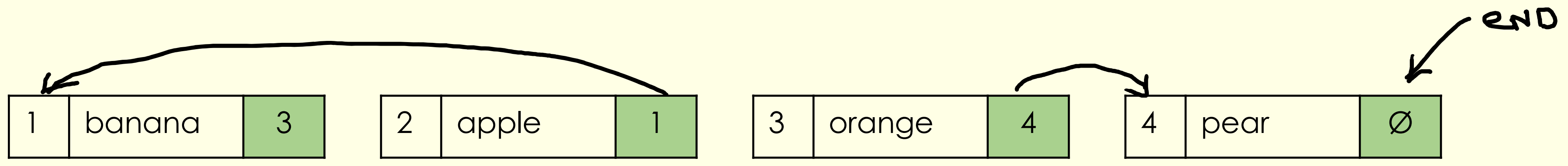
Linked Lists

- A linked list is a **dynamic data structure -** grow and shrink during runtime
- A linked list can be **unordered** or it can be **ordered** in some way, such as in alphabetical or numerical order.
- A linked lists uses **pointers** to order the data
- The items which form the sequence are not necessarily held in contiguous data locations, or in the order in which they occur in the sequence
- Items can be added and removed without having to shift any of the other items in the list
- Traversing a list always begins at the start/head node



What is the difference between a linked list and array?	What are some pros and cons of using linked lists?
	• The flexibility of a linked list is a pro, because you can insert at or delete from any position in constant time. • Another pro is that linked list use dynamic allocation, which allows the size of the linked list to not be a requirement for it to be known in advance. • A few cons of linked list is their complexity to use and access relative to arrays, the pointers require extra space, and they do not allow random access.

To implement a linked list requires **4 features**:

Node object – stores the data along with the pointer to the next item
Start Pointer is used to the first item
Null Pointer (∅) is used to specify the end of the list
Free Node is used to specify the next free node in the list

Node Object

```
class Node:
   public procedure new (pData)
      data = pData # instance variable to store the data (a name)
      next = None # instance variable with address of next node
   end procedure
end class
```

Using getter and setters methods in the node class

```
data =input()
new_node = new node(data) #node object
new_node.getData() #accessing the node data
new_node.getNext() # accessing the node pointer
new_node.setNext(next)
new_node.setData(data)
```

```
class node
   ....

   public function getData()
      return data
   endfunction

   public function getNext()
      return next
   endfunction

   public procedure setNext(pNext)
      next = pNext
   endprocedure

   public procedure setData(pData)
      data = pData
   endprocedure

endclass
```

A second class is defined to represent the list. This needs a single attribute:

Pointer to the head of the list

```
class linkedlist
   private head: node

   public procedure linkedlist() // constructor method
      head = null
   endprocedure
endclass

my_list = new linkedlist()// instantise an empty linked list object
```

Inserting a node

If the list is **ordered**, the new element must be inserted into the correct position.

Before

Node	Data	Pointer
1	banana	3
2	apple	1
3	orange	4
4	pear	Ø (Null))
5		

The following steps define the process for inserting a new element into a list where the elements are ordered in **ascending order**, from lowest to highest value. These steps work for both numerical order (from lowest to highest number), or alphabetical order (from A-Z)

1. Store the data into the free node pointer
   Free Node = 5
2. Identify where in the list it is to be inserted
3. Update the pointer of the previous item to use the new node pointer
4. New item will use the pointer of the previous item
5. Change the Free Node to the next free node

After:

Node	Data	Pointer
1	banana	5
2	apple	1
3	orange	4
4	pear	Ø (Null))
5	cherry	3
6		

Insert item algorithm – in order

1. **Create a new node** with the given data.
2. **Check if the list is empty**:
   - If yes, make the new node the **head** of the list.
3. **Check if the new node's data is less than the head's data**:
   - If yes, insert the new node at the **beginning** of the list.
   - Set the new node's next pointer to the current head.
   - Update the head to point to the new node.
4. **Otherwise, find the correct position in the list**:
   - Start from the head and **traverse the list**.
   - Continue looping through nodes while:
     - The current node's next is not null, **and**
     - The next node's data is still **less than** the new node's data.
   - Once the correct place is found:
     - Set the new node's next pointer to the current node's next.
     - Set the current node's next to the new node.

The following example shows the code for adding a new node into the correct position within a linked list using OOP.

```
procedure insert_in_order(data)

  new_node = new node(data) // create a new node with the given data

  current = head // start traversal from the head of the list

  // case 1: the list is empty, insert as the first node
  if current == null then
    head = new_node

  // case 2: new node should be inserted before the head (i.e., smallest element so far)
  elseif new_node.getdata() < current.getdata() then
    new_node.setnext(head)  // point new node to current head
    head = new_node         // update head to new node

  // case 3: inserting somewhere in the middle or end of the list
  else
    // traverse the list to find the correct position for the new node
    while (current.getnext() != null and current.getnext().getdata() < new_node.getdata())
      current = current.getnext()
    endwhile

    // insert the new node by adjusting pointers
    new_node.setnext(current.getnext()) // link new node to the next node in sequence
    current.setnext(new_node)           // link current node to the new node
  endif
endprocedure
```

Unordered linked list insertion

The item is typically inserted at the **head** (for efficiency), or optionally at the end.

Insert at head

This is the **quickest** way to insert into an unordered list because it avoids traversal.

1. **Create a new node** with the given data.
2. Set the new node's next pointer to point to the current head.
3. Update the head of the list to be the new node.

```
procedure insert_unordered(data)
    // create a new node with the given data
    new_node = new node(data)

    // insert at the beginning of the list (most common approach for unordered lists)
    new_node.setnext(head)  // point the new node to the current head
    head = new_node         // update head to point to the new node
endprocedure
```

Insert at end:

1. Create a new node.
2. If the list is empty, set the head to the new node.
3. Otherwise, traverse to the end of the list.
4. Set the last node's next pointer to the new node.

Used when the order of elements doesn't matter—like a simple stack (inserting at head) or queue (inserting at end).

```
procedure insert_unordered(data)
    // create a new node with the given data
    new_node = new node(data)

    // if the list is empty, insert as the first node
    if head == null then
        head = new_node
    else
        // traverse to the end of the list
        current = head
        while current.getnext() != null
            current = current.getnext()
        endwhile

        // insert the new node at the end
        current.setnext(new_node)
    endif
endprocedure
```

Removing a node algorithm:

To delete a node from a linked list, you first need to find it and then adjust the necessary pointers so that the node is no longer part of the list.

To delete a node you must:

1. Traverse the linked list starting from the head to **find the node to be deleted**, keeping track of the previous node.
2. **Update the next pointer** of the previous node to skip the node being deleted and point to the node that comes after it.
3. **Free the memory** occupied by the deleted node and **add it to a pool of free nodes.**

Notice that we don't need to move any nodes to delete an element from the list; all we need to do is to update the pointers.

Before

	Node	Data	Pointer
	1	banana	5
	2	apple	1
	3	orange	4
	4	pear	∅ (Null)
	5	cherry	3
Free Node	6		

After

	Node	Data	Pointer
	1	banana	5
	2	apple	1
Free Node	3		
	4	pear	∅ (Null)
	5	cherry	4
Free Node	6		

Deleting Algorithm

1. **Start at the head** of the list.
2. **Check if the list is empty**:
   - If yes, there's nothing to delete.
3. **Check if the head node contains the target data**:
   - If yes, update the head to point to the next node (effectively removing the head).
4. **If the node to delete is not the head**:
   - Traverse the list to find the **node before** the one with the matching data.
   - While traversing, check that you haven't reached the end (null).
5. **If the target node is found**:
   - Update the next pointer of the current node to skip over the target node.
6. **If the target node is not found**:
   - Do nothing, or optionally report that the item was not found.

The following example shows the code for deleting a node from a linked list using an OOP implementation.

The getter and setter methods have been used as appropriate to access the private attributes of the node object

```
procedure delete(data)

  current = head // start at the head of the list

  if current == null then     // check if the list is empty
    return  // nothing to delete

  // case 1: the head node is to be deleted
  elseif current.get_data() = data then
    head = current.get_next()  // move head to next node

  // case 2: node to delete is somewhere else
  else
    // traverse the list to find the node before the one to delete
    while current.get_next() != null and current.get_next().get_data() != data
      current = current.get_next()
    endwhile
    // if we found the node to delete
    if current.get_next() != null then
      current.set_next(current.get_next().get_next())
    endif
    // else: node with the given data was not found – optionally handle it
  endif
endprocedure
```

Traversing a linked list:

Describe how to traverse a linked list of names to find the number of times that a particular name occurs.

1. First create a variable named total and initialise it with the value 0.
2. Now visit the first node (head of the linked list).
3. If the pointer is NULL, the list is empty.
4. If the pointer is not empty then follow it to go to the next item in the list.
5. Check the name. If it matches the name being, searched increment (add 1 to) the total variable.
6. Continue doing this until the NULL pointer is reached.
7. The total variable will now hold the number of times the particular name has occurred.

```
class linkedlist


   ....
    procedure traverse()
        // set the current node as the head
        current = head
        // repeat until there are no more linked nodes
        while current != null
                print(current.getData())
                current = current.getNext()
        endwhile
    endprocedure

endclass
```

Searching a linked list

- You have to use a linear search with a linked list
- Starting at the start node and use an if statement to check for item is stored in the node.

```
function search(item)
        // set the current node as the head
        current = head
        found = false
        // repeat until there are no more linked nodes
        while current != null and not found:
                if current.getData() == item:
                     found = true
                current = current.getNext()
        endwhile
        return found
 end function
```