Symbol Tables

- API
- sequential search
- binary search
- BSTs
- ordered operations
- deletion in BSTs

Symbol tables

Key-value pair abstraction.

- Insert a value with specified key.
- Given a key, search for the corresponding value.

Ex. DNS lookup.

- Insert URL with specified IP address.
- Given URL, find corresponding IP address.

URL	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

key ↑ (URL) value ↑ (IP address)

Symbol table applications

application	purpose of search	key	value
dictionary	look up word	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
financial account	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find properties of variables	variable name	value and type
routing table	route Internet packets	destination	best route
DNS	find IP address given URL	URL	IP address
reverse DNS	find URL given IP address	IP address	URL
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

Symbol table API

Associative array abstraction. Associate one value with each key.

`public class ST<Key, Value>`

	`ST()`	*create a symbol table*
`void`	`put(Key key, Value val)`	*put key-value pair into the symbol table (remove key from table if value is null)* ← `a[key] = val;`
`Value`	`get(Key key)`	*value paired with key (null if key is absent)* ← `a[key]`
`void`	`delete(Key key)`	*remove key (and its value) from table*
`boolean`	`contains(Key key)`	*is there a value paired with key?*
`boolean`	`isEmpty()`	*is the table empty?*
`int`	`size()`	*number of key-value pairs in the table*
`Iterable<Key>`	`keys()`	*all the keys in the symbol table*

API for a generic basic symbol table

Conventions

- Values are not `null`.
- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.

Intended consequences.

- Easy to implement `contains()`.

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

- Can implement lazy version of `delete()`.

```
public boolean delete(Key key)
{  put(key, null);            }
```


Keys and values

Value type. Any generic type.

Key type: several natural assumptions.

- Assume keys are `Comparable`, use `compareTo()`.
- Assume keys are any generic type, use `equals()` to test equality.
- Assume keys are any generic type, use `equals()` to test equality and `hashCode()` to scramble key.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: `String`, `Integer`, `BigInteger`, ...
- Mutable in Java: `Date`, `GregorianCalendar`, `StringBuilder`, ...

ST test client for traces

Build ST by associating value i with ith command-line argument.

```
public static void main(String[] args)
{
   ST<String, Integer> st = new ST<String, Integer>();
   for (int i = 0; i < args.length; i++)
      st.put(args[i], i);
   for (String s : st)
      StdOut.println(s + " " + st.get(s));
}
```

keys	S	E	A	R	C	H	E	X	A	M	P	L	E
values	0	1	2	3	4	5	6	7	8	9	10	11	12

output

```
A  8
C  4
E  12
H  5
L  9
M  11
P  10
R  3
S  0
X  7
```


ST test client for analysis

Frequency Counter.
Read a sequence of strings from standard input and print out the number of times each string appears.

```
% more tiny.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness

% java FrequencyCounter 0 < tiny.txt
2 age
1 best
1 foolishness
4 it
4 of
4 the
2 times
4 was
1 wisdom
1 worst
```

tiny example
24 words
10 distinct

```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
...

% java FrequencyCounter 0 < tale.txt
2941 a
1 aback
1 abandon
10 abandoned
1 abandoning
1 abandonment
1 abashed
1 abate
1 abated
...
```

real example
137177 words
9888 distinct

Frequency counter implementation

```
public class FrequencyCounter
{
   public static void main(String[] args)
   {
      int minlen = Integer.parseInt(args[0]);
      ST<String, Integer> st = new ST<String, Integer>();      // create ST
      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();                      // ignore short strings
         if (word.length() < minlen) continue;                  // read string and
         if (!st.contains(word)) st.put(word, 1);               // update frequency
         else                    st.put(word, st.get(word) + 1);
      }
      String max = "";
      for (String word : st.keys())                              // print all strings
         if (st.get(word) > st.get(max))
            max = word;
      StdOut.println(max + " " + st.get(max));
   }
}
```

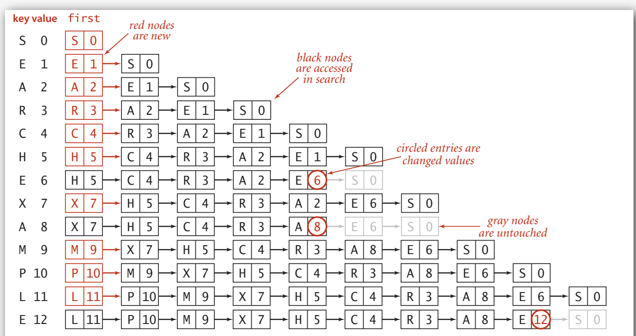

- API
- **sequential search**
- binary search
- BSTs
- applications

Sequential search in a linked list

Data structure. Maintain an (unordered) linked list of key-value pairs.

Search. Scan through all keys until find a match.

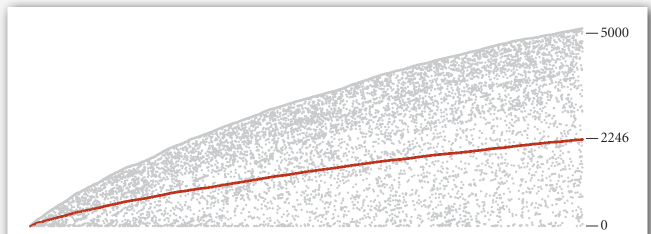
Insert. Scan through all keys until find a match; if no match add to front.



Trace of linked-list ST implementation for standard indexing client

Elementary ST implementations: summary

ST implementation	worst case search	worst case insert	average case search hit	average case insert	ordered iteration?	operations on keys
sequential search (unordered list)	N	N	$N / 2$	N	no	`equals()`



Costs for `java FrequencyCounter 8 < tale.txt` using `LinkedListST`

Challenge. Efficient implementations of both search and insert.

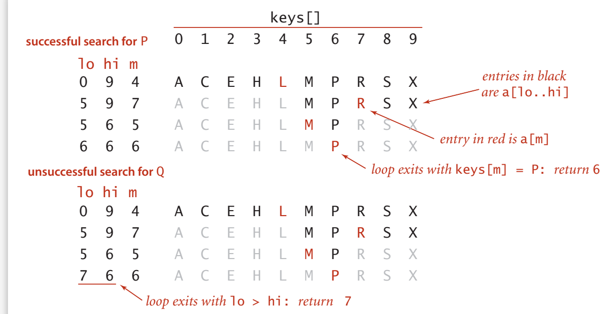
- API
- sequential search
- **binary search**
- BSTs
- applications

Binary search

Data structure. Maintain an ordered array of key-value pairs.

Search. Binary search.

Insert. Binary search for key; if no match insert and shift larger keys.



Trace of binary search for rank in an ordered array

Binary search: Java implementation

```
public Value get(Key key)
{
   int i = bsearch(key);
   if (i == -1) return null;
   return vals[i];
}
```

symbol table method

```
private int bsearch(Key key)
{
   int lo = 0, hi = N-1;
   while (lo <= hi)
   {
       int m = lo + (hi - lo) / 2;
       int cmp = key.compareTo(keys[m]);
       if      (cmp  < 0) hi = m - 1;
       else if (cmp  > 0) lo = m + 1;
       else if (cmp == 0) return m;
  }
  return -1;
}
```

helper binary search method

not found

Binary search: mathematical analysis

Proposition. Binary search uses $\sim \lg N$ compares to search any array of size N.

Def. $T(N) \equiv$ number of compares to binary search in a sorted array of size N.

$$\leq \underbrace{T(N/2)}_{\text{left or right half}} + 1$$

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

- Not quite right for odd N.
- Same recurrence holds for many algorithms.

Solution. $T(N) \sim \lg N$.

- For simplicity, we'll prove when N is a power of 2.
- True for all N. [see COS 340]

Binary search recurrence

Binary search recurrence. $T(N) \le T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

Proposition. If N is a power of 2, then $T(N) \le \lg N + 1$.

Pf.

$T(N) \le T(N/2) + 1$	given
$\le T(N/4) + 1 + 1$	apply recurrence to first term
$\le T(N/8) + 1 + 1 + 1$	apply recurrence to first term
...	
$\le T(N/N) + 1 + 1 + \ldots + 1$	stop applying, T(1) = 1
$= \lg N + 1$	

Binary search: trace of standard indexing client

Problem. To insert, need to shift all greater keys over.

key	value	keys[] 0	1	2	3	4	5	6	7	8	9	N	vals[] 0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

entries in red were inserted

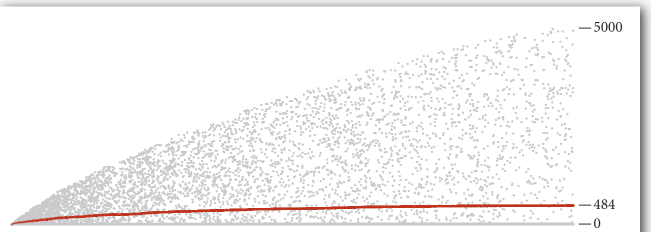
entries in gray did not move

entries in black moved to the right

circled entries are changed values

Elementary ST implementations: summary

ST implementation	worst case		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	$N/2$	N	no	`equals()`
binary search (ordered array)	$\log N$	N	$\log N$	$N/2$	yes	`compareTo()`



Costs for `java FrequencyCounter 8 < tale.txt` using `OrderedArrayST`

Challenge. Efficient implementations of both search and insert.

- API
- sequential search
- binary search
- **challenges**

Searching challenge 1A

Problem. Maintain symbol table of song names for an iPod.
Assumption A. Hundreds of songs.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 1B

Problem. Maintain symbol table of song names for an iPod.
Assumption B. Thousands of songs.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 2A:

Problem. IP lookups in a web monitoring device.
Assumption A. Billions of lookups, millions of distinct addresses.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 2B

Problem. IP lookups in a web monitoring device.
Assumption B. Billions of lookups, thousands of distinct addresses.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 3

Problem. Frequency counts in "Tale of Two Cities."
Assumptions. Book has 135,000+ words; about 10,000 distinct words.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 4

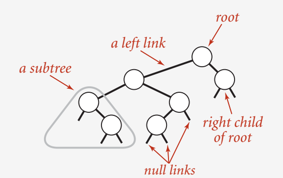
Problem. Spell checking for a book.
Assumptions. Dictionary has 25,000 words; book has 100,000+ words.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Binary search trees

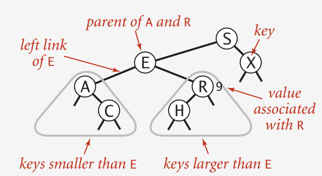
Def. A BST is a binary tree in symmetric order.

A binary tree is either:
- Empty.
- Two disjoint binary trees (left and right).



Anatomy of a binary tree

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.



Anatomy of a binary search tree

BST representation in Java

A BST is a reference to a root node.

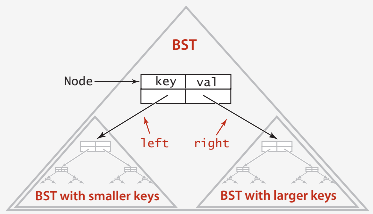
A `Node` is comprised of four fields:

- A `Key` and a `Value`.
- A reference to the left and right subtree.

smaller keys   larger keys

```
private class Node
{
   private Key key;
   private Value val;
   private Node left, right;
   public Node(Key key, Value val)
   {
      this.key = key;
      this.val = val;
   }
}
```

`Key` and `Value` are generic types; `Key` is `Comparable`



Binary search tree

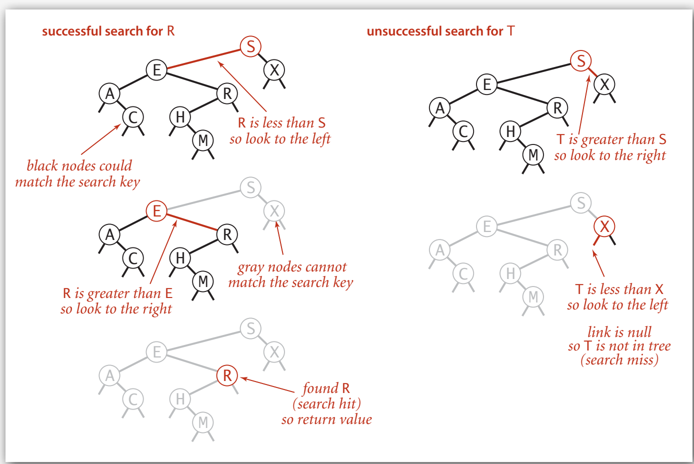
BST implementation (skeleton)

```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;

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

   public void put(Key key, Value val)
   {  /* see next slides */  }

   public Value get(Key key)
   {  /* see next slides */  }

   public void delete(Key key)
   {  /* see next slides */  }

   public Iterable<Key> iterator()
   {  /* see next slides */  }

}
```

root of BST

BST search

Get. Return value corresponding to given key, or `null` if no such key.



BST search: Java implementation

Get. Return value corresponding to given key, or `null` if no such key.

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

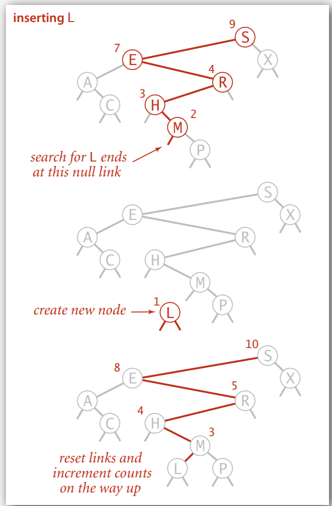
Running time. Proportional to depth of node.

BST insert

Put. Associate value with key.

Search for key, then two cases:

- key in tree: reset value
- key not in tree: add new node

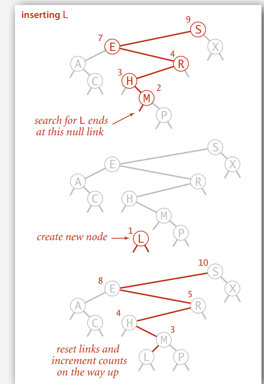


BST insert: Java implementation

Put. Associate value with key.

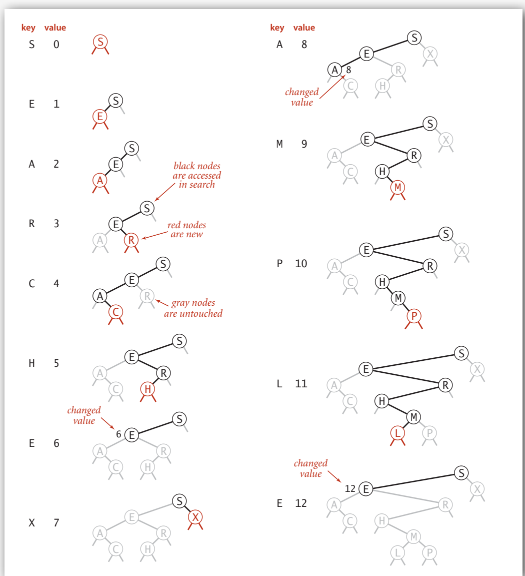
concise, but tricky, recursive code; read carefully!

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0)
      x.left  = put(x.left,  key, val);
   else if (cmp  > 0)
      x.right = put(x.right, key, val);
   else if (cmp == 0)
      x.val = val;
   return x;
}
```



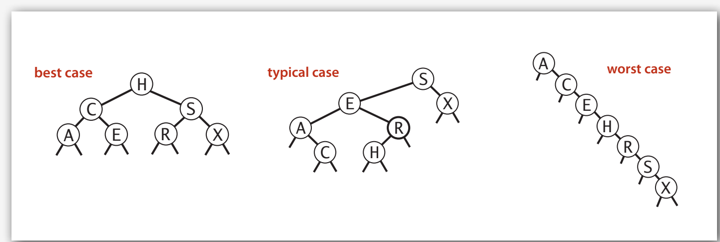
Running time. Proportional to depth of node.

BST trace: standard indexing client



Tree shape

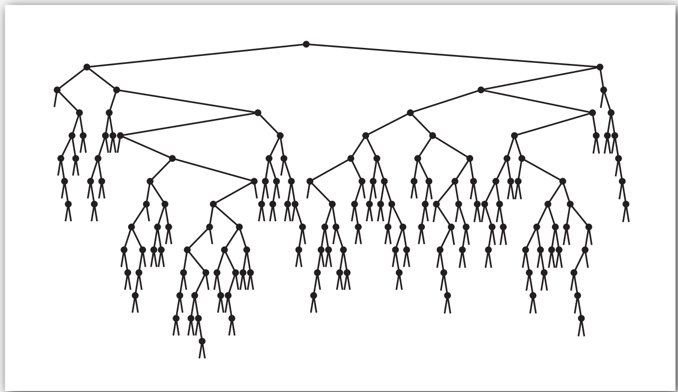
- Many BSTs correspond to same set of keys.
- Cost of search/insert is proportional to depth of node.



Remark. Tree shape depends on order of insertion.

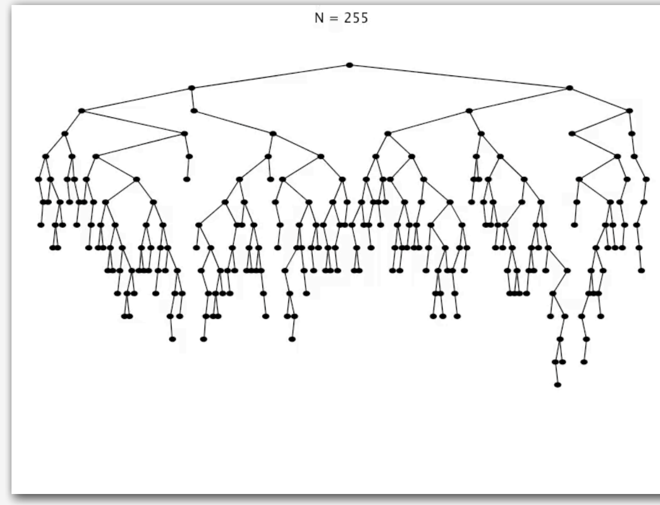
BST insertion: random order

Observation. If keys inserted in random order, tree stays relatively flat.

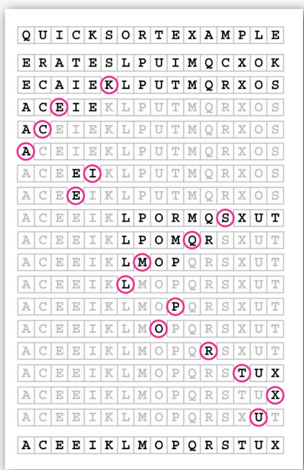


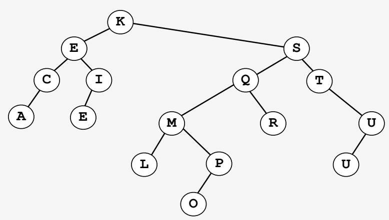
BST insertion: random order visualization

Ex. Insert keys in random order.



Correspondence between BSTs and quicksort partitioning





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

BSTs: mathematical analysis

Proposition. If keys are inserted in random order, the expected number of compares for a search/insert is ~ 2 ln N.

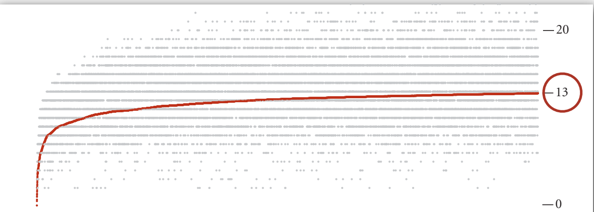
Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If keys are inserted in random order, expected height of tree is ~ 4.311 ln N.

But... Worst-case for search/insert/height is N.
(exponentially small chance when keys are inserted in random order)

ST implementations: summary

implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	`equals()`
binary search (ordered array)	lg N	N	lg N	N/2	yes	`compareTo()`
BST	N	N	1.39 lg N	1.39 lg N	?	`compareTo()`



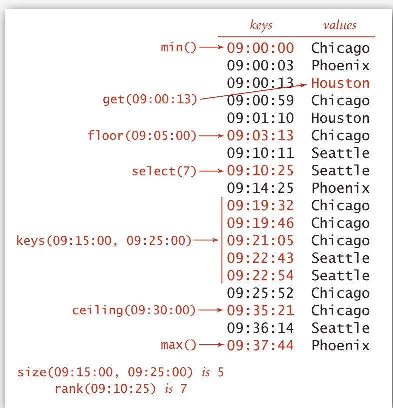
Costs for `java FrequencyCounter 8 < tale.txt` using BST

Next challenge. Ordered symbol tables ops in BSTs.

‣ basic implementations
‣ randomized BSTs
‣ **ordered symbol table ops**

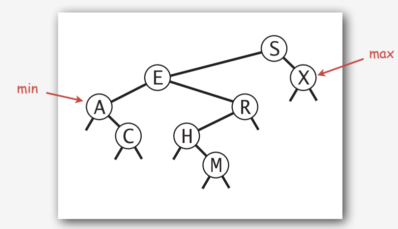
Ordered symbol table operations

Minimum. Smallest key in table.
Maximum. Largest key in table.
Floor. Largest key ≤ to a given key.
Ceiling. Smallest key ≥ to a given key.
Rank. Number of keys < than given key.
Select. Key of given rank.
Size. Number of keys in a given range.
Iterator. All keys in order.



Minimum and maximum

Minimum. Smallest key in table.
Maximum. Largest key in table.

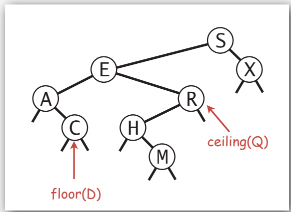


Q. How to find the min / max.
A.

Floor and ceiling

Floor. Largest key ≤ to a given key.

Ceiling. Smallest key ≥ to a given key.

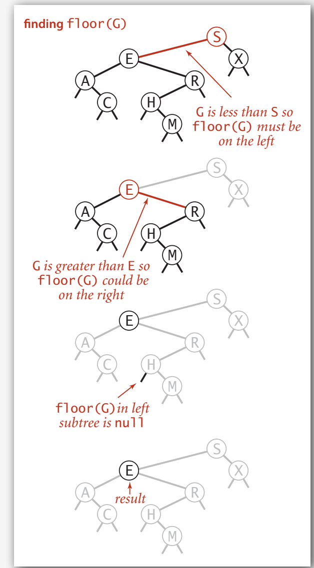


Q. How to find the floor /ceiling.

A.

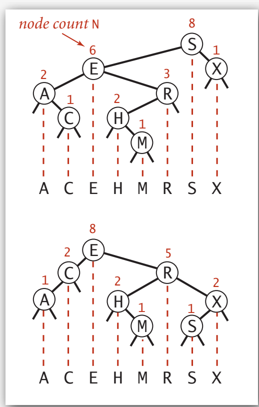
Computing the floor

```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   if (cmp == 0) return x;

   if (cmp < 0)  return floor(x.left, key);

   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;

}
```



Subtree counts and `size()`

In each node, we store the number of nodes in the subtree rooted at that node.
To implement `size()`, return the count at the root.



Remark. This facilitates efficient implementation of `rank()` and `select()`.

BST implementation: subtree counts and `size()`

```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
   private int N;
}
```

nodes in subtree

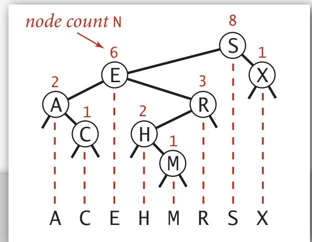
```
public int size()
{  return size(root);  }

private int size(Node x)
{
   if (x == null) return 0;
   return x.N;
}
```

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0)  x.val = val;
   x.N = 1 + size(x.left) + size(x.right);
   return x;
}
```

Rank

How many keys < k ?

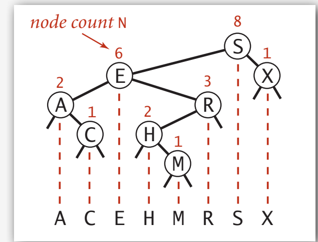
Easy recursive algorithm (4 cases!)




```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;

   int cmp = key.compareTo(x.key);
   if      (cmp < 0) return rank(key, x.left);

   else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);

   else              return size(x.left);
}
```


Range count

How many keys between `lo` and `hi`?



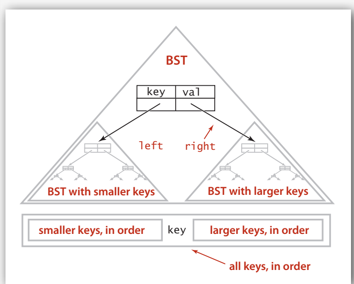

```
public int size(Key lo, Key hi)
{
   if (contains(hi)) return rank(hi) - rank(lo) - 1;
   else              return rank(hi) - rank(lo);
}
```

number of keys < hi

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
public Iterable<Key> allKeys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, queue);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```

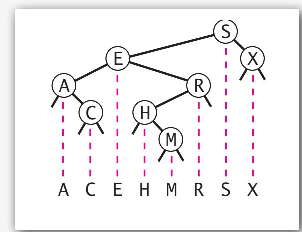


Property. Inorder traversal of a BST yields keys in ascending order.

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

recursive calls	queue	function call stack
visit(S)		S
visit(E)		S E
visit(A)		S E A
enqueue A	A	
visit(C)		S E A C
enqueue C	C	
enqueue E	E	
visit(R)		S E R
visit(H)		S E R H
enqueue H	H	
visit(M)		S E R H M
enqueue M	M	
print R	R	
enqueue S	S	
visit(X)		S X
enqueue X	X	



ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`

Next.
- Deletion in BSTs
- Can we guarantee logarithmic performance?

Searching challenge 3 (revisited):

Problem. Frequency counts in “Tale of Two Cities”
Assumptions. Book has 135,000+ words; about 10,000 distinct words.

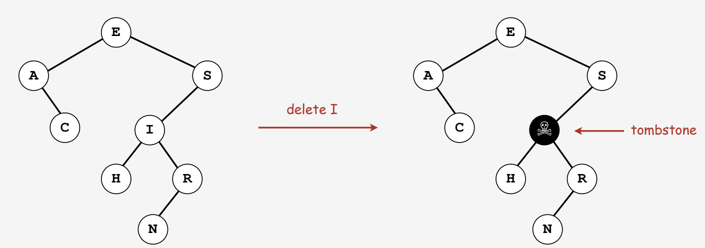
Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn’t matter much, all fast enough.
✓ 5) BSTs.

insertion cost < 10000 * 1.38 * lg 10000 < .2 million
lookup cost < 135000 * 1.38 * lg 10000 < 2.5 million

- basic implementations
- randomized BSTs
- **deletion in BSTs**

BST deletion: lazy approach

To remove a node with a given key:
- Set its value to `null`.
- Leave key in tree to guide searches (but don't consider it equal to search key).



Cost. O(log N') per insert, search, and delete (if keys in random order), where N' is the number of elements ever inserted in the BST.

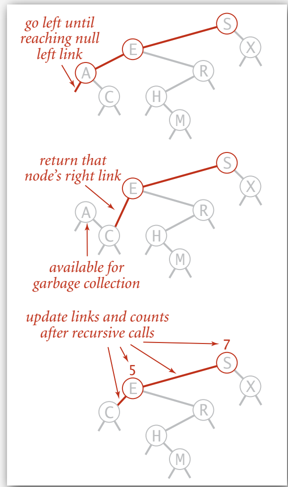
Unsatisfactory solution. Tombstone overload.

Deleting the minimum

To delete the minimum key:

- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

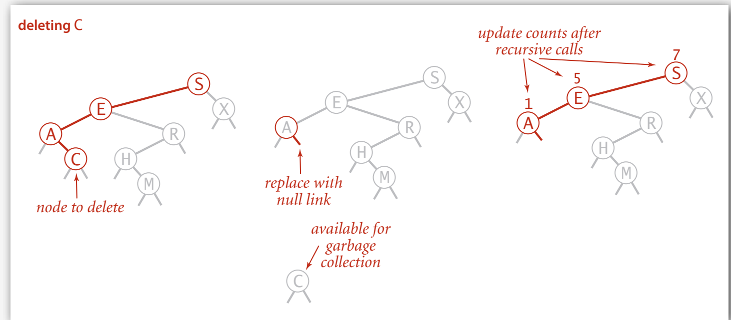
```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.N = 1 + size(x.left) + size(x.right);
   return x;
}
```



Hibbard deletion

To delete a node with key k: search for node t containing key k.

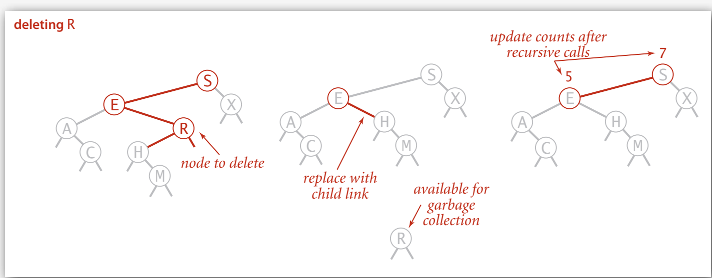
Case 0. [0 children] Delete t by setting parent link to null.



Hibbard deletion

To delete a node with key k: search for node t containing key k.

Case 1. [1 child] Delete t by replacing parent link.

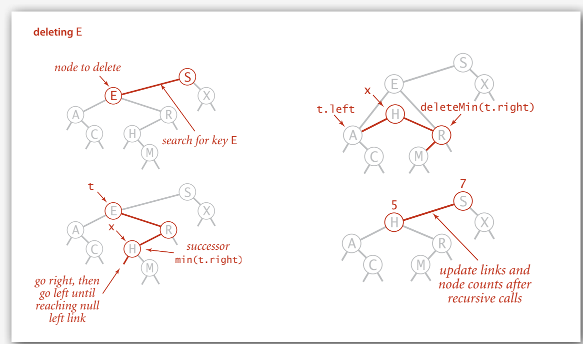


Hibbard deletion

To delete a node with key k: search for node t containing key k.

Case 2. [2 children]

- Find successor x of t. ← x has no left child
- Delete the minimum in t's right subtree. ← but don't garbage collect x
- Put x in t's spot. ← still a BST



Hibbard deletion: Java implementation

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

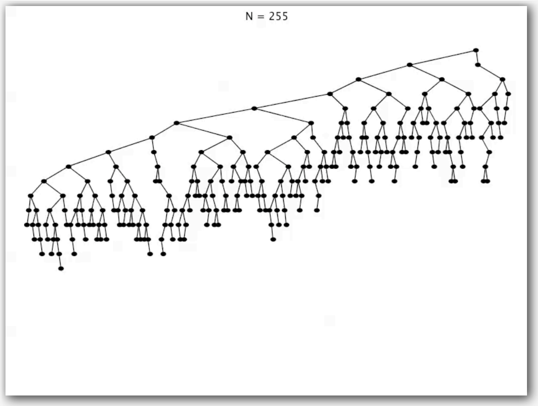
no right child

replace with successor

update subtree counts

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!) ⇒ sqrt(N) per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	√N	yes	`compareTo()`

other operations also become √N if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.