Card Sorting by Algorithms and Tree

Introduction

This report includes the code and the explanation for the code for a sorting program and a tree program. Sorting program sorts playing cards by utilising different methods like the Bubble Sort method and the Quick Sort method. The tree program also sorts playing cards but using the binary tree method. Relevant output and comments are displayed.

Sorting Program

Data Structures Used: A struct named aCard is used to represent a playing card. It consists of an integer named cardVal and a variable named cardSuit, which is type Suit. Integer cardVal is to represent the value of the card from 1 to 13. Suit type cardSuit is to represent the suit of the card. Suit is an enumeration that consists of hearts, clubs, diamonds, spades respectively that corresponds to values from 0 to 3.

```
enum Suit { hearts, clubs, diamonds, spades };      // define suits

struct aCard {                          // defines a card
    int cardVal;                        // number 1..13
    Suit cardSuit;                      // suit
};
```

cardToStr: Takes one aCard. Converts the given card's value and suit to characters, makes a string from them and returns the string.

```
string cardToStr(aCard c) {
    string res;                         //String to store the result.
    string cval = "A23456789TJQK";      //String that represents the value of a card from ace to king.
    string csuit = "HCDS";              //String that represents the suit of a card from hearts to
spades.
    res = cval.substr(c.cardVal-1, 1);  //Takes the character from cval according to the cardVal value of
the card and writes it to result string.
    res += csuit.substr(c.cardSuit, 1); //Takes the character from csuit according to the cardSuit value
of the card and appends it to result string.
    return res;
}
```

printPack: Takes a string for message and returns nothing. Prints the message and then the cards as a string.

```
void printPack(string mess) {
    string res;              //String to store the result
    res = mess + ":\t";      //Writes the message from the argument and a colon to the result string
    for (int ct = 0; ct < maxCard; ct++)           //For each card in the pack
        res += cardToStr(thePack[ct]) + ",";       //Call cardToStr for that card and add it to the
result string with a comma
    cout << res.substr(0, res.length() - 1) + "\n"; //Removes the last character from res (which is a
comma) and adds a new line character and prints
}
```

compareCards: Takes two aCards and retuns an integer. Compares two cards by their suits then their values.

```
int compareCards(aCard c1, aCard c2) {
    compareCount++;                                 //Increase the compare counter
    if (c1.cardSuit < c2.cardSuit) return -1;   //Returns -1 if c2 has a greater suit
    if (c1.cardSuit > c2.cardSuit) return 1;    //Returns 1 if c1 has a greater suit
    if (c1.cardVal < c2.cardVal) return -1;     //If the suits are the same, returns -1 if c2 has a
greater value
    if (c1.cardVal > c2.cardVal) return 1;      //If the suits are the same, returns 1 if c1 has a
greater value
    return 0;                                   //Returns 0 if both the suits and the values are the same
}
```

swapCards: Takes 2 integers as indexes for the cards in the pack and returns nothing. Swaps the cards on given indexes.

```
void swapCards(int n1, int n2) {
    aCard temp = thePack[n1];   //Saves the card on the first given index to a temporary aCard variable.
    thePack[n1] = thePack[n2];  //Puts the card on the second given index to the first given index.
    thePack[n2] = temp;         //Puts the card in temp to the second given index.
    moveCount += 3;             //Adds 3 moves to the move counter
}
```

bubbleSort: Takes no arguments and returns nothing. Sorts the cards in the pack using bubble sort method.

```
void bubbleSort() {
    for (int i = 1; i < maxCard; i++)              //For 1 to number of cards as i
        for (int j = maxCard - 1; j > i - 1; j--)   //For number of cards to i-1 as j
            if (compareCards(thePack[j - 1], thePack[j]) == 1) swapCards(j - 1, j); //If the card behind
the card at index j is greater than the card at index j, then swap those two cards
}
```

quickSort: Takes 2 integers for the low index and the high index, and another one to keep track of the depth of recursion (default value of 0 is given). Returns nothing. Sorts the cards in the pack using quick sort method.

```
void quickSort(int low, int high, int recursiveDepth = 0) {
    aCard pivot = thePack[(low + high) / 2];    //Sets the pivot to the card at the middle
    int i = low;                                //Sets i to the low value
    int j = high;                               //Sets j to the high value
    do {
        while (compareCards(thePack[i], pivot) == -1) i++;  //From the start of the current pack,
increase i while the card is less than pivot
        while (compareCards(thePack[j], pivot) == 1) j--;   //From the end of the current pack, decrease
j while the card is greater than pivot
        if (i <= j){
            if (i < j) swapCards(i, j); //If i is less than j, swap the cards at indexes i and j
            i++;                        //Increase i if it is less than or equals to j
            j--;                        //Decrease j if it is greater than i
        }
    } while (i < j);                    //Keep executing the loop while i is less than j
    if (low < j) quickSort(low, j, recursiveDepth+1);                   //If low index is less than j,
then call the function recursively by passing j as the high index and increasing the depth value
    if (i < high) quickSort(i, high, recursiveDepth+1);                 //If i is less than the high
index, then call the function recursively by passing i as the low index and increasing the depth value
    maxDepth = (recursiveDepth > maxDepth) ? recursiveDepth : maxDepth;  //When this call is finished, set
global variable to store the max depth to the current depth if the current depth is greater
}
```

main (and getCards): (Placed already given for loop to get the cards in function getCards.) Inside main, same pack of cards are sorted by calling bubbleSort and quickSort functions. Unsorted and sorted packs are printed along with the counter values.

```
void getCards() {
    for (int ct = 0; ct < maxCard; ct++)
        thePack[ct] = getCard("30021984");
}

int main()
{
    cout << "Card Sorting!\n";

    srand(100); //Set the random seed
    getCards(); //Get the cards from the student number
    printPack("Unsorted");  //Print the unsorted pack
    cout << "\n";

    compareCount = moveCount = 0;   //Set the counters to 0 before sorting
    bubbleSort();
    printPack("Bubble Sorted");     //Print the bubble sorted pack
```

```
    cout << "Comparisons made: " << compareCount << "\nTimes cards were moved: " << moveCount <<
"\n";  //Print the counter values
    cout << "\n";

    srand(100);
    getCards();
    compareCount = moveCount = maxDepth = 0;    //Set the counters and the recursion depth count to 0
    quickSort(0, maxCard - 1);                  //Call quick sort with 0 as the low index and number of
cards - 1 as the high index
    printPack("Quick Sorted");                  //Print the quick sorted pack
    cout << "Comparisons made: " << compareCount << "\nTimes cards were moved: " << moveCount << "\n
Maximum depth of recursion: " << maxDepth <<"\n";   //Print the counter values

}
```

Results:

```
Card Sorting!
Unsorted:       AS,9D,6D,3D,KD,9H,9H,AH,5S,4D,9H,3D,3H,4H,TH,6D,4H,TC,AH,TH

Bubble Sorted:  AH,AH,3H,4H,4H,9H,9H,9H,TH,TH,TC,3D,3D,4D,6D,6D,9D,KD,AS,5S
Comparisons made: 190
Times cards were moved: 375

Quick Sorted:   AH,AH,3H,4H,4H,9H,9H,9H,TH,TH,TC,3D,3D,4D,6D,6D,9D,KD,AS,5S
Comparisons made: 103
Times cards were moved: 69
Maximum depth of recursion: 5
```

Tree Program

Explanation of The Concepts Used: A binary tree consists of tree nodes, each storing a data (a playing card) and 2 pointers, one pointing to a node less than this node, other one to a node more than this node. When a new node (a card) is inserted to the tree, its position on the tree is determined by comparing its value to the other nodes, therefore it ends up in the correct relative position to other nodes.

Structure Used for Tree Nodes (Modified): Set the actual data to be stored to an aCard.

```
// structure for a node in a binary sorted tree
struct treeNode {
    aCard card;                    //Actual data to be stored, which is an aCard variable in this case
    treeNode* less, * more;         // pointers to node with data less or more than in this node
};
```

treeTop:

```
treeNode* treeTop;                  // pointer to top of tree
```

newNode (Modified): Takes an aCard value, creates a new node from it and returns a treeNode pointer to this node.

```
treeNode* newNode(aCard c) {
    treeNode* p = new treeNode;     // create space for node
    p->card = c;                    //Add data aCard c
    p->less = NULL;                 // pointers less and more are set to NULL
    p->more = NULL;
    return p;                       // return pointer to this new node
}
```

printTree (Modified): Takes a treeNode pointer and prints the tree from that node. Returns nothing. Calls itself recursively to print from the far end of the less sub tree to the far end of the more sub tree in order.

```
void printTree(treeNode* p) {
    if (p != NULL) {
        printTree(p->less);                    // print any nodes in less sub tree
        cout << cardToStr(p->card) << ", ";    //Print this node by using cardToStr to convert the card
to a string
        printTree(p->more);                    // print any nodes in more sub tree
    }
}
```

insertTree (Modified): Takes a treeNode pointer and an aCard, returns a treeNode pointer. If the node given as the argument is NULL, makes a new node from the given card and returns that node, otherwise calls itself recursively to find the correct relative position for the card/node and sets the node returned from it to the less/more of the node from the argument. Returns the pointer to the new node if it is set, or a pointer to the this node.

```
treeNode* insertTree(treeNode* p, aCard c) {
    treeNode* ans = p;
    if (p == NULL) ans = newNode(c);    // if found NULL pointer, create new node and this is returned
connecting to node above
    else if (compareCards(p->card, c) >= 0) p->less = insertTree(p->less, c);    //If c is less than the
card in the current node, insert the card in less by calling insertTree recursively as the less of this
node as p
    else p->more = insertTree(p->more, c);    //Else, insert the card in more by recursively calling
insertTree as more of this node as p
    return ans;         // return pointer to new node, or to this node, as appropriate
}
```

main: Main function that gets 20 cards from student ID and inserts it in the tree then prints the tree.

```
int main()
{
    cout << "Tree Program!\n";

    treeTop = NULL;                 //Initialising empty tree

    for (int ct = 0; ct < 20; ct++) //For 20 times
        treeTop = insertTree(treeTop, getCard("30021984")); //Get a card from the student ID and insert
it to the tree

    printTree(treeTop);             //Print the tree
}
```

Note: compareCards and cardToStr functions used in tree program are the same as the ones used in sorting program.

Results:

```
Tree Program!
AH, 3H, 9H, JH, KH, 2C, 6C, 7C, 8C, 8C, 8C, TC, 4D, 5D, 5D, 5D, 7D, JD, AS, TS,
```

Reflection

I enjoyed working with different methods of sorting and comparing them. My knowledge is increased on these concepts.