ASSIGNMENT 2 – CMPT 225

MINH PHAT TRAN – 301 297 286

Question 1:

Test case 1: Array is filled with random numbers between 0 and RAND_MAX (use rand() function.

Test case 2: An already sorted array.

Test case 3: An array that is reversely sorted.

Test case 4: Array is filled with random number between 0 to 5 (use rand()%6).

Modified Bubble Sort:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
using namespace std;

void modifiedBubbleSort(int A[], int size)
{
	for (int i = 0; i < size - 1; i++)
	{
		int swapping = 0;
		for (int j = 0; j < size - i -1; j++)
		{
			if(A[j] > A[j+1])
			{
				int temp = A[j];
				A[j] = A[j+1];
				A[j+1] = temp;
				swapping = swapping + 1;
			}
		}
		if(swapping == 0)
		{
			break;
		}
	}
}

int main()
{
	clock_t start1, end1;
	srand((unsigned)time(0));
	int Array1[2000], Array2[2000], Array3[2000], Array4[2000];
	for (int i = 0; i < 2000; i++)
	{
		Array1[i] = 0 + (rand() % (RAND_MAX + 1 - 0));
	}
	start1 = clock();
	modifiedBubbleSort(Array1, 2000);
```

```
	end1 = clock();
	double duration1 = double(end1 - start1)/double(CLOCKS_PER_SEC);
    cout << "Time taken by modified bubble sort for array 1 : " << fixed << duration1 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array2[i] = i;
	}
	clock_t start2, end2;
	start2 = clock();
	modifiedBubbleSort(Array2, 2000);
	end2 = clock();
	double duration2 = double(end2 - start2)/double(CLOCKS_PER_SEC);
    cout << "Time taken by modified bubble sort for array 2 : " << fixed << duration2 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array3[i] = 2000-i;
	}
	clock_t start3, end3;
	start3 = clock();
	modifiedBubbleSort(Array3, 2000);
	end3 = clock();
	double duration3 = double(end3 - start3)/double(CLOCKS_PER_SEC);
    cout << "Time taken by modified bubble sort for array 3 : " << fixed << duration3 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array4[i] = rand()%6;
	}
	clock_t start4, end4;
	start4 = clock();
	modifiedBubbleSort(Array4, 2000);
	end4 = clock();
	double duration4 = double(end4 - start4)/double(CLOCKS_PER_SEC);
    cout << "Time taken by modified bubble sort for array 4 : " << fixed << duration4 <<
setprecision(5) << " sec " << endl;

	system("pause");
	return 0;
}
```

Selection Sort:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
using namespace std;

void selectionSort(int A[], int size)
{
	int min;
```

```
	for (int i = 0; i < size-1; i++)
	{
		min = i;
		for (int j = i+1; j < size; j++)
		{
			if(A[min] > A[j])
			{
				int temp = A[min];
				A[min] = A[j];
				A[j] = temp;
			}
		}
	}
}
int main()
{
	clock_t start1, end1;
	srand((unsigned)time(0));
	int Array1[2000], Array2[2000], Array3[2000], Array4[2000];
	for (int i = 0; i < 2000; i++)
	{
		Array1[i] = 0 + (rand() % (RAND_MAX + 1 - 0));
	}
	start1 = clock();
	selectionSort(Array1, 2000);
	end1 = clock();
	double duration1 = double(end1 - start1)/double(CLOCKS_PER_SEC);
    cout << "Time taken by selection sort for array 1 : " << fixed << duration1 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array2[i] = i;
	}
	clock_t start2, end2;
	start2 = clock();
	selectionSort(Array2, 2000);
	end2 = clock();
	double duration2 = double(end2 - start2)/double(CLOCKS_PER_SEC);
    cout << "Time taken by selection sort for array 2 : " << fixed << duration2 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array3[i] = 2000-i;
	}
	clock_t start3, end3;
	start3 = clock();
	selectionSort(Array3, 2000);
	end3 = clock();
	double duration3 = double(end3 - start3)/double(CLOCKS_PER_SEC);
    cout << "Time taken by selection sort for array 3 : " << fixed << duration3 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array4[i] = rand()%6;
	}
```

```
	clock_t start4, end4;
	start4 = clock();
	selectionSort(Array4, 2000);
	end4 = clock();
	double duration4 = double(end4 - start4)/double(CLOCKS_PER_SEC);
    cout << "Time taken by selection sort for array 4 : " << fixed << duration4 <<
setprecision(5) << " sec " << endl;

	system("pause");
	return 0;
}
```

Insertion Sort:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
using namespace std;

void insertionSort(int A[], int size)
{
	for (int i = 1; i < size; i++)
	{
		int insert = A[i];
		int move = i;
		while((move > 0) && (A[move - 1] > insert))
		{
			A[move] = A[move-1];
			move = move - 1;
		}
		A[move] = insert;

	}
}

int main()
{
	clock_t start1, end1;
	srand((unsigned)time(0));
	int Array1[2000], Array2[2000], Array3[2000], Array4[2000];
	for (int i = 0; i < 2000; i++)
	{
		Array1[i] = 0 + (rand() % (RAND_MAX + 1 - 0));
	}
	start1 = clock();
	insertionSort(Array1, 2000);
	end1 = clock();
	double duration1 = double(end1 - start1)/double(CLOCKS_PER_SEC);
    cout << "Time taken by insertion sort for array 1 : " << fixed << duration1 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array2[i] = i;
	}
	clock_t start2, end2;
```

```
	start2 = clock();
	insertionSort(Array2, 2000);
	end2 = clock();
	double duration2 = double(end2 - start2)/double(CLOCKS_PER_SEC);
    cout << "Time taken by insertion sort for array 2 : " << fixed << duration2 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array3[i] = 2000-i;
	}
	clock_t start3, end3;
	start3 = clock();
	insertionSort(Array3, 2000);
	end3 = clock();
	double duration3 = double(end3 - start3)/double(CLOCKS_PER_SEC);
    cout << "Time taken by insertion sort for array 3 : " << fixed << duration3 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array4[i] = rand()%6;
	}
	clock_t start4, end4;
	start4 = clock();
	insertionSort(Array4, 2000);
	end4 = clock();
	double duration4 = double(end4 - start4)/double(CLOCKS_PER_SEC);
    cout << "Time taken by insertion sort for array 4 : " << fixed << duration4 <<
setprecision(5) << " sec " << endl;

	system("pause");
	return 0;
}
```

Merge Sort:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
using namespace std;

void merge(int A[], int start, int end)
{
	int size = (end - start) + 1;
	int* temp = new int[size];
	int i = start;
	int mid = (start + end)/2;
	int k = 0;
	int j = mid + 1;
	while(k < size)
	{
		if(i <= mid && j <= end)
		{
			if(A[i] <= A[j])
```

```
                        temp[k] = A[i++];
                else
                        temp[k] = A[j++];
        }
        else if (i <= mid)
                temp[k] = A[i++];
        else if (j <= end)
                temp[k] = A[j++];
        k = k + 1;
    }

    for (k = 0; k < size; k++)
    {
        A[start + k] = temp[k];
    }
    delete [] temp;
}
void mergeSort(int iA[], int start, int end)
{
    if (start >= end)
        return;
    int mid = (start + end)/2;
    mergeSort(iA, start, mid);
    mergeSort(iA, mid + 1, end);
    merge(iA, start, end);
}

int main()
{
    clock_t start1, end1;
    srand((unsigned)time(0));
    int Array1[2000], Array2[2000], Array3[2000], Array4[2000];
    for (int i = 0; i < 2000; i++)
    {
        Array1[i] = 0 + (rand() % (RAND_MAX + 1 - 0));
    }
    start1 = clock();
    mergeSort(Array1, 0, 1999);
    end1 = clock();
    double duration1 = double(end1 - start1)/double(CLOCKS_PER_SEC);
  cout << "Time taken by merge sort for array 1 : " << fixed << duration1 <<
setprecision(5) << " sec " << endl;

    for (int i = 0; i < 2000; i++)
    {
        Array2[i] = i;
    }
    clock_t start2, end2;
    start2 = clock();
    mergeSort(Array2, 0, 1999);
    end2 = clock();
    double duration2 = double(end2 - start2)/double(CLOCKS_PER_SEC);
  cout << "Time taken by merge sort for array 2 : " << fixed << duration2 <<
setprecision(5) << " sec " << endl;

    for (int i = 0; i < 2000; i++)
    {
        Array3[i] = 2000-i;
```

```
	}
	clock_t start3, end3;
	start3 = clock();
	mergeSort(Array3, 0, 1999);
	end3 = clock();
	double duration3 = double(end3 - start3)/double(CLOCKS_PER_SEC);
    cout << "Time taken by merge sort for array 3 : " << fixed << duration3 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array4[i] = rand()%6;
	}
	clock_t start4, end4;
	start4 = clock();
	mergeSort(Array4, 0, 1999);
	end4 = clock();
	double duration4 = double(end4 - start4)/double(CLOCKS_PER_SEC);
    cout << "Time taken by merge sort for array 4 : " << fixed << duration4 <<
setprecision(5) << " sec " << endl;
	system("pause");
	return 0;
}
```

Quick Sort:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
using namespace std;

void swap(int* a, int* b)
{
	int temp = *a;
	*a = *b;
	*b = temp;
}

int partition(int a[], int first, int last)
{
	srand((unsigned)time(0));
	int random = first + (rand() % (last + 1 - first));
	swap(&a[last], &a[random]);
	int pivotElement = a[last];
	int i = first - 1;
	for (int  j = first; j <= last - 1; j++)
	{
		if(a[j] <= pivotElement)
		{
			i++;
			swap(&a[i], &a[j]);
		}
	}
	swap(&a[i+1], &a[last]);
	return (i+1);
}
```

```
void quickSort(int A[], int low, int high)
{
	if(low < high)
	{
		int pi = partition(A, low, high);
		quickSort(A, low, pi - 1);
		quickSort(A, pi + 1, high);
	}
}

int main()
{
	clock_t start1, end1;
	srand((unsigned)time(0));
	int Array1[2000], Array2[2000], Array3[2000], Array4[2000];

	for (int i = 0; i < 2000; i++)
	{
		Array1[i] = 0 + (rand() % (RAND_MAX + 1 - 0));
	}
	start1 = clock();
	quickSort(Array1, 0, 1999);
	end1 = clock();
	double duration1 = double(end1 - start1)/double(CLOCKS_PER_SEC);
    cout << "Time taken by quick sort for array 1 : " << fixed << duration1 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array2[i] = i;
	}
	clock_t start2, end2;
	start2 = clock();
	quickSort(Array2, 0, 1999);
	end2 = clock();
	double duration2 = double(end2 - start2)/double(CLOCKS_PER_SEC);
    cout << "Time taken by quick sort for array 2 : " << fixed << duration2 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array3[i] = 2000-i;
	}
	clock_t start3, end3;
	start3 = clock();
	quickSort(Array3, 0, 1999);
	end3 = clock();
	double duration3 = double(end3 - start3)/double(CLOCKS_PER_SEC);
    cout << "Time taken by quick sort for array 3 : " << fixed << duration3 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array4[i] = rand()%6;
	}
	clock_t start4, end4;
	start4 = clock();
```

```
	quickSort(Array4, 0, 1999);
	end4 = clock();
	double duration4 = double(end4 - start4);
    cout << "Time taken by quick sort for array 4 : " << fixed << duration4 <<
setprecision(10) << " sec " << endl;

	system("pause");
	return 0;
}
```

Heap Sort:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
using namespace std;

void swap(int* a, int* b)
{
	int temp = *a;
	*a = *b;
	*b = temp;
}


void heapify(int A[], int size, int i)
{
	int largest = i;
	int left = 2*i + 1;
	int right = 2*i + 2;

	if(left < size && A[left] > A[largest])
	{
		largest = left;
	}

	if(right < size && A[right] > A[largest])
	{
		largest = right;
	}

	if(largest != i)
	{
		swap(&A[i], &A[largest]);
		heapify(A, size, largest);
	}
}

void heapSort(int A[], int size)
{
	for (int i = size/2 - 1; i >= 0; i--)
	{
		heapify(A,size,i); // max heap
	}
```

```
	for (int i = size-1; i > 0; i--)
	{
		swap(&A[0], &A[i]);
		heapify(A, i, 0);
	}
}

int main()
{
	clock_t start1, end1;
	srand((unsigned)time(0));
	int Array1[2000], Array2[2000], Array3[2000], Array4[2000];
	for (int i = 0; i < 2000; i++)
	{
		Array1[i] = 0 + (rand() % (RAND_MAX + 1 - 0));
	}
	start1 = clock();
	heapSort(Array1, 2000);
	end1 = clock();
	double duration1 = double(end1 - start1)/double(CLOCKS_PER_SEC);
    cout << "Time taken by heap sort for array 1 : " << fixed << duration1 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array2[i] = i;
	}
	clock_t start2, end2;
	start2 = clock();
	heapSort(Array2, 2000);
	end2 = clock();
	double duration2 = double(end2 - start2)/double(CLOCKS_PER_SEC);
    cout << "Time taken by heap sort for array 2 : " << fixed << duration2 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array3[i] = 2000-i;
	}
	clock_t start3, end3;
	start3 = clock();
	heapSort(Array3, 2000);
	end3 = clock();
	double duration3 = double(end3 - start3)/double(CLOCKS_PER_SEC);
    cout << "Time taken by heap sort for array 3 : " << fixed << duration3 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array4[i] = rand()%6;
	}
	clock_t start4, end4;
	start4 = clock();
	heapSort(Array4, 2000);
	end4 = clock();
	double duration4 = double(end4 - start4)/double(CLOCKS_PER_SEC);
    cout << "Time taken by heap sort for array 4 : " << fixed << duration4 <<
setprecision(5) << " sec " << endl;
```

```
	system("pause");
	return 0;
}
```

Balanced Binary Search Tree:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <queue>
#include <string>
using namespace std;

#include <iostream>
#include <string>
using namespace std;
struct Node
{
	int data;
	Node* left;
	Node* right;
	int height;
};
class AVL
{
private:
	Node* root;
	Node* insert(int data, Node* root);
	Node* remove(int data, Node* root);
	int getBalance(Node* N);
	int max(int a, int b);
	void DestroyRecursive(Node* node);
	Node* singleRightRotate(Node*&y);
	Node* singleLeftRotate(Node*&x);
	Node* doubleRightLeftRotate(Node*& node);
	Node* doubleLeftRightRotate(Node*& node);
	Node* copyTree_helper(const Node* source);
public:
	Node* getRoot();
	int height(Node* N);
	Node* newNode(int data);
	bool isComplete(Node* root);
	Node* findMin(Node* node);
	Node* findMax(Node* node);
	Node* parent(Node* root, int data);
	int childCount(Node* root, int data);
	bool search(Node* root, int data);
	int originalHeight();
	bool isRoot(Node* root, int data);
	bool isInternal(Node* root, int data);
	bool isExternal(Node* root, int data);
	int size(Node* root);
	void inOrder(Node* root);
	void postOrder(Node* root);
```

```
        void preOrder(Node* root);
        AVL();
        ~AVL();
        AVL(const AVL& rhs);
        const AVL& operator=(const AVL& rhs);
        bool isFull(Node* root);
        bool isAVL(Node* root);
        void insert(int data);
        void remove(int data);
};
//Pre: Nothing is changed.
//Post: This function will return the address which is pointed by pointer root.
Node* AVL::getRoot()
{
        return root;
}
//Pre: Nothing is changed.
//Post: This function will return the height of AVL tree.
int AVL::height(Node* N)//In lecture notes, the height of empty tree is -1, but the
sample code in AVL lecture is 0 ???
{
        if(N == nullptr)
        {
                return -1;
        }
        return N->height;
}
//Pre: Nothing is changed.
//Post: This function will return the max value between 2 variables.
int AVL::max(int a, int b)
{
        if(a > b)
        {
                return a;
        }
        else
        {
                return b;
        }
}
//Pre: Nothing is changed.
//Post: This function will return the balance factor of one node in the AVL tree.
int AVL::getBalance(Node* N)
{
        if(N == nullptr)
        {
                return 0;
        }
        return (height(N->left) - height(N->right));
}
//Pre: Nothing is changed.
//Post: This function will create the new node which will be inserted into AVL tree.
Node*AVL::newNode(int data)
{
        Node* node = new Node;
        node->data = data;
        node->height = 0;
        node->left = nullptr;
```

```
	node->right = nullptr;
	return node;
}
//Pre: Nothing is changed.
//Post: This function will insert a new node into AVL tree.
Node*AVL::insert(int data, Node* root)
{
	//Step 1: Perform the normal BST insertion
	if(root == nullptr)
	{
		return newNode(data);
	}
	if(data < root->data)
	{
		root->left = insert(data, root->left);
	}
	else if(data > root->data)
	{
		root->right = insert(data, root->right);
	}
	else
	{
		root->left = insert(data, root->left);
	}
	//Step 2: Update height of this ancestor node
	root->height = 1 + max(height(root->left), height(root->right));
	//Step 3: Get the balance factor of this ancestor node to check whether this node
became unbalanced
	int balance = getBalance(root);
	//Left Left Case, then single right rotate
	if(balance > 1 && data <= root->left->data)
	{

		return singleRightRotate(root);
	}
	//Right Right Case, then single left rotate
	if(balance < -1 && data >= root->right->data)
	{
		return singleLeftRotate(root);
	}
	//Left Right Case, then single left rotate then single right rotate
	if(balance > 1 && data >= root->left->data)
	{

		root->left = singleLeftRotate(root->left);
		return singleRightRotate(root);
	}
	//Right Left Case, then single right rotate then single left rotate
	if (balance < -1 && data <= root->right->data)
	{

		root->right = singleRightRotate(root->right);
		return singleLeftRotate(root);
	}
	return root;
}
//Pre: Nothing is changed.
//Post: This function will do single right rotate to make sure AVL tree is balanced.
```

```
Node* AVL::singleRightRotate(Node*&y)
{
	Node* x = y->left;
	Node* T2 = x->right;
	//Rotate process
	x->right = y;
	y->left = T2;
	//Update new heights
	y->height = max(height(y->left), height(y->right)) + 1;
	x->height = max(height(x->left), height(x->right)) + 1;
	return x;
}
//Pre: Nothing is changed.
//Post: This function will do single left rotate to make sure AVL tree is balanced.
Node*AVL::singleLeftRotate(Node*&x)
{
	Node* y = x->right;
	Node* T2 = y->left;
	//Rotate process
	y->left = x;
	x->right = T2;
	//Update new heights
	y->height = max(height(y->left), height(y->right)) + 1;
	x->height = max(height(x->left), height(x->right)) + 1;
	return y;
}
//Pre: The balance factor is not between -1 and 1.
//Post : The balance factor will be between -1 and 1.
Node* AVL::doubleRightLeftRotate(Node*& node)//Similar Right Left Case
{
	node->right = singleRightRotate(node->right);
	return singleLeftRotate(node);
}
//Pre: The balance factor is not between -1 and 1.
//Post : The balance factor will be between -1 and 1.
Node* AVL::doubleLeftRightRotate(Node*& node)//Similar Left Right Case
{
	node->left = singleLeftRotate(node->left);
	return singleRightRotate(node);
}
//Pre: Nothing is changed.
//Post: This function will find the minimum node in the AVL tree.
Node* AVL::findMin(Node* node)
{
	Node* current = node;
	while(current->left != nullptr)
	{
		current = current->left;
	}
	return current;
}
//Pre: Nothing is changed.
//Post: This function will find the maximum in the AVL tree.
Node*AVL::findMax(Node* node)
{
	Node* current = node;
	while(current->right != nullptr)
	{
```

```
                current = current->right;
        }
        return current;
}
//Pre: Nothing is changed.
//Post: This function will remove a node from AVL tree.
Node*AVL::remove(int data, Node* root)
{
        //Step 1: Perform standard BST delete
        if(root == nullptr)
        {
                return root;
        }
        if(data <= root->data)
        {
                root->left = remove(data, root->left);
        }
        else if(data >= root->data)
        {
                root->right = remove(data, root->right);
        }
        else
        {
                if((root->left == nullptr) || (root->right == nullptr))//Case has one child
or no child
                {
                        Node* temp = root->left? root->left:root->right; //root->right will
be assigned to temp if root->left is nullptr
                        if(temp == nullptr) //Case has no child
                        {
                                temp = root;
                                root = nullptr;
                        }
                        else //Case has one child, we have to copy contents of non-empty
child to the root node and delete the pointer of child node
                        {
                                *root = *temp;
                        }
                        delete temp;
                }
                else//Case has two children
                {
                        Node* temp = findMin(root->right);
                        root->data = temp->data; //Copy the inorder successor's data to this
node
                        root->right = remove(temp->data,root->right);
                }
        }
        if(root == nullptr)
        {
                return root;
        }
        //Step 2: Update height of this ancestor node
        root->height = 1 + max(height(root->left), height(root->right));
        //Step 3: Get the balance factor of this ancestor noe to check whether this node
became unbalanced
        int balance = getBalance(root);
        //Left Left Case, then single right rotate
```

```
	if(balance > 1 && data <= root->left->data)
	{
		return singleRightRotate(root);
	}
	//Right Right Case, then single left rotate
	if(balance < -1 && data >= root->right->data)
	{
		return singleLeftRotate(root);
	}
	//Left Right Case, then single left rotate then single right rotate
	if(balance > 1 && data >= root->left->data)
	{
		//root->left = singleLeftRotate(root->left);
		return doubleLeftRightRotate(root);
	}
	//Right Left Case, then single right rotate then single left rotate
	if (balance < -1 && data <= root->right->data)
	{
		//root->right = singleRightRotate(root->right);
		return doubleRightLeftRotate(root);
	}
	return root;
}
//Pre: Nothing is changed.
//Post: This function will return address of a node's parent.
Node*AVL::parent(Node* root, int data)//I assume that this function will find the parent
of node which has the given data.
{
	if(root == nullptr)
	{
		return root;
	}
	else if(root->right == nullptr && root->left == nullptr)//root->height == 0 does
not work ???!!!
	{
		return nullptr;
	}
	else if(root->left != nullptr && root->left->data == data)
	{
		return root;
	}
	else if(root->right != nullptr && root->right->data == data)
	{
		return root;
	}
	else if(data < root->data)
	{
		return parent(root->left,data);
	}
	else if(data > root->data)
	{
		return parent(root->right,data);
	}
	else
	{
		cout << "Something wrong happen ! You are trying to find parent of root
node or given data does not exist in this AVL tree !" << endl;
	}
```

```
}
//Pre: Nothing is changed.
//Post: This function will return the number of children of a node.
int AVL::childCount(Node* root, int data)//I assume that this function will count the
number of children of node which has the given data.
{
	if(root == nullptr)
	{
		return -1;
	}
	else if(root->data == data && root->left == nullptr && root->right == nullptr)
	{
		return 0;
	}
	else if(root->data == data && root->left == nullptr && root->right != nullptr)
	{
		return 1;
	}
	else if(root->data == data && root->left != nullptr && root->right == nullptr)
	{
		return 1;
	}
	else if(root->data == data && root->left != nullptr && root->right != nullptr)
	{
		return 2;
	}
	else if(data < root->data)
	{
		return childCount(root->left,data);
	}
	else if(data > root->data)
	{
		return childCount(root->right,data);
	}
	else
	{
		cout << "Something wrong happen ! Given data does not exist in this AVL
tree !" << endl;
	}
}
//Pre: Nothing is changed.
//Post: This funtion will find the node has the same data with parameter.
bool AVL::search(Node* root, int data)
{
	if(root == nullptr)
	{
		return false;
	}
	else if(root->data == data)
	{
		return true;
	}
	else if(data < root->data)
	{
		return search(root->left, data);
	}
	else
	{
```

```
                return search(root->right, data);
        }
}
//Pre: Nothing is changed.
//Post: This function will return the height of tree and this function does not have any
parameters. This is useful for isRoot function.
int AVL::originalHeight()
{
        return height(root);
}
//Pre: Nothing is changed.
//Post: This function will check a node whether it is a root or not.
bool AVL::isRoot(Node* root, int data)//Firstly, I will search the node has the given
data, then I will check its height to know that it is root node or not
{
        if(root == nullptr)
        {
                return false;
        }
        else if(root->data == data && root->height == originalHeight())
        {
                return true;
        }
        else if(root->data == data && root->height != originalHeight())
        {
                return false;
        }
        else if(data < root->data)
        {
                return isRoot(root->left, data);
        }
        else if(data > root->data)
        {
                return isRoot(root->right, data);
        }
}

//Pre: Nothing is changed.
//Post: This function will check a node is internal or not.
bool AVL::isInternal(Node* root, int data)
{
        if(root == nullptr)
        {
                return false;
        }
        else if(isRoot(root, data) == true)
        {
                return false;
        }
        else if(root->data == data && root->left == nullptr && root->right == nullptr)
        {
                return false;
        }
        else if(root->data == data && root->left == nullptr && root->right != nullptr)
        {
                return true;
        }
        else if(root->data == data && root->left != nullptr && root->right == nullptr)
```

```
	{
		return true;
	}
	else if(root->data == data && root->left != nullptr && root->right != nullptr)
	{
		return true;
	}
	else if(data < root->data)
	{
		return isInternal(root->left,data);
	}
	else if(data > root->data)
	{
		return isInternal(root->right,data);
	}
	else
	{
		return false;
	}
}
//Pre: Nothing is changed.
//Post: This function will check a node is external or not.
bool AVL::isExternal(Node* root, int data)
{
	if(root == nullptr)
	{
		return false;
	}
	else if(isRoot(root, data) == true)
	{
		return false;
	}
	else if(root->data == data && root->left == nullptr && root->right == nullptr)
	{
		return true;
	}
	else if(root->data == data && root->left == nullptr && root->right != nullptr)
	{
		return false;
	}
	else if(root->data == data && root->left != nullptr && root->right == nullptr)
	{
		return false;
	}
	else if(root->data == data && root->left != nullptr && root->right != nullptr)
	{
		return false;
	}
	else if(data < root->data)
	{
		return isExternal(root->left,data);
	}
	else if(data > root->data)
	{
		return isExternal(root->right,data);
	}
	else
	{
```

```
                return false;
        }
}
//Pre: Nothing is changed.
//Post: This function will return the size of AVL tree.
int AVL::size(Node* root)
{
        if(root == nullptr)
        {
                return 0;
        }
        else if(root != nullptr)
        {
                return (1 + size(root->left) + size(root->right));
        }
}
//Pre: Nothing is changed.
//Post: All nodes will be printed of order Left, Root, Right.
void AVL::inOrder(Node* root)
{
        if(root != nullptr)
        {
                inOrder(root->left);
                //cout << root->data << " " << endl;
                inOrder(root->right);
        }
        return;
}
//Pre: Nothing is changed.
//Post: All nodes will be printed of order Left, Right, Root.
void AVL::postOrder(Node* root)
{
        if(root != nullptr)
        {
                postOrder(root->left);
                postOrder(root->right);
                cout << root->data << " " << endl;
        }
        return;
}
//Pre: Nothing is changed.
//Post: All nodes will be printed of order Root, Left, Right.
void AVL::preOrder(Node* root)
{
        if(root != nullptr)
        {
                cout << root->data << " " << endl;
                preOrder(root->left);
                preOrder(root->right);
        }
        return;
}
//Pre: Nothing is changed.
//Post: We can initialize a new AVL tree with root = nullptr.
AVL::AVL()
{
        root = nullptr;
}
```

```
//Pre: Dynamic variables still exist in memory.
//Post: Dynamic variables are deallocated in memory.
void AVL::DestroyRecursive(Node* node)
{
	if(node)
	{
		DestroyRecursive(node->left);
		DestroyRecursive(node->right);
		delete node;
	}
}
//Pre: Dynamic variables still exist in memory.
//Post: Dynamic variables are deallocated in memory.
AVL::~AVL()
{
	DestroyRecursive(root);
}
//Pre: Nothing is changed.
//Post: Copy all data of source object to the this object.
Node* AVL::copyTree_helper(const Node* source)
{
	if(source == nullptr)
	{
		return nullptr;
	}
	Node* result = new Node;
	result->data = source->data;
	result->left = copyTree_helper(source->left);
	result->right = copyTree_helper(source->right);
	return result;
}
//Pre: We cannot assign the values of old AVL tree to the new AVL tree.
//Post: We can assign the values of old AVL tree to the new AVL tree.
AVL::AVL(const AVL& rhs)
{
	root = copyTree_helper(rhs.root);
}
//Pre: We cannot assign the values of old AVL tree to the new AVL tree.
//Post: We can assign the values of old AVL tree to the new AVL tree.
const AVL& AVL::operator=(const AVL& rhs)
{
	if(this != &rhs)
	{
		if(root != nullptr)
		{
			DestroyRecursive(root);
		}
		if(rhs.root == nullptr)
		{
			root = nullptr;
		}
		else
		{
			root = copyTree_helper(rhs.root);
		}
	}
	return *this;
}
```

```
//Pre: Nothing is changed.
//Post: This function will check a AVL tree is full or not.
bool AVL::isFull(Node* root)
{
	if(root == nullptr)
	{
		return true;
	}
	else if(root->left == nullptr && root->right == nullptr)
	{
		return true;
	}
	else if(root->left != nullptr && root->right != nullptr)
	{
		return (isFull(root->left) && isFull(root->right));
	}
	else
	{
		return false;
	}
}

bool AVL::isComplete(Node* root)
{
    if (root == NULL)
       {
        return true;
       }
    queue<Node*>queue;
    queue.push(root);
    bool flag = false;
    while(queue.empty() == false)
    {
        Node* temp =queue.front();
        queue.pop();
        if(temp->left != nullptr)
        {
                    if (flag == true)
                    {
                        return false;
                    }
                    queue.push(temp->left);
        }
        else if(temp->left == nullptr)
              {
                    flag = true;
              }
        if(temp->right != nullptr)
        {
                    if(flag == true)
                    {
                        return false;
                    }
                    queue.push(temp->right);
        }
        else if(temp->right == nullptr)
              {
                    flag = true;
```

```
                }
        }
        return true;
}

//Pre: Nothing is changed.
//Post: This function will check a AVL tree is AVL or not. Besides, after you inserting,
you have already done rotates to make tree balance. So, isAVL will always show the result
of true.
bool AVL::isAVL(Node* root)
{
        /*if(root == nullptr)
        {
                return true;
        }
        int left = height(root->left);
        int right = height(root->right);
        if(abs(left-right) <= 1 && isAVL(root->left) == true && isAVL(root->right) ==
true)
        {
                return true;
        }
        return false;*/
        if(root == nullptr)
        {
                return true;
        }
        else if(root != nullptr)
        {
                isAVL(root->left);
                if(abs(getBalance(root)) > 1)
                {
                        return false;
                }
                isAVL(root->right);
        }
        return true;
}
//Pre: Nothing is change.
//Post: We can call insert function without root parameter.
void AVL::insert(int data)
{
        root = insert(data, root);
}
//Pre: Nothing is change.
//Post: We can call remove function without root parameter.
void AVL::remove(int data)
{
        root = remove(data, root);
}

int main()
{
        clock_t start1, end1;
        srand((unsigned)time(0));
        int Array1[10], Array2[10], Array3[10], Array4[10];
        for (int i = 0; i < 10; i++)
        {
```

```
		Array1[i] = 0 + (rand() % (RAND_MAX + 1 - 0));
	}
	AVL tree1;
	start1 = clock();
	for (int i = 0; i < 10; i++)
	{
		tree1.insert(Array1[i]);
	}
	tree1.inOrder(tree1.getRoot());
	end1 = clock();
	double duration1 = double(end1 - start1)/double(CLOCKS_PER_SEC);
    cout << "Time taken by BST for array 1 : " << fixed << duration1 << setprecision(5)
<< " sec " << endl;

	clock_t start2, end2;
	for (int i = 0; i < 10; i++)
	{
		Array2[i] = i;
	}
	AVL tree2;
	start2 = clock();
	for (int i = 0; i < 10; i++)
	{
		tree2.insert(Array2[i]);
	}
	tree2.inOrder(tree2.getRoot());
	end2 = clock();
	double duration2 = double(end2 - start2)/double(CLOCKS_PER_SEC);
    cout << "Time taken by BST for array 2 : " << fixed << duration2 << setprecision(5)
<< " sec " << endl;

	clock_t start3, end3;
	for (int i = 0; i < 10; i++)
	{
		Array3[i] = 10-i;
	}
	AVL tree3;
	start3 = clock();
	for (int i = 0; i < 10; i++)
	{
		tree3.insert(Array3[i]);
	}
	tree3.inOrder(tree3.getRoot());
	end3 = clock();
	double duration3 = double(end3 - start3)/double(CLOCKS_PER_SEC);
    cout << "Time taken by BST for array 3 : " << fixed << duration3 << setprecision(5)
<< " sec " << endl;

	clock_t start4, end4;
	for (int i = 0; i < 10; i++)
	{
		Array4[i] = rand()%6;
	}
	AVL tree4;
	start4 = clock();
	for (int i = 0; i < 10; i++)
	{
		tree4.insert(Array4[i]);
```

```
        }
        tree4.inOrder(tree4.getRoot());
        end4 = clock();
        double duration4 = double(end4 - start4)/double(CLOCKS_PER_SEC);
    cout << "Time taken by BST for array 4 : " << fixed << duration4 << setprecision(5)
<< " sec " << endl;

        system("pause");
        return 0;
}
```

Radix Sort:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
using namespace std;
void countSort(int A[], int size, int base)
{
        int * output = new int[size];
        int count[10];
        for (int i = 0; i < 10; i++)
        {
                count[i] = 0;
        }

        for(int i = 0; i < size; i++)
        {
                count[(A[i]/base)%10]++;
        }

        for(int i = 1; i <= 9; i++)
        {
                count[i] += count[i-1];
        }

        for (int i = size - 1; i >= 0; i--)
        {
                output[count[(A[i]/base)%10] - 1] = A[i];
                count[(A[i]/base)%10]--;
        }
        for(int i = 0; i < size; i++)
        {
                A[i] = output[i];
        }

}
void radixSort(int A[], int size)
{
        int max = A[0];
        for (int i = 0; i < size; i++)
        {
                if(A[i] > max)
                {
                        max = A[i];
                }
```

```
	}
	for(int base = 1; max/base > 0; base = base*10)
	{
		countSort(A,size,base);
	}
}
int main()
{
	clock_t start1, end1;
	srand((unsigned)time(0));
	int Array1[2000], Array2[2000], Array3[2000], Array4[2000];
	for (int i = 0; i < 2000; i++)
	{
		Array1[i] = 0 + (rand() % (RAND_MAX + 1 - 0));
	}
	start1 = clock();
	radixSort(Array1, 2000);
	end1 = clock();
	double duration1 = double(end1 - start1)/double(CLOCKS_PER_SEC);
    cout << "Time taken by radix sort for array 1 : " << fixed << duration1 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array2[i] = i;
	}
	clock_t start2, end2;
	start2 = clock();
	radixSort(Array2, 2000);
	end2 = clock();
	double duration2 = double(end2 - start2)/double(CLOCKS_PER_SEC);
    cout << "Time taken by radix sort for array 2 : " << fixed << duration2 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array3[i] = 2000-i;
	}
	clock_t start3, end3;
	start3 = clock();
	radixSort(Array3, 2000);
	end3 = clock();
	double duration3 = double(end3 - start3)/double(CLOCKS_PER_SEC);
    cout << "Time taken by radix sort for array 3 : " << fixed << duration3 <<
setprecision(5) << " sec " << endl;

	for (int i = 0; i < 2000; i++)
	{
		Array4[i] = rand()%6;
	}
	clock_t start4, end4;
	start4 = clock();
	radixSort(Array4, 2000);
	end4 = clock();
	double duration4 = double(end4 - start4)/double(CLOCKS_PER_SEC);
    cout << "Time taken by radix sort for array 4 : " << fixed << duration4 <<
setprecision(5) << " sec " << endl;
	system("pause");
```

```
        return 0;

}
```

Since my memory locations are not enough to handle the big size arrays, I changed size of arrays from 20000 to 2000 for Modified Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort and Radix Sort. I changed size of arrays from 20000 to 10 for Balanced Binary Search Tree because my algorithm for BST wastes too many memory locations.

	Test case 1	Test case 2	Test case 3	Test case 4
Modified Bubble Sort	0.007 s	0 s	0.009 s	0.007 s
Selection Sort	0.007 s	0.005 s	0.007 s	0.005 s
Insertion Sort	0.003 s	0 s	0.006 s	0.003 s
Merge Sort	0.005 s	0.002 s	0.002 s	0.003 s
Quick Sort	0.001 s	0 s	0 s	8 s
Heap Sort	0.001 s	0.001 s	0 s	0.001 s
Balanced Binary Search Tree	0 s	0 s	0 s	0 s
Radix Sort	0 s	0 s	0 s	0.001 s

Question 2:

heap.h:

```
/*****************************************************
  Heap.h -- Declarations for Basic Heap-of-Pair-of-Ints Class

  Stores pairs <element,priority> of ints.
  Supports O(log n) insertion, O(1) peeking at min priority and element
  with min priority, and O(log n) extraction of element with min priority.
*****************************************************/
#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.
   void display();
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);
};
//This will display all elements and their priorities in Heap
void Heap::display()
{
       for (int i = 0; i < size(); i++)
       {
              cout << "Element: " << A[i].element << endl;
              cout << "Priority: " << A[i].priority << endl;
       }
}
//This is constructor of Heap with default capacity of 10.
Heap::Heap()
{
   hCapacity = 10 ;
   A = new Pair[hCapacity];
```

```
   hSize = 0 ;
}
//This is constructor of Heap with default capacity of given c.
Heap::Heap(int c)
{ // New empty Heap with capacity c.
   // Complete this.
       hCapacity = c;
       A = new Pair[hCapacity];
       hSize = 0;
}

// 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)
{
       // 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
   // Complete this.
       hSize = s;
       hCapacity = s+c;
       A = new Pair[hCapacity];
       for (int i = 0; i < s; i++)
       {
              A[i].element = Elements[i];
              A[i].priority = Priorities[i];
       }
       heapify();
}

// New Heap with combined contents and of the two given heaps.
Heap::Heap( const Heap & Heap1, const Heap & Heap2, 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.
   hCapacity = Heap1.hSize + Heap2.hSize + c ;
   // Complete this.
   hSize = Heap1.hSize + Heap2.hSize;
   A = new Pair[hCapacity];
   for(int i = 0; i < Heap1.hSize; i++)
   {
          A[i].element = Heap1.A[i].element;
          A[i].priority = Heap1.A[i].priority;
   }

   for(int i = 0; i < Heap2.hSize; i++)
   {
          A[i + Heap1.hSize].element = Heap2.A[i].element;
          A[i + Heap1.hSize].priority = Heap2.A[i].priority;
   }
   heapify();
}

// Destructor
Heap::~Heap()
{
```

```
   delete[] A;
}

// Modifiers
void Heap::insert(int element, int priority)
{
   A[hSize].element = element;
   A[hSize].priority = priority;
   trickleUp(hSize);
   hSize ++;
}

// Repairs the heap ordering invariant after adding a new element.
// Initial call should be trickleUp(hSize-1).
void Heap::trickleUp(int i)
{// Repairs ordering invariant after adding leaf at A[i].
   // Complete this.
       int parent = (i-1)/2;
       if(i > 0 && A[parent].priority > A[i].priority)
       {
              Pair temp = A[i];
              A[i] = A[parent];
              A[parent] = temp;
              trickleUp(parent);
       }
}
//This will swap two elements.
void Heap::swap(int i, int j)
{
   Pair temp = A[i];
   A[i] = A[j];
   A[j] = temp ;
}

// Removes and returns the element with highest priority.
// (That is, the element associated with the minimum priority value.)
int Heap::extractMin()
{
   // Complete this.
       if(hSize == 0)
       {
              return -1;
       }
       if(hSize == 1)
       {
              hSize = hSize - 1;
              return A[0].element;
       }
       int rootPriority = A[0].priority;
       int rootElement = A[0].element;
       A[0].priority = A[hSize-1].priority;
       A[0].element = A[hSize-1].element;
    hSize--;
    trickleDown(0);
    return rootElement;
}

// 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)
{
   // Complete this.
       int Node = i;
       int Left = 2*i + 1;
       int Right = 2*i + 2;
       if(Left < size() && A[Left].priority < A[Node].priority)
       {
              Node = Left;
       }
       if(Right < size() && A[Right].priority < A[Node].priority)
       {
              Node = Right;
       }
       if(Node != i)
       {
              swap(i, Node);
              trickleDown(Node);
       }
}

// Turns A[0] .. A[size-1] into a heap.
void Heap::heapify()
{
   for( int i = (hSize/2)-1; i>=0 ; i-- )
   {
          trickleDown(i);
   }
}
```

test1.cpp:

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

void heapTest();

int main(){
      heapTest();
         system("pause");
      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 heapTest(){

      // 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();
      }

}
```

test2.cpp:

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

void heapTest();

int main(){
      heapTest();
         system("pause");
      return 0;
}

void heapTest(){

      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 ;
}
```

test3.cpp:

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

void heapTest();

int main(){
        heapTest();
```

```
        system("pause");
    return 0;
}

void heapTest(){

  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();
  }

}
```

test4.cpp:

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

void heapTest();

int main(){
      heapTest();
         system("pause");
      return 0;
}

void heapTest(){


// 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 ;

}
```

<u>Question 3</u>:

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?

The purpose of the "break" and if statement "if j < i" is to make sure that if there are 2 similar words in the array, we will ignore and not count the second appearance of that word in the array.

For example: tran minh phat tran. The first "tran" will have count of 2, the second "tran" will have count of 0.

If we remove "if j < i" and "break", we will not ignore and not count the second appearance of that word in the array.

For example: tran minh phat tran. The first "tran" will have count of 2, the second "tran" will have count of 2 as well.

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

The time complexity of the naive word frequency computation algorithm: O(w*w) = 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.

The time complexity of the naive document finding algorithm: 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?

The time complexity of *all* of their searches: O(d*w*u)

Part 0: Implementing the hash table

Because I am using Windows and Microsoft Visual Studio, I cannot test cpp files with py file without using Linux

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 smarterHash(string key);

};
```

HashTable.cpp:

```
// Implement HashTable methods.
#include "HashTable.h"
#include <math.h>
#include <cmath>
#include <string>
using namespace std;

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;
}


bool HashTable::remove(string key, int value)
{
	// TO DO:: Write this.
	int hval = hash(key);
	int pval = hval;
	for(int iter = 0; iter < tsize; iter++)
	{
		if(sentinels[pval] == curr_used && values[pval] == value)
		{
			sentinels[pval] = prev_used;
			nstored--;
			return true;
		}
		pval = probeFunction(hval,iter);
	}
	return false;
}


bool HashTable::lookup(string key, int& value)
{
	// TO DO:: Write this.
	int hval = hash(key);
	int pval = hval;
	for(int iter = 0; iter < tsize; iter++)
	{
		if(sentinels[pval] == curr_used && values[pval] == value)
		{
			values[pval] = value;
			return true;
		}
		pval = probeFunction(hval,iter);
	}
	return false;
}


bool HashTable::modify(string key, int value)
{
```

```
	// TO DO:: Write this.
	int hval = hash(key);
	int pval = hval;
	for(int iter = 0; iter < tsize; iter++)
	{
		if(values[pval] == value && sentinels[pval] == curr_used)
		{
			values[pval] = value;
			return true;
		}
		pval = probeFunction(hval,iter);
	}
	return false;
}


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++;
		}
	}
}

int HashTable::numStored() {
	return nstored;
}


int HashTable::hash(string key) {
	return smarterHash(key);
}

//=============================================================================================
===============
int HashTable::probeFunction(int val, int iter)
{
	// Linear probing.
	return (val + iter) % tsize;
}


int HashTable::sillyHash(string key) {
```

```
	return tsize/2;
}
//I create this function to convert character to number
int characterToNumber(char a)
{
	if(a == 'a' || a == 'A')
		return 1;
	else if(a == 'b' || a == 'B')
		return 2;
	else if(a == 'c' || a == 'C')
		return 3;
	else if(a == 'd' || a == 'D')
		return 4;
	else if(a == 'e' || a == 'E')
		return 5;
	else if(a == 'f' || a == 'F')
		return 6;
	else if(a == 'g' || a == 'G')
		return 7;
	else if(a == 'h' || a == 'H')
		return 8;
	else if(a == 'i' || a == 'I')
		return 9;
	else if(a == 'j' || a == 'J')
		return 10;
	else if(a == 'k' || a == 'K')
		return 11;
	else if(a == 'l' || a == 'L')
		return 12;
	else if(a == 'm' || a == 'M')
		return 13;
	else if(a == 'n' || a == 'N')
		return 14;
	else if(a == 'o' || a == 'O')
		return 15;
	else if(a == 'p' || a == 'P')
		return 16;
	else if(a == 'q' || a == 'Q')
		return 17;
	else if(a == 'r' || a == 'R')
		return 18;
	else if(a == 's' || a == 'S')
		return 19;
	else if(a == 't' || a == 'T')
		return 20;
	else if(a == 'u' || a == 'U')
		return 21;
	else if(a == 'v' || a == 'V')
		return 22;
	else if(a == 'w' || a == 'W')
		return 23;
	else if(a == 'x' || a == 'X')
		return 24;
	else if(a == 'y' || a == 'Y')
		return 25;
	else if(a == 'z' || a == 'Z')
		return 26;
}
```

```
int HashTable::smarterHash(string key) {
	// TO DO:: Write this.

	int product;
	for (int i = 0; i < key.length(); i++)
	{
		double num = 32;
		int power = key.length() - 1 - i;
		product = characterToNumber(key[i])*(pow(num,power));
	}
	int index = product % tsize;
	return index;
}


HashTable::HashTable() {
	init(default_size);
}


HashTable::HashTable(int nkeys) {
	init(4*nkeys);
}


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;
}



HashTable::~HashTable() {
	delete[] keys;
	delete[] values;
	delete[] sentinels;
}


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;

}
```

Part 1: Word frequencies

I have modified the smart smarterHash function by writing the new code below:

```
//I create this function to convert character to number
int characterToNumber(char a)
{
	if(a == 'a' || a == 'A')
		return 1;
	else if(a == 'b' || a == 'B')
		return 2;
	else if(a == 'c' || a == 'C')
		return 3;
	else if(a == 'd' || a == 'D')
		return 4;
	else if(a == 'e' || a == 'E')
		return 5;
	else if(a == 'f' || a == 'F')
		return 6;
	else if(a == 'g' || a == 'G')
		return 7;
	else if(a == 'h' || a == 'H')
		return 8;
	else if(a == 'i' || a == 'I')
		return 9;
	else if(a == 'j' || a == 'J')
		return 10;
	else if(a == 'k' || a == 'K')
		return 11;
```

```
        else if(a == 'l' || a == 'L')
                return 12;
        else if(a == 'm' || a == 'M')
                return 13;
        else if(a == 'n' || a == 'N')
                return 14;
        else if(a == 'o' || a == 'O')
                return 15;
        else if(a == 'p' || a == 'P')
                return 16;
        else if(a == 'q' || a == 'Q')
                return 17;
        else if(a == 'r' || a == 'R')
                return 18;
        else if(a == 's' || a == 'S')
                return 19;
        else if(a == 't' || a == 'T')
                return 20;
        else if(a == 'u' || a == 'U')
                return 21;
        else if(a == 'v' || a == 'V')
                return 22;
        else if(a == 'w' || a == 'W')
                return 23;
        else if(a == 'x' || a == 'X')
                return 24;
        else if(a == 'y' || a == 'Y')
                return 25;
        else if(a == 'z' || a == 'Z')
                return 26;
}
int HashTable::smarterHash(string key) {
        // TO DO:: Write this.

        int product;
        for (int i = 0; i < key.length(); i++)
        {
                double num = 32;
                int power = key.length() - 1 - i;
                product = characterToNumber(key[i])*(pow(num,power));
        }
        int index = product % tsize;
        return index;
}
```

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.

The average case time complexity of the pseudocode for word_frequencies, using the provided, naive hash function is O(n + n) = O(2*n), because the complexity time of first loop is O(n) and the average case time for second loop of hash table is O(n) without using good hash function. We need define the position of each word in hash table by using hash function h(k) and h(k) will let us know the index of each word in has table.

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

If we used a "good" hash function, I think the average case time complexity would be constant O(1).

Part 2: Indexing documents (bonus marks)

For the open-ended part, I have modified InvertedIndex to use separate chaining. I created a struct called ChainNode and wrote 2 more more functions called bool addChain(string key, string value) and bool lookupChain(string key, string value).

index_directory.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.getI;
                c = tolowerI;

                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;
                }
        }
}
```

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.
Struct ChainNode
{
        string value;
        ChainNode* link;
};
class InvertedIndex
{
 public:
        bool addChain(string key, string value);
        bool lookupChain(string key, string value);

        // 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.
        ChainNode* table[tsize];
```

```
        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);

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

};
```

InvertedIndex.h:

```
// Implement InvertedIndex methods
#include “InvertedIndex.h”
using namespace std;
bool InvertedIndex::addChain(string key, string value)
{
        int hval = hash(key);
        ChainNode* temp = new ChainNode();
        temp->value = value;
        temp->link = table[hval];
        table[hval] = temp;
        return true;
}

bool InvertedIndex::lookupChain(string key, string value)
{
        int hval = hash(key);
        ChainNode* temp = table[hval];
        while(temp != nullptr && temp->value != value)
        {
                temp = temp->link;
        }
        if(temp == nullptr)
        {
                return false;
        }
        else
        {
                return true;
        }
}

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);
	}

	// TO DO:: Grow if trouble.

	Return false;
}

// 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;
		}
	}
}



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);
        }
        return false;
}



int InvertedIndex::hash(string key) {
        return smarterHash(key);
}


int InvertedIndex::probeFunction(int val, int iter) {
        // Linear probing.
        Return (val + iter) % tsize;
}


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;
        }
}

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;
}
```

```
InvertedIndex::InvertedIndex() {
	init(default_size);
}


InvertedIndex::InvertedIndex(int nkeys) {
	init(4*nkeys);
}


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;
}



InvertedIndex::~InvertedIndex() {
	delete[] keys;
	delete[] values;
	delete[] sentinels;
}


int InvertedIndex::numStored() {
	return nstored;
}
```

THE END