Linked Lists

In this representation, the structure does not necessarily reflect the logical organization. Items may appear in any order. The logical organization is provided through pointers. Each item in the list, called a node, consists of a data portion containing the item information and a pointer, which contains the location (address) of the next item in the logical sequence that has to be maintained.

As with all ADT's we can define a set of operations on linked lists

Operation	**Description**
Initialize	Initialize internal structure; create an empty list
IsEmpty	True iff the list has no elements
Insert	Insert an Element into the list
Remove	Remove an element from the list
Empty	Delete all elements in the list and free the memory
Display	Display all elements in the list

Access Types

In so far, we have only seen static data types. For instance, the size of the arrays has to be declared ahead of time. If all the locations are not used, then the memory is wasted. If the size of the array is too small, then the array gets filled and the programs capability is limited.

Dynamic memory allocation is a means of providing more memory when it is needed. The ***new*** construct is used to allocate memory.

my_access_variable := **new** *my_access_type*;

The ***new*** operation takes memory from a storage pool (often called the **heap**) and reserves it for use of the variable *my_access_variable*. The reference to the memory location is stored in *my_access_variable*.

type Int_Ptr **is access** Integer;

P : Int_Ptr;

Issues with Access Types:

- Aliasing: referencing the same object with multiple names

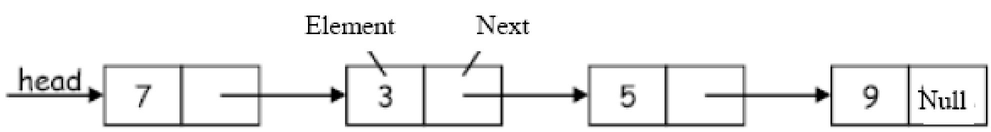
- Dangling reference: Referencing a deallocated object by another name leads to anomalous behavior. There is no guarantee on what will happen because the released memory may have been reallocated for some other purpose.

- Pointer dropping: disassociating an object from all names even when it is valid. The major drawback that arises from dropping a pointer is that it cannot be referenced again or be explicitly deallocated. A dropped pointer depends on an implicit memory manager for reclamation of space. An Ada environment is not required to provide deallocation of dynamically allocated objects.

Whenever you use dynamic allocation, it is possible to run out of space. Ada provides a facility (a length clause) for requesting the size of the pool of allocation space at compile time. However, you can still run out at run time, so you can create an exception handler for the `Storage_Error` exception.

Singly Linked List

A singly linked list of nodes (records) with fields: element and next, can be visualized as shown below:



We specify a node as follows:

Declarations

```
type Listnode;
type Listptr is access Listnode; --we define an access type or pointer to the node
type Listnode is record
        Element : Elementtype; --we define one field in the record to be of type element
        Next : Listptr; --we define the other field as a pointer to the next node
end record;

type List is record
        Head : Listptr; -- we define the head to be of type record.
end record;
```

Initialize

Preconditions : none
Post-Conditions: List with head pointer set to null.

Pseudo-Code :

```
        List.Head := Null;
```

Return List to the user

IsEmpty

Preconditions : none
Post-Conditions: Returns a Boolean variable determining if the list is empty.

Pseudo-Code :

```
If List.Head = Null then
    Return True
Else
    Return False
```

Insert

The insertion operation can take place anywhere in the list. We will list the pseudo code for inserting at the beginning and the end of the list.

Food for thought: Is it hard to insert anywhere in the list?

Insert at the beginning of the list

Preconditions : Element to be inserted and the List
Post-Conditions: List with element inserted

Pseudo-Code :

```
Create a new node and store the pointer to the node in NodePtr
Set NodePtr.Element to Element
Set NodePtr.Next to List.Head
List.Head is set to NodePtr
Return List to the user
```

Insert at the end of the list

Preconditions : Element to be inserted and the List
Post-Conditions: List with element inserted

Pseudo-Code :

```
Create a new node and store the pointer to the node in NodePtr
Set NodePtr.Element to Element
Set NodePtr.Next to Null
If the list is empty then
    List.Head := NodePtr
```

```
        Else
                Create a temporary pointer called Temp
                Set Temp to List.Head
                Traverse to the end of the list as follows:
                        While Temp.Next /= Null
                                Temp:= Temp.Next
                        Temp.Next := NodePtr
        Return List to the user
```

Removal

The removal operation can take place anywhere in the list. We will list the pseudo code for removing from the beginning and the end of the list.

Food for thought: Is it hard to remove from anywhere in the list?

Removal from the beginning of the list

Preconditions : Non-Empty List
Post-Conditions: List with the first element removed, and the element removed

Pseudo-Code :

```
        If the list is empty then
                Display cannot delete from an empty list
        Else
                Create a temporary pointer called Temp
                Set Temp to List.Head
                If Temp.Next = Null then
                        List.Head := Null
                Else
                        List.Head := Temp.Next
        Element:= Temp.Element
        Free Temp

    Return List and Element to the user
```

Remove from the end of the list

Preconditions : A non-empty List
Post-Conditions: List with element removed from the end and the element removed

Pseudo-Code :

```
        If the list is empty then
                Display cannot delete from an empty list
```

```
Else
        Create two temporary pointers called Temp and Prev
        Set Temp to List.Head
        Set Prev to null
        Traverse to the end of the list as follows:
                While Temp.Next /= Null
                        Prev :=Temp
                        Temp:= Temp.Next
                If Prev = Null then
                        List.Head:= Null
                Else
                        Prev.Next := Null
                Element := Temp.Element
                Free Temp
Return List and Element to the user
```

Empty

Preconditions : none
Post-Conditions: frees all the allocated nodes in the list.

Pseudo-Code :

```
Create a temporary pointer called Temp
While List.Head /= Null loop
        Temp := List.Head
        List.Head := Temp.Next
        Free Temp
```

Display

Preconditions : none
Post-Conditions: displays all the elements in the list.

Pseudo-Code :

```
Create a temporary pointer called Temp
Temp := List.Head
While Temp /= Null loop
        Display Temp.Element
        Temp:= Temp.Next
```

Jayakanth Srinivasan
I. Kristina Lundqvist