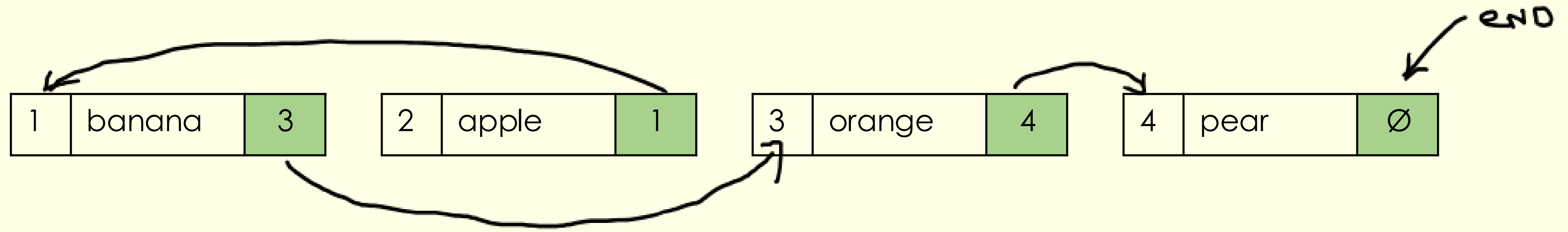
Learning Objectives Lists and linked lists

- Explain how a list may be implemented as either a static (array) or dynamic data structure (linked list)
- Show how items may be added to or deleted from an array
- Describe the linked list data structure
- Show how to create, traverse, add data to and remove data from a linked list

Linked List

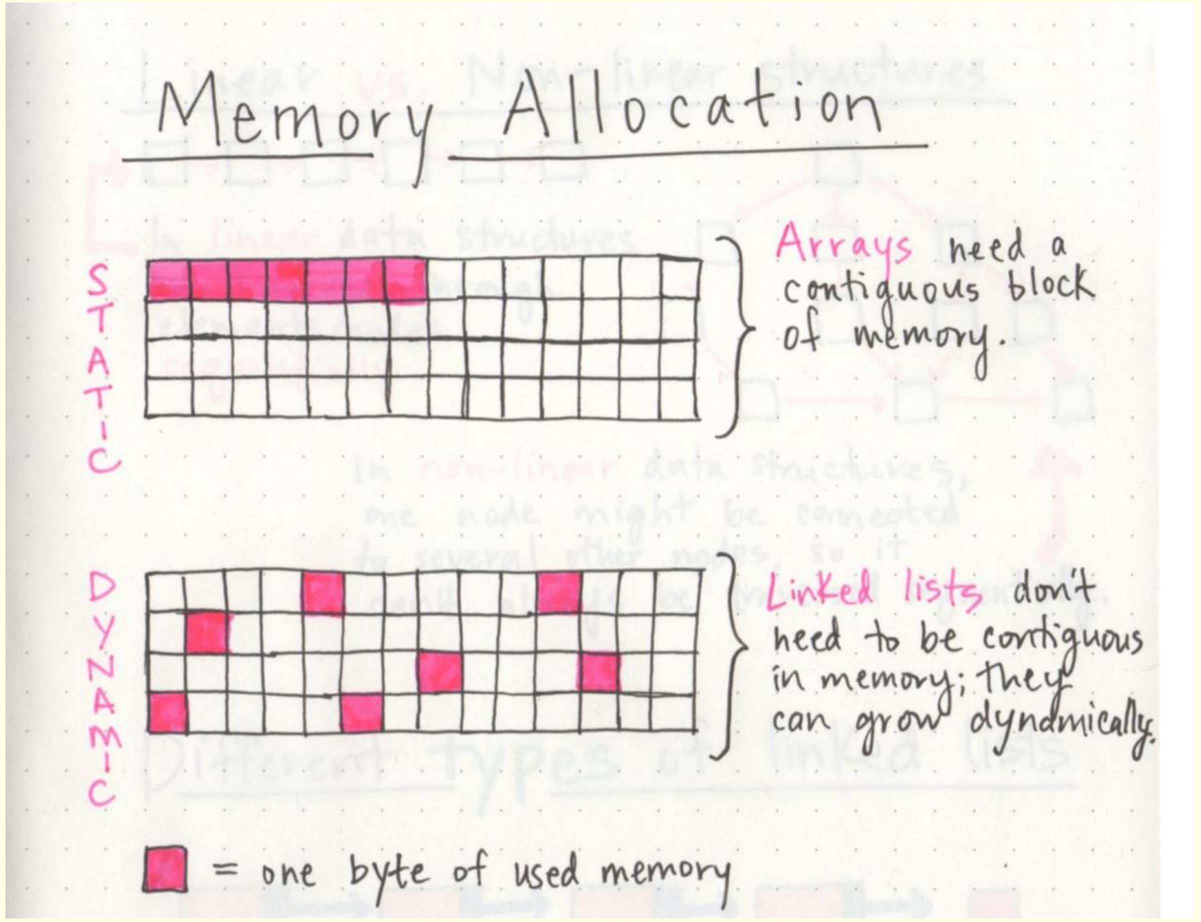
- A linked list is a **dynamic data structure -** grow and shrink during runtime
- A linked list can be **unordered** or it can be **ordered** in some way, such as in alphabetical or numerical order.
- A linked lists uses **pointers** to order the data
- The items which form the sequence are not necessarily 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

Start node 2:



What is the difference between a linked list and array

Linked List	**Array**
Dynamic data structure	Static (fixed size)
Contents of a linked list may not be	Stored together in memory
Easier to insert/delete from linked list,	Room has to be created for the new elements and to create room existing elements have to be shifted.
Only a linear search can be used always starting at the first node.	quicker to search using a binary search
Extra memory space for a pointer is required with each element of the linked list.	



Memory Allocation
STATIC
Arrays need a contiguous block of memory.
DYNAMIC
Linked lists don't need to be contiguous in memory; they can grow dyndmically.
= one byte of used memory

Implementing a linked lists

Pointers are used to link the data in the list in a specific order

This refers to the a nodes address in memory

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

In this example the pointer is being used to order the items alphabetically

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 (Head pointer)

Null Pointer (Ø) is used to specify the end of the list

Free Node is used to specify the next free node in the list

Linked List Diagram

- In the exam you may need to fill in a table to describe how to insert and delete an item in a linked list.

Node	Data	Pointer
1	banan a	3
2	apple	1
3	orange	4
4	pear	∅ (Null)

Implementing a node object

A class is defined to represent a node. The node contains:

- a variable to store the the data value of the element
- a pointer to the next node

```
class Node
    private data
    private next
    public procedure new (pData)
        data = pData # instance variable to store the data (a name)
        next = null # instance variable with address of next node (or -1)
    end procedure
end class
```

Notice how the data of the node is passed to the constructor method when initialising a new node.

However, the pointer to the next node will be set in another method so that the new node is inserted into the correct position in the linked list.

```
new_node = new node(data) #node object
```

Using getter and setters methods in the node class

```
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
```

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)
```

Implementing a linked list class

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_index
    private linkedlist
    private nextFreeIndex

    public procedure new()// constructor method
        linkedlist = [] //index of a dynamic list can be used as pointers
        head_index = -1 #or null
        nextFreeIndex = 0
    endprocedure
endclass

theLinkedList = new Linkedlist()// instantise an empty linked list object
```

Inserting an item (ordered)

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

Before

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

After

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

Linked list in order algorithm

3 Parts to the algorithm:

1. Check if the linked list is empty

2. Check is the new node is less than the head node

3. Otherwise traverse the linked list using the next pointers

Insert item algorithm – In order

Create a new node with the given data.

Check if the list is empty:

- If yes, make the new node the **head** of the list.

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.

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 pointer.
  - Set the current node's next to the new node.

Algorithm

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 = Node(data)
    new_index = nextFreeIndex
    if head_index == -1 then   #checks if the linked list is empty
      new_node.setNext(head_index)
      head_index = new_index # updates the head_index
    else
      if data < linkedlist[head_index].getData() then
        new_node.setNext(head_index)
       head_index = new_index
      else
        current_index = head_index
      # Traverse the linked list
        while self.linkedlist[current_index].getNext() != -1 and linkedlist[linkedlist[current_index].getNext()].getData() < data
          current_index = linkedlist[current_index].getNext() #get the index of the next node
        new_node.setNext(linkedlist[current_index].getNext()) #update new nodes pointer using the previous node's next pointer
        linkedlist[current_index].setNext(new_index) #updating the previous node's next pointer to new node's index
    linkedlist[new_index] = new_node
    nextFreeIndex = nextFreeIndex + 1
```

Index	**Node {Data, Next}**
0	Node{data:1,next:1}
1	Node{data:3,next:2}
2	Node{date:5,next:3}
3	Node{data:9, next: Ø (Null or -1)}
4	

head_index = 0 → start at index 0
nextFreeIndex = 4

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_at_head(data)
    new_node = Node(data)
    new_index = nextFreeIndex
    new_node.setNext(head_index)   # Link new node to current head
    head_index = new_index         # Update head to new node
    linkedlist[new_index] = new_node
```

Insert at end

1. Create a new node.
2. If the list is empty, make it the head.
3. Otherwise, traverse the list until you find the last node (where next is null).
4. Set the next of the last node to point to the new node.
5. Append the new node to the nodes array.

```
procedure insert_at_end(data)
    new_node = Node(data)
    new_index = nextFreeIndex

    if head_index == -1:
        head_index = new_index
        linkedlist[new_index] = new_node
    else:
        current_index = head_index
        while linkedlist[current_index].getNext() != null:
            current_index = .linkedlist[current_index].getNext()


        linkedlist[current_index].setNext(new_index)     #Link last node
to new node
```

Removing an item

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

To delete a node with a specific data value:

1. Start at the head
2. Traverse the list to find the node with that value.
3. Keep track of the **previous node** during traversal.
4. Once found:
   - If it's the **head**, update head index.
   - Otherwise, update previous.next to skip the deleted node.
5. Add index to free nodes list

Delete Algorithm

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_node(value)
    # Case 1: List is empty
    if head_index == -1:
        print("Linked list empty")
    else:
        current_index = head_index
        previous_index = -1
        # Traverse list to find the node with the target value
        while current_index != -1 and linkedlist[current_index].getData() != value:
            previous_index = current_index
            current_index  = linkedlist[current_index].getNext()

        # Case 2: Value not found
        if current_index == -1:
            print("Item not found")
        # Case 3: Deleting the head node
        elif previous_index == -1:
            head_index = linkedlist[current_index].getNext()
        #Case 4: Deleting from middle or end node
        else:
            new_next = linkedlist[current_index].getNext()
            linkedlist[previous_index].setNext(new_next)
```

Traversing a linked List

- Visit each node starting from the head
- Use getData() to read the value
- Use getNext() to move to the next node

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_list()

      current_index = head_index  # Start at the head
      if head_index == -1:
            return
      else:
         while current_index != -1:
            data = linkedlist[current_index].getData()
            print (data)
            current_index = linkedlist[current_index].getNext()
    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.

- Start from head_index
- Traverse using getNext()
- Compare each node's data using getData()
- Return the index or position where the value is found (or indicate not found)

```
procedure search(value)

    current_index = head_index # Start from head

    while current_index != -1:
        if current_index].getData() == value:
            return current_index     #return index position

        current_index = linkedlist[current_index].getNext()

    return "Not found" # Value not found

endprocedure
```