Lecture 19

Hash Tables

CS61B, Spring 2024 @ UC Berkeley

Slides Credit: Josh Hug

Motivation, Set Implementations

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- BobaCounterSet
- DynamicArrayOfListsSet
- lowercase strings
- Integer Overflow
- Hash Codes

Hash Tables in Java

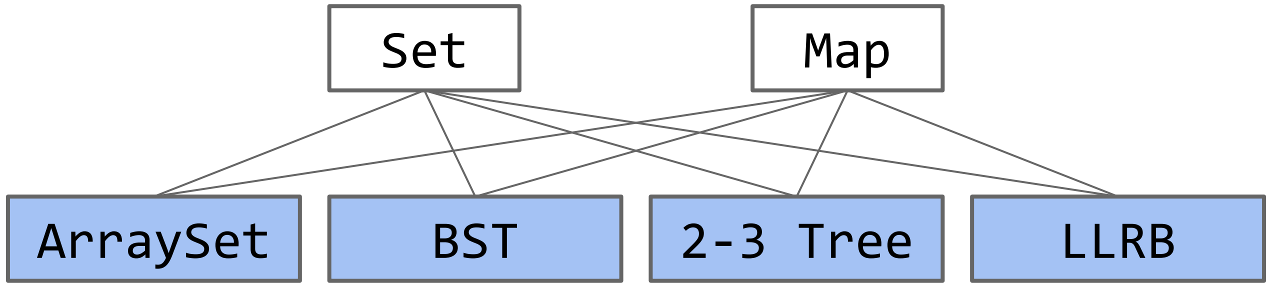
Hash Table Performance and Summary

Creating a Good Hash Code (extra)

Linear Probing (extra)

Sets

We've now seen several implementations of the Set (or Map) ADT.



Worst case runtimes

	contains(x)	add(x)	Notes
ArraySet	Θ(N)	Θ(N)	
BST	Θ(N)	Θ(N)	Random trees are Θ(log N).
2-3 Tree	Θ(log N)	Θ(log N)	Beautiful idea. Very hard to implement.
LLRB	Θ(log N)	Θ(log N)	Maintains bijection with 2-3 tree. Hard to implement.

Limits of Search Tree Based Sets

Our search tree based sets require items to be comparable.

- Need to be able to ask “is X < Y?” Not true of all types (ex. How do you compare 苹 and 橙?).
- Could we somehow avoid the need for objects to be comparable?

Our search tree sets have excellent performance, but could maybe be better?

- Θ(log N) is amazing. 1 billion items is still only height ~30.
- Could we somehow do better than Θ(log N)?

Today we’ll see the answer to both of the questions above is yes.

WriteItOnTheWallSet

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- **WriteItOnTheWallSet**
- BobaCounterSet
- DynamicArrayOfListsSet
- lowercase strings
- Integer Overflow
- Hash Codes

Hash Tables in Java

Hash Table Performance and Summary

Creating a Good Hash Code (extra)

Linear Probing (extra)

Data Structures Reflect Real Life

Data Structures tend to be analogous to real-life things, so it's often useful to try playing the role of a data structure as a human (to get ideas on how they work)

Let's think about a simplified Set of Integers, which requires these two operations:

- Add: Adds a new item to the Set
    - Assumption for now: We never try to add something already in the Set
        - Can make this assumption because we can call Contains before adding.
- Contains: Checks if a given number is in the set.

Our goal is to make these operations as fast as possible.

WriteItOnTheWallSet

Let's introduce a human implementation of Set: WriteItOnTheWall Set

We will have a wall, and a pencil.

- Add: Write the number at a random place on the wall
  - If the wall is full, get a bigger wall (we saw from ArrayList that this can be done in constant time amortized, so we can ignore this safely)
- Contains: Look for our number on the wall. If we find it, return true. Otherwise return false

Strongly analogous to an "ArraySet"

Two questions:

- Is it fast to add?
- Is it fast to contains?

Is it fast to add: Adding "5" to a wall of 10 numbers

212				131
759		281		
			670	
953	984			104
		958		526

Is it fast to add: Adding "5" to a wall of 10 numbers

212				131
759		281		
			670	
953	984	5		104
		958		526

Is it fast to add: Adding "5" to a wall of 100 numbers

281	953	104	958	212	131	984	670	759	526
161	560	815	289	462	591	828	981	603	922
175	455	286	605	543	375	669	970	651	65
995	13	916	616	721	913	872	881	22	830
584	137	228	86	861	109	821	253	305	530
317	340	494	719	737	677	786	672	216	702
35	770	480	557	74	52	632	765	753	73
194	77	226	764	173	979	454	106	967	551
556	644	739	547	973	796	525	573	920	28
290	708	298	56	288	891	867	579	417	

Is it fast to add: Adding "5" to a wall of 100 numbers

281	953	104	958	212	131	984	670	759	526
161	560	815	289	462	591	828	981	603	922
175	455	286	605	543	375	669	970	651	65
995	13	916	616	721	913	872	881	22	830
584	137	228	86	861	109	821	253	305	530
317	340	494	719	737	677	786	672	216	702
35	770	480	557	74	52	632	765	753	73
194	77	226	764	173	979	454	106	967	551
556	644	739	547	973	796	525	573	920	28
290	708	298	56	288	891	867	579	417	5

Is it fast to contains: Search for "439" on a wall of 10 numbers

212				131
759		281		
			670	
953	984			104
		958		526

Is it fast to contains: Search for "605" on a wall of 100 numbers

281	953	104	958	212	131	984	670	759	526
161	560	815	289	462	591	828	981	603	922
175	455	286	605	543	375	669	970	651	65
995	13	916	616	721	913	872	881	22	830
584	137	228	86	861	109	821	253	305	530
317	340	494	719	737	677	786	672	216	702
35	770	480	557	74	52	632	765	753	73
194	77	226	764	173	979	454	106	967	551
556	644	739	547	973	796	525	573	920	28
290	708	298	56	288	891	867	579	417	5

Is it fast to contains: Search for "605" on a wall of 100 numbers

281	953	104	958	212	131	984	670	759	526
161	560	815	289	462	591	828	981	603	922
175	455	286	**605**	543	375	669	970	651	65
995	13	916	616	721	913	872	881	22	830
584	137	228	86	861	109	821	253	305	530
317	340	494	719	737	677	786	672	216	702
35	770	480	557	74	52	632	765	753	73
194	77	226	764	173	979	454	106	967	551
556	644	739	547	973	796	525	573	920	28
290	708	298	56	288	891	867	579	417	5

Is it fast to contains: Search for "白" and "黒" on a wall of 1000 Kanji

須献資竹奈正賓方富秩苗移辞斤灯情腹縄典械迄淵執伐速寛藻具彰嗣謙役傾嘘於柵企娯李燃複主賂院雅設祥
憾貌溝庁傲霊号酔旬停冖剤寡墳称遺原貧⺍頻甲逮幕徒⺹私素片署弧落兵坪感所農腐論眼式齢孝角略宛郎森
好価認宅壁洲拐志反渋睡菓紋針縮銅蜂捨廾探確拷率央干堂窃林鋭⺌翔愚賀革月懸挑哀凵拙寝蒲以窟拒数骸
頓障豕惨除体帽屏唄且狙伺膨製査用挨石釈雖投預爽繰品暑圏眺滴轄庭怠筒絞幌完築年憎莫高核煙机漬胎般
擬岬販華斉遂晩社乃鈍鹿征傷縫苑沈獲重髄酌負党陸誠伊棟晶沼脱炎惚佐弊漠妖宿貿帥柴賦審飠緊散旺塊餅
刺左後騰戯攵抗胸吊週症吟丸縁紳拳昇閲境披甫妨余錬謡厚井二痛渡匂白入盲淑廊攻萌蘇検使平鍛徹薪怒現
笛腕賜康許舎押手矢霧悟連荘乏拠浩瓶遮員撈朱零濁⻍成君囚鮮侮書柳貸鑑火恋悔朔樹偵求服廷謝尹泰坊贈
去貴次救量剰慨逸制作哲駆裾血悠曹罒展対無迷福遊殊語俳端云外詩逆士忍避拭放得礁脈彑斜性飛婿底牲油
職汽鬼弘隻導局惰将束粘毎妥観神妬加慮腺訟諸残痢必熊侯相根壱訴胡郷慢都系第向勧光著秀喩挿筆擦益肥
籍履循勝溶析就巨渚蜜依塔咲溺席含拝慶案返揺礎紙望囗赴雑畏惧盛徐協旗爪競澄倣宜心泥止誓嚇育羅冫鼻
七吐皇到穴庄頼堀滅豆篤裏河棺甘紀省虜醸労恒把浜侍途距足迅辣呪憂亭免控還暁子恵日矛描疒互右排駒盗
程玄拘畿虐咽人稼軽満営応閑聴道枢文混拾由国柔箇鉱隆別丁襲邦紫灬枠肺器勉乳菊税口整繕状呂玉恭液我
塩窯仁磯脅復嘱侵悲袋親捉計杯孟磨辛挙裸胃能刷柏他誘驚那粉灰雨巣央表泡沿姻腫扉藩妻掛褐可栽藍湯叶
危息殺与演弦掃橋夏当景怪合覇癖裂釜頃⻂也射紡懇通彦疫崇彡層集賠換適吹忙仙漂読栓薬抜隈癒同花臨否
線寄常紅蒼出箸町凝苛編恨公裁瑠欲羊刂派奪報瘍廴画或時穫色伯効討併改腸木麓秘喚宝視汚死顧漫髪保威
議喪戸一然引幸摩監斐単雇羨冶汎哉脊間股評訓縦祖領待星倫腰宙柱東登弟英貝矯暇痩章医前況達治其削蚊
卓陽司洋鯉凸膚句奔頂元衰殻津解建乞幣男鬱戻琴宀下毛巾忄斥斎装謂附竜兼創熟翁添繁発憧岡露殳凹説歓
嫁存眠迎玩県克管悼幽艇漱決違触衆為厳理璃稽請列膝航倒条校帆屯俵棄冥亦刑組村蘭鼓門脂卵刈昆奥敷父
吉丑欄塞禍低傘僅要這精飼較慌洪翌多輩追婦持鐘経漆衣難型敬母鋼渇肖猛峡俺振茨遥夕冗舛糧叔州屋覚隔
誕抵犠穀隶弁損係活臓娘綱姉抑区利叱陰部叩傍亜伎両版韋夢衡基美曽民焦滑阜肪蔑記安概付湾円艦舌訂務
容卜瞭契注亻芽懲優操賞唐野暫飲筈銘静酵牧扱雀調流飢冒覧予尻馴喝弄粧雷質踊便嫡慈悩褒冠棒殿着笠剛

Is it fast to contains: Search for "白" and "黒" on a wall of 1000 Kanji

須献資竹奈正賓方富秩苗移辞斤灯情腹縄典械迄淵執伐速寛藻具彰嗣謙役傾嘘於柵企娯李燃複主賂院雅設祥
憾貌溝庁傲霊号酔旬停冖剤寡墳称遺原貧癶頻甲逮幕徒耂私素片署弧落兵坪感所農腐論眼式齢孝角略宛郎森
好価認宅壁洲拐志反渋睡菓紋針縮銅蜂捨廾探確拷率央干堂窃林鋭屮翔愚賀革月懸挑哀凵拙寝蒲以窟拒数骸
頓障豕惨除体帽屏唄且狙伺膨製査用挨石釈雌投預爽繰品暑圏眺滴轄庭怠筒絞幌完築年憎莫高核煙机漬胎般
擬岬販華斉遂晩社乃鈍鹿征傷縫苑沈獲重髄酌負党陸誠伊棟晶沼脱炎惚佐弊漠妖宿貿帥柴賦審飠緊散旺塊餅
刺左後騰戯攵抗胸吊週症吟丸縁紳拳昇閲境披甫妨余錬謡厚井二痛渡匂 白入盲淑廊攻萌蘇検使平鍛徹薪怒現
笛腕賜康許舎押手矢霧悟連荘乏拠浩瓶遮員拶朱零濁辶成君囚鮮侮書柳貸鑑火恋悔朔樹偵求服廷謝尹泰坊贈
去貴次救量剰慨逸制作哲駆裾血悠曹罒展対無迷福遊殊語俳端云外詩逆士忍避拭放得礁脈彑斜性飛婿底牲油
職汽鬼弘隻導局惰将束粘毎妥観神妬加慮腺訟諸残痢必熊侯相根壱訴胡郷慢都系第向勧光著秀喩挿筆擦益肥
籍履循勝溶析就巨渚蜜依塔咲溺席含拝慶案返揺礎紙望囗赴雑畏惧盛徐協旗爪競澄倣宜心泥止誓嚇育羅冫鼻
七吐皇到穴庄頼堀滅豆篤裏河棺甘紀省虜醸労恒把浜侍途距足迅辣呪憂亭免控還暁子恵日矛描疒互右排駒盗
程玄拘畿虐咽人稼軽満営応閑聴道枢文混拾由国柔箇鉱隆別丁襲邦紫灬枠肺器勉乳菊税口整繕状呂玉恭液我
塩窯仁磯脅復嘱侵悲袋親捉計杯孟磨辛挙裸胃能刷柏他誘驚那粉灰雨巣央表泡沿姻腫扉藩妻掛褐可栽藍湯叶
危息殺与演弦掃橋夏当景怪合覇癖裂釜頃礻也射紡懇通彦疫崇彡層集賠換適吹忙仙漂読栓薬抜隈癒同花臨否
線寄常紅蒼出箸町凝苛編恨公裁瑠欲羊刂派奪報瘍廴画或時穫色伯効討併改腸木麓秘喚宝視汚死顧漫髪保威
議喪戸一然引幸摩監斐単雇羨冶汎哉脊間股評訓縦祖領待星倫腰宙柱東登弟英貝矯暇痩章医前況達治其削蚊
卓陽司洋鯉凸膚句奔頂元衰殻津解建乞幣男鬱戻琴宀下毛巾忄斥斎装謂附竜兼創熟翁添繁発憧岡露殳凹説歓
嫁存眠迎玩県克管悼幽艇漱決違触衆為厳理璃稽請列膝航倒条校帆屯俵棄冥亦刑組村蘭鼓門脂卵刈昆奥敷父
吉丑欄塞禍低傘僅要這精飼較慌洪翌多輩追婦持鐘経漆衣難型敬母鋼渇肖猛峡俺振茨遥夕冗舛糧叔州屋覚隔
誕抵犠穀隷弁損係活臓娘綱姉抑区利叱陰部叩傍亜伎両版韋夢衡基美曽民焦滑阜肪蔑記安概付湾円艦舌訂務
容卜瞭契注亻芽懲優操賞唐野暫飲筈銘静酵牧扱雀調流飢冒覧予尻馴喝弄粧雷質踊便嫡慈悩褒冠棒殿着笠剛

WriteItOnTheWallSet

WriteItOnTheWallSet had fast adds (Θ(1)), but slow contains (Θ(N), where N is the number of elements)

- Good thing: It didn't matter what type the items were; it worked as well with ints as it did with Kanji

How can we make this faster?

- Sort the data?
  - Makes contains faster, but add slower (since we need to rewrite everything)
  - Doesn't work on Kanji
  - Optimizations to this leads to the TreeSet approach discussed earlier and log(N) runtime
- Categorize the data?
  - How do we do that? Get boba.

BobaCounterSet

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- **BobaCounterSet**
- DynamicArrayOfListsSet
- lowercase strings
- Integer Overflow
- Hash Codes

Hash Tables in Java

Hash Table Performance and Summary

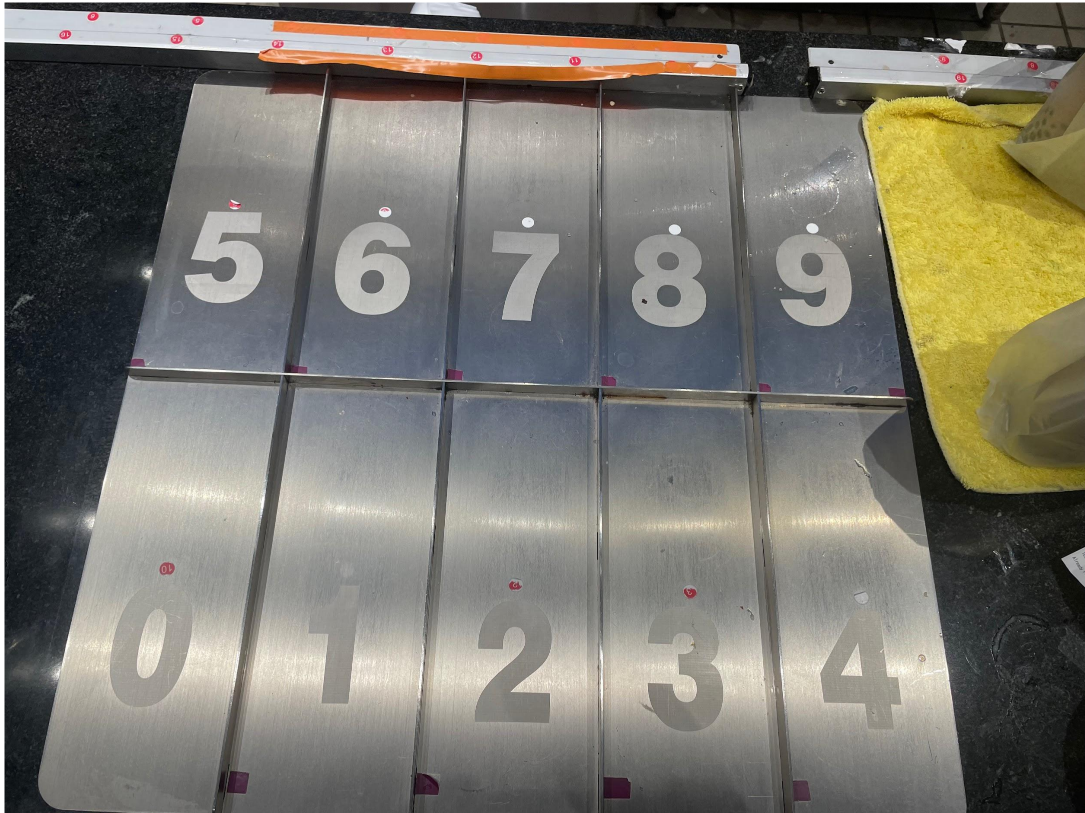
Creating a Good Hash Code (extra)

Linear Probing (extra)

When stuck on a hard problem, get boba

TPT has this interesting device to help organize their boba

- When you order boba, you get an order number
- Once the boba is made, your boba gets placed on the counter for pickup
- Often 5-10 boba on the counter at a time, so to avoid customers searching for their boba, so they place the boba in the slot corresponding to the last digit of your order number
- Fast to add, fast to find



BobaCounterSet

Let's try to formalize this with a BobaCounter Set

We will have a wall split into 10 "bins", and a pencil.

- Add: Write the number to the place on the wall **corresponding to the last digit of the number**
  - Ex. "193" goes in the 3 segment, "100" goes in the 0 segment
  - If the wall is full, get a bigger wall (we saw from ArrayList that this can be done in constant time amortized, so we can ignore this safely)
- Contains: Look for our number on the wall **in the bin corresponding to our last digit**. If we find it, return true. Otherwise return false

Two questions:

- Is it fast to add?
- Is it fast to contains?

Is it fast to add: Adding "5" to a wall of 100 numbers

560	920	161	131	212	922	603	73	194	644
670	830	281	651	22	672	953	753	104	984
530	770	591	981	462	702	13	253	764	494
970	290	721	891	52	632	173	973	584	74
480	340	551	821	872		573	913	454	
		861	881			543			
765	375	216	556	737	677	288	828	109	669
35	305	796	786	137	77	708	298	719	979
65	995	226	86	557	967	228	28	739	579
815	175	526	616	417	867	958		289	759
605	455	106	286	317	547				
525		916	56						

Is it fast to add: Adding "5" to a wall of 100 numbers

560	920	161	131	212	922	603	73	194	644
670	830	281	651	22	672	953	753	104	984
530	770	591	981	462	702	13	253	764	494
970	290	721	891	52	632	173	973	584	74
480	340	551	821	872		573	913	454	
		861	881			543			
765	375	216	556	737	677	288	828	109	669
35	305	796	786	137	77	708	298	719	979
65	995	226	86	557	967	228	28	739	579
815	175	526	616	417	867	958		289	759
605	455	106	286	317	547				
525	5	916	56						

Is it fast to contains: Search for "605" on a wall of 100 numbers

560	920	161	131	212	922	603	73	194	644
670	830	281	651	22	672	953	753	104	984
530	770	591	981	462	702	13	253	764	494
970	290	721	891	52	632	173	973	584	74
480	340	551	821	872		573	913	454	
		861	881			543			
765	375	216	556	737	677	288	828	109	669
35	305	796	786	137	77	708	298	719	979
65	995	226	86	557	967	228	28	739	579
815	175	526	616	417	867	958		289	759
605	455	106	286	317	547				
525	5	916	56						

Is it fast to contains: Search for "605" on a wall of 100 numbers

560	920	161	131	212	922	603	73	194	644
670	830	281	651	22	672	953	753	104	984
530	770	591	981	462	702	13	253	764	494
970	290	721	891	52	632	173	973	584	74
480	340	551	821	872		573	913	454	
		861	881			543			
765	375	216	556	737	677	288	828	109	669
35	305	796	786	137	77	708	298	719	979
65	995	226	86	557	967	228	28	739	579
815	175	526	616	417	867	958		289	759
605	455	106	286	317	547				
525	5	916	56						

BobaCounterSet

BobaCounterSet still has equally fast adds, and contains is now only as slow as the wall segment with the most elements

- If the numbers are random, runtime is reduced by a factor of 10

Any problems with this approach?

- Since we split the wall into 10 bins, we have more wasted space. How to minimize that?
- What do we do when the number of elements gets so large that even one bin has 1000 items?
- What happens if the numbers aren't random (e.g. most numbers end in a 0)?
- What if we want to deal with things that aren't numbers, like Kanji?

DynamicArrayOf ListsSet

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- BobaCounterSet
- **DynamicArrayOfListsSet**
- lowercase strings
- Integer Overflow
- Hash Codes

Hash Tables in Java

Hash Table Performance and Summary

Creating a Good Hash Code (extra)

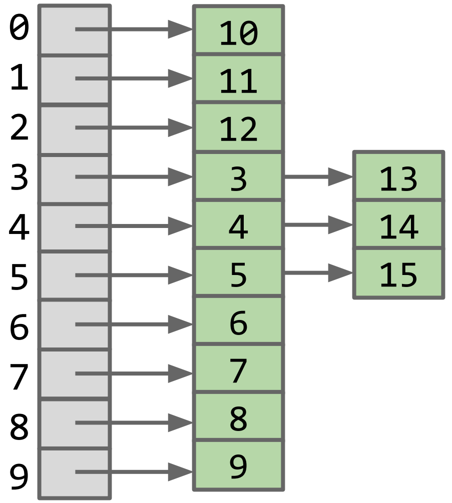
Linear Probing (extra)

How to minimize wasted space?

Instead of assigning the same amount of wall space per section, dynamically increase the size of each section as items get added there

Easiest solution here is to use Linked Lists

- Other solutions exist, but we'll focus on the Linked List approach for this class
- We still use one unit of wasted space per empty section (e.g. if 0 had no elements)
    - But overall, this uses less memory than before



How to handle large numbers of items?

Let N = number of items in all bins, M = number of bins

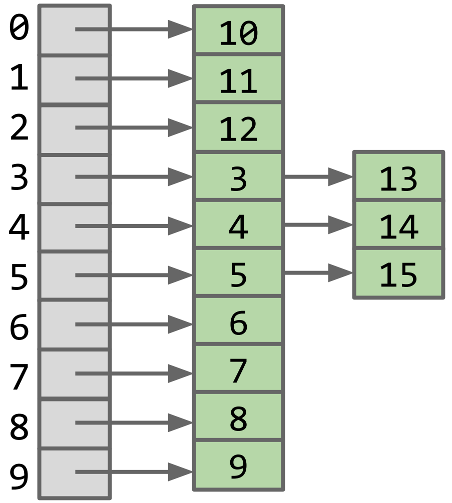
If we assume that values are evenly distributed, each bin has about N/M items

So contains runs in Θ(N/M) time.

If M is constant, that reduces to Θ(N).

- Solution: Have M grow with N so that each bucket has on average a constant number of elements.
- Needs a way to categorize numbers into M groups for arbitrary M: "last digit" only works with M=10.

Is there a common mathematical function that generalizes "last digit"?



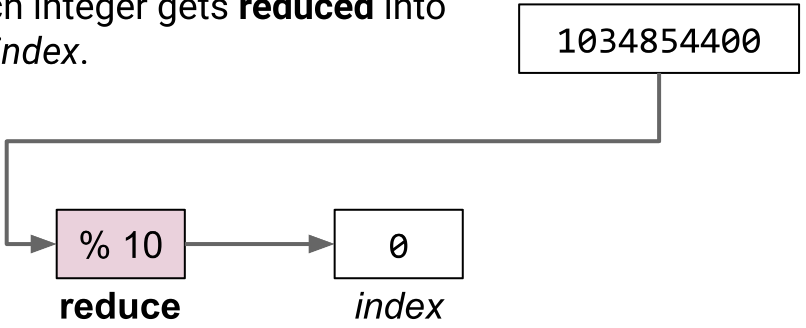
Reduction Functions

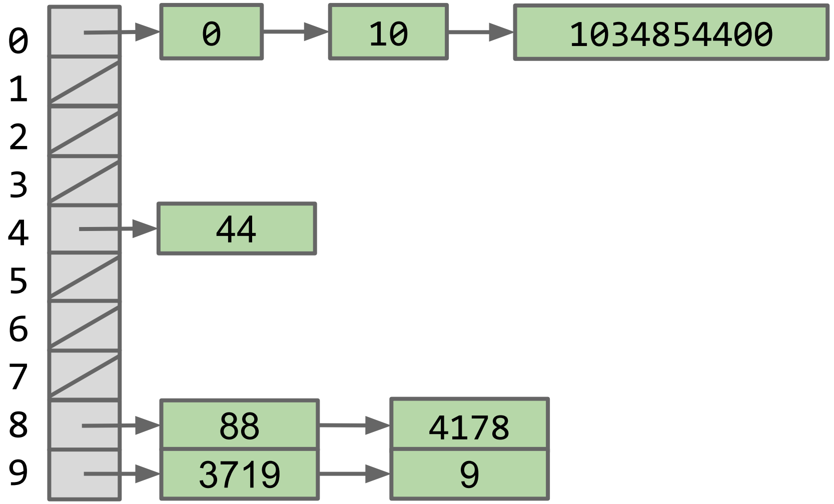
Easiest solution is the "modulus" operator.

- Can apply to any value of M
- Evenly distributes randomly-generated numbers
- Relatively prime moduli are statistically independent
- Multiplying M by an integer splits each bin into smaller bins independently

Other reductions are possible, e.g. number of digits. But modulus is the most natural and best reduction function.

Each integer gets **reduced** into an *index*.





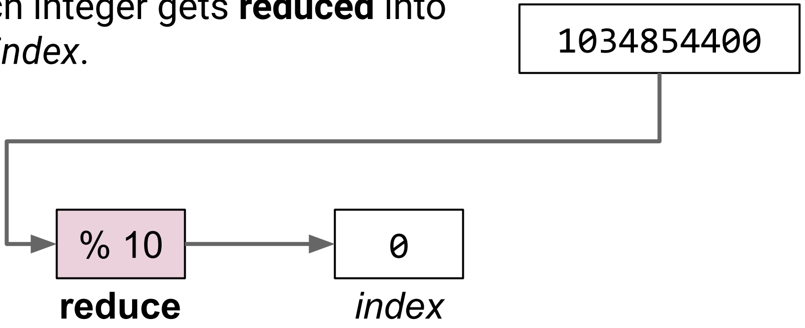
Increasing M

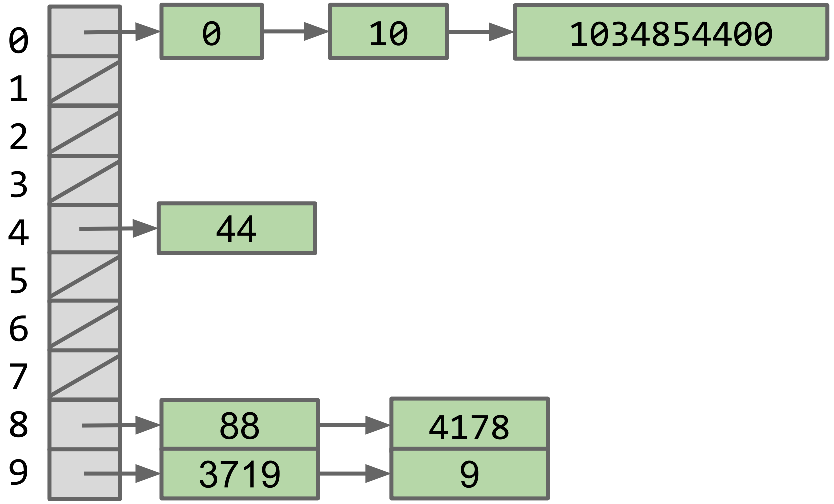
To keep constant time contains, we need N/M to stay less than some constant k.

Two approaches:

- Increase M when the largest bin exceeds k.
  - Generally leads to a lot of empty bins, so not used.
- Increase M when the average bin size exceeds k.

Each integer gets **reduced** into an *index*.

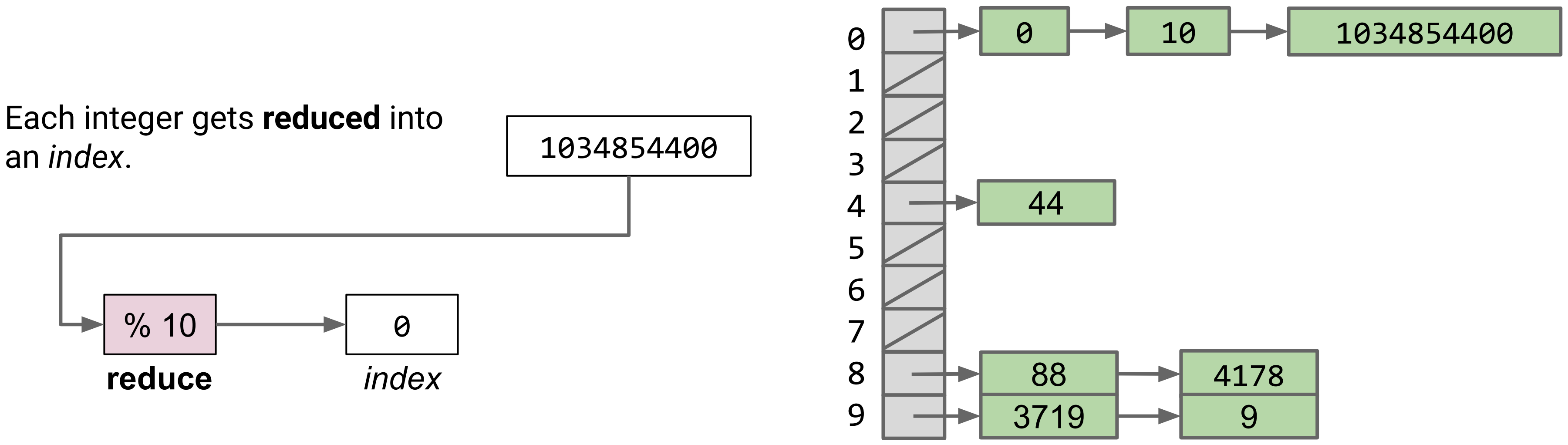




Increasing M

How much to increase M?

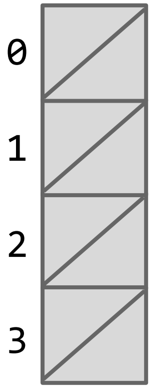
- When we increase M, we'll have to reassign every number to a new box, which will take Θ(N) time during that add operation
- Our goal is to have Θ(1) amortized runtime, and we've seen from ArrayLists that we can get that as long as we do Θ(N) steps rarely enough
- Therefore, M should double every time we resize



Hash Table Resizing Example

Suppose we set a rule that when N/M is ≥ 1.5, we double M.

N = 0 M = 4 N / M = 0

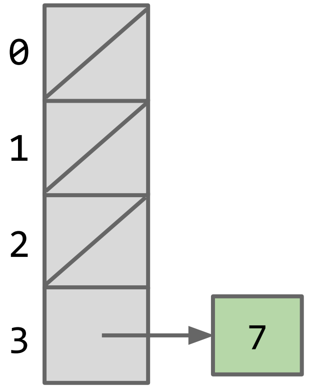


Hash Table Resizing Example

Suppose we set a rule that when N/M is ≥ 1.5, we double M.

- add(7)

N = 1 M = 4 N / M = 0.25

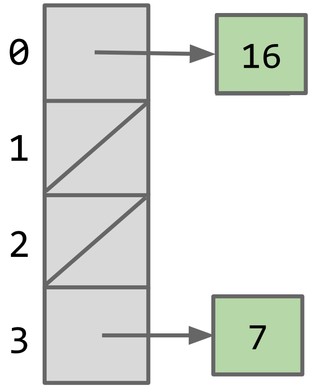


Hash Table Resizing Example

Suppose we set a rule that when N/M is ≥ 1.5, we double M.

- add(7), add(16)

N = 2 M = 4 N / M = 0.5

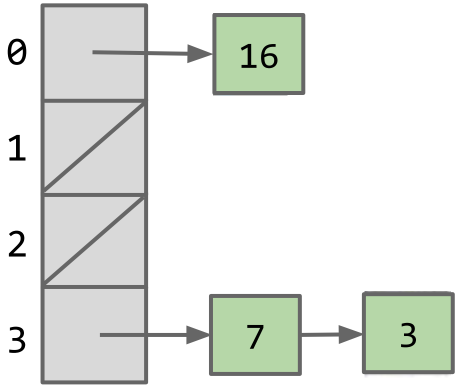


Hash Table Resizing Example

Suppose we set a rule that when N/M is ≥ 1.5, we double M.

- add(7), add(16), add(3)

N = 3   M = 4   N / M = 0.75

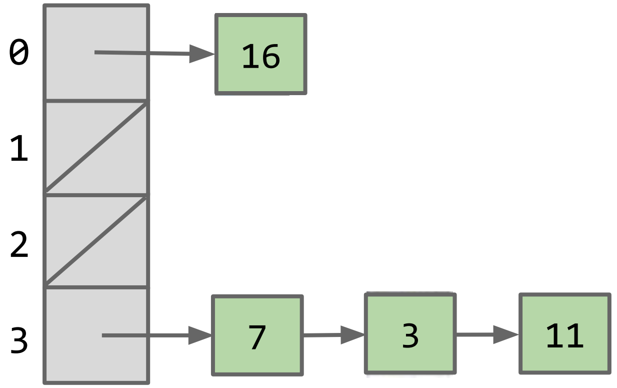


Hash Table Resizing Example

Suppose we set a rule that when N/M is ≥ 1.5, we double M.

- add(7), add(16), add(3), add(11)

N = 4 M = 4 N / M = 1

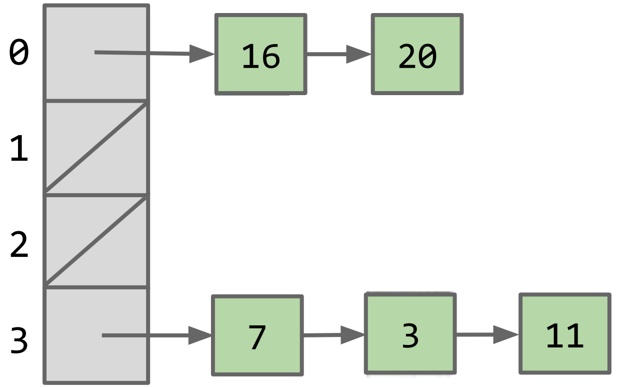


Hash Table Resizing Example

Suppose we set a rule that when N/M is ≥ 1.5, we double M.

- add(7), add(16), add(3), add(11), add(20)

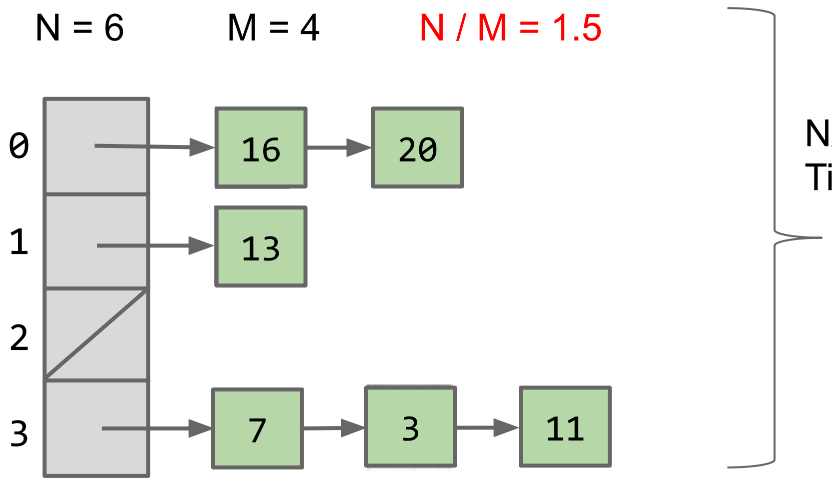
N = 5 M = 4 N / M = 1.25



Hash Table Resizing Example

Suppose we set a rule that when N/M is ≥ 1.5, we double M.

- add(7), add(16), add(3), add(11), add(20), add(13). Resize triggered.

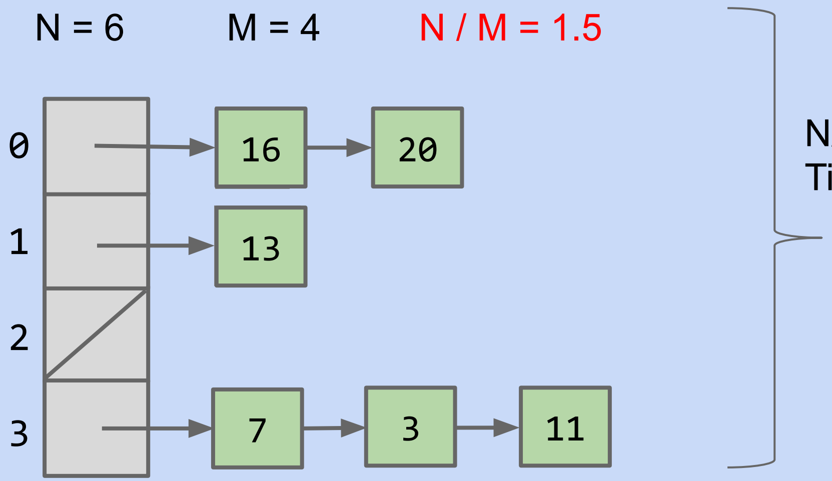


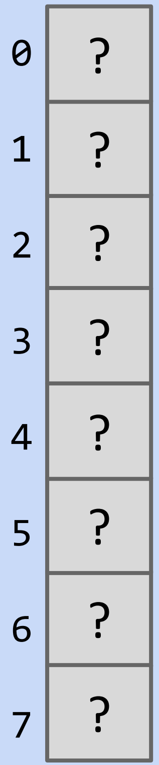
N/M is too large.
Time to double!

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

- Draw the results after doubling M.

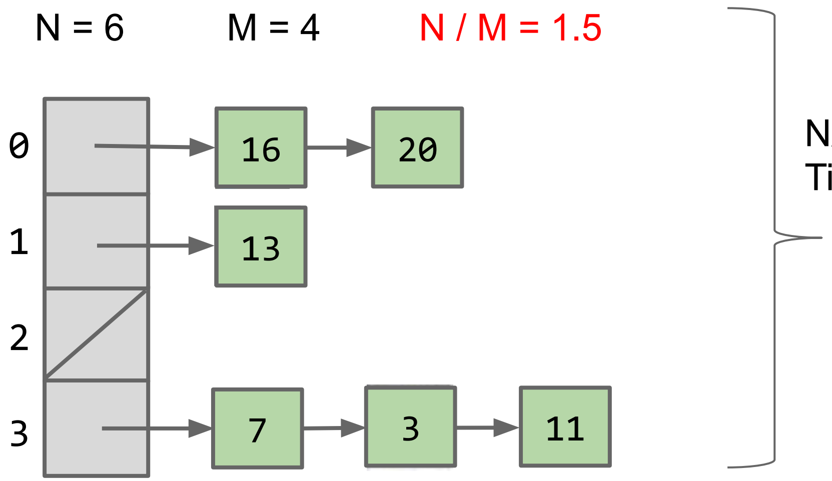




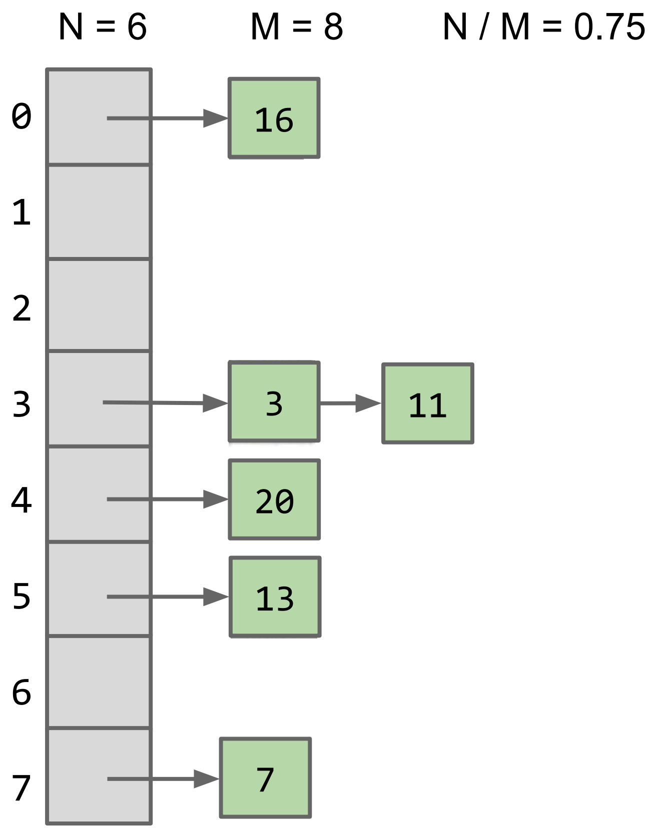
Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

- Draw the results after doubling M.



N/M is too large.
Time to double!



DynamicArrayOfListsSet

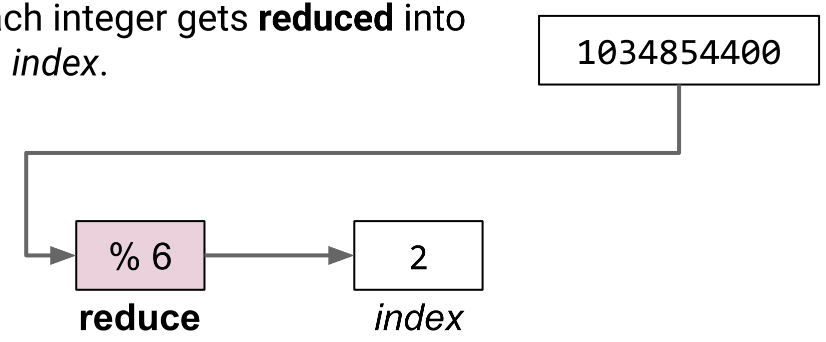
The data structure we just built might be called a DynamicArrayOfListsSet.

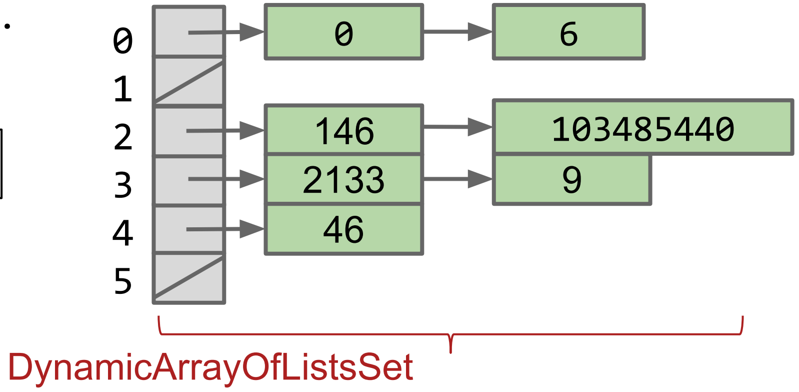
- Not as intuitive as our original idea, but the core idea stayed the same

If we have N items that are evenly distributed, length of each list is ~N/M.

- N/M is **constant** asymptotically.
- So operations are constant on average.

Each integer gets **reduced** into an *index*.





We'll think more carefully about runtime later.

lowercase strings

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- BobaCounterSet
- DynamicArrayOfListsSet
- **lowercase strings**
- Integer Overflow
- Hash Codes

Hash Tables in Java

Hash Table Performance and Summary

Creating a Good Hash Code (extra)

Linear Probing (extra)

Goal: Storing Strings

The data structure we have so far is great for storing integers.

- Let’s try to figure out how to store Strings of lowercase characters.

Storing the Word cat

Suppose we want to add("cat")

The key question:

- Which bucket do we put "cat" in?
- One idea: Use the order in the alphabet of the first letter as the list number.
    - a = 0, b = 1, c = 2, ..., z = 25
    - So cat would go in bucket 2.
    - Forces us to start with 26 buckets

What about after resize?

- After the first resize, look at the first two letters
    - aa = 0, ab = 1, ac = 2, ..., zz = 675
    - cat would go in bucket 52.

What are some issues with this approach?

Storing the Word cat (your answer)

Suppose we want to add("cat")

The key question:

- Which bucket do we put "cat" in?
- One idea: Use the order in the alphabet of the first letter as the list number.
  - a = 0, b = 1, c = 2, ..., z = 25
  - So cat would go in bucket 2.
- After the first resize, look at the first two letters, then first three, and so on

What are some issues with this approach?

- Not a random distribution of letters
- Single-letter "a" can't be placed after resize -> extends as resizes grow
-

Storing the Word cat (my answer)

Suppose we want to add("cat")

The key question:

- Which bucket do we put "cat" in?
- One idea: Use the order in the alphabet of the first letter as the list number.
    - a = 0, b = 1, c = 2, ..., z = 25
    - So cat would go in bucket 2.
- After the first resize, look at the first two letters, then first three, and so on

What are some issues with this approach?

- Where to put short strings (e.g. "a") after resize? (can probably fix)
- It feels wrong for Strings to force our Set to resize to 26, 676, etc. buckets, when ints allowed for any number of buckets. (CRITICAL)
    - Are we going to have to define a new resize for every type of object???

Design Philosophy: Stringy stuff should be done in the String class

The big problem with the previous approach was that Set was responsible for figuring out how to categorize Strings

- That shouldn't be the Set's job, since otherwise Set would have to know about every single Object in existence (including ones that aren't built yet), and how to categorize them

At the same time, String shouldn't be able to dictate when Set decides to resize

- With ints, Set could decide the M/N threshold and bin multiplier, so Set could decide which values made the most sense (given memory/time constraints).

Solution: Set was most flexible when working on ints, so make it so that Set only works on ints.

Define a method f (in String) to convert Strings into an int, and store String s in the bin corresponding to f(s).

- String gets to decide how it wants to be categorized, Set gets to decide when it wants to resize.

Finding a Way to Store the Word cat

Suppose we want to add(“cat”)

The key question:

- **How do I convert “cat” into a number?**

What is another idea? Assume for now we’re dealing with only lower case letters in English.

Finding a Way to Store the Word cat (Your Answer)

Suppose we want to add("cat")

The key question:

- How do I convert "cat" into a number?

What is another idea? Assume for now we're dealing with only lower case letters in English.

- Sum up each letter
- ASCII/Unicode
- Base 26
- Number of characters

Finding a Way to Store the Word cat (My Answer)

Suppose we want to add(“cat”)

The key question:

- How do I convert “cat” into a number?

What is another idea? Assume for now we’re dealing with only lower case letters in English.

- Ideally we should evenly distribute Strings; we don't want any int to have significantly more associated strings than average.
- Treat cat as a base 26 number.

Treating cat as a Base 26 Number

Use all digits by multiplying each by a power of 26.

- a = 1, b = 2, c = 3, …, z = 26
- Thus the index of “cat” is $\underbrace{(3 \text{ x } 26^{2}) + (1 \text{ x } 26^{1}) + (20 \text{ x } 26^{0})}_{}$ = 2074.

Why this specific pattern?

- Let’s review how numbers are represented in decimal.

The Decimal Number System vs. My System for Strings

In the decimal number system, we have 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Want numbers larger than 9? Use a sequence of digits.

Example: 7091 in base 10

- $7091_{10} = (7 \times 10^{3}) + (0 \times 10^{2}) + (9 \times 10^{1}) + (1 \times 10^{0})$

Our system for strings is almost the same, but with letters.

- One difference: In decimal numbers, 000 is the same as 0, but with strings aaa is different from a.
- To deal with this, we just don't have a 0 in our system, i.e. a is 1, not 0.

Test Your Understanding

Convert the word “bee” into a number by using our “powers of 26” strategy.

Reminder: $\text{cat}_{26} = (3 \times 26^{2}) + (1 \times 26^{1}) + (20 \times 26^{0}) = 2074_{10}$

Hint: ‘b’ is letter 2, and ‘e’ is letter 5.

Test Your Understanding

Convert the word “bee” into a number by using our “powers of 26” strategy.

Reminder: $\text{cat}_{26} = (3 \times 26^{2}) + (1 \times 26^{1}) + (20 \times 26^{0}) = 2074_{10}$

Hint: ‘b’ is letter 2, and ‘e’ is letter 5.

- $\text{bee}_{26} = (2 \times 26^{2}) + (5 \times 26^{1}) + (5 \times 26^{0}) = 1487_{10}$

Uniqueness

- $\text{cat}_{26} = (3 \times 26^{2}) + (1 \times 26^{1}) + (20 \times 26^{0}) = 2074_{10}$
- $\text{bee}_{26} = (2 \times 26^{2}) + (5 \times 26^{1}) + (5 \times 26^{0}) = 1487_{10}$

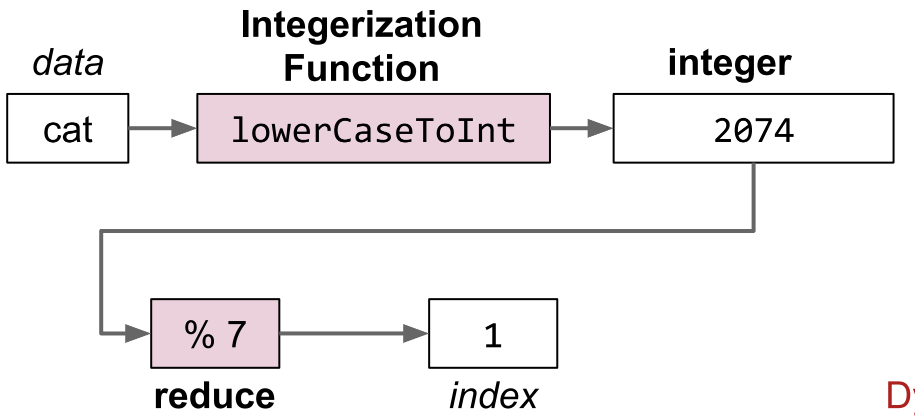
As long as we pick a base ≥ 26, this algorithm is guaranteed to give each lowercase English word a unique number!

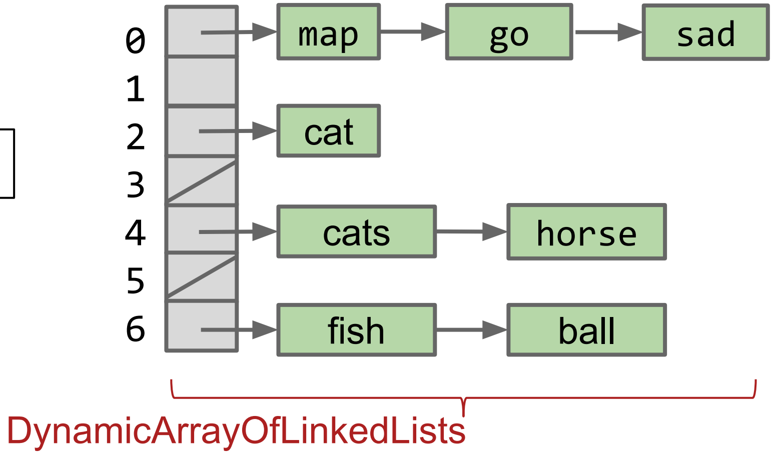
- Using base 26, no other words will get the number 1487.

The Hash Table

We've now extended our DynamicArrayOfLinkedLists to handle strings.

- *Data* is converted by a **integerization function** into an integer representation.
- The **integer** is then **reduced** to a valid *index*, usually using the modulus operator, e.g. 2348762878 % 10 = 8.





Implementing `englishToInt` (optional)

Optional exercise: Try to write a function `englishToInt` that can convert English strings to integers by adding characters scaled by powers of 26.

Examples:

- a: 1
- z: 26
- aa: 27
- bee: 1487
- cat: 2074
- dog: ??
- potato: ??

Implementing `englishToInt` (optional) (solution)

```java
/** Converts ith character of String to a letter number.
    * e.g. 'a' -> 1, 'b' -> 2, 'z' -> 26 */
public static int letterNum(String s, int i) {
    int ithChar = s.charAt(i);
    if ((ithChar < 'a') || (ithChar > 'z'))
        { throw new IllegalArgumentException(); }
    return ithChar - 'a' + 1;
}

public static int englishToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i += 1) {
        intRep = intRep * 26;
        intRep = intRep + letterNum(s, i);
    }
    return intRep;
}
```

Integer Overflow

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- BobaCounterSet
- DynamicArrayOfListsSet
- lowercase strings
- **Integer Overflow**
- Hash Codes

Hash Tables in Java

Hash Table Performance and Summary

Creating a Good Hash Code (extra)

Linear Probing (extra)

DataIndexedStringSet

Using only lowercase English characters is too restrictive.

- What if we want to store strings like “2pac” or “eGg!”?

Suppose we wanted a unique integer for each possible such string.

- Need to assign an integer to all possible characters, e.g. what integer goes with !

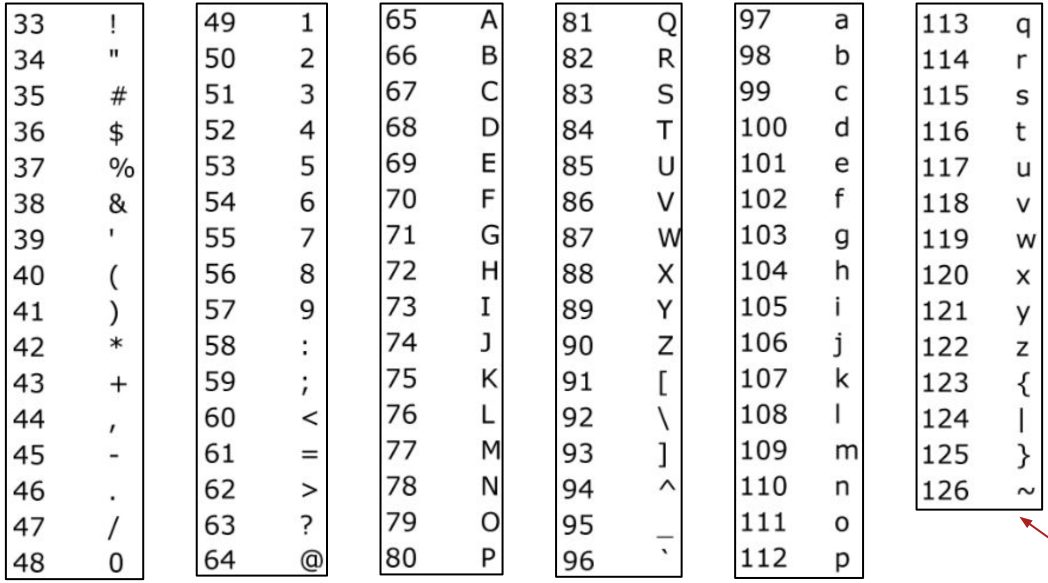
Someone has already done this.

- Let’s first discuss briefly discuss the ASCII standard.

ASCII Characters

The most basic character set used by most computers is ASCII format.

- Each possible character is assigned a value between 0 and 127.
- Characters 33 - 126 are “printable”, and are shown below.
- For example, `char c = ’D’` is equivalent to `char c = 68`.



33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		
48	0	64	@	80	P	96	`	112	p		

biggest value is 126

DataIndexedStringSet

Maximum possible value for english-only text including punctuation is 126, so can use 126 as our base in order to ensure unique values for all possible strings.

Examples:

- $\text{bee}_{126} = (98 \times 126^{2}) + (101 \times 126^{1}) + (101 \times 126^{0}) = 1{,}568{,}675$
- $\text{2pac}_{126} = (50 \times 126^{3}) + (112 \times 126^{2}) + (97 \times 126^{1}) + (99 \times 126^{0})$
  $= 101{,}809{,}233$
- $\text{eGg!}_{126} = (98 \times 126^{3}) + (71 \times 126^{2}) + (98 \times 126^{1}) + (33 \times 126^{0})$
  $= 203{,}178{,}213$

Implementing `asciiToInt`

Below is a simple formula which converts a String to an integer.

- Treats String as a base 126 number.

```
public static int asciiToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i += 1) {
        intRep = intRep * 126;
        intRep = intRep + s.charAt(i);
    }
    return intRep;
}
```

What if we want to use characters beyond ASCII?

Going Beyond ASCII

chars in Java also support character sets for other languages and symbols.

- `char c = ’☂’` is equivalent to `char c = 9730`.
- `char c = ’鼈’` is equivalent to `char c = 40140`.
- `char c = ’혜’` is equivalent to `char c = 54812`.
- This encoding is known as Unicode. Table is too big to list.

Example: Computing Unique Representations of Kanji

The largest possible value for Kanji is 40,959*, so we'd need to use this as our base if we want to have a unique representation for all possible strings of Kanji.

Example:

- 横田誠司$_{40959}$ = $(27178 \times 40959^{3}) + (30000 \times 40959^{2}) + (35488 \times 40959^{1}) + (21496 \times 40959^{0})$ = 1,867,571,481,361,683,550

...	...
横田誠号	1,867,571,481,361,683,549
横田誠司	1,867,571,481,361,683,550
横田誠叹	1,867,571,481,361,683,551
...	...

*If you're curious, the last character is: 黎

Hash Codes

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- BobaCounterSet
- DynamicArrayOfListsSet
- lowercase strings
- Integer Overflow
- **Hash Codes**

Hash Tables in Java

Hash Table Performance and Summary

Creating a Good Hash Code (extra)

Linear Probing (extra)

Finitely Many Integers

So far, we've tried to map any possible string to a unique integer.

- But in Java, there are only finitely many integers.

That is, we tried to map 横田誠司 as a base 40959 number, yielding 1,867,571,481,361,683,550, but this number doesn't exist in Java as an int.

- Integer value grows exponentially with number of characters. Even limiting to haiku, we'll get numbers in the quinvigintillions.

Note: Other programming languages do not have finitely many integers. Python, for example, allows an integer to take on any value.

- On actual physical computers, some integers will not be able to be stored.
- Even when stored, large numbers tend to take much more time to do math on.

What Happens in Practice: Integer Overflow

In Java, the largest possible integer is 2,147,483,647.

- If you go over this limit, you overflow, starting back over at the smallest integer, which is -2,147,483,648.
- In other words, the next number after 2,147,483,647 is -2,147,483,648.

```java
int x = 2147483647;
System.out.println(x);
System.out.println(x + 1);
```

```
jug ~/Dropbox/61b/lec/hashing
$ javac BiggestPlusOne.java
$ java BiggestPlusOne
2147483647
-2147483648
```

Consequence of Overflow

Because Java has a maximum integer, we won’t get the numbers we expect!

- With base 126, we will run into overflow even for short strings.
  - Example: omens_{126} = 28,196,917,171, which is much greater than the maximum integer!
  - `asciiToInt(’omens’)` will give us -1,867,853,901 in Java.

Hash Codes

The official term for the number we're computing is "hash code".

- Via Wolfram Alpha: a hash code "projects a value from a set with many (or even an infinite number of) members to a value from a set with a fixed number of (fewer) members."
- Here, our target set is the set of Java integers, which is of size 4,294,967,296.

That is, our integerization function is a "hash code" because the set we're projected onto is fixed.

Java Uses Base 31

Because the range of our hashCode is finite, it is impossible to pursue unique factorizations for each String.

Instead of base 40,959 or something larger, **Java uses 31**.

- Fixed mod prevents the issue of having different mods for different Strings
- 横田誠司$_{40959}$ = (27178 x 40959^{3}) + (30000 x 40959^{2}) + (35488 x 40959^{1}) + (21496 x 40959^{0}) = 1,867,571,481,361,683,550
- 横田誠司$_{31}$ = (27178 x 31^{3}) + (30000 x 31^{2}) + (35488 x 31^{1}) + (21496 x 31^{0}) = 839,611,422

You c

```java
System.out.println("横田誠司".hashCode());
```

Java Uses Base 31

Because the range of our hashCode is finite, it is impossible to pursue unique factorizations for each String.

Instead of base 40,959 or something larger, **Java uses 31**.

- 橫田誠司$_{31}$ = (27178 x 31^3) + (30000 x 31^2) + (35488 x 31^1) + (21496 x 31^0) = 839,611,422

Of course there are infinitely many other strings that also map to 839,611,422.

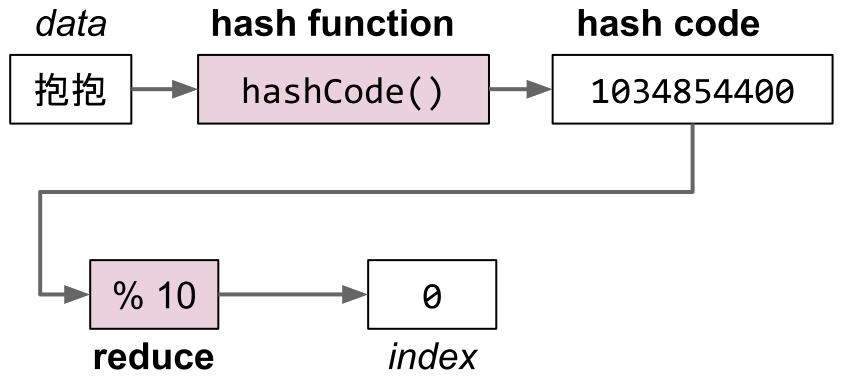
- Example: ±ErеWn$_{31}$ = (177 x 31^5) + (69 x 31^4) + (114 x 31^3) + (101 x 31^2) + (87 x 31^1) + (110 x 31^0) = 5,134,578,718
- After overflow, 5,134,578,718 is just 839,611,422.

```java
System.out.println("橫田誠司".hashCode());
System.out.println("±ErеWn".hashCode());
```

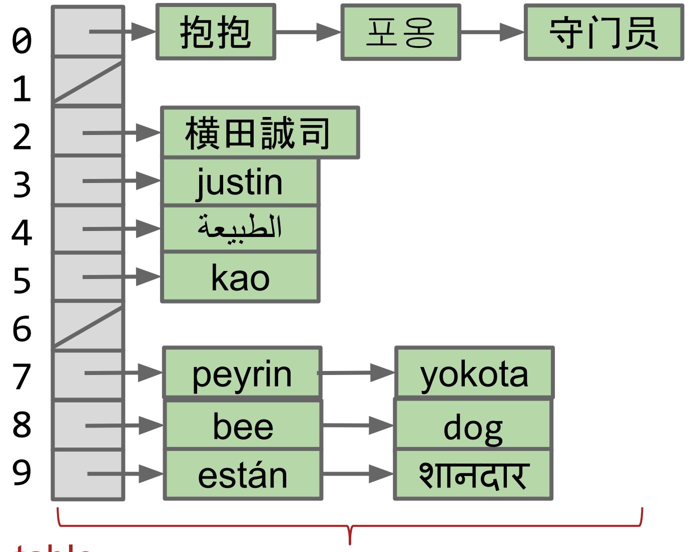
The Hash Table

What we've just created here is called a **hash table**.

- *Data* is converted by a **hash function** into an integer representation called a **hash code**. Range of possible hash codes is -2,147,483,648 to 2,147,483,647.
- The **hash code** is then **reduced** to a valid *index*, usually using the modulus operator, e.g. 2348762878 % 10 = 8.



In Java there's a caveat here. Will revisit later.

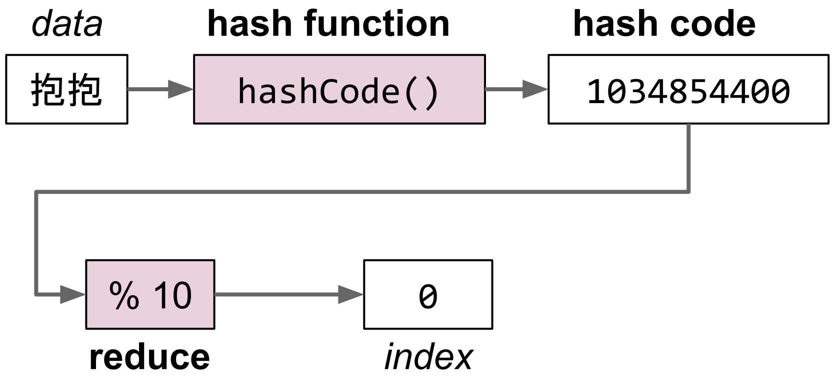


hash table

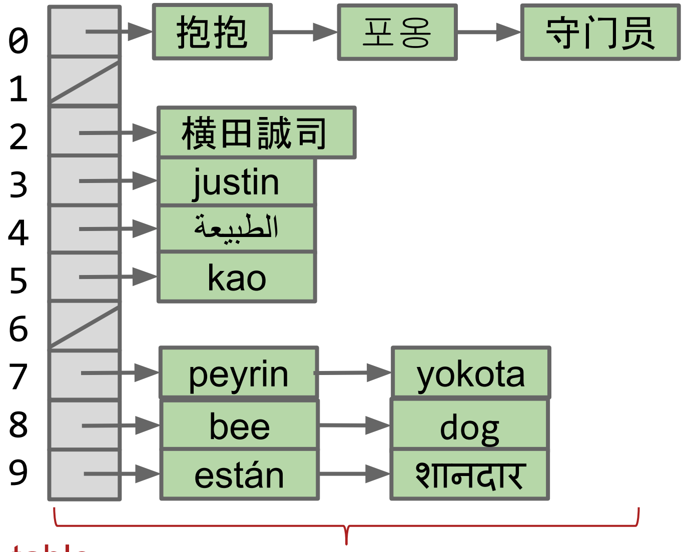
The Hash Table

Note, there are other versions of hash tables out there.

- The version we're using is an array of lists.
- This is sometimes called "separate chaining", where each bucket is a separate chain of items.
- Many more exotic solutions exist (linear probing, cuckoo hashing, using things other than linked lists for buckets, etc).



In Java there's a caveat here. Will revisit later.



hash table

Hash Tables in Java

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- BobaCounterSet
- DynamicArrayOfListsSet
- lowercase strings
- Integer Overflow
- Hash Codes

Hash Tables in Java

Hash Table Performance and Summary

Creating a Good Hash Code (extra)

Linear Probing (extra)

The Ubiquity of Hash Tables

Hash tables are the most popular implementation for sets and maps.

- Great performance in practice.
- Don’t require items to be comparable.
- Implementations often relatively simple.
- Python dictionaries are just hash tables in disguise.

In Java, implemented as `java.util.HashMap` and `java.util.HashSet`.

- How does a HashMap know how to compute each object’s hash code?
    - Good news: It’s not “`implements Hashable`”.
    - Instead, all objects in Java must implement a `.hashCode()` method.

Object Methods

All classes are hyponyms of `Object`.

- `String toString()`
- **`boolean equals(Object obj)`**
- `int hashCode()`
- `Class<?>  getClass()`
- `protected Object clone()`
- `protected void finalize()`
- `void notify()`
- `void notifyAll()`
- `void wait()`
- `void wait(long timeout)`
- `void wait(long timeout, int nanos)`

From earlier in class.

This is where Java implements hash codes.

Won't discuss or use in 61B.

Default implementation of `hashCode` returns memory address.

Hash Codes in Java

Java’s actual hashCode function for Strings below (code cleaned up slightly):

- “横田誠司” and “±EreWn” map to 839,611,422.

```
public int hashCode(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i += 1) {
        intRep = intRep * 31;
        intRep = intRep + s.charAt(i);
    }
    return intRep;
}
```

That is, the two calls below both return 839,611,422.

- “横田誠司”.hashCode()
- “±EreWn”.hashCode()

More examples of Real Java HashCodes for Strings

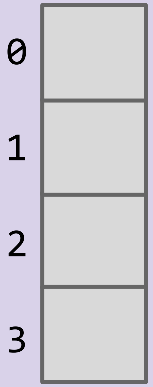
```java
System.out.println("a".hashCode());
System.out.println("bee".hashCode());
System.out.println("포옹".hashCode());
System.out.println("kamala lifefully".hashCode());
System.out.println("đậu hũ".hashCode());
```

```
                    jug ~/Dropbox/61b/lec/hashing
                    $ java JavaHashCodeExamples
                 "a" 97
               "bee" 97410
                "포옹" 1732557
  "kamala lifefully" 1732557
            "đậu hũ" -2108180664
```

Using Negative hash codes: yellkey.com/above

Suppose that 's hash code is `-1`.

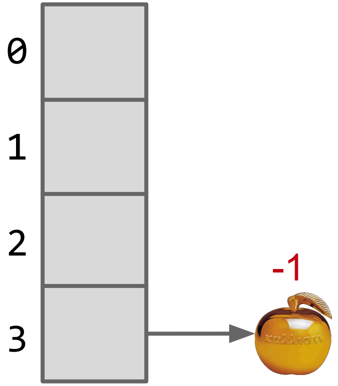
- Philosophically, into which bucket is it most natural to place this item?



Using Negative hash codes

Suppose that 's hash code is `-1`.

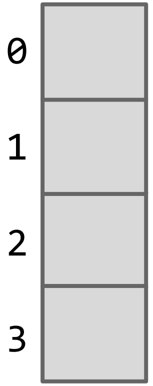
- Philosophically, into which bucket is it most natural to place this item?
  - I say 3, since -1 → 3, 0 → 0, 1 → 1, 2 → 2, 3 → 3, 4 → 0, ...



Using Negative hash codes in Java

Suppose that 's hash code is -1.

- Unfortunately, -1 % 4 = -1. Will result in index errors!
- Use Math.floorMod instead.



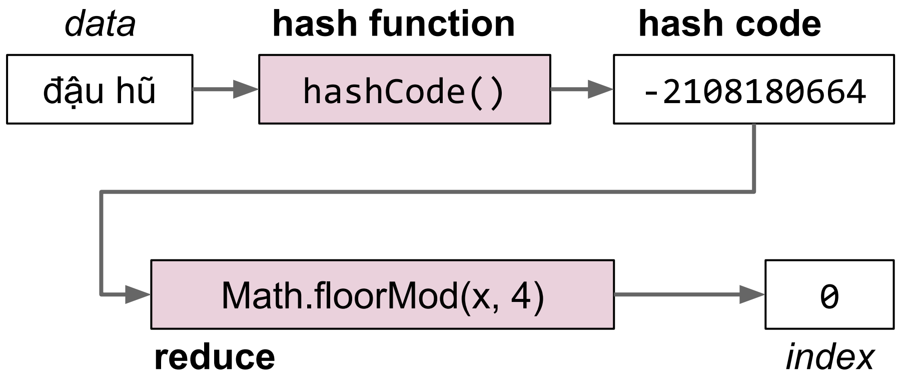
```java
public class ModTest {
  public static void main(String[] args) {
    System.out.println(-1 % 4);
    System.out.println(Math.floorMod(-1, 4));
  }
}
```

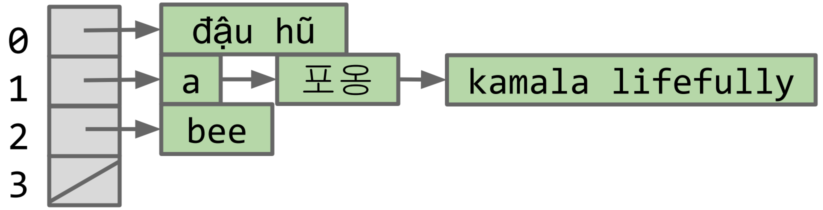
```
$ java ModTest
-1
3
```

Hash Tables in Java

Java hash tables:

- *Data* is converted by the **hashCode** method an integer representation called a **hash code**.
- The **hash code** is then **reduced** to a valid *index*, using something like the floorMod function, e.g. Math.floorMod(1732557 % 4) = 8.





Two Important Warnings When Using HashMaps/HashSets

Warning #1: Never store objects that can change in a `HashSet` or `HashMap`!

- Such objects are also called “mutable” objects, e.g. they can change.
  - Example: You’d never want to make a `HashSet<List<Integer>>`.
- If an object’s variables changes, then its hashCode changes. May result in items getting lost.

Warning #2: Never override `equals` without also overriding `hashCode`.

- Can also lead to items getting lost and generally weird behavior.
- HashMaps and HashSets use equals to determine if an item exists in a particular bucket.

We’ll come back to these warnings later.

Hash Table Performance and Summary

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- BobaCounterSet
- DynamicArrayOfListsSet
- lowercase strings
- Integer Overflow
- Hash Codes

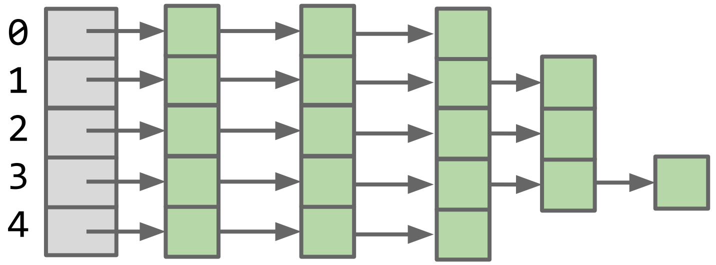
Hash Tables in Java

Hash Table Performance and Summary

Creating a Good Hash Code (extra)

Linear Probing (extra)

Hash Table Runtime with No Resizing



N = 19 M = 5 N / M = 3.8

Suppose we have:

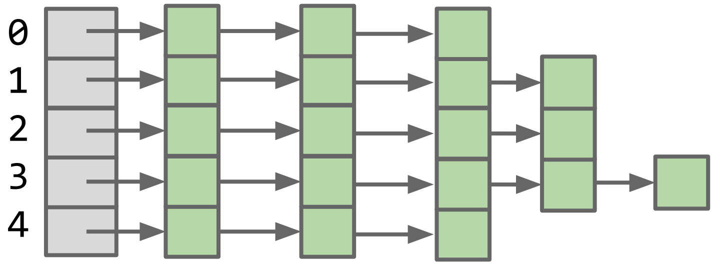
- An fixed number of buckets M.
- An increasing number of items N.

Average list is around N/M items

Even if items are spread out evenly, lists are of length Q = N/M.

- For M = 5, that means Q = Θ(N). Results in linear time operations.

Resizing Hash Table Runtime



N = 19 M = 5 N / M = 3.8

Suppose we have:

- An increasing number of buckets M.
- An increasing number of items N.

As long as M = Θ(N), then O(N/M) = O(1).

Assuming items are evenly distributed (as above), lists will be approximately N/M items long, resulting in Θ(N/M) runtimes.

- By doubling every time N gets too big, we ensure that N/M = O(1).
- Thus, worst case runtime for all operations is Θ(N/M) = Θ(1).
  - … unless that operation causes a resize.
  - … and again, we're assuming even distribution of items.

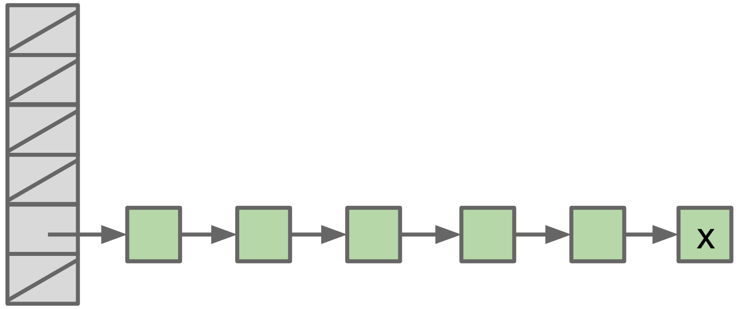
Regarding Even Distribution

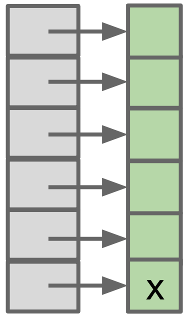
Even distribution of item is critical for good hash table performance.

- Both tables below have load factor of N/M = 1.
- Left table is much worse!
  - Contains is Θ(N) for x.

Will need to discuss how to ensure even distribution.

- See extra video and slides for more on ensuring an even distribution.

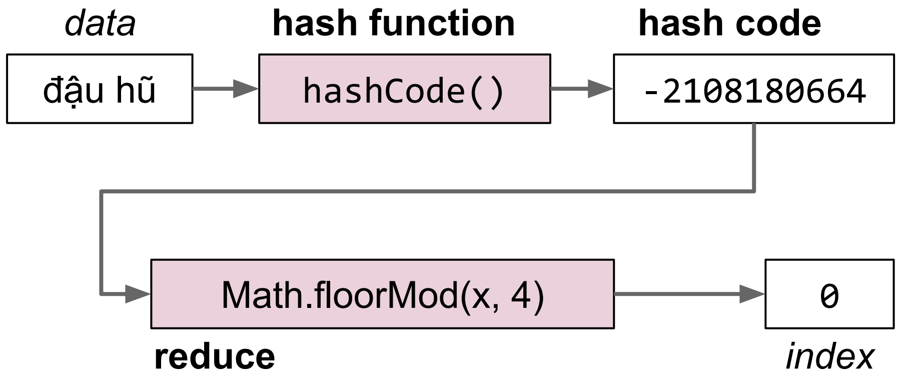




Hash Tables in Java

Hash tables:

- *Data* is converted into a hash code.
- The **hash code** is then **reduced** to a valid *index*.
- *Data* is then stored in a bucket corresponding to that *index*.
- Resize when load factor N/M exceeds some constant.
- If items are spread out nicely, you get Θ(1) average runtime.



	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
Separate Chaining Hash Table With No Resizing	Θ(N)	Θ(N)
… With Resizing	Θ(1)†	Θ(1)*†

*: Amortized.
†: Assuming items are evenly spread.

Creating a Good Hash Code (extra)

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- BobaCounterSet
- DynamicArrayOfListsSet
- lowercase strings
- Integer Overflow
- Hash Codes

Hash Tables in Java

Hash Table Performance and Summary

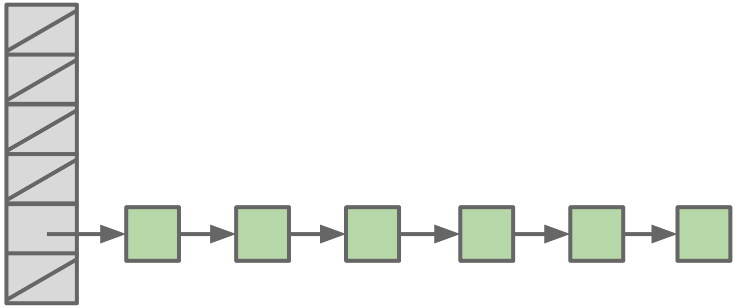
Creating a Good Hash Code (extra)

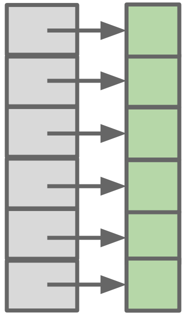
Linear Probing (extra)

What Makes a good .hashCode()?

Goal: We want hash tables that look like the table on the right.

- Want a hashCode that spreads things out nicely on real data.
  - Example #1: return 0 is a bad hashCode function.
  - Example #2: just returning the first character of a word, e.g. “cat” → 3 was also a bad hash function.
  - Example #3: Adding chars together is bad. “ab” collides with “ba”.
  - Example #4: returning string treated as a base B number can be good.
- Writing a good hashCode() method **can be tricky**.





Hashbrowns and Hash Codes

How do you make hashbrowns?

- Chopping a potato into nice predictable segments? No way!
- Similarly, adding up the characters is not nearly “random” enough.

Can think of multiplying data by powers of some base as ensuring that all the data gets scrambled together into a seemingly random integer.

Example hashCode Function

The Java 8 hash code for strings. Two major differences from our hash codes:

- Represents strings as a base 31 number.
    - Why such a small base? Real hash codes don’t care about uniqueness.
- Stores (caches) calculated hash code so future `hashCode` calls are faster.

```java
@Override
public int hashCode() {
    int h = cachedHashValue;
    if (h == 0 && this.length() > 0) {
        for (int i = 0; i < this.length(); i++) {
            h = 31 * h + this.charAt(i);
        }
        cachedHashValue = h;
    }
    return h;
}
```

Example: Choosing a Base

Java's `hashCode()` function for Strings:

- $h(s) = s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + \ldots + s_{n-1}$

Our `asciiToInt` function for Strings:

- $h(s) = s_0 \times 126^{n-1} + s_1 \times 126^{n-2} + \ldots + s_{n-1}$

Which is better?

- Might seem like 126 is better. Ignoring overflow, this ensures a unique numerical representation for all ASCII strings.
- … but overflow is a particularly bad problem for base 126!

Example: Base 126

Major collision problem:

- “geocronite is the best thing on the earth.”.hashCode() yields 634199182.
- “flan is the best thing on the earth.”.hashCode() yields 634199182.
- “treachery is the best thing on the earth.”.hashCode() yields 634199182.
- “Brazil is the best thing on the earth.”.hashCode() yields 634199182.

Any string that ends in the same last 32 characters has the same hash code.

- Why? Because of overflow.
- Basic issue is that 126^32 = 126^33 = 126^34 = ... 0.
  - Thus upper characters are all multiplied by zero.
  - See CS61C for more.

Typical Base

A typical hash code base is a small prime.

- Why prime?
  - Never even: Avoids the overflow issue on previous slide.
  - Lower chance of resulting hashCode having a bad relationship with the number of buckets: See study guide problems and hw3.
- Why small?
  - Lower cost to compute.

A full treatment of good hash codes is well beyond the scope of our class.

Hashbrowns and Hash Codes

How do you make hashbrowns?

- Chopping a potato into nice predictable segments? No way!

Using a prime base yields better "randomness" than using something like base 126.

Example: Hashing a Collection

Lists are a lot like strings: Collection of items each with its own hashCode:

```
@Override
public int hashCode() {
    int hashCode = 1;
    for (Object o : this) {
        hashCode = hashCode * 31;
        hashCode = hashCode + o.hashCode();
    }
    return hashCode;
}
```

elevate/smear the current hash code

add new item’s hash code

To save time hashing: Look at only first few items.

- Higher chance of collisions but things will still work.

Example: Hashing a Recursive Data Structure

Computation of the hashCode of a recursive data structure involves recursive computation.

- For example, binary tree hashCode (assuming sentinel leaves):

```
@Override
public int hashCode() {
   if (this.value == null) {
       return 0;
   }
   return  this.value.hashCode() +
    31 * this.left.hashCode() +
    31 * 31 * this.right.hashCode();
}
```

Linear Probing (extra)

Lecture 19, CS61B, Spring 2024

Motivation, Set Implementations

Deriving Hash Tables

- WriteItOnTheWallSet
- BobaCounterSet
- DynamicArrayOfListsSet
- lowercase strings
- Integer Overflow
- Hash Codes

Hash Tables in Java

Hash Table Performance and Summary

Creating a Good Hash Code (extra)

Linear Probing (extra)

Open Addressing: An Alternate Disambiguation Strategy (Extra)

Instead of using linked lists, an alternate and more exotic strategy is “open addressing”.

- Set is stored as an array of items. Index tells you where to put the item.

If target location is already occupied, use a different location, e.g.

- Linear probing: Use next address, and if already occupied, just keep scanning one by one.
  - Demo: http://goo.gl/o5EDvb
- Quadratic probing: Use next address, and if already occupied, try looking 4 ahead, then 9 ahead, then 16 ahead, …
- Many more possibilities. See the optional reading for today (or CS170) for a more detailed look.

In 61B, we’ll use the “separate chaining” approach, where we have linked lists.

Citations

http://www.nydailynews.com/news/national/couple-calls-911-forgotten-mcdonalds-hash-browns-article-1.1543096

http://en.wikipedia.org/wiki/Pigeonhole_principle#mediaviewer/File:TooManyPigeons.jpg

https://cookingplanit.com/public/uploads/inventory/hashbrown_1366322674.jpg

FAQ

What is the distinction between hash set, hash map, and hash table?

A hash set is an implementation of the Set ADT using the “hash table” as its engine.

A hash map is an implementation of the Map ADT using the “hash table” as its engine.

A “hash table” is a way of storing information, where you have M buckets that store N items. Each item has a “hashCode” that tells you which of M buckets to put that item in.