Hashing: Implementation Issues

EECS 233

Recap

- Hash functions desiderata
- Handling collisions with separate chaining
- Handling collisions with open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
  - Need to distinguish between "removed" and "empty" positions

Implementation of Hash Tables

- A simple hash table with open addressing.

```
public class HashTable {
        private class Entry {
                private String key;
                private String etymology;
                private boolean removed;
        }
        private Entry[] table;
        private int tableSize;
        ...
}
```

- for an empty position, table[i] will equal null

- for a removed position, table[i] will refer to an Entry object whose *removed* field equals true

- for an occupied position, table[i] will refer to an Entry object whose *removed* field equals false

- open (“unoccupied”, “available”) position is either empty or removed

Constructors

- Initializing the hash table

```
public HashTable(int size) {
	table = new Entry[size];
	tableSize = size;
}
```

- Initializing an entry (before insertion)

```
private Entry(String key, String etymology) {
	this.key = key;
	this.etymology = etymology;
	removed = false;
}
```

```
public class HashTable {
	private class Entry {
		private String key;
		private String etymology;
		private boolean removed;
	}
	private Entry[] table;
	private int tableSize;
	...
}
```

Finding An Open Position

- Using double hashing

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function

        // keep probing while the current position is occupied (non-empty and non-
           removed)
        while (  table[i] != null && table[i].removed==false  )

                i = (i + j) % tableSize;

        return i;
}
```

- Does it always terminate?

Finding An Open Position: Infinite Loops

- The loop in our probe method could become infinite.
- To avoid infinite loops, we can stop probing after checking n positions (n = table size) because the probe sequence will just repeat after that point.
  - for double hashing:
    - $(h1 + n*h2) \% n = h1 \% n$
    - $(h1 + (n+1)*h2) \% n = (h1 + n*h2 + h2) \% n = (h1 + h2)\%n$
    - $(h1 + (n+2)*h2) \% n = (h1 + n*h2 + 2*h2) \% n = (h1 + 2*h2)\%n$
    - …
  - for quadratic probing:
    - $(h1 + n^2) \% n = h1 \% n$
    - $(h1 + (n+1)^2) \% n = (h1 + n^2 + 2n + 1) \% n = (h1 + 1)\%n$
    - $(h1 + (n+2)^2) \% n = (h1 + n^2 + 4n + 4) \% n = (h1 + 4)\%n$
    - …

Finding An Open Position: Infinite Loop Protection

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing until we get an empty or removed position
        while (table[i] != null && table[i].removed==false) {
                i = (i + j) % tableSize;
                iterations++;
                if (iterations >= tableSize) return -1;
        }

        return i;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while (   table[i] != null                         ) {
                // return if key is found, otherwise continue
                if (table[i].removed==false && table[i].key.equals(key))
                    return i;

                i = (i + j) % tableSize;
                iterations++;
                if (iterations >= tableSize) return -1;
        }

        return -1;
}
```

Search() Method

- Search for the entry with the key, and return the associated data (string etymology in our example)

```
public String search(String key) {
        int i = findKey(key);
        if (i == -1)
                return null;
        else
                return table[i].etymology ;
}
```

- It calls the helper method findKey() to locate the position of the key.

Remove() Method

- Search the hash table and delete the key if found

```
public void remove(String key) {
    int i = findKey(key);
    if (i == -1)
        return;
    table[i].removed = true;
}
```

- It also uses findKey().

Insertion

- We begin by probing for the key to find an empty position.

- Two cases:

  - Encountered an open (empty or removed) position while probing
    - put the (key, value) pair in the open position

  - No removed or empty position encountered
    - Overflow: throw an exception

Insert() Method

- If no empty or removed position is available, report error; otherwise, insert the new entry

```
public void insert(String key, String etymology) {
        int i = probe(key);
        if (i == -1)
                throw new RuntimeException("HashTable full");
        else
                ?
}
```

Hashing: Implementation Issues

EECS 233

Issues and Questions

- As the hash table is used, more positions will be marked removed
  - How does this affect running time?
    - Insertion?
    - Search? (Successful/unsuccessful)?
    - Deletion?
- If the hash table gets almost full, how does this affect running time?
  - Insertion?
  - Search?
  - Deletion?

Hash Table Performance May Degrade

- When many entries are "removed", search must continue
  - If the searched key is in the table, it is likely to be found after a few iterations
  - However, if the key is not in the table at all, search continues until reaching an "empty" entry, potentially approaching N iterations (N is the size of the table)

- When most entries are occupied, insertion method may not be able to find an open position quickly
  - the *load factor* λ of a hash table is the ratio of the number of keys to the table size.
  - The expected number of iterations obviously grows with λ

Rehashing

- What to do in the first case?
  - Here the load factor is not high at all, but that many entries are not utilized (marked as "removed").
    - the load factor can be easily tracked.
  - "Rehash" the entries: Create another hash table, and move every entry to the new table

- What to do in the second case?
  - Here the problem is high load factor
  - Expand the hash table: create a new hash table of roughly doubled size, and move the entries to the new table

- What is the running time of rehashing? Does it affect the overall running of hash table operations?
  - May not happen very often, e.g., once every O(N) insertions
  - But when it happens, may take some running time

Rehash() Method (Expanding)

- Let us still consider the example hash table:

```
public class HashTable {
        private class Entry {
                private String key;
                private String etymology;
                private boolean removed;
        }
        private Entry[] table;
        private int tableSize;
        ...
}
```

```
public void rehash( ) {
        int oldSize = tableSize;
        Entry[ ] oldTable = table;

        tableSize = nextPrime(2 * oldSize);
        table = new Entry[tableSize];
        for(i = 0; i < oldSize; i++)
                if ( ? )
                    ?
}
```

- We define rehash() as public so the user can decide when to rehash

Efficiency of Hash Tables

- In the best case, search and insertion are *O*(1).

- In the worst case, search and insertion are linear.
  - open addressing: *O*(m), where m = the size of the hash table
  - separate chaining: *O*(n), where n = the number of keys

- With a good choice of hash function and table size, the time complexity is generally better than *O*(log n) and approaches *O*(1).

- As the load factor increases, performance tends to decrease.
  - Open addressing: try to keep the load factor < ½
  - Separate chaining: try to keep the load factor < 1

- Time-space tradeoff: bigger tables tend to have better performance, but they use up more memory.

Limitations of Hash Tables

- It can be hard to come up with a good hash function for a particular data set.
  - The choice of hash functions may depend on the characteristics of the data set.

- The items are not ordered by key. As a result, we can't easily
  - Print the contents in sorted order
  - Solve the selection problem - get the k-th largest item

- We *can* do all of these things with many tree structures.

Implementation of Hash Function

Hash Functions Desiderata Revisited

- Example: String as key (e.g., Webster)
  - keys = character strings composed of lower-case letters
  - hash function:
    - h(key) = (the byte sum of all characters) mode table-size
    - example: h("cat") = ('a' + 'c' + 't') mod 100000
    - But: assume words are mostly up to 20 character-long
      - Max sum is 127*20 = 2540; any larger table is of no use!
    - But: permutations are not distinguished
      - h("cat") = h("act")

- Requirements for good hash functions:
  - Full table size utilization
  - Even ("uniform") key mapping throughout the table
    - Utilizing known key distribution in the objects
    - Making hash function "random" - the distribution of keys is independent of distribution of indexes to which they map

      Using the entire key to compute the hash value

  Must be efficient to compute!

Hash Functions for Strings

- Bad example: the sum of the character codes
- A better example: a weighted sum of the character codes
  - $h_b = (a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b^1 + a_{n-1}) \bmod (\text{tableSize})$
    - a_i is encoding of the i-th character,
    - b is a constant
  - All characters contribute
  - Contribution of a character depends on its position

- Examples for b = 31:
  - $h_b(\text{"table"}) = 116*31^4 + 97*31^3 + 98*31^2 + 108*31 + 101 = 110115790$
  - $h_b(\text{"eat"}) = 101*31^2 + 97*31 + 116 = 100184$
  - $h_b(\text{"tea"}) = 116*31^2 + 101*31 + 97 = 114704$

- Java uses this hash function with b = 31 in the hashCode() method of the String class.

What happens if we use a table size of 31?

Hash Functions for Strings

- How to calculate h_b(supercalifragilisticexpialidocious)?
  - 's' b^{33} + 'u' b^{32} + … + 's'.
  - A lot of multiplications!
- Better way: Horner's method
  - example: $101*31^2 + 97*31 + 116 = (101*31 + 97)*31 + 116$
  - $101*31^3 + 97*31^2 + 116*31 + 110 = ((101*31 + 97)*31 + 116)*31 + 110$
  - $a_0 b^{n-1} + a_1 b^{n-2} + \dots + a_{n-2} b^1 + a_{n-1} = (\dots((a_0 b + a_1)b + a_2)b + \dots + a_{n-2})b + a_{n-1}$

```
int hash = s.charAt(0);
for (int i = 1; i < s.length(); i++)
    hash = hash * b + s.charAt(i);
```

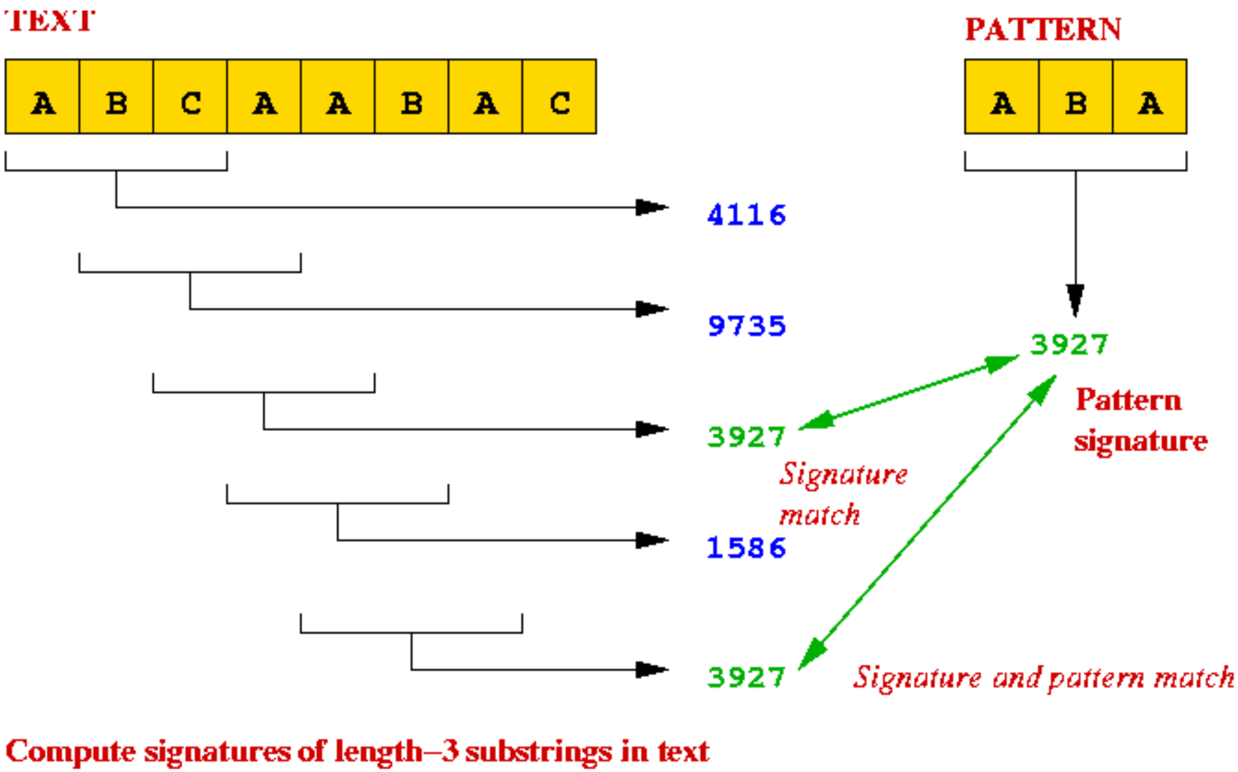
Substring Pattern Matching

- Input: A text string t and a pattern string p.
- Problem: Does t contain the pattern p as a substring, and if so where?
- **Brute Force:** search for the presence of pattern string p in text t overlays the pattern string at every position in the text. → O(mn)

(m: size of pattern, n: size of text)

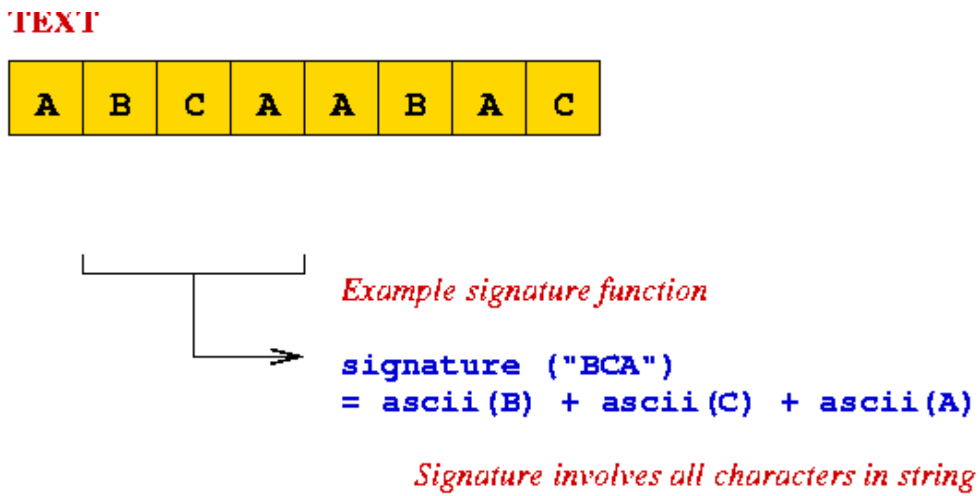
- **Via Hashing:** compute a given hash function on both the pattern string p and the m-character substring starting from the ith position of t. → O(n)

Substring Pattern Matching (Rabin–Karp Algorithm)



- Compute the "signature" of the pattern.
- Compute the "signature" of each substring of the text.
- Scan text until signature matches => potential match, so perform string comparison

Substring Pattern Matching

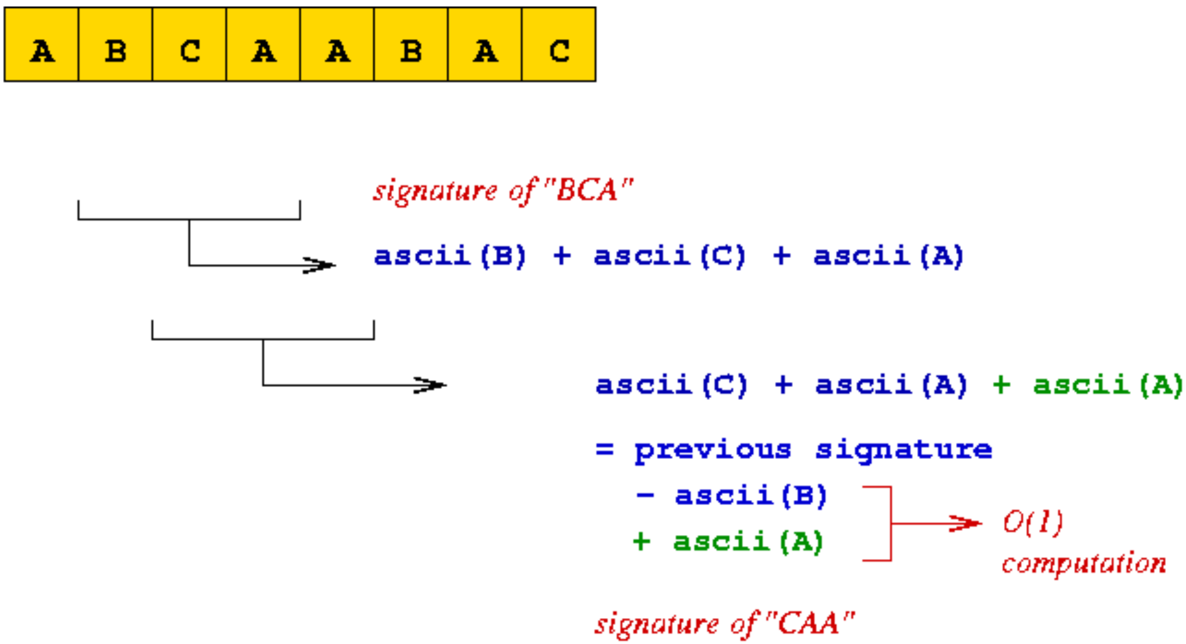


$\Rightarrow$ signature involves all characters

=> computing signature for each substring takes *O(m)* time.

=> no faster than naive search.

Substring Pattern Matching



Observation: two successive substrings differ by only two characters.

=> next signature can be computed quickly (*O(1)*).