Lecture 3: Hashing

EECG142- Data Structures

Textbook:
Data Structures via C++: Objects by Evolution
by A. Michael Berman

First year - EECE Department

Spring 2025

Retrieving an item

- Searching is a recurrent task in different application. You may need to retrieve an item using a key (e.g., record of a student)
- A straight-forward way is to <u>compare the key with every stored key</u>
- The complexity grows <u>linearly</u> with the number of stored keys.

Warehouse Example

- Instead of sorting parts in the warehouse alphabetically, you could assign each part an ID (e.g., P0687-Z23). The ID is then mapped to a storage bin (e.g., A12)
- When workers need a part, they input its ID to a mapping function that tells them its storage bin.
- This leads to efficient storage and fast retrieval

Fast retrieval using Hashing

- Hashing is a technique that enables searching the data with a complexity O(1).

- Hashing is the process of turning the key value into a pointer (or index) that is used to locate an item stored a larger array.

- Hash functions should be designed to give different values for different keys to avoid collisions (although this cannot be guaranteed).

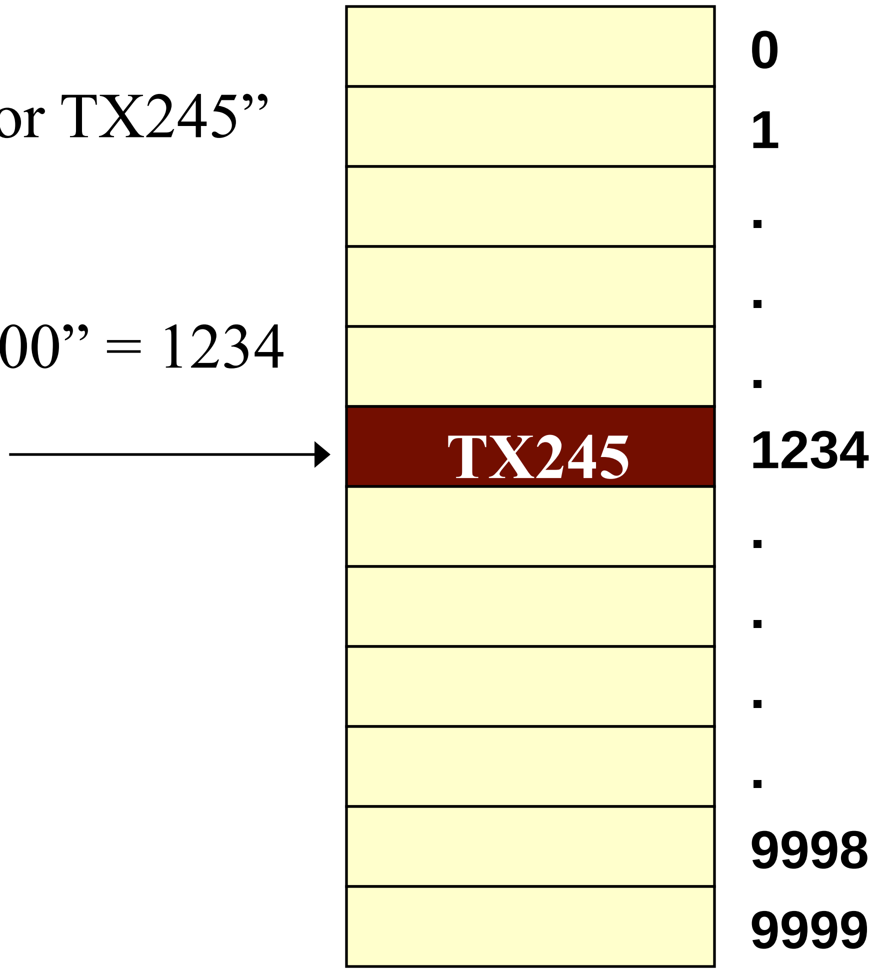
Product code hash example

- Assume we want to store products datasheets by the product's 5-digit code which starts from "10000".
- A simple hash function would be "Product code – 10000" or "Product code % 10000"
- This hashing returns a number between "0" and "9999" which can be the index of the storge bin having the product's datasheet.

Product code hash example

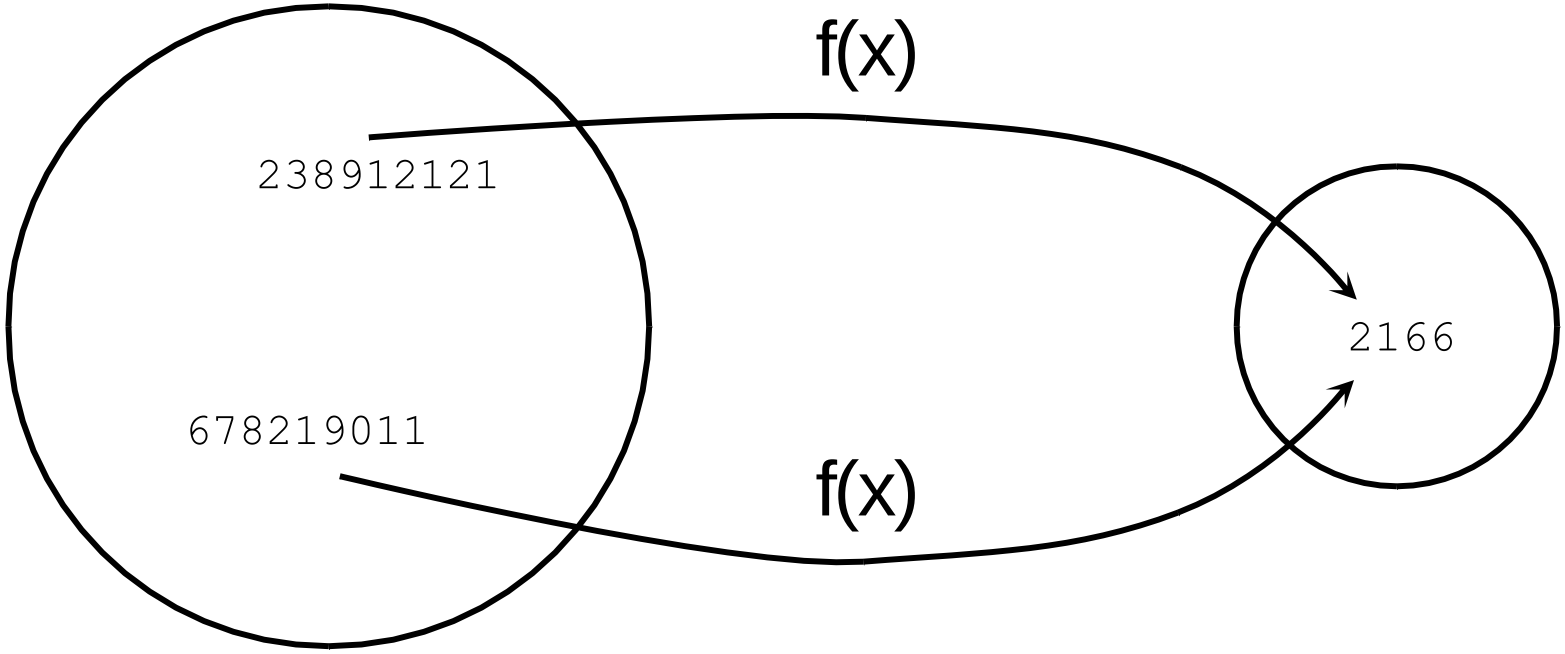
Product: "Temperature Sensor TX245"
Product code: "11234"

Storage index: "11234 – 10000" = 1234



Collisions in Hashing

- A hash function should map different keys to different values <u>(ideally a one-to-one mapping)</u> otherwise a collision happens



Collision Example

Try the following hash function:

storage index = “Product code % 1000”

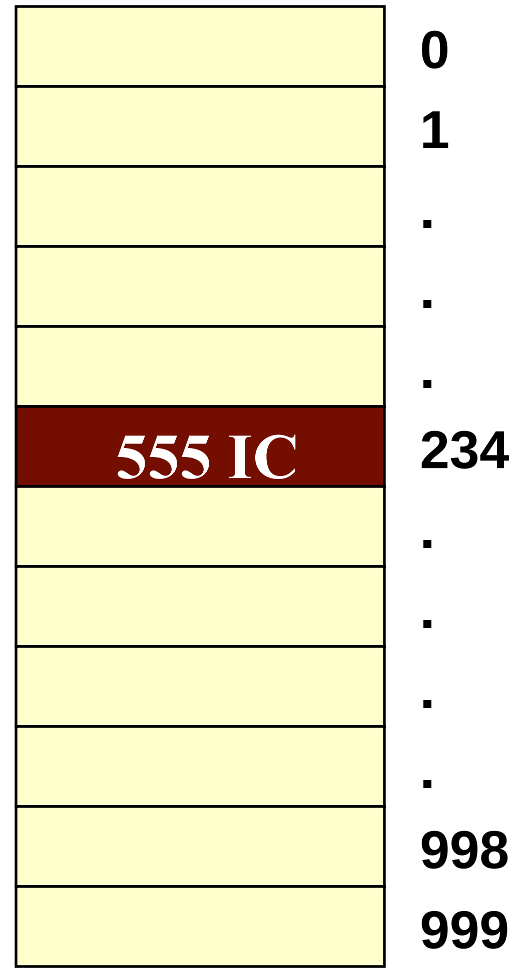
<u>Product:</u> “Temperature Sensor TX245”

<u>Product code: </u>11234

<u>Product:</u> “ 555 IC”

<u>Product code: </u>12234

TX245



Both products have storage index 234

What is a good hashing function?

- **Good Hash functions should**
  - Be consistent (generate same value for same key)
  - minimize collisions
  - resolve collisions whenever they happen.

Reducing collisions

- Generally, a collision-free hash function is not easy to design.
  - A usual approach is to use an array size that is larger than needed. The extra array positions make the collisions less likely
  - A good hash function will distribute the keys uniformly throughout the locations of the array.

Collison Resolution

Liner Probing

- If a collision occurs, check the next available slot sequentially
- Linear probing is easy but causes clustering of keys
- If the hash function is (key % 7), then linear probing would handle the keys 15, 22, 29 as follows

Key	Hash Calculation	Initial Index	Final Placement
15	$15 \mod 7 = 1$	1	Placed at **1**
22	$22 \mod 7 = 1$	1 (collision) → **check next**	Placed at **2**
29	$29 \mod 7 = 1$	1 (collision) → check 2 (collision) → **check next**	Placed at **3**

Collison Resolution

<u>Double Hashing</u>

- Instead of moving sequentially, use a second hash function to determine the step size (this reduces clustering)
- **For example,** we have two hash functions

  <u>Primary hash: </u>hash1(key) = key % 7 and

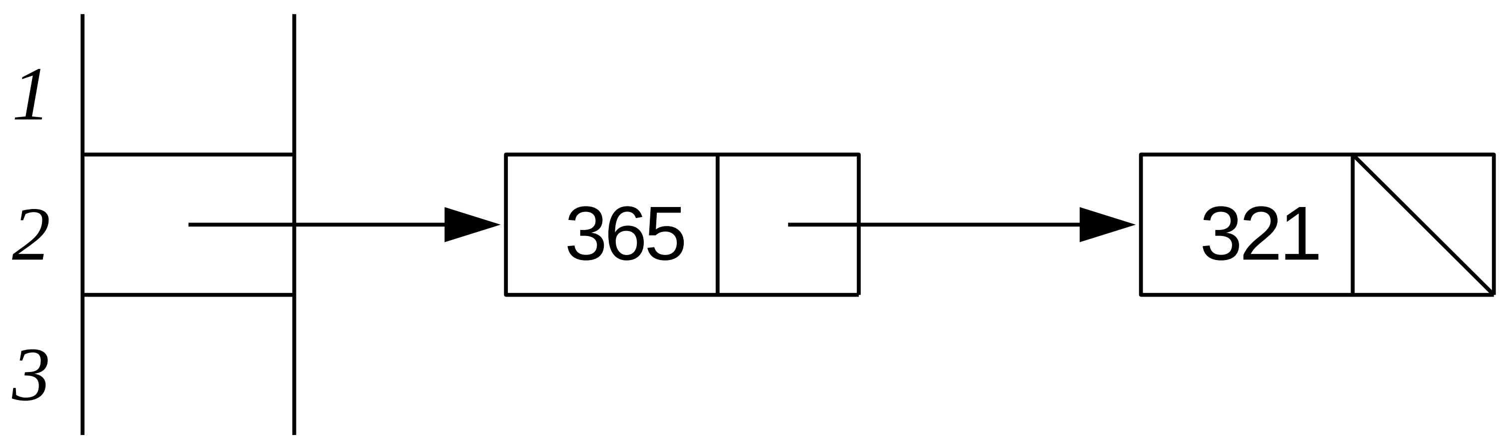
  <u>Step size: </u>hash2(key) = 5 - (key % 5)
- **Back to the old example, to insert 22 we have**

hash1(22) = 1 (collision), hash2(22) = 3, then insert 22 in location (1+3=4) if empty. If location 4 is busy, then try location (1+2*3=7) and so on.

Collison Resolution

<u>Chained Hashing</u>

- **Each component of the hash table's array can hold more than one entry.**
- We hash the key of the item. If a collision happened, we simply append the new item to the other items in the hashed index
- The most common implementation is having each array element be a linked list.



Solve these problems

1. **Consider a hash table of size 13 and hash function key % 13. Draw a representation of a chained hash table if entries with the following keys are added to the table in the sequence provided: 2, 13, 15, 3, 26, 5, 18, 16.**
2. **After insertions are done in problem 1, find how many steps are needed when the following lookup operations are performed on the table: lookup(9), lookup (26), lookup(16).**
3. **Repeat 1 and 2 for a table with linear probing**
4. **Repeat 1 and 2 for a table with double hashing with the second hash function key% 7**

Table Header File

```
const int MAX_TABLE_SIZE = 11;
template < class tableDataType > class Table
{
public:
Table( ); // Table constructor

void insert(int key, tableDataType data);
//  data and associated key are stored in the Table, If there was already data stored with this key, the insert call will replace it

bool lookup(int key, tableDataType & data);
// If key is in the table, returns true and associated data is returned; otherwise, false is returned and data is undefined.
void deleteKey(int key); // deletes the entry associated with key
```

Table Header File

```
private:
enum ItemType {Empty, Deleted, InUse}; //define data type that takes  specified values

struct Item{ ItemType Status;  int key; tableDataType data; };
Item Table[MAX_TABLE_SIZE]; // stores the items in the table
int num_entries; // keep track of number of entries in table
int hash(int key){ return key % MAX_TABLE_SIZE}
bool search(int & pos, int target)
{for ( ; Table[pos].Status != Empty; pos = (pos+1) % MAX_TABLE_SIZE)
if (Table[pos].Status == InUse && Table[pos].key == target)  return true;
return false; }
// pos is the hash address of target. if target is in the table, pos is set to its actual slot
// Returns: true if target is in the table, else false
```

Linear Probing Implementation

```
template < class tableDataType >Table < tableDataType >::Table( )
// implementation of Table constructor
{entries = 0;
for (int i = 0; i < MAX_TABLE_SIZE; i++)
Table[i].Status = Empty;}


template < class tableDataType >void Table < tableDataType >::deleteKey(int key)
{
int pos = hash(key);
if (search(pos, key)) {
Table[pos].Status = Deleted;
entries--;
}

}
```

Linear Probing Implementation

```
template < class tableDataType >void Table <tableDataType >::insert(int key,
tableDataType data)
{
assert(entries < MAX_TABLE_SIZE );
int pos = hash(key);  // find a position to insert the item
if (!search(pos, key)) // key was not in the table
{
pos = hash(key);
while (Table[pos].Status == InUse)
pos = (pos+1) % MAX_TABLE_SIZE;
entries++;
}
//Now pos is the index where we will insert the key.
Table[pos].Status = InUse;
Table[pos].key = key;
Table[pos].data = data;

}
```

Linear Probing Implementation

```
template < class tableDataType >bool Table < tableDataType > ::lookup(int key,
tableDataType & data)
int pos = hash(key);  // find a position to insert the item
if (search(pos, key)) // key found in the table
{
data = Table[pos].data;
return true;
}
return false;
}
```

Chained Hash Table

```
template <class ListElementType>
class List
{
public:
List();
void insert(ListElementType elem);
bool first(ListElementType & elem);
bool next(ListElementType & elem);
void UpdateCurrent(ListElementType e);
void deleteCurrent();
```

Chained Hash Table

```
const int MAX_TABLE_SIZE = 11;
template < class tableDataType > class Table
#include "List.h"
{
public:  // same functions as before
private:
struct Item {
int key;
tableDataType data;
};
List<Item> Table[MAX_TABLE_SIZE]; // stores the items in the table
int hash(int key)
{ return key % MAX_TABLE ;}
```

Chained Hashing Table

```
template < class tableDataType > void Table<tableDataType>::insert(int key,
tableDataType data)
{Item ThisItem, NewItem;
 NewItem.key=key;    NewItem.data=data;
int pos = hash(key);
bool ListExists=Table[pos].first(ThisItem);
While (ListExists)
{
if(ThisItem.key==key)
{Table[pos].UpdateCurrent(NewItem) ;return;}
ListExists=Table[pos].next(ThisItem);
}
Table[pos]. insert (NewItem);
}
```

Chained Hashing Table

```
template < class tableDataType >bool Table<tableDataType>::lookup(int key,
tableDataType & data)
{
Item ThisItem; int pos=hash(key);
ListExists=Table[pos].first(ThisItem);
while(ListExists)
{
if(ThisItem.key==key) {data=ThisItem.data; return true;}
ListExists=Table[pos].next(ThisItem);
}
return false;
}
```

Chained Hashing Table

```
template < class tableDataType > void Table < tableDataType >::deleteKey(int
key)
{
int pos = hash(key); Item ThisItem; bool ListExists

ListExists=Table[pos].first(ThisItem);
while(ListExists)
{
if(ThisItem.key==key) {Table[pos].deleteCurrent();return;}
ListExists=Table[pos].next(ThisItem);
}

}
```

Extra functions in List

```
template <class ListElementType>
void List<ListElementType>::UpdateCurrent(const ListElementType & e)
{
assert(current);
current->elem=e;

}
```

Extra Functions in List

```
template <class ListElementType> void List<ListElementType>::deleteCurrent()
{
assert(current);
Link del, p;
if(head==current)
{ del=current; head=current->next; current=current->next;
delete del;
return;
}
for(p=head;p->next!=current;p=p->next);
del=current;
p->next=current->next;
current=current->next;
delete del;

}
```