Tries

CSE 373 Winter 2020

Guest Instructor: Aaron Johnston!

Teaching Assistants:

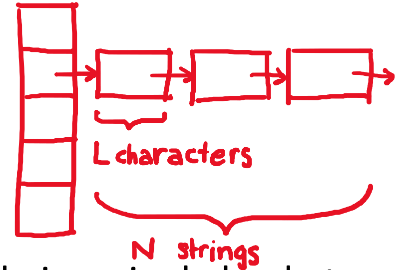
Aaron Johnston	Ethan Knutson	Nathan Lipiarski
Amanda Park	Farrell Fileas	Sam Long
Anish Velagapudi	Howard Xiao	Yifan Bai
Brian Chan	Jade Watkins	Yuma Tou
Elena Spasova	Lea Quan	

Announcements

- HW7 is out
  - Due this Friday, February 28
  - Lots of code to look through! Start early

- Midterm Regrades are open
  - Please consult the posted sample solution before submitting a regrade request

Feedback from the Reading Quiz



- Why is contains O(NL) for a hash table?
  - Consider the worst case, where all strings collide in a single bucket. That means scanning through N strings.
  - **It takes time to compare strings** – we have to go character by character!
  - For each string, there may be L characters to examine.

- How does DataIndexedCharMap relate to a trie?
  - We need a mapping from a character to the corresponding child in each node of the trie

- How to pronounce trie? "try"

Learning Objectives

- **By the end of today's lecture, you should be able to:**
  - Identify when a Trie can be used, and what useful properties it provides
  - Describe common Trie implementations and how they affect the amount of space required
  - Write code for prefix algorithms to run over a Trie

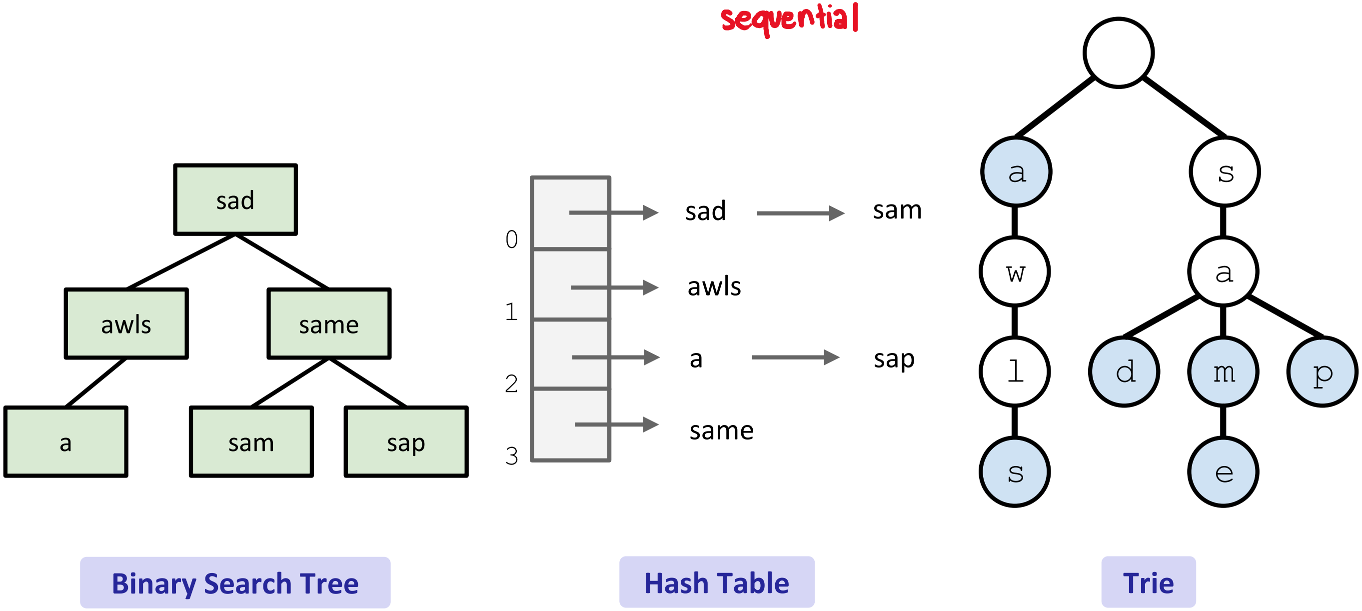
Lecture Outline

- **Tries**
  - **When does a Trie make sense?**
- Implementing a Trie
  - How do we find the next child?
- Advanced Implementations: Dealing with Sparsity
  - Hash Tables, BSTs, Ternary Search Tries
- Prefix Operations
  - Finding keys with a given prefix

The Trie: A Specialized Data Structure

Tries are a character-by-character **set-of-strings implementation**.

must be sequential



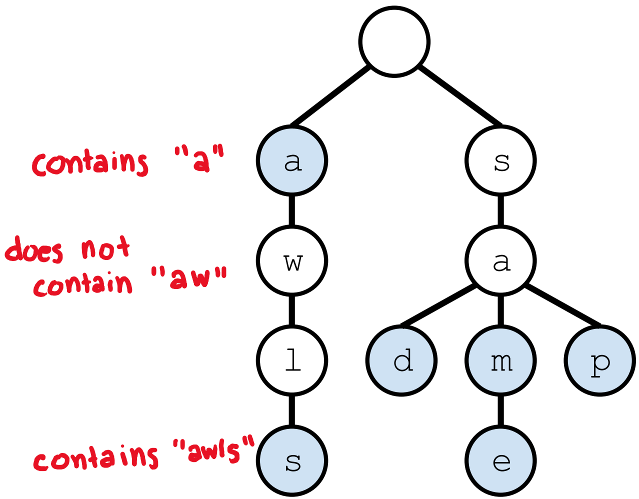
An Abstract Trie

Each level of the tree represents an index, and the children represent possible characters at that index.

This trie stores the set of strings:

`awls, a, sad,`

`same, sap, sam`



How to deal with `a` and `awls`?

- Mark which nodes *complete* strings (shown in blue)

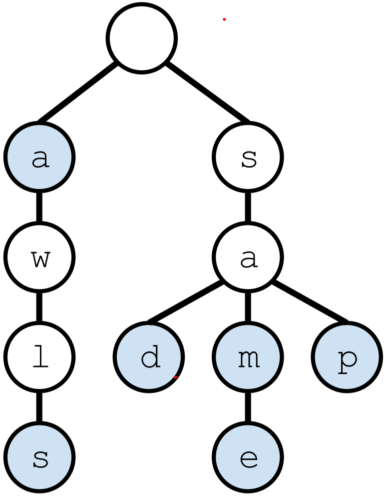
Searching in Tries

contains("sam"): true, blue. **hit**.

contains("sa"): false, white. **miss**.

contains("a"): true, blue. **hit**.

contains("saq"): false, fell off. **miss**.



Two ways to have a **search miss**.

1. If the final node is not blue (not a key).
2. If we fall off the tree.

Poll Everywhere

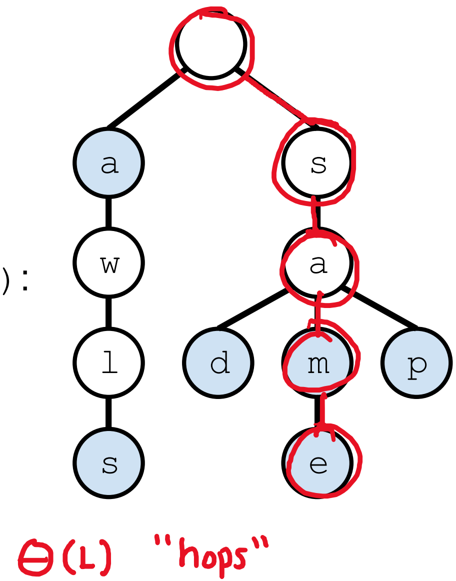
pollev.com/uwcse373

Given a trie with N keys, what is the runtime for `contains` given a key of length L?

A. $\Theta(\log L)$
B. $\Theta(L)$
C. $\Theta(\log N)$
D. $\Theta(N)$
E. $\Theta(N + L)$
F. We're not sure

In this trie:
N = 6

For `contains("same")`:
L = 4



Lecture Outline

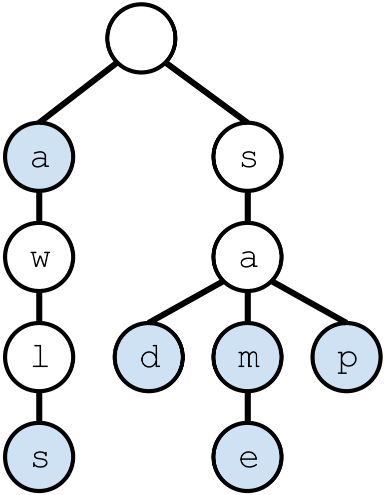
- Tries
  - When does a Trie make sense?
- **Implementing a Trie**
  - **How do we find the next child?**
- Advanced Implementations: Dealing with Sparsity
  - Hash Tables, BSTs, Ternary Search Tries
- Prefix Operations
  - Finding keys with a given prefix

Simple Trie Implementation

Design 1

```
public class TrieSet {
  private static final int R = 128; // ASCII
  private Node root;

  private static class Node {
    private char ch;
    private boolean isKey;
    private DataIndexedCharMap<Node> next;
    private Node(char c, boolean b, int R) {
      ch = c; isKey = b;
      next = new DataIndexedCharMap<Node>(R);
    }
  }
}
```



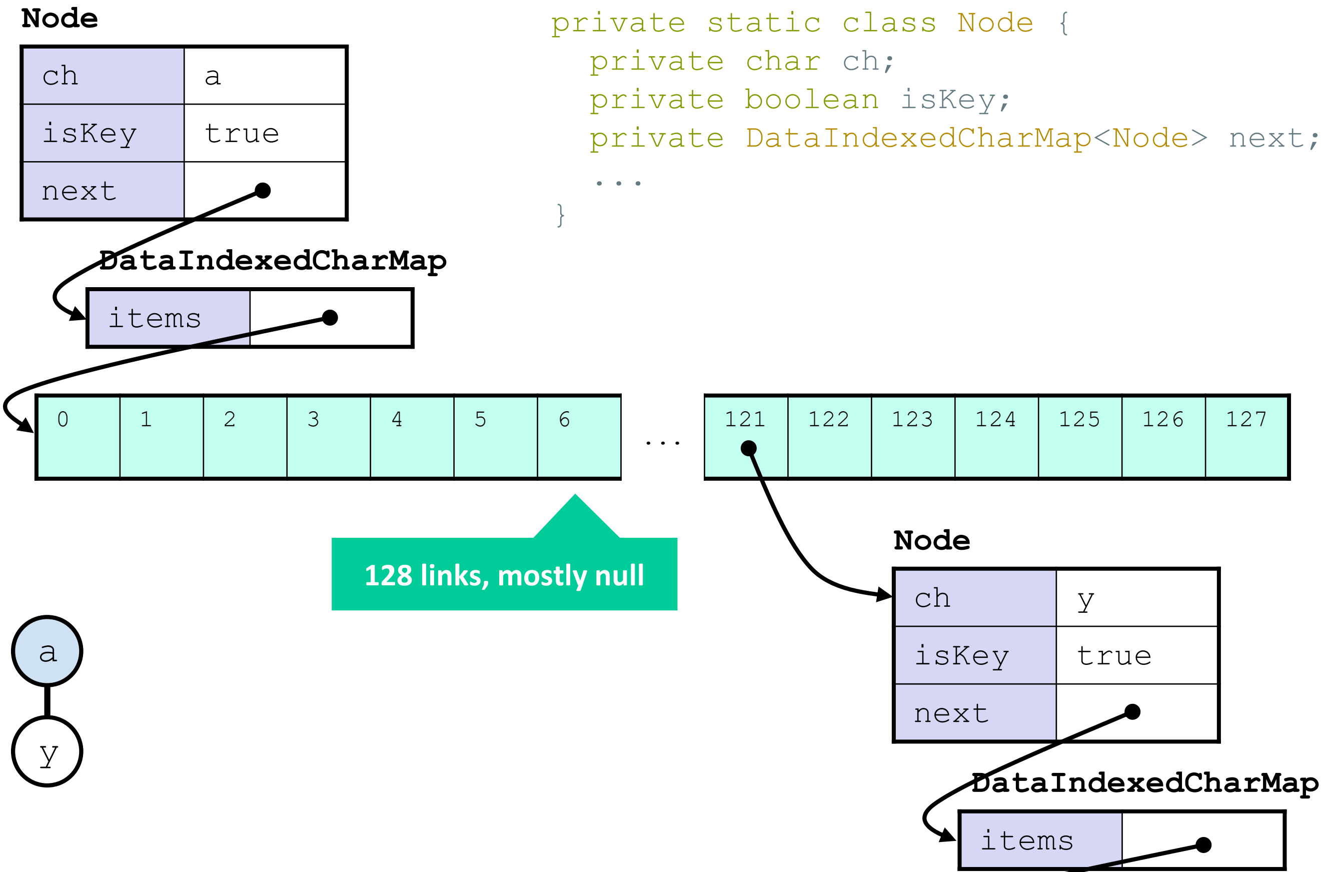
ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Simple Trie Node Implementation

Design 1

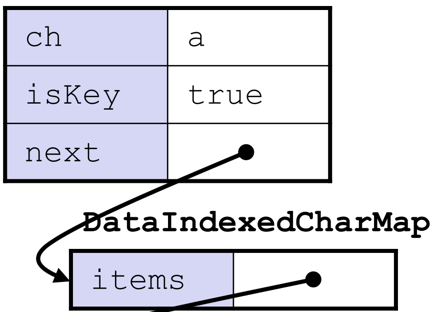
```
private static class Node {
   private char ch;
   private boolean isKey;
   private DataIndexedCharMap<Node> next;
   ...
}
```



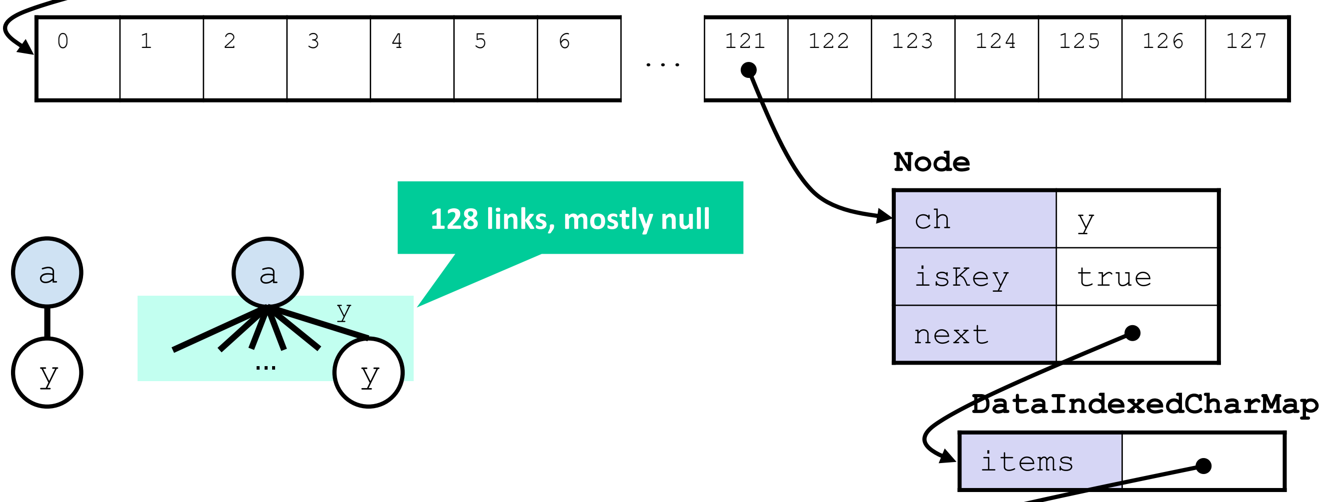
Simple Trie Node Implementation

Design 1

Node




```
private static class Node {
  private char ch;
  private boolean isKey;
  private DataIndexedCharMap<Node> next;
  ...
}
```

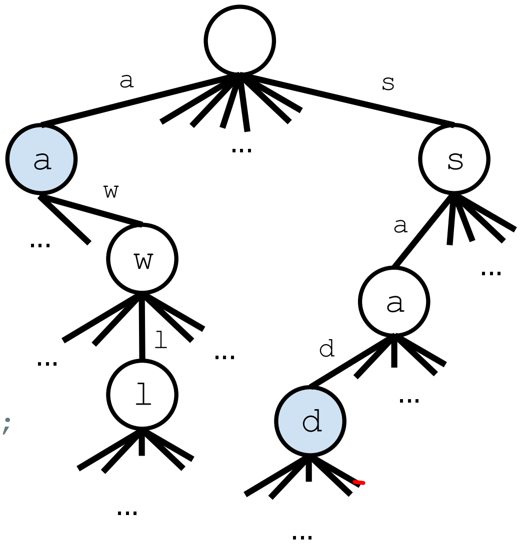


Simple Trie Implementation

Design 1

```
public class TrieSet {
  private static final int R = 128; // ASCII
  private Node root;

  private static class Node {
    private char ch;
    private boolean isKey;
    private DataIndexedCharMap<Node> next;
    private Node(char c, boolean b, int R) {
      ch = c; isKey = b;
      next = new DataIndexedCharMap<Node>(R);
    }
  }
}
```



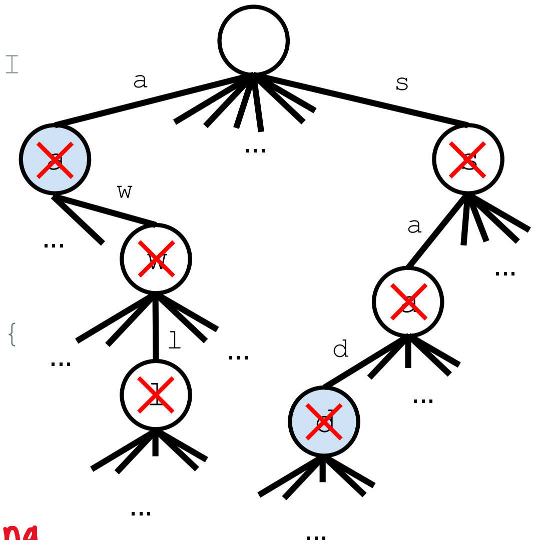
Removing Redundancy

Design 1.5

```
public class TrieSet {
  private static final int R = 128; // ASCII
  private Node root;

  private static class Node {
    private char ch;
    private boolean isKey;
    private DataIndexedCharMap<Node> next;
    private Node(char c, boolean b, int R) {
      ch = c; isKey = b;
      next = new
DataIndexedCharMap<Node>(R);
    }
  }
}
```

already know char from accessing map

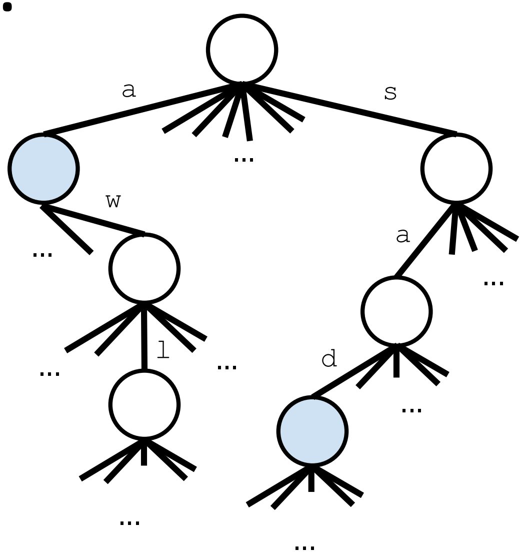


Poll Everywhere

pollev.com/uwcse373

Does the structure of a trie depend on the order in which strings are inserted?

A. Yes
B. No
C. We're not sure



Trie Runtimes

Design 1.5

Typical runtime when treating length of keys as a constant

	Key Type	`contains(x)`	`add(x)`
Balanced BST	Comparable	$\Theta(\log N)$	$\Theta(\log N)$
Hash Table	Hashable	$\Theta(1)^*$	$\Theta(1)^{*\dagger}$
Data-Indexed Array	Char	$\Theta(1)$	$\Theta(1)$
Trie (Design 1.5)	String	$\Theta(1)$	$\Theta(1)$

* : Assuming items are evenly spread

† : Indicates "on average"

Takeaways:

- When our keys are strings, Tries give us slightly better performance on contains and add.

- However, DataIndexedCharMap wastes a ton of memory storing R links per node.

Lecture Outline

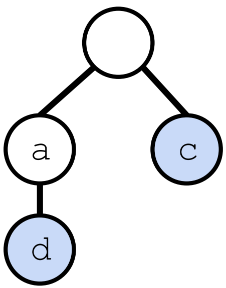
- Tries
  - When does a Trie make sense?
- Implementing a Trie
  - How do we find the next child?
- **Advanced Implementations: Dealing with Sparsity**
  - **Hash Tables, BSTs, Ternary Search Tries**
- Prefix Operations
  - Finding keys with a given prefix

v1.5: DataIndexedCharMap

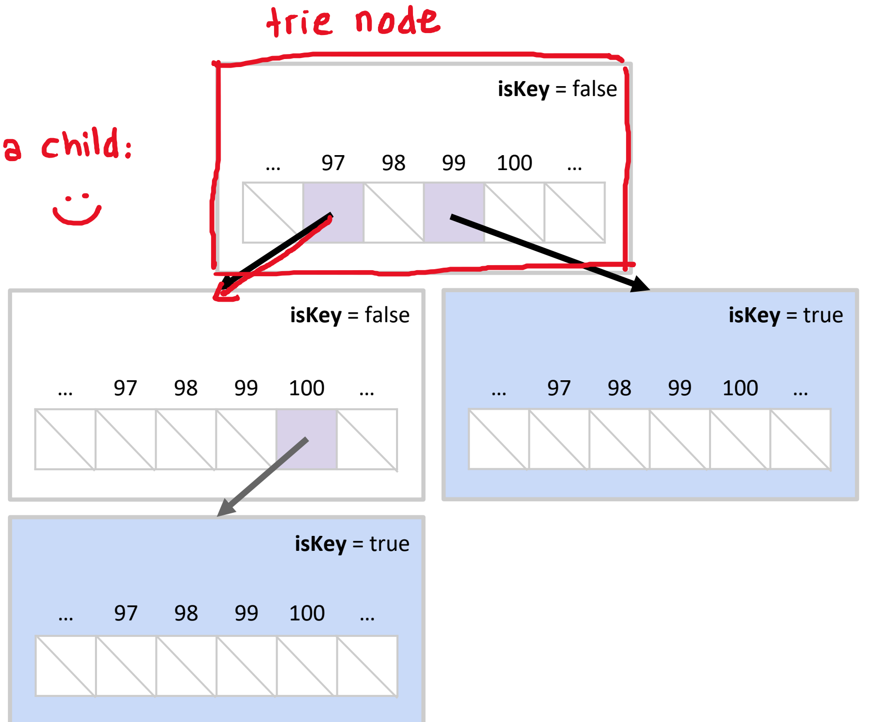
Design 1.5

looking up a child:

$\Theta(1)$:)



Abstract Trie

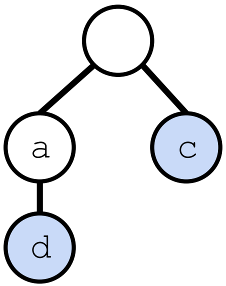


v2.0: Hash-Table-Based Trie

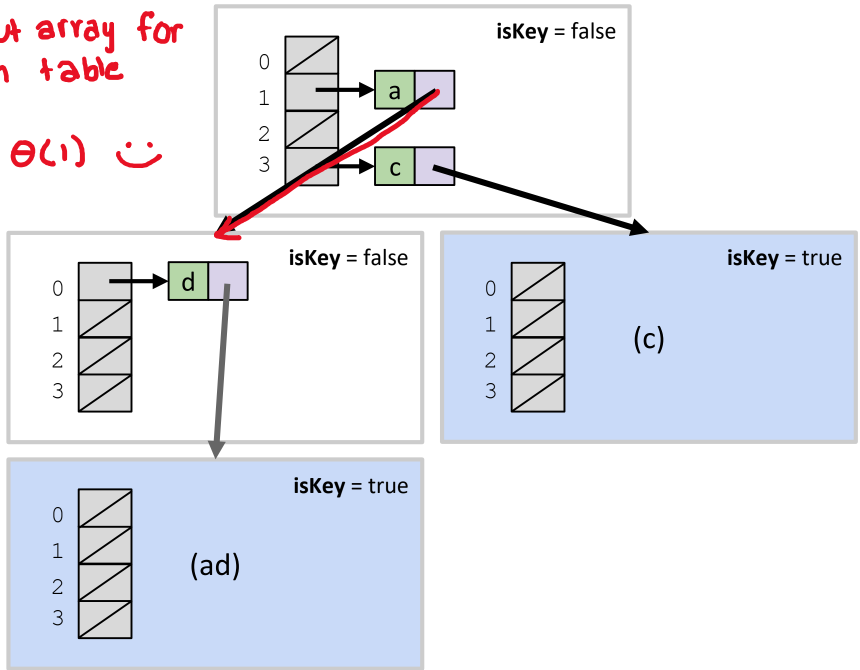
Design 2

swap out array for a hash table

still Θ(1) ◡̈



Abstract Trie

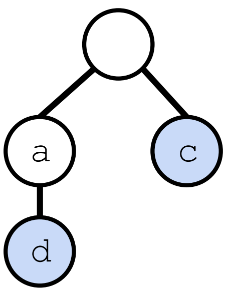


v3.0: BST-Based Trie

Design 3

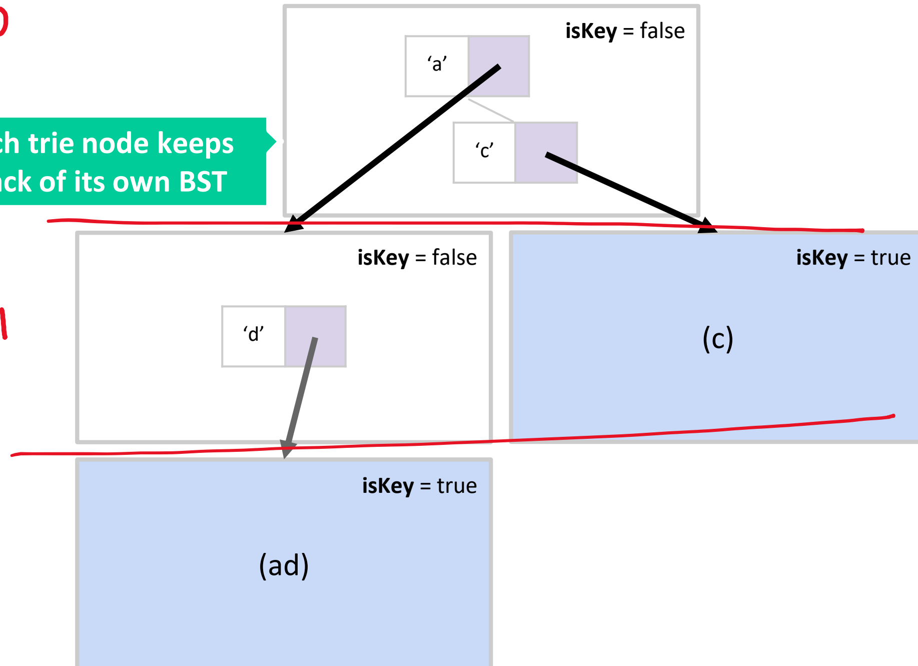
index 0

Each trie node keeps track of its own BST



Abstract Trie

index 1

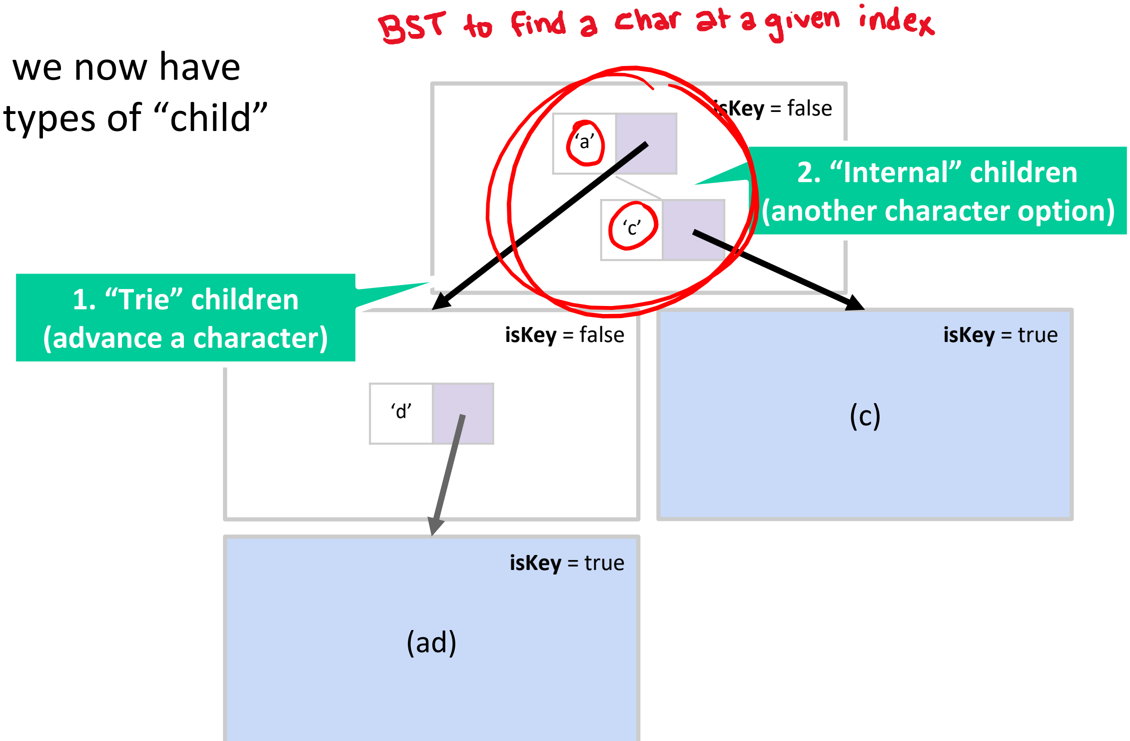


...

v3.0: BST-Based Trie

Design 3

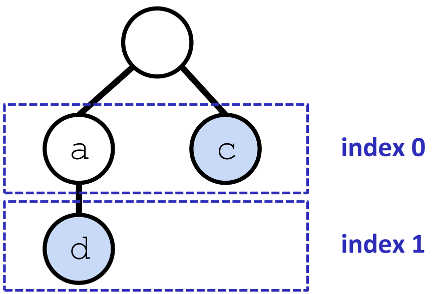
In this design, we now have two different types of “child” nodes:



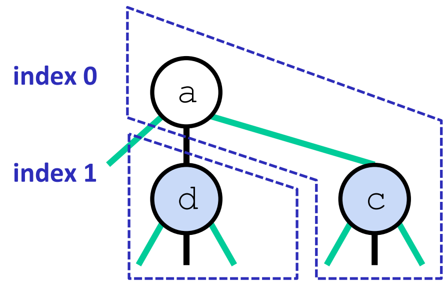
But both are essentially child references – could we simplify this design?

v4.0: Ternary Search Trie (TST)

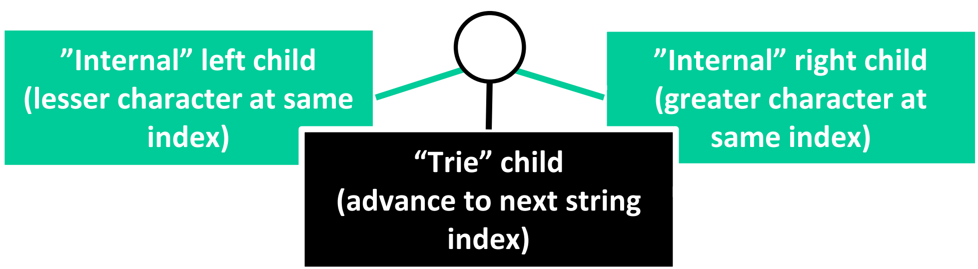
Design 4



Abstract Trie



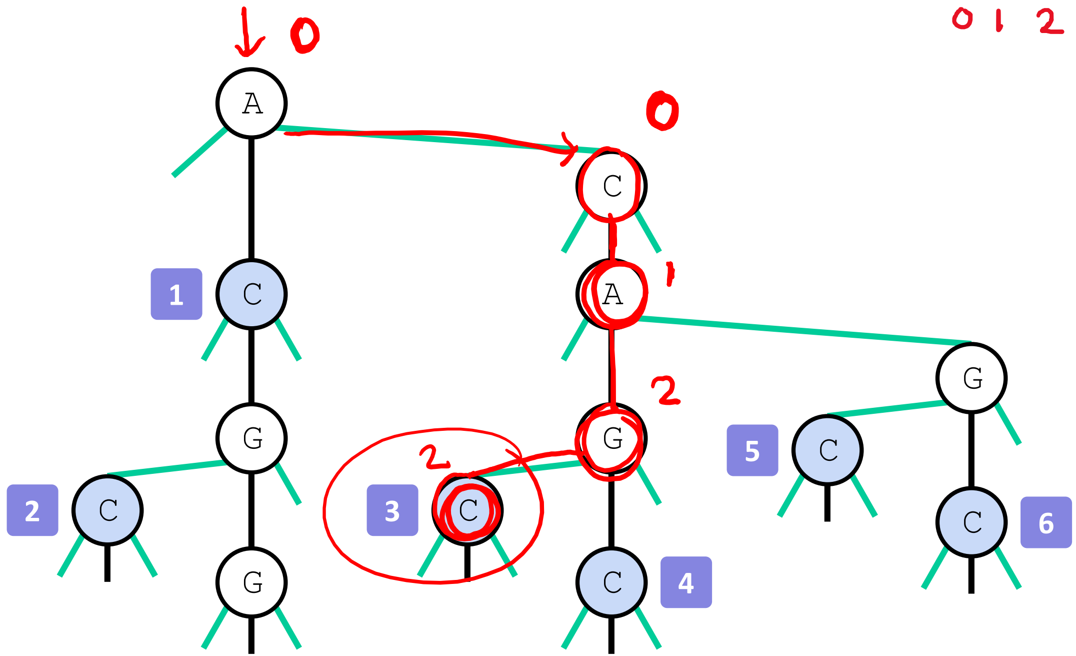
Ternary Search Trie



Poll Everywhere

pollev.com/uwcse373

Which node is associated with the key "CAC"?



Tries in COS 226 (Sedgewick, Wayne/Princeton)

Searching in a TST

Design 4

Follow links corresponding to each character in the key.

- If less, take left link; if greater, take right link.
- If equal, take the middle link and move to the next key character.

Search hit. Final node is blue (`isKey == true`).

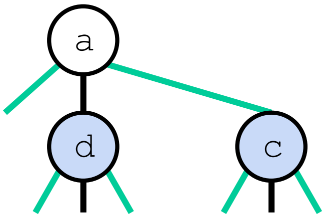
Search miss. Reach a null link or final node is white (`isKey == false`).

Poll Everywhere

pollev.com/uwcse373

Does the structure of a TST depend on the order in which strings are inserted?

A. Yes
B. No
C. We're not sure



Lecture Outline

- Tries
  - When does a Trie make sense?
- Implementing a Trie
  - How do we find the next child?
- Advanced Implementations: Dealing with Sparsity
  - Hash Tables, BSTs, Ternary Search Tries
- **Prefix Operations**
  - **Finding keys with a given prefix**

String-Specific Operations

Abstract Trie

Theoretical asymptotic speed improvement is nice.

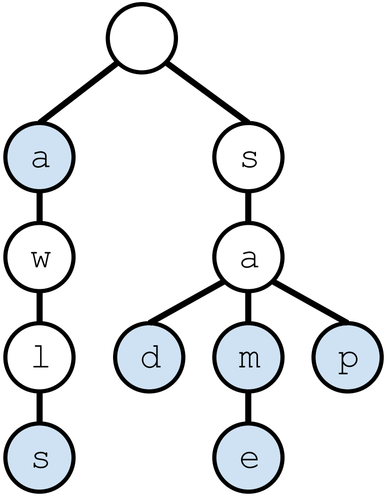
But the main appeal of tries is their efficient prefix matching!

Prefix match.

```
keysWithPrefix("sa")
```

Longest prefix.

```
longestPrefixOf("sample")
```



In this section, we'll use the abstract trie representation.

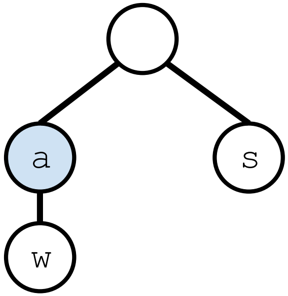
Collecting Trie Keys

Abstract Trie

Describe in English an algorithm to collect all the keys in a trie.

`collect():`
`["a","awls","sad","sam","same","sap"]`

1. Create an empty list of results `x`.
2. For character `c` in `root.next.keys()`:
   Call `colHelp(c, x, root.next.get(c))`.
3. Return `x`.



`colHelp(String s, List<String> x, Node n)`

1. ???

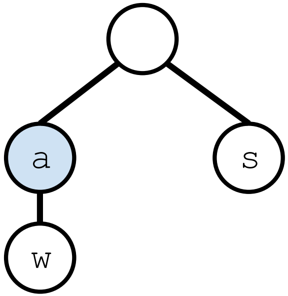
Collecting Trie Keys

Abstract Trie

Describe in English an algorithm to collect all the keys in a trie.

`collect():`
`["a","awls","sad","sam","same","sap"]`

1. Create an empty list of results `x`.
2. For character `c` in `root.next.keys()`:
   Call `colHelp(c, x, root.next.get(c))`.
3. Return `x`.



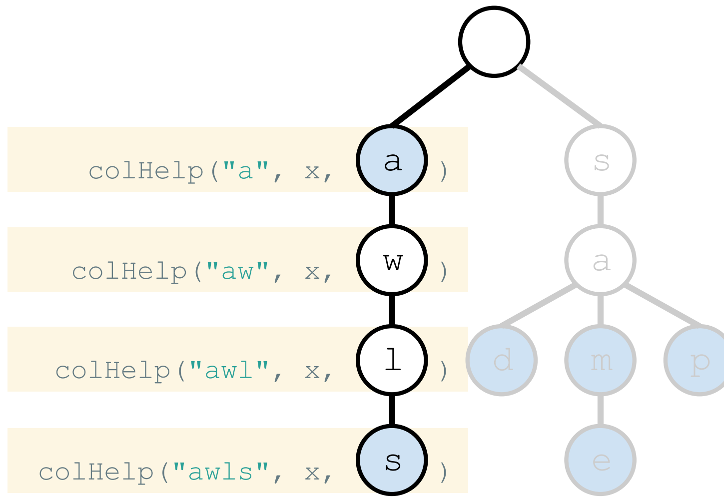
`colHelp(String s, List<String> x, Node n)`

1. If `n.isKey`, then `x.add(s)`.
2. For character `c` in `n.next.keys()`:
   Call `colHelp(s + c, x, n.next.get(c))`.
   "accumulate" the current string

Collecting Trie Keys

Abstract Trie

```
collect(): [
    "a",
    "awls",
]
```



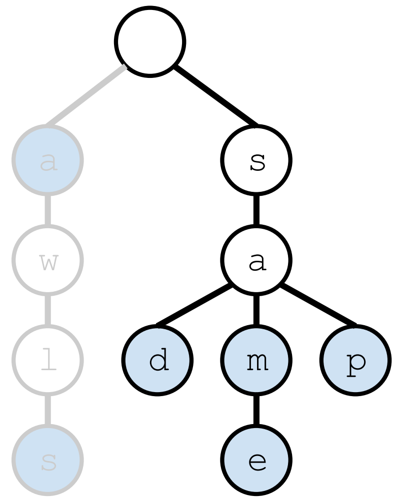
`colHelp(String s, List<String> x, Node n)`

1. If `n.isKey`, then `x.add(s)`.
2. For character `c` in `n.next.keys()`:
   Call `colHelp(s + c, x, n.next.get(c))`.

Collecting Trie Keys

Abstract Trie

```
collect(): [
    "a",
    "awls",
    "sad",
    "sam",
    "same",
    "sap"
]
```



`colHelp(String s, List<String> x, Node n)`

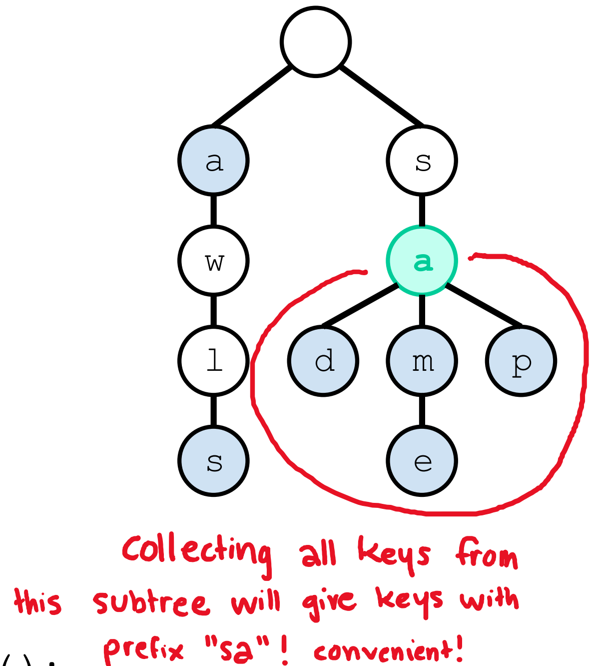
1. If `n.isKey`, then `x.add(s)`.
2. For character `c` in `n.next.keys()`:
   Call `colHelp(s + c, x, n.next.get(c))`.

Prefix Operations with Tries

Abstract Trie

Describe in English an algorithm for `keysWithPrefix`.

```
keysWithPrefix("sa"):
["sad","sam","same","sap"]
```



1. Find the node α corresponding to the string (in green).
2. Create an empty list `x`.
3. For character `c` in `α.next.keys()`:
   Call `colHelp("sa" + c, x, α.next.get(c))`.

tl;dr

- Tries can be used for storing strings (sequential data)
- Real-world performance is often better than a hash table or search tree
- Many different implementations
  - Could store DataIndexedCharMaps/Hash Tables/BSTs within nodes, or combine overall structure to get a TST
- Tries enable very efficient prefix operations like `keysWithPrefix`

Extra: Autocomplete with Tries

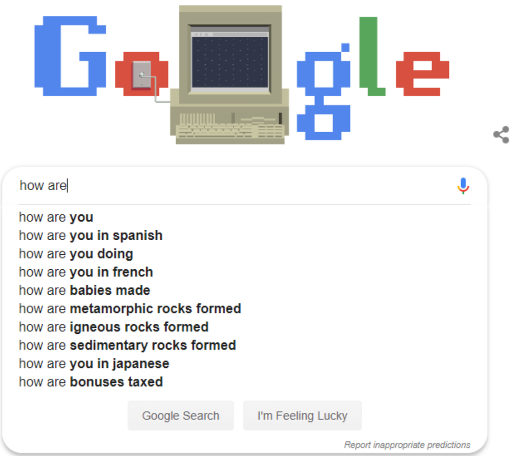
Abstract Trie

Autocomplete should return the **most relevant results**.

One way: a Trie-based Map<String, Relevance>.

When a user types in a string `"hello"`,

1. Call `keysWithPrefix("hello")`.
2. Return the 10 strings with the highest relevance.

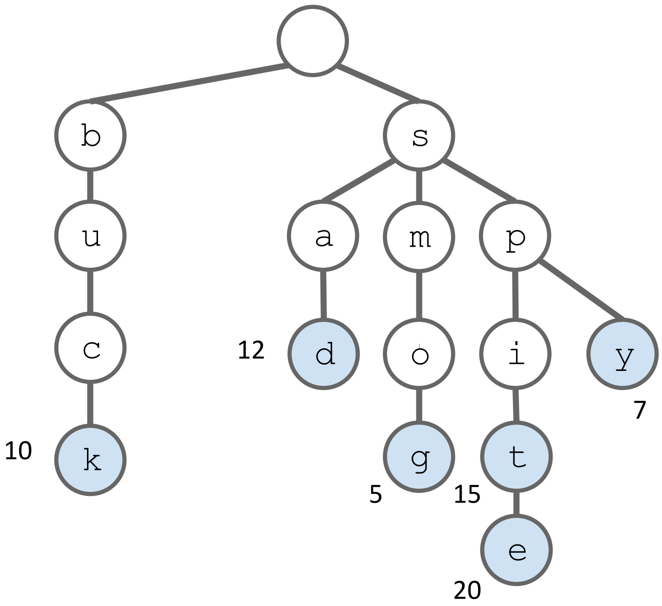


Extra: Autocomplete with Tries

Abstract Trie

One approach to find top 3 matches with prefix "s":

1. Call `keysWithPrefix("s")`.
   sad, smog, spit, spite, spy
2. Return the 3 keys with highest value.
   spit, spite, sad



This algorithm is slow. Why?

Extra: Autocomplete with Tries

Improving Autocomplete

Very short queries, e.g. "s", will require checking billions of results.

But we only need to keep the top 10.

Prune the search space. Each node stores its own relevance as well as the max relevance of its descendents.