UNIT I
PART II

The Stack Frame

- **Tree of Stack Frames**: Tracing out the entire computation of a recursive algorithm, one line of code at a time, can get incredibly complex.
- For this, the tree-of-stack-frames level of abstraction is best
- The key thing to understand is the difference between a particular routine and a particular execution of a routine on a particular input instance. A single routine can at one moment in time have many executions going on. Each such execution is referred to as a ***stack frame***

The Stack Frame

- If each routine makes a number of subroutine calls (recursive or not), then the stack frames that get executed form a tree

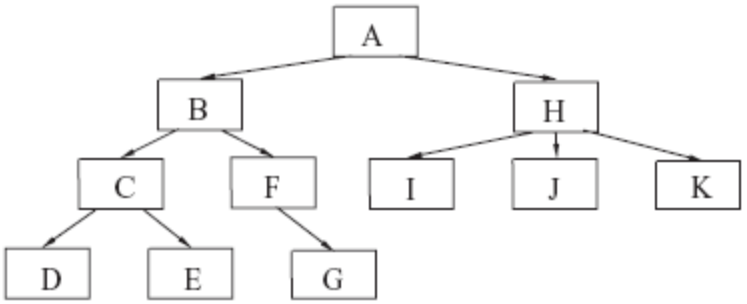


Figure 8.2: Tree of stack frames.

- In the example in Figure 8.2, instance *A* is called first
- It executes for a while and at some point recursively calls *B*. *When B* returns, *A then executes for a while longer before calling H. When H returns, A executes* for awhile before completing

The Stack Frame

- We have skipped over the details of the execution of *B. Let's go back to when instance A calls B. Then B calls C, which calls D. D completes;* then *C calls E. After E, C completes. Then B calls F, which calls G. Then G* completes, *F completes, B completes, and A goes on to call H. It does get complicated.*
- **Stack of Stack Frames:** The algorithm is actually implemented on a computer by a stack of stack frames. What is stored in the computer memory at any given point in time is only a single path down the tree. The tree represents what occurs throughout time

The Stack Frame

- In Figure 8.2,when instance *G is active, A, B, F, and G are in the stack. C, D, and E have been removed from memory as these have completed. H, I , J , and K have*
- not been started yet. Although we speak of many separate stack frames executing on the computer, the computer is not a parallel machine. Only the top stack frame *G is* actively being executed. The other instances are on hold, waiting for the return of a subroutine call that it made.

Proving Correctness with Strong Induction

- **Strong Induction:** Strong induction is similar to induction, except that instead of assuming only *S(n – 1) to prove S(n), you must assume all of S(0), S(1), S(2), . . . , S(n – 1).*
- **A Statement for Each *n*:** *For each value of n ≥ 0, let S(n) represent a Boolean* statement. For some values of *n this statement may be true, and for others it*
- may be false.
- **Goal: Our goal is to prove that it is true for every value of *n, namely that* ∀*n*** ≥ 0, *S(n).*

Proving Correctness with Strong Induction

- **Proof Outline: Proof by strong induction on *n*.**
- **Induction Hypothesis: For each $n \geq 0$, *let S(n) be the statement that* . . . . *(It* is important to state this clearly.)**
- **Base Case: Prove that the statement *S(0) is true*.**
- **Induction Step: For each $n \geq 0$, *prove S(0), S(1), S(2), . . . , $S(n-1) \Rightarrow S(n)$*.**
- **Conclusion: By way of induction, we can conclude that $\forall n \geq 0$, *S(n)*.**

Proving Correctness with Strong Induction

- **Proving the Recursive Algorithm Works:**
- **Induction Hypothesis:** For each $n \geq 0$, *let S(n) be the statement "The recursive* algorithm works for *every instance of size n."*
- **Goal: Our goal is to prove that $\forall n \geq 0$, *S(n), i.e. that the recursive algorithm*** works for *every instance.*
- **Proof Outline: The proof is by strong induction on *n*.**
- **Base Case: Proving *S(0) involves showing that the algorithm works for the*** base cases of size $n = 0$.

Proving Correctness with Strong Induction

- **Induction Step: The statement *S(0), S(1), S(2), . . . , S(n – 1) ⇒ S(n)* *is proved*** as follows. First assume that the algorithm works for every instance of size strictly smaller than *n, and then prove that it works for every instance* of size *n*.
- To prove that the algorithm works for every instance of size *n, consider*
- an arbitrary instance of size *n. The algorithm constructs subinstances that* are strictly smaller.
- By our induction hypothesis we know that our algorithm works for these. Hence, the recursive calls return the correct solutions. On the friends level of abstraction, we proved that the algorithm constructs the correct solutions to our instance from the correct solutions to the subinstances.
- Hence, the algorithm works for this arbitrary instance of size *n. The*
- *S(n) follows.*
- **Conclusion: By way of strong induction, we can conclude that ∀*n* ≥ *0, S(n),*** i.e., the recursive algorithm works for every instance.

EXAMPLES OF RECURSIVE ALGORITHMS- merge sort

- **Sorting and Selecting Algorithms**
- The classic divide-and-conquer algorithms aremerge sort and quick sort. They both have the following basic structure.
- **General Recursive Sorting Algorithm:**
- Take the given list of objects to be sorted (numbers, strings, student records, etc.).
- Split the list into two sublists.
- Recursively have friends sort each of the two sublists.
- Combine the two sorted sublists into one entirely sorted list.
- This process leads to four different algorithms, depending on the following factors (see Exercise 9.1.1):

EXAMPLES OF RECURSIVE ALGORITHMS

- **Sizes:** Split the list into two sub lists each of size $n/2$,
- **Work:** Do you put minimal effort into splitting the list but put lots of effort into recombining the sub lists, or put lots of effort into splitting the list but put minimal
- effort into recombining the sub lists?

Merge Sort (Minimal Work to Split in Half)

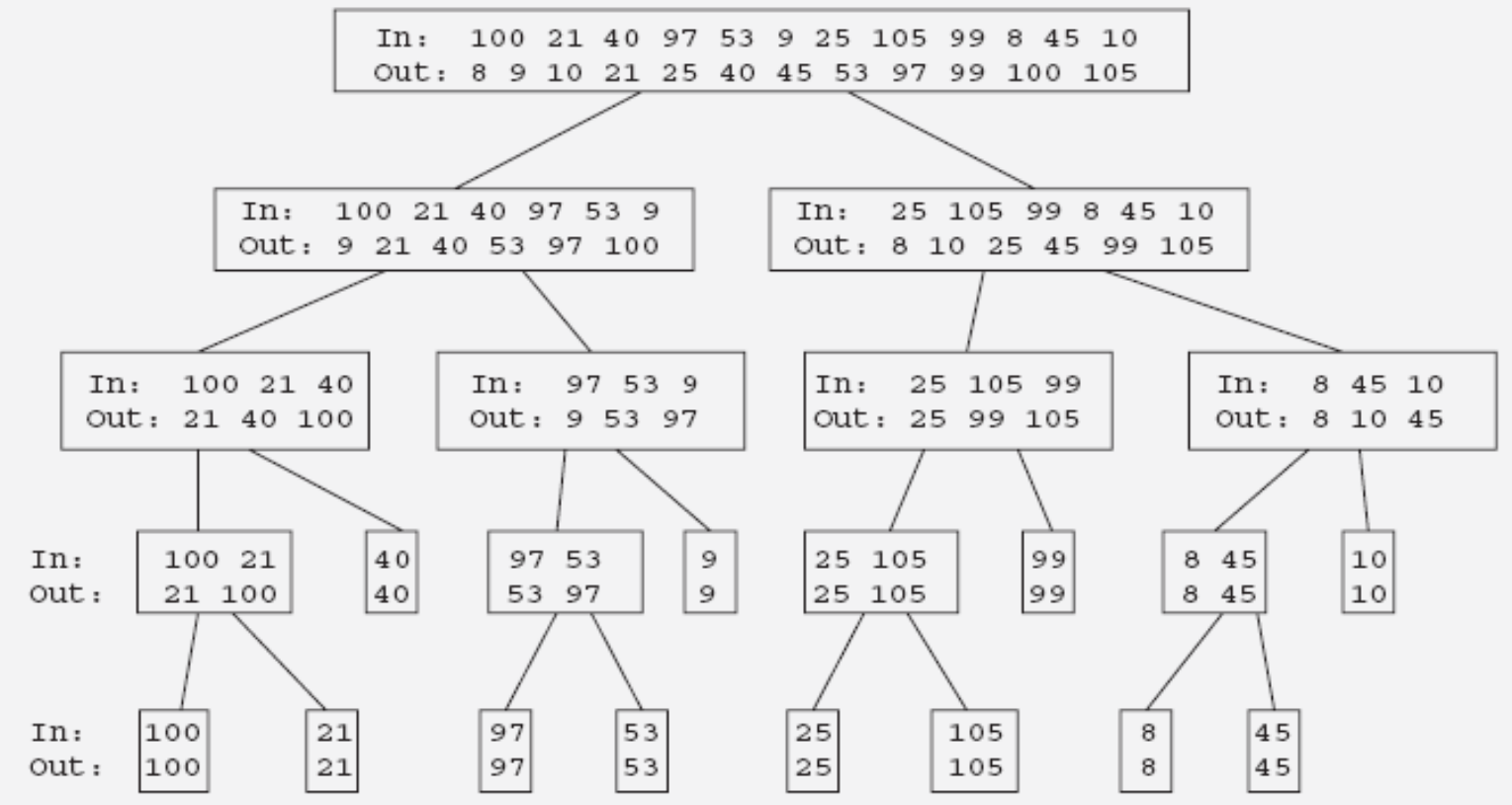
- This is the classic recursive algorithm.
- **Friend's Level of Abstraction:** Recursively give one friend the first half of the input to sort and another friend the second half to sort. Then combine these two sorted sub lists into one completely sorted list. This combining process is referred to as *merging*.
- **Size:** The size of an instance is the number of elements in the list. If this is at least two, then the sub lists are smaller than the whole list. On the other hand, if the list contains only one element, then by default it is already sorted and nothing needs to be done

EXAMPLES OF RECURSIVE ALGORITHMS

- **Generalizing the Problem:** If the input is assumed to be received in an array indexed from 1 to *n, then the second half of the list is not a valid instance, because it is not* indexed from1.Hence,we redefine the preconditions of the sorting problem to require as input both an array *A and a subrange [i, j]. The postcondition is that the specified* sublist is to be sorted in place.
- **Running Time:** Let *T(n) be the total time required to sort a list of n elements. This* total time consists of the time for two subinstances of half the size to be sorted, plus (*n) time for merging the two sublists together. This gives the recurrence relation T(n) = 2T(n/2) + (n).*. In this example, log*a* /log*b* = *log* 2/ log 2 = 1 and *f (n) = (n1), so c = 1. Because* Log*a*/log*b* = *c, the technique concludes that the time is dominated by all levels and* ***T(n) = (f (n) logn) = (n logn).***

EXAMPLES OF RECURSIVE ALGORITHMS – MERGE SORT

- **Tree of Stack Frames**: The following is a tree of stack frames for a concrete example:



ACKERMANN'S FUNCTION

- If you are wondering just how slowly a program can run, consider the algorithm below.
- Assume the input parameters *n and k are natural numbers*
- **Algorithm:**

```
algorithm A(k, n)
    if( k = 0) then
        return( n +1 + 1 )
    else
        if( n = 0) then
            if( k = 1) then
                return( 0 )
            else
                return( 1 )
        else
            return( A(k − 1, A(k, n − 1)))
        end if
    end if
end algorithm
```

ACKERMANN'S FUNCTION

Recurrence Relation: Let $T_k(n)$ denote the value returned by $A(k, n)$. This gives $T_0(n) = 2 + n$, $T_1(0) = 0$, $T_k(0) = 1$ for $k \geq 2$, and $T_k(n) = T_{k-1}(T_k(n-1))$ for $k > 0$ and $n > 0$.

Solving:

$$T_0(n) = 2 + n$$

$$T_1(n) = T_0(T_1(n-1)) = 2 + T_1(n-1) = 4 + T_1(n-2)$$
$$= 2i + T_1(n-i) = 2n + T_1(0) = 2n$$

$$T_2(n) = T_1(T_2(n-1)) = 2 \cdot T_2(n-1) = 2^2 \cdot T_2(n-2) = 2^i \cdot T_2(n-i) = 2^n \cdot T_2(0) = 2^n$$

$$T_3(n) = T_2(T_3(n-1)) = 2^{T_3(n-1)} = 2^{2^{T_3(n-2)}} = \left[\underbrace{2^{2^{2^{\cdots^2}}}}_{i}\right]^{T_3(n-i)} = \left[\underbrace{2^{2^{2^{\cdots^2}}}}_{n}\right]^{T_3(0)} = \underbrace{2^{2^{2^{\cdots^2}}}}_{n}$$

ACKERMANN'S FUNCTION

$$T_4(0) = 1.\ T_4(1) = T_3(T_4(0)) = T_3(1) = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}_{1} = 2.$$

$$T_4(2) = T_3(T_4(1)) = T_3(2) = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}} = 2^2 = 4.$$

$$T_4(3) = T_3(T_4(2)) = T_3(4) = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{4} = 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65,536.$$

Note that

$$\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{5} = 2^{65,536} \approx 10^{21,706}$$

while the number of atoms in the universe is less than 10^{100}. We have

$$T_4(4) = T_3(T_4(3)) = T_3(65,536) = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{65,536}$$

Ackermann's function is defined to be $A(n) = T_n(n)$. We see that $A(4)$ is bigger than any number in the natural world. $A(5)$ is unimaginable.

ACKERMANN'S FUNCTION

- **Running Time:** The only way that the program builds up a big number is by continually incrementing it by one. Hence, the number of times one is added is at least as huge as the value *Tk (n) returned*.

RECURSION ON TREES

- One key application of recursive algorithms is to perform actions on trees, because trees themselves have a recursive definition. Terminology for trees is summarized in the following table:

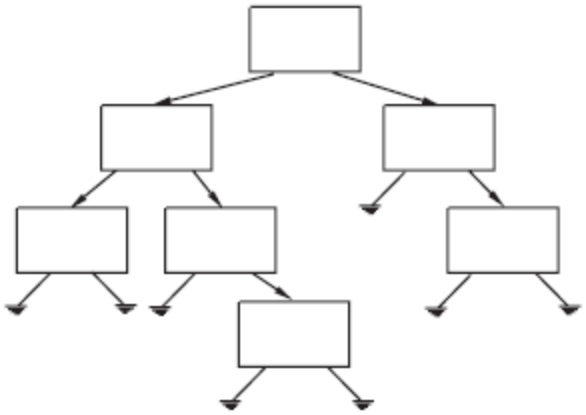
RECURSION ON TREES

Term	Definition
Root	Node at the top
RootInfo(*tree*)	The information stored at the root node
Child of node *u*	One of the nodes just under node *u*
Parent of node *u*	The unique node immediately above node *u*
Siblings	Nodes with same parent
Ancestors of node *u*	The nodes on the unique path from the root to the node *u*
Descendants of node *u*	All the nodes below node *u*
Leaf	A node with no children
Height of tree	The maximum level. Some definitions say that a tree with a single node has height 0, others say height 1. It depends on whether you count nodes or edges.
Depth of node *u*	The number of nodes (or edges) on the path from the root to *u*.
Binary tree	Each node has at most two children. Each of these is designated as either the right child or the left child.
leftSub(*tree*)	Left subtree of root
rightSub(*tree*)	Right subtree of root

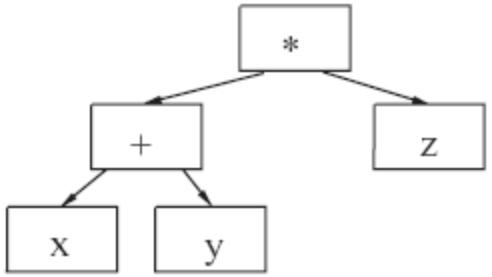
RECURSION ON TREES

- **Recursive Definition of Tree:** A tree is either**:**
- an empty tree (zero nodes) or
- a root node with some subtrees as children.
- A *binary tree is a special kind of tree where each node has a right and a left* subtree.

Binary Tree



Tree representing (x + y) * z



RECURSION ON TREES

- **Number of Nodes in a Binary Tree**
- We will now develop a recursive algorithm that will compute the number of nodes in a binary tree.
- **Specifications:**
- ***Preconditions: The input is any binary tree. Trees with an empty subtree are valid*** trees. So are trees consisting of a single node and the empty tree.
- ***Postconditions: The output is the number of nodes in the tree.***
- **Size: The *size of an instance is the number of nodes in it.***

RECURSION ON TREES

Code:

```
algorithm NumberNodes(tree)
⟨pre-cond⟩: tree is a binary tree.
⟨post-cond⟩: Returns the number of nodes in the tree.
begin
    if( tree = emptyTree ) then
        result( 0 )
    else
        result( NumberNodes(leftSub(tree))
            + NumberNodes(rightSub(tree)) + 1 )
    end if
end algorithm
```

Running Time: Because there is one recursive stack frame for each node in the tree and each stack frame does a constant amount of work, the total time is linear in the number of nodes in the input tree, i.e., $T(n) = \Theta(n)$. Proved another way, the recurrence relation is $T(n) = T(n_{left}) + T(n_{right}) + \Theta(1)$. Plugging the guess $T(n) = cn$ gives $cn = cn_{left} + cn_{right} + \Theta(1)$, which is correct because $n = n_{left} + n_{right} + 1$.

TREE TRAVERSALS

- A task one needs to be able to perform on a binary tree is to traverse it, visiting each node once, in one of three defined orders
- Recursion, on the other hand, provides a very easy and slick algorithm for traversing a binary tree. Such a tree is composed of three parts. There is the root node, its left subtree, and its right subtree.
- The three classic orders to visit the nodes of a binary tree are *prefix, infix, and postfix, in which the root is visited before,* between, or after its left and right subtrees are visited.

TREE TRAVERSALS

```
algorithm PreFix(tree))
⟨pre-cond⟩: tree is a binary tree.
⟨post-cond⟩: Visits the nodes
in prefix order.
begin
    if(tree ≠ emptyTree) then
        put rootInfo(tree)
        PreFix(leftSub(tree))
        PreFix(rightSub(tree))
    end if
end algorithm
```

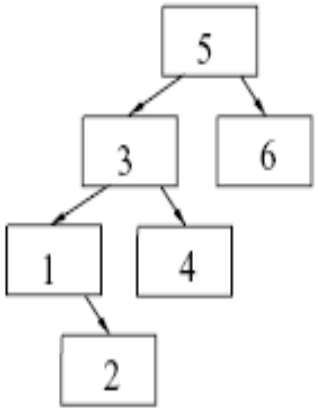
```
algorithm InFix(tree))
⟨pre-cond⟩: tree is a binary tree.
⟨post-cond⟩: Visits the nodes
in infix order.
begin
    if(tree ≠ emptyTree) then
        InFix(leftSub(tree))
        put rootInfo(tree)
        InFix(rightSub(tree))
    end if
end algorithm
```

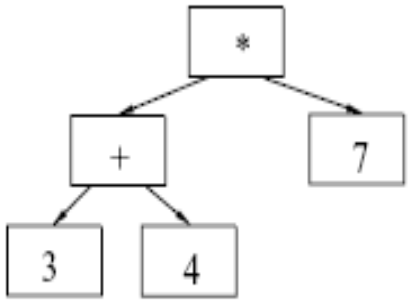
```
algorithm PostFix(tree))
⟨pre-cond⟩: tree is a binary tree.
⟨post-cond⟩: Visits the nodes
in postfix order.
begin
    if( tree ≠ emptyTree ) then
        PostFix(leftSub(tree))
        PostFix(rightSub(tree))
        put rootInfo(tree)
    end if
end algorithm
```

TREE TRAVERSALS

The following order is produced if you tracing out these computations on the two trees displayed below:

PreFix	InFix	PostFix
5 3 1 2 4 6	1 2 3 4 5 6	2 1 4 3 6 5
* + 3 4 7	3 + 4 * 7	3 4 + 7 *





TREE TRAVERSALS

- *PreFix visits the nodes in the same order that a depth-first search finds the nodes.*

```
algorithm IterativeTraversal(tree)
⟨ pre-cond ⟩: tree is a binary tree. As usual, each node has a value and pointers to the
roots of its left and right subtrees. In addition, each node has a pointer to its parent.
⟨ post-cond ⟩: Does an infix traversal of tree.
 begin
      element = root(tree)        % Current node in traversal
      count = zero                % Current count of nodes
      loop
           ⟨loop-invariant⟩: element is some node in the tree, and some nodes
           have been visited.
           if( element has a left child and it has not been visited ) then
                    element = leftChild(element)
           elseif( element has no left child or its left child have been visited
                  and element has not been visited ) then
                     visit element
```

TREE TRAVERSALS

```
        elseif( element's left subtree and element itself has been visited
              and element has a right child and it has not been visited ) then
                  element = rightChild(element)
        elseif( element's left subtree, element itself, and right subtree have been
              visited and element has a parent ) then
                  element = parent(element)
        elseif( Everything has been visited and element is the root of the global
              tree ) then
             exit
        end if
    end loop
end algorithm
```

TREE TRAVERSALS

- **Simple Examples**
- Here is a list of problems involving binary trees.
- 1. Return themaximum of data fields of nodes.
- 2. Return the height of the tree.
- 3. Return the number of leaves in the tree. (A harder one.)
- 4. Copy the tree.

TREE TRAVERSALS

Maximum: Given a binary tree, your task is to determine its maximum value.

```
algorithm Max(tree)
⟨pre-cond⟩: tree is a binary tree.
⟨post-cond⟩: Returns the maximum of data fields of nodes.
begin
    if( tree = emptyTree ) then
        result( −∞ )
    else
        result( max(Max(leftSub(tree)), Max(rightSub(tree)), rootData(tree)))
    end if
end algorithm
```

TREE TRAVERSALS

- **Height:** In this problem, your task it to find the height of your binary tree.

algorithm *Height*(*tree*)

⟨*pre-cond*⟩: *tree* is a binary tree.

⟨*post-cond*⟩: Returns the height of the tree measured in nodes, e.g., a tree with one node has height 1.

```
begin
    if(tree = emptyTree ) then
        result( 0 )
    else
        result( max(Height(leftSub(tree)), Height(rightSub(tree))) + 1 )
    end if
end algorithm
```

NUMBER OF LEAVES

- For this, the number of leaves in the entire tree is the sum of the numbers in the left and right subtrees. If the tree has one subtree, but the other is empty, then this same algorithm still works. If the tree is empty, then it has zero leaves.

```
algorithm NumberLeaves(tree)
⟨pre-cond⟩: tree is a binary tree.
⟨post-cond⟩: Returns the number of leaves in the tree.
begin
    if( tree = emptyTree ) then
        result( 0 )
    else if( leftSub(tree) = emptyTree and rightSub(tree) = emptyTree ) then
        result( 1 )
    else
        result( NumberLeaves(leftSub(tree)) + NumberLeaves(rightSub(tree)) )
    end if
end algorithm
```

HEAP SORT AND PRIORITY Q'S

- Heap sort is a fast sorting algorithm that is easy to implement
- **Completely Balanced Binary Tree: We will visualize the values being sorted as**
- stored in a binary tree that is completely balanced, i.e., every level of the tree is completely full except for the bottom level, which is filled in from the left.
- **Array Implementation of a Balanced Binary Tree**In actuality, the values are stored in a simple array *A[1, n]*. The mapping between the visualized tree structure and the actual array structure is done by indexing the nodes of the tree 1, 2, 3, . . . , *n, starting with the root of the tree* and filling each level in from left to right.
- The root is stored in *A[1]*.
- The parent of *A[i] is A[i/2*].
- The left child of *A[i] is A[2 x i]*.
- The right child of *A[i] is A[2 x i + 1]*.
- The node in the far right of the bottom level is stored in *A[n]*.
- If *2i + 1 > n, then the node does not have a right child.*

HEAP SORT AND PRIORITY Q'S

- **Definition of a Heap:** A heap imposes a partial order (see Section 14.6) on the set of values, requiring that the value of each node be greater than or equal to that of each of the node's children. There are no rules about whether the left or the right child is larger. See Figure

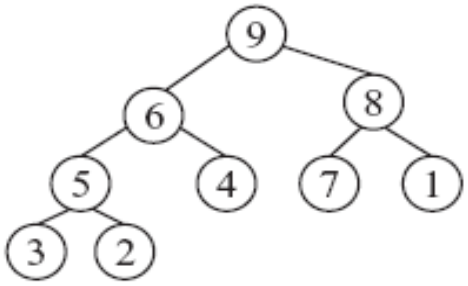


Figure 10.3: An example of nodes ordered into a heap.

- **Maximum at Root:** An implication of the heap rules is that the root contains the maximum value. The maximum may appear repeatedly in other places as well.

HEAP SORT AND PRIORITY Q'S

- **The Heapify Problem:**
- **Specifications:**
- ***Precondition:*** *The input is a balanced binary tree such that its left and right* subtrees are heaps. (That is, it is a heap except that its root might not be larger than that of its children.)
- ***Postcondition:*** *Its values are rearranged in place to make it complete heap*

HEAP SORT AND PRIORITY Q'S

- **Recursive Algorithm:** The first task in making this tree into a heap is to put its maximum value at the root. See Figure 10.4. Because the left and right subtrees are heaps, the maxima of these trees are at their roots.
- Hence, the maximum of the entire tree is either at the root, at its left child node, or at its right child node. You find the maximum among these three. If the maximum is at the root, then you are finished.
- Otherwise, for the purpose of discussion, assume that the
- maximum is in the root's left child. Swap this maximum value with that of the root. The root and the right subtree now form a heap, but the left subtree might not.

HEAP SORT AND PRIORITY Q'S

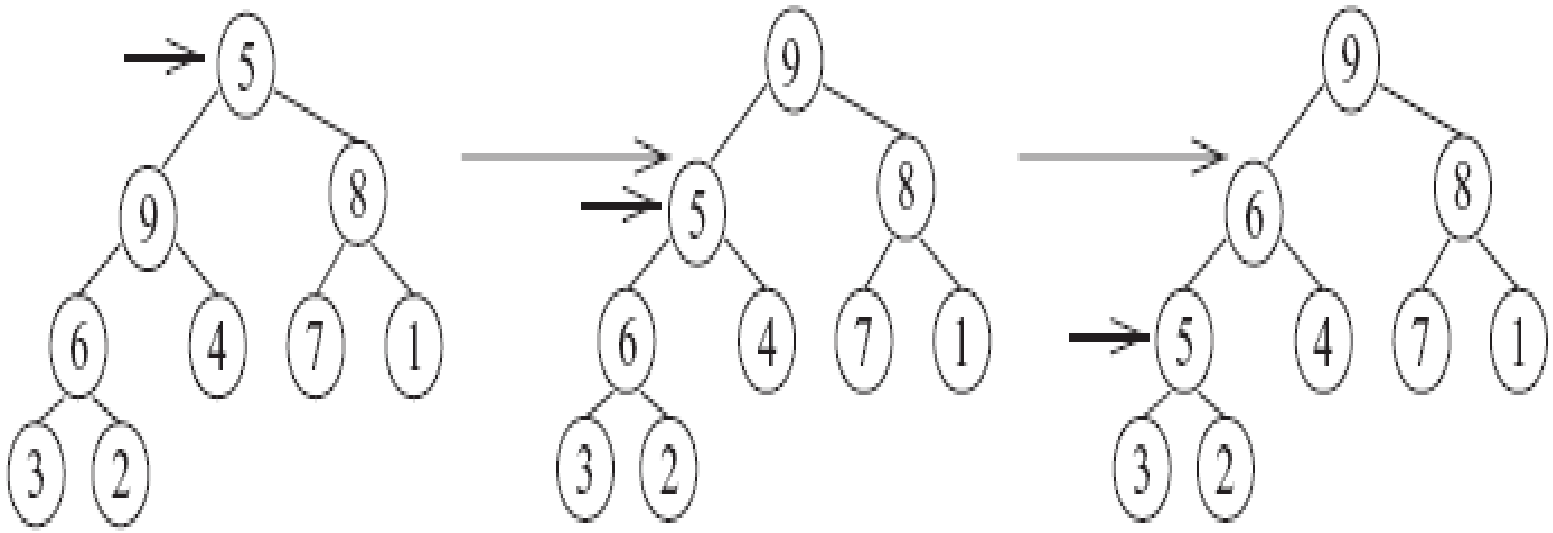


Figure 10.4: An example computation of *Heapify*.

HEAP SORT AND PRIORITY Q'S

Code:

```
algorithm Heapify(r)
⟨pre-cond⟩: The balanced binary tree rooted at A[r] is such that its left
and right subtrees are heaps.
⟨post-cond⟩: Its values are rearranged in place to make it complete
heap.
begin
    if(A[rightchild(r)] is max of {A[r], A[rightchild(r)],
    A[leftchild(r)]} ) then
        swap(A[r], A[rightchild(r)])
        Heapify(rightchild(r))
    elseif(A[leftchild(r)] is max of {A[r], A[rightchild(r)],
    A[leftchild(r)]} ) then
        swap(A[r], A[leftchild(r)])
        Heapify(leftchild(r))
    else % A[r] is max of {A[r], A[rightchild(r)], A[leftchild(r)]}
        exit
    end if
end algorithm
```

Running Time: $T(n) = 1 \cdot T(n/2) + \Theta(1)$. From Chapter 27 we know that $\frac{\log a}{\log b} = \frac{\log 1}{\log 2} = 0$ and $f(n) = \Theta(n^0)$, so $c = 0$. Because $\frac{\log a}{\log b} = c$, we conclude that time is dominated by all levels and $T(n) = \Theta(f(n) \log n) = \Theta(\log n)$.

HEAP SORT AND PRIORITY Q'S

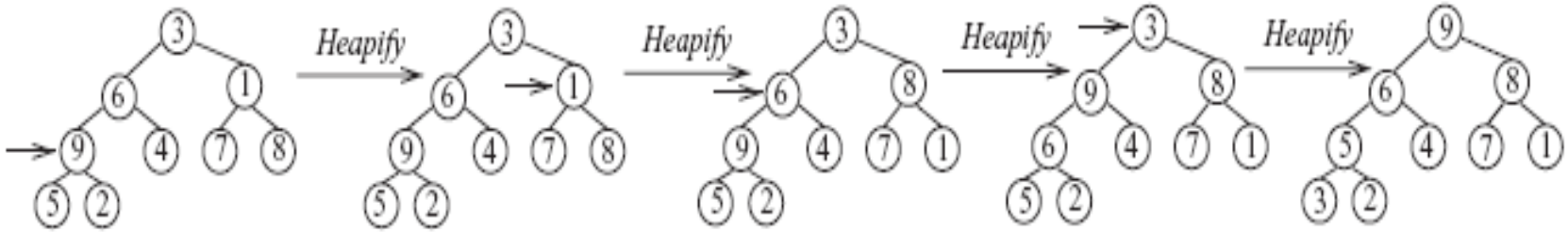


Figure 10.5: An example of the iterative version of *MakeHeap*.

Code:

algorithm *MakeHeap*( )

⟨ ***pre-cond***⟩**:** The input is an array of numbers, which can be viewed as a balanced binary tree of numbers.

⟨ ***post-cond***⟩**:** Its values are rearranged in place to make it a heap.

begin

loop $k = \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor - 1, \lfloor \frac{n}{2} \rfloor - 2, \ldots, 2, 1$

Heapify(k)

end loop

end algorithm

HEAP SORT AND PRIORITY Q'S

- **The HeapSort Problem:** Specifications:
- *Precondition: The input is an array of numbers.*
- *Postcondition: Its values are rearranged in place to be in sorted order.*
- **Algorithm:** The loop invariant is that for some $i \in [0, n]$, *the* $n - i$ *largest elements* have been removed and are sorted on the side, and the remaining i *elements* form a heap. See Figures 10.6 and 10.7. The loop invariant is established for $i = n$ *by forming a heap from the numbers using the MakeHeap algorithm.* When $i = 0$, *the values are sorted.*
- Suppose that the loop invariant is true for i. *The maximum of the remaining* values is at the root of the heap. Remove it and put it in its sorted place on the left end of the sorted list. Take the bottom right-hand element of the heap, and fill the newly created hole at the root. This maintains the correct shape of the tree. The tree now has the property that its left and right subtrees are heaps. Hence, you can use *Heapify to make it into a heap. This maintains the loop invariant* while decreasing i *by one.*

HEAP SORT AND PRIORITY Q'S

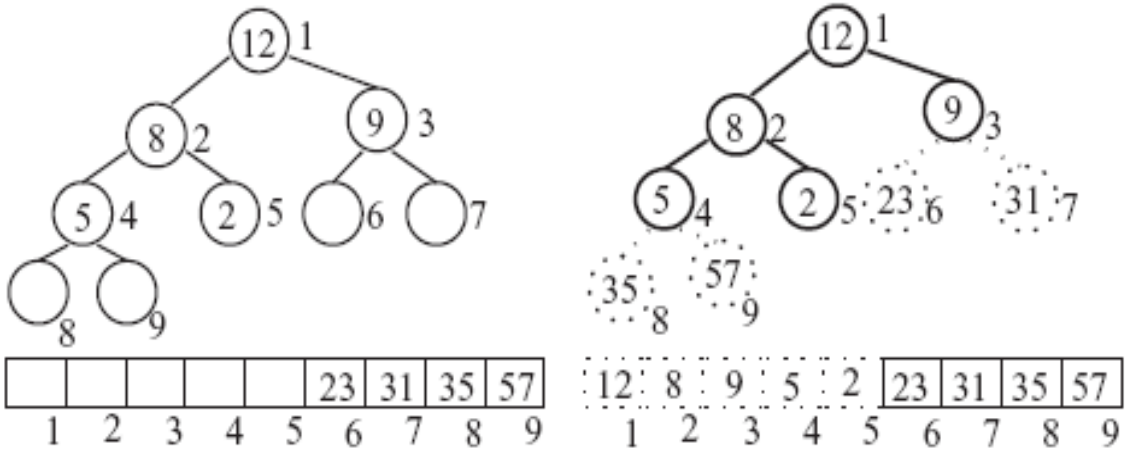


Figure 10.6: The left diagram shows the loop invariant with $n - i = 9 - 5 = 4$ of the largest elements in the array and the remaining $i = 5$ elements forming a heap. The right diagram emphasizes the fact that though a heap is viewed as being stored in a tree, it is actually implemented in an array. When some of the elements are in still in the tree and some are in the array, these views overlap.

HEAP SORT AND PRIORITY Q'S

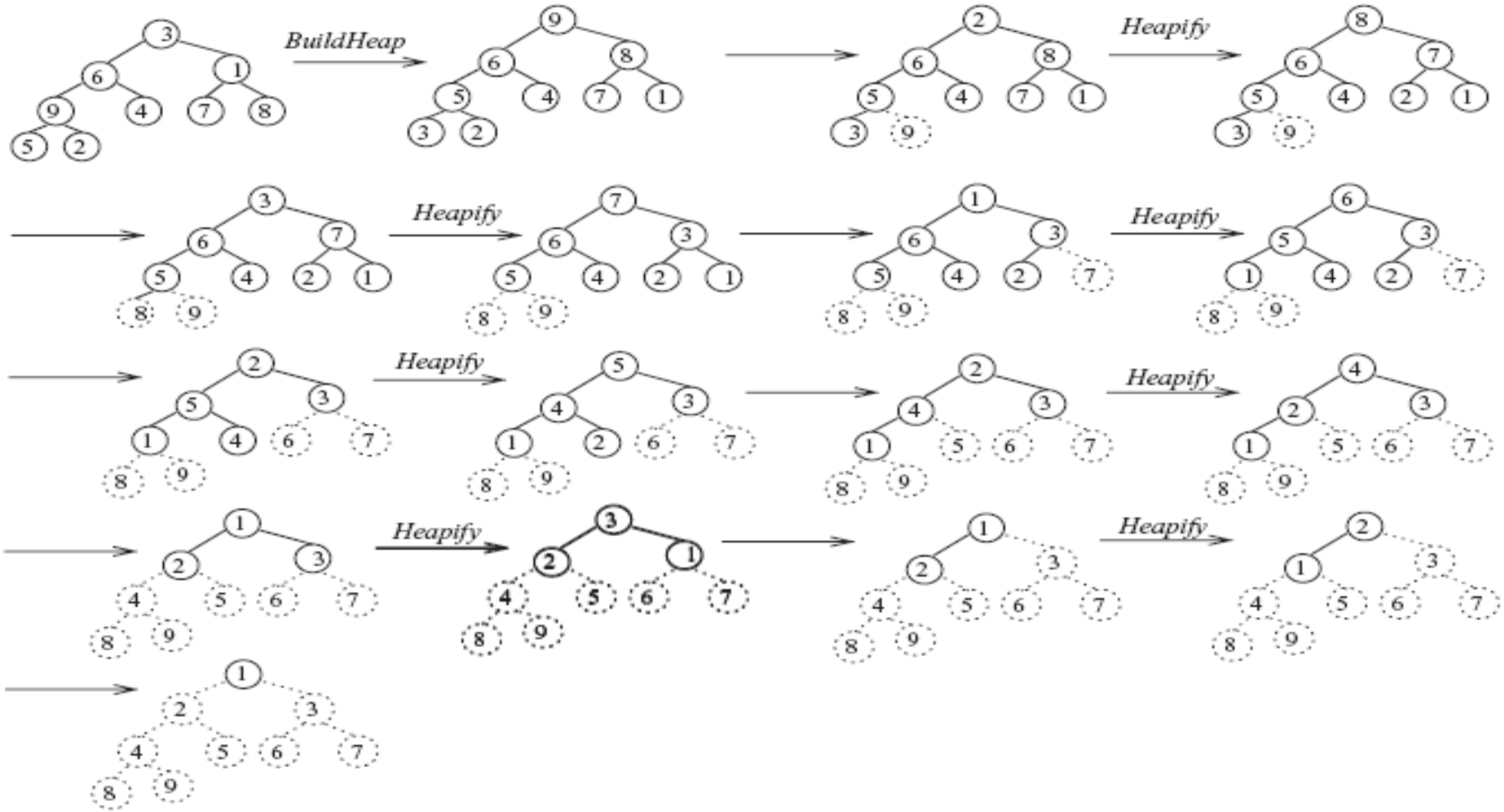


Figure 10.7: An example computation of *HeapSort*.

HEAP SORT AND PRIORITY Q'S

Code:

algorithm *HeapSort*( )

⟨ *pre-cond*⟩: The input is an array of numbers.
⟨ *post-cond*⟩: Its values are rearranged in place to be in sorted order.

begin
    MakeHeap( )
    $i = n$
    loop
        ⟨*loop-invariant*⟩: The $n - i$ largest elements have been removed and are sorted in $A[i + 1, n]$, and the remaining i elements form a heap in $A[1, i]$.
        exit when $i = 1$
        swap($A[root], A[i]$)
        $i = i - 1$
        Heapify(*root*) % On a heap of size i.
    end loop
end algorithm

Running Time: *MakeHeap* takes $\Theta(n)$ time, heapifying a tree of size i takes time $\log(i)$, for a total of $T(n) = \Theta(n) + \sum_{i=n}^{1} \log i$. This sum behaves like an arithmetic sum. Hence, its total is n times its maximum value, i.e., $\Theta(n \log n)$.

PRIORITY QUEUES

- **Priority Queues:** Like stacks and queues, priority queues are an important ADT.
- Definition: A *priority queue consists of:*
- Data: A set of elements, each of which is associated with an integer that is referred to as the *priority of the element*.
- ***Insert an Element: An element, along with its priority, is added to the*** queue.
- ***Change Priority: The priority of an element already in the queue is*** changed. The routine is passed a pointer to the element within the priority queue and its new priority.
- ***Remove an Element: Removes and returns an element of the highest priority*** fromthe queue.

PRIORITY QUEUES

Implementations:

Implementation	Insert Time	Change Time	Remove Time
Sorted in an array or linked list by priority	$O(n)$	$O(n)$	$O(1)$
Unsorted in an array or linked list separate queue for each priority level	$O(1)$	$O(1)$	$O(n)$
(To add, go to correct queue; to delete, find first nonempty queue)	$O(1)$	$O(1)$	O(No. of priorities)
Heaps	$O(\log n)$	$O(\log n)$	$O(\log n)$

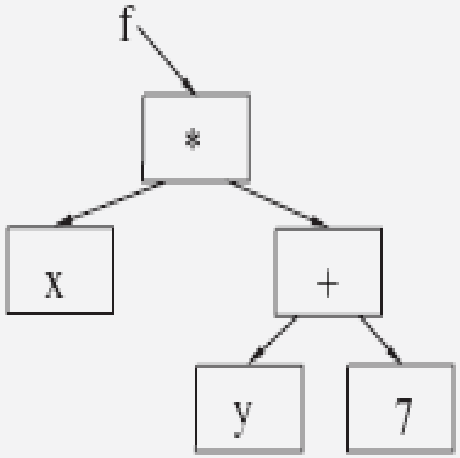
EXPRESSION TREE

- **Recursive Definition of an Expression:**
- Single variables *x, y, and z and single real values are themselves expressions.*
- If *f and g are expressions, then f + g, f – g, f * g, and f/g are also expressions.*
- **Tree Data Structure: The recursive definition of an expression directly mirrors that** of a binary tree. Because of this, a binary tree is a natural data structure for storing an expression. (Conversely, you can use an expression to represent a binary tree.)

EXPRESSION TREE

EXAMPLE 10.5.1 Evaluate Expression

This routine evaluates an expression that is represented by a tree. For example, it can evaluate $f = x * (y + 7)$, with $xvalue = 2$, $yvalue = 3$, and $zvalue = 5$, and return $2 * (3 + 7) = 20$.



EXPRESSION TREE

Code:

```
algorithm Eval(f, xvalue, yvalue, zvalue)
⟨pre-cond⟩: f is an expression whose only variables are x, y, and z, and xvalue,
yvalue, and zvalue are the three real values to assign to these variables.
⟨post-cond⟩: The returned value is the evaluation of the expression at these values
for x, y, and z. The expression is unchanged.
begin
    if( f = a real value ) then
        result( f )
    else if( f = "x" ) then
        result( xvalue )
    else if( f = "y" ) then
        result( yvalue )
    else if( f = "z" ) then
        result( zvalue )
    else if( rootOp(f) = "+" ) then
        result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
            + Eval(rightSub(tree), xvalue,yvalue, zvalue) )
    else if( rootOp(f) = "−" ) then
        result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
            − Eval(rightSub(tree), xvalue, yvalue, zvalue) )
    else if( rootOp(f) = "*" ) then
        result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
            × Eval(rightSub(tree), xvalue, yvalue, zvalue) )
    else if( rootOp(f) = "/" ) then
        result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
            / Eval(rightSub(tree), xvalue, yvalue, zvalue) )
    end if
end algorithm
```

EXPRESSION TREE

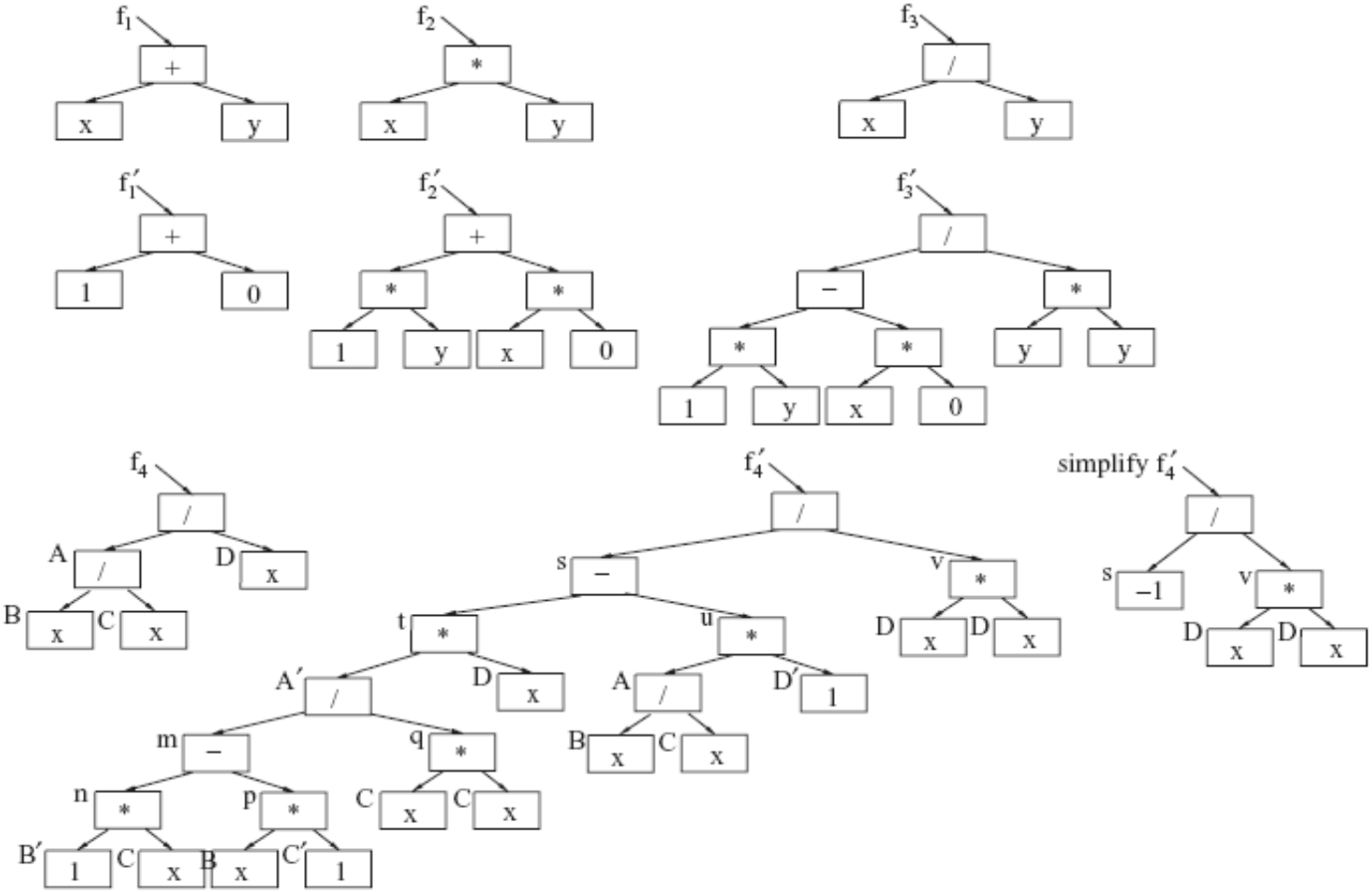


Figure 10.8: Four functions and their derivatives. The fourth derivative has been simplified.

EXPRESSION TREE

EXAMPLE 10.5.3 Simplify Expression

This routine simplifies a given expression. For example, the derivative of $x * y$ with respect to x will be computed to be $1 * y + x * 0$. This should be simplified to y.

Specification:

Preconditions: The input consists of an expression f represented by a tree.

Postconditions: The output is another expression that is a simplification of f. Its nodes should be separate from those of f, and f should remain unchanged.

Code:

```
algorithm Simplify(f)
⟨pre-cond⟩: f is an expression.
⟨post-cond⟩: The output is a simplification of this expression.
begin
    if( f = a real value or a single variable ) then
        result( Copy(f) )
    else % f is of the form (g' op h')
        g = Simplify(leftSub(f))
        h = Simplify(rightSub(f))
```

EXPRESSION TREE

EXAMPLE 10.5.3 Simplify Expression (cont.)

if(one of the following forms applies:

$1 * h = h$ $g * 1 = g$ $0 * h = 0$ $g * 0 = 0$

$0 + h = h$ $g + 0 = g$ $g - 0 = g$ $x - x = 0$

$0 / h = 0$ $g / 1 = g$ $g / 0 = \infty$ $x / x = 1$

$6 * 2 = 12$ $6 / 2 = 3$ $6 + 2 = 8$ $6 - 2 = 4$) then

result(the simplified form)

else

result(g op h)

end if

end if

end algorithm

THANK YOU