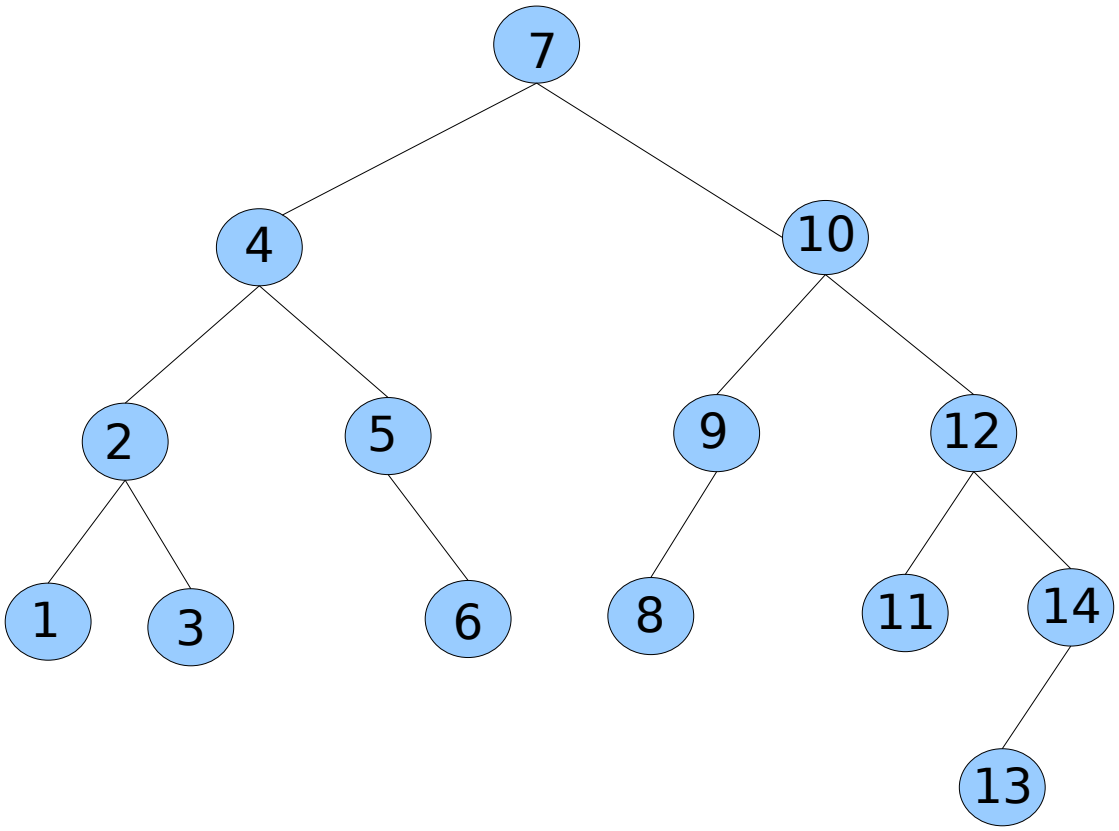
Løsningsforslag

Oppgave 1.1



Oppgave 1.2

1-2-3-4-5-6-7-8-9-10-11-12-13-14

Oppgave 1.3

Rekursiv løsning:

```
public Node settInn(Person ny, Node rot)
{
	if (rot == null)
		return new Node(ny, null, null);

	int verdi = ny.compareTo(rot.data);

	if (verdi < 0)
		rot.venstre = settInn(ny, rot.venstre);
	else if (verdi > 0)
		rot.hoyre = settInn(ny, rot.hoyre);

	return rot;
}
```

Oppgave 1.4

Iterativ løsning:

```
public Node finnPerson(int id, Node rot)
{
	while(rot!= null)
	{
		if (id< rot.data.hentId())
			rot = rot.venstre;
		else if (id > rot.data.hentId())
			rot = rot.hoyre;
		else
			return rot;
	}
	return null;
}
```

Oppgave 1.5

Skriver ut personer med inntekt over 1 mill i inorden rekkefølge:

```
public void skrivMillionInntekter(Node rot)
{
      if (rot != null)
      {
            skrivMillionInntekter(rot.venstre);

            if(rot.data.inntekt > 1000000)
                  System.out.println(rot.data.hentInntekt());

            skrivMillionInntekter(rot.hoyre);
      }
}
```

Oppgave 1.6

SettInn: O(log N), kan bli O(N)

finnPerson: O(log N), kan bli O(N)

skrivMillionInntekter: O(N)

Oppgave 2.1

Trinn	*Besøker node*	*Etterfølgere*	*Prioritetskø (Node, avstand, forgjenger)*
0	-	-	(Sarpsborg, 0, null)
1	Sarpsborg	Sandbakken Fredrikstad	(Sandbakken, 6.5, Sarpsborg) (Fredrikstad, 16.8, Sarpsborg)
2	Sandbakken	Rahaugen Stasjonsbyen	(Stasjonsbyen, 10.4, Sandbakken) (Fredrikstad, 16.8, Sarpsborg) (Rahaugen, 22.1, Sandbakken)
3	Stasjonsbyen	Løkkeberg Fredrikstad	(Fredrikstad, 16.8, Sarpsborg) (Rahaugen, 22.1, Sandbakken) (Løkkeberg, 22.4, Stasjonsbyen)
4	Fredrikstad	-	(Rahaugen, 22.1, Sandbakken) (Løkkeberg, 22.4, Stasjonsbyen)
5	Rahaugen	Halden	(Løkkeberg, 22.4, Sandbakken) (Halden, 28.4, Rahaugen)
6	Løkkeberg	Halden	(Halden, 28.4, Rahaugen) (Halden, 29.1, Løkkeberg)
7	Halden	-	(Halden, 29.1, Løkkeberg)

Korteste vei fra Sarpsborg til Halden: Sarpsborg – Sandbakken – Rahaugen – Halden, 28.4 km

Oppgave 2.2

```
private class VeiIterator implements Iterator<Vei>
{
	int index=0;
	public boolean hasNext()
	{
		return index<veier.size();
	}

	public Vei next()
	{
		if (!hasNext())
			return null;
```

```
            return veier.get(index++);
      }

      public void remove() throws UnsupportedOperationException {
            throw new UnsupportedOperationException();
}
```

Oppgave 2.3

```
public static void naboKnutepunkt(Knutepunkt k)
{
      Iterator<Vei> veiIt = k.hentVeier();
      while(veiIt.hasNext())
      {
            Vei v = veiIt.next();
            System.out.println(v.andreEnde(k).hentNavn()+": "+v.hentNavn()+"
            "+v.hentAvstand()+" km");
      }
}
```

Oppgave 2.4

Rekursiv løsning:

```
public static void muligeDestinasjoner(Knutepunkt k)
{
     ArrayList<Knutepunkt> dest = new ArrayList<Knutepunkt>();
     muligeDestinasjoner(k,dest);
     for( Knutepunkt kd:dest)
             System.out.println(kd.hentNavn());
}
private static void muligeDestinasjoner(Knutepunkt k, ArrayList<Knutepunkt> besokt)
{
      if (besokt.contains(k))
            return;
      besokt.add(k);
      Iterator<Vei> veiIt = k.hentVeier();
      while(veiIt.hasNext())
```

```
        {
                Vei v = veiIt.next();
                muligeDestinasjoner(v.andreEnde(k),besokt);
        }
}
```

Oppgave 3.1

Vi antar at antall elementer i tabeller, lister etc. er N.

A) O(1), forutsatt fornuftig implementasjon av stakken.

B) Er innom alle elementene en gang, O(N)

C) Kan bruke en annen stakk til å snu rekkefølgen, eller bygge opp en streng etterhvert som vi tar elementer av stakken. Uansett O(N).

D) Indeksere tabell, O(1)

E) Må lete oss frem til femte siste element, O(N).

F) O(log N) forutsatt balansert søketre. Kan bli O(N) hvis treet er skjevt.

G) Trenger kun å manipulere referanser. O(1)

H) Å hente ut elementet gjøres i en operasjon, men hvis plassen skal fylles igjen avhenger arbeidsmengden av om rekkefølgen til elementene er av betydning eller ikke. Hvis rekkefølgen skal bevares, må alle elementer flyttes ett hakk til venstre. Da blir arbeidsmengden O(N). Ellers kan siste element flyttes til første posisjon, og da blir arbeidsmengden O(1).

Oppgave 3.2

A) En prioritetskø er en kø der hvert element har en prioritet. Elementet med høyest prioritet er det som som står først for tur for å tas ut av køen, i motsetning til en vanlig kø der elementet som har stått lengst i køen er det som vil tas ut først. Brukes for eksempel i Dijkstras algoritme for å finne korteste vei i en graf. Se ellers pensumlitteraturen.

B) Vi må (minst) kunne utføre følgende to operasjoner på prioritetskøen:

1. sette inn nytt element med gitt prioritet
2. ta ut elementet med høyest prioritet

Arbeidsmengden for disse operasjonene for de ulike alternativene:

1. Lenket liste
    - Sette inn: Setter nytt element først i listen - O(1)
    - Ta ut: Må traversere hele listen for å finne elementet med høyest prioritet – O(N)
2. Sortert lenket liste:
    - Sette inn: Må finne riktig plass – O(N)
    - Ta ut: Kan alltid ta ut første element – O(1)
3. Binært søketre:
    - Både sette inn og ta ut i gjennomsnitt O(log N), kan bli O(N) hvis treet er skjevt, for eksempel ved sortert input.

C) En minimumsheap er et komplett binært tre hvor vi for hver node har at nodens verdi er mindre eller lik verdiene i nodens eventuelle barn. Brukes gjerne til å implementere prioritetskøer fordi innsetting av nytt element og uttak av minste element (høyest prioritet) utføres svært effektivt. Innsetting i gjennomsnitt O(1), verste tilfelle O(log N). Uttak av minste element O(log N).

Oppgave 3.3

God sommer!