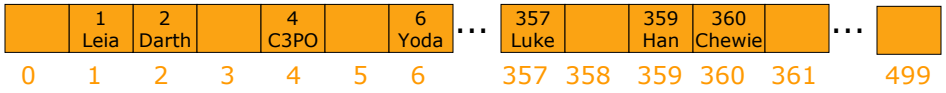
# Lecture 23: Implementing A Hash Table

PIC 10B

Todd Wittman

## Review: The Hash Concept

- A hash table is a fixed size list of records organized according to a unique key.
- The key is usually one of the data fields in the record.
- Each block of the hash table is called a bucket.
- Buckets may be empty, so the hash table wastes memory.
- The hash function maps the record's key to an integer called the hash index. This tells us which bucket to put the record into.

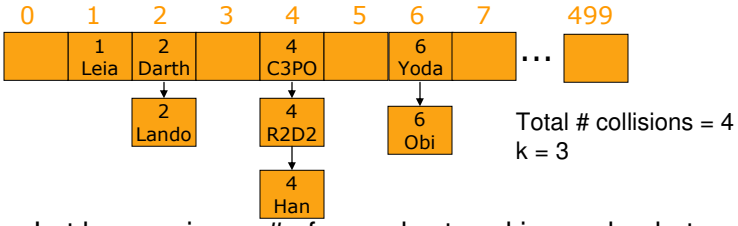



- If every key maps to a unique hash index, then the hash table operations are very fast.

| Search | Erase | Insert |
|---|---|---|
| O(1) | O(1) | O(1) |

# Review: Collisions & Chaining

- But it is very difficult to map each key to a unique hash index.
- A collision occurs when two keys are mapped to the same hash index.
- One way to resolve collisions is to allow each bucket to store multiple records. This is called chaining.




- Let k = maximum # of records stored in one bucket.

If implemented with vectors...

| Search | Erase | Insert |
|---|---|---|
| O(k) | O(k) | O(1) |

# Templating Our HashTable Class

- The other containers we built (LinkedList, Tree) could hold any type.
- But to be stored in a hash table, the data type must have a key associated with it.
- Also, we must have a way to hash that key to an integer index.
- We will build our HashTable class on 2 templated types:
  - T -- the data type of the record we will store
  - K -- the data type of the key
- We will assume that the record class T we are storing has the following 3 member functions:
  - K getKey () -- returns the key of the record
  - void setKey (K key) -- sets the key of the record
  - int getHash (int M) -- gets the hash index of the record, based on the hash table size M

# Example: The Record Class

- Suppose student records at Jedi Academy are indexed by the student name, e.g. "Luke Skywalker".

Name: Luke Skywalker
ID: 002-345-285
Class Rank: 357
GPA: 2.85
Major: History

```
class Record {
public:
    Record();
    string getKey();
    void setKey(string key);
    int getHash(int M);
    friend istream& operator>> (istream& in, Record& right);
    friend ostream& operator<< (ostream& out, const Record& right);
private:
    string name;      string id_number;
    int rank;     double GPA;      string major;
};
```

We need these 3 functions to use the Record class in our hash table.

# Example: The Record Class

- To organize the records, we need to know which data field is the key for the Record class.
- You could modify these functions to set the key as class rank, GPA, ID#, etc.

```
string Record::getKey() {
    return name;
}

void Record::setKey(string key) {
    name = key;
}
```

- We have to provide a way to map the key to a index, so we know which bucket to put the record into.
- Note the hash function depends on the size of the table M.

```
int Record::getHash(int M) {
    string key = getKey();
    int index = 0;
    for (int i=0; i<key.length(); i++)
            index += (int) key[i];
    index = index%M;
    return index;
}
```

# The HashTable Class

```
template <typename T,typename K>
class HashTable {
public:
    HashTable(int tableSize);
    void insert(T newRecord);
    T* find(K key);
    void erase(T* pos);
    template <typename T,typename K>
    friend ostream& operator<< (ostream& out, const HashTable& right);
private:
    vector< vector<T> > table;
};
```

- Do we need the Big 4?

# Vectors Within Vectors

- The private variable table is a vector of vectors of type T.
- To figure out how many buckets we have:

```
table.size()
```

- To find out how many records are in the $i^{th}$ bucket:

```
table[i].size()
```

- To access the $j^{th}$ record in the $i^{th}$ bucket:

```
table[i][j]
```

- To look up the key of the $j^{th}$ record in the $i^{th}$ bucket:

```
table[i][j].getKey();
```

# Constructing A Hash Table

- To create a HashTable, we specify how many buckets we want in the table vector.

```
template <typename T,typename K>
HashTable<T,K>::HashTable(int tableSize) {
        table.resize(tableSize);
}
```

- The general rule for hash tables is:

  More Buckets = More Memory Used
  = Fewer Collisions
  = Faster Operations

- Note our code specifies 2 templates: record type and key type.
- For example, to create a hash table with 50 buckets that stores our Record class organized by the student's name:

```
HashTable<Record, string> myHashTable (50);
```

# Inserting A Record

- First we need to figure out which bucket the new record should go into.
- We look up the hash index for that record, sending it the number of buckets M. The returned value should be in range [0,M-1].
- To insert a record into bucket i, we push_back onto the bucket vector table[i].

```
template <typename T,typename K>
void HashTable<T,K>::insert(T newRecord) {
        int index = newRecord.getHash(table.size());
        table[index].push_back(newRecord);
}
```

- With this implementation, inserting a record is O(1).
- But records within a bucket are unsorted, so our search and erase operations will be slower.

# Finding A Record

- We want to return a pointer to the record in the hash table that contains the given key.
- First we need to figure out what bucket the key maps to.

```
template <typename T,typename K>
T* HashTable<T,K>::find(K key) {
    T tempRecord;
    tempRecord.setKey(key);
    int index = tempRecord.getHash(table.size());
    for (int i=0; i<table[index].size(); i++)
        if (table[index][i].getKey() == key)
            return &table[index][i];
    return NULL;
}
```

Create a dummy record so that we can look up this key's hash index. This tells us what bucket to look in.

Check every record in bucket table[index]. Return the record address if we find it.

Return NULL if the record was not found.

- To print the record that contains the key "Luke Skywalker", we write:

```
cout << *(myHashTable.find("Luke Skywalker"));
```

# Erasing A Record

```
template <typename T,typename K>
void HashTable<T,K>::erase(T* pos) {
   if (pos == NULL)   return;
   int index = pos->getHash(table.size());
   int i=0;
   while (&table[index][i] != pos && i < table[index].size())
       i++;
   for (int j=i; j<table[index].size()-1; j++)
       table[index][j] = table[index][j+1];
   table[index].pop_back();
}
```

First look up the hash index of the record that pos points to. This tells us to look in bucket table[index].

Find the position i of record in its bucket.

Shift all remaining records after position i to the left one step.

Now we can safely erase the last record.

- To erase the record with the key "Luke Skywalker", we write:

```
myHashTable.erase( myHashTable.find("Luke Skywalker") );
```

- What happens if "Luke Skywalker" was not in the table?

# Printing A Hash Table

- We made the operator<< a friend function, so it can access the records in the hash table directly.
- This assumes operator<< is defined for the data type stored in the table.

```
template <typename T,typename K>
ostream& operator<< (ostream& out, const HashTable<T,K>& right) {
    for (int i=0; i < right.table.size(); i++)
        for (int j=0; j < right.table[i].size(); j++)
            out << "Bucket " << i  << ", Record " << j
                << "\n" << right.table[i][j] << "\n\n";
    return out;
}
```

- Print-out for `cout << myHashTable;` looks like:

```
Bucket 0, Record 0
Luke Skywalker

Bucket 0, Record 1
Darth Vader
```

# Example: Jedi Academy Records

- Suppose we have a file "students.txt" that lists student records at Jedi Academy. Organize the records by name.

```
int main() {
    HashTable<Record,string> myHash(50);
    ifstream fin;
    fin.open("students.txt");
    Record newStudent;
    string blank_line;
    while (fin >> newStudent) {
        myHash.insert(newStudent);
        getline(fin,blank_line);
    }
    fin.close();
    myHash.erase(myHash.find("Luke Skywalker"));
    cout << myHash;
}
```

students.txt

```
Luke Skywalker
333-222-111
357
2.85
History

Leia Organa
283-528-233
1
3.96
Political Science

Darth Vader
666-666-666
3
3.87
Evil
```

# HashTable Statistics

- We'd like to track some basic hash table statistics.
  - The number of records in the table: countRecords()
  - The total number of collisions: countCollisions()
  - The maximum number of records in one bucket: largestBucket()
- You will write these functions in HW9.
- Note the number of records does not necessarily equal the number of buckets.
- To count the total collisions, we add up the size of every non-empty bucket - 1.

```
int numCollisions = 0;
for (int i=0; i<table.size(); i++)
        if (table[i].size() > 1)
                numCollisions += table[i].size()-1;
```

# Hash Tables & Vectors & Trees. Oh My!

- We implemented our hash table as a vector of vectors.
- What if we implemented our hash table as a vector of trees?
- Let k = maximum # records in a bucket.
- On average, the run time would be

| Search | Erase | Insert |
|---|---|---|
| O(log k) | O(log k) | O(log k) |