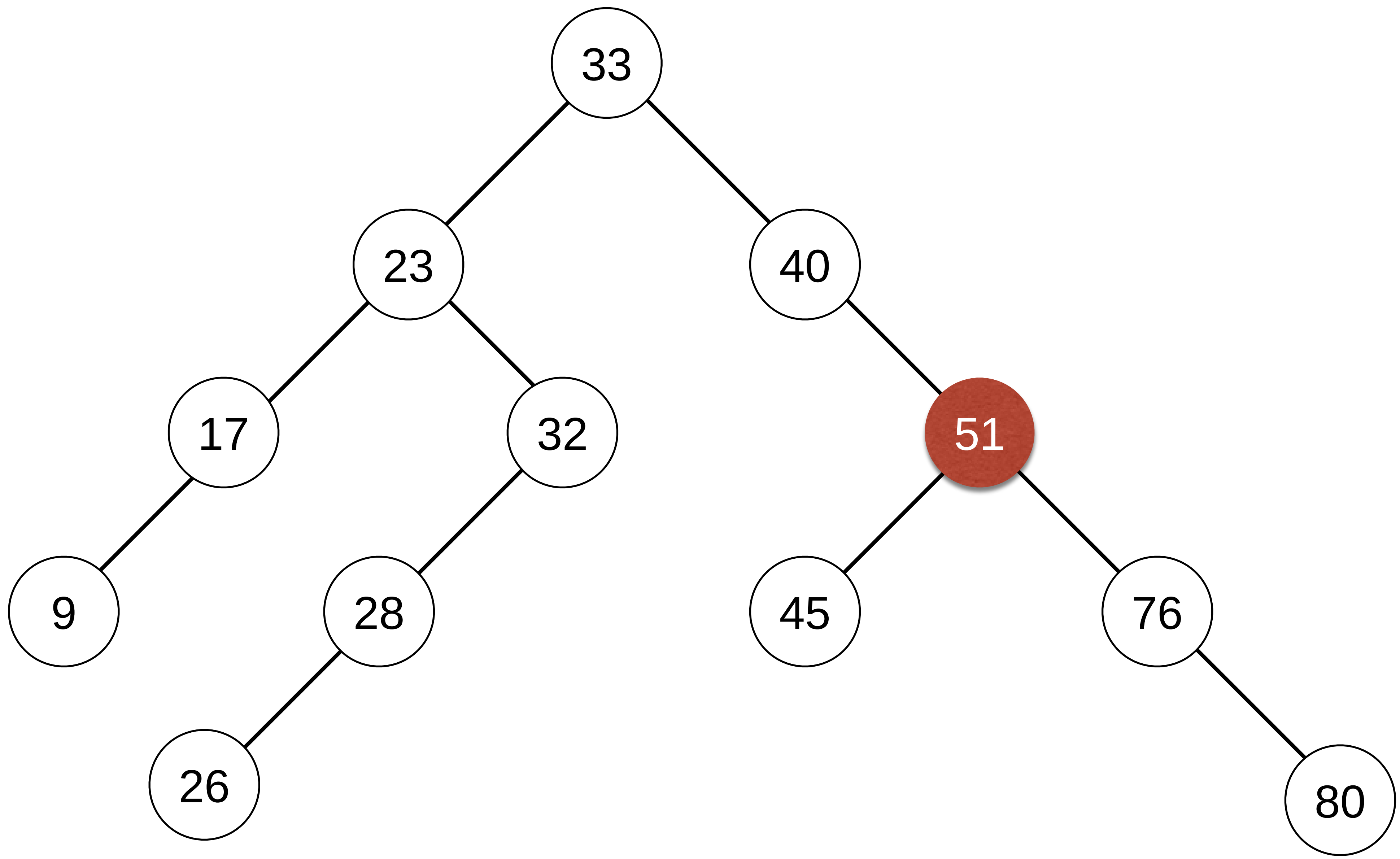
Binary Search Trees

continued

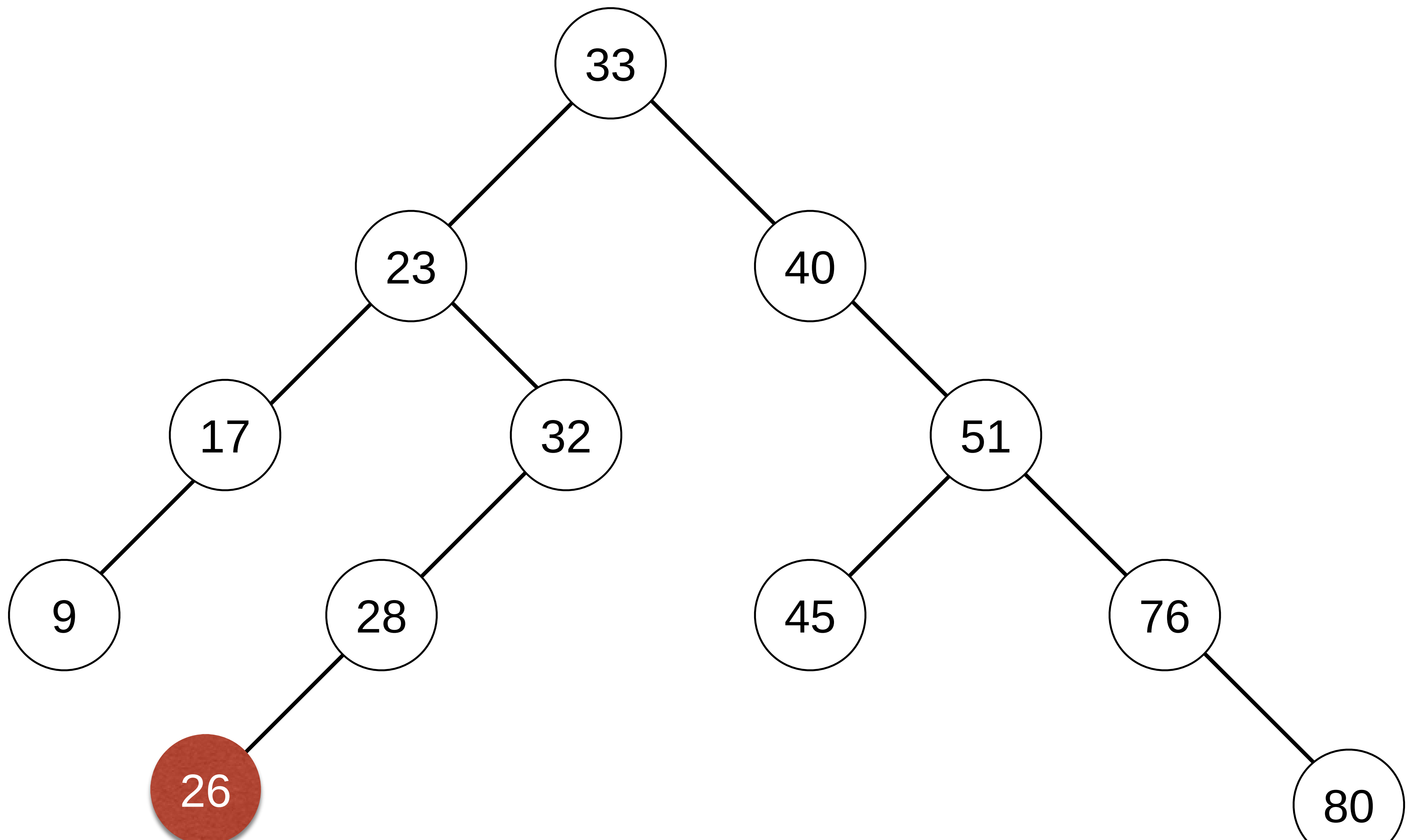
Draw the BST

- Insert the elements in this order 50, 70, 30, 37, 43, 81, 12, 72, 99

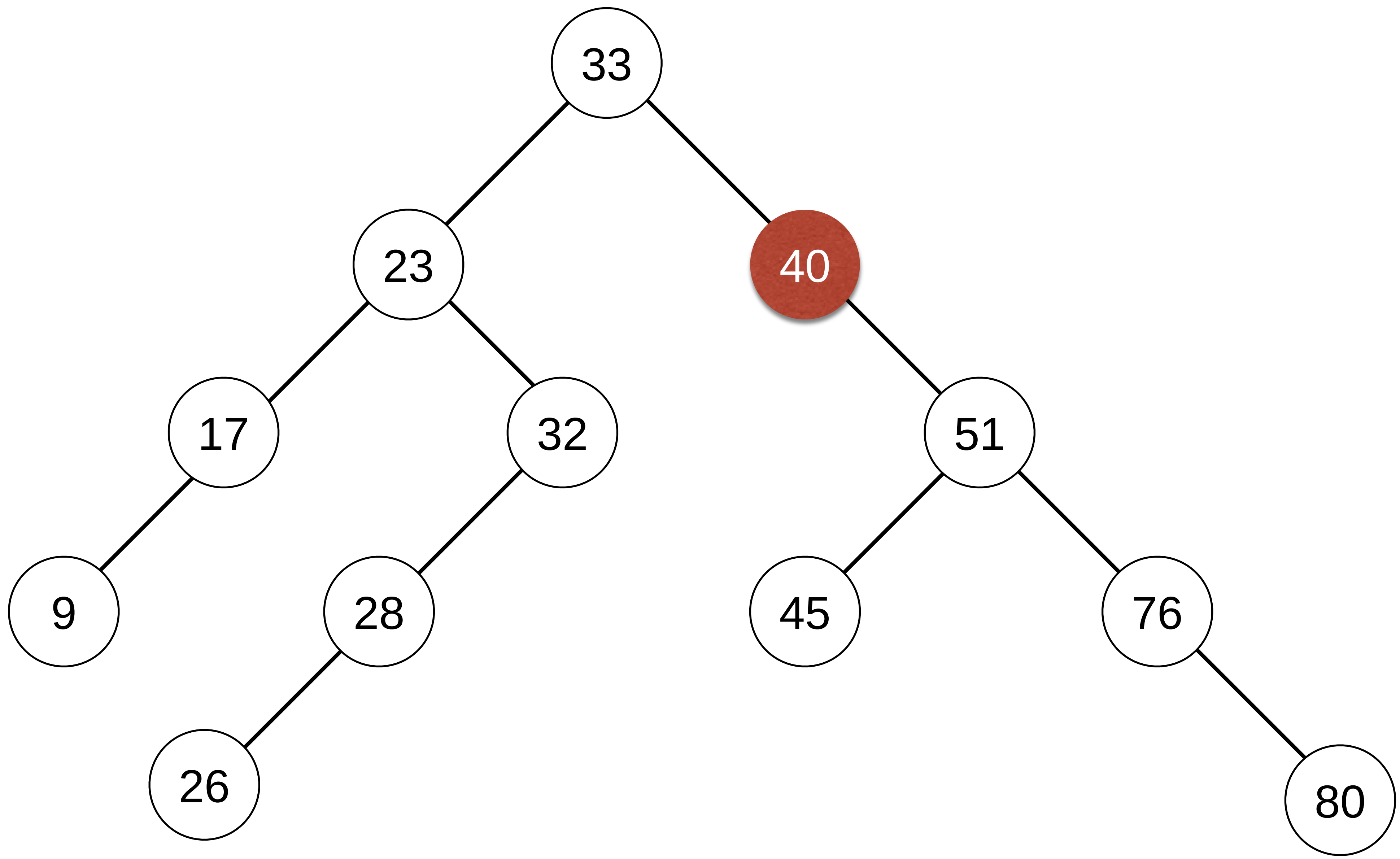
Delete the red element



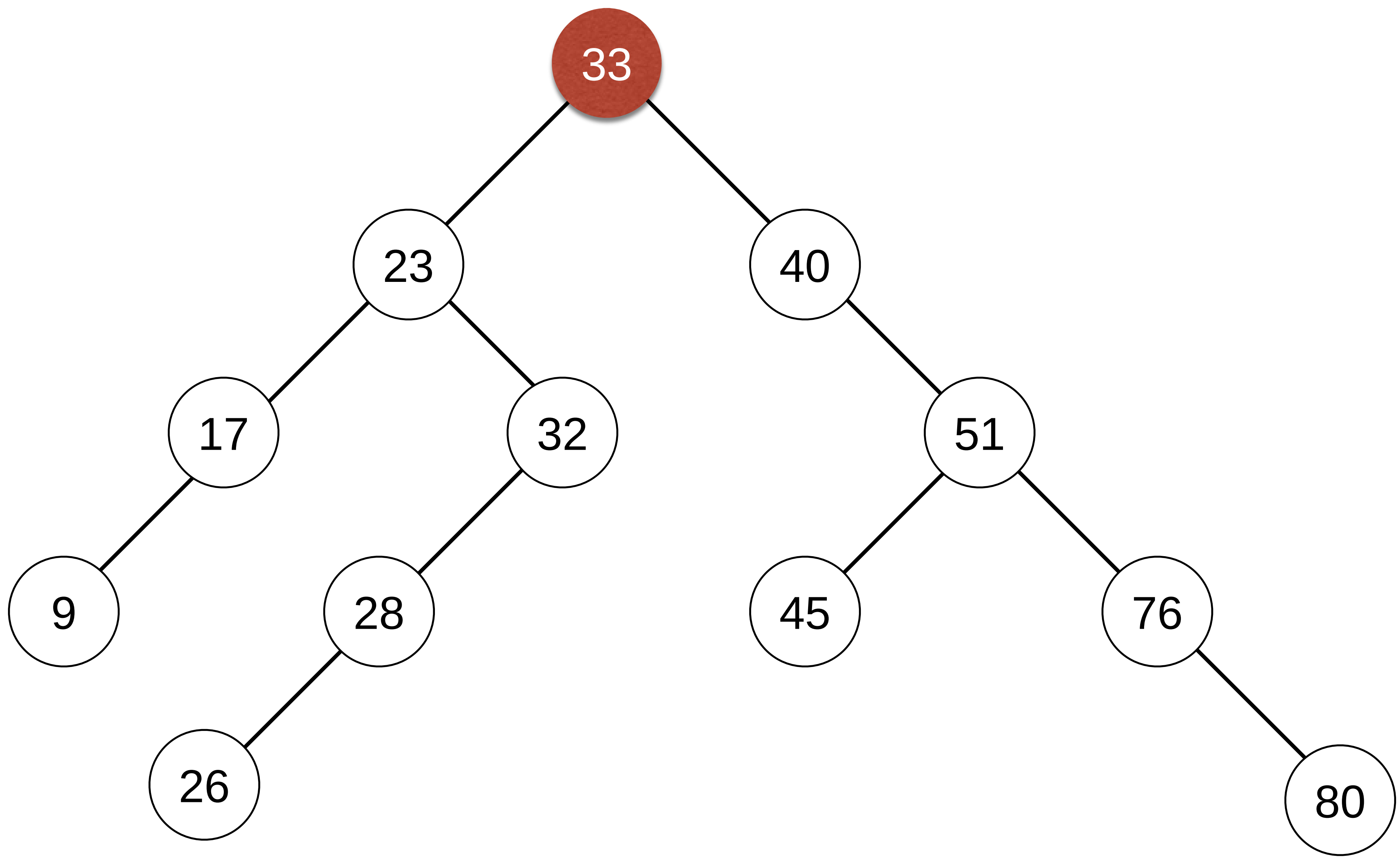
Delete the red element



Delete the red element



Delete the red element



Deleting Code

- We need to find the node the value is stored in.
- There are three cases
  - the node has two children
  - the node has one child
  - the node has no children

```
def delete(self, value):
    """Delete value from the BST."""
    node = self.locate(value) # saw this last lecture
    if node:
        node.delete_this_node()

def delete_this_node(self):
    left = self.left
    right = self.right
    if left and right:  # two children
        self.delete_with_children()
    elif left or right: # one child
        self.delete_with_child()
    else:               # no children
        self.delete_no_children()
```

Deleting without children

- Just delete the node and fix up its parent.

```
def delete_no_children(self):
    if self.parent:
        if self.parent.left == self:
            self.parent.left = None
        else:
            self.parent.right = None
    else: # special case the root node
        self.value = None
```

Deleting with one child

- Delete the node and shift its child up to take its place by cha

```
def delete_with_child(self):
    child = self.left if self.left else self.right
    if self.parent:
        if self.parent.left == self:
            self.parent.left = child
        else:
            self.parent.right = child
        child.parent = self.parent
    else: # special case the root node
        self.replace(child)
```

Replacing the node contents

```
# We have deleted the root value however we don’t want
# to remove the root node (as this defines our BST).
# So we put new info into the root node.

    def replace(self, other):
        """Replace this node with the values from other.

        Also need to reattach other's children.
        """
        self.value = other.value
        self.left = other.left
        if self.left: self.left.parent = self
        self.right = other.right
        if self.right: self.right.parent = self
```

Deleting with children

- Replace the value in the node with its inorder successor.
- We also have to delete the inorder successor node.

```
def delete_with_children(self):
   replacement = self.right.min() # the successor
   successor_value = replacement.value
   replacement.delete_this_node() # max of one child of this
   self.value = successor_value
```

Inorder successor code

```
def min(self):
    """Returns the left most node of the BST."""
    min_node = self
    while min_node.left:
        min_node = min_node.left
    return min_node
```

Big O of the operations

- It depends on how the tree has been constructed.
- A full binary tree or one close to it (every node not a leaf node has two children) is ideal.
- The algorithms depend on how far down the tree you have to go in order to insert or delete nodes.
- With a full binary tree the number of nodes in level d is 2^d. And the number of nodes in a tree of height h is $2^{h+1} - 1$.

Balanced and Unbalanced

- A balanced tree has approximately the same number of nodes in the left subtree as in the right subtree. This will give O(log n) operations for retrieval, insertion and deletion.

- Unbalanced trees have runs of single children. This can be as bad as a simple list and so has O(n) operations.

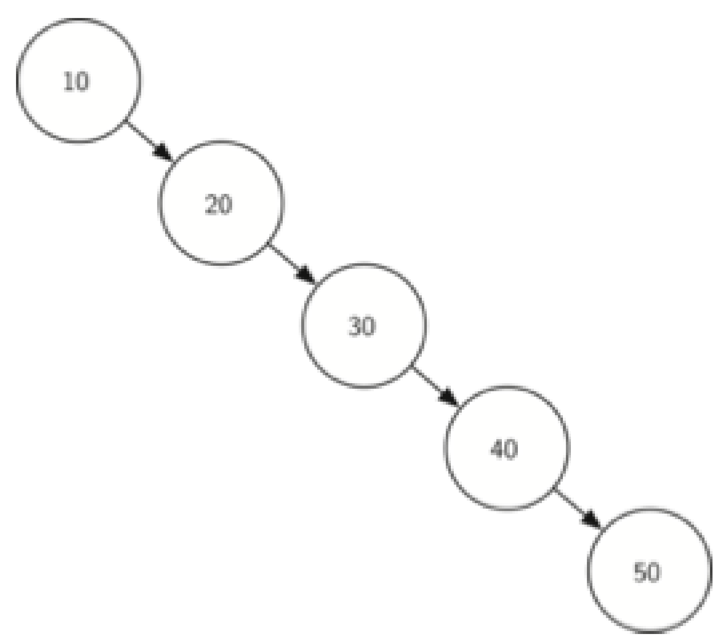


Figure 6.23: A skewed binary search tree would give poor performance

Regular Expressions

Scanning text

- In many applications we have to accept strings of information and extract parts of those strings.
  - e.g. a program which reads the files in a directory and finds text files which start with a university UPI - rshe001.txt, afer002.txt, alux003.txt
- We can read each character of the file names and compare them to what they should be, or we can use regular expressions.
- Regular expressions or regexes are faster and more powerful - but the regex language has to be learnt.

What is a regular expression?

- They are expressions designed to match sequences of characters in strings.
- They use their own language to define these expressions.
  - We will only look at a tiny subset of some of the things you can do with regular expressions.

Simple matching

- Regular expressions are compared with strings (or vice versa) looking for matches between the sequence of characters in the string and the regular expression.
- Most characters in a regular expression just mean the character.
  - e.g. the regular expression `robert` would match the string `robert`
  - but we can match the strings `Robert` or `robert` with the regex `[Rr]obert`
    - the brackets make a character group and either of the characters can be acceptable in the string

Matching a University UPI

- Let’s start with the “simple” University UPI of 4 letters followed by 3 digits.
- This can be matched with the regular expression
  - `[a-z][a-z][a-z][a-z][0-9][0-9][0-9]`
  - this uses the “`-`” shortcut which indicates a range of characters
    - e.g. we could have used `[0123456789]` for a digit
  - remember the brackets indicate a character class - one (and only one) of the values in the class must match, so in this case we have 4 lowercase letters followed by 3 digits

Making it smaller

- We could also match a UPI with these regular expressions
  - `[a-z][a-z][a-z][a-z]\d\d\d`
    - where `\d` is shorthand for `[0-9]` the digit character class
  - `[a-z]{4}\d{3}`
    - the numbers in `{}` say how many of each preceding character we are wanting

Trickier

- Some UPIs only have 3 letters (for people with 2 letter surnames)
- We can match those with
  - `[a-z]{3,4}\d{3}`
  - 3 or 4 characters from a to z.

Some special characters

- Some characters (we have already seen [and] and { and })are special.
- If we want to match them they have to be “escaped” by a \.
- Other special characters include . ? + and *

What do they mean?

- A full stop `.` matches any character
- A question mark `?` means the character before it is optional e.g. `colou?r` matches both `colour` and `color`
- A plus `+` means one or more of the character before it e.g. `ab+a` matches `abba` and `aba`
- A star `*` means zero or more of the character before it e.g. `ab*a` matches `abba` and `aa`

Finding a phone number

- We want to write a regular expression which would match a phone number like (09)876-1234
  - We have to escape the special characters “(” and “)”.
  - The “-” is only special within [and]
    - `\(\d{2}\)\d{3}-\d{4}`

Doing this in Python

- The regular expression module is called `re`

```
import re
```

- We write our regexes as *raw* strings e.g. `r'[aA]nd'`
- The `r` tells Python not to do any escapes, this means any `\` escape characters get sent to the regex interpreter.

search

- The search method is the normal way of checking a string with a regex.
- It returns a match object if the search was successful or `None` otherwise.
- A match object has several useful methods but the most useful are `start()` and `end()`. They tell us where the match started in the string and where it ended (the index one past the end as traditional in Python).

Handy display function

```
import re

def show_regexp(regex, string):
    print(regex, '-', string)
    match = re.search(regex, string)
    if match:
        start = match.start()
        end = match.end()
        print(string[:start],’<<',string[start:end],
              '>>',string[end:],sep='')
    else:
        print('no match')
    print()
```

Input and Output

```
show_regexp(r'[rR]obert', ‘proberty')

# produces

[rR]obert - proberty
p<<robert>>y
```