CSC 301: Data Structures and Algorithms

Recursion and Trees

Denis L. Nkweteyim

Recursion

- Fundamental to computer science
- Recursive program/function
  - Program/function that calls itself
- So that a recursive function (or program) does not call itself for eternity
  - There must be a terminating (base) condition when the function ceases to call itself
- Trees
  - A category of recursively defined data structures
  - We study trees not only as data structures, but also as tools to help us understand and analyze recursion

Recursion

- Recursive algorithm
  - Solves a problem by solving one or more smaller instances of the same problem
  - Implemented in C using recursive functions
- Two key requirements for recursion to be successful
  - Every recursive call must simplify the computation in some way
  - There must be one or more special cases (base cases) to handle the simplest computations
- Note
  - Functions that require repetition can be coded both iteratively and recursively

Example: Factorial function

- Defined by the recurrence relation:

$$N! = N \cdot N-1! \text{ for } N>0$$
$$= 1 \text{ for } N=0$$

- Code should be easy to follow
- Notes on recursive version
  - Base case: factorial(0) = 1
  - Recursive case: (factorial(n) = n * factorial(n – 1))
    - converges to the base case

```
Recursive version – fact_r.c
int fact_r(int n){
  if (n == 0)
    return 1;
  return (n * fact_r(n - 1));
}
```

```
Iterative version – fact_i.c
int fact(int n){
  int i, t;
  for (t = 1, i = 1; i <= n; i++)
    t *= i;
  return t;
}
```

Example: Factorial function

- Recursive calls made (for n=5)

```
fact(5)
5 * fact(4)
5 * 4 * fact(3)
5 * 4 * 3 * fact(2)
5 * 4 * 3 * 2 * fact(1)
5 * 4 * 3 * 2 * 1 * fact(0)
5 * 4 * 3 * 2 * 1 * 1
```

Some notes on the recursive version

- At the time the recursive call is made
  - There are no unfinished statements
    - All that is required is for the function to end
  - This kind of recursion is called end or tail recursion
  - A function that uses end recursion is usually easy to write iteratively, as we saw with the iterative version of the factorial program

Greatest Common Divisor (GCD)

- GCD of two or more integers, when at least one of them is not zero
  - The largest positive integer that divides the numbers without a remainder
    - gcd(8, 12) = 4
  - Also
    - gcd(a,0) = a

GCD

```
int gcd_i(int a, int b) {
  int c;
  while (a != 0) {
     c = a; a = b%a;  b = c;
  }
  return b;
}
```

```
int gcd_r(int a,int b) {
  int r;
  if ((r = a%b) == 0)
     return b;
  else
     return gcd_r(b,r);
}
```

Evaluating prefix expressions

- An infix expression can be converted to a prefix expression
- Example
  - Infix: `5*(((9+8)*(4*6))+7)`
  - Prefix equivalent: `* + 7 * * 4 6 + 8 9 5`
  - Program can be written to easily evaluate prefix expressions

Evaluating prefix expressions - example

```c
#include <stdio.h>
int eval();
int i=0;char a[30];
int main(void){
   int e1,e2;
   a[0]='*';a[1]=' ';a[2]='+';a[3]=' ';a[4]='7';a[5]=' ';a[6]='*';
   a[7]=' ';a[8]='*';a[9]=' ';a[10]='4';a[11]=' ';a[12]='6';a[13]=' ';
   a[14]='+';a[15]=' ';a[16]='8';a[17]=' ';a[18]='9';a[19]=' ';
   a[20]='5';a[21]=' ';a[22]=0;
   printf("Expression evaluates to %d\n",eval());
}
int eval() {
   int x = 0;
   while (a[i] == ' ')  i++;
   if (a[i] == '+') {   i++;   return eval() + eval();  }
   if (a[i] == '*') {   i++;   return eval() * eval();  }
   while ((a[i] >= '0') && (a[i] <= '9')){
     x = 10*x + (a[i++] - '0');
   }
   return x;
}
```

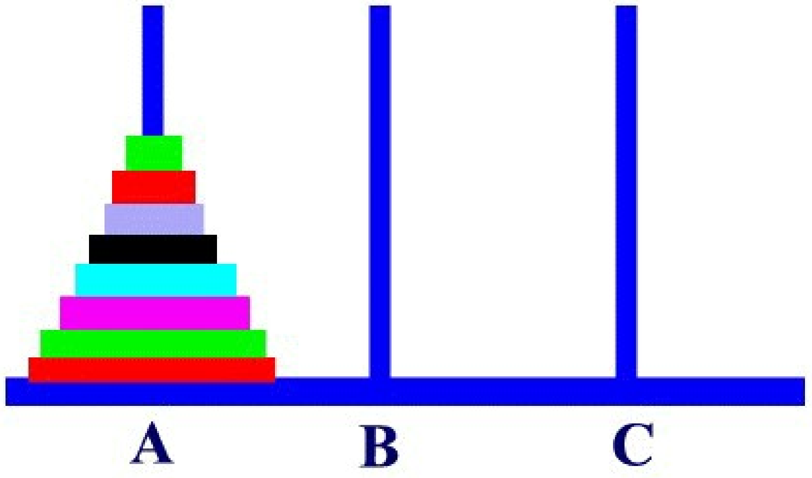
```
Expression evaluates to 2075
```

Evaluating prefix expressions – recursive calls

```
eval() * + 7 * * 4 6 + 8 9 5
  eval() + 7 * * 4 6 + 8 9 5
    eval() 7 * * 4 6 + 8 9 5
    return 7
    eval() * * 4 6 + 8 9 5
      eval()* 4 6 + 8 9 5
        eval()4
        eval()6
        return 24 = 4 * 6
      eval() + 8 9 5
        eval() 8
        eval() 9
        return 17 = 8 + 9
      return 408 = 24 *17
    return 415 = 7 + 408
  eval() 5
return 2075 = 415*5
```

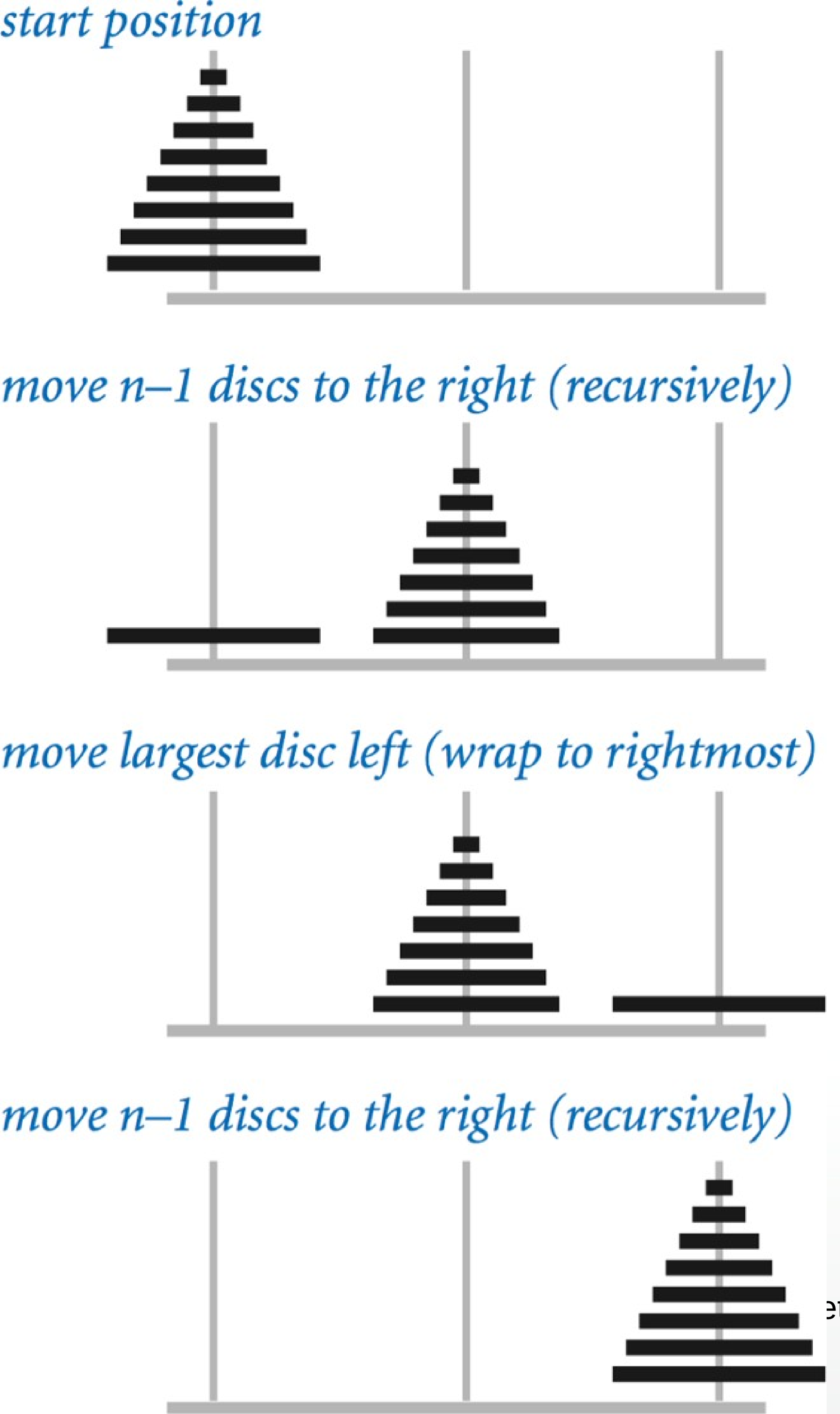
Tower of Hanoi

- Trival Case
  - Moving One Disk: Move the disk directly from source to destination
- Moving Two Disks
  1. Place the top (i.e., smaller) disk out of the way to intermediate tower
  2. Move the larger, bottom disk from source to destination
  3. Move the smaller disk from intermediate to destination tower
- Moving Three Disks from A to C, say
  1. Move 2 disks from A to B
  2. Move third disk from A to C
  3. Move 2 disks from B to C
  - We already know how to move 2 disks from A to C
    - By using the Tower B to temporarily hold the top disk
    - So, to move 2 disks from A to B, Tower C will act as the temporary Tower; and Tower A will be the temporary Tower as we move disks between B and C



- General Rule for Moving n Disks
  1. Move n-1 disks from A to B
  2. Move nth disk from A to C
  3. Move n-1 disks from B to C

Moving n Disks


start position
move n–1 discs to the right (recursively)
move largest disc left (wrap to rightmost)
move n–1 discs to the right (recursively)

Tower of Hanoi

```c
#include <stdio.h>
#include <stdlib.h>
int cnt=0; /*keep track of number of moves*/
/*a tower comprises ints representing the tower tag and the disks*/
typedef int * tower;
int get_n(void); void move(int n,tower a,tower b,tower c, int sz);
int top_index(tower a, int sz);
void write_moves(tower a, tower b, tower c, int d, int sz);
int main(void) {
  int i;
  int n; /*number of disks*/
  int sz; /*size of tower arrays*/
  tower a, b, c; /*the 3 towers*/
  n = get_n(); /*get number of disks to use*/
  sz = n + 1;
  /*each tower has space for a tag and n disks*/
  a=calloc(sz,sizeof(int));b=calloc(sz,sizeof(int));
  c=calloc(sz,sizeof(int));
  /*store the tag of each tower*/
  a[0] = 'A'; b[0] = 'B'; c[0] = 'C';
  /*start with all n disks on tower a*/
  for (i = 1; i < sz; i++)  a[i] = i; //disk 1 is smallest
  write_moves(a, b, c, 0, sz); /*display initial info*/
  move(n, a, b, c, sz); /*move n disks from A to C*/
}
```

Tower of Hanoi

```
void move(int n, tower a, tower b, tower c, int sz) {
  int i, j; int d; /*disk*/
  if (n == 1) { cnt++; i = top_index(a, sz); j = top_index(c, sz);
                d=a[i]; a[i]=0; c[j-1]=d; write_moves(a,b,c,d,sz); }
  else { move(n-1,a,c,b,sz);move(1,a,b,c,sz);move(n-1,b,a,c,sz);  }
}
int get_n(void) {
  int n;
  printf("How many disks are there? ");
  while (1)
    if (scanf("%d",&n) != 1 || n<1)
      printf("\nError: +ve int expected\n");
    else  break;
  return n;
}
int top_index(tower a, int sz) { //index of disk to move
  int i;
  for (i = 1; i < sz && a[i] == 0; i++)    ; //do nothing
  return i;
}
void write_moves(tower a, tower b, tower c, int d, int sz){
  if (cnt == 0)  printf("%5d%-26s\n", cnt, ": Start:");
  else   printf("%5d%s%d%s%c%s%c%s\n",cnt,": Move disk ",d,
                " from ",a[0]," to ",c[0],":");  }
```

Sample output

```
Sample Output
How many disks are there? 4
0: Start:
1: Move disk 1 from A to B:
2: Move disk 2 from A to C:
3: Move disk 1 from B to C:
4: Move disk 3 from A to B:
5: Move disk 1 from C to A:
6: Move disk 2 from C to B:
7: Move disk 1 from A to B:
8: Move disk 4 from A to C:
9: Move disk 1 from B to C:
10: Move disk 2 from B to A:
11: Move disk 1 from C to A:
12: Move disk 3 from B to C:
13: Move disk 1 from A to B:
14: Move disk 2 from A to C:
15: Move disk 1 from B to C:
```

Tower of Hanoi

- Arrays a, b, and c
  - Size n + 1 to hold a maximum of n disks and the tower tag (A, B, or C)
  - Initialized to zero values, followed by the tag labels at index 0
    - a: 'A' 0 0 0 0
    - b: 'B' 0 0 0 0
    - c: 'C' 0 0 0 0
  - Array a then set to
    - a: 'A' 1 2 3 4
- As the disks are moved, the program ensures that the highest index position of any array always holds the highest disk number
  - e.g., if array a holds disks 1 and 2, the contents of that array will be as follows:
    - a: 'A' 0 0 1 2
  - Similarly, if array b holds disks 4 and 2, the contents of array b will be:
    - b: 'B' 0 0 2 4

Tower of Hanoi

- Function move(no of disks, from_tower, intermediate_tower, to_tower, array_size)
  - Does most of the work in the program
    - If trivial case (no of disks to move is 1)
      - increment counter for number of moves
      - Move the first disk with non-zero value
        - Function top_index determines index value of disk to move
        - Function skips all the zero values until it meets the first non-zero value.
        - Corresponding disk a[i] deleted/set to 0
        - Determine top index (j) of destination tower, set c[j-1] to id of moved disk
      - Function write_moves() informs user of the move just made

Tower of Hanoi

- Function move(no of disks, from_tower, intermediate_tower, to_tower, array_size)
  - Non-trivial case (number of disks to move is not 1)
    - Call function recursively 3 times to
      - Move n – 1 disks to the intermediate tower, with destination as intermediate
      - Move the 1 disk that was not moved in the first call to the proper destination tower (trivial case)
      - Moves the n -1 disks that were moved to the intermediate tower to the destination, with source tower as intermediate

Divide-and-conquer

- Many recursive programs use two recursive calls, each operating on about one half of the input
  - Such programs are an important example of the divide-and-conquer paradigm
- We use the example of finding the maximum of n numbers to illustrate the approach
- Note
  - This particular problem is more easily solved using iteration as illustrated below
  - But we use recursion to illustrate the divide-and-conquer approach

```
for (t = a[0], i = 1; i < n; i++)
  if (a[i] > t)
    t = a[i];
```

Divide-and-conquer – recursive

```
/*l and r are left and right index values of the array*/
int max(int a[], int l, int r) {
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l]];
  u = max(a, l, m);
  v = max(a, m + 1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

Dynamic Programming

- Essential characteristic of the divide-and-conquer algorithm we saw
  - The problem is partitioned into independent sub-problems
- When the sub-problems are not independent, situation can be a lot more complicated
  - Recursive calls can require huge amounts of time
- Next we consider an example of a hugely inefficient implementation of a divide-and-conquer problem
  - Then consider how dynamic programming can be used to make the computations a lot more efficient

Fibonacci numbers

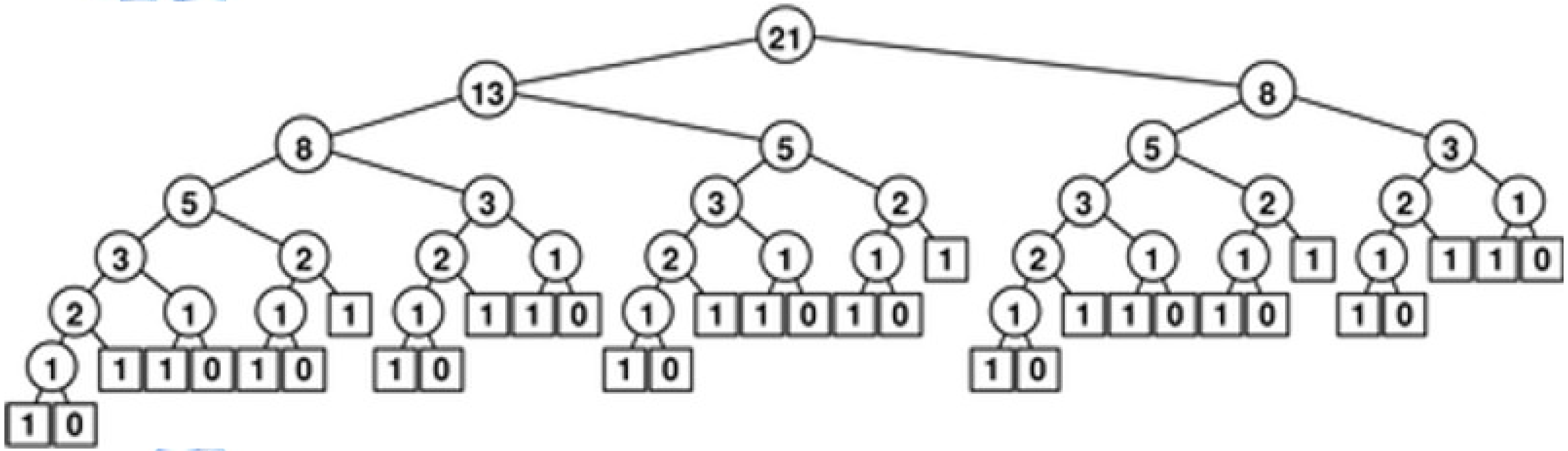
- Fibonacci numbers
  - 0 1 1 2 3 5 8 13 …
  - Each Fibonacci number is obtained from the formula:
    - $F_N = F_{N-1} + F_{N-2}$ for $N \geq 2$ with $F_0 = 0$ and $F_1 = 1$

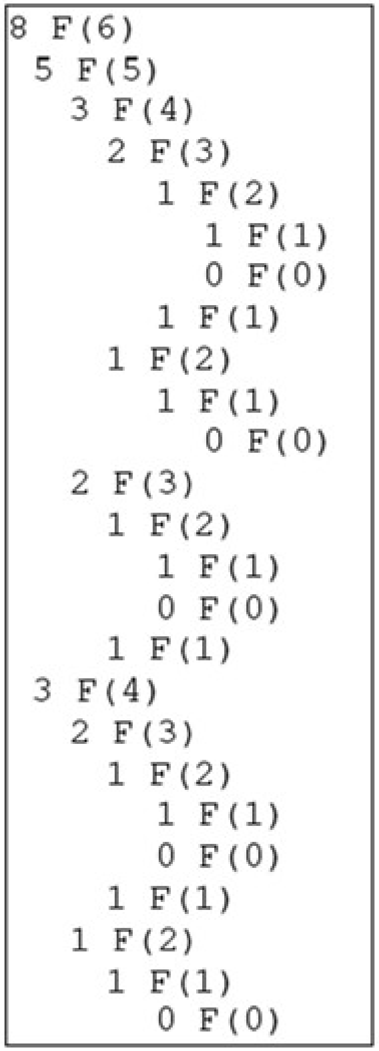
```
int F(int i) {
  if (i < 1) return 0;
  if (i == 1) return 1;
  return F(i - 1) + F(i - 2);
}
```

Fibonacci numbers

- The given recursive function to compute the ith Fibonacci number is hugely inefficient
  - Requires exponential time
- Reason
  - The recursion involves overlapping subproblems
    - Second recursive call F(i - 2) ignores the computations done during the first F(i – 1)
    - Results in massive re-computation because the effect multiplies recursively
    - (see example for computation of F_6)

Fibonacci numbers





```
8 F(6)
 5 F(5)
   3 F(4)
     2 F(3)
        1 F(2)
           1 F(1)
           0 F(0)
        1 F(1)
     1 F(2)
        1 F(1)
           0 F(0)
   2 F(3)
     1 F(2)
        1 F(1)
        0 F(0)
     1 F(1)
 3 F(4)
   2 F(3)
     1 F(2)
        1 F(1)
        0 F(0)
     1 F(1)
   1 F(2)
     1 F(1)
        0 F(0)
```

Dynamic Programming

```
F[0] = 0; F[1] = 1;
for (i = 2; i <= N; i++)
F[i] = F[i-1] + F[i-2];
```

- Code above
  - More efficient (linear time) iterative algorithm for computing Fibonacci numbers
  - Computes first N Fibonacci numbers, and stores them in an array
  - The program exploits the fact that we can afford to save all previously computed values, which can then be used for the current computation. This is known as dynamic programming.
  - In this example, we may even dispense of the array and keep track of only the last two numbers

Bottom-up Dynamic Programming

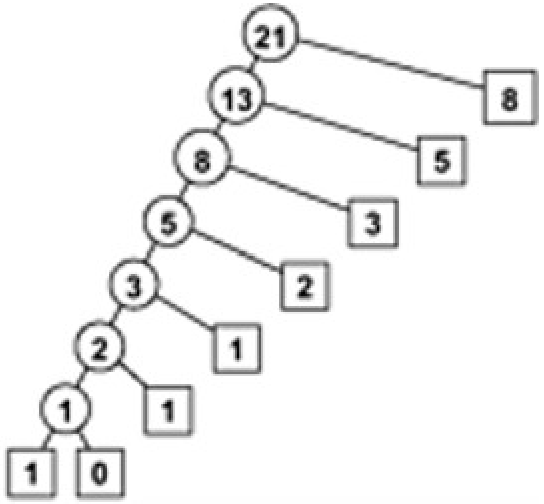
- Previous slide
  - Evaluate a function by computing all the function values in order
    - Starting at the smallest, and
    - Using previously computed values at each step to compute the current value
  - This is bottom-up dynamic programming
    - Applies to all recursive computation, provided we can afford to save all previously computed values
    - Technique has been used successfully for a large range of problems

Top-down Dynamic Programming

- Enables us to execute recursive functions at the same (or lower) cost than bottom-up dynamic programming
  - We save each value that the function computes (as its final action), and
  - Check the saved values to avoid any recomputation (as its first action)
  - Program on next slide transforms the expensive, recursive program we saw earlier, to one with linear running time

Top-down Dynamic Programming

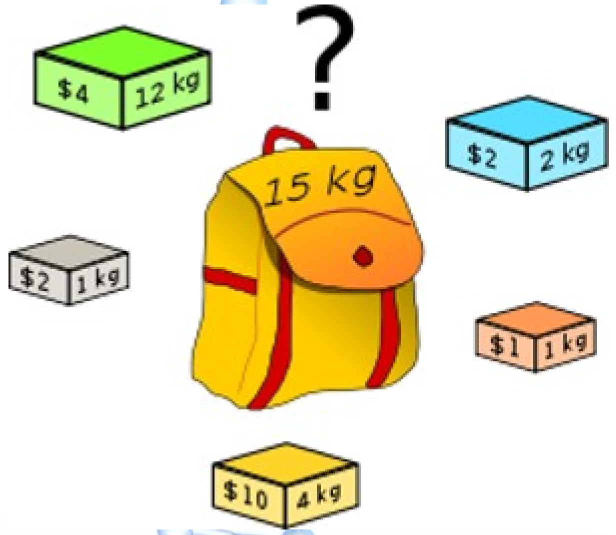
```
int F(int i) {
  int t;
  if (knownF[i] != unknown) return knownF[i];
  if (i == 0) t = 0;
  if (i == 1) t = 1;
  if (i > 1) t = F(i - 1) + F(i - 2);
  return knownF[i] = t;
}
```



- Makes use of an external array `knownF` initialized to some value called unknown
- Array knownF keeps track of the known Fibonacci numbers
  - Hence, no need for re-computation
- The diagram on the right shows how saving computed values cuts costs from exponential to linear

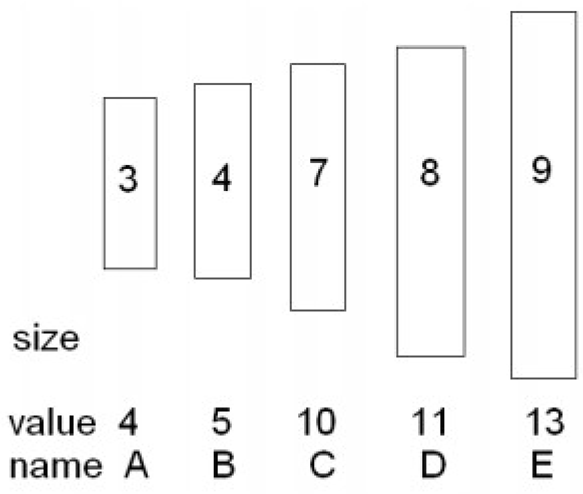
Knapsack Problem

- A problem in combinatorial optimization
- Several ways to describe the problem
  - One-dimensional (constraint) knapsack problem
    - Given a set of boxes, each with a weight and a value, determine the number of each box to include in a collection so that the total weight is less than or equal to a given limit (of the knapsack, i.e. 15 kg) and the total value is as large as possible?
      - If any number of each box is available, then three yellow boxes and three grey boxes
      - If only the shown boxes are available, then all but the green box
  - A multiple constrained problem could consider both the weight and volume of the boxes



Knapsack Problem – Another Illustration

- A thief robbing a safe
  - Safe is filled with N types of items of varying size and value
  - Thief only has a small knapsack of capacity M to use to carry the goods
- Knapsack problem is to find the combination of items which the thief should choose for his knapsack in order to maximize the total value of all the items he takes
- Example
  - Knapsack with capacity 17
  - Safe contains many items of the sizes and values shown on the left
  - Possible solutions
    - Five A's (but not six) – size 15, value 20
    - One D and one E – size 17, value 24
    - Many other combinations, but which will maximize his total take?



Example Application of the Knapsack Problem

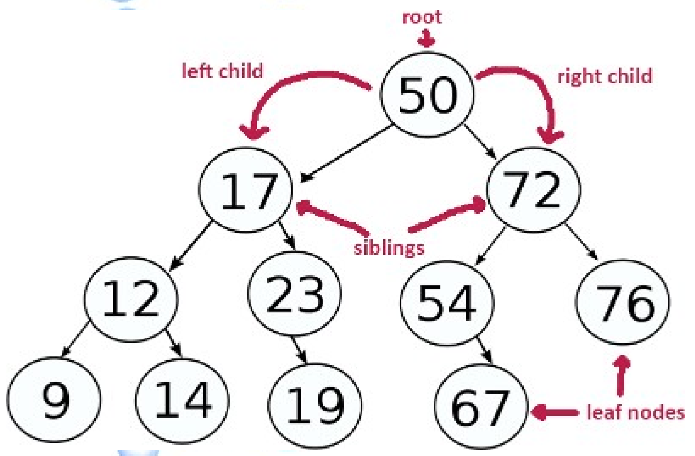
- A shipping company looking for the best way to load a truck or cargo plane with items for shipment
- Other variants of the problem possible
  - There might be a limited number of each kind of item available
- In a dynamic programming solution to the knapsack problem, we calculate the best combination for all knapsack size up to M
  - This can be done by doing things in an appropriate order

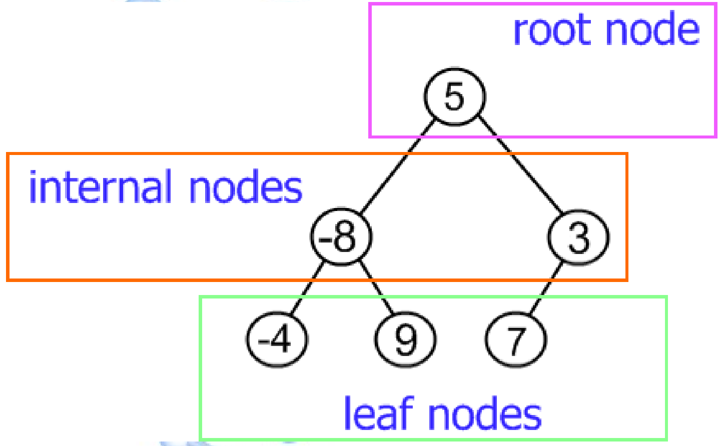
Trees

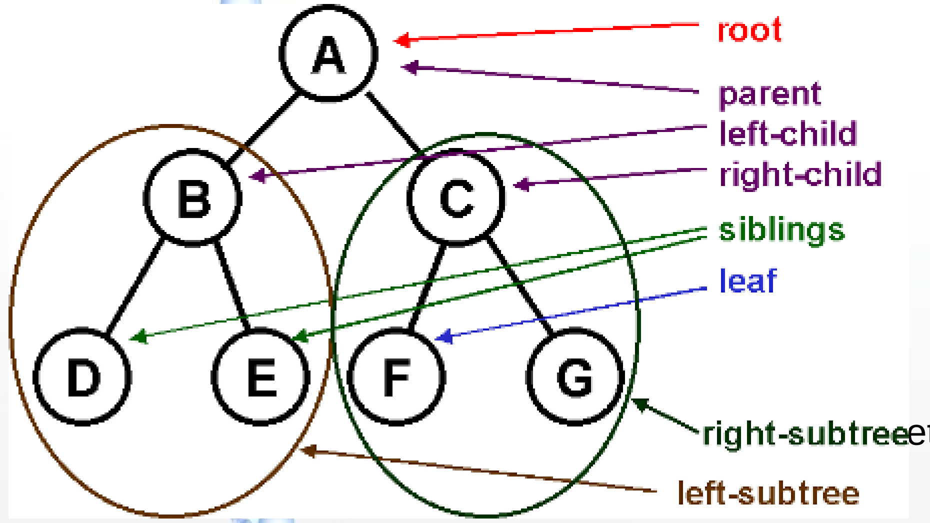
- Trees are frequently encountered
  - Family tree to keep track of ancestors and descendants
  - Organization of sports tournaments
  - Organizational charts
  - Organization of computer file systems
  - etc.

Trees terminology

- Tree
  - A non-empty collection of vertices (nodes) and edges that satisfies certain requirements
    - A vertex can have a name, as well as carry other information
    - An edge connects two vertices.
  - Path
    - A list of distinct vertices with successive vertices linked by edges
    - In a tree, there is exactly one path between any pair of nodes
      - If there is more than one path between any two nodes, or no path between some pair of nodes, then we have a graph (not a tree)





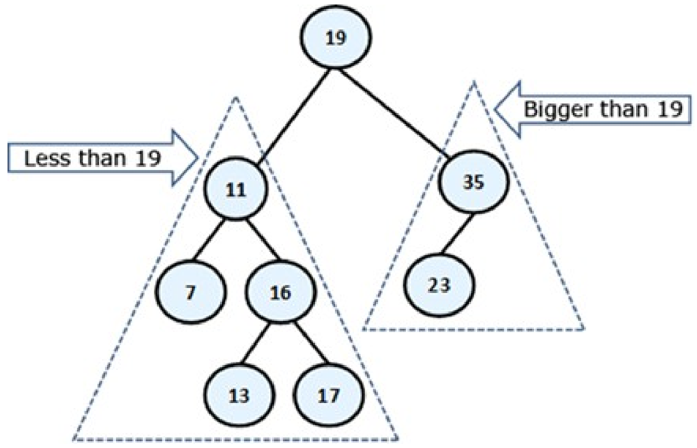


Trees terminology

- Rooted tree (also called unordered tree)
  - One where one node is designated as the root of the tree
    - If no such node is designated, then the tree is a free tree
  - By definition
    - A rooted tree is a node (called the root) connected to a multiset of rooted trees. (Such a multiset is called an unordered forest.)
      - Ordering in the trees (top left) is a coincidence, not required for rooted tree
  - Every node in a rooted tree is the root of a subtree consisting of the node and the nodes below it
    - There is exactly one path between the root and each of the other nodes in a tree.
  - A node y is said to be below a node x, if x is on the path from y to the root.
  - Each node (except the root node) has exactly one node above it, called its parent.
  - The nodes directly below a node are called its children
  - Nodes with no children are called leaf (or terminal) nodes
  - Nodes with one or more child(ren) are called internal (or non-terminal) nodes.

Trees terminology

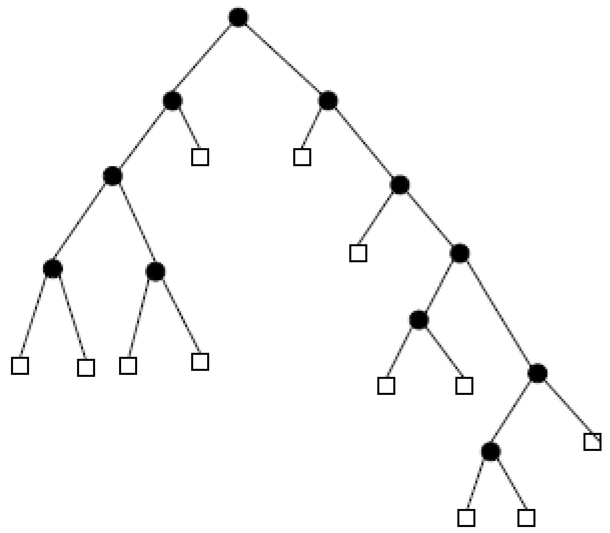
- Ordered tree
  - A node (called the root) connected to a sequence of disjoint trees (such a sequence is called a forest)
  - Notice that ordering of nodes is important here

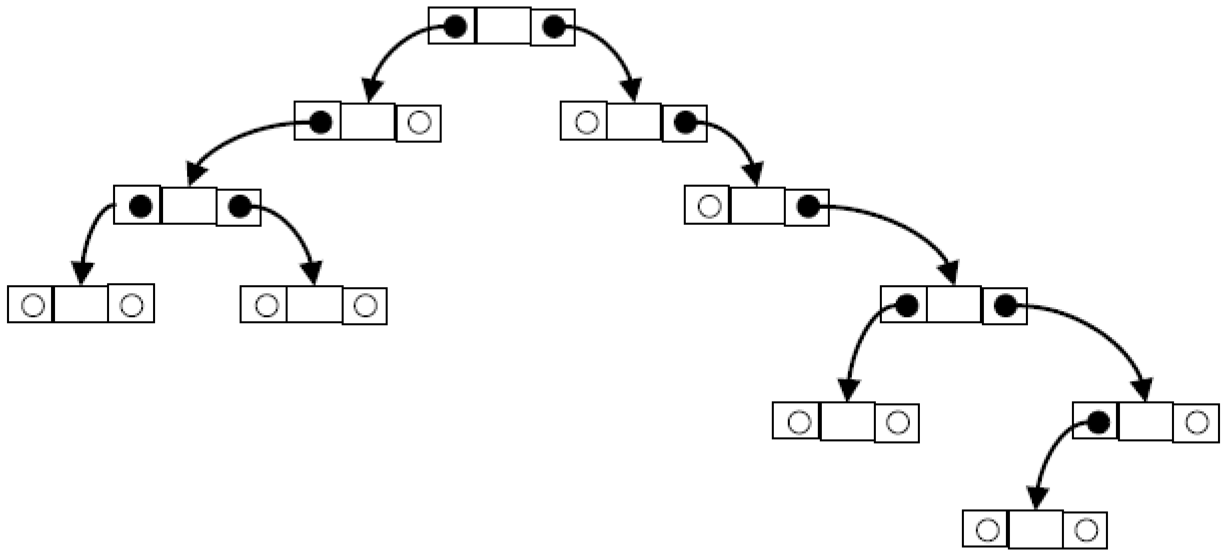


Trees terminology

- M-*ary* tree
  - If each node must have a specific number of children (M) appearing in a specific order, then we have an M-ary tree
  - i.e., either an external node or an internal node connected to an ordered sequence of M trees that are also M-ary trees
- Binary tree
  - The simplest M-ary tree
  - An ordered tree consisting of two types of nodes
    - External nodes with no children
    - Internal nodes with exactly two children called the left child and right child respectively
  - In other words, a binary tree is
    - Either an external node or an internal node connected to a pair of binary trees, called the left and right subtrees of that node

Data Structure to Represent a binary tree

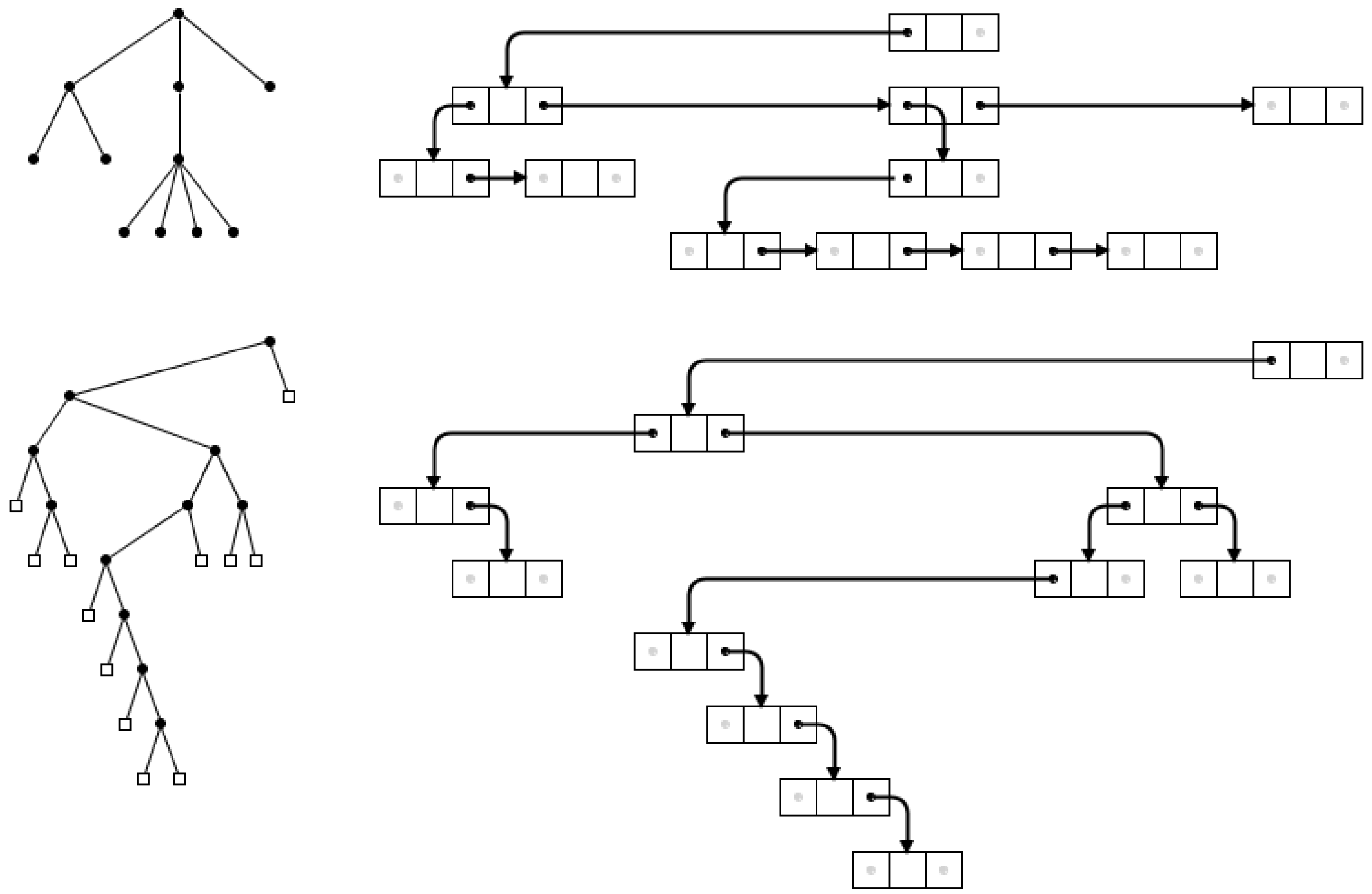




- Illustration suggests a linked list-type structure to represent a node

```
typedef struct node *Link;
struct node {
  Item data;
  Link left, right;
};
```

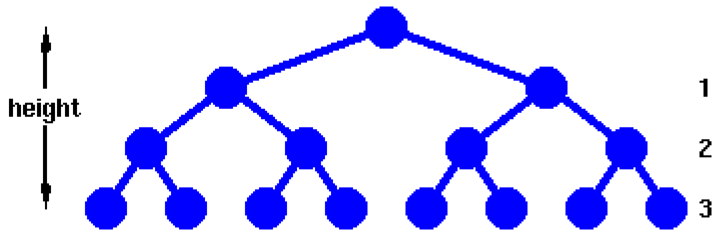
Data Structure to Represent an Mary tree



- Representing an ordered tree by keeping a linked list of the children of each node is equivalent to representing it as a binary tree
- Top right
  - Llinked-list-of-children representation of the tree on the left at the top
  - Right links of nodes point to siblings
  - Left link points to the first node in the linked list of its children
- Bottom right
  - Slightly rearranged version of the diagram above it and represents the binary tree at the bottom left

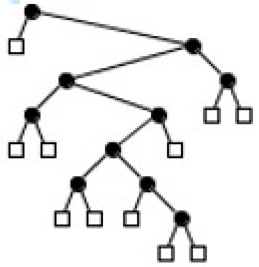
Some binary tree properties and definitions

- Level of a node
  - One higher than the level of its parent (with the root at level 0)
- Height of a tree
  - Maximum of the levels of the tree’s nodes

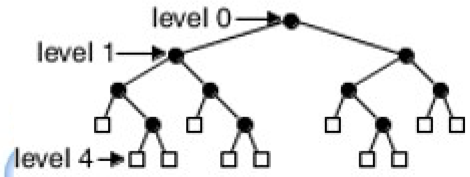


- Path length of a tree is the sum of the levels of all the tree’s nodes.
- Internal path length of a binary tree
  - The sum of the levels of all the tree’s internal nodes
- External path length of a binary tree
  - The sum of the levels of all the tree’s external nodes

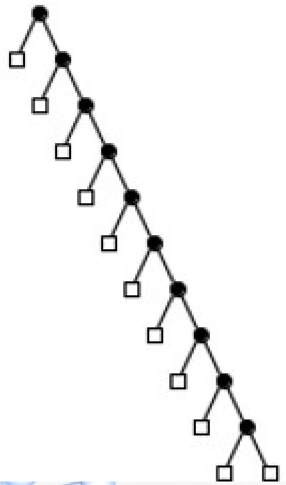
Examples



- Height 7
- Internal path length 31
- External path length 51



- Fully balanced binary tree with 10 internal nodes
  - Height 4
  - Internal path length 19
  - External path length 39
    - no binary tree with 10 nodes has smaller values for any of these quantities



- Degenerate binary tree with 10 internal nodes
  - Height 10
  - Internal path length 45
  - External path length 65
    - no binary tree with 10 nodes has larger values for any of these quantities

Tree traversal

- Given a pointer to a tree, we want to visit (and probably do some processing) on every node in the tree systematically
  - For linked lists, the process is straightforward
    - Simply follow the next link until the null pointer is encountered
    - For a tree however, decisions need to be made because there are several links one can follow
- Noting recursive definition of trees
  - A tree can be traversed most naturally recursively.
  - To understand how, it is instructive to see how a linked list can be traversed recursively

Recursive linked list traversal

- In-order traversal
  - Process a node as it is encountered until we reach the end of the list

```
void traverse(Link h) {
  if (h == NULL) return;
  visit(h); /*user function*/
  traverse(h->next);
}
```

- Reverse order traversal
  - Move to the end of the list before processing the nodes encountered

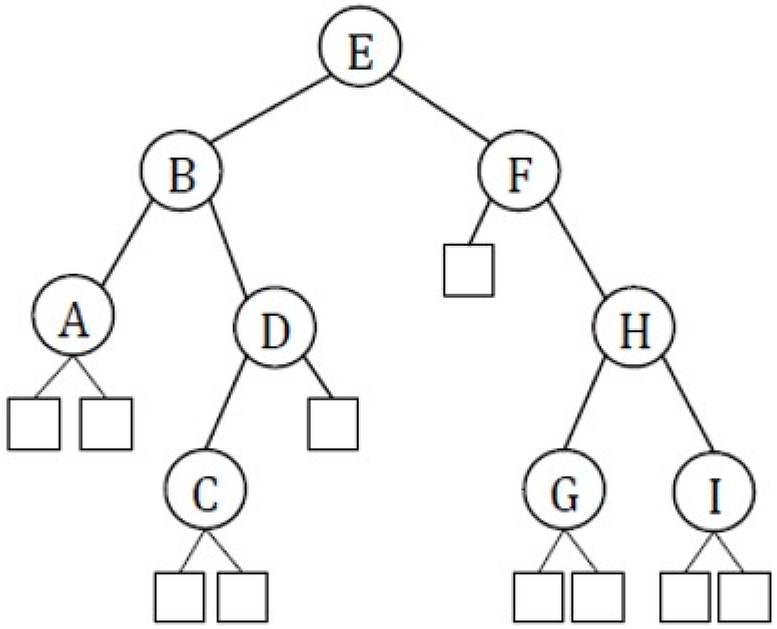
```
void traverseR(Link h) {
  if (h == NULL) return;
  traverseR(h->next);
  visit (h); /*user function*/
}
```

3 common ways to traverse binary tree

- Preorder
  - Visit the parent node, then the left subtree, and then the right subtree.
- Inorder
  - Visit the left subtree, then the parent node, and then the right subtree.
- Postorder
  - Visit the left subtree, then the right subtree, and then the parent node
- Level order traversal
  - Process nodes in the order in which they appear on the page: top-to-bottom, left-to-right

3 common ways to traverse binary tree

- Pre-order Transversal
  - EBADCFHGI
- In-order Transversal
  - ABCDEFGHI
- Post-order Transversal
  - ACDBGIHFE
- Level order traversal
  - EBFADHCGI



Pre-order binary tree traversal

```
void traverse(Link h) {
  if (h == null) return;
  visit(h);
  traverse(h -> left);
  traverse(h -> right);
}
```

- For in-order traversal
  - The only change needed would be to place the call to function visit between the two recursive calls
- For post-order traversal
  - Place the call to the function after the second recursive call

Sample recursive binary tree algorithms

```
int count(Link h) {
  if (h == NULL) return 0;
  return count(h -> left) + count(h -> right) + 1;
}

int height(Link h) {
  int u, v;
  if (h == NULL)
     return 0;
  u = height(h -> left);
  v = height(h -> right);
  if (u > v)
    return u + 1;
  else
    return v + 1;
}
```

Sample recursive binary tree algorithms

- Function count()
  - Counts the number of nodes in a binary tree
  - Recursively calls the left and right subtrees and then adds 1 to the node count, each time a node is encountered (i.e., h != NULL).

Sample recursive binary tree algorithms

- Function height()
  - Determines the height of a binary tree
  - Uses divide-and-conquer approach to split the tree into left and right subtree
    - Then recursively determines and compares the height of each subtree starting with the height set to 0 for each leaf node