Linked List

definitions are highlighted!

Linked list

- *A data structure consisting of a set of nodes to form a sequence*
    - Each node consists of the data value and a link pointer to one of the other nodes.
        - The data field may consist of several subfields to hold more than 1 piece of data.
- **Purpose:** Link the nodes in some particular order (e.g. alphabetical order of data items)
- The nodes are not necessarily held in contiguous (i.e. physical) data locations, or in the order in which they occur in the sequence.
- Linking process:
    - A pointer variable contains the address of the first node in the linked list.
    - The pointer field in each node contains the address of the next node in the sequence.
    - The pointer field of the last node indicates that there are no further nodes.
        - e.g. Has a value of 0 or -1, a null pointer

Abstract data type (ADT)

- *An ADT describes a data structure based on its behaviour from the point of view of a user, without specifying the implementation.*
    - Focus is on what the structure can do rather than how it is implemented
- A linked list is an ADT, with the following core operations:
    - Initialise — set up an empty linked list
    - Insert (item)

        - e.g. at head/tail, in order
    - Delete (item)
    - Search (item) — look for an item and return its position or reference
    - Traverse — visit, and optionally display, all elements from start to end
        - Done by following the pointer chain from the 'start' node to the null pointer at the end of the list
        - Used while searching, inserting, deleting or counting items
        - Pseudocode:

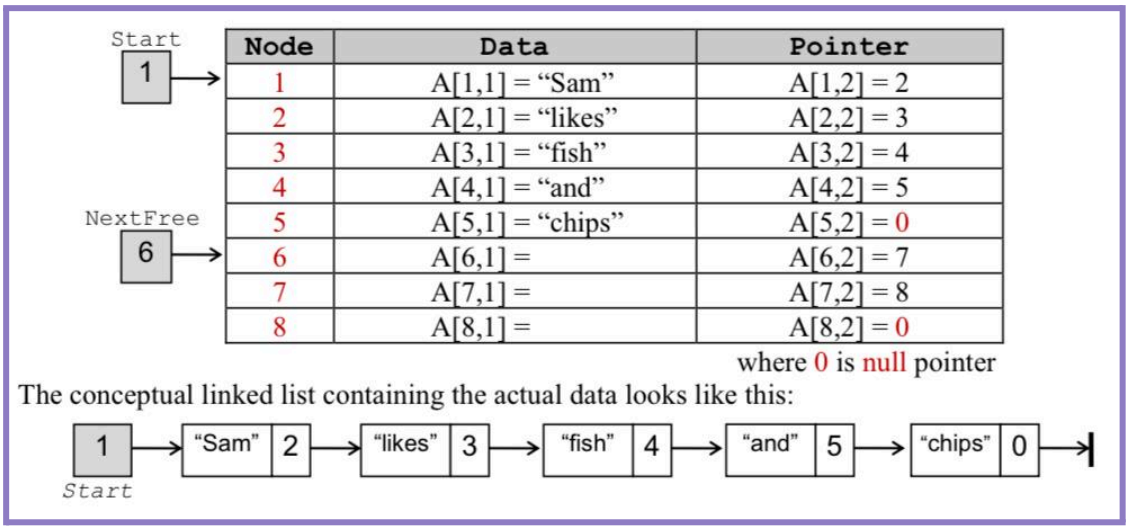
```
current ← Start
WHILE current ≠ NULL
    OUTPUT Data[current]
    current ← Pointer[current]
ENDWHILE
```

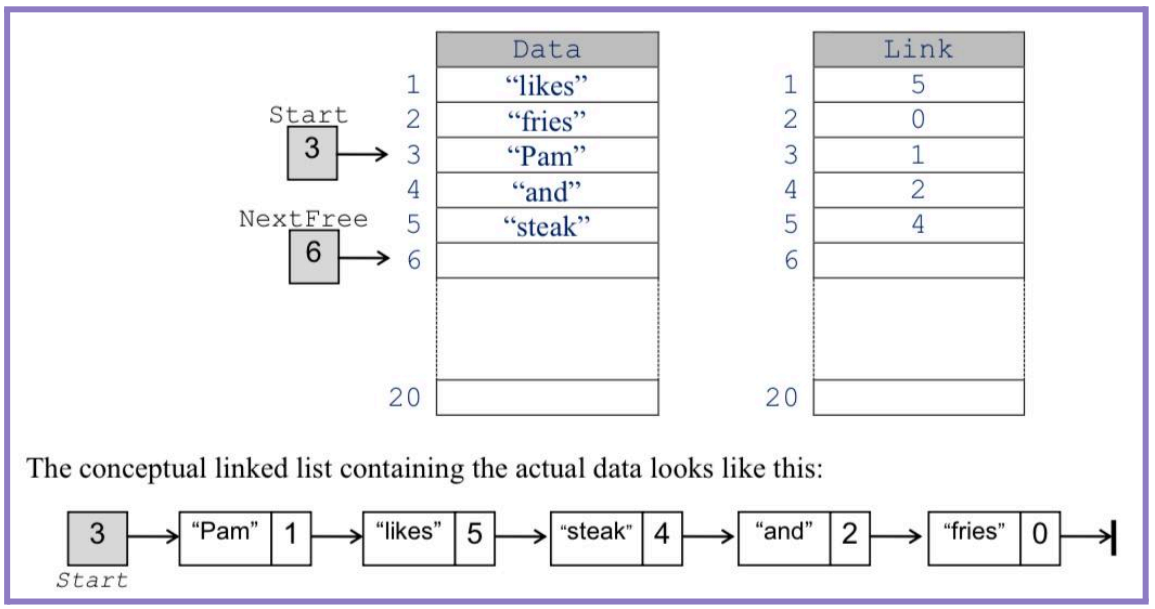
    - IsEmpty — check whether the list has no elements

Free space list

- *Operates by connecting unallocated regions of memory together in a linked list*
    - The pool of memory contains objects of the same size i.e. the nodes.
- To manage the free space for a linked list data structure, 2 linked lists are kept — one for the actual data and one for the free nodes.
- At the point of initialisation, there is just 1 linked list of free nodes, and the linked list for the actual data is empty.
    - So, the 'start' node of the linked list is a null pointer.

Implementation

- Using a single 2D array:
    - The total number of nodes is fixed.
    - Has 2 variables — “Start” (integer of the first node of the linked list) and “NextFree” (index of the next available free node)
    - 

        - Each row in the array represents a node.
- Using two 1D arrays:
    - 

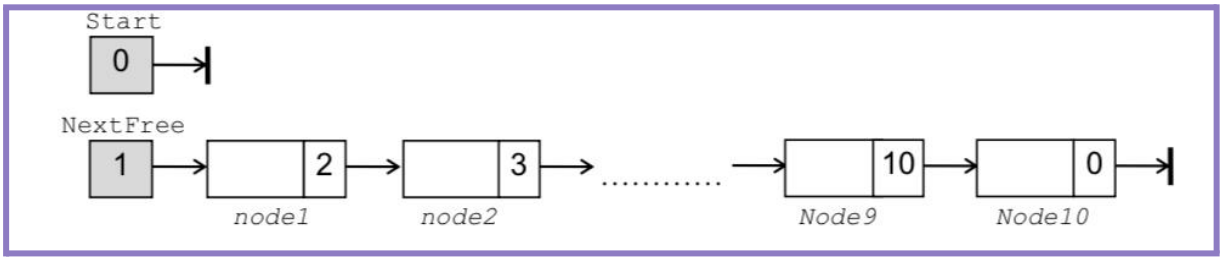
- Using an array of nodes:
    - A linked list could be defined using user-defined types of Node and LinkedList.

Node	
Data:	STRING
Pointer:	INTEGER
Constructor()	
GetData(): STRING	
SetData (item: STRING)	
GetPointer() : INTEGER	
SetPointer (pointer: INTEGER)	

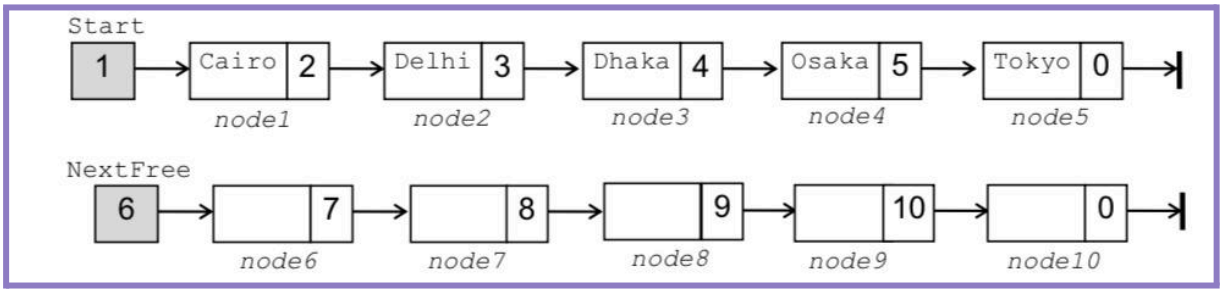
LinkedList	
NodesArray:	ARRAY[1..10] of **Node**
Start:	INTEGER
NextFree:	INTEGER
Constructor()	
Insert (item: STRING)	
Delete (item: STRING)	
Search (item: STRING)	
Display()	

- Initialisation
  - Initial values for the attributes of Node:
    - **Data:** empty string
    - **Pointer:** all nodes have the next index value, and the last node pointer is the null value.
  - Initial values for the attributes of NextFree:
    - Start: null value
    - NextFree: first index

- Linked list and free space list:

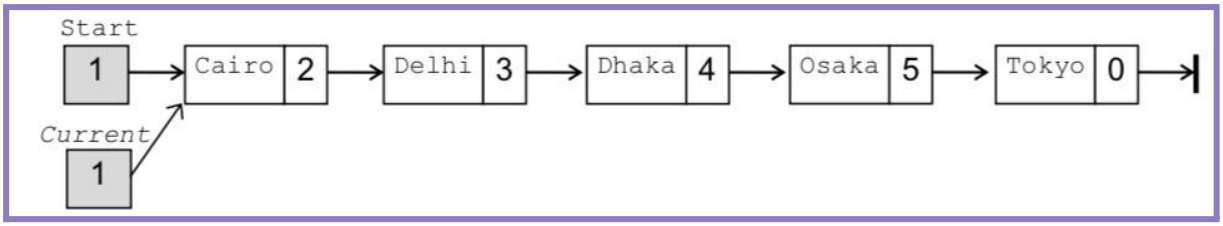


- After adding data into the list:



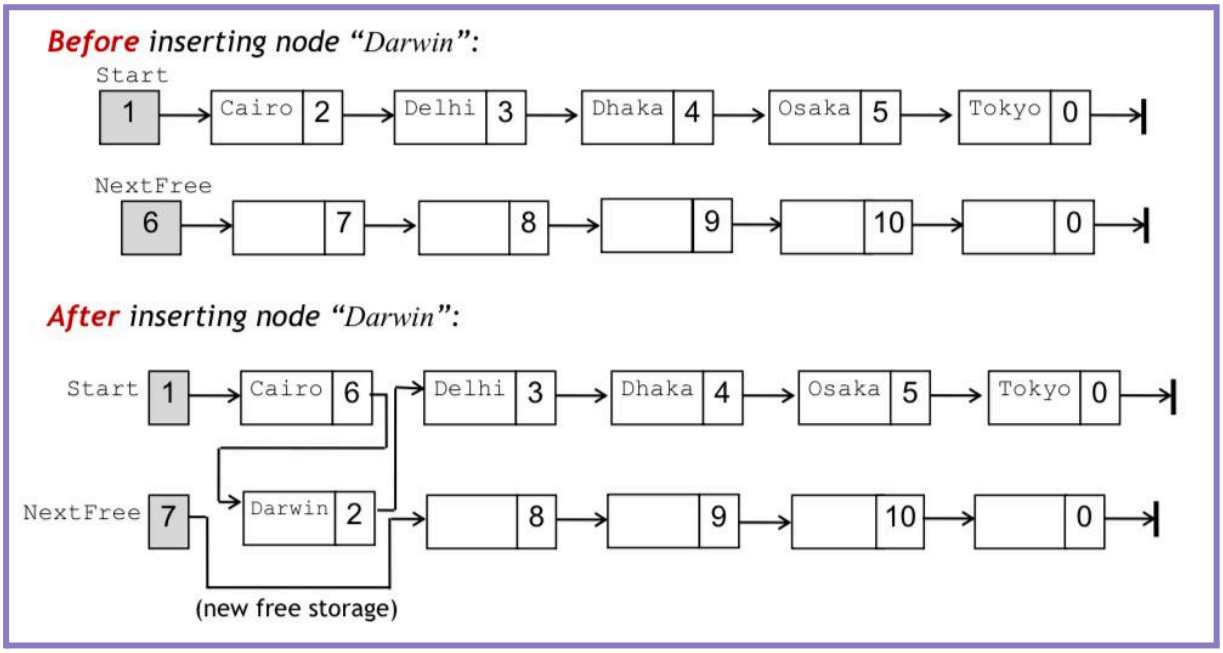
Performing operations on a linked list

- Searching for a node:



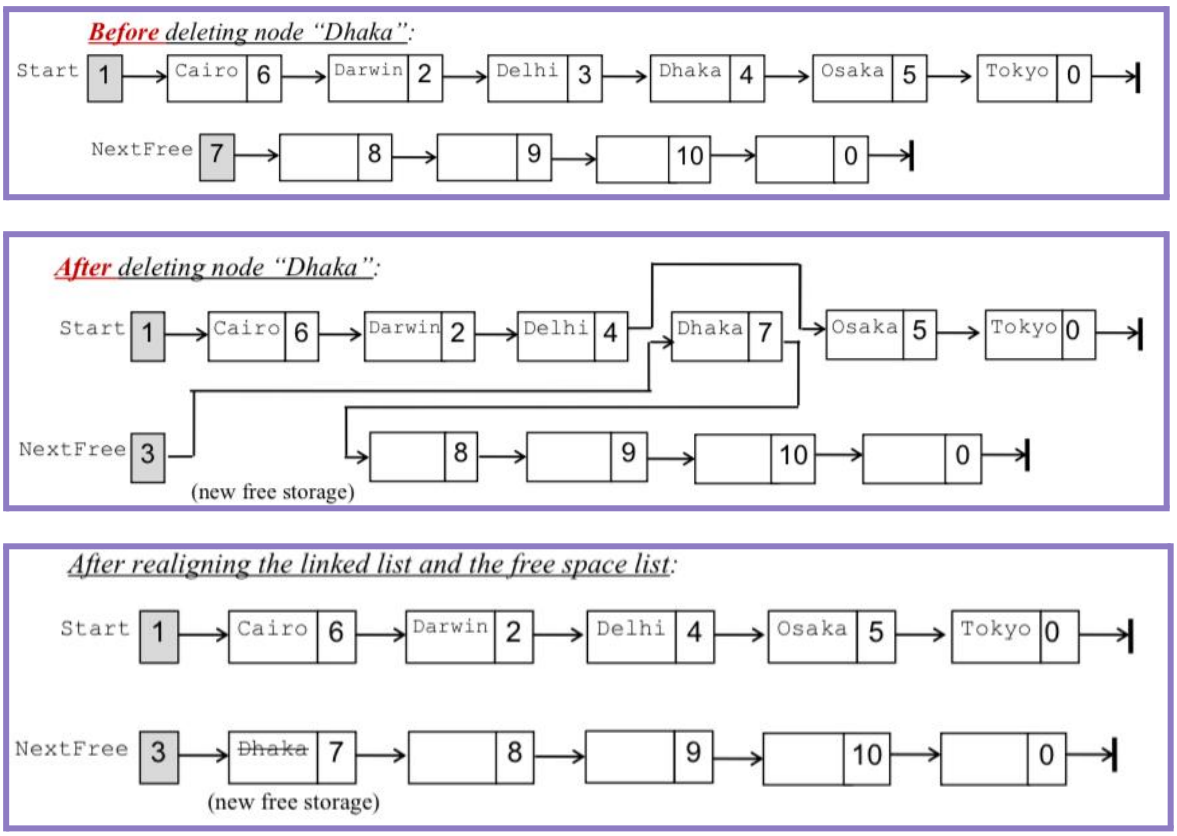
  - **#1:** Let **Current** = **Start** pointer and **data** = required data.
  - **#2:** If **Current** = null, print "Not found" and exit.
  - **#3:** If content of present location = **data**, print data and exit.
  - **#4:** Let **Current** = next pointer of present location and go to step 2.

- Inserting a node at a specific position:



  - **#1:** If NextFree (the pointer for free storage) = null, print "No more space" and exit.
  - **#2:** Search for the position at which the new data is to be inserted. If it cannot be found, print "Error" and exit.
  - **#3:** Save the location of the first free space and let NextFree point to the next free space.

- **#4:** Place the new data item in the saved free space and let it point to the element after the required position.
- **#5:** Change the pointer of the element before the required position to point to the saved element and exit.

- Deleting a node:
  - 

    - **#1:** Search for the node containing data to be deleted. If not found, print "Not found" and exit.
    - **#2:** Save the location of the node to be deleted by assigning a **Temp** pointer to it.
    - **#3:** Change the pointer of the node before **Temp** to point to the node after **Temp**.
    - **#4:** Change the pointer of **Temp** to point to the first free space.
    - **#5:** Change NextFree to point to **Temp** and exit.

Ordered link list

- Stores data in some order (e.g. alphabetical, ascending)
- Notation used:
  - [array identifier][index].name refers to the data held in node[index].
  - [array identifier][index].pointer refers to the value of the pointer in node[index].
- Pseudocode for inserting a node:

```
PROCEDURE Insert(newData)
    IF nextfree IS NULL THEN                    //check if free node exists
        output("No free node available") and RETURN
    ENDIF
    Make[nextfree].name ← newData               //store newData in free node
    IF start IS NULL THEN                       //store into empty linked list
        temp ← Make[nextfree].pointer
        Make[nextfree].pointer ← NULL
        start ← nextfree
        nextfree ← temp
    ELSE
        previous ← NULL
        current ← start
        WHILE current IS NOT NULL               //traverse from start of linked list
            IF newData < Make[current].name THEN
                BREAK                           //Position found between previous and current node
            ENDIF
            previous ← current
            current ← Make[current].pointer
        ENDWHILE
        IF previous IS NULL THEN                // store as first node
            temp ← Make[nextfree].pointer
            Make[nextfree].pointer ← start
            start ← nextfree
            nextfree ← temp
        ELSE                                    // store between previous and current
            temp ← Make[nextfree].pointer
            Make[nextfree].pointer ← current
            Make[previous].pointer ← nextfree
            nextfree ← temp
        ENDIF
    ENDIF
ENDPROCEDURE
```

- Pseudocode for deleting a node:

```
PROCEDURE Delete(Data)
    IF start IS NULL THEN                          //check if linked list is empty
        output("List is empty!") and RETURN
    ENDIF
    previous ← NULL
    current ← start
    WHILE current IS NOT NULL                      //traverse from start of linked list
        IF Data = Make[current].name THEN          //Data to be deleted is found
            read data and BREAK                    //current points to node to be deleted
        ENDIF
        previous ← current
        current ← Make[current].pointer
    ENDWHILE
    IF current IS NULL THEN                        //reach end of linked list, node not found
        output("Node not found") and RETURN
    ENDIF
    IF previous IS NULL                            //first node to be deleted
        temp ← nextfree
        nextfree ← current
        start ← Make[current].pointer
        Make[current].pointer ← temp
    ELSE                                           //non-first node to be deleted
        temp ← nextfree
        nextfree ← current
        Make[previous].pointer ← Make[current].pointer
        Make[current].pointer ← temp
    ENDIF
ENDPROCEDURE
```

Dynamic data structures

- *A collection of data in memory that has the ability to grow or shrink in size during program execution*
- Linked lists are inherently dynamic data structures, as they grow or shrink by allocating and deallocating memory dynamically as needed.
- The built-in array in Python is a dynamic array, not a linked list.
- In OOP in real-life applications (e.g. Python), we do not need to manually create a free space list — memory is automatically allocated when a new object is created and released when it is no longer referenced.

Linked list vs array

Linked lists	Arrays
Nodes are not stored in contiguous memory locations; each node points to the next	Elements are stored in contiguous memory locations
Dynamic in size — grows/shrinks during runtime	Fixed in size — size must be declared in advance
Memory is allocated during runtime, thus size need not be declared	Memory is allocated while compiling (initialisation), so size must be known in advance
Can grow by allocating memory in non-contiguous free space	Resizing may require copying to new memory (for large arrays)
Each node stores data and a pointer, using slightly more memory. **Note:** The pointer takes up a very small amount of memory, so avoid writing this as a disadvantage.	Only stores data, using less memory and eliminating overhead from pointers
Only allows sequential access — requires traversal (time complexity of (O(n)); does not support direct indexing	Supports direct indexing, allowing random access (time complexity of O(1))
Insertion and deletion at known positions is more efficient, especially at the head of the list (time complexity is O(1)).	Insertion and deletion may require shifting elements, making these processes slower (time complexity of O(n)).