Datastrukturer Och Algoritmer - En Sammanfattning

Datastrukturer och algoritmer (Mittuniversitetet)

Datastrukturer och algoritmer

Innehållsförteckning

Akronymer

- **Patologisk** Ett exempel som är korrekt men saknar egenskaper man vanligtvis tar för givna.
- **Invariant** En invariant är en regel/egenskap som inte kan förändras.
- **Asymptotisk tillväxttakt** Kort sagt; vilken komplexitet en algoritm har. Om vi vill sortera efter asymptotisk tillväxttakt så sorterar vi efter vilken komplexitet den har. Det här gör vi för att asymptot betyder att ”inte mötas” och om vi representerar olika komplexiteter som räta linjer så ser vi även att avståndet mellan dem ökar ju längre den arbetar.

Träd

En datastruktur som är formad som ett träd. Trädet består av noder som oftast innehåller någon form av information, exempelvis vilka närliggande noder som finns samt längden emellan, eller någon annan form av data. Trädstrukturen används för att kunna färdas igenom den på ett effektivt sätt, beroende på vilken typ av träd det är.

- **Vertices** Ett annat ord för noder.
- **Edges** En kant, som definieras av två noder
- **Weight** Avstånd/kostnad

Träd är oftast **riktade** (directed) nedåt, vilket innebär att sökning och operation endast kan ske åt ett håll.

Graf

Ett träd som innehåller loopar.

- **Directed Graph (digraph)** Riktad. En graf där varje kant (varje förhållande mellan två noder) antingen bara kan gå åt ett håll eller båda.
- **Undirected Graph** Oriktad. Man kan färdas åt vilket håll som helst i grafen.
- **Weighted Graph**

Adjancency Matrix

Struktur

	1	**2**	**3**	**4**	**5**
1	0	1	1	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	0	0	1
5	0	0	0	0	0

Ett sätt att med hjälp av heltal kunna representera en graf, direkted eller undirected. Detta skrivs som en $n * n$ matris, där varje column/rad motsvarar antalet noder i grafen. Nodernas förhållanden (d.v.s. hur grafens utseende) definieras igenom vilka värden som $\neq 0$. Om vi i exemplet till höger letar upp nod 2 i översta raden och vandrar ner till 3, så ser vi att deras förhållande är 0. Om vi däremot letar upp nod 4 i översta raden och vandrar ner till kolumn 3 (nod 3) så ser vi att värdet är 1, vilket innebär att det finns en länk som går från 4 -> 3, men inte nödvändigtvis från 3 -> 4. Det kontrollerar vi genom att vandra till kolumn 4 (nod 4) och sedan till värdet 3 i rad 4. Eftersom det är en nolla där så vet vi att inget förhållande existerar åt det hållet. Med andra ord, detta är i digraph.

Notera även att om vi byter ut ettorna mot andra siffror så kan vi representera förhållandets vikt, eller *avstånd* på så vis.

Komplexitetsteori

Ordo – Big O notation

Ett sätt att mäta effektiviteten hos en algoritm. $O(n)$ är mindre komplex än $O(n^2)$ och därför snabbare. ”n” är storleken på problemet d.v.s. hur snabbt problemet i algoritmen växer. Om vi exempelvis söker linjärt i en lista av heltal efter ett specifikt heltal så kan vi kalla komplexiteten för $O(n)$ eftersom operationen repeteras n antal gånger innan rätt heltal hittats. Om vi däremot påstår att varje heltal innehåller en egen lista med information, låt oss säga en lista med färger. Nu kan vi säga att vi vill hitta färgkombinationen röd-23 (vi kontrollerar färgen först så att varje heltal söks igenom) och därför kan vi kalla algoritmen för $O(n^2)$.

Big-O representerar en ”upper-bound/övre gräns”. Det innebär att det inte är fel att säga att en algoritm som är O(n) också är O(n^2) eftersom n^2 klart är en högre gräns än O(n).

Tillväxttakt

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Exempel på asymptotiska tillväxttakter. Detta ger oss en bild hur dem skulle förhålla sig till varandra i en graf.

Typer

Linjär Komplexitet $O(n)$

Exempel: Linjär sökning.

Binär komplexitet $O(\log n)$

Exempel: Binärsökning

Exponent komplexitet $O(n^2)$

Exempel: Selection sort

Ännu sämre komplexitet $O(n^3)$

Exempel: Kanske att vi lägg på en till operation för varje operation i selection sort. Dåligt implementerade algoritmer med $O(n^2)$ får ofta denna komplexitet i slutändan. Även ännu sämre implementationer av algoritmer med lägre komplexitet.

Kombinerad komplexitet $O(n + m)$

Exempel: När två olika algoritmer med linjär komplexitet är inblandade.

Bättre komplexitet baserad på binär $O(n\ log\ n)$

Exempel: Quicksort, mergesort och heap sort.

Konstant tid $O(1)$

Exempel: Hashtabell med $bucketsize\ ==\ storlek\ på\ tabellen.$

Faktoriserande tid $O(n!)$ d.v.s. $n * (n-1) * (n-2) * \ldots * 1.$

Exempel: Att läsa The Traveling Salesman Problem via Brute Force Search.

Cases

Worst case Det värsta typen av input en algoritm kan ha att göra med, som för algoritmen att ta maximal tid och/eller minne att genomföra.

Binärsökning

Komplexitet $O(log\ n)$. En sökningsmetod för sorterade listor, ej osorterade.

Metod

En binärsökning implementeras på olika sätt beroende på den samling information som söks igenom. Om samlingen är ett binärt sökträd så söker man helt enkelt binärt för varje element i trädet om det elementet som söks finns åt höger eller vänster i trädet. Om vi däremot söker binärt i en lista så behöver vi vara lite smartare än så.

Exempel

Hitta 6 i listan $l = 14938546$.

Steg 1 Sortera listan, om den inte redan är sorterad. En sorterad lista krävs för att binärsökning ska fungera.

Steg 2 Partitionera; hitta det mittersta elementet. Jämför detta element med det sista elementet.

Avgör i vilken halva som det sökta elementet finns.

123456789 → 5<6

Steg 3 Upprepa **steg 2** till dess att det mittersta elementet == elementet som söks.

6789 → 8>6

67 → 7>6

6 → Hittad!

Linjärsökning

Engelska: ”sequental search”. Komplexitet $\boldsymbol{O(n)}$. En jättesimpel sökningsmetod som söker igenom en hel lista, element för element, där vi kontrollerar varje element om det är det som söks. Likvärdigt effektiv på sorterade och icke-sorterade listor (om vi inte vet någonting om dem sedan innan).

Metod

Steg 1 Kontrollera om det första elementet är == elementet som söks.

Steg 2 Gå vidare till nästa element i listan och upprepa steg 1 till rätt element påträffas. Bryt när det rätta elementet påträffas, eller inte finns med.

Huffmankodning

Vanligt använd inom filkomprimering med format såsom *.zip*, *JPEG* och *MP3*. Fördelen med Huffmankodning är att det minimerar utrymmet genom att göra vanligt förekommande karaktärer billigare att allokera och resultatet sparar diskutrymme.

http://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/

Huffmanträd

1. Skriv ner varje bokstavs frekvens från texten
2. Lägg till dessa frekvenser i en minHeap (större värden mot roten).
3. Hämta två element ur heapen och kombinera dessa två rotnoder och skapa en föräldernod till dem som heter deras gemensamma frekvens. Lägg till den gemensamma frekvensen i heapen. Notera: Om kombineringen av lövnoderna inte är bredvid varandra så flyttar vi dem till den nod som är längst åt vänster.
4. Upprepa steg 3 så länge det finns värden i heapen.
5. Följ nu vägen vänster eller höger (1 eller 0) för att ta reda på vilken bitsträng så motsvarar vilken bokstav.

MinHeap exempel

Char	Freq
A	3
D	6
B	7
Osv	

2-3-4 träd

Även kallad B-tree. 2-3-4 träd är en självbalanserande datastruktur. Den vill alltså balansera sig själv utefter dem datanoder som ligger överst. Nya element stoppas in i rätt ”fack” genom att vandra.

Invarianter

- En 2-nod har ett dataelement och två barn (om intern (inte löv eller rot))
- En 3-nod har två dataelement och 3 barn (om intern)
- En 4-nod har tre dataelement och 4 barn (om intern)
- Allt som är till vänster om en nod är mindre, och allt som är till höger är större.

Metod

Det går helt enkelt ut på att leta sig ner igenom noderna efter värdet man är intresserad av med binära val (höger eller vänster, d.v.s. mer eller mindre...). Man utför operationen och försöker balansera under tiden utefter de invarianter som finns fördefinierade.

Bra källa: https://www.cs.usfca.edu/~galles/visualization/BTree.html

Mergesort

Komplexitetsgrad $O(n\ log\ n)$. En *divide-and-conquer* algoritm.

Metod (Top-down)

Denna metod går ut på att rekursivt dela upp en array i partitioner till dess att varje partition består av ett element. Härifrån så sorteras varje partition med sin granne utifrån hur den partitionerades i samma nivå. På så vis så jobbar mergesort Top Down i det antagande att varje lista som den mergar redan är sorterad.

Exempel

Uför mergesort (top down) på texten ”*examquestion*”.

Steg 1 **Divide**; Partitionera i två delar. Upprepa så länge $partition > 1$.

Examquestion	Nivå 1	Nivåerna representerar rekursioner.
examqu estion	Nivå 2	
exa mqu est ion	Nivå 3	
e xa **m** qu **e** st **i** on	Nivå 4	
x a q u s t o n	Nivå 5	

Steg 2 **Conquer**; i samma ordning som vi partitionerade delarna så sorterar vi dem genom att kombinera dem med sin granne.

Nivå 5: x jämför med a => ax

Samma sak för resterande på nivå 5.

Nivå 4: e jämförs med ax => aex (e < a ? Nej, L[0]=a. e < x? Ja, L[1]=e. L[2]=x)

Samma sak för resternde på nivå 4.

Nivå 3: osv…

Nivå 1: En färdig sorterad text.

Metod (Bottom-up)

Denna metod går ut på att partitionera en lista (array) i partitioner där varje element motsvarar en partition och sedan kombinera och sortera partitioner parvis. Detta upprepas till dess inga partitioner finns kvar. Här antar vi att listan redan är partitioner och sorterar partitionerna med olika storleksintervall.

Exempel

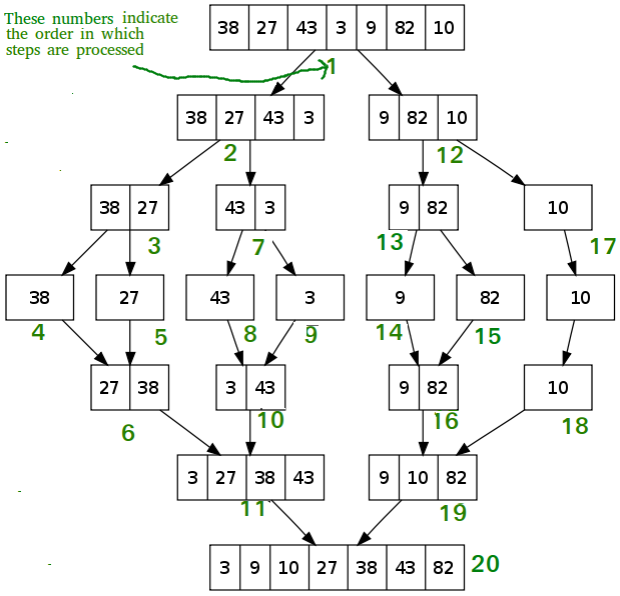
Utför mergesort (bottom up) på texten *”examquestion”*. *Operation:* $Mergesort(examquestion)$

E x a m q u e s t i o n

E x a m q u e s i t n o

A e m x e q s u i t n o

Osv…



Röd-svarta träd

Läs igenom labb 4 för en genomgående beskrivning.

Prioritetskö/Heap

En prioritetskö fungerar ungefär som en vanlig kö där varje element istället ges en egen prioritet. Implementering sker gärnas med hjälp av en maxHeap. En maxHeap är en heap, ett binärt sökträd, där dem största värdena är högst upp. Roten är alltid det element som har högst prioritet och det värde som ”poppar” ur kön.

Bra källa med bilder: http://pages.cs.wisc.edu/~vernon/cs367/notes/11.PRIORITY-Q.html

Heapsort

$O(n \log n)$ i komplexitet, worst case och best case. Heapsort kan även ses som en förbättrat selection sort eftersom även heapsort sorterar arrayen gradvis (en del är sorterad, den andra delen är osorterad). Fördelen med att använda heapsort ligger i att den har ett bra worst case performance, d.v.s. $O(n \log n)$.

Metod

Heapsort består av två huvudsteg; skapa en heap utifrån listan och hämta sedan värden ur heapen.

Notera att vi kan bygga trädet antingen med stigande (minHeap) eller fallande värden (maxHeap), båda fungerar lika bra i slutändan så länge vi vet vilket håll trädet går åt.

Bygg heapträdet, en Max Heap (insättning)

Steg 1 Lägg till första värdet n ur listan i trädet och kalla denna för rot.

Steg 2 Lägg till nästa värde n ur listan och lägg till den längst till höger i den lägsta nivån. Om alla noder redan har två barn, skapa en ny nivå till vänster om det barn som befinner sig allra längst till vänster.

Steg 3 Om $n < föräldernoden$ så byt plats på n och föräldern. Upprepa så länge $n < föräldernod$. Upprepa från steg 2 och för nästa värde n ur listan tills listan är tom.

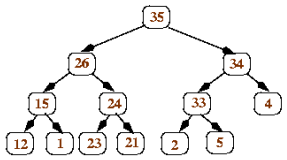
Ett balanserat max heapträd är nu skapat och arrayen som representerar elementen ser ut på ett visst sätt.

Ta bort noder rekursivt och lägg till i den färdiga listan

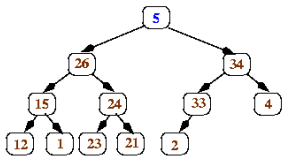
Steg 1 Vi byter plats på värdet som är längst ner till höger med värdet i roten. Vi tar bort den gamla roten och lägger till den i vår array.

Steg 2 Balansera Max Heapen. Flytta ner den nya (lilla) rotnoden genom att byta plats på var nods största barnnod och rotnoden.

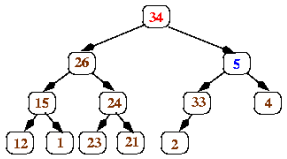
Steg 3 Gå tillbaka till steg 1 och upprepa så länge trädet har noder.



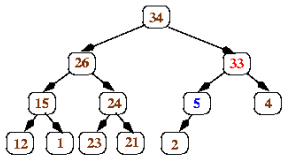
original heap



step 1: extract root value and replace with last leaf



step 2: swap with larger child



step 2: swap with larger child again
All done!

Implementering av prioritetskö

Det bästa sättet att implementera en prioritetskö är via **heap**. En heap är ett vänsterbalanserat träd där roten är störst och barnen är större värden än sina barn osv.

Invarianter

- Varje nods värde är större eller lika med värdena i nodens barn
- Trädets grenar är så lika långa som möjligt. Om det inte går så fylls den ifrån vänster nederifrån (vänsterbalanserat).

Hashtabell

En datastruktur sparar data tillsammans med en nyckel, där positionen i tabellen beräknas med en hashfunktion. Exempel, en telefonbok där vi vill hitta ett nummer utan att leta igenom hela listan. Worst case $O(n)$ och best case $O(1)$.

Dynamisk Array

Komplexitet vid insättning är $O(1)$ och omallokering $O(n)$, vilket gör omallokering en tidskrävande process. Omallokeringen sker däremot exponentiellt bitvis så att elementen ska få plats inom 2^x, vilket innebär att arrayen inte behöver omallokeras alltför ofta vid större minnesanvändning.

En dynamisk array lägger till element i slutet utav arrayen. Om arrayen är för liten när ett nytt element försöker läggas till så måste den omallokeras, vilket tar $O(n)$ lång tid.

En std::vector är ett exempel på en dynamisk array.

RLE-Komprimering (run-length)

RLE går ut på att korta ner text som består av sekventiellt förekommande karaktärer genom att ersätta dem med en siffra som talar om hur många det är. Därför används denna teknik när bitsträngar i källdatan är återkommande och repeterande, exempelvis bitmaps (bmp), d.v.s. när datan innehåller många ”runs” av återkommande källdatakaraktärer. Komprimeringsgrad (Compression ratio; exempelvis bildkvalité) här är högsta komprimeringsgraden ”hög”, när bra algoritmer används.

Exempel

Källdata	wwwppwwwwbbbc
Komprimerad data	3w2p4w3b1c

Quicksort

Quicksort är en Divide-and-Conquer algoritm.

Komplexitet

Worst case	$O(n^2)$	Händer när listan redan är sorterad kombinerat med en dålig pivot.
Average case/best case	$O(n\ log\ n)$	

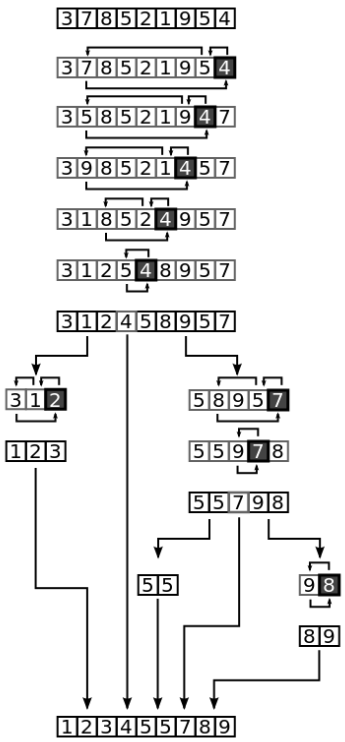
Median-of-three

En metod för att välja ett bättre pivot värde för att undvika $O(n^2)$. Istället för att välja mittenelementet som pivot så tar vi det första, mittersta och översta elementen och tar medianen av dem som pivot. För att få ut så mycket som möjligt av median-of-three så sorterar vi dem här tre elementen innan vi fortsätter.

Metod

Steg 1: Välj pivot.	Välj ett element som vi ska kalla pivot p, ur arrayen. (alternativt median-of-three)
Steg 2: Partitionering.	För varje värde i arrayen, om värdet är $< p$ placera det innan pivot i listan. Annars placera det till höger om p.
Steg 3: Rekursion.	Repetera från steg 1 för varje partition. Om partitionen < 2: avsluta.

Exempel



Polynom

Polynom:	Anledning:
5	En konstant.
$3x$	
$x-2$	
$-6x^2-(\frac{7}{9})x$	
$0{,}1xz-200y+0{,}5+3xyz$	
$512v^5+99w^5$	
$\frac{x}{2}$	Man får dividera med en konstant, men inte tvärtom.
$\sqrt{2}$	2 är en konstant (1,41...)

Ej Polynom:	Anledning:
$3xy^{-2}$	Exponenten måste vara $n \leq 0$.
$\frac{2}{(x+2)}$	(får inte dividera med en variabel)
$\frac{1}{x}$	
$\sqrt{x}$	Får inte ta roten ur en variabel.
2^N	Får inte var upphöjt i en variabel.
$x^{\frac{1}{2}}$	Får inte vara upphöjt i en fraktionsexponent.
$(5x+1)+(3x)$	Inte ett polynom på grund av **divisionen**. (parenteserna) Om det däremot hade gått att förenkla så att paranteserna försvann så hade det varit ett polynom.

Insertion Sort

Visualisering: http://visualgo.net/sorting

Komplexitet

Worst case $\boldsymbol{O(n^2)}$.

Best case $\boldsymbol{O(n)}$ och $\boldsymbol{O(1)}$ swappningar.

Fördelar

- Stabil
- Låg overhead (omkostnad), eftersom den är in-place (d.v.s. behöver inget extra utrymme än själva listan som genomsöks).
- Snabb ($O(n)$) när input är nästan sorterad.
- Simpel och effektiv vid sortering av mindre listor.

Selection Sort

Kontrollerar alla värden sekventiellt från höger till vänster i en lista mot resten av den okontrollerade delen av listan.

Komplexitet

Worst case och best case: $O(n^2)$

Fördelar

- Presterar bra när listan som genomsöks är liten.
- Låg overhead (omkostnad), eftersom den är in-place (d.v.s. behöver inget extra utrymme än själva listan som genomsöks).
- Effektiv när listan är liten och innehållet är i random order.

Metod

1. Börja på listans första värde, kalla denna för n.
2. Vandra åt höger i listan och markera det första värdet som m. Vandra nu vidare och för varje mindre element än m som hittas så är detta värde nu m. Kontrollera alla värden till listans slut.
3. Byt plats på m och n. n är nu [m+1]
4. Börja om från steg 2 så länge $n \neq L.size()$

Shellsort

En in-place jämförelsesorteringsmetod.

Bra källa: https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Komplexitet

Worst case $O(nlog_2^2 n)$

Best case $O(n\ log\ n)$

Metod

Algoritmen består av att sortera en lista med hjälp av insertion sort på ett interval av listans element enligt 2^n. För varje iteration så markeras 2^(n+1) element i listan enligt ett jämt interval baserat på listans storlek. Sedan körs insertionsort på dem markerade elementen ända till 2^n = listans storlek då en insertion sort körs på alla kvarvarande element.

Utför shellsort på texten ”examquestion”.

$Size\ =\ 12$ $n = 1$ $punkter\ =\ n * 2$ $(size)/punkter = 6$

Examquestion

Examquestion Byt plats på x och s. Insertion sort.

Esamquextion

Esamquextion Byt plats på m och i.

Esaiquextmon Byt plats.

Esaiouextmqn Byt plats.

Esaionextmqu Färdig med första iteration.

$n = 2, punkter\ =\ n * 2, (size)/punkter = 3$

Esaionextmqu $e\ <\ i = sant, i\ <\ x = sant, x < u = falskt.$

Byt plats på x och u och backtracka från u. (insertion sort)

Esaioneutmqx $u > i = sant$, därför stanna.

Esaioneutmqx $s < o = sant$

Byt plats på s och o.

Eoaisneutmqx $s < u = sant, u < q = falskt$

Byt plats på u och q, backtracka från q.

Eoaisneqtmux $s < q = falskt$

Byt plats på q och s, backtracka från q

Eoaiqnestmux $o < q = sant$, stanna.

Eoaiqnestmux $a < n = sant, n < t = sant, t < x = sant$

Färdig med andra iterationen.

$n = 3, punkter = n * 2, size/n = 2$

Eoaiqnestmux $e < a = falskt$

Byt plats på e och a.

aoeiqnestmux $e < q = sant, q < e = falskt$

Byt plats på q och e_2, backtracka från e.

aoeienqstmux $a < e = sant$, slut på backtrack.

Aoeienqstmux $q < t = sant, t < u = sant$

Aoeienqstmux $o < i = falskt$

Byt plats på o och i

Aieoenqstmux

Osv…

I slutändan så är alla element markerade och vanlig insertion sort körs. Den kommer i dem flesta fall få ett best case performance på O(n) eftersom listan redan är ganska väl sorterad.

Länkad lista

Insättning: $O(1)$ när vi har gett noden en specifik pekare.

Borttagning: $O(1)$, av samma anledning. Annars O(n) om vi behöver leta igenom listan.

Notera: Insättningen må vara $O(1)$, men att hitta själva platsen att att sätta in elementet/ta bort är en $O(n)$ operation.

Exempelklass i C++:

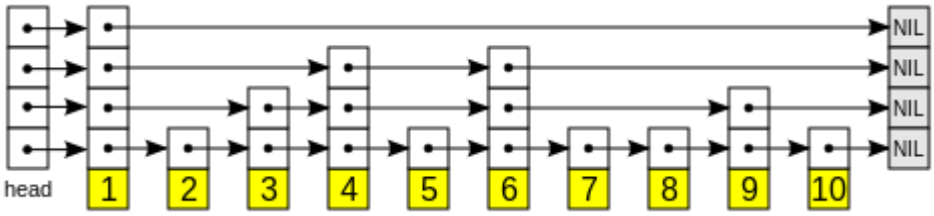
Breadth-first Search

Bra källa: https://www.cs.usfca.edu/~galles/visualization/BFS.html

1. Söker på djupet, våningsvis. Kontrollerar alla närliggande noder och färdas sedan till en av dem.
2. Ny nod. Kontrollerar närliggande noder. Hoppar upp till föräldern igen.
3. Kontrollerar om det fanns andra noder som inte var löv. I så fall, färdas till dem.

Skiplista

Komplexitet $log(n)$.



Skapande sker genom att en sorterad lista läggs i en tvådimmensionell array/länkad lista där varje höjdnivå läggs till genom ett ”coin flip”.

Sökning sker binärt från den högsta head-nivån och kontrollerar varje nod åt vänster så länge < n. Om inte noden hittats och < n så färdas vi ner en våning från den nod vi senast var på.