

Tentamen i Algoritmer & Datastrukturer i Java

Hjälpmedel: Skrivhjälpmedel, miniräknare.

Ort / Datum: Halmstad / 2011-03-15

Skrivtid: 4 timmar

Kontakt person: Mattias Wecksten 7396

Poäng / Betyg: Max poäng 40

_ >=20 poäng ger betyg 3

_ >=28 poäng ger betyg 4

_ >=37 poäng ger betyg 5

Övrigt: Det finns två olika sorters frågekategorier, de som skall besvaras genom programmerings och de som skall besvaras genom förklaringar med text och illustrationer. Programmeringslösningarna skall i största mån vara skrivna i korrekt Java-syntax.

Koden skall vara välstrukturerad och lättläst.

Koden skall innehålla lämpliga ledtexter (utskrifter) som instruerar användaren vad som krävs av honom/henne.

Om en lösning är uppenbart "klumpig" och det anses att tentanden skall känna till en smidigare lösning kan den "klumpiga" lösningen medföra poängavdrag.

Förklaringslösningar bör, om tillämpningsbart, innehålla illustrationer på relevanta datastrukturer och använda algoritmer.

Tänk på att vara noggrann och strukturerad. Det är Du som skall visa vad Du kan!

Lycka Till!

Uppgift 1 - Tidskomplexitet (4p)

a) För nedanstående delar av olika algoritmer ange en uppskattning av exekveringstiden i form av Big-Oh notation. Motivera också din uppskattning.

```
A1) public class Statistics{

public static double average( double da[ ])
{
    double n=da.length;
    return sum(da)/n;

}

public static double sum (double da[ ])
{
    double s=0;
    for(int i=0; i<da.length;i++)
        s=s+da[i];
    return s;
}
```

```
A2) while(i>=1){
    for( int j=1;j<n;j++){
        x=x+1;
    }
    i=i/2;

}
for(int i=0; i<n;i++)
    x=x+1;
```

A1. Den totala tidskomplexiteten är $O(n)$.

Motivering: Metoden `sum()` har en tidskomplexitet i $O(n)$ p.g.a. en for loop $0 \rightarrow n$ där för varje iteration exekveras operation som har konstant tidskomplexitet $O(1)$.

I metoden `average()` anropas metoden `sum()`. Själva anropet har tidskomplexiteten $O(1)$ men eftersom `sum()` har sin tidskomplexitet av $O(n)$ blir den totala tidskomplexiteten $O(n)$.

A1. Den totala tidskomplexiteten är $O(n \log n)$.

Motivering. Kodfragmentet består av en while loop som för varje iteration halverar sin storlek och som för varje iteration exekverar en for loop som loopar $0 \rightarrow n$. Detta innebär en tidskomplexitet $O(\log N * N) = O(N \log N)$

Den andra for loopen har en exekveringstid $O(n)$.

Den totala tidskomplexiteten blir då den dominanta som är $O(n \log N)$ eftersom $O(n) < O(N \log N)$.

Uppgift 2 - Datastrukturer (2p+4p+6p+4p)

a) På föreläsningen pratade vi om en sk. "wrap-around" implementation av en kö.

Förklara kort datastrukturen kö samt vad som menas med "wrap-around" implementation av kö. Vilka är fördelarna med "wrap-around" implementation?

En kö är en FIFO data struktur. First in first out som kan implementeras med statisk med array eller dynamisk med länkad lista. De vanligaste operationerna i en Kö är enqueue, dequeue som motsvarar en insert() och remove(). Wrap-around implementationen implementeras med arrayer och referenser till första respektive sista elementen i kön.

När ett enqueue exekveras incrementeras back av kö. När dequeue exekveras imcrementeras front av kö. I Wrap around lösningar incrementeras front och back till början av arrayen om kö inte är full. Detta gör att minnet används effektivt och datastrukturen är snabb.

b) Förklara principen med de datastrukturer som vi kallar "**Symbol tabeller**". Din förklaring skall innehålla bl.a. svar på följande frågor:

Vad är dessa för typ av datastrukturer? Vilka olika sätt att representera datastrukturen känner du till? Vilka operationer är vanligt förekommande? Vad kan du säga om tidskomplexiteten för operationerna. Vilka är fördelar/nackdelar med de olika sätt att representera "Symbol tabeller"?

Vi kallar symbol tabeller de datastrukturer som lagrar data i för av associerade objekt-nyckel par.

Det finns array baserade symboltabbeler som håller nycklarna i en array och objekten i en annan motsvarande array eller hashtabeller.

Länkade datastrukturer är vanligförekommande BinarySearchTree.

De vanligaste operationerana är put() get() och lookupp(). Tidskomplexiteten för operationerna är tänkta att vara $O(1)$. I visa fall kan operationerna $O(\log n)$.

c) I en multipop-stack tillåter man förutom de vanliga stackoperationerna även en som tar bort ett antal av de översta elementen (utan att returnera något). Antalet element som skall tas bort är in parameter till metoden.

Implementera en klass för en sådan stack. Stacken skall vara arraybaserd, och innehålla metoderna, **push(Object x)** **pop()**, och **multipop(int n)**

OBS! Tänk på att metoden multipop(int n) in parameter **n** kan vara större än stackens storlek. Du får hitta ett sätt att hantera det fallet. Metoden skall returnera true om multipop kan genomföras annars false.

Klassen behöver inte nödvändigtvis vara precis som den som finns i boken eller pratat om på föreläsningen vad gäller exception-hantering. Det viktiga är att alla väsentliga metoder finns med och att de är utformade så att rimliga resultat alltid ges.

```
public class ArrayStack<AnyType>
{
    private AnyType [ ] theArray;
    private int      topOfStack;
    private int size;

    /**
     * Construct the stack.
     */
    public ArrayStack( )
    {
        theArray = (AnyType [ ]) new Object[10];
        topOfStack = -1;
        size++;
    }

    /**
     * Test if the stack is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return topOfStack == -1;
    }

    /**
     * Make the stack logically empty.
     */
    public void makeEmpty( )
    {
        topOfStack = -1;
    }
}
```

```

/**
 * Insert a new item into the stack.
 * @param x the item to insert.
 */
public void push( AnyType x )
{
    if( topOfStack + 1 == theArray.length )
        doubleArray( );
    theArray[ ++topOfStack ] = x;
}

/**
 * Return and remove the most recently inserted item
 * from the stack.
 * @return the most recently inserted item in the
stack.
 * @throws Underflow if the stack is empty.
 */
public AnyType pop( )
{
    if( isEmpty( ) )
        throw new UnderflowException( "ArrayStack pop"
);
    return theArray[ topOfStack-- ];
}

/**
 * Remove the most recently inserted item from the
stack.
 * @throws UnderflowException if the stack is empty.
 */
public void multipop( int n)
{
    if( n<=size){

for( int i=0;i<n;i++)
    // pop();
    topOfStack--
}
}

```

d) Du jobbar på en implementation av en enkel *min heap* i Java. Du använder dig av en array som innehåller alla element i heapen. Här är en del av koden som du har skrivit hittills:

```
public class MINSpecialHeap
{
private final int maxSize;
private Comparable[] heap;
private int currentSize;

public MINSpecialHeap(int maxSize)
{
this.maxSize = maxSize;
heap = new Comparable[maxSize];
currentSize = 0;
}
...
}
```

Din uppgift är att **lägga till och implementera** två metoder med följande signatur:

```
// lägger till värdet x på rätt plats i heapen
public void add (Comparable x)

    int hole = ++currentSize;
    array[ 0 ] = x;

    for( ; x.compareTo( array[ hole / 2 ] ) < 0; hole /
= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;

}

// returnerar en lista som innehåller alla värden från
//heapen som är mindre än x. Du väljer själv vilken typ av
Collection datastruktur som metoden skall returnera.
public Collection getLessThan(Comparable x){
ArrayList list= new ArrayLisy();
```

```
for( ; heap[i].compareTo( x) < 0; )
list.add(heap[i]);
return
list;

}
```

Tips 1: Som du vet en heap är en "sorterad datastruktur". I detta fallet är det en MIN heap, d.v.s minsta värde finns på första platsen.

Tips 2: Välj bland klasserna som implementerade interfacen **Collection**. Vilka var dem?

Uppgift 3 - Kända algoritmer (1p+1p+1p+2p)

a) Anta att vi måste söka efter ett element bland en stor mängd data som är lagrad i en array. Vad måste vi veta om datan som skall sökas igenom för att kunna tillämpa binärt sökning?

BinärSökning kräver att datan vi söker igenom är sorterad i stigande ordning eftersom algoritmen delar arrayen i halvor och söker bara inom ena halvan beroende på om sökningselementet är mindre eller större än elementen i mitten.

b) Varför är **INTE** binärt sökning lämpligt för länkade listor?

Eftersom binärsökning måste hitta mitten av sökfältet upprepeterade gånger kräver det ett effektivt sätt att hitta mitten av sökfältet.

I länkade listor är inte indexering möjligt därför måste algoritmen loopa från head till mitt i varje iteration. Loop har en tidskomplexitet i $O(n)$ vilket försämrar exekveringstiden för binär sökning.

c) I vanligt fall är *quicksort* algoritmen en mycket effektiv sorterings algoritm. Förklara i vilka fall kan *quicksort* algoritmen ge exekveringstid i $O(N^2)$.

Quick sort algoritmen är en algoritm som använder rekursiv indelnings teknik och sedan rekonstruktion för att sortera.

Det väsentliga i en effektiv quick sort är att valet av delning elementen (pivot). I de fall pivotens värde är den minsta eller det högsta värdet i fältet som skall sorteras kommer delarna av fältet vara en liten (en eller få element och en stor nästan alla andra element). Detta innebär att indelningen i halvor i stället för att vara $\log N$ gånger blir det N gånger. Resultatet är att algoritmen blir $O(n^2)$:

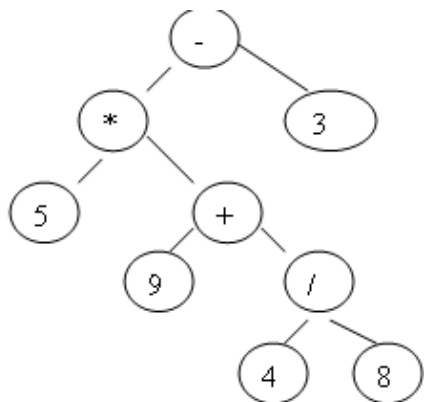
d) Förklara hur exekveringstiden påverkas om arrayen som skall sorteras är "nästan" sorterad. Vilken sorterings algoritm skulle du använda om du vet att data i arrayen är "nästan" sorterad.

Algoritmen är insertion sort. Den här algoritmen har i average time en exekveringstid i $O(n^2)$. Men, när arrayen är sorterad kommer den innersta for loopen i algoritmen aldrig att exekveras. Detta gör att algoritmen tenderar vara $O(n)$ när fältet är nästan sorterad.

Uppgift 4 - Träd och HashTabell (2p+2p+2p)

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 en aritmetiskt uttryckt i postfix notation. Se nedan implementation av metoden samt ett träd.

```
public void printPostOrder( )
{
    if( left != null )
        left.printPostOrder( );           // Left
    if( right != null )
        right.printPostOrder( );         // Right
    System.out.println( element );      // Node
}
```



a) Vilken blir uttrycket för trädet ovan.

5 9 4 8/+*3-

b) För att beräkna aritmetiska uttryck i postfix notation används datastrukturen **stack**. Visa hur stacken förändras när du beräknar *postfix uttrycket* som du fått fram i uppgift a. Lyckas du inte lösa uppgift a välj själv ett aritmetiskt uttryck och visa hur beräkningen med hjälp av stacken sker.

Push 5948

Pop 8 sen 4. Applicera / mellan dessa

Push resultat 4/8

Pop 4/8 och 9 och applicera +

Mmmm

44.5

c) Förklara vad en hash funktionen används till samt varför nedanstående hashfunktion är dålig.

```

public static int badHashFunc(String data) {
    int val = 0;
    for(int i = 0; i < data.length(); i++)
        val += data.charAt(i);
    return val;
}

```

En hash funktion omvandlar en ett objekt till ett numeriskt värde. Dessa värde kallar ofta för nyklar och används i olika typer av datastrukturer för effektiv sökning.

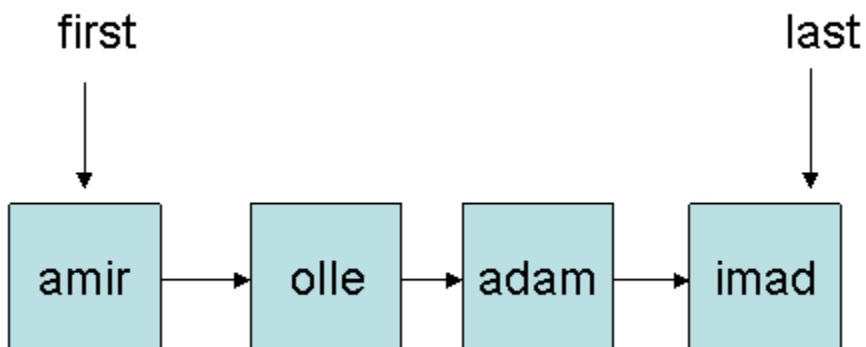
Den här hashfunktione är tänkt att omvandla en sträng till ett numerinks värde.

Anlednigen att den är dåligt är att det kommer att omvandla alla strängar som innehåller samma bokstäver till samma värde vilket försämrar hash tabellen kvalitээр.

Uppgift 5 - Länkad lista (9p)

En s.k. "double-ended linked list" man tillåter insättning och borttagning både i början och i slutet av listan.

Antag att listan är en enkellänkad och i klassen finns två referenser, en till första noden och en till sista.



Noden i listan definieras som vanligt, se nedan.

```

class Node {
    Object element; // innehållet i listnoderna
    Node next; // referens till nästa nod i listan
    public Node(Object data, Node n) // konstruerare
    {
        element=data;
        next=n
    }
    public Node(Object data)
    {
        element=data;
        next=null;
    }
}

```

Implementera klassen enligt nedanstående specifikation.

```

public class DoubleEndedLinkedList {

    /** Obs! first och last pekar på första och sista noden i
    listan. Inga noder är "tomma" */
    private Node first, last;

    public DoubleEndedLinkedList {
        first=last=null;
    }

    /** Sätt in x sist i listan */
    public void insertLast(Object x);
    Node n=new Node (x);
    if( (first==last)==null)
    first=last=n;

    last.next=n;
    last=n;

    /** Sätt in x först i listan */
    public void insertFirst(Object x);
    Node n=new Node (x);
    if( (first==last)==null)
    first=last=n;

    else
    n.next=first;
    first=n;
}

```

```
/** Tag bort första noden i listan och returnera dess
innehåll. Returnera null om listan är tom */
public Object removeFirst();

Object o= first.element;
first=first.next;
return o;

/** Tag bort sista noden i listan och returnera dess
innehåll. Returnera null om listan är tom */
public Object removeLast();

while( temp.next!=last)
temp=temp.next;

Object o= last.element;
temp.next=null;
last=temp;
return o;

/** Returnerar objektet som hittas på en viss index,
returnerar null om index är större än listan storlek
public Object getElement(int index);

}

/** Returnerar innehållet i listan, returnera null om
listan är tom */
public String toString();

}
```