LEC 25

CSE 373

Tries

Slides originally made by Eric Fan

BEFORE WE START

Instructor Hunter Schafer

TAs
Ken Aragon
Khushi Chaudhari
Joyce Elauria
Santino Iannone
Leona Kazi
Nathan Lipiarski
Sam Long
Amanda Park
Paul Pham
Mitchell Szeto
Batina Shikhalieva
Ryan Siu
Elena Spasova
Alex Teng
Blarry Wang
Aileen Zeng

Learning Objectives

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

Today is an optional, “cool” data structure day. Since it’s not part of our core course, we don’t outline learning objectives.

The class session for today will just be Q&A since we don’t have practice for this material.

Lecture Outline

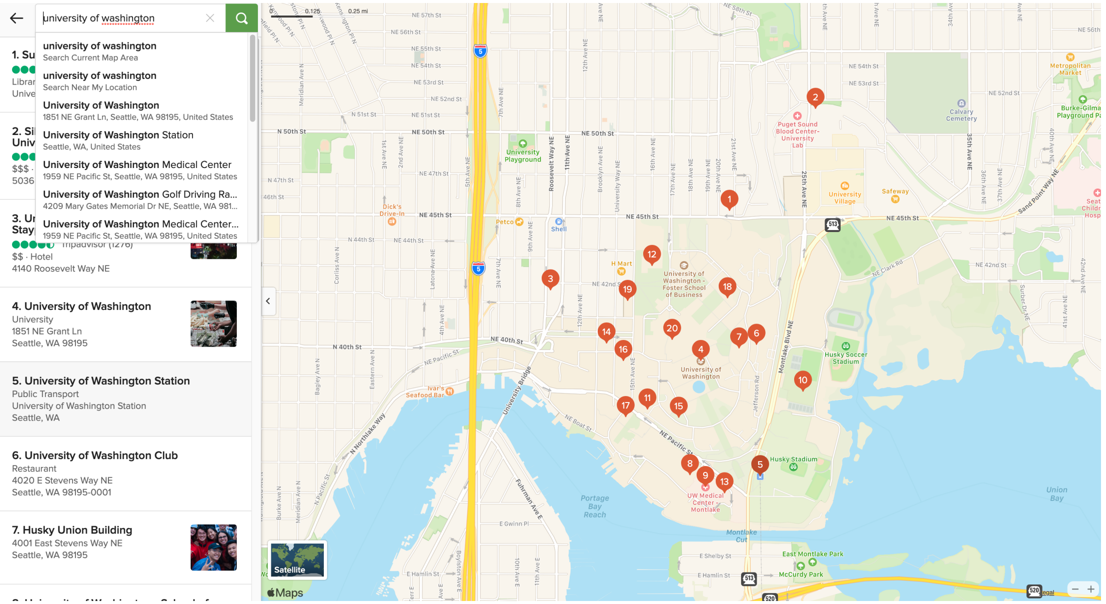
- Tries Introduction

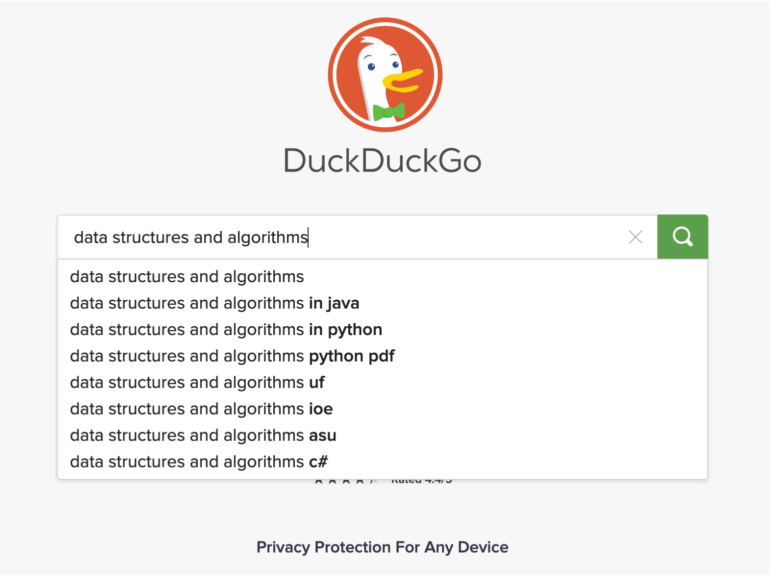
- Implementing a Trie using arrays

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

Autocomplete

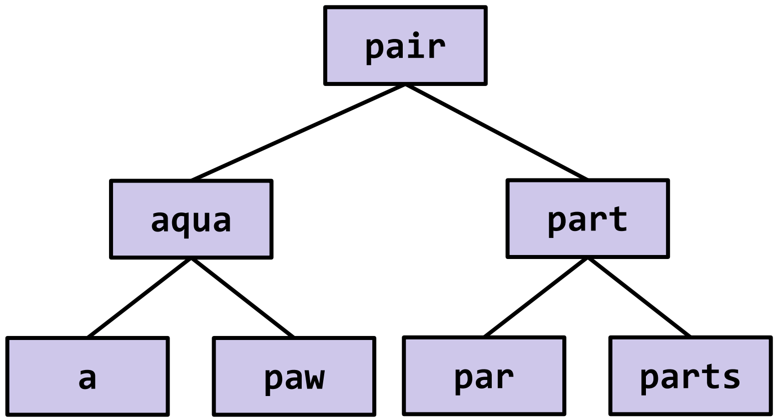
- Search Engines support autocomplete.
- How do you efficiently implement autocomplete with the ADTs we know so far?
- Formal Problem: Given a "prefix" of a string, find all strings in a set of possible strings that have the given prefix.



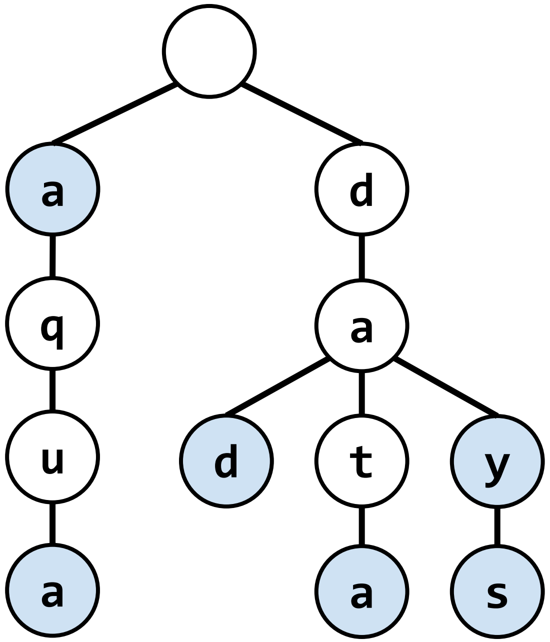


Tries: A *Specialized* Data Structure

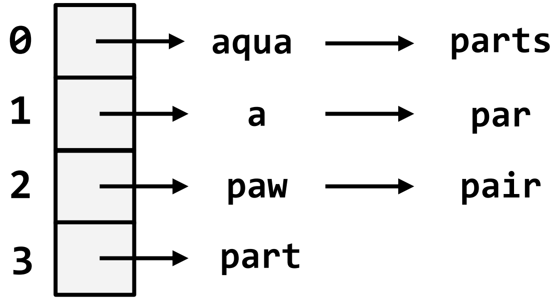
- Tries are a character-by-character set-of-Strings implementation
- Nodes store *parts of keys* instead of *keys*



Binary Search Tree



Trie



Hash Table

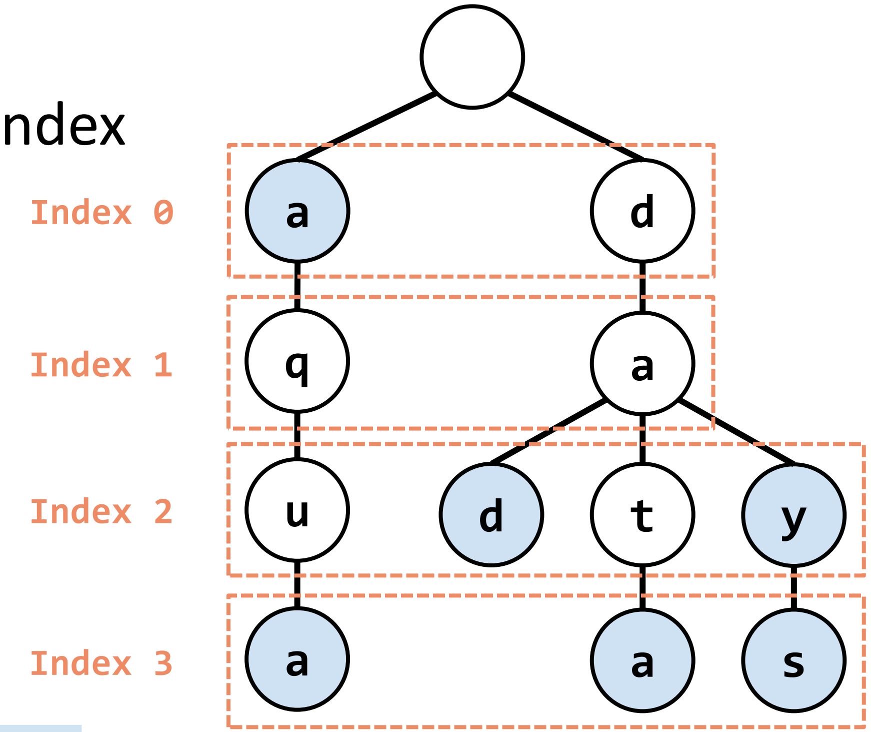
Abstract Trie

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

  a, aqua, dad,
  0 0 1 2 3 0 1 2

  data, day, days
  0 1 2 3 0 1 2 0 1 2 3

- 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

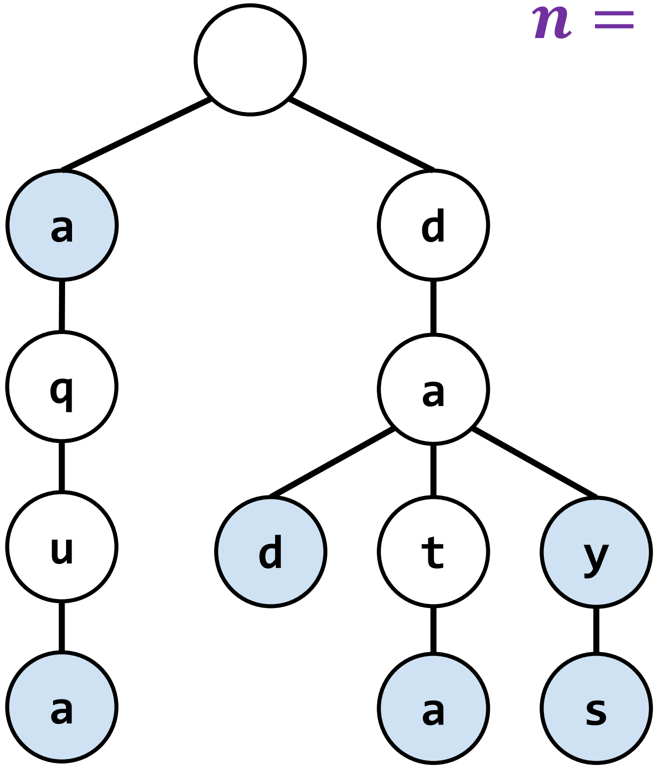
a, aqua, dad, data, day, days

$n = 6$

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



Abstract Trie

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

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

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

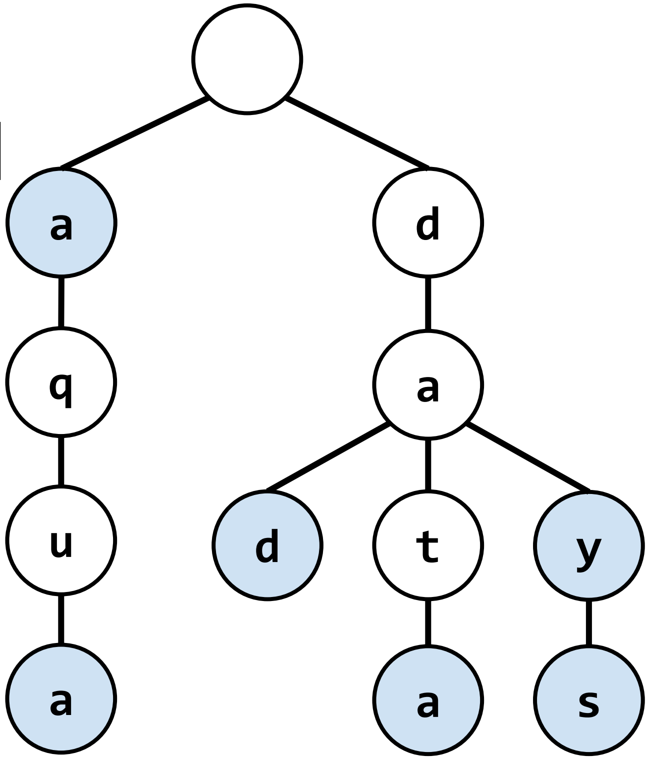
  `keysWithPrefix("day")` returns `["day", "days"]`

- **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"`



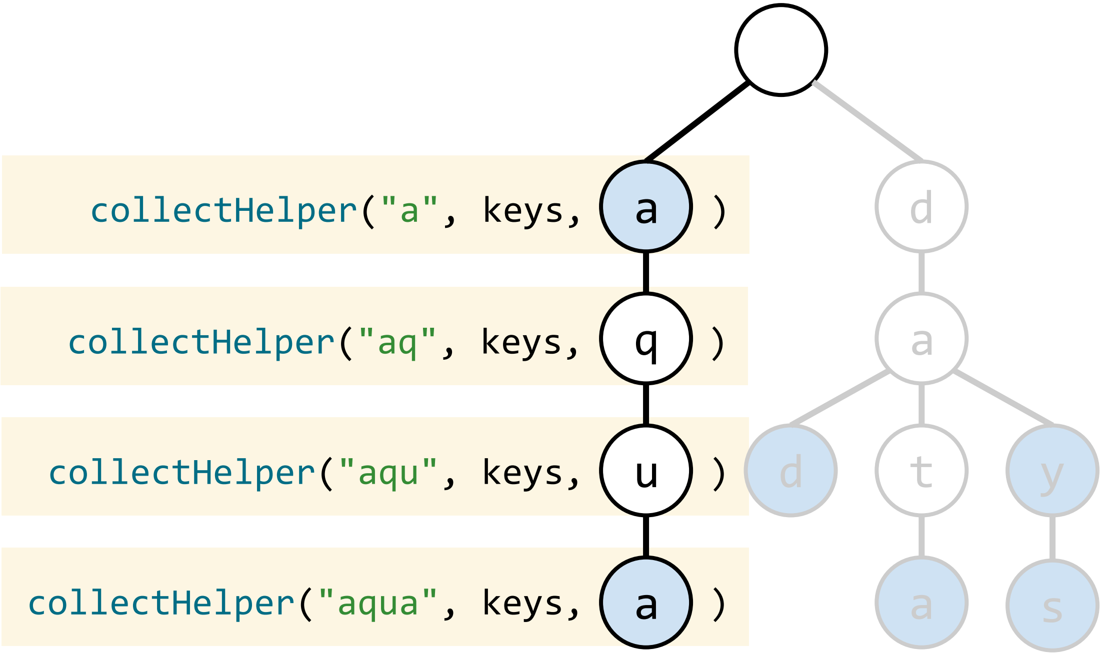
Abstract Trie

Collecting Trie Keys

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

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

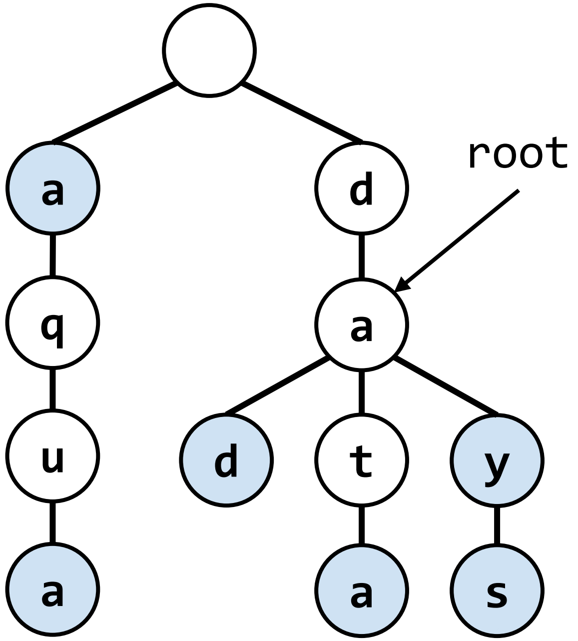
```
List collect() {
    List keys;
    for (Node c : root.children) {
        collectHelper(n.char, keys, c);
    }
    return keys;
}

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



`keysWithPrefix` Implementation

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

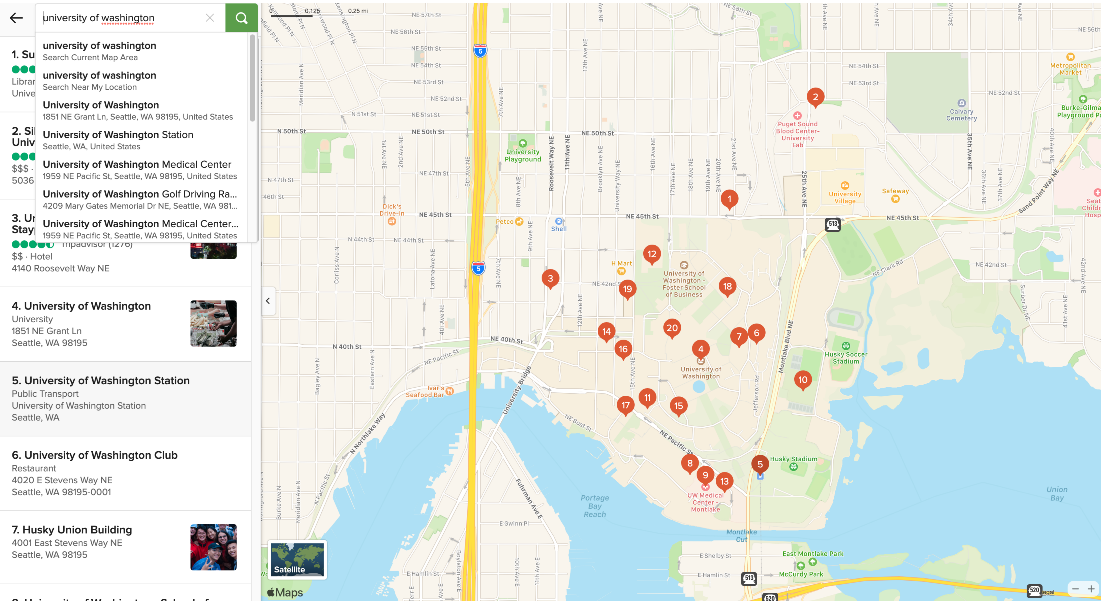
```
List keysWithPrefix(String prefix) {
    Node root;  // Node corresponding to given prefix
    List keys;  // Empty list to store keys

    for (Node n : root.children) {
        collectHelper(prefix + n.char, keys, c);
    }
}

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

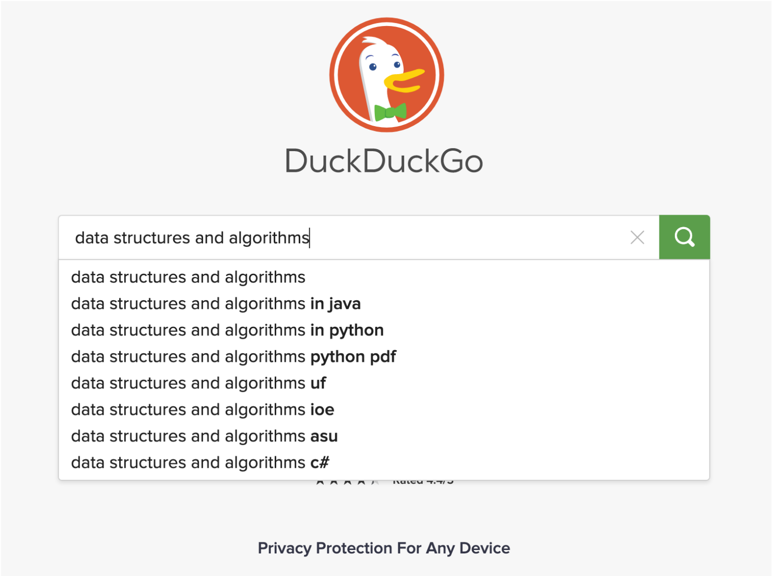


Autocomplete with Tries

- 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





Lecture Outline

- Tries Introduction

- **Implementing a Trie using arrays**

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

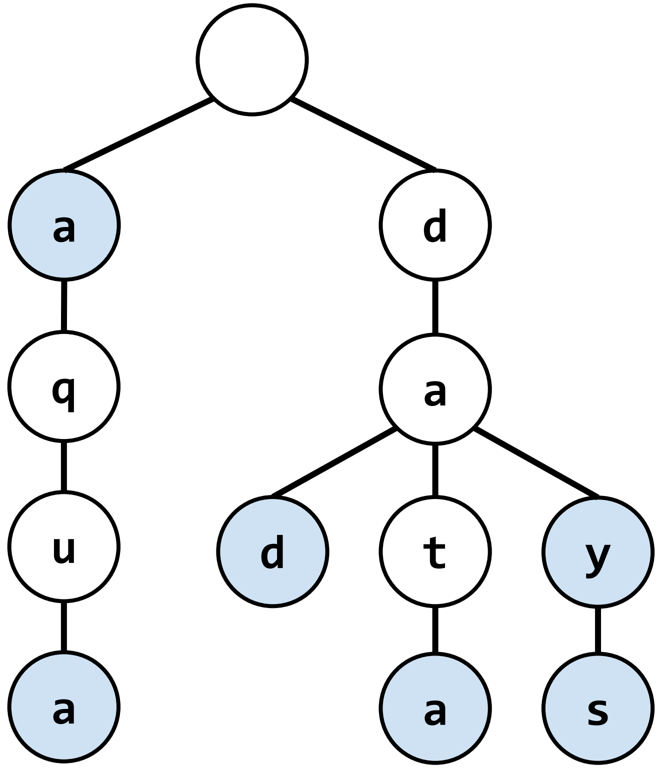
Trie Implementation Idea: *Encoding*

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	

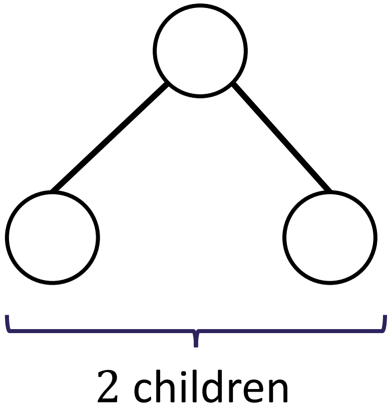
DataIndexedCharMap Pseudocode

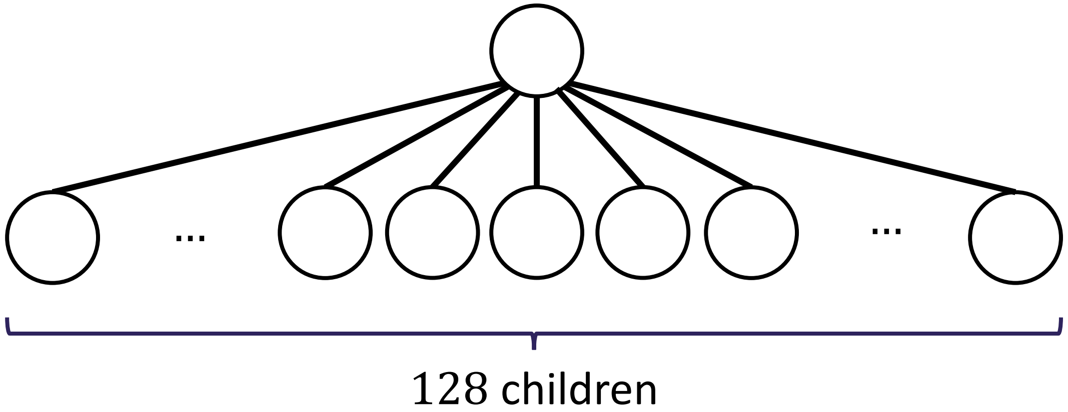
```
class TrieSet {
    final int R = 128;  // # of ASCII encodings
    Node overallRoot;

    // 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 Structure for Trie Implementation

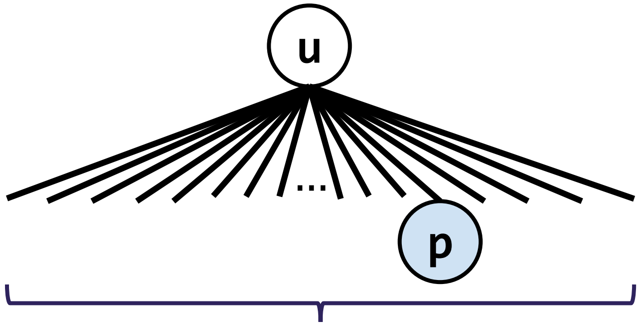
- Think of a Binary Tree
  - Instead of two children, we have 128 possible children
  - Each child represents a possible next character of our Trie
- How could we store these 128 children?



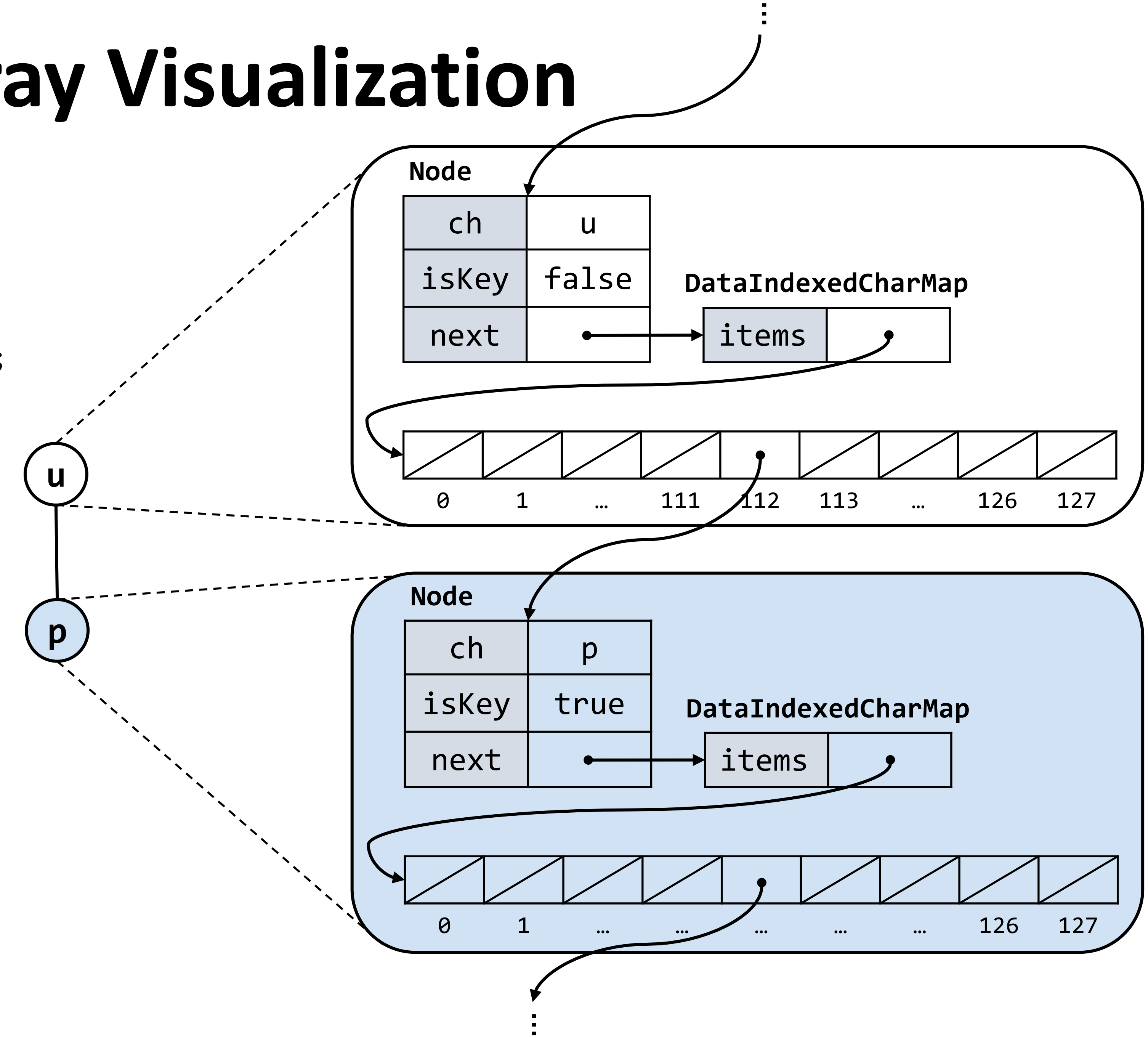


Data-Indexed Array Visualization

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

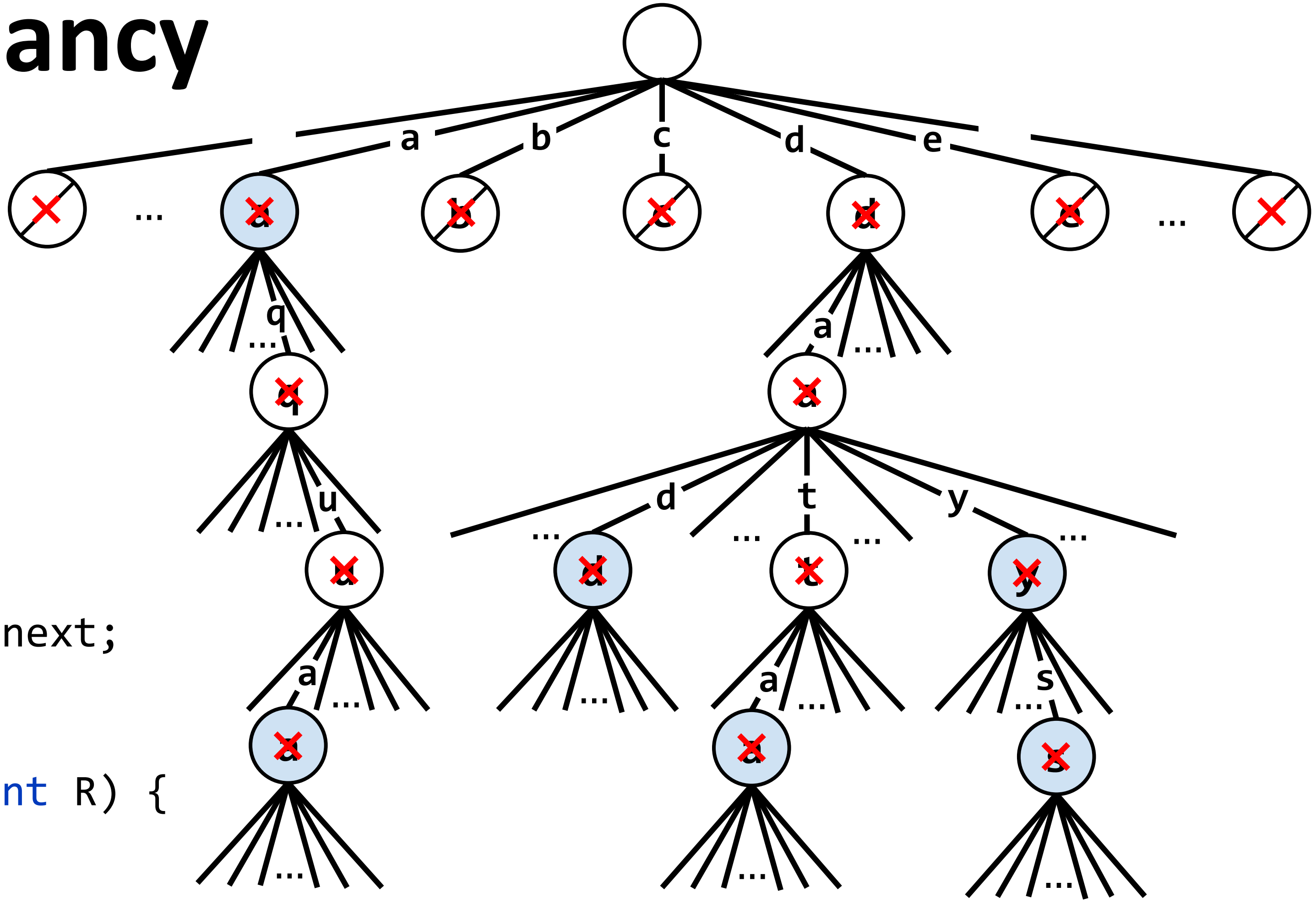


R = 128 links, 127 null



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);
        }
    }
}
```



Itempool

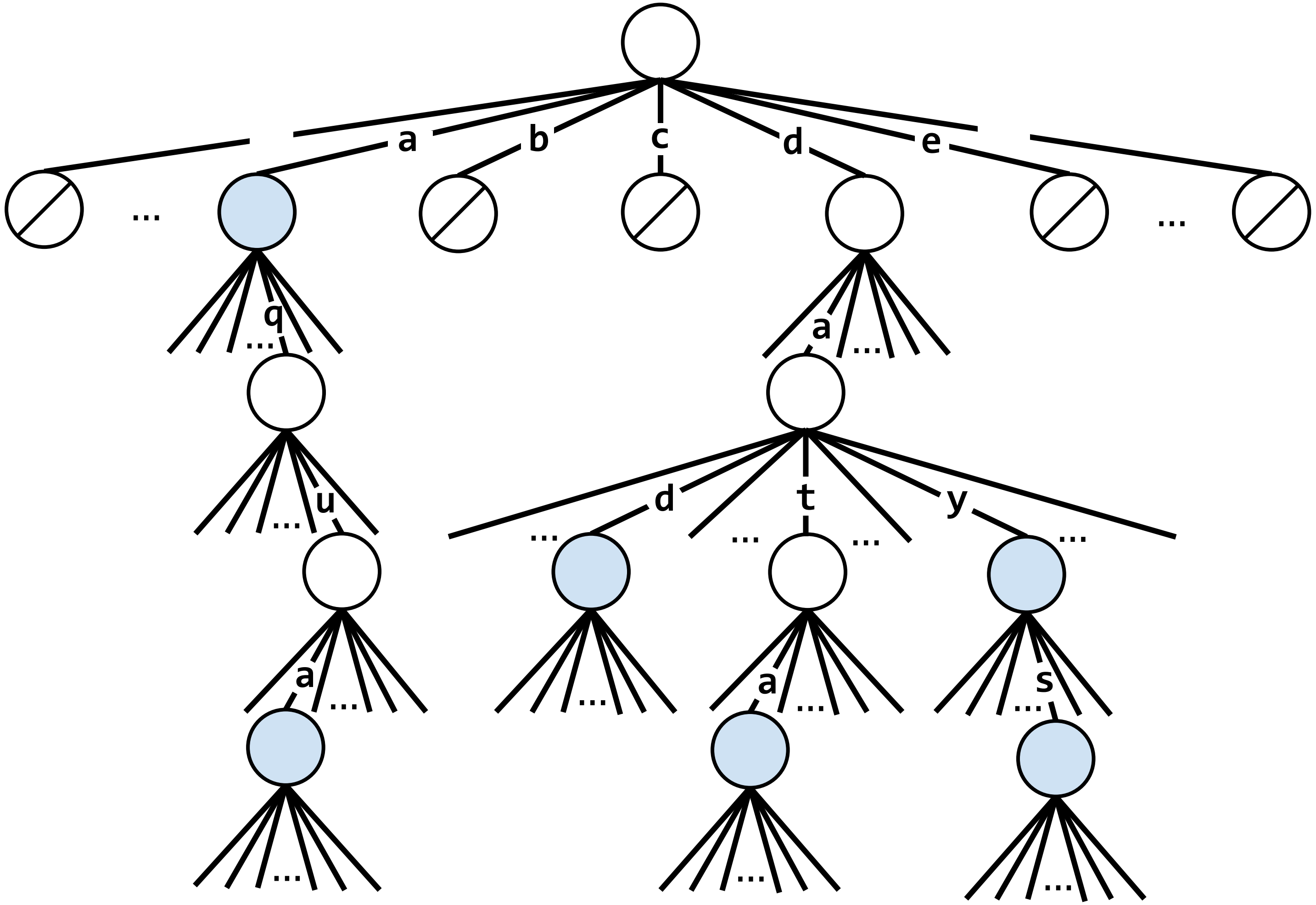
Pause Video when Prompted

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

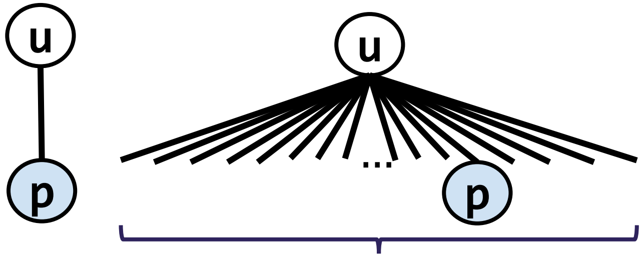
a) Yes

b) No

c) I'm not sure...



Runtime Comparison



R = 128 links, 127 `null` (unused)

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

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

* In-practice runtime

** Where p is the number of strings with the given prefix. Usually p << n.

- 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

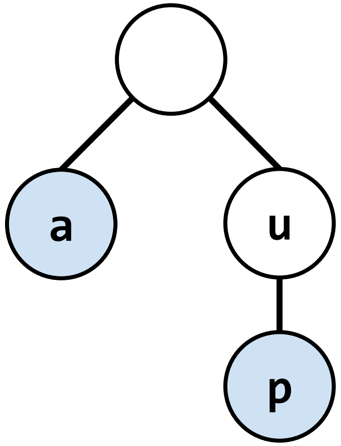
Lecture Outline

- Tries Introduction

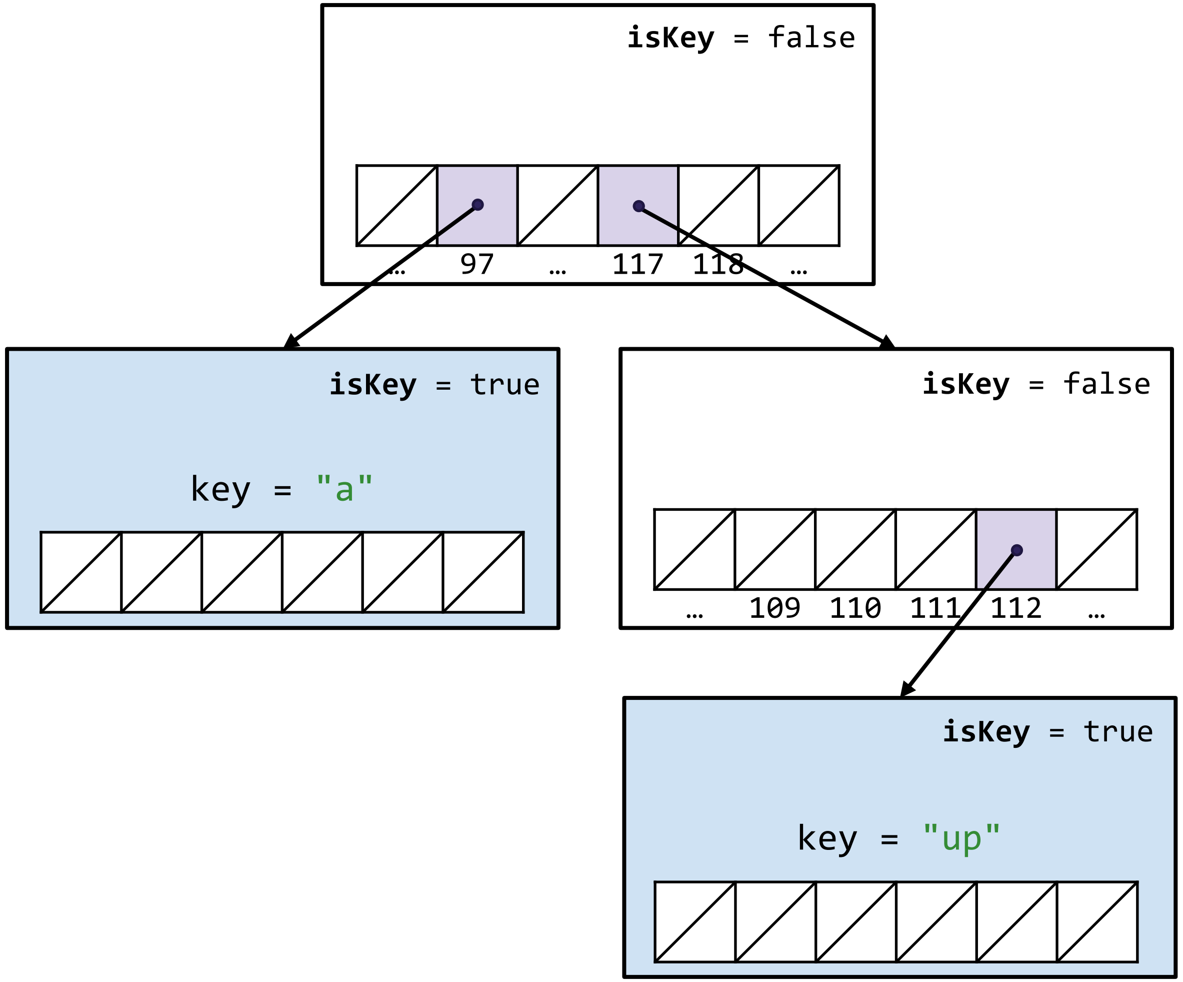
- Implementing a Trie using arrays

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

DataIndexedCharMap Implementation



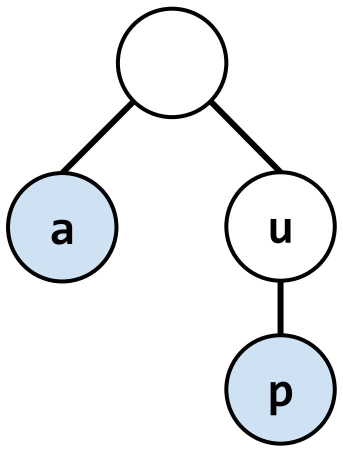
Abstract Trie



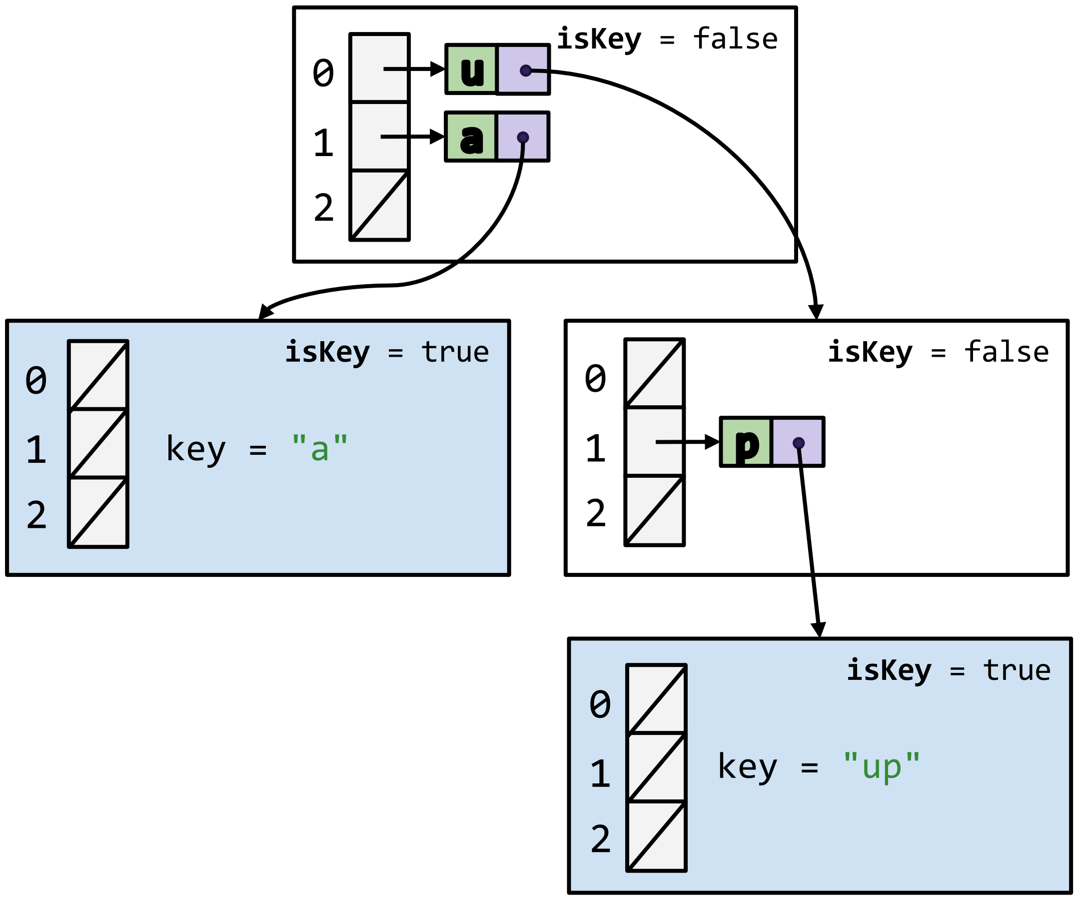
Data-Indexed Array Trie

Hash Table-based Implementation

- Use Hash Table to find character at a given index



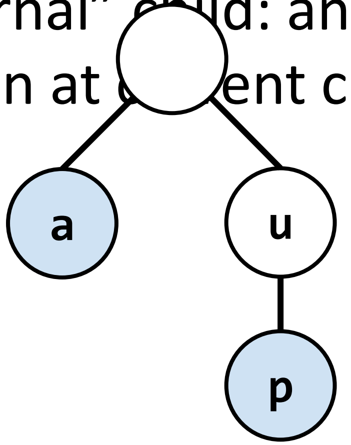
Abstract Trie



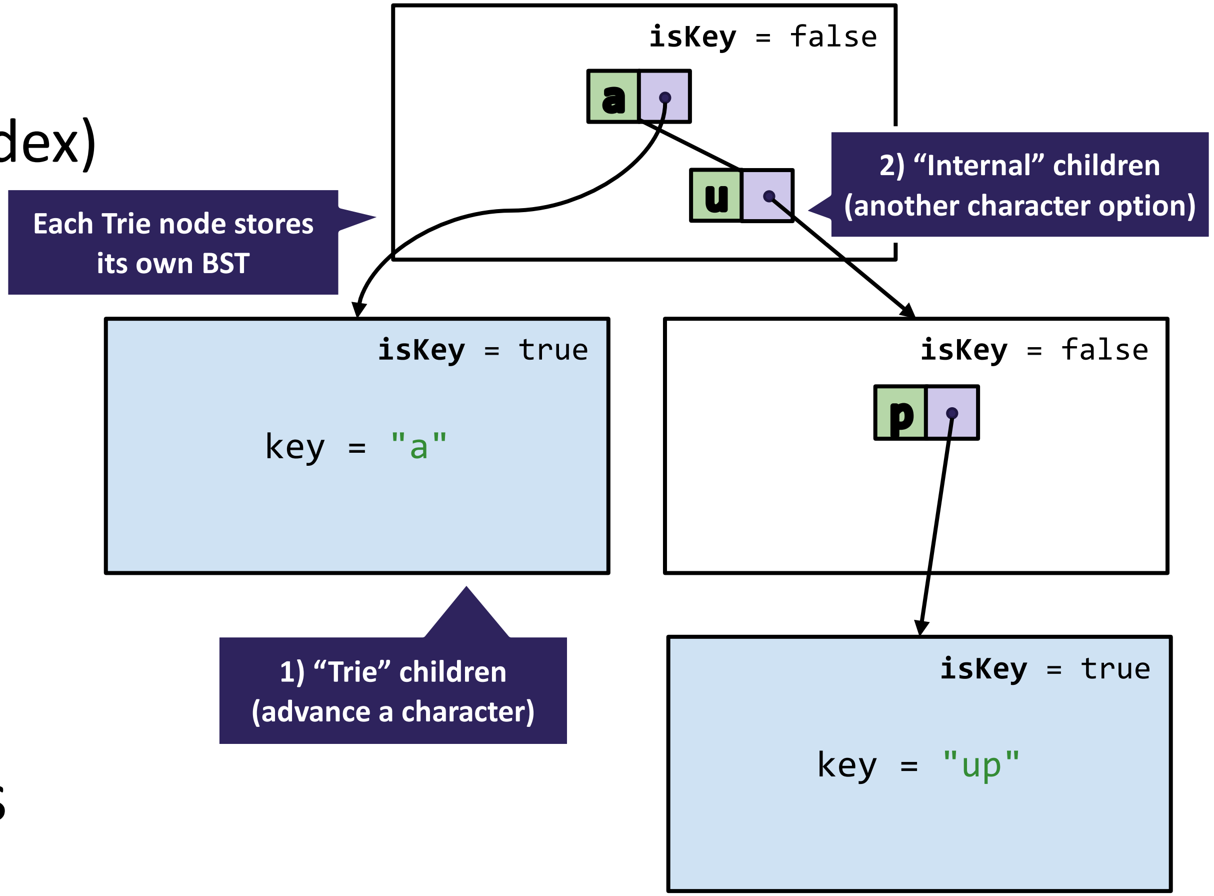
Hash Table-based Trie

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

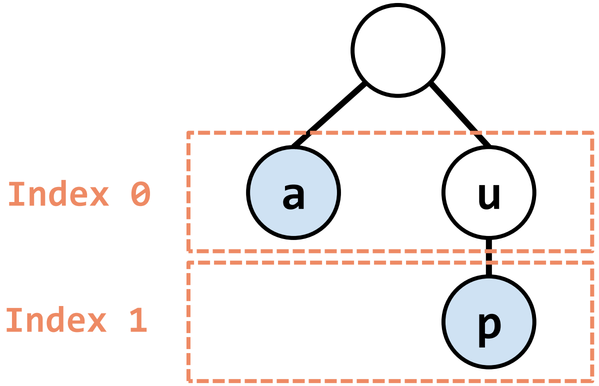


BST-based Trie

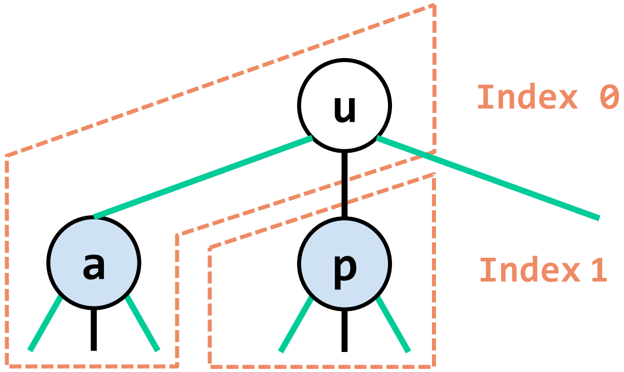
- Both are essentially child references
  - Could we simplify this design?

Ternary Search Tree (TST) Implementation

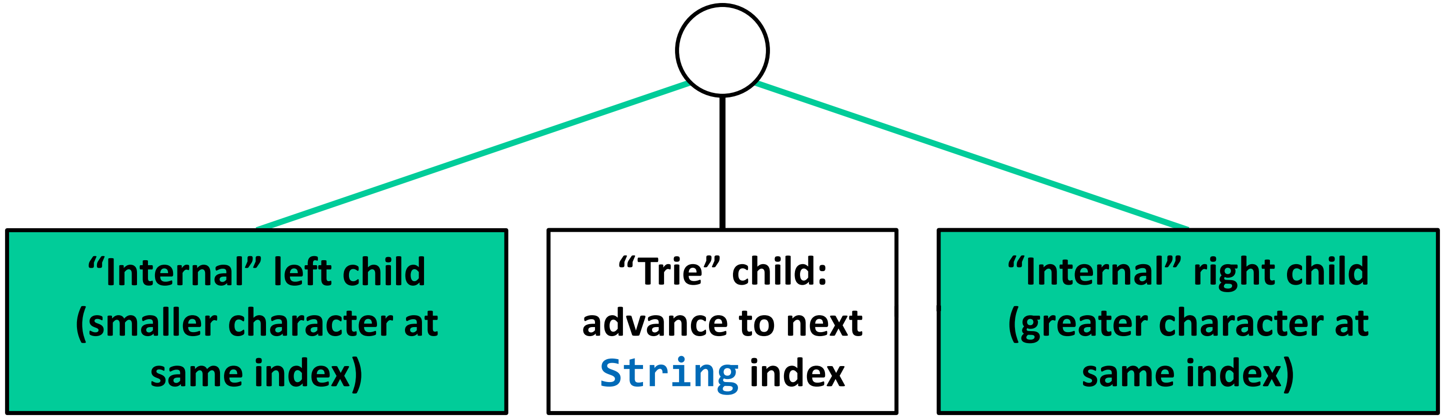
- Combines character mapping with Trie itself



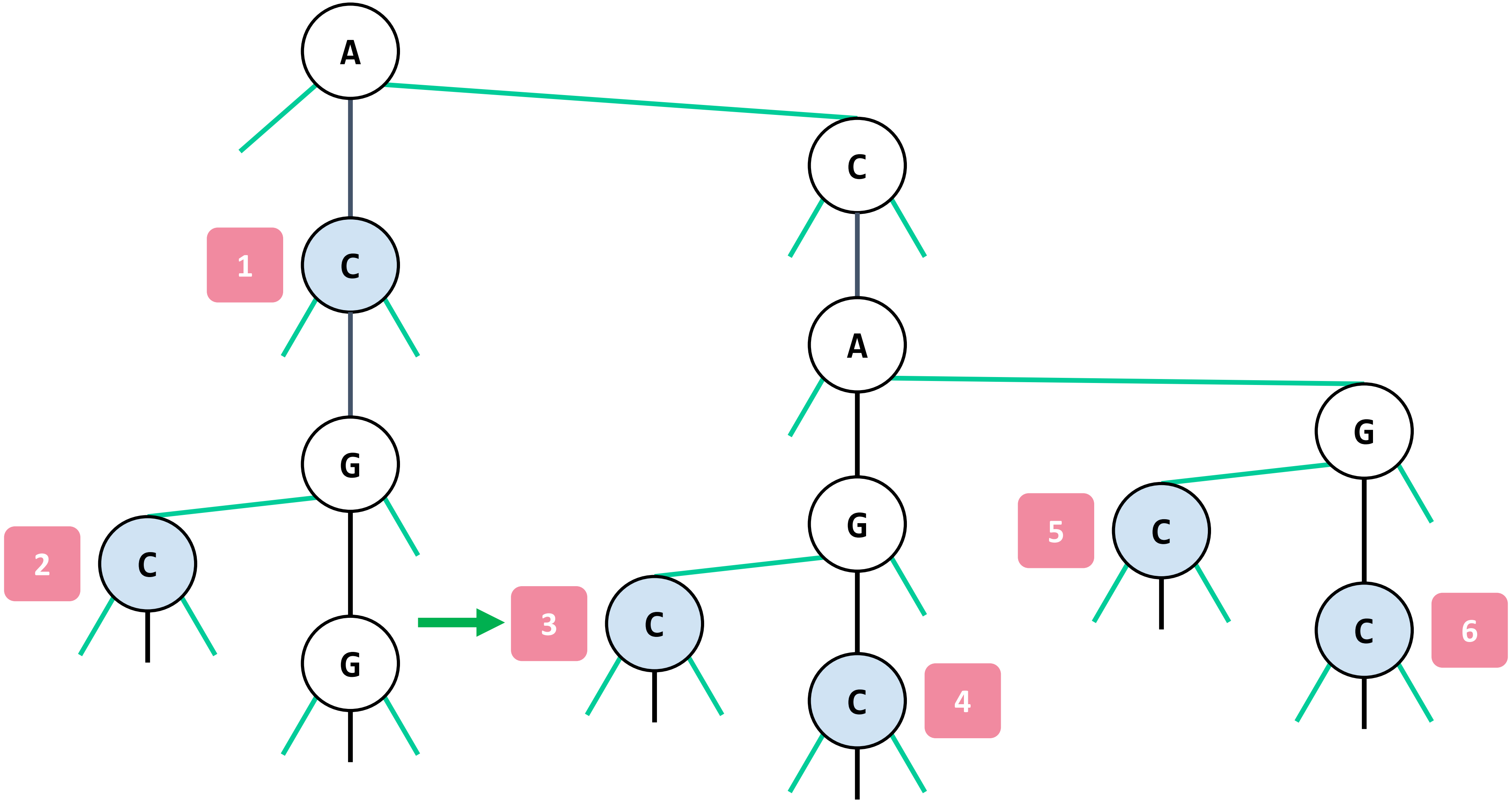
Abstract Trie



Ternary Search Tree (TST)



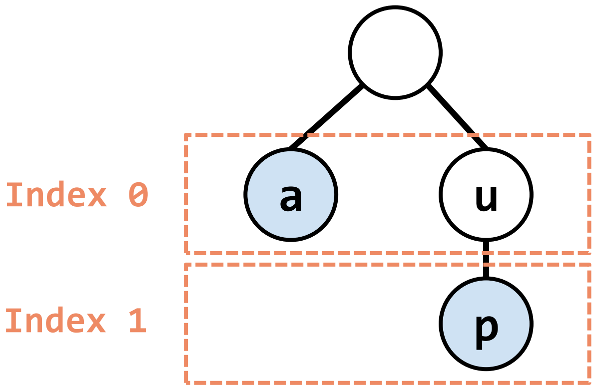
Which node is associated with the key "CAC"?



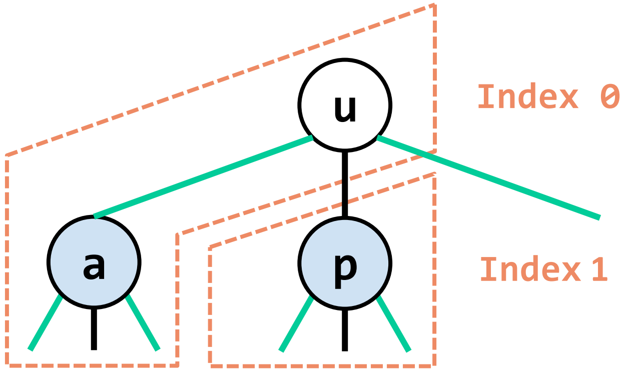
Searching in a TST

- 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



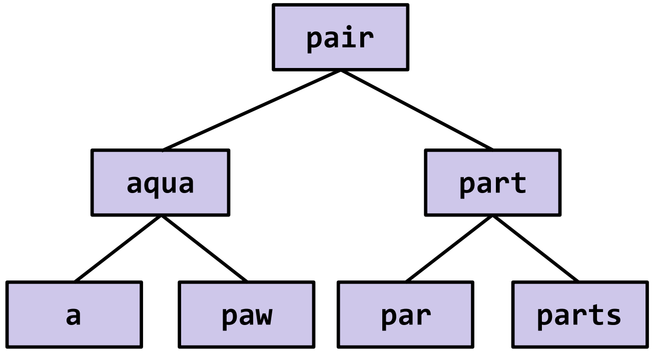
Abstract Trie



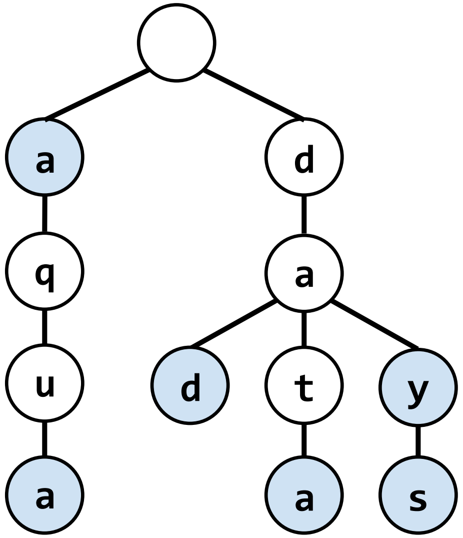
Ternary Search Tree (TST)

Trie Takeaways

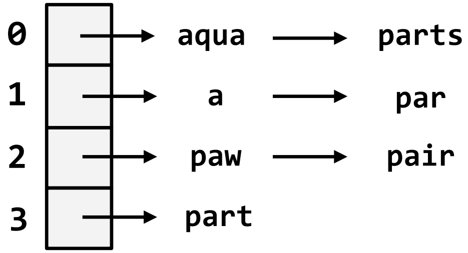
- Tries can be used for storing Strings (or any sequential data)
- Real-world performance often better than Hash Table or Search Tree
- Many different implementations: `DataIndexedCharMap`, Hash Tables, BSTs (and more possible data structures within nodes), and TSTs
- Tries enable efficient prefix operations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



Binary Search Tree



Trie



Hash Table