template <class TYPE>

bool SimpleTable<TYPE>::update(const string& key, const TYPE& value){

int idx=search(key);

bool rc=false;

if(idx==-1){

if(size\_ < max\_){

records\_[size\_++]=new Record(key,value);

sort();

rc=true;

}

}

else{

rc=true;

records\_[idx]->data\_=value;

}

return rc;

}

//returns index of where key is found.

template <class TYPE>

int SimpleTable<TYPE>::search(const string& key){

int rc=-1;

for(int i=0;i<size\_;i++){

if(records\_[i]->key\_==key){

rc=i;

}

}

return rc;

}

**Search function**

**Unsuccessful search**

\* Let n represent the amount of data currently in the table

\* Let T(n) represent number of operations needed to find key in the table.

This function has 3 operations that execute exactly one time.

int rc=-1;

int i=0;

return rc;

There are 2 operations that are executed inside the loop

i<size\_;

i++;

**T(n) = 2(n) + 3**

Therefore T(n) is O(n)

**Successful search**

\* Let n represent the amount of data currently in the table

\* Let T(n) represent number of operations needed to find key in the table.

This function has 3 operations that execute exactly one time.

int rc=-1;

int i=0;

return rc;

There are 4 operations that are executed inside the loop

i<size\_;

i++;

records\_[i]->key\_==key

rc=i

**T(n) = 4(n) + 3**

Therefore T(n) is O(n)

template <class TYPE>

void SimpleTable<TYPE>::sort(){

int minIdx=0;

for(int i=0;i<size\_;i++){

minIdx=i;

for(int j=i+1;j<size\_;j++){

if(records\_[j]->key\_ < records\_[minIdx]->key\_){

minIdx=j;

}

}

Record\* tmp=records\_[i];

records\_[i]=records\_[minIdx];

records\_[minIdx]=tmp;

}

}

**Sort Function**

\* Let n represent the amount of data currently in the table

\* Let T(n) represent number of operations needed to sort keys inside the table.

This function has 2 operations that execute exactly one time.

int minIdx=0;

int i=0;

The first loop has 8 operations

i<size\_;i++ - 2operations

minIdx=i;

int j=i+1; 2 operations

Record\* tmp=records\_[i];

records\_[i]=records\_[minIdx];

records\_[minIdx]=tmp;

The second loop has 4 operations

j<size\_;

j++

records\_[j]->key\_ < records\_[minIdx]->key

minIdx=j;

T(n) = n\*(8+n\*(4))+2 = n(8+4n) + 2 = 8n + 4n^2 + 2

T(n) = 4n^2 + 8(n) + 2

**Therefore T(n) is O(n^2)**

template <class TYPE>

bool SimpleTable<TYPE>::update(const string& key, const TYPE& value){

int idx=search(key);

bool rc=false;

if(idx==-1){

if(size\_ < max\_){

records\_[size\_++]=new Record(key,value);

sort();

rc=true;

}

}

else{

rc=true;

records\_[idx]->data\_=value;

}

return rc;

}

**update() - if item does not exists so you need to add it as a new record**

Counting Operations:

int idx=search(key); 1 operation + unsuccessful search(**2(n) + 3)**

bool rc=false; 1 op

idx==-1 1 op

size\_ < max 1 op

records\_[size\_++]=new Record(key,value); 4 op (++,=, and 2 op in constructor)

sort() - 4n^2 + 8(n) + 2

rc=true; 1 op

return rc; 1 op

T(n) = 1+2(n)+3+7+4n^2+8(n)+2+2 = 4n^2 + 10(n) + 15

T(n) = 4n^2 + 10(n) + 15

**Therefore T(n) is O(n^2)**

**update() - if item does exists and you are just modifying what is there**

Counting Operations:

int idx=search(key); 1 operation + Successful search(**4(n) + 3)**

bool rc=false; 1 op

rc=true; 1 op

records\_[idx]->data\_=value; 1 op

return rc; 1 op

T(n) = 4(n) + 3 + 5

T(n) = 4(n) + 8

**Therefor T(n) is O(n)**

template <class TYPE>

bool SimpleTable<TYPE>::find(const string& key, TYPE& value){

int idx=search(key);

if(idx==-1)

return false;

else{

value=records\_[idx]->data\_;

return true;

}

}

**find() - if item is there**

Counting Operations:

int idx=search(key); 1 op + Successful search(**4(n) + 3)**

value=records\_[idx]->data\_; 1 op

return true; 1 op

T(n) = 4(n) + 3 + 3

T(n) = 4(n) + 6

Therefore T(n) is O(n)

**find() - if item is not there**

Counting Operations

int idx=search(key); 1 op + unsuccessful search(**2(n) + 3)**

if(idx==-1) 1 op

return false; 1 op

T(n) = 2(n) + 3 + 3

T(n) = 2(n) + 6

Therefore T(n) is O(n)

template <class TYPE>

bool SimpleTable<TYPE>::remove(const string& key){

int idx=search(key);

if(idx!=-1){

delete records\_[idx];

for(int i=idx;i<size\_-1;i++){

records\_[i]=records\_[i+1];

}

size\_--;

return true;

}

else{

return false;

}

}

**remove() - if item is there**

Counting Operations

This function has 6 operations that executed exactly 1 time and call search function results in 4(n) + 3

int idx=search(key); /1 op + Successful search(**4(n) + 3)**

idx!=-1 / 1 op

delete records\_[idx];/ 1 op

int i=idx / 1 op

size\_--;/ 1 op

return true;/ 1 op

This function has 5 operations that executed in the loop

i<size\_-1;i++ / 3 op

records\_[i]=records\_[i+1]; / 2 op

T(n) = 5(n-1) + 6 + 4(n) + 3 = 5(n) – 5 + 6 + 4(n) + 3 = 9(n) + 4

T(n) = 9(n) + 4

Therefore T(n) is O(n)

**remove() - if item is not there**

Counting Operations

int idx=search(key); 1 op + unsuccessful search(**2(n) + 3)**

return false; / 1 op

T(n) = 2(n) + 3 + 2

T(n) = 2(n) + 5

Therefore T(n) is O(n)

//copy constructor

template <class TYPE>

SimpleTable<TYPE>::SimpleTable(const SimpleTable<TYPE>& other){

records\_=new Record\*[other.max\_];

max\_=other.max\_;

size\_=0;

for(int i=0;i<other.size\_;i++){

update(other.records\_[i]->key\_,other.records\_[i]->data\_);

}

}

**copy constructor**

Counting Operations

This function has 4 operations that are executed exactly 1 time

records\_=new Record\*[other.max\_]; / 1 op

max\_=other.max\_; / 1 op

size\_=0; / 1 op

int i=0 / 1 op

This function has 2 operation and 1 function call inside the loop

i<other.size\_;i++ / 2 po

update(other.records\_[i]->key\_,other.records\_[i]->data\_); / 4n^2 + 10(n) + 15

T(n) = 4 + n(4n^2 + 10n + 15 + 2) = 4n^3 + 10n^2 + 17n + 4

T(n) = 4n^3 + 10n^2 + 17n + 4

Therefore T(n) is O(n^3)

//assignment operator

template <class TYPE>

const SimpleTable<TYPE>& SimpleTable<TYPE>::operator=(const SimpleTable<TYPE>& other){

if(this!=&other){

if(records\_){

int sz=size\_;

for(int i=0;i<size\_;i++){

delete records\_[0];

for(int j=0;j<sz-1;j++){

records\_[j]=records\_[j+1];

}

sz--;

}

delete [] records\_;

}

records\_=new Record\*[other.max\_];

max\_=other.max\_;

size\_=0;

for(int i=0;i<other.size\_;i++){

update(other.records\_[i]->key\_,other.records\_[i]->data\_);

}

}

return \*this;

}

**assignment operator**

Counting Operations

This function has 9 operations that executed exactly 1 time

this!=&other / 1 op

int sz=size\_; / 1 op

int i=0 / 1 op

delete [] records\_; / 1 op

return \*this; / 1 op

records\_=new Record\*[other.max\_]; / 1 op

max\_=other.max\_; / 1 op

size\_=0; / 1 op

int i=0 / 1 op

1st Inside loop has 5 operations

i<size\_;i++ / 2 op

delete records\_[0]; / 1 op

int j=0 / 1 op

sz--; / 1 op

2nd Inside loop has 5 operations

j<sz-1;j++ / 3 op

records\_[j]=records\_[j+1]; / 2 op

3rd Inside loop has 2 operations and a function call

i<other.size\_;i++ / 2 op

update(other.records\_[i]->key\_,other.records\_[i]->data\_); / / 4n^2 + 10(n) + 15

T(n) = 9 + n(5 + (n-1)(5)) + n(2 + 4n^2 + 10n +15)

T(n) = 9 + n(5+5n-5) + 2n + 4n^3 + 10n^2 + 15n

T(n) = 9 + 5n^2 + 10n^2 + 4n^3 + 17n

T(n) = 4n^3 + 15n^2 + 17n + 9

Therefore T(n) is O(n^3)

**Destructor**

template <class TYPE>

SimpleTable<TYPE>::~SimpleTable(){

if(records\_){

int sz=size\_;

for(int i=0;i<size\_;i++){

delete records\_[0];

for(int j=0;j<sz-1;j++){

records\_[j]=records\_[j+1];

}

sz--;

}

delete [] records\_;

}

}

Counting Operations

This function has 4 operations that executed exactly one time

if(records\_) / 1op

int sz=size\_; / 1 op

int i=0 / 1 op

delete [] records\_; 1 op

1st loop 5 operation in the loop

i<size\_;i++ / 2 op

delete records\_[0]; / 1 op

int j=0 / 1 op

sz--; / 1 op

2nd loop 5 operations

j<sz-1;j++ / 3 op

records\_[j]=records\_[j+1]; / 2 op

T(n) = 4 + n(5+(n-1)(5)) = 4+ n(5+5n – 5) = 4+5n^2

T(n) = 5n^2 + 4

Therefore T(n) is O(n^2)

**Part B (6 marks) Reflection On Simple Table**

template <class TYPE>

int SimpleTable<TYPE>::search(const string& key){

int rc=-1;

for(int i=0;i<size\_;i++){

if(records\_[i]->key\_==key){

rc=i;

break;

}

}

return rc;

}

1. **Add break; inside the loop.**

**It will stop the loop when the key is found. It will be more efficient to add this line as if the key is in the middle of the array, it just needs to go through half of the array which will decrease the amount of time and memory.**

//copy constructor

template <class TYPE>

SimpleTable<TYPE>::SimpleTable(const SimpleTable<TYPE>& other){

records\_=new Record\*[other.max\_];

max\_=other.max\_;

size\_=0;

for(int i=0;i<other.size\_;i++){

//update(other.records\_[i]->key\_,other.records\_[i]->data\_);

records\_[i] = new Record(other.records\_[i]->key\_, other.records\_[i]->data\_);

}

}

**Instead of Update function I will use the following code:**

records\_[i] = new Record(other.records\_[i]->key\_, other.records\_[i]->data\_);

In the update function if the key doesn’t exist it will add it at the end of the array in the table and will sort it which significantly increase the run time of the function.

Since the other table is already sorted and we just copying it, there is no need to use the sort function on the new copied table. We will just loop through all the elements of the other table and insert it in the new table. It will decrease the run time for the copy constructor which will become O(n) instead of O(n^3).

This code snippet will be used as well in the copy assignment operator instead of the update function.

**Destructor**

template <class TYPE>

SimpleTable<TYPE>::~SimpleTable(){

if(records\_){

int sz=size\_;

for(int i=0;i<size\_;i++){

delete records\_[0];

for(int j=0;j<sz-1;j++){

records\_[j]=records\_[j+1];

}

sz--;

}

delete [] records\_;

}

}

There is no need to delete and shift all the elements of the array in the destructor as the destructor should only delete and deallocate the elements of the array.

I will simply loop through the array and set each record to nullptr.

for(int i=0;i<size\_;i++){

records\_[i] = nullptr;