Searching: Symbol Tables & Binary Search Trees

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 3.1, 3.2), Prof. Janicki's course slides

Smybol Table

A **symbol table** is a data structure for key-value pairs that supports two operations:

- Insert (put) a new pair into the table.
- Search for (get) the value associated with a given key.

Also known as: maps, dictionaries, associative arrays.

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

Language support: Numerous languages support symbols tables either as external libraries, built-in libraries or built-into the language.

Smybol Table Example

DNS Look-up

- 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

Smybol 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
account management	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find type and value	variable name	type and value

Typical symbol-table applications

Symbol Table API

Associative array/Symbol Table 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 table (remove* `key` *from table if value is* `null`*)*
`Value`	`get(Key key)`	*value paired with* `key` *(*`null` *if* `key` *is absent)*
`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 table*

API for a generic basic symbol table

Symbol Table Conventions

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

Intended consequences of Value $\neq$ `null`

- 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);  }
```

Ordered symbol table API

public class ST<Key extends Comparable<Key>, Value>		
	`ST()`	*create an ordered symbol table*
`void`	`put(Key key, Value val)`	*put key-value pair into the table (remove* `key` *from table if value is* `null`*)*
`Value`	`get(Key key)`	*value paired with* `key` *(*`null` *if* `key` *is absent)*
`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*
`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 in* `[lo..hi]`
`Iterable<Key>`	`keys(Key lo, Key hi)`	*keys in* `[lo..hi]`*, in sorted order*
`Iterable<Key>`	`keys()`	*all keys in the table, in sorted order*

API for a generic ordered symbol table

Elementary ST implementations

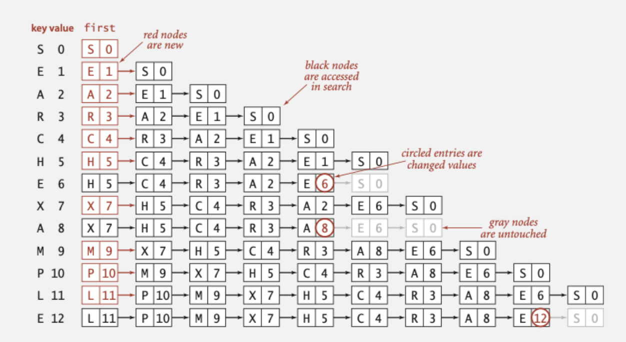
- Unordered array
- Ordered array
- Unordered linked list
- Ordered linked list

ST implementations - Unordered Linked List

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

Search. Scan through all keys until find a match (sequential search).

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

ST implementations - Ordered Array

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

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

Can we do better than sequential search in a sorted array?

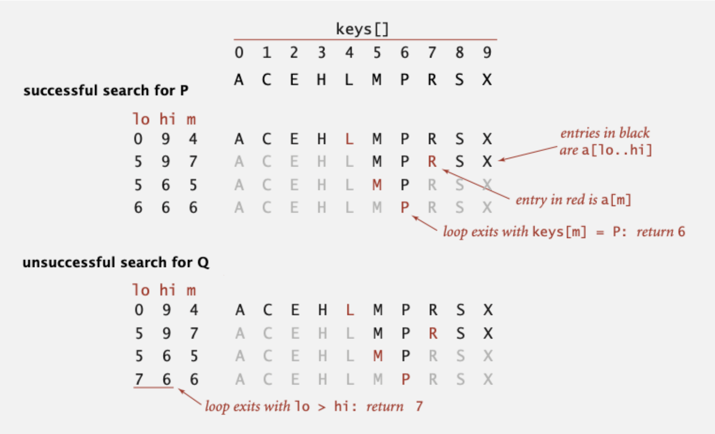
Solution: Binary search! (See Demo under chapter 1: https://algs4.cs.princeton.edu/lectures/)

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

Binary search uses at most $1 + \log_2 N$ key compares to search in a sorted array of size N, i.e. it has time complexity $T(N) = O(\log_2 N)$.

ST implementations - Ordered Array



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)                    number of keys < 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;
}
```

Binary search

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

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

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

Pros and Cons of Symbol Table Implementation

underlying data structure	implementation	pros	cons
linked list (sequential search)	`SequentialSearchST`	best for tiny STs	slow for large STs
ordered array (binary search)	`BinarySearchST`	optimal search and space, order-based ops	slow insert
binary search tree	`BST`	easy to implement, order-based ops	no guarantees space for links
balanced BST	`RedBlackBST`	optimal search and insert, order-based ops	space for links
hash table	`SeparateChainingHashST` `LinearProbingHashST`	fast search/insert for common types of data	need hash for each type no order-based ops space for links/empty

Pros and cons of symbol-table implementations

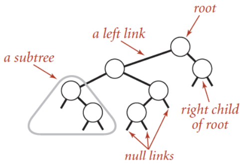
Binary Search Trees

Binary Trees Structure

We now examine a symbol-table implementation that combines the flexibility of insertion in a linked list with the efficiency of search in an ordered array!

Binary tree structure:

- Every node is pointed to by just one other node, which is called its parent (except the root),
- Each node has exactly two links, which are called its left and right links, that point to nodes called its left child (left subtree) and right child (right subtree), respectively.



Anatomy of a binary tree

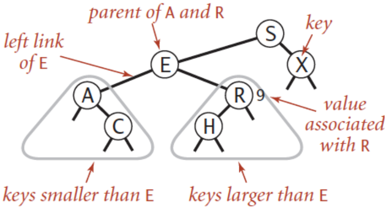
A binary tree is either:

- Empty, or
- a node with a left link and a right link, each referencing to (disjoint) subtrees that are themselves binary trees.

Binary Search Trees

A **binary search tree (BST)** is a binary tree where 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.

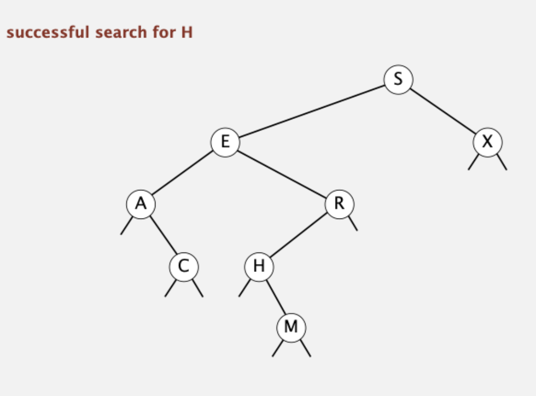


Anatomy of a binary search tree

Can you have duplicates in BST?

Binary Search Tree – Find/Search

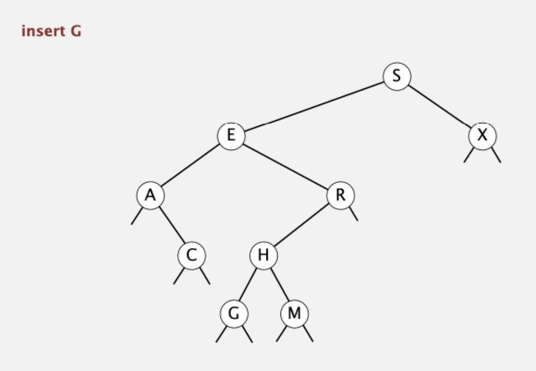
Find/Search: If less, go left; if greater, go right; if equal, found; if null, not found.



See Demo: `https://algs4.cs.princeton.edu/lectures/`

Binary Search Tree – Insert

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



Is a new node alway inserted as leaf node?

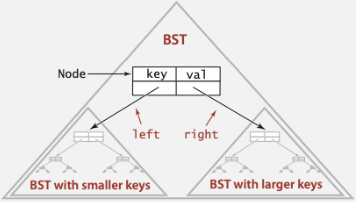
BST implementation: Node

A Node is composed of four fields:

- A Key and a Value.
- A reference to the left (smaller) and right (larger) subtree.
- An instance variable N that gives the node count in the subtree rooted at the node.

```
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)

Below is the implementation of the ordered symbol-table API using a binary search tree built from Node objects.

```
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 implementation: Get()

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

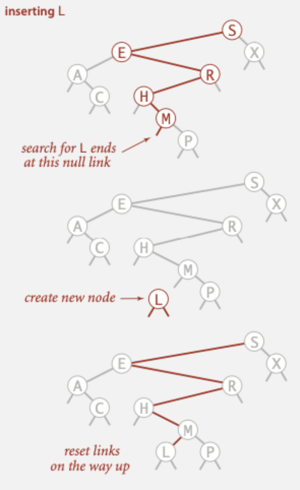
```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

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

BST Insert()/Put()

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

BST Implementation: Insert()/Put()

Put: Associate value with key.

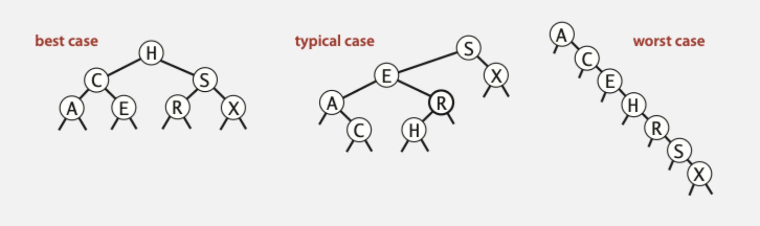
concise, but tricky, recursive code; read carefully!

```
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;
}
```

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

BST Tree Shape

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



Bottom Line. Tree shape depends on the order of insertion.

BST Tree Randomization

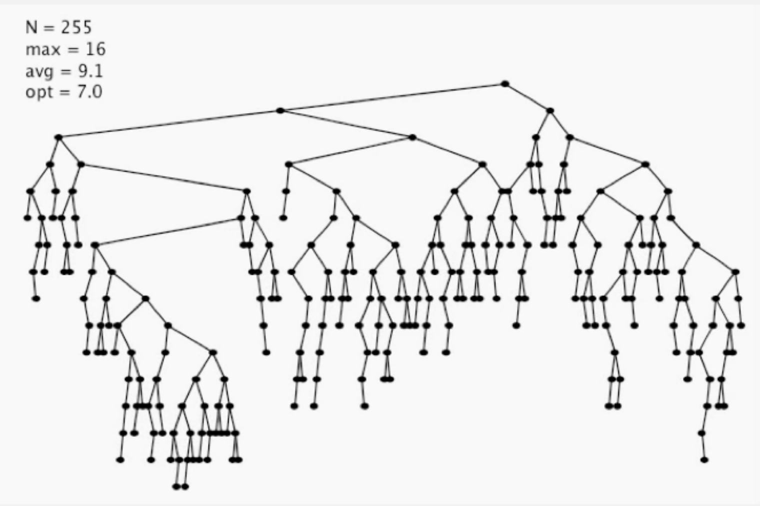
Assume that the keys are (uniformly) random or that they are inserted in random order.

In a BST built from N random keys:

- Search hits/misses and insertions require $\sim 2 \log_2 N$ (about $1.39 \log_2 N$) compares, on the average.

BST insertion: random order visualization

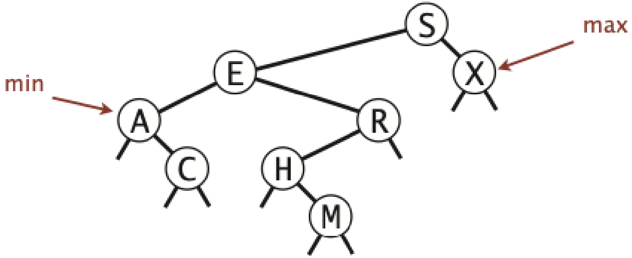
Ex. Insert keys in random order.



BST: Min. and Max. Operations

Minimum - returns the smallest key in table. Go to the left as far as possible.

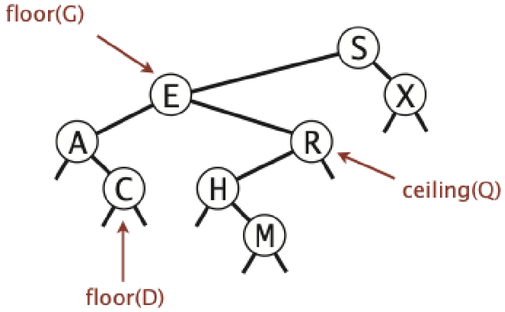
Maximum - returns the largest key in table. Go to the right as far as possible.



BST: Floor and Ceiling Operations

Floor - the largest key in the BST less than or equal to key.

Ceiling - the smallest key in the BST greater than or equal to key.

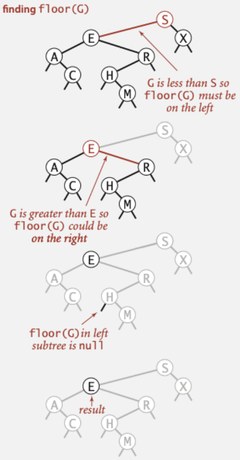


Computing the floor (and ceiling)

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.

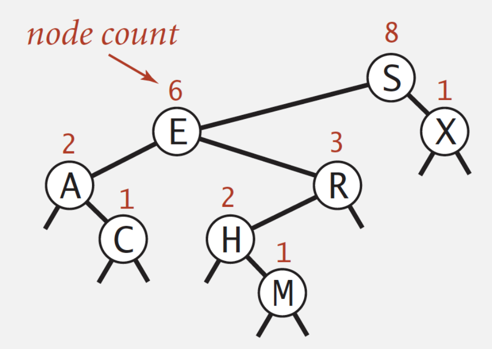
Case 3. [k is greater than the key in the node]
The floor of k is in the right subtree
(if there is any key $\leq k$ in right subtree);
otherwise it is the key in the node.



Interchanging right and left (and less and greater) gives ceiling()

BST: subtree

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.



BST: subtree contd...

```
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

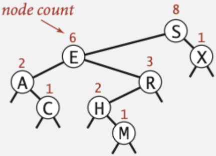
```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   x.count = 1 + size(x.left)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

initialize subtree count to 1

BST: Rank

Rank. How many keys < k?

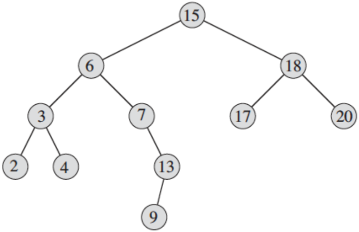
Easy recursive algorithm (3 cases!)




```
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);
}
```

Inorder Tree Walk/Traversal

An **Inorder** tree walk prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree.

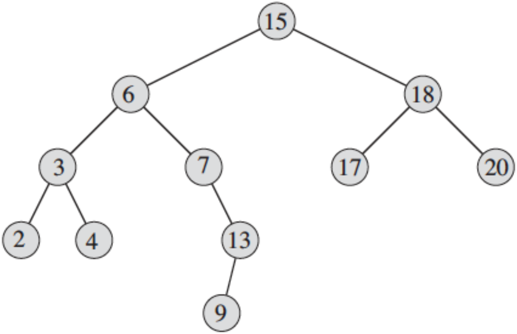


Sequence of nodes printed: `<2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20>`

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

Preorder Tree Traversal

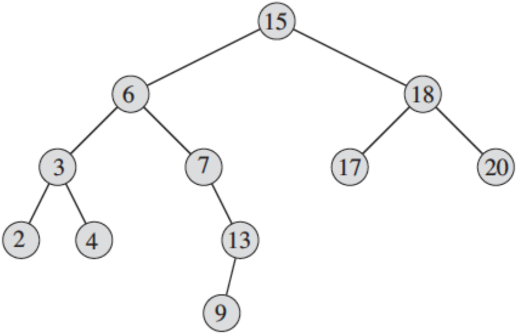
A **preorder** tree walk prints the root before the values in either subtree,



Sequence of nodes printed: `<15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20>`

Postorder Tree Traversal

A **postorder** tree walk prints the root after the values in its subtrees

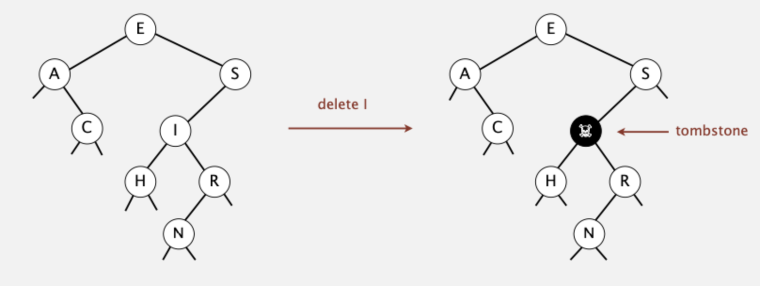


Sequence of nodes printed: `<2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15>`

BST: Delete, 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).



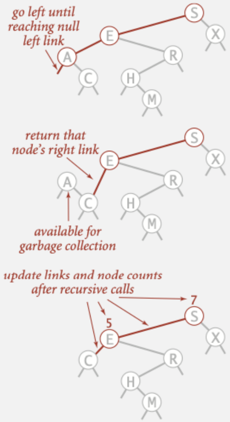
Cost. $\sim 2\log_2 N'$ per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.
Unsatisfactory solution. Tombstone (memory) overload.

BST: Delete 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.

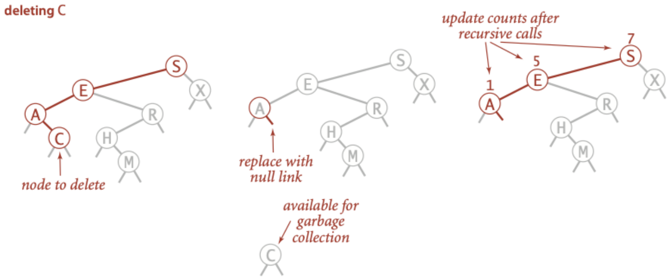
```
public void deleteMin()
{  root = deleteMin(root);  }

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;
}
```



BST: Hibbard Deletion

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

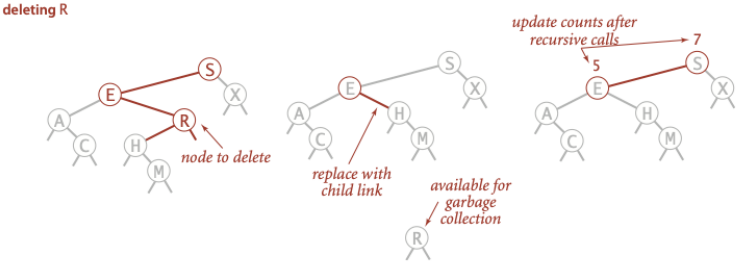
Case 0. [0 children] Delete t by setting its parent link to null.



BST: Hibbard Deletion

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

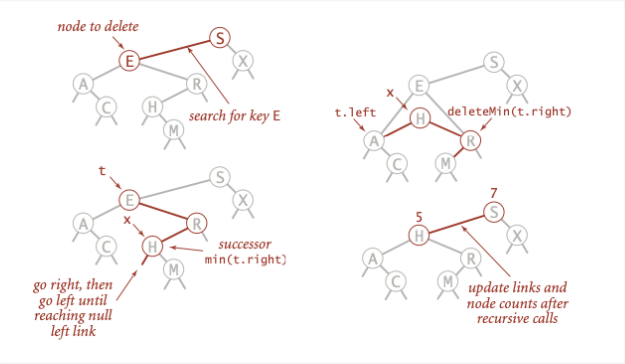
Case 1. [1 child] Delete t and connect its single child to t's parent.



BST: Hibbard Deletion: Case 2. [2 children]

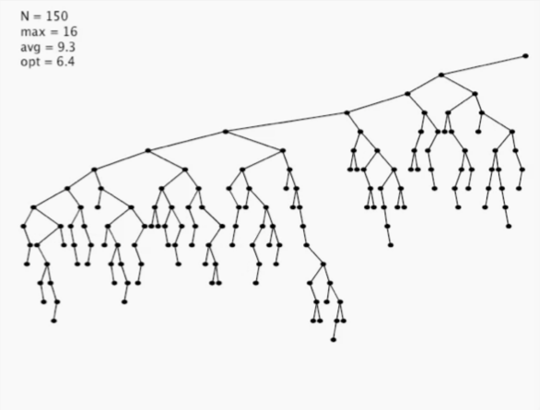
To delete a node with key k: search for node t containing key k.

- t is replaced by x = the minimum key in t's right subtree.
- x's right child is x's replacement.
- x. left = t.left, x.right=t.right (if x = t.right, then x.right =null).



BST: 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.

BST Cost

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
sequential search (unordered linked list)	N	N	$N/2$	N	no
binary search (ordered array)	$\lg N$	N	$\lg N$	$N/2$	yes
binary tree search (BST)	N	N	$1.39 \lg N$	$1.39 \lg N$	yes

Cost summary for basic symbol-table implementations (updated)