SUBMITTED BY: NAVJOT KAUR

SFU ID: 301404765

SUBMITTED TO: POOYA TAHERI

ASSIGNMENT 2

QUESTION 1:

Implementation of sorting methods and their testing.

NOTE: The AVL tree's declaration and implementation are provided because of the inserting and sorting purposes in tree from an array.

- **AVL.h**

```
//Balanced Search tree (AVL)
#pragma once
#include <iostream>
using namespace std;

struct Node
{
	int data;
	Node* left;
	Node* right;
	int height;
};
class AVL
{
	Node* root;
	int max(int, int);
	Node* insert(int, Node*);
	Node* singleRightRotate(Node*&);
	Node* singleLeftRotate(Node*&);
	int height(Node*);
	int getBalance(Node*);
	void inOrder_helper(Node*);
	void destructor_helper(Node*);
public:
	AVL();
	~AVL();
	void insert(int);
	void inOrder();
};
```

- **AVL.cpp**

```
//This is AVL.cpp
#include "avl.h"
//PRE: none
//POST: returns the maximum of the two integers.
int AVL::max(int a,int b)
{
      if(a>b)
            return a;
      else
            return b;
}

//PRE: the argument should be an integer.
//POST: inserts the integer to the AVL tree and returns the inserted node.
Node* AVL::insert(int x, Node* node)
{
      //creates a new node everytime an integer is inserted.
      if(node==nullptr)
      {
            Node* temp=new Node;
            temp->data=x;
            temp->left=nullptr;
            temp->right=nullptr;
            temp->height=0;
            return temp;
      }
      //checks if the integer being inserted is smaller or greater than the
argument node
      if(x<node->data)
            node->left=insert(x,node->left); //if the integer is smaller then
the new node is inserted on the left side of the argument node.
      else if(x>node->data)
            node->right=insert(x,node->right); //if the integer is greater then
the new node is inserted on the right side of the argument node.
      else
            return node; //if the integer is equal to the node then return the
node because equal nodes are not allowed in BST.
      //checks the left subtree and right subtree's height in order to assign
the height of the argument node.
      int left_height=height(node->left);
      int right_height=height(node->right);
      node->height=1+max(left_height,right_height); // 1+ the maximun of the two
heights being compared is the desired height.
      //compute the balance factor of the node.
      int balance=getBalance(node);
      //if the tree becomes unbalanced i.e the balance factor exceeds 1 or
decreases from -1,
	//four cases(LL, RR, LR, RL) arise and the following codes checks the
balance factor
	//and rotates the tree accordingly.
	//LL case
	if(balance>1 && x<node->left->data)
		return singleRightRotate(node);
	//RR case
	if(balance<-1 && x>node->right->data)
		return singleLeftRotate(node);
	//RL case
	if(balance<-1 && x<node->right->data)
	{
		node->right=singleRightRotate(node->right);
		return singleLeftRotate(node);
	}
	//LR case
	if(balance>1 && x>node->left->data)
	{
		node->left=singleLeftRotate(node->left);
		return singleRightRotate(node);
	}
	return node;
}

//PRE: AVL tree is not empty and one of the nodes has lost its AVL property.
//POST: rotates the node of the tree to right so that the tree doesn't lose its
AVL properties
//	and returns the node being rotated.
Node* AVL::singleRightRotate(Node*& node)
{
	Node* temp1=node->left;
	Node* temp2=temp1->right;
	temp1->right=node;
	node->left=temp2;
	node->height=1+max(height(node->left),height(node->right));
	temp1->height=1+max(height(temp1->left),height(temp1->right));
	return temp1;
}

//PRE: AVL tree is not empty and one of the nodes has lost its AVL property.
//POST: rotates the node of the tree to left so that the tree doesn't lose its
AVL properties
//	and returns the node being rotated.
Node* AVL::singleLeftRotate(Node*& node)
{
	Node* temp1=node->right;
	Node* temp2=temp1->left;
	temp1->left=node;
	node->right=temp2;
	node->height=1+max(height(node->left),height(node->right));
	temp1->height=1+max(height(temp1->left),height(temp1->right));
	return temp1;
```

```
}


//PRE: none
//POST: returns the height of the tree.
int AVL::height(Node* node)
{
	if(node==nullptr)
		return -1;
	else
		return node->height;
}

//PRE: none
//POST: returns the balance factor of the tree.
int AVL::getBalance(Node* node)
{
	if(node==nullptr)
		return 0;
	else
	{
		int h=height(node->left)-height(node->right);
		return h;
	}
}

//PRE: AVL tree is not empty
//POST: displays the inorder traversal of the tree.
void AVL::inOrder_helper(Node* node)
{
	if(node)
	{
		inOrder_helper(node->left);
		cout << node->data << "    ";
		inOrder_helper(node->right);
	}
}

//PRE: AVL tree is not empty
//POST: displays the inorder traversal of the tree by using a helper function.
void AVL::inOrder()
{
	cout << "                      ";
	inOrder_helper(root);
	cout << endl;
}


//default constructor
AVL::AVL()
{
	this->root=nullptr;
```

```
}

//PRE: none
//POST: deletes the AVL tree.
void AVL::destructor_helper(Node* node)
{
	if(node!=nullptr)
	{
		destructor_helper(node->right);
		destructor_helper(node->left);
		delete node;
	}
}

//destructor
AVL::~AVL()
{
	destructor_helper(this->root);
}

//PRE: the argument should be an integer.
//POST: inserts the integer to the AVL tree
void AVL::insert(int x)
{
	root=insert(x,root);
}
```

- **MAIN FUNCTION**

```
#include "avl.h"
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
using namespace std;
using namespace std::chrono;
#define SIZE 20000

//MODIFIED BUBBLE SORT
//PRE: function takes array of integers and it is not empty.
//POST: sorts the provided array using modified bubble sort.
void bubbleSort(int A[], const int size)
{
	for(int i=0;i<size;i++)
	{
		int count=0;
		for (int j=0;j<size-1-i;j++)
		{
			if(A[j]>A[j+1])
			{
				int temp=A[j];
```

```
                        A[j]=A[j+1];
                        A[j+1]=temp;
                        count++;
                    }
                }
                if(count==0)
                    break;
        }
}



//SELECTION SORT
//PRE: function takes the array A of integers and it is not empty.
//POST: sorts the provided array using selection sort.
void selectionSort(int A[], const int size)
{
        for(int i=0;i<size;i++)
        {
                int index=i;
                for(int j=i+1;j<size;j++)
                {
                        if(A[j]<A[index])
                                index=j;
                }
                int temp=A[i];
                A[i]=A[index];
                A[index]=temp;
        }
}

//INSERTION SORT
//PRE: function takes the array A of integers and it is not empty.
//POST: sorts the provided array using insertion sort.
void insertionSort(int A[], const int size)
{
        for(int i=1;i<size;i++)
        {
                int temp=A[i];
                int j;
                for(j=i-1;j>=0;j--)
                {
                        if(A[j]>temp)
                                A[j+1]=A[j];
                        else
                                break;
                }
                A[j+1]=temp;
        }
}

//MERGE SORT
```

```
//PRE: function takes the array A of integers and it is not empty.
//POST: merges the two halves of the array A each of which is sorted
independently into the same array.
void merge(int A[], int start_index, int middle_index, int last_index)
{
      //creating temporary array where to merge
      int size=last_index-start_index+1;
      int temp[size];
      //as long as there are elements in the two halves,
      //copy the smallest element from either sub array
      int i=start_index, j=middle_index+1, k=0;
      while(i<=middle_index && j<=last_index)
      {
            if(A[i]<A[j])
                  temp[k++]=A[i++];
            else
                  temp[k++]=A[j++];
      }
      //the elements of one half is finished now copy the remaining elements if
any from the other array.
      while(i<=middle_index)
            temp[k++]=A[i++];
      while(j<=last_index)
            temp[k++]=A[j++];
      //copy the merged elements from the temp to A
      for(int i=start_index, k=0;i<=last_index;i++,k++)
            A[i]=temp[k];
}

//PRE: function takes the array A of integers and it is not empty.
//POST: sorts the provided array using merge sort recursively.
void mergeSort(int A[], int start_index, int last_index)
{
      if(start_index >= last_index)
            return;
      else
      {
            int middle_index=(start_index+last_index)/2;
            mergeSort(A, start_index, middle_index);
            mergeSort(A, middle_index+1, last_index);
            merge(A, start_index, middle_index, last_index);
      }
}

//QUICK SORT
//PRE: none
//POST: swaps the two integer values.
void swap(int& a,int& b)
{
      int temp=a;
      a=b;
      b=temp;
```

```
}

//QUICK SORT
//PRE: none
//POST: finds the random pivot element and swap it with the first element of the
array.
void pivot(int arr[], int first, int last)
{
	int pivot_index=rand()%((last-first+1)-first);
	int pivot=arr[pivot_index];
	swap(pivot, arr[first]);
}

//QUICK SORT
//PRE: none
//POST: sorts the given array by using random pivot found in above function.
int partition(int arr[], int first, int last)
{
	int pivot_index=first;
	int pivot=arr[first];
	for(int i=first+1;i<=last;i++)
	{
		if(arr[i]<=pivot)
		{
			pivot_index++;
			swap(arr[i], arr[pivot_index]);
		}
	}
	swap(arr[pivot_index],arr[first]);
	return pivot_index;
}

//QUICK SORT
//PRE: none
//POST: sorts the given array using partition and calling the quickSort function
recursively.
void quickSort(int arr[], int first, int last)
{
	if(first<last)
	{
		int pi=partition(arr,first,last);
		quickSort(arr,first,pi-1);
		quickSort(arr,pi+1,last);
	}
}

//HEAP SORT
//PRE: none
//POST: returns the left child of i^th node.
int leftChild(int i)
{
	return 2*i+1;
```

```
}

//HEAP SORT
//PRE: none
//POST: bubble down in order to sort the array making it a heap from index i.
void bubbleDown(int A[], int i, const int size)
{
	int child;
	int temp;
	for(temp=A[i];leftChild(i)<size;i=child)
	{
		child=leftChild(i);
		if(child!=size-1 && A[child]<A[child+1])
			child++;
		if(temp<A[child])
			A[i]=A[child];
		else
			break;
	}
	A[i]=temp;
}

//HEAP SORT
//PRE: function takes the array A of integers and it is not empty.
//POST: sorts the provided array using heap sort.
void heapSort(int A[], const int size)
{
	for(int i=size/2;i>=0;i--)
		bubbleDown(A, i, size);
	for(int j=size-1;j>0;j--)
	{
		swap(A[0],A[j]);
		bubbleDown(A,0,j);
	}
}

//RADIX SORT
//PRE: none
//POST: returns the maximum element of the array arr.
int getMax(int arr[], int n)
{
    int max=arr[0];
    for (int i=1; i<n; i++)
        if (arr[i]>max)
            max=arr[i];
    return max;
}

//RADIX SORT
//PRE: function takes the array A of integers and it is not empty.
//POST: sorts the provided array according to d digit.
void countSort(int arr[], int size, int d)
```

```
{
        int temp[size];
        int count[10]={0};
        //storing count
        for(int i=0; i<size; i++)
        {
                int x=(arr[i]/d)%10;
                count[x]++;
        }
    for(int i=1; i<10; i++)
                count[i]+=count[i-1];
        for(int i=size-1; i>=0; i--)
        {
                int x=(arr[i]/d)%10;
                temp[count[x]-1]=arr[i];
                count[x]--;
        }
        for(int i=0; i<size; i++)
                arr[i]=temp[i];
}

//RADIX SORT
//PRE: function takes the array A of integers and it is not empty.
//POST: sorts the provided array using radix sort.
void radixSort(int arr[], int size)
{
        int m=getMax(arr, size);
        for(int d=1; m/d>0; d*=10)
                countSort(arr, size, d);
}

int main()
{
        srand(time(0));
        cout << endl << endl;
        cout << "------------EXECUTION TIME FOR MODIFIED BUBBLE SORT-----------"
<< endl << endl;
        int arr[SIZE];
        for(int i=0;i<SIZE;i++)
                arr[i]=rand();
        auto start=high_resolution_clock::now();
        bubbleSort(arr, SIZE);
        auto stop=high_resolution_clock::now();
        auto duration=duration_cast<microseconds>(stop-start);
        cout << "Time taken when array is filled with random numbers between 0 and
RAND_MAX: ";
        cout << duration.count() << " microseconds." << endl;


        int arr1[SIZE];
        for(int i=0;i<SIZE;i++)
                arr1[i]=i+1;
        auto start1=high_resolution_clock::now();
        bubbleSort(arr1,SIZE);
        auto stop1=high_resolution_clock::now();
        auto duration1=duration_cast<microseconds>(stop1-start1);
        cout << "Time taken when array is already sorted: ";
        cout << duration1.count() << " microseconds." << endl;


        int arr2[SIZE];
        for(int i=0;i<SIZE;i++)
              arr2[i]=SIZE-i;
        auto start2=high_resolution_clock::now();
        bubbleSort(arr2,SIZE);
        auto stop2=high_resolution_clock::now();
        auto duration2=duration_cast<microseconds>(stop2-start2);
        cout << "Time taken when array is reversely sorted: ";
        cout << duration2.count() << " microseconds." << endl;


        int arr3[SIZE];
        for(int i=0;i<SIZE;i++)
              arr3[i]=rand()%6;
        auto start3=high_resolution_clock::now();
        bubbleSort(arr3,SIZE);
        auto stop3=high_resolution_clock::now();
        auto duration3=duration_cast<microseconds>(stop3-start3);
        cout << "Time taken when array is filled with random numbers between 0 and
5: ";
        cout << duration3.count() << " microseconds." << endl;


        cout << endl << endl;
        cout << "---------------EXECUTION TIME FOR SELECTION SORT---------------"
<< endl << endl;
        for(int i=0;i<SIZE;i++)
              arr[i]=rand();
        start=high_resolution_clock::now();
        selectionSort(arr, SIZE);
        stop=high_resolution_clock::now();
        duration=duration_cast<microseconds>(stop-start);
        cout << "Time taken when array is filled with random numbers between 0 and
RAND_MAX: ";
        cout << duration.count() << " microseconds." << endl;


        for(int i=0;i<SIZE;i++)
              arr1[i]=i+1;
        start1=high_resolution_clock::now();
        selectionSort(arr1,SIZE);
        stop1=high_resolution_clock::now();
        duration1=duration_cast<microseconds>(stop1-start1);
        cout << "Time taken when array is already sorted: ";
```

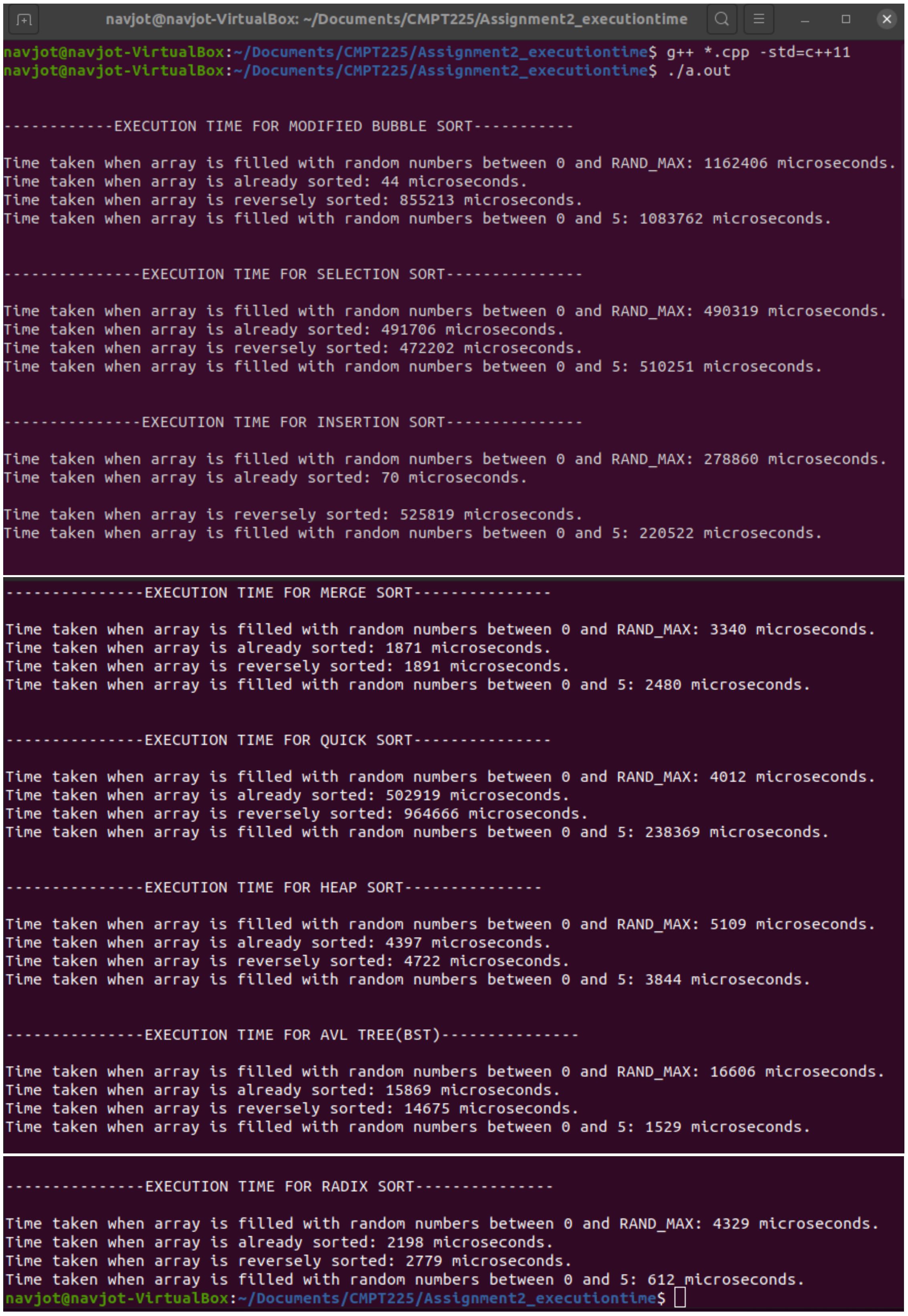
```
	cout << duration1.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr2[i]=SIZE-i;
	start2=high_resolution_clock::now();
	selectionSort(arr2,SIZE);
	stop2=high_resolution_clock::now();
	duration2=duration_cast<microseconds>(stop2-start2);
	cout << "Time taken when array is reversely sorted: ";
	cout << duration2.count() << " microseconds." << endl;

	for(int i=0;i<SIZE;i++)
		arr3[i]=rand()%6;
	start3=high_resolution_clock::now();
	selectionSort(arr3,SIZE);
	stop3=high_resolution_clock::now();
	duration3=duration_cast<microseconds>(stop3-start3);
	cout << "Time taken when array is filled with random numbers between 0 and 
5: ";
	cout << duration3.count() << " microseconds." << endl;


	cout << endl << endl;
	cout << "---------------EXECUTION TIME FOR INSERTION SORT---------------" 
<< endl << endl;
	for(int i=0;i<SIZE;i++)
		arr[i]=rand();
	start=high_resolution_clock::now();
	insertionSort(arr, SIZE);
	stop=high_resolution_clock::now();
	duration=duration_cast<microseconds>(stop-start);
	cout << "Time taken when array is filled with random numbers between 0 and 
RAND_MAX: ";
	cout << duration.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr1[i]=i+1;
	start1=high_resolution_clock::now();
	insertionSort(arr1,SIZE);
	stop1=high_resolution_clock::now();
	duration1=duration_cast<microseconds>(stop1-start1);
	cout << "Time taken when array is already sorted: ";
	cout << duration1.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr2[i]=SIZE-i;
	start2=high_resolution_clock::now();
	insertionSort(arr2,SIZE);
	stop2=high_resolution_clock::now();
	cout << endl;
	duration2=duration_cast<microseconds>(stop2-start2);
	cout << "Time taken when array is reversely sorted: ";
	cout << duration2.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr3[i]=rand()%6;
	start3=high_resolution_clock::now();
	insertionSort(arr3,SIZE);
	stop3=high_resolution_clock::now();
	duration3=duration_cast<microseconds>(stop3-start3);
	cout << "Time taken when array is filled with random numbers between 0 and 
5: ";
	cout << duration3.count() << " microseconds." << endl;


	cout << endl << endl;
	cout << "---------------EXECUTION TIME FOR MERGE SORT---------------" << 
endl << endl;
	for(int i=0;i<SIZE;i++)
		arr[i]=rand();
	start=high_resolution_clock::now();
	mergeSort(arr, 0, SIZE-1);
	stop=high_resolution_clock::now();
	duration=duration_cast<microseconds>(stop-start);
	cout << "Time taken when array is filled with random numbers between 0 and 
RAND_MAX: ";
	cout << duration.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr1[i]=i+1;
	start1=high_resolution_clock::now();
	mergeSort(arr1, 0, SIZE-1);
	stop1=high_resolution_clock::now();
	duration1=duration_cast<microseconds>(stop1-start1);
	cout << "Time taken when array is already sorted: ";
	cout << duration1.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr2[i]=SIZE-i;
	start2=high_resolution_clock::now();
	mergeSort(arr2, 0, SIZE-1);
	stop2=high_resolution_clock::now();
	duration2=duration_cast<microseconds>(stop2-start2);
	cout << "Time taken when array is reversely sorted: ";
	cout << duration2.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
```

```
		arr3[i]=rand()%6;
	start3=high_resolution_clock::now();
	mergeSort(arr3, 0, SIZE-1);
	stop3=high_resolution_clock::now();
	duration3=duration_cast<microseconds>(stop3-start3);
	cout << "Time taken when array is filled with random numbers between 0 and 
5: ";
	cout << duration3.count() << " microseconds." << endl;


	cout << endl << endl;
	cout << "---------------EXECUTION TIME FOR QUICK SORT---------------" << 
endl << endl;
	for(int i=0;i<SIZE;i++)
		arr[i]=rand();
	start=high_resolution_clock::now();
	quickSort(arr, 0, SIZE-1);
	stop=high_resolution_clock::now();
	duration=duration_cast<microseconds>(stop-start);
	cout << "Time taken when array is filled with random numbers between 0 and 
RAND_MAX: ";
	cout << duration.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr1[i]=i+1;
	start1=high_resolution_clock::now();
	quickSort(arr1, 0, SIZE-1);
	stop1=high_resolution_clock::now();
	duration1=duration_cast<microseconds>(stop1-start1);
	cout << "Time taken when array is already sorted: ";
	cout << duration1.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr2[i]=SIZE-i;
	start2=high_resolution_clock::now();
	quickSort(arr2, 0, SIZE-1);
	stop2=high_resolution_clock::now();
	duration2=duration_cast<microseconds>(stop2-start2);
	cout << "Time taken when array is reversely sorted: ";
	cout << duration2.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr3[i]=rand()%6;
	start3=high_resolution_clock::now();
	quickSort(arr3, 0, SIZE-1);
	stop3=high_resolution_clock::now();
	duration3=duration_cast<microseconds>(stop3-start3);
	cout << "Time taken when array is filled with random numbers between 0 and 
5: ";
```

```
	cout << duration3.count() << " microseconds." << endl;


	cout << endl << endl;
	cout << "---------------EXECUTION TIME FOR HEAP SORT---------------" << 
endl << endl;
	for(int i=0;i<SIZE;i++)
		arr[i]=rand();
	start=high_resolution_clock::now();
	heapSort(arr, SIZE);
	stop=high_resolution_clock::now();
	duration=duration_cast<microseconds>(stop-start);
	cout << "Time taken when array is filled with random numbers between 0 and 
RAND_MAX: ";
	cout << duration.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr1[i]=i+1;
	start1=high_resolution_clock::now();
	heapSort(arr1, SIZE);
	stop1=high_resolution_clock::now();
	duration1=duration_cast<microseconds>(stop1-start1);
	cout << "Time taken when array is already sorted: ";
	cout << duration1.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr2[i]=SIZE-i;
	start2=high_resolution_clock::now();
	heapSort(arr2, SIZE);
	stop2=high_resolution_clock::now();
	duration2=duration_cast<microseconds>(stop2-start2);
	cout << "Time taken when array is reversely sorted: ";
	cout << duration2.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr3[i]=rand()%6;
	start3=high_resolution_clock::now();
	heapSort(arr3, SIZE);
	stop3=high_resolution_clock::now();
	duration3=duration_cast<microseconds>(stop3-start3);
	cout << "Time taken when array is filled with random numbers between 0 and 
5: ";
	cout << duration3.count() << " microseconds." << endl;


	cout << endl << endl;
	cout << "---------------EXECUTION TIME FOR AVL TREE(BST)---------------" 
<< endl << endl;
	AVL tree;
	for(int i=0;i<SIZE;i++)
		arr[i]=rand();
	start=high_resolution_clock::now();
	for(int i=0;i<SIZE;i++)
		tree.insert(arr[i]);
	stop=high_resolution_clock::now();
	duration=duration_cast<microseconds>(stop-start);
	cout << "Time taken when array is filled with random numbers between 0 and 
RAND_MAX: ";
	cout << duration.count() << " microseconds." << endl;

	AVL tree1;
	for(int i=0;i<SIZE;i++)
		arr1[i]=i+1;
	start1=high_resolution_clock::now();
	for(int i=0;i<SIZE;i++)
		tree1.insert(arr1[i]);
	stop1=high_resolution_clock::now();
	duration1=duration_cast<microseconds>(stop1-start1);
	cout << "Time taken when array is already sorted: ";
	cout << duration1.count() << " microseconds." << endl;


	AVL tree2;
	for(int i=0;i<SIZE;i++)
		arr2[i]=SIZE-i;
	start2=high_resolution_clock::now();
	for(int i=0;i<SIZE;i++)
		tree2.insert(arr2[i]);
	stop2=high_resolution_clock::now();
	duration2=duration_cast<microseconds>(stop2-start2);
	cout << "Time taken when array is reversely sorted: ";
	cout << duration2.count() << " microseconds." << endl;


	AVL tree3;
	for(int i=0;i<SIZE;i++)
		arr3[i]=rand()%6;
	start3=high_resolution_clock::now();
	for(int i=0;i<SIZE;i++)
		tree3.insert(arr3[i]);
	stop3=high_resolution_clock::now();
	duration3=duration_cast<microseconds>(stop3-start3);
	cout << "Time taken when array is filled with random numbers between 0 and 
5: ";
	cout << duration3.count() << " microseconds." << endl;


	cout << endl << endl;
	cout << "---------------EXECUTION TIME FOR RADIX SORT---------------" << 
endl << endl;
	for(int i=0;i<SIZE;i++)
```

```
		arr[i]=rand();
	start=high_resolution_clock::now();
	radixSort(arr, SIZE);
	stop=high_resolution_clock::now();
	duration=duration_cast<microseconds>(stop-start);
	cout << "Time taken when array is filled with random numbers between 0 and 
RAND_MAX: ";
	cout << duration.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr1[i]=i+1;
	start1=high_resolution_clock::now();
	radixSort(arr1, SIZE);
	stop1=high_resolution_clock::now();
	duration1=duration_cast<microseconds>(stop1-start1);
	cout << "Time taken when array is already sorted: ";
	cout << duration1.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr2[i]=SIZE-i;
	start2=high_resolution_clock::now();
	radixSort(arr2, SIZE);
	stop2=high_resolution_clock::now();
	duration2=duration_cast<microseconds>(stop2-start2);
	cout << "Time taken when array is reversely sorted: ";
	cout << duration2.count() << " microseconds." << endl;


	for(int i=0;i<SIZE;i++)
		arr3[i]=rand()%6;
	start3=high_resolution_clock::now();
	radixSort(arr3, SIZE);
	stop3=high_resolution_clock::now();
	duration3=duration_cast<microseconds>(stop3-start3);
	cout << "Time taken when array is filled with random numbers between 0 and 
5: ";
	cout << duration3.count() << " microseconds." << endl;

	return 0;
}
```

OUTPUT WINDOW



navjot@navjot-VirtualBox: ~/Documents/CMPT225/Assignment2_executiontime

```
navjot@navjot-VirtualBox:~/Documents/CMPT225/Assignment2_executiontime$ g++ *.cpp -std=c++11
navjot@navjot-VirtualBox:~/Documents/CMPT225/Assignment2_executiontime$ ./a.out


------------EXECUTION TIME FOR MODIFIED BUBBLE SORT-----------

Time taken when array is filled with random numbers between 0 and RAND_MAX: 1162406 microseconds.
Time taken when array is already sorted: 44 microseconds.
Time taken when array is reversely sorted: 855213 microseconds.
Time taken when array is filled with random numbers between 0 and 5: 1083762 microseconds.


---------------EXECUTION TIME FOR SELECTION SORT---------------

Time taken when array is filled with random numbers between 0 and RAND_MAX: 490319 microseconds.
Time taken when array is already sorted: 491706 microseconds.
Time taken when array is reversely sorted: 472202 microseconds.
Time taken when array is filled with random numbers between 0 and 5: 510251 microseconds.


---------------EXECUTION TIME FOR INSERTION SORT---------------

Time taken when array is filled with random numbers between 0 and RAND_MAX: 278860 microseconds.
Time taken when array is already sorted: 70 microseconds.

Time taken when array is reversely sorted: 525819 microseconds.
Time taken when array is filled with random numbers between 0 and 5: 220522 microseconds.

---------------EXECUTION TIME FOR MERGE SORT---------------

Time taken when array is filled with random numbers between 0 and RAND_MAX: 3340 microseconds.
Time taken when array is already sorted: 1871 microseconds.
Time taken when array is reversely sorted: 1891 microseconds.
Time taken when array is filled with random numbers between 0 and 5: 2480 microseconds.


---------------EXECUTION TIME FOR QUICK SORT---------------

Time taken when array is filled with random numbers between 0 and RAND_MAX: 4012 microseconds.
Time taken when array is already sorted: 502919 microseconds.
Time taken when array is reversely sorted: 964666 microseconds.
Time taken when array is filled with random numbers between 0 and 5: 238369 microseconds.


---------------EXECUTION TIME FOR HEAP SORT---------------

Time taken when array is filled with random numbers between 0 and RAND_MAX: 5109 microseconds.
Time taken when array is already sorted: 4397 microseconds.
Time taken when array is reversely sorted: 4722 microseconds.
Time taken when array is filled with random numbers between 0 and 5: 3844 microseconds.


---------------EXECUTION TIME FOR AVL TREE(BST)---------------

Time taken when array is filled with random numbers between 0 and RAND_MAX: 16606 microseconds.
Time taken when array is already sorted: 15869 microseconds.
Time taken when array is reversely sorted: 14675 microseconds.
Time taken when array is filled with random numbers between 0 and 5: 1529 microseconds.

---------------EXECUTION TIME FOR RADIX SORT---------------

Time taken when array is filled with random numbers between 0 and RAND_MAX: 4329 microseconds.
Time taken when array is already sorted: 2198 microseconds.
Time taken when array is reversely sorted: 2779 microseconds.
Time taken when array is filled with random numbers between 0 and 5: 612 microseconds.
navjot@navjot-VirtualBox:~/Documents/CMPT225/Assignment2_executiontime$
```

TABLE OF EXECUTION TIMES

	Array containing 20000 integers from 0 to RAND_MAX	Sorted array of 20000 integers	Reversely sorted array of 20000 integers	Array containing 20000 integers from 0 to 5.
Modified Bubble Sort	**1162406**	**44**	**855213**	**1083762**
Selection Sort	**490319**	**491706**	**472202**	**510251**
Insertion Sort	**278860**	**70**	**525819**	**220522**
Merge Sort	**3340**	**1871**	**1891**	**2480**
Quick Sort	**4012**	**502919**	**964666**	**238369**
Heap Sort	**5109**	**4397**	**4722**	**3844**
AVL Tree	**16606**	**15869**	**14675**	**1529**
Radix Sort	**4329**	**2198**	**2779**	**612**

NOTE:

1. ***All the time calculations are in microseconds.***
2. ***Most efficient time durations are highlighted with yellow and green colors.***

QUESTION 2
HEAPS

- **Heap.h**

```
#pragma once
#include <iostream>
using namespace std;

class Heap{

public:
   // Constructors and Destructor

   // New empty Heap with default capacity.
   Heap();

   // New empty Heap with capacity c.
   Heap(int c);

   // New Heap with size s, consisting of pairs <Pi,Ei> where,
   // for 0 <= i < s, Pi is Priorities[i] and Ei is value Elements[i].
   // Capacity is s + c, where c is the "spare capacity" argument.
   // Requires: Priorities and Elements are of size at least s.
   Heap( const int * Priorities, const int * Elements, int s, int c);

   // New Heap with combined contents the two Heap arguments.
   // Size of the new Heap is the sum of the sizes of argument Heaps.
   // Capacity of the new Heap is its size plus the "spare capacity" c.
   Heap( const Heap & Heap1, const Heap & Heap2, int c );

   // Destructor.
   ~Heap();

   // Accessors
   bool empty() const {return hSize == 0;}; // True iff Heap is empty.
   int size() const { return hSize ;} ; // Current size of Heap.
   int capacity() const { return hCapacity ;} ; // Current capacity.
   int peekMin() const { return A[0].element ;} // Peek at minimum priority
element.
   int peekMinPriority() const { return A[0].priority ;} // Peek at minimum
priority.

   // Modifiers
   void insert( int element, int priority ); // Insert the pair
<element,priority>.
   int extractMin(); // Remove and return the highest (minimum) priority element.

private:
   class Pair{
      public:
        int element ;
        int priority ;
   };

   Pair* A ; // Array containing heap contents.
   int hCapacity ; // Max number of elements (= size of A).
   int hSize ; // Current number of elements.

   // Repairs ordering invariant after adding leaf at A[i].
   void trickleUp(int i);

   // Repairs ordering invariant for sub-tree rooted at index i,
   //   when A[i] may be have larger priority than one of its children,
   //   but the subtrees of its children are heaps.
   void trickleDown(int i);

   // Establishes ordering invariant for entire array contents.
   void heapify(); //(Same as "make_heap" in lectures.)

   // Useful for implementing trickle up and down
   void swap(int i,int j);
};
```

- **Heap.cpp**

```
#include "Heap.h"

//Constructor
//PRE: none
//POST: constructs an object of Heap with default capacity of 10.
Heap::Heap()
{
      hCapacity = 10 ;
      A = new Pair[hCapacity];
      hSize = 0 ;
}

//Non default constructor
//PRE: none
//POST: constructs an object of Heap with heap capacity c.
Heap::Heap(int c) // New empty Heap with capacity c.
{
      hCapacity = c;
      A = new Pair[hCapacity];
      hSize = 0;
}

//PRE: none
```

```
//POST: construct a heap with <priority, element> pair.
// New Heap with capacity c+s, with s elements, consisting of pairs <Pi,Vi> where
//  Pi is Priorities[i], Ei is value Elements[i], for 0 <= i < s.
Heap::Heap( const int * Priorities, const int * Elements, int s, int c)
{
	hCapacity = s+c;
	A = new Pair[hCapacity];
	hSize = 0;
	for(int i=0;i<s;i++)
		insert(Elements[i],Priorities[i]);
}

//PRE: none
//POST: constructs a heap from the elements of Heap1 and Heap2 combined.
// New Heap with combined contents and of the two given heaps.
Heap::Heap( const Heap & Heap1, const Heap & Heap2, int c )
{
	int size1=Heap1.hSize;
	int size2=Heap2.hSize;
	hCapacity = Heap1.hSize + Heap2.hSize + c ;
	A=new Pair[hCapacity];
	hSize=0;
	for(int i=0;i<Heap1.hSize;i++)
		insert(Heap1.A[i].element,Heap1.A[i].priority);
	for(int j=0;j<Heap2.hSize;j++)
		insert(Heap2.A[j].element,Heap2.A[j].priority);
}

// Destructor
Heap::~Heap()
{
	delete[] A;
}

// Modifiers
//PRE: none
//PRE: inserts an <priority, element> pair in heap.
void Heap::insert(int element, int priority)
{
	A[hSize].element = element;
	A[hSize].priority = priority;
	trickleUp(hSize);
	hSize ++;
}

//PRE: heap should not be empty
//POST: heapify the heap if it loses its heap order property.
// Repairs the heap ordering invariant after adding a new element.
// Initial call should be trickleUp(hSize-1).
void Heap::trickleUp(int i)
{
	if(i==0)
```

```
			return;
		else
		{
			int parent=(i-1)/2;
			if(A[i].priority<A[parent].priority)
			{
				swap(i,parent);
				trickleUp(parent);
			}
		}
}

//PRE: none
//POST: swaps two integer elements
void Heap::swap(int i, int j)
{
		Pair temp = A[i];
		A[i] = A[j];
		A[j] = temp ;
}

//PRE: heap shuld not be empty
//POST: returns and removes the minimum element of heap i.e the root node.
// Removes and returns the element with highest priority.
// (That is, the element associated with the minimum priority value.)
int Heap::extractMin()
{
		if(hSize>0)
		{
			Pair temp=A[0];
			swap(0,hSize-1);
			hSize--;
			trickleDown(0);
			return temp.element;
		}
}

//PRE: heap should not be empty
//POST: Repairs the heap ordering invariant after replacing the root.
// (extractMin() calls trickleDown(0)).
// (trickleDown(i) performs the repair on the subtree rooted a A[i].)
// (heapify() calls trickleDown(i) for i from (hSize/2)-1 down to 0.)
void Heap::trickleDown(int i)
{
		int min=i;
		int leftChild=2*i+1;
		int rightChild=2*i+2;
		if(leftChild<hSize && A[leftChild].priority<A[min].priority)
			min=leftChild;
		if(rightChild<hSize && A[rightChild].priority<A[min].priority)
			min=rightChild;
		if(min!=i)
```

```
        {
                swap(i,min);
                trickleDown(min);
        }
}

//PRE: none
//POST: heapify the heap so that it does not lose its properties.
// Turns A[0] .. A[size-1] into a heap.
void Heap::heapify()
{
        for( int i = (hSize/2)-1; i>=0 ; i-- ) trickleDown(i);
}
```

- **Heap_main.cpp**

```
/**************************************************************
   Test Program for Basic Heap Class - Preliminiary Version.
**************************************************************/
#include <iostream>
#include "heap.h"
using namespace std;

void heapTest1();
void heapTest2();
void heapTest3();
void heapTest4();

int main(){
      cout << "----------------TEST 1-----------------------" << endl << endl;
      heapTest1();
      cout << "----------------TEST 2-----------------------" << endl << endl;
      heapTest2();
      cout << "----------------TEST 3-----------------------" << endl << endl;
      heapTest3();
      cout << "----------------TEST 4-----------------------" << endl << endl;
      heapTest4();
      return 0;
}

/*
void Show( Heap & H, string s ){
      cout << s << ".capacity=" << H.capacity() << endl;
      cout << s << ".size=" << H.size() << endl;
      cout << s << "=" ; H.display(); cout << endl;
      cout << "-----------------------\n";
}
*/
```

```
void heapTest1(){

      // Test default constructor and basic functions
      Heap H;

      H.insert(91,7);
      H.insert(92,6);
      H.insert(93,8);
      H.insert(94,5);
      H.insert(95,9);
      while( !H.empty() ){
        cout << "min priority: " << H.peekMinPriority() << endl;
        cout << "min priority element: " << H.peekMin() << endl;
        H.extractMin();
      }

}

void heapTest2(){

      bool OK ;

      // Test TrickleUp
      // Use: default constructor, insert, peekMin, and peekMinPriority.
      Heap H;
      OK = true ;

      H.insert(91,7);
      if( H.peekMin() != 91 || H.peekMinPriority() != 7 ) OK = false ;
      H.insert(92,6);
      if( H.peekMin() != 92 || H.peekMinPriority() != 6 ) OK = false ;
      H.insert(94,5);
      if( H.peekMin() != 94 || H.peekMinPriority() != 5 ) OK = false ;
      H.insert(93,8);
      H.insert(95,9);
      H.insert(85,10);
      H.insert(84,12);
      if( H.peekMin() != 94 || H.peekMinPriority() != 5 ) OK = false ;
      H.insert(83,4);
      if( H.peekMin() != 83 || H.peekMinPriority() != 4 ) OK = false ;
      H.insert(82,6);
      H.insert(81,3);
      if( H.peekMin() != 81 || H.peekMinPriority() != 3 ) OK = false ;

      if( H.size() != 10 ) OK = false ;

      cout << OK << endl ;

      // Test extractMin
      // Use: insert, peekMin, peekMinPriority, extractMin
      Heap * HH = new Heap();
      OK = true ;
```

```
int x ;

HH->insert(91,7);
HH->insert(92,6);
HH->insert(94,5);
HH->insert(93,8);
HH->insert(95,9);
HH->insert(85,10);
HH->insert(84,12);
HH->insert(83,4);
HH->insert(82,6);
HH->insert(81,3);

// 3
if( HH->peekMin() != 81 || HH->peekMinPriority() != 3 ) OK = false ;
x = HH->extractMin();
if( x != 81 ) OK = false ;
if( HH->size() != 9 ) OK = false ;

// 4
if( HH->peekMin() != 83 || HH->peekMinPriority() != 4 ) OK = false ;
x = HH->extractMin();
if( x != 83 ) OK = false ;
if( HH->size() != 8 ) OK = false ;

// 5
if( HH->peekMin() != 94 || HH->peekMinPriority() != 5 ) OK = false ;
x = HH->extractMin();
if( x != 94 ) OK = false ;
if( HH->size() != 7 ) OK = false ;

// 6, 6
if( HH->peekMinPriority() != 6 ) OK = false ;
x = HH->extractMin();
if( HH->size() != 6 ) OK = false ;
x = HH->extractMin();
if( HH->size() != 5 ) OK = false ;

// 7
if( HH->peekMin() != 91 || HH->peekMinPriority() != 7 ) OK = false ;
x = HH->extractMin();
if( x != 91 ) OK = false ;
if( HH->size() != 4 ) OK = false ;

// 8
if( HH->peekMin() != 93 || HH->peekMinPriority() != 8 ) OK = false ;
x = HH->extractMin();
if( x != 93 ) OK = false ;
if( HH->size() != 3 ) OK = false ;

// 9
if( HH->peekMin() != 95 || HH->peekMinPriority() != 9 ) OK = false ;
```

```
      x = HH->extractMin();
      if( x != 95 ) OK = false ;
      if( HH->size() != 2 ) OK = false ;

      // 10
      if( HH->peekMin() != 85 || HH->peekMinPriority() != 10 ) OK = false ;
      x = HH->extractMin();
      if( x != 85 ) OK = false ;
      if( HH->size() != 1 ) OK = false ;

      // 12
      if( HH->peekMin() != 84 || HH->peekMinPriority() != 12 ) OK = false ;
      x = HH->extractMin();
      if( x != 84 ) OK = false ;
      if( HH->size() != 0 ) OK = false ;

      cout << OK << endl ;

      // Test Heap(c) constructor.
      OK = true ;
      Heap HHH(25);
      if( HHH.capacity() != 25 ) OK = false ;
      if( HHH.size() != 0 ) OK = false ;

      for( int i = 0 ; i < 25 ; i++ ){
          HHH.insert(i,i);
      }
      if( HHH.size() != 25 ) OK = false ;
      for( int i = 0 ; i < 25 ; i++ ){
          HHH.extractMin();
      }
      if( HHH.size() != 0 ) OK = false ;

      cout << OK << endl ;
}

void heapTest3(){

  int * ElementArr = new int[10];
  int * PriorityArr = new int[10];

  // Some priorities.
  // (In an order that is not a heap, to help spot bugs.)
  // The 999 is supposed to not end up in the heap.
  PriorityArr[0] = 9;
  PriorityArr[1] = 1;
  PriorityArr[2] = 7;
  PriorityArr[3] = 3;
  PriorityArr[4] = 2;
  PriorityArr[5] = 8;
  PriorityArr[6] = 999;
  // Some elements.
  // (Numbered so that the last digit is the corresponding
  // priority, and the first digit is the order they appear,
  // as a debugging aid.)
  // The 999 is supposed to not end up in the heap.
  ElementArr[0] = 109;
  ElementArr[1] = 201;
  ElementArr[2] = 307;
  ElementArr[3] = 403;
  ElementArr[4] = 502;
  ElementArr[5] = 608;
  ElementArr[6] = 999;

  // A heap made from the first 6 elements of the two arrays.
  Heap * H1 = new Heap( PriorityArr, ElementArr, 6, 10 );

  // A second heap.  Some priorities are distinct from
  // those in the first, and some are duplicates.
  Heap * H2 = new Heap(16);
  H2->insert(91,0);
  H2->insert(92,4);
  H2->insert(93,5);
  H2->insert(94,6);
  H2->insert(94,7);
  H2->insert(94,3);

  // A heap containing the union of pairs from the first two heaps.
  Heap * H3 = new Heap( *H1, *H2, 10 );

  cout << "Contents of heap H1:\n";
  while( !H1->empty() ){
      cout << "< " << H1->peekMinPriority() << ", " << H1->peekMin() << ">" <<
endl;
      H1->extractMin();
  }

  cout << "\nContents of heap H2:\n";
  while( !H2->empty() ){
      cout << "< " << H2->peekMinPriority() << ", " << H2->peekMin() << ">" <<
endl;
      H2->extractMin();
  }

  cout << "\nContents of heap H3:\n";
  while( !H3->empty() ){
      cout << "< " << H3->peekMinPriority() << ", " << H3->peekMin() << ">" <<
endl;
      H3->extractMin();
  }

}
```

```
void heapTest4(){


      // Test Heap(E,P,s,c)
      //
      //
      bool OK = true;

      int * ElementArr = new int[10];
      int * PriorityArr = new int[10];

      // Some priorities.
      // (In an order that is not a heap, to help spot bugs.)
      // The 999 is supposed to not end up in the heap.
      PriorityArr[0] = 9;
      PriorityArr[1] = 1;
      PriorityArr[2] = 7;
      PriorityArr[3] = 3;
      PriorityArr[4] = 2;
      PriorityArr[5] = 8;
      PriorityArr[6] = 999;

      // Some elements.
      // (Numbered so that the last digit is the corresponding
      // priority, and the first digit is the order they appear,
      // as a debugging aid.)
      // The 999 is supposed to not end up in the heap.
      ElementArr[0] = 109;
      ElementArr[1] = 201;
      ElementArr[2] = 307;
      ElementArr[3] = 403;
      ElementArr[4] = 502;
      ElementArr[5] = 608;
      ElementArr[6] = 999;

      // A heap made from the first 6 elements of the two arrays.
      Heap * H = new Heap( PriorityArr, ElementArr, 6, 10 );

      if( H->size() != 6 )  OK = false ;
      if( H->capacity() != 16 ) OK = false ;
      if( H->peekMin() != 201 || H->peekMinPriority() != 1 ) OK = false ;
      while( H->size() > 1 ){
         H->extractMin();
      }
      if( H->peekMin() != 109 || H->peekMinPriority() != 9 ) OK = false ;

      cout << OK << endl ;


      // Test Heap(H1,H2,c)
      //
```

```
        OK = true ;

        // A heap.
        Heap * H1 = new Heap(7);
        H1->insert(92,2);
        H1->insert(91,1);
        H1->insert(94,4);
        H1->insert(94,5);
        H1->insert(93,3);

        // A second heap.  Some priorities are distinct from
        // those in the first, and some are duplicates.
        Heap * H2 = new Heap(8);
        H2->insert(84,4);
        H2->insert(86,6);
        H2->insert(80,0);
        H2->insert(88,8);
        H2->insert(82,2);

        // A heap containing the union of pairs from the first two heaps.
        Heap * H3 = new Heap( *H1, *H2, 3 );

        if( H3->size() != 10 ) OK = false ;
        if( H3->capacity() != 13 ) OK = false ;
        if( H3->peekMin() != 80 ) OK = false ;
        if( H3->peekMinPriority() != 0 ) OK = false ;

        cout << OK << endl ;

}
```

Question 3
HASHING

PART 0:

- **Hashtable.h**

```
#pragma once

#include<iostream>
#include<iomanip>
using namespace std;

// A hash table class for mapping strings to ints.
// Note, this could be templated to allow mapping anything to anything.
class HashTable {
 public:
	// Insert a (key,value) pair into hash table.  Returns true if successful,
false if not.
	bool insert(string key, int value);

	bool remove(string key, int value);

	// Lookup a key in hash table.  Copies value to value if found and returns
true, returns false if key not in table.
	bool lookup(string key, int& value);

	// Modify a (key,value) pair in hash table.  Changes value to value if
found and returns true, returns false if key not in table.
	bool modify(string key, int value);

	// Return an array of all the keys in the table.  Stores these nkeys in
array keys.
	void getKeys(string*& all_keys, int& nkeys);
	// Return the number of (key,value) pairs currently in the table.
	int numStored();

	// Create a default sized hash table.
	HashTable();

	// Create a hash table that can store nkeys keys (allocates 4x space).
	HashTable(int nkeys);
	~HashTable();

	// Print the contents of the hash table data structures, useful for
debugging.
	void printTable();

 private:
	int tsize;  // size of hash table arrays
	int nstored;  // number of keys stored in table
	string *keys;
	int *values;
	int *sentinels; // 0 if never used, 1 if currently used, -1 if previously
used.
	static const int curr_used = 1;
	static const int never_used = 0;
	static const int prev_used = -1;

	static const int default_size = 10000;  // Default size of hash table.


	// Probing function, returns location to check on iteration iter starting
from initial value val.
	int probeFunction(int val, int iter);
	void init(int tsizei);


	int hash(string key);
	// A couple of hash functions to use.
	int sillyHash(string key);
	int charToInt(char c);
	int smarterHash(string key);

};
```

- **HashTable.cpp**

```
// Implement HashTable methods.
#include "HashTable.h"
using namespace std;

//PRE: hashtable is not full
//POST: inserts the string key and integer value in hashtable at a specified
index
//	calculated by smarterhash function using Horner's Rule,
//	returns true if insertion is complete and false otherwise.
bool HashTable::insert(string key, int value)
{
	// Try to insert key,value pair at pval, increment by probe function.
	int hval = hash(key);
	int pval = hval;

	for (int iter=0; iter<tsize; iter++) {
		if (sentinels[pval] != curr_used) {
			// Found an empty spot, insert the (key,value) pair here.
			sentinels[pval] = curr_used;
			keys[pval] = key;
			values[pval] = value;
			nstored++;
```

```
                    return true;
                }
                pval = probeFunction(hval,iter);
        }
        return false;
}

//PRE: hash table is not empty
//POST: removes the string key and integer value arguments from the hash table,
//       returns true if the removal is complete and false otherwise.
bool HashTable::remove(string key, int value)
{
        int hval=hash(key);
        int pval=hval;
        for(int i=0;i<tsize;i++)
        {
                if(sentinels[pval]==curr_used && keys[pval]==key &&
values[pval]==value)
                {
                        sentinels[pval]=prev_used;
                        keys[pval]="";
                        values[pval]=-1;
                        nstored--;
                        return true;
                }
                pval=probeFunction(hval,i);
        }
        return false;
}

//PRE: hashtable is not empty;
//POST: search for a specified key in the hash table, if found return true and
copies the value associated
//       with key in the integer argument value and false otherwise.
bool HashTable::lookup(string key, int& value)
{
        int hval=hash(key);
        int pval=hval;
        for(int i=0;i<tsize;i++)
        {
                if((sentinels[pval]==curr_used || sentinels[pval]==prev_used) &&
keys[pval]==key )
                {
                        value=values[pval];
                        return true;
                }
                pval=probeFunction(hval,i);
        }
        return false;
}

//PRE: hashtable is not empty
```

```
//POST: modifies the value associated with the key to the integer argument value,
if key is present
//      in the hash table return true and false otherwise.
bool HashTable::modify(string key, int value)
{
	int hval=hash(key);
	int pval=hval;
	for(int i=0;i<tsize;i++)
	{
		if((sentinels[pval]==curr_used || sentinels[pval]==prev_used) &&
keys[pval]==key)
		{
			values[pval]=value;
			return true;
		}
		pval=probeFunction(hval,i);
	}
	return false;
}

//PRE: none
//POST: Return an array of all the keys in the table.  Stores these nkeys in
array keys.
void HashTable::getKeys(string*& all_keys, int& nkeys)
{
	// Allocate an array to hold all the keys in the table.
	all_keys = new string[nstored];
	nkeys = nstored;


	// Walk the table's array.
	int key_i=0;
	for (int i=0; i<tsize; i++) {
		if (sentinels[i]==curr_used) {
			// Debug check: there shouldn't be more sentinels at curr_used
than nstored thinks.
			if (key_i > nkeys) {
				cerr << "Error: more keys in table than nstored
reports." << endl;
			}

			all_keys[key_i] = keys[i];
			key_i++;
		}
	}
}

//PRE: none
//POST: Return the number of (key,value) pairs currently in the table.
int HashTable::numStored()
{
	return nstored;
```

```
}

//PRE: none
//POST: returns an index for the key using smarterHash function.
int HashTable::hash(string key)
{
	return smarterHash(key);
}

//PRE: none
//POST: returns the index in the table where the key and value are to be stored.
int HashTable::probeFunction(int val, int iter)
{
	// Linear probing.
	return (val + iter) % tsize;
}

//PRE: none
//POST: returns the index using tsize/2, where the key and value are to be
stored.
int HashTable::sillyHash(string key)
{
	return tsize/2;
}

//PRE: takes a char argument
//POST: returns the integer data type value that is to be used in smarterHash.
int HashTable::charToInt(char c)
{
	if(c>='a' && c<='z')
		return c-96;
	else if(c>='A' && c<='Z')
		return c-64;
	else
		return 0;
}
//PRE: none
//POST: returns the index in the table where the key is to be stored using
Horner's method.
int HashTable::smarterHash(string key)
{
	int key_size=key.length();
	int x=0;
	if(key_size>1)
		x=((charToInt(key[0]))*32+(charToInt(key[1])))%tsize;
	else
		x=(charToInt(key[0]))%tsize;
	int index=1;
	if(key_size == 1)
		return x;
	for(int i=0;i<key_size-2;i++)
	{
```

```
            index++;
            x = (x*32 + (charToInt(key[index])))%tsize;
      }
      return x;
}

//PRE: none
//POST: constructs a hashtable of default size 10000.
//constructor
HashTable::HashTable()
{
      init(default_size);
}

//PRE: none
//POST: constructs a hashtable of size 4*nkeys.
//non default constructor
HashTable::HashTable(int nkeys)
{
      init(4*nkeys);
}

//PRE: none
//POST: allocate dynamic memory to the member variables of hash table.
void HashTable::init(int tsizei)
{
      tsize = tsizei;
      nstored = 0;

      keys = new string[tsize];
      values = new int[tsize];
      sentinels = new int[tsize];

      // Initialize all sentinels to 0.
      for (int i=0; i<tsize; i++)
            sentinels[i]=0;
}


//PRE: none
//POST: destructs the dynamically allocated memory.
HashTable::~HashTable()
{
      delete[] keys;
      delete[] values;
      delete[] sentinels;
}

//PRE: none
//POST: prints the hash table.
void HashTable::printTable()
{
	// Print the current state of the hashtable.
	// Note, prints actual data structure contents, entry might not be "in"
the table if sentinel not curr_used.
	// left, setw() are part of <iomanip>

	// Find longest string.
	const int indw = 5;
	int long_string = 3; // Length of "Key", nice magic number.
	const int intw = 10;
	const int sentw = 8;
	for (int i=0; i<tsize; i++) {
		if (keys[i].length() > long_string)
			long_string = keys[i].length();
	}

	// Print title
	cout << setw(indw) << left << "Index" << " | " << setw(long_string) <<
left << "Key" << " | " << setw(intw) << "Value" << " | " << "Sentinel" << endl;

	// Print a separator.
	for (int i=0; i < indw+long_string+intw+sentw+9; i++) {
		cout << "-";
	}
	cout << endl;

	// Print each table row.
	for (int i=0; i<tsize; i++) {
		cout << setw(indw) << left << i << " | " << setw(long_string) <<
left << keys[i] << " | " << setw(intw) << values[i] << " | " << sentinels[i] <<
endl;
	}

	// Print a separator.
	for (int i=0; i < indw+long_string+intw+sentw+9; i++) {
		cout << "-";
	}
	cout << endl;

}
```

- **Hashtable_main.cpp**

```
//============================================================================
// Name        : hashtable_test.cpp
//============================================================================

// A driver program to perform basic tests of hashtable functionality.

#include "HashTable.h"

#include <fstream>
```

```
#include <iostream>
#include <sstream>
#include <stdlib.h>

string int2letter(int i);




// This program takes command-line arguments
// The two arguments are:
//    test case to run, an integer
//    output filename (optional)
int main(int argc, char *argv[]) {

	// ostream for writing result.
	// Default to cout.
	// This code is to allow us to use either cout (an ostream) or an output
file (ofstream)
	ostream *outputp = &cout;
	ofstream outputfile;

	// Parse command-line arguments.
	int tcase=0;
	if (argc < 2 || argc > 3) {
		// Print error message, need a test case number.
		cerr << "Error: incorrect number of arguments" << endl;
		cerr << "  Usage:" << endl;
		cerr << "    " << argv[0] << " case_no [outfilename]" << endl;

		return 1;
	} else {
		// Turn the argument into an integer.
		tcase = atoi(argv[1]);

		if (argc > 2) {
			// Open the filename specified for output.
			outputfile.open (argv[2]);
			outputp = &outputfile;
		}
	}
	// This is a reference, which seems strange.  However, one can't directly
assign the *outputp to output because there is no copy constructor to use.
	ostream& output = *outputp;

	// cout << "Running case " << tcase << endl;


	/*
	  Test cases.
	  1: Insert a (key,value), get the (key,value).
```

```
        2: Insert a (key,value), modify the (key, value), lookup the
(key,value).
        3: Insert a (key,value), delete (key,value), try to lookup the
(key,value).
        4: Insert a (key,value), insert (key,value2) to cause collision, delete
(key,value), lookup the (key,value2).
        5: Many entries.  Insert until table is 75% full.  lookup (last key,
value)
        6: Many entries, test reuse.  Insert 200% of table size, with deletions
so table is never more than 50% full.  lookup (last key, value).
       */

      // This will make a hashtable with 100 spaces.
      HashTable ht(25);
      string key = "a";
      int val = 0;

      // Select test case.
      switch (tcase) {
      case 1: {
            ht.insert(key,20);
            ht.lookup(key,val);

            output << val << endl;
            break;
      } case 2:
            ht.insert(key,20);
            ht.modify(key,18);
            ht.lookup(key,val);

            output << val << endl;
            break;
      case 3:
            ht.insert(key,20);
            ht.remove(key,20);

            output << ht.lookup(key,val) << endl;
            break;
      case 4:
            ht.insert(key,20);
            ht.insert(key,18);
            ht.remove(key,20);
            ht.lookup(key,val);

            output << val << endl;
            break;
      case 5:
            for (int i=30; i<105; i++) {
                  ht.insert(int2letter(i),i);
            }

            key = "a";
```

```
			ht.insert(key,400);
			ht.lookup(key,val);

			output << val << endl;
			break;
		case 6:
			// Insert 50 to get started
			for (int i=0; i<50; i++) {
				ht.insert(int2letter(i),i);
			}

			for (int j=0; j<6; j++) {
				// Remove 25
				for (int i=25*j; i < 25*j + 25; i++) {
					ht.remove(int2letter(i),i);
				}
				// Add another 25
				for (int i=25*(j+1) ; i < 25*(j+2); i++) {
					ht.insert(int2letter(i),i);
				}
			}



			key = "a";
			ht.insert(key,400);
			ht.lookup(key,val);

			output << val << endl;
			output << "Number of pairs stored in the table: " << ht.numStored()
<< endl;
			break;
		default:
			cerr << "Invalid test case number" << endl;
			return 2;
		}


		// Close up the output file, if we had one.
		if (argc > 2) {
			outputfile.close();
		}

		return 0;
}


// Take an integer and convert it into a letter in A-Z.
// Does ASCII of 65 + (i % 26).
string int2letter(int i) {
	char the_char = (char) (65 + i % 26);
```

```
        // Put it in a string to return.
        string rtn_string(1,the_char);
        return rtn_string;
}
```

- **Word_frequencies.cpp**

```
// A program to calculate word frequencies in a single document.
#include <iostream>
#include <fstream>
#include "HashTable.h"

using namespace std;

#define MAX_STRING_LEN 256
#define DICT_SIZE 20000

// This program takes command-line arguments.
// By convention, these parameters are called argc (argument count) and argv
(argument vector).
// This program wants only a single argument, the filename of the document to be
processed.
int main (int argc, char* argv[]) {

        // Local variables.
        ifstream inputfile;  // ifstream for reading from input file.
        HashTable dict(DICT_SIZE);  // Dictionary for storing words and their
counts.

        // Parse command-line arguments.
        if (argc != 2) {
                // Note that the program name is the first argument, so argc==1 if
there are no additional arguments.
                cerr << "Expected one argument." << endl;
                cerr << "  Usage: " << argv[0] << " input_filename" << endl;
                return 1;
        } else {
                // Open the filename specified for input.
                inputfile.open (argv[1]);
        }



        // Tokenize the input.
        // Read one character at a time.
        // If the character is not in a-z or A-Z, terminate current string.
        char c;
        char curr_str[MAX_STRING_LEN];
        int str_i = 0;  // Index into curr_str.
```

```
bool flush_it = false;  // Whether we have a complete string to flush.

while (inputfile.good()) {
      // Read one character, convert it to lowercase.
      inputfile.get(c);
      c = tolower(c);

      if (c >= 'a' && c <= 'z') {
            // c is a letter.
            curr_str[str_i] = c;
            str_i++;

            // Check over-length string.
            if (str_i >= MAX_STRING_LEN) {
                  flush_it = true;
            }
      } else {
            // c is not a letter.
            // Create a new string if curr_str is non-empty.
            if (str_i>0) {
                  flush_it = true;
            }
      }

      if (flush_it) {
            // Create the new string from curr_str.
            string the_str(curr_str,str_i);
            // cout << the_str << endl;


            // Handle the string, insert/increment count in dictionary.
            int count;
            if (dict.lookup(the_str,count)) {
                  dict.modify(the_str,count+1);
            } else {
                  dict.insert(the_str,1);
            }


            // Reset state variables.
            str_i = 0;
            flush_it = false;
      }
}


// Get the hash table as a vector, print it.
string *keys;
int nkeys;
dict.getKeys(keys,nkeys);
for (int i=0; i<nkeys; i++) {
      int count;
```

```
            // Lookup key in table, store value in count.
            if (!dict.lookup(keys[i],count)) {
                  // Return with an error if not in table.
                  cerr << "Key not in table" << endl;
                  return 2;
            }
            cout << keys[i] << ": " << count << endl;
      }


      // Close input file.
      inputfile.close();
      return 0;
}
```

PART 1:
Answers.txt

Naive 1.

a) What is the purpose of the "break" and if statement "if j < i"? I.e. if you remove "if j < i" and "break" (and the corresponding "end"), what would be different about the output?

ANSWER: The purpose of "break" statement and if statement "if j<i" is to break the inner loop i.e, terminate the inner for loop. This will print the original count of the elements of array, however overprinting them.

Let an array of size 6, words[6]={“ab”, “ab”, “hash”, “hash”, “trash”, “bash”};

Output:

ab:1 (because i=1 and does not count the 0th element.)

hash:2

hash:0 (overprints because the print statement is in the outer for loop and not increasing the value of count, hence printing the value one time)

trash:1

bash:1

If the “break” and if statement is removed, then it will print the occurrences of the same elements multiple times and overprints them because the printing is happening in outer loop.

Output when if and break statements are removed:

```
ab:1

hash:2

hash:2(printing the count of elements multiple times)

trash:1

bash:1
```

b) What is the time complexity of the naive word frequency computation algorithm. Use variable w, the number of words in a document.

ANSWER: The time complexity is: O(w^2)

Naive 2.

a) What is the time complexity of the naive document finding algorithm? Use variables d, the number of documents, and w, the maximum number of words in each document.

ANSWER: The time complexity is: O(d*w)

b) Suppose we are building a large search engine, and many users want to run queries to find documents containing words. If we have u users who each want to run a query for a different keyword, what is the time complexity of *all* of their searches?

ANSWER: The time complexity is: O((d*w) ^u)

Part 1

a) What is the average case time complexity of the pseudocode for word_frequencies, using the provided, naive hash function? Define any variables you need to use, and explain your answer.

ANSWER: The average case time is: O(n^2). Using Horner's Rule to lookup for any word will reduce the time complexity. By computing hashValue (index in the table where a key is to be stored) and modifying the hashtable.cpp using Horner’s Rule in smarterHash function (already implemented) will make reduce the time complexity.

b) If we used a "good" hash function, what do you think the average case time complexity would be?

ANSWER: The time complexity for "good" hash function is: O(n).

PART 2

- **InvertedIndex.h**

```
#pragma once

#include <iostream>
#include <string>
#include <set>
using namespace std;

// An inverted index class for mapping strings to sets of strings.
// Underlying data structure is a hash table.

class InvertedIndex {
 public:
      // Lookup a key in index.  Copies values to values if found and returns
true, returns false if key not in index
      bool lookup(string key, set<string>& value);

      // Add a <key,value> pair to the index.  If the key has not previously
been stored, create an entry.  Add the value to it.
      bool add(string key, string value);

      // Create a default sized index.
      InvertedIndex();

      // Create an index that can store nkeys keys (allocates 4x space).
      InvertedIndex(int nkeys);
      ~InvertedIndex();

      // Print the contents of the index.
      void printIndex();

      // Returns the number of keys stored.
      int numStored();

 private:
      int tsize;  // size of hash table arrays
	int nstored;  // number of keys stored in table
	string *keys;
	set<string> *values;
	int *sentinels; // 0 if never used, 1 if currently used, -1 if previously
used.
	static const int curr_used = 1;
	static const int never_used = 0;
	static const int prev_used = -1;

	static const int default_size = 10000;  // Default size of hash table.


	// Probing function, returns location to check on iteration iter starting
from initial value val.
	int probeFunction(int val, int iter);  //linear probing
	int quad_prob(int val, int iter);      //quadratic probing
	void init(int tsizei);

	// Hash functions.
	int hash(string key);
	int smarterHash(string key);
	int char26(char);

};
```

- **InvertedIndex.cpp**

```
// Implement InvertedIndex methods
#include "InvertedIndex.h"
using namespace std;

//PRE: hashtable is not full
//POST: inserts the string key and string value at a specified index
//	 calculated by smarterhash function using quadratic probing,
//	 returns true if insertion is complete and false otherwise.
bool InvertedIndex::add(string key, string value) {
	// Either find the key or find an empty bucket in the table.
	int hval = hash(key);
	int pval = hval;

	for (int iter=0; iter<tsize; iter++) {
		if (sentinels[pval] == curr_used && keys[pval]==key) {
			// Found the key, add the value to the set (it checks for
duplicates).
			values[pval].insert(value);

			return true;
		} else if (sentinels[pval] != curr_used) {
```

```
                    // The key wasn't previously inserted and we've found an empty
spot.
                    // Insert the (key,value) pair here.
                    sentinels[pval] = curr_used;
                    keys[pval] = key;
                    values[pval].insert(value);
                    nstored++;
                    return true;
             }
             // pval = probeFunction(hval,iter); //linear probing.
             pval=quad_prob(hval,iter); //quadratic probing.
      }

      // TO DO:: Grow if trouble.

      return false;
}

//PRE: none
//POST: Print all the values in the index.
void InvertedIndex::printIndex() {
      for (int i=0; i < tsize; i++) {
             // If this entry is used, print the key and its set of values.
             if (sentinels[i] == curr_used) {
                    cout << keys[i] << " maps to {";
                    for (set<string>::iterator it = values[i].begin(); it !=
values[i].end(); it++) {
                           cout << *it << ", ";
                    }
                    cout << "}" << endl;
      }
}


//PRE: hashtable is not empty;
//POST: search for a specified key at an index, if found return true and copies
the value associated
//       with key in the value argument, and false otherwise.
bool InvertedIndex::lookup(string key, set<string>& value) {
      // Start search at pval, increment by probe function until found or unused
spot encountered.
      int hval = hash(key);
      int pval = hval;

      for (int iter=0; iter<tsize; iter++) {
             if (keys[pval] == key && sentinels[pval] == curr_used) {
                    value = values[pval];
                    return true;
             } else if (sentinels[pval] == never_used) {
                    return false;
             }
```

```
            // pval = probeFunction(hval,iter);
            pval = quad_prob(hval,iter);
      }
      return false;
}


//PRE: none
//POST: returns an index for the key using smarterHash function.
int InvertedIndex::hash(string key) {
      return smarterHash(key);
}

//PRE: none
//POST: returns the index where the key and value are to be stored using linear
probing.
int InvertedIndex::probeFunction(int val, int iter) {
      // Linear probing.
      return (val + iter) % tsize;
}

//PRE: none
//POST: returns the index where the key and value are to be stored using
quadratic probing.
int InvertedIndex::quad_prob(int pval, int iter)
{
      //Quadratic Probing
      return (pval + iter*iter)%tsize;
}

//PRE: none
//POST: returns the index in the table where the key is to be stored using
Horner's method.
int InvertedIndex::smarterHash(string key) {
      // Return hash function value for str.
      // Use base 32 representation mod table size as hash function.
      // Compute using Horner's rule to avoid overflow.

      // Handle empty strings.
      if (key.length() == 0) {
            return 0;
      } else {
            int base = 32;
            int sum = char26(key.at(0));
            for (int i=1; i<key.length(); i++) {
                  sum = (base*sum + char26(key.at(i))) % tsize;
            }
            return sum;
      }
}

//PRE: takes a char argument
```

```
//POST: returns the integer data type value that is to be used in smarterHash.
int InvertedIndex::char26(char c) {
	// Return the character c as a number in 1-26.
	// Case-insensitive, returns 0 if c is outside of A-Z.
	int diffa = 0;
	if (c >= 'A' && c <= 'Z') {
		diffa = c - 'A' + 1;
	} else if (c >= 'a' && c <= 'z') {
		diffa = c - 'a' + 1;
	} else {
		diffa = 0;
	}

	return diffa;
}


//PRE: none;
//POST: constructs a hashtable of default size 10000
//Constructor
InvertedIndex::InvertedIndex() {
	init(default_size);
}

//PRE: none
//POST: constructs a hashtable of size 4*nkeys.
//non default constructor
InvertedIndex::InvertedIndex(int nkeys) {
	init(4*nkeys);
}

//PRE: none
//POST: allocate dynamic memory to the member variables of hash table.
void InvertedIndex::init(int tsizei) {
	tsize = tsizei;
	nstored = 0;

	keys = new string[tsize];
	values = new set<string>[tsize];
	sentinels = new int[tsize];

	// Initialize all sentinels to 0.
	for (int i=0; i<tsize; i++)
		sentinels[i]=0;
}


//PRE: none
//POST: destructs the dynamically allocated memory.
InvertedIndex::~InvertedIndex() {
	delete[] keys;
	delete[] values;
	delete[] sentinels;
}

//PRE: none
//POST: returns the number of keys stored.
int InvertedIndex::numStored() {
	return nstored;
}
```

- **InvertedIndex_main.cpp**

```
/*
  index_directory.cpp

  * Load the contents of a directory into a hashtable, allow searches.
 */

#include <iostream>
// dirent.h is a library for reading directory entries.
// It defines DIR, the struct dirent, and other functions/constants used below.
#include <dirent.h>
#include <vector>
#include <fstream>
#include <string.h>
#include "InvertedIndex.h"

using namespace std;

#define MAX_STRING_LEN 256
#define INDEX_SIZE 100000


void processDirectory (const char * dname, vector<string> valid_extensions,
InvertedIndex& inverted_index, int& nfiles, int& ndirs);
bool validExtension (char *extension, vector<string> valid_extensions);
void processFile(const char *fname, InvertedIndex& inverted_index);



// This program processes all files in all subdirectories rooted at a directory.
// It builds an inverted index from these files.
// It then answers queries.
int main (int argc, char* argv[]) {

	// Parse command-line arguments.
	if (argc < 2) {
		// Note that the program name is the first argument, so argc==1 if
there are no additional arguments.
		cerr << "Expected one argument." << endl;
```

```
            cerr << "  Usage: " << argv[0] << " input_dirname [list of
.extensions]\n    e.g. " << argv[0] << " .  .cpp .h" << endl;
            cerr << "indexes all files if no extensions are provided." << endl;
            return 1;
      } else {
            vector<string> valid_extensions;
            for (int i=2; i<argc; i++) {
                  string ext(argv[i]);
                  valid_extensions.push_back(ext);
            }


            // Build an inverted index.
            InvertedIndex inverted_index(INDEX_SIZE);

            // Numbers of files and directories.
            int nfiles = 0;
            int ndirs = 0;

            // Open the directory name specified for input.
            processDirectory(argv[1], valid_extensions, inverted_index, nfiles,
ndirs);

            // inverted_index.printIndex();
            cout << "Indexed " << argv[1] << endl;
            cout << "  found " << ndirs << " subdirectories and " << nfiles << "
files" << endl;
            cout << "  built index with " << inverted_index.numStored() << "
keys" << endl;


            // Allow a user to query the index.
            string terminate_str = "q";
            string input;
            set<string> results;
            while (1) {
                  cout << "Enter a word to search for, " << terminate_str << "
to terminate" << endl;
                  cin >> input;

                  // Clear results.
                  results.clear();

                  if (input == terminate_str) {
                        break;
                  } else {
                        inverted_index.lookup(input,results);
                        cout << "Searched for " << input << endl;
                        cout << " found " << results.size() << " occurrences: "
<< endl;
                        if (results.size() > 0) {
                              cout << "{";
```

```
                                        for (set<string>::iterator it = results.begin();
it != results.end(); it++) {
                                                cout << *it << ", ";
                                        }
                                        cout << "}" << endl;
                                }
                        }
                }


                return 0;
        }
}





// Recursively process a directory.
// Find all files in all subdirectories that have an extension in
valid_extensions.
// Runs processFile on each such file.
void processDirectory (const char *dname, vector<string> valid_extensions,
InvertedIndex &inverted_index, int& nfiles, int& ndirs) {
        DIR *dir = opendir(dname);

        if (dir != 0) {
                // Iterate over each entry in the directory.
                for (struct dirent *ent = readdir(dir); ent != 0; ent=readdir(dir))
{
                        // Check if it is a directory
                        if (ent->d_type == DT_DIR) {
                                // Make sure not the current (".") nor parent ("..")
directory.
                                if ((strcmp(ent->d_name,".") != 0) && (strcmp(ent-
>d_name,"..") != 0)) {
                                        // If so, do a recursive call to process that
directory.
                                        string fulldname(dname);
                                        fulldname += "/";
                                        fulldname += ent->d_name;
                                        cout << "Found directory: " << fulldname << endl;
                                        ndirs++;

                                        processDirectory(fulldname.c_str(),
valid_extensions, inverted_index, nfiles, ndirs);
                                }
                        } else {
                                // Check to see if this file ends in a valid extension.
                                // strrchr returns last occurrence of character, null if
not found.
                                char *extension = strrchr(ent->d_name,'.');
                                // Compare to the set of valid extensions.
                if (validExtension(extension,valid_extensions)) {
                    string fulldname(dname);
                    fulldname += "/";
                    fulldname += ent->d_name;
                    cout << "Found a file: " << fulldname << endl;
                    nfiles++;
                    processFile(fulldname.c_str(), inverted_index);
                }
            }
        }

    }
}


// Checks whether extension is contained in the vector of valid extensions.
// Returns true if so, false if not.
// If valid_extensions is empty, everything is valid.
bool validExtension (char *extension, vector<string> valid_extensions) {
    if (valid_extensions.size()==0) {
        return true;
    } else {
        if (extension != 0) {
            for (int i=0; i < (int) valid_extensions.size(); i++) {
                // Compare extension to this valid extension.
                if (strcmp(extension, valid_extensions[i].c_str()) == 0)
{
                    return true;
                }
            }
        }
        return false;
    }
}


// Read every word in this file.
// Insert a pair into the hash table (word,fname)
void processFile (const char *fname, InvertedIndex& inverted_index) {

    ifstream inputfile;  // ifstream for reading from input file.
    inputfile.open (fname);
    string fnames(fname); // file name as a string object, not as a char * (c-
style string, which is an array of characters with \0 at the end).

    // Tokenize the input.
    // Read one character at a time.
    // If the character is not in a-z or A-Z, terminate current string.
    char c;
    char curr_str[MAX_STRING_LEN];
    int str_i = 0;  // Index into curr_str.
    bool flush_it = false;  // Whether we have a complete string to flush.
```

```
		while (inputfile.good()) {
			// Read one character, convert it to lowercase.
			inputfile.get(c);
			c = tolower(c);

			if (c >= 'a' && c <= 'z') {
				// c is a letter.
				curr_str[str_i] = c;
				str_i++;

				// Check over-length string.
				if (str_i >= MAX_STRING_LEN) {
					flush_it = true;
				}
			} else {
				// c is not a letter.
				// Create a new string if curr_str is non-empty.
				if (str_i>0) {
					flush_it = true;
				}
			}

			if (flush_it) {
				// Create the new string from curr_str.
				string the_str(curr_str,str_i);
				// cout << the_str << endl;


				// Insert the string-file_name tuple into the inverted index.
				inverted_index.add(the_str,fnames);

				// cout << "Add " << the_str << "," << fname << endl;


				// Reset state variables.
				str_i = 0;
				flush_it = false;
			}
		}
	}
```