Gagnaskipan glósur

**3. kafli**

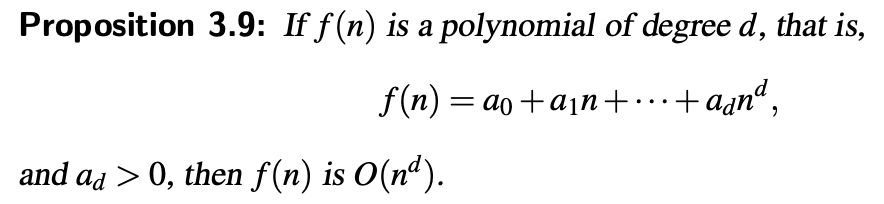
* Data structure er leið til að skipuleggja og sækja gögn
* Algóriðmi er skref fyrir skref uppskrift af því hvernig eitthvað er gert á ákveðnum tíma
* Skoðum running time á forritum ásamt minnismagni
* Taka tíma á forriti:
  + from time import time  
    start time = time( )  
    run algorithm  
    end time = time( )  
    elapsed = end time − start time
* Það getur verið tímamunur á milli keyrlsna þó að forritinu hafi ekkert verið breytt, timeit moduleinn gerir þetta aðeins betra.
* Til að meta niðurstöður er gott að vera með á x-ás input stærð og y-ás tímann.
* Gallar við þessa aðferð (nota tilraunir):
  + Erfitt að bera saman tvo algóriðma án þessa ð vera í sömu tölvu.
  + Bara hægt að gera tilraunir á takmörkuðum inputs, running tími fyrir input sem eru ekki í tilrauninni er ekki tekinn með, gætu verið mikilvæg input!
  + Það þarf að vera 100% búið að gera kóðann til að tékka á tímanum.
* Okkar markmið er að:
  + Bera saman efficiency tveggja algóriðma, skiptir ekki máli í hvaða tölvum
  + Það er gert með því að lýsa mjög vel algóriðmanum án þess að implementa
  + Tekur tillit til allra inputa
* Svo við notum ekki tilraunir
* Það sem við gerum:
  + Framkvæmum analysis á high level description á algoriðma
  + Skilgreinum grunnaðgerðir
    - Grunnaðgerð er með fastan execution tíma.
    - Í stað þess að meta tímann á öllum grunnaðgerðum teljum við hve margar eru framkvæmdar og notum þá tölu, t, sem mælikvarða á keyrslutímann
  + Til að meta stækkun á keyrslutíma er fall tengt við hvern algóriðma sem líkir eftir fjölda aðgerða sem eru framkvæmdar sem fall af input stærð, n. Fallið er f(n).
* Það fer mismikill tími í hvert input, erfitt að nota meðaltal af því gætum fengið input sem er bara með hæstu gildi.
* Í bókinni verður alltaf tekið worst case tímana (hæstu). Leiðir yfirleitt til betri algóriðma.
* Sjö föll sem eru notuð í bókinni til að greina algóriðma:
  + Constant function
    - F(n) = c
    - Öll input fá gildið c.
    - Notum oftast g(n) = 1 sem er sama og c\*g(n)
    - Þegar einhver aðgerð tekur alltaf sama tíma, sama hvert inntakið er
  + Logarithm function
    - F(n) = logb(n), b>1.
    - x = logb(n) þá og því aðeins af b^x = n
    - log(1) = 0
    - í bókinni er b-ið yfirleitt 2 þannig ef það stendur ekki þá er það 2
    - Hægt að finna minnsta int stærri en eða jafn logb(n) með því að deila n með b. Minnsta intið er þá hversu oft við náum því.
      * log3 27à is 3, because ((27/3)/3)/3 = 1. Likewise, àlog4 64à is 3, because ((64/4)/4)/4 = 1, and àlog2 12à is 4, because (((12/2)/2)/2)/2 = 0.75 ≤ 1.
    - T.d. ef ég er með lista og framkvæmi eina aðgerð og þá þarf ég bara helminginn af listanum, framkvæmi svo aðra og þá helminginn af því osfrv.
  + Linear function
    - F(n) = n
    - Gerist þegar við þurfum að gera eina basic aðgerð fyrir hvert af n elementum.
  + N-log-N function
    - F(n) = n\*log(n)
    - Stækkar hraðar en linear en hægar en quadratic
  + Quadratic function
    - f(n) = n^2
    - T.d. þegar nested loop og innri loopan framkvæmir línulegt magn af aðgerðum, ytri loopan er svo líka framkvæmt línulega oft.
    - Getur líka verið þegar nested loops og fyrsta gerir eina aðgerð, önnur tvær, þriðja þrjár osfrv
      * 1+2+3+···+(*n*−2)+(*n*−1)+*n* = *n*(*n*+1)/ 2
  + Cubic function and other polynomials
    - F(n) = n^3
    - Kemur sjaldnar til sögunnar en það sem fyrir hefur komið
    - Polynomial (margliða) fall er:
      * *f*(*n*) = *a*0 +*a*1*n*+*a*2*n*2 +*a*3*n*3 +···+*adnd*
    - Keyrslutími margliðna af lægra stigi er styttri en af hærra stigi
    - Summation:A close up of a logo

      Description automatically generated
    - Summurithátturinn gefur styttri leið í að útskýra föllin hér að ofan
  + Exponential Function
    - F(n) = b^n
    - Þar sem b er jákvæður fasti, kallast base
    - Oftast er b=2
    - Td ef það er loop sem framkvæmir fyrst eina aðgerð en tvöfaldar svo
* Geometric sums
  + Loopa þar sem í hver ítrun tekur inn margföldunarþátt lengri en fyrri ítrun
  + *an*+1 − 1/ *a*−1
  + For example, everyone working in computing should know that
  + 1+2+4+8+···+2*n*−1 = 2*n* −1,  
    for this is the largest integer that can be represented in binary notation using *n* bits.
* Samantekt af 7 föllunum!A screenshot of a cell phone

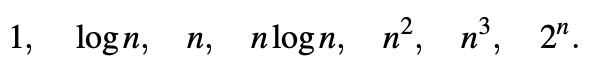
  Description automatically generated
* Algoriðmar með quadric eða cubic keyrslutíma eru óhentug og með exponential eru nánast ómöguleg nema fyrir mjög lítil input.
* A close up of a piece of paper

  Description automatically generated
* - floor function, stærsta int minna eða jafnt og x
* - ceiling function, minnsta int stærra eða jafnt og x

**Asymptotic analysis**

* Big oh notation - f(n) er O(g(n)) ef f(n)<=c\*g(n) fyrir n>= n0. Þar sem c er fasti stærri en 0 og n0 er jákvæð heil tala stærri en 1. Stærra en eða jafnt og annað fall.
* Big oh er notuð til að finna keyrslutíma
* Big oh leyfir okkur að hunsa fastaþætti og low order terms og fókusa á stærri hluta falla sem hafa áhrif á stækkun þess.
* 5*n*4 +3*n*3 +2*n*2 +4*n*+1 is *O*(*n*4) því 5*n*4+3*n*3+2*n*2+4*n*+1≤(5+3+2+4+1)*n*4 =*cn*4, for *c*=15,when *n*≥*n*0 =1.
* 
* Segja alltaf lægsta veldi sem við getum, ef það er O(n^2) þá er það líka O(n^3) en -segjum 2.
* Ef ég er með lúppu sem er línuleg og aðra nested sem er lógaríðmísk þá er hún O(n\*log(n))
* So, for example, we would say that an algorithm that runs in worst-case time 4*n*2 + *n* log *n* is a ***quadratic-time*** algorithm, since it runs in *O*(*n*2) time. Likewise, an algorithm running in time at most 5*n* + 20 log *n* + 4 would be called a ***linear-time*** algorithm.
* Big omega - minna en eða jafnt og annað fall
  + A screenshot of a cell phone

    Description automatically generated
  + Ath stærra og jafnt og merkið er öfugt við big oh
* Big Theta - stækka á sama hraðaA screenshot of a cell phone

  Description automatically generated
* Ef ég er með tvö forrit og annað er O(n) og hitt O(n^2) þá er keyrslutími O(n) minni fyrir öll n sem eru stærri en 1.
* Í stækkandi röð:
* 
* Það er hægt að leysa fleiri vandamál á stuttum tíma eftir því hve öflug tölvan er.
* Ef það er mjög stór fasti fyrir framan big oh þarf að skoða það sérstaklega. Ef t.d. það er verið að bera saman 10^100n og 10nlog(n) þá myndi maður nota big oh af nlog(n) í stað n!
* nlog(n) er metið efficient og n^2 fyrir lág n en 2^n verður nánast aldrei efficient.
* array er listi
* len(list) og list[i] er O(1)
* Prefix averages - Runa S með n tölur, við viljum fá runu A þannig að A[j] sé meðaltal elementa S[0},…,S[j] fyrir j=0,…,n-1. Þ.e.:
* A picture containing object

  Description automatically generated

**Simple justification techniques**

* Hægt að nota dæmi eða mótdæmi til að sanna staðhæfingu
* Contrapositive: To justify the statement “if p is true, then q is true,” we establish that “if q is not true, then p is not true” instead
* DeMorgan’s Law: This law helps us deal with negations, for it states that the negation of a statement of the form “p or q” is “not p and not q.” Likewise, it states that the negation of a statement of the form “p and q” is “not p or not q.”
* Contradiction: we establish that a statement q is true by first supposing that q is false and then showing that this assumption leads to a contradiction
* Líka hægt að nota þrepun!
* Loop invariants:
* To prove some statement L about a loop is correct, define L in terms of a series of smaller statements L0,L1,...,Lk, where:
* 1. The initial claim, L0, is true before the loop begins.  
  2. If Lj−1 is true before iteration j, then Lj will be true after iteration j.
* 3. The final statement, Lk, implies the desired statement L to be true.

**Glósur í tíma**

Hægt að vera með O(log(n)) tímaflækju til að finna stak í lista ef hann er sortaður, skippum helmingnum af listanum í hverju skrefi.

O(1) dæmi er t.d. að appenda lista því það tekur alltaf jafn langan tíma óháð fjölda staka

Ef tímaflækjan er margfeldi þá erum við að vinna með nested loopur, þ.e. n\*log(n) eða n^2 osfrv..

Ef það eru tveir forritsbútar þá er hægt að segja + á milli tímaflækjanna þeirra en þá er tímaflækjan í rauninni bara sú sem er lengri, t.d. ef annar búturinn er log(n) og hinn n þá er tímaflækjan n+log(n) en við segjum bara O(n) því það tekur lengri tíma. Tökum líka fasta út ef t.d. 2\*n þá er það O(n). Líka ef það eru tveir bútar af n þá er það samt bara O(n)

T.d. hér er heildartímaflækjan O(n)

n = 1000000

i = 1

# log(n)

counter = 0

while (i<n):

i \*= 2

counter += 1

print(counter)

# n

for j in range(n):

print(j)

Dæmi um log(n) - skippum helmingnum af stökunum í hverju skrefi:

n = 1000000

i = 1

counter = 0

while (i<n):

i \*= 2

counter += 1

print(counter)

Dæmi um n

for j in range(n):

print(j)

Dæmi um nlog(n):

# nlog(n)

# n þessi ytri

for j in range(n):

print(j)

i = 1

counter = 0

#log(n) er innri

while (i < n):

i \*= 2

counter += 1

Dæmi um n^2:

# n^2, ytri er n og innri er n

for bla in range(n):

print(bla)

for bla in range(n):

var = 1+1

Dæmi um n^3:

# n^3, ysta er n, miðjan er n og innsta er n

for bla in range(n):

print(bla)

for bla in range(n):

var = 1+1

for bla in range(n):

var2 = 2+2

Dæmi um 2^n - í hverju skrefi tvöfalda ég fjölda “skrefa”, kalla í fallið inní fallinu

# 2^n

def fib(n):

if n == 1:

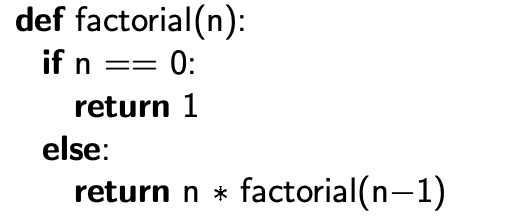
return 1

if n == 2:

return

return fib(n-1) + fib(n-2)

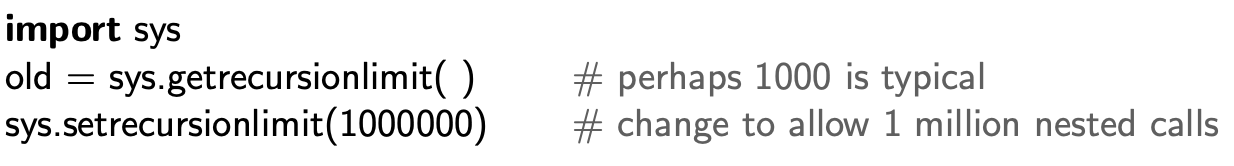
**4. kafli – recursion**

* Recursion = Fall kallar í sjálft sig einu sinni eða oftar eða a data structure relies upon smaller instances of the very same type of structure in its representation.
* Fjögur dæmi um recursion:
  + Factirial function (n!): Fall kallar í sjálft sig n sinnum
    - Dæmi:
    - 
  + English ruler: Reglustika með major tick length og svo nokkrar minor tick lengths. Þegar lengd bilsins minnkar um helming minnkar lengd ticks um 1.
    - In general, an interval with a central tick length *L* ≥ 1 is composed of:
    - • An interval with a central tick length *L* − 1
    - • A single tick of length *L*• An interval with a central tick length *L* − 1
  + Binary search: Notað til að finna target value inní runu af n elements. Ef runan er óröðuð er hægt að fara í gegnum öll element þar til targetið er fundið (eða finnst ekki ef það er farið í gegnum alla rununa) – þetta kallast sequential search, keyrir á O(n). Ef runan er sorted og „indexable“ er hægt að skipta henni upp í helming, athuga í hvor helmingnum hún er og skipta þeim helming aftur í helming osfrv – kallast binary search, keyrir á O(log(n)).
  + File system: Stýrikerfi kalla í möppur á recursive hátt. Efsta mappan er með fleiri möppur sem eru svo með fleiri osfrv. Stýrikerfin leyfa að vera með nested möppur svo lengi sem það er nóg minni en á endanum þarf bara að vera files, engar submöppur.
    - Immediate disk space er minni notað í hverri entry
    - Cumulative disk space r minni notað í þessu entry og öllum nested features.
    - Þetta cumulative disk space er hægt að reikna með recursive algorithm. Það er samasem immediate disk space plús summan af cumulative disk space af nested entries.
    - Os module hjá python:
      * os.path.getsize(path) – skilar immediate disk usage (í bytes) fyrir fileinn eða möppuna sem er identified með path strengnum
      * os.path.isdir(path) – returns True ef strengurinn path er mappa, annars False
      * os.listdir(path) – skilar lista af strengjum með nöfnum á öllu sem er í möppunni skilgreindri af path.
      * os.path.join(path, filename) – setur saman path streng við filename, notar / fyrir mac, skilar strengnum sem er full path to the file.

4.2 - analyzing recursive algorithms

* Tökum tímann sem eitt fall tekur í einu og leggjum svo saman.
* Factorial functions – hvert skipti er O(1) og það eru n+1 skipti þannig að tímaflækjan er O(n).
* English ruler – í dæmi - For c ≥ 0, a call to draw interval(c) results in precisely 2^c − 1 lines of output.
* Binary search – O(log(n)) fyrir sorted sequence með n elementum.
* Disk space usage – n er fjöldi skjala og mappa, ferð bara einu sinni inn í hvert skjal/möppu. Tíminn er O(n) því hvert recursive call er O(1). Ef maður tekur worst case væri það O(n^2) en við notum cumulative effect í stað þess að skoða worst case, það er kallað amortization.

4.3 – Recursion run amok

* Slæma við bad fibonacci er að það kallar tvisvar sinnum í hvert skipti í sjálft sig, þess vegna 2^n
* Ef recursion kallar í recursion og nær aldrei base case þá verður þetta endalaust.
* Alltaf að passa að recursion call sé að fara í átt að base case, range að minnka eða eh svoleiðis.
* Hvert recursive action býr til activation record sem er stored í minni
* Python er með limit á recursive depth sem er 1000 nested functions, hægt að breyta svona: 

4.4 Further examples of recursion

* Linear recursion – ef recursive call starts at most one other, kallar s.s. bara einu sinni í sjálft sig í hvert skipti, s.s. n-1 sinni yfir allt
  + Þetta segir ekki til um tíma heldur frekar um recursion traceið
* Binary recursion – ef recursive call may start two others
  + Kallar í sjálfan sig 2\*n-1 sinni.
* Muliple recursion – ef recursive call may start three or more others
  + 3\*n-1 eða stærra en 3

4.5 – Designing Recursive Algorithms

* Skref til að hanna:
  + Testa base cases, passa að allar mögulegar chains of recursive calls munu enda á base case
  + Recur, test sem ákveður hvaða recursive call á að framkvæma, passa að hvert recursive call sé þannig að það leiði á endanum í base case

4.6 – Eliminating tail recursion

* Tail recursion er ef recursive call sem er kallað í í einu tilviki er síðasta aðgerðin í því tilviki og skilar valueinu úr þessu recursive calli beint í recursionið sem er að enda. Þarf að vera linear recursion, ekki hægt að gera annað recursive call ef það verður að skila strax niðurstöðu þess fyrsta.
  + = A recursion is a tail recursion if any recursive call that is made from one context is the very last operation in that context, with the return value of the recursive call (if any) immediately returned by the enclosing recursion.
* Oft hægt að breyta recursive calls í for eða while loopur.

Myndband

* 3 hlutir sem þarf að hafa í huga þegar búið til recursive föll
  + Kalla í sjálft sig
  + Þurfa að fara nær base case þegar þau kalla í sjálft sig
  + Vera með base case
* Í stack er hvert skref geymt, með öll n sem koma fyrir áður en það nær base case.
* Hlutir sem maður gerir fyrir ofan endurkvæmnina gerist þegar maður fer niður í endurkvæmnina en það sem er fyrir neðan gerist þegar maður ferð til baka upp aftur.

**5.kafli – las bara** 5.2.1 (and beginning of 5.2), 5.3 and 5.4.1

5-1

* List, tuple og str – hægt að nota index, nota array til að representa röðina

5-2

* Byte er 8 bits
* Tölvan notar memory address til að vita hvaða upplýsingar eru skráðar í hvaða byte. Hvert byte er tengt við unique númer sem er addressa þessa bytes.
* Any byte af main memory er hægt að sækja með memory addressinu hans.
* Main memoryið er random access memoru (RAM), þ.e. jafn auðvelt að sækja byte 242534 og 12. Hægt að senda eða sækja byte á O(1) tíma.
* Hver stafur (unicode) í python eru 2 bytes.
* Hver sella í array þarf að nota sama fjölda bytes.
* start + cellsize page208image6448index – hægt að nota ef maður veit hvar arrayið sem maður er að skoða byrjar í memory address, sjá dæmi á bls 208 í tölvubók

5-2-1

* Listar og tuplur sem geta innihaldið mislanga strengi t.d. nota innri minni mechanisma af gerð references. Hvert element í listanum/túplunni hefur eitt memory address og vísar (references) svo í strenginn.
* Ef listi/túpla geymir immutable objects hefur það engin áhrif ef tveir listar deila elementi.
* Reference ef það eru mutable objects í listum/túplum, hefur áhrif ef þeim er breytt.

Sleppi rest af 52 í bili

5-3

* Ekki hægt að stækka arrays með því að fara í næstu memory sellur, því þar gæti verið annað data. Ekki vandamál í túplum eða strengjum því þær eru immutable.
* Hægt að bæta við elementum í lista, notað dynamix array til þess. Kerfi bætir oft við auka plássi fyrir aftan array in case, t.d. ef maður býr til 5 staka lista þá býr kerfið til 8 staka í minninu. Ef auka plássið er búið færir (býr til nýtt og eyðir gamla) arrayið sig annað þar sem er meira pláss.
* Hvernig á að fatta hversu stórt array á að búa til? Algeng regla er að búa til array sem hefur 2x stærð þess arrays sem var verið að fylla
* getsizeof úr sys module segir manni hversu mikil bytes er verið að nota til að geyma object í python
* Í tilrauninni þeirra: array getur geymt 4 references
* Þar sem að listar referencea í elementin sín þá skoðar getsizeof fallið bara stærðina fyrir grunn byggingu, s.s. ekki minnið sem er notað af elementum listans.
* Vegna keyrslutíma er sniðug að núllstilla array í upphafi og bæta inní það
* Amortized analysis – lýst með pening sem er fyrir fastan keyrslutíma. Borgar með honum fyrir operation running time. Hægt að overchargea sumar aðgerðir til að spara pening annars staðar.
* Keyrslutími fyrir að keyra frá tómum array í array með n stökum er O(n) – proposition 5.1
* Verður að passa að taka ekki of mikið pláss frá fyrir tóm stök í array ef userinn skyldi vilja bæta við
* Performing a series of *n* append operations on an initially empty dynamic array using a fixed increment with each resize takes (*n*2) time.
* Þarf að passa þegar element eru removeuð að arrayið sé ekki alltof stórt miðað við fjölda elementa. T.d. ef fjöldi actual elementa eru undir ¼ þá er hægt að eyða hluta arraysins.

5-4

5-4-1

Immutating:

A screenshot of a cell phone

Description automatically generated

Mutating:

A screenshot of a cell phone

Description automatically generated

Myndband

* Ýmsar aðgerðir eru nú þegar til í python, ætlum að prófa að útfæra þær sjálf.
* Getum gert tvær aðgerðir á listum
* Arr = [0] \* size
* Arr[i] eitthvað
* Setja fram capacity breytu sem heldur utan um hversu mikið magn af plássum eru en size sem heldur utan um hversu mörg pláss er verið að nota
* Tvöföldum fylkið sem fylltist, þ.e. tvöföldum capacity
* Til að búa til nýtt fylki, síðasta skipunin er til að setja rétta vísun á nýja fylkið
* for i in range(size):
* new\_arr[i] = arr[i]
* arr = new\_arr

**6. kafli**

6-1 - Stacks

* stack er collection af objects sem eru inserted eða removed með last-in, first-out reglu. Notandi getur alltaf insertað object í stack en getur bara fengið aðgang að og removeað nýjustu objectin.
* Instance S af týpu stack hefur þessar aðgerðir:
  + S.push(e) – addar element e efst í stack S
  + S.pop() – removear og skilar efsta elementinu í S, error ef empty
  + S.top() – skilar efsta staki
  + S.is\_empty() – True ef empty
  + Len(s) – skilar fjölda elementa í S
* Nýr stack er empty
* S er empty í byrjun á þessari mynd
* A screenshot of a cell phone

  Description automatically generated
* Búum til stack með listum í python. Má ekki nota allar aðgerðir sem við notum á lista á stack!
* Hægt að taka þetta frá listum og implementa fyrir stack:
* A screenshot of a cell phone

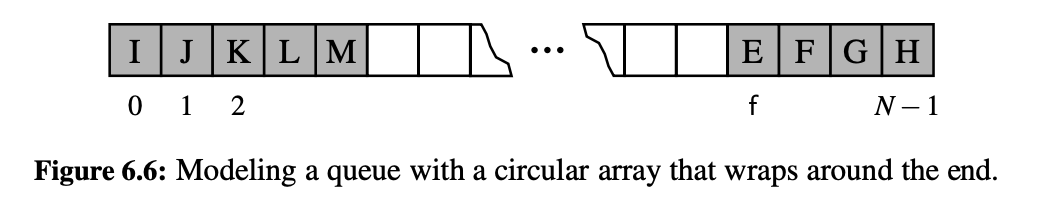
  Description automatically generated
* Erum með lista undirliggjandi í minninu.
* ArrayStack er með data breytu í init sem er listi. Aðgerðirnar eru svo gerðar fyrir stack.
* Erum með capacity breytu í stack líka, data fær þá þá lengd
* Leið til að reversea stack
* A screenshot of a cell phone

  Description automatically generated
* Til að tékka hvort svigar séu réttir, expr er t.d. [(5+x)-(y+z)]: A screenshot of a cell phone

  Description automatically generated

6-2 Queues – first in first out regla (FIFO)

* + Element geta verið insertuð hvenær sem er aftast en bara fremsta stakið getur verið tekið út
  + Element ertu insertuð að aftan en removeuð að framan.
  + Aðgerðir:
    - Q.enqueue(e) – addar elementi aftast í Q
    - Q.dequeue() – removear fremsta stakið og skilar því
    - Q.first() – skilar fyrsta stakinu
    - Q.is\_empty() – skilar True ef empty
    - Len(Q) – skilar fjölda elementa í Q
  + Ný queue er empty
  + A screenshot of a cell phone

    Description automatically generated
  + Í stað þess að nota pop(0) sem er með tímaflækju O(n) er hægt að breyta stakinu í None og færa f breytuna áfram um 1 index. Ekki gott ef það er verið að bæta við og eyða úr queue oft!
  + Undirlying listi er með fasta lengd N sem er stærri en fjöldi elementa í röðinni. Ný element eru sett í endann, fara frá byrjunninni í N-1 og svo 0, svo 1... Sjá mynd þar sem E er fyrsta element og M síðasta. 
  + F = (f+1)%N
  + Init
    - Data sem er listi
    - Size sem er fjöldi elementa í listanum
    - Front sem er indexið þar sem fyrsta elementið í röðinni er
  + Avail = ( self.front + self.size) %len(self.data) – fyrsta sætið sem er laust
  + Þegar það er resizeað þarf að passa að indexin í gamla listanum passi við nýja af því að front er þá byrjunin og á að vera í index 0 í nýja listanum.

6-3 Double ended queues - deque

* Insert og delete virkar bæði að framan og aftan.
* Aðgerðir:
* A screenshot of a cell phone

  Description automatically generated
* A screenshot of a cell phone

  Description automatically generated
* back = (self. front + self. size − 1) % len(self. data), síðasta stakið
* self. front = (self. front − 1) % len(self. data) þegar kallað í add\_first, s.s. front minnkar um 1 bil
* Það er til svona module í Python
* A screenshot of a cell phone

  Description automatically generated

Myndband

Hann skýrir enqueue - add og dequeue – remove

f er start\_index hjá honum og size er end\_index. Þegar end == start þá er size == capacity og þarf að stækka listann.

get\_size():

if end\_index <= start\_index:

return end\_index + capacity-start\_index

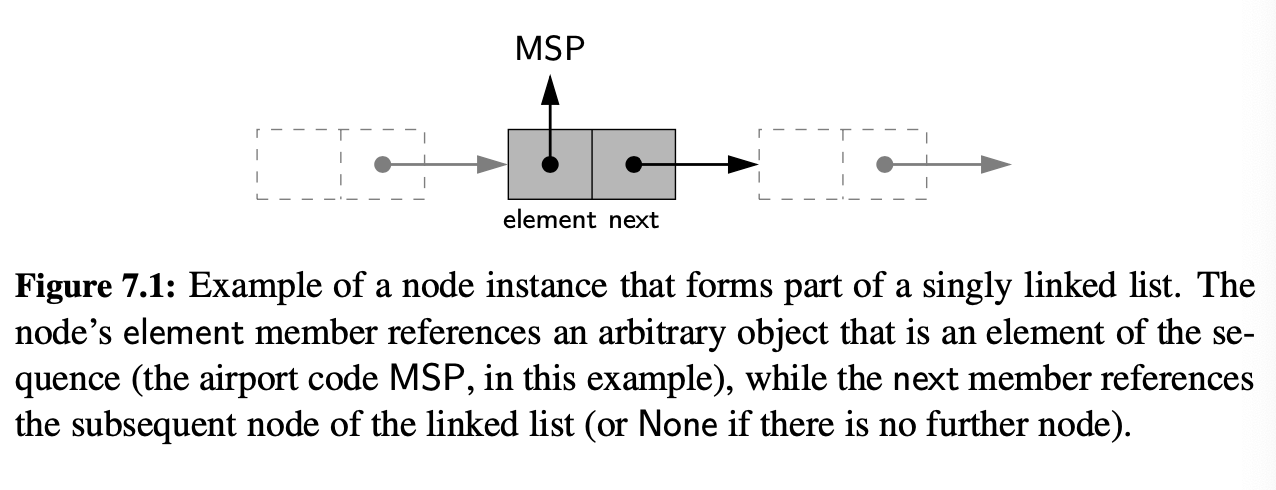
else:

return end\_index – start\_index

deque væri alltaf eins útfærsla og á queue nema fleiri aðgerðir

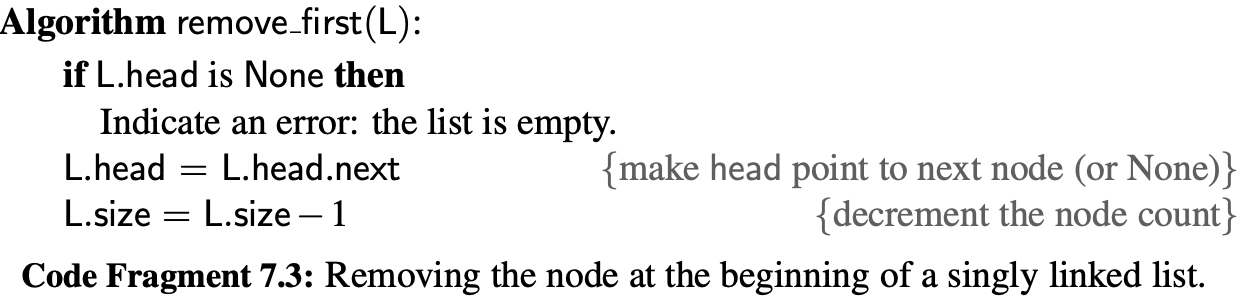
**7. kafli**

**7-1**

* Node notaðir til að vera með ref í element og eina eða fleiri ref í nágranna nodes.
* Singly linked list – safn nodes sem mynda línulega röð. Hver node er með ref í object sem er element í röðinni en líka ref í næsta node í listanum.
* 
* Flæðið:
* A picture containing object, clock

  Description automatically generated
* Fyrsti node er head og síðasti tail. Tail hefur alltaf None sem næsta reference.
* Linked list er nýtt object sem er með ref í head og oft líka í tail, oft líka með size breytu sem er þá fjöldi nodes. Einn node er s.s. tveir litlir kassar hér f. ofan.
* Bæta við breytu fremst:
  + Búa til nýjan node með elementinu og reference í gamla headið, setja svo headið á nýja nodeið. Forritunin:A screenshot of a cell phone

    Description automatically generated
* Bæta við breytu aftast:
  + Búa til nýjan node, setja elementið inní og assigna next breytuna í None. Setja next referencið í tail í nýja nodeinn og uppfæra svo tail referencina í nýja nodeinn.
  + Forritunin:
  + A screenshot of a cell phone

    Description automatically generated
* Taka element fremst úr singly linked lists
  + Öfugt við að bæta í head. Breyta head í næsta node og minnka svo stærð listans um 1.
  + Forritunin:
* Ekki hægt að taka aftast úr lista með aðferðunum sem komnar eru.
* Implementa stack með singly linked list, látum toppinn vera head því þar er hægt að bæta við og taka út element.
* Node class:A screenshot of a cell phone

  Description automatically generated
* Setja nýtt stak í head: A close up of a logo

  Description automatically generated
* Sjá kóða í bók fyrir linked stack og linked queue, O(1) tími fyrir allar operations.

7-2 Circularly linked lists

* Next referenceið í tail verður head stakið.
* Current breyta notuð til að færa sig til í listanum
* Til að setja gamla head sem nýja tail (þegar verið að snúa við röð) er gert
  + self.\_tail = self.\_tail.\_next, því next er í rauninni head
* Sjá kóða fyrir Circular Queue í bók bls 268

7-7 Link based vs array based raðir

* Kostir array-based:
  + Arrays provide O(1)-time access to an element based on an integer index.
  + Operations with equivalent asymptotic bounds typically run a constant factor more efficiently with an array-based structure versus a linked structure.
  + Array-based representations typically use proportionally less memory than linked structures.
* Kostir link-based
  + Link-based structures provide worst-case time bounds for their operations.
  + Link-based structures support O(1)-time insertions and deletions at arbi- trary positions.

Myndband

* Class Node:
  + Def \_\_init\_\_(self, data = None, next = None):
    - Self.data = data
    - Self.next = next
* Def Print\_contens(head):
  + If head != None:
    - Print(head.data)
    - Print\_contents(head.next)

Queue myndi bæta við aftast, með push back en poppa að framan með pop front. Queue þarf því bæði head og tail í init. Stack bætir alltaf við á toppinn og tekur af toppnum þannig þarf bara eina nóðu í init.

7-3 Doubly linked lists

* Node í stakið á undan (prev) og stakið sem er á eftir (next).
* Header í upphafi og trailer í endann (sentinels), eru ekki með element. Í upphafi bendir next skipunin í header á trailer og prev í trailer á header.
* Þegar elementi er eytt er gott að setja next,prev og elementið = None til að spara minni.

7-4 the positional list ADT

* Viljum ekki þurfa kalla í nodes til að inserta og deletea gögnum. Hægt þá að inserta node sem er ekki í listanum, óþarfi fyrir notandann að vita hvaða nodes eru í kring osfrv.
* p.element() skilar elementi í stöðu p.
* Aðgerðir sem er hægt að gera á lista L eftir að position hefur verið bætt við.
* A screenshot of a cell phone

  Description automatically generatedA screenshot of a cell phone

  Description automatically generated
* Prenta í positional list:
* A picture containing indoor

  Description automatically generated
* Dæmi um aðgerðir
* A screenshot of a cell phone

  Description automatically generated

Myndband

* Hver node er með next og prev breytur og data. Setja auto None á allt.
* curr=curr.prev þá breytist curr í nodeinn sem er fyrir framan
* setja inn node á undan curr:
  + Node(„E“, curr.prev,curr)
  + curr.prev.next = node
  + curr.prev = node
* fjarlægja nodeinn curr
  + curr.prev.next = curr.next
  + curr.next.prev = curr.prev
  + curr = curr.next
  + ef curr=tail þá getum við ekki tekið hann
* Hægt að vera með eina sentinal nóðu þar sem sentinal.next bendir á fremstu nóðuna og fremsta nóðan bendir á sentinal nóðuna í prev
  + Þá er kóðinn í init:
    - Self.sentinel=Node()
    - Self.sentinel.next = sentinel
    - Self.sentinel.prev = sentinel

8.kafli

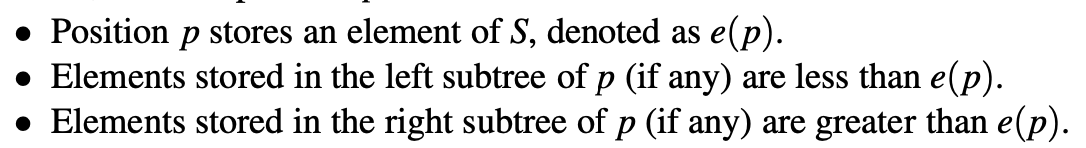
* Öll element í tjrám (nema efsta) hafa parent element og 0 eða fleiri children elements. Efsta elementið kallast root.
* T er set of nodes þar sem nóðurnar eru með parent-child relationship.
* Node er external ef hann á enginn börn, annars internal.
* Tré er ordered ef það er linear order milli barna hverrar nóðu.

8-2

* Binary tree er ordered tree með:
  + Hver node hefur mesta lagi 2 börn
  + Hvert barn er merkt sem annað hvort vinstri eða hægri
  + Vinstra barn er á undan hægra barni
* Það er proper ef hver node er með 0 eða 2 börn
* Recursive leið – annað hvort er tréð empty eða með:
  + r, kallað root sem geymir element
  + binary tree (mögulega tómt) kallað vinstra subtree af T
  + binary tree (mögulega tómt) kallað hægra subtree af T
* Binary aðgerðir:
* A picture containing indoor

  Description automatically generated
* Í binary trjám er level d með mesta lagi 2^d nóður.

8-4-1 til 8-4-4

* Preorder traversal:
  + Root of T fyrst visited og svo subtrees traversed recursively. Ef tréð er ordered þá er það gert í þeirri röð. Visit action gert fyrir recursive skrefið.
* Postorder traversal:
  + Fyrst í gegnum subtrees og svo í root. S.s. öfug röð við preorder. Þá er visit action gerður eftir recursive skrefið.
* Traversal tímaflækja er O(n) þar sem n er fjöldi positions í trénu.
* Breadth first traversal
  + Skoðað allar positions í dýpt d áður en d+1 er skoðað
* Inorder traversal af binary tré
  + Skoðað nóður T frá vinstri til hægri.
* Binary search tree
  + Röð elementa stored í binary tree.
  + Binary search tree fyrir S:
  + Keyrslutími fer eftir hæð trésins.

Myndband

* Lauf eru hnútar sem eiga engin börn
* Binary – hver hnútur á í mesta lagi tvö börn
  + Node er með data, left og right í stað data, prev og next
* Dýpt á hnút er lengd frá rótinni, rót er 0, barn rótar 1 og svo framvegis
* Hæð á tré er hæsta dýpt hnúts í trénu.
* Hægt að skipta niður í hluttré og skoða dýpt í þeim líka.
* Preorder, skrifum gögnin okkar áður en við höldum áfram í trénu, frá rót í vinstra hluttré og svo hægra hluttré
* Postorder, skrifum gögnin okkar eftir að við erum búin að skrifa út restina af trénu. Neðst í vinstra hluttrénu er skrifað fyrst út, svo hægra, svo rótin.
* Inorder, skrifar fyrst út vinstra hluttré (neðsta fyrst), svo rót og svo hægra hluttré (neðsta fyrst). Inorder virkar bara fyrir binary!!!
* Almennt tré:
  + Node er með data, children sem er listi.
  + Pre og post order virka

Uppröðun á hvernig kóðinn lítur út miðað við pre, in og post order

pre

Kall til vinstri

in

Kall til hægri

Post

**11. kafli**

* Binary search tree er tré sem inniheldur k,v par þar sem vinstra megin er minna en k og hægra megin stærra en k
* Inorder traversal í binary search tree fer frá lægsta gildi til hæsta
* Worst case running time á binary search tree er O(h) þar sem h er hæð trésins.
* Insert
  + Fyrst athuga hvort key sé nú þegar til og þá bara updatea valueið, annars setja í annað hvort vinstra eða hægra barn, eftir gildi. Alltaf í enda path.
* Worst case height er log(n) þ.a. versti keyrslutími er O(log(n)) fyrir flest sem er gert í binary leitar trjám.

Myndband – Stak fjarlægt úr tvíundarleitartré

**Insert**

\_ins(value, node):

If node == None:

return Node(value)

elif value < node.value:

node.left = \_ins(value, node.left)

elif value > node.value:

node.right = \_ins(value, node.right)

return node

Ins(value):

root = \_ins(value, root)

**Remove**

Þrír möguleikar:

* Ef nóðan er lauf skilar maður None þ.a. nodeinn fyrir ofan tengist við None.
* Ef eitt barn undir current node þá skilum við barninu
* Ef tvö börn, hægt að taka node sem er lengst til hægri í vinstra hluttré eða gildið sem er lengst til vinstri í hægra hluttré og svissa á þeim, eyða svo hnútnum lengst niðri.

\_rem(value, node):

If node == None:

return None

elif value < node.value:

node.left = \_rem(value, node.left)

elif value > node.value:

node.right = \_rem(value, node.right)

else:

return \_remove\_node(node)

return node

remove(value):

root = \_rem(value, root)

\_remove\_node(node):

If node == None:

Elif node.right == None:

Return node.left

Elif node.left == None:

Return node.right

Else: (tvö börn)

swapvalues\_leftmost(node.right)

sniðugt að hafa upphaflegu nóðuna inní leftmost til að það sé auðvelt að swappa

Fullt tré ef næst neðsta levelið er fullt og nóðurnar sem eftir sitja í neðsta levelinu eru vinstra megin í því leveli.

9.kafli

* Heap tré, hver nóða er með gildi stærra en foreldri sitt (nema rótin). Tréð verður að vera fullt, alltaf bæta í það þannig.
* Hæð trés með n element er h = log(n)
* Setjum alltaf minnsta barnið í foreldrasætið ef það er minna en foreldrið og foreldrið í barnasætið.

9.kafli myndband

* Viljum geyma lægsta gildið í rótinni, skiptir ekki máli hvernig röðin þar fyrir neðan er.
* Viljum alltaf bæta við þannig að við höldum áfram að vera með fullt tvíundartré.
  + T.d. ef ég er með lauf, þá bæti ég fyrst inn vinstra megin og svo hægra megin, svo vinstra megin í vinstra hluttréð osfrv.
  + Þarf að geta vísað í parent breytuna og athugað hvort að hún sé stærri, ef hún er stærri þá víxlað gildunum. Þarf að skoða fyrir allt tréð ef nýju gildi er bætt við.
  + Gott að vera með vísun á rót og vísun á nóðuna sem var síðast bætt við
* Til að tékka hvort barnið nóðan er:
  + If last\_node is last\_node.parent.left: - ef True þá er þetta vinstra barnið og því má bæta við hægra barni næst.
  + Ef ég er í hægra barninu þá vil ég labba upp þangað til að ég er komin í vinstra barn og fara þaðan einn upp og í hægra barn, lengst niður í vinstra subtré þess og bæti við nóðu þar
* Remove, skilum gildinu í rótinni
  + Ef last node er hægra barn, þá fært í vinstra barn
  + Ef það er vinstra barn þarf að fara upp þangað til að maður er kominn í hægra barn og þaðan einn upp, niður einn í vinstra hluttré þess og svo lengst í hægra hluttrénu. Nema maður sé kominn í rótina, þá fer maður ekki upp og einn til vinstri heldur bara beint lengst í hægra hluttré.
  + Setjum gildið af því sem við fjarlægðum í rótina og skoðum röðina á ný, eru börnin lægri en rótin? Ef svo er þá svissa á gildum.
* Ef það eru tvær nóður með sama gildi á að skila þeirri nóðu sem var sett inn á undan, þarf að búa til eitthvað skilyrði fyrir það