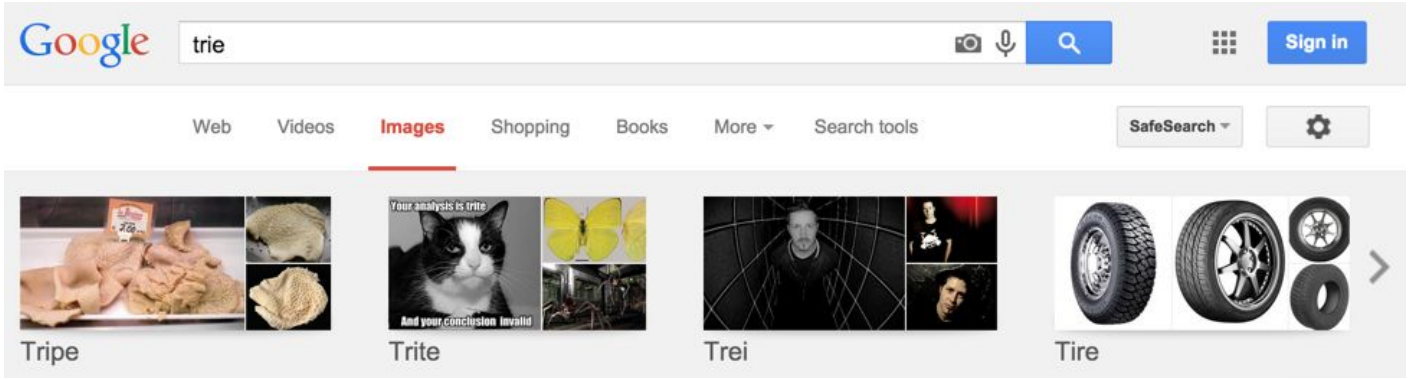
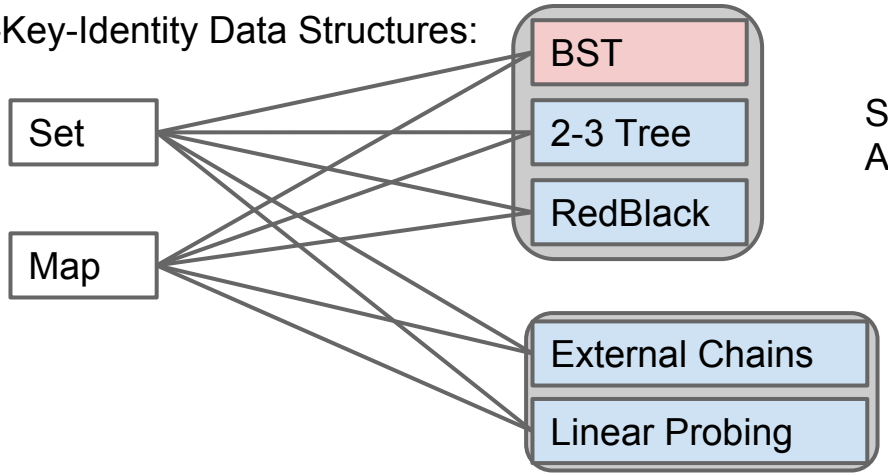



# CS61B

Lecture 37: Tries

- Tries
- Trie Implementation
- Child Link Optimizations
- Ternary Search Tries

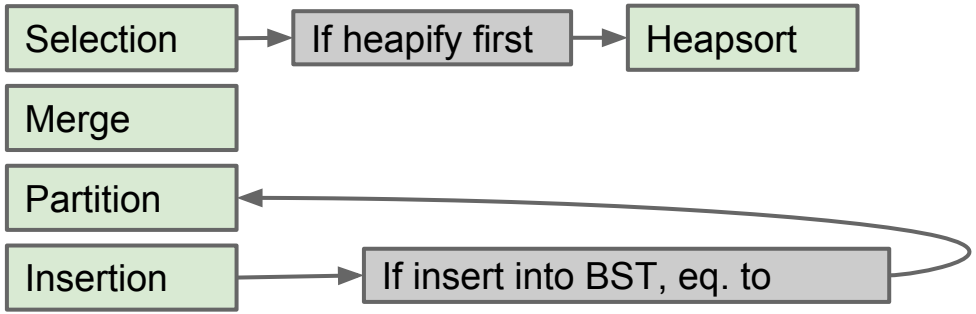
Search-By-Key-Identity Data Structures:



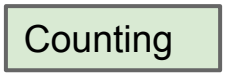

Searches using compareTo()
Analogous to **Comparison-Based**

Searches using hashCode() and equals()
Roughly Analogous to **Integer Sorting**

**Comparison Based** Sorting Algorithms:

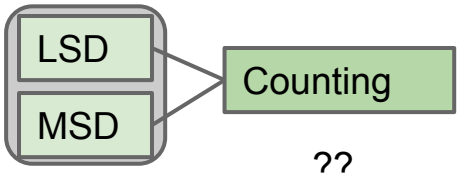



Small-Alphabet **(e.g. Integer) Sorting Algorithms**:




Digit-by-digit Sorting Algorithms:
(require a sorting subroutine)

??

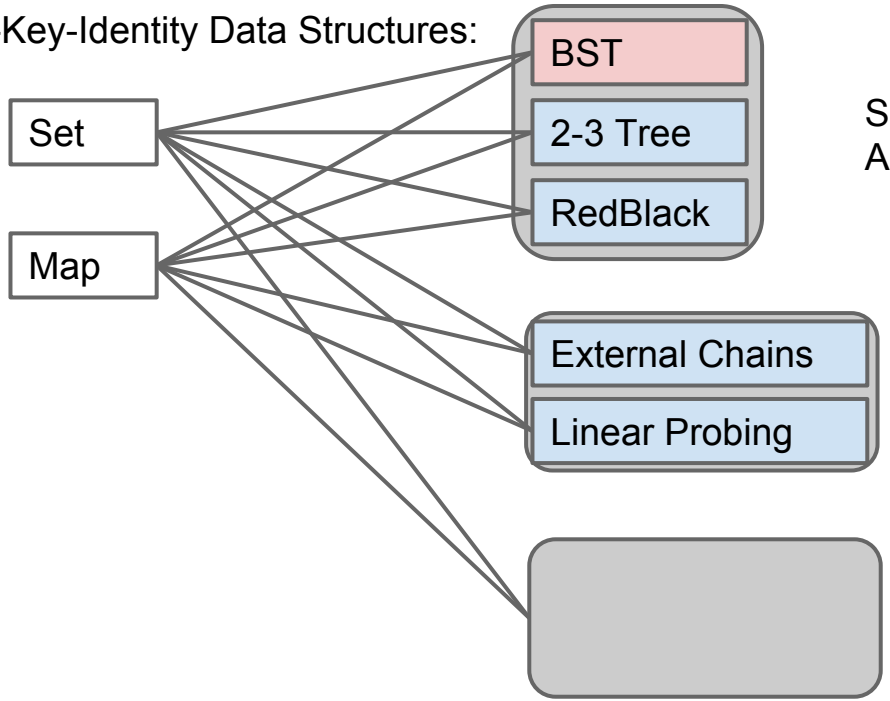



??

??

??

Search-By-Key-Identity Data Structures:




Searches using compareTo()
Analogous to **Comparison-Based**

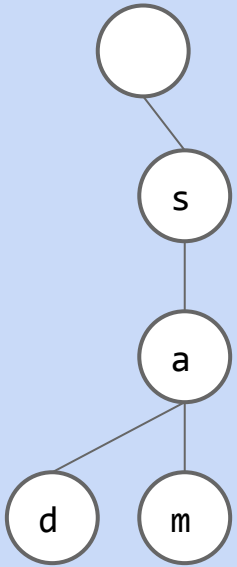
Searches using hashCode() and equals()
Roughly Analogous to **Integer Sorting**

Searches uses … digits?
Analogous to digit-by-digit (radix) sorting.

# Tries

# Digit-by-digit Search

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

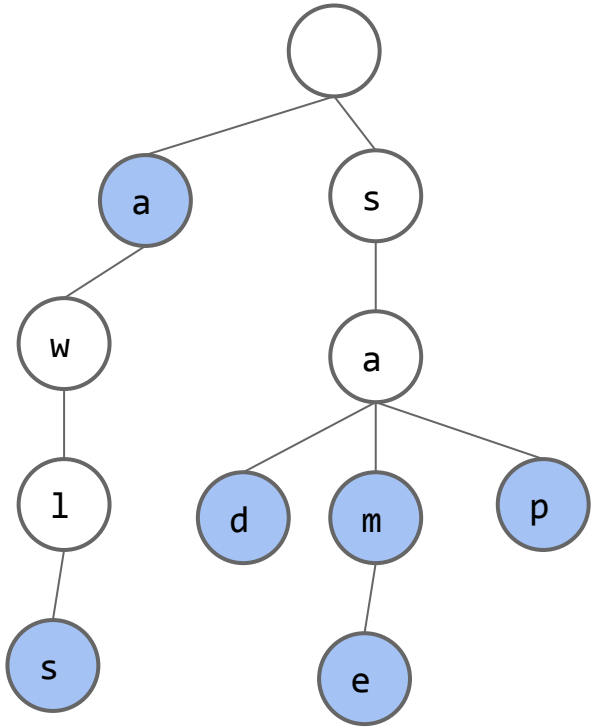



Try to insert the rest.

# Digit-by-digit Search

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(“”): false
- contains(“saq”): false

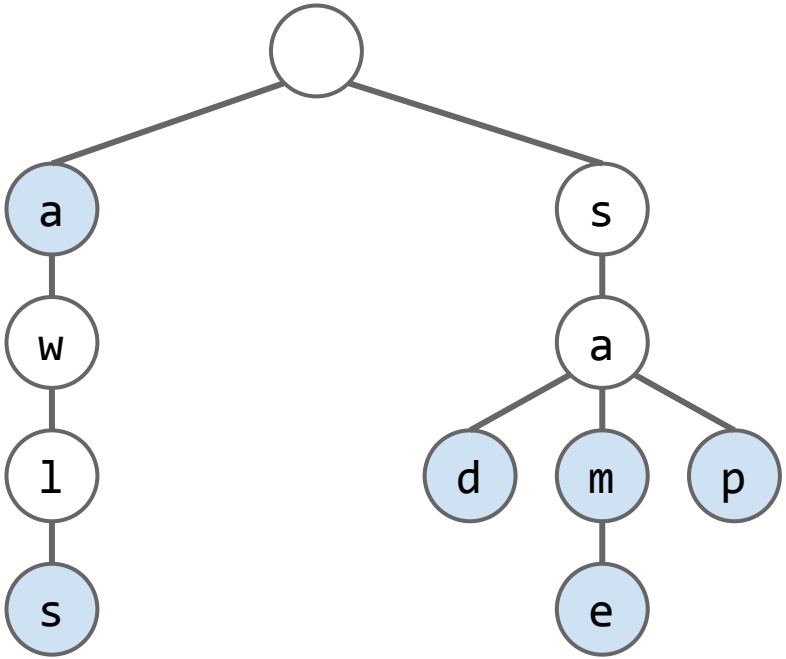



We have a search “miss”:

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

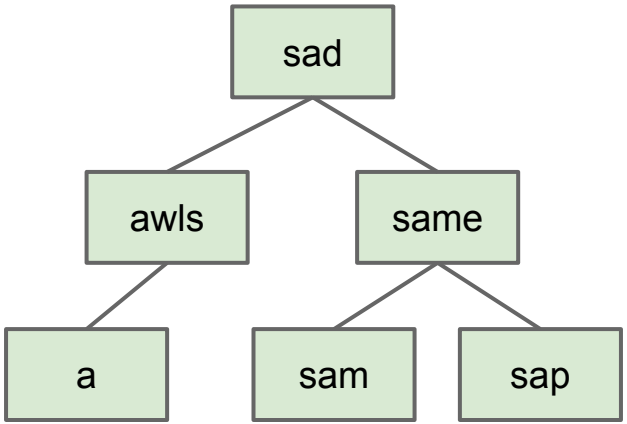
# Digit-by-digit Search

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

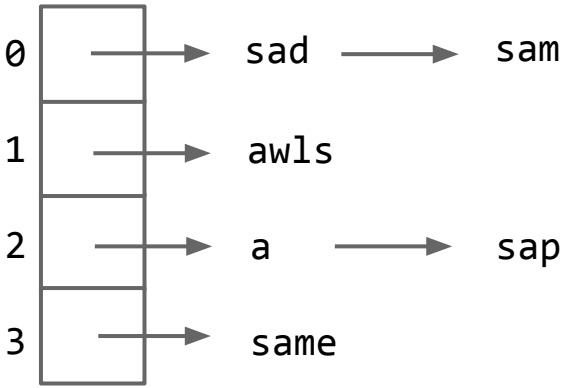



For an animated demo, see [demo from Algorithms textbook](#).

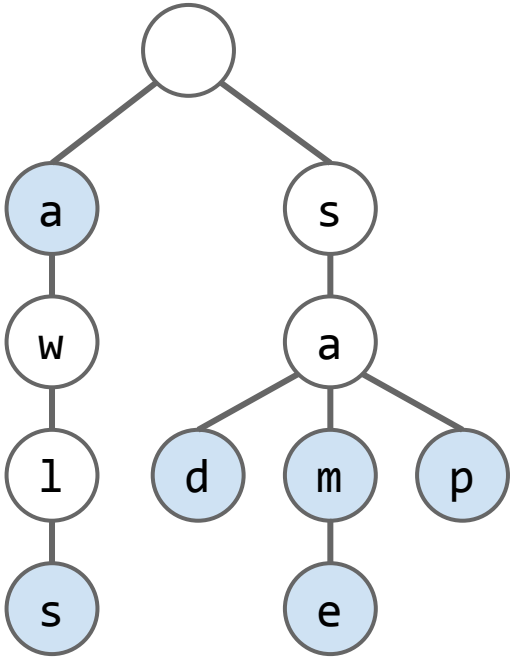
# Tries: A Digit-by-Digit Set Representation




BST




HashSet



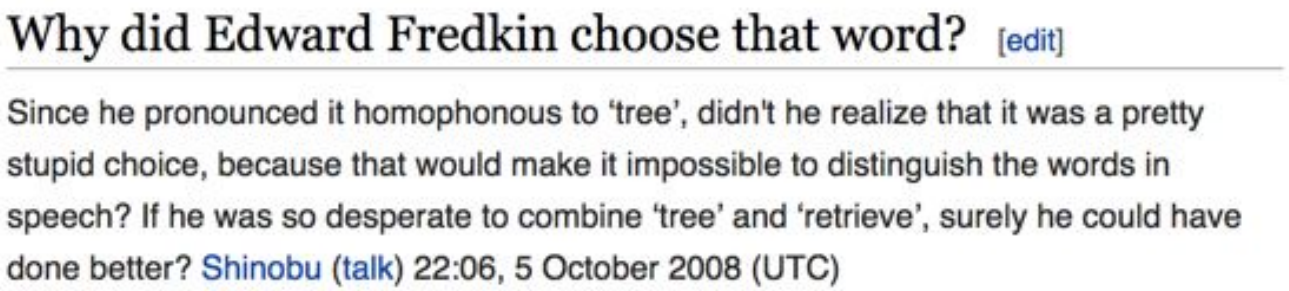

Trie

DAWG

# Tries

Trie:

- Short for Re**trie**val Tree.
- Pronounced same as tree:
  - I'll pronounce as "try", like everyone else except its inventor, Edward Fredkin.



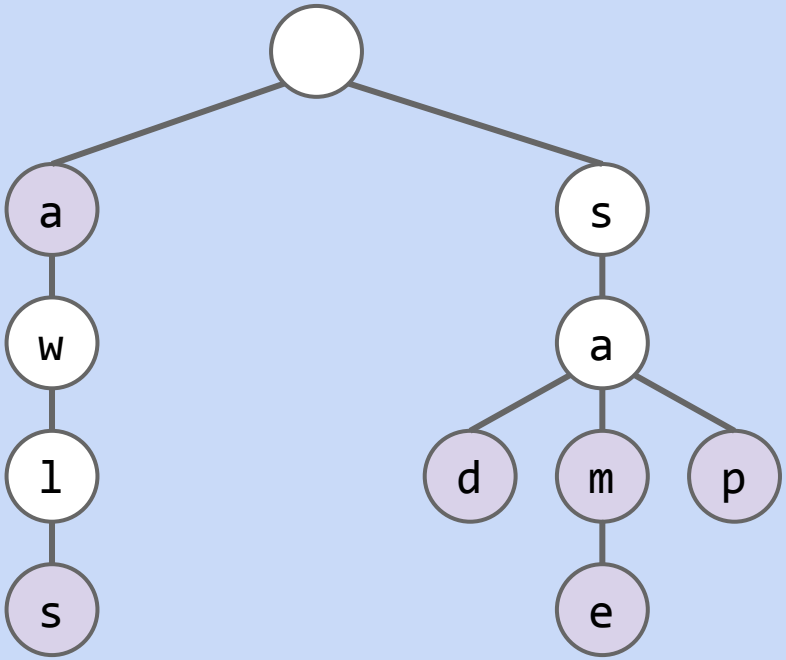
Why did Edward Fredkin choose that word? [edit]

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 Performance in Terms of N and L

Given a Trie with N keys, and a key with L digits. What is the: [n = 6]

- Worst case insert runtime?
- Worst case search runtime?
- Best case search runtime?



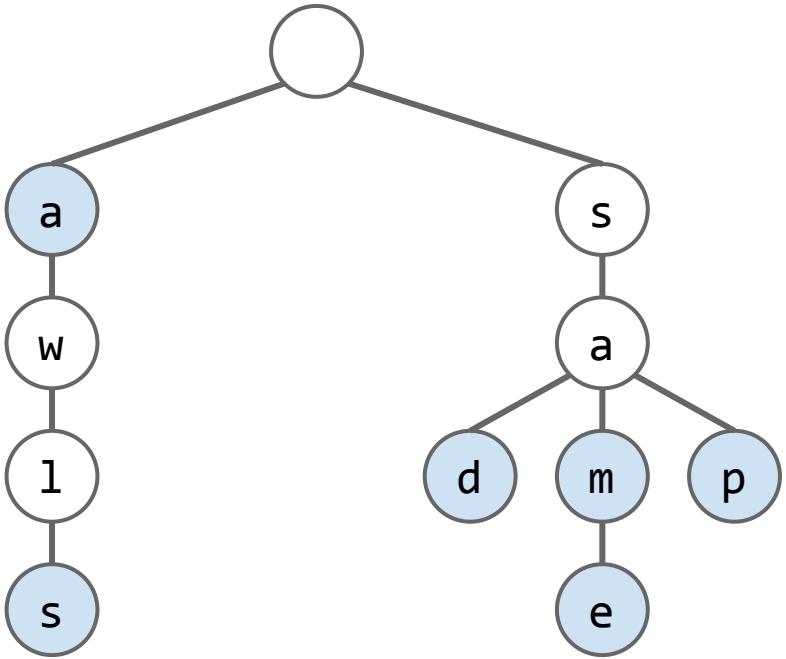

Assume child can be found in constant time.

Bonus Q: How do these compare to a hash table?

# Trie Performance in Terms of N and L

Given a Trie with N keys, and a key with L digits. What is the:

- Worst case insert runtime? Θ(L)
- Worst case search runtime? Θ(L)
- Best case search runtime? Θ(1)
  - Corresponds to a miss on first digit.




Hash table (search): Θ(NL) worst case, Θ(L) typical case, Θ(L) best case.

# Set/Map Data Structures (Raw Performance for Search)

Runtimes for contains()

| | Amortized average | Worst case | Best case |
|---|---|---|---|
| BST (Balanced) | | Θ(L log N) | Θ(L) (hit) |
| Trie | | Θ(L) | Θ(1) (miss) |
| Hash Table | Θ(L)* | | |

N keys, L digits per key. A “miss” means the key isn’t present.

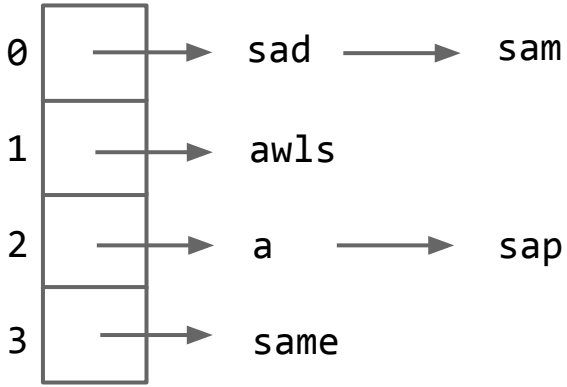
Performance for tries:

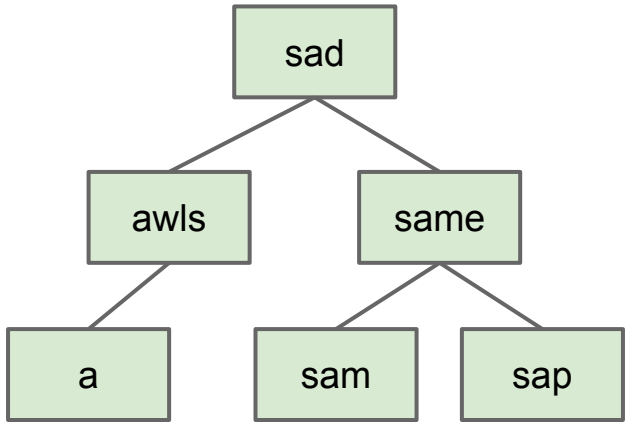
- Like BSTs, comparison is done digit-by-digit, allowing us to skip some characters.
  - Hash tables may look at many unnecessary characters (see sp15 midterm #6b) as the full hashCode needs to be computed.
- Like hash tables, searching is constant with respect to the number of keys.

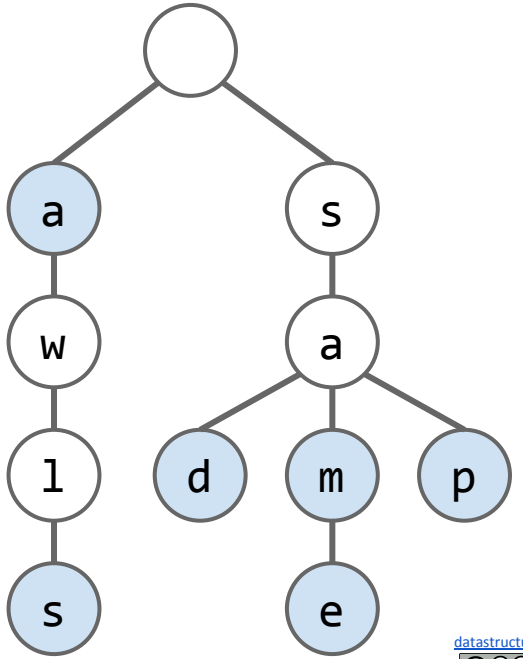
# Usages of Tries

Theoretical asymptotic speed improvement is nice, but **main appeal of tries** is their ability to support rapid **prefix matching** and **approximate matching**.

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



# Trie Implementation and Usage

# Trie Implementation

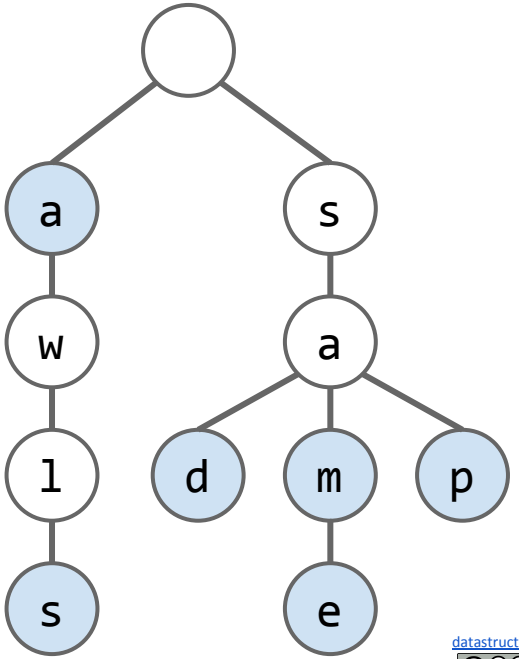
The somewhat counter-intuitive but easiest Java representation of a Trie does not store letters inside nodes, instead letter is stored implicitly on each link.

```
public class TrieSet {
    //Support characters up through #128
    private static final int R = 128;

    private class Node {
        /* Up to R links */
        boolean exists;
        Node[] links;

        public Node() {
            links = new Node[R];
            exists = false;
        }
    }

    private Node root = new Node();
```



# Trie Implementation

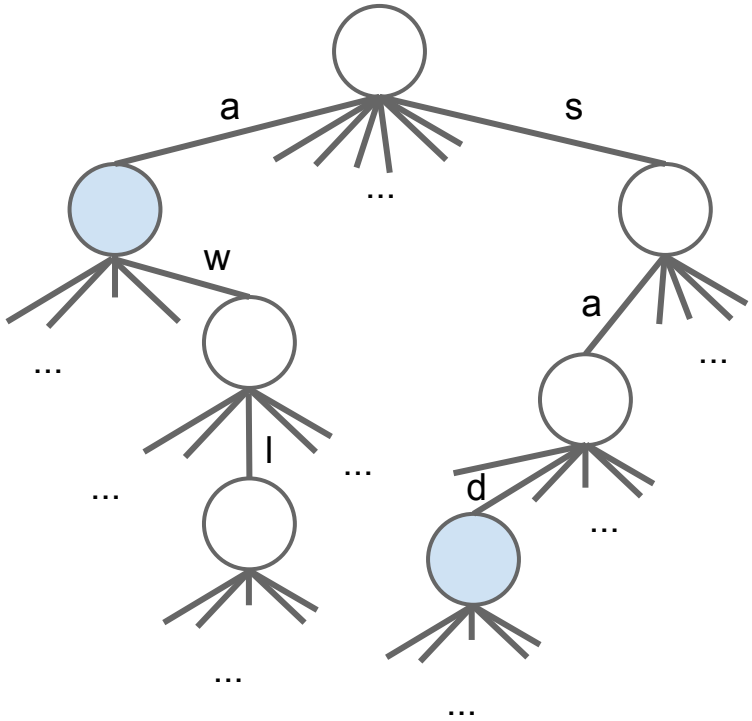
The somewhat counter-intuitive but easiest Java representation of a Trie does not store letters inside nodes, instead letter is stored implicitly on each link.

```
public class TrieSet {
    //Support characters up through #128
    private static final int R = 128;

    private class Node {
        /* Up to R links */
        boolean exists;
        Node[] links;

        public Node() {
            links = new Node[R];
            exists = false;
        }
    }

    private Node root = new Node();
```



# Trie Implementation

```
public class TrieSet {
    //Support characters up through #128
    private static final int R = 128;

    private class Node {
        /* Up to R links */
        boolean exists;
        Node[] links;

        public Node() {
            links = new Node[R];
            exists = false;
        }
    }

    private Node root = new Node();
```

```
public void put(String key) {
    put(root, key, 0);
}

private Node put(Node x, String key, int d) {
    if (x == null) {
        x = new Node();
    }

    if (d == key.length()) {
        x.exists = true;
        return x;
    }

    char c = key.charAt(d);
    x.links[c] = put(x.links[c], key, d+1);
    return x;
}
```

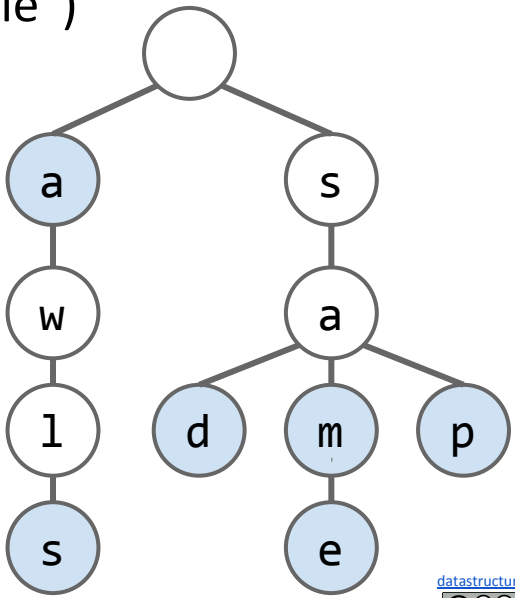
# Usages of Tries

Theoretical asymptotic speed improvement is nice, but **main appeal of tries** is their ability to support rapid **prefix matching** and **approximate matching**.

- Finding all keys that match a given prefix: keysWithPrefix("sa")
  - sad, sam, same, sap
- Finding longest key starting with: longestPrefixOf("sample")
  - sam

Won't cover implementations of these two ops in lecture.

- … but let's try designing one of them.



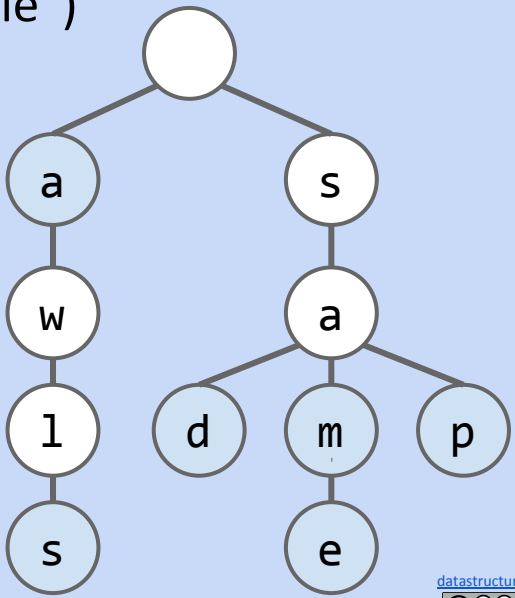
# Usages of Tries

Theoretical asymptotic speed improvement is nice, but **main appeal of tries** is their ability to support rapid **prefix matching** and **approximate matching**.

- Finding all keys that match a given prefix: keysWithPrefix("sa")
  - sad, sam, same, sap
- Finding longest key starting with: longestPrefixOf("sample")
  - sam

Give an algorithm for longestPrefixOf.

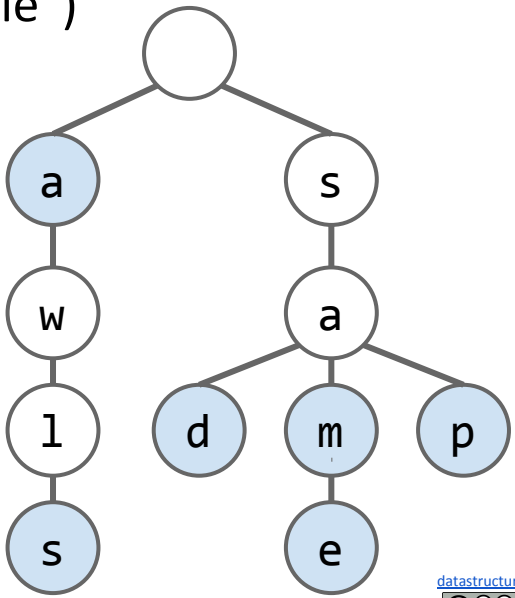
(Bonus question: Give an algorithm for keysWithPrefix)



# Usages of Tries

Theoretical asymptotic speed improvement is nice, but **main appeal of tries** is their ability to support rapid **prefix matching** and **approximate matching**.

- Finding all keys that match a given prefix: keysWithPrefix("sa")
  - sad, sam, same, sap
- Finding longest key starting with: longestPrefixOf("sample")
  - sam




Give an algorithm for longestPrefixOf.

- Check each digit in turn, walking down the tree,
- Keeping track of the most recent blue thing.
- At some point you find the next isn't there, then
- Return that most recent blue thing.

# T9 Texting

For non smart-phone users, texting usually involves coopting your number buttons to type messages.

Approach #1, multi-tapping the word good:

- good: 4 666 666 3

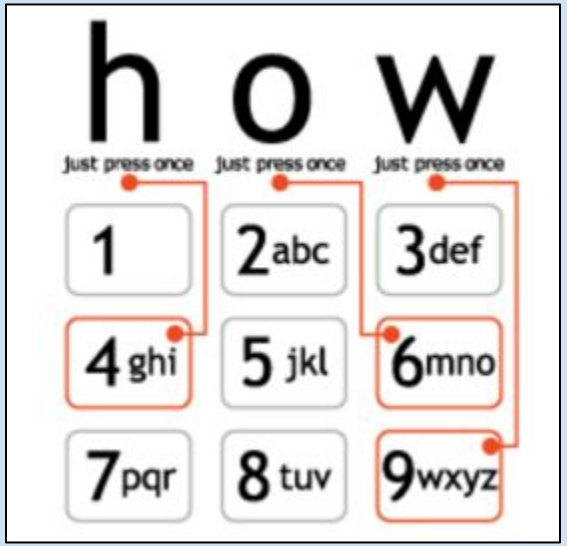
Approach #2, T9 text-input:

- Type 4663, choices: good, home, gone, hoof.
- Press * to select next option.
- Press 0 to see all options.
- Adapts to user's tendencies.

# Implementing T9 Texting

How would you implement T9? TrieSet<?> TrieMap<?, ?>

- Type 4663, choices: good, home, gone, hoof.
- Press * to select next option.
- Press 0 to see all options.
- Adapts to user's tendencies.



# Implementing T9 Texting

How would you implement T9?

- Type 4663, choices: good, home, gone, hoof.
- Press * to select next option.
- Press 0 to see all options.
- Adapts to user's tendencies.

# Implementing T9 Texting (My Answer)

.get(4663) → {good home …}

How would you implement T9?

- Type 4663, choices: good, home, gone, hoof.
- Press * to select next option.
- Press 0 to see all options.
- Adapts to user's tendencies.


TrieMap<String, TreeSet<WeightedWord>>:

- Alphabet (R=8): 2 through 9
- Key: Sequences of numbers (e.g. 4663)
- Value: All possible words: {good, home, gone, hoof, hood}
  - TreeSet<WeightedWord> where a WeightedWord is a string with a compareTo method that contains some sort of weight.
    - When a word is used, increase its weight, moving its position in the TreeSet if necessary.

A TreeMap or HashMap would have worked instead of a TrieMap as well.

# Caveat

## Predictive Text Error Leads to Fatal Stabbing

ANDY CHALK | 10 FEBRUARY 2011 9:36 PM

149

**A U.K. man has been convicted of manslaughter for stabbing his friend to death after a predictive text error ballooned into an argument and ultimately a vicious knife attack.**

33-year-old Neil Brook and his neighbor, 27-year-old Josef Witkowski, had known each other for about six months before getting into a beef over a text message misunderstanding. Brook told police he'd sent Witkowski a text message asking "What are you on about mutter?", "mutter" being "a local colloquialism for a person who behaves in an antisocial or vulgar manner." But thanks to the predictive text on his phone, "mutter" was corrected to "nutter," a slang term meaning "deranged."

http://www.escapistmagazine.com/news/view/107702-Predictive-Text-Error-Leads-to-Fatal-Stabbing

# Tries

Excellent performance, support character-based operations.

Naive implementation is extremely memory hungry.

- Fundamental issue, mapping from a node to its children is done with a data-indexed array (of size R).
- Why so much memory? Every node uses R memory.

Memory usage can be improved through various optimizations:

- Tracking children in a better way than with an array of length R.
- Ternary Search Tries
- Patricia/Radix Tries
- DAWG
- Suffix/Prefix Array

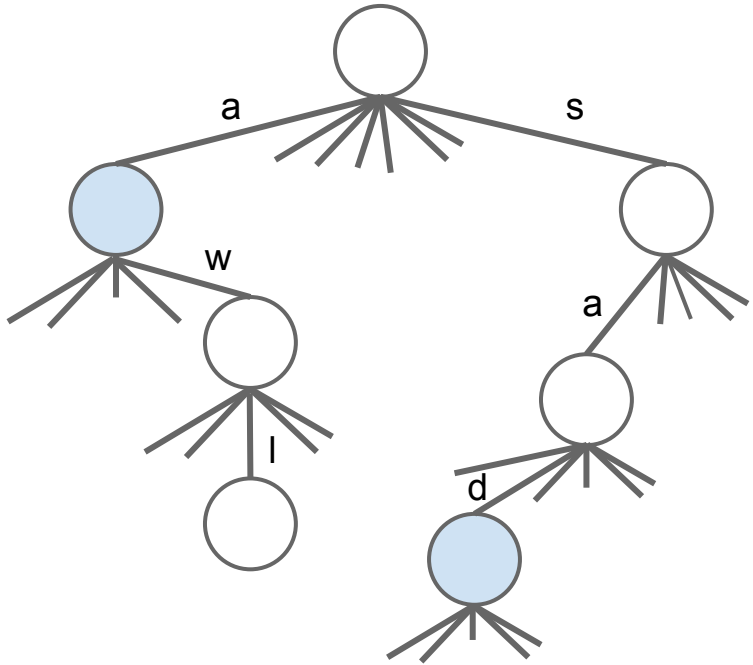
# Child Link Optimizations

# More Accurate Visualization for an Array Based Trie

Each node has R links, where R is the alphabet size.

```
private class Node {
    /* R links */
    boolean exists;
    Node[] links;

    public Node() {
        links = new Node[R];
        exists = false;
    }
}
```

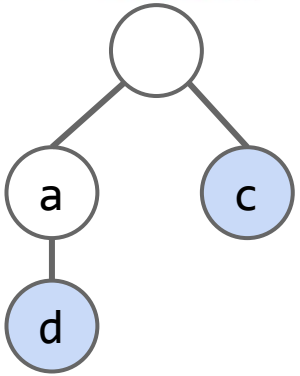



How can we save memory?

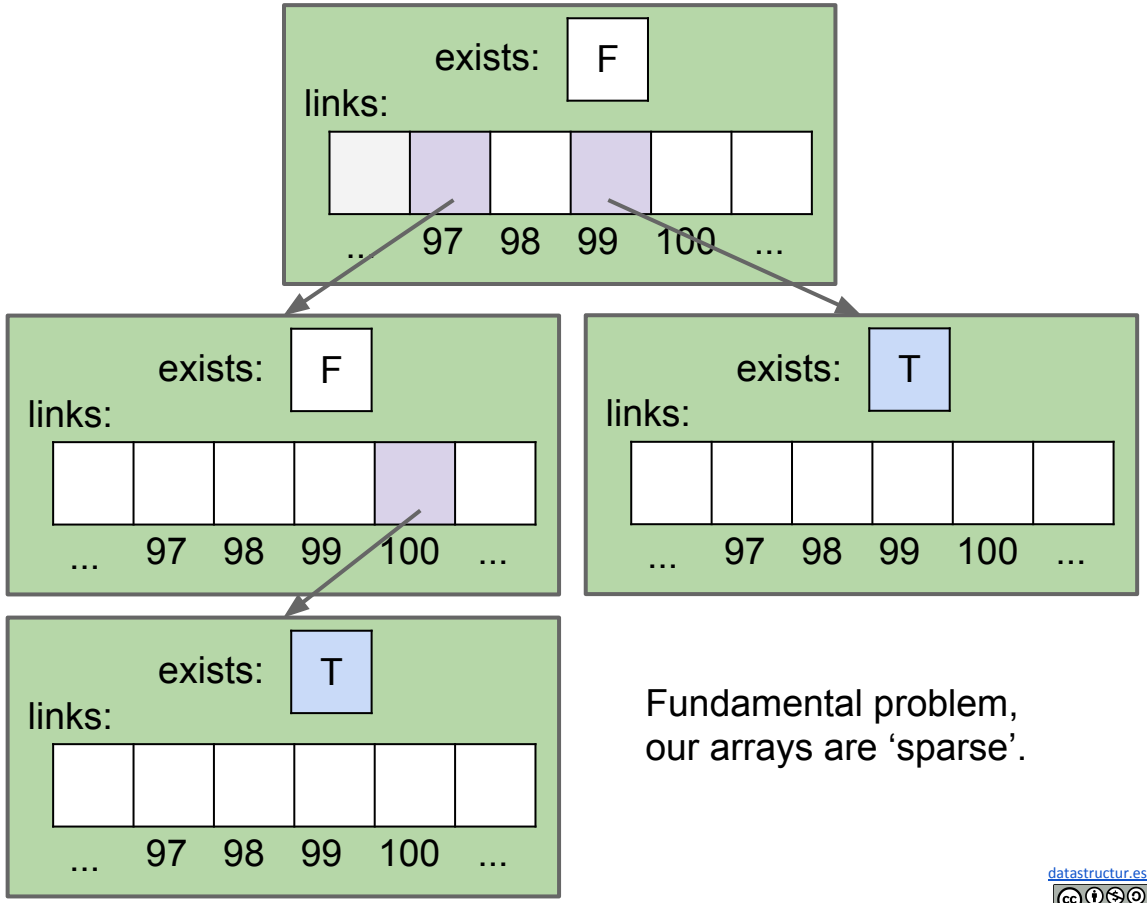
# Even More Accurate Visualization for an Array Based Trie

```
private class Node {
    /* R links */
    boolean exists;
    Node[] links;

    public Node() {
        links = new Node[R];
        exists = false;
    }
}
```

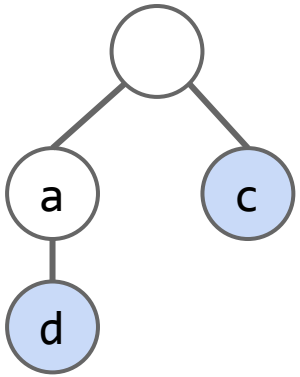


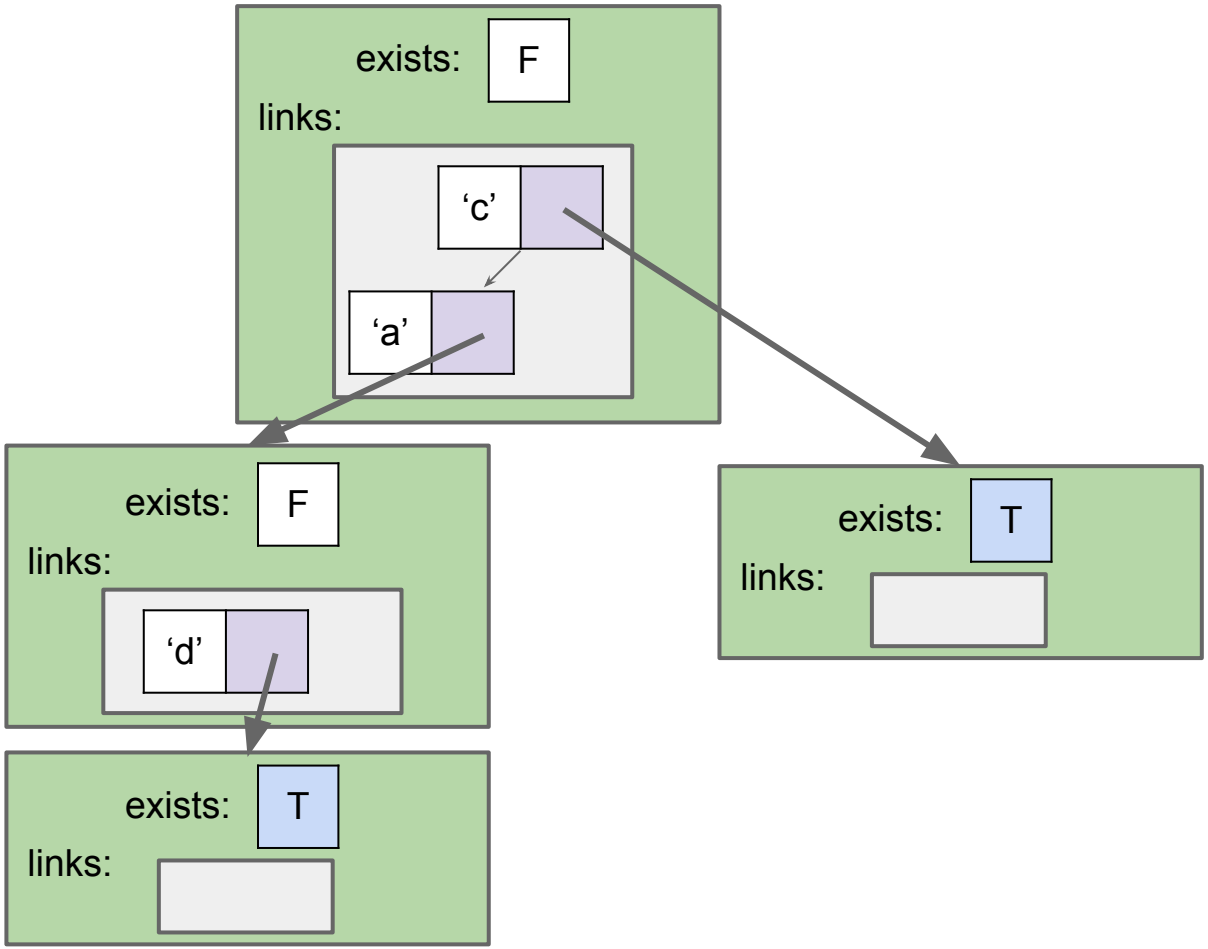

How can we save memory?




Fundamental problem, our arrays are ‘sparse’.

# Visualization for a BST Based Trie



# Generalizing our Trie Implementation

What abstraction can we use in place of the Node[] array that is more general?

```
private class Node {
    /* R links */
    boolean exists;
    Node[] links;

    public Node() {
        links = new Node[R];
        exists = false;
    }
}
```

# A More Modular TrieSet

Replace array with a Map.

```
private class Node {
    /* Up to R links */
    boolean exists;
    Map<Character, Node> links;

    public Node() {
        links = new TreeMap<Character, Node>();
        exists = false;
    }
}
```

- Version above still requires hard coded choice of map instantiation i.e HashMap vs TreeMap.

# A More Modular TrieSet

Replace array with a Map.

```
private class Node {
    /* Up to R links */
    boolean exists;
    Map<Character, Node> links;

    public Node() {
        links = new TreeMap<Character, Node>();
        exists = false;
    }
}
```

Each choice of map comes with its own runtime / space tradeoff:

- Data-indexed array: Max speed, max memory.
- TreeMap/HashMap: Slower query performance, but less memory wasted.
  - Can mitigate performance issues by initializing HashMap to be small

# Ternary Search Tries

# Ternary Search Trie: Restricting the Number of Links

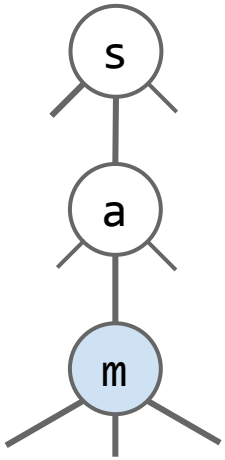
Alternately approach: Redesign Trie so that we have a fixed number of links.

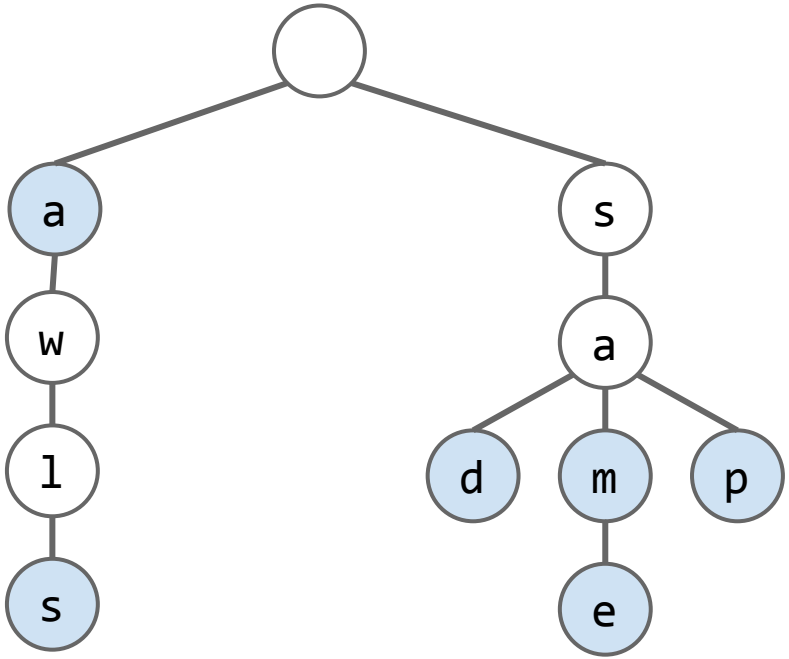
One approach: The Ternary Search Trie

- Assign a character to each node.
- Give each node 3 links:
  - Left link if key's next character < node's character.
  - Middle link if key's next character == node's character.
  - Right link if key's next character > node's character.

# Ternary Search Trie

After inserting “sam”

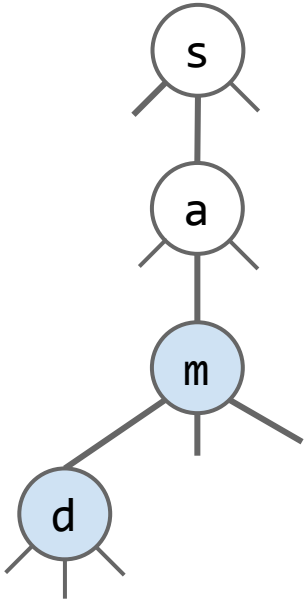


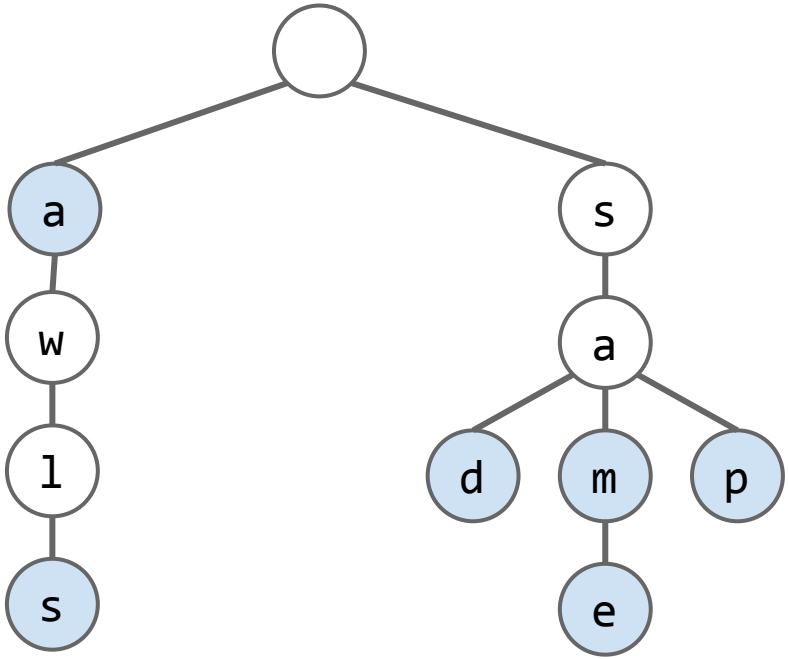




Animated Demo: http://goo.gl/dq1EDa

# Ternary Search Trie

After inserting “sam”, “sad”

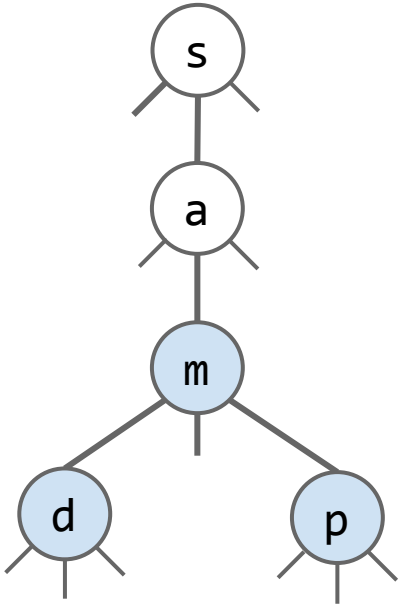


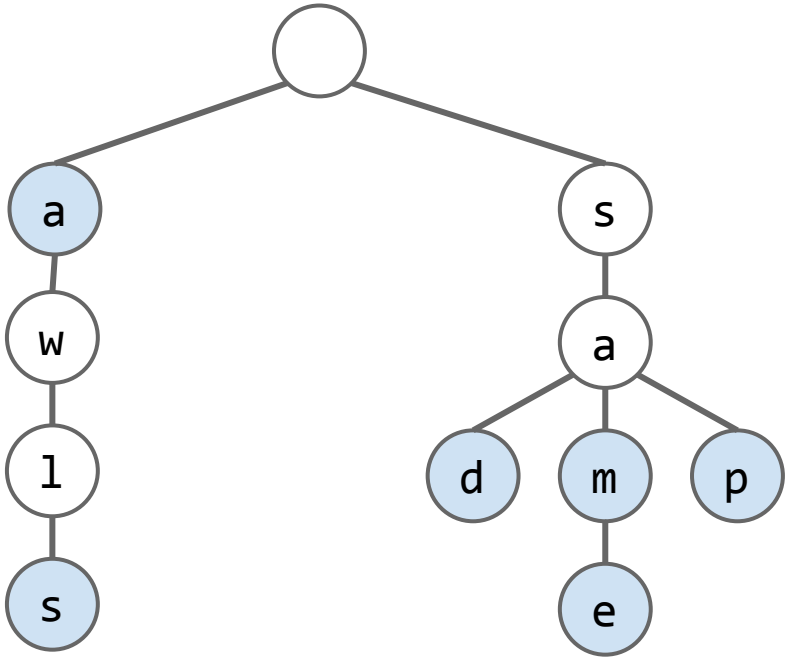




Animated Demo: http://goo.gl/dq1EDa

# Ternary Search Trie

After inserting “sam”, “sad”, “sap”

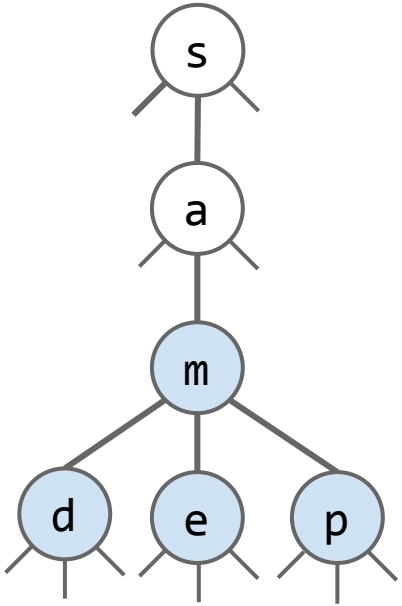


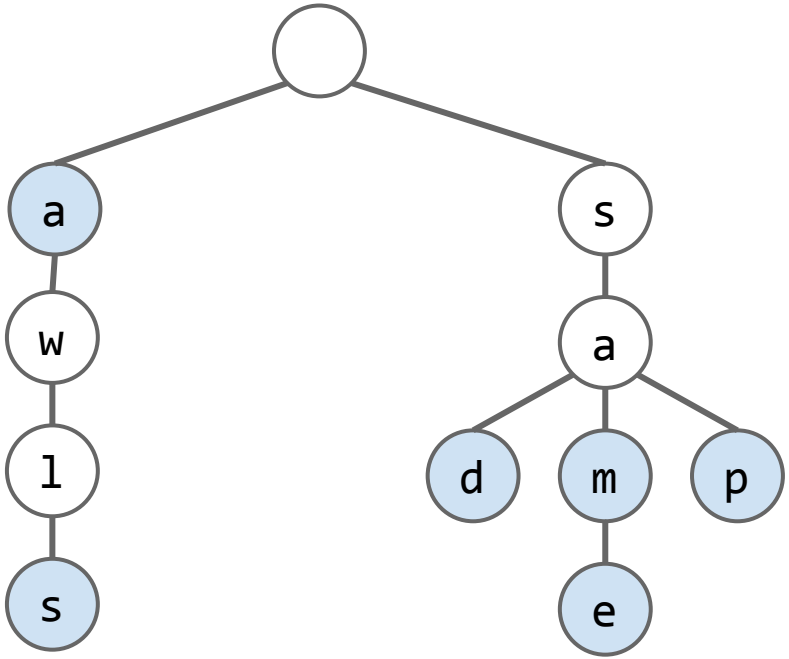




Animated Demo: http://goo.gl/dq1EDa

# Ternary Search Trie

After inserting “sam”, “sad”, “sap”, “same”

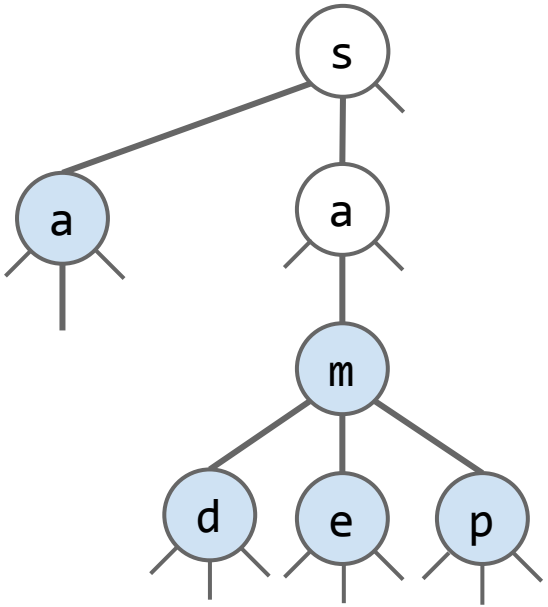


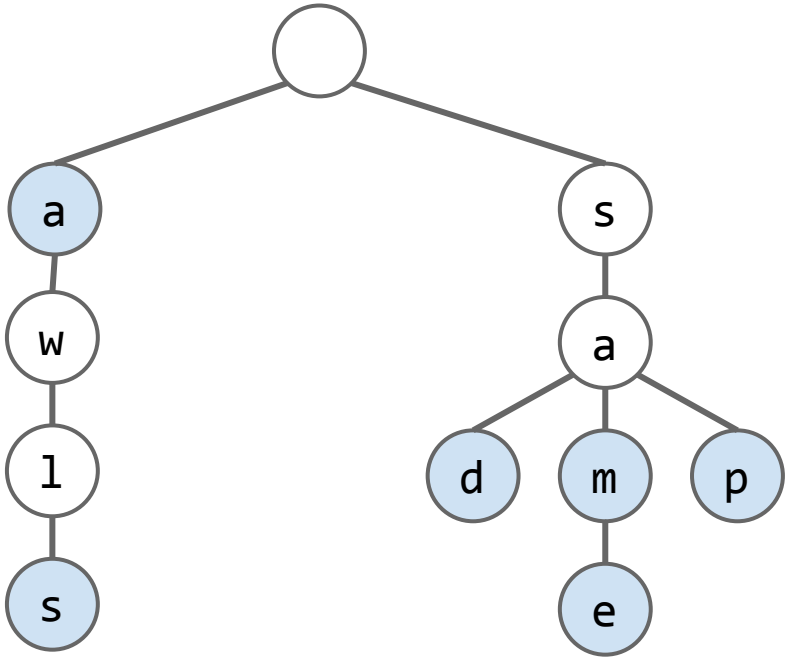




Animated Demo: http://goo.gl/dq1EDa

# Ternary Search Trie

After inserting “sam”, “sad”, “sap”, “same”, “a”

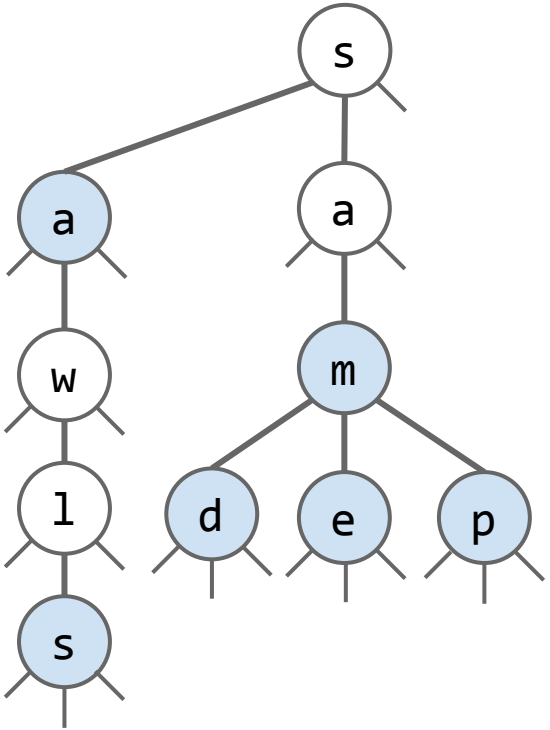


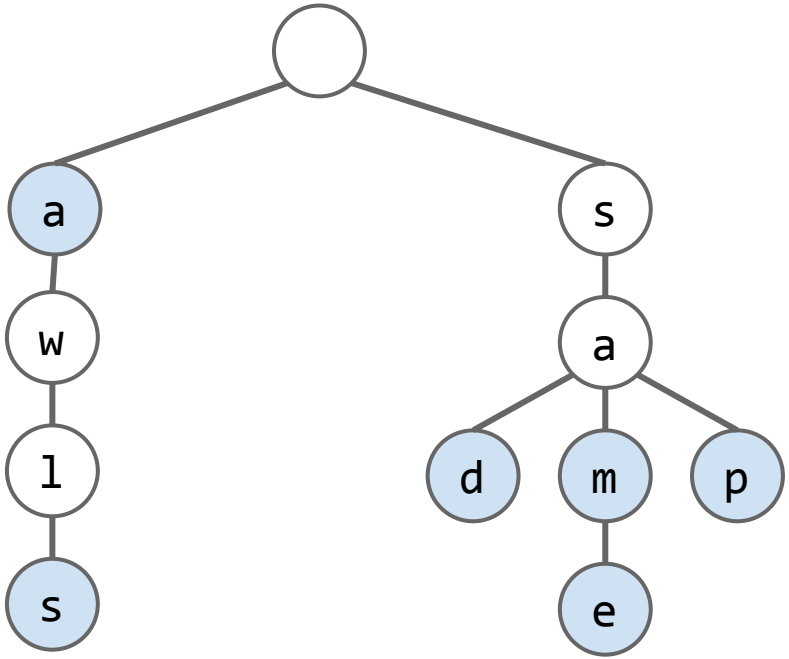




Animated Demo: http://goo.gl/dq1EDa

# Ternary Search Trie

After inserting “sam”, “sad”, “sap”, “same”, “a”, and “awls”




Animated Demo: http://goo.gl/dq1EDa

# TST Implementation

```
public class TSTSet<Value> {
    private Node<Value> root;

    private static class Node<Value> {
        /* char in this node */
        private char c;
        /* left, middle, and right subtries */
        private Node<Value> left, mid, right;
    }
}
```

Full implementation available from Algorithms textbook.

# TST Performance

Give an example of a sequence of insert operations that results in much worse performance than a standard trie.

# TST Performance

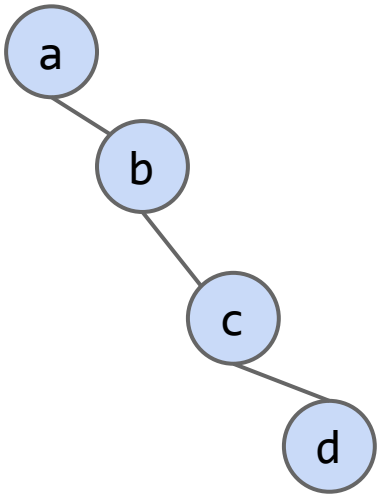
Give an example of a sequence of insert operations that results in much worse performance for a TST than a standard trie.

insert(“A”)

insert(“B”)

insert(“C”)

insert(“D”)



# Set/Map Data Structures

Runtimes for contains()

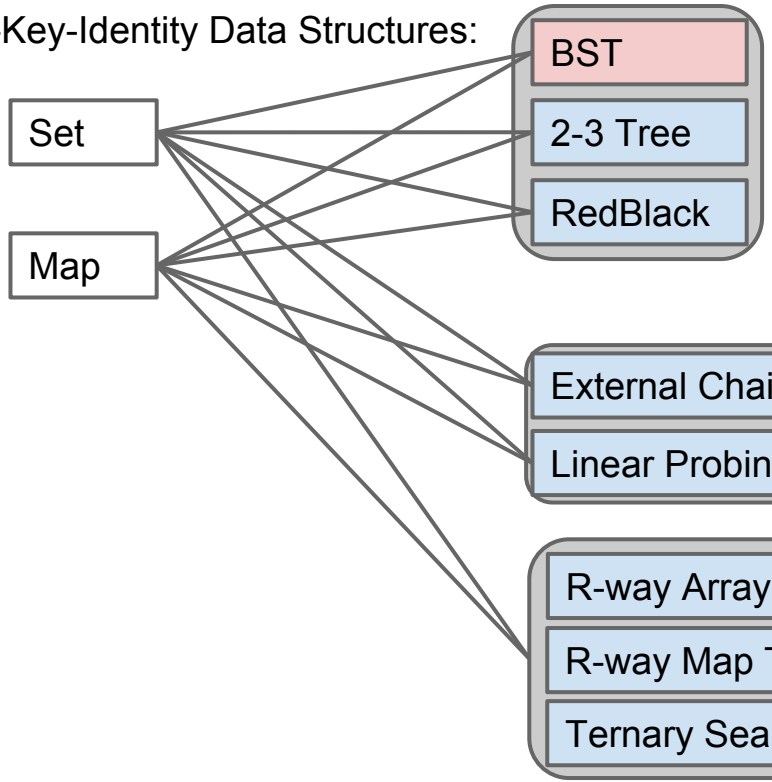
| | Amortized average | Worst case | Best case (miss) | Memory |
|---|---|---|---|---|
| Hash Table | Θ(L)* | | | Θ(NL) |
| BST | | Θ(L log N) | Θ(1) | Θ(NL) |
| Trie (array map) | | Θ(L) | Θ(1) | Θ(NLR) |
| Trie (HashMap) | | Θ(L) | Θ(1) | Θ(NL) |
| Trie (TreeMap) | | Θ(L log R) | Θ(1) | Θ(NL) |
| TST | | Θ(NL) | Θ(1) | Θ(NL) |

N keys, L digits per key, R alphabet size. A miss means the key isn't present.

Tries and TSTs:

- Performance improvement over previous data structures.
- Support character-based operations
- Various implementations trade off memory vs. time.

Search-By-Key-Identity Data Structures:




Searches using compareTo()
Analogous to **Comparison-Based**

Searches using hashCode() and equals()
Roughly Analogous to **Integer Sorting**

Searches uses digits.
Analogous to digit-by-digit (radix) sorting.

# Extra Fall 2014 Slides:
# A Tricky OOP Exercise Somewhat Beyond the Scope of 61B

# Further Generalizing our Trie Implementation

As an OOP exercise, let's consider how we might further generalize.

- Idea: Let client decide how links are handled.

```
public BruteForceGeneralTrieSet(String linkType) {
    if (linkType.equals("BST-Based")) {
        ...
    } else if (linkType.equals("Hash-Based")) {
        ...
    } else ...
}
```

Better ideas?

```
BruteForceGeneralTrieSet treeset
    = new BruteForceGeneralTrieSet("BST-Based");
```

```
private class Node {
    /* Up to R links */
    boolean exists;
    Map<Character, Node> links;

    public Node() {
        links = new TreeMap<Character, Node>();
        ...
```

# Approach #1: Use Generics?

Can we somehow use generics? This approach is doomed. Why?

```
public class GenericMapTrieSet<SomeTypeOfMap
      extends Map<Character, GenericMapTrieSet.Node>> {


    public class Node<SomeTypeOfMap> {
        /* Up to R links */
        boolean exists;
        SomeTypeOfMap links;

        public Node() {
            links = new SomeTypeOfMap();
            exists = false;
        }
    }

    private Node root;
    public GenericMapTrieSet() {
        root = new Node<SomeTypeOfMap>();
    }
```

# Implementing Special Cases: Extension, Delegation, Adaptation

Lecture 17 retrospective: Suppose we have a List of some type. How do we get a Stack?

- Approach #3 from lecture 17 was adaptation.

```
public class StackAdapter<Item> {
    private List L;

    public StackAdapter(List<Item> worker) {
        L = worker;
    }

    public void push(Item x) {
        L.add(x);
    }
}
```

```
ArrayList<String> al = new ArrayList<String>();
StackAdapter<String> x = new StackAdapter<String>(al);
x.push("hello");
```

# Approach #2: Use the idea of Adaptation?

What is wrong with the idea below?

```
public class AdapterTrieSet {

    public class Node {
        /* Up to R links */
        boolean exists;
        Map<Character, Node> links;

        public Node(Map<Character, Node> m) {
            links = m;
            exists = false;
        }
    }

    private Node root;
    public AdapterTrieSet(Map<Character, Node> m) {
        root = new Node(m);
    }
```

# Tweaking the Adapter Idea

Every time we create a node, we need a new map.

- What should we provide to the AdapterTrieSet constructor?

```
public class AdapterTrieSet {

    public class Node {
        /* Up to R links */
        boolean exists;
        Map<Character, Node> links;

        public Node(Map<Character, Node> m) {
            links = m;
            exists = false;
        }
    }

    private Node root;
    public AdapterTrieSet(Map<Character, Node> m) {
        root = new Node(m);
    }
```

# Approach #3: Give the AdapterTrieSet a Factory.

Every time we create a node, we need a new map.

- AdapterTrieSet needs ability to generate new maps of some specific type.
  - Needs a “factory”.

```
public class Node {
    /* Up to R links */
    boolean exists;
    Map<Character, Node> links;

    public Node(Map<Character, Node> m) {
        links = m;
        exists = false;
    }
}

private Node root;
public AdapterTrieSet(MapFactory mf) {
    root = new Node(mf.newMap());
}
```

# Citations

T9 text slides and interview question: Kevin Wayne, Princeton University