Algortihms and datastructures

Kristoffer Klokker

2022

Contents

1 Algorithm analyse

An algorithm must stop for all input and give the correct output.
An algorithms quality is determined from:

- Speed
- Memory used
- Complexity of implmentation
- Other use cases like stability

1.1 Algorithm speed

The measure speed and memory the worst case of the algorithm is always used due to its simplicity in calculations and gurantee.
The measurement is used as a function of input size using the big O notation, which says for a given input with n size it will take n^2 time to run for example. In reality the speed of an algorithm is not equivalent to the input size due to small constant in every operation but these are so miniscule, they should not be considered. A function may be tuned to have lower constants but this is more a topic for theoretical algorithm optimization.
When comparing algorithm speed the math relations symbols are replaced as such:

- $=-\Theta$ Theta
- $\leq - O$ O
- $\geq - \Omega$ Omega
- $<- o$ little o
- $>- \omega$ little omega

When comparing the speed of two algorithms the following methods can be used:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \qquad f(n) = \Theta(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad f(n) = o(g(n))$$

The generel speeds of alogirthms in order is:

$$1,\ \log_2 n,\ \sqrt{n},\ n,\ n\log_2 n,\ n\sqrt{n},\ n^2,\ n^3,\ n^{10},\ 2^n$$

1.2 Ram-model

To analyze an algorithm a model is used most often the ram model. The ram model is a very simple interpretation of a computer.
The model consist of a CPU, memory and basic operation (add, sub, mult, shift, move, jump)
The time of the algorithm is then measured in amount of basic operations done.
The memory is determined as the amount of memory cell used.

2 Sorting algorithms

Sorting algorithms are used to sort an array of items in an ascending order.

2.1 Insertion sort

One of the most simple sorting algorithms.
Works by going through the list from index 1 and moves moves every entry before the element 1 up until the element is to the right of an element smaller than the element.
This will therefore have a run time of $O(n^2)$ due to the scenario where the array is in decending order where it will moves $n-1, n-2, .., 1$ elements which is $\sum_{j=1}^{n} j = \frac{(n-1)n}{2} \leq \frac{n^2-n}{2} = n^2$

```
1: Insertion-Sort(A, n)
2: for j = 1 to n do
3:     key = A[j]
4:     i = j − 1
5:     while i ≥ 0 and A[i] > key do
6:         a[i + 1] = A[i]
7:         i = i − 1
8:     end while
9:     A[i + 1] = key
10: end for
```

2.2 Selectionsort

Selectionsort is done by taking all elements and then searching for the smallest element and inserting it into the list.
This will result in the same run time $O(n^2)$ due to the same reasoning as

insertionsort.

```
1: Selection-Sort(A, n)
2: Array output = newArray(n)
3: i = 0
4: while i < n do
5:     output[i] = A[i]
6:     A[i].remove()
7:     i++
8: end while
```

2.3 Merge sort

Merge sort is an sorting algorithm which is based on the simple merge of two sorted list. Here a new list is created and for every element of the two list, is the lowest value choosen from the first place of the list and putted into the new list. This is is done until both list are empty and when sorting the undefined element is simply equal to infinity when compared.

To make this into an sorting algorithm, every element in an unsorted list is made into to list of length 1. Then merge is performed on the lists in pairs of two, from which new sorted lists emerges. This is done until a single lists is left with all the elements which is now sorted.

The speed of this sorting algorithm will therefore be $n\lceil \log_2 n \rceil$ due to for every level of merge there will be performed n operations. The amount of levels of merges are equal to $\log_2 n$ due to first there will be n lists, then $n/2$ lists, and then $n/2^2$. This will continue until $n/2^k = 1$ which rearranged to find k will be $log_2 n = k$.
The ceiling of $\log_2 n$ is due to in scenarios where the list length is not a potens of two and a list has no pair in this case it will go on to the next level and thus creating a new level once the list has 1 more item than a potens of two.

2.4 Quick sort

A recursive sorting algorithm which runs mostly at $O(n \log n)$ time but worst case is $O(n^2)$.
The sorting algorithm is smart due to it not needing subarrays copies and working in the original array.
The main idea, is choosing an element in the array, then sorting everything

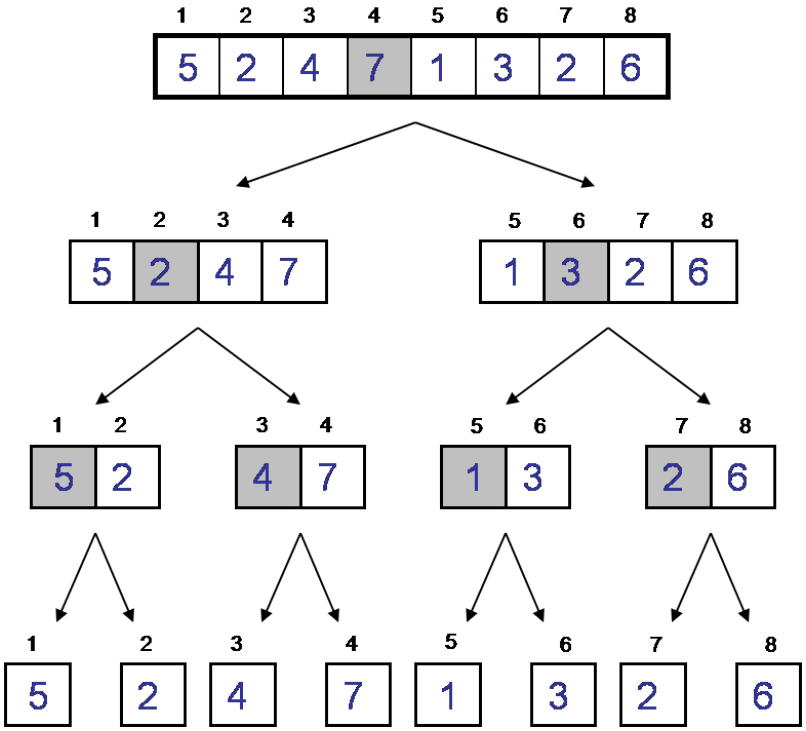

Figure 1: Merge sort list split up illustration from j2eereference.com

in two baskets lower or equal and higher.
Then the element is placed between the two baskets and quick sort is ran upon the two baskets.

```
1: Quick-Sort(A, p, r)
2: if r ≤ p then
3:     return;
4: end if
5: i = p
6: for j = p to r do
7:     if A[j] ≤ x then
8:         temp = A[i];
9:         A[i] = A[j];
10:        A[j] = temp;
11:        i = i + 1;
12:    end if
13: end for
14: temp = A[i];
15: A[i] = A[r];
16: A[r] = temp;
17: Quick − Sort(A, p, i − 1);
18: Quick − Sort(A, i + 1, r);
```

The arguments is the array A the start of the sort p and the end r.
Here it is seen that in the for loop if the current element is smaller than the choosen element it is switched with the first element in larger basket such that the smaller basket is in front.
In case the element is larger nothing is done. Then at the end the choosen element is switched with the first element of the larger bucket.

2.5 Heap sort

Heap sort is based on the Heap and runs in $O(n \log n)$ and uses no extra space. The heap which is a binary tree which follows heap order. The heap order dictates that a parent should always be greater or equal to its children. The tree should also have heap shape, which is all branches should be the saem height with a maximum difference of 1. In case of a larger branch it should be on the left side of the tree.
This heap can is illustrated as a tree but is actually an array. Here the arrays first entry is the root, the roots children will then be at index 2 times index and 2 times index + 1
This means the parent is at $\lfloor i/2 \rfloor$ relative to the child. Heap sort then uses

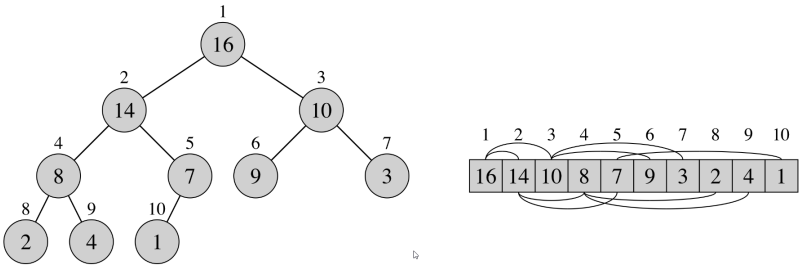

Figure 2: The heap illustration and the heap array

the heap array, and when sorting the first entry or the root is switched with the last entry of the heap array.
By this the size of the heap array shrinks with 1 and the rest of the array is the sorted array.
When switched the heap is reconstructed.
The reconstruction process is as follows:

1: Max-Heapify(A, i, n)
2: $l = 2i$;
3: $r = 2i + 1$;
4: **if** $l \leq n$ and $A[l] > A[i]$ **then**
5: $\quad largest = l$;
6: **else**
7: $\quad largest = i$;
8: **end if**
9: **if** $r \leq n$ and $A[r] > A[largest]$ **then**
10: $\quad largest = r$;
11: **end if**
12: **if** $largest! = i$ **then**
13: $\quad temp = A[i]$;
14: $\quad A[i] = A[largest]$;
15: $\quad A[largest] = temp$;
16: $\quad Max - Heapify(A, largest, n)$;
17: **end if**

The n is used as the length of the heap and i is the current node to check. The first if check if the left child is larger, the second checks if the right child is larger, and the third check if something should be changed and if then call itself upon the newly changed child.
With this function in mind heap sort becomes:

1: HeapSort(A, n)
2: Max-Heapify(A,n);

3: **for** $i = n$; $2 < i$; $i - -$; **do**
4: $temp = A[i]$;
5: $A[i] = A[0]$;
6: $A[0] = temp$;
7: Max-Heapify$(A, 1, i - 1)$;
8: **end for**

The run time of this is $O(n \log n)$ dut to the trees height will maximally be $\log n$ which will be the number of exhanges n times.

2.6 Counter sort

Counter sort works in $O(n + k)$ time on arrays of objects represented by integers by having an array with k length. Where k is the highest integer.
It then goes through the orginal array and takes every elements value and uses it as index for the k array and itterate by 1.
Then a new array with length n is made and every element in the k array is added to the new array with the value of elements index as many times as the value of element.

2.7 Radix sort

Radis sort runs in $O(d(n + k))$ time, where d is amount of digits in largest number and k is the max value of the base of the number system. First Counter sort is done upon every elements lasts digit. Then from the new array counter sort is done again on the second lasts digit. This is repeated as many times as the most digits in the largest number.
This therefore means in the counter sort, the k array only has the length the max value of the number system and instead of itterate it is now a list with references.
This therefore solves the issue of now knowing the largest digit, and the temperary array is a lot shorter.

3 algorithms

3.1 Divide-and-Conquer algorithms

The same as recursive algorithms.
When illustrated a tree is often used to describe the calls and calls backs
When finding the run time of divide and conquer algorithms, the master theorum is used.

It states:

$$T(n) = aT(n/b) + f(n)$$

Where a is the amount of branches created by the algorithm and b is the amount each branch elements is divided by.
$f(n)$ is the run time of the node, for sorting or creating new branches.
It can here be seen that all divide and conquer algorithms fall into 3 categories:

- if $f(n) = O(n^{\log_b(a)-\epsilon})$ then the run time is dominated by the leaves and the running time is $T(n) = \Theta(n^{\log_b(a)})$
- if $f(n) = \Theta(n^{\log_b(a)})$ then the run time is even on all layers of the tree and the running time is $T(n) = \Theta(n^{\log_b(a)} \cdot \log(n)$
- if $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ then the root is the dominating rung time and the running time is $T(n) = \Theta(f(n))$

Where ϵ is some constant bigger than 0.
Remember: $O- \leq$, $\Theta- =$ and $\Omega- \geq$
If the theorum does not apply, an alternative is writing, a tree with a children at each node, the number of elements in the node b and then then sum up all layers run time.

There is instances where to function may fall into two of the cases, for example $T(n) = @T(n/2) + n \log n$.
This case it will fall between case 1 and 2 and can only be found through the tree method.
It shall be noted that, in the case of an uneven divide of the tree, but it does not matter in the runtime since it will only differ with a few constants.

3.1.1 Strassen matrix multiplikation

Strassen mutliplikation is an algorithm which recursively split up a matrix such the run time get to $O(n^2)$ over the normal calculation method which takes $O(n^3)$
In a given matrix over the size 2x2 the matrix will be divided up in four matrix of each corner. Then strassen discovered a method such only 7 calculation is needed in a 2x2 matrix instead of 8. This therefore makes the problem into a $T(n) = 7T(n/2) + n^2$, 7 from the amount of problems, n/2 by the matrix size always halfing and n^2 for each of the 7 problems taking. Therfore makign the runtime $O(n^2)$ using the master theorum. The 7 calculation is:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

- $p1 = a(f - h)$
- $p2 = (a + b)h$
- $p3 = (c + d)e$
- $p4 = d(g - e)$
- $p5 = (a + d)(e + h)$
- $p6 = (b - d)(g + h)$
- $p7 = (a - c)(e + f)$

3.1.2 Dynamic programming

Dynamic programming is used in recursive algorithms which may have, re-calculate a lot of subtasks.
A classic example is fibonacci sequence, where the results are stored in an array, such when f(n)=f(n-1)+f(n-2) is calculated only the first time it is recursively calculated and f(n-2) has already all values in the array.

3.1.3 Longest common subsequence

To find the longest common subsequence of two string a 2D dyncamic approach can be taken.
The common subsequence is the lenght of the sequence which can be found in both strings.
When finding the sequence it starts and the first character of each string. From here there are 3 cases:

- They match, so it returns LCS(n.substring(1),m.substring(1))
- They do not match, so it returns max(LCS(n.substring(1),m),LCS(n,m.substring(1))
- One of the strings are empty, returns 0

This is illustrated by, the first case an arrow is diagonal, second two arrow to the right and down are created. The last case is at the border where no subsequence is possible.
As seen it will create a path of longest sequence.
The run time is $O(mn)$ and it will also store mn spaces.

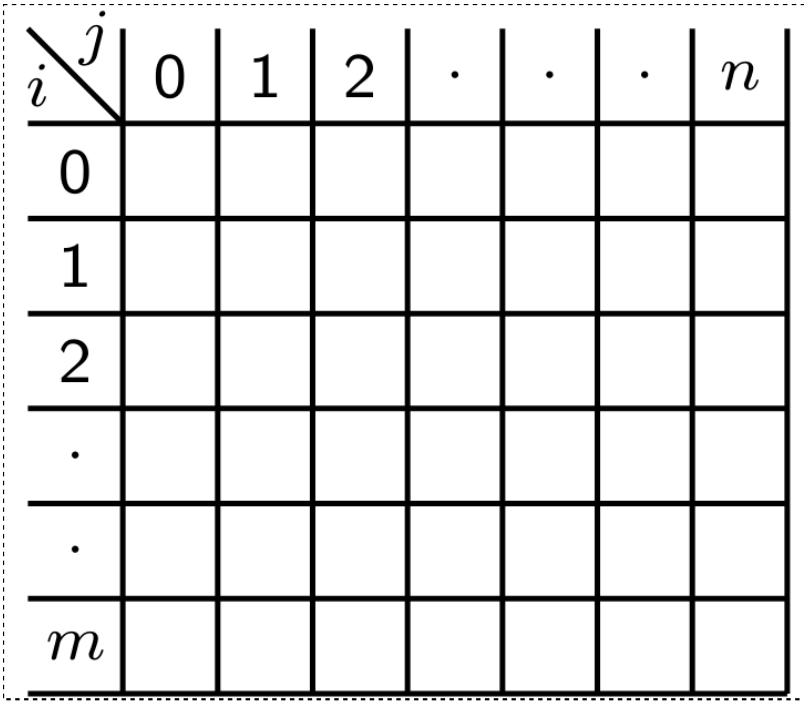

Figure 3: A 2D image of finding the longest subsequence

3.2 Lower bound of comparison sorting algorithms

To find the lower bound of comparison sorting algorithms, the comparison model can be used.
The model consist of a binary tree where every branch is a comparison, and the leafs are possible answers.
The run time here is the comparisons, therefore the run time to a leaf is the length of the path. Therefore making worst case height of tree.
It can therefore be seen when sorting an array of n elements there are $n!$ different rearrangement.
Each rearrangement needs a leaf therefore the tree has to have $n!$ leafs.
The maximum amount of leafs in a tree is 2^h where h is the height.
Therefore the minimum amount of height will be $2^h \geq n! \rightarrow h \geq \log n!$

$$
\begin{aligned}
h &\geq \log n! \\
&= \log(n \cdot (n-1) \cdot (n-2) \cdot ...) \\
&= \log n + \log(n-1) + ... \\
&\geq \frac{n}{2} \cdot \log(\frac{n}{2}) \\
&= \frac{n}{2}(\log(n) - 1)
\end{aligned}
$$

It can here be derived via the summation, that $h = \Omega(n \log n)$

3.3 Maximum summation of sequential elements in array

The most simple version of this, is two pointers with start of seqence and end, where for each element in the array the start is appointed from which all possible ends if summed and the largest value is taken.
This means the algorithm will take $O(n^3)$ due to first every element being appointed as start, from which all elements is tested as the end of sequence, from which the summation is done. This will round up to the worst case being $O(n^3)$
This can be optimized to a run time of $O(n)$, by instead of recalculating everything for every pointer a pointer is started at the start from which an end pointer is also at the start.
Here the largest summation is found, if it is larger than the best sequence it is assigned. Then the end pointer is moved forward and the new element is summed to the recent best sequence. If the sequence summation is now below zero the start pointer is moved to the end otherwise the end pointer is moved forward.

1: MaxSequence(A)
2: $max = 0$
3: $tempMax = 0$
4: $i = 0$
5: **while** $i < A.length$ **do**
6: $\quad tempMax = max(tempMax + A[i], 0)$
7: $\quad max = max(max, tempMax)$
8: $\quad i++$
9: **end while**

3.4 Hashing

Hashing is method of getting a idenfiable key to some data.
While the data may have range of indexible data, the hash function is a way to get a smaller index.
The motivation is seen in the following example.
Storing 5 credit card numbers in an array with size of every possible credit-card number would not be optimal.
A simple solution is taking the integer value of the object an using modulo on the desired size of the array.
This will create problems due to some object will collide, as a resolution a list can be created at every index which then can be added.
This will result in slower run time worst case being $O(1)$ for the index and $O(n)$ for finding it in the list.
An alternative is a binary search tree to optimize it to $O(\log n)$.
The ideal hash function will spread out objects as much as possible and a simple an reliable function being

$$h(k) = ((a \cdot k + b) \; mod \; p) \; mod \; m$$

Where p is a prime larger than the size of the array and a and b are random constant integers $1 \leq a \leq p-1$ and $0 \leq b \leq p-1$

3.4.1 Open addressing

An alternative to having a list as hash chain, is using different methods of finding next open position.
The following methods uses a given hash function $h(k, i)$, where k is the given value, and i is the shift.
For every same entry the i variable is increased by 1.
Linear hashing: if the index is taken i is added to index
Quadratic hashing: if the index is taken i^2 is added to index
Double hashing: If the index is taken $i \cdot h(k)$ is added.
The reason to use methods which space out index more is to prevent clustering, which can worsen search times.

4 Datastructures

4.1 Priority queues

Priority queues are based on the heap structure, which is described in Heap sort.

Here the queue objects are given a key from which the value iindicates the priority of the object.
It can here selected if the wanted heap is a max heap or min heap, but most often a max heap.
For the queue to work it uses the following functions:

- Max-Heapify(A,i,n) - makes the array in max heap order with the length n and looks at node i
- Heap-Extract-Max(A,n) - returns the highest priority object and remove from heap
- Heap-Increase-Key(A,i,key) - modifies the key i to a higher value key and place it correctly in heap
- Max-Heap-Insert(A,key) - Inserts key into the heap

4.1.1 Pseudo code

Max-Heapify code

1: Heap-Extract-Max(A, n)
2: **if** $n < 1$ **then**
3: error 'Heap underflow'
4: **end if**
5: $max = A[0]$
6: $A[0] = A[n-1]$
7: $Max - Heapify(A, 1, n-1)$
8: return max

.

1: Heap-Increase-Key(A, i, key)
2: **if** $key < key[i]$ **then**
3: error 'new key is smaller than current'
4: **end if**
5: **while** $i > 0$ AND $A[Parent(i)] < A[i]$ **do**
6: $temp = A[i]$;
7: $A[i] = A[i/2]]$;
8: $A[i/2] = temp$;
9: $i = i/2$
10: **end while**

.

1: Max-Heap-Insert(A,key,n)
2: $A.insert(n, -\infty)$
3: $Heap - Increase - Key(A, n, key)$

4.2 Binary search tree

A binary search tree is a tree which follows inorder.
Inorder states that for a node the left child is smaller and the right child is larger.
By this to get the ordered list a recursive call can be:

1: list-tree(x)
2: list-tree(x.left)
3: print(x)
4: list-tree(x.right)

This will go through the tree in order, this pseudo code will here print every key, and it should be noded the code does not handle NIL objects.
This operation takes $O(n)$ run time and all other oeparations as mention below will run in $O(height)$. The balanced binary tree will therefore have a height of $\log_2(n+1) \leq$ height and making the run time $O(\log n)$.

4.2.1 Searching a binary tree

To search the tree, a comparison is done at each node from which the right child is choosen:

1: Tree-Search(x,k)
2: **if** $x == NIL$ OR $k == key[x]$ **then**
3: return x
4: **end if**
5: **if** $k < x.key$ **then**
6: return $Tree - Search(x.left, k)$
7: **else**
8: return $Tree.search(x.right, k)$
9: **end if**

Find the lowest key, .

1: Tree-Minimum(x)
2: **while** $x.left! = NIL$ **do**
3: $x = x.left$

4: **end while**
5: return x

To find the smallest key bigger than key x. Works by if right side is not NIL it will be the minumum on the right side otherwise it moves up to the parent until the given node is the left child to the parent.

1: Tree-Successor(x)
2: **if** $x.right != NIL$ **then**
3: return Tree-Minimum($x.right$)
4: **end if**
5: $y = x.parent$
6: **while** $y != NIL$ and $x == y.right$ **do**
7: x= y
8: y = y.parent
9: **end while**
10: return y

4.2.2 Inserting in the tree

Tree-Insert(T,z)

1: $y = NIL$
2: $x = T.root$
3: **while** x!=NIL **do**
4: $y = x$
5: **if** $z.key < x.key$ **then**
6: $x = x.left$
7: **else**
8: $x = x.right$
9: **end if**
10: **end while**
11: $z.p = y$
12: **if** $y == NIL$ **then**
13: $T.root = z$
14: **else if** $z.key < y.key$ **then**
15: $y.left = z$
16: **else**
17: $y.right = z$
18: **end if**

y is the current nodes parent and x is the current node.
Then in the while loop the node is moved down in the correct direction, until it hits NILL.
Then a check is done, if y is NILL the tree is empty and we are at root.
Otherwise the node is placed according to the correct direction to parent.

4.2.3 Deleting a node in the tree

When removing an element there is three case according to the number of children equal to NIL:

- 2 - The element is replaced by NIL
- 1 - The parent reference is changed to the child
- 0 - The succesor to the remove node is inserted instead.

Tree-Delete(T,z)

```
1:  if z.left == NIL then
2:      Transplant(T, z, z.right)
3:  else if z.right == NIL then
4:      Transplant(T, z, z.left)
5:  else
6:      y = Tree − Min(z.right)
7:      if y.p! = z then
8:          Transplant(T, y, y.right)
9:          y.right = z.right
10:         y.right.p = y
11:     end if
12:     Transplant(T, z, y)
13:     y.left = z.left
14:     y.left.p = y
15: end if
```

The delete method uses a $Transplant$ function which arguments are:
Tree, node1, node2.
Which then switches node1 and node2.
The first if and else if covers the first case and second case with 2 or 1 child equal to NILL
In this case the other child is swapped with the node.

The else covers the last case of 0 NILL children.
First the succesor is found by the minimum node of z's right child.
The the if at line 7, checks for if the MIN node is not z's right child.
If not then the min node y is rotated up as the root of z's right branch, with its parent as right child and the parent left child being ys right child.
Then y substitutes z and takes z left child aswell.

4.3 Balanced binary tree

The idea behind a balanced binary tree, is a node has another bit representing a color red or black.
The rule is then:

- The root has to be black
- A red notes child can not be red

- Leafs has to be black

The rules ensures that the roots childrens height will atmost be different by a factor by 2.
This is due to one branch being all black and the other being red and black all the way.
The implementation of a balanced tree will not be slower due to all the manipulations still running at constant time $O(1)$.

4.3.1 Rotation of node

Rotations are a needed operation on a node in order to insert and remove nodes, and still uphold balance.

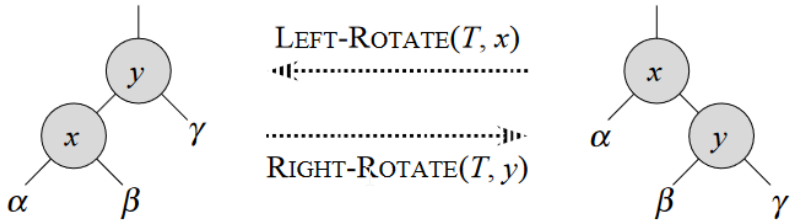

Figure 4: Illustration of rotation

```
Left-Rotate(T,x)
 1: y = x.right
 2: x.right = y.left
 3: if y.left != T.nil then
 4:     y.left.p =x
 5: end if
 6: y.p = x.p
 7: if x.p == T.nil then
 8:     T.root =y
 9: else if x ==x.p.left then
10:     x.p.left = y
11: else
12:     x.p.right = y
13: end if
14: y.left = x
15: x.p = y
```

First y is established and β is moved at line 2.
The if at line 3 changes the y left parent reference to x if not nil
The if at line 7 set y as root if x is root
The if at 9 and 11 sets x parents reference to x to y according to direction.
It can also be seen the rotation will keep inorder due to:

$$\alpha \leq x \leq \beta \leq y \leq \gamma$$

Which is followed both before and after rotation.

4.3.2 Insertion in tree

First the new element is inserted per usual inserting in a binary search tree as a red node. After this rebalancing may be needed, from which four sceanrios are possible:

1. z = root
2. z.uncle = red
3. z.uncle = black (trangle)
4. z.uncle = black (line)

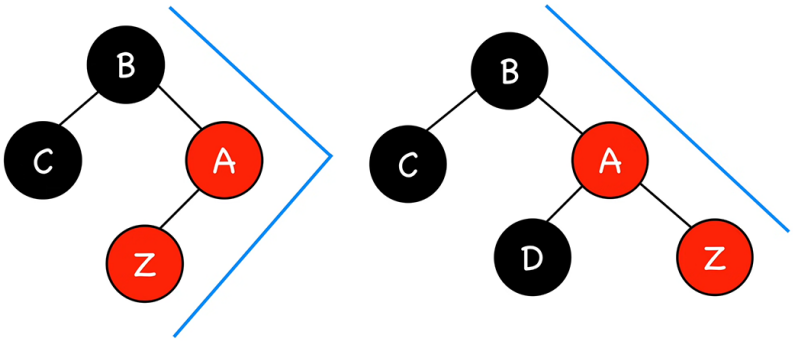

Figure 5: A triange shape on left and line shape on right.

1.
The nodes color should be switched to black
2.
Parent, uncle and grandparent change color
3.
Rotate z.parent in opposite direction of z
4.
Recolor parent and grandparent an rotate z.granparent in oppsite direction of z
RB-Insert-Fix(T,z)

```
1: while z.p.color == red do
2:     if z.p == z.p.p.left then
3:         y = z.p.p.right
4:         if y.color == red then
5:             z.p.color = BLACK
6:             y.color = BLACK
```

7: $z.p.p.color = RED$
8: $z = z.p.p$
9: **else**
10: **if** $z == z.p.right$ **then**
11: $z = z.p$
12: $Left - Rotate(T, z)$
13: **end if**
14: $z.p.color = BLACK$
15: $z.p.p.color = RED$
16: $Right - Rotate(T, z.p.p)$
17: **end if**
18: **else**
19: line 3 - 15 but with left instead of right
20: **end if**
21: **end while**
22: $T.root.color = BLACK$

The while loop runs until, the parent node is not red.
The first check is if the current branch is to the grandparents left.
Then the uncle y is defined.
Then the uncles color is checked. If red then color changes according to case 2.
If black, then if the triangle shape in case 3 is present the rotation is done.
From which case 4 will accour and recolor and rotation is done.

4.3.3 deletetion in tree

First a deletion is done, as normal. The only difference is a fixup function is called if the removed node is black or the successor node was black.
This due to a black therefore being removed or recolored and therefore the black length is not equal on both sides of root.
The fixup function has 4 cases which has to be fixed.

1. red sibling
2. black sibling with 2 black children
3. black sibling and nearest child is red and furthest is black
4. black sibling and the furthest child is red

The special child is x and its sibling is w
1. the parent and w changes color and left rotation on parent
2. w changes color to red, x pointer is parent and parent is red

3. w and its red child γ changes color and right rotation is performed on γ
4. w changes color to x's parent, which changes color to black aswell w right child. Left rotation on x's parent

RB-Delete-Fix(T,x)

```
1: while x! = T.root and x.color == BLACK do
2:     if x == x.p.left then
3:         w = x.p.right
4:         if w.color == RED then
5:             w.color = BLACK
6:             x.p.color = RED
7:             Left − Rotate(T, x.p)
8:             w = x.p.right
9:         end if
10:        if w.left.color == BLACK and w.right.color == BLACK
   then
11:            w.color = RED
12:            x = x.p
13:        else
14:            if w.right.color == BLACK then
15:                w.left.color = BLACK
16:                w.color = RED
17:                Right − Rotate(T, w)
18:                w = x.p.right
19:            end if
20:            w.color = x.p.color
21:            x.p.color = BLACK
22:            w.right.color = BLACK
23:            Left − Rotate(T, x.p)
24:            x = T.root
25:        end if
26:    else
27:        line 3 - 25 with right and left exhanged
28:    end if
29: end while
30: x.color = BLACK
```

The code consist of a while loop which only terminates when x is root or x is black.
This is due to if x is the root the color can be changed to black and it will have the same black length on both branches.
The while loop states by checking if the current working branch is to the

parents left.
Then the first if covers case 1, and the sibling pointer moves up to x's uncle.
The second if covers case 2, notice here it is guranteed due to the first case that the sibling is black.
The else covers case 3 and 4, where 3 is in the inner if.
It can here be seen that 1 can lead up to all cases, 2 does not lead up to 3 or 4 and 3 always leads up to 4.

4.4 Extra data in trees

When working with trees, extra information can be stored, such as amount of children in a node.
Saving the amount of children will help deduct the position of the node in a tree walk.
This kind of tree is called an order-statistic tree.
This can be done as seen here, where select returns the node at index i in the walk, and rank returns the nodes index in the walk.

OS-Select(x,i)

1: r = x.left.children+1
2:
3: **if** $i == r$ **then**
4: return x
5: **else if** $i < r$ **then**
6: return $OS - Select(x.left, i)$
7: **else**
8: return $OS-Select(x.right, i-r)$
9: **end if**

OS-Rank(T,x)

1: $r = x.left.children + 1$
2: $y = x$
3: **while** $y \neq T.root$ **do**
4: **if** $y == y.parent.right$ **then**
5: $r = r + y.parent.left.children + 1$
6: **end if**
7: $y = y.parent$
8: **end while**
9: return r

When updating the tree, it is just required to recursively from the button update the affected parents.