```
 1: /*****************************************************************************
 2: Course:     EECS 280, Winter 2002       Section:     005
 3: Author:     James Glettler              Uniquename:  JGLETTLE
 4: Assignment: Project05 - C
 5: Filename:   proj5.cpp                   Date:        13 April 2002
 6: Version:    v0.01a
 7: Related:    associationlist.h node.h pair.h
 8: Descrip:    N/A
 9: Notes:      Some code has been copied from that provided by the instructors, but
10:            most of that code has had to be majorly reworked if not for logic,
11:            just for style and readability
12:
13:            Episode Names are stored without leading -- or surounding quotes
14:               See getPair function to change this
15: *****************************************************************************/
16:
17: //======================Preprocessor Directives==============================
18: using namespace std;
19:
20: #include <iostream>
21: #include <fstream>
22: #include <string>
23: #include <cctype>
24: #include "associationlist.h"
25: #include "node.h"
26: #include "pair.h"
27:
28: /* #Define Macros for templating Lists: Templates cannot be typedef'd*/
29: #define Quote Pair<string,string>
30:   //Quote is a Pair comprised of the EpisodeName(key) and the QuoteString(entry)
31: #define Q_L string,string
32: #define QuoteList AssociationList< Q_L >
33:   //QuoteList is a List that has the QuoteString as both entry and key
34: #define D_B string, QuoteList
35: #define DataBase AssociationList< D_B >
36:   //DataBase is a List that has EpisodeName(key) and QuoteList(entry)
37:
38: //============================Global Constants================================
39: enum Message {outOfRangeError, invalidDataError, fatalError, exitProgram,
40:               episodePrompt, wordPrompt, newQuotePrompt, deletePrompt};
41:
42: //=================Fuction Protypes with Descriptions=======================
43: void printTitle();
44: void printMenu();
45: void printMessage(ostream& ostr, Message messageNum);
46: void readInDataFile(DataBase& QuoteDB);
47: /*****************************************************************************
48: Name:    readInDataFile()
49: I/O:     DataBase(A.L.) - QuoteDB : contents of file loaded into QuoteDB
50: Output:  console : User Interface
51: Use:     This function takes care of all the U.I. handeling, calls openFile to
52:          connect the input file stream.
53: *****************************************************************************/
54: void openFile(ifstream& inFile);
55: /*****************************************************************************
56: Name:    openFile()
57: Output:  ifstream - inFile : returns inFile stream connected to a file
58:          console : User Interface
59: Use:     inFile is a stream not connected to a file.  Call this fucntion to read
60:          console for a valid file name.  Loops until valid file or EOF(Exits)
```

```
61: ***************************************************************************/
62: bool getPair(istream& ins, Quote& thisPair);
63: /***************************************************************************
64: Name:    getPair()
65: I/O:     istream - ins : input stream to read Quote Pair from
66:          Quote - thisPair : Returned Quote Pair <Episode string, Quote String>
67: Output:  bool : returns true if getPair operation was sucessful and valid
68: Use:     This function returns a quote pair read from whatever intput stream
69: ***************************************************************************/
70: void eatSomeYummyWS(string& line);
71: /***************************************************************************
72: Name:    eatSomeYummyWS()
73: I/O:     string - line : line to remove leading whitespace from
74: Use:     Call function and it removes any leading whitespace from the string
75: ***************************************************************************/
76: bool chkForEpisode(istream& ins);
77: /***************************************************************************
78: Name:    chkForEpisode()
79: I/O:     istream - ins : Stream to check for '--' and remove leading WS
80: Output:  bool : Returns true if next non-WS on the same line is '--'
81: Use:     self explanitory
82: ***************************************************************************/
83: void listQuotationsEpisode(DataBase& QuoteDB);
84: /***************************************************************************
85: Name:    listQuotationsEpisode()
86: I/O:     DataBase(A.L.) - QuoteDB : Main Quote Database list
87:          console : User Interface
88: Use:     User prompted for exact Episode name. If it exists, prints all quotes
89:          in order.  If Episode DNE, throw NonExistantKey.
90: Notes:   N/A
91: ***************************************************************************/
92: void listQuotationsWord(DataBase& QuoteDB);
93: /***************************************************************************
94: Name:    listQuotationsWord()
95: I/O:     DataBase(A.L.) - QuoteDB : Main Quote Database list
96:          console : User Interface
97: Use:     User prompted for phrase.  Prints all quotes that contain this phrase.
98: Notes:
99: ***************************************************************************/
100: void listAllQuotations(DataBase& QuoteDB);
101: /***************************************************************************
102: Name:    listAllQuotations()
103: I/O:     DataBase(A.L.) - QuoteDB : Main Quote Database list
104:          console : User Interface
105: Use:     Prints all quotations from all episodes in order.
106: Notes:   N/A
107: ***************************************************************************/
108: void addQuotation(DataBase& QuoteDB);
109: /***************************************************************************
110: Name:    addQuotation()
111: I/O:     DataBase(A.L.) - QuoteDB : Main Quote Database list
112:          console : User Interface
113: Use:     Prompts user for Episode and quote.  Inserts into DB
114: Notes:   ?????What happens in an error condition (bad console or bad pair)
115: ***************************************************************************/
116: void deleteQuotation(DataBase& QuoteDB);
117: /***************************************************************************
118: Name:    deleteQuotation()
119: I/O:     DataBase(A.L.) - QuoteDB : Main Quote Database list
120:          console : User Interface
```

```
121: Use:     N/A
122: Notes:   N/A
123: *******************************************************************************/
124:
125:
126: //==============================Main Function=================================
127: int main()
128: { //Variable Declarations
129:   DataBase QuoteDB;
130:         //Key <string> holds name of episode
131:         //Entry <A.L.> holds list with Key <string> of quotes,
132:         //Entry of embeded A.L. is char, to take up the least amount of memory
133:   int menuSelection(-1);
134:   bool noQuit = true;
135:
136:   //Main Program Excecution.
137:   printTitle();
138:
139:   readInDataFile(QuoteDB);
140:
141:   do
142:   {
143:     printMenu();
144:     cin >> menuSelection;
145:
146:     try
147:     {
148:     if(cin.good())
149:     {
150:       switch(menuSelection)
151:       {
152:         case 1: listQuotationsEpisode(QuoteDB);       break;
153:         case 2: listQuotationsWord(QuoteDB);          break;
154:         case 3: listAllQuotations(QuoteDB);           break;
155:         case 4: addQuotation(QuoteDB);                break;
156:         case 5: deleteQuotation(QuoteDB);             break;
157:         case 6: noQuit = false;                       break;
158:         default: printMessage(cout, outOfRangeError); break;
159:       }
160:     }
161:     else if(cin.eof())
162:     {
163:       printMessage(cout,fatalError);
164:       exit(1);
165:     }
166:     else
167:     { //Some other error on Read
168:       printMessage(cout, invalidDataError);
169:       cin.clear();
170:       cin.ignore(100000,'\n');
171:     }
172:     } //End of try block
173:
174:     //Catch the Throws                    //OR NOT
175:     /*
176:     catch(EmptyList& Msg)
177:     {
178:       cout << "\nCannot get an element from an empty list: "
179:            << Msg.getMessage() << endl;
180:     }
```

```
181:     catch(NonExistentKey<string>& Msg)
182:     {
183:       cout << endl << Msg.getKey() << " is non-existent for this list\n" ;
184:     }
185:     catch(DuplicateKey<string>& Msg)
186:     {
187:       cout << "\nThis key is already in the list: " << Msg.getKey()  << endl;
188:     }
189:     catch(LastElement LE)
190:     {
191:       cout << "\nCannot go past the last element in the list: " << endl;
192:     }
193:     catch(bad_alloc& err)
194:     {
195:       cout << "\nRan out of memory! \n";
196:     }
197:     catch(...)
198:     { //Some other error
199:       cout << "\nUnexplained Exception. \n";
200:     }
201:     */
202:     catch(...)
203:     {
204:       //DO NOTHING BECAUSE CHRIS RIEGEL SAID SO
205:     }
206:
207:
208:   } while(noQuit);
209:
210:
211:   //End of Main Program
212:   printMessage(cout, exitProgram);
213:   return 0;
214: }
215:
216:
217: //============================Function Definitions==============================
218: void printTitle()
219: { //Print the Title
220:   cout << endl << "Welcome to the Quotations Database" << endl << endl << endl;
221: }
222: void printMenu()
223: { //Prints the Menu
224:   cout << endl << endl << endl
225:        << " 1. List all the quotations from a particular episode" << endl
226:        << " 2. List all quotations that contain a given word or phrase" << endl
227:        << " 3. List all the quotations" << endl
228:        << " 4. Add a quotation" << endl
229:        << " 5. Delete a specific quotation or delete all the quotations from "
230:        <<     "an episode" << endl
231:        << " 6. Exit the program" << endl << endl
232:        << ">> ";
233: }
234:
235: void printMessage(ostream& ostr, Message messageNum)
236: { //Print the Message to the output stream OSTR
237:   switch (messageNum)
238:   { //Error Messages
239:     case outOfRangeError:
240:          ostr << "\n\nOut of range.  Please try  again.\n\n"; break;
```

```
241:     case invalidDataError:
242:          ostr << "\n\nError in data.  Please try again.\n\n"; break;
243:     case fatalError:
244:          ostr<<"\n\nFatal error has occured.  Exiting the  program\n\n\n\n";break;
245:     // Prompts
246:     case exitProgram:
247:          ostr << "\n\nExiting Quotations Database\n\n"; break;
248:     case episodePrompt:
249:          ostr << "\n\nPlease enter the episode:  "; break;
250:     case wordPrompt:
251:          ostr << "\n\nPlease enter the word/phrase:  "; break;
252:     case newQuotePrompt:
253:          ostr << "\n\nPlease enter the episode followed by the quote. \n"
254:               << "Format of episode: \n   --episode name  \n"
255:               << " A blank line, a line with only white space,"
256:               << " or EOF will end the quote.\n\n"; break;
257:     case deletePrompt:
258:          ostr << "\n\nPlease enter the episode/quotation that you wish "
259:               <<"to delete: "; break;
260:     default:
261:          ostr << "\n\nError in parameter" << endl << endl; break;
262:   }
263: }
264:
265: void readInDataFile(DataBase& QuoteDB)
266: { //Variable Declarations
267:   ifstream inFile;
268:   bool keepReading = true;
269:   Quote tmpQuote;
270:   //QuoteList tempQuoteList is declared within loop body for deletion purposes
271:   string quoteStrn;
272:
273:
274:   //Ask user for file name and open File stream
275:   openFile(inFile);
276:
277:   //Loop structure to read the entire file into QuoteDB
278:   do
279:   { //Get a Quote From File Stream
280:     if(getPair(inFile,tmpQuote))
281:     { //Was able to sucessfully get a Pair, try to load Quote into database
282:
283:       /* //DEBUG LINES
284:       cerr << "Pair Read (Episode/Quote): " << endl
285:            << tmpQuote.getKey() << endl
286:            << tmpQuote.getEntry() << endl;
287:       */
288:
289:       QuoteList tempQuoteList; //Create Temporary List here so it goes out
290:                                //of scope and is deleted after each run
291:
292:       quoteStrn = tmpQuote.getEntry(); //Store Quote itself into a temp string
293:
294:       //Check if episode exists
295:       if(QuoteDB.has(tmpQuote.getKey()))
296:       { //QDB has this Episode, current pointer set
297:         tempQuoteList = QuoteDB.getCurrent(); //Gets the Quote List of this Ep.
298:         //Attempts to insert new Quote into tempQuoteList
299:         tempQuoteList.insert(Pair<Q_L>(quoteStrn,quoteStrn));
300:         //If no errors (no throw) then replace tempQuoteList into QuoteDB
```

```
301:         QuoteDB.erase(tmpQuote.getKey()); //Remove Episode
302:         QuoteDB.insert(Pair<D_B>(tmpQuote.getKey(),tempQuoteList));
303:           //Re-inserts Quote list for episode back into QDB
304:       }
305:       else
306:       { //QDB does NOT have this episode, need to insert it
307:         tempQuoteList.insert(Pair<Q_L>(quoteStrn,quoteStrn));
308:         //If no errors (no throw) then insert tempQuoteList into QuoteDB
309:         QuoteDB.insert(Pair<D_B>(tmpQuote.getKey(),tempQuoteList));
310:       }
311:     }
312:     else
313:     { //Unable to get pair for some reason,
314:       keepReading = false;
315:     }
316:   } while(keepReading);
317:
318:   inFile.close(); //Close file
319: }
320:
321: void openFile(ifstream& inFile)
322: { //Variable Declarations
323:   string File_Name;
324:   bool tryAgain = true;
325:
326:   do
327:   {
328:     inFile.clear();
329:     cin.clear();   //Clear out any states of inFile and cin
330:
331:     cout << "Please enter the name of the data file:  ";
332:     getline(cin,File_Name);
333:
334:     if(cin.good())
335:     { //Read of filename was good
336:       inFile.open(File_Name.c_str());
337:
338:       if(inFile.good())
339:       { //input File opened without errors
340:         tryAgain = false;
341:       }
342:       else
343:       { //Error in opening file
344:         cout << "Error in opening file " << File_Name << endl << endl;
345:       }
346:     }
347:     else if(cin.eof())
348:     { //EOF condition - Print message and Exit
349:       printMessage(cout,fatalError);
350:       exit(1);
351:     }
352:     else
353:     { //Other Error
354:       cout << "Error in opening file " << File_Name << endl << endl;
355:     }
356:   } while(tryAgain);
357: }
358:
359: bool getPair(istream& ins, Quote& thisPair)
360: { //Variable Declarations
```

```
361:   string line;
362:   string EpisodeName, Quotation;
363:   bool runLoop = true;
364:   bool Success = true;
365:
366:
367:   //Read Episode Name
368:   do
369:   {
370:     getline(ins,line); //Read one line in from input stream
371:     if(ins.good())
372:     {
373:       eatSomeYummyWS(line);
374:
375:       if(line[0] == '-' && line[1] == '-')
376:       { //Line is proper episode name
377:         runLoop = false;
378:
379:         if(line[2] == '"')
380:         {
381:           EpisodeName = line.substr(3, line.size()-4); //Remove --"___"      //<<<
382:         }
383:         else
384:         {
385:           EpisodeName = line.substr(2, line.size()-2); //Remove start --
386:         }
387:
388:       }
389:       //else - Line is not episode name or blank, skip it and get the next line
390:     }
391:     else
392:     { //Line read was not good
393:       Success = false;
394:     }
395:   } while(runLoop && Success);
396:
397:   //Read Quote
398:   runLoop = true;  //reset flag
399:
400:   do
401:   {
402:     if(chkForEpisode(ins)) //Removes whitespace from line --ok??
403:     { //Next episode -- found, stop loop
404:       runLoop = false;
405:       if(Quotation.empty())
406:       { //Episode follows episode,
407:         Success = false;
408:       }
409:     }
410:     else
411:     { //Read in Line
412:       getline(ins,line);
413:       if(ins.good())
414:       { //Line read is good
415:         eatSomeYummyWS(line);
416:         if(line.empty())
417:         { //Line is emtpy, check to see if its after of before the quote
418:           if(!Quotation.empty())                                          ///<<<
419:           { //This prevents blank lines before a quote is entered from exiting
420:             runLoop = false;
```

```
421:           }
422:         }
423:         else
424:         { //Concatinate line onto Quotation
425:           line += "\n"; //Put newline on to end of line
426:           Quotation += line;
427:         }
428:       }
429:       else
430:       { //Error in read
431:         runLoop = false;
432:       }
433:     }
434:   } while(Success && runLoop);
435:
436:   if(Success)
437:   { //Check to see if Quotation is empty
438:     if(Quotation.empty())
439:     { //Quote is empty, bad read
440:       Success = false;
441:     }
442:     else
443:     { //Read was good, write into pair values
444:       thisPair.setKey(EpisodeName);
445:       thisPair.setEntry(Quotation);
446:     }
447:   }
448:
449:   return Success;
450: }
451:
452: void eatSomeYummyWS(string& line)
453: { //Variable Dec.
454:   int size = line.size();
455:   int i  = 0;
456:
457:   if (!line.empty())
458:   { //Itterate through the string to find first character that is not space
459:     while (isspace(line[i]) && i < size)
460:     { i++;  }
461:
462:     line = line.substr(i, size-i);
463:   }
464: }
465:
466: bool chkForEpisode(istream& ins)
467: { //Removes whitespace but not EOL from ins looking for --
468:
469:   //Variable Declarations
470:   char ch1, ch2;
471:   bool foundEpisode = false, loopCntrl = true;
472:
473:   //Read past any whitespace except EOL
474:   do
475:   {
476:     ins.get(ch1);
477:     if(!ins.good()) { loopCntrl = false; }
478:   } while((ch1 == ' ' || ch1 == '\t') && ch1 != '\n' && loopCntrl);
479:
480:   if(loopCntrl) //No read error
```

```
481:   { //Read next consecutive character
482:     ins.get(ch2);
483:     if(ins.good())
484:     { //both reads are good, but them back into the istream
485:       ins.putback(ch2);
486:       ins.putback(ch1);
487:       //Check for '--'
488:       if(ch1 == '-' && ch2 == '-') { foundEpisode = true; }
489:     }
490:   }
491:
492:   return foundEpisode;
493: }
494:
495:
496: void listQuotationsEpisode(DataBase& QuoteDB)
497: { //Variable Declarations
498:   QuoteList tempQuoteList;
499:   string EpisodeName, line;
500:   string tempQuoteStrn;
501:
502:   //Prompt for Episode Name
503:   printMessage(cout,episodePrompt);
504:   cin.clear(); cin.ignore(1000000,'\n');  //Clear stream before getting input
505:   getline(cin,line);
506:
507:   if(cin.good())
508:   { //Read of line good
509:     //Atempt to retrieve Episode List
510:
511:     if(line[2] == '"')
512:     {
513:       EpisodeName = line.substr(3, line.size()-4); //Remove --"___"      //<<<
514:     }
515:     else
516:     {
517:       EpisodeName = line.substr(2, line.size()-2); //Remove start --
518:     }
519:
520:
521:
522:     //if(EpisodeName[0] == '"')
523:     //{
524:      //  EpisodeName = EpisodeName.substr(1, EpisodeName.size()-2); //Remove "___"
    //<<<
525:     //}
526:
527:     tempQuoteList = QuoteDB.retrieveItem(EpisodeName);
528:     //If sucessful (no throw) then output Quotes
529:     cout << endl;
530:     tempQuoteList.gotoBeginning(); //Start at beginning of Quote List
531:
532:     while(!tempQuoteList.empty())
533:     { //Loop through list until empty, deleting elements one at a time
534:       tempQuoteStrn = tempQuoteList.getCurrent(); //Get first Quote
535:       cout << tempQuoteStrn << endl; //Print Quote
536:       tempQuoteList.erase(tempQuoteStrn);  //Remove quote from temp list
537:     }
538:   }
539:   else if(cin.eof())
```

```
540:   { //EOF error
541:     printMessage(cout,fatalError);
542:     exit(1);
543:   }
544:   else
545:   { //Other read error
546:     printMessage(cout,invalidDataError);
547:   }
548: }
549:
550: void listQuotationsWord(DataBase& QuoteDB)
551: { //Variable Declarations
552:   bool EndOfList = false;
553:   string SrchPhrase;
554:   string tempQuoteStrn;
555:   QuoteList tempQuoteList;
556:
557:   //Prompt for word or phrase to search for
558:   printMessage(cout,wordPrompt);
559:   cin.clear(); cin.ignore(1000000,'\n');  //Clear stream before getting input
560:   getline(cin,SrchPhrase);
561:
562:   if(cin.good())
563:   { //Read was good, now must search for phrase
564:     cout << endl;
565:
566:     //Goto Beginning of Episode List
567:     QuoteDB.gotoBeginning();
568:
569:     while(!EndOfList) //Loop through the Episodes
570:     {
571:       tempQuoteList = QuoteDB.getCurrent(); //Copy the quote list of 1st Ep.
572:
573:       tempQuoteList.gotoBeginning(); //Goto beginning of Quote List
574:
575:       while(!tempQuoteList.empty()) //Loop through the Quotes
576:       {
577:         tempQuoteStrn = tempQuoteList.getCurrent(); //Get a Quote
578:         tempQuoteList.erase(tempQuoteStrn); //Remove it from the List
579:         if(tempQuoteStrn.find(SrchPhrase) != string::npos)
580:         { //This Quote contains the Search Phrase - Print it
581:           cout << tempQuoteStrn << endl;
582:         }
583:       }
584:
585:       try
586:       { //Try to advance to next episode
587:         QuoteDB.next();
588:       }
589:       catch(LastElement& LE)
590:       { //If unable to advance to next episode, we're at the end of the list
591:         EndOfList = true;
592:       }
593:     } //End of QuoteDB loop
594:   }
595:   else if(cin.eof())
596:   { //EOF error
597:     printMessage(cout,fatalError);
598:     exit(1);
599:   }
```

```
600:   else
601:   { //Other read error
602:     //No Error message, do nothing
603:   }
604: }
605:
606: void listAllQuotations(DataBase& QuoteDB)
607: { //Variable Declarations
608:   bool EndOfList = false;
609:   string tempQuoteStrn;
610:   QuoteList tempQuoteList;
611:
612:   //Goto Beginning of Episode List
613:   QuoteDB.gotoBeginning();
614:
615:   cout << endl; //Put in first blank line ???
616:
617:   while(!EndOfList) //Loop through the Episodes
618:   {
619:     tempQuoteList = QuoteDB.getCurrent(); //Copy the quote list of 1st Ep.
620:
621:     tempQuoteList.gotoBeginning(); //Goto beginning of Quote List
622:
623:     while(!tempQuoteList.empty()) //Loop through the Quotes
624:     {
625:       tempQuoteStrn = tempQuoteList.getCurrent(); //Get a Quote
626:       tempQuoteList.erase(tempQuoteStrn); //Remove it from the List
627:       cout << tempQuoteStrn << endl; //Print the Quote
628:     }
629:
630:     try
631:     { //Try to advance to next episode
632:       QuoteDB.next();
633:     }
634:     catch(LastElement& LE)
635:     { //If unable to advance to next episode, we're at the end of the list
636:       EndOfList = true;
637:     }
638:   } //End of QuoteDB loop
639: }
640:
641: void addQuotation(DataBase& QuoteDB)
642: { //Variable Declarations
643:   Quote tmpQuote;
644:   string quoteStrn;
645:   QuoteList tempQuoteList;
646:
647:
648:
649:   //Prompt for New Quote
650:   printMessage(cout,newQuotePrompt);
651:   cin.clear(); cin.ignore(1000000,'\n');
652:   if(getPair(cin,tmpQuote))
653:   { //getPair was sucessful at getting Quote from console
654:
655:     quoteStrn = tmpQuote.getEntry(); //Store Quote itself into a temp string
656:
657:     //Check if episode exists
658:     if(QuoteDB.has(tmpQuote.getKey()))
659:     { //QDB has this Episode, current pointer set
```

```
660:       tempQuoteList = QuoteDB.getCurrent(); //Gets the Quote List of this Ep.
661:       //Attempts to insert new Quote into tempQuoteList
662:       tempQuoteList.insert(Pair<Q_L>(quoteStrn,quoteStrn));
663:       //If no errors (no throw) then replace tempQuoteList into QuoteDB
664:       QuoteDB.erase(tmpQuote.getKey()); //Remove Episode
665:       QuoteDB.insert(Pair<D_B>(tmpQuote.getKey(),tempQuoteList));
666:         //Re-inserts Quote list for episode back into QDB
667:     }
668:     else
669:     { //QDB does NOT have this episode, need to insert it
670:       tempQuoteList.insert(Pair<Q_L>(quoteStrn,quoteStrn));
671:       //If no errors (no throw) then insert tempQuoteList into QuoteDB
672:       QuoteDB.insert(Pair<D_B>(tmpQuote.getKey(),tempQuoteList));
673:     }
674:   }
675:   else if(cin.eof())
676:   { //EOF when reading console
677:     printMessage(cout,fatalError);
678:     exit(1);
679:   }
680:   else if(!cin.good())
681:   { //Other console state error
682:     //No Error message, do nothing
683:   }
684:   else
685:   { //Unable to get good Quote not due to stream state
686:     //No Error message, do nothing
687:   }
688:
689:   //Clear out Console
690:   cin.clear(); cin.ignore(1000000,'\n');
691: }
692:
693: void deleteQuotation(DataBase& QuoteDB)
694: { //Variable Declarations
695:   bool runLoop = true, Success = true;
696:   string line, EpisodeName, Quotation;
697:   //Further Declarations below in Delete Quote
698:
699:   //Prompt for Episode of Quote
700:   printMessage(cout,deletePrompt);
701:   cin.clear(); cin.ignore(1000000,'\n');
702:
703:   //Quote or Episode?
704:   if(chkForEpisode(cin)) //need to check stream state
705:   { //Episode to Delete
706:     if(cin.good())
707:     { //cin is at start of episode
708:       getline(cin,line); //get episode name
709:
710:       if(cin.good())
711:       { //Read of episdoe was good, try to delete episdoe
712:
713:         if(line[2] == '"')
714:         {
715:           EpisodeName = line.substr(3, line.size()-4); //Remove --"___"      //<<<
716:         }
717:         else
718:         {
719:           EpisodeName = line.substr(2, line.size()-2); //Remove start --
```

```
720:         }
721:
722:         QuoteDB.erase(EpisodeName); //Will throw if it DNE
723:       }
724:     }
725:
726:     if(cin.eof())
727:     { //EOF in --
728:       printMessage(cout,fatalError);
729:       exit(1);
730:     }
731:   }
732:   else if(cin.eof())
733:   { //EOF when reading console
734:     printMessage(cout,fatalError);
735:     exit(1);
736:   }
737:   else if(!cin.good())
738:   { //Other read error
739:     //No Error message, do nothing
740:   }
741:   else
742:   { //Quote to Delete, read in quote
743:
744:     do
745:     {
746:       if(chkForEpisode(cin)) //Removes whitespace from line --ok??
747:       { //Next episode -- found, stop loop
748:         runLoop = false;
749:         if(Quotation.empty())
750:         { //Episode follows episode,
751:           Success = false;
752:         }
753:       }
754:       else
755:       { //Read in Line
756:         getline(cin,line);
757:         if(cin.good())
758:         { //Line read is good
759:           eatSomeYummyWS(line);
760:           if(line.empty())
761:           { //Line is emtpy, check to see if its after of before the quote
762:             if(!Quotation.empty())
///<<<
763:             { //This prevents blank lines before a quote is entered from exiting
764:               runLoop = false;
765:             }
766:           }
767:           else
768:           { //Concatinate line onto Quotation
769:             line += "\n"; //Put newline on to end of line
770:             Quotation += line;
771:           }
772:         }
773:         else
774:         { //Error in read
775:           runLoop = false;
776:         }
777:       }
778:     } while(Success && runLoop);
```

```
779:
780:     if(!cin.good())
781:     { //error in quote read
782:       if(cin.eof())
783:       {
784:         printMessage(cout,fatalError);
785:         exit(1);
786:       }
787:     }
788:     else if(Success && !Quotation.empty())
789:     { //Quote good, attempt to delete any matching quote
790:       //Variable Declarations
791:       bool loopEp=true, loopQot=true;
792:
793:       QuoteDB.gotoBeginning(); //Start at beginning of QuoteDB
794:
795:       //Loop through episodes
796:       while(loopEp)
797:       {
798:         QuoteList tempQuoteList(QuoteDB.getCurrent()); //Current QL of QDB
799:
800:         EpisodeName = QuoteDB.getCurrKey();
801:
802:         tempQuoteList.gotoBeginning(); //Start at beginning
803:
804:         if(!tempQuoteList.empty()) { loopQot = true; }
805:
806:         while(loopQot)
807:         { //Loop through quote list
808:
809:           //Test Quote
810:           if(Quotation == tempQuoteList.getCurrent())
811:           { //tempQuoteList contains quote to delete
812:             tempQuoteList.erase(Quotation); //delete quotation from templist
813:             loopQot = false; //No need to check rest of quote string
814:           }
815:           else
816:           { //Try to look at next quote in templist
817:             try
818:             {
819:               tempQuoteList.next();
820:             }
821:             catch(LastElement& LE)
822:             { //Cannot move past end of tempQuoteList
823:               loopQot = false;
824:             }
825:           }
826:         }
827:
828:         //tempQuoteList now has had quotes deleted, need to reinsert into QDB
829:
830:         QuoteDB.erase(EpisodeName); //Temporarily delete Episode
831:
832:         if(!tempQuoteList.empty())
833:         { //As long as tempQuotelist is not empty, reinsert it into QDB
834:           QuoteDB.insert(Pair< D_B >(EpisodeName,tempQuoteList));
835:         }
836:
837:         //Try to look at next Episode in QDB
838:         try
```

```
839:         {
840:           QuoteDB.next();
841:         }
842:         catch(LastElement& LE)
843:         { //Cannot move past end of QDB
844:           loopEp = false;
845:         }
846:
847:       }
848:     }
849:   }
850: }
851:
852: /*****************************************************************************
853: HONOR CODE:
854:       I have neither given nor recieved aid on this project, nor have I
855:       concealed any violations of the honor code.
856:                                                          JAMES GLETTLER
857: *****************************************************************************/
858:
```

```
 1: /************************TEMPLATED CLASS INTERFACE*****************************
 2: Course:     EECS 280, Winter 2002       Section:     005
 3: Author:     James Glettler              Uniquename:  JGLETTLE
 4: Assignment: Project05 - B (Template)
 5: Filename:   associationlist.h           Date:        09 April 2002
 6: Version:    v1.21a
 7: Related:    associationlist.cpp node.h pair.h
 8: Descrip:    N/A
 9: Notes:      N/A
10: ********************************************************************************/
11: #ifndef ASLST
12: #define ASLST
13: //======================Preprocessor Directives================================
14: #include "node.h"
15: #include <string> //For EmptyList class
16: using namespace std;
17: //==========================Global Constants====================================
18:
19: //==================Class Definition of AssociationList=======================
20: template<class KeyType, class EntryType> //Template prefix for two var types
21: class AssociationList
22: {
23:   public:
24:     /* Constructors for the class */
25:       AssociationList();
26:         //Constructor creates empty AssociationList
27:       AssociationList(const AssociationList& cpyFrom);
28:         //Deep Copy Constructor
29:       ~AssociationList();
30:         //Deconstructor deletes all allocated variables
31:
32:     /* Testing Member Functions */
33:       bool has (KeyType key);
34:         //Returns true if "key" exists in list, points current to that location
35:         //Otherwise does not change current and wont throw except'n if empty
36:       bool empty() const;
37:         //Returns true if List is empty
38:       EntryType retrieveItem(KeyType key);
39:         //Returns EntryType in the list corresponding to "key" or else
40:         //throws an exception of type EmptyList or NonExistentKey
41:         //Current changes to found item if found.
42:
43:     /* Modification Member Functions */
44:       void insert(Pair<KeyType, EntryType> pair);
45:         //adds entry to list in lexicographical order,returns true if successful
46:         //sets current pointer to inserted entry, or throws DuplicateKey
47:       void erase(KeyType key);
48:         //deletes element of "key" or throws either NonExistentKey or EmptyList
49:
50:     /* Navigation Member Functions */
51:       void gotoBeginning();
52:         //Sets current pointer to start of the list
53:       void next();
54:         //Sets current to the next entry in the list or throws LastElement
55:       void previous();
56:         //Sets current to the previous entry in the list or throws LastElement
57:       EntryType getCurrent();
58:         //Returns EntryType of where the current pointer is pointing
59:         //or it throws EmptyList
60:
```

```
 61:
 62:       KeyType getCurrKey(); //ADDED FOR PROJ5C
 63:
 64:     /*Overloaded Operators*/
 65:       void operator = (const AssociationList& cpyFrom);
 66:         //Assigns cpyFrom to cpyTo with a deep copy
 67:
 68:     /*Friend Functions*/
 69:       friend bool operator ==<>(const AssociationList<KeyType,EntryType>& ListA,
 70:                                 const AssociationList<KeyType,EntryType>& ListB);
 71:         //Returns true if all members of A are equal to B and lengths are equal
 72:         //Does NOT check to see if current points to the same Node in order
 73:
 74:   private:
 75:     int length; //Number of Nodes
 76:     Node<KeyType, EntryType>* start;
 77:     Node<KeyType, EntryType>* current;
 78: };
 79:
 80: //====================Class Definitions of Exceptions=========================
 81: class EmptyList
 82: {
 83:   public:
 84:     EmptyList();
 85:       //Default constructor, no string initialized
 86:     EmptyList(string theMessage);
 87:       //Explicit constructor sets Message to theMessage
 88:     string getMessage() const;
 89:       //Returns the message contained by the EmptyList exception class
 90:
 91:   private:
 92:     string Message;
 93: };
 94:
 95: template<class KeyType>
 96: class NonExistentKey
 97: {
 98:   public:
 99:     NonExistentKey();
100:       //Default Constructor
101:     NonExistentKey(KeyType errKey);
102:       //Explicit Constructor set Key to errKey
103:     KeyType getKey() const;
104:       //Returns the Key that had the error
105:
106:   private:
107:     KeyType key;
108: };
109:
110: class LastElement
111: {
112:   //Default constructor - automatic no needed functions
113: };
114:
115: template<class KeyType>
116: class DuplicateKey
117: {
118:   public:
119:     DuplicateKey();
120:       //Default Constructor
```

```
121:     DuplicateKey(KeyType errKey);
122:       //Explicit Constructor set Key to errKey
123:     KeyType getKey() const;
124:       //Returns the Key that had the error
125:
126:   private:
127:     KeyType key;
128:
129: };
130:
131: //====================(De)Constructor Function Definitions=====================
132: template<class KeyType, class EntryType>
133: AssociationList<KeyType, EntryType>::AssociationList()
134: { //Constructor creates empty AssociationList
135:   length = 0; //Setup empty AL
136:   start = NULL;
137:   current = NULL;
138: }
139:
140: template<class KeyType, class EntryType>
141: AssociationList<KeyType, EntryType>::AssociationList
142:                          (const AssociationList<KeyType, EntryType>& cpyFrom)
143: { //Deep Copy Constructor
144:   //Variable Declarations
145:   Node<KeyType, EntryType>* ToPrev = NULL;
146:   Node<KeyType, EntryType>* ToTmp = NULL;
147:   Node<KeyType, EntryType>* FromTmp = NULL;
148:
149:   //Check if cpyFrom is empty
150:   if(cpyFrom.empty())
151:   { //isEmpty so make pointers NULL and length 0
152:     start = NULL;
153:     current = NULL;
154:     length = 0;
155:   }
156:   else
157:   { //Copy First node
158:     FromTmp = cpyFrom.start; //Get location of first node (From)
159:     ToTmp = new Node<KeyType, EntryType>; //Create First node (To)
160:     start = ToTmp; //Set start pointer (To)
161:     ToTmp->setData(FromTmp->getData()); //Copy Data into Node
162:     current = ToTmp; //Set first node as current, it will be changed later
163:     length = 1; //Set start length
164:
165:     FromTmp = FromTmp->getNext(); //Advance FromTmp to next node
166:
167:     //Loop to copy rest of Nodes
168:     while(FromTmp != NULL) //Until the end of cpyFrom List
169:     {
170:       ToPrev = ToTmp; //Keep track of last Node that was copied
171:       ToTmp = new Node<KeyType, EntryType>; //Create a New node
172:       ToTmp->setData(FromTmp->getData()); //Copy Data
173:       ToTmp->setPrevious(ToPrev); //Prev <- NEW
174:       ToPrev->setNext(ToTmp);  //Prev -> NEW if Prev
175:       if(cpyFrom.current == FromTmp) //Check if node is "current"
176:       { current = ToTmp; }
177:       length++; //Increment Length
178:       FromTmp = FromTmp->getNext(); //Advance FromTmp
179:     }//End of Copy Loop
180:   }
```

```
181: }
182:
183: template<class KeyType, class EntryType>
184: AssociationList<KeyType, EntryType>::~AssociationList()
185: { //Deconstructor deletes all allocated variables
186:   //cerr<<"DESTRUCTOR!!"<< endl;
187:   //cerr<<"|Length/Start/Current:"<<length<<" / "<< start<<" / "<<current<<endl;
188:
189:   current = start; //Set current to starting point
190:
191:   for(/**/ ; length > 0 ; length--) //Cycle for length of AssociationList
192:   { //Loop will not run if List has no points
193:     current = start; //Set current pointer to starting point
194:     start = start->getNext(); //Set new start (after deletion) to Next
195:     delete current; //Delete current node
196:   }
197: }
198: //====================Public Member Function Definitions=====================
199: /* Testing Member Functions */
200: template<class KeyType, class EntryType>
201: bool AssociationList<KeyType, EntryType>::has (KeyType key)
202: { //Returns true if "key" exists in list, points current to that location
203:   //Variable Declarations
204:   Node<KeyType, EntryType>* lookAt = NULL;
205:   bool keyExists = false;
206:
207:   if(empty())
208:   { //List is empty, return false
209:     keyExists = false;
210:   }
211:   else
212:   { //List is not empty, check to see if key would be before or after CURRENT
213:     //This is to decrease the amount of search time for big lists
214:     if(key == current->getData().getKey())
215:     { //Current is already pointing at locatin of Key
216:       keyExists = true;
217:     }
218:     else if(key < current->getData().getKey())
219:     { //Set lookAt to beginning
220:       lookAt = start;
221:     }
222:     else
223:     { //Set lookAt to Node after current
224:       lookAt = current->getNext();
225:     }
226:     //Loop through remaining portion of list looking for key (start at lowest)
227:     while(lookAt != NULL && lookAt != current && !keyExists)
228:     { //check value of key
229:       if(key == lookAt->getData().getKey())
230:       { //Key exists, set current pointer
231:         keyExists = true;
232:         current = lookAt;
233:       }
234:       else if(key < lookAt->getData().getKey())
235:       { //You passed where it would have been, stop looking, it DNE
236:         lookAt = NULL; //Set to NULL to kick out of while loop
237:       }
238:       else
239:       { //Key is greater than lookAt.Key, move pointer forward
240:         lookAt = lookAt->getNext();
```

```
241:       }
242:     }
243:   }
244:
245:   return keyExists;
246: }
247:
248: template<class KeyType, class EntryType>
249: bool AssociationList<KeyType, EntryType>::empty() const
250: { //Returns true if List is empty
251:   return (length == 0);
252: }
253:
254: template<class KeyType, class EntryType>
255: EntryType AssociationList<KeyType, EntryType>::retrieveItem(KeyType key)
256: { //Returns EntryType in the list corresponding to "key" or else
257:   //throws an exception of type EmptyList or NonExistentEntry
258:   //changes current to key if found.
259:   //Variable Declarations
260:   EntryType Entry;
261:
262:   if(empty())
263:   { //list is empty, throw EmptyList
264:     throw EmptyList("AssociationList::retrieveItem");
265:   }
266:   else
267:   { //Look for item and if so, return it
268:     if(has(key)) //If the list has the key (sets current to the proper loc.
269:     { //Then return Entry Type
270:       Entry = current->getData().getEntry(); //Retrive Entry Data
271:     }
272:     else
273:     { //Key DNE -> Throw exception of type NonExistantKey
274:       throw NonExistentKey<KeyType>(key);
275:     }
276:   }
277:   return Entry;
278: }
279:
280: /* Modification Member Functions */
281: template<class KeyType, class EntryType>
282: void AssociationList<KeyType, EntryType>::insert(Pair<KeyType, EntryType> pair)
283: { //adds entry to list in lexicographical order,returns true if successful
284:   //sets current pointer to inserted entry, or throws DuplicateKey
285:   //Variable Declarations
286:   KeyType key = pair.getKey(); //Get Key from Pair
287:   Node<KeyType, EntryType>* lookAt = current;
288:   bool insertHere = false;
289:
290:   //Check for existance of Key or if list is empty
291:   if(has(key))
292:   { //List already has Key -> throw DuplicateKey
293:     //Should Current be returned to original condition before throw ???
294:     throw DuplicateKey<KeyType>(key);
295:   }
296:   else if(empty())
297:   { //List is empty Plug in values
298:     start = new Node<KeyType, EntryType>;
299:     current = start;
300:     length = 1;
```

```
301:     current->setData(pair);
302:   }
303:   else
304:   { //Else if key DNE then look for where to insert key.  Search
305:
306:     lookAt = start; //start searching at the beginning
307:                     //Possibly include fnct to search from current
308:     //Search with while loop
309:     while(lookAt != NULL && !insertHere)
310:     {
311:       if(key < lookAt->getData().getKey())
312:       { //Key is less than key at lookAt location, put before lookAt
313:         insertHere = true;
314:       }
315:       else
316:       { //Move lookAt one forward in list
317:         current = lookAt; //Store last lookAt value to current
318:         lookAt = lookAt->getNext();
319:       }
320:     } //End of searching while loop
321:
322:     //Insert Node
323:     if(lookAt == NULL && !insertHere)
324:     { //insert Node onto end of list and crosslink pointers
325:       lookAt = new Node<KeyType, EntryType>; //Create new node with NULL pntr
326:       lookAt->setData(pair); //Copy in data from pair to Node
327:       lookAt->setPrevious(current); //Prev <- Current
328:       current->setNext(lookAt); //Prev -> Current
329:       //Now change current to point to new inserted node
330:       current = lookAt;
331:     }
332:     else                                                          //WORKS
333:     { //insert Node before this (lookAt) node and cross link pointers
334:       current = new Node<KeyType, EntryType>; //Create new Node at Current
335:       current->setData(pair); //Copy in data from pair
336:       current->setNext(lookAt); //Set Curr -> Next
337:       current->setPrevious(lookAt->getPrevious()); //Set Prev <- Curr
338:       if (current->getPrevious() != NULL) //If not at start, Set Prev -> Curr
339:       { current->getPrevious()->setNext(current); }
340:       else //New node is new start of List
341:       { start = current; }
342:       lookAt->setPrevious(current); //Set Curr <- Next
343:     }
344:
345:     //Increment length
346:     length++;
347:   }
348: }
349:
350: template<class KeyType, class EntryType>
351: void AssociationList<KeyType, EntryType>::erase(KeyType key)
352: { //deletes element of "key" or throws either NonExistentKey or EmptyList
353:   Node<KeyType, EntryType>* TmpNext, TmpPrev;
354:
355:   if(empty())
356:   { //list is empty, throw EmptyList
357:     throw EmptyList("AssociationList::erase");
358:   }
359:   else
360:   { //Check for existance of key, set current to that point if it exists
```

```
361:     if(has(key)) //sets current to Node to be removed
362:     { //Set Temp pointers
363:       Node<KeyType, EntryType>* TmpNext = current->getNext();
364:       Node<KeyType, EntryType>* TmpPrev = current->getPrevious();
365:
366:       if(TmpNext != NULL) //Not at Tail End, set pointer for NEXT
367:       { TmpNext->setPrevious(TmpPrev); }
368:       if(TmpPrev != NULL) //Not at Head, set pointer for PREV
369:       { TmpPrev->setNext(TmpNext); }
370:       else //Deleting start of List, reset start to NEXT
371:       { start = TmpNext; }
372:
373:       //Safe to delete Node and decrement length
374:       delete current;
375:       length--;
376:
377:       //Set current to beginning of list
378:       current = start;
379:     }
380:     else
381:     { //Key DNE -> Throw exception of type NonExistantKey
382:       throw NonExistentKey<KeyType>(key);
383:     }
384:   }
385: }
386:
387:
388: /* Navigation Member Functions */
389: template<class KeyType, class EntryType>
390: void AssociationList<KeyType, EntryType>::gotoBeginning()
391: { //Sets current pointer to start of the list
392:   current = start;
393: }
394:
395: template<class KeyType, class EntryType>
396: void AssociationList<KeyType, EntryType>::next()
397: { //Sets current to the next entry in the list or throws LastElement
398:   Node<KeyType, EntryType>* next;
399:   if(empty())
400:   { throw LastElement(); }
401:   else
402:   { next = current->getNext(); } //pointer to next is set from curent NODE
403:
404:   if(next == NULL)
405:   { //At the end of List, throw LastElement
406:     throw LastElement();
407:   }
408:   else
409:   { //Set current to next
410:     current = next;
411:   }
412: }
413:
414: template<class KeyType, class EntryType>
415: void AssociationList<KeyType, EntryType>::previous()
416: { //Sets current to the previous entry in the list or throws LastElement
417:   Node<KeyType, EntryType>* previous;
418:   if(empty())
419:   { throw LastElement(); }
420:   else
```

```
421:   { previous = current->getPrevious(); } //pointer to prev set from curent NODE
422:
423:   if(previous == NULL)
424:   { //At the end of List, throw LastElement
425:     throw LastElement();
426:   }
427:   else
428:   { //Set current to next
429:     current = previous;
430:   }
431: }
432:
433: template<class KeyType, class EntryType>
434: EntryType AssociationList<KeyType, EntryType>::getCurrent()
435: { //Returns EntryType of where the current pointer is pointing
436:   //or it throws EmptyList
437:   if(empty())
438:   { //list is empty, throw EmptyList
439:     throw EmptyList("AssociationList::getCurrent");
440:   }
441:   else
442:   { //Otherwise, return Entry(EntryType) of Data(Pair) of Current(Node)
443:     return current->getData().getEntry();
444:   }
445: }
446:
447: template<class KeyType, class EntryType>
448: KeyType AssociationList<KeyType, EntryType>::getCurrKey()
449: {
450:   if(empty())
451:   { //list is empty, throw EmptyList
452:     throw EmptyList("AssociationList::getCurrKey");
453:   }
454:   else
455:   { //Otherwise, return Entry(EntryType) of Data(Pair) of Current(Node)
456:     return current->getData().getKey();
457:   }
458: }
459:
460: /*Overloaded Operators*/
461: template<class KeyType, class EntryType>
462: void AssociationList<KeyType, EntryType>::operator =
463:                              (const AssociationList<KeyType, EntryType>& cpyFrom)
464: { //Assigns cpyFrom to cpyTo with a deep copy
465:   //First Delete everything (~AssociationList())
466:   //Then deep copy (AssociationList(const AssociationList& cpyFrom)
467:
468:   //Variable Declarations for copy part
469:   Node<KeyType, EntryType>* ToPrev = NULL;
470:   Node<KeyType, EntryType>* ToTmp = NULL;
471:   Node<KeyType, EntryType>* FromTmp = NULL;
472:
473:
474:   /* ++++++Copy Code from destructor++++++ */
475:   current = start; //Set current to starting point
476:
477:   for(/**/ ; length > 0 ; length--) //Cycle for length of AssociationList
478:   { //Loop will not run if List has no points
479:     current = start; //Set current pointer to starting point
480:     start = start->getNext(); //Set new start (after deletion) to Next
```

```
481:     delete current; //Delete current node
482:   }
483:
484:   /* ++++++Copy Code from Copy Constructor++++++ */
485:   //Check if cpyFrom is empty
486:   if(cpyFrom.empty())
487:   { //isEmpty so make pointers NULL and length 0
488:     start = NULL;
489:     current = NULL;
490:     length = 0;
491:   }
492:   else
493:   { //Copy First node
494:     FromTmp = cpyFrom.start; //Get location of first node (From)
495:     ToTmp = new Node<KeyType, EntryType>; //Create First node (To)
496:     start = ToTmp; //Set start pointer (To)
497:     ToTmp->setData(FromTmp->getData()); //Copy Data into Node
498:     current = ToTmp; //Set first node as current, it will be changed later
499:     length = 1; //Set start length
500:
501:     FromTmp = FromTmp->getNext(); //Advance FromTmp to next node
502:
503:     //Loop to copy rest of Nodes
504:     while(FromTmp != NULL) //Until the end of cpyFrom List
505:     {
506:       ToPrev = ToTmp; //Keep track of last Node that was copied
507:       ToTmp = new Node<KeyType, EntryType>; //Create a New node
508:       ToTmp->setData(FromTmp->getData()); //Copy Data
509:       ToTmp->setPrevious(ToPrev); //Prev <- NEW
510:       ToPrev->setNext(ToTmp);  //Prev -> NEW if Prev
511:       if(cpyFrom.current == FromTmp) //Check if node is "current"
512:       { current = ToTmp; }
513:       length++; //Increment Length
514:       FromTmp = FromTmp->getNext(); //Advance FromTmp
515:     }//End of Copy Loop
516:   }
517: }
518:
519: //=========================Friend Function Definitions=========================
520: template<class KeyType, class EntryType>
521: bool operator == ( const AssociationList<KeyType, EntryType>& ListA,
522:                    const AssociationList<KeyType, EntryType>& ListB)
523: { //Returns true if all members of A are equal to B
524:   //Variable Declaration
525:   bool isEqual = true;
526:   Node<KeyType, EntryType>* NodeA = NULL;
527:   Node<KeyType, EntryType>* NodeB = NULL;
528:
529:   //Check Lengths
530:   if(ListA.length == ListB.length)
531:   { //Lengths are equal, so far so good
532:     isEqual = true;
533:   }
534:   else
535:   { //Lengths are unequal so two Lists are not equal
536:     isEqual = false;
537:   }
538:
539:   //Cycle through Nodes, check each for == of keys
540:   if(isEqual && (ListA.length != 0))
```

```
541:   { //Set Node pointers to start of lists
542:     NodeA = ListA.start;
543:     NodeB = ListB.start;
544:     for(int i = 0 ; i < ListA.length && isEqual ; i++)
545:     {
546:       if(NodeA->getData() == NodeB->getData()) //Pair==Pair compares only keys
547:       { //Node Pairs (Keys) are equal, advance to next node in each list
548:         NodeA = NodeA->getNext();
549:         NodeB = NodeB->getNext();
550:       }
551:       else
552:       { //Node Pair is unequal, therefore List is unequal
553:         isEqual = false;
554:       }
555:     }
556:   }
557:
558:   return isEqual;
559: }
560:
561: //============================================================================
562: //==============Member Function Definitions of Exception Classes===============
563: EmptyList::EmptyList()
564: { //Default constructor, no string initialized
565:   //Do nothing
566: }
567: EmptyList::EmptyList(string theMessage)
568: { //Explicit constructor sets Message to theMessage
569:   Message = theMessage;
570: }
571: string EmptyList::getMessage() const
572: { //Returns the message contained by the EmptyList exception class
573:   return Message;
574: }
575:
576: template<class KeyType>
577: NonExistentKey<KeyType>::NonExistentKey()
578: { //Default Constructor
579: }
580: template<class KeyType>
581: NonExistentKey<KeyType>::NonExistentKey(KeyType errKey)
582: { //Explicit Constructor set Key to errKey
583:   key = errKey;
584: }
585: template<class KeyType>
586: KeyType NonExistentKey<KeyType>::getKey() const
587: { //Returns the Key that had the error
588:   return key;
589: }
590:
591: template<class KeyType>
592: DuplicateKey<KeyType>::DuplicateKey()
593: { //Default Constructor
594: }
595: template<class KeyType>
596: DuplicateKey<KeyType>::DuplicateKey(KeyType errKey)
597: { //Explicit Constructor set Key to errKey
598:   key = errKey;
599: }
600: template<class KeyType>
```

```
601: KeyType DuplicateKey<KeyType>::getKey() const
602: { //Returns the Key that had the error
603:   return key;
604: }
605:
606: /*********************************************************************************
607: HONOR CODE:
608:       I have neither given nor recieved aid on this project, nor have I
609:       concealed any violations of the honor code.
610:                                                              JAMES GLETTLER
611: *********************************************************************************/
612: #endif
```

```
 1: /************************TEMPLATED CLASS INTERFACE*****************************
 2: Course:     EECS 280, Winter 2002       Section:     005
 3: Author:     James Glettler              Uniquename:  JGLETTLE
 4: Assignment: Project05 - B (Template)
 5: Filename:   node.h                      Date:        09 April 2002
 6: Version:    v1.1a
 7: Related:    node.cpp
 8: Descrip:    N/A
 9: Notes:      N/A
10: ************************************************************************/
11:
12: //======================Preprocessor Directives==============================
13: #ifndef NULL
14: #define NULL 0
15: #endif
16:
17: #ifndef NODE
18: #define NODE
19: #include "pair.h"
20: using namespace std;
21:
22: //==========================Global Constants=================================
23: //==========================Class Definition=================================
24: template<class KeyType, class EntryType> //Template prefix for two var types
25: class Node
26: {
27:   public:
28:     /* Constructors for the class */
29:     Node();
30:       //Default constructor sets up Node with all pointers to NULL, no data
31:
32:     /* Constant member functions: getters*/
33:     Pair<KeyType, EntryType> getData() const;
34:       //Returns datatype PAIR of data stored in NODE
35:     Node* getNext() const;
36:       //Returns a pointer to the NEXT NODE
37:     Node* getPrevious() const;
38:       //Returns a pointer to the PREVIOUS NODE
39:
40:     /* Modification member functions: setters */
41:     void setData(Pair<KeyType, EntryType> vals);
42:       //Sets data in NODE to vals of type PAIR
43:     void setNext(Node* newNext);
44:       //Sets NEXT NODE pointer to newNext
45:     void setPrevious(Node* newPrev);
46:       //Sets PREVIOUS NODE pointer to newPrev
47:
48:   private:
49:     Pair<KeyType, EntryType> data;
50:     Node* next;
51:     Node* previous;
52:     Node* current;
53: };
54:
55: //====================(De)Constructor Function Definitions====================
56: template<class KeyType, class EntryType>
57: Node<KeyType, EntryType>::Node()
58: { //Default Constructor, set all pointers to null
59:   //Pair is initialized by the Pair constructor
60:   next = NULL;
```

```
 61:   previous = NULL;
 62: }
 63:
 64: //====================Public Member Function Definitions=====================
 65: /* Constant member functions: getters*/
 66: template<class KeyType, class EntryType>
 67: Pair<KeyType, EntryType> Node<KeyType, EntryType>::getData() const
 68: { //Returns datatype PAIR of data stored in NODE
 69:   return data;
 70: }
 71:
 72: template<class KeyType, class EntryType>
 73: Node<KeyType, EntryType>* Node<KeyType, EntryType>::getNext() const
 74: { //Returns a pointer to the NEXT NODE
 75:   return next;
 76: }
 77:
 78: template<class KeyType, class EntryType>
 79: Node<KeyType, EntryType>* Node<KeyType, EntryType>::getPrevious() const
 80: { //Returns a pointer to the PREVIOUS NODE
 81:   return previous;
 82: }
 83:
 84: /* Modification member functions: setters */
 85: template<class KeyType, class EntryType>
 86: void Node<KeyType, EntryType>::setData(Pair<KeyType, EntryType> vals)
 87: { //Sets data in NODE to vals of type PAIR
 88:   data = vals;
 89: }
 90:
 91: template<class KeyType, class EntryType>
 92: void Node<KeyType, EntryType>::setNext(Node* newNext)
 93: { //Sets NEXT NODE pointer to newNext
 94:   next = newNext;
 95: }
 96:
 97: template<class KeyType, class EntryType>
 98: void Node<KeyType, EntryType>::setPrevious(Node* newPrev)
 99: { //Sets PREVIOUS NODE pointer to newPrev
100:   previous = newPrev;
101: }
102: /*****************************************************************************
103: HONOR CODE:
104:       I have neither given nor recieved aid on this project, nor have I
105:       concealed any violations of the honor code.
106:                                                          JAMES GLETTLER
107: *****************************************************************************/
108: #endif
```

```
 1: /************************TEMPLATED CLASS INTERFACE*****************************
 2: Course:     EECS 280, Winter 2002       Section:     005
 3: Author:     James Glettler              Uniquename:  JGLETTLE
 4: Assignment: Project05 - B (Template)
 5: Filename:   pair.h                      Date:        09 April 2002
 6: Version:    v1.21a
 7: Related:    NONE
 8: Descrip:    N/A
 9: Notes:      N/A
10: ********************************************************************************/
11: #ifndef PAIR
12: #define PAIR
13: //=======================Preprocessor Directives================================
14:
15: using namespace std;
16: //==========================Global Constants====================================
17:
18: //==========================Class Definition====================================
19: template<class KeyType, class EntryType> //Template prefix for two var types
20: class Pair
21: {
22:   public:
23:     /* Constructors for the class */
24:     Pair();
25:       //Default Constructor
26:     Pair(KeyType keyVal,EntryType entryVal);
27:       //Explicit Constructor
28:     Pair(const Pair& CpyFrom);
29:       //Explicit Constructor
30:
31:     /* Constant member functions: getters*/
32:     KeyType getKey() const;
33:       //Returns the Key of the pair (KeyType)
34:     EntryType getEntry() const;
35:       //Returns the Entry of the pair (EntryType)
36:
37:     /* Modification member functions: setters */
38:     void setKey(KeyType newValue);
39:       //Sets the Key to newValue
40:     void setEntry(EntryType newEntry);
41:       //Sets the Entry to newEntry
42:
43:     /*Friend Functions*/
44:     friend bool operator ==<> ( const Pair<KeyType, EntryType>& pairA ,
45:                                 const Pair<KeyType, EntryType>& pairB );
46:       //Returns true iff Keys of both pairs are equal
47:       //--->> CONFUSED ON TEMPLATING FRIEND FUNCTIONS, why <>
48:
49:   private:
50:     KeyType key;
51:     EntryType entry;
52: };
53:
54: //===================(De)Constructor Function Definitions====================
55: template<class KeyType, class EntryType>
56: Pair<KeyType, EntryType>::Pair()
57: { //Default Constructor
58:   //No default conditions
59: }
60:
```

```
 61: template<class KeyType, class EntryType>
 62: Pair<KeyType, EntryType>::Pair(KeyType keyVal,EntryType entryVal)
 63: { //Explicit Constructor
 64:   key = keyVal;
 65:   entry = entryVal;
 66: }
 67:
 68: template<class KeyType, class EntryType>
 69: Pair<KeyType, EntryType>::Pair(const Pair& CpyFrom)
 70: { //Explicit Constructor
 71:   key = CpyFrom.key;
 72:   entry = CpyFrom.entry;
 73: }
 74: //====================Public Member Function Definitions=====================
 75: /* Constant member functions: getters*/
 76: template<class KeyType, class EntryType>
 77: KeyType Pair<KeyType, EntryType>::getKey() const
 78: { //Returns the Key of the pair (KeyType)
 79:   return key;
 80: }
 81:
 82: template<class KeyType, class EntryType>
 83: EntryType Pair<KeyType, EntryType>::getEntry() const
 84: { //Returns the Entry of the pair (EntryType)
 85:   return entry;
 86: }
 87:
 88: /* Modification member functions: setters */
 89: template<class KeyType, class EntryType>
 90: void Pair<KeyType, EntryType>::setKey(KeyType newValue)
 91: { //Sets the Key to newValue
 92:   key = newValue;
 93: }
 94:
 95: template<class KeyType, class EntryType>
 96: void Pair<KeyType, EntryType>::setEntry(EntryType newEntry)
 97: { //Sets the Entry to newEntry
 98:   entry = newEntry;
 99: }
100:
101: //=========================Friend Function Definitions=========================
102: /*Friend Functions*/
103: template<class KeyType, class EntryType>
104: bool operator == (const Pair<KeyType, EntryType>& pairA,
105:                   const Pair<KeyType, EntryType>& pairB)
106: { //Returns true iff Keys of both pairs are equal
107:   return (pairA.key == pairB.key);
108: }
109:
110:
111: /*****************************************************************************
112: HONOR CODE:
113:       I have neither given nor recieved aid on this project, nor have I
114:       concealed any violations of the honor code.
115:                                                          JAMES GLETTLER
116: *****************************************************************************/
117: #endif
```