CSCI 104
Simple Recursion

CSCI 104 Teaching Team

Spring 2025

Revised: 01/28/2025

Recursion in CS 104

- Problem in which the solution can be expressed in terms of itself (usually a smaller instance/input of the same problem) ***and a base/terminating case***
- Recursion is a **key concept** in this course
  - But it rarely comes easily to students. You must work at it!
- Many problems that would be VERY difficult to solve without recursion (i.e. only loops) have extremely ***elegant solutions*** to problems
  - Learn to look for those elegant solutions
  - In this class, assume the recursive approach has an elegant/simple solution
  - If you find yourself writing a large, complex recursive solution, assume you are doing something you should not!
    - Stop and reconsider how it should be done

Recursive Definition of C++

Rule	Expansion
expr	constant | variable_id | function_call | assign_statement | '(' **expr** ')' | **expr** binary_op **expr** | unary_op **expr**
assign_statement	variable_id '=' expr
expr_statement	';' | expr ';'

Example:

```
5 * (9 + max);
expr * ( expr + expr );
expr * ( expr );
expr * expr;
expr;
expr_statement
```

Example:

```
x + 9 = 5;
expr + expr = expr;
expr = expr;

NO SUBSTITUTION
Compile Error!
```

C++ Grammar

Rule	Expansion
statement	expr_statement | if (expr) statement | while (expr) statement …
statement_list	statement | statement_list statement | '{' statement_list '}'

Example:

```
while(x > 0) { doit(); x = x-2; }
while(expr) { expr; assign_statement; }
while(expr) { expr; expr; }
while(expr) { expr_statement expr_statement }
while(expr) { statement statement }
while(expr) { statement_list statement }
while(expr) { statement_list }
while(expr) statement_list
statement_list
```

Example:

```
while(x > 0)
  x--;
  x = x + 5;
```

```
while(expr)
  statement
  statement
```

```
statement
statement
```

FORMULATING PROBLEMS USING RECURSION

Steps to Formulating Recursive Solutions

1. Solve a few instances of the problem to discover the recursive structure
2. Identify how the problem can be decomposed into smaller problems of the same form
   - Does solving the problem on an input of smaller value or size help formulate the solution to the larger
3. Identify the base case
   - An input for which the answer is trivial
4. Assume the recursive call for the smaller problem "magically" computes the correct solution(s) to those problem(s) and **identify how to combine those solution(s)** from the smaller problem(s) into the solution for the larger problem

Recursive Formulation Ex

- Suppose we only have 1 and 3 dollar bills
- You need to pay off an n dollar debt but must only pay 1 bill per day.
- How many ways, f(n), are there to use 1 and 3 dollar bills to pay n dollars?
  - Follow the suggested steps:
    - Write out solutions to some problems
    - Try to find how solutions to smaller problems can be combined to solve the solution to the harder problem
    - What is the "1 thing" we can handle and then use recursion for the remaining problem?

```
f(1)=__:
f(2)=__:
f(3)=__:
f(4)=__:

f(5)=__:


f(6)=__:


f(7)=__:
```

- Now formulate the recursive answer
  - Base case: _____
  - Recursive case: _____

Recursive Formulation Ex

- Suppose we only have 1 and 3 dollar bills
- You need to pay off an n dollar debt but must only pay 1 bill per day.
- How many ways, f(n), are there to use 1 and 3 dollar bills to pay n dollars?
- Now formulate the recursive answer
  - Base case: _____
  - Recursive case: _____

```
int f(int n)
{
  if(n <= 2) return __;
  else if(n == 3) return __;
  else {
     return _________________
  }
}
```

Simple vs. Multiple Recursion

- "**Simple**" recursion refers to functions that contain just **ONE recursive call**
  - Can be head or tail recursion (explained soon)
  - **Can easily be replaced by a loop**

```
int fact(int n) {
 if(n == 1) return 1;
 return n * fact(n-1);
}
```

Simple Recursion
(1 recursive call)

- The power of recursion usually comes when the function makes **2 OR MORE recursive calls** (aka "**multiple recursion**")
  - Elegant recursive solutions that would be **MUCH harder to implement iteratively** (usually need a separate stack data structure)

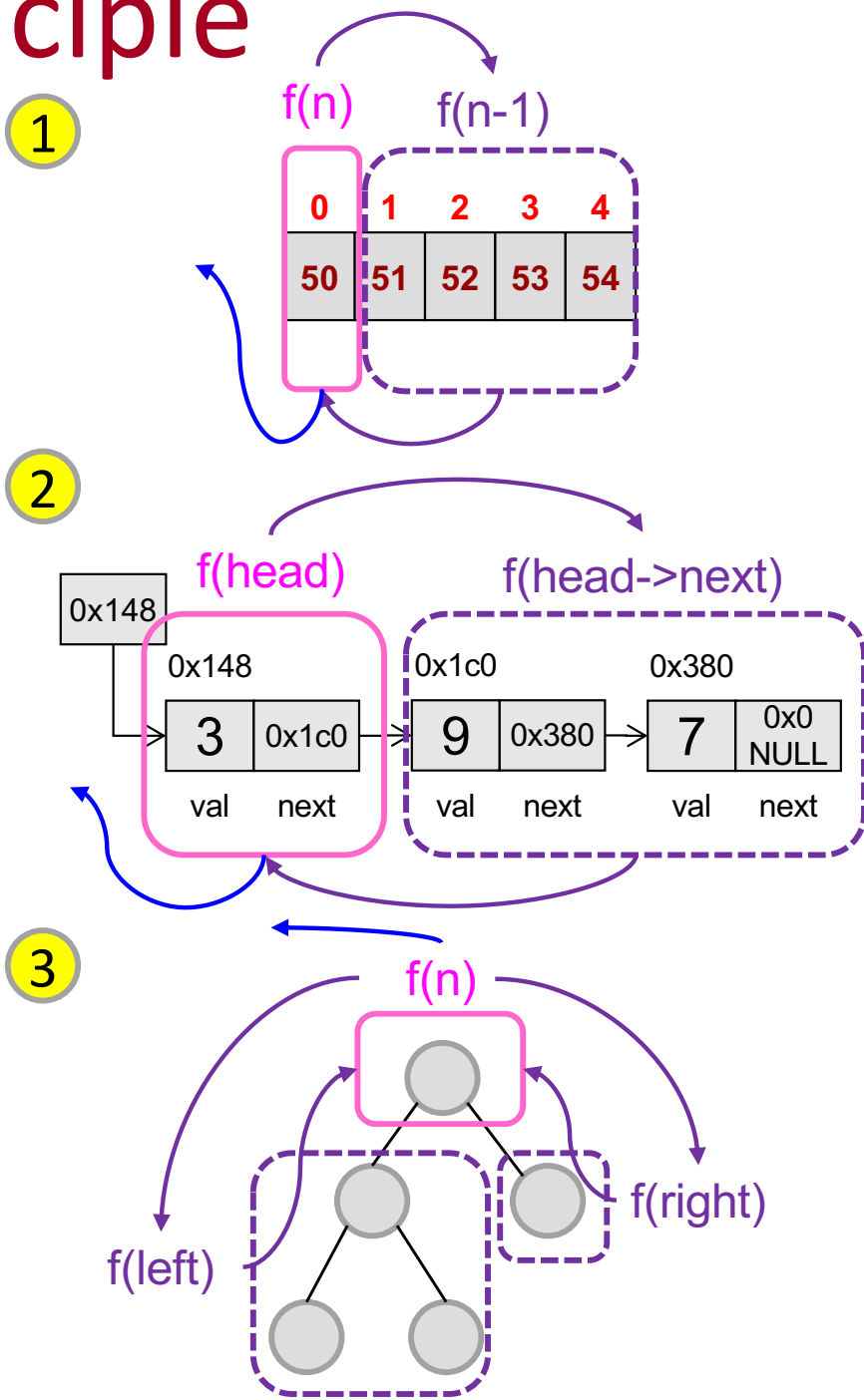
```
int f(int n)
{
  if(n <= 2) return 1;
  else if(n == 3) return 2;
  else {
     return f(n-1)+f(n-3);
  }
}
```

Multiple Recursion
(2 or more recursive calls)

- We'll focus on simple recursion first and later on multiple recursion)

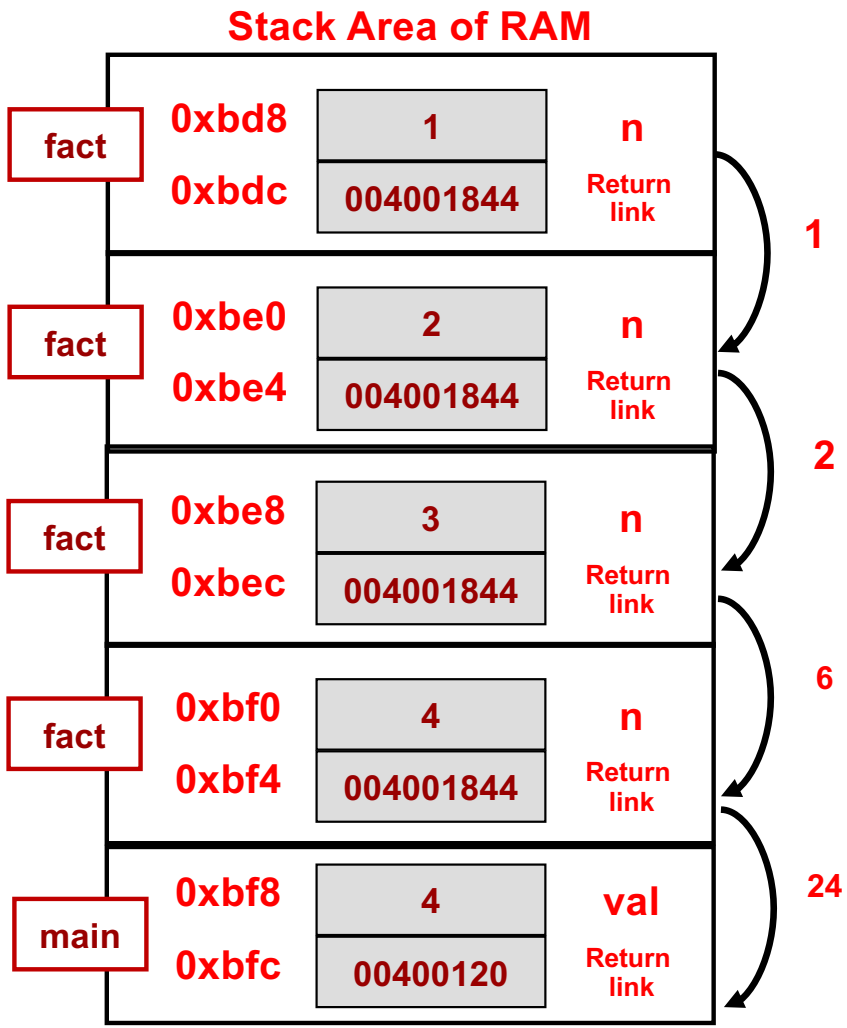
Guiding Principle

- A useful principle when trying to develop recursive solutions is that the recursive code should handle only 1 element, which might be:
  1. An element in an array
  2. A node a linked list
  3. A node in a tree
  4. One choice in a sequence of choices
- Then use recursion to handle the remaining elements
- And finally combine the solution(s) from the recursive call(s) with the one element being handled



Recursion & the Stack

- Must return back through the each call

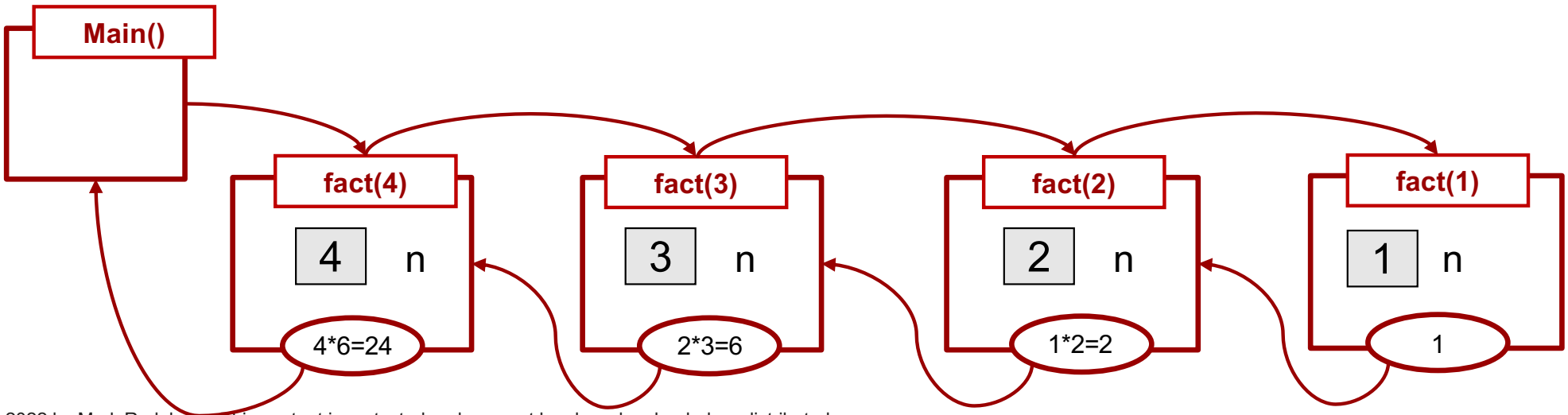



```
int fact(int n)
{
  if(n == 1){
     // base case
     return 1;
  }
  else {
     // recursive case
     return n * fact(n-1);

  }
}


int main()

{

  int val = 4;

  cout << fact(val) << endl;

}
```

Recursive Analysis Tip: Box Diagrams

- To analyze recursive functions draw a box diagram which is…
  - A simplified view of each function instance on the stack
- One box per function
  - Show arguments, pertinent local variables, and return values

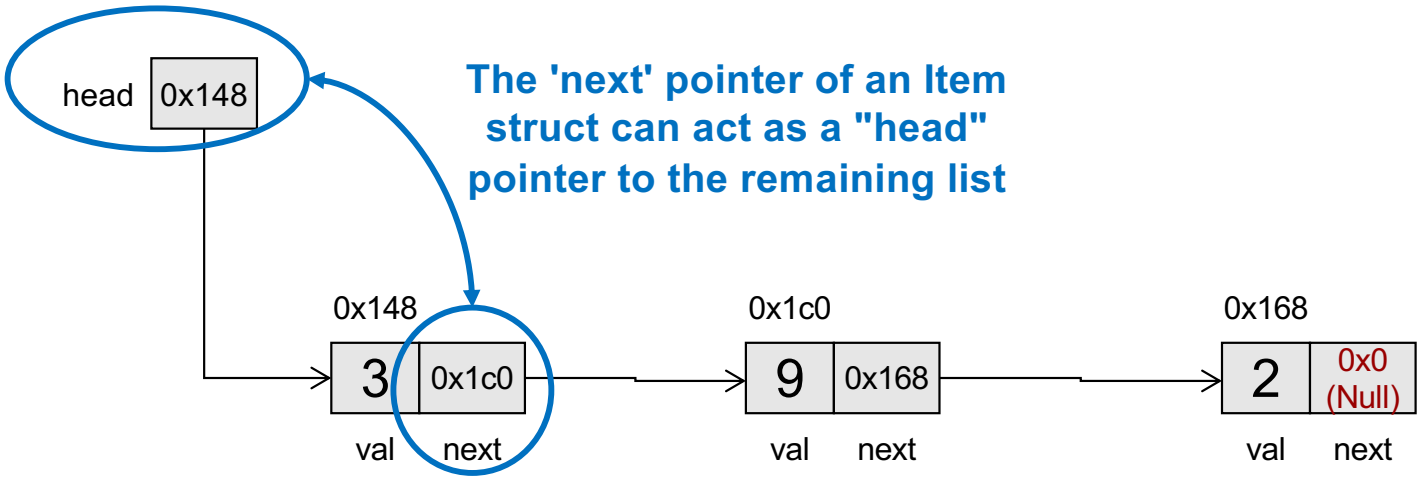
```
int fact(int n)
{
  if(n == 1){
     return 1;
  }
  else {
     return n * fact(n-1);

  }
}
```



RECURSION & LINKED LISTS

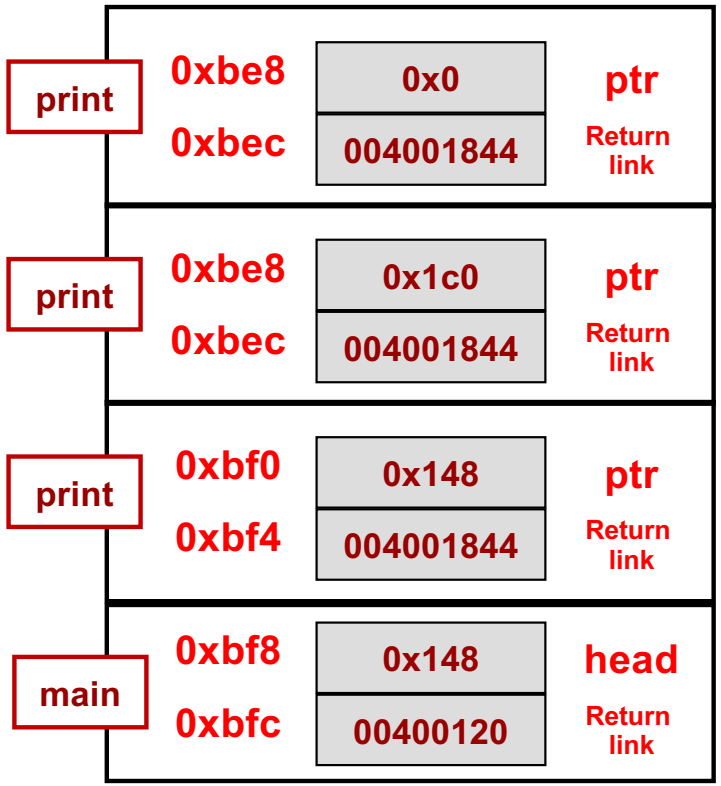
Recursion and Linked Lists

- Notice that one Item's next pointer looks like a head pointer to the remainder of the linked list
  - If we have a function that processes a linked list by receiving the head pointer as a parameter we can recursively call that function by passing our 'next' pointer as the 'head'

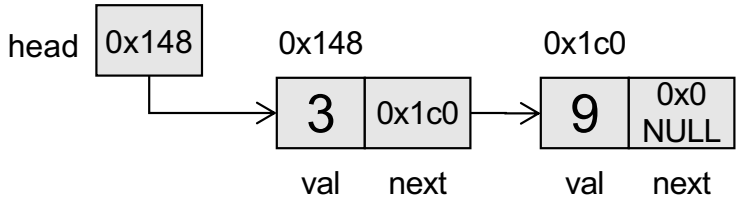


Recursive Operations on Linked List

- Many linked list operations can be recursively defined
- Can we make a recursive iteration function to print items?
  - Recursive case: Print one item then the problem becomes to print the n-1 other items.
    - Notice that any 'next' pointer can be though of as a 'head' pointer to the remaining sublist
  - Base case: Empty list (i.e. Null pointer)
- How could you print values in reverse order?




```
void print(Item* ptr)
{
  if(ptr == NULL) return;
  else {
    cout << ptr->val << endl;
    print(ptr->next);
  }
}
int main()
{ Item* head;
  ...
  print(head);
}
```



Recursive Operations on Linked List

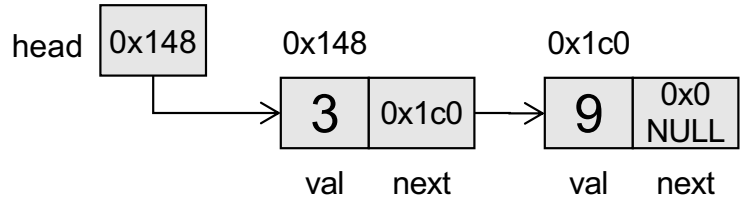
- How could you print values in reverse order?

Print in Order

```
void print(Item* ptr)
{
  if(ptr == NULL) return;
  else {
    cout << ptr->val << endl;
    print(ptr->next);
  }
}
int main()
{ Item* head;
  ...
  print(head);
}
```

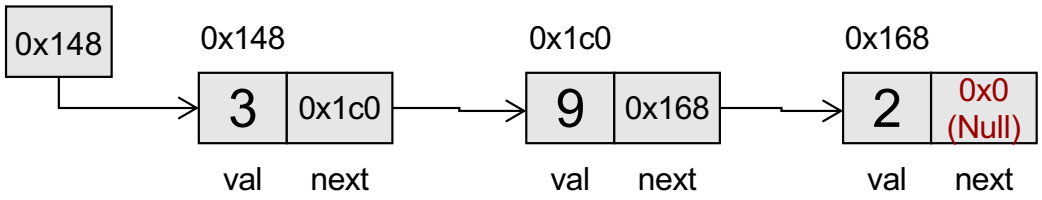
Print in Reverse Order

```
void print(Item* ptr)
{
  if(ptr == NULL) return;
  else {


  }
}
int main()
{ Item* head;
  ...
  print(head);
}
```

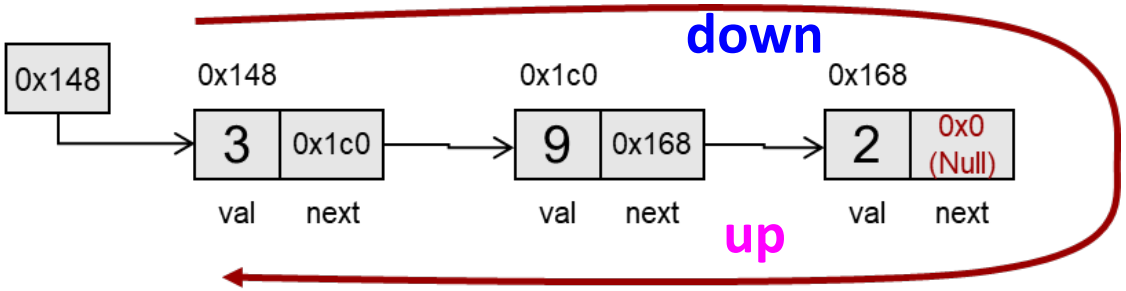


Summing a Linked List

- Given a linked list of integers, write a recursive routine to find the sum of all values.



Head vs. Tail Recursion

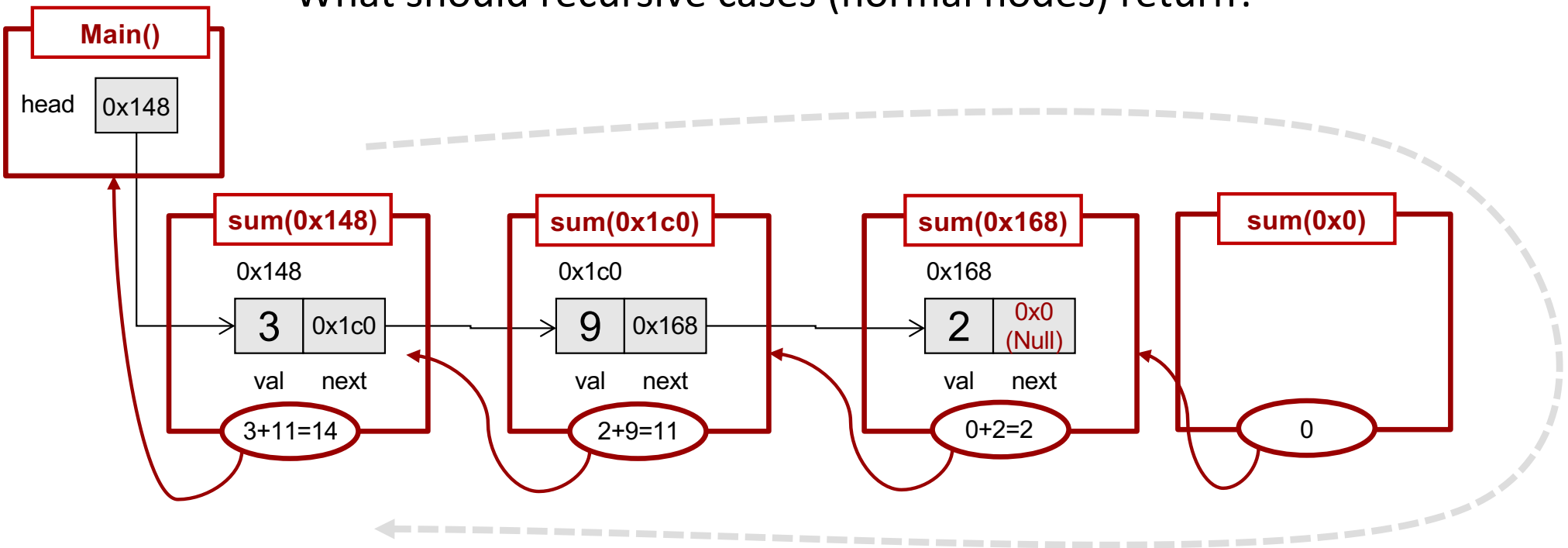


- Using recursion provides a choice for when to do our work (processing)
  - **Before recursing** (on our way ***down*** the recursive call sequence) = **TAIL recursion**
    - Process one element first and then recurse by passing the results to the next
    - Once we hit the base case we return the answer back up through the call sequence
  - **After recursing** (on our way back ***up*** the recursive call sequence) = **HEAD recursion**
    - Recurse first and then do our processing on the way back up (in reverse order) by combining our one element with the solution returned by the recursive call

Tail Recursion: Process before recursing (i.e. on the way down)
Head Recursion: Process after recursing (i.e. on the way back up)

Head Recursion

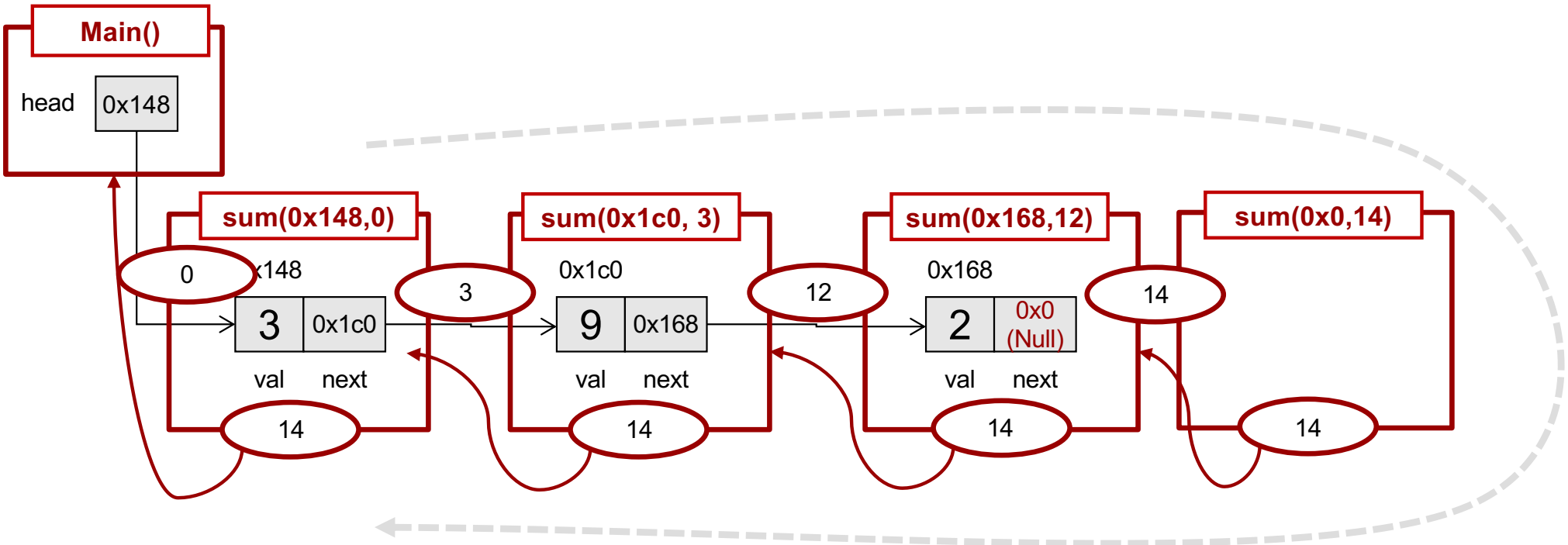
- Recurse first (to the end of the chain when our argument is NULL) and then start summing as we return from each recursion (on the way back **up**)
  - What should the base case return
  - What should recursive cases (normal nodes) return?



What would the prototype of this recursive function be?

Tail Recursion

- Produce sum as you walk down the list then just return the final answer back up the list



What would the prototype of this recursive function be?

ANALYZING RUNTIME OF RECURSION

What about Recursion

- Assume N items in the linked list
- Setup and solve a recurrence relationship for the runtime
- T(n) = 1 + _____; T(0) = __
- Now unroll the recurrence relationship to find a series/summation that you can solve

```
void print(Item* head)
{
   if(head==NULL) return;
   else {
     cout << head->val << endl;
     print(head->next);
   }
}
```

What about Recursion

- Assume N items in the linked list
- Setup and solve a recurrence relationship for the runtime
- T(n) = 1 + T(n-1); T(0) = 1
- Now unroll the recurrence relationship to find a series/summation that you can solve
- = 1 + 1 + T(n-2)
- = 1 + 1 + 1 + T(n-3)
- = 1 + 1 + 1 + … + 1 + T(0)
- = n + 1 = θ(n)

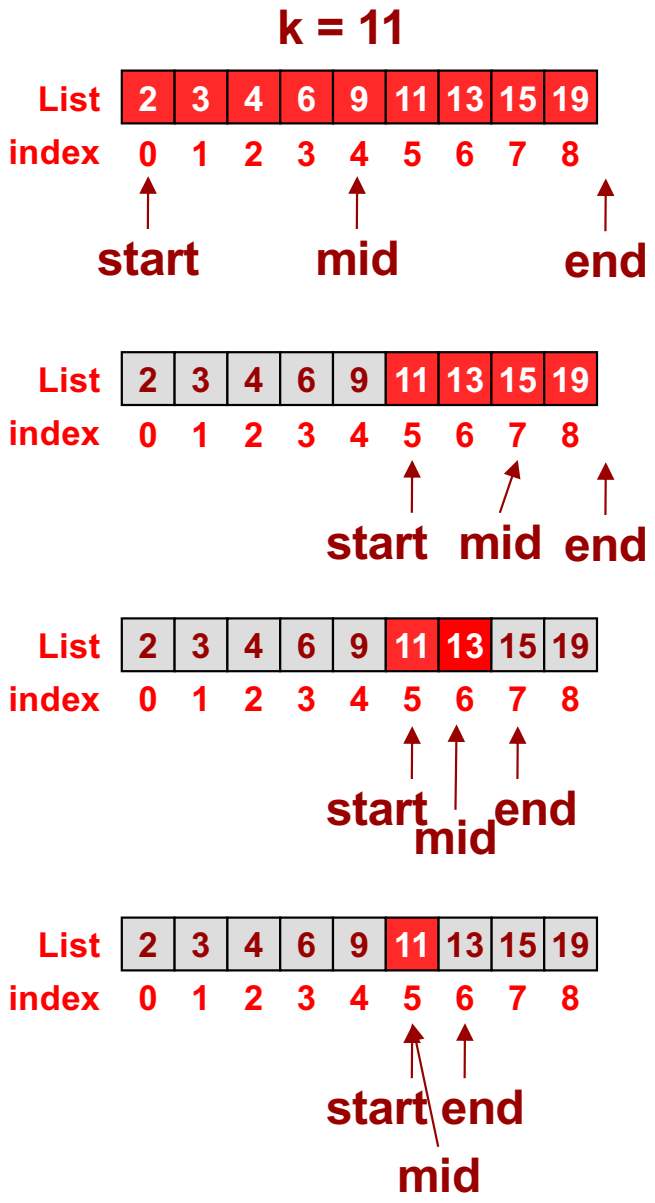
```
void print(Item* head)
{
   if(head==NULL) return;
   else {
     cout << head->val << endl;
     print(head->next);
   }
}
```

Solving Recurrence Relationships

1) Find the recurrence relationship
   - T(n) = T(n-1) + n and T(0) = 1 or
   - T(n) = T(n/2) + 1 and T(1) = 1
2) Unroll/unravel the relationship a few times to see the pattern emerging
3) Write an expression for T(n) for the k-th iteration/unrolling
4) Determine what value of k will cause you to hit the base case
5) Substitute the value you found for k from part 4) into the expression for T(n) you found in part 3)

Binary Search

- Search an ordered list (array) for a specific value, k, and return the location
- Binary Search
  - Compare k with middle element of list and if not equal, rule out ½ of the list and repeat on the other half
  - "Range" Implementations in most languages are [start, end)
  - Start is inclusive, end is non-inclusive (i.e. end will always point to 1 beyond true ending index to make arithmetic work out correctly)



Binary Search

- Assume n is (end-start)
  - # of items to be searched
- T(n) = _____
  and T(1) = $\theta(1)$

```
int bsearch(int data[],
            int start, int end,
            int target)
{
  if(start >= end)
    return -1;
  int mid = (start+end)/2;
  if(target == data[mid])
    return mid;
  else if(target < data[mid])
    return bsearch(data, start, mid,
                   target);
  else
    return bsearch(data, mid+1, end,
                   target);
}
```

Binary Search

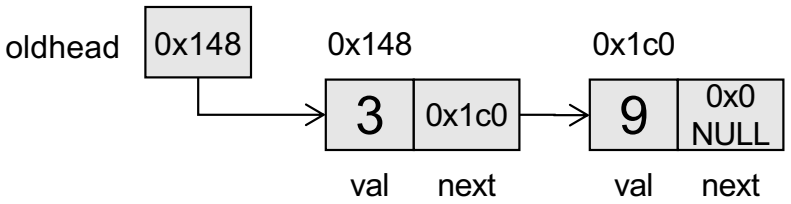
- Assume n is (end-start)
  - # of items to be searched
- T(n) = $\theta(1)$ + T(n/2) and T(1) = $\theta(1)$
- = 1 + T(n/2)
- = 1 + 1 + T(n/4)
- = 1 + 1 + 1 + T(n/8)
- = k + T($n/2^k$)
- Stop when $2^k = n$
  - Implies $\log_2(n)$ recursions
- $\theta(\log_2(n))$

```
int bsearch(int data[],
            int start, int end,
            int target)
{
  if(end >= start)
    return -1;
  int mid = (start+end)/2;
  if(target == data[mid])
    return mid;
  else if(target < data[mid])
    return bsearch(data, start, mid,
                   target);
  else
    return bsearch(data, mid+1, end,
                   target);
}
```

MORE RECURSIVE LINKED LIST EXERCISES

Recursive Linked List Copy

- How could you make a copy of a linked list using recursion



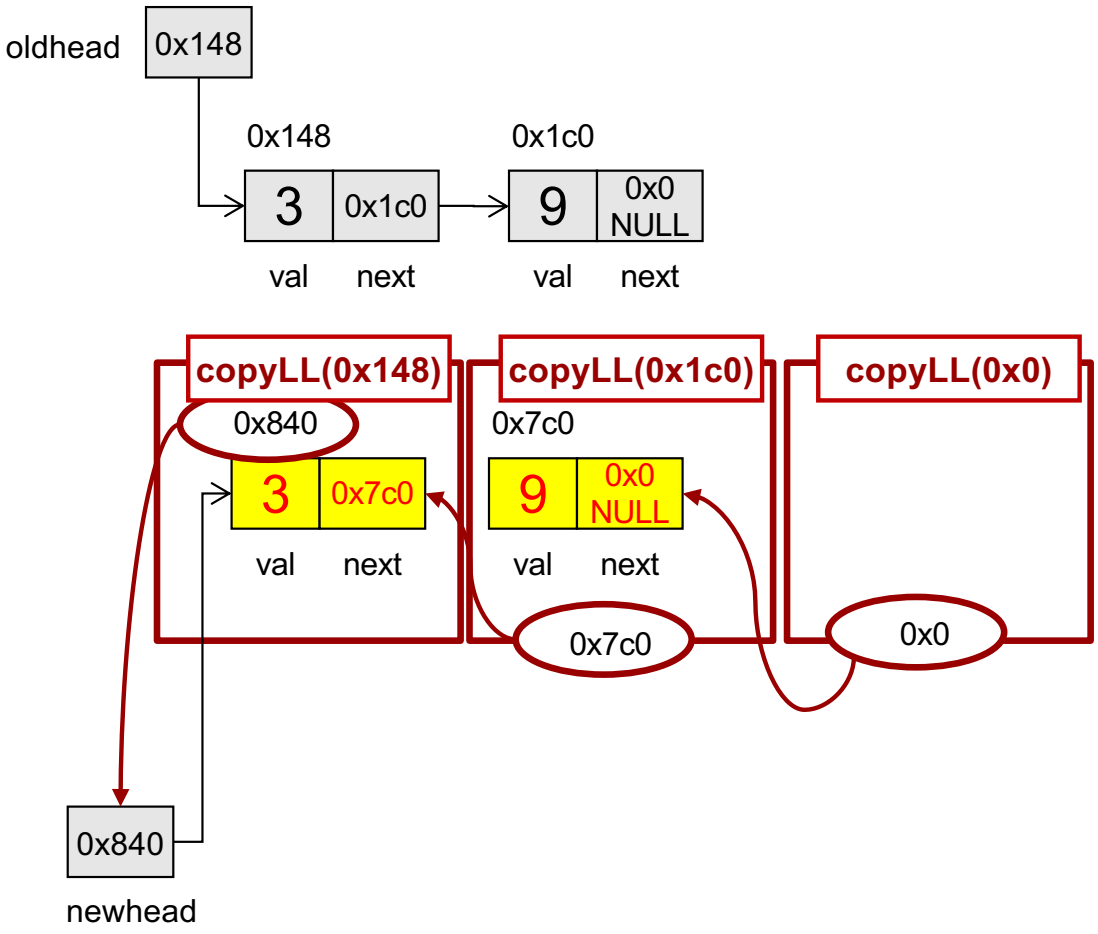
newhead ???

What work can you do as you recurse down the list and what needs to be done on the way back up?

```
struct Item {
 int val;
 Item* next;
 Item(int v, Item* n){
   val = v; next = n;
 }
};

Item* copyLL(Item* head)
{
  if(head == NULL) return NULL;
  else {




  }
}
int main()
{ Item* oldhead, *newhead;
  ...
  newhead = copyLL(oldhead);
}
```

Recursive Linked List Copy

- We always work inside out so start with the recursive call and then the 'new' and then the return.




```
struct Item {
 int val;
 Item* next;
 Item(int v, Item* n){
   val = v; next = n;
 }
};

Item* copyLL(Item* head)
{
  if(head == NULL) return NULL;
  else {      2
    return new Item(head->val,
      3              1 copyLL(head->next));
  }
}
int main()
{ Item* oldhead, *newhead;
  ...
  newhead = copyLL(oldhead);
}
```

Recursive DL Append

- Add a new item to end of doubly linked list given head and new value (and no tail pointer)

```
struct DLItem {
  int val; DLItem* prev; DLItem* next;
  DLItem(int v, DLItem* p, DLItem* n);
};

void append(DLItem*& head, int v)
{
  if (__________________){
    head = new DLItem(v, NULL, NULL);
  } else if (__________________){
    head->next = new DLItem(v, ______, NULL);
  } else {
    append(head->next, v);
  }
}
```

Practice Exercises

- Exercises on course web page: llsum_head, llsum_tail, llmax_head
- Recursively reverse a singly linked list: Given a pointer to the head, return pointer to the new head without allocating any more memory,
  - `Item* reverse(Item * head);`
- Recursively remove an item with a specified value from a singly linked list : Given a pointer to the head and the value to remove,
  - `void remove(Item *& head, int valToRemove);`

HELPER FUNCTIONS AND RECURSIVE PARAMETER PASSING

Exercise – Helper Function

- Head recursion
- Tail recursion

```
int data[4] = {8, 9, 7, 6};
```

```
// The client only wants this
int maxVal(int* data, int len);

// And with head recursion we can do our job
// with that same signature
```

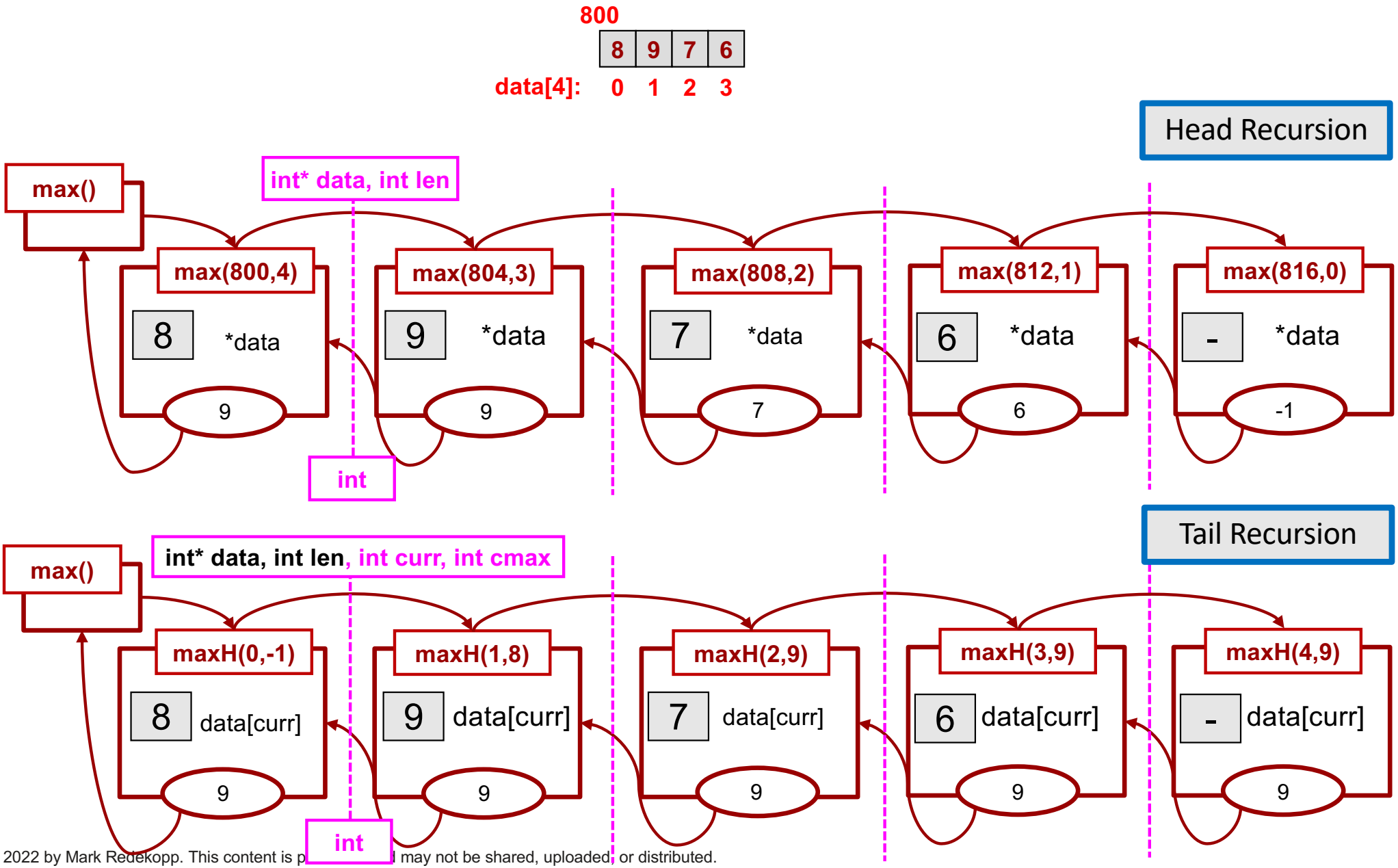
```
int maxVal(int* data, int len)
{
  if(0 == len) return 0;
  else {
    int prevmax = maxVal(data+1, len-1);
    return std::max(*data, prevmax);
  }
}
```

We don't need 'curr' here and can iterate just by incrementing 'data' and decrementing 'len'

```
// The client only wants this
int maxVal(int* data, int len);

// But to do the job we need this
int maxHelp(int* data, int len, int curr, int cmax);
```

```
int maxVal(int* data, int len)
{ int mymax = 0;
  mymax = maxHelp(data, len, 0, mymax);
  return mymax;
}

int maxHelp(int* data, int len, int curr, int cmax)
{
  if(curr == len) return cmax;
  else {
    if(data[curr] > cmax)
      cmax = data[curr];
    return maxHelp(data, len, curr+1, cmax);
}
```

Head vs. Tail Recursion Box Diagram



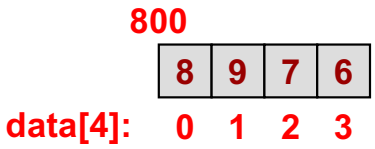
Another Tail Recursion Impl.

- Suppose we couldn't use the return value? We can use a reference parameter

```
// The client only wants this
int maxVal(int* data, int len);

// But to do the job we need this
void maxHelp(int* data, int len, int curr, int& mx);
```

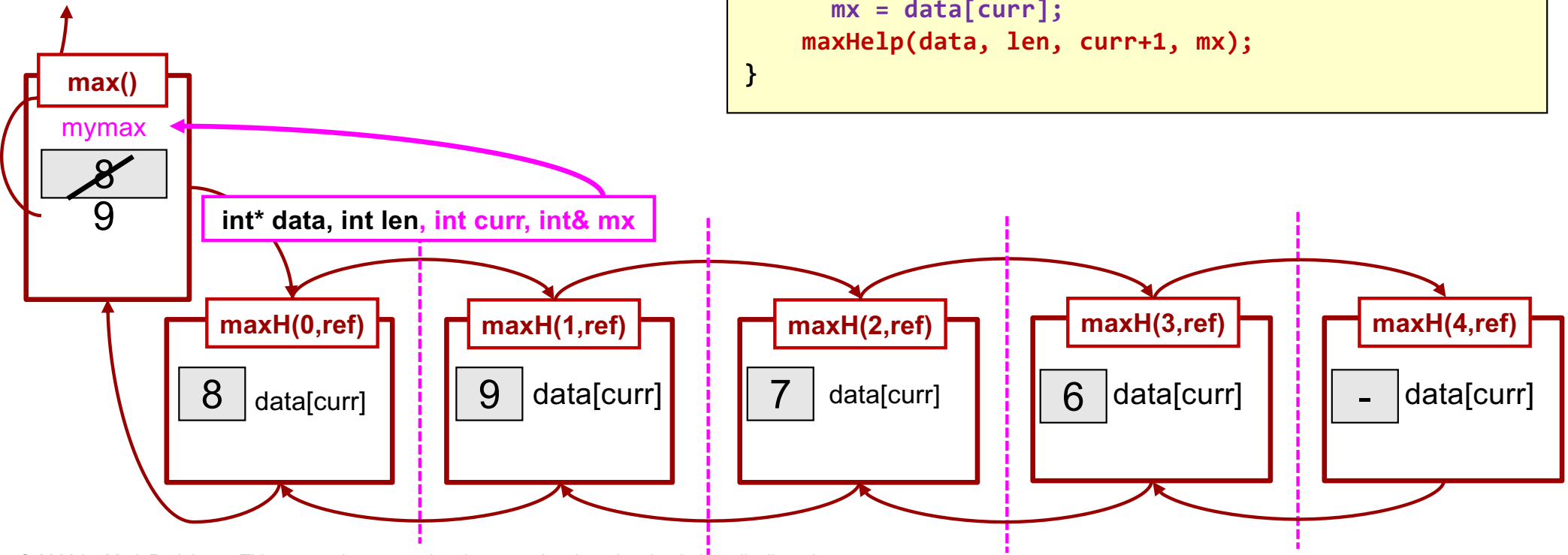
```
int maxVal(int* data, int len)
{ int mymax = 0;
  maxHelp(data, len, 0, mymax);
  return mymax;
}

void maxHelp(int* data, int len, int curr, int& mx)
{
  if(curr == len) return;
  else {
    if(data[curr] > mx)
      mx = data[curr];
    maxHelp(data, len, curr+1, mx);
}
```

Tail Recursion w/ Reference Parameter




```
int maxVal(int* data, int len)
{ int mymax = 0;
  maxHelp(data, len, 0, mymax);
  return mymax;
}

void maxHelp(int* data, int len, int curr, int& mx)
{
  if(curr == len) return;
  else {
    if(data[curr] > mx)
      mx = data[curr];
    maxHelp(data, len, curr+1, mx);
}
```

Tail Recursion



Recursive Do's and Don'ts

- When using recursion for linked lists (and later trees) some generally useful guidelines are:
  - Use the check for head==NULL (when possible) rather than head->next == NULL (in case you get an empty list)
    - A more general guideline is make sure you can prove the pointer you are about to dereference (with * or ->) is not NULL otherwise it might segfault
    - Another implication of this is we will actually recurse off the end of the list (call our function with a NULL pointer) but the immediate base case check will catch this and return
  - Each function should only access/modify the data of one node/item (usually the one pointed to by the argument)
  - If you need multiple "return" values you can use reference parameters

SOLUTIONS

Recursive Formulation Ex

- Suppose we only have 1 and 3 dollar bills
- You need to pay off an n dollar debt but must only pay 1 bill per day.
- How many ways, f(n), are there to use 1 and 3 dollar bills to pay n dollars?
  - Follow the suggested steps:
    - Write out solutions to some problems
    - Try to find how solutions to smaller problems can be combined to solve the solution to the harder problem
    - What is the "1 thing" we can handle and then use recursion for the remaining problem? What bill do we pay for 1 day?
- Now formulate the recursive answer
  - Base case: f(1) = 1, f(2) = 1, f(3) = 2;
  - Recursive case: f(n) = f(n-1) + f(n-3)

```
f(1)=1: 1

f(2)=1: 1+1

f(3)=2: 1+1+1, 3

f(4)=3:
   1+1+1+1, 1+3, 3+1

f(5)=4:
   1+1+1+1+1, 1+1+3,
   1+3+1, 3+1+1

f(6)=6:

   1+1+1+1+1+1, 1+1+1+3,
   1+1+3+1, 1+3+1+1,
   3+1+1+1, 3+3

f(7)=9:
```

Recursive Formulation Ex

- Suppose we only have 1 and 3 dollar bills
- You need to pay off an n dollar debt but must only pay 1 bill per day.
- How many ways, f(n), are there to use 1 and 3 dollar bills to pay n dollars?
- Now formulate the recursive answer
  - Base case: f(1) = 1, f(2) = 1, f(3) = 2;
  - Recursive case: f(n) = f(n-1) + f(n-3)

```
int f(int n)
{
  if(n <= 2) return 1;
  else if(n == 3) return 2;
  else {
     return f(n-1)+f(n-3);
  }
}
```

Recursive DL Append

- Add a new item to end of doubly linked list given head and new value (and no tail pointer)

```
struct DLItem {
  int val; DLItem* prev; DLItem* next;
  DLItem(int v, DLItem* p, DLItem* n);
};

void append(DLItem*& head, int v)
{
  if (head == NULL){
    head = new DLItem(v, NULL, NULL);
  } else if (head->next == NULL){
    head->next = new DLItem(v, head, NULL);
  } else {
    append(head->next, v);
  }
}
```

BACKUP

What about Recursion

- Assume N items in the linked list
- Setup and solve a recurrence relationship for the runtime
- T(n) = 1 + _____; T(0) = __
- Now unroll the recurrence relationship to find a series/summation that you can solve

```
void print(Item* head)
{
   if(head==NULL) return;
   else {
     cout << head->val << endl;
     print(head->next);
   }
}
```

What about Recursion

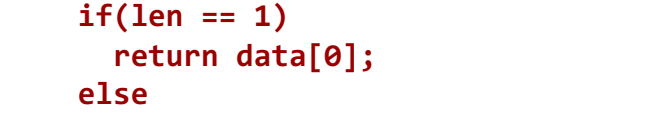
- Assume N items in the linked list
- Setup and solve a recurrence relationship for the runtime
- T(n) = 1 + T(n-1); T(0) = 1
- Now unroll the recurrence relationship to find a series/summation that you can solve
- = 1 + 1 + T(n-2)
- = 1 + 1 + 1 + T(n-3)
- = 1 + 1 + 1 + … + 1 + T(0)
- = n + 1 = θ(n)

```
void print(Item* head)
{
   if(head==NULL) return;
   else {
     cout << head->val << endl;
     print(head->next);
   }
}
```

Summing an Array: Recursive Timeline

```
int main(){
  int data[4] = {8, 6, 7, 9};
  int size=4;
  cout << rsum_it(data, size);
  ...
}
```

```
int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum = rsum_it(data, len-1);
    return sum + data[len-1];
}
```



Time

rsum_it(data,4)
int sum=
rsum_it(data,4-1)

len = 4

int sum = 21
return 21+data[3];
30

rsum_it(data,3)
int sum=
rsum_it(data,3-1)

len = 3

int sum = 14
return 14+data[2];
21

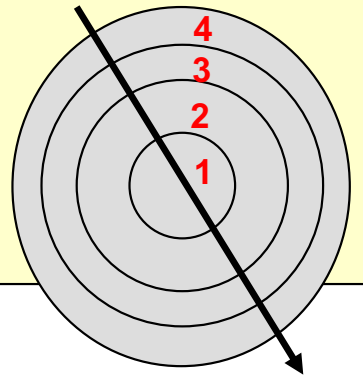
len = 2

rsum_it(data,2)
int sum=
rsum_it(data,2-1)

int sum = 8
return 8+data[1];
14

len = 1

rsum_it(data,1)
return data[0];
8



Each instance of rsum_it has its own len argument and sum variable

Every instance of a function has its own copy of local variables

Head vs. Tail Recursion

```
int data[4] = {8, 9, 7, 6};
```

- Recall: When using recursion we have to come back through and return from each function call we made
- So we have a choice to do our work
  - On our way ***down*** the recursive call sequence
    - Process our element first and then recurse by passing the results to the next
    - This is known as **TAIL recursion** since we recurse AFTER processing
    - Once we hit the base case we return the answer back up through the call sequence
  - On our way back ***up*** the recursive call sequence
    - Recurse first and then do our processing on the way back up combining our work with the solution from the recursion
    - This is known as **HEAD recursion** since we recurse BEFORE processing

Tail Recursion: Process on the way down (process then recurse)
Head Recursion: Process on the way back up (recurse then process)