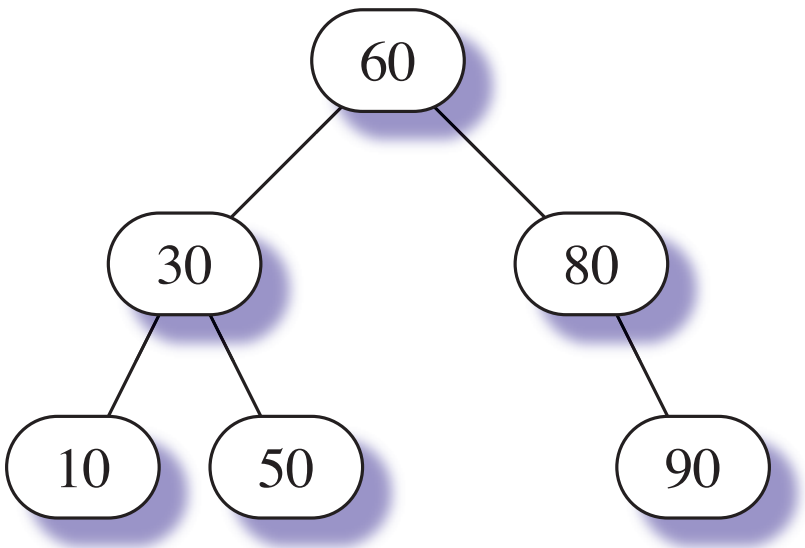
The Binary Search Tree Visitor Implementation

Definition of a binary tree

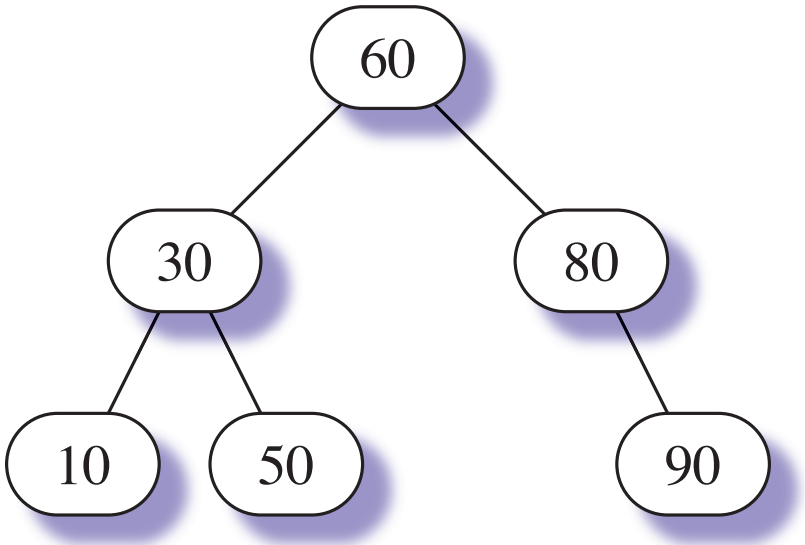
- The empty tree is a binary tree.
- A nonempty tree has three parts:
  - A left child, which is a binary tree.
  - A data value.
  - A right child, which is a binary tree.

Definition of a binary search tree

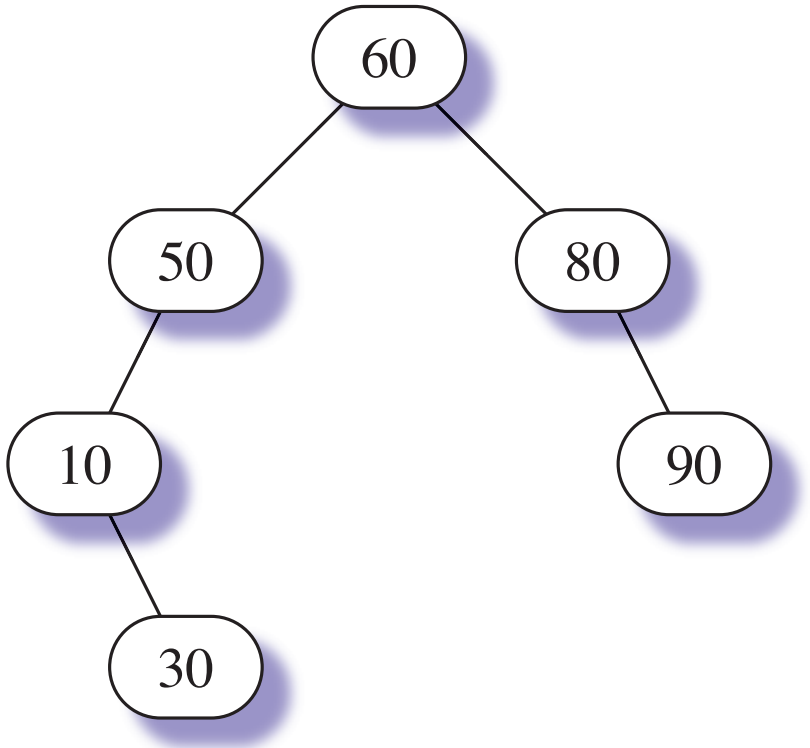
- A binary search tree is a binary tree.
- Every data value in the left subtree is less than the data value of the root.
- Every data value in the right subtree is greater than the data value of the root.
- The left subtree is a binary search tree.
- The right subtree is a binary search tree.



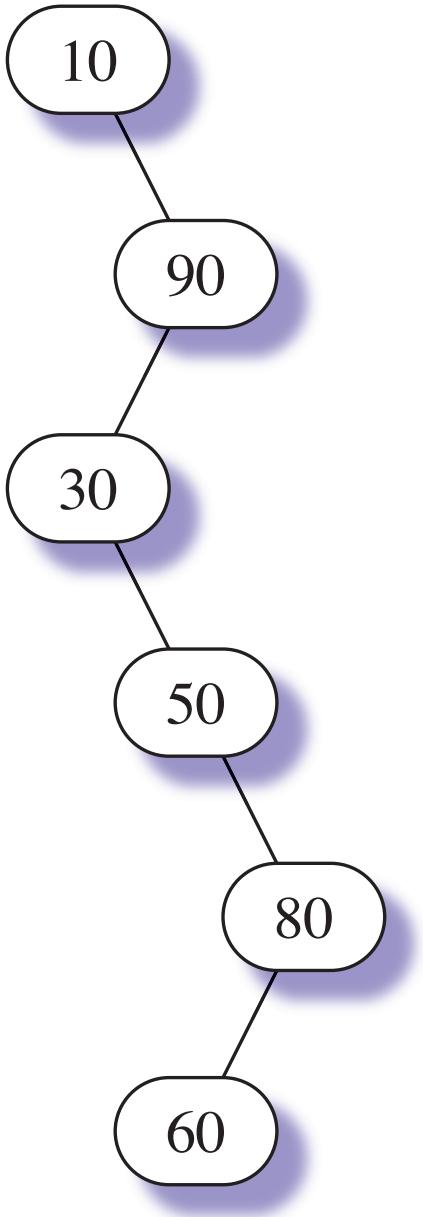
(a) A perfectly balanced search tree.



(a) A perfectly balanced search tree.



(b) A slightly unbalanced search tree.



(c) A degenerate search tree.

Demo DictionaryBST

```
// ========== DictionaryBST ==========
template <class K, class V>
class DictionaryBST : public ADictionary<K, V> {
private:
    BiTreeCSV< DictPair<K,V> > _tree;

public:
    DictionaryBST() = default;

    void clear() override {
        BiTCSVclearVis< DictPair<K,V> > clearVis;
        _tree.accept(clearVis);
    }
```

```
void insert(K const &key, V const &val) override {
    DictTinsertVis<K, V> dictTinsertVis(key, val);
    _tree.accept(dictTinsertVis);
}

bool remove(K const &key, V &val) override {
    DictTremoveVis<K, V> dictTremoveVis(key);
    _tree.accept(dictTremoveVis);
    return dictTremoveVis.result(val);
}
```

```
    bool contains(K const &key, V &val) const override {
        DictTcontainsVis<K, V> dictTcontainsVis(key);
        _tree.accept(dictTcontainsVis);
        return dictTcontainsVis.result(val);
    }

    void toStream(ostream &os) const override {
        BiTCSVtoStreamVerticalVis< DictPair<K, V> >
            dictTtoStreamVis(os);
        _tree.accept(dictTtoStreamVis);
    }
};
```

```
int main() {
    DictionaryBST<int, string> treeDict;
    int key;
    string value;
    char response;
    do {
        cout << "\n(c)lear  (i)nsert  (r)emove  co(n)tains ...
        cin >> response;
        switch (toupper(response)) {
        case 'C':
            treeDict.clear();
            break;
        case 'I':
            cout << "Insert what integer key? ";
            cin >> key;
            cout << "Insert what string value? ";
            cin >> value;
            treeDict.insert(key, value);
            break;
```

value is called by reference

```
case 'N':
    cout << "Search for what integer key? ";
    cin >> key;
    if (treeDict.contains(key, value)) {
        cout << "\nThe value for key " << key
             << " is "<< value << "." << endl;
    }
    else {
        cout << "\nKey " << key
             << " is not in the dictionary." << endl;
    }
    break;
```

Algorithm for contains operation

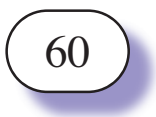
```
bool contains(key, val)
    if (tree is empty)
        return false
    else if (key < root value)
        return left subtree.contains(key, val)
    else if (key > root value)
        return right subtree.contains(key, val)
    else
        set val
        return true
```

```
// ========= DictTcontainsVis =========
template <class K, class V>
class DictTcontainsVis : public ABiTreeCSVVis< DictPair<K, V> > {
private:
    K const &_key; // Input parameter.
    V _val; // Output result.
    bool _found; // Output result.

public:
    // ========= Constructor =========
    DictTcontainsVis(K const &key):
        _key(key) {
    }
```

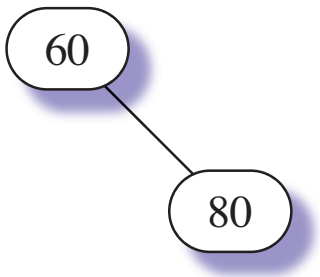
```
// ========= visit =========
void emptyCase(BiTreeCSV< DictPair<K, V> > &host) override {
    cerr << "DictTcontainsVis: Exercise for the student." << endl;
    throw -1;
}

void nonEmptyCase(BiTreeCSV< DictPair<K, V> > &host) override {
    cerr << "DictTcontainsVis: Exercise for the student." << endl;
    throw -1;
}
```

```
// ========= result =========
// Pre: This visitor has been accepted by a host tree.
// Post: If key is found, then val is the associated value,
// and true is returned; otherwise false is returned.
bool result(V &val) const {
    if (_found) {
        cerr << "DictTcontainsVis: Exercise for the student." << endl;
        throw -1;
    }
}
```

(a) Insert 60.

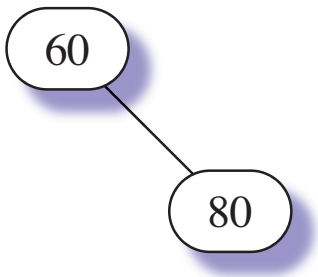


(a) Insert 60.

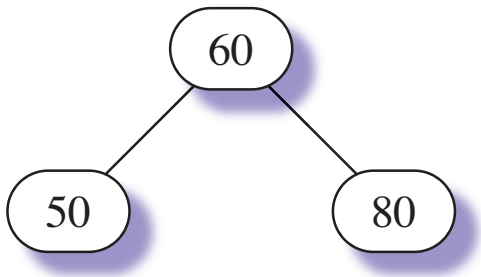


(b) Insert 80.

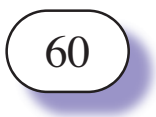
(a) Insert 60.



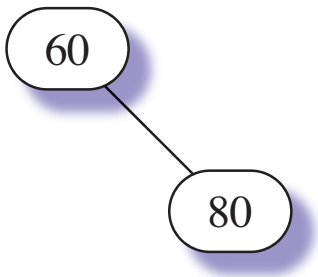
(b) Insert 80.



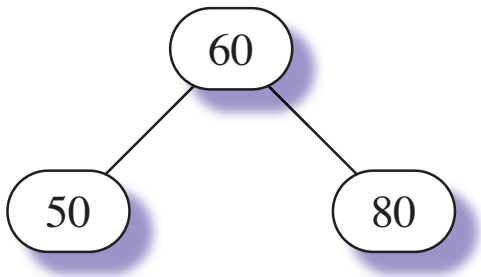
(c) Insert 50.



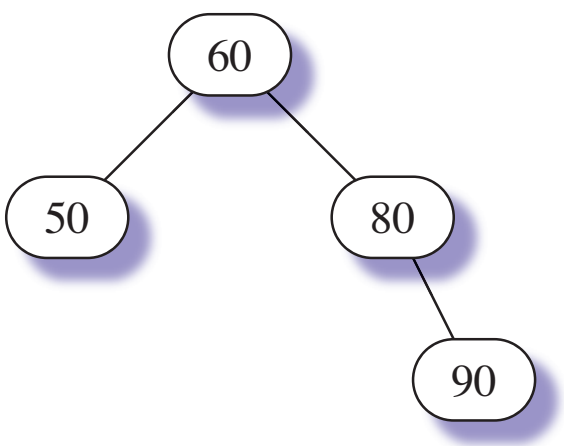
(a) Insert 60.



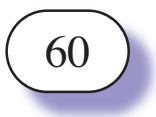
(b) Insert 80.



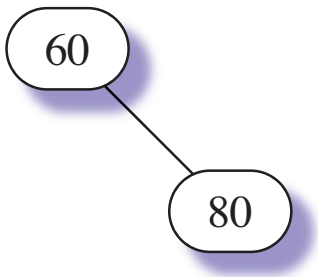
(c) Insert 50.



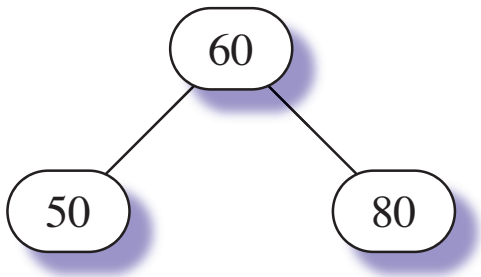
(d) Insert 90.



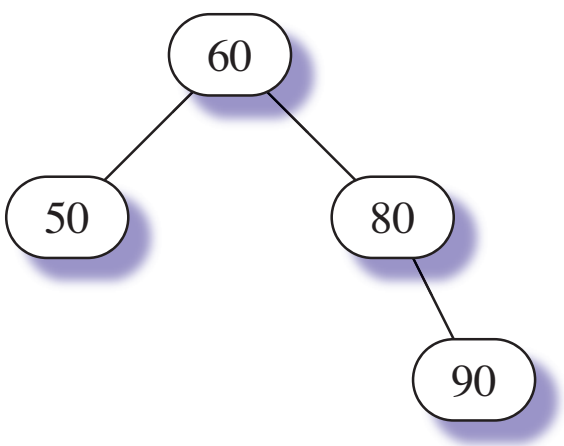
(a) Insert 60.



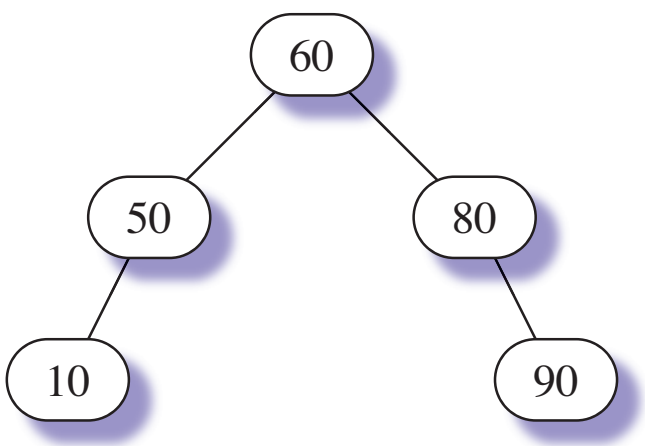
(b) Insert 80.



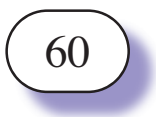
(c) Insert 50.



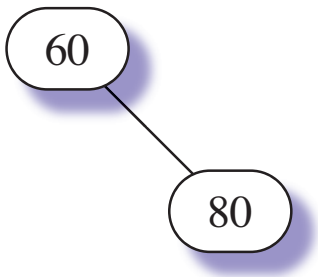
(d) Insert 90.



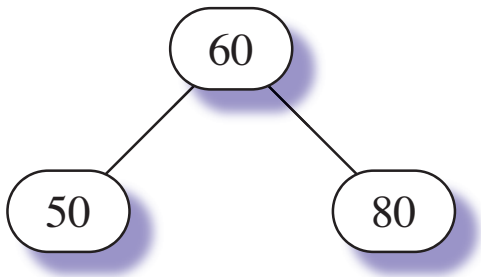
(e) Insert 10.



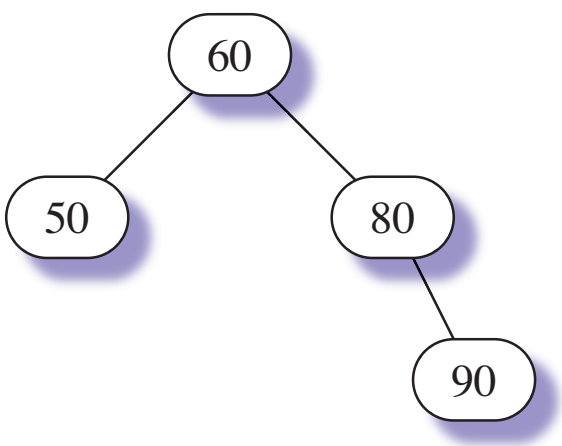
(a) Insert 60.



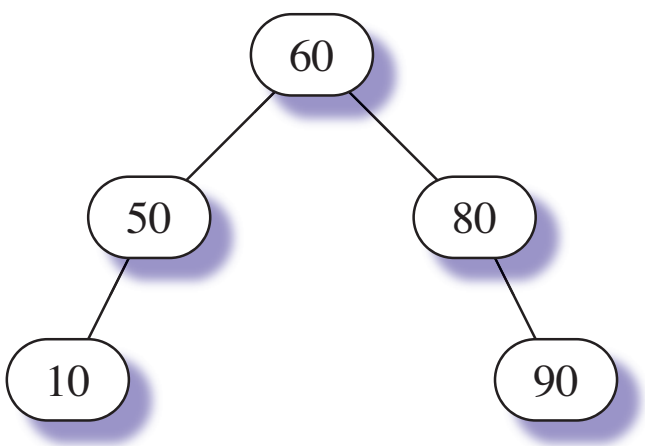
(b) Insert 80.



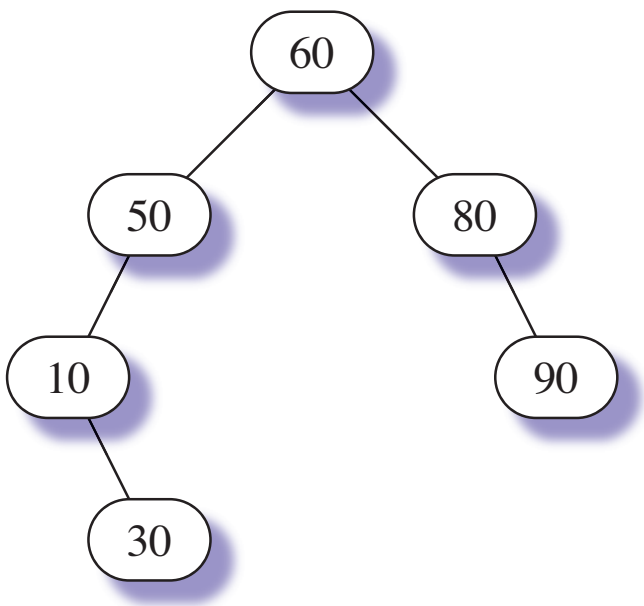
(c) Insert 50.



(d) Insert 90.



(e) Insert 10.



(f) Insert 30.

Algorithm for insertion operation

```
insert(key, val)
    if (tree is empty)
        insert root key val pair
    else if (key < root value)
        left subtree.insert(key, val)
    else if (key > root value)
        right subtree.insert(key, val)
    else // duplicate key, change val
        assign key val pair to this node
```

```
// ========= DictTinsertVis =========
template < class K, class V>
class DictTinsertVis: public ABiTreeCSVVis< DictPair<K, V> > {
private:
    K const &_key; // Input parameter.
    V const &_val; // Input parameter.

public:
    // ========= Constructor =========
    DictTinsertVis(K const &key, V const &val):
        _key(key),
        _val(val) {
    }
```

```
// ========= visit =========
// Pre: This visitor has been accepted by a host tree.
// Post: The host dictionary contains key and its associated
// value, val.
void emptyCase(BiTreeCSV< DictPair<K, V> > &host) override {
    cerr << "DictTinsertVis: Exercise for the student." << endl;
    throw -1;
}

void nonEmptyCase(BiTreeCSV< DictPair<K, V> > &host) override {
    cerr << "DictTinsertVis: Exercise for the student." << endl;
    throw -1;
}
```

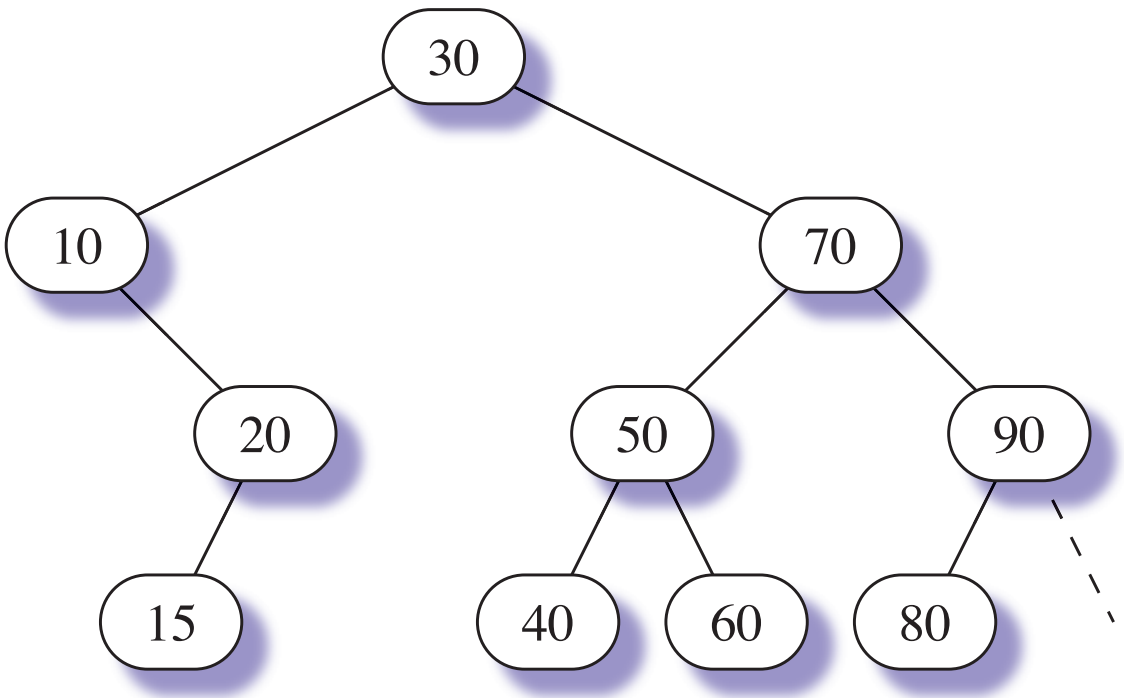
```
// ========= visit const =========
void emptyCase(BiTreeCSV< DictPair<K, V> > const &host) override {
    cerr << "DictTinsertVis precondition violated: "
         << "Cannot insert into a const tree." << endl;
    throw -1;
}

void nonEmptyCase(BiTreeCSV< DictPair<K, V> > const &host) override {
    cerr << "DictTinsertVis precondition violated: "
         << "Cannot insert into a const tree." << endl;
    throw -1;
}
```

Fact:

The maximum value of a search tree is in a subtree with an empty right child.

Fact:

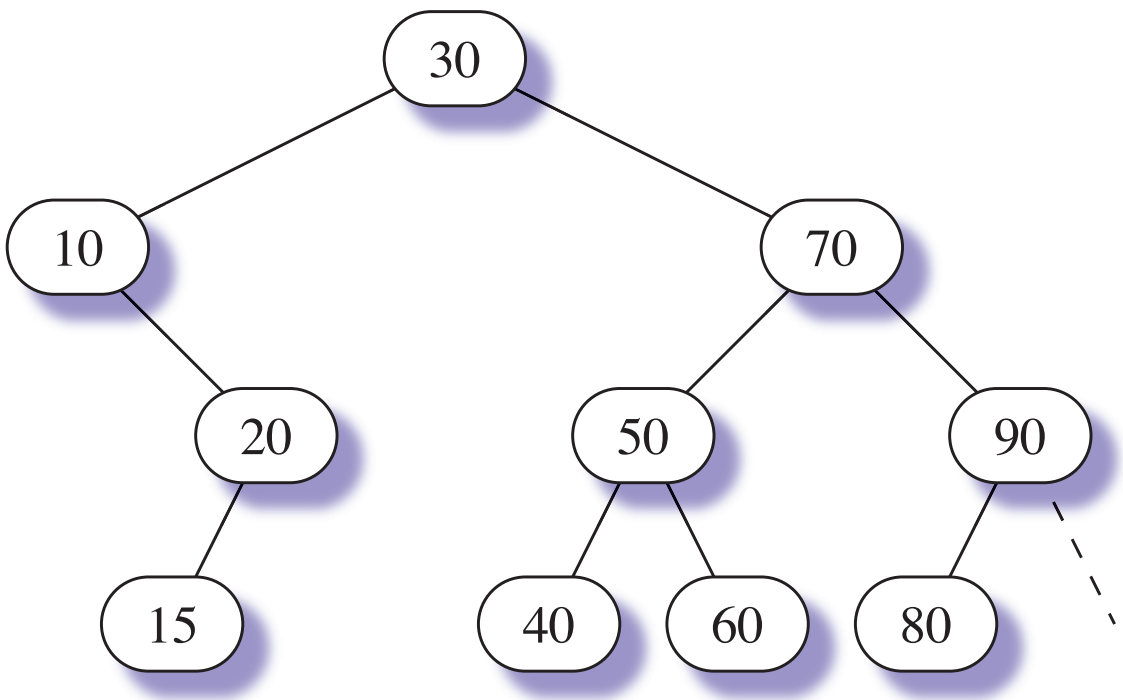
The maximum value of a search tree is in a subtree with an empty right child.



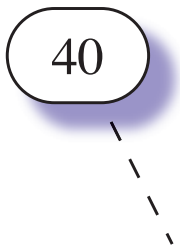
(a) Maximum 90 has no right child.

Fact:

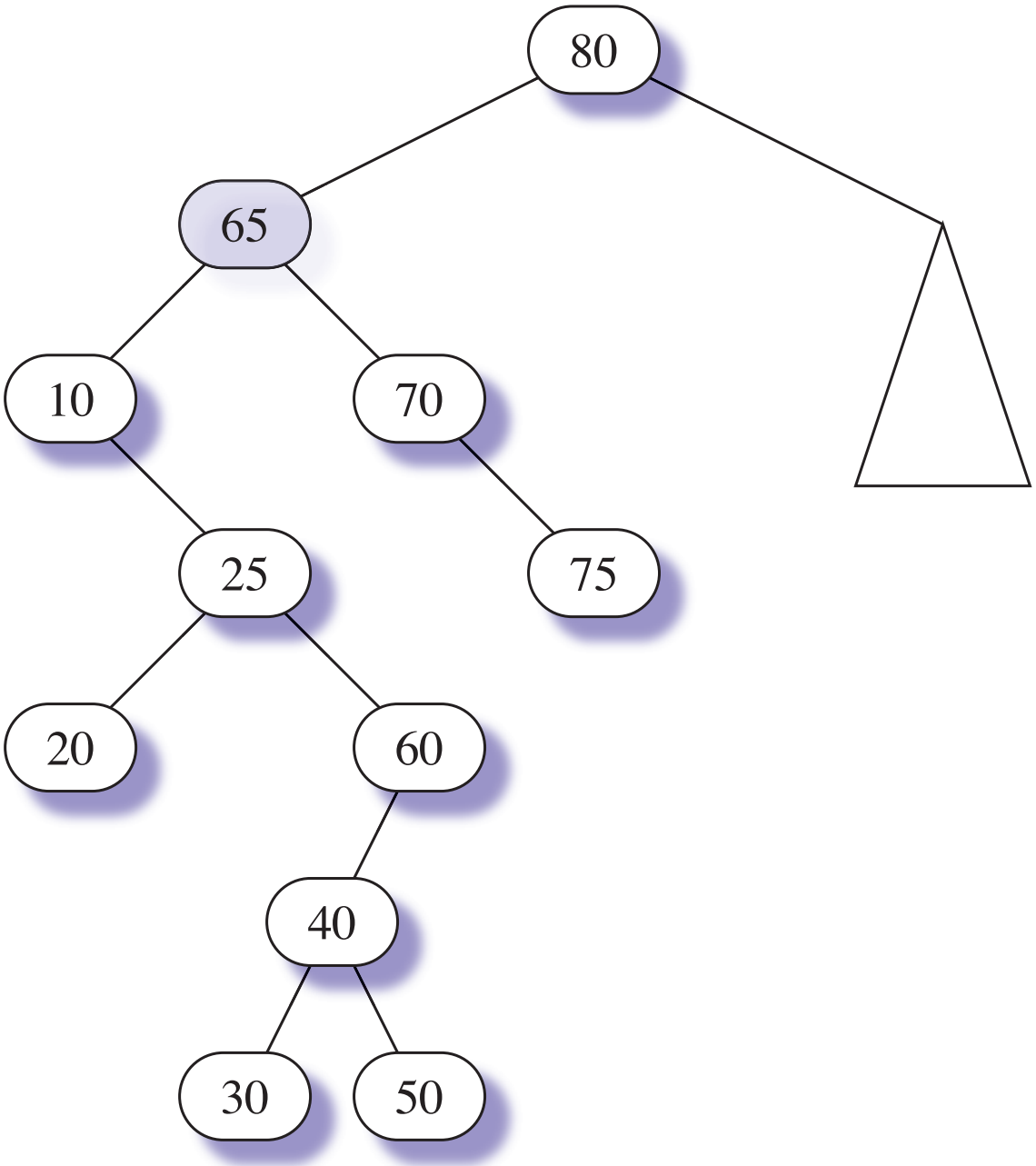
The maximum value of a search tree is in a subtree with an empty right child.



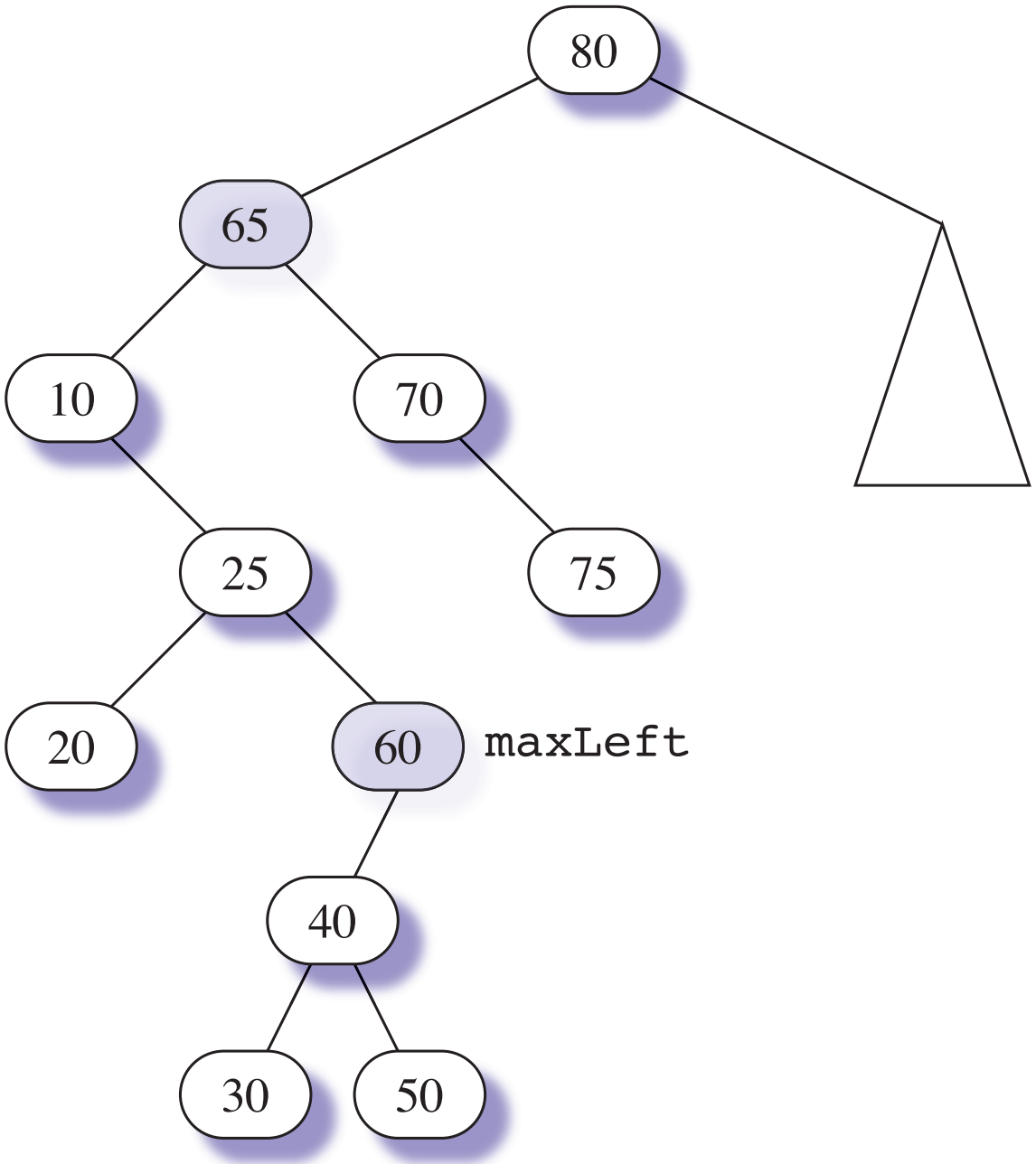
(a) Maximum 90 has no right child.



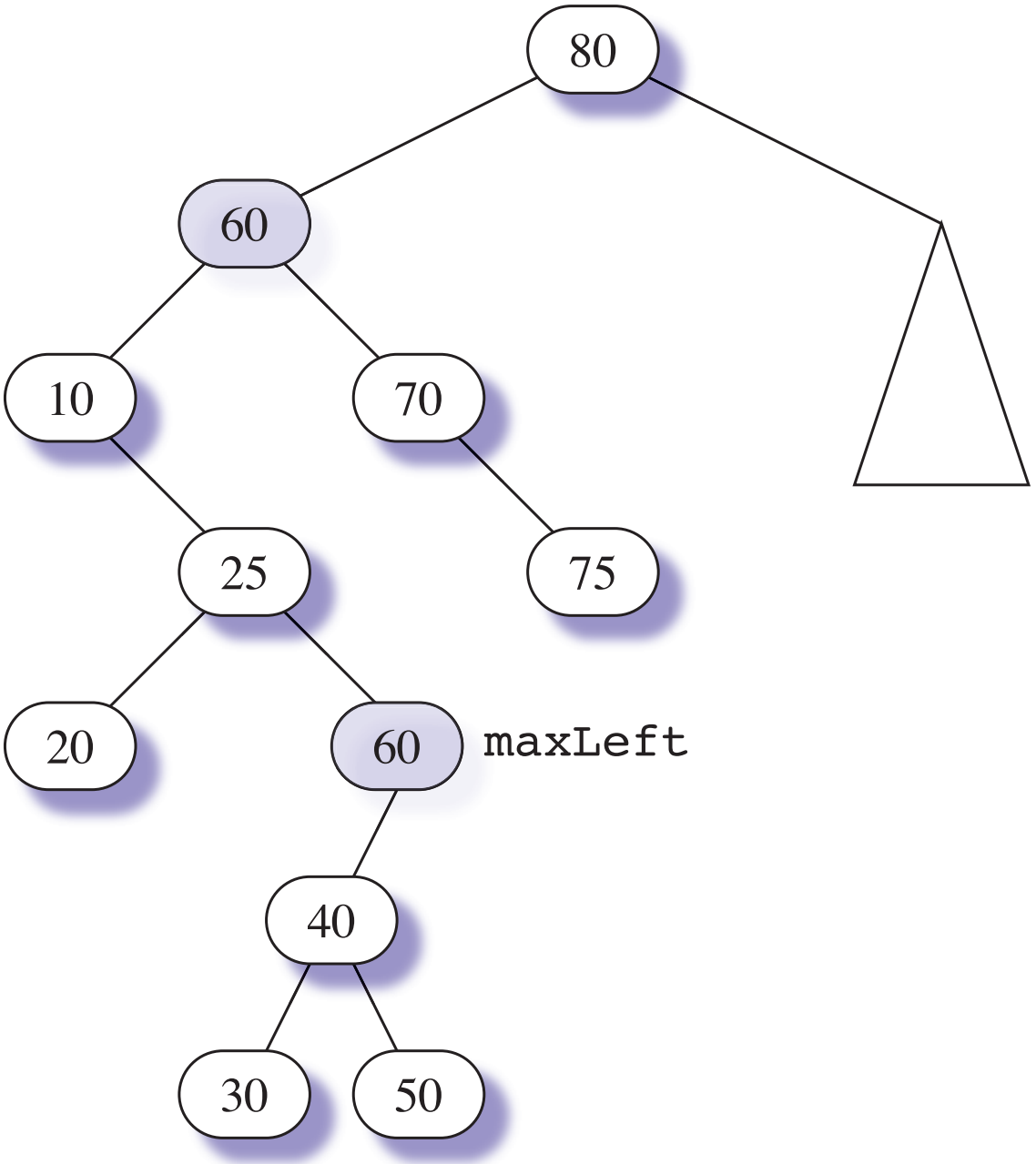
(b) Maximum 40 has no right child.



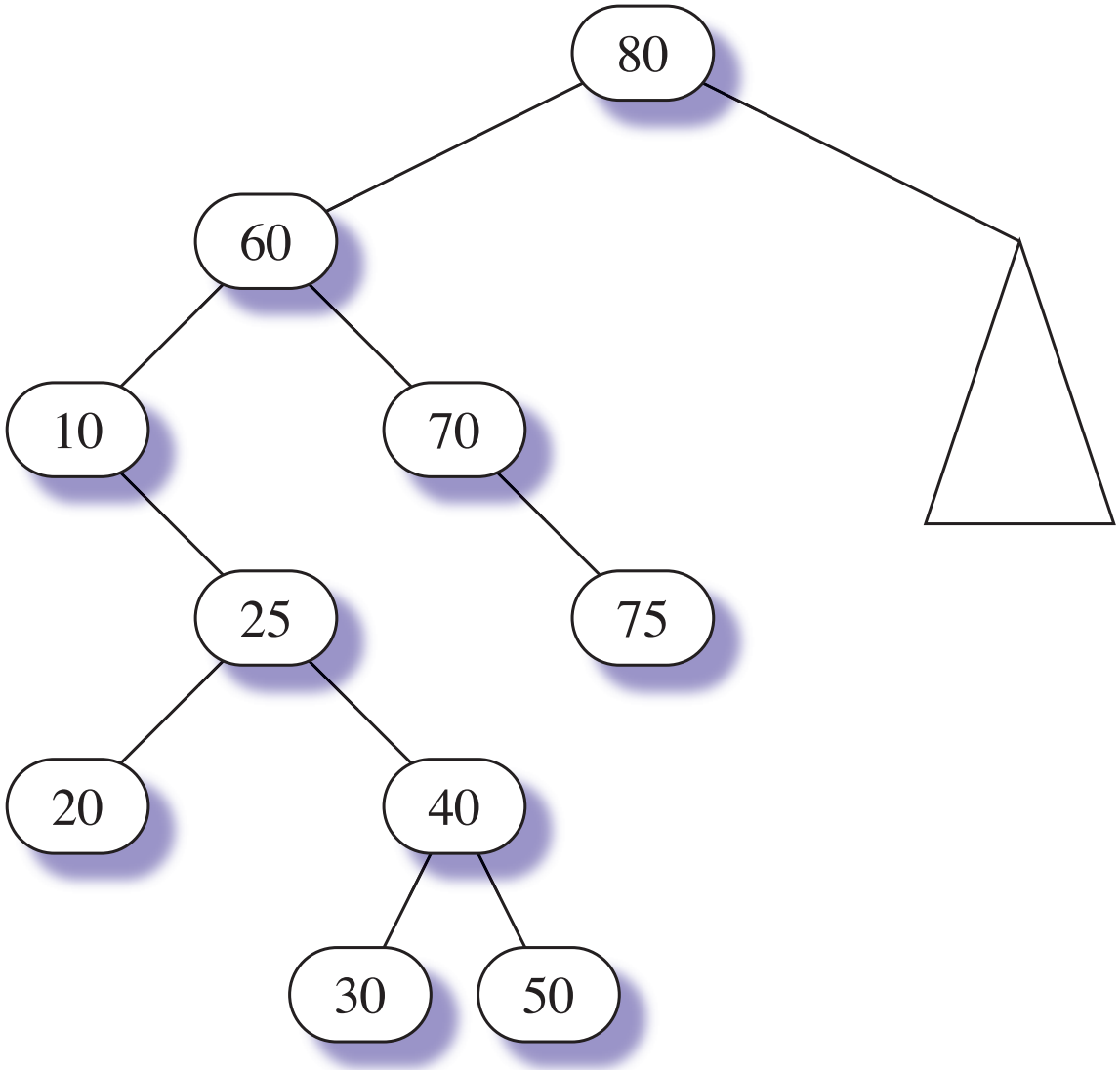
(a) Search for 65 to remove.



(b) Find the maximum of the left subtree of 65.



(c) Copy key value pair to the node to delete.



(d) Remove the root of the local maximum.

Algorithm for remove operation

```
bool remove(key, val)
    if (tree is empty)
        return false
    else if (key < root value)
        left subtree.remove(key, val)
    else if (key > root value)
        right subtree.remove(key, val)
    else
        set val
        if (left child is empty)
            remove this root
        else // find the maximum of the left subtree
            shared_ptr<BiTreeCSV<DictPair<K,V>>> maxLeft = host.left();
            while (the right child of maxLeft is not empty)
                advance maxLeft to its right child
            assign key val pair of maxLeft to node to delete
            remove this root
        return true
```

```
// ========= DictTremoveVis =========
template <class K, class V>
class DictTremoveVis : public ABiTreeCSVVis< DictPair<K, V> > {
private:
    K const &_key; // Input parameter.
    V _val; // Output result.
    bool _found; // Output result.

public:
    // ========= Constructor =========
    DictTremoveVis(K const &key):
        _key(key) {
    }
```

```
// ========= visit =========
void emptyCase(BiTreeCSV< DictPair<K, V> > &host) override {
    cerr << "DictTremoveVis: Exercise for the student." << endl;
    throw -1;
}

void nonEmptyCase(BiTreeCSV< DictPair<K, V> > &host) override {
    cerr << "DictTremoveVis: Exercise for the student." << endl;
    throw -1;
}
```

```
// ========= result =========
// Pre: This visitor has been accepted by a host tree.
// Post: If key is found, then val is the associated value,
// and true is returned; otherwise false is returned.
// Post: The host dictionary does not contain key and its
// associated value.

bool result(V &val) const {
    if (_found) {
        cerr << "DictTremoveVis: Exercise for the student." << endl;
        throw -1;
    }
}
```