9. Record Maintenance using Linked Lists

1. Introduction

In Lecture 7 and Lecture 8 we saw how a dynamic linked list structure could be used as a basis with which to build two further ADTs to which a specific ordering and access protocol pertained: the ADTs **Stack** and **Queue**. There are, of course, ways of realising these particular structures other than through the use of a dynamic linked list structure.

In this lecture we look at the use of linked lists to assist in maintaining a collections of information, or **database records** so as to admit the following actions:

1. Maintain the collection so that records are ordered according the value of a particular field (or **key**) attribute.
2. Allow new records to be inserted into the collection, in their 'correct' position within the current collection.
3. Allow 'inactive' records to be removed from the collection.
4. Allow the details held within a specific record to be reported to on request, when its identifying key attribute is supplied.
5. Ensure (when a new record is added) that its identifying key is unique, i.e. that no other record held in the collection has the value held in its key attribute.

In the next part of COMP102, you will meet more general and powerful systems that are used to provide facilities to construct and interrogate **Databases**. The properties and design theory of such systems form a major part of Computer Science and are important both in the context of modern 'practical' commercial and industrial environments and in the nature of the theoretical issues that effective realisation raises: methods for very-large (10^6+ records) data organisation so as to facilitate 'rapid' access to specific information; formalisms to allow arbitrarily complex **queries** to be made, e.g. in the student record example described earlier, one may wish to see all the records corresponding to students who are registered for a particular module and who obtained a mark within a given range on some pre-requisite module. In such cases not only is the 'expressivity' of the **query language** an important issue, but also the ease with which a request can be translated to identify what is required, and the speed with which the actual records can be delivered. The Part II module, COMP207, which a number of you will study next year, presents a much more extensive treatment of the design and development of Database Systems.

In terms of the mechanisms just outlined, the set of methods described in the current lecture effect only an extremely naive and simplistic approach, suitable merely in the context of very basic record maintenance operations. In Lecture 10, an alternative organisational structure that addresses one of the primary weaknesses of a linked list scheme is considered. Even this structure, however, is quite some distance from a practicable database system facility.

Our aims in this lecture are twofold:

1. To illustrate some of the elements that comprise a typical database systems, e.g. records, their attributes, the idea of a **primary** key.
2. To describe how an, albeit very rudimentary, system built around linked list structures can be used to maintain and manipulate such collections.

Again it is worth stressing the earlier *caveat* to the effect that the disadvantages of a basic linked list form are of such a nature that it would be unlikely for this structure to be used in a large-scale 'real' application.

2. A 'Toy' Database Example: Video Collections

[Important Note: A number of the themes introduced in this section will be developed more fully in the next part of **COMP102**, and yet further in COMP207. For our present purposes, we are mainly concerned with describing the concept of capturing information regarding a particular item in a collection of items by defining a set of *attributes* associated with it. We will not address questions regarding issues such as: is the set of attributes 'complete' (e.g. can all the information that should be recoverable actually be obtained from a given record); are some attributes 'redundant' (in the sense that their values could be deduced from other information that is already held), etc. This is not because such concerns are unimportant (in fact, quite the contrary is true), but because there is a rich and extensive theory and design methodology that deals with precisely such questions and other related issues. The development of these ideas is dealt with later on (in this module and in COMP207).**]**

Rather than simply present a list of the methods required and a description of how these may be realised in an abstract setting, we shall present these in the context of a specific application, i.e. in terms of maintaining information on a 'real' data set.

The application we use is similar to the standard 'Library Collection' example, that is often used as an elementary 'non-trivial database' example in introductory Database Design textbooks. As a variation we consider the problem of tracking information regarding cinema films that might occur in an individual or public collection on video format.

Below are some data one might wish to record for a given film:

1. An **Identifying number** for the film, cf. systems such as ISBN/ISSN for books and journals.
2. The film **title**.
3. Original **language**.
4. Running **time**.
5. Video **format** (VHS-PAL (U.K. format), BetaMax (yes, it still exists!), VHS-NTSC (U.S. format) etc.)
6. BBFC **Certificate** (and/or other minimum viewing age classification)
7. **Director**.
8. **Other** information.

(Of course this is far from being complete).

For example, an 20 record fragment of a particular collection might be presented in the format given in

Figure 9.1., below:

ID	Title	Language	Time	Format	Cert.	Director	Other
01	Ai No Corrida	Japanese	96	VHS-NTSC	NC-17	Oshima	Japanese 'art' film: refused BBFC cert.
02	Le Mepris	French	104	VHS-PAL	15	Godard	-
03	Black Rain	English	125	VHS-PAL	18	Ridley Scott	-
04	The Offence	English	120	VHS-PAL	18	Sydney Lumet	-
05	M	German	94	VHS-PAL	15	Fritz Lang	B/W
06	Showgirls	English	115	VHS-PAL	18	P. Verhoeven	Yes, it is *really bad*
07	Dirty Weekend	English	100	VHS-PAL	18	M. Winner	but *not* as bad as **this**.
08	Deux ou trois choses que je sais d'elle	French	105	VHS-PAL	15	Godard	-
09	Krótki film o milosci	Polish	90	VHS-PAL	15	Kieslowski	Cinema version from television *Dekalog*
10	Novecento	Italian	300	VHS-PAL	18	Bertolucci	-
11	On the Waterfront	English	90	VHS-PAL	PG	Elia Kazan	B/W
12	After Hours	English	90	VHS-PAL	15	M. Scorsese	-
13	Unforgiven	English	120	VHS-PAL	15	C. Eastwood	-
14	The Unforgiven	English	116	VHS-PAL	PG	John Huston	Unrelated to (13)
15	Trois Couleurs: Bleu	French	100	VHS-PAL	15	Kieslowski	-
16	Tokyo monogatari	Japanese	139	VHS-PAL	U	Y. Ozu	B/W
17	Taxi Driver	English	110	VHS-PAL	18	M. Scorsese	-
18	Triumph des Willens	German	114	VHS-PAL	E	L. Riefenstahl	B/W
19	The Vanishing	Dutch/French/English	107	VHS-PAL	12	G. Sluizer	-
20	The Vanishing	English	110	VHS-PAL	15	G. Sluizer	1988 remake of (19)

Figure 9.1: Partial table of records and attributes.

A fairly simple numerical key has been chosen to identify each distinct record. It is often the case that a, superficially redundant, mechanism such as this is used: in a typical database system many records will have identical values in a given field (e.g. the Format or Director attributes in the example); even an attribute that might be expected uniquely to identify a record in a natural way might not always do so, for example, the **Title** attribute would not suffice in general: in addition to the similarity between the titles of (13) and (14), there are (although not listed) at least 2 other films with the same title as (17), and another film which might be entered with the same title as (3) (i.e. Imamura's *Kuroi ame* from which (3)'s English title derives); the most obvious example in Figure 9.1. is, of course, given by (19) and (20): the latter being an inferior near scene-for-scene remake of (19) by the same director with a similar running time, the major difference being the weakening of the original's ending to suit the (perceived) tastes of U.S. audiences.

As regards the observations made at the start of this section, it is not evidently the case that any atributes can be deduced from the values of other fields: while it might be thought that the **Director** field could be used, safely to determine the **Language** attribute, (9) and (15) show that this is not the case (as would (5) and any of the director's U.S. films; similarly, the director of (1) has made at least one film in English). Even combining the **Director** and **Title** fields does not provide sufficient information, e.g. (5) and any of Lang's U.S. films; or (19) and (20).

Supposing that we have organised a collection of records of the form above, typical **queries** that one might wish to make could be,

- List all of the film **Titles** with a given **Director**.
- Give the total number of films with an 'adult' **Certificate**, i.e. **18** (in U.K.) or **NC-17** (in U.S: No Children under 17).
- List all the film **Titles** and **Directors** of records whose **Language** is not English and **Running Time** exceeds 110 minutes.

In the next part of this module you will be introduced to a specific system for formulating queries such as these with respect to data organised in a given form. While, in principle, similar capabilities could be realised by extending the mechanisms described in this lecture, in practice, such an approach would have several disadvantages, not least of which is its trendency to become rather application specific.

We present, in this lecture, realisations of the following methods, that can be used as a rudimentary framework for maintaining a collection of records in which each record has a unique **key** identifying it, the collection being organised as a linked list structure 'ordered' according to the value of the keys:

- A method to determine if a given record (i.e. key value) is present in a collection.
- A method to return the record data associated with a given key value.
- A method to insert a record into a collection.
- A method to delete a record from a collection.

In order to facilitate the realisation it is useful to develop the linked list class of Lecture 6 to the form presented in Figure 9.2 below:

```
//
// COMP102
```

```
// Example 15: Development of Linked List class for Records
//             with unique Key fields.
//
//             N.B. This is effectively identical to Example 9
//
// Paul E. Dunne 30/12/99
//
public class Records
  {
  //*******************************************************
  // Embed a RecordCell Class as earlier.                 *
  //*******************************************************
  private class RecordCell
    {
    private Object Key;            // Unique identifying Key value
    private Object Datum;
    private RecordCell Link;
    //
    private RecordCell(Object KeyVal, Object head, RecordCell next_cell)
      {
      Key=KeyVal; Datum = head; Link=next_cell;
      }
  }
  //*******************************************************
  // Records Fields                                       *
  //*******************************************************
  private RecordCell Head;
  private RecordCell Tail;
  private int CellCount;         // The number of records in this instance.
  //*******************************************************
  // Constructor                                          *
  //*******************************************************
  public Records()
    {
    Head = null;
    Tail = null;
    CellCount=0; // Initiates the empty record set;
    }
  //*******************************************************
  // Instance Methods                                     *
  //*******************************************************
  //
  //  Test if this instance is Empty
  //
  public boolean IsEmpty()
    {
    return (CellCount==0);       // 'Safer' than testing for null reference.
    }
  //
  //*************************************************************
  //  Add a new RecordCell as the first cell in this           *
  //*************************************************************
  //
  public void AddHead ( Object Datum, Object KeyVal )
    {
    if (IsEmpty())               // If it's currently empty;
      {
      Head = new RecordCell(KeyVal,Datum,Head);
      Tail = null;                       // The link is still 'null'
      CellCount=1;                       // and this list now contains 1 cell.
      }
    else                                  // If it's not empty
      {
      Head = new RecordCell(KeyVal,Datum,Head);   // Add the new head Datum,
      Tail = Head.Link;                   // Reset the 'Tail' of the list.
      CellCount++;                        // and this list now has 1 extra cell.
      };
    }
  //
  //*********************************************************************
  //  Remove the RecordCell at the start of this instance              *
  //*********************************************************************
  //
  public void RemoveHead()
    {
    if (CellCount<=1)                     // There is an argument for raising
      {                                   // an exception if CellCount=0
                                          // however, we adopt the convention
                                          // that RemoveHead() applied to
                                          // empty leaves empty.
      Head = null;
      CellCount=0;
      }
    else
      {
      Head = Tail;                       // Tail is never the null reference here.
      CellCount--;                       // and this instance has one fewer record.
      if (CellCount==1)
        Tail=null;                       // If there's only a single cell then its L
                                         // must point to empty (i.e. null reference
      else
        Tail = Head.Link;                // Otherwise we can update Tail without any
      };
    }
  //
  //********************************************************************
  //  Obtain the Object in the Datum field of the RecordCell at        *
  //  the start of this instance.                                      *
  //********************************************************************
  //
  public Object GetHeadDatum()
    {
    if (!(CellCount==0))                 // If there's anything to return
      return Head.Datum;                 // then return it.
    else
      return null;                       // otherwise return a null reference.
    }
  //
  //********************************************************************
  //  Obtain the Object in the Key field of the RecordCell at          *
  //  the start of this instance.                                      *
  //********************************************************************
  //
  public Object GetHeadKey()
    {
    if (!(CellCount==0))                 // If there's anything to return
      return Head.Key;                 // then return it.
    else
      return null;                       // otherwise return a null reference.
    }
  //
  //********************************************************************
```

```
//  Obtain the (reference) to the Records for the Link field of the    *
//  RecordCell at the start of this instance.
//  This method will not change the current instantiation
//**********************************************************************
//
public Records GetTail()
  {
  Records temporary= new Records();
  temporary.Head=Head; temporary.Tail=Tail;
  temporary.CellCount = CellCount;
  temporary.RemoveHead();
  return temporary;
  }
}
```

Figure 9.2: Development of Linked List Methods to Incorporate Key field.

3. Maintaining a 'sorted' list of records: Insertion and Deletion

One important point of which we have said little so far, concerns the concept of maintaining a **sorted** collection (using the **Key** field to determine the ordering). This field is defined as an arbitrary Object instance, and with the exception of sub-classes such as Number or String there may be no 'natural' obvious way of defining what is meant by an Object instance **X** say, preceding another instance **Y**. We overcome this difficulty, in a rather naive way, by using the method presented in Fgure 9.3 below,

```
//*********************************************//
// Compare the two Objects, X and Y:           //
// a numeric comparison if both are Number     //
// instances, and String comparison otherwise//
// Returns true if X is 'less than' Y          //
//*********************************************//
public static boolean IsBefore ( Object X, Object Y)
  {
  double xv, yv;
  if ( (X instanceof Number) &&
         (Y instanceof Number) )
    {
    xv = Double.valueOf(X.toString()).doubleValue();
    yv = Double.valueOf(Y.toString()).doubleValue();
    return (xv < yv);
    }
  else
    {
    return ( (X.toString()).compareTo(Y.toString()) < 0);
    };
  }
```

Figure 9.3: Simple mechanism for Ordering Object Instances

The method **IsBefore(X,Y)**, uses a standard numeric interpretation of < if both parameters are instances of Number Objects. Otherwise, the comparison is made by invoking the relevant Object class methods for converting an instance to a String and using a lexicographic comparison. Notice that, for user-defined Object classes, it is usual to over-ride the method toString(), and so this could always be implemented to achieve the required ordering needed.

It should also be observed, that when using Number instances, making a comparison between String representations will **not** in general give the same answers as comparing the numeric values represented, e.g. **2 < 10** (numerical) but the String "10" precedes the String "2", using a standard lexicographic ordering.

Before illustrating how Insertion and Deletion can be handled we deal with 2 methods, whose structure is very similar: those for tersting if a given **Key** value is already present, and for return the Record Datum tied to a given Key. These methods, **IsMember(Key, RBase)** and GetRecord(Key,Rbase) are shown in Figure 9.4:

```
//*********************************************//
// Determine if Key occurs in RB               //
//*********************************************//
public static boolean IsMember ( Object Key, Records RB )
  {
  Records temp = RB;
  boolean found_key = false;
  while ((!found_key) && (!temp.IsEmpty()) )
    {
    if ( !IsBefore(Key,temp.GetHeadKey()) && !IsBefore(temp.GetHeadKey(),Key) )
      {
      found_key=true;
      }
    else
      temp = temp.GetTail();
    };
  return found_key;
  }
//*********************************************//
// Return the Record Data of the entry in RB //
// whose Key field matches the Key parameter //
//*********************************************//
public static Object GetRecord ( Object Key, Records RB )
  {
  Records temp = RB;
  Object result = null;
  boolean found_key = false;
  while ((!found_key) && (!temp.IsEmpty()) )
    {
    if ( !IsBefore(Key,temp.GetHeadKey()) && !IsBefore(temp.GetHeadKey(),Key) )
      {
      result = temp.GetHeadDatum(); found_key=true;
      }
    else
      temp = temp.GetTail();
    };
  return result;
  }
```

Figure 9.4: Testing Membership and Obtaining Record Data

The process required in order to ensure that a new record is correclty placed within an ordered list of records is a little bit more complicated than the preceding methods that have been discussed. Not only do wish to ensure that the same key does not occur more than once in a given list, we also have to locate the correct point at which to insert the new record without otherwise changing the list organisation.

We can consider four cases when inserting a data record **Entry** with key value **Key** into a collection of records **RB**:

1. **Key** already occurs as the identifying key of some record in **RB**.
2. **RB** contains no records, i.e. is the empty collection.
3. **Key** precedes the value of the identifying key of the first record in **RB**.
4. None of the above cases hold, i.e. **Key** is not in the collection being updated, this collection has at least one record in it, and the record being added does not precede the the first record in the collection.

What actions do each of these cases entail?

In Case 1, the simplest approach is just to return the collection into which **Key** is being inserted, unchanged.

For Case 2 and Case 3, the record being added will become the new first record in the ordered collection and so the collection resulting by invoking instance method **AddHead** with paramters **Entry** and **Key** can be reported.

Finally, in Case 4, the only possibility is that **Key** comes **after** the key value held in the first record of **RB**. It follows that the pair (Key,Entry) have to inserted into the collection formed by all records in **RB** after the first. Hence we can recursively insert the new record into the tail of the list and append the existing first element onto the result.

This gives the realisation of Figure 9.5:

```
//************************************************************************
public static Records InsertRecord ( Object Key, Object Entry, Records RB)
  {
  Records res = RB;   // Result returned.
  Records temp;
  if (IsMember(Key,RB))
    {
    return res;
    }
  else if ( (RB.IsEmpty()) || (IsBefore(Key,RB.GetHeadKey())) )
    {
    res.AddHead(Entry,Key);
    return res;
    }
  else
    {
    temp = new Records();
    temp.AddHead(RB.GetHeadDatum(),RB.GetHeadKey());
    return Concatenate(temp, InsertRecord(Key,Entry,RB.GetTail()) );
    };
  }
```

Figure 9.5: Inserting a new Record into an existing Collection

Note the use of the (adapted) **Concatenate** method that was described in Lecture 6.

We have a similar breakdown in the case of deleting a record with a specified key from an ordered collection.

If no record with the supplied **Key** is present in **RB** then no action is required.

If the key field of the first record in **RB** precedes the **Key** specified, then the record to be deleted must occur in the collection formed by the tail of **RB** and so should be deleted from this.

Otherwise, the **Key** value must be that of the **first** record and so the result required is simply the tail of the collection **RB**.

So we have,

```
//*******************************************************************
public static Records DeleteRecord ( Object Key, Records RB)
  {
  Records res = RB;
  Records temp = new Records();
  if (!IsMember(Key,RB))
    {
    return res;
    }
  else if ( IsBefore(RB.GetHeadKey(),Key) )
    {
    temp.AddHead(RB.GetHeadDatum(),RB.GetHeadKey());
    res = Concatenate(temp,DeleteRecord(Key,RB.GetTail()));
    return res;
    }
  else
    {
    res = RB.GetTail(); return res;
    };
  }
```

Figure 9.6: Deleting a record from a Collection.

The full implementation is given in Figure 9.7 below (including the revised Concatenate and Reverse methods).

```
//
// COMP102
// Example 16: Class Methods for Maintaining Records.
//
// Paul E. Dunne 28/12/99
//
import Records;
public class RecordBase
  {
  //*******************************************//
  // Compare the two Objects, X and Y:         //
  // a numeric comparison if both are Number   //
  // instances, and String comparison otherwise//
  // Returns true if X is 'less than' Y        //
  //*******************************************//
  public static boolean IsBefore ( Object X, Object Y)
    {
    double xv, yv;
    if ( (X instanceof Number) &&
           (Y instanceof Number) )
      {
      xv = Double.valueOf(X.toString()).doubleValue();
      yv = Double.valueOf(Y.toString()).doubleValue();
      return (xv < yv);
      }
    else
      {
      return ( (X.toString()).compareTo(Y.toString()) < 0);
      };
```

```
    }
  //*******************************************//
  // Determine if Key occurs in RB             //
  //*******************************************//
  public static boolean IsMember ( Object Key, Records RB )
    {
    Records temp = RB;
    boolean found_key = false;
    while ((!found_key) && (!temp.IsEmpty()) )
      {
      if ( !IsBefore(Key,temp.GetHeadKey()) && !IsBefore(temp.GetHeadKey(),Key) )
        {
        found_key=true;
        }
      else
        temp = temp.GetTail();
      };
    return found_key;
    }
  //*******************************************//
  // Return the Record Data of the entry in RB //
  // whose Key field matches the Key parameter //
  //*******************************************//
  public static Object GetRecord ( Object Key, Records RB )
    {
    Records temp = RB;
    Object result = null;
    boolean found_key = false;
    while ((!found_key) && (!temp.IsEmpty()) )
      {
      if ( !IsBefore(Key,temp.GetHeadKey()) && !IsBefore(temp.GetHeadKey(),Key) )
        {
        result = temp.GetHeadDatum(); found_key=true;
        }
      else
        temp = temp.GetTail();
      };
    return result;
    }
  //*************************************************//
  // The following methods are used in implementing  //
  // Insert and Delete.                              //
  //*************************************************//
  public static Records Reverse ( Records RB )
    {
    Records res = new Records();   // For the result.
    Records temp = RB;
    while ( !temp.IsEmpty() )
      {
      res.AddHead(temp.GetHeadDatum(),temp.GetHeadKey());
      temp = temp.GetTail();
      };
    return res;
    }
  //*************************************************************************
  public static Records Concatenate ( Records Start, Records End)
    {
    Records res = new Records();
    Records temp = new Records();
    Records StartPoint = Start;
    Records EndPoint = End;
```

```
    // Build the first part by stacking the Start
    while (!StartPoint.IsEmpty())
      {
      temp.AddHead(StartPoint.GetHeadDatum(),StartPoint.GetHeadKey());
      StartPoint = StartPoint.GetTail();
      };
    // Stack the End onto the list just built.
    while (!EndPoint.IsEmpty())
      {
      temp.AddHead(EndPoint.GetHeadDatum(),EndPoint.GetHeadKey());
      EndPoint = EndPoint.GetTail();
      };
    res = Reverse(temp);
    return res;
    }
  //*************************************************************************
  public static Records InsertRecord ( Object Key, Object Entry, Records RB)
    {
    Records res = RB;   // Result returned.
    Records temp;
    if (IsMember(Key,RB))
      {
      return res;
      }
    else if ( (RB.IsEmpty()) || (IsBefore(Key,RB.GetHeadKey())) )
      {
      res.AddHead(Entry,Key);
      return res;
      }
    else
      {
      temp = new Records();
      temp.AddHead(RB.GetHeadDatum(),RB.GetHeadKey());
      return Concatenate(temp, InsertRecord(Key,Entry,RB.GetTail()) );
      };
    }
  //*************************************************************************
  public static Records DeleteRecord ( Object Key, Records RB)
    {
    Records res = RB;
    Records temp = new Records();
    if (!IsMember(Key,RB))
      {
      return res;
      }
    else if ( IsBefore(RB.GetHeadKey(),Key) )
      {
      temp.AddHead(RB.GetHeadDatum(),RB.GetHeadKey());
      res = Concatenate(temp,DeleteRecord(Key,RB.GetTail()));
      return res;
      }
    else
      {
      res = RB.GetTail(); return res;
      };
    }
  }
```

Figure 9.7: Realisation of Record manipulation methods.

4. Summary

1. Maintaining information in the form of record structures is one of the major applications of computer systems and is the central topic of study in the field of **Database Systems**
2. Using linked lists as a dynamic data structure (with an imposed ordering regime) is an extremely crude and basic method for providing some rudimentary databse features. This use is, however, unsuitable for all but very small scale applications.
3. Among the drawbacks to list structures in this context are: searching for a given key in an ordered list of keys will (on average) require inspecting half the records in the collection: for all but small (10s-100s) numbers this length of time becomes unacceptable (notice that searching for a given key occurs as an element of all the methods described above).
4. Some of the drawbacks of a simple linked list structure can be overcome by more intricate dynamic forms, one such - Binary Trees will be considered in the next lecture.
5. The study of suitable forms of data organisation to deal with very large record systems (10^6 and more records) continues to be an important research domain in Computer Science.

(Notes prepared and maintained by Paul E. Dunne, December 1999)
