Introduction to Computers and Programming

Prof. I. K. Lundqvist

Lecture 5
Mar 29 2004

Today: Access Types

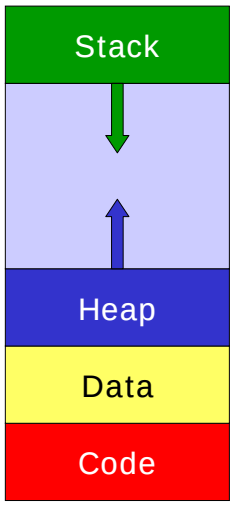
- Stack vs. Heap
- Allocating memory
- Deallocating memory
- Difference from C model
- Linked lists

Dynamic Data Structures

- Arrays have disadvantages:
  - size must be known at compile time
  - space wasted or insufficient (overflow)
  - inserting to/removing from middle requires shifting the elements
- **Idea**: Use dynamic, variable sized memory (linked lists)
  - size starts at zero, changes as necessary
  - space (de)allocated by programmer
  - ***access variables*** needed

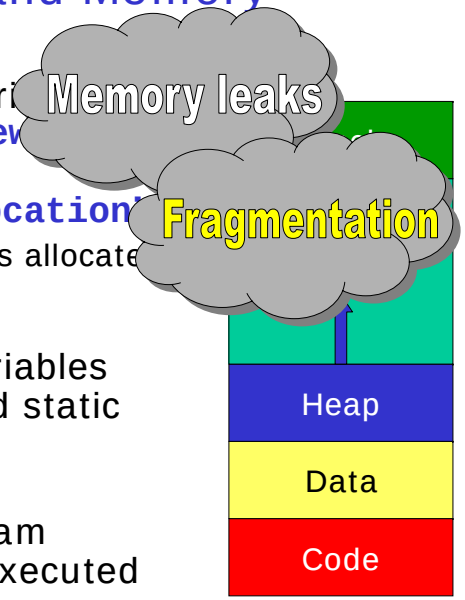
Programs and Memory

- **Stack**: stores variables that are local to functions
  - All **static** memory is allocated from the stack
  - when a functions is called, its automatic variables are allocated on the top of the stack
  - when it ends its variables are de-allocated



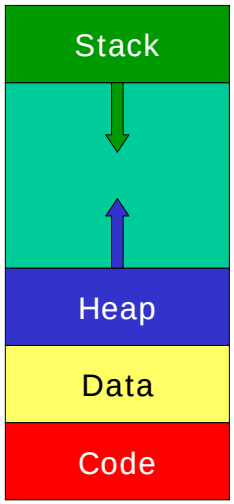
Programs and Memory

- **Heap**: place for vari
  are created with 'new
  disposed by
  'unchecked_deallocation'
  - **Dynamic** memory is allocate
    from the heap
- **Data**: initalized variables including global and static variables
- **Code** (text): program instructions to be executed



Programs and Memory

- **Heap**: place for variables that are created with 'new' and disposed by 'unchecked_deallocation'
  - **Dynamic** memory is allocated from the heap
- **Data**: initalized variables including global and static variables
- **Code** (text): program instructions to be executed

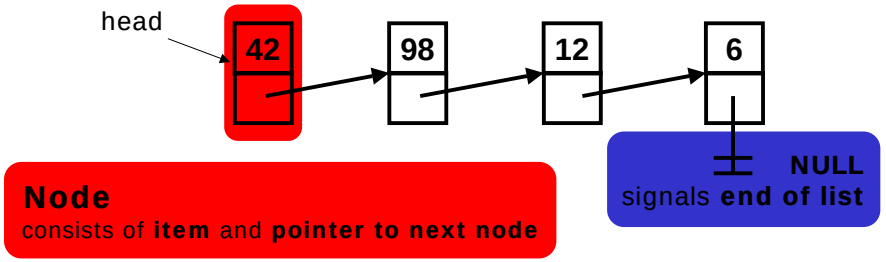


Stack vs. Heap

- Stack
  - Grows "down"
  - Operations always take place at the top
    - Push and pop are well organized
  - Support for nested functions and recursion
- Heap
  - Grows "up"
  - The order in which objects are created or destroyed is completely under the control of the programmer
  - You can have 'holes'
  - Dynamic memory management
  - Memory fragmentation - memory fragments into small blocks over lifetime of program

Linked Lists

- **Linked list**: a list of nodes (records), each pointing to the next node
- List data type:
  - insert/delete anywhere
  - sequence of nodes
  - implement using linked memory
- Singly linked list:

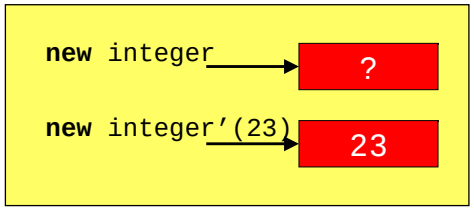


List Operations

- **Go to** a position in the list.
- **Insert**: an item at a position in the list.
- **Delete**: an item from a position in the list.
- **Retrieve**: an item from a position.
- **Replace**: an item at a position.
- **Traverse**: a list.
- **Search**: for an item in the list.

Storage allocation

- Allocator
  - **new T**
    - T is arbitrary data type
    - memory is *allocated* for an object of type T
    - pointer to that memory is returned
  - **new T'(value)**
    - memory allocated as above
    - initial value stored in that memory
    - pointer to the memory is returned
  - examples:

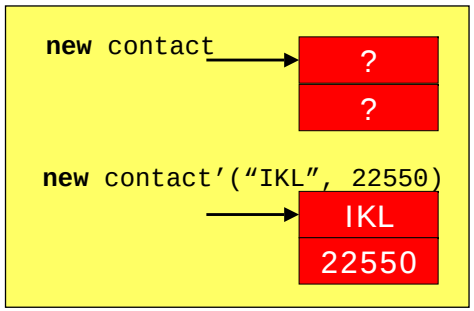


Allocation with records

- data type for dynamic data structures is usually a record

```
type contact is
   record
      initials  : string(1..3);
      extension : integer;
   end record;
```

- examples:



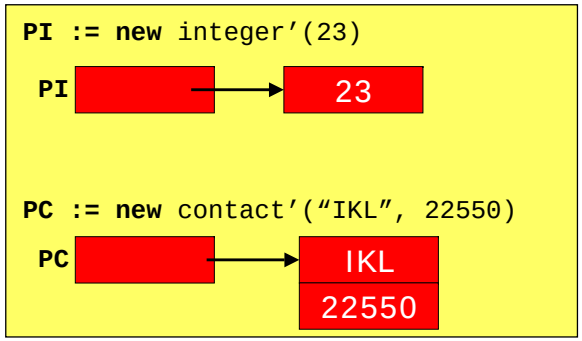
Access types

- Access types
  - Can declare variables of ***access types***

```
type int_pointer     is access integer;
type contact_pointer is access contact;

PI : int_pointer;
PC : contact_pointer;
```

Access variables

- Access variables
  - use *access variables* to save pointers to allocated objects



Access variables

- Access variables are ***pointers***
  - Do ***not*** contain data
  - Contain ***pointer*** to data
- Access variables provide ***indirect*** access to data
- Note: there are **no** arithmetic operations defined for Access Types
- Access variables are initialized by default to **null**
  - **null** = does not point to any data

Accessing an object

- If the object is a **record**; to access a field:
  - **P.fieldname**
- to access the **entire object** pointed to by P
  - **P.all**
- If the object is an **array;** to access an element:
  - **P(i)**
- CONSTRAINT_ERROR if P has value **null**
- P is not affected when the object is accessed

Storage Deallocation

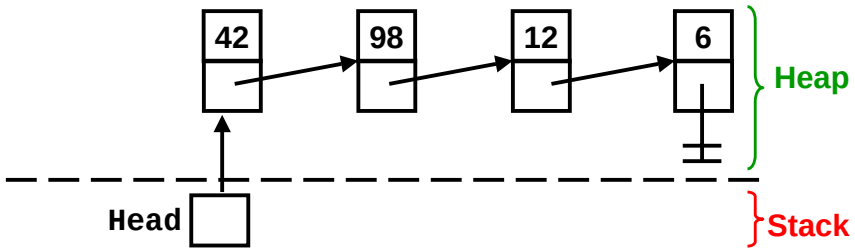
- Deallocation on exit
  - Automatically carried out by execution environment
  - Safe, Avoids dangling pointers
  - Inefficient, effect execution time
- Programmer Controlled
  - Need to use Unchecked_Deallocation
  - Risky, may have dangling pointers
  - Efficient, space returned to heap

```
with UNCHECKED_DEALLOCATION;
…
type ListNode is ... ; -- type definition
type ListPtr is access ListNode;
…
procedure free is new unchecked_deallocation (ListNode,ListPtr);
```

Linked lists

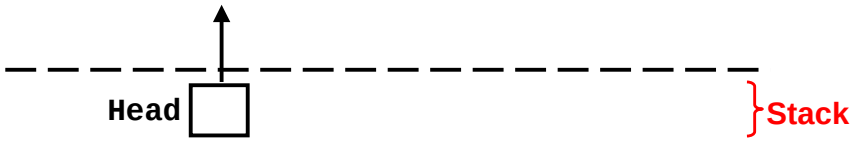
- Define a node:
  - Define a record type
  - Define an access type

```
type List_Node;
type List_Ptr  is access List_Node;
type List_Node is
   record
      Element : Element_Type;
      Next    : List_Ptr;
   end record;
type List is
   record
      Head : List_Ptr;
   end record;
```



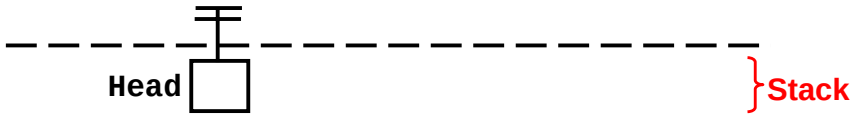
Linked List Creation

```
   procedure Initialize (L : in out List) is
   begin
      L.Head := null;
   end Initialize;

procedure List_Test is
   My_List : List;
   Element : Element_Type;

begin
   Initialize(My_List);
```



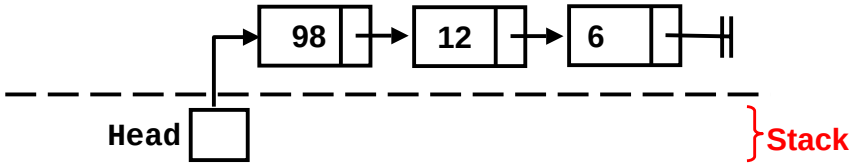
Linked List Creation

```
   procedure Initialize (L : in out List) is
   begin
      L.Head := null;
   end Initialize;

procedure List_Test is
   My_List : List;
   Element : Element_Type;

begin
   Initialize(My_List);
```



Linked List Creation

```
procedure List_Test is
   My_List : List;
   Element : Element_Type;

begin
   Initialize(My_List);
   Add_To_Front(My_List, 6);
   Add_To_Front(My_List,12);
   Add_To_Front(My_List,98);
```

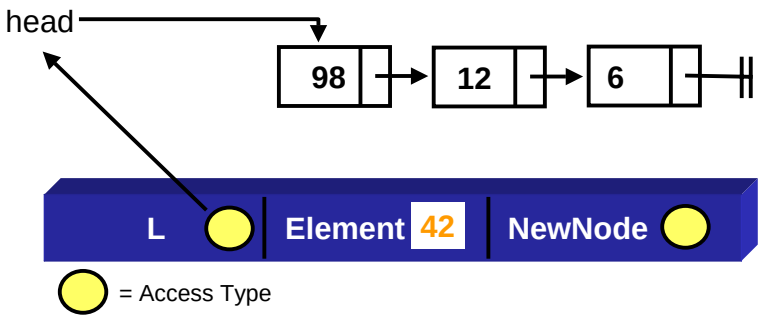


Ex: Linked lists

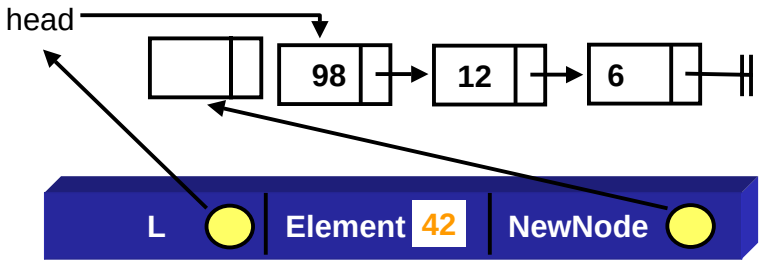
- Insert:
  - at front
  - at end
  - after element *x*
- Delete:
  - front
  - end
  - element *x*

Linked_List.ads
Linked_List.adb
List_Test.adb

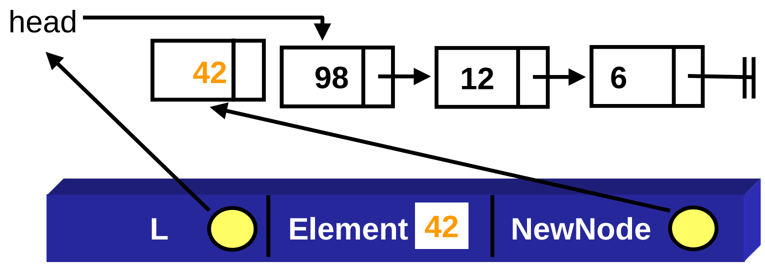
```
procedure AddToFront (L: in out List; Element: in ElementType) is
   NewNode: ListPtr;
 begin
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := L.Head;
   L.Head := NewNode;
 end AddToFront;
```



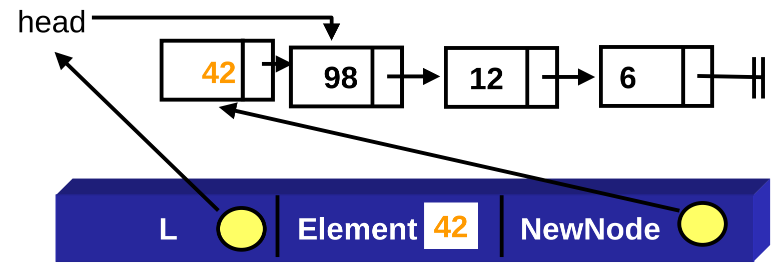

```
procedure AddToFront (L: in out List; Element: in ElementType) is
   NewNode: ListPtr;
 begin
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := L.Head;
   L.Head := NewNode;
 end AddToFront;
```



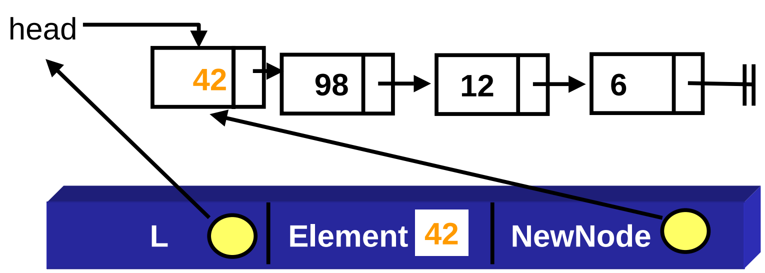

```
procedure AddToFront (L: in out List; Element: in ElementType) is
   NewNode: ListPtr;
 begin
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := L.Head;
   L.Head := NewNode;
 end AddToFront;
```




```
procedure AddToFront (L: in out List; Element: in ElementType) is
   NewNode: ListPtr;
 begin
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := L.Head;
   L.Head := NewNode;
 end AddToFront;
```




```
procedure AddToFront (L: in out List; Element: in ElementType) is
   NewNode: ListPtr;
 begin
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := L.Head;
   L.Head := NewNode;
 end AddToFront;
```

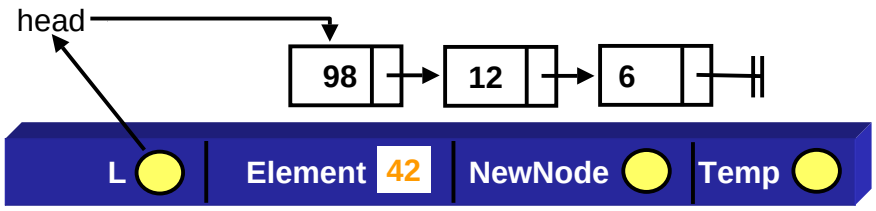



```
procedure AddToFront (L: in out List; Element: in ElementType) is
   NewNode: ListPtr;
 begin
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := L.Head;
   L.Head := NewNode;
 end AddToFront;
```

Ex: Linked lists

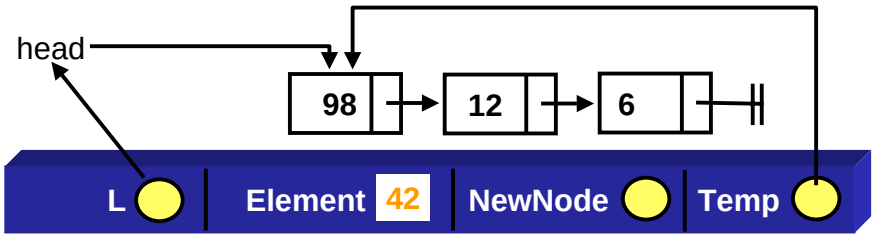
- Insert:
  - at front
  - at end
  - after element *x*
- Delete:
  - front
  - end
  - element *x*

```
procedure AddToEnd(L: in out List; Element: in ElementType) is
   Temp : ListPtr;
   NewNode: ListPtr;
 begin
   Temp:= L.Head;
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := null;
   while Temp.next /= null loop
         Temp := Temp.next;
   end loop;
   Temp.next := NewNode;
 end AddToFront;
```



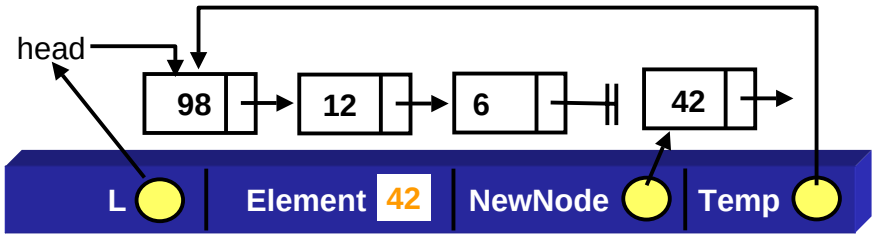
= Access Type

```
procedure AddToEnd(L: in out List; Element: in ElementType) is
   Temp : ListPtr;
   NewNode: ListPtr;
 begin
   Temp:= L.Head;
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := null;
   while Temp.next /= null loop
         Temp := Temp.next;
   end loop;
   Temp.next := NewNode;
 end AddToFront;
```



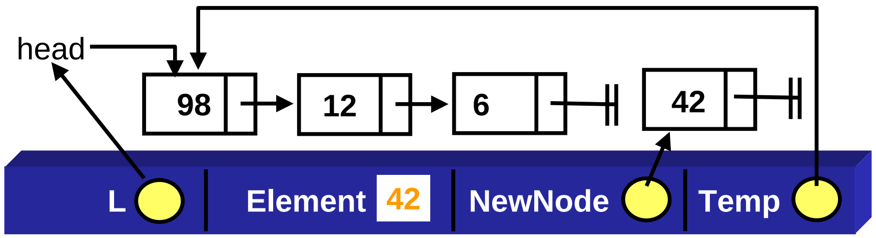
= Access Type

```
procedure AddToEnd(L: in out List; Element: in ElementType) is
   Temp : ListPtr;
   NewNode: ListPtr;
 begin
   Temp:= L.Head;
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := null;
   while Temp.next /= null loop
         Temp := Temp.next;
   end loop;
   Temp.next := NewNode;
 end AddToFront;
```



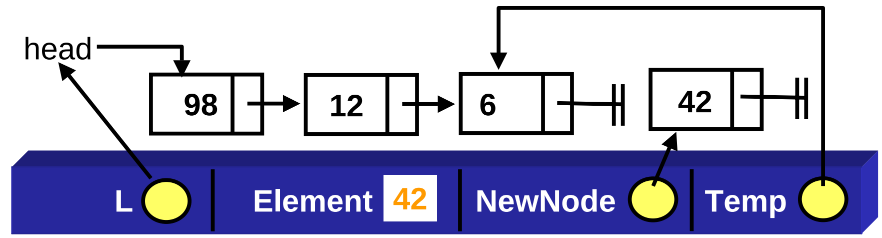
= Access Type

```
procedure AddToEnd(L: in out List; Element: in ElementType) is
   Temp : ListPtr;
   NewNode: ListPtr;
 begin
   Temp:= L.Head;
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := null;
   while Temp.next /= null loop
         Temp := Temp.next;
   end loop;
   Temp.next := NewNode;
 end AddToFront;
```



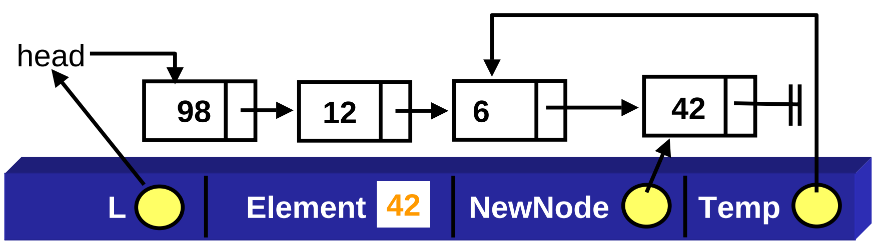
= Access Type

```
procedure AddToEnd(L: in out List; Element: in ElementType) is
   Temp : ListPtr;
   NewNode: ListPtr;
 begin
   Temp:= L.Head;
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := null;
   while Temp.next /= null loop
         Temp := Temp.next;
   end loop;
   Temp.next := NewNode;
 end AddToFront;
```



= Access Type

```
procedure AddToEnd(L: in out List; Element: in ElementType) is
   Temp : ListPtr;
   NewNode: ListPtr;
 begin
   Temp:= L.Head;
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := null;
   while Temp.next /= null loop
         Temp := Temp.next;
   end loop;
   Temp.next := NewNode;
 end AddToFront;
```



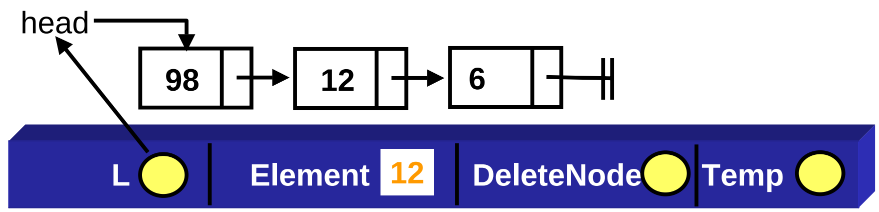
= Access Type

```
procedure AddToEnd(L: in out List; Element: in ElementType) is
   Temp : ListPtr;
   NewNode: ListPtr;
 begin
   Temp:= L.Head;
   NewNode:= new ListNode;
   NewNode.Element := Element;
   NewNode.next := null;
   while Temp.next /= null loop
         Temp := Temp.next;
   end loop;
   Temp.next := NewNode;
 end AddToFront;
```

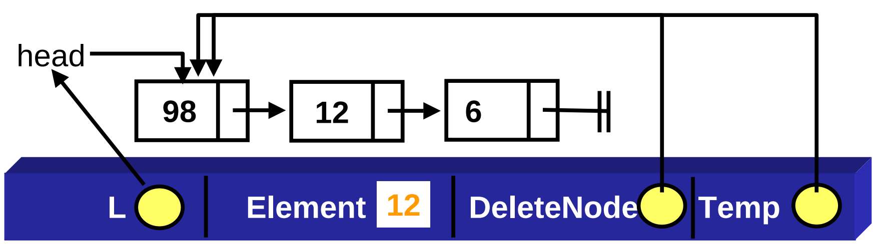
Ex: Linked lists

- Insert:
  - at front
  - at end
  - after element *x*
- Delete:
  - front
  - end
  - element *x*
- Additional variation: keep track of tail

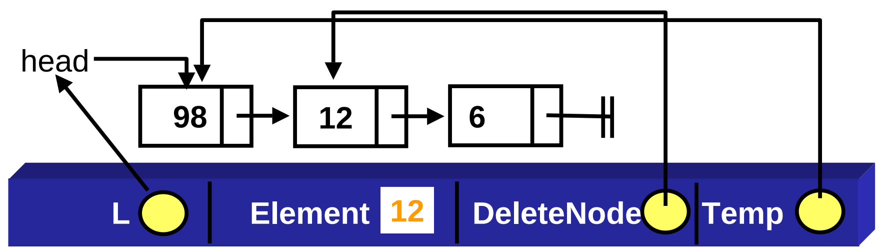
```
procedure DeleteElement(L: in out List; Element: in ElementType) is
    Temp : ListPtr;
    DeleteNode: ListPtr;
  begin
     Temp:= L.Head;
     DeleteNode := L.Head;
    while DeleteNode.Element /= Element and DeleteNode /=null loop
            Temp := DeleteNode;
            DeleteNode := DeleteNode.Next;
     end loop;
      if DeleteNode /= null then
            Temp.Next := DeleteNode.Next;
            free (DeleteNode);
      end if;
end DeleteElement;
```



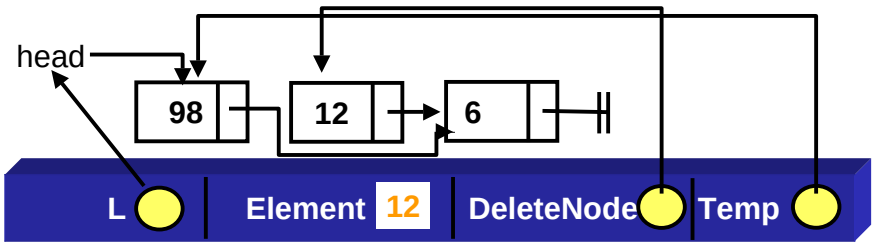

```
procedure DeleteElement(L: in out List; Element: in ElementType) is
    Temp : ListPtr;
    DeleteNode: ListPtr;
  begin
     Temp:= L.Head;
     DeleteNode := L.Head;
    while DeleteNode.Element /= Element and DeleteNode /=null loop
            Temp := DeleteNode;
            DeleteNode := DeleteNode.Next;
     end loop;
      if DeleteNode /= null then
            Temp.Next := DeleteNode.Next;
            free (DeleteNode);
      end if;
end DeleteElement;
```



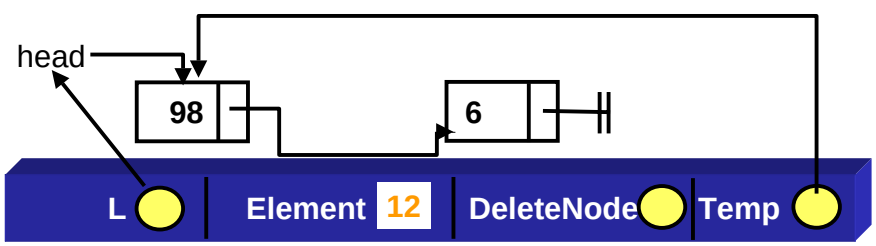

```
procedure DeleteElement(L: in out List; Element: in ElementType) is
    Temp : ListPtr;
    DeleteNode: ListPtr;
  begin
     Temp:= L.Head;
     DeleteNode := L.Head;
    while DeleteNode.Element /= Element and DeleteNode /=null loop
            Temp := DeleteNode;
            DeleteNode := DeleteNode.Next;
     end loop;
      if DeleteNode /= null then
            Temp.Next := DeleteNode.Next;
            free (DeleteNode);
      end if;
end DeleteElement;
```




```
procedure DeleteElement(L: in out List; Element: in ElementType) is
    Temp : ListPtr;
    DeleteNode: ListPtr;
  begin
     Temp:= L.Head;
     DeleteNode := L.Head;
    while DeleteNode.Element /= Element and DeleteNode /=null loop
            Temp := DeleteNode;
            DeleteNode := DeleteNode.Next;
     end loop;
      if DeleteNode /= null then
            Temp.Next := DeleteNode.Next;
            free (DeleteNode);
      end if;
end DeleteElement;
```




```
procedure DeleteElement(L: in out List; Element: in ElementType) is
   Temp : ListPtr;
   DeleteNode: ListPtr;
 begin
   Temp:= L.Head;
   DeleteNode := L.Head;
  while DeleteNode.Element /= Element and DeleteNode /=null loop
        Temp := DeleteNode;
        DeleteNode := DeleteNode.Next;
   end loop;
   if DeleteNode /= null then
        Temp.Next := DeleteNode.Next;
        free (DeleteNode);
   end if;
end DeleteElement;
```




```
procedure DeleteElement(L: in out List; Element: in ElementType) is
   Temp : ListPtr;
   DeleteNode: ListPtr;
 begin
   Temp:= L.Head;
   DeleteNode := L.Head;
  while DeleteNode.Element /= Element and DeleteNode /=null loop
        Temp := DeleteNode;
        DeleteNode := DeleteNode.Next;
   end loop;
   if DeleteNode /= null then
        Temp.Next := DeleteNode.Next;
        free (DeleteNode);
   end if;
end DeleteElement;
```

Other lists (pset)

- Doubly-linked
  - references linked to next and previous node
  - enables backward traversal, but more references to keep track of