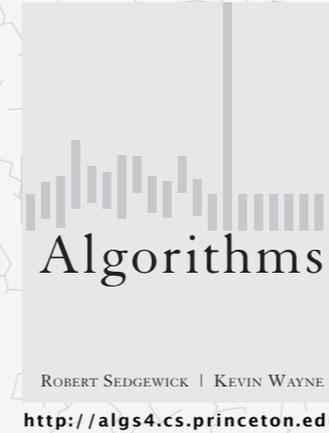


3.1 SYMBOL TABLES

- ▶ API
- ▶ elementary implementations
- ▶ ordered operations



3.1 SYMBOL TABLES

- ▶ API
- ▶ elementary implementations
- ▶ ordered operations

Symbol tables

Key-value pair abstraction.

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

Ex. DNS lookup.

- Insert domain name with specified IP address.
- Given domain name, find corresponding IP address.

domain name	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

↑
value

Symbol table applications

application	purpose of search	key	value
dictionary	find definition	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	type and value
routing table	route Internet packets	destination	best route
DNS	find IP address	domain name	IP address
reverse DNS	find domain name	IP address	domain name
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and $N-1$.

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

every array is an associative array every object is an associative array table is the only primitive data structure

```
hasNiceSyntaxForAssociativeArrays["Python"] = true
hasNiceSyntaxForAssociativeArrays["Java"]   = false
```

legal Python code

5

Basic symbol table API

Associative array abstraction. Associate one value with each key.

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

6

Conventions

- Values are not null. ← Java allows null value
- 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 void delete(Key key)
{ put(key, null); }
```

7

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; use hashCode() to scramble key.

built-in to Java
(stay tuned)

specify Comparable in API.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: Integer, Double, String, java.io.File, ...
- Mutable in Java: StringBuilder, java.net.URL, arrays, ...

8

Equality test

All Java classes inherit a method `equals()`.

Java requirements. For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is true.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is false.

} equivalence relation

Default implementation. `(x == y)`

Customized implementations. Integer, Double, String, java.io.File, ...

User-defined implementations. Some care needed.

do x and y refer to the same object?

9

Implementing equals for user-defined types

Seems easy.

```
public class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Date that)
    {
        if (this.day != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year ) return false;
        return true;
    }
}
```

check that all significant fields are the same

10

Implementing equals for user-defined types

Seems easy, but requires some care.

typically unsafe to use `equals()` with inheritance (would violate symmetry)

```
public final class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Object y)
    {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass())
            return false;

        Date that = (Date) y;
        if (this.day != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year ) return false;
        return true;
    }
}
```

must be Object. Why? Experts still debate.

optimize for true object equality

check for null

objects must be in the same class (religion: `getClass()` vs. `instanceof`)

cast is guaranteed to succeed

check that all significant fields are the same

11

Equals design

"Standard" recipe for user-defined types.

- Optimization for reference equality.
- Check against null.
- Check that two objects are of the same type and cast.
- Compare each significant field:
 - if field is a primitive type, use `==` ← but use `Double.compare()` with double (or otherwise deal with -0.0 and NaN)
 - if field is an object, use `equals()` ← apply rule recursively
 - if field is an array, apply to each entry ← can use `Arrays.deepEquals(a, b)` but not `a.equals(b)`

Best practices.

- No need to use calculated fields that depend on other fields. ← e.g., cached Manhattan distance
- Compare fields mostly likely to differ first.
- Make `compareTo()` consistent with `equals()`.

`x.equals(y)` if and only if `(x.compareTo(y) == 0)`

12

ST test client for traces

Build ST by associating value i with i^{th} string from standard input.

```
public static void main(String[] args)
{
    ST<String, Integer> st = new ST<String, Integer>();
    for (int i = 0; !StdIn.isEmpty(); i++)
    {
        String key = StdIn.readString();
        st.put(key, i);
    }
    for (String s : st.keys())
        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 11
M 9
P 10
R 3
S 0
X 7
```

13

ST test client for analysis

Frequency counter. Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```
% more tinyTale.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
it was the spring of hope
it was the winter of despair
```

```
% java FrequencyCounter 1 < tinyTale.txt
it 10
```

```
% java FrequencyCounter 8 < tale.txt
business 122
```

```
% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

← tiny example
(60 words, 20 distinct)

← real example
(135,635 words, 10,769 distinct)

← real example
(21,191,455 words, 534,580 distinct)

14

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>();
        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();
            if (word.length() < minlen) continue;
            if (!st.contains(word)) st.put(word, 1);
            else st.put(word, st.get(word) + 1);
        }
        String max = "";
        st.put(max, 0);
        for (String word : st.keys())
            if (st.get(word) > st.get(max))
                max = word;
        StdOut.println(max + " " + st.get(max));
    }
}
```

← create ST

← ignore short strings

← read string and
update frequency

← print a string
with max freq

15



3.1 SYMBOL TABLES

- ▶ API
- ▶ elementary implementations
- ▶ ordered operations

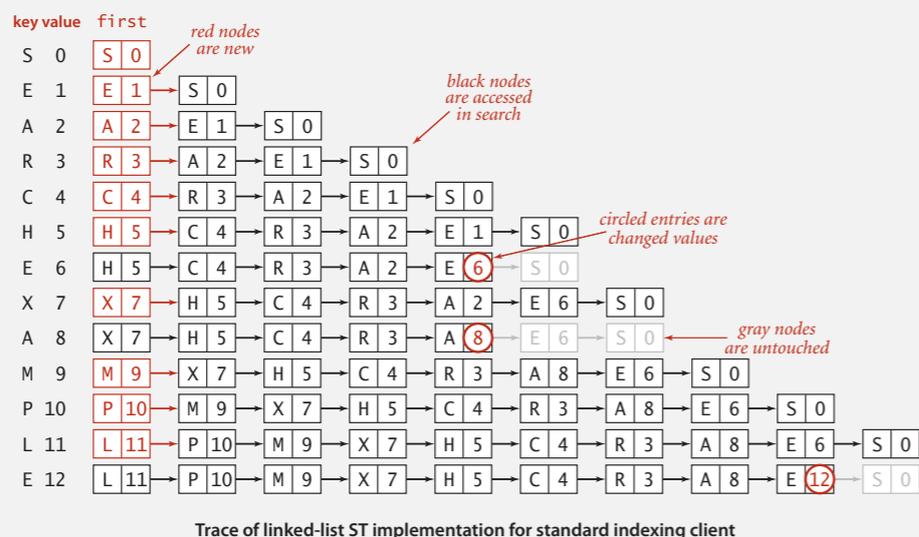
ROBERT SEDGEWICK | KEVIN WAYNE
<http://algs4.cs.princeton.edu>

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.



17

Elementary ST implementations: summary

ST implementation	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N/2$	N	<code>equals()</code>

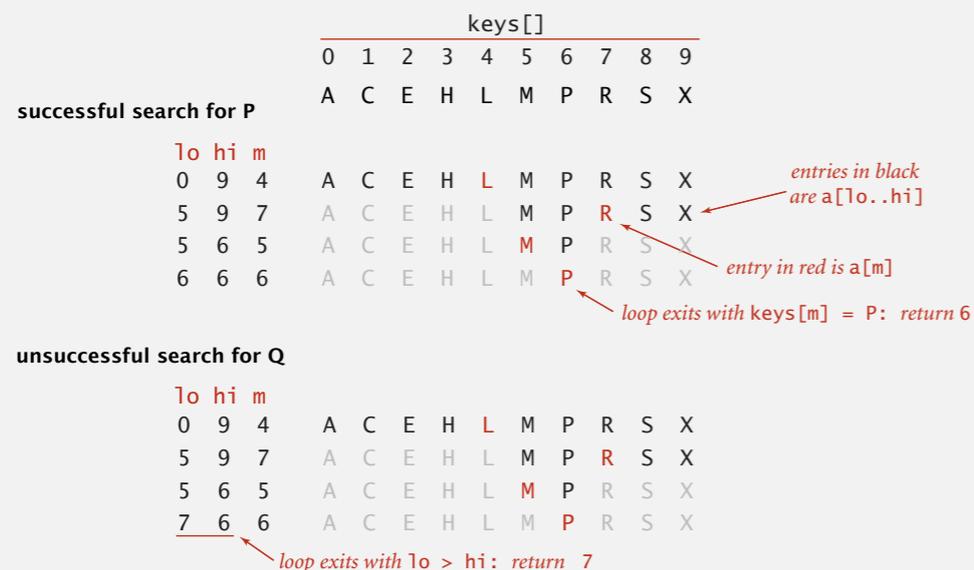
Challenge. Efficient implementations of both search and insert.

18

Binary search in an ordered array

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

Rank helper function. How many keys $< k$?



19

Binary search: Java implementation

```

public Value get(Key key)
{
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) return vals[i];
    else return null;
}

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

number of keys < key

20

Binary search: trace of standard indexing client

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

key	value	keys[]										N	vals[]										
		0	1	2	3	4	5	6	7	8	9		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	8	4	12	5	11	9	10	3	0	7	

Annotations:
 - entries in red were inserted
 - entries in black moved to the right
 - entries in gray did not move
 - circled entries are changed values

21

Elementary ST implementations: summary

ST implementation	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N/2$	N	equals()
binary search (ordered array)	$\log N$	N	$\log N$	$N/2$	compareTo()

Challenge. Efficient implementations of both search and insert.

22

3.1 SYMBOL TABLES

- ▶ API
- ▶ elementary implementations
- ▶ ordered operations



ROBERT SEDGWICK | KEVIN WAYNE
<http://algs4.cs.princeton.edu>

Examples of ordered symbol table API

	keys	values
min()	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	Houston
get(09:00:13)	09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00)	09:03:13	Chicago
	09:10:11	Seattle
select(7)	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00)	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00)	09:35:21	Chicago
	09:36:14	Seattle
max()	09:37:44	Phoenix

size(09:15:00, 09:25:00) is 5
 rank(09:10:25) is 7

24

Ordered symbol table API

```
public class ST<Key extends Comparable<Key>, Value>
    ...
    Key min()                smallest key
    Key max()                largest key
    Key floor(Key key)      largest key less than or equal to key
    Key ceiling(Key key)    smallest key greater than or equal to key
    int rank(Key key)       number of keys less than key
    Key select(int k)        key of rank k
    void deleteMin()        delete smallest key
    void deleteMax()        delete largest key
    int size(Key lo, Key hi) number of keys between lo and hi
    Iterable<Key> keys()     all keys, in sorted order
    Iterable<Key> keys(Key lo, Key hi) keys between lo and hi, in sorted order
```

25

Binary search: ordered symbol table operations summary

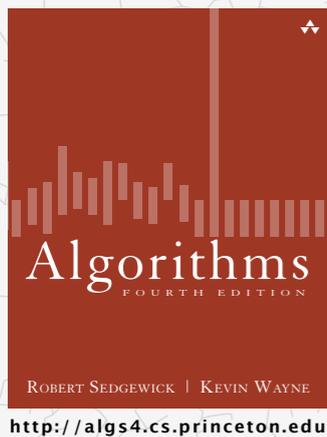
	sequential search	binary search
search	N	$\log N$
insert / delete	N	N
min / max	N	1
floor / ceiling	N	$\log N$
rank	N	$\log N$
select	N	1
ordered iteration	$N \log N$	N

order of growth of the running time for ordered symbol table operations

26

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



3.2 BINARY SEARCH TREES

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

<http://algs4.cs.princeton.edu>



3.2 BINARY SEARCH TREES

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

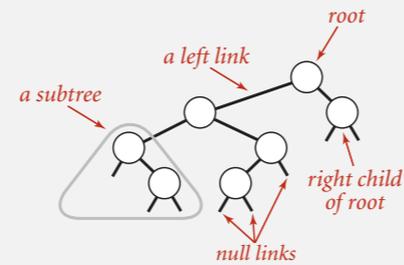
<http://algs4.cs.princeton.edu>

Binary search trees

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

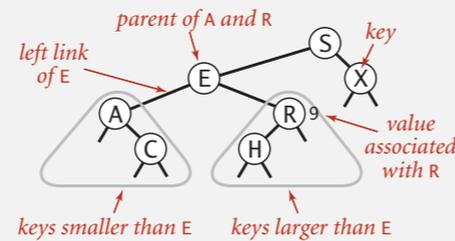
A binary tree is either:

- 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.

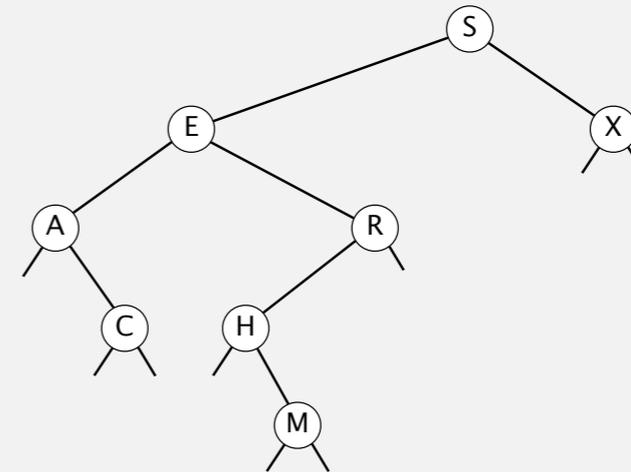


3

Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

successful search for H

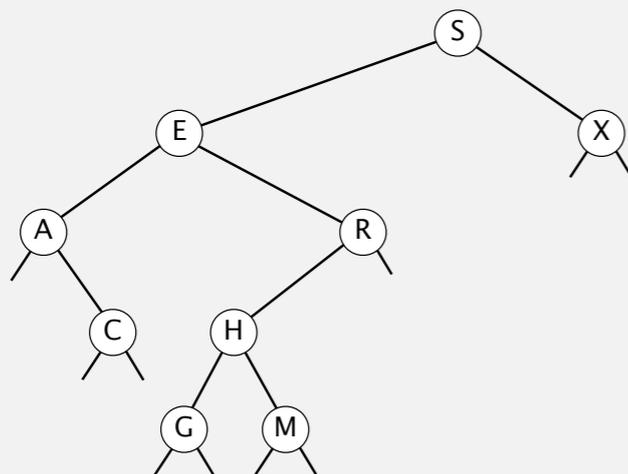


4

Binary search tree demo

Insert. If less, go left; if greater, go right; if null, insert.

insert G



5

BST representation in Java

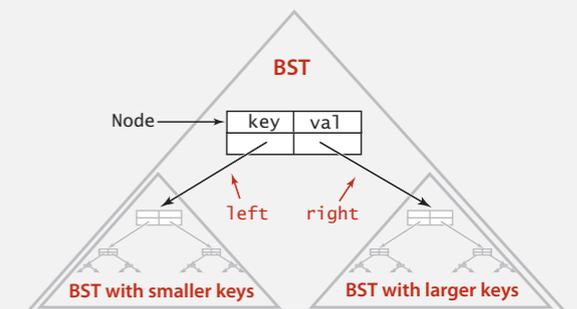
Java definition. A BST is a reference to a root Node.

A Node is composed 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;
    }
}
```



Binary search tree

Key and Value are generic types; Key is Comparable

6

BST implementation (skeleton)

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

    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 */ }
}
```

7

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;
}
```

Cost. Number of compares is equal to 1 + depth of node.

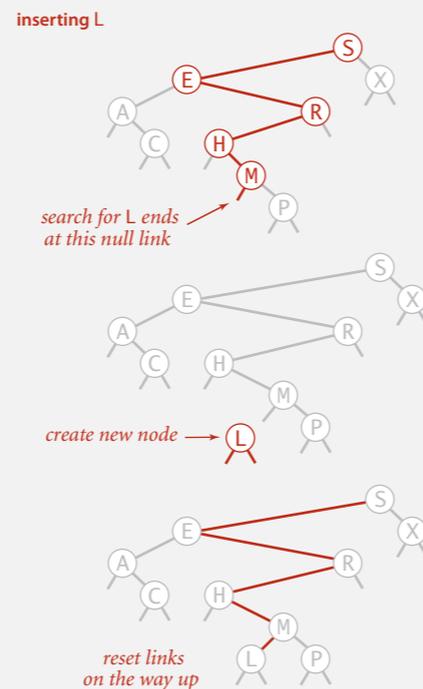
8

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.



Insertion into a BST

9

BST insert: Java implementation

Put. Associate value with key.

```
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;
}
```

concise, but tricky,
recursive code;
read carefully!

Cost. Number of compares is equal to 1 + depth of node.

10

BSTs: mathematical analysis

Proposition. If N distinct keys are inserted into a BST in **random** order, the expected number of compares for a search/insert is $\sim 2 \ln N$.

Pf. 1-1 correspondence with quicksort partitioning.

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

How Tall is a Tree?

Bruce Reed
CNRS, Paris, France
reed@moka.ccr.jussieu.fr

ABSTRACT
Let H_n be the height of a random binary search tree on n nodes. We show that there exists constants $\alpha = 4.31107\dots$ and $\beta = 1.95\dots$ such that $\mathbb{E}(H_n) = \alpha \log n - \beta \log \log n + O(1)$. We also show that $\text{Var}(H_n) = O(1)$.

But... Worst-case height is N .

[exponentially small chance when keys are inserted in random order]

15

ST implementations: summary

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$\frac{1}{2} N$	N	equals()
binary search (ordered array)	$\lg N$	N	$\lg N$	$\frac{1}{2} N$	compareTo()
BST	N	N	$1.39 \lg N$	$1.39 \lg N$	compareTo()

Why not shuffle to ensure a (probabilistic) guarantee of $4.311 \ln N$?

16

3.2 BINARY SEARCH TREES

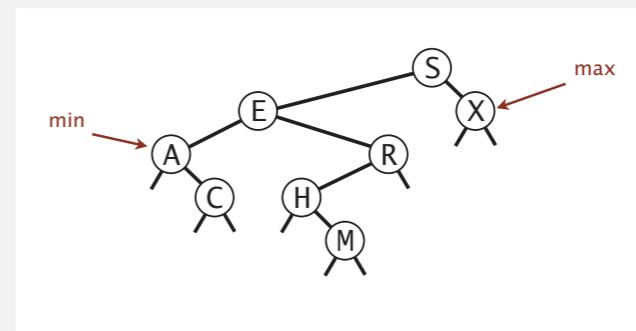
- ▶ BSTs
- ▶ ordered operations
- ▶ deletion



Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.



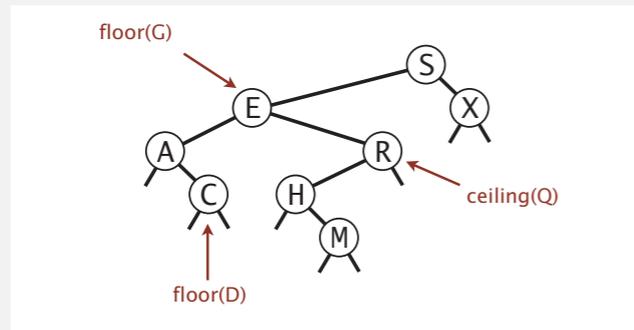
Q. How to find the min / max?

18

Floor and ceiling

Floor. Largest key \leq a given key.

Ceiling. Smallest key \geq a given key.



Q. How to find the floor / ceiling?

19

Computing the floor

Case 1. [k equals the key in the node]

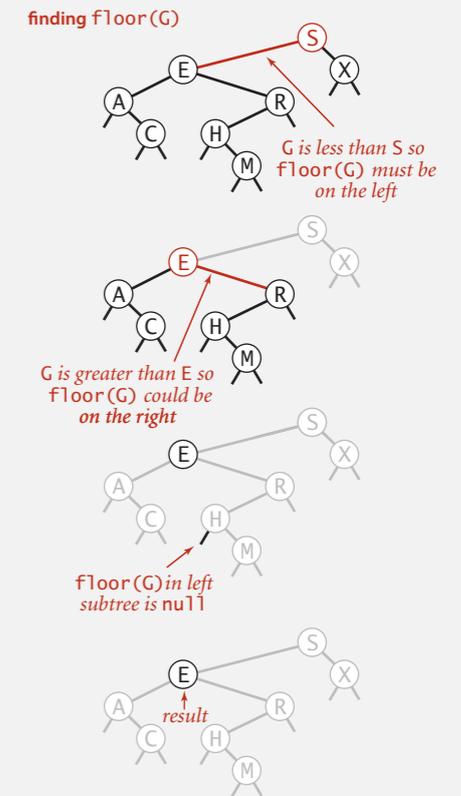
The floor of k is k .

Case 2. [k is less than the key in the node]

The floor of k is in the left subtree.

Case 3. [k is greater than the key in the node]

The floor of k is in the right subtree
(if there is any key $\leq k$ in right subtree);
otherwise it is the key in the node.



20

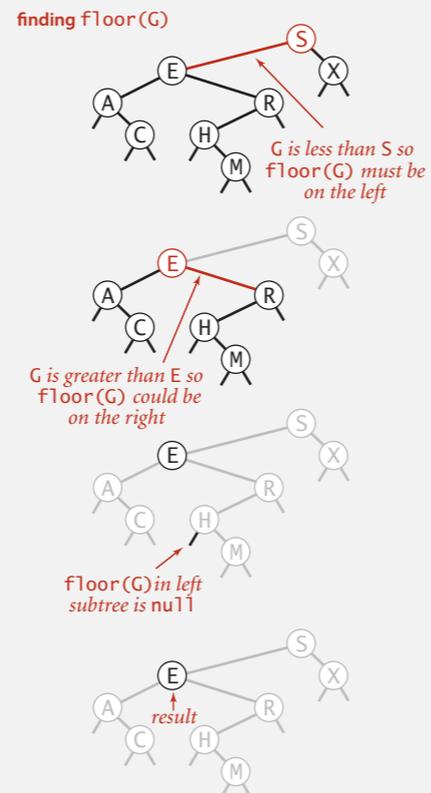
Computing the floor

```
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;
}
```

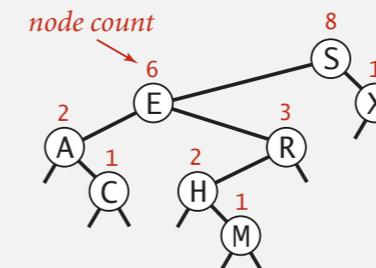


21

Rank and select

Q. How to implement rank() and select() efficiently?

A. 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.



22

BST implementation: subtree counts

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

number of nodes in subtree

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

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

ok to call when x is null

```
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val, 1);
    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.count = 1 + size(x.left) + size(x.right);
    return x;
}
```

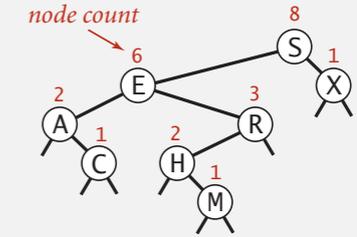
initialize subtree count to 1

23

Rank

Rank. How many keys $< k$?

Easy recursive algorithm (3 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 if (cmp == 0) return size(x.left);
}
```

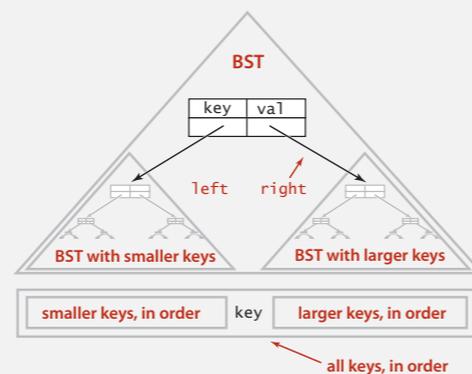
24

Inorder traversal

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

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    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.

25

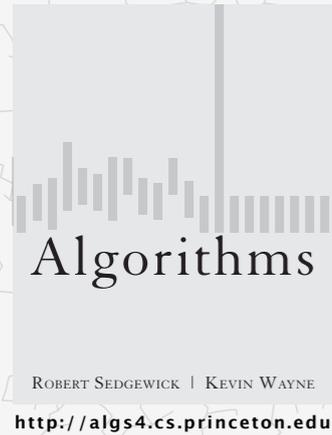
BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	$\lg N$	h
insert	N	N	h
min / max	N	1	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N

h = height of BST
(proportional to $\log N$
if keys inserted in random order)

order of growth of running time of ordered symbol table operations

26



3.2 BINARY SEARCH TREES

- ▶ BSTs
- ▶ ordered operations
- ▶ deletion

ST implementations: summary

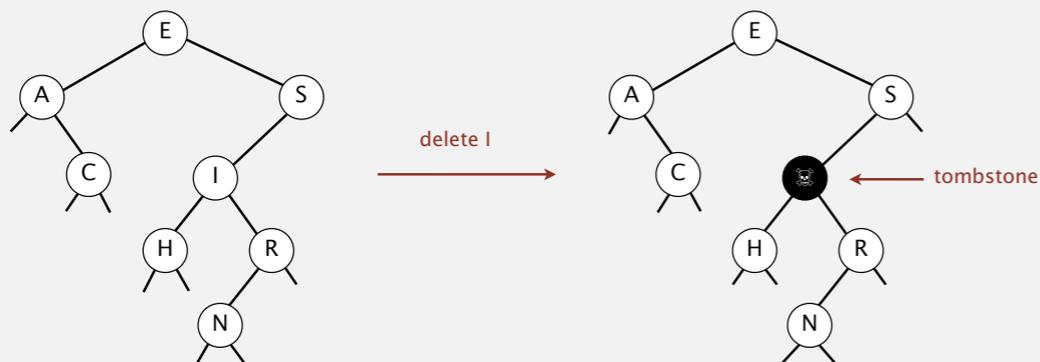
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		equals()
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	compareTo()
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$???	✓	compareTo()

Next. 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 search (but don't consider it equal in search).



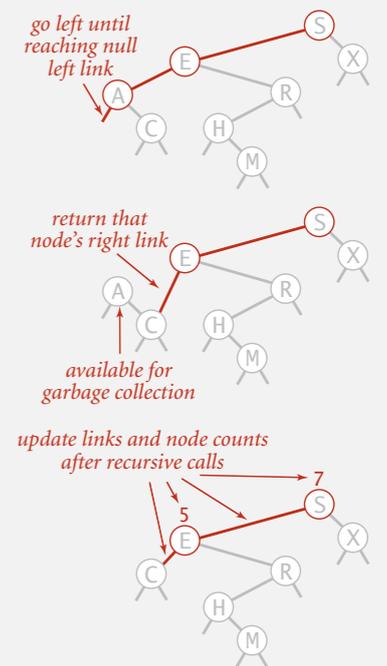
Cost. $\sim 2 \ln N'$ per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone (memory) 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.



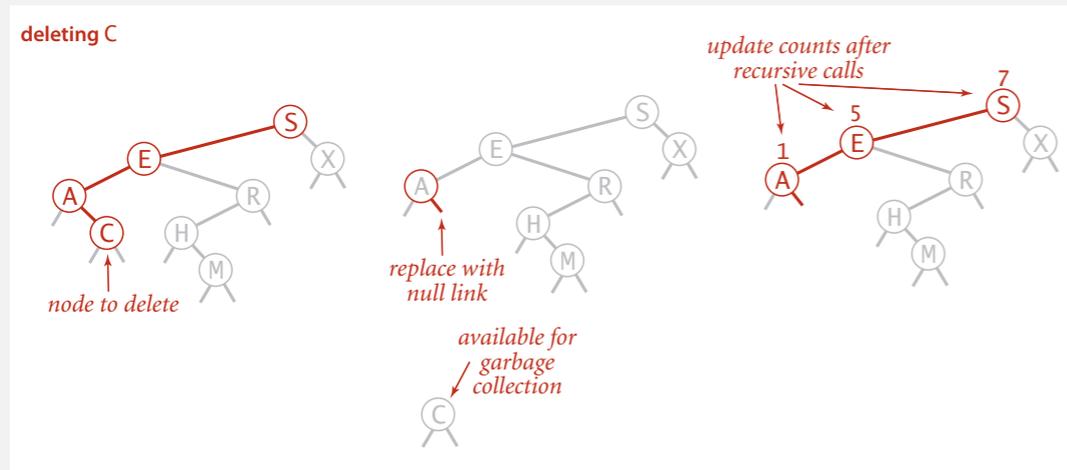
```
public void deleteMin()
{ root = deleteMin(root); }

private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.count = 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.

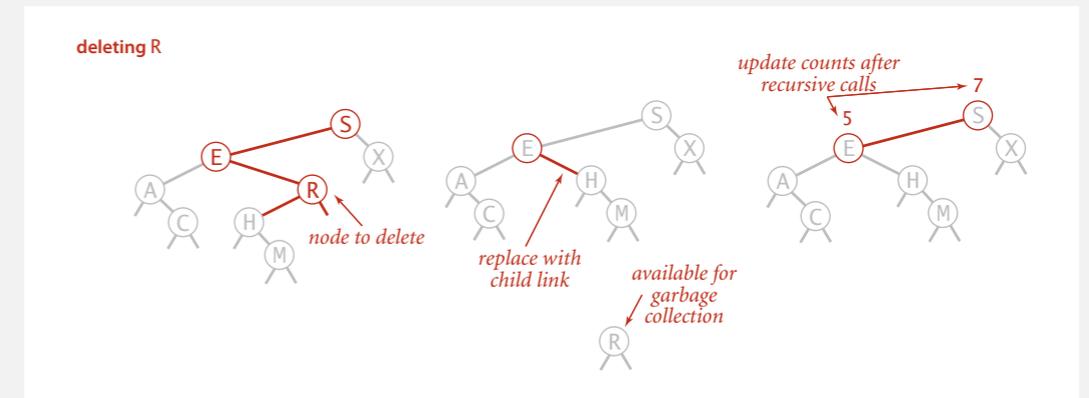


31

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.



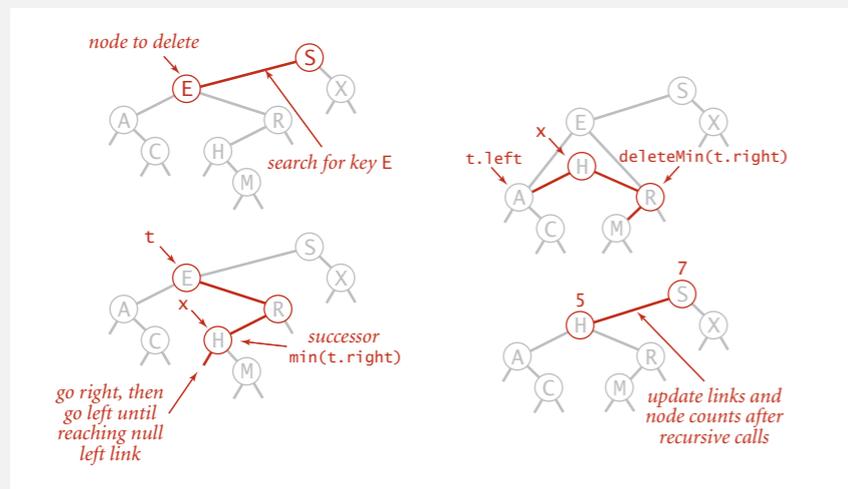
32

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



33

Hibbard deletion: Java implementation

```

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

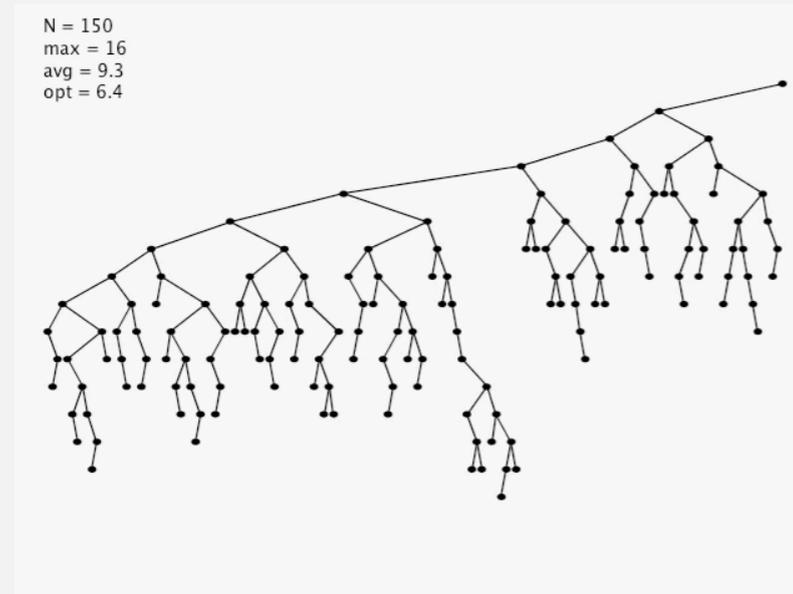
private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left;
        if (x.left == null) return x.right;

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

34

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!) $\Rightarrow \sqrt{N}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

35

ST implementations: summary

implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>

other operations also become \sqrt{N} if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.

36