Andreas Landgrebe
Computer Science 250: Analysis of Algoithms
Lab 8: de la Briandais Tries and (Sub-) String Search

Part 1: Creating the dela Briandais Trie
DLBnode.java

```
public class DLBnode {
   public char key;
   public int value;
   public DLBnode sibling;
   public DLBnode child;

   public DLBnode(char keyValue, int nodeValue, DLBnode sibling, DLBnode
       child) {
      this.key = keyValue;
      this.value = nodeValue;
      this.sibling = sibling;
      this.child = child;
   }

   public DLBnode() {

   }

}
```

DLB.java

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

public class DLB {

   public DLBnode firstNode;
   public char empty = ’ ’;
    public TreeSet<String> solutionSet= new TreeSet<>(); // A TreeSet is
        sorted, elements are unique

   public DLB(){
      this.firstNode = new DLBnode(empty,0,null,null); //establishing
          DLB node, value = 0, meaning it is not a word
   }

   public void add(String input) {


      if(search(input) == 1) {
         return; //if input already exists, DO NOT ADD IT AGAIN
      }
   DLBadd(input,0,firstNode);
}

public void DLBadd(String input, int position, DLBnode current) {



   char c = input.charAt(position);

   DLBnode temp = current;


   while(temp != null){
      current = temp;
      if(current.key == c) { // found the character in current tree
         if(position+1 == input.length()){
            current.value = 1; // if end of string then just mark it
                as a word
            return;
         } else {
            if (current.child==null) { // no children, allocate new
                subtree with remaining characters.
               for (int i=position+1; i<input.length(); i++) {
                  DLBnode d = new DLBnode();
                  current.child=d;
                  d.key = input.charAt(i);
                  d.value = 0;
                  current=d;
               }
               current.value=1; // mark word finished
               return;
            } else {
               DLBadd(input, position+1,current.child);
               return;
            }
         }
      }
      temp = current.sibling;
   }

   if(position+1 == input.length()){ //checking for the the end of
       the word
      current.sibling = new DLBnode(c,1,null,null); //value is 1 if
          it is the end of the word

   } else { //else the value is 0 and a new DLB node will be
       established for the sibling
      DLBnode ch = new DLBnode();
      current.sibling = new DLBnode(c,0,null,ch);
      DLBadd(input, position+1, current.sibling.child); //recursive
          call seeing that the word is not over
   }

}



public int search(String input) {
   return searchFromPosition(input,0,firstNode);
}

public int searchFromPosition(String input, int position, DLBnode
    current) { //search to see if a DLB trie will be established

   if(current == null) { //if there is nothing in the node, return 0
      return 0;
   } //if

   char c = input.charAt(position); //used for checking levels to see
       if it is the same character

   do { //do-while if there is something in the node

      if(c == current.key) { //if a match is found
         if(position+1 == input.length()){
            if(current.value == 1) {
               return 1; //if value of node is 1, meaning ending, then
                   return 1
            } else {
               return 0;
            }
         } else {
            return searchFromPosition(input,position+1,current.child);
                //recursive call to check whether a DLB will be created
         }
      } else {
         current = current.sibling;
      }
   }
   while(current != null); //end of do-while loop

   return 0;

}//searchfromPosition method


public void writeNode(DLBnode secondNode, int position) { //print out
    node for debugging proposes
   while(secondNode.sibling != null){ //while a sibling exists
       instead of the DLB



      for(int i = 0; i <= position; i++) { //for every position of
          the sibling, print out a -
         System.out.print("-");
      }
      System.out.println(secondNode.key);
      if(secondNode.child != null) {
         writeNode(secondNode.child, position+1);
      }

      secondNode = secondNode.sibling;
   }
}



public int beginsWith(String input){

   //return values
   // 0 means no words starts with this
   // 1 means input is in dictionary
   // 2 means a word in the dictionary starts with the string input

   return beginsWithFromPosition(input,0,firstNode);
}


public int beginsWithFromPosition(String input, int position, DLBnode
    current) { //search to see if a DLB trie will be established

   if(current == null) { //if there is nothing in the node, return 0
      return 0;
   } //if

   char c = input.charAt(position); //used for checking levels to see
       if it is the same character

   do { //do-while if there is something in the node

      if(c == current.key) { //if a match is found
         if(position+1 == input.length()){
            if(current.value == 1) {
               return 1; //if value of node is 1, meaning ending, then
                   return 1
            } else {
               return 2; // end of input string but not end of DLB
                    branch
            }
         } else {
            return
                 beginsWithFromPosition(input,position+1,current.child);
                 //recursive call to check whether a DLB will be created
         }
      } else {
         current = current.sibling;
      }
   }
   while(current != null); //end of do-while loop

   return 0;

}//beginsWithFromPosition method


public void tryNeighbors(String candidate, int row, int col, char[][]
     boggleBoard, int n){
   char c;
   int res;
   if (search(candidate)==1){
      solutionSet.add(candidate);
   }
   if (col>0) { // Explore to the left of row,col
      c = boggleBoard[row][col-1];
      res = beginsWith(candidate+c);
      if (res!=0) {
         tryNeighbors(candidate+c, row, col-1, boggleBoard, n);
      }
   }

   if (col < n - 1) {// Explore to the right of row,col
      c = boggleBoard[row][col + 1];
      res = beginsWith(candidate+c);
      if(res!=0){
         tryNeighbors(candidate+c, row, col+1, boggleBoard, n);
      }
   }

   if (row > 0){// Explore above of row,col
      c = boggleBoard[row-1][col];
      res = beginsWith(candidate+c);
      if(res!=0) {
         tryNeighbors(candidate+c, row-1, col, boggleBoard, n);
      }

   }
```

```
      if(row < n - 1){// Explore below of row,col
         c = boggleBoard[row+1][col];
         res = beginsWith(candidate+c);
         if(res!=0){
            tryNeighbors(candidate+c, row+1, col, boggleBoard, n);
         }
      }
   }
}
```

DictionaryTest.java

```
import java.io.*;
import java.util.*;
public class DictionaryTest {

   public static void main(String [] args) throws IOException {

      Scanner fileScan = new Scanner(new
          FileInputStream("/Users/andreas/Documents/cs250/cs250s2015-grebes15/lab8/Part1/dictionary.tx
      String st;
      DLB D = new DLB();
      //DLBnode thirdNode = new DLBnode();

      while (fileScan.hasNext()) {
         st = fileScan.nextLine();
         D.add(st);

      } //while

      String[] tests = {"abc", "abe", "abet", "abx", "ace", "acid",
          "hives",
              "iodin", "iodine", "idodinet", "inval", "zoo", "zool",
              "zoology", "zoologys", "zurich"};

      for (int i = 0; i < tests.length; i++) {
         int ans = D.search(tests[i]);
         //D.writeNode(thirdNode,ans);
         System.out.print(tests[i] + " -- ");
         System.out.println(ans);
         switch (ans) {
            case 0:
               System.out.println("FALSE");
               break;
            case 1:
               System.out.println("TRUE");
               break;
         } //switch

      } //for
```

```
  } //main

} //DictionaryTest (class)
```

Output from Part 1 to the DictionaryTest.java file to show that my DLB works

```
abc -- 0
FALSE
abe -- 0
FALSE
abet -- 1
TRUE
abx -- 0
FALSE
ace -- 1
TRUE
acid -- 1
TRUE
hives -- 1
TRUE
iodin -- 0
FALSE
iodine -- 1
TRUE
idodinet -- 0
FALSE
inval -- 0
FALSE
zoo -- 1
TRUE
zool -- 0
FALSE
zoology -- 1
TRUE
zoologys -- 0
FALSE
zurich -- 0
FALSE
```

Part Two: Source Code:

DLBnode.java

```
public class DLBnode {
  public char key;
```

```
   public int value;
   public DLBnode sibling;
   public DLBnode child;

   public DLBnode(char keyValue, int nodeValue, DLBnode sibling, DLBnode child) {
      this.key = keyValue;
      this.value = nodeValue;
      this.sibling = sibling;
      this.child = child;
   }

   public DLBnode() {

   }

}
```

DLB.java

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

public class DLB {

   public DLBnode firstNode;
   public char empty = ' ';
    public TreeSet<String> solutionSet= new TreeSet<>(); // A TreeSet is sorted, elements are unique

   public DLB(){
      this.firstNode = new DLBnode(empty,0,null,null); //establishing DLB node, value = 0, meaning it is not a word
   }

   public void add(String input) {


      if(search(input) == 1) {
         return; //if input already exists, DO NOT ADD IT AGAIN
      }

      DLBadd(input,0,firstNode);
   }

   public void DLBadd(String input, int position, DLBnode current) {



      char c = input.charAt(position);
   DLBnode temp = current;


   while(temp != null){
      current = temp;
      if(current.key == c) { // found the character in current tree
         if(position+1 == input.length()){
            current.value = 1; // if end of string then just mark it
               as a word
            return;
         } else {
            if (current.child==null) { // no children, allocate new
               subtree with remaining characters.
               for (int i=position+1; i<input.length(); i++) {
                  DLBnode d = new DLBnode();
                  current.child=d;
                  d.key = input.charAt(i);
                  d.value = 0;
                  current=d;
               }
               current.value=1; // mark word finished
               return;
            } else {
               DLBadd(input, position+1,current.child);
               return;
            }
         }
      }
      temp = current.sibling;
   }

   if(position+1 == input.length()){ //checking for the the end of
       the word
      current.sibling = new DLBnode(c,1,null,null); //value is 1 if
          it is the end of the word

   } else { //else the value is 0 and a new DLB node will be
       established for the sibling
      DLBnode ch = new DLBnode();
      current.sibling = new DLBnode(c,0,null,ch);

      DLBadd(input, position+1, current.sibling.child); //recursive
          call seeing that the word is not over
   }

}
```

```
public int search(String input) {
   return searchFromPosition(input,0,firstNode);
}

public int searchFromPosition(String input, int position, DLBnode
    current) { //search to see if a DLB trie will be established

   if(current == null) { //if there is nothing in the node, return 0
      return 0;
   } //if

   char c = input.charAt(position); //used for checking levels to see
       if it is the same character

   do { //do-while if there is something in the node

      if(c == current.key) { //if a match is found
         if(position+1 == input.length()){
            if(current.value == 1) {
               return 1; //if value of node is 1, meaning ending, then
                   return 1
            } else {
               return 0;
            }
         } else {
            return searchFromPosition(input,position+1,current.child);
                //recursive call to check whether a DLB will be created
         }
      } else {
         current = current.sibling;
      }
   }
   while(current != null); //end of do-while loop

   return 0;

}//searchfromPosition method


public void writeNode(DLBnode secondNode, int position) { //print out
    node for debugging proposes

   while(secondNode.sibling != null){ //while a sibling exists
       instead of the DLB



      for(int i = 0; i <= position; i++) { //for every position of
          the sibling, print out a -
         System.out.print("-");
      }
      System.out.println(secondNode.key);
      if(secondNode.child != null) {
         writeNode(secondNode.child, position+1);
      }

      secondNode = secondNode.sibling;
   }
}



public int beginsWith(String input){

   //return values
   // 0 means no words starts with this
   // 1 means input is in dictionary
   // 2 means a word in the dictionary starts with the string input

   return beginsWithFromPosition(input,0,firstNode);
}


public int beginsWithFromPosition(String input, int position, DLBnode
    current) { //search to see if a DLB trie will be established

   if(current == null) { //if there is nothing in the node, return 0
      return 0;
   } //if

   char c = input.charAt(position); //used for checking levels to see
       if it is the same character

   do { //do-while if there is something in the node

      if(c == current.key) { //if a match is found
         if(position+1 == input.length()){
            if(current.value == 1) {
               return 1; //if value of node is 1, meaning ending, then
                   return 1
            } else {
               return 2; // end of input string but not end of DLB
                   branch
            }
         } else {
            return
                beginsWithFromPosition(input,position+1,current.child);
                //recursive call to check whether a DLB will be created
         }
      } else {
         current = current.sibling;
      }
   }
   while(current != null); //end of do-while loop

   return 0;

}//beginsWithFromPosition method


public void tryNeighbors(String candidate, int row, int col, char[][]
    boggleBoard, int n){
   char c;
   int res;
   if (search(candidate)==1){
      solutionSet.add(candidate);
   }
   if (col>0) { // Explore to the left of row,col
      c = boggleBoard[row][col-1];
      res = beginsWith(candidate+c);
      if (res!=0) {
         tryNeighbors(candidate+c, row, col-1, boggleBoard, n);
      }
   }

   if (col < n - 1) {// Explore to the right of row,col
      c = boggleBoard[row][col + 1];
      res = beginsWith(candidate+c);
      if(res!=0){
         tryNeighbors(candidate+c, row, col+1, boggleBoard, n);
      }
   }

   if (row > 0){// Explore above of row,col
      c = boggleBoard[row-1][col];
      res = beginsWith(candidate+c);
      if(res!=0) {
         tryNeighbors(candidate+c, row-1, col, boggleBoard, n);
      }

   }
   if(row < n - 1){// Explore below of row,col
      c = boggleBoard[row+1][col];
      res = beginsWith(candidate+c);
      if(res!=0){
         tryNeighbors(candidate+c, row+1, col, boggleBoard, n);
      }
   }
   }
}
```

boggle.java

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

public class DLB {

   public DLBnode firstNode;
   public char empty = ' ';
    public TreeSet<String> solutionSet= new TreeSet<>(); // A TreeSet is
        sorted, elements are unique

   public DLB(){
      this.firstNode = new DLBnode(empty,0,null,null); //establishing
          DLB node, value = 0, meaning it is not a word
   }

   public void add(String input) {


      if(search(input) == 1) {
         return; //if input already exists, DO NOT ADD IT AGAIN
      }

      DLBadd(input,0,firstNode);
   }

   public void DLBadd(String input, int position, DLBnode current) {



      char c = input.charAt(position);

      DLBnode temp = current;


      while(temp != null){
         current = temp;
         if(current.key == c) { // found the character in current tree
            if(position+1 == input.length()){
               current.value = 1; // if end of string then just mark it
                   as a word
               return;
            } else {
               if (current.child==null) { // no children, allocate new
                   subtree with remaining characters.
                  for (int i=position+1; i<input.length(); i++) {
```

```
                  DLBnode d = new DLBnode();
                  current.child=d;
                  d.key = input.charAt(i);
                  d.value = 0;
                  current=d;
               }
               current.value=1; // mark word finished
               return;
            } else {
               DLBadd(input, position+1,current.child);
               return;
            }
         }
      }
      temp = current.sibling;
   }

   if(position+1 == input.length()){ //checking for the the end of
        the word
      current.sibling = new DLBnode(c,1,null,null); //value is 1 if
           it is the end of the word

   } else { //else the value is 0 and a new DLB node will be
        established for the sibling
      DLBnode ch = new DLBnode();
      current.sibling = new DLBnode(c,0,null,ch);

      DLBadd(input, position+1, current.sibling.child); //recursive
           call seeing that the word is not over
   }

}




public int search(String input) {
   return searchFromPosition(input,0,firstNode);
}

public int searchFromPosition(String input, int position, DLBnode
    current) { //search to see if a DLB trie will be established

   if(current == null) { //if there is nothing in the node, return 0
      return 0;
   } //if

   char c = input.charAt(position); //used for checking levels to see
        if it is the same character
   do { //do-while if there is something in the node

      if(c == current.key) { //if a match is found
         if(position+1 == input.length()){
            if(current.value == 1) {
               return 1; //if value of node is 1, meaning ending, then
                   return 1
            } else {
               return 0;
            }
         } else {
            return searchFromPosition(input,position+1,current.child);
                //recursive call to check whether a DLB will be created
         }
      } else {
         current = current.sibling;
      }
   }
   while(current != null); //end of do-while loop

   return 0;

}//searchfromPosition method


public void writeNode(DLBnode secondNode, int position) { //print out
    node for debugging proposes

   while(secondNode.sibling != null){ //while a sibling exists
       instead of the DLB



      for(int i = 0; i <= position; i++) { //for every position of
          the sibling, print out a -
         System.out.print("-");
      }
      System.out.println(secondNode.key);
      if(secondNode.child != null) {
         writeNode(secondNode.child, position+1);
      }

      secondNode = secondNode.sibling;
   }
}



public int beginsWith(String input){
   //return values
   // 0 means no words starts with this
   // 1 means input is in dictionary
   // 2 means a word in the dictionary starts with the string input

   return beginsWithFromPosition(input,0,firstNode);
}


public int beginsWithFromPosition(String input, int position, DLBnode
    current) { //search to see if a DLB trie will be established

   if(current == null) { //if there is nothing in the node, return 0
      return 0;
   } //if

   char c = input.charAt(position); //used for checking levels to see
       if it is the same character

   do { //do-while if there is something in the node

      if(c == current.key) { //if a match is found
         if(position+1 == input.length()){
            if(current.value == 1) {
               return 1; //if value of node is 1, meaning ending, then
                   return 1
            } else {
               return 2; // end of input string but not end of DLB
                   branch
            }
         } else {
            return
                beginsWithFromPosition(input,position+1,current.child);
                //recursive call to check whether a DLB will be created
         }
      } else {
         current = current.sibling;
      }
   }
   while(current != null); //end of do-while loop

   return 0;

}//beginsWithFromPosition method


public void tryNeighbors(String candidate, int row, int col, char[][]
    boggleBoard, int n){
   char c;
   int res;
```

```
      if (search(candidate)==1){
         solutionSet.add(candidate);
      }
      if (col>0) { // Explore to the left of row,col
         c = boggleBoard[row][col-1];
         res = beginsWith(candidate+c);
         if (res!=0) {
            tryNeighbors(candidate+c, row, col-1, boggleBoard, n);
         }
      }

      if (col < n - 1) {// Explore to the right of row,col
         c = boggleBoard[row][col + 1];
         res = beginsWith(candidate+c);
         if(res!=0){
            tryNeighbors(candidate+c, row, col+1, boggleBoard, n);
         }
      }

      if (row > 0){// Explore above of row,col
         c = boggleBoard[row-1][col];
         res = beginsWith(candidate+c);
         if(res!=0) {
            tryNeighbors(candidate+c, row-1, col, boggleBoard, n);
         }

      }
      if(row < n - 1){// Explore below of row,col
         c = boggleBoard[row+1][col];
         res = beginsWith(candidate+c);
         if(res!=0){
            tryNeighbors(candidate+c, row+1, col, boggleBoard, n);
         }
      }
   }
}
```

Sample Output From data2.txt

```
BoggleBoard:
f u n n
i g e y
t i v a
a l o b

There were 41 total words:
a
above
alive
at
```

```
aye
bay
bob
eve
even
evil
eye
fig
fit
fug
fugitive
fun
funny
gene
gig
give
given
gun
if
it
lit
live
liven
lob
love
nu
nun
olive
ova
oven
tit
uneven
vigil
vital
viva
ye
yen
```

Part Three: While You Have Some Downtime

1. We can calculate the compression ratio of a compression scheme by using the simple formula $bits\_with\_compression/bits\_without\_comrpession$. What is the compression ratio of the example shown on Slide #13 of Lecture #18? Try it with both 7 and 8 bits per character for the uncompressed text.

   The compression ratio of the example shown $LF : 6 * 1 = 6$
   $SP : 2 * 11 = 22$

$a : 5 * 2 = 10$
$b : 6 * 1 = 6$
$e : 3 * 5 = 15$
$f : 5 * 2 = 10$
$h : 5 * 2 = 10$
$i : 4 * 4 = 16$
$m : 5 * 2 = 10$
$o : 4 * 3 = 12$
$r : 5 * 1 = 5$
$s : 3 * 6 = 18$
$t : 3 * 8 = 24$
$w : 4 * 3 = 12$
$= 176$ bits
7 bits per character: 7∗51 (this is the total amount of characters) $= 357bits$
$\frac{176}{357} = 0.493$
8 bits per character: $8 * 51 = 408$
$\frac{176}{408} = 0.431$

2. Prove that the two longest codewords in a Huffman code have the same length.

   The two longest codewords in a Huffman code have the same length. This is the case since when it comes to fixed length code, each codeword uses the same number of bits. In order to evaluation this code segment, per codeword need to have a expected number of bits.

3. Create a Huffman code for the string "AAAAAAAAAABBBBCCDDDDDE-EFFG! What is its compression ratio, with both 7 and 8 bits per character uncompressed?
   $G : 5 * 1 = 5$
   $! : 5 * 1 = 5$
   $C : 4 * 2 = 8$
   $E : 4 * 2 = 8$
   $F : 4 * 2 = 8$
   $B : 3 * 4 = 12$
   $D : 3 * 5 = 15$
   $A : 1 * 10 = 10$
   7 bits: $7 * 27 = 189$
   $\frac{72}{189} = 0.381$
   8 bits: $8 * 28 = 216$
   $\frac{72}{216} = 0.333$