COMP225: Algorithms and Data Structures

Hashing and Maps

Mark Dras
Mark.Dras@mq.edu.au
E6A380

Reading

- Drozdek, Chapter 10

Hashing and Maps

- Hash Functions
- Collision Resolution
- Hashing and Maps in `java.util`

Abstract Data Type Table

- Very common
- Have a key and an associated value
  - StudentID Name Degree ...
  - Employee Name Address Grade …
- A collection of these is an ADT Table
- Operations:
  - insert(key, value)
  - retrieve(key)
  - delete(key)

Table Example

40112377	Jane Smith
40125666	Fred Wu
44123456	Sue Bell
44444444	Sam Soong

Accessing Tables

key	value
0	
1	
2	
3	(3, Fred Wu, 21)
4	
5	
6	
7	
8	

- Ideally: Use key as index to array e.g. employee #3 info stored at array[3]
- Insert, delete and retrieve would be O(1)
- However, not feasible for keys in general
  - Consider your student numbers
  - Would waste 40000000 locations

Accessing Tables

- For student number case: an obvious first fix is to subtract 40000000 from StudentId before using as index into array
- In general need to transform key to a valid index in array
- Hashing is any transformation of a key to an index (or address)
- Hash function

  Index = hash(key)

Hashing

- To find a function (*h*) that can transform a particular key (*K*) (a string, number or record) into an index in the table used for storing items of the same type as *K,* the function *h* is called a **hash function**
- If *h* transforms different keys into different numbers, it is called a **perfect hash function**

Hash Functions: Division

- The **division** method is the preferred choice for the hash function if very little is known about the keys

  TSize =*sizeof*(*table*), as in $h(K) = K \bmod Tsize$

- Example:
  - Assume a table of size 20 (i.e. with indices 0..19), and value 36 to be inserted.
  - Insert at h(36) = 16.

Hash Functions: Division

- Can be extended to non-numeric keys: use hash function that converts to numeric.

- Example:
  - For string s and table size 20, let h(K) = sum of ASCII values of letters mod TSize
  - h(“ape”) = (97 + 112 + 101) mod 20 = 10

Hash Functions: Folding

- In the **folding** method, the key is divided into several parts which are combined or folded together and are often transformed in a certain way to create the target address

- Example:
  - Assume a table of size 1000 and a social security number 123-45-6789.
  - Can divide into three parts 123, 456, 789; add these three parts together, and then take mod 1000.

Hash Functions: Mid-square

- In the **mid-square** method, the key is *squared* and the middle or *mid* part of the result is used as the address

- Example:
  - Assume a table of size 1000, and a value 3121 to be inserted.
  - Calculate 3121^2 = 97**406**41, and take middle
  - Then h(3121) = 406

Hash Functions: Extraction

- In the **extraction** method, only a part of the key is used to compute the address

- Example:
  - Assume table of size 1000, social security number 123-45-6789.
  - Take first and last two.
  - Then h(123-45-6789) = 189.

Hash Functions: Radix

- Using the **radix transformation**, the key *K* is transformed into another number base; *K* is expressed in a numerical system using a different radix

- Example:
  - Assume table of size 100, insert value 345.
  - Transform into base 9: 423.
  - Then take mod 100 to fit to table.
  - h(345) = 23.

Hashing and Maps

- Hash Functions
- Collision Resolution
- Hashing and Maps in `java.util`

Collision Resolution

- In the **open addressing** method, when a key collides with another key, the collision is resolved by finding an available table entry other than the position (address) to which the colliding key is originally hashed
- The simplest method is **linear probing**, for which $p(i) = i$, and for the ith probe, the position to be tried is $(h(K) + i) \bmod TSize$

Linear Probing

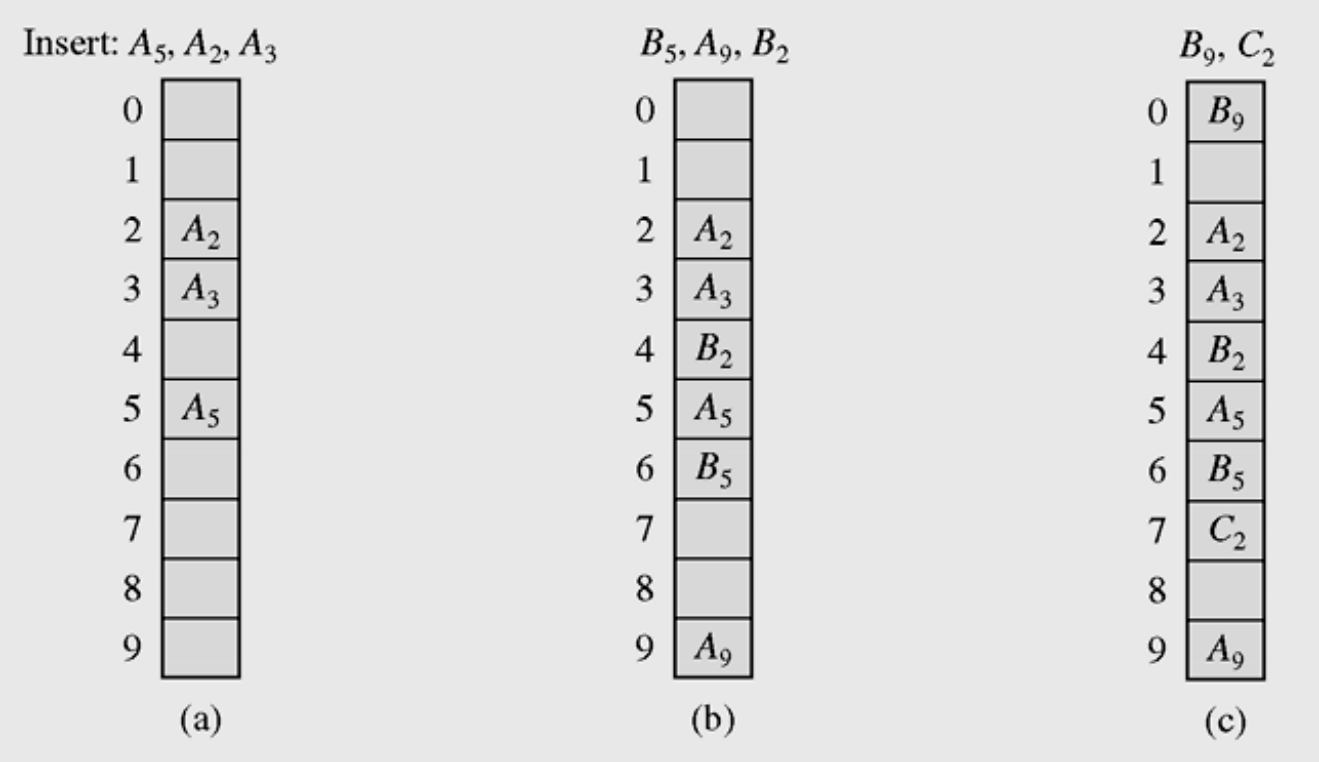


Figure 10-1 Resolving collisions with the linear probing method. Subscripts indicate the home positions of the keys being hashed. (p(i) = i, or step size = 1)

Collision Resolution: Exercise

- Assume hash function h(K) = K mod 11, linear probing with step size 1, and the following existing table

0	1	2	3	4	5	6	7	8	9	10
		13		26	5	37				21

- Insert the values 16, 15, 32.

Java Code

```
static int a[];
static int N;

HashLinear(int n)
{
  N = n;
  a = new int[N];
  for (int i = 0; i < N; i++)
    a[i] = 0;
}

int find (int key)
// returns index of key if found, -1 otherwise
{
  int pos = key % N ; // hash the key
  while (a[pos] != key && a[pos] != 0)
    pos = (pos + 1) % N; // wrap around
  if (a[pos] == key)
    return pos;
  else
    return -1; // not there
}
```

Deletion

- Can't just delete
  - Consider deleting 37 from table in exercise
- Need to fill slot with some marker (e.g. if values can only be positive integers, use -1)
  - Retrieve should skip over this
  - Insert should fill in at this location
- Note that retrieve will therefore have to be modified

Deletion Example

- Assume hashing as in previous exercise

0	1	2	3	4	5	6	7	8	9	10
		13		26	5	37				21

- Delete 5

0	1	2	3	4	5	6	7	8	9	10
		13		26	DEL	37				21

- Retrieve 37? (OK)
- Insert 16

0	1	2	3	4	5	6	7	8	9	10
		13		26	16	37				21

Efficiency of Linear Probing

- Analysis of **average case** efficiency depends on how full the table is
- Let N = number of filled slots
- Load Factor = N / TSize
- As the table fills:
  - Load factor increases
  - Chance of collision increases
  - Search time increases
  - Efficiency decreases

Efficiency of Linear Probing

- Note that insertion, deletion and retrieval all terminate with an empty slot
- If empty slots are spread throughout table, hash function is sensible, and table is <2/3 full (roughly) then
  - average case complexity of all operations is O(1)
- Worst case complexity is O(N)
  - when table is (almost) full
  - when else?

Aside: Rehashing

- Keeping the load factor below a certain threshold is vital for linear probing
- If the load factor goes above a certain threshold, the table can be resized
- New load factor is then below threshold
- Objects are inserted into new, resized table (using a new hash function)
- Resizing should increase table size geometrically rather than arithmetically; doubling is reasonable
- Analysis of efficiency requires amortised complexity (beyond the scope of this unit)
  - It turns out this is O(1): we no longer get the worst-case full table behaviour

Clustering in Linear Probing

- Empty cells following clusters have a much greater chance to be filled than other positions.
  - Probability is (sizeof(cluster)+1) / TSize
  - Other cells only 1 / TSize
- “Rich get richer” behaviour of clusters
  - Bad, because tends towards linear search
- Avoid cluster build-up by choosing different probing function

Quadratic Probing

- Uses probing function

  $p(i) = h(K) + (-1)^{i-1} ((i+1)/2)^2$

  for i = 1, 2, …, TSize – 1

- Can be written as the sequence

  $h(K) + i^2, h(K) - i^2$ for i = 1, 2, …, TSize – 1

- Including the first attempt to hash K, this results in the sequence

  h(K), h(K)+1, h(K)-1, h(K)+4, h(K)-4, …, h(K)+ $(TSize-1)^2/4$, h(K)-$(TSize-1)^2/4$ (all mod TSize)

Quadratic Probing

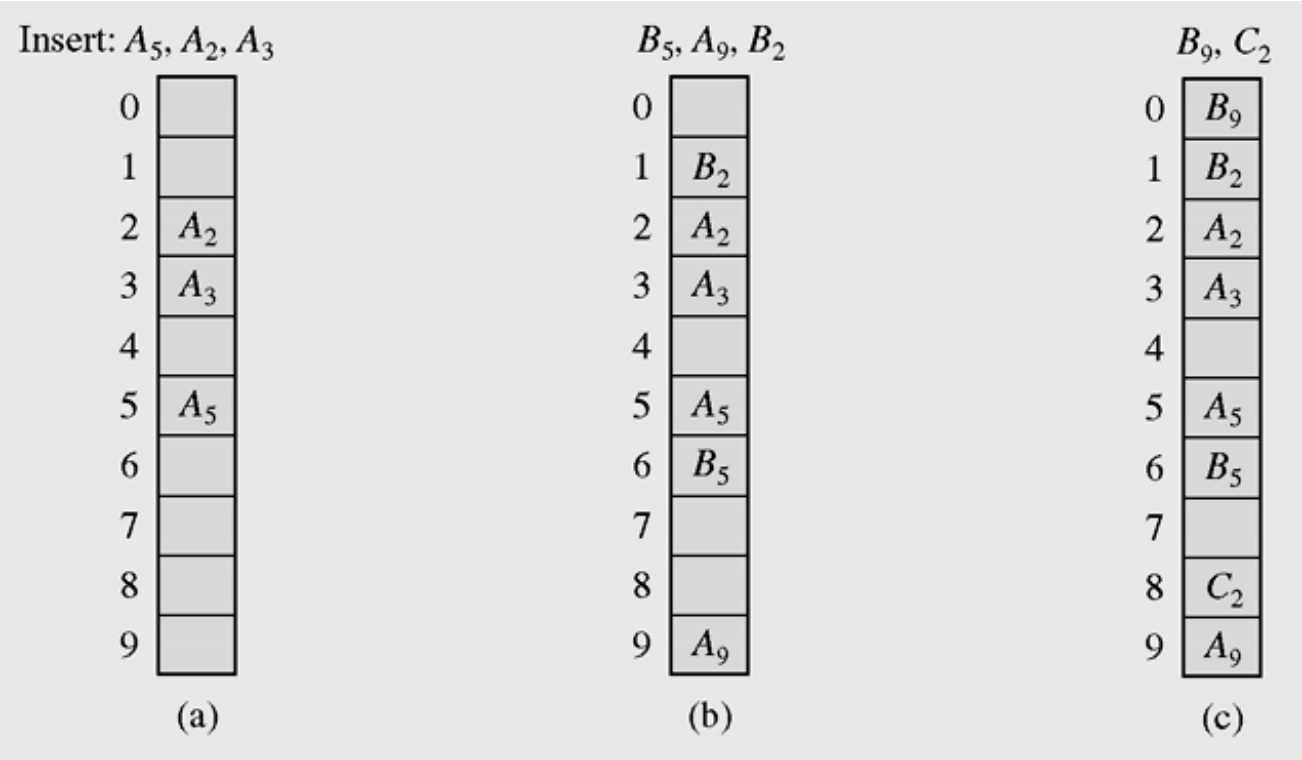


Figure 10-2 Using quadratic probing for collision resolution

Collision Resolution: Exercise

- Assume hash function h(K) = K mod 11, **quadratic** probing, and the following existing table

0	1	2	3	4	5	6	7	8	9	10
		13		26	5	37				21

- Insert the values 16, 15, 32.

Quadratic Probing Issues

- Table size should not be an even number
- Ideally, size should be a prime of the form 4j+3, for some integer j
  - Guarantees inclusion of all positions in the probing sequence

Double Hashing

- Linear probing leads to large clusters
- Quadratic probing leaves some spaces, but leads to secondary clusters
  - When there's a collision, every key will try the same next spot
- An alternative is to use a second hash function h_p when there's a collision

Double Hashing

- Try probing with multiples of h_p
  - $h(K), h(K)+h_p(K), h(K)+2h_p(K), ..., h(K)+i.h_p(K)$
- (Take all of these mod TSize)
- A common type of second hash function is based on mod M, for some M < TSize

Collision Resolution: Exercise

- Assume hash function h(K) = K mod 11, second hash function h_p(K)= K mod 9, and the following existing table

0	1	2	3	4	5	6	7	8	9	10
		13		26	5					21

- Insert the values 37,16, 15, 32.
- Is this a good h_p?

Double Hashing

- Pros
  - No primary and no secondary clustering
- Cons
  - An extra hash calculation
  - Delete cumbersome to implement

Double Hashing

- A possible efficiency: define h_p in terms of h
  - e.g. $h_p(K) = i.h(K) + 1$
- Is this useful? Yes
  - Consider some K1 such that h(K1) = j
  - Hash sequence is then j, 2j+1, 5j+2, ...
  - Consider some other K2 such that h(K2) = 2j+1
  - Hash sequence is then 2j+1, 4j+3, 10j+11, ...

Collision Resolution Comparison

	Linear Probing	Quadratic Probing[a]	Double Hashing
successful search	$\frac{1}{2}\left(1+\frac{1}{1-LF}\right)$	$1-\ln(1-LF)-\frac{LF}{2}$	$\frac{1}{LF}\ln\frac{1}{1-LF}$
unsuccessful search	$\frac{1}{2}\left(1+\frac{1}{(1-LF)^2}\right)$	$\frac{1}{1-LF}-LF-\ln(1-LF)$	$\frac{1}{1-LF}$

Load Factor $LF = \frac{\text{number of elements in the table}}{\text{table size}}$

[a] The formulas given in this column approximate any open addressing method that causes secondary clusters to arise, and quadratic probing is only one of them.

Figure 10-3 Formulas approximating, for different hashing methods, the average numbers of trials for successful and unsuccessful searches (Knuth, 1998)

Collision Resolution Comparison

	Linear		Quadratic		Double	
LF	success	failure	success	failure	success	failure
0.1	1.1	1.1	1.1	1.1	1.1	1.1
0.2	1.1	1.3	1.1	1.3	1.1	1.2
0.3	1.2	1.5	1.2	1.5	1.2	1.4
0.4	1.3	1.9	1.3	1.8	1.3	1.7
0.5	1.5	2.5	1.4	2.2	1.4	2.0
0.6	1.8	3.6	1.6	2.8	1.5	2.5
0.7	2.2	6.1	1.9	3.8	1.7	3.3
0.8	3.0	13.0	2.2	5.8	2.0	5.0
0.9	5.5	50.5	3.5	22.0	2.6	10.0

Chaining

- In **chaining**, each position of the table is associated with a linked list or **chain** of structures whose `info` fields store keys or references to keys
- This method is called **separate chaining**, and a table of references (pointers) is called a **scatter table**

Chaining (continued)

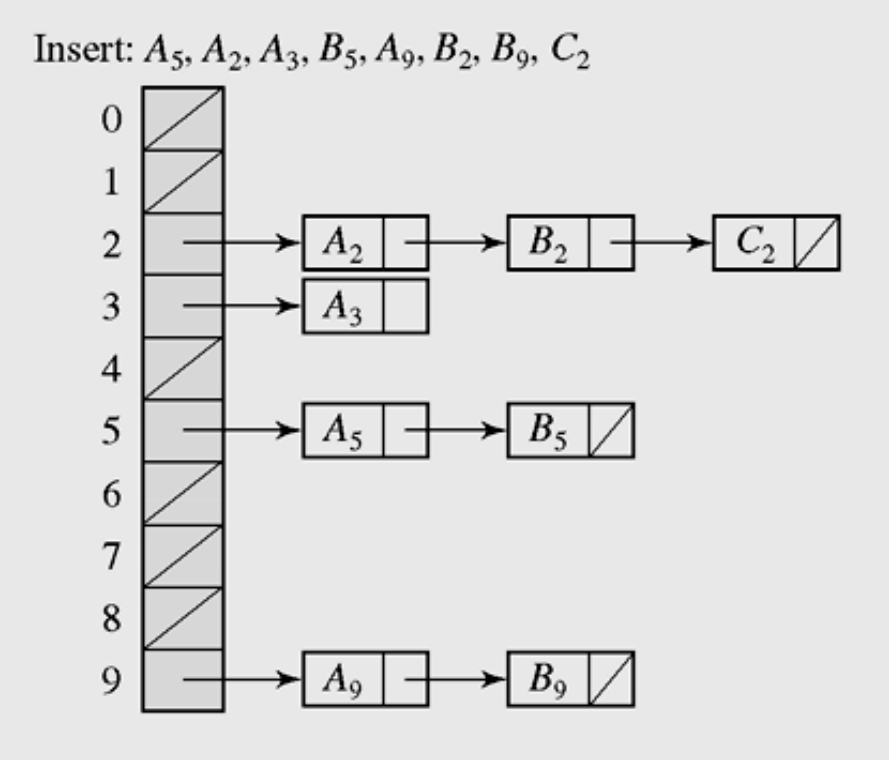


Figure 10-5 In chaining, colliding keys are put on the same linked list

Chaining (continued)

- A version of chaining called **coalesced hashing** (or **coalesced chaining**) combines linear probing with chaining
- An overflow area known as a **cellar** can be allocated to store keys for which there is no room in the table

Chaining (continued)

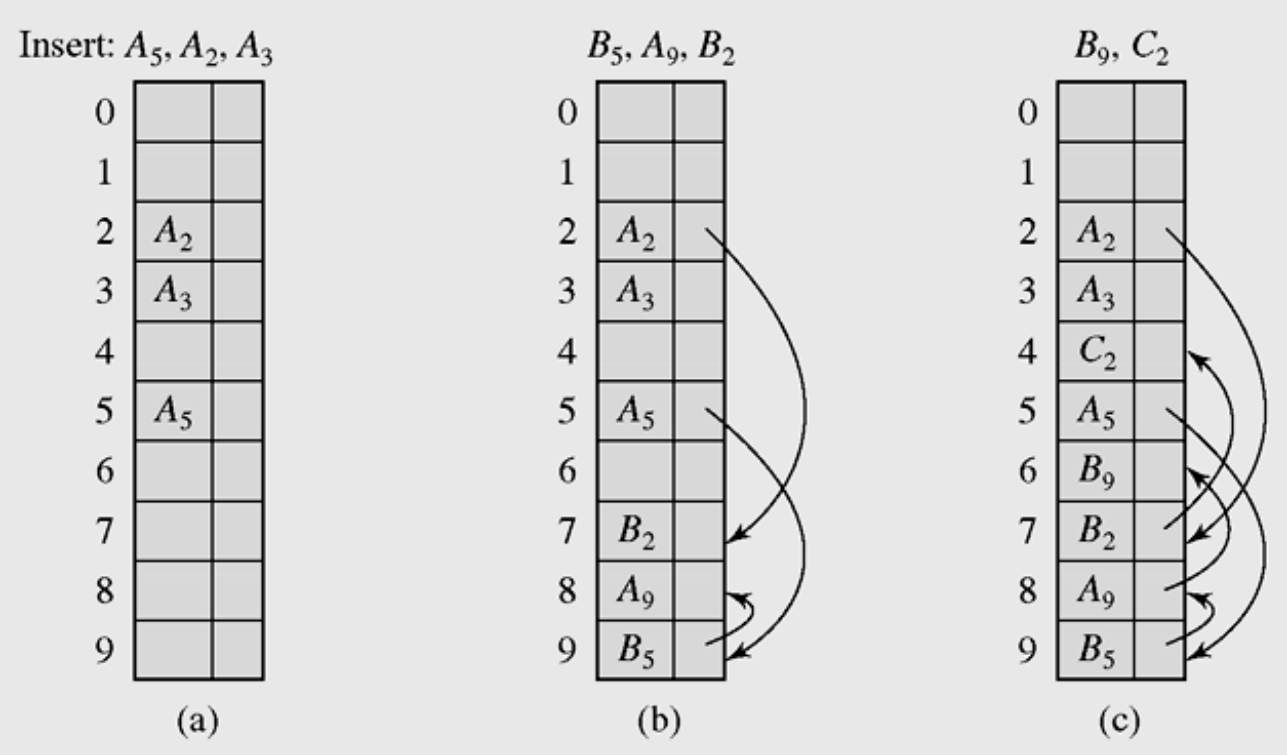


Figure 10-6 Coalesced hashing puts a colliding key in the last available position of the table

Chaining (continued)

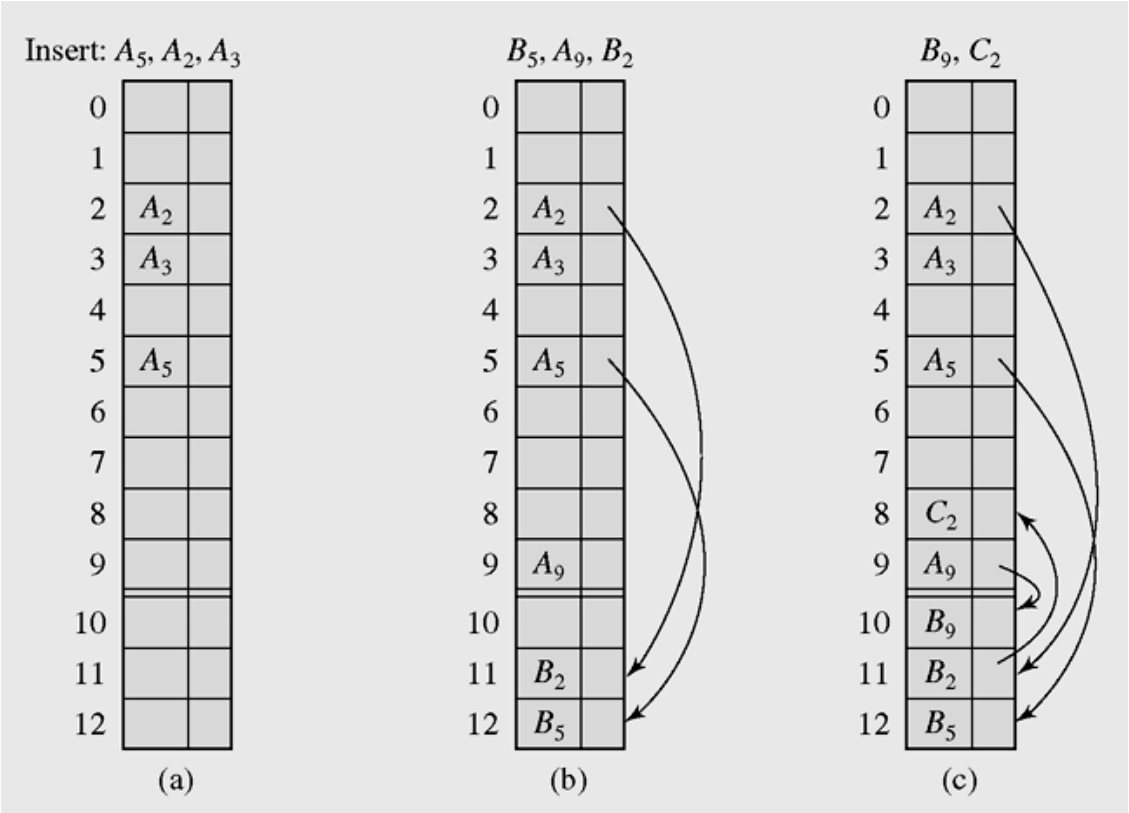


Figure 10-7 Coalesced hashing that uses a cellar

Chaining Issues

- Requires additional space for maintaining references
  - Table stores only references
  - For n keys, n + TSize references are needed
- Complexity
  - Insertion worst case is O(1)
  - Retrieval worst case is O(n) (when?)
- Increasingly long lists degrade performance
  - Can help this by ordering lists
  - Priority queue!

Bucket Addressing

- To store colliding elements in the same position in the table can be achieved by associating a bucket with each address
- A **bucket** is a block of space large enough to store multiple items

Bucket Addressing (continued)

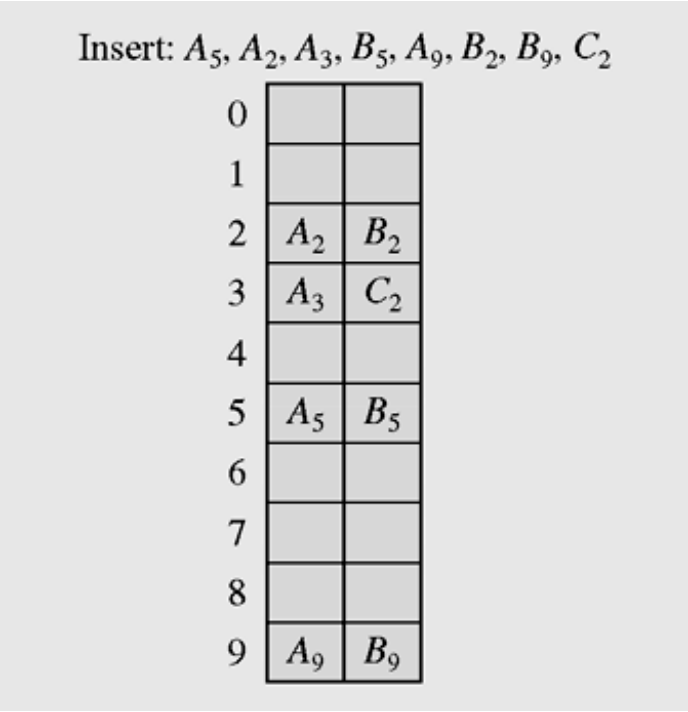


Figure 10-8 Collision resolution with buckets and linear probing method

Bucket Addressing (continued)

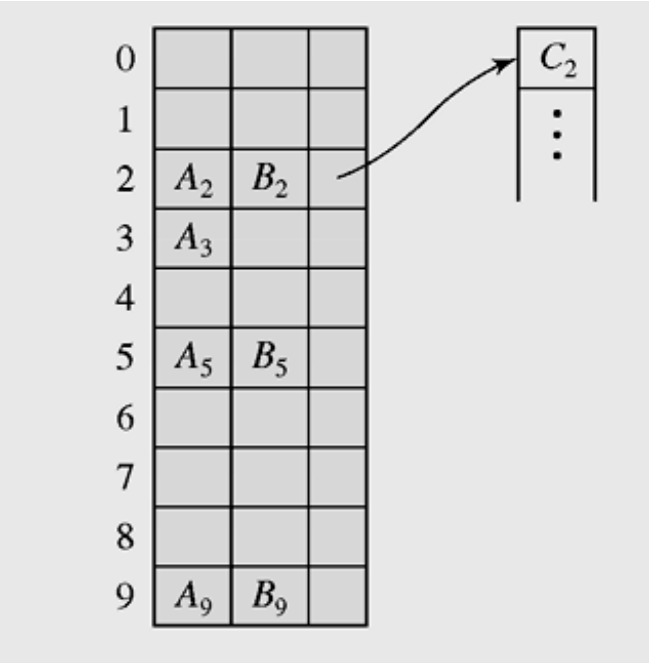


Figure 10-9 Collision resolution with buckets and overflow area

Hashing and Maps

- Hash Functions
- Collision Resolution
- Hashing and Maps in `java.util`

Maps

- Java 8's version of Table is

  ```
  public interface Map<K,V>
  ```

  where

  `K` - the type of keys maintained by this map

  `V` - the type of mapped values
- A Map is an object that maps keys to values
- The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.

Map Methods

Method Summary	
void	**clear**() Removes all of the mappings from this map (optional operation).
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet**() Returns a Set view of the mappings contained in this map.
boolean	**equals**(Object o) Compares the specified object with this map for equality.
V	**get**(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	**hashCode**() Returns the hash code value for this map.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	**putAll**(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map (optional operation).
V	**remove**(Object key) Removes the mapping for a key from this map if it is present (optional operation).
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a Collection view of the values contained in this map.

`HashMap`

- `HashMap` is an implementation of the interface `Map` that uses hashing to map keys to values
- A **hash map** is a collection of singly linked lists (buckets); that is, chaining is used as a collision resolution technique
- In a hash map, both null values and null keys are permitted

HashMap

```
package hashing.student;

import java.util.HashMap;

class Person {
  private String name;
  int age;
  private int hashcode = 0;
  public Person(String n, int a) {
    name = n;
    age = a;
    for (int i = 0; i < name.length(); i++)
      hashcode += name.charAt(i);
  }
  public Person() {
    this("",0);
  }
  public boolean equals(Person p) {
    return name.equals(p.name);
  }
  public int hashCode() {
    return hashcode;
  }
  public String toString() {
    return "(" + name + "," + age + ")";
  }
}
```

HashMap

```
class TestHashMap {
    public static void main(String[] a) {
        HashMap cities = new HashMap();
        cities.put(new Person("Gregg",25),"Pittsburgh");
        cities.put(new Person("Ann",30),"Boston");
        cities.put(new Person("Bill",20),"Belmont");
        System.out.println(cities);
        // {(Ann,30)=Boston, (Gregg,25)=Pittsburgh, (Bill,20)=Belmont}
        cities.put(new Person("Gregg",30),"Austin");
        System.out.println(cities);
        // {(Ann,30)=Boston, (Gregg,25)=Austin, (Bill,20)=Belmont}
        System.out.println(cities.containsKey(new Person("Ann",30)));
        // true
        System.out.println(cities.containsValue("Boston"));
        // true
        System.out.println(cities.size());                    // 3
```

Figure 10-17 Demonstrating the operation of the methods in class `HashMap` (continued)

HashMap (continued)

```
        System.out.println(cities.get(new Person("Ann",30)));   // Boston
        System.out.println(cities.entrySet());
        // [(Ann,30)=Boston, (Gregg,25)=Austin, (Bill,20)=Belmont]
        System.out.println(cities.values());
        // [Boston, Austin, Belmont]
        System.out.println(cities.keySet());
        // [(Ann,30), (Gregg,25), (Bill,20)]
        System.out.println(cities.remove(new Person("Bill",20)));
        // Belmont
        System.out.println(cities);
        // [(Ann,30), (Gregg,25)]
        cities.put(null,"Nashville");
        cities.put(new Person("Kay",44),null);
        System.out.println(cities);
        // [(Ann,30)=Boston, (Gregg,25)=Austin, (Kay,44)=null,
        // null=Nashville]
        System.out.println(cities.get(new Person("Kay",44)));
        // null
        System.out.println(cities.get(new Person("Stan",55)));
        // null
        System.out.println(cities.containsKey(new Person("Kay",44)));
        // true
        System.out.println(cities.containsKey(new Person("Stan",55)));
        // false
    }
}
```

Figure 10-17 Demonstrating the operation of the methods in class `HashMap` (continued)

HashMap Example

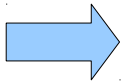
```
class Person2 {
  private String name;
  int age;
  private int hashcode = 0;
  public Person2(String n, int a) {
    name = n;
    age = a;
//  for (int i = 0; i < name.length(); i++)
//    hashcode += name.charAt(i);
    hashcode = age;
  }
  public Person2() {
    this("",0);
  }
  public boolean equals(Object p) {
//  return name.equals(((Person2)p).name);
    return (age == ((Person2)p).age);
  }
  public int hashCode() {
    return hashcode;
  }
}
```

Aside: Iterators

- Objects that allow you to go over all the elements of a collection in sequence

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

public class TestIterator {

public static void main(String[] argv) {

  String words[] = {"in", "the", "second", "century", "of", "the",
                    "christian",  "era", "the", "empire", "of", "rome",
                    "comprehended", "the", "fairest", "part", "of",
                    "the", "earth"};

  // convert array to list
  List<String> lList = Arrays.asList(words);
```

Aside: Iterators



```
        // for loop
        System.out.println("#1 for");
        for (int i = 0; i < lList.size(); i++) {
          System.out.println(lList.get(i));
        }

        // iterator loop
        System.out.println("#2 iterator");
        Iterator<String> iterator = lList.iterator();
        while (iterator.hasNext()) {
          System.out.println(iterator.next());
        }

        // for loop advance
        System.out.println("#3 for advance");
        for (String temp : lList) {
           System.out.println(temp);
        }
    }
}
```

HashMaps: Another Example

- Can have more complex objects as values, including collections
  - Following example: replicating Dict3 from last week
- May want to iterate over elements
  - Use iterator

HashMaps: Another Example

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Vector;

class CharListMap {
  private Map<Character, List<String> > m;
  CharListMap() {
    m = new HashMap<Character, List<String>>();
  }
  public void add(String s) {
    char key = s.charAt(0);
    if (m.containsKey(key)) {
      m.get(key).add(s);
    }
    else {
      List<String> l = new Vector<String>();
      l.add(s);
      m.put(key,  l);

    }
  }
```

HashMaps: Another Example

```
public void print() {
  Iterator<Map.Entry<Character, List<String>>> it =
m.entrySet().iterator();
  System.out.println("--CharListMap--");
  while (it.hasNext()) {
    Map.Entry<Character, List<String> > pairs = it.next();
    System.out.println(pairs.getKey() + " : " + pairs.getValue());
  }
}
```

- Caveats:
  - Order is arbitrary
  - Can use alternatives (e.g. looping through entrySet())
  - Don't use iterators, or these alternatives, to search!

TreeMaps

- A TreeMap is a Map that is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time.
- So you can still access individual elements through e.g. `get()`.
- You can also iterate through ordered keys.

HashMap vs TreeMap

```
import java.util.HashMap;
import java.util.TreeMap;
import java.util.Iterator;
import java.util.Map;

public class HashVTreeMap {
  public static void main(String[] a) {
    HashMap<String, String> hm = new HashMap<String, String>();
    hm.put("donkey", "definition of donkey");
    hm.put("giraffe", "definition of giraffe");
    hm.put("antelope", "definition of antelope");
    hm.put("walrus", "definition of walrus");

    Iterator<Map.Entry<String, String>> hmit = hm.entrySet().iterator();
    System.out.println("--HashMap--");
    while (hmit.hasNext()) {
      Map.Entry<String, String> pairs = hmit.next();
      System.out.println(pairs.getKey() + " : " + pairs.getValue());
    }
```

HashMap vs TreeMap

```
    TreeMap<String, String> tm = new TreeMap<String, String>();
    tm.put("donkey", "definition of donkey");
    tm.put("giraffe", "definition of giraffe");
    tm.put("antelope", "definition of antelope");
    tm.put("walrus", "definition of walrus");

    Iterator<Map.Entry<String, String>> tmit = tm.entrySet().iterator();
    System.out.println("--TreeMap--");
    while (tmit.hasNext()) {
      Map.Entry<String, String> pairs = tmit.next();
      System.out.println(pairs.getKey() + " : " + pairs.getValue());
    }
  }
}
```

HashMap vs TreeMap

- Output:

```
--HashMap--
giraffe : definition of giraffe
donkey : definition of donkey
antelope : definition of antelope
walrus : definition of walrus
--TreeMap--
antelope : definition of antelope
donkey : definition of donkey
giraffe : definition of giraffe
walrus : definition of walrus
```

HashMap vs TreeMap

- Complexity
  - HashMap operations are O(1) (because of rehashing – see Java doc for discussion)
  - TreeMap operations are O(log n)
- Use
  - Use TreeMap if you frequently need an ordered (sub)set of keys
  - Use HashMap otherwise