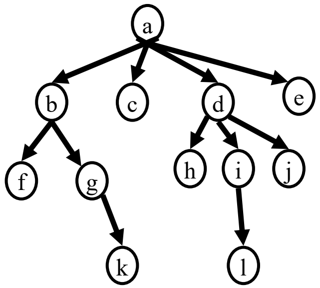
How to prove the correctness of this algorithm?

```
algorithm levelOrder(TreeNode t)
   Input: a tree node (can be considered to be a tree)
   Output: None.

   Let Q be a Queue
   Q.enqueue(t)
   while the Q is not empty
      n = Q.dequeue()
      Visit node n
      if n has a left child
            Q.enqueue(left child of n)
      if tree has a right child
            Q.enqueue(right child of n)
```



Example:
Level-Order Queue

Process	Enqueue	Q
	a	a
a	b,c,d,e	bcde
b	f,g	cdefg
c		defg
d	h,i,j	efghij
e		fghij
f		ghij
g	k	hijk
h		ijk
i	l	jkl
j		kl
k		l
l		



Prof. Yen, "Data Structure" class

Not just learn. Use smartly...

- Which kind of tree traversal to use?
  - Pre-order
  - Post-order
  - In-order
  - Level-order

Quiz!!

- Given a tree and we like to perform tree traversal, numbering from 1 to n...

1. If the number in a node should be smaller than its children (i.e. patent first)...
   ➔ Pre-order traversal

2. If the number in a node should be greater than its children (i.e. children first)...
   ➔ Post-order traversal

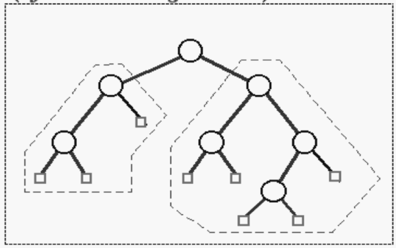
Do we need the "_sibling" field?

- For tree traversal, NO.
  All we need to know is "_children".
- How about other functions?
  - Insert?
  - Erase?

  Where?????
  ➔ How do these functions operate in a Tree?
- What if TreeNodes need to be sorted?
  ➔ We will discuss general sorted tree later.
- We will look at one special kind of sorted tree
  --- Binary Tree first

Binary Trees

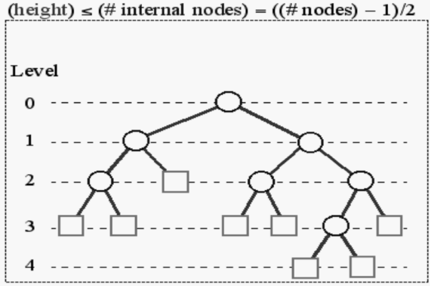
- ***Ordered tree:*** the children of each node are ordered.
- ***Binary tree:*** ordered tree with all internal nodes of *degree* 2.
- Recursive definition of binary tree:
- A ***binary tree*** is either
  - an external node (leaf), **or**
  - an internal node (the ***root***) and two binary trees (***left subtree*** and ***right subtree***)



Prof. Yen, “Data Structure” class

Properties of Binary Trees

- (# external nodes) = (# internal nodes) + 1
- (# nodes at level *i*) $\leq 2^{i}$
- (# external nodes) $\leq 2^{\text{(height)}}$
- (height) $\geq \log_2$ (# external nodes)
- (height) $\geq \log_2$ (# nodes) − 1
- (height) $\leq$ (# internal nodes) = ((# nodes) − 1)/2

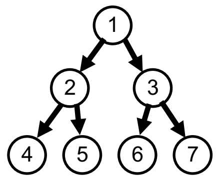


Special Binary Trees

1. Full / Complete binary tree
2. Binary Search Tree (BST)
3. (Balanced) Binary Search Tree

Full Binary Tree

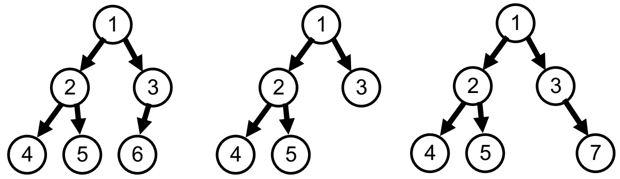
- A **full** binary tree of height h is a binary tree of height h having exactly $2^{(h+1)} - 1$ nodes
  - All external nodes have same depth = h
  - All internal nodes have non-empty left and right children



height = 2
#nodes = 7

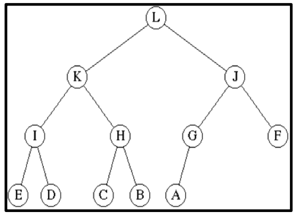
Complete Binary Tree

◆ A **complete** binary tree is a special case of a binary tree, in which all the levels, except perhaps the last, are full; while on the last level, any missing nodes are to the right of all the nodes that are present.

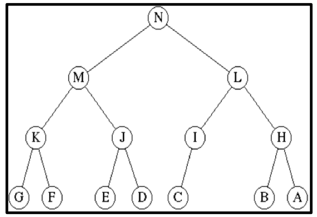


yes yes no

Complete/Full Binary Tree Example



- •Is this a full binary tree?
  - •No – not all leaf nodes are at the same level.
  - •No - node G has an empty right and a non-empty left.
- •Is it a complete binary tree?
  - •Yes



- •Is this a full binary tree?
  - •No - node I has an empty right and a non-empty left.
- •Is it a complete binary tree?
  - •No – the final level is not completed in left-to-right fashion

Prof. Yen, "Data Structure" class

Binary Tree Implementation (1)

◆ Since the number of children of a binary tree is fixed, we can implement it as

```
template <class T>
class BinaryTreeNode
{
   T                     _data;
   BinaryTreeNode<T>*    _left;
   BinaryTreeNode<T>*    _right;
};
template <class T>
class BinaryTree
{
   BinaryTreeNode<T>*    _root;
};
```

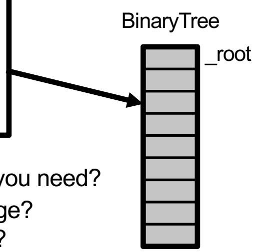
➔ **Memory usage: d*n + 8 * 2n**

Memory Usage Consideration

◆ An observation ---

- In 64-bit or higher platform, the memory usage of pointer variables is bigger than that of (unsigned) integers. So we can use "indices" for the child nodes, instead of pointers. That is,

```
class BinaryTreeNode {
   T          _data;
   unsigned   _left;
   unsigned   _right;
};
```

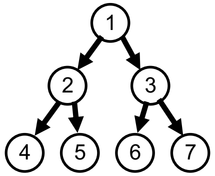


- What do you think?
- What extra data structure do you need?
- What is the total memory usage?
- What if some node is deleted?

Binary Tree Implementation (2)

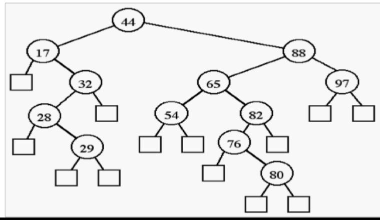
- If a binary tree is complete
  ➔ Use array for implementation

- Let the height of the tree = ‘h’
  - #nodes must >= 2^h and <= $2^{(h+1)} – 1$
  - root has index = 1
  - A node with index t
    - index of left child = 2t
    - index of right child = 2t + 1
    - Index of parent = t / 2



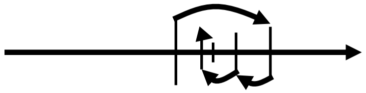
Binary Search Trees (BST)

- A binary search tree is a binary tree T such that
  - each internal node stores an item (k, e) of a dictionary.
  - keys stored at nodes in the left subtree of v are less than or equal to k.
  - keys stored at nodes in the right subtree of v are greater than or equal to k.
  - external nodes do not hold elements but serve as place holders.



Prof. Yen, “Data Structure” class

Search in BST



- A binary search tree T is a ***decision tree***, where the question asked at an internal node v is whether the search key k is less than, equal to, or greater than the key stored at v.

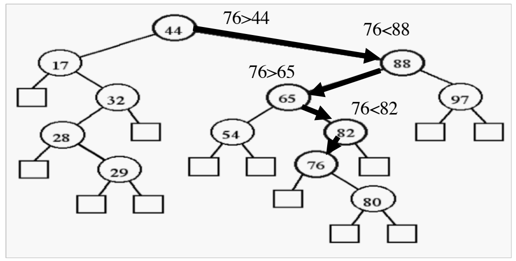
```
if v is an external node then
   return  v   // mean the key should be inserted here
if k = key(v) then
   return  v   // find a match
else if k < key(v) then
   return TreeSearch(k, T.leftChild(v))
else { k > key(v) }
   return TreeSearch(k, T.rightChild(v))
```

Prof. Yen, “Data Structure” class

Search Example I

What is the running time ?

Successful findElement(76)



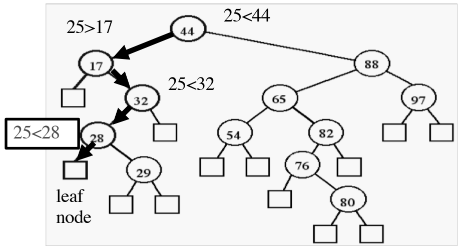
- A successful search traverses a path starting at the root and ending at an internal node

Prof. Yen, “Data Structure” class

Search Example II

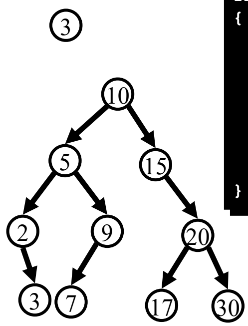
Unsuccessful findElement(25)

What is the running time ?



- An unsuccessful search traverses a path starting at the root and ending at an external node

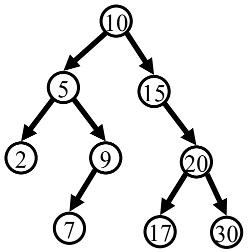
Insert a Key




```
TreeNode insert(int x, TreeNode T)
{ if  ( T == NULL )
      return new TreeNode(x,null,null);
  if (x == T.Element)
      return T;
  if (x < T.Element)
     T.Left = insert(x, T.Left);
  else T.Right = insert(x, T.Right);
  return T;
}
```

What is the running time ?

Delete a Key



How do you delete:

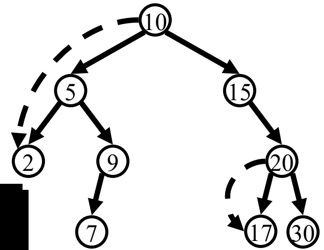
17 ?

9 ?

20 ?????

Let’s look at two basic operations:
min() and successor() first!!

FindMin




```
TreeNode min(Node T) {
  if (T.Left == NULL)
    return T;
  else
    return min(T.Left); }
```

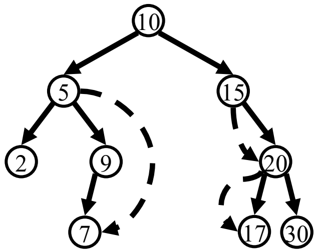
How many children can the min of a node have?

Successor

Find the next larger node in this node's subtree.

- Find "min" of the right child
  (What if no right child?)
- [Compare] second largest

```
TreeNode succ(TreeNode T) {
  if (T.right == NULL)
    return NULL;
  else
    return min(T.right);
}
```

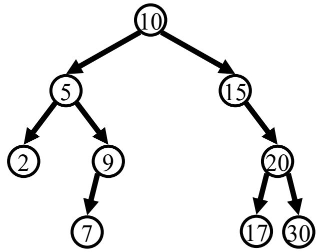


How many children can the successor of a node have?

Prof. Yen, "Data Structure" class

Deletion (1) - Leaf Case

Delete(17)

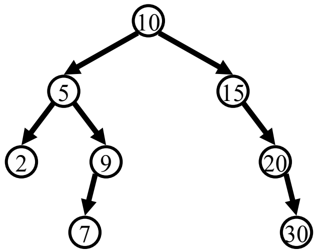


Trivial !!

Prof. Yen, "Data Structure" class

Deletion (2) - One Child Case

Delete(15)

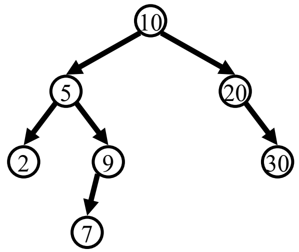


Trivial !!

Prof. Yen, "Data Structure" class

Deletion (3) - Two Children Case

Delete(5)



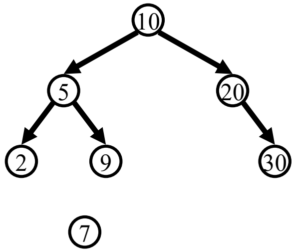
Replace node with value guaranteed to be between the left and right subtrees

➔ the successor

Prof. Yen, "Data Structure" class

Deletion (3) - Two Children Case

Delete(5)

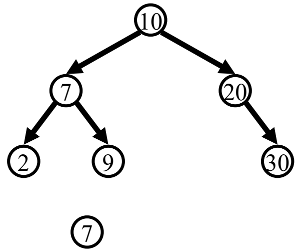


Always easy to delete the successor – always has either 0 or 1 children!

Prof. Yen, "Data Structure" class

Deletion (3) - Two Child Case

Delete(5)



Finally copy data value from deleted successor into original node

What is the cost of a delete operation ?

Prof. Yen, "Data Structure" class

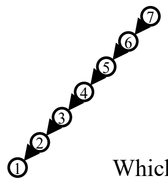
Cost of the Operations

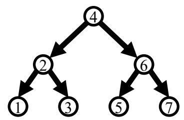
- find, insert, delete : time = O(height(T))
- Need to compute height(T)
- For a tree T with n nodes:
  - height(T) $\leq$ n
  - height(T) $\geq \log_2(n)$ (why ?)

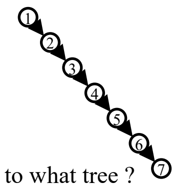
Prof. Yen, "Data Structure" class

Height of the Binary Search Tree

- Height depends critically on the order in which we insert the data:
  - E.g. 1,2,3,4,5,6,7 or 7,6,5,4,3,2,1, or 4,2,6,1,3,5,7







Which insertion order corresponds to what tree ?
Which tree do we prefer and why ?

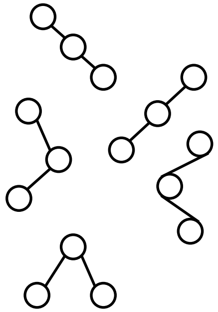
Prof. Yen, "Data Structure" class

Random Input vs. Random Trees

For three items, the shallowest tree is twice as likely as any other – effect grows as *n* increases. For *n*=4, probability of getting a shallow tree > 50%

Inputs
1,2,3
3,2,1
1,3,2
3,1,2
2,1,3
2,3,1

Trees



Prof. Yen, "Data Structure" class

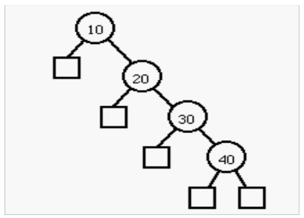
Average cost

- The *average, amortized* cost of n insert/find operations is O(log(n))
- But the *average, amortized* cost of n insert/find/delete operations can be as bad as sqrt(n)
  - log 10000 vs. sqrt(10000)
  - Deletions make life harder
  - Read the book for details
- Need guaranteed cost O(log n)

Prof. Yen, "Data Structure" class

Time Complexity

- The height of binary search tree is n in the worst case, where a binary search tree looks like a sorted sequence



- To achieve good running time, we need to keep the tree ***balanced***, i.e., with O(logn) height.

Self Adjusting Binary Search Trees

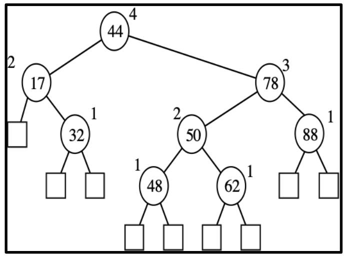
- Insertions/removals may "deepen" and "unbalance" a binary search tree.
- Self-adjusting binary search trees automatically restore balance after each insertion/removal by performing a series of ***rotations***.
- Self-adjusting binary search trees **insure** good worst-case performance.

Balanced Binary Search Trees

Prof. Yen, "Data Structure" class

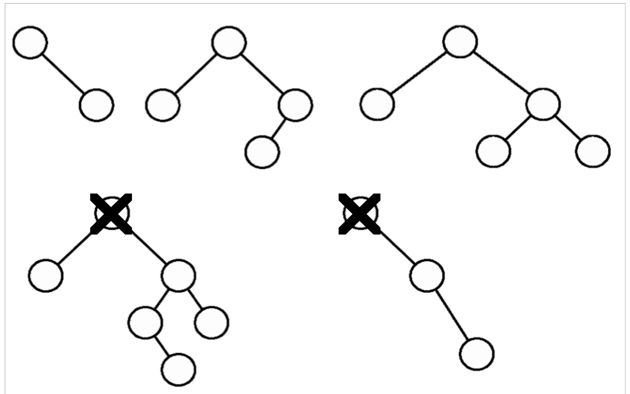
AVL Tree

- G. M. Adel'son-Vel'skii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Math. Doklady* **3** (1962), pp. 1259—1262.
- **AVL trees are balanced.**
- An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1.*



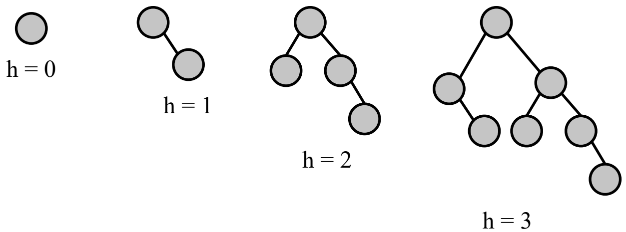
An example of an AVL tree where the heights are shown next to the nodes:

Example



Prof. Yen, "Data Structure" class

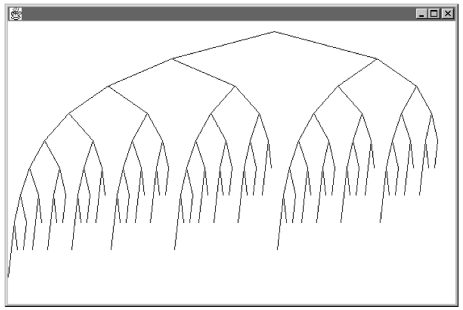
AVL Trees



Given an AVL tree of height h, what is the minimal number of nodes that the tree may contain?

Prof. Yen, "Data Structure" class

AVL - height 9



Prof. Yen, "Data Structure" class

Height Of An AVL Tree

The height of an AVL tree that has n nodes is at most 1.44 $\log_2 (n+2)$.

The height of every n node binary tree is at least $\log_2 (n+1)$.

$$\log_2 (n+1) \le \text{height} \le 1.44 \log_2 (n+2)$$

$\underline{O(\log(n))}$

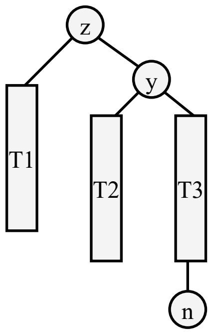
Prof. Yen, "Data Structure" class

Insertion

- A binary search tree T is called ***balanced*** if for every node v, the height of v's children differ by at most one.
- Inserting a node into an AVL tree involves performing an expandExternal(w) on T, which changes the heights of some of the nodes in T.
  - If an insertion causes T to become unbalanced, we travel up the tree from the newly created node until we find the first node x such that its grandparent z is unbalanced node.
  - Since z became unbalanced by an insertion in the subtree rooted at its child y, height(y) = height(sibling(y)) + 2
  - Now to rebalance...

Prof. Yen, "Data Structure" class

The "outie" case



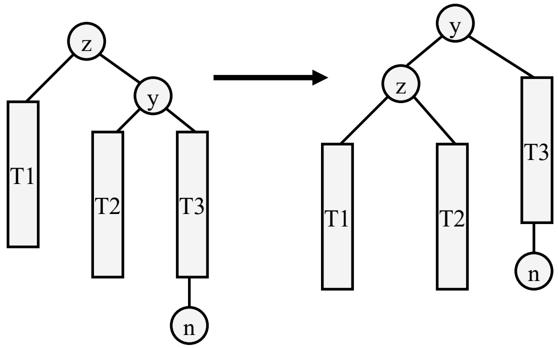
- z - the nearest ancestor which is out of balance
- n - the newly inserted node
- height of T1, T2, and T3 are all the same, say *h*

Prof. Yen, "Data Structure" class

single rotation

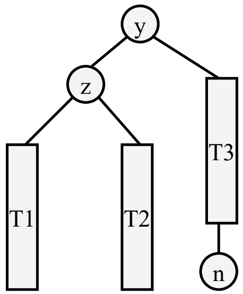
Rotate at 'z→y':

1. y is z's right child → find y's left child T2
2. Make T2 z' new right child, and z y's new left child.



Prof. Yen, "Data Structure" class

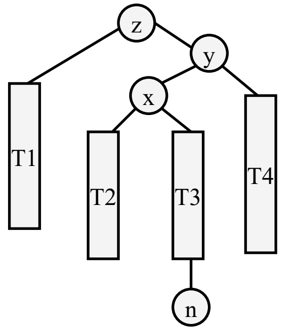
After the rotation



- y is now the root
- the height of the tree is the same as it was before inserting the node, so no other ancestor is unbalanced
- the root y is balanced

Prof. Yen, “Data Structure” class

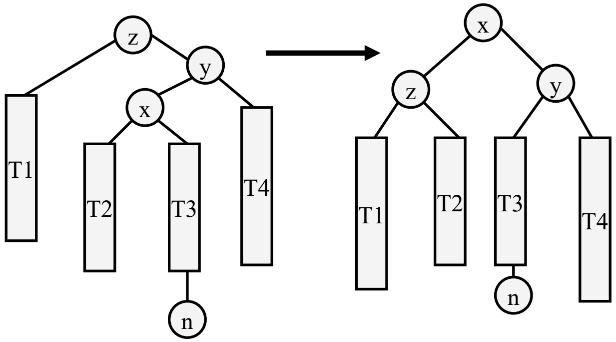
The “innie” case



- z is the nearest out-of-balance ancestor
- T1 and T4 have height *h*
- T2 and T3 have height *h*-1
- n is the newly inserted node - either in T2 or T3

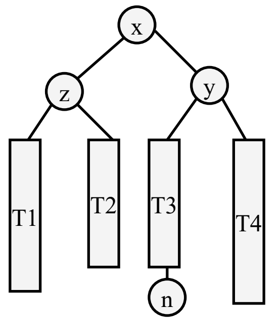
Prof. Yen, “Data Structure” class

Double Rotation



Prof. Yen, “Data Structure” class

After the Rotation

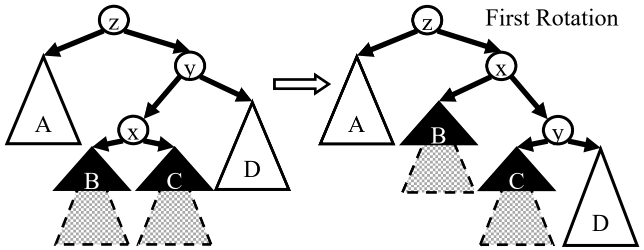


- x is now the root with left child z and right child y
- The height of the tree is the same as before the insertion, so no other ancestor is now out-of-balance
- This tree is balanced

Prof. Yen, “Data Structure” class

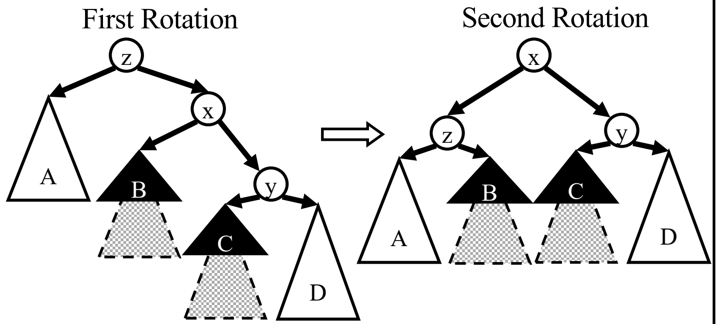
More precisely

Rotate at ‘y→x’:
1. x is y’s left child → find x’s right child C
2. Make C y’ new left child, and y x’s new right child.



Prof. Yen, “Data Structure” class

Double Rotation Completed



Prof. Yen, “Data Structure” class

The other rotations

- These two demonstrations show the Single Left rotation and the Double Left rotation (used when the nearest out-of-balance ancestor is too heavy on the right)
- Similar rotations are performed when the nearest out-of-balance ancestor is heavy on the left -- these are called Single Right and Double Right Rotations

Prof. Yen, “Data Structure” class

Deletion from an AVL Tree

- Deletion of a node from an AVL tree requires the same basic ideas, including single and double rotations, that are used for insertion
- We are NOT going into details here.... (Don’t need to memorize the steps; understand the principles!!)
  - Please refer to any DS books or the appendix slides at the end

Building an AVL Tree

Input: sequence of n keys (unordered)

19 3 4 18 7

Insert each into initially empty AVL tree

$$\boxed{\sum_{i=1}^{n}\log i \le \sum_{i=1}^{n}\log n = O(n\log n)}$$

But, suppose input is already sorted ...

3 4 7 18 19

Can we do better than O(*n* log *n*)?

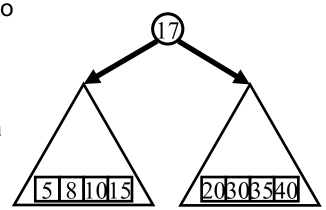
Prof. Yen, "Data Structure" class

AVL BuildTree

5	8	10	15	17	20	30	35	40

Divide & Conquer

- Divide the problem into parts
- Solve each part recursively
- Merge the parts into a general solution

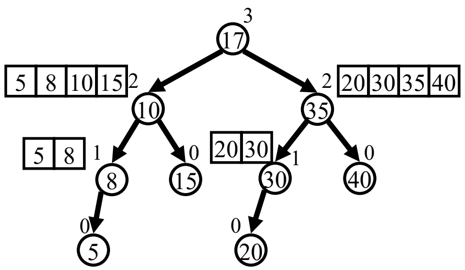


How long does divide & conquer take?

Prof. Yen, "Data Structure" class

BuildTree Example

5	8	10	15	17	20	30	35	40



Prof. Yen, "Data Structure" class

Thinking About AVL

Observations

+ Worst case height of an AVL tree is about 1.44 log n
+ Insert, Find, Delete in worst case O(log n)
+ Only one (single or double) rotation needed on insertion
+ Compatible with lazy deletion
- O(log n) rotations needed on deletion
- Height fields must be maintained (or 2-bit balance)

Coding complexity?

Prof. Yen, "Data Structure" class

AVL Performance

Method	Worst Case
void insert(Comparable element)	O(Log N)
boolean contains(Comparable element)	O(Log N)
void delete(Comparable element)	O(Log N)
int size()	O(1)
boolean isEmpty()	O(1)

Prof. Yen, "Data Structure" class

Pros and Cons of AVL Trees

Pro:

- All operations guaranteed O(log N)
- The height balancing adds no more than a constant factor to the speed of insertion

Con:

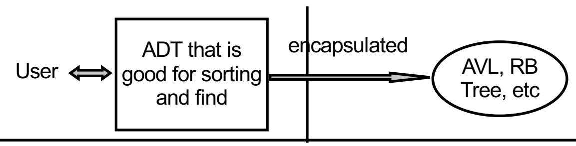
- Space consumed by height field in each node
- Slower than ordinary BST on random data

Can we guarantee O(log N) performance with less overhead?

Prof. Yen, "Data Structure" class

One more note about the Tree ADT

- A good DS for representing hierarchy or relationship
- Important variations: binary tree, binary search tree, balanced binary search tree
- Balanced Binary Search Tree
  - All operations are equal or less than O(log(n))
  - Good example for "Abstract" DT



Alternatives to AVL Trees

- Weight balanced trees
  - keep about the same number of nodes in each subtree
  - not nearly as nice
- Others
  - Splay trees
  - 2-3-4 trees
  - red-black trees
  - B-Tree
  - ➔ Will be covered in "Tree Part II" later!

Prof. Yen, "Data Structure" class

Appendix Slides

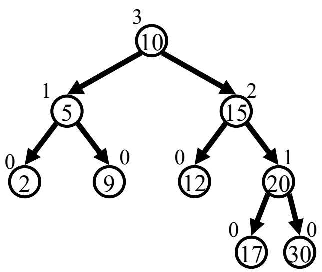
Steps in deleting X in an AVL tree

- Reduce the problem to the case where X has only one child
- Delete the node X. The height of the subtree formerly rooted at X has been reduced by one
- We must trace the effect on the balance from X all the way back to the root until we reach a node which does not need adjustment

Prof. Yen, "Data Structure" class

Easy Insert

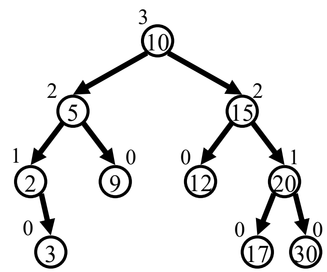
Insert(3)



Prof. Yen, "Data Structure" class

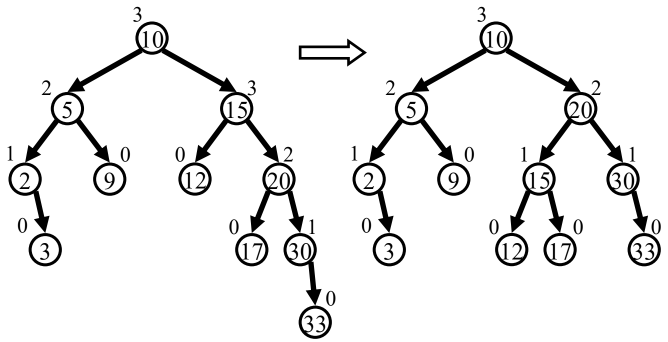
Hard Insert (Bad Case #1)

Insert(33)



Prof. Yen, "Data Structure" class

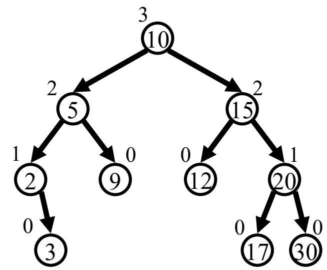
Single Rotation



Prof. Yen, “Data Structure” class

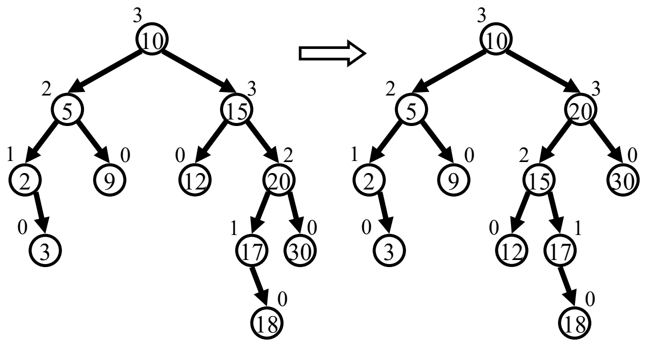
Hard Insert (Bad Case #2)

Insert(18)



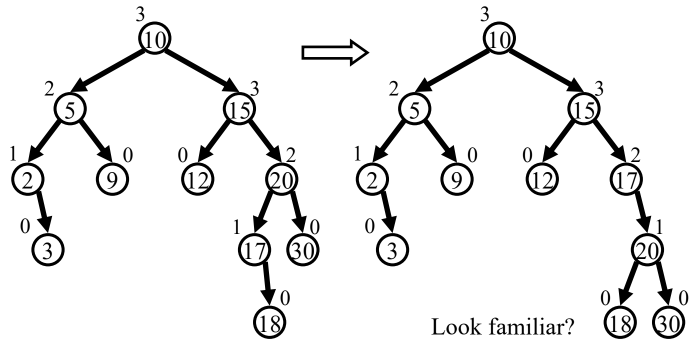
Prof. Yen, “Data Structure” class

Single Rotation (oops!)



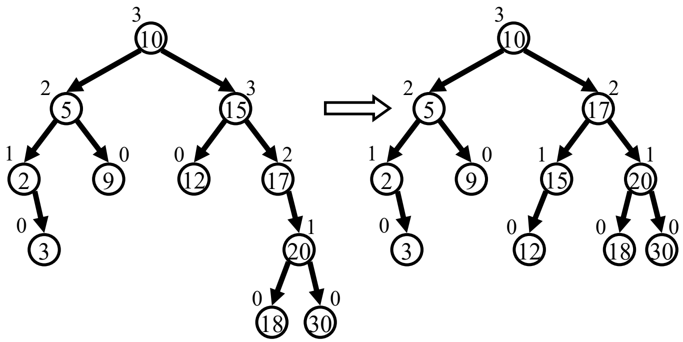
Prof. Yen, “Data Structure” class

Double Rotation (Step #1)



Prof. Yen, “Data Structure” class

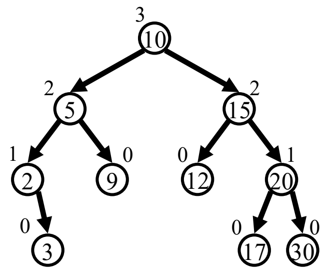
Double Rotation (Step #2)



Prof. Yen, “Data Structure” class

Deletion: Really Easy Case

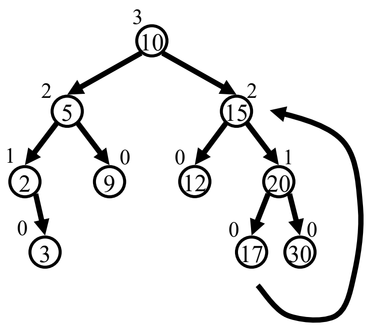
Delete(17)



Prof. Yen, “Data Structure” class

Deletion: Pretty Easy Case

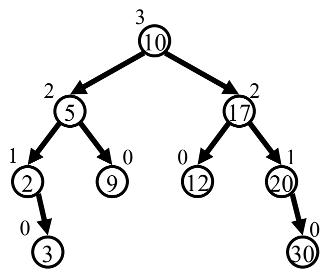
Delete(15)



Prof. Yen, “Data Structure” class

Deletion: Pretty Easy Case (*cont.*)

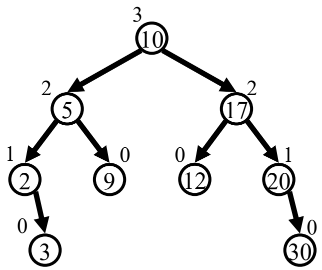
Delete(15)



Prof. Yen, “Data Structure” class

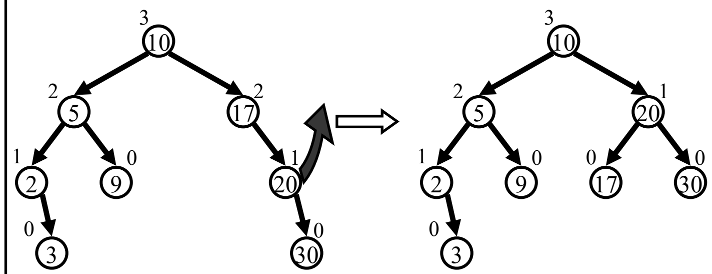
Deletion (Hard Case #1)

Delete(12)



Prof. Yen, “Data Structure” class

Single Rotation on Deletion

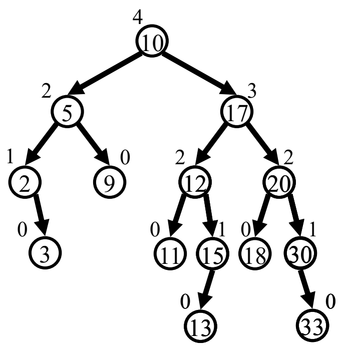


Deletion can differ from insertion – ***How?***

Prof. Yen, “Data Structure” class

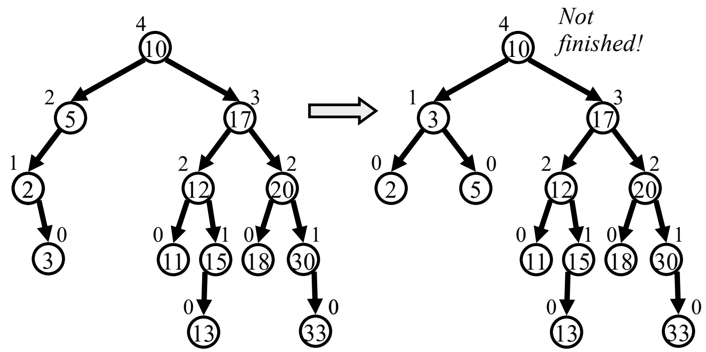
Deletion (Hard Case)

Delete(9)



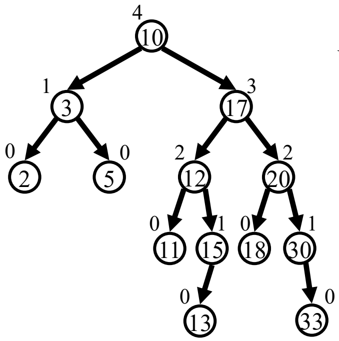
Prof. Yen, “Data Structure” class

Double Rotation on Deletion



Prof. Yen, “Data Structure” class

Deletion with Propagation

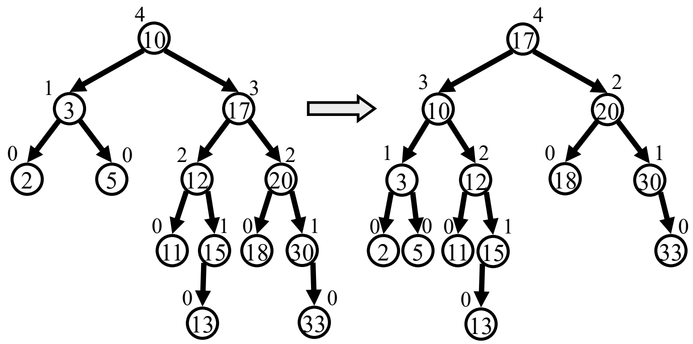


What different about this case?

We get to choose whether to single or double rotate!

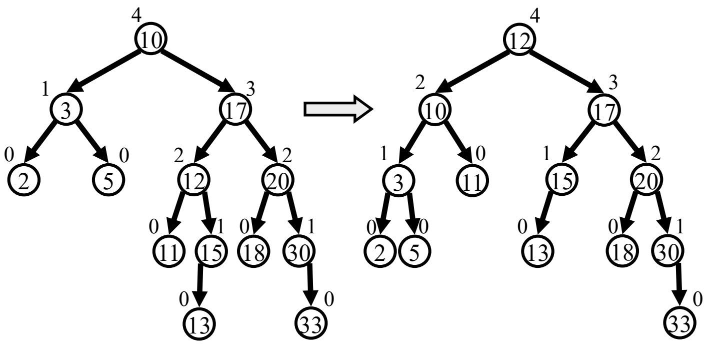
Prof. Yen, “Data Structure” class

Propagated Single Rotation



Prof. Yen, “Data Structure” class

Propagated Double Rotation



Prof. Yen, “Data Structure” class