Lecture 15
Binary Search Trees

15-122: Principles of Imperative Computation (Fall 2024)
Frank Pfenning, André Platzer, Rob Simmons, Iliano Cervesato

In this lecture, we will continue considering ways to implement the dictionary (or associative array) interface. This time, we will implement this interface with *binary search trees*. We will eventually be able to achieve $O(\log n)$ worst-case asymptotic complexity for insert and lookup. This also extends to delete, although we won't discuss that operation in lecture.

Additional Resources

- Review slides (`https://cs.cmu.edu/~15122/handouts/slides/review/15-bst.pdf`)
- Code for this lecture (`https://cs.cmu.edu/~15122/handouts/code/15-bst.tgz`)

This fits as follows with respect to our learning goals:

Computational Thinking: We discover binary trees as a way to organize information. We superimpose to them the notion of sortedness, which we examined in the past, as a way to obtain exponential speedups.

Algorithms and Data Structures: We present binary search trees as a space-efficient and extensible data structure with a potentially logarithmic complexity for many operations of interest — we will see in the next lecture how to guarantee this bound.

Programming: We define a type for binary trees and use recursion as a convenient approach to implement specification functions and operations on them.

1 Ordered Collections

Hash dictionaries organize entries in an array at indices that are determined from their key using a hash function. If the hash function is good, this means that the entries will be placed at a reasonably random position spread out across the whole array. If it is bad, linear search is needed to locate the entry.

There are many alternative ways of implementing dictionaries. For example, we could have stored the entries in an array, sorted by key. Then lookup by binary search would have been $O(\log n)$, but insertion would be $O(n)$, because it takes $O(\log n)$ steps to find the right place, but then $O(n)$ steps to make room for that new entry by shifting all elements with a bigger key over. (We would also need to grow the array as in unbounded arrays to make sure it does not run out of capacity.) Arrays are not flexible enough for fast insertion, but the data structure that we will be devising in this lecture will be.

2 Abstract Binary Search

What are the operations that we needed to be able to perform binary search? We needed a way of comparing the key we were looking for with the key of a given entry in our data structure. Depending on the result of that comparison, binary search returns the position of that entry if they were the same, advances to the left if what we are looking for is smaller, or advances to the right if what we are looking for is bigger. For binary search to work with the complexity $O(\log n)$, it was important that binary search advances to the left or right *many steps at once*, not just by one element. Indeed, if we would follow the abstract binary search principle starting from the middle of the array but advancing only by one index in the array, we would obtain linear search, which has complexity $O(n)$, not $O(\log n)$.

Thus, binary search needs a way of comparing keys and a way of advancing through the elements of the data structure very quickly, either to the left (towards entries with smaller keys) or to the right (towards bigger ones). In the array-based binary search we've studied, each iteration calculates a midpoint

```
int mid = lo + (hi - lo) / 2;
```

and a new bound for the next iteration is (if the key we're searching for is smaller than that of the entry at `mid`)

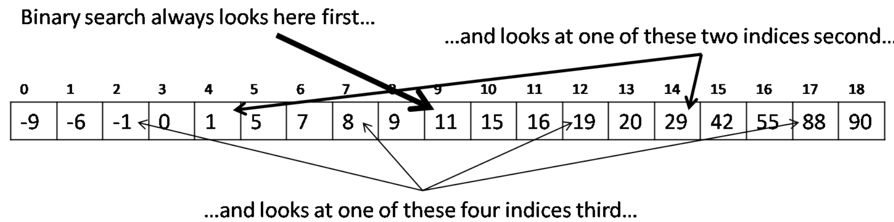
```
hi = mid;
```

or (if the key is larger)

```
lo = mid + 1;
```

Therefore, we know that the next value `mid` will be either `(lo + mid) / 2` or `((mid + 1) + hi) / 2` (ignoring the possibility of overflow).

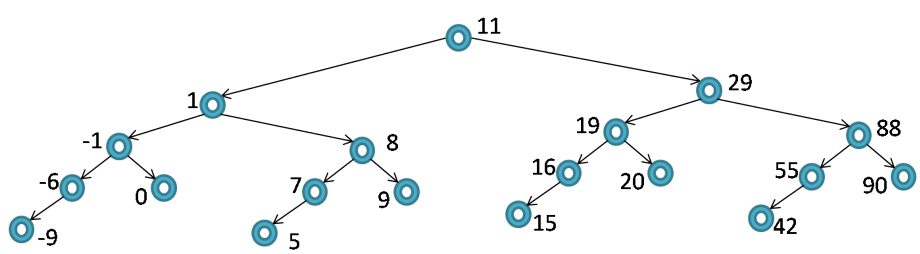
This pattern continues, and given any sorted array, we can enumerate all possible binary searches:



This pattern means that constant-time access to an array element at an *arbitrary* index isn't necessary for doing binary search! To do binary search on the array above, all we need is constant time access from array index 9 (containing 11) to array indices 4 and 14 (containing 1 and 29, respectively), constant time access from array index 4 to array indices 2 and 7, and so on. At each point in binary search, we know that our search will proceed in one of at most two ways, so we will explicitly represent those choices with a pointer structure, giving us the structure of a *binary tree*. The tree structure that we got from running binary search on this array...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-9	-6	-1	0	1	5	7	8	9	11	15	16	19	20	29	42	55	88	90

...corresponds to this binary tree:



3 Representing Binary Trees with Pointers

To represent a binary tree using pointers, we use a struct with two pointers: one to the left child and one to the right child. If there is no child, the pointer is `NULL`. A leaf of the tree is a node with two `NULL` pointers.

```
typedef struct tree_node tree;
struct tree_node {
  entry data;       // Non NULL
```

```
  tree* left;
  tree* right;
};
```

Rather than the fully generic data implementation that we used for hash tables, we'll assume for the sake of simplicity that the client is providing us with two types, a pointer type `entry` corresponding to entries and a type `key` for their keys, and with two functions, `entry_key` that returns the key of an entry and `key_compare` that compares two keys:

```
/* Client-side interface */
// typedef ______  key;
// typedef ______* entry;

key entry_key(entry e)
/*@requires e != NULL; @*/ ;

int key_compare(key k1, key k2) ;
```

We require that valid values of type `entry` be non-`NULL`. As usual, our implementation of dictionaries based on trees will use `NULL` to signal that an `entry` is not there.

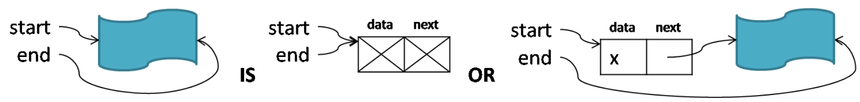
The function `key_compare` provided by the client is different from the equivalence function we used for hash tables. For binary search trees, we need to compare keys k_1 and k_2 and determine if $k_1 < k_2$, or $k_1 = k_2$, or $k_1 > k_2$. A common approach to this is for a comparison function to return an integer r, where $r < 0$ means $k_1 < k_2$, $r = 0$ means $k_1 = k_2$, and $r > 0$ means $k_1 > k_2$.

Trees are the second *recursive* data structure we've seen: a tree node has two fields that contain pointers to tree nodes. Thus far we've only seen recursive data structures as linked lists, either chains in a hash table or list segments in a stack or a queue.

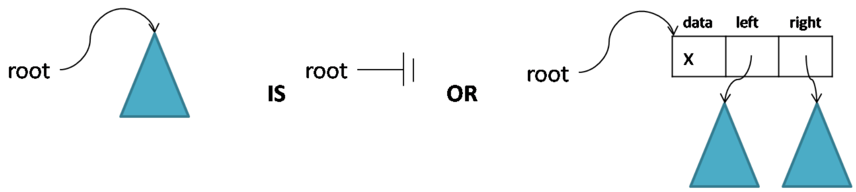
Let's remember how we picture list segments. Any list segment is referred to by two pointers, `start` and `end`, and there are two possibilities for how this list can be constructed, both of which require `start` to be non-`NULL` (and `start->data` also to satisfy our constraints on `entry` values).

```
bool is_segment(list* start, list* end) {
  if (start == NULL) return false;
  if (start->data == NULL) return false;
  if (start == end) return true;
  return is_segment(start->next, end);
}
```

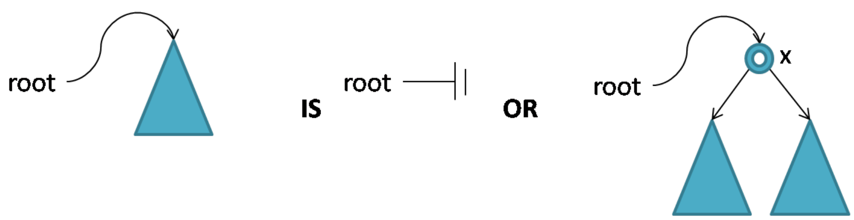
We can represent these choices graphically by using a picture like  to represent an arbitrary segment. Then we know every segment has one or two forms:



We'll create a similar picture for trees: the tree containing no entries is `NULL`, and a non-empty tree is a struct with three fields: the data and the `left` and `right` pointers, which are themselves trees.



Rather than drawing out the `tree_node` struct with its three fields explicitly, we'll usually use a more graph-like way of presenting trees:



This recursive definition can be directly encoded into a very simple data structure invariant `is_tree`. It checks very little: just that all the `data` fields are non-`NULL`, as the client interface requires. If it terminates, it also ensures that there are no cycles; a cycle would cause non-termination, just as it would with `is_segment`.

```
bool is_tree(tree* root) {
  if (root == NULL) return true;
  return root->data != NULL
      && is_tree(root->left) && is_tree(root->right);
}
```

3.1 The Ordering Invariant

Binary search was only correct for arrays if the array was sorted. Only then do we know that it is okay not to look at the upper half of the array if the element we are looking for is smaller than the middle element, because, in a sorted array, it can then only occur in the lower half, if at all. For binary search to work correctly on binary search trees, we, thus, need to maintain a corresponding data structure invariant: all entries to the right of a node have keys that are bigger than the key of that node. And all the nodes to the left of that node have smaller keys than the key at that node. This *ordering invariant* is a core idea of binary search trees; it's what makes a binary tree into a binary *search* tree.

> **Ordering Invariant.** At any node with key k in a binary search tree, the key of all entries in the left subtree is strictly less than k, while the key of all entries in the right subtree is strictly greater than k.

This invariant implies that no key occurs more than once in a tree, and we have to make sure our insertion function maintains this invariant.

We won't write code for checking the ordering invariant just yet, as that turns out to be surprisingly difficult. We'll first discuss the lookup and insertion functions for binary search trees. As we carry out this discussion, we will assume we have a function `is_bst(T)` that incorporates `is_tree` seen earlier and the ordering invariant. We will implement `is_bst` later in this lecture.

4 Searching for a Key

The ordering invariant lets us find an entry with key k in a binary search tree the same way we found an entry with binary search, just on the more abstract tree data structure. Here is a recursive algorithm for search, starting at the root of the tree:

1. If the tree is empty, stop.
2. Compare the key k' of the current node to k. Stop if equal.
3. If k is smaller than k', proceed to the left child.
4. If k is larger than k', proceed to the right child.

The implementation of this search captures the informal description above. Recall that `entry_key(e)` extracts the key component of entry `e` and that `key_compare(k1,k2)` returns `-1` if $k_1 < k_2$, `0` if $k_1 = k_2$, and `1` if $k_1 > k_2$.

```
entry bst_lookup(tree* T, key k)
/*@requires is_bst(T); @*/
/*@ensures \result == NULL
        || key_compare(entry_key(\result), k) == 0; @*/
{
  if (T == NULL) return NULL;

  int cmp = key_compare(k, entry_key(T->data));
  if (cmp == 0)  return T->data;
  if (cmp <  0)  return bst_lookup(T->left, k);
  //@assert cmp > 0;
  return bst_lookup(T->right, k);
}
```

We chose here a recursive implementation, following the structure of a tree, but in practice an iterative version may also be a reasonable alternative (see Exercise 3). We also chose to return not a Boolean but either the entry itself if it matches the key `k` given in input, and `NULL` otherwise. In this way, we can use our burgeoning binary search tree support both to implement dictionaries and sets — for a set, the types `key` and `entry` are the same and the function `entry_key` simply returns its argument.

5 Complexity

If our binary search tree were perfectly balanced, that is, had the same number of nodes on the left as on the right for every subtree, then the ordering invariant would ensure that search for an entry with a given key has asymptotic complexity $O(\log n)$, where n is the number of entries in the tree. Every time we compare the key `k` with the root of a perfectly balanced tree, we either stop or throw out half the entries in the tree.

In general we can say that the cost of lookup is $O(h)$, where h is the *height* of the tree. We will define height to be the maximum number of nodes that can be reached by any sequence of pointers starting at the root. An empty tree has height 0, and a tree with two children has the maximum height of either child, plus 1.

6 The Interface

Before we talk about insertion into a binary search tree, we should recall the interface of dictionaries and discuss how we will implement it. Remember that we're assuming a client definition of types `entry` and `key`, and functions `entry_key` and `key_compare`, rather than the fully generic version using void pointers and function pointers.

```
/* Library interface */
// typedef ______* dict_t;

dict_t dict_new()
/*@ensures \result != NULL; @*/ ;

entry dict_lookup(dict_t D, key k)
/*@requires D != NULL; @*/
/*@ensures \result == NULL
        || key_compare(entry_key(\result), k) == 0; @*/ ;

void dict_insert(dict_t D, entry e)
/*@requires D != NULL && e != NULL; @*/
/*@ensures dict_lookup(D, entry_key(e)) != NULL; @*/ ;
```

We can't define the type `dict_t` to be `tree*`, for two reasons. One reason is that a new tree should be empty, but an empty tree is represented by the pointer `NULL`, which would violate the `dict_new` postcondition. More fundamentally, if `NULL` was the representation of an empty dictionary, there would be no way to write a function to imperatively insert additional entries. This is because a function call makes copies of the (small) values passed as arguments.

The usual solution here is one we have already used for stacks, queues, and hash tables: we have a *header* which in this case just consists of a pointer to the root of the tree. We often keep other information associated with the data structure in these headers, such as the size.

```
struct dict_header {
  tree* root;
};
typedef struct dict_header dict;

bool is_dict(dict* D) {
  return D != NULL && is_bst(D->root);
}
```

Lookup in a dictionary then just calls the recursive function we've already defined:

```
entry dict_lookup(dict* D, key k)
/*@requires is_dict(D); @*/
/*@ensures \result == NULL
```

```
        || key_compare(entry_key(\result), k) == 0; @*/
{
  return bst_lookup(D->root, k);
}
```

The relationship between the functions `is_dict` and `is_bst` and between `dict_lookup` and `bst_lookup` is a common one. The non-recursive function `is_dict` is given the non-recursive struct `dict_header`, and then calls the recursive helper function `is_bst` on the recursive structure of tree nodes.

7 Inserting an Entry

With the header structure, it is straightforward to implement `bst_insert`. We just proceed as if we were looking for the given entry. If we find a node with the same key, we just overwrite its `data` field. Otherwise, we insert the new entry in the place where it would have been, had it been there in the first place. This last clause, however, creates a small difficulty. When we hit a null pointer (which indicates the key was not already in the tree), we cannot replace what it points to (it doesn't point to anything!). Instead, we *return* the new tree so that the parent can modify itself.

```
tree* bst_insert(tree* T, entry e)
//@requires is_bst(T) && e != NULL;
//@ensures is_bst(\result) && \result != NULL;
//@ensures bst_lookup(\result, entry_key(e)) == e;
{
  if (T == NULL) {
    /* create new node and return it */
    tree* R  = alloc(tree);
    R->data  = e;
    R->left  = NULL;  // Not required (initialized to NULL)
    R->right = NULL;  // Not required (initialized to NULL)
    return R;
  }

  int cmp = key_compare(entry_key(e), entry_key(T->data));
  if (cmp == 0)     T->data = e;
  else if (cmp < 0) T->left = bst_insert(T->left, e);
  else {
    //@assert cmp > 0;
    T->right = bst_insert(T->right, e);
  }
  return T;
}
```

The returned subtree will also be stored as the new root:

```
void dict_insert(dict* D, entry e)
//@requires is_dict(D) && e != NULL;
/*@ensures  is_dict(D)
        && dict_lookup(D, entry_key(e)) != NULL; @*/
{
  D->root = bst_insert(D->root, e);
}
```

8 Checking the Ordering Invariant

When we analyze the structure of the recursive functions implementing search and insert, we are tempted to try defining a simple, *but wrong!* ordering invariant for binary trees as follows: tree T is ordered whenever

1. T is empty, or
2. T has key k at the root, T_L as left subtree and T_R as right subtree, and

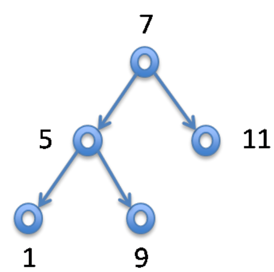
- T_L is empty, or T_L's key is less than k and T_L is ordered; and
- T_R is empty, or T_R's key is greater than k and T_R is ordered.

This would yield the following code:

```
/* THIS CODE IS BUGGY */
bool is_ordered(tree* T) {
  if (T == NULL) return true;  /* an empty tree is a BST */
  key k = entry_key(T->data);
  return (T->left == NULL
          || key_compare(entry_key(T->left->data), k) < 0)
      && (T->right == NULL
          || key_compare(k, entry_key(T->right->data)) < 0)
      && is_ordered(T->left)
      && is_ordered(T->right);
}
```

While this should always be true for a binary search tree, it is far weaker than the ordering invariant stated at the beginning of lecture. Before reading on, you should check your understanding of that invariant to exhibit a tree that would satisfy the above code, but violate the ordering invariant.

There is actually more than one problem with this. The most glaring one is that following tree would pass this test:



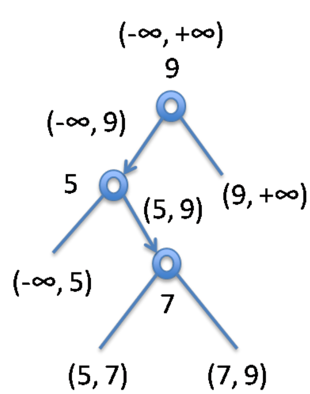
Even though, locally, the key of the left node is always smaller and on the right is always bigger, the node with key 9 is in the wrong place and we would not find it with our search algorithm since we would look in the right subtree of the root.

An alternative way of thinking about the invariant is as follows. Assume we are at a node with key k.

1. If we go to the *left* subtree, we establish an *upper bound* on the keys in the subtree: they must all be smaller than k.

2. If we go to the *right* subtree, we establish a *lower bound* on the keys in the subtree: they must all be larger than k.

The general idea then is to traverse the tree recursively, and pass down an interval with lower and upper bounds for all the keys in the tree. The following diagram illustrates this idea on a tree with integer entries. We start at the root with an unrestricted interval, allowing any key, which is written as $(-\infty, +\infty)$. As usual in mathematics we write intervals as $(x, z) = \{y \mid x < y \text{ and } y < z\}$. At the leaves we write the interval for the subtree. For example, if there were a left subtree of the node with key 7, all of its keys would have to be in the interval $(5, 7)$.



The only difficulty in implementing this idea is the unbounded intervals, written above as $-\infty$ and $+\infty$. Here is one possibility: we pass not just the key value, but the particular entry from which we can extract the key that bounds the tree. Since `entry` must be a pointer type, this allows us to pass `NULL` in case there is no lower or upper bound.

```
bool is_ordered(tree* T, entry lo, entry hi)
//@requires is_tree(T);
{
  if (T == NULL) return true;
  key k = entry_key(T->data);
  return T->data != NULL
      && (lo == NULL || key_compare(entry_key(lo), k) < 0)
      && (hi == NULL || key_compare(k, entry_key(hi)) < 0)
      && is_ordered(T->left, lo, T->data)
      && is_ordered(T->right, T->data, hi);
}
```

We can then combine our earlier (and admittedly minimal) `is_tree` and `is_ordered` into a function that checks whether a given tree is a binary search tree, and using it we can define a representation invariant function for dictionaries implemented as binary search trees:

```
bool is_bst(tree* T) {
  return is_tree(T) && is_ordered(T, NULL, NULL);
}

bool is_dict(dict* D) {
  return D != NULL && is_bst(D->root);
}
```

A word of caution: the call to `is_ordered(T, NULL, NULL)` embedded in the pre- and post-condition of the function `bst_insert` is actually not strong enough to prove the correctness of the recursive function. A similar remark applies to `bst_lookup`. This is because of the missing information of the bounds. We will return to this issue later in the course.

9 BST Sets

As we saw when talking about hash tables, a set is a dictionary whose keys and entries are the same thing — they are the elements of the set. Therefore, just like with hash tables, we can base an implementation of sets on binary search trees. The interface a BST set library is as follows:

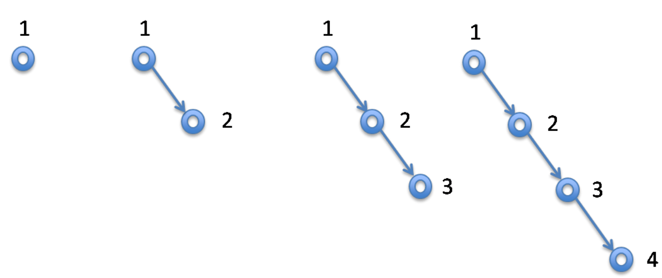
```
/* Client-side interface */
// typedef ______  elem;

int elem_compare(elem e1, elem e2)
/*@ensures -1 <= \result && \result <= 1; @*/ ;


/* Library interface */
// typedef ______* set_t;

set_t set_new()
/*@ensures \result != NULL; @*/ ;

bool set_contains(set_t S, elem e)
/*@requires S != NULL; @*/ ;

void set_insert(set_t S, elem e)
/*@requires S != NULL; @*/
/*@ensures set_contains(S, e); @*/ ;
```

Because lookup (now a set membership test) returns a boolean, there is no reason for elements to be pointers.

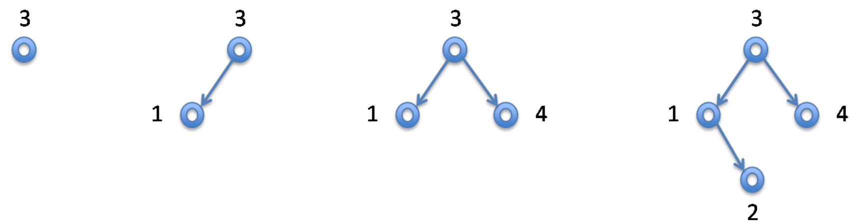
10 The Shape of Binary Search Trees

We have already mentioned that balanced binary search trees have good properties, such as logarithmic time for insertion and search. The question is if binary search trees will be balanced. This depends on the order of insertion. Consider the insertion of numbers 1, 2, 3, and 4.

If we insert them in increasing order we obtain the following trees in sequence.



Similarly, if we insert them in decreasing order we get a straight line, always going to the left. If we instead insert in the order 3, 1, 4, 2, we obtain the following sequence of binary search trees:



Clearly, the last tree is much more balanced. In the extreme, if we insert entries with their keys in order, or reverse order, the tree will be linear, and search time will be $O(n)$ for n items.

These observations mean that it is extremely important to pay attention to the balance of the tree. We will discuss ways to keep binary search trees balanced in the next lecture.

11 Exercises

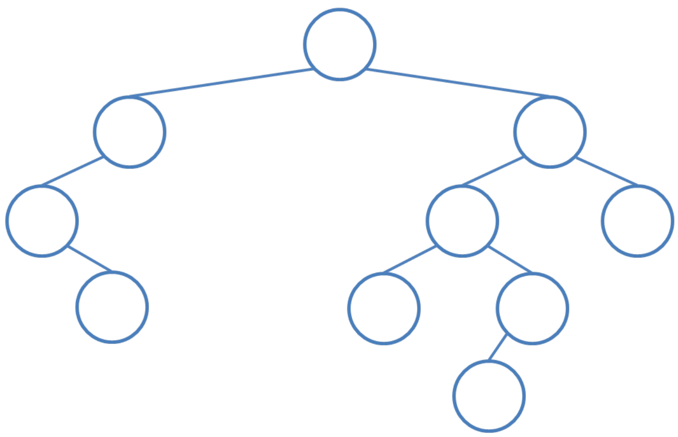
Exercise 1 (sample solution on page 17). *Draw the binary search tree resulting from inserting the following numbers in the given order starting from an empty tree.*

24, 33, 10, 2, 1, 7, 6

Exercise 2 (sample solution on page 17). *Inserting the keys*

2, 8, 10, 11, 12, 13, 14, 15, 17, 20

in some order into an initially empty tree yields a binary search tree with the following shape:



Fill in each node with the value that goes into it and write down a possible insertion sequence that would result into this tree.

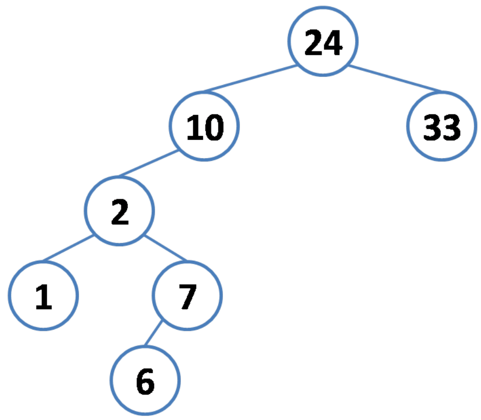
Exercise 3 (sample solution on page 18). *Rewrite* `dict_lookup` *to be iterative rather than rely on the recursive helper function* `bst_lookup`*.*

Exercise 4 (sample solution on page 18). *Rewrite* `dict_insert` *to be iterative rather than rely on the recursive helper function* `bst_insert`*.*
*[**Hint:** The difficulty will be to update the pointers in the parents when we replace a node that is* `NULL`*. For that purpose we can keep a "trailing" pointer which should be the parent of the node currently under consideration.]*

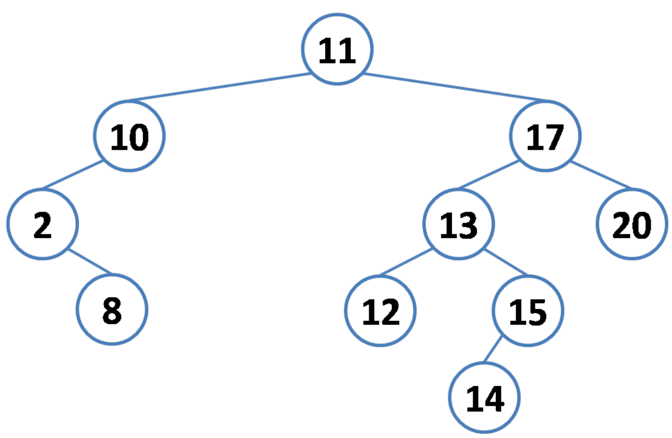
Exercise 5. *Provide an implementation of the BST set interface in Section 9.*

Sample Solutions

Solution of exercise 1 The resulting tree is as follows. Note that the first inserted element is always at the root.



Solution of exercise 2 To achieve the ordering invariant, the key should occur in increasing order as we examine the tree from left to right. For example, the leftmost node must contain 2, while the root must contain 11 (because there are exactly three smaller keys). The resulting tree is then as follows:



Note that this is the only solution.

Now, any insertion sequence where a parent node occurrs before a child node will produce this tree. Here's an example:

11, 17, 13, 10, 20, 15, 2, 14, 8, 12,

Solution of exercise 3 Here's an iterative solution for `dict_lookup`. Note that there is no need for a helper function that searches the underlying tree.

```
entry dict_lookup(dict* D, key k)
/*@requires is_dict(D); @*/
/*@ensures \result == NULL
        || key_compare(entry_key(\result), k) == 0; @*/
{
  tree* T = D->root;
  while (T != NULL) {
    int cmp = key_compare(k, entry_key(T->data));
    if (cmp == 0) return T->data;
    if (cmp < 0) T = T->left;
    else { //@assert cmp > 0;
      T = T->right;
    }
  }
  //@assert T == NULL;
  return NULL;
}
```

Solution of exercise 4 An iterative version of `dict_insert` is as follows. For readability, we moved the creation of a singleton node into the helper function `leaf`. As hinted, we need to maintain a pointer to the parent node so that we can set its left or right child to the newly added node. We do not need to do this when overwriting an existing entry with the same key, and inserting a node to the empty tree. We achieve this by initializing the parent pointer to `NULL` as the root of the tree does not have a parent. If the tree is empty, this pointer will remain `NULL` by the time we are about to attach the new node.

Note that again we didn't need to define a helper function to insert `e` in the underlying tree.

```
tree* leaf(entry e)
//@requires e != NULL;
//@ensures is_bst(\result) && \result != NULL;
{
  tree* T = alloc(tree);
  T->data = e;
  T->left  = NULL;  // Not necessary
  T->right = NULL;  // Not necessary
  return T;
}

void dict_insert(dict* D, entry e)
/*@requires is_dict(D) && e != NULL; @*/
/*@ensures is_dict(D) && dict_lookup(D, entry_key(e)) != NULL; @*/
{
  tree* T = D->root;

  tree* parent = NULL;
  int cmp = 0;
  while (T != NULL) {
    cmp = key_compare(entry_key(e), entry_key(T->data));
    if (cmp == 0) {
      T->data = e;
      return;
    }
    parent = T;
    if (cmp < 0)
      T = T->left;
    else { //@assert cmp > 0;
      T = T->right;
    }
  }
  //@assert T == NULL;

  T = leaf(e);               // Reusing T to hold the new node

  if (parent == NULL)        // Inserting at the root
    D->root = T;
  else if (cmp < 0)          // Inserting to the left of a node
    parent->left = T;
  else { //@assert cmp > 0;  // Inserting to the right of a node
    parent->right = T;
  }
}
```

Solution of exercise 11 The resulting code is shown next. We started from the code for BST dictionaries developed in this chapter. We identified the types `key` and `entry`, which we renamed to `elem`. Because `set_contains` returns a boolean, there is no need for a special value to indicate that an element is not present in a set. This means that an `elem` does not need to be a pointer — another simplification (except in the specification function `is_ordered`, which relies on pointers).

```
typedef struct tree_node tree;
struct tree_node {
  elem data;
  tree* left;
  tree* right;
};

bool is_tree(tree* root) {
  if (root == NULL) return true;
  return is_tree(root->left) && is_tree(root->right);
}

bool is_ordered(tree* T, elem* lo, elem* hi)
//@requires is_tree(T);
{
  if (T == NULL) return true;
  elem e = T->data;
  elem* lo_left  = alloc(elem);  *lo_left = e;
  elem* hi_right = alloc(elem);  *hi_right = e;
  return (lo == NULL || elem_compare(*lo, e) < 0)
      && (hi == NULL || elem_compare(e, *hi) < 0)
      && is_ordered(T->left, lo, hi_right)
      && is_ordered(T->right, lo_left, hi);
}

bool is_bst(tree* T) {
  return is_tree(T) && is_ordered(T, NULL, NULL);
}
```

```
bool bst_lookup(tree* T, elem k)
/*@requires is_bst(T); @*/
{
  if (T == NULL) return false;

  int cmp = elem_compare(k, T->data);
  if (cmp == 0)  return true;
  if (cmp <  0)  return bst_lookup(T->left, k);
  //@assert cmp > 0;
  return bst_lookup(T->right, k);
}

tree* bst_insert(tree* T, elem e)
//@requires is_bst(T);
//@ensures is_bst(\result) && \result != NULL;
//@ensures bst_lookup(\result, e);
{
  if (T == NULL) {
    /* create new node and return it */
    tree* R  = alloc(tree);
    R->data  = e;
    R->left  = NULL;  // Not required (initialized to NULL)
    R->right = NULL;  // Not required (initialized to NULL)
    return R;
  }

  int cmp = elem_compare(e, T->data);
  if (cmp == 0)     T->data = e;
  else if (cmp < 0) T->left = bst_insert(T->left, e);
  else {
    //@assert cmp > 0;
    T->right = bst_insert(T->right, e);
  }
  return T;
}
```

```
struct set_header {
    tree* root;
};
typedef struct set_header set;

set* set_new() {
  set* new = alloc(set);
  new->root = NULL;
  return new;
}

bool set_contains(set* S, elem e)
//@requires S != NULL;
{
  return bst_lookup(S->root, e);
}

void set_insert(set* S, elem e)
//@requires S != NULL;
//@ensures set_contains(S, e);
{
  S->root = bst_insert(S->root, e);
}

typedef set* set_t;
```