CS 161 (Stanford, Winter 2023) Lecture 7

Adapted from Virginia Williams' lecture notes. Additional credits go to Ilan Goodman, Vishnu Sundaresan, Wilbur Yang and Gregory Valiant.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Heaps and Binary Search Trees

1 Data Structures

Thus far in this course we have mainly discussed algorithm design, and have specified algorithms at a relatively high level. For example, in describing sorting algorithms, we often assumed that we could insert numbers into a list in constant time; we didn't worry too much about the actual implementation of this, and took it as an assumption that this could actually be implemented (e.g. via a linked-list).

In this lecture, we will talk about the actual design and implementation of some useful data-structures. The purpose of this lecture is mainly to give you a taste of data-structure design. We won't discuss this area too much more in the course, and will switch back to discussing higher-level algorithms. If you like data-structures, CS166 is an entire course about them!

To motivate the data structures that we will discuss in this lecture, consider the following table that lists a bunch of basic operations that we would like to perform on a set/list of numbers, together with the worst-case runtime of performing those operation for two of the data structures that you are (hopefully) familiar with: linked lists of n (unsorted) numbers, and an array of n sorted numbers. In this lecture, we will assume that all the numbers we store are unique.

Operation	Runtime with Unsorted Linked List	Runtime with Sorted Array
Search (i.e. find 17)	$\Theta(n)$	$\Theta(\log n)$ (via binary search)
Select (i.e. find 12 smallest element)	$\Theta(n)$	$\Theta(1)$ (just look at 12th elt.)
Rank (i.e. # elt. less than 17)	$\Theta(n)$	$\Theta(\log n)$
Predecessor/Sucessor[a]	$\Theta(n)$	$\Theta(1)$
Insert element	$\Theta(1)$	$\Theta(n)$
Delete element	$\Theta(1)$ [b]	$\Theta(n)$

[a]Here, the "successor" of an element is asking for the smallest element that is larger than the element we are currently pointing to.

[b]This requires a pointer to the element in the linked list. If we just want to delete some value, we need to first search for it, which takes a $\Theta(n)$ time.

The above table makes Sorted Arrays look like a pretty decent data structure for *static* data.

In many applications, however, the data changes often, in which case the $\Theta(n)$ time it takes to insert or delete an element is prohibitive. This prompts the question: *Is it possible to have one data structure that is the best of both worlds (logarithmic or constant time for all these operations)?* The answer is, essentially, "Yes", provided we don't mind if some of the $\Theta(1)$'s turn into $\Theta(\log n)$. We will sketch out one such data structure, the *Binary Search Tree.*

Before talking about binary search trees, we sketch another data structure called *Heap*. Heaps are worse than both unsorted linked lists and sorted arrays in just about any parameter in the table above, except that they can allow for efficient ($O(\log n)$ time) insertions - still worse than linked lists. But there is one operation that Heaps do exceptionally well (again, $\Theta(\log n)$): $\mathsf{extract} - \mathsf{min}$, which outputs the minimum element, and then deletes it from the heap. Heaps (in their simple form that we consider here) still don't beat the asymptotic runtime of binary search trees for any operation, but they have two other advantages: they are much simpler, and if you only want to run insert and $\mathsf{extract} - \mathsf{min}$ they are also much faster in practice.

2 Heaps

Definition 1 (Complete binary tree)**.** A complete binary tree is a rooted binary tree where each level is full except maybe the last level, and all nodes on the last level are as far left as they can be.

Definition 2 (Binary min-heap)**.**

A *Binary min-heap* is a data structure that stores elements that have keys from a totally ordered universe (say, the integers). A binary min-heap supports the following operations:

- $\mathsf{insert}(i)$: Inserts an element with key i into the data structure
- $\mathsf{extract} - \mathsf{min}$: Returns the element with the minimum key and deletes it from the data structure.

A binary heap stores elements in a complete binary tree with a root r. Each node x has $\mathsf{key}(x)$ (the key of the element stored in x), $\mathsf{p}(x)$ (the parent of x, where $p(r) = \mathsf{NIL}$), $\mathsf{left}(x)$ (the left child of x), and $\mathsf{right}(x)$ (the right child of x). The children of x are either other nodes or NIL.

A binary heap should satisfy two key properties: The first key property is that for every node x, the keys of all nodes under x are greater than $\mathsf{key}(x)$. The second key property of a heap is that its tree is complete (see 1).

As explained above, binary min-heaps (which we call heaps for short) support two operations[1]: $\mathsf{insert}(i)$ and $\mathsf{extract} - \mathsf{min}$. These are particularly useful if you want to have a *priority queue* where elements (e.g. jobs) arrive in an arbitrary order, but always leave in order of their key/priority (e.g. when the highest priority job is executed first).

[1] You can implement other operations such as $\mathsf{search}(i)$ and $\mathsf{delete}(i)$ on a heap, but they would not be efficient (take $\Theta(n)$ time).

2.1 Basic Operations on Heaps

In the heap data structure we always maintain a pointer to the last node in the heap (the rightmost node in the last level), as well as a pointer to the next node to be created (typically the one to the right of the last node; except when the last level is full, in which case this is the left-most node in the next level). Maintaining those is easy when we think of the representation of a heap as an array. But to make things simpler, we will abstract these details and simply assume access to these pointers. (See CLRS for details.)

2.1.1 insert(i)

To insert an element, we create a new node with key i at the bottom of the heap. Then recursively "propagate it up the heap" until the heap property is restored: That is compare i to the key of the new element's parent; if i is smaller than the key of its parent, replace the keys, and continue to compare with the parent's parent, etc. Once the algorithm reaches a parent whose key is smaller, the insertion is complete.

Algorithm 1: Heap insert(i)

```
$x \leftarrow$ first node without two children
$y \leftarrow$ new node with key($y$) $\leftarrow i$, left($y$) $\leftarrow$ NIL, right($y$) $\leftarrow$ NIL, p($y$) $\leftarrow x$
**if** left($x$) == NIL **then**
    left($x$) $\leftarrow y$
**else**
    right($x$) $\leftarrow y$
**while** p($y$) $\neq$ NIL & key(p($y$)) > key($y$) **do**
    $v \leftarrow$ key($y$)
    key($y$) $\leftarrow$ key(p($y$))
    key(p($y$)) $\leftarrow v$
    $y \leftarrow$ p($y$)
```

2.1.2 extract − min

To extract-min, we save the key of the root, replace it with the key of the last node, and delete the last node. Then we recursively "propagate the key copied from the last node down the tree". That is compare to the children of the root; if the key of the root is larger than the keys of one of the children, swap the smaller key up, and recurse on the corresponding sub-tree. Once the algorithm reaches children whose keys are larger, the heap property is restored. Then we can return the value we saved at the beginning.

Algorithm 2: Heap extract – min

```
$x \leftarrow$ last node
$r \leftarrow$ root; $m \leftarrow$ key($r$)
key($r$) $\leftarrow$ key($x$)
if right(p($x$)) == NIL then
    left(p(($x$)) $\leftarrow$ NIL
else
    right(p($x$)) $\leftarrow$ NIL
delete node($x$)
while True do
    if left($r$) == NIL then
        break
    else if key(left($r$)) < key($r$) & right($r$) == NIL then
        $v \leftarrow$ key($r$)
        key($r$) $\leftarrow$ key(left($r$))
        key(left($r$)) $\leftarrow v$
        $r \leftarrow$ left($r$)
    else if key(left($r$)) < key($r$), key(right($r$)) then
        $v \leftarrow$ key($r$)
        key($r$) $\leftarrow$ key(left($r$))
        key(left($r$)) $\leftarrow v$
        $r \leftarrow$ left($r$)
    else if key(right($r$)) < key($r$), key(left($r$)) then
        $v \leftarrow$ key($r$)
        key($r$) $\leftarrow$ key(right($r$))
        key(right($r$)) $\leftarrow v$
        $r \leftarrow$ right($r$)
    else
        break
return $m$
```

3 Binary Search Trees

Definition 3 (Binary search tree)**.**

A *binary search tree (BST)* is a data structure that stores elements that have keys from a totally ordered universe (say, the integers). In this lecture, we will assume that each element has a unique key. A BST supports the following operations:

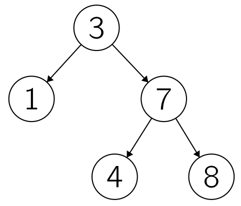
- search(i): Returns an element in the data structure associated with key i
- insert(i): Inserts an element with key i into the data structure
- delete(i): Deletes an element with key i from the data structure, if such an element

exists

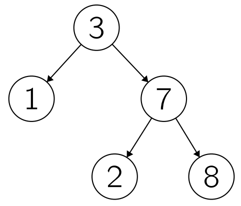
A BST stores the elements in a binary tree with a root r. Each node x has $\mathsf{key}(x)$ (the key of the element stored in x), $\mathsf{p}(x)$ (the parent of x, where $p(r) = \mathsf{NIL}$), $\mathsf{left}(x)$ (the left child of x), and $\mathsf{right}(x)$ (the right child of x). The children of x are either other nodes or NIL.

The key BST property is that for every node x, the keys of all nodes under $\mathsf{left}(x)$ are less than $\mathsf{key}(x)$ and the keys of all nodes under $\mathsf{right}(x)$ are greater than $\mathsf{key}(x)$.

Example 4. In the following example, the root node r stores 3 in it ($\mathsf{key}(r) = 3$), its left child $\mathsf{left}(x)$ stores 1, its right child $\mathsf{right}(x)$ stores 7, and all leaf nodes (storing 1, 4, and 8, respectively) have NIL as their two children.



Example 5. The following binary tree is not a BST since $2 > 3$ and 2 is a child of 7, which is the right child of 3:



Some properties. *Relationship to Quicksort*: We can think of each node x as a pivot for quicksort for the keys in its subtree. The left subtree contains $A_<$ for $\mathsf{key}(x)$ and the right subtree contains $A_>$ for $\mathsf{key}(x)$.

Sorting the keys: We can do an *inorder* traversal of the tree to recover the nodes in sorted order from left to right (the smallest element is in the leftmost node and the largest element is in the rightmost node). The Inorder procedure takes a node x and returns the keys in the subtree under x in sorted order. We can recursively define Inorder(x) as: (1) If left(x) $\neq$ NIL, then run Inorder(left(x)), then: (2) Output key(x) and then: (3) If right(x) $\neq$ NIL, run Inorder(right(x)). With this approach, for every x, all keys in its left subtree will be output before x, then x will be output and then every element in its right subtree.

Subtree property: If we have a subtree where x has y as a left child, y has z as a right child, and z is the root for the subtree T_z, then our BST property implies that all keys in T_z are $> y$ and $< x$. Similarly, if we have a subtree where x has y as a right child, y has z as a left child, and z is the root of the subtree T_z, then our BST property implies that all keys in T_z are between x and y.

3.1 Basic Operations on BSTs

The three core operations on a BST are search, insert, and delete. For this lecture, we will assume that the BST stores distinct numbers, i.e. we will identify the objects with their names and we will have each name be represented by a number.

3.1.1 search

To search for an element, we start at the root and compare the key of the node we are looking at to the element we are searching for. If the node's key matches, then we are done. If not, we recursively search in the left or right subtree of our node depending on whether this node was too large or too small, respectively. If we ever reach NIL, we know the element does not exist in our BST. In the following algorithm in this case, we simply return the node that would be the parent of this node if we inserted it into our tree.

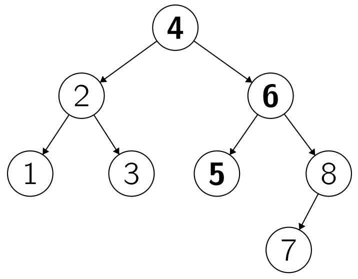
Algorithm 3: search(i)

```
return search($root$, $i$)
```

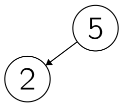
Algorithm 4: search(x, i)

```
if key($x$) == $i$ then
    return $x$
else if $i$ < key($x$) then
    if left($x$) == NIL then
        return $x$
    else
        return search(left($x$), $i$)
else if $i$ > key($x$) then
    if right($x$) == NIL then
        return $x$
    else
        return search(right($x$), $i$)
```

Example 6. If we call **search**(5.5) on the following BST, we will return the node storing 5 (by taking the path in bold). This also corresponds to the path taken when calling **search**(4.5) or **search**(5).



Claim 7. search(i) *returns a node containing* i *if* $i \in$ BST, *otherwise a node* x *such that either* key(x) *is the smallest in* BST $> i$ *or the largest in* BST $< i$, *where node* x *happens to be the parent of the new node if it were to be inserted into* BST. *This follows directly from the BST property. Try to formally prove this claim as an exercise.*



Remark 8. search(i) does **not** necessarily return the node in BST that is closest in value. Consider the tree above. If we search for an element with a key of 4, the node with key 2 is returned, whereas the element with key 5 is the closest element by value.

3.1.2 insert

As before, we will assume that all keys are distinct. We will search(i) for a node x to be the parent and create a new node y, placing it as a child of x where it would logically go according to the BST property.

Algorithm 5: insert(i)

```
$x \leftarrow$ search($i$)
$y \leftarrow$ new node with key($y$) $\leftarrow i$, left($y$) $\leftarrow$ NIL, right($y$) $\leftarrow$ NIL, p($y$) $\leftarrow x$
if $i <$ key($x$) then
    left($x$) $\leftarrow y$
else
    right($x$) $\leftarrow y$
```

Remark 9. Notice that x needed to have NIL as a child where we want to put y by the properties of our search algorithm.

3.1.3 delete

Deletion is a bit more complicated. To delete a node x that exists in our tree, we consider several cases:

1. If x has no children, we simply remove it by modifying its parent to replace x with NIL.
2. If x has only one child c, either left or right, then we elevate c to take x's position in the tree by modifying the appropriate pointer of x's parent to replace x with c, and also fixing c's parent pointer to be x's parent.
3. If x has two children, a left child c_1 and right child c_2, then we find x's immediate successor z and have z take x's position in the tree. Notice that z is in the subtree under x's right child c_2 and we can find it by running $z \leftarrow$ search(c_2, key(x)). Note that since z is x's successor, it doesn't have a left child, but it might have a right child. If z has a right child, then we make z's parent point to that child instead of z (also fixing

the child's parent pointer). Then we replace x with z, fixing up all relevant pointers: the rest of x's original right subtree becomes z's new right subtree, and x's left subtree becomes z's new left subtree.

(Note that alternatively, we could have used x's immediate predecessor y and followed the same analysis in a mirrored fashion.)

In the following algorithm, if p is the parent of x, child(p) refers to left(p) if x was the left child of p and to right(p) otherwise.

Algorithm 6: delete(i)

```
$x \leftarrow$ search($i$)
if key($x$) $\neq i$ then return
if NIL = left($x$) and NIL = right($x$) then
    child(p($x$)) $\leftarrow$ NIL
    delete-node($x$)
if NIL = left($x$) then
    $y \leftarrow$ right($x$)
    p($y$) $\leftarrow$ p($x$)
    child(p($y$)) $\leftarrow y$
    delete-node($x$)
else if NIL = right($x$) then
    $y \leftarrow$ left($x$)
    p($y$) $\leftarrow$ p($x$)
    child(p($y$)) $\leftarrow y$
    delete-node($x$)
else $x$ has two children
    $z \leftarrow$ search(right($x$), key($x$))
    $z' \leftarrow$ right($z$)
    left(p($z$)) $\leftarrow z'$
    p($z'$) $\leftarrow$ p($z$)
    replace $x$ with $z$
    delete-node($x$)
```

3.1.4 Runtimes

The worst-case runtime for search is O(height of tree). As both insert and delete call search a constant number of times (once or twice) and otherwise perform $O(1)$ work on top of that, their runtimes are also O(height of tree).

In the best case, the height of the tree is $O(\log n)$, e.g., when the tree is completely balanced. However, in the worst case it can be $O(n)$ (a long rightward path, for example). This could happen because insert can increase the height of the tree by 1 every time it is called. Currently our operations do not guarantee logarithmic runtimes. To get $O(\log n)$ height we would need to rebalance our tree. There are many examples of self-balancing BSTs, including AVL trees, red-black trees, splay trees (somewhat different but super cool!), etc. Today, we will talk about red-black trees.

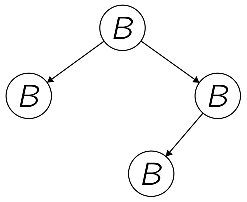
4 Red-Black Trees

One of the most popular balanced BST is the red-black tree developed by Guibas and Sedgewick in 1978. In a red-black tree, all leaves are assumed to have NILs as children.

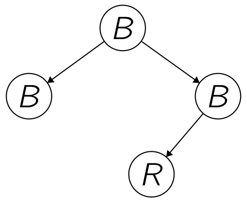
Definition 10. A *red-black tree* is a BST with the following additional properties:

1. Every node is red or black
2. The root is black
3. NILs are black
4. The children of a red node are black
5. For every node x, all x to NIL paths have the same number of black nodes on them

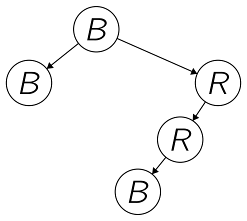
Example 11. (B means a node is black, R means a node is red.) The following tree is *not* a red-black tree since property 5 is not satisfied:



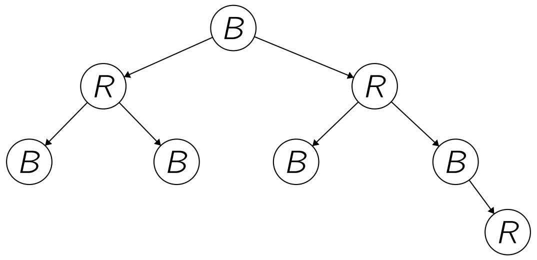
Example 12. (B means a node is black, R means a node is red.) The following tree *is* a red-black tree since all the properties are satisfied:



Example 13. (B means a node is black, R means a node is red.) The following tree is *not* a red-black tree since property 4 is not satisfied:



Example 14. (B means a node is black, R means a node is red.) The following tree *is* a red-black tree:



Remark 15. Intuitively, red nodes represent when a path is becoming too long.

Claim 16. *Any valid red-black tree on n nodes (non-*NIL*) has height* $\leq 2\log_2(n+1) = O(\log n)$.

Proof. For some node x, let $b(x)$ be the "black height" of x, which is the number of black nodes on a $x \to$ NIL path excluding x. We first show that the number of non-NIL descendants of x is at least $2^{b(x)} - 1$ (including x) via induction on the height of x.

Base case: NIL node has $b(x) = 0$ and $2^0 - 1 = 0$ non-NIL descendants. ✓

For our inductive step, let $d(x)$ be the number of non-NIL descendants of x. Then

$$\begin{aligned} d(x) &= 1 + d(\text{left}(x)) + d(\text{right}(x)) \\ &\geq 1 + (2^{b(x)-1} - 1) + (2^{b(x)-1} - 1) \text{ (by induction)} \\ &= 2^{b(x)} - 1 \checkmark \end{aligned}$$

Notice that $b(x) \geq \frac{h(x)}{2}$ (where $h(x)$ is the height of x) since on any root to NIL path there are no two consecutive red nodes, so the number of black nodes is at least the number of red nodes, and hence the black height is at least half of the height. We apply this and the above inequality to the root r (letting $h = h(r)$) to obtain $n \geq 2^{b(r)} - 1 \geq 2^{\frac{h}{2}} - 1$, and hence $h \leq 2\log(n+1)$. □

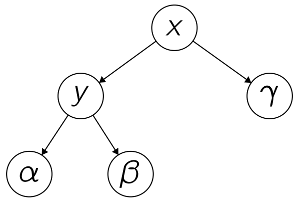
Here is some intuition on why the tree is roughly balanced.

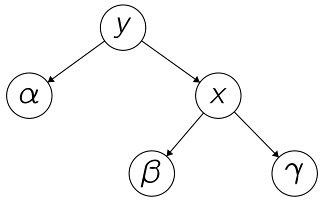
Intuition: By Property (5) of a red-black tree, all $r \to NIL$ paths have $b(r)$ black nodes (excluding the root). Therefore, all these paths have length $\geq b(r)$. However, they also have length $\leq 2 \cdot b(r)$: by Property (4), the number of red nodes is limited to half of the path, since every red node must be followed by a black node, and hence the number of black nodes is at least half of the length of the path. Hence, the lengths of all paths from r to a NIL are within a factor of 2 of each other, and the tree must be reasonably balanced.

Today we will take a brief look at how the red-black tree properties are maintained. Our coverage here is detailed, but not comprehensive, and meant as a case study. For complete coverage, please refer to Chapter 13 of CLRS.

4.1 Rotations

Red-black trees, as do other balanced BSTs, use a concept called rotation. A tree rotation restructures the tree shape locally, usually for the purpose of balancing the tree better. A rotation preserves the BST property (as shown in the following two diagrams). Notably, tree rotations can be performed in $O(1)$ time.





Moving from the first tree to the second is known as a *right rotation of* x. The other direction (from the second tree to the first) is a *left rotation of* y. Notice that we only move the β subtree, which is why we preserve the BST property.

4.2 Insertion in a Red-Black Tree

Let's see how we can perform Insert(i) on a red-black tree while still maintaining all of its properties. The process for inserting a new node is initially similar to that of insertion into any BST.

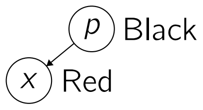
Algorithm 7: insert_rb(i)

```
$p \leftarrow$ search($i$)
$x \leftarrow$ new node with key($x$) $\leftarrow i$, left($x$) $\leftarrow$ NIL, right($x$) $\leftarrow$ NIL, p($x$) $\leftarrow p$
if $i <$ key($p$) then
    left($p$) $\leftarrow x$
else
    right($p$) $\leftarrow x$
color($x$) $\leftarrow$ red
recolor if needed
```

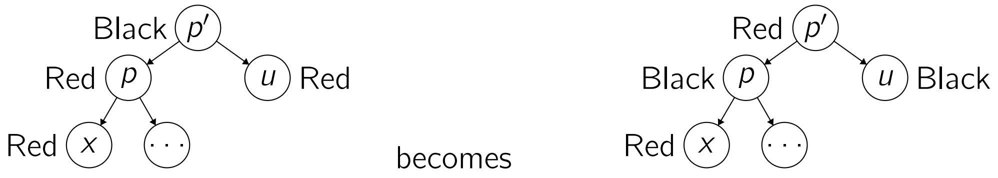
Note that when x is inserted as a red node: Property (1) is satisfied, as we colored the new node red; Property (2) is satisfied, as we did not touch the root; Property (3) is satisfied, as we can color the new NILs black; and Property (5) is satisfied, as we did not change the number of black nodes in the tree. Thus, the only invariant we have to worry about is Property (4), that red nodes have black children.

The recoloring step is broken down into multiple cases. We consider each of them:

Case 1: p is black. In this case, Property (4) is also maintained. So, we simply add x as a new red child of p, and the red-black tree properties are maintained.



Case 2: p is red, and x's uncle u is red. In this case, we insert a red x, change p and u to black, and change p' to red. Because we switched the colors of two nodes on each of these paths (one red$\rightarrow$black and one black$\rightarrow$red), the number of black nodes on each path is unchanged, so Property (5) remains unchanged. If the parent of p', p'', is black, then Property (4) is maintained. Otherwise, if p'' is red and breaks Property (4) by introducing a "double-red" pair of nodes (p' and p''), then we have to recolor recursively starting at p'.

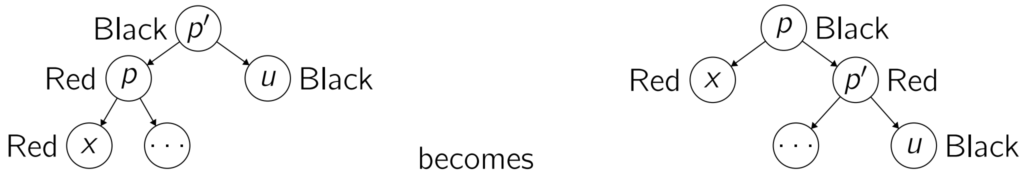


Case 3: p is red, and u is black. There are two possibilities here:

1. We are inserting x as a leaf node. Then, u must be NIL for the red-black tree to have been valid before inserting x. We insert a red x.

2. We are not inserting x; rather, we are recoloring the tree at x from the recursive call in Case 2. We aim to recolor the tree and maintain the number of black nodes on each path from the root to NIL. Note that in this case, x actually has nodes under it.

In both cases, x is red, so we make p black, make p' red, and do a right rotation at p'. We can see that this also maintains the same number of black nodes on each path from the root to NIL, and satisfies Property (4) below p because the original tree was a red-black tree.



If we end up in Case 2 and recursively call recolor, then in the worst case the recursion will bottom out when we hit the root, with a constant number of relabelings and rotations at each level. So, it will be an $O(h)$ operation overall, where h is the height of the tree.

In the analysis above, we considered the cases where x is a left child of p and u is a right child of its parent p'. These cases are representative, showing most of the machinery that we'll need to insert an arbitrary element into an arbitrary red-black tree. (Within Case 2 and Case 3, there are actually a total of four cases each, where p's tree and p''s children could each be swapped, but the recoloring procedure is similar. You are encouraged to read the text for details.)

To summarize, the following is the algorithm for recoloring, in the case where x is a left child and u is a right child.

Algorithm 8: recolor(x) `// x is a left child, u is a right child`

```
$p \leftarrow$ parent($x$)
if $black$ = color($p$) then
    return
$p' \leftarrow$ parent($p$)
$u \leftarrow$ right($p'$)
if $red$ = color($u$) then
    color($p$) $\leftarrow$ black
    color($u$) $\leftarrow$ black
    color($p'$) $\leftarrow$ red
    recolor($p'$)
else if $black$ = color($u$) then
    color($p$) $\leftarrow$ black
    color($p'$) $\leftarrow$ red
    right_rotate($p'$)
```

Based on our analysis above, we can update our red-black trees in $O(h)$ time upon insertion, where h is the height of the tree. The other operations are similar, and also give the guarantee of worst-case performance of $O(h)$ search, insertion, and deletion. Together with Claim 2, which states that $h = O(\log n)$, we get:

Claim 17. *Red-black trees support* insert, delete, *and* search *in $O(\log n)$ time.*

As we have seen, BSTs are very nice – they allow us to maintain a set and report membership, insert, and delete in $O(\log n)$ time. In addition to these basic underlying operations, we can also support other types of queries efficiently. Because the elements are stored maintaining the binary search tree property, we can search for the next largest element or the elements on a range very efficiently. But what if we don't care about these properties? What if we only need to support membership queries? Can we improve our performance of $O(\log n)$ time to nearly constant time? This question motivates our discussion of hash tables, which we will cover in the next lecture.