Pointers cont'd

Pointers can help save memory!

- In astronomy, we sometimes work with very large images
- Having multiple copies of those images “in memory” can crash your machine
  - good idea to avoid duplication where possible

memory

- There is a way to find out how much memory IDL is using
- It's the `memory` function!
- `print,memory()` returns a 4-element array
  - the first element is the memory in use in *bytes*

memory checking

```
IDL> print,memory()
      1803740           3078            946      1807690
IDL> print,(memory())[0]/1024
         1761
IDL> print,(memory())[0]/1024L^2
            1
```

- Memory given in # of bytes
- Divide by 1024 bytes/kilobyte
- Or 1024^2 bytes/megabyte

Pointers & Memory

- Copying pointers is “cheap”, copying data is “expensive”
- not going to quantify that though

Free Pointers

```
x = ptr_new(5)
y = x
*x = 2
print,*x,*y

ptr_free,x
print,*y
% Invalid pointer: Y.
% Execution halted at: $MAIN$
```

- `y` points to the same data as `x`, but now it's gone

```
x = ptr_new(5)
y = x
*x = 2
print,*x,*y
```

What will print?

A)5,2

B)2,5

C)2,2

D)5,5

E) None of the above

Arrays of Pointers

- Can declare an array of pointers!

```
pa = ptrarr(10)
```

- Defaults to all NULL pointers

Which line is a valid assignment operation?

```
pa = ptrarr(10)
```

```
*pa[1] = 5                ; A
pa[2] = 5                 ; B
*pa[3] = ptr_new(5)       ; C
pa[4] = ptr_new(5)        ; D
; None of the above       E
```

Arrays of Pointers

- However, there is a keyword to `ptrarr` that "allocates memory on the heap"

ALLOCATE_HEAP

Normally, PTRARR sets every element of the result to the null pointer. It you wish IDL to allocate heap variables for every element of the array instead, set the ALLOCATE_HEAP keyword. In this case, every element of the array will be initialized to point at an undefined heap variable.

- All elements will start as `!NULL`

Which line is *not* a valid assignment operation?

```
pa = ptrarr(10,/allocate)
```

```
*pa[1] = 5                    ; A
pa[2] = 5                     ; B
*pa[3] = ptr_new(5)           ; C
pa[4] = ptr_new(5)            ; D
; None of the above           E
```

What will be printed?

```
pa = ptrarr(10,/allocate)

*pa[1] = 5
print,*pa[1]
```

A) `<PtrHeapVar22>`

B) `% Unable to convert variable to type pointer.`

C) `5`

D) `1`

E) None of the above

What will be printed?

```
*pa[3] = ptr_new(5)
pa[4] = ptr_new(5)
print,*pa[3],*pa[4]
```

A) <PtrHeapVar22> 5

B) 5<PtrHeapVar22>

C) % Unable to convert variable to type pointer.

D) 5 5

E) None of the above

Coding Example

- Problem: We want to create a sorted array, and then add elements into it. We want to keep the array sorted at all times, so we need to add each new element in the right place.

Add a new element in order

- Could do this:

```
x = [2,3,1,5,6,9,12]
x = x[sort(x)]
print,x
x = [x,4]
x = x[sort(x)]
```

- But it's kind of a waste, since you can figure out pretty easily where that 1 new number (4) should go

Inserting Elements

```
IDL> x = [2,3,1,5,6,9,12]
IDL> ; sort(x) returns indices
IDL> x = x[sort(x)]
IDL> print,x
       1       2       3       5       6       9      12
IDL> print,fix(sort(x))
       0       1       2       3       4       5       6
IDL> ; insert 4 - should be between 3 and 5
IDL> x = [x[0:2],4,x[3:*]]
IDL> print,x
       1       2       3       4       5       6       9
```

Inserting Generally

- You can insert any number this way
- How do we automate it?

```
IDL> print,x,fix(sort(x))
       1       2       3       4       5       6       9      12
       0       1       2       3       4       5       6       7
```

```
new_number = 7
ii = 0
while new_number gt x[ii] do ii++
print,ii
```

```
IDL> print,x,fix(sort(x))
       1       2       3       4       5       6       9      12
       0       1       2       3       4       5       6       7
```

```
new_number = 7
ii = 0
while new_number gt x[ii] do ii++
print,ii
```

What's ii?

A) 0

B) 7

C) 6

D) 5

E) None of the above

What's the best (i.e., most efficient) way to put 7 in the right place?
(note that all of these *should* work)

```
x = [x,7] & x = x[sort(x)]                          ; A
x = [7,x] & x = x[sort(x)]                          ; B
x = [x[0:ii],7,[ii+1]]                              ; C
x = [x[where(x lt 7)],7,x[where(x gt 7)]] ; D
; None of the above                                  E
```

```
IDL> print,x,fix(sort(x))
       1       2       3       4       5       6       9      12
       0       1       2       3       4       5       6       7
```

We want to add a new number, 0. Will all of the same approaches work?

```
x = [x,0] & x = x[sort(x)]                          ; 1
x = [0,x] & x = x[sort(x)]                          ; 2
x = [x[0:ii-1],0,x[ii:*]]                           ; 3
x = [x[where(x lt 0)],0,x[where(x gt 0)]]; 4
```

A) Yes

B) No, 1 & 2 will fail

C) No, 3 & 4 will fail

D) No, just 3 will fail

E) No, just 4 will fail

Building up our algorithm...

- We need an “if” statement to check the first number:

```
if new_number lt x[0] then begin
    x = [new_number,x]
endif else begin
    ii = 0
    while new_number gt x[ii] do ii++
    x = [x[0:ii-1],new_number,x[ii:*]]
endelse
```

```
IDL> print,x,fix(sort(x))
       1       2       3       4       5       6       9      12
       0       1       2       3       4       5       6       7
```

What if `new_number` is greater than every element in `x`? (`new_number=13`)

```
ii = 0
while new_number gt x[ii] do ii++
x = [x[0:ii-1],new_number,x[ii:*]]
```

A) No problem, it will stick on the end

B) We'll get an infinite loop

C) We'll end up in the wrong place

D) We'll get an indexing error

E) None of the above

Building up our algorithm...

- Need another “if” statement

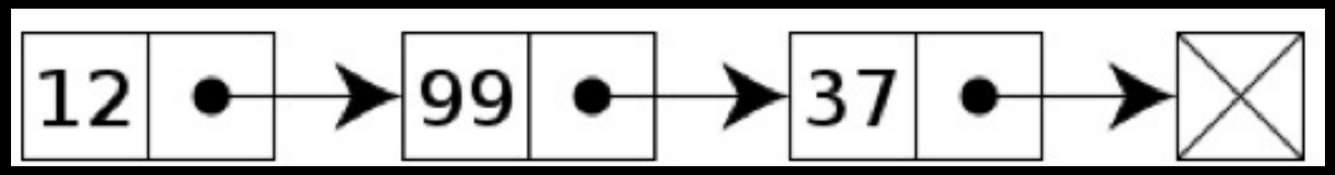
```
if new_number lt x[0] then begin
     x = [new_number,x]
endif else if new_number gt x[-1] then begin
     x = [x,new_number]
endif else begin
     ii = 0
     while new_number gt x[ii] do ii++
     x = [x[0:ii-1],new_number,x[ii:*]]
endelse
```

```
; insert a number into a sorted array
function insert,number,srtd_arr

    if number lt srtd_arr[0] then begin
        srtd_arr = [number,srtd_arr]
    endif else if number gt srtd_arr[-1] then begin
        srtd_arr = [srtd_arr,number]
    endif else begin
        ii = 0
        while number gt srtd_arr[ii] do ii++
        srtd_arr = [srtd_arr[0:ii-1],$
            number,srtd_arr[ii:*]]
    endelse

    return,srtd_arr
end ; insert
```

Linked Lists

- Linked Lists are another data type
- They’re very efficient for inserting new elements (adding new elements in order)
- As far as I know, they’re (almost) never used directly, but they serve as the conceptual basis for other things

Linked Lists



- Made of “nodes”
- Each node has two things:
  1. data
  2. a pointer to the next node

Linked Lists

- Nodes aren't arrays (they have two different data types)
- They COULD be lists
- But it makes more sense to make each node a struct

```
node = {Node, data: 0, next: ptr_new()}
```

LL notation

- Here's a linked list:
  `[1|next] -> [2|next] -> [3|next] -> !null`
- Each `[ | ]` is a node.
  - The left side is data
  - The right side is a pointer to the next node

Make a linked list

- Let’s start with a linked list with 3 elements: 1, 4, 2
- We’re going to keep it sorted
- We’ll add each new value in turn

First Node

```
node1 = {Node, data: 1, next: ptr_new()}
```

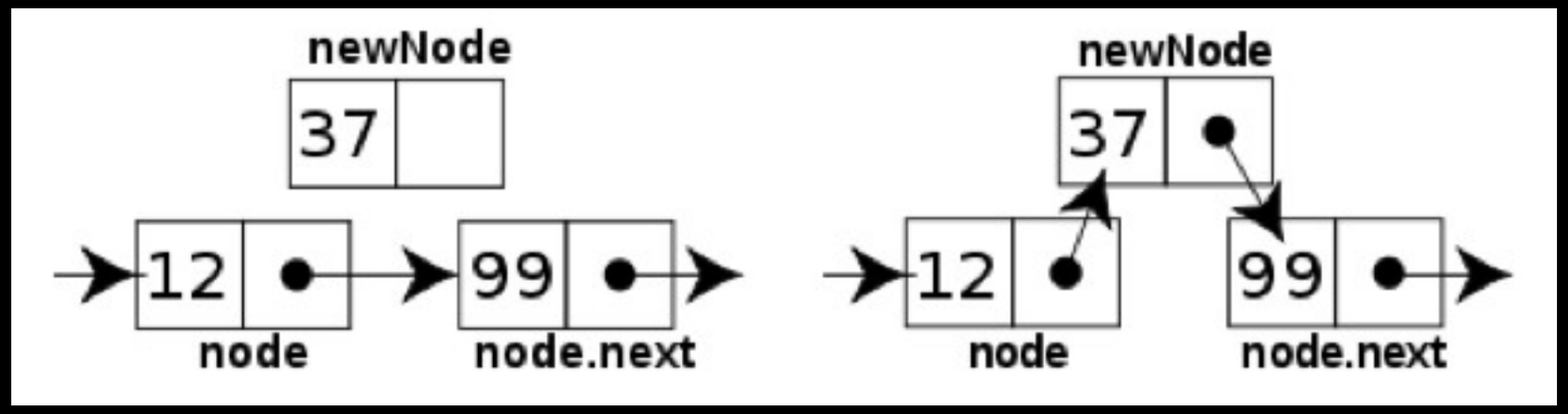
- [1|!null]
- We'll add a second node, with `data=4`, after the first

Second Node

```
node1 = {Node, data: 1, next: ptr_new()}
node1.next = ptr_new({Node, data:4, next:ptr_new()})
```

- `[1|node2] -> [4|!null]`
- For the next node, we'll loop through all nodes until we find the right place
- How do we know if we've hit the end?
  - if `node.next eq !null`, there's no next node

Linked List Insertion



- Actually, really quite difficult to do.
- You need to loop through the linked list, while keeping track of both the previous and next element of the linked list

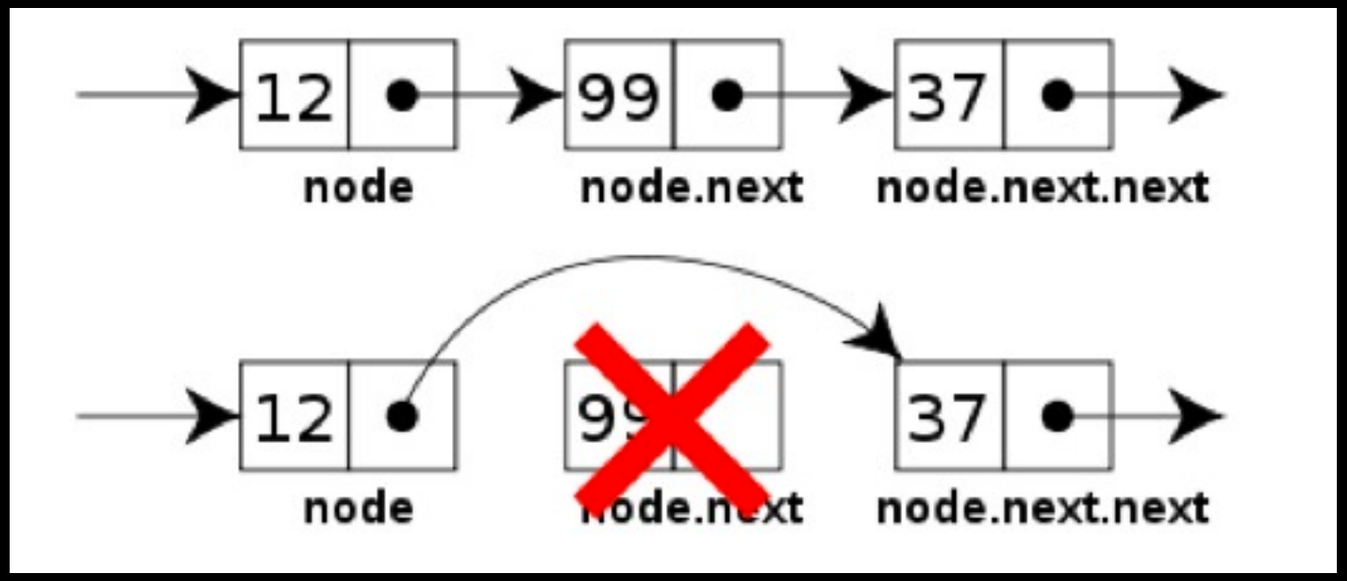
Linked List Insertion

- If you have some new data "`number`" you want to insert into a linked list, you loop through the list until you have the previous & next node in hand...

- Then, make a new node, and stick it in

```
new_node = ptr_new({Node,$
    data: number,$
    next: node_ptr})
(*prev).next = new_node
```

Linked List Removal

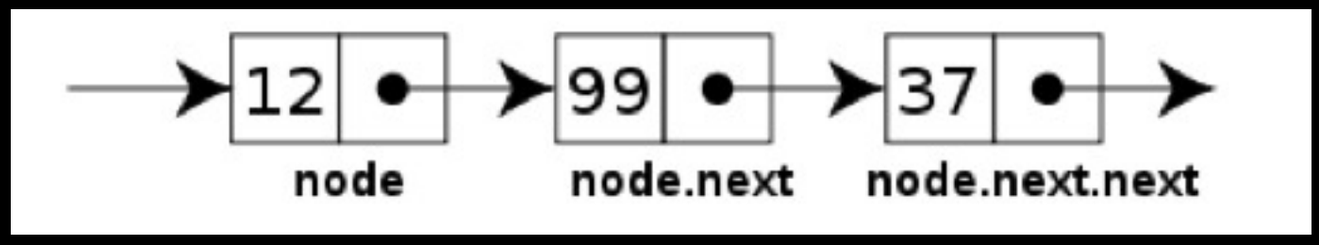


- What if you want to get rid of an element in a linked list?
- You need to “know about” 3 different nodes! You have to have the first point to the third

Dereferefererencing

- Not necessarily too hard to have 3 nodes in hand...
- `(*(*(*node).next).next).data`

is pretty darned hard to read, though



Third Node

```
node1 = {Node,1,ptr_new()}
node1.next = ptr_new({Node,4,ptr_new()})
node2 = *node1.next
node3 = ptr_new({Node,2,ptr_new(node2)})
node1.next = node3
```

- [1|node3] -> [2|node2] -> [4|!null]

Did it really work?

- `[1|node3] -> [2|node2] -> [4|!null]`

```
IDL> print,node1
{       1<PtrHeapVar10>}
IDL> print,*node1.next
{       2<PtrHeapVar9>}
IDL> print,*(*node1.next).next
{       4<NullPointer>}
```

Accessing Linked Lists

- How do you store a linked list?
  - For normal variables, you'd do something like:

```
x = fltarr(5)
```

- For linked lists, you store a single pointer to the beginning of the list

The "Head" of a LL

```
[1|next] -> [2|next] -> [3|next] -> !null
```

- To store this LL, we need to point to its start

```
head = ptr_new()
```

- `head` is not a node, but `*head` is

```
head->[1|next]->[2|next]->[3|next]->!null
```

Building a LL from scratch

```
pro node__DEFINE
    node = {Node, data: 0, next: ptr_new()}
end


IDL> first_node = {Node,data:1,next:ptr_new()}
IDL> head = ptr_new(first_node)
IDL> print,head
<PtrHeapVar5>
IDL> print,*head
{       1<NullPointer>}
IDL> print,first_node
{       1<NullPointer>}
```

```
IDL> first_node = {Node,data:1,next:ptr_new()}
print,first_node.next
```

What will print?

```
A. % Unable to dereference NULL pointer:
   <POINTER (<NullPointer>)>.
B. {      1<NullPointer>}
C. <NullPointer>
D. <PtrHeapVar6>
E. None of the above
```

What will this do?

```
first_node.next = ptr_new({Node,7,ptr_new()})
```

A) Crash

B) Extend the linked list with a new `Node`

C) Make a new node, then lose it

D) Make a new pointer to nothing

E) None of the above

Our linked list now...

```
head->[1|next]->[7|next]->!null
```

- Note that `*head` and `first_node` are both the same thing
  - `head` is a pointer to a `node`
  - `first_node` is a variable of type `node`

Looping through Nodes

```
new_data = 2
while (node.data lt new_data) and $
       (node.next ne !null) do $
    node = *(node.next)
```

- Now "`node`" is our loop variable
- The while loop will quit if either:

A. `new_data` is greater than or equal to `node.data`

B. we've reached the end of the linked list

Traversing a Linked List

```
head->[1|next]->[7|next]->!null
```

```
node = *head
```

```
head->[1|next]->[7|next]->!null
```

node is this LL item

```
node = *(node.next)
```

```
head->[1|next]->[7|next]->!null
```

now node is this LL item

Inserting [2|...]

```
head->[1|next]->[7|next]->!null
```

```
node = *head
```

```
head->[1|next]->[7|next]->!null
```

node is this LL item

`node.data = 1`, so `node.data lt 2`

```
node = *(node.next)
```

```
head->[1|next]->[7|next]->!null
```

now node is this LL item

`node.data = 7`, so `node.data lt 2` is false!

Inserting [2|...]

`head->[1|next]->[7|next]->!null`

`node` is this LL item

`node.data = 7`, so `node.data lt 2` is false!

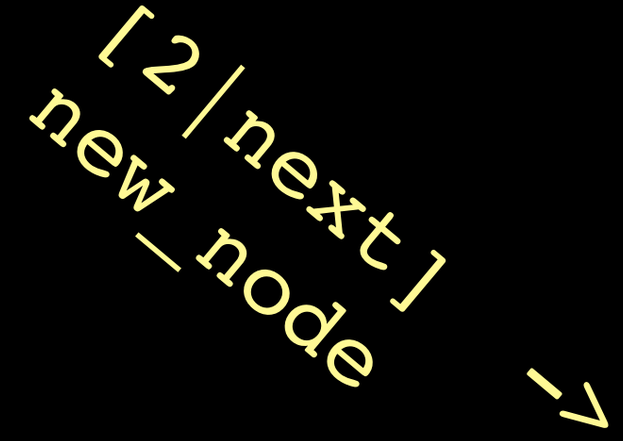
The while loop quits now.

We can make a new node:

```
new_node = {Node,2,ptr_new(node)}
```

But where does this leave us?

Inserting [2|...]




```
head->[1|next]->[7|next]->!null
```

`node` is this LL item

How do we get the previous node to point to our `new_node`? We should have kept track of where we were before.

Traversing a Linked List

```
head->[1|next]->[7|next]->!null
```

```
node = *head
next_node = *(node.next)
```

```
head->[1|next]->[7|next]->!null
       node        next_node
```

Now we can do

```
new_node = {Node,2,ptr_new(next_node)}
node.next = ptr_new(new_node)
```

to get

```
head->[1|next]->[2|next]->[7|next]->!null
       node        new_node   next_node
```

Oh no... that's all wrong still.

- Turns out, if you do
  `node = *head`
  it creates a *copy* of the node `head` points to
- This is bad - you actually have to use
  `current_ptr = head`
  to copy the *pointer* instead of the *node*
- We'll see this in tutorial....

Working with nodes

- In tutorial, you'll write a toolkit for linked lists
- Be sure to TEST your results! Make sure you get out what you expect!