Data Structures and Algorithms (ECEG 4171)

Chapter Six
Hash Tables

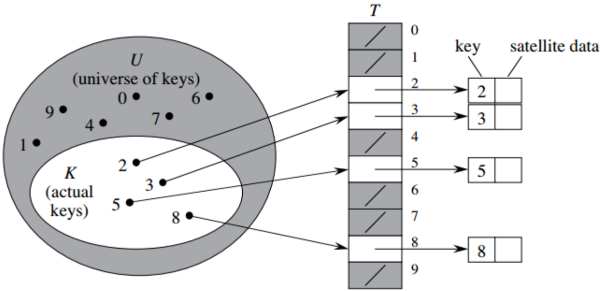
Ephrem A. (M.Sc.)
School of Electrical and Computer Engineeing
Chair of Computer Engineering
11th November, 2019

Introduction

- Hash tables are very efficient data structures for implementing dictionaries.
- With hashing, the average time to search, insert and delete an element is $O(1)$. However, the worst-case time is $O(n)$.
- The **hash table** data structure is merely an array of some fixed size, containing the items.
- Generally a search is performed on some part of the item called the **key**.
- Each key is mapped into some number in the range 0 to $M - 1$ and placed in the appropriate cell.
- The mapping is called a **hash function**, which ideally should be simple to compute and should ensure that any two distinct keys get different cells.

Direct-address tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.
- Scenario
  - Maintain a dynamic set.
  - Each element has a key drawn from a universe $U = 0, 1, \ldots, m - 1$ where m isn't too large.
  - No two elements have the same key.
- Represent by a direct-address table, or array, $T[0 \ldots m - 1]$:
  - Each slot, or position, corresponds to a key in U
  - If there's an element x with key k, then $T[k]$ contains a pointer to x.
  - Otherwise, $T[k]$ is empty, represented by NIL (or null in Java).



- Dictionary operations are trivial and take $O(1)$ time each:
- To search for key k in table T, return T[k].
- To insert element x into the table, store x at slot x.key (T[x.key] = x).
- To delete element x from the table, change slot x.key value to NIL.

Direct Addressing

- **Advantage**
  1. Very fast: search/insert/delete is $\Theta(1)$
- **Disadvantage**
  1. Universe size has to be small or otherwise, the table size has to be very large! In reality, universe of keys is not small. For example, if keys are 32-bit integers, you need 2^{32} entries; more than 4 billion.
  2. If only few elements are stored, lots of table elements are unused (waste of memory).
  3. Keys are not always nonnegative integers (they can be floats, strings or other user defined types).

Hash Tables

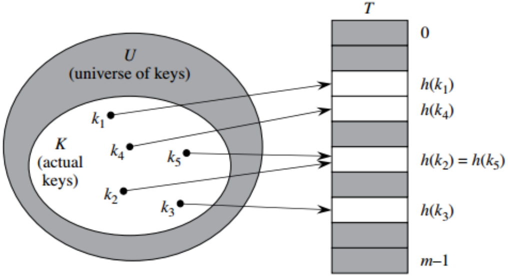
- With direct addressing, an element with key k is stored in slot k.
- With hashing, this element is stored in slot $h(k)$. We call h a hash function.

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

, so that $h(k)$ is a legal slot number in T. We say that k hashes to slot $h(k)$.

- Instead of a size of $|U|$, the array can have size M. Since now $|U| > M$, two or more keys may hash to the same slot. This is knowns as collision.
- Therefore, must be prepared to handle collisions in all cases.
- Use two methods: **chaining** and **open addressing**.
- Avoiding collsion altogether is an impossible task, but can be minimized by choosing a hash function that uniformly and indenpendtly distributes keys across the slots.

An Example of Collision



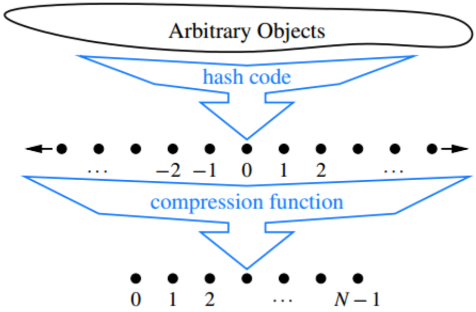
Because k_1 and k_2 map to the same slot, they collide.

How to Choose a Good Hash Function

- A good hash function is a function that satisfies the assumption of simple uniform hashing.
  - One that uniformly and indepently distributes keys across slots.
  - One that is easy to compute.
- The first characteristic is hard to achieve because we rarely know the distribution from which keys are drawn.
- Designing a good hash function also depends on the key type.
  - we need a different hash function for each key type that we use.

How to Choose a Good Hash Function

- The evaluation of a hash function can be viewed as consisting of two steps:
  - The **hash code** that maps k to an integer and the **compression function** that maps to the hash code to within the range $\{0, 1, \ldots, M-1\}$.



How to Choose a Good Hash Function

1. Positive integers: most commonly used is modular hashing.

$$h(k) = k\%M$$

   - M must be prime (so that all digits play a role).
   - E.g. if M is 10^k, then only the k least significant digits are used.
   - If M is 2^k, then only the k least significant bits are used.

2. Floating pt: If the keys are real numbers, use modular hashing on the binary representation of the key. That's what Java does through `Float.floatToIntBits(var)`.
3. Strings: treat the string as a huge number in base-R.

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = (R * hash + s.charAt(i)) % M; // charAt(i) returns 16-bit integer.
```

Good choices of R are 31, 33, 37, 39, and 41.

How to Choose a Good Hash Function

4. Compound keys: If the key type has multiple integer fields, mix them together as in `String` values. E.g. Type `Date` with day, month, and year fileds. `int hash = (((day * R + month) % M ) * R + year) % M;`
5. User-defined type: Java provides `hashCode` method which returns a 32-bit integer.
   - By default returns the address of the object in integer.
   - To use `hashCode` with user-defined type, override `hashCode` and `equals` such that:
     - If `a.equals(b)` is true, `a.hashCode()` and `b.hashCode()` must have the same value.
     - If `hashCode()` values are different, `a.equals(b)` is false.
     - If `hashCode()` values are the same, `a.equals(b)` may or may not be true.

How to Choose a Good Hash Function

- The default `equals()` method of the `Object` class checks if two object references `x` and `y` refer to the same object (shallow comparsion).
- When overriding, the `equals` method should implement equivalence relation i.e.
  - reflexive: `x.equals(x)` should return `true`.
  - symmetric: `x.equals(y)` is `true` if and only if `y.equals(x)` is true.
  - transitive: If `x.equals(y)` is `true` and `y.equals(z)` is `true`, then `x.equals(z)` should return `true`.
  - consistent: If two objects are not equal they should remain unequal as long as they are not modified.
  - null comparison: For `x` not `null`, `x.equals(null)` should return `false`.

hashCode and equals for Transaction

```
public class Transaction {
    private String who;
    private Date when;
    private double amount;
    ....
    public int hashCode(){
        int hash = 17;
        hash = 31*hash + who.hashCode();
        hash = 31*hash + when.hashCode();
        hash = 31*hash + ((Double) (amount)).hashCode();
        return hash;
    }
    public boolean equals(Object obj){
        if(this == obj) return true;
        if(obj == null || this.getClass() != obj.getClass()) return false;
        Transaction trans = (Transaction) obj;
        boolean strComp = (who == trans.who || (who != null && who.equals(trans.who)));
        boolean dateComp = (when == trans.when || (when != null && when.equals(trans.when)));
        return strComp && dateComp && amount == trans.amount;
    }
}
```

How to Choose a Good Hash Function

- After properly defining the `hashCode` method, hash function can be defined as follows:

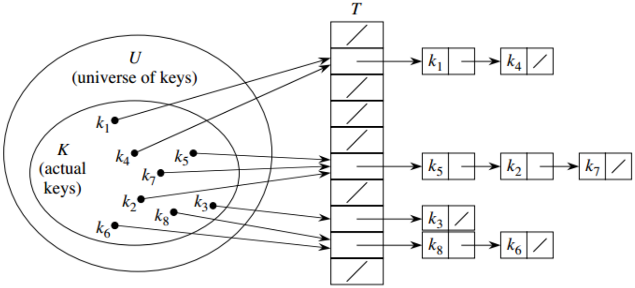
```
private int hash (Key x){
   return (x.hashCode () & 0 x7fffffff ) % M;
}
```

  The code masks off the sign bit: to turn the 32-bit number into a 31-bit nonnegative integer.

- In summary, a good hash function should have the following characteristics:
  - It should be consistent (equal keys must produce the same hash value).
  - It should be efficient to compute.
  - It should uniformly distribute the keys.

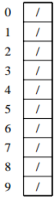
Collision resolution by Chaining

- Put all elements that hash to the same slot into a linked list.



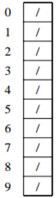
- This Figure shows **singly linked lists**. If we want to delete elements, it's better to use **doubly linked lists**.
- Slot j contains a pointer to the head of the list of all stored elements that hash to j.
- If there are no such elements, slot j contains NIL.

Example

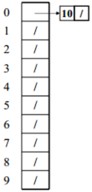


Insert 10, 22, 107, 12, 42 with M = 10, and $h(k) = k \,\%\, M$.

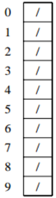
Example



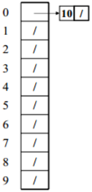
Insert 10, 22, 107, 12, 42 with M = 10, and $h(k) = k \,\%\, M$.

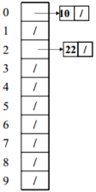


Example

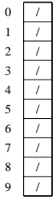


Insert 10, 22, 107, 12, 42 with M = 10, and $h(k) = k \,\%\, M$.

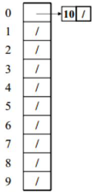


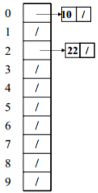


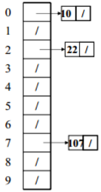
Example



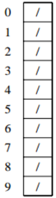
Insert 10, 22, 107, 12, 42 with M = 10, and $h(k) = k \% M$.



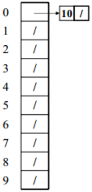


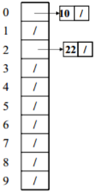


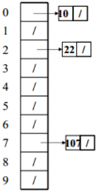
Example

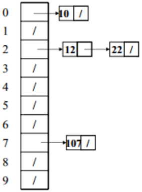


Insert 10, 22, 107, 12, 42 with M = 10, and $h(k) = k \,\%\, M$.

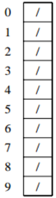




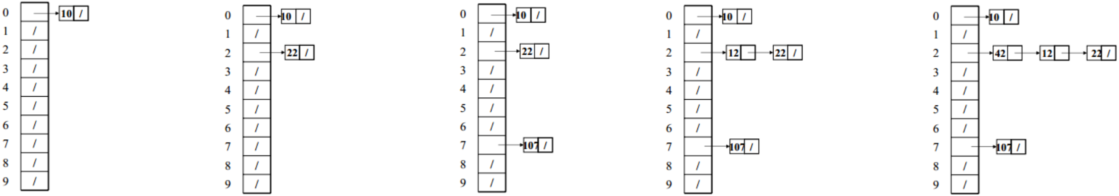




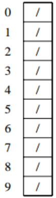
Example



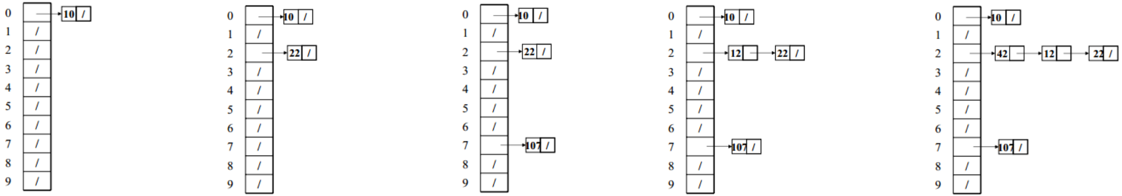
Insert 10, 22, 107, 12, 42 with M = 10, and $h(k) = k \,\%\, M$.



Example



Insert 10, 22, 107, 12, 42 with M = 10, and $h(k) = k \,\%\, M$.



Implementing a Hash Table with Separate Chaining

```java
public class SeparateChainingHashTable<K, V> {
    private int M = 97;
    private int N;
    private Node[] table = new Node[M];
    private static class Node{
        private Object key;
        private Object val;
        private Node next;
        public Node(Object key, Object val, Node next){
            this.key = key;
            this.val = val;
            this.next = next;
        }
    }
    private int hash(K key){
        return (key.hashCode() & 0x7fffffff) % M;
    }
    public V get(K key){
        int i = hash(key);
        for(Node x = table[i]; x != null; x = x.next)
            if(key.equals(x.key))
                return (V) x.val;
        return null;
    }
    public void put(K key, V val){
        int i = hash(key);
        for(Node x = table[i]; x != null; x = x.next){
            if(key.equals(x.key)){
                x.val = val;
                return;
            }
        }
        table[i] = new Node(key, val, table[i]);
        N++;
    }
    public void delete(K key){
        int i = hash(key);
        Node temp = table[i];
        if(temp == null) return; // Slot i is empty
        if(temp.next == null && temp.key.equals(key)){ // One
              item stored at slot i
            table[i] = null;
            N--;
            return;
        }
        while(!temp.next.key.equals(key))
            temp = temp.next;
        Node n = temp.next.next;
        temp.next.next = null;
        temp.next = n;
        N--;
    }
}
```

Analysis of hashing with chaining

- Given a key, how long does it take to find an element with that key, or to determine that there is no element with that key?
- Analysis is in terms of the load factor $\alpha = n/m$:
  - n = # elements in the table.
  - m = # of slots in the table = # of (possibly empty) linked lists.
  - Load factor is average number of elements per linked list.
  - Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$.
- Worst case is when all n keys hash to the same slot

  $\Rightarrow$ get a single list of length n

  $\Rightarrow$ worst-case time to search is $\Theta(n)$, plus time to compute hash function.
- Average case depends on how well the hash function distributes the keys among the slots.

Average-case performance of hashing with chaining

- Assume *simple uniform hashing:* any given element is equally likely to hash into any of the m slots.
- For $j = 0, 1, \ldots, m-1$, denote the length of list $T[j]$ by n_j. Then $n = n_0 + n_1 + \cdots + n_{m-1}$.
- Average value of n_j is $E[n_j] = \alpha = n/m$.
- Assume that we can compute the hash function in $O(1)$ time, so that the time required to search for the element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.
- We consider two cases:
  - If the hash table contains no element with key k, then the search is unsuccessful.
  - If the hash table does contain an element with key k, then the search is successful.

Unsuccessful search:

- An unsuccessful search takes expected time $\Theta(1+\alpha)$.

Proof:

- Simple uniform hashing $\Rightarrow$ any key not already in the table is equally likely to hash to any of the m slots.
- To search unsuccessfully for any key k, need to search to the end of the list $T[h(k)]$.
- This list has expected length $E[n_{h(k)}] = \alpha$. Therefore, the expected number of elements examined in an unsuccessful search is α.
- Adding in the time to compute the hash function, the total time required is $\Theta(1+\alpha)$.

Successful search:

- The expected time for a successful search is also $\Theta(1+\alpha)$.
- Note that the list that is being searched contains the one node that stores the match plus zero or more other nodes. The expected number of “other nodes” in a table of n elements and m lists is $(n-1)/m = \alpha - 1/m \approx \alpha$, since m is presumed to be large.
- On average, half the “other nodes” are searched, so combined with the matching node, we obtain an average search cost of $1+\alpha/2$ nodes.
- Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(1+1+\alpha) = \Theta(1+\alpha)$

Open addressing

An alternative to chaining for handling collisions.

Idea:

- Store all keys in the hash table itself.
- Each slot contains either a key or NIL.
- To search for key k:
  - Compute $h(k)$ and examine slot $h(k)$. Examining a slot is known as a **probe**.
  - There's a third possibility: slot $h(k)$ contains a key that is not k. We compute the index of some other slot, based on k and on which probe (count from 0: $0th$, $1st$, $2nd$, etc.) we're on.
  - Keep probing until we either find key k (successful search) or we find a slot holding NIL (unsuccessful search).
- In general, we have some probe function f and use

$$(h(key) + f(i)) \,\%\, m$$

How to compute probe sequences

3 techniques:

1. Linear probing
2. Quadratic probing
3. Double probing

Linear probing: Given auxiliary hash function h', the probe sequence starts at slot h'(k) and continues sequentially through the table, wrapping after slot $m - 1$ to slot 0.

Give key k and probe number $0 \leq i < m$,

$$h(k, i) = (h'(k) + i) \bmod m$$

.

Linear probing suffers from *primary clustering:* long runs of occupied sequences build up.

Example

Insert 38, 19, 8, 109, 10: $h(key) = key \,\%\, m$.

Idea: If $h(key)$ is already full,

- try (h(key) + 1) % m. If full,
- try (h(key) + 2) % m. If full,
- try (h(key) + 3) % m. If full, . . .

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	/

Example

Insert 38, 19, 8, 109, 10: $h(key) = key \,\%\, m$.

Idea: If $h(key)$ is already full,

- try (h(key) + 1) % m. If full,
- try (h(key) + 2) % m. If full,
- try (h(key) + 3) % m. If full, . . .

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	/

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

0	8
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

0	8
1	109
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

0	8
1	109
2	10
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Primary Clusters

- Linear probing are good because the probe function is quick to compute.
- However, in general they are a bad idea beacause they tend to produce clusters, which lead to long probing sequences, called primary clusters.
- Since all table positions are equally likely to be the hash value of the next key to be inserted, long clusters are more likely to increase in length than short ones.

Linear Probing Implementation

```java
import java.util.LinkedList;
import java.util.List;
public class LinearProbingHashTable<K, V>{
    private static final int INIT_CAPACITY = 97;
    private int N; // number of key-value pairs in the table
    private int M; // size of linear-probing table
    private K[] keys; // the keys
    private V[] vals; // the values

    // create an empty hash table - use 16 as default size
    public LinearProbingHashTable() {
        this(INIT_CAPACITY);
    }

    // create linear proving hash table of given capacity
    public LinearProbingHashTable(int capacity) {
        M = capacity;
        keys = (K[])  new Object[M];
        vals = (V[]) new Object[M];
    }
    private int hash(K key){
        return (key.hashCode() & 0x7fffffff) % M;
    }
```

```java
    public void put(K key, V val){
        if (N >= M/2) resize(2*M);
        int i;
        for (i = hash(key); keys[i] != null; i = (i + 1) % M){
            if (keys[i].equals(key)){
                vals[i] = val;
                return;
            }
        }
        keys[i] = key;
        vals[i] = val;
        N++;
    }
    public V get(K key){
        for (int i = hash(key); keys[i] != null; i = (i + 1) % M)
            if (keys[i].equals(key))
                return vals[i];
        return null;
    }
    public boolean contains(K key) {
        return get(key) != null;
    }
    // delete the key (and associated value) from the symbol table
    public void delete(K key) // see next section
}
```

Deletion in Linear Probing

- When deleting a key, setting its value to `null` will not work.
- This is because this may prematurely terminate the search for a key that was inserted into the table later.
- To remedy this situation, we need to reinsert into the table all of the keys in the cluster below the deleted key.

```
public void delete(K key) {
    if (!contains(key)) return;
    // find position i of key
    int i = hash(key);
    while (!key.equals(keys[i])) {
        i = (i + 1) % M;
    }
    // delete key and associated value
    keys[i] = null;
    vals[i] = null;
    // rehash all keys in same cluster
    i = (i + 1) % M;
    while (keys[i] != null) {
        // delete keys[i] an vals[i]
        // and reinsert
        K  keyToRehash = keys[i];
        V valToRehash = vals[i];
        keys[i] = null;
        vals[i] = null;
        N--;
        put(keyToRehash, valToRehash);
        i = (i + 1) % M;
    }
    if (N > 0 && N <= M/8) resize(M/2);
}
```

Analysis of Linear Probing

- For any $\alpha < 1$, linear probing will find an empty slot: no infinite loop unless table is full.
- Average number of probes, given α:
  - Unsuccessful search (or insert): $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$
  - Successful search: $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$
- If α approaches 1, the number of probes grows very large.
- But if $\alpha < \frac{1}{2}$, the expected number of probes is between 1.5 and 2.5.

Quadratic probing

- Avoids primary clustering by changing the probe function.
- As in linear probing, the probe sequence starts at $h'(k)$. Unlike linear probing, it jumps around in the table according to a quadratic function of the probe number:

  $$h(i, k) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

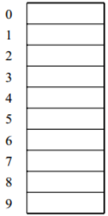
  where $c_1, c_2 \neq 0$ are constants.
- Can suffer from *secondary clustering:* if two keys have the same h' value, then they have the same probe sequence.

Example

Insert 89, 18, 49, 58, 79: $h(key) = key\%m$, and
$h(key, i) = (h(key) + i^2)\%m$

Idea: If $h(key)$ is already full,

- try (h(key) + 1) % m. If full,
- try (h(key) + 4) % m. If full,
- try (h(key) + 9) % m. If full, . . .



Example

Insert 89, 18, 49, 58, 79: $h(key) = key\%m$, and
$h(key, i) = (h(key) + i^2)\%m$

Idea: If $h(key)$ is already full,

- try (h(key) + 1) % m. If full,
- try (h(key) + 4) % m. If full,
- try (h(key) + 9) % m. If full, . . .

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

0		0		0	49	0	49	0	49
1		1		1		1		1	
2		2		2		2	58	2	58
3		3		3		3		3	79
4		4		4		4		4	
5		5		5		5		5	
6		6		6		6		6	
7		7		7		7		7	
8		8	18	8	18	8	18	8	18
9	89	9	89	9	89	9	89	9	89

Quadratic Probing Performance

- In quadratic probing, we start cycling through the same indices after probing m (table size) probes. This is because for probe number i, key k, and table size m:

$$(k + i^2)\,\%\,m \;=\; (k + (i - m)^2)\,\%\,m$$

- However, if table size m is prime and $\alpha < 1/2$, then the quadratic probing will find an empty slot in at most $m/2$ probes.
- Therefore, if you keep $\alpha < 1/2$, no cycles will be detected.

Example: Insert 76, 40, 48, 5, 55, 47: $h(key, i) = (h(key) + i^2)\%m)$

0	
1	
2	
3	
4	
5	
6	

We can see that, to insert 47, first it probes slot 5, 6, 2, 0 which are already full. Then after, it probes slot 5 again after 7 total probes and keeps probing the same slots indefinitely.

0	48
1	
2	5
3	55
4	
5	40
6	76

Double hashing

Double hashing avoids secondary clustering with a probe function that depends on the key.

Use two auxiliary hash functions, h_1 and h_2. h_1 gives the initial probe $(T[h_1(k)])$, and h_2 gives the remaining probes:

$$h(k, i) = (h_1(k) + ih_2(k))\%m$$

Probe sequences:

- 0^{th} probe: $h_1(k)\%m$
- 1^{st} probe: $(h_1(k) + h_2(k))\%m$
- 2^{nd} probe: $(h_1(k) + 2 * h_2(k))\%m$
- 3^{rd} probe: $(h_1(k) + 3 * h_2(k))\%m$
- ...
- i^{th} probe: $(h_1(k) + i * h_2(k))\%m$

Make sure $h_2(k)$ can't be 0 (E.g. $h_2(k) = m - (k\%m)$)

Analysis of Double Hashing

- The average number of probes given the load factor α
  - Unsuccessful search: $\frac{1}{1-\alpha}$
  - Successful search: $\frac{1}{\alpha}\log_e\left(\frac{1}{1-\alpha}\right)$
- For more on analysis of open-addressing hash tables and their proves see **Introduction to Algorithms** on chapter 11.

Rehashing

- In open addressing, it's possible to maintain constant dictionary operations if we ensure the table is always half full ($\alpha < 1/2$) and choose a good hash function.
- To keep the table half full we can always resize the table size using the following `resize` method:

```java
private void resize(int cap) {
    int size = nextPrime(cap);
    LinearProbingHashTable<K, V> temp = new LinearProbingHashTable<>(size);
    for (int i = 0; i < M; i++) {
        if (keys[i] != null)
            temp.put(keys[i], vals[i]);
    keys = temp.keys;
    vals = temp.vals;
    M    = temp.M;
}
```

- To keep the table half full, we follow these rules:
  - If $N \geq M/2$, call `resize(2*M)` (double table size) *before* a new item is inserted (at the beginning of the `put` method).
  - And if $N \leq M/8$, call `resize(M/2)` (reduce table size by half) *after* a new item is deleted.

Java Libraries for Hash Table

The `java.util` package provides two classes the hash table data structure: `HashMap` and `IdentityHashMap`.

Method	Description
`boolean containsKey(Object key)`	returns `true` if `key` is in the `Map` `false` otherwise.
`boolean containsValue(Object value)`	Returns true if this map maps one or more keys to the specified value.
`V get( K key )`	returns the value associated with key in the `Map`, or `null` if key is not present.
`V put( K key, V value )`	Associates the specified value with the specified key in this map.
`V remove(Object key)`	Removes the mapping for the specified key from this map if present.
`Set<K> keySet()`	Returns a `Set` view of the keys contained in this map. A `Set` is an interface that doesn't allow duplicates.
`Collection<V> values()`	Returns a Collection view of the values contained in this map.
`Set<Map.Entry<K,V>> entrySet()`	Returns a `Set` view of the mappings contained in this map.

`Map` also includes common methods such as `isEmpty`, `clear`, and `size`.

Examples

1. Write a program that removes repeated elements from an array.

Examples

1. Write a program that removes repeated elements from an array.
2. Given two two arrays A and B, check whether both arrays have the same set of numbers. E.g. $A = \{2, 5, 6, 8, 10, 2, 2\}$ and $B = \{2, 5, 5, 8, 10, 5, 6\}$ are do not have the same set of numbers.

Examples

1. Write a program that removes repeated elements from an array.
2. Given two two arrays A and B, check whether both arrays have the same set of numbers. E.g. $A = \{2, 5, 6, 8, 10, 2, 2\}$ and $B = \{2, 5, 5, 8, 10, 5, 6\}$ are do not have the same set of numbers.
3. Given an array of integers, return indices of the two members such that they add up to a specific target. E.g. for $A = \{5, 3, 1, 8, 4\}$ and target $= 11$, the output is $\{1, 3\}$.

Examples

1. Write a program that removes repeated elements from an array.
2. Given two two arrays A and B, check whether both arrays have the same set of numbers. E.g. $A = \{2, 5, 6, 8, 10, 2, 2\}$ and $B = \{2, 5, 5, 8, 10, 5, 6\}$ are do not have the same set of numbers.
3. Given an array of integers, return indices of the two members such that they add up to a specific target. E.g. for $A = \{5, 3, 1, 8, 4\}$ and target $= 11$, the output is $\{1, 3\}$.
4. Given an array A, where each element of the array represents a vote in the election. Assume that each vote is given as an integer representing the ID of the chosen candidate. Give an algorithm for determining who wins the election.