Dictionary - Data structure designed to perform **search operations** very quickly. Keys are assigned to data elements.

Methods for Insertion, Removal and searching.

Dictionary entries consist of key/element pairs (k,e)

There are ordered and unordered dictionaries.

**Unordered Dictionary**

Allows multiple elements to have the same key.

Can be implemented by an unsorted sequence (vector/list), in which case it is called a “log file”

**Hash Table**

If the keys correspond to addresses, we can use a hash table.

Corresponds of **bucket arrays** and a **hash function**.

An element e, with address k, is inserted into A at A[k]

bucket arrays require unique integers, but this is not always the case.

Instead of storing at A[k] we use a hashing function to calculate h(k) and then store at A[h(k)].

Hash functions should be efficient at computing h(k) and should produce minimum collisions.

**Hash Codes**

First action performed in hashing is to assign an integer value to the key (hash code). Usually taken as either upper/lower part of key, or summation of the two (summation hash code)

**Polynomial Hash Codes**

Best choice for keys that are character strings or other multiple-length objects.

Lots of words map to the same hash code, so we use the polynomial coefficient to get a better spread for words.

**Compression Maps**

Second step in the hashing function. If the range of hash codes produced exceeds the size of the bucket array, we must compress, to restrict the integer values to values 0 to N-1.

Use h(k) = k % N - Division Method

h = |ak + b| % N = MAD (multiply add divide method) has a collision probability of <1/N

**Collision Handling**

collision occurs when two distinct keys k1 and k2 have h(k1) = h(k2).

Simplest form of resolution is to use **separate chaining**, where each bucket implements a vector/array, allowing for multiple elements to be stored in a single bucket. Worst case scenario, reduces to a linked list, very inefficient.

**Load Factors and Rehashing**

Load factor = number of elements / bucket size

Should be kept small to ensure that search/insertion/deletion operations are kept fast.

Keeping the load factor small, the bucket size must be changed.

This is called rehashing

**Open Addressing**

Another collision handling technique.

Only one element per bucket.

Cycle until empty space is found if we detect a collision.

Simple to implement, but can be very costly. Addition/Removal can take O(n) in worst case.

**Requires “dirty” bit**, to avoid cases where elements before the one you are searching for get removed. White space encountered before existing element.

Linear probing - cycle for free position

Quadratic probing - A[i + j^2 modN) for j = 0,1,2….. until free space is found. Prevents clustering

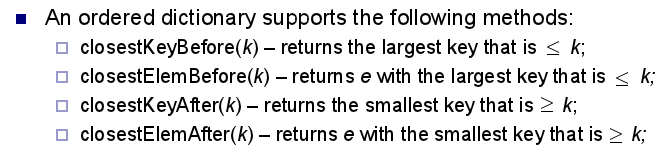
N should always be a prime number

Double hashing - Second non-zero hash function. A[(i + j\*h’(k))modN]

Eliminates the clustering found with linear/quadratic probing.

**Ordered Dictionary**

A comparator is used to establish the order relation between keys.



The nature of the above methods implies that a logfile/hash table is inappropriate for a dictionary, as neither implement an ordering among keys.

**Sorted (Lookup) Table implementation of a Dictionary**

If the Dictionary is ordered, the items can be stored in a vector. The ordering of the keys makes searching much faster.

The ordered vector implementation of a dictionary is referred to as a **Lookup Table**

Insertion takes O(n) time in the worst case, as all keys greater than k need to be shifted up a space to make room.

However, searching is a lot faster:

**Binary search**

best case - log(n)

worst case - log(n), if already ordered, produces a list.

removal of internal node - removing A, find leftmost child of right-hand child of A. Insert into A and reorder tree.

Number of elements in the working set halves with each recursive call, so we have

T(n) = n/2^m

where m is the number of recursive calls.

This can be visualized as a tree, as the working set is split each time (branches) until we reach a single item (leaf node). We know that the the “depth” of a tree is log(n), so we know

T(n) = n/2^(log(n))

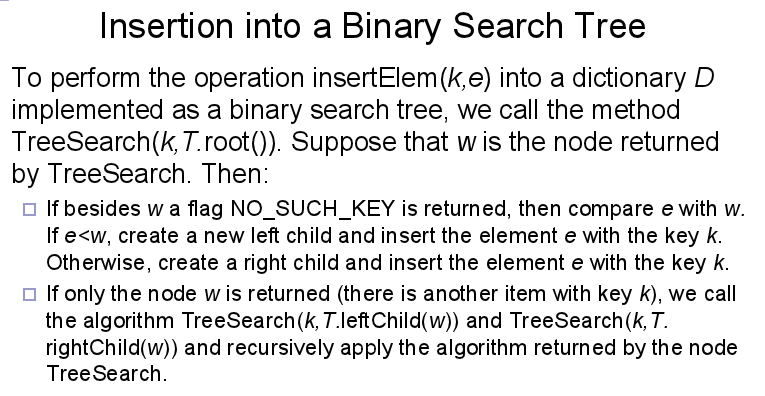
Therefor, the time complexity of a binary search scales with O(log(n))

It follows then, that we can implement the algorithm in an actual tree structure.

**Binary Search Trees**

A tree data structure adapted to a binary search algorithm, with each node of the tree holding an element.

An inorder traversal of the tree will visit each node in ascending order.



In the above, TreeSearch returns a node whether it finds the key or not. This allows for instant insertion in the case that the key does not already exist in the tree, as we can simply add a new child onto the node of where the key “might” have been (where the search terminated).

**Removal From a Binary Search Tree**

The big problem with removing an element from a binary tree is ensuring that the tree remains connected.

We must firstly perform a search for the key, k to check whether the key exists in the tree.

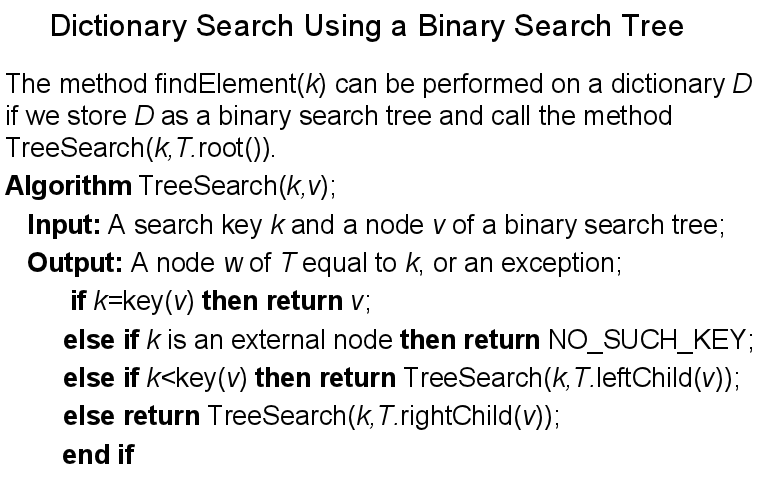
If the key exists, we handle it in two ways.

* If the key is a leaf node, removal is trivial
* If the key is internal, we must perform the following steps to maintain a complete tree.

Removal of an internal node:

1. Find the node, **y** that, in an inorder traversal, follows the node we wish to remove (this should be the **leftmost** internal node of the right subtree)
2. Store our removed node, **w**, in a temporary variable “t”
3. Overwrite w’s position in the tree with y, removing w from the tree.
4. Remove the element y from the tree
5. Return the element stored in t (the element we removed)

**Searching**



In this implementation, the search time of the tree is proportional to the height of the tree.

This can become very inefficient if the height begins to become comparable to n.

We want to keep the height as close to O(log(n)) as possible.

One way to achieve this is using an AVL tree to auto-balance after each insertion.

**AVL Trees(BALANCED BINARY TREES)**

Designed to improve efficiency for basic dictionary operations.

Normal trees have a problem with becoming extremely inefficient if the height of the tree is comparable to n (becomes a list)

If you read this for the first time now it’s too late!

Each node stores the height of the tree at that point (**Height Balance** property).

Sibling heights can differ by at most 1 in an AVL tree(they are balanced).

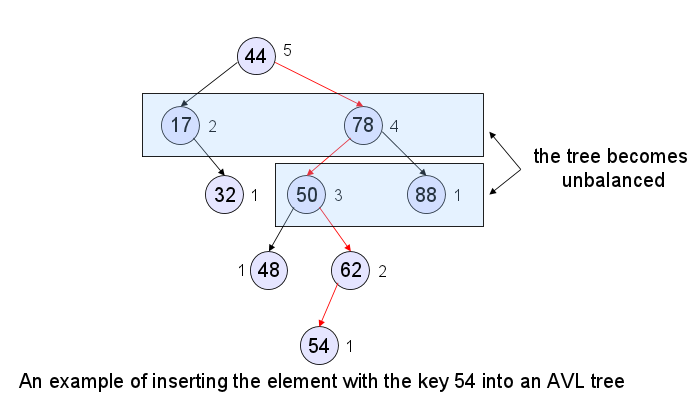
AVL Tree’s inherit the property that the height of a tree storing n items is O(log(n))

Insertion causes an imbalance so we use a search and repair strategy.

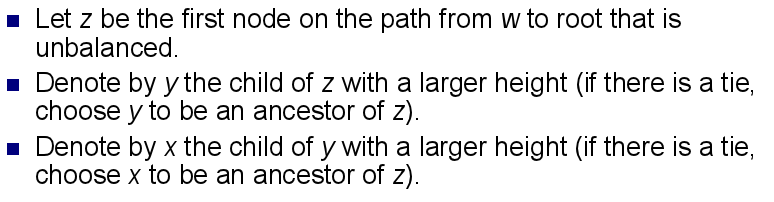
**Insertion into an AVL Tree**

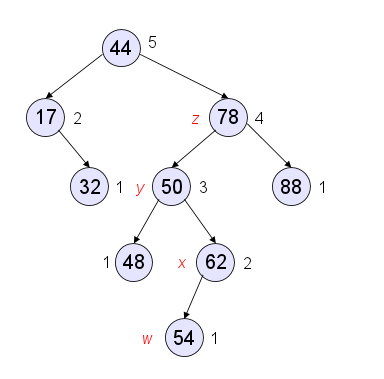
Begin by inserting the element as a leaf node, as you would with a generic tree (remembering to update height values along the path traversed during insertion)

The tree becomes imbalanced if we find siblings with a difference of heights greater than 1.

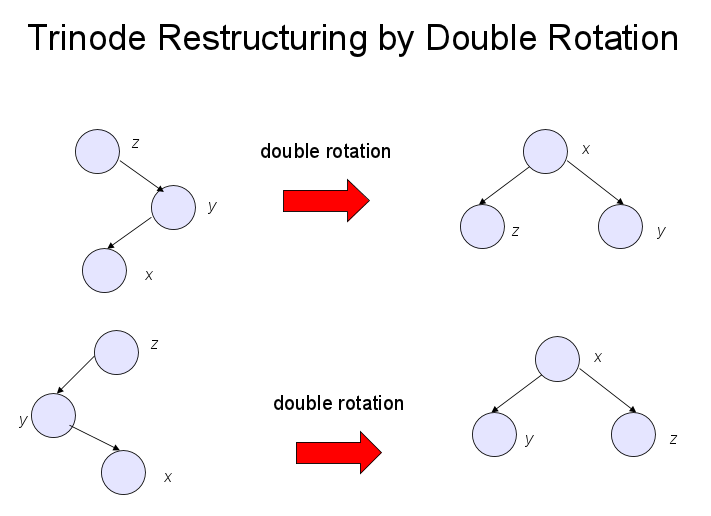
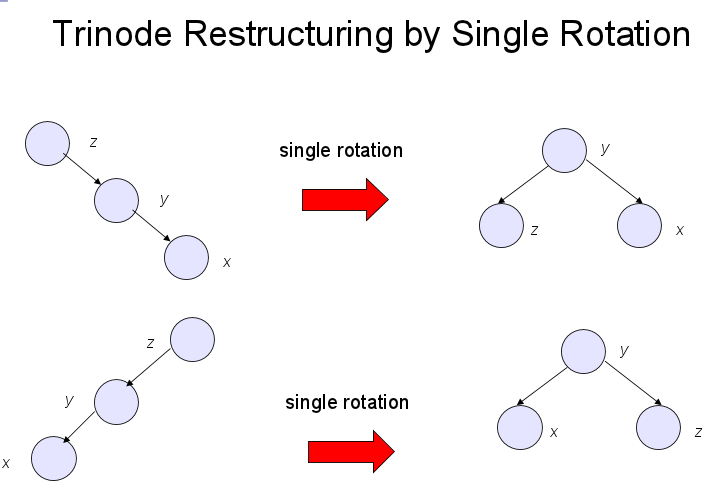


If the tree is imbalanced, we must use a search and repair strategy to rebalance the tree.





We must re-balance the subtree rooted at z, using the **Trinode Restructuring** method (rotation).



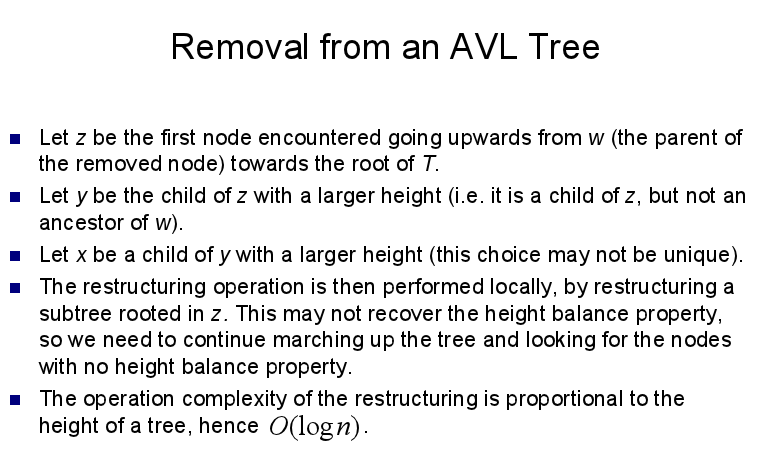
**Removal From an AVL Tree**

The first stage is the same as a binary tree, although this can violate the height-balance property of AVL trees.

Removal of external nodes does not violate this property.

However, removal of internal nodes can cause imbalance in nodes between the parent of the previously removed node and the root node.

We use the trinode restructuring method to restore balance.



(see lecture notes for example)