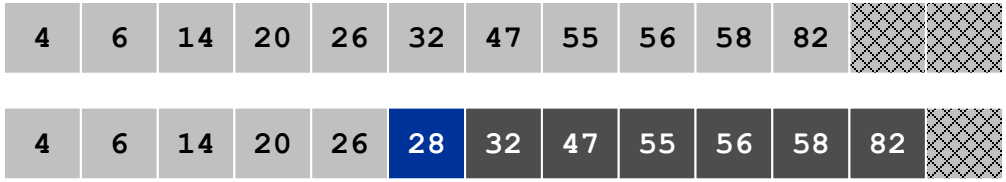
Binary Search Trees

Unsorted array.

- Put: add key to the end (if not already there).
- Get: scan through all keys to find desired value.

Sorted array.

- Put: find insertion point, and shift all larger keys right.
- Get: binary search to find desired key.



insert 28

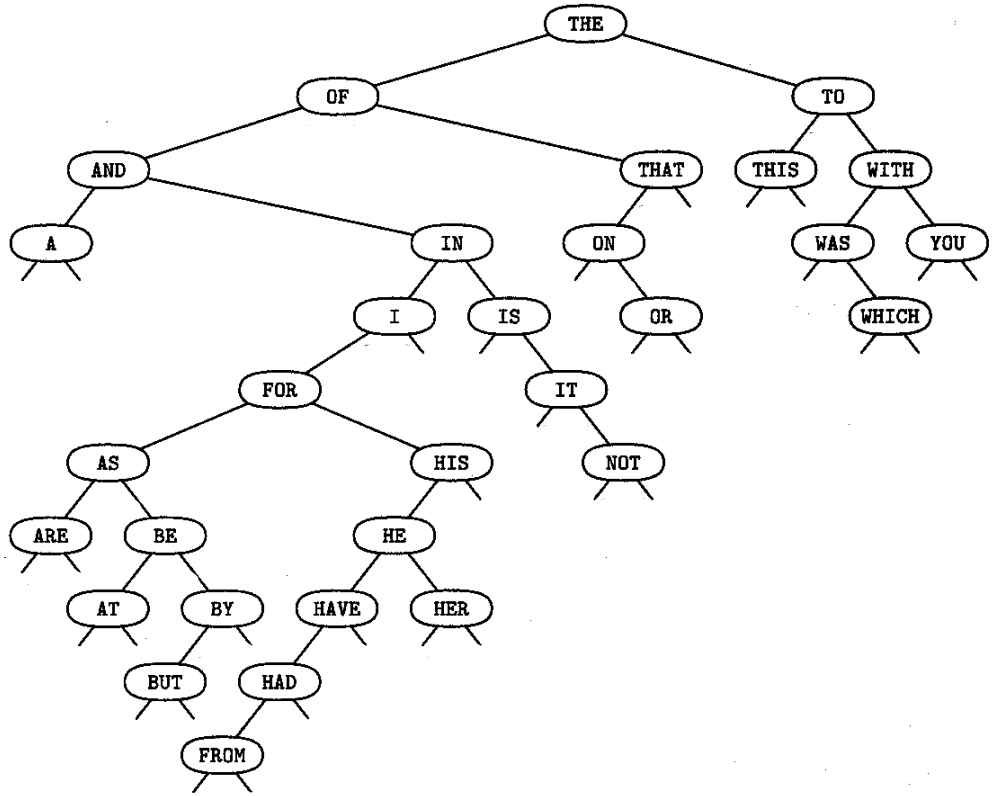
Unordered array. Hopelessly slow for large inputs.

Ordered array. Acceptable if many more searches than inserts; too slow if many inserts.

	Running Time		Frequency Count			
implementation	get	put	Moby	100K	200K	1M
unordered array	N	N	170 sec	4.1 hr	-	-
ordered array	$\log N$	N	5.8 sec	5.8 min	15 min	2.1 hr

Challenge. Make all ops logarithmic.

Binary Search Trees



Reference: Knuth, The Art of Computer Programming

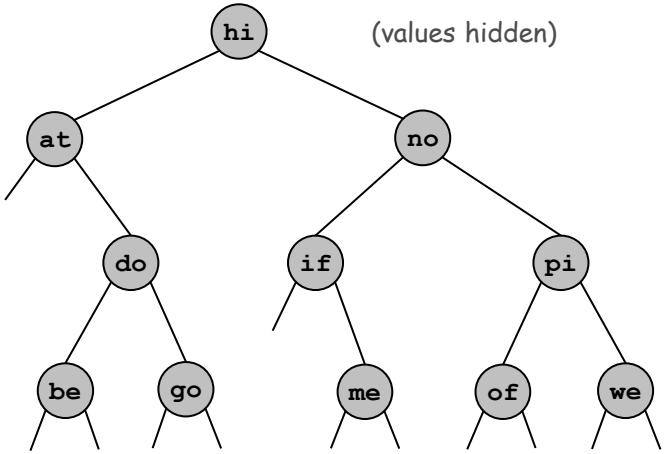
Binary Search Trees

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

Binary tree is either:

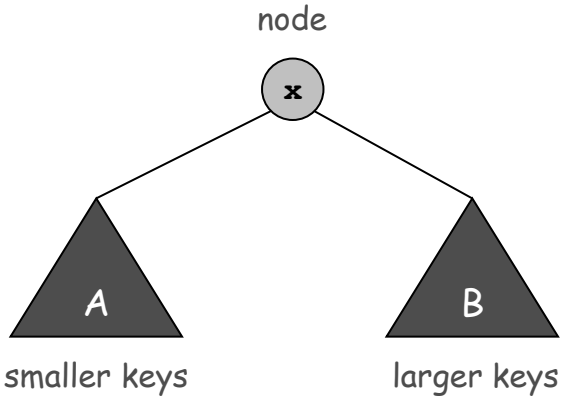
- Empty.
- A key-value pair and two binary trees.

we suppress values from figures

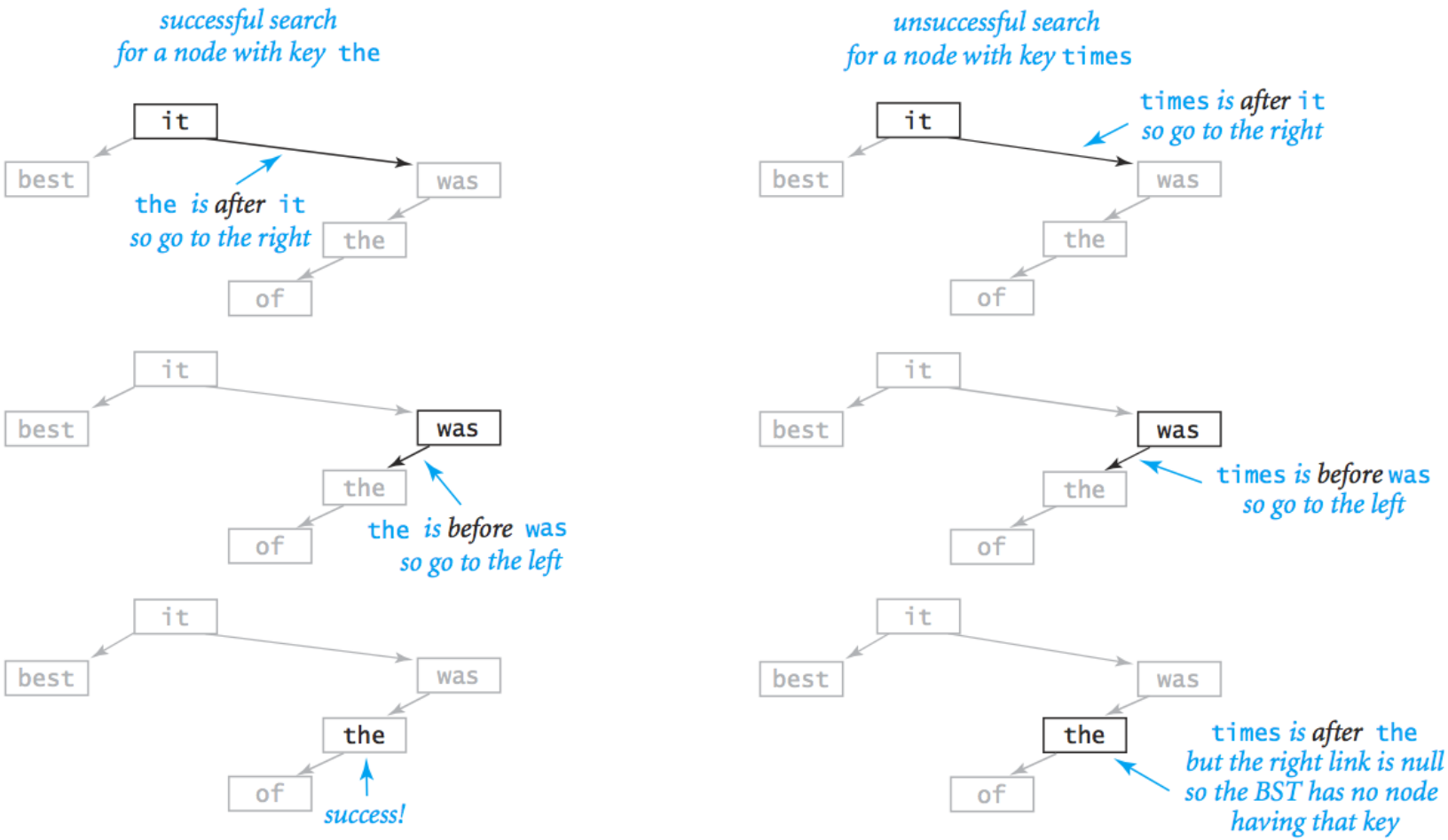


Symmetric order.

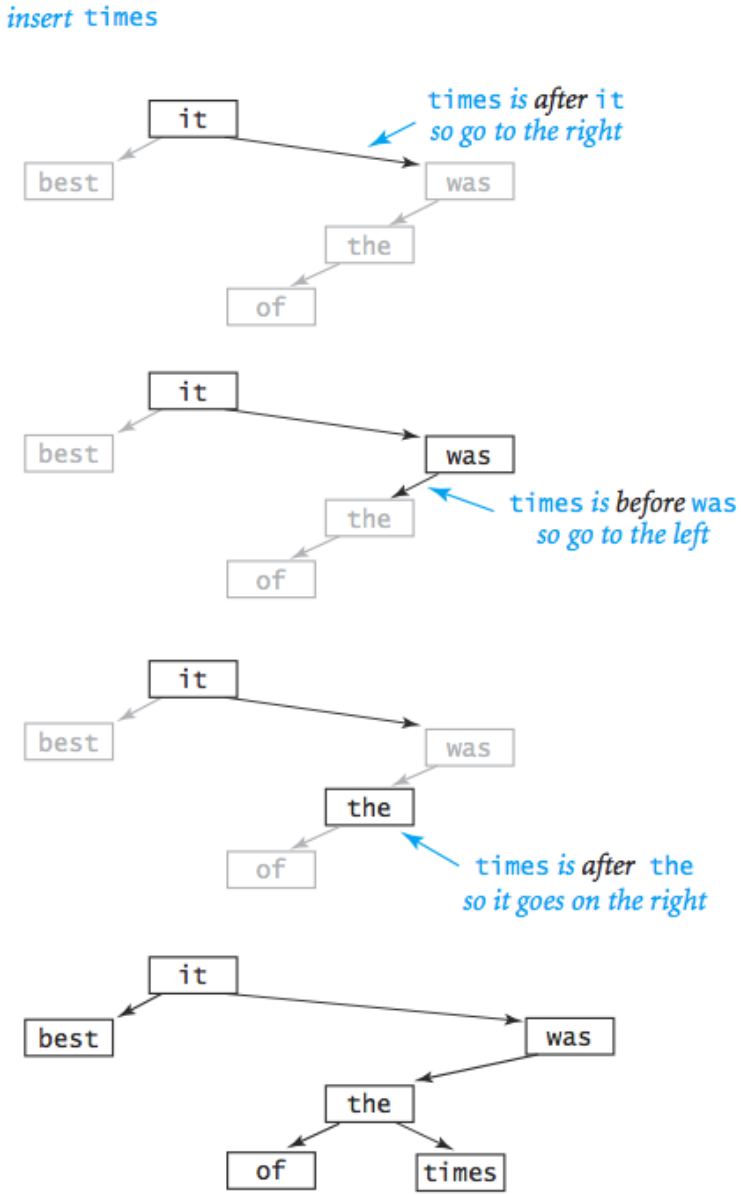
- Keys in left subtree are smaller than parent.
- Keys in right subtree are larger than parent.



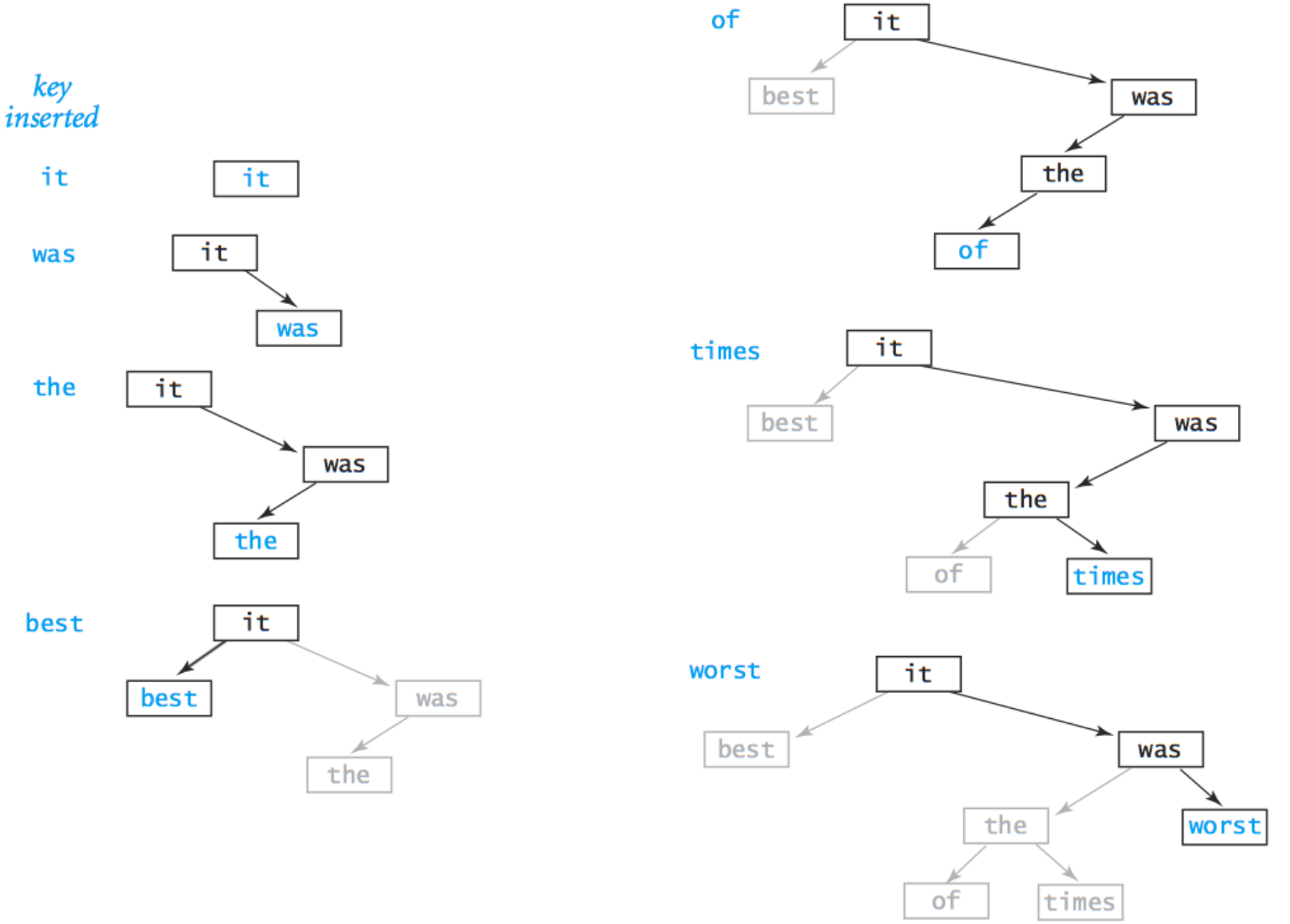
BST Search


successful search
for a node with key the
it
best
was
the
of
the is after it
so go to the right
the is before was
so go to the left
success!
unsuccessful search
for a node with key times
times is after it
so go to the right
times is before was
so go to the left
times is after the
but the right link is null
so the BST has no node
having that key

BST Insert


insert times
it
best
was
the
of
times is after it
so go to the right
it
best
was
the
of
times is before was
so go to the left
it
best
was
the
of
times is after the
so it goes on the right
it
best
was
the
of
times

BST Construction



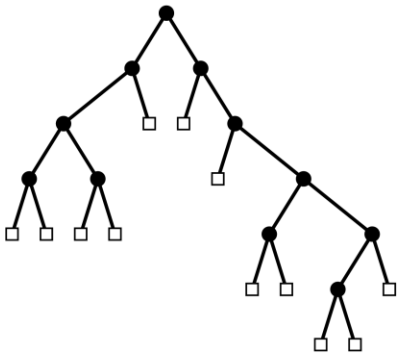
Binary Search Tree: Java Implementation

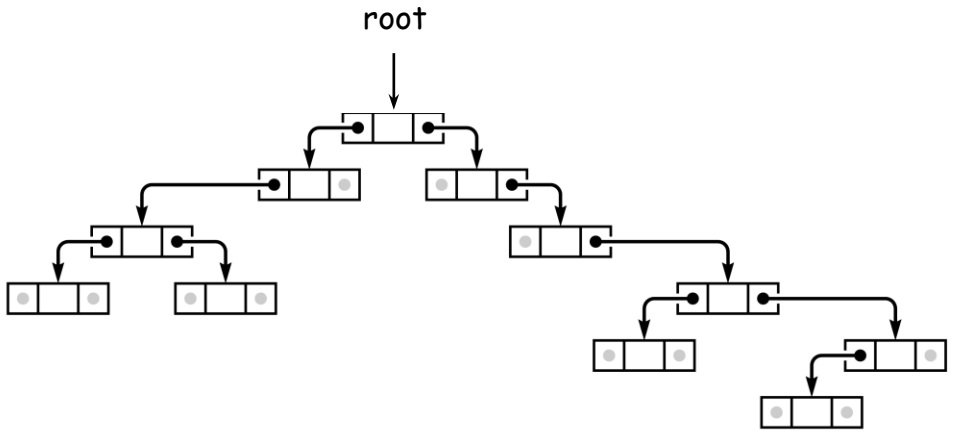
To implement: use two links per `Node`.

A `Node` is comprised of:

- A key.
- A value.
- A reference to the left subtree.
- A reference to the right subtree.

```
private class Node {
   private Key key;
   private Val val;
   private Node left;
   private Node right;
}
```





BST: Skeleton

BST. Allow generic keys and values.

requires `Key` to provide `compareTo()` method; see book for details

```
public class BST<Key extends Comparable<Key>, Val> {

   private Node root;   // root of the BST

   private class Node {
      private Key key;
      private Val val;
      private Node left, right;

      private Node(Key key, Val val) {
         this.key = key;
         this.val = val;
      }
   }

   public void put(Key key, Val val)  { … }
   public Val get(Key key)            { … }
   public boolean contains(Key key)   { … }

}
```

BST: Search

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

```
public Val get(Key key) {
   return get(root, key);
}

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

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

negative if less,
zero if equal,
positive if greater

BST: Insert

Put. Associate `val` with `key`.

- Search, then insert.
- Concise (but tricky) recursive code.

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

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

overwrite old value with new value

BST Implementation: Practice

Bottom line. Difference between a practical solution and no solution.

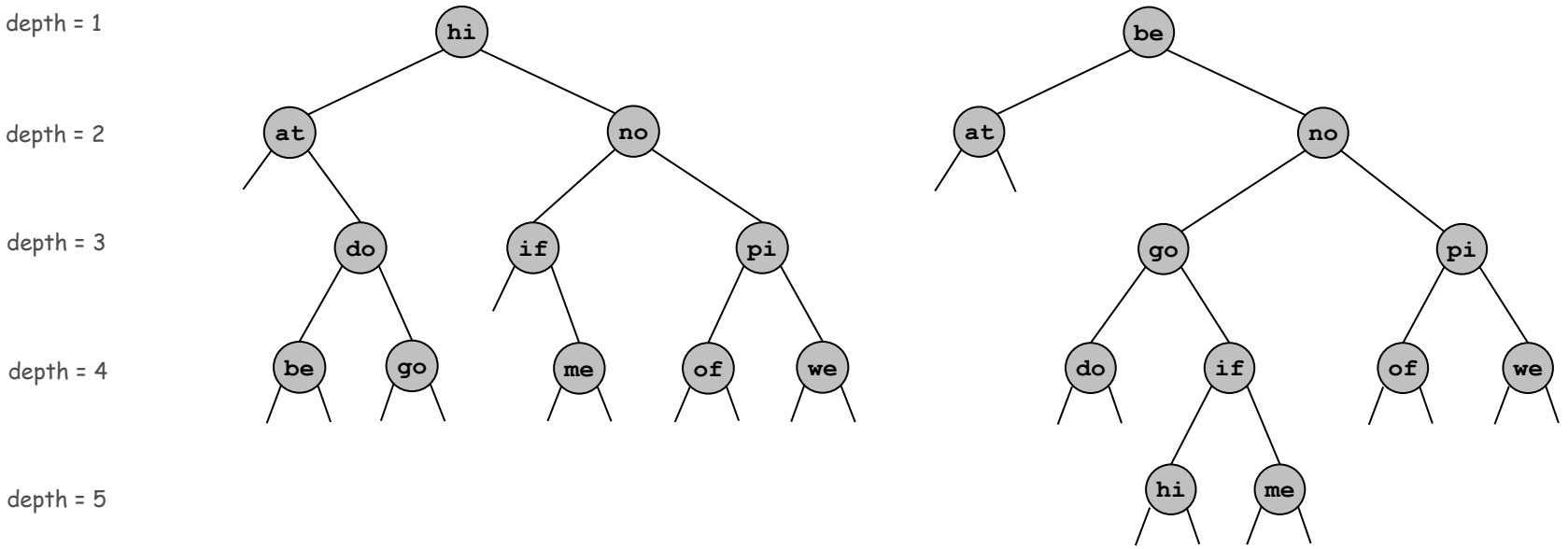
	Running Time		Frequency Count			
implementation	get	put	Moby	100K	200K	1M
unordered array	N	N	170 sec	4.1 hr	-	-
ordered array	$\log N$	N	5.8 sec	5.8 min	15 min	2.1 hr
BST	?	?	.95 sec	7.1 sec	14 sec	69 sec

BST: Analysis

Running time per put/get.

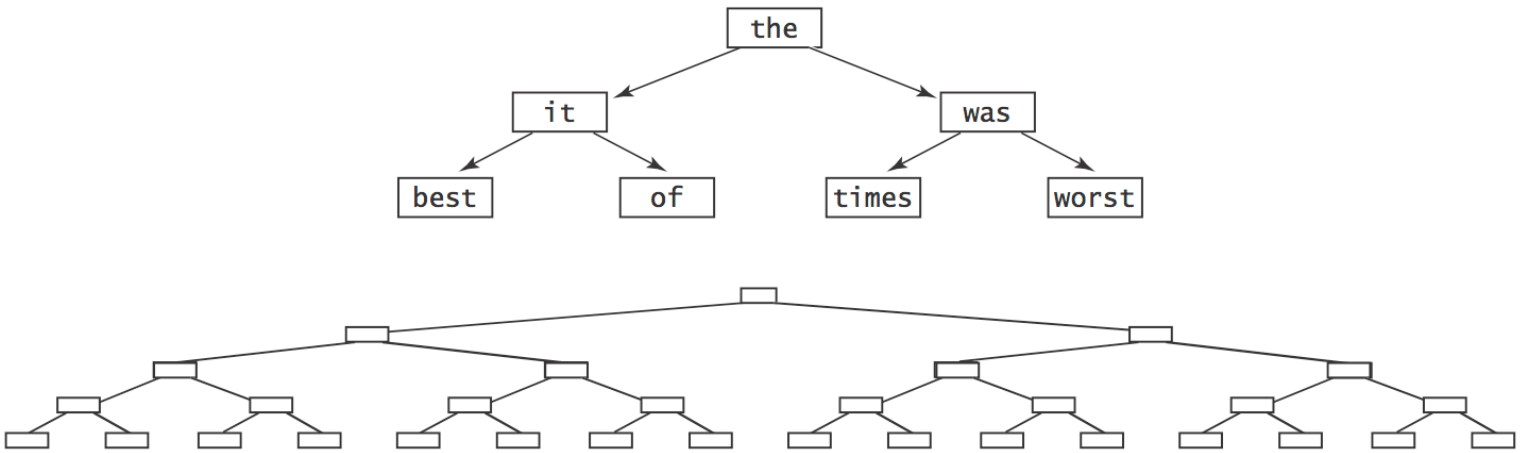
- There are many BSTs that correspond to same set of keys.
- Cost is proportional to depth of node.

number of nodes on path from root to node



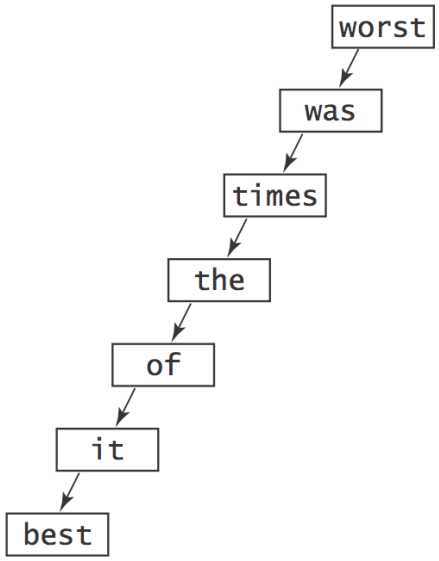
BST: Analysis

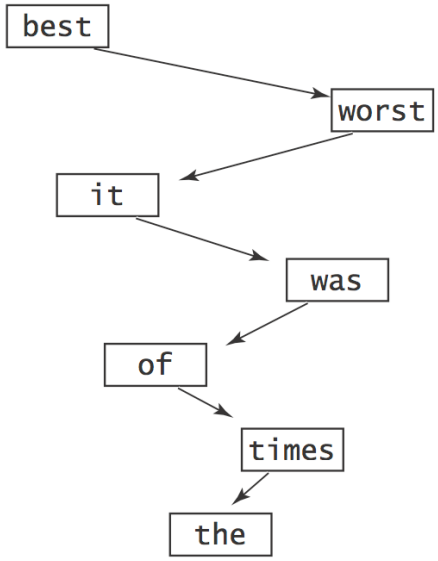
Best case. If tree is perfectly balanced, depth is at most $\lg N$.

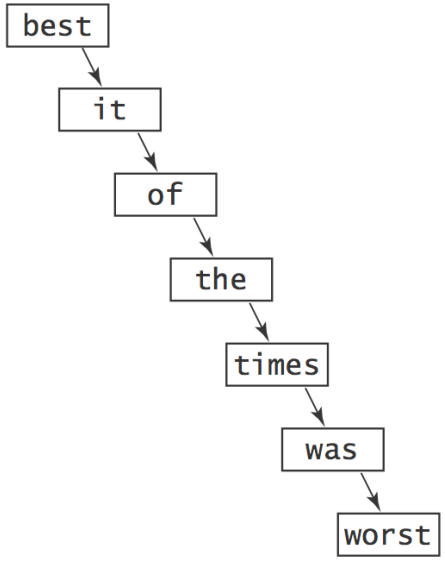


BST: Analysis

Worst case. If tree is unbalanced, depth is N.

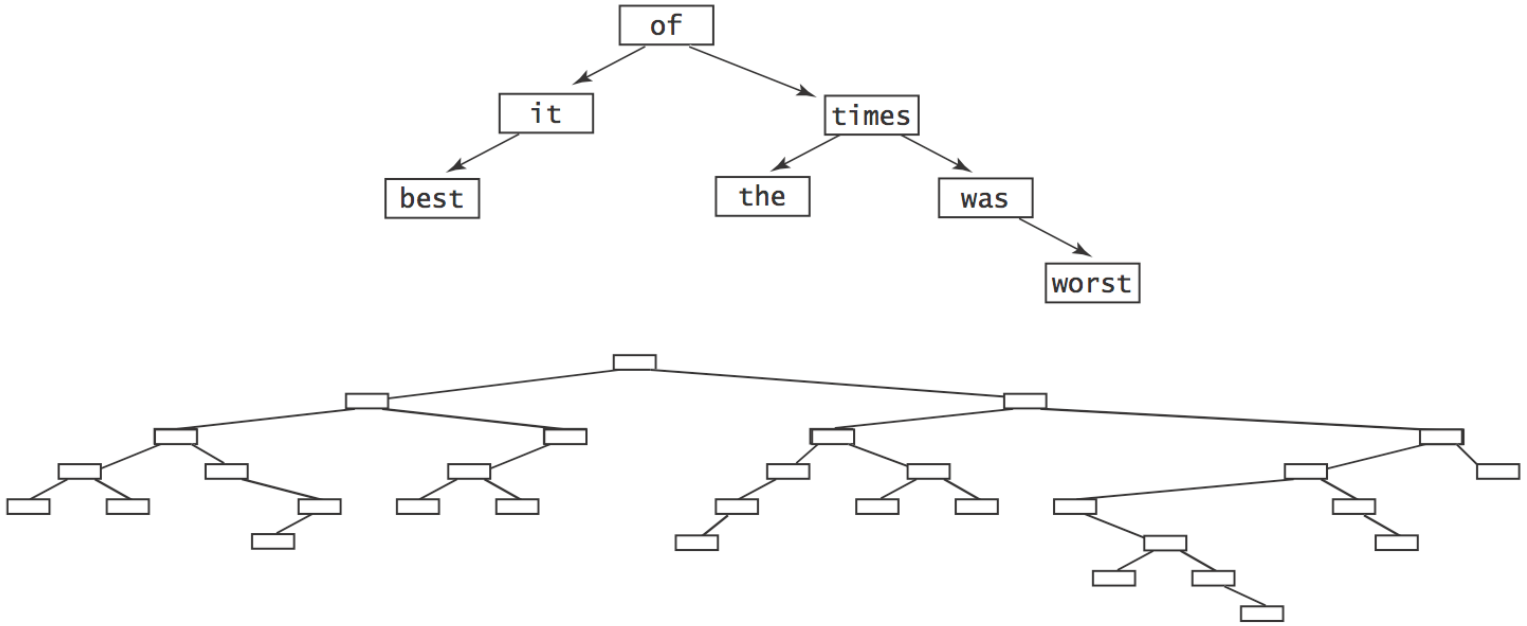






BST: Analysis

Average case. If keys are inserted in random order, average depth is $2 \ln N$.



Typical BSTs constructed from randomly ordered keys

Symbol Table: Implementations Cost Summary

BST. Logarithmic time ops if keys inserted in random order.

	Running Time		Frequency Count			
implementation	get	put	Moby	100K	200K	1M
unordered array	N	N	170 sec	4.1 hr	-	-
ordered array	$\log N$	N	5.8 sec	5.8 min	15 min	2.1 hr
BST	$\log N$ †	$\log N$ †	.95 sec	7.1 sec	14 sec	69 sec

† assumes keys inserted in random order

Q. Can we guarantee logarithmic performance?

BST: Iterative Search

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

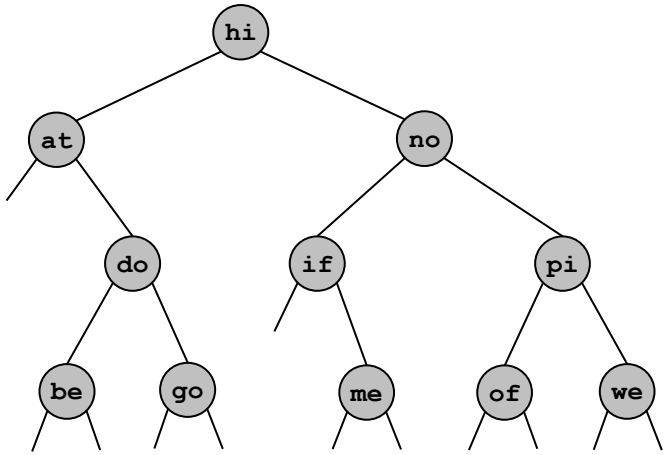
```
public Val get(Key key) {
   Node x = root;
   while (x != null) {
      int cmp = key.compareTo(x.key);
      if      (cmp < 0) x = x.left;
      else if (cmp > 0) x = x.right;
      else return x.val;
   }
   return null;
}

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

Preorder Traversal

Preorder traversal.

- Visit node.
- Recursively visit left subtree.
- Recursively visit right subtree.



preorder: hi at do be go no if me pi of we

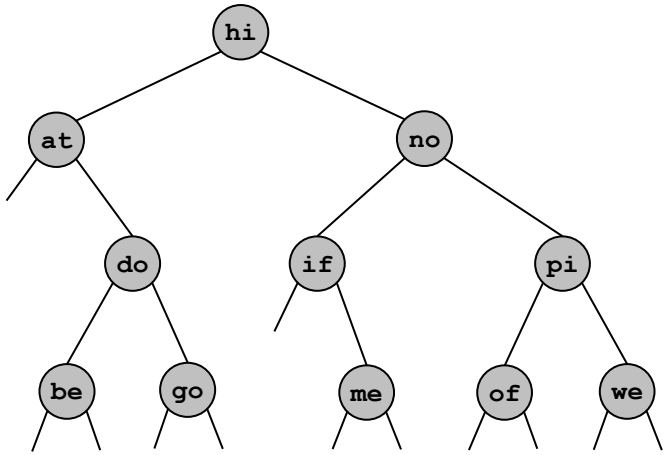
```
public preorder() { preorder(root); }

private void preorder(Node x) {
   if (x == null) return;
   StdOut.println(x.key);
   preorder(x.left);
   preorder(x.right);
}
```

Postorder Traversal

Postorder traversal.

- Recursively visit left subtree.
- Recursively visit right subtree.
- Visit node.




```
postorder: be go do at me if of we pi no hi
```

```
public postorder() { postorder(root); }

private void postorder(Node x) {
   if (x == null) return;
   postorder(x.left);
   postorder(x.right);
   StdOut.println(x.key);
}
```