Dictionary and Set ADTs; Tries

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- (Remember: forming an opinion and answering questions – even if the opinion turns out to be wrong – helps you learn better. Please *engage* in these activities as you prepare for lecture)

- We've discussed Stack, Queue, and List ADTs. Let's imagine a "Dictionary" ADT, which maps words ("keys") to their definitions ("values")

- Design a data structure to implement this ADT
  - What methods should it have?
  - How should it store the data?

- This data structure should be *new to you*; please do not design something you already know!

Announcements

- Before section tomorrow, try gitlab and IntelliJ, so TAs can help debug any issues during section

- Lecture recordings are in *Panopto*, not in Zoom.

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

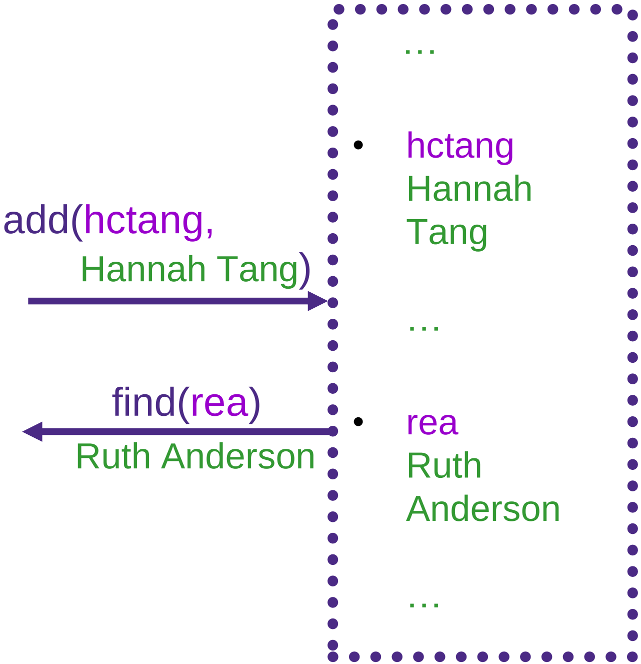
Terminology: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

- 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)

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

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

LinkedList<Item> set;
```

gradescope

gradescope.com/courses/256241

- What, if any, differences are there between a Set and a Dictionary ADT?
  - Remember that this is a difference in *functionality*, not in implementation

- Similar to our earlier example with savory pies, can the same data structure(s) be used to implement a Set and a Dictionary?
  - Yes
  - No

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

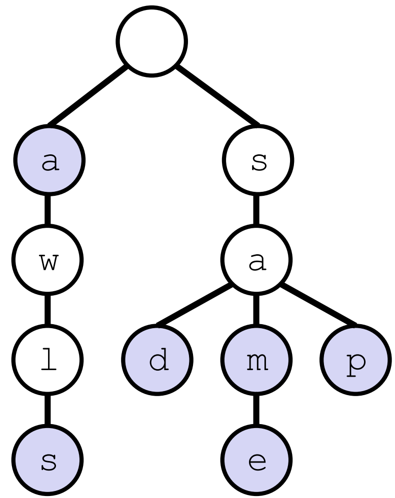
- Review: ADTs we know
- Dictionary and Set ADTs
- The trie data structure
  - **Introduction**
  - Implementation
  - Prefix matching

The Trie: A Specialized Data Structure

- Tries view its keys as:
  - a **sequence of characters**
  - some (hopefully many!) sequences share common prefixes

- sap
- sad
- awls
- a
- same
- sam

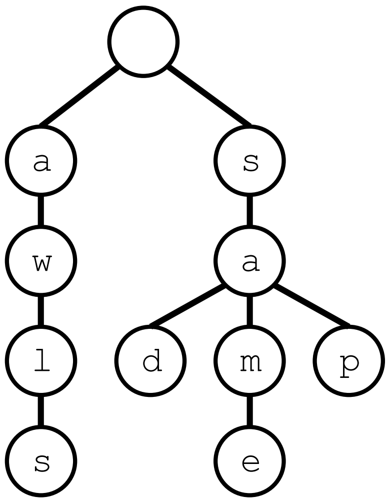
Set ADT



Trie

Trie: An Introduction

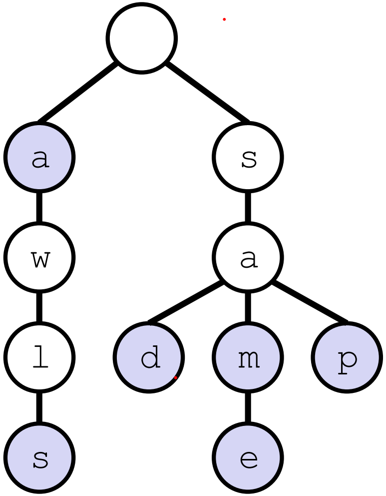
- Each level of the tree represents an index in the string
  - Children at that level represent possible characters at that index
- This abstract trie stores the set of strings:
  - `awls, a, sad, same, sap, sam`
- How to deal with `a` and `awls`?
  - Mark which nodes *complete* a string (shown in purple)



Searching in Tries

Two ways to fail a contains() check:

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



Input String	Fall Off? / Is Key?	Result
contains("sam")	hit / purple	True
contains("sa")	hit / white	False
contains("a")	hit / purple	True
contains("saq")	fell off / n/a	False

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**
  - `String`s are made of `Character`s

- 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

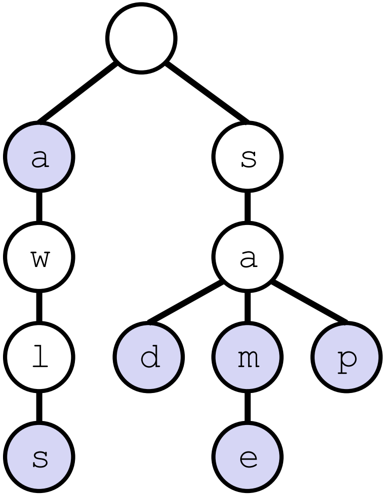
Lecture questions: pollev.com/cse332

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	_					

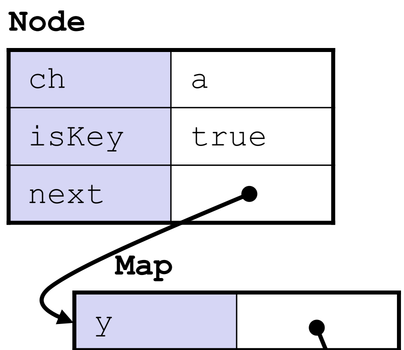
Simple Trie Implementation*

```
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();
    }
  }
}
```

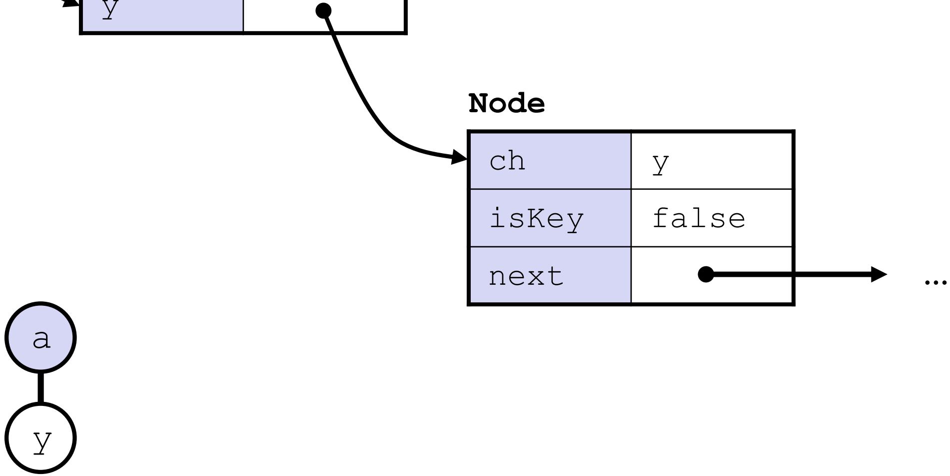


* This implementation won't work for your `HashTrieNode`; don't bother copy-and-pasting

Simple Trie Node Implementation

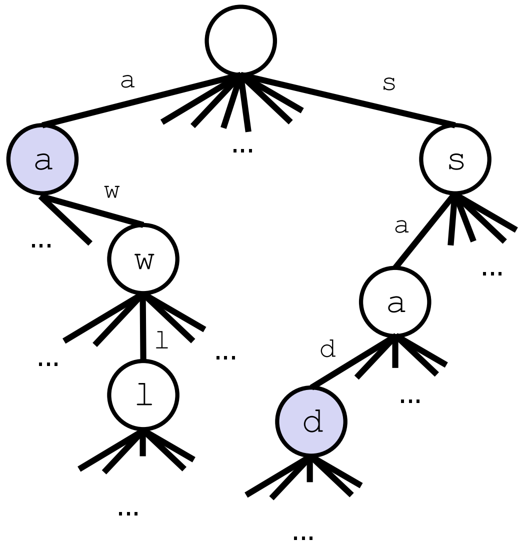



```
private static class Node {
  private char ch;
  private boolean isKey;
  private Map<char, Node> next;
  ...
}
```



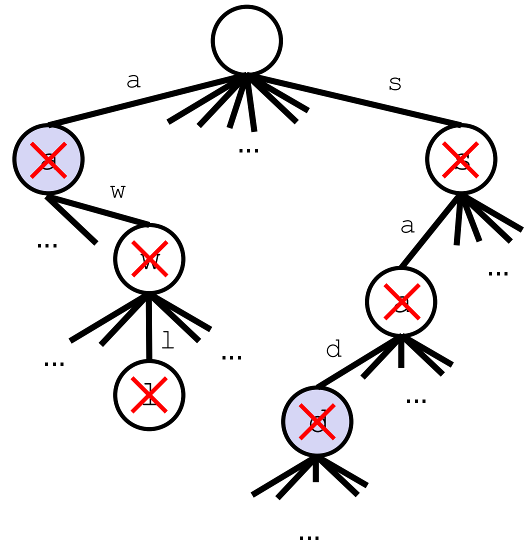
Simple Trie Implementation

```
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();
    }
  }
}
```

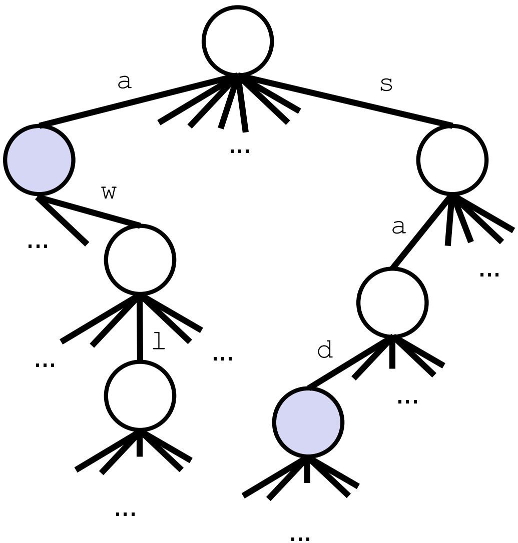


gradescope

gradescope.com/courses/256241

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

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



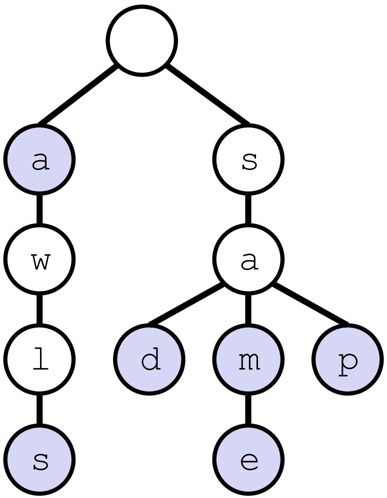
Lecture Outline

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

Lecture questions: pollev.com/cse332

Trie-Specific Operations

- The main appeal of tries is prefix matching!
  - Why? Because they view their keys as sequences that can have prefixes



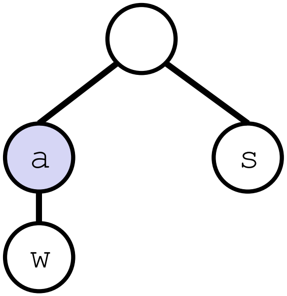
- **Longest prefix**
  - `longestPrefixOf("sample")`
  - Want: `{"sam"}`
- **Prefix match**
  - `findPrefix("sa")`
  - Want: `{"sad", "sam", "same", "sap"}`

Related Problem: Collecting Trie Keys

- Imagine an algorithm that collects *all* the keys in a trie:
  - `collect(): ["a","awls","sad","sam","same","sap"]`

- It could be implemented as follows:

```
Create an empty list of results x
Foreach character c in root.next.keys():
  call colHelp(c, x, root.next.get(c))
return x
```



- How would `colHelp()` be implemented?

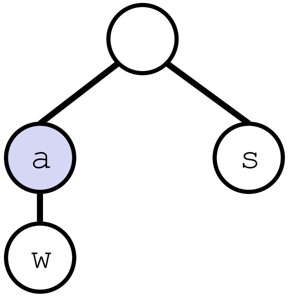
```
colHelp(String s, List<String> x, Node n) {
  // TODO(me): implement this
}
```

gradescope

gradescope.com/courses/256241

- Implement `colHelp()` in pseudocode

```
List<String> collect(Node root) {
  List<String> x;
  Foreach character c in root.next.keys():
    colHelp(c, x, root.next.get(c))
  return x
}
```

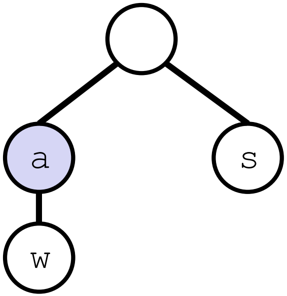



```
colHelp(String s, List<String> x, Node n) {
  // TODO(me): implement this
}
```

Collecting Trie Keys: Solution

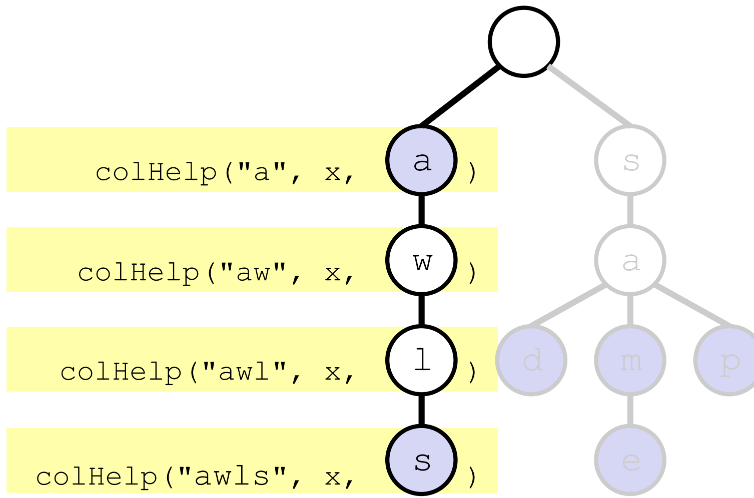
- Imagine an algorithm that collects *all* the keys in a trie:
  - `collect(): ["a","awls","sad","sam","same","sap"]`
- How would `colHelp()` be implemented?

```
colHelp(String s, List<String> x,
        Node n) {
  If n.isKey
    x.add(s)
  Foreach character c in n.next.keys():
    colHelp(s + c, x, n.next.get(c)).
}
```



Collecting Trie Keys: Demo

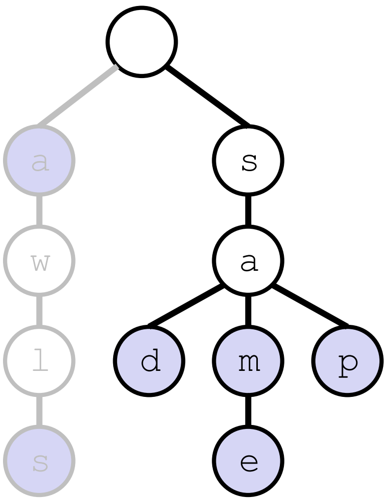
```
collect(): [
    "a",
    "awls",
]
```




```
colHelp(String s, List<String> x, Node n) {
  If n.isKey
    x.add(s)
  Foreach character c in n.next.keys():
    colHelp(s + c, x, n.next.get(c)).
}
```

Collecting Trie Keys: Demo

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

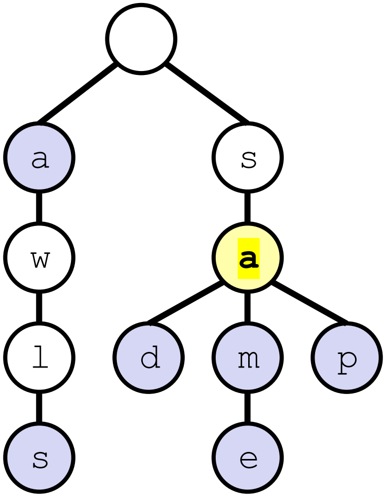



```
colHelp(String s, List<String> x, Node n) {
  If n.isKey
    x.add(s)
  Foreach character c in n.next.keys():
    colHelp(s + c, x, n.next.get(c)).
}
```

Prefix Operations with Tries

- Now that we have `colHelper()`, how would you implement an algorithm for `findPrefix()`?

- `findPrefix("sa")` should return:
  - `["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`