CHAPTER 9:

MAPS, HASH TABLES AND SKIP LISTS

2 MAPS

MAPS

- A **map** is a data structure that follows the composition method, i.e., **it stores key-value pairs** (k,v).
- Each pair (k,v) is called an "entry"
  - The key k and the value v can be of any object type
- **Question:** Can we have multiple entries that have the same key?

  Answer: Not in maps

MAP ITERATOR ADT

- A map provides an **iterator**. Given an iterator p:
  - We can access the **entry** associated with p by typing *p
  - We can access the **key** of this entry by typing p->key()
  - We can access the **value** of this entry by typing p->value()
  - We can **advance** the iterator to the next entry by typing ++p
  - We can **enumerate** all entries in a map M by typing:

```
for( iterator p = M.begin();  p != M.end();  p++ ) . . .
```

MAP ADT

- A map ADT supports the following methods:

  size(): Return the **number of entries** in *M*.

  empty(): Return **true if *M* is empty** and false otherwise.

  begin(): Return an **iterator to the first entry** of *M*.

  end(): Return a sentinel **iterator to an imaginary entry just beyond the end** of *M*.

  find(*k*): If *M* contains an entry *e* whose key equals *k*, then **return an iterator referring to *e***; **otherwise return the special iterator "end".**

  put(*k*,*v*): If *M* does not have an entry whose key equals *k*, **add entry (*k*,*v*) to *M***; otherwise replace the value of this entry with *v*; **return an iterator to the entry**.

  erase(*k*): **Remove the entry whose key equals *k*** (ERROR if *M* has no such entry).

  erase(*p*): **Remove the entry referenced by iterator *p*** (ERROR if *p* points to iterator "end")

MAP – METHOD IMPLEMENTATION

- Here are the algorithms (in pseudo code) of the main methods in the Map ADT:

Algorithm find(k): $O(n)$
  Input: A key k
  Output: The position of the matching entry of L, or end if there is no key k in L

```
for each position p ∈ [L.begin(),L.end()] do
    if p.key() = k then
        return p
return end {there is no entry whose key = k}
```

Algorithm erase(k): $O(n)$
  Input: A key k ***Output:*** None

```
flag ← 0
for each position p ∈ [L.begin(),L.end()] do
    if p.key() = k then
        L.erase(p); flag ← 1
        n ← n−1 { update number of entries }
        break
if flag = 0 then
    print "Entry_not_found_message"
```

Algorithm put(k,v): $O(n)$
  Input: A key-value pair (k,v)
  Output: The position of the inserted/modified entry

```
for each position p ∈ [L.begin(),L.end()] do
    if p.key() = k then
        *p ← (k,v)
        return p { return the position of
                   the modified entry }
p ← L.insertBack((k,v)) or p ← L.push_back((k,v))
{ for linked list or array implementation, when we
don't find an entry with key = k }
n ← n+1 {update number of entries}
return p {return the position of the inserted entry}
```

MAP EXAMPLE

- In this example, we write:

$$p_i : [(k, v)]$$

to mean that the operation returns an iterator p_i that refers to the entry (k, v).

- The entries of the map **are not** listed in any particular order.

Operation	*Output*	*Map*
empty()	**true**	$\emptyset$
put$(5,A)$	$p_1 : [(5,A)]$	$\{(5,A)\}$
put$(7,B)$	$p_2 : [(7,B)]$	$\{(5,A),(7,B)\}$
put$(2,C)$	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C)\}$
put$(2,E)$	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
find(7)	$p_2 : [(7,B)]$	$\{(5,A),(7,B),(2,E)\}$
find(4)	end	$\{(5,A),(7,B),(2,E)\}$
find(2)	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
size()	3	$\{(5,A),(7,B),(2,E)\}$
erase(5)	–	$\{(7,B),(2,E)\}$
erase(p_3)	–	$\{(7,B)\}$
find(2)	end	$\{(7,B)\}$

MAP – C++ IMPLEMENTATION

- Here, **an entry** (k,v) **will be implemented as the following class**, i.e., each entry will be an object of this class:

```
template <typename K, typename V>
class Entry {
public:
    Entry(const K& k = K(), const V& v = V()) : _key(k), _value(v) { }
    const K& key() const { return _key; } // get key
    const V& value() const { return _value; } // get value
    void setKey(const K& k) { _key = k; } // set key
    void setValue(const V& v) { _value = v; } // set value
private:
    K _key; // the key of the entry
    V _value; // value
};
```

an initializer list where we set _key = k and _value = v

MAP – C++ IMPLEMENTATION

- This class represents the entire Map:

```
template <typename K, typename V>
class Map {
public:
    class Entry;      // a nested class of a (key,value) pair
    class Iterator;   // a nested class of an iterator
    int size() const; // number of entries in the map
    bool empty() const; // returns “true” if the map is empty
    Iterator find(const K& k) const; // find entry with key k
    Iterator put(const K& k, const V& v); // insert/replace pair (k,v)
    void erase(const K& k) throw(NonexistentElement); // remove entry with key k
    void erase(const Iterator& p); // erase entry at p
    Iterator begin(); // returns an iterator to first entry in the map
    Iterator end(); /* returns an iterator to an imaginary entry just beyond the end
                       of the map */
};
```

Outside the class, these would be accessed by writing:

- Map<K,V>::Entry
- Map<K,V>::Iterator

e.g., we may write:

- Map<string, int>::Iterator p;

STL MAP

- When using C++'s **STL Map**, here is how we create entries:

  - First you have to create the map variable and its iterator:

```
#include <iterator>
#include <map>
map<dataType1, dataType2> myMap;
map<dataType1, dataType2>::iterator p;
```

  - Then, given two objects, a **key** k and a **value** v, we can create an object consisting of these two, i.e., an entry (k,v), by writing:

```
pair<dataType1, dataType2>(k, v)
```

  - This way, given an iterator p referring to an entry, we can write:

    p->first (which returns the **key**)   p->second (which returns the **value**)

STL MAP

size(): Return the **number of entries** in *M*.

empty(): Return **true if *M* is empty** and false otherwise.

begin(): Return an **iterator to the first entry** of *M*.

end(): Return a sentinel **iterator to an imaginary entry just beyond the end** of *M*.

find(*k*): **Find the entry with key *k*** and return an iterator to it; if no such key exists return end

insert(pair<dataType1, dataType2> (*k*,*v*)): **Insert pair (*k*,*v*)**, returning an iterator to its position.

erase(*k*): **Remove the entry whose key equals *k***

erase(*p*): **Remove the entry referenced by iterator *p***

operator[*k*]: Return a **reference to the value of the entry whose key is *k***; if no such key exists, create a new entry whose key is *k*.

With this, we can write: M[**k**]=**v** which would either:

- **modify** the value of the entry whose key is **k** (if there is an entry with key = **k**)
- or **insert** the pair (**k**,**v**) in M (if there is no entry in M with key = **k**)

We can also write M[**k**], which would be equivalent to performing find(**k**) and accessing the value of the entry whose key is **k**

STL MAP

- Here is an example of how to use C++’ map STL:

```
map<string, int> myMap; // a (string,int) map
map<string, int>::iterator p; // an iterator to the map
myMap.insert(pair<string, int>("Rob", 28)); // insert (“Rob”,28)
myMap["Joe"] = 38; // insert(“Joe”,38)
myMap["Joe"] = 50; // change to (“Joe”,50)
myMap["Sue"] = 75; // insert(“Sue”,75)
p = myMap.find("Joe"); // *p = (“Joe”,50)
myMap.erase(p);  // remove (“Joe”,50)
myMap.erase("Sue"); // remove (“Sue”,75)
p = myMap.find("Joe"); // now p refers to end, i.e., the imaginary entry
if (p == myMap.end()) cout << "nonexistent\n"; // outputs: “nonexistent”
for (p = myMap.begin(); p != myMap.end(); ++p) // print all entries
    cout << "(" << p−>first << "," << p−>second << ")\n";
```

HASH FUNCTION

- With arrays, we can instantly access an **element** given its **index**

- It would be great if we could also instantly access the **value** of an entry within a map given its **key**

The problem is: the **key** can be of any object type, and not necessarily integer!

Solution: Use a function that **converts keys into positive integers**

- Such a function is called a **hash function**, and the array where the entries will be stored is called a ***hash table*.**

- Thus, to store an entry (k,v) in a **hash table** A:

    1. **Convert** k into a positive integer i using a hash function

    2. Then, **store** v in A[i]

HASH TABLE – EXAMPLE

- Suppose our **keys are real values**, and we use a hash function that **rounds the absolute value** of the key to produce a positive integer

Entries:	(1.3, A)	(-2.2, B)	(3.2, C)	(-4.6, D)	(7.4, E)	(5.1, F)
Output of **hash function:**	1	2	3	5	7	5

Hash table:

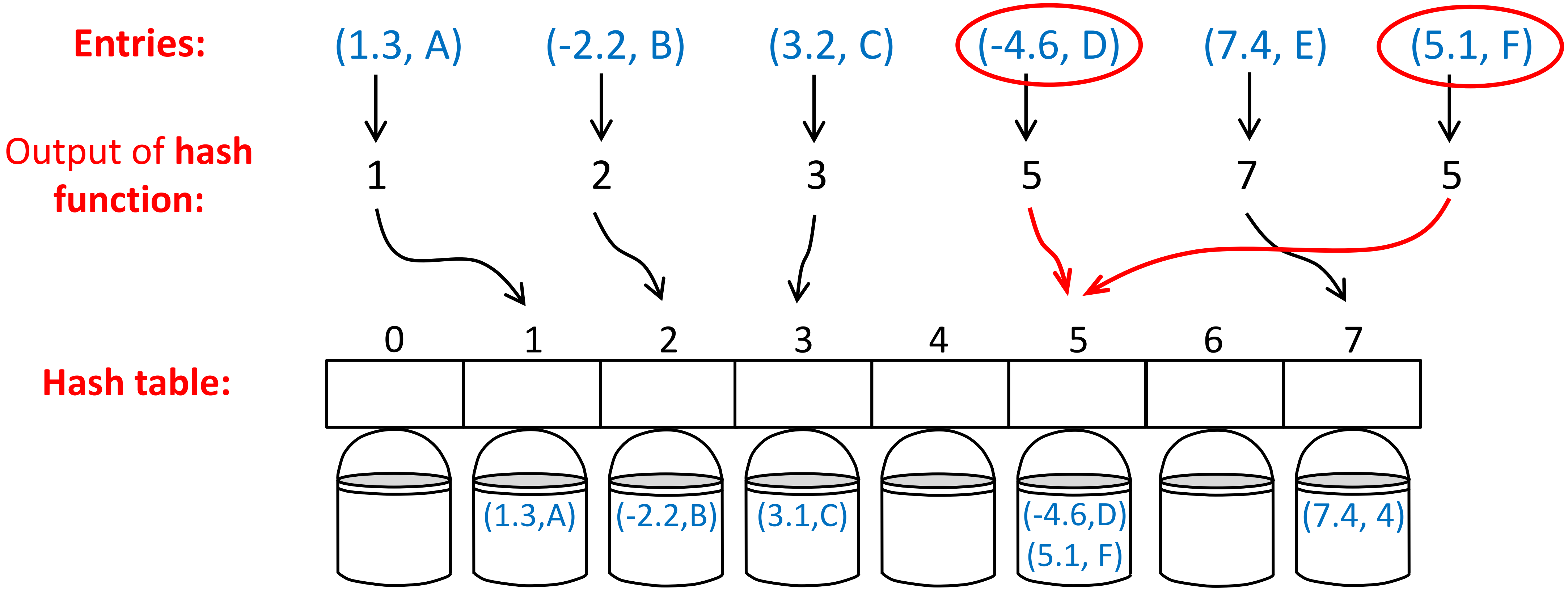
0	1	2	3	4	5	6	7
	(1,3, A)	(-2.2, B)	(3.1, C)		(-4.6,D)		(7.4, 4)

- This could cause a problem. **Can you spot it?**

There could be two entries whose keys are different, but after using the hash function, their keys become identical! This is called a “collision”

HANDLING COLLISION

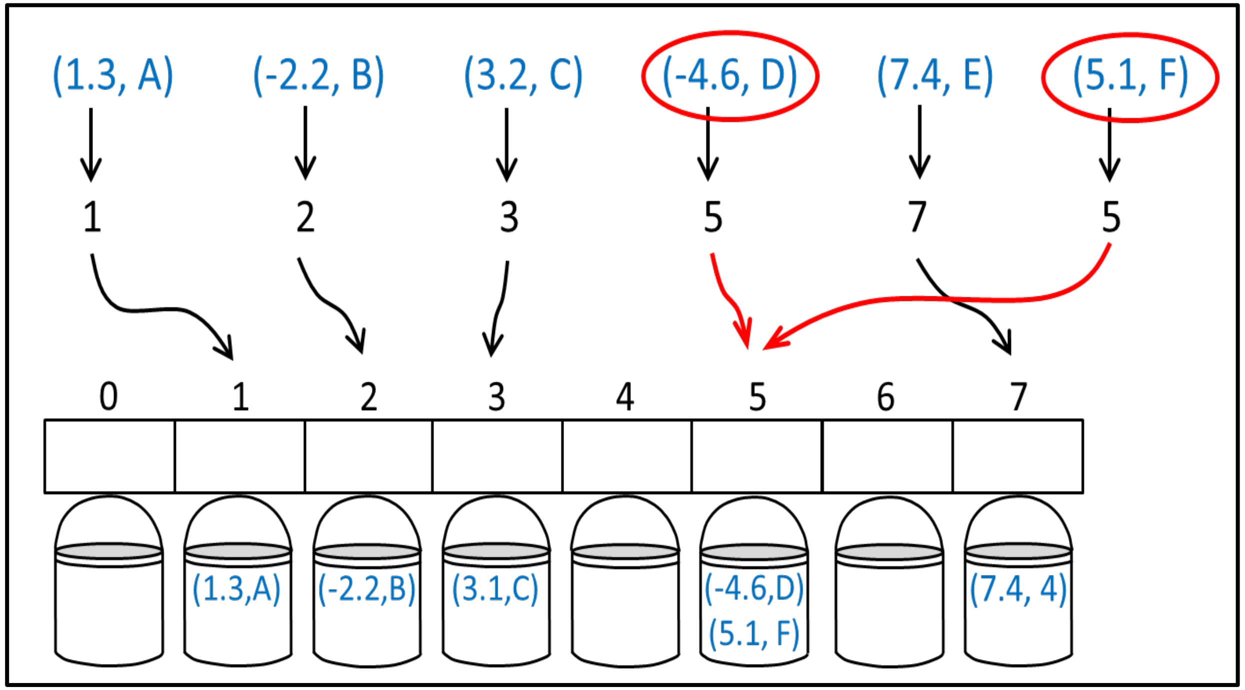
- One way to deal with collision is to **store in each slot a list that can store multiple entries!**



- This is called the **bucket array**, since list can be illustrated as a bucket.

SEARCHING THROUGH BUCKETS

- If we use a bucket array, then to retrieve a value given its key, we have to **search the bucket at that key!**
- For example, given the key 5.1, we would have to convert 5.1 to 5 using the hash function, then search the bucket at A[5] for an entry whose key = 5.1

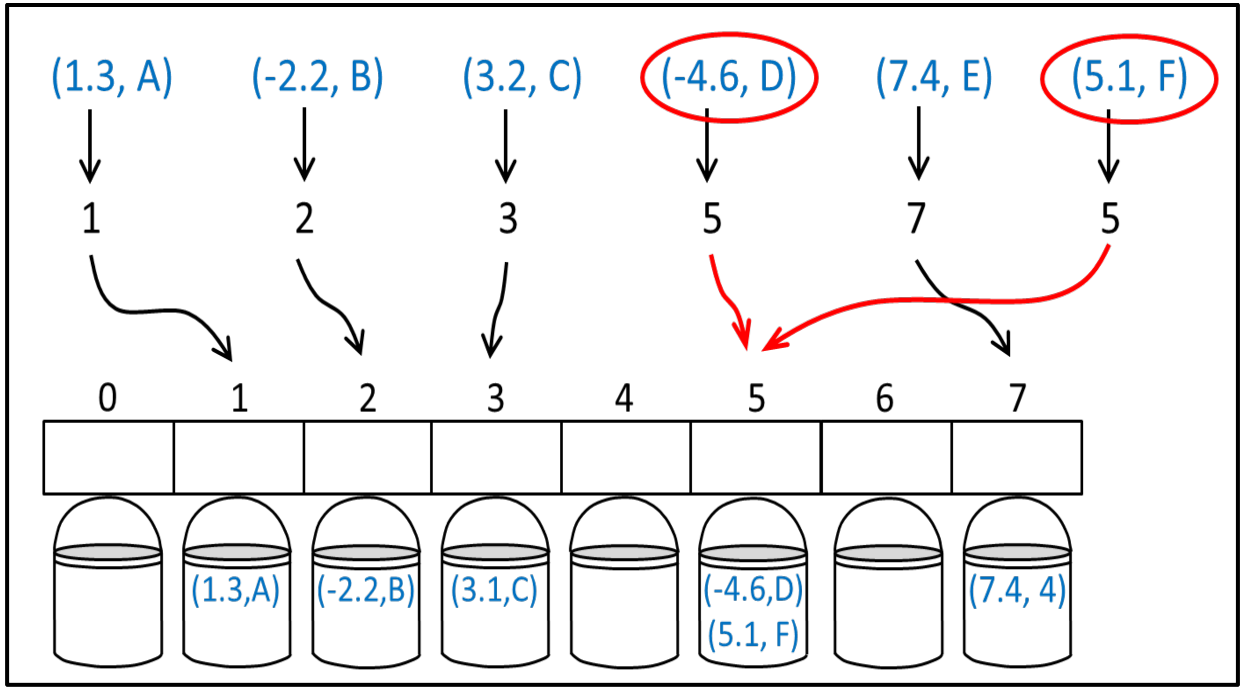


- A bad hash function results in too many entries placed in a single bucket, even to the point of making the search through that bucket take *O(n)* time!
- Thus, ideally we would have a hash function that minimizes collision, so that the entries are distributed over the hash table as evenly as possible

SEARCHING THROUGH BUCKETS

- **Each bucket can be map implemented as a linked list.** This way, we can readily use the methods that come with the map!

P.S., here the hash function is denoted "h"



Algorithm find(k):
 Output: The position of the matching entry of the map, or end if there is no key k in the map
 return A[$h(k)$].find(k) {delegate the find(k) to the map at $A[h(k)]$}

Algorithm put(k,v):
 $p \leftarrow A$[$h(k)$].put(k,v) {delegate the put to the map at $A[h(k)]$}
 $n \leftarrow n+1$
 return p

Algorithm erase(k):
 Output: None
 A[$h(k)$].erase(k) {delegate the erase to the map at $A[h(k)]$}
 $n \leftarrow n-1$

SUMMARY

- Summary: we said we wanted a way to **quickly access an entry given its key**
- To achieve this, we presented the following:
  - **Hash functions**
  - **Hash tables**
  - **Collisions**
  - **Bucket arrays**
- We also mentioned that **a good hash function minimizes collision** as much as possible, to spread the entries as evenly as possible over the hash table.
- Let us now dig into hash functions and they work

HASH FUNCTION

- A hash function can involve **two steps**:

  1. Convert the key to an integer $\in [-\infty, \infty]$ called the **hash code**

  2. Use something called a "**compression function**" that converts the hash code to a positive integer in $\in [0, N-1]$, where N is the size of the hash table.

- The alternative hash codes that we will discuss here are based on the assumption that *the number of bits of each type is known*.

- To know the number of bits for any type T, write: #include <limits>, then write: std::numeric_limits<T>::digits, For example:

```
#include <limits>
cout << std::numeric_limits<char>::digits; //this would print "7" -> 8 bits
```

HASH CODE

- We need to convert a key to a **hash code**, i.e., an integer $\in [-\infty, \infty]$
- If the key k is of type `char`, `byte`, or `short`, how can we convert it to `int`?
- Since k is **8-bits**, and `int` takes **32-bits**, just fill the remaining **24 bits** with zeros

Example:

- Give k = ‘a’, the binary representation of its ascii code (i.e. 97):

2^7 $\boxed{2^6}$ $\boxed{2^5}$ 2^4 2^3 2^2 2^1 $\boxed{2^0}$ = 01100001

We **cast** k to `int`, e.g., by writing:

```
int myHashCode = (unsigned int) k;
```

`int` can take values from $-(2^{31})$ to 2^{31}-1, whereas `unsigned int` can take values from 0 to 2^{32}-1

- This would produce a hash code equal to **97**, whose binary representation is:

00000000000000000000000001100001

SUMMATION HASH CODE

- What if the key k is of a type that requires **more than 32 bits**?
- We can use a **summation** hash code, which works as follows:
  - **Cut** k into pieces, $x_0, x_1, \ldots, x_{m-1}$ that each takes ≤ 32 bits
  - **Cast** each piece into `int`
  - **Sum** up all the pieces
- This could cause a problem, can you spot it?

  E.g., given two keys, k_1 and k_2 each of which is cut into **two pieces** such that:
  - for k_1 the piece x_0 is converted to **200**, while x_1 is converted to **50**
  - for k_2 the piece x_0 is converted to **50**, while x_1 is converted to **200**

  Both keys will be converted to the same hash code, **250**, leading to a **collision**
- It ignores the order of the pieces, which could lead to collisions!

POLYNOMIAL HASH CODE

- Instead of summation, let's use a **polynomial** hash code, which works as follows:
  - **Cut** k into multiple pieces, $x_0, x_1, \ldots, x_{m-1}$ that each takes ≤ 32 bits
  - **Cast** each piece into `int`
  - **Set** a constant a
  - **Compute**: $x_0 a^{m-1} + x_1 a^{m-2} + \cdots + x_{m-3} a^2 + x_{m-2} a + x_{m-1}$
- This is better than summation because **it considers the order of the pieces**,

Given k_1 and k_2 each cut into **two pieces** such that:

- for k_1 the piece x_0 is converted to **200**, while x_1 is converted to **50**
- for k_2 the piece x_0 is converted to **50**, while x_1 is converted to **200**

Then, if we set a = 3:

- The hash code of k_1 would be: $3 \times \mathbf{200} + \mathbf{50}$ = 650
- The hash code of k_1 would be: $3 \times \mathbf{50} + \mathbf{200}$ = 350