Hash Tables

CS206

March 9

Lab from Friday

- What is the time complexity, big O, O(?) of these 2 functions?

```
long complex(int[] data) {
    long res = 0;
    for (int i = 0; i < data.length / 2; i++) {              O(n) loop
        for (int j = data.length / 2; j < data.length; j++) {   O(n) loop
            res++;
        }
    }
    return res;
}
```

O(n^2)

```
long complex2(int[] data) {
    long res = 0;
    for (int i = 0; i < data.length; i++) {           O(n) loop
        for (int j = 1; j < data.length; j=j*2) {     O(lg n) loop
            res++;
        }
    }
    return res;
}
```

$O(n \lg_2 n)$

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
- $h(x) = x\%N$ is such a function for integers
  - item (k, v) is stored at index $h(k)$
- Collision Handling
  - A "collision" occurs when two different heys 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

Hash Codes

- Polynomial accumulation: partition bits of key into a sequence of components of fixed length $a_0 a_1 \ldots a_{n-1}$
  - For instance, if the key is a string, then a's are just the characters
    - convert characters to numbers such that a:97, … z:122 (The ASCII table)
- Evaluate the polynomial $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$
- Why not convert a=>0?

ASCII
American Standard Code for Information Interchange.

Dec	Hx	Oct	Char	
0	0	000	NUL	(null)
1	1	001	SOH	(start of heading)
2	2	002	STX	(start of text)
3	3	003	ETX	(end of text)
4	4	004	EOT	(end of transmission)
5	5	005	ENQ	(enquiry)
6	6	006	ACK	(acknowledge)
7	7	007	BEL	(bell)
8	8	010	BS	(backspace)
9	9	011	TAB	(horizontal tab)
10	A	012	LF	(NL line feed, new line)
11	B	013	VT	(vertical tab)
12	C	014	FF	(NP form feed, new page)
13	D	015	CR	(carriage return)
14	E	016	SO	(shift out)
15	F	017	SI	(shift in)
16	10	020	DLE	(data link escape)
17	11	021	DC1	(device control 1)
18	12	022	DC2	(device control 2)
19	13	023	DC3	(device control 3)
20	14	024	DC4	(device control 4)
21	15	025	NAK	(negative acknowledge)
22	16	026	SYN	(synchronous idle)
23	17	027	ETB	(end of trans. block)
24	18	030	CAN	(cancel)
25	19	031	EM	(end of medium)
26	1A	032	SUB	(substitute)
27	1B	033	ESC	(escape)
28	1C	034	FS	(file separator)
29	1D	035	GS	(group separator)
30	1E	036	RS	(record separator)
31	1F	037	US	(unit separator)

Dec	Hx	Oct	Html	Chr
32	20	040		Space
33	21	041	!	!
34	22	042	"	"
35	23	043	#	#
36	24	044	$	$
37	25	045	%	%
38	26	046	&	&
39	27	047	'	'
40	28	050	(	(
41	29	051	)	)
42	2A	052	*	*
43	2B	053	+	+
44	2C	054	,	,
45	2D	055	-	-
46	2E	056	.	.
47	2F	057	/	/
48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9
58	3A	072	:	:
59	3B	073	;	;
60	3C	074	<	<
61	3D	075	=	=
62	3E	076	>	>
63	3F	077	?	?

Dec	Hx	Oct	Html	Chr
64	40	100	@	@
65	41	101	A	A
66	42	102	B	B
67	43	103	C	C
68	44	104	D	D
69	45	105	E	E
70	46	106	F	F
71	47	107	G	G
72	48	110	H	H
73	49	111	I	I
74	4A	112	J	J
75	4B	113	K	K
76	4C	114	L	L
77	4D	115	M	M
78	4E	116	N	N
79	4F	117	O	O
80	50	120	P	P
81	51	121	Q	Q
82	52	122	R	R
83	53	123	S	S
84	54	124	T	T
85	55	125	U	U
86	56	126	V	V
87	57	127	W	W
88	58	130	X	X
89	59	131	Y	Y
90	5A	132	Z	Z
91	5B	133	[	[
92	5C	134	\	\
93	5D	135	]	]
94	5E	136	^	^
95	5F	137	_	_

Dec	Hx	Oct	Html	Chr
96	60	140	`	`
97	61	141	a	a
98	62	142	b	b
99	63	143	c	c
100	64	144	d	d
101	65	145	e	e
102	66	146	f	f
103	67	147	g	g
104	68	150	h	h
105	69	151	i	i
106	6A	152	j	j
107	6B	153	k	k
108	6C	154	l	l
109	6D	155	m	m
110	6E	156	n	n
111	6F	157	o	o
112	70	160	p	p
113	71	161	q	q
114	72	162	r	r
115	73	163	s	s
116	74	164	t	t
117	75	165	u	u
118	76	166	v	v
119	77	167	w	w
120	78	170	x	x
121	79	171	y	y
122	7A	172	z	z
123	7B	173	{	{
124	7C	174			|
125	7D	175	}	}
126	7E	176	~	~
127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Polynomial accumulation on Strings

```
static int POLY_MULT=33;
    public  int stringHasher(String ss) {
        BigInteger accumul = new BigInteger("0");
        for (int i=0; i<ss.length(); i++) {
            BigInteger bb =
BigInteger.valueOf(POLY_MULT).pow(i).multiply(BigInteger.valueOf((int)ss.
charAt(i)));
            accumul = accumul.add(bb);
        }
        accumul =
accumul.mod( BigInteger.valueOf( backingArray.length ));
        return accumul.intValue();
    }
```

Recommended by textbook

Handles integers of unbounded size

33^15=59938945498865420543457

The ASCII value

Array storing the hashtable

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 Code

```
public class SepChainHT<K,V> implements Map206Interface<K,V> {

    private Map206<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 stringHasher(k.toString()) % backingArray.length;
    }

    // stringHasher – as on slide 5
```

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 Map206 for its chains (as in the previous slide) write:
  - boolean containsKey(K key)
  - V get(K key)

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;
}
```

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 linked list
- Efficient probing requires soft/lazy deletions – tombstoning
  - de-tombstoning

Lab

- Show the final contents of the hashtable using linear probing assuming
  - table size is 13
  - h(t) = t % 13
- 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>
- What is the most number of steps you needed to take to find a free location?