K-d Trees; Hashing

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Aaron Johnston
Amanda Park
Anish Velagapudi
Brian Chan
Elena Spasova
Ethan Knutson
Farrell Fileas
Howard Xiao
Jade Watkins
Lea Quan
Nathan Lipiarski
Sam Long
Yifan Bai
Yuma Tou

Announcements

- New workshops on *Wednesdays*
  - 2:30-3:20 in CSE1 203
  - We also have workshops Friday 11:30-12:20 in CSE 203
  - Topic survey is being phased out
- HW 4 (heaps) due *Wednesday* (🙌 extra day!!! 🙌)
  - But HW 5 (k-d Tree) will be still be released on Tuesday

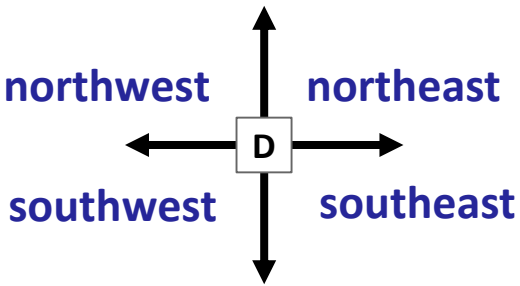
Lecture Outline

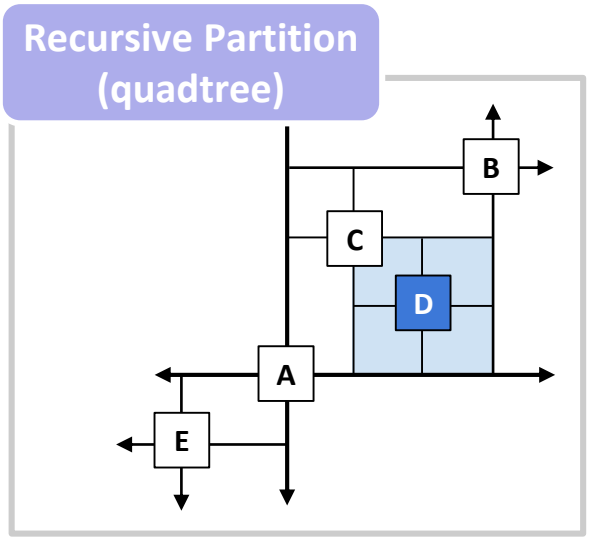
- **Multidimensional Data, cont.: k-d trees**

- Hashing
  - Designing Our Own Hash Function
  - Hashing Applications

Review: Quadtree

- **2-dimensional data: Quadtree**
  - Keys are located on a plane
  - Recursive decision: northwest, northeast, southwest, southeast



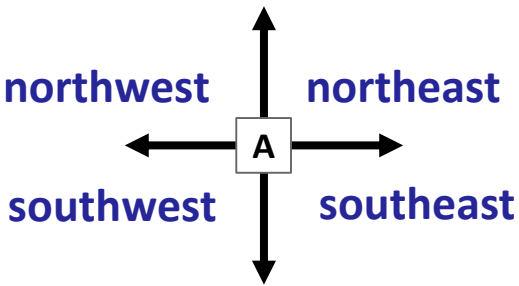


Another approach: k-d Trees

- **Quadtree**: pick the "single correct region" at each recursive step
- **k-d Tree**: pick "partially-correct region**s**" at each recursive step; we'll select the correct region after k recursive steps

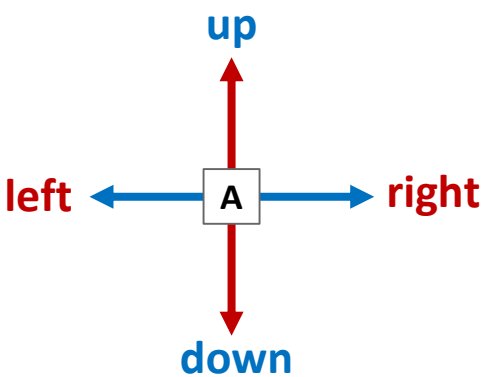
2-dimensional data: Quadtree

- Recursive decision: northwest, northeast, southwest, southeast



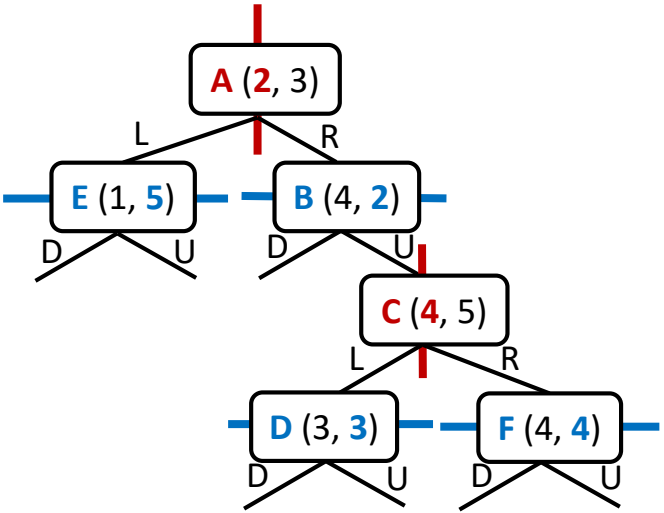
2-dimensional data: 2-d tree

- Recursive decision k_1: left or right
- Recursive decision k_2: up or down



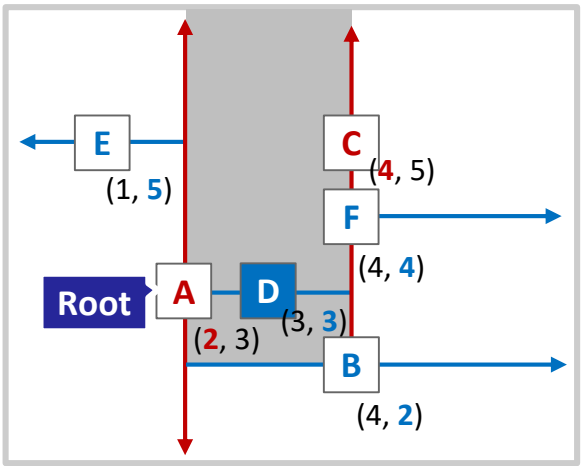
2-d Tree

- The root node partitions entire space left and right (by x-coordinate)
- All depth 1 nodes partition its subspaces into up and down (by y-coordinate)
- All depth 2 nodes partition its subspaces into left and right (by x-coordinate)
- …



2-d Tree

- Each point owns 2 subspaces whose dimensions may be constrained by its ancestors
  - D's subspaces are constrained on the left and right by its ancestors A and C
  - The subspace below D is constrained by its ancestor B
  - The subspace above D is infinitely large



Poll Everywhere

pollev.com/uwcse373

Where would G (5, 3) go in our 2-d tree?

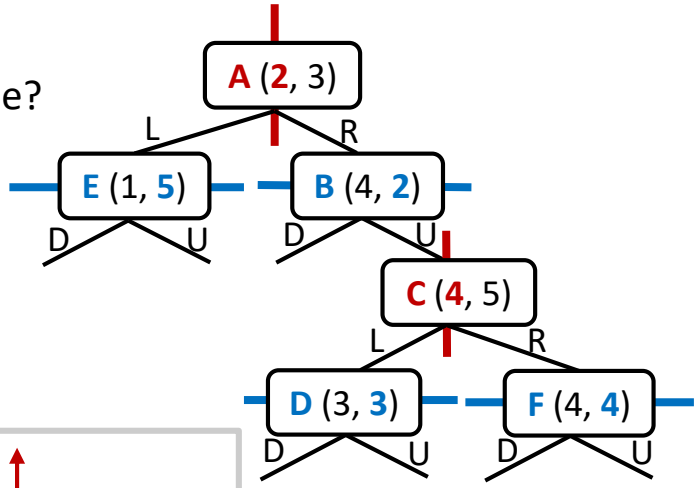
A. Right child of D

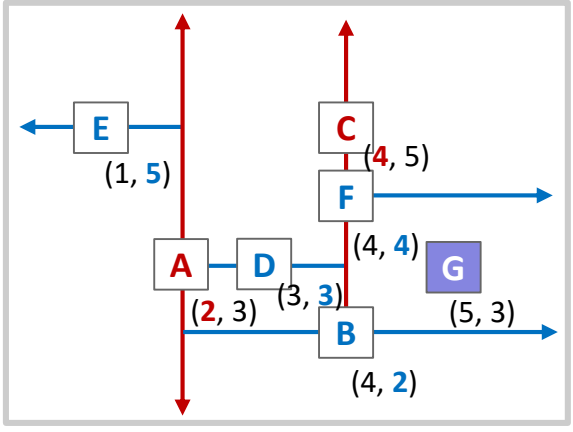
B. Left child of B

C. Left child of F

D. Right child of F

E. I'm not sure …





k-d Tree: Insertion

- Walk down the tree until you find a suitable leaf, then add
  - What, if any, balance properties does a k-d Tree have?

- 2-d Tree Insertion Demo: https://docs.google.com/presentation/d/1WW56RnFa3g6UJEquuIBymMcu9k2nqLrOE1ZlnTYFebg/present?ueb=true&slide=id.g54b6045b73_0_38

k-d Tree: Nearest Neighbour

- Walk the tree, visiting every node but exploring the "better " side first
  - ... Can we "prune " known-bad sides?
- 2-d Tree Nearest Neighbour
  - Demo: https://docs.google.com/presentation/d/1DNunK22t-4OU_9c-OBgKkMAdly9aZQkWuv_tBkDg1G4/edit#slide=id.g54b6045cf5_150_1378
  - Video: https://www.youtube.com/watch?v=mxrUFkdXaR8

```
nearest(Node n, Point goal, Node best):
  if n is null:
    return best
  if n.distance(goal)
      < best.distance(goal):
    best = n
  if goal < n:
    goodSide = n."left"Child
    badSide = n."right"Child
  else:
    goodSide = n."right"Child
    badSide = n."left"Child
  best = nearest(goodSide, goal, best)
  if mightHaveSomethingUseful(badSide):
    best = nearest(badSide, goal, best)
  return best
```

optimization #1: determine order of sides to examine

optimization #2 Prune "known bad" subtrees

Applications

- Lots of simulations require finding nearest neighbor (or k-nearest neighbor)
  - Astronomy, biology, etc.

- Range-searching multidimensional data used often in machine learning and other optimization problems
  - Eg, your Instagram profile has gender, age range, preference level for home décor, preference level for DIY, etc. If an advertiser wants to reach the 10,000 "best" customers for its face cream, whom should be targeted?

Multidimensional Data: Summary

- Operations:
  - Range Searching: What are all the objects inside this (rectangular) region?
  - Nearest Neighbour: What is the closest object top a specific point (this is often the k-nearest in machine learning)

- Spatial Partitioning: Dividing space into non-overlapping subspaces, allowing us to prune the search space.
  - Uniform partitioning
  - Quadtree
  - k-d Tree

Lecture Outline

- Multidimensional Data, cont.: k-d trees

- **Hashing**
  - Designing Our Own Hash Function
  - Hashing Applications

Feedback from the Reading Quiz

- Is hashing related to HashSets/HashMaps?
- Do we need to use hash functions in HW4?
- What do we do when we have collisions?
- What is uniformity and why is it important?

What is Hashing?

- **Hashing** is taking data of arbitrary size and type and converting it to an fixed-size integer (ie, an integer in a predefined range)

- Running example: design a hash function that maps strings to 32-bit integers [-2147483648, 2147483647]

- A good hash function exhibits the following properties:
  - Deterministic: the same input should generate the same output
  - Efficiency: it should take a reasonable amount of time
  - Uniformity: inputs should be spread "evenly" over its output range

Bad Hashing

```
int hashFn(String s) {
  return
    Random.nextInt();
}
```

Deterministic?

```
int hashFn(String s) {
  int retVal = 0;

  for (int i = 0;
    i < s.length();
    i++) {

    for (int j = 0;
      j < s.length();
      j++) {
     retVal += helperFn(
       s, i, j);
    }
  }

  return retVal;
}
```

Efficient?

```
int hashFn(String s) {
  if (s.length()%2 == 0)
    return 17;
  else
    return 42;
}
```

Uniform?

Lecture Outline

- Multidimensional Data, cont.: k-d trees

- Hashing
  - **Designing Our Own Hash Function**
  - Hashing Applications

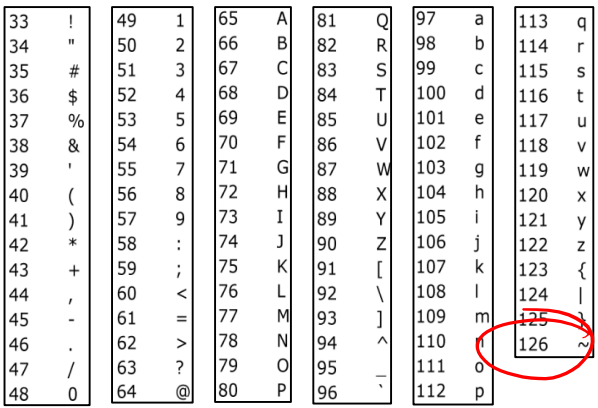
Attempt #1: hash("cat")

- One idea: Assign each letter a number, use the first letter of the word
  - a = 1, b = 2, c = 3, …, z = 26
  - hash("cat") == 3

- What's wrong with this approach?
  - Other words start with c
    - hash("chupacabra") == 3
  - Can't hash "=abc123"

Attempt #2: hash("cat")

- Next idea: Add together all the letter codes, add new values for symbols
  - hash("cat") == 99 + 97 + 116 == 312
  - hash("=abc123") == 505
- What's wrong with this approach?
  - Other words with the same letters
    - hash("act") == 97 + 99 + 116 == 312



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		

Attempt #3: hash("cat")

- Max possible value for English-only text (including punctuation) is 126
- Another idea: Use 126 as our base to ensure unique values across all possible strings
  - hash("cat") == $99*126^0 + 97*126^1 + 116*126^2$ == 232055937
  - hash("act") == $97*126^0 + 99*126^1 + 116*126^2$ == 232056187
- What's wrong with this approach?
  - Only handles English!

Attempt #4: hash("cat")

- If we switch to another character set we can encode strings such as "¡Hola!"
  - The Unicode "Basic Multilingual Plane" contains 65,472 codepoints

- hash("cat") == $99*65472^0 + 97*65472^1 + 116*65472^2$ == 497,249,953,827

- What's wrong with this approach?
  - Our range was [-2,147,483,648, 2,147,483,647]
    - 497,249,953,827 % 2,147,483,647 == 1,181,231,370 == hash("覞")
  - We could use the modulus operator (%) to "wrap around", but now we've introduced the possibility of collisions
  - The BMP excludes most emoji (), characters outside the "Han Unification" (兩 vs 两 vs 両 vs 网), and much, much more

hash("cat"): Lessons Learned

- Writing a hash function is hard!
  - So don't do it ☺
- Common hash algorithms include:
  - MD5
  - SHA-1
  - SHA-256
    - the only one that hasn't been proven to be *cryptographically insecure* (yet)
  - xxHash
  - CityHash
  - SuperFastHash

Lecture Outline

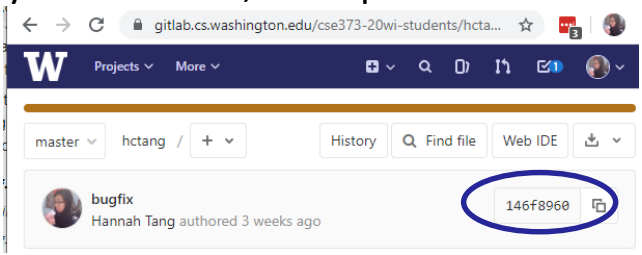
- Multidimensional Data, cont.: k-d trees

- Hashing
  - Designing Our Own Hash Function
  - **Hashing Applications**

Content Hashing: Applications

- Caching:
  - You've downloaded a large video file. You want to know if a new version is available. Rather than re-downloading the entire file, compare your file's hash value with the server's hash value.

- File Verification / Error Checking:
  - Same implementation
  - Can be used to verify files on your machine, files spread across multiple servers, etc.



- Fingerprinting
  - Git hashes ("identification")
  - Ad tracking ("identification"): see https://panopticlick.eff.org/
  - YouTube ContentID ("duplicate detection")

Content Hashing: Defining a Salient Feature

- Hash function implementors can choose what's salient:
  - hash("cat") == hash("CAT") ???

- What's salient in detecting that an image or video is unique?

- What's salient in determining that a user is unique?

Content Hashing vs Cryptographic Hashing

- In addition to the properties of "regular" hash functions, cryptographic hashes also have the following properties:
  - It is infeasible to find or generate two different inputs that generate the same hash value
  - Given a hash value, it is infeasible to calculate the original input
  - Small changes to the input generate an uncorrelated hash values

- Security is *very hard* to get right!
  - If you don't know what you're doing, you're probably making it worse
  - Most algorithms, including MD5 and SHA-1, are not cryptographically secure

Poll Everywhere

pollev.com/uwcse373

- Can hashing be appropriately used for this application?
  - Verifying files or messages are untampered ("integrity")
  - Verifying the identity of the other party ("authentication")
  - Verifying that an entered password matches a previous password *without storing the password itself*

A. Yes / Yes / No
B. Yes / Yes / Yes
C. Yes / No / No
D. Yes / No / Yes
E. I'm not sure …

tl;dr

- **k-d Trees** allow you to recursively partition k-dimensional data using a k 2-way questions

Range Search	Nearest Neighbour	Add
Θ(log N)	Θ(log N)	Θ(log N)

- Hash functions map arbitrary data to a fixed-size integer
  - Please don't write your own hash function if you don't have to
  - There are lots of cool applications of hashing (see next lecture!)