Section 7: Linked Lists

- Motivation
- Setting up the data types
- Prepend a data item
- Append a data item
- Search for a data item
- Deleting data items, deallocation
- Deleted the first element
- Delete the last element

Motivation

- Collection of data items
  - How many data items will the program have at any stage?
  - If a known fixed amount or realistic upper limit, an array is an appropriate data structure
  - If the size is unknown and we want the collection to grow (and possibly shrink), a dynamic data structure is needed
- Linked list is the simplest (but usually not the most efficient) dynamic data structure

Required Data Types

- Suppose the data items are records, for example:

```
subtype name_string is string(1..10);
type Data_Item is record
    count : integer;
    name: name_string;
end record;
```

- To form a linked list, we need to add an access type component to the record
    - Access to the modified record, not Data_Item

Required Data Types

- If the extended record is type Data_Node and the access type is type Data_Link
  - Type Data_Node must have a Data_Link component
  - Data_Node is defined in terms of Data_Link
  - Data_Link is defined in terms of Data_Node
  - Circular dependency in type definitions

Required Data Types

```
type Data_Link is access Data_Node;
type Data_Node is record
    count : integer;
    name  : name_string;
    next  : Data_Link;
end record;
```

Required Data Types

- If the extended record is type Data_Node and the access type is type Data_Link
    - Type Data_Node must have a Data_Link component
    - Data_Node is defined in terms of Data_Link
    - Data_Link is defined in terms of Data_Node
    - Circular dependency in type definitions
        - Break the cycle by starting with an incomplete declaration of Data_Node

```
type Data_Node;
```

Required Data Types

- The type declarations to set up the linked list

```
type Data_Node;
type Data_Link is access Data_Node;
type Data_Node is record
    count : integer;
    name  : name_string;
    next  : Data_Link;
end record;
```

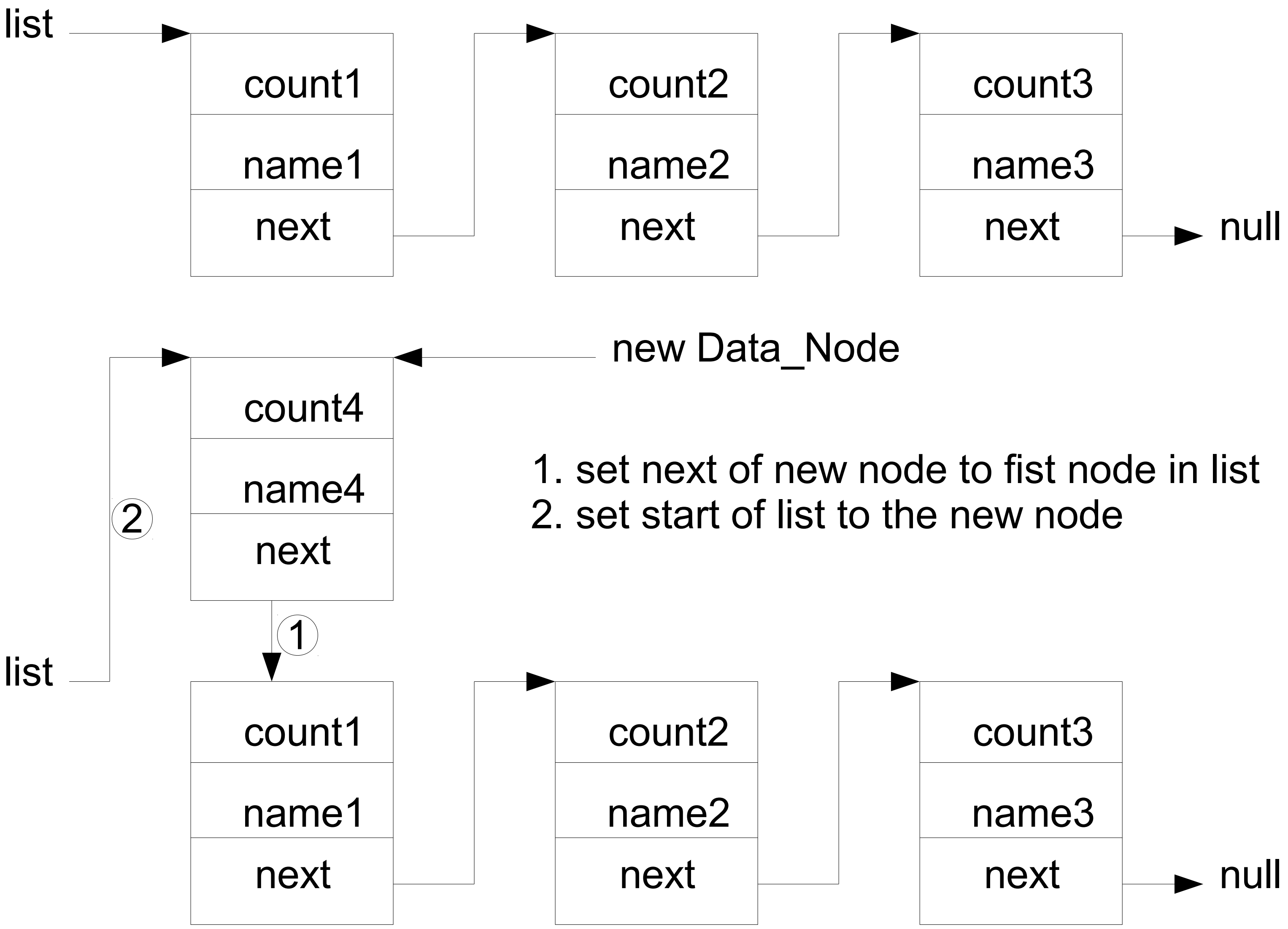
- Now, a linked list is just a variable of type Data_Link

```
mylist : Data_Link;    -- null by default
```

Adding Data Items to a List

- In adding an item to the list, we may have particular needs about where on the list we want to add the item
    - Prepend to the front of the list
    - Append to the end of the list
    - Insert in an ordered list, for example in order of the count component

Prepend a Data Item



1. set next of new node to fist node in list
2. set start of list to the new node

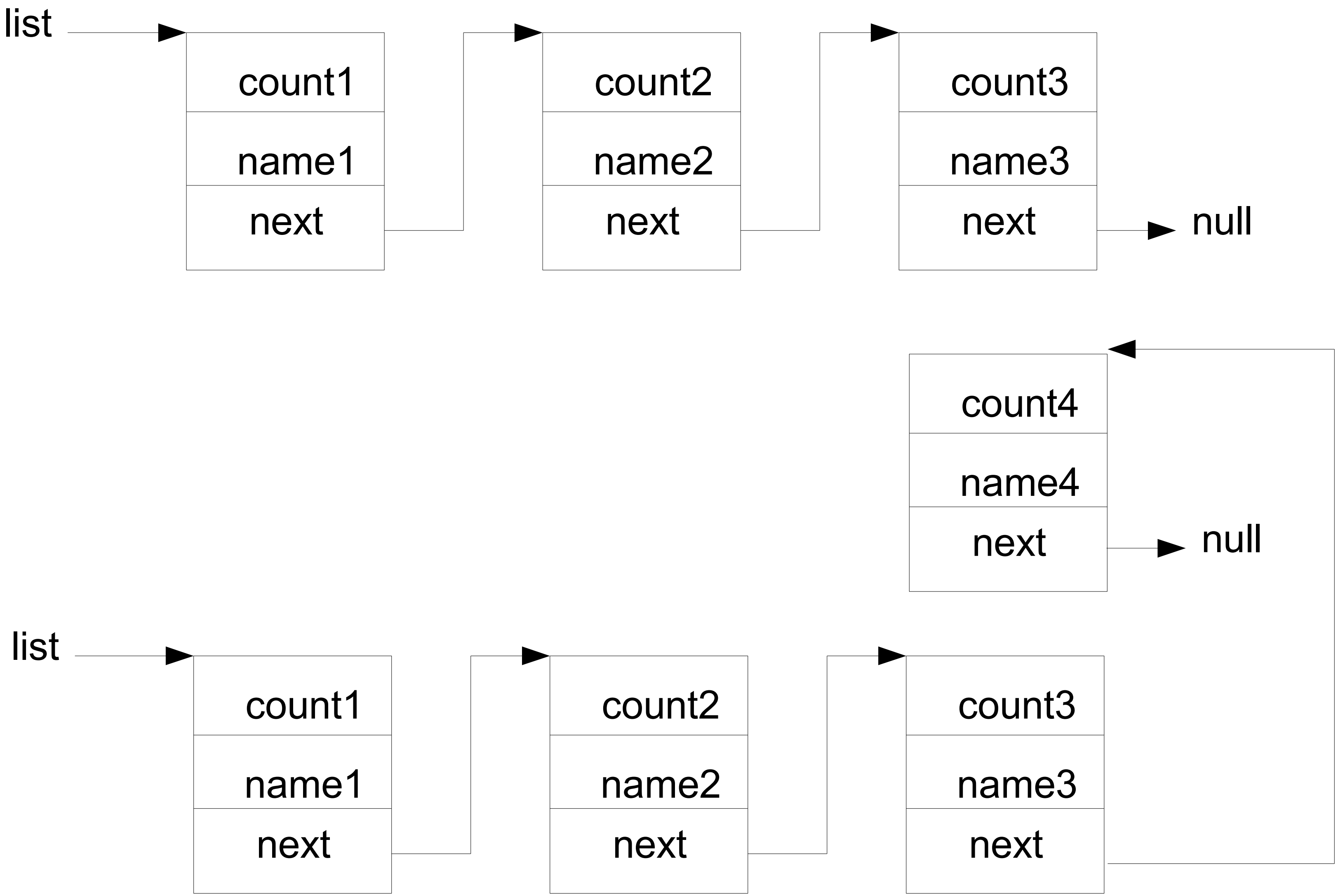
Prepend a Data Item

```
procedure prepend(list : in out Data_Link;
                  num  : in integer;
                  who  : in name_string) is

    node : Data_Link
           := new Data_Node'(num, who, null);

begin
    node.next := list;
    list := node;
end prepend;
```

Append a Data Item



Append a Data Item - Iterative

```
procedure append(list : in out Data_Link;
                 num  : in integer; who : in name_string) is
   node : Data_Link := new Data_Node'(num, who, null);
   curr : Data_link := list;
begin
   -- if the list is empty, the new node becomes the list
   if list = null then
      list := node;
   else -- list is not empty

      -- step through the list to the last node
       while curr.next /= null loop
        curr := curr.next;
      end loop;

      -- link the last node to the new node
      curr.next := node;
   end if;
end append;
```

Append a Data Item - Recursive

```
procedure append(list : in out Data_Link;
                 num  : in integer;
                 who  : in name_string) is
begin
   if list = null then
      list := new Data_Node'(num. who, null);
   else
      append(list.next, num, who);
   end if;
end append;
```

Search for an Item in the List

- Example: return the name component of an item given its count component

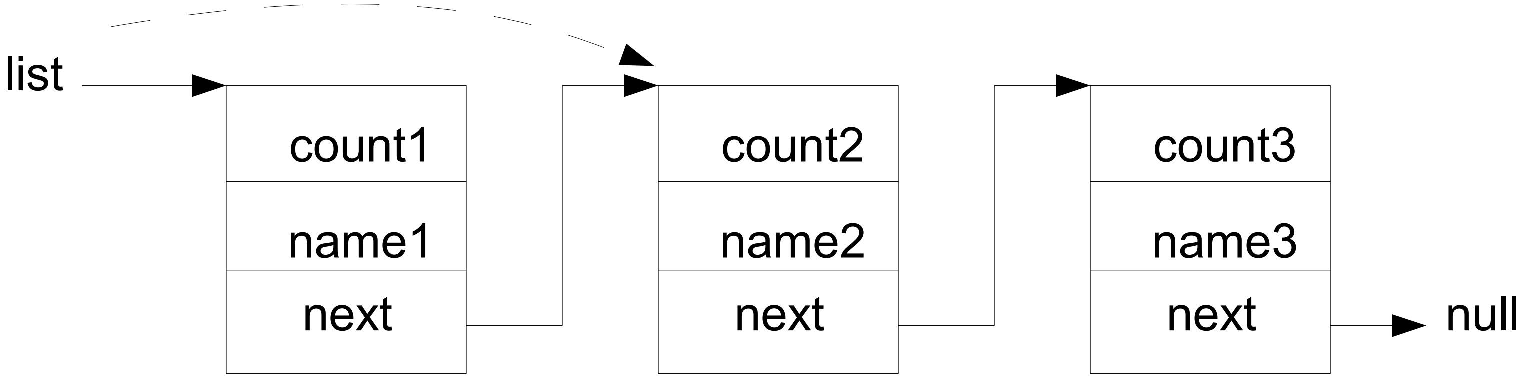
```
function find_name(list : Data_Link;
                   num : integer)
                   return name_string is
   curr : Data_Link := list;
begin
   while curr /= null loop
      if curr.count = num then
         return curr.name;
   end loop;
   raise not_found; – or return some string
end find_name;
```

Deleting Items from a List

- When deleting dynamically allocated items we should deallocate the memory assigned to those items
  - Otherwise, memory leakage.
- Ada provides a generic procedure for this purpose
  - Need to instantiate for the data item type and the access type

```
Procedure Kill is new
Ada.Unchecked_Deallocation(Data_node,
                           Data_Link);
```

Deleting the First Item



```
procedure delete_first
          (list : in out Data_Link) is
   node : Data_Link := list;
begin
   if list = null then
      return;
   else
      list := list.next;   -- reference the second Data_Node
      Kill(node);           -- clean up the deleted Data_Node
   end if;
end delete_first;
```

Deleting the Last Item

```
procedure delete_last(list : in out Data_Link) is
   curr : Data_Link := list;
   prev : Data_Link := null;
begin
   -- if it's an empty list there is nothing to be done
   if curr = null then
      return;
   end if;

   -- walk along the list until curr is the last node
   while curr.next /= null loop
      prev := curr;
      curr := curr.next;
   end loop;

   -- curr is the node to be deleted
   Kill(curr);
   if prev = null then
      list := null;
   else
      prev.next := null;
   end if;
end delete_last;
```