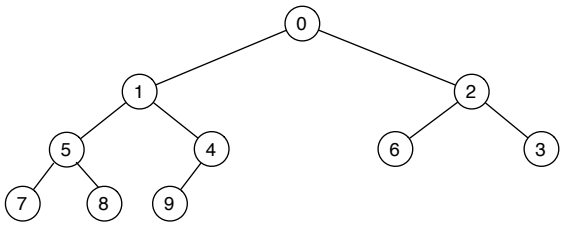
Lösningsförslag

Heapar

U 1. Heap efter insättning av element med nycklarna: 2, 5, 1, 7, 9, 6, 3, 0, 8, 4.



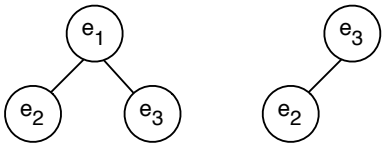
U 2. En heap kan lagras i en vektor. Roten lagras på plats 0. Barnen till noden på plats i finns på platserna 2i + 1 och 2i + 2 i vektorn. Nod på plats i har alltså sin förälder på plats (i-1)/2).

Heapen från uppgift U 1:

0	1	2	5	4	6	3	7	8	9
0	1	2	3	4	5	6	7	8	9

U 3. Tag bort noden på plats 0 i vektorn. Ersätt med den som finns på sista plats. Detta ger rätt form, men roten har nu troligtvis fel storleksförhållande till sina barn. Byt med minsta av barnen tills ordningen ok (”percolate down”).

U 4. a) Antag t ex att vi sätter in tre lika element e_1, e_2 och e_3 (i den ordningen) i en heap. Heapens utseende efter dessa insättningar visas till vänster i figuren nedan. Antag nu att vi gör en borttagning. Det blir då e_1 som tas ut. Efter borttagningen har heapen det utseende som visas till höger i figuren nedan. Nästa borttagning kommer därför att ta ut e_3. Detta element är ”yngre” än e_2. Alltså är heapen inte stabil.



b) Man kan sätta in element som består av prioritet plus ett nummer. Man numrerar sina element 1,2,... efterhand som man sätter in dem. Vidare definierar man `compareTo`-metoden så att den jämför på prioritet i första hand och nummer i andra hand. Av två element med lika prioritet kommer ett som är senare insatt då att anses vara större än det tidigare insatta. Då kommer element med lika prioritet att komma ut ur heapen i den ordning de sattes in.

U 5. I en heap kan man snabbt hitta minsta elementet. Sökning av ett godtyckligt element blir däremot dyrare. Vi kan inte söka oss ner på en gren som i ett sökträd, en heap är ju inte ordnad på samma sätt. Vi måste söka i både vänster och höger underträd tills vi eventuellt hittar elementet. I värsta fall behöver vi söka igenom hela heapen, vilket kostar $O(n)$.

U 6. Om trädet är skevt t.ex. om alla noder bara har höger barn så kommer noderna att hamna på platserna $0, 2, 6, ..., 2^i - 2, ..., 2^n - 2$ i vektorn.

I det andra fallet inträffar värsta fallet när noden på nivå $k+1$ är höger barn till noden längst till höger på nivå k. Noden på nivå 1 finns på plats 0 i vektorn. Noderna på nivå 2 finns på platserna 1 och 2, noderna på nivå 3 på platserna 3, 4, 5 och 6 etc. Noderna på nivå i finns alltså på platserna $2^{i-1}-1 \ldots 2^i-2$. Alla noder på nivåerna 1..k kommer därför att fylla platserna $1..2^k-2$. Lägger vi till ett höger barn till den sista noden hamnar det på plats $2^{k+1}-2$ d.v.s. vi behöver ungefär dubbelt så stor vektor trots att vi bara lägger till en enda nod på sista nivån.

U 7. Man kan använda listor, sorterade eller osorterade. Har man en osorterad lista blir operationerna för att söka minsta och ta bort minsta långsamma, $O(n)$, men insättning blir $O(1)$. För en sorterad lista är det tvärtom.

U 8. Man kan införa en vektor av listor. Vektorns storlek = antalet olika prioriteter. Ett elements prioritet avgör i vilken lista det placeras. Insättning blir $O(1)$. Tag bort minsta och sök minsta blir också $O(1)$. Man måste visserligen söka upp första icke-tomma listan i vektorn, men vektorns storlek är en konstant. Man åstadkommer stabilitet genom att sätta in ett nytt element sist i den lista där det hör hemma. Det kräver då att man har en listimplementation där insättning sist kostar $O(1)$.