Basic Datastructures

Definition 2.1. A *Dictionary* is a set of $(key, value)$ pairs where *key* is a comparable value. This is used to sort the dictionary for fast access.

Operations on dictionaries are as follows:

- insert (key, value)
- find (key)
- delete (key)

Implementation. There are many possible implementation strategies for a dictionary, each with different times for these basic operations.

Implementation Style	Insert	Find	Delete
Unsorted LinkedList	$O(1)$	$O(n)$	$O(n)$
Unsorted Array	$O(1)$	$O(n)$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted Array	$O(n)$	$O(\log n)$	$O(n)$
Binary Tree	$O(n)$	$Best : O(log(n))\|Worst : O(n)$	$O(n)$
AVL Tree	$\theta(log(n))$	$\theta(log(n))$	$\theta(log(n))$
B-Tree	?	?	?
Hashtable	$O(1)$	$O(1)$	$O(1)$

A *Lazy Deletion* is an implementation where a pair is marked as deleted, but is not actually deallocated and removed from the dictionary. This allows for removed *node*s to be gotten rid of quickly and in batches.

However, this implementation requires extra space for marking the *node* as deleted and may complicate the operations thus increasing time.

Definition 2.2. A *Tree* is a datastructure with nodes where a node has children, or nodes that is is connected to. A *Binary Tree* is a tree where each node has at most two children (left, right).

A *Root Node* is the first *node* in the tree.

A *Leaf* is a *node* with no children

A *Child* is a *node* which follows directly after any given *node*.

A *Parent* is the *node* which this *node* is linked from.

A *Sibling* is any *node* whose parent is the same

Descendents or a *Subtree* is the set of all children of a *node* and thier children.

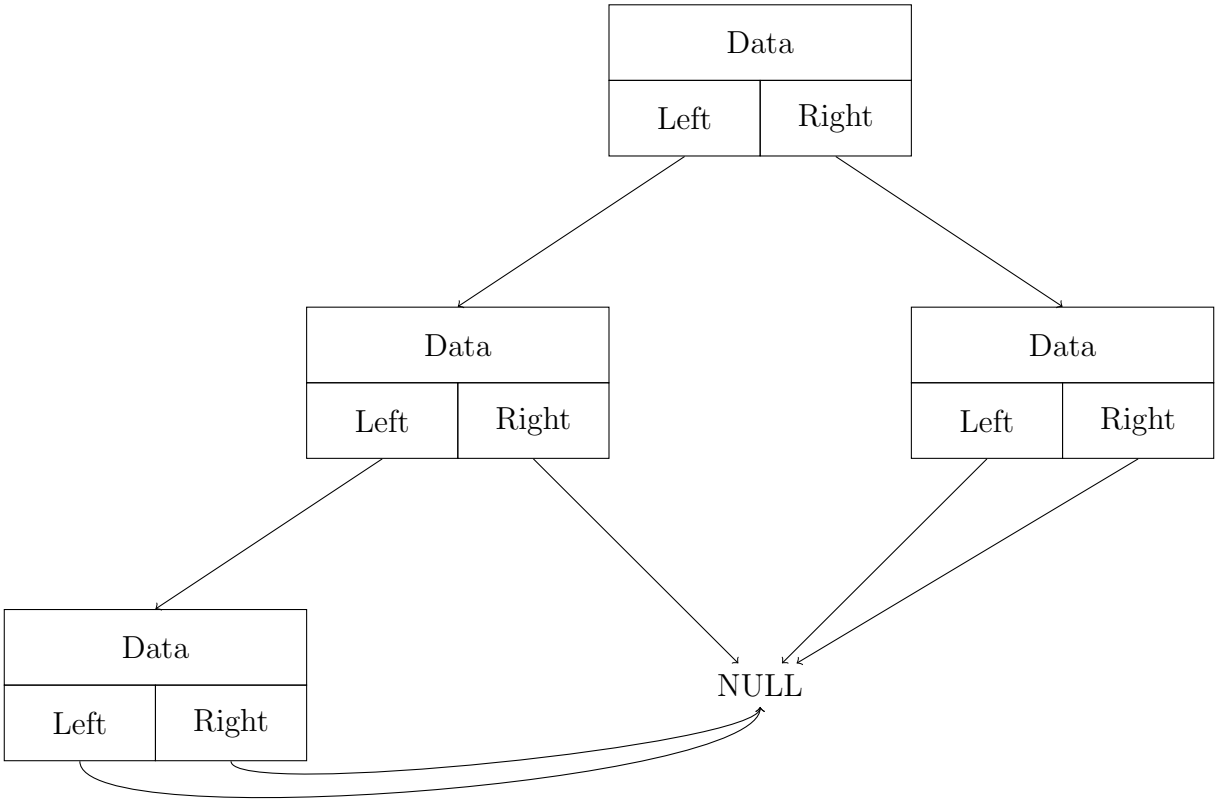
The *Depth* of a *node* is the distance of that *node* from the root.

The *Height* of a tree is the distance from teh root to the furthest leaf.

The *Branching Factor* of a *node* the number of children it has. In a binary tree this is a constant 2.

A *Balanced Tree* has a height of $O(\log n)$ where n is the number of *node*s in the tree. This is because each *node* has the same number of children except those on the largest depth.

Implementation. of a *Binary Tree*



The maximum number of nodes for a given depth, d is 2^d.

Hence, the maximum number of leaves for a tree is 2^h where h is the height of the tree.

And so it follows that the total maximum number of leaves is $2^{h+1} - 1$.

Min number of *leaves* is given by 1.

The minimum number of *nodes* is given by $h + 1$.

This is because a tree where every *node* has only one connection will have *one* leaf and $h + 1$ nodes.

Binary Trees are ordered...

- All keys in the left subtree are smaller than that of the node containing them.
- All keys in the right subtree are larger than that of the node containing them.

Hence, $find$ has a best case of $log(n)$ when the tree is balanced. However, in the worst case with a fully unbalanced tree, it takes n time.

Building a tree of n items has n^2 time because each item could be at the end of alist of n items.

Hence, we try to keep the Binary Tree balanced even after build time (after inserts and deletes). To do this we try to keep a certain condition balanced. Hence, we come up with a set of conditions in search of the best balancing.

1. Left and right subtrees of the root have the same number of nodes

   Ineffective because it is too weak. The right side could be unbalanced themselves

2. Left and right of the root have equal height

   Ineffective because too linear chains could have equal heights

3. Left and right of subtrees of every node have equal number of descendents

   This will maintain balance, but will be too strong because this ends up being very expensive.

4. Left and right subtrees of every node have equal height

   Ends up being too strong for similar reasons to the previous condition

5. Left and right subtrees of every node height differ by at most 1.

   This ends up being the sweetspot and is called the AVL condition after the people who came up with it.

   This has Order, height of log(n) + 1, *insert*, *find* and *delete* are $\theta(log(n))$ because it is balanced.

Implementation. for finding the height of a tree.

```
int treeHeight(Node* root){
    if (root == NULL)
        return -1;
    return 1 + max(treeHeight(root -> left), treeheight(root -> right));
}
```

Implementation. there are three different *Orders* for traversing a tree.

	In-order	Pre-order	Post-order
Processes the element	between each child	before proccessing children	after processing children
Example	2 * 4 + 5	+ * 2 4 5	2 4 * 5 +

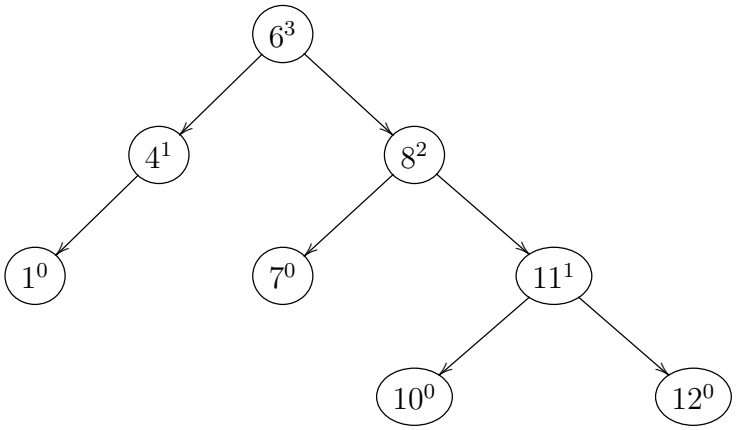
A function for traversing can be written as...

```
void traverse(Node* t){
    if (t != NULL) {
        preOrderProcess(t -> element);
        traverse(t -> left);
        inOrderProcess(t -> element);
        traverse(t -> right);
        postOrderProcess(t -> element);
    }
}
```

Implementation. of a *AVL Tree*.

This is a binary tree which is easily balanceable, it enforces the AVL condition as described before:

AVL Condition: Left and right subtrees of every node have heights which differ by at most one.

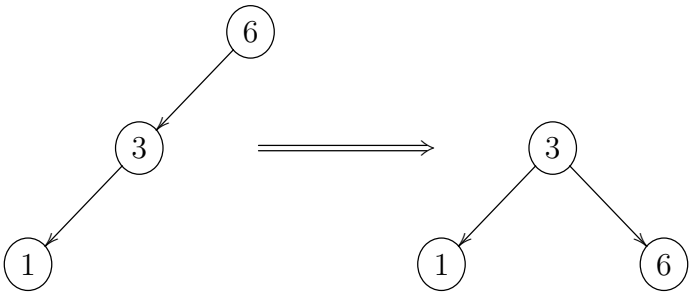


(Note, the superscript in each node above represents the height of that node. This is stored in the data structure.)

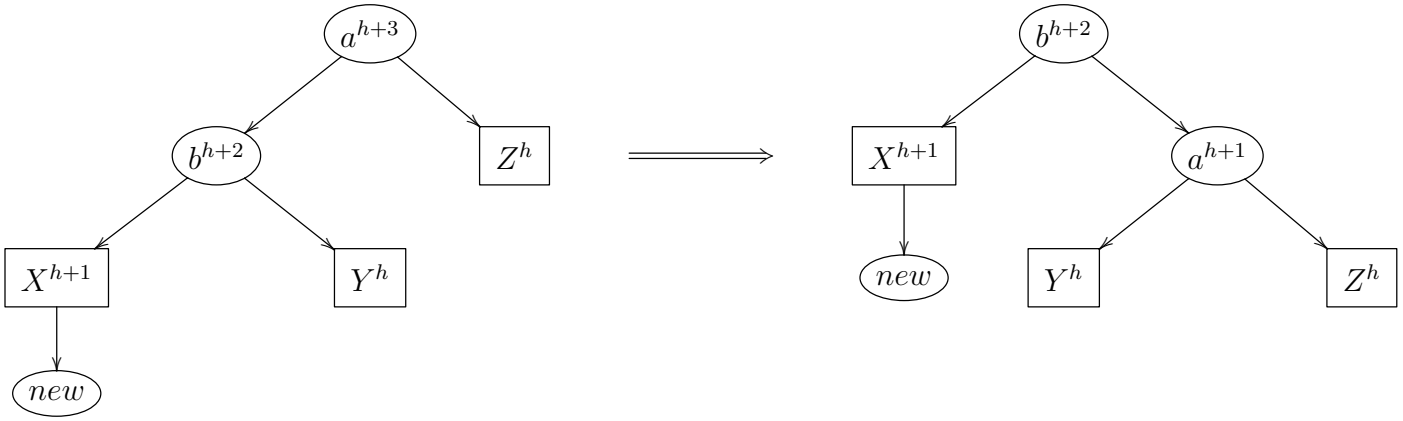
When an incorrect node is found, we create a rotation.

Implementation. Rotation.

Left-Left case



or more generically (where superscripts represent the height of the node)..



So the series of transformations for this case is, b becomes the parent, a takes on the right child of b as its left child, and b takes on a as its right child.

Hence:

$a \rightarrow$left $:= b \rightarrow$right

$b \rightarrow$right $:= a$

This is called a left-rotation.

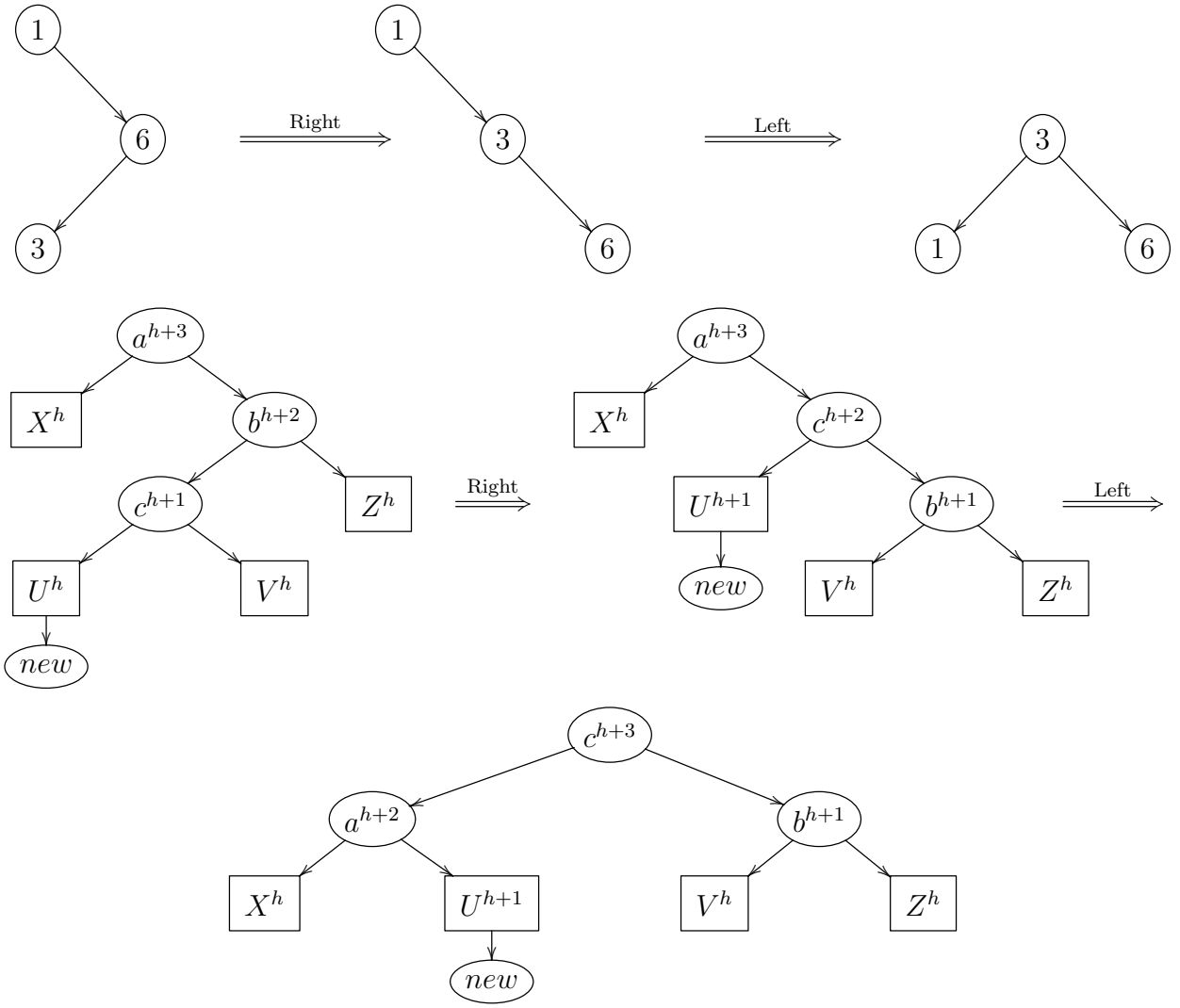
The Right-Right case is mirror to the Left-Left case. The transformation is,

$a \rightarrow$right $:= b \rightarrow$left

$b \rightarrow$left $:= a$

and is called a right rotation.

Right-Left case

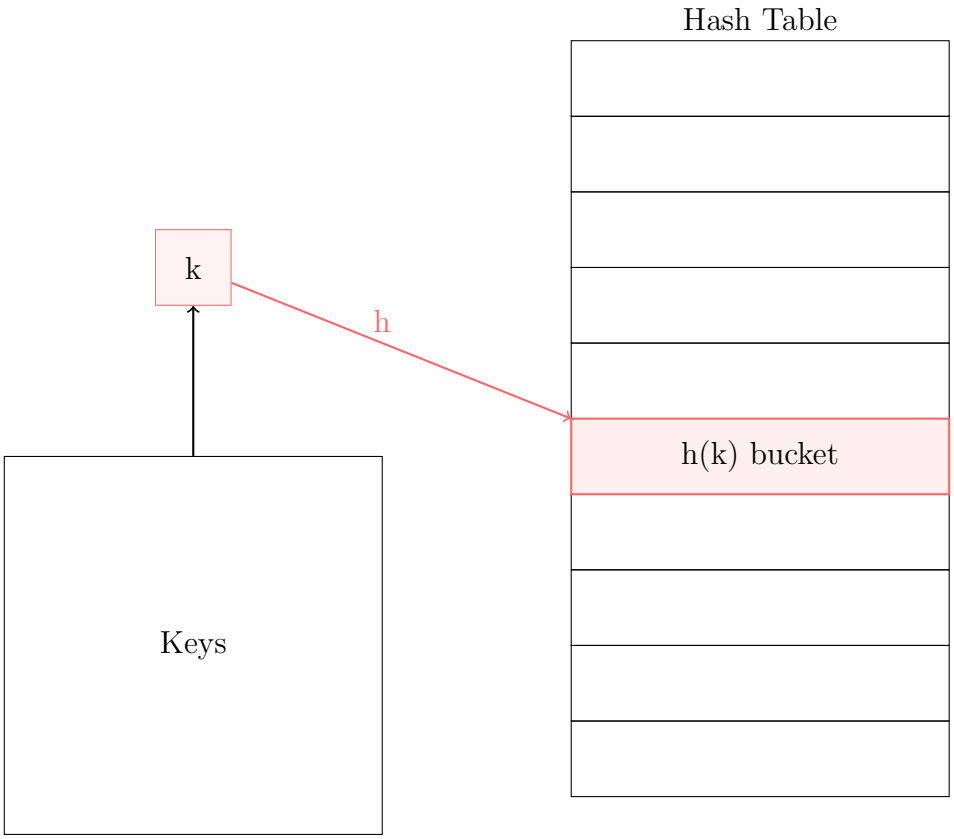


The general idea is to move c to grandparent position, then move everything else where it needs to go is the basic idea. More rigorously, do a right rotation on the b tree, then a left rotation on the a tree.

The left-right case is mirror and one would do the left rotation of the b tree and right rotation on the a tree.

Definition 2.3. A *Hashtable* is a datastructure which is useful for dictionaries. It is magical and has $O(1)$ on all basic operations.

The ways it works is we take a *key* k, and a *hash* function h, and an array t. $t[h(k)]$ is the value associated with the given key. Often keys are integers or strings.



Hashtables are specialized in the way that they word well for the basic operations, but it is unsorted so things like finding maxima and minima, predecessors and successors take $O(n)$ time. And furthermore, it takes $n \log(n)$ time to sort elements.

This is very useful where the keyspace is large, but the number of keys used is small. This limits the number of *collisions*. A *collision* exists when two keys which are used map to the same spot. Many implementations use linked lists to handle these and keep specific keys. This limits the amout of looping for finding and inserting.

The most simple hashing function ends up being $h(k) = k\%tablesize$. We want a table size which is a prime number so that keys which are multiples of each other do not overlap.

The *load factor*, λ, of a hashtable is given by

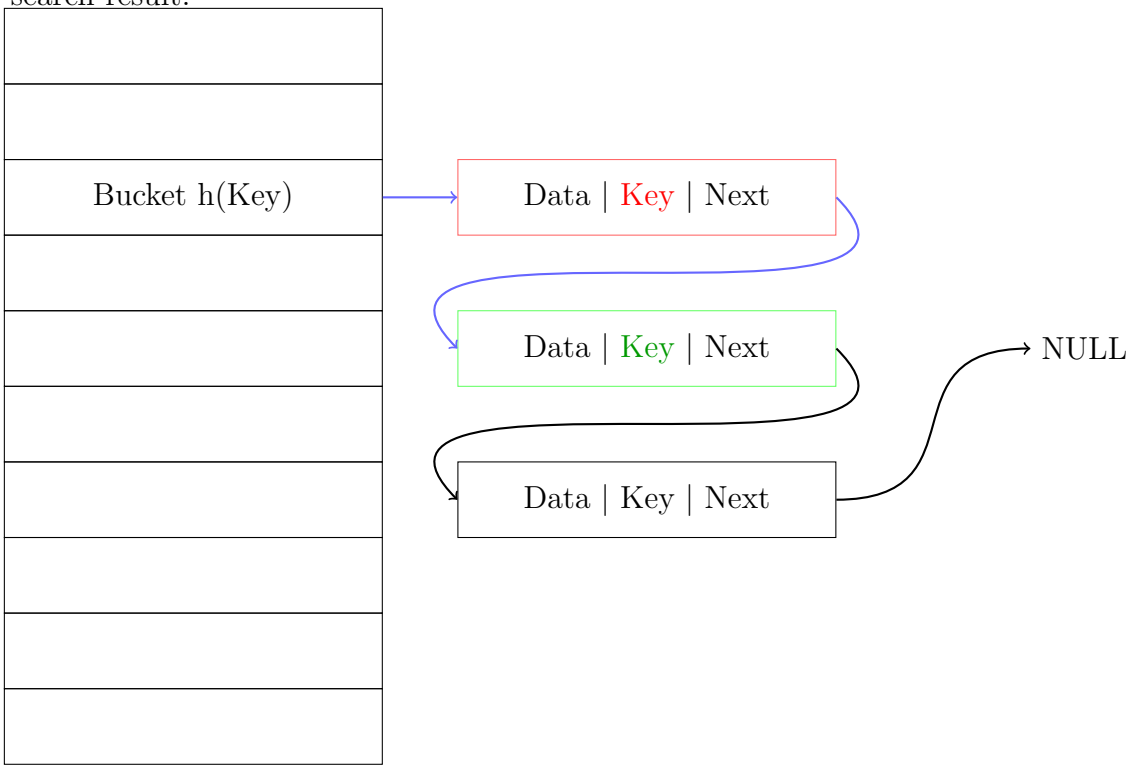
$$\lambda = \frac{N}{tablesize}$$

where N is the number of elements in the table. This load factor gives the average number of elements per bucket of the table.

So consider the find operation, if unsucessful, there are λ compares whereas if $successful$, we have $\lambda/2$ compares.

Implementation. *Chaining* is when one uses a linked list at each table index to handle collisions. Insertion of new entries with keys that link to the same index append to that linked list.

When one attempts to find the value associated with a specific key k, you do a search through the linked list at the bucket location $h(k)$. The following diagram gives an idea of this approach. The blue lines represent the links in the list searched through and the green key is the actual key that was to be found. Hence, you get the data in that green bucket as the search result.



Implementation. *Linear Probing* is where if a block in the hashtable is used when inserting a key, one skips to the next open index in the table. This approach is often slow and can cause large lag when multiple collisions occur.