Lösningsförslag

Tentamen - Datastrukturer, algoritmer och programkonstruktion.

DVA104

Akademin för innovation, design och teknik

Måndag 2015-06-01

Skrivtid: 14.30-19.30
Hjälpmedel: Bok relaterad till ämnet – ej digital (inga föreläsningsanteckningar eller egna anteckningar är tillåtna)
Lärare: Caroline Uppsäll, 021-101456

Betygsgränser

Betyg 3:	23p - varav minst 6 poäng är P-uppgifter
Betyg 4:	33p
Betyg 5:	40p
Max:	44p - varav 11 poäng är P-uppgifter

Allmänt

- Kod skriven i tentauppgifterna är skriven i pseudokod.
- På uppgifter där du ska skriva kod (P-uppgifter) ska koden skrivas i det språk som var aktuellt då du tog kursen (C eller Ada). Ange högst upp på bladet vilket språk koden är skriven i.
- Skriv endast en uppgift per blad
- Skriv bara på ena sidan av pappret.
- Referera inte mellan olika svar.
- Om du är osäker på vad som avses i någon fråga, skriv då vad du gör för antagande.
- Oläsliga/Oförståeliga svar rättas inte.
- Kommentera din kod!
- Eventuellt intjänade bonuspoäng kan endast användas på ordinarie tentamenstillfälle.
- Tips: Läs igenom hela tentan innan du börjar skriva för att veta hur du ska disponera din tid.

Lycka till!

Uppgift 0: *(0p)*

Läs noga igenom instruktionerna på förstasidan och följ dem!

Uppgift 1: *(7p)*

a) Beskriv kortfattat hur rekursion fungerar – glöm inte att få med de fyra viktiga parametrar som måste finnas för att en funktion ska vara rekursiv. *(2p)*
b) Vad innebär det att en sorteringsalgoritm är naturlig? *(1p)*
c) Vad innebär det att ett binärt träd är balanserat? *(1p)*
d) Nämn två olika sätt att implementera en LIFO-kö på. *(1p)*
e) Nämn minst två anledningar till varför man i en ADT ska skilja på interface, implementation och användning. *(1p)*
f) Vad innebär det att en algoritm har linjär komplexitet? *(1p)*

Lösningsförslag:

a) En rekursiv funktion är en funktion som anropar sig själv. För att den ska kunna göra det måste den ha ett rekursivt anrop. För att de rekursiva anropen ska kunna avslutas så behövs ett basfall. Om det rekursiva fallet eller basfallet ska köras beror på något villkor. För att inte den rekursiva funktionen ska fastna i sina rekursiva anrop måste det rekursiva fallet på något sätt förändra villkoret. Om aldrig basfallet nås kommer funktionen köra tills programmet krashar.
b) En sorteringsalgoritm är naturlig om den är snabbare för en redan sorterad sekvens.
c) Att djupet som mest skiljer med 1 mellan två delträd. Trädet/delträdet är balanserat om det är lika tungt till vänster som till höger i den mån det går att uppnå.
d) En LIFO-kö (stack) kan implementeras med hjälp av en array eller en länkad lista.
e) För att underlätta modifiering, återanvändning, konstruktion, testning, felsökning, läsbarhet, vidareutveckling. Genom att ”kapsla in” implementationsdetaljer kan man undvika onödiga bindningar.
f) Att effektiviteten/körtiden är direkt beroende av mängden data O(n).

Uppgift 2: *(3p)*

Vi har i kursen diskuterat arraybaserade cirkulära köer. Antag från en början tom sådan typ av kö med 7 platser. Funktionen för att lägga till data i kön heter `enqueue`, där anges det nya datat som parameter. Funktionen för att ta bort data ur kön heter `dequeue`. Till kön finns det två iteratorer (***`front`*** och ***`back`***).

Illustrera steg för steg hur den arraybaserade cirkulära kön ser ut när nedanstående anrop körs, markera tydligt vart front och back befinner sig.
Uppstår det några problem?

```
enqueue(1); enqueue(2); enqueue(3); dequeue(); enqueue(4);
dequeue(); enqueue(5); dequeue(); dequeue(); enqueue(6);
```

```
enqueue(7); enqueue(8); enqueue(9); enqueue(10); enqueue(11);
dequeue()
```

Lösningsförslag:
→ X(front, back) X X X X X X → //när front och back pekar på samma index anses kön vara tom
1(front) X(back) X X X X X
1(front) 2 X(back) X X X X
1(front) 2 3 X(back) X X X
X 2(front) 3 X(back) X X X
X 2 (front) 3 4 X(back) X X
X X 3(front) 4 X(back) X X
X X 3(front) 4 5 X(back) X
X X X X 5(front) X(back) X
X X X X 5(front) 6 X(back)
X(back) X X X 5(front) 6 7
8 X(back) X X 5(front) 6 7
8 9 X(back) X 5(front) 6 7
8 9 10 X(back) 5(front) 6 7
om 11 läggs till kommer back och front att ligga på samma index och kön anses då vara tom. Något ytterligare element kan alltså inte läggas till utan att ett annat först tas bort.

Uppgift 3: *(3p)*

Ange för nedanstående delar av olika algoritmer en uppskattning av exekveringstiden i form av Big-Oh/Ordo notation. Motivera också din uppskattning.

a)

```
for(int i=0; i<n; i++)
        x=x+1;
for(int j=1; j<n/2; j++)
        x=x+2;
```

b)

```
for(int i=n; i>=1; i=i/2)
{
        for(int j=1; j<n; j++)
                x = x+1;
}
```

c)

```
for(int i=1; i<=42; i=i*2)
{
        for(int j=0; j<i; j++)
                x = x+1;
}
```

Lösningsförslag:

a) O(n) – Linjär: Första loopen kör n ggr, andra loopen kör n/2 ggr. Looparna ligger sekventiellt alltså används addition: n + n/2 = O(n).
b) O(n log n) – den yttrle loopen halverar hela tiden mängden, den inre loopen kör n ggr varje gång den körs. Looparna är nästlade, alltså multipliceras komplexiteterna.

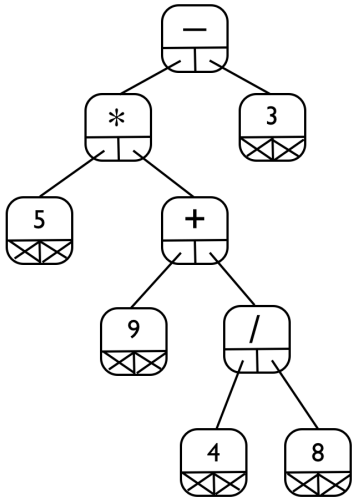
c) O(1) – Konstant: Ingen av looparna är beroende av mängden data.

Uppgift 4: *(4p)*

Vi kan använda binära träd för att representera algebraiska uttryck. Dessa träd kallas ”expressions tree”. Om man använder metoden `printPostOrder` för att skriva ut trädet får vi ett aritmetiskt utryckt i postfix notation.

Se implementation av funktionen nedan samt en grafisk representation av ett träd.

```
printPostOrder(Node subTree ) //delträd skickas in
        if subTree != NULL then //inte lika med NULL
                printPostOrder(subTree.left)
                printPostOrder(subTree.right)
                print subTree //skriv ut aktuell nod
        end if
```



a) Vilken blir uttrycket för trädet ovan då det skrivs ut med funktionen `printPostOrder`. *(1p)*

b) För att beräkna aritmetiska uttryck i postfix notation används datastukturen stack. Visa hur stacken förändras när du beräknar postfix uttrycket som du fått fram i uppgift a. Lyckas du inte lösa uppgift a väljer du istället själv ett artimetiskt uttryck i postfix och visar hur beräkningen med hjälp av stacken sker. *(3p)*

Exempel:

Vanligt räkneuttryck	Postfix
a + b	ab+
a + b * c	abc*+
(a + b) * c	ab+c*
(a – (b + c)) * (c – d)	abc+-cd-*

Lösningsförslag:

a) 5948/+*3-

b) läs uttrycker från vänster till höger, tecken för tecken. Vid siffra, pusha på stacken. Vid operand, poppa två från stacken, utför beräkningen med den aktuella operanden, pusha resultatet till stacken. När uttrycket är slut ligger resultatet av hela uttrycket högst upp (ensamt) på stacken.

Uppgift 5: *(5p)*

a) Skriv koden för en optimerad bubblesort. Algoritmen ska vara optimerad på två olika sätt jämfört med standardimplementationen. Du ska även beskriva vilka de två optimeringarna är. ---***P-uppgift*** *(4p)*

b) Hur påverkar optimeringarna komplexiteten på algoritmen? *(1p)*

Lösningsförslag:

a)

```
void swap(int *tal1, int *tal2)
{
    int temp = *tal1;
    *tal1 = *tal2;
    *tal2 = temp;
}

void bubbleSort(int arr[], int n)
{
    int i, j, swapped = 0;
    for(i = 0; i < n; i++)
    {
       swapped = 0;
       for(j = n-1; j > i; j--)
       {
           if(arr[j] < arr[j-1])
           {
               swap(&arr[j], &arr[j-1]);
               swapped = 1;
           }
       }
       if (swapped == 0)
           break;
    }
}
```

Optimering: varje varv vet vi att ett ytterligare värde (det minsta i delmängden) fått sin rätta plats. Detta/dessa behöver vi inte jämföra med nästa varv, vi kan alltså för varje varv avbryta jämförelserna ett steg tidigare.

Optimering: Har inga byten skett på ett helt varv är mängden sorterad och algoritmen kan avbrytas.

b) Komplexiteten i worst case är fortfarande oförändrad men best case (en sorterad/bakåtsorterad mängd beroende på åt vilket håll man sorterar) får O(n) istället för $O(n^2)$ i standardimplementationen.

Uppgift 6: *(6p)*

a) Beskriv sorteringsalgoritmerna mergesort och quicksort med ord. *(4p)*

b) Påverkas tidskomplexiteten för någon/några av algoritmerna i a om elementen redan är sorterade. Oavsett om du kommer fram till att svaret är Ja eller Nej ska du motivera ditt svar. *(2p)*

Lösningsförslag:

a) MergeSort: använder sig av paradigmen ”divide & conquer”, algoritmen är rekursiv. Den halverar mängden tills delmängderna är av storleken 1, då ska de sättas ihop i sorterade delmängder. Detta görs genom att delmängderna kopieras ut till temporära arrayer och sedan läggs värdena tillbaka i originalarrayen i sorterad ordning.
QuickSort: Quicksort är också rekursiv och använder sig av ”divide & conquer” tillsammans med byten. Algoritmen väljer för varje varv ett värde i mängden/delmängden till pivot-värde. Värden mindre än pivot flyttas till vänster i arrayen och värden större än pivot flyttas till höger. Därefter sorteras vänster och höger-delmängd mha quicksort.

b) MergeSort: Nej, algoritmen genomför samma arbete oberoende av hur mängden är sorterad.
QuickSort: Ja och Nej, det beror på vilket pivot-värde man använder sig av. Väljer man det första eller sista värdet som pivot kommer algoritmen att degenerera och komplexiteten blir $O(n^2)$ istället för $O(n \log n)$. Väljs t.ex. ett slumpvis pivot eller ett från mitten så är sannolikheten större att komplexiteten blir $O(n \log n)$ även för en sorterad mängd.

Uppgift 7: *(6p)*

a) Antag följande träd och nod:

```
BTree
    Node root

Node
    Integer data
    Node leftChild
    Node rightChild
```

Skriv den rekursiva funktionen som tar reda på hur djupt ett träd är. Du kan anta att trädet är sorterat och att alla länkar som inte pekar på någon nod är satta till NULL. Funktionen ska returnera det aktuella djupet på trädet (alltså djupet på det djupaste delträdet). Trädets root ligger på nivå 1. ---***P-uppgift*** *(4p)*

b) Beskriv två olika sätt att balansera ett binärt sökträd på. Det räcker inte att endast ange namnet på balanseringen utan en beskrivning av hur balanseringen går till måste ges. *(2p)*

Lösningsförslag:

a)

```
int depthCount(Node *subTree)
{
    int x=1,y=1;

    if (subTree == NULL)
        return 0;

    x += depthCount(subTree->left); //depth of left subtree
    y += depthCount(subTree->right); //depth of right subtree
    if(x >= y) //witch one is bigger
        return x;
    else
        return y;
}
```

b) 1: Skriv trädet i inorder (sorterat) till en array, töm trädet, bygg trädet rekursivt från arrayen enligt följande: Det mittersta elementet blir root i trädet/delträdet, vänster delarray bygger vänster delträd, höger delarray bygger höger delträd.
2:AVL-träd, så fort insättning och borttagning görs och ett delträd blir obalanserat (nivåerna skiljer med mer än 1) så görs en vänster- eller högerrotation på delträdet för att upprätthålla balanseringen.

Uppgift 8: *(5p)*

a) Förklara vad som är olämpligt i följande exempel. Beskriv också hur man istället bör göra. Motivera. *(2p)*

Exempel: En idrottsförening vill ha ordning på alla sina medlemmar. Idrottsföreningen väljer att lagra medlemmarna (som är av typen Person-objekt) i en hashtabell. Som hashnyckel väljer man de två första siffrorna i medlemmens personnummer.

b) Antag att man har en hashtabell där elementeten lagras i en array av storleken 11. Hash-funktionen är h(x) = x % 11. Öppen adressering (linear probing) används för att hantera kollisioner. Antag att man satt in följande element (i angiven ordning) i hashtabellen: 34, 45, 3, 67, 65, 19, 1, 17.

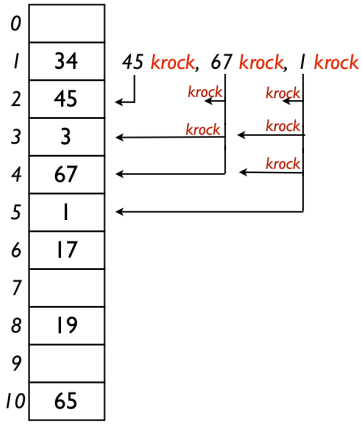
Hur ser tabellen ut efter dessa insättningar? *(2p)*

c) Om man önskar ta bort elementet 34 och bara lämnar dess plats i tabellen tom uppstår det problem. Vilket är problemet? *(1p)*

Lösningsförslag:

a) Om t.ex. personnumret lagras på formatet 19850811XXXX så kommer det lagras på hashnyckel 19. Troligtvis är många andra i föreningen också födda på 1900-talet (samma gäller med 2000-talet) och det blir därför väldigt många krockar (onödigt långa listor eller linjärt sökande). En bättre lösning är att utföra någon form av beräkning på personnumret som genererar ett index mellan 0 och M där M är det största indexet i hashtabellen.

b) 34%11 = 1
45%11 = 1
3%11 = 3
67%11= 1
65%11= 10
19%11= 8
1%11= 1
17%11= 6



c) Problemet uppstår då man ska söka i hashtabellen efter någonting med nyckel 1 som lagts till efter 34 (45, 67 eller 1 i uppgiften ovan). När man söker efter ett värde i hashtabell med öppen adressering måste man söka linjärt från den beräknade nyckeln tills man hittar det man söker eller en tom plats. Skulle 34 tas bort och platsen lämnas tom skulle det innebära att sökningar efter 45, 67 och 1 skulle resultera i att de inte hittas.

Uppgift 9: *(5p)*

a) Antag att en vän inte gått Algoritmer och Datastruktur kursen och anropar binärsökningsfunktionen med en osorterad array. Vad kommer hända? Kraschar programmet? Hur påverkar detta sökningen? Motivera med exempel. *(2p)*

b) Skriv koden för funktionen som utför binärsökning på en array av heltal.
---***P-uppgift*** *(3p)*

Lösningsförslag:

a) Det kommer bara vara tur om datat hittas i mängden. Binärsökning kräver en sorterad mängd för att kunna avgöra om sökningen ska fortsätta i vänstra eller högra halvan. Då programmet avslutas med ett resultat att ”datat inte hittats” går inte att lita på då det eftersökta datat fortfarande kan ligga i mängden. Följer algoritmen normalimplementationen av binärsökning kommer den inte att krascha.
Ex: sökning efter 2 i mängden 3 8 9 1 5 7 2 4 6.
Först kommer mittersta elementet kontrolleras (5), 2 < 5 så algoritmen söker i vänster halva (alltså 3 8 9 1) där 9 testas och sedan 3 för att efter det ”konstatera” att den eftersökta 2:an inte fanns i mängden, redan i första halveringen så har halvan där 2:an finns förkastats.

b)

```
int binarySearch(int arr[], int searchFor, int low, int high,)
{
    int mid;
    if(low <= high)
    {

        mid = (low+high)/2;
        if(arr[mid] == searchFor)
            return 1;
        else if (searchFor < arr[mid])
            return binarySearch(arr, searchFor, low, mid-1);
        else
            return binarySearch(arr, searchFor, mid+1, high);
    }
    return 0;
}
```