IN2010 - innlevering 1

a)

Pseudokode for de fire metodene:

```
Procedure push_back(x)
    newNode ← en ny node med verdi x
    if tail = null then
            head, tail ← newNode
    else
            tail.next ← newNode
            newNode.prev ← tail
            tail ← newNode
    size ← size + 1
```

```
Procedure push_front(x)
    newNode ← en ny node med verdi x.
    if head = null then
            head, tail ← newNode
    else
        newNode.next ← head
        head.prev ← newNode
        head ← newNode
    size ← size + 1
```

```
Procedure push_middle(x)
    mid ← (size+1)/2
    if mid = 0 then
            push_front(x)
    else if mid = size then
            push_back(x)
    else
            newNode ← en ny node med verdi x.
            current ← head
            for i ← 0 to mid - 1 do
                current ← current.next
            newNode.next ← current.next
            newNode.prev ← current
            if current.next ≠ null then
                current.next.prev ← newNode
                current.next ← newNode
    size ← size + 1
```

```
Procedure get(i)
    current ← head
    for j ← 0 to i - 1 do
        current ← current.next
    return current.value
```

b)

`push_back(int x)`

I en dobbeltkoblet liste kan man legge til en ny node ved slutten i konstant tid O(1), siden man har en referanse til `tail` .

- **Verste-tilfelle tid:** O(1)

`push_front(int x)`

I en dobbeltkoblet liste kan man legge til en ny node foran i konstant tid O(1), siden man har en referanse til `head` .

- **Verste-tilfelle tid:** O(1)

`push_middle(int x)`

For å legge til i midten må man først finne midtpunktet i listen. Dette krever å gå gjennom halve listen, noe som tar n/2 tid, der n er antall elementer i listen. Siden man i O-notasjon kan se bort ifra konstanter får man den lineære tiden O(n). Selve innsettingen etter at man har funnet riktig sted er O(1).

Verste-tilfelle tid: O(n), fordi vi må traversere halve listen i verste fall.

`get(int i)`

For å hente verdien ved posisjon `i` , må vi traversere fra `head` til den i-te posisjonen, noe som tar O(i).

- **Verste-tilfelle tid:** O(n), der n er lengden på listen (dette skjer når `i` er nær slutten av listen).

c)

Når N er begrenset, kan det gjøre et vanskelig å oppdage at en algoritme er ineffektiv, fordi de verste tilfellene ikke blir synlige. O-notasjon beskriver hvordan kjøretiden vokser når N blir veldig stor, noe som gjør at man kan ignorerere konstanter. Når N er begrenset, kan operasjoner med O(n) eller O(n^2) virke håndterbare i praksis, men de kan bli betydelig tregere når N øker. Det er viktig å fjerne begrensningen på N for å forstå algoritmens reelle skalerbarhet og ytelse, spesielt i scenarier hvor N kan være vilkårlig stor. Dette gir en mer realistisk analyse av algoritmens effektivitet.

d)

Når binærsøk brukes på en array har den en tidskompleksitet på O(log n), men når den impelementeres på en lenket liste blir kompleksiteten mye dårligere. Enhvert oppslag i en lenket liste må gjøres med `get`-metoden, der man traverserer hele listen frem til node i, elementet man trenger. Hvert oppslag tar derfor det lineære O(i) tid. Den totale kompleksiteten for binærsøk i en lenket liste blir derfor (O(n) log(n)). Dette er mye dårilgere enn den originale logaritmiske tidskompleksiteten til binærsøk på array. Siden litt av grunnen til at binærsøk er såppass effektivt er at man ikke trenger å iterere gjennom hvert element arrayen for å finne elementet man er på jakt etter, fungerer det litt mot sin hensikt å bruke denne algoritmen på en lenket liste der man alltid må traversere gjennom for å finne elementer.