File Structures

Ch08. A. Cosequential Processing and the Sorting of Large Files

2020. Spring

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab

data science laboratory

Slides are based on the textbook.

Outline

Cosequential operations

- Coordinated processing of two or more sequential lists to produce a single list

- Kinds of operations
  - merging, or union
  - matching, or intersection
  - combination of matching and merging

CosequentialProcess class

- A single, simple model that can be the basis for the construction of any kind of consequential process
  - supports processing of any type of list
  - Includes operations to match and merge lists
  - Defines the list processing operations required for cosequential processing as virtual methods

Matching names in two lists (1/6)

- Two lists are sorted in an ascending order
- not allow duplicate names within a list
- reading initial item from each list

List1	List2
ADAMS	ADAMS
CARTER	ANDERSON
CHIN	ANDREWS
DAVIS	BECH
GARWICK	BURNS
JAMES	CARTER
JOHNSON	DAVIS
KARNS	DEMPSEY
LAMBERT	GRAY
TURNER	JSAMES
	JOHNSON
	KATZ
	WILLIS

Matching names in two lists (2/6)

- Comparison: matched case
  - go to outputlist
  - read next item from each list

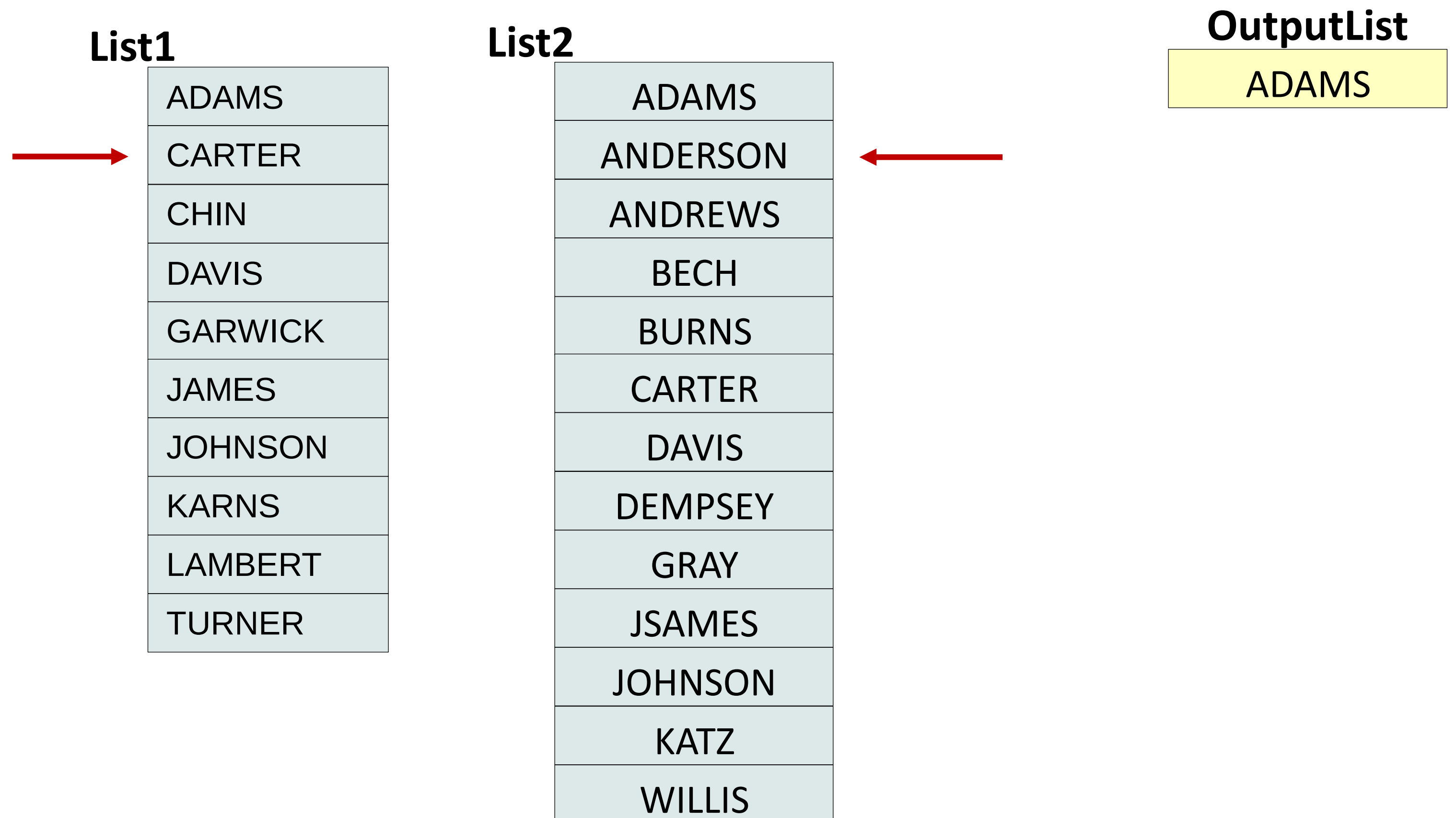
List1
ADAMS
CARTER
CHIN
DAVIS
GARWICK
JAMES
JOHNSON
KARNS
LAMBERT
TURNER

List2
ADAMS
ANDERSON
ANDREWS
BECH
BURNS
CARTER
DAVIS
DEMPSEY
GRAY
JSAMES
JOHNSON
KATZ
WILLIS

OutputList
ADAMS

Matching names in two lists (3/6)

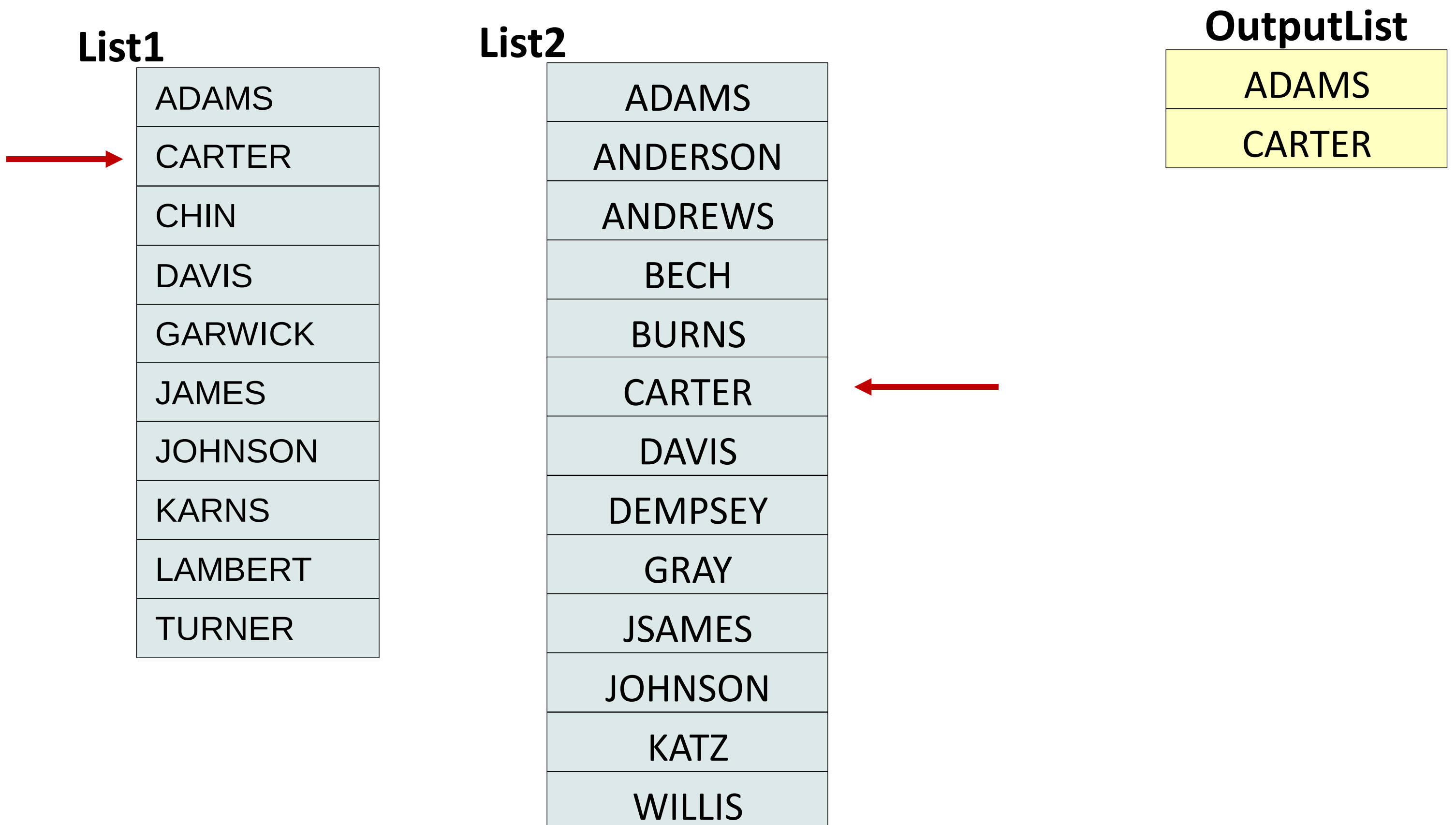
- Comparison: item1 > item2
  - go to outputlist
  - read next item from list2 (same cases until CARTER)



List1	List2	OutputList
ADAMS	ADAMS	ADAMS
CARTER	ANDERSON	
CHIN	ANDREWS	
DAVIS	BECH	
GARWICK	BURNS	
JAMES	CARTER	
JOHNSON	DAVIS	
KARNS	DEMPSEY	
LAMBERT	GRAY	
TURNER	JSAMES	
	JOHNSON	
	KATZ	
	WILLIS	

Matching names in two lists (4/6)

- Comparison: matched case
  - go to outputlist
  - read next item from each list



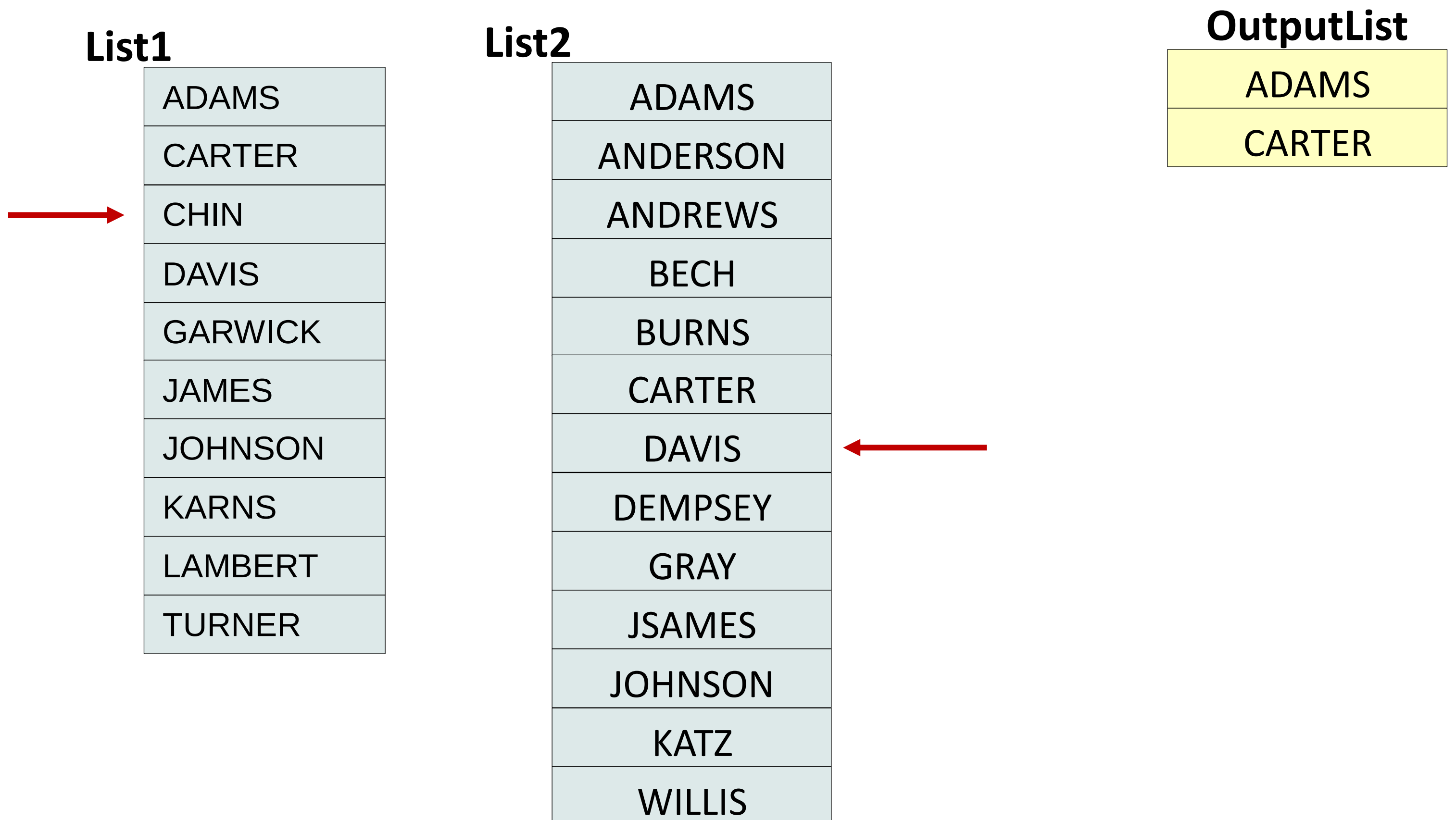
List1
ADAMS
CARTER
CHIN
DAVIS
GARWICK
JAMES
JOHNSON
KARNS
LAMBERT
TURNER

List2
ADAMS
ANDERSON
ANDREWS
BECH
BURNS
CARTER
DAVIS
DEMPSEY
GRAY
JSAMES
JOHNSON
KATZ
WILLIS

OutputList
ADAMS
CARTER

Matching names in two lists (5/6)

- Comparison: item1 < item2
  - go to outputlist
  - read next item from List1



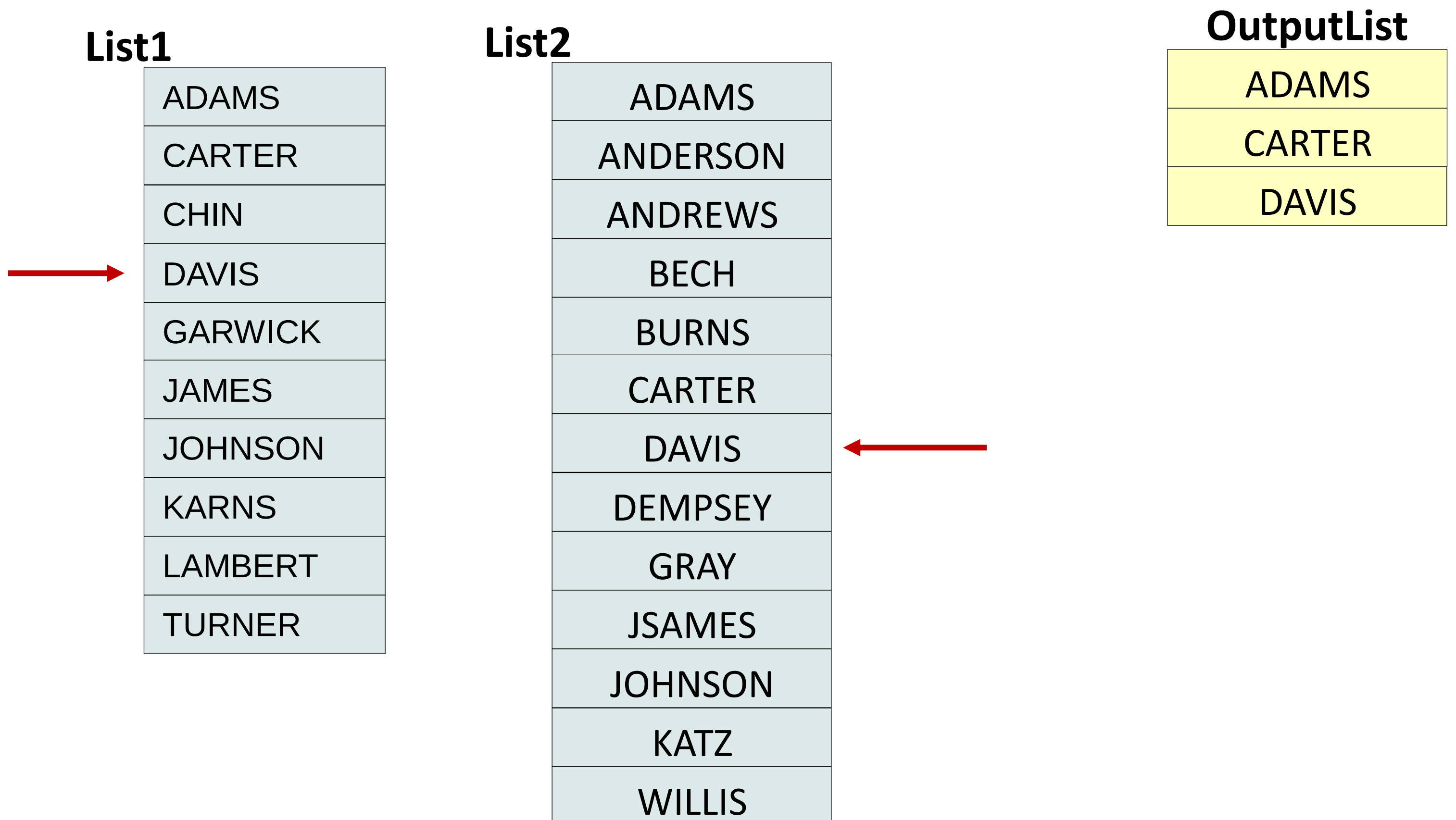
List1
ADAMS
CARTER
CHIN
DAVIS
GARWICK
JAMES
JOHNSON
KARNS
LAMBERT
TURNER

List2
ADAMS
ANDERSON
ANDREWS
BECH
BURNS
CARTER
DAVIS
DEMPSEY
GRAY
JSAMES
JOHNSON
KATZ
WILLIS

OutputList
ADAMS
CARTER

Matching names in two lists (6/6)

- Comparison: matched case
  - go to outputlist
  - read next item from each list



List1

ADAMS
CARTER
CHIN
DAVIS
GARWICK
JAMES
JOHNSON
KARNS
LAMBERT
TURNER

List2

ADAMS
ANDERSON
ANDREWS
BECH
BURNS
CARTER
DAVIS
DEMPSEY
GRAY
JSAMES
JOHNSON
KATZ
WILLIS

OutputList

ADAMS
CARTER
DAVIS

The match procedure

- Major steps
  - initializing that is to arrange things
  - getting and accessing the next list item
  - synchronizing between two lists
    - The current item from one list in never so far ahead of the current item on the other list
      - A match will be missed
  - handling EOF conditions
  - recognizing errors
    - e.g. duplicate names or names out of sequence

Comparison

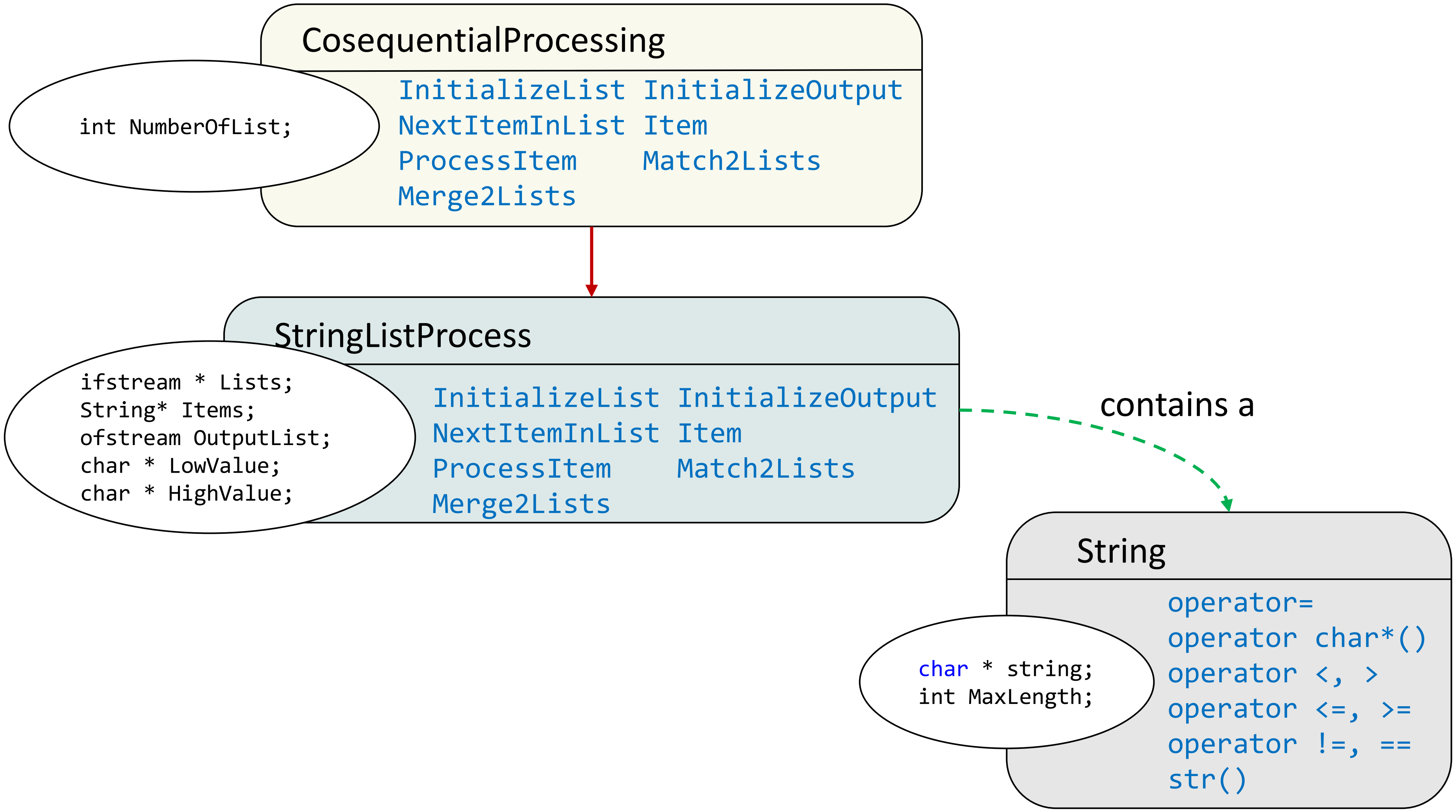
- Three-way-test in synchronizing
  - if Item(1) < Item(2)
    - read the next from List 1
  - if Item(1) > Item(2)
    - read the next name from List 2
  - if Item(1) == Item(2)
    - output the name
    - read the next names from the two lists

Macth2Lists()

```
template <class ItemType>
int CosequentialProcess<ItemType>::Match2Lists
    (char * List1Name, char * List2Name, char * OutputListName)
{
    int MoreItems;// true if items remain in lists
    InitializeList (1, List1Name);
    InitializeList (2, List2Name);
    InitializeOutput(OutputListName);
    MoreItems = NextItemInList(1) && NextItemInList(2);
    while (MoreItems){
        if (Item(1) < Item(2))
            MoreItems = NextItemInList(1);
        else if (Item(1) == Item(2)) // Item1 == Item2
        {
            ProcessItem (1); // match found
            MoreItems = NextItemInList(1) && NextItemInList(2);
        }
        else // Item(1) > Item(2)
            MoreItems = NextItemInList(2);
    }
    FinishUp();
    return 1;
}
```

Class for Cosequential Processing (1/2)

- Class hierarchy



Class for Cosequential Processing (2/2)

- Class CosequentialProcess
  - an abstract class, since it does not contain definitions of the supporting methods.
- Subclass StringListProcess that defines the specific supporting methods
  - allows any number of input lists.
  - Member LowValue is a value that is smaller than any value that can appear in a list

Class CosequentialProcess

```
template <class ItemType>
class CosequentialProcess
{
public:
    CosequentialProcess(int numberOfLists);
    // The following methods provide basic list processing
    // These must be defined in subclasses
    virtual int InitializeList (int ListNumber, char * ListName)=0;
    virtual int InitializeOutput (char * OutputListName)=0;
    virtual int NextItemInList (int ListNumber)=0;
    //advance to next item in this list
    virtual ItemType Item (int ListNumber) = 0;
    // return current item from this list
    virtual int ProcessItem (int ListNumber)=0; // process the item in this list
    virtual int FinishUp()=0; // complete the processing

    // 2-way sequential matching algorithm
    virtual int Match2Lists (char * List1Name,
         char * List2Name, char * OutputListName);
    // 2-way sequential merging algorithm
    virtual int Merge2Lists (char * List1Name,
         char * List2Name, char * OutputListName);
protected:
    int NumberOfLists; // number of lists to be processed
};
```

Class StringListProcess

- StringListProcess
  - support lists that are files of strings, one per line

```
class StringListProcess: public CosequentialProcess<String&>
{public:
    StringListProcess (int NumberOfLists); // constructor

    // Basic list processing methods
    int InitializeList (int ListNumber, char * ListName);
    int InitializeOutput (char * OutputListName);
    int NextItemInList (int ListNumber); //get next item from this list
    String& Item (int ListNumber); // return current item from this list
    int ProcessItem (int ListNumber); // process the item in this list
    int FinishUp(); // complete the processing
protected:
    ifstream * Lists; // array of list files
    String * Items; // array of current Item from each list
    ofstream OutputList;
    static const char * LowValue;
    static const char * HighValue;
};
```

Methods of StringListProcess (1/2)

- NextIemInList

```
int StringListProcess::NextItemInList (int ListNumber)
{
    char ThisItem[MaxItemLength];
    String PreviousItem = Items[ListNumber];
    // get line from file
    Lists[ListNumber].getline(ThisItem, MaxItemLength+1);
    // test for errors and termination
    if (!Lists[ListNumber].good()) // end of file
    {   Items[ListNumber]=HighValue; return 0;}
    if (strlen(ThisItem)==0) // no string
    {   Items[ListNumber]=LowValue; return 0;}
    if (strcmp(ThisItem, (char*)PreviousItem) < 0)
    {
        cerr << "Items out of order: current "<<ThisItem
            << " previous "<<(char*)PreviousItem<<endl;
        Items[ListNumber]=HighValue; return 0;
    }
    // this is a new item, store it
    Items[ListNumber]=ThisItem;//store this Item as current item
    return 1;
}
```

Methods of StringListProcess (2/2)

```
int StringListProcess::InitializeList (int ListNumber, char * ListName)
{
    Lists[ListNumber].open(ListName);
    Items[ListNumber]=LowValue;
    return 1;
}

// return current item from this list
String& StringListProcess::Item (int ListNumber)
{   return Items[ListNumber];}

// process the item in this list
// output a line containing the item
int StringListProcess::ProcessItem (int ListNumber)
{
    OutputList << Items[ListNumber] <<endl;
    return 1;
}

int StringListProcess::FinishUp()
{
    for (int i = 1; i <= NumberOfLists; i++)
        Lists[i].close();
    OutputList.close();
    return 1;
}
```

Merging Two Lists(1/3)

- Based on matching operation
- Difference
  - must read each of the lists completely
  - must change MoreItems behavior
    - keep this flag set to true as long as there are records in either list
- HighValue
  - the special value (we use “\xFF”)
  - come after all legal input values in the files to ensure both input files are read to completion

Merging Two Lists(2/3)

- merge procedure based on a single loop
  - This method has been added to class CosequentialProcess
  - No modifications are required to class StringListProcess

```
template <class ItemType>
int CosequentialProcess<ItemType>::Merge2Lists
    (char * List1Name, char * List2Name, char * OutputListName)
{
    int MoreItems1, MoreItems2; // true if more items in list
    InitializeList (1, List1Name);
    InitializeList (2, List2Name);
    InitializeOutput (OutputListName);
    MoreItems1 = NextItemInList(1);
    MoreItems2 = NextItemInList(2);

    while (MoreItems1 || MoreItems2){// if either file has more
        if (Item(1) < Item(2))
        {// list 1 has next item to be processed
            ProcessItem (1);
            MoreItems1 = NextItemInList(1);
        }
```

Merging Two Lists(3/3)

```
        else if (Item(1) == Item(2))
        {// lists have the same item, process from list 1
            ProcessItem (1);
            MoreItems1 = NextItemInList(1);
            MoreItems2 = NextItemInList(2);
        }
        else // Item(1) > Item(2)
        {// list 2 has next item to be processed
            ProcessItem (2);
            MoreItems2 = NextItemInList(2);
        }
    }
    FinishUp();
    return 1;
}
```

Test program

```
#include "strlist.h"

int main ()
{
    StringListProcess List(2);// declare process with 2 lists
    List.Match2Lists ("list1.txt","list2.txt","match.txt");
    List.Merge2Lists ("list1.txt","list2.txt","merge.txt");
    return 1;
}
```

Summary of the Cosequential Processing Model (1/2)

- Assumptions
  - two or more input files are processed in a parallel fashion
  - each file is sorted
  - in some cases, there must exist a high key value or a low key
  - records are processed in a logical sorted order
  - for each file, there is only one current record
  - records should be manipulated only in internal memory

Summary of the Cosequential Processing Model (2/2)

- Essential Components
  - initialization - reads from first logical records
  - one main synchronization loop
    - continues as long as relevant records remain
    - selection in main synchronization loop
  - Input files & Output files are sequence checked by comparing the previous item value with new one
  - substitute high values for actual key when EOF
    - main loop terminates when high values have occurred for all relevant input files
    - no special code to deal with EOF

Outline

General Ledger Problem

- Ledger
  - A book containing accounts to which debits and credits are posted from books of original entry.
- Goal
  - design a general ledger as part of an accounting system
  - includes a journal file and a ledger file (have two lists)
    - Master File:ledger file
      - monthly summary of account balance for each of the book keeping accounts.
    - Transaction File:journal file
      - contains the monthly transactions to be posted to the ledger

Sample ledger fragment

- Containing checking and expense accounts

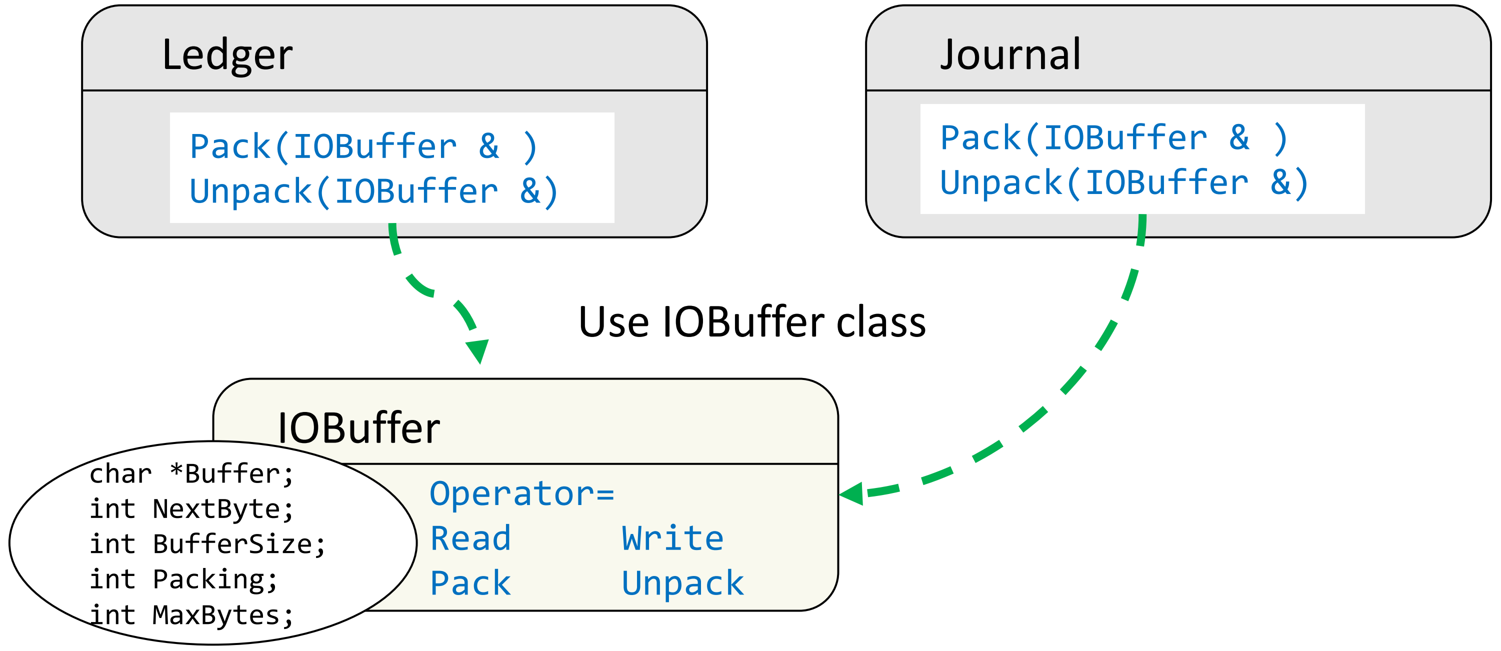
AcctNo	Account title	Jan	Feb	Mar	Apr
101	Checking account #1	1032.57	2114.56	5219.23	
102	Checking account #3	543.78	3094.17	1321.20	
505	Advertising expense	25.00	25.00	25.00	
510	Auto expense	195.40	307.92	501.12	
515	Bank charges	0.00	0.00	0.00	
520	Books and publications	27.95	27.95	87.00	
525	Interest expense	103.50	255.20	380.27	
535	Miscellaneous expense	12.45	17.87	23.87	
540	Postage and shipping	21.00	27.63	57/45	
550	Rent	500.00	1000.00	1500.00	
555	Supplies	112.00	167.50	2441.80	

Sample journal entries

AcctNo	CheckNo.	Date	Description	Debut/Credit
101	1271	04/02/86	Auto expense	-78.70
510	1271	04/02/97	Tune-up and minor repair	78.70
101	1272	04/02/97	Rent	-500.00
550	1272	04/02/97	Rent for April	500.00
101	1273	04/04/97	Advertising	-87.50
505	1273	04/04/97	Newspaper ad renew product	87.50
102	670	04/02/97	Office expense	-32.78
540	670	04/02/97	Printer catridge	32.78
101	1274	04/02/97	Auto expense	-31.83
510	1274	04/09/97	Oil change	31.83

The classes Ledger and Journal

- extensive use of the IOBuffer and RecordFile classes for their file operations
  - Similar to Recording class (in ch5.3, ch7.2)



The classes Ledger and Journal (1/2)

- extensive use of the IOBuffer and RecordFile classes for their file operations

```
class Ledger
{
public:
    int Acct;
    char Title [30];
    double Balances[12];
    int Pack(IOBuffer & buffer) const;
    int Unpack (IOBuffer & buffer);
    ostream & Print (ostream &);
    ostream & PrintHeader (ostream &);
    Ledger ();
    Ledger (int, char *, double, double, double);
};
```

The classes Ledger and Journal (2/2)

```
class Journal
{
public:
    int Acct;
    int CheckNum;
    char Date[10];
    char Description[30];
    double Amount;
    int Pack (IOBuffer &) const;
    int Unpack (IOBuffer &);
    ostream & PrintLine (ostream &);
    Journal ();
    Journal (int, int, char*, char*, double);
};
```

Sample Ledger Printout

- What information
  - Once the journal file is complete for a given month, the journal must be posted to the ledger
  - **Posting** involves associating each transaction with its account in the ledger

```
101      Checking account #1
           1271     04/02/86   Auto expense                    -78.70
           1272     04/02/97   Rent                           -500.00
           1273     04/04/97   Advertising                     -87.50
           1274     04/02/97   Auto expense                    -31.83
                          Prev.bal: 5219.23          New Bal:         4521.20
102      Checking account #3
            670     04/02/97   Office expense                  -32.78
                          Prev.bal: 1321.20          New Bal:         1288.42
505      Advertising expense
           1273     04/04/97   Newspaper ad renew product       87.50
                          Prev.bal: 25.00            New Bal:         112.50
```

Posting Process (1/3)

- How to implement?
  - Method1:
    - uses the account number as a key to relate the journal transactions to the ledger records.
    - building an index for the ledger
  - Method2:
    - collecting all the journal transactions that relate to a given account
      - Sorting journal → working through both ledger and the sorted journal consequentially
    - referred to as **a master-transaction process**

Posting Process (2/3)

- Cosequential matching or the ledger and journal files

Ledger List

AcctNo	Account title
101	Checking account #1
102	Checking account #3
505	Advertising expense
510	Auto expense

Journal List

AcctNo	CheckNo.	Description
101	1271	Auto expense
101	1272	Rent
101	1273	Advertising
101	1274	Auto expense
102	670	Office expense
505	1273	Newspaper ad renew product
510	1271	Tune-up and minor repair
510	1274	Oil change

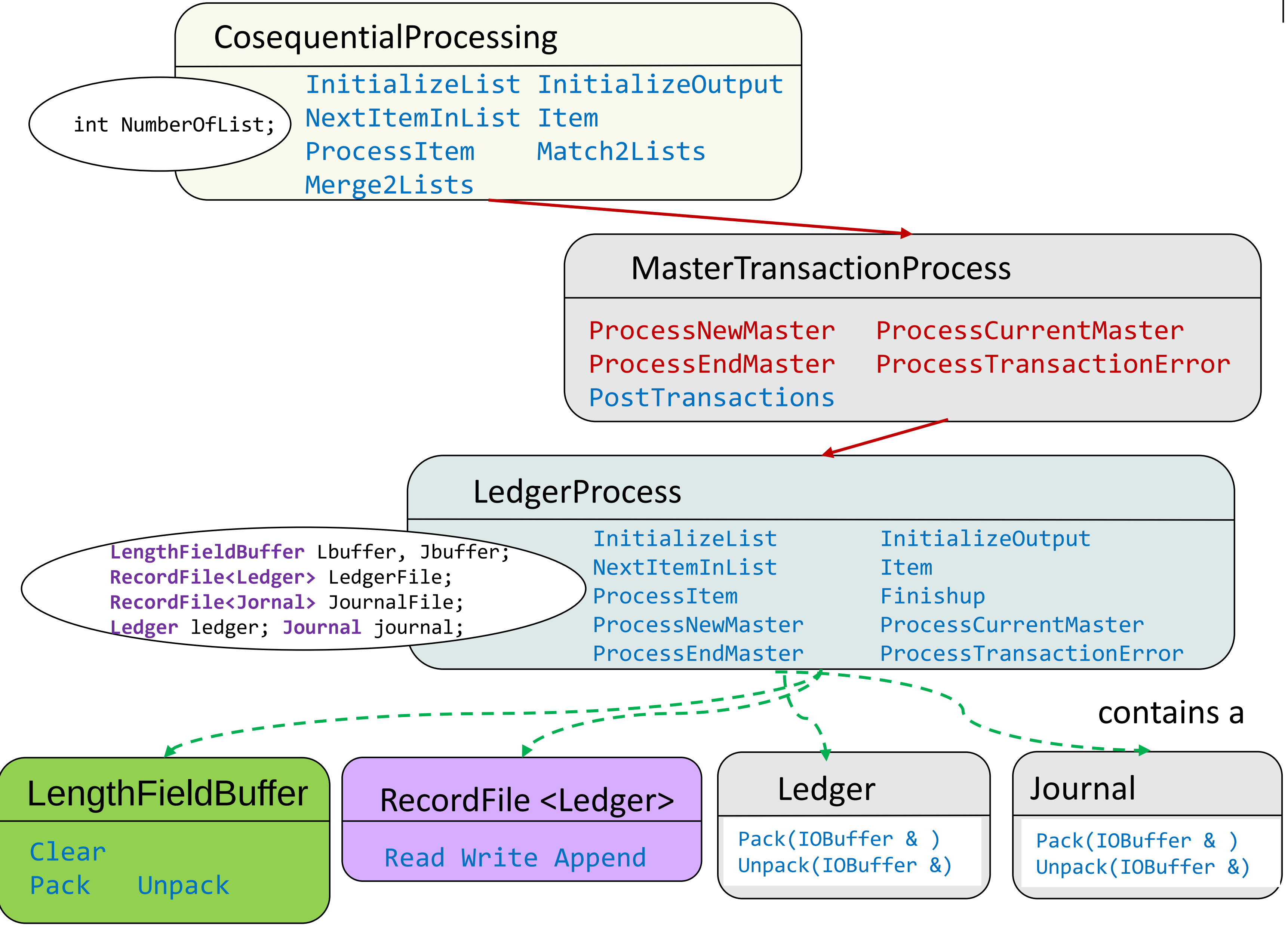
Posting Process (3/3)

- From the point of view of ledger account
  - Merging
    - unmatched accounts go to printout
- From the point of view of journal account
  - Matching
    - unmatched accounts in journal constitute an error
- The posting method is a combined merging/ matching.

The monthly ledger posting program

- Tasks to be performed
  - update the ledger file with the correct balance for each account
  - Produce printout as in the example
    - to print the header line and initialize the balance for the next month from the previous month's balance.
    - For each transaction object that matches, we need to update the account balance
    - After the last transaction for the account, the balance line should be printed.

Class Hierarchy



Class MasterTransactionProcess (1/2)

- MasterTransactionProcess
  - Is a subclass of CosequentialProcess
  - defining four new pure virtual methods
  - the full implementation of the posting operation

Class MasterTransactionProcess (2/2)

```
template <class ItemType>
class MasterTransactionProcess: public CosequentialProcess<ItemType>
{
public:
    MasterTransactionProcess ();//constructor
    // when new master read
    virtual int ProcessNewMaster ()=0;
    // each transaction for a master
    virtual int ProcessCurrentMaster ()=0;
    // after all transactions for a master
    virtual int ProcessEndMaster ()=0;
    // no master for transaction
    virtual int ProcessTransactionError ()=0;

    // cosequential processing of master and transaction records
    int PostTransactions
        (char * MasterFileName, char * TransactionFileName,
            char * OutputListName);
};
```

PostTransaction (1/2)

- Actions
  - Initialization
  - Beginning with the first element from each list

```
// Item(1): always stores the current master record
// Item(2): always stores the current transactions record

template <class ItemType>
int MasterTransactionProcess<ItemType>::PostTransactions
(char * MasterFileName, char * TransactionFileName, char * OutputListName)
{
    int MoreMasters, MoreTransactions; // true if more items in particular list
    InitializeList (1, MasterFileName);
    InitializeList (2, TransactionFileName);
    InitializeOutput (OutputListName);
    MoreMasters = NextItemInList(1);
    MoreTransactions = NextItemInList(2);
    if (MoreMasters) ProcessNewMaster(); // process first master
```

PostTransaction (2/2)

- the three-way-test loop

```
    while (MoreMasters || MoreTransactions){// if either file has more
         if (Item(1) < Item(2))
         {// finish this master record
             ProcessEndMaster();
             MoreMasters = NextItemInList(1);
             // If read successful, then print title line for new account
             if (MoreMasters) ProcessNewMaster();
         }
         else if (Item(1) == Item(2)) // Transaction matches Master
         {
             ProcessCurrentMaster(); // another transaction for the master
             ProcessItem (2);// output transaction record
             MoreTransactions = NextItemInList(2);
         }
         else // Item(1) > Item(2)
         {// transaction with no master
             ProcessTransactionError();
             MoreTransactions = NextItemInList(2);
         }
     } // end of while
     FinishUp();
     return 1;
}
```

Class LedgerProcess (1/2)

- implementation of the ledger posting application

```
// ledger processing of a ledger file and a journal file
// the item type is int to represent an account number
class LedgerProcess: public MasterTransactionProcess<int>
{
public:
    LedgerProcess(int monthNumber); // constructor

    // Basic list processing methods
    int InitializeList (int ListNumber, char * List1Name);
    int InitializeOutput (char * OutputListName);
    int NextItemInList (int ListNumber); //get next item from this list
    int Item (int ListNumber); // return current item from this list
    int ProcessItem (int ListNumber); // process the item in this list
    int FinishUp(); // complete the processing

    // master/transaction methods
    virtual int ProcessNewMaster ();//  when new master read
    virtual int ProcessCurrentMaster ();// each transaction for a master
    virtual int ProcessEndMaster ();// after all transactions for a master
    virtual int ProcessTransactionError ();// no master for transaction
```

Class LedgerProcess (2/2)

```
protected:
    // members
    int MonthNumber; // number of month to be processed
    LengthFieldBuffer Lbuffer, Jbuffer; // buffers for files
    RecordFile<Ledger> LedgerFile ; // list 1
    RecordFile<Journal> JournalFile;// list 2
    int AccountNumber [3]; // current item in each list
    Ledger ledger; // current ledger object
    Journal journal; // current journal object
    ofstream OutputList; // text output file for post method
    static int LowAcct;// lower than the lowest account number
    static int HighAcct;// higher than the highest account number

    int NextItemInLedger ();
    int NextItemInJournal ();
};
```

Methods of LedgerProcess

```
int LedgerProcess::ProcessNewMaster ()//  when new master read
{// first step in proceeing master record
 // print the header and setup last month's balance
    ledger.PrintHeader(OutputList);
    ledger.Balances[MonthNumber] = ledger.Balances[MonthNumber-1];
    return TRUE;
}

int LedgerProcess::ProcessCurrentMaster ()// each transaction for a master
{// add the transaction amount to the balance for this month
    ledger.Balances[MonthNumber] += journal.Amount;
    return TRUE;
}

int LedgerProcess::ProcessEndMaster ()// after all transactions for a master
{// print the balances line to output
    PrintBalances(OutputList,ledger.Balances[MonthNumber-1],ledger.Balances
         [MonthNumber]);
    return TRUE;
}
```

Q&A