Hash Tables

CS151

HashTables

- A hash table is a form of a map that has better time complexity
- A hash table consists of
  - an array of size N
  - an associated hash function h that maps keys to integers in [0, N-1]
  - A "collision" handling scheme
- Hash Function
  - $h(x) = x\%N$ is such a function for integers
  - item (k, v) is stored at index $h(k)$
- Collision Handling
  - A "collision" occurs when two <u>**different**</u> keys hash to the same value

Hash Functions

- The goal of a hash function is to disperse the keys
- A hash function is usually specified as the composition of two functions:
  - hash code: key —> integers
  - compression: integers —> [0, N-1]
    - where the backing array is of size N

Char-by-char in String

- String s = “abc”;
  - s.charAt(0) == ‘a’;
  - s.charAt(0) == 97;
    - both are correct.

- Suppose Hash func is just add ASCII values of all chars in string.

Key	Char values	As integer
aba	97+98+97	292
baa	98+97+97	292
aab	97+97+98	292

ASCII
American Standard Code for Information Interchange.

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Hash Codes

- Why use ASCII values rather than a==0, b==1, …

Horner’s method:Convert any object to integer

Start with
an object, then just call
its toString

Almost any
prime number

```java
public BigInteger objectHasher(Object ob) {
    return stringHasher(ob.toString());
}
public BigInteger stringHasher(String ss) {
    BigInteger mul = BigInteger.valueOf(23);
    BigInteger ll = BigInteger.valueOf(0);
    for (int i=0; i<ss.length(); i++) {
        ll = ll.multiply(mul);
        ll = ll.add(BigInteger.valueOf(ss.charAt(i)));
    }
   return ll;
}
```

Handles really
large numbers

33^15=59938945498865420543457

Collisions

drawing 500 unique words from Oliver Twist and assuming a hashtable size of 1009, get these collisions

16 probable child when

42 fagins xxix importance that xv administering

104 stage pledge near

132 surgeon can night

271 things fang birth

341 alone sequel life

415 maylie check circumstances

418 mentioning containing growth

625 meet she first

732 there affording encounters

749 possible out acquainted

761 never xviii after goaded where

833 marks jew gentleman

985 adventures inseparable experience

Collisions

- Handling of collisions is one of the most important topics for hashtables
  - Approach 1:
    - Whenever you have a collision “Rehash”
      - make the table bigger
      - O(n) time so want to avoid
  - Approach 2
      - Separate Chaining
  - Approach 3
      - Probing

Separate Chaining

- Idea: each spot in hashtable holds a array list of key value pairs when the key maps to that hashvalue.
- Replace the item if the key is the same
- Otherwise, add to list
- Generally do not want more than about number of objects as size of table
- Chains can get long

Hash tables get crowded, chains get long

HT_SIZE=1009

Using unique words drawn from “Oliver Twist”.
Unique count at top of table

	278	473	1550	2510
0	762	622	210	87
1	217	308	342	198
2	29	73	252	268
3	1	5	136	208
4	0	1	55	140
5	0	0	9	70
6	0	0	4	26
7	0	0	1	10
8	0	0	0	2
9	0	0	0	0

Separate Chaining Example

- Suppose
  - hashtable size is 3
  - hashtable has lower case character keys and strings for values
  - h(x) = (x-97) % 3

<a,”Neville”>	(97-97)%3
<b,”Ray”>	(98-97)%3
<f, “Hurum”>	(102-97)%3
<g, “Quinn”>	(103-97)%3
<m, “Amina”>	(109-97)%3
<a, “Juno”>	(97-97)%3

Separate Chaining Code

```
public class SepChainHT<K,V> implements Map151Interface<K,V> {

    private Map151Impl<K,V>[] backingArray;

    private int count;

    public SepChainHT(int size) {
        count = 0;
        backingArray = (Map206<K, V>[]) new Map206[size];
    }

    private int h(K k) {
        return objectHasher(k);
    }
```

Separate Chaining Code

```
public void put(K key, V value) {
    int loc = h(key);
    if (backingArray[loc] == null) {
        backingArray[loc] = new Map206<>();
    }
    if (!backingArray[loc].containsKey(key)) {
        count++;
    }
    backingArray[loc].put(key, value);
}

public V get(K key) {
    int loc = h(key);
    if (backingArray[loc]==null) {
        return null;
    }
    return backingArray[loc].get(key);
}
```

```
public boolean containsKey(K key) {
    int loc = h(key);
    if (backingArray[loc] == null) {
        return false;
    }
    return backingArray[loc].containsKey(key);
}

public Set<K> keySet() {
    TreeSet<K> set = new TreeSet<>();
    for (int i = 0; i < backingArray.length; i++) {
        if (backingArray[i] != null)
            set.addAll(backingArray[i].keySet());
    }
    return set;
}
```

In class exercise

- Show the final contents of the hashtable using separate chaining assuming. Ie. show the contents of all chains
  - table size is 7
  - h(t) = t % 7
  - Data: <0,a> <32,b> <39,c> <12,d> <14,e> <35,f> <27,g> <13,h> <15,i> <5,j> <12,k> <13,l> <4,m> <0,n> <35,o>,<17,o>,<3,o>

- For a separate chaining hashtable that uses Map151 for its chains (as in the previous slide) write:
  - boolean containsKey(K key)
  - V get(K key)

Open Addressing
Linear Probing

- Store only <K,V> at each location in array
  - No awkward lists
- If key is different and location is in use then go to next spot in array
  - repeat until free location found

Linear Probing Example

- Suppose
  - hashtable size is 7
  - h(t)=t%7
  - add:
    - <3,A>
    - <10,B>
    - <17,C>
    - <24,Z>
    - <3,D>
    - <4,E>

Linear Probing

- Store only <K,V> at each location in array
- If key is different and location is in use then go to next spot in array
  - if key is same, replace value
  - repeat until free location found

Probing Distance

- Given a hash value $h(x)$, linear probing generates $h(x),\ h(x)+1, h(x)+2, \ldots$
  - Primary clustering – the bigger the cluster gets, the faster it grows
- Quadratic probing – $h(x),\ h(x)+1, h(x)+4,\ h(x)+9, \ldots$
  - Quadratic probing leads to secondary clustering, more subtle, not as dramatic, but still systematic
- Double hashing
  - Use a second hash function to determine jumps

Performance Analysis for probing

- In the worst case, searches, insertions and removals take $O(n)$ time
  - when all the keys collide
- The load factor α affects the performance of a hash table
  - expected number of probes for an insertion with open addressing is $\frac{1}{1-\alpha}$
- Expected time of all operations is $O(1)$ provided α is not close to 1
  - NOTE: cheating here O() is about true worst case

Open Addressing vs Chaining

- Probing is significantly faster in practice
- locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a list
- Efficient probing requires soft/lazy deletions – tombstoning
  - de-tombstoning