

Data Structures and Algorithms in Java

Searching: Hash Tables

Outline

① Hashing

② Separate-Chaining Symbol Table

Hashing

Hashing

The basic idea is to save items in a key-indexed array, where the index is a function of the key

Hashing

The basic idea is to save items in a key-indexed array, where the index is a function of the key

Hash function provides a method for computing an array index from a key

Hashing

The basic idea is to save items in a key-indexed array, where the index is a function of the key

Hash function provides a method for computing an array index from a key

Issues

- Computing the hash function
- Equality test: method for checking whether two keys are equal
- Collision resolution: algorithm and data structure to handle two keys that hash to the same array index

Hashing

The basic idea is to save items in a key-indexed array, where the index is a function of the key

Hash function provides a method for computing an array index from a key

Issues

- Computing the hash function
- Equality test: method for checking whether two keys are equal
- Collision resolution: algorithm and data structure to handle two keys that hash to the same array index

Classic space-time tradeoff

- No space limitation: trivial hash function with key as index
- No time limitation: trivial collision resolution with sequential search
- Space and time limitations: hashing (the real world)

Hashing

Hashing

Idealistic goal: scramble the keys uniformly to produce a table index that is

- Efficiently computable
- Equally likely for each key

Hashing

Idealistic goal: scramble the keys uniformly to produce a table index that is

- Efficiently computable
- Equally likely for each key

Example 1: phone numbers

- Bad: first three digits
- Better: last three digits

Hashing

Idealistic goal: scramble the keys uniformly to produce a table index that is

- Efficiently computable
- Equally likely for each key

Example 1: phone numbers

- Bad: first three digits
- Better: last three digits

Example 2: social security numbers

- Bad: first three digits
- Better: last four digits

Hashing

Idealistic goal: scramble the keys uniformly to produce a table index that is

- Efficiently computable
- Equally likely for each key

Example 1: phone numbers

- Bad: first three digits
- Better: last three digits

Example 2: social security numbers

- Bad: first three digits
- Better: last four digits

Practical challenge: need different approach for each type of key

Hashing

Hashing

Java's hash code conventions

- All Java classes inherit a method `hashCode()`, which returns a 32-bit `int`
- Requirement: if `x.equals(y)`, then `x.hashCode() == y.hashCode()`
- Highly desirable: if `!x.equals(y)`, then `x.hashCode() != y.hashCode()`
- Default implementation: return memory address of `x`
- Legal (but poor) implementation: always return 17
- Customized implementations: `Integer`, `Double`, `String`, `File`, `URL`, `Date`, ...
- User-defined types: users are on their own

Hashing

Hashing

Java library implementations

```
public final class Boolean {  
    private final boolean value;  
  
    public int hashCode() { return value ? 1231 : 1237; }  
}
```


Hashing

Java library implementations

```
public final class Boolean {  
    private final boolean value;  
  
    public int hashCode() { return value ? 1231 : 1237; }  
}
```

```
public final class Integer {  
    private final int value;  
  
    public int hashCode() { return value; }  
}
```

Hashing

Java library implementations

```
public final class Boolean {  
    private final boolean value;  
  
    public int hashCode() { return value ? 1231 : 1237; }  
}
```

```
public final class Integer {  
    private final int value;  
  
    public int hashCode() { return value; }  
}
```

```
public final class Double {  
    private final double value;  
  
    public int hashCode() {  
        long bits = doubleToLongBits(value);  
        return (int) (bits ^ (bits >>> 32));  
    }  
}
```

Hashing

Java library implementations

```
public final class Boolean {
    private final boolean value;

    public int hashCode() { return value ? 1231 : 1237; }
}
```

```
public final class Integer {
    private final int value;

    public int hashCode() { return value; }
}
```

```
public final class Double {
    private final double value;

    public int hashCode() {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

```
public final class String {
    private int hash = 0;
    private final char[] s;

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

Hashing

Hashing

Implementing hash code for user-defined types

```
public final class Transaction implements Comparable<Transaction> {  
    private final String who;  
    private final Date when;  
    private final double amount;  
  
    public int hashCode() {  
        int hash = 17;  
        hash = 31 * hash + who.hashCode();  
        hash = 31 * hash + when.hashCode();  
        hash = 31 * hash + ((Double) amount).hashCode();  
        return hash;  
    }  
}
```

Hashing

Implementing hash code for user-defined types

```
public final class Transaction implements Comparable<Transaction> {  
    private final String who;  
    private final Date when;  
    private final double amount;  
  
    public int hashCode() {  
        int hash = 17;  
        hash = 31 * hash + who.hashCode();  
        hash = 31 * hash + when.hashCode();  
        hash = 31 * hash + ((Double) amount).hashCode();  
        return hash;  
    }  
}
```

Hash code design

- Combine each significant field using the $31x + y$ rule
- If field is a primitive type, use wrapper type `hashCode()`
- If field is `null`, return 0
- If field is a reference type, use `hashCode()`
- If field is an array, apply to each entry

Hashing

Hashing

Modular hashing

- Hash code: an `int` between -2^{31} and $2^{31} - 1$
- Hash function: an `int` between 0 and $m - 1$ (for use as array index)

Hashing

Modular hashing

- Hash code: an `int` between -2^{31} and $2^{31} - 1$
- Hash function: an `int` between 0 and $m - 1$ (for use as array index)

Implementation

```
private int hash(Key key) {  
    return (key.hashCode() & 0x7fffffff) % m;  
}
```

Hashing

Modular hashing

- Hash code: an `int` between -2^{31} and $2^{31} - 1$
- Hash function: an `int` between 0 and $m - 1$ (for use as array index)

Implementation

```
private int hash(Key key) {  
    return (key.hashCode() & 0x7fffffff) % m;  
}
```

Uniform hashing assumption: each key is equally likely to hash to an integer between 0 and $m - 1$

Hashing

Modular hashing

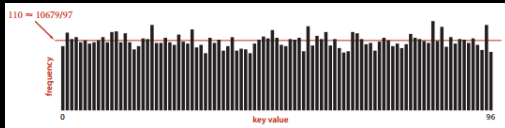
- Hash code: an `int` between -2^{31} and $2^{31} - 1$
- Hash function: an `int` between 0 and $m - 1$ (for use as array index)

Implementation

```
private int hash(Key key) {  
    return (key.hashCode() & 0x7fffffff) % m;  
}
```

Uniform hashing assumption: each key is equally likely to hash to an integer between 0 and $m - 1$

Example (hash value frequencies for words in Tale of Two Cities; 10,679 keys; $m = 97$)



Hashing

Modular hashing

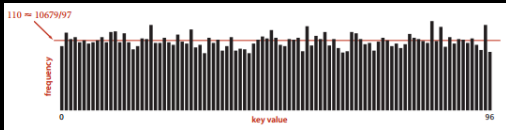
- Hash code: an `int` between -2^{31} and $2^{31} - 1$
- Hash function: an `int` between 0 and $m - 1$ (for use as array index)

Implementation

```
private int hash(Key key) {  
    return (key.hashCode() & 0x7fffffff) % m;  
}
```

Uniform hashing assumption: each key is equally likely to hash to an integer between 0 and $m - 1$

Example (hash value frequencies for words in Tale of Two Cities; 10,679 keys; $m = 97$)



Collision: two distinct keys hash to the same index

- Can't avoid collisions unless you have a ridiculous amount of memory
- Collisions are evenly distributed
- Challenge: deal with collisions efficiently

Separate-Chaining Symbol Table

Symbol	Address	Next
1	0	2
2	1	3
3	2	4
4	3	5
5	4	6
6	5	7
7	6	8
8	7	9
9	8	10
10	9	11
11	10	12
12	11	13
13	12	14
14	13	15
15	14	16
16	15	17
17	16	18
18	17	19
19	18	20
20	19	21
21	20	22
22	21	23
23	22	24
24	23	25
25	24	26
26	25	27
27	26	28
28	27	29
29	28	30
30	29	31
31	30	32
32	31	33
33	32	34
34	33	35
35	34	36
36	35	37
37	36	38
38	37	39
39	38	40
40	39	41
41	40	42
42	41	43
43	42	44
44	43	45
45	44	46
46	45	47
47	46	48
48	47	49
49	48	50
50	49	51
51	50	52
52	51	53
53	52	54
54	53	55
55	54	56
56	55	57
57	56	58
58	57	59
59	58	60
60	59	61
61	60	62
62	61	63
63	62	64
64	63	65
65	64	66
66	65	67
67	66	68
68	67	69
69	68	70
70	69	71
71	70	72
72	71	73
73	72	74
74	73	75
75	74	76
76	75	77
77	76	78
78	77	79
79	78	80
80	79	81
81	80	82
82	81	83
83	82	84
84	83	85
85	84	86
86	85	87
87	86	88
88	87	89
89	88	90
90	89	91
91	90	92
92	91	93
93	92	94
94	93	95
95	94	96
96	95	97
97	96	98
98	97	99
99	98	100
100	99	101
101	100	102
102	101	103
103	102	104
104	103	105
105	104	106
106	105	107
107	106	108
108	107	109
109	108	110
110	109	111
111	110	112
112	111	113
113	112	114
114	113	115
115	114	116
116	115	117
117	116	118
118	117	119
119	118	120
120	119	121
121	120	122
122	121	123
123	122	124
124	123	125
125	124	126
126	125	127
127	126	128
128	127	129
129	128	130
130	129	131
131	130	132
132	131	133
133	132	134
134	133	135
135	134	136
136	135	137
137	136	138
138	137	139
139	138	140
140	139	141
141	140	142
142	141	143
143	142	144
144	143	145
145	144	146
146	145	147
147	146	148
148	147	149
149	148	150
150	149	151
151	150	152
152	151	153
153	152	154
154	153	155
155	154	156
156	155	157
157	156	158
158	157	159
159	158	160
160	159	161
161	160	162
162	161	163
163	162	164
164	163	165
165	164	166
166	165	167
167	166	168
168	167	169
169	168	170
170	169	171
171	170	172
172	171	173
173	172	174
174	173	175
175	174	176
176	175	177
177	176	178
178	177	179
179	178	180
180	179	181
181	180	182
182	181	183
183	182	184
184	183	185
185	184	186
186	185	187
187	186	188
188	187	189
189	188	190
190	189	191
191	190	192
192	191	193
193	192	194
194	193	195
195	194	196
196	195	197
197	196	198
198	197	199
199	198	200
200	199	201
201	200	202
202	201	203
203	202	204
204	203	205
205	204	206
206	205	207
207	206	208
208	207	209
209	208	210
210	209	211
211	210	212
212	211	213
213	212	214
214	213	215
215	214	216
216	215	217
217	216	218
218	217	219
219	218	220
220	219	221
221	220	222
222	221	223
223	222	224
224	223	225
225	224	226
226	225	227
227	226	228
228	227	229
229	228	230
230	229	231
231	230	232
232	231	233
233	232	234
234	233	235
235	234	236
236	235	237
237	236	238
238	237	239
239	238	240
240	239	241
241	240	242
242	241	243
243	242	244
244	243	245
245	244	246
246	245	247
247	246	248
248	247	249
249	248	250
250	249	251
251	250	252
252	251	253
253	252	254
254	253	255
255	254	256
256	255	257
257	256	258
258	257	259
259	258	260
260	259	261
261	260	262
262	261	263
263	262	264
264	263	265
265	264	266
266	265	267
267	266	268
268	267	269
269	268	270
270	269	271
271	270	272
272	271	273
273	272	274
274	273	275
275	274	276
276	275	277
277	276	278
278	277	279
279	278	280
280	279	281
281	280	282
282	281	283
283	282	284
284	283	285
285	284	286
286	285	287
287	286	288
288	287	289
289	288	290
290	289	291
291	290	292
292	291	293
293	292	294
294	293	295
295	294	296
296	295	297
297	296	298
298	297	299
299	298	300
300	299	301
301	300	302
302	301	303
303	302	304
304	303	305
305	304	306
306	305	307
307	306	308
308	307	309
309	308	310
310	309	311
311	310	312
312	311	313
313	312	314
314	313	315
315	314	316
316	315	317
317	316	318
318	317	319
319	318	320
320	319	321
321	320	322
322	321	323
323	322	324
324	323	325
325	324	326
326	325	327
327	326	328
328	327	329
329	328	330
330	329	331
331	330	332
332	331	333
333	332	334
334	333	335
335	334	336
336	335	337
337	336	338
338	337	339
339	338	340
340	339	341
341	340	342
342	341	343
343	342	344
344	343	345
345	344	346
346	345	347
347	346	348
348	347	349
349	348	350
350	349	351
351	350	352
352	351	353
353	352	354
354	353	355
355	354	356
356	355	357
357	356	358
358	357	359
359	358	360
360	359	361
361	360	362
362	361	363
363	362	364
364	363	365
365	364	366
366	365	367
367	366	368
368	367	369
369	368	370
370	369	371
371	370	372
372	371	373
373	372	374
374	373	375
375	374	376
376	375	377
377	376	378
378	377	379
379	378	380
380	379	381
381	380	382
382	381	383
383	382	384
384	383	385
385	384	386
386	385	387
387	386	388
388	387	389
389	388	390
390	389	391
391	390	392
392	391	393
393	392	394
394	393	395
395	394	396
396	395	397
397	396	398
398	397	399
399	398	400
400	399	401
401	400	402
402	401	403
403	402	404
404	403	405
405	404	406
406	405	407
407	406	408
408	407	409
409	408	410
410	409	411
411	410	412
412	411	413
413	412	414
414	413	415
415	414	416
416	415	417
417	416	418
418	417	419
419	418	420
420	419	421
421	420	422
422	421	423
423	422	424
424	423	425
425	424	426
426	425	427
427	426	428
428	427	429
429	428	430
430	429	431
431	430	432
432	431	433
433	432	434
434	433	435
435	434	436
436	435	437
437	436	438
438	437	439
439	438	440
440	439	441
441	440	442
442	441	443
443	442	444
444	443	445
445	444	446
446	445	447
447	446	448
448	447	449
449	448	450
450	449	451
451	450	4

Separate-Chaining Symbol Table

Use an array of $m < n$ linked lists

- Hash: map key to integer $i \in [0, m - 1]$
- Insert: put at front of i th chain (if not already there)
- Search: need to search only the i th chain

value

0

1

2

3

4

5

6

7

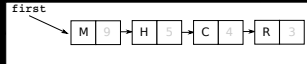
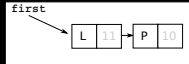
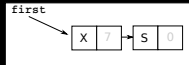
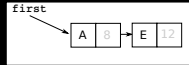
8

9

10

11

12



Separate-Chaining Symbol Table

Use an array of $m < n$ linked lists

- Hash: map key to integer $i \in [0, m - 1]$
- Insert: put at front of i th chain (if not already there)
- Search: need to search only the i th chain

value

0

1

2

3

4

5

6

7

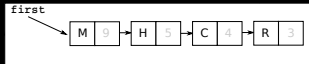
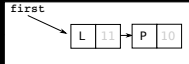
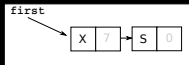
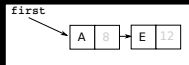
8

9

10

11

12



The ratio n/m is called the load factor and is denoted by α , and is interpreted as the average number of keys per list

Separate-Chaining Symbol Table

Symbol	Address	Next
1	0	2
2	1	3
3	2	4
4	3	5
5	4	6
6	5	7
7	6	8
8	7	9
9	8	10
10	9	11
11	10	12
12	11	13
13	12	14
14	13	15
15	14	16
16	15	17
17	16	18
18	17	19
19	18	20
20	19	21
21	20	22
22	21	23
23	22	24
24	23	25
25	24	26
26	25	27
27	26	28
28	27	29
29	28	30
30	29	31
31	30	32
32	31	33
33	32	34
34	33	35
35	34	36
36	35	37
37	36	38
38	37	39
39	38	40
40	39	41
41	40	42
42	41	43
43	42	44
44	43	45
45	44	46
46	45	47
47	46	48
48	47	49
49	48	50
50	49	51
51	50	52
52	51	53
53	52	54
54	53	55
55	54	56
56	55	57
57	56	58
58	57	59
59	58	60
60	59	61
61	60	62
62	61	63
63	62	64
64	63	65
65	64	66
66	65	67
67	66	68
68	67	69
69	68	70
70	69	71
71	70	72
72	71	73
73	72	74
74	73	75
75	74	76
76	75	77
77	76	78
78	77	79
79	78	80
80	79	81
81	80	82
82	81	83
83	82	84
84	83	85
85	84	86
86	85	87
87	86	88
88	87	89
89	88	90
90	89	91
91	90	92
92	91	93
93	92	94
94	93	95
95	94	96
96	95	97
97	96	98
98	97	99
99	98	100
100	99	101
101	100	102
102	101	103
103	102	104
104	103	105
105	104	106
106	105	107
107	106	108
108	107	109
109	108	110
110	109	111
111	110	112
112	111	113
113	112	114
114	113	115
115	114	116
116	115	117
117	116	118
118	117	119
119	118	120
120	119	121
121	120	122
122	121	123
123	122	124
124	123	125
125	124	126
126	125	127
127	126	128
128	127	129
129	128	130
130	129	131
131	130	132
132	131	133
133	132	134
134	133	135
135	134	136
136	135	137
137	136	138
138	137	139
139	138	140
140	139	141
141	140	142
142	141	143
143	142	144
144	143	145
145	144	146
146	145	147
147	146	148
148	147	149
149	148	150
150	149	151
151	150	152
152	151	153
153	152	154
154	153	155
155	154	156
156	155	157
157	156	158
158	157	159
159	158	160
160	159	161
161	160	162
162	161	163
163	162	164
164	163	165
165	164	166
166	165	167
167	166	168
168	167	169
169	168	170
170	169	171
171	170	172
172	171	173
173	172	174
174	173	175
175	174	176
176	175	177
177	176	178
178	177	179
179	178	180
180	179	181
181	180	182
182	181	183
183	182	184
184	183	185
185	184	186
186	185	187
187	186	188
188	187	189
189	188	190
190	189	191
191	190	192
192	191	193
193	192	194
194	193	195
195	194	196
196	195	197
197	196	198
198	197	199
199	198	200
200	199	201
201	200	202
202	201	203
203	202	204
204	203	205
205	204	206
206	205	207
207	206	208
208	207	209
209	208	210
210	209	211
211	210	212
212	211	213
213	212	214
214	213	215
215	214	216
216	215	217
217	216	218
218	217	219
219	218	220
220	219	221
221	220	222
222	221	223
223	222	224
224	223	225
225	224	226
226	225	227
227	226	228
228	227	229
229	228	230
230	229	231
231	230	232
232	231	233
233	232	234
234	233	235
235	234	236
236	235	237
237	236	238
238	237	239
239	238	240
240	239	241
241	240	242
242	241	243
243	242	244
244	243	245
245	244	246
246	245	247
247	246	248
248	247	249
249	248	250
250	249	251
251	250	252
252	251	253
253	252	254
254	253	255
255	254	256
256	255	257
257	256	258
258	257	259
259	258	260
260	259	261
261	260	262
262	261	263
263	262	264
264	263	265
265	264	266
266	265	267
267	266	268
268	267	269
269	268	270
270	269	271
271	270	272
272	271	273
273	272	274
274	273	275
275	274	276
276	275	277
277	276	278
278	277	279
279	278	280
280	279	281
281	280	282
282	281	283
283	282	284
284	283	285
285	284	286
286	285	287
287	286	288
288	287	289
289	288	290
290	289	291
291	290	292
292	291	293
293	292	294
294	293	295
295	294	296
296	295	297
297	296	298
298	297	299
299	298	300
300	299	301
301	300	302
302	301	303
303	302	304
304	303	305
305	304	306
306	305	307
307	306	308
308	307	309
309	308	310
310	309	311
311	310	312
312	311	313
313	312	314
314	313	315
315	314	316
316	315	317
317	316	318
318	317	319
319	318	320
320	319	321
321	320	322
322	321	323
323	322	324
324	323	325
325	324	326
326	325	327
327	326	328
328	327	329
329	328	330
330	329	331
331	330	332
332	331	333
333	332	334
334	333	335
335	334	336
336	335	337
337	336	338
338	337	339
339	338	340
340	339	341
341	340	342
342	341	343
343	342	344
344	343	345
345	344	346
346	345	347
347	346	348
348	347	349
349	348	350
350	349	351
351	350	352
352	351	353
353	352	354
354	353	355
355	354	356
356	355	357
357	356	358
358	357	359
359	358	360
360	359	361
361	360	362
362	361	363
363	362	364
364	363	365
365	364	366
366	365	367
367	366	368
368	367	369
369	368	370
370	369	371
371	370	372
372	371	373
373	372	374
374	373	375
375	374	376
376	375	377
377	376	378
378	377	379
379	378	380
380	379	381
381	380	382
382	381	383
383	382	384
384	383	385
385	384	386
386	385	387
387	386	388
388	387	389
389	388	390
390	389	391
391	390	392
392	391	393
393	392	394
394	393	395
395	394	396
396	395	397
397	396	398
398	397	399
399	398	400
400	399	401
401	400	402
402	401	403
403	402	404
404	403	405
405	404	406
406	405	407
407	406	408
408	407	409
409	408	410
410	409	411
411	410	412
412	411	413
413	412	414
414	413	415
415	414	416
416	415	417
417	416	418
418	417	419
419	418	420
420	419	421
421	420	422
422	421	423
423	422	424
424	423	425
425	424	426
426	425	427
427	426	428
428	427	429
429	428	430
430	429	431
431	430	432
432	431	433
433	432	434
434	433	435
435	434	436
436	435	437
437	436	438
438	437	439
439	438	440
440	439	441
441	440	442
442	441	443
443	442	444
444	443	445
445	444	446
446	445	447
447	446	448
448	447	449
449	448	450
450	449	451
451	450	4

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Consequence: number of probes for search/insert is proportional to α

- m too large \implies too many empty chains
- m too small \implies chains too long

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Consequence: number of probes for search/insert is proportional to α

- m too large \implies too many empty chains
- m too small \implies chains too long

Goal: $\alpha = \text{constant}$

- Double the size of array when $\alpha \geq 10$
- Halve the size of array when $\alpha \leq 2$
- Need to rehash all keys when resizing

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Consequence: number of probes for search/insert is proportional to α

- m too large \implies too many empty chains
- m too small \implies chains too long

Goal: $\alpha = \text{constant}$

- Double the size of array when $\alpha \geq 10$
- Halve the size of array when $\alpha \leq 2$
- Need to rehash all keys when resizing

Deleting a key (and its associated value) is easy — need only consider chain containing key

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Consequence: number of probes for search/insert is proportional to α

- m too large \implies too many empty chains
- m too small \implies chains too long

Goal: $\alpha = \text{constant}$

- Double the size of array when $\alpha \geq 10$
- Halve the size of array when $\alpha \leq 2$
- Need to rehash all keys when resizing

Deleting a key (and its associated value) is easy — need only consider chain containing key

The cost of search, insert, and delete, under the uniform hashing assumption, is constant (between 3 and 5)

Separate-Chaining Symbol Table

Symbol	Address	Next
1	0	2
2	1	3
3	2	4
4	3	5
5	4	6
6	5	7
7	6	8
8	7	9
9	8	10
10	9	11
11	10	12
12	11	13
13	12	14
14	13	15
15	14	16
16	15	17
17	16	18
18	17	19
19	18	20
20	19	21
21	20	22
22	21	23
23	22	24
24	23	25
25	24	26
26	25	27
27	26	28
28	27	29
29	28	30
30	29	31
31	30	32
32	31	33
33	32	34
34	33	35
35	34	36
36	35	37
37	36	38
38	37	39
39	38	40
40	39	41
41	40	42
42	41	43
43	42	44
44	43	45
45	44	46
46	45	47
47	46	48
48	47	49
49	48	50
50	49	51
51	50	52
52	51	53
53	52	54
54	53	55
55	54	56
56	55	57
57	56	58
58	57	59
59	58	60
60	59	61
61	60	62
62	61	63
63	62	64
64	63	65
65	64	66
66	65	67
67	66	68
68	67	69
69	68	70
70	69	71
71	70	72
72	71	73
73	72	74
74	73	75
75	74	76
76	75	77
77	76	78
78	77	79
79	78	80
80	79	81
81	80	82
82	81	83
83	82	84
84	83	85
85	84	86
86	85	87
87	86	88
88	87	89
89	88	90
90	89	91
91	90	92
92	91	93
93	92	94
94	93	95
95	94	96
96	95	97
97	96	98
98	97	99
99	98	100
100	99	101
101	100	102
102	101	103
103	102	104
104	103	105
105	104	106
106	105	107
107	106	108
108	107	109
109	108	110
110	109	111
111	110	112
112	111	113
113	112	114
114	113	115
115	114	116
116	115	117
117	116	118
118	117	119
119	118	120
120	119	121
121	120	122
122	121	123
123	122	124
124	123	125
125	124	126
126	125	127
127	126	128
128	127	129
129	128	130
130	129	131
131	130	132
132	131	133
133	132	134
134	133	135
135	134	136
136	135	137
137	136	138
138	137	139
139	138	140
140	139	141
141	140	142
142	141	143
143	142	144
144	143	145
145	144	146
146	145	147
147	146	148
148	147	149
149	148	150
150	149	151
151	150	152
152	151	153
153	152	154
154	153	155
155	154	156
156	155	157
157	156	158
158	157	159
159	158	160
160	159	161
161	160	162
162	161	163
163	162	164
164	163	165
165	164	166
166	165	167
167	166	168
168	167	169
169	168	170
170	169	171
171	170	172
172	171	173
173	172	174
174	173	175
175	174	176
176	175	177
177	176	178
178	177	179
179	178	180
180	179	181
181	180	182
182	181	183
183	182	184
184	183	185
185	184	186
186	185	187
187	186	188
188	187	189
189	188	190
190	189	191
191	190	192
192	191	193
193	192	194
194	193	195
195	194	196
196	195	197
197	196	198
198	197	199
199	198	200
200	199	201
201	200	202
202	201	203
203	202	204
204	203	205
205	204	206
206	205	207
207	206	208
208	207	209
209	208	210
210	209	211
211	210	212
212	211	213
213	212	214
214	213	215
215	214	216
216	215	217
217	216	218
218	217	219
219	218	220
220	219	221
221	220	222
222	221	223
223	222	224
224	223	225
225	224	226
226	225	227
227	226	228
228	227	229
229	228	230
230	229	231
231	230	232
232	231	233
233	232	234
234	233	235
235	234	236
236	235	237
237	236	238
238	237	239
239	238	240
240	239	241
241	240	242
242	241	243
243	242	244
244	243	245
245	244	246
246	245	247
247	246	248
248	247	249
249	248	250
250	249	251
251	250	252
252	251	253
253	252	254
254	253	255
255	254	256
256	255	257
257	256	258
258	257	259
259	258	260
260	259	261
261	260	262
262	261	263
263	262	264
264	263	265
265	264	266
266	265	267
267	266	268
268	267	269
269	268	270
270	269	271
271	270	272
272	271	273
273	272	274
274	273	275
275	274	276
276	275	277
277	276	278
278	277	279
279	278	280
280	279	281
281	280	282
282	281	283
283	282	284
284	283	285
285	284	286
286	285	287
287	286	288
288	287	289
289	288	290
290	289	291
291	290	292
292	291	293
293	292	294
294	293	295
295	294	296
296	295	297
297	296	298
298	297	299
299	298	300
300	299	301
301	300	302
302	301	303
303	302	304
304	303	305
305	304	306
306	305	307
307	306	308
308	307	309
309	308	310
310	309	311
311	310	312
312	311	313
313	312	314
314	313	315
315	314	316
316	315	317
317	316	318
318	317	319
319	318	320
320	319	321
321	320	322
322	321	323
323	322	324
324	323	325
325	324	326
326	325	327
327	326	328
328	327	329
329	328	330
330	329	331
331	330	332
332	331	333
333	332	334
334	333	335
335	334	336
336	335	337
337	336	338
338	337	339
339	338	340
340	339	341
341	340	342
342	341	343
343	342	344
344	343	345
345	344	346
346	345	347
347	346	348
348	347	349
349	348	350
350	349	351
351	350	352
352	351	353
353	352	354
354	353	355
355	354	356
356	355	357
357	356	358
358	357	359
359	358	360
360	359	361
361	360	362
362	361	363
363	362	364
364	363	365
365	364	366
366	365	367
367	366	368
368	367	369
369	368	370
370	369	371
371	370	372
372	371	373
373	372	374
374	373	375
375	374	376
376	375	377
377	376	378
378	377	379
379	378	380
380	379	381
381	380	382
382	381	383
383	382	384
384	383	385
385	384	386
386	385	387
387	386	388
388	387	389
389	388	390
390	389	391
391	390	392
392	391	393
393	392	394
394	393	395
395	394	396
396	395	397
397	396	398
398	397	399
399	398	400
400	399	401
401	400	402
402	401	403
403	402	404
404	403	405
405	404	406
406	405	407
407	406	408
408	407	409
409	408	410
410	409	411
411	410	412
412	411	413
413	412	414
414	413	415
415	414	416
416	415	417
417	416	418
418	417	419
419	418	420
420	419	421
421	420	422
422	421	423
423	422	424
424	423	425
425	424	426
426	425	427
427	426	428
428	427	429
429	428	430
430	429	431
431	430	432
432	431	433
433	432	434
434	433	435
435	434	436
436	435	437
437	436	438
438	437	439
439	438	440
440	439	441
441	440	442
442	441	443
443	442	444
444	443	445
445	444	446
446	445	447
447	446	448
448	447	449
449	448	450
450	449	451
451	450	4

Separate-Chaining Symbol Table

</> SeparateChainingHashST.java

```
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

public class SeparateChainingHashST<Key, Value> implements BasicST<Key, Value> {
    private LinearSearchST<Key, Value>[] st;
    private int m;
    private int n;

    public SeparateChainingHashST() {
        this(4);
    }

    public SeparateChainingHashST(int m) {
        this.m = m;
        st = (LinearSearchST<Key, Value>[]) new LinearSearchST[m];
        for (int i = 0; i < m; i++) {
            st[i] = new LinearSearchST<Key, Value>();
        }
    }

    public boolean isEmpty() {
        return size() == 0;
    }

    public int size() {
        return n;
    }

    public void put(Key key, Value value) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        if (value == null) {
```

Separate-Chaining Symbol Table

</> SeparateChainingHashST.java

```
        throw new IllegalArgumentException("value is null");
    }
    if (n >= 10 * m) {
        resize(2 * m);
    }
    int i = hash(key);
    if (!st[i].contains(key)) {
        n++;
    }
    st[i].put(key, value);
}

public Value get(Key key) {
    if (key == null) {
        throw new IllegalArgumentException("key is null");
    }
    int i = hash(key);
    return st[i].get(key);
}

public boolean contains(Key key) {
    if (key == null) {
        throw new IllegalArgumentException("key is null");
    }
    return get(key) != null;
}

public void delete(Key key) {
    if (key == null) {
        throw new IllegalArgumentException("key is null");
    }
    int i = hash(key);
    if (st[i].contains(key)) {
        n--;
    }
}
```


Separate-Chaining Symbol Table

</> SeparateChainingHashST.java

```
        st[i].delete(key);
        if (m > 4 && n <= 2 * m) {
            resize(m / 2);
        }
    }

    public Iterable<Key> keys() {
        LinkedList<Key> queue = new LinkedList<Key>();
        for (LinearSearchST<Key, Value> chain : st) {
            for (Key key : chain.keys()) {
                queue.enqueue(key);
            }
        }
        return queue;
    }

    private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % m;
    }

    private void resize(int chains) {
        SeparateChainingHashST<Key, Value> temp = new SeparateChainingHashST<Key, Value>(chains);
        for (LinearSearchST<Key, Value> chain : st) {
            for (Key key : chain.keys()) {
                temp.put(key, chain.get(key));
            }
        }
        this.m = temp.m;
        this.n = temp.n;
        this.st = temp.st;
    }

    public static void main(String[] args) {
        SeparateChainingHashST<String, Integer> st = new SeparateChainingHashST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
```

Separate-Chaining Symbol Table

</> SeparateChainingHashST.java

```
        String key = StdIn.readString();
        st.put(key, i);
    }
    for (String s : st.keys()) {
        StdOut.println(s + " " + st.get(s));
    }
}
}
```