```
printf \\n\\n\\n          #print three blank lines



cat p7.sh              #display the shell script file for the program
#!/bin/bash

set -v                  #turn on echo
printf \\n\\n\\n          #print three blank lines
cat p7.sh              #display the shell script file for the program
printf \\f              #issue a form feed (top of a new page)
cat -b p7.java         #display the source file with line numbers
cat -b p7b.java          #display the source file with line numbers
:                       #null command
:
:
echo Running p7
javac p7.java          #compile the java file
java p7            #execute the file from the current directory
echo Running p7b
javac p7b.java          #compile the java file
java p7b           #execute the file from the current directory
:
:
:
date                    #print the date
printf \\f              #issue a form feed (top of a new page)

cat -b p7.java         #display the source file with line numbers
     1  /*
     2  -------------------------------------------------------------------
     3  PROGRAM NAME: Program 7
     4  PROGRAMMER:   Samuel Jentsch
     5  CLASS:        CSC 241.001, Fall 2013
     6  INSTRUCTOR:   Dr. D. Dunn
     7  DATE STARTED: November 18, 2013
     8  DUE DATE:     November 20, 2013
     9  REFERENCES:   Computer Science
    10                Data Abstraction and Problem Solving with Java
    11                Janet J. Prichard & Frank M. Carrano
    12              Dr. Dunn: assignment information sheet

    13  PROGRAM PURPOSE:
    14  a. This program reads in a series of unique words stored a file.
    15  b. The program stores these words in a hash table based on a hash
    16     key conversion.

    17  VARIABLE BANK:
    18              hTable - HashTable, stores the values read from the file and
    19                                   handles their storage.

    20  ADTs:
    21    Hash Table

    22  FILES USED:
    23    p7.dat - a file containing unique strings.

    24  -------------------------------------------------------------------
```

```
 */
import java.io.*;
import java.util.*;

public class p7 {

	static HashTable hTable;

	public static void main(String[] args) {

		hTable = new HashTable();

		File wordFile = new File("../instr/p7.dat");
		readFile(wordFile);

		hTable.printHashTable();
	}//end main

	public static void readFile(File wordFile) {
		//---------------------------------------------------------------------
		//Handles the processing of the file parameter passed. Adds each item
		//in the file to the hash table hTable.
		//Preconditions: Class variable hTable initialized as a HashTable.
		//file containing unique strings passed as parameter.
		//Postconditions: Each string in the file is added to the Hash Table.
		//---------------------------------------------------------------------

		Scanner fileReader;
		try {
			fileReader = new Scanner(wordFile);
			while (fileReader.hasNext()) {
				String word = fileReader.next();
				hTable.addItem(word);
			}
		} catch (FileNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}//readFile

}//end class

class HashTable {
	/*
	---------------------------------------------------------------------
	CLASS NAME: HashTable
	VARIABLE BANK:
		ALPHABET_VALUES - int[], contains the numeric values for
                                                        each letter in the alphabet.
		ALPHABET - Character[], contains each letter in the alphabet.
                                        Used for finding the numeric value of a letter.
		HASH_TABLE_SIZE - int, contains the maximum size of the Hash Table.
                                                        Used to initialize the array for storage.
		hashTable - String[], stores the data present in the Hash Table.

	DESCRIPTION:
		This class handles the addition, retrieval, and removal and storage
		of items in a Hash Table. This version of hash table uses a perfect
```

```
74              hash function to prevent collisions.
75          ---------------------------------------------------------------------
76          */
77
78          final static int[] ALPHABET_VALUES = {11, 15, 1, 0, 0, 15, 3, 15, 13, 0, 0, 15, 15, 13, 0, 15, 0, 14, 6, 6, 14, 10, 6, 0, 13, 0};
79          final static Character[] ALPHABET = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
80              'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};


81          final static int HASH_TABLE_SIZE = 36;
82          private String[] hashTable;

83          public HashTable() {
84              //Initialize the hashTable array.
85              this.hashTable = new String[HASH_TABLE_SIZE];
86          }

87          public boolean addItem(String item) {
88              //---------------------------------------------------------------
89              //Adds the item passed to the hash table at an index calculated using
90              //getHashIndex with the key passed.
91              //Preconditions: String item passed as parameter.
92              //Postconditions: The item passed is added to the hash table. If
93              //the addition is successful, true is returned. If not, false is
94              //returned.
95              //---------------------------------------------------------------
96
97              boolean success = true;

98              int hashIndex = getHashIndex(item);
99              if(hashIndex >= HASH_TABLE_SIZE || hashIndex < 0) {
100                 success = false;
101             } else {
102                 hashTable[hashIndex] = item;
103             }

104             return success;
105         }

106         public boolean removeItem(String item) {
107             //---------------------------------------------------------------
108             //Iterates through the hash table and finds the item with a key
109             //matching the String passed. The matching item is deleted.
110             //Preconditions: String item passed as parameter.
111             //Postconditions: The item present in the table is deleted if found
112             //and true is returned. If not found, false is returned.
113             //---------------------------------------------------------------
114             boolean success = false;

115             int hashIndex = getHashIndex(item);
116             if(hashIndex <= HASH_TABLE_SIZE || hashIndex >= 0) {
117                 hashTable[hashIndex] = null;
118             }//end else

119             return success;
120         }

121         public String getItem(String item) {
```

```
122                 //-----------------------------------------------------------------
123                 //Iterates through the hash table and finds the item with a key
124                 //matching the String passed. The matching item is returned.
125                 //Preconditions: String item passed as parameter.
126                 //Postconditions: The item present in the table is returned if found.
127                 //If not found, the method returns null.
128                 //-----------------------------------------------------------------
129
130                 String tableItem = null;

131                 int hashIndex = getHashIndex(item);
132                 if(hashIndex <= HASH_TABLE_SIZE || hashIndex >= 0) {
133                         for(String s: hashTable) {
134                                 if(s.matches(item))
135                                         return s;
136                         }
137                 }//end else

138                 return tableItem;
139         }

140         public int getHashIndex(String word) {
141                 //-----------------------------------------------------------------
142                 //Returns the hash index for the word passed based on the hashing
143                 //method.
144                 //Preconditions: String word passed as parameter. Method letterValue
145                 //that returns numeric values for the letters passed.
146                 //Postconditions: The hash index for the word passed is returned by
147                 //the method.
148                 //-----------------------------------------------------------------
149
150                 int hashIndex;
151                 char first = word.charAt(0);
152                 char last = word.charAt(word.length() - 1);

153                 //
154                 hashIndex = word.length() + letterValue(first) + letterValue(last) - 2;

155                 return hashIndex;
156         }//getHashIndex

157         public int letterValue(char c) {
158                 //-----------------------------------------------------------------
159                 //Returns the letter value for the character passed based on the
160                 //values stored in the ALPHABET_VALUES array.
161                 //Preconditions: ALHABET_VALUES array containing the values for each
162                 //letter in the alphabet.
163                 //Postconditions: The value for the letter passed is found and
164                 //and returned. The method returns -1 if the character couldn't be
165                 //found.
166                 //-----------------------------------------------------------------
167
168                 boolean found = false;
169                 int letterValue = -1;
170                 for(int i  = 0; i < ALPHABET.length && !found; i++) {
171                         if(ALPHABET[i] == c) {
172                                 letterValue = ALPHABET_VALUES[i];
173                                 found = true;
```

```
174                                     }
175                             }//end for

176                             return letterValue;
177                     }//end letterValue

178                     public void printHashTable() {
179                             //-----------------------------------------------------------------
180                             //Iterates through the hash table and prints the keys for the items
181                             //present.
182                             //Preconditions: Class variable hashTable initialized as an array
183                             //containing strings.
184                             //Postconditions: Each string in the hashTable array is printed.
185                             //-----------------------------------------------------------------
186
187                             int i = 0;
188                             for(String s: hashTable) {
189                                     System.out.print(i + " ");
190                                     System.out.println(s);
191                                     i++;
192                             }
193                     }
194     }
cat -b p7b.java          #display the source file with line numbers
  1     /*
  2     ---------------------------------------------------------------------
  3     PROGRAM NAME: Program 7b
  4     PROGRAMMER:   Samuel Jentsch
  5     CLASS:        CSC 241.001, Fall 2013
  6     INSTRUCTOR:   Dr. D. Dunn
  7     DATE STARTED: November 18, 2013
  8     DUE DATE:     November 20, 2013
  9     REFERENCES:   Computer Science
  10                    Data Abstraction and Problem Solving with Java
  11                    Janet J. Prichard & Frank M. Carrano
  12                  Dr. Dunn: assignment information sheet

  13    ROGRAM PURPOSE:
  14    a. This program reads in a series of unique words stored a file.
  15    b. The program stores these words in a hash table based on a hash
  16       key conversion.

  17    VARIABLE BANK:
  18              hTable - HashTableB, stores the values read from the file and
  19                                      handles their storage. HashTableB uses separate chaining.

  20    ADTs:
  21      Hash Table

  22    FILES USED:
  23      p7.dat - a file containing unique strings.

  24    ---------------------------------------------------------------------
  25     */
  26    import java.io.*;
  27    import java.util.*;

  28    public class p7b {
```

```
29          static HashTableB hTable;

30          public static void main(String[] args) {
31                  hTable = new HashTableB();

32                  File wordFile = new File("../instr/p7.dat");
33                  readFile(wordFile);

34                  hTable.printHashTable();
35          }//end main

36          public static void readFile(File wordFile) {
37                  //---------------------------------------------------------------------
38                  //Handles the processing of the file parameter passed. Adds each item
39                  //in the file to the hash table hTable.
40                  //Preconditions: Class variable hTable initialized as a HashTable.
41                  //file containing unique strings passed as parameter.
42                  //Postconditions: Each string in the file is added to the Hash Table.
43                  //---------------------------------------------------------------------
44
45                  Scanner fileReader;
46                  try {
47                          fileReader = new Scanner(wordFile);
48                          while (fileReader.hasNext()) {
49                                  String word = fileReader.next();
50                                  hTable.addItem(word);
51                          }
52                  } catch (FileNotFoundException e) {
53                          // TODO Auto-generated catch block
54                          e.printStackTrace();
55                  }
56          }//readFile

57  }//end class

58  class Node<T> {
59          /*
60          ---------------------------------------------------------------------
61          CLASS NAME: Node
62          VARIABLE BANK:
63                  item - T, data item contained by the node.
64                  Node<T> - next, reference to the next node.
65          DESCRIPTION:
66                  This class is the Node used by the ADT Stack class. It allows for
67                  a single data item and a reference to the next node.
68          ---------------------------------------------------------------------
69           */

70          T item;
71          Node<T> next;

72          public Node() {
73                  //Create a default node.
74                  next = null;
75                  item = null;
76          }
```

```
77              public Node(T item, Node<T> next) {
78                      //Create a Node with the values passed.
79                      this.next = next;
80                      this.item = item;
81              }
82      }

83      class HashTableB {
84              /*
85              ----------------------------------------------------------------------
86              CLASS NAME: HashTableB
87              VARIABLE BANK:
88                      ALPHABET - Character[], contains each letter in the alphabet.
89                                              Used for finding the numeric value of a letter.
90                      HASH_TABLE_SIZE - int, contains the maximum size of the Hash Table.
91                                                              Used to initialize the array for storage.
92                      hashTable - Node[], stores the data present in the Hash Table.
93                                              Storage uses nodes to allow implementation of separate
94                                              chaining to resolve collisions.
95
96              DESCRIPTION:
97                      This class handles the addition, retrieval, and removal and storage
98                      of items in a Hash Table. This version of Hash Table uses separate
99                      chaining to handle collisions.
100             ----------------------------------------------------------------------
101              */

102             final static Character[] ALPHABET = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
103                     'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};


104             final static int HASH_TABLE_SIZE = 36;
105             private Node[] hashTable;

106             public HashTableB() {
107                     //Initialize the hashTable array.
108                     this.hashTable = new Node[HASH_TABLE_SIZE];
109             }

110             public boolean addItem(String item) {
111                     //-------------------------------------------------------------
112                     //Adds the item passed to the hash table at an index calculated using
113                     //getHashIndex with the key passed.
114                     //Preconditions: String item passed as parameter.
115                     //Postconditions: The item passed is added to the hash table. If
116                     //the addition is successful, true is returned. If not, false is
117                     //returned.
118                     //-------------------------------------------------------------
119
120                     boolean success = true;

121                     int hashIndex = getHashIndex(item);
122                     if(hashIndex >= HASH_TABLE_SIZE || hashIndex < 0) {
123                             success = false;
124                     } else if(hashTable[hashIndex] == null) {
125                             hashTable[hashIndex] = new Node<String>(item, null);
126                     } else {
127                             Node<String> list = hashTable[hashIndex];
```

```
128                         Node<String> newNode = new Node<String>(item, list);
129                         list = newNode;
130                         hashTable[hashIndex] = list;
131                 }

132                 return success;
133         }

134         public boolean removeItem(String item) {
135                 //-----------------------------------------------------------------
136                 //Iterates through the hash table and finds the item with a key
137                 //matching the String passed. The matching item is deleted.
138                 //Preconditions: String item passed as parameter.
139                 //Postconditions: The item present in the table is deleted if found
140                 //and true is returned. If not found, false is returned.
141                 //-----------------------------------------------------------------
142
143                 boolean success = false;

144                 int hashIndex = getHashIndex(item);
145                 if(hashIndex <= HASH_TABLE_SIZE || hashIndex >= 0 ||
146                                 hashTable[hashIndex] != null) {
147                         Node<String> head = hashTable[hashIndex];
148                         Node<String> cur = head;
149                         Node<String> prev = cur;

150                         boolean found = false;
151                         while(cur != null && !success) {
152                                 if(cur.item.matches(item)) {
153                                         if(cur == head)
154                                                 head = head.next;
155                                         else
156                                                 prev.next = cur.next;
157                                         success = true;
158                                 }//end if
159                                 prev = cur;
160                                 cur = cur.next;
161                         }//end while
162                         hashTable[hashIndex] = head;
163                 }//end else

164                 return success;
165         }

166         public String getItem(String item) {
167                 //-----------------------------------------------------------------
168                 //Iterates through the hash table and finds the item with a key
169                 //matching the String passed. The matching item is returned.
170                 //Preconditions: String item passed as parameter.
171                 //Postconditions: The item present in the table is returned if found.
172                 //If not found, the method returns null.
173                 //-----------------------------------------------------------------
174
175                 String tableItem = null;

176                 int hashIndex = getHashIndex(item);
177                 if(hashIndex <= HASH_TABLE_SIZE || hashIndex >= 0 ||
178                                 hashTable[hashIndex] != null) {
```

```
179                         Node<String> head = hashTable[hashIndex];
180                         Node<String> cur = head;

181                         boolean found = false;
182                         while(cur != null && !found) {
183                                 if(cur.item.matches(item)) {
184                                         tableItem = cur.item;
185                                         found = true;
186                                 }//end if
187                                 cur = cur.next;
188                         }//end while
189                 }//end else

190                 return tableItem;
191         }

192         public int getHashIndex(String word) {
193                 //-------------------------------------------------------------------
194                 //Returns the hash index for the word passed based on the hashing
195                 //method.
196                 //Preconditions: String word passed as parameter. Method letterValue
197                 //that returns numeric values for the letters passed.
198                 //Postconditions: The hash index for the word passed is returned by
199                 //the method.
200                 //-------------------------------------------------------------------
201
202                 int hashIndex;
203                 char first = word.charAt(0);
204                 char last = word.charAt(word.length() - 1);

205                 //Value of key's first character + value of last character.
206                 //If a prime number were selected for the table size (37 maybe),
207                 //and the hashIndex were calculated using mod, the hashIndex could
208                 //be ensured to not exceed the bounds of the array and data should
209                 //be more evenly scattered than with a non-prime table size.
210                 hashIndex = letterValue(first) + letterValue(last) - 2;

211                 return hashIndex;
212         }//getHashIndex

213         public int letterValue(char c) {
214                 //-------------------------------------------------------------------
215                 //Returns the letter value for the character passed based on the
216                 //index of the value stored in the ALPHABET array.
217                 //Preconditions: AlPHABET array containing the values for each
218                 //letter in the alphabet.
219                 //Postconditions: The index for the letter passed is found and
220                 //and returned. The method returns -1 if the character couldn't be
221                 //found.
222                 //-------------------------------------------------------------------
223
224                 boolean found = false;
225                 int letterValue = -1;
226                 for(int i  = 0; i < ALPHABET.length && !found; i++) {
227                         if(ALPHABET[i] == c) {
228                                 letterValue = i + 1;
229                                 found = true;
230                         }
```

```
  231                         }//end for

  232                         return letterValue;
  233                 }//end letterValue

  234                 public void printHashTable() {
  235                         //-------------------------------------------------------------------
  236                         //Iterates through the hash table and prints the keys for the items
  237                         //present.
  238                         //Preconditions: Class variable hashTable initialized as an array
  239                         //containing strings.
  240                         //Postconditions: Each string in the hashTable array is printed.
  241                         //-------------------------------------------------------------------
  242
  243                         for(int i = 0; i < hashTable.length; i++) {
  244                                 Node<String> list = hashTable[i];
  245                                 Node<String> cur = list;

  246                                 System.out.print(i+ ":");

  247                                 while(cur != null) {
  248                                         System.out.print(" " + cur.item);
  249                                         cur = cur.next;
  250                                 }
  251                                 System.out.println();
  252                         }
  253                 }
  254     }
:                       #null command
:
:
echo Running p7
Running p7
javac p7.java           #compile the java file
java p7             #execute the file from the current directory
0 DO
1 END
2 ELSE
3 CASE
4 DOWNTO
5 GOTO
6 TO
7 OTHERWISE
8 TYPE
9 WHILE
10 CONST
11 DIV
12 AND
13 SET
14 OR
15 OF
16 MOD
17 FILE
18 RECORD
19 PACKED
20 NOT
21 THEN
22 PROCEDURE
```

23 WITH
24 REPEAT
25 VAR
26 IN
27 ARRAY
28 IF
29 NIL
30 FOR
31 BEGIN
32 UNTIL
33 LABEL
34 FUNCTION
35 PROGRAM
echo Running p7b
Running p7b
javac p7b.java #compile the java file
Note: p7b.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
java p7b #execute the file from the current directory
0:
1:
2:
3: AND
4:
5:
6: CASE
7: END
8: ELSE
9: FILE
10:
11:
12:
13: IF
14: BEGIN
15: MOD
16:
17: DOWNTO DO
18: PACKED OTHERWISE FUNCTION
19: PROCEDURE OF
20: RECORD GOTO
21: IN CONST
22: LABEL FOR
23: TYPE
24: NIL DIV ARRAY
25:
26: WHILE
27: PROGRAM
28:
29: WITH
30:
31: UNTIL OR
32: THEN NOT
33: TO
34:
35:
:
:
:

```
date                    #print the date
Wed Nov 20 09:08:09 CST 2013
```