Symbol Tables & Binary Search Trees

Nicholas Moore

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)

Symbol Table

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

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

Also known as: maps, dictionaries, associative arrays.

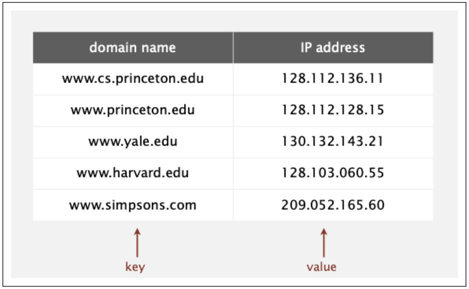
Symbol tables are generalizes arrays – Keys need not be between 0 and $N - 1$.

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

Tabula Rasa

Examples:

- DNS Lookup
  - key $\mapsto$ domain name
  - value $\mapsto$ IP address
- Dictionary
  - key $\mapsto$ word
  - value $\mapsto$ definition
- Compiler
  - key $\mapsto$ variable name
  - value $\mapsto$ type

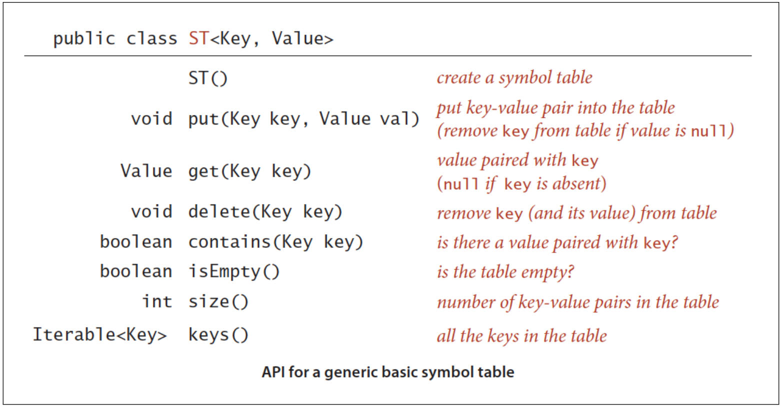


domain name	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

Many, many more examples exist!

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

Symbol table conventions adopted in the text book:

- Neither Keys nor Values are permitted to be `null`.
- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.

Intended consequences of Value $\neq$ `null`

- It makes it easy to implement `contains()`.

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

- It allows a lazy version of `delete()`.

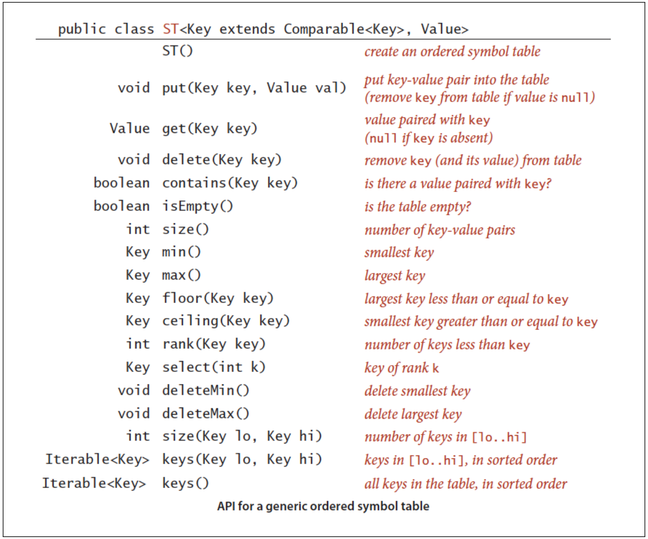
```
public void delete(Key key)
{  put(key, null);  }
```

Ordered vs Unordered Symbol Tables

Symbol tables can be more or less generic, and as we know well, we can often improve algorithms by adding properties to data structures.

- In its most basic version, we only need a test of equality between keys.
    - Item lookup wold thus become a linear search.
    - In prinicple this is how Python does it, since keys of different data types may be combined in one dictionary.
- If inequality operators are defined for the key data type, we can construct an **ordered symbol table**.
- We can introduce many useful operations, and improve runtimes of existing ones.
- The price we pay is key monotyping.

Ordered symbol table API



public class ST<Key extends Comparable<Key>, Value>

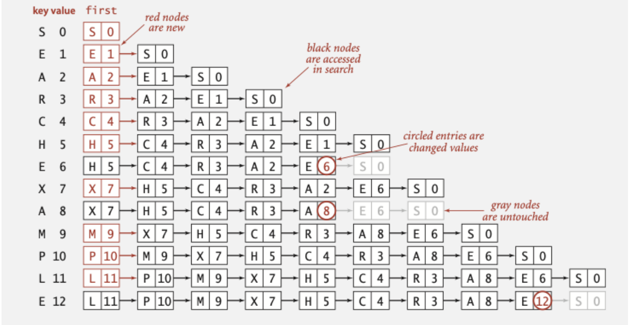
	`ST()`	*create an ordered symbol table*
`void`	`put(Key key, Value val)`	*put key-value pair into the table (remove* `key` *from table if value is* `null`*)*
`Value`	`get(Key key)`	*value paired with* `key` *(*`null` *if* `key` *is absent)*
`void`	`delete(Key key)`	*remove* `key` *(and its value) from 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

ST implementation - Unordered Linked List

Search: All nodes must be searched sequentially.

Insert: Search for the key. If present, overwrite the value, otherwise prepend a new pair to the list.


key value
first
red nodes are new
black nodes are accessed in search
circled entries are changed values
gray nodes are untouched

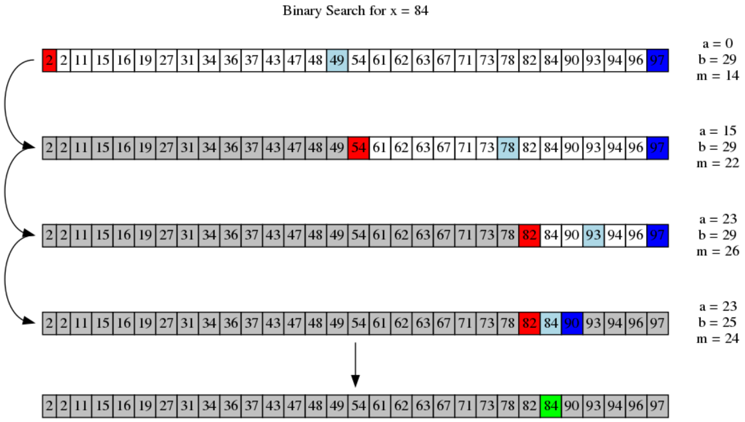
Trace of linked-list ST implementation for standard indexing client

Binary Search!

Maintaining a sorted table enables the powerful **Binary Search** algorithm. Let's say we are looking for x in A:

1. Create two variables to keep track of the search range:
    - a to track the lower bound
    - b to track the upper bound
2. Examine the number at index $m = \lfloor \frac{a+b}{2} \rfloor$
    1. If the numbers at m, a, or b equal to x, we have found x!
    2. If $m < x$, we know that the index of x must be at a greater than than $\lfloor \frac{a+b}{2} \rfloor$.
        - Set a to $\lfloor \frac{a+b}{2} \rfloor + 1$ and return to step 2.
    3. If $m > x$, we know that x must be at a lower index than $\lfloor \frac{a+b}{2} \rfloor$.
        - Set b to $\lfloor \frac{a+b}{2} \rfloor - 1$ and return to step 2.

Visualizing Binary Search



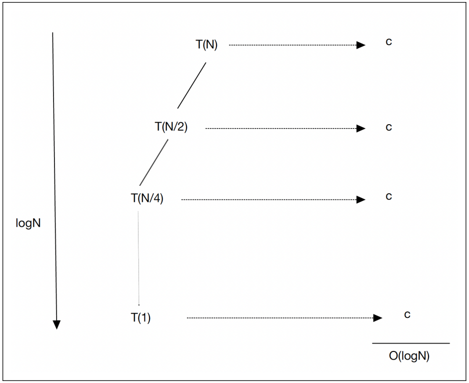
Binary Search - Java Implementation

```
int a = 0;
int b = A.length - 1;
while (A <= b) { // Key is in a[a..b] or not present.
       int m = a + (b - a) / 2;
       if (x < A[m]) b = m - 1;
       else if (x > A[m]) a = m + 1;
       else return m;
}
return -1;
```

Binary Search Complexity

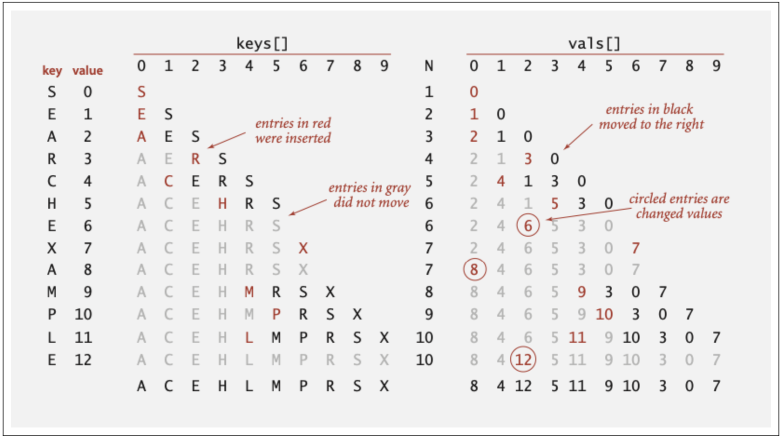
Binary search is a **bisection method**, as the size of the searchable section is divided by 2 at each step. Our recurrance equation is therefore $T(N) = T(N/2) + c$, with $c > 0$ and $T(1) = 1$

- For simplicity assume N is a power of 2.
- From the recursion tree ($\Rightarrow$), we have $T(N) = 1 + c\log_2 N$
- Therefore $T(N) \in O(\log N)$
- Binary search uses at most $1 + \log_2 N$ key comparisons to search a sorted array.



Ordered ST: Insert

Unfortunately, insertion into the middle of an array requires that all items greater than the item inserted be shifted one place to the right.



		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

Symbol table (ordered and unordered) operations summary

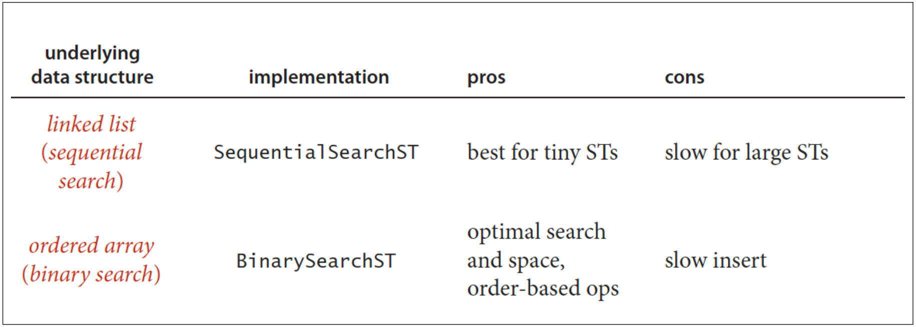
Sequential Search (unordered linked list):

- Search: $O(N)$
- Insert: $O(N)$

Binary Search (ordered array):

- Binary Search: $O(\lg N)$
- Insert: $O(N)$

In Summary...



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 Trees

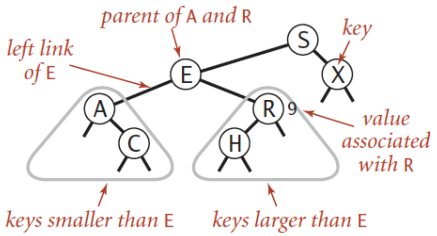
A **Binary Search Tree (BST)** is a binary tree where each node has a key.

Each key is:

- larger than all keys in its left subtree
- smaller than all keys in its right subtree

In contrast to binary heaps, *BSTs need not be complete binary trees.*

- Later on we'll have to address the problems this introduces.



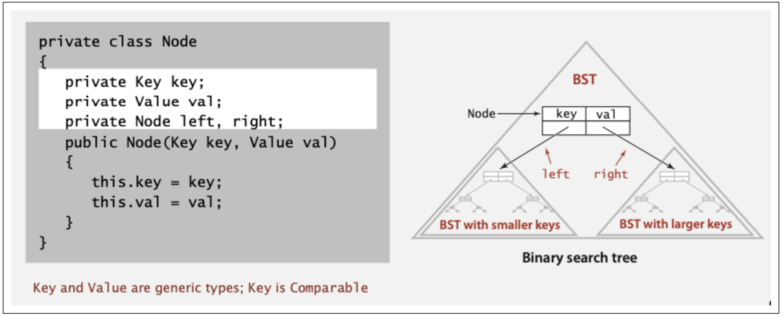
Anatomy of a binary search tree

If we're using this to implement a symbol table, can you have duplicate keys in the BST? What about in general?

BST implementation: Node

A BST Node is composed of four fields:

- A Key and a Value.
- A reference to the left (smaller) and right (larger) subtree.
- (For later) An instance variable N that gives the node count in the subtree rooted at the node. This field facilitates the implementation of various ordered symbol-table operations

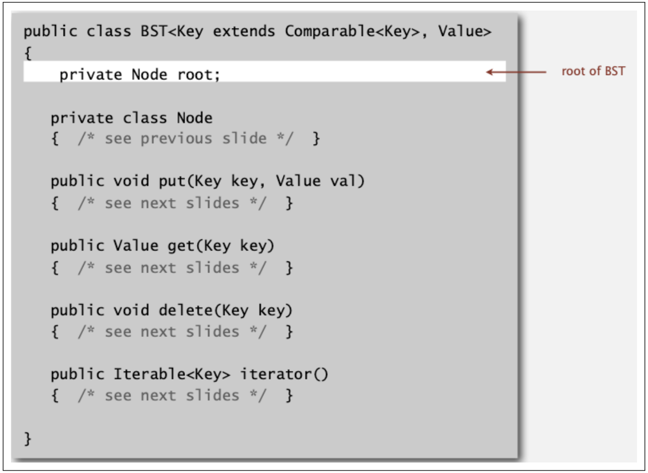


```
private class Node
{
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val)
    {
        this.key = key;
        this.val = val;
    }
}
```

Key and Value are generic types; Key is Comparable

Binary search tree

BST Skeleton

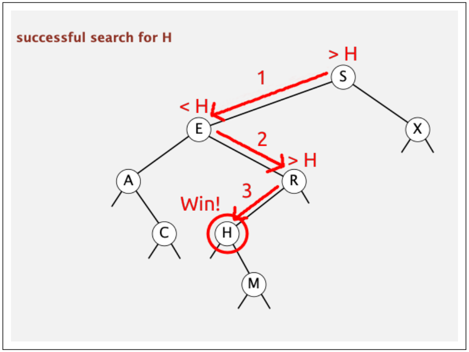


```
public class BST<Key extends Comparable<Key>, Value>
{
    private Node root;

   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 */  }

}
```

Binary Search Tree – Find/Search

BST Search Procedure (k is the searched for key):

- If the current node's key is greater than k, recurse on the left branch.
- If the current node's key is less than k, recurse on the right branch.
- If the current node's key is equal to k, return the value.
- If the branch you're recursing on is empty, the search fails!

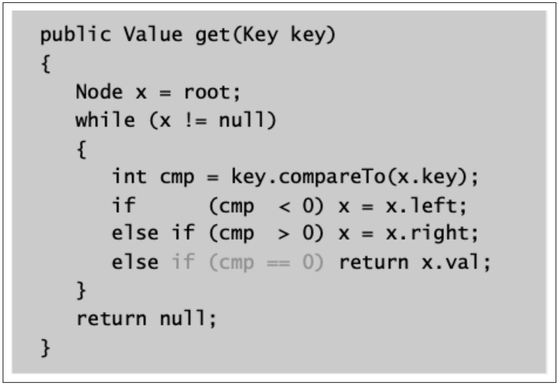


BST Implementation: get()

Get: Return value corresponding to given key, or null if no such key. We look for the *key* starting from the *root* node, and do the below for each node.

Get algorithm outline:

- If $key = node.key$ return node's value.
- If $key < node.key$ recurse on the left subtree.
- If $key > node.key$ recurse on the right subtree.



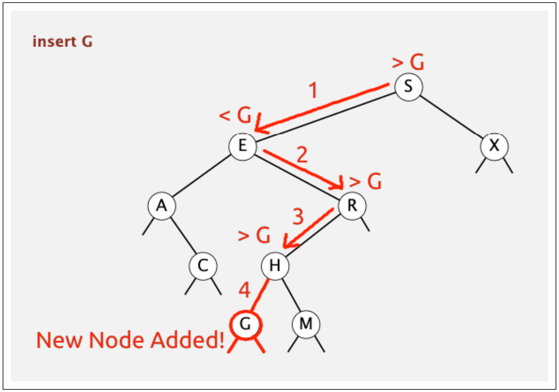
```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

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

Binary Search Tree – Insert

Same as search procedure, except for what happens when you reach a null branch.

- Rather than an empty branch indicating failure, this is where we insert the new node.
- The structure of a BST is *dependent on the order items are added!*
- The best case scenario is a complete binary tree, which is unlikely to happen naturally.



BST `insert()/put()`

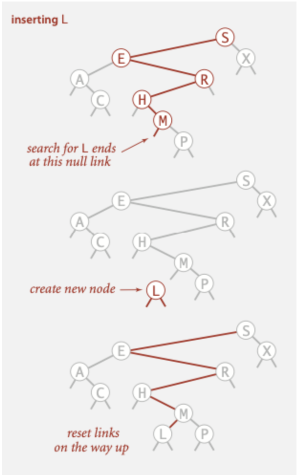
Put: Associates a key with a value.

Search for the key.

- If the key is in tree, overwrite the value.
- If the key is not in the tree, add a new node for it.

Implementation:

- Can be recursive or iterative (similar to `get()`)
- Cost: Number of comparisons is equal to 1 + the depth of node.

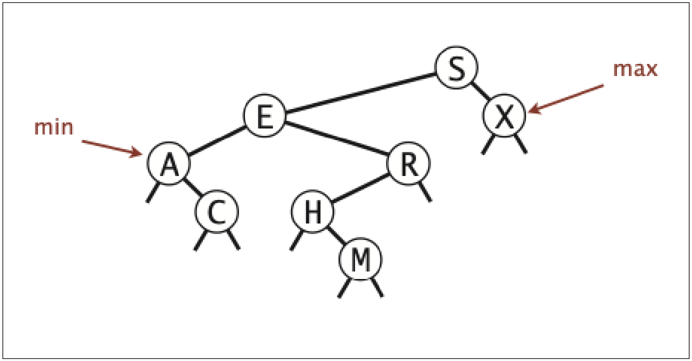

inserting L
S
E
X
A
R
C
H
M
P
search for L ends at this null link
create new node
L
reset links on the way up

Insertion into a BST

BST: Min. and Max. Operations

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

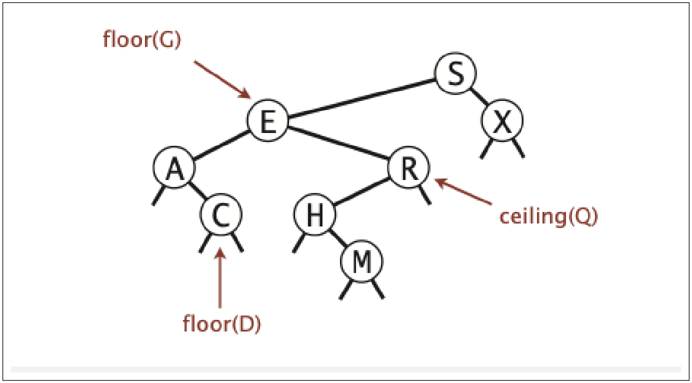
Maximum - returns the largest key in table. Go to the right as far as possible.



BST: Floor and Ceiling Operations

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

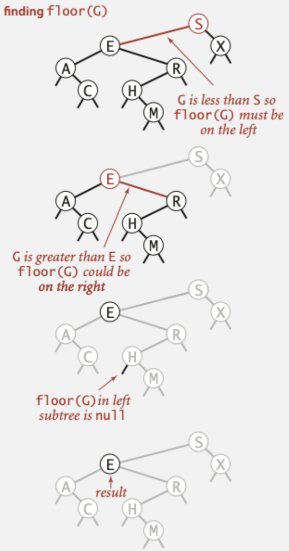
Ceiling - the smallest key in the BST greater than or equal to the key we are cieling...ing...



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.

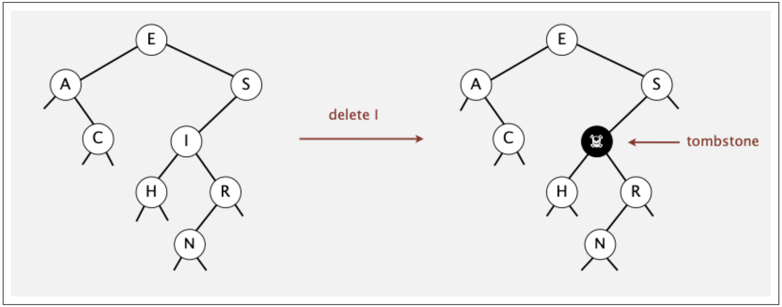
Case 3. [k is greater than the key in the node]
The floor of k is in the right subtree
(if there is any key $\le k$ in right subtree);
otherwise it is the key in the node.


finding floor(G)
G is less than S so floor(G) must be on the left
G is greater than E so floor(G) could be on the right
floor(G) in left subtree is null
result

Lazy Deletion

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).

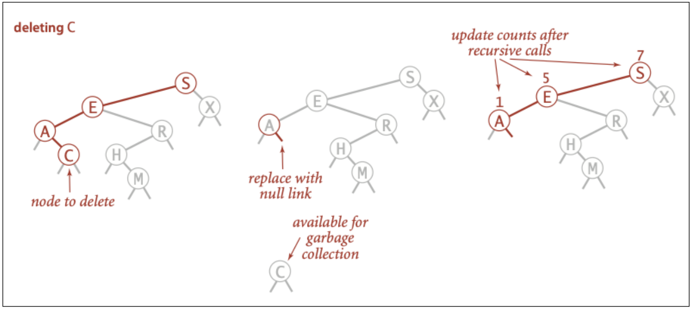


Unsatisfactory solution. Tombstones occupy memory!

BST: Hibbard Deletion

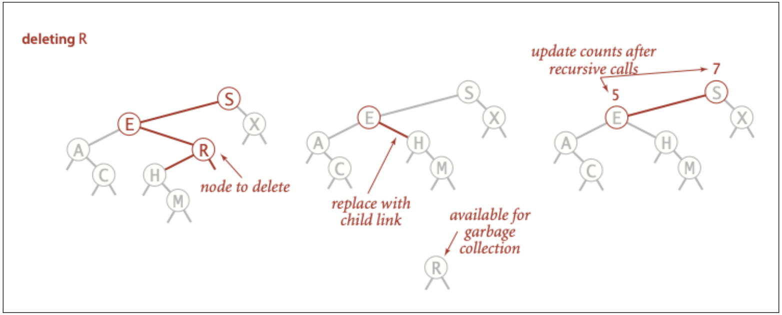
To delete a node with key k: search for node t containing key k.

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


deleting C
node to delete
replace with null link
available for garbage collection
update counts after recursive calls
7
5
1
S
E
A
C
X
R
H
M

BST: Hibbard Deletion

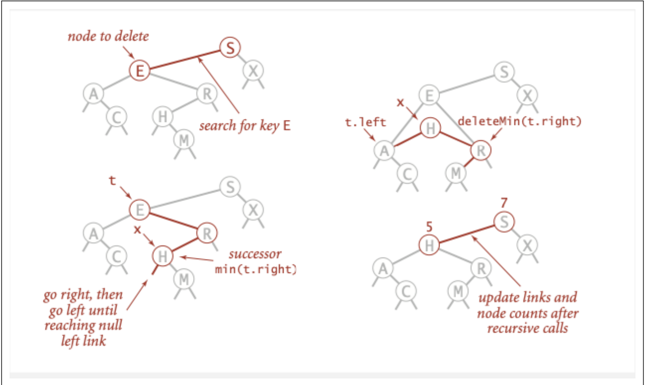
Case 1: [1 child] Delete t and connect its single child to t's parent.


deleting R
node to delete
replace with child link
available for garbage collection
update counts after recursive calls
7
5

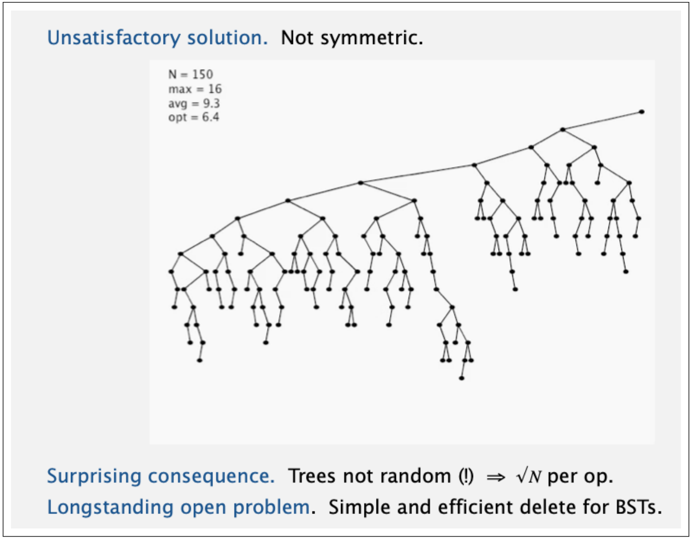
BST: Hibbard Deletion:

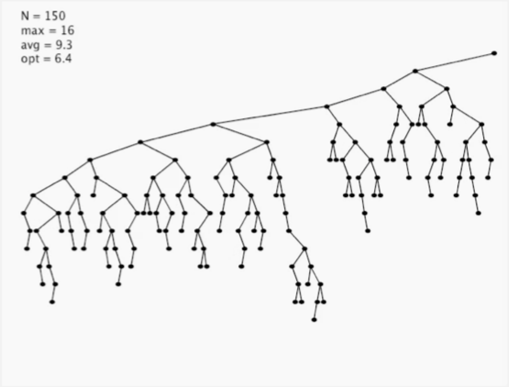
Case 2. [2 children]

- 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).


node to delete
search for key E
t
x
successor
min(t.right)
go right, then go left until reaching null left link
t.left
x
deleteMin(t.right)
7
5
update links and node counts after recursive calls

BST: Hibbard deletion analysis

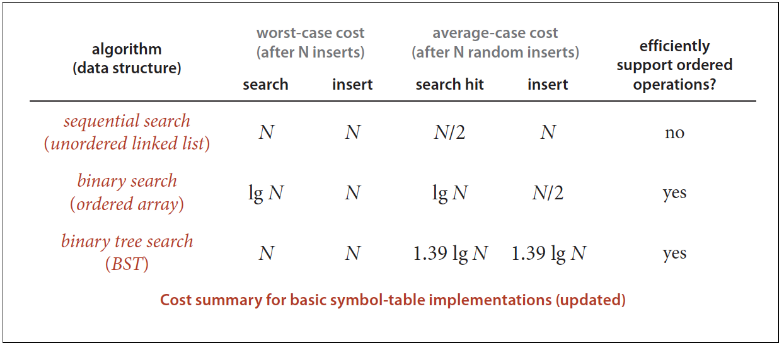

Unsatisfactory solution. Not symmetric.



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

Longstanding open problem. Simple and efficient delete for BSTs.

BST Cost

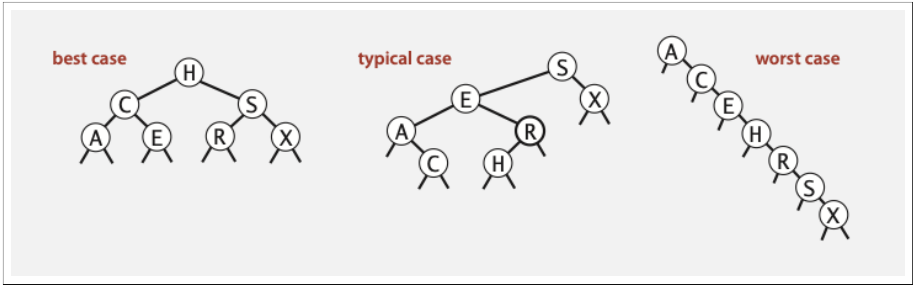


algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
sequential search (unordered linked list)	N	N	$N/2$	N	no
binary search (ordered array)	$\lg N$	N	$\lg N$	$N/2$	yes
binary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

Cost summary for basic symbol-table implementations (updated)

BST Tree Shape

- One set of keys can be stored in many differently structured BSTs.
- Remember: the number of comparisons for search/insert is proportional to the depth of the tree!



Tree shape, *and therefore runtime*, depends on the order of insertion! Not fantastic!

BST Tree Randomization

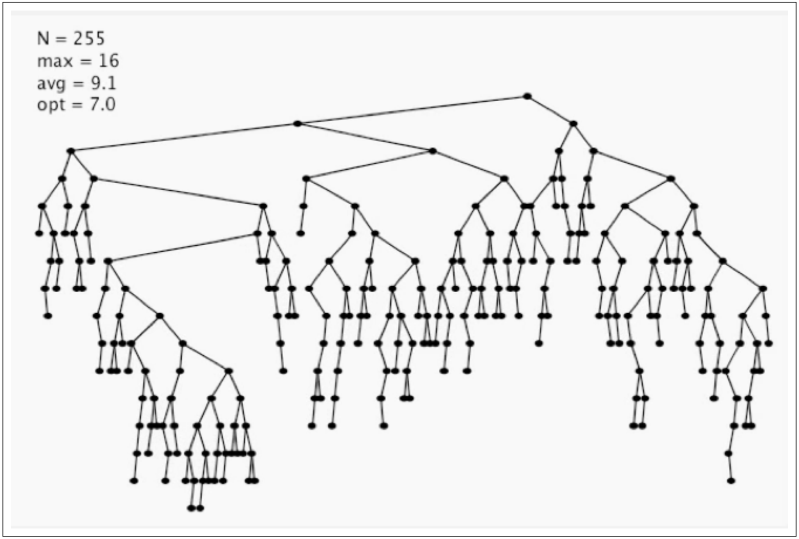
Assume that the keys inserted in a uniform random order.

- That is, the probability that any remaining node is the next node to be added is a **uniform probability distribution**.

In a BST built from N random keys:

- Search hits/misses and insertions about $1.39 \log_2 N$ comparisons on average.

BST insertion: random order visualization



Tree Balancing Algorithms

There are several tree balancing algorithms which are beyond the scope of this course:

- T-Tree - Used in main-memory databases such as mySQL
- Treap - Randomizes tree structure with every insertion.
- Red-Black Tree - Nodes are dynamically "colored" red or black, which informs insert procedures.
- B-Tree - Generalizes BSTs to allow nodes with more than 2 children. Good for file systems.
- 2-3 Tree - Specific type of B-Tree, where nodes either have 2 children and one datum or 3 children and two data.