Dictionaries
Hash Tables

Generic Inner Class!

```
public class OutCLGen<R,S> {
    /**
     * The inner class, Generically
     */
    private class InnCl<Y,Z> {
        private Y value1; // a value
        private Z value2; // another value

        public InnCl(Y v1, Z v2) {
            this.value1 = v1;
            this.value2 = v2;
        }

        public String toString() {
            return value1 + " " + value2;
        }
    }
```

Inner class has different generic parameters than its surrounding class. Realistically, they will almost always be the same, but this allows for difference

```
public void worker(R rValue, S sValue) {
    InnCl<String, String> icl1 = new InnCl<>(“Alice”, "Bob");
    InnCl<R,S> icl2 = new InnCl<>(rValue, sValue);
    icl1.value1 = 3;
    System.out.println(icl1 + "\n" + icl2);
}

public static void main(String[] args) {
    OutCLGen<Integer, String> ocl = new OutCLGen<>();
    ocl.worker(42, "Carol");
}
```

Dictionary (Map)

- A searchable collection of key-value pairs
  - A lot of this course will be involve key value pairs
    - A lot of life is about key value pairs
      - SSN, tax history
      - BMID no, student record
      - …..
- Multiple entries with the same key are not allowed
- AKA associative array (perl)

What do you do with dictionaries (Physical)

- Look up based on a key item (word)
  - to get defintion
- Add items (word and definition)
- Remove Items (word)
- Others??

Count, list keys, iterators, contains, clear

Map Interface

- https://docs.oracle.com/javase/7/docs/api/java/util/Map.html

```
public interface Map151<K, V> {
    public void put(K key, V val);
    public V get(K key);
    public boolean containsKey(K key);
    public int size();
    public List151Impl<K> keySet();
}
```

Overwrite if key is already in map

Alternately return an ArrayList or a Set. The java definition returns a Set.

Map Implementation

```
public class Map151Impl<K,V> implements Map151<K,V>{

    private ArrayList<Pair<K,V>> underlying = new ArrayList<>();

    private class Pair<L,W> {
        public L ky;
        public W vl;
        Pair(L key, W value) {
            ky=key;
            vl=value;
        }
    }
```

Uses Java standard class rather than List151Impl

Map Implementation (pt 2)

```
public boolean containsKey(K key) {
     return null != getKV(key);
 }

 private Pair<K,V> getKV(K ky) {
     for (Pair<K,V> pair : underlying) {
         if (pair.key.equals(ky)) {
             return pair;
         }
     }return null;
 }

 public int size() {
     return underlying.size();
 }

 public List151Impl<K> keySet() {
       // Write Me
 }
```

```
public void put(K key, V val) {
    Pair<K,V> pair =getKV(key);
    if (pair==null) {
        Pair<K,V> np = new Pair<>(key, val);
        underlying.add(np);
    } else {
        pair.value=val;
    }
}

public V get(K key) {
    Pair<K,V> pair = getKV(key);
    if (pair!=null)
        return pair.value;
    return null;
}
```

In the book's parlance, this is an unsorted, ArrayList-based dictionary.

Maps - time complexity

- Put
  - need to search the existing items (iContainsKey)
  - so time to add 1 item in O(n)
    - if you are adding n items the you are doing an O(n) process n times so your time is O(n^2)
- Get
  - similar analysis so O(n^2) to do n gets
- Conclusion: maps get slow when n gets large.

Using Maps to track stock positions

- Suppose you are an active stock trader, buying and selling all the time.
- One challenge is simply keeping track of how much you have of what
- Further suppose,
  - you have a csv file
    - STOCK ID, AMT
  - Idea
    - Use ReadCSV and Map151 to track

StockTracker

```java
Map151Impl<String, Integer> positions;

public StockTracker() {
        positions = new Map151Impl<>();
   }

public void track(String filename) {
        ReadCSV csvReader;
        try {
            csvReader = new ReadCSV(filename, 4000);
        }
        catch (IOException ioe) {
            System.err.println("Ending. Cannot read. " + ioe.toString());
            return;
        }

        for (String[] datum : csvReader) {
            int newamt = Integer.parseInt(datum[1]);
            Integer oldVal = positions.get(datum[0]);
            if (oldVal != null) {
                newamt += oldVal; }
            positions.put(datum[0], newamt);
        }
        System.out.println(positions);
   }
```

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]
    - For example, given integer keys, $h(x) = x\%N$ is a hash function
- pair(K,V) is stored at index $h(k)$
  - pair is an inner class a la Map

Simple Hashtable Implementation

ridiculously simple

```
public class SimpleHT {
    private class Pair {
        // the key.  Once set it cannot be changed
        public final Integer key;
        // the value
        public String value;
        //Create a key value pair.
        Pair(Integer ky, String val) {
            key = ky;
            value = val;
        }
    }

    private Pair[] backingArray = new Pair[4];

    private int h(int k) {
        return k%4;
    }
    public void put(Integer key, String value) {
        backingArray[h(key)] = new Pair(key, value);
    }
    public String get(Integer key) {
        return backingArray[h(key)].value;
    }
}
```

NO generics … too simple

4!!!???

SimpleHT Example/Test

```
public static void main(String[] args) {
        SimpleHT sht = new SimpleHT();
        for (int i=0; i<10; i++) {
            System.out.println("adding item with key=" + i + "
value=" + String.format("%c", 'a'+i));
            sht.put(i, String.format("%c", 'a'+i));
        }
        for (int i=0; i<10; i++)
            System.out.println("getting key="+i+"
value="+sht.get(i));
    }
```

Two problems:

1. **a poor hashing function.**
2. **Storing more than there is room for**

```
adding item with key=0 value=a
adding item with key=1 value=b
adding item with key=2 value=c
adding item with key=3 value=d
adding item with key=4 value=e
adding item with key=5 value=f
adding item with key=6 value=g
adding item with key=7 value=h
adding item with key=8 value=i
adding item with key=9 value=j
getting key=0  value=i
getting key=1  value=j
getting key=2  value=g
getting key=3  value=h
getting key=4  value=i
getting key=5  value=j
getting key=6  value=g
getting key=7  value=h
getting key=8  value=i
getting key=9  value=j
```

Hash Functions

- The goal is to “disperse” the keys in an appropriately random way
  - A hash function is usually specified as the composition of two functions:
  - hash code: key —> integers
  - compression: integers —> [0, N-1]

see SepChainHT.java

Hash Functions

- Goals:
  - Minimize collisions
  - Quickly transform key to integer
- Common Approach for non-integer keys:
  - 1. Transform key into String
  - 2. Do something character-by-character on string

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 Table

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

www.alpharithms.com

Horner's method:Convert any object to integer

Start with an object, then just call its toString

Almost any prime number

Handles really large numbers

```
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;
}
```

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
  - Rehashing
    - make the table bigger
      - O(n) time so want to avoid
      - Also, simply making table bigger does not always eliminate collisions
  - Alternative to rehashing
    - Separate Chaining
    - Probing

Separate Chaining

- Idea: each spot in hashtable holds a 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

0	762
1	217
2	29
3	1
4	0
5	0
6	0
7	0
8	0
9	0

473

0	622
1	308
2	73
3	5
4	1
5	0
6	0
7	0
8	0
9	0

1550

0	210
1	342
2	252
3	136
4	55
5	9
6	4
7	1
8	0
9	0

2510

0	87
1	198
2	268
3	208
4	140
5	70
6	26
7	10
8	2
9	0

In class exercise

- Show the final contents of the hashtable using separate chaining assuming
  - 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>
- What is the longest chain?

Open Addressing
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

- If spot is occupied, go to spot+1, spot+2, …
- Suppose
  - hashtable size is 7
  - h(t)=t%7
  - add:
    - <3,A>
    - <10,B>
    - <17,C>
    - <24,Z>
    - <3,D>
    - <4,E>

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

In class exercise

- 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?

Using Hashtables

- No worries about hashing functions, rehashing, …
  - Someone else responsibility
- Example: who is visiting my site, and how often?
  - for instance, hackers?
- web servers keep access logs
- java.util.HashMap