CS214 Systems Programming
Assignment 0 String Sorting
by Chris Gong and Eric Giovannini

RUIDs
Chris Gong: 164008349
Eric Giovannini: 162008912

I. What the program does

PointerSorter is a program that takes an argument through the command line, parses it into different sequences of alphabetic characters, and sorts them in ascending alphabetical order (with uppercase having precedence over lowercase). The strings within the command line argument will be separated by non-alphabetic characters like so,

“this program is amazing”

“this:program,is9amazing”

Both input arguments contain the words, “this”, “program”, “is”, and “amazing”. As a result, the output corresponding with each string input should be the same,

```
[cg646@null cg646]$ ./pointersorter "this:program,is9amazing"
amazing
is
program
this
[cg646@null cg646]$ ./pointersorter "this program is amazing"
amazing
is
program
this
```

II. How the input is being parsed

A couple problems were met when gathering words from the string input.

- The number of contiguous sequences of alphabetic letters is unknown
- The length of each string component is also unknown

To deal with the unknown number of words, there is a loop that looks at each individual character within the string.

```
char *curr = argv[1];
char letter = *curr;
while (letter != '\0') {
```

```
    curr++;
    currentIndex++;
    letter = *curr;
}
```

argv[1] represents the inputted string, so looking at the past examples above *argv[1]* would be either “this program is amazing” or “this:program,is9amazing”. The variable, *letter*, signifies the current letter being traversed through in the loop. *letter* is incremented by first incrementing *curr*, which points to the address of the current character being looked at in the loop, then setting *letter* to the dereferenced value of *curr*. And to take care of the unknown component length, we have to keep another variable, called *count*, to measure the length of the current contiguous sequence of alphabetic characters being traversed through.

```
long long count = 0;
long long beginningIndex = 0;
long long currentIndex = 0; /
if (isalpha(letter)) {
  count++;
}
```

In the case that an alphabetic letter is encountered, the *count* variable is incremented to denote that the length of the current word has increased by one newly found letter.

```
if (count != 0) { //signifies the end of a consecutive sequence of alphabetic characters
  word = (char *) malloc(sizeof(char) * (count + 1)); //+1 for null terminating character
  memcpy(word, argv[1] + beginningIndex, count);
  word[count] = '\0';
  count = 0;
```

Once a non-alphabetic character is found, the program goes to this else case. The first check is to see whether the *count* variable is zero or not. If *count* is zero, then that means we did not find any alphabetic characters yet or we encountered multiple non alphabetic characters in a row. In both cases, it is not needed to store any data. However, if *count* is not zero, then that means not only did we encounter a non-alphabetic character after a succession of alphabetic characters but also we need to store the word somewhere by using *memcpy*. When allocating dynamic memory to store the word, one extra character must be allocated for the insertion of a null-terminating character. Note that *beginningIndex* keeps track of the index of the first character in a contiguous sequence of alphabetic letters. Therefore, whenever a non-alphabetic character is found, regardless of what *count* is, *beginningIndex* is updated no matter what using *currentIndex*, the index of the current letter being looked at in the loop.

```
beginningIndex = currentIndex + 1;
```

Once the loop has ended, we revisit the *count* variable again just in case we encounter an input such as “cow”. Notice that there are no non-alphabetic characters in this input, so we need to check whether *count* is zero or not once the loop ends. If it is not zero, then that means the last word in the string has not been stored yet and we need to repeat the code segment above

involving *memcpy*. Otherwise, the input ended on a non-alphabetic character and we don't need to worry about that.

III. How the words are being sorted and outputted

In order to store the words, a red-black tree is being implemented for its unique advantages over other data structures

- It sorts the words for us, so there is no need to insert the words into a linked list only to be sorted later with another method such as mergesort, quicksort, bubblesort, etc..
- Unlike a regular tree, the red-black tree is self-balancing so even its worst-case time complexity for insertion will be log(n), with n being the number of nodes/words in the tree. A non-self-balancing tree may become skewed causing the worst-case time complexity for insertion to be n.

Besides being self-balancing, the red-black tree exhibits these properties,

- Each node in the tree contains a word and a color (either red or black)
- The root of the tree is always black
- The leaf nodes are considered black
- Every path from the root to a leaf node contains the same number of black node

```
Tree t;
Node *node;
initNode(node, RED, word);
insert(&t, node);
```

Every time a word is added into the tree, the node is initialized with both the word and the color red.

```
if(tree->root == NULL){
  tree->root = node;
  tree->root->col = BLACK;
}
else {
  BSTinsert(tree, node);
```

Within the *insert* method, we first check if there are any nodes already in the tree. If not, then the root must be set and the color of the root must be black. If there are already nodes in the tree, then we insert the new node into the tree as if it were a binary search tree.

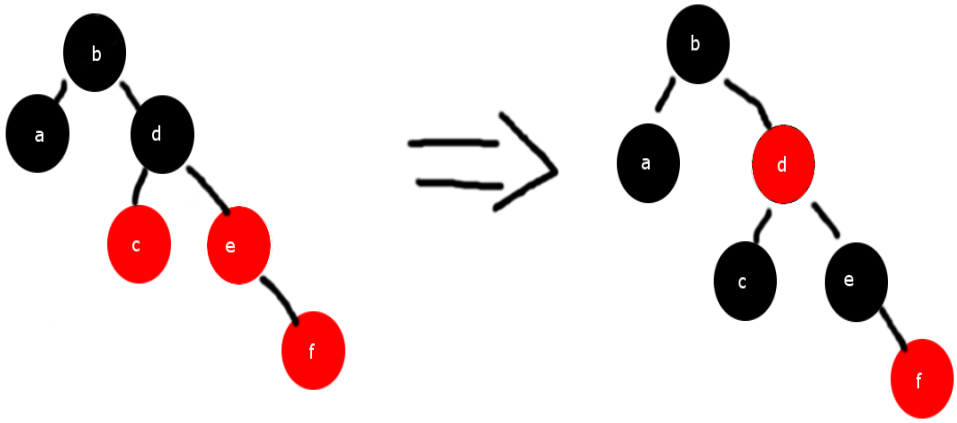
```
while((node != tree->root) && (node->parent->col == RED)) {
```

In the case the newly added node and its parent are both red, action must be taken to rebalance the tree. Depending on the color of the new node's uncle, one of two things can happen,

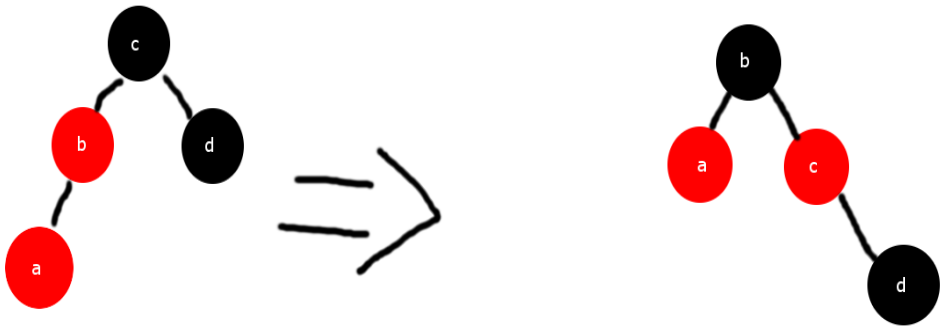
```
tree->root->col = BLACK;
```

1. If the uncle is red, then the node's parent, uncle, and grandparent must all be recolored. Then, the node's grandparent comes into selection and in the case where there is another

red-red relationship, we must examine whether its uncle is red or black to determine whether more action is needed. Here is an example of recoloring initiated by the red uncle case (*f* being the newly added node),



2. If the uncle is black, then either one or two rotations occur (depending on the orientation of the node, its parent, and its grandparent), and some recoloring occurs. Unlike the red uncle case, once this case occurs, no other node needs to be checked, the rebalancing process is done. With the red uncle case, the balancing process could go all the way back up to the root of the tree where it will eventually conclude. Here is an example of a rotation initiated by the black uncle case (*a* being the newly added node),



After the recoloring and/or rotations are finished, the root of the tree is colored black just in case it was colored red during the rebalancing process.

```
tree->root->col = BLACK;
```

When printing the output, we used a simple in-order traversal method. This traversal was also used to *free* each node and its word to ensure for no memory leaks. Both methods below are called on the root of the tree.

```c
void BSTinorder(Node *node) {
  if (node == NULL) return;

  // keep going left
  BSTinorder(node->left);

  // print node
  printf("%s\n", node->word);

  // go right
  BSTinorder(node->right);

}
```

```c
void freeNodes(Node *node) {
  if (node == NULL) return;
  freeNodes(node->left);

  freeNodes(node->right);

  // free the node and the string:
  free(node->word);
  free(node);
}
```