Informasjon

Du måtte klare 9 av 20 oppgaver for å få øvingen godkjent.

Finner du feil eller mangler eller har forslag til forbedringer, ta gjerne kontakt!

Oppgave 2

$\langle 12, 21, 20, 30, 39, 26, 7, 2 \rangle$ er riktig.

Kommentar:
Ved PUSH plasseres alltid verdien på toppen av stakken. Dermed ender man opp med endringene som er vist under.

Før PUSH(S, 2):

1	2	3	4	5	6	7	8	9	10
12	21	20	30	39	26	7			

top

Etter PUSH(S, 2):

1	2	3	4	5	6	7	8	9	10
12	21	20	30	39	26	7	2		

top

Oppgave 3

9 er riktig.

Kommentar:
For å kunne endre det tredje nederste elementet til å være 8, må vi først fjerne alle elementene fra toppen av stakken til og med det tredje nederste elementet. Dette krever 5 POP-operasjoner. Videre kan vi legge på de fire elementene $8, 41, 39, 26$ i rekkefølge, det vil si 4 PUSH-operasjoner.

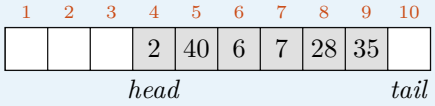
Generelt vil man alltid måtte fjerne elementene til og med det elementet lengst nede som er forskjellig, for så å bygge opp med de nye elementene. Derfor er det ingen forskjell i antall operasjoner som må utføres om man skal gå fra $\langle 12, 21, 20, 30, 39, 26, 7 \rangle$ til $\langle 12, 21, 8, 41, 39, 26 \rangle$ eller om man skal gå fra $\langle 12, 21, 20, 30, 39, 26, 7 \rangle$ til $\langle 12, 21, 7, 30, 39, 26 \rangle$.

Oppgave 4

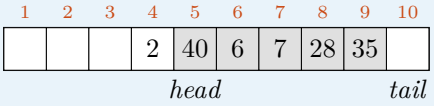
$\langle 35, 28, 7, 6, 40 \rangle$ er riktig.

Kommentar:
Når man utfører DEQUEUE blir elementet ved hodet fjernet og returnert. Dermed ender man opp med endringene som er vist under.

Før DEQUEUE(Q):



Etter DEQUEUE(Q):



Her er det også viktig å merke seg at oppgaven spesifiserer at vi ikke tenker på hvordan køen ser ut i minnet. Vi er bare ute etter innholdet i den faktiske køen.

Oppgave 5

12 er riktig.

Kommentar:
Her må vi fjerne alle elementene som finnes i køen og legge til alle de nye elementene. Det vil si 6 DEQUEUE-operasjoner og 6 DEQUEUE-operasjoner.

Generelt må vi fjerne alle elementene i køen, med unntak av de i bakerste elementene, hvor i er antallet av de bakerste elementene som opptrer i samme rekkfølge som de første elementene i den ønskede køen. Deretter må de resterende elementene legges til.

Oppgave 6

```
class Queue:

    def __init__(self, max_size):
        self.head = 0
        self.tail = 0
        # Bruker en liste til representere tabellen i bakgrunnen
        self.array = [0] * max_size
        self.max_size = max_size

    def enqueue(self, value):
        # Sett verdien og flytt halen et hakk
        self.array[self.tail] = value
        self.tail = (self.tail + 1) % self.max_size

    def dequeue(self):
        # Hent ut verdien og flytt hodet et hakk
        value = self.array[self.head]
        self.head = (self.head + 1) % self.max_size
        return value
```

Kommentar:
Koden over ligner veldig på pseudokoden i boken. Selv om både ENQUEUE og DEQUEUE har tidskompleksitet på O(1) er det fortsatt mulig å oppnå en bedre kjøretiden enn denne koden. Dette

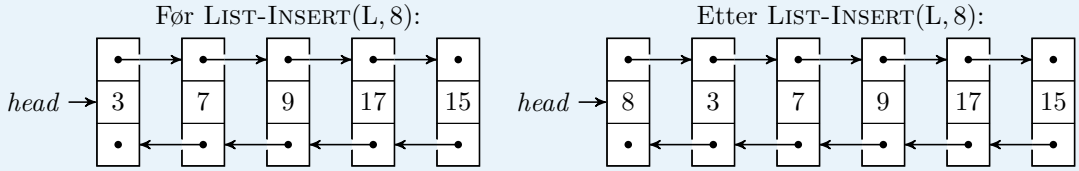
er i all hovedsak grunnet tilfeller hvor *max_size* er mye større enn antallet elementer som til en hver tid finnes i køen. Hvis man anvender dynamiske tabeller eller bruker en lenket liste i steden for, så kan man oppnå bedre resultater.

Grunnet hvordan Python fungerer er det også mulig å oppnå veldig gode resultater ved å bruke `self.array.append(value)` og `self.array.pop(0)`. Dette har egentlig ingen algoritmisk forklaring, men er mer et produkt av at innebygde funksjoner i Python som regel er veldig mye mer effektive enn kode du skriver selv (siden de innebygde funksjonene er optimalisert og kompilert).

Oppgave 7

$\langle 8, 3, 7, 9, 17, 15 \rangle$ er riktig.

Kommentar:
Når man utfører List-Insert blir den nye verdien hodet og kobles til det gamle hodet. Dermed ender man opp med endringene som er vist under.



Oppgave 8

$\langle 3, 7, 9, 17, 15 \rangle$ er riktig, siden List-Search ikke endrer på listen.

Oppgave 9

«O(n) *for en enkel-lenket liste og* O(1) *for en dobbel-lenket liste.*» er riktig.

Kommentar:
I en enkel-lenket liste må man først søke igjennom listen etter elementet som kommer før det man skal slette. Dette har en tidskompleksitet på O(n), og er nødvendig fordi man må oppdatere *next*-verdien til dette elementet. I en dobbel-lenket liste kan man se på *prev* verdien til elementet man skal slette for finne det foregående elementet, og trenger derfor ikke dette søket.

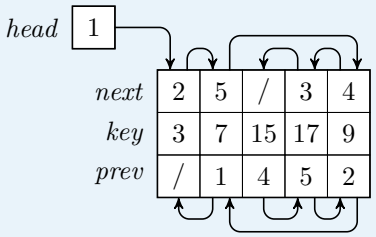
Oppgave 10

Følgende svar er riktige

- $next = \langle 2, 5, /, 3, 4 \rangle$, $key = \langle 3, 7, 15, 17, 9 \rangle$, $prev = \langle /, 1, 4, 5, 2 \rangle$, $head = 1$

- $next = \langle 0, 6, 0, 5, /, 4, 2, 0\rangle$, $key = \langle 0, 7, 0, 17, 15, 9, 3, 0\rangle$, $prev = \langle 0, 7, 0, 6, 4, 2, /, 0\rangle$, $head = 7$

Kommentar:
Denne oppgaven krever litt arbeid, selv hvis man allerede vet hvordan representasjonene fungerer. Dette skyldes at man må sjekke at alle verdiene og pekerene deres ligger på riktig plass. En grei måte å gjøre dette på er å lage figurer og sjekke at disse stemmer, slik som gjort under.



Oppgave 11

Stakk og *lenket liste* er riktig.

Kommentar:
En stakk oppfyller alle kravene vi stiller til datastrukturen, og det er som regel dette man bruker for angre-funksjonalitet. Både enkel-lenket og dobbel-lenket liste vil også fungere, da man har en referanse til hodet og kan hente ut og fjerne elementet ved hode.

En kø kan ikke brukes, siden elementene må hentes ut i samme rekkefølge som de ble lagt til.

En hashtabell støtter ikke O(1) innsetting og er derfor ikke relevant å bruke her.

Oppgave 12

Kø og *lenket liste* er riktig.

Kommentar:
En kø oppfyller alle kravene vi satte for datastrukturen, og det vil som regel være dette man anvender i denne sammenhengen. Man kan her også anvende en dobbel-lenket liste hvis man tar vare på en referanse til enden av listen. Da kan man enkelt få tak i og fjerne det siste elementet i listen. Det er dog ikke mulig å bruke en standard enkel-lenket liste her, siden fjerning av elementet i enden av listen vil kreve at man går igjennom hele listen for å finne ut hvilket element som finnes før det siste elementet.

En stakk tilfredstiller ikke kravet om å kunne hente ut elementene i samme rekkefølge som de ble satt inn.

En hashtabell støtter ikke O(1) innsetting og er derfor ikke relevant å bruke her.

Oppgave 13

«*x elementet, m nøkkelen og j hashen.*» er riktig.

Oppgave 14

«*Flere ulike faktiske nøkler gir samme hashverdi.*» er riktig.

Kommentar:
En hashfunksjon er av definisjon en avbildning (*map*) fra settet av alle nøkler til settet av alle hashverdier. Av definisjon er det dermed ikke garantert at to nøkler ikke kan ha samme hashverdi, og det er dette som kalles en kollisjon.

Oppgave 15

«*Hashfunksjonen fordeler nøklene omtrentlig uniformt over hashtabellen.*» er riktig.

Kommentar:
Definisjonen av hva en god hashfunksjon er, sier at hashfunksjonen skal fordele nøklene omtrentlig uniformt over hashverdiene (i dette tilfellet hashtabellen). Det vi ikke si at det finnes noen garanti for at x elementer med tilfeldig plukket nøkler fordeler seg uniformt over hashtabellen, men betyr heller at hvis man i snitt kan forvente at elementene fordeller seg nogenlunde uniformt over hashtabellen.

Når man bruker ordet «uniformt» i denne sammehengen har det samme betydning som ordet «jevnt».

Oppgave 16

$h(k) = 33587$, «$h(k) = \lfloor m(k\mathrm{A} \bmod 1) \rfloor$ hvor $0.5 < \mathrm{A} < 0.75$» , $h(k) = k \bmod 13$ og $h(k) = 0$ er riktig.

Kommentar:
Her er poeng å forstå at $h(k)$ er en hashfunksjon for de definisjonene hvor man for en gitt nøkkel k alltid får samme hashverdi $h(k)$. Det vil si at $h(k)$ må være en deterministisk funksjon, slik som $h(k)$ er i alle tilfellene over. I alle de andre tilfellene består $h(k)$ av en del som returnerer varierende verdier og $h(k)$ er dermed ikke-deterministisk.

Merk: hashfunksjonene som alltid returnerer samme konstante verdi ($h(k) = 0$, for eksempel) er ikke veldig gode hash funksjoner, og vil føre til kollisjon mellom alle nøklene.

Oppgave 17

Det er mange mulige svar her, men man må generelt passe på at selv om et fødselsnummer har 11 siffer, så er ikke alle nummere med 11 siffere mulige å ha som fødselsnummer. Dermed må man være påpasselig med at man ikke velger en hashfunksjon som bare ser på noen av sifferene i fødselsnummeret. For eksempel kan det være fristende å velge divisjonsmetoden (*the division method*, CLRS s.263), men med $m = 2^{16}$ gir dette oss kun de 16 siste bitsene i tallet. Denne delen av nummeret er ikke uniformt fordelt og dermed blir dette da ikke en god hashfunksjon.

Siden $m = 2^{16}$ er alternativet - med metodene i pensum - å bruke multiplikasjonsmetoden (*the multiplication method*, CLRS s.263/264). Siden vi ikke kjenner den faktiske fordelingen av fødselsnummer, er det vanskelig å oppnå noe bedre enn å velge en verdi for A som er kjent til å virke ganske bra i de fleste tilfeller. For eksempel kan vi velge verdien foreslått av Donald Knuth (se CLRS s.264) $A \approx (\sqrt{5}-1)/2$. Det er ikke garantert at dette er den beste mulige verdien for A, men vi kan ikke velge noe særlig bedre uten å vite mer om distribusjonen av fødselsnummer som er i bruk.

Merk: divisjonsmetoden kunne fungert godt hvis fødselsnummeret bare var et løpenummer fra 1 og oppover. Det kunne også vært en mulighet hvis vi kunne justert på tabellstørrelsen vår.

Oppgave 18

«*Å bruke kjøretiden for verste tilfelle kan være for pessimistisk.*» er riktig.

Kommentar:
I en del algoritmer kan en operasjon koste O(1) de $n-1$ første gangene den blir brukt, men man den n-te gangen må gjøre ekstra arbeid og derfor får en høyere kostnad. Det er kostnaden i dette ene tilfellet som blir fanget opp når man ser på tidskompleksiteten i verste tilfelle. Dette selv om man på tvers av disse n gangene totalt har en kostnad på for eksempel $2n$, noe som i snitt vil tilsi en kostnad på O(1) per kall. Da er det å se på verste tilfelle for pessimistisk i forhold til den kostnaden man faktisk vil oppleve.

Oppgave 19

O(1) er riktig.

Kommentar:
Ved innsettelse av element nummer i i hashtabellen har man en kostand på c_i

$$c_i = \begin{cases} 4i+1 & \text{hvis i kan skrives på formen } 3^k \text{for et ikke-negativt tall } k \\ 1 & \text{ellers.} \end{cases}$$

Med unntak av de situasjonene hvor man må utvide hashtabellen, så vil insetting være O(1). Ved utvidelse må man gå igjennom alle $m = 3i$ plassene i hashtabellen samt eventuelle kjeder, for å sette de gamle verdiene inn i den nye tabellen. Dette krever maksimalt $3i + i$ operasjoner, samt en operasjoner for å sette inn den nye verdien.

Det vil si at om man skal sette inn n elementer i tabellen får man at kostnaden er

$$\begin{aligned}\sum_{i=1}^{n} c_i &\leqslant n + \sum_{j=0}^{\lfloor \log_3 n \rfloor} 4 \cdot 3^j \\ &= n + 4 \sum_{j=0}^{\lfloor \log_3 n \rfloor} 3^j \\ &\leqslant n + 4 \cdot 3n \\ &= 13n\end{aligned}$$

I snitt tilsvarer dette en kostnad på 13 per innsettelse og den amortiserte kjørtiden til innsettelse i den dynamiske hashtabellen er O(1).

Merk: Man kan finne en tettere grense til $\sum_{i=1}^{n} c_i$ enn $13n$, men dette er ikke nødvendig da det ikke vil forbedre den amortiserte kjøretiden.

Oppgave 20

O(n) er riktig.

Kommentar:
Dette er et spørsmål hvor det kan være lett å tenke feil. Amortisert kjøretid handler ikke om å finne en generell gjennomsnittskostnad. Det handler kun om å finne gjennomsnittskostnaden for hver operasjon i verste tilfille. Det vil si hva er gjennomsnittskostnaden for hver operasjon (for eksempel et funksjonskall) hvis man utfører n operasjoner hvor alle resulterer i det verste tilfellet som går an (som en samlet mengde av operasjoner).

I vårt tilfelle vet vi at hver innsetting er uavhengig av andre innsettninger - utenom rekkefølge - hvor det for hver operasjon er verst hvis man må iterere over hele den lenkede listen før man kan sette inn elementet. Dette kan skje for hvert element som settes inn hvis man setter inn elementene i reversert sortert rekkefølge. Derfor har operasjon i følgende kostnad, c_i, når vi utfører n innsettelser.

$$c_i = i$$

Det vil si at ved n innsettelser har man en kostnad på

$$\begin{aligned}\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} i \\ &= \frac{n^2}{2} + \frac{n}{2}\end{aligned}$$

Det vil si at innsettelse har en kostnad på $\frac{1}{2}n + \frac{1}{2}$ i snitt over n operasjoner i verste tilfelle. Det vil si O(n).

Oppgave 21

$O(n^2)$ er riktig.

Kommentar:
Algoritmen fungerer ved at vi har en stakk med punkter og medhørende retninger som vi ønsker å utforske. For hvert punkt beveger algoritmen seg i labyrinten, ved hjelp av EXPLORE, inntil den enten finner et kryss eller en utgangen. Videre sjekker den om vi har nådd målet. Hvis vi ikke har nådd målet, så legger den til i stakken alle retninger den kan gå i krysset, uten om retningen den akkurat kom fra. Siden det ikke finnes sykler i labyrinten, så garanterer dette at vi ikke kommer tilbake til samme kryss to ganger. Dermed må algoritmen til slutt gå tom for steder og retninger den ønsker å utforske. I tillegg betyr det at algoritmen ikke kommer til å bevege seg over samme rute mer enn en gang. Altså er den totale kjøretiden av EXPLORE bundet av $O(n^2)$.

I det verste tilfellet må algoritmen utforske hele labyrinten før den enten finner utgangen eller går tom for veier å bevege seg. Det vil si at algoritmen, ved hjelp av EXPLORE, må bevege seg igjennom alle rutene i labryinten som ikke er vegger. Hvor mange vegger det er i labyrinten vil variere fra labyrint til labyrint, men i det verste tilfellet vil det være mindre enn $\frac{n^2}{2}$ vegger. Noe som vil si at EXPLORE må gå igjennom minst $\frac{n^2}{2}$ ruter, og garanterer at den totale kjøretiden for EXPLORE i det verste mulige tilfellet er $\Theta(n^2)$.

De resterende delene av algoritmen utføres maksimalt en gang per kryss. Antall kryss er bundet av $O(n^2)$ og siden hver av de resterende operasjonene, inkludert stakkoperasjonene, har kjøretid $O(1)$, vil kjøretiden til CAN-ESCAPE være $O(n^2)$ i verste tilfelle.

Oppgave 22

```
# Sortering av stakker ved hjelp av selection sort
def sort(stack1, stack2, stack3):
  # Flytt alle elementer til stack2
  while not stack1.empty():
    stack2.push(stack1.pop())

  # Bruk stakk 2 og 3 til å lese igjennom elementene
  # og finne det største gjenværende elementet
  src, dst = stack2, stack3
  # Fortsett inntil alle elementene er sortert
  while not src.empty():
    # Iterer over de gjenværende elementene i src, og finn det største
    element1 = src.pop()
    while not src.empty():
      element2 = src.pop()
      # Av de to elementene vi ser på nå, putt det minste i dst
      if element2 > element1:
        element1, element2 = element2, element1
      dst.push(element2)
    # Putt det største gjenværende elementet i stack1
    stack1.push(element1)

    # Bytt om src og dst før neste iterasjon
```

```
    src, dst = dst, src
```

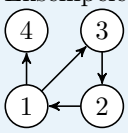
Kommentar:
Det er veldig mange mulige løsninger på denne oppgaven, da de fleste sorteringsalgoritmer kan modifiseres til å fungere på tre stakker. Koden over er en uoptimalisert implementasjon av Selection-Sort, som anvender stakk 2 og 3 til gå igjennom de gjenstående elementene, plukke ut det største elementet og plassere dette på toppen av stakk 1. Altså, koden bygger stakk 1 opp fra bunnen av, element for element. Det skal også være ganske greit å implementere Insertion-Sort.

Både Selection-Sort og Insertion-Sort vil i snitt bruke $\Theta(n^2)$ operasjoner, hvor man generelt vil se at Insertion-Sort bruker noe færre operasjoner grunnet at den har en bedre kjøretid i beste tilfelle. Generelt vil man kunne oppnå mye bedre resultater hvis man anvender sorteringsalgoritmer med kjøretid $\Theta(n \lg n)$, slik som Merge-Sort. Med 3 stakker, kan man implementere en iterativ variant av Merge-Sort med tidskompleksitet $\Theta(n \lg n)$. Merge-Sort introduseres i forelesning 3.

Oppgave 23

Den nye datastrukturen vår er egentlig en enkel uvektet rettet graf, hvor problemet er å finne kostnaden av korteste vei mellom to noder i grafen. Dette kan gjøres ved for eksempel å anvende BFS («Breadth-first search») hvor man besøker nodene (elementene) i rekkefølge basert på kostand (i vårt tilfelle antall hopp, lavest til høyest). Dette gjør man ved å starte med en kø bestående av noden man starter i. Så lenge man ikke har funnet noden man skal til og har flere noder igjen i køen, så plukker man den første i køen, og finner alle noder man kan hoppe til fra denne. De ubesøkte nodene man kan hoppei direkte til marker man som besøkt og legger dem til i køen med en kostnad (antall hopp) som er én høyere enn for den noden man hoppet fra. Hvis man ikke har funnet noden man skal til og det er tomt i køen, så er det ikke mulig å komme seg til denne noden fra startnoden.

Eksempelet i oppgaven tegnet som en graf:



Merk: BFS og lignende grafalgoritmer kommer i forelesning 8. Grafer blir introdusert en gang før dette.

Oppgave 25

For eksamensoppgavene, se løsningsforslaget til den gitte eksamenen.

For oppgaver i CLRS, så finnes det mange ressurser på nettet som har fullstendige eller nesten fullstendige løsningsforslag på alle oppgavene i boken. Det er ikke garantert at disse er 100% korrekte, men de kan gi en god indikasjon på om du har fått til oppgaven. Det er selvfølgelig også mulig å spørre studassene om hjelp med disse oppgavene.

Et eksempel på et ganske greit løsningsforslag på CLRS, laget av en tidligere doktorgradsstudent ved Rutgers, finnes her.