**Sandesh Ghimire**

**CSCI 301 section 3**

**Computer Science 2**

**Project 8**

**3rd November**

**Design Document**

**Introduction:**

           Sorting is the process of arranging number, characters, or words systematically. We learned about different algorithms of sorting. In this program we did the algorithm for Heap sort, Merge sort, Quick sort and Insertion Sort. The Big O notation for Merge and Quick sort are (n\**log*n) and Insertion sort is *n*. This Proves that merge sort and Quick sort are faster than Insertion sort. We did it all theoretically on the lector but we are doing all of it practically to see which sort performs faster.

Insertion sort basically works by removing a value from the list, finds the location it belongs within the sorted list and replace it. Quick sort works by selecting a pivot. It then separates the list itself into two parts by checking which part is greater or smaller than pivot. Lastly, the separated sub list sorts recursively. Merge sorts works by dividing the list into different sub lists and sorts each and every sub list recursively. Heap sort works by using binary tree. In this program we used single array to represent the tree structure. Heap property has the largest value at the root and it decreases as we move down. It gives us advantage to sort the integers more efficiently.

**Data Structure**

           The data structure we used to do this program is array. We used the array that can hold random integer values. Then the program sorts the Array in ascending order using three different algorithms.

**Functions**

*int reheap\_down()* : This function is under the Priority Queue class which contains three parameter including an array and two integer variables. It starts with a loop with the condition weather the value of done variable is true. Under the loop we used a condition to check if the child value is smaller then the last value i.e. leaf and the value of array itself is greater. If the condition is fulfilled the index of child is increased by one. This follows another condition that checks of the parent value is larger than child value. If the condition is satisfied, the value of those array is exchanged giving the current index of child to the parent. If not satisfied the value of done is made true which terminates the loop.

*Queue::Queue():* It is a constructor that initializes the values of the priority queue.

*int Insertion()* : This function basically sorts the array that contains random integers. It starts with a for loop storing the value to the temp variable. It again starts a pre-condition loop to determine weather the is to be sorted or not. At last when the while loop ends the temp value is replaced back to the sorted position.

*int Quick()* : This function takes three parameter that contains an array, and two integer values. It basically sets up the recursion for the Quick sort algorithm.

*int MergeS()* : This function takes three parameter that contains an array, and two integer values. It basically sets up the recursion for the Merge sort algorithm.

*void Partition()* : It takes six parameters which includes an array and five integer variables. Tow of them are called by reference. This function is the heart of Quick sort. It performs the whole sorting algorithm. A while loop is used at first checking if the minimum value is less then the maximum value of not. If the condition is not satisfied the loop terminates. So, it can be called a pre-condition. Then the program starts the conation checking if the values in array are smaller then pivot or not. If the condition is satisfied the program swaps the values of the array.

*void merge()* : This function takes four parameters which includes an array and three integer variables. This function is the heart of the Merge sort algorithm. It starts with a for loop. Inside the for loop again a pre-conditional loop checking the high and mid values. If the condition is satisfied the value are inserted inside the array by sorting.

*void printArray*() : This function prints the arrays which are sorted by the using all the three sorting algorithm. It basically uses the for loop three times, printing every value in the array.

*Swap() and rand()* : Swap and rand are built in function gave by the IDE itself. Swap exchanges the two values. Rand generates the random integer within given boundary.

**The Main Program:**

The main function firstly asks some inputs for the users. After getting the input the array of the required length is generated. The values are inserted in each array by using rand function. Now the main function calls the all four functions of sorting algorithm, lastly printing the values if needed. The main function also asks user if they want to print the sorted list. If yes, the function prints it.

**Code**

**Main.cpp**

/\*Sorting is the process of arranging number, characters, or words systematically.

We learned about different algorithms of sorting. In this program we did the algorithm for Merge sort,

Quick sort and Insertion Sort. The Big O notation for Merge and Quick sort are (n\*logn) and Insertion sort is n.

This Proves that merge sort and Quick sort are faster than Insertion sort.

We did it all theoretically on the lector but we are doing all of it practically to see which sort performs faster.

Insertion sort basically works by removing a value from the list, finds the location it belongs within the sorted list and replace it.

Quick sort works by selecting a pivot.

It then separates the list itself into two parts by checking which part is greater or smaller than pivot.

Lastly, the separated sub list sorts recursively.

Merge sorts works by dividing the list into different sub lists and sorts each and every sub list recursively.

Heap sort works by using binary tree. In this program we used single array to represent the tree structure. Heap property has the largest value at the root and it decreases as we move down. It gives us advantage to sort the integers more efficiently.

\*/

#include <iostream>

#include "queue.h"

#include "queue.cpp"

#include<iomanip>

#include<cstdlib>

#include <cmath>

using namespace std;

int Insertion(int[], int);

int Quick(int[], int, int);

int MergeS(int[], int, int);

void printArray(int[], int[], int[], int);

int Partition(int[], int, int, int, int & , int & );

int Merge(int[], int, int, int);

int Num;//global variables

int main() { //The main function

int CountQuick = 0;

int CountInser = 0;

int CountMerge = 0;

int CountHeap = 0;

int Seed;

int Count = 0;

Queue l;

char Yn;

//Asking the input to the users

cout << "Enter the number of values to generate and sort, between 1 and 5000: ";

cin >> Num;

cout << " Enter an integer seed value: ";

cin >> Seed;

cout << "Print the values: ";

cin >> Yn;

srand(Seed);//ramdomizing the seed

int ToSort[Num];//declaring the array

int TempArray[Num];//declaring the array

int TempArray1[Num];//declaring the array

int TempArray2[Num];//declaring the array

for (int i = 1; i <= Num; i++) {//starting for loop to insert values to the array

ToSort[i] = (rand() % 5000) + 1;

TempArray[i] = ToSort[i];

TempArray1[i] = ToSort[i];

TempArray2[i]=ToSort[i];

}

CountInser=Insertion(ToSort, Num);//calling the function

CountQuick=Quick(TempArray, 0, Num);// calling the function

CountMerge=MergeS(TempArray1, 0, Num);//calling the function

for (int i=(Num-2)/2;i>=0;--i)

{

++CountHeap;

CountHeap=CountHeap+l.reheap\_down(TempArray2,i,Num-1);

}

for(int i=Num-1;i>0;--i)

{

++CountHeap;

swap(TempArray2[0],ToSort[i]);

CountHeap=CountHeap+l.reheap\_down(TempArray2,0,i-1);

}

if (Yn == 'y') {

printArray(ToSort, TempArray, TempArray1, Num);

}

cout<<endl;//Printing the values for counts

cout << "Count for Insertion Sort " << CountInser << endl;

cout << "Count for Merge Sort " << CountMerge << endl;

cout << "Count for Quick Sort " << CountQuick << endl;

cout << "Count for Heap Sort " << CountHeap << endl;

return 0;

}

int Quick(int TempArray[], int Min, int Max) {//quick function is used for recursion

int pivot;

int lastS1, FirstS3;

static int CountQuick=0;

if (Min < Max) {//starting the pre-condition

pivot = TempArray[Min];//giving value to pivot

CountQuick=Partition(TempArray, Min, Max, pivot, lastS1, FirstS3);//calling the function

Quick(TempArray, Min, lastS1);//calling the function

Quick(TempArray, FirstS3, Max);//calling the function

}

return CountQuick;

}

int Insertion(int ToSort[], int n) {//function to perform insertion sort

static int CountInser=0;

int j, Temp, i;//declaring variable

for (i = 1; i <= n; i++) {//starting for loop

Temp = ToSort[i];//giving the first value to the temp

j = i - 1;

while (j >= 0 && ToSort[j] > Temp) {//starting while loop

++CountInser;

ToSort[j + 1] = ToSort[j];//exchanging the value

j = j - 1;

}

ToSort[j + 1] = Temp;//giving back the temp value

}

return CountInser;

}

void printArray(int ToSort[], int TempArray[], int TempArray1[], int n) {//function to print the sorted arrays

int i;

int Count = 0;

cout << "=======================================================================================================================" << endl;

cout << endl;

cout << "-------------------------------------------------------Insertion Sort-----------------------------------------------------" << endl;

cout << endl;

for (i = 1; i < n; i++) {//starting for loop

cout << ToSort[i] << " ";//printing the datas in the array

}

cout << endl;

cout << endl;

cout << endl;

cout << "----------------------------------------------------------Quick sort--------------------------------------------------" << endl;

cout << endl;

for (i = 1; i < n; i++) {

cout << TempArray[i] << " ";

}

cout << endl;

cout << endl;

cout << "----------------------------------------------------------Merge sort--------------------------------------------------" << endl;

cout << endl;

for (i = 0; i < n - 1; i++) {

cout << TempArray1[i] << " ";

}

cout << endl;

cout << "=======================================================================================================================" << endl;

cout << endl;

cout << endl;

cout << "----------------------------------------------------------Heap sort--------------------------------------------------" << endl;

cout << endl;

for (i = 0; i < n - 1; i++) {

cout << TempArray1[i] << " ";

}

cout << endl;

cout << "=======================================================================================================================" << endl;

}

int Partition(int TempArray[], int Min, int Max, int pivot, int & i, int & j) {//function to sort using quick sort algorithm

int FirstU = Min;

int LastS1 = Min - 1;

int FirstS3 = Max + 1;

static int CountQuick;

while (FirstU < FirstS3) {//starting the pre-condition loop

++CountQuick;

if (TempArray[FirstU] < pivot) {//checking if first value us less then pivot

++LastS1;//increament

swap(TempArray[FirstU], TempArray[LastS1]);//exchanging the value

++FirstU;//increase value by 1

} else if (TempArray[FirstU] == pivot) {//another condition if first is not satisfied

++FirstU;

} else {//if all not

--FirstS3;//decreasing the value

swap(TempArray[FirstU], TempArray[FirstS3]);//exchanging the vale

}

}

i = LastS1;

j = FirstS3;

return CountQuick;

}

int MergeS(int TempArray1[], int low, int high) {//function for merge sort

int mid;

if (low < high) {//starting the pre condition

mid = (low + high) / 2;//giving mid value

MergeS(TempArray1, low, mid);//calling the function recursively

MergeS(TempArray1, mid + 1, high);//calling the function recursively

Merge(TempArray1, low, mid, high);//calling the function

}

}

int Merge(int TempArray1[], int low, int mid, int high) {//merge sort algorithm

int a[Num];//local copy of array

static int CountMerge;

int i1, i2, index;

for (index = low; index <= high; ++index) {//starting the for loop

a[index] = TempArray1[index];//giving the value to the local array

}

i1 = low;

i2 = mid + 1;

index = low;

while (i1 <= mid && i2 <= high) {//starting the while loop

++CountMerge;//increment

if (a[i1] < a[i2]) {//if condition to check the value of array

TempArray1[index] = a[i1];//give the value to the main array from local copy

++i1;//increment

} else {//if not

TempArray1[index] = a[i2];//give the value to the main array from local copy

++i2;//increment

}

++index;//increment

}

while (i1 <= mid) {//while loop for mid value

TempArray1[index] = a[i1];//give the value to the main array from local copy

++i1;//increment

++index;//increment

}

while (i2 <= high) {//while loop for high value

++CountMerge;//increment

TempArray1[index] = a[i2];//give the value to the main array from local copy

++i2;//increment

++index;//increment

}

return CountMerge;

}

**Queue.cpp**

#include <iostream>

#include "queue.h"

using namespace std;

Queue::Queue(){ //constructor

front=0;

rear=CAPACITY-1;

count=0;

}

void Queue::reheap\_up(int child)

{

int parent;

parent=(child-1)/2;

while(child>0 && data[child]>data[parent])

{

child=parent;

parent=(child-1)/2;

}

}

int Queue::reheap\_down(int a[],int parent,int last)

{

int max\_child; //declaring child variable

int CountHeap=0;//for counts

bool done;//for base condition

done=false;

while(2\*parent+1<=last && !done){ //starting the base condition loop

++CountHeap;//increment

max\_child=2\*(parent+1);

if(max\_child<last &&a[max\_child+1]>a[max\_child]){ //condition checking the max value

++max\_child;//increment

}

if (a[parent]<a[max\_child])

{

swap(a[parent],a[max\_child]);//exchanging the value

parent=max\_child;//giving current index to parent

}

else{

done=true;//else terminate loop

}

}

return CountHeap;//return the counts

}

void Queue::enqueue(int entry){//inserts the value into queue

data[count]=entry;

reheap\_up(count);

++count;

}

Queue::Item Queue::dequeue()//removes the item from queue

{

Item temp;

temp=data[0];

--count;

data[0]=data[count];

//reheap\_down(0);

return temp;

}

ostream& operator <<(ostream& out\_s, Queue& q){//friend function

Queue::Item it;

int n;

n=q.Size();

for(int i=0;i<n;++i){

it=q.dequeue();

out\_s<<it<<" ";

q.enqueue(it);

}

return out\_s;

}

**Queue.h**

#include <iostream>

#ifndef QUEUE\_H

#define QUEUE\_H

using namespace std;

class Queue{

public://public functions of the class

typedef int Item; //defining the Item using int

//Item first;

static const int CAPACITY=20;

Queue();

void enqueue(int entry);//to make list empty

int dequeue();//to insert the data in the list

int reheap\_down(Item a[],int parent,int last);//to maintain he heap structure

void reheap\_up(int child);

//Member functions

int Size(){return count;};//returns the size of queue

bool empty(){return count==0;}//returns if queue is empty or not

friend std::ostream& operator << (std::ostream& out\_s, Queue& Queue1); //this is a friend function

private://private functions of class

Item data[CAPACITY];

int front;//points front of queue

int rear;//points rear of queue

int count; //counts the number of data in queue

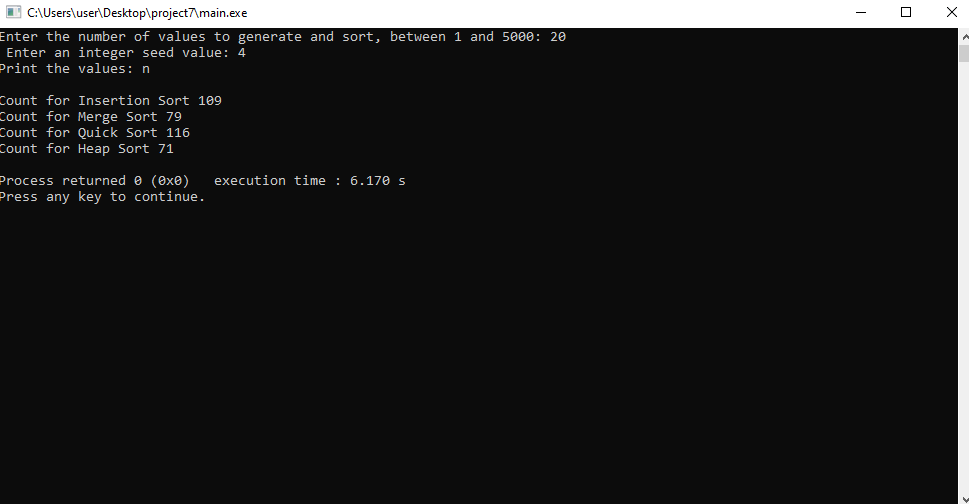
int next\_index(int i){ //Private member function

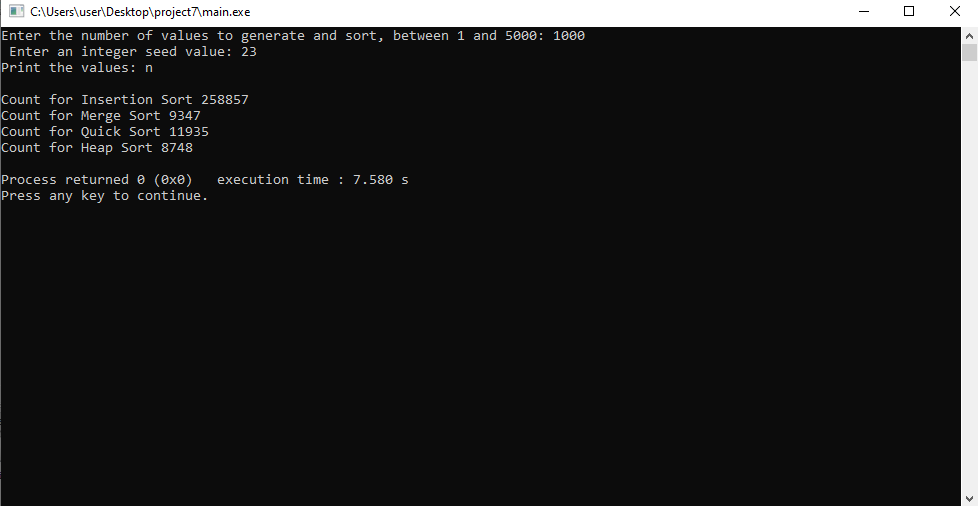
return (i+1)%CAPACITY;}

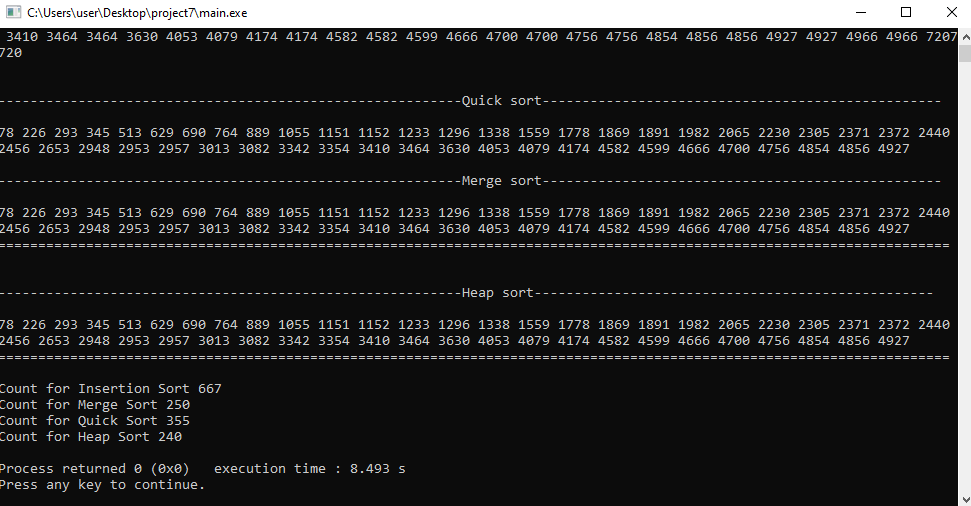
};

#endif // QUEUE\_H

**Test**

****

****

****

**User Document**

To compile the program simply enter:

*g++ -o main main.cpp*

To run the program, enter. */main,* then you can input an integer. When the program opens it asks user to enter values.

For example, when you run the program it asks input:

*Prompt>./main*

*Enter the number of values to generate and sort, between 1 and 5000: 700*

*Enter an integer seed value: 4*

*Print the values? (y/n) y*

If you enter those three lines the program displays the required output.

**Summary**

In this project, we learned about a new sorting algorithm i.e. heap sort. We did three different algorithms for sorting i.e. Insertion sort, merge sort, and quick sort before. Heap sort uses completely different approach then these three kinds of sort. Heap sort works by using binary tree. In this program we used single array to represent the tree structure. Heap property has the largest value at the root and it decreases as we move down. It gives us advantage to sort the integers more efficiently. According to the result we found that heap sort, merge sort and quick sort are efficient while we have a large amount of data. But Insertion sort was also good while we have small amount of data. This program proves that Heap sort is the most efficient sorting algorithm. We have graph for O(n^2) and O(nlogn). The graph suggest that it takes much less time for merge, quick and heap sort to complete the sorting then insertion sort. We got the same result above while we did it practically. When we used 1000 integers value to sort. Heap sort completed it within 8748 counts whereas insertion sort uses 258857 counts.

Yes, the results confirm our discussions of the algorithms' big-O times in class.

