Computer Science

Dr. Y. Liow (October 28, 2020)

Contents

Chapter 107

Trees

File: chap.tex

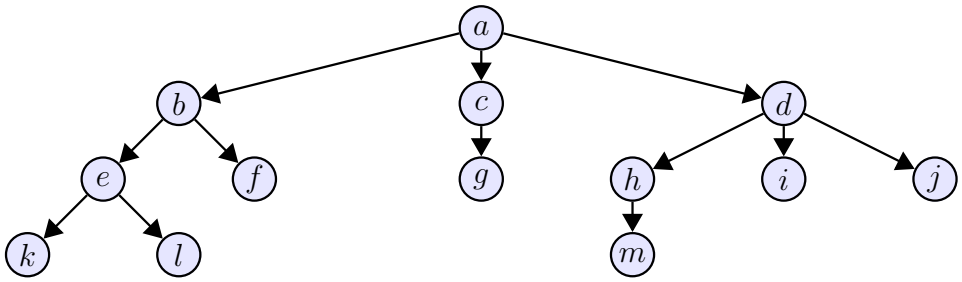
File: tree.tex

107.1 Tree

A tree is just a connected graph without cycles. In this case, I'm talking about a tree as an undirected graph. You have to be careful, sometimes, a tree can also refer to a tree as a directed graph and acyclic.

Sometimes a special node is selected. In this case, such a tree is sometimes called a rooted tree (or a tree with a root).

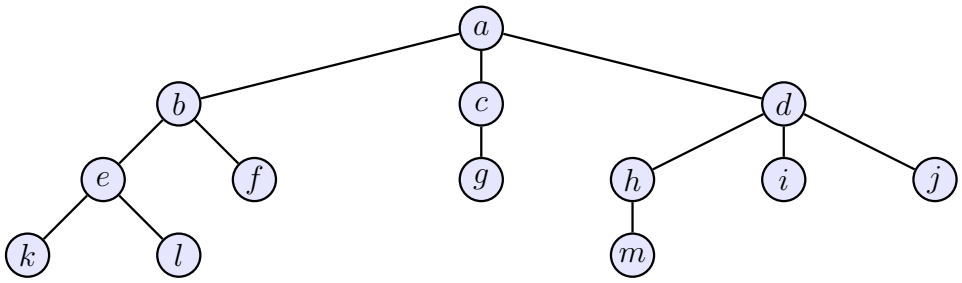
Here's a tree (directed version):



- a is the **root** of the tree. — root
- a has 3 **children**, i.e., b, c, d; a has a **branching factor** of 3. Sometimes, it's convenient to number the children. I'll call b child 0 of a; d is child 2 of a. — children, branching factor
- a is the **parent** of b. — parent
- k,l,f,g,m,i,j are called **leaves** or **non-internal** nodes because they don't have children. — leaves, non-internal
- a, b, c, d, e, h are called **non-leaf** nodes or **internal** nodes because each have at least one child. — non-leaf, internal
- b has a **depth** of 1, i.e., the length of the path from the root to b is 1. I will also say that b is at level 1. A root (see a above) has depth 0. — depth
- b has a **height** of 2, i.e., the length of the longest path from b to a leaf is 2. A leaf (see k above) has height 0. — height

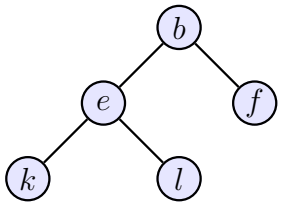
- The **height** of a tree is the height of the root of the tree. The above tree has a height of 3. The height of an empty tree is -1. height
- The root is at level 0. b, c, d are at level 1. e, f, g, h, i, j are at level 2. k, l, m are at level 3.
- Two nodes u and v are **siblings** if they have the same parent. For instance b, c, d are siblings. siblings
- u has **descendent** v if there is a sequence of nodes $u = v_0$, v_1, ..., $v_n = v$ where v_i is the parent of v_{i+1}. v is the **ancestor** of u if u is a descendent of v. For instance in the above tree b is an ancestor of i and i is a descendent of l. descendent ancestor

Note that the above tree is a directed graph. The following is also a tree (as in an *undirected* tree).

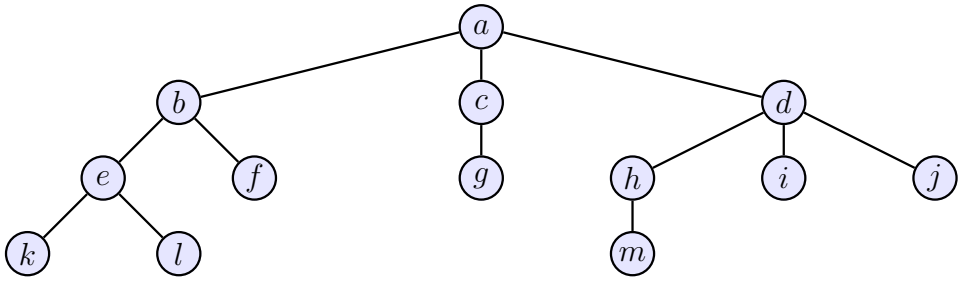


In the case of implementation, you can think of the directed version as having pointers from a parent to the children. In the case of the undirected version, you can think of each edge as allowing you to go both up and down – nodes having pointers going to their children or parent. Of course exceptions being that roots have no parents and leaves have no children.

For each node of a tree T, if you consider all the descendents of the node, you have a tree. This is called a **subtree** of T. For instance subtree



is a subtree of

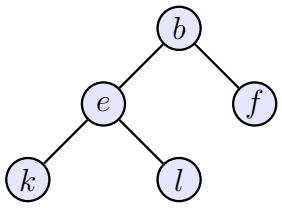


I will say that this is the subtree at b.

In the case of a node with two child, there are two subtrees, the subtree on the left is called the **left subtree** and the one on the right is called the **right subtree**. (Duh.) Look at the subtree at b:

left subtree

right subtree



The nodes e, k, l form the left subtree of b and the single node f form the right subtree of b.

A tree is said to be k–**ary** if every node has at most k children; that's the same as saying that every node has a maximum branching factor of k. In particular, a **binary** tree is a tree with at most 2 children.

k–ary

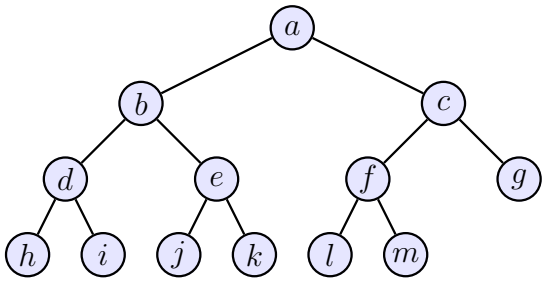
binary

Exercise 107.1.1. Draw all the binary tree with 0, 1, 2, 3, 4, 5 nodes. □

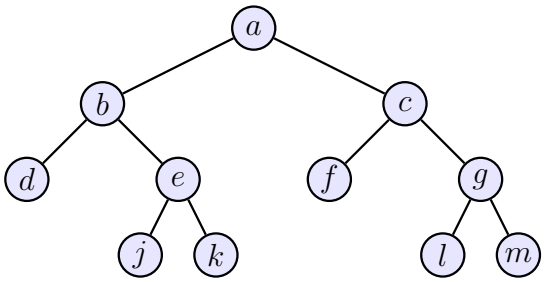
We include the empty graph as a tree, except of course such trees don't have roots. Consider a k–ary tree T.

- T is **full** if every node has exactly k children, except for the leaves, i.e., if every node has either k or 0 children. Here's a full binary tree:

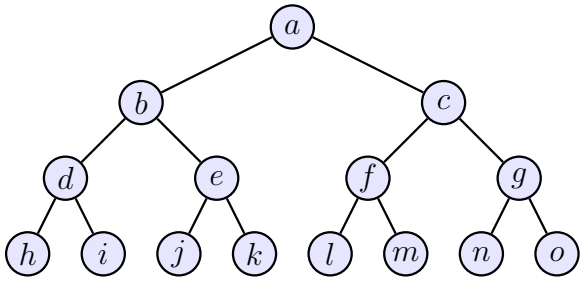
full



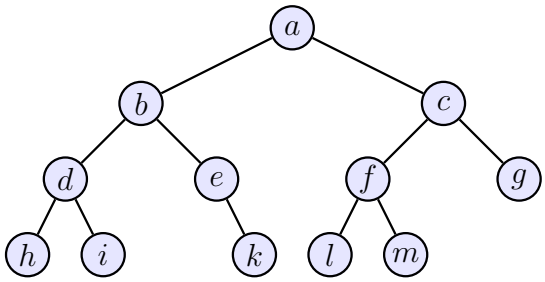
Here's another:



- T is **perfect** if T is full and the leaves are at the same level. Here's a perfect binary tree: perfect



- T is **complete** if it's "almost full" except that the last level might not have all the nodes. Here's a complete binary tree that is not full: complete



- T is **balanced** if at each node, every pair of children have heights differ by 0 or 1. balanced

Exercise 107.1.2. If T is a balanced tree, does it mean that T is complete? If T is complete, does it mean that T is balanced? □

Let T be a k–ary tree with height h and the total number of nodes is n. At level ℓ, there are at most k^ℓ nodes. (Don't forget that the first level that contains only the root is called level 0.) In particular for a binary tree, at level ℓ, there are 2^ℓ nodes.

Therefore if the height is h, T can have at most

$$k^0 + k^1 + \cdots + k^h = \frac{k^{h+1} - 1}{k - 1}$$

nodes. (Remember your geometric sum formula?) In particular, for a binary tree, T can have at most

$$2^{h+1} - 1$$

nodes. Since T has n nodes, we must have the following relation between n and h for a k–ary tree:

$$h + 1 \leq n \leq \frac{k^{h+1} - 1}{k - 1}$$

And for the case of binary tree, we have

$$h + 1 \leq n \leq 2^{h+1}$$

File: implementation.tex

107.2 Implementation

There are many different ways to implement a tree node.

- The node might or might not have a pointer back to the parent. (And as stated in class the parent pointer is usually not necessary.)
- The children (or rather pointers to the children nodes) can be collected together in different ways. For instance they can form an array, a vector or a linked list (singly linked or doubly linked). If the maximum number of children (i.e., the branching factor) is small, then sometimes you can have a number of pointer instance variables directly attached to the node object rather than putting the pointers into a container. This is usually the case for binary trees.

107.2.1 Vector of children

For a general tree, one way to implement a tree node is to do this:

```
class Node
{
private:
    int key_;
    std::vector< Node * > child_;
};
```

(Of course instead of keeping an `int` for each node, you might want to keep other values.) The `child_` member variable will allow a node to point to any number of children.

For the case of a k–ary tree, if k is small, instead of using some kind of container class, it's also common to use a fixed size array:

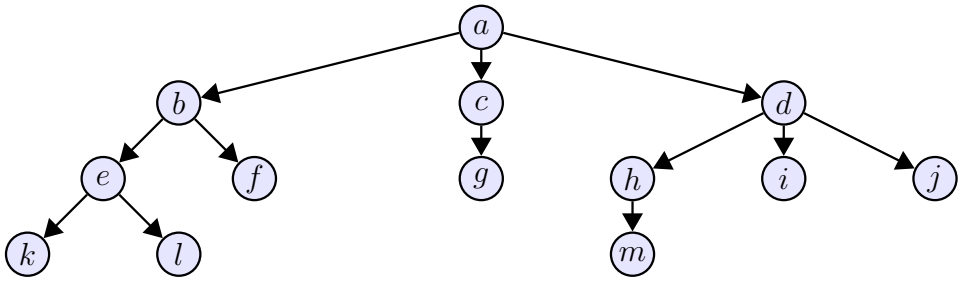
```
const int k = 4;
class Node
{
private:
    int key_;
    Node * child_[k];
};
```

Of course in the case when a node does not have a "child 2", then `child_[2]` is `NULL`.

In the case of a binary tree, it's common to see this:

```
class Node
{
private:
    int key_;
    Node * left_;
    Node * right_;
};
```

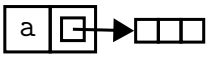
Exercise 107.2.1. Here a tree (the first one in this section):



The key values are characters. Construct the above tree with this

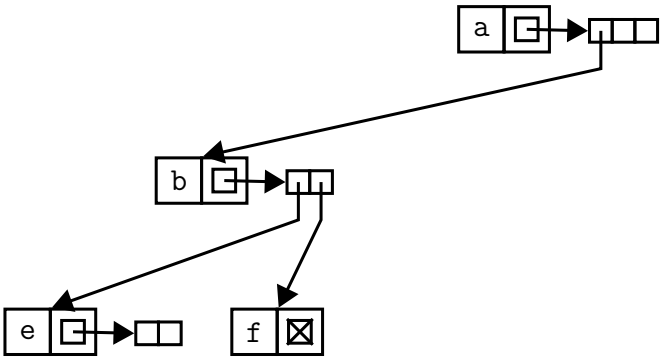
```
class Node
{
private:
    char key_;
    std::vector< Node * > child_;
}
```

Here's a picture of the memory model (using the given class) representing the node with key `'a'`.



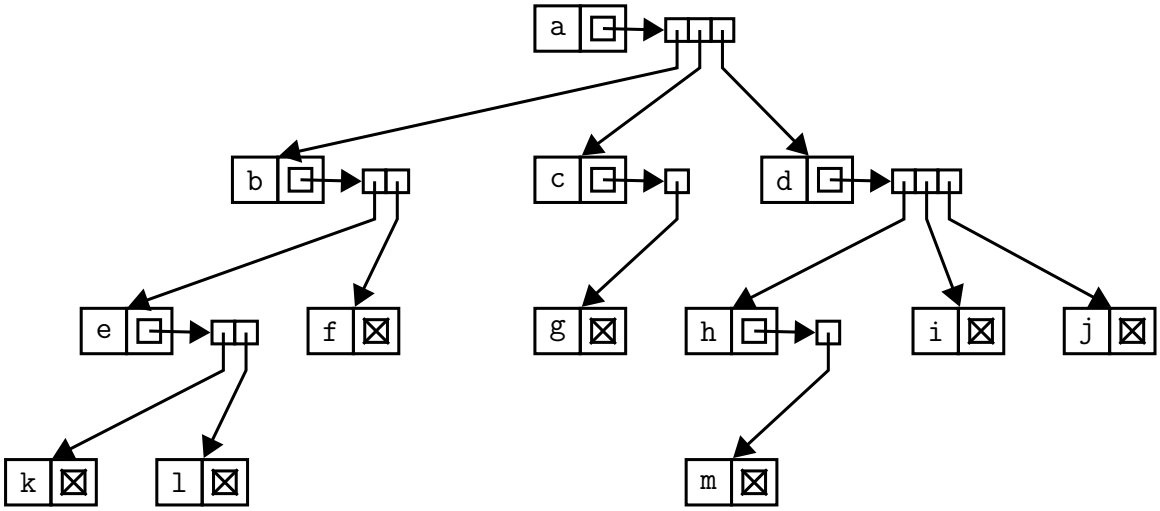
The first box is holds the key value – the box is `key_`. The second box is `child_` which is a `std::vector` object which of course models a dynamic array and therefore contains a pointer to an array (of `Node *` values). The first pointer of this dynamic array of pointers points to a node with key value `'b'`, the second points to a node with key value `'c'`, and the third points to a node with key value `'d'`.

Here's part of the memory model of the above tree:



The pointers in the node with keys `'f'` is `NULL` since the `std::vector` inside the node models array of size 0, i.e., I'm assuming that the pointer in the `std::vector` objects have not been allocated memory yet.

Complete the picture. Solution on next page.



Exercise 107.2.2. First add an obvious constructor to this class. Assume that the values in all the `child_` vectors are not `NULL`. Add a method `child(int)` that does the obvious (see the code below). The three nodes on the bottom left of the above diagram is already constructed for you.

```
int main()
{
    Node * nk = new Node('k');
    Node * nl = new Node('l');
    Node * ne = new Node('e');
    ne->child(0) = nk; // i.e., ne->child_[0] = nk
    ne->child(1) = nl; // i.e., ne->child_[1] = nl

    return 0;
}
```

□

107.2.2 List of children

Depending on how the children are going to be processed, you might want to use something other than `std::vector`. For instance, here's another option:

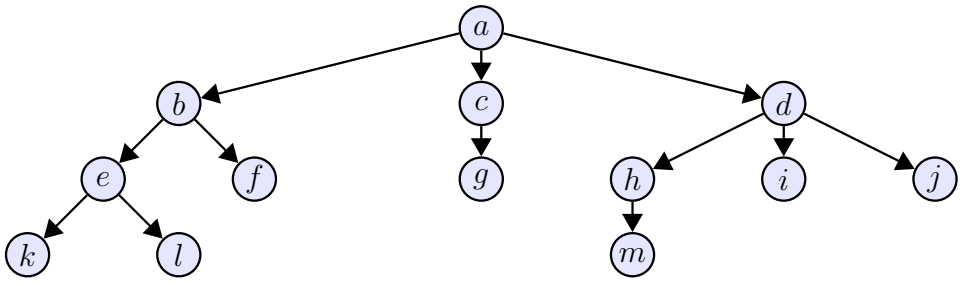
```
class Node
{
private:
    int key_;
    std::list< Node * > child_;
};
```

where a node has a doubly-linked list of pointers to `Node`. You can also use circular list. In the case of the undirected version (think of an edge as bi-directional so that it allows one to go up or down the tree), you might see this:

```
class Node
{
private:
    int key_;
    Node * parent_;
    std::list< Node * > child_;
};
```

Frequently, it's not necessary to keep information on how to get back to the parent in the node since most tree computations start with the root and when you "go down", you keep information of the parents (i.e. where you came from) somewhere such as a stack.

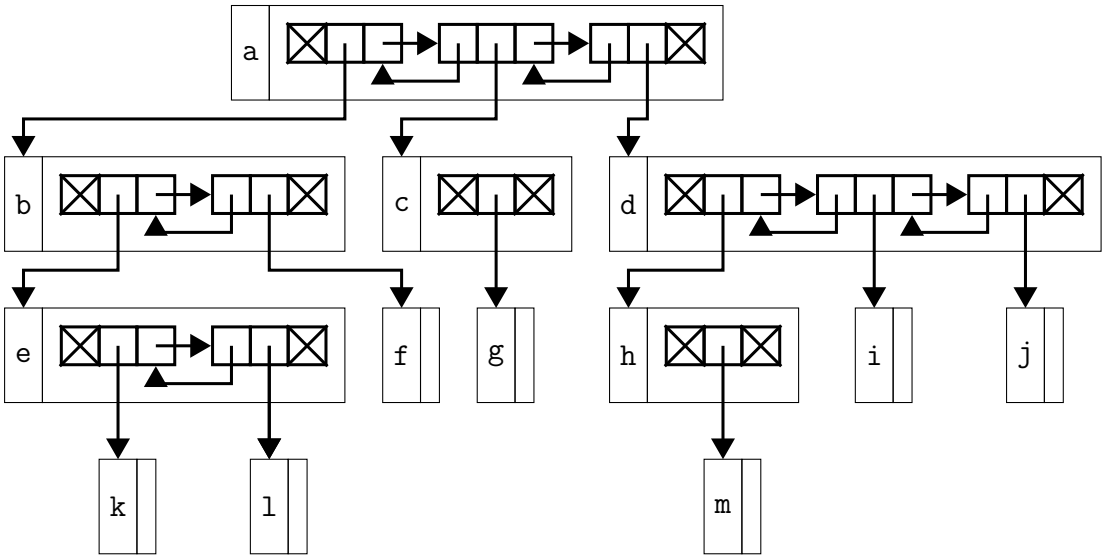
Exercise 107.2.3. Here a tree (the first one in this section):



The key values are characters. Construct the above tree with this

```
class Node
{
private:
int key_;
std::list< Node * > child_;
}
```

The following diagram shows the memory model when the above class is used to model the tree:



Each node is representation by a box containing two boxes where the first box contains the value of the key and the second box contains a doubly linked list. For the doubly linked list, I'm not drawing the pointer to head and pointer to tail just to simplify the diagram. If the linked list is empty, I draw the second box empty.

□

107.2.3 Binary trees

Recall that for the case of a k–ary tree, if k is small, instead of using some kind of container class, it's also common to use a fixed size array:

```
const int k = 4;
class Node
{
private:
    int key_;
    Node * child_[k];
};
```

Now in the case of a *binary* tree, instead of using an array of only two pointers for the children, it's common to see this:

```
class Node
{
private:
    int key_;
    Node * parent_;
    Node * left_;
    Node * right_;
};
```

i.e., you hardcode the pointers (to children) into the node object.

Exercise 107.2.4. Start with this node class:

```
class Node
{
private:
    int key_;
    Node * parent_;
    Node * left_;
    Node * right_;
};
```

In the case of k–ary tree node when you have an array for the children pointers, of course you can loop over children pointers. For binary tree nodes like the above, you can't but this is not a problem since we usually do not loop over them anyway.

Write a constructor so that you can do this:

```
Node * p2 = new Node(2);
```

and this

```
Node * p2 = new Node(2, parent_pointer);
```

and this

```
Node * p2 = new Node(2, parent_pointer,
                     left_child_ptr, right_child_ptr);
```

If a `node` is an object of the above class, when you do

```
std::cout << node << std::endl;
```

you will get this output:

```
<Node 0x97900b0 key:5, parent:0, left:0x9790050, right:0x9790068>
```

107.2.4 Tree class

Of course once you have the definition of a tree node, the important things left would be how to link up the nodes (just like the case of linked lists, double or single), how to delete nodes, how to run through the tree to search for things, etc.

Now, it's not too shocking that if you have a class for the tree, you just need to have access to the root of the tree:

```
class Tree
{
private:
    Node * root_;
};
```

Since it's just a single node (or pointer or reference to a node), frequently, you won't see tree classes at all. Many books or data structure libraries will just have tree node classes and have function/methods attached to the node class. However if you have a tree class, then you can store tree-level information in a tree object. For instance

```
class Tree
{
private:
    Node * root_;
    int size_;
};
```

where `size_` contains the number of nodes in the tree. There are also tree level operations (instead of node level operations) that you attach to a tree class. For instance you can have a node level comparison operator

```
class TreeNode
{
public:
    ...
    bool & operator=(const TreeNode & n) const
    {
        ...
    }
    ...
};
```

and a class level comparison operator

```
class Tree
{
public:
    ...
    bool operator==(const Tree & t) const
    {
        ...
    }
    ...
};
```

In this case, the assignment operator at the node level, say you compute `n1 == n2`, then this is the same as comparing the key or `n1` again the key of `n2`. However the comparison operator at the tree level, say you have the boolean expression `t1 == t2`, will have to compare the tree structure of `t1` (i.e., the layout of the nodes) against the structure of `t2` and at the same time comparing their keys.

In some cases where a tree is self-organizing, you only need to specify values to be added or to be removed from whole tree. (Example: binary search trees, AVL trees, quadtrees, etc. See later notes.). In those cases have a tree class is very helpful. You would envision that a tree is used like this:

```
Tree tree;
tree.insert(5);
tree.insert(7);
tree.insert(2);
tree.insert(0);
...
tree.insert(42);
```

This is similar to for instance a stack or queue which are self-organizing data structures.

There are however many cases where you have to manually decide where to attach nodes:

```
Tree tree;
...
Node * p = tree.find(42);
if (p->left() == NULL)
{
    p->insert_left(51);
}
```

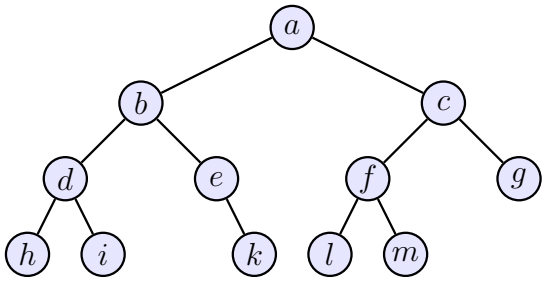
For instance in the case of expression trees (trees from expressions like `1 + 2 * 3`), you have to build the tree in a specific way. You can just insert `2`, insert `+`, etc. You have to use a specific algorithm to build the expression tree yorself.

File: traversal.tex

107.3 Traversal

When you have an array, vector or a linked list, you can "travel through" (or to traverse) the container from the front to the back or the back to the front for instance when you want to print the contents of the container or when you want to search for a value.

In the case of a tree, there are many more ways to traverse the container. I'm going to focus on binary trees since it's the easiest case. Let me illustrate 4 different traversals using this example:



The first three are called **depth first** (DF) traversals because they tend to dive deep down into the tree quickly. The last one is called **breadth first** (BF) traversal before you go down one level at a time.

depth first

breadth first

- DF: Pre-order traversal: Here's a memory aid: "root, left, right". Here is the printing of the node of the above tree using pre-order traversal:

$$a, b, d, h, i, e, k, c, f, l, m, g$$

- DF: In-order traversal: In this case you do "left, root, right". Here's the printing of the above tree using this method of traversal:

$$h, d, i, b, e, k, a, l, f, m, c, g$$

- DF: Post-order traversal: And for this one, use "left, right, root:

$$h, i, d, k, e, b, l, m, f, g, c, a$$

- BF: Breadth-first goes through the tree one level at a time. Here's the printing the tree using breadth first traversal:

$$a, b, c, d, e, f, g, h, i, k, l, m$$

  In this case, we are scanning left-to-right for each level. The important is that all the nodes are processed at a level before going down to the next level.

(There are actually many other traversals. You can come across them if you take the AI class, different traversals will impact the performance of the algorithms that search for an intelligent solutions.)

Here's the pseudocode for the first DF traversal:

```
PREORDER-PRINT
INPUT: p = pointer to root of a tree

if p is NULL:
    return
else:
    print p->key
    PREORDER-PRINT(p->left)
    PREORDER-PRINT(p->right)
```

This version uses recursion. Here's a version that uses a stack instead:

```
PREORDER-PRINT
INPUT: p = pointer to root of a tree

if p is NULL:
    return

let s = empty stack of pointer to node
push p onto s

while s is not empty:
    x = s.pop()
    print x->key
    if x->right is not NULL:
        push x->right onto s
    if x->left is not NULL:
        push x->left onto s
```

It's clear what the pseudocode for inorder and postorder should look like. In

the above, NULL pointers are not inserted into the stack. If we do allow NULL pointers in the stack, then the pseudocode would look like this:

```
PREORDER-PRINT
INPUT: p = pointer to root of a tree

let s = empty stack of pointer to node
push p onto s

while s is not empty:
    x = s.pop()
    if x is not NULL:
        print x->key
        push x->right onto s
        push x->left onto s
```

which is simpler.

For the BF traversal, you use a queue instead of a stack, and do this:

```
BREADTH-FIRST-PRINT
INPUT: p = pointer to root of a tree

if p == NULL:
    return

let q = empty queue of pointers to node
q.enqueue(p)

while q is not empty:
    x = q.dequeue()
    print x->key
    if x->left is not NULL:
        put x->left into q
    if x->right is not NULL:
        put x->right into q
```

This version does not put NULL pointers into the stack. If you allow NULL pointers in the stack, then:

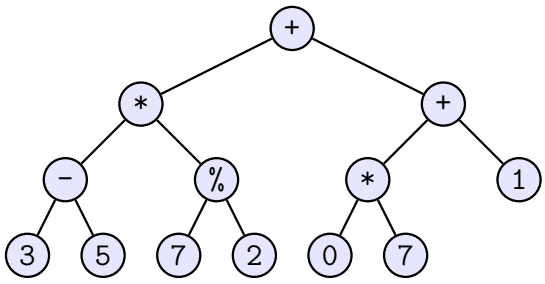
```
BREADTH-FIRST-PRINT
INPUT: p = pointer to root of a tree
```

```
let q = empty queue of pointers to node
q.enqueue(p)

while q is not empty:
    x = q.dequeue()
    if x is not NULL:
        print x->key
        put x->left into q
        put x->right into q
```

which is simpler.

107.4 Trees and arithmetic expressions

Trees can be used to represent arithmetic expressions. For instance look at this:



The value obtained from "evaluating" the tree is this:

```
(((3 - 5) * (7 % 2)) + ((0 * 7) + 1))
```

You can think of the tree as a device that can be used to describe order of operations, or if you like, it's a device that can be used to play the role of parentheses. So really, you have been doing trees since elementary school, right?

Exercise 107.4.1. Assuming the above tree is a tree with node that contain characters, Write a function to traverse the tree and produce the string

```
(((3 - 5) * (7 % 2)) + ((0 * 7) + 1))
```

What tree traversal would you use? □

107.5 Some basic operations

Here are some basic methods on a node. Note that there are so many different variations of trees that some of the methods might not apply for some cases. So you have to understand the context.

The following are some basic queries (they don't modify the tree in any way):

```
node.is_root()          true iff node is a root
node.is_leaf()          true iff node is a leaf
node.is_nonleaf()       true iff node is non-leaf

node.level()            level of node
node.height()           height of node
node.depth()            depth of node
node.size()             number of descendents + 1
```

Note that for instance in the case of `is_root()`, the only way for this to be meaningful is when the node has a parent pointer: a node is a root is the parent pointer is `NULL`. So if your node class does not have a parent pointer member, then this method is not meaningful.

For querying ancestor information, you have the following:

```
node.parent()           pointer/reference to parent
node.root()             pointer/reference to root
```

The next two methods are related to siblings of a node. This is usually only available if the siblings are organized in an iterable manner such as for instance if the siblings form a linked list.

```
node.next()             pointer/reference to next
                        sibling; NULL if there is no
                        next sibling.
node.prev()             pointer/reference to previous
                        sibling; NULL if there is no
                        previous sibling.
```

In the case where the siblings form a circular list, it's of course important that you don't run into an infinite loop!

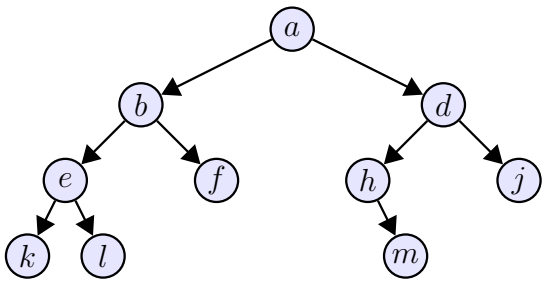
For querying children related information, you might have the following:

`node.num_children()`	number of children
`node.children()`	pointer/reference to iterable container with children
`node.first_child()`	first child (assuming children are ordered)
`node.last_child()`	pointer/reference to last child (assuming children are ordered)
`node.child(i)`	pointer/reference to the i-th child

Sometimes the following descendent queries are useful too:

`node.leftmost()`	pointer/reference to leftmost starting at node; for a binary tree, keep following the left pointer/reference
`node.rightmost()`	pointer/reference to rightmost starting at node; for a binary tree, keep following the right pointer/reference

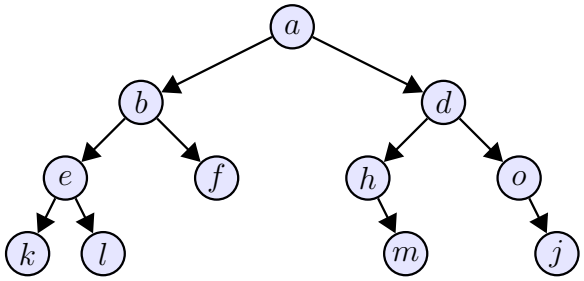
For insertion, look at this binary tree:



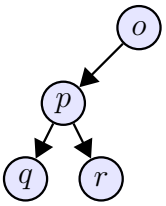
The most common case of insertion is as a child at a position that is available. For instance in the above, note that at node l you can insert a node at the left or the right. However note that at h, you can only insert at the left. Note that the above also makes sense when the node that is inserted is actually the root of a tree. In the case of a k–ary tree, if child-i is available, you can insert a node at that position.

Most tree API do not include inserting a node at a place that is not available. Although that's entirely reasonable. For instance there's no reason why you cannot do an insertion right at the node d with new data o (i.e., it goes between

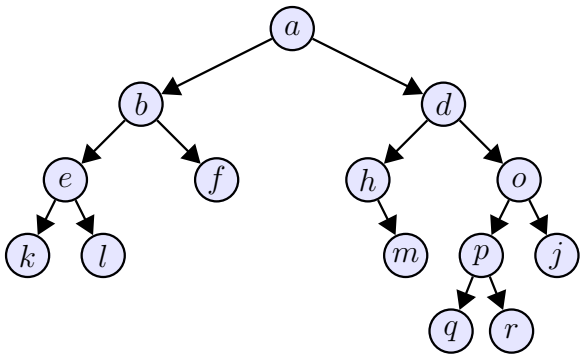
d and j):



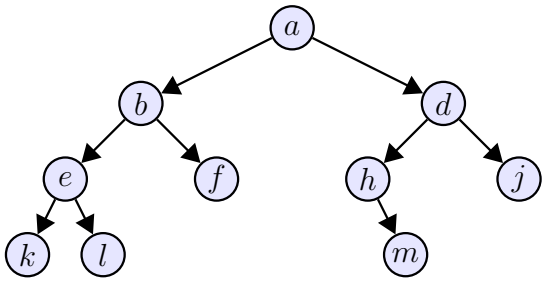
In this case, one has to make sure that the node o has an available right position. For instance suppose o looks like this instead:



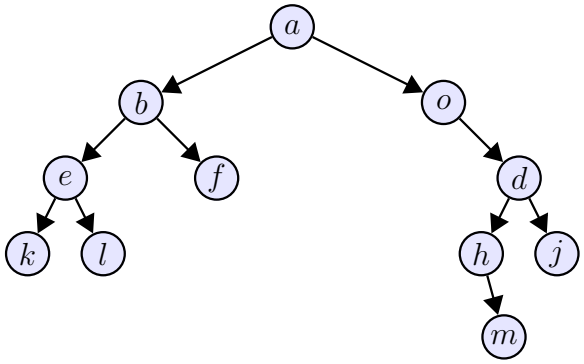
Then we can insert o into the above tree (with root a) between d and j like this:



In the same manner, it shouldn't be too surprising that if we do an "insert parent" o for d in this tree:



that we get this:

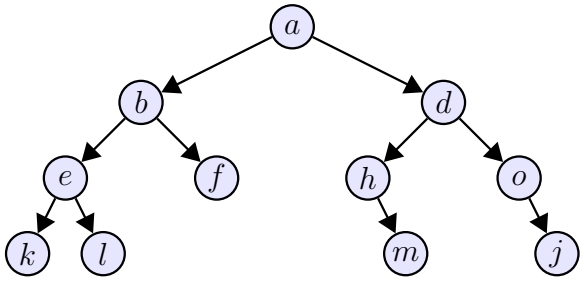


The above also makes sense if node o has a left child.

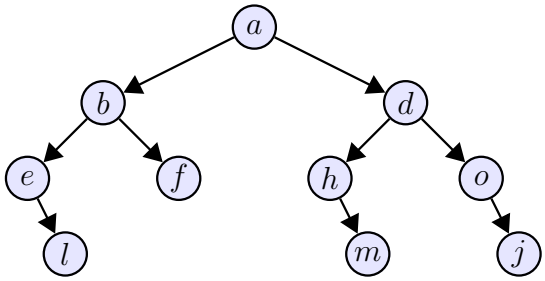
`node.insert_parent(node2)`	make node2 the parent of node and node2 the child of node's original parent
`node.insert(i, data)`	create a new node for data and attach the new node as the i-th child of node. If there's already an i-th child, an exception is thrown.
`node.insert(i, node2)`	similar to above
`node.insert_left(data)`	create a new node for data and attach the new node to the left of node (this is for binary tree). If there's already a left child, an exception is thrown.
`node.insert_left(node2)`	Similar to above
`node.insert_right(node2)`	Similar to above.

For removal, we can remove a single node that is a leaf. I'll call this `remove_leaf`.

Look at this:

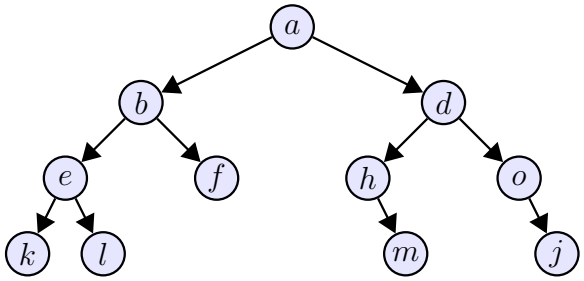


On removing k we get

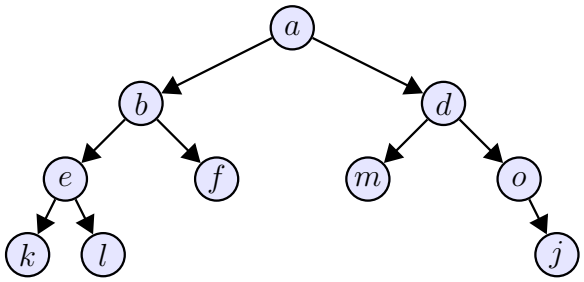


This is the usual node removal operation for a *general* tree.

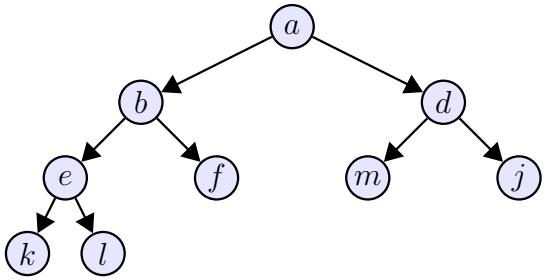
I want to add another node remove operation: Suppose the node to be removed has only *one* child, then there's an obvious way to remove the node. For instance look at



On removing h, we get



This is obviously the same even when m is the root of a subtree. Likewise, if I now remove o, I will get



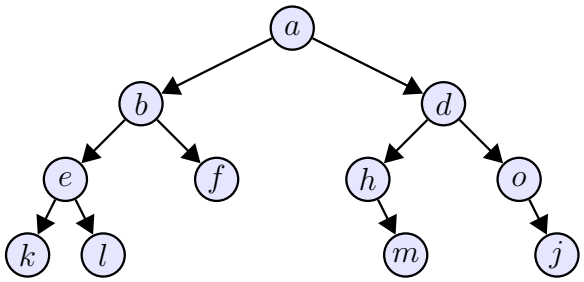

```
node.remove_if_leaf()     remove the node if it's leaf.
                          Otherwise an exception it
                          thrown.
node.remove_if_leaf(i)    remove the i-th child if the
                          child is a leaf.
node.remove_left_leaf()   remove left child if it is a
                          leaf; if the left child is not a
                          leaf, an exception is thrown.
node.remove_right_leaf()  remove right child if it is a
                          leaf; if the right child is not
                          a leaf, an exception is thrown.
```

We can also move the whole tree by specifying the root. Of course this includes the case of removing a subtree by specifying the root of a subtree.

File: height.tex

107.6 Example: height computation

Let me do a simple example. Let's talk about the height. Look at this:



The height of the tree (or the height of a) is 3. The **height** of a is defined to the length of the longest path from a to the leaves. height

Let's think about the height recursively. We break up the tree (at a) into the left subtree L (subtree at b), the root a, and the right subtree R (subtree at d). A longest path from a must either go into L or into R. Let's look at L first.

Of course there's an edge from a into L. If the longest path from a to a leaf actually went into L, then the longest path, after removing the edge from a to b, would also be a longest path for b to a leaf. Clearly if this is the case,

$$\text{height}(a) = 1 + \text{height}(b)$$

Of course it could be the case that the longest path from a actually went into R. In that case, we have the similar fact:

$$\text{height}(a) = 1 + \text{height}(d)$$

By definition height(a) is the larger of the two. So we get:

$$\text{height}(a) = 1 + \max(\text{height}(b), \text{height}(d))$$

It should be clear now that if n is any node,

$$\text{height}(n) = 1 + \max(\text{height}(n.\,\text{left}), \text{height}(n.\,\text{right}))$$

So the function to compute height is probably something like this:

```
ALGORITHM: height
INPUT:     node

if ???:
    return ???
else:
    return 1 + max(height(node.left), height(node.right))
```

Now for the base case. If you want a single-node tree to be the base (look at node k above), then clearly the height of such a tree is 0.

```
ALGORITHM: height
INPUT:     node

if node.is_leaf(): ??? DOES THIS WORK ???
    return 0
else:
    return 1 + max(height(node.left), height(node.right))
```

But you'll see that you get into some nasty ugliness. To be concrete, think of the case of using C++ so that the above function accept `p`, a pointer to node. The problem is when `p->left` might be `NULL` – and our base case does not include this case!!! We stopped earlier, before we reach a `NULL` pointer. This means that I have to do this:

```
ALGORITHM: height
INPUT:     node

if node.is_leaf():
    return 0
else:
    if node does not have a left child:
        return 1 + height(node.right))
    else if node does not have a right child:
        return 1 + height(node.left))
    else:
        return 1 + max(height(node.left),
                       height(node.right))
```

Nothing wrong with that!!! ... but let me show you a cleaner way.

If we include the `NULL` pointer as a tree then we'll need to define the height of an empty tree. Our intuition tells us that `height(NULL)` should be -1. But

the important thing is the recursion:

$$\text{height}(\text{node}) = 1 + \max(\text{height}(\text{node.left}), \text{height}(\text{node.right}))$$

In other words, `height(NULL)` is whatever we want it to be as long as it satisfies the recursion. If we set `height(NULL) = -1` then this makes sense because for instance if you look at node k which should have height 0, if you look at this:

$$\text{height}(k) = 1 + \max(\text{height}(\text{NULL}), \text{height}(\text{NULL}))$$

you do get

$$\text{height}(k) = 1 + \max(-1, -1) = 0$$

You can also verify for yourself that if a node has a left child but no right child, then the above recursion also makes sense.

With this base condition, we have the following when I'm using `NONE` as a sort of fake sentinel node that is attached to a node at the left is it does not have a left child and to the right if the node does not have a right child.

```
ALGORITHM: height
INPUT:     node

if node is NONE:
    return -1
else:
    return 1 + max(height(node.left), height(node.right))
```

The subtree with a node of `NONE` is just the empty tree.

Obviously the C++ code is this:

```
int height(const Node * const p)
{
    if (p == NULL)
    {
        return -1;
    }
    else
    {
        return 1 + max(height(p->left), height(p->right));
    }
}
```

(Why the two `const`? The pointer value does not have and the attributes of

the node does not change.)

What you should take away from the above is this: in terms of recursive–thinking, for tree computations it's cleaner to include the case of an empty tree. In other words we *do* want an empty graph to be a tree too. The definition of a tree should therefore be the following: T is a tree if

- T is an empty tree, or
- T has a root node with children $T_1, ..., T_k$ where each T_i is a tree.

In the case of binary tree this means that a recursion function `F` might have this shape:

```
ALGORITHM: F
INPUT:     node

if node is NONE:
    ...
else:
    DO SOMETHING WITH node, F(node.left), F(node.right)
```

Exercise 107.6.1. Analyze the `size()` method and write down the pseudocode. Implement it in C++. □

Exercise 107.6.2. Analyze the `depth()` function and write down the pseudocode. Of course you have to assume that each node has a parent pointer. Implement it in C++. □

File: copy-constructor.tex

107.7 Copy constructor

Suppose we have a tree T, say the pointer to the root is q and I want to clone T to another tree where the pointer to the root is p. At this point, say p has not been allocated.

```
void copy_constructor(Node *& p, Node * q)
{
    if (q == NULL)
    {
        p = NULL;
    }
    else
    {
        p = new Node(q->key());
        // A: If node has parent pointer, then
        // make *p point to parent node
        copy_constructor(p->left_, q->left_);
        copy_constructor(p->right_, q->right_);
    }
}
```

At A, we have a problem: How do we set *p to point to its parent? Of course if our nodes do not have parent pointers, then we can forget about A.

Method 1. One way is to pass the address of the parent into the function:

```
void copy_constructor(Node *& parent, Node *& p, Node * q)
{
    ...
    else
    {
        p = new Node(q->key());
        p->parent = parent;
        copy_constructor(p, p->left_, q->left_);
        copy_constructor(p, p->right_, q->right_);
    }
}
```

Method 2. The other way is to ensure that the function is clone the child node of q to a child node of p, i.e., p is the parent pointer of the node being

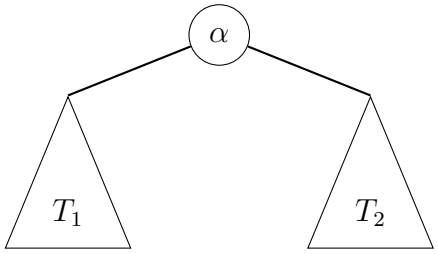
cloned. In this way, the parent is still in scope.

```
void copy_constructor(Node *& p, Node * q)
{
    if (q == NULL)
    {
        p = NULL;
    }
    else
    {
        p = new Node;
        p->key_ = q->key_;
        copy_constructor_(p, q);
    }
}
void copy_constructor_(Node *& p, Node * q)
{
    if (q->left_ != NULL)
    {
        p->left_ = new Node(q->left_->key());
        p->left_->parent = p;
        copy_constructor(p->left_, q->left_);
    }
    if (q->right_ != NULL)
    {
        p->right_ = new Node(q->right_->key());
        p->right_->parent = p;
        copy_constructor(p->right_, q->right_);
    }
}
```

File: destructor.tex

107.8 Destructor/deallocation

Suppose you have a tree T in the heap. Say a pointer `p` points to the root of the tree (with value α in the diagram):



The memory used by the node (with value α) and the subtrees (T_1 and T_2) must all be returned to the heap. Suppose the function is called `deallocate` and the parameter the a pointer to the root.

```
deallocate(p);
```

Of course this would result in

```
delete p;
deallocate(q);
deallocate(r);
```

where `q` and `r` are pointers to the left and right subtrees of node with value α. Therefore the code would look like:

```
void deallocate(Node * p)
{
    if (p != NULL)
    {
        deallocate(p->left_);
        deallocate(p->right_);
        delete p;
    }
}
```

Of course you *cannot* do this:

```
void deallocate(Node * p)
{
    if (p != NULL)
    {
        delete p;
        deallocate(p->left_);
        deallocate(p->right_);
    }
}
```

Do you see why?

The above is for the case where the tree is a binary tree. It's clear what you need to do for a general tree.

Notice that the above uses the postorder traveral on the tree, i.e., we perform a postorder traveral of the tree and perform deallocate the memory used by each node.

Note that if you have a tree class (for instance a binary tree class):

```
class Tree
{
private:
    Node * proot_;
};
```

the destructor can then be

```
class Tree
{
public:
    ...
    ~Tree()
    {
        deallocate(proot_);
    }
...
};
```

You should probably also have a method to clear the tree:

```
class Tree
```

```
{
public:
    ...
    void clear()
    {
        deallocate(proot_);
        proot_ = NULL;
    }
...
};
```

This is sort of like the destructor except that the tree object does not go out of scope and you are in control of calling this method.

File: prefix.tex

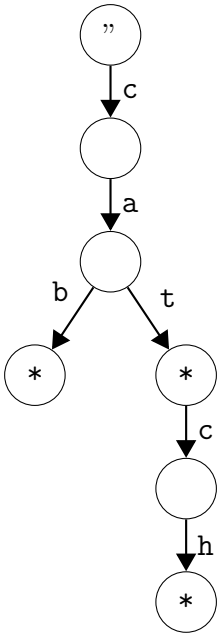
107.9 Application: prefix tree

A **prefix tree** or a **trie** is a tree for storing data with lots of common prefixes. What instance the word `cat` and `cab` share a common prefix of `ca`.

As an example, say I want to store a list of words in lowercase. I use this

```
class Node
{
public:
    char flag;
    Node * v[26];
};
```

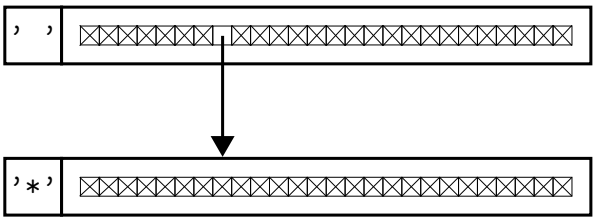
Note that each node has 26 pointers, i.e., 26 edges going to 26 children, each pointer corresponding to one fo the lower case letter. Say I want to store the words `cab`, `cat`, `catch`. Here's the picture:



What this means is that from the root node, say I call it `r`, there's an edge corresponding to a pointer `r.v[2]` (2 corresponding to `c`) that points to a node, say I call it `s`. `s.v[0]` (0 corresponding to `a`) pointing to a node, say I call it `t`. Etc. If the node's flag is `*`, then it means that I've reached a word when I start from the root.

When I draw this this edge between the lowest two nodes:

I mean this:



i.e., each node has a character and a vector of 26 pointers. The node at the top has a character value of `' '` and for the vector of pointers, 25 of them at `NULL` while the pointer at index 7 (which corresponds to character `h`) points to the next node which contains a character value of `*` a vector of 26 pointers, all of which are `NULL`.

Get it? Neat right?

Note that the prefix above is a 26-ary tree since each edge corresponds to one of the lowecase letter. (If you want lower and upper case, then it's a 52-ary tree.)

Exercise 107.9.1. Build a prefix tree for `cab`, `cat` and `catch` using the class above. Do it with this function:

```
void insert(Node ** p, const std::string & word);
```

Write a function that accepts a string (use `std::string`) and checks if the word is stored in the prefix tree. In other words when you call the function with `cat` it returns `true`; if you call the function with `cas` is returns `false`. Here's an appropriate prototype:

```
bool is_found(const Node * const, const std::string &);
```

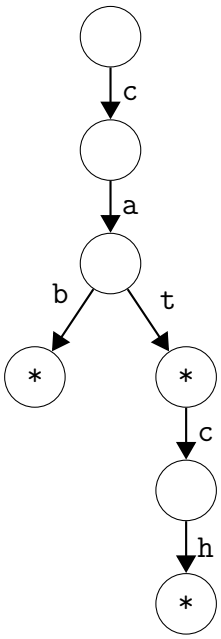
□

Exercise 107.9.2. In case you added a wrong word, you want to be able to

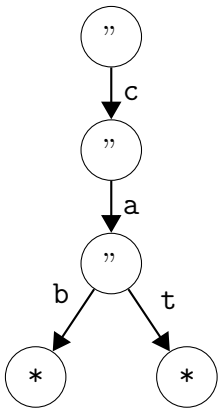
remove a "word". So do this:

```
void delete(Node ** p; const std::string & word);
```

Fail silently, i.e., if you attempt to delete a word not in the prefix tree, don't throw an exception. Also, reclaim memory as much as you can. For instance in the above prefix tree:



if you delete the word `catch`, the memory model of the prefix tree should be



In particular, the node that the edge labeled `t` reaches has a vector of 26 `NULL` pointers. It's not enough just to make the character in the node that the ledge labeled `h` leads to with a space. □

Because words in English (and many other languages too) contains lots of common prefixes, this is a very efficient way of storing words and therefore can be used to create dictionaries.

Not only that, this is also how word autocompletion works inside programs. If you type part of a word into a software, the program can sometimes give you a list of options to complete the word once the number of options is small enough. How does it know? Well, as you type, the program walks down the above tree. From where you stop typing, it can perform a traversal of the subtree where the root corresponds to where you stopped. In the traversal, when it sees a `*` it knows that it's a word (from the root of the subtree to that node). Therefore the program can collect a list of words with the same prefix as what you have typed. The program then shows you the options. Neat right?

Exercise 107.9.3. Continuing the above exercise, write a function that gives you a list of word completion. if you call the function with `ca` it returns an array (use `std::vector`) containing `cat` and `catch`. □

Exercise 107.9.4. Download a dictionary file from the web. Convert all uppercase to lowercase. How many bytes are there in the file. Next, build a prefix tree and then compute the amount of memory used. How much memory do you save? □

Instead of having a vector for pointers where there's one pointer *every* character `a..z`, you can have a vector of (character, pointer) only for characters that are actually used. In that case the root node in the above prefix tree has only one pointer and not 26. However this will slow you down when you look for the word. You can use a sorted `std::vector` to speed up the search. This will of course save you on memory. This is again the story of the life of algorithms: it's always about the balance between speed and memory.

File: bst.tex

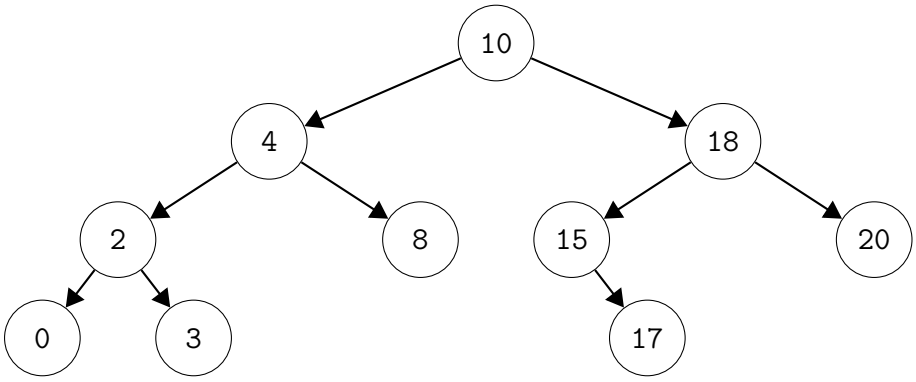
107.10 Binary Search Tree

A **binary search tree** (BST) is a binary tree where every node contains data in such a way that

binary search tree

- T is an empty tree, or
- The values in the nodes of T are distinct and if x is the data in the root of T, then every value in the left subtree of x is less than x and every value in the right subtree of x is greater than x. Furthermore, every node in T satisfies the above property.

Here's a BST:



Of course if you perform an infix traversal of a BST and print the nodes that you visit during the traversal, you will get an ascending sequence of values taken from tree.

Exercise 107.10.1. How many ways are there to build a BST given

- 2 key values?
- 3 key values?
- 4 key values?

□

107.10.1 Search

The point of the BST is to aid in search. For instance if you're search for `17`, you start at the root and see the value `10`. Since the tree is a BST and you're looking for `17`, then if it's in the tree, it must be in the right subtree at `10`. Following the right subtree, we arrive at teh node with value `18`. Since 17 is less than 18, we follow the left subtree and arrive at `15`. Then following the right subtree, we arrive at `17`.

Of course in this case the number of steps needed is the height of three.

Recall that if n is the number of nodes in a BST and suppose the height is h. From

$$h + 1 \leq n \leq 2^{h+1} - 1$$

we see that if the tree has roughly the same number of nodes on the right as on the left for each node, i.e., when the tree is balanced. the search takes

$$n = 2^{h+1}$$

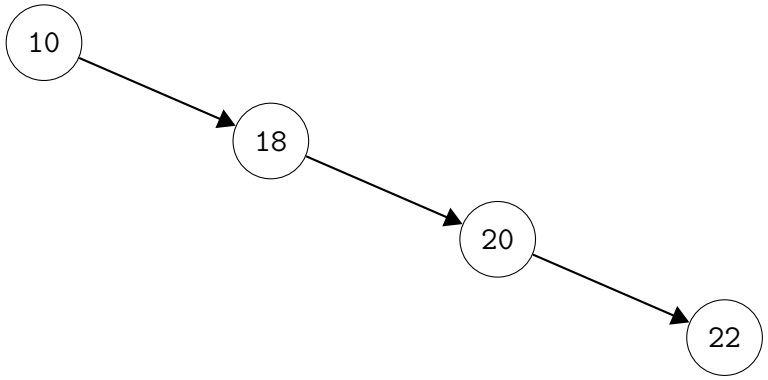
i..e,

$$h + 1 = \lg n$$

In the worst case search when we see that the number of steps to search for a value is

$$h + 1 = n$$

steps for instance when the BST looks like this



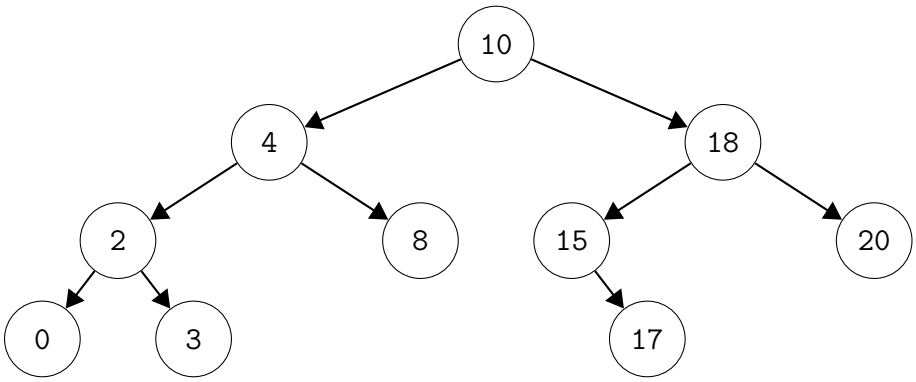
This is the worst case, i.e., the worst runtime for BST search is

$$O(n)$$

The point is that if the tree is heavily "unbalanced", i.e., all the nodes are always "on the left" or "on the right". In the average case, the BST is "roughly" balanced and therefore the average runtime for BST search is

$$O(\log n)$$

Exercise 107.10.2. Given this BST



what nodes were visited while performing a BST search for the following cases: 4, 9, 20? □

Here's the algorithm to perform a search in a BST:

```
ALGORITHM: BST-SEARCH
INPUT: node
       key (the target)

if node is NONE:
    return NOT FOUND

if node.key is key:
    return node
else:
    if key < node.key:
        BST-SEARCH(node.left, key)
    else data > node.data:
        BST-SEARCH(node.right, key)
```

Here's a version that's close to C++:

```
Node * bst_search(Node * p, int key)
{
    if p is NULL, return NULL

    if p->key is key
        return p
    else if key < p->key
        bst_search(p->left, key)
    else key > p->key
        bst_search(p->right, key)
}
```

Exercise 107.10.3. Implement the following C++ function so that it performs a BST search and returns the pointer to the node with the given key value; `NULL` is returned is the key value is not found in the BST.

```
Node * bst_search(Node * p, int key);
```

107.10.2 Insert

Of course when you insert a value into a BST, you have to make sure it's still a BST so that search for it later is still fast. Note that a BST is a self-organizing container.

```
ALGORITHM: BST-INSERT
INPUT: node
       newvalue

if node is NONE:
    set node to a new node with key newvalue
    return

if node.key < newvalue:
    if node.right is NONE:
        create a new node with key newvalue and
        make it the left child of node
    else:
        BST-INSERT(node.right, newvalue)
else if node.key > newvalue:
    if node.left is NONE:
        create a new node with key newvalue and
        make it the right child of node
    else:
        BST-INSERT(node.left, newvalue)
else:
    return ERROR -- DUPLICATE NODE
```

The runtime behavior is similar to the runtime behavior for BST search. The worst case is

$$O(n)$$

The average runtime is

$$O(\log n)$$

In the above, I'm assuming that the key values in the BST are unique.

Exercise 107.10.4. Now C++ pseudocode this time ... implement the following C++ functions

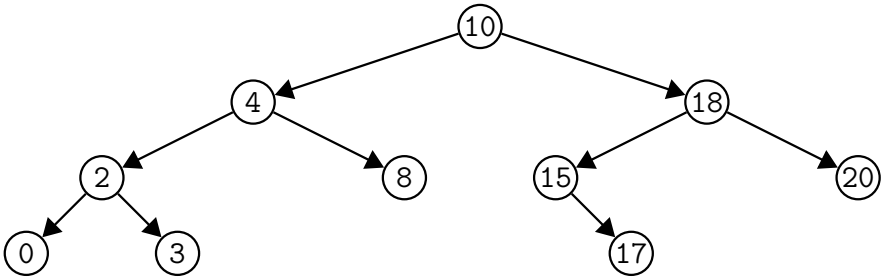
```
void bst_insert(Node ** p, int key);
void bst_insert(Node *& p, int key);
```

(Just for practice, throw an exception when there's a duplicate key.) □

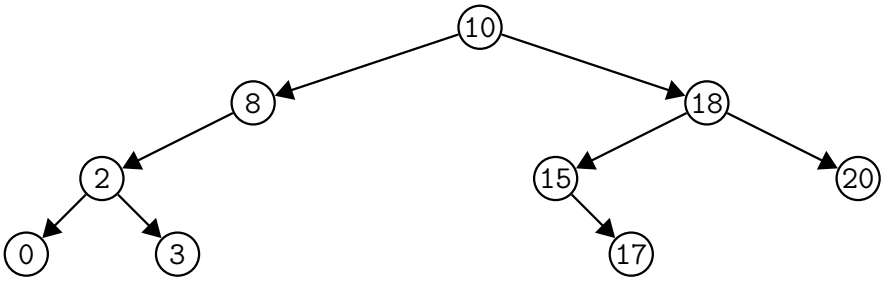
Exercise 107.10.5. The above algorithms are rather similar. Can you find a useful helper function? (Instead of a pointer `q` walking down the tree, you need another pointer `r` that is "behind" `q`.) □

107.10.3 Delete

But what about node deletion? No problem when deleting leaves of course. What about deleting a non-leaf node? Look at this again:



If I delete `4`, maybe we can move `8` up to occupy the position taken up by `4`:



Why is that possible? Because `8` has no left child. If `8` has a left child, when you move `8` up, its left child will "collide" with `2`. Right? Think about this: if `8` has a right child (or even a right subtree) but no left child, you can *still* move `8` up to `4`'s place.

But what if I want to move something up from the left of `4`? For instance, remember that we do want our BSTs to be as balanced as possible. Since there are more node on the left of `4`, maybe we want to move something on the left of `4` up to `4`'s position instead of using some node on the right. How are we going to do that???

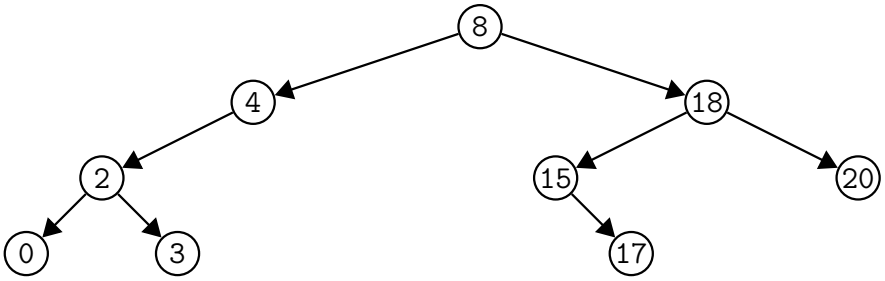
Furthermore, what if I need to delete `10`? You see that it's not so clear what we should do ...

You can of course do an inorder traversal of the tree to create an array/vector/list of the nodes, omitting `10`, and then rebuild the tree. But that's extremely costly!!! We want to disrupt the tree minimally!!!

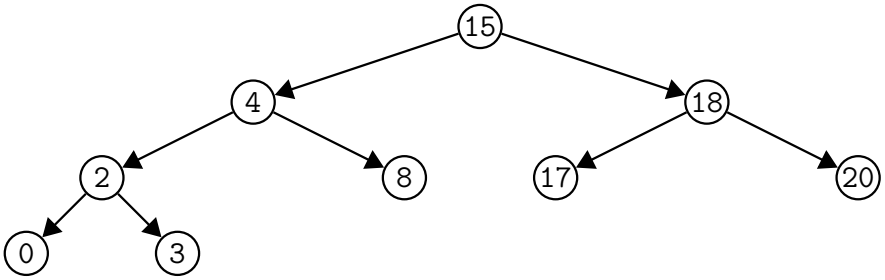
If you think about it, you see that to delete root `10`, you want to you can move the `8` to the same place as the root or move `15` to the root's place. Why?

Two reasons: because they have at most one child and because they are values closest to 10!!! The fact that they have at most one child means that their single child can move to their place without worry about collisions when they are moved to their new places. For instance, see 17 the right child of 15 above. The fact that they are values closest to 10 means that there won't be nodes between them and 10 that might be disrupted because of the move. Right?

In the case of moving 8, we get this



and in the case of moving 15, we get this:



In the second case, the value that is moved, i.e. 15, has a right child 17 which becomes the left child of 18.

Specifically, the value that is moved up to the root is the rightmost of the left subtree or the leftmost of the right subtree. 8 (or the node with 8) is called the **predecessor** of the node with value 10 and the node with 15 is the **successor** of the node with value 10.

predecessor

successor

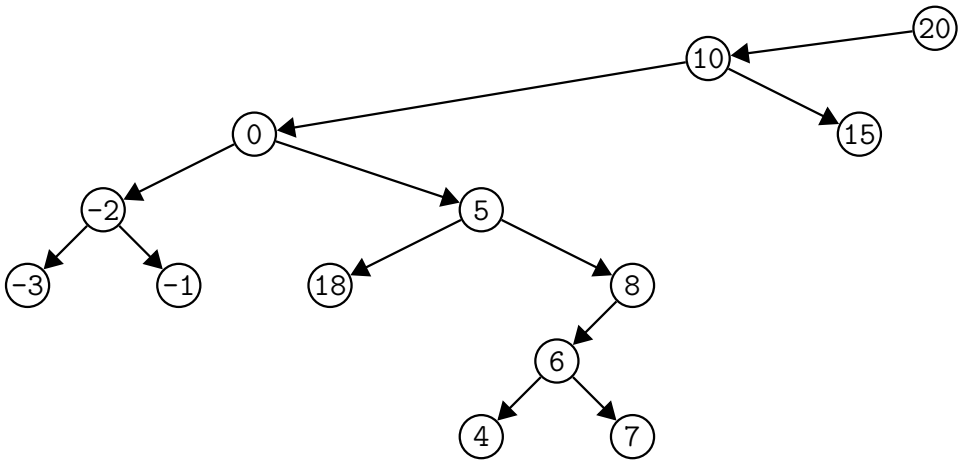
The algorithm for the predecessor is easy: take one left step and take as many right steps as you can:

```
// bst_predecessor of p
if p is NULL:
    return NULL
else:
    let q = p->left_
    while q is not NULL:
        q = q->NULL
    return q
```

I'll leave it to you to write `bst_sucessor` function.

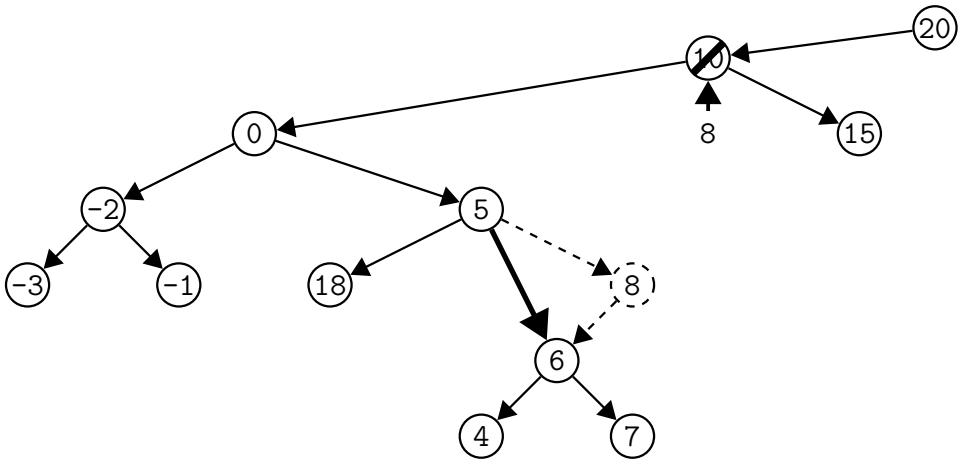
Clearly, since `8` is the rightmost of the left subtree at `4`, it is also the largest value in this left subtree. Therefore moving it to the root will preserve the ordering requirement for a BST. Similarly, moving the leftmost of the right subtree at `18` will preserve the BST property.

There are several details to consider. Consider the following bst where we want to delete 10:



(the same idea below applies to the right of 10 if 10 has a right child but no left child.)

Method 1. We just copy 8 to 10, link the right pointer of 5 to 6, and then deallocate 8:

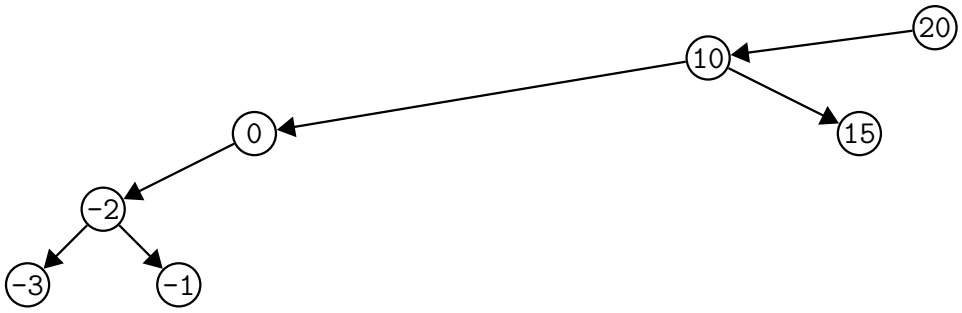


Suppose p points to the node to be deleted and q points to the predecessor. The above is

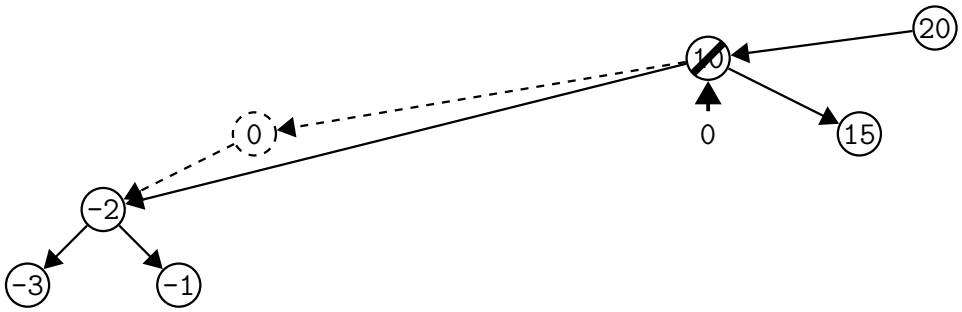
```
// move-predecessor-up algorithm
q->parent_->right_ = q->left_ // point 5 to 6
p->key_ = q->key_             // move 8 up to 10
delete q
```

Exercise 107.10.6. Note that in the above code we used `->` in several places. Can we do that? For instance we used `q->parent_` – can we do that? For instance we used `q->parent_->right_`: can we do that?

Exercise 107.10.7. What if the node with 0 is in fact the predecessor? (i.e., assume the left pointer of the node with 0 is NULL). In other words, in the algorithm above, at `p`, we take one left step and then as many right steps as we can. What if after a left step, we cannot take any right step at all? Here's the picture for this case:



Do you need to change the move-predecessor-up algorithm? The resulting picture should obviously be this:




```
// move-predecessor-up algorithm
if q is not p->left_:
    q->parent_->right_ = q->left_
    p->key_ = q->key_
    delete q
else:
```

After you're done, clean up the algorithm. Hint: It should look like this:

```
// move-predecessor-up algorithm
if q is not p->left_:
    q->parent_->right_ = q->left_
else:

p->key_ = q->key_
delete q
```

Exercise 107.10.8. What if the 8 does not have a left child?

Exercise 107.10.9. What will happen if the node to be deleted has exactly one child? For instance the above algorithm assume that `p` has a non-NULL left pointer (i.e., `*p` has a left child node). But what if `p->left_` is not NULL and `p->right_` is NULL?

The above assume `p->left_` or `p->right_` is not NULL. The case left is where both `p->left_` and `p->right_` are NULL. This is the case where `*p` is a leave node. This is easy: we simply remove the node `*p`. Don't forget, we need to adjust the pointer of `*p`'s parent! You need to set either the left or the right pointer of the parent to NULL. Of course when you arrive at the parent, you wouldn't know where you came from, whether it's through the left or the right pointer. No problem ...

```
// bst delete leaf
let parent = p->parent_
if parent->left_ is p: // *p is on the left of parent
    parent->left_ = NULL
else
    parent->right_ = NULL
delete p
```

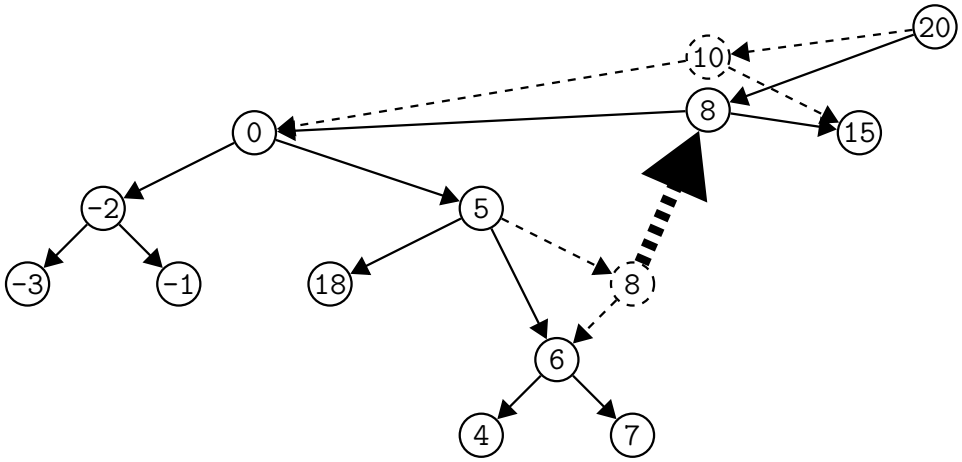
Exercise 107.10.10.

- Warning: The above assume that `p` is really pointing to a node – obviously you can't excecute the code when `p` is NULL.
- Also, if the parent is NULL, the above (obviously) won't work either.
- Correct the bst delete leaf algorithm based on the above points.

Exercise 107.10.11. Let T be the empty bst (of integers). Draw T after each of the following operations:

- insert 10
- insert 15
- insert 12
- insert 5
- insert 0
- insert 3
- insert 4
- insert 9
- insert 1
- insert 6
- delete 10
- delete 6
- delete 5
- delete 12
- delete 1
- delete 3
- delete 4
- delete 15
- delete 0

Method 2. If copying the key value is more expensive than adjusting pointers, then the node with 8 can be take the place of the node with value 10, i.e., the node with value 10 is deallocate.




```
q->parent_->right_ = q->left_ // point 5 to 6
q->parent_ = p->parent_       // move 8 up to 10
q->left_ = p->left_
q->right_ = p->right_
delete p
```

What is the runtime? Again just like the BST search, to reach the predessor or successor of the node you are trying to delete, in the worst case, you have travel along a path to a leaf. So again the worst runtime is

$$O(n)$$

and again in the average case it's

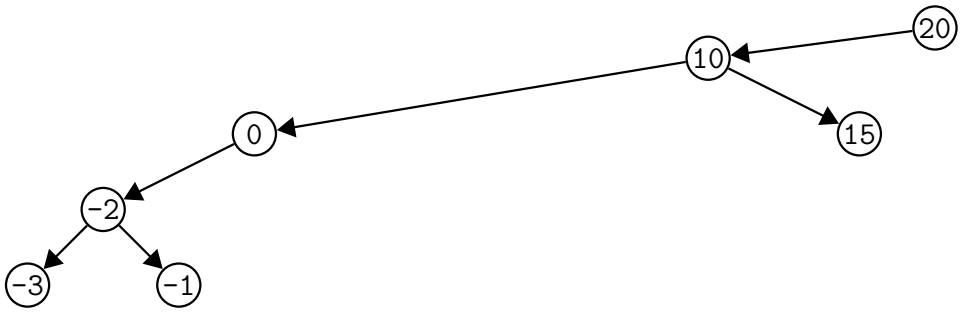
$$O(\log n)$$

since all the above depends on the height of the BST tree.

```
ALGORITHM: BST-DELETE
INPUT: node

if node has a left child:
    perform the operation to move the precedessor
    of the node up to the node's position
    and deallocate the appropriate node
else if node has a right child:
    perform the operation to move the sucessor
    of the node to the node's position
    and deallocate the appropriate node
else: // leaf case
    perform bst leaf delete on the node
```

Don't forget to correct the pseudocode using the exercises and warnings given. For instance look at the leaf case: what if the node does not have a parent – i.e., the node is in fact the root.

Exercise 107.10.12. Now what if at the node to be deleted, there's exactly one child? Can you use this information to get a better delete operation? For instance notice that in this bst, the node with 0 has a left child but no right child:



If you perform the move-predecessor-up, then -1 replaces 0: draw the resulting picture. Can you draw another bst diagram that requires fewer changes to the original bst? Once you have this algorithm, modify the bst delete algorithm so that it has this form:

```
ALGORITHM: BST-DELETE
INPUT: node

if node has a left child but no right child:
    ...
else if node has a right child but no left child:
    ...
else if node has a right child and a left child:
    ...
else:
    ...
```

Exercise 107.10.13.

- For the case where the node to be deleted has a left and a right child, write an algorithm that chooses the predecessor or successor based on the greedy approach: choose one that has the shortest depth from the node to be deleted.
- Is there any reason to choose the longest instead?
- Which of the above two is faster?
- What about flipping a coin? Why would you want to do that?
- How would you modify your BST to include information to better decide on whether to move predecessor or sucessor up (instead of just tossing a coin)?

BST-DELETE algorithm:

Frequently it's convenient to have access to the parent's child point that points to p. For instance if *p is the right child or the parent *(p->parent_), then it's convenient to have a function to return a reference to p->parent_->right_. This can be used for any binary tree (not just bst). This can be easily generalize to any tree.

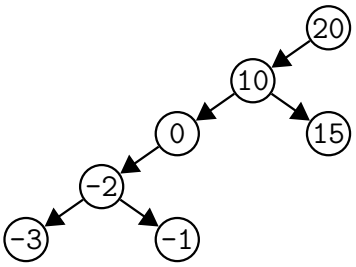
```
ALGORITHM: PARENT-CHILD-POINTER
INPUT:  p
RETURN: reference to left_ or right_ of *(p->parent)

let parent = p->parent_
if parent == NULL:
    return NULL // *p is root
else:
    if parent->left_ is p:
        return parent->left_ (by reference)
    else:
        return parent->right_
```

The following deletes the node *p assuming that it's a leaf. This can be easily generalized for any tree.

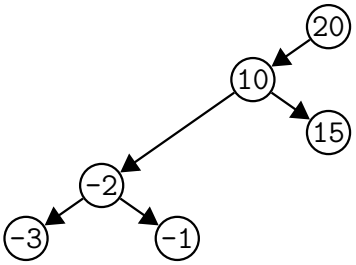
```
ALGORITHM: LEAF-DELETE
INPUT: proot
       p

if *p is root:
    proot = NULL
else:
    PARENT-CHILD-POINTER(p) = NULL
delete p
```

(This is a general algorithm for binary trees, not just BST.)

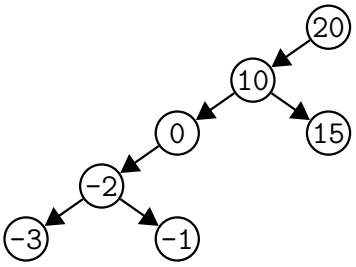
The following cuts out a node n directly from the tree so that the parent of n is joined to either the left or right child of n. For instance if you perform punch through at 0 in the left direction. It is also assumed that p is not NULL and p->parent_ is not NULL.



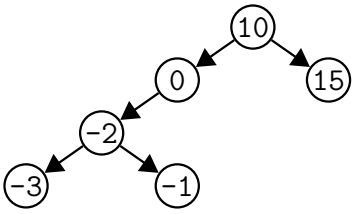
we get



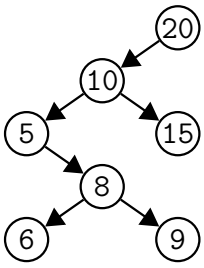
The algorithm deallocates the node that is punched through but does not deallocate subtrees. For instance in the above it's assumed that 0 does not have any right subtree to deallocate. It allows `p->parent_` to be NULL. In that case a child of `*p` becomes the root. For instance if we punch through at 20 in the left direction for the following tree



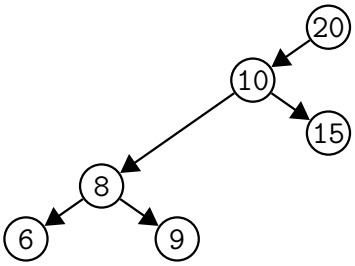
we get



We can also punch through a node and go to the right. For instance in the following,



we can remove 5 by punching through at 5 and go to the right to get this:



Here are the two algorithms for punch-through delete:

```
ALGORITHM: PUNCH-THROUGH-TO-LEFT
INPUT: proot
       p

if p->parent is NULL:
    proot = p->left_
else:
    p->parent->left_ = p->left_
delete p
```

```
ALGORITHM: PUNCH-THROUGH-TO-RIGHT
INPUT: proot
       p

if p->parent is NULL:
    proot = r->right_
else:
    p->parent->right_ = p->right_
delete p
```

This is the version without the punch-through deletion:

```
ALGORITHM: BST-DELETE
INPUT: proot
       p

if p->left_ is NULL:
    if p->right_ is NULL:
        BST-DELETE-LEAF(proot, p)
    else:
        BST-MOVE-SUCC-UP(proot, p)
else:
    if p->right_ is NULL:
        BST-MOVE-PRED-UP(proot, p)
    else:
        # two children case
        if rand() % 2 == 0:
            BST-MOVE-PRED-UP(proot, p)
        else:
            BST-MOVE-SUCC-UP(proot, p)
```

This one uses the punch-through:

```
ALGORITHM: BST-DELETE
INPUT: proot
       p

if p->left_ is NULL:
    if p->right_ is NULL:
        BST-DELETE-LEAF(proot, p)
    else:
        PUNCH-THROUGH-TO-RIGHT(proot, p)
else:
    if p_->right_ is NULL:
        PUNCH-THROUGH-TO-LEFT(proot, p)
    else:
        # two children case
        if rand() % 2 is 0:
            BST-MOVE-PRED-UP(proot, p)
        else:
            BST-MOVE-SUCC-UP(proot, p)
```

Exercise 107.10.14. Write a function

```
bool is_bst(Node * p);
```

that checks if the binary tree at `p` is a BST.

Exercise 107.10.15. Suppose you're given an array of integers which is already sorted in ascending order. How would your construct a BST with the values in this array so that the height of the tree is as small as possible?

Exercise 107.10.16. Range search: Let $a < b$ and T be a BST.

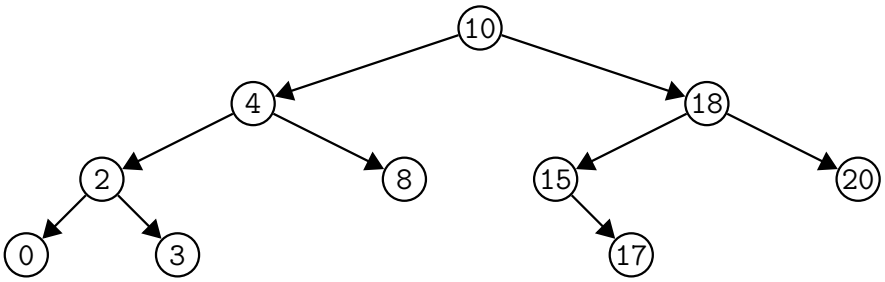
- Write a function that returns true if there is a key k such that $a \leq k < b$.
- Write a function that returns a pointer that points to the node in T with the smallest key k such that $a \leq k < b$. If there is none, NULL is returned.
- Write a function that accepts a pointer p, and a value b and that returns a pointer that points to the node in T with the smallest key k such that `p->key_` $< k < b$. If there is none, NULL is returned.
- Write a function that returns a vector or list of pointers pointing to all nodes in T with values in $[a, b)$. The pointers must be arranged in order of the key values of their nodes.
- Write an iterator class so that that does the same as the above. For instance `BST_iterator p(proot, a, b)` will create the iterator and you iterate through all the pointers of the required nodes by doing

```
while (p != NULL)
{
    std::cout << p->key() << '\n';
    ++p;
}
```

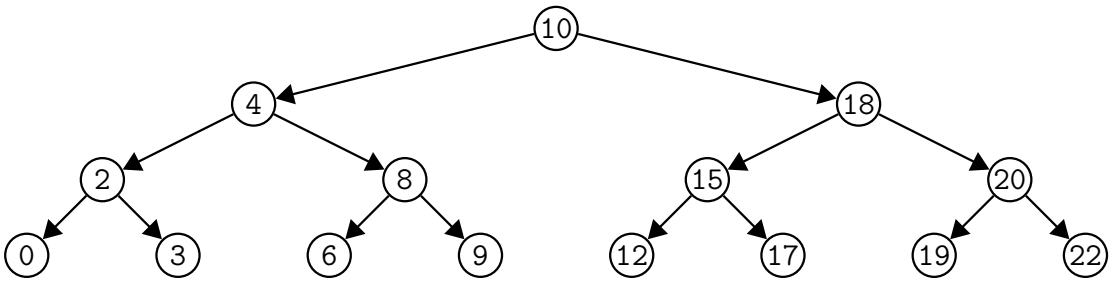
107.10.4 Comparing BST and sorted array

I hope that you see that the BST search if the tree is balanced (i.e., roughyl equal number of nodes on the left and right for each node), the nodes that you visit is similar to the values you would visit in a sorted array with the same values.

Here's our first BST again:



Suppose I make the tree completedly balanced:



Here's another way to store the data, i.e., as a sorted array:

0	2	3	4	6	8	9	10	12	15	17	18	19	20	22

If you search for 6 in the BST above using BST search you will visit the same values if you perform binary search on the sorted array for 6.

Exercise 107.10.17. Perform BST search on the above tree to look for `6` and note down the values you visit. Now do binary search on the sorted array to look for `6`. Maker sure you visit the same values in both cases. Do the same for `0`. □

Same runtime performance!

Remember ... the memory usage of array is not good if the array size is big because you need a huge contiguous chunk of memory to satisfy the memory allocation. That's why we used nodes, forming a linked list. *But* ... the problem with the linked list is that although memory usage is great, binary search is not possible. Search for a value in a linked list is $O(n)$ in the worst case. It's true that in the case of a BST search, if the tree is roughly balanced (which is the average case), the BST search has an average runtime of

$$O(\log n)$$

But that's not the end of the story: Insert and delete for an array (sorted or not) is in the average/worst case

$$O(n)$$

But if a BST tree is balanced (which is the average), the insert and delete are both

$$O(\log n)$$

It does have a worst case of $O(n)$.

Everything is perfect and in the average case, insert, delete, and search are all

$$O(\log n)$$

Meaning to say that average over all BST of size n, the above runtimes hold. `But` ... if you're always working with `one` fixed BST that is heavily unbalanced, then you're out of luck. The question is then ... can we force a tree to be balanced?

File: self-balancing.tex

107.11 Rotations; Self-Balancing AVL BST

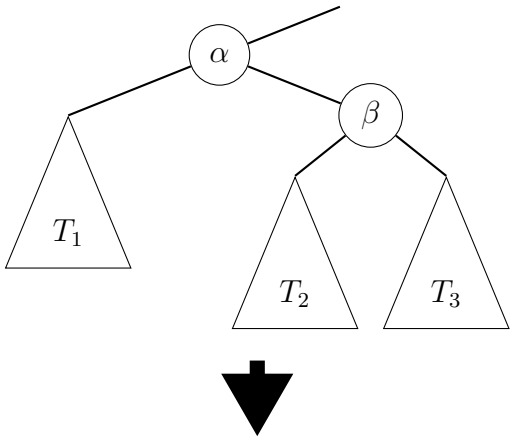
107.11.1 Left and right rotations

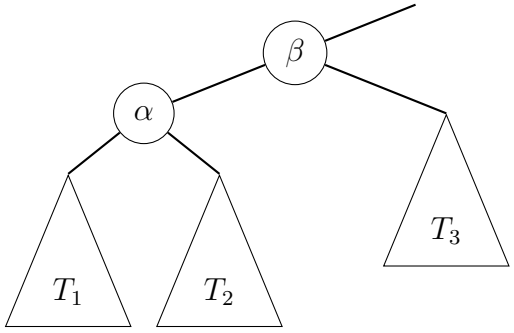
As I said in previous sections, we want to use search trees to store data and we want to search for a particular data very quickly. The number of iterations that must be carried out is at worse the height of the tree. So we want the height to be as small as possible. Which is the same as saying, for the case of a binary search tree, the height of the left subtree is roughly the same as the height of the right subtree.

The sense of being height balanced is a definition not just at the root of the BST, but at every node. Why? Because if the tree is not height balanced, then there are two leaves that will meet at a common ancentor c and the lengths of their path are significantly different. The way to "repair" the the tree is to repair the subtree with root c first. The operation is more or less to more some nodes from the taller subtree of c to the shorter subtree. This is achieved through two operators: the left rotation and right rotation about a node.

You can define left and right rotations about any node. So first I'll describe the left and right rotations and then talk about AVL trees, i.e., height balanced BST.

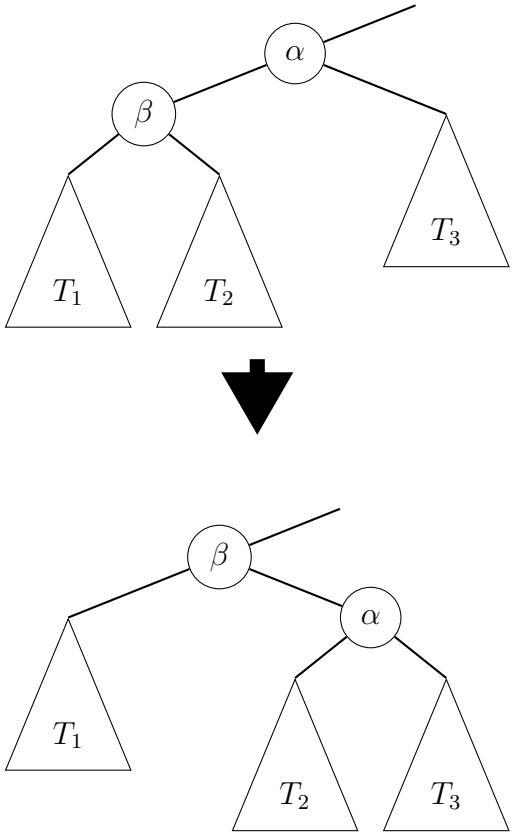
The following describes the left rotation operation about α:





Note that the trees T_1, T_2, and T_3 can be empty. Also, note that to left rotate at α, α must have a right child.

Right rotation is similar. Here's the right rotation about α:

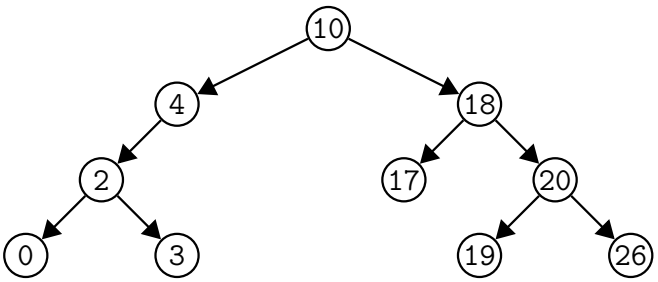


Note that α must have a left child if you want to right rotate at α.

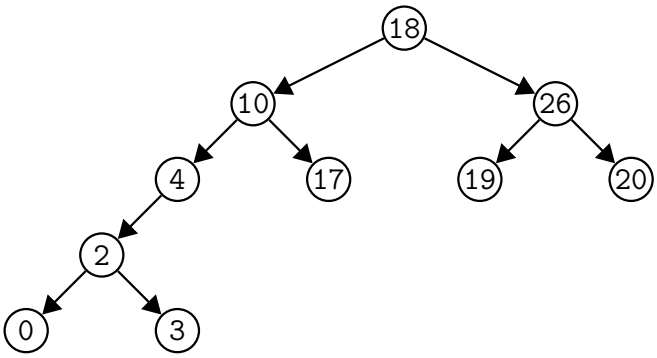
It's clear that the above left and right rotations required constant time.

Here are some examples on left and right rotations. At this point, don't worry about the balance of the tree yet. We'll come to that after you understand the left and right rotations.

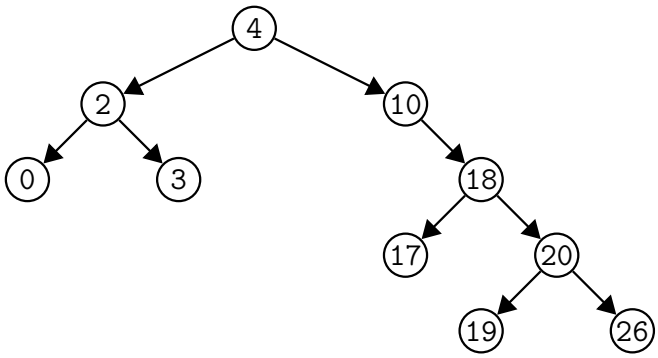
Example 107.11.1. Suppose I have this BST T:



(a): When I perform a left rotation on T about node `10` I get:

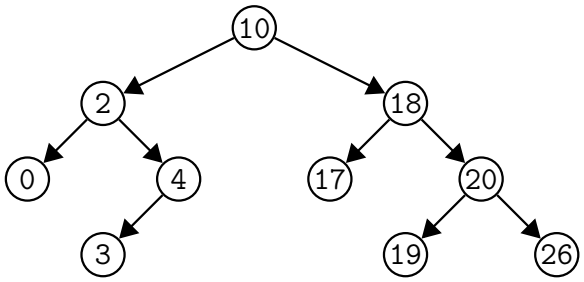


(b): When I perform a right rotation on T about node `10` I get:



(c): I cannot perform a left rotation on T about node 4 because 4 does not have a right child.

(d): When I perform a right rotation on T about node `4` I get:



(e): Perform a left rotation at 18.

(f): Perform a right rotation at 18.

(g): Perform a left rotation at 2.

(h): Perform a right rotation at 2.

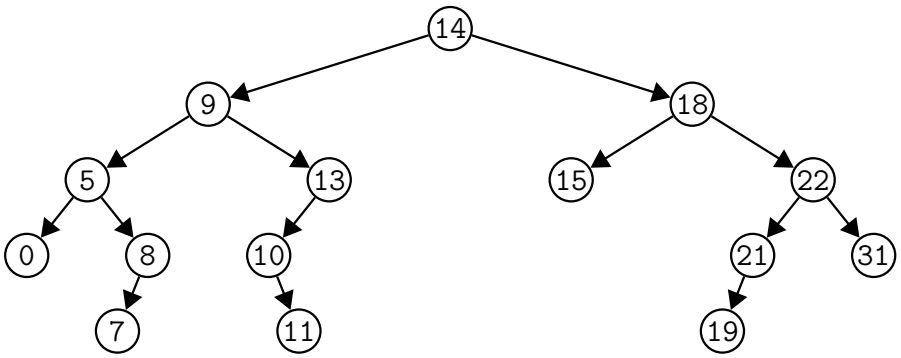
(i): Why can't you perform a left rotation at 17?

(j): Why can't you perform a right rotation at 17?

(k): Perform a left rotation at 20.

(l): Perform a right rotation at 20.

Exercise 107.11.1. You are given the following BST T:



Perform left and right rotation at every node of T whenever possible. [For instance you cannot left nor right rotate at 0; you cannot left rotate at 8, but you can right rotate at 8.]

107.11.2 Algorithms for left and right rotations

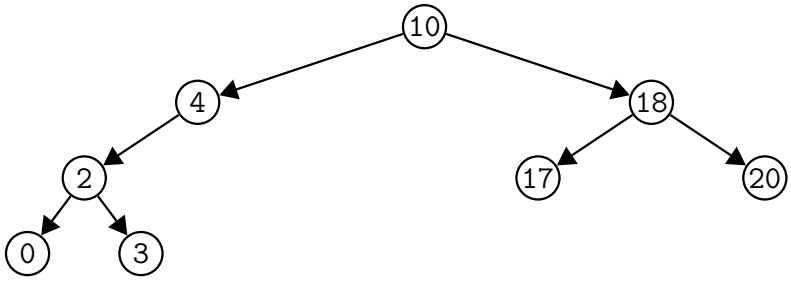
Here are the algorithms for the left and right rotations.

```
ALGORITHM: LEFT ROTATION
INPUT    : p

/* We want to perform left rotation about node (a). p points
to (a). From
                    ...
                     |
            p----> (a)
                   / \
                 T1  (b) <-----q
                     / \
                   T2   T3
we want to obtain this:
                     ...
                      |
                     (b) <-----q
                     / \
           p----> (a)  T3
                  / \
                T1  T2
Note that we assume p and q are not NULL.
*/

    Node * q = p->right()

    // Move (a) down and (b) up
    q->parent_ = p->parent_
    p->parent_ = q

    // Move T2 from (b) to (a)
    p->right_ = q->left_
    q->left_ = p
```

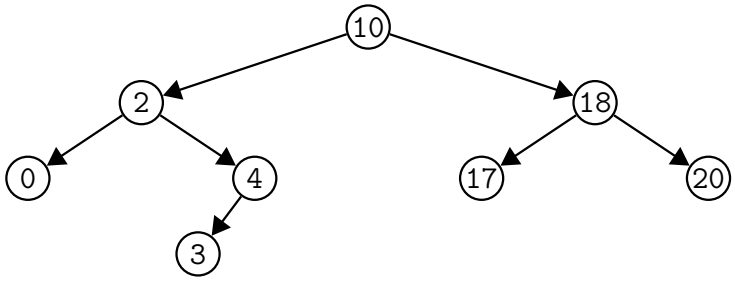
```
ALGORITHM: RIGHT-ROTATION
INPUT    : p

/* We want to perform right rotation about node (a). p points
to (a). From
                    ...
                     |
                    (a) <----p
                    / \
          q----> (b)  T3
                 / \
               T1   T2

we want to obtain
                     ...
                      |
               q---> (b)
                     / \
                  T1  (a) <----p
                      / \
                     T2   T3

Note that we assume p and q are not NULL.
*/

    Node * q = p->left_;

    // Move (a) down and (b) up
    q->parent_ = p->parent_;
    p->parent_ = q;

    // Move T2 from (b) to (a)
    p->left_ = q->right_;
    q->right_ = p;
```

107.11.3 Using rotations to balance BST

Suppose I have this BST:



The tree is clearly not balanced at `4`: the left height us 1 and the right height is -1. I'm going to perform a left rotation about `4`:



Note that the tree is now height–balanced.

107.11.4 Height changes due to rotations

Now let's analyze the general case.

Let T be the subtree tree with root α before the rotation and LEFT(T) be this subtree after. The following are the relevant heights before and after left rotation about node α:

$$\begin{aligned}\text{height}(T) &= 1 + \max(\text{height}(T_1), 1 + \max(\text{height}(T_2), \text{height}(T_3)))\\ \text{height}(\text{LEFT}(T)) &= 1 + \max(1 + \max(\text{height}(T_1), \text{height}(T_2)), \text{height}(T_3)))\end{aligned}$$

or, to make it easier to read, if I write h_i for height(T_i), then

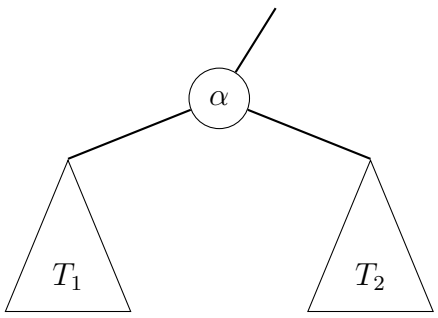
$$\begin{aligned}\text{height}(T) &= 1 + \max(h_1, 1 + \max(h_2, h_3))\\ \text{height}(\text{LEFT}(T)) &= 1 + \max(1 + \max(h_1, h_2), h_3)\end{aligned}$$

So it the heights are $h_1 = 2, h_2 = 1, h_3 = 3$, then

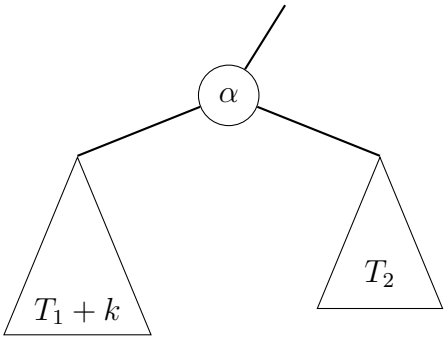
$$\begin{aligned}\text{height}(T) &= 1 + \max(2, 1 + \max(1, 3)) = 1 + \max(2, 4)\\ \text{height}(\text{LEFT}(T)) &= 1 + \max(1 + \max(2, 1), 3) = 1 + \max(3, 3)\end{aligned}$$

For the pre-left-rotation case, you see that the two subtrees of α have heights 2 and 4 and is therefore unbalanced. After rotation, the two subtrees of β has heights 3 and 3 is balanced.

Suppose you have added a node with value k into a BST and it become unbalanced. It should be clear unbalanced nodes (if any) must be along the path that k takes within the tree as it finds its place of insertion. When you go from the inserted new node back to the root, suppose the first node that is unbalanced is α:



CASE LEFT: k is inserted into the left subtree of α.



Of course before adding k, α is balanced:

$$h(T_1) = h(T_2) + \epsilon$$

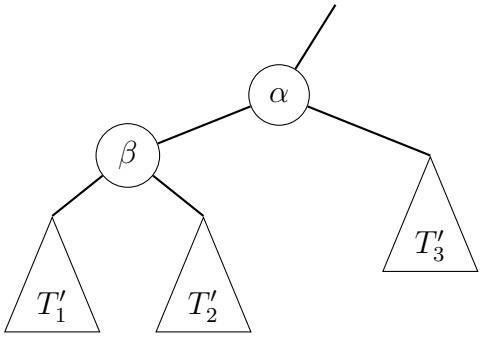
where $\epsilon = -1, 0, 1$. Since α is unbalanced after adding k, we must have

$$h(T_1 + k) = h(T_2) + 2$$

(the 2 on the right can't be more than 2, right?) and

$$h(T_1) = h(T_2) + 1$$

This implies that T_1 must have at least one node. So let me draw the tree, before inserting k, at α like this:



Tying this to the previous diagram, we have

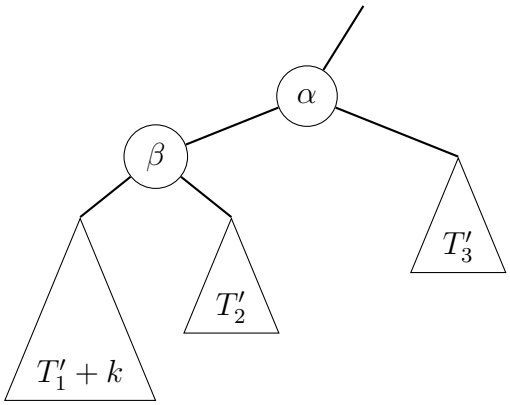
$$1 + \max(h(T_1'), h(T_2')) = h(T_3') + 1$$

i.e.,

$$\max(h(T_1'), h(T_2')) = h(T_3')$$

CASE LEFT-LEFT: k is inserted into the left subtree of the left subtree of α.

Here's the picture:



I claim that

$$h(T_1') = h(T_2') = h(T_3')$$

By our assumption, this makes the tree unbalanced by α. We had this before adding k:

$$\max(h(T_1'), h(T_2')) = h(T_3')$$

On adding k for this case, i.e., to T_1', α becomes unbalanced. Therefore

$$h(T_1' + k) = h(T_1') + 1, \quad h(T_1') \geq h(T_2')$$

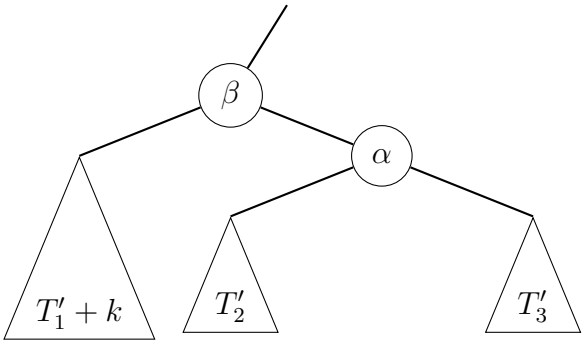
That means that

$$h(T_1') = h(T_3')$$

Note also that if $h(T_1') > h(T_2')$, then when k is added to T_1', the resulting tree is unbalanced at β which contradicts out assumption that the resulting tree is unbalanced at α. Hence we must have $h(T_1') = h(T_2')$. Altogether we have

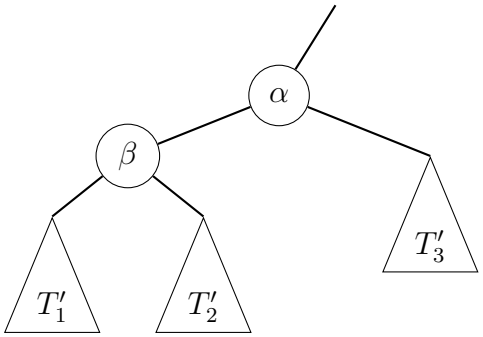
$$h(T_1') = h(T_2') = h(T_3')$$

I right–rotate at α to get this:

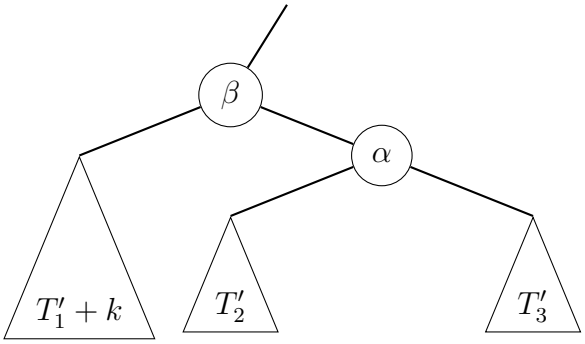


The height of the left subtree of β is $h(T_1') + 1$ and the height of the right subtree at β is $1 + \max(h(T_2'), h(T_3')) = 1 + h(T_1')$. Therefore this resulting tree is balanced at β.

Note for the original tree, before inserting k and when the tree was balanced,

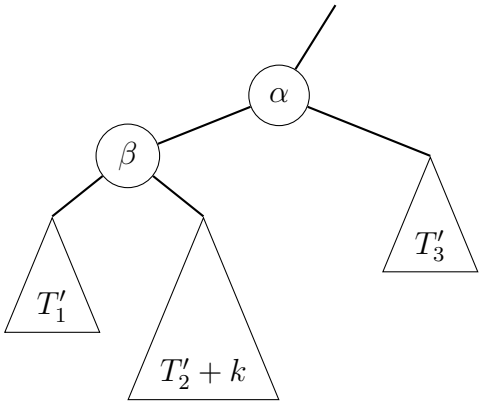


the height at α is $h(T_1') + 2$. After inserting k and the right rotation:



the height at β is also $h(T_1') + 2$. This means that the whole tree must be balanced.

CASE LEFT-RIGHT: k is inserted into the right subtree of the left subtree of α.

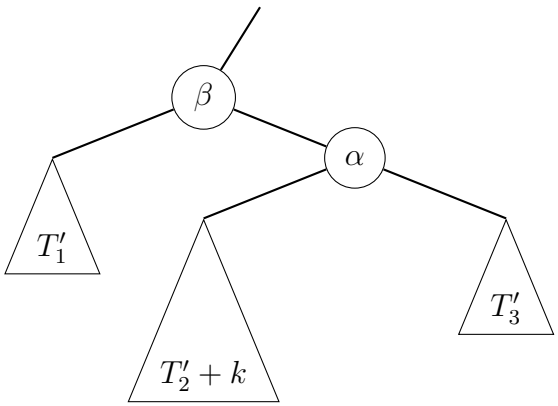


Let's look at $h(T_1')$ and $h(T_2')$. Since β is balanced, they are at most one away. So $h(T_2') = h(T_1') + \epsilon$, $\epsilon = -1, 0, 1$. If $\epsilon = -1$, then T_2' is one shorter than T_1'. In that case inserting k into T_2 will mean that the height of T_2 will either remain the same or increase by 1 which implies that the height at β is unchanged. This means that α cannot become unbalanced after the insertion of k. Also, ϵ cannot be 1 otherwise after inserting k, β becomes unbalanced which contradicts the fact that the α is the lowest node to be unbalanced when k is inserted. Therefore ϵ must be 0: $h(T_1') = h(T_2')$.

Since adding k to T_2' unbalances α, the lowest node of T_3' must be 2 away from k after it's inserted into T_2'. Since T_2' is one below β, we must have $h(T_2') = h(T_3')$.

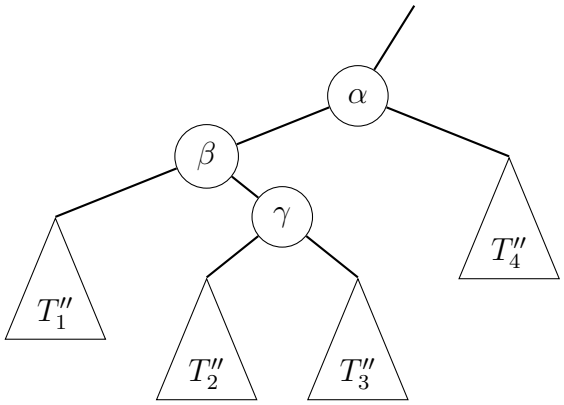
Therefore altogether $h(T_1') = h(T_2') = h(T_3')$.

Look at this:

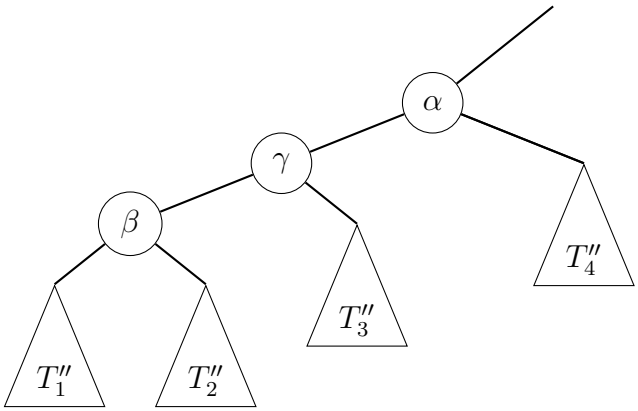


This is NOT the right "rotation". (Find an example on your own to prove what I just said.)

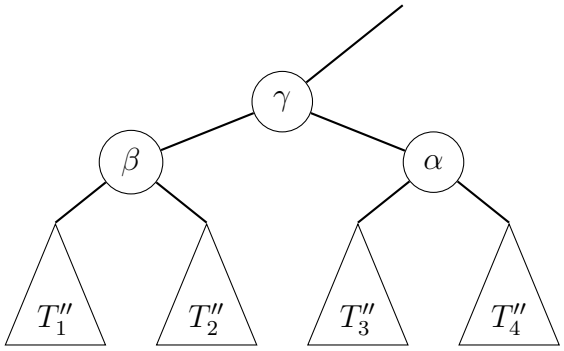
The right thing to do is to do a *left* rotation at β so that β is "heavier" on the left and then then do a right rotation at α. Let me show you. We start here where k is now in T_2'' or T_3'':



After a left rotation at β, I get this:



Now I right rotation at α to get this:



Let me summarize. Suppose after inserting a node k into a BST, α is the lower

node that is unbalanced.

- LEFT-LEFT CASE: If k took a left-left direction from α, then you should do a right rotation.
- LEFT-RIGHT: If k took at left-right direction from α, then you should a left rotation on the left child β of α followed by a right rotation at β which occupies the original position of α.

If should not be too surprising that the mirror image of the above is true: Suppose after inserting a node k into a BST, α is the lower node that is unbalanced.

- RIGHT-RIGHT CASE: If k took a right-right direction from α, then you should do a left rotation.
- RIGHT-LEFT: If k took at right-left direction from α, then you should a right rotation on the right child β of α followed by a left rotation at β which occupies the original position of α.

Draw a picture similar to the above to see what's happening.

107.11.5 AVL trees

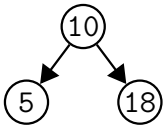
An AVL tree T is a tree that is self-balancing BST in the sense that the insert and delete operations include self-balancing operations (using left and right rotations) so that the tree is balanced after each insert or delete operation.

107.11.6 AVL insert

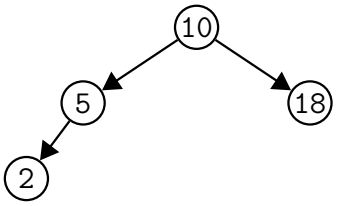
Suppose a node n is inserted into T. Suppose the new tree is T'. There's a path from the root of T' to n. You go up along this path and when you are at a node m that is unbalanced (i.e., the subtree with m as root is unbalanced), we balance at m:

- CASE: left-left. If the path from m along P going down to the new node is left-left, you perform a right rotation at m.
- CASE: left-right. If the path from m along P going down to the new node is left-right, you perform a left rotation one step below m and then a right rotation at m.
- CASE: right-right. If the path from m along P going down to the new node is right-right, you perform a left rotation at m.
- CASE: right-left. If the path from m along P going down to the new node is right-left, you perform a right rotation one step below m and then a left rotation at m.

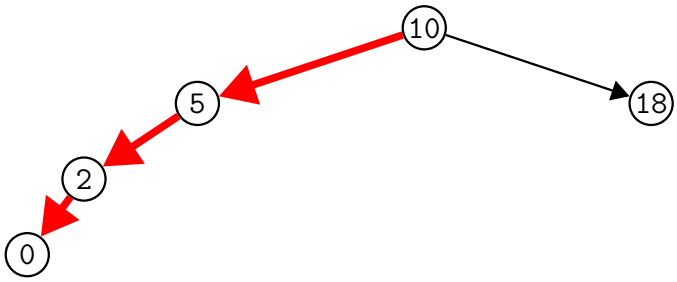
For instance suppose I start off with this:



This is height balanced. After I insert 2, I get:

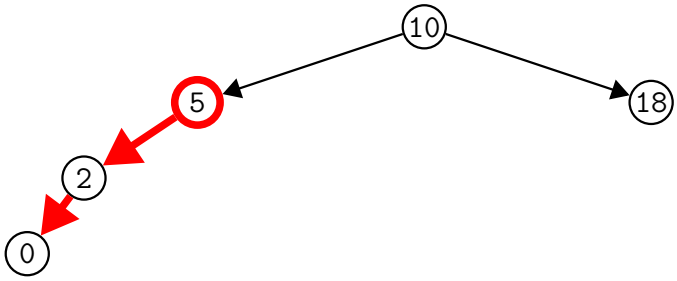


Checking all the nodes along the insertion path, I see that they are all balanced. (Make sure you check that.) Now if I insert a 0, I get this:

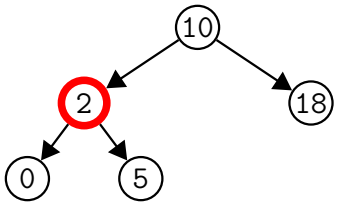


The path of insertion is in red.

When I go back up the tree from the new node, I see that the first node that is not balanced is 5:

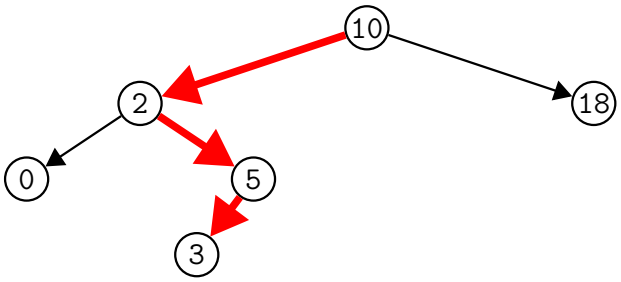


Along this path, at 5, I took two lefts down – we are in the left-left case. Therefore I will perform a right rotation at 5 to get:

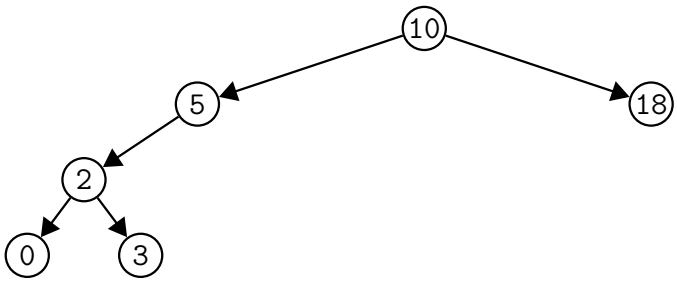


The tree is now balanced at 2 (the previous location of 5). it's also balanced at 10.

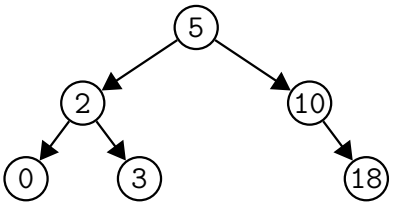
If I insert a 3, I get this:



The part of insertion is shown in red. Going up along the path of insertion, the first node that is unbalanced is 10. At 10, going down, I take a left (at 10) then a right (at 2). Therefore, I will first make a left rotation at 2, and then a right rotation at 10. The left rotation at 2 gives me this:

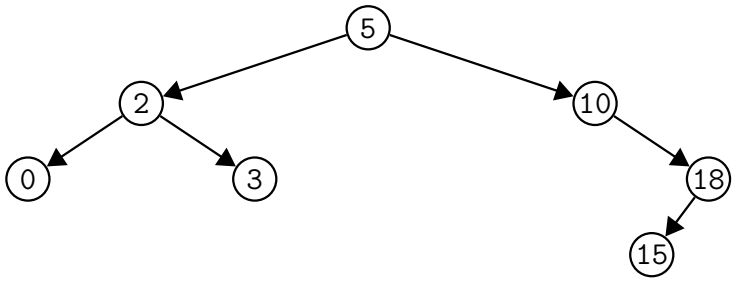


and then I do a right rotation at 10:

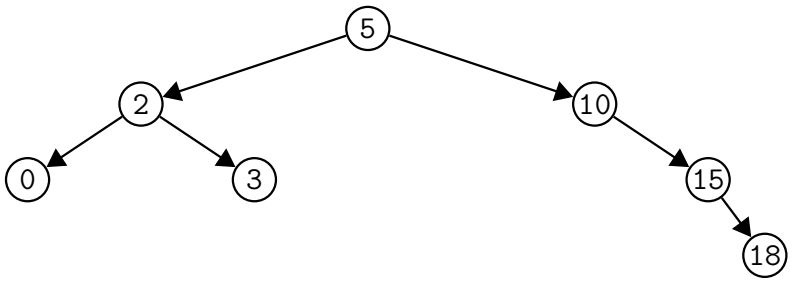


The tree is now balanced.

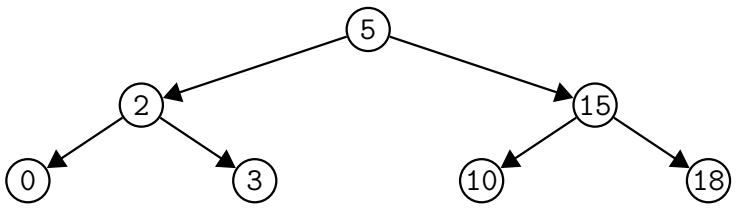
If I insert a 15, I get this:



Going from 15 back to the root, the first node that is not balanced is 10. At 10, going down, I have to take a right (at 10) and then left (at 18). This is the right-left case. So I have to do a right rotation (at 18) and then a left rotation (at 10). On doing a right rotation at 18, I get



and after a left rotation at 10, voila, I get



Note for AVL insertion, once you have found an unbalanced node, you just need to work on that location (doing left or right or left-right or right-left rotation) once. The resulting tree is balanced again. This assumes that you start out with an AVL tree.

The runtime of an insert into AVL is

$$T(n) = O(\lg n)$$

Why? We know that worst runtime of BST insert is based on the height of

the tree. Since the tree is balanced, the height is $O(\lg n)$. After the insertion, we have to walk up the tree, which takes $O(\lg n)$ steps and then perform one or two rotations, which takes $O(1)$ time. Therefore altogether, AVL insertion has a worst runtime of $O(\lg n + \lg n + 1) = O(\lg n)$.

107.11.7 Balance factor

You notice that we need to compute the left height and right height (i.e., height of left subtree and height of right subtree) of nodes and compare which is larger. It's convenient to store height information of a node in the node. I'll leave it to you to figure out how to update the heights of nodes in an AVL tree when a node is inserted into it.

Exercise 107.11.2. Draw some AVL trees and label heights on the right of the all nodes. Next, some inserts and recompute the heights of all nodes. Do you see how to update the heights in general?

But there's another analogous method ...

Note that since we're always comparing which of left or right height is greater, we can simply store

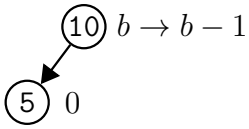
$$\text{(height of right subtree)} - \text{(height of left subtree)}$$

(or "left minus right") in a node. This is frequently called the **balance factor** of the tree. So if the balance factor is $+1$, then the right subtree has a greater height, if the balance factor is -1, then the left subtree has a greater height, and if the balance factor is 0, then the left and right subtrees have the same height.

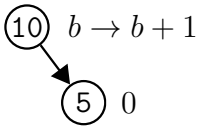
Exercise 107.11.3. Draw an AVL tree and compute the balance factor for each node. Next, perform an insert and see how to balance factors change. Can you figure out how to update balance factors in general? (Draw more pictures if necessary.)

[HINT: Leaf nodes have balance factor of 0 (duh). During an insert, a node acquires a new child. Suppose this is the node (value 10) with balance factor b and the new child is going to be a left child:

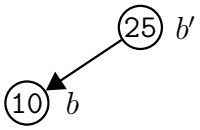
The number b on the right of 10 is the balance factor. The b is either 0 or 1 (it can't be -1 since the new node is going to be attached to 10 as a left child). If b is 0, then 10 does not have a right child and if b is 1, then 10 has a right child. The balance factor of 10 decrements by 1:



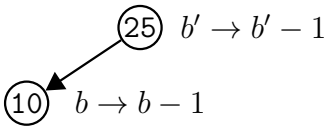
If the new node is attached to 10 as a right child, then we have this update instead:



Of course we update the balance factors on the path from the new node to the root of the AVL tree. So what do we do to the balance factor of the parent of 10? Suppose 10 is a left child of 25 and their balance factors before the insert operation are b and b' respectively:



The update on the balance factors of 10 and 25 is going to be:



In other words, if 10 is a left child and the balance factor of 10 decrements, then the balance factor for the parent also decrements by 1.

All the above is saying is that if the height of the left subtree at a node grows by one, then the left subtree at its parent also grows by one. There's a similar update when the node is a right child: If the node's balance factor increments by 1 and it's a right child, then the parent's balance factor increments by 1.

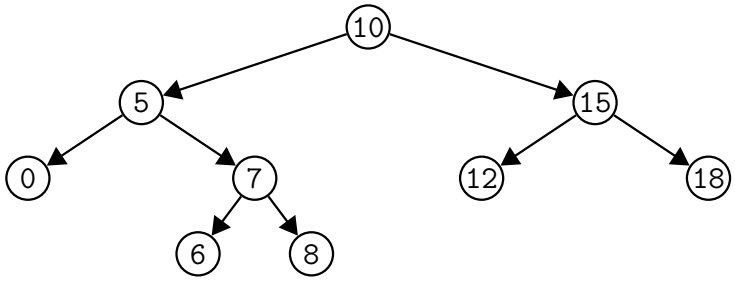
Now what will happen if a new node is attached to node n on the left, but n is a right child?

By drawing more pictures of AVL tree, you should be able to figure out all the cases for the balance factor updates. Of course once a balance factor hits -2 or 2, you balance at that node and update the balance factors of nodes affected.]

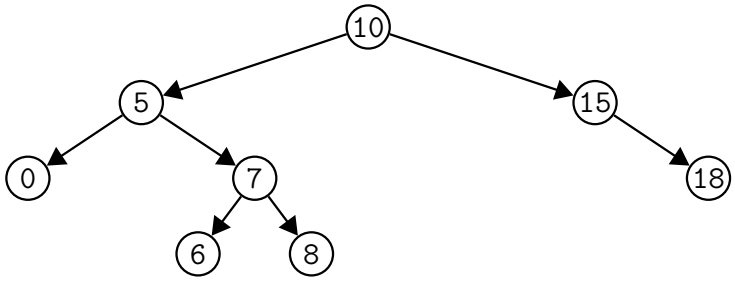
107.12 AVL delete

AVL deletion is just BST deletion followed by rebalancing, just like AVL insertion. The main difference is that, whereas for insert when one rebalancing act is enough (left or right or left-right or right-left), for delete, you might need to rebalance more than once.

Suppose I have this AVL tree:

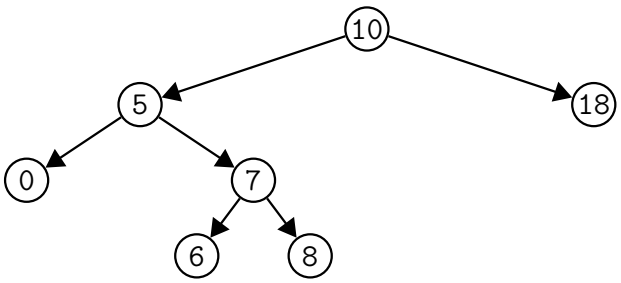


(This is balanced.) If I delete 12, I get this:

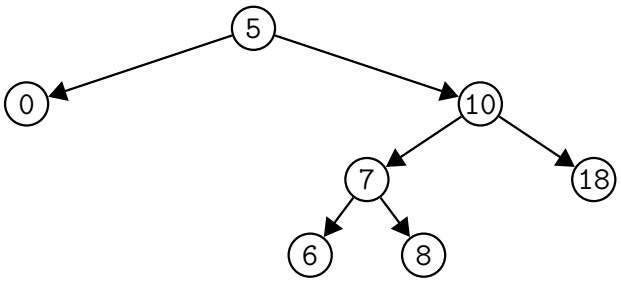


which is again balanced – I just need to check along the deletion path, from the root to the parent of 10.

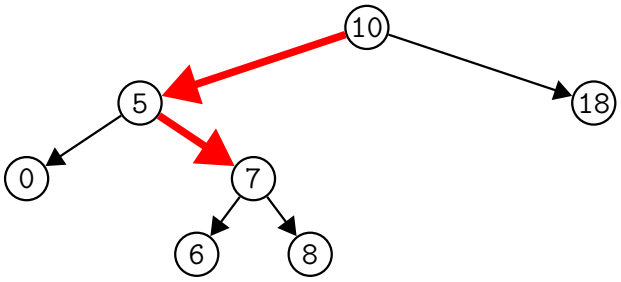
If I delete 15, I get this:



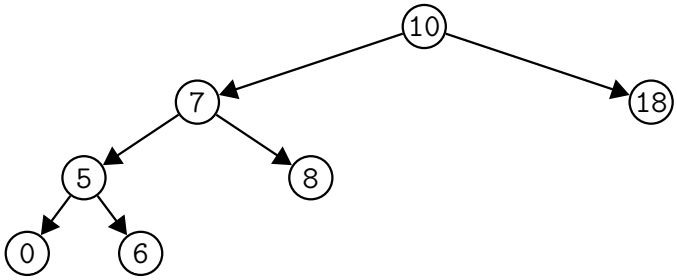
Going from 18 back to the root, I see that 10 is not balanced. Clearly the left subtree of 10 has a height that is too large. Now, if I perform a right rotation at 10, I get



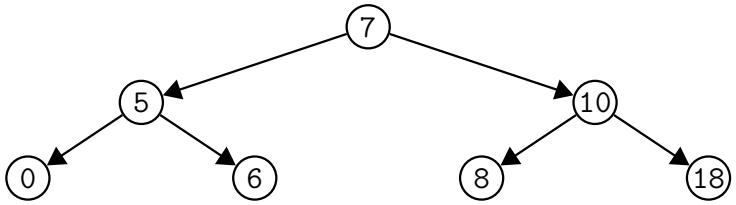
This does not work! The same location (where 10 was and is now occupied by 5) is still unbalanced!!! The reason is because the path $10 \to 5 \to 7 \to 6$ (or 8) is the one that is causing the unbalancedness at 10:



Since $10 \to 5 \to 7$ is a left-right, we use the same idea from AVL insert and perform a left rotation at 5 and then a right rotation at 10. After a left rotation at 5, we get



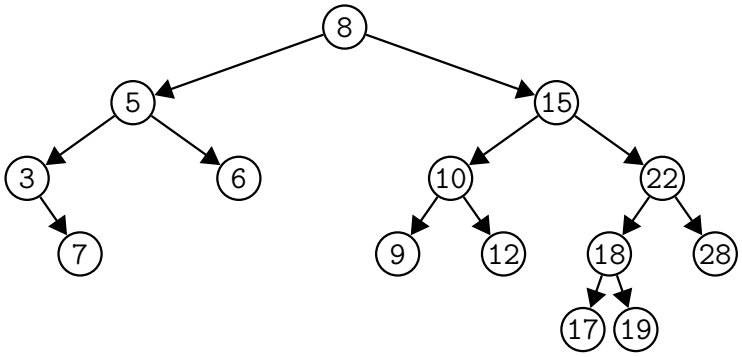
And after a right rotation at 10, we get



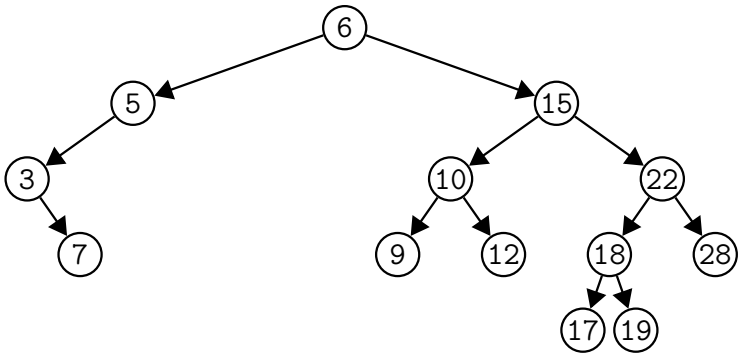
Now node 7, the previous location of 10, balanced. Since 7 is now the root, we stop. If 7 is not the root, we have to keep going up and checking for unbalanced nodes. In general, you want to work your way up, starting with the parent node of the node that was *physically removed*. This is *not* necessarily the node you are deleting. For instance recall that in some cases, you need to delete the predecessor or successor of a node. So in these cases, you start at the parent of the predecessor or successor of the node and go up to the root.

Here's an example where after a delete, we need to rebalance at two points.

Suppose you want to delete 8 from this tree, using the move-predecessor-up:



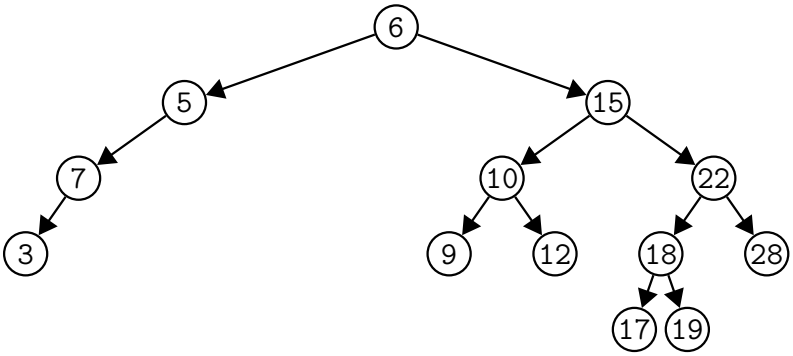
we get



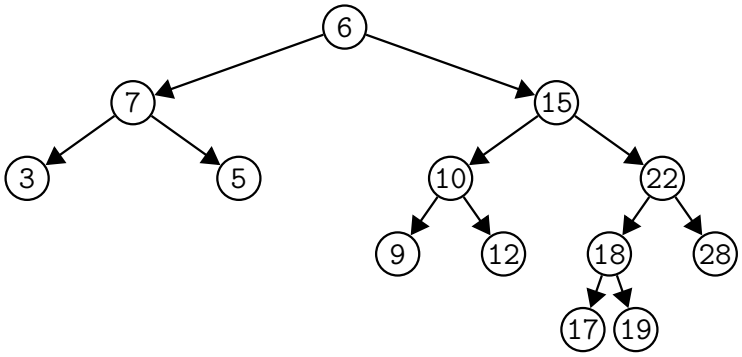
Now for the balancing. We do *not* start at the node with value 6 – where 8 was. Rather, *we start at 5*, i.e., the parent of the predecessor of 8 that was removed

(or moved) in the case of move-predecessor-up or the parent of the sucessor of 8 that was removed (or moved), depending on which one was moved.

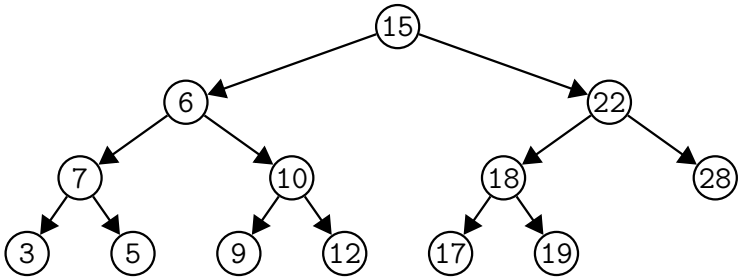
We see that 5 is not balanced. The left subtree of 5 has a larger height than the right. Following two steps of the longest path in the left subtree (in this case the only path), we see that the steps are left-right. So we perform a left rotation at 3:



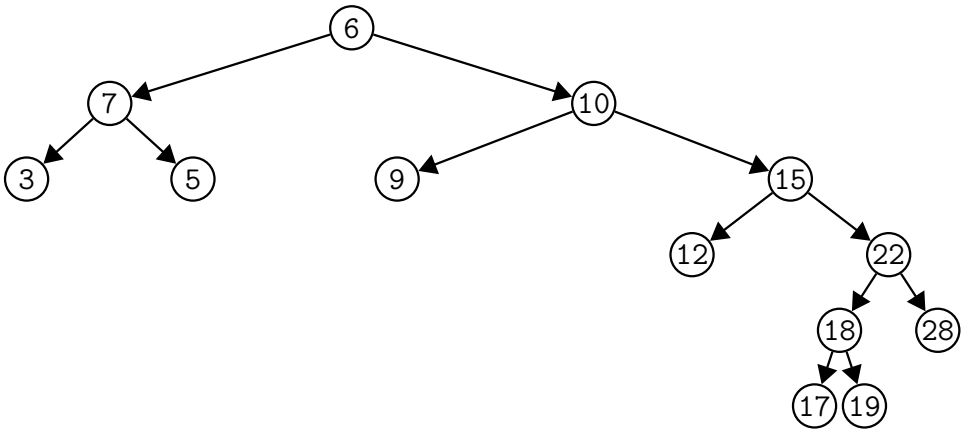
and then a right rotation at 5 to get:



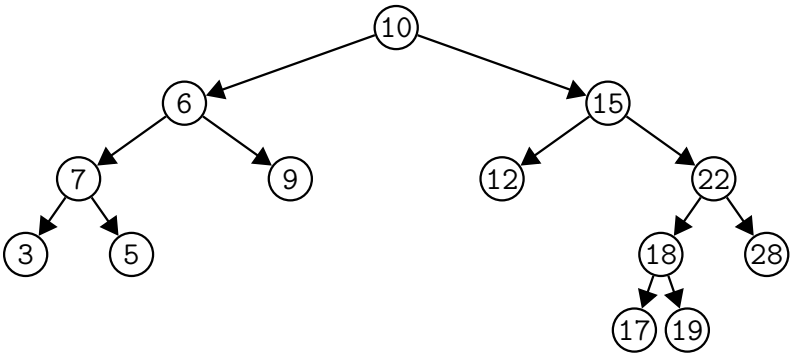
The location we were balancing now has value 7 (it was previously 5). Now we move up to 6. 6 is not balanced. Following two steps of the longest path into the right, i.e., $6 \rightarrow 15 \rightarrow 22$, which is the right-right case, we perform a left rotation at 6 to get:



If we followed right-left at 6 (which is incorrect because this follows the shorter subtree!!!), then we would have performed two rotations, first a right rotation at 15 to get:



and then a left rotation at 6 to get:



This results in an unbalanced node at 15. So in general, you want to follow the node which is unbalanced (as much as possible) to determine which of the left-left, left-right, right-left, right-right case for balancing you want to execute. In terms of balance factor, you want to choose $+1$ (right is taller – go right) or -1 (left is taller – go left) whenever possible.

Exercise 107.12.1. Starting with an empty AVL tree, perform the following:

- AVL-insert 6
- AVL-insert 5
- AVL-insert 4
- AVL-insert 3
- AVL-insert 2
- AVL-insert 1
- AVL-insert 7
- AVL-insert 8
- AVL-insert 9
- AVL-insert 10
- AVL-insert 11
- AVL-insert 12
- AVL-delete 4
- AVL-delete 3
- AVL-delete 2
- AVL-delete 1
- AVL-delete 5
- AVL-delete 7
- AVL-delete 6
- AVL-delete 8
- AVL-delete 11
- AVL-delete 12
- AVL-delete 9
- AVL-delete 10

Exercise 107.12.2. Draw all BSTs with values 0, 1, 2, 3, 4. How many are AVLs?

Exercise 107.12.3 (Merge)**.** Given two BSTs, T_1 and T_2, construct a BST T containing all the values in T_1 and T_2.

Exercise 107.12.4.

1. Design an experiment to compare the performance of insert, delete, find on BSTs and AVLs.
2. Do the same as above, but now include binary search trees which are like AVL but the delete operation does not rebalance.
3. Do the same as above, but now include binary search trees which are like AVL but the delete operation will only rebalance at most once.

Exercise 107.12.5. A k-AVL tree is basically an AVL where nodes are allowed to have the absolute value of their height balance factor $\leq k$ without balancing. Once the factor is $> k$, it is balanced. So a 1–AVL is our regular AVL. Study this type of trees. Start with 2–AVL trees. □

Exercise 107.12.6. Study the following BST. When a node is initially create, it have a counter of 10. Whenever a node is visited during a search, insert, delete, the counter drops by 1. Accessing a node durng balancing does not decrement the count. After an insert or delete, excluding new nodes during insert, if the starting node has a counter of 0, rebalancing occurs with balance factors updated and counters of nodes on the balancing path with counter set to 10. □

File: btree.tex

107.13 k-ary trees and B-trees

Recall that a k–ary tree is just a tree where node can have at most k children. So a binary tree is a 2–ary tree. Recall that the height of a binary tree of size n is in the best case

$$\log_2 n$$

and in the worse case

$$n$$

In general if you have a k–ary tree, in the best case the height is

$$\log_k n$$

and n in the worse case.

It's not difficult to imagine that the same idea used in a binary search tree can be applied to a k–ary search tree. For instance, in a 5–ary tree would look like

The first leaf has keys in $(-\infty, 5]$, the second has leaves with keys $(5, 10]$, the third has leaves with keys $(10, 25]$, etc.

Refer to CISS430 for more information on this variation of the binary search tree.

File: quadtree.tex

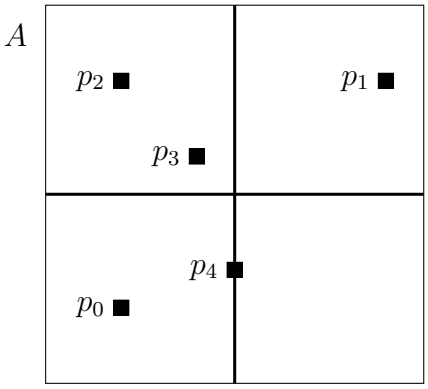
107.14 Quadtrees

Quadtrees is a type trees used in situations where you want to say something about closeness of two things. These (and many other related data structures) can be used in games for collision detection and they are also used in geospatial databases that is used in geographic information systems. I'm go focus only on quadtrees – there are other data structures for spatial indexing.

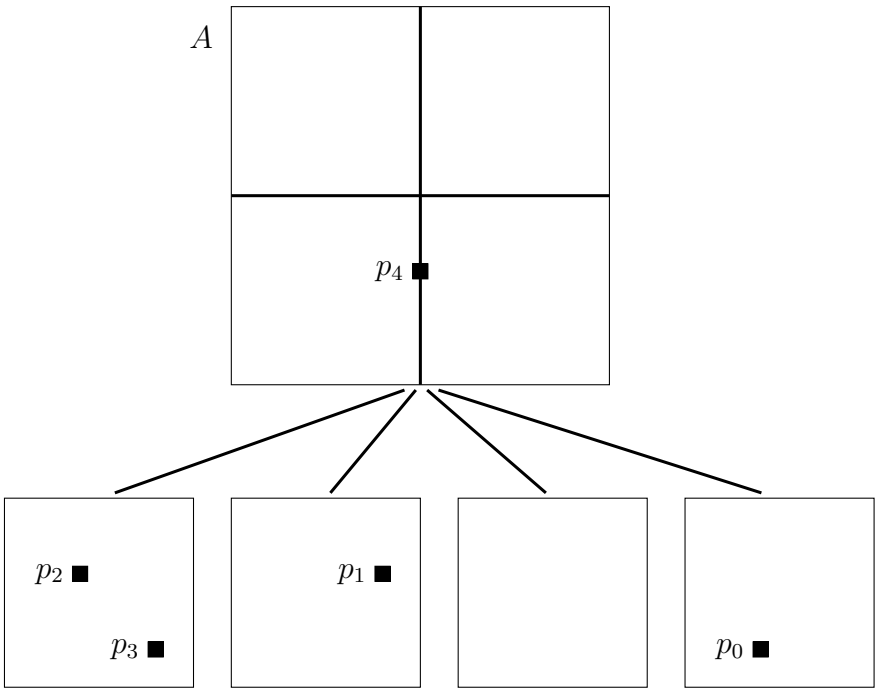
Suppose you have a collection of objects in a 2D space. One common question to ask is when a pair (or more) of these objects collided. For simplicity, you can think of these objects are rectangles (or circles, etc.) The simplistic way for pairwise collision detection is to perform a double for-loop on the objects and check when they collde. This has a runtime of $O(n^2)$.

Here's another way: You think of a 2D area of your game (the whole window) as a container of your game objects. Suppose the window area is $[0, 1000] \times [0, 1000]$ and there are 5 game objects:

If there are too many game objects, say more than $t = 4$, then you divide your window into 4 equal size subwindows and distribute the game objects into the appropriate subwindows of the game object is completely in a subwindow. If the game object is not completely in a subwindow, it stays in the parent window.



Conceptually, as a tree,



Once A has children, when a game object is placed in A, if it is completely in one of the four subwindows, it is sent in the relevant subwindow for recursive insert. If the game object intersects the boundary (look at the cross in the window of A), then it stays in A.

Once a subwindow (again) has more than t game objects. you give this subwindow four children and repeat the distribution of game objects into the children of the subwindow.

Once every game objects is distributed, for the computation of collision detection, you only need to check among the game objects in each leaf subwindow, except that there might be game objects in the parent node.

t is called the **threshold** of this quadtree.

To prevent building a tree that's has too many nodes, you can also include a **maximum depth** in the nodes. Say you want to limit the tree to maximum depth of 10. Then once a node has reached a depth of 10, the node will not expand into children nodes. (Recall that the root has a depth of 0.)

Here are some obvious member variables each node of the tree should have:

- The geometry of the window: `minx`, `maxx`, `miny`, `maxy`
- Threshold (the t from above)
- Depth of this node
- Maximum depth
- Array of pointers to game objects (it's probably best to use `std::vector`, otherwise you would also need to have a size variable)

Note that the threshold and maximum depth are both applied throughout the whole tree and therefore should probably be in the quad tree and not the quad tree node. Note that, of course, the quadtree must have access to the geometry/shape of the game objects to know which subwindow a game object fits into. It's convenient to have the following in a quad tree node:

- A method to return the index of the child a game object should go into.
- A boolean member variable that tells you if the node has children.

A quad tree object has a pointer to the root node. Besides that, you might also want the threshold and the maximum depth in the quad tree. Obviously methods includes a destructor, copy constructor, `operator=`. Besides that, of course you need to insert (the pointer of) game objects into the tree. Once all the (pointers of) game objects are placed in the quad tree, you can start computing potential collisions groups. This is done in two different ways:

- Either get the quad tree to return a collection of *all* colliding groups. The return type is `std::vector< std::vector< GameObject * > >`. (In general, a collision involves not just a pair of game objects. Depending on what you want to achieve in your game, in some cases, it is sufficient to compute *pairs* of colliding game objects.)

- Or you might want to get a collection of game objects colliding with *a given game object*. The return type is `std::vector< GameObject * >`.

A quad tree has many uses besides game collision detection. It is for instance used in computer graphics, image processing, robotics, etc. (which is natural since a quad tree is a spatial data structure).

[Note: Instead of rebuilding the whole quad tree for each game loop, you can also update the quad tree. If the the space is each cell is sufficiently large, the game object will usually not change from one cell to another. For this case, you would need to know if the (pointer of the) game object has already been inserted into the quad tree or not. If it's already in the quad tree, then it's an update operation. However, the location of (pointer of) the game object in the quad tree depends on the position of the game object during the last insert/update. Therefore (and this is very important), each game object that

is updating its location in the quad tree should contain the previous position and the current The previous location allows you to find that game object in the quad tree. Once the game object is found, if the current location does not require the game object to move to a different quad tree node, then nothing is done. Otherwise it is deleted and then re-inserted into the quad tree. Deleting of nodes in a quad tree is more complicated – if the number of pointers in all children of node n plus all the pointers in n is $\leq$ threshold, then all these pointers can go to n and all the children nodes can be deleted. However, it's possible that all these nodes might be re-created again some time in the future.]

Exercise 107.14.1. Based on what was described above, when computing collisions, do you need to check a node with it's grandparent?

Exercise 107.14.2. The quadtree is for subdividing a 2D space (a rectangular area). The same idea can also be applied to an octtree where a cube is divided into 8 equal sub-cubes in the obvious way. Implement an octtree.

107.15 kd trees

Index