CS2010: Data Structures and Algorithms II

Tries

Ivana.Dusparic@scss.tcd.ie

Where are we?

- Sorting algorithms
  - General
  - String-specific
- Exact pattern matching algorithms/substring search
- Symbol tables – ordered/unordered
  - Lists, queues, hashtables
  - Trees – binary search tree, 2-3 tree, red-black BST

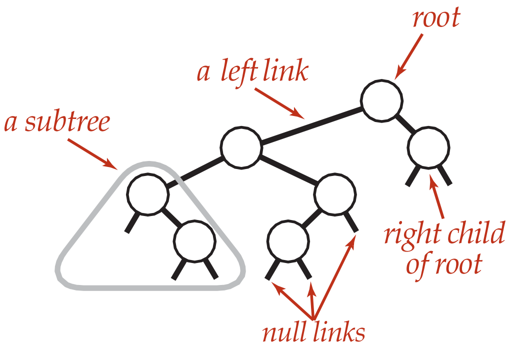
  - Can we do better with string-specific symbol tables?

Binary search tree (BST)

Definition. A BST is a binary tree in symmetric order.

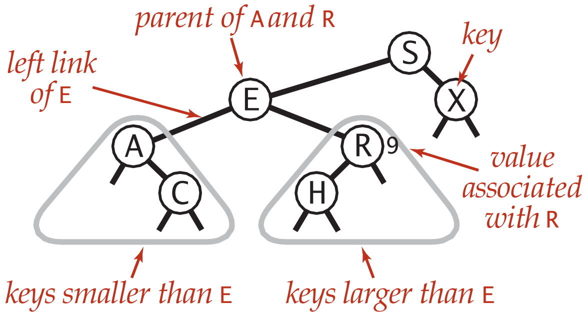
A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).



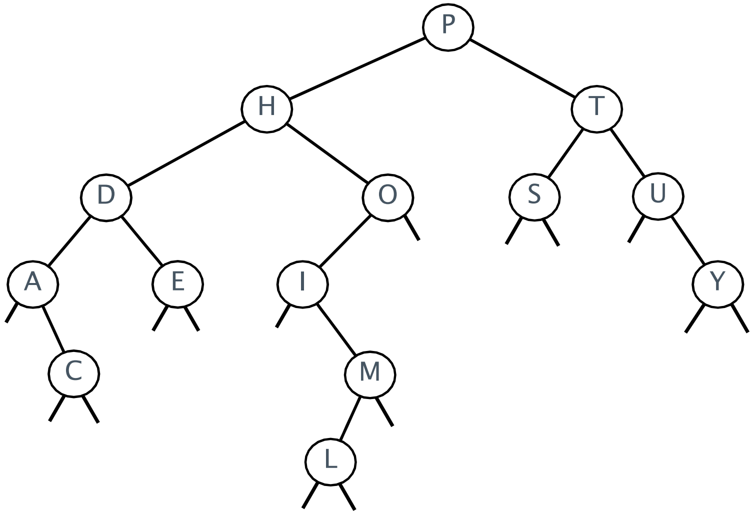
Symmetric order. Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



Correspondence between BSTs and quicksort partitioning

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y



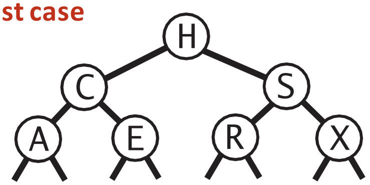
Remark. Correspondence is 1-1 if array has no duplicate keys.

Property. Inorder traversal of a BST yields keys in ascending order.

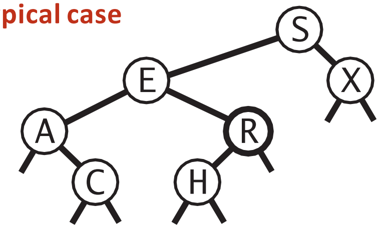
Tree shape

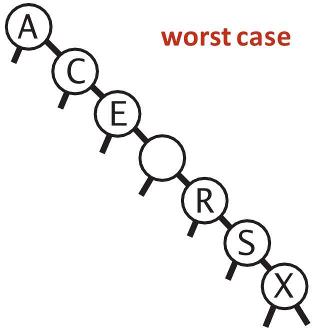
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to 1 + depth of node.

best case



typical case





worst case

Bottom line. Tree shape depends on order of insertion.

Balanced Search Trees

- 2-3 tree
- Red-black BST – BST implementation of 2-3 tree

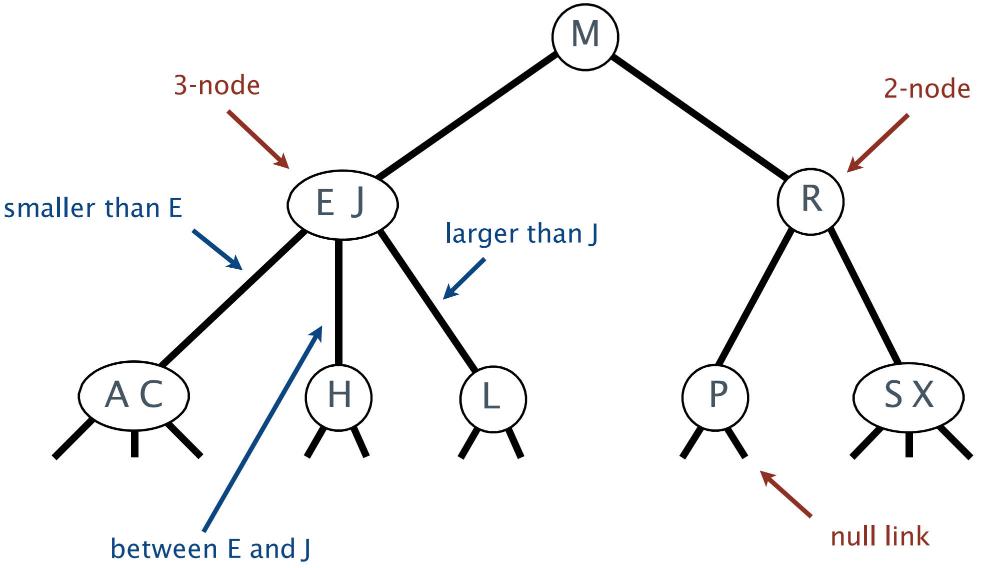
2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

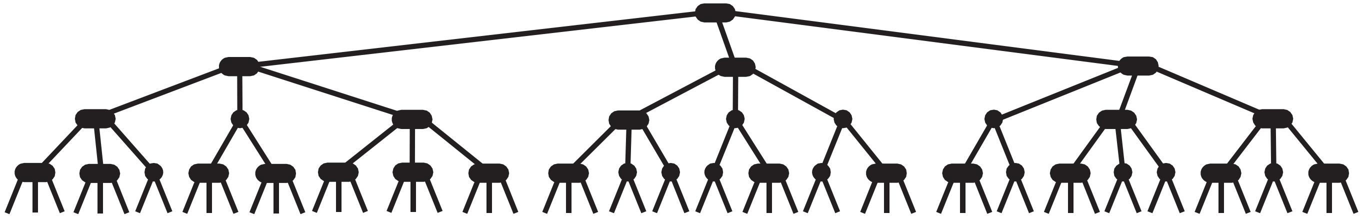
Symmetric order. Inorder traversal yields keys in ascending order.

Perfect balance. Every path from root to null link has same length.



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.



Tree height.

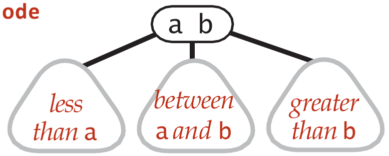
- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

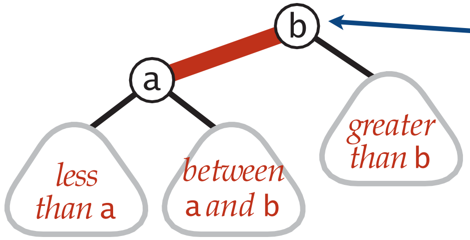
Bottom line. Guaranteed logarithmic performance for search and insert.

Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3–nodes.

3-node

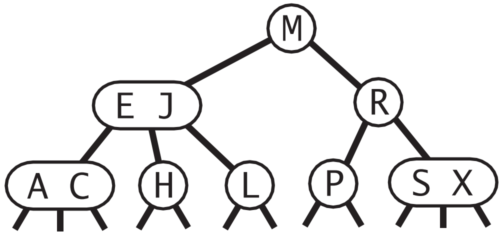




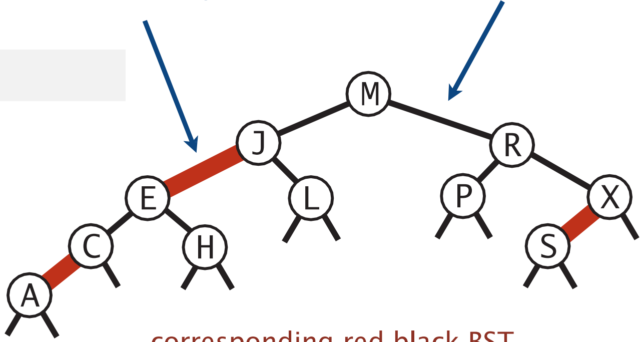
larger key is root

red links "glue" nodes within a 3-node

black links connect 2-nodes and 3-nodes



2-3 tree

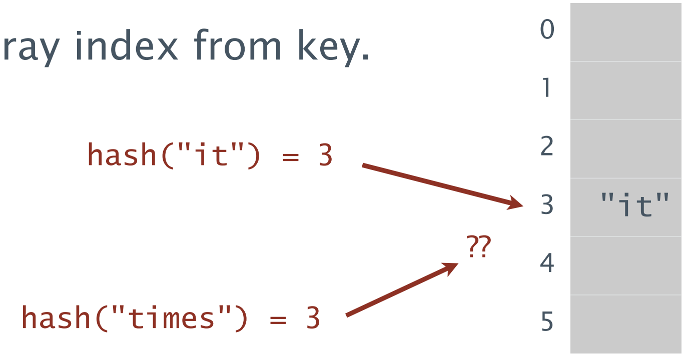


corresponding red-black BST

Hash tables

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



hash("it") = 3

hash("times") = 3

??

0

1

2

3 "it"

4

5

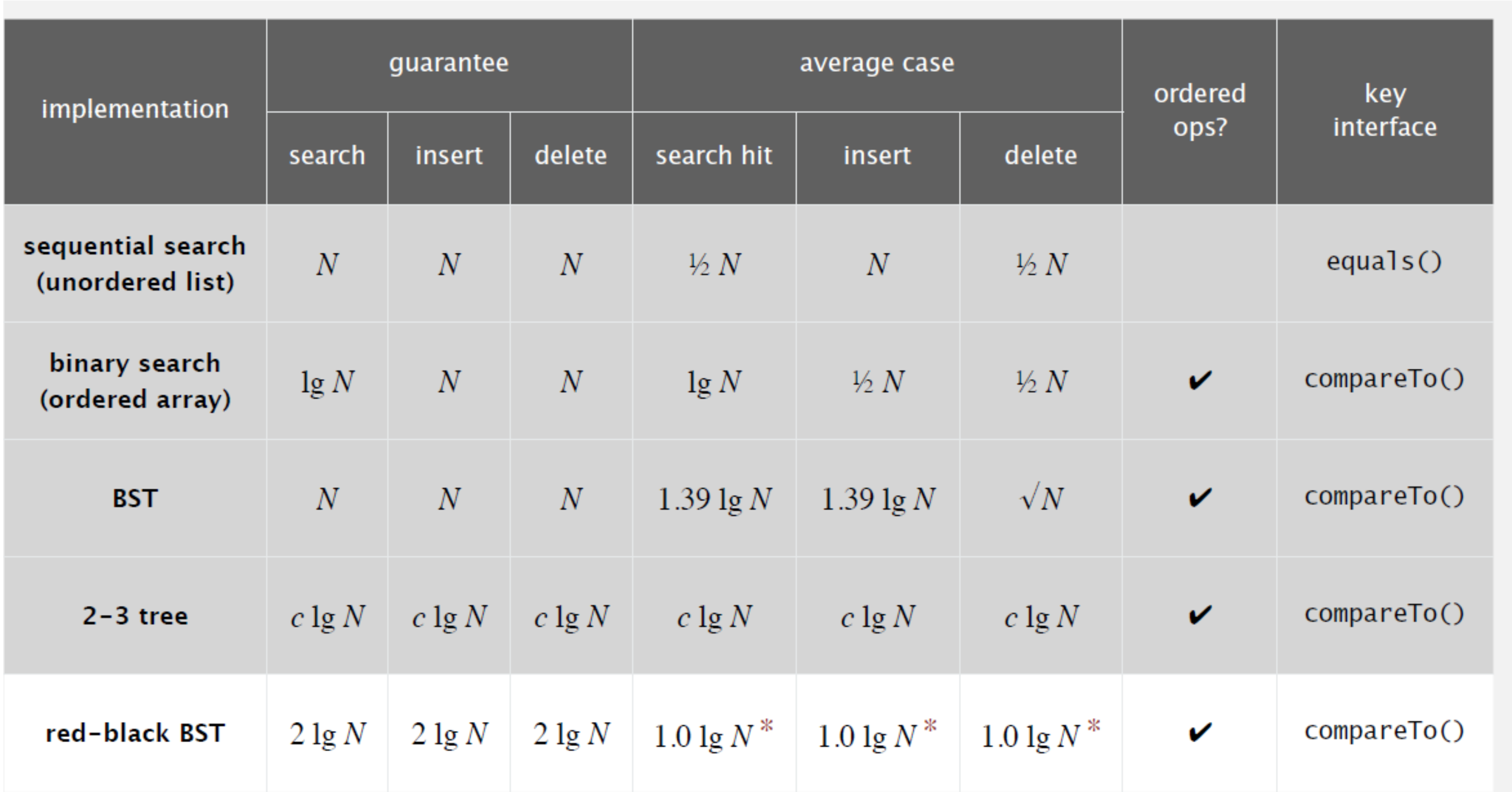
Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.

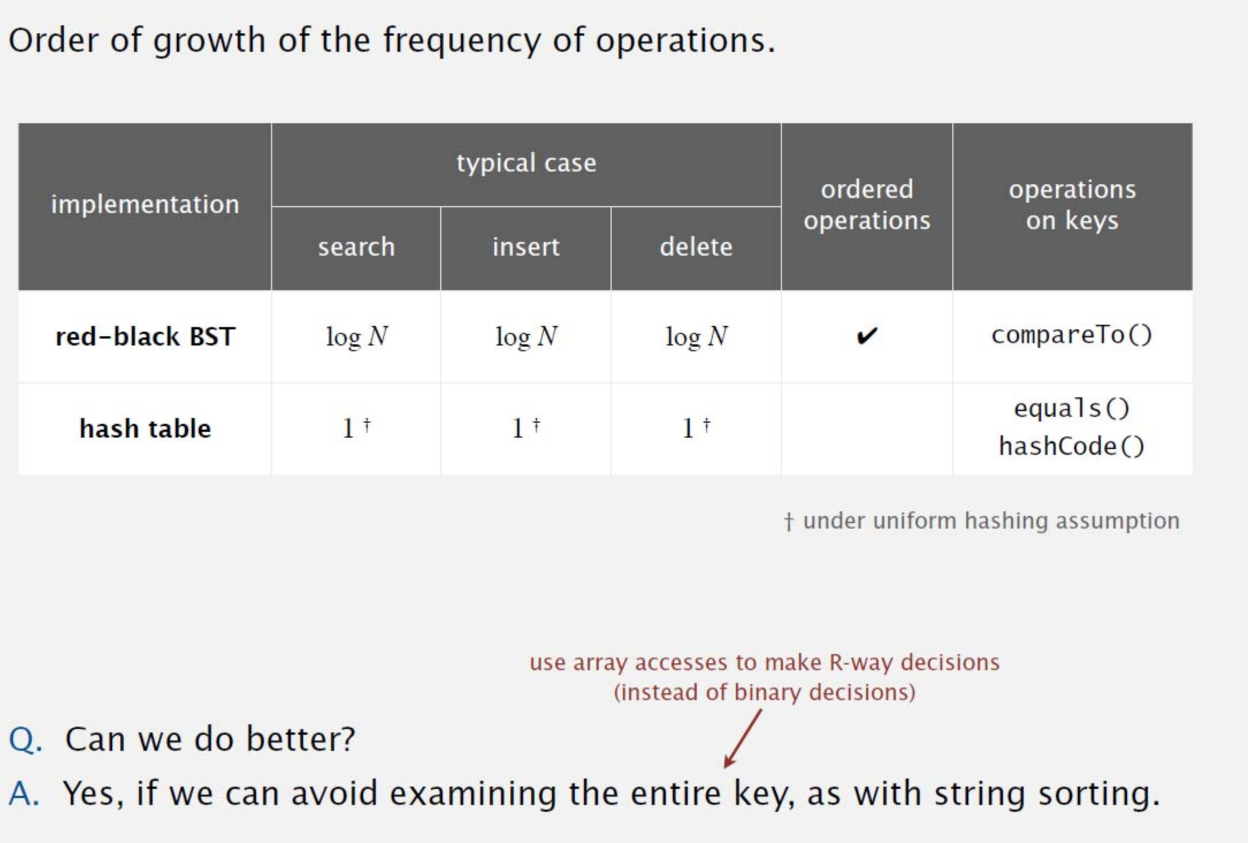
- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

Symbol Table summary so far



implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N^{*}$	$1.0 \lg N^{*}$	$1.0 \lg N^{*}$	✔	`compareTo()`

Symbol Table summary so far



Order of growth of the frequency of operations.

implementation	typical case			ordered operations	operations on keys
	search	insert	delete		
red-black BST	$\log N$	$\log N$	$\log N$	✔	`compareTo()`

hash table	1 †	1 †	1 †		`equals()`
					`hashCode()`

† under uniform hashing assumption

Q. Can we do better?

use array accesses to make R-way decisions
(instead of binary decisions)

A. Yes, if we can avoid examining the entire key, as with string sorting.

Tries

- Data structure for searching with string keys
- From word “retrieval”, but read as “try” to be different than “tree”

Symbol Tables

- Generic ST
- https://algs4.cs.princeton.edu/35applications/ST.java

```
public class ST<Key extends Comparable<Key>, Value>
implements Iterable<Key> {
        private TreeMap<Key, Value> st;
        public void put(Key key, Value val) {…}
        public void delete(Key key) {…}
        public Value get(Key key) {…}
}
```

Symbol Tables

- StringST

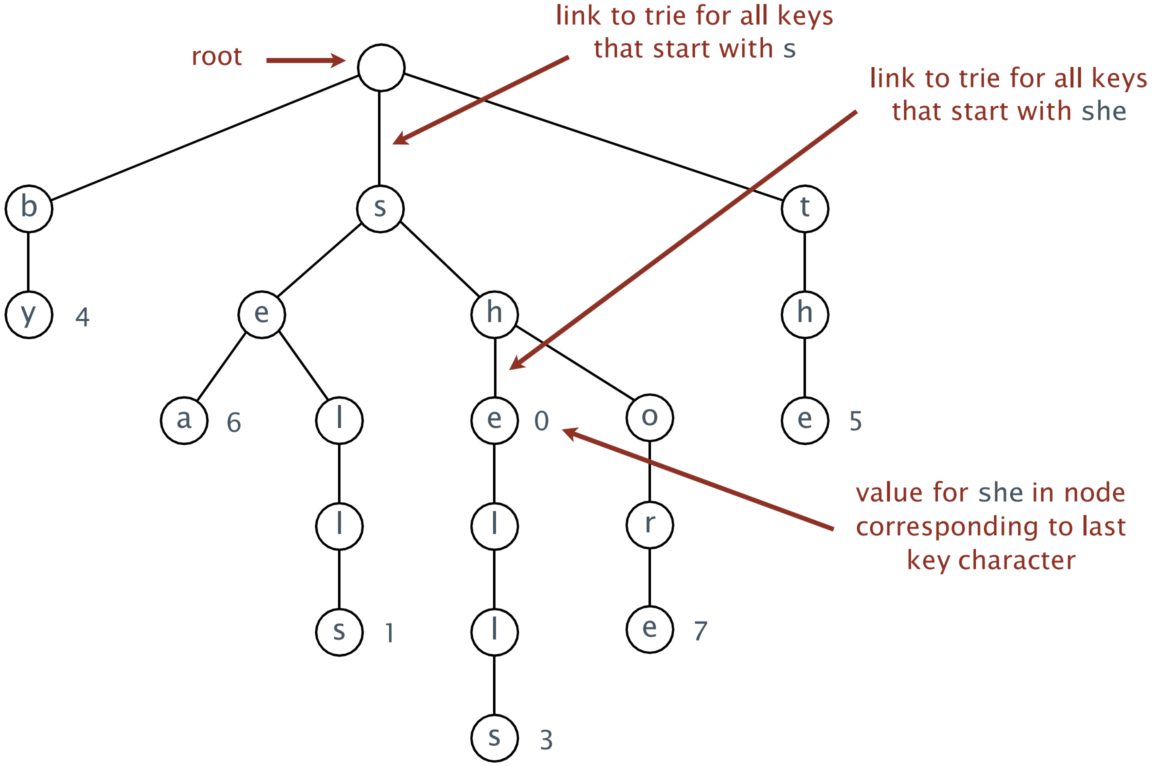
```
public class StringST<Value> {
        public void put(String key, Value val) {…}
        public void delete(String key) {…}
        public Value get(String key) {…}
}
```

Tries

Tries.

- Store characters in nodes (not keys).
- Each node has R children, one for each possible character.
  (for now, we do not draw null links)

key	value
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5



link to trie for all keys that start with s

link to trie for all keys that start with she

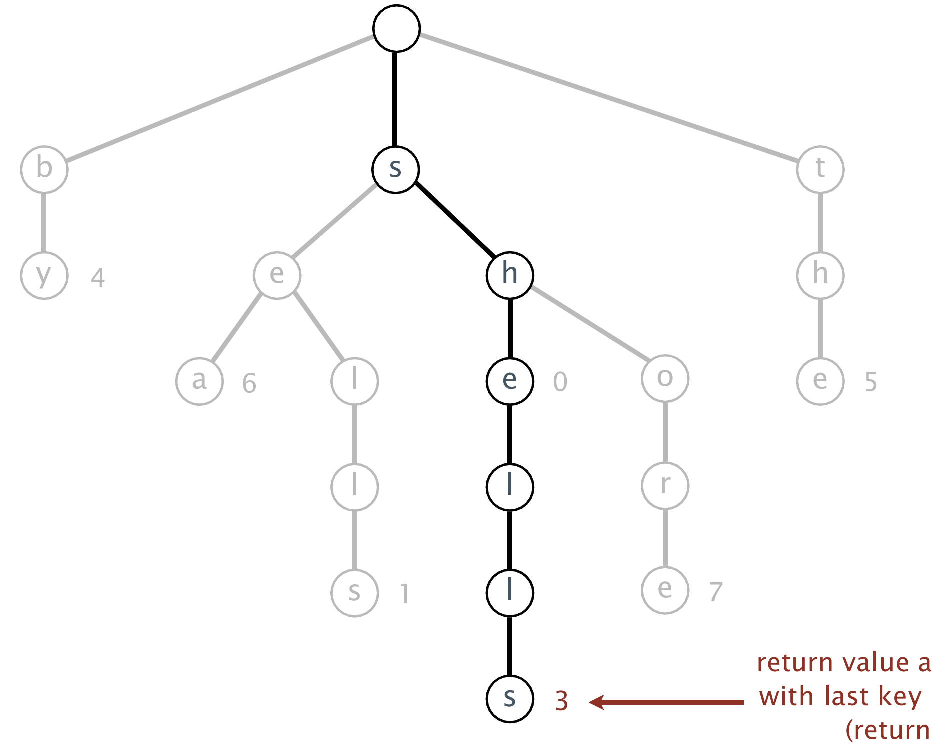
value for she in node corresponding to last key character

Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.

get("shells")



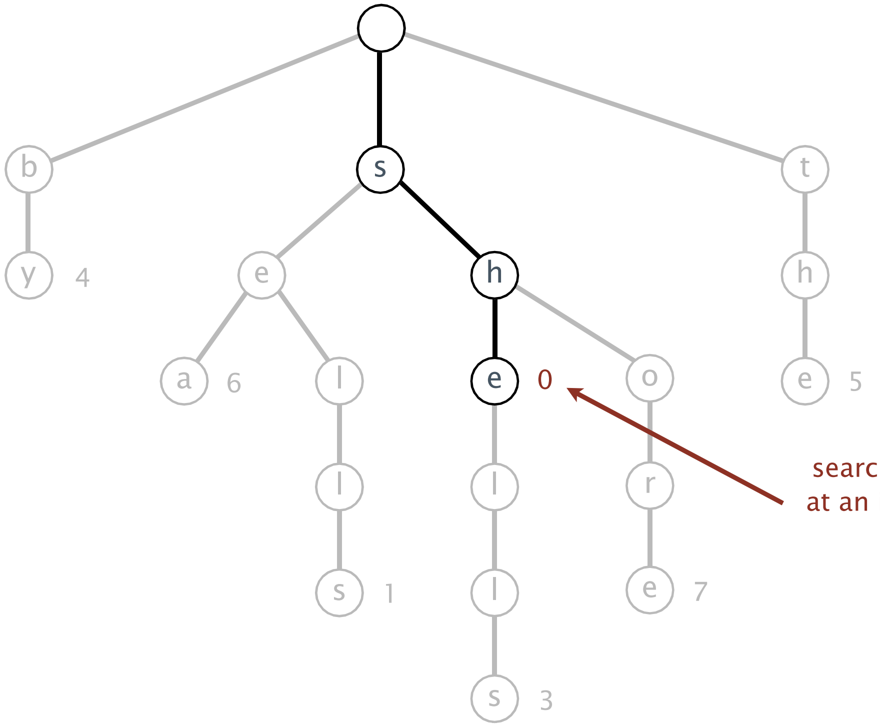
return value associated with last key character (return 3)

Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.

get("she")



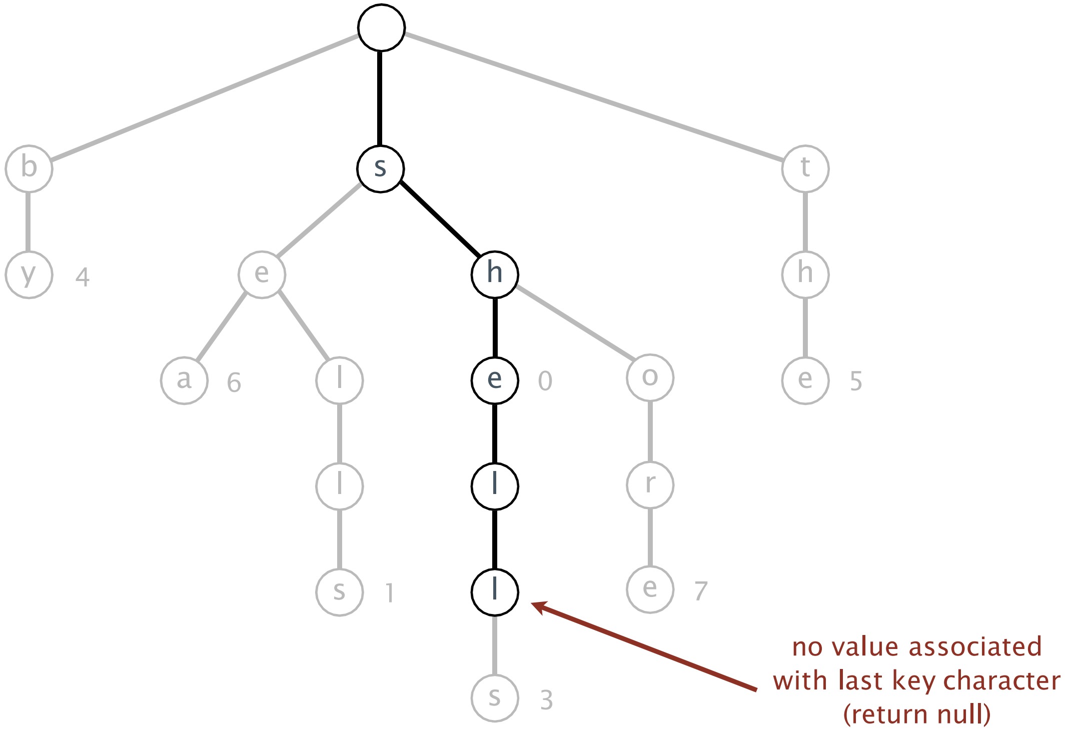
search may terminated at an intermediate node (return 0)

Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.

get("shell")



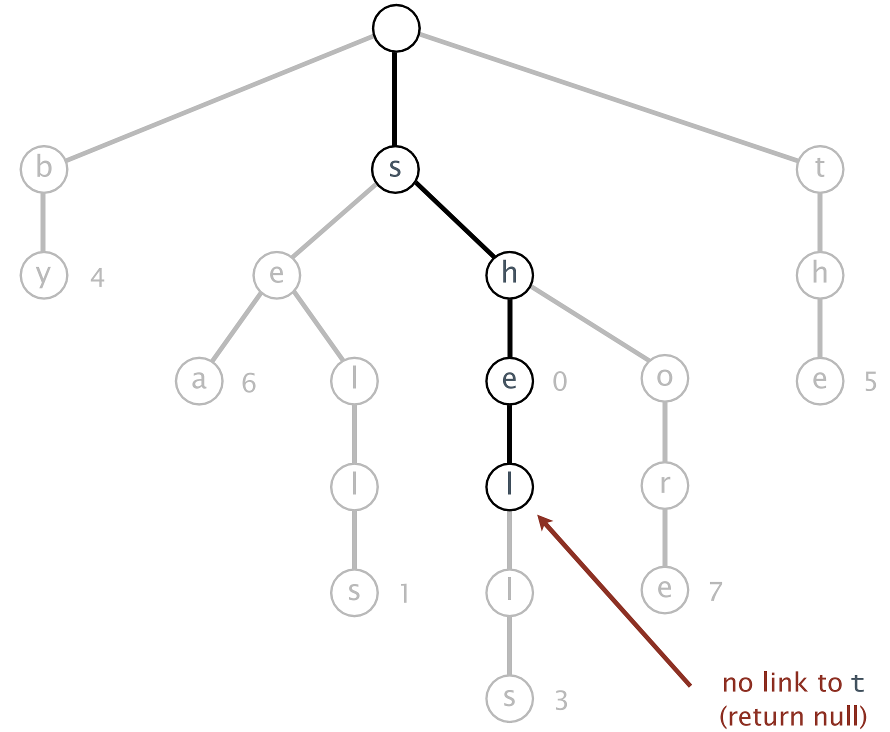
no value associated with last key character (return null)

Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.

get("shelter")

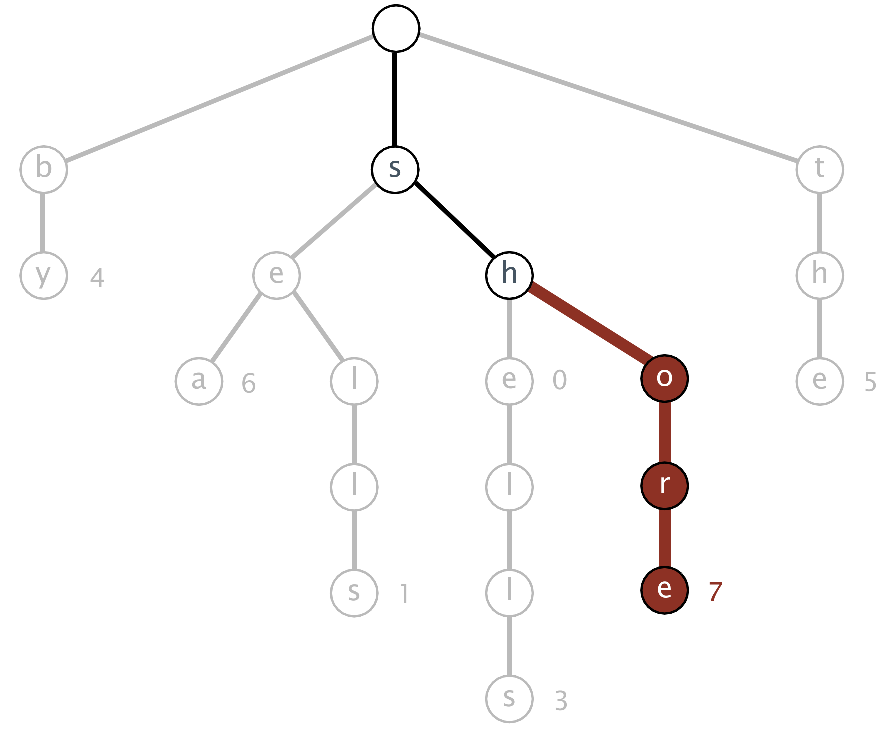


Insertion into a trie

Follow links corresponding to each character in the key.

- Encounter a null link: create new node.
- Encounter the last character of the key: set value in that node.

put("shore", 7)



Trie construction demo

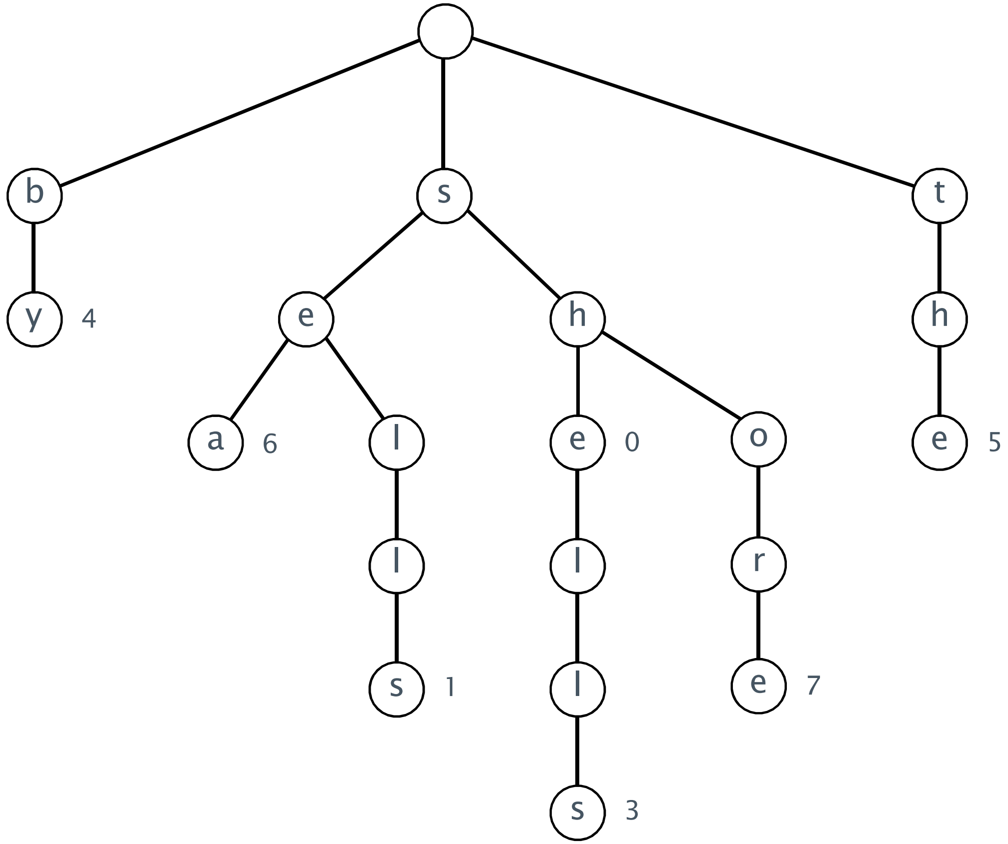
Algorithms ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

5.2 TRIE CONSTRUCTION DEMO

Trie construction demo

trie

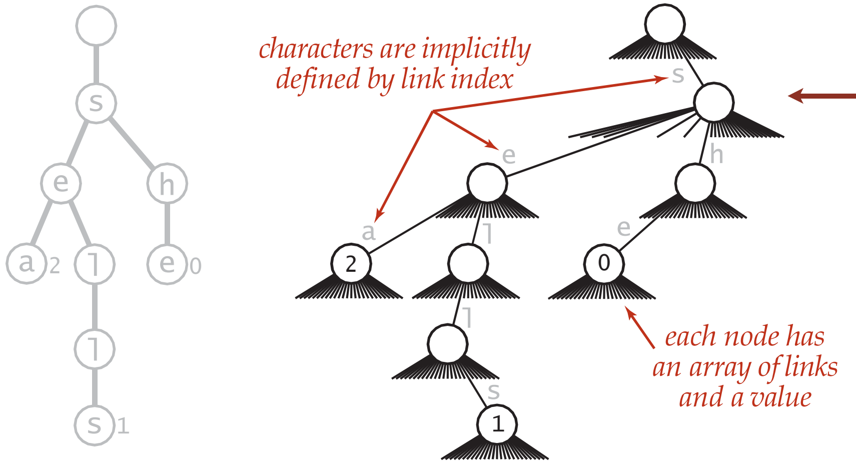


Trie representation: Java implementation

Node. A value, plus references to R nodes.

```
private static class Node
{
   private Object value;
   private Node[] next = new Node[R];
}
```

use Object instead of Value since no generic array creation in Java



characters are implicitly defined by link index

each node has an array of links and a value

neither keys nor characters are explicitly stored

R-way trie: Java implementation

```
public class TrieST<Value>
{
   private static final int R = 256;
   private Node root = new Node();

   private static class Node
   {  /* see previous slide */  }

   public void put(String key, Value val)
   {  root = put(root, key, val, 0);  }

   private Node put(Node x, String key, Value val, int d)
   {
      if (x == null) x = new Node();
      if (d == key.length()) { x.val = val; return x; }
      char c = key.charAt(d);
      x.next[c] = put(x.next[c], key, val, d+1);
      return x;
   }

   ⋮
```

extended ASCII

R-way trie: Java implementation (continued)

```
   ⋮
   public boolean contains(String key)
   {  return get(key) != null;  }

   public Value get(String key)
   {
      Node x = get(root, key, 0);
      if (x == null) return null;
      return (Value) x.val;
   }

   private Node get(Node x, String key, int d)
   {
      if (x == null) return null;
      if (d == key.length()) return x;
      char c = key.charAt(d);
      return get(x.next[c], key, d+1);
   }

}
```

cast needed

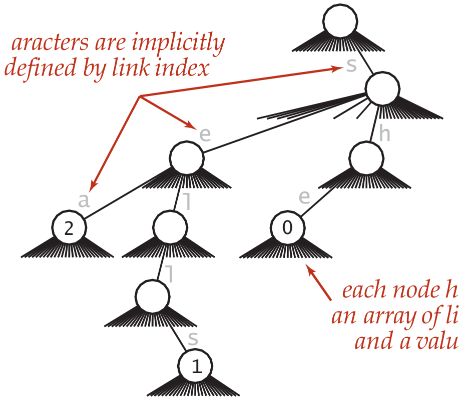
Trie performance

Search hit. Need to examine all L characters for equality.

Search miss.

- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

Space. R null links at each leaf.
(but sublinear space possible if many short strings share common prefixes)



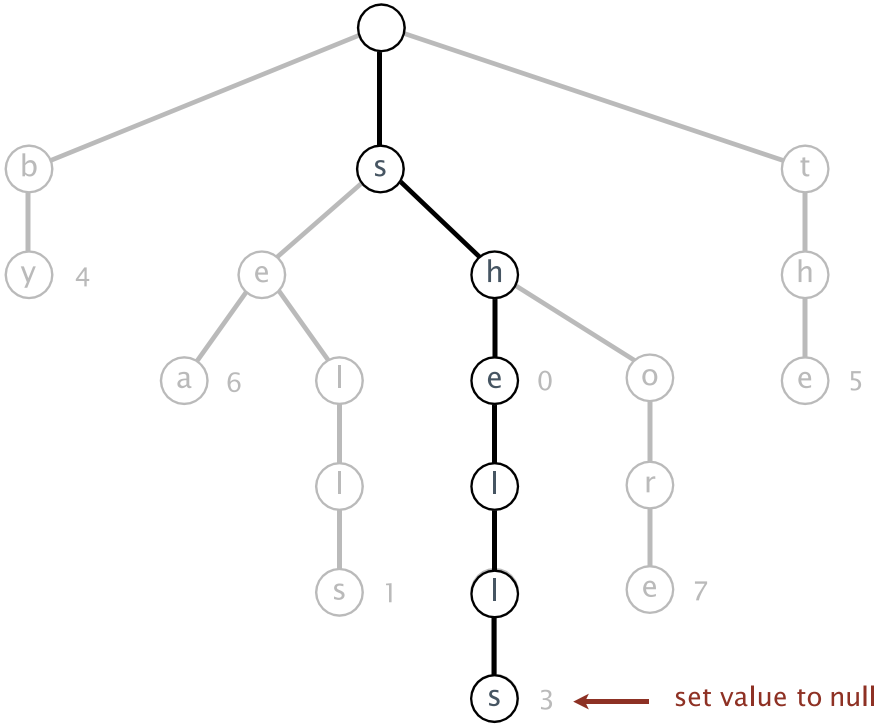
Bottom line. Fast search hit and even faster search miss, but wastes space.

Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")



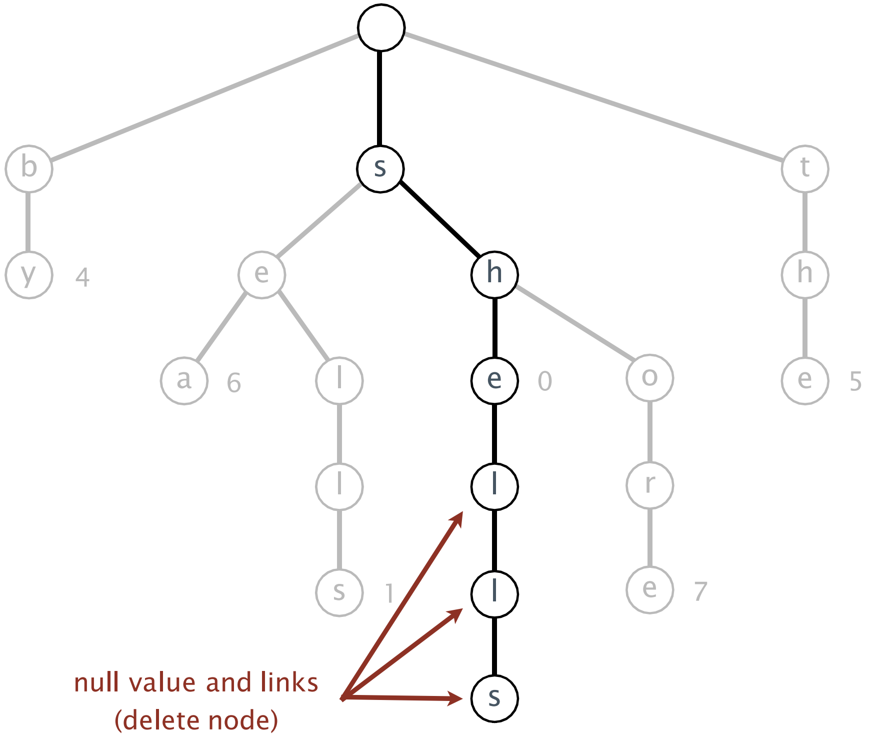
set value to null

Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")

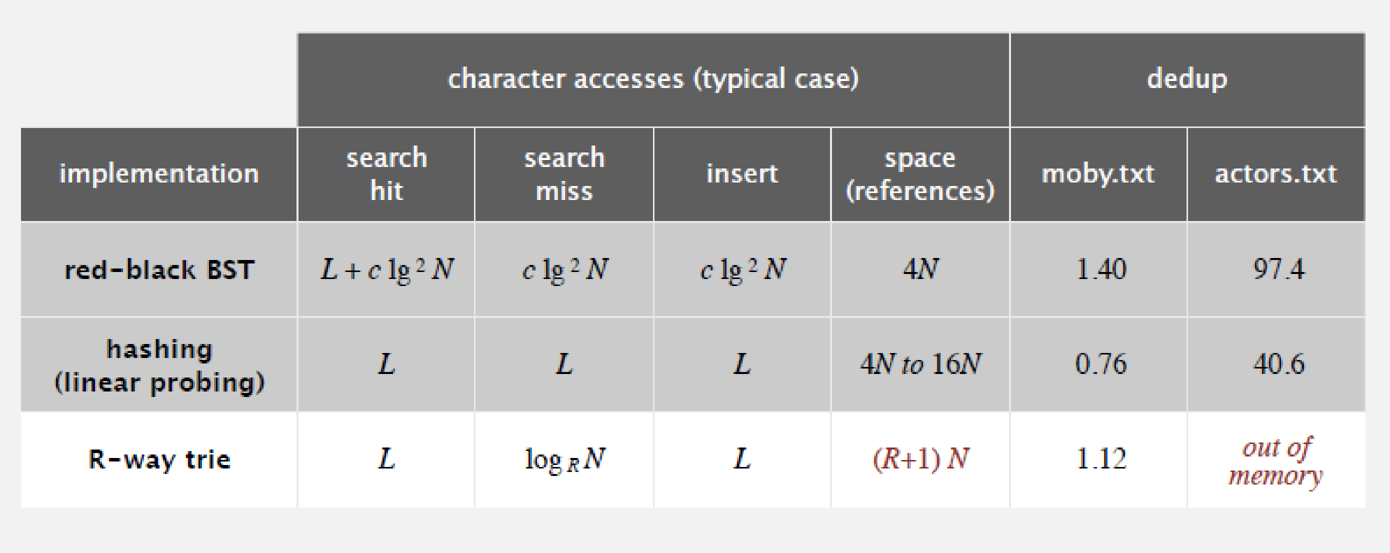


Trie exercise

Key	value
bed	50
better	3
backend	30
backup	1
auto	1
test	3
summary	5
sum	40
end	3
blah	3

- Construct a trie with following key-value pairs
- Assuming 26-digit radix (lower case letters), think about how many null links does the trie have?
- Is there a more optimal way to do this?

String symbol table implementations cost summary



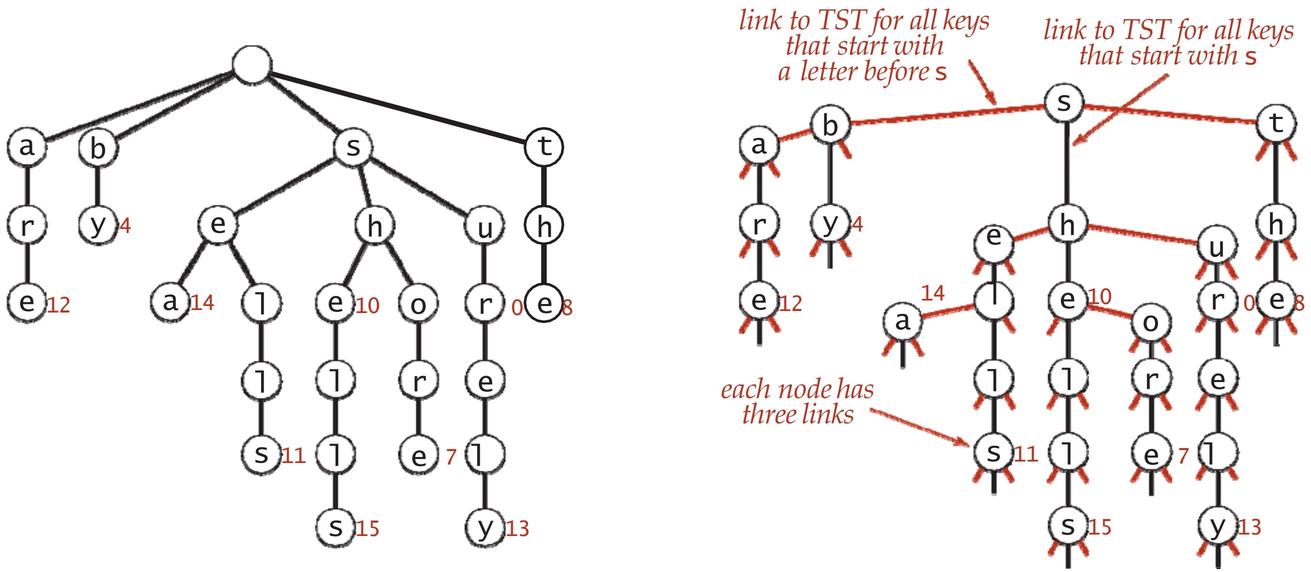
	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N$ *to* $16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)\,N$	1.12	*out of memory*

- Too much memory for large R.

Ternary Search Tries (TSTs)

Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has 3 children: smaller (left), equal (middle), larger (right).

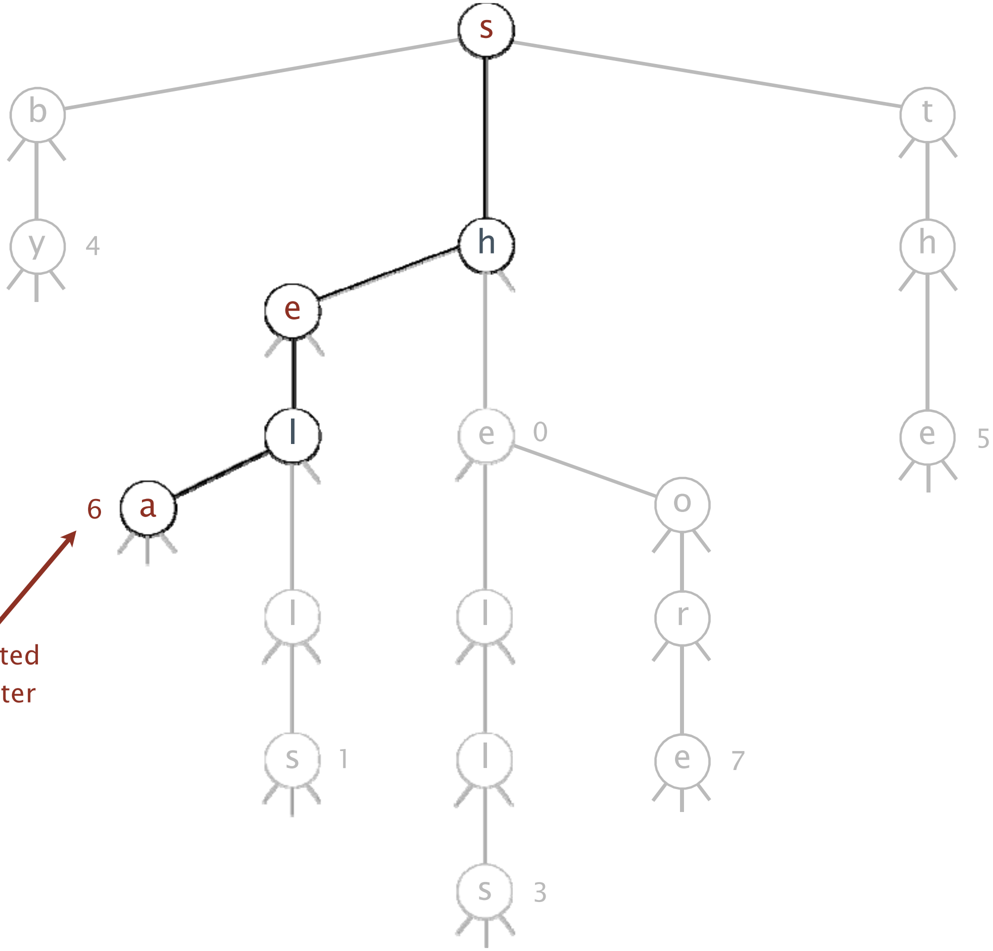


a
b
s
t
r
y
4
e
h
u
h
e
12
a
14
l
e
10
o
r
0
e
8
l
l
r
e
s
11
l
e
7
l
s
15
y
13
link to TST for all keys that start with a letter before s
link to TST for all keys that start with s
each node has three links

TST representation of a trie

Search hit in a TST

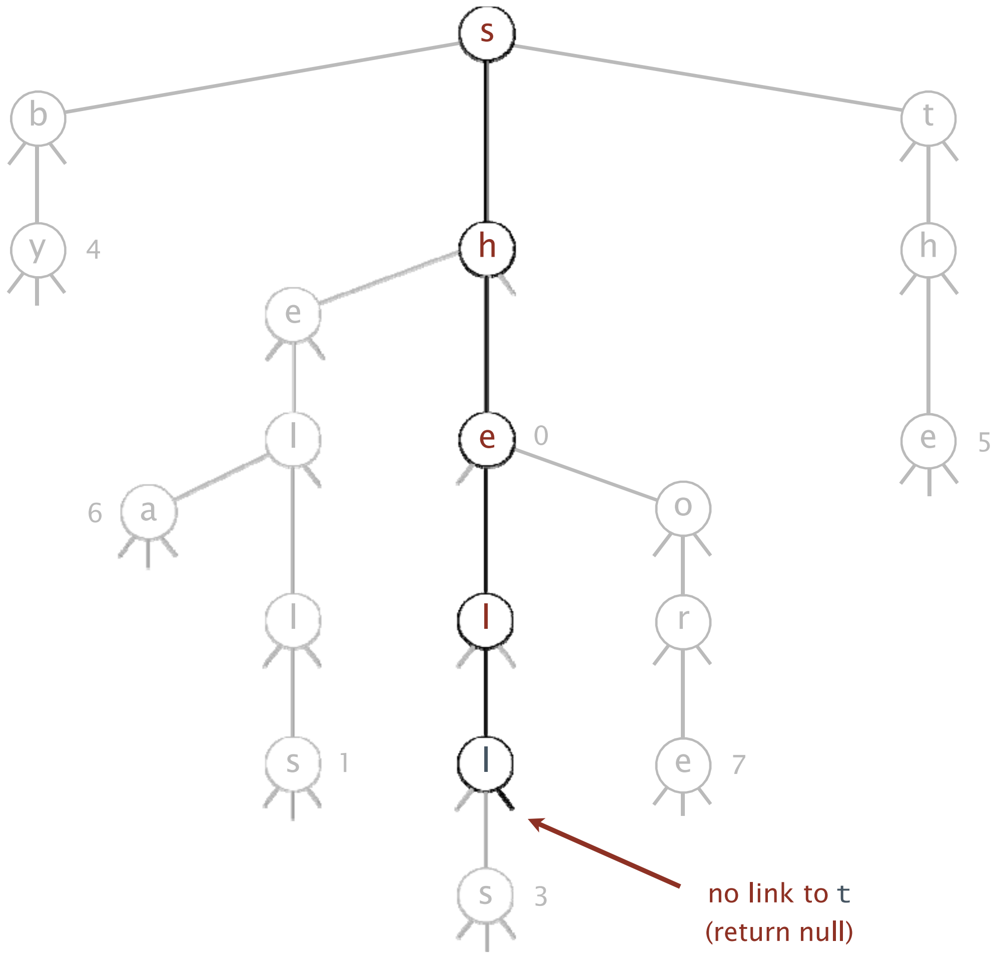
get("sea")



return value associated with last key character

Search miss in a TST

get("shelter")



no link to t
(return null)

TST construction demo

Algorithms

Robert Sedgewick | Kevin Wayne

Algorithms

Fourth Edition

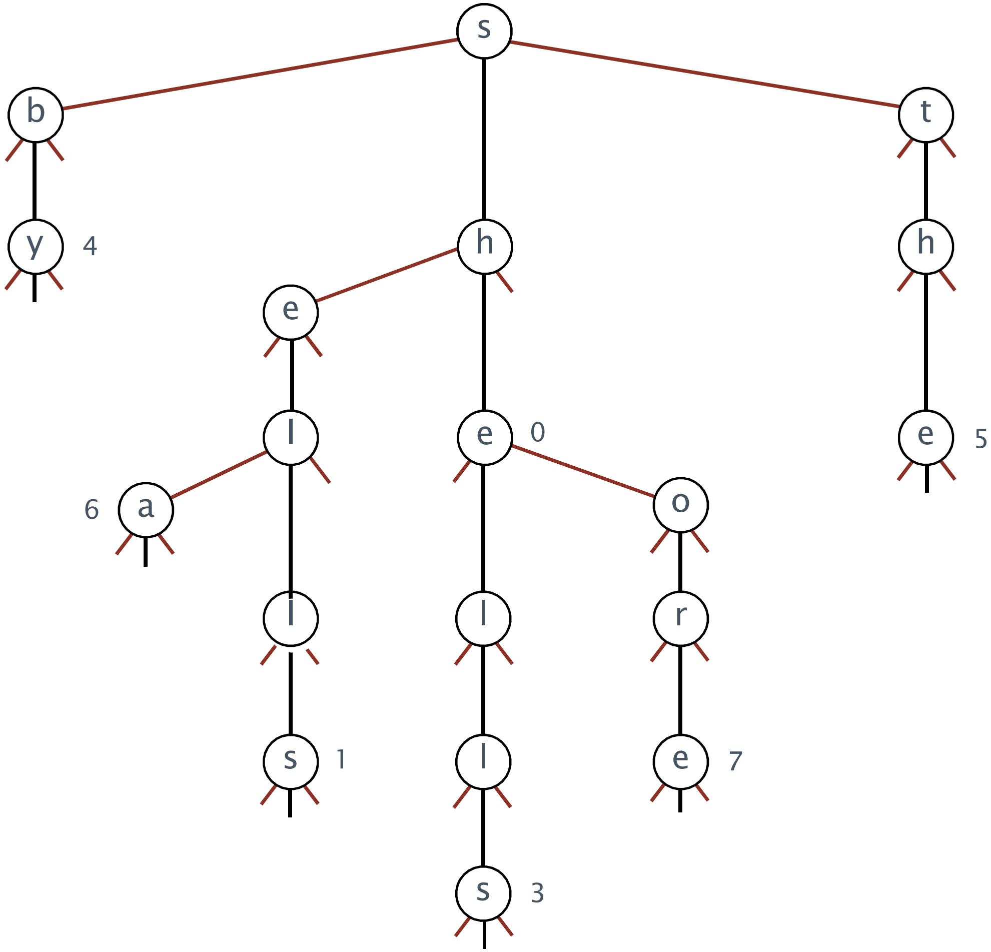
Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

5.2 TST Construction Demo

Ternary search trie construction demo

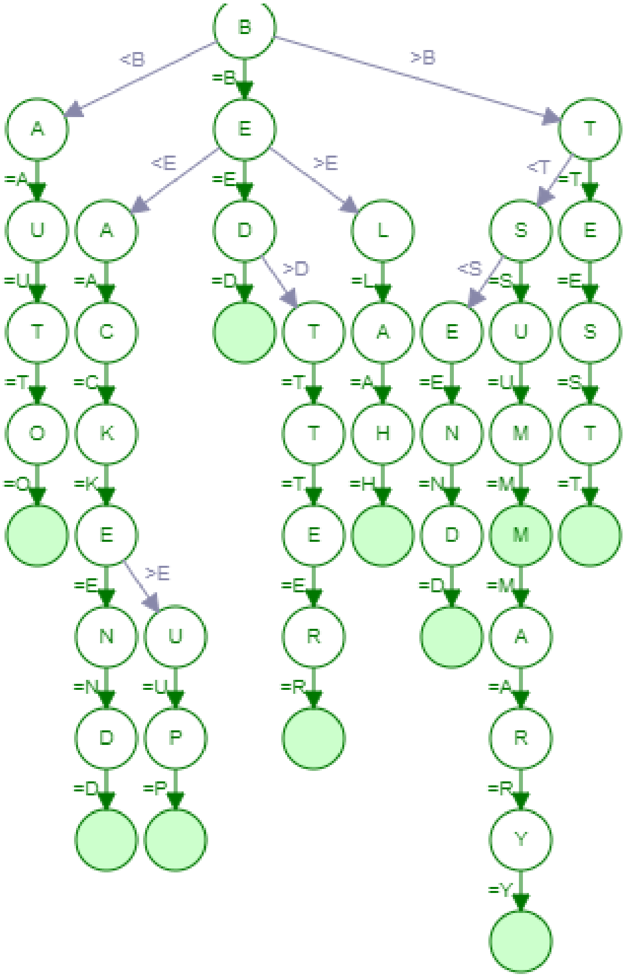
ternary search trie



Demo – build your own

- https://www.cs.usfca.edu/~galles/visualization/TST.html

Key	value
bed	50
better	3
backend	30
backup	1
auto	1
test	3
summary	5
sum	40
end	3
blah	3



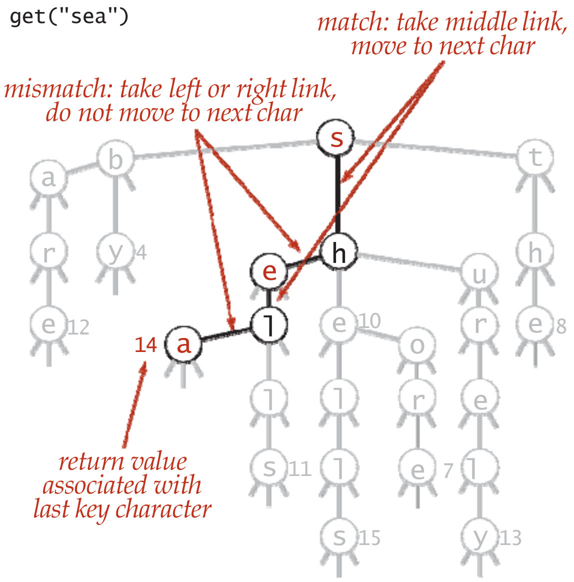
Search in a TST

Follow links corresponding to each character in the key.

- If less, take left link; if greater, take right link.
- If equal, take the middle link and move to the next key character.

Search hit. Node where search ends has a non-null value.

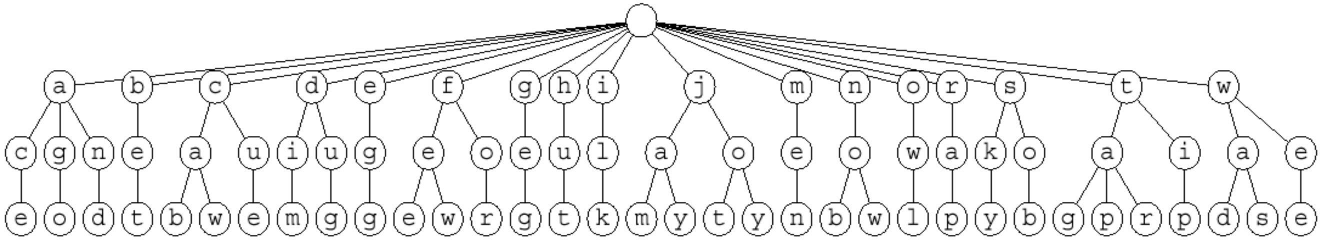
Search miss. Reach a null link or node where search ends has null value.



get("sea")
match: take middle link, move to next char
mismatch: take left or right link, do not move to next char
return value associated with last key character
s
b
a
t
r
y
4
h
e
u
h
e
12
14
a
l
e
10
o
r
e
8
l
l
r
e
s
11
l
e
7
l
s
15
y
13

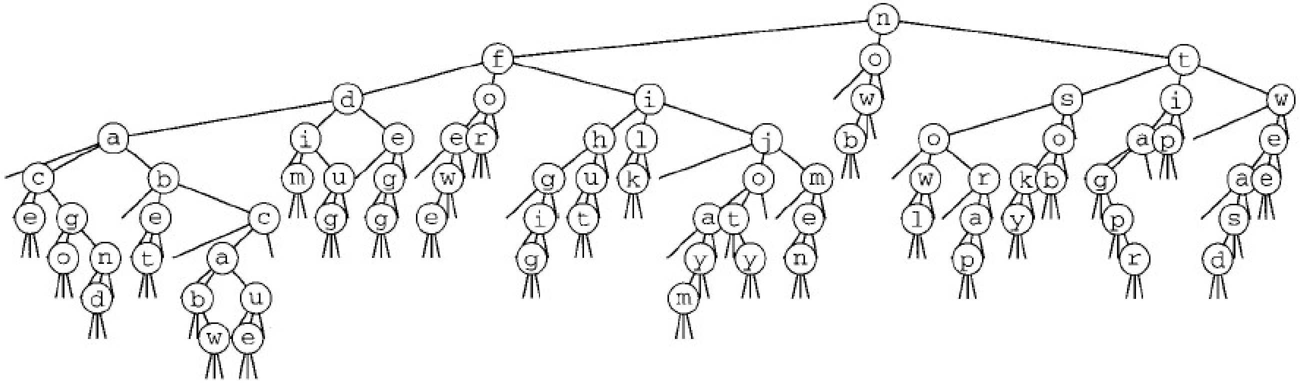
26-way trie vs. TST

26-way trie. 26 null links in each leaf.


a b c d e f g h i j m n o r s t w
c g n e a u i u g e o e u l a o e o w a k o a i a e
e o d t b w e m g g e w r g t k m y t y n b w l p y b g p r p d s e

26-way trie (1035 null links, not shown)

TST. 3 null links in each leaf.



TST (155 null links)

now
for
tip
ilk
dim
tag
jot
sob
nob
sky
hut
ace
bet
men
egg
few
jay
owl
joy
rap
gig
wee
was
cab
wad
caw
cue
fee
tap
ago
tar
jam
dug
and

TST java implementation

- Assignment 2

String symbol table implementation cost summary

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N$ *to* $16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)\,N$	1.12	*out of memory*
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4\,N$	0.72	38.7

TST vs. hashing

Hashing.

- Need to examine entire key.
- Search hits and misses cost about the same.
- Performance relies on hash function.
- Does not support ordered symbol table operations.

TSTs.

- Works only for string (or digital) keys.
- Only examines just enough key characters.
- Search miss may involve only a few characters.
- Supports ordered symbol table operations (plus extras!).

Bottom line. TSTs are:

- Faster than hashing (especially for search misses).
- More flexible than red-black BSTs
  - supports character-based operations

String symbol table API

Character-based operations. The string symbol table API supports several useful character-based operations.

key	value
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5

Prefix match. Keys with prefix `sh`: `she`, `shells`, and `shore`.

Wildcard match. Keys that match `.he`: `she` and `the`.

Longest prefix. Key that is the longest prefix of `shellsort`: `shells`.

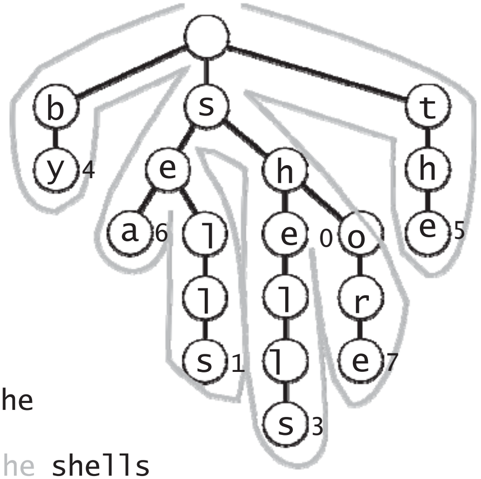
Ordered iteration

To iterate through all keys in sorted order:

- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.

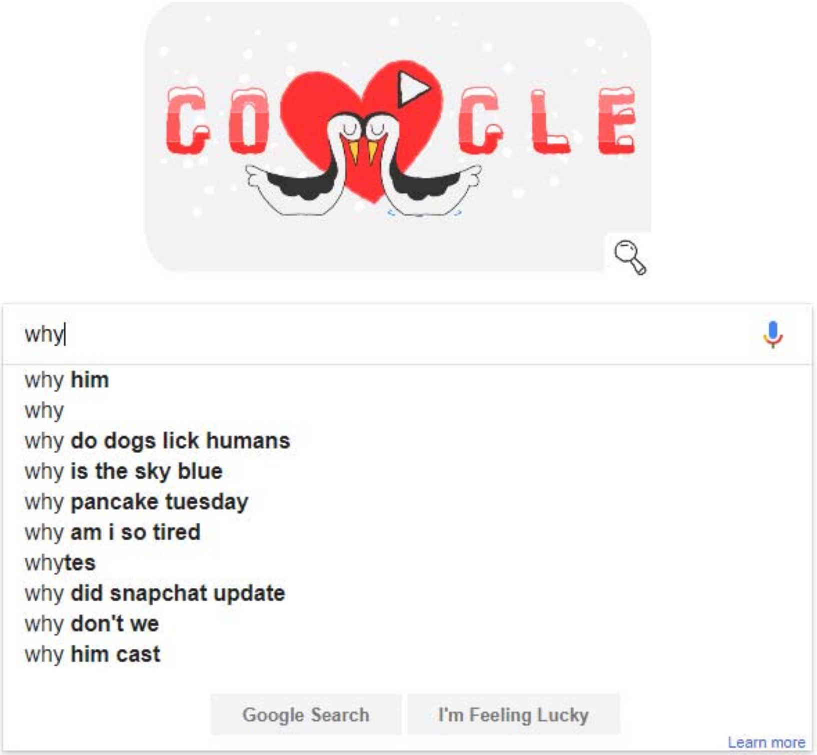
```
keys()
```

key	q
b	
by	by
s	
se	
sea	by sea
sel	
sell	
sells	by sea sells
sh	
she	by sea sells she
shell	
shells	by sea sells she shells
sho	
shor	
shore	by sea sells she shells shore
t	
th	
the	by sea sells she shells shore the



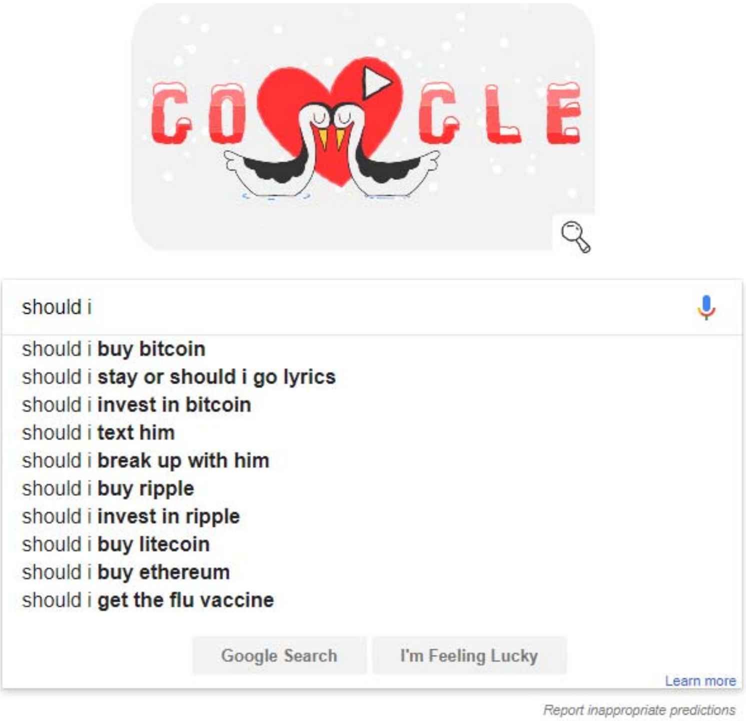
Applications -prefix matches

› Eg dropdown lists



why

why **him**
why
why **do dogs lick humans**
why **is the sky blue**
why **pancake tuesday**
why **am i so tired**
why**tes**
why **did snapchat update**
why **don't we**
why **him cast**



should i

should i **buy bitcoin**
should i **stay or should i go lyrics**
should i **invest in bitcoin**
should i **text him**
should i **break up with him**
should i **buy ripple**
should i **invest in ripple**
should i **buy litecoin**
should i **buy ethereum**
should i **get the flu vaccine**

Applications – prefix match

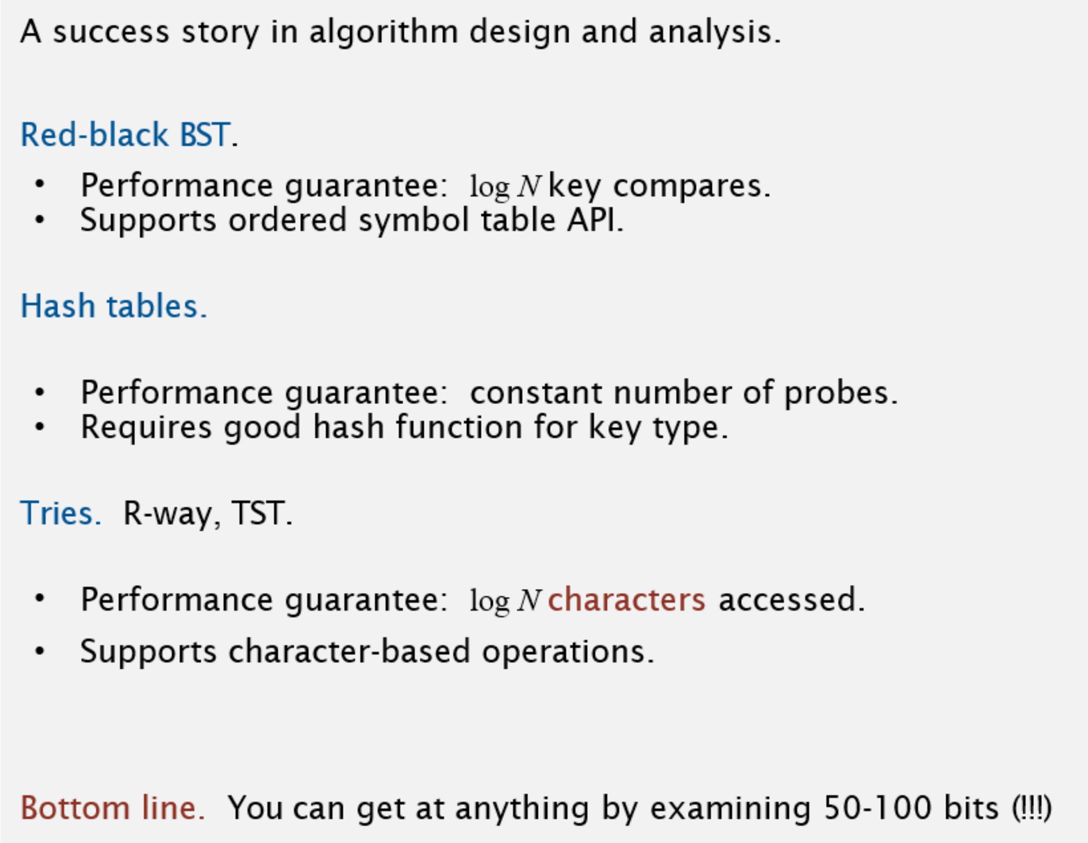
Ex. To send packet toward destination IP address, router chooses IP address in routing table that is longest prefix match.

```
"128"
"128.112"
"128.112.055"
"128.112.055.15"
"128.112.136"
"128.112.155.11"
"128.112.155.13"
"128.222"
"128.222.136"
```

represented as 32-bit binary number for IPv4 (instead of string)

```
longestPrefixOf("128.112.136.11") = "128.112.136"
longestPrefixOf("128.112.100.16") = "128.112"
longestPrefixOf("128.166.123.45") = "128"
```

Symbol tables summary



A success story in algorithm design and analysis.

Red-black BST.

- Performance guarantee: $\log N$ key compares.
- Supports ordered symbol table API.

Hash tables.

- Performance guarantee: constant number of probes.
- Requires good hash function for key type.

Tries. R-way, TST.

- Performance guarantee: $\log N$ characters accessed.
- Supports character-based operations.

Bottom line. You can get at anything by examining 50-100 bits (!!!)