LEC 23

CSE 373

Tries

BEFORE WE START

Which of the following is true about Topological Sort and Reductions?

a) Any given graph can be topologically sorted
b) The standard BFS algorithm can be used to topologically sort a graph
c) A reduction is a problem-solving strategy of reducing a problem into smaller chunks
d) Seam carving can be reduced to the BFS algorithm
e) None of the above

pollev.com/uwcse373

Instructor: Eric Fan! 🎉

TAs:
Timothy Akintilo
Brian Chan
Joyce Elauria
Eric Fan
Farrell Fileas
Melissa Hovik
Leona Kazi
Keanu Vestil
Siddharth Vaidyanathan
Howard Xiao

Announcements

- EX4 (MSTs & Sorting) due tonight 11:59 PM PDT
  - Late cutoff Thursday, August 20th
- P4 (Mazes) due Wednesday 11:59 PM PDT
  - Late cutoff Saturday, August 22nd (day until eternal mastery of CSE 373)
- All extra credit due Saturday, August 22nd
- EXAM 2 Logistics & Information
  - Released Friday 08/21 12:01 AM PDT, due Saturday 08/22 11:59 PM PDT
  - No submissions accepted after Saturday deadline
  - See Exams page for more detailed logistics and relevant review materials
- Optional Exam II Office Hours during Friday's lecture
  - For clarifying or logistical questions
  - We'll also be actively monitoring Piazza for questions

Announcements

- Please fill out course evaluations!
  - We *do* read your feedback and take it into consideration
  - Aaron and your TAs would be so appreciative!

Learning Objectives

After this lecture, you should be able to...

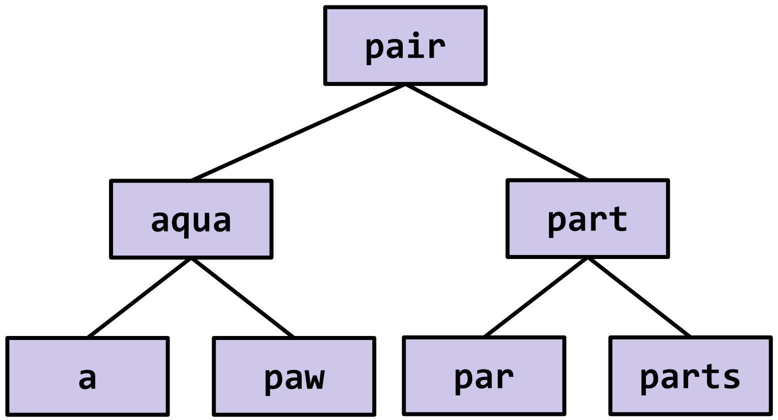
1. Identify when a Trie can and should be used, and describe the useful properties a Trie provides
2. Describe and implement the abstract Trie and argue how they are more efficient than using Hash Tables for storing Strings
3. Compare and contrast more advanced Trie designs and explain their differences in runtime and space complexity
4. Implement Trie prefix algorithms and explain how autocomplete algorithms are designed

Lecture Outline

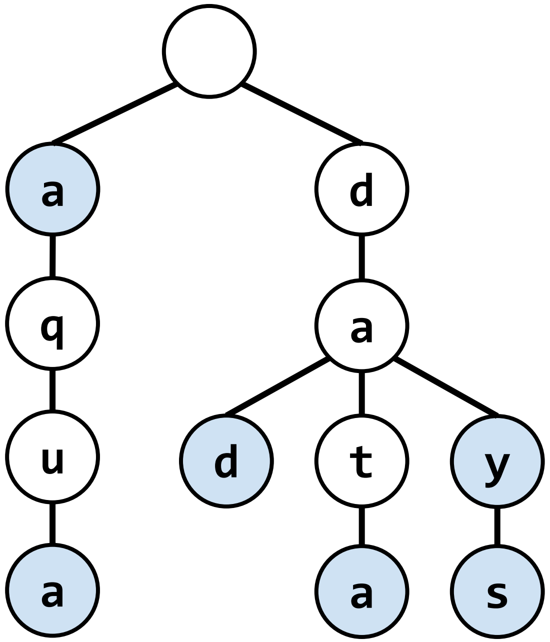
- **Tries Introduction**
  - **When does using a Trie make sense?**
- Implementing a Trie using an array
  - How do we find the next child?
- Advanced Implementations: dealing with sparsity
  - Hash Tables, BSTs, and Ternary Search Trees
- Prefix Operations and Autocomplete
  - Find the keys associated with a given prefix

Tries: A *Specialized* Data Structure

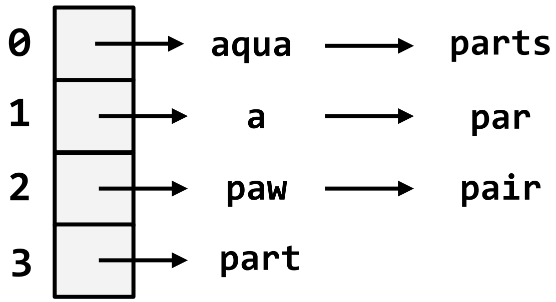
- Tries are a character-by-character set-of-Strings implementation
- Nodes store *parts of keys* instead of *keys*



Binary Search Tree



Trie



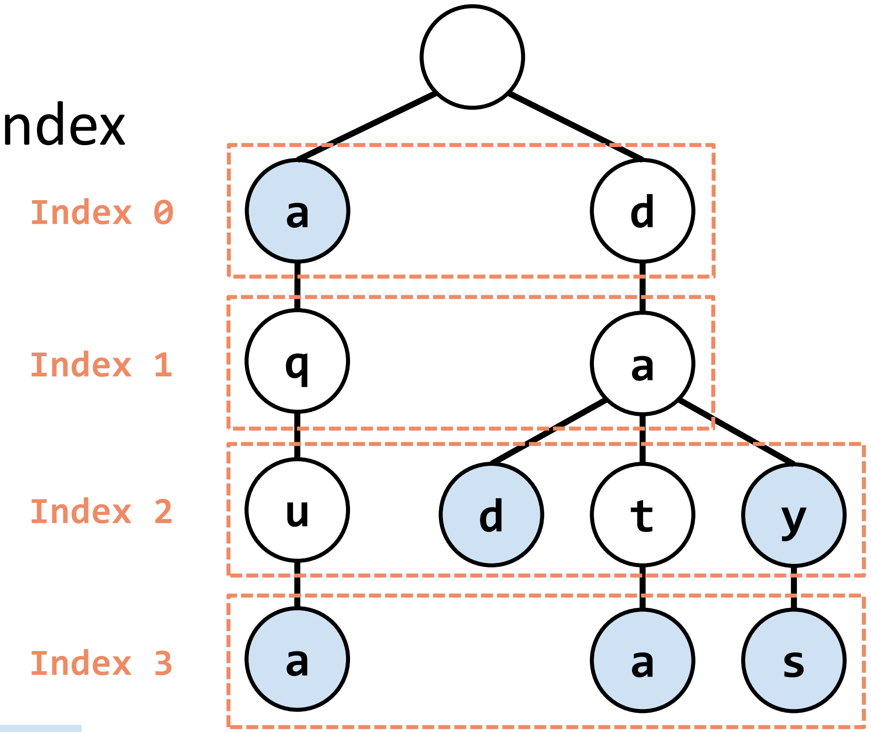
Hash Table

Abstract Trie

- Each level represents an index
  - Children represent next possible characters at that index
- This Trie stores the following set of Strings:

  a, aqua, dad,
  data, day, days

- How do we deal with a and aqua?
  - Mark complete Strings with a boolean (shown in blue)
  - Complete string: a String that belongs in our set



Abstract Trie

Searching in Tries

Search hit: the final node is a key (colored blue)

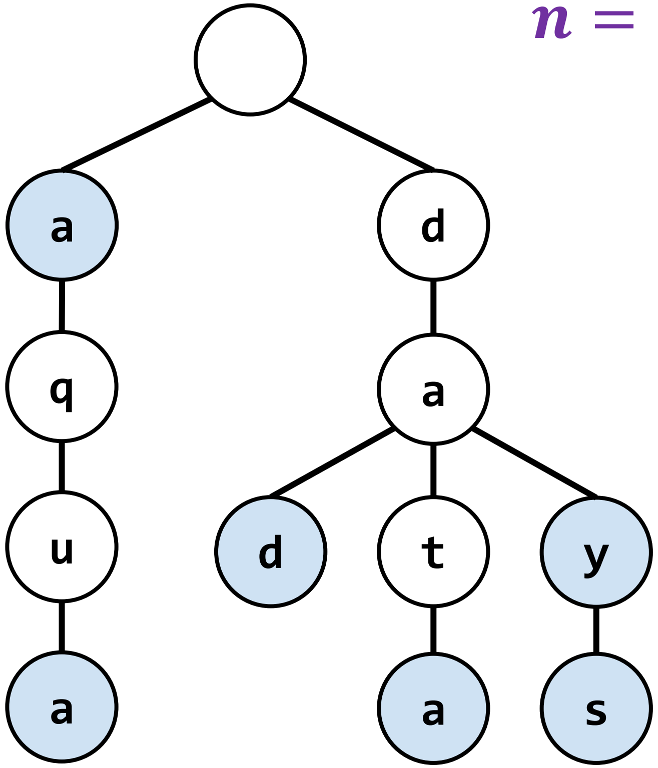
Search miss: caused in one of two ways

1. The final node is not a key (not colored blue)
2. We "fall" off the Trie

```
contains("data")   // hit,  l = 4
contains("da")     // miss, l = 2
contains("a")      // hit,  l = 1
contains("dubs")   // miss, l = 4
```

a, aqua, dad, data, day, days

$n = 6$



Abstract Trie

contains runtime given key of length l with n keys in Trie: $\Theta(l)$

Lecture Outline

- Tries Introduction
  - When does using a Trie make sense?

- **Implementing a Trie using an array**
  - **How do we find the next child?**

- Advanced Implementations: dealing with sparsity
  - Hash Tables, BSTs, and Ternary Search Trees

- Prefix Operations
  - Find keys with a given prefix

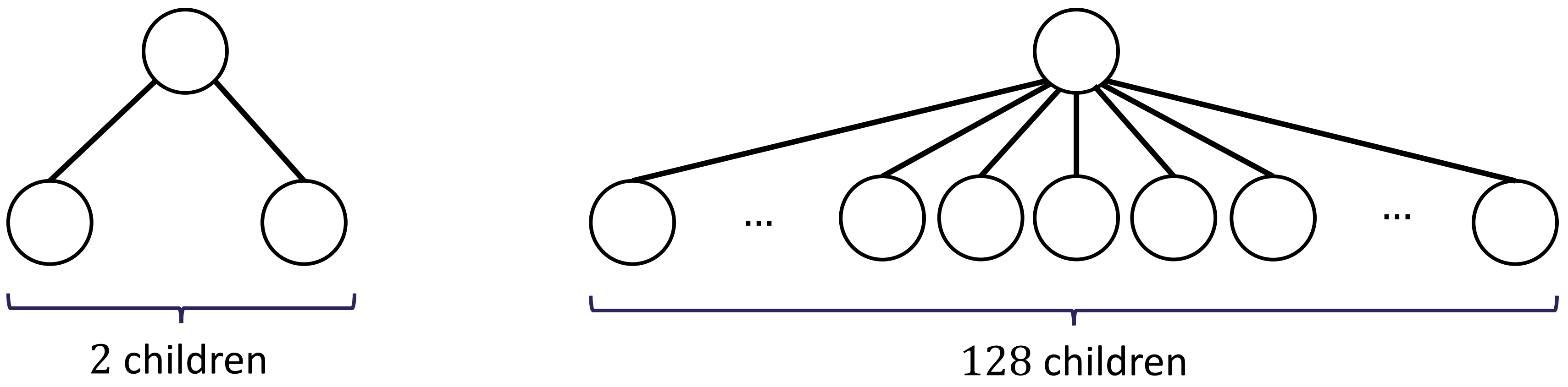
Trie Implementation Idea: *Encoding*

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

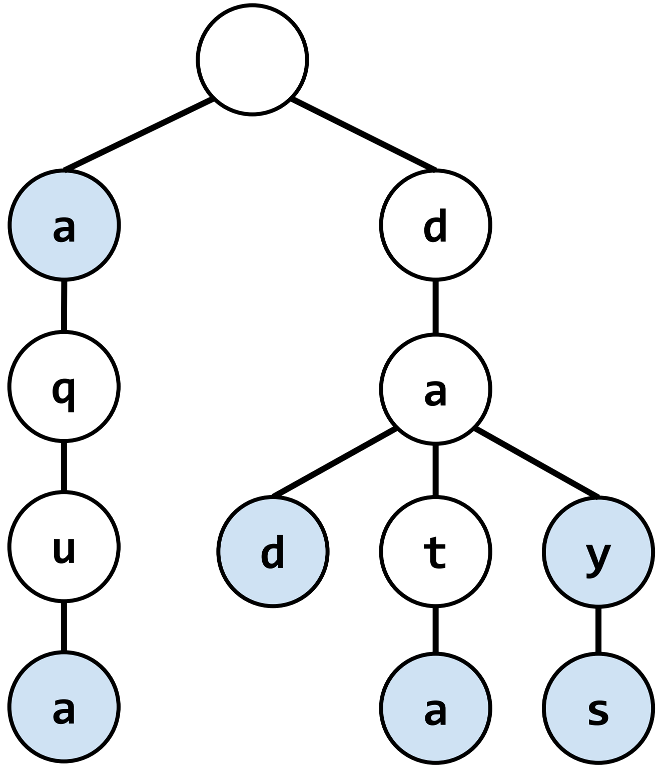
Data Structure for Trie Implementation

- Think of a Binary Tree
  - Instead of two children, we have 128 possible children
  - Each child represents a possible next character of our Trie
- How could we store these 128 children?



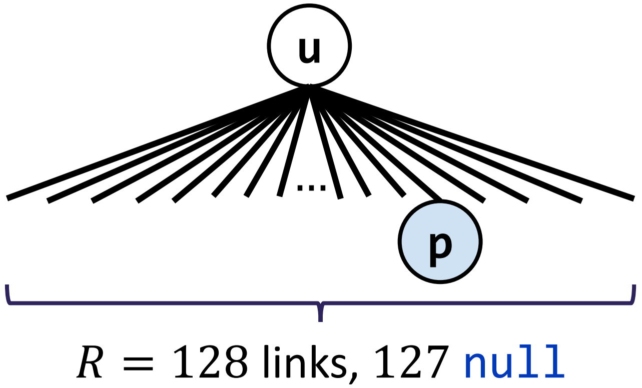
DataIndexedCharMap Pseudocode

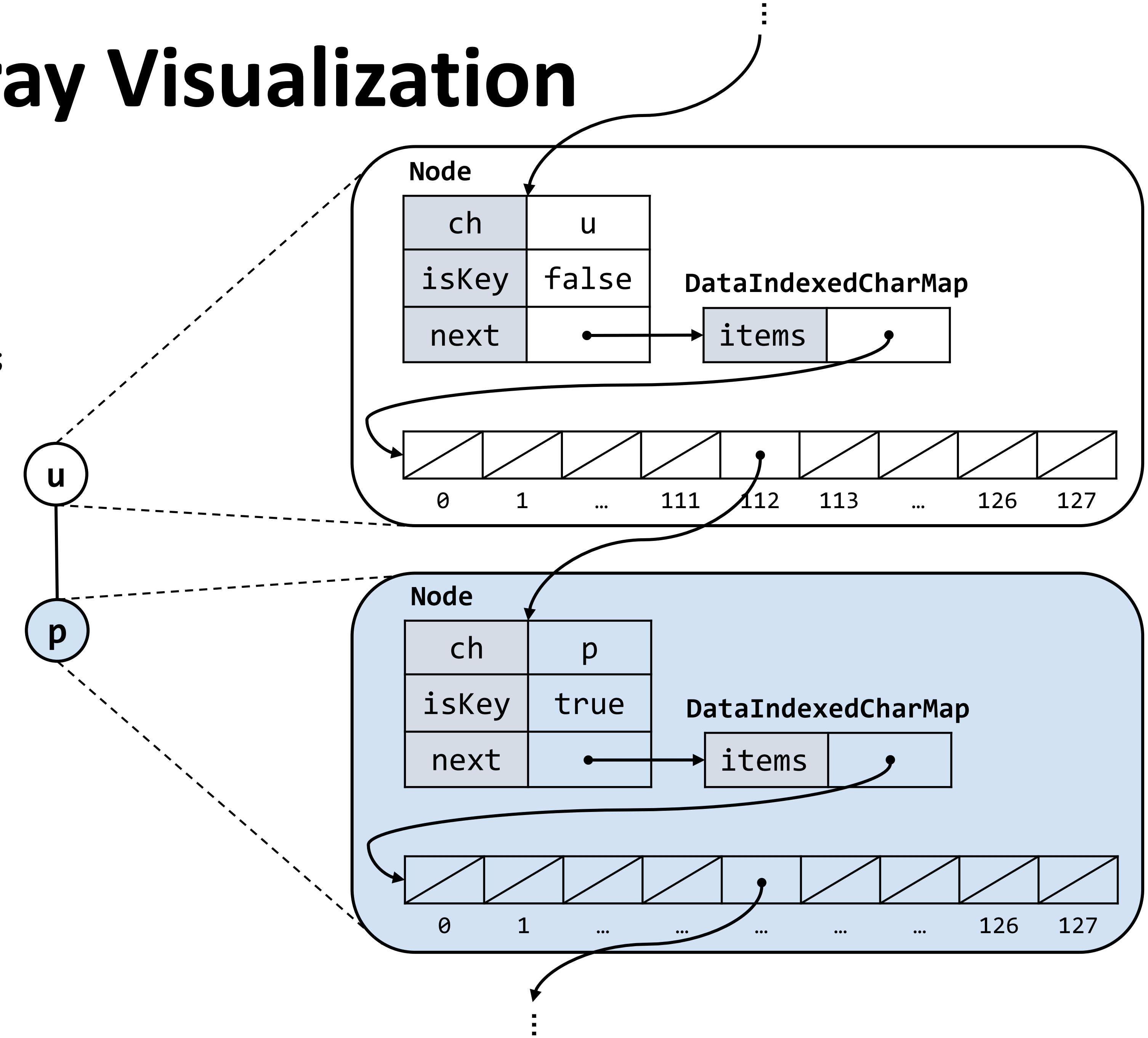
```
class TrieSet {
    final int R = 128;  // # of ASCII encodings
    Node overallRoot;

    // Private internal class
    class Node {
        // Field declarations
        char ch;
        boolean isKey;
        DataIndexedCharMap<Node> next;  // array encoding

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



Data-Indexed Array Visualization

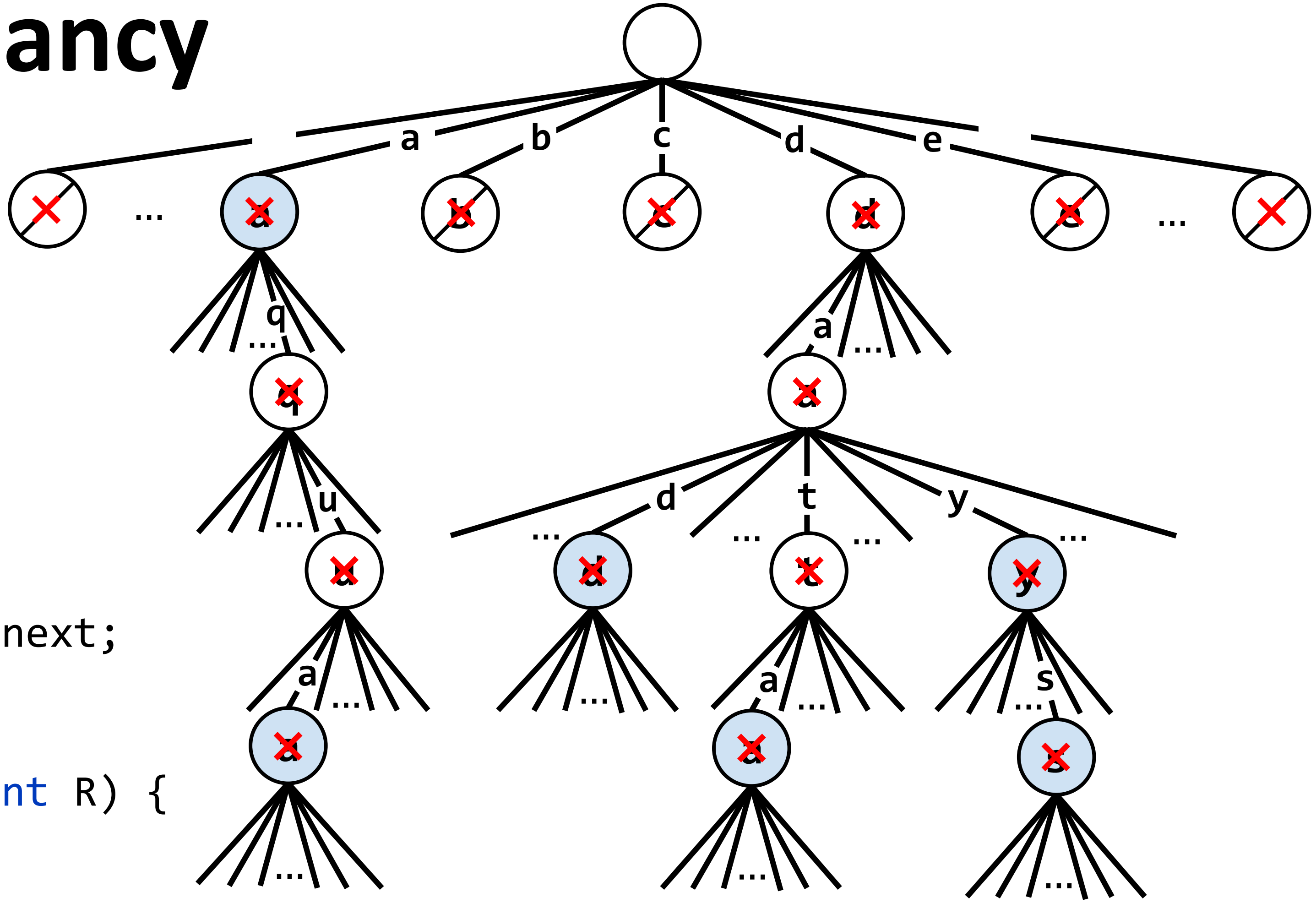
```
// Private internal class
class Node {
    // Field declarations
    char ch;
    boolean isKey;
    DataIndexedCharMap<Node> next;
}
```





Removing Redundancy

```
class TrieSet {
    final int R = 128;
    Node overallRoot;

    // Private internal class
    class Node {
        // Field declarations
        char ch;
        boolean isKey;
        DataIndexedCharMap<Node> next;

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



Poll Everywhere

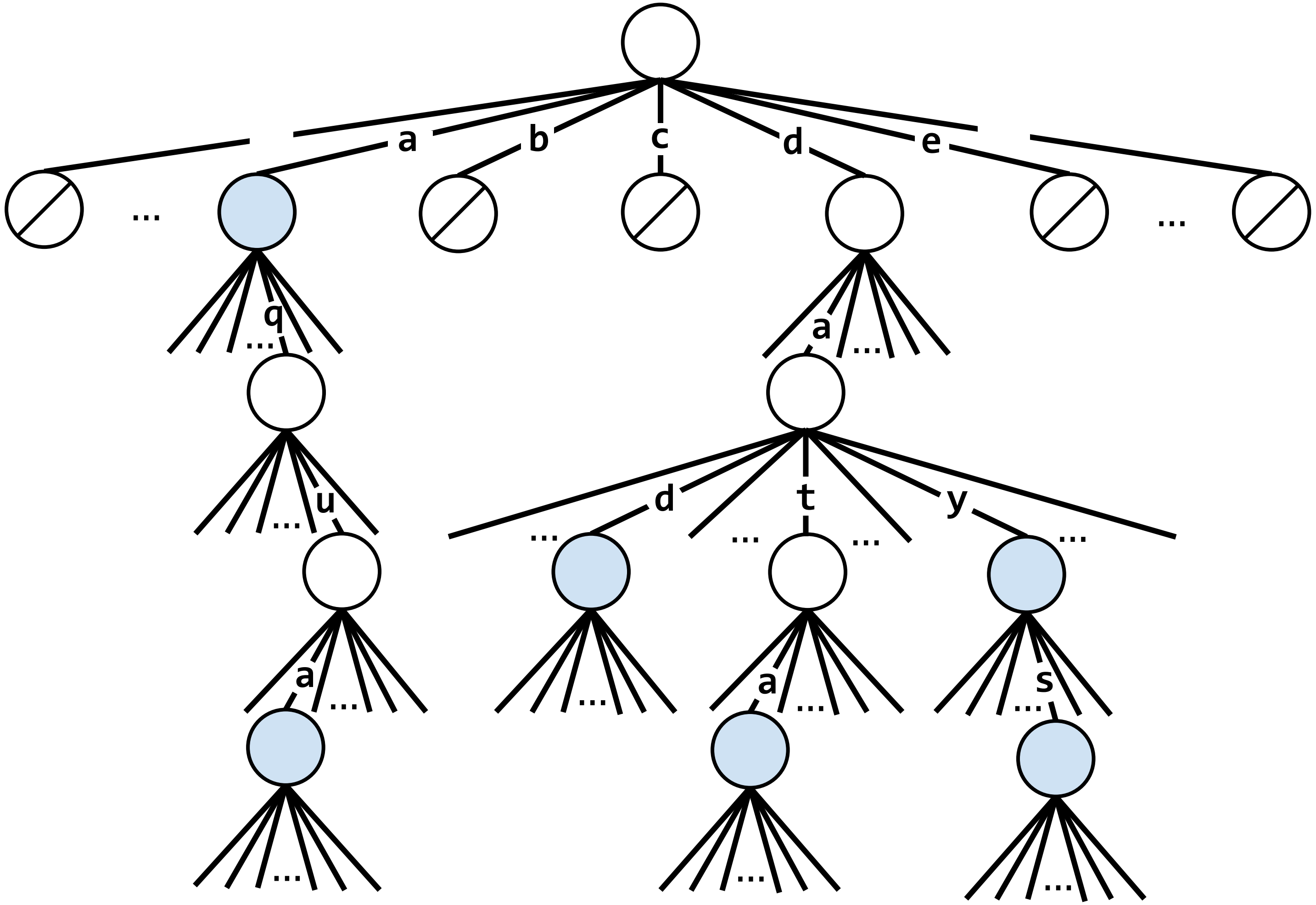
pollev.com/uwcse373

Does the structure of a Trie depend on the order of insertion?

a) Yes

b) No

c) I'm not sure…

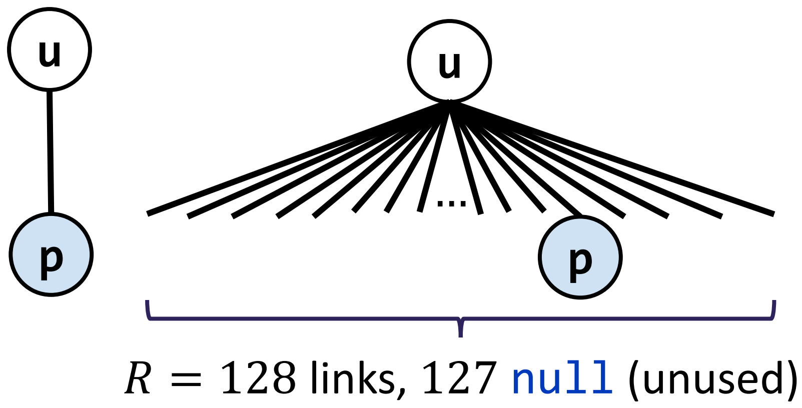


Runtime Comparison

- Typical runtime when treating length l of keys as a constant:

Data Structure	Key Type	contains	add
Balanced BST	Comparable	$\Theta(\log(n))$	$\Theta(\log(n))$
Hash Map	Hashable	$\Theta(1)$*	$\Theta(1)$*
Trie (Data-Indexed Array)	String (Character)	$\Theta(1)$	$\Theta(1)$

* In-practice runtime

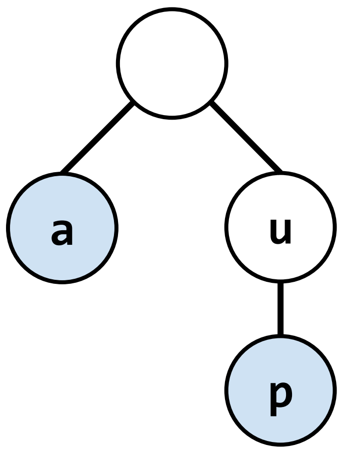


- Takeaways:
  - \+ When keys are Strings, Tries give us a better add and contains runtime
  - – DataIndexedCharMap takes up a lot of space by storing R links per node

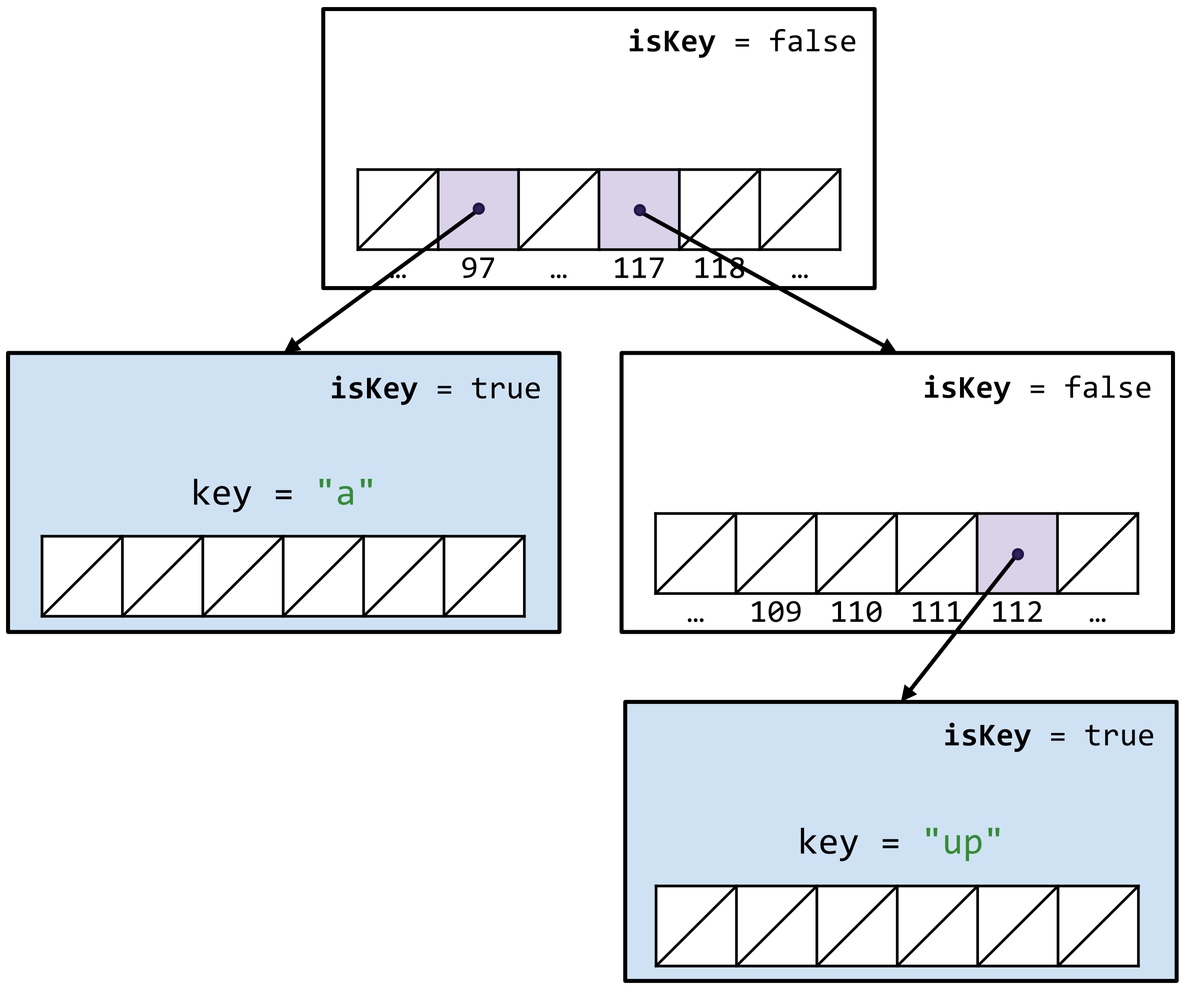
Lecture Outline

- Tries Introduction
  - When does using a Trie make sense?
- Implementing a Trie using an array
  - How do we find the next child?
- **Advanced Implementations: dealing with sparsity**
  - **Hash Tables, BSTs, and Ternary Search Trees**
- Prefix Operations
  - Find keys with a given prefix

`DataIndexedCharMap` Implementation



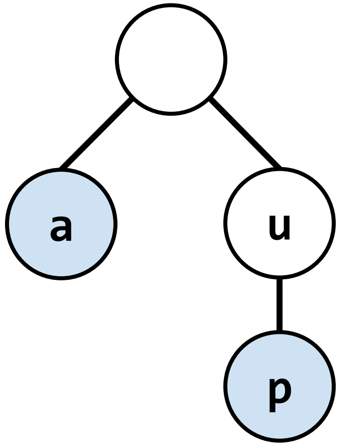
Abstract Trie



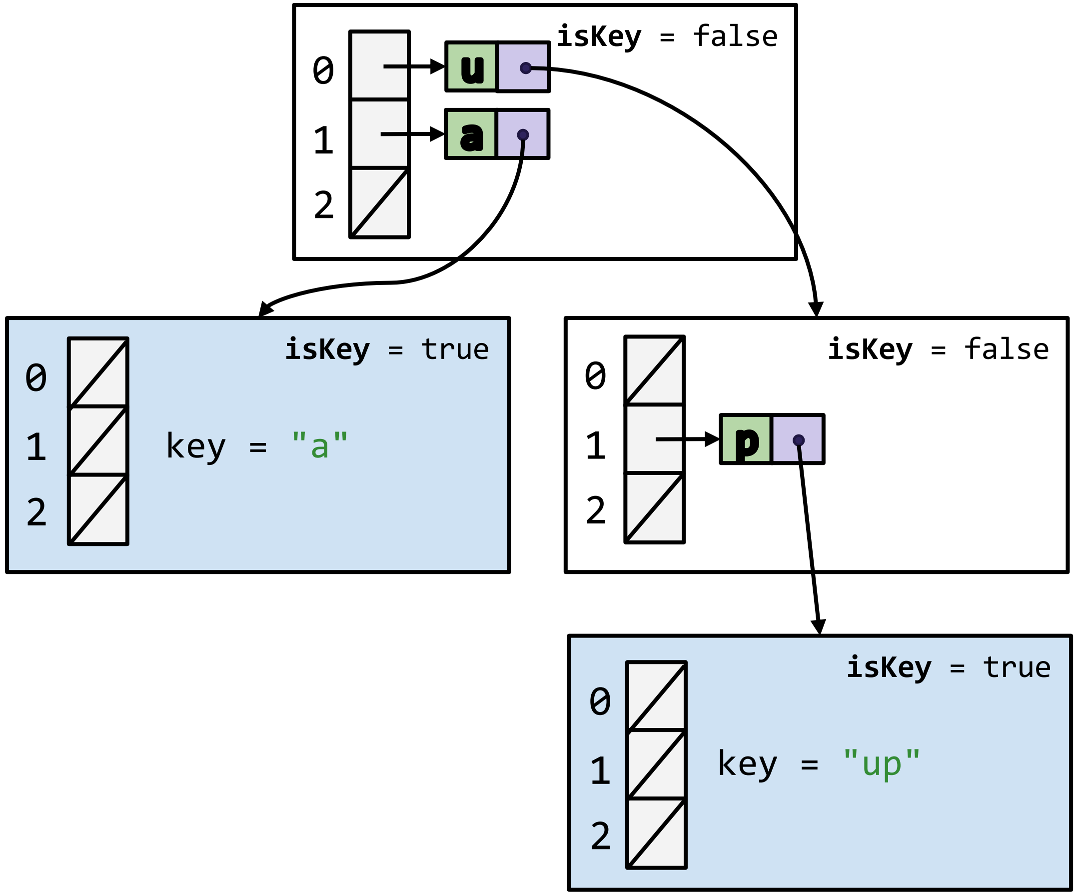
Data-Indexed Array Trie

Hash Table-based Implementation

- Use Hash Table to find character at a given index



Abstract Trie

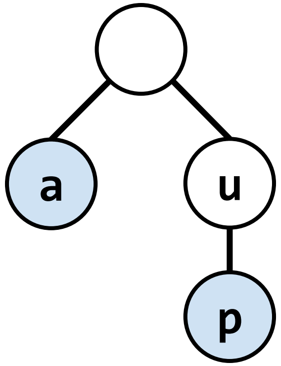


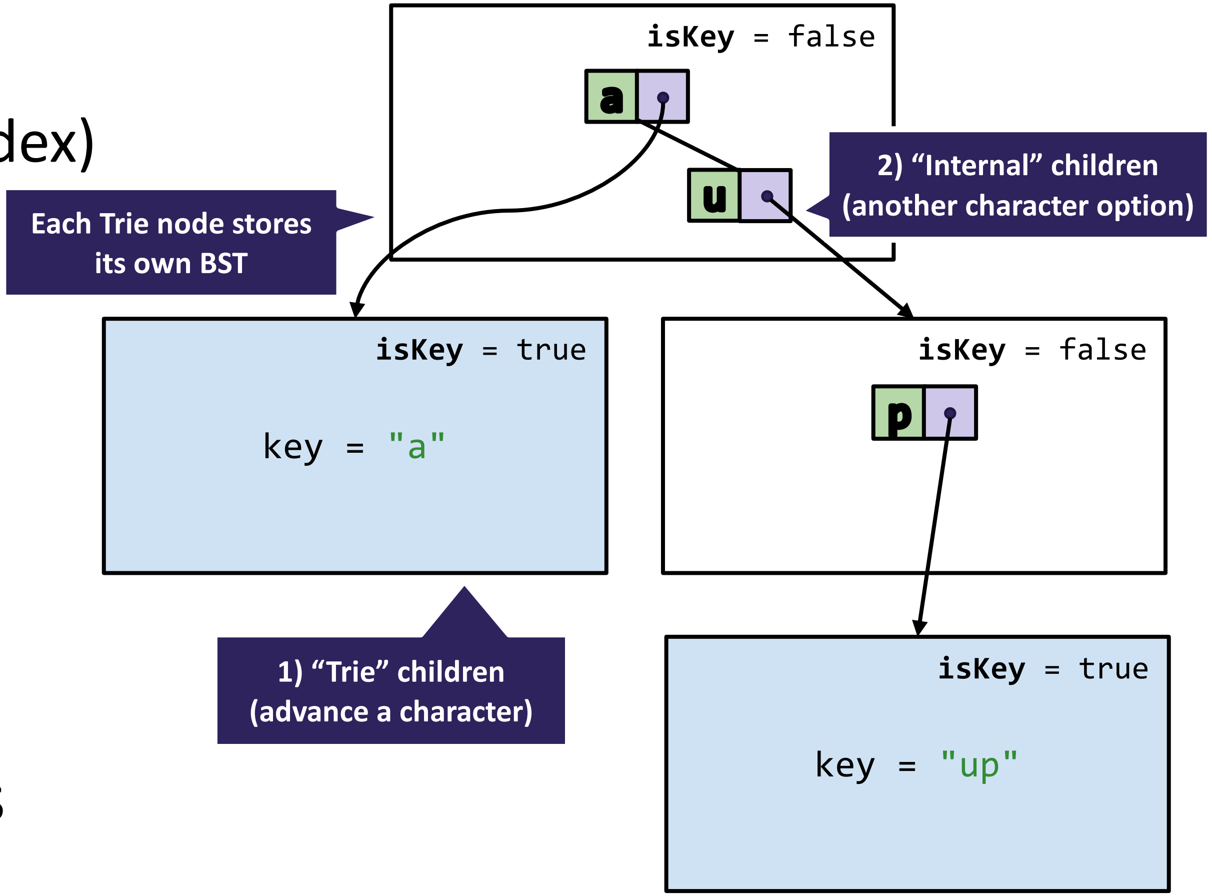
Hash Table-based Trie

BST-based Implementation

- Use Binary Search Tree to find character at a given index
- Two types of children:
    1) "Trie" child: advance a character (index)
    2) "Internal" child: another character option at current character (index)

Abstract Trie



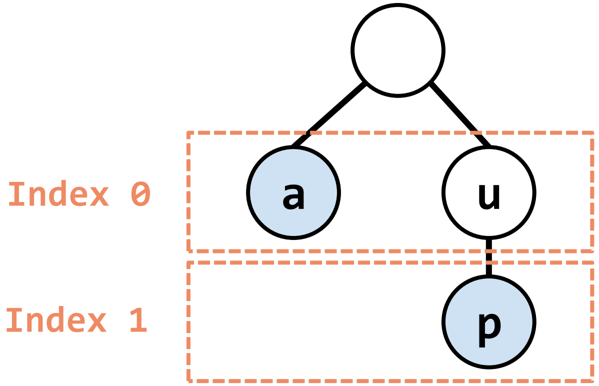


BST-based Trie

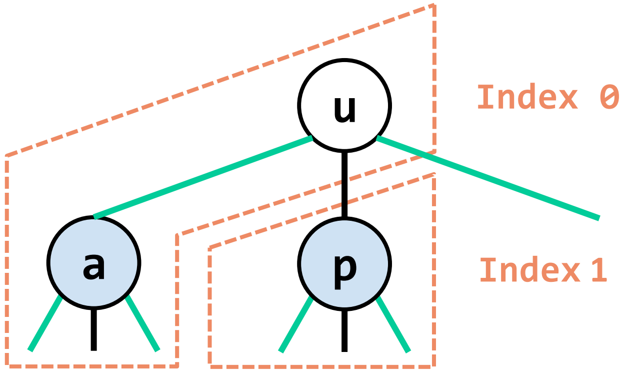
- Both are essentially child references
    - Could we simplify this design?

Ternary Search Tree (TST) Implementation

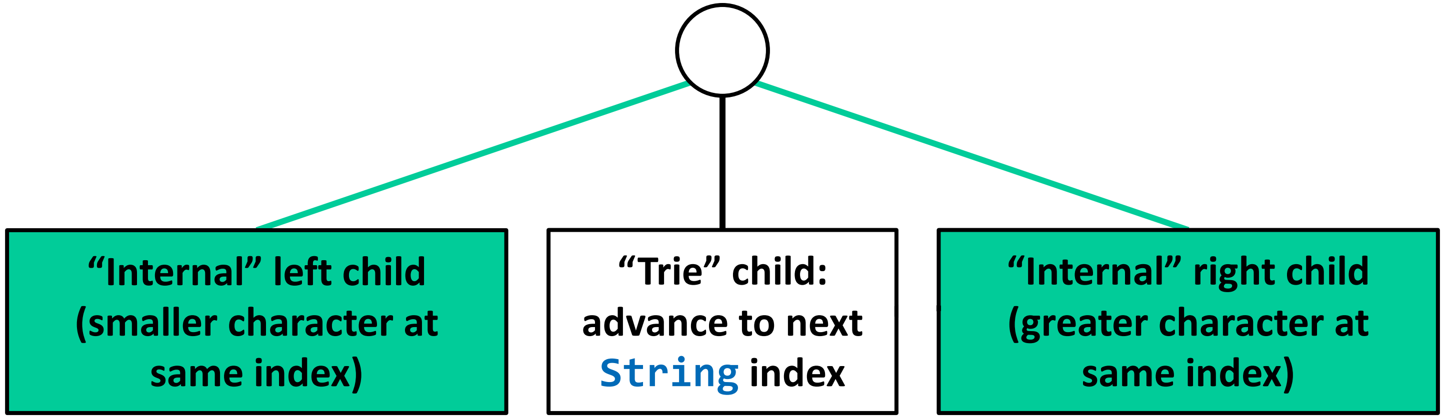
- Combines character mapping with Trie itself



Abstract Trie



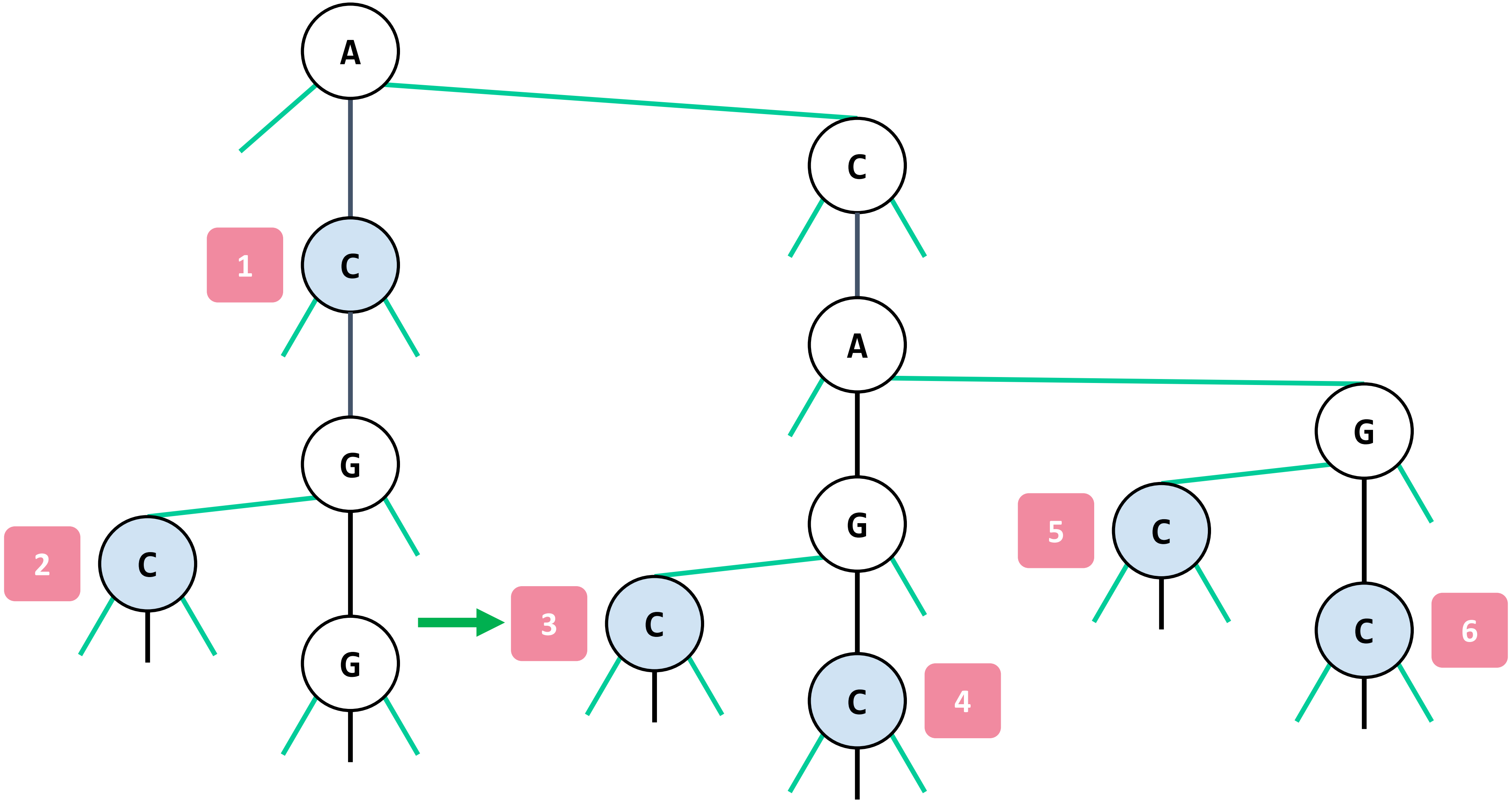
Ternary Search Tree (TST)



Poll Everywhere

pollev.com/uwcse373

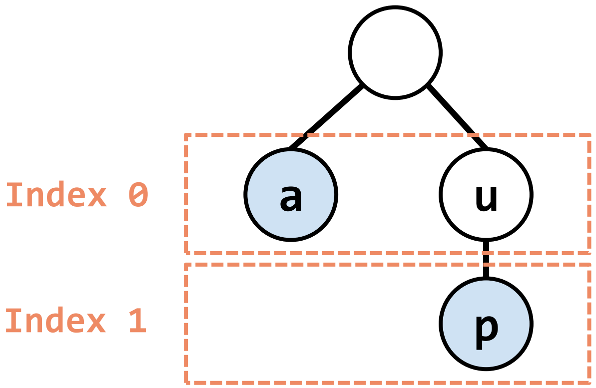
Which node is associated with the key "CAC"?



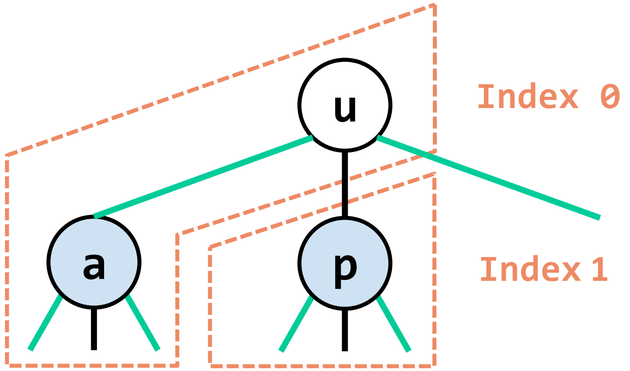
Searching in a TST

- Searching in a TST
  - If smaller, take left link
  - If greater, take right link
  - If equal, take the middle link and move to next character
- **Search hit:** final node yields a key that belongs in our set
- **Search miss:** reach `null` link or final node is yields a key not in our set

Keys: [a , u p]
Index: 0 0 1



Abstract Trie



Ternary Search Tree (TST)

Lecture Outline

- Tries Introduction
  - When does using a Trie make sense?
- Implementing a Trie using an array
  - How do we find the next child?
- Advanced Implementations: dealing with sparsity
  - Hash Tables, BSTs, and Ternary Search Trees
- **Prefix Operations**
  - **Find keys with a given prefix**

Prefix Operations with Tries

a, aqua, dad, data, day, days

- The main appeal of Tries is its efficient prefix matching!
- **Prefix:** find set of keys associated with given prefix

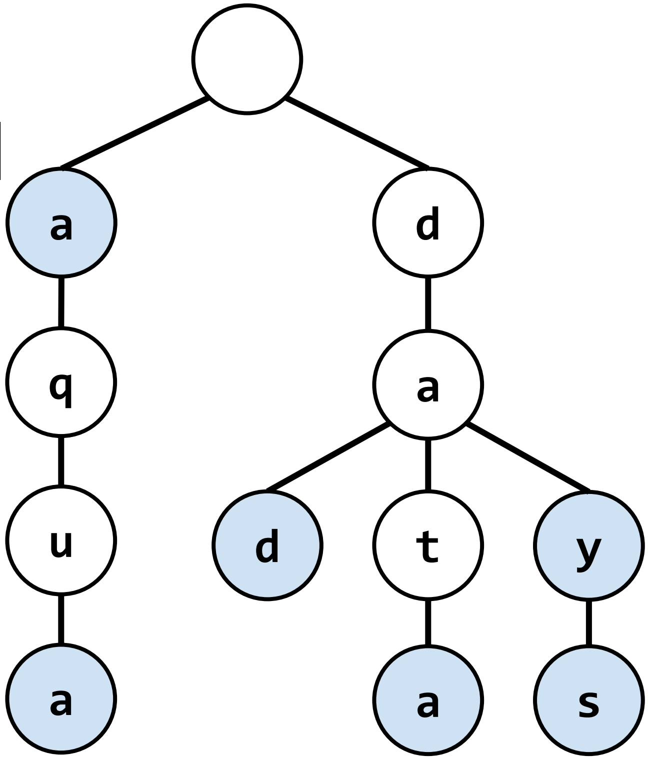
  `keysWithPrefix("day")` returns `["day", "days"]`

- **Longest Prefix From Trie:** given a String, retrieve longest prefix of that String that exists in the Trie

  `longestPrefixOf("aquarium")` returns `"aqua"`

  `longestPrefixOf("aqueous")` returns `"aqu"`

  `longestPrefixOf("dawgs")` returns `"da"`



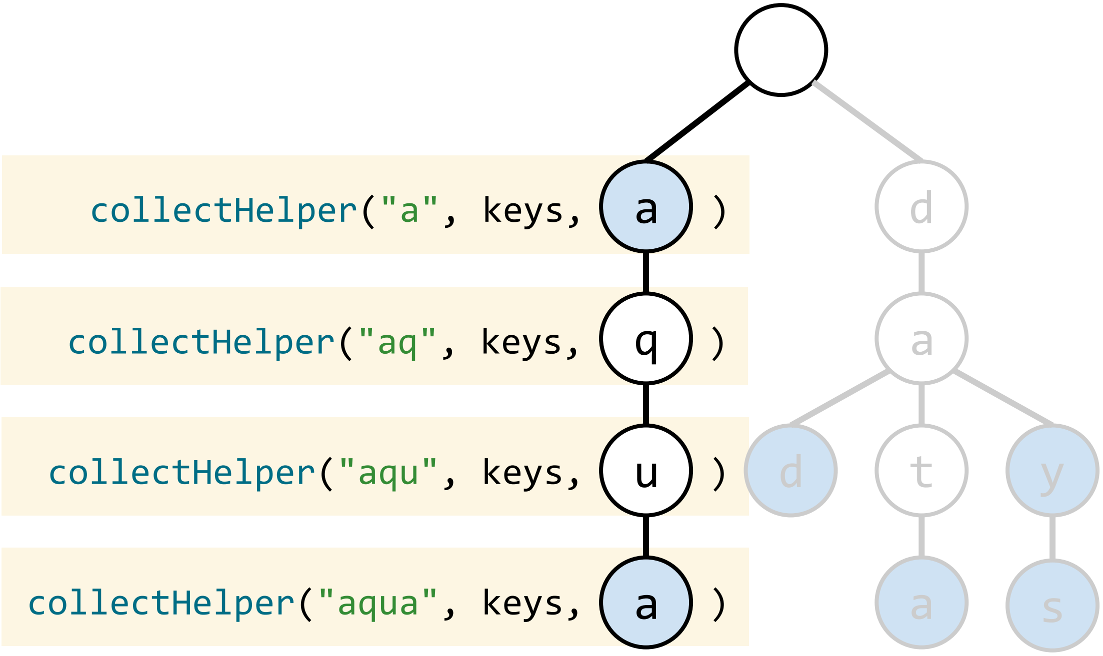
Abstract Trie

Collecting Trie Keys

- **Collect:** return set of all keys in the Trie (like `keySet()`)

  `collect(trie) = ["a", "aqua", "dad", "data", "day", "days"]`

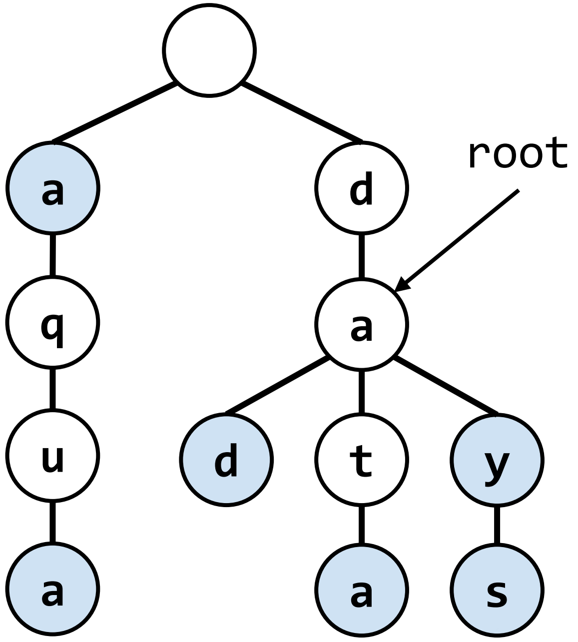
```
List collect() {
    List keys;
    for (char ch : root.next.keys()) {
        collectHelper(ch, keys, root.next.get(ch));
    }
    return keys;
}

void collectHelper(String str, List keys, Node n) {
    if (n.isKey()) {
        keys.add(s);
    }
    for (char ch : n.next.keys()) {
        collectHelper(str + ch, keys, n.next.get(ch));
    }
}
```



keysWithPrefix Implementation

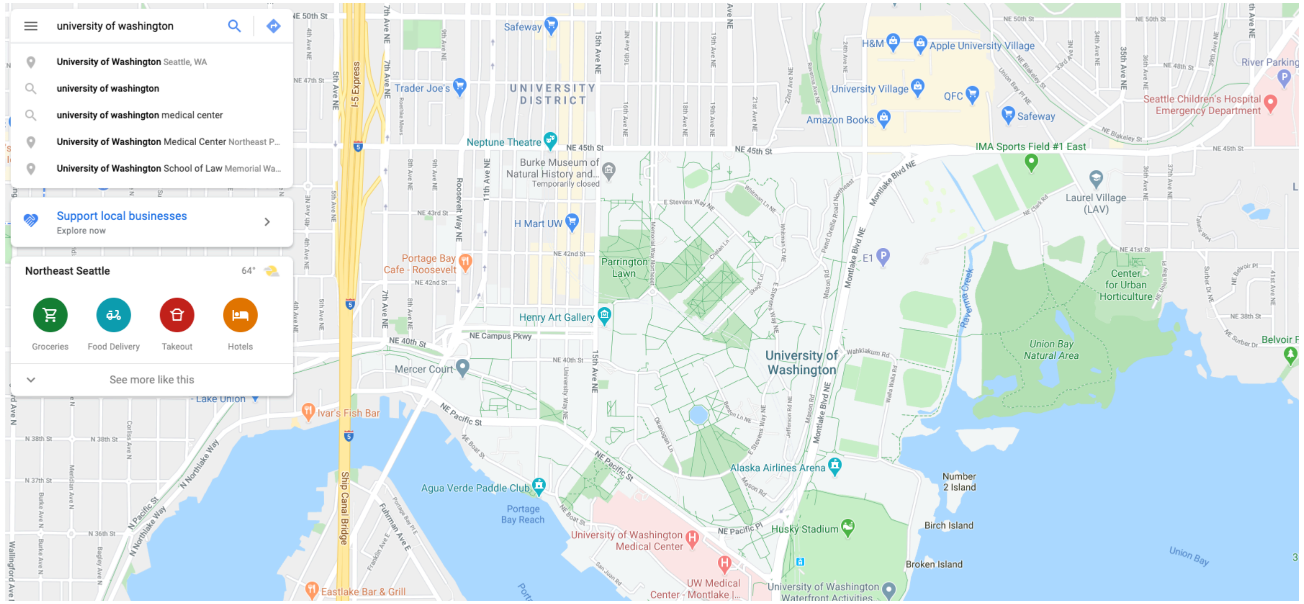
- keysWithPrefix(String prefix)
  - Find all the keys that corresponds to the given prefix

```
List keysWithPrefix(String prefix) {
    Node root;  // Node corresponding to given prefix
    List keys;  // Empty list to store keys

    for (char ch : root.next.keySet()) {
        collectHelper(prefix + c, keys, node.next.get(ch));
    }
}

void collectHelper(String str, List keys, Node n) {
    if (n.isKey()) {
        keys.add(s);
    }
    for (char ch : n.next.keys()) {
        collectHelper(str + ch, keys, n.next.get(ch));
    }
}
```

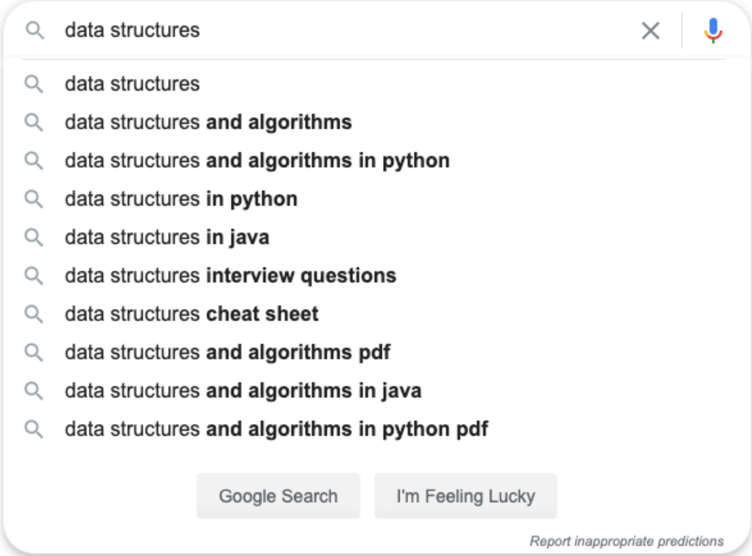


Autocomplete with Tries

- Autocomplete should return the **most relevant results**
- One method: a Trie-based `Map<String, Relevance>`
  - When a user types in a string `"hello"`, call `keysWithPrefix("hello")`
  - Return the 10 Strings with the highest relevance



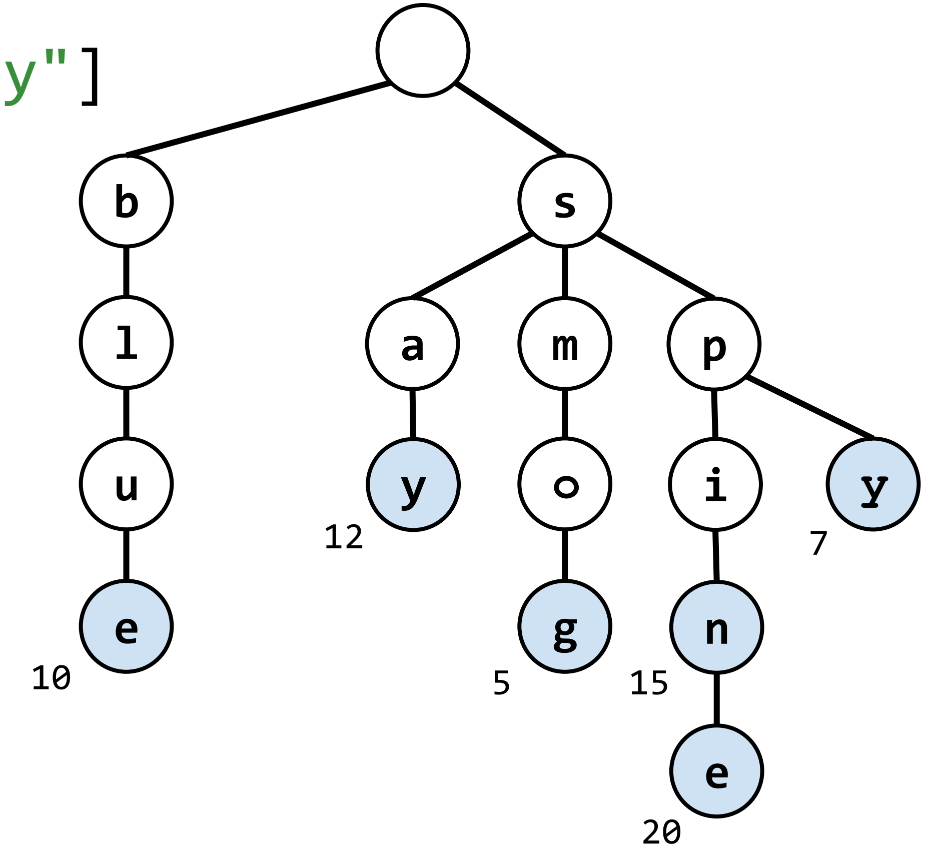
Google



Autocomplete with Tries

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

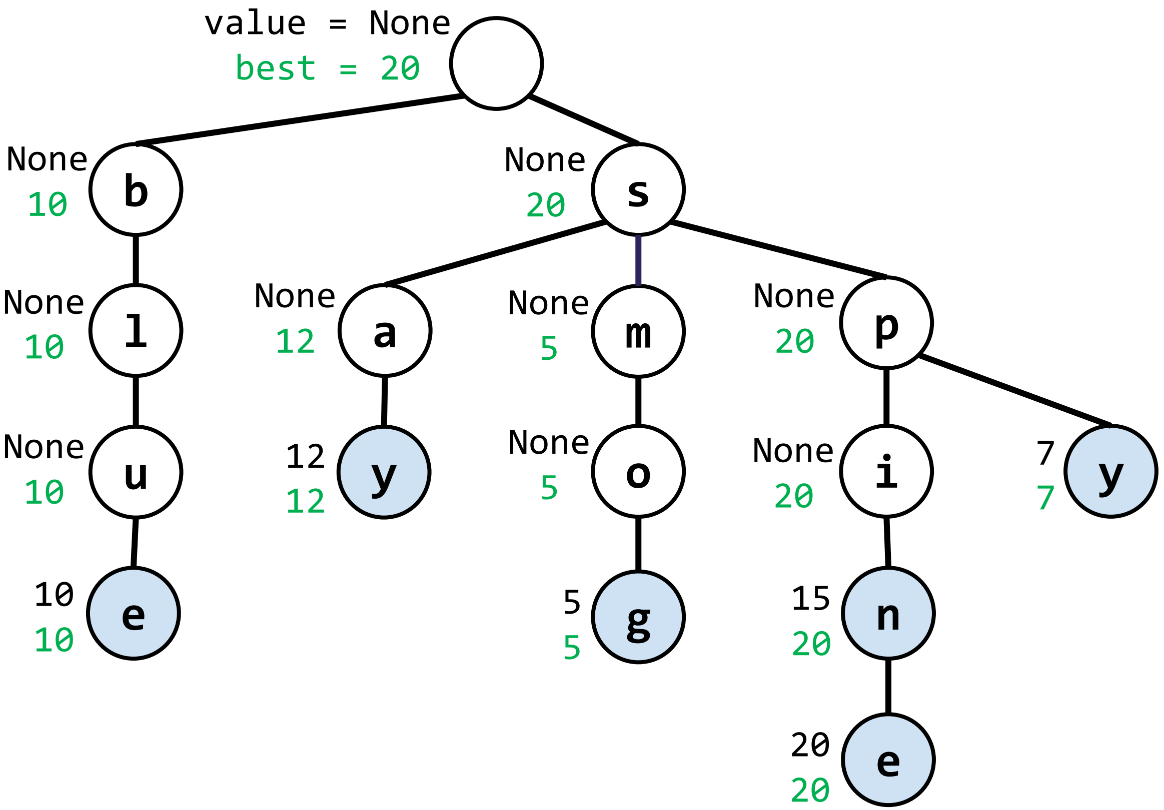
1. Call `keysWithPrefix("s")`
   ["say", "smog", "spin", "spine", "spy"]
2. Return the 3 keys with highest relevance
   ["spine", "spin", "say"]



Q: This algorithm is slow — why? How can we optimize?

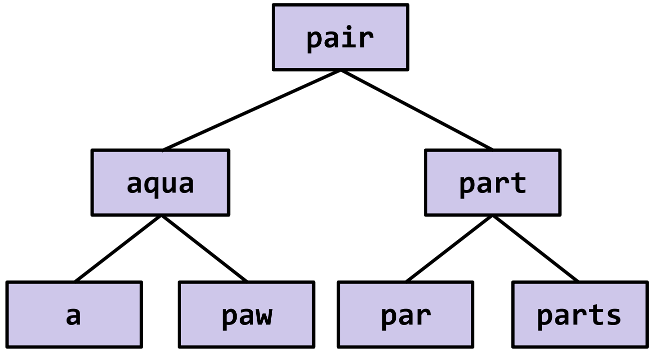
Improving Autocomplete with Tries

- **A**: short queries, such as `"s"`, require checking the relevance for billions of matching Strings
  - We only need to keep the top 10
- **Solution:** prune the search space
  - Each node stores its own relevance and maximum relevance of descendants
  - Check that maximum relevance of a subtree is greater than top 10 Strings collected so far before exploring

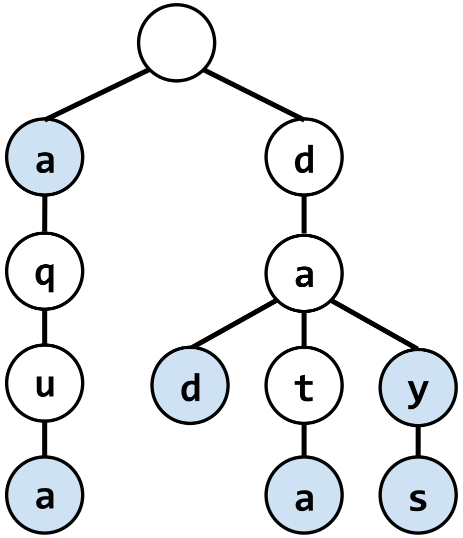


Trie Takeaways

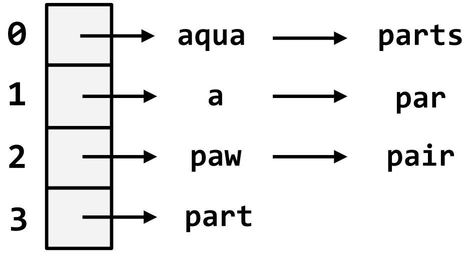
- Tries can be used for storing Strings (or any sequential data)
- Real-world performance often better than Hash Table or Search Tree
- Many different implementations: `DataIndexedCharMap`, Hash Tables, BSTs (and more possible data structures within nodes), and TSTs
- Tries enable efficient prefix operations like `keysWithPrefix`



Binary Search Tree



Trie



Hash Table