Recursion

CS 308 – Data Structures

What is recursion?

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type

When you turn this into a program, you end up with functions that call themselves (*recursive functions*)

```
int f(int x)
{
 int y;

 if(x==0)
    return 1;
 else {
    y = 2 * f(x-1);
    return y+1;
 }
}
```

Problems defined recursively

- There are many problems whose solution can be defined recursively

Example: *n factorial*

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! * n & \text{if } n > 0 \end{cases}$$

(*recursive* solution)

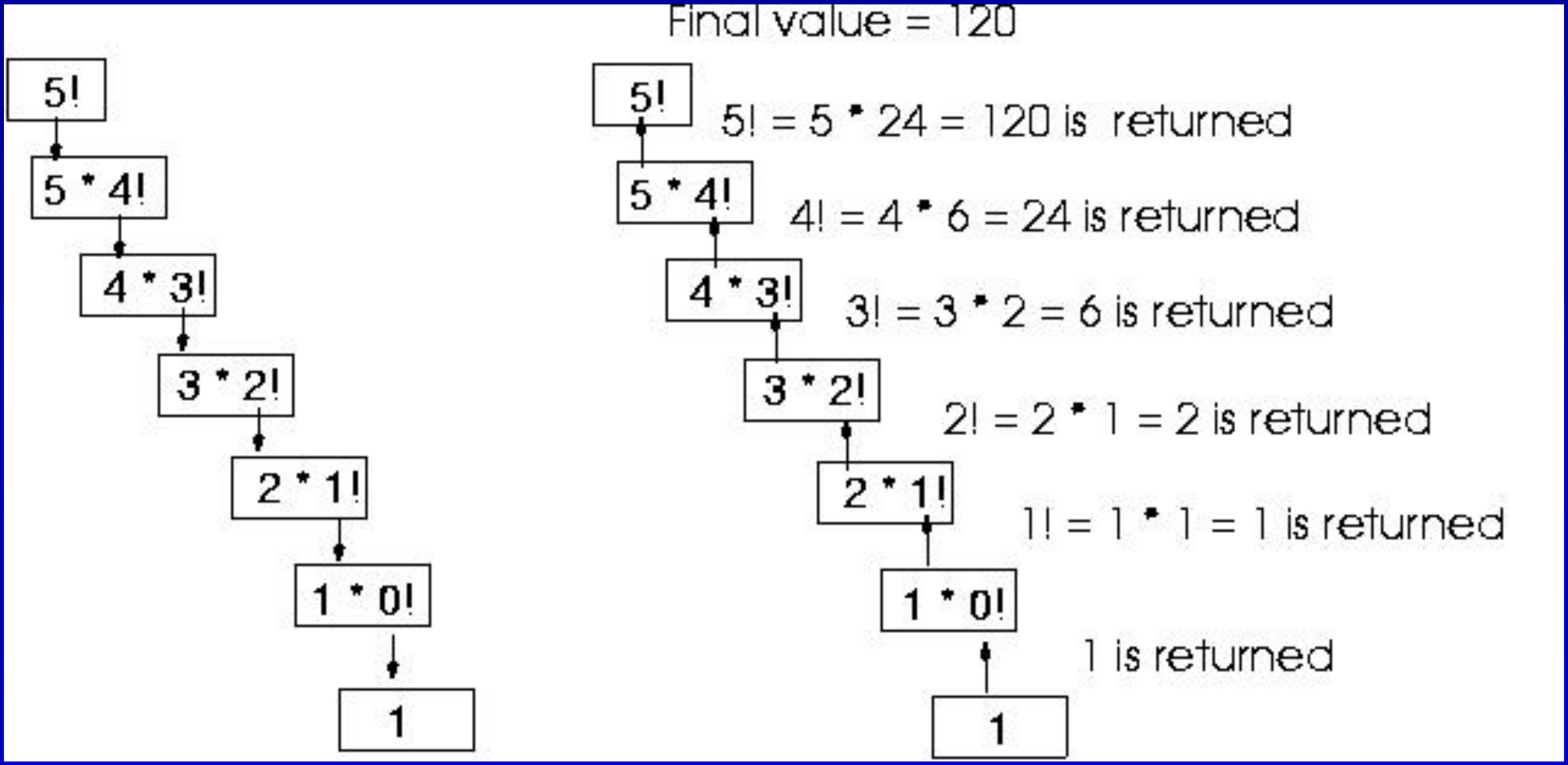
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n-1)*n & \text{if } n > 0 \end{cases}$$

(*closed form* solution)

Coding the factorial function

- Recursive implementation

```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```


Final value = 120
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1! = 1 * 1 = 1 is returned
1 * 0!
1 is returned
1

Coding the factorial function (cont.)

- Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Another example:
n choose *k* (combinations)

- Given *n* things, how many different sets of size *k* can be chosen?

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \quad 1 < k < n \qquad \textit{(recursive solution)}$$

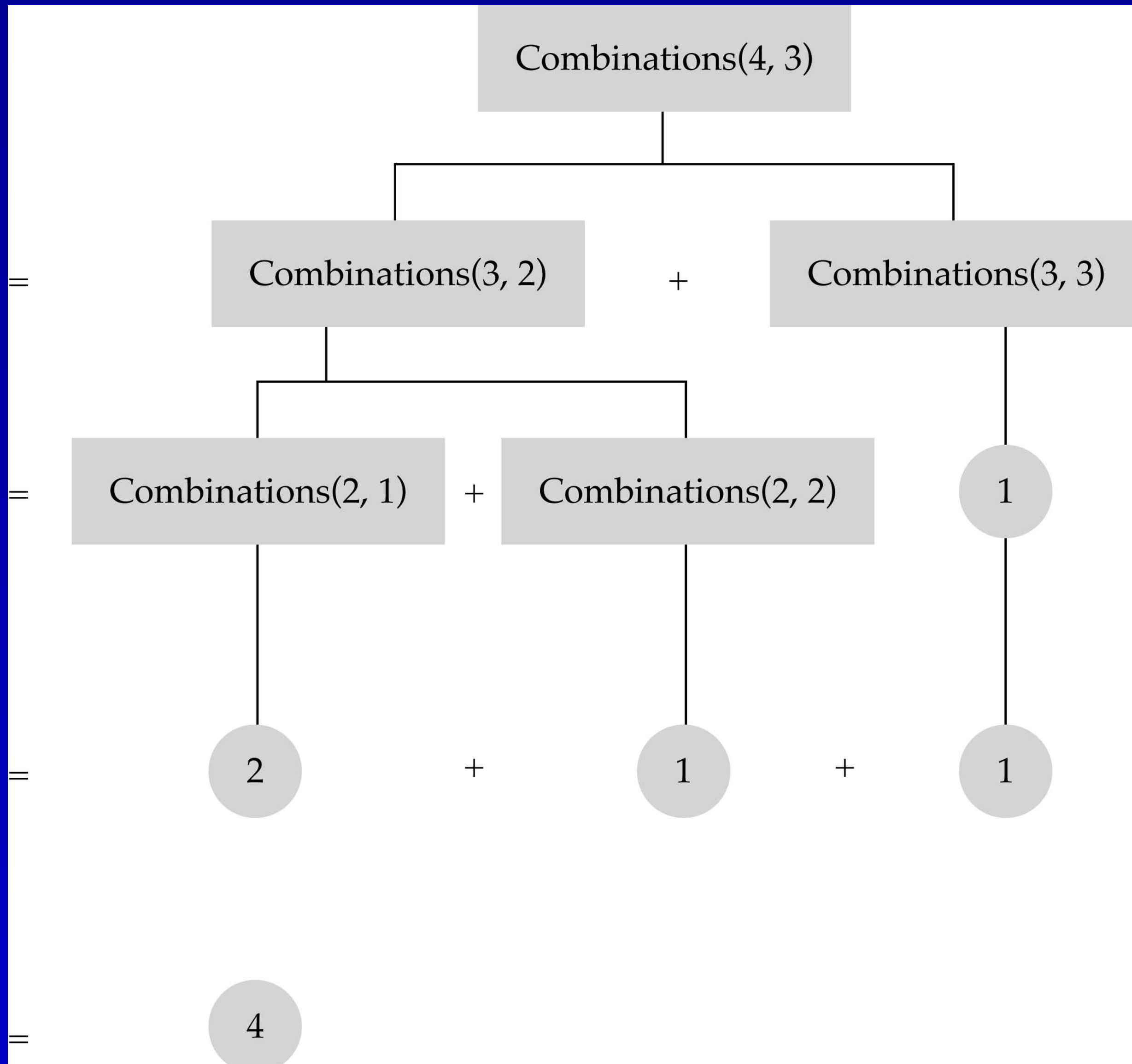
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \qquad \textit{(closed-form solution)}$$

with base cases:

$$\binom{n}{1} = n \quad (k = 1), \quad \binom{n}{n} = 1 \quad (k = n)$$

n choose *k* (combinations)

```
int Combinations(int n, int k)
{
 if(k == 1)  // base case 1
   return n;
 else if (n == k)  // base case 2
   return 1;
 else
   return(Combinations(n-1, k) + Combinations(n-1, k-1));
}
```


Combinations(4, 3)
=
Combinations(3, 2)
+
Combinations(3, 3)
=
Combinations(2, 1)
+
Combinations(2, 2)
1
=
2
+
1
+
1
=
4

Recursion vs. iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

How do I write a recursive function?

- Determine the <u>size factor</u>
- Determine the <u>base case(s)</u>

  (the one for which you know the answer)
- Determine the <u>general case(s)</u>

  (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm

  (use the "Three-Question-Method")

Three-Question Verification Method

1. The Base-Case Question:

   Is there a nonrecursive way out of the function, and does the routine work correctly for this "base" case?

2. The Smaller-Caller Question:

   Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?

3. The General-Case Question:

   Assuming that the recursive call(s) work correctly, does the whole function work correctly?

Recursive binary search

- Non-recursive implementation

```
template<class ItemType>
void SortedType<ItemType>::RetrieveItem(ItemType& item, bool& found)
{
 int midPoint;
 int first = 0;
 int last = length - 1;

 found = false;
 while( (first <= last) && !found) {
   midPoint = (first + last) / 2;
   if (item < info[midPoint])
          last = midPoint - 1;
   else if(item > info[midPoint])
      first = midPoint + 1;
   else {
       found = true;
       item = info[midPoint];
    }
  }
}
```

Recursive binary search (cont'd)

- What is the *size factor*?

  The number of elements in (*info[first] ... info[last]*)

- What is the *base case(s)*?

  (1) If *first* > *last*, return *false*

  (2) If *item==info[midPoint]*, return *true*

- What is the *general case*?

  if *item* < *info[midPoint]* <u>search the first half</u>

  if *item* > *info[midPoint]*, <u>search the second half</u>

Recursive binary search (cont'd)

```
  template<class ItemType>
bool BinarySearch(ItemType info[], ItemType& item, int first, int last)
{
 int midPoint;

 if(first > last)  // base case 1
   return false;
 else {
   midPoint = (first + last)/2;
   if(item < info[midPoint])
     return BinarySearch(info, item, first, midPoint-1);
   else if (item == info[midPoint]) { // base case 2
     item = info[midPoint];
     return true;
   }
   else
     return BinarySearch(info, item, midPoint+1, last);
 }
}
```

Recursive binary search (cont'd)

```
template<class ItemType>
 void SortedType<ItemType>::RetrieveItem
    (ItemType& item, bool& found)
 {
  found = BinarySearch(info, item, 0, length-1);
 }
```

How is recursion implemented?

- What happens when a function gets called?

```
int a(int w)
{
 return w+w;
}


int b(int x)
{
 int z,y;
 ……………… // other statements
 z = a(x) + y;

 return z;
}
```

What happens when a function is called? (cont.)

- An **activation** record is stored into a stack **(run-time stack)**
  1) The computer has to stop executing function ***b*** and starts executing function **a**
  2) Since it needs to come back to function ***b*** later, it needs to store everything about function **b** that is going to need (**x, y, z**, and the place to start executing upon return)
  3) Then, **x** from **a** is bounded to **w** from **b**
  4) Control is transferred to function **a**

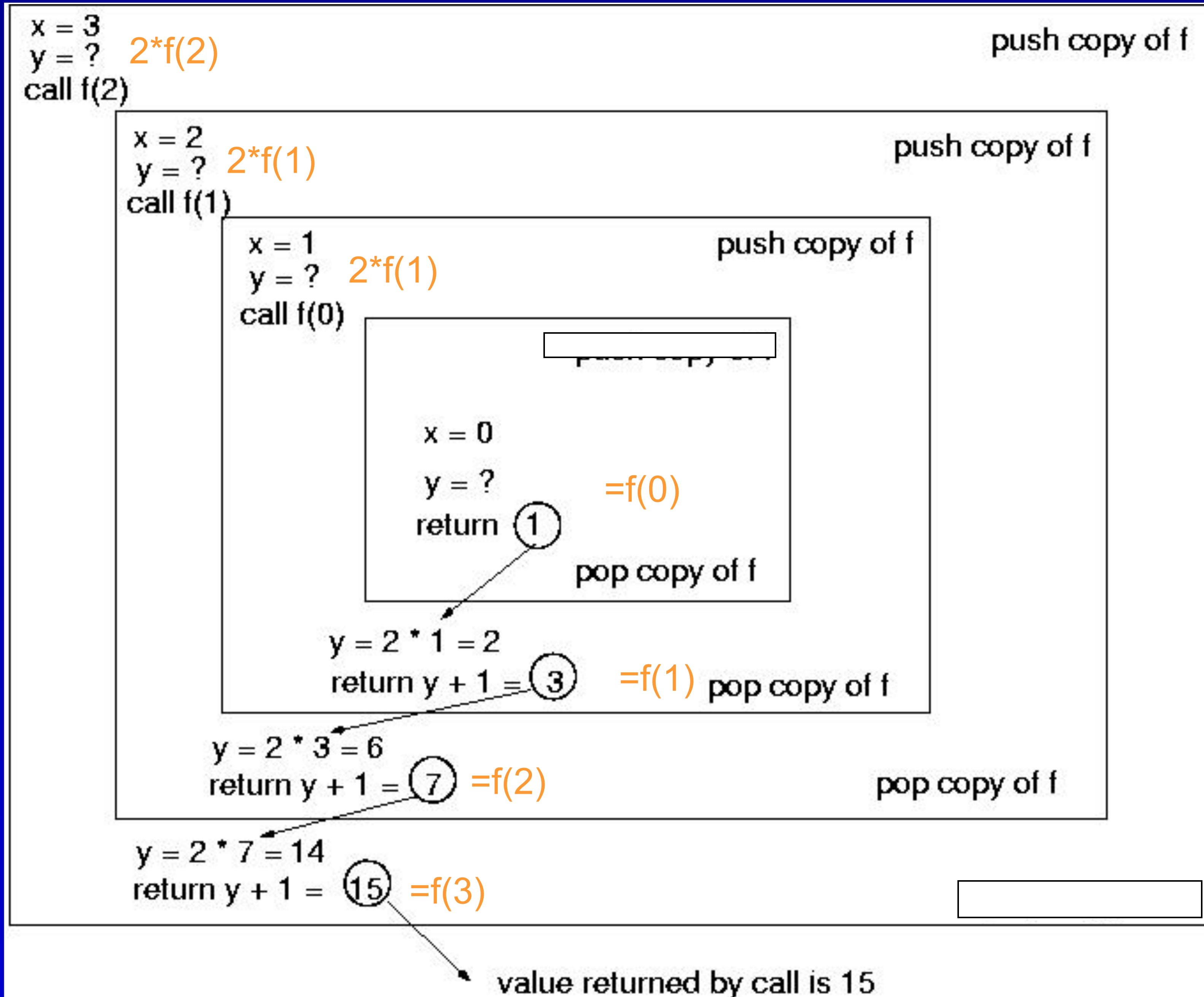
What happens when a function is called? (cont.)

- After function **a** is executed, the activation record is popped out of the run-time stack
- All the old values of the parameters and variables in function **b** are restored and the return value of function **a** replaces **a(x)** in the assignment statement

What happens when a recursive function is called?

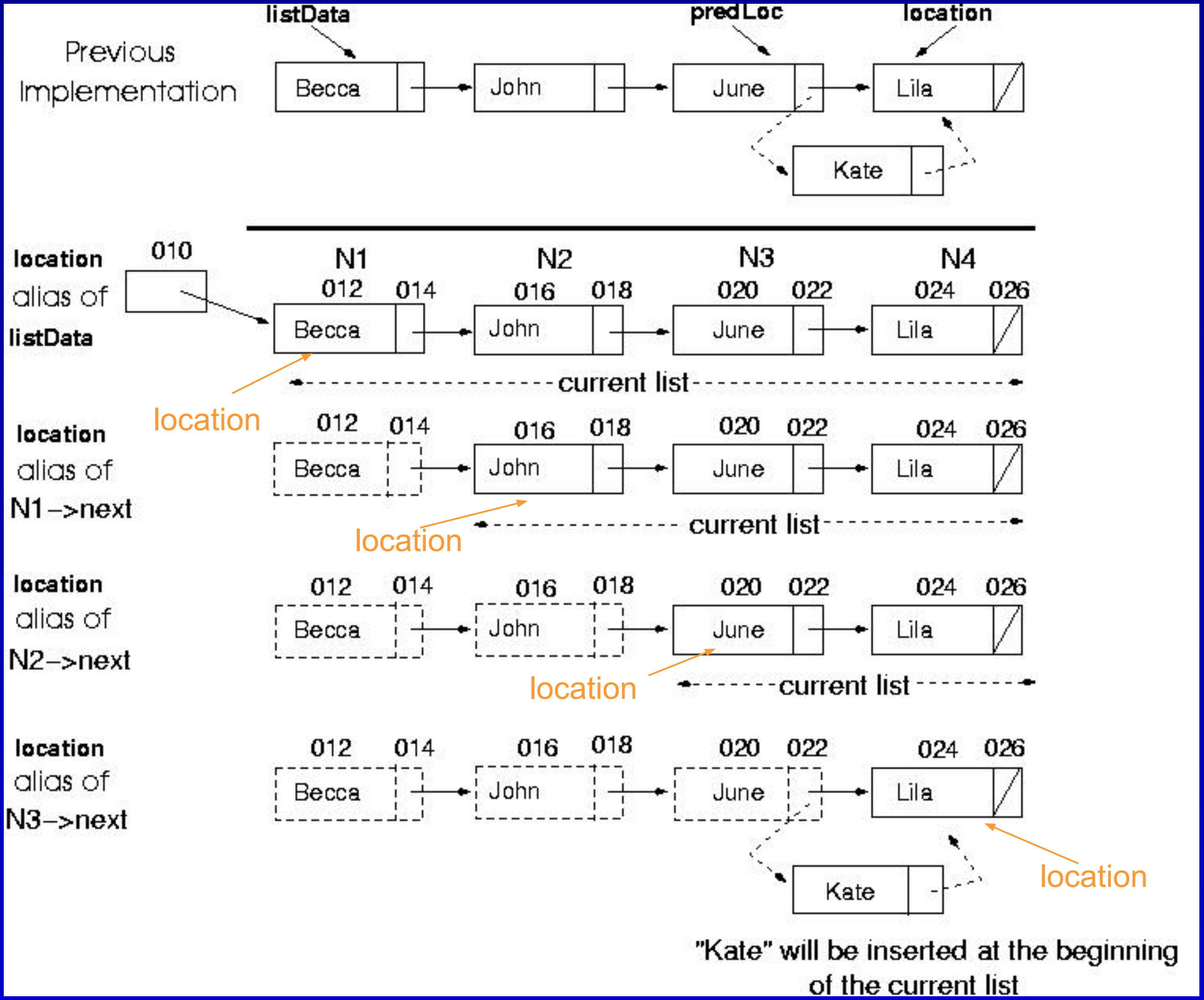
- Except the fact that the calling and called functions have the same name, there is really no difference between recursive and nonrecursive calls

```
int f(int x)
{
 int y;

 if(x==0)
  return 1;
 else {
   y = 2 * f(x-1);
   return y+1;
 }
}
```



x = 3
y = ?
call f(2)
2*f(2)
push copy of f
x = 2
y = ?
call f(1)
2*f(1)
push copy of f
x = 1
y = ?
call f(0)
2*f(1)
push copy of f
x = 0
y = ?
return 1
=f(0)
pop copy of f
y = 2 * 1 = 2
return y + 1 = 3
=f(1)
pop copy of f
y = 2 * 3 = 6
return y + 1 = 7
=f(2)
pop copy of f
y = 2 * 7 = 14
return y + 1 = 15
=f(3)
value returned by call is 15

Recursive InsertItem (sorted list)



Recursive InsertItem (sorted list)

- What is the *size factor*?

  The number of elements in the current list

  What is the *base case(s)*?

  1) If the list is empty, insert item into the empty list
  2) If *item < location->info,* insert item as the first node in the current list

- What is the *general case*?

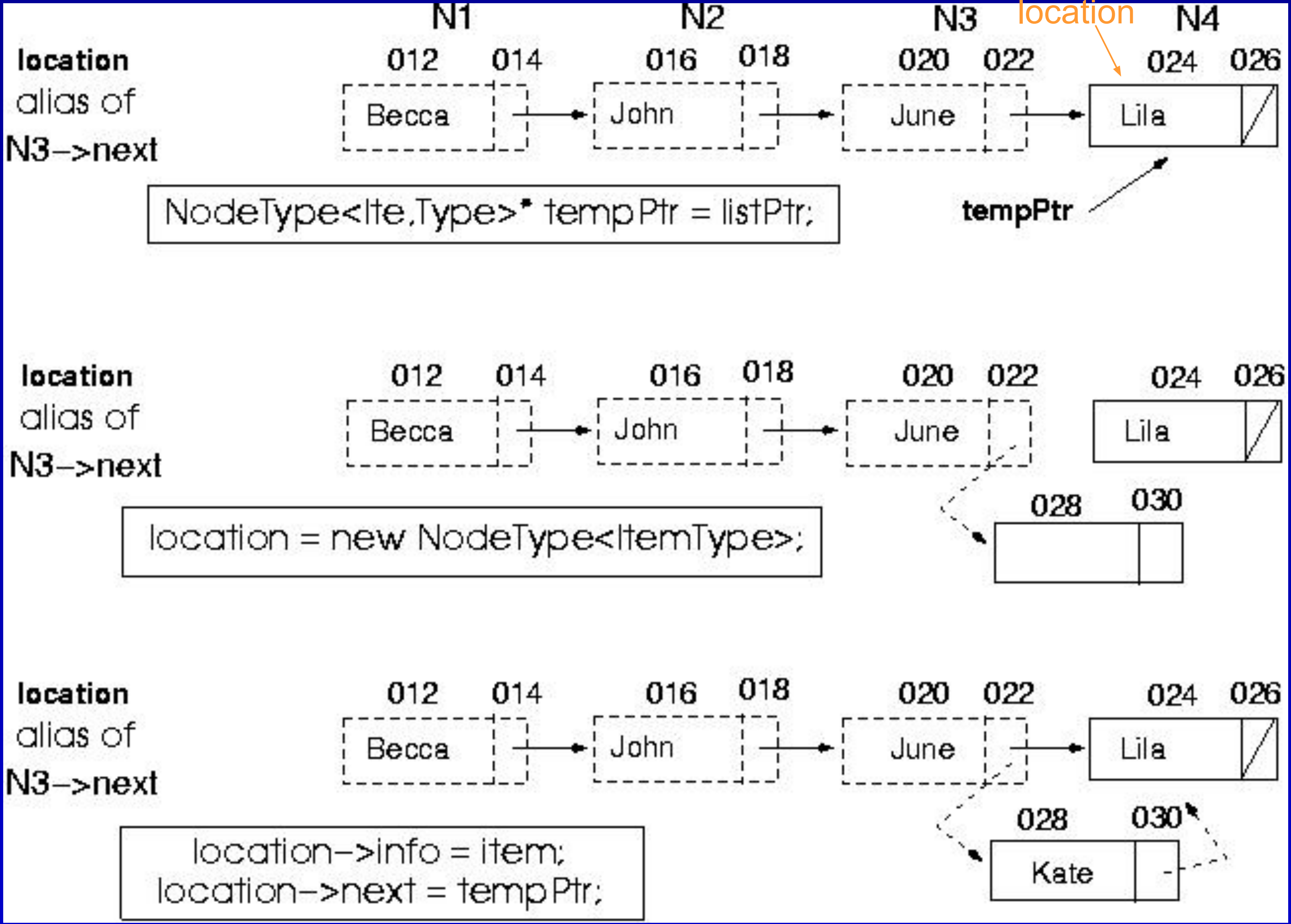
  Insert(location->next, item)

Recursive InsertItem (sorted list)

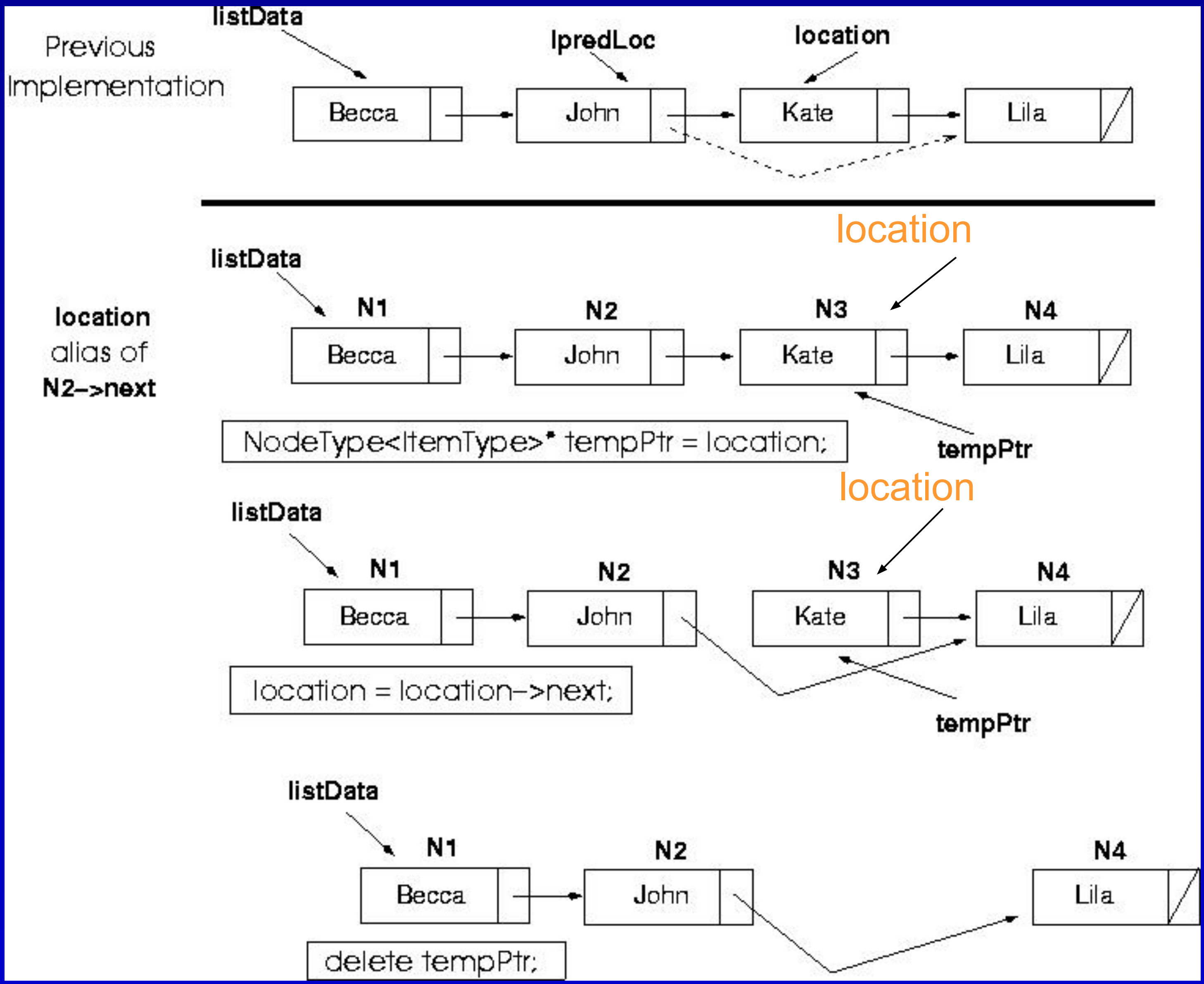
```
template <class ItemType>
void Insert(NodeType<ItemType>* &location, ItemType item)
{
 if(location == NULL) || (item < location->info)) {  // base cases

   NodeType<ItemType>* tempPtr = location;
   location = new NodeType<ItemType>;
   location->info = item;
   location->next = tempPtr;
 }
 else
   Insert(location->next, newItem);  // general case
}
```

```
template <class ItemType>
void SortedType<ItemType>::InsertItem(ItemType newItem)
{
 Insert(listData, newItem);
}
```

- No "predLoc" pointer is needed for insertion



Recursive DeleteItem (sorted list)



Recursive DeleteItem (sorted list) (cont.)

- What is the *size factor*?

  The number of elements in the list

- What is the *base case(s)*?

  If *item == location->info*, delete node pointed by *location*

- What is the *general case*?

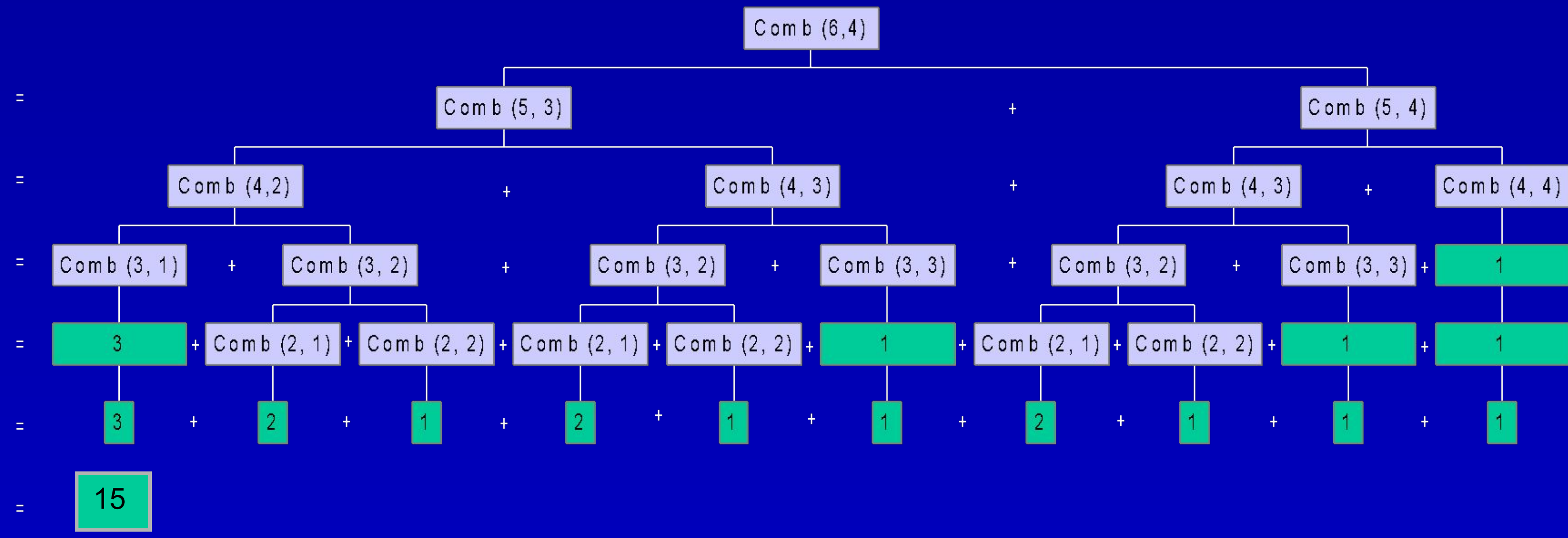
  Delete(location->next, item)

Recursive DeleteItem (sorted list) (cont.)

```
template <class ItemType>
void Delete(NodeType<ItemType>* &location, ItemType item)
{
 if(item == location->info)) {

   NodeType<ItemType>* tempPtr = location;
   location = location->next;
   delete tempPtr;
 }
 else
   Delete(location->next, item);
}
```

```
template <class ItemType>
void SortedType<ItemType>::DeleteItem(ItemType item)
{
 Delete(listData, item);
}
```

Recursion can be very inefficient is some cases



Deciding whether to use a recursive solution

- When the depth of recursive calls is relatively "shallow"
- The recursive version does about the same amount of work as the nonrecursive version
- The recursive version is shorter and simpler than the nonrecursive solution

Exercises

- 7-12, 15