Dictionary and Set ADTs; Tries

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- "Due dates":
  - Exercise 1 is "due" this Friday
  - Quiz 0 is available on Gradescope and "due" tomorrow night
- Before section tomorrow:
  - Try gitlab and IntelliJ, so TAs can help debug any issues during section
  - Please verify your section's Zoom links are in Canvas
- Syllabus posted!
- But previous lecture's video isn't (thank you for your patience!)

Learning Objectives

- Describe the List, Stack, Queue, Set, and Dictionary ADTs and the use-cases they are good for
- Identify when a Trie can be used and the useful properties it provides
- Write code for prefix algorithms to run over a Trie

Lecture Outline

- **Review: ADTs we know**
- Dictionary and Set ADTs
- The trie data structure
  - Introduction
  - Implementation
  - Prefix matching

ADTs So Far (1 of 2)

List ADT. A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index
- A list has a size defined as the number of elements in the list
- Elements can be added to the front, back, *or any index in the list*
- Optionally, elements can be removed from the front, back, *or any index in the list*

- Data structures that implement the List ADT include LinkedList and ArrayList

- When we restrict List's functionality, we end up with the 2 other ADTs we've seen so far

ADTs So Far (2 of 2)

Stack ADT. A collection storing an ordered sequence of elements.

- A stack has a size defined as the number of elements in the stack
- Elements can only be added and removed from the top ("LIFO")

Queue ADT. A collection storing an ordered sequence of elements.

- A queue has a size defined as the number of elements in the queue
- Elements can only be added to one end and removed from the other ("FIFO")

- Data structures that implement these ADTs are variants of LinkedList and ArrayList

Lecture Outline

- Review: ADTs we know
- **Dictionary and Set ADTs**
- The trie data structure
  - Introduction
  - Implementation
  - Prefix matching

Dictionary ADT (1 of 2)

Dictionary ADT. A collection of keys, each associated with a value.

- A dictionary has a size defined as the number of elements in the dictionary
- You can add and remove (key, value) pairs , but the keys are unique
- Each value is accessible by its key via a "find" or "contains" operation

Terminology: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

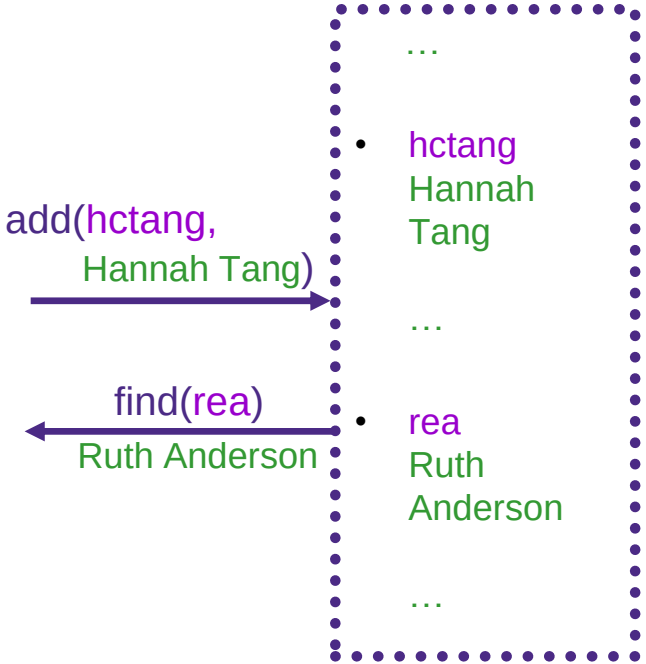
- Also known as "**Map ADT**"
  - add(k, v)
  - contains(k)
  - find(k)
  - remove(k)
- Naïve implementation: a list of (key, value) pairs

```
class KVPair<Key, Value> {
  Key k;
  Value v;
}

LinkedList<KVPair> dict;
```

Dictionary ADT (2 of 2)

- Operations:
  - **`add(k, v)`:**
    - places (k,v) in dictionary
    - if key already present, typically overwrites existing entry
  - **`find(k)`:**
    - Returns v associated with k
  - **`contains(k)`:**
    - Returns true if k is in the dictionary
  - **`remove(k)`:**
    - …



We will tend to emphasize the keys, but don't forget about the stored values!

A Modest Few Uses for Dictionaries

- Any time you want to store information according to some key and be able to retrieve it efficiently – a **dictionary** is the ADT to use!
  - Lots of programs do that!

Networks	Router tables
Operating systems	Page tables
Compilers	Symbol tables
Databases	Dictionaries with other nice properties
Search	Inverted indices, phone directories, …
Biology	Genome maps

Set ADT

Set ADT. A collection of keys.

- A set has a size defined as the number of elements in the set
- You can add and remove keys, but the contained values are unique
- Each key is accessible via a "contains" operation

- Operations:
  - add(v)
  - contains(v)
  - remove(v)

No find()! (why?)

- Naïve implementation: a dictionary where we ignore the "value" portion of the (key, value) pair

```
class Item<Key> {
  Key k;
}

LinkedList<Item> set;
```

Comparison: Set ADT vs. Dictionary ADT

- The *Set* ADT is like a Dictionary without any values
  - A key is *present* or not (no repeats)

- For **`contains`**, **`add`**, **`remove`**, there is little difference
  - In dictionary, values are "just along for the ride"
  - So *same data-structure ideas* work for dictionaries and sets
    - Java HashSet implemented using a HashMap, for instance

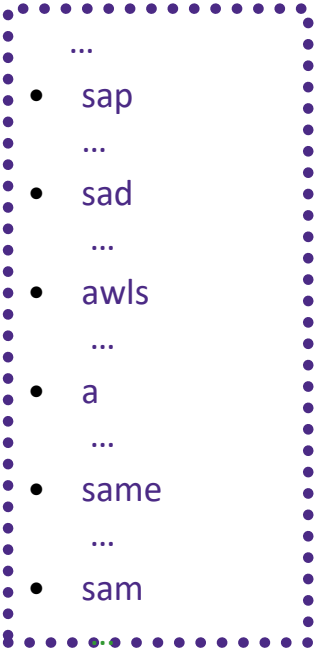
- Set ADT may have other important operations
  - **`union`**, **`intersection`**, **`isSubset`**, etc.
  - Notice these are binary operators on sets
  - We will want different data structures to implement these operators

Lecture Outline

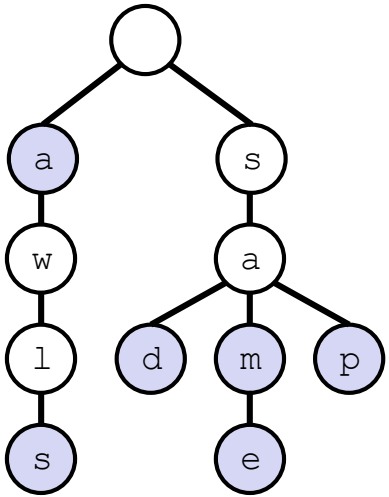
- Review: ADTs we know
- Dictionary and Set ADTs
- The trie data structure
  - **Introduction**
  - Implementation
  - Prefix matching

The Trie: A Specialized Data Structure

Tries view keys as a **sequence of characters**, some with common prefixes



Abstract Set ADT



Trie

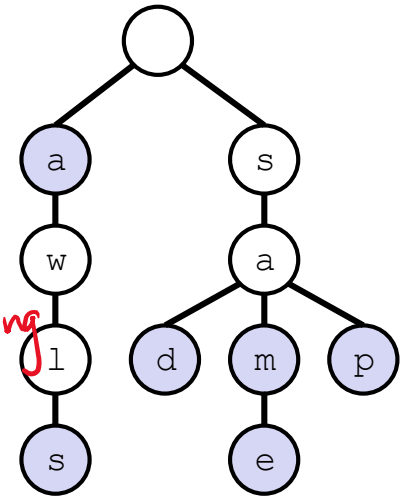
An Abstract Trie

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`

Can make this a map by attaching values to every blue node



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

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

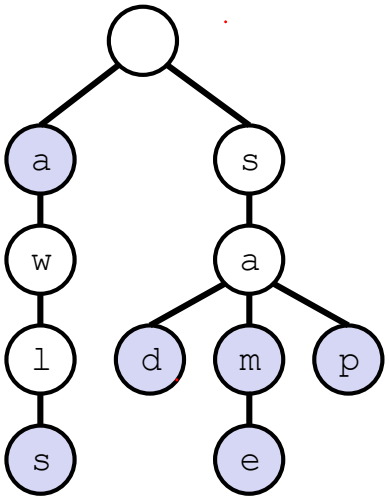
Searching in Tries

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

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

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

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



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

1. If the final node isn't purple (not a key).
2. If we fall off the tree.

Keys as "a sequence of characters" (1 of 2)

- Most dictionaries treat their keys as an "atomic blob": you can't disassemble the key into smaller components

- Tries take the opposite view: keys are a **sequence of characters**
  - `Strings` are made of `Characters`

- But "characters" don't have to come from the Latin alphabet
  - `Character` includes most Unicode codepoints (eg, 蛋糕)
  - `List<E>`
  - `byte[]`

Keys as "a sequence of characters" (2 of 2)

- But "characters" don't have to come from the Latin alphabet
  - `Character` includes most Unicode codepoints (eg 蛋糕)
  - `List<E>`
  - `byte[]`

- Tries are defined by 3 types instead of 2:
  - An "alphabet": the domain of the characters
  - A "key": a sequence of "characters" from the alphabet
  - A "value": the usual Dictionary value

Lecture Outline

- Review: ADTs we know
- Dictionary and Set ADTs
- The trie data structure
  - Introduction
  - **Implementation**
  - Prefix matching

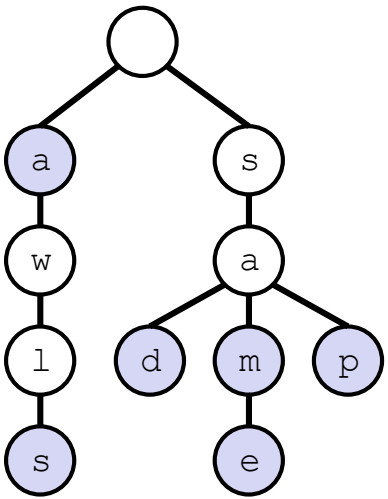
ASCII TABLE

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

Simple Trie Implementation*

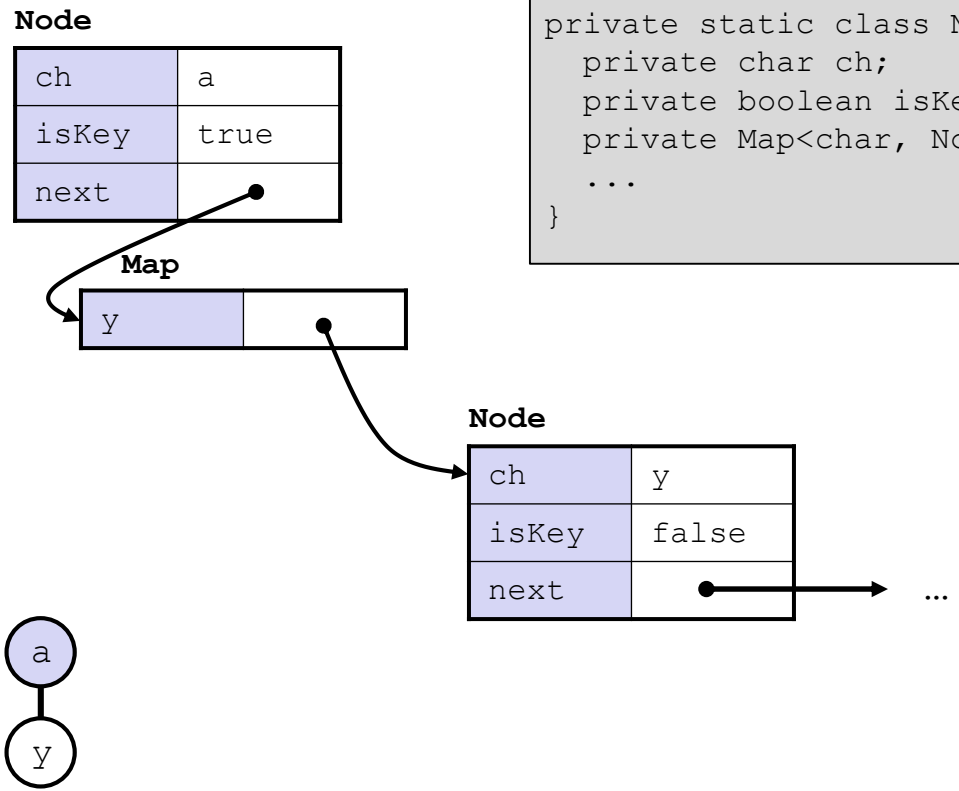
```
public class TrieSet {
  private Node root;

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



* This implementation won't work for your `HashTrieNode`; don't bother copy-and-pasting

Simple Trie Node Implementation

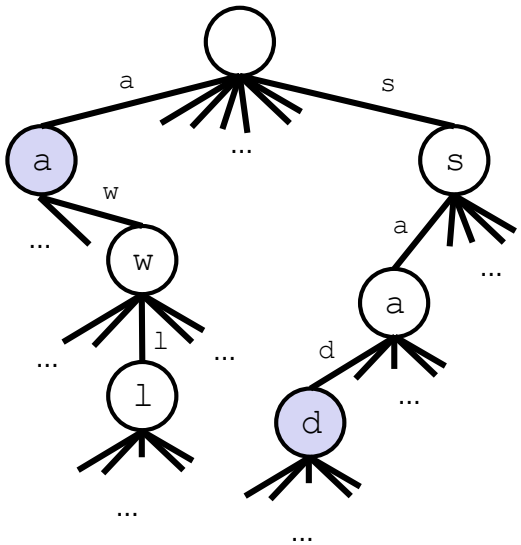



```
private static class Node {
  private char ch;
  private boolean isKey;
  private Map<char, Node> next;
  ...
}
```

Simple Trie Implementation

```
public class TrieSet {
  private Node root;

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

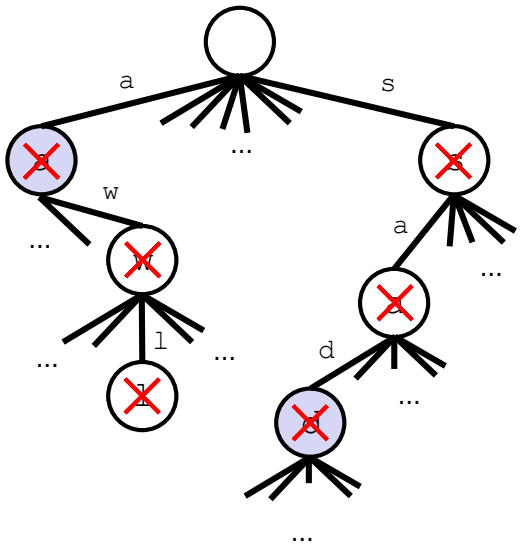


Removing Redundancy

```
public class TrieSet {
  private Node root;

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

redundant!

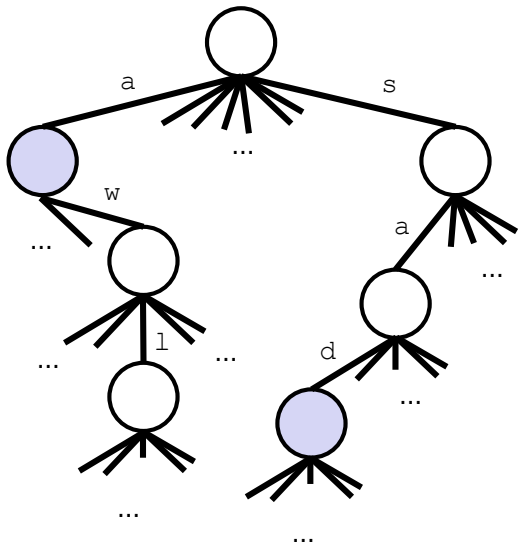


Poll Everywhere

pollev.com/cse332

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

A. Yes
B. No
C. I'm not sure



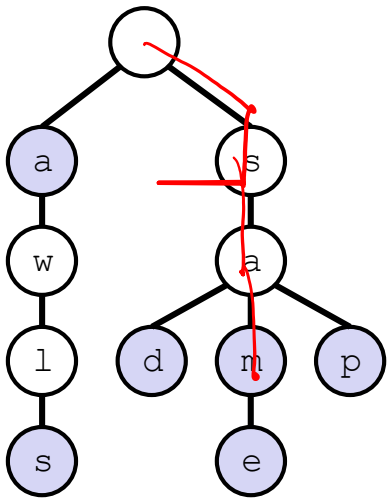
Lecture Outline

- Review: ADTs we know
- Dictionary and Set ADTs
- The trie data structure
  - Introduction
  - Implementation
  - **Prefix matching**

String-Specific Operations

- The main appeal of tries is their efficient prefix matching!

- **Prefix match**
  - `findPrefix("sa")`
- **Longest prefix**
  - `longestPrefixOf("sample")`

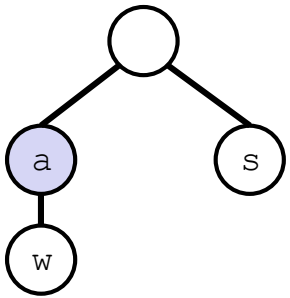


Collecting Trie Keys

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`.



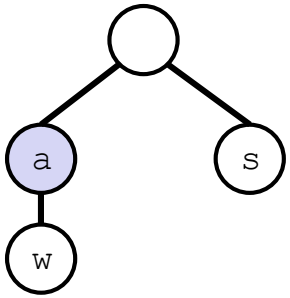
`colHelp(String s, List<String> x, Node n)`

1. ???

Collecting Trie Keys

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

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



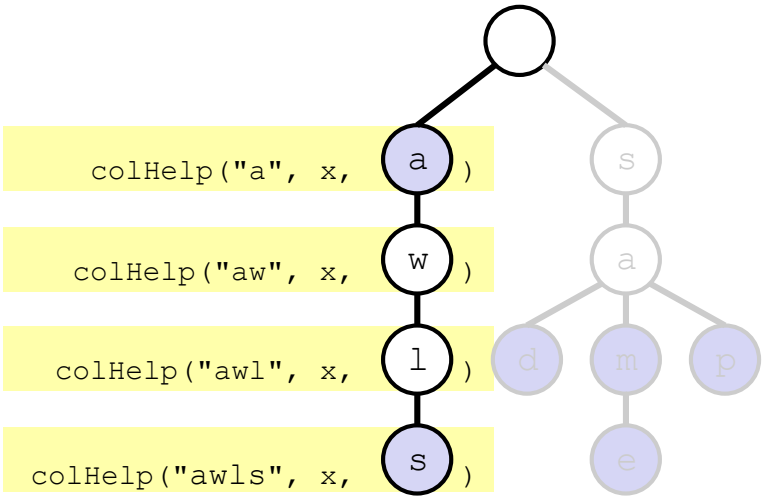
1. Create an empty list of results `x`.
2. For character `c` in `root.next.keys()`:
   Call `colHelp(c, x, root.next.get(c))`.
3. Return `x`.

`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))`.

Collecting Trie Keys

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

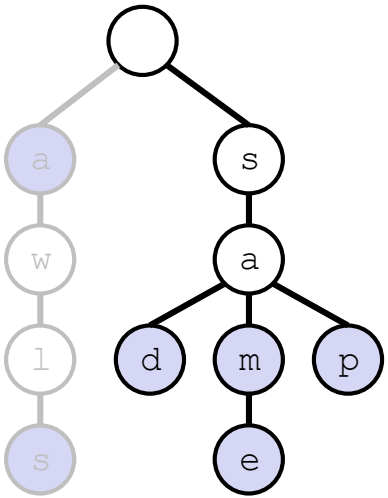


`colHelp(String s, List<String> x, Node n)`

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

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



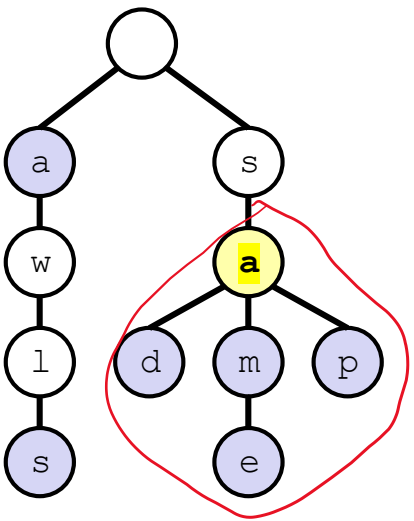
`colHelp(String s, List<String> x, Node n)`

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

Describe in English an algorithm for `findPrefix`.

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



Summary

- The **Dictionary** ADT maps keys to values
- The **Set** ADT is like a Dictionary without any values

- A trie data structure implements the Dictionary and Set ADTs

- Tries have many different implementations
  - Could store HashMap/TreeMap/any-dictionary within nodes
  - Much more exotic variants change the trie's representation, such as the Ternary Search Trie

- Tries store sequential keys
  - … which enables very efficient prefix operations like `findPrefix`