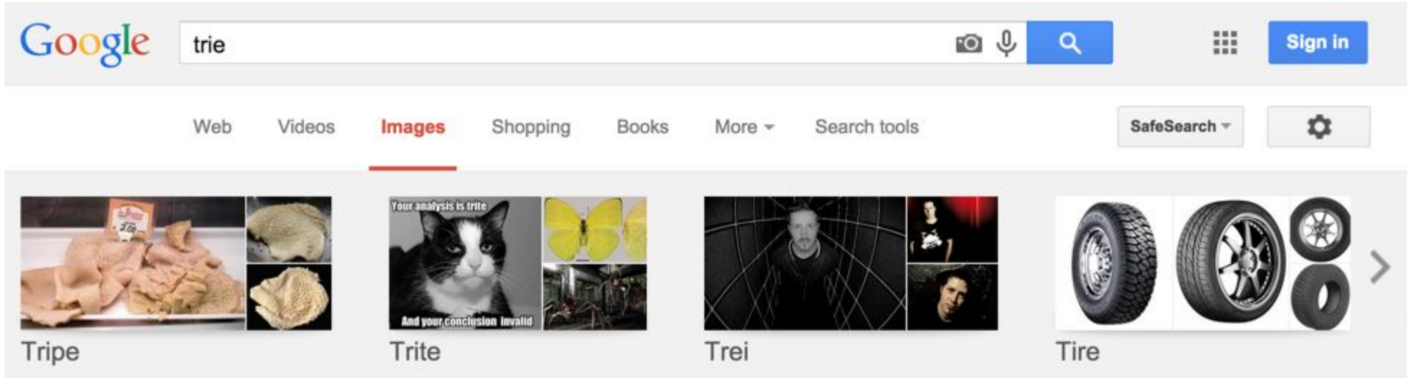
Announcements

Project 2A/2B is out.

- 2A is due Saturday, get started ASAP.
- 2A and 2B are both about as difficult as project 1A (Deques).
- The autograder is very minimal. Provide only sanity checks.
- Even if you pass all the sanity checks, you will not get all the points.
    - Full autograder will be run exactly one time after the deadline.
- Material for 2B will be covered on Friday's lecture.

Reminder: Challenge labs.

- This week's challenge lab is a classic really tough midterm problem from last Spring.
- Sign up here: http://yellkey.com/century (not required)



CS61B, 2019

Lecture 21: Tries

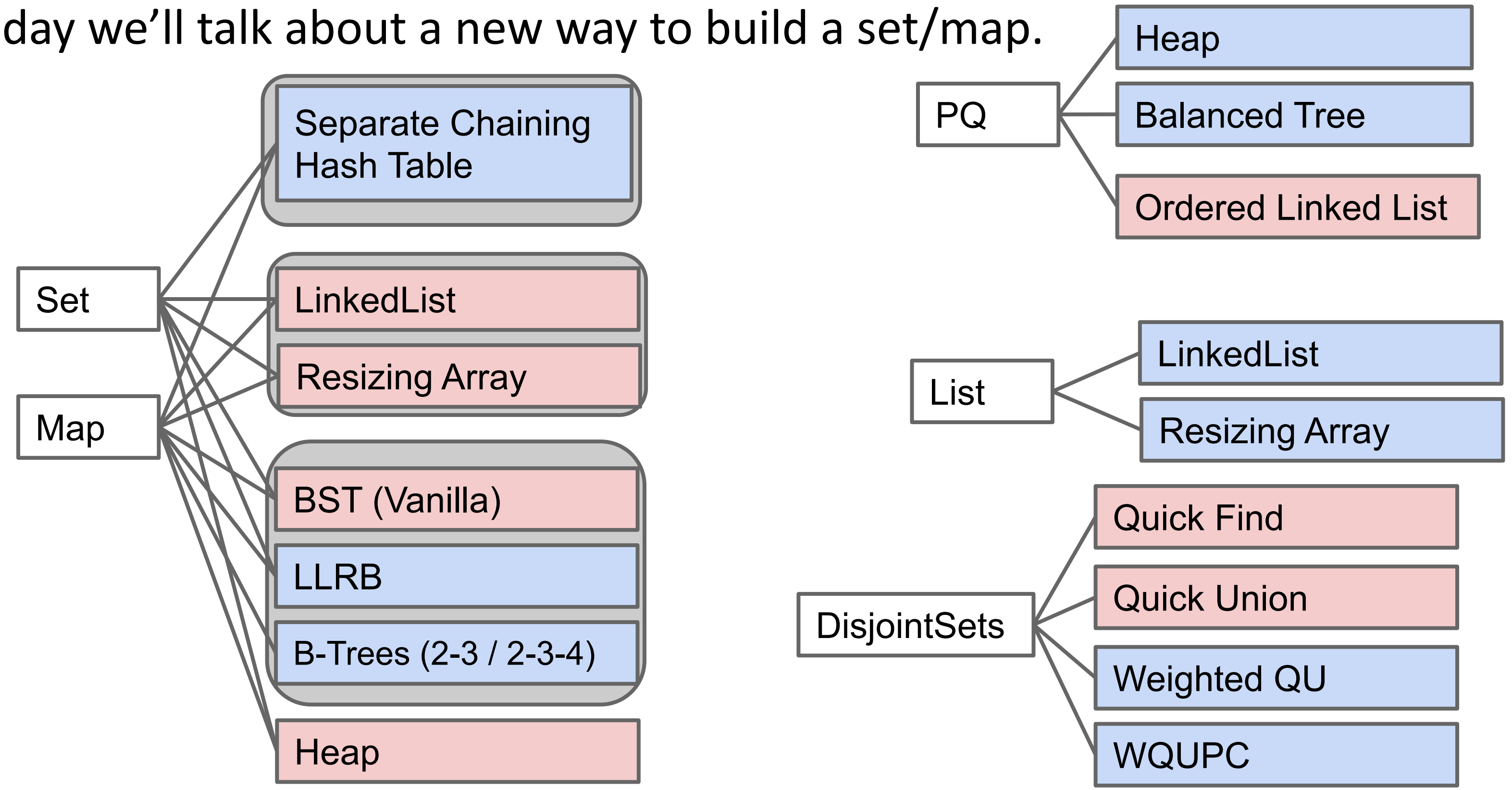
- Tries
- Trie Implementation and Performance
- Alternate Child Tracking Strategies
- Trie String Operations
- Autocomplete

Tries

Abstract Data Types vs. Specific Implementations

There are many ways to implement an abstract data type.

- Today we'll talk about a new way to build a set/map.



BST and Hash Table Set Runtimes

Runtimes for our Balanced Search Tree and Hash Table implementations were very fast.

If we know that our keys all have some common special property, we can sometimes get even better implementations.

Example: Suppose we know our keys are always single ASCII characters.

- e.g. ‘a’, ‘g’, ‘!’

	contains(x)	add(x)
Balanced Search Tree	$\Theta(\log N)$	$\Theta(\log N)$
Resizing Separate Chaining Hash Table	$\Theta(1)^{\dagger}$	$\Theta(1)^{*\dagger}$

*: Indicates “on average”.
†: Assuming items are evenly spread.

Special Case 1: Character Keyed Map

Suppose we know that our keys are always ASCII characters.

- Can just use an array!
- Simple and fast.

```java
public class DataIndexedCharMap<V> {
   private V[] items;
   public DataIndexedCharMap(int R) {
       items = (V[]) new Object[R];
   }
   public void put(char c, V val) {
       items[c] = val;
   }
   public V get(char c) {
        return items[c];
   }
}
```

	key type	get(x)	add(x)
Balanced BST	comparable	Θ(log N)	Θ(log N)
RSC Hash Table	hashable	Θ(1)†	Θ(1)*†
data indexed array	chars	Θ(1)	Θ(1)

*: Indicates "on average".
†: Assuming items are evenly spread.

R is the number of possible characters, e.g. 128 for ASCII.

Special Case 2: String Keyed Map

Suppose we know that our keys are always strings.

- Can use a special data structure known as a Trie.
- Basic idea: Store each letter of the string as a node in a tree.

	key type	get(x)	add(x)
Balanced BST	comparable	Θ(log N)	Θ(log N)
RSC Hash Table	hashable	Θ(1)†	Θ(1)*†
data indexed array	chars	Θ(1)	Θ(1)
Tries	Strings	?	?

*: Indicates "on average".
†: Assuming items are evenly spread.

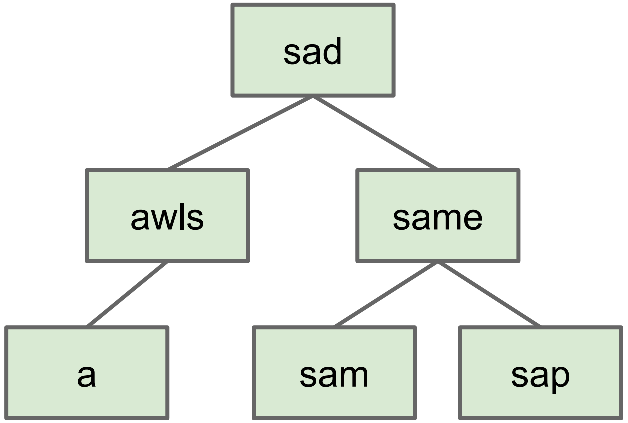
Tries will have great performance on:

- get
- add
- special string operations

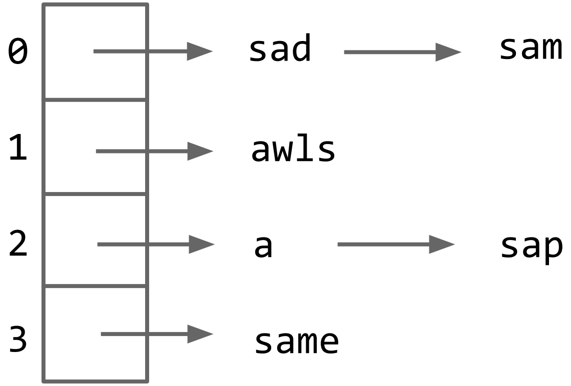
Sets of Strings

Suppose we have a set containing “sam”, “sad”, “sap”, “same”, “a”, and “awls”.

- Below, we see the BST and Hash Table representation.



BST

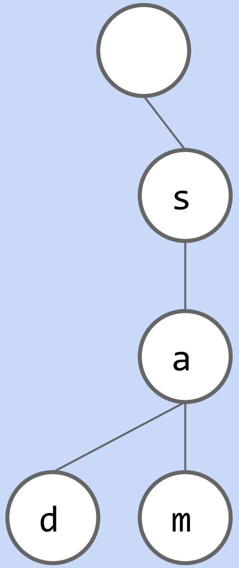


Hash Table

Tries: Each Node Stores One Character

For String keys, we can use a “Trie”. Key ideas:

- Every node stores only one letter.
- Nodes can be shared by multiple keys.

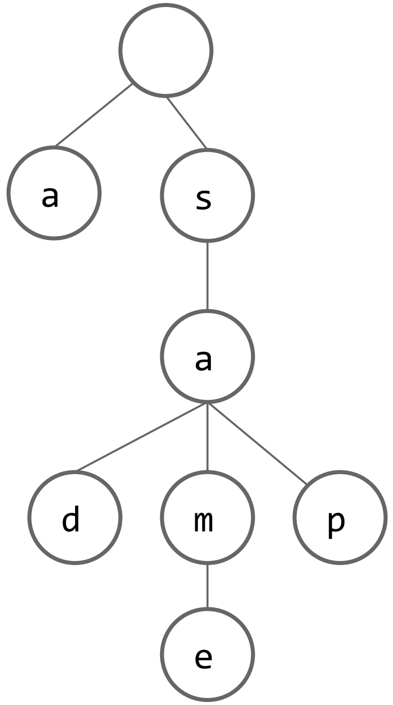


Above, we show the results of adding “sam” and sad”. Use your intuition to try to insert the remaining items “sap”, “same”, “a”, and “awls”.

Tries: Each Node Stores One Character

For String keys, we can use a "Trie". Key ideas:

- Every node stores only one letter.
- Nodes can be shared by multiple keys.

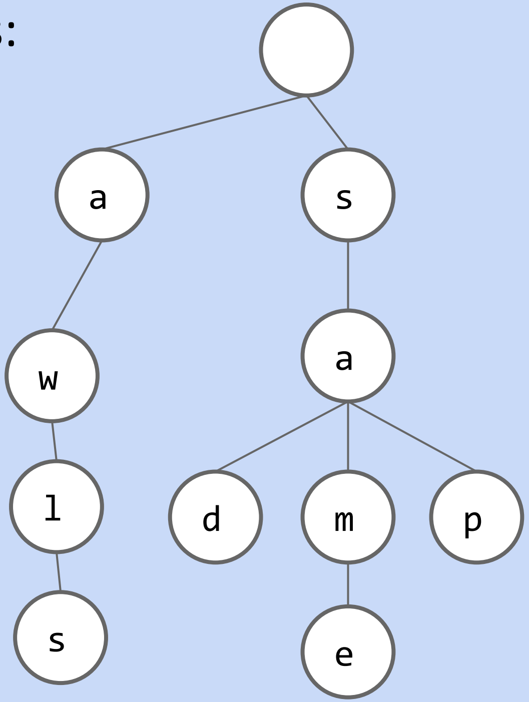


Above, we show the results of adding "sam" and sad". Use your intuition to try to insert the remaining items "sap", "same", "a", and "awls".

Tries: Each Node Stores One Character

For String keys, we can use a “Trie”. Key ideas:

- Every node stores only one letter.
- Nodes can be shared by multiple keys.

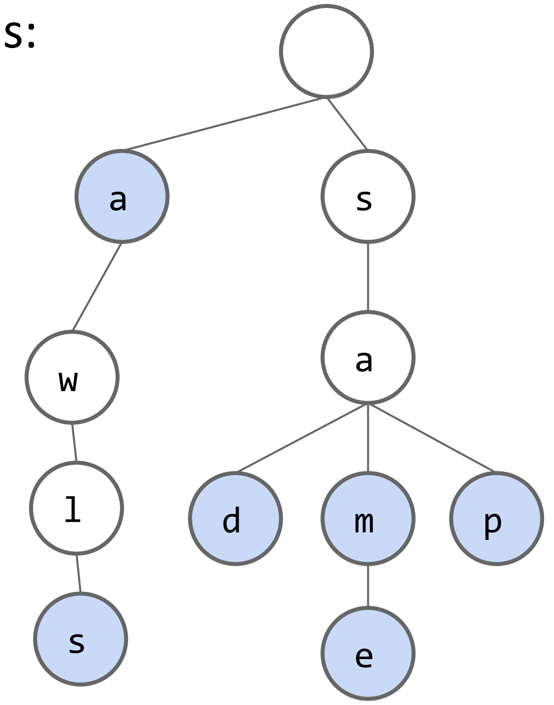


Try to figure out a way to make it clear that our set contains “sam”, “sad”, “sap”, “same”, “a”, and “awls”, but not “aw”, “awl”, “sa”, etc.

Tries: Each Node Stores One Character

For String keys, we can use a “Trie”. Key ideas:

- Every node stores only one letter.
- Nodes can be shared by multiple keys.

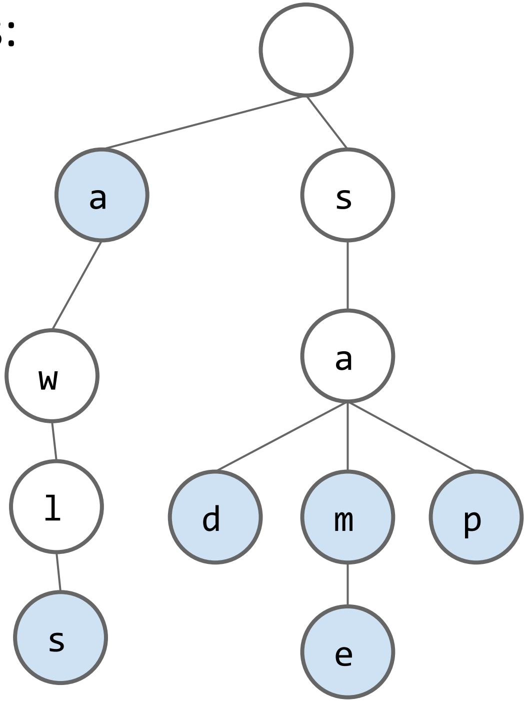


Try to figure out a way to make it clear that our set contains “sam”, “sad”, “sap”, “same”, “a”, and “awls”, but not “aw”, “awl”, “sa”, etc.

Tries: Each Node Stores One Character

For String keys, we can use a "Trie". Key ideas:

- Every node stores only one letter.
- Nodes can be shared by multiple keys.



Try to figure out a way to make it clear that our set contains "sam", "sad", "sap", "same", "a", and "awls", but not "aw", "awl", "sa", etc.

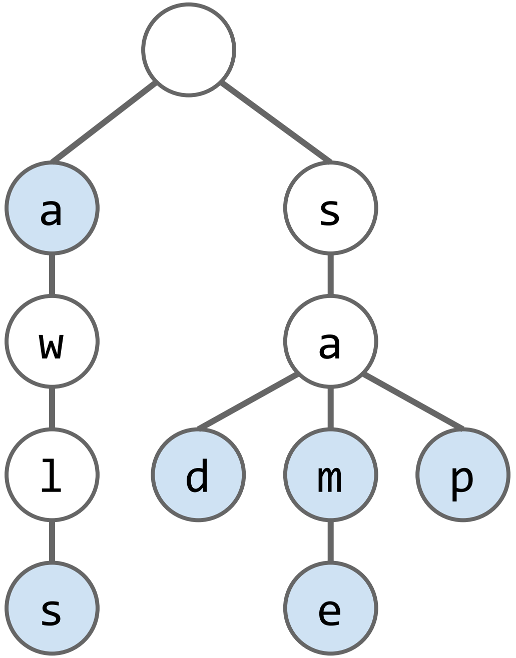
Tries: Search Hits and Misses

Suppose we insert “sam”, “sad”, “sap”, “same”, “a”, and “awls”.

- contains(“sam”): true, blue node
- contains(“sa”): false, white node
- contains(“a”): true, blue node
- contains(“saq”): false, fell off tree

“hit”

“miss”

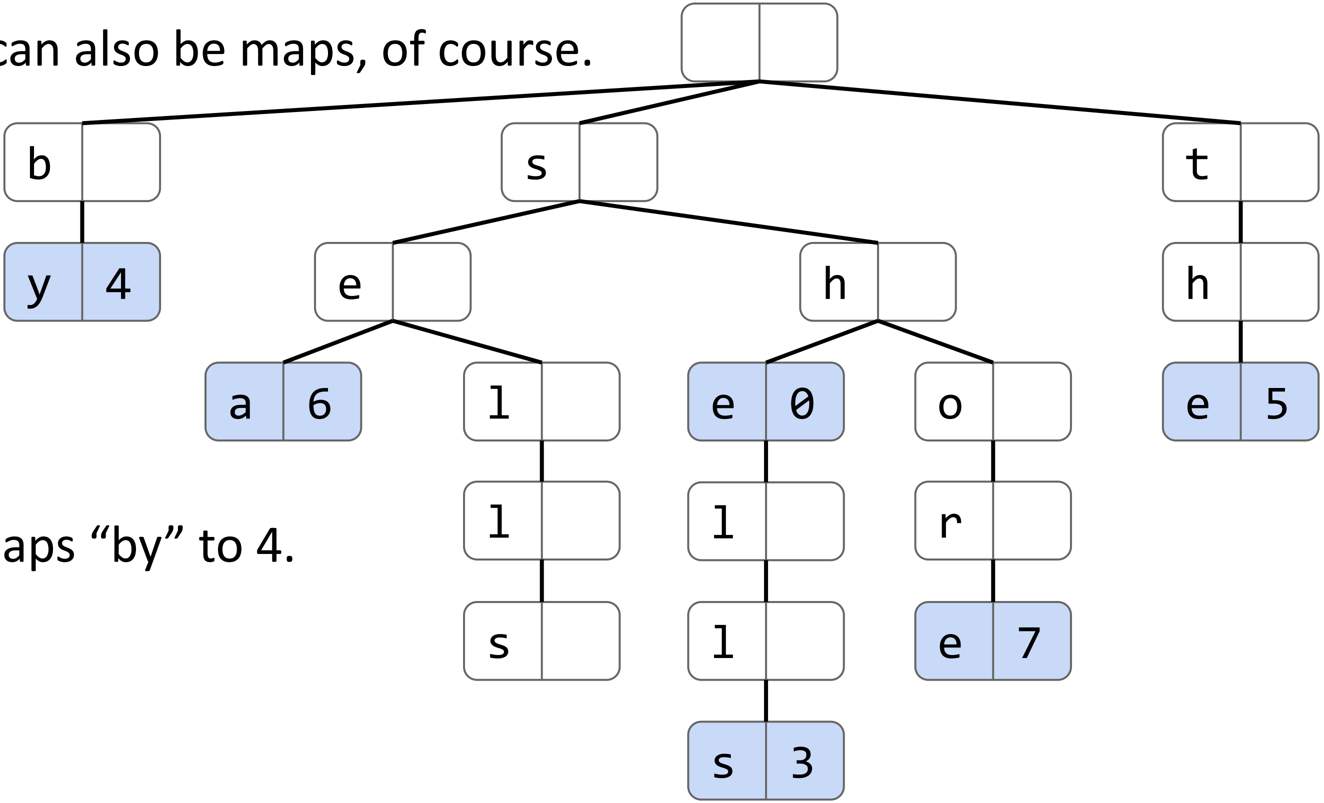


Two ways to have a search “miss”:

- If the final node is white.
- If we fall off the tree, e.g. contains(“sax”).

Trie Maps

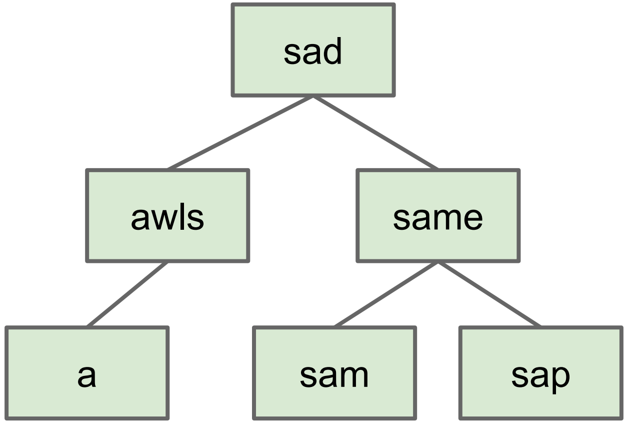
Tries can also be maps, of course.



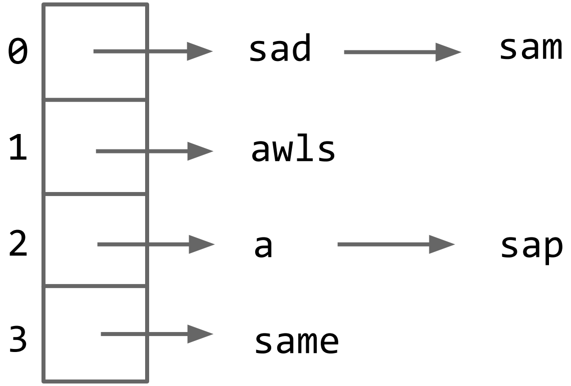
e.g. maps “by” to 4.

For an animated demo of the creation of this map, see this demo from our optional Algorithms textbook.

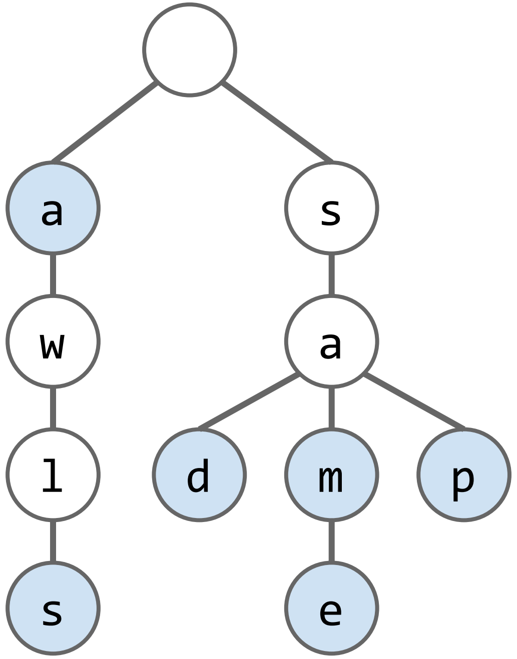
Tries: A Digit-by-Digit Set Representation



BST



HashSet

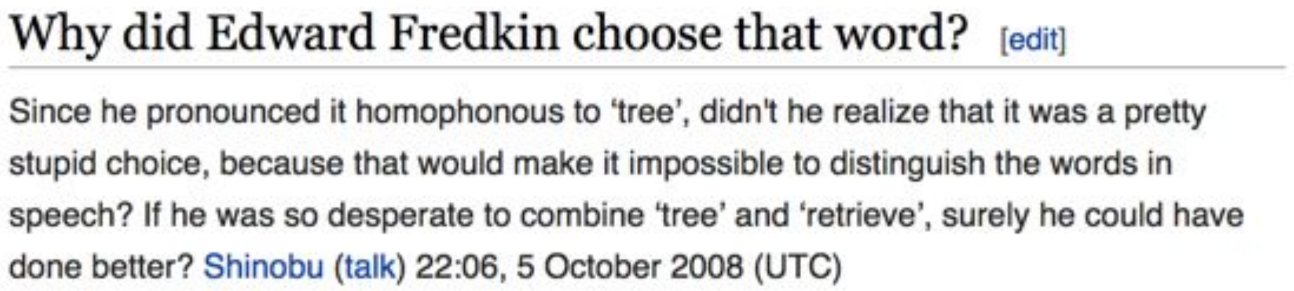


Trie

Tries

Trie:

- Short for Re**trie**val Tree.
- Inventor Edward Fredkin suggested it should be pronounced “tree”, but almost everyone pronounces it like “try”.



Why did Edward Fredkin choose that word? [edit]

Since he pronounced it homophonous to ‘tree’, didn't he realize that it was a pretty stupid choice, because that would make it impossible to distinguish the words in speech? If he was so desperate to combine ‘tree’ and ‘retrieve’, surely he could have done better? Shinobu (talk) 22:06, 5 October 2008 (UTC)

Trie Implementation and Performance

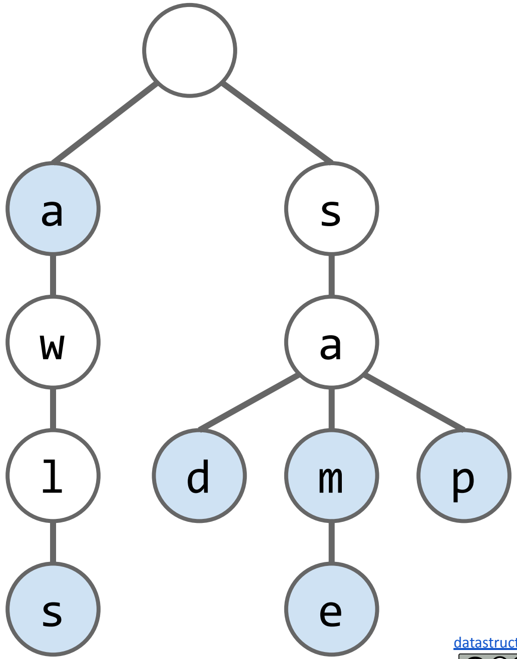
Very Basic Trie Implementation

The first approach might look something like the code below.

- Each node stores a letter, a map from c to all child nodes, and a color.

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

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

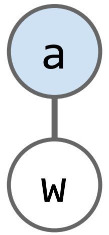
Since we know our keys are characters, can use a DataIndexedCharMap.

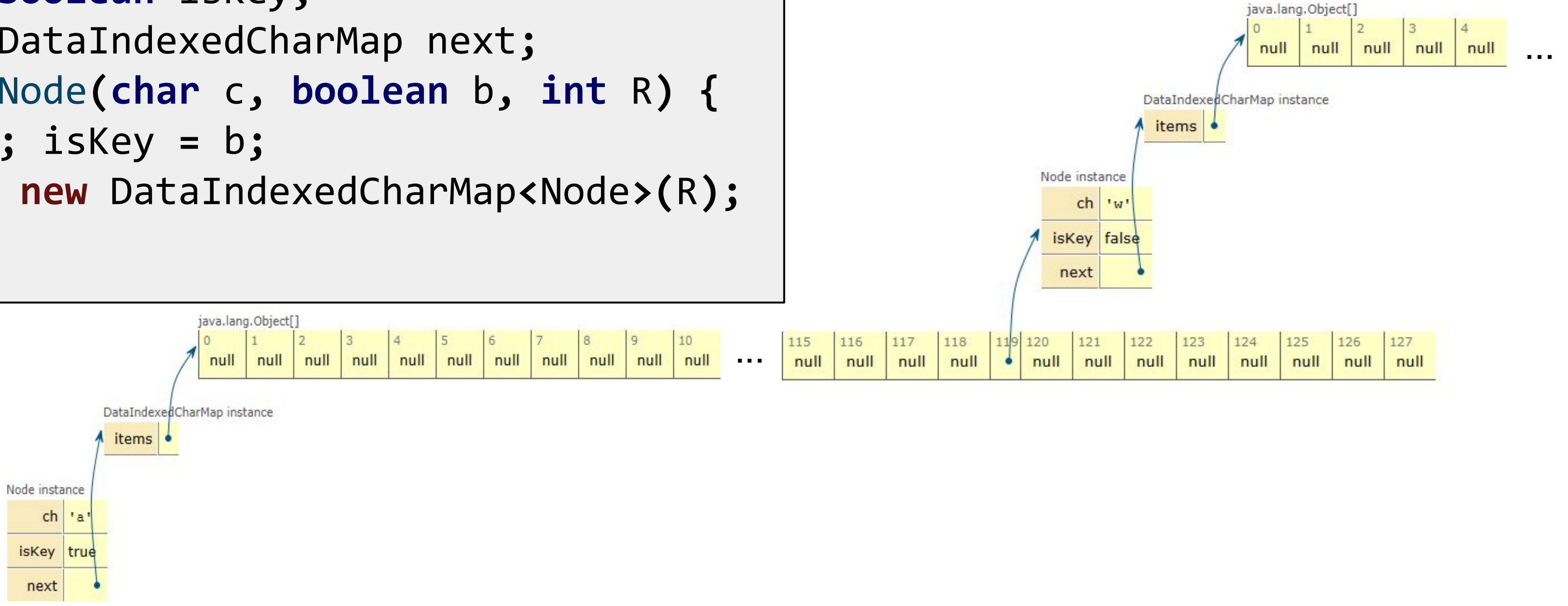


Zooming in On a Node

Each DataIndexedCharMap is an array of 128 possible links, mostly null.

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

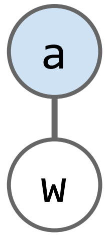


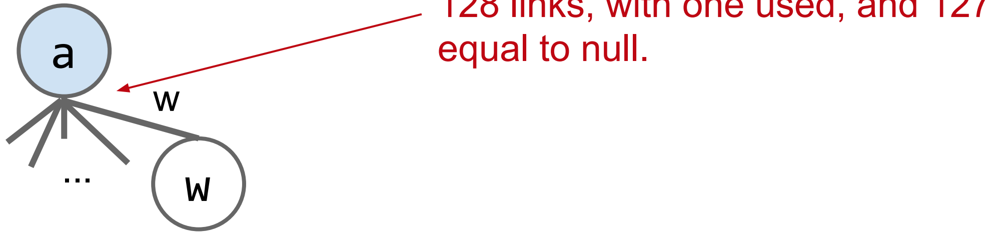


Zooming in On a Node

Better drawing of a DataIndexedCharMap based trie is shown to the right.

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



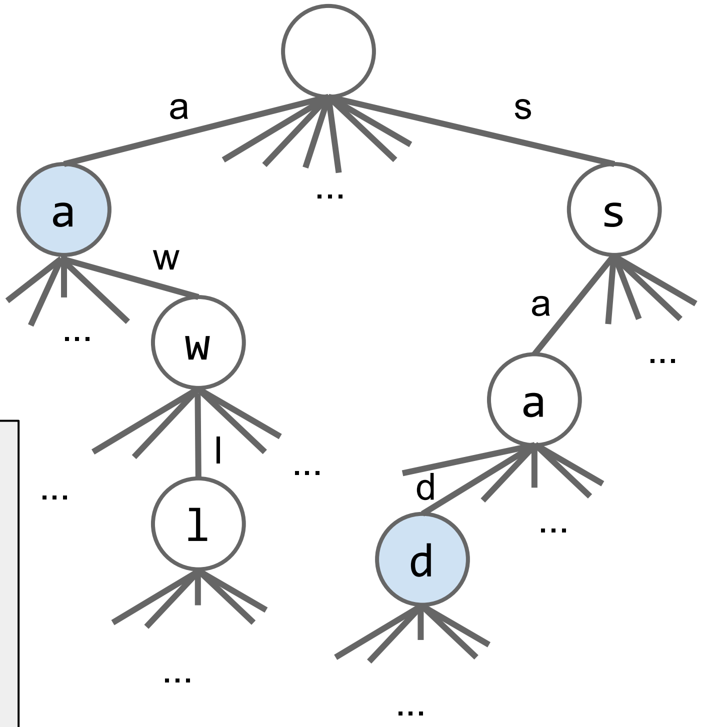


Very Basic Trie Implementation

If we use a DataIndexedCharMap to track children, every node has R links.

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

```java
public class DataIndexedCharMap<V> {
   private V[] items;
   public DataIndexedCharMap(int R) {
        items = (V[]) new Object[R];
   }
   ...
}
```

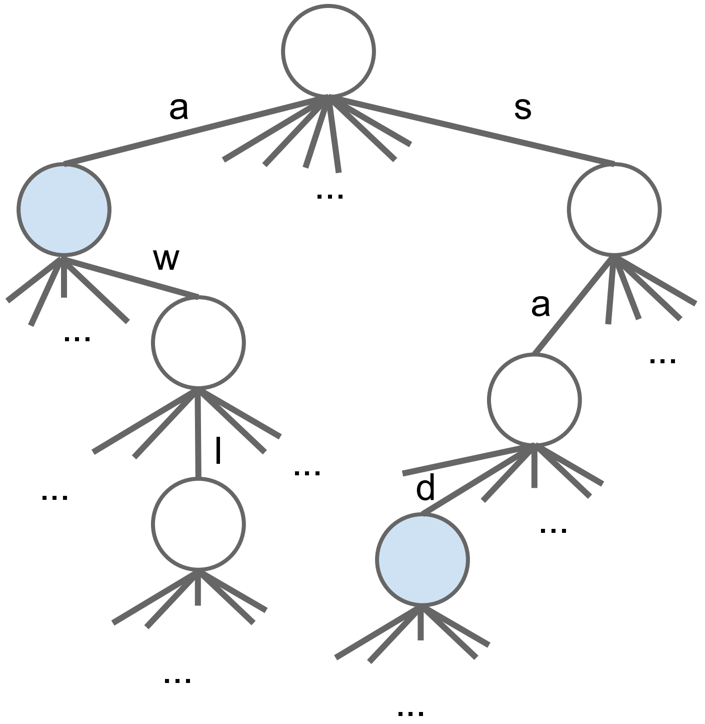


Very Basic Trie Implementation

Observation: The letter stored inside each node is actually redundant.

- Can remove from the representation and things will work fine.

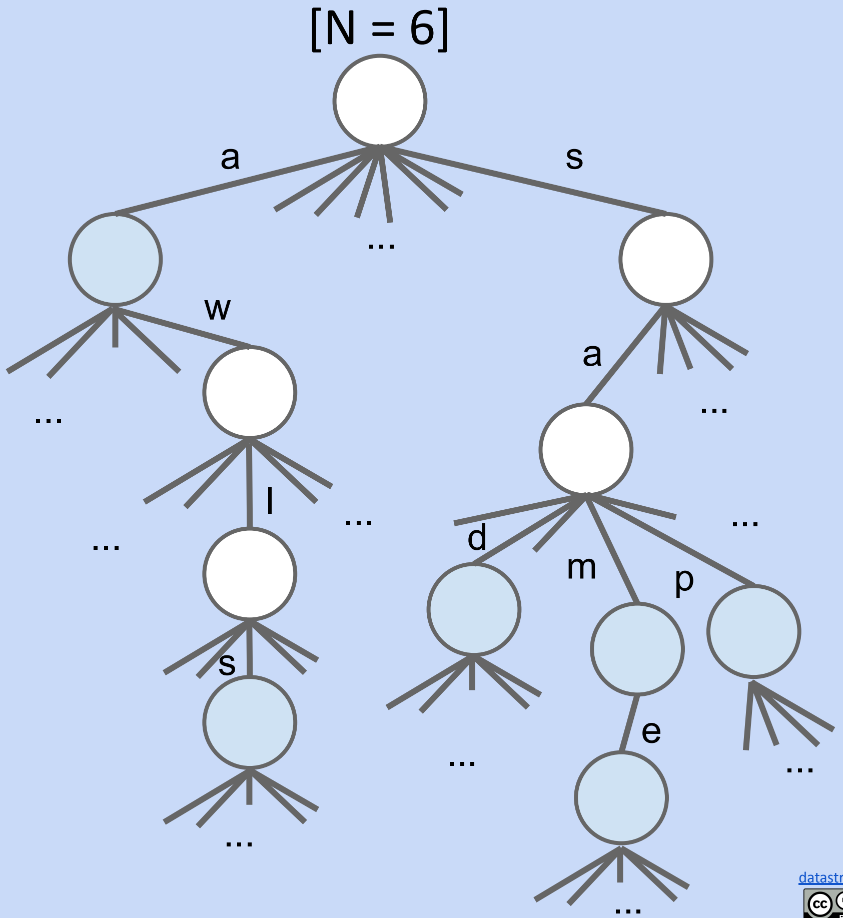
```
public class TrieSet {
  private static final int R = 128; // ASCII
  private Node root; // root of trie

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



Trie Performance in Terms of N

Given a Trie with N keys. What is the:

- Add runtime?
- Contains runtime?



Trie Performance in Terms of N

Given a Trie with N keys. What is the:

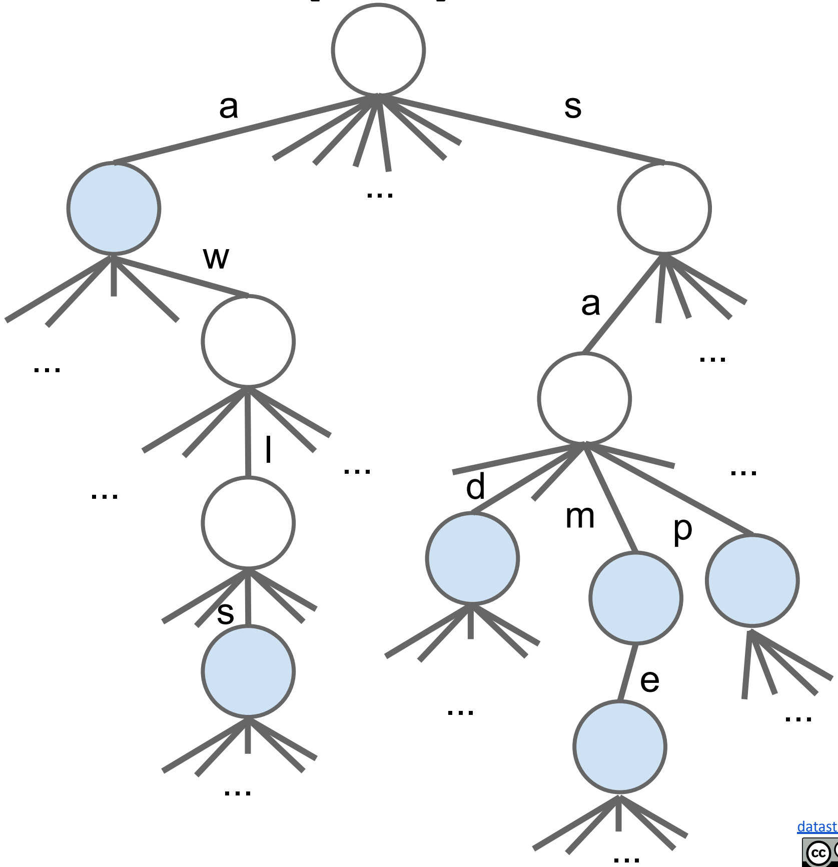
- Add runtime? Θ(1)
- Contains runtime? Θ(1)

Runtimes independent of number of keys!

Or in terms of L, the length of the key:

- Add: Θ(L)
- Contains: O(L)

[N = 6]



Trie Performance in Terms of N

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

Runtimes treating length of keys as a constant

	key type	get(x)	add(x)
Balanced BST	comparable	Θ(log N)	Θ(log N)
RSC Hash Table	hashable	$\Theta(1)^{\dagger}$	$\Theta(1)^{*\dagger}$
data indexed array	chars	Θ(1)	Θ(1)
Tries (Data Indexed Char Map)	Strings	Θ(1)	Θ(1)

*: Indicates "on average".
†: Assuming items are evenly spread.

One downside of the DataIndexedCharMap-based Trie is the huge memory cost of storing R links per node.

- Wasteful because most links are unused in real world usage.

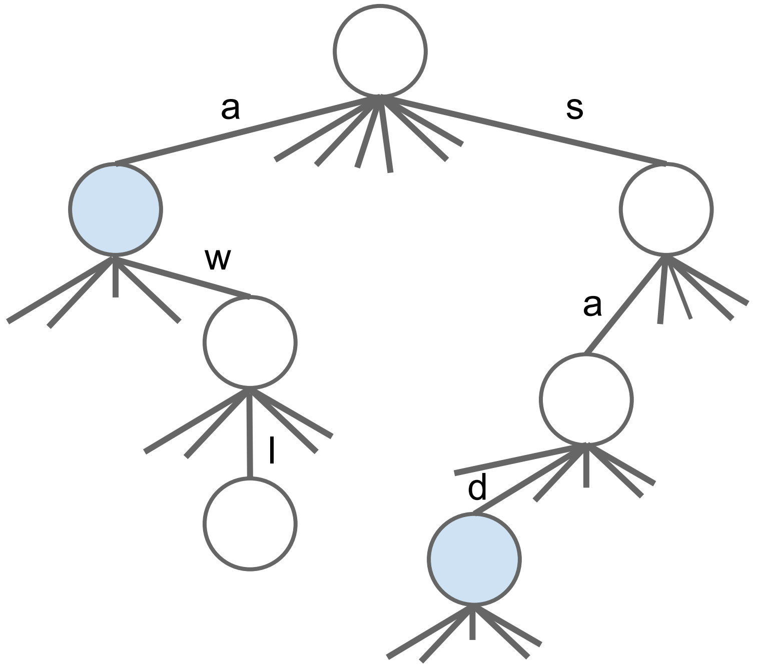
Alternate Child Tracking Strategies

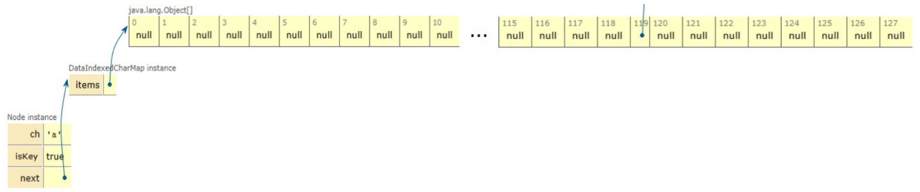
Trie Performance in Terms of N

Using a DataIndexedCharMap is very memory hungry.

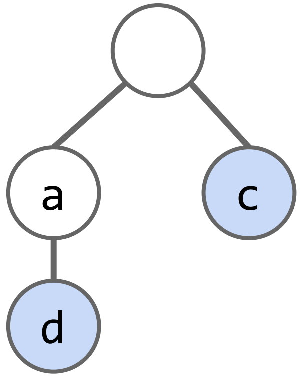
- Every node has to store R links, most of which are null.

```
private static class Node {
  private boolean isKey;
  private DataIndexedCharMap next;

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



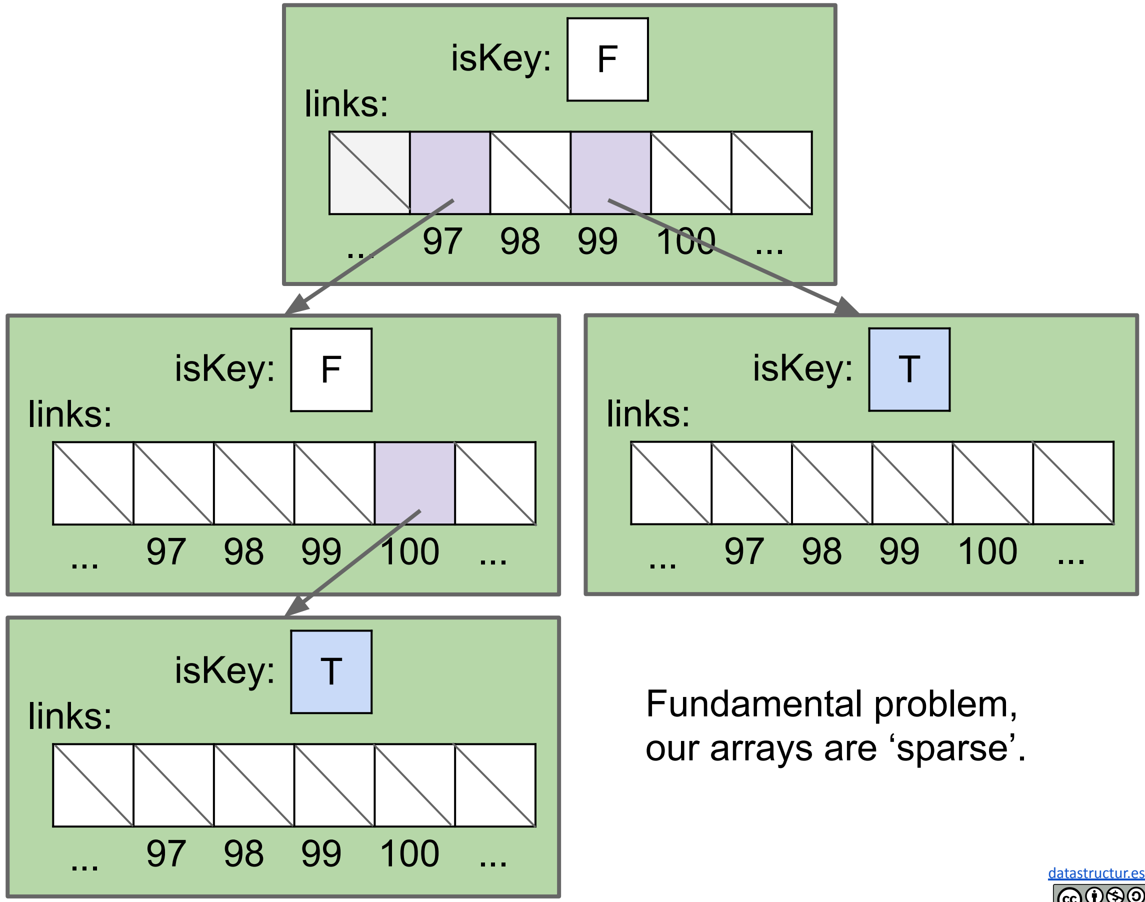


The DataIndexedCharMap Trie



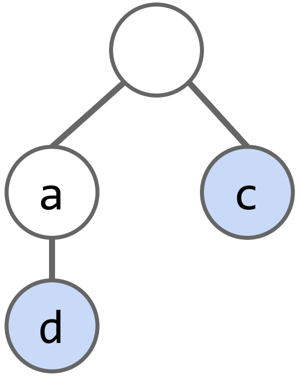
Can use ANY kind of map from character to node, e.g.

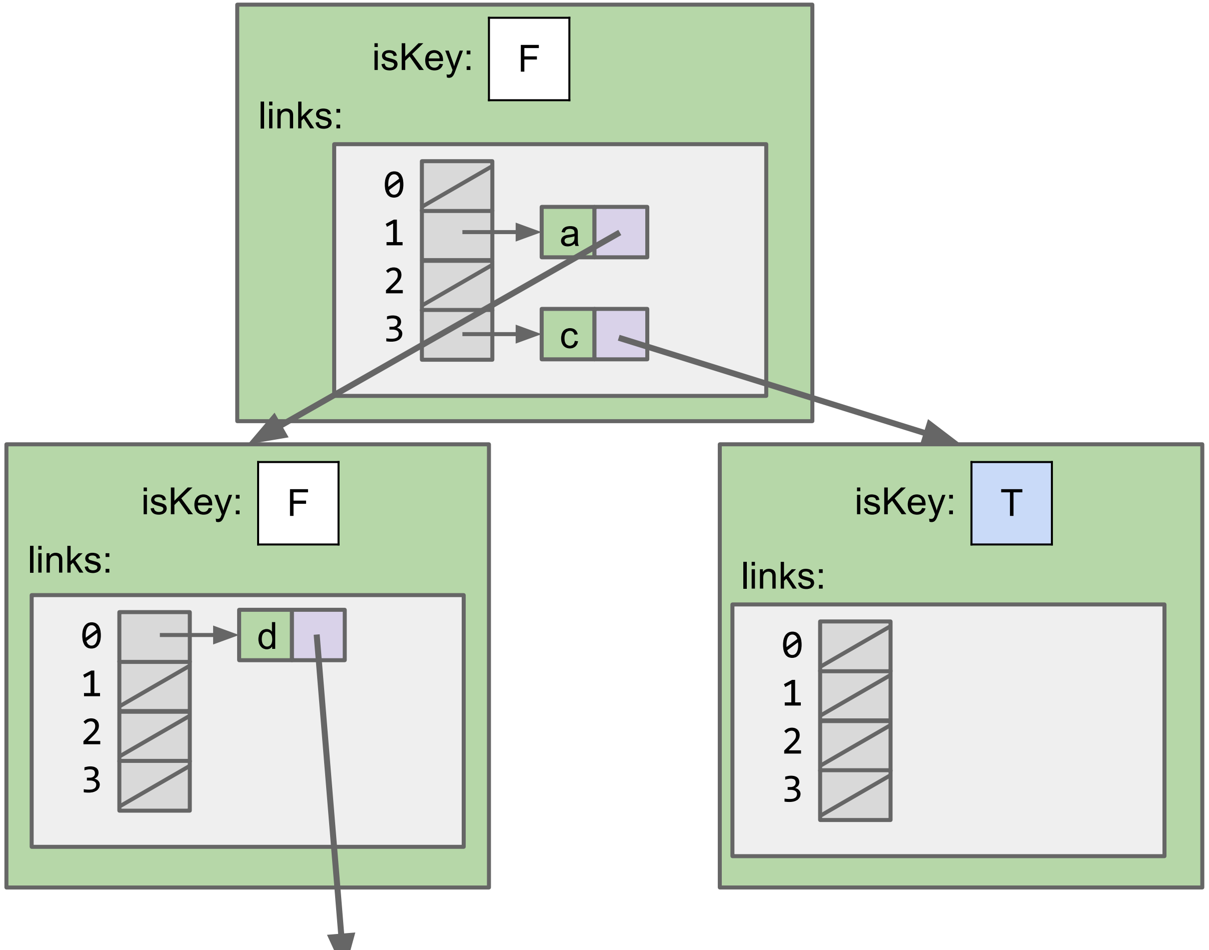
- BST
- Hash Table



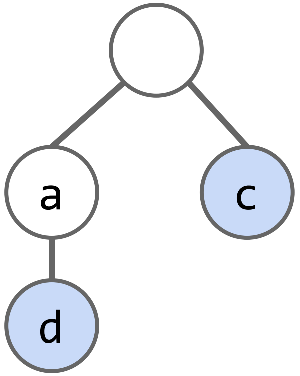
Fundamental problem, our arrays are ‘sparse’.

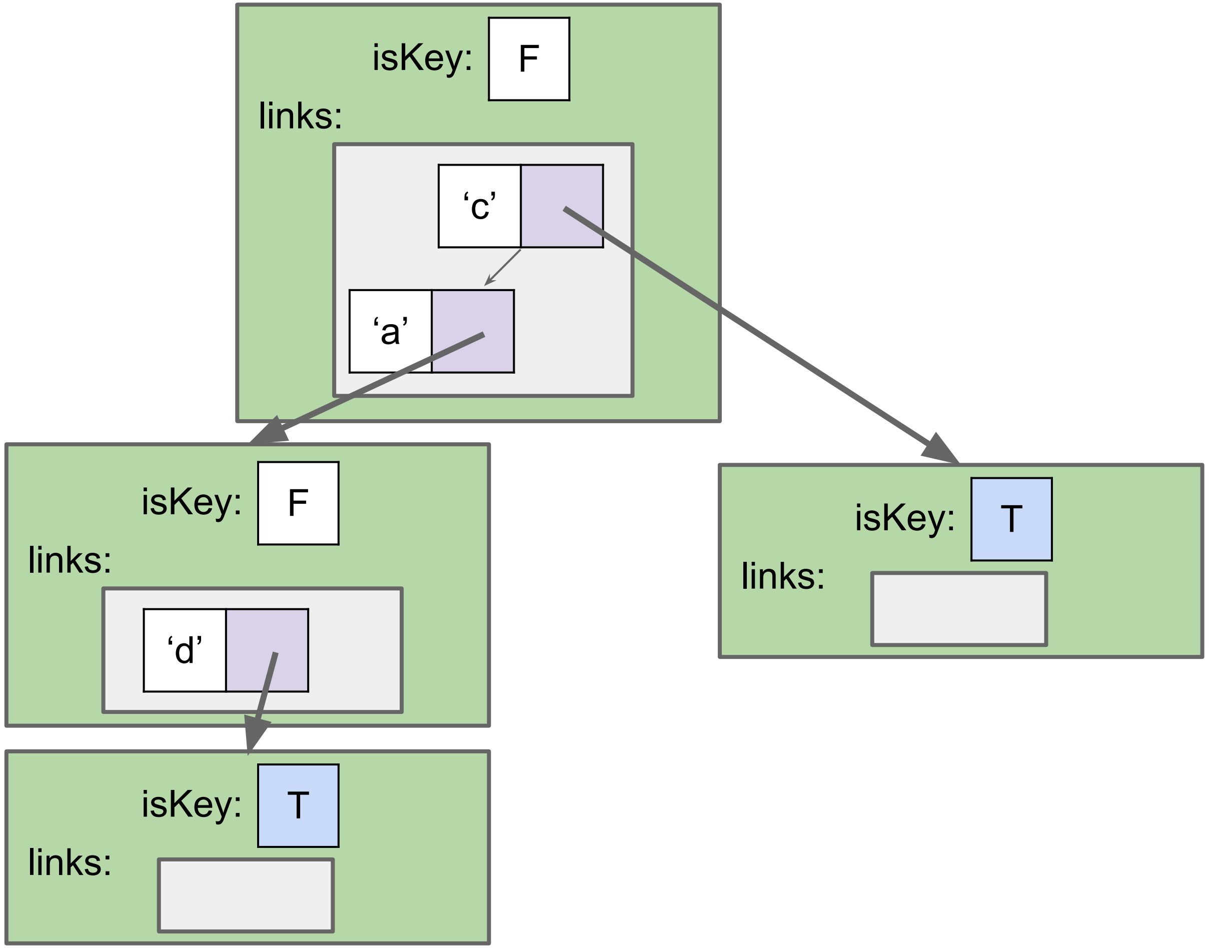
Alternate Idea #1: The Hash-Table Based Trie





Alternate Idea #2: The BST-Based Trie



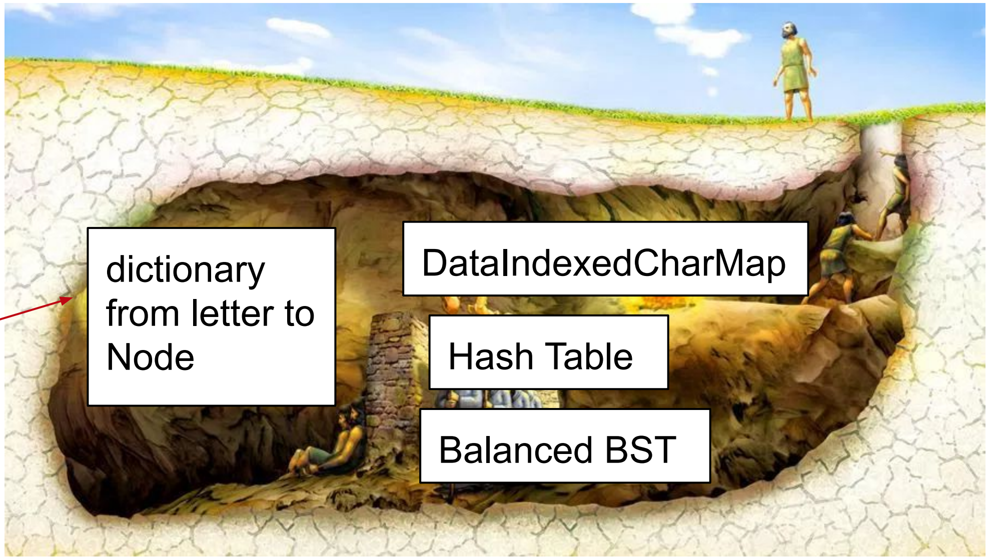


The Three Trie Implementations

When we implement a Trie, we have to pick a map to our children

- DataIndexedCharMap: Very fast, but memory hungry.
- Hash Table: Almost as fast, uses less memory.
- Balanced BST: A little slower than Hash Table, uses similar amount of memory?

`this.next`



Performance of the DataIndexedCharMap, BST, and Hash Table Trie

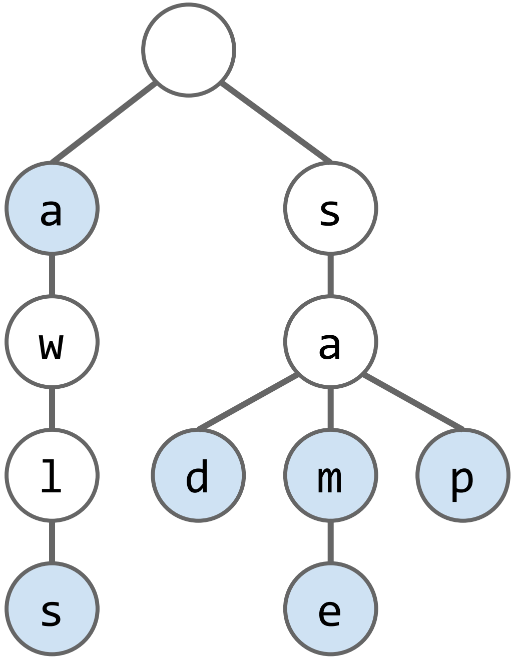
Using a BST or a Hash Table to store links to children will usually use less memory.

- DataIndexedCharMap: 128 links per node.
- BST: C links per node, where C is the number of children.
- Hash Table: C links per node.
- Note: Cost per link is higher in BST and Hash Table.

Using a BST or a Hash Table will take slightly more time.

- DataIndexedCharMap is Θ(1).
- BST is O(log R), where R is size of alphabet.
- Hash Table is O(R), where R is size of alphabet.

Since R is fixed (e.g. 128), can think of all 3 as Θ(1).



Trie Performance in Terms of N

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

- Using BST or Hash Table will be slightly slower, but more memory efficient.
- Would have to do computational experiments to see which is best for your application.

Runtimes treating length of keys as a constant

	key type	get(x)	add(x)
Balanced BST	comparable	Θ(log N)	Θ(log N)
RSC Hash Table	hashable	Θ(1)†	Θ(1)*†
data indexed array	chars	Θ(1)	Θ(1)
Tries (BST, Hash Table, Data Indexed Char Map)	Strings	Θ(1)	Θ(1)

*: Indicates "on average".
†: Assuming items are evenly spread.

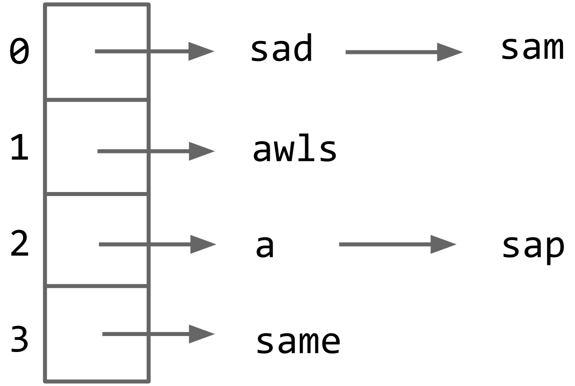
… but where Tries really shine is their efficiency with special string operations!

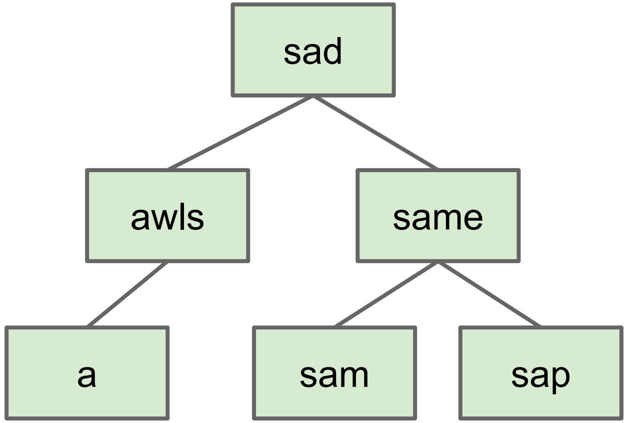
Trie String Operations

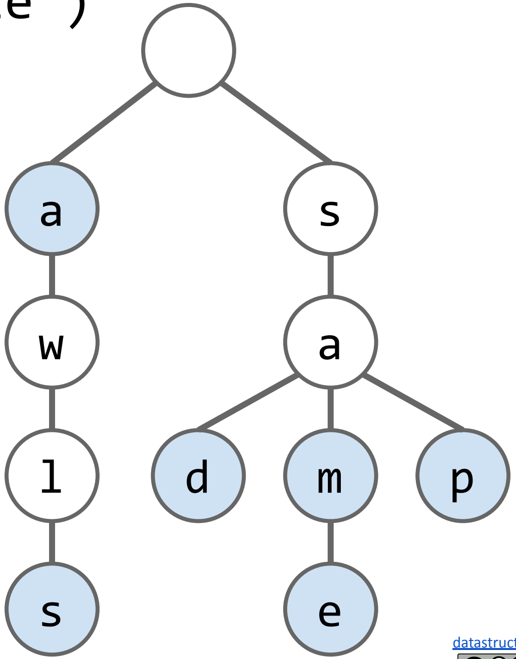
String Specific Operations

Theoretical asymptotic speed improvement is nice. But the **main appeal of tries** is their ability to efficiently support string specific operations like **prefix matching**.

- Finding all keys that match a given prefix: `keysWithPrefix("sa")`
- Finding longest prefix of: `longestPrefixOf("sample")`

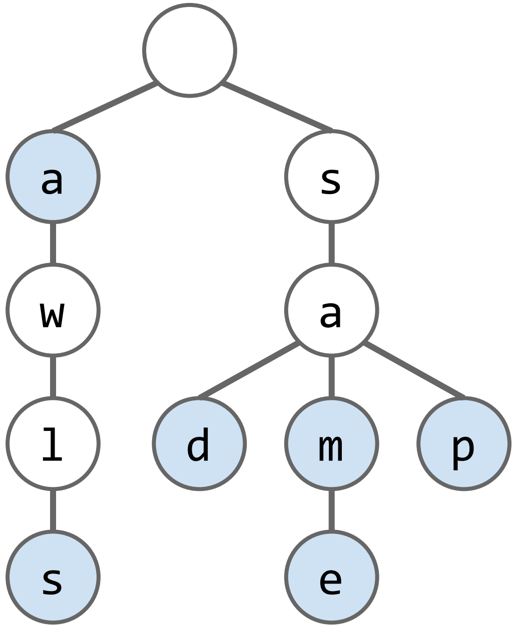






Prefix Matching Operations

Theoretical asymptotic speed improvement is nice. But the **main appeal of tries** is their ability to efficiently support string specific operations like **prefix matching**.



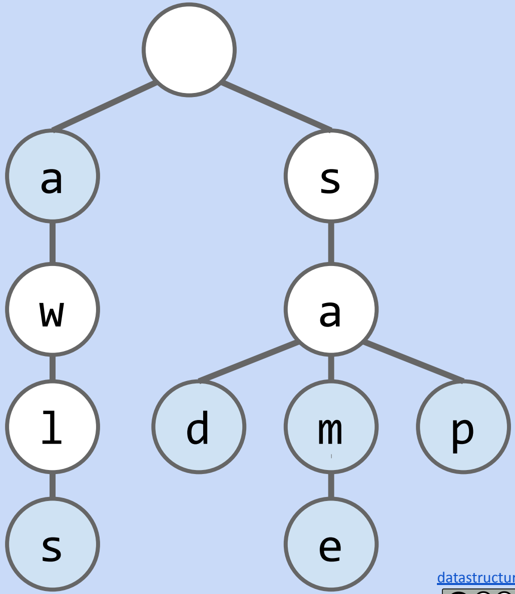
Examples:

- Finding the longest prefix of a string: `longestPrefixOf(“sample”)`
  - Result: `sam`
- Finding all keys that match a given prefix: `keysWithPrefix(“sa”)`
  - Result: `[sad, sam, same, sap]`

Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

`collect()` returns [`“a”, “awls”, “sad”, “sam”, “same”, “sap”`]



Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

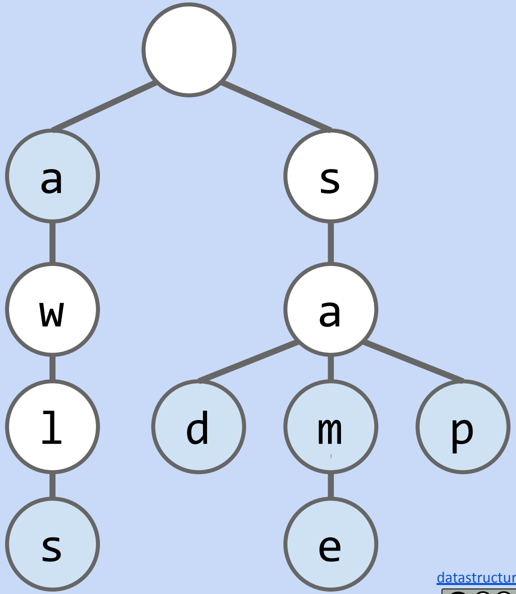
`collect()` returns [“a”, “awls”, “sad”, “sam”, “same”, “sap”]

`collect():`

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

Create `colHelp`.

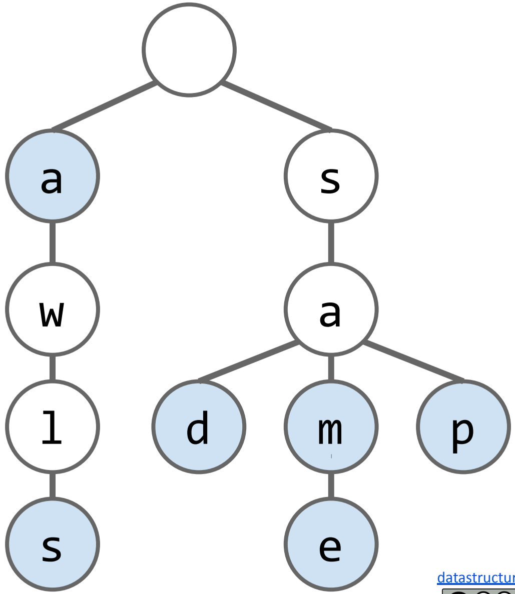
- `colHelp(String s, List<String> x, Node n)`



Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

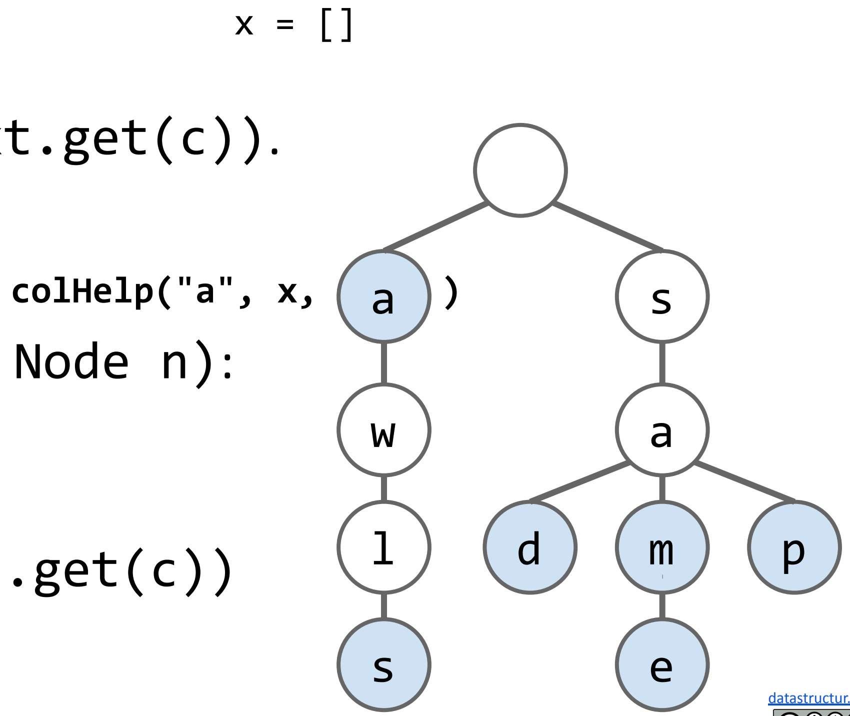
```
collect():
  ● Create an empty list of results x.
  ● For character c in root.next.keys():
      ○ Call colHelp("c", x, root.next.get(c)).
  ● Return x.

colHelp(String s, List<String> x, Node n):
  ● If n.isKey, then x.add(s).
  ● For character c in n.next.keys():
      ○ Call colHelp(s + c, x, n.next.get(c))
```



Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

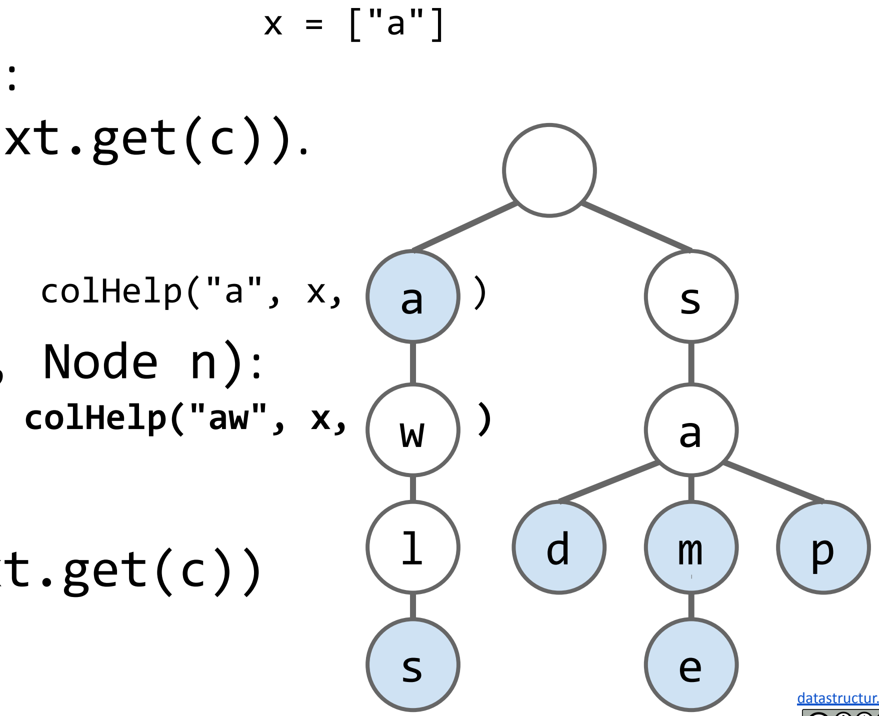
```
collect():
  ● Create an empty list of results x.
  ● For character c in root.next.keys():
      ○ Call colHelp("c", x, root.next.get(c)).
  ● Return x.

colHelp(String s, List<String> x, Node n):
  ● If n.isKey, then x.add(s).
  ● For character c in n.next.keys():
      ○ Call colHelp(s + c, x, n.next.get(c))
```



Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

```
collect():
    • Create an empty list of results x.
    • For character c in root.next.keys():
        ○ Call colHelp("c", x, root.next.get(c)).
    • Return x.

colHelp(String s, List<String> x, Node n):
    • If n.isKey, then x.add(s).
    • For character c in n.next.keys():
        ○ Call colHelp(s + c, x, n.next.get(c))
```

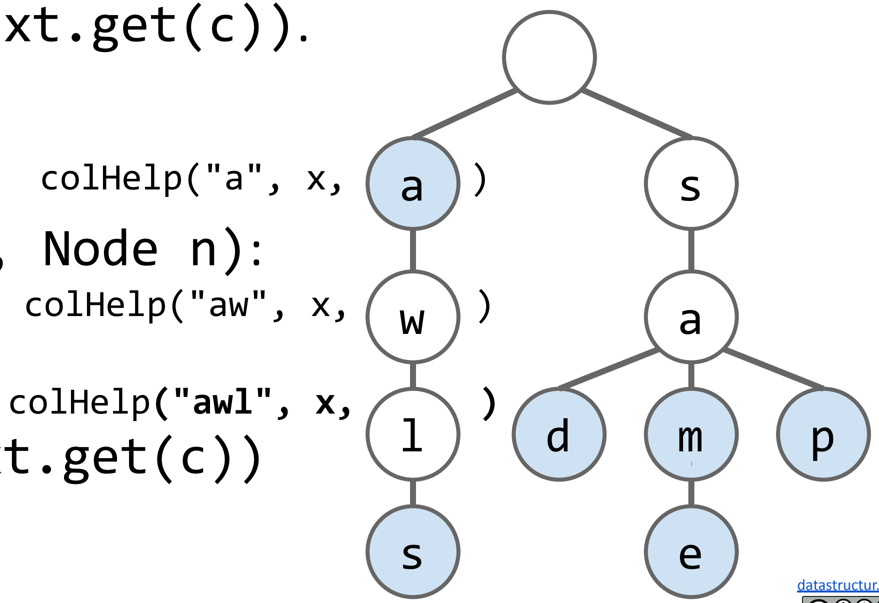


Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

```
collect():
  ● Create an empty list of results x.
  ● For character c in root.next.keys():
      ○ Call colHelp("c", x, root.next.get(c)).
  ● Return x.

colHelp(String s, List<String> x, Node n):
  ● If n.isKey, then x.add(s).
  ● For character c in n.next.keys():
      ○ Call colHelp(s + c, x, n.next.get(c))
```

x = ["a"]

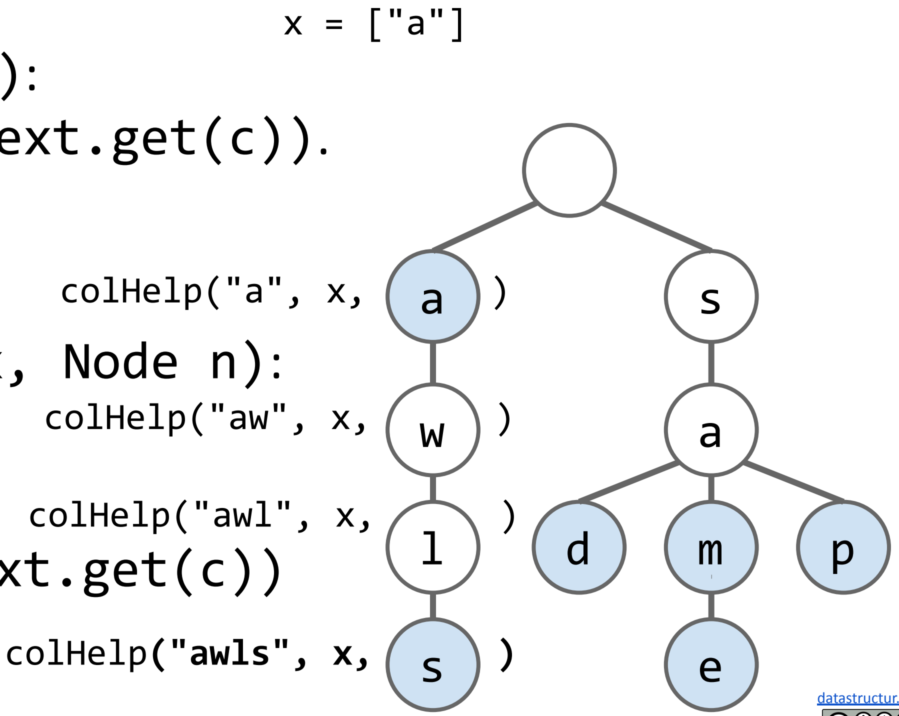


Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

```
collect():
    ● Create an empty list of results x.
    ● For character c in root.next.keys():
        ○ Call colHelp("c", x, root.next.get(c)).
    ● Return x.
```

```
colHelp(String s, List<String> x, Node n):
    ● If n.isKey, then x.add(s).
    ● For character c in n.next.keys():
        ○ Call colHelp(s + c, x, n.next.get(c))
```

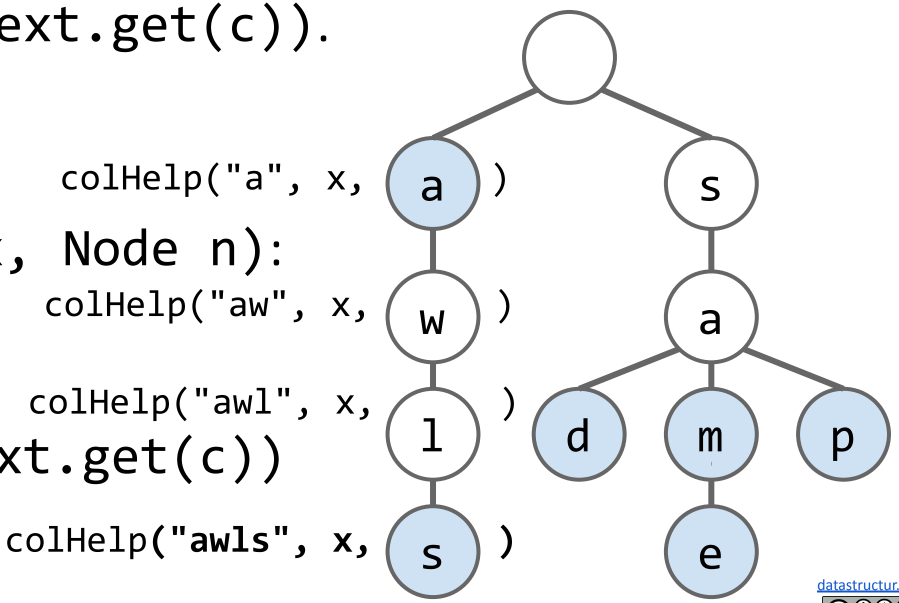


Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

```
collect():
    ● Create an empty list of results x.
    ● For character c in root.next.keys():
        ○ Call colHelp("c", x, root.next.get(c)).
    ● Return x.

colHelp(String s, List<String> x, Node n):
    ● If n.isKey, then x.add(s).
    ● For character c in n.next.keys():
        ○ Call colHelp(s + c, x, n.next.get(c))
```

`x = ["a", "awls"]`

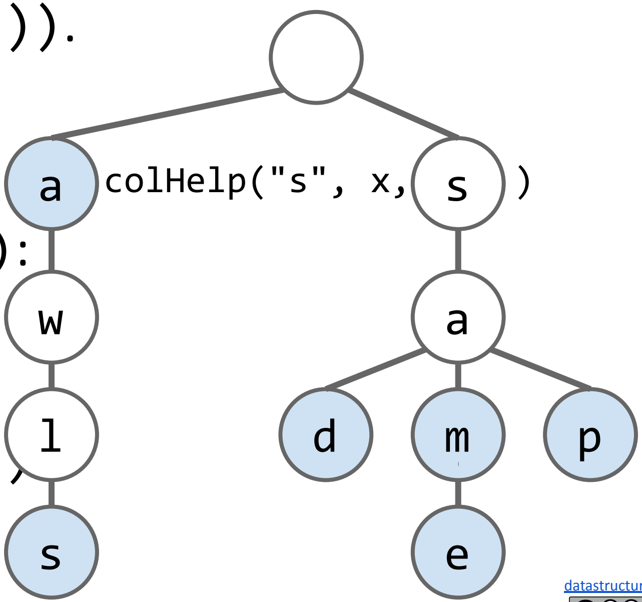


Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

```
collect():
    ● Create an empty list of results x.
    ● For character c in root.next.keys():
        ○ Call colHelp(“c”, x, root.next.get(c)).
    ● Return x.

colHelp(String s, List<String> x, Node n):
    ● If n.isKey, then x.add(s).
    ● For character c in n.next.keys():
        ○ Call colHelp(s + c, x, n.next.get(c))
```

`x = ["a", "awls"]`

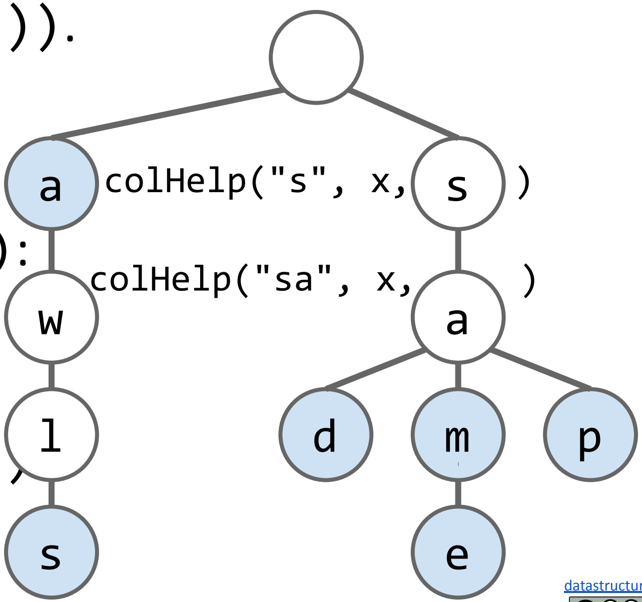


Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

```
collect():
  ● Create an empty list of results x.
  ● For character c in root.next.keys():
      ○ Call colHelp("c", x, root.next.get(c)).
  ● Return x.

colHelp(String s, List<String> x, Node n):
  ● If n.isKey, then x.add(s).
  ● For character c in n.next.keys():
      ○ Call colHelp(s + c, x, n.next.get(c)).
```

x = ["a", "awls"]



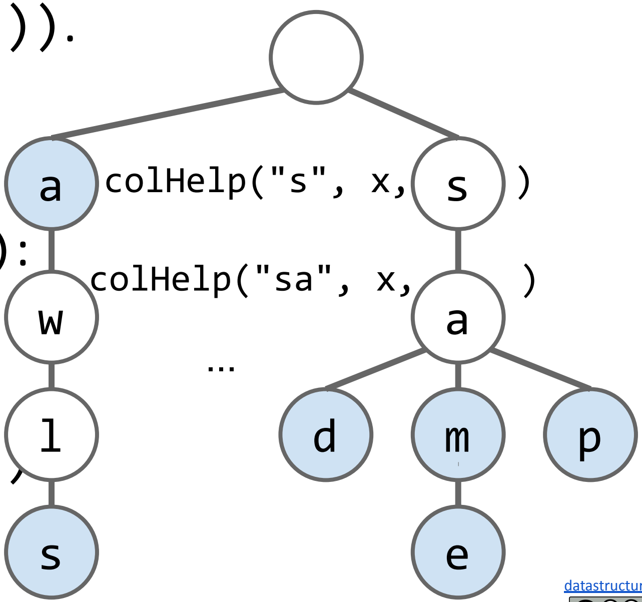
Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

```
collect():
    • Create an empty list of results x.
    • For character c in root.next.keys():
        ○ Call colHelp(“c”, x, root.next.get(c)).
    • Return x.
```

```
colHelp(String s, List<String> x, Node n):
    • If n.isKey, then x.add(s).
    • For character c in n.next.keys():
        ○ Call colHelp(s + c, x, n.next.get(c))
```

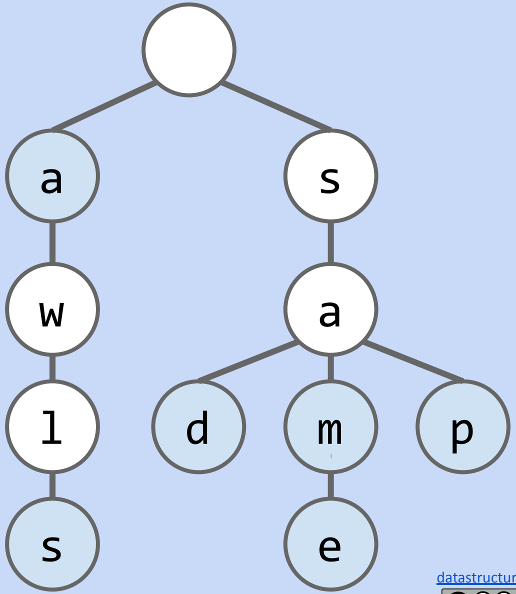
```
x = ["a", "awls", "sad",
     "sam", "same", "sap"]
```



Usages of Tries

Challenge: Give an algorithm for `keysWithPrefix`.

- Example: `keysWithPrefix(“sa”)` is [`“sad”, “sam”, “same”, “sap”`].



Usages of Tries

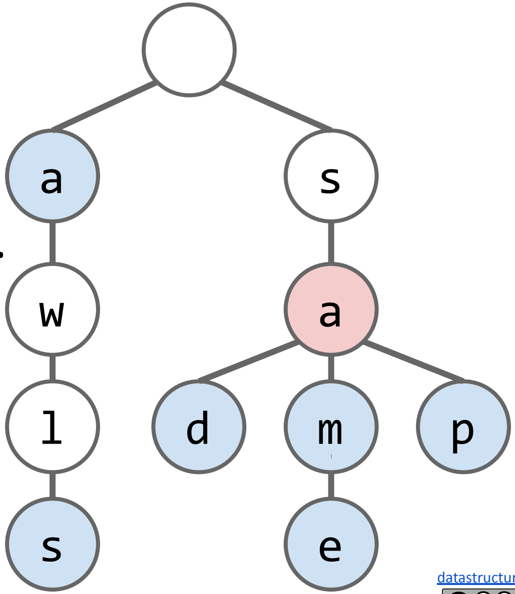
Challenge: Give an algorithm for `keysWithPrefix`.

- Example: `keysWithPrefix("sa")` is `["sad", "sam", "same", "sap"]`.

Algorithm:

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

Another common operation: `LongestPrefixOf`. See lab.



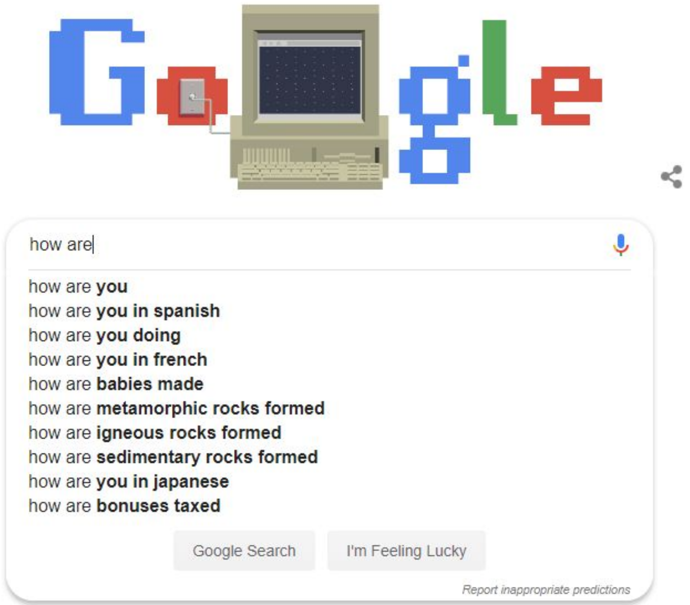
Autocomplete

The Autocomplete Problem

Example, when I type "how are" into Google, I get 10 results, shown to the right.

One way to do this is to create a Trie based map from strings to values

- Value represents how important Google thinks that string is.
- Can store billions of strings efficiently since they share nodes.
- When a user types in a string "hello", we:
  - Call `keysWithPrefix("hello")`.
  - Return the 10 strings with the highest value.



how are

how are **you**
how are **you in spanish**
how are **you doing**
how are **you in french**
how are **babies made**
how are **metamorphic rocks formed**
how are **igneous rocks formed**
how are **sedimentary rocks formed**
how are **you in japanese**
how are **bonuses taxed**

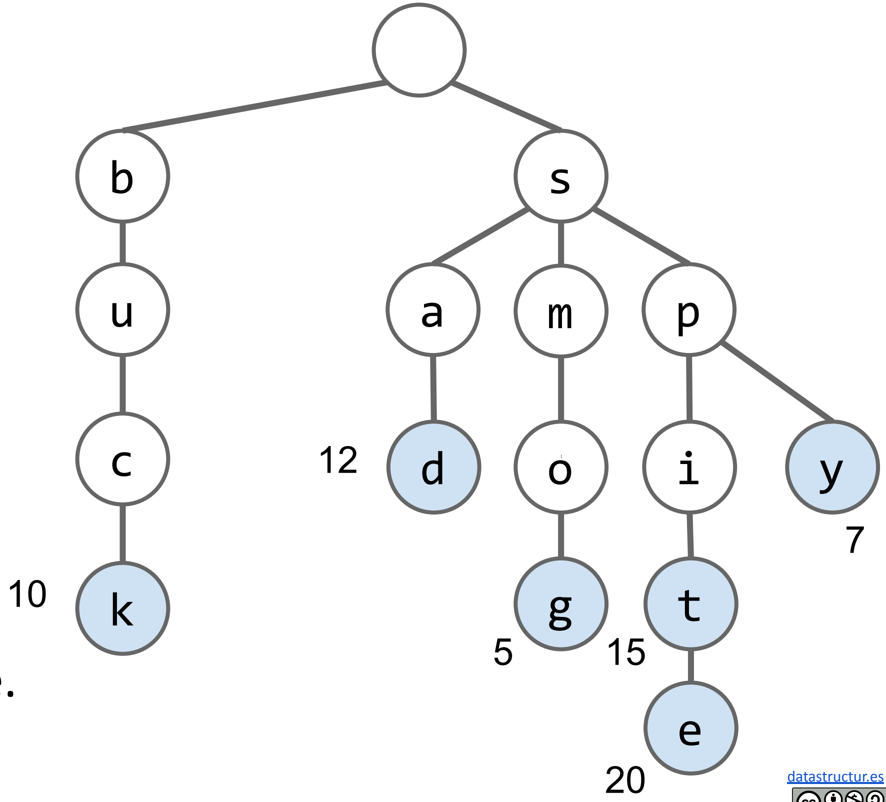
Autocomplete Example, for Top Three Matches

Suppose we have six strings with values shown below:

- buck: 10
- sad: 12
- smog: 5
- spit: 15
- spite: 20
- spy: 7

If the user types “s”, we:

- Call `keysWithPrefix(“s”)`.
  - sad, smog, spit, spite, spy
- Return the three keys with highest value.
  - spit, spite, sad



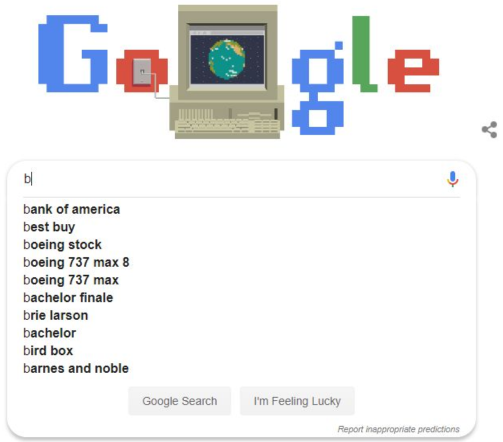
The Autocomplete Problem

One way to do this is to create a Trie based Dictionary that maps strings to values.

- When a user types in a string hello, we:
  - Call `keysWithPrefix(“hello”)`.
  - Return the ten strings with the highest value.

The approach above has one major flaw. If we enter a short string, the number of keys with the appropriate prefix will be too big.

- We are collecting billions of results only to keep 10!
- This is extremely inefficient.

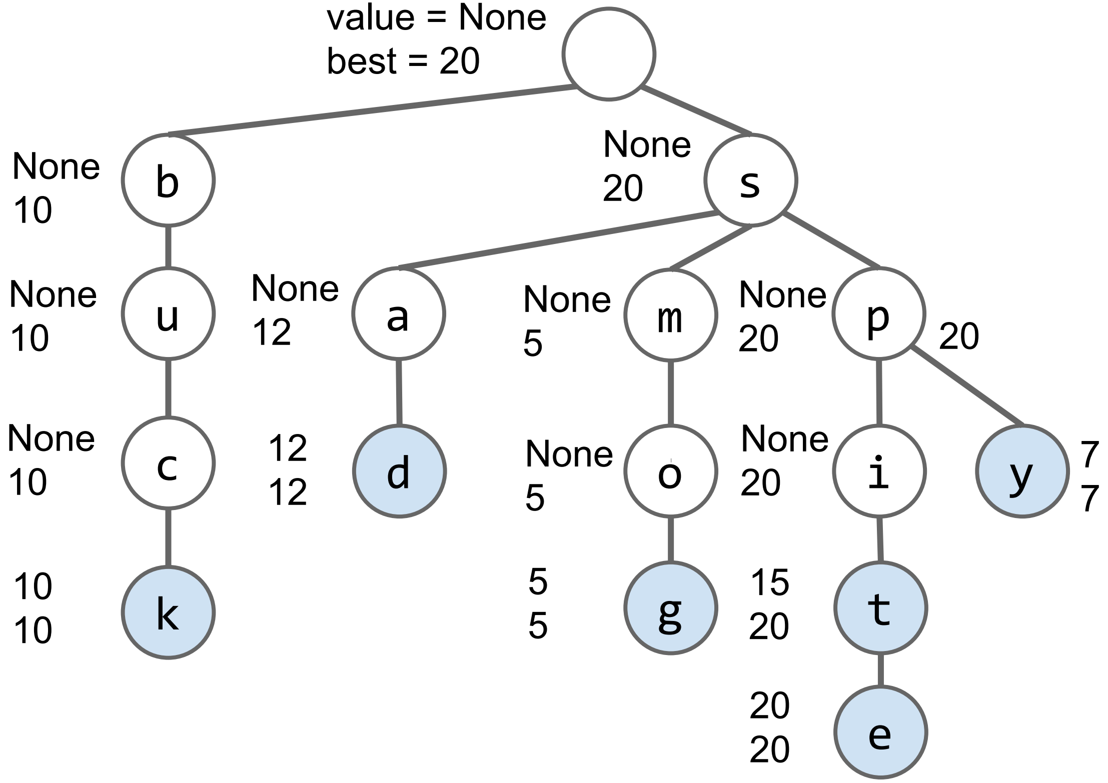

b|

bank of america
best buy
boeing stock
boeing 737 max 8
boeing 737 max
bachelor finale
brie larson
bachelor
bird box
barnes and noble

A More Efficient Autocomplete

One way to address this issue:

- Each node stores its own value, as well as the value of its best substring.



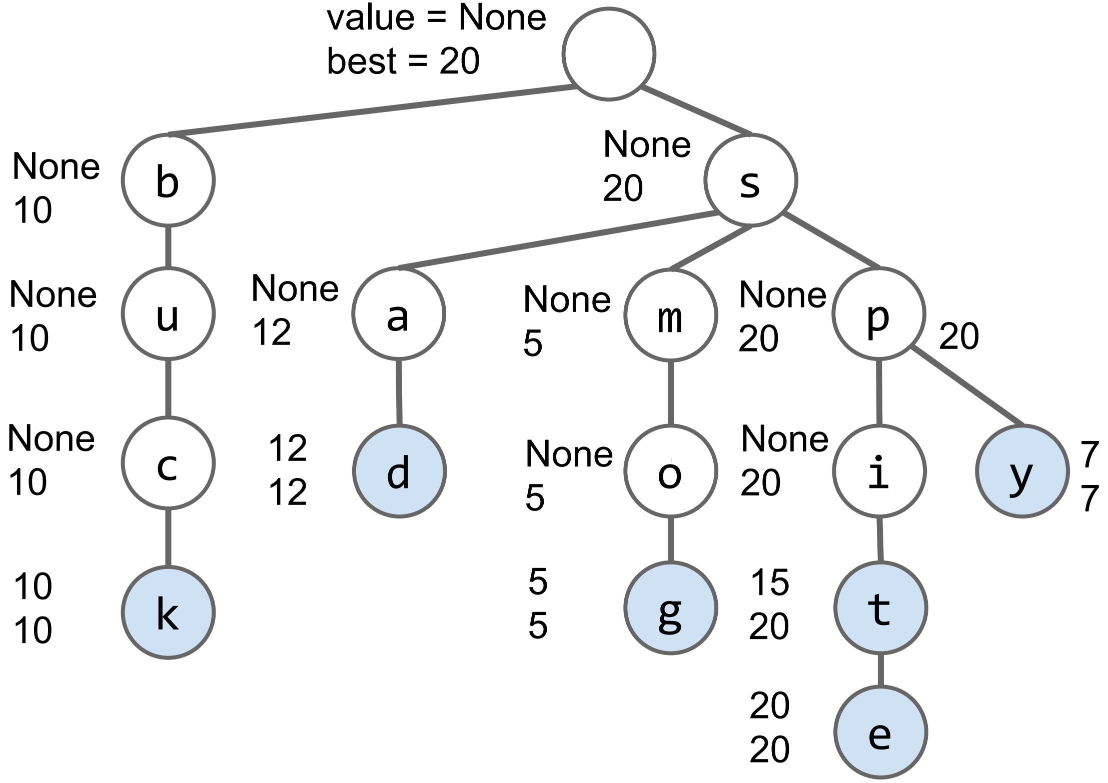
A More Efficient Autocomplete

One way to address this issue:

- Each node stores its own value, as well as the value of its best substring.

Search will consider nodes in order of "best".

- Consider 'sp' before 'sm'.
- Can stop when top 3 matches are all better than best remaining.

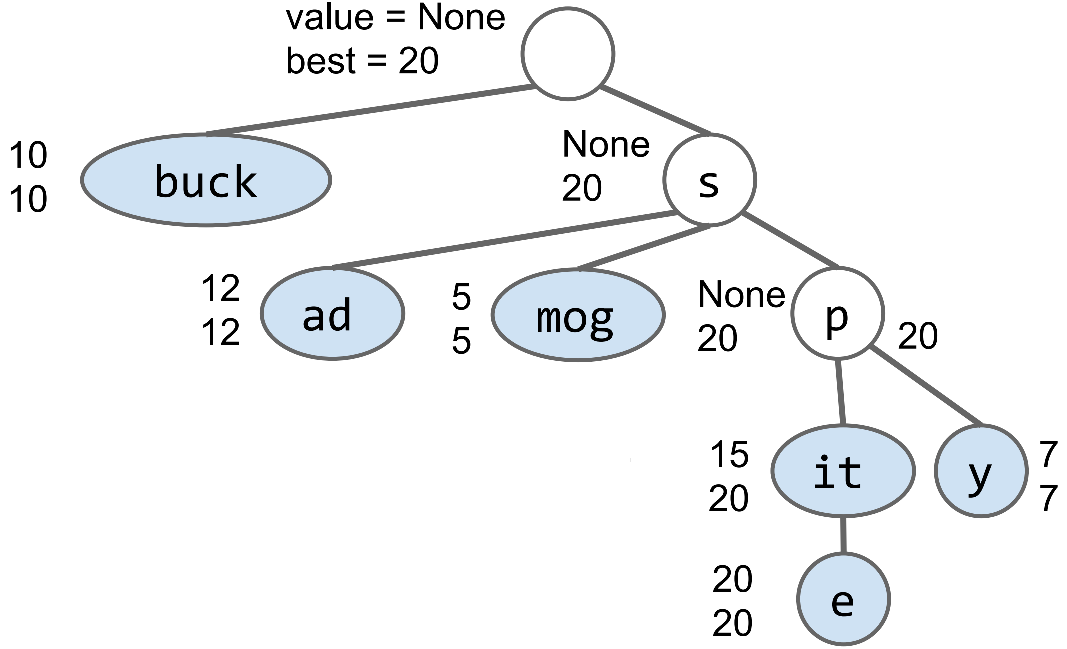


Details left as an exercise. Hint: Use a PQ! See Bear Maps gold points for more.

Even More Efficient Autocomplete

Can also merge nodes that are redundant!

- This version of trie is known as a “radix tree” or “radix trie”.
- Won’t discuss.



Trie Summary

Tries

When your key is a string, you can use a Trie:

- Theoretically better performance than hash table or search tree.
- Have to decide on a mapping from letter to node. Three natural choices:
  - DataIndexedCharMap, i.e. an array of all possible child links.
  - Bushy BST.
  - Hash Table.
- All three choices are fine, though hash table is probably the most natural.
- Supports special string operations like longestPrefixOf and keysWithPrefix.
  - keysWithPrefix is the heart of important technology like autocomplete.
  - Optimal implementation of Autocomplete involves use of a priority queue!

Bottom line: Data structures interact in beautiful and important ways!

Domain Specific Sets and Maps

More generally, we can sometimes take special advantage of our key type to improve our sets and maps.

- Example: Tries handle String keys. Allow for fast string specific operations.
- Note: There are many other types of string sets/maps out there.
    - Suffix Trees (Link).
    - DAWG (Link).
    - Won't discuss in our course.