SWEN 601
Software Construction

Heaps, & B-Trees

Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
    a. *The project **<u>may</u>** already contain some code!*
2. Create a `session` package. This is where you will write your solutions to today's activities.
3. Create a `homework` package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

<u>Do not</u> submit code that ***does not compile***. Comment it out if necessary.

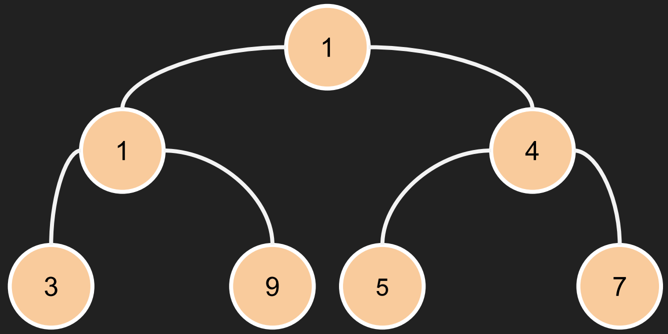
Next Two Weeks

WEEK 10	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #14		Quiz #15		
LECTURE			Binary Trees & Binary Search Trees		Heaps, N-ary Trees, & B-Trees		
HOMEWORK	Hwk 14 Due (**11:30pm**)		*Hwk 15 Assigned*		*Hwk 16 Assigned*	Hwk 15 Due (**11:30pm**)	

WEEK 11	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #16		Quiz #17		
LECTURE			Graphs		BFS & DFS		
HOMEWORK	Hwk 16 Due (**11:30PM**)		*Hwk 17 Assigned*		*Hwk 18 Assigned*	Hwk 17 Due (**11:30pm**)	

Heaps

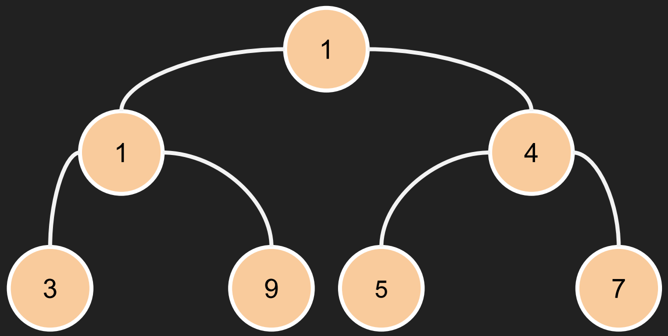
- A ***heap*** is a special binary tree that is similar to a binary search tree.
- Each node in the heap has a value that is an ***equal or lower priority*** than the value of its ***parent***.
    - The exception of course is the root, which has no parent.
    - The root always contains the ***highest priority*** value.
- The relative order of the nodes (i.e. from left to right) is arbitrary.
    - In other words, there is no horizontal relationship between nodes; the left ***or*** the right child may be higher priority.



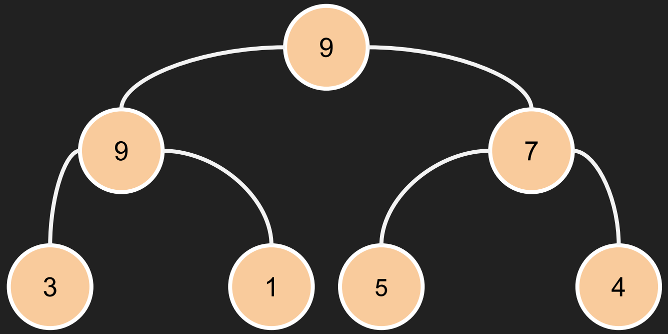
Note that there is no horizontal relationship between nodes in the heap.

Min Heap vs. Max Heap

- Before we go on it's important to note that ***priority*** is determined by the specific problem statement.
  - What does it mean to organize a heap of ***non-numerical*** *data* in priority order?
  - That's up to the programmer to decide.
- However, a heap can be organized in one of two different ways:
  - A ***min-heap*** sorts values from ***smallest*** to ***largest***. The root node will always have the ***smallest*** value in the heap.
  - A ***max-heap*** sorts values from ***largest*** to ***smallest***. The root node will always have the ***largest*** value in the heap.

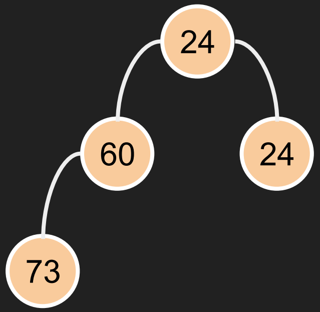


An example of a ***min-heap***.



An example of a ***max-heap***.

Activity: A Heap Interface

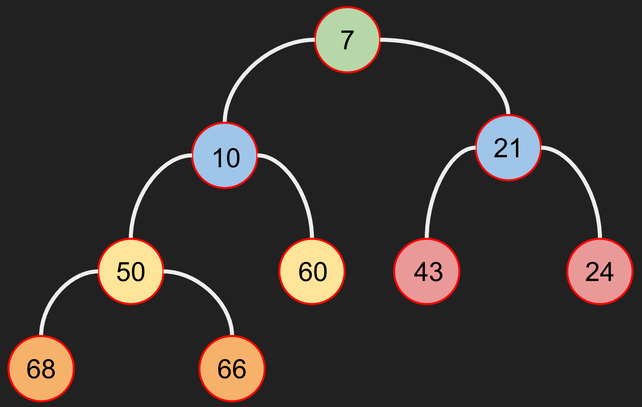


Write an interface to represent a Heap data structure. It should define the following methods:

- A parameterless constructor.
- `public void add(int value)` - adds a new value to the heap.
- `public int remove()` - removes and returns the highest priority value in the heap. For now, just return `0`.
- `public int size()` - returns the number of values in the heap. For now, just return `0`.

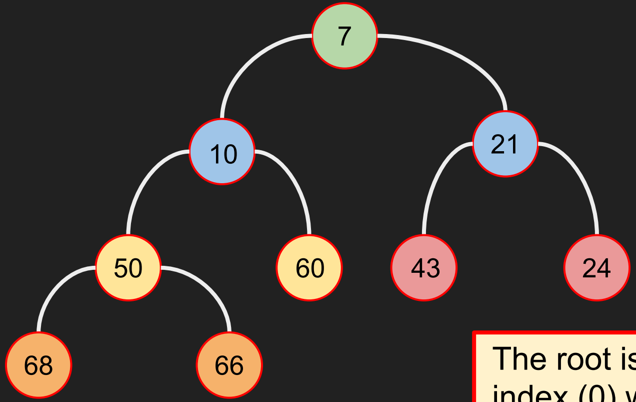
Heap Implementations

- How might a heap be implemented using an ***array***?
- The heap can be created using an ***overlay*** on an array by adhering to the following rules assuming an array of size ***N*** with indexes ranging from ***0 to N***.
    - The root of the heap is at ***index 0***.
    - Let ***k*** be the index of a node in the tree where ***0 ≤ k < N***.
    - The ***left*** child of node *k* is located at index ***2k+1***.
    - The ***right*** child of node *k* is located at index ***2k+2***.



Let’s take a look at how this heap would be overlayed onto an array.

An Array-Based Heap



7	10	21	50	60	43	24	68	66

The root is in the first index (0) within the array. Let this be k.

The children of the root are in index *2k+1* (1) and *2k+2* (2).

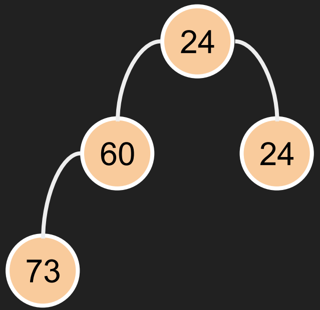
Let *k* now be the index of the left child: 1.

The children of the left child are ar *2k+1* (3) and *2k+2* (4).

Now let k be the index of *this* left child: 3.

The children of the left child are ar *2k+1* (7) and *2k+2* (8).

Activity: An `ArrayHeap` Class



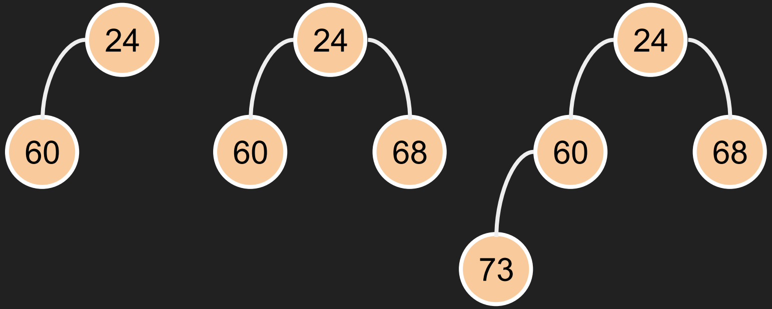
You will notice that there is a class in your project named ArrayHeap. This class already contains some helper methods for implementing your array-based heap.

- Implement the Heap interface and stub out the methods.
- Add an `int[] array` field.
- Add an `int size` field.
- In the constructor:
  - Initialize the `array` to a length of `8`.
  - Initialize the `size` to `0`.
- Update your `size()` method to return the `size` field.

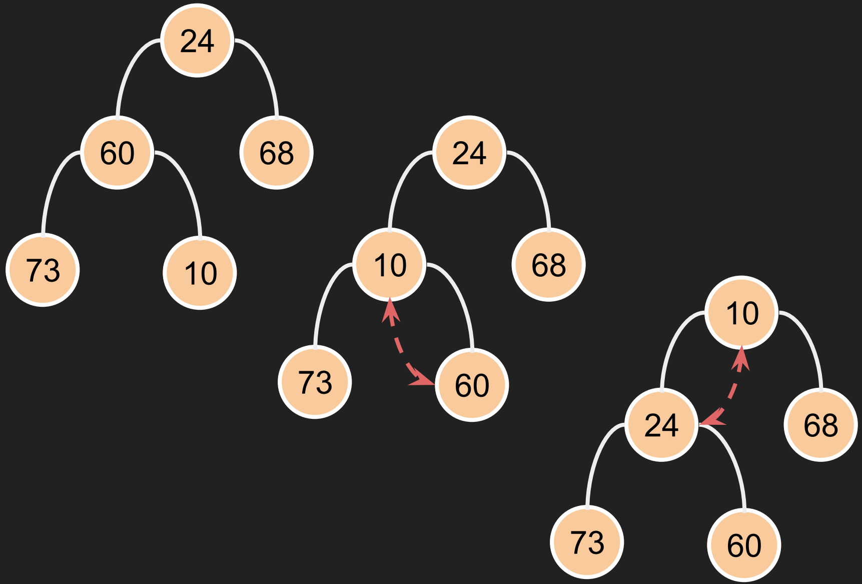
Adding to a Heap

The first node added to the heap becomes the ***root***.

Each subsequent node is added to the ***leftmost*** open position in the ***bottom*** level of the tree.



If a newly added node has a higher priority than it's parent, the values are ***swapped***.

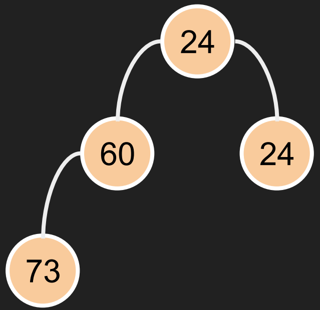


The swaps continue until the value is in the correct priority position. This is called ***sifting up***.

Activity: Adding to the Heap

Let's begin implementing the `add(int value)` method. At first it will work just like an array-based list. Remember that the new value is always added to the ***leftmost open position*** in the tree. It just so happens that this is always the `size` of the heap!

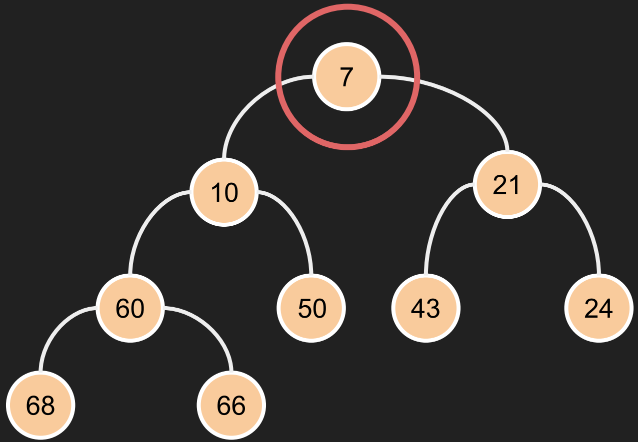
- If `array.length == size`, use `Arrays.copyOf()` to make a copy that is `size * 2`.
- Add the new value at `array[size]`.
- Increment `size`.
- Call the `static siftUp()` helper method that has been provided for you.



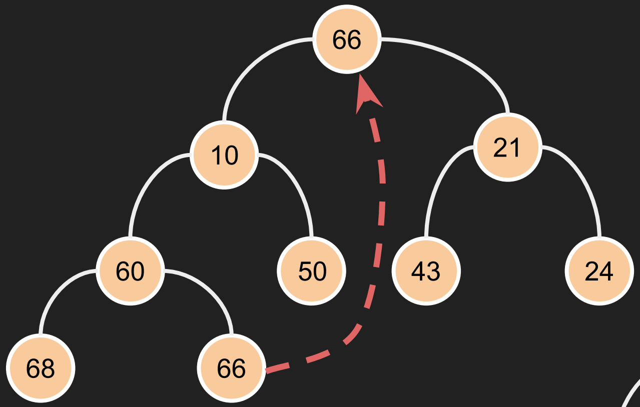
Let's take a look at how the `siftUp()` method works...

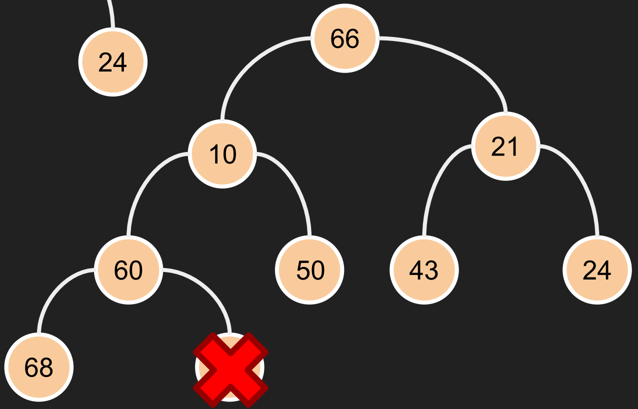
Removing From the Heap

The root always contains the ***highest priority*** value, and so the root is always the value ***removed*** from the heap.



Then, ***copy*** the value from the ***rightmost*** node in the ***bottom*** level to the root...

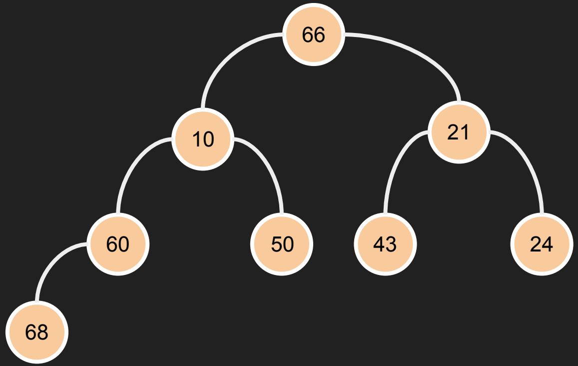




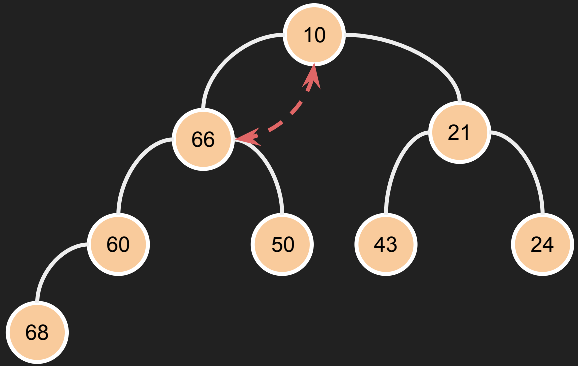
...and then ***remove*** that rightmost node from the bottom level.

Removing From a Heap: Part 2

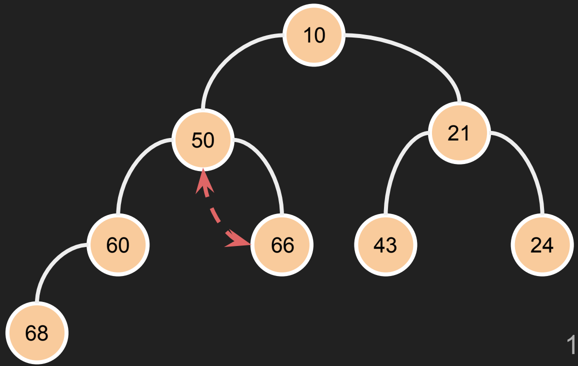
Now the value in the root needs to be ***sifted down*** to its proper position in the heap.



Compare its value to both children and, if it is not ***higher priority*** than both, ***swap*** it's value with the ***higher priority*** of the two.



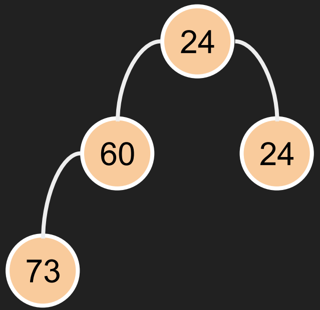
Repeat this until the value is lower priority than both of its children.



Activity: Removing from the Heap

Let's implement the `remove()` method. If our `add()` function is working properly, then the highest priority value is always at index `0` in the array.

- Save the highest priority value in a variable (the root).
- Decrement `size`.
- Copy the rightmost value at the bottom of the tree into index `0`.
  - Where should this value be?
  - After you've copied it, erase the old value (overwrite with a value of `0`).
- Use the provided `static siftDown()` method to sift the value down through the heap.
- Return the value that you saved.



Let's take a look at how the `siftDown()` method works...

Heap Complexity

Q: What is the worst case time complexity for ***adding*** a new value to a heap?

A: We are using the `size` of the heap to find the correct location, which means that adding the value to the array is **O(C)**.

In the worst case a value added to the bottom of the tree is ***sifted up*** all the way to the root of the tree. What is the complexity of that operation?

The maximum number of swaps is determined by the height of the tree, which is $\log_2 N$. Therefore the worst case time complexity is also **O($\log_2$N)**.

Q: What is the worst case time complexity for ***removing*** a value from a heap?

A: We use the `size` of the heap to find the rightmost bottom value in the tree, which is an **O(C)** operation.

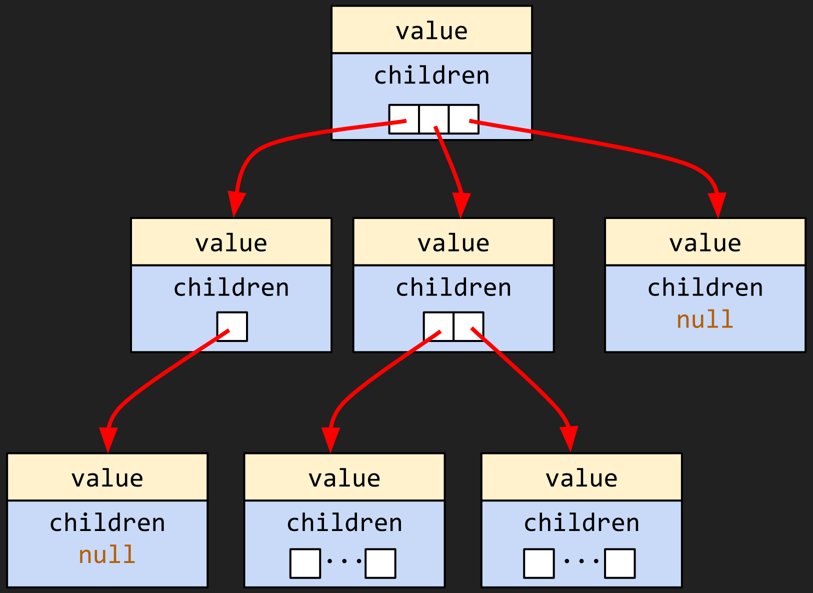
In the worst case the value copied into the root must be ***sifted down*** all the way back to the bottom of the tree, which is again the height of the tree: $\log_2 N$.

The worst case complexity of removing is therefore **O(log2N)**.

QUESTIONS?!

N-Ary Trees

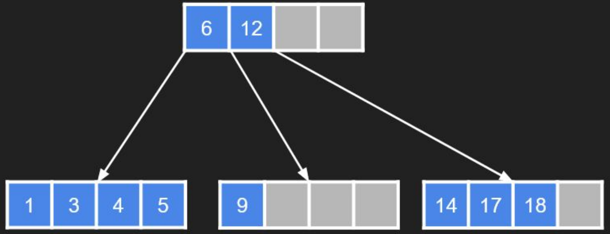
- Each Node in a Binary Tree has exactly ***two*** subtrees (***left*** & ***right***).
  - Either or both tree may be empty.
- An N-ary Tree may have ***any number*** of subtrees.
  - This means that an N-ary Tree must use ***another*** data structure (e.g. a ***list***) to store its subtrees.
- An N-ary Tree may be one of following:
  - The empty tree (no nodes).
  - At least one Node with:
    - A ***value*** of some generic type.
    - ***Zero or more*** children, each of which is the root of a subtree.
- As with binary trees, nodes that do not have children are ***leaves***.



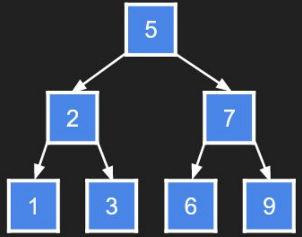
There are many different kinds of N-ary trees, but today, we are going to look at a special kind of N-ary tree: a ***B-Tree***.

B-Trees

- A ***B-Tree*** is very similar to a ***binary search tree***. The major differences are:
  - Each node may have ***more than one value***, which we call ***keys***.
  - Each node may have ***more than two children***.
- The ***order*** of a tree determines the ***maximum number of children*** that a node can have.
  - In a ***binary tree*** the order is 2.
  - In a B-Tree, the degree can be ***greater than 2***.
  - We use ***m*** to represent the order of a tree.
- Each node in a B-Tree may have ***up to m-1 keys***.
  - We us ***k*** to represent the number of keys in a node.
  - The keys are ***sorted***.
  - Each node may have up to ***k+1*** children.
  - Like a binary search tree, keys are added to children ***from left to right*** based on their value relative to the parent.



B-Trees will be the most complex data structure that we have talked about so far this semester.



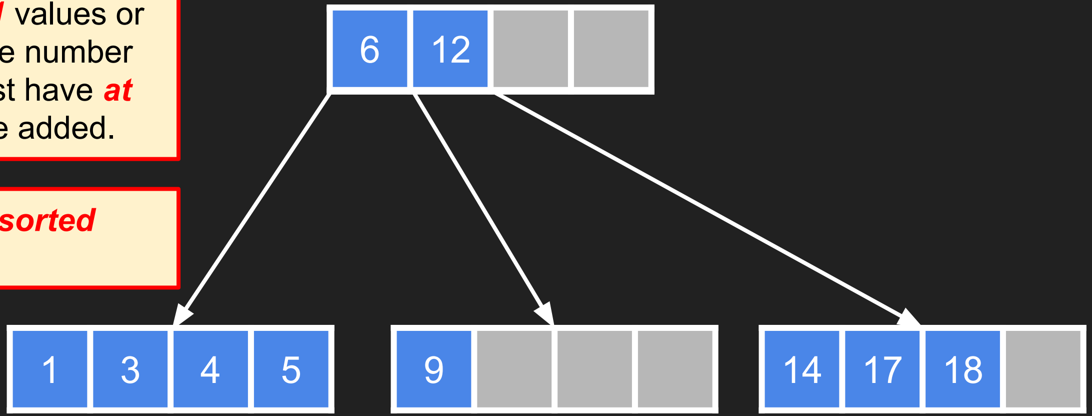
Fun Fact: a binary search tree ***is*** a B-Tree with ***m=2***, and a ***k=1***.

B-Tree Diagram

The ***order*** or ***degree*** determines the ***maximum*** number of children that each node will have. We use ***m*** to represent the order. This B-Tree is ***m=5***.

Each node may have up to ***m-1*** values or ***keys***. We use ***k*** to represent the number of keys in a node. The root must have ***at least 2*** keys before children are added.

The keys in a node are kept in ***sorted order***.



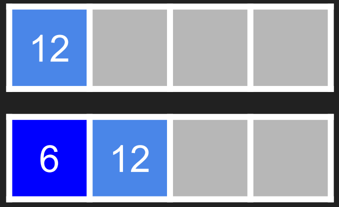
Each node will have up to ***k+1*** children. The ***leftmost child*** holds keys ***before*** the first key in its parent.

Any other children hold values that are ***between*** the keys in its parent.

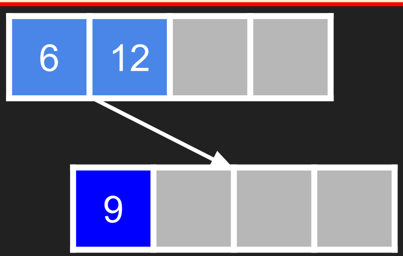
The ***rightmost child*** holds keys that are ***after*** the last key in its parent.

Inserting Into a B-Tree

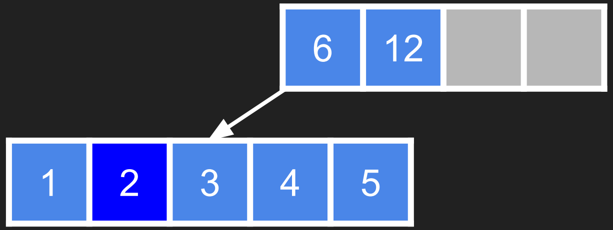
Like any tree, insertion begins at the ***root***. If there are ***fewer than 2 keys***, the new key is added to the root: `bt.insert(6)`



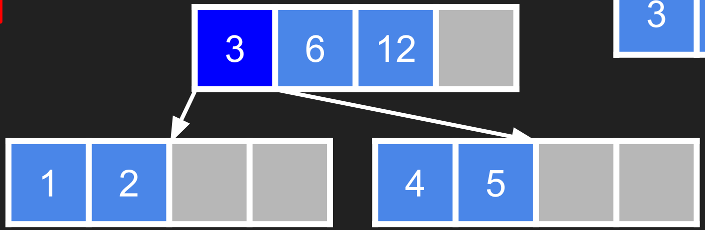
Otherwise, we iterate over the keys in the node, to find the appropriate child node into which the key should be inserted. If it does not exist, it is created: `bt.insert(9)`



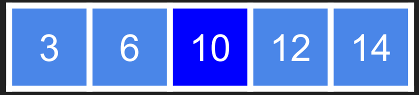
If the insert causes the number of children in the node to exceed its maximum (m-1), the node must be ***split*** in two: `bt.insert(2)`



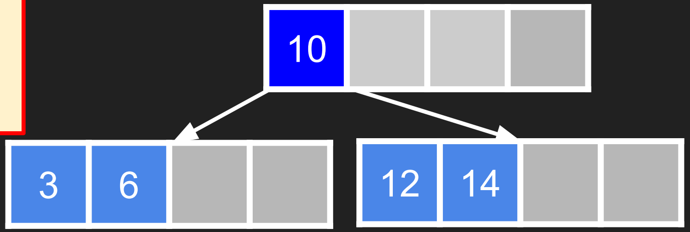
Half the children are placed in a new node, and the median value is ***promoted*** to the parent node.



If promotion causes the parent node to exceed ***its*** maximum number of children, ***it*** will be split.



If the root is split, this will cause the creation of a ***new root***.



The expected complexity of the insert operation $O(\log_2 N)$.

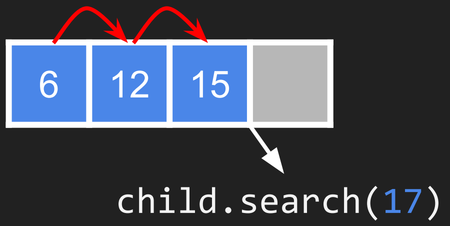
Searching a B-Tree

Searching a B-Tree is a lot like searching a binary search tree, only each node has ***more than one*** value (***key***).

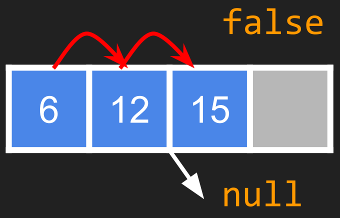
Like any tree search, it begins at the root. Iterate over the keys. If the target matches a key, return `true`: `bt.search(12)`.

If the `target` is ***less than*** one of the keys, ***recursively search*** the corresponding child node: `bt.search(14)`.

If the `target` is ***greater than*** all of the keys, recursively search the ***rightmost child***: `bt.search(17)`



If no such child exists (because no keys have been inserted), return `false`: `bt.search(14)`



Because each node has up to ***m*** children, ***m-1*** of the children are eliminated at each level of the tree.

For example, if ***m=5***, then ***4/5ths*** of the tree is eliminated with each iteration of the recursive search.

However, an ***O(M)*** search must be performed over the keys of each node along the search path.

This, the expected complexity of searching a B-Tree is $O(\log_2 N)$ (where N is the total number of keys in the tree).