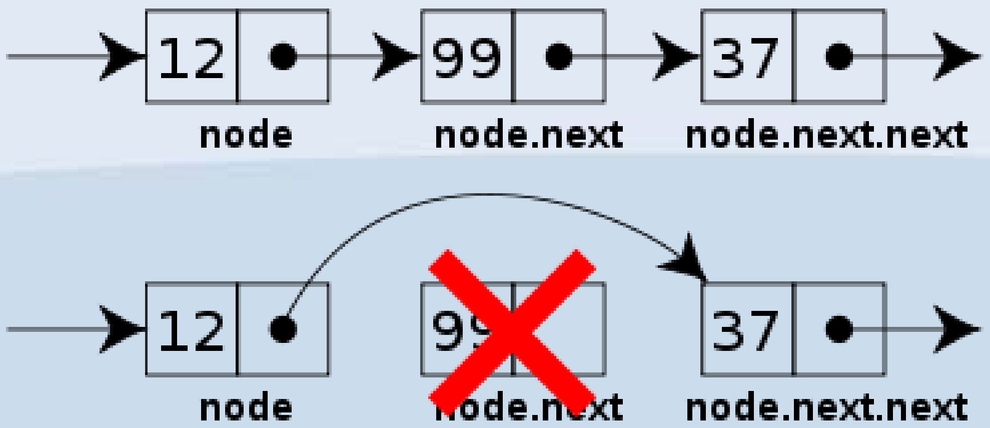
Linked Lists Part I

Ciaran Kelly



Linked lists

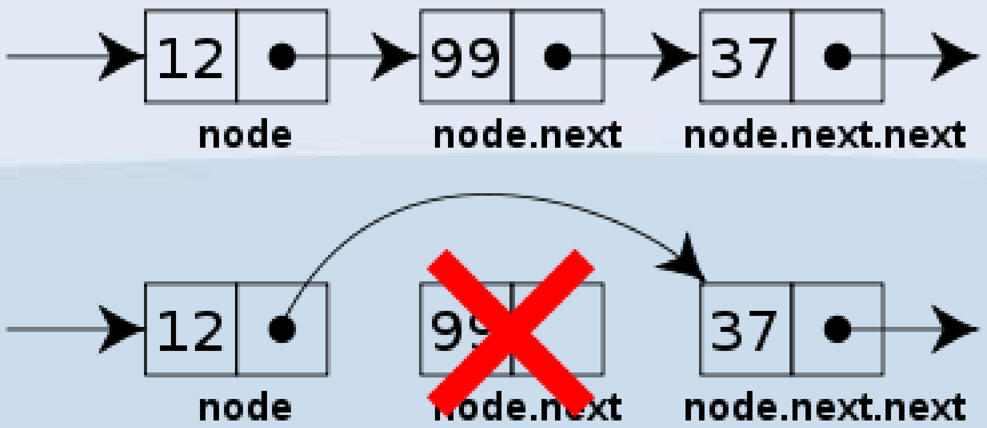
❖ Definition

- Linear list of ordered sequence of items.
- Each item is called a NODE and consists of two parts; item data and pointer to next node.
- A pointer is a variable in a program which contains the memory address of another variable.
- We are interested in the data stored at the pointer address.

Characteristics of a Linked List

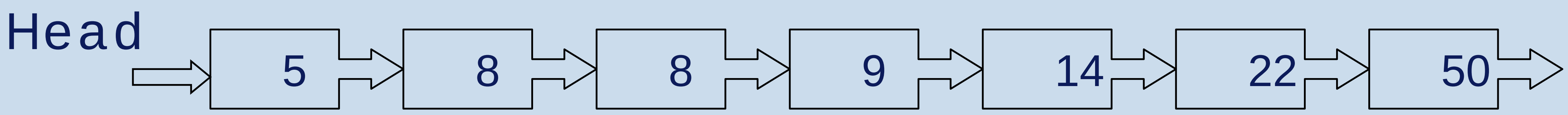
- Linked lists are non-sequentially mapped, i.e. non-contiguous.
- Start of list is a stand-alone pointer. (head)
- End of list points to NULL. (tail)
- Nodes may be ordered by data contents, or in the order in which they are processed by the program.

Declaration



- List structure:
  - A node in a list may be called NODE
    - consists of NODE.data, which is the data part
    - and NODE.next, which is the pointer to the next node in the list.
  - The pointer which points to the start of the list is usually called head.
  - Note: The pointer to the node NODE is called Nodeptr, i.e. we usually refer to node by its pointer; language uses an arrow to indicate pointer.

Pictorial Representation of List



Each node is made up

of a data part NODE.data and

a pointer to the next node, NODE.next

The final node points to NULL

Linked-List Usage

- Any algorithmic function requiring non fixed-length, non-contiguous, yet sequential structures.
- Linked list Operations
  - Initialise list
  - Empty list
  - Print list elements in order
  - Add a node in order
  - Delete a node in order
  - Reverse print

Initialise list

- This can be used to initialise the list to empty.

Head

```
algorithm InitializeList(Head)
    Head := NULL              // an empty list
end InitializeList
```

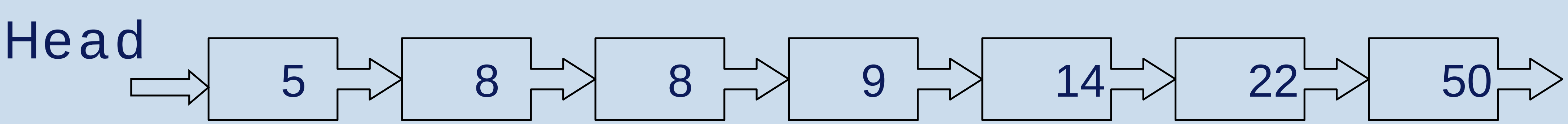
Emptylist

This can be used to check if the list is empty

```
bool  EmptyList (Head)      // returns  true/false
   If Head = NULL then
      List_is_empty = true
   else
      List_is_empty = false
   end if
  return  List_is_empty
end EmptyList
```

Print all list elements in order

```
Algorithm PrintList(Head)
NODEPTR = Head
while NODEPTR not = NULL
  Print NODEPTR→DATA                    // print data
NODEPTR = NODEPTR→NEXT  // next node
end while
end PrintList
```



Recursion

- This is where an algorithm calls itself, from within itself. Recursion is powerful way to solve some problems.

- Recursive procedure to print a linked-list:

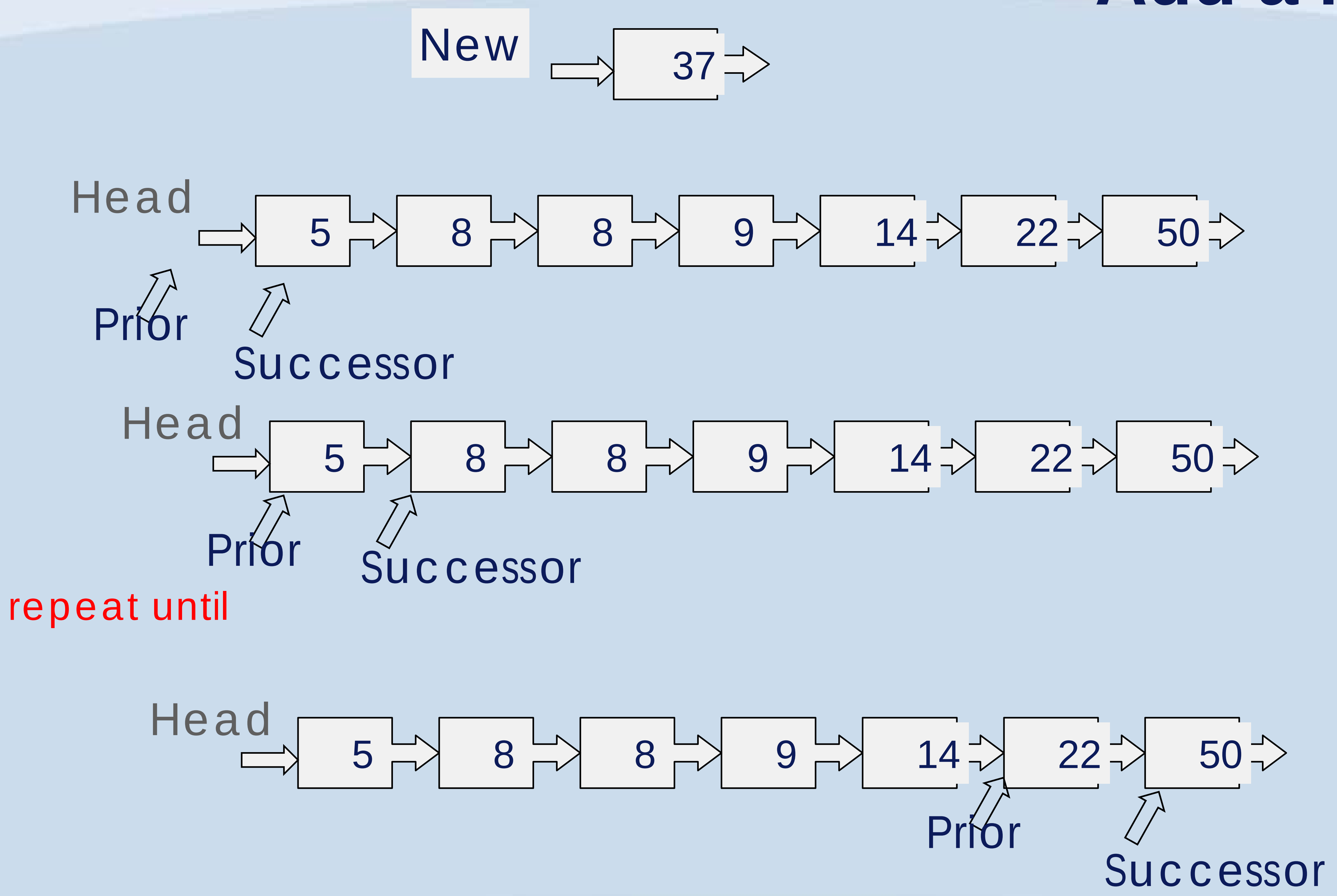
```
PRINTLIST (NODEPTR)     // print list starting at NODEPTR
    If NODEPTR not NULL
         PRINT "Data is ", NODEPTR->DATA
         // Recursive call to print the list starting at the next node.
         PRINTLIST (NODEPTR->NEXT)
    end if
end PRINTLIST
```

- To invoke this routine:PRINTLIST(Head)

Add a node

- Add a new node NEW, in the correct position in the list.
- Algorithm
  - Find the position where the new node goes.
    - To do this, use two other node pointers PRIOR and SUCCESSOR, to denote the node before and after the new node.
  - Connect the new node to its SUCCESSOR
  - Disconnect the PRIOR node from the SUCCESSOR and reconnect it to the new node.

Add a node


New
37
Head
5
8
8
9
14
22
50
Prior
Successor
Head
5
8
8
9
14
22
50
Prior
Successor
repeat until
Head
5
8
8
9
14
22
50
Prior
Successor

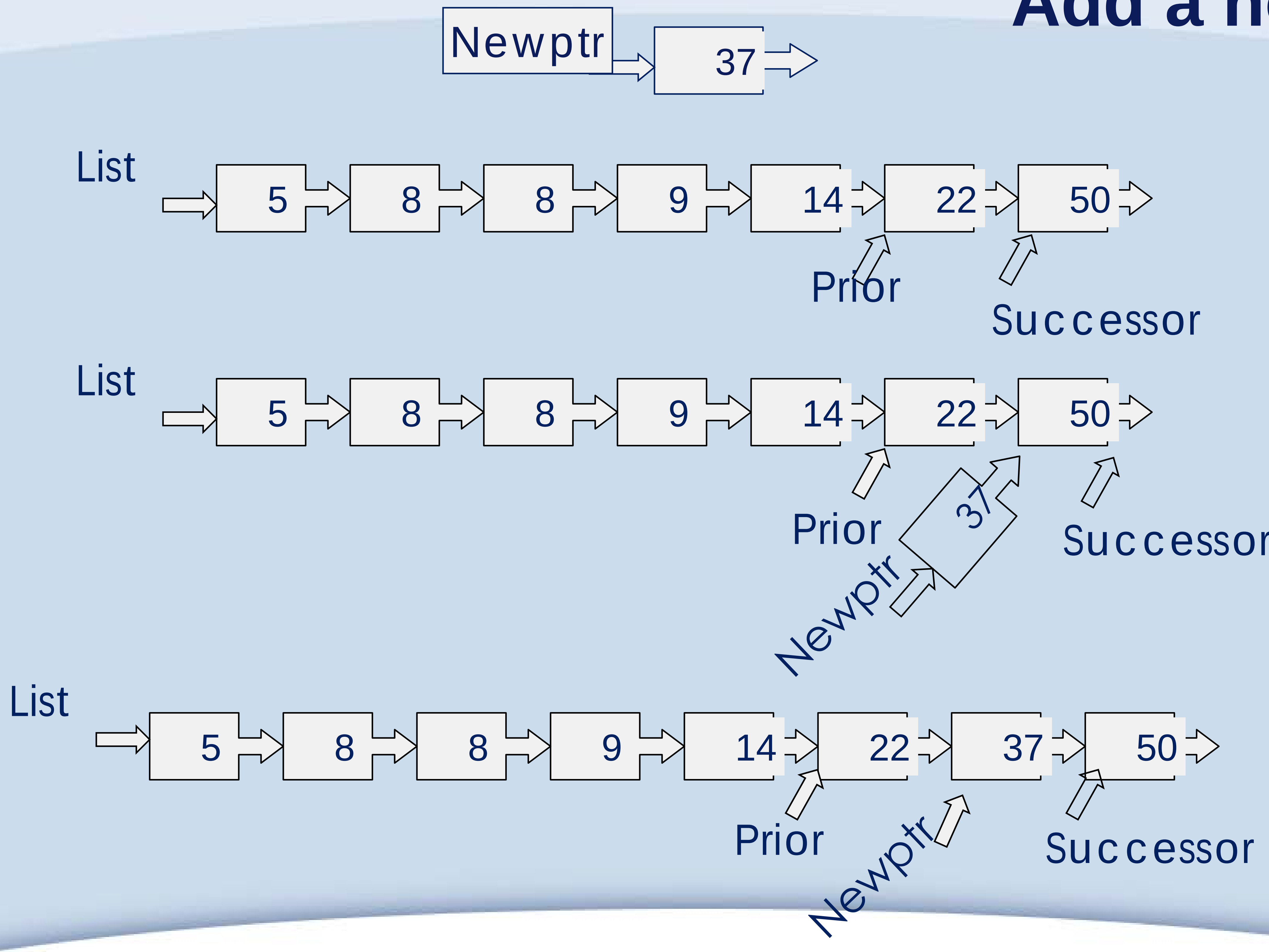
Find position for new node

- Constructs
  - Start at start of list, assuming the new node will be the first node.
  - Loop
    - Compare the new node data with the current node data.
      - Selection. If new node data > current node data, it will go after the current node. The current node becomes PRIOR and the next node becomes SUCCESSOR.
  - While the current node data < new node data or SUCCESSOR = NULL (i.e. end of list)

Find position for a new node

```
Algorithm  FINDPOS(Head, NEWPTR)
   PRIOR = NULL
   SUCCESSOR = Head
   FOUND = false            //  boolean, true when position is found
   While SUCCESSOR not = NULL and FOUND = false
      If SUCCESSOR→DATA < NEWPTR→DATA then
         PRIOR = SUCCESSOR
         SUCCESSOR = SUCCESSOR→NEXT
      else
         FOUND = true
      end if
   end-while
end procedure
```

Add a node


Newptr
37
List
5
8
8
9
14
22
50
Prior
Successor
List
5
8
8
9
14
22
50
Prior
37
Newptr
Successor
List
5
8
8
9
14
22
37
50
Prior
Newptr
Successor

AddNode

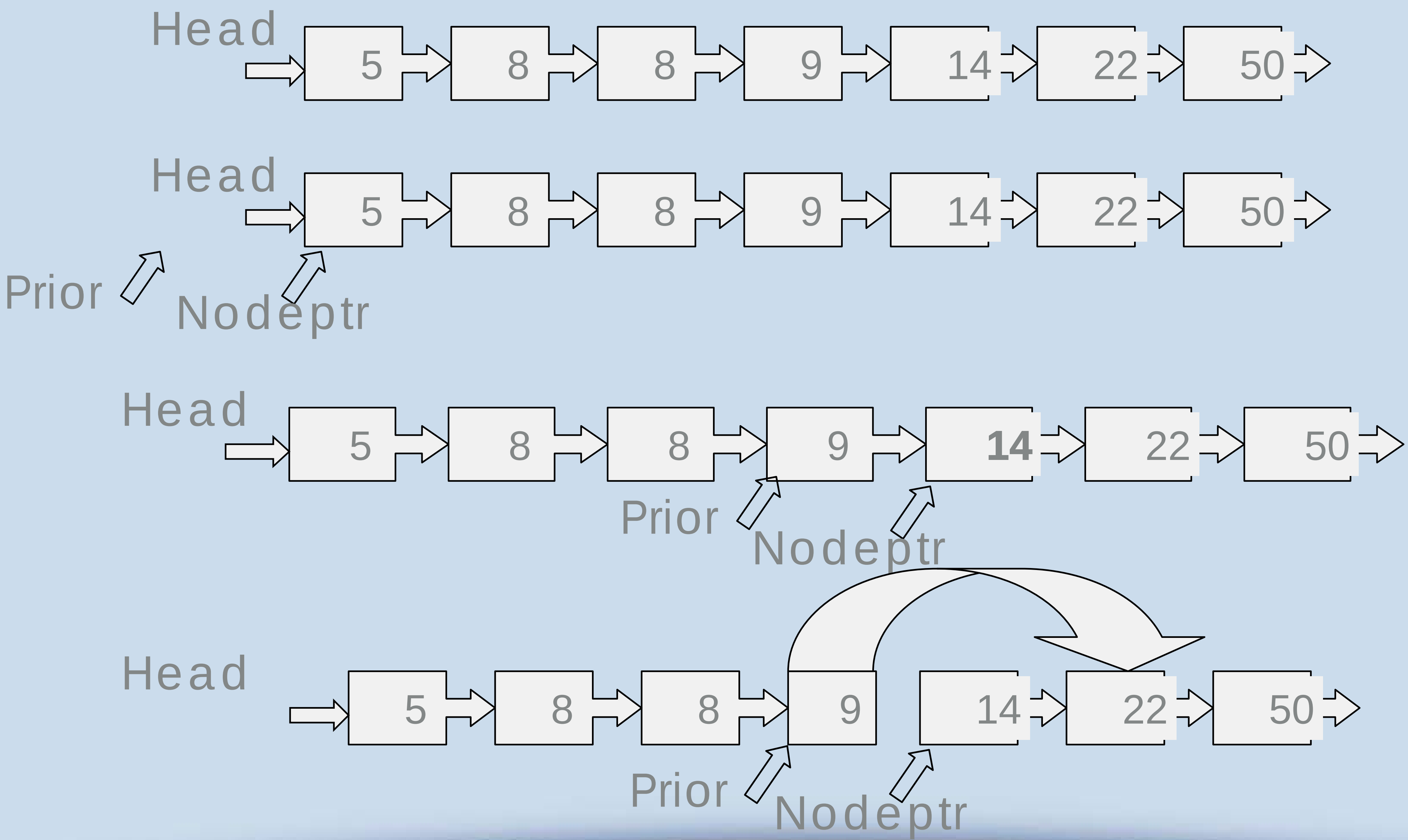
```
Algorithm AddNode(NEWPTR)
 NEWPTR→NEXT = SUCCESSOR
 If SUCCESSOR = Head
    Head = NEWPTR  // new node is at head
 else
    PRIOR→NEXT = NEWPTR
     // new node  is after PRIOR Node
 end if

end AddNode
```

Delete a node

- Delete a node NODE (pointer NODEPTR), containing a value X, from the list.
- Find the position of NODE and the previous node PRIOR in the list.
- Point PRIOR->NEXT to NODEPTR->NEXT
- Free up memory used by the NODE

Delete a node

Delete a node with data value **14**.



FindNode in a sorted list

```
Algorithm FindNode(X, PRIOR, FOUND)
  PRIOR = NULL
  NODEPTR = Head
  FOUND = false
  while NODEPTR<>NULL and NODEPTR→DATA < X
    PRIOR = NODEPTR
    NODEPTR = NODEPTR→NEXT
  end while
  if NODEPTR→DATA = X
    FOUND = true
  end if
end FindNode
```

Delete node NODE (pointer NODEPTR) from the list

```
Algorithm DeleteNode(NODEPTR, PRIOR)
   If NODEPTR = Head then
      Head = NODEPTR→NEXT
   else
      PRIOR→NEXT = NODEPTR→NEXT
   end if
   //Delete memory occupied by NODE
end DeleteNode
```

Implementation- C

```
struct listNode {          // self-referential structure
   int      data;
   struct listNode  *nextPtr;
};
```

- nextPtr is declared as a pointer to a listNode.
- The "*****" means that nextPtr is a pointer variable.
- the node consists of:
  - (i) the data,
  - (ii) a pointer to a node of the same type.
- In C the node is implemented as a ***struct***.