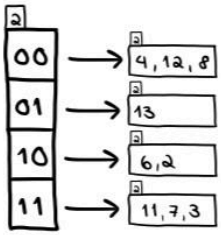
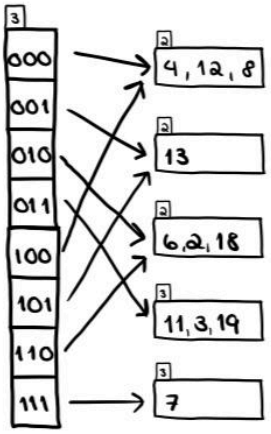
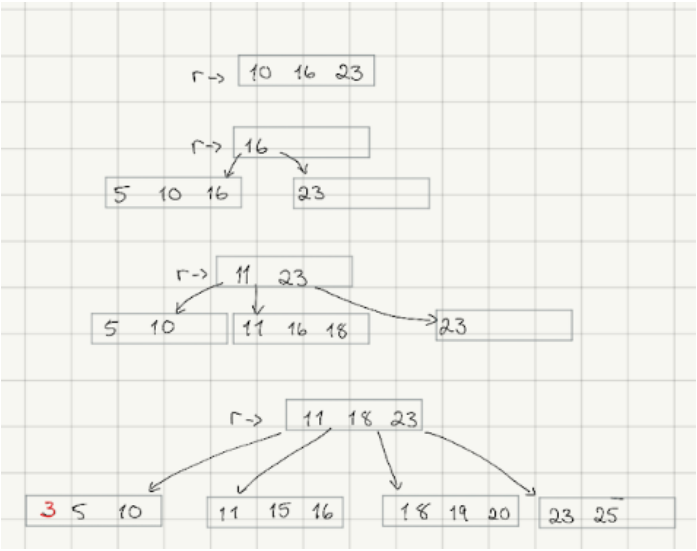
Oppgave 1: Extendible hashing

6	110
4	100
11	1011
13	1101
12	1100
7	111
8	1000
3	11
2	10
19	10011
18	10010





Oppgave 2: Konstruksjon av B+-tre



Oppgave 3: Lagring av queries med clustered B+-tre

a) Totalt antall personer: 10 000

Størrelsen på hver post: 250 byte

Størrelsen på hver blokk: 2048 byte

Antall poster per blokk: $\frac{størrelsen\ på\ hver\ blokk}{størrelsen\ på\ hver\ post} = \frac{2048\ byte}{250\ byte\ per\ post} \approx 8\ poster\ pr\ blokk$

Fyllgraden på $\frac{2}{3}$ betyr at hver blokk i gjennomsnitt er fylt til to tredjedeler av sin kapasitet. Siden hver post kan inneholde 8 poster, vil gjennomsnittlig fylling være $\frac{2}{3} \times 8 \approx 5.33$ poster per blokk.

Derav får vi at totalt antall blokker på løvnivå B er

$$\text{B} = \frac{totalt\ antall\ poster}{poster\ per\ blokk\ ved\ fyllgrad} = \frac{10\ 000}{5.33} \approx 1875$$

b) Størrelsen på PersonID: 4 byte
Størrelsen på blokkidentifikator: 4 byte
Størrelsen på hver blokk: 2048 byte
Fyllgrad: $\frac{2}{3}$

Hver indeksoppføring i en B+-tre indeksblokk inneholder en nøkkel og en peker. Altså blir størrelsen på hver indeksoppføring:

$$Str\ på\ en\ indeksoppføring = Str\ på\ nøkkel + Str\ på\ peker = 4\ byte + 4\ byte = 8\ byte$$

Antall indeksoppføringer per blokk ved full kapasitet:

$$\frac{2048\ byte\ per\ blokk}{8\ byte\ per\ indeksoppføring} = 256\ indeksoppføringer\ per\ blokk$$

Ved en fyllgrad på $\frac{2}{3}$:

$$Faktisk\ antall\ indeksoppføringer\ pr\ blokk = \frac{2}{3} \times 256 \approx 171\ indeksoppføringer$$

Vi vet at det er 1875 blokker på løvnivå og at hver blokk på nivå 1 vil peke til blokker på løvnivået. Antall blokker på nivå 1 blir da

$$Nivå\ 1 = \frac{antall\ blokker\ på\ nivå\ 0}{antall\ pekere\ per\ indeksblokk} = \frac{1875}{171} \approx 11$$

Siden vi avrunder, og for å sikre at alle blokker på nivå 0 blir dekket, runder vi opp til nærmeste hele tall, som gir 12 blokker på nivå 1.

Nivå 2 inneholder rotblokken, som peker til blokkene på nivå 1. Hvis alle blokkene på nivå 1 kan pekes til fra en enkelt blokk på nivå 2, vil vi ha kun én blokk på nivå 2.
Med 171 pekere per blokk og kun 12 blokker på nivå 1

$$Nivå\ 2 = \frac{12}{171} \approx 0.07$$

Siden vi bare trenger én blokk for å dekke 12 pekere, konkluderer vi med at det kun er 1 blokk på nivå 2, som er rotblokken.

c) SQL-setninger

1. For en søkning etter en spesifikk **PersonID**, vil B+-treet navigere fra roten ned til det relevante løvnivået. Dette betyr at for hver etasje i treet, vil en blokk bli aksessert. Må altså aksessere 3 blokker.
2. Siden denne SQL-spørringen henter alle rader fra tabellen, må alle blokker på løvnivået leses. Ingen behov for å aksessere indeksblokkene siden det er et fullstendig tabell-scan. Alle løvnivåblokk må aksesseres. Totalt må man aksessere alle blokkene på løvnivået, altså 1875 blokker.
3. Denne spørringen henter alle rader, men krever ikke ekstra sortering siden radene allerede er sortert på PersonID. Alle løvnivåblokker må aksesseres, igjen 1875 blokker.
4. B+-treet brukes til å finne den første relevant blokken der PersonID er mindre enn 100000. Deretter må spørringen fortsette sekvensielt gjennom løvnivåene til kriteriet ikke lenger er oppfylt. Anta at de første 5% av blokkene vil inneholde personer med PersonID mindre enn 100000. Det betyr

$$Totalt\ estimert\ antall\ blokker\ aksessert = 0.05 \times 1875 \approx 95$$

Oppgave 4: Queries med heap og unclustered B+-tre

1. Siden spørringen henter alle poster fra tabellen og disse er lagret i en heapfil, vil hele heapfilen måtte leses. Dette er fordi ingen spesifikk indeksering er nyttig for en fullstendig skanning av tabellen. 1250 blokker må aksesseres.
2. Uten en indeks på PersonID* må vi gjennomføre en full skanning av hele heapfilen for å finne den spesifikke personen. Dette er ineffektivt og tidkrevende da hver blokk må sjekkes for den aktuelle PersonID. Alle blokker i heapfilen må aksesseres, altså 1250 blokker.
3. Bruken av B+-treet lar oss raskt finne nødvendige blokker som peker til personpostene i heapfilen for spesifikke etternavn. Aksesserer blokker i B+-treet for å finne nødvendige pekere (i verste fall, hele treet ned til løvnivå). Hver peker fra løvnivå i treet leder til en spesifikk blokk i heapfilen hvor en personpost er lagret. Må totalt aksessere 3 blokker (ett for hvert nivå i B+-treet) pluss antall pekere funnet i løvnivåblokken som tilfredsstiller søkekriteriet.

4. For å hente alle unike etternavn, vil en effektiv måte være å skanne gjennom løvnivået av B+-treet hvor hver unik nøkkel (etternavn) er lagret. Kun løvnivået av B+-treet trenger å aksesseres, da dette inneholder alle nøklene (etternavn). Må aksessere 300 blokker (alle løvnivåblokker).
5. En ny post må først legges til i heapfilen. Dette kan kreve en søk i heapfilen for å finne et ledig sted eller legge til i slutten. Deretter må B+-treet oppdateres for å inkludere den nye posten basert på LastName. Minst 1 blokk fra heapfilen blir aksessert. Oppdatering i B+-treet vil potensielt aksessere flere blokker (fra løv til rot om nødvendig, avhengig av om nøkkelen (etternavnet) allerede finnes). Totalt må vi aksessere 1 (heap) + flere i B+-treet avhengig av trestrukturen og om det er behov for splitting av blokker.

Oppgave 5: Nested-loop-join

- Ytre tabell: Student (800 blokker)
- Indre tabell: Eksamensregistrering (12,800 blokker)

Vi har 34 blokker tilgjengelig i bufferen. Den mest effektive bruken av bufferen ville være å laste så mange blokker som mulig av den ytre tabellen inn i bufferen, for så å skanne den indre tabellen for hver av disse.

1. Blokker av ytre tabell i buffer: Anta at vi bruker 1 blokk i bufferen for output og resultatadministrasjon, og resten for blokker av ytre tabellen. Det betyr at vi kan ha opptil 34 - 1 = 33 blokker av ytre tabellen i bufferen om gangen.
2. Totalt antall iterasjoner over ytre tabellen: Antall ganger vi må laste inn blokker av ytre tabellen er gitt ved

$$\frac{800\ blokker}{33\ blokker\ per\ iterasjon} = 24.24,\ runder\ opp\ og\ f\text{å}r\ 25$$

3. Lesing av indre tabellen per iterasjon: For hver iterasjon hvor vi har 33 blokker av den ytre tabellen i bufferen, må vi skanne alle 12,800 blokker av den indre tabellen.
4. Totalt antall lesinger av den indre tabellen:

$$25\ iterasjoner \times 12\,800\ blokker\ per\ iterasjon = 320\,000\ blokklesinger$$

5. Lesinger av ytre tabellen: Den ytre tabellen må lastes inn i bufferen for hver av de 25 iterasjonene, så totalt antall lesinger av den ytre tabellen er

$$25\ iterasjoner \times 800\ blokker\ per\ iterasjon = 20\,000\ blokklesinger$$

Totalt antall blokker som leses

$$320\,000 + 20\,000 = 340\,000\ blokklesinger$$

Oppgave 6: Transaksjoner

a) Nevn to årsaker til hvorfor vi ønsker å ha transaksjoner i utgangspunktet.
To sentrale drivkrefter bak å benytte transaksjoner er at
   - Transaksjoner gir samtidighetskontroll slik at flere kan bruke databasen samtidig.
   - Loggen over transaksjoner gjør det også mulig å hente tilbake tidligere tilstander slik at databasen kan gjenskapes med forekomst av feil eller svikt.

b) Forklar kort ACID-egenskapene til transaksjoner.
   - Atomic - enten skal en transaksjon gå fullstendig gjennom, eller så skal den bli avbrutt uten at den har endret på noe. Dette gjør at en avbrutt transaksjon ikke påvirker databasen.
   - Cosisent - garanterer at en transaksjon overfører databasen fra en gyldig tilstand til en annen gyldig tilstand, uten å bryte noen integritetsregler eller begrensninger.
   - Isolation - bestemmer hvordan transaksjonsendringer er synlige for andre brukere og systemer. Isolasjonsnivåer kontrollerer hvordan og når endringene gjort i en transaksjon blir synlige for andre transaksjoner.
   - Durability - Når en transaksjon er fullført, vil endringene være permanente, selv i tilfelle av et systemkræsj.

c) H1:
   - Recoverable: Ja, fordi alle transaksjoner som leser en verdi skrevet av en annen, venter til den skrivende transaksjonen er commited.
   - ACA: Nei, fordi r2(Y) leser Y etter w1 men før c1, noe som kan føre til cascading abort hvis c1 aborter.
   - Strict: Nei, fordi r2(Y) leser før c1.

   H2:
   - Recoverable: Ja, fordi ingen transaksjon leser data skrevet av en annen før den er commited.
   - ACA: Ja, fordi ingen transaksjon leser data skrevet av en annen før den er commited.
   - Strict: Ja, fordi alle skrivinger er commited før noen leser dem.

   H3:
   - Recoverable: Ja, siden c1 og c3 skjer før c2 og ingen data leses før den er commited.
   - ACA: Ja, siden ingen data leses før den er commited.
   - Strict: Ja, siden alle skrivinger er commited før noen leser dem

d) To operasjoner er i konflikt hvis de tilhører forskjellige transaksjoner, opererer på samme datavare og minst én av operasjonene er en skriveoperasjon.

e) **r1(Y); w3(X); r3(Y); w2(X); r1(X); c1; c2; c3**

r1(Y) → w3(Y): Konflikt (r3 før w3)

w3(X) → w2(X): Konflikt

r1(X) → w2(X): Konflikt

Dersom grafen er syklusfri, er historien konfliktserialiserbar.

f) En deadlock oppstår når to eller flere transaksjoner venter på hverandre for å frigjøre ressurser de har låst. Ingen av transaksjonene kan fortsette før den andre gir slipp på den nødvendige ressursen.

g) H4 med låser

rl1(Y), r1(Y), wl3(X), w3(X), ul3(X), rl3(Y), r3(Y), ul3(Y), wl2(X), w2(X), ul2(X), rl1(X), r1(X), ul1(X), c1, c2, c3

Oppgave 7: Recovery etter krasj med ARIES

a)