Trees: intro

Vectors versus lists

- Vectors are very good for random accesses
  - Can read or write at any index in one step
  - Can only *efficiently* insert at the end
- Linked lists are very good for insertions
  - Can insert or remove at the front in one step
  - Any other operation may be slow
- Both vector and list are oriented towards positions
  - What value is at a given position?
- How can we access efficiently by value?
  - Is value x present?
  - Remove value x if it exists

A simple binary tree

```
struct my_tree
{
    string value;
    my_tree *left;
    my_tree *right;
};
```

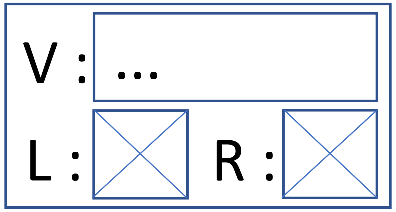
- A binary tree consists of nodes, where each node has:
    - A value
    - A pointer to: a left sub-tree; or nothing
    - A pointer to: a right sub-tree; or nothing

A simple binary tree

```
struct my_tree
{
    string value;
    my_tree *left;
    my_tree *right;
};
```

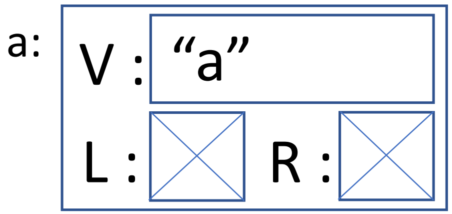
For the sake of compact diagrams, we'll use:

```
struct my_tree
{
    string V;
    my_tree *L;
    my_tree *R;
};
```

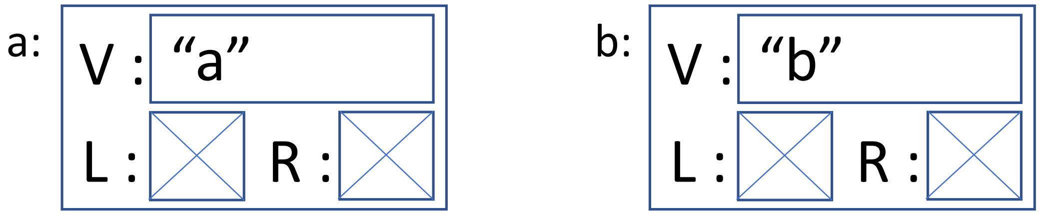


In code you should use descriptive names

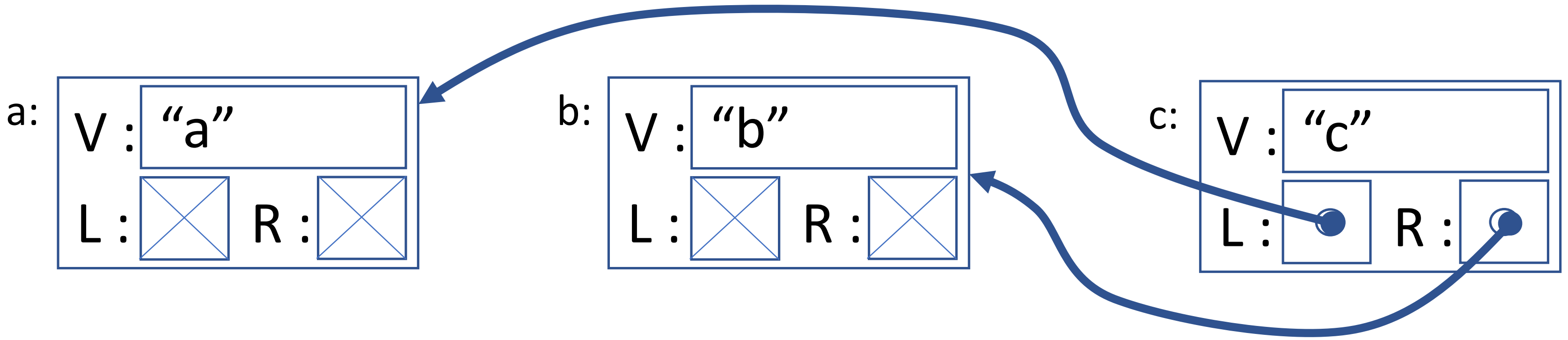
```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



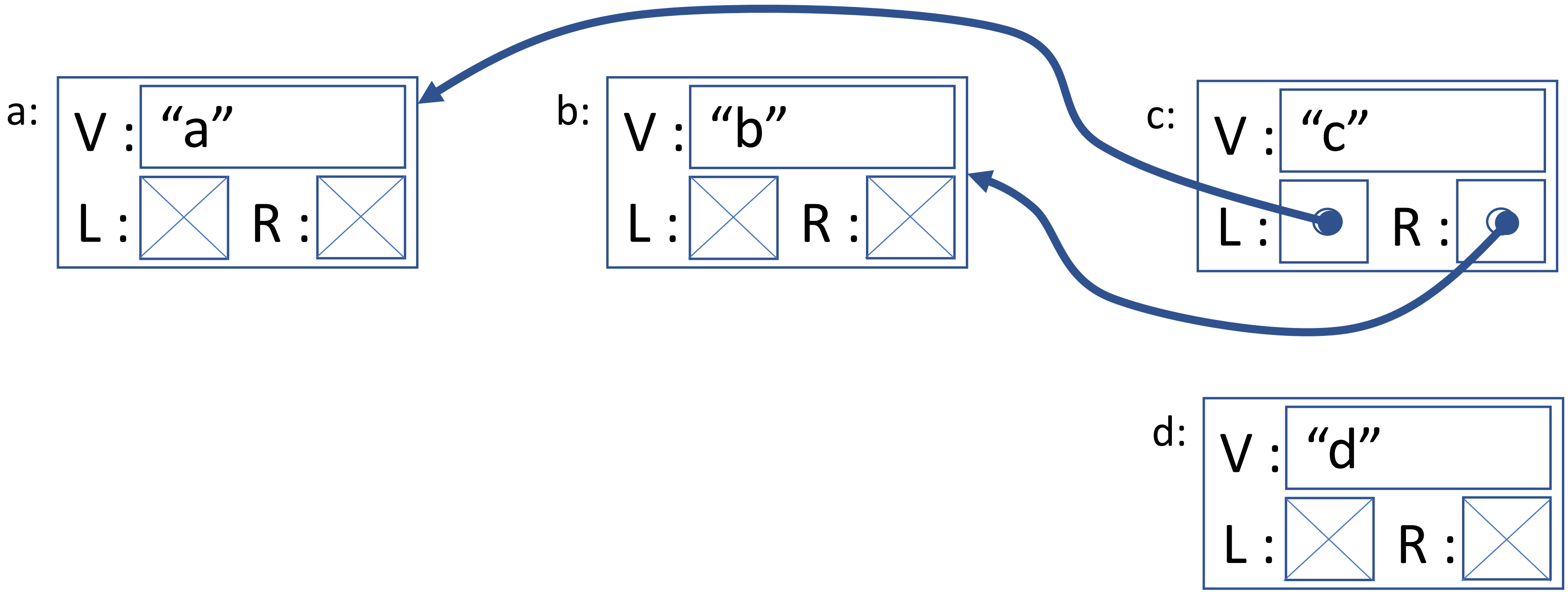

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



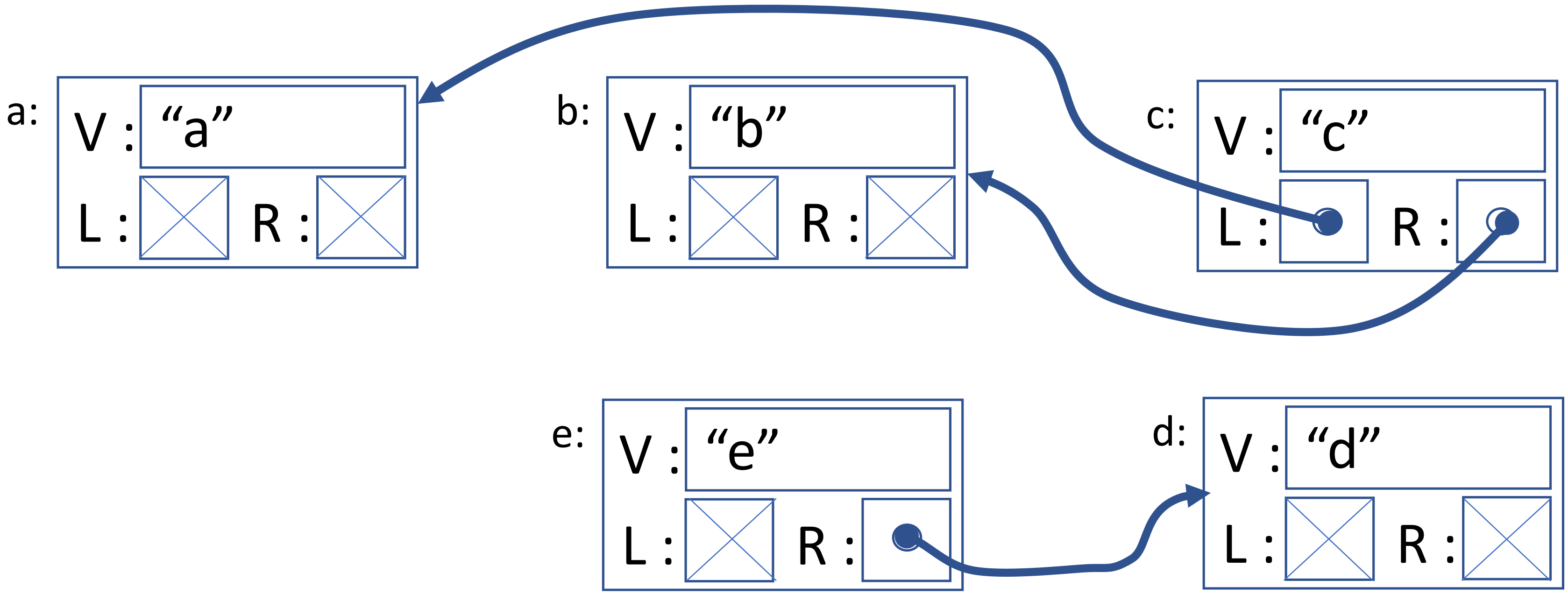

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



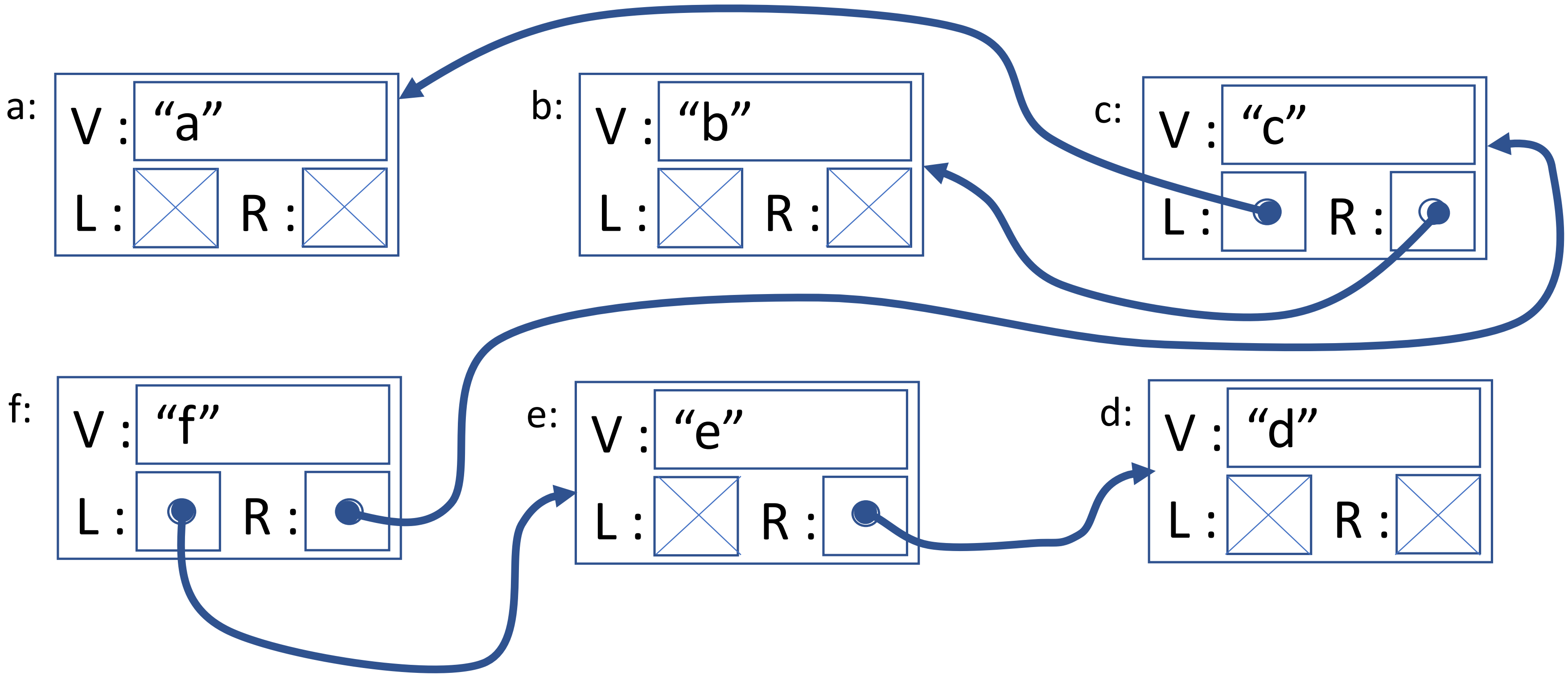

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



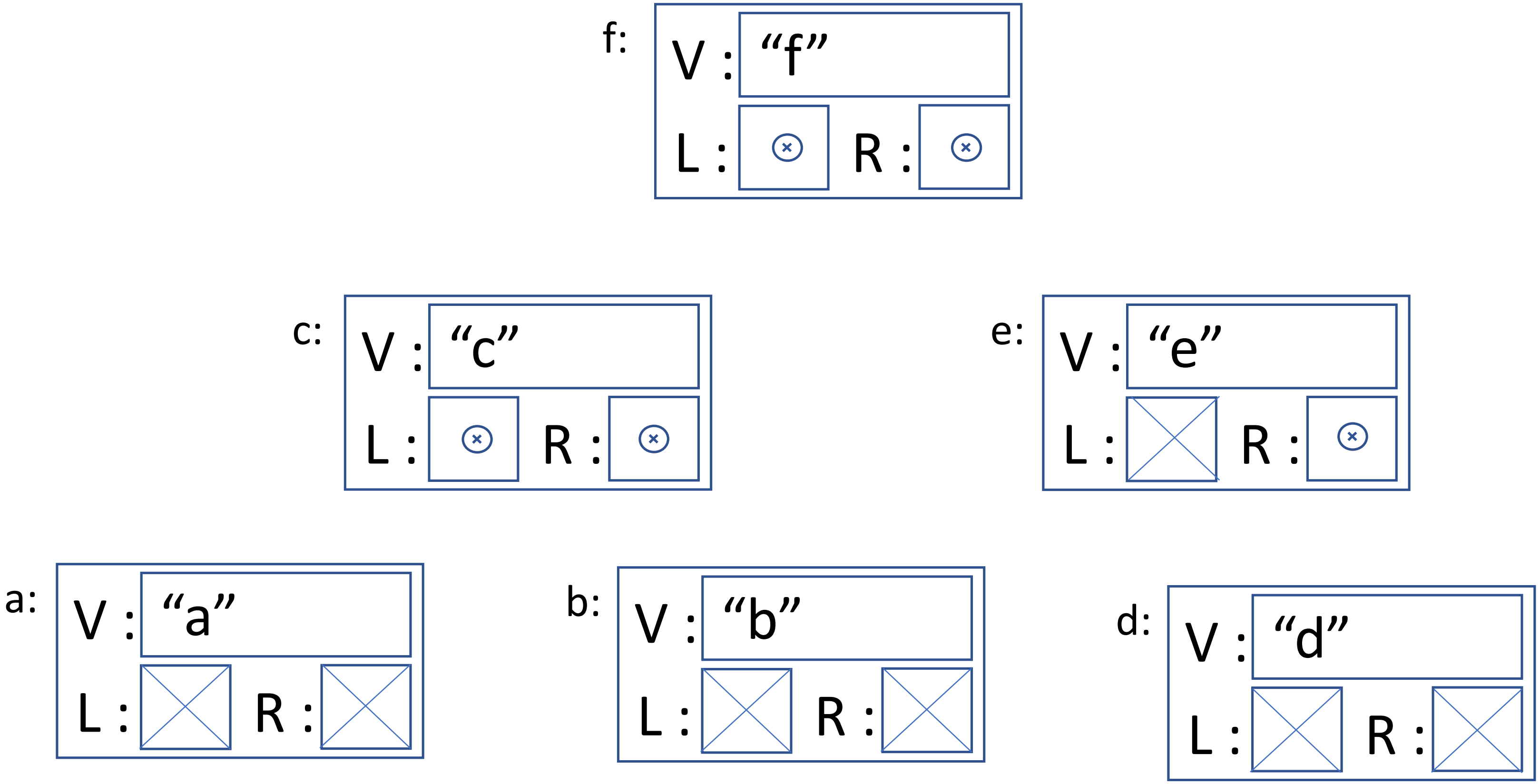

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



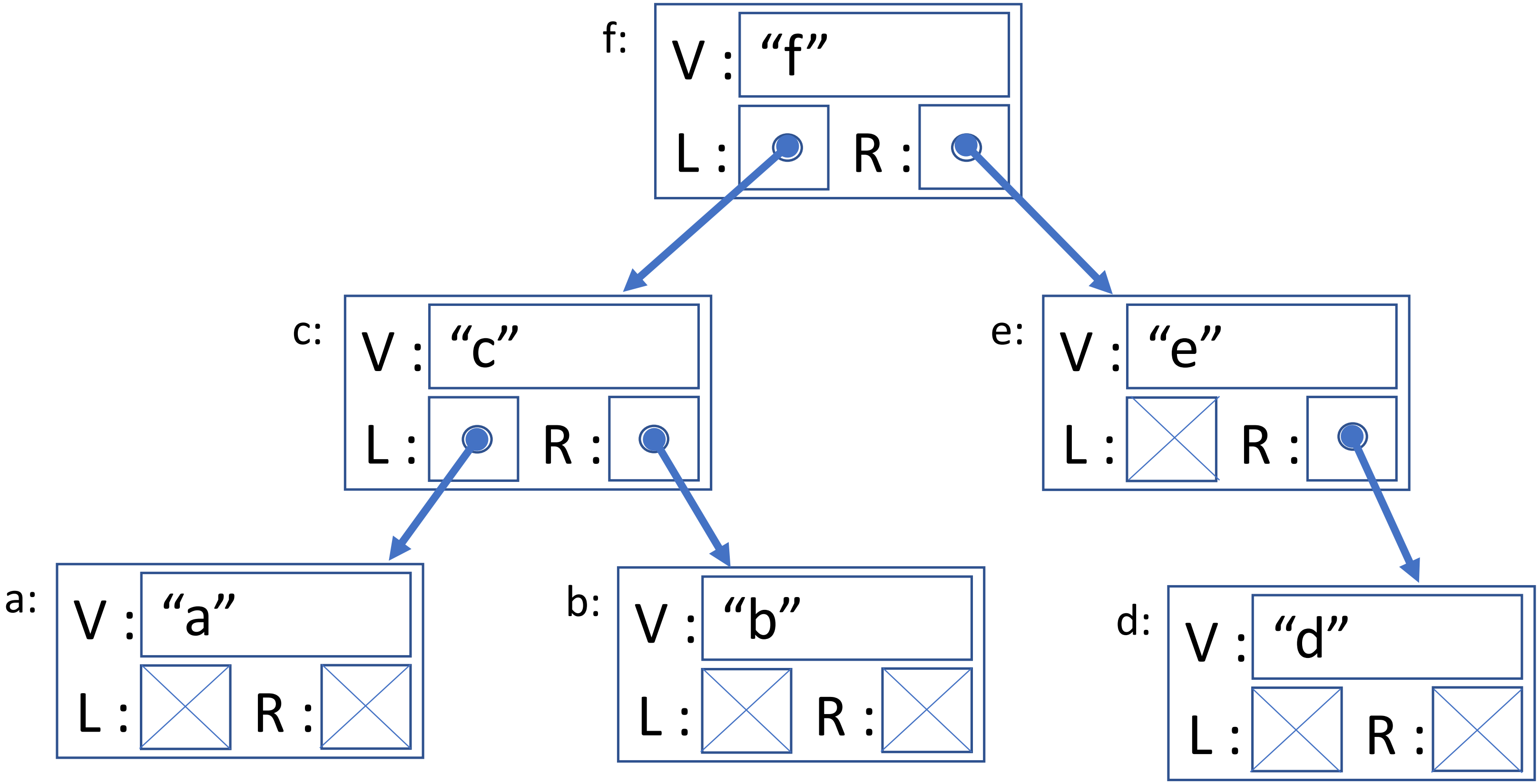

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```




```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```

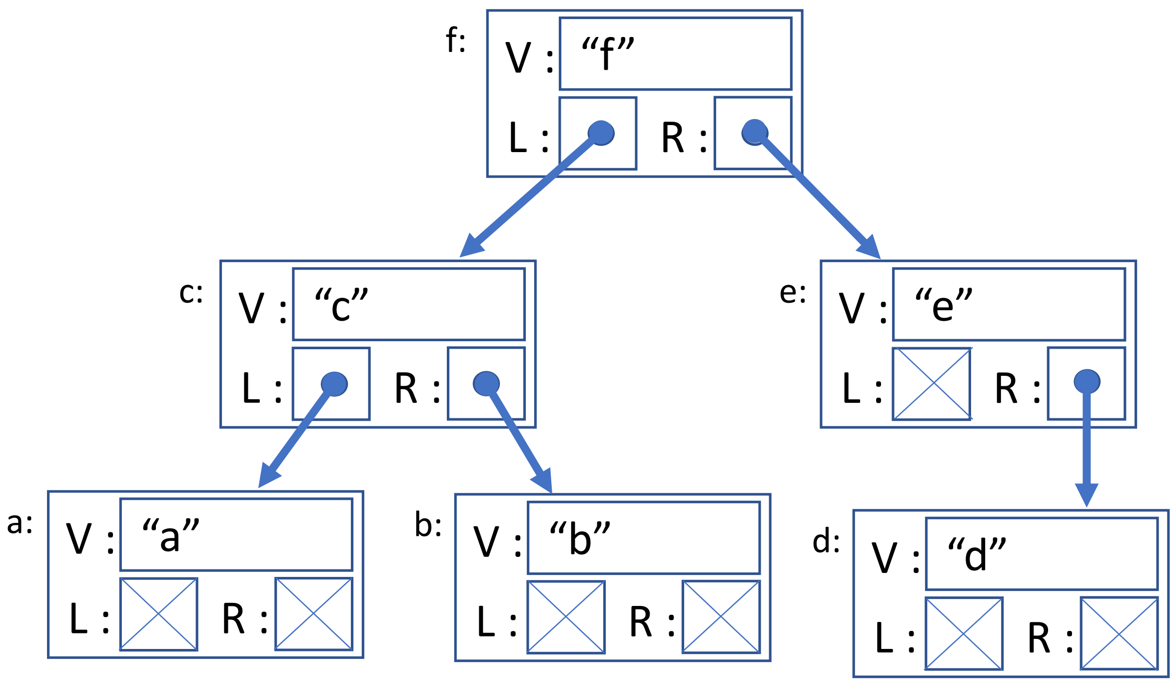



```cpp
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```




```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```

```
void print(my_tree *t)
{
  if(t != nullptr){
     print(t->left);
     cout << t->value << endl;
     print(t->right);
  }
}

int main()
{
  // ... build tree
  print(&f);
}
```



Binary Search Trees

A binary search tree

```
struct my_tree
{
    string value;
    my_tree *left;
    my_tree *right;
};
```

- A binary tree consists of nodes, with each node having:
  - A value
  - A pointer to a left binary sub-tree
  - A pointer to a right binary sub-tree
- A binary search tree adds two constraints:
  - Every value in the left sub-tree is less than our value
  - Every value in the right sub-tree is not less than our value

A binary search tree

```
struct my_tree
{
    string value;
    my_tree *left;
    my_tree *right;
};
```

- A ***binary search tree*** consists of nodes with:
  - A value
  - A pointer to a left ***binary search sub-tree***
  - A pointer to a right ***binary search sub-tree***
  - Every value in left sub-tree is less than our value
  - Every value in right sub-tree is greater than our value

We could choose left < val or left <= val for slightly different semantics

A binary search tree

```
struct my_tree
{
    string value;
    my_tree *left;
    my_tree *right;
};
```

- A binary search tree requires that
  - is_binary_search_tree(left)
  - is_binary_search_tree(right)
  - largest_value(left) is less-than value
  - smallest_value(right) is greater-than value

A binary search tree

```
struct my_tree
{
    string value;
    my_tree *left;
    my_tree *right;
};
```

```
bool is_binary_search_tree(my_tree *t)
{
  return (t == nullptr) // An empty tree is a binary search_tree
    || (
        is_binary_search_tree(t->left)
        && is_binary_search_tree(t->right)
        && is_tree_less_than(t->left, t->value)
        && is_tree_greater_than(t->right, t->value)
    );
}
```

A binary search tree

```
struct my_tree
{
    string value;
    my_tree *left;
    my_tree *right;
};
```

```
bool is_tree_less_than(my_tree *t, string v)
{
  return (t==nullptr) // An empty tree has no value
    || (
        ( t->value < v )
        && is_tree_less_than(t->left, v)
        && is_tree_less_than (t->right, v)
       );
}
```

A binary search tree

```
struct my_tree
{
    string value;
    my_tree *left;
    my_tree *right;
};
```

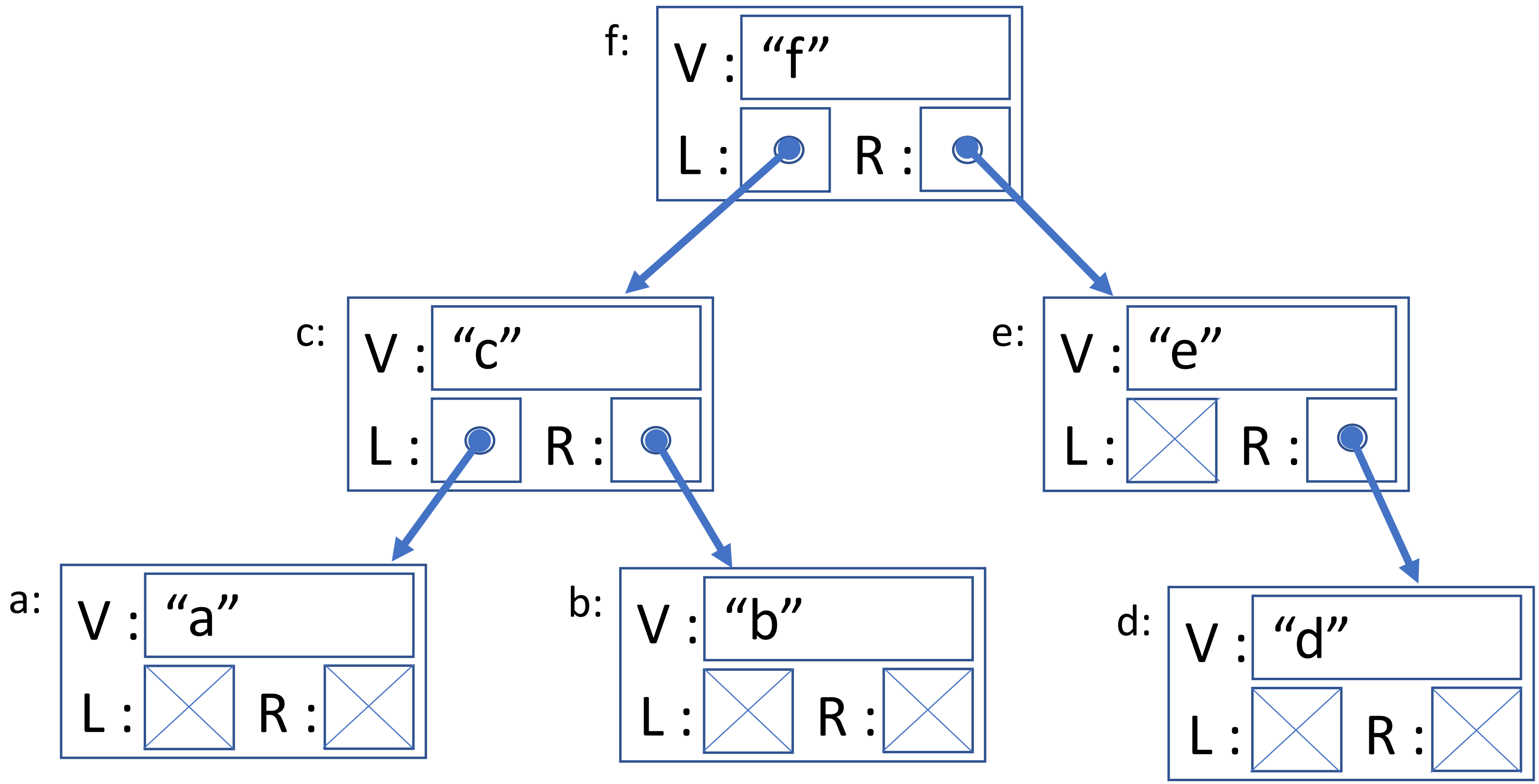
```
bool is_tree_greater_than(my_tree *t, string v)
{
  return (t==nullptr)
    || (
        ( v < t->value )
        && is_tree_greater_than(t->left, v)
        && is_tree_greater_than(t->right, v);
       );
}
```

Binary search tree : type + constraint

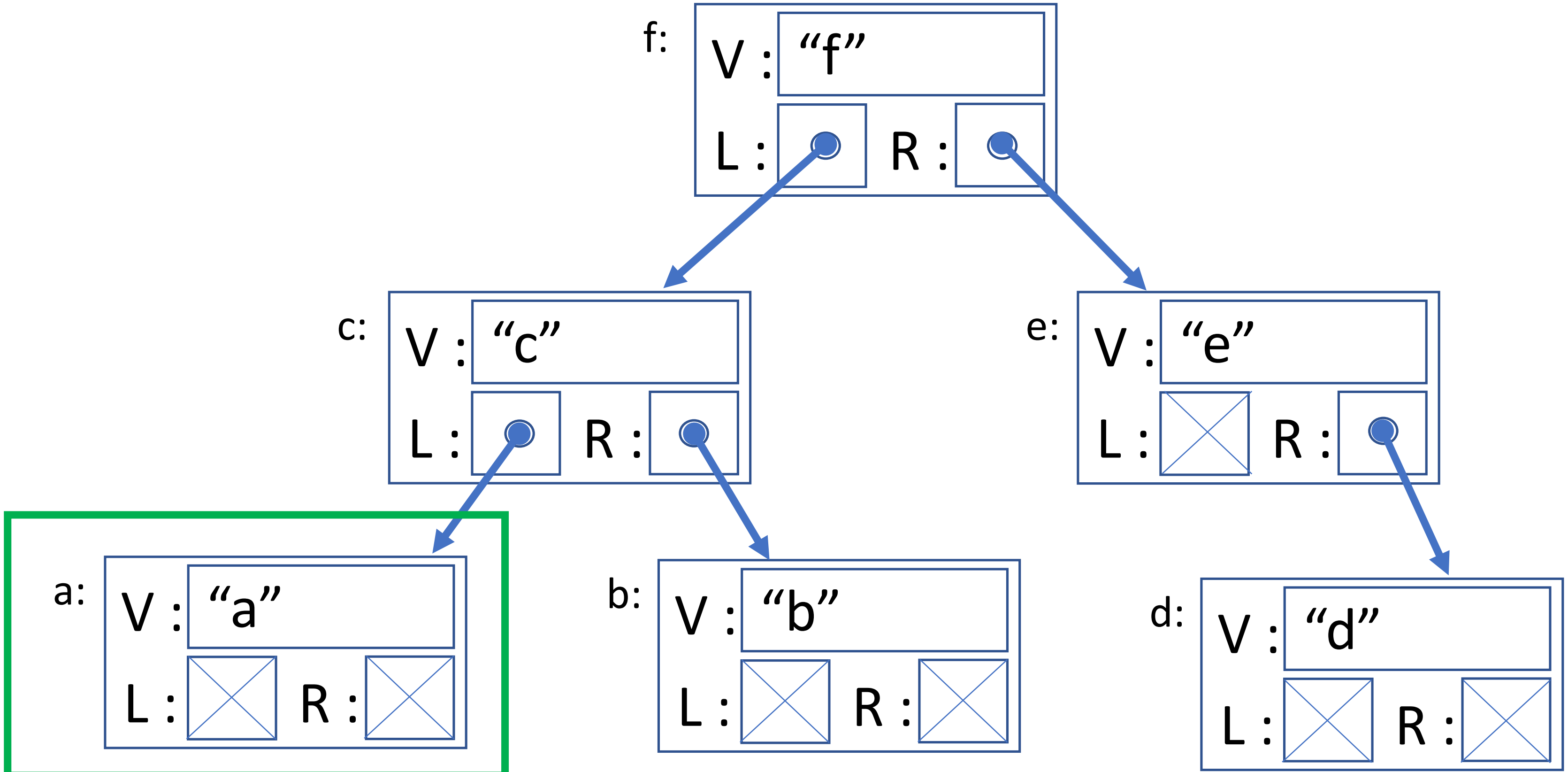
- Constraints on sub-trees refines the type
  - ***Some*** binary trees are binary search tree
  - ***All*** binary search trees are binary trees
- C++ can only enforce compile-time types
  - Are the values of the right type?
  - Are the left and right pointers of the right type?
- Run-time constraints are up to the programmer
  - *Either*: make sure constraints are always met;
  - *Or*: check constraints each time you use it

A caveat : stack vs heap trees

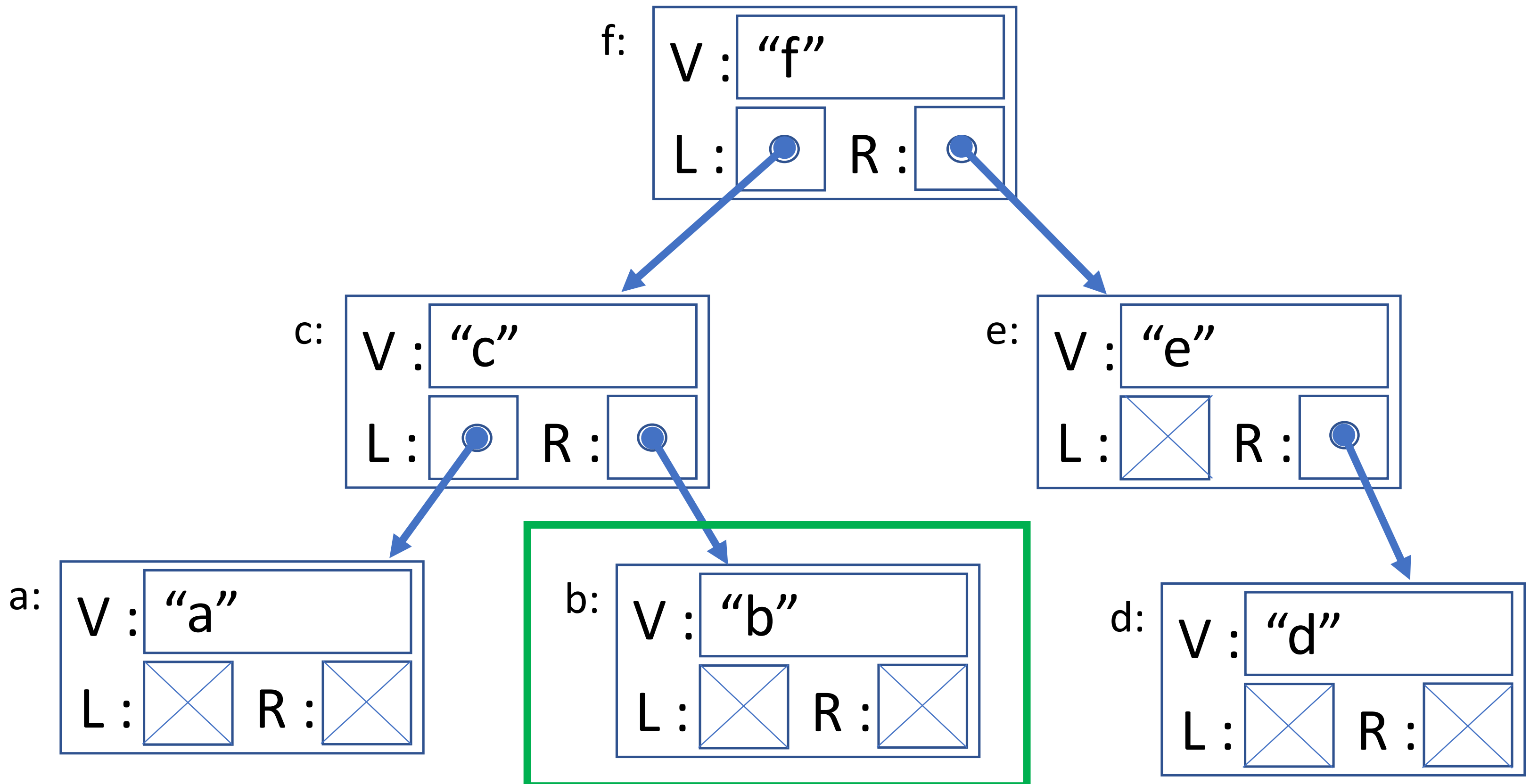
- Most trees are dynamically allocated
  - We'll see it's much easier to manage
- This section uses stack-allocated trees:
  1. To remind you that pointers can point to local variables
  2. To make explicit how tedious manual constraints are
- Later we'll look at heap-allocated trees



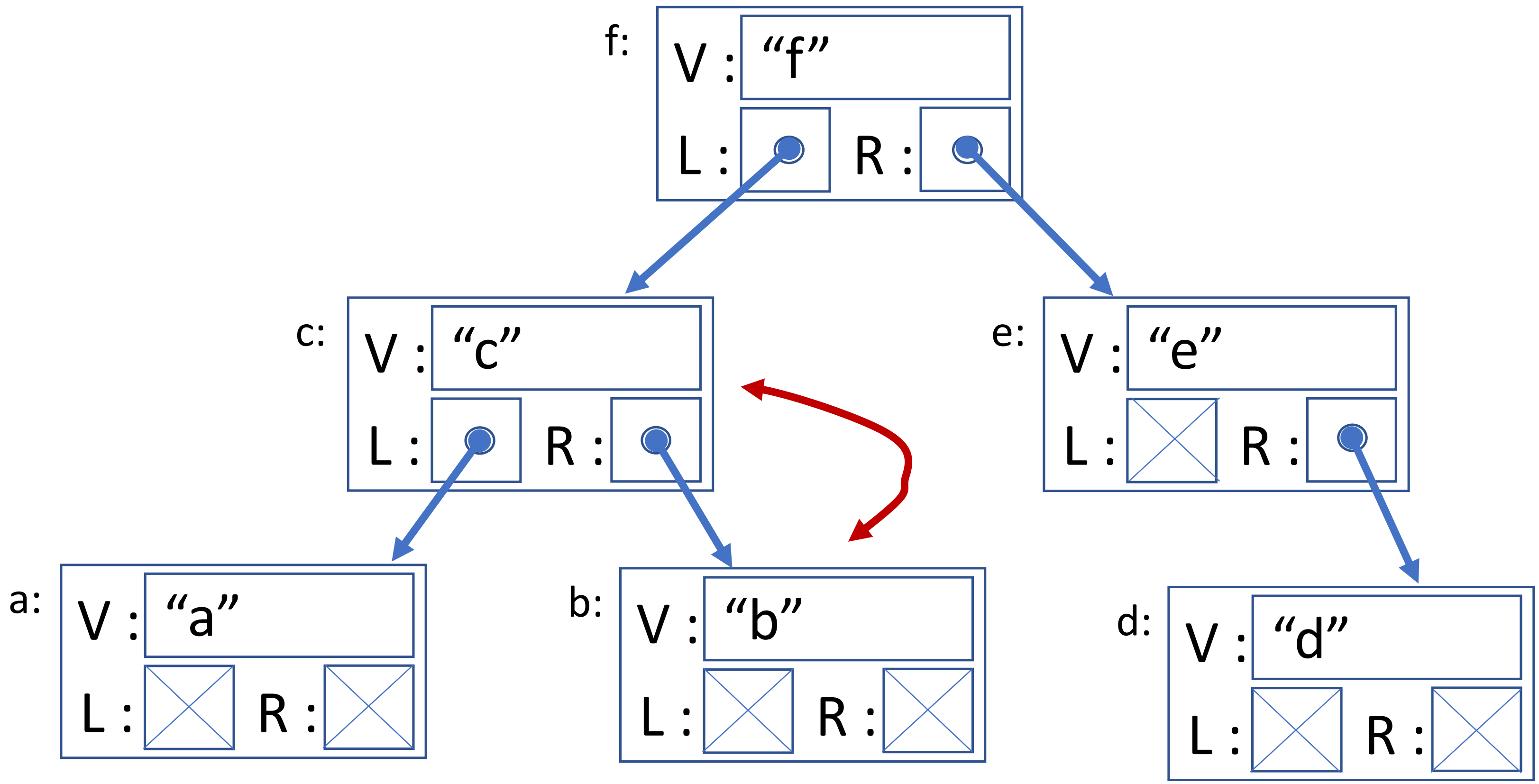

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



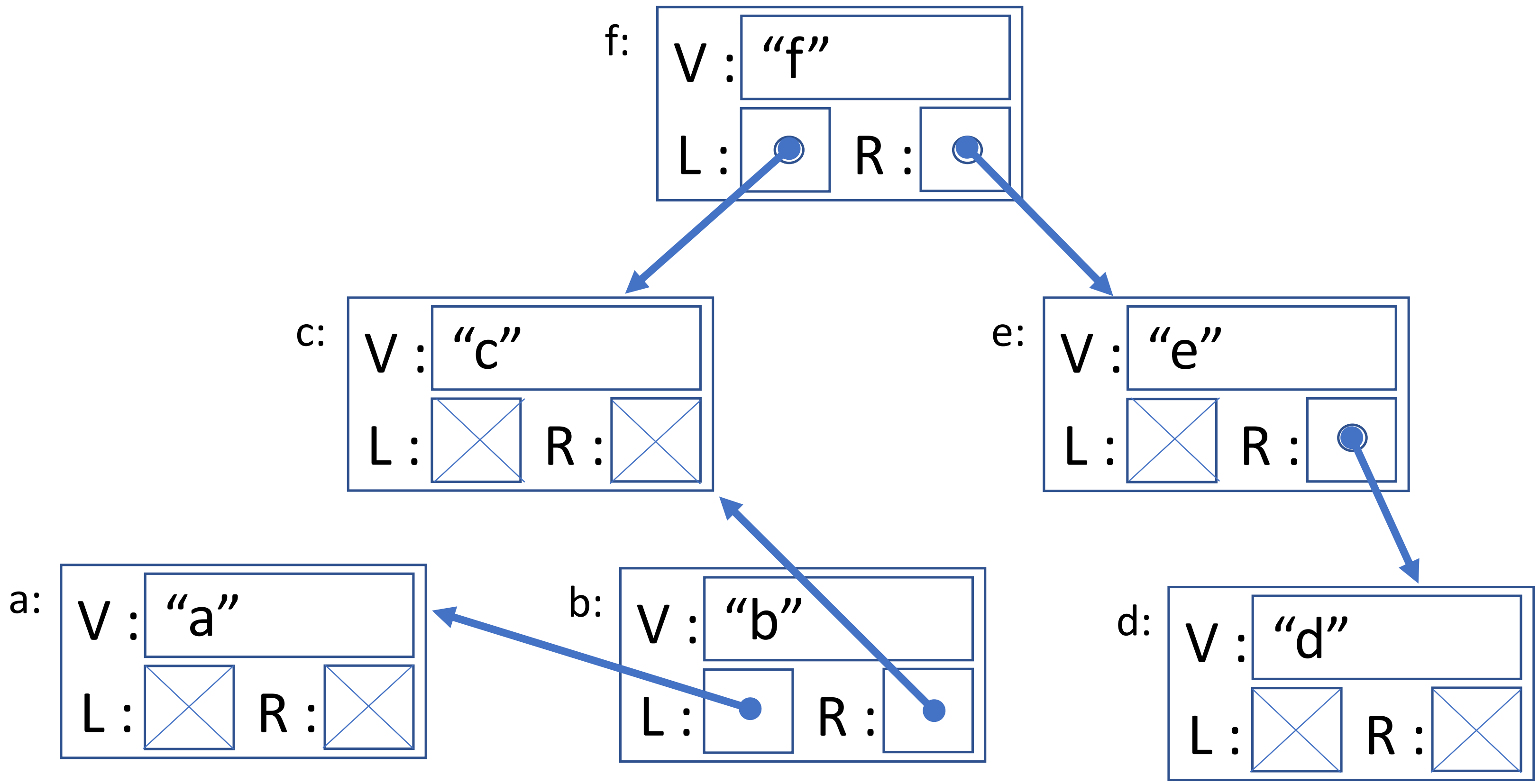

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



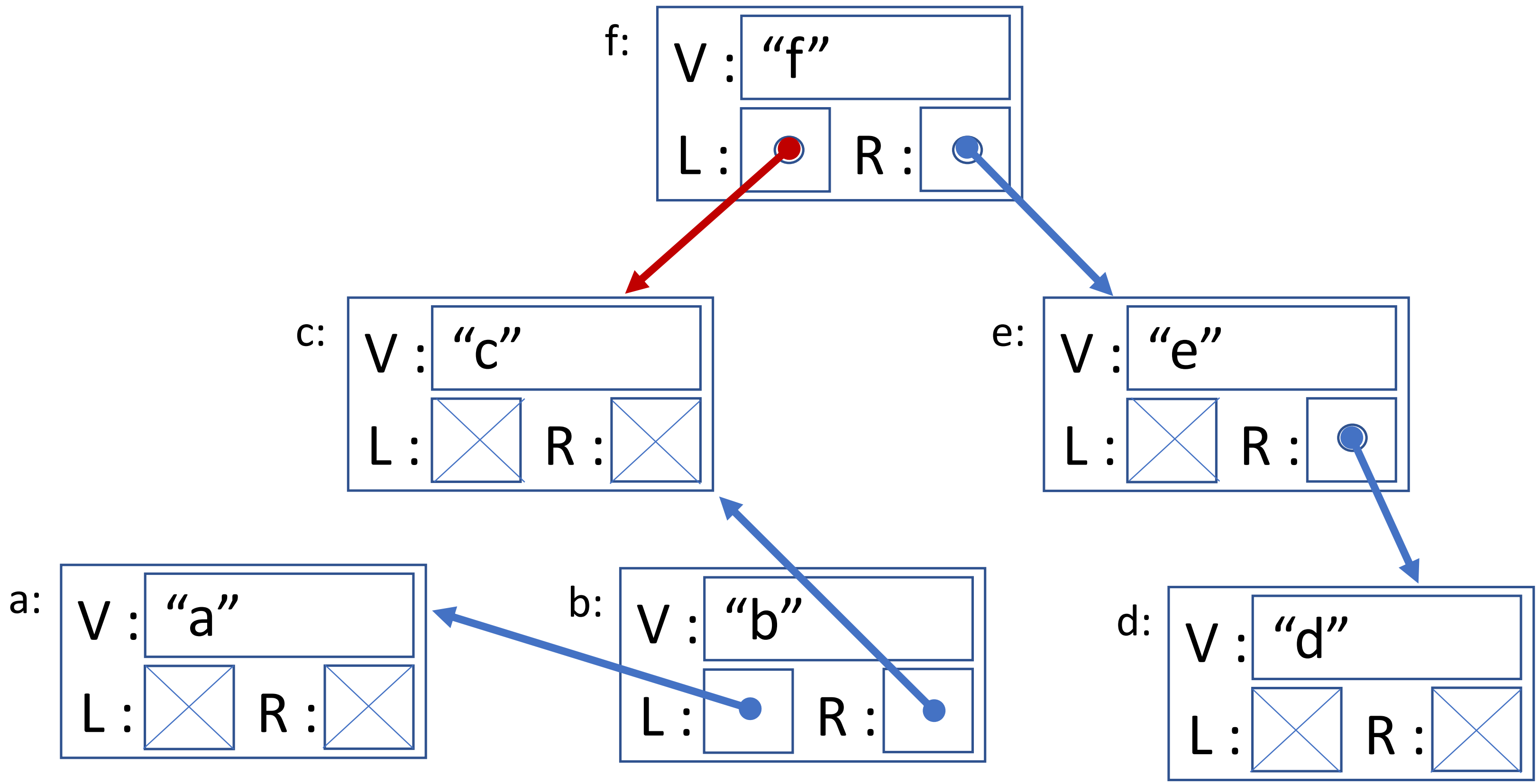

```
int main()
{
      my_tree a={"a", nullptr, nullptr};
      my_tree b={"b", nullptr, nullptr};
      my_tree c={"c", &a,      &b};
      my_tree d={"d", nullptr, nullptr};
      my_tree e={"e", nullptr, &d};
      my_tree f={"f", &e,      &c};
}
```



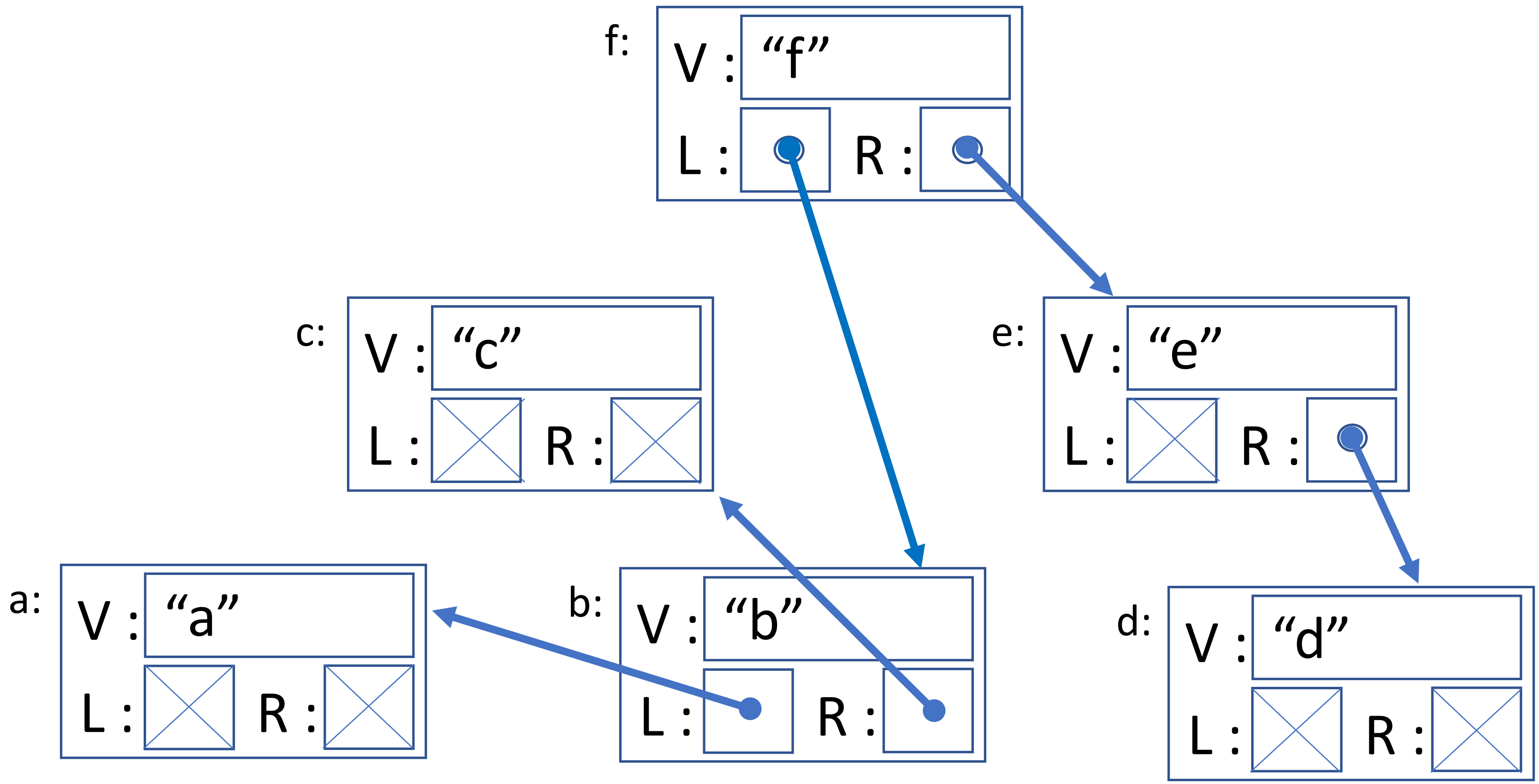

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree b={"b", nullptr, nullptr};
    my_tree c={"c", &a,      &b};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



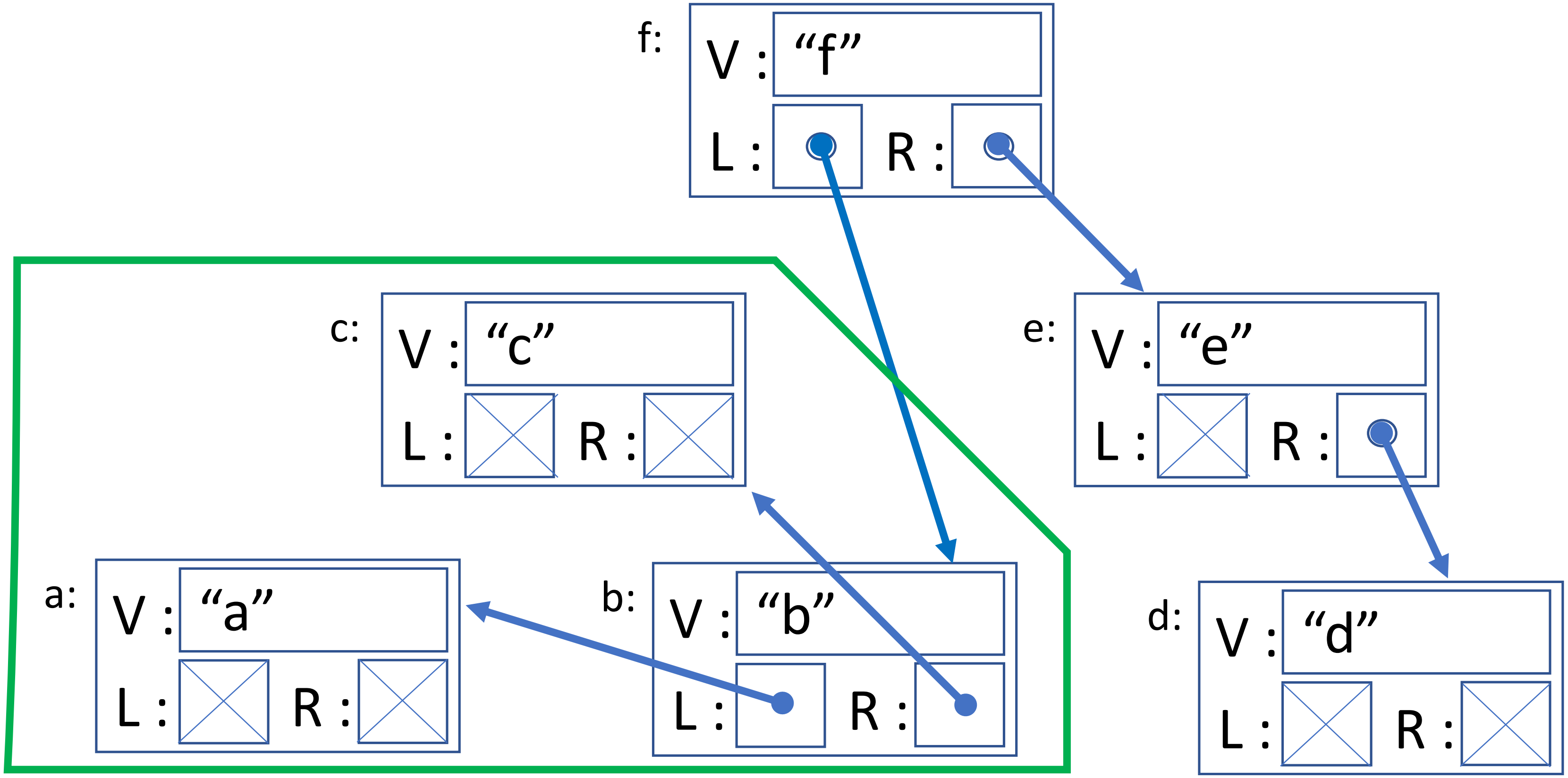

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"b", nullptr, nullptr};
    my_tree b={"c", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



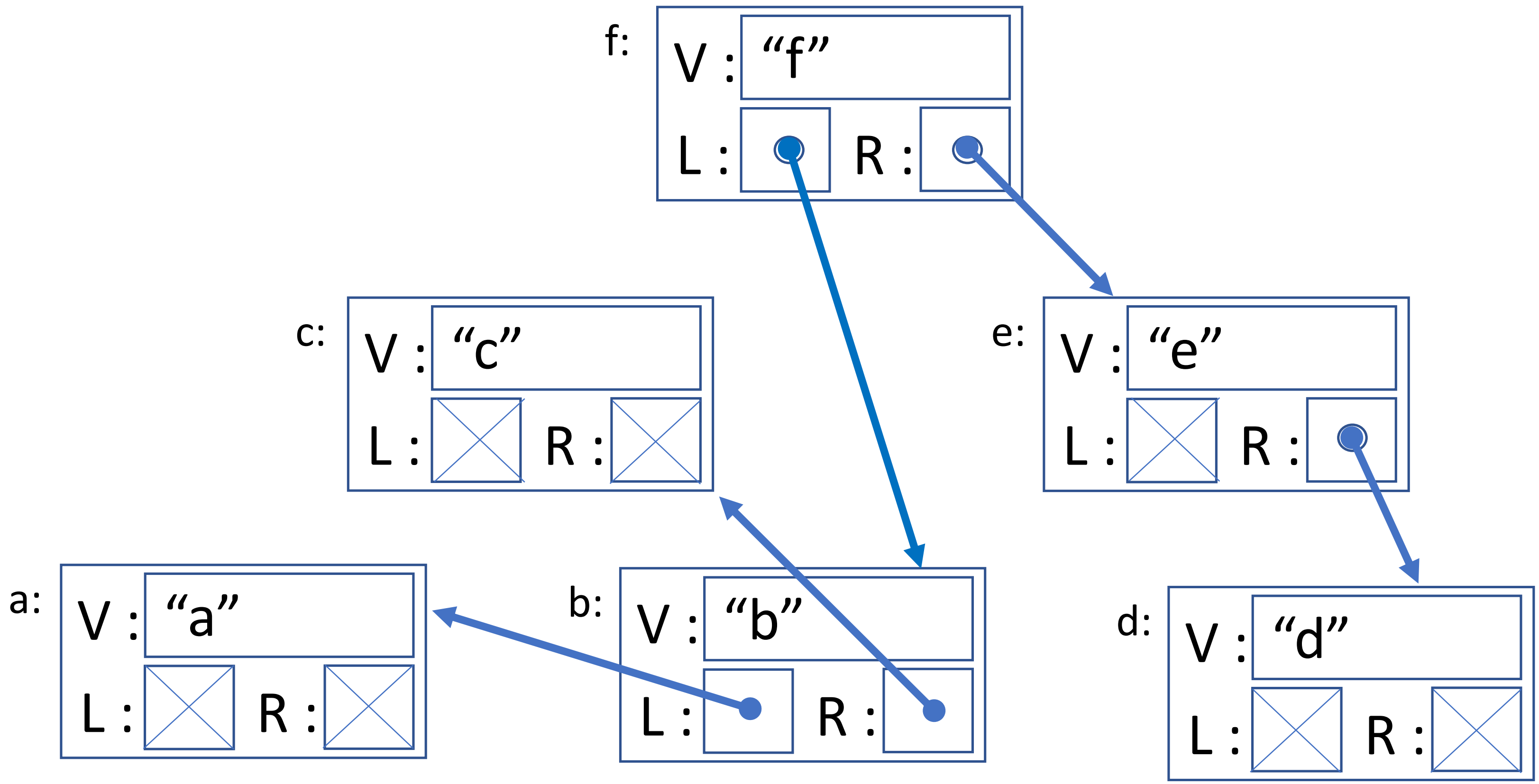

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"b", nullptr, nullptr};
    my_tree b={"c", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &c};
}
```



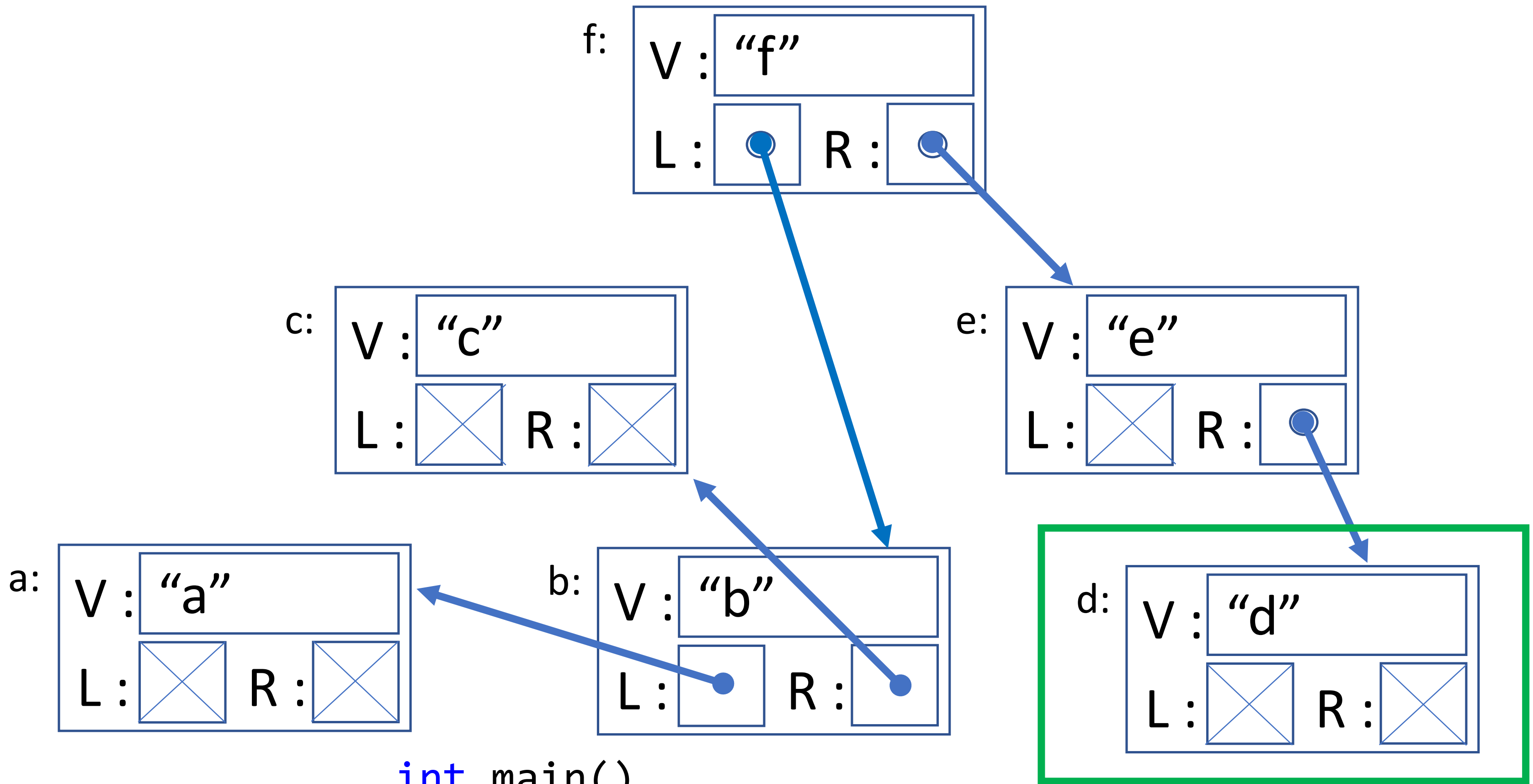

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &b};
}
```



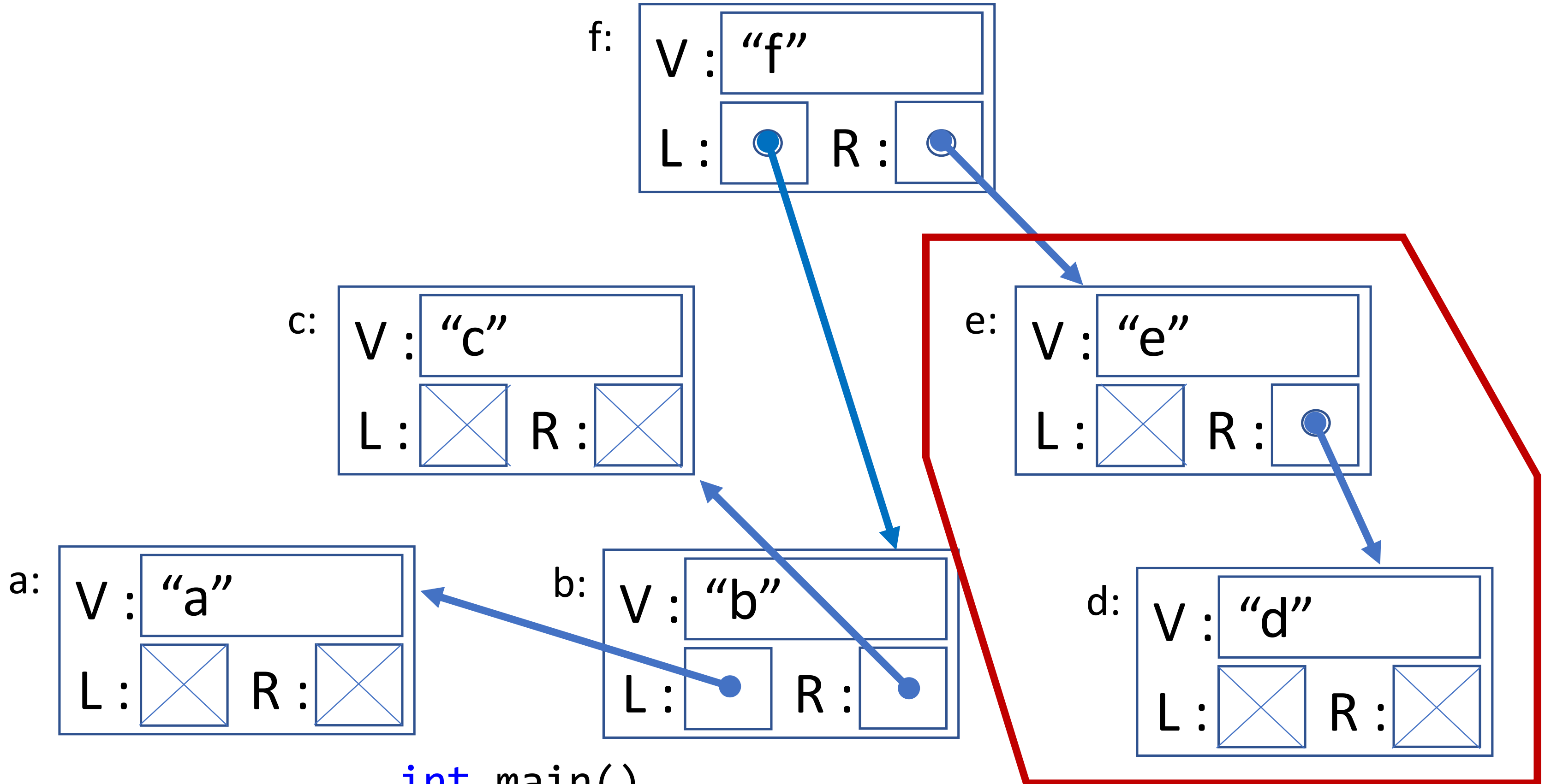

```cpp
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &b};
}
```



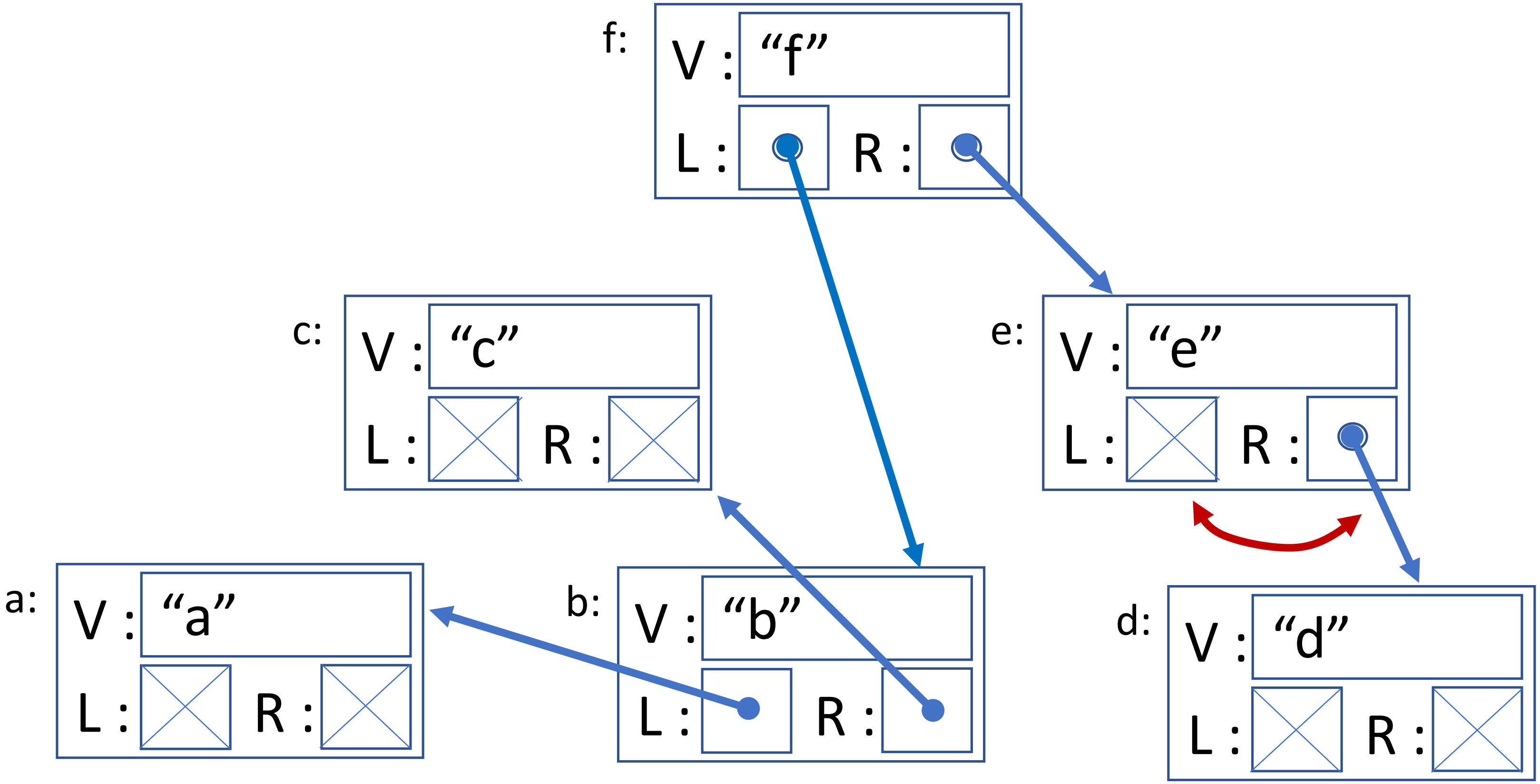

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &b};
}
```



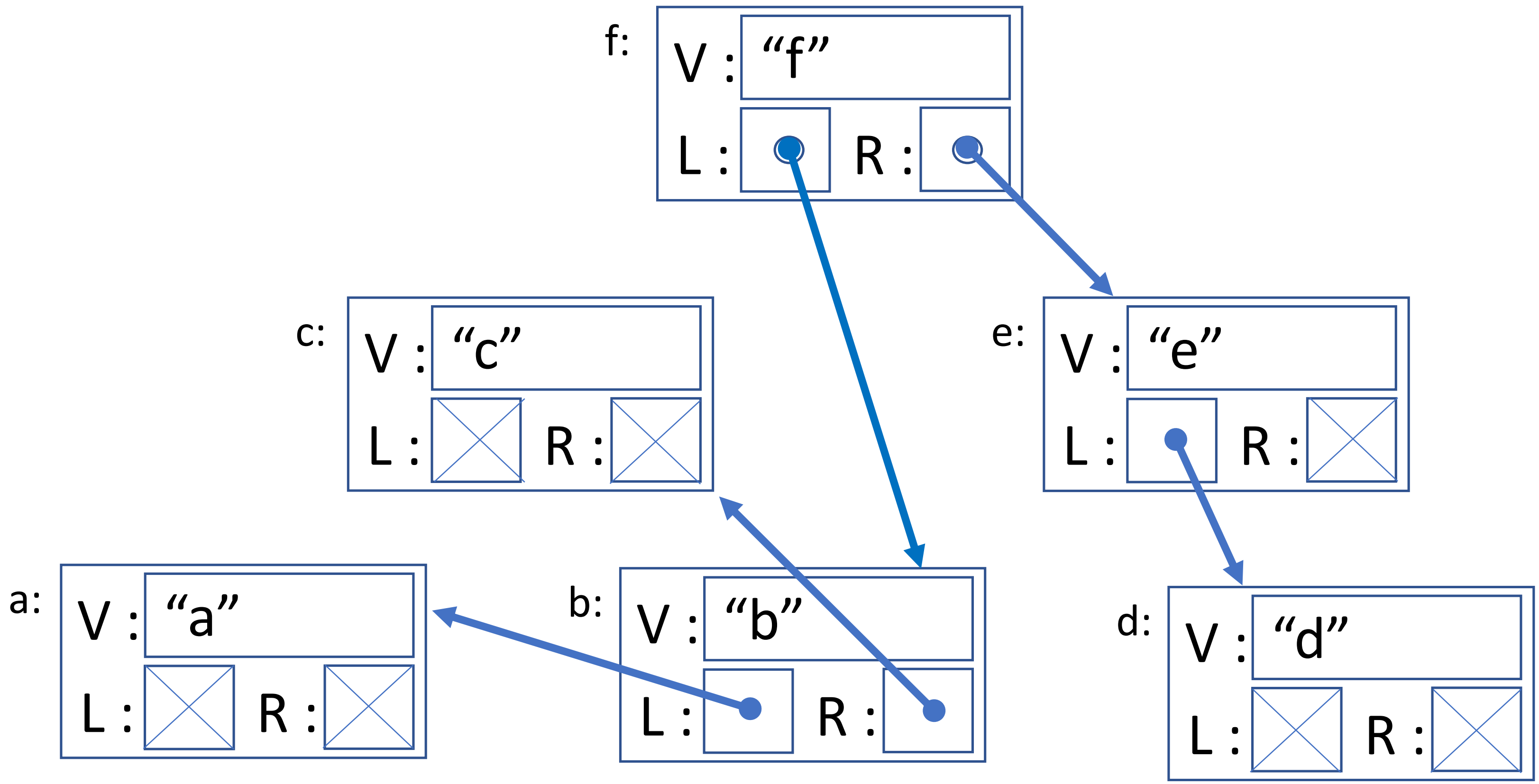

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &b};
}
```



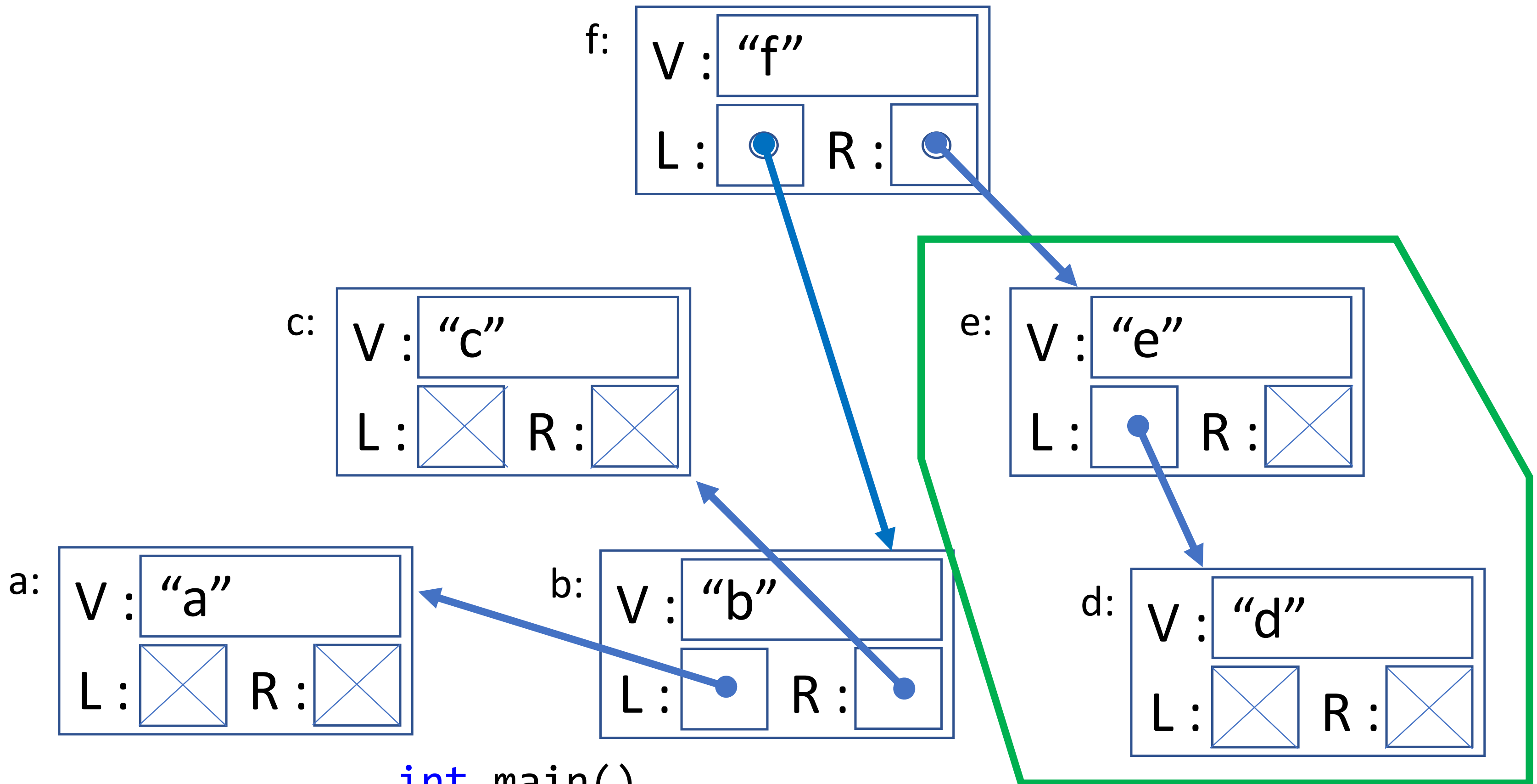

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &b};
}
```



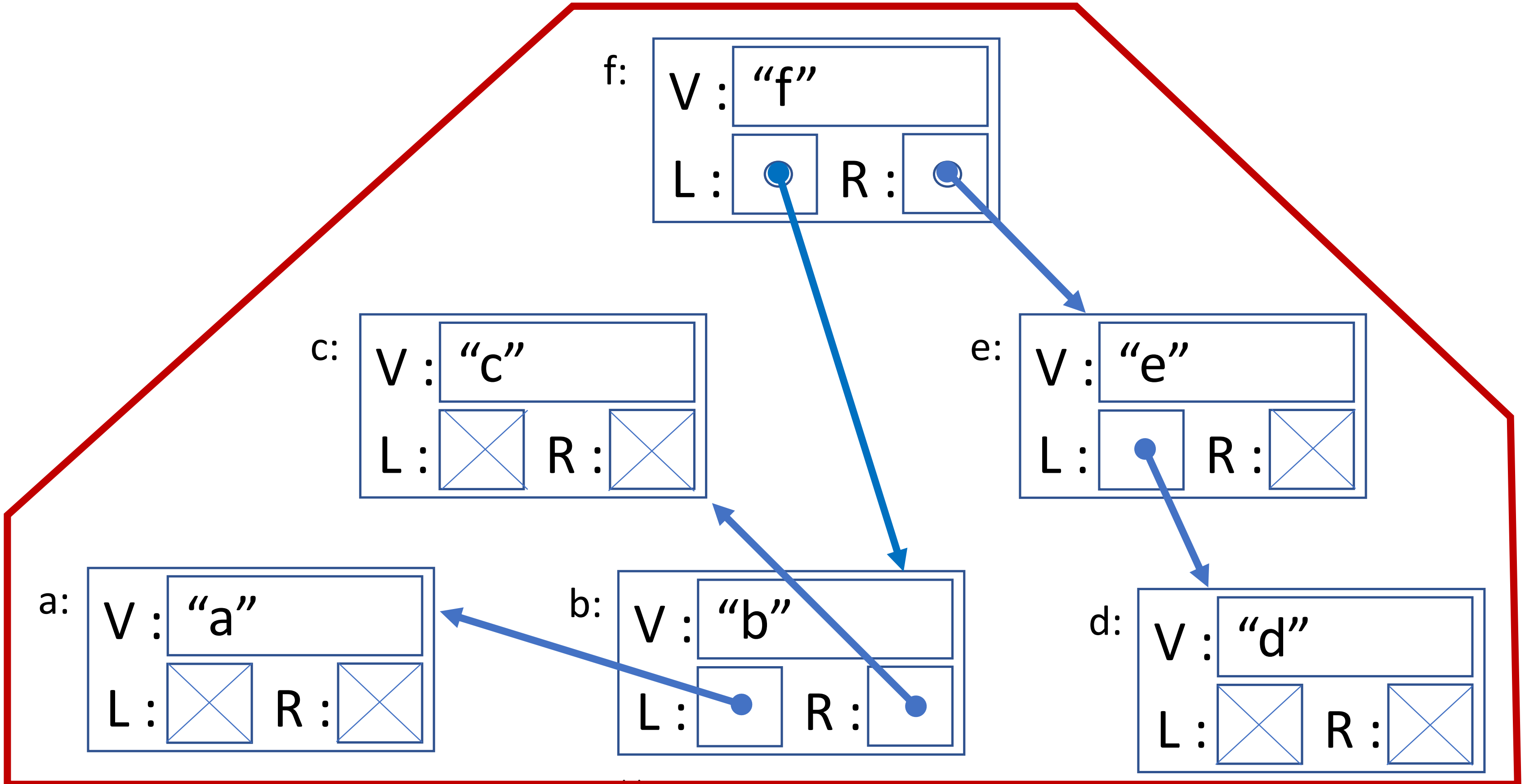

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", nullptr, &d};
    my_tree f={"f", &e,      &b};
}
```



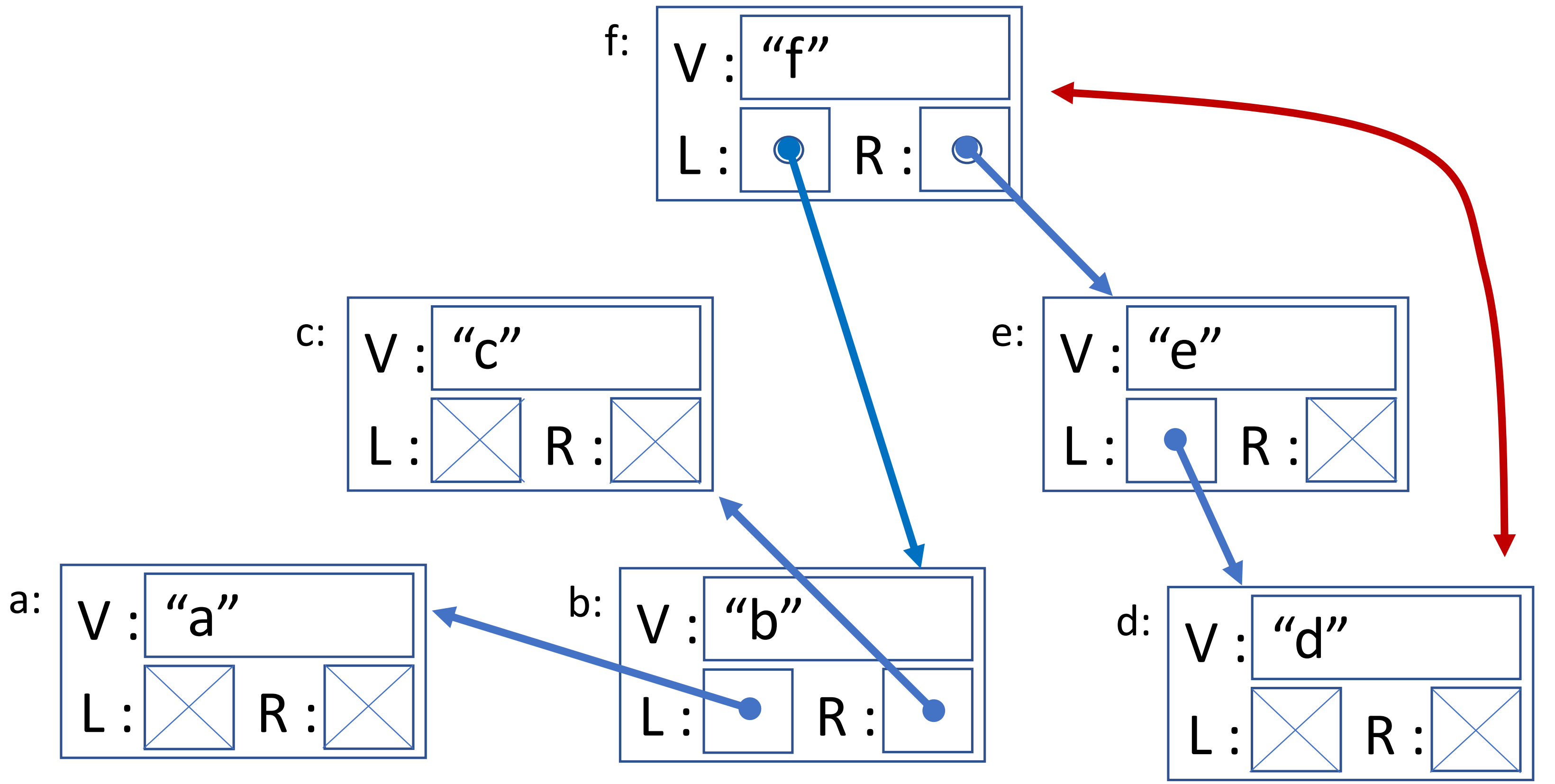

```cpp
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", &d ,     nullptr};
    my_tree f={"f", &e,      &b};
}
```



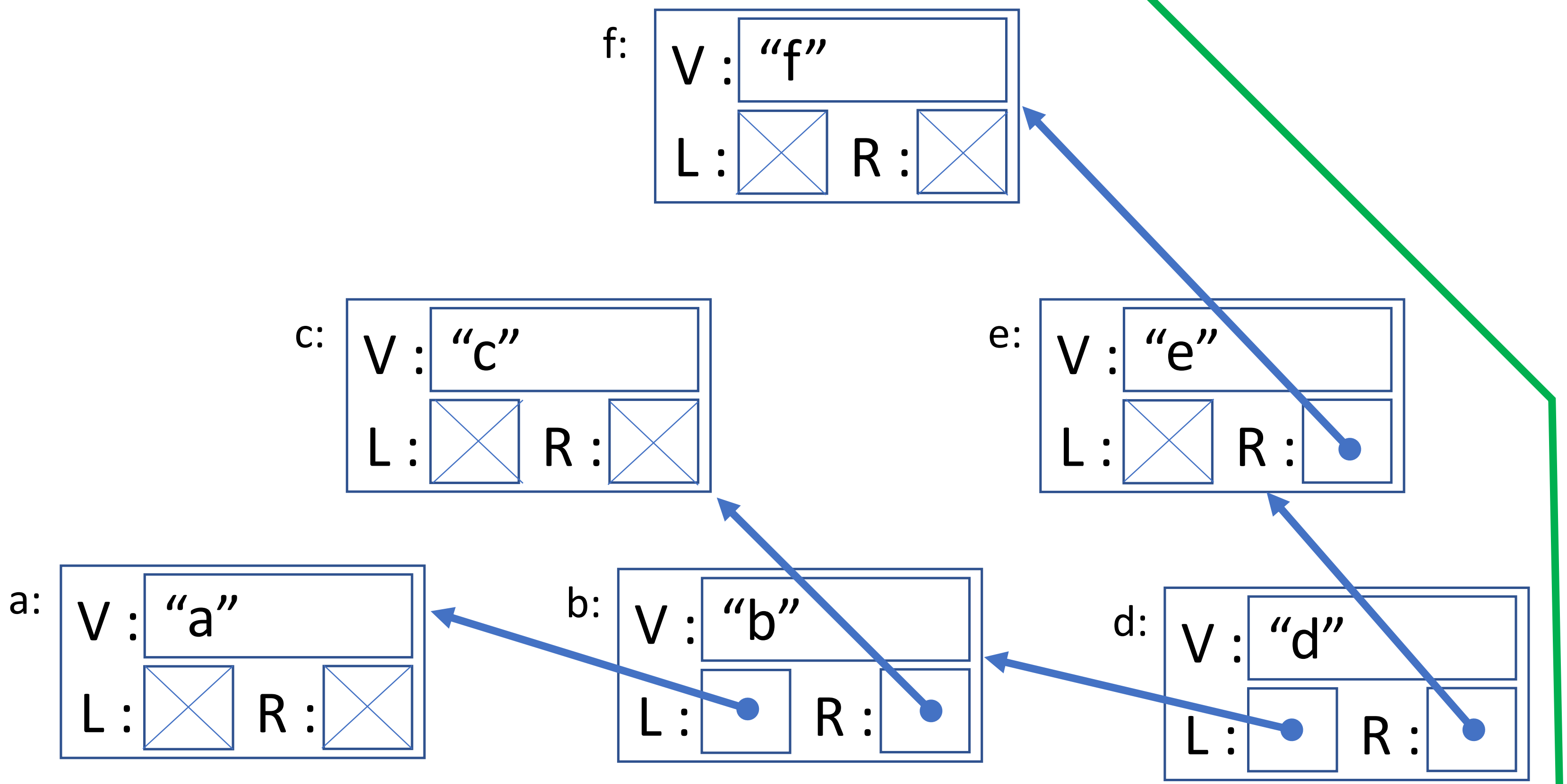

```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", &d ,     nullptr};
    my_tree f={"f", &e,      &b};
}
```



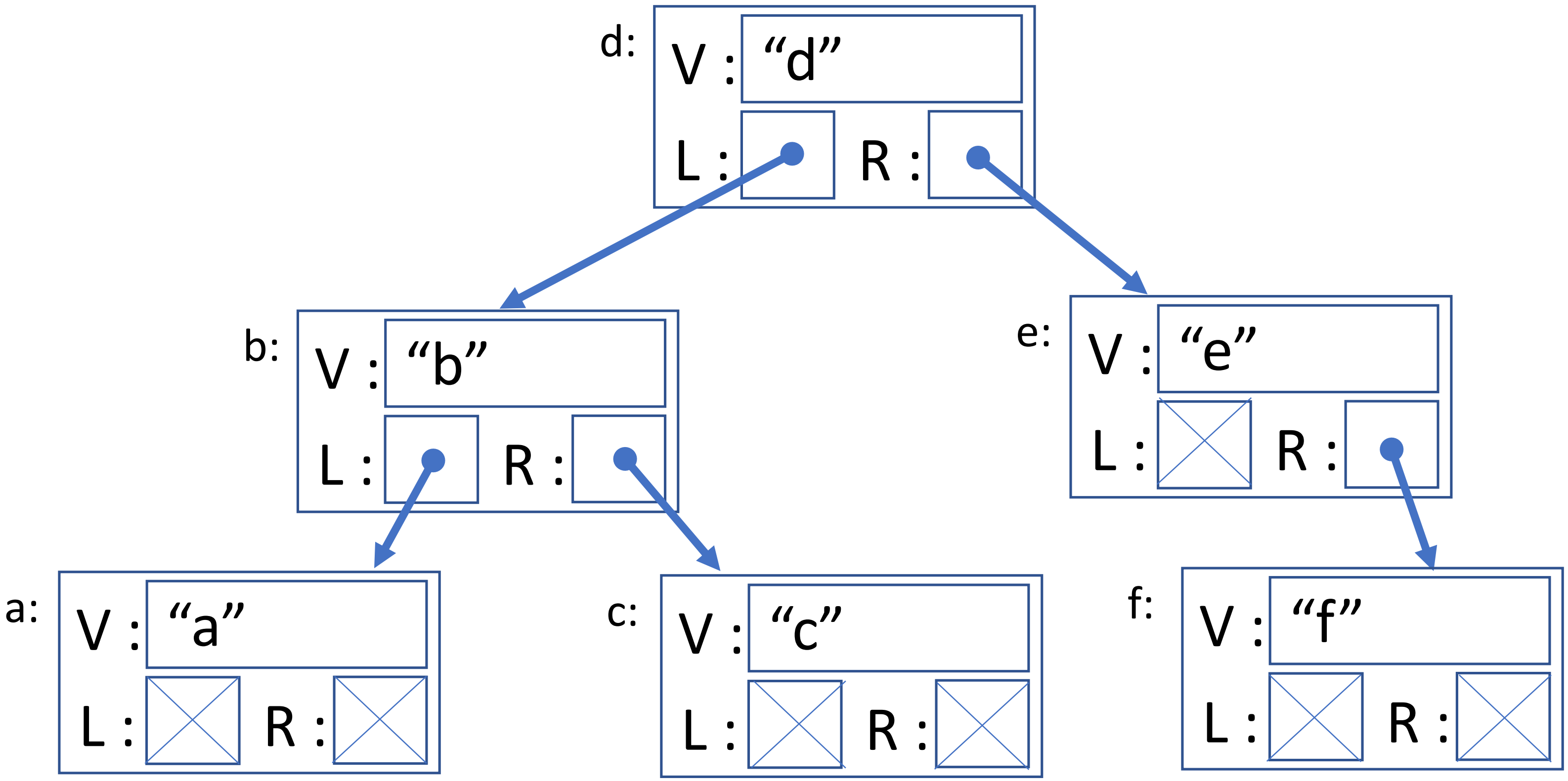

```
int main()
{
     my_tree a={"a", nullptr, nullptr};
     my_tree c={"c", nullptr, nullptr};
     my_tree b={"b", &a,      &c};
     my_tree d={"d", nullptr, nullptr};
     my_tree e={"e", &d ,     nullptr};
     my_tree f={"f", &e,      &b};
}
```




```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree d={"d", nullptr, nullptr};
    my_tree e={"e", &d ,     nullptr};
    my_tree f={"f", &e,      &b};
}
```




```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree f={"f", nullptr, nullptr};
    my_tree e={"e", nullptr, &f};
    my_tree d={"d", &b,      &e};
}
```




```
int main()
{
    my_tree a={"a", nullptr, nullptr};
    my_tree c={"c", nullptr, nullptr};
    my_tree b={"b", &a,      &c};
    my_tree f={"f", nullptr, nullptr};
    my_tree e={"e", nullptr, &f};
    my_tree d={"d", &b,      &e};
}
```

So… we have a binary search tree

- Manually creating a binary search tree is slow
  - Need to re-order and re-link trees to meet constraint
  - Quite an error-prone process

- It is better to preserve constraint throughout
  - Ensure the tree is always a binary search tree

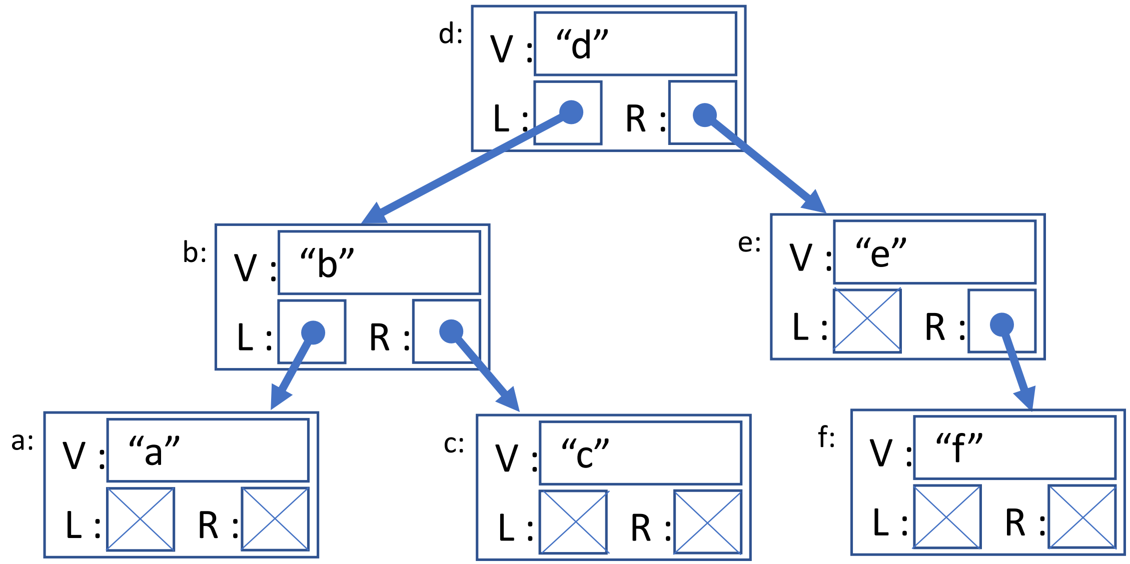
So what is it good for?

A binary ***search*** tree

- Binary search trees are good for searching

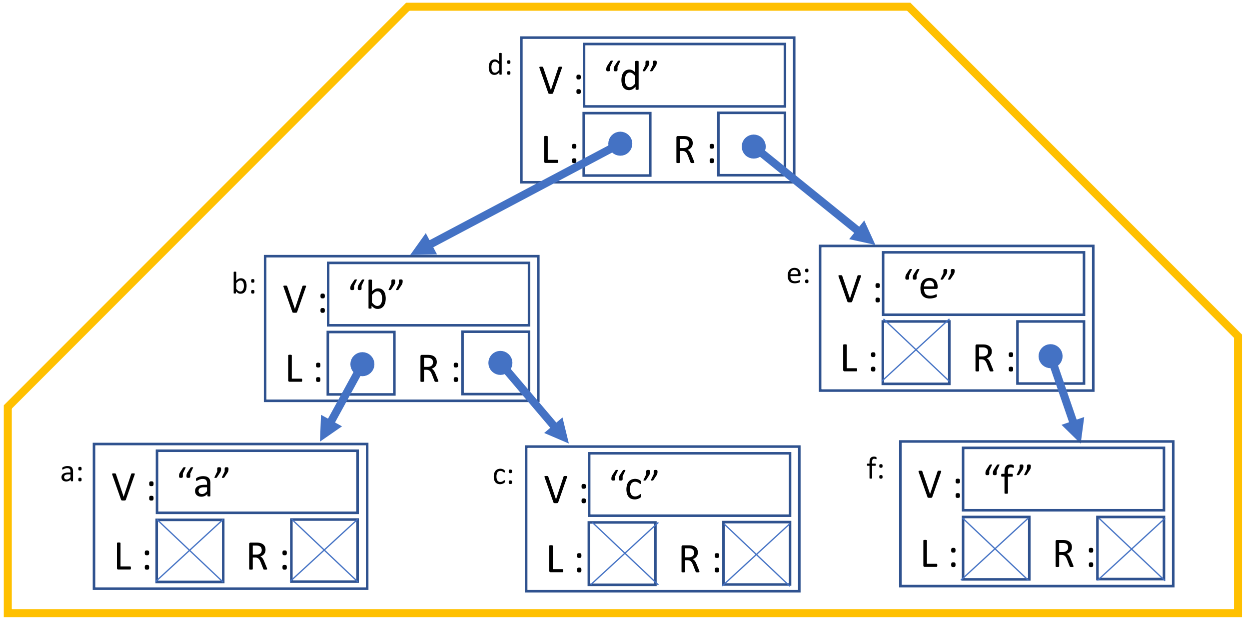
  *"Is this **value** present in the tree?"*

```
bool is_present(my_tree *t, string value)
{
    if(t==nullptr){
        return false;
    }else if(t->value == value){
        return true;
    }else if(value < t->value){
        return is_present(t->left, value);
    }else{
        return is_present(t->right, value);
    }
}
```



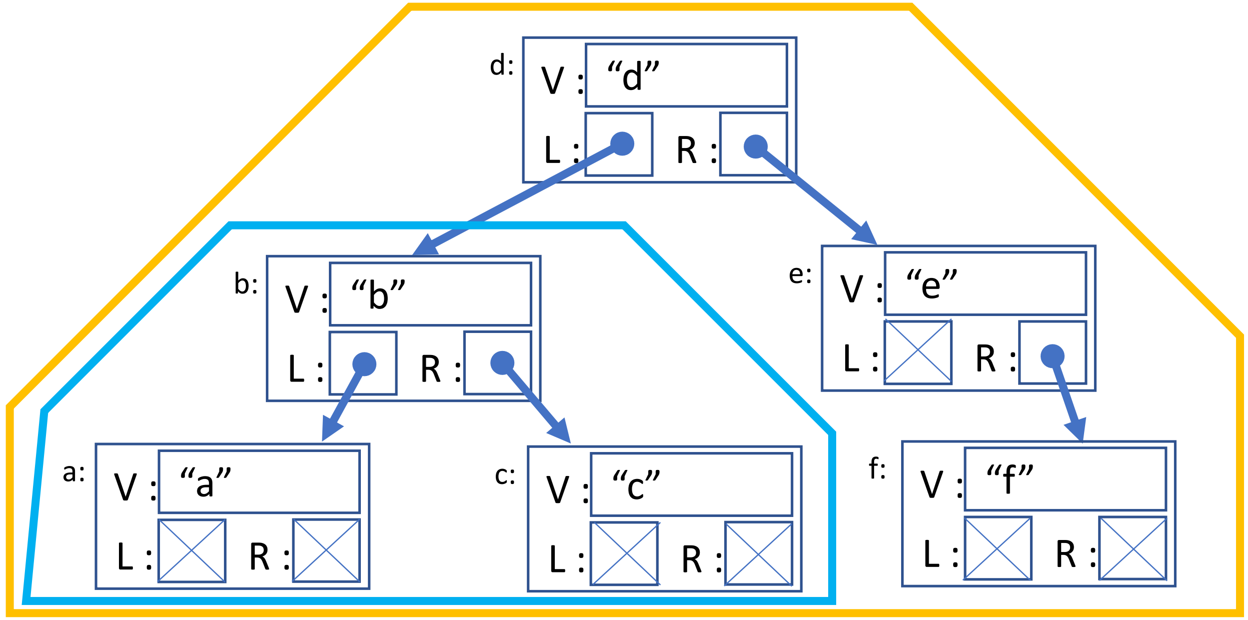

```
bool is_present(my_tree *t, string value)
{
    if(t==nullptr){
        return false;
    }else if(t->value == value){
        return true;
    }else if(value < t->value){
        return is_present(t->left, value);
    }else{
        return is_present(t->right, value);
    }
}
```

```
is_present(&d, "a")
```



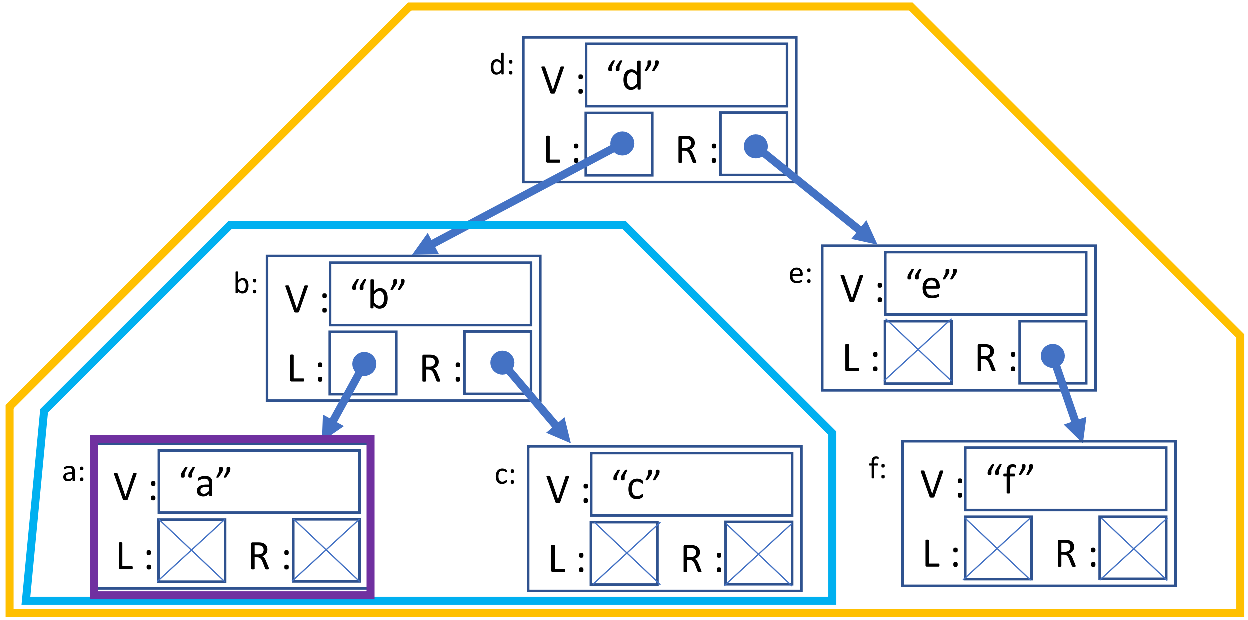

```
bool is_present(my_tree *t, string value)
{
    if(t==nullptr){
        return false;
    }else if(t->value == value){
        return true;
    }else if(value < t->value){
        return is_present(t->left, value);
    }else{
        return is_present(t->right, value);
    }
}
```

```
is_present(&d, “a”)

is_present(&b, “a”)
```




```
bool is_present(my_tree *t, string value)
{
    if(t==nullptr){
        return false;
    }else if(t->value == value){
        return true;
    }else if(value < t->value){
        return is_present(t->left, value);
    }else{
        return is_present(t->right, value);
    }
}
```

```
is_present(&d, "a")
is_present(&b, "a")
is_present(&a, "a")
```




```
bool is_present(my_tree *t, string value)
{
    if(t==nullptr){
        return false;
    }else if(t->value == value){
        return true;
    }else if(value < t->value){
        return is_present(t->left, value);
    }else{
        return is_present(t->right, value);
    }
}
```

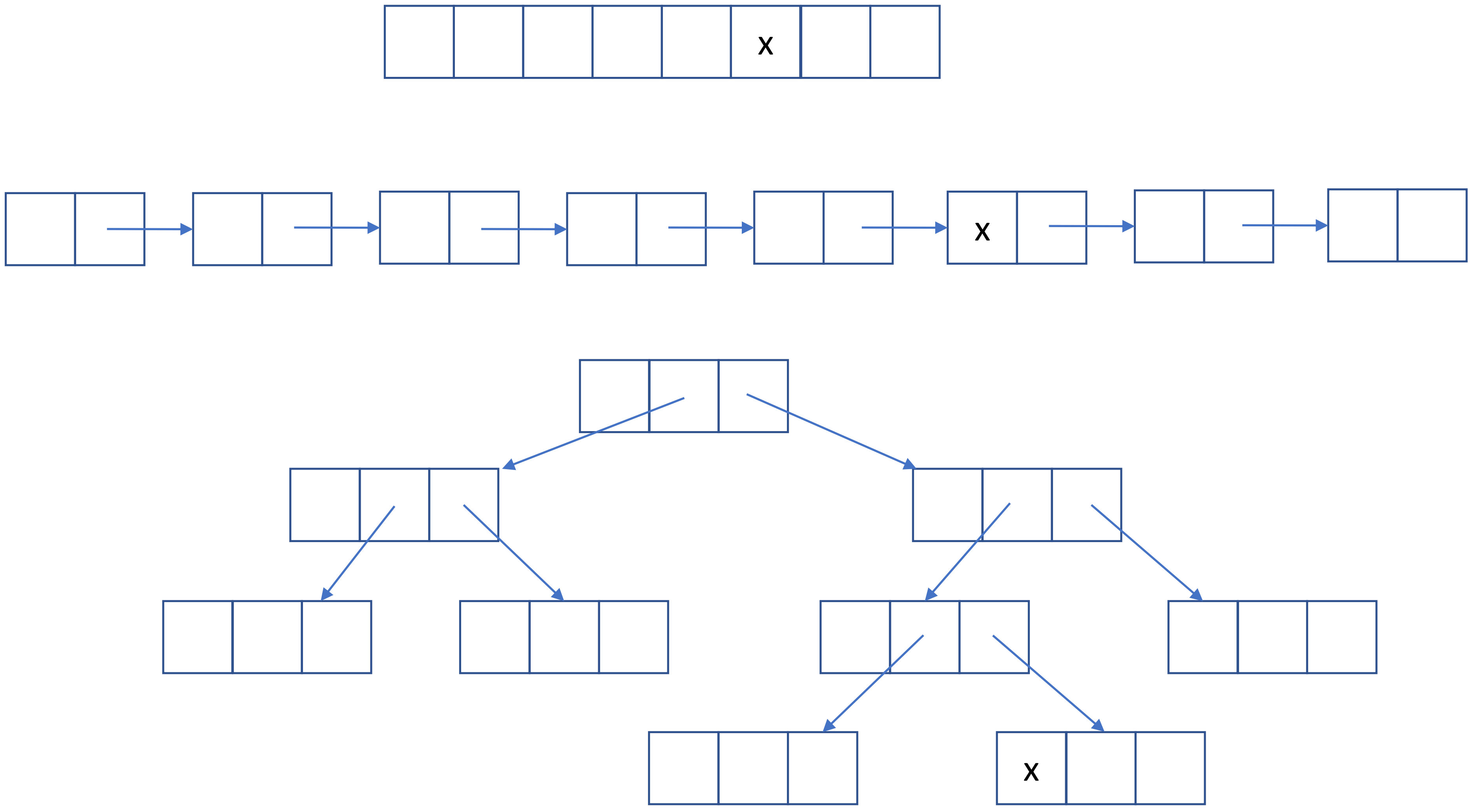
Why is this “good” for search?

```
bool is_present(vector<string> *v, string value)
{
    for(int i=0; i<v->size(); i++){
        if((*v)[i]==value){
            return true;
        }
    }
    return false;
}
```

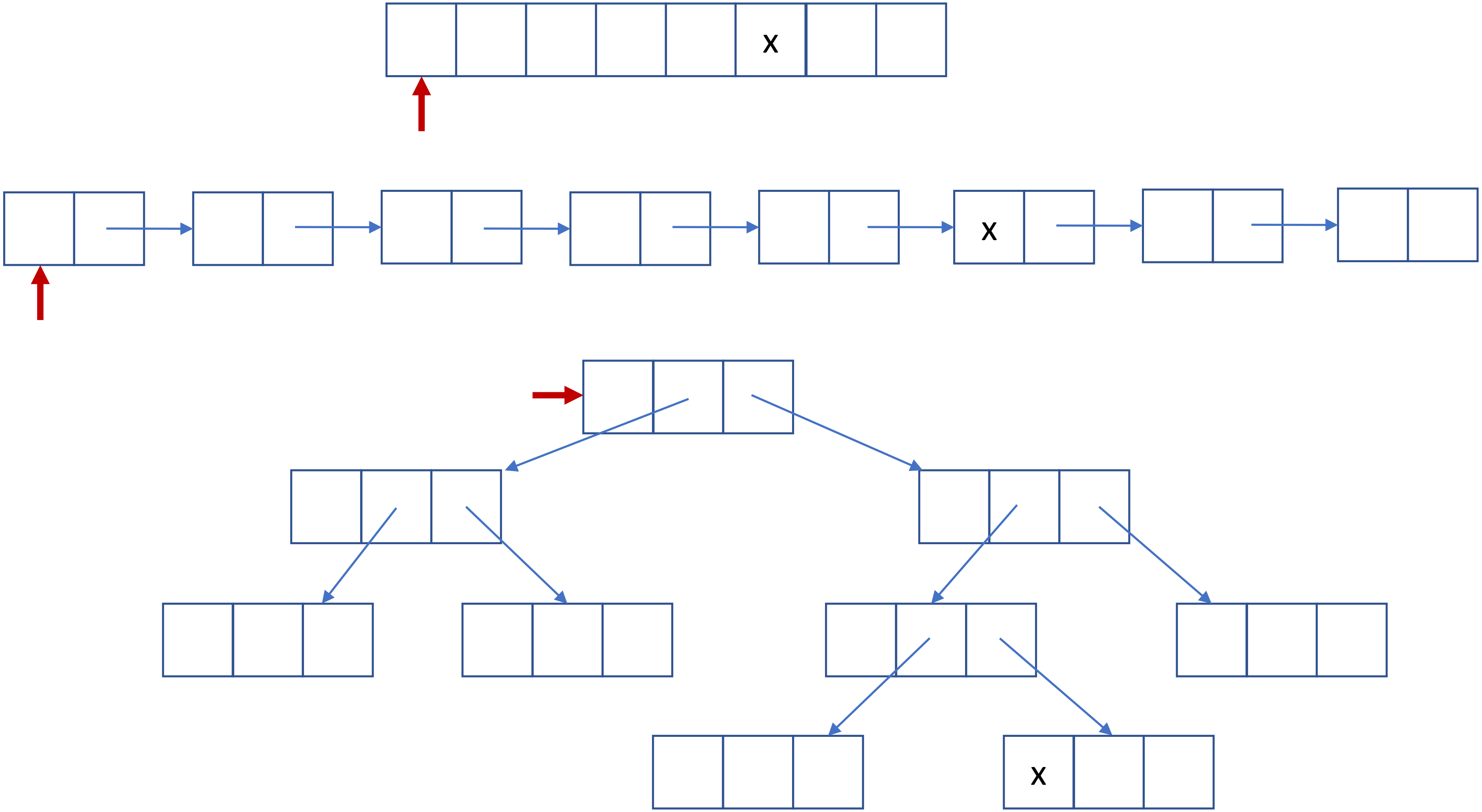
```
bool is_present(my_tree *t, string value)
{
    if(t==nullptr){
        return false;
    }else if(t->value == value){
        return true;
    }else if(value < t->value){
        return is_present(t->left, value);
    }else{
        return is_present(t->right, value);
    }
}
```

```
bool is_present(my_list *l, string value)
{
    if(l==nullptr){
        return false;
    }else if(l->value==value){
        return true;
    }else{
        return is_present(l->next, value);
    }
}
```

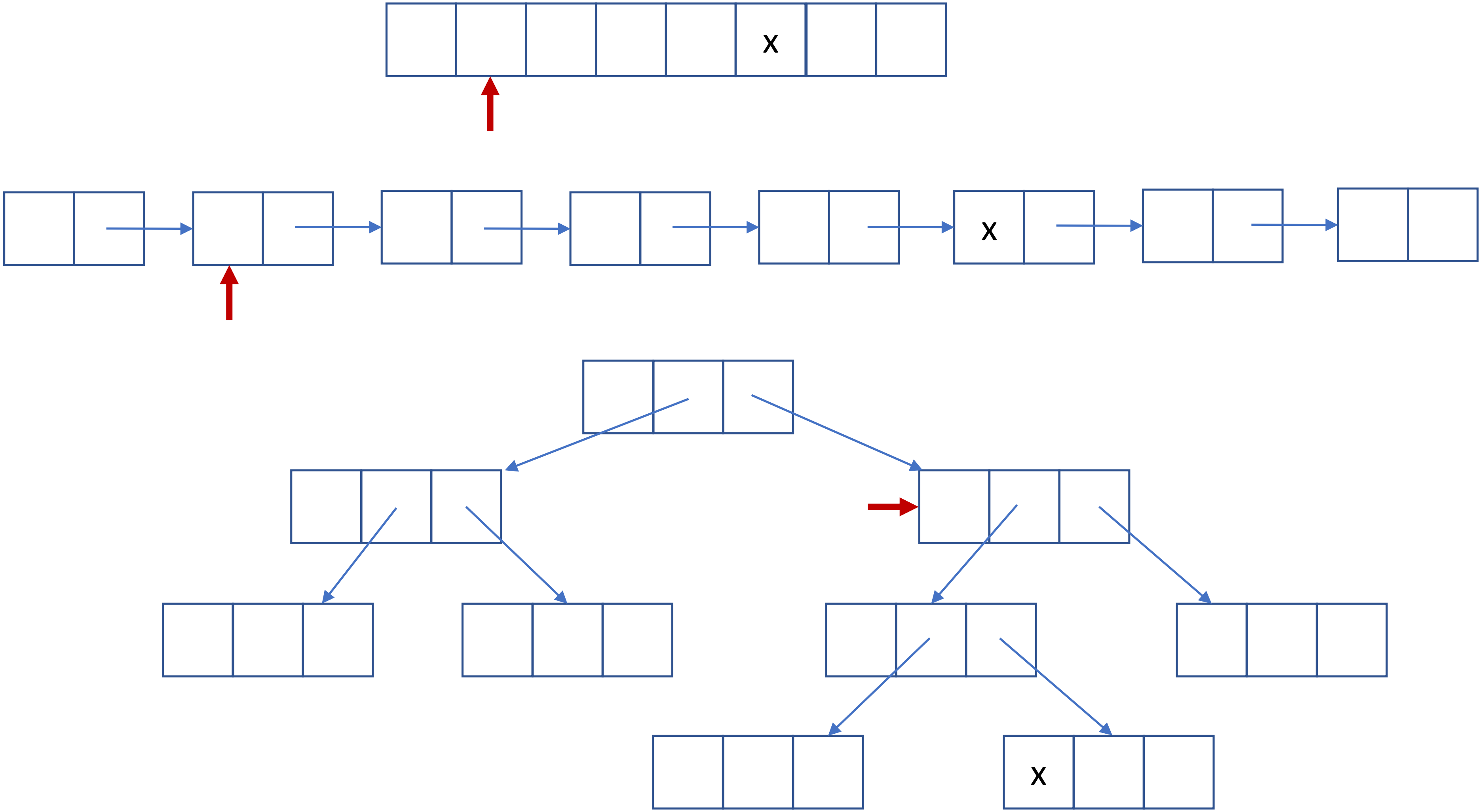
Tree vs List vs Vector



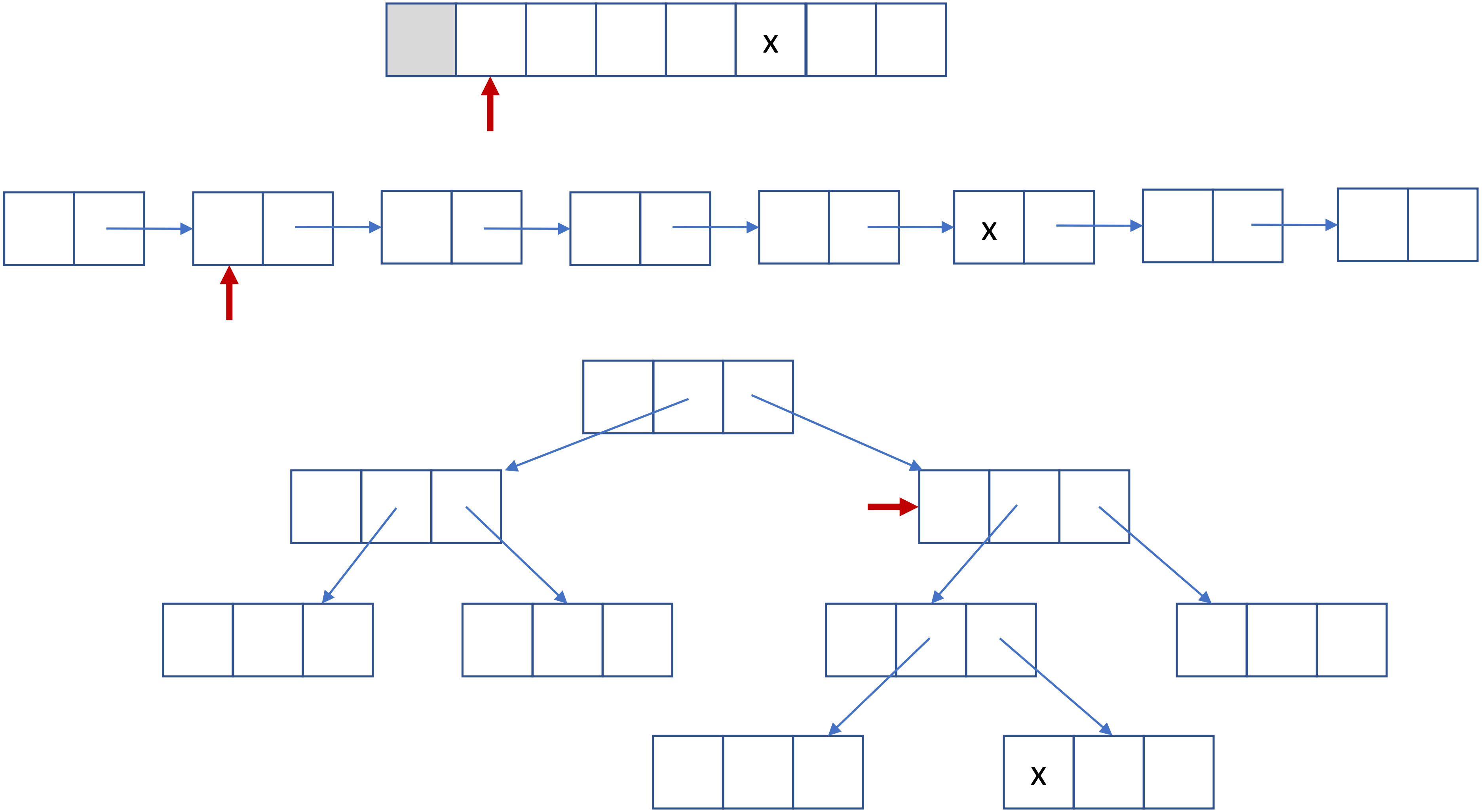
Tree vs List vs Vector



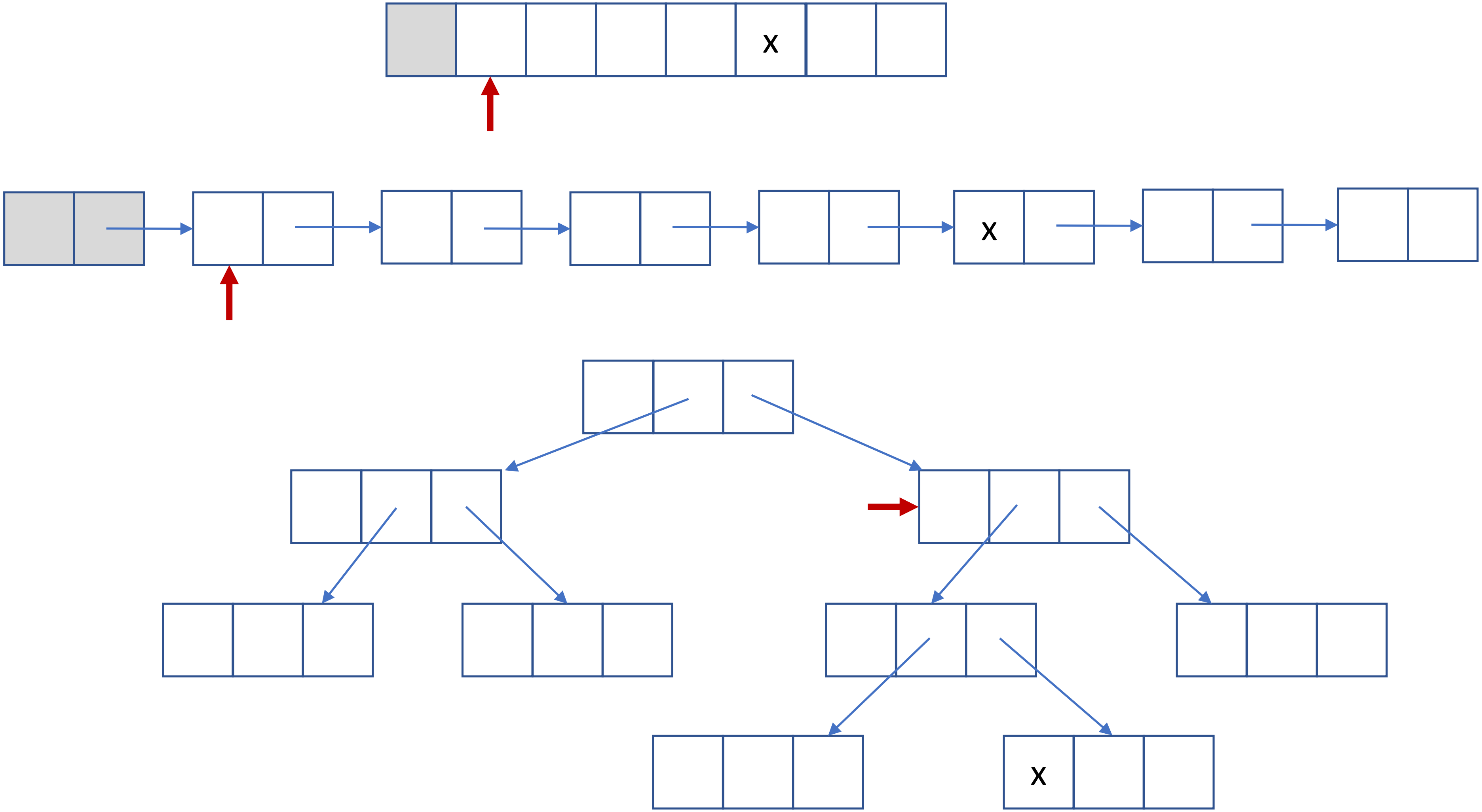
Tree vs List vs Vector



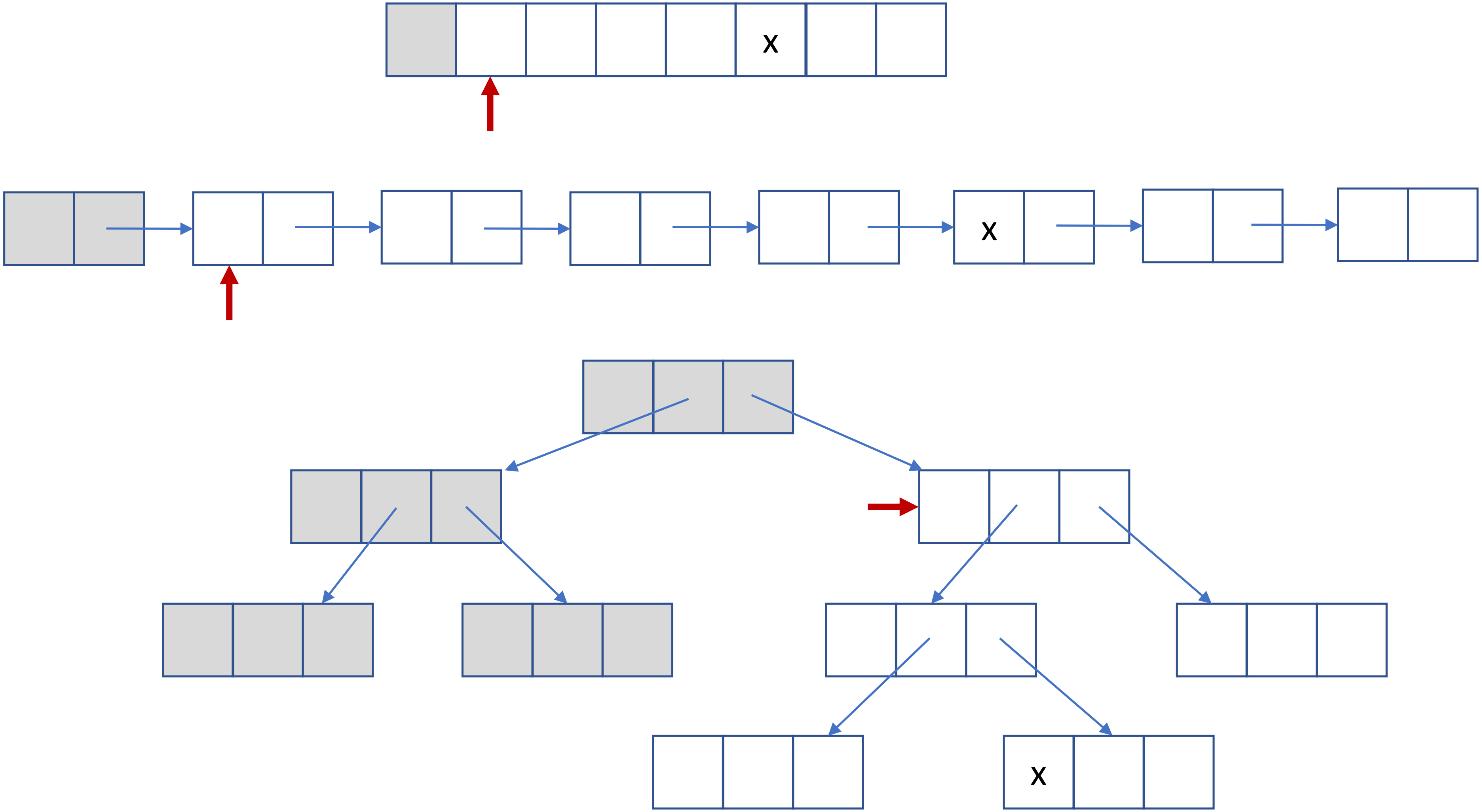
Tree vs List vs Vector



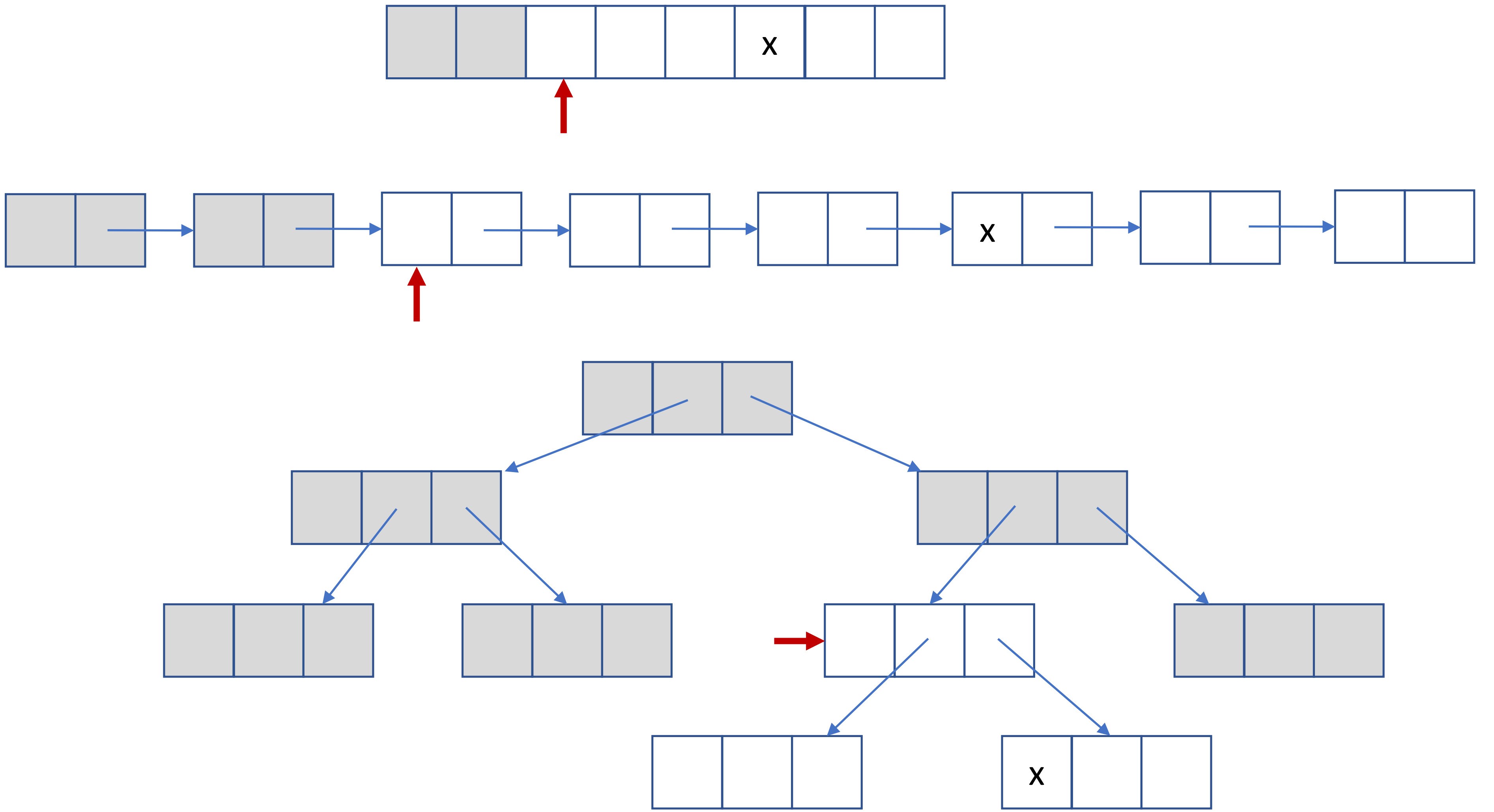
Tree vs List vs Vector



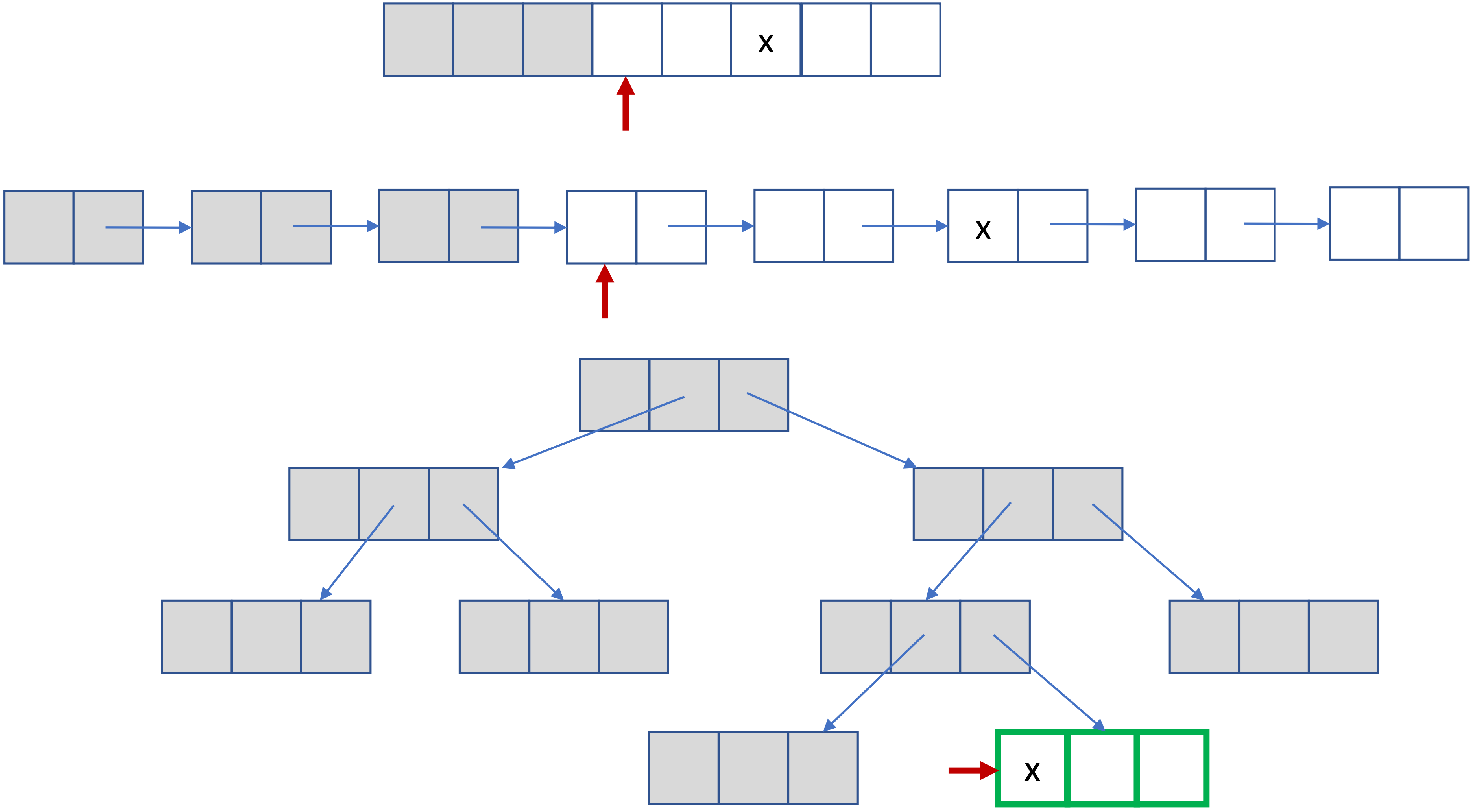
Tree vs List vs Vector



Tree vs List vs Vector



Tree vs List vs Vector



Asymptotic performance

- We care about ***large*** data structures
  - The performance for 10 data items is not important
  - What is the performance for 10^6 or 10^9 elements?

- Consider a data-structure with n elements

  What is the typical cost to find a value in "steps" ?

Vector:	~ n steps
List:	~ n steps
Binary tree:	~ $\log_2$ n steps

Tree operations

Maintaining constraints

- Checking if a tree is a binary search tree is expensive
  - Takes at least n steps to check ordering

- It is much more efficient to maintain constraints
  - Each tree function should ensure ordering

If the input is a valid binary search tree
then the output is a valid binary search tree

Inserting a node

```
// Before:  is_binary_search_tree(t)
//
//  t' = tree_insert(t, v)
//
// After:   is_binary_search_tree(t')
//          &&
//          is_present(t', v)
my_tree *tree_insert(my_tree *t, string value);
```

Inserting a node

```
my_tree *tree_insert(my_tree *t, string value)
{
    if(t==nullptr){                     // Case 1 : empty tree
        t=new my_tree; // Scalar new: see next slide
        t->value=value;
        t->left=nullptr;
        t->right=nullptr;

    }else if(value < t->value){  // Case 2 : insert on the left
        t->left = tree_insert(t->left, value););

    }else if(t->value == value){ // Case 3 :value already exists
        // Do nothing

    }else{                             // Case 4 : insert on the right
        t->right = tree_insert(t->right, value););
    }
    return t;
}
```

Inserting a node

```
my_tree *tree_insert(my_tree *t, string value)
{
    if(t==nullptr){                   // Case 1 : empty tree
        t=new my_tree; // Scalar new: see next slide
        t->value=value;
        t->left=nullptr;
        t->right=nullptr;

    }else if(value < t->value){  // Case 2 : insert on the left
        t->left = tree_insert(t->left, value);)

    }else if(t->value == value){ // Case 3 :value already exists
        // Do nothing

    }else{                            // Case 4 : insert on the right
        t->right = tree_insert(t->right, value);)
    }
    return t;
}
```

Inserting a node

```
my_tree *tree_insert(my_tree *t, string value)
{
    if(t==nullptr){                    // Case 1 : empty tree
        t=new my_tree; // Scalar new: see next slide
        t->value=value;
        t->left=nullptr;
        t->right=nullptr;

    }else if(value < t->value){  // Case 2 : insert on the left
        t->left = tree_insert(t->left, value););

    }else if(t->value == value){ // Case 3 :value already exists
        // Do nothing

    }else{                         // Case 4 : insert on the right
        t->right = tree_insert(t->right, value););
    }
    return t;
}
```

Inserting a node

```
my_tree *tree_insert(my_tree *t, string value)
{
    if(t==nullptr){                     // Case 1 : empty tree
        t=new my_tree; // Scalar new: see next slide
        t->value=value;
        t->left=nullptr;
        t->right=nullptr;

    }else if(value < t->value){  // Case 2 : insert on the left
        t->left = tree_insert(t->left, value););

    }else if(t->value == value){ // Case 3 :value already exists
        // Do nothing

    }else{                       // Case 4 : insert on the right
        t->right = tree_insert(t->right, value););
    }
    return t;
}
```

Scalar new

So far we have seen vector new[]

Allocate n contiguous instances of T

```
T *a = new T[n];
delete[] a;
```

There is also "scalar" new

Allocate exactly 1 instance of T

```
T *p = new T;
delete p;
```

You can use either for single node allocation, but the delete form must match the new form

De-allocating a tree

```
void tree_delete(my_tree *t)
{
    if(t!=nullptr){
        tree_delete(t->left);
        tree_delete(t->right);
        delete t;
    }
}
```

Difficult operations

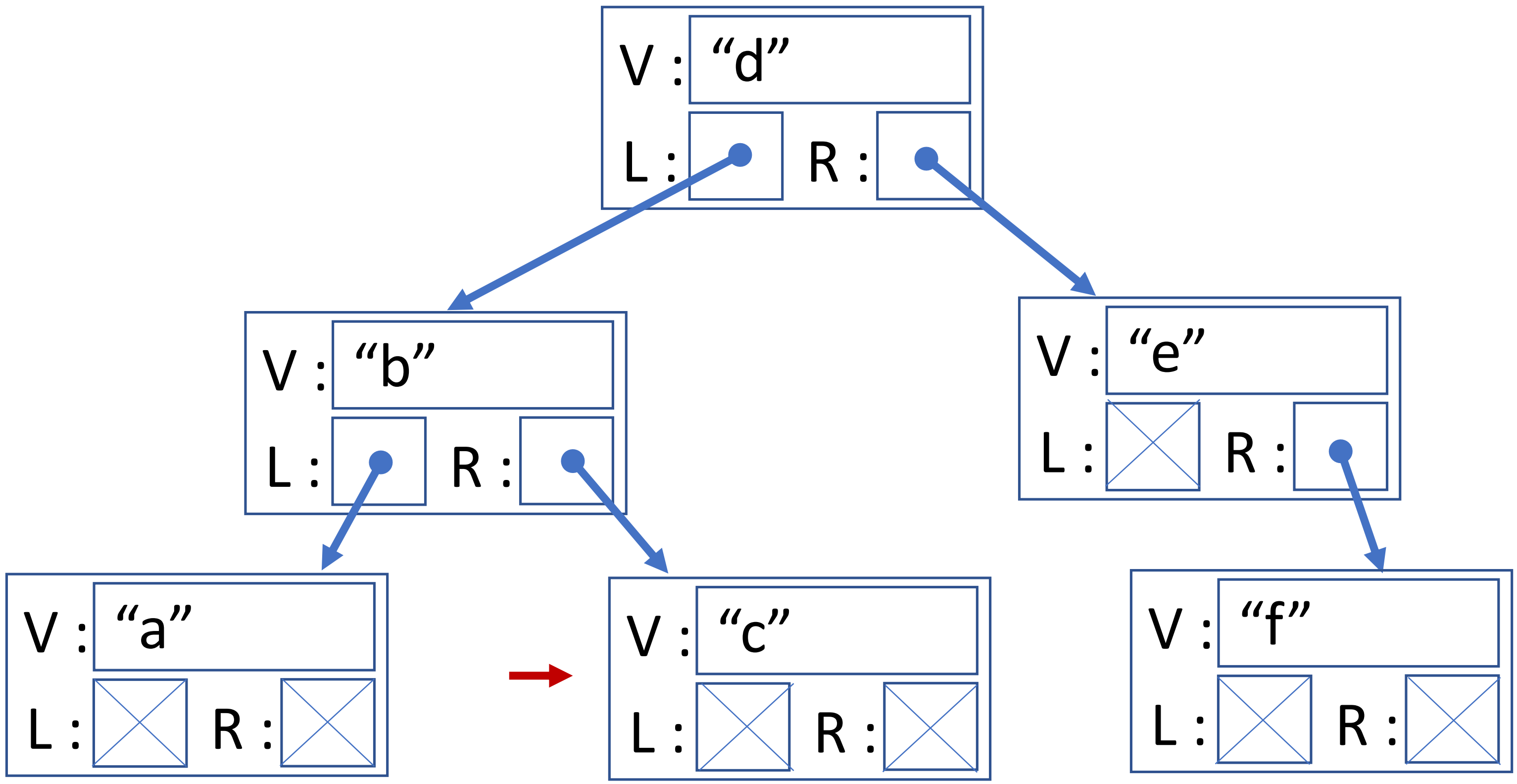
1. Removing a node
   - Must preserve the ordering constraints
2. Keeping the tree balanced
   - It’s only fast if the depth is small

Removing a node

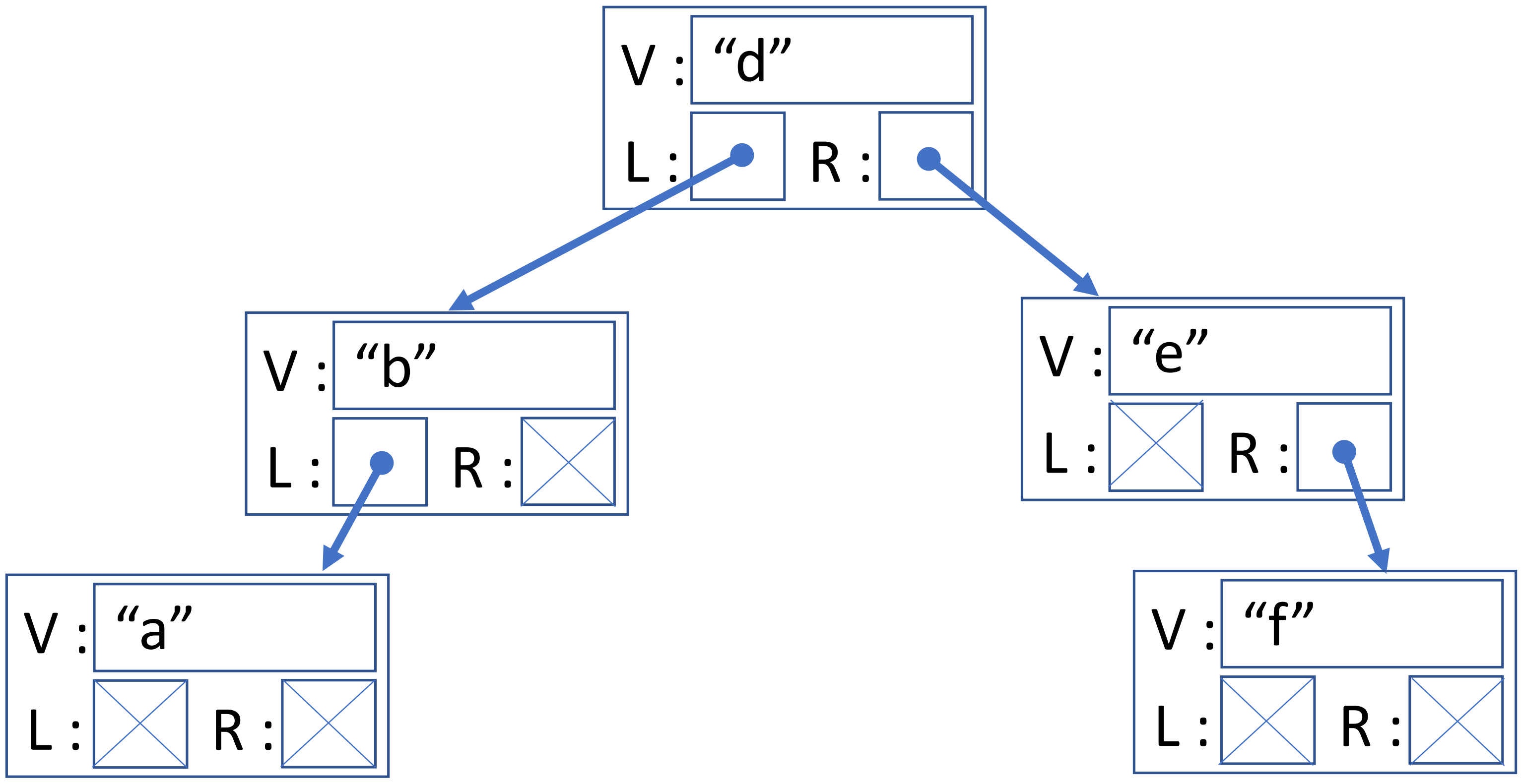
```
// Before:  is_binary_search_tree(t)
//
//  t' = tree_delete(t, v)
//
// After:   is_binary_search_tree(t')
//          && !is_present(t', v)
my_tree *tree_delete(my_tree *t, string value);
```

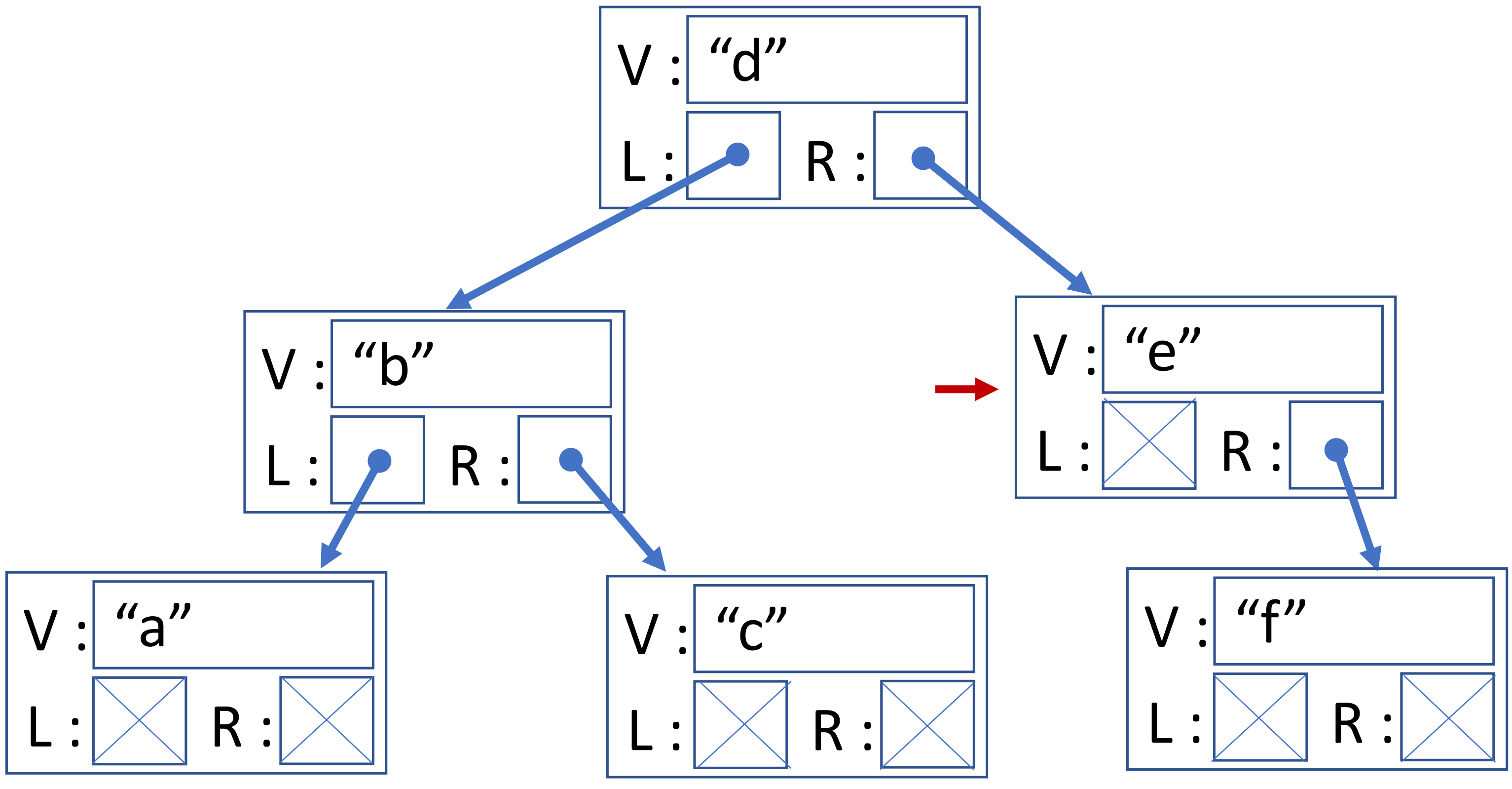
Removing a node

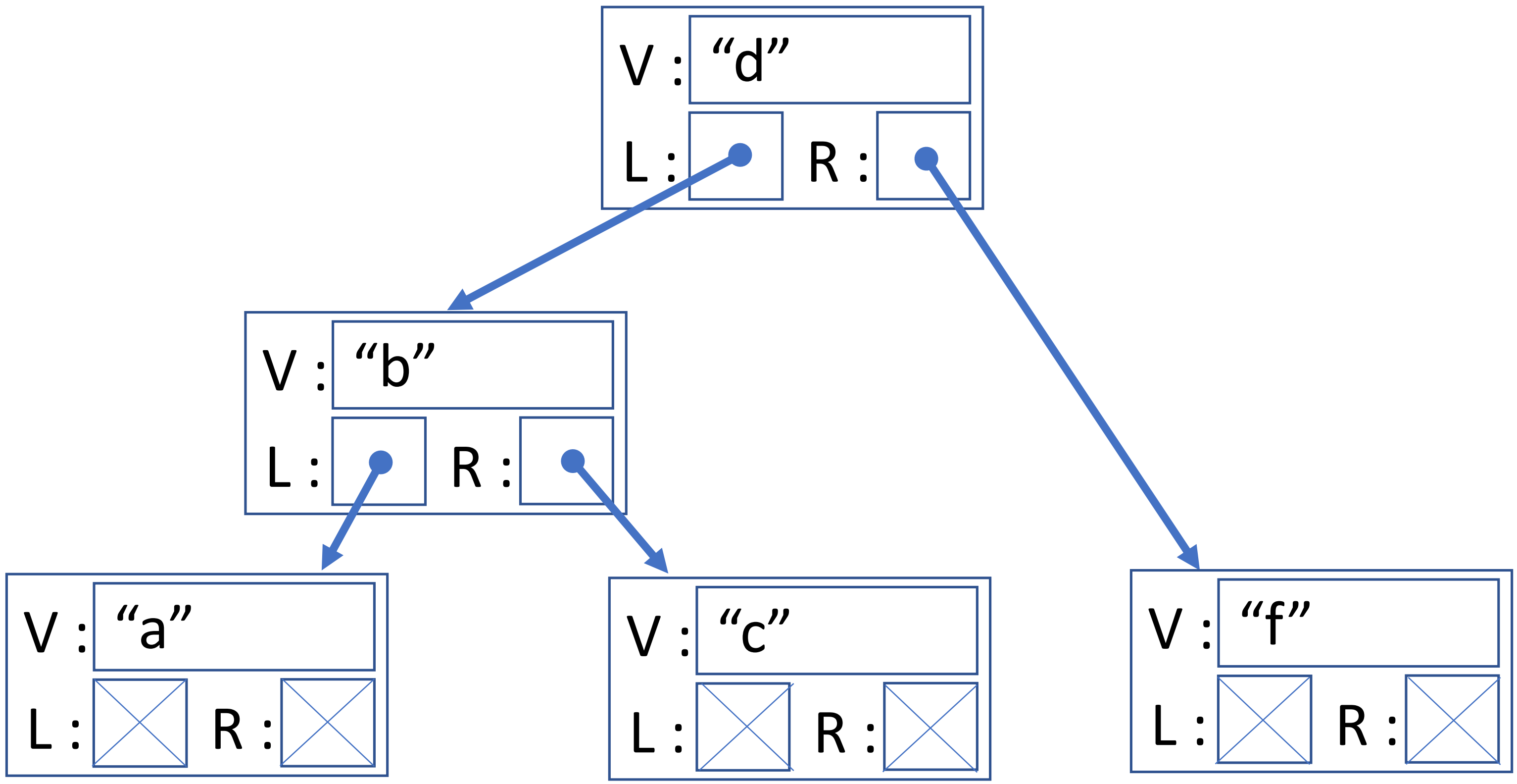
```
my_tree *tree_delete(my_tree *t, string value)
{
  if(t==nullptr){                // Case 1: do nothing

  }else if(value < value){    // Case 1: Delete from left tree
    t->left = tree_delete(t->left, value);

  }else if(t->value == value){  // Case 3: Delete this node
    //
    // TODO : What do we do here?
    //

  }else{                         // Case 4: Delete from right tree
    t->right = tree_delete(t->right, value);
  }
  return t;
}
```



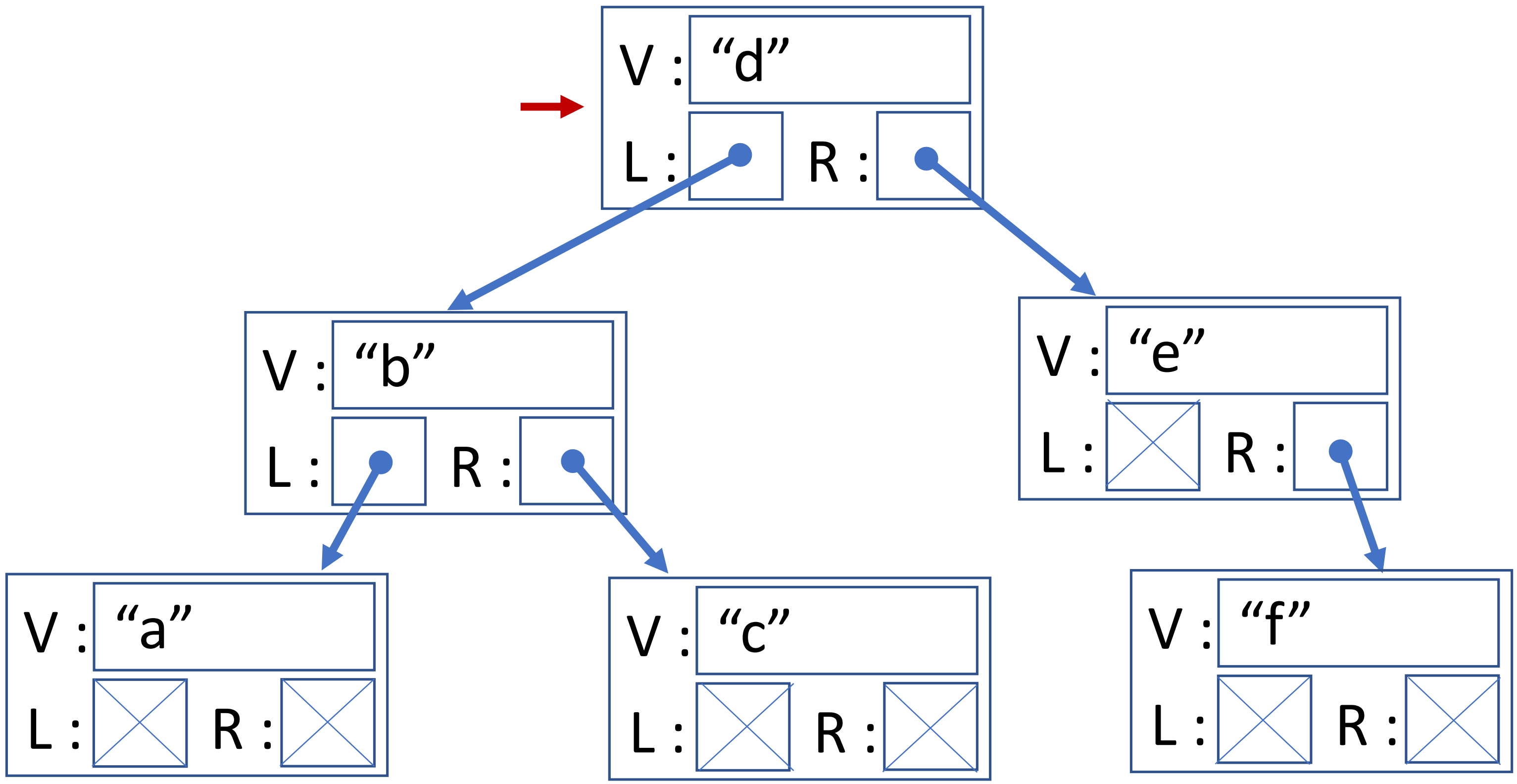
V : "d"
L :
R :
V : "b"
L :
R :
V : "e"
L :
R :
V : "a"
L :
R :
V : "c"
L :
R :
V : "f"
L :
R :

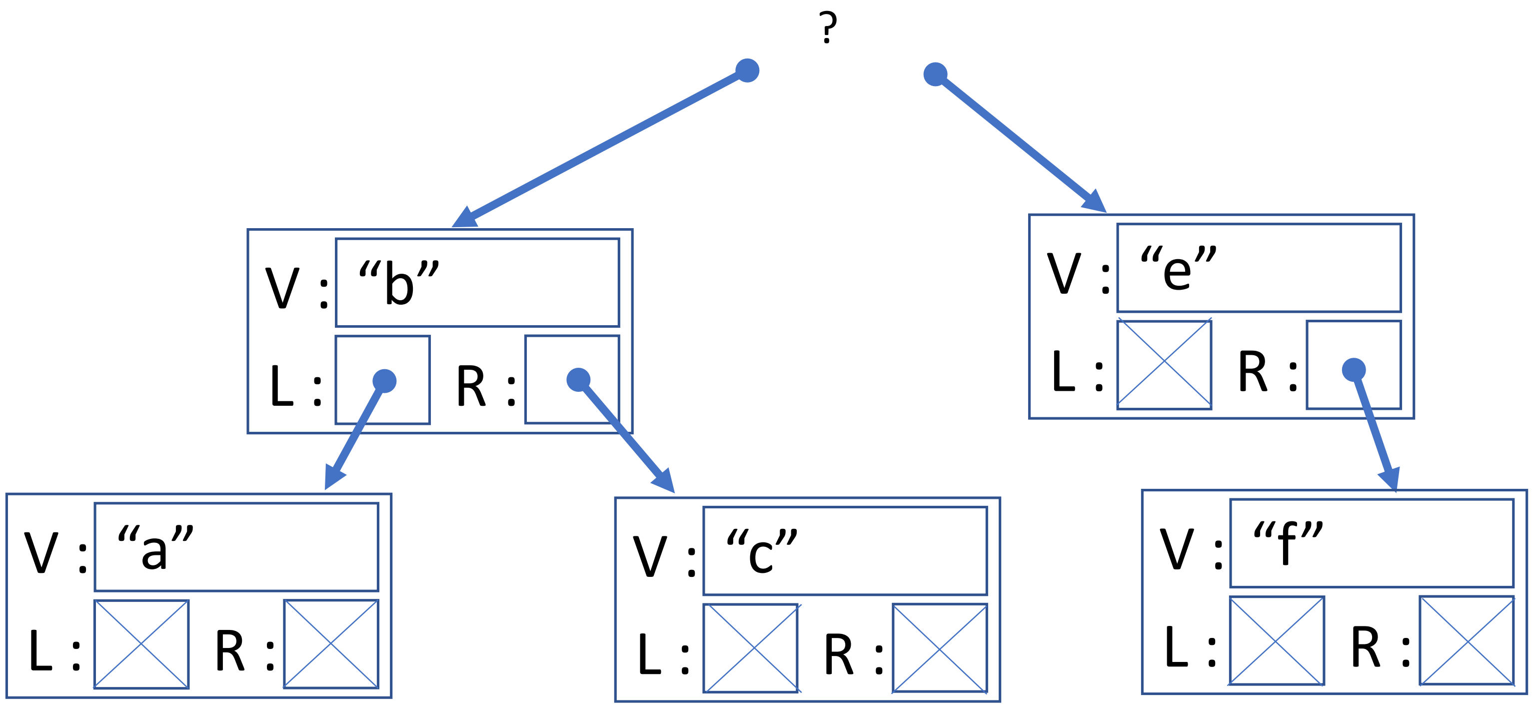

V : “d”
L :
R :
V : “b”
L :
R :
V : “e”
L :
R :
V : “a”
L :
R :
V : “f”
L :
R :


V : "d"
L :
R :
V : "b"
L :
R :
V : "e"
L :
R :
V : "a"
L :
R :
V : "c"
L :
R :
V : "f"
L :
R :



V : “d”
L :
R :
V : “b”
L :
R :
V : “a”
L :
R :
V : “c”
L :
R :
V : “f”
L :
R :


V : "d"
L :
R :
V : "b"
L :
R :
V : "e"
L :
R :
V : "a"
L :
R :
V : "c"
L :
R :
V : "f"
L :
R :



We’ll come back to this case

Difficult operations

1. Removing a node
   - Must preserve the ordering constraints
2. Keeping the tree balanced
   - It’s only fast if the depth is small

We’ll come back to these when we look at invariants

set<T> and map<K,V>

Trees as sets

Our tree provides the following operations

- Test if a value is in the tree
- Add a value to the tree
- Remove a value from the tree

Plus an additional guarantee

- A value can only exist once in the tree

This is a great model for a mathematical set

`set<T>`

The standard library provides `set<T>`

```
set<int> s;
s.insert( 5 );
s.insert( 10 );
```

Unfortunately it does not have `contains`
we need "iterators"; covered next term

Set is mainly useful for discrete math algorithms

Trees as maps

We can augment our node to create a map
- A “key” : used to preserve order
- A “value” : arbitrary value associated with the key

It is now a “map” from a key to a value

```
struct my_tree
{
    string key;
    string value;
    my_tree *left;
    my_tree *right;
};
```

`map<K,V>`

The standard library provides `map<K,V>`

```cpp
map<string,int> m;

m["hello"] = 4;
m["x"] = 10;

int x = m["x"];
assert(x==10);
```

`map<K,V>` is more generally useful

Using `map<K,V>`

To use `map<K,V>` you need two types

K : A type to use as the key, or index
V : A type to use as the value associated with a key

The map is implemented using a tree, so

The type K must be *comparable*

```
K a, b;
bool eq = a==b;
bool lt = a<b;
```

Where next?

- We’re reaching the limit of low-level stuff
  - Need some higher-level C++ features to use map + set
- We’re reaching the end of term
  - Need to recap
- We’ve reached a point of understanding
  - What exactly is the final coursework?