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 $\sum_{j=1}^{n} j = \frac{(n-1)n}{2} \leq \frac{n^2-n}{2} = n^2$
The run time of the same elements will only be $O(n)$

```
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.

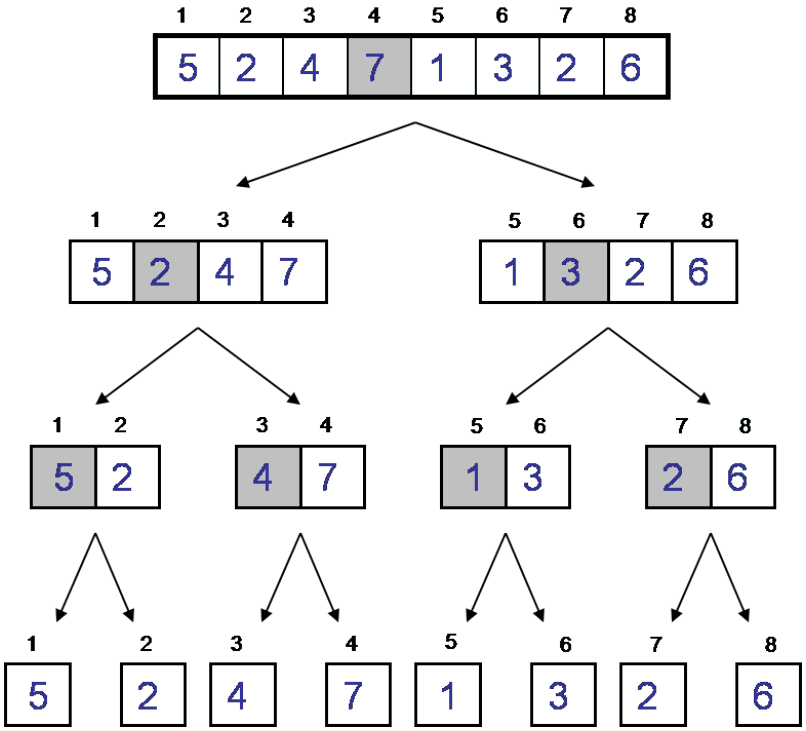

Figure 1: Merge sort list split up illustration from j2eereference.com

The main idea, is choosing an element in the array, then sorting everything 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

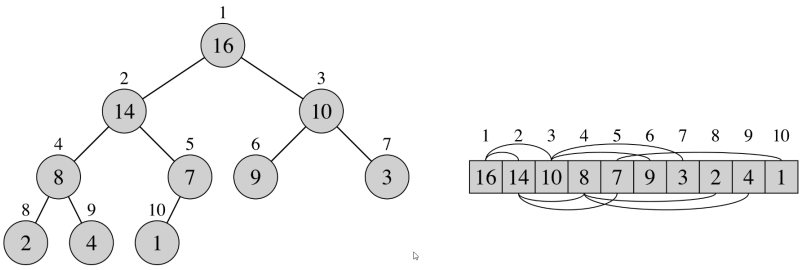

Figure 2: The heap illustration and the heap array

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 $\Omega- \geq$
And to get special log calc: $\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$ where k is just a constant such as 2.
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.
In case of $T(n) = aT(kn/b)$ where k is constant the b value is actually k/b.

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, recalculate 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

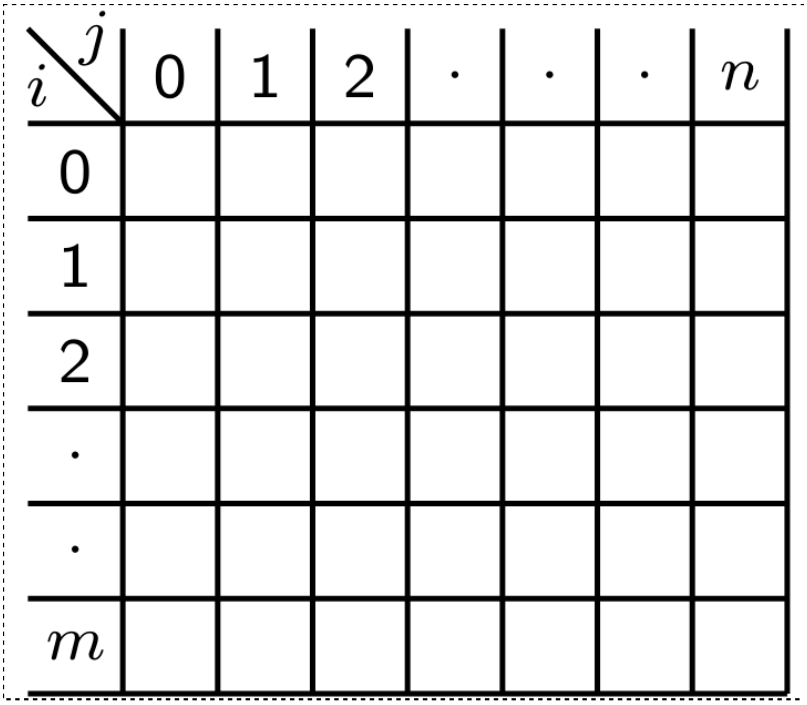

Figure 3: A 2D image of finding the longest subsequence

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.

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/ pobing: 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 from $h(x)$ is taken then $h'(x)$ is added until an open space is found.
The reason to use methods which space out index more is to prevent clustering, which can worsen search times.

3.5 Greedy algorithm

A greedy algorithm is an algorithm which takes the first possible solution. This will always find a local optimum, which may not be global optimal.
The algorithm is based upon taking the problem and taking the best solution

at the moment.
To use a greedy algorithm it is important the find and argument for an invariant which holds true such the solution it will find is an optimum.

An example to this type of problem is a packing example. The packages have each a weight and a value.
A greedy algorithm will here each package and find it value pr. weight, it then fills up with the most value pr. weight, if there is not enough space it goes to the next package.

3.6 Bit compression using Huffmans algorithm

Huffmans algorithm is a compression algorithm based on shortening bits using frequency.
An example is word shortening, in ASCII a charachter is represented with 8 bits.
But by finding the frequency of each character, the most used characters, can have a shorter bit code.
In order to assign each character a bit code, a tree is made.
Each charchter and its frequency is made as a leaf. Then the two lowest frequency nodes are combined such the two nodes are leafs in a tree where the parent is the combined frequency and the parent is now in upon all nodes. This is repeated until a true binary tree is made. Then each character is then assigned a bit code by going to the right in the tree will equal to 1 and left equal to 0.

This is greedy algorithm which finds the optimum, which can be argued by at first the tree is optimum of two characters.
Then for a the two scenarios where one tree is split such it has a depth of 1+ another tree, it can be seen that, due to the frequency and the single bit in difference, the difference will be constant, therefore neglible. Such that the found tree will always be the optimal tree.

The run time will then be $O(n \log n)$ using a heap algorithm and finding frequency is just $O(n)$, so neglible.

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 inorder will take n in both balanced and unbalanced.
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
- All path from a node to leafs must contain the same amount of black nodes

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.

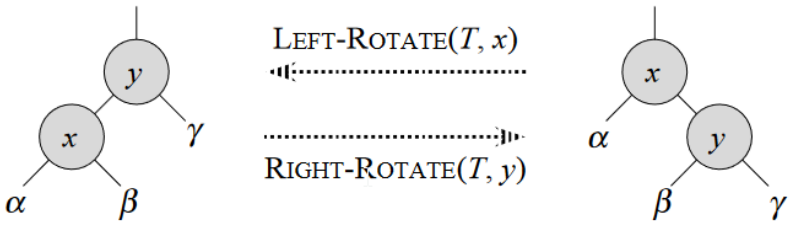

Figure 4: Illustration of rotation

```
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)

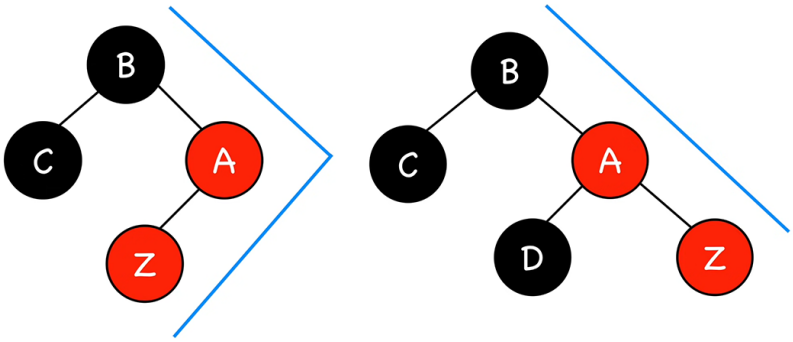

Figure 5: A triange shape on left and line shape on right.

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 ≠ 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.

4.5 Disjoint sets

Disjoint sets being small partitions of a set.
There are different ways of representing this in a datastructure.
The core of the datastructures are:

- Create a new set
- Join two sets
- Find set

4.5.1 Linked list representation

A linked list may be used for the dijoint set. Find set will be constant time and return header of linked list.
Make set also be constant creating a new linked list.
Union must be n time due to one lists reference the header must be changed and other list last node must be linked furhter.
Weighted-union heuristic is a union where the lower lsit is added to the larger list.
Path compression refers to all object in a branch is moved up to be only a depth of 1.
Thus for a making a set of n objects, the run time will be $O(m+n\log n)$, this is due to$\log n$ comes from joining, but since it is done with two sets of same size, the total size will double. Therefore it is only required to perform $\log n$ joins, which is done n times. The m is the amount of find set operations, which is each done in constant time and therefore the total is m time.

4.5.2 Tree representation

Another alternative is a tree, where a object in node is representative and root. Find will therefore travel up the tree until found root, make will make its own root, union takes the tree with biggest height as root and make the joining child of the root.
The make set of n and its union and find set will therefore be almost linear with a run time of $O(n+m)$ with m being a slow growing function.
Union by rank refers to making the union tree the optimal height, such when two trees are unioned, the larger tree becomes root such the height is the same, if the same height one is choosen and the height is added by 1.
Path compression is done on find operations. When root is searched for every nodes in the path to find root parent will be set to root, in constant time. Such the height of the tree wil lbe 1.

5 Graph

Naming conventions:

- Oriented graph - directed graph
- Unoriented graph - non directed

- Weighted graph - edges has a value
- V the set of all vertices
- E the set of all edges
- $n = |V|$,$m = |E|$
- $0 \leq m \leq n^2$ for oriented and $0 \leq m \leq n^2/2$ for unoriented graph
- Minimum spanning tree - MST A tree with lowest optimal path to all vertices
- DAG - Directed acyclic graph - A graph which is directed with no cycles

5.1 Representation

Graph may be setup as a graph.
List may also be used for each node containing connected vertices.
A matrix can also be used with size n^2 if n_1 and n_3 are connected a 1 is at n_1, n_3 and also n_3, n_1 .
Therefore in theory the actual size is $n^2/2$ due to it being reflective in an unoriented graph and oriented needs the n^2 due to it not being reflective.
Wheras the list version need $O(n + m)$ space

5.2 Graph traversal

To create a traversal a color coding is used on the vertices.

- white - no edges are used
- grey - some edges are used
- black - all edges are used

At the basic the traversal is done by:

```
1: GenericGraphTraversal(s)
2: s.color = grey
3: while There exists grey vertices do
4:     v = grey vertice
5:     if All edges in v are used then
6:         v.color = black
7:     else
8:         Choose an unused edge in v, u
```

```
 9:          if u.color == white then
10:              u.color = grey
11:          end if
12:      end if
13: end while
```

Once the algorithm is done all black nodes is connected to S and all white nodes are not.
For finding edges there are different algorithms such as:

- Breadth-First-Search
- Depth-First-Search
- Priority-Search

In some cases it may be needed to save for each which node discovered it. With this info a tree may be made from each node or leaf to root will exist the opposite path in the graph.

5.2.1 Breadth first traversal

This algorithm, start at a vertice, uses all possible edges and then goes to the next vertice.
This algorithm is therefore presented in a queue format.
The idea is all vertices are white and have a distance of infinity.
Then at a vertice the distance is 0, then all vertices directly connected are greyed and gets a distance +1.
Also the added by variable is added.
Then each vertice is added to the queue and everything is ran until the queue is empty.
This will result in every node has the shortest distance, due to essencially every node being taken in order of distance to root and therefore once a node is reached it is the shortest distance.

5.2.2 Depth first traversal

This method is a recursive method, and therefore is refered to using the stack.
Run time: $O(|V| + |E|)$
The method works by every node is white, and the one choosen is made grey.
A recursive function is then called on a connected node, which makes it grey

and call the recursive function on all connected nodes.
By this every connected node will be found recursively. This is seen in the stack by when a node is called it is added to the stack and becomes grey, once every connected node is found and popped of the stack it returns to the grey node and it becomes black and pops off.
Instead of measuring distance a time variable is made, this time variable is added once a node is discovered and once the node turns black.
At both the start and the end of the node the time is saved, such given two nodes (v,u) there relation can be described through their time:

- v and u tims are disjoint and they are note related
- v's time interval container u times interval and v is some form of parent to u
- the last scenario but flipped around

The time interval is often written as s/e where s is the start and e is end time
When doing the traversal there are 4 different types of edges which may be classified:

- Tree edge - The most common edge from parent to undiscovered child
- Back edge - Edge from child to an anchestor
- Forward edge - Edge from anchestor to some form of existing child
- Cross edge - The rest of the edges, mainly edges between branches

In non oriented graph only tree and back edges exists.

Sorting may also be done, if there does not exist any back edges.
No back edges will result in no cycles, and when each vertic is done it can be added to a list.
This will result in every vertice will only point to the right and be in topological order.

5.3 Strongly connected compoenents

To find strongly connected components Kosarajus algorithm can be used.
The algorithm start be doing a DFS on the graph. Once a node is turned black it is added to a second stack. The order should be decreasing in $u.f$ for each vertex u

Then the graphs directions are reversed.
Again DSF is performed, and starts at the end of the second stack. Once the selected stack element is popped off all the traversed nodes is strongly connected.
All other traversed notes in the strongly connected may be also popped of the second stack.
This is done until the second stack is empty.
When reversing the graph all connected components will stay the same due to the SCC's be definition are cycles and can get all around. Whereas the connecting edges between SCC's are cross edges, and by reversing them it is seen that before connected components no longer are connected therefore splitting the connected components.
The reasoning for the reverse topological order, is due to in the topological order, SCC's will be found last, due to them not being connected to the first choosen node.
Therefore be reversing it the strongly conncected components are found earlier and fewer nodes are checked.
This result in the running time being $O(|n| + |e|)$

5.4 MST

Navn	Prim	BFS	Kruskal
Directed	UD	Both	UD
Only positive	No	No	No
Run time	$O(\|E\| \log \|V\|)$	$O(\|E\| + \|V\|)$	$O(\|E\| \log \|V\|)$

Navn	DFS	Dijkstras	Bellman ford
Directed	Both	D	Both
Only positive	No	Yes	No
Run time	$\theta(\|E\| + \|V\|)$	$\theta(\|E\| + \|V\| \log \|V\|)$	$\theta(\|E\|\|V\|)$

A minimal spanning tree (MST) is a tree which visits every node in a weighted graph, at minimal cose.
This will form a tree and therefore can not contain any cycles.

5.4.1 Prim Jarnik

This is a greedy algorithm, which work with a queue.
A node is then choosen from which all edges is added to the queue.

The minimum is then choosen and used in the tree, which all edges from to not used nodes are inserted to the queue.
The last step is then repeated until the queue is empty, and when meeting a dead edge it is discarded.
This will result in a MST if the graph is all connected, due to the invariant being always choosing the best edge until all edges are evaluated, therefore all edges will be checked.

5.4.2 Kruskal

Kruskal works by, taking all edges, sorting them in order weight increasing, and then adding all edges which, connects two otherwise unconnected edges. This can be done by sets in the following algorithm:

1: A = $\emptyset$
2: **for** $v \in G.V$ **do**
3: $\quad Make - Set(v)$
4: **end for**
5: Sort sets in weight
6: **for** (u, v) from sorted list **do**
7: $\quad$ **if** $Find - Set(u) \neq Find - Set(v)$ **then**
8: $\quad\quad A = A \cup \{(u, v)\}$
9: $\quad\quad Union(u, v)$
10: $\quad$ **end if**
11: **end for**
12: return A

This will work by same argument as Prim Jarnik, but this will gurantee a better MST, in case of a bad start for Prim Jarnik.

5.4.3 Relax

Relax is a method which, simply works by a handler, which goes through every edge and sends it to the relax method.
This method simply handles a new edge and compare to an already known lightest weight. This weight the lightest is saved.

5.4.4 Dijkstras

This method is a more sophisticated version of relax. Instead of just trying every edge, a start edge is selected. Then every edge from the start node is added to a minimum queue. Then a loop is done until the queue is empty.
Run time: $O(|V| \log |V| + |E|)$

but can be up to $\omega(|E| \log |V|)$
The minumum edge in weight is choosen and the receiving nodes edges is added to the queue and called relax upon.
Every used node is saved, such if an edge from the queue is choosen towards an already checked node, the edge is dumped.

5.4.5 Bellman ford

Bellman ford is a tool to check if a graph has a negative loop such a MST is not existing.
The tool simply works by using relax to get the MST, and then for every edge tries to see if adding it will result in a lower weight. Therefore if that exist a negative loop in the graph it will return true.