1 Binary Search Trees.

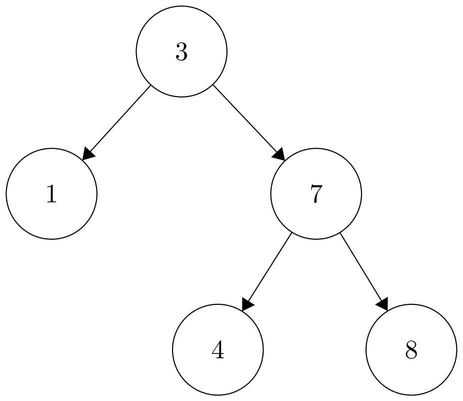
Definition 1.1. *A* binary search tree (BST) *is a data structure that stores some elements that have names from a totally ordered universe (say, the integers). We will assume that each element has a distinct name. A BST supports the following operations:*

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

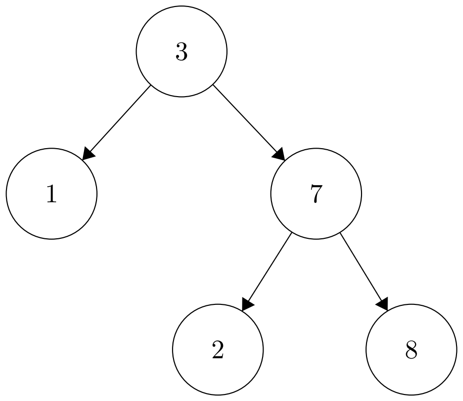
A BST stores the numbers in a binary tree with a root r. Each node x has key(x) *(the name of the element stored in x),* p(x) *(the parent of x, where $p(r)$ =* NIL*),* left(x) *(the left child of x), and* right(x) *(the right child of x). The children of x are either other nodes or NIL.*

The key BST property is that for every node x, the keys of all nodes under left(x) *are $\leq$* key(x) *and the keys of all nodes under* right(x) *are $\geq$* key(x).

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



Example 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 $\mathsf{left}(x) \neq \mathsf{NIL}$, then run Inorder($\mathsf{left}(x)$), then: (2) Output $\mathsf{key}(x)$ and then: (3) If $\mathsf{right}(x) \neq \mathsf{NIL}$, run Inorder($\mathsf{right}(x)$). 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 $\geq y$ and $\leq x$. 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.

1.1 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.

1.1.1 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.

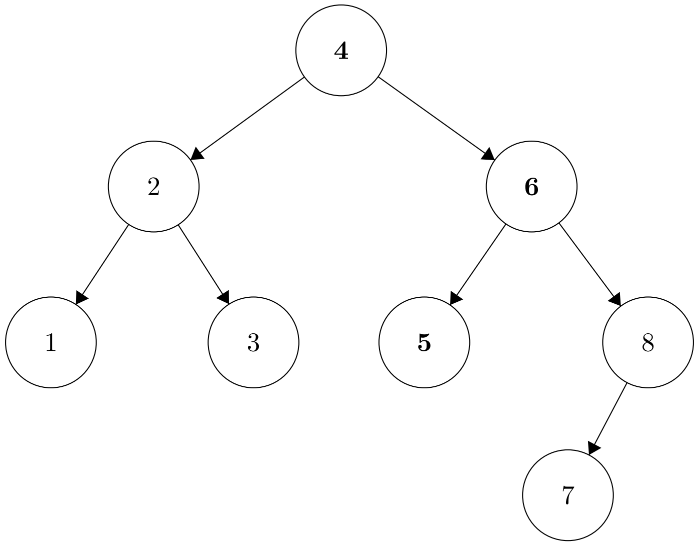
Algorithm 1: search(i)

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

Example 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).

Algorithm 2: 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($x$), $i$)
```



Claim 1. 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 $< i$*. This follows directly from the BST property.*

1.1.2 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 3: 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 1. *Notice that* x *needed to have* NIL *as a child where we want to put* y *by the properties of our* search *algorithm.*

1.1.3 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 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 $z \leftarrow \mathsf{search}(c_2, \mathsf{key}(x))$. 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.

For simplicity, we will leave out case 1 and the symmetric case of case 2 (swapping left and right).

In case 3, x has two children, so we find the smallest element in its right subtree (its successor, call it z) and replace x with z. Clearly z does not have a left child, otherwise it could not be x's successor. Therefore we can cut z out, replacing it with its right subtree (case 1 or 2, which we already know how to handle), and then swap x out for z.

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 t0 right(p) otherwise.

Algorithm 4: delete(i)

```
$x \leftarrow$ search($i$);
**if** key($x$) $\neq i$ **then return**;
**if** left($x$) == NIL **then**
    $y \leftarrow$ right($x$);
    p($y$) $\leftarrow$ p($x$);
    child(p($y$)) $\leftarrow y$;
    delete-node($x$);
**else if** right($x$) == NIL **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($x$);
```

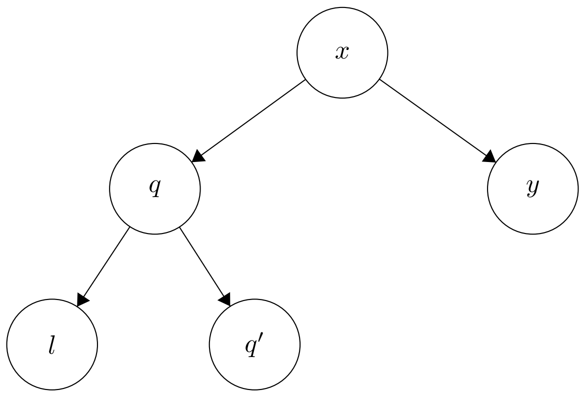
1.1.4 Runtimes

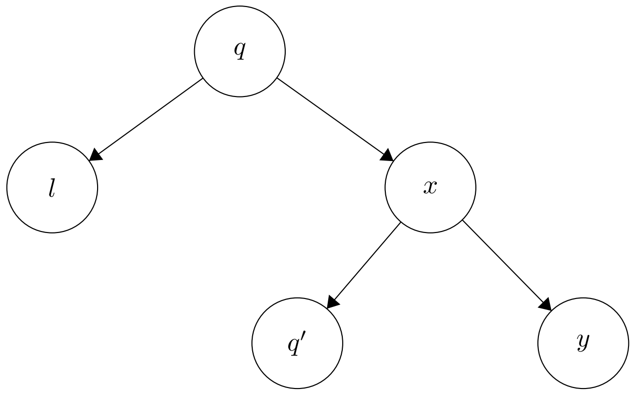
The worst-case runtime for search is O(height of tree). As both insert, and delete call search a constant number of times (once or twice) and otherwise perform $O(1)$ work on top of that, their runtimes are also O(height of tree).

In the best case, the height of the tree is $O(\log n)$, e.g. when the tree is completely balanced, but in the worst case it can be $\Theta(n)$ (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. All such BSTs use a concept called rotation.

1.2 Rotations

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

2 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 2.1. *A* red-black tree *is a BST with the following additional properties:*

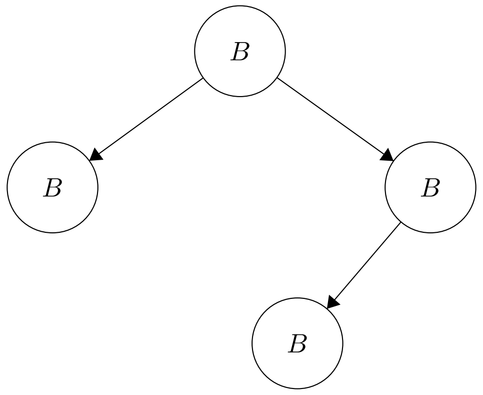
1. *Every node is red or black*
2. *The root is black*

3. NIL*s are black*

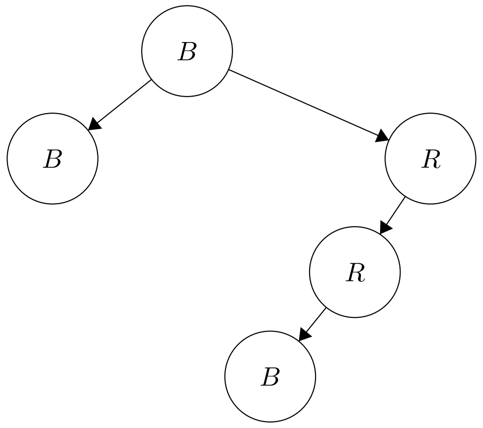
4. *The children of a red node are black*

5. *For every node x, all x to* NIL paths have the same number of black nodes on them

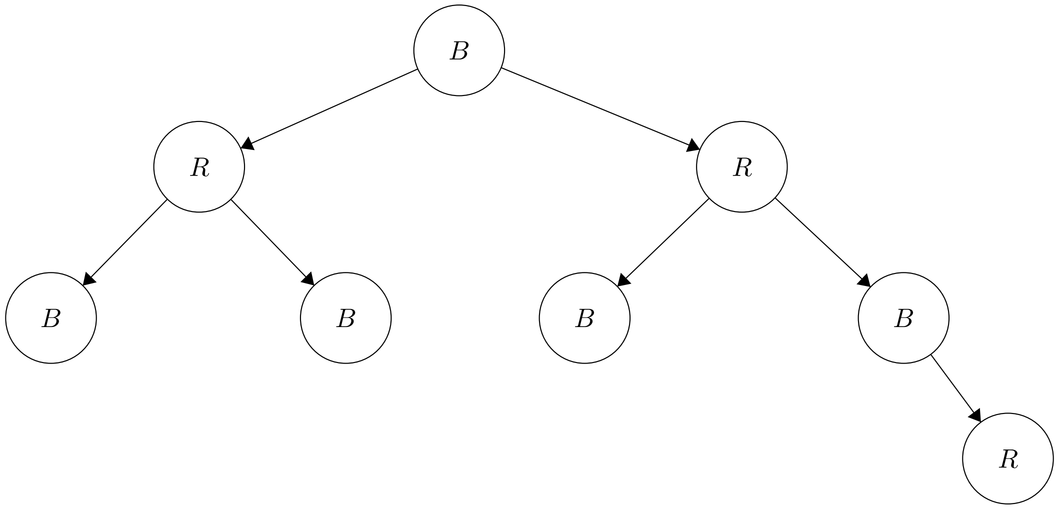
Example (B means a node is black, R means a node is red.) The following tree is *not* a red-black tree since property 5 is not satisfied:



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



Remark 2. *Intuitively, red nodes represent when a path is becoming too long.*

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

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

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

For our inductive step, let $d(x)$ be the number of descendents of x. Then

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

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 descendents inequality to the root r ($h(r) = h$) to obtain $n \geq 2^{b(r)} - 1 \geq 2^{\frac{h}{2}} - 1$, and hence $h \leq 2\log(n+1)$. $\square$

Next lecture we will show that Red-Black trees augment the regular BST insert and delete operations to maintain invariants 1 to 5 in $O(h)$ time where h is the height of the tree. Together with the Claim above, we get:

Claim 3. *Red-black trees support* insert*,* delete*, and* search *in $O(\log n)$ time.*