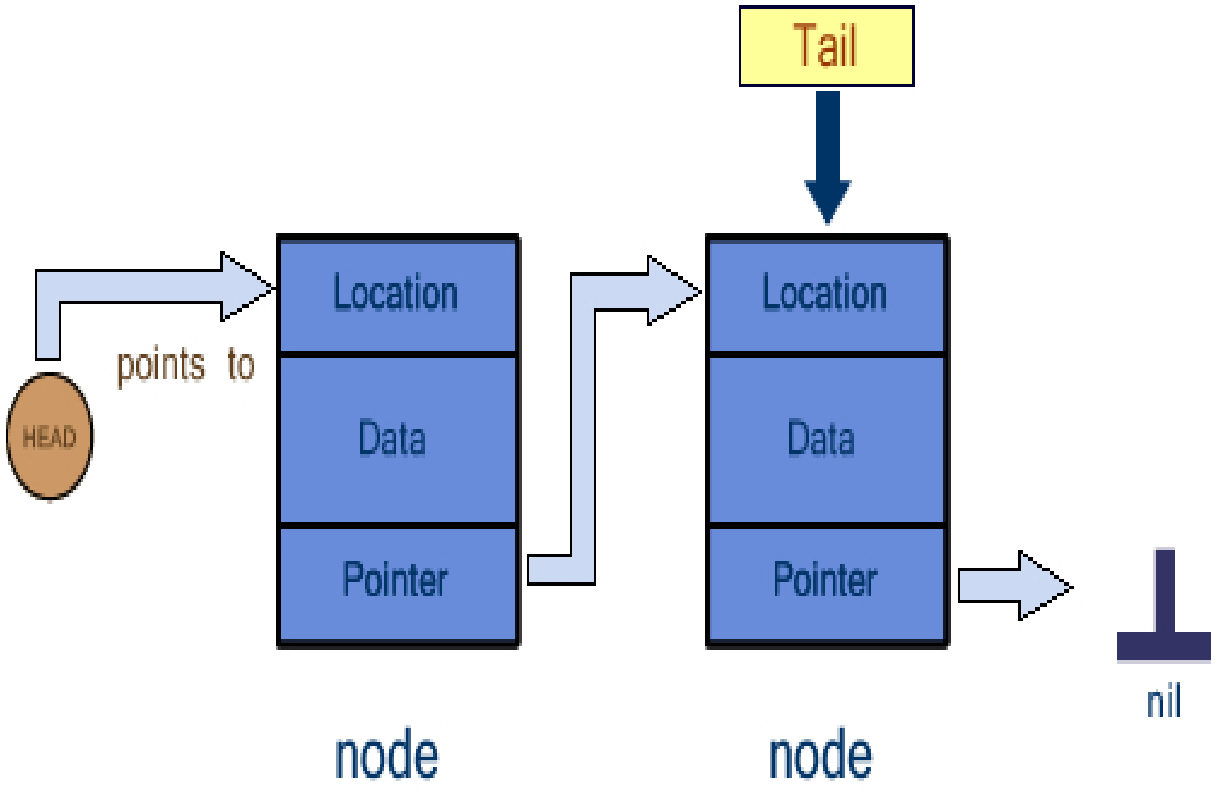
Lesson 16: Article 4: Intro to Linked Lists

A linked list is a powerful tool that enables us to store a list of data items. Regardless of the type of data we are going to store, linked lists may store data *infinitely* since its size is expandable.

Working with Linked Lists

Linked lists make vast use of pointers since each node points to another in memory to keep track where the next node is found and therefore the whole list remains intact. Let's talk about the linked list in general. An element of a linked list is the node. The node is what carries the data and contains a pointer to the next node in the list. However we must take into consideration the first and last nodes. The linked list maybe empty and in this case we've got the head of the list pointing to nil. Make sure that the last node (i.e. the tail) always points to nil to indicate the end of the list. The first node maybe the first and last node of the list.

This can be illustrated as follows:



As you can see, the first node points to the second node and if we are to add another node to the end of the list, we will simply make the last node to point to the new node and the new node will point to nil. The last added node always should point to nil. If we were to insert a node in the middle of the list, we take the location of the node pointed to by the node after which we will insert our new node. Then we make the previous node pointed to the new inserted node, and using the pointed location of the previous node we make the new node pointing to the node that was previously pointed by the other node.

Implementing a Linked List

A linked list may store data of one type. The type to be used should be chosen before the linked list is implemented. We won't use linked list to store integer numbers or just strings/characters or any other basic data types. Linked lists are generally used to store records and other complex user-defined data types.

We will now start to implement a linked list. It will be done step by step in order to let you comprehend and learn the way a linked list is implemented. In order to demonstrate the use of linked list, we should use a sample data type and it will be the following user-defined data type:

```
Type
          StudRec = Record;
                    Name, Surname : String[40];
                    ID, Age : Integer;
                    Gender : Char;
          End;
```

As you can see, we will going to use a student record as our record structure to be stored in a linked list. Next we will start making the components of the linked list: the node. The node will be defined in just like we have done for defining the student record.

```
Type
          TStudRec = Record
                    Name, Surname : String[40];
                    ID, Age : Integer;
                    Gender : Char;
          End;

          TNodePtr = ^TNode;
          TNode = Record
                    StudRec : TStudRec;
                    NodePtr : TNodePtr;
          End;
```

We have just defined the node of a linked list. Next we define our head which is the pointer to the first node and the tail which points to the last node. The head helps us find our first

node in the list whereas the tail helps us to keep track of the last node in the list. Both are simple pointers to a node (in our case TNodePtr).

```
Type
          TStudRec = Record
                    Name, Surname : String[40];
                    ID, Age : Integer;
                    Gender : Char;
          End;

          TNodePtr = ^TNode;
          TNode = Record
                    StudRec : TStudRec;
                    NodePtr : TNodePtr;
          End;

Var
          Head, Tail : TNodePtr;
          SampRec : TStudRec;
```

The *SampRec* will be used to pass it as an argument to linked list routines. We will make sample records, store them individually in *SampRec* and pass them as arguments to procedures. Now we do the initialisation part of the linked list to be called as the first routine before any other calls to linked-list related functions.

```
Procedure InitLinkedList;
Begin
          Head := nil;
          Tail := Head;
End;
```

Next is the node addition procedure. It will accept a student record and add the new node to the end of the list.

```
Procedure AddNode(StudRec : TStudRec);
Var
          Node : TNode;

Begin
          Node.StudRec := StudRec;
          New(Node.NodePtr);
          If Head = nil Then
          Begin
                    New(Head);
                    New(Tail);
                    Head^ := Node;
          End Else
          Begin
                    Tail^.NodePtr^ := Node;
          End;
          Tail^ := Node;
End;
```

Let us see clearly what is really happening in AddNode(). This module accepts StudRec as argument i.e. the record to be added at the end of our list. We first test if the Head is nil. If it as such, then it means that our list is still empty and what we should do is create a new Head

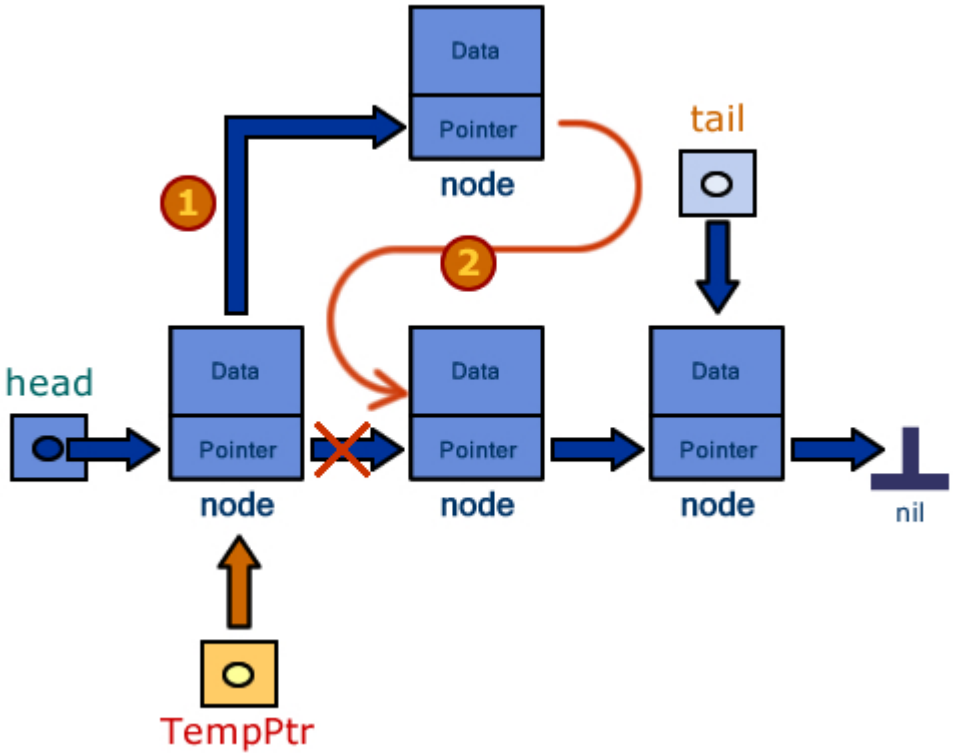
and Tail and set Head to point to the first node. Otherwise, if the linked list is not full, the tail dereferences to the last node's pointer and set it to point to the new node as show in the line,

Tail^.NodePtr^ := Node;

Finally, we should update the Tail and set it to point to the most recent added node. We have implemented the most fundamental routine of the linked list - that of adding a new node at the end of the list. However, having a linked list that just adds a new node is too poor. We have to strengthen it with other features.

Let's now consider how to insert nodes in the linked list since it is highly probable that programs require to insert records in between a list of records. When inserting records, we must know after which node we should insert the new node. To insert the record, we need a temporary pointer to traverse the linked list and find the node after which the new node will be inserted. Our task is to find this node, and change the pointers concerned accordingly to properly insert the new node. When the TempPtr (our temporary pointer) finds the node, (don't let the english language confuse you ;) we insert the new node by first [1] set the node pointed by the temporary node to point to the new node and [2] then set the new node's pointer to point to the node pointed to by the node that is pointed to by our temporary pointer.

It is better to see it pictorially as follows:



Programatically, this is implemented as follows:

```
Procedure InsertRecordByIndex(Index : Integer; StudRec : TStudRec);
Var
          i : Integer;
```

```
          TempPtr : TNodePtr;
          Node, TempNode : TNode;
          Done : Boolean;


Begin
          Done := False;

          If Head = nil Then
                    Exit;

          i := 0;
          TempPtr := Head;
          Node.StudRec := StudRec;
          New(Node.NodePtr);

          If (Index = 0) Then
          Begin
                    TempNode := Head^;
                    Head^ := Node;
                    Node.NodePtr^ := TempNode;
                    Done := True;
          End;

          If not Done Then
                    While (i < Index-1) do
                    Begin
                              If (TempPtr^.NodePtr^.NodePtr = nil) then
                              Begin
                              Done := True;
                              Break;
                              End;

                              i := i + 1;
                              TempPtr := TempPtr^.NodePtr;
                    End;

                    If not Done Then
                    Begin
                              TempNode := TempPtr^.NodePtr^;
                              TempPtr^.NodePtr^ := Node;
                              Node.NodePtr^ := TempNode;
                    End;
End;
```

This procedure inserts a record after the *n*th node (using the index). The state variable*Done* is used to indicate whether the insert operation is performed or not. If the index is 0, we peform the insert operation straight away since we know that the node after which we insert the new node is the head. We then skip the traversing and we're done. But if the index does not refer to the head, we start counting up a counter until it matches the index. If the index exceeds the number of present nodes in the list, the counting halts and the insertion becomes an addition of a new node at the end of the list. The important part which is the insertion of the node is at the end of the routine. A temporary node is assigned with the node that is pointed to by the temporary pointer. The node pointer that is pointed to by the temporary pointer is set to point to the new node and the pointer of the new node is set to point to the temporary node, affecting the insertion of the new node.

The previous insertion is performed after the node indicated by the index. Now we will do another insertion that requires that the new node is inserted after a particular node having some ID. The only difference is that instead of counting up to *n*, we will compare the temporary node student ID with the given ID and insert the node if these ID's equal each other.

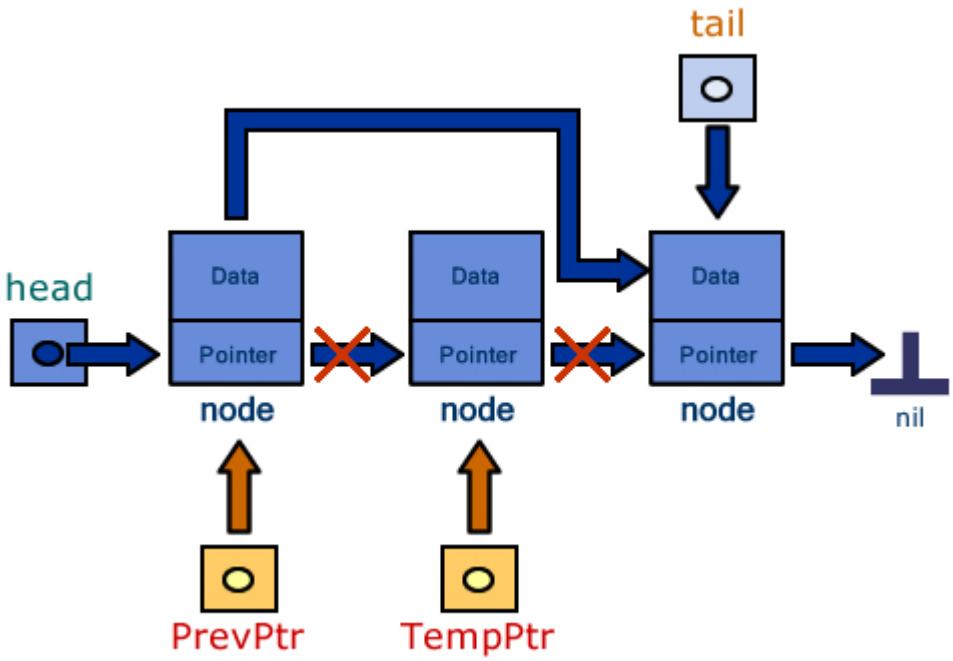
```
Procedure InsertRecordByID(ID : Integer; StudRec : TStudRec);
Var
          TempPtr : TNodePtr;
          Node, TempNode : TNode;
          Done : Boolean;

Begin
          Done := False;
          If Head = nil Then
                    Exit;
          TempPtr := Head;
          Node.StudRec := StudRec;
          New(Node.NodePtr);

          If (TempPtr^.StudRec.ID = ID) Then
          Begin
                    TempNode := Head^;
                    Head^ := Node;
                    Node.NodePtr^ := TempNode;
                    Done := True;
          End;

          While not Done do
          Begin
                    If (TempPtr^.StudRec.ID = ID) Then
                              Break;
                    If (TempPtr^.NodePtr^.NodePtr = nil) Then
                    Begin
                              Done := True;
                              Break;
                    End;
                    TempPtr := TempPtr^.NodePtr;
          End;

          If not Done Then
          Begin
                    TempNode := TempPtr^.NodePtr^;
                    Node.NodePtr^ := TempNode;
                    TempPtr^.NodePtr^ := Node;
          End;
End;
```

We have included a powerful feature in our linked list: inserting a node in the list. In order to have a fully working featured linked list, we should also provide a delete node feature.

When deleting a node from a linked list, we require to traversing pointers, one is the previous node pointer (*PrevPtr*) and the other is the ordinary temporary pointer (*TempPtr*). The previous pointer points to the node behind the one pointed to by the temporary pointer, except in one exceptional case: when the temporary pointer is pointing to the head, the previous pointer also points to the head. The node identified as the deleted one will be removed from the linked list using another pointer manipulation. The temporary pointer will find the one that is to be deleted (in our case we

will use the student ID as the key) by matching the key ID with student ID. If they match the following simple operation occurs:

Set the previous pointer to point the node that is pointed by the node to be deleted (the node to be deleted will be pointed to by the temporary pointer). Pictorially, this will be illustrated as follows:



Here is the implementation of the delete node:

```
Procedure DeleteNodeWithID(ID : Integer);
Var
          TempPtr, PrevPtr : TNodePtr;
          Done : Boolean;

Begin
          Done := False;
          If Head = nil Then
                    Exit;
          PrevPtr := Head;
          TempPtr := Head;

          While True do
          Begin
                    If (TempPtr^.StudRec.ID = ID) Then
                              Break;
                    If (TempPtr^.NodePtr^.NodePtr = nil) Then
                    Begin
                              Done := True;
                              Break;
                    End;
                    PrevPtr := TempPtr;
                    TempPtr := TempPtr^.NodePtr;
          End;

          If not Done Then
          Begin
                    If TempPtr = Head Then
                              Head := Head^.NodePtr
```

```
                Else
                Begin
                        PrevPtr^.NodePtr := TempPtr^.NodePtr;
                End;
        End;
End;
```

And that's all of our story. We have implemented a simple Linked List that is capable of adding a node at the end of the list, inserts a node in between the list, and delete a node from the list.

The whole compact source code

```
Program linkedlists;
Uses Crt;

Type
     TStudRec = Record
      Name, Surname : String;
      Id, Age : Integer;
      Gender : Char;
     End;

     TNodePtr = ^TNode;
     TNode = Record
      StudRec : TStudRec;
      NodePtr : TNodePtr;
     End;

Var
    Head, Tail : TNodePtr;
    SampRec : TStudRec;

Procedure InitLL;
Begin
 Head := nil;
 Tail := Head;
End;

Procedure AddRecord(StudRec : TStudRec);
Var
    Node : TNode;

Begin
 Node.StudRec := StudRec;
 New(Node.NodePtr);
 If Head = nil then
  Begin
   New(Head);
   New(Tail);
   Head^ := Node;
  End Else
  Begin
   Tail^.NodePtr^ := Node;
  End;
 Tail^ := Node;
End;
```

```
Procedure InsertRecordByIndex(Index : Integer; StudRec : TStudRec);
Var
    i : Integer;
    TempPtr : TNodePtr;
    Node, TempNode : TNode;
    Done : Boolean;

Begin
 Done := False;
 if Head = nil then
  Exit;
 i := 0;
 TempPtr := Head;
 Node.StudRec := StudRec;
 New(Node.NodePtr);
 If (Index = 0) then
  Begin
   TempNode := Head^;
   Head^ := Node;
   Node.NodePtr^ := TempNode;
   Done := True;
  End;
 If not Done then
  While (i < Index-1) do
   Begin
    If (TempPtr^.NodePtr^.NodePtr = nil) then
     Begin
      Done := True;
      Break;
     End;
    i := i + 1;
    TempPtr := TempPtr^.NodePtr;
   End;
 If not Done then
  Begin
   TempNode := TempPtr^.NodePtr^;
   TempPtr^.NodePtr^ := Node;
   Node.NodePtr^ := TempNode;
  End;
End;

Procedure InsertRecordByID(ID : Integer; StudRec : TStudRec);
Var
    TempPtr : TNodePtr;
    Node, TempNode : TNode;
    Done : Boolean;

Begin
 Done := False;
 if Head = nil then
  Exit;
 TempPtr := Head;
 Node.StudRec := StudRec;
 New(Node.NodePtr);
 If (TempPtr^.StudRec.ID = ID) then
  Begin
   TempNode := Head^;
   Head^ := Node;
   Node.NodePtr^ := TempNode;
   Done := True;
  End;
```

```
 While not Done do
  Begin
   If (TempPtr^.StudRec.ID = ID) then
    Break;
   If (TempPtr^.NodePtr^.NodePtr = nil) then
    Begin
     Done := True;
     Break;
    End;
   TempPtr := TempPtr^.NodePtr;
  End;
 If not Done then
  Begin
   TempNode := TempPtr^.NodePtr^;
   Node.NodePtr^ := TempNode;
   TempPtr^.NodePtr^ := Node;
  End;
End;

Procedure DeleteNodeWithID(ID : Integer);
Var
    TempPtr, PrevPtr : TNodePtr;
    Done : Boolean;

Begin
 Done := False;
 if Head = nil then
  Exit;
 PrevPtr := Head;
 TempPtr := Head;
 While True do
  Begin
   If (TempPtr^.StudRec.ID = ID) then
    Break;
   If (TempPtr^.NodePtr^.NodePtr = nil) then
    Begin
     Done := True;
     Break;
    End;
   PrevPtr := TempPtr;
   TempPtr := TempPtr^.NodePtr;
  End;
 If not Done then
  Begin
   If TempPtr = Head then
    Head := Head^.NodePtr
   Else
   Begin
    PrevPtr^.NodePtr := TempPtr^.NodePtr;
   End;
  End;
End;

Procedure PrintAll(Head : TNodePtr);
Var
    Node : TNodePtr;

Begin
 New(Node);
 Node := Head;
 While Node^.NodePtr <> nil do
```

```
  Begin
   Writeln('================');
   Writeln(Node^.StudRec.Name);
   Writeln(Node^.StudRec.Surname);
   Writeln(Node^.StudRec.Id);
   Writeln(Node^.StudRec.Age);
   Writeln(Node^.StudRec.Gender);
   Writeln('================');
   Node := Node^.NodePtr;
  End;
 Writeln('Done Printing.');
End;

Procedure AssignRecord(StudRec : TStudRec; Name, Surname : String; ID, Age
: Integer; Gender : Char);
Begin
 SampRec.Name    := Name;
 SampRec.Surname := Surname;
 SampRec.Age     := Age;
 SampRec.Id      := Id;
 SampRec.Gender  := Gender;
End;

Begin
 ClrScr;
 InitLL;
 AssignRecord(SampRec, 'Victor', 'Saliba', 19, 12345, 'M');
 AddRecord(SampRec);
 AssignRecord(SampRec, 'Mario', 'Petrack', 42, 00011, 'M');
 AddRecord(SampRec);
 AssignRecord(SampRec, 'Mary', 'Kels', 22, 20211, 'F');
 AddRecord(SampRec);
 AssignRecord(SampRec, 'Ken', 'Bolimpart', 19, 04148, 'M');
 AddRecord(SampRec);
 AssignRecord(SampRec, 'Kelly', 'Becks', 16, 04148, 'F');
 InsertRecordByID(00011, SampRec);
 DeleteNodeWithID(20211);
 Writeln('Done.');
 PrintAll(Head);
 Readln;
End.
```