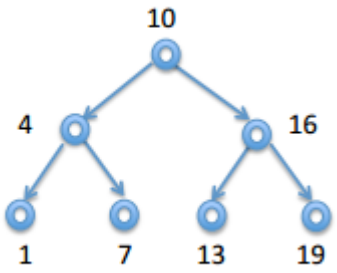
AVL TREES

*An **AVL tree** (Adelson-Velskii and Landis' tree) is a self-balancing binary search tree.

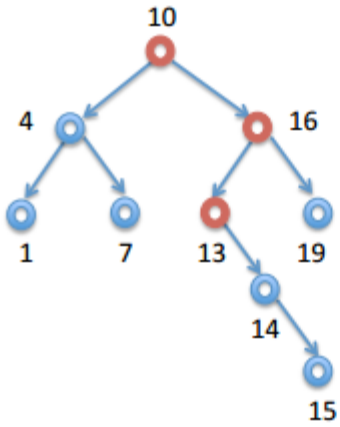
* They were the first dynamically balanced trees to be proposed

*An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.



height = 3
height inv. satisfied



height = 5
height inv. violated at 13, 16, 10

*if at any time they differ by more than one, rebalancing is done to restore this property.

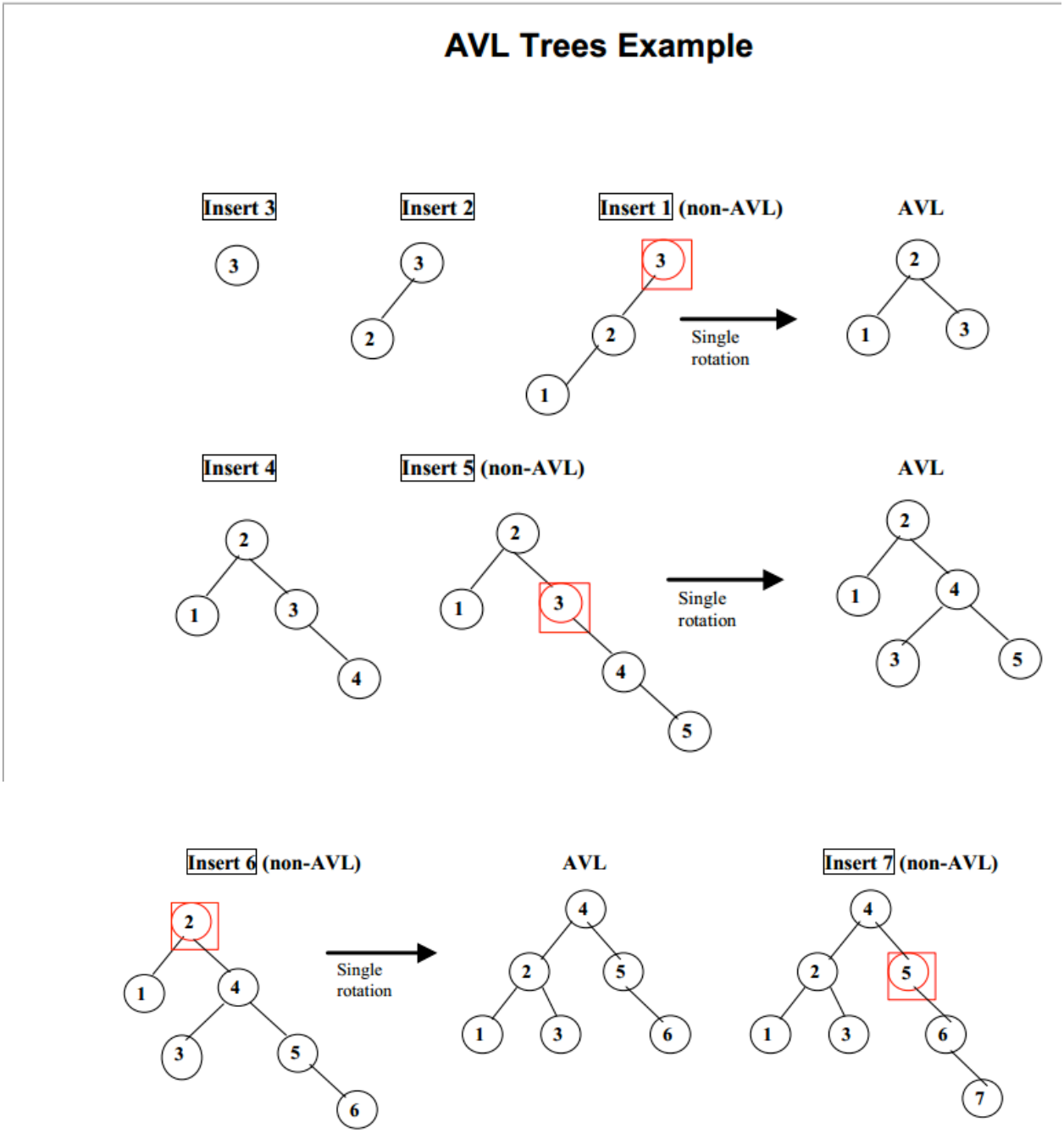
Insertion

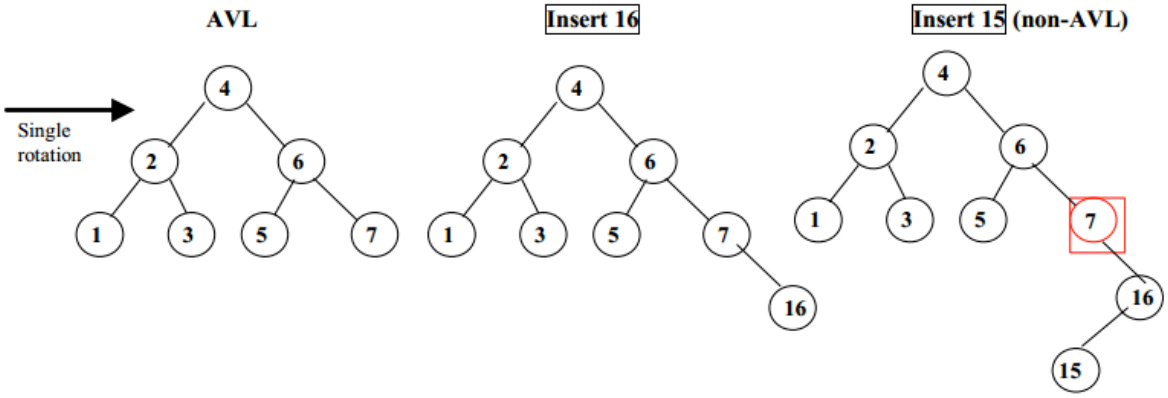
*After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL.

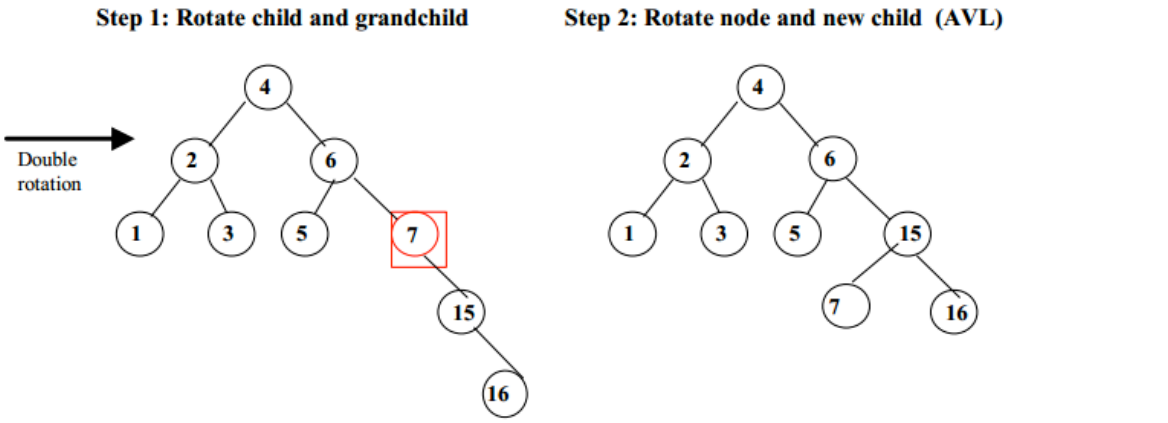
* The balance factor is calculated as follows:

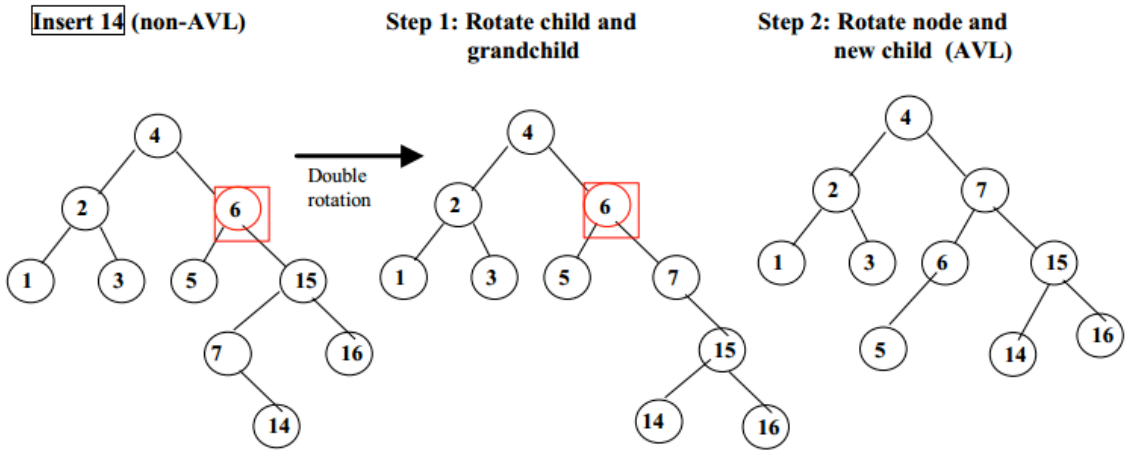
balanceFactor = height(left subtree) - height(right subtree).

*For each node checked, if the balance factor remains −1, 0, or +1 then no rotations are necessary. However, if balance factor becomes less than -1 or greater than +1, the subtree rooted at this node is unbalanced. If insertions are performed serially, after each insertion, at most one of the following cases needs to be resolved to restore the entire tree to the rules of AVL.

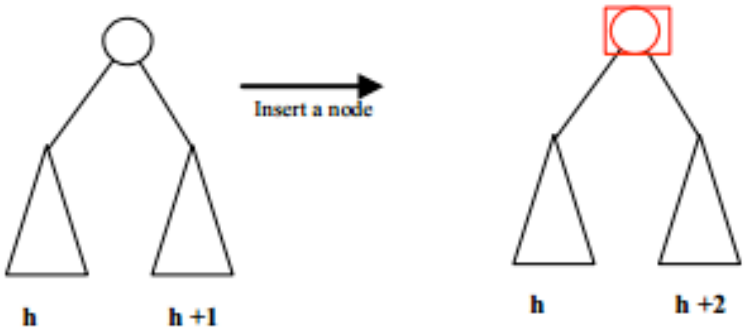



AVL
Insert 16
Insert 15 (non-AVL)
Single rotation
4
2
6
1
3
5
7
16
15

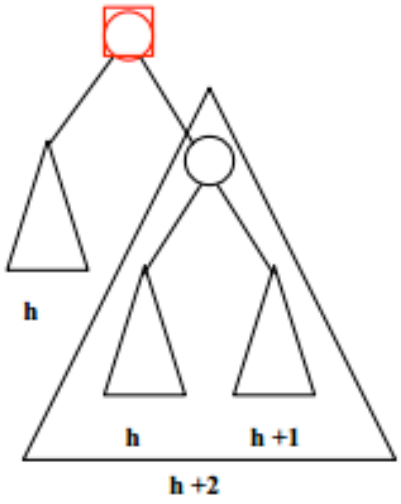

Step 1: Rotate child and grandchild
Step 2: Rotate node and new child (AVL)
Double rotation
4
2
6
1
3
5
7
15
16


Insert 14 (non-AVL)
Step 1: Rotate child and grandchild
Step 2: Rotate node and new child (AVL)
Double rotation
4
2
6
1
3
5
15
7
16
14

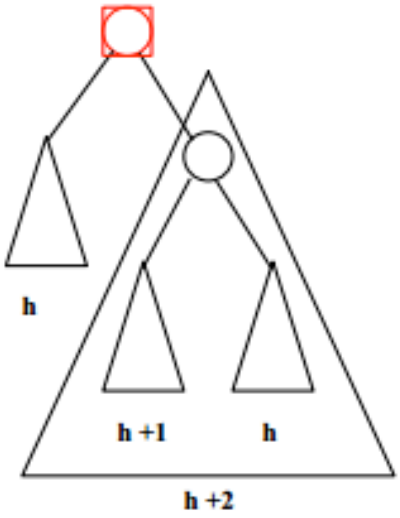
AVL Tree **No longer AVL, must rebalance**



Two ways to expand subtree of height h+2:



Apply a Single Rotation



Apply a Double Rotation

Note: the symmetrical case is handled identically (i.e. mirror image)

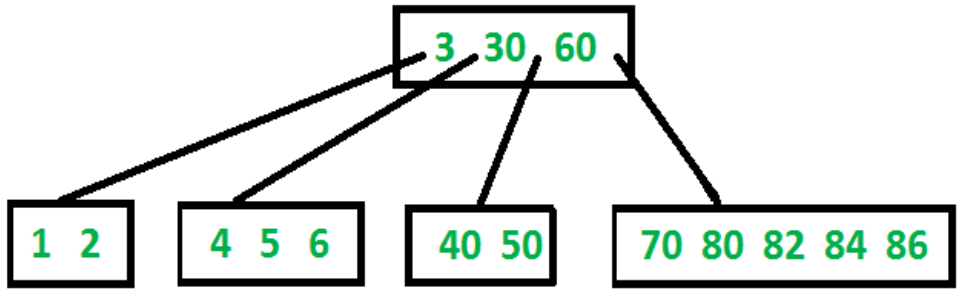
B-Trees

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVLand Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

Properties of B-Tree

1) All leaves are at same level.
2) A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
3) Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.
4) All nodes (including root) may contain at most 2t – 1 keys.
5) Number of children of a node is equal to the number of keys in it plus 1.
6) All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in range from k1 and k2.
7) B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.



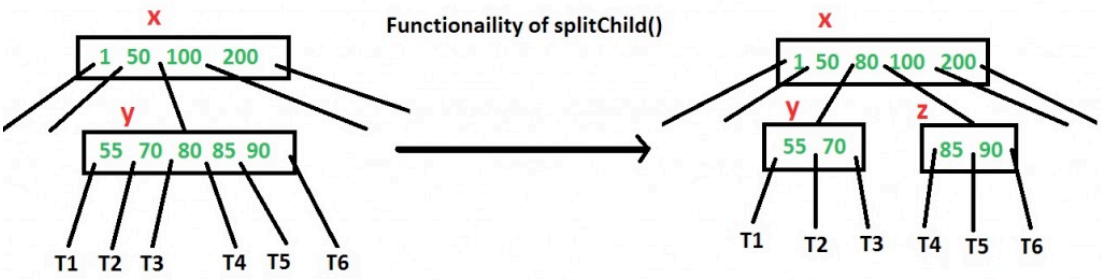
Search

Search is similar to search in Binary Search Tree. Let the key to be searched be k. We start from

root and recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traverse
Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

A new key is always inserted at leaf node. Let the key to be inserted be k. Like BST, we start from root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on number of keys that a node can contain. So before inserting a key to node, we make sure that the node has extra space. *How to make sure that a node has space available for key before the key is inserted?* We use an operation called splitChild() that is used to split a child of a node. See the following diagram to understand split. In the following diagram, child y of x is being split into two nodes y and z. Note that the splitChild operation moves a key up and this is the reason B-Trees grow up unlike BSTs which grow down.



As discussed above, to insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is complete algorithm.

Insertion
1) Initialize x as root.
2) While x is not leaf, do following
..**a)** Find the child of x that is going to to be traversed next. Let the child be y.
..**b)** If y is not full, change x to point to y.
..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as first part of y. Else second part of y. When we split y, we move a key from y to its parent x.

3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Note that the algorithm follows the Cormen book. It is actually a proactive insertion algorithm where before going down to a node, we split it if it is full. The advantage of splitting before is, we never traverse a node twice. If we don't split a node before going down to it and split it only if new key is inserted (reactive), we may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because parent was already full). This cascading effect never happens in this proactive insertion algorithm. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

Initially root is NULL. Let us first insert 10.

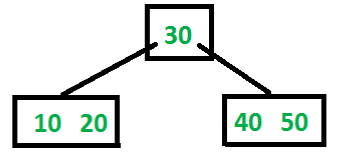
Insert 10

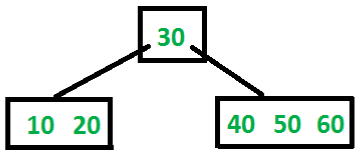
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because maximum number of keys a node can accommodate is 2*t – 1 which is 5.

Insert 20, 30, 40 and 50

Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

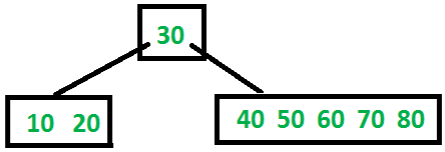
Insert 60





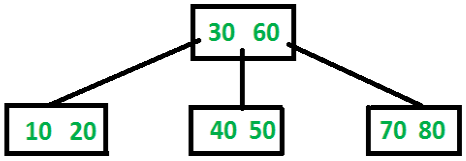
Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

Insert 90



SPLAY TREES

A Splay tree is a binary SEARCH tree which reorganises himself by every (creation, deletion or location) operation. These reorganisations can all be bad, but it's sure that every reorganisation will bring the selected node to root. These reorganisations are called "splaying". Wikipedia says "A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. For many sequences of nonrandom operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Dominic Sleator and Robert EndreTarjan in 1985."

What the man or women who wrote this on wiki means, is that every sequence of m operations (this can be: locating, deleting and adding) takes O(m lg n). Basically every single operation from the m-operations is O(lg n) in amortized time. This doesn't mean that every single operations is really O(lg n). It's not guaranteed that every single operation is O(lg n), they all can be very bad.

The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is O(n). The worst case occurs when the tree is skewed. We can get the worst case time complexity as O(Logn) with AVL and Red-Black Trees.

Can we do better than AVL or Red-Black trees in practical situations?
Like AVL and Red-Black Trees, Splay tree is also self-balancing BST. The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we

have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

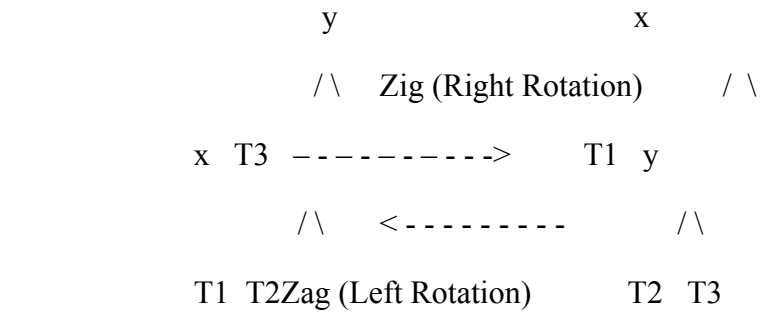
All splay tree operations run in O(log n) time on average, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case.

Search Operation
The search operation in Splay tree does the standard BST search, in addition to search, it also splays (move a node to the root). If the search is successful, then the node that is found is splayed and becomes the new root. Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.

There are following cases for the node being accessed.

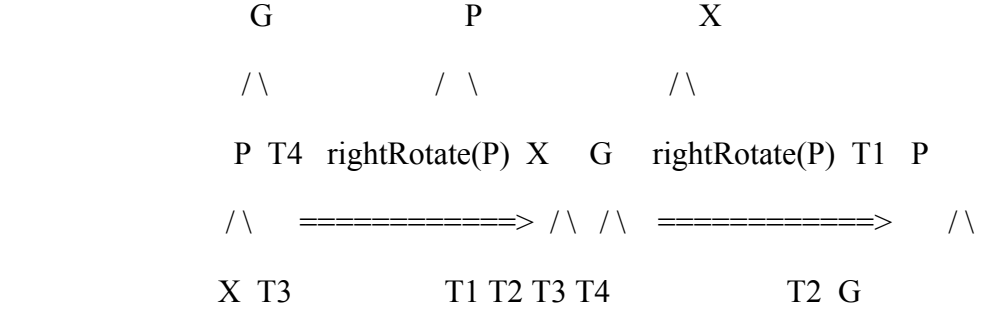
1) Node is root We simply return the root, don't do anything else as the accessed node is already root.

2) Zig: ***Node is child of root*** (the node has no grandparent). Node is either a left child of root (we do a right rotation) or node is a right child of its parent (we do a left rotation).
T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

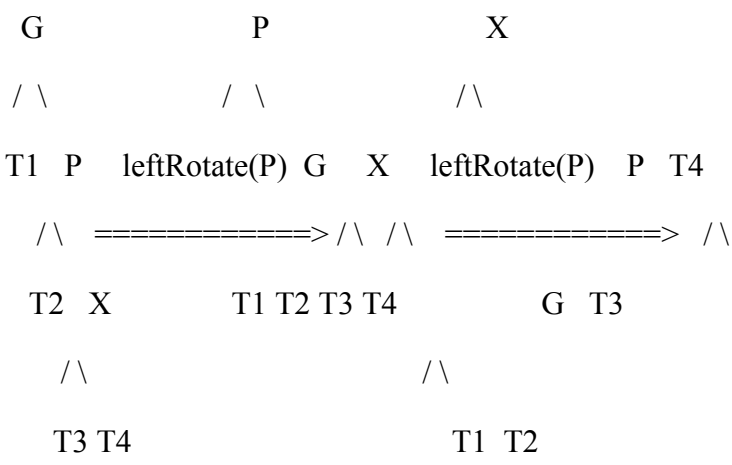


3) ***Node has both parent and grandparent***. There can be following subcases.
.....**3.a) Zig-Zig and Zag-Zag** Node is left child of parent and parent is also left child of grand parent (Two right rotations) OR node is right child of its parent and parent is also right child of grand parent (Two Left Rotations).

Zig-Zig (Left Left Case):

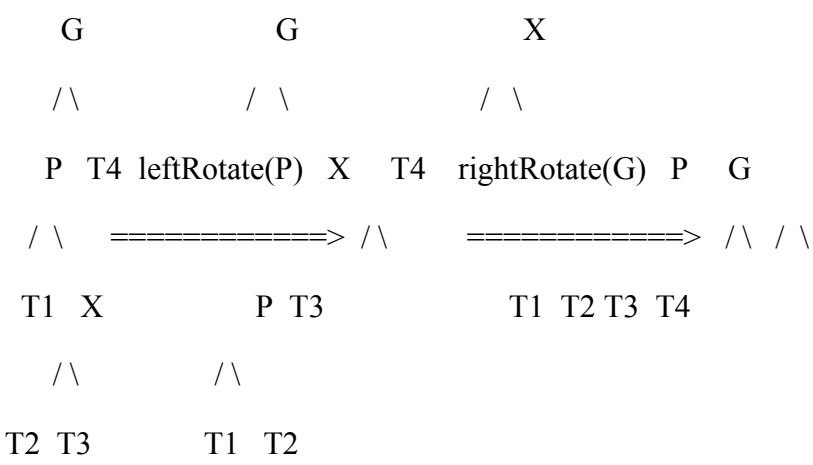


Zag-Zag (Right Right Case):

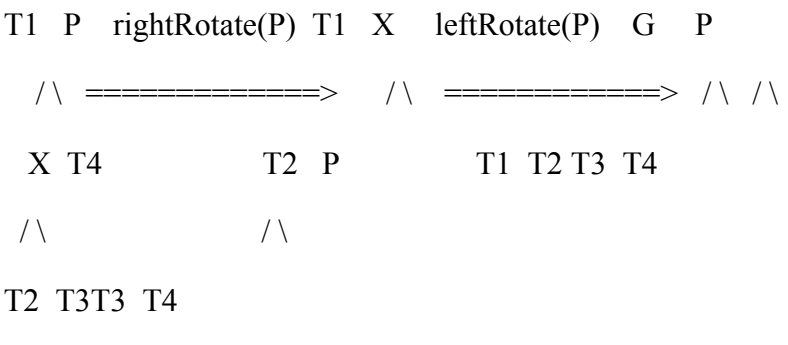


.....**3.b) Zig-Zag and Zag-Zig** Node is left child of parent and parent is right child of grand parent (Left Rotation followed by right rotation) OR node is right child of its parent and parent is left child of grand parent (Right Rotation followed by left rotation).

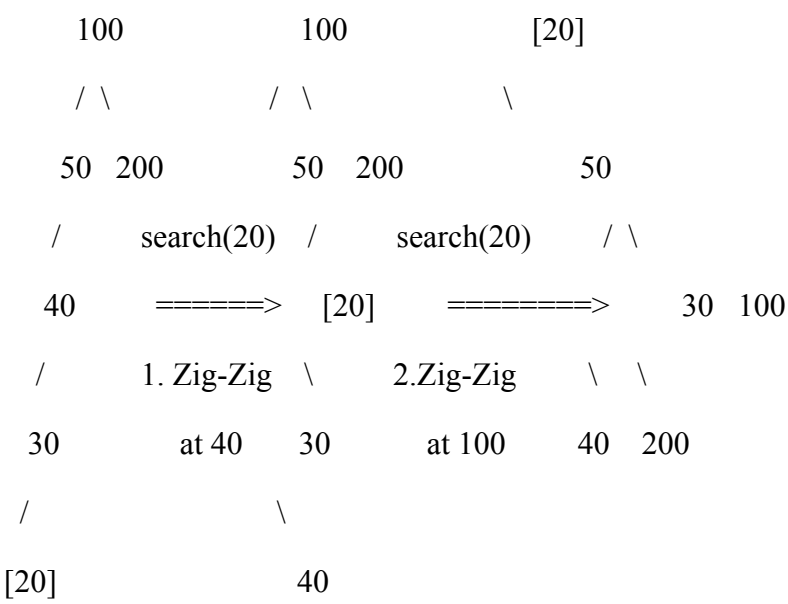
Zig-Zag (Left Right Case):



Zag-Zig (Right Left Case):



Example:



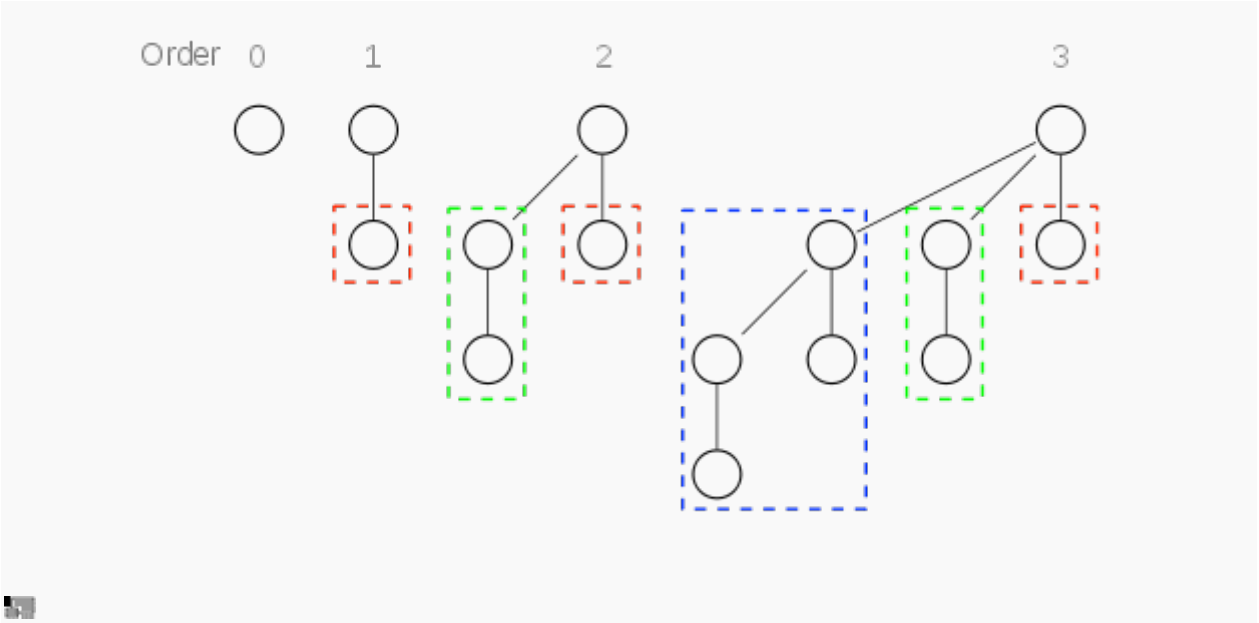
The important thing to note is, the search or splay operation not only brings the searched key to root, but also balances the BST. For example in above case, height of BST is reduced by 1.

BINOMIAL HEAPS

Binomial Heap

A binomial heap is implemented as a collection of binomial trees (compare with a binary heap, which has a shape of a single binary tree). A **binomial tree** is defined recursively:

- A binomial tree of order 0 is a single node
- A binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*−1, *k*−2, ..., 2, 1, 0 (in this order).



Binomial trees of order 0 to 3: Each tree has a root node with subtrees of all lower ordered binomial trees, which have been highlighted. For example, the order 3 binomial tree is connected to an order 2, 1, and 0 (highlighted as blue, green and red respectively) binomial tree.

A binomial tree of order *k* has 2^k nodes, height *k*.

Because of its unique structure, a binomial tree of order *k* can be constructed from two trees of order *k*−1 trivially by attaching one of them as the leftmost child of root of the other one. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps.

The name comes from the shape: a binomial tree of order n has $\binom{n}{d}$ nodes at depth d.

Structure of a binomial heap

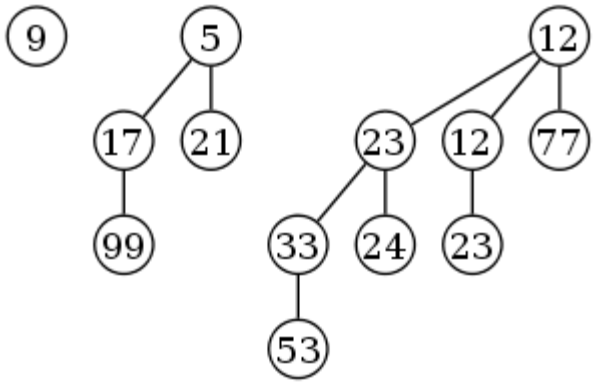
A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:

- Each binomial tree in a heap obeys the *minimum-heap property*: the key of a node is greater than or equal to the key of its parent.
- There can only be either *one* or *zero* binomial trees for each order, including zero order.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap.

The second property implies that a binomial heap with *n* nodes consists of at most log *n* + 1 binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of nodes *n*: each binomial tree corresponds to one digit in the binary representation of number *n*. For

example number 13 is 1101 in binary, $2^3 + 2^2 + 2^0$, and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0 (see figure below).



Example of a binomial heap containing 13 nodes with distinct keys.
The heap consists of three binomial trees with orders 0, 2, and 3.

FIBNOCCI HEAP

Fibonacci heap is a heap data structure consisting of a collection of trees. It has a better amortized running time than a binomial heap. Fibonacci heaps were developed by Michael L. Fredman and Robert E. Tarjan in 1984 and first published in a scientific journal in 1987. The name of Fibonacci heap comes from Fibonacci numbers which are used in the running time analysis.

Find-minimum is *O*(1) amortized time.[1] Operations insert, decrease key, and merge (union) work in constant amortized time.[2] Operations delete and delete minimum work in*O*(log *n*) amortized time.[2] This means that starting from an empty data structure, any sequence of *a* operations from the first group and *b* operations from the second group would take *O*(*a* + *b* log *n*) time. In a binomial heap such a sequence of operations would take *O*((*a* + *b*) log (*n*)) time. A Fibonacci heap is thus better than a binomial heap when *b* isasymptotically smaller than *a*.

Using Fibonacci heaps for priority queues improves the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing the shortest path between two nodes in a graph.

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations. For example merging heaps is done

simply by concatenating the two lists of trees, and operation *decrease key* sometimes cuts a node from its parent and forms a new tree.

However at some point some order needs to be introduced to the heap to achieve the desired running time. In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth Fibonacci number. This is achieved by the rule that we can cut at most one child of each non-root node. When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree (see Proof of degree bounds, below). The number of trees is decreased in the operation *delete minimum*, where trees are linked together.

As a result of a relaxed structure, some operations can take a long time while others are done very quickly. For the amortized running time analysis we use the potential method, in that we pretend that very fast operations take a little bit longer than they actually do. This additional time is then later combined and subtracted from the actual running time of slow operations. The amount of time saved for later use is measured at any given moment by a potential function. The potential of a Fibonacci heap is given by

Potential = $t + 2m$

where t is the number of trees in the Fibonacci heap, and m is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked). The amortized time for an operation is given by the sum of the actual time and c times the difference in potential, where c is a constant (chosen to match the constant factors in the O notation for the actual time).

Thus, the root of each tree in a heap has one unit of time stored. This unit of time can be used later to link this tree with another tree at amortized time 0. Also, each marked node has two units of time stored. One can be used to cut the node from its parent. If this happens, the node becomes a root and the second unit of time will remain stored in it as in any other root.

DISJOINT SETS

A disjoint set data structure maintains a collection S={ S1 , S2 ,…, Sk } of disjoint dynamic sets. Each set is identified by a representative, which usually is a member in the set.

1 Operations on Sets

Let x be an object. We wish to support the following operations.

MAKE-SET(x) creates a new set whose only member is pointed by x; Note that x is not in the other sets.

UNION(x,y) unites two dynamic sets containing objects x and y, say Sx and Sy , into a new set that Sx∪Sy , assuming that Sx ∩ Sy =∅;

FIND-SET(x) returns a pointer to the representative of the set containing x.

INSERT(a,S) inserts an object a to S, and returns S∪{a}.

DELETE(a,S) deletes an object a from S, and returns S-{a}.

SPLIT(a,S) partitions the objects of S into two sets S1 and S2 such that S1 ={b | b≤a b∈S}, and S2 =S- S1 .

MINIMUM(S) returns the minimum object in S.

2 Applications of Disjoint-set Data Structures

Here we show two application examples.

Connected components (CCs)

Minimum Spanning Trees (MSTs)

2.1 Algorithm for Connected Components

```
CONNECTED-COMPONENTS (G)
1  for each v∈V
2        do  MAKE-SET (v)
3  for every edge (u,v)∈E
4        do if  FIND-SET (u) ≠ FIND-SET (v)
5              then  UNION (u,v)
SAME-COMPONENTS (u,v)
1  if  FIND-SET (u) = FIND-SET (v)
2      then return TRUE
3  return FALSE
```

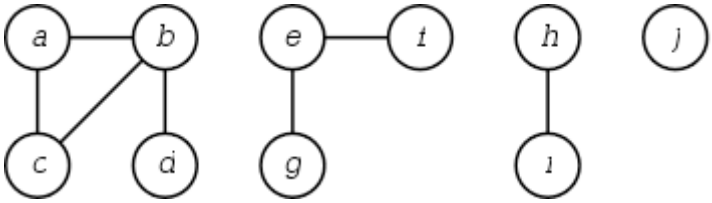


Figure 1: A graph with four connected components: {a,b,c,d}, {e,f,g}, {h,i}, and {j}.

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}					{e,g}	{f}		{h,i}	{j}
(e,f)	{a,b,c,d}					{e,f,g}			{h,i}	{j}
(b,c)	{a,b,c,d}					{e,f,g}			{h,i}	{j}

Table 1: This table shows the state of the collection of disjoint sets as each edge is processed. The processed edge is listed on the left and the rest of the columns show the state of the collection.

3 The Disjoint Set Representation

3.1 The Linked-list Representation

A set can be represented by a linked list. In this representation, each node has a pointer to the next node and a pointer to the first node.

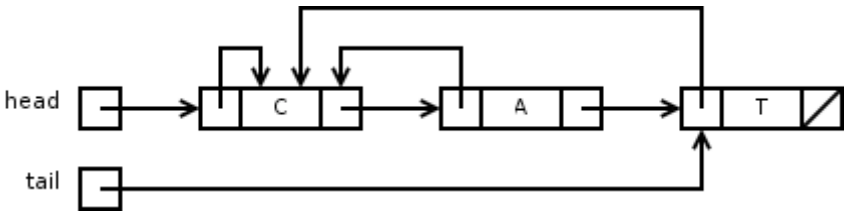


Figure 2: Linked-list representation. Each disjoint set can be represented by a linked-list. Each node has a pointer to the head node and a pointer to the next node.

Consider the following operation sequence:

MAKE-SET (x1),

MAKE-SET (x2),

:,

MAKE-SET (xn-1),

MAKE-SET (xn),

UNION (x1 , x2),

UNION (x2 , x3),

:,

UNION (xn-1 , xn).

What's the total time complexity of the above operations?

A weighted-union heuristic: Assume that the representative of each set maintains the number of objects in the set, and always merge the smaller list to the larger list, then

Theorem: Using the linked list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, Union, and Find_Set operations, n of which are MAKE-SET, takes O(m+nlogn) time.

Hint: observe that for any $k \leq n$, after x's representative pointer has been updated $\lceil \log k \rceil$ times, the resulting set containing x must have at least k members.

4 Disjoint Set Forests

A faster implementation of disjoint sets is through the rooted trees, with each node containing one member and each tree representing one set. Each member in the tree has only one parent.

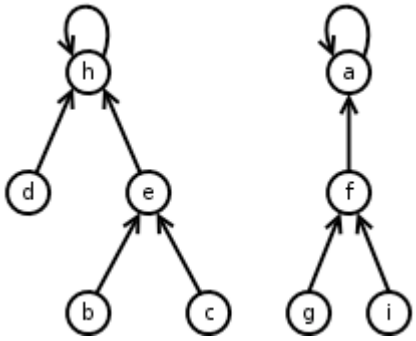


Figure 3: Rooted tree representation of disjoint sets. Each tree has a "representative" h for the left tree and a for the right tree.

4.1 The Heuristics for Disjoint Set Operations

1. union by rank.
2. path compression

The idea of union by rank is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, for each node we maintain a rank that approximates the logarithm of the subtree size and is also an upper bound on the height of the node.

Time complexity: O(mlogn), assuming that there are m union operations.

Path compression is quite simple and very effective. We use this approach during FIND-SET operations to make each node on the path point directly to the root. Path compression does not change any ranks.

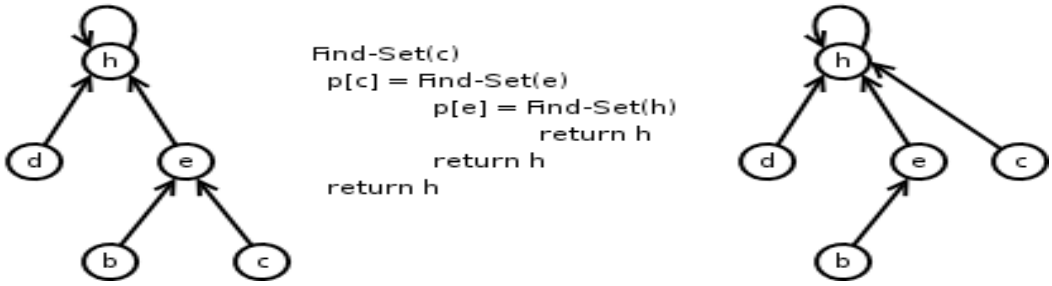


Figure 4: Path compression takes place during the FIND-SET operation. This works by recursing from the given input node up to the root of the tree, forming a path. Then the root is returned and assigned as the parent for each node in path. The parent of c after FIND-SET (c) is h.

Time complexity: $\Theta(f \log 1+f/n)\ n)$ if $f \geq n$ and $\Theta(n+f\log n)$ otherwise, assume that there are n MAKE-SET operations and f FIND-SET operations.

```
MAKE-SET (x)

1    p[x] ← x

2    rank[x] ← 0

FIND-SET (x)

1   if x≠p[x]

2        then p[x] ← FIND-SET (p[x])

3   return p[x]

UNION (x,y)

1    LINK( FIND-SET (x), FIND-SET (y))

LINK (x,y)

1   if rank[x]>rank[y]

2        then p[y] ← x

3        else  p[x] ← y

4                if rank[x]=rank[y]

5                    then rank[y] ← rank[y]+1
```

where rank[x] is the height of x in the tree. If both of the above methods are used together, the time complexity is $O(m\alpha(m,n))$.

The Rank properties

$rank[x] \leq rank[p[x]]$

for any tree root x, $size(x) \geq 2rank[x]$ (Link operation)

for any integer r, there are at most $n/ 2r$ nodes of rank r

each node has rank at most $\lfloor \log n \rfloor$, assuming there are at n objects involved.

AMORTIZED ALGORITHM

Amortized analysis is a method of analyzing algorithms that considers the entire sequence of operations of the program. It allows for the establishment of a worst-case bound for the performance of an algorithm irrespective of the inputs by looking at all of the operations. This analysis is most commonly discussed using big O notation.

At the heart of the method is the idea that while certain operations may be extremely costly in resources, they cannot occur at a high enough frequency to weigh down the entire program because the number of less costly operations will far outnumber the costly ones in the long run, "paying back" the program over a number of iterations. It is particularly useful because it defines the worst-case limit of a program's performance rather than making assumptions about the state of the program.

METHOD

The method requires knowledge of which series of operations are possible. This is most commonly the case with data structures, which have state that persists between operations. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost.

There are generally three methods for performing amortized analysis: the aggregate method, the accounting method, and the potential method. All of these give the same answers, and their usage difference is primarily circumstantial and due to individual preference.

Aggregate analysis determines the upper bound T(n) on the total cost of a sequence of n operations, then calculates the amortized cost to be T(n) / n.

The accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a "debt" of unfavorable state in small increments, while rare long-running operations decrease it drastically.

The potential method is like the accounting method, but overcharges operations early to compensate for undercharges later.

COMMONLY USED

In common usage, an "amortized algorithm" is one that an amortized analysis has shown to perform well.

Online algorithms commonly use amortized analysis.

ACCOUNTING METHOD

In this method, we assign changes to different operations, with some operations charged more or less than they actually cost. In other words, we assign artificial charges to different operations.

Any overcharge for an operation on an item is stored (in an bank account) reserved for that item.

Later, a different operation on that item can pay for its cost with the credit for that item.

The balanced in the (bank) account is not allowed to become negative.

The sum of the amortized cost for any sequence of operations is an upper bound for the actual total cost of these operations.

The amortized cost of each operation must be chosen wisely in order to pay for each operation at or before the cost is incurred.

Application 1: Stack Operation

Recall the actual costs of stack operations were

PUSH (s, x) 1

POP (s) 1

MULTIPOP (s, k) min(k,s)

The amortized cost assignments are

PUSH 2

POP 0

MULTIPOP 0

Observe that the amortized cost of each operation is O(1). We must show that one can pay for any sequence of stack operations by charging the amortized costs.

The two units costs collected for each PUSH is used as follows:

1 unit is used to pay the cost of the PUSH.

1 unit is collected in advanced to pay for a potential future POP.

Therefore, for any sequence for n PUSH, POP, and MULTIPOP operations, the amortized cost is an

Ci= j=1Si 3 - Ciactual = 3i - (2floor(lg1) + 1 + i -floor(lgi) - 2)

If i = 2k, where k ≥ 0, then

Ci = 3i - (2k+1 + i - k -2) = k + 2

If i = 2k + j, where k ≥ 0 and 1 ≤ j ≤ 2k, then

Ci = 3i - (2k+1 + i - k - 2) = 2j + k + 2

This is an upperbound on the total actual cost. Since the total amortized cost is O(n) so is the total cost.

As an example, consider a sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. The accounting method of amortized analysis determine the amortized cost per operation as follows:

Let amortized cost per operation be 3, then the credit Ci after the ith operation is: Since k ≥ 1 and j ≥ 1, so credit Ci always greater than zero. Hence, the total amortized cost 3n, that is O(n) is an upper bound on the total actual cost. Therefore, the amortized cost of each operation is O(n)/n = O(1).

Another example, consider a sequence of stack operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made. We must show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

There are,ofcourse, many ways to assign amortized cost to stack operations. One way is:

PUSH 4,

POP 0,

MULTIPOP 0,

STACK-COPY 0.

Every time we PUSH, we pay a dollar (unit) to perform the actual operation and store 1 dollar (put in the bank). That leaves us with 2 dollars, which is placed on x (say) element. When we POP x element off the stack, one of two dollar is used to pay POP operation and the other one (dollar) is again put into a bank account. The money in the bank is used to pay for the STACK-COPY operation. Since after kk dollars in the bank and the stack size is never exceeds k, there is enough dollars (units) in the bank (storage) to pay for the STACK-COPY operations. The cost of n stack operations, including copy the stack is therefore O(n). operations, there are atleast

Application 2: Binary Counter

We observed in the method, the running time of INCREMENT operation on binary counter is proportion to the number of bits flipped. We shall use this running time as our cost here.

For amortized analysis, charge an amortized cost of 2 dollars to set a bit to 1.

When a bit is set, use 1 dollar out of 2 dollars already charged to pay the actual setting of the bit, and place the other dollar on the bit as credit so that when we reset a bit to zero, we need not charge anything.

The amortized cost of psuedocode INCREMENT can now be evaluated:

```
INCREMENT (A)
1. i = 0
2. while i < length[A] and A[i] = 1
3.       do A[i] = 0
4.          i = i +1
5. if i < length [A]
6.          then A[i] = 1
```

Within the while loop, the cost of resetting the bits is paid for by the dollars on the bits that are reset.At most one bit is set, in line 6 above, and therefore the amortized cost of an INCREMENT operation is at most 2 dollars (units). Thus, for n INCREMENT operation, the total amortized cost is O(n), which bounds the total actual cost.

Consider a Variant

Let us implement a binary counter as a bit vector so that any sequence of n INCREMENT and RESET operations takes O(n) time on an initially zero counter,. The goal here is not only to increment a counter but also to read it to zero, that is, make all bits in the binary counter to zero. The new field , max[A], holds the index of the high-order 1 in A. Initially, set max[A] to -1. Now, update max[A] appropriately when the counter is incremented (or reset). By contrast the cost of RESET, we can limit it to an amount that can be covered from earlier INCREMENT'S.

```
INCREMENT (A)
1. i = 1
2. while i < length [A] and A[i] = 1
3.     do A[i] = 0
4.         i = i +1
5. if i < length [A]
6.     then A[i] = 1
7.     if i > max[A]
8.         then max[A] = i
9.         else max[A] = -1
```

Note that lines 7, 8 and 9 are added in the CLR algorithm of binary counter.

```
RESET(A)
For i = 0 to max[A]
do A[i] = 0
max[A] = -1
```

For the counter in the CLR we assume that it cost 1 dollar to flip a bit. In addition to that we assume that we need 1 dollar to update max[A]. Setting and Resetting of bits work exactly as the binary counter in CLR: Pay 1 dollar to set bit to 1 and placed another 1 dollar on the same bit as credit. So, that the credit on each bit will pay to reset the bit during incrementing.

In addition, use 1 dollar to update max[A] and if max[A] increases place 1 dollar as a credit on a new high-order 1. (If max[A] does not increase we just waste that one dollar). Since RESET manipulates bits at some time before the high-order 1 got up to max[A], every bit seen by RESET has one dollar credit on it. So, the zeroing of bits by RESET can be completely paid for by the credit stored on the bits. We just need one dollar to pay for resetting max[A].

Thus, charging 4 dollars for each INCREMENT and 1 dollar for each RESET is sufficient, so the sequence of n INCREMENT and RESET operations take O(n) amortized time.

This method stores pre-payments as potential or potential energy that can be released to pay for future operations. The stored potential is associated with the entire data structure rather than specific objects within the data structure.

Notation:

D0 is the initial data structure (e.g., stack)

Di is the data structure after the ith operation.

ci is the actual cost of the ith operation.

The potential function Ψ maps each Di to its potential value Ψ(Di)

The amortized cost ^ci of the ith operation w.r.t potential function Ψ is defined by

^ci = ci + Ψ(Di) - Ψ (Di-1) --------- (1)

The amortized cost of each operation is therefore

^ci = [Actual operation cost] + [change in potential].

By the eq.I, the total amortized cost of the n operation is

ni=1 ^ci = ni=1(ci + Ψ(Di) - Ψ (Di-1))

= ni=1 ci + ni=1 Ψ(Di) - ni=1Ψ (Di-1)

= ni=1 ci + Ψ(D1) + Ψ(D2) + . . . + Ψ (Dn-1) + Ψ(Dn) - {Ψ(D0) + Ψ(D1) + . . . + Ψ (Dn-1)

= ni=1 ci + Ψ(Dn) - Ψ(D0) ---------- (2)

If we define a potential function Ψ so that Ψ(Dn) ≥ Ψ(D0), then the total amortized cost ni=1 ^ci is an upper bound on the total actual cost.

As an example consider a sequence of n operations performed on a data structure. The ith operation costs i if i is an exact power of 2 and 1 otherwise. The potential method of amortized determines the amortized cost per operation as follows:

Let Ψ(Di) = 2i - 2ëlgi+1û + 1, then

Ψ(D0) = 0. Since 2ëlgi+1û ≤ 2i where i >0 ,

Therefore, Ψ(Di) ≥ 0 = Ψ(D0)

If i = 2k where k ≥ 0 then

2ëlgi+1û = 2k+1 = 2i

2ëlgiû = 2k = i

^ci = ci + Ψ(Di) - Ψ(Di-1)

= i + (2i -2i+1) -{2(i-1)-i+1}

= 2If i = 2k + j where k ≥ 0 and 1 ≤ j ≤ 2k

then 2ëlgi+1û = 2[lgi]

^ci = ci + Ψ(Di) - Ψ(Di-1) = 3

Because ni=1^ci = ni=1 ci + Ψ(Dn) - Ψ(D0)

and Ψ(Di) ≥ Ψ(D0), so, the total amortized cost of n operation is an upper bound on the total actual cost. Therefore, the total amortized cost of a sequence of n operation is O(n) and the amortized cost per operation is O(n) / n = O(1).

Application 1- Stack Operations

Define the potential function Ψ on a stack to be the number of objects in the stack. For empty stack D0 , we have Ψ(D0) = 0. Since the number of objects in the stack can not be negative, the stack Di after the ith operation has nonnegative potetial, and thus

Ψ(Di) ≥ 0 = Ψ(D0).

Therefore, the total amortized cost of n operations w.r.t. function Ψ represents an upper bound on the actual cost.

Amortized costs of stack operations are:

PUSH

If the ith operation on a stack containing s object is a PUSH operation, then the potential difference is

Ψ(Di) - Ψ(Di-1) = (s + 1) - s = 1. In simple words, if ith is PUSH then (i-1)th must be one less. By equation I, the amortized cost of this PUSH operation is ^ci = ci + Ψ(Di) - Ψ(Di-1) = 1 + 1 = 2

MULTIPOP

If the ith operation on the stack is MULTIPOP(S, k) and k` = min(k,s) objects are popped off the stack.

The actual cost of the operation is k`, and the potential difference is

Ψ(Di) - Ψ(Di-1) = -k`

why this is negative? Because we are taking off item from the stack. Thus, the amortized cost of the MULTIPOP operation is

^ci = ci + Ψ(Di) - Ψ(Di-1) = k`-k` = 0

POP

Similarly, the amortized cost of a POP operation is 0.

Analysis

Since amortized cost of each of the three operations is O(1), therefore, the total amortized cost of n operations is O(n). The total amortized cost of n operations is an upper bound on the total actual cost.

Lemma If data structure is Binary heap: Show that a potential function is O(nlgn) such that the amortized cost of EXTRACT-MIN is constant.

Proof

We know that the amortized cost ^ci of operation i is defined as

$\hat{c}_i = c_i + \Psi(D_i) - \Psi(D_{i-1})$

For the heap operations, this gives us

$c_1 \lg n = c_2 \lg(n+c_3) + \Psi(D_i) - \Psi(D_{i-1})$ (Insert) ----------(1)

$c_4 = c_5 \lg(n + c_6) + \Psi(D_i) - \Psi(D_{i-1})$ (EXTRACT-MIN) -----(2)

Consider the potential function $\Psi(D) = \lg(n!)$, where n is the number of items in D.

From equation (1), we have

$(c_1 - c_2)\lg(n + c_3) = \lg(n!) - \lg((n-1)!) = \lg n.$

This clearly holds if $c_1 = c_2$ and $c_3 = 0$.

From equation (2), we have

$c_4 - c_5 \lg(n + c_6) = \lg(n!) - \lg((n+1)!) = -\lg(n+1).$

This clearly holds if $c_4 = 0$ and $c_4 = c_6 = 1$.

Remember that stirlings function tells that $\lg(n!) = \theta(n \lg n)$, so

$\Psi(D) = \theta(n \lg n)$

And this completes the proof.

Application 2: Binary Counter

Define the potential of the counter after ith INCREMENT operation to be bi, the number of 1's in the counter after ith operation.

Let ith INCREMENT operation resets ti bits. This implies that actual cost = atmost (ti + 1).

Why? Because in addition to resetting ti it also sets at most one bit to 1.Therefore, the number of 1's in the counter after the ith operation is therefore $b_i \leq b_{i-1} - t_i + 1$, and the potential difference is

$\Psi(D_i) - \Psi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$

Putting this value in equation (1), we get

$\hat{c}_i = c_i + \Psi(D_i) - \Psi(D_{i-1})$

$= (t_i + 1) + (1 - t_i)$

$= 2$

If counter starts at zero, then $\Psi(D_0) = 0$. Since $\Psi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n INCREMENT operation is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is O(n).

If counter does not start at zero, then the initial number are 1's (= b0).

After 'n' INCREMENT operations the number of 1's = bn, where $0 \leq b0, bn \leq k$.

Since $ni=1^ci = (ci + \Psi(Di) + \Psi(Di-1))$

$=>ni=1^ci = ni=1\ ci + ni=1\ \Psi(Di) + ni=1\ \Psi(Di-1)$

$=>ni=1^ci = nSi=1\ ci + \Psi(Dn) - \Psi(D0)$

$=>ni=1ci = ni=1\ ^ci + \Psi(D0) - \Psi(Dn)$

We have $^ci \leq 2$ for all $1 \leq i \leq n$. Since $\Psi(Di) = b0$ and $\Psi(Dn) = b$, the total cost of n INCREMENT operation isSince $ni=1ci = ni=1\ ^ci + \Psi(Dn) + \Psi(D0)$

$\leq ni=1\ 2 - bn + b0$ why because $ci \leq 2$

$= 2\ ni=1 - bn + b0$

$= 2n - bn + b$

Note that since $b0 \leq k$, if we execute at least $n = \Omega(k)$ INCREMENT Operations, the total actual cost is O(n), no matter what initial value of counter is.

Implementation of a queue with two stacks, such that the amortized cost of each ENQUEUE and each DEQUEUE Operation is O(1). ENQUEUE pushes an object onto the first stack. DEQUEUE pops off an object from second stack if it is not empty. If second stack is empty, DEQUEUE transfers all objects from the first stack to the second stack to the second stack and then pops off the first object. The goal is to show that this implementation has an O(1) amortized cost for each ENQUEUE and DEQUEUE operation. Suppose Di denotes the state of the stacks after ith operation. Define $\Psi(Di)$ to be the number of elements in the first stack. Clearly, $\Psi(D0) = 0$ and $\Psi(Di) \geq \Psi(D0)$ for all i. If the ith operation is an ENQUEUE operation, then $\Psi(Di) - \Psi(Di-1) = 1$

Since the actual cost of an ENQUEUE operation is 1, the amortized cost of an ENQUEUE operation is 2. If the ith operation is a DEQUEUE, then there are two case to consider.

Case i: When the second stack is not empty.

In this case we have $\Psi(Di) - \Psi(Di-1) = 0$ and the actual cost of the DEQUEUE operation is 1

Case ii: When the second stack is empty.

In this case, we have $\Psi(Di) - \Psi(Di-1) = - \Psi(Di-1)$ and the actual cost of the DEQUEUE operation is $\Psi(Di-1) + 1$

In either case, the amortize cost of the DEQUEUE operation is 1. It follows that each operation has O(1) amortized cost

AGGREGATE ANALYSIS

Aggregate analysis is the process through which your new lender will determine your monthly impound payment and the initial deposit that you will be required to make at closing into the impound account.

Aggregate analysis is the process through which your new lender will determine your monthly impound payment and the initial deposit that you will be required to make at closing into the impound account.

Your monthly impound payment is determined by estimating your annual tax and insurance expenses and dividing this by twelve. To determine the required initial deposit, your lender will estimate what the balance will be in your account at the end of each month for one year, beginning with the month in which your first payment is due. This is accomplished by projecting the payments into and disbursements out of your account during this period. Based on this projection, your lender will collect from you at the time of closing the amount of money necessary to ensure that your account is not in a negative position, including any cush- ion permitted by federal and state law.

BENEFITS

Prior to regulatory changes, many lenders used a process known as single item analysis to determine your initial impound deposit. Since this process analyzed each impound account item individually, it often resulted in a larger initial deposit being collected from the borrower. By using aggre- gate analysis, the amount of money the lender will require you to deposit in your impound account at closing will often be less than or equal to what the lender would have required under single item analysis.

RED BLACK TREE

A red-black tree is a binary search tree with one extra attribute for each node: the colour, which is either red or black. We also need to keep track of the parent of each node, so that a red-black tree's node structure would be:

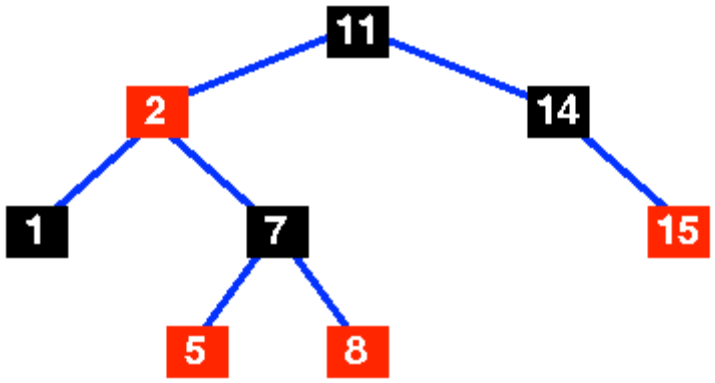
```
        structt_red_black_node {
enum { red, black } colour;
void *item;
structt_red_black_node *left,
                       *right,
                       *parent;
    }
```

For the purpose of this discussion, the NULL nodes which terminate the tree are considered to be the leaves and are coloured black.

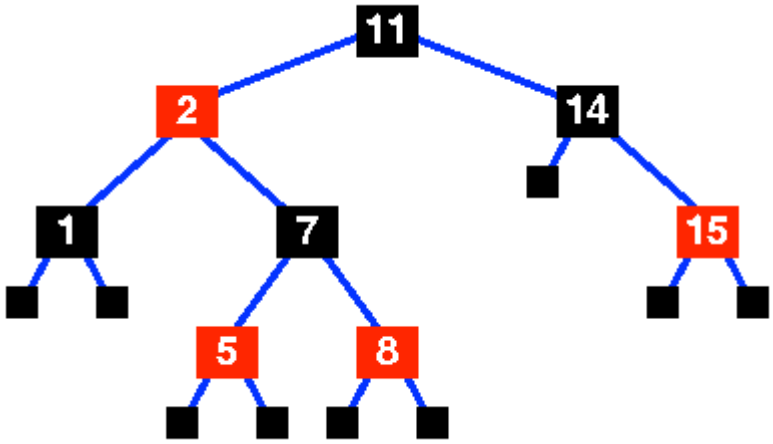
Definition of a red-black tree

A red-black tree is a binary search tree which has the following red-black properties:

1. Every node is either red or black.
2. Every leaf (NULL) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.



A basic red-black tree



Basic red-black tree with the sentinel nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

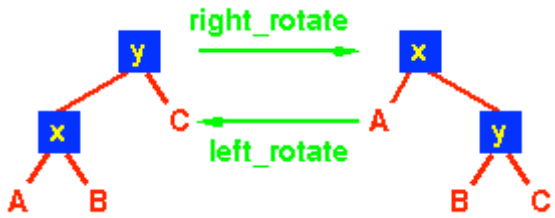
TheLemma

A red-black tree with n internal nodes has height at most 2log(n+1).

This demonstrates why the red-black tree is a good search tree: it can always be searched in O(log n) time.

As with heaps, additions and deletions from red-black trees destroy the red-black property, so we need to restore it. To do this we need to look at some operations on red-black trees.

Rotationsy are the NULL black nodes of property 2.



A rotation is a local operation in a search tree that preserve

s in-order traversal key ordering.

Note that in both trees, an in-order traversal yields:A x B y C

The left_rotate operation may be encoded:

```
left_rotate( Tree T, node x ) {

node y;

    y = x->right;

    /* Turn y's left sub-tree into x's right sub-tree */

x->right = y->left;

if ( y->left != NULL )

y->left->parent = x;
```

```
   /* y's new parent was x's parent */
y->parent = x->parent;
   /* Set the parent to point to y instead of x */
   /* First see whether we're at the root */
if ( x->parent == NULL ) T->root = y;
else
if ( x == (x->parent)->left )
        /* x was on the left of its parent */
x->parent->left = y;
else
        /* x must have been on the right */
x->parent->right = y;
   /* Finally, put x on y's left */
y->left = x;
x->parent = y;
```

Insertion

Insertion is somewhat complex and involves a number of cases. Note that we start by inserting the new node, x, in the tree just as we would for any other binary tree, using the tree_insert function. This new node is labelled red, and possibly destroys the red-black property. The main loop moves up the tree, restoring the red-black property.

```
rb_insert( Tree T, node x ) {
    /* Insert in the tree in the usual way */
tree_insert( T, x );
    /* Now restore the red-black property */
x->colour = red;
while ( (x != T->root) && (x->parent->colour == red) ) {
if ( x->parent == x->parent->parent->left ) {
           /* If x's parent is a left, y is x's right 'uncle' */
           y = x->parent->parent->right;
if ( y->colour == red ) {
               /* case 1 - change the colours */
x->parent->colour = black;
y->colour = black;
x->parent->parent->colour = red;
               /* Move x up the tree */
               x = x->parent->parent;
               }
else {
               /* y is a black node */
if ( x == x->parent->right ) {
                   /* and x is to the right */
                   /* case 2 - move x up and rotate */
```

```
                    x = x->parent;
left_rotate( T, x );
                    }
                /* case 3 */
x->parent->colour = black;
x->parent->parent->colour = red;
right_rotate( T, x->parent->parent );
                }
            }
else {
            /* repeat the "if" part with right and left
exchanged */
            }
        }
    /* Colour the root black */
    T->root->colour = black;
    }
```