Algoritmer och Datastrukturer ID1020

Data Abstraction 1,1 – 1,2

The followinng should always hold for a recursive method

- A smart compiler can reduce usage of stack space if a recursive method is tail-recursive.
- Recursive calls should always adress smaller sub-problems for the recursion to terminate.
- Recursive calls should not adress overlapping sub-problems.
- It should always have a base case – i.e. the first statement is a conditional statement which has a return.

Argument passing, values and references, the following is true

- An array or an object are always referred to by a reference.
- In C you can create a reference (i.e. a pointer) to an int but not in java
- Primitive datatypes are easily mapped to the native datatypes of the hardware and thus more efficient.
- In java arguments are passed by values
- A reference in JAVA is something which refers to where in memory something is stored.

Abstract data types

- APIs should not be changed if possible
- Abstract data types facilitates reuse of code

- Transparency means that the internal implementation of an abstract data type is externally invisible.

- Abstract data types should have a well defined API

- Abstract data types is one of the basic ideas behind object orientation

---

Documenatation and testing

- All programs/classes should have a README file which clearly describes what the program/class does, how it is implemented, how it is used, whom wrote the code and bug fixes.

- Unit testing should if possible test all possible inputs

- Verification gives stronger proof of correctness than validation

- Unit testing is a form of validation

- The main method of a class should perform extensive unit testing of the class unless it is the class driving the execution of the program

- All java classes implementing ADTs should have a main method

---

1.2

Recursion, methods that call themselves:

- Base case must exist, otherwise the stack will overflow and the method will never return

- Tail recursion, if no statements exist after the recursive call the recursion can be optimized by the compiler.

Padding makes all objects take space that is a mulitple of 8 bytes. This can waste some memory but it speeds up memory access and garbage collection.

Bags, Queues and Stacks: 1.3

Which of the following could be used to implement a collection?

- A FIFO queue
- A stack
- A LIFO queue
- A bag

Which of the following could be implemented by arrays?

- A FIFO queue
- A stack
- A LIFO queue
- A bag

Order the following ways of implementing a collection in ascending order of memory over head for storing the items in the ADT:

- Array, single linked list, double linked list

JAVA allows generic and iterable designs of ADTs. The following are true:

- Int, Double etc are necessary to have in addition to the data types int, double etc. To be able to implement generic ADTs
- An instance of an ADT can be modified (i.e. elements added/deleted/or modified) when iterated.
- A generic ADT can only be instantiated with a reference data type
- All iterable ADTs in JAVA need to implement a specific API for iteration

---

1.3

Following are the points in favour of Linked Lists

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to be shifted.

So Linked Lists provides following two advantages over arrays:

1) Dynamic size

2) Ease of insertion/deletion

Linked lists have following drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.

2) Extra memory space for a pointer is required with each element of the list.

3) Arrays have better cache locality that can make a pretty big difference in performance.

Validation = testing
Verification = proving

Generics – “something that works for different types” using type parameter instead of specific data type

---

Analysis of Algorithms: 1.4

- Tilde and Big-Oh analyses serve to determine how the worst case of an algorithm grows when the size of the problem grows
- Amortized complexity means that the cost of executing costly operations can be amortized over less costly operations
- An algorithm with amortized complexity of O(Nlog(N)) can have individual operations that grows as N*N
- The input is essential when determining the worst and best case performance of an algorithm
- When determining tilde approximations and/or the Big-Oh classification of a problem one should, in most cases, determine the “inner loop” of the algorithm

Order the complexites in ascending order:

- O(1), O(log(N)), O(N), O(Nlog(N)), O(N*N)

The sets P and NP are used to classify the tractability of problems (how easy or not a problem is to solve). What is true for these?

- One cannot find a general solution for NP-complete problems
- NP stands for Non-deterministic polynomial
- A problem can be both P and NP
- A problem belonging to P can be solved in polynomial time

1.4

Analysis of Algorithms

Big O = Ordo är att det MAX växer enligt en modell. Typ O(NlogN) innebär att den definitivt växer mindre än eller lika snabbt som NlogN. Men det skulle också kunna växa lika snabbt som N. Det relaterar till bevis. Så även om den kanske växer enligt N så kanske det bara går att bevisa för NlogN. Worst case

Big θ = Växer exakt enligt en modell. Båda stämmer. Average case

Big Ω – Beskriver att den växer minst enligt en modell. Best case

O - <=
Θ - ==
Ω - >=

Tilde approximations. We use tilde approximations, where we throw away low-order terms that complicate formulas. We write ~ *f(N)* to represent any function that when divided by *f(N)* approaches 1 as *N* grows. We write *g(N) ~ f(N)* to indicate that *g(N) / f(N)* approaches 1 as *N* grows.

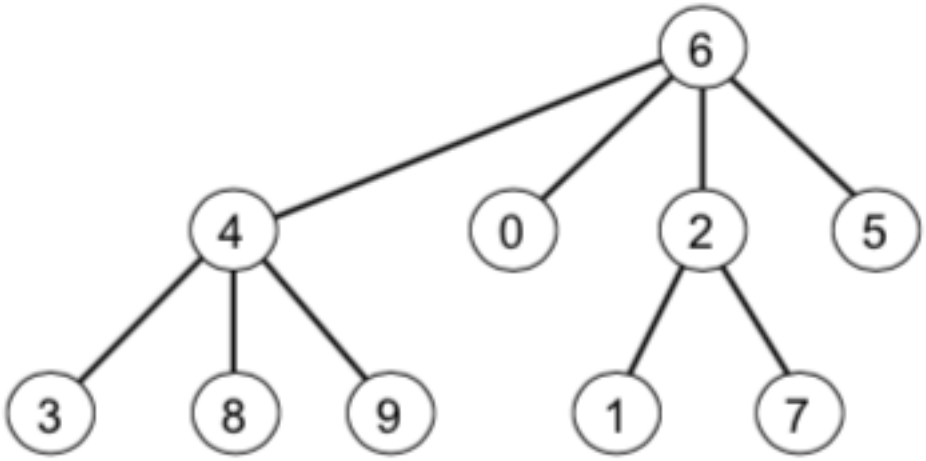
Tilde gives an approximation

function	tilde approximation	order of growth
$N^3/6 - N^2/2 + N/3$	$\sim N^3/6$	N^3
$N^2/2 - N/2$	$\sim N^2/2$	N^2
$\lg N + 1$	$\sim \lg N$	$\lg N$
3	~ 3	1

Union Find – 1.5

- A component in the “Weighted Union Find” is represented as a tree
- Copying data and/or updating data several times may affect performance adversely
- Unbalanced trees may have linear access time

1.5



	0	1	2	3	4	5	6	7	8	9
id[]	6	2	6	4	6	6	6	2	4	4

algorithm	worst-case tid
quick-find	M N
quick-union	M N
viktad QU	N + M log N
QU + stig komprimering	N + M log N
viktad QU + stig komprimering	N + M lg* N

Så länge det inte har en key fler än en gång kan det vara ett quick union träd

---

implicit tree of quick-union: can't have duplicate keys

Vid quick-find innehåller arrayen komponent-id:n, vilket gör att operationen find går snabbt men union går långsammare. Vid quick-union bildar man en trädstruktur istället och arrayen innehåller förälderns id. Vid quick-union går både union och find lika snabbt

---

Elementary sorts - 2.1

When implementing algorithms based on loops computer scientists use loop-invariants. What holds?

- A loop invariant should be true before the first execution of the loop
- Loop invariants can be used to prove correctness by induction
- Loop invariants can be used to prove the correctness of algorithms
- When the loop is terminated the loop invariant should tell us something useful that helps us understand the algorithm
- If the loop invariant is true before the execution of a loop it must also be true after the execution of the loop

---

Java allows us to implement general sorting algorithms. What is true for this?

- Callback in the comparable interface means that the compare methods is implemented as method of the object compared.
- The compareTo method must implement a total order for general sorting method to be able to sort all elements correctly.
- Transitive is defined such that if x <= y & y <= z, then x <= z
- An algorithm sorting arrays that are “Comparable” can sort any array containing the same type of elements that implement the comparable interface.
- Antisymmetric is defined such that if x <= y & y<=x, then x == y.

- An inversion are two elements in an array that are not correctly sorted. (Out of order)
- Insertion sort is more effective than most sorting algorithms when the number of inversions is low.
- A low number of inversions means that the array is nearly sorted
- The number of swaps performed by insertion sort is equal to the number of inversions in the array.
- Shellsort improve the performance of insertion sort by moving elements longer when it performs a swap.
- Selection sort and insertion sort are both $O(N^2)$

2.1

```
class SelectionSort
{
    void sort(int arr[])
    {
        int n = arr.length;

        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++)
        {
            // Find the minimum element in unsorted array
            int min_idx = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;

            // Swap the found minimum element with the first
            // element
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
}
```

```
class InsertionSort
{
    /*Function to sort array using insertion sort*/
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i=1; i<n; ++i)
        {
            int key = arr[i];
            int j = i-1;

            /* Move elements of arr[0..i-1], that are
               greater than key, to one position ahead
               of their current position */
            while (j>=0 && arr[j] > key)
            {
                arr[j+1] = arr[j];
                j = j-1;
            }
            arr[j+1] = key;
        }
    }
```

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))

Proof that Quicksort is not stable

An example is the array [B, C1, C2, A] where C appears twice (as C1 and C2). B will be chosen as the pivot and then A and C1 will swap places. [B, A, C2, C1]. In the end, we get [A, B, C2, C1]. As C1 and C2 have swapped places, the algorithm is not stable.

Equals design

If field is a primitive type, use ==
If field is an object, use equals( )

Mergesort - 2.2

- No comparison based algorithm can sort N arbitrary elements with unique keys using less than Nlog(N) comparisons.
- A stable sorting algorithm preserves the relative order of elements with equal keys.
- It is possible to implement classes that supports different method to compare objects (of the same type) in java
- Insertion and merge sort are stable

- A comparison based sorting algorithm may sort N elements using less then O(Nlog(N)) comparisons for certain inputs.

- Mergesort has optimal time complexity

- An “in-place” algorithm solves a problem using little auxililary (extra) memory. At most (~ logN).

Quicksort – 2.3

- Quicksort is efficient for unordered (randomized) input

- The first step of Quicksort is to randomize the input

- Mergesort has better worst case behaviour than Quicksort

- Quicksort is in-place

- Quicksort implementing the three way partitioning of Bentley & McIlroy is efficient for large inputs.

- Quicksort is general faster than mergesort

- On average Mergesort uses less comparisons but moves more elements than Quicksort

---

Priority Queues - 2.4

*The basic operations on a priority queues is to insert elements and retrive the elements with the highest priority.

* It is possible to implement special purpose priority queues that have O(1) access time.

*A binary heap has the property that the children of any node has lower priority than the node/parent.

*A heap makes inefficient use of the caches

*Splay tree is a fast heuristically balanced priority queue

*A high priority is in many cases associated with a low numerical value

*All operations on a heap has O(logN) time complexity

Symbol Tables & Binary Search Trees 3.1 – 3.2:

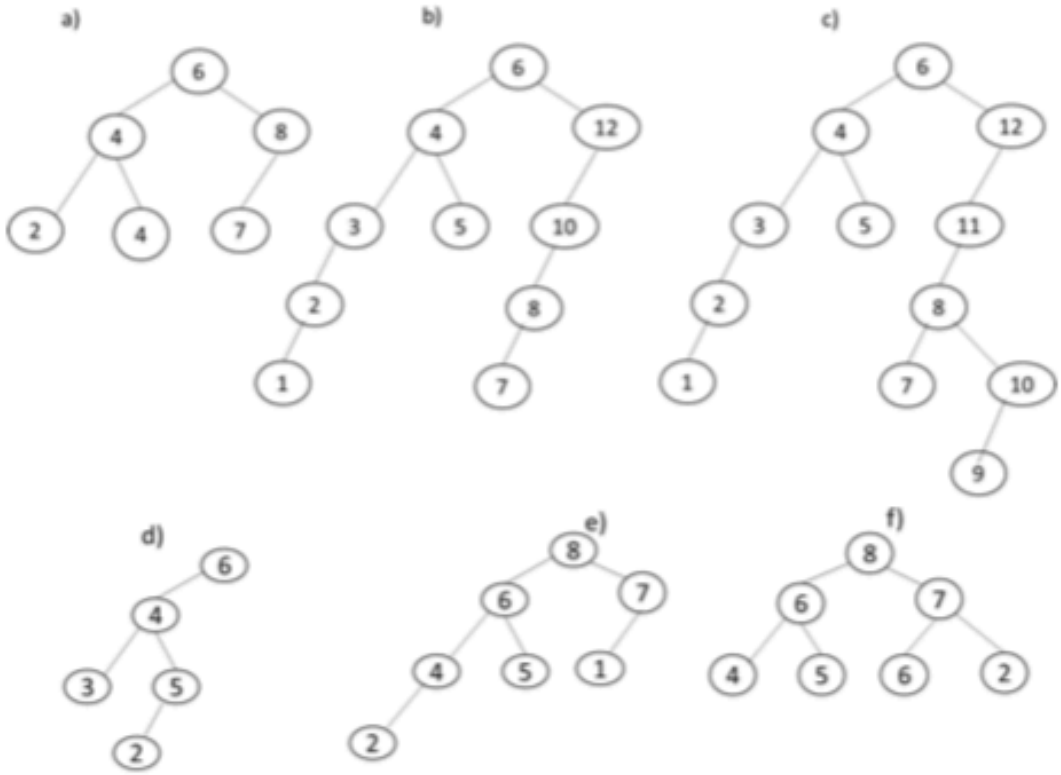
- An un-ordered linked list can be used to implement a ST
- A symbol table is used to associate keys with values
- Basic operations on STs are put( ) and get( )
- A key may be a string, numerical value or another type of data
- The reason for keeping keys sorted in an array implementation of a ST is to be able to search with a binary search
- In Java reference data types are used for keys
- Two arrays can be used to implement a ST

- In a balanced BST operations take on average O(log(N)) compares
- The worst case execution time for get( ) / put( ) operations on a BST is O(N)
- A binary search tree is a binary tree where the key of any node is larger than any of the keys in its left subtree and smaller than any key in its right subtree.

Balanced Search Trees – 3.3

- A red black tree is perfectly balanced by the number of black links from the root to any null link
- Red-black trees allows for operations on ordered STs, i.e. the tree is ordered
- Two nodes connected by a red link in a red-black tree corresponds to a 3-node in 2-3 tree
- A red-black tree is an implementation of a datastructure directly corresponding to a 2-3 tree.
- A 2-3 tree tree is unnecessary complicated to implement directly
- A 2-3 tree is a perfectly balanced search tree
- Operations on ordered STs. Includes selecting the Nth key and a range of keys between two boundaries
- A 2-3 tree has nodes with two or three children
- Red-black trees can be traversed in the same manner as a binary search tree
- A node in a 2-3 tree may contain one or two keys

3.3



Solution

a) Binary Heap: only f (not option e because in a binary heap, node 1 would get a right child before node 2 is added on the far left).

b) Binary search tree: b and c

c) Red-black Binary search tree: only b

d) Quick union: b,c, d, and e

Hash tables – 3.4

- More than one key may be mapped to the same hash value (i.e. to the same index in the array implementing the hashtable)
- Hashcode and hash methods should be consistent – equal keys should produce the same value
- Hashcode and hashmethods should uniformly distrubute the set of keys

- Hashtables allows for O(1) amortized access time for insertions and lookups for many applications

- Hashcodes and hash methods should be efficient to compute

- Hashcodes and hash methods should in general, whenever possible, use as many bits of the key and give equal weight (importance) to each bit

- A hashcode maps a large range of possible key values to a smaller range of (integer) values

- Collision resolution with separate chaining means that each element in the hashtable array holds <key,value> pairs in a short linked list

- Collision resolution with linear probing means that when an index in the hashtable array is already occupied the <key,value> pair is inserted at the next free index modulus the array size

---

3.4

The optimal size of a the hash table array with linear probing is between 0,125 and 0.5 full

---

- Standard arrays in hash tables are kept between 0.5 and 0.125 full

- Hash function hashes to multiples of 32, standard resizing, linear probing, array access for hit and miss: array is between kept between 4 and 16, average is (4+16)/2 = 10. The array will have clusters of size 10. Linear probing will on average use 5.5 access for a hit and 11 for a miss.

- Same hash function as above but uses separate chaining, array access for hit and miss: the number of array access for hit and miss will be one. The list will have to be traversed but that does not involve array accesses.

- Hash tables vs. red-black BST for symbol table, pros and cons: red-black BSTs are useful if you need to iterate the elements and they have a better worse case (logarithmic) which may be necessary for mission-critical applications or open systems where malicious users supply keys that have been designed for maximum collision. Hash tables have linear worst case.

Hash tables are generally better, since time complexity is most likely constant rather than logarithmic.

Undirected Graphs – 4.1

- A bipartite graph can be divided into two disjoint sets of nodes such that each edge connects a vertex in one set in the other set
- Breadth First Search finds the shortest paths from a starting vertex to all reachable vertexes
- Depth First Search can be used to efficiently find if a graph has (at least) one cycle
- A Hamilton cycle is a cycle which visits each vertex in a graph exactly once
- Sparse graphs can be efficiently implemented by an array of adjacency lists
- An Eulerian cycle is a cycle which visit each edge in a graph exactly once
- A spanning tree is an acyclic fully connected graph
- Many real world applications are based on sparse graphs
- A connected graph in which all nodes have an even degree has an Eulerian cycle

---

4.1

DFS – steps

1, Pick a starting A node
2, push on stack
3, visit it and mark as visited
4, Look at the adjacent nodes from A
5, Visit node with first alphabetical order which is B
6, Push B on stack

7, Visit B and mark as visited
8, For the node on the top of the stack, B.
9, Visit the alphabetically first node from B
10, In this case E.
11, Push on stack
12, visit it
13, when all nodes are visited from the current one
14, pop node off stack until a node with unmarked neighbours appears
15, Repeat until only A is left on stack.
16, pop A
17, All node traversed

BFS – steps

1, Start at A
2, mark A as visited
3, Go to the alphabetically first node B
4, enqueue B
5, continue from A and visit next node C from A
6, Enqueue C
7, When all nodes reachable from A are visited dequeue (B)
8, Reapeat step 2-7. i.e. start at B
9, When Queue is empty, graph is traversed

Directed Graphs – 4.2

- Digraphs (implicit) are used for mark and sweep garbage collection
- DAG is a directed acyclic graph
- A DAG can be used to construct a working precedence constrained scheduling
- A topological sort can only be found for acyclic directed graphs
- A strongly connected component has at least one directed cycle

- The transtive closure of a digraph G is another digraph with the same set of vertices but with an edge from v to w iff w is reachable from v in G

Kruskals algoritm

• Kruskals algoritm är en girig algoritm för att skapa ett minimalt uppspännande träd från en godtycklig sammanhängande, viktad och oriktad graf.

• Algoritmen bygger en skog av träd som allt eftersom växer ihop.

Algoritmen är girig, eftersom den hela tiden lägger till den kortaste kanten den kan hitta till sina delträd.

Prims algoritm

Är en girig algoritm för att skapa ett minimalt uppspännande träd från en godtycklig sammanhängande, viktad och oriktad graf. Algoritmen finner i varje iteration den kant med lägst vikt som kan förbinda trädet med ett hörn som ännu inte finns med i trädet, varpå trädet utökas med denna kant (och det hörn som den ansluter till). Iterationen fortsätter så länge det finns hörn som inte lagts till i trädet.

Dijkstras algoritm

Är en matematisk algoritm för att hitta den billigaste vägen från en given nod till alla andra noder i en viktad och riktad graf med positiva bågkostnader.

A) Recursive methods should always have a base case

D) Object oriented languages are well-suited to implement ADTs

E) A stack is a special form of a queue

G) The Tilde approximation for complexity approximates the time complexity by the fastest growing term including constants

H) One cannot find a general, optimal, algorithm to solve a NP-complete problem

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	θ(1)	θ(n)	θ(n)	θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	θ(1)	θ(1)	θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	θ(log(n))	θ(log(n))	θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	θ(log(n))	θ(log(n))	θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Hit/miss för linear probing

Standard resizing innebär att arrayen dubbleras i storlek när N/2 platser fyllts och halveras i storlek när N/8 platser fyllts.

Linear probing innebär att om två element har samma hashvärde så kommer det elementet som läggs in efter det andra att hamna vid nästa lediga plats i arrayen. Så om X och Y (två godtyckliga element) båda hashas till 5, X läggs in först och hamnar vid index 5. Y läggs in sen men 5 är upptaget och hamnar då på 6 om det inte är upptaget där också. När en sökning görs kommer hashvärdet för både X och Y fortfarande vara 5. X hittas vid 5 (1 access) och Y hittas vid 6 (2 accesser, 1 som kollar om värdet vid index 5 är rätt och 1 som kollar om värdet vid index 6 är rätt). Detta gäller vidare för alla krockar av hashningar.

Om alla element perfekt uniformt hashas till multiplar av 128 innebär det några saker. Perfekt uniformt betyder i hashtabeller jämnt fördelat. Multipler av 128 innebär att alla element kommer få hashvärde 0, 128, 256 osv. Eftersom elementen blir jämnt fördelade kommer som mest 128 element kunna hashas till resp. index 0, 128, 256, osv. Alltså kommer som mest 128 element hamna på index 0-127, 128-255 osv. Det kommer alltså aldrig ske någon krock mellan element som har olika hashvärden, bara mellan de som hashas till samma värde. T.ex. om det istället skulle kunna hashas 129 element till index 0 så hade det sista elementet hamnat vid index 128 och krocka med element som hashas till 128.

Dock gäller även i det här fallet, pga standard resizing, att som max N/2 element kommer finnas i arrayen. Eftersom de är jämnt fördelade kommer som max 128/2 element hashas till samma index och således hamna i samma spann (0-127, 128-255 osv).

Själva lösningen:
När arrayen då är fylld med N/2 element kommer alltså det finnas som mest 64 element i vardera spann (varje spann är fyllt till hälften) och eftersom alla blir hashade till multiplar av 128 kommer de alla att ligga i första halvorna av spannen (0-63, 128-191 osv).

Om en sökning görs efter ett element som inte finns i hashtabellen kommer hela klustret på 64 element att letas igenom, indexen efter klustret kollas också och upptäcks vara tom, då är resultatet en sökmiss med 64+1 accesser.

En sökning efter ett element som finns i hashtabellen kommer kräva allt mellan 1-64 accesser. Givet ett slumpmässigt element som söks av dessa 64 kommer det förväntat att hittas efter 32 accesser, eftersom det är lika stor sannolikhet att det hittas efter vilket som helst av allt mellan 1 och 64 accesser. Resultatet blir alltså en sökträff efter 32 accesser.

För N/8 innebär det att arrayen kommer som mest ha kluster av storlek 128/8=16 vid varje spann. Samma princip gäller som med N/2. Sökmiss efter att ha letat igenom alla + (1 tom plats) och inte hittat elementet. Dvs 16+1=17 accesser vid sökmiss. Vid

sökträff blir det mellan 1-16 accesser, förväntade blir då 8 accesser.

För att få snittet givet att vi inte vet hur fylld arrayen är så tar vi snittet mellan mest fylld (N/2) och minst fylld (N/8).
Träff: (32+8)/2 = 20
Miss: (64+1+16+1)/2=41

---

3. Ange tidskomplexiteten i värsta fall (worst-case) för följande javakodstycken. Skriv dina svar i tilde-notation.

(a)

```
for (int i = 0; i < n * n; i++)
    sum++;
```

(b)

```
for (int i = n; i > 0; i=i/2)
    sum++;
```

(c)

```
for (int i = 0; i < n * n; i++)
  for (int j = i; j < n; j++)
    sum++;
```

(d)

```
for (int i = 0; i < n * n; i++)
  for (int j = 1; j < n; j = j*2)
sum++;
```

Lösningen (2,5 pts each)

Notera att tilde-notation är efterfrågat.

(a) O(N^2)

(b) O(log(N))

(c) O(N^3).

(d) O(log(N^2)log(N)) (Notera att j=1.)

---

4. Skriv (i Java eller pseudokod) en algoritm för att hitta en element i en sorterad array med unika element i logaritmisk tidskomplexitet (i värsta fall).

Lösningen (10 pts)

```
public static int binarySearch(int[] a, int key)   {
     int lo = 0, hi = a.length-1;
     while (lo <= hi) {
         int mid = lo + (hi - lo) / 2;
         if (key < a[mid]) {
             hi = mid - 1;
         }
         else if (key > a[mid]) {
             lo = mid + 1;
         }
```



```
         else {
             return mid;
         }
     }
    return -1;
 }
```