NOTES FOR COP 3530
EXAM II - SUMMER 2002
Michael Knopf
mknopf@ufl.edu

Hashing:

- Always use an odd divisor to prevent odd/even home bucket bias
- Use larger prime numbers to prevent bias or choose a divisor that has no prime factor smaller than 20, also make sure that it is odd
- Collision: when two keys map to the same home bucket
- **Overflow**: when the home bucket for a new pair (key, element) is full
  - *Eliminating overflow*: use Array Linear List or Chains to allow for more than one (key, element) pair to be stored in it's home bucket
- Searching:
  - *Linear probing:* searching through the table for the next available bucket that is to the right of the home bucket. Worst-case performance is O(n), this happens when all pairs are in the same cluster.
    - Alpha = Loading density = the # of pairs / the size of the table
    - Expected performance
      - for successful search = 0.5(1+1/(1-alpha))
      - for unsuccessful search = 0.5(1+1/(1-alpha)^2)
      - then take the minimum of the two as your alpha
    - Alpha <=0.75 is recommended
  - Quadratic probing:
  - Random probing:

Data compression:

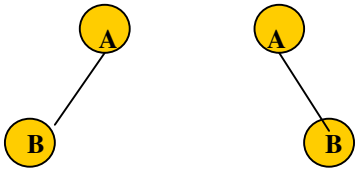
- Compression ratio = original data size / compressed data size
- Reduces time to receive and transmit data
- Loss less compression: when no data whatsoever is lost during compression and decompression (zip)
- Lossy compression: when data IS lost during compression and decompression (jpeg)
- LZW compression:
  - Character sequences in the original text are replaced by codes that are dynamically determined
  - The code table is not encoded into the compressed text, because it may be reconstructed from the compressed text during decompression
  - *Example:* **compression**: where the letters in the text are limited to {a, b}
    - The characters in the alphabet are assigned code numbers beginning at zero
    - The initial code is: code: a b
      key: 0 1
    - Original text = abababbabaabbabbaabba
    - Compression is done by scanning the original text from left to right.
    - Find longest prefix p for which there is a code in the code table.
    - Represent p by its code pCode and assign the next available code number to pc, where c is the next character in the text that is to be compressed.

- *Example:* decompression:
  - The initial code is: code: a b
    Key: 0 1
  - Where the compressed data is 012233588
  - We set is up just like with compression
    - p = a pCode = 0 lastP = --

decompressed text = a
    - then move on to the next code which is 1
    - p = b
      pCode = 1
      lastP = a
      Decompressed text = ab
      add lastP to the first character of p and enter it into the table under the next number
      The table is now: code: 0 1 2
      key: a b ab
      If you encounter a code that is not yet in the table then enter it into the table as The last value of p with the first letter of p added to the end: so if we encounter 8 and the last number (IN THE COMPRESSED CODE - NOT THE TABLE) was 5 and 5 = abb then 8 will become abb + a = abba, enter this into the table.

Trees:

- Root is at level 1
- Tree is broken up into sub-trees that lay under the root
- Height = depth = the number of levels
- Node degree = the number of children it has (**not** grandchildren **or** great grandchildren, etc...)
- Tree degree = the maximum node degree
- Binary tree:
  - Finite (possibly empty) collection of elements
  - A non-empty binary tree has a root element
  - The remaining elements (if any) are partitioned into TWO binary trees which are referred to as the left and right sub-trees of the binary tree
  - The differences between a tree and a binary tree:
    - *No node in a binary tree may have a degree greater than 2*, where as there is no limit on the degree a node can have in a tree
    - A binary tree may be empty, a tree cannot be empty
    - The sub-trees of a binary tree are ordered, those of a tree are not ordered



These are different when viewed as a binary tree, but they would be considered the same for a tree.

Arithmetic expressions:

- (a + b) * (c + d) + e – f/g*h + 3.25
- Expressions comprise three kinds of entities.
  - Operators (+, -, /, *).
  - Operands (a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc.).

- Delimiters ((,)).

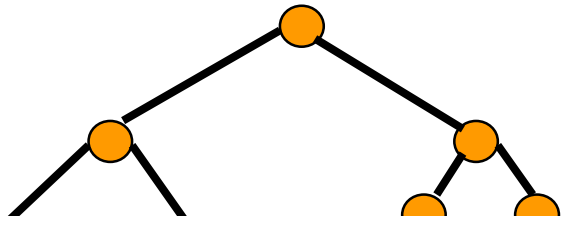
- Operator degree:
  - The number of operands an operator requires
  - Binary operators require to operands:
    - A + B
    - C / D
    - E – F

    - **Infix form:** Operators come between their left and right operands

  - Unary operators require one operand
    - + g
    - - h
  - **Postfix form**:
    - The form of a variable or constant is the same as in infix form
    - The relative order of the operands is also the same as in infix form
    - Operators come immediately after the operands in postfix form:
      Infix: a + b
      Postfix: ab+
    - *Examples*:
      - Infix: a + b * c
        Postfix: a b c * +
      - Infix: a * b + c
        Postfix: a b * c +
      - Infix: (a + b) * (c – d) / (e + f)
        Postfix: a b + c d - * e f + /

    Think of it like this: the first 2 operands and the first operator make up a single operation. So ab+ can be thought of as a + b, then the next 2 operands and the next operator, cd- becomes c – d. Now we have the * so its (a + b) (c – d) * which can be thought of as (a + b) * (c – d) etc...

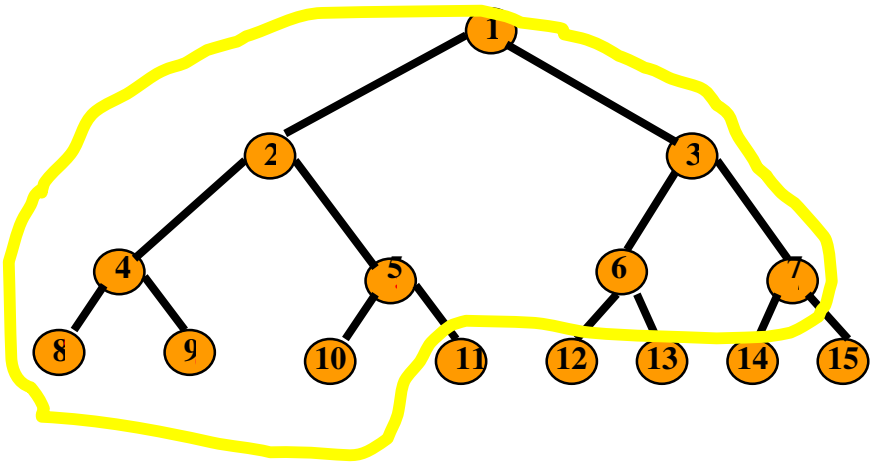
    - Scan postfix expression from left to right pushing operands on to a stack
    - When an operator is encountered, pop as many operands as this operator needs; evaluate the operator; push the result on to the stack.
    - This works because in postfix operators come immediately after there operandsThe operands are stored on the stack until an operator is encounter. At that time the operands are popped from the stack, the operation performed, and the result stored back onto the stack, this continues until the final result is computed
  - **Prefix form**:
    - The form of the variable or constant is the same as infix form
    - The relative order of operands is the same as in infix form
    - Operators come immediately before there operands
      Infix: a + b – c
      Postfix: a b + c –
      Prefix: + a b – c
  - **Binary tree form**: (a + b) * (c – d) / (e + f)



- **Merits of binary tree form:**
  - Left and right operands are easy to visualize
  - Code optimization algorithms work with the binary tree form of an expression
  - *Simple recursive evaluation of an expression:* Evaluate an expression– if there is no root, return null; if the root is a leaf, return the value of variable or constant in the leaf; otherwise, evaluate the left and right operands recursively; compute the root operator; return the result.

Binary tree properties and their representations:

- The minimum number of nodes is equal to the height h
- A pointer from the parent node to the child node represents each edge in the drawing of a tree
- A **full binary tree** of height h contains exactly $2^h - 1$ elements
- The maximum number of nodes is $2^h - 1$
- Let n be the number of nodes in a binary tree of height h:
  - $h <= n <= 2^h - 1$
  - $\log_2(n+1) <= h <= n$
- Nodes are numbered starting at 1 (the root) to $2^h - 1$
- Number by levels from top to bottom
- Within a level number from left to right
- The parent of node i is i/2 unless i = 1
- The left child of node i is node 2i, unless 2i > n, where n is the number of nodes
- If 2i > n, node i has no left child
- The right child of node i is node 2i + 1, unless 2i + 1 > n, where n is the number of nodes
- If 2i + 1 > n, node i has no right child
- A **complete n node binary tree** is defined by the nodes numbered 1 through n the following is a complete binary tree with n = 10. The height of a complete binary tree containing n elements is $\log_2 (n+1)$



- Binary trees can be represented as arrays or linked lists. *The array based is only useful when the number of missing elements is small*
- An n node binary tree needs an array whose length is between n + 1 and 2^n
- **Array representation:** where the node that is numbered i is stored in tree[i]

- **Linked representation**:
  - Each binary tree node is represented as an Object whose data type is BinaryTreeNode
  - The space required by an n-node binary tree is n * (the space required by one node)


```
public class BinaryTreeNode
{
   // package visible data members
   Object element;
   BinaryTreeNode leftChild;     // left subtree
   BinaryTreeNode rightChild;    // right subtree

   // constructors
   public BinaryTreeNode() {}

   public BinaryTreeNode(Object theElement)
      {element = theElement;}

   public BinaryTreeNode(Object theElement,
                         BinaryTreeNode theleftChild,
                         BinaryTreeNode therightChild)
   {
       element = theElement;
       leftChild = theleftChild;
       rightChild = therightChild;
   }

   // accessor methods
   public BinaryTreeNode getLeftChild() {return leftChild;}
   public BinaryTreeNode getRightChild() {return rightChild;}
   public Object getElement() {return element;}

   // mutator methods
   public void setLeftChild(BinaryTreeNode theLeftChild)
      {leftChild = theLeftChild;}
   public void setRightChild(BinaryTreeNode theRightChild)
      {rightChild = theRightChild;}
   public void setElement(Object theElement)
      {element = theElement;}

   // output method
   public String toString()
      {return element.toString();}
}
```

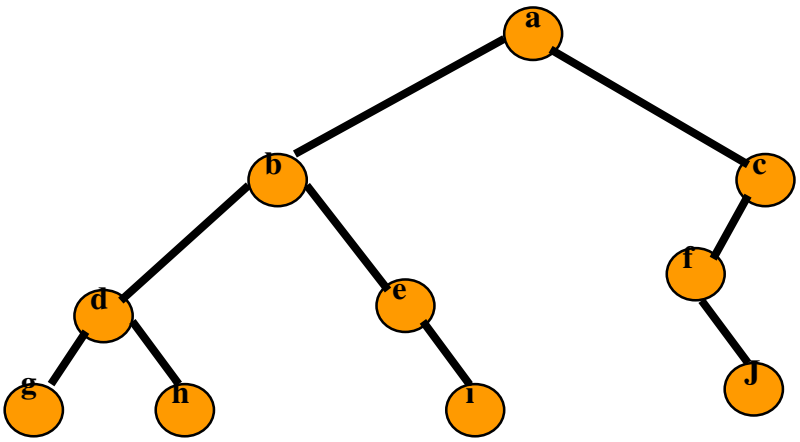
  - Some binary tree operation you should know:
    - Determine the height
    - Determine the number of nodes
    - Make a clone
    - Determine if two binary trees are clones
    - Display the binary tree
    - Evaluate the arithmetic expression represented by a binary tree

- Obtain the infix, prefix, and postfix forms of an expression

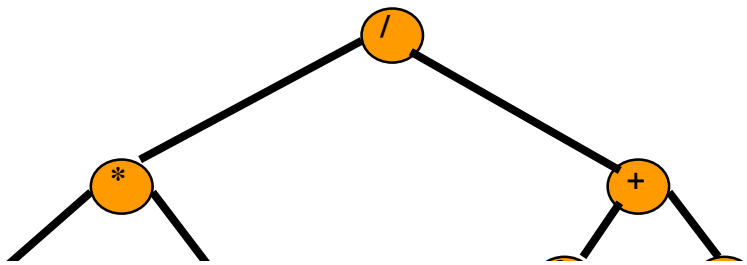
- **Binary tree traversals:**
  - Many operations are done by performing a traversal of the binary tree
  - In a traversal, each element of the binary tree is visited exactly once
  - During the visit of an element, all actions (make a clone, display the tree, evaluate an operator, etc...) with respect to this element is taken
  - There are four traversal methods: **Pre-order, In-order, Post-order, level-order**

  - ***Pre*-order**: the output of a traversal is called prefix form

```
public static void preOrder(BinaryTreeNode t)
{
    if (t != null)
    {
        visit(t);
        preOrder(t.leftChild);
        preOrder(t.rightChild);
    }
}
```

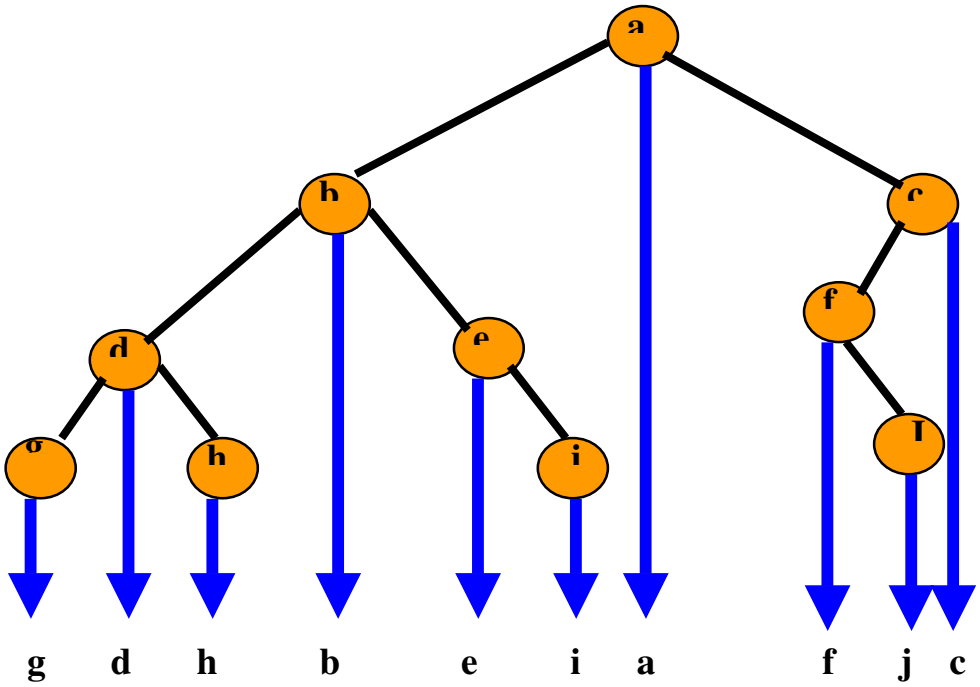


- If we traversed the above tree and visit = print then during the traversal, t starts at the root, moves to the left child b of the root, then to the left child d of b. When the traversal of the left subtree of b is complete, t, once again, points to the node b. The t moves into the right subtree of b. When the traversal of this right subtree is complete, t again points to b. Following this, t points to a. We see that t points to every node in the binary tree three times – once when you get to the node from its parent (or in the case of the root, t is initially at the root), once when you return from the left subtree of the node, and once when you return from the node's right subtree. Of these three times that t points to a node, the node is visited the first time.
- Below it the prefix form of the expression / * + a b – c d + e f



- ***In*-order**: the output of a traversal is called infix form

```
public static void inOrder(BinaryTreeNode t)
 {
      if (t != null)
      {
            preOrder(t.leftChild);
            visit(t);
            preOrder(t.rightChild);
       }
}
```



- ***post*-order**: the output of a traversal is called postfix form

```
public static void postOrder(BinaryTreeNode t)
 {
      if (t != null)
      {
            preOrder(t.leftChild);
            preOrder(t.rightChild);
            visit(t);
       }
 }
```

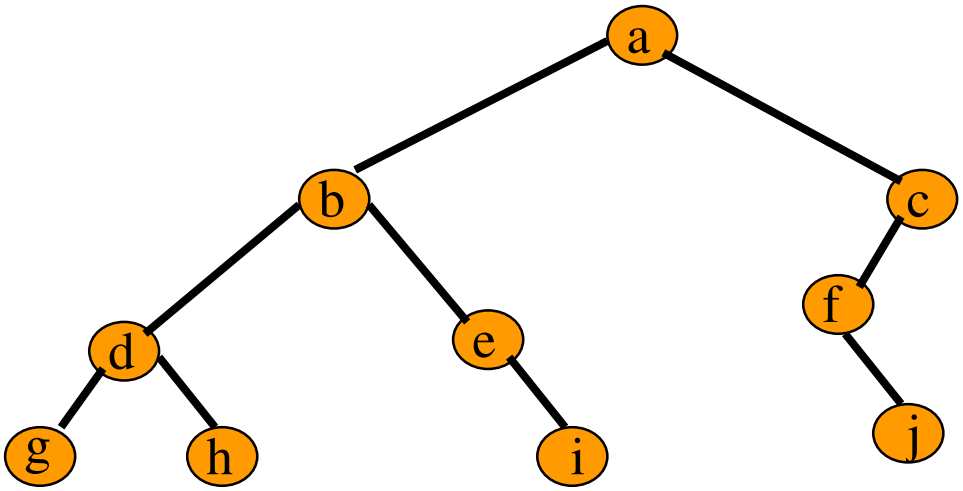
- **Traversal applications**:

- Make a clone using post-order traversal … clone the left sub-tree, clone the right sub-tree, clone the root in the visit step.
- Determine height using post-order traversal … determine the height of the left sub-tree, determine the height of the right sub-tree, in the visit step add 1 to the max of the already determined heights of the left and right subtrees.
- Determine number of nodes using preorder, in-order, or post-order traversal … initialize a counter to 0, add 1 to the counter in the visit step.

- ***Level*-order**:

```
Let t be the root of the tree;
While (t!= null)
{
  visit t and put its children on a FIFO queue;
  remove a node from the FIFO queue and call it t;
  // remove returns null when queue is empty
 }
```

The following is listed as: **a b c d e f g h i j**

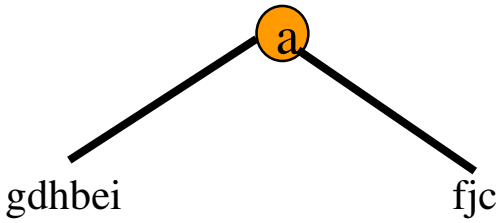


- **Binary tree construction:**
  - Suppose the elements in a binary tree are distinct
  - Can you construct the binary tree from which a given traversal sequence came?
  - When a traversal sequence has more than 1 element then the binary tree is **NOT** uniquely defined, therefore the tree from which the sequence was obtained cannot be reconstructed uniquely, in other words *we may not be able to conclude that a child of a node is the left child or the right child*.
  - If we are given two sequences can we reconstruct the tree? Pre-order and post-order DO NOT uniquely define a binary tree, neither does post-order with level-order, nor pre-order with level-order. *The only way we can do it is if we are given in-order and either post-order or pre-order.*

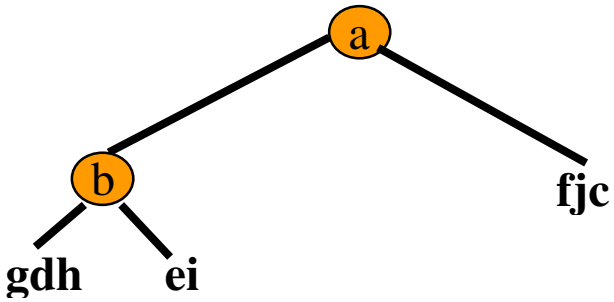
Example: If we are given the following sequences we can construct the binary tree:

In-order = gdhbeiafjc
Pre-order = abdgheicfj

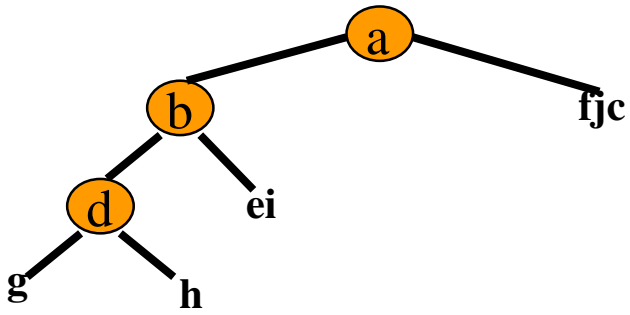
- Scan the pre-order left to right using the in-order to separate left and right sub-trees
- **a** is the root of the tree because it comes first in the pre-order, the left sub-tree then must consist of **gdhbei** and the right sub-tree must consist of **fjc**



- because **b** is next in the pre-order we know that it is the left child of **a**
- this allows us to break up the sequence **gdhbei** into left and right sequences



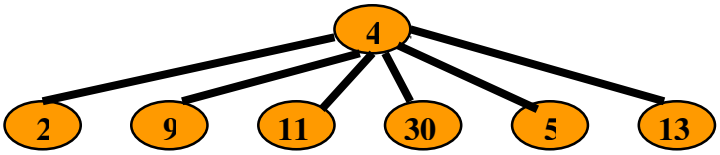
Which becomes the following:



Union-find problem:

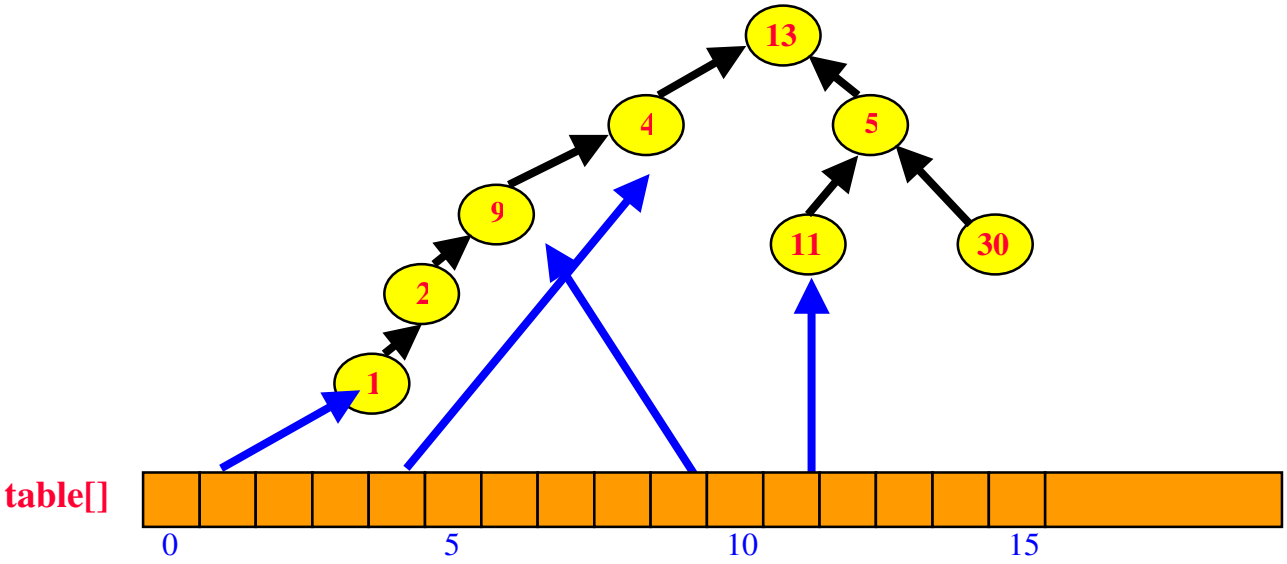
- A set consists of element that themselves are sets: so s = {1, 2, 3, 4, 5} is initially {1}, {2}, {3}, {4}, {5}
- **The weight rule**: if the number of nodes in the tree with root *i* is less than the number of nodes in the tree with root *j*, then make *i* a child of j; otherwise, make *j* a child of *i*.
- **The height rule:** if the height of tree *i* is less than that of tree *j*, then make *j* the parent of *i*; otherwise, make *i* the parent of *j*.
- An intermixed sequence of union and find operations is performed; a *union* operation combines two sets into one, a *find* operation identifies a set that contains a certain element.
- The *best time complexity is O(n + u log u + f)* where u = the # of union operations done and f = the # of find operation done.
- Using a tree (not binary tree) to represent a set, the time complexity becomes almost O(n + f) assuming at least n/2 union operations are performed.
- The *find(i)* operation is to identify the set that contains element i.
- The requirement is that *find(i)* and *find(j)* return the same value iff element i and j are in the same set

A call to *find(9)* would return the value stored in the root = 4 because this is considered one set



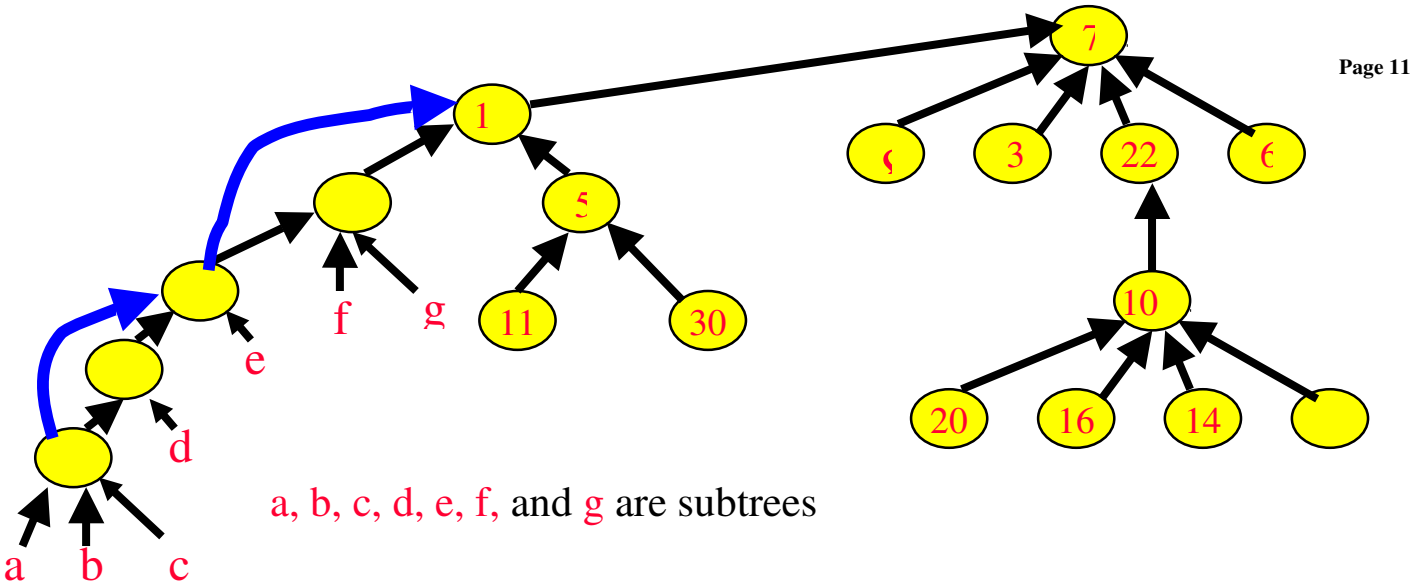
- **Strategy for** *find(i)* **:**
  - Start at the node that represents *i* and climb to the root, return the element in the root. **To climb the tree each node must have a pointer to its parent node**.

- **Possible node structure**:
  - Use nodes that have two fields: element and parent
  - Use an array table[i] such that table[i] is a pointer to the node whose element is i.
  - To do a *find(i)* operation start at the node given by table[i] and follow parent fields until a node whose parent field is null.
  - Return the element stored in this root node.



- **A smart union strategy**:
  - **Use the height rule**: **make the tree with the smaller height the sub-tree of the other tree**. Break ties arbitrarily. ***Don't use the height rule when using a path compaction strategy*** because compaction changes the height and considerable effort must be taken to calculate the new tree height
  - **Use the weight rule**: make the tree with a fewer number of elements the sub-tree of the other tree. Break ties arbitrarily.
  - **To implement the height and weight strategies**:
    - The root of each tree must record its height or the number of elements in the tree.
    - If using the **height rule** when a union is performed the height increases ***only*** when two trees of equal height are united.
    - If the **weight rule** is used then the weight of the new tree is the sum of the weights of the trees that are united.
  - **The height of a tree**:
    - If we start with a single element tree and perform unions using either the height or the weight rule the height of the tree with **p** elements is at most the **floor** $(\log_2 p) + 1$

- **Path compaction**:
  - Make all the nodes (that are traversed with the call to find) point to the root node instead of it's parent node
- **Path splitting**:
  - Make all the nodes (that are traversed with the call to find) point to the it's grandparent node instead of its parent node
- **Path halving**:
  - Make the parent pointer *of every other node visited* during the traversal point to its grandparent (see diagram on next page)



Priority queues:

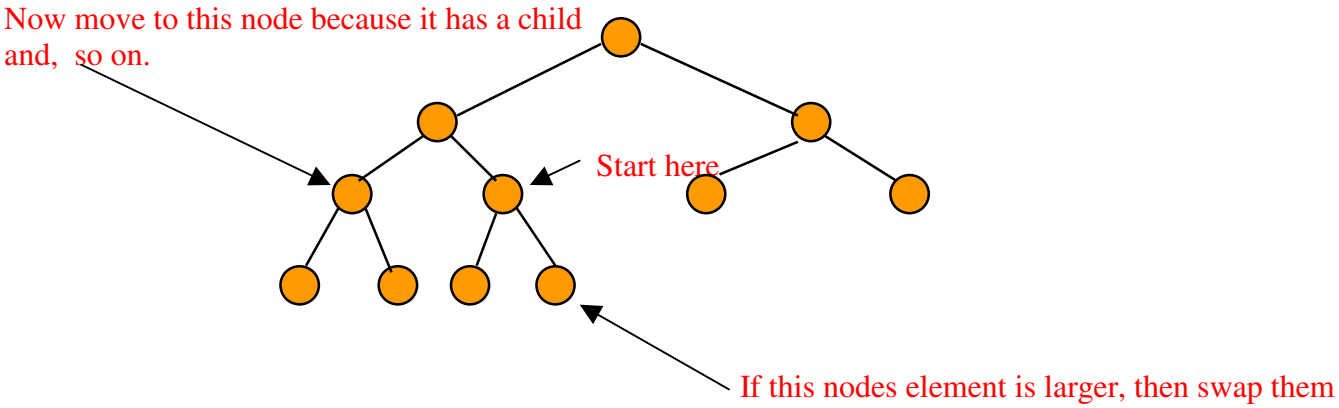
- 2 kinds: min priority and max priority
- 2 good implementations are heaps and leftist trees
- **Min priority queue**:
  - A collection of elements where each element has a priority or key
  - In the machine-scheduling example when the machine becomes available the job that takes the minimum amount of time to finish (this is its priority) is processed first.
  - Supports the following operations:
    - IsEmpty() O(1)
    - Size() O(1)
    - Add/put an element into the priority queue O(log n)
    - Get an element with min priority O(1)
    - Remove an element with min priority O(log n)
- **Max priority queue**: The simplest way to represent a max priority queue is as an unordered linear list
  - Is exactly the same as min priority but with max replacing min.
- **Applications of priority queues:**
  - **Sorting**: the complexity is O(n log n)
    - Use the element key as priority
    - Put elements to be sorted into priority queue
    - Extract elements in priority order
      - ☼ If a min priority queue is used the elements are extracted in ascending order of priority (or key)
      - ☼ If a max priority queue is used the elements are extracted in descending order of priority (or key)
  - Scheduling:

- LPT (longest processing time): each job is scheduled on a machine in which it finishes earliest
- LPT does not guarantee minimum finish time schedule

Min and Max trees and the heap:

- Each node has a value
- **For a Min tree**: No descendent has a smaller value than its root (the root is the min value)
- **For a Max tree**: No descendent has a larger value than its root (root is the max value)
- Both are used with **the Heap:**
  - The heap is a complete binary tree

  - The height of an **n** node heap is $\log_2 (n+1)$
  - A heap can be efficiently represented as an array – *see page 10 of these notes*
  - When we remove the max element in a max tree the root is removed, we then must reposition the other elements to maintain the max tree
- **Initializing a max heap**: *a max heap is a max tree that is also a **complete** binary tree*
  - the input array = [-, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] the tree originally has a level order node structure with 1 at the root, 2 at the first left child, 3 at the first right child, etc...
  - Start at the **rightmost** array position **that has a child**. Index is n/2



  - Move to the next lower array position evaluating if its parent node is larger or smaller, if its smaller than swap them, total time complexity is O(n).
  - **When we insert a new element into the heap**, if its a max heap we use the formula theElement <= heap [currentNode/2] to ensure that we are not violating the max heap rules. If a violation does occur we enter the body of a while loop that moves the element at currentNode/2 down to currentNode and then advances currentNode up to its parent (currentNode/2)

Leftist trees:

- Is a *linked binary tree*
- Can do everything a max heap can do in the same asymptotic time
- Can meld two leftist tree priority queues in O(log n) time
- **Height biased leftist tree (HBLT):** a binary tree is a height biased leftist tree iff for every *internal* node x, s(leftChild(x)) >= s(rightChild(x))
- In a leftist tree the rightmost past is the shortest path to an external node and the length of this path is s(root) *[look at the next paragraph for the definition of s(x)]*
- The number of internal nodes is at least $\mathbf{2^{s(root)} - 1}$ because level 1 – s(root) has no external nodes
- The length of the rightmost path is O(log n), where n is the number of nodes in the leftist tree
- Max leftist trees are used as a max priority queue
- Min leftist trees are used as a min priority queue

Extended binary trees: a binary tree with external nodes added

- Start with any binary tree and add a leftist node whenever there is an empty sub-tree
- The result is an **extended binary tree**
- The number of external nodes is **n+1**
- **The function s(x):**
  - For any node **x** in an extended binary tree, let **s(x)** be the length of the shortest path from **x** to an external node in the sub-tree rooted at **x**
  - **If x is an external node** then s(**x**) = 0, otherwise s(**x**) = min{s.leftChild(**x**), s.rightChild(**x**)} +1 (we add the 1 because we want to account for the node that is calling the function)
  - S( ) values can be computed easily using a post-order traversal

Tournament trees:

- **Winner trees**: a complete binary tree with n nodes and n-1 external nodes
  - The external nodes represent tournament players
  - Each internal node represents a match played between its two children; the winner of the match is stored at the internal node
  - The root is the overall winner
  - These are good for sorting; place the elements to be sorted into a winner tree, repeatedly extracting the winner and replace by a large value
  - The total sort time is O(n log n)
  - To replay you replace the winner and repeat the process; the relay complexity is O(log n)
- **Loser trees:** are exactly like winner trees except the node stores the loser of the match rather than the winner, the winner is "bubbled" up to its root to play the next match.
- **Truck loading / bin packing**:
  - **n** packages are to be loaded in trucks or bins
  - Each package has a weight or size
  - Each truck or bin has a capacity of **c**
  - We want to use the minimum number of trucks or bins
  - **Bin packing heuristics**:
    - **First fit**:
      - Bins are arranged in left to right order
      - Items are packed one at a time in a given order
      - The current item is packed into the leftmost bin in which it fits
      - If there is no bin into which the current item fits then start a new bin
      - This strategy is often not the optimal choice
      - Use a max tournament tree in which the players are the n bins and the value of a player is the available capacity in the bin
    - **First fit decreasing**:
      - Items are sorted into *decreasing* order, then first fit is applied
    - **Best fit**:
      - Items are packed one at a time in given order
      - To determine the bin for an item first determine a set **S** of bins into which the item fits
      - If **S** is empty then start a new bin and put the item into it
      - Otherwise pack the item into a bin that is in the set **S** that has the least available space.
    - **Best fit decreasing**:
      - Items are sorted into decreasing order and then best fit is applied

First fit decreasing and best fit decreasing are more efficient, (11/9)(minimum bins) + 4,

than first fit and best fits (17/10)(minimum bins) + 2

Binary search trees:

- **A binary search tree** where each node has a (key, value) pair, and for every node x all keys in the left sub-tree are smaller than x and all keys in the right sub-tree are greater than x.
- **Dictionary operations**:
  - get(key)
  - put(key, value)
  - remove(key)

- **Additional operations**:
  - ascend( )
  - get(index) – indexed binary search tree
  - remove(index) – indexed binary search tree
    - **Three cases**:
      1. When the element is in a leaf: simple break the link to the leaf node
      2. When the element is in a degree 1 node: break the link to the node from its parent and link the parent to the grandchild node.
      3. When the element is in a degree 2 node: replace the node to be removed with the largest key in the left sub-tree (or the smallest key in the right sub-tree)

Complexity of additional operations: D is the number of buckets

Data Structure	ascend	get and remove
Hash Table	O(D + n log n)	O(D + n log n)
Indexed BST	O(n)	O(n)
Indexed Balanced BST	O(n)	O(log n)

Complexity of dictionary operations: n is the number of elements in the dictionary

Data Structure	Worst Case	Expected
Hash Table	O(n)	O(1)
Binary Search Tree	O(n)	O(log n)
Balanced Binary Search Tree	O(log n)	O(log n)

A binary search tree has the following properties:

- Every element has a key and no two elements have the same key, therefore they are all distinct
- The keys (if any) in the left sub-tree of the root are **smaller** than the key in the root
- The keys in the right sub-tree of the root are **larger** than the key in the root
- The left and right sub-trees of the root are also binary search trees

- **Indexed binary search tree:**
  - Each node has an additional field named **leftSize** = the number of nodes in its left sub-tree
  - The **rank** of an element is its position in in-order
    - *Example*: if we have [12,13,14,15,16,17,18,19]
      Then rank(12) = 0, rank(13) = 1, rank(17) = 5, rank(19) = 7

- **Balanced binary search trees**:
  - **Indexed AVL trees**:

- A binary search tree that for every node x it has a balance factor defined by:
  - Balance factor of x = height of left sub-tree minus height of right sub-tree
  - The balance factor of every node *should be* either –1, 0, or 1 or it is out of balance
- 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 $\log_2 (n+1)$
- RR imbalance occurs when the balance factor is not –1, 0, or 1 but something else
- RR rotation is when we must rotate the sub-tree to make its balance factor correct
- There are 3 other kinds of rotations: RL, LR, and LL

- **Indexed red-black trees**:
  - A binary search tree where each node is colored red or black
  - The root and all external nodes are black
  - **No** root-to-external-node-path has two or more consecutive red nodes (so at most only 1 red node)
  - **All** root-to-external-node-paths have the same number of black nodes
  - The child pointers are colored red or black
  - The pointer to an external node is black
  - No root-to-external-node-path has two consecutive red pointers
  - Every root-to-external-node-path has the same number of black pointers
- Indexed operations also take O(log n) time

Things to study further:

- Take a look at the solution for exercise 45 in chapter 12 (pg 478) to understand the implementation of the compare(x), clone(x), and swapSubtees() methods for ADT BinaryTree
- Thoroughly study and know the classes BinaryTreeNode and LinkedBinaryTree. Problems involving these are sure to be on the exam.
- Take a good look at the meld( ) method for trees

```
/** the class linked binary tree */

package dataStructures;

import java.lang.reflect.*;

public class LinkedBinaryTree implements BinaryTree
{
   // instance data member
   BinaryTreeNode root;  // root node

   // class data members
   static Method visit;      // visit method to use during a traversal
   static Object [] visitArgs = new Object [1];
                             // parameters of visit method
   static int count;         // counter
   static Class [] paramType = {BinaryTreeNode.class};
                             // type of parameter for visit
   static Method theAdd1;    // method to increment count by 1
   static Method theOutput;  // method to output node element

   // method to initialize class data members
   static
   {
      try
      {
         Class lbt = LinkedBinaryTree.class;
         theAdd1 = lbt.getMethod("add1", paramType);
         theOutput = lbt.getMethod("output", paramType);
```

```
      }
      catch (Exception e) {}
         // exception not possible
   }

   // only default constructor available

   // class methods
   /** visit method that outputs element */
   public static void output(BinaryTreeNode t)
      {System.out.print(t.element + " ");}



   /** visit method to count nodes */
```



```
public static void add1(BinaryTreeNode t)
      {count++;}

   // instance methods
   /** @return true iff tree is empty */
   public boolean isEmpty()
      {return root == null;}


   /** @return root element if tree is not empty
     * @return null if tree is empty */
   public Object root()
      {return (root == null) ? null : root.element;}

   /** set this to the tree with the given root and subtrees
     * CAUTION: does not clone left and right */
   public void makeTree(Object root, Object left, Object right)
   {
      this.root = new BinaryTreeNode(root,
                     ((LinkedBinaryTree) left).root,
                     ((LinkedBinaryTree) right).root);
   }

   /** remove the left subtree
     * @throws IllegalArgumentException when tree is empty
     * @return removed subtree */
   public BinaryTree removeLeftSubtree()
   {
      if (root == null)
         throw new IllegalArgumentException("tree is empty");

      // detach left subtree and save in leftSubtree
      LinkedBinaryTree leftSubtree = new LinkedBinaryTree();
      leftSubtree.root = root.leftChild;
      root.leftChild = null;

      return (BinaryTree) leftSubtree;
   }

   /** remove the right subtree
     * @throws IllegalArgumentException when tree is empty
     * @return removed subtree */
   public BinaryTree removeRightSubtree()
   {
      if (root == null)
         throw new IllegalArgumentException("tree is empty");

      // detach right subtree and save in rightSubtree
      LinkedBinaryTree rightSubtree = new LinkedBinaryTree();
      rightSubtree.root = root.rightChild;
      root.rightChild = null;
      return (BinaryTree) rightSubtree;
   }

   /** preorder traversal */
   public void preOrder(Method visit)
   {
      this.visit = visit;
      thePreOrder(root);
   }
```

```
/** actual preorder traversal method */
static void thePreOrder(BinaryTreeNode t)
{
   if (t != null)
   {
      visitArgs[0] = t;
      try {visit.invoke(null, visitArgs);}  // visit tree root
      catch (Exception e)
         {System.out.println(e);}
      thePreOrder(t.leftChild);             // do left subtree
      thePreOrder(t.rightChild);            // do right subtree
   }
}
```



```
/** inorder traversal */
public void inOrder(Method visit)
{
   this.visit = visit;
   theInOrder(root);
}

/** actual inorder traversal method */
static void theInOrder(BinaryTreeNode t)
{
   if (t != null)
   {
      theInOrder(t.leftChild);              // do left subtree
      visitArgs[0] = t;
      try {visit.invoke(null, visitArgs);}  // visit tree root
      catch (Exception e)
         {System.out.println(e);}
      theInOrder(t.rightChild);             // do right subtree
   }
}

/** postorder traversal */
public void postOrder(Method visit)
{
   this.visit = visit;
   thePostOrder(root);
}

/** actual postorder traversal method */
static void thePostOrder(BinaryTreeNode t)
{
   if (t != null)
   {
      thePostOrder(t.leftChild);            // do left subtree
      thePostOrder(t.rightChild);           // do right subtree
      visitArgs[0] = t;
      try {visit.invoke(null, visitArgs);}  // visit tree root
      catch (Exception e)
         {System.out.println(e);}
   }
}

/** level order traversal */
public void levelOrder(Method visit)
{
   ArrayQueue q = new ArrayQueue();
   BinaryTreeNode t = root;
   while (t != null)
   {
      visitArgs[0] = t;
      try {visit.invoke(null, visitArgs);}  // visit tree root
      catch (Exception e)
         {System.out.println(e);}

      // put t's children on queue
      if (t.leftChild != null)
         q.put(t.leftChild);
      if (t.rightChild != null)
         q.put(t.rightChild);

      // get next node to visit
      t = (BinaryTreeNode) q.remove();
```

```
   }
}

/** output elements in preorder */
public void preOrderOutput()
   {preOrder(theOutput);}

/** output elements in inorder */
public void inOrderOutput()
   {inOrder(theOutput);}

/** output elements in postorder */
public void postOrderOutput()
   {postOrder(theOutput);}

/** output elements in level order */
public void levelOrderOutput()
   {levelOrder(theOutput);}

/** count number of nodes in tree */
public int size()
{
   count = 0;
   preOrder(theAdd1);
   return count;
}

/** @return tree height */
public int height()
   {return theHeight(root);}

/** @return height of subtree rooted at t */
static int theHeight(BinaryTreeNode t)
{
   if (t == null) return 0;
   int hl = theHeight(t.leftChild);  // height of left subtree
   int hr = theHeight(t.rightChild); // height of right subtree
   if (hl > hr) return ++hl;
   else return ++hr;
}
```