General *n*-Way Trees

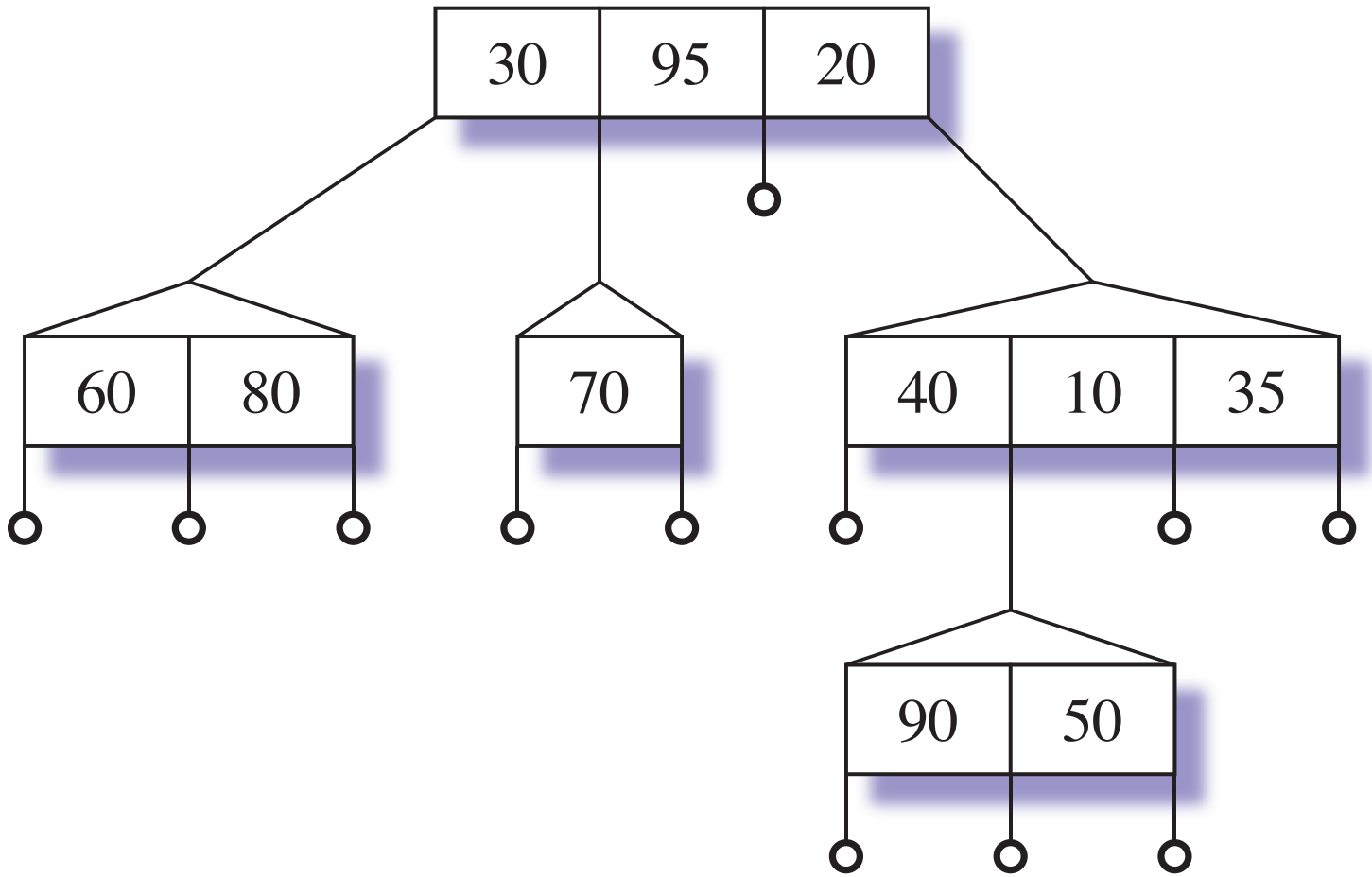
Definition of an *n*-way tree

The empty tree is an *n*-way tree.

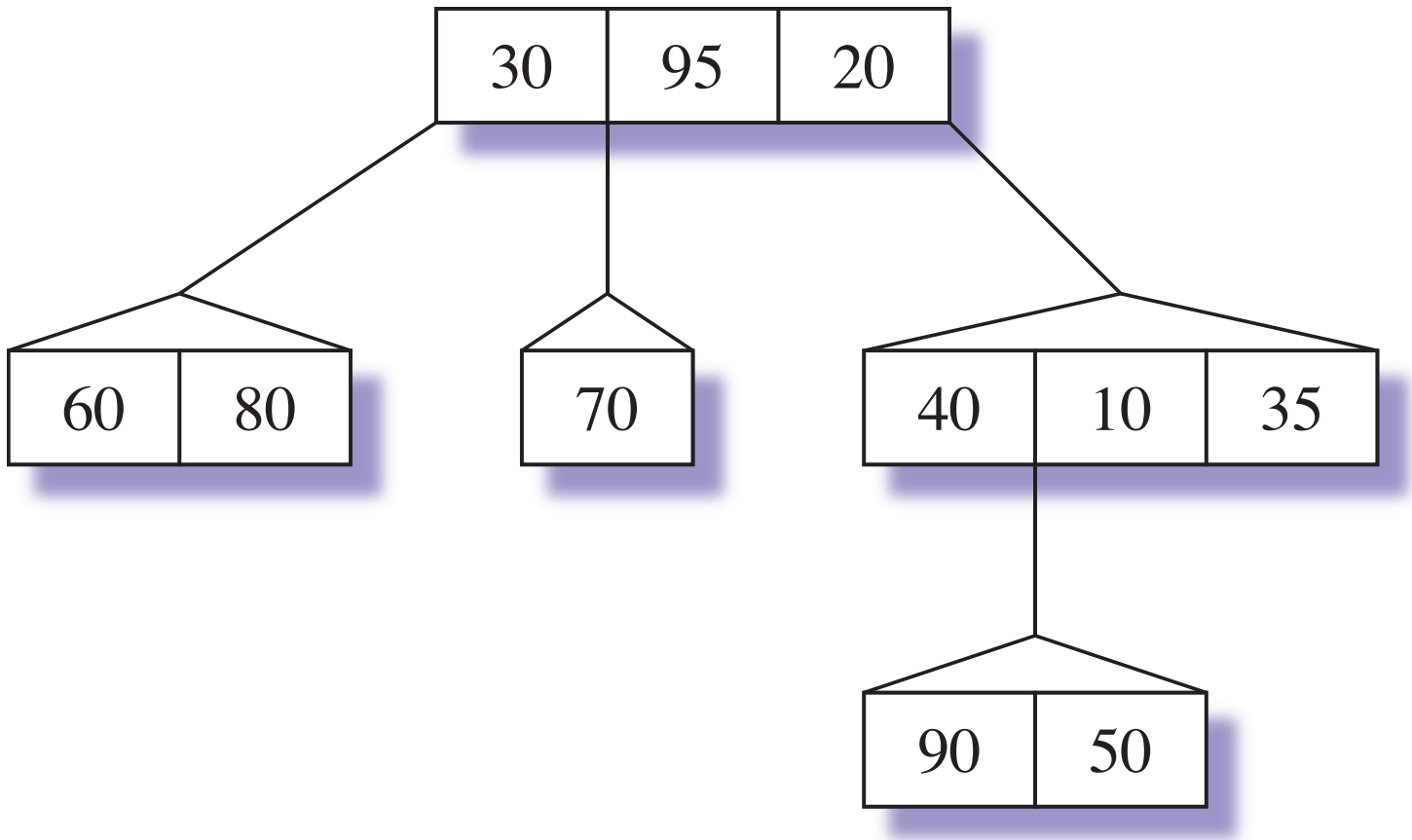
A nonempty *n*-way tree has
n data elements, and
n+1 children, each of which is an *n*-way tree.

Child *i* is the left child of data *i*
Child *i*+1 is the right child of data *i*

A general *n*-way tree



An abbreviated rendering of an *n*-way tree



An *n*-way tree contains

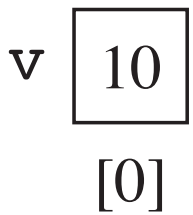
- a vector of data values (`VectorT`), and
- a vector of child trees (`VectorT`).

A vector has a variable size, and can shrink and grow.

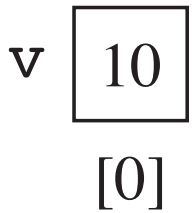
An empty tree has size zero for both vectors.

A vector class

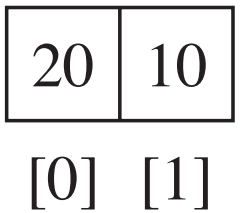
- Similar to an array, can subscript.
- Capacity increases automatically.
- Insert operation shifts current values to the right.
- Remove operation shifts current values to the left.



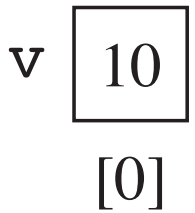
(a) `v.insert(0, 10);`



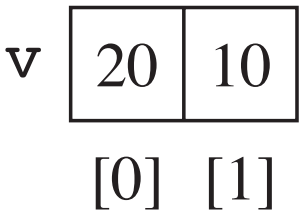
(a) `v.insert(0, 10);`



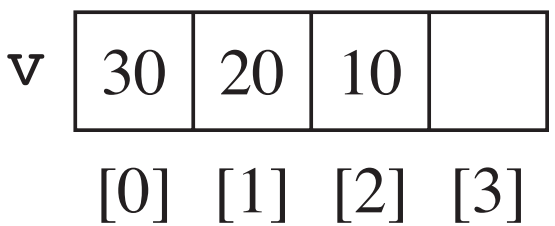
(b) `v.insert(0, 20);`
Capacity is doubled.



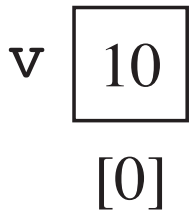
(a) `v.insert(0, 10);`



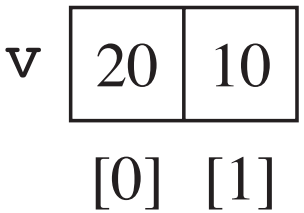
(b) `v.insert(0, 20);`
Capacity is doubled.



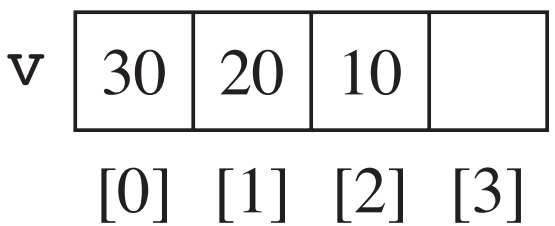
(c) `v.insert(0, 30);`
Capacity is doubled.



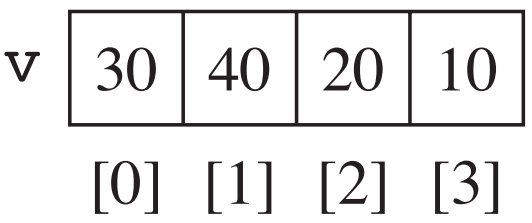
(a) `v.insert(0, 10);`



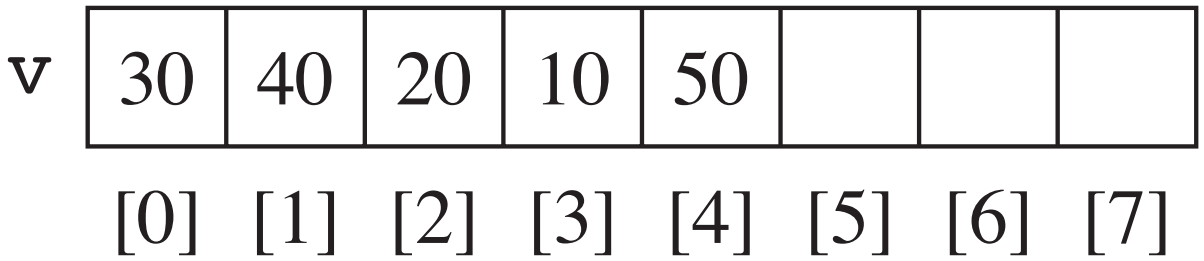
(b) `v.insert(0, 20);`
Capacity is doubled.



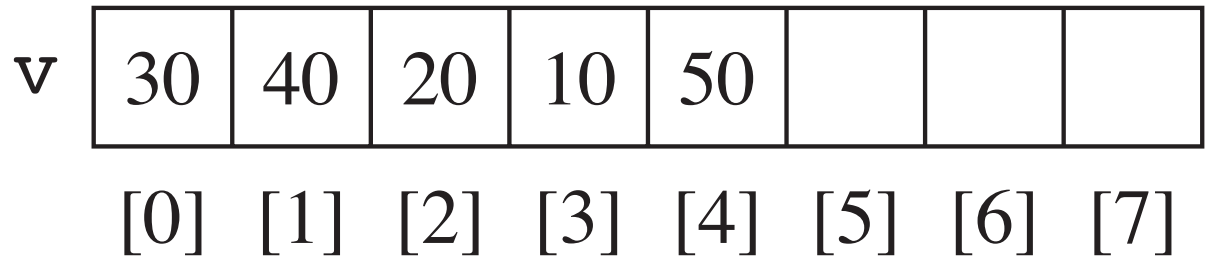
(c) `v.insert(0, 30);`
Capacity is doubled.



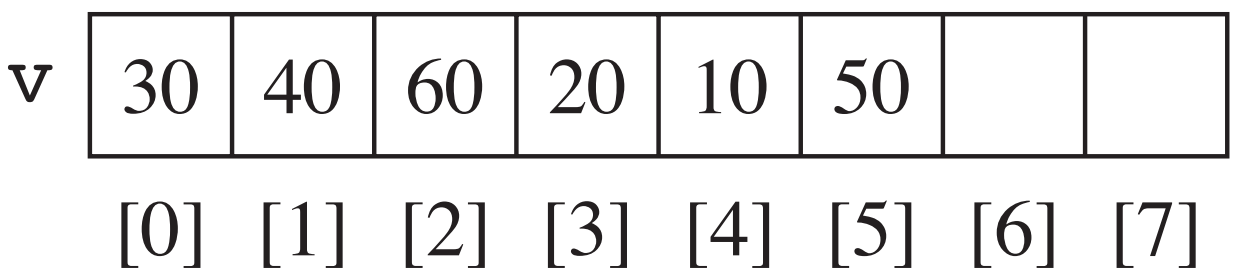
(d) `v.insert(1, 40);`



(e) `v.insert(4, 50);`
Capacity is doubled.



(e) `v.insert(4, 50);`
Capacity is doubled.



(f) `v.insert(2, 60);`

A vector implementation

```
template<class T>
// ========= VectorT =========
template<class T>
class VectorT : public ASeq<T> {
private:
    unique_ptr<T[]> _data;
    int _cap;  // Invariant: 0 < _cap, and _cap is a power of 2.
    int _size; // Invariant: 0 <= _size <= _cap.

    void doubleCapacity();
```

A vector implementation

```
public:
    VectorT();
    // Post: This vector is initialized with capacity of 1 and size of 0.

    int cap() const override { return _cap; }
    // Post: The capacity of this vector is returned.

    int size() const { return _size; }
    // Post: The size of this vector is returned.
```

(a) Specification of the constructor. Specification and implementation of `cap()` and `size()`.

```
// ========= Constructor =========
template<class T>
VectorT<T>::VectorT() {
    _data = make_unique<T[]>(1);
    _cap = 1;
    _size = 0;
}
```

(b) Implementation of the constructor.

A vector implementation

```
public:
    void append(T const &e);
    // Post: Element e is appended to this vector, possibly increasing cap().

    void insert(int i, T const &e);
    // Pre: 0 <= i && i <= size().
    // Post: Items [i..size()-1] are shifted right and element e is
    // inserted at position i.
    // size() is increased by 1, possibly increasing cap().

    T remove(int i);
    // Pre: 0 <= i && i < size(). T has a copy constructor.
    // Post: Element e is removed from position i and returned.
    // Items [i+1..size()-1] are shifted left.
    // size() is decreased by 1 (and cap() is unchanged).
```

(a) Specification of `append()`, `insert()`, and `remove()`.

The NTree implementation

```
template<class T>
class NTree {
private:
    shared_ptr<VectorT<T>> _data;
    shared_ptr<VectorT<shared_ptr<NTree<T>>>> _children;

private: // Constructors
    NTree(NTree<T> const &rhs);
    // Copy constructor disabled.

    NTree(shared_ptr<NTree<T>> left,
          T const &val,
          shared_ptr<NTree<T>> right);
    // Pre: left and right are not nullptr.
    // Post: this tree has one root element with left as the left subtree
    // and right as the right subtree.

    NTree(shared_ptr<VectorT<T>> data,
          shared_ptr<VectorT<shared_ptr<NTree<T>>>> children);
    // Pre: data and children are not nullptr.
    // Post: The data vector are the data
    // and the children vector are the children of this tree.
```

The NTree public constructors

```
public: // Constructors
    NTree();
    // Post: This tree is initialized to be empty.

    NTree(T const &val);
    // Post: This tree is initialized to have a single value val
    // and empty left and right children.
```

The NTree getters

```
T const &getData(int i) const;
// Post: The data value at position i is returned.

shared_ptr<NTree<T>> getChild(int i) const;
// Post: A pointer to the child tree at position i is returned.
```

The NTree spliceAt operator

```
void spliceAt(int i, NTree<T>& tree);
// Pre: 0 =< i <= _data->size().
// Post: tree is spliced into this tree at position i.
// Post: If tree is empty, this tree is unchanged.
// Post: tree is the empty tree.
```

The NTree splitUpAt operator

```
void splitUpAt(int i);
// Pre:  If _data->size() > 1,  0 <= i < _data->size().
// Post: If _data->size() > 1,  the element at position i is split up.
// Post: If _data->size() <= 1, nothing is done.
```

The NTree splitDownAt operator

```
void splitDownAt(int i);
// Pre:  If _data->size() != 0, 0 <= i < _data->size().
// Post: If _data->size() == 0, nothing is done.
// Post: If _data->size() != 1, the element at position i is split down.
// Post: If _data->size() == 1, the single value is deleted, and
//                              this tree is empty.
```

The NTree accept operator

```
void accept(ANTreeVis<T> &visitor);
void accept(ANTreeVis<T> &visitor) const;
```

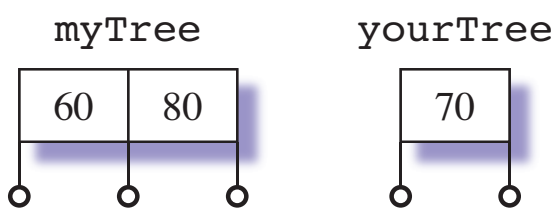
Demo NTree project

The NTree private constructors

```
// ========= Private constructors =========
template<class T>
NTree<T>::NTree(shared_ptr<NTree<T>> left,
                T const &val,
                shared_ptr<NTree<T>> right):
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
    _data->append(val);
    _children->append(left);
    _children->append(right);
}
```

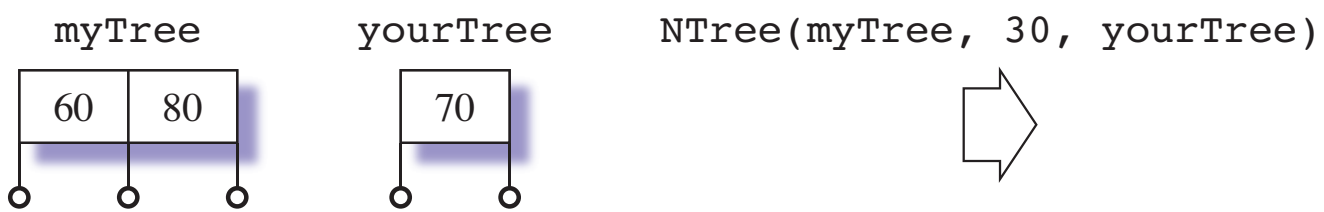
The NTree private constructors

```
// ========= Private constructors =========
template<class T>
NTree<T>::NTree(shared_ptr<NTree<T>> left,
                T const &val,
                shared_ptr<NTree<T>> right):
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
    _data->append(val);
    _children->append(left);
    _children->append(right);
}
```



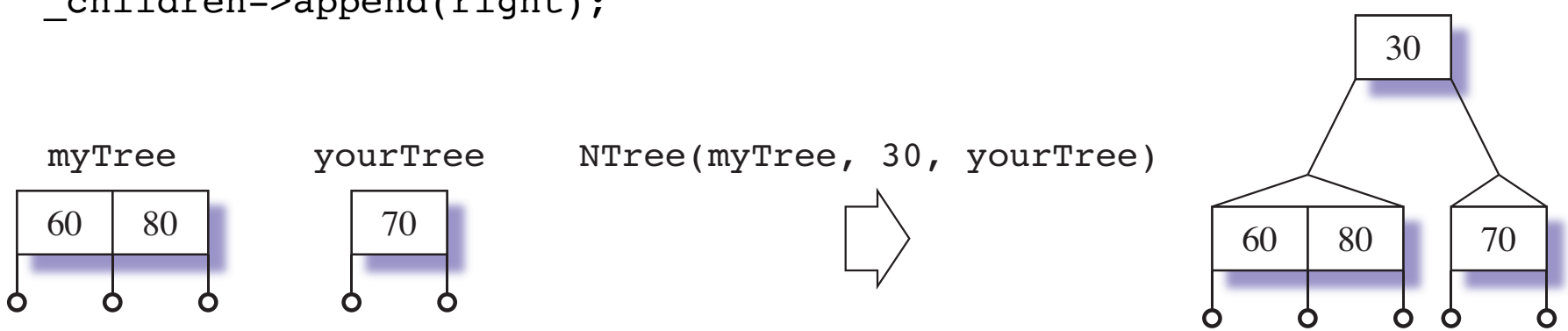
The NTree private constructors

```
// ========= Private constructors =========
template<class T>
NTree<T>::NTree(shared_ptr<NTree<T>> left,
                T const &val,
                shared_ptr<NTree<T>> right):
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
    _data->append(val);
    _children->append(left);
    _children->append(right);
}
```



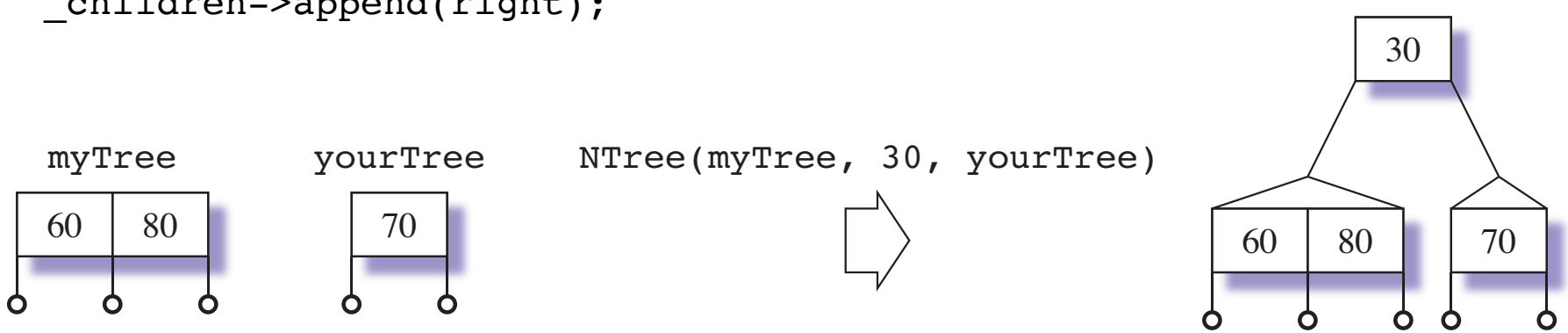
The NTree private constructors

```
// ========= Private constructors =========
template<class T>
NTree<T>::NTree(shared_ptr<NTree<T>> left,
                T const &val,
                shared_ptr<NTree<T>> right):
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
    _data->append(val);
    _children->append(left);
    _children->append(right);
}
```



The NTree private constructors

```
// ========= Private constructors =========
template<class T>
NTree<T>::NTree(shared_ptr<NTree<T>> left,
                T const &val,
                shared_ptr<NTree<T>> right):
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
    _data->append(val);
    _children->append(left);
    _children->append(right);
}
```




```
template<class T>
NTree<T>::NTree(shared_ptr<VectorT<T>> data,
                shared_ptr<VectorT<shared_ptr<NTree<T>>>> children) :
    _data(data), _children(children) {
}
```

The NTree public constructors

```
// ========= Public constructors =========
template<class T>
NTree<T>::NTree() :
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
}
```

The NTree public constructors

```
// ========= Public constructors =========
template<class T>
NTree<T>::NTree() :
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
}
```

NTree()

The NTree public constructors

```
// ========= Public constructors =========
template<class T>
NTree<T>::NTree() :
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
}
```

NTree()

The NTree public constructors

```
// ========= Public constructors =========
template<class T>
NTree<T>::NTree() :
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
}
```

NTree()

```
template<class T>
NTree<T>::NTree(T const &val) :
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
    _data->append(val);
    _children->append(make_shared<NTree<T>>());
    _children->append(make_shared<NTree<T>>());
}
```

The NTree public constructors

```
// ========= Public constructors =========
template<class T>
NTree<T>::NTree() :
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
}
```

NTree()

```
template<class T>
NTree<T>::NTree(T const &val) :
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
    _data->append(val);
    _children->append(make_shared<NTree<T>>());
    _children->append(make_shared<NTree<T>>());
}
```

NTree(30)

The NTree public constructors

```
// ========= Public constructors =========
template<class T>
NTree<T>::NTree() :
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
}
```

NTree()

```
template<class T>
NTree<T>::NTree(T const &val) :
    _data(new VectorT<T>), _children(new VectorT<shared_ptr<NTree<T>>>) {
    _data->append(val);
    _children->append(make_shared<NTree<T>>());
    _children->append(make_shared<NTree<T>>());
}
```

NTree(30)

The NTree getters

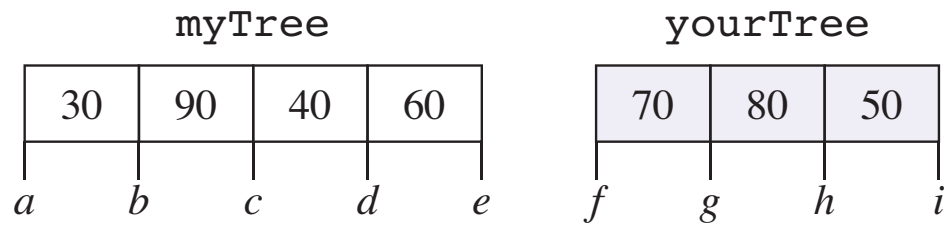
```
// ========== Getters ==========
template<class T>
T const &NTree<T>::getData(int i) const {
    return (*_data)[i];
}

template<class T>
shared_ptr<NTree<T>> NTree<T>::getChild(int i) const {
    return (*_children)[i];
}
```

The NTree splice operation

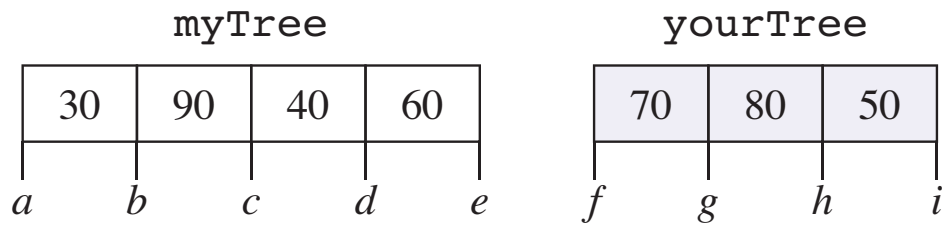
`myTree.spliceAt(2, yourTree)`

(a) Initial trees.

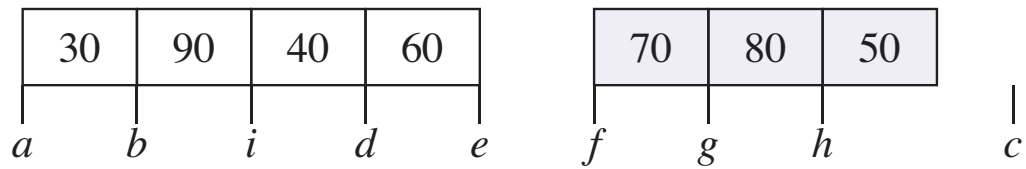


`myTree.spliceAt(2, yourTree)`

(a) Initial trees.

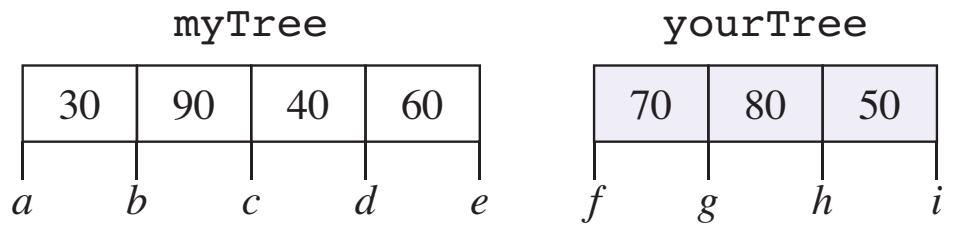


(b) Remove *i* from `yourTree`. Replace right child of 90 in `myTree` with it.

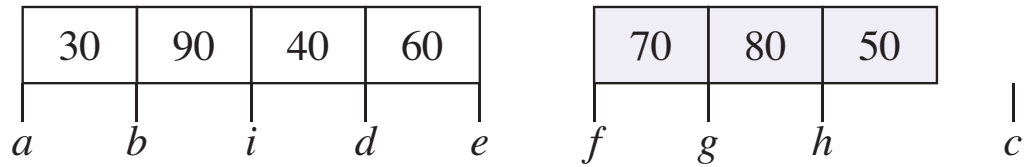


`myTree.spliceAt(2, yourTree)`

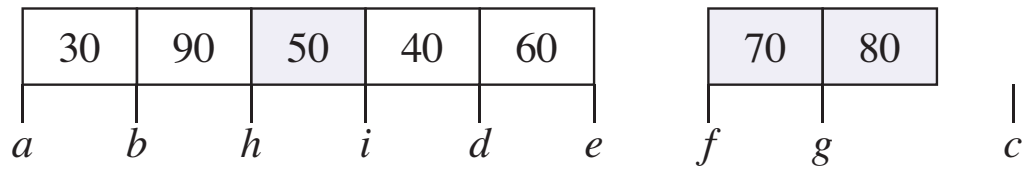
(a) Initial trees.



(b) Remove *i* from `yourTree`. Replace right child of 90 in `myTree` with it.

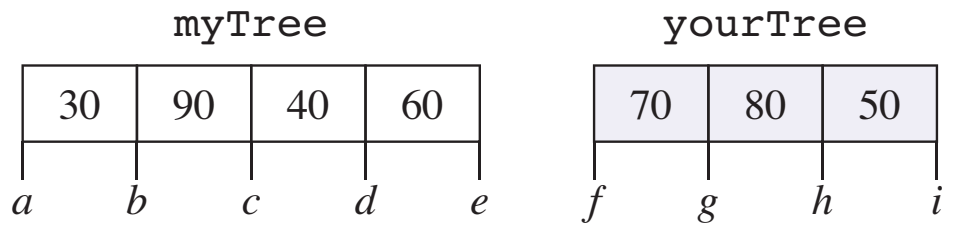


(c) Remove 50 and *h* from `yourTree`. Insert them after 90 in `myTree`.

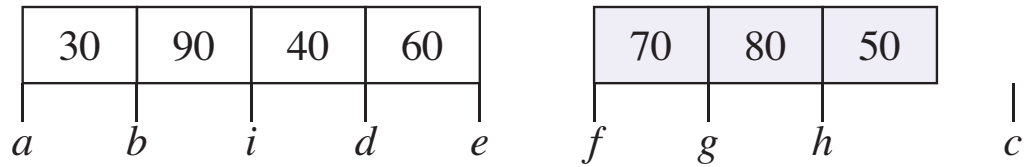


`myTree.spliceAt(2, yourTree)`

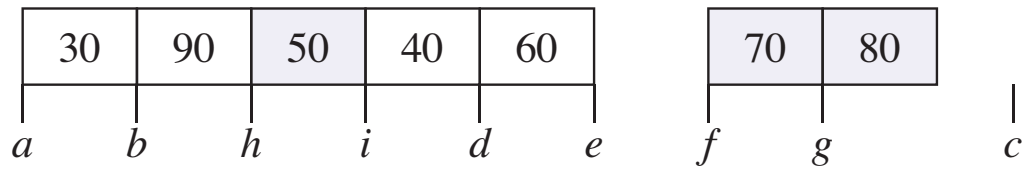
(a) Initial trees.



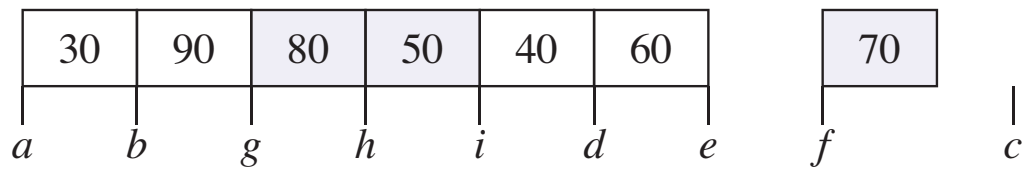
(b) Remove *i* from `yourTree`. Replace right child of 90 in `myTree` with it.



(c) Remove 50 and *h* from `yourTree`. Insert them after 90 in `myTree`.

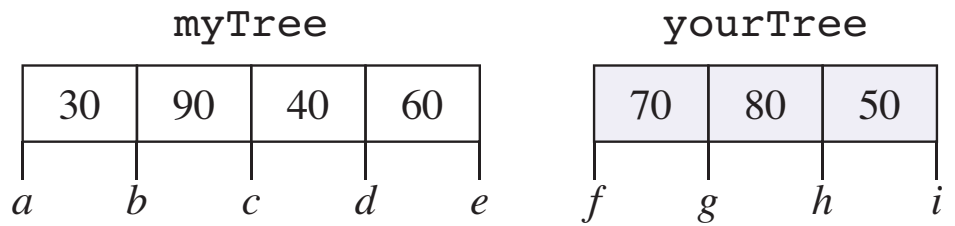


(d) Remove 80 and *g* from `yourTree`. Insert them after 90 in `myTree`.

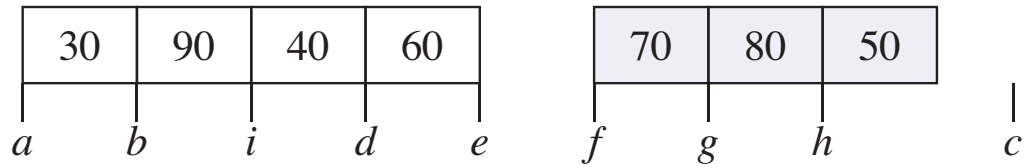


`myTree.spliceAt(2, yourTree)`

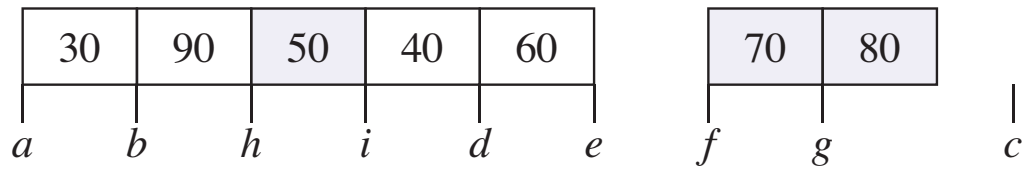
(a) Initial trees.



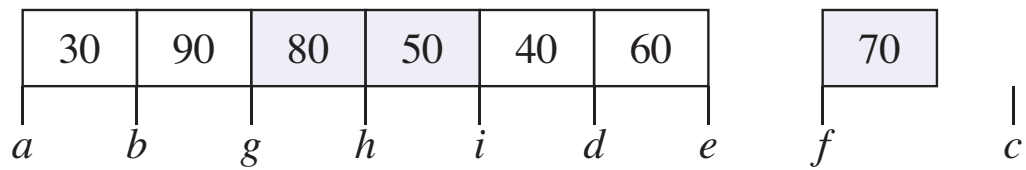
(b) Remove *i* from `yourTree`. Replace right child of 90 in `myTree` with it.



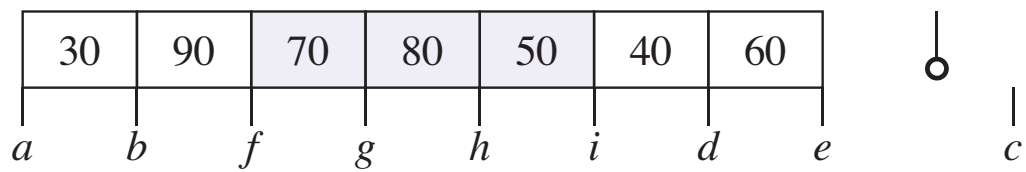
(c) Remove 50 and *h* from `yourTree`. Insert them after 90 in `myTree`.



(d) Remove 80 and *g* from `yourTree`. Insert them after 90 in `myTree`.



(e) Remove 70 and *f* from `yourTree`. Insert them after 90 in `myTree`. The disposition of *c* is the responsiblity of the caller.



Caller of `splice()`

Responsible for the leftover child pointer.

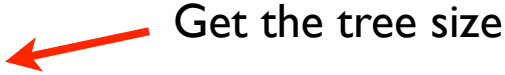
Two special cases:

- Receiver tree is empty.
  There is no leftover child pointer.

- Tree spliced into the receiver is empty.
  `splice()` does nothing.
  So, there is no leftover child pointer.

```
// ========= spliceAt =========
template<class T>
void NTree<T>::spliceAt(int i, NTree<T>& tree) {
    if (i < 0 || _data->size() < i) {
        cerr << "spliceAt precondition 0 =< i <= _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    int treeSize = tree._data->size();
    if (treeSize == 0) { // Tree being inserted is empty.
        return;
    }
    if (_data->size() == 0) { // Receiver tree is empty.
        _children->insert(i, tree._children->remove(treeSize--));
    } else {
        (*_children)[i] = tree._children->remove(treeSize--);
    }
    for (int k = treeSize; k >= 0; k--) {
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```

```
// ========= spliceAt =========
template<class T>
void NTree<T>::spliceAt(int i, NTree<T>& tree) {
    if (i < 0 || _data->size() < i) {
        cerr << "spliceAt precondition 0 =< i <= _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    int treeSize = tree._data->size();
    if (treeSize == 0) { // Tree being inserted is empty.
        return;
    }
    if (_data->size() == 0) { // Receiver tree is empty.
        _children->insert(i, tree._children->remove(treeSize--));
    } else {
        (*_children)[i] = tree._children->remove(treeSize--);
    }
    for (int k = treeSize; k >= 0; k--) {
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```



```
// ========= spliceAt =========
template<class T>
void NTree<T>::spliceAt(int i, NTree<T>& tree) {
    if (i < 0 || _data->size() < i) {
        cerr << "spliceAt precondition 0 =< i <= _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    int treeSize = tree._data->size();
    if (treeSize == 0) { // Tree being inserted is empty.
        return;
    }
    if (_data->size() == 0) { // Receiver tree is empty.
        _children->insert(i, tree._children->remove(treeSize--));
    } else {
        (*_children)[i] = tree._children->remove(treeSize--);
    }
    for (int k = treeSize; k >= 0; k--) {
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```

```
// ========= spliceAt =========
template<class T>
void NTree<T>::spliceAt(int i, NTree<T>& tree) {
    if (i < 0 || _data->size() < i) {
        cerr << "spliceAt precondition 0 =< i <= _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    int treeSize = tree._data->size();
    if (treeSize == 0) { // Tree being inserted is empty.
        return;
    }
    if (_data->size() == 0) { // Receiver tree is empty.
        _children->insert(i, tree._children->remove(treeSize--));
    } else {
        (*_children)[i] = tree._children->remove(treeSize--);
    }
    for (int k = treeSize; k >= 0; k--) {
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```

(b) Remove *i* from yourTree. Replace right child of 90 in myTree with it.

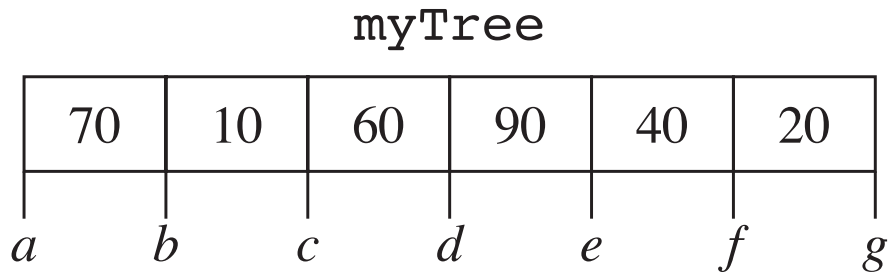
```
// ========= spliceAt =========
template<class T>
void NTree<T>::spliceAt(int i, NTree<T>& tree) {
    if (i < 0 || _data->size() < i) {
        cerr << "spliceAt precondition 0 =< i <= _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    int treeSize = tree._data->size();
    if (treeSize == 0) { // Tree being inserted is empty.
        return;
    }
    if (_data->size() == 0) { // Receiver tree is empty.
        _children->insert(i, tree._children->remove(treeSize--));
    } else {
        (*_children)[i] = tree._children->remove(treeSize--);
    }
    for (int k = treeSize; k >= 0; k--) {
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```

(c) Remove 50 and *h* from yourTree.
Insert them after 90 in myTree.

The NTree split down operation

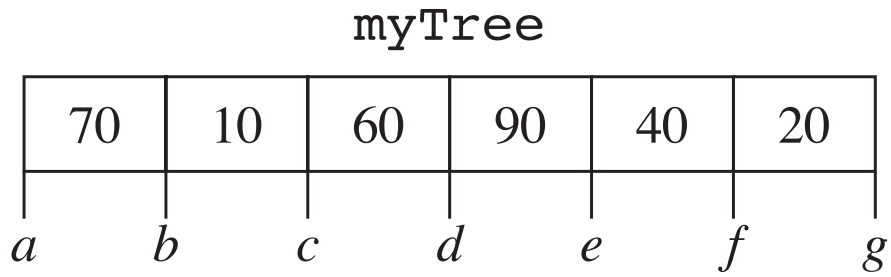
`myTree.splitDownAt(2)`

(a) Initial tree.

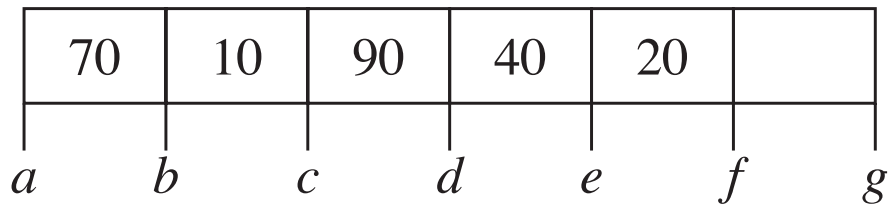


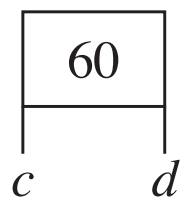
`myTree.splitDownAt(2)`

(a) Initial tree.



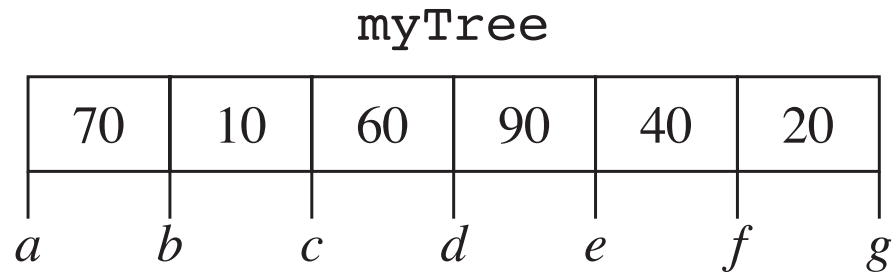
(b) Create a new single tree with 60 removed from `myTree`, and the same left and right children.



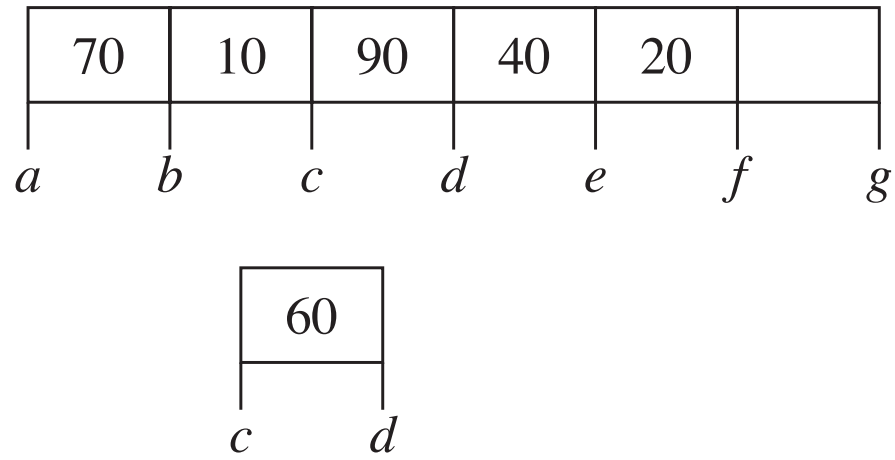


`myTree.splitDownAt(2)`

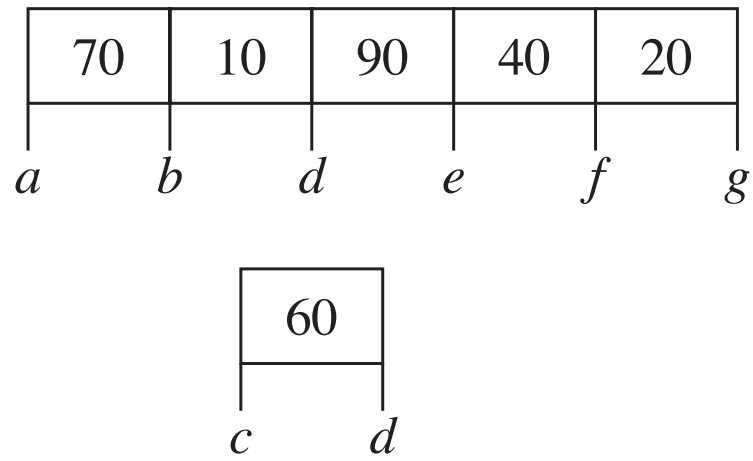
(a) Initial tree.



(b) Create a new single tree with 60 removed from `myTree`, and the same left and right children.

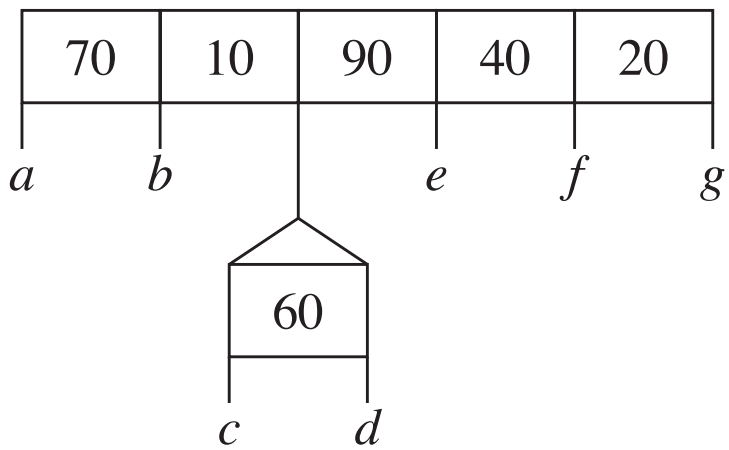


(c) Remove the old left child of 60.



`myTree.splitDownAt(2)`

(d) Set the old left child of 60 to the new single tree.



```
// ========= splitDownAt =========
template<class T>
void NTree<T>::splitDownAt(int i) {
    if (_data->size() == 0) {
        return;
    }
    if (i < 0 || _data->size() <= i) {
        cerr << "splitDownAt precondition 0 <= i < _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    if (_data->size() == 1) {
        _data->remove(0); // _data is now empty.
    } else {
        auto newChild = shared_ptr<NTree<T>>(new NTree<T>(getChild(i),
                                                          _data->remove(i),
                                                          getChild(i + 1)));
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```

```
// ========= splitDownAt =========
template<class T>
void NTree<T>::splitDownAt(int i) {
    if (_data->size() == 0) {
        return;
    }
    if (i < 0 || _data->size() <= i) {
        cerr << "splitDownAt precondition 0 <= i < _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    if (_data->size() == 1) {
        _data->remove(0); // _data is now empty.
    } else {
        auto newChild = shared_ptr<NTree<T>>(new NTree<T>(getChild(i),
                                                          _data->remove(i),
                                                          getChild(i + 1)));
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```

Special case.

```
// ========= splitDownAt =========
template<class T>
void NTree<T>::splitDownAt(int i) {
    if (_data->size() == 0) {
        return;
    }
    if (i < 0 || _data->size() <= i) {
        cerr << "splitDownAt precondition 0 <= i < _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    if (_data->size() == 1) {
        _data->remove(0); // _data is now empty.
    } else {
        auto newChild = shared_ptr<NTree<T>>(new NTree<T>(getChild(i),
                                                          _data->remove(i),
                                                          getChild(i + 1)));
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```

```
// ========= splitDownAt =========
template<class T>
void NTree<T>::splitDownAt(int i) {
    if (_data->size() == 0) {
        return;
    }
    if (i < 0 || _data->size() <= i) {
        cerr << "splitDownAt precondition 0 <= i < _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    if (_data->size() == 1) {
        _data->remove(0); // _data is now empty.
    } else {
        auto newChild = shared_ptr<NTree<T>>(new NTree<T>(getChild(i),
                                                          _data->remove(i),
                                                          getChild(i + 1)));

        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```

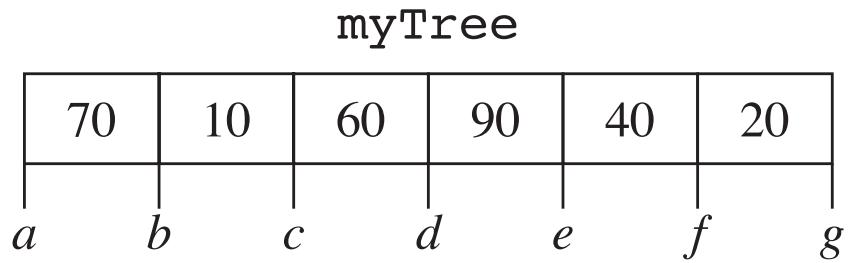
(b) Create a new single tree with 60 removed from myTree, and the same left and right children.

```
// ========= splitDownAt =========
template<class T>
void NTree<T>::splitDownAt(int i) {
    if (_data->size() == 0) {
        return;
    }
    if (i < 0 || _data->size() <= i) {
        cerr << "splitDownAt precondition 0 <= i < _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    if (_data->size() == 1) {
        _data->remove(0); // _data is now empty.
    } else {
        auto newChild = shared_ptr<NTree<T>>(new NTree<T>(getChild(i),
                                                          _data->remove(i),
                                                          getChild(i + 1)));
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
}
```

(c) Remove the old left child of 60.
(d) Set the old left child of 60 to the new single tree.

The NTree split up operation

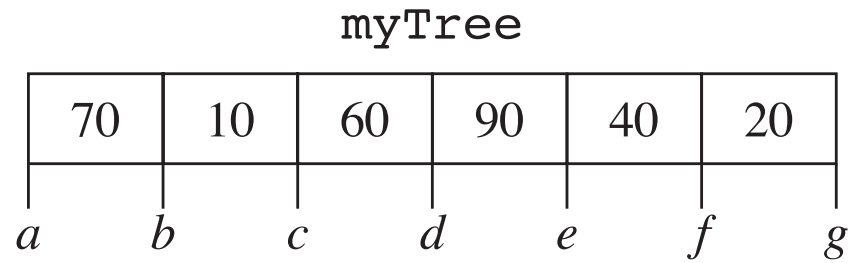
`myTree.splitUpAt(2)`



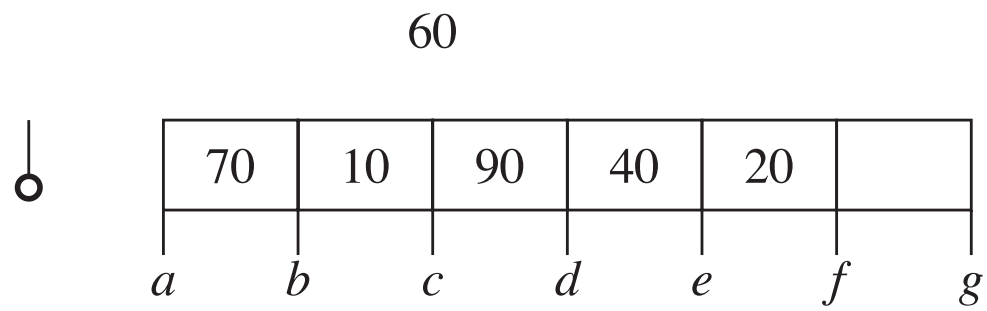
(a) Initial tree.

`myTree.splitUpAt(2)`

(a) Initial tree.

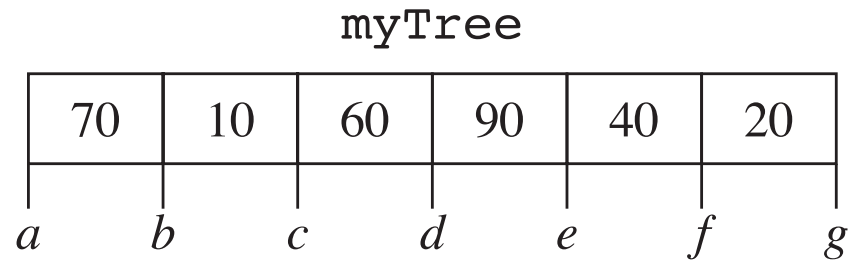


(b) Allocate new data and children vectors.
Remove and save 60.

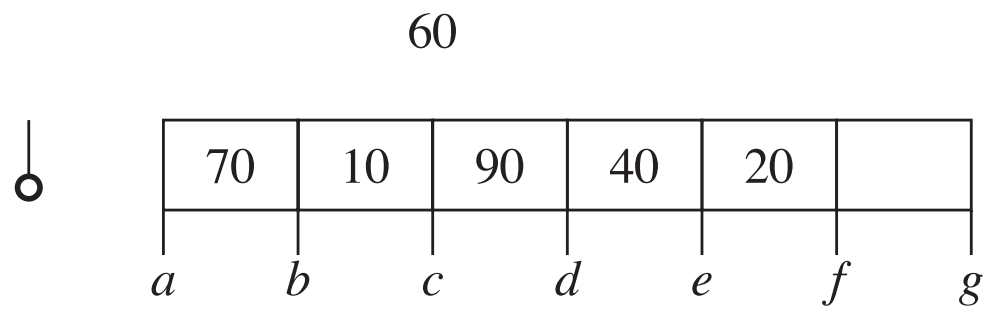


`myTree.splitUpAt(2)`

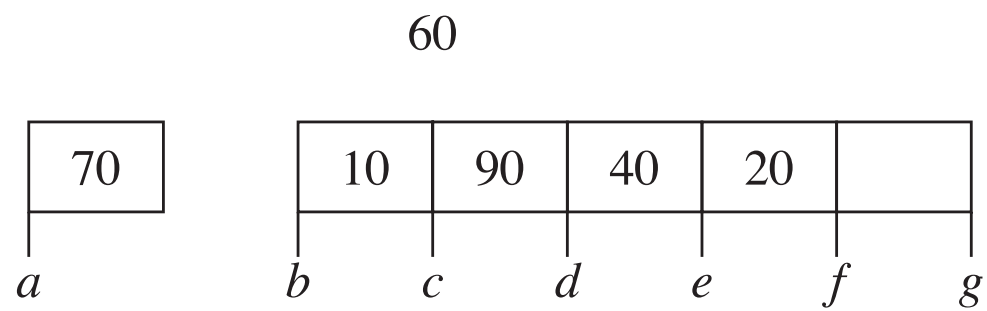
(a) Initial tree.



(b) Allocate new data and children vectors. Remove and save 60.

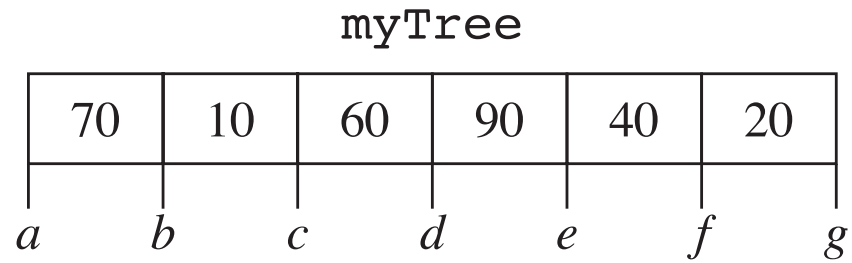


(c) In a loop, remove 70 and *a* and append to new data and new children.

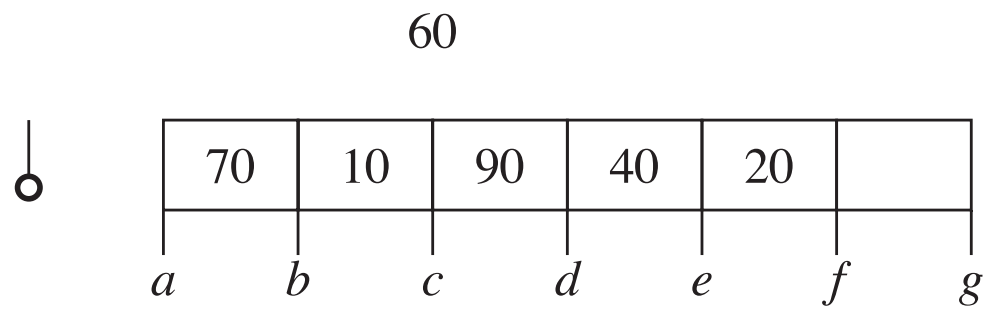


`myTree.splitUpAt(2)`

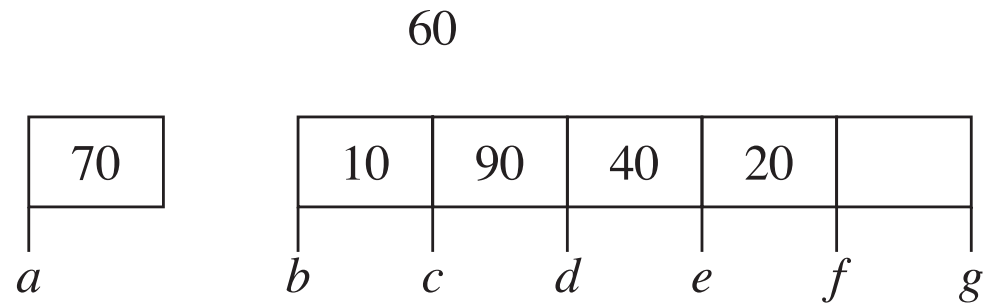
(a) Initial tree.



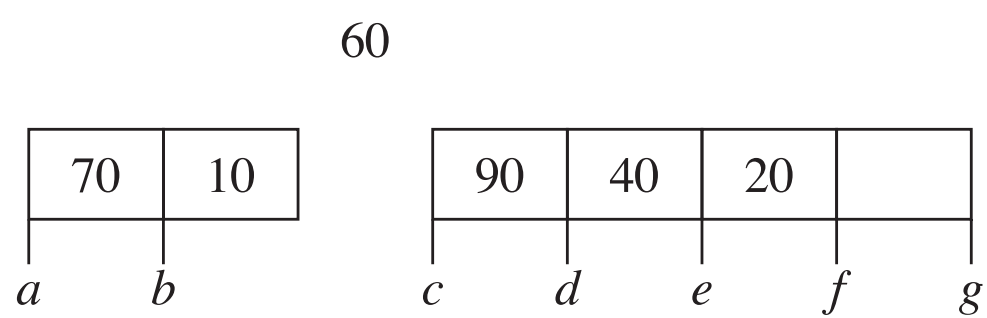
(b) Allocate new data and children vectors. Remove and save 60.



(c) In a loop, remove 70 and *a* and append to new data and new children.

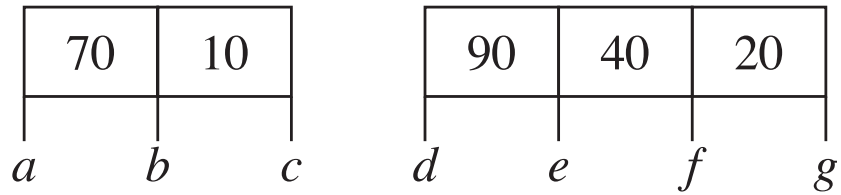


(d) In a loop, remove 10 and *b* and append to new data and new children.



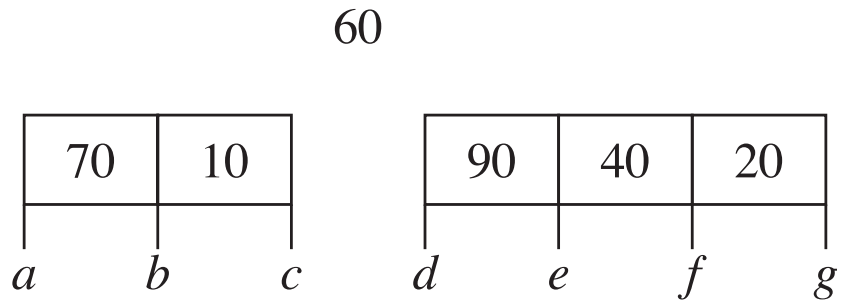
`myTree.splitUpAt(2)`

(e) Remove *c* and append to new children.

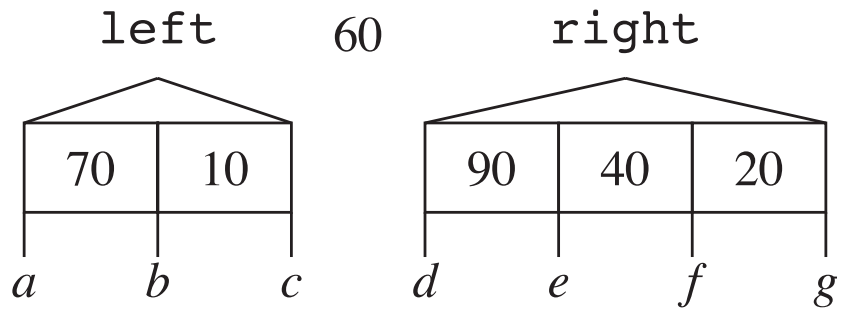


`myTree.splitUpAt(2)`

(e) Remove *c* and append to new children.

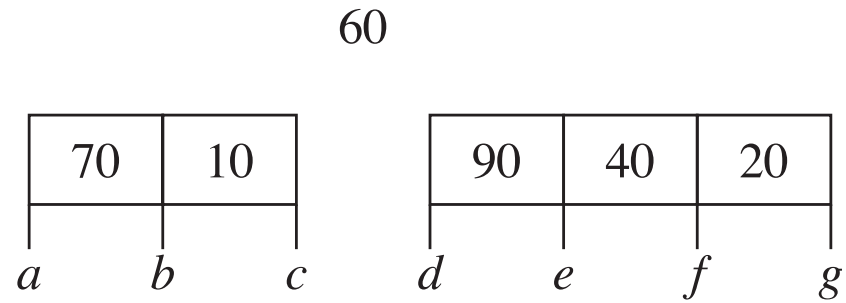


(f) Make a new tree for `left` with new data and new children, and a new tree for `right` with current data and current children.

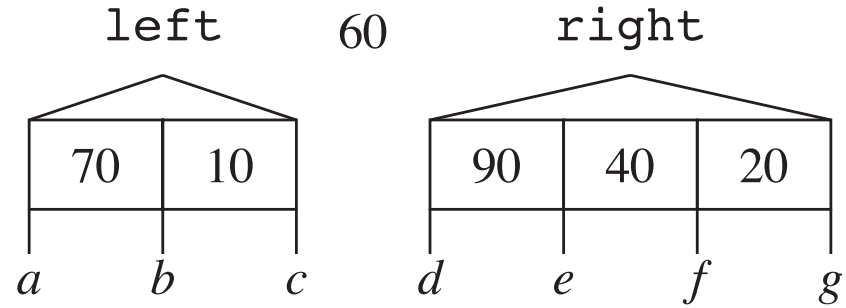


`myTree.splitUpAt(2)`

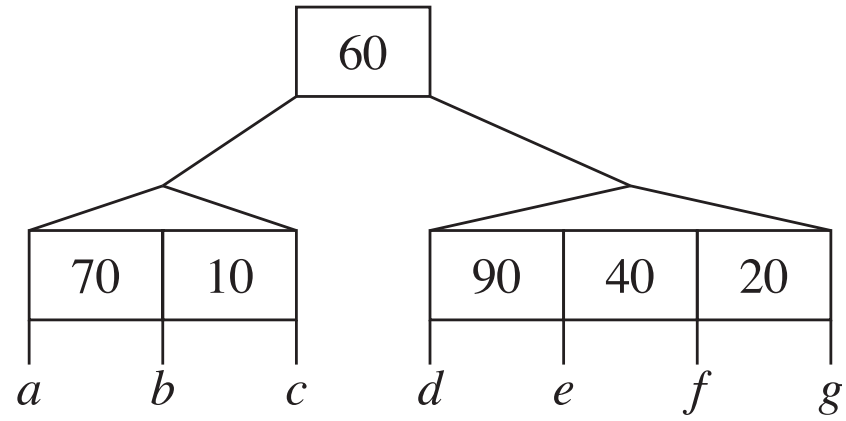
(e) Remove *c* and append to new children.



(f) Make a new tree for `left` with new data and new children, and a new tree for `right` with current data and current children.



(g) Allocate a new vector for `_data` and append 60. Allocate a new vector for `_children` and append `left` and then `right`.



```
// ========= splitUpAt =========
template<class T>
void NTree<T>::splitUpAt(int i) {
    if (_data->size() <= 1) {
        return;
    }
    if (i < 0 || _data->size() <= i) {
        cerr << "splitUpAt precondition 0 <= i < _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    // Build a new left subtree with new data and new children:
    auto newData = make_shared<VectorT<T>>();
    auto newChildren = make_shared<VectorT<shared_ptr<NTree<T>>>>();
    T rootDat = _data->remove(i); // This element will be at the new root.
    for (int k = 0; k < i; k++) {
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
```

```
// ========= splitUpAt =========
template<class T>
void NTree<T>::splitUpAt(int i) {
    if (_data->size() <= 1) {
        return;
    }
    if (i < 0 || _data->size() <= i) {
        cerr << "splitUpAt precondition 0 <= i < _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    // Build a new left subtree with new data and new children:
    auto newData = make_shared<VectorT<T>>();
    auto newChildren = make_shared<VectorT<shared_ptr<NTree<T>>>>();
    T rootDat = _data->remove(i); // This element will be at the new root.
    for (int k = 0; k < i; k++) {
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
```

Special case.

```
// ========= splitUpAt =========
template<class T>
void NTree<T>::splitUpAt(int i) {
    if (_data->size() <= 1) {
        return;
    }
    if (i < 0 || _data->size() <= i) {
        cerr << "splitUpAt precondition 0 <= i < _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    // Build a new left subtree with new data and new children:
    auto newData = make_shared<VectorT<T>>();
    auto newChildren = make_shared<VectorT<shared_ptr<NTree<T>>>>();
    T rootDat = _data->remove(i); // This element will be at the new root.
    for (int k = 0; k < i; k++) {
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
```

(b) Allocate new data and children vectors. Remove and save 60.

```
// ========= splitUpAt =========
template<class T>
void NTree<T>::splitUpAt(int i) {
    if (_data->size() <= 1) {
        return;
    }
    if (i < 0 || _data->size() <= i) {
        cerr << "splitUpAt precondition 0 <= i < _data->size() violated."
             << endl;
        cerr << "i == " << i << ", _data->size() == " << _data->size()
             << endl;
        throw -1;
    }
    // Build a new left subtree with new data and new children:
    auto newData = make_shared<VectorT<T>>();
    auto newChildren = make_shared<VectorT<shared_ptr<NTree<T>>>>();
    T rootDat = _data->remove(i); // This element will be at the new root.
    for (int k = 0; k < i; k++) {
        cerr << "Exercise for the student." << endl;
        throw -1;
    }
```

(c) In a loop, remove 70 and *a* and append to new data and new children.
(d) In a loop, remove 10 and *b* and append to new data and new children.

```
    newChildren->append(_children->remove(0));
    shared_ptr<NTree<T>> left, right;
    if (newData->size() > 0) {
        left = shared_ptr<NTree<T>>(new NTree<T>(newData, newChildren));
    } else {
        left = newChildren->remove(0);
    }
    if (_data->size() > 0) {
        right = shared_ptr<NTree<T>>(new NTree<T>(_data, _children));
    } else {
        right = _children->remove(0);
    }
    _data = make_shared<VectorT<T>>();
    _data->append(rootDat);
    cerr << "Exercise for the student." << endl;
    throw -1;
}
```

```
    newChildren->append(_children->remove(0));
    shared_ptr<NTree<T>> left, right;
    if (newData->size() > 0) {
        left = shared_ptr<NTree<T>>(new NTree<T>(newData, newChildren));
    } else {
        left = newChildren->remove(0);
    }
    if (_data->size() > 0) {
        right = shared_ptr<NTree<T>>(new NTree<T>(_data, _children));
    } else {
        right = _children->remove(0);
    }
    _data = make_shared<VectorT<T>>();
    _data->append(rootDat);
    cerr << "Exercise for the student." << endl;
    throw -1;
}
```

(e) Remove *c* and append to new children.

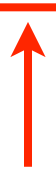
```
    newChildren->append(_children->remove(0));
    shared_ptr<NTree<T>> left, right;
    if (newData->size() > 0) {
        left = shared_ptr<NTree<T>>(new NTree<T>(newData, newChildren));
    } else {
        left = newChildren->remove(0);
    }
    if (_data->size() > 0) {
        right = shared_ptr<NTree<T>>(new NTree<T>(_data, _children));
    } else {
        right = _children->remove(0);
    }
    _data = make_shared<VectorT<T>>();
    _data->append(rootDat);
    cerr << "Exercise for the student." << endl;
    throw -1;
}
```

(f) Make a new tree for left with new data and new children, and a new tree for right with current data and current children.

```
    newChildren->append(_children->remove(0));
    shared_ptr<NTree<T>> left, right;
    if (newData->size() > 0) {
        left = shared_ptr<NTree<T>>(new NTree<T>(newData, newChildren));
    } else {
        left = newChildren->remove(0);
    }
    if (_data->size() > 0) {
        right = shared_ptr<NTree<T>>(new NTree<T>(_data, _children));
    } else {
        right = _children->remove(0);
    }
    _data = make_shared<VectorT<T>>();
    _data->append(rootDat);
    cerr << "Exercise for the student." << endl;
    throw -1;
}
```

(g) Allocate a new vector for _data and append 60.

```
    newChildren->append(_children->remove(0));
    shared_ptr<NTree<T>> left, right;
    if (newData->size() > 0) {
        left = shared_ptr<NTree<T>>(new NTree<T>(newData, newChildren));
    } else {
        left = newChildren->remove(0);
    }
    if (_data->size() > 0) {
        right = shared_ptr<NTree<T>>(new NTree<T>(_data, _children));
    } else {
        right = _children->remove(0);
    }
    _data = make_shared<VectorT<T>>();
    _data->append(rootDat);
    cerr << "Exercise for the student." << endl;
    throw -1;
}
```



(g) ...Allocate a new vector for _children and append left and then right.

Visiting `NTree`

Composite State lists and trees have two kinds of nodes:

- Empty
- Nonempty

The composite State *n*-way tree has one kind of node.
The state is the number of data values in the node.

`accept()` only executes one visitor method, `caseAt()`.
The visitor has access to two parameters:

- `size`
- `host`

The abstract NTree visitor

```
// ========= ANTreeVis =========
template<class T>
class ANTreeVis {
public:
    virtual ~ANTreeVis() = default;

    virtual void caseAt(int size, NTree<T> &host) = 0;
    virtual void caseAt(int size, NTree<T> const &host) = 0;
};
```

The accept() method

```
// ========== accept ==========
template<class T>
void NTree<T>::accept(ANTreeVis<T> &visitor) {
    visitor.caseAt(_data->size(), *this);
}

// ========== accept const ==========
template<class T>
void NTree<T>::accept(ANTreeVis<T>& visitor) const {
    visitor.caseAt(_data->size(), *this);
}
```

The RootSize visitor

```
// Pre: This visitor has been accepted by a host tree.
// Post: The size of the root of the host tree is returned.
// The root size of an empty tree is 0.
```

```
// ========= NTreeRootSizeVis =========
template<class T>
class NTreeRootSizeVis : public ANTreeVis<T> {
private:
    int _result; // Output result.

public:
    // ========= Constructor =========
    NTreeRootSizeVis() :
        _result(-1) {
    }

    // ========= visit =========
    void caseAt(int size, NTree<T>& host) override {
        _result = size;
    }

    // ========= visit const =========
    void caseAt(int size, NTree<T> const &host) override {
        _result = size;
    }
```

```
    // ========= result =========
    // Pre: This visitor has been accepted by a host tree.
    // Post: The size of the root of the host tree is returned.
    // The root size of an empty tree is 0.

    int result() const {
        return _result;
    }
};

// Global function for convenience
template<class T>
int rootSize(NTree<T> const &tree) {
    NTreeRootSizeVis<T> rootSizeVis;
    tree.accept(rootSizeVis);
    return rootSizeVis.result();
}
```

The InOrder visitor

Review the PreOrderTraversal visitor of BiTreeCSV

`BiTCSVpreOrderVis`

The `BiTreeCS` version of `preOrder()` is void.

Therefore, the visitor version does not provide a `result()` method.

The `BiTreeCS` version has output parameter `os`, so the visitor version has reference variable attribute `_os`.

```
// ========= BiTCSVpreOrderVis =========
template<class T>
class BiTCSVpreOrderVis : public ABiTreeCSVVis<T> {
private:
    ostream &_os; // Input paramter.

public:
    // ========= Constructor =========
    BiTCSVpreOrderVis(ostream &os):
        _os(os) {
    }

    // ========= visit =========
    // Pre: This visitor has been accepted by a host tree.
    // Post: A preorder representation of this tree is sent to os.
    void emptyCase(BiTreeCSV<T> &host) override {
    }

    void nonEmptyCase(BiTreeCSV<T> &host) override {
        _os << host.root() << "  ";
        host.left().accept(*this);
        host.right().accept(*this);
    }
};
```

```
// Global function for convenience
template<class T>
void preOrder(ostream &os, BiTreeCSV<T> const &tree) {
    BiTCSVpreOrderVis<T> preOrderVis(os);
    tree.accept(preOrderVis);
}
```

```
// ========== NTreeInOrderVis ==========
template<class T>
class NTreeInOrderVis : public ANTreeVis<T> {
private:
    ostream &_os; // Input paramter.

public:
    // ========== Constructor ==========
    NTreeInOrderVis(ostream &os) :
        _os(os) {
    }
```

```
    // ========= visit =========
    void caseAt(int size, NTree<T>& host) override {
        switch (size) {
        case 0:
        {
            cerr << "NTreeInOrderVis: Exercise for the student."
                 << endl;
            throw -1;
        }
        default:
        {
            cerr << "NTreeInOrderVis: Exercise for the student."
                 << endl;
            throw -1;
        }
        }
    }
    // ========= visit const =========
    ...
};
```

```
// Global function for convenience
template<class T>
void inOrder(ostream &os, NTree<T> const &tree) {
    cerr << "inOrder: Exercise for the student." << endl;
    throw -1;
}
```