Data Structures

Hash Tables

CS284

Hash Tables

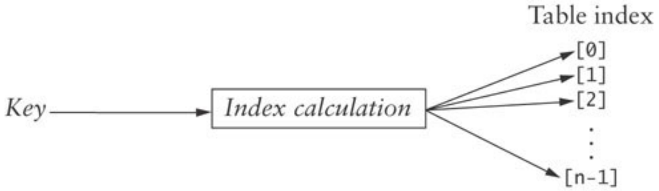
- Goal

  Access an entry based on its key, not its location

- Highlight of hash table:
  - Retrieve an entry in constant time (on average, $\mathcal{O}(1)$)

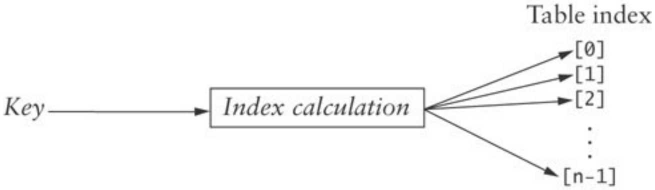
Hash Codes and Index Calculation

Basis: transform the item's key value into a table index



- A good hash function should be:
  - relatively simple and efficient to compute
  - each table index should be equally likely for each key

Index Calculation



Typically two parts:

- Hash code computation
- Index calculation using the hash code

Example

- Consider the Huffman code problem
  - Determine the frequency with which a character occurs in a text
- If a text contains only ASCII values, which are the first 128 Unicode values, we could use a table of size 128 and let its Unicode value be its location in the table
- You would compute the index using

```
int index = asciiChar
```

- Eg. the index for “A” would be 65

...	...
65	A, 8
66	B, 2
67	C, 3
68	D, 4
69	E, 12
70	F, 2
71	G, 2
72	H, 6
73	I, 7
74	J, 1
75	K, 2
...	...

Example

- However, what if all 65,536 Unicode characters were allowed?
- If you assume that on average 100 characters were used, you could use a table of 200 characters and compute the index by

```
int index = unicode % 200
```

...	...
65	A, 8
66	B, 2
67	C, 3
68	D, 4
69	E, 12
70	F, 2
71	G, 2
72	H, 6
73	I, 7
74	J, 1
75	K, 2
...	...

Methods for Generating Hash Codes

- If a text contains this snippet:

```
...mañana (tomorrow), I'll finish my program...
```

- Given the following Unicode values:

Hexadecimal	Decimal	Name	Character
0x0029	41	right parenthesis	)
0x00F1	241	small letter n with tilde	ñ

- The indices for letters 'ñ' and ')' are both 41:

$$41 \;\%\; 200 = 41 \text{ and } 241 \;\%\; 200 = 41$$

- This is called a collision; we will discuss how to deal with collisions shortly

Hash Function Examples

- Phone numbers
  - Bad: first three digits
  - Better: last three digits
- Social Security numbers
  - Bad: first three digits
  - Better: last three digits
- Strings
  - Bad: summing the int values of all characters (returns the same hash code for “sign” and “sing”)
  - Better: Account for position of characters as well (continued)

Java and `hashcode()`

- All classes implement `hashCode()`
  - It returns a 32-bit int (between -2147483648 and 2147483647)
- A simple hash function that can be uses as a table index:
- Example:

```java
String s = "call";
int code = s.hashCode();
int hash = code % M;
```

- Note:
  - `hash` could be negative
  - Use `((code & 0x7fffffff) % M)` as array index
  - This sets the sign bit to positive

Java HashCode Method for `String` Class

- Java provides default implementation of `hashCode()`
- Some classes override it (eg. `String, URL, Integer, Date`)
- `String.hashCode()` returns the integer calculated by:

$$s_0 * 31^{(n-1)} + s_1 * 31^{(n-2)} + ... + s_{n-1}$$

  where s_i is i-th character of the string, and n its length

  - "Cat" has a hash code of: 'C' * 31^2 + 'a' * 31 + 't' = 67,510
  - 31 is a prime number, and prime numbers generate relatively few collisions

```
public int hashCode() {
  int hash = 0;
   for (int i = 0; i < length(); i++) {
     hash = s[i] + (31 * hash);
   }
   return hash;
}
```

Open Addressing

- We now consider two ways to organize hash tables:
  - open addressing
  - chaining
- In open addressing, linear probing can be used to access the value associated to a key `k` in a hash table
  - If the index calculated for `k` is occupied by an item `(k,v)`, we have found the item
  - If that element contains an item with a different key `(k',v)`, increment the index by one
  - Keep incrementing until you find the key or a null entry (assuming the table is not full)

The Algorithm – `Find(Key k)`

```
Compute index by taking hashcode of k % table.length
if (table[index]==null) {
   item is not in the table
} else if (table[index]==(k,v)) {
  the item is in the table
} else {
  Continue to search the table by incrementing the index until
    either k or a null entry is found
}
```

- As you increment the table index, your table should wrap around as in a circular array

Table Wraparound and Search Termination

- Since this method wraps around after the end of the table, you can search the part of the table before the hash code value in addition to the part of the table after the hash code value
- But it could lead to an infinite loop
- How do you know when to stop searching if the table is full and you have not found the correct value?
    - Stop when the index value for the next probe is the same as the hash code value for the object
- Ensure that the table is never full by increasing its size after an insertion when its load factor exceeds a specified threshold

Hash Code Insertion Example

Name	**hashCode()**	**hashCode()%5**
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

0.	
1.	
2.	
3.	
4.	Tom

Implementing `find`

```java
public class HashtableOpen<K, V>
    extends AbstractMap<K, V>{

    private Entry<K, V>[] table;
    private static final int START_CAPACITY = 101;
    private double LOAD_THRESHOLD = 0.75;
    private int numKeys;
    private int numDeletes;
    private final Entry<K, V> DELETED =
            new Entry<K, V>(null, null);

    public static class Entry<K, V> implements Map.Entry<K, V> {

        private K key;
        private V value;

        public Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
        // continued
```

Implementing `find`

```
        @Override
        public K getKey() {
            return key;
        }

        @Override
        public V getValue() {
            return value;
        }

        @Override
        public V setValue(V val) {
            V oldVal = value;
            value = val;
            return oldVal;
        }

        @Override
        public String toString() {
            return key.toString() + "=" + value.toString();
        }
    }
```

Implementing `find`

```
/**
   * Finds either the target key or the first empty slot in the
   * search chain using linear probing.
   * @pre The table is not full.
   * @param key The key of the target object
   * @return Position of the target or the first empty slot if
   *         the target is not in the table.
   */
   private int find(Object key) {
       int index = key.hashCode() % table.length;
       if (index < 0) {
           index += table.length; // Make it positive.
       }
       while ((table[index] != null)
               && (!key.equals(table[index].key))) {
           index++;

           if (index >= table.length) {
               index = 0; // Wrap around.
           }
       }
       return index;
   }
```

Deleting an Item Using Open Addressing

- When an item is deleted, you cannot simply set its table entry to null. Why?

Deleting an Item Using Open Addressing

- When an item is deleted, you cannot simply set its table entry to null. Why?
    - If we search for an item that may have collided with the deleted item, we may conclude incorrectly that it is not in the table.
- Instead, store a dummy value or mark the location as available, but previously occupied
- Deleted items reduce search efficiency which is partially mitigated if they are marked as available
- You cannot simply replace a deleted item with a new item until you verify that the new item is not in the table

Deleting an Item Using Open Addressing

```
@Override
 public V remove(Object key) {
     int index = find(key);
     if (table[index] == null) {
         return null;
     }
     V oldValue = table[index].value;
     table[index] = DELETED;
     numKeys--;
     return oldValue;
 }}
```

Reducing Collisions by Expanding the Table Size

- Use a prime number for the size of the table to reduce collisions
- A fuller table results in more collisions, so, when a hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted
- You must reinsert (rehash) values into the new table; do not copy values as some search chains which were wrapped may break
- Deleted items are not reinserted, which saves space and reduces the length of some search chains

Reducing Collisions Using Quadratic Probing

- Linear probing tends to form clusters of keys in the hash table, causing longer search chains
- Quadratic probing can reduce the effect of clustering
  - Increments form a quadratic series $(1 + 2^2 + 3^2 + ...)$
  - `probeNum` starts at 0

```
probeNum++;
index=(startIndex+probeNum*probeNum) % table.length
```

- If an item has a hash code of 5, successive values of index will be 6 (5+1), 9 (5+4), 14 (5+9), ...

Reducing Collisions Using Quadratic Probing

Insert elements with hash codes 5,6,5,6,7 using

- Linear probing
  - Problem: clustering
- Quadratic probing

```
probeNum++;
index=(startIndex+probeNum*probeNum) % table.length
```

  - If an item has a hash code of 5, successive values of index will be 6 (5+1), 9 (5+4), 14 (5+9), ...

Problems with Quadratic Probing

- Next index calculation is time-consuming, involving multiplication, addition, and modulo division
- A more efficient way to calculate the next index is:

```
k += 2;
index = (index + k) % table.length;
```

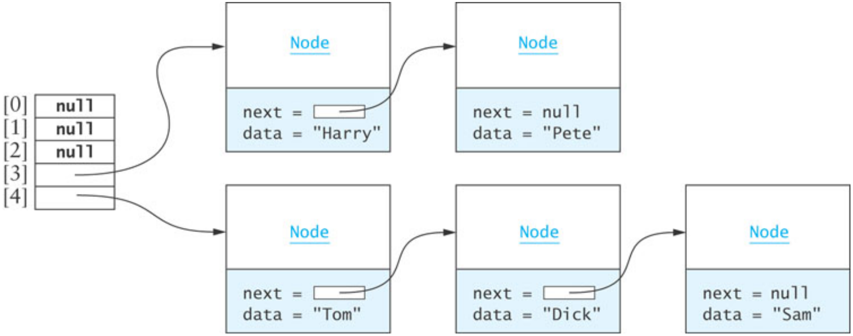
  - If initial value of k is -1, successive values of k will be 1, 3, 5, ...
  - If the initial value of index is 5, successive value of index will be 6 ($= 5 + 1$), 9 ($= 5 + 1 + 3$), 14 ($= 5 + 1 + 3 + 5$), ...
  - The proof of the equality of these two calculation methods based on mathematical series: $n^2 = 1 + 3 + 5 + ... + 2n - 1$

Problems with Quadratic Probing

- A more serious problem is that not all table elements are examined when looking for an insertion index; this may mean that an item can't be inserted even when the table is not full
- It is also possible that the program will get stuck in an infinite loop searching for an empty slot
  - If the table size is a prime number and it is never more than half full, this won't happen
  - However, requiring a half empty table wastes a lot of memory

Alternative to Open Addressing – Chaining

- Each table element references a linked list that contains all of the items that hash to the same table index
  - The linked list often is called a bucket
  - The approach sometimes is called bucket hashing



Chaining (cont.)

Advantages relative to open addressing:

- Only items that have the same value for their hash codes are examined when looking for an object
- You can store more elements in the table than the number of table slots (indices)
- Once you determine an item is not present, you can insert it at the beginning or end of the list
- To remove an item, you simply delete it; you do not need to replace it with a dummy item or mark it as deleted

Performance of Hash Tables

Load factor: $\frac{\text{number of filled cells}}{\text{table size}}$

- Load factor has the greatest effect on hash table performance
- The lower the load factor, the better the performance as there is a smaller chance of collision when a table is sparsely populated
- If there are no collisions, performance for search and retrieval is $\mathcal{O}(1)$ regardless of table size

Performance of Open Addressing versus Chaining

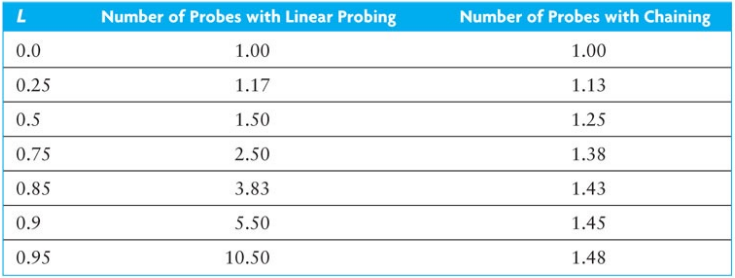
Open Addressing with Linear Probing

- Expected number of comparisons c required for finding an item that is in a hash table with load factor L (Donald Knuth)
$$c = \frac{1}{2}(1 + \frac{1}{1-L})$$

Chaining

- If an item is in the table, on average we must examine the table element corresponding to the item's hash code and then half of the items in each list
$$c = 1 + \frac{L}{2}$$

Performance of Open Addressing versus Chaining (cont.)



L	Number of Probes with Linear Probing	Number of Probes with Chaining
0.0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

Performance of Hash Tables versus Sorted Array and Binary Search Tree

- The number of comparisons required for a binary search of a sorted array is $\mathcal{O}(\log n)$
  - A sorted array of size 128 requires up to 7 probes (2^7 is 128) which is more than for a hash table of any size that is 90% full
  - A binary search tree performs similarly
- Insertion or removal

hash table	$\mathcal{O}(1)$ expected; $\mathcal{O}(n)$ worst case
sorted array	$\mathcal{O}(n)$
BST	$\mathcal{O}(\log n)$; worst case $\mathcal{O}(n)$

Storage Requirements for Hash Tables, Sorted Arrays, and Trees

- The performance of hashing is superior to that of binary search of an array or a binary search tree, particularly if the load factor is less than 0.75
- However, the lower the load factor, the more empty storage cells
    - there are no empty cells in a sorted array
- A binary search tree requires three references per node (item, left subtree, right subtree), so more storage is required for a binary search tree than for a hash table with load factor 0.75

Storage Requirements for Open Addressing and Chaining

- For open addressing, the number of references to items (key-value pairs) is n (the size of the table)
- For chaining, the average number of nodes in a list is L (the load factor) and n is the number of table elements
  - Using the Java API `LinkedList`, there will be three references in each node (item, next, previous)
  - Using our own single linked list, we can reduce the references to two by eliminating the previous-element reference
  - Therefore, storage for $n + 2L$ references is needed

Example

Open addressing

- Assume 60,000 items in the hash table, and load factor of 0.75
- This requires a table of size 80,000 and results in an expected number of comparisons of 2.5

Chaining

- Calculating the table size *n* to get similar performance
  $2.5 = 1 + \mathrm{L}/2$
  $5.0 = 2 + \mathrm{L}$
  $3.0 = 60{,}000/\mathrm{n}$
  $\mathrm{n} = 20{,}000$
- A hash table of size 20,000 provides storage space for 20,000 references to lists
- There are 60,000 nodes in the table (one for each item)
- This requires storage for 140,000 references (2 x 60,000 + 20,000), which is 175% of the storage needed for open addressing