Module Code: CS1FC16

Assignment report Title: Card Sorting by Algorithms and Tree

Student Number: 30002734

Date (when the work completed): 16/03/2022

Actual hrs spent for the assignment: 15

Assignment evaluation (3 key points): Quicksort is confusing to program with cards. Binary trees are very interesting. I think the skeleton code was good and made sense, it was made to be developed and modified with ease.

Introduction

We were provided a card library header file, card library C++ source code file and a sortcard source code file with the goal of creating a program that can sort a deck of cards using bubble sort, quicksort, and a binary sorted tree. We must create the sorting functions ourselves and document them.

Sorting Program

There are 2 data structures in the header file we were provided. The first one is an enum type called suit. It contains the 4 suits of a deck of cards. The second structure is called aCard and it contains the card number and the card suit using the Suit enum.

Card to string function

This function takes in a card as a parameter. There is a string called csuit containing the first letter of each suit. This is so that the cardSuit enum can be used to take a substring of this string. Then the number is concatenated onto the letter, and this makes a new string that is returned by the function.

string cardToStr(aCard c) // takes a card and returns a string

{

string csuit = "HCDS";

return csuit.substr(c.cardSuit, 1) + to\_string(c.cardVal);

// c.cardSuit represents the suit, stored as an integer. If the cardSuit is 1 then the first letter of csuit is used.

// to\_string(c.cardSuit) gives the cards number and is concatenated to the string.

}

Print pack function

This function takes in a string which contains a message. The function prints out the message to the console then uses the card to string function to print out each card separated by a comma. A for loop is used to iterate through the pack of cards.

void printPack(string mess)

{

cout << mess

<< ":"

<< "\n"; // prints message

for (int ct = 0; ct < maxCard; ct++) // loops through the pack of cards

{

cout << cardToStr(thePack[ct]) << ", "; // prints each card seperated by a comma

}

cout << "\n";

}

Compare cards function

This function takes in 2 cards as parameters. Firstly, the two card’s suits are compared, if they both have the same suit, then the value is compared. If card 1 is greater, the function returns 1. If card 2 is greater, the function returns -1. If the 2 cards are equal, then the function returns 0. An integer named comp is used to store this value.

int compareCards(aCard c1, aCard c2) // function to compare cards.

{

int comp;

// H<C<D<S

if (c1.cardSuit > c2.cardSuit){

comp = 1; // c1 bigger than c2

}

else if (c1.cardSuit < c2.cardSuit){

comp = -1; // c1 smaller than c2

}

else if (c1.cardSuit == c2.cardSuit) // c1 and c2 are equal

{

if (c1.cardVal > c2.cardVal)

{

comp = 1; // c1 bigger than c2

}

else if (c1.cardVal < c2.cardVal)

{

comp = -1; // c1 smaller than c2

}

else if (c1.cardVal == c2.cardVal)

{

comp = 0; // c1 equals c2

}

}

// return -1 (c1 < c2), 0 (c1 == c2), 1 (c1 > c2)

return comp;

}

Swap card function

This function takes in 2 integers, n1 and n2. These are used to represent the indexes of the deck. There is a temporary card variable called temp that is used to store the value of thePack[n1]. Then thePack[n2] replaces thePack[n1], and then the temporary card replaces thePack[n2]. The cards are now swapped. This function doesn’t return a value.

void swapCards(int n1, int n2) // function to swap cards

{

aCard temp; // temporary variable

temp = thePack[n1];

thePack[n1] = thePack[n2];

thePack[n2] = temp; // n1 and n2 are swapped

}

Bubblesort function

This function takes in the pack of cards as a parameter. There are integer values for the number of moves and the number of comparisons. There are two for loops. The first one loops through the whole pack and the second one loops through the pack but each time it is called it does one less iteration. This is because each time the loop completes, 1 number will have bubbled to the top and be in the right place so there is no need to iterate more. Inside these loops is the compare cards function. If the two cards are in the wrong order, then they are swapped. With every iteration, the compare int is incremented and every time 2 cards are swapped, the moves int is incremented. At the end these numbers are outputted to the console.

void bubbleSort(aCard thePack[]) // bubbleSort function

{

int moves, compares; // variables to count the amount of moves and comparisons

moves = 0;

compares = 0;

for (int i = 0; i < maxCard; i++) // first loop goes through the whole pack

{

for (int j = 0; j < maxCard - i - 1; j++) // each time the pack is looped through, 1 card will have bubbled to the top so the loop has to start again but 1 less time.

{

compares++; // incrementing comparisons

if (compareCards(thePack[j], thePack[j + 1]) == 1) // compares two cards next to eachother

{

moves++;

swapCards(j, j + 1); // if the first card is larger, then the cards will be swapped.

}

}

}

cout << "Moves: " << moves << " Compares: " << compares << endl; // outputting the number of moves and comparisons.

}

Main function

This is the main function. There is a for loop to generate the pack of cards using my student number. The printpack function is called to print the unsorted pack. The bubblesort program is then called to sort the pack and then the pack is printed out again.

int main()

{

cout << "Card Sorting!\n";

for (int ct = 0; ct < maxCard; ct++) // for loop for each card

{

thePack[ct] = getCard("30002734"); // generate random card

}

printPack("Unsorted"); // prints unsorted pack

bubbleSort(thePack); // calls bubblesort function

printPack("Sorted"); // prints sorted pack

}

Quicksort function

this is the quicksort function. It takes in the pack of cards, a low index and a high index as parameters. Firstly the pivot is chosen as the first card. Then I and j are set as the indexes. Then there is a while loop that loops whilst I is less than j. this is to make sure they do not pass the pivot index. Then there are 2 more while loops to check if each card is bigger or smaller than the pivot. Then there are if statements to check if the cards should be swapped. if the low index is smaller than j the quicksort function is recursively called again. If I is less than the high index then the function is called again.

void qSort(aCard thePack[], int lndx, int hndx)

{

int i, j;

aCard pivot = thePack[lndx];

// sort in array between indices

// Calculate pivot

// in the middle

i = lndx;

j = hndx;

while (i < j)

{

while (compareCards(pivot, thePack[i]) == 1)

{

i++;

// find element > pivot to swap

}

while (compareCards(pivot, thePack[j]) == -1)

{

j--;

// find element < pivot to swap

}

if (i <= j)

{

if (i < j)

{

swapCards(i, j);

i++;

j--;

}

}

}

if (lndx < j)

qSort(thePack, lndx, j);

if (i < hndx)

qSort(thePack, i, hndx);

}

Sorting program results

Text

Description automatically generated

Tree Program

There are a series of nodes that are connected together. Each node connects to 2 more nodes, which contain data that are either less or more than the data in the current node. In this case, the data is a card. When a new node is added, the compare card function is used to compare the cards, if the card is greater or equal to the previous one, then the card is added to the right and if it is less, then it is added to the left. The print tree function is recursive, it prints the less nodes first, then it prints the higher ones. This produces a sorted list of cards.

New node function

This function takes in a card as a parameter. Firstly, a new node is created. The card is put into the node data and there are no less or more data yet. The new node is then returned.

treeNode \*newNode(aCard c)

{

// create a new node with data s,

// return pointer to it

treeNode \*p = new treeNode;

// create space for node

p->data = c; // add data

p->less = NULL; // pointers less and more

p->more = NULL; // are set to NULL

return p; // return pointer to new node

}

Insert into tree function

This function takes a tree node and a card. If the node is empty, then a card is set into the node data. Once the node has data, there is a check to see whether the node should be placed in the less or more connection using the compare cards function. Then this new node is returned.

treeNode \*insertTree(treeNode \*p, aCard c)

{

treeNode \*ans = p;

if (p == NULL)

ans = newNode(c); // data entered into this node

else if (compareCards(p->data, c) > 0) // checks if card is less than the previous one

p->less = insertTree(p->less, c);

else if (compareCards(p->data, c) <= 0) // checks if card is greater than or equal to the previous one

p->more = insertTree(p->more, c);

return ans;

// return pointer to new node, or this

}

Print tree function

This function takes in a tree node as a parameter. Firstly there is a check to see if the node is empty. The print tree function is called recursively until the p value in print tree is the lowest. Then each value is printed using card to string function and the tree is ran through until all values are printed out. They are separated by a comma. This function returns no value but prints out the cards.

void printTree(treeNode \*p)

{

// print the tree from node p

if (p != NULL)

{

printTree(p->less);

// print any nodes in less sub tree

cout << cardToStr(p->data) << ", "; // print node

printTree(p->more);

// print any nodes in more sub tree

}

}

Main function

This is the main function of the tree sort program. Firstly, the tree is set empty by treeTop = NULL. Then there is a for loop to generate the pack of random cards using my student number. Then the tree is generated with the pack. The print tree function is then called to print out the sorted tree.

int main()

{

cout << "RJM's Tree Program!\n";

treeTop = NULL; // initially an empty tree

for (int ct = 0; ct < maxCard; ct++) // for loop for each card

{

thePack[ct] = getCard("30002734"); // generates the pack of random cards

treeTop = insertTree(treeTop, thePack[ct]); // generates the tree with random cards

}

printTree(treeTop); // and print

cout << "\n";

}

Binary tree results

Text

Description automatically generated

Reflection

I enjoyed this project however less than the last one. The quicksort algorithm is confusing, and I think it should be worth more marks and the other parts of the coursework less marks. Once I got the algorithms finished it was very satisfying to see them working after lots of gruelling debugging. I thought the binary tree method was very interesting and is quite similar to quick sort. I am glad I learn these new things and I am starting to understand pointers a bit more.