

JICSCI803

Algorithms and Data Structures
September to December 2020

Highlights of Lecture 10

Hashing

Faster Searching

Say we have a set of data consisting of pairs:

E.g. Name and Telephone Number.

How do we find the number associated with a given name?

What is the best data structure to use?

Assume that there are n pairs of data.

Linear list:

Look at every entry.

$\Theta(n)$.

BST:

Traverse the tree from the root.

$\Theta(\log n)$.

$O(n)$ worst case...

...so use a balanced tree.

Even Faster

Can we do better than $\Theta(\log n)$?

How about $\Theta(1)$?

Constant time searching.

To do this we use a dictionary:

Map;

Hash table.

This is a data structure that allows you to determine:

Whether a key is present;

If a key is present what its associated data is.

Operations on Dictionaries

Given a dictionary **D**, with contents consisting of pairs of the form **<key: value>**, we require the following operations to be defined.

Insert:

D[key]=value

Delete:

delete(D[key])

Search:

value=D[key], value == nil if **key** has not been stored.

Note that the dictionary behaves like an array with non-integer index.

Ubiquity

Dictionaries form a part of every modern computer language:

C++:

```
Std::map<key_type, value_type> dictionary_name;
```

Java:

```
Map dictionary_name = new Hashtable();
```

```
Map dictionary_name = new HashMap();
```

```
Map dictionary_name = new LinkedHashMap();
```

Python:

Dictionary data type – created by reference.

```
E.g. en_fr = {"red" : "rouge", "green" : "vert",  
              "blue" : "bleu", "yellow": "jaune"}
```

Motivation

Dictionaries are used in many applications:

- Databases;

 - Fast access to record given key

- Compilers;

 - Maintenance of symbol table

- Network routers;

 - Looking up IP address

- String matching

 - Genetic analysis.

- Security

 - Password checking.

Implementation

There are several ways to implement the dictionary data type:

Let us start with the simplest (and, in most cases, worst) approach:

The Direct Access Table:

Implementation 0: The Direct Access Table

This is simply a big array where the index of the array is the key and the contents of the array is the value.

Only works if keys are integers

E.g. key = phone number, value = name.

So, should we use it in this case?

Typical phone number:

+86 27 6786 7777

11 digits – one hundred billion possible entries

20 characters per name

Two terabytes of storage

For 100,000,000,000 numbers

For 100 numbers!

If \mathcal{U} is the universe of keys $n = |\mathcal{U}|$.

0	value ₀
1	value ₁
2	-
3	-
4	value ₄
...	...
n	value _{n}
$n+1$	-

Fixing the Problems

Problem 1: Keys must be (non-negative) integers.

Solution: define a function

prehash(key) : integer

This function, when given a key of whatever type we need to store returns a non-negative integer value.

So **D[key]=value** becomes **T[prehash(key)]=value**.

T is the direct access table we are using to implement the dictionary, **D**.

Hold on!

That was too easy!

What exactly does **prehash()** do?

Implementing **prehash** ()

In theory:

Every piece of data in a computer is a sequence of bits

Every sequence of bits can be interpreted as a non-negative integer.

Problem solved!

Really?

Consider 8-character keys:

8 characters = 64 bits

Does this mean we need an array with 2^{64} entries?

Implementing **prehash** ()

In Practice:

There are many different possible prehash functions.

Ideally:

$$\text{prehash}(x) = \text{prehash}(y) \Leftrightarrow x = y$$

This is not usually always true, sometimes two different keys may have the same prehash value.

For the sake of simplicity we will assume that the above relationship holds.

Fixing the Problems

Problem 2:

Direct access tables are huge!

Phone numbers:- 2^{11} records

8-letter words:- 2^6 records

Clearly this is a BAD THING™

The problem here is the size of the universe of possible keys $|\mathcal{U}|$.

Solution: Hashing.

Reduce the (huge) universe of all possible keys down to a manageable size, m .

Our table will be of size m .

We have a hash function h so that $0 \leq h(key) < m$ for all valid keys.

Hashing

Ideally, if we have n keys with associated values, we would like $m \in \Theta(n)$.

$$m = 2n, m = 3n.$$

This presents a problem:

Although $m > n$, the number of keys we are storing, it is far smaller than the number of *possible* keys.

There will always be circumstances where $key_1 \neq key_2$ but $h(key_1) = h(key_2)$.

This leads to a collision:

- Two different keys with the same hash value;

- Two different keys with the same location in the table.

How do we fix this?

Chaining

One simple solution to the collision problem is *chaining*.

If two keys hash to the same value store both the records in the same location:

- As a list!

- Yes, I really said, as a list.

- A linked list.

- A *dynamic* data structure.

An Example: Hashing With Chaining

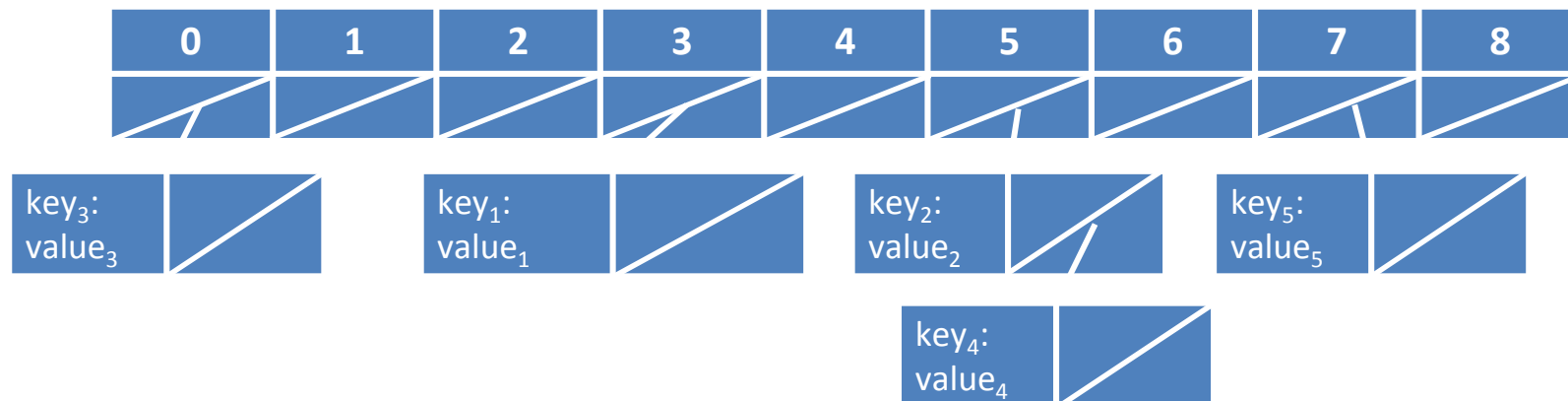
Consider the following data:

$\langle \text{key}_1:\text{value}_1, \text{key}_2:\text{value}_2, \text{key}_3:\text{value}_3, \text{key}_4:\text{value}_4, \text{key}_5:\text{value}_5 \rangle$

With hash values:

$h(\text{key}_1) = 3, h(\text{key}_2) = 5, h(\text{key}_3) = 0, h(\text{key}_4) = 5, h(\text{key}_5) = 7, m=9.$

Our hash table looks like this.



Worst Case

What if $h(\text{key})$ has the same value for all the keys in our set?

Our hash table has just become a complicated way of storing a single linked list!

Access to a given key:value pair is now $\Theta(n)$.

So should we give up on hashing?

No!

In practice this does not happen.

More on Hashing



Hashing: Picking m

As we saw we want m , the number of slots in the dictionary, to be $\Theta(n)$, where n is the number of entries in the dictionary.

Remember: operations on a dictionary are $O(1+n/m)$, so if n grows too large we get less and less efficient.

The problem we face is that, often, we do not know how many records we will need to store.

If m is too small, the dictionary becomes inefficient.

If m is too large, we waste storage (memory or disc).

How do we get the right value for m ?

Let's say we want $m \geq n$ at all times.

Lucky Guess?

If we have no knowledge of the ultimate size of n , what can we do?

Guess.

Pick m based on an optimistic assessment of the likely size of n .

No idea?

Pick your favourite small number ☺.

$m = 8$, say.

Now what?

What if n turns out to be greater than 8?

Make m bigger.

How much bigger?

Changing m

Hang on a sec.

If we change m we have problems:

- Our hash array is too small.

- Our hashed keys will be wrong.

- They depend on the value of m .

Does this mean that we have to recreate the hash table from scratch?

- It sure does.

Isn't this a BAD THING™?

Growing a Hash Table

What exactly has to happen if we change m ?

Let's say the new table size is m' .

We now need a new array with m' elements.

We also need to move all of the existing elements from the old table to the new one.

Build a new hash function h' .

Remember, the hash function depends on m .

Insert the existing data into the new table.

This involves re-hashing every key.

So, the first question is:

How much do we grow m ?

$$m' = ?$$

Each time we grow the table we perform $\Theta(m+n+m')$ operations.

This is $\Theta(n)$.

Let's look at some options:

$$m' = m+1.$$

What is the cost of n insertions?

$\Theta(1)$ for the first m insertions.

$\Theta(m')$ for each insertion after that.

Overall $\Theta(n^2)$

$$m' = ?$$

$$m' = 2m$$

$\Theta(1)$ for the first m insertions.

$\Theta(m)$ for the next insertion.

$\Theta(1)$ for the next $m-1$ insertions.

$\Theta(2m)$ for the next insertion.

$\Theta(1)$ for the next $2m-1$ insertions.

Overall $\Theta(n + (n/2) + (n/4) + \dots) = \Theta(2n) = \Theta(n)$

The cost of expanding the table gets spread over the extra elements we are making room for.

This is known as *Amortized* cost.

Note: an amortized cost of $\Theta(1)$ per operation does not mean that every operation has this cost.

Just that this is the average cost per operation.

Amortized Cost

We say an operation has a cost of “ $T(k)$ Amortized” if k operations take a total of $k \times T(k)$ time.

Table doubling takes $\Theta(n)$ operations for n insertions so the amortized cost is $\Theta(1)$.

This is, actually, a GOOD THING™.

Note: we can use table doubling to implement any solution where we do not know the size of the data structure in advance and it grows in a “well behaved” way.

Table doubling minimizes the cost associated with dynamic data structures.

Deletions

What about deletions?

Each deletion is still $\Theta(1)$.

They simply increase the number of operations (insertions and deletions) we can perform between doublings.

What if it's all deletions?

In this case the table becomes progressively less and less full.

Solution: Shrink the table.

How, exactly?

Shrinking Tables

What should our strategy for reducing the size of the table be?

How about “if $n < m/2$ make $m' = m/2$ ”?

What if the next operation is an insertion?

Double the table size!

Then a deletion?

Halve the table!

Insertion?

Double...

We now have $\Theta(n)$ operations for each change in the data.

Instead use “if $n < m/4$ make $m' = m/2$ ”.



Constant Time?

Although, in our example, $T(n)$ is in $\Theta(1)$; for some operations the actual cost is in $\Theta(n)$.

What does this mean if we have a real-time application?

Every so often we get an insertion /deletion that takes a really looooong time.

Can we remedy this so that **every** operation is in $\Theta(n)$?

The answer is Yes!

We simply adopt the following strategy...

...when a table starts to become full—perform the table doubling in the background.

Keep two sets of the data until you either actually need the double size or until the panic is over.

Hashing With Chaining Considered Bad

There is still one small issue with this method.

We have a hybrid data structure—an array of linked lists.

A second approach uses just a simple array.

Clearly, we still have a potential problem with collision—two keys which hash to the same value.

We resolve this with a technique known as *Open Addressing*.

Open Addressing

We wish to hash n items into an array with m slots.

We may only store one item per slot.

Clearly, $m \geq n$.

We insert an item into the table using an iterative technique known as *probing*.

Probing

This process works as follows: (for insertion)

Set hash function to starting value, h_0

repeat

 calculate $probe = hash(key)$

 if $table(probe)$ contains data then

 go to the next hash function

 else

 store the item in $table(probe)$

 fi

until we have stored the item

This means we must have a sequence of hash functions, h_0, h_1, h_2, \dots

... or a hash function which produces a sequence of values.

The Hash Function

Our new hash function requires two arguments:

The key;

The iteration count.

Thus: **probe=OpenHash (key, count)**

Here:

key is a valid element of \mathcal{U} , the universe of keys;

count is a non-negative integer.

As usual, $0 \leq \mathbf{probe} < m-1$.

- ▶ In addition, we want our hash function to have the following property:
- ▶ For any arbitrary key k the sequence of m probes:
 - ▶ $h(k, 0), h(k, 1), h(k, 2), \dots, h(k, m-1)$;
- ▶ Must be a permutation of the integers:
 - ▶ $0, 1, 2, \dots, m-1$.
- ▶ This property guarantees that we must eventually find a vacant slot to insert the item into.
- ▶ Clearly, the sequence of probes must be different for different keys.
- ▶ We can see this with an example.

Example: Insertion with Open Addressing

Consider the following table:

k	$h(0,k)$	$h(1,k)$	$h(2,k)$	$h(3,k)$	$h(4,k)$	$h(5,k)$	$h(6,k)$	$h(7,k)$	$h(8,k)$	$h(9,k)$
899	9	8	5	6	0	7	8	2	4	1
950	5	7	4	9	2	3	1	6	8	0
12	3	8	7	2	5	9	1	6	0	4
367	7	1	2	3	4	5	6	8	9	0
359	2	1	9	5	6	7	3	8	0	4
980	4	7	1	8	9	3	0	5	2	6
229	0	8	2	7	1	6	3	9	4	5
598	8	6	3	5	0	7	9	1	4	2
838	6	2	6	7	1	3	8	2	0	2
549	9	8	4	6	7	5	0	1	2	3

Let us insert the keys into our hash table in order

	k	h(0,k)	h(1,k)	h(2,k)	h(3,k)	h(4,k)	h(5,k)	h(6,k)	h(7,k)	h(8,k)	h(9,k)
→	899	9	8	5	6	0	7	8	2	4	1
→	950	5	7	4	9	2	3	1	6	8	0
→	12	3	8	7	2	5	9	1	6	0	4
→	367	7	1	2	3	4	5	6	8	9	0
→	359	2	1	9	5	6	7	3	8	0	4
→	980	3	7	1	8	9	4	0	5	2	6
→	229	0	8	2	7	1	6	3	9	4	5
→	598	8	6	3	5	0	7	9	1	4	2
→	838	6	2	4	7	1	3	8	2	0	2
→	549	9	8	4	6	7	5	0	1	2	3

0	1	2	3	4	5	6	7	8	9
229	980	359	12	549	950	838	367	598	899

Search with Open Addressing

The procedure used to search using open addressing is similar to insertion.

```
count=0
```

```
repeat
```

```
    probe=hash(key, count)
```

```
    if table(probe)==key then
```

```
        return item
```

```
    else
```

```
        count++
```

```
    fi
```

```
until table(probe)==empty or count==n
```

```
return not found
```

This is pretty straightforward.

Deletion with Open Addressing

When we get to deletion we have a new problem.

```
count=0
```

```
repeat
```

```
    probe=hash(key, count)
```

```
    if table(probe)==key then
```

```
        delete item
```

```
        return
```

```
    else
```

```
        count++
```

```
    fi
```

```
until table(probe)==empty or count==n
```

```
return not found
```

How, exactly, do we delete the item?

Deletion...

If we simply replace the item with our empty value we will have an issue:

What if the key we next search for is after the probe corresponding to the deleted key's location.

If, in our previous example, we delete 899, where $h(899,0)=9$, and then search for 549, where the sequence of hash values are 9, 8, 4...

We test $D(9)$ and discover it has the value **empty**.

We conclude that 549 is not in the table.

Wrong! It is in $D(4)$.

To fix this we need a second special value, **deleted**.

Deletion concluded

Our deletion process becomes:

```
count=0
repeat
    probe=hash(key, count)
    if table(probe)==key then
        table(probe)==deleted
        return
    else
        count++
    fi
until table(probe)==empty or count==n
return not found
```

This fixes search but introduces a problem with insertion.

Insertion Revisited.

We note that we can insert a new item into the dictionary in two circumstances:

```
D(i)==empty  
D(i)==deleted
```

We modify our insert process as follows:

```
count=0  
repeat  
    probe=hash(key, count)  
    if table(probe)==empty or table(probe)==deleted then  
        store item in table(probe)  
        return  
    else  
        count++  
    fi  
until count==n  
return no room
```

Now we can insert into the first vacant slot, empty or deleted, that we find in the table.

Search Revisited

Because **empty** and **deleted** are different, we do not have to modify our search procedure.

The search will skip over deleted records because they do not match the key but will still terminate when it reaches an empty record.

Open Addressing Hash Functions

One question remains.

Can we find a function $h(k, i)$ which is:

- Easy to compute;

- Produces a permutation of $\{0, 1, \dots, m-1\}$ as i varies over $\{0, 1, \dots, m-1\}$?

Let us examine two possible strategies.

Strategy I: Linear Probing

In this approach we simply take a standard hash function, $h(k)$ and compute the probe $p(k, i)$ as follows:

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

In other words, we simply look at sequential entries in the dictionary starting at the entry corresponding to $h(k)$.

This is certainly easy to compute.

It does satisfy the permutation.

Is it any good?

No!

It produces sets of consecutive occupied slots.

Clustering.

The bigger the cluster, the more likely it is to be hit..

...and it gets even bigger!

Strategy II: Double Hashing

In this strategy we have two standard hash functions, $h_1(k)$ and $h_2(k)$.

We compute $p(k)$, our probe value as follows:

$$p(k, i) = (h_1(k) + i \times h_2(k)) \bmod m.$$

Do we still satisfy our requirements?

This is still easy to compute.

Do we always get a permutation?

No.

Unless we are clever in how we define h_2 .

Choosing h_2

We need $h_2(k)$ to be relatively prime to m .
I.e. $h_2(k)$ and m must have no common factors except 1.

This is easy in many cases.

If we select m to be a power of 2; say $m = 2^r$ then all we need is for $h_2(k)$ to always be an odd number.

For example, if we have a standard hash function $h'(k)$, we can create $h_2(k)$ as follows:

$$h_2(k) = (2h'(k) + 1) \bmod m$$

Table Doubling.

Once again, we need to expand the dictionary whenever it becomes too full.

What does “too full” mean in this case?

We define the occupancy of a table, α , to be the ratio of n , the number of entries to m , the number of slots.

$$\alpha = n/m$$
$$0 \leq \alpha \leq 1$$

We can show that the average cost of an operation on a table with occupancy α is in $\Theta(1/(1 - \alpha))$.

In practice we want this value to be reasonably close to 1 so we double as soon as α exceeds 0.5 or thereabouts. This keeps operations between $\Theta(1)$ and $\Theta(2)$.

An Important Note on α

When calculating the occupancy value, α , we must count slots with a value of **deleted** as containing data.

This is because some operations, notably searching, treat deleted records as still containing data.

Slots containing **deleted** may be removed in two ways:

- Being overwritten with valid data as a result of an insert operation;
- Being cleaned up when the table is expanded.

If we did not count **deleted** records in calculating α we could have a notionally empty table in which every slot was **deleted**.

Search (and delete) in this table would be $\Theta(m)$, not $\Theta(1)$, as we might expect.

Chaining vs. Open Addressing

So, which is the better scheme?

Open Addressing:

- Uses less memory—no need for pointers;
- Is faster—provided α is kept below 0.5;
- Is a little harder to implement and understand.
- Is clean—one data structure, the array.

Chaining:

- Uses more memory;
- Is faster—if we are not careful with open addressing.
- Is a little easier to implement and understand.
- Is a bit messy—arrays of linked lists.

I know where my vote goes!

Applications of Hashing

String Searching

Looking for Text (In all the right places)

Consider the problem of *String Searching*:

Given a text, t , is the subtext, s , present in it?

This problem occurs in many real-life applications:

- grep;
- find in a text editor;
- Genome matching;
- Google search.

There are a wide number of techniques to achieve this.

Let us look at a couple of examples.

The Naïve Approach: Linear Search

The simplest possible approach is linear search:

Try to match s starting at each location in t .

for i in $0 \dots \text{length}(t) - \text{length}(s)$

$j=0$

 while $j < \text{length}(s)$ do

 if $(s(i+j) \neq t(i+j))$ break

$j++$

 od

 if $j == \text{length}(s)$ print(" string found starting at location " i)

rof

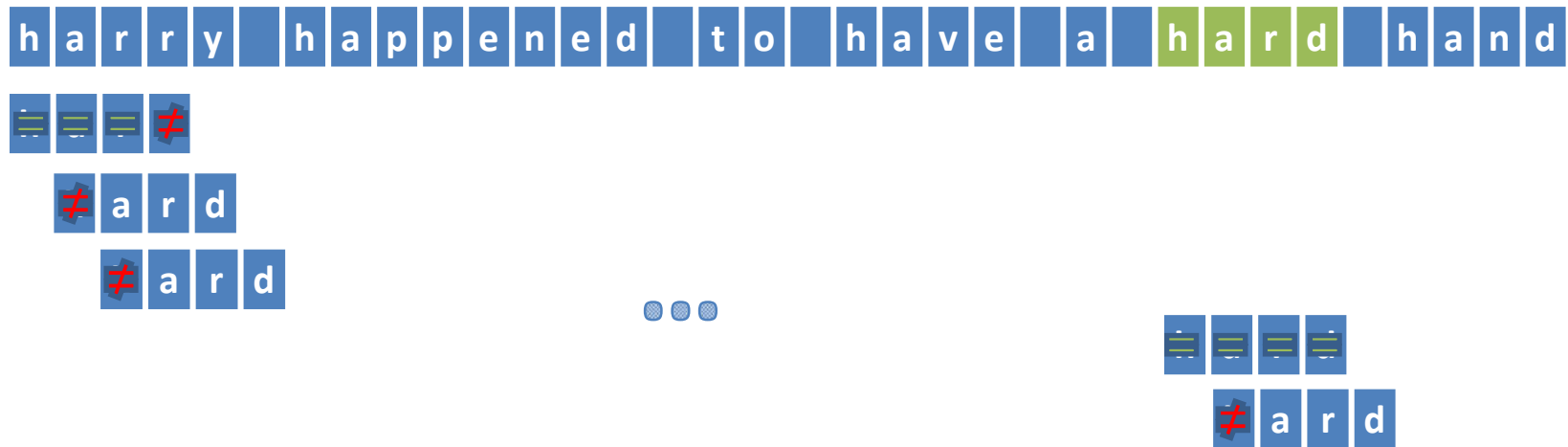
We can see this with an example.

Linear Search: an Example

Let t be the string “harry happened to have a hard hand”.

Let s be the string “hard”.

The search proceeds as follows:



Linear Search \neq Linear Time Search

The outer loop in our algorithm is repeated $|t|-|s|$ times.

Typically the string t is much longer than the string s , so this is $\Theta(t)$.

The inner loop is repeated up to $|s|$ times for each time round the outer loop.

This is $\Theta(s)$

The total number of comparisons is $\Theta(|s| \times |t|)$.

Is this the best we can do?

The best we can possibly do is $\Theta(|s|+|t|)$;

we have to at least look at each string!

Can we actually achieve this goal of a linear time algorithm?

Linear Time Search

To do this we will use hashing.

We compare the hash of string s with the hash of each substring of t with the same length:

```
hash_s = hash(s)
for i in 0...length(t)-length(s)
    hash_t = hash(t[i..i+length(s)-1])
    if hash_s == hash_t then
        brute-force compare s and the substring
        if they match print("string found starting at
        location" i)
    fi
rof
```

Linear Time Search

This algorithm takes linear time, provided:

- The hash function only collides rarely;

- The hash function takes constant time to compute;

- Independent of the length of string s !

Surely, the second requirement is impossible.

To hash a string of length $|s|$ must take $\Theta(|s|)$ operations.

- Yes?

- No!

Not if we are clever.

Clever Hashing

We note that we need $\Theta(|s|)$ time to compute $h(s)$.

We also need $\Theta(|s|)$ time to compute $h(t[0..|s|-1])$, the initial substring of t .

The trick is to compute the hash of each successive substring of t in constant time.

If we look closely at these substrings we see an interesting feature:

Successive substrings differ only by two characters.

The first character of the first substring;

h a r r

The last character of the next substring.

a r r y

Rolling Hash

Maybe we can define a hash function which, given $h(\text{"harr"})$ can compute $h(\text{"arry"})$ in constant time.

Let us define a *rolling hash* function, $r()$, so that:

$$h(\text{"arry"}) = r(h(\text{"harr"}), \text{"h"}, \text{"y"})$$

We compute the hash of the next substring by removing the first and appending the new last characters;

In this case we remove "h" and append "y".

If we can compute a rolling hash in constant time than we can do string matching in linear time.

How?

Karp-Rabin String Search

The Karp-Rabin algorithm looks like this:

```
hash_s=hash(s)
hash_t=hash(t[0..length(s)-1])
for i in 0..length(t)-length(s)
    if hash_s == hash_t then
        brute-force compare s and the substring
        if they match print("string found starting at
        location " i)
    fi
    hash_t=roll(hash_t,t[i],t[i+length(s)])
rof
```

The function $\text{roll}(h,p,s)$ computes the rolling hash of the next substring given the hash of the existing substring, h , with the prefix p removed and the suffix, s , appended. We need only find a suitable function $\text{roll}()$.

How We Roll

One popular way to compute `roll()` is to use something called the Rabin fingerprint.

We start by treating each symbol in the alphabet as an integer – use the ASCII code for example.

We then find a random prime number $>$ the size of the alphabet—let's pick 257.

We now compute $h(\text{"harr"})$ as:

$$257^3 \cdot 104 + 257^2 \cdot 97 + 257^1 \cdot 114 + 257^0 \cdot 114 \\ = 1,771,793,837$$

Note: "h" = 104, "a" = 97 and "r" = 114.

The Next Hash

Ok, so given that $h(\text{"harr"}) = 1,771,793,837$ how do we get $h(\text{"arry"})$?

It's easy:

Simply compute $r(h,p,s) = 257 \cdot (h - 257^3 \cdot p) + s$

$257 \cdot (1,771,793,837 - 257^3 \cdot 104) + 121$

In this case the result is 1,654,094,526 which is exactly the same as $h(\text{"arry"})$

$257^3 \cdot 97 + 257^2 \cdot 114 + 257^1 \cdot 114 + 257^0 \cdot 121$

Note: if these values become too large we can reduce them modulo m , where m is a convenient value—say 2^{15} or 2^{31} .

Efficient?

We can compute our hash values for s and the initial substring of t using *compact evaluation*.

$$p^{k-1}.c_1 + p^{k-2}.c_2 + \dots + p.c_{k-1} + c_k$$

$$h = c_k + p(c_{k-1} + p(c_{k-2} + \dots + p(c_3 + (p.c_2 + c_1))..))$$

Where $k=|s|$ and c_i is the i^{th} character of s .

This requires $|s|-1$ multiplications and $|s|-1$ additions.

Efficiency!

If we precompute $q=p^{k-1}$ we can find the next hash value, h' as:

$h'=p.(h-q.c_i)+c_j$ where we remove character i and add character j .

This requires only 1 multiplication and 1 addition.

Constant time.

Thus we have $\Theta(|s|)$ operations to perform the initial hashes and $|t|-|s| * \Theta(1)$ operations to do the rehashing.

Overall: our algorithm operates in $\Theta(|s|+|t|)$ time.

We win!

Discussions

1. List the problems that can use HASH.

Homework

1. Implement STRING SEARCH in hash.