**CSC 323-21: Project5: Radix Sort (CPP)**

**Adil Abuwani**

**Due date: Oct. 30, 2018**

Algorithm steps for computing:

Algorithm steps in Main(…)

Step 1: inFile 🡨 open the input file

outFile1 🡨 open the out file 1

outFile2 🡨 open the out file 2

Step 2: loadStack (inFile, Top) // see algorithm below

Step 3: RadixSort (Top) // see algorithm below

Step 4: Output sorted data to outFile1

listNode loadStack (inFile)

Step 1: Top 🡨 create a new stack and let Top points to it

largestNum 🡨 0

Step 2: data 🡨 read a data from inFile

If data > largestNum

largestNum 🡨 data // find the biggest positive number

Step 3: newNode <-- create a new listNode for the data

push (Top, newNode) 🡨 push newNode onto the top of the stack

step 4: repeat step 2 to step 3 until inFile is empty

step 5: call printStack (Top) // Print \*\*\* Below is the constructed stack \*\*\*

// then the stack, see format in the above.

dumpStack ( \*listNode Top, currentDigit, currentTable)

Step 1: // move each listNode from stack to hashTable[0]

- node 🡨 pop from the Top

- digit 🡨 getDigit (node, currentDigit)

// get the currentDigit of the data in the node, make sure it returns a single digit

- hashIndex 🡨 digit

- addTail (hashTable[currentTable][hashIndex], node)

// add the node at the tail of the queue at hashTable[currentTable][hashIndex]

Step 2: repeat step 1 until stack is empty

Step 3: printTable (hashTable[currentTable]) // call printQueue to print none-empty queues in hashTable.

RadixSort (\*listNode Top)

step 0: create hashTable[2][tableSize]

// create each linked list queue array, the head and tail point to a dummy node.

maxDigits🡨 getMaxDigits (largestNum) // determine the length of largestNum

Step 1: currentDigit 🡨 0 // the first digit/position from the right of the data.

currentTable 🡨 0

Step 2: dumpStack ( \*listNode Top, currentDigit, currentTable)

// see the algorithm in the above

Step 3: - currentDigit++

- previousTable 🡨 0

- currentTable 🡨 1

- currentQueue 🡨 0

Step 4: // moving nodes from previous table to current table

node 🡨 deleteHead from the currentQueue in

hashTable[previousTable][currentQueue]

digit 🡨 getDigit (node, currentDigit)

// get the digit from the currentDigit of the data in the node

hashIndex 🡨 hashIndex (digit)

addTail (hashTable[currentTable][hashIndex])

// add the node at the tail of the queue at hashTable[currentTable][hashIndex]

step 5: repeat steps 4 until the currentQueue is empty

Step 6: currentQueue ++ // process the next queue in the previous hashTable

Step 7: repeat step 4 to step 6 until currentQueue >= tableSize - 1

Step 8: printTable((hashTable[currentTable])

Step 9: temp 🡨 currentTable

currentTable 🡨 previousTable

previousTable 🡨temp

currentQueue 🡨 0

Step 10: repeat step 4 to step 9 while currentDigit < maxDigits

Step 11: printTable (hashTable[currentTable]) // print to outFile2

//call printQueue to print none-empty queues in hashTable.

Step 12: Close all files

b) Source code in CPP:

**#include** <iostream>

**#include** <fstream>

**#include** <cmath>

**using** **namespace** std;

**class** listNode{

**public**:

**int** data;

listNode\* next;

**listNode**(**int** d){

**this**->data=d;

**this**->next=NULL;

}//end constructor

//friends of these classes

**friend** **class** linkedListStack;

**friend** **class** linkedListQuene;

**friend** **class** RadixSort;

}; //class listNode

**class** linkedListStack{

**public**:

listNode\* top;

**linkedListStack**(){

**this**->top=NULL; //stack doesnot need a dummy node

}

**linkedListStack**(**const** linkedListStack\* s){

**this**->top=s->top;

}

**void** **push**(listNode\* newNode){ //end push

**if**(**this**->top==NULL){

**this**->top=newNode; //if top is null, set to top

}**else**{

newNode->next=top;

top=newNode;

}

}//end push

listNode\* **pop**(){

**if**(top==NULL){

**return** NULL;

}

listNode\* temp;

temp=top; //set temp to top

top=top->next; //move top to next

**return** temp; //return the top

}//end pop

**bool** **isEmpty**(){

**return** (**this**->top==NULL);

}

listNode\* **peak**(){

**return** top;

}

**void** **printStack**(linkedListStack\* theStack, ofstream& outFile){

linkedListStack\* temp=theStack;

outFile<<"Below is the output of the stack"<<**endl**;

**if**(temp->top==NULL){

outFile<<"Top -> NULL";

}**else**{

**while**(temp->top->next!=NULL){

outFile<<"("<<temp->top->data<<", "<<temp->top->next->data<<")->";

temp->top=temp->top->next;

}//end while

outFile<<"("<<temp->top->data<<", "<<"NULL) -> NULL";

}

}

};//end class

**class** linkedListQuene{

**public**:

listNode\* head;

listNode\* tail;

**linkedListQuene**(){

listNode\* dummy;

dummy= **new** listNode(-1); //listnode with a dummy -1 data

**this**->head=dummy;

**this**->tail=dummy;

}//end linkedListQueue

**void** **addTail**(listNode\* node){

node->next=NULL;

tail->next=node;

tail=node;

}//end addTail

listNode\* **deleteFront**(){

listNode\* temp;

**if**(**this**->head->next==NULL){

temp=NULL;

}**else**{

temp=head->next;

**this**->head->next=**this**->head->next->next;

**if**(head->next==NULL){

**this**->tail=head;

}

}//end else

**return** temp;

}

**bool** **isEmpty**(){

**return**(**this**->head->data==-1&&**this**->tail->data==-1); //is empty if the dummy are both -1

}//end isEmpty

**void** **printQueue**(**int** index, ofstream& outFile){

listNode\* temp;

temp=**this**->head->next;

listNode\* tempTail;

tempTail=**this**->tail;

outFile<<"Front ("<<index<<")->";

**while**(temp->next!=NULL){

outFile<<"("<<temp->data<<", "<<temp->next->data<<")->";

temp=temp->next;

}//end while

//last element

outFile<<"("<<temp->data<<", "<<"NULL) -> NULL"<<**endl**;

outFile<<"Tail ("<<index<<")->";

outFile<<"("<<tempTail->data<<", "<<"NULL) -> NULL"<<**endl**;

}//end printQueue

**void** **printQueueSorted**(**int** index, ofstream& outFile){

listNode\* temp;

temp=**this**->head->next;

**while**(temp->next!=NULL){

outFile<<temp->data<<", ";

temp=temp->next; //move temp to next

}//end while

//last element

outFile<<temp->data<<**endl**;

}

**friend** **class** RadixSort;

};

**class** RadixSort{

**public**:

**int** tableSize;

linkedListQuene\* hashTable[2][10];

**int** data;

**int** currentTable; //for 0 or 1

**int** previousTable; //for 0 or 1

**int** maxDigits; //the LENGTH of the largest integer in the data

**int** currentDigit; //which digit we are currently looking at in sorting

linkedListStack\* theStack;

**int** largestNum;

**RadixSort**(){

**this**->tableSize=10;

**this**->largestNum=0;

**this**->previousTable=0;

**this**->currentTable=0;

**this**->maxDigits=0;

**this**->data=0;

**this**->currentDigit=0;

**this**->tableSize=10;

**for** (**int** r=0; r<2; r++){

**for** (**int** c=0; c<10; c++){

**this**->hashTable[r][c] = **new** linkedListQuene();

}

}

**this**->theStack= **new** linkedListStack();

}//end constructor

**void** **loadStack**(ifstream& inFile, ofstream& outFile){

linkedListStack\* myStack;

myStack= **new** linkedListStack(); //top pointing to null

**int** largestNum=0;

**int** data;

**if**(inFile.is\_open()){

**while**(inFile>>data){

**if**(data>=largestNum){

largestNum=data;

}//end if

//step3

listNode\* myNode;

myNode=**new** listNode(data);

//push the data to the top

myStack->push(myNode);

**this**->theStack->push(myNode);

}//end while-repeat until eof baby

myStack->printStack(myStack, outFile); //printout myStack

**this**->largestNum=largestNum; //get the largestnum

}//end inFileis open

}//loadStack

**int** **getMaxDigits**(**int** largestNum){

**return** **ceil**(log10(largestNum));

} //DonaldTRUMP cant figure this out!!

**void** **printTable**(**int** currentTable, ofstream& outfile){

**for** (**int** i=0; i<10; i++){

**if** (!**this**->hashTable[currentTable][i]->isEmpty()){ //the current index is not empty

**this**->hashTable[currentTable][i]->printQueue(i, outfile);

}

}

}//end printTable

**void** **printTableSorted**(**int** currentTable, ofstream& outfile){

**for** (**int** i=0; i<10; i++){

**if** (!**this**->hashTable[currentTable][i]->isEmpty()){ //the current index is not empty

**this**->hashTable[currentTable][i]->printQueueSorted(i, outfile);

}

}

}//end printTable

**void** **dumpStack**(ofstream& outFile2){

//move each digit from top of stack to to hashTable 0

**while**(!**this**->theStack->isEmpty()){

listNode\* temp=**this**->theStack->pop();

**int** tempData=temp->data;

**int** digit=getDigit(tempData); //returns the digit

**int** hashIndex=digit;

**this**->hashTable[**this**->currentTable][hashIndex]->addTail(temp); //add data to tail

} //repeat step 1 until not empty

**this**->printTable(**this**->currentTable, outFile2);

}

**void** **radixSort**(ofstream& outFile1, ofstream& outFile2){

//create hastTable

createHashTable();

//get maxDigits

**this**->maxDigits=getMaxDigits(**this**->largestNum); //length of maxNum

**this**->currentTable=0;

**this**->dumpStack(outFile2); //first iteration-currentDigit=1;

**this**->currentDigit++; //move currentDigit to next as one iteration is complete

**this**->previousTable=0; //prevTable

**this**->currentTable=1; //currentTable is now 1

**int** currentQueue = 0; //begin from first quque

//move all the nodes from currentTable to currenttable

**while**(**this**->currentDigit<=**this**->maxDigits){ //1<3

**while**(currentQueue<10){

**while**(! **this**->hashTable[**this**->previousTable][currentQueue]->isEmpty()){

listNode\* node;

node=**this**->hashTable[**this**->previousTable][currentQueue]->deleteFront();

**int** digit=**this**->getDigit(node->data);

**int** hashIndex=digit;

**this**->hashTable[**this**->currentTable][hashIndex]->addTail(node);

}

currentQueue++;

}

**if**(**this**->currentDigit!=**this**->maxDigits){

**this**->printTable(**this**->currentTable, outFile2);

}//end if

**int** temp=**this**->currentTable;

**this**->currentTable=**this**->previousTable;

**this**->previousTable=temp;

currentQueue=0;

**this**->currentDigit++;

}//end while

//print sorted Data to outFile1

**this**->printTableSorted(**this**->previousTable, outFile1);

}//end radixSort

**int** **getDigit**(**int** cd){

**int** digit=cd;

**for**(**int** i=0;i<currentDigit; i++) {

digit=digit/10;

}

**return** digit % 10;

}

**int** **hashIndex**(**int** digit){

**return** digit;

}

**void** **createHashTable**(){

**for** (**int** r=0; r<2; r++){

**for** (**int** c=0; c<10; c++){

**this**->hashTable[r][c] = **new** linkedListQuene();

}

}

}//end createHashTable

};

**int** **main**(**int** argc, **char**\*\*argv) {

//step1

ifstream inFile1;

inFile1.open(argv[1]);

ofstream outFile1;

ofstream outFile2;

outFile1.open(argv[2]);

outFile2.open(argv[3]);

//step2

RadixSort\* theRadixSort= **new** RadixSort();

theRadixSort->loadStack(inFile1, outFile2);

outFile2<<**endl**<<**endl**; //newLine

theRadixSort->radixSort(outFile1, outFile2);

//close all files

inFile1.close();

outFile1.close();

outFile2.close();

**return** 0;

}

c) Input- a text file argv[1]:

19 322 702

8

9999

12 133

14

127

538 29

730

361

637 2255

213

388 91

322

739

4

95

16 702 8

95 568

6

d) OutputFile1- a text file arv[2]:

4, 6, 8, 8, 12, 14, 16, 19, 29, 91, 95, 95, 127, 133, 213, 322, 322, 361, 388, 538, 568, 637, 702, 702, 730, 739, 2255, 9999

d) OutputFile2- a text file arv[3]:

Below is the output of the stack

(6, 568)->(568, 95)->(95, 8)->(8, 702)->(702, 16)->(16, 95)->(95, 4)->(4, 739)->(739, 322)->(322, 91)->(91, 388)->(388, 213)->(213, 2255)->(2255, 637)->(637, 361)->(361, 730)->(730, 29)->(29, 538)->(538, 127)->(127, 14)->(14, 133)->(133, 12)->(12, 9999)->(9999, 8)->(8, 702)->(702, 322)->(322, 19)->(19, NULL) -> NULL

Front (0)->(730, NULL) -> NULL

Tail (0)->(730, NULL) -> NULL

Front (1)->(91, 361)->(361, NULL) -> NULL

Tail (1)->(361, NULL) -> NULL

Front (2)->(702, 322)->(322, 12)->(12, 702)->(702, 322)->(322, NULL) -> NULL

Tail (2)->(322, NULL) -> NULL

Front (3)->(213, 133)->(133, NULL) -> NULL

Tail (3)->(133, NULL) -> NULL

Front (4)->(4, 14)->(14, NULL) -> NULL

Tail (4)->(14, NULL) -> NULL

Front (5)->(95, 95)->(95, 2255)->(2255, NULL) -> NULL

Tail (5)->(2255, NULL) -> NULL

Front (6)->(6, 16)->(16, NULL) -> NULL

Tail (6)->(16, NULL) -> NULL

Front (7)->(637, 127)->(127, NULL) -> NULL

Tail (7)->(127, NULL) -> NULL

Front (8)->(568, 8)->(8, 388)->(388, 538)->(538, 8)->(8, NULL) -> NULL

Tail (8)->(8, NULL) -> NULL

Front (9)->(739, 29)->(29, 9999)->(9999, 19)->(19, NULL) -> NULL

Tail (9)->(19, NULL) -> NULL

Front (0)->(702, 702)->(702, 4)->(4, 6)->(6, 8)->(8, 8)->(8, NULL) -> NULL

Tail (0)->(8, NULL) -> NULL

Front (1)->(12, 213)->(213, 14)->(14, 16)->(16, 19)->(19, NULL) -> NULL

Tail (1)->(19, NULL) -> NULL

Front (2)->(322, 322)->(322, 127)->(127, 29)->(29, NULL) -> NULL

Tail (2)->(29, NULL) -> NULL

Front (3)->(730, 133)->(133, 637)->(637, 538)->(538, 739)->(739, NULL) -> NULL

Tail (3)->(739, NULL) -> NULL

Front (5)->(2255, NULL) -> NULL

Tail (5)->(2255, NULL) -> NULL

Front (6)->(361, 568)->(568, NULL) -> NULL

Tail (6)->(568, NULL) -> NULL

Front (8)->(388, NULL) -> NULL

Tail (8)->(388, NULL) -> NULL

Front (9)->(91, 95)->(95, 95)->(95, 9999)->(9999, NULL) -> NULL

Tail (9)->(9999, NULL) -> NULL

Front (0)->(4, 6)->(6, 8)->(8, 8)->(8, 12)->(12, 14)->(14, 16)->(16, 19)->(19, 29)->(29, 91)->(91, 95)->(95, 95)->(95, NULL) -> NULL

Tail (0)->(95, NULL) -> NULL

Front (1)->(127, 133)->(133, NULL) -> NULL

Tail (1)->(133, NULL) -> NULL

Front (2)->(213, 2255)->(2255, NULL) -> NULL

Tail (2)->(2255, NULL) -> NULL

Front (3)->(322, 322)->(322, 361)->(361, 388)->(388, NULL) -> NULL

Tail (3)->(388, NULL) -> NULL

Front (5)->(538, 568)->(568, NULL) -> NULL

Tail (5)->(568, NULL) -> NULL

Front (6)->(637, NULL) -> NULL

Tail (6)->(637, NULL) -> NULL

Front (7)->(702, 702)->(702, 730)->(730, 739)->(739, NULL) -> NULL

Tail (7)->(739, NULL) -> NULL

Front (9)->(9999, NULL) -> NULL

Tail (9)->(9999, NULL) -> NULL

Front (0)->(4, 6)->(6, 8)->(8, 8)->(8, 12)->(12, 14)->(14, 16)->(16, 19)->(19, 29)->(29, 91)->(91, 95)->(95, 95)->(95, 127)->(127, 133)->(133, 213)->(213, 322)->(322, 322)->(322, 361)->(361, 388)->(388, 538)->(538, 568)->(568, 637)->(637, 702)->(702, 702)->(702, 730)->(730, 739)->(739, NULL) -> NULL

Tail (0)->(739, NULL) -> NULL

Front (2)->(2255, NULL) -> NULL

Tail (2)->(2255, NULL) -> NULL

Front (9)->(9999, NULL) -> NULL

Tail (9)->(9999, NULL) -> NULL

Front (0)->(4, 6)->(6, 8)->(8, 8)->(8, 12)->(12, 14)->(14, 16)->(16, 19)->(19, 29)->(29, 91)->(91, 95)->(95, 95)->(95, 127)->(127, 133)->(133, 213)->(213, 322)->(322, 322)->(322, 361)->(361, 388)->(388, 538)->(538, 568)->(568, 637)->(637, 702)->(702, 702)->(702, 730)->(730, 739)->(739, NULL) -> NULL

Tail (0)->(739, NULL) -> NULL

Front (2)->(2255, NULL) -> NULL

Tail (2)->(2255, NULL) -> NULL

Front (9)->(9999, NULL) -> NULL

Tail (9)->(9999, NULL) -> NULL