CLRS 12 Binary Search Trees
The search tree data structure supports many dynamic-set operations, including

- **SEARCH**
- **MINIMUM**
- **MAXIMUM**
- **PREDECESSOR**
- **SUCCESSOR**
- **INSERT**
- **DELETE**

Thus, we can use a search tree both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with \( n \) nodes, such operations run in \( \Theta(lg \ n) \) worst-case time. If the tree is a linear chain of \( n \) nodes, however, the same operations take \( \Theta(n) \) worst-case time.

In practice, we can design variations of binary search trees with good guaranteed worst-case performance on basic operations. Chapter 13 presents one such variation, **RED-BLACK** trees, which have height \( O(lg \ n) \).
We can represent a binary search tree (BST) by a linked data structure in which each node is an object.

In addition to a key and satellite data, each node contains attributes left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL.

**binary-search-tree property:**
Let \( x \) be a node in a binary search tree.
If \( y \) is a node in the left subtree of \( x \), then \( y.key \leq x.key \).
If \( y \) is a node in the right subtree of \( x \), then \( y.key \geq x.key \).
**T.root:**

- **key:** “pointer”
- **left:** y
- **right:** z
- **p:** NIL

**satellite data:**

“[poin-ter] noun
1. a person or thing that points.
2. a long, tapering stick used by teachers, lecturers, etc., in pointing things out on a map, blackboard, or the like.
3. the hand on a watch dial, clock face, scale, etc.
4. one of a breed of short-haired hunting dogs trained to point game.
5. a piece of advice, especially on how to succeed in a specific area:
The food expert gave some good pointers on making better salads.
6. Computers. an identifier giving the location in storage of something of interest, as a data item, table, or subroutine.”

**BST objects**

T.root: [Diagram showing BST object with key “pointer” and satellite data as described above]
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2    INORDER-TREE-WALK(x.left)
3    print x.key
4    INORDER-TREE-WALK(x.right)
TREE-SEARCH($x, k$)
1  if $x == \text{NIL}$ or $k == x.key$
2     return $x$
3  if $k < x.key$
4     return TREE-SEARCH($x.left, k$)
5  else return TREE-SEARCH($x.right, k$)

ITERATIVE-TREE-SEARCH($x, k$)
1  while $x \neq \text{NIL}$ and $k \neq x.key$
2     if $k < x.key$
3         $x = x.left$
4     else $x = x.right$
5  return $x$
TREE-MINIMUM\(x\)
1 \(\textbf{while} \ x.\text{left} \neq \text{NIL}\)
2 \(x = x.\text{left}\)
3 \textbf{return} \(x\)

TREE-MAXIMUM\(x\)
1 \(\textbf{while} \ x.\text{right} \neq \text{NIL}\)
2 \(x = x.\text{right}\)
3 \textbf{return} \(x\)
TREE-SUCCESSOR(x)
1 if x.right ≠ NIL
2 return TREE-MINIMUM(x.right)
3 y = x.p
4 while y ≠ NIL and x == y.right
5 x = y
6 y = y.p
7 return y

TREE-PREDECESSOR(x)
1 if x.left ≠ NIL
2 return TREE-MAXIMUM(x.left)
3 y = x.p
4 while y ≠ NIL and x == y.left
5 x = y
6 y = y.p
7 return y
BST successor

successor(x)
BST successor

successor(x)
BST insertion
TREE-INSERT\((T, z)\)

1.  \(y = \text{NIL}\)
2.  \(x = T\.root\)
3.  \textbf{while} \(x \neq \text{NIL}\)  
   \hspace{1em} // finds \(y\) the parent  
   \hspace{1em} // of the new node \(z\)
4.     \(y = x\)
5.     \textbf{if} \(z\.key < x\.key\)
6.         \(x = x\.left\)
7.     \textbf{else} \(x = x\.right\)
8.  \(z\.p = y\)
9.  \textbf{if} \(y == \text{NIL}\)  
   \hspace{1em} // tree \(T\) was empty
10. \(T\.root = z\)
11. \textbf{elseif} \(z\.key < y\.key\)  
   \hspace{1em} // assigns \(z\) as the  
   \hspace{1em} // correct child of \(y\).  
12. \(y\.left = z\)
13. \textbf{else} \(y\.right = z\)
BST insertion
**BST Deletion**

**Figure 12.4:** Deleting a node $z$ from a binary search tree. Node $z$ may be the root, a left child of node $q$, or a right child of $q$.

(a) Node $z$ has no left child. We replace $z$ by its right child $r$, which may or may not be NIL.

(b) Node $z$ has a left child $l$ but no right child. We replace $z$ by $l$.

(c) Node $z$ has two children; its left child is node $l$, its right child is its successor $y$, and $y$’s right child is node $x$. We replace $z$ by $y$, updating $y$’s left child to become $l$, but leaving $x$ as $y$’s right child.

(d) Node $z$ has two children (left child $l$ and right child $r$), and its successor $y \neq r$ lies within the subtree rooted at $r$. We replace $y$ by its own right child $x$, and we set $y$ to be $r$’s parent. Then, we set $y$ to be $q$’s child and the parent of $l$. 
Figure 12.4: Deleting a node $z$ from a binary search tree. Node $z$ may be the root, a left child of node $q$, or a right child of $q$.

(a) Node $z$ has no left child. We replace $z$ by its right child $r$, which may or may not be NIL.
Figure 12.4: Deleting a node $z$ from a binary search tree. Node $z$ may be the root, a left child of node $q$, or a right child of $q$.

(b) Node $z$ has a left child $l$ but no right child. We replace $z$ by $l$. 
Figure 12.4: Deleting a node $z$ from a binary search tree. Node $z$ may be the root, a left child of node $q$, or a right child of $q$.

(c) Node $z$ has two children; its left child is node $l$, its right child is its successor $y$, and $y$’s right child is node $x$. We replace $z$ by $y$, updating $y$’s left child to become $l$, but leaving $x$ as $y$’s right child.
Figure 12.4: Deleting a node \( z \) from a binary search tree. Node \( z \) may be the root, a left child of node \( q \), or a right child of \( q \).

(d) Node \( z \) has two children (left child \( l \) and right child \( r \)), and its successor \( y \neq r \) lies within the subtree rooted at \( r \). We replace \( y \) by its own right child \( x \), and we set \( r \) to be \( y \)'s right child. Then, we set \( y \) to be \( q \)'s child and we set \( l \) to be \( y \)'s left child.
TRANSPLANT\( (T, u, v) \)
1 if \( u.p == \text{NIL} \)
2 \( T.root = v \)
3 elseif \( u == u.p.left \)
4 \( u.p.left = v \)
5 else \( u.p.right = v \)
6 if \( v \neq \text{NIL} \)
7 \( v.p = u.p \)

Replaces \( u \) by \( v \).

TREE-DELETE\( (T, z) \)
1 if \( z.left == \text{NIL} \)
2 TRANSPLANT\( (T, z, z.right) \)
3 elseif \( z.right == \text{NIL} \)
4 TRANSPLANT\( (T, z, z.left) \)
5 else \( y = \text{TREE-MINIMUM}(z.right) \)
6 if \( y.p \neq z \)
7 TRANSPLANT\( (T, y, y.right) \)
8 \( y.right = z.right \)
9 \( y.right.p = y \)
10 TRANSPLANT\( (T, z, y) \)
11 \( y.left = z.left \)
12 \( y.left.p = y \)

BST deletion
CLRS 12 Binary Search Trees
Figure 13.2: The rotation operations on a binary search tree. The operation \textsc{Left-Rotate}(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation \textsc{Right-Rotate}(T, y) transforms the configuration on the left into the configuration on the right. The letters \(\alpha\), \(\beta\), and \(\gamma\) represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property—the keys in \(\alpha\) precede \(x.key\), which precedes the keys in \(\beta\), which precede \(y.key\), which precedes the keys in \(\gamma\).
LEFT-ROTATE(T, x)
The pseudocode for \textbf{LEFT-ROTATE} assumes that $x.right \neq T.nil$.

\textbf{LEFT-ROTATE}($T, x$)

1. $y = x.right$
2. $x.right = y.left$
3. \textbf{if} $y.left \neq T.nil$
4. \hspace{1em} $y.left.p = x$
5. \hspace{1em} $y.p = x.p$
6. \textbf{if} $y.p == T.nil$
7. \hspace{1em} $T.root = y$
8. \textbf{elseif} $x == x.p.left$
9. \hspace{1em} $x.p.left = y$
10. \textbf{else} $x.p.right = y$
11. \hspace{1em} $y.left = x$
12. \hspace{1em} $x.p = y$

// set y

// turn y’s left subtree into x’s right subtree

// link x’s parent to y

// put x on y’s left
1. Every node is either red or black.
2. The root is black.
3. Every leaf ($T.nil$) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.
Figure 13.1: A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. (b) The same red-black tree but with each NIL replaced by the single sentinel T.nil, which is always black, and with black-heights omitted. The root’s parent is also the sentinel. (c) The same red-black tree but with leaves and the root’s parent omitted entirely. We shall use this drawing style in the remainder of this chapter.
Figure 13.1: A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. (b) The same red-black tree but with each NIL replaced by the single sentinel $T.nil$, which is always black, and with black-heights omitted. The root’s parent is also the sentinel. (c) The same red-black tree but with leaves and the root’s parent omitted entirely. We shall use this drawing style in the remainder of this chapter.
**Figure 13.1:** A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. 

(a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. 

(b) The same red-black tree but with each NIL replaced by the single sentinel \( T.nil \), which is always black, and with black-heights omitted. The root’s parent is also the sentinel. 

(c) The same red-black tree but with leaves and the root’s parent omitted entirely. We shall use this drawing style in the remainder of this chapter.
Height of a red-black tree

- **Height of a node** is the number of edges in a longest path to a leaf.
- **Black-height** of a node $x$: $bh(x)$ is the number of black nodes (including $nil[T]$) on the path from $x$ to leaf, not counting $x$. By property 5, black-height is well defined.

**Claim**
Any node with height $h$ has black-height $\geq h/2$.

**Proof** By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black. $\blacksquare$ (claim)
Claim
The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal nodes.

Proof: We prove this claim by induction on the height of $x$.

If the height of $x$ is 0, then $x$ must be a leaf ($T.nil$), and the subtree rooted at $x$ indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

For the inductive step, consider a node $x$ that has positive height $h(x)>0$ and is an internal node with two children. Each child has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of $x$ is less than the height of $x$ itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at $x$ contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes, which proves the claim.
For the inductive step, consider a node $x$ that has positive height $h(x) > 0$ and is an internal node with two children. Each child has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of $x$ is less than the height of $x$ itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at $x$ contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes, which proves the claim.
Lemma
A red-black tree with \( n \) internal nodes has height at most \( 2 \lg(n + 1) \).

Proof
Let \( h \) be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least \( h/2 \). Given that the subtree rooted at the root \( r \) contains at least \( 2^{bh(r)} - 1 \) internal nodes, we have \( n \geq 2^{h/2} - 1 \). Moving the 1 to the left-hand side and taking logarithms on both sides yields

\[
\lg(n + 1) \geq h/2, \text{ or } h \leq 2 \lg(n + 1).
\]
RED-BLACK Tree INSERTION
RB-INSERT(T, z)

1   y = T.nil
2   x = T.root
3   while x ≠ T.nil
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == T.nil
10      T.root = z
11   elseif z.key < y.key
12       y.left = z
13       else y.right = z
14   z.left = T.nil
15   z.right = T.nil
16   z.color = RED
17   RB-INSERT-FIXUP(T, z)
Figure 13.5: Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since \( z \) and its parent \( z.p \) are both red. We take the same action whether (a) \( z \) is a right child or (b) \( z \) is a left child. Each of the subtrees \( \alpha, \beta, \gamma, \delta, \) and \( \varepsilon \) has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5—all downward simple paths from a node to a leaf have the same number of blacks. The while loop continues with node \( z \)'s grandparent \( z.p.p \) as the new \( z \). Any violation of property 4 can now occur only between the new \( z \), which is red, and its parent, if it is red as well.
Figure 13.6: Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because $z$ and its parent $z.p$ are both red. Each of the subtrees $\alpha$, $\beta$, $\gamma$, and $\delta$ has a black root ($\alpha$, $\beta$, and $\gamma$ from property 4, and $\delta$ because otherwise we would be in case 1), and each has the same black-height. We transform case 2 into case 3 by a left rotation, which preserves property 5—all downward simple paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The while loop then terminates, because property 4 is satisfied—there are no longer two red nodes in a row.
RB-INSERT-FIXUP \((T, z)\)

1. while \(z.p.color == \text{RED}\)
2. 
3. if \(z.p == z.p.p.left\)
4. 
5. \(y = z.p.p.right\)
6. 
7. if \(y.color == \text{RED}\)
8. 
9. \(z.p.color = \text{BLACK}\)  // case 1
10. \(y.color = \text{BLACK}\)  // case 1
11. \(z.p.p.color = \text{RED}\)  // case 1
12. \(z = z.p.p\)  // case 1
13. elseif \(z == z.p.right\)
14. 
15. \(z = z.p\)  // case 2
16. \(\text{LEFT-ROTATE} (T, z)\)  // case 2
17. 
18. \(z.p.color = \text{BLACK}\)  // case 3
19. \(z.p.p.color = \text{RED}\)  // case 3
20. \(\text{RIGHT-ROTATE}(T, z.p.p)\)  // case 3
21. else (same as 3—14 with “right” and “left” exchanged)
22. \(T.root.color = \text{BLACK}\)
(c) Case 3

(d)
Application of Red-Black Trees

First Repetition. Given \( n \) numbers \( a_1, \ldots, a_n \), find the smallest \( i \) such that there exists \( 1 \leq j < i \) such that \( a_j = a_i \).

Solution:
Build a Red-Black Tree with the values encountered so far. Whenever you consider a new value in the list, binary search for it in the RB tree. If found, you have solved your problem, and if absent from the RB tree, insert the new value in the Red-Black tree where it belongs.

This is \( O(i \log i) \) where \( i \) is the location of the first repetition. Most natural solutions will be \( \Theta(i^2) \).
First Repetition from RED-BLACK Trees

NIL

22 31 44 7 12 19 22 35 3 40 27
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
First Repetition from RED-BLACK Trees
RED-BLACK Tree DELETION

new $x = T.root$
RED-BLACK Trees deletion

(a)

(b)

(c)

(d)
RED-BLACK Trees deletion

RB-DELETE(T, z)

1. \( y = z \)
2. \( y\text{-original-color} = y\text{.color} \)
3. \( \text{if } z\text{.left} == T\text{.nil} \) \( x = z\text{.right} \)
4. \( \text{RB-TRANSPLANT}(T, z, z\text{.right}) \) \( \) (a) \( \)
5. \( \text{elseif } z\text{.right} == T\text{.nil} \) \( x = z\text{.left} \) \( \) (b) \( \)
6. \( \text{RB-TRANSPLANT}(T, z, z\text{.left}) \) \( \)
7. \( \text{else } y = \text{TREE-MINIMUM}(z\text{.right}) \) \( \) (c) & (d) \( \)
8. \( \text{y\text{-original-color}} = y\text{.color} \)
9. \( x = y\text{.right} \)
10. \( \text{if } y\text{.p} == z \) \( x\text{.p} = y \)
11. \( \text{else } \text{RB-TRANSPLANT}(T, y, y\text{.right}) \) \( \) (d) only \( \)
12. \( \text{y\text{.right} = z\text{.right}} \)
13. \( y\text{.right.p} = y \)
14. \( \text{RB-TRANSPLANT}(T, z, y) \) \( \) (c) & (d) \( \)
15. \( y\text{.left} = z\text{.left} \)
16. \( y\text{.left.p} = y \)
17. \( y\text{.color} = z\text{.color} \)
18. \( \text{if } y\text{-original-color} == \text{BLACK} \) \( \text{RB-DELETE-FIXUP}(T, x) \)
RED-BLACK Trees deletion

If node $y$ was BLACK, three problems may arise, which the call of RB-DELETEFIXUP will remedy.

1. If $y$ had been the root and a RED child of $y$ becomes the new root, property 2 is violated.

2. If both $x$ and $x.p$ are RED, then we have violated property 4.

3. Moving $y$ within the tree causes any simple path that previously contained $y$ to have one fewer BLACK node. Thus, property 5 is violated by any ancestor of $y$ in the tree.

...
3. Moving $y$ within the tree causes any simple path that previously contained $y$ to have one fewer \texttt{BLACK} node. Thus, \textit{property 5} is violated by any ancestor of $y$ in the tree.

We can restore \textit{property 5} by saying that node $x$, now occupying $y$'s original position, has an “extra” \texttt{BLACK}. That is, if we add 1 to the count of \texttt{BLACK} nodes on any simple path that contains $x$, then under this interpretation, \textit{property 5} holds.

When we remove or move the \texttt{BLACK} node $y$, we “push” its \texttt{BLACK}ness onto $x$. The problem is that now node $x$ is either “doubly \texttt{BLACK}“ or “\texttt{RED}-and-\texttt{BLACK}“, and it contributes either 2 or 1, respectively, to the count of \texttt{BLACK} nodes on simple paths containing $x$.

The color attribute of $x$ will remain either \texttt{RED} (if \texttt{RED}-and-\texttt{BLACK}) or \texttt{BLACK} (if doubly \texttt{BLACK}) and the extra \texttt{BLACK} on the node is simply indicated by $x$ pointing to that node (rather than in the color attribute).
Case 1: x’s sibling w is RED
(lines 5–8 of RB-DELETE-FIXUP)

- w must have black children.
- Make w black and x.p red.
- Then left rotate on x.p.
- New sibling of x was a child of w before rotation ⇒ must be black.
- Go immediately to case 2, 3, or 4.
Case 2: $x$’s sibling $w$ is BLACK, and both of $w$’s Children are BLACK (lines 10–11 of RB-DELETE-FIXUP)

[Node with gray outline is of unknown color, denoted by $c$.]

- Take 1 black off $x$ ($\Rightarrow$ singly black) and off $w$ ($\Rightarrow$ red).
- Move that black to $x.p$.
- Do the next iteration with $x.p$ as the new $x$.
- If entered this case from case 1, then $x.p$ was red $\Rightarrow$ new $x$ is red & black $\Rightarrow$ color attribute of new $x$ is RED $\Rightarrow$ loop terminates. Then new $x$ is made black in the last line.
Case 3: \( w \) is BLACK, \( w \)'s left child is RED, and \( w \)'s right child is BLACK (lines 13–16 of \textbf{RB-DELETE-FIXUP})

- Make \( w \) red and \( w \)'s left child black.
- Then right rotate on \( w \).
- New sibling \( w \) of \( x \) is black with a red right child \( \Rightarrow \) case 4.
Case 4: $x$’s sibling $w$ is BLACK, and $w$’s right child is RED (lines 17–21 of RB-DELETE-FIXUP)

[Now there are two nodes of unknown colors, denoted by $c$ and $c'$.]

- Make $w$ be $x.p$’s color ($c$).
- Make $x.p$ black and $w$’s right child black.
- Then left rotate on $x.p$.
- Remove the extra black on $x$ ($\Rightarrow x$ is now singly black) without violating any red-black properties.
- All done. Setting $x$ to root causes the loop to terminate.
RB-DELETE-FIXUP \((T, x)\)

1. \(\text{while } x \neq T.\text{root and } x.\text{color} = \text{BLACK} \)
2. \(\text{if } x = x.p.\text{left} \)
3. \(w = x.p.\text{right} \)
4. \(\text{if } w.\text{color} = \text{RED} \)
5. \(w.\text{color} = \text{BLACK} \quad // \text{case 1} \)
6. \(x.p.\text{color} = \text{RED} \quad // \text{case 1} \)
7. \(\text{LEFT-ROTATE}(T, x.p) \quad // \text{case 1} \)
8. \(w = x.p.\text{right} \quad // \text{case 1} \)
9. \(\text{if } w.\text{left.color} = \text{BLACK and } w.\text{right.color} = \text{BLACK} \)
10. \(w.\text{color} = \text{RED} \quad // \text{case 2} \)
11. \(x = x.p \quad // \text{case 2} \)
12. \(\text{else if } w.\text{right.color} = \text{BLACK} \)
13. \(w.\text{left.color} = \text{BLACK} \quad // \text{case 3} \)
14. \(w.\text{color} = \text{RED} \quad // \text{case 3} \)
15. \(\text{RIGHT-ROTATE}(T, w) \quad // \text{case 3} \)
16. \(w = x.p.\text{right} \quad // \text{case 3} \)
17. \(w.\text{color} = x.p.\text{color} \quad // \text{case 4} \)
18. \(x.p.\text{color} = \text{BLACK} \quad // \text{case 4} \)
19. \(w.\text{right.color} = \text{BLACK} \quad // \text{case 4} \)
20. \(\text{LEFT-ROTATE}(T, x.p) \quad // \text{case 4} \)
21. \(x = T.\text{root} \quad // \text{case 4} \)
22. \(\text{else (same as 3–21 with “right” and “left” exchanged)} \)
23. \(x.\text{color} = \text{BLACK} \)
CLRS 13 Red-Black Trees