9608/42/PRE/O/N/20

Last edited: Anuj Verma, 23:53 11/10/2020

This is all the **program code** for the solution.

The cell below declares a subroutine to insert new elements, and one to traverse the linked list and print out its elements.

Identifier	Data Type	Purpose
`insert()`	`PROCEDURE`	A subroutine to insert a new item at the end of the linked list (unless it is full).
tempPointer (`insert()` scope)	`INTEGER`	A temporary holding for the start pointer while a new item is inserted.
`traverse()`	`PROCEDURE`	A subroutine to traverse the linked list and print out its elements.

In [1]:

```
## Subroutine to insert a new element
def insert(newItem):
    global startPointer, heapStartPointer

    if heapStartPointer != nullPointer:
        tempPointer = startPointer
        startPointer = heapStartPointer
        heapStartPointer = DestinationsPointers[heapStartPointer]
        Destinations[startPointer] = newItem
        DestinationsPointers[startPointer] = tempPointer

    else:
        print("Linked list full, cannot insert.")

## Subroutine to traverse the linked list and print out its elements
def traverse():
    print("")

    global startPointer
    itemPointer = startPointer

    print(Destinations[itemPointer])

    while DestinationsPointers[itemPointer] != nullPointer:
        itemPointer = DestinationsPointers[itemPointer]
        print(Destinations[itemPointer])
```

TASK 3.5

> Write **program code** to declare the linked list, using an array.

Identifier	Data Type	Purpose
startPointer	`INTEGER`	A pointer to the first element of the linked list.
heapStartPointer	`INTEGER`	A pointer to the next free location in the linked list.
nullPointer	`INTEGER`	Constant for a termanating pointer.
Destinations	`ARRAY[0:9] OF STRING`	Data stored in the linked list.
DestinationsPointers	`ARRAY[0:9] OF INTEGER`	Linked list pointers.
Destination	`STRING`	A `for` loop element used while inserting items to the linked list.

In [2]:

```
## Declare constants and variables
startPointer = -1           # INTEGER
heapStartPointer = 0        # INTEGER
nullPointer = -1            # INTEGER CONSTANT

## Arrays of the linked list itself
Destinations = [None for i in range(10)]                # ARRAY[1:10] OF VOID
DestinationsPointers = [(i + 1) for i in range(10)]     # ARRAY[1:10] OF INTEGER
DestinationsPointers[9] = -1

## Insert the given destinations one by one
for Destination in ["Paris, France", "Rome, Italy", "New Delhi, India", "Kuala Lumpur,
 Malaysia", "Wellington, New Zealand", "New York, USA"]:
    insert(Destination)

## Traverse the linked list and print out each element to test
traverse()
```

```
New York, USA
Wellington, New Zealand
Kuala Lumpur, Malaysia
New Delhi, India
Rome, Italy
Paris, France
```

TASK 3.6

> Extend your **program code** by writing a subroutine that adds a new destination to the end of your linked list.

We already implemented this routine and used it to initiaize the linked list. But a copy along with the identifier table is given here.

Identifier	Data Type	Purpose
`insert()`	`PROCEDURE`	A subroutine to insert a new item at the end of the linked list (unless it is full).
tempPointer (`insert()` scope)	`INTEGER`	A temporary holding for the start pointer while a new item is inserted.

In [3]:

```
def insert(newItem):
    global startPointer, heapStartPointer

    if heapStartPointer != nullPointer:
        tempPointer = startPointer
        startPointer = heapStartPointer
        heapStartPointer = DestinationsPointers[heapStartPointer]
        Destinations[startPointer] = newItem
        DestinationsPointers[startPointer] = tempPointer

    else:
        print("Linked list full, cannot insert.")

## Test the routine by adding Reykjavik, Iceland as in TASK 3.3
insert("Reykjavik, Iceland")

## Traverse the linked list and print out each element to test
traverse()
```

```
Reykjavik, Iceland
New York, USA
Wellington, New Zealand
Kuala Lumpur, Malaysia
New Delhi, India
Rome, Italy
Paris, France
```

TASK 3.7

> Extend your **program code** by writing a subroutine to delete the destination node entered by the user from the linked list.

Identifier	Data Type	Purpose
`delete()`	`PROCEDURE`	Delete the given element from the linked list.
index (`delete()` scope)	`INTEGER`	The pointer to the element to be deleted.
oldIndex (`delete()` scope)	`INTEGER`	Pointer to the next element.
tempPointer (`delete()` scope)	`INTEGER`	A temporary holding for the start pointer while a new item is inserted.

In [4]:

```
def delete(itemDelete):
    global startPointer, heapStartPointer

    if startPointer == nullPointer:
        print("Linked list empty")

    else:
        index = startPointer

        while (Destinations[index] != itemDelete) and (index != nullPointer):
            oldIndex = index
            index = DestinationsPointers[index]

        if index == nullPointer:
            print("Item", itemDelete, "not found")

        else:
            Destinations[index] = None
            tempPointer = DestinationsPointers[index]
            DestinationsPointers[index] = heapStartPointer
            heapStartPointer = index
            DestinationsPointers[oldIndex] = tempPointer

## Delete Kuala Lumpur, Malaysia to test
delete("Kuala Lumpur, Malaysia")

## Traverse the linked list and print out each element to test
traverse()
```

```
Reykjavik, Iceland
New York, USA
Wellington, New Zealand
New Delhi, India
Rome, Italy
Paris, France
```

TASK 3.8

> Discuss other linked list operations that could be implemented.
>
> Write **program code** to implement the operation(s) you discuss.

We already wrote a routine to traverse the linked list and print out elements, but we can write two additional routines:

- `find()` to find the given element in the linked list.
- `update()` to change the value of an element.

Identifier	Data Type	Purpose
`find()`	`FUNCTION`	Fuction returns the index of the item passed as the argument, or `-1` if it could not be found.
index (`find()` scope)	`INTEGER`	The temporary variable for the index of the element if it was found.
itemToFind (`find()` scope)	`STRING`	The value of the item to be searched for, passed as a parameter.
`update()`	`PROCEDURE`	Procedure replaces `itemToUpdate` with `newItem` if `itemToUpdate` was found, or throws an error message otherwise.
index (`update()` scope)	`INTEGER`	The temporary variable for the index of the element if it was found.
itemToUpdate (`update()` scope)	`STRING`	The value of the item to be searched for, passed as a parameter.
newItem (`update()` scope)	`STRING`	The new value of `itemToUpdate` , passed as a parameter.

In [5]:

```python
## Procedure to find the given item in the linked list
def find(itemToFind):
    print("")

    global startPointer
    itemPointer = startPointer
    index = -1

    if itemToFind == Destinations[itemPointer]:
        index = itemPointer

    while (DestinationsPointers[itemPointer] != nullPointer) and (index == -1):
        itemPointer = DestinationsPointers[itemPointer]

        if itemToFind == Destinations[itemPointer]:
            index = itemPointer

    if index == -1:
        print(itemToFind + " could not be found in the list.")

    return index

## Procedure to update the given item in the linked list
def update(itemToUpdate, newItem):
    index = find(itemToUpdate)

    if index != -1:
        Destinations[index] = newItem

## Print out the list before any updates
traverse()

## Look for New Delhi, India and output the index if it was found
print ("New Delhi, India was found at index " + str(find("New Delhi, India")))

## Look for Bangalore, India and output the index if it was found
find("Bngalore, India")

## Look for New Delhi, India and update it to Bangalore, India if it was found
update("New Delhi, India", "Bangalore, India")

## Print out the list after all updates
traverse()
```

```
Reykjavik, Iceland
New York, USA
Wellington, New Zealand
New Delhi, India
Rome, Italy
Paris, France

New Delhi, India was found at index 2

Bngalore, India could not be found in the list.


Reykjavik, Iceland
New York, USA
Wellington, New Zealand
Bangalore, India
Rome, Italy
Paris, France
```